Problem Set 6 Solutions

Individual Problem [40]: Going Loopy

a. [5]: Variable Scoping Figure 1 shows a copy of the given expression with lines from each variable reference occurrence to its associated declaration (i.e., binding occurrence). Free variables are circled.

b. [20]: Implementing loop in OCaml

i [2]: freeVarsExp The scope of the bound variables of a loop expression includes the updater expressions, test expression, and body expression, but does not include the initializer expressions. This means that the free variables of a loop expression is the union of (1) the free variables of the initializer expressions and (2) the result of subtracting off the loop-bound variables from the updaters, test, and body. This can be expressed in OCAML code by extending the freeVarsExp function with the following clause:

```ocaml
| Loop(vars,inits,updates,test,body) ->
  S.union (freeVarsExps inits)
  (S.diff (freeVarsExps (body :: test :: updates))
   (listToSet vars))
```

ii [4]: subst When performing substitution on a loop expression, variable capture can be avoided by renaming loop-bound variables in their scope – i.e., within the updater, test, and body expressions. This is expressed in OCAML by extending the subst function with the following clause:

```ocaml
| Loop(vars,inits,updates,test,body) ->
  let vars' = map StringUtils.fresh vars in
  Loop(vars',
    map (flip subst env) inits,
    map (flip subst env) (map (renameAll vars vars') updates),
    subst (renameAll vars vars' test) env,
    subst (renameAll vars vars' body) env)
```

iii [8]: Environment model eval You were asked to flesh out the following skeleton for the evaluation of loop expressions in LoopexEnvInterp.eval:

```ocaml
| Loop(vars, inits, updates, test, body) ->
  eval body (iterate E_1 E_2 E_3)
```

Because the invocation of iterate appears as the second argument to eval, it must return an environment. This means that the seed expression $E_3$ must be an environment representing the initial state of the iteration, $E_1$ must be a function that updates one environment to the next, and $E_2$ must be a predicate on environments that returns true when the iteration is done.

The initial environment $E_3$ must extend the given environment with bindings that bind each state variable to its initial value:
Figure 1: Wiring diagram showing the scope of variables in the expression from part (a).
Env.bindAll vars (map (flip eval env) inits) env

Here (flip eval env) is just a concise way of writing (fun init -> eval init env). One error commonly made in writing $E_3$ is to forget to use eval to evaluate each of the expressions in inits. The OCAML type checker will catch this because Env.bindAll expects to associate vars with a list of values, not a list of expressions. An error that the type checker will not catch is creating an environment that contains only bindings for the state variables rather than extending the surrounding environment env with these bindings. This is problematic because then the variables of env are not available for evaluating body.

On each iteration, the current environment is updated by extending it with updated values for the state variables:

$$E_1 = (\text{fun } e \rightarrow \text{Env.bindAll vars (map (flip eval e) updates) env})$$

The same result is obtained whether the base environment extended by Env.bindAll is env or e.

The iteration should stop when the continuation test of the loop expression is false:

$$E_2 = (\text{fun } e \rightarrow (\text{eval test e}) = \text{Bool false})$$

Note that the loop expression is controlled by a continuation test (i.e., terminates when the test is false) while the iterate iteration is controlled by a termination test (i.e., terminates when the test is true). It took many students a long time to appreciate that the sense of these two tests are inverted. Also, converting between LOOPEx booleans and OCAML booleans proved to be a hurdle for many students. An alternative way to perform this conversion is to use an explicit match:

$$\text{(fun } e \rightarrow \text{match eval test e with } (\text{Exp2 } \ast))$$

$$\text{Bool false } \rightarrow \text{true}$$

$$\_ \rightarrow \text{false}$$

Here is the fully fleshed out eval clause for loop:

$$\text{| Loop(vars, inits, updates, test, body) }\rightarrow\text{ eval body}$$

$$\text{let state } = E_1 \text{ in}$$

$$\text{match eval (substAll state vars test) with}$$

$$\text{Bool false } \rightarrow \text{eval } E_2$$

$$\_ \rightarrow \text{eval } E_3$$

iv [6]: Substitution model eval You were asked to flesh out the following skeleton:

$$\text{| Loop(vars, inits, updates, test, body) }\rightarrow\text{ let state } = E_1 \text{ in}$$

$$\text{match eval (substAll state vars test) with}$$

$$\text{Bool false } \rightarrow \text{eval } E_2$$

$$\_ \rightarrow \text{eval } E_3$$

Note that state is used as the first argument to substAll, which expects an list of expressions in this position. Thus, $E_1$ must evaluate to a list of expressions — in particular, a list of literal expressions holding the values that result from evaluating inits:

$$E_1 = \text{map (fun } v \rightarrow \text{Lit v) (map eval inits)}$$

The incorrect solution $E_1 = \text{inits}$ has the right type and will work in situations where
inits happens to be a list of literal expressions. But it is incorrect in the case where inits
contains non-literal expressions. In such cases, it implements call-by-name semantics for the
expressions rather than call-by-value semantics. To see that these can differ, imagine that
one of the state variables is not actually used in the iteration and that its corresponding
initialization or update expression loops forever. Then under the call-by-value evaluation
semantics of LOOPEX, the loop should never terminate, while under call-by-name semantics
it can terminate.

$E_2$ appears in a context where the valu that is the final value of the loop expression is
expected. This can be obtained by evaluating the result of substituting the final state into
the body expression:

$$E_2 = \text{eval (substAll state vars body)}$$

$E_3$ appears in a context that also returns a valu: the value of the loop after the next iteration.
This can be expressed by evaluating a new loop expression whose initial values are the result
of evaluating the updater expressions relative to the current state:

$$E_3 = \text{eval (Loop(vars, map (substAll state vars) updates, updates, test, body))}$$

Here is the fully fleshed out eval clause for loop:

$$\text{Loop(vars, inits, updates, test, body)} \rightarrow$$

let state = map (fun v -> Lit v) (map eval inits) (* Exp1 *) in
match eval (substAll state vars test) with
  Bool false -> eval (substAll state vars body) (* Exp2 *)
| _ -> eval (Loop(vars, map (substAll state vars) updates, updates, test, body)) (* Exp3 *)

\[c.\, [10]: \text{Desugaring least into loop}\]

You were given the following desugaring rule for least:

$$(\text{least} \ I \ var \ E \ pred) \leadsto (\text{loop} \ ((I \ var \ 0 \ (+ \ I \ var \ 1))) \ (\text{not} \ E \ pred) \ I \ var)$$

\[i \ [2]\] Here is an English description of the least construct:

$(\text{least} \ I \ var \ E \ pred)$ returns the least natural number (i.e., non-negative integer) var
such that body evaluates to true.

\[ii \ [6]\] What are the values of the following expressions using least? Show your work in
order to receive partial credit.

* $(\text{least} \ x \ (> \ (* \ x \ x) \ 100))$
  The value of the above express is 11, since 11 is the least natural number whose square
  is greater than 100.

* $(\text{least} \ i \ (> \ (* \ i \ (\text{least} \ j \ (\leq \ (/ \ 100 \ (+ \ j \ 1)) \ i)) \)))$
  Consider the value of the product $(* \ i \ (\text{least} \ j \ ...))$ as a function of $i$, starting with
  $i = 0$:
  - If $i = 0$, then the product is 0, which is certainly not $\geq 80$.
  - If $i = 1$, then the least $j$ such that $(\leq \ (/ \ 100 \ (+ \ j \ 1)) \ i)$ is 50. $1 \cdot 50 = 50$,
    which is not $\geq 80$. 

4
\[ i = 2, \text{ then the least } j \text{ such that } (\leq (\frac{1}{100} (j + 1)) i) \text{ is } 33. \ 2 \cdot 33 = 66, \text{ which is not } \geq 80. \]
\[ i = 3, \text{ then the least } j \text{ such that } (\leq (\frac{1}{100} (j + 1)) i) \text{ is } 25. \ 3 \cdot 25 = 75, \text{ which is not } \geq 80. \]
\[ i = 4, \text{ then the least } j \text{ such that } (\leq (\frac{1}{100} (j + 1)) i) \text{ is } 20. \ 4 \cdot 20 = 80, \text{ which is } \geq 80. \text{ So } i = 4 \text{ is the least value of } i \text{ satisfying the predicate.} \]

i. [2] The key advantage of implementing \texttt{least} as syntactic sugar rather than as a kernel construct is that the only component of the \textsc{loopex} implementation which needs to be extended is the desugarer. The parser, unparser, free-variables computation, substitution function, and the substitution-model and environment-model evaluators need not be extended.

d. [10]: Desugaring \texttt{simprec} into \texttt{loop}  

i [6] As part of implementing any desugaring, it is first important to develop a rule that explains how the desugaring should rewrite a given expression. Here is a first cut at desugaring \texttt{simprec} that is close to correct but not quite right:

\[
\begin{align*}
& (\texttt{simprec } E_{\text{zero}} \ (I_{\text{num}} \ I_{\text{ans}} \ E_{\text{combine}}) \ E_{\text{arg}}) \\
& \sim \\
& (\texttt{loop} ((I_{\text{num}} \ E_{\text{arg}} \ (- \ I_{\text{num}} \ 1))) \\
& \quad (I_{\text{ans}} \ E_{\text{zero}} \ E_{\text{combine}}) \\
& \quad (> \ I_{\text{num}} \ 0)) \\
& \quad I_{\text{ans}}))
\end{align*}
\]

The desugaring uses the given variable \( I_{\text{num}} \) to iterate through the integers from \( E_{\text{arg}} \) down to 1, and uses the given variable \( I_{\text{ans}} \) to maintain a running answer that starts at \( E_{\text{zero}} \) and is updated each iteration by using \( E_{\text{combine}} \) to combine \( I_{\text{num}} \) and \( I_{\text{ans}} \) at each iteration.

The above iteration actually works for the factorial and Fibonacci examples in the problem description. So what’s wrong with it? The problem is that it only works for combination operations in \( E_{\text{combine}} \) that are both commutative and associative. In general, the desugaring will combine elements in the wrong order. For example, consider the \textsc{loopex} program

\[
(\texttt{loopex} \ (x) \ (\texttt{simprec} \ #e \ (y \ ys \ (\texttt{prep} \ y \ ys)) \ x))
\]

This program should return a list of integers from the value of \( x \) down to 1. But the buggy desugaring given above will return a list of integers from 1 up to the value of \( x \). This is the reverse of the correct answer!

To combine elements in the correct order, it is necessary for the loop to start from 1 and count up to the value of \( E_{\text{arg}} \). Here is an improved version of the desugaring that does this:

\[
\begin{align*}
& (\texttt{simprec } E_{\text{zero}} \ (I_{\text{num}} \ I_{\text{ans}} \ E_{\text{combine}}) \ E_{\text{arg}}) \\
& \sim \\
& (\texttt{loop} ((I_{\text{num}} \ 1 \ (* \ I_{\text{num}} \ 1)) \\
& \quad (I_{\text{ans}} \ E_{\text{zero}} \ E_{\text{combine}}) \\
& \quad (\leq \ I_{\text{num}} \ E_{\text{arg}})) \\
& \quad I_{\text{ans}}))
\end{align*}
\]

This desugaring gives the correct answer, but will evaluate \( E_{\text{arg}} \) multiple times if at least one iteration of the \texttt{loop} is made. But the problem specification requires that \( E_{\text{arg}} \) be evaluated exactly once.
To satisfy the problem specification, we can evaluate $E_{\text{arg}}$ once before the loop, name this value with a fresh name $I_{\text{arg}}$, and use $I_{\text{arg}}$ within the loop. This yields the final (correct) version of the desugaring:

$$(\text{simprec } E_{\text{zero}} \ (I_{\text{num}} \ I_{\text{ans}} \ E_{\text{combine}}) \ E_{\text{arg}})$$

$$(\text{bind } I_{\text{arg}} \ E_{\text{arg}} \ ; \ I_{\text{arg}} \ \text{fresh}$$

$$(\text{loop} \ ((I_{\text{num}} \ 1 \ (+ \ I_{\text{num}} \ 1))$$

$$(I_{\text{ans}} \ E_{\text{zero}} \ E_{\text{combine}}))$$

$$ (<= I_{\text{num}} \ I_{\text{arg}})$$

$$I_{\text{ans}}))$$

Note that $I_{\text{arg}}$ must be fresh rather than a fixed variable name. Any fixed name could lead to variable capture problems.

ii [4] The above rule can be expressed in OCAML by extending the desguarRules function with the following clause

```ocaml
| Seq [Sym "simprec"; zerox; Seq [Sym inum; Sym ians; combinex]; argx] ->
  let iarg = StringUtils.fresh "arg" in
  Seq [Sym "bind";;
    Sym iarg;
    argx;
    Seq [Sym "loop";
        Seq[Seq[Sym inum; Sexp.Int 1; Seq[Sym "+"; Sym inum; Sexp.Int 1]];
            Seq[Sym ians; zerox; combinex]];
        Seq[Sym "<<"; Sym inum; Sym iarg];
        Sym ians]];
```

**Group Problems**

**Group Problem 1 [25]: Static and Dynamic Scope in HOFL**

In this problem, you considered the following HOFL program:

$$\text{(hofl (a)}$$

$$\text{ (bind linear (fun (a b))}$$

$$\text{ (fun (x)}$$

$$\text{ (+ (* a x) b)))}$$

$$\text{ (bind line1 (linear 1 2)}$$

$$\text{ (bind line2 (linear 3 4)}$$

$$\text{ (bind try (fun (b)}$$

$$\text{ (prep (line1 b)}$$

$$\text{ (prep (line2 (+ b 1))}$$

$$\text{ (prep (line2 (+ b 2))}$$

$$\text{ #e))})$$

$$\text{ (try (+ a a))))))))$$

*a.* [10] The environment diagram for the above program in statically scoped HOFL is shown in Fig. 2. Note that under static scoping, each function is represented as a closure that pairs an abstraction together with an environment that specifies the meaning of the free variables in the abstraction. The chains of environment frames have a structure that mirrors the nesting of lexical contours in the program. For example, in the program, the contour for $(\text{fun (x) ...}$ is nested inside the contour for $(\text{fun (a b) ...}$, which is nested inside the contour for $(\text{hofl (a) ...}$. This nesting structure is reflected in the environment chains $F9 \rightarrow F3 \rightarrow F1$, $F10 \rightarrow F5 \rightarrow F1$, and $F11 \rightarrow F5 \rightarrow F1$. The chain $F8 \rightarrow F6 \rightarrow F4 \rightarrow F2 \rightarrow F1$ reflects the nesting of
(fun (b) ...) within the bind expressions for line2, line1, and linear and the top-level program.

b. [1] When the program is run on 5 in statically scoped HOFL, the result is the list (list 12 37 40).

c. [6] The environment diagram for the above program in dynamically scoped HOFL is shown in Fig. 3. Here the structure of the environment frames has the shape of the invocation tree for the program. Note that under dynamic scoping, each function is represented simply as an abstraction and so cannot “remember” the values of its free variables at the point where the function is created. Instead, such free variables are resolved relative to the environment where the function is invoked. In particular, the body expression (+ (* a x) b) of both line1 and line2 is always evaluated relative to an environment in which a is the program parameter 5 and b is the argument 10 to try.

d. [1] When the program is run on 5 in statically scoped HOFL, the result is the list (60 65 70).

e. [2] Modularity is the principle that programs should be composed out of reusable parts that can be combined in mix-and-match ways. In order for the parts to be truly mix-and-match, the behavior of a part should not depend on inessential details of the other parts which which it is combined.

Lexical scoping supports modularity better than dynamic scoping because the meaning of a free variable name within the body of a higher-order function does not depend on the naming context where the function is called. In particular, in lexically scoped programs, the particular names chosen for variables don’t matter, because it is always possible to consistently rename variables.

In contrast, in dynamical scoping, renaming a free variable in a function body can make a working program stop working because of name capture problems. To avoid such problems, programmer working on the same project have to agree on naming conventions; this seriously detracts from the notion of functions as black-box abstractions. Furthermore, dynamical scoping seriously impairs the power associated with the ability to return functions as values from other functions. Without a means of capturing an environment, functions with free variables cannot reliably be used for a host of important applications, such as encoding data structures (think of the church pairs and church sets we studied earlier this semester) and object-oriented programming.

f. [5] Here is a HOFL expression that evaluates to #t for a statically-scoped interpreter but evaluates to #f for a dynamically-scoped interpreter:

\[
\text{(bind b #t}
\text{(bind f (fun () b)}
\text{  (bind b #f)}
\text{  (f))})
\]

In static scope, the free b in (fun () b) always refers to the outermost declaration of b, so (f) always returns #t. In dynamic scope, the free b refers to whichever declaration of b appears most recently in the call chain. In the above example, (f) is wrapped in the declaration (bind b #f ...), so (f) returns #f.

Group Problem 2 [15]: bindrec
You were asked to consider the following HOFL expression E:
Figure 2: Environment diagram for the static \texttt{linear} example.
Figure 3: Environment diagram for the dynamic linear example.
(bind f (abs x (+ x 1)))
(bindrec ((f (abs n
    (if (= n 0)
      1
      (* n (f (- n 1)))))))
(f 3)))

a. [5] Fig. 4 shows an environment diagram for the case where $E$ is evaluated in \textit{statically scoped} HoFL. Assume that FO is the global frame in which $E$ is evaluated. Because \texttt{bindrec} is recursively scoped, the environment of closure $C_2$ points to the frame (F2) that contains the name $f$ bound to the closure. Due to static scoping, all four frames binding $n$ point to F2 as well. The value of of $E$ in statically scoped HoFL is 6 because the inner $f$ denotes a recursive factorial function.

Figure 4: Environment diagram for the statically-scoped \texttt{bindrec} example.

b. [4] $E'$ is obtained from $E$ by replacing \texttt{bindrec} by \texttt{bindseq}. Fig. 5 shows an environment diagram for the case where $E'$ is evaluated in \textit{statically scoped} HoFL. Because \texttt{bindrec} is changed to \texttt{bindseq}, the closure $C_2$ now points to frame F1 rather than F2, so the free $f$ in the body of $(\texttt{abs n} \ldots)$ refers to the outer $f$, not the inner $f$. The value of $E'$ in statically scoped HoFL is 9, because the $f$ in the body of $(\texttt{abs n} \ldots)$ refers to the outer $f$, not the inner one.

c. [4] Fig. 5 shows an environment diagram for the case where $E'$ is evaluated in \textit{dynamically scoped} HoFL. In dynamic scoped, function names are bound to abstractions, not closures, and new environments are built on top of the current active environment (because they follow the shape of the invocation tree). Because the $f$ in the body of $(\texttt{abs n} \ldots)$ refers to most recent dynamic binding for $f$, which is the inner one, this abstraction calculates the factorial function. So the value of $E'$ in dynamically scoped HoFL is 6.

d. [2] As indicated by the example in part (e), a dynamically scoped language does not need a recursive binding construct like \texttt{bindrec} in order to support the creation of local recursive
functions. If the body of a recursive function is evaluated in an environment created by a simple
bind, bindseq, or bindpar, the names bound by the construct are visible within the body.
Figure 6: Environment diagram for the dynamically-scoped `bindseq` example.
Group Problem 3 [60]: Translating **FOFL** to **POSTFIX**

a. [13]: **Integers and Arithmetic Operations**

In this part, you assumed that the given FOFL-- program (1) had zero parameters, (2) no function declarations, and (3) its body was formed out of only (i) integer literals and (ii) applications of the primitive arithmetic operators +, -, *, /, and %. Fig. 7 shows the code for this part. In this and all the following parts, it is helpful to define two auxiliary functions:

```ocaml
val transExps: Fofl.exp list -> PostFix.com list
  transExps es translates a list of FOFL-- expressions es into a POSTFIX command list that pushes the result of evaluating each expression on the stack.
val transPrimop: Fofl.primop -> PostFix.com list
  transPrimop p translates a FOFL-- primitive operator p into a POSTFIX command sequence that pops the operands of p off the stack (from last to first) and pushes onto the stack the result of performing p on these operands.
```

```ocaml
let rec transPgm (F.Pgm(_,body,_)) = (* assume no formals and no function definitions *)
P.Pgm(0, transExp body)

(* val transExp: Fofl.exp -> PostFix.com list *)
and transExp exp =
  match exp with
  F.Lit (F.Int i) -> [P.Int i]
| F.PrimApp (op, rands) -> (transExps rands) @ (transPrimop (F.primopName op))
| _ -> raise (TransError("Unhandled Fofl exp: " ^ (F.expToString exp)))

and transExps exps = flatten (map transExp exps)

and transPrimop op =
  match op with
  "+" -> [P.Add]
| "-" -> [P.Sub]
| "*" -> [P.Mul]
| "/" -> [P.Div]
| "%" -> [P.Rem]
| _ -> raise (TransError("Unhandled Fofl primop: " ^ op))
```

Figure 7: The FOFL-- to POSTFIX translator from part (a).
b. [12]: Booleans and Related Operations

Booleans and related operations can be added to the translator by extending the code in Fig. 7 as follows:

- Add following clauses to `transExp` to translate the boolean literals `#t` (to the integer 1) and `#f` (to the integer 0):

  ```
  | F.Lit (F.Bool false) -> [P.Int 0]
  | F.Lit (F.Bool true) -> [P.Int 1]
  ```

- Add following clause to `transExp` to translate conditional expressions:

  ```
  | F.If(test,thn,els) -> (transExp test)
  @ [P.Seq (transExp thn)]
  @ [P.Seq (transExp els)]
  @ [P.Sel; P.Exec]
  ```

  Since at most one branch of a conditional can be executed, it is necessary to delay the evaluation of the command sequences associated with the then and else branches of the conditionals. Execution of `POSTFIX` command sequence can be delayed by wrapping it in an executable sequence command that is later executed via the `exec` command. This is the `POSTFIX` equivalent of wrapping a `Hofl` expression in a thunk and later evaluating it by invoking the thunk on zero arguments. The above translation uses `P.Seq` to delay the execution of both branches, `P.Sel` to choose one of the delayed branches, and `P.Exec` to execute the chosen branch.

- Add the following clauses to `transPrimop` to translate the relational and logical operators:

  ```
  | "<" -> [P.LT]
  | "<=" -> [P.LE]
  | "=" -> [P.EQ]
  | "!=" -> [P.NE]
  | ">=" -> [P.GE]
  | ">" -> [P.GT]
  | "not" -> [P.Int(0); P.EQ]
  | "and" -> [P.Mul]
  | "or" -> [P.Add; P.Int(0); P.NE]
  | "bool=" -> [P.EQ]
  ```

  The translations of the relational operators and `bool=` are straightforward, but the translation of `not`, `and`, and `or` deserve some explanation. These three operators could be translated using the `sel` command, but there are cleverer translations that do not use `sel`. The cleverer translations all depend on the type safety of `Fofl`, which guarantees that the operands to these three operators can only be the integers 0 (for false) and 1 (for true). Negating one of these values (`not`) can be accomplished by testing if it is equal to 0. The logical conjunction of two of these values (`and`) is equivalent to multiplication: multiplying two 1s yields 1, but any multiplication involving 0 yields 0. The logical disjunction of two of these values (`or`) can be accomplished by testing if their sum is not 0. This is true (1) for any pair of logical values that has at least one 1, but is false (0) for two 0s.
c. [10]: Program Parameters

Program parameters can be handled by extending the translator as shown in Fig. 8. Here is a summary of the extensions:

- The POSTFIX program must specify a number of arguments equal to the length of the formal parameters fmls to the FOFL-- program.
- The transExp and transExps functions must be extended to take an extra argument, genv, that models the positions of the global program parameters. Since these are stored from top down at the bottom of the stack but will be indexed from the bottom up, the static environment representing these parameters is created from the reversal of the formal parameter list. Note that the global environment is never modified by the translator; it is just passed unchanged through the translator. This means that the explicit passing of the genv parameter could be avoided by reorganizing the translator to use block structure; transExp and transExps could be defined as functions nested inside a scope in which the initial genv is defined.
- The transExp function is extended with a clause to handle a variable reference. This translates a reference to a global parameter to a bget of the associated index.

```ocaml
let rec transPgm (F.Pgm(fmls,body,_)) = (* assume no function definitions *)
    let n = List.length fmls
    and genv = Senv.make (List.rev fmls) in
    P.Pgm(n, (transExp body genv))

and transExp exp genv =
    match exp with
        F.Lit (F.Int i) -> [P.Int i]
    | F.Lit (F.Bool false) -> [P.Int 0]
    | F.Lit (F.Bool true) -> [P.Int 1]
    | F.Var v -> (match Senv.lookup v genv with
           Some gindex -> [P.Int gindex; P.Bget]
        | None -> raise (TransError ("Unbound variable: " ^ v)))
    | F.PrimApp (op, rands) -> (transExps rands genv) @ (transPrimop (F.primopName op))
    | F.If(test,thn,els) -> (transExp test genv)
        @ [P.Seq (transExp thn genv)]
        @ [P.Seq (transExp els genv)]
        @ [P.Sel;P.Exec]
    | _ -> raise (TransError ("Unhandled Fofl exp: " ^ (F.expToString exp)))

and transExps exps genv = flatten (map transExp exps genv)

and transPrimop op = (* same as for part b *)
```

Figure 8: The FOFL-- to POSTFIX translator from part (c).

d. [10]: Local Bindings

Local bindings can be handled by extending the translator as shown in Fig. 9. Here is a summary of the extensions:

- The transExp and transExps functions must be extended to take an extra argument, lenv,
that models the positions of local (i.e., \texttt{bind}-bound) names. Initially, there are no local names, so the initial local static environment is empty.

- The variable reference clause within \texttt{transExp} must be modified to first look up a name in the local environment before looking it up in the global environment. A locally bound name translates to a \texttt{get} of the associated (top-based) index.

- The \texttt{transExp} function must be extended with a clause to handle \texttt{bind}. This first generates code for the definition expression. This code is followed by code for the body expression, which is translated relative to a local environment that accounts for the newly bound variable. Finally, a \texttt{swap pop} sequence is generated to remove the locally bound value from the stack before the \texttt{bind} returns. This ensures that the only effect of the commands translated from the \texttt{bind} is to push the value of the \texttt{bind} expression on the stack.

- The \texttt{transExp} function must be modified to account for the fact that the value pushed by each expression in the list increments the top-based indices of locally bound variables. This is easy to express via a recursion that performs an \texttt{Senv.push} on the local environment for each expression in the list.

```ml
let rec transPgm (F.Pgm(fmls,body,_)) = (* assume no function definitions *)
  let n = List.length fmls
  and genv = Senv.make (List.rev fmls) in
  P.Pgm(n, (transExp body Senv.empty genv))

  and transExp exp lenv genv =
    match exp with
    | F.Lit (F.Int i) -> [P.Int i]
    | F.Lit (F.Bool false) -> [P.Int 0]
    | F.Lit (F.Bool true) -> [P.Int 1]
    | F.Var v -> (match Senv.lookup v lenv with
              Some lindex -> [P.Int lindex; P.Get]
              | None -> (match Senv.lookup v genv with
                        Some gindex -> [P.Int gindex; P.Bget]
                        | None -> raise (TransError ("Unbound variable: " ^ v)))))
    | F.PrimApp (op, rands) ->
      (transExps rands lenv genv) @ (transPrimop (F.primopName op))
    | F.If(test,thn,els) -> (transExp test lenv genv)
      @ [P.Seq (transExp thn lenv genv)]
      @ [P.Seq (transExp els lenv genv)]
      @ [P.Sel;P.Exec]
    | F.Bind(name,defn,body) -> (transExp defn lenv genv)
      @ (transExp body (Senv.bind name lenv) genv)
      @ [P.Swap; P.Pop] (* remove bound variable *)
    | _ -> raise (TransError ("Unhandled Fofl exp: " ^ (F.expToString exp)))

  and transExps exps lenv genv =
    match exps with
    | [] -> []
    | e::es -> (transExp e lenv genv) @ (transExps es (Senv.push lenv) genv)

  and transPrimop op = (* same as for part b *)
```

Figure 9: The \texttt{Fofl-- to Postfix} translator from part (d).
The final version of the translator, which handles function declarations and function applications, is presented in Fig. 10. Here is a summary of the changes made to the version in part (d) for this part:

- The \texttt{transPgm} function is extended to push an executable sequence for each FOFL-- function declaration on the stack before the commands translated for the program body. We have chosen to push them in their order of definition, but we could have chosen the reverse order (as long as we are consistent in handling function references later).

- The \texttt{transFcn} auxiliary function is used to translate a single FOFL-- function declaration to a single \texttt{Postfix} executable sequence command. The commands in the sequence correspond to the evaluation of the body of the FOFL-- function under the assumption that the operands to the function can be found on the stack in reverse order. Note that the \texttt{transFcn} function must take the function environment \texttt{fenv} as an argument because FOFL-- functions have mutually recursive scope in the function namespace.

- The \texttt{transExp} and \texttt{transExps} must be extended to take a third static environment parameter, \texttt{fenv}, that models the positions of the function on the stack.

- The \texttt{transExp} function must be extended with a clause to handle function applications. A function application is translated to \texttt{Postfix} commands that (1) push the operands on the stack, (2) push the executable sequence for the function on the stack, and (3) execute the executable sequence. Since the executable sequences are stored at the bottom of the stack above the program parameters, they are accessed via \texttt{bget}. This solution uses \texttt{Senv.max genv} to shift the function indices by the number of program parameters. An alternative approach is to use \texttt{Senv.push n} times when creating \texttt{fenv} (where \texttt{n} is the number of program parameters).

- The only effect of the translated code for a function application should be to push the value of the application on the stack. To achieve this, it is necessary to remove the operands to the application, which are on the stack below the result after the commands for the translated body are executed. This removal can be performed by executing \texttt{swap pop n} times, but it is more efficient to use \texttt{n put} followed by \texttt{n – 1 pops}. This operand removal sequence can be performed either at the end of the translated function body, or at the end of each translated function application. Assuming that each declared function is called at least once, it is preferable to performing this sequence at the end of the translated program body since this minimizes program size.
let rec transPgm (F.Pgm(fmls,body,fcns)) =  
  let n = List.length fmls  
  and genv = Senv.make (List.rev fmls)  
  and fenv = Senv.make (List.map F.fcnName fcns) in  
  P.Pgm(n, (List.map (fun f -> transFcn f genv fenv) fcns)  
    @ (transExp body Senv.empty genv fenv))

and transFcn (F.Fcn(_,fmls,body)) genv fenv =  
  P.Seq ((transExp body (Senv.make fmls) genv fenv)  
    (* would need to reverse fmls if processed non-reversed rands*)  
    @ (popArgs (List.length fmls)))  
  (* Need to pop args before returning;  
    if we were cleverer, we would pop *before* a tail call to  
    make the implementation properly tail recursive. *)

and popArgs n = (* Preserve the top element, popping the n elements below it *)  
  (* Could use n copies of swap pop, but more efficient to use  
    a single put followed by (n-1) pops. Would even be more efficient  
    if had a way to move top-of-stack pointer by a specified amount *)  
  [P.Int n; P.Put] @ (map (fun _ -> P.Pop) (range 1 (n - 1)))

and transExp exp lenv genv fenv =  
  match exp with  
  | F.Lit (F.Int i) -> [P.Int i]  
  | F.Lit (F.Bool false) -> [P.Int 0]  
  | F.Lit (F.Bool true) -> [P.Int 1]  
  | F.Var v -> (match Senv.lookup v lenv with  
    Some lindex -> [P.Int lindex; P.Get]  
    | None -> (match Senv.lookup v genv with  
      Some gindex -> [P.Int gindex; P.Bget]  
      | None -> raise (TransError ("Unbound variable: " ^ v))))  
  | F.PrimApp (op, rands) ->  
    (transExps rands lenv genv fenv) @ (transPrimop (F.primopName op))  
  | F.If(test,thn,els) -> (transExp test lenv genv fenv)  
    @ [P.Seq (transExp thn lenv genv fenv)]  
    @ [P.Seq (transExp els lenv genv fenv)]  
    @ [P.Sel; P.Exec]  
  | F.Bind(name,defn,body) -> (transExp defn lenv genv fenv)  
    @ (transExp body (Senv.bind name lenv) genv fenv)  
    @ [P.Swap; P.Pop] (* remove bound variable *)  
  | F.App(f, rands) -> (* f is a function name *)  
    (transExps (List.rev rands) lenv genv fenv)  
    (* reverse rands so their indices are front to back *)  
    @ (match Senv.lookup f fenv with  
      Some findex -> [P.Int (findex + (Senv.max genv)); P.Bget; Exec]  
      | None -> raise (TransError ("Unbound function: " ^ f)))  
    | _ -> raise (TransError ("Unhandled FOFL exp: " ^ (F.expToString exp)))

and transExps exps lenv genv fenv =  
  match exps with  
  | [] -> []  
  | e::es -> (transExp e lenv genv fenv) @ (transExps es (Senv.push lenv) genv fenv)

and transPrimop op = (* same as for part b *)

Figure 10: The FOFL-- to PostFix translator from part (e).