Midterm II — LDI, Spring 2016

UVa ID: (yes, again!) KEY

Problem	Max	Your Points
1 — Type Checking and Let	15	
2 — Adding New Expressions	22	
3 — Optimization	13	
4 — Exceptions	9	
5 — Automatic Memory Management	8	
6 — Code Generation	12	
7 — Game Theory	5	
8 — Debugging, Typing and Opsem	16	
Extra Credit	0	
TOTAL	100	

Honor Pledge:

How do you think you did? ______________________________

1 Type Checking and Let (15 points)

Consider the following two *incorrect* typing judgments for the non-SELF_TYPE case of initialized variable introduction:

$$\frac{\begin{array}{l} T_0' = \begin{cases} \texttt{SELF\_TYPE}_\texttt{C} & \text{if } T_0 = \texttt{SELF\_TYPE} \\ T_0 & \text{otherwise} \end{cases} \\ O, M, C \vdash e_1 : T_1 \\ T_1 \leq T_0' \\ O[T_1/x], M, C \vdash e_2 : T_2 \end{array}}{O, M, C \vdash \text{let } x : T_0 \leftarrow e_1 \text{ in } e_2 : T_2} \text{ wrong}$$

$$\frac{\begin{array}{l} T_0' = \begin{cases} \texttt{SELF\_TYPE}_\texttt{C} & \text{if } T_0 = \texttt{SELF\_TYPE} \\ T_0 & \text{otherwise} \end{cases} \\ O, M, C \vdash e_1 : T_1 \\ T_0' \leq T_1 \\ O[T_0'/x], M, C \vdash e_2 : T_2 \end{array}}{O, M, C \vdash \text{let } x : T_0 \leftarrow e_1 \text{ in } e_2 : T_2} \text{ alsobad}$$

(a) *[7 pts]* The modified typing rule wrong is *too strict*: it rejects good programs that are accepted by the normal typing rules. Write a Cool expression that is accepted by the normal typing rules but is rejected by rule wrong.

The issue is that wrong has $O[T_1/x]$ but a correct rule would have $O[T_0'/x]$. That is, when evaluating the let-body, x is given the static type of the initializer. So the code below will not type check with wrong:

```
let x : Object <- 2 in x <- new Object
```

(b) *[8 pts]* The modified typing rule alsobad is also *unsound*: it allows programs that will lead to run-time errors. Write a Cool expression that is accepted by rule alsobad but that would violate the normal typing rules and lead to undefined or unsafe behavior at runtime.

The issue is that alsobad uses $T_0' \leq T_1$ while the correct rule would use $T_1 \leq T_0'$. This means that supertypes are accepted where only subtypes should be allowed! The code below will lead to undefined behavior at runtime because the m object will not have a *main* method:

```
let m : Main <- new Object in m.main()
```

2 Adding New Expressions (22 points)

We would like to add an `extend to` expression to Cool. In standard Cool, the *dynamic type* of an object is set when it is created by `new` and can never be changed. The `extend to` expression changes the dynamic type of an object to that of the specified subtype, allocates space for additional fields not already present at runtime, assigns them default values, and then executes initializers for the new fields only. The extend expression returns `void` and does not change static types. Consider this example:

```
class Point inherits IO {
  x : Int ;
  setX(newX : Int) : Object { x <- newX };
  identify() : Object { out_string("I am a Point") };
}
class ColorPoint {
  c : String <- "red" ;
  y : Int <- x + 1 ;
  identify() : Object { { out_string(c) ; out_int(x+y); }};
} ;
class Main {
  main() : Object {
    let p : Point <- new Point in {
      p.setX(2) ;
      extend p to ColorPoint ;
      p.identify() ; -- prints "red5"
  } } ;
} ;
```

Intuitively, `extend to` is like a "partial" combination of `new` and assignment that leaves existing fields alone but processes fields in the extension.

(a) *[6 pts]* Complete the typing rule for `extend`.

$$\frac{\begin{array}{c} O(id) = T_{id} \\ T \leq T_{id} \end{array}}{O, M, C \vdash \texttt{extend}\ \text{id}\ \texttt{to}\ T\ : Object}\ \textsf{extend - typecheck}$$

(b) *[12 pts]* Complete the operational semantics rule for `extend`.

We assume without loss that all fields in all classes have distinct names.

$$\frac{\begin{array}{l} E[id] = addr \\ S_1[addr] = X(a_1 = l_1, \ldots, a_n = l_n) \\ class(T) = (a_1 : T_1 \leftarrow e_1, \ldots, a_m : T_m \leftarrow e_m) \\ n \leq m \\ l_i = newloc(S_1), for\ i = n_1 \ldots m\ and\ each\ l_i\ is\ distinct \\ v = T(a_1 = l_1, \ldots, a_n = l_n, \ldots a_m = l_m) \\ S_2 = S_1[D_{T_{n+1}}/l_{i_{n+1}}, \ldots, D_{T_m}/l_m] \\ S_3 = S_2[v/addr] \\ v, S_3, [a_1 : l_1, \ldots, a_m : l_m] \vdash a_{n_1} \leftarrow e_{n_1}; \ldots a_m \leftarrow e_m : v', S_4 \end{array}}{so, S_1, E \vdash \texttt{extend}\ \mathsf{id}\ \texttt{to}\ T\ : void, S_4}\ \textsf{extend2}$$

In the first line we look up the address of the identifier, and in the second we load its value from the store, noting the current attributes $a_1 \ldots a_n$. We then look up the subtype T and find that it has more attributes, all the way up to a_m ($m \geq n$). Because $T \leq X$, they agree on the first n attributes. We then allocate new space for the attributes from n_1 to m — the "new" attributes — and build a new object v that holds them (as well as the "old" ones). We then give all of the new attributes their default values based on their types. Then (*careful!*) we replace the current value of id (which lives at $addr$) with the new value v. Finally, we execute all of the initializers for the new attributes.

This is a (partial) combination of the New rule and the Assign rule.

As Alec pointed out, the above formulation does not handle multiple uses of *extend* correctly because it assumes the old fields will always be a strict prefix of the new fields. Students were not expected to handle this correctly for full credit, but the (more complicated) answer below is one way to do so (a_i and b_i both range over field names; l and l' both range over addresses; and *newfields* indicates the indices of the fields in T that are not yet found in v):

$$\frac{\begin{array}{l} E[id] = addr \\ S_1[addr] = X(a_1 = l_1, \ldots, a_n = l_n) \\ class(T) = (b_1 : T_1 \leftarrow e_1, \ldots, b_m : T_m \leftarrow e_m) \\ newfields = \{j \mid j \in [1 \ldots m] \ \wedge\ \neg\exists i.\ a_i = b_j\} \\ l'_j = newloc(S_1), for\ j \in newfields\ and\ each\ l_j\ is\ distinct \\ v = T(a_1 = l_1, \ldots, a_n = l_n, \ldots, b_j = l'_j, \ldots)\ (j \in newfields) \\ S_2 = S_1[\ldots, D_{T_j}/l'_j, \ldots]\ (j \in newfields) \\ S_3 = S_2[v/addr] \\ v, S_3, [a_1 : l_1, \ldots, a_n : l_n, \ldots, b_j : l'_j, \ldots] \vdash \ldots a_j \leftarrow e_j; \ldots : v', S_4\ (j \in newfields) \end{array}}{so, S_1, E \vdash \texttt{extend}\ \mathsf{id}\ \texttt{to}\ T\ : void, S_4}\ \textsf{extend} - \textsf{opsem}$$

(c) *[4 pts]* The `extend to` expression allows us to simulate some aspects of multiple inheritance. Write a Cool example that returns an object that, at run-time, has all fields from two *siblings* in the class hierarchy. Write any class declarations first (keep them simple) and then write the expression. (We do not care about Cool *syntax* like semicolons.)

```
class Shape { x : Int; y : Int; } ;
class Circle inherits Shape { radius : Int ; } ;
class Triangle inherits Shape { angle : Int ; } ;

let s : Shape <- new Shape in {
  extend s to Circle ;
  extend s to Triangle ;
  (* s now has the radius field and the angle field, as well as x and y *)
}
```

3 Optimization (13 points)

(a) *[8 pts]* The following block of code makes use of five variables: `a`, `b`, `c`, `d`, and `e`. However, we have erased many of the variable references from the original program. In the right-hand column, we provide the results of liveness analysis (i.e., the variables that are live at each program point). Please fill in each blank with a single **variable** so that the program is consistent with the results of liveness analysis.

Please note that there are **no dead instructions** in this program. (This means only that `X` is always live right after each assignment to `X`; it doesn't mean that you couldn't personally think of some optimizations to apply here.) You will need this information to fill in some of the blanks correctly!

Code **Live Variables**

{a, b, c}

$\boxed{b}$:= $\boxed{a}$ - b

{b, c}

$\boxed{e}$:= b + $\boxed{c}$

{b, e}

$\boxed{d}$:= $\boxed{e}$ + 1

{b, d}

b := $\boxed{b}$

{b, d}

$\boxed{c}$:= 123

{b, c, d}

$\boxed{e}$:= b + $\boxed{c}$

{e, d}

print $\boxed{d}$

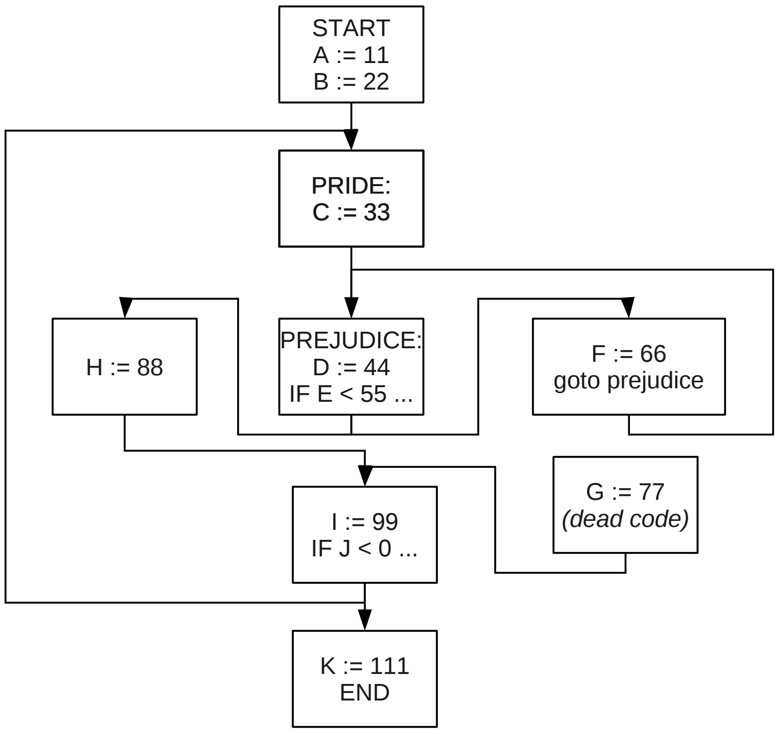
{e}

print $\boxed{e}$

{}

(b) *[5 pts]* Draw a control-flow graph for the following code. Each node in your control-flow graph should be a *basic block*. Do not worry about static single assignment form. *Every* statement in the code should appear somewhere in your control-flow drawing.

```
           START
           a <- 11
           b <- 22
pride:     c <- 33
prejudice: d <- 44
           if (e > 55) then {
                 f <- 66
                 goto prejudice
                 g <- 77
           } else {
                 h <- 88
           }
           i <- 99
           if (j > 0) then { goto pride }
           k <- 111
           END
```



4 Exceptions (9 points)

(a) *[2 pts]* Give one example of an exception that is the clear result of a mistake in a program. Then give one example of an exception that might be raised even in a perfect program.

Mistake: Dereferencing a null pointer. Division by zero. And so on.

Environmental: Out of disk space. Network packet loss. And so on.

(b) *[3 pts]* Explain where and why the least-upper-bound operator $\sqcup$ is used in our formal treatment of language-level exception handling.

In *try* e_1 *catch* e_2 we do not know statically whether e_1 or e_2 will be executed. As a result, the type for the expression must by a conservative approximation that safely describes both T_1 and T_2. We thus use $T_1 \sqcup T_2$ as the return type.

(c) *[4 pts]* Consider the following *incorrect* unsound typing rule for `try-finally`.

$$\frac{O, M, C \vdash e_1 : T_1 \quad O, M, C \vdash e_2 : T_2}{O, M, C \vdash \texttt{try } e_1 \texttt{ finally } e_2 : T_1}\ \textsf{nofun}$$

Give a Cool *expression* that does typecheck using this rule but would lead to a run-time error.

This issue is that this typing rule always returns the type associated with e_1, even though the operational semantics return the value associated with e_2. Example that would add a string to an int at runtime:

```
(try 1 finally "hello") + 2
```

5 Automatic Memory Management (8 points)

(a) *[2 pts]* Name two specific disadvantages of *Mark and Sweep.* (Be specific. Just saying that it it slow, for example, is not adequate.)

Mark and Sweep requires you to store a "mark bit" with each allocated object. Mark and Sweep takes time proportional to all available memory, even if you are only using a little bit of it. Mark and Sweep can lead to fragmentation.

(b) *[3 pts]* Consider the following program:

```
while not_done() {
  ptr = malloc(100 * MEGABYTE);
  do_work(ptr);
  /* done with ptr */
}
```

You are running this program with 4 gigabytes of physical memory and want to use automatic memory management. Would you choose *Stop and Copy* or *Mark and Sweep*? Why?

Interestingly, Stop and Copy is provably/mathematically better in this case. With Mark and Sweep, you will invoke the garbage collector every $4GB/100MB = 40$ iterations around the loop. Once invoked, Mark and Sweep takes time proportional to 4 GB.

By contrast, Stop and Copy will be invoked twice as often (because it cuts usable memory in half) — so once every 20 iterations around the loop. But once invoked, it will only take time proportional to reachable memory (100 MB). So Stop and Copy is invoked twice as often, but costs 40 times less: ultimately Stop and Copy is 20 times faster!

(c) *[3 pts]* Suppose that you have already decided to use garbage collection. Which automatic memory management scheme would you apply to a C or C++ program that was *not* written with garbage collection in mind? Which scheme would run into trouble?

You could potentially apply a conservative version of Mark and Sweep. Because you cannot tell the difference between integers and pointers, you will have to "trace" some integers, which means that you will mistakenly fail to free some dead memory. The Boehm-Weiser conservative garbage collected for C and C++ does just that (`http://hboehm.info/gc/` — it works, check it out).

You *cannot* use Stop and Copy, because you cannot safely move around potential objects in C. (For exampe, if you think an integer is a pointer to an object and you "copy" it and move it, you will change the meaning of the program.)

6 Code Generation (12 points)

(a) *[6 pts]* Consider the following *incorrect* stack-machine code generation rule:

```
          cgen(if e1 = e2 then e3 else e4) =
                                cgen(e1)
                                push r1
                                cgen(e2)
                                pop t1
                                beq r1 t1 true_branch
                false_branch:   cgen(e4)
                true_branch:    cgen(e3)
                end_if:
```

Write an expression e for which the above rule generates incorrect code. Indicate specifically what should happen when your expression e is evaluated as well as what mistakenly happens when the above rule is used to generate code.

```
if 1 = 2 then print "a" else print "b"
```

With the incorrect rule, both "b" and "a" will be printed (because there is no jump after the false branch around the true branch). In a correct implementation, only "b" should be printed.

(b) *[6 pts]* Consider three classes, `A`, `B`, and `C`. Consider the following object (field) and dispatch table (vtable) layouts:

	0 – 2	3	4	5	6	7
A	(header)	e	t	a	o	n
B	(header)	e	t			
C	(header)	e	t	r		

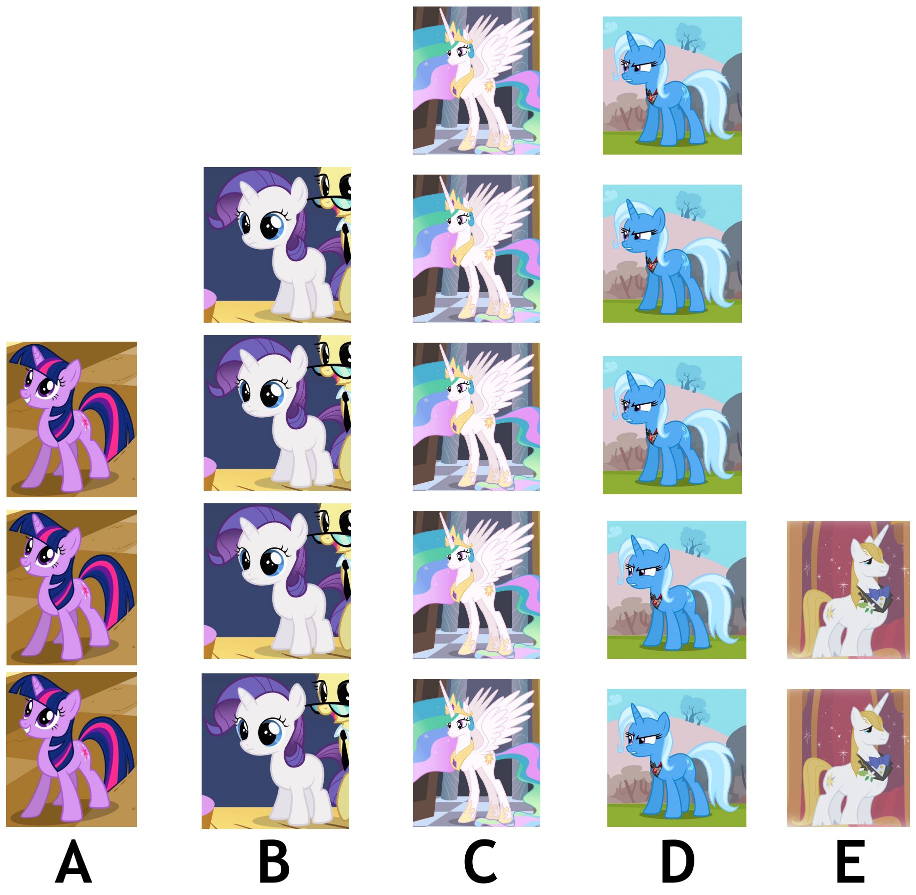
	0	1	2	3
A	A's x()	A's y()	B's z()	A's w()
B	B's x()	B's y()	B's z()	
C	B's x()	C's y()	C's z()	

Write the three class declarations, showing all inheritance and field and method declaraitons. (All of the fields are `Int`s.)

```
class B {
  e : Int ; t : Int ;
  x() { ... }; y() { ... }; z() { ... };
};
class A inherits B {
  a : Int; o: Int; n: Int;
  x() { (* override *) ... }; y() { (* override *) ... };
  w() { ... };
};
class C inherits B {
  r : Int ;
  y() { (* override *) ... }; z() { (* override *) ... };
}
```

7 Game Theory (5 points)

(a) *[5 pts]* Consider the following *Nim* scenario. It is your turn. Indicate a winning move (e.g., write it out textually or circle the items you would take from a single heap). In the game board below, heap **A** has three items, heap **B** has four items, etc.



The board has value $3 \oplus 4 \oplus 5 \oplus 5 \oplus 2$. The first thing to note is that $5 \oplus 5 = 0$, so you can just ignore columns C and D. That leaves us with $3 \oplus 4 \oplus 2 = 5$, which is a win for the current player.

The winning move is to take 3 from B, leaving $3 \oplus 1 \oplus 2 = 0$.

8 Debugging, Typing and Opsem (16 points)

Consider these OCaml programs that *do not type-check* and their corresponding error messages (including the implicated code, shown underlined). Each has comments detailing what the program *should* do as well as sample invocations that *should* type-check.

```
(* "sumlist xs" returns the sum of the
    integer elements of "xs" *)
let rec sumList xs = match xs with
    | []       -> []
    | y ::  ys -> y + sumList ys

assert( sumList [1;2] = 3 );;

This expression has type
    'a list
but an expression was expected of type
    int
```

```
(* "wwhile (f, x)" returns x' where there exist
    values v0, ..., vn such that:

    - x is equal to v0
    - x' is equal to vn
    - for each i between 0 and n-2, we have
        (f vi) equals (vi+1, true)
    - (f vn-1) equals (vn, false) *)

let f x =
    let xx = x * x in
    (xx, (xx < 100))

let rec wwhile (f,b) =
    match f with
    | (z, false) -> z
    | (z, true)  -> wwhile (f, z)

assert( wwhile (f, 2) = 256 ) ;;

This expression has type
    int -> int * bool
but an expression was expected of type
    'a * bool
```

(a) *[2 pts]* Why is the `sumList` program not well-typed?

It is trying to add an integer to a list during the recursive step.

(b) *[2 pts]* Fix the code so that `sumList` works correctly.

Change the base case so that it returns 0 instead of [ ].

(c) *[2 pts]* Why is the `wwhile` program not well-typed?

It is treating the function f as if it were a tuple.

(d) *[2 pts]* Fix the `wwhile` program.

Change *match f with* to *match* (f b) *with*.

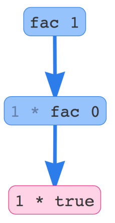
Consider an *execution trace* that shows a high-level overview of a program execution focusing on function calls. For example, the trace on the right tells us that:

i. We start off with `fac 1`.

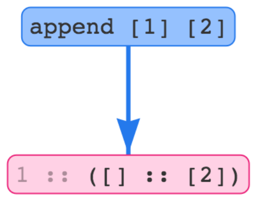
ii. After performing some computation, we have the expression `1 * fac 0`. The `1 *` is grayed out, indicating that `fac 0` is the next expression to be evaluated.

iii. When we return from `fac 0`, we are left with `1 * true`, indicating a program error: we cannot multiply an `int` with a `bool`.

```
let rec fac n =
    if n <= 0 then
        true
    else
        n * fac (n - 1)

assert( fac 1 = 1 ) ;;
```



```
(* "append xs ys" returns a list containing the
   elements of "xs" followed by the elements of "ys" *)
let rec append xs ys =
    match xs with
    | []   -> ys
    | h::t -> h ::  t ::  ys

assert( append [1] [2] = [1;2] ) ;;

Error encountered because
    int
is incompatible with
    int list
```



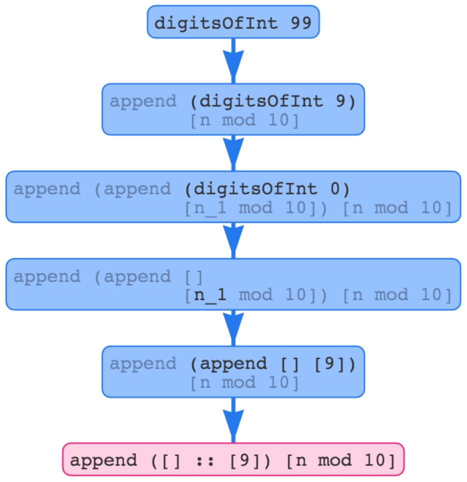
(e) *[2 pts]* Why is the `append` program not well-typed?

It is trying to cons a list onto a list (instead of consing an element onto a list).

(f) *[2 pts]* Fix the code so that `append` works correctly.

Replace $h :: t :: ys$ with $h :: (appendtys)$.

```
(* "digitsOfInt n" returns "[]" if "n" is
    not positive, and otherwise returns the
    list of digits of "n" in the order in
    which they appear in "n".  *)
let rec append x xs =
    match xs with
    | [] -> [x]
    | _  -> x ::  xs

let rec digitsOfInt n =
    if n <= 0 then
       []
    else
       append (digitsOfInt (n/10))
              [n mod 10]

assert( digitsOfInt 99 = [9;9] ) ;;

Error encountered because
    'a list
is incompatible with
    int
```



```
digitsOfInt 99

append (digitsOfInt 9)
       [n mod 10]

append (append (digitsOfInt 0)
               [n_1 mod 10]) [n mod 10]

append (append []
               [n_1 mod 10]) [n mod 10]

append (append [] [9])
       [n mod 10]

append ([] :: [9]) [n mod 10]
```

(g) *[2 pts]* Why is the `digitsOfInt` program not well-typed?

It is trying to cons the empty list on to a list of integers.

(h) *[2 pts]* Fix the code so that `digitsOfInt` works correctly.

Swap the order of the arguments to the call to append in digitsOfInt:

```
let rec append x xs =
  match xs with
  | [] -> x
  | _ -> x @ other
```

(many other solutions are possible)

9 Extra Credit (0 points)

"No Answer" is *not* valid on extra credit questions.

(a) *[1 pt]*Answer the following true-false questions about `SELF_TYPE`.

i. TRUE: `SELF_TYPE` is a static type.

ii. false: `SELF_TYPE` helps us to reject incorrect programs that are not rejected by the normal type system.

iii. TRUE: $\texttt{SELF\_TYPE}_T \leq T$

iv. false: A formal parameter to a method can have type `SELF_TYPE`.

v. TRUE: If the return type of method f is `SELF_TYPE` then the static type of $e_0.f(e_1, \ldots, e_n)$ is the static type of e_0.

(b) *[1 pt]* List three important considerations when linking a program with a shared library.

Multiple programs should share copies of the library code segment. Multiple programs should get their own private copies of library data. A linkage (GP) register may be involved in locating data. Library code may have to call back to program code. Import, export and relocation tables are used.

(c) *[1 pt]* Explain the interaction between Python's duck typing and interfacing Python with C.

When writing the C interface code, all functions called from Python appear to accept a generic "Object" as an argument. Because a Python function foo could be called $foo(5)$ or $foo(\text{“}hello'', 5)$, the C programmer must test to see if that Object is an integer or a tuple of a string and an integer. This is done by querying Python's runtime system, as in:

```
if ( PyArg_ParseTuple ( args , "s#s", & n_plain , & plain_size , & n_keytext))
```

or

```
if ( PyArg_ParseTuple ( args , "s#i" , & n_plain , & plain_size , & n_mask))
```

The former looks for two strings, the latter looks for a string followed by an integer. Python makes this determination via duck-typing (e.g., something qualifies as a string or tuple if it has the right properties).

(d) *[2 pts]* Cultural literacy. Below are the English translations or names for ten concepts or figures in world folklore, legend, religion or mythology. Each concept is associated with one of the ten most common languages (by current number of native-language speakers; Ethnologue estimate). For each concept, give the associated language. Be specific.

- Chinese (Mandarin). The Eight Immortals.
- Javanese. Panji.
- English. King Arthur.
- Japanese. Kami.
- Hindi. Kamayani.
- Arabic. Jinn.
- Russian (Slavic). Ilya Muromets.
- Portuguese. Endovelicus.
- Spanish. Don Juan.
- Bengali. Bankubabur Bandhu.