Problem Set 6
Due: 11:59pm Monday April 25

Revisions: Apr 20: The specs for push and bind in Fig. 6 on p. 18 were swapped. Apr 23: An extension was given through Mon. Apr. 25. Apr 24: The missing swap pop sequence for exiting bind was added to bind examples in Problem 3 part (d). May 4: In Ind. Prob. part 1, div should be /.

Overview:
The individual problem on this assignment tests your understanding of interpreter extensions and desugaring. The group problems on this assignment cover static and dynamic scope, recursive scoping, and translation.

Reading:
- Handout #31: VALEX: Primitive Operators and Desugaring
- Handout #34: HOFL: First-class Functions and Scoping
- Handout #35: FOFL and FOBS: Restricted Functions

Individual Problem Submission:
Each student should turn in a hardcopy submission packet for the individual problem by slipping it under Lyn’s office door by 6pm Wed. April 13. The packet should include:
1. an individual problem header sheet;
2. your “wiring diagram” from Problem 1a.
3. your pencil-and-paper answers to Problems 1c and 1d.i.
4. your final versions of Loopex.ml, LoopexEnvInterp.ml, and LoopexSubstInterp.ml.
Each student should also submit a softcopy (consisting of your final ps6-individual directory) to the drop directory by executing:

```
cd /students/username/cs251
cp -R ps6-individual ~cs251/drop/ps6/username/
```

Working Together:
If you want to work with a partner on this assignment, you should try to find a different partner than you worked with on a previous assignment. If this proves difficult, please email Lyn describing your situation.

Group Problem Submission:
Each team should turn in a single hardcopy submission packet for all problems by slipping it under Lyn’s office door by 6pm on Fri. Apr. 15. The packet should include:
1. a team header sheet indicating the time that you (and your partner, if you are working with one) spent on the parts of the assignment.
2. your pencil and paper solutions to Group Problems 1 and 2.
3. your final version of FoflToPostFix.ml from Problem 3 (you do not need to submit the intermediate programs for the individual parts).
Each team should also submit a single softcopy (consisting of your final ps6-group directory) to the drop directory ~cs251/drop/p6/username, where username is the username of one of the team members (indicate which drop folder you used on your hardcopy header sheet). To do this, execute:

```
cd /students/username/cs251
cp -R ps6-group ~cs251/drop/ps6/username/
```
Individual Problem  [45]: Going Loopy

This is an individual problem. Each student must solve this problem on her own without consulting any other person (except Lyn).

In this problem you will extend VALEX with a looping construct and explore desugarings involving this construct.

The loop construct

Due to your extensive experience with VALEX in CS251, you have been elected head of the VALEX Users Group, a worldwide consortium of VALEX programmers. At your most recent consortium meeting, there was much grumbling from attendees about the lack of expressiveness of VALEX. As one dissatisfied VALEX programmer put it, “Sure, simprec helps a little bit. But how can we be expected to write general programs in this language if it doesn’t even have a real looping construct?”

You decide it’s high time to pay a visit to Ida Ray-Sun, the CTO of Loopster, a company that specializes in loop constructs for programming languages. Ida agrees to help develop a looping construct for VALEX if you will help with the implementation.

Ida quickly designs a looping construct for VALEX and christens the extended language LOOPEX. Here is Ida’s looping construct:

\[
\text{loop} \left( (sv_1 E_{init_1}, E_{update_1}), \ldots, (sv_n E_{init_n}, E_{update_n}) \right) E_{test} E_{body}
\]

The loop construct describes an iteration over the state variables \(sv_1 \ldots sv_n\), which are assumed to be pairwise distinct. The iteration consists of a sequence of steps between abstract units of time starting with 0, where the state of the iteration at time \(t\) is characterized by the values of the state variables at time \(t\). The state variables are initialized at time \(t = 0\) to the corresponding values of the initializers \(E_{init_1}, \ldots, E_{init_n}\). On each step of the iteration, the updaters \(E_{update_1}, \ldots, E_{update_n}\) are evaluated relative to the state at time \(t\) to determine the state at time \(t + 1\). The iteration continues as long as the test expression \(E_{test}\) gives any non-false value when evaluated relative to the current state. If \(E_{test}\) yields false for the initial state, the updaters are never evaluated. The loop construct returns the value of \(E_{body}\) relative to the first state for which \(E_{test}\) yields false.

The scope of state variables declared in \text{loop} includes the updater expressions, the test expression, and the body expression. The scope does not include the initializer expressions.

For example, the following LOOPEX program calculates the factorial of \(n\):

\[
\text{(loopex } n)\text{ (loop } ((i n (- i 1)) \text{ (prod 1 (* i prod)))) (> i 0)\text{ prod})
\]

Below is an iteration table that shows the values of the state variables of the loop iteration at each point in time when the factorial of 5 is computed. Note that the values in a given row are the “state” of the iteration at that time.
As another example, here are an LOOPEX program that calculates the \( n \)th Fibonacci number, and an iteration table that summarizes the iteration for \( n = 6 \).

\[
\begin{align*}
\text{(loopex (n)} \\
\text{ (loop ((i 0 (+ 1 i))} \\
\text{ (fib_i 0 fib_i+1) } \\
\text{ (fib_i+1 1 (+ fib_i fib_i+1)))} \\
\text{ (< i n)} \\
\text{ fib_i))}
\end{align*}
\]

Note that when evaluating the updater expressions \( \text{fib}_i+1 \) and \( (+ \text{fib}_i \text{fib}_i+1) \) to determine the state for time \( t + 1 \), both of these expressions are evaluated with respect to the values of the state variables \( \text{fib}_i \) and \( \text{fib}_i+1 \) at time \( t \). Because the updaters are effectively evaluated “in parallel”, there is no need for “temporary variables” that would often be necessary if such iteration were expressed via a while or for loop in a language like Java or C.

Your Task
Your task is to solve the following problems related to the loop construct. Parts (a), (c), and (d.i) are pencil-and-paper problems; parts (b) and (d.ii) require fleshing out parts of the LOOPEX interpreter in "/cs251/ps6-individual". To use any parts of the LOOPEX interpreter, you must first evaluate the following in an OCAML interpreter:

```
#cd "/students/your-username/ps6-individual"
#use "load-loopex.ml"
```

Parts (a) – (d.i) are independent and can be done in any order. The code for part (d.ii) can be written independently of the other parts, but testing it requires the completion of one of parts (b.iii) or (b.iv).

a. [5]: Variable Scoping

Fig. 1 shows a (contrived) LOOPEX expression. In this expression, (1) circle every free variable reference occurrence and (2) draw a line from every bound variable reference occurrence to the binding occurrence of that reference.
Figure 1: Sample LOOPEX expression for part a.
**b. [20]: Implementing loop in OCAML**

To implement LOOPEX, Ida begins by making a copy of the VALEX interpreter described in Handout #31. The abstract syntax of LOOPEX is the same as that as VALEX except that the exp data type has been extended with the following clause to handle the loop construct:

```ocaml
| Loop of var list (* state variable names *) |
| * exp list (* initializer expressions *) |
| * exp list (* updater expressions *) |
| * exp (* test expression *) |
| * exp (* body expression *) |
```

Note that the state variables, initializers, and updaters are stored in “unzipped form” rather than being zipped together in some sort of binding structure.

Ida modifies `sexpToExp` and `expToSexp` to correctly handle the parsing and unparsing of loop expressions. However, she asks you to modify the rest of the interpreter to handle the loop construct.

i [2]: `freeVarsExp` In Loopex.ml, add a `loop` clause to the `freeVarsExp` function that calculates the free variables of a loop expression. Test `freeVarsExp` via the `testFreeVarsExp` function, which takes a string representation of an expression and returns a list of the free variables in the expression. E.g.:

```ocaml
# testFreeVarsExp "(+ b (* a b))";;
- : Loopex.S.elt list = ["a"; "b"]
```

ii [4]: `subst` In Loopex.ml, add a `loop` clause to the `subst` function that performs substitutions on a `loop` expression. You can test `subst` via the `testSubst` function, which takes (1) a string representation of an expression and (2) a string representation of an s-expression list of name/expression bindings; it prints the result of performing substitution on the expression using an environment made from the bindings. E.g.:

```ocaml
# testSubst "(bind b (/ a b) (+ a b))" "((a (* a b)) (b (- a b)))";;
(bind b.0 (/ (* a b) (- a b)) (+ (* a b) b.0))
- : unit = ()
```

iii [8]: Environment model `eval` In LoopexEnvInterp.ml, add a `loop` clause to the `eval` function that correctly specifies the evaluation of the `loop` construct in the environment model. You should do this by fleshing out the three OCAML expressions $E_1$, $E_2$, and $E_3$ in the following skeleton:

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

This skeleton uses the following higher-order `iterate` function from the ListUtils module (discussed in Section 4.2 of Handout #20):

```ocaml
let rec iterate next isDone state =
  if isDone state then
    state
  else
    iterate next isDone (next state)
```

**Notes:**
- Think carefully about types when doing this problem. What type of value should be
returned by the call to **iterate** in the skeleton? What does this imply about the types of values returned by \( E_1, E_2, \) and \( E_3 \)?

- Keep in mind that **loop** treats any non-true test value as false. So any non-boolean value is treated like \#f in a **loop**.

- Use the environment operations in `/cs251/utils/Env.ml` to manipulate environments.

- You can test your **loop** clause for **eval** by evaluating `testEnvInterp()`. This runs the factorial and Fibonacci examples described earlier. You are encouraged to add more test programs containing **loop** to the list of entries `loopexEntries` in the file `LoopexInterpTest.ml`.

iv [6]: **Substitution model** **eval** In `LoopexSubstInterp.ml`, add a **loop** clause to the **eval** function that correctly specifies the evaluation of the **loop** construct in the substitution model. You should do this by fleshing out the three OCAML expressions \( E_1, E_2, \) and \( E_3 \) in the following skeleton:

\[
\text{Loop}(\text{vars}, \text{inits}, \text{updates}, \text{test}, \text{body}) ->
\begin{align*}
\text{let state} &= E_1 \text{ in} \\
\text{match eval (substAll state vars test)} \text{ with} \\
\text{Bool false} &= \text{eval } E_2 \\
\_ &= \text{eval } E_3
\end{align*}
\]

**Notes:**

- Think carefully about types when doing this problem. From the way that `state` is used in the skeleton, what type must \( E_1 \) be? Similarly use the contexts of \( E_2 \) and \( E_3 \) to determine what types they must have.

- You can test your **loop** clause for **eval** by evaluating `testSubstInterp()`. This will test the substitution model **eval** function on the entries `loopexEntries` in the file `LoopexInterpTest.ml`.

c. [10]: Desugaring **least** into **loop**

Ida notes that many iteration constructs can be desugared into an appropriate **loop** expression. As an example, she invents a **least** construct defined by the following desugaring rule:

\[
(\text{least } I \text{ var } E \text{ pred}) \rightarrow (\text{loop } ((I \text{ var } 0 (+ I \text{ var } 1))) \ (\text{not } E \text{ pred}) \ I \text{ var})
\]

i [2] Based on the above desugaring, give an English description for the meaning of `(least I var E pred)`. Your description should be very concise.

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

- \( (\text{least } x (> (* x x) 100)) \)
- \( (\text{least } i (>= (* i (\text{least } j (<= (/ 100 (+ j 1)) \ 1))) \ 80)) \)

Recall that `/` denotes **integer division**; it gives the integer quotient of dividing two numbers. For example `(100 50)` yields 2 but `(100 51)` yields 1.

iii [2] Briefly explain the key advantage of implementing **least** as syntactic sugar rather than as a kernel **VALEX** construct (like **if**, **bind**, or **loop**).
d. [10]: Desugaring `simprec` into `loop`

Inspired by Ida’s `least` construct, you decide to extend LOOPEX with the `simprec` construct from Problem Set 5. Rather than implementing `simprec` “from scratch”, as you did in Problem Set 5, you instead implement it as syntactic sugar by rewriting all `simprec` expressions into expressions using `loop`.

You should do this problem in two parts:

i [6] Write a high-level desugaring rule or rules (like those in Handout #31), that specifies how to rewrite the expression `(simprec `E_zero` `I_ans` `E_combine` `E_arg`) into an expression that uses `loop` in addition to any other kernel LOOPEX constructs that you need. The desugared expression should evaluate each of `E_zero` and `E_arg` exactly once. You will need to introduce one or more new names as part of your desugaring. You should specify which of your new names needs to be “fresh” in order to avoid accidental variable capture.

ii [4] Extend the `desugarRules` function in the file `Loopex.ml` so that it correctly desugars `simprec` into `loop` by implementing your high-level desugaring rule(s). Use `StringUtils.fresh` to introduce fresh variable names. You can test your desugaring by adding examples containing `simprec` to the list of entries `loopexEntries` in the file `LoopexInterpTest.ml` and executing `testEnvInterp()` or `testSubstInterp()`.

Group Problems

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

a. [10] Suppose that the program in Figure 2 is run on the input argument list [5]. Draw an environment diagram that shows all of the environments and closures that are created during the evaluation of this program in *statically scoped* HOFL. In order to simplify this diagram:

- you should treat `bind` as if it were a kernel construct and ignore the fact that it desugars into an application of an `abs`. That is, you should treat the evaluation of `(bind `I` `E_defn` `E_body`) in environment `F` as the result of evaluating `E_body` in the environment frame `F'`, where `F'` binds `I_defn` to `V_defn`, `V_defn` is the result of evaluating `E_defn` in `F`, and `F'` is the parent pointer of `F`.

- you should treat `fun` as if it were a kernel construct and ignore the fact that it desugars into nested abstractions. In particular, (1) the evaluation of `(fun (`I_1` ... `I_n`) `E_body`) should be a closure consisting of (a) the `fun` expression and (b) the environment of its creation and (2) the application of the closure `<(fun (`I_1` ... `I_n`) `E_body`), `F_creation`>` to argument values `V_1` ... `V_n` should create a new environment frame `F` whose parent is `F_creation` and which binds the variables `I_1` ... `I_n` to the values `V_1` ... `V_n`.

b. [1] What is the final value of the program from part (a) in statically scoped HOFL? You should figure out the answer on your own, but may wish to check it using the statically scoped HOFL interpreter.

c. [6] Draw an environment diagram that shows all of the environments created in *dynamically scoped* HOFL when running the program from Figure 2 on the input argument list [5].

d. [1] What is the final value of the program from part (c) in dynamically scoped HOFL?
(hofl (a)
  (bind linear (fun (a b)
        (fun (x)
            (+ (* a x) b))
      (bind line1 (linear 1 2)
        (bind line2 (linear 3 4)
          (bind try (fun (b)
                        (prep (line1 b)
                           (prep (line2 (+ b 1))
                             (prep (line2 (+ b 2))
                               #e)))))
          (try (+ a a)))))))

Figure 2: A sample HOFL program used to illustrate the difference between static and dynamic scope.

e. [2] In a programming language with higher-order functions, which supports modularity better: lexical scope or dynamic scope? Explain your answer.

f. [5] Suppose that you are given a HOFL interpreter, but you are not told whether it is a statically-scoped or dynamically-scoped version of the interpreter. Write a simple HOFL expression that will evaluate to #t for a statically-scoped interpreter but will evaluate to #f for a dynamically-scoped interpreter. The only types of values your expression should manipulate are booleans and functions; it should not use integers, pairs, or lists.

Group Problem 2 [15]: bindrec
Consider the following HOFL expression E:

(bind f (abs x (+ x 1))
  (bindrec ((f (abs n
         (if (= n 0)
           1
           (* n (f (- n 1))))))
     (f 3)))

a. [5] Draw an environment diagram showing the environments created when E is evaluated in statically scoped HOFL, and show the final value of evaluating E.

b. [4] Consider the expression $E'$ that is obtained from E by replacing bindrec by bindseq. Draw an environment diagram showing the environments created when $E'$ is evaluated in statically scoped HOFL, and show the final value of evaluating $E'$.

c. [4] Draw an environment diagram showing the environments created when $E'$ is evaluated in dynamically scoped HOFL, and show the final value of evaluating $E'$.

d. [2] Does a dynamically scoped language need a recursive binding construct like bindrec in order to support the creation of local recursive procedures? Briefly explain your answer.
Group Problem 3 [60]: Translating FOFL to POSTFIX

On PS4 you manually translated an OCAML gcd program into POSTFIX. In this problem, you will develop a translator that automatically translates programs with recursive functions to POSTFIX.

The Source Language: FOFL--

The source language of the translator is a dialect of the FOFL language. FOFL (First-Order Functional Language) is itself a restricted subset of HOFL in which all functions are second-class and must be declared at top-level in a single recursive scope. FOFL programs have the s-expression form

\[(\text{fofl } (I_{fml_1} \ldots I_{fml_k}) \ E_{\text{body}} \ FD_1 \ldots FD_n)\]

where each \(FD_i\) is a function declaration of the form

\[(\text{def } (I_{fcnName} I_{fml_1} \ldots I_{fml_m}) \ E_{\text{body}}).\]

\(I_{fcnName}\) is the name of the function, \(I_{fml_1} \ldots I_{fml_m}\) are the formal parameter names of the function, and \(E_{\text{body}}\) is the body of the function. Here is a sample FOFL program that determines whether its non-negative input is even.

\[(\text{fofl } (n) \ (\text{even? } n))\]

\[
(\text{def } (\text{even? } x) \ (\text{if } (= x 0) \ #t \ (\text{odd? } (- x 1))))\\
(\text{def } (\text{odd? } y) \ (\text{if } (= y 0) \ #f \ (\text{even? } (- y 1))))\]

Fig. 3 presents OCAML data types that express the abstract syntax of FOFL. The full FOFL language supports many value types (characters, symbols, lists) and operations (e.g. string concatenation) that are not supported by POSTFIX language that is the target language of the translation. For this reason, the source language will be a restricted subset of FOFL called FOFL-- that includes only those features that can be easily translated to POSTFIX. A valid FOFL-- program is a FOFL program satisfying the following restrictions:

- The only types of values manipulated by the program are integers and booleans. In particular, only the following primitives may be used: the arithmetic operators +, -, *, /, and \%; the relational operators <, <=, =, !=, >=, and >; and the logical operators \text{not}, \text{and}, \text{or}, and \text{bool=}.
- The program is type safe. That is, when run on any integer inputs, the program does not encounter a dynamic type error. Since values are restricted to be integers and booleans, this means that the program never attempts to perform an integer operation on a boolean (as in (+ 1 #f)) and never attempts to perform a boolean operation on an integer (as in (not 3)). Note that a FOFL-- may encounter other dynamic errors, in particular division-by-zero and remainder-by-zero errors.
Figure 3: OCAML data types for the abstract syntax of FOFL.
The Target Language: Postfix

The target language of the translator is the Postfix language described in Handout #27 that has been extended with two additional commands, bget and bput\(^1\) (Fig. 4). Whereas get and put load and store values using indices relative to the top of the stack, bget and bput use indices relative to the bottom of the stack. For example, for a stack whose elements (from top down) are 6, 5, 9, 8:

1 get denotes 6 1 bget denotes 8
2 get denotes 5 1 bget denotes 9
3 get denotes 9 3 bget denotes 5
4 get denotes 8 4 bget denotes 6

Fig. 5 presents OCAML data types that express the abstract syntax of Postfix.

- **bget**: Call the top stack value \(v_{\text{index}}\) and the remaining stack values (from top down) \(v_1, v_2, \ldots, v_n\). Pop \(v_{\text{index}}\) off the stack. If \(v_{\text{index}}\) is a numeral \(i\) such that \(1 \leq i \leq n\), push \(v_{(n+1-i)}\) onto the stack. Signal an error if the stack does not contain at least one value, if \(v_{\text{index}}\) is not a numeral, or if \(i\) is not in the range \([1, n]\).

- **bput**: Call the top stack value \(v_{\text{index}}\), the next-to-top stack value \(v_{\text{val}}\), the remaining stack values (from top down) \(v_1, v_2, \ldots, v_n\). Pop \(v_{\text{index}}\) and \(v_{\text{val}}\) off the stack. If \(v_{\text{index}}\) is a numeral \(i\) such that \(1 \leq i \leq n\), replace the slot holding \(v_{(n+1-i)}\) on the stack by \(v_{\text{val}}\). Signal an error if the stack does not contain at least two values, if \(v_{\text{index}}\) is not a numeral, or if \(i\) is not in the range \([1, n]\).

**Figure 4: Semantics of Postfix commands.**

```ocaml
type pgm = (* PostFix programs *)
  Pgm of int * com list
and com = (* PostFix commands *)
  Int of int (* push integer literal *)
  | Str of string (* push string literal *)
  | Seq of com list (* executable sequence *)
  | Pop (* pop top value from stack *)
  | Swap (* swap top two values of stack *)
  | Sel (* choose one of two values from stack *)
  | Get (* push value at given stack index *)
  | Put (* store top of stack at given stack index *)
  | Prs (* print string *)
  | Pri (* print integer *)
  | Exec (* execute sequence at top of stack *)
  | Add | Sub | Mul | Div | Rem (* arithmetic ops *)
  | LT | LE | EQ | NE | GE | GT (* relational ops *)
  | Bget (* push value at given bottom-relative stack index *)
  | Bput (* store top value at given bottom-relative stack index *)
```

**Figure 5: OCAML data types for the abstract syntax of Postfix.**

\(^1\)Only bget is actually required by the translator; bput is provided for symmetry.
Your Task: Translating FOFL-- to POSTFIX

Your task is to implement the following translation function:

val transPgm: Fofl.pgm -> PostFix.pgm

Suppose that $P_F$ is a FOFL-- program and transPgm $P_F$ yields the POSTFIX program $P_P$. Then $P_P$ should have the same meaning as $P_F$, in the following sense:

- if $P_F$ returns an integer $i$ when run on argument list args, then running $P_P$ on args should also return $i$;
- if $P_F$ returns a boolean $b$ when run on argument list args, then running $P_P$ on args should return 1 if $b$ is true and 0 if $b$ is false.
- if $P_F$ signals a divide-by-zero or remainder-by-zero error when run on argument list args, then running $P_P$ on args should signal the same error.

If $P_F$ is not a valid FOFL-- program, the behavior of transPgm is unspecified.

To complete this problem, you will need to flesh out the transPgm skeleton and several auxiliary functions in the file FoflToPostFix.ml in the ps6-group directory. You will proceed by first defining a translator for a very simple subset of FOFL-- and then incrementally add features to the subset until your translator can handle all of FOFL--. You will use the working POSTFIX implementation in `/cs251/postfix` and the working FOFL implementation in `/cs251/fofl`.

a. [13]: Integers and Arithmetic Operations

Assume that the given FOFL-- program has zero parameters and no function declarations — i.e., it has the form (fofl () $E_{body}$). Further assume that $E_{body}$ is formed out of only (1) integer literals and (2) applications of the primitive arithmetic operators +, -, *, /, and %. Implement transPgm for this restricted set of programs. As part of your solution, you should define and use the following auxiliary translation function:

val transExp: Fofl.exp -> PostFix.com list

Suppose that:

- $E$ is a FOFL-- expression satisfying the restrictions described above;
- transExp $E$ yields the POSTFIX command list coms;
- $s$ is any stack (i.e. list) of integers;

Then:

- If evaluating $E$ returns the integer $i$, then executing coms on stack $s$ should yield the stack $i :: s$.
- If evaluating $E$ on args signals an error, then executing coms on $s$ should signal a similar error.

transExp is effectively an infix-to-postfix converter for arithmetic expressions. For example, the FOFL-- program

(fofl () (* (% (+ 3 4) 5) (/ (- 8 1) 2)))

should be translated to the POSTFIX program

(postfix 0 3 4 add 5 rem 8 1 sub 2 div mul)
Notes:

- As usual, start this problem set by performing a `cvs update -d`, and perform an update every time you log in to work on this problem.
- To load the translator, connect to the `ps6-group` directory in OCAML via `#cd "/students/your-account-name/ps6-group"` and then execute `#use "load-fofl-to-postfix.ml"`. As usual, carefully examine the output of the `#use` to see errors that may have scrolled by.
- You will need to manipulate both FOFL and POSTFIX program elements in the same module. To facilitate this, the following abbreviations are defined in `FoflToPostFix`:
  ```ocaml
  module F = Fofl
  module P = PostFix
  ```
  For example, `F.Int` is the constructor for a FOFL integer value and `P.Int` is the constructor for a POSTFIX integer command.
- If you encounter any error during the translation process (e.g., a FOFL expression that is not in the restricted subset being translated), you should raise a `TransError` exception. This exception takes a string argument that describes the nature of the error.
- You can test your translations using the following functions that have been provided for you:
  ```ocaml
  val trans : string -> unit
  Given a string representing a FOFL-- program `PF`, print out the POSTFIX program that is the result of `transPgm PF`. For example:
  ```ml
  # trans "(fofl () (* (% (+ 3 4) 5) (/ (- 8 1) 2)))";;
  (postfix 0 3 4 add 5 rem 8 1 sub 2 div mul)
  - : unit = ()
  ```
  ```ocaml
  val runTrans : string -> int list -> PostFixInterp.ans
  Suppose that `PF` is a FOFL-- program and `PF` is the POSTFIX program that is the result of `transPgm PF`. Given a string representation of `PF` and a list of integer arguments `args`, `runTrans` returns the result of running `PF` on `args`. For example:
  ```ml
  # runTrans "(fofl () (* (% (+ 3 4) 5) (/ (- 8 1) 2)))" [];;
  - : PostFixInterp.ans = IntAns 6
  ```
  ```ocaml
  val testTrans : string -> int list -> unit
  Suppose that `PF` is a FOFL-- program and `PF` is the POSTFIX program that is the result returned by `transPgm PF`. Given a string representation of `PF` and a list of integer arguments `args`, `testTrans` computes the result of running `PF` on `args` and of running `PF` on `args`. If the results match, the result is printed along with OK!. If the results do not match, a description of the mismatch is printed.
  ```
  val test : unit -> unit
  Performs `testTrans` on each of the entries in `transEntries` within the file `FoflToPostFixTest.ml`. This list contains the examples in this problem description. You are encouraged to add more entries to this list. There are also testing functions `testa`, `testb`, `testc`, `testd`, and `teste` for the individual parts.

b. [12]: Booleans and Related Operations
In this part, extend your translator from part (a) to handle booleans, relational operations (`<`, `<=`, `=`, `!=`, `>`, and `>`) , logical operations (`not`, `and`, `or`, and `bool=`) , and conditionals (`if`). Recall
that the boolean true value should translate to the integer 1 and the boolean false value should translate to the integer 0. Continue to assume that all programs have the form \((\text{fofl } E_{\text{body}})\).

For example:

```plaintext
# runTrans "(fofl () (or (and (< 1 2) (= 3 4)) (not (and (> 6 5) (!= 7 8)))) )" [] ;
- : PostFixInterp.ans = IntAns 0
# runTrans "(fofl () (or (and (< 1 2) (= 3 4)) (not (and (> 6 5) (= 7 7)))) )" [] ;
- : PostFixInterp.ans = IntAns 1
# runTrans "(fofl () (|| (&& (< 1 2) (= 3 4)) (not (&& (> 6 5) (!= 7 8)))) )" [] ;
- : PostFixInterp.ans = IntAns 0
# runTrans "(fofl () (|| (&& (< 1 2) (= 3 4)) (not (&& (> 6 5) (= 7 7)))) )" [] ;
- : PostFixInterp.ans = IntAns 1
# runTrans "(fofl () (if (> 1 2) (+ 3 4) (* 5 6)))" [] ;
- : PostFixInterp.ans = IntAns 7
# runTrans "(fofl () (if (< 1 2) (+ 3 4) (* 5 6)))" [] ;
- : PostFixInterp.ans = IntAns 30
```

c. [10]: Program Parameters

In this part, extend your translator from part (b) to handle programs with a non-empty list of formal parameter that are referenced in the body of the program. In other words, the translator should handle programs of the form \((\text{fofl } (I_1 \ldots I_n) E_{\text{body}})\) where \(E_{\text{body}}\) is built out of any valid FOFL-- expressions except for bind expressions and function applications. For example:

```plaintext
# runTrans "(fofl (a b) (/ (+ a b) 2))" [5; 15];
- : PostFixInterp.ans = IntAns 10
# runTrans "(fofl (x y) (- (* x x) (* y y)))" [6;10];
- : PostFixInterp.ans = IntAns -64
# runTrans "(fofl (a b c x) (+ c (* x (+ b (* x a)))) )" [1;2;3;4];
- : PostFixInterp.ans = IntAns 27
# runTrans "(fofl (n) (if (< n 0) (- 0 n) n))" [-17];
- : PostFixInterp.ans = IntAns 17
# runTrans "(fofl (n) (if (< n 0) (- 0 n) n))" [42];
- : PostFixInterp.ans = IntAns 42
```

It is necessary for the translator to keep track of the locations of the program parameters on the stack. A data structure that tracks locations associated with variable names in a translator is called a static environment (in contrast with the dynamic environment that associates names with values in an evaluator). You have been provided with an Senv module for manipulating static environments. This module exports an senv type for static environments that associate integer stack offsets with variable names. See Appendix A for details. For example:

```plaintext
# let testSenv se = ListUtils.map (fun id -> Senv.lookup id se) ["a";"b";"c";"d"];
val testSenv : Senv.senv -> int option list = <fun>
# testSenv (Senv.make ["a";"b";"c";]) ;
- : int option list = [Some 1; Some 2; Some 3; None]
# testSenv (Senv.make ["c";"b";"a";]) ;
- : int option list = [Some 3; Some 2; Some 1; None]
```

14
In order to translate variable references, the `transExp` function should be modified to include a static environment argument that tracks the locations of the program parameters:

```ocaml
val transExp: Fofl.exp -> Senv.senv -> PostFix.com list
```

The static environment should be consulted whenever a variable is translated. For this part, the static environment could store either top-based stack indices (used in conjunction with the Postfix `get` command) or bottom-based stack indices (used in conjunction with the Postfix `bget` command). It turns out that bottom-based indices for the program parameters are easier because they do not have to be adjusted every time a new value is pushed on the stack. Moreover, bottom-based indices for program parameters will work better with the remaining parts of the translator. So you should use bottom-based indices for the program parameters. For example:

```ocaml
# trans "(fofl (a b c) (* (+ a b) (- a c)))";;
(postfix 3 3 bget 2 bget add 3 bget 1 bget sub mul)
- : unit = ()
```

d. [10]: Local Bindings

In this part, extend the translator to handle `bind` expressions within the program body. The tricky aspect of translating `bind` is that the value defined by a `bind` is most naturally pushed on the top of the stack. In general, this means that `bind`-bound variables should be translated using top-based indices rather than bottom-based indices. And top-based indices must change whenever any new value is pushed onto the stack. Carefully study the indices in the following examples:

```ocaml
# trans "(fofl (a b c) (bind d (+ a b) (bind e (/ d c) (* (+ e d) (- e d))))))";;
(postfix 3 3 bget 2 bget add 1 get 1 bget div 1 get 3 get add 2 get 4 get sub mul swap pop swap pop)
- : unit = ()
```

```ocaml
# trans "(fofl (a b c) (bind d (+ a b) (* (bind e (/ c d) (- e d))) (bind f (+ c d) (% f d))))";;
(postfix 3 3 bget 2 bget add 1 bget 2 get div 1 get 3 get sub swap pop 1 bget 3 get add 1 get 4 get rem swap pop mul swap pop)
- : unit = ()
```

Note that the `swap pop` commands are used to remove the `bind` definition value from the stack when returning the value of the `bind` body from the `bind` expression.

In order to handle `bind`, the `transExp` function should use two static environments: a local static environment that tracks `bind`-bound variables using top-based indices, and a global static environment that tracks program parameters using bottom-based indices:

```ocaml
val transExp: Fofl.exp (* FOFL expression being translated *)
  -> Senv.senv (* local static environment *)
  -> Senv.senv (* global static environment *)
  -> PostFix.com list
```

The `Senv.bind` and `Senv.push` functions can be used to adjust top-based indices in the local environment; see Appendix A for details.

e. [15]: Function Declarations

To complete the translator, it is necessary to handle FOFL-- programs with function declarations and function applications. For example:
# runTrans 
"(fofl (x) (factrec x))
(def (factrec n)
   (if (= n 0) 1 (* n (factrec (- n 1))))))" [5];;
- : PostFixInterp.ans = IntAns 120

# runTrans 
"(fofl (x) (factloop x))
(def (factloop n)
   (loop n 1))
(def (loop num ans)
   (if (= num 0) ans (loop (- num 1) (* ans num))))" [5];;
- : PostFixInterp.ans = IntAns 120

# runTrans 
"(fofl (x) (fibrec x))
(def (fibrec n)
   (if (<= n 1) n (+ (fibrec (- n 1)) (fibrec (- n 2))))))" [10];;
- : PostFixInterp.ans = IntAns 55

# runTrans 
"(fofl (x) (fibloop x))
(def (fibloop n)
   (loop n 0 1))
(def (loop num a b)
   (if (= num 0) a (loop (- num 1) b (+ a b))))" [10];;
- : PostFixInterp.ans = IntAns 55

# runTrans 
"(fofl (x y) (gcd x y))
(def (gcd a b)
   (if (= b 0) a (gcd b (% a b))))" [36;60];;
- : PostFixInterp.ans = IntAns 12

# runTrans 
"(fofl (x y) (gcd x y))
(def (gcd a b)
   (if (= b 0) a (gcd b (% a b))))" [42;60];;
- : PostFixInterp.ans = IntAns 6

# runTrans 
"(fofl (n) (even? n))
(def (even? x)
   (if (= x 0) #t (odd? (- x 1))))
(def (odd? y)
   (if (= y 0) #f (even? (- y 1))))" [100];;
- : PostFixInterp.ans = IntAns 1

# runTrans 
"(fofl (n) (even? n))
(def (even? x)
   (if (= x 0) #t (odd? (- x 1))))
(def (odd? y)
   (if (= y 0) #f (even? (- y 1))))" [101];;
- : PostFixInterp.ans = IntAns 0

Now the transExp function needs a third static environment to track function names:

val transExp: Fofl.exp (* FOFL expression being translated *)
  -> Senv.senv (* local static environment *)
  -> Senv.senv (* global static environment for program parameters *)
  -> Senv.senv (* global static environment for function names *)
  -> PostFix.com list

Each function declaration can be translated to an executable sequence that expects its arguments
to be at the top of the stack when it is executed. You need to think carefully about the convention for storing these executable sequences and the convention for passing arguments to them. There are many possible correct solutions using different conventions; you need to make sure that your code is consistent with your conventions!

Notes:

- You may find the `Senv.max` operation helpful for determining the number of elements bound in a static environment; see Appendix A.
- Don’t forget to pop the arguments of a function application off the stack before returning from a function call.

If you do this in a straightforward way (which is what I expect), tail calls in the FOFL program will end up translating to non-tail calls in Postfix (because the commands that remove the arguments will be pending operations performed after the commands for the call). A consequence of this is that programs you might expect to execute in constant space (like `factloop` and `fibloop` above) will require linear stack space.

However, if you are very clever, you can remove the arguments of a function before invoking the tail call in its body. In this case, FOFL tail calls translate to Postfix programs that do not keep extraneous information on the stack, and `factloop` and `fibloop` will take only constant stack space. I do not expect you to do this, but will award extra credit to you if you succeed in doing this.

Extra Credit 1 [20]: Efficient Implementation of Tail Calls  Implement the space-efficient approach to tail calls suggested above in the notes for part (e) of the FOFL-to-Postfix translator.

Extra Credit 2 [20]: Implementing HOFL in HOFL  In class, we saw how to implement a BINDEX interpreter in HOFL. Implement a complete HOFL interpreter in HOFL. An interpreter for a language $L$ implemented in language $L$ is called a meta-circular interpreter.
Appendix A: The Senv Module

The Senv module provides an implementation of static environments that associate variable names with positive integer locations. In the case of the FOFL to POSTFIX translator, the integer locations can be interpreted as offsets relative to the top or bottom of the stack.

Here is the signature of the Senv module:

```ocaml
datatype senv =
end
```

The senv type is an abstract type that denotes static environments in which every variable name (string) is associated with an int option that is either None or Some i for some i > 0. The operations on this type are specified in Fig. 6. Examples illustrating these operations are presented in Fig. 7.

<table>
<thead>
<tr>
<th>val empty: senv</th>
</tr>
</thead>
<tbody>
<tr>
<td>This denotes the empty static environment — an environment in which all variable names are associated with the None option.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>val make: string list -&gt; senv</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assume that vars is a length-n list of strings that are pairwise distinct. Then make vars returns a static environment in which (1) for each integer i in [1..n], the ith string in vars (1-indexed) is associated with Some i and (2) all other strings are associated with None.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>val bind: string -&gt; senv -&gt; senv</th>
</tr>
</thead>
<tbody>
<tr>
<td>bind var senv returns a new static environment senv’ in which (1) var maps to Some 1; (2) each name not equal to var that maps to Some i in senv maps to Some (i + 1) in senv’ and (1) each name not equal to var that maps to None in senv maps to None in senv’.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>val push: senv -&gt; senv</th>
</tr>
</thead>
<tbody>
<tr>
<td>push senv returns a new static environment senv’ in which (1) each name that maps to Some i in senv maps to Some (i + 1) in senv’ and (1) each name that maps to None in senv maps to None in senv’.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>val lookup: string -&gt; senv -&gt; int option</th>
</tr>
</thead>
<tbody>
<tr>
<td>lookup var senv returns the int option in senv associated with var.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>val max: senv -&gt; int</th>
</tr>
</thead>
<tbody>
<tr>
<td>max senv returns the largest integer i such that Some i is associated with a variable name in senv. If senv is the empty environment, than max returns 0.</td>
</tr>
</tbody>
</table>

Figure 6: The contract for operations in the Senv module.
# let test se = (Senv.lookup "a" se, Senv.lookup "b" se, Senv.lookup "c" se, Senv.max se);
val test : Senv.senv -> int option * int option * int option * int = <fun>

# let se0 = Senv.empty;;
val se0 : Senv.senv = <abstr>

# test se0;;
- : int option * int option * int option * int = (None, None, None, 0)

# let se1 = Senv.bind "a" se0;;
val se1 : Senv.senv = <abstr>

# test se1;;
- : int option * int option * int option * int = (Some 1, None, None, 1)

# let se2 = Senv.push se1;;
val se2 : Senv.senv = <abstr>

# test se2;;
- : int option * int option * int option * int = (Some 2, None, None, 2)

# let se3 = Senv.bind "b" se2;;
val se3 : Senv.senv = <abstr>

# test se3;;
- : int option * int option * int option * int = (Some 3, Some 1, None, 3)

# let se4 = Senv.push se3;;
val se4 : Senv.senv = <abstr>

# test se4;;
- : int option * int option * int option * int = (Some 4, Some 2, None, 4)

# let se5 = Senv.push se4;;
val se5 : Senv.senv = <abstr>

# test se5;;
- : int option * int option * int option * int = (Some 5, Some 3, None, 5)

# let se6 = Senv.bind "c" se5;;
val se6 : Senv.senv = <abstr>

# test se6;;
- : int option * int option * int option * int = (Some 6, Some 4, Some 1, 6)

# let se2' = Senv.bind "b" se1;;
val se2' : Senv.senv = <abstr>

# test se2';;
- : int option * int option * int option * int = (Some 2, Some 1, None, 2)

# let se7 = Senv.make ["a"; "b"; "c"];;
val se7 : Senv.senv = <abstr>

# test se7;;
- : int option * int option * int option * int = (Some 1, Some 2, Some 3, 3)

# let se8 = Senv.make ["c"; "b"; "a"];;
val se8 : Senv.senv = <abstr>

# test se8;;
- : int option * int option * int option * int = (Some 3, Some 2, Some 1, 3)

Figure 7: Examples of the Senv static environment module.
Name:

Date & Time Submitted:

By signing below, I attest that I have followed the policy for individual problems set forth in the Course Information handout. In particular, I have not consulted with any person except Lyn about these problems and I have not consulted any materials from previous semesters of CS251.

Signature:

In the Time column, please estimate the time you spent on the parts of this problem set. Please try to be as accurate as possible; this information will help me design future problem sets. I will fill out the Score column when grading you problem set.

<table>
<thead>
<tr>
<th>Part</th>
<th>Time</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Reading</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem 1a [5]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem 1b [20]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem 1c [10]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem 1d [10]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Names of Team Members:

Date & Time Submitted:

Collaborators (anyone you or your team collaborated with):

By signing below, I/we attest that I/we have followed the collaboration policy as specified in the Course Information handout.
Signature(s):

In the Time column, please estimate the time you or your team spent on the parts of this problem set. Team members should be working closely together, so it will be assumed that the time reported is the time for each team member. Please try to be as accurate as possible; this information will help me design future problem sets. I will fill out the Score column when grading you problem set.

<table>
<thead>
<tr>
<th>Part</th>
<th>Time</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Reading</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem 1 [25]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem 2 [15]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem 3 [60]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>