What to turn in

Turn in the assignment during class on the due date.

1. Substitution Lemma (10 pts)

   Prove Lemma 9.12 in Winskel: that for call-by-name REC$^+$, where $\mathcal{C}[\![\cdot]\!]$ is the B3 meaning function,

   $$\mathcal{C}[\![e]\!]\phi\rho[x \mapsto \mathcal{C}[\![e']\!]\phi\rho] = \mathcal{C}[\![e\{e'/x\}]\!]\phi\rho$$

   (Translated into the notation we have been using in class)

2. Static vs. Dynamic Scoping (10 pts)

   What is the value of the following uF+let program under static and dynamic scope, respectively? (The if expression has been written with explicit then and else keywords to make the parsing clear.)

```
let f = fn n (fn g1 (fn g2
    let x = n+10 in
      let g = fn u x in
        if n=0 then (f 1 g #u)
          else if n=1 then (f 2 g1 g)
            else (g1 #u) + (g2 #u) + (g #u)
    ))
in
    (f 0 #u #u)
```

3. First-class Modules (20 pts)

   The simple module values we developed a semantics for in class had more expressive power than those in many programming languages in that they were *first-class values*. In fact, they look suspiciously close to objects in an object-oriented language. In this problem we will explore the correspondence further.

   Suppose we extend call-by-value uF with the following expression forms similar to those we saw in the simpler module mechanism presented in class:

   $e ::= \ldots \mid \mathsf{module}\ m\ (\mathsf{fields}\ X; \mathsf{methods}\ Y) \mid e.x \mid \mathsf{extend}\ e_1\ \mathsf{by}\ e_2$
   $f ::= \mathsf{fn}\ x\ e$
   $X ::= x_1 = e_1, \ldots, x_n = e_n$
   $Y ::= y_1 = f_1, \ldots, y_n = f_n$

   In this new language, uF$_m$, a module expression introduces a name for the module value itself, m. This identifier is in scope in the remainder of the module expression and may be used in the method expressions f_i (but not the field definitions e_i). None of the other x_i, y_i are in scope in the e_i, f_i, but you can get to them indirectly in the f_i's via m. The field bindings are "by-value", in that the e_i are evaluated when the module expression is encountered, not when the fields are selected. The methods are constrained to be functions and so are automatically values.

   The expression $e.x$ (or $e.y$) selects the field value named x (or method named y) that is exported by the module value that e evaluates to.

   The expression extend e_1 by e_2 produces a new module values from two existing ones that are obtained by evaluating e_1 and e_2. The new module value defines an identifier if it is defined in either e_1 or e_2, and the definitions in e_2 take precedence.

   Using these features, we can write code that is at least superficially object-oriented (assuming the usual desugaring for let):

```
let make_point = rec fn coords
    (module p (fields x = first coords,
                      y = rest coords;
               methods lengthsq = fn u p.x*p.x + p.y*p.y,
                       minus    = fn p2 make_point <p.x - p2.x, p.y - p2.y>
    )) in
  let p1 = make_point <1,3> in
    let p2 = make_point <2,4> in
      p1.minus(p2).lengthsq #u
```

The result of this program is 5.

In this problem we will extend the semantics of uF to describe uF$_m$. Unlike in the module semantics given in class, we will consider modules to be *environment extenders*: that is, members of the domain *Env* $\rightarrow$ *Env* that extend an existing environment with a set of possibly new bindings.

Questions

(a) (3 pts) Make all changes to the domain equations of uF necessary to support these language features.

(b) (3 pts) The module expression only permits the module name to occur in method definitions. Why is this limitation important for the ability to define a semantics for this language?

(c) (10 pts) Define the semantic function $\mathcal{C}$ for the new uF$_m$ expression forms.

(d) (4 pts) Read Section 15.12.4.8 of the Java Language Specification, Second Edition [Gosling, Joy, Steele, and Bracha 2000], available online at

`http://java.sun.com/docs/books/jls/`

Informally, in a paragraph, use this example to illustrate how method overriding in Java `extends` differs from uF_m's extend operator. If necessary, give references to other parts of the Java specification that explain how this part of Java works.

(Don't use the first edition, which is available from the same page).

4. Deterministic parallelism (25 pts)

Concurrency is one language feature for which we have so far avoided giving a denotational semantics. One reason is that concurrent execution is virtually always nondeterministic, and models for nondeterminism involve powerdomains or equivalently complex constructions. However, we can write a semantics for *deterministic* parallelism using continuation-passing style. We extend the uF$_!$ language with an expression form to spawn a new thread of execution:

$$e ::= \ldots \mid \mathsf{spawn\ e}$$

Informally, this expression generates a new thread of execution that evaluates the expression e. The spawn expression itself returns immediately with the value #u, and the fact that e is evaluated can be observed only by its side effects.

As an example, given inputs m and n, the following program spawns a single thread to perform a calculation, proceeds until the answer is needed, then waits until the answer becomes available.

```
(fn n (fn m
    let FLAG = ref #f in
    let Y = ref 0 in
    let u1 = spawn (let u1 = Y := n*n + m*m in FLAG := #t) in
    let x = n*m + m*n in
    let wait = rec w (fn x (if !FLAG then x else w x)) in
    let u2 = wait #u in
    !Y - x
))
```

This program, applied to arguments 3 and 4, gives $3^2 + 4^2 - 2 \cdot 3 \cdot 4 = 25 - 24 = 1$.

Because execution is deterministic, we must model the action of the thread scheduler. We assume that a suitable domain, *Scheduler*, exists to capture the state of the scheduler. This state consists of a set of threads that are waiting to run. A waiting thread will be represented by a member of the domain *Resumption*:

$$\mathit{Resumption} = \mathit{Store} \to \mathit{Scheduler} \to \mathit{Answer}$$

We are not particularly interested in exactly how the scheduler does its work, so we will assume the existence of a pair of continuous functions:

push-thread $\in$ *Scheduler* $\to$ *Resumption* $\to$ *Scheduler*
pop-thread $\in$ *Scheduler* $\to$ (*Resumption* $\times$ *Scheduler*)

such that *push-thread* adds a resumption to the pool of waiting threads in the scheduler, and *pop-thread* selects (somehow) one of the waiting threads and returns both it and the updated scheduler state with that resumption removed. We also assume the existence of an empty, "initial" scheduler $S_0 \in$ *Scheduler*, from which popping is forbidden.

If a thread never gets a chance to execute, it is starved. *Fairness* means that no thread is starved. Assuming that *pop-thread* is fair (for example, if the scheduler is a FIFO queue), the only way a thread can starve is if some thread gets to run forever. We want to make sure that enough preemption occurs that threads cannot starve other threads. One way to do this is to preempt each thread after each execution step, and that is what this language should do.

Continuations must also be extended to take the scheduler as an argument:

$$\mathit{Cont} = \mathit{Value} \to \mathit{Resumption}$$

In this language, there is one main thread. It may spawn child threads, which might themselves have children, etc. All threads other than the main thread are children. When a child thread executes to a value, that value is discarded and the thread terminates. When the main thread executes to a value, that value is returned as the result of the program and all threads are terminated. Note that it is therefore possible to have a program that contains nonterminating threads but itself does terminate.

You should use the following two continuations to model this behavior in your semantics:

$$\begin{array}{lcl} k_{\mathsf{main}} & = & \lambda v \in \mathit{Value}.\lambda\sigma \in \mathit{Store}.\lambda S \in \mathit{Scheduler}. \\ & & \quad \lfloor \langle \mathit{Value}(v), \sigma \rangle \rfloor \\ k_{\mathsf{child}} & = & \lambda v \in \mathit{Value}.\lambda\sigma \in \mathit{Store}.\lambda S \in \mathit{Scheduler}. \\ & & \quad \mathsf{let}\ \langle R, S' \rangle = \mathit{pop\text{-}thread}\ S\ \mathsf{in} \\ & & \quad R\ \sigma\ S' \end{array}$$

Questions

(a) (5 pts) What are the possible results of executing the following programs under different scheduling or preemption implementations?

Program 1

```
(let f1 = (fn n (fn m
   let FLAG = ref #f in
   let Y = ref 0 in
   let u1 = spawn (let u = Y := n*n + m*m in FLAG := #t) in
   let u2 = spawn (let u = Y := n + n*m + m in FLAG := #t) in
   let x = n*m + m*n in
   let wait = rec wait' (fn z (if !FLAG then z else wait' z)) in
   let u3 = wait #u in
   !Y - x)) in
  f1 3 4)
```

Program 2

```
(let testwait = (fn test (rec wait'
                   (fn x (if (test #u) then x else wait' x)))) in
   (let f2 = (fn n (fn m
      let FLAG1 = ref #f in
      let FLAG2 := ref #f in
      let Y := ref 0 in
      let u1 = spawn (let u = Y := n*n in
                      FLAG1 := #t) in
      let u2 = spawn (let u = Y := !Y + m*m in
                      let u = testwait (fn u (!FLAG1)) #u in
                      let u = FLAG2 := #t in
                      let u = FLAG1 := #f in
                      FLAG2 := #f) in
      let u = testwait (fn u (!FLAG2)) #u in
      !Y)) in
     f2 3 4))
```

(b) (10 pts) Finish the necessary modifications to the uF$_!$ domain equations.

(c) (10 pts) Define the semantic function $\mathcal{C}$ for the extended language. Given a closed expression e, its denotation should be $\mathcal{C}[\![e]\!]\rho_0 k_{\mathsf{main}}\sigma_0 S_0$, where ρ_0 and σ_0 are the initial environment and store, k_{main} is the continuation for the main thread, and S_0 is the provided initial, empty scheduler. (*Hint:* Define helper function(s) to do any repetitive operations)