Fall, 1999 Name: ________________________________

Com S 541 — Programming Languages 1

Test on the OO Paradigm, Smalltalk, and a bit of Logic Programming and λProlog

Special Directions for this Test

This test has 6 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.

For Smalltalk code, please use the shortened form used in lecture when writing a class and its methods. The following is an example.

```
class: Counter          subclass of: Object
instance variables: count

"class methods"
new
  ^super new initialize

"instance methods"
initialize
  count := 0

increment
  count := count + 1

value
  ^count
```

When you write Smalltalk code on this test, you may use anything in Smalltalk Express 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 Smalltalk Express, write them into your test.

Similarly, with λProlog, you may use the built-in facilities on the language. You are also encouraged to write helping modules and relations if they are useful to your programming; if they are not built-in, please write them into your test.

1. (5 points) Consider the following λProlog module.

```
module movies.

type  acted   string -> string -> o.
type directed string -> string -> o.

directed "Woody Allen"    "Hannah and Her Sisters".
directed "Billy Wilder"   "Some Like it Hot".
directed "Gene Kelly"     "Singin' in the Rain".
directed "Stanley Donen"  "Singin' in the Rain".

acted "Diane Keaton"      "Manhattan".
acted "Woody Allen"       "Manhattan".
acted "Gene Kelly"        "Singin' in the Rain".
acted "Debbie Reynolds"   "Singin' in the Rain".
acted "Marilyn Monroe"    "Some Like it Hot".
acted "Jack Lemon"        "Some Like it Hot".
```

Fill in the results in the queries below.

```
Terzo> #query movies.
?- directed Director "Singin' in the Rain".

Director = ____________________________ ;

Director = ____________________________

?- acted "Marilyn Monroe" Movie, directed Director Movie.

Movie = ____________________________
Director = ____________________________
```

2. (10 points) In λProlog, write a module that imports `movies` and adds an additional relation `actordirector` such that `(actordirector Name)` is satisfied just when `Name` is both an actor and a director. Although Woody Allen and Gene Kelly are the only examples in the movies module, your relation should work regardless of the particular facts in that module.

3. (15 points) In Smalltalk, write a class `CompactDisc` as a subclass of `Object`. It should have methods illustrated by the following transcript. This transcript shows several results of a modified Smalltalk "show it" command. (This modification was done to the `printIt` method in the `TextPane` class.) Each printed result is placed in a comment and preceded by an arrow `==>`; it is the result of doing "show it" on all the preceding part of the workspace.

```
| bj myFavs |
bj := CompactDisc new artist: 'Billy Joel'.
bj title: 'Piano Man'.
bj.                       "==> CD(Piano Man by Billy Joel)"
bj title.                 "==> 'Piano Man'"
bj artist.                "==> 'Billy Joel'"
myFavs := Set new.
myFavs add: 1; add: 5.
myFavs.                   "==> Set(5 1 )"
bj favorites: myFavs.
bj favorites.             "==> Set(5 1 )"
myFavs add: 7.
myFavs.                   "==> Set(7 5 1 )"
bj favorites.             "==> Set(5 1 )"
bj addFavorites: 3; removeFavorites: 5.
bj favorites.             "==> Set(3 1 )"
myFavs.                   "==> Set(7 5 1 )"
```

In your code be sure to avoid exposing the representation. (You are supposed to know what that means.) Include all necessary methods, and any needed helping methods; you may want to double check to make sure you have them all when you're done.

4. (10 points) In this problem, you will implement in Smalltalk a class `Thunk`, which is like a lazy expression in Haskell. The class `Thunk` has the following class method.

for: *self: Thunk class, aBlock: HomeContext* $\rightarrow$ *t: Thunk*
  Requires: *aBlock* is a zero-argument block.
  Ensures: *t* is a *Thunk* that will return the first value of *aBlock* when asked.

Abstractly, an instance of `Thunk` contains a block, a value (some object), and a Boolean that tells whether the block has been evaluated yet. The block is evaluated by sending it the **`value`** message at most once, and the object returned is stored in the `Thunk`'s value to be used in further queries about the value. The `Thunk`'s value can be obtained by calling the following instance method.

value *self: Thunk* $\rightarrow$ *o: Object*
  Modifies at most: *self*'s value and boolean components, and the objects modified by *self*'s block.
  Ensures: *o* is the *self*'s value.

For example, when the following code is executed,

```
| myThunk |
myThunk := Thunk for:  [ MessageBox confirm: 'thunks are fun'].
(1 to: 3) do: [ :i |
    i printOn: Transcript.
    Transcript space.
    myThunk value printOn: Transcript.
    Transcript space ].
Transcript cr.
```

it can produce on the transcript the output `1 true 2 true 3 true` or `1 false 2 false 3 false`, depending on the answer selected from the `MessageBox`, but the `MessageBox` only appears once, so it cannot produce a mix of true and false outputs.

Fill in your code below.

```
class:  Thunk subclass of:  Object
instance variables: ______________________________

"class methods"



"instance methods"
```

5. In this problem you will investigate immutable lazy lists in Smalltalk. That is, you will design and implement a set of classes including: `LazyEmptyList`, `LazyTake`, and `LazyInfiniteSeries`. These should work as shown in the following transcript, where, at each step where a result can be shown, the result of doing "show it" is indicated by a comment with an arrow in it.

```
| nats |
nats := LazyInfiniteSeries from: 0 by: 1.
nats isKindOf: Collection. "==> true"
nats head.                 "==> 0"
nats tail head.            "==> 1"
(nats take: 0) tail.       "==> LazyEmptyList()"
(nats take: 1) head.       "==> 0"
(nats take: 1) tail.       "==> LazyEmptyList()"
nats take: 7.              "==> LazyTake(0 1 2 3 4 5 6 )"
(nats drop: 7) take: 10.   "==> LazyTake(7 8 9 10 11 12 13 14 15 16 )"
(nats select: [ :n | n odd ]) take: 10.
                           "==> LazyTake(1 3 5 7 9 11 13 15 17 19 )"
```

We are being a bit vague about the specification of lazy lists, as part of this problem is for you to design the appropriate behavior, as long as it's lazy. We will say that it is okay for `LazyEmptyList` to give errors for most messages, and that `select:` will have to produce lazy lists. Also, you don't have to worry about any methods other than the ones mentioned in the above transcript and `do:`. However, the rest of the design is up to you; you are expected to make reasonable decisions, and to know what makes a "good" design. That's part of the problem, so don't ask us for more guidance.

(a) (10 points) How could you organize the classes necessary to implement the behavior shown in the above transcript (and any other classes you may want) to maximize (i) reuse of methods and (ii) consistency of protocol? By consistency of protocol we mean, for example, that all lazy lists should respond to `select:`. Show the organization by drawing a class hierarchy diagram, placing superclasses above subclasses, and drawing a line from each subclass to its superclass. All your classes should be subclasses of `Collection`, perhaps indirectly, so put `Collection` at the top of your diagram.

This continues the problem on the previous page.

(b) (40 points) In Smalltalk, write the necessary code for the classes you designed. Note in comments any assumptions you needed to make.

You may use the class `Thunk` in your implementation if you wish; you may assume it works according to the previous problem's specification (no matter what your code does). You do not need to give a specification for this answer.

Hints: it is quite helpful to use recursion and to create new objects, so don't hesitate to do so if that seems useful. Don't worry about a superclass method creating an instance of some subclass. Remember that you may, if you wish, write some helping classes that are not explicitly called for in the problem.

There is more space on the next 3 pages for your answer also.

"(space for your answer)"

"(space for your answer)"

"(space for your answer)"

6. (10 points) You have been hired as a consultant for "e-games," a new start-up company that plans to hire people who know lots about computer and video games to write computer and video games for sale on the retail market. Your task is to design domain-specific languages or frameworks to help make programming easier for these people, many of whom do not know how to program. The Chief Technical Officer of e-games is a former 541 student, and is only willing to accept a solution based on either Haskell or Smalltalk. Compare and contrast the suitability of Haskell and Smalltalk for this task. Illustrate your answer with experience from your group's projects.