Problem Set 7 Solutions

These solutions are complete except for missing environment diagrams for Group Problem 2 (Parameter Passing).

Revisions: May 20: Added missing F5aa frame to Figure 2. May 19: Fixed bugs in the make-counter environment diagrams and descriptions (Group Problem 3).

Individual Problems

Individual Problem 1 [25]: Environmental Action

a. [20] You were asked to draw an environment diagram showing all the environments and closures created when the following program is run on the input argument list [3;5] in statically scoped HOFL:

\[
\text{hofl \( (i \ j) \) (bind \( q \ (f \ p) \) (bind \( r \ (q \ i \ j) \) (bind \( \text{answer} \ (p \ (r \ m) \ (r \ (f \ m))) \ \text{answer})))) (def \( f \ (g) \ (\text{fun} \ (a \ b) \ (g \ b \ a))) \ (def \( p \ (d \ e) \ (\text{fun} \ (h) \ (h \ d \ e))) \ (def \( m \ (x \ y) \ (\% \ x \ y)))
\]

Fig. 1 shows a contour diagram for the desugared program. You did not have to draw this, but it is helpful for understanding the environment diagram. Fig. 2 shows the final environment diagram. Each frame name beginning with \( F_i \) has been chosen to match up with the corresponding contour \( C_i \) in the contour diagram. The key features to note are:

- Chains of frames have the same shape as nested contours;
- The evaluation of a function abstraction yields a closure whose “umbilical cord” points to a frame corresponding to the contour in which the abstraction is nested.
- A frame resulting from the application of a closure contains the parameter names of the closure abstraction and points to the same frame as the “umbilical cord” of the closure. To emphasize this feature, each frame created by application of a closure has been positioned to the right of the closure at the same height.
- Frame created by \( \text{bind} / \text{bindrec} \) extend the frame in which they are evaluated.

Many of you found this program challenging to reason about. It’s worth noting that it illustrated functions we’ve talked about all semester. The \( f \) function is the “flip” function: it transforms a two-argument function \( g \) into another two-argument function that takes its arguments in reverse order. The \( p \) function is the Church-style “pair” function that pairs any two values \( d \) and \( e \) by wrapping them in a function that uses its single functional argument \( h \) to select between (or combine) the two values. So \( q \), which is defined as \( (f \ p) \), is the flip of the pairing function – i.e., a function that pairs its two arguments in opposite order. Then \( r \), which is defined as \( (q \ i \ j) \), must be the flipped pair of 3 and 5 – i.e., the Church pair of 5 and 3, which is equivalent to \( (\text{fun} \ (h) \ (h \ 5 \ 3)) \). It is apparent from Fig. 2 that \( r \) is bound to a closure with this meaning.
Figure 1: Contours for the desugared HOFL program.
Figure 2: Environment diagram for the Hofl program called on arguments 3 and 5.
Finally, \textbf{answer} is the result of pairing two values, \((r\ m)\) and \((r\ (f\ m))\). The first value, \((r\ m)\) combines the values in the pair \((5,3)\) by the mod function, so it yields \(5\mod 3 = 2\). The second value, \((r\ (f\ m))\) combines the values in the pair \((5,3)\) by the flipped mod function, so it yields \(3\mod 5 = 3\). The result of the pairing is a Church pair representing \((2,3)\) – i.e., a closure equivalent to \((\text{fun}\ (h)\ (h\ 2\ 3))\).

\textbf{b.} [5] Dina McScoop has declared “The desugaring of multiple-argument function abstractions doesn’t work in dynamically scoped HOFL.” Here’s what Dina means, using the application

\[ ((\text{fun}\ (a\ b)\ (+\ a\ b))\ 1\ 2) \]

as an example. Desugaring of this application yields:

\[ (((\text{abs}\ a\ (\text{abs}\ b\ (+\ a\ b)))\ 1)\ 2) \]

Let’s suppose this expression is evaluated in an environment frame \(F_0\). Then the first application performed is the inner application \(((\text{abs}\ a\ (\text{abs} b\ (+\ a\ b)))\ 1)\), which creates a new frame \(F_1\) that extends \(F_0\) and binds \(a\) to 1:

\[
\begin{array}{c}
F_0 \\
F_1 \begin{array}{c} a \end{array} 1
\end{array}
\]

The result of this application is the result of evaluating the body expression \((\text{abs} b\ (+\ a\ b))\) in \(F_1\). In dynamic scope, this evaluation returns the raw abstraction rather than a closure over this abstraction. Moreover, when the inner application returns, the frame \(F_1\) created for the inner application is no longer accessible (and in practice would be deallocated). Then the outer application is performed by applying the abstraction \((\text{abs} b\ (+\ a\ b))\) resulting from the inner application to the argument 2. This creates a new frame \(F_2\) that extends the original evaluation frame \(F_0\) and binds \(b\) to 2:

\[
\begin{array}{c}
F_0 \\
F_1 \begin{array}{c} a \end{array} 1 \quad F_2 \begin{array}{c} b \end{array} 2
\end{array}
\]

Now we evaluate the body expression \((+\ a\ b)\) in frame \(F_2\). Although \(b\) is bound to 2 as expected, the binding of \(a\) to 1 from frame \(F_1\) has been lost. Indeed, we do not know what \(a\) is bound to! It might have a value relative to \(F_0\), but it might not. And if it does have a value, that value could be arbitrary.

So Dina is right: in dynamic scope, even the basic desugaring of curried functions does not work! A dynamically scoped language \textit{must} support multi-argument abstractions as kernel forms.

\textbf{Individual Problem 2 [25]: Partial Evaluation}

The complete partial evaluator is shown in Fig. 3. A few notes:

- As specified in the assignment, the environment argument to \texttt{peval} binds names to values of type \texttt{Fofl.valu}. But it would also be possible to bind the names to \textit{literal expressions}. 

4
• According to the **Var** case, any name not bound in the environment remains unchanged by partial evaluation. So in **partialEval**, **peval** is initially called with an empty environment on the program body and the body of each top-level function declaration. Since the values of the program formal parameters and function formal parameters are unknown.

• In the **PrimApp** case, all operands must be recursively simplified before any decisions can be made. Otherwise, we would miss simplification opportunities like (** (+ 1 2) (+ 3 4)**). If some operand is not a literal value, the operation **cannot** be performed by the partial evaluator, and a residual application of the operator to the simplified operands is returned. (It’s important to use the simplified operands rather than the original ones in the case where partial evaluation has successfully simplified some operands.) But if all simplified operands are literal values, the partial evaluator attempts to perform the operation at simplification time. If the operation succeeds with a **valu**, the result of must be wrapped it in a **Lit** to convert it to an expression. But the operation might fail due to “bad” operands. To catch this case, the operation is performed in a **try/with** to catch possible exceptions. If an **EvalError** is caught, simplification of the operation is aborted, and the partial evaluator returns a residual application of the operator to the simplified operands.

• In the **If** case, if the test expression can be simplified to a boolean literal, the partial evaluator returns the result of simplifying the appropriate branch. Otherwise, a residual conditional expression is returned containing the simplified subexpressions.

• In the **Bind** case, the definition expression must be recursively simplified before any decisions are made. Otherwise, we would miss optimization opportunities like (**bind x (+ 1 2) ...**). If the simplified definition is an literal value, the **bind** expression can be eliminated by substituting the literal value for the bound variable name in its body; this substitution is accomplished by binding the bound name to the value in the environment. If the simplified definition is not a literal, a residual **bind** is constructed, whose parts include simplified versions of the definition and body.

• As explained in the assignment, this partial evaluator does not do anything fancy for function applications. (In practice, a partial evaluator might expand calls to non-recursive functions.) So a function application is simplified by constructing a residual function application with simplified subexpressions.

**Group Problems**

**Group Problem 1 [16]: Safe Transformations**

**a.** (** + I I**) $$\mapsto$$ (** * 2 I**)  
   i. **HOF**: safe because **I** is referentially transparent (i.e., evaluating the same expression in the same environment always yields the same result.)  
   ii. **HOILIC**: safe because one variable reference can’t change the meaning of the other.

**b.** (** + E E**) $$\mapsto$$ (** * 2 E**)  
   i. **HOF**: safe because **E** is referentially transparent (i.e., evaluating the same expression in the same environment always gives the same result).  
   ii. **HOILIC**: unsafe because **E** may perform a side effect. For example:
module FoflPartialEval = struct

open FunUtils
open ListUtils
open Fofl

let isLit exp =
  match exp with
  Lit _ -> true
  | _ -> false

let litVal exp =
  match exp with
  Lit v -> v
  | _ -> raise (Failure ("not a lit" ^ (expToString exp)))

let rec partialEval (Pgm(fmls, body, fcns)) =
  Pgm(fmls, peval body Env.empty,
      map (fun (Fcn(name, fs, bd)) -> Fcn(name, fs, peval bd Env.empty))
      fcns)

and peval exp env =
  match exp with
  Lit v -> exp
  | Var s -> (match Env.lookup s env with
               None -> exp
               | Some v -> Lit v)
  | PrimApp(Primop(name, f) as op, rands) ->
    let rands’ = map (flip peval env) rands in
    if for_all isLit rands’ then
      try Lit (f (map litVal rands’))
      with EvalError e -> PrimApp(op, rands’)
    else PrimApp(op, rands’)
  | If(tst, thn, els) ->
    (match peval tst env with
     Lit (Bool true) -> peval thn env
     Lit (Bool false) -> peval els env
     | tst’ -> If(tst’, peval thn env, peval els env))
  | Bind(name, defn, body) ->
    (match peval defn env with
     Lit v -> peval body (Env.bind name v env)
     | defn’ -> Bind(name, defn’, peval body env))
  | App(fname, rands) -> App(fname, map (flip peval env) rands)
end

Figure 3: FOFL partial evaluator written in OCAML.
hoilic> (bind a 0 (+ (begin (<- a (+ a 1)) a) (begin (<- a (+ a 1)) a)))
3

hoilic> (bind a 0 (* 2 (begin (<- a (+ a 1)) a)))
2
c. \((+ E_1 E_2) \implies (+ E_2 E_1)\)

i. HOF: The answer for this problem depends on whether different errors are observably distinct from each other and from infinite loops. If errors are observable, then this transformation is unsafe, since the order of \(E_1\) and \(E_2\) can affect which errors are observed, or whether an error or an infinite loop is observed. Suppose that \texttt{loop} denotes a nullary function that loops infinitely when called.

hoilc> (+ (loop) (* (/ 1 0) (head (null))))
; loops infinitely

hoilc> (+ (* (/ 1 0) (head (null))) (loop))
; divide-by-zero error

hoilc> (+ (* (head (null)) (/ 1 0)) (loop))
; head-of-empty-list error

However, in some models of functional programming, errors are indistinguishable from infinite loops. In such models, the transformation is safe and all three of the above examples are indistinguishable.

By the way, here are two implementations of \texttt{loop}:

;; Recursive version
(def loop (fun () (loop)))

;; Non-recursive version
(def loop (fun () ((fun (x) (x x)) (fun (x) (x x)))))

ii. HOILIC: unsafe because side effects don’t commute in general:

hoilic> (bind a 3 (+ (begin (<- a (+ a 1)) a) (begin (<- a (* a 2)) a)))
=> 8

hoilic> (bind a 3 (+ (begin (<- a (* a 2)) a) (begin (<- a (+ a 1)) a)))
=> 7
d. \((+ E_1 E_2) \implies (bind x E_1 (+ x E_2))\)

i. HOF: unsafe because of name capture:
\[ \text{hofl}\> (\text{bind } x 1 (+ 17 x)) \]
\[ 18 \]

\[ \text{hofl}\> (\text{bind } x 1 (\text{bind } x 17 (+ x x))) \]
\[ 34 \]

ii. **HOILIC**: unsafe because of name capture (same example as for HOFL).

e. \((+ E_1 E_2) \implies (\text{bindpar } ((x \> (\text{fun } () E_1)))
\>(y \> (\text{fun } () E_2))) \>
(+ (x) (y)))

i. **HOFL**: safe because thunking circumvents name capture.

ii. **HOILIC**: safe because thunked expressions are evaluated same number of times in same order as in original.

f. \((\text{if true } E_1 E_2) \implies E_1\)

i. **HOFL**: safe because of meaning of if.

ii. **HOILIC**: safe because of meaning of if.

g. \((\text{if } E_1 E_2 E_2) \implies E_2\)

i. **HOFL**: unsafe because termination is not preserved.

\[ \text{hofl}\> (\text{if (loop) 1 1); never terminates} \]

\[ \text{hofl}\> 1 \]
\>(; terminates with 1)

ii. **HOILIC**: unsafe for same reason as HOFL.

h. \((\text{if if } E_1 E_2 E_3 E_4 E_5) \implies (\text{if } E_1 (\text{if } E_2 E_4 E_5) (\text{if } E_3 E_4 E_5))\)

i. **HOFL**: safe because of meaning of if and the preservation of relative order of expressions.

ii. **HOILIC**: safe because of meaning of if and the preservation of relative order of expressions.
Group Problem 2 [24]: Parameter Passing

These solutions do not yet include any environment diagrams.

You were asked to consider the following HOCIC expression:

```
(bind a 1
  (bind inc! (fun () (seq (<- a (+ a 1)) a))
  (bind f (fun (y z)
    (seq (<- y (+ y 3))
    (+ a (* z z))))
  (f a (inc!))))))
```

Here is a brief explanation for the value returned in each of the four parameter-passing mechanisms:

- **Call-by-value returns 6:** In `(f a (inc))`, `a` is initially 1 and `(inc!)` returns 2 after setting `a` to 2. So when the body of `f` is evaluated, `y` is 1 and `z` is 2. The assignment `(<- y (+ y 3))` is irrelevant, and `(+ a (* z z))` is returned. Since `a` is now 2 and `z` is 2, this evaluates to `(+ 2 (* 2 2)) = 6`.

- **Call-by-reference returns 9:** For this expression, call-by-reference is just like call-by-value except that the `y` in the body of `f` names the same cell as `a`. So `(<- y (+ y 3))` changes `a` from 2 to 5, and `(+ a (* z z)) = (+ 5 (* 2 2)) = 9`.

- **Call-by-name returns 7:** In call-by-name, `(f a (inc!))` is equivalent to `(+ a (* (inc!) (inc!)))` (since the assignment `(<- y ...)` can be safely ignored). Since no increments have been performed yet, `a` is 1, the first `(inc!)` returns 2 and the second `(inc!)` returns 3. So the result is `(+ 1 (* 2 3)) = 7`.

- **Call-by-lazy returns 5:** Call-by-lazy is just like call-by-name except that the second `((inc!))` returns the same result as the first `(inc!)` due to memoization. So the result is `(+ 1 (* 2 2)) = 5`. 
Group Problem 3 [25]: Counters
You were given the following code:

```
(def make-counter1
 (bind count 0
      (fun ()
        (fun () (seq (<- count (+ count 1)) count))))))

(def make-counter2
 (fun ()
      (bind count 0
        (fun () (seq (<- count (+ count 1)) count))))))

(def make-counter3
 (fun ()
      (fun () (bind count 0
              (seq (<- count (+ count 1)) count))))))

(def test-counter
 (fun (make-counter)
      (bindseq ((a (make-counter))
                (b (make-counter)))
                (list (a) (b) (a)))))
```

Each of the `make-counter` procedures is a thunk (the outer `fun`) that creates a new counter when invoked. Each counter (the inner `fun`) increments a `count` variable when invoked. In each case, the outer `fun` corresponds to a Java class declaration, while the inner `fun` corresponds to a Java instance method declaration. The behavior of the counters depends on how the `count` variable is shared.

a. [15] `make-counter1`

In `make-counter1`, the `count` variable is shared by all counters created by invoking `make-counter1`. Here the `count` variable is analogous to a class variable in Java.

Fig. 4 is an environment diagram showing the environments and closures created during the evaluation of `(def make-counter1 ...), (def test-counter ...), and (test-counter make-counter1)`. Both `test-counter` and `make-counter1` are defined in an environment frame `F1` created by the implicit `bindrec` for top-level definitions. The definition `(def test-counter ...)` binds the name `test-counter` in `F1` to (an implicit cell containing) a closure `C1` closed over `F1`. The definition `(def make-counter1 ...)` binds the name `make-counter1` in `F1` to (an implicit cell containing) the result of the `(bind count 0 ...)` expression. Evaluating this `bind` expression first creates an environment frame `F2` with a binding of `count` to (an implicit cell containing) 0, and then returns a closure `C2` that is closed over this frame `F2`. So `C2` is the contents of the cell named `make-counter1` in frame `F1`.

Invoking closure `C1` on closure `C2` creates a new frame `F3` with a binding of `make-counter` to (an implicit cell containing) the closure `C2`. Evaluating the `bindseq` in the body of `C1` invokes `C2` twice, creating empty frames `F4` and `F6` and closures `C3` and `C4` closed over these frames, respectively. (Implicit cells containing) the resulting closures are named `a` and `b`, respectively, in frames `F5` and `F7`. Evaluating `(list (a) (b) (a))` in `F7` creates empty frames `F8`, `F9`, and `F10`, each of which modifies the same implicit `count` cell in `F2` by incrementing it. So
the result returned is (list 1 2 3). Note how environment chains \( F_8 \rightarrow F_4 \rightarrow F_2 \rightarrow F_1, \) 
\( F_9 \rightarrow F_4 \rightarrow F_2 \rightarrow F_1, \) and \( F_{10} \rightarrow F_6 \rightarrow F_2 \rightarrow F_1 \) match the nesting structure of contours in the definition of \textit{make-counter1}.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{environment_diagram.png}
\caption{Environment diagram for (\textit{test-counter make-counter1}).}
\end{figure}

\textit{make-counter2}

In \textit{make-counter2}, each counter has its own \texttt{count} variable that is shared by all invocations of the counter. Here the \texttt{count} variable is analogous to an instance variable in Java.

Fig. 5 is an environment diagram showing the environments and closures created during the evaluation of (\texttt{def test-counter ...}), (\texttt{def make-counter2 ...}), and (\texttt{test-counter make-counter2}). Similar to the first example, both \texttt{test-counter} and \texttt{make-counter2} are defined in an environment frame \( F_1 \) created by the implicit \texttt{bindrec} for top-level definitions and \texttt{test-counter} is
bound in $F_1$ to (an implicit cell containing) a closure $C_1$ closed over $F_1$. However, in this case, the closure to which the counter maker `make-counter2` refers is closed over the same environment $F_1$ in which `make-counter2` is bound.

![Environment diagram for (test-counter make-counter2).](image)

Invoking closure $C_1$ on closure $C_2$ creates a new frame $F_2$ with a binding of `make-counter` to (an implicit cell containing) closure $C_2$. Evaluating the `bindseq` in the body of $C_1$ in environment $F_2$ invokes $C_2$ twice:

- The first time $C_2$ is invoked, an empty frame $F_3$ is created, and evaluating `(bind count 0 ...)` in this frame creates a frame $F_4$ with `count` bound to a cell containing 0. The result of this `bind` is a closure $C_3$ closed over $F_4$, and this closure is stored in a cell named `a` within the new frame $F_5$. The parent of $F_5$ is $F_2$ because the first definition of the `bindseq` is evaluated in $F_2$.

- The second time $C_2$ is invoked, an empty frame $F_6$ is created, and evaluating `(bind count 0 ...)` in this frame creates a frame $F_7$ with `count` bound to a cell containing 0. The result of this `bind` is a closure $C_4$ closed over $F_7$, and this closure is stored in a cell named `b` within
the new frame \( F_8 \). The parent of \( F_8 \) is \( F_5 \) because the second definition of the \texttt{bindseq} is evaluated in the environment created for the first definition.

Evaluating \((\text{list } (a) (b) (a))\) in \( F_8 \) creates empty frames \( F_9, F_{10}, \) and \( F_{11} \); the calls to \( a \) modify the \texttt{count} cell in \( F_4 \) while the call to \( b \) modifies the \texttt{count} cell in \( F_7 \). So the result returned is \((\text{list } 1 1 1)\).

Note how environment chains \( F_9 \rightarrow F_4 \rightarrow F_3 \rightarrow F_1, F_{10} \rightarrow F_4 \rightarrow F_3 \rightarrow F_1, \) and \( F_{11} \rightarrow F_7 \rightarrow F_6 \rightarrow F_1 \) match the nesting structure of contours in the definition of \texttt{make-counter2}.

\texttt{make-counter3}

In \texttt{make-counter3}, each invocation of the counter creates a fresh \texttt{count} variable, which is never shared with any other invocation. Here the \texttt{count} variable is analogous to a local variable in a Java execution frame.

Fig. 6 is an environment diagram showing the environments and closures created during the evaluation of \((\text{def test-counter ...}), (\text{def make-counter3 ...}),\) and \((\text{test-counter make-counter3})\).

Similar to the second example, both \texttt{test-counter} and \texttt{make-counter2} are bound in an environment frame \( F_1 \) created by the implicit \texttt{bindrec} for top-level definitions and both are bound to (implicit cells containing) closures \( C_1 \) and \( C_2 \), respectively, both of which are closed over \( F_1 \).

Invoking closure \( C_1 \) on closure \( C_2 \) creates a new frame \( F_2 \) with a binding of \texttt{make-counter} to (an implicit cell containing) closure \( C_2 \). Evaluating the \texttt{bindseq} in the body of \( C_1 \) in frame \( F_2 \) invokes \( C_2 \) twice:

- The first time \( C_2 \) is invoked, an empty frame \( F_3 \) is created, and closure \( C_3 \) closed over \( F_3 \) is returned and stored in a cell named \( a \) within the new frame \( F_4 \). The parent of \( F_4 \) is \( F_2 \) because the first definition of the \texttt{bindseq} is evaluated in \( F_2 \).
- The second time \( C_2 \) is invoked, an empty frame \( F_5 \) is created, and closure \( C_4 \) closed over \( F_5 \) is returned and stored in a cell named \( a \) within the new frame \( F_6 \). The parent of \( F_6 \) is \( F_4 \) because the second definition of the \texttt{bindseq} is evaluated in the environment created for the first definition.

Evaluating \((\text{list } (a) (b) (a))\) in \( F_6 \) creates empty frames \( F_7, F_9, \) and \( F_{11} \). Evaluating \((\text{bind count } 0 ... )\) in these frames creates, respectively, frames \( F_8, F_{10}, \) and \( F_{12}, \) each of which has a variable \texttt{count} bound to a distinct cell initially holding \( a \). The invocations \( (a), (b), (a) \) each increment the \texttt{count} variable in a different cell, so the result returned is \((\text{list } 1 1 1)\).

Note how environment chains \( F_8 \rightarrow F_7 \rightarrow F_3 \rightarrow F_1, F_{10} \rightarrow F_9 \rightarrow F_3 \rightarrow F_1, \) and \( F_{12} \rightarrow F_{11} \rightarrow F_5 \rightarrow F_1 \) match the nesting structure of contours in the definition of \texttt{make-counter3}.

b. [10] A fleshed-out version of \texttt{Counters.java} is presented in Fig. 7. As suggested above, in \texttt{Counter1}, \texttt{count} is modeled by a class variable; in \texttt{Counter2}, \texttt{count} is modeled by an instance variable; and in \texttt{Counter2}, \texttt{count} is modeled by a local variable. Note that the expression sequence \( \texttt{count = count + 1: return count;} \) can be expressed more concisely as \( \texttt{return ++count;} \), where \( \texttt{++count} \) increments the \texttt{count} variable and returns the value \texttt{after} the increment.
Figure 6: Environment diagram for (test-counters make-counter3).
interface Counter {
    public int invoke();
}

class Counter1 implements Counter {
    private static int count = 0;
    public int invoke() {
        count = count + 1;
        return count;
    }
}

class Counter2 implements Counter {
    private int count = 0;
    public int invoke() {
        count = count + 1;
        return count;
    }
}

class Counter3 implements Counter {
    public int invoke() {
        int count = 0;
        count = count + 1;
        return count;
    }
}

public class Counters {
    public static IntList IL;

    public static IntList testCounters(Counter a, Counter b) {
        return IL.prepend(a.invoke(),
            IL.prepend(b.invoke(),
                IL.prepend(a.invoke(),
                    IL.empty())));
    }

    public static void main (String [] args) {
        System.out.println("testCounters(new Counter1(), new Counter1())=
            + testCounters(new Counter1(), new Counter1()));
        System.out.println("testCounters(new Counter2(), new Counter2())=
            + testCounters(new Counter2(), new Counter2()));
        System.out.println("testCounters(new Counter3(), new Counter3())=
            + testCounters(new Counter3(), new Counter3()));
    }
}

Figure 7: Fleshed out version of Counters.java.
Group Problem 4 [35]: Manually Converting OCAML to JAVA

Figs. 8 and 9 show the translation of the OCAML process function into JAVA. Nested function declarations are translated to “flat” ones by adding extra arguments, and closures are translated to instances of classes satisfying the IntFun interface.

Rather than defining new class IdFun, AddFun, and MulFun, an alternative is to create these classes “on the fly” using anonymous inner classes. Below are the alternative definitions of process, scan1, and scan2 using anonymous inner classes:

```java
// Define process here:
public static IntList process (IntList xs) {
    return scan1 (xs, xs, new IntFun() {public int apply (int a) {return a;}});
}

public static IntList scan1 (IntList xs, IntList ys, final IntFun f) {
    if (IL.isEmpty(ys)) {
        return mapxs(xs,f);
    } else {
        int y = IL.head(ys);
        if ((y % 2) == 0) {
            return scan2(xs,IL.tail(ys),f);
        } else {
            final int yprime = y/2;
            return IL.prepend(yprime,
                scan1(xs,IL.tail(ys),
                    new IntFun() {
                        public int apply (int a) {
                            return f.apply(a + yprime);
                        }
                    }));
        }
    }
}

public static IntList scan2 (IntList xs, IntList zs, final IntFun g) {
    if (IL.isEmpty(zs)) {
        return mapxs(xs,g);
    } else {
        int z = IL.head(zs);
        if ((z % 2) == 0) {
            return scan1(xs,IL.tail(zs),g);
        } else {
            final int zprime = z/2;
            return IL.prepend(zprime,
                scan2(xs,IL.tail(zs),
                    new IntFun() {
                        public int apply (int a) {
                            return g.apply(a * zprime);
                        }
                    }));
        }
    }
}
```
// Interface for int -> int functions
interface IntFun {
    public int apply (int x);
}

// The identity function
class IdFun implements IntFun {
    public int apply (int x) {
        return x;
    }
}

// Put other classes implementing the IntFun interface here:
class AddFun implements IntFun {
    private IntFun f;
    private int n;

    public AddFun (IntFun f, int n) {
        this.f = f;
        this.n = n;
    }

    public int apply (int a) {
        return f.apply(a + n);
    }
}

class MulFun implements IntFun {
    private IntFun g;
    private int n;

    public MulFun (IntFun g, int n) {
        this.g = g;
        this.n = n;
    }

    public int apply (int b) {
        return g.apply(b * n);
    }
}

// The Process class defines the process, scan1, scan2, mapxs and mapq methods.
public class Process {
    // Handy way of introducing abbreviation IL. for IntList operations:
    // IL.empty(), IL.isEmpty, IL.head(), IL.tail, IL.prepend.
    public static IntList IL;

    public static IntList process (IntList xs) {
        return scan1 (xs, xs, new IdFun());
    }

Figure 8: Java translation of the OCAML process function, Part 1
public static IntList scan1 (IntList xs, IntList ys, IntFun f) {
    if (IL.isEmpty(ys)) {
        return mapxs(xs,f);
    } else {
        int y = IL.head(ys);
        if ((y % 2) == 0) {
            return scan2(xs,IL.tail(ys),f);
        } else {
            int yprime = y/2;
            return IL.prepend(yprime, scan1(xs,IL.tail(ys), new AddFun(f,yprime)));
        }
    }
}

public static IntList scan2 (IntList xs, IntList zs, IntFun g) {
    if (IL.isEmpty(zs)) {
        return mapxs(xs,g);
    } else {
        int z = IL.head(zs);
        if ((z % 2) == 0) {
            return scan1(xs,IL.tail(zs),g);
        } else {
            int zprime = z/2;
            return IL.prepend(zprime, scan2(xs,IL.tail(zs), new MulFun(g,zprime)));
        }
    }
}

public static IntList mapxs (IntList xs, IntFun q) {
    return mapq(q,xs);
}

public static IntList mapq (IntFun q, IntList ws) {
    if (IL.isEmpty(ws)) {
        return IL.empty();
    } else {
        return IL.prepend(q.apply(IL.head(ws)), mapq(q,IL.tail(ws)));
    }
}

// Testing method. E.g.:
// [lyn@jaguar ps5] java Process [3,4,5,6,7]
// [1, 2, 3, 13, 15, 17, 19, 21]
// [lyn@jaguar ps5] java Process [5,7,4,5,7]
// [2, 3, 2, 35, 47, 29, 35, 47]
public static void main (String [] args) {
    if (args.length == 1) {
        System.out.println(process(IntList.fromString(args[0])));
    } else {
        System.out.println("unrecognized main option");
    }
}