
This is a 75 minute, CLOSED notes, books, etc. exam.
ASK if anything is not clear.
WORK INDIVIDUALLY.
Strategy: Scan the entire exam first. Work on the easier ones before the harder ones. Don't waste too much time on any one problem. Show all work on the space provided. Write your name on each page. Check to make sure you have 6 pages.

Question	Points	Score
1	5	
2	5	
3	5	
4	5	
5	5	
6	20	
7	10	
8	10	
9	20	
10	15	
Total:	100	

1. (5 points) Select **all** of the following true statements about Haskell.
   A. Haskell is a *purely functional* language.
   B. One of the unusual aspects of Haskell is that it uses *eager evaluation*; for instance, with an `if` expression, it evaluates both branches before testing the condition.
   C. Haskell uses *function currying*, meaning that arguments are not evaluated until they are used.
   D. Haskell is a dynamically typed language.
   E. All functions in Haskell are curried by default.

2. (5 points) Select **all** of the following true statements about functors, applicative functors, and monads.
   A. The usual analogy for a functor is a "box", though the analogy is perhaps less fitting for cases like IO.
   B. A monad is a "box" containing a function.
   C. The `fmap` function maps a function over a functor.
   D. The `do` syntax is syntactic sugar for chaining calls to the "bind" (`>>=`) operator.
   E. Some examples of monads in Haskell include `Maybe`, `IO`, and lists.

3. (5 points) Select **all** of the following true statements about formal semantics.
   A. Small-step operational semantics evaluate an expression to a value in one step.
   B. Big-step operational semantics, sometimes called "natural semantics" are often easier to translate into programming logic.
   C. With operational semantics, there is no way to represent "state", such as mutable variables.
   D. An advantage of small-step operational semantics over big-step semantics is that it can be used to reason about intermediary state, whereas big-step semantics can only reason about the final result of the evaluation.
   E. Operational semantics are often unnecessary, since it is obvious what most language constructs should do.

4. (5 points) Select **all** of the following true statements about functional programming.
   A. Java is a functional programming language, since it supports recursion.
   B. A *higher-order* function is a function that takes another function as a parameter or returns a function as its return value.
   C. Functional solutions generally rely on iteration rather than recursion.
   D. Compilers can convert *tail recursive* code to more efficient iterative solutions.
   E. *Purely functional* languages like Haskell allow programmers to more easily write code without side effects.

5. (5 points) Select **all** of the following true statements about types and kinds in Haskell.
   A. A kind can be thought of as "the type of a type".
   B. An *abstract data type* in Haskell is similar to an interface in Java, except that it may specify functionality.
   C. The type of a function in Haskell is specified by an "arrow" (`->`), indicating the type of the input and the type of the output.
   D. In Haskell, `null` is a special value (similar to `null` in Java) that matches any type, and therefore will always typecheck as a return value.
   E. A *typeclass* is a type formed by combining other types.
   For example: `data Tree = Empty | Node Tree Tree`

6. (20 points) Consider the following language:

$e ::=$		*Expressions*
	`true`	true constant
	`false`	false constant
	`if` e `then` e `else` e	conditional expressions
	$e \ + \ e$	addition
$v ::=$		*Values*
	`true`	true constant
	`false`	false constant

Convert the following small-step operational semantics to use contexts instead of evaluation order rules.

[SS-IF-ORDER] $$\frac{e_1 \to e_1'}{\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \to \texttt{if } e_1' \texttt{ then } e_2 \texttt{ else } e_3}$$

[SS-IF-TRUE] $$\frac{}{\texttt{if true then } e_2 \texttt{ else } e_3 \to e_2}$$

[SS-IF-FALSE] $$\frac{}{\texttt{if false then } e_2 \texttt{ else } e_3 \to e_3}$$

[SS-PLUS-ORDER1] $$\frac{e_1 \to e_1'}{e_1 \ + \ e_2 \to e_1' \ + \ e_2}$$

[SS-PLUS-ORDER2] $$\frac{e_2 \to e_2'}{v_1 \ + \ e_2 \to v_1 \ + \ e_2'}$$

[SS-PLUS] $$\frac{v_3 \ = \ v_1 \ + \ v_2}{v_1 \ + \ v_2 \to v_3}$$

7. (10 points) Write a Haskell function called `makeListOfAdders` that takes a list of Integers and returns a list of adders (i.e. functions that add a number to their input). **Include the proper type signature.**

```
-- Sample usage
adders = makeListOfAdders [1,2]
(head adders) 10          -- results in 11
(head (tail adders)) 10   -- results in 12
```

8. (10 points) Implement the following higher-order functions. You may *not* use `map` or any of Haskell's built-in fold functions.

(a)
```
-- Usage:  myMap (+1) [1,2] ---------> [2,3]
myMap ::  (a -> b) -> [a] -> [b]
```

(b)
```
-- Usage:  myFoldLeft (+) 0 [1,3,5] ---------> 9
myFoldLeft ::  (a -> b -> a) -> a -> [b] -> a
```

9. (20 points) Consider the following Haskell code:

```
x -: f = f x

myadd x y = y + x
mysub x y = y - x
mymul x y = y * x
mydiv d = (\n -> case d of
    0 -> error "div by zero"
    d -> div n d)

-- Evaluates to 4
10 -: mydiv 5 -: myadd 4 -: mysub 2

-- Error
10 -: mydiv 5 -: myadd 4 -: mysub 2 -: mydiv 0 -: mymul 10
```

(a) Rewrite this program to return `Maybe` values, where `Nothing` is returned instead of an error. Use the bind operator (`>>=`) instead of `-:`.

(b) **Using `do` syntax**, write the function `foo`, which takes a parameter `x` and

1. Sets `y` to x divided by 60 (using `mydiv`; pay attention to the order of arguments).
2. Sets `z` to 9 plus 3 (using `myadd`).
3. Returns the result of multiplying `y` and `z` (using `mymul`).

10. (15 points) Consider the following language and big-step operational semantics:

$e ::=$		*Expressions*
	0	zero
	$\texttt{succ}\ e$	successor function
	$\texttt{pred}\ e$	predecessor function
	$\texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$	conditional expressions
$v ::=$		*Values*
	0	zero
	$\texttt{succ}\ v$	number

$$[\text{SUCC}]\ \frac{e \Downarrow v}{\texttt{succ}\ e \Downarrow \texttt{succ}\ v} \qquad [\text{ZERO}]\ \frac{}{0 \Downarrow 0}$$

$$[\text{PRED}]\ \frac{e \Downarrow \texttt{succ}\ v}{\texttt{pred}\ e \Downarrow v} \qquad [\text{IF-TRUE}]\ \frac{e_1 \Downarrow \texttt{succ}\ 0 \quad e_2 \Downarrow v}{\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow v}$$

$$[\text{PRED-ZERO}]\ \frac{e \Downarrow 0}{\texttt{pred}\ e \Downarrow 0} \qquad [\text{IF-FALSE}]\ \frac{e_1 \Downarrow 0 \qquad e_3 \Downarrow v}{\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow v}$$

(a) What is the result of evaluating

```
if (if (pred succ 0) then (succ succ 0)
                     else (pred succ succ 0))
   then (succ succ 0)
   else 0
```

(b) What is the result of evaluating

```
if (succ succ 0)
   then (succ 0)
   else (pred succ 0)
```