UCLA Computer Science 131 (Spring 2005)
Section 1 final
180 points, 180 minutes total
open book, open notes

Name:_____

Student ID:_____

1	2	3	4	5	6	
7	8	9	10	11	12	total

1 (5 minutes). Why is heap fragmentation not a problem in a traditional Lisp implementation where "cons" is the only heap storage allocation primitive?

Because each cons cell is the same size, there will be no internal or external fragmentation. There is no internal fragmentation because the cons cell is completely filled by the head and tail. There is no external fragmentation because there will not be any block too big to fit in any available memory space.

2. Consider the following OCaml program. Its names have been changed to protect the innocent.

```
let s x = x + 1

let rec i min max =
  if max < min then [] else min :: i (s min) max

let rec f p = function
 | []  -> []
 | a::r ->
   let fpr = f p r in if p a then a::fpr else fpr

let x n =
  f (fun m -> not (m mod n = 0))
```



```
let t max =
  let rec f1 = function
    | [] -> []
    | n::r ->
       n :: if max < n*n then r else f1 (x n r)
  in
    f1 (i 2 max)
```

2a (10 minutes). What are the types of the top-level OCaml identifiers s, i, f, x, t?

```
s : int -> int
i : int -> int -> int list
f : ('a -> bool) -> 'a list -> 'a lists
x : int -> int list -> int list
t : int -> int list
```

2b (10 minutes). What does the expression "t 20" evaluate to? Show your work.

What do the functions do?

s increments its argument.

i creates a list of integers from min to max (inclusive on both ends, because when min=max, the else branch is taken, adding min(=max) to the list)

f acts as a filter. It accepts a predicate and a list, and iterates through the elements of the list. When an element passes "p a", it is added to the result list, otherwise it is dropped.

x creates a curried function, by partially evaluating f with a predicate which returns true if some element of the list is not divisible by the parameter, n. So (x n l) would return a list which is l with all elements divisible by n removed.

t is the Sieve of Eratosthenes. It accepts an integer max, creates a list from 2 to max, and applies it to f1. f1 puts the first prime number, 2, on the result list, and filters out all elements in the rest of the list divisible by 2. It does this on each successive prime remaining in the list until max < n*n, which results in a list of primes from 2 to max (inclusive).

t 20 = [2, 3, 5, 7, 11, 13, 17, 19]

3 (15 minutes). The paper "The Twisted Network Framework" gives the following as one of the disadvantages of Twisted.

> As each callback must be finished as soon as possible, it is not possible to keep persistent state in function-local variables.

What does this remark mean? Give an example of

why it might be nice to keep persistent state in a function-local variable, and describe what would happen to a Twisted server if you tried to do that.

This remark is saying that you cannot keep persistent state in a function-local variable in Twisted the same way you can in a multithreaded app, because functions must finish quickly so as to not block the single thread.

A desirable use case for a function-local variable may be a file descriptor or a database handle which you are using while running a long-running query. If you did this with a Twisted server, it would hang while the query ran.

4 (15 minutes). Consider the following argument:

> The programmers of the world are still using the first major imperative language (Fortran), the first major functional language (Lisp), and the first major logic-programming language (Prolog), but they no longer use the first major object-oriented language (Simula 67). That is because object-oriented languages are fundamentally different. Here's why:

If you agree with this argument, finish it by adding some justifications and examples. If not, argue the contrary.

(Note: Prof. Eggert did not cover Simula67, so you would not be responsible for that detail on the exam.)

You can argue either way. Some might say the OOP crowd is more trendy. A better reason is that OOP languages like C++ and Java have much more dependence on external libraries, due to the fact that OOP concepts like classes, subclassing, etc, make it very easy to create and maintain external modules. But as time passes, people will begin to write their modules in more modern OOP languages, which causes earlier OOP languages to be unusable due to lack of library support.

5. Consider the static-scoping semantics given in pages 512-513 of Webber. (508-509 in the 2nd edition)

5a (10 minutes). Can these semantics be implemented easily on a machine in which all storage is allocated on the stack, or do they assume the existence of a heap? If the former, give a nontrivial example and show why the heap is not needed for it; if the latter, give an example of why a heap is necessary.

Take a look at the rule (fn(x,E),C) -> (x,E,C)

The static scoping semantics carry the context of the caller into the function body, i.e. all its variable bindings. This assumes existence of a heap, since it would not be easy to store a caller's variables on the new stack frame of a called function.

5b (10 minutes). This question is the same as (a), except use the dynamic-scoping semantics.

Take a look at the rule (fn(x,E),C) -> (x,E)

With dynamic scoping, the callee does not carry the caller's context into the function body, so it would be easier to allocate on the stack, because all the callee's data will live in the callee's stack frame.

6 (15 minutes). Generally speaking, languages that use call-by-name or call-by-need parameter-passing conventions avoid hangs and/or crashes that would occur if the same program were evaluated using call-by-value. Can you think of any counterexamples? That is, can you think of a program that has well-defined behavior with call-by-value, but which hangs with call-by-name or call-by-need? If so, give a counterexample; if not, explain why it's not possible to give a counterexample.

In call-by-value, the parameter is evaluated before the function is executed. In call-by-name, the parameter is evaluated each and every time it is accessed. Call-by-need is a memoized version of call-by-name, where the parameter is evaluated the first time it is accessed its value is cached for future accesses.

Call-by-name could cause a hang if the parameter was a function that changed some global state. For example,

```
int count = 0; //global state
int bar() {
  count++;
  if (count > 1) {
    while (true) { ... }
  }
  return 1;
}

void foo(int a) {
  printf("%d", a);
  printf("%d", a);
}

foo(bar());
```

In this example, bar will change global state and hang depending on that state. In call-by-value, bar is called only once, updates count to 1, and returns 1. In call-by-name, bar is called each time foo's parameter is accessed, which results in bar being called twice, and hanging on the second

```
call.

This example would not work for call-by-need. Another example, which would
be fine for call-by-value but would hang for call-by-name AND call-by-need:

bool __is_foo_called = false; //global state
int bar() {
  if (__is_foo_called) {
    while (true) { ... }
  }
  return 1;
}

void foo(int a) {
  __is_foo_called = true;
  print("%d", a);
}

foo(bar());

Here, call-by-value would succeed, because bar is evaluated before any part
of the body of foo.  In call-by-name and call-by-need, foo sets the global
state to true before its parameter is accessed.  Then, once its parameter
is accessed, bar is evaluated and goes into a loop.

7.  Write implementations of the following Prolog
predicates.  Use a clear and simple style.  Do
not assume that the input arguments are ground
terms.  You may assume the standard Prolog
predicates like member/2 and append/3, but avoid
them in favor of unification if you can.  Define
auxiliary predicates as needed.

(Note: you can do what you like with the empty list)

7a (5 minutes).  shift_left(L, R) succeeds if R
is the result of "shifting left" the list L by 1.
The leading element of L is lost.  For example,
shift_left([a,b,c], [b,c]).

shift_left([_|R], R).

7b (5 minutes).  shift_right(L, R) is similar,
except it shifts right.  For example,
shift_right([a,b,c], [a,b]).

shift_right([_], []).
shift_right([H|L], [H|R]) :- shift_right(L,R).

7c (5 minutes).  shift_left_circular(L, R) is
like shift_left, except the leading element of L
is reintroduced at the right.  For example,
shift_left_circular([a,b,c], [b,c,a]).

shift_left_circular([H|L], R) :- append(L, [H], R).
```



```
--or--
shift_left_circular([H|L], R) :- slc(L, H, R).
slc([], X, [X]).
slc([Y|L], H, [Y|R]) :- slc(L, H, R).

7d (5 minutes).  shift_right_circular(L, R) is
similar, except it shifts right.  For example,
shift_right_circular([a,b,c], [c,a,b]).

shift_right_circular(L,R) :- shift_left_circular(R,L).

7e (5 minutes).  Which of your implementation of
the predicates (a)-(d) might be nondeterministic?
For each such predicate, give an example call
that succeeds more than once.

There will be nondeterminism with any predicate which has multiple rules,
because the unification process can choose either and backtrack to try
another.  So (b), (c), and (d) will all have nondeterminism, because
(b) has multiple rules, append/3 in (c) has multiple rules, and (d) is
defined in terms of (c).

8 (10 minutes).  What is the set of possible
behaviors for each of the following Scheme
top-level expressions?  Assume that each one is
typed in to a fresh instance of a Scheme
interpreter.  Justify any tricky or obscure parts
of your answers by citing particular parts of the
formal semantics of Scheme.

(not (if #f #f))
You are not responsible for knowing the two-argument if statement, but
FYI it will nondeterministically return #t or #f if the conditional is
#f. So this can either evaluate to #f or #t.

(car (begin (set! car cdr) (list car cdr)))
See Dybvig section 2.3 for comments on order of evaluation of Scheme
expressions. To summarize, different implementations may evaluate in
different orders, possibly right-to-left or left-to-right. The value
of 'car' may be looked up to be the 'car' function before the right
expression (begin ...) is evaluated, or the begin expression may be
evaluated first.

In the first case (left-to-right), the first 'car'
would be evaluated to the 'car' function, and then the 'car' identifier
would be set to the 'cdr' function. Then, the list of the 'car' identifier
(looked up to be 'cdr') and the 'cdr' function would be created. The 'car'
from the beginning would access the first element, which is the 'cdr'
function (from the 'car' identifier).

In the second case (right-to-left), the (list...) subexpression would get
evaluated first, created a list of the 'car' function and the 'cdr' function.
Then the 'car' identifier would get bound to the 'cdr' function, and finally
the leftmost 'car' identifier would be looked up to be the 'cdr' function.
So (cdr (car cdr)) would evaluate to the 'cdr' function.
```



```
Note, however, that it is not even guaranteed that if an expression is
evaluated left-to-right, its subexpressions will also be evaluated
left-to-right.  Directionality may change in subexpressions, so you may
get some combination of the above behavior.

9 (5 minutes).  The abstract syntax for Scheme in
R5RS section 7.2.1 lists only constants,
identifiers, procedure calls, lambda, if, and
set!.  The following expressions aren't any of
these things, so why are they valid expressions
in Scheme?  Or if they're not valid expressions,
say why not.

(let ((a '())) a)
(cond ((eq? '() '()) 1) (#t 0))

The above two are macros that are replaced with the abstract syntax
defined in R5RS.

(1) - This is not valid because 1 is not a function.

10 (20 minutes).  Consider the following Java
code snippet, taken from BioJava.  Translate it
as best you can into idiomatic Python.  For parts
that you cannot easily translate, say what the
problems are, and how you'd go about addressing
these problems if you actually had to translate
all of BioJava into Python.
package org.biojava.bio.program.ensembl;
import java.util.*;
import java.lang.ref.*;
import org.biojava.bio.*;
...
class CloneDB extends ImmutableSequenceDBBase
       implements SequenceDB {
  private Ensembl ensembl;

  private Map cloneCache;
  private Set ids_cache;

  { cloneCache = new HashMap(); }

  public CloneDB(Ensembl db) {
      this.ensembl = db;
  }

  public String getName() {
      return "http://www.ensembl.org/";
  }

  Sequence getCloneSequence(Clone cont)
      throws EnsemblException
  {
    Sequence seq = (Sequence)
```



```
      cloneCache.get(cont.getID());
    if (seq == null) {
      if (ensembl.getUseHeavyClones()) {
        seq = new EnsemblHeavyCloneSequence
          (cont, ensembl);
      } else {
        seq = new EnsemblCloneSequence
          (cont, ensembl);
      }
      cloneCache.put(cont.getID(), seq);
    }
    return seq;
  }
  ...
}

(Note: you will not be responsible for knowing this exact problem,
 since it covers a HW which is no longer assigned. However, you
 should be able to convert Java code to Python and comment on the
 limitations.)

from java.util import *
from java.lang.ref import *
from org.biojava.bio import *

class CloneDB(ImmutableSequenceDBBase):
  def __init__(self, db):
    self.ensembl = db
    self.cloneCache = HashMap()

  def getName(self):
    return "http://www.ensembl.org/"

  def getCloneSequence(self, cont):
    seq = self.cloneCache.get(cont.getID())
    if seq == None:
      if self.ensembl.getUseHeavyClones():
        seq = EnsemblHeavyCloneSequence(cont, self.ensembl)
      else:
        seq = EnsemblCloneSequence(cont, self.ensembl)
      self.cloneCache.put(cont.getID(), seq)
    return seq

There are many issues.
1. There is not an equivalent to the package keyword. To achieve the same
   package semantics, you have to use directory-local __init__.py files as
   described here: https://docs.python.org/2/tutorial/modules.html#packages

2. There are no interfaces in Python, so you have to assume or manually check
   that your CloneDB class is implementing the correct functions.

3. Python does not really have a concept of private/public fields and methods.
   There is a trick you can do, detailed here:
   http://stackoverflow.com/questions/70528
```

4. Type information cannot be translated to Python because Python is
   dynamically typed.

5. The { cloneCache = new HashMap(); } line is an "initializer block", which
   is not really an issue, because in Java, the line is simply copied into
   every constructor by the compiler. It simply allows you to share large
   code blocks between different constructors. To translate to Python, just
   stick it in the __init__ method.

6. Python has no exception annotations, but you throw and catch similar to
   Java, so the "throws EnsemblException" is just left off.

7. You would have to assume that the Java libraries imported have also been
   ported to Python.

11 (15 minutes). For each of the following seemingly-arbitrary language rules, explain what performance gains (if any) arise because of the rules. Analyze both time and space performance. List the rules roughly in decreasing order of importance, performance-wise.

  * Arrays are origin-zero, not origin 1.

  Slightly faster. For origin-zero, to get the address of a[4], you will calculate a + 4*N, where N is the size of the array cell. For origin-1, to get the address of a[5], you calculate a + (5-1)*N, which is extra arithmetic.

  * Arrays must be allocated statically or on the
    stack; they cannot be allocated on the heap.

  This allows you to do away with any runtime memory manager (e.g. garbage collector), because all your memory will be cleaned up when a stack frame is popped off.

  * Each object in an array must have the same
    size.

  Faster, because you can do arithmetic indexing. However, this may waste space, because you have to make the size the maximum of whatever objects you want to put in.

  * Subscript violations result in undefined
    behavior; an exception may or may not be
    raised.

  Performance gain because you don't have any bounds-checking overhead. You also get a small space gain because you don't have to store the array length anywhere.

  * The array must have just one dimension.

  Performance loss due to the extra index calculation of both dimensions. Also the possibility of external fragmentation preventing the allocation

of a large array (which would otherwise be smaller chunks given a C++ or Java way of doing multidimensional arrays).

* The number of elements in an array is fixed
  at compile-time.

If you're using stack allocation of arrays, fixed size makes this faster. However, you may have to allocate a large array according to the max size it would become if it was dynamically allocated.

* Arrays must always be accessed directly;
  there are no pointers to arrays, and arrays
  cannot be passed by reference to procedures.

Very space-wasting, because you must copy arrays when passing them into functions. However, this avoids aliasing, which allows for better compiler optimizations, including constant propagation and code reordering.

12 (15 minutes). Suppose we add a new statement "return[N]EXPR;" to our C implementation, as an extension to C. N must be an integer constant expression with a nonnegative value, and EXPR can be any valid C expression. This new statement acts like "return EXPR;", except that it causes the Nth caller of the current function to return with the value of the expression EXPR. For example, suppose F calls G which calls H, and suppose H then executes "return[2]5;". This causes F to return immediately, with the value 5; G and H are silently exited in the process. By definition, "return[0]EXPR;" is equivalent to "return EXPR;".

What problems do you see in implementing "return[N]EXPR;"? Are these problems inherent to C, or can they be worked around by changing the semantics of the new statement slightly? Would it be easier to implement this new statement in some of the other languages that we have studied? If so, which ones and why? If not, why not?

A huge problem with this is centered around static typing. Because some function F does not know how deep in the call chain its return value will come from, it cannot know the type of that return value. Imagine the functions F, G, and H in the problem have return types void, int, and double, respectively. H executing "return[2]1.0" would cause F to try to return 1.0. These problems are inherent to C.

This would be much easier to implement in Python, for example, because of dynamic typing. The return type of the function is not specified, so returning some alternative value would not be a problem.

This sample midterm is the midterm from Spring 2008. If you see any errors in the sample answer, please email me or tell me on Piazza. Good luck on your midterms!

1. "Ireland has leprechauns galore." is an example of a particular kind of syntactic construct in English. Can you construct a similar example in C++, OCaml, or Java? If so, give an example; if not, explain why not

- Artificial languages are generally more carefully-designed than natural languages. Thus, these kinds of exceptions should be rare, if they exist at all.

2.
a) Write an OCaml function 'twice' that accepts a function f and returns a function g such that g(x) equals f(f(x)). For simplicity's sake, you can assume that f is free of side effects, and you can impose other restrictions on f and x. Try to keep the restrictions as minor as possible, and explain any restrictions you impose. Or, if 'twice' cannot be written easily in OCaml, explain why not.

- let twice f x = f(f(x))
- restriction is that f is 'a -> 'a

b) Same as a) except write a function 'half' that accepts a function f and returns a function g such that f(x) equals g(g(x)).

- It is difficult, since its implementation must depend on the specific function f used.
- For example, if f is identity function f(x) = x, then g is easy-- also the identity function.
- However, if f is the sine function, then g is super difficult-- given x, g(g(x)) = sin(x)???
- The restriction, like a), is that f is 'a->'a

c) Give the types of 'twice' and 'half'
- twice: ('a->'a) -> 'a -> 'a
- half: ('a -> 'a) -> 'a -> 'a

3. Consider the following grammar for a subset of the C++ language.

```
expression:
      expression ? expression : expression
      expression != expression
      expression + expression
      ! expression
      INTEGER-CONSTANT
      ( expression )
```

For example, (!! 0+1 != 2 ? 3 : 4) is read as "if not-not-0 plus 1 does not equal 2, then 3 else 4, and evaluates to 4.

a) What are the tokens of this subset of C++?
      ? : != + ! INTEGER_CONSTANT ( )

b) Show that this grammar is ambiguous
      Draw the two parse trees for expression 1+2+3

c) Rewrite the grammar so that it is no longer ambiguous, resolving any ambiguities in the same way that c++ does. Recall that in C-+, the expression

(0 != 1 != 2 || 3 + !4 + 5 || 6 ? 7 : 8 ? 9 : 10) is like

((((((0 != 1) != 2) || ((3 + (!4)) + 5)) || 6) ? 7 : (8 ? 9 : 10))

```
E -> E2 ? E2 : E | E2
E2 -> E2 != E3 | E3
E3 -> E3 + E4 | E4
E4 -> !E4 | E5
E5 -> INTEGER-CONST | ( E )
```

d) Translate the rewritten grammar into a syntax diagram

```
eg. E2
      o------> E3 ----->o
          |<-- != <--|
```

4. A numerical analyst is really bothered by the special value of IEEE floating point, and asks you to modify ~~Java~~ C++ to fix what she view as a serious conceptual flaw. She wants her Java programs to throw an exception instead of returning infinities and NaNs. Is her request reasonable for ~~Java~~ C++ programs? Is it implementable? Why/why not? Don't worry about compatibility with existing compilers, etc.; assume that you are the inventor of ~~Java~~ C++ and she is asking for this feature early in your language design process.

- It's kind of reasonable...maybe the numerical analyst has applications that don't allow Infs and NaNs. However, this gets rid of the choice of having these special values. To implement it, ~~Java~~ C++ can throw an exception whenever it encounters a special value.

5. Give an example of four distinct ~~Java~~ C++ types A, B, C, D such that A is a subtype of B, A is a subtype of C, B is a subtype of D, and C is a subtype of D. Or, if such an example is impossible, explain why not.

- class D {...}
- class A: public B, public C {...}
- class B: public D {...}
- class C: public D {...}

6. Explain how you would implement OCaml-style type checking, in an implementation that uses dynamic linking heavily. What problems do you foresee in programs that relink themselves on the fly?

- (verbatim from the professor) The problem is that one needs to do type checking as well as the usual name checking that link editing normally does. And the type checking needs to follow OCaml's rules. In particular it needs to work with generic types, where the type of the linked-to function does not exactly match the type needed by the caller, but the types will match after applying a unifier.

7.
a) Write a curried OCaml function "interleave C S L1 L2" that constructs a new list L from the lists L1 and L2, using the chooser C with seed S, and returns a pair (S1,L) where S1 is the resulting seed and L is the interleaved list. For example, "interleave C S [1;2] [3;4;5]" might invoke C four times and then return (S1, [1;3;4;2;5]). At each step of the iteration, "interleave" should use the choose to decide whether to choose the first item of L1 or the first item of L2, when deciding which of the two items to put next into L. If L1 is empty, the choose need not be used, since "interleave" will just return L2; and similarly if L2 is empty, "interleave" should just return L1 without invoking C.

Chooser C is defined
http://www.cs.ucla.edu/classes/spring08/cs131/hw/hw1.html

"interleave" should invoke C a minimal number of times, left to right across the lists L1 and L2. "interleave" should avoid side effects; it should be written in a functional style, without using OCaml libraries.

```
let rec interleave c s l1 l2 =
if l1 = [] then (s,l2) else if l2 = [] then (s,l1) else
match (c s) with
| (true, s1) -> (match l1 with
                  | h::t -> (let (s2,l) = (interleave c s1 t l2) in
                                          (s2, h::l)))
| (false, s1) -> (match l2 with
                  | h::t -> (let (s2,l) = (interleave c s1 l1 t) in
                                          (s2, h::l)))
```

b) Write a function "outerleave" that does the opposite of what "interleave does": it splits a list into two sublists that can be interleaved to get the original list, and returns a triplet consisting of the new seed and the two sublists. That is, "outerleave C S [1;3;4;2;5]" might yield (S1, [1;3;2], [4;5]). If given a list of length N, "outerleave" always invokes the chooser N times.

```
let rec outerleave c s l =
match l with
| [] -> (s, [], [])
| h::t -> match (c s) with
                  | (true, s1) -> (let (s2, l1, l2) = (outerleave c s1 t) in
                                                  (s2, h::l1, l2))
                  | (false, s1) -> (let (s2, l1, l2) = (outerleave c s1 t) in
                                                  (s2, l1, h::l2))
```

c) Give the data types of all top-level values or functions defined in your answer to a) and b).

- interleave: ('a -> bool * 'a) -> 'a -> 'b list -> 'b list -> 'a * 'b list
- outerleave: ('a -> bool * 'a) -> 'a -> 'b list -> 'a * 'b list * 'b list
- chooser c: 'a -> (bool, 'a)
- seed s: 'a
- list l1/l2/l: 'b list

Scanned by CamScanner
Scanned by CamScanner
Scanned by CamScanner

UCLA CS 131 Midterm, Fall 2017
100 minutes total, open book, open notes
closed computer

Name:_____ Student ID:_____

1	2	3	4	5	6	total

1a (10 minutes). Write an OCaml function merge_sorted that merges two sorted lists. Its first argument should be a comparison function 'lt' that compares two list elements and returns true if the first element is less than the second. Its second and third arguments should be the lists to be merged. For example, (merge_sorted (<) [21; 49; 61] [-5; 20; 25; 49; 50; 100]) should yield [-5; 20; 21; 25; 49; 49; 49; 50; 61; 100].

1b (3 minutes). What is the type of merge_sorted?

1c (3 minutes). What does the following expression yield, and what is its type?

```
merge_sorted (fun a b -> List.length a < List.length b)
```

1d (8 minutes). Is your implementation of merge_sorted tail-recursive? If so, briefly say why it won't have any problem with stack overflow. If not, briefly say why not, and explain any problems you would have in rewriting your implementation to make it tail-recursive.

2 (9 minutes). Consider the following top-level OCaml definitions:

```
let f f = f 1 1
let g g = g 0.0 g
let h h = h f "x"
```

For each identifier declared in this code, give the identifier's scope and type. Or, if there is a scope or type error, briefly explain the error.

3a (5 minutes). In Java, is the subtype relation transitive? That is, if A is a subtype of B and B is a subtype of C, is A a subtype of C? If so, explain why; if not, give a counterexample.

3b (5 minutes). In Java, is the graph of the subtype relation a tree? If so, explain why, and say what the root is; if not, give a counterexample.

4. Consider the following grammar for declarations in a subset of C.

Ths grammar uses a form of EBNF in which the left hand side is not indented and is followed by ":", each right hand alternative is indented, nonterminals are strings of letters and "-", terminal symbols are either surrounded by single quotes or are INT (meaning an integer constant) or ID (meaning an identifier), and X? stands for zero or one instances of X.

```
declaration:
  declaration-specifiers init-declarator-list? ';'

declaration-specifiers:
  storage-class-specifier declaration-specifiers?
  type-specifier declaration-specifiers?
  type-qualifier declaration-specifiers?
  function-specifier declaration-specifiers?

storage-class-specifier:
  'typedef'
  'static'

type-specifier:
  'void'
  'char'
  'int'

type-qualifier:
  'const'
  'volatile'

function-specifier:
  'inline'
  '_Noreturn'

init-declarator-list:
  init-declarator
  init-declarator-list ',' init-declarator

init-declarator:
  declarator
  declarator '=' initializer

declarator:
  pointer? direct-declarator

direct-declarator:
  ID
  '(' declarator ')'
  direct-declarator '[' INT ']'
  direct-declarator '(' 'void' ')'

pointer:
  '*' type-qualifier-list? pointer?

type-qualifier-list:
  type-qualifier-list? type-qualifier

initializer:
  ID
  INT
```

4a (2 minutes). What makes this grammar EBNF and not simply BNF?

4b (8 minutes). Give an example declaration that is syntactically correct (i.e., it is produced by this grammar) but is semantically

incorrect for C. Prove that it is syntactically correct. Briefly explain why it is semantically incorrect.

4c (5 minutes). Suppose we changed the grammar by replacing the ruleset for type-qualifier-list with the following:

```
type-qualifier-list:
  type-qualifier type-qualifier-list?
```

Would this cause any problems? If so, describe a problem and give an example. If not, briefly explain why not.

4d (10 minutes). Suppose we changed the original grammar by replacing the two rulesets for declarator and direct-declarator with the following single ruleset:

```
declarator:
  pointer? declarator
  ID
  '(' declarator ')'
  declarator '[' INT ']'
  declarator '(' 'void' ')'
```

Would this cause any problems? If so, describe a problem and give an example. If not, briefly explain why not.

4e (10 minutes). Draw a syntax chart for the original grammar.

5 (10 minutes). Suppose we write Java code in a purely functional style, in that we never assign to any variables except when initializing them. That is, we always initialize local variables and never assign to them later, and we always initialize instance variables once at the start of constructors and never assign to them later.

In our purely-functional Java programs, is the Java Memory Model still relevant, or can we ignore it? If it's still relevant, explain which parts of it still apply and give an example. If not, briefly explain why not.

6 (12 minutes). Consider the following code, taken from the answer to the older version of Homework 2.

```
let match_empty frag accept = accept frag

let match_nothing frag accept = None

let rec match_star matcher frag accept =
  match accept frag with
    | None ->
        matcher frag
                (fun frag1 ->
                   if frag == frag1
                   then None
                   else match_star matcher frag1 accept)
    | ok -> ok

let match_nucleotide nt frag accept =
  match frag with
    | [] -> None
    | n::tail -> if n == nt then accept tail else None

let append_matchers matcher1 matcher2 frag accept =
  matcher1 frag (fun frag1 -> matcher2 frag1 accept)

let make_appended_matchers make_a_matcher ls =
  let rec mams = function
    | [] -> match_empty
    | head::tail -> append_matchers (make_a_matcher head) (mams tail)
  in mams ls
```

In this code, a matcher is a curried function taking two arguments: first, a fragment 'frag' and second, an acceptor 'accept'. Suppose we change the API for matchers by interchanging their arguments, so that the acceptor comes first (all the functions remain curried). Rewrite the above code to use the altered API, and simplify the resulting code as much as possible.