Fall, 2003 Name: _____

Com S 541 — Programming Languages 1

Test on Functional Languages, Haskell, OO Languages, and Smalltalk

Special Directions for this Test

This test has 8 questions and pages numbered 1 through 10.
This test is open book and notes.
Before you begin, please take a moment to look over the entire test so that you can budget your time.
When you write Haskell code on this test, you may use anything in the Haskell Prelude without writing it in your test. You are encouraged to define functions not specifically asked for if they are useful to your programming; if they are not in the Prelude, write them into your test.
When you write Smalltalk code below on this test, you may use anything in Squeak without writing it in your test; however, please tell us about anything you use that might be a bit obscure. You are encouraged to define methods and classes not specifically asked for if they are useful to your programming; if they are not in Squeak, write them into your test. For Smalltalk code, please use the shortened form used in Goldberg and Robson's book when writing a class and its methods. The following is an example.

```
class name:                Counter
superclass:                Object
instance variable names:   value

"class methods"
new
  ^super new initialize

"instance methods"
initialize
  value := 0

increment
  "Increment the value of this counter by 1"
  value := value + 1

value
  "Return the value of this counter"
  ^value
```

As a reminder of our specification notation, the following specifies the type `Counter` above.

new *self: Counter class → c: Counter*
Ensures: *c* is freshly allocated and the value of *c* is 0.

initialize *self: Counter*
Modifiable: the value of *self*.
Effect: make the value of **post**(*self*) be 0.

increment *self: Counter*
Modifiable: the value of *self*.
Effect: make the value of **post**(*self*) be the value of **pre**(*self*) + 1.

value *self: Counter → i: Integer*
Ensures: *i* is the value of *self*.

1. This is a question about expressive power. Please give brief answers.

   (a) (5 points) Why are syntactic sugars useful in programming language design and implementation?

   (b) (5 points) In exploring the Squeak Smalltalk system, we discovered that calls to the `whileTrue:` method of `BlockContext` are usually "compiled in-line," instead of being processed normally by a call to the `whileTrue:` method's code. Is `whileTrue:` a syntactic sugar in Squeak Smalltalk? Is the compilation of calls to `whileTrue:` an eliminable feature of Squeak?

2. (5 points) In Haskell, write a function

```
> associated :: (Eq a) => a -> [(a,b)] -> [b]
```

   such that `associated x pairs` is the list, in order, of the second elements of pairs in `pairs`, whose first element is equal to the argument `x`.

   For example:

```
associated 3 ([]::[(Integer,Float)]) = []
associated 3 [(3,4), (5,7), (3,6), (9,3)] = [4, 6]
associated 2 [(1,'a'), (3,'c'), (2,'b'), (4,'d')] = ['b']
associated 'c' (zip ['c', 'c' ..] [1, 2 ..]) = [1, 2 ..]
```

3. (10 points) In Haskell, write a function

```
> mapN :: ([a] -> b) -> [[a]] -> [b]
```

that takes a function, `f`, and a finite list of lists of the function's argument type, `arglists`, and returns a list of the results of applying `f` to a the corresponding elements of each list in `arglists`. That is, the i^{th} element of the result list is the application of the `f` to the list of the i^{th} elements of `arglists` (in order). The length of the result list should be the length of shortest of the argument lists. For example,

```
mapN null ([]::[[Integer]]) = []
mapN (\(x:y:z:_) -> (x,y,z)) [[], [], [10], [], [1 ..]] = []
mapN (\(x:y:z:_) -> (x,y,z)) [[1,2,3],[4,5,6],[7,8,9,10]] = [(1,4,7),(2,5,8),(3,6,9)]
mapN null [[1,2,3], [4,5,6], [7,8,9,10]] = [False, False, False]
mapN sum [[1,2 ..], [7,8,9,10], [4,5 ..], [0,0 ..]] = [12,15,18,21]
mapN product [[1,1,1,1,1], [2, 4 ..], [3, 6 ..], [4, 8 ..]] = [24,192,648,1536,3000]
```

4. (25 points) Consider the following grammar for types in a simple programming language with variant record types.

```
> data TypeExpr = IntType | BoolType
>               | VariantType TypedLabels | TypeExpr :-> TypeExpr
>                deriving (Eq, Show)
> type TypedLabels = [LabelTypeBinding]
> type LabelTypeBinding = (Label, TypeExpr)
> type Label = String
```

Your task is to program the subtyping test for this grammar.

```
> (<:) :: TypeExpr -> TypeExpr -> Bool
```

Assuming the variant records are immutable, a variant record type `VariantType`$[l_1{:}T_1, \ldots, l_n{:}T_n]$ `<:` `VariantType`$[l_1{:}T'_1, \ldots, l_n{:}T'_n, l_{n+1}{:}T'_{n+1}, \ldots, l_m{:}T'_m]$ if $n \leq m$ and for each $1 \leq i \leq n$, T_i `<:` T'_i. Although the above presentation assumes that the label-type bindings are conveniently ordered, we consider the type `TypedLabels` to be unordered. For example, the types `VariantType[("i",Int), ("b",Bool)]` and `VariantType[("b",Bool), ("i",Int)]` are the same type. We use the usual contravariant rule for subtyping of function types; that is, (T`:->`S) `<:` (T'`:->`S') if T' `<:` T and S `<:` S'. Every type is considered to be a subtype of itself, but there are no other subtyping relationships (we ignore transitivity).

There are examples on the next page.

The following are examples for the problem on the previous page.

```
IntType <: VariantType [] = False
IntType <: BoolType = False
IntType <: IntType = True
BoolType <: BoolType = True
VariantType [] <: VariantType [("i",IntType)] = True
VariantType [("i",IntType)] <: VariantType [] = False
VariantType [("b",BoolType), ("i",IntType)]
   <: VariantType [("i",IntType), ("b",BoolType)] = True
VariantType [("b",BoolType), ("i",IntType)]
   <: VariantType [("i",IntType), ("norm", IntType),("b",BoolType)] = True
(IntType :-> VariantType [("norm", IntType)])
   <: (IntType :-> VariantType [("norm", IntType), ("ex", IntType)]) = True
(IntType :-> VariantType [("norm", (IntType :-> BoolType))])
   <: (IntType :-> VariantType [("norm", (IntType :-> BoolType)), ("ex", IntType)]) = True
VariantType [("f", (VariantType [("i",IntType)]
                    :-> VariantType [("norm", (IntType :-> BoolType))]))]
   <: VariantType
      [("b", BoolType),
       ("f", (VariantType []
              :-> VariantType [("norm", (IntType :-> BoolType)), ("ex", IntType)]))] = True
```

5. (10 points) Using the equational rules for the lambda calculus, formally derive the normal form (if any) for the following.

$$(\lambda f \,.\, \lambda x \,.\, f\ (f\ x))\ (\lambda t \,.\, t)$$

If this term does not have a normal form, show using the reduction rules why it does not.

6. (15 points) Consider the specification of the class JoinPoint below. Abstractly, a join point has three parts: a receiver, which is an object identity, a method name, which is a symbol, and an array of argument object identities.

receiver:sent:withArgs: *self: JoinPoint class*, *receiver: Object*, *mn: Symbol*, *args: Array* → *jp: JoinPoint*

Ensures: **obj**(*jp*) is freshly allocated, and the receiver of **post**(*jp*) is *receiver*, the message name of **post**(*jp*) is *mn*, and the arguments of **post**(*jp*) are *args*.

receiver:sent: *self: JoinPoint class*, *receiver: Object*, *mn: Symbol* → *jp: JoinPoint*

Ensures: **obj**(*jp*) is freshly allocated, and the receiver of **post**(*jp*) is *receiver*, the message name of **post**(*jp*) is *mn*, and the arguments of **post**(*jp*) is an empty array.

receiver *self: JoinPoint* → *res: Object*

Ensures: **obj**(*res*) is the *receiver* of *self*.

methodName *self: JoinPoint* → *res: Symbol*

Ensures: *res* is the *methodName* of *self*.

arguments *self: JoinPoint* → *res: Array*

Ensures: *res* is an array of the object identities of the arguments of *self*.

printOn: *self: JoinPoint*, *str: WriteStream*

Modifiable: *str:*

Effect: **post**(*str*) is changed to be **pre**(*str*) with a string representing *self* written to its end.

The following are some examples, where after each chunk of code the output follows the "==>".

```
| c |
c := Counter new.
JoinPoint receiver: c sent: #initialize withArgs: #()
 ==> a JoinPoint[a Counter(0) performing: #initialize withArguments: #()]

| c |
c := Counter new.
JoinPoint receiver: c sent: #increment
 ==> a JoinPoint[a Counter(0) performing: #increment withArguments: #()]

| c jp |
c := Counter new.
jp := JoinPoint receiver: c sent: #increment.
jp receiver.
 ==> a Counter(0)

| c jp |
c := Counter new.
jp := JoinPoint receiver: c sent: #increment.
jp methodName.
 ==> #increment

| c jp args |
c := Counter new.
jp := JoinPoint receiver: c sent: #initialize withArgs: #(1 2).
args := jp arguments.
args at: 1 put: 7.
jp arguments
 ==> #(1 2)
```

There is space for your answer on the next page.

```
class name:               JoinPoint
superclass:               ------------------
instance variable names:  ------------------

"class methods"









"instance methods"
```

7. (20 points) Consider the specification of the class AdvisedCounter below. Assume that AdvisedCounter is a subclass of the class `Counter` defined on the first page of this test. An AdvisedCounter is like a Counter, but it also contains a set of blocks, called its "advice", which it calls before running the inherited code for the method increment.

   addBeforeAdvice: *self: AdvisedCounter*, *blk: Block*
   Requires: *blk* takes one argument, of type JoinPoint.
   Modifiable: the advice of *self*
   Ensures: **post**(*self*)'s advice is **pre**(*self*)'s advice plus *blk*.

   increment *self: Counter*
   Modifiable: the value of *self*.
   Effect: send the message value: with a JoinPoint representing this method invocation (i.e., `JoinPoint receiver:` *self* `sent: #increment`) to each of the blocks in **pre**(*self*)'s advice, and then make the value of **post**(*self*) be the value of **pre**(*self*) + 1.

   The following is an example.

```
| c oldcounts |
c := AdvisedCounter new.
oldcounts := OrderedCollection new.
c addBeforeAdvice: [ :jp | jp methodName == #increment
                                ifTrue: [ oldcounts add: jp receiver value ] ].
c addBeforeAdvice: [ :jp | (jp arguments size > 0 or: [jp receiver ~~ c])
                                ifTrue: [ self error: 'shouldn''t happen' ] ].
c increment.
c increment.
c increment.
oldcounts.
 ==> an OrderedCollection(0 1 2)
```

   Implement the specification of AdvisedCounter by writing a subclass of Counter in Smalltalk below. Write the minimal amount of code; that is, don't write out code for methods that AdvisedCounter can inherit from `Counter`.

   There is space for your answer on the next page.

```
class name:               AdvisedCounter
superclass:               Counter
instance variable names:  ___________________

"class methods"




"instance methods"
```

8. (5 points) Is AdvisedCounter a subtype of Counter?