Problem Set 4  
Due: Monday, March 5, 2001

In this assignment, you will further study naming issues and will extend the IBEX language implementation to support several new features.

Submission:

• Problem 1 is a pencil-and-paper problem that only needs to appear in your hardcopy submission.

• For Problems 3 and 4, your softcopy submission should include a copy of your entire ps4 directory.

• Your hardcopy submission for Problem 2 should be the file primops.scm.

• Your hardcopy submission for Problem 3 should be the file desugar.scm.

• Your hardcopy submission for Problem 4 should be the file type-check.scm.

Problem 0: Studying IBEX

All of the problems on this problem set involve the IBEX language discussed in class or extensions to this language. IBEX is an extension of BINDEX that supports boolean values and conditionals, as well as other primitive datatypes and operations.

Before attempting the problems, you should study the code for the implementation of the IBEX language, which can be found in `/cs251/ps4` after you perform `cvs update -d`.

Although there is nothing to turn in for this problem, the rest of the problems will be significantly easier once you understand how IBEX works.

To use any of the functions defined within files in the ps4 directory, you should evaluate the following in Scheme:

```
(cd "~/cs251/ps4")
(load "load-ibex.scm")
```

Having done this, you can now experiment with any functions in the IBEX interpreter. For example:

```scheme
;; Run the absolute value program on the input -4
;; under the environment model
(env-run '(program (a)
  (if (< a 0)
    (- 0 a)
    a))
  '(-4))
```

4
;; Run the absolute value program on the input -4
;; under the substitution model
(subst-run '(program (a)
    (if (< a 0)
        (- 0 a)
        a))
    '(-4))
4

;; Calculate free variables of an expression.
(free-vars '(if (< a b) (+ a c) (* b d)))
(a b c d)

;; Rename a variable in an expression.
(rename 'a 'b '(bind b (+ a b) (* a b)))
(bind b_1 (+ b b) (* b b_1))

;; Perform a substitution in an expression.
(subst (env-make '(a c)
    (map make-literal '(3 5)))
    '(bind c (+ a b) (* c d)))
(bind c (+ 3 b) (* c d))

Problem 1 [25]: bindpar and bindseq

a [5]: Suppose that the following program is run on the arguments 3 and 5. Indicate the value that each name will be bound to during the execution, and also indicate the resulting value of the program.

```scheme
;; BINDSEQ test program
(program (a b)
  (bindseq ((a (* a b)))
    (b (+ a b)))
  (bindseq ((a (- b a)))
    (b (div b a)))
  (+ a b))
```

b [5]: Redo part a, except using a version of the program in which every bindseq has been replaced by bindpar.

c [5]: Write the result of desugaring the programs from both part a and part b into BINDEX programs that use only bind in place of bindseq and bindpar. (You should perform the desugaring by hand and not use Scheme to do it for you!) Assume a reasonable convention for $\alpha$-renaming bound variables when necessary.

d [10]: Fig. 1 shows the clause for handling bind within the subst function of the BINDEX and IBEX implementations.

1. Explain why the substitution on the body of the bind is performed with respect to new-env (in which any binding for the bound variable of the bind has been removed) rather than to env. Use example(s) to illustrate would would go wrong if the substitution on the body used env instead of new-env.

2. Consider the final if expression within the bind clause of subst in Fig. 1. In a call-by-value substitution model interpreter, it turns out that only the else branch of this if will ever be taken. Explain why this is so.

3. Give a BINDEX expression whose evaluation under a call-by-name substitution model interpreter would cause the then branch of the final if to be taken. Argue that taking the else branch instead would cause the evaluator to give the wrong answer.
Figure 1: Clause for handling bind within the subst function.
**Problem 2 [15]: Extending IBEX with string operations**

*Strings*

The IBEX language implementation is designed to make it fairly easy to add new primitive datatypes and operations on these datatypes. As an example of this, you will be extending IBEX to handle strings.

Conceptually, a string is just a sequence of characters. We will adopt the convention used in most languages (including C, Java, Scheme, ML, and Haskell) that a string literals are denoted by text delimited by double quotes. For example, here are some string literals: "", "a", "cs251", "I do not like them, Sam I am!".

For adding strings to IBEX, it is easiest to assume that IBEX strings are simply represented as strings in the underlying Scheme implementation. (This is the same decision made for IBEX integers; but recall that IBEX booleans and symbols are represented differently than in Scheme.) We add strings to the abstract syntax of IBEX via the following functions:

```scheme
;; Define an IBEX string as a Scheme string
(define ibex-string? string?)

;; Extend LITERAL? to recognize strings.
(define literal?
  (lambda (exp)
    (or (ibex-integer? exp)
        (ibex-boolean? exp)
        (ibex-symbol? exp)
        (ibex-string? exp) ; *** Strings are a new kind of literal )))

;; Extend TYPE-OF with a new STRING type.
(define type-of
  (lambda (val)
    (cond ((ibex-integer? val) 'int)
          ((ibex-boolean? val) 'bool)
          ((ibex-symbol? val) 'sym)
          ((ibex-string? val) 'string) ; *** New type for strings
          (else (throw 'type-of-unknown-value val))
          )))
```

With the above additions, it is possible to use string literals in IBEX programs. For example:

```scheme
(program (n)
  (if (> n 0)
    "positive"
    (if (= n 0)
      "zero"
      "negative")))
```
In addition to including string literals, however, we would also like to extend IBEX with operations that allow analyzing the structure of a string and synthesizing new strings. In particular, consider the following four string operations:

\( \text{strlen} \ str \)
Returns the length (number of characters in) the string \( str \).

\( \text{strlt} \ str1 \ str2 \)
Returns \texttt{true} if \( str1 \) is less than \( str2 \) in the lexicographic (dictionary) ordering of strings, and \texttt{false} otherwise. For example, the following are arranged in lexicographic order: "", "a", "aa", "ab", "b", "ba", "bb".

\( \text{str+} \ str1 \ str2 \)
Returns the string that has all the characters of \( str1 \) followed by all of those of \( str2 \). For example, \( \text{str+ "ab" "bcd"} \) yields "abccd".

\( \text{substr} \ lo \ hi \ str \)
Assume that the characters of a length-\( n \) string are indexed from 1 (the first character) to \( n \) (the last character). Returns the string consisting of all the characters between indices \( lo \) and \( hi \), inclusive. If \( lo \) is greater than \( hi \), then the empty string is returned. If either index is out of the range \([1..n]\), then throws an exception whose tag is \texttt{substr: index-out-of-bounds} and whose value is the offending index. For example:

\[
\begin{align*}
\text{substr 1 1 "abcdef"} &; \text{Returns "a"} \\
\text{substr 2 5 "abcdef"} &; \text{Returns "bcde"} \\
\text{substr 3 2 "abcdef"} &; \text{Returns ""} \\
\text{substr 0 5 "abcdef"} &; \text{Throws exception substr: index-out-of-bounds 0} \\
\text{substr 2 7 "abcdef"} &; \text{Throws exception substr: index-out-of-bounds 7}
\end{align*}
\]

\textit{Your Task}
Your task is to extend IBEX with the above four string operations by adding appropriate bindings to the \texttt{primop-env} environment in the file \texttt{primops.scm}. Each binding associates a name with a \texttt{primitive operator descriptor} created by invoking \texttt{make-pdesc} on four arguments:

1. the name of the operator (a symbol);
2. the types of the operands (a list of symbols);
3. the return type (a symbol);
4. a Scheme function that takes the specified number and types of arguments and returns the specified type of result.

You will need to use Scheme string operations in your implementation. For documentation on these, consult the section on strings in the \textit{Revised$^5$ Report on the Algorithmic Language Scheme (R5RS)}.

You can test your implementation by invoking \texttt{(test-strings)}.
Problem 3 [25]: Desugaring classify

The classify construct

You are a summer programming intern at Sweetshop Coding, Inc. Your supervisor, Dexter Rose, has been studying the syntactic sugar for Scheme and is very impressed by the cond and case constructs. He decides that it would be neat to extend IBEX with a related classify construct that classifies an integer relative to a collection of ranges. For instance, using his construct, Dexter can write the following grade classification program:

(program (grade)
  (classify grade
   ((90 100) (symbol A))
   ((80 89) (symbol B))
   ((70 79) (symbol C))
   ((60 69) (symbol D))
   (otherwise (symbol F)))
)

This program takes an integer grade value and returns a symbol indicating which range the grade falls in.

In general, the classify construct has the following form:

(classify \(E_{\text{disc}}\)
  \(((E_{lo1} \ E_{hi1}) \ E_{\text{body1}})\)
  \[\vdots\]
  \(((E_{lon} \ E_{hin}) \ E_{\text{bodyn}})\)
  (otherwise \(E_{\text{defn}}\))

The evaluation of classify should proceed as follows. First the discriminant expression \(E_{\text{disc}}\) should be evaluated to the value \(V_{\text{disc}}\). Then \(V_{\text{disc}}\) should be matched against each of the clauses \(((E_{lo1} \ E_{hi1}) \ E_{\text{body1}})\) from top to bottom until one matches. The value matches a clause if it lies in the range between \(V_{loj}\) and \(V_{hij}\), inclusive, where \(V_{loj}\) is the value of \(E_{loj}\), and \(V_{hij}\) is the value of \(E_{hij}\). When the first matching clause is found, the value of the associated expression \(E_{\text{bodyj}}\) is returned. If none of the clauses matches \(V_{\text{disc}}\), the value of the default expression \(E_{\text{defn}}\) is returned.

Here are a few more examples of the the classify construct in action:

; Program 2
(program (a b c d)
  (classify (* c d)
   ((a (- (div (+ a b) 2) 1)) (* a c))
   (((+ (div (+ a b) 2) 1) b) (* b d))
   (otherwise (- d c)))))
; Program 3
(program (a)
  (classify a
    ((0 9) a)
    (((div 20 a) 20) (+ a 1))
    (otherwise (div 100 (- a 5)))))

Program 2 emphasizes that any of the subexpressions of classify may be an arbitrary expression that requires evaluations. In particular, the upper and lower bound expressions need not be integer literals. For instance, here are some examples of the resulting value returned by Program 2 for some sample inputs.

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>20</td>
<td>3</td>
<td>4</td>
<td>30</td>
</tr>
<tr>
<td>10</td>
<td>20</td>
<td>3</td>
<td>6</td>
<td>120</td>
</tr>
<tr>
<td>10</td>
<td>20</td>
<td>3</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

Program 3 emphasizes that (1) ranges may overlap (in which case the first matching range is chosen) and (2) expressions in clauses after the matching one are not evaluated. For instance, here are here are some examples of the resulting value returned by Program 3 for some sample inputs.

<table>
<thead>
<tr>
<th>a</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
</tr>
<tr>
<td>25</td>
<td>5</td>
</tr>
<tr>
<td>30</td>
<td>4</td>
</tr>
</tbody>
</table>

Your Task

Dexter has asked you to implement the classify construct in IBEX as syntactic sugar by extending the desugar function in desugar.scm with a clause for classify. You should use the abstract syntax functions in Fig. 2 to manipulate classify expressions. Your desugaring should only evaluate $E_{\text{disc}}$ once; to guarantee this, you will need to name the value with a “fresh” variable (one that does not appear elsewhere in the program). Use the name-not-in function described in Fig. 3 to choose a variable not in a given list.

You can test your implementation by invoking (test-classify). This will use env-run to run various programs containing classify expressions to make sure that they evaluate to an expected answer. If not, the original (sugared) program and the desugared program will be displayed.
(make-classify  disc clauses default)
Returns a classify construct with discriminant disc, clauses clauses, and default expression default.

(classify-discriminant classify-node)
Returns the discriminant of classify-node.

(classify-clauses classify-node)
Returns a list of the clauses of classify-node.

(classify-default classify-node)
Returns the default expression of classify-node.

(classify? node)
Returns #t if node is a classify node, and #f otherwise.

(classify-clause-lo classify-clause)
Returns the lower bound expression of classify-clause.

(classify-clause-hi classify-clause)
Returns the upper bound expression of classify-clause.

(classify-clause-body classify-clause)
Returns the body expression of classify-clause.

Figure 2: Abstract syntax for classify.

(name-not-in name names)
Returns the first “subscripted” version of name that is not an element of name list names. For example, (name-not-in 'a '(b c d)) returns a_1 and (name-not-in 'a '(a_2 a_4 a_1)) returns a_3. If name is already subscripted, the existing subscript is removed before computing the new one. For instance (name-not-in 'a_7 '(a_2 a_4 a_1)) returns a_3.

Figure 3: Specification for the name-not-in function.
Problem 4 [35]: Static Type-Checking of IBEX Programs

Dynamic vs. Static Type-Checking

In IBEX, when a primitive operator is applied to the wrong number or wrong types of arguments, an exception is thrown that reports the error. For example:

- Evaluating (+ 4) throws an exception with tag `primapply:wrong-num-args` and with information value (expected 2 got 1 in (+ 4)).
- Evaluating (+ 4 5 6) throws an exception with tag `primapply:wrong-num-args` and with information value (expected 2 got 3 in (+ 4 5 6)).
- Evaluating (+ 4 true) throws an exception with tag `primapply:wrong-arg-type` and with information value (expected int got true in (+ 4 true)).

Catching and reporting such type errors at run-time is run is called **dynamic type-checking**. Scheme and Logo are examples of languages in which all type-checking is dynamic.

A disadvantage of dynamic type-checking is that type errors are not caught until the program is run on actual arguments. Yet, without knowing the actual arguments, we can often deduce that a program will have a type error. Finding type errors in program without running it is called **static type-checking**. Java, Pascal, ML, and Haskell are examples of languages with static type-checking.\(^1\) Even in languages with static type-checking, certain kinds of checks must still be performed dynamically; for example:

- checking if an array subscript is out of bounds;
- checking if an attempt is made to take the head or tail of an empty list;
- checking if the second argument of a divide operator is zero;
- checking (in Java) to see if an object cast to a type is really of that type.

In many statically type-checked languages (such as Java and Pascal), it is necessary for the programmer to explicit annotate the program to indicate the types of various variables and expressions. However, some languages (such as ML and Haskell) are able to infer almost all type information without requiring the programmer to include explicit annotations. Such languages use an approach known as **type inference** to infer the types of expressions.

Static Type-Checking in IBEX

Because IBEX is such a simple language, it is easy to deduce the types of all expressions without any explicit type information. All IBEX program parameters are required to be integers, and the number and types of all IBEX primitive operators are known. This means that it is easy to determine the type of any tree of primitive applications and conditionals whose leaves are either literals and references to the program parameters. Assuming inductively that the type of the definition of a `bind` can be determined, it is also possible to handle expression trees with `bind` nodes whose leaves include variables declared by a `bind`.

For example, consider the following program:

\(^1\)The C programming language has a very crude notion of type that is used more to tell the compiler how big values are than it is used to guarantee the safety of a program. Indeed, it is so easy to "fool" the C type system that it can hardly be considered to be a statically type-checked language.
(program (a b c)
  (bind d (* b c)
   (if (< a d)
     (- a d)
     (+ a c))))

We can reason about the types in this program as follows:

- Assume program parameters a, b, and c denote integers.
- Since b and c are integers, and * maps two integers to an integer, d is an integer.
- Since a and d are integers, and < maps two integers to a boolean, (< a d) denotes a boolean.
- Since a and d are integers, and - maps two integers to an integer, (- a d) denotes an integer.
- Since (< a d) denotes a boolean and both (- a d) and (+ a c) denote integers, the conditional expression (if (< a d) (- a d) (+ a c)) denotes an integer.

Thus, we are able to reason that (1) the program has no type errors and (2) the program returns an integer.

If we change the * to = in the above program, we would discover a type error as follows:

- Since b and c are integers, and = maps two integers to a boolean, d is a boolean.
- In (< a d), < expects that both operands are integers. But d is a boolean, so this is a type error.

Note that the type error is discovered without having run-time values for a, b, and c.

The process of type checking in IBEX can be performed in a manner similar to evaluation via the environment model interpreter. In analogy with env-eval, we can write a function type-eval that takes two arguments: (1) an IBEX expression to be type-checked and (2) a type environment that binds names to types (rather than to values). Here a type is assumed to be represented as one of the Scheme symbols int, bool, sym, or string. If the expression is well-typed in the given environment (i.e., has no static type errors), the type-eval returns the type of the given expression. But if the expression contains a static type error, type-eval throws an exception indicating the kind of type error encountered. (The kinds of static type errors that should be caught are summarized in Fig. 4.) Using type-eval, it is easy to write a function (type-check program) that returns the type of the value computed by a program, assuming that all of its parameters are integers.

Note that it is considered a static type error for the two branches of an if expression to have different types even though an expression with such an “error” may not generate a dynamic type error. For example, (+ 1 (if (< 1 2) (+ 3 4) (< 5 6))) has a static type error but not a dynamic type error. In order to catch errors without running the program, we often have to add additional restrictions to our language that prohibit programs that would not give errors at runtime. In return, we get a guarantee that any program without a static type error will not generate a dynamic type error when it is executed.
<table>
<thead>
<tr>
<th>Kind of Type Error</th>
<th>Symbolic Tag</th>
<th>Information Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>The number of actual arguments $n$ in primitive application $primapp$ does not match expected number of arguments $k$.</td>
<td>$primapply\text{\text{-}wrong\text{-}num\text{-}args}$</td>
<td>(expected $k$ got $n$ in $primapp$)</td>
</tr>
<tr>
<td>The actual type $atype$ of a supplied argument does not match the expected type $etype$ for that argument in primitive application $primapp$.</td>
<td>$primapply\text{\text{-}wrong\text{-}arg\text{-}type}$</td>
<td>(expected $etype$ got $atype$ in $primapp$)</td>
</tr>
<tr>
<td>The test expression $testexp$ of an if expression denotes a non-boolean value.</td>
<td>$if\text{-}non\text{-}boolean\text{-}test$</td>
<td>$testexp$</td>
</tr>
<tr>
<td>Given an if expression (if $testexp$ thenexp elseexp), the type $thenty$ of $testexp$ does not match the type $elsety$ of $elseexp$.</td>
<td>$if\text{-}branch\text{-}mismatch$</td>
<td>(then: $thenexp$ has-type $thenty$ else: $elseexp$ has-type $elsety$)</td>
</tr>
<tr>
<td>A variable reference to name $name$ is unbound.</td>
<td>$unbound\text{-}variable$</td>
<td>$name$</td>
</tr>
<tr>
<td>The operator $rator$ of a primitive application is not recognized.</td>
<td>$unknown\text{-}primop$</td>
<td>$rator$</td>
</tr>
</tbody>
</table>

Figure 4: Exceptions that may be thrown during the static type-checking of and IBEX program.
Your Task

For this problem, you are to flesh out the definitions of the functions `type-check`, `type-eval`, and `static-primapply` within the file `type-check.scm`. These have the following contracts:

(type-check pgm)
If the IBEX program `pgm` has no static type errors, returns the type of the value that would be returned by `pgm` if it were run on the appropriate number of integer arguments. If `pgm` has static type errors, throws an exception indicating the first static type-checking error encountered in the type-checking process.

(type-eval exp tenv)
Assume that `tenv` is a type environment mapping the free variables in `exp` to their static types. Returns the result of type-checking `exp` in the context of the type environment `tenv`. If `exp` contains static type errors, throws an exception indicating the first static type-checking error encountered in the type-checking process.

(static-primapply primapp actual-types)
Returns the type of value that would result from applying the operator of `primapp` to the values whose types are given by the list of types `actual-types`. In cases where a type error is found, throws an appropriate exception. The first argument is the entire primitive application node `primapp` rather than just its operator because some of the exceptional cases require including the `primapp` node in their information value.

Notes:

- Your definitions of `type-check`/`type-eval` should have the same basic structure as `env-run/env-eval` in `env-eval.scm`, and as `simplify/simp` in Problem 4 of PS3. Your definition of `static-primapply` should have the same basic structure as `primapply` in `primops.scm`. For this reason, be sure to carefully study `env-run`, `env-eval`, and `primapply` before attempting this problem.

- You can test your type-checker by evaluating `(test-type-check)`. This will run your type checker on a suite of sample programs (both with and without type errors).

- To throw an exception like that for the wrong number of arguments, adapt the following idiom:

  (throw 'primapply:wrong-num-args
       (list 'expected E_expected
             'got E_actual
             'in E_primapp))

  Here, `E_expected` is a Scheme expression denoting the expected number of arguments, `E_actual` denotes the actual number of arguments, and `E_primapp` denotes the primitive application node.

- For determining the types of literals and finding type mismatches, you should use the following functions defined in `primops.scm`:
(define type-of
  (lambda (val)
    (cond ((ibex-integer? val) 'int)
          ((ibex-boolean? val) 'bool)
          ((ibex-symbol? val) 'sym)
          ((ibex-string? val) 'string)
          (else (throw 'type-of-unknown-value val))
    )))

(define type-error?
  (lambda (type val)
    (not (eq? type (type-of val))))))

- The wrong argument type exception requires finding an operand expression that is not of the expected type. For this purpose, you will probably find it handy to use the some2 higher-order list operator, which returns the first duple of corresponding items from two lists satisfying a given binary predicate. For example:

```
(some2 (lambda (x y) (> (* x y) 100))
   '(60 50 40 30 20 10)
   '(1 2 3 4 5 6))
; Returns (40 3)
```

As with some, some2 returns a distinguished none token if there is no duple satisfying the binary predicate. The none? function tests for the none token. For example:

```
(none? (some2 (lambda (x y) (> (* x y) 100))
               '(60 50 40 30 20 10)
               '(1 2 3 4 5 6))
; Returns #f because some2 returns (40 3)

(none? (some2 (lambda (x y) (> (* x y) 200))
               '(60 50 40 30 20 10)
               '(1 2 3 4 5 6))
; Returns #t because some2 returns the none token.
```
CS251 Problem Set 4
Due Monday, March 5

Names of Team Members:

Date & Time Submitted:

Collaborators (anyone you or your team collaborated with on the problem set):

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 [25]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem 4 [35]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>