CS 242 Final Exam

This is a closed-book three-hour exam. You are allowed to use 2 single-sided pages of notes. 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.

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)

<table>
<thead>
<tr>
<th>Prob</th>
<th># 1</th>
<th># 2</th>
<th># 3</th>
<th># 4</th>
<th># 5</th>
<th># 6</th>
<th># 7</th>
<th># 8</th>
<th># 9</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Score</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max</td>
<td>5</td>
<td>10</td>
<td>6</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>16</td>
<td>15</td>
<td>12</td>
<td>100</td>
</tr>
</tbody>
</table>
1. (5 points) ................................................................. True or False

Mark each statement true or false, as appropriate. Incorrect answers will be scored negatively, so that the expected score from a random guess is zero. Blank answers will not be scored negatively.

____ (a) The lambda term \((\lambda x. x + 1) ((\lambda y. y * 2) 3)\) reduces to \(3 * 2 + 1\).

____ (b) Under static scope, the access link in an activation record is used to locate activation records that contain global variables.

____ (c) A C++ vtable begins with a pointer to the vtable of the superclass so that inherited methods can be called more efficiently.

____ (d) If the type function \(f\) is a subtype of the type of function \(g\), then the argument type of \(f\) must be a subtype of the argument type of \(g\).

____ (e) If the type function \(f\) is a subtype of the type of function \(g\), then the return type of \(f\) must be a subtype of the return type of \(g\).
2. (10 points) ...................................................... Short Answer

Answer each question in a few words or phrases.

(a) (3 points) It is not always necessary to implement all of the functions in a Haskell typeclass. For example, the minimal complete definition of the Monad typeclass is:

\[
(\gg=) \, :: \, m \, a \rightarrow (a \rightarrow m \, b) \rightarrow m \, b \\
\text{return} \, :: \, a \rightarrow m \, a
\]

This is because the third function of the typeclass can be implemented in terms of the first two:

\[
(\gg) \, :: \, m \, a \rightarrow m \, b \rightarrow m \, b \\
x \gg y = x \gg= \_ \rightarrow y
\]

An alternate minimal complete definition is based on two different functions:

\[
\text{map} \, :: \, (a \rightarrow b) \rightarrow m \, a \rightarrow m \, b \\
\text{join} \, :: \, m \, (m \, a) \rightarrow m \, a
\]

Show how these two can be used to implement \((\gg=)\):

\[
(\gg=) \, :: \, m \, a \rightarrow (a \rightarrow m \, b) \rightarrow m \, b \\
x \gg= y = \text{______________________________}
\]

(b) (2 points) The following snippet of Java code produces an error. Explain which line produces the error, and whether the compiler or the run-time system produces the error and why. Be specific.

1: Number[] numbersArray = new Float[2];
2: numbersArray[0] = new Float(0);
3: numbersArray[1] = new Long(1);
4: System.out.println("Sum: " +
   (numbersArray[0] + numbersArray[1]));

Which line of code and why?

Compile-time or run-time?
(c) (2 points) The following snippet of Java code produces an error. Explain which line produces the error, and whether the compiler or the run-time system produces the error and why. Be specific.

1. List<Number> numbersList = new ArrayList<Float>(2);
2. numbersList.add(new Float(0));
3. numbersList.add(new Long(1));
4. System.out.println("Sum: " +
   (numbersList.get(0) + numbersList.get(1)));

Which line of code and why?

Compile-time or run-time?

(d) (3 points) In C++, objects may be stored in activation records that are deallocated when an exception is thrown. When an activation record is deallocated, the destructors of all of the objects stored on the stack are called. Your friend says that it is unnecessary to call the destructors because the memory allocated to stack objects is reclaimed when the activation records are deallocated. Explain why calling destructors of objects on the stack is a useful language mechanism, in spite of your friend’s observation.
3. (6 points) .................................................. Type Inference

The following parse tree represents the Haskell code:

\[ f \ (g, \ h) \ x = \text{let } y = g \ x \ \text{in } \text{if } y \ \text{then } x \ \text{else } h \ x \]

(a) (4 points) List the remaining constraints that are derivable for the given parse tree:

- \( t_0 \ = \ t_1 \rightarrow t_6 \)
- \( t_1 \ = \ t_2 \ t_5 \)
What is the type of the function \( f \)? Be as specific as possible.

\[
f :: ________________________________
\]

4. (8 points) .................................................... Disjoint Unions

A union type is a type that allows the values from two different types to be combined in a single type. For example, an expression of type \( \text{union}(A, B) \) might have a value of type \( A \) or a value of type \( B \). Union types can be made space efficient as they can take on only one type at a time, allowing their data fields to be reused across instantiations. The languages C and Haskell both have forms of union types.

The following C program fragment is written using a union type.

```c
void print (int);
...
union IntString {
    int i;
    char *s;
} x;
...
if ( ... ) x.i = 3 else x.s = "three";
...
print (x.i);
```

The program is considered well-typed by a C compiler.

(a) (2 points) Assuming a 32-bit machine, explain how the union-typed variable \( x \) can be stored using as little memory as possible.

(b) (2 points) Despite the fact that the program type-checks, the print statement may not work correctly. Why not? Will the run-time system catch the problem, or will the bug go undetected?
In Haskell, a union type \( \text{union}(A,B) \) would be written in the form \( \text{data UnionAB} = \text{TypeA} A \mid \text{TypeB} B \). Haskell’s union types are tagged, meaning a Haskell compiler associates the type that a union-typed variable assumes. In Haskell, the if statement above could appear as:

```haskell
data IntString = ISInt Int | ISString String;
...
double val = case val of
    ISInt v -> v * 2
...
x = if ... then ISInt 3 else ISString "three";
double x
...
```

(c) (2 points) A Haskell compiler considers this code to be well-typed, but the call to “double” may not execute correctly. Why not? Will the run-time system catch the problem, or will the bug go undetected?

(d) (2 points) Can the Haskell and C implementations be made equally space efficient? Explain.
C++ supports both public and private inheritance. Sometimes these are described by saying that public inheritance establishes an is-a relationship, while private inheritance establishes a has-a relationship. One distinction is that private inheritance makes the public functions of the parent class private in the child class.

Here is some example code to illustrate the difference between public and private inheritance:

```cpp
class Engine {
    public:
        virtual void start() {
            cout << getEngineType() << " started" << endl;
        }
        virtual string run() {
            cout << "attempting to run " << getEngineType() << endl;
            start();
        }
        virtual string getEngineType() {
            return "generic engine";
        }
};

class SteamEngine:public Engine { //A SteamEngine is an Engine
    public:
        virtual string getEngineType() {
            return "steam engine";
        }
};

class Car : private Engine { //A Car has an Engine
    public:
        virtual void startCar() {
            cout << "starting car" << endl;
            Engine::run();
        }
};
```

(a) (3 points) Will the code on lines 6-8 produce any compile-time or run-time errors?
1: int main()
2: {
3:     Engine e;
4:     SteamEngine s;
5:     Car c;
6:     e.start();
7:     s.start();
8:     c.start();
9:     return 0;
10: }

(b) (3 points) When Car :: startCar is executed, it makes a call to Engine :: run which in turn executes Engine :: start. What is the type of the this pointer that is passed implicitly to Engine :: run, and what does this imply about the virtual tables for Engine and Car?

(c) (3 points) List the public interfaces of Engine and Car. What can you conclude about the subtype relationship between a parent and child class using private inheritance?

(d) (3 points) Will the code on lines 7-9 produce any compile-time or run-time errors?
void checkUp(const Engine& e) {...}

int main()
{
    Engine e;
    SteamEngine s;
    Car c;
    checkUp(e);
    checkUp(s);
    checkUp(c);
    return 0;
}

6. (16 points) ........ Sequential and Concurrent Garbage Collection

Building a garbage collector is a complex software engineering task. Many strategies exist for doing so, each with their pros and cons. This question asks you to consider four well known implementations.

Reference Counting

Reference counting works by keeping track of the number of pointers which point to an object. When that number reaches zero, the object is deleted.

The C++ Standard Template Library provides an implementation of reference counting in the templated class auto_ptr. Pointer objects provided by auto_ptr work like regular pointers. However, the class is designed so that when the last auto_ptr that points to an object is deleted, the object is deleted as well. The test for doing so uses the reference count.

Consider the following code; you may assume that auto_ptrs will not attempt to delete a null pointer.

struct Node {
    int val_;
    auto_ptr<Node> next_
};

auto_ptr<Node> p(new Node());
p->val_ = 1;
p->next_ = 0;
p = 0;

When p is set to 0 on line 10, the number of auto_ptrs pointing to the Node becomes zero, and Node is deleted. Thus, running this code will produce the output:
Deleting node 1

Now consider a slightly more complicated piece of code:

```cpp
1: auto_ptr<Node> p(new Node());
2: p->val_ = 3;
3: p->next_ = 0;
4:
5: for ( int i = 2; i >= 1; --i ) {
6:   auto_ptr<Node> q(new Node());
7:   q->val_ = i;
8:   q->next_ = p;
9:   p = q;
10: }
11:
12: // p->next_->next_->next_ = p->next_;
13: p = 0;
```

(a) (2 points) What will this code print after `p` is set to 0 on line 13? Note that line 12 has been commented out.

(b) (2 points) Will the output change if line 12 is uncommented? If the answer is yes, what is the new output? Explain why.

(c) (2 points) If this loop made many iterations, and was part of an application that had to be very fast, would it make sense to use Reference Counting? Explain why. (Hint: you may want to read ahead and think about mark-and-sweep for comparison.)
**Mark and Sweep**

In typesafe languages like Java and Haskell, garbage collection can be implemented by the runtime system. In class we discussed one way of doing this called *Mark and Sweep*. Recall that *Mark and Sweep* works as follows:

- **Mark**: Find all objects reachable by pointers held by the program. Mark the heap cells holding these objects.
- **Sweep**: Iterate over every cell in the heap. Free the cells which have not been marked.

(c) *(2 points)* You are using Mark and Sweep to garbage collect a class project which is running too slow. Your code frequently allocates many small objects, but doesn’t hold on to many pointers. Your teammate is convinced that adding more memory to your machine will help. But after profiling your garbage collector, you notice that its time is split 10% in marking and 90% in sweeping. What will happen if you take his advice and add more memory?

**Copying Collector**

A *Copying Collector* is a variation on *Mark and Sweep*. Copying collectors maintain two heaps: a working heap and a copy heap. Memory is allocated on the working heap; the copy heap is used during garbage collection. Garbage collection works as follows:

- **Mark**: Identical to *Mark and Sweep*.
- **Copy**: Copy marked objects from the working heap to the copy heap.
- **Swap**: Make the copy heap the working heap, and vice versa.

(e) *(2 points)* Will switching to a *Copying Collector* improve your class project’s performance? Explain why or why not.
Concurrent Garbage Collection

*Reference Counting, Mark and Sweep and Copying Collectors* are examples of *stop-the-world* algorithms. Their correctness depends on the fact that no other threads are running at the same time as the garbage collector. In a concurrent environment, running one of these algorithms in parallel with a thread that is using garbage-collected memory could lead to incorrect behavior.

(f) (2 points) For *Reference Counting*, explain how two threads with an `auto_ptr` to the same object could produce a memory error. Remember that the STL implementation of `auto_ptr` does not use locks.

(g) (2 points) For *Mark and Sweep* or a *Copying Collector*, explain what could go wrong if a second thread performs a write during marking, and changes a pointer, `p`, to point from `o1` to `o2`.

(h) (2 points) Again considering a *Mark and Sweep* or a *Copying Collector*, could something go wrong if a second thread were to perform only reads through pointers during marking? Explain why or why not.
Recall that Java source code is compiled into bytecode, and Java bytecode is examined by the bytecode verifier before it is executed in the Java virtual machine. Since bytecode could be written by hand, the verifier must check for properties that are normally guaranteed by the Java compiler. One of the properties checked by the verifier is that every derived class constructor calls a base class constructor. This is part of checking that every object is initialized before it is used.

The verifier must check that for every derived-class constructor, every execution path within the constructor calls either a base-class constructor, or calls another derived class constructor that calls a base-class constructor. Some past versions of the Microsoft Internet Explorer virtual machine used a bytecode verifier that incorrectly analyzed certain uses of exception handling. This problem explores some of the issues surrounding that bug. Note that it is acceptable for a constructor to raise an exception; it is only an initialize-before-use error to return normally from a constructor without initializing the object.

Initialization table. Let us assume that the verifier associates an initialized bit with each line of constructor code, and examines lines of constructor code one-by-one. The initialized bit is set to zero by default, and set to one if every code path to this line of code calls a base-class constructor. We can visualize the algorithm by assuming an initialization table, storing the initialized bit for each line number. Since there may be jumps in the bytecode, we make use of the control flow graph of the program. The control flow graph is a directed graph with line numbers as nodes and an edge from \( i \) to \( j \), if it is possible for execution/control to go directly from line \( i \) to line \( j \). An example is shown in Figure 1, using some simplification of the actual bytecode syntax.

**Figure 1: Bytecode and control flow graph**

1 : iconst 2  
2 : load X  
3 : ifgreater else jump 6  
4 : call super()  
5 : jump 7  
6 : nop  
7 : return

Algorithm on control flow graph. Using the control flow graph, the bytecode verifier can fill in the initialization table as follows: if line \( i \) calls `super()` , then the initialization bit for this line is 1. Otherwise, the initialization bit for line \( i \) is the minimum of the initialization bits associated with all the line numbers that point to \( i \) in the control flow graph. This sets the initialization bit for line \( i \) to 1 iff all possible execution paths to \( i \) call a base-class constructor.

Calls to other constructors. We assume that the verifier rejects a class if its constructors are mutually recursive and that the verifier has some way of placing the con-
constructors of the class in an ordered list \( C_1, \ldots, C_n \) such that constructor \( C_i \) only invokes constructors \( C_{i+1}, \ldots, C_n \). The verifier then starts at constructor \( C_n \) and uses the algorithm above to see if the initialized bit is 1 at every return statement. If so, it marks the constructor as *good*. In analyzing constructor \( C_i \), the analysis algorithm treats a call to good constructors \( C_{i+1}, \ldots, C_n \) as if they were calls to \texttt{super}(). If some constructor is not good, then the class is rejected.

**Example.** We illustrate the algorithm presented so far on the following example class \( B \), which has two constructors, \( B() \) and \( B(\text{int}) \):

```java
class public B
  .super A

  .method public <init> ()
 1:  load 0
 2:  call B/<init>(1) // call to other constructor
 3:  aconst_null
 4:  return
  .end method

  .method public <init> (i)
1:  load 0
2:  call A/<init>() // call to super
3:  return
  .end method
```

**Second constructor.** The verifier first analyzes \( B(\text{int}) \) because this constructor does not call any further constructors, and fills in the initialization table as shown below. Since the initialized bit is 1 for every return statement, the algorithm concludes that the constructor \( B(\text{int}) \) is good.

<table>
<thead>
<tr>
<th>Line No</th>
<th>Code</th>
<th>Initialized</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>load0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>call A/&lt;int&gt;()</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>return</td>
<td>1</td>
</tr>
</tbody>
</table>

**First constructor.** The verifier then analyzes constructor \( B() \). When the call \( B(1) \) is encountered on line 2, the algorithm uses the fact that \( B(\text{int}) \) is *good* to set the initialized bit to 1.

<table>
<thead>
<tr>
<th>Line No</th>
<th>Code</th>
<th>Initialized</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>load0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>call B/&lt;int&gt;(1)</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>aconst_null</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>return</td>
<td>1</td>
</tr>
</tbody>
</table>
Questions:

(a) (4 points) Consider the following byte code, whose first constructor uses the code (and control flow graph) shown in Figure 1:

```
.class public B
.super A

.method public <init> ()
1 : iconst 2
2 : load X
3 : ifgreater else jump 6 // if true move to next line else jump to 6
4 : call A/<init>() // call to super()
5 : jump 7
6 : nop
7 : return
.end method

.method public <init> (i)
1 : load 0
2 : call A/<init>() // call to super
3 : aconst_null
4 : return
.end method
```

Fill out the initialization table for the constructor \texttt{B()} in the above code. (This can be done independently of the other constructor because \texttt{B()} does not call any other constructor.)

<table>
<thead>
<tr>
<th>Line No</th>
<th>Code</th>
<th>Initialized</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>iconst 2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>load X</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>ifgreater else jump 6 // if true move to next line else jump to 6</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>call A/&lt;init&gt;() // call to super()</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>jump 7</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>nop</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>return</td>
<td></td>
</tr>
</tbody>
</table>

(b) (1 point) What would the byte code verifier conclude for the above constructor? (Circle one)

Good  | Bad

(c) (3 points) Suppose we replace the \texttt{nop} on line 6 with the call \texttt{B(1)} to the other constructor. What would the verifier conclude in that case? Why? (One sentence)

```
1 | 2 | 3 | 4 | 5 | 6 | 7     
---|--|--|--|--|--|--
```

```
<table>
<thead>
<tr>
<th>Line No</th>
<th>Code</th>
<th>Initialized</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>call B/&lt;init&gt;(1)</td>
<td></td>
</tr>
</tbody>
</table>
```

(d) (4 points) In the remaining parts of this problem we will look at verification with exceptions. Try-catch blocks in Java source code are compiled into an exception table. This table lists how to handle specific types of exceptions, when they are thrown in
specified parts of the code. The `.catch` indication in the following symbolic representation of bytecode says to handle a SecurityException thrown between labels l1 and l2 by jumping to label l3. (Label l1 appears on line 2, label l2 on line 3, and label l3 on line 4.)

*Ignoring the exception table* in the code below, fill in the initialization tables for both constructors.

```java
.class public B
.super com/ms/security/SecurityClassLoader

.method public <init> ()
1 : load 0
2 : l1 : call B/<init>(1) // call to other constructor
3 : l2 : aconst_null
4 : l3 : return
 .catch java/lang/SecurityException from l1 to l2 using l3.
 .end method

.method public <init> (i)
1 : load 0
2 : call com/ms/security/SecurityClassLoader/<init>() // call to super
3 : return
 .end method
```

<table>
<thead>
<tr>
<th>Line No</th>
<th>Code</th>
<th>Initialized</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>load0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>call A/&lt;int&gt;()</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>return</td>
<td></td>
</tr>
</tbody>
</table>

(Fill in the blanks)

<table>
<thead>
<tr>
<th>Line No</th>
<th>Code</th>
<th>Initialized</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>load0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>call B/&lt;int&gt;(1)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>aconst_null</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>return</td>
<td></td>
</tr>
</tbody>
</table>

(Fill in the blanks)

(e) (2 points) Now considering the effect of `.catch` in the code given in the last part of this question, draw the control flow graph for the constructor B() by drawing edges between the vertices 1,2,3,4 below. Recall that the vertices of the control flow graph (as defined in this problem) refer to line numbers, not labels in the code. If an instruction on line i may not complete if an exception is raised on line i, represent this as a jump from line i − 1.

```
1
2
3
4
```

(f) (2 points) How does `.catch` change the initialization table for B() in the previous part of this problem? Taking `.catch` into account, is constructor B() good or bad?
8. (15 points) Futures in Haskell

In this problem we will explore various design considerations for Haskell futures. Recall from the lecture on MultiLisp the benefits of using futures: they are notationally lightweight, they provide a simple means for thread coordination, and concurrency is determined by the run-time system.

The MultiLisp future design we saw in lectures was an example of explicit futures. With explicit futures, the programmer has to create the future, and then explicitly request (and possibly block on) the result of the future. This request is also known as a “force” or “touch”.

To implement explicit futures in Haskell, we use the MVar data structure. Recall that an MVar, of type MVar a, is mutable location that can be in one of two states: it be empty, or it can be full with a value of type a. The function putMVar is used to fill an empty MVar, while takeMVar is used to read (and clear) a filled MVar.

First, we encode a future, or promise, using the EFuture type:

```haskell
data EFuture a = EFuture ThreadId (MVar a)
```

Here, ThreadId is used to keep track of the thread executing the future, while the MVar a is the result place-holder.

(O) (2 points) Here is a partial implementation of the two core primitives for explicit futures.

```haskell
efuture :: IO a -> IO (EFuture a)
efuture act = do v <- newEmptyMVar
               tid <- forkIO ________________
               return (EFuture tid v)

force :: EFuture a -> IO a
force (EFuture _ v) = takeMVar v
```

The efuture function takes an IO action act, spawns a new thread in which it executes act, and immediately returns the corresponding future (EFuture). Conversely, force take a future and returns the result of the corresponding act computation (blocking until available).

Complete the implementation of efuture.

Observe that with explicit futures, the burden of fully evaluating the (future) computation is placed on the programmer by requiring them to use use force. Careless programming choices can result in a loss of performance or underutilization of
available parallelism. For instance, a programmer can force a future immediately after it is created (effectively forcing a sequential semantics). Such a program will likely be slower than the future-less sequential program because of the cost of doing a fork.

Hence, in a call-by-need language such as Haskell, we prefer to leverage the run-time system to force the evaluation of a future when its value is needed. Unfortunately, we cannot implement such behavior only with explicit futures. Luckily, GHC provides a function:

\[
\text{unsafeInterleaveIO} :: \text{IO } a \rightarrow \text{IO } a
\]

The function \text{unsafeInterleaveIO} allows an IO computation to be deferred lazily. When passed a value of type \text{IO } a, the IO will only be performed when the value of the \text{a} is demanded. (This is used to implement lazy IO, such as file reading with \text{hGetContents}). In other words, we can use \text{unsafeInterleaveIO} to delay the execution of a sequential monadic IO-computation, breaking sequentially. Using our explicit future implementation and \text{unsafeInterleaveIO}, we implement implicit futures as shown below.

\[
\text{ifuture} :: \text{IO } a \rightarrow \text{IO } a
\]

\[
\text{ifuture } \text{act} = \text{do}
\]

\[
\text{f}@\text{(EFuture tid v)} \leftarrow \text{efuture } \text{act} \quad \text{-- create explicit future}
\]

\[
\text{unsafeInterleaveIO} \text{ (do res } \leftarrow \text{force f } \text{-- force explicit future}
\]

\[
\text{killThread tid } \quad \text{-- kill thread (for GC)}
\]

\[
\text{return res} \quad \text{-- return result}
\]

When the value returned by \text{ifuture} is needed, the future is forced, the executing thread is killed, and the computation result is made available.

We will use the example below to explore the difference between explicit and implicit futures. (Note that \text{threadDelay} \times puts a thread to sleep for \times微秒s.)

\[
\text{testFun futureConstrFun futureForceFun } = \text{do}
\]

\[
\text{putStrLn } "A"
\]

\[
\text{f } \leftarrow \text{futureConstrFun} \text{ (do threadDelay 1000000 -- sleep 1 second}
\]

\[
\text{putStrLn } "B"
\]

\[
\text{putStrLn } "C"
\]

\[
\text{futureForceFun } \text{f}
\]

(b) \textit{(2 points)} What does the following program print out? (Assume a round-robin scheduler.)

\[
\text{test1 } = \text{do} \quad \text{testFun ifuture } (\_ \rightarrow \text{return } ())
\]

\[
\text{threadDelay 5000000 } \text{-- sleep 5 seconds}
\]

Note that the argument for \text{futureForceFun} is a function that ignores its argument and simply returns unit.
(c) (2 points) If we remove unsafeInterleaveIO in the definition of ifuture, what do you expect test1 to print out? (Assume a round-robin scheduler.)

(d) (2 points) Consider the programs:

```haskell
  testI = testFun ifuture (\_ -> return ())
  testE = testFun efuture force
```

Assuming a round-robin scheduler, and that executing putStrLn "..." takes considerably less than 1 second, what is the expected output difference between the program using implicit futures (testI) and the one using explicit futures (testE), if any? If there is a difference, explain in 1 or 2 sentences what changes to the implicit future implementation are necessary for testI and testE to produce the same output.

(e) (2 points) If there is a difference, explain in 1 or 2 sentences what changes to the testE are necessary for testI and testE to produce the same output. (The first argument to testFun must be kept to efuture.)

Though the above explicit and implicit implementations are well-suited concurrency "primitives", modifying existing sequential code to take advantage of additional processing power is not always as simple as wrapping code by the future combinator (and adding force in the case of explicit future). Consider, for example, the following function and programs composed with it. (Recall that forM_ [1..n] (\_ -> act) executes the act action n times.)
simple futureFunc forceFunc = do
  let n = 1000
  v <- newIORef 0 -- create new reference
  f1 <- futureFunc (do threadDelay 10000 -- sleep 10ms
    forM_ [1..n] (\_ -> do
      x <- readIORef v -- read ref
      writeIORef v (x+1) -- and increment
    ))
  f2 <- futureFunc (forM_ [1..n] (\_ -> do
    x <- readIORef v -- read ref
    writeIORef v (x-1) -- and increment
  ))
  x <- readIORef v -- read ref
  writeIORef v (x+1) -- and increment
  forceFunc f1 -- apply force function
  forceFunc f2 -- apply force function
  readIORef v -- read ref, return read value

-- Sequential program:
simpleS = simple (\x -> x) (\_ -> return ())
-- Program using explicit futures:
simpleE = simple efuture force
-- Program using implicit futures:
simpleI = simple ifuture (\_ -> return ())

(f) (2 points) Regardless of whether we use explicit futures (simpleE) or implicit futures (simpleI), the result will not always correspond to that of the sequential program (simpleS). This is the case even when the operations on the shared mutable reference v are linear. In fact, although simpleS only returns 1, simpleE and simpleI have returned 0 and -999, respectively. Briefly explain why the programs with the futures differ from the sequential program.

(g) (3 points) Safe futures are used when sequential consistency is desired. Specifically, they guarantee the following safety properties: (i) a future cannot witness the effects of its continuation, and (ii) a continuation cannot witness the intermediate effects of the future. The latter point guarantees that the continuation will only see the (logically) last effect of the future. Consider, for example, the future created on line 4 of the function simple above. In this case, the future cannot witness the effects of the code on lines 9-17, while the continuation (lines 9-17) cannot witness the intermediate side effects of the code on lines 4-8.

Suppose we wish to implement safe futures in Haskell by keeping a log for each reference of the source line number where it was modified and to what value. A readIORef then consults the log and reads the value whose line number is less than that of the readIORef. If there is a value with a greater line number, it
“signals” the responsible future or continuation to start over (from last reference read) and removes the entry from the log.

For example:

```haskell
1: x <- newIORef 0
2: f <- sfuture (do threadDelay 10000 -- 10ms
3:   v <- readIORef x
4:   writeIORef x (v+1))
5: v <- readIORef x
6: writeIORef (x+42)
```

will have the following log for reference `x`, regardless of whether the continuation executes before the future (if it does execute before the future, it will have to start over):

```
1: 0
4: 1
6: 43
```

Is this design approach correct? If yes, no explanation necessary. If no, explain in 1 or 2 lines why not.

---

9. **(12 points)** ........................................... Software Transactional Memory

One advantage of using transactions over locks is that transactions are “composable”, while locks are not. This problem asks you to compose transactions, using Haskell STM. For comparison we also explore a Java implementation using locks.

Consider the following implementation of a bank account, with a deposit combinator.

```haskell
type AccountH = TVar Int

deposit :: AccountH -> Int -> STM()
deposit acc amount = do bal <- readTVar acc
                          writeTVar acc (bal + amount)
```

(a) **(3 points)** A partial implementation of the withdraw combinator is given below.

```haskell
withdraw :: AccountH -> Int -> STM()
withdraw acc amount = do bal <- readTVar acc
                         if bal < amount
                         then ______________________________
                         else ______________________________
                         writeTVar acc (bal - amount)
```

---
The function `withdraw` should withdraw `amount` from `acc` only if the current balance is above `amount`. Complete the definition of `withdraw`.

(b) (2 points) In the implementation of `withdraw` we explicitly check that the balance of an account is never below 0. Suppose we change the implementation to:

```hs
withdraw :: AccountH -> Int -> STM()
withdraw acc amount = do bal <- readTVar acc
                         writeTVar acc (bal - amount)
```

We wish to make sure that `withdraw` does not succeed if the balance is below the amount. Is there a way to provide this guarantee without ever comparing `acc`'s balance with `amount`? Explain in a sentence.

(c) (3 points) Implement an atomic `transfer2` transaction using `withdraw` and `deposit` that withdraws an amount from one of two source accounts (withdraw from first if it has enough money) and deposits the money into a destination account. (Assume the implementation of `withdraw` is as described in (b) above.)

```hs
transfer2 :: AccountH -> AccountH -> AccountH -> Int -> IO ()
transfer2 sAcc1 sAcc2 dAcc amount =
                                
                                
                                deposit dAcc amount

transfer :: AccountH -> AccountH -> Int -> IO ()
transfer sAcc = transfer2 sAcc sAcc
```

Complete the implementation of `transfer2`. Your solution should not use an if-then-else. 
*Hint:* Observe the difference in the type of `transfer` and `withdraw/deposit`, and recall the STM compositional building blocks.

For comparison, the following lightweight Java class correctly uses locks to implement bank accounts with `deposit` and `withdraw` methods:
class AccountJ {
    int balance;
    synchronized void deposit(float amt) {
        balance += amt;
    }
    synchronized void withdraw(float amt) {
        if(balance < amt)
            throw new OutOfMoneyError();
        balance -= amt;
    }
}

(d) (2 points) Suppose we modify the class to add the transfer method as defined below:

    void transfer(AccountJ other, float amt) {
        other.withdraw(amt);
        this.deposit(amt);
    }

What problem, if any, arises if we use transfer method as implemented above?

(e) (2 points) Suppose, instead, the transfer method was defined as follows:

    synchronized void transfer(AccountJ other, float amt) {
        other.withdraw(amt);
        this.deposit(amt);
    }

What problem, if any, arises if we use transfer method as implemented above?