

CS 242 Midterm Exam

This is a *closed*-book exam. The maximum possible score is 100 points. 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.

The following is a statement of the Stanford University Honor Code:

- A. The Honor Code is an undertaking of the students, individually and collectively:
- (1) that they will not give or receive aid in examinations; that they will not give or receive unpermitted aid in class work, in the preparation of reports, or in any other work that is to be used by the instructor as the basis of grading;
 - (2) that they will do their share and take an active part in seeing to it that others as well as themselves uphold the spirit and letter of the Honor Code.
- B. The faculty on its part manifests its confidence in the honor of its students by refraining from proctoring examinations and from taking unusual and unreasonable precautions to prevent the forms of dishonesty mentioned above. The faculty will also avoid, as far as practicable, academic procedures that create temptations to violate the Honor Code.
- C. While the faculty alone has the right and obligation to set academic requirements, the students and faculty will work together to establish optimal conditions for honorable academic work.

I acknowledge and accept the Honor Code.

(Signature)

(Print your name)

Prob	# 1	# 2	# 3	# 4	# 5	# 6	# 7	# 8	# 9	# 10	Total
Score											
Max	10	9	12	7	6	10	12	10	16	8	100

1. (10 points) True or False

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

- _____ (a) Allocation and deallocation of activation records is always done using a stack (LIFO) structure. (*False*)
- _____ (b) Two function parameters passed by value are never aliased inside the function body. (*True*)
- _____ (c) Two function parameters passed by reference are never aliased inside the function body. (*False*)
- _____ (d) Two function parameters, one passed by value and the other by reference, are never aliased inside the function body. (*True*)
- _____ (e) A dangling pointer refers to a region of memory that has been deallocated. (*True*)
- _____ (f) Determining whether a C function has a side effect is undecidable. (*True*)
- _____ (g) The tree reduce function (from homework) is a higher-order function. (*True*)
- _____ (h) A value on the left-hand-side of an operator is called an L-value. (*False*)
- _____ (i) In principle, it is possible to parallelize C code by first translating a C program to a pure functional language and then evaluating independent parts of the functional program in parallel. (*True*)
- _____ (j) Java, C, and Lisp programming languages are Turing complete but ML is not. (*False*)

2. (9 points) Terminology

Define the following terms.

- (a) (3 points) alpha-conversion **Answer:** Its definition.
- (b) (3 points) polymorphic function **Answer:** Its definition.
- (c) (3 points) garbage **Answer:** Its definition.

3. (12 points) Short Answer

- (a) (3 points) Write the “dotted pair” Lisp expression for the list (A B C D). Your answer might look something like ((X . Y) . (Z . W)).
Answer: (A . (B . (C . (D . nil))))
- (b) (3 points) Some variables in the following lambda expression are free and some are bound. Circle each bound variable and draw a line from the bound variable to the corresponding λ that binds it. For example, if you were asked to do this for $\lambda x. x + y$, you would circle x and draw a line from this circle to the letter λ at the beginning of the expression.

$(\lambda a. \lambda b. \lambda c. c (a b)) (\lambda a. \lambda c. c a c) a$

Answer:

(c) (6 points)

In a language that does not provide automatic storage management, a *memory leak* occurs when a program no longer has a pointer to a heap location that has been allocated to it. The reason this is called a memory leak is that a program is supposed to free locations allocated to it from the heap. But if the program no longer has a pointer to a location, it may be impossible to deallocate the location.

Suppose you are asked to design a “leak detector” tool for C. Explain how you might use a garbage collection algorithm to detect potential memory leaks. Describe briefly:

- any additional data structures you require
- whether you can identify all leaks
- two features of the C language make this task difficult

Answer: A memory leak is similar to garbage, since both involve inaccessible locations. Therefore, it is likely that garbage collection algorithms could be useful for finding memory leaks. Since mark-and-sweep searches for inaccessible locations, this would be a good algorithm to consider for a leak-detection tool. One complication is that mark-and-sweep requires an extra bit for each memory location. However, this does not need to be part of the data in memory. It is possible to store a table of mark bits separately. This would be useful since we would like the debugging tool to work with the standard compiler.

Reference counting could also be considered, but this is more complicated since it requires a change in the way code is compiled. Specifically, reference counting involves changing the count associated with each datum whenever pointers are added or removed. In addition, more memory is required by the algorithm, since a count is typically more than the single bit per datum needed for mark-and-sweep. Finally, mark-and-sweep works with circular structures but reference counting does not.

These techniques are language dependent since it is necessary to know which data stored in memory are pointers. In C, for example, it is possible to reference a location using pointer arithmetic. This makes it impossible to tell just by looking at data in memory whether a location will be used later in the program. However, the question just asks for a tool that looks for memory leaks. A debugging tool that finds some but not all memory leaks can still be very useful.

4. (7 points) Symbolic evaluation of Lisp expression

Lisp expressions can be evaluated symbolically, using β -reduction and simple calculational properties of conditional, `cond`, and arithmetic. Evaluate the following expression to a number, showing your work. The first two steps are done for you.

```
((lambda (fact)           // function with one argument
  (fact fact 2))
 (lambda (ft k)          // this lambda function is the argument
  (cond ((= k 0) 1)
        (t (* k (ft ft (- k 1)))))) )
) )

= // after substituting lambda function for both fact references
((lambda (ft k)          // lambda function with 2 arguments
  (cond ((= k 0) 1)
        (t (* k (ft ft (- k 1)))))) )
)
(lambda (ft k)          // this lambda will be the first argument
  (cond ((= k 0) 1)
        (t (* k (ft ft (- k 1)))))) )
)
2 // this is the second argument
)

= // after applying the first & second arguments and resolving cond
(* 2
  ((lambda (ft k)       // lambda function with 2 arguments
    (cond ((= k 0) 1)
          (t (* k (ft ft (- k 1)))))) )
  (lambda (ft k)       // first argument
    (cond ((= k 0) 1)
          (t (* k (ft ft (- k 1)))))) )
  )
  1 // second argument
)
)
=
```

Answer:

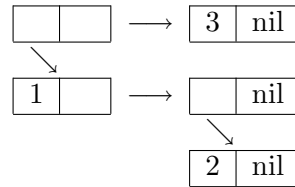
```
(* 2 (* 1
  ((lambda (ft k)
    (cond ((= k 0) 1)
          (t (* k (ft ft (- k 1)))))) )
  )
  (lambda (ft k)
    (cond ((= k 0) 1)
          (t (* k (ft ft (- k 1)))))) )
  )
  0
)
))

= (* 2 (* 1 1))

= 2
```

5. (6 points) Nested Lists in ML program

Lisp supports arbitrarily nested lists, like ((1 (2)) 3), which are represented internally by cons cells:



To support such nested lists in ML, we can define the following datatype:

```
datatype 'a lisplist = nil | atom of 'a | cons of 'a lisplist * 'a lisplist;
```

This allows us to simulate Lisp representations of nested lists:

```
(1) is cons(atom(1),nil)
(1 2) is cons(atom(1),cons(atom(2),nil))
```

Flattening a list means removing all internal nesting, while presenting the atoms in the same order. For example, flattening the list ((1 (2)) 3) yields: (1 2 3).

Below is a skeleton for a `flatten` function in ML. Fill in the missing code. As an example of how the `flatten` function should behave, `flatten(cons(cons(atom(1),nil),nil))` should produce: `cons(atom(1), nil)` That is, `((1))` is flattened to `(1)`.

To simplify your task, you can assume that a `concat` function is provided, which takes a pair of `lisplist` as arguments and concatenates them, producing a single `lisplist`. That is, `concat` applied to `(1 2)` and `(3 4)` gives `(1 2 3 4)`.

```
fun flatten (nil) =
| flatten (atom(X)) =
| flatten (cons(Y,Z)) =
```

Answer:

```
fun flatten (nil) = nil
| flatten (atom(X)) = cons(atom(X),nil)
| flatten (cons(Y,Z)) = concat(flatten(Y), flatten(Z));
```

6. (10 points) Lazy and Eager Evaluation

Haskell and ML are based on different evaluation orders. In Haskell, function arguments are not evaluated until they are needed. In contrast, a function argument is evaluated before the call is performed in ML, Lisp, C, Java, and many other languages. The Haskell evaluation order is called *lazy evaluation*; the more common order is called *eager evaluation*.

Lazy evaluation can be implemented using *thunks*, which are no-argument functions that are called when an expression needs to be evaluated. For example, a Haskell expression $f(e)$ can be compiled to code that passes a thunk for e to the function body of f . If $f(x)$ contains an expression $x+1$, for example, then the thunk for e can be called to get the value of x to add it to 1. This question asks about the following function:

```
times3(x) = x + x + x
```

- (a) (2 points) Haskell is a pure functional language. What does this imply about the three values of x in any call to `times3`?

Answer: The three occurrences of the same expression will have the same value.

- (b) (4 points) Since `times3(x)` contains three occurrences of x , the thunk for the function argument will be called three times. What trick could you use in a Haskell compiler to improve the efficiency of the compiled code for `times3`? (*Hint:* think about how futures worked in Lisp.)

Answer: The thunk stores its value the first time it is computed. Haskell creates a thunk to represent unevaluated expressions. Once a thunk's value is computed, then Haskell replaces it with the computed value. This way, each argument's value will be computed only once, or perhaps not at all. (With eager evaluation, each argument's value is computed exactly once.)

- (c) (2 points) Algol 60 call-by-name can also be implemented by thunks. Would your optimization in part (b) work in Algol 60? Explain why or why not.

Answer: This will change the behavior of programs because of side effects.

- (d) (2 points) A major reason why programs written in lazy languages can be less efficient than eager languages is the overhead associated with thunks. Give one reason why thunks reduce time efficiency and one reason why thunks reduce space efficiency.

Answer: Thunks take time to construct and evaluate, they occupy space in the heap, and they cause the garbage collector to retain other structures needed for the evaluation of the thunk.

7. (12 points) Denotational Semantics

This problem asks about a nonstandard semantics that characterizes a form of type analysis for expressions given by the following grammar:

$$e ::= 0 \mid 1 \mid \text{true} \mid \text{false} \mid x \mid e + e \mid e = e \mid \text{if } e \text{ then } e \text{ else } e \mid \text{fun } f(x:\sigma):\tau = e \text{ in } e \mid f(e)$$

In the `fun...in...` expression, which declares a function and provides a scope where the function can be called, the types σ and τ may be either *int* or *bool*. The meaning $\mathcal{V}[[e]](s)$ of an expression e depends on the state s . For the purpose of this problem, a state s is a mapping from variables to values, with function names considered as variables. In type analysis, we will use three basic type values and function types involving *int* and *bool*:

$$\text{Values} = \{ \text{int}, \text{bool}, \text{type_error}, \text{int} \rightarrow \text{int}, \text{int} \rightarrow \text{bool}, \text{bool} \rightarrow \text{int}, \text{bool} \rightarrow \text{bool} \}$$

Intuitively, $\mathcal{V}[[e]](s) = \text{int}$ means that the value of expression e is an integer (in state s) and $\mathcal{V}[[e]](s) = \text{type_error}$ means that evaluation of e may involve a type error.

Here are the semantic clauses for the first few expression forms.

$$\begin{aligned}\mathcal{V}[[0]](s) &= int \\ \mathcal{V}[[1]](s) &= int \\ \mathcal{V}[[\text{true}]](s) &= bool \\ \mathcal{V}[[\text{false}]](s) &= bool\end{aligned}$$

The value of a variable in some state is simply the value the state gives to that variable.

$$\mathcal{V}[[x]](s) = s(x)$$

For addition, the value will be *type_error* unless both operands are integers.

$$\mathcal{V}[[e_1 + e_2]](s) = \begin{cases} int & \text{if } \mathcal{V}[[e_1]](s) = int \text{ and } \mathcal{V}[[e_2]](s) = int \\ type_error & \text{otherwise} \end{cases}$$

For boolean equality, the value will be *type_error* unless both operands are integers or booleans.

$$\mathcal{V}[[e_1 = e_2]](s) = \begin{cases} bool & \text{if } \mathcal{V}[[e_1]](s) = int \text{ and } \mathcal{V}[[e_2]](s) = int \\ bool & \text{if } \mathcal{V}[[e_1]](s) = bool \text{ and } \mathcal{V}[[e_2]](s) = bool \\ type_error & \text{otherwise} \end{cases}$$

Since function declarations involve declaring the types of parameters and return values, and these types are recorded in a state, the semantics of declarations involves changing the state. Before giving the clause for function declaration, we define the notation $s[x \mapsto \sigma]$ for a state that is similar to s , except that it must map x to type σ . More precisely,

$$s[x \mapsto \sigma] = \lambda y \in Variables. \text{ if } y = x \text{ then } \sigma \text{ else } s(y)$$

Using this notation, we can define the semantics of function declarations by

$$\mathcal{V}[[\text{fun } f(x:\sigma):\tau=e_1 \text{ in } e_2]](s) = \begin{cases} \mathcal{V}[[e_2]](s[f \mapsto (\sigma \rightarrow \tau)]) & \text{if } \mathcal{V}[[e_1]](s[x \mapsto \sigma]) = \tau \\ type_error & \text{otherwise} \end{cases}$$

Similarly, the semantics of function application are

$$\mathcal{V}[[f(e_1)]](s) = \begin{cases} \tau & \text{if } s(f) = \sigma \rightarrow \tau \text{ and } \mathcal{V}[[e_1]](s) = \sigma \\ type_error & \text{otherwise} \end{cases}$$

The clause for conditional is

$$\begin{aligned}\mathcal{V}[[\text{if } e_1 \text{ then } e_2 \text{ else } e_3]](s) = \\ \begin{cases} bool & \text{if } \mathcal{V}[[e_1]](s) = \mathcal{V}[[e_2]](s) = \mathcal{V}[[e_3]](s) = bool \\ int & \text{if } \mathcal{V}[[e_1]](s) = bool \text{ and } \mathcal{V}[[e_2]](s) = \mathcal{V}[[e_3]](s) = int \\ type_error & \text{otherwise} \end{cases}\end{aligned}$$

For example, using $s_0 = \lambda y \in Variables. type_error$,

$$\mathcal{V}[[\text{if true then } 0 + 1 \text{ else } x + 1]](s_0) = type_error$$

since the expression $x + 1$ produces a type error if evaluating x produces a type error.

Question:

(12 points) Show how to calculate the meaning of the expression

$$\text{fun } f(x:\text{int}):\text{bool}=(x=0) \text{ in } (\text{if } f(0) \text{ then true else false})$$

in state $s_0 = \lambda y \in \text{Variables. type_error}$.

Answer: First, a few definitions:

$$e_1 = (x = 0)$$

$$e_2 = (\text{if } f(0) \text{ then true else false})$$

First, we evaluate the definition of the function f to make sure that its type matches the function's declared return type:

$$\begin{aligned} v_1 &= \mathcal{V}[[e_1]](s_0[x \mapsto \text{int}]) \\ &= \mathcal{V}[(x = 0)](s_0[x \mapsto \text{int}]) \\ &= \text{bool if } \mathcal{V}[[x]](s_0[x \mapsto \text{int}]) = \mathcal{V}[[0]](s_0[x \mapsto \text{int}]) = \text{int} \\ &= \text{bool if } \text{int} = \text{int} = \text{int} \\ &= \text{bool} \end{aligned}$$

We have now shown that the definition of f matches its declared types. We now move on to evaluate the meaning of $\text{if } f(0) \text{ then true else false}$ in the context where $s(f)$ is $\text{int} \rightarrow \text{bool}$. To do this, we must evaluate the conditional part of the if statement and both branches. So we move on to evaluate the meaning of the conditional, which is a function application. By definition, the meaning of a function application is the declared return type of the function, assuming that the actual parameter passed to the function is the declared input type:

$$\begin{aligned} v_2 &= \mathcal{V}[[f(0)]](s_0[f \mapsto (\text{int} \rightarrow \text{bool})]) \\ &= \text{bool if } \mathcal{V}[[0]](s_0[f \mapsto (\text{int} \rightarrow \text{bool})]) = \text{int} \\ &= \text{bool if } \text{int} = \text{int} \\ &= \text{bool} \end{aligned}$$

Thus, the meaning of the conditional is bool . Now we can evaluate the meaning of the entire if statement:

$$\begin{aligned} v_3 &= \mathcal{V}[[e_2]](s_0[f \mapsto (\text{int} \rightarrow \text{bool})]) \\ &= \mathcal{V}[[\text{if } f(0) \text{ then true else false}]](s_0[f \mapsto (\text{int} \rightarrow \text{bool})]) \\ &= \text{bool if } v_2 = \mathcal{V}[[\text{true}]](s_0[f \mapsto (\text{int} \rightarrow \text{bool})]) = \mathcal{V}[[\text{false}]](s_0[f \mapsto (\text{int} \rightarrow \text{bool})]) = \text{bool} \\ &= \text{bool if } \text{bool} = \text{bool} = \text{bool} \\ &= \text{bool} \end{aligned}$$

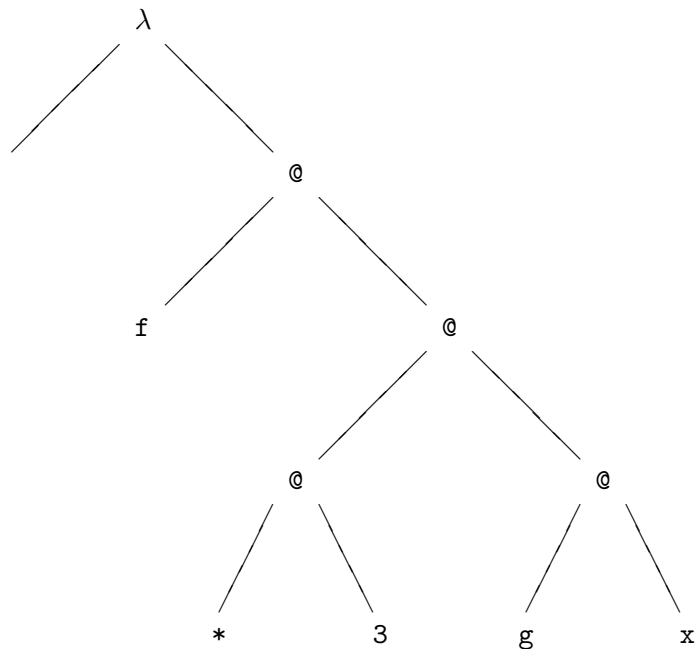
Finally, we can evaluate the entire meaning of the question:

$$\begin{aligned} v_4 &= \mathcal{V}[[\text{fun } f(x:\text{int}):\text{bool}=(x=0) \text{ in } e_2]](s_0) = \\ &= \mathcal{V}[[e_2]](s_0[f \mapsto (\text{int} \rightarrow \text{bool})]) \text{ if } \mathcal{V}[[e_1]](s_0[x \mapsto \text{int}]) = \text{bool} \\ &= v_3 \text{ if } v_1 = \text{bool} \\ &= \text{bool if } \text{bool} = \text{bool} \\ &= \text{bool} \end{aligned}$$

8. (10 points) Type inference on parse graph

Use the parse graph to perform type inference for the following ML function:

```
fun arith (f, g, x) = f(3 * g(x));
```



Answer: $f : int \rightarrow s$
 $g : t \rightarrow int$
 $x : t$
 $arith : (int \rightarrow s) * (t \rightarrow int) * t \rightarrow s$

9. (16 points) Activation Records and Scope

This question asks about memory management in the evaluation of the following statically-scoped ML expression.

```
fun sum_compose(f, g) =
  let val x : int ref = ref 0
  in
    fn (y) => (x := (!x) + f(g(y))); !x
  end;
val x = 2;
fun add_two(y) = y + x;
fun square(x) = x * x;
val comp = sum_compose(add_two, square);
comp(2);
```

- (a) (13 points) Fill in the missing information in the following depiction of the run-time stack at the time when the call on line 5 of this code fragment is made. Remember that function values are represented by closures, and that a closure is a pair consisting of an

environment (pointer to an activation record) and compiled code. Remember also that in ML, function arguments are evaluated before the function is called.

In this drawing, a bullet (\bullet) indicates that a pointer should be drawn from this slot to the appropriate closure, compiled code or heap cell. Since the pointers to activation records cross and could become difficult to read, each activation record is numbered at the far left. In each activation record, place the number of the activation record of the statically enclosing scope in the slot labelled “access link.” Closures and heap cells are labeled with capital letters and compiled code with lowercase letters. Use activation record numbers for the environment pointer part of each closure pair, and write the lowercase letter corresponding to the compiled code for the second part of the pair. Write the values of local variables and function parameters in the activation records. The first two activation records are done for you.

<i>Activation Records</i>	<i>Closures and Cells</i>	<i>Compiled Code</i>												
<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 15%; text-align: center;">(1)</td> <td style="width: 40%;">access link</td> <td style="width: 45%; text-align: center;">(0)</td> </tr> <tr> <td></td> <td>sum_compose</td> <td style="text-align: center;">A \bullet</td> </tr> </table>	(1)	access link	(0)		sum_compose	A \bullet	(A)((1), i \bullet)	(i)sum_compose						
(1)	access link	(0)												
	sum_compose	A \bullet												
<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 15%; text-align: center;">(2)</td> <td style="width: 40%;">access link</td> <td style="width: 45%; text-align: center;">(1)</td> </tr> <tr> <td></td> <td>x</td> <td style="text-align: center;">2</td> </tr> </table>	(2)	access link	(1)		x	2		(j)add_two						
(2)	access link	(1)												
	x	2												
<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 15%; text-align: center;">(3)</td> <td style="width: 40%;">access link</td> <td style="width: 45%; text-align: center;">()</td> </tr> <tr> <td></td> <td>add_two</td> <td style="text-align: center;">\bullet</td> </tr> </table>	(3)	access link	()		add_two	\bullet	(B)((), \bullet)	(k)square						
(3)	access link	()												
	add_two	\bullet												
<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 15%; text-align: center;">(4)</td> <td style="width: 40%;">access link</td> <td style="width: 45%; text-align: center;">()</td> </tr> <tr> <td></td> <td>square</td> <td style="text-align: center;">\bullet</td> </tr> </table>	(4)	access link	()		square	\bullet	(C)((), \bullet)	(l) $\lambda y \dots$						
(4)	access link	()												
	square	\bullet												
<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 15%; text-align: center;">(5) comp = (...)</td> <td style="width: 40%;">access link</td> <td style="width: 45%; text-align: center;">()</td> </tr> <tr> <td></td> <td>comp</td> <td style="text-align: center;">\bullet</td> </tr> </table>	(5) comp = (...)	access link	()		comp	\bullet	(D)((), \bullet)							
(5) comp = (...)	access link	()												
	comp	\bullet												
<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 15%; text-align: center;">(6)</td> <td style="width: 40%;">access link</td> <td style="width: 45%; text-align: center;">()</td> </tr> <tr> <td></td> <td>f</td> <td style="text-align: center;">\bullet</td> </tr> <tr> <td></td> <td>g</td> <td style="text-align: center;">\bullet</td> </tr> <tr> <td></td> <td>x</td> <td style="text-align: center;">\bullet</td> </tr> </table>	(6)	access link	()		f	\bullet		g	\bullet		x	\bullet	(E) 	
(6)	access link	()												
	f	\bullet												
	g	\bullet												
	x	\bullet												
<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 15%; text-align: center;">(7) comp(2)</td> <td style="width: 40%;">access link</td> <td style="width: 45%; text-align: center;">()</td> </tr> <tr> <td></td> <td>y</td> <td></td> </tr> </table>	(7) comp(2)	access link	()		y									
(7) comp(2)	access link	()												
	y													

Answer:

	<i>Activation Records</i>	<i>Closures and Cells</i>	<i>Compiled Code</i>
(1)	access link (0) sum_compose A •	(A)⟨(1), i •⟩	(i)code for sum_compose
(2)	access link (1) x 2		(j)code for add_two
(3)	access link (2) add_two B •	(B)⟨(3), j •⟩	(k)code for square
(4)	access link (3) square C •	(C)⟨(4), k •⟩	(l)code for λy...
(5) comp = (...)	access link (4) comp D •	(D)⟨(6), l •⟩	
(6)	access link (1) f B • g C • x E •	(E) 6	
(7) comp(2)	access link (6) y 2		

Note: the question was ambiguous as to whether the stack should be shown at the time of calling `comp(2)` or just after `comp(2)` has finished. Therefore, putting 0 in the heap cell will also get full credit.

- (b) (3 points) What is the value of the expression `comp(2)`? Why?

Answer: The value is 6. Essentially the function `sum_compose` returns the sum of successive calls to a composed function. First `sum_compose`, `add_two`, and `square` are defined. `sum_compose` is called with `add_two` and `square` as parameters. Inside `sum_compose`, an integer reference cell named `x` is created on the heap and initialized to 0. The call to `sum_compose` returns a function that takes a parameter `y`, applies the composed function to `y`, sums the result with `x` and returns `x`. `comp` now points to a closure representing this function. `comp(2)` then is $2^2 + 2 + 0 = 6$. The reference cell `x` is set to 6.

10. (8 points) Implementing Exceptions

One way of implementing exceptions is to make a table mapping exception names to code for handlers, for each scope, and store this table on the run-time stack. This has little performance impact at run-time, unless an exception is raised, since the tables can be determined at compile time. However, when an exception is raised, there is some cost. Specifically, if the current activation record contains a handler, control is transferred to this handler. If not, then the exception will have to be “re-raised” in another scope.

- (a) (2 points) If an exception is raised and there is no handler in the current scope, which pointer in the activation record should be used to find the next scope?

Answer: previous record on stack, reached through control pointer

- (b) (2 points) Can the compiler determine the number of pointers to follow, for a given exception and scope, at compile time? Explain why or why not.

Answer: No, depends on calling sequence

- (c) (2 points) Optimizing compilers often change the order of instructions for various reasons. Why do languages with exceptions make this kind of optimization more difficult?

Answer: Jump could prevent an instruction from being executed

- (d) (2 points) What information would a compiler like to know, at compile time, about a given expression such as a function call, in order to reorder instructions?

Answer: Whether function can raise/throw exception, and if so, which ones