Problem Set 5
Due: 6pm Friday, March 18

Revisions:
Mar 17: A recent change to the interpreter causes empty lists to be displayed as #e rather than (list); this affects the expected result in Problem 3. Mar 14: This is a complete version of PS5, containing all four problems.

Scheduling:
This assignment is due at 6pm on Fri. March 18, the day before Spring Break begins. For the purposes of lateness coupons, Spring Break does not exist. That is, if you turn in PS5 on Mon. Mar. 28, you need to use only one lateness coupon. PS6 will be handed out on Mon. Mar. 28, so there will be no assigned work during Spring Break.

Overview:
The purpose of this assignment is to give you practice with reasoning about the Bindex and Ibex languages and writing Ocaml programs that manipulate programs in these languages. It contains only Group Problems; there are no Individual Problems on this assignment.

Reading:
• Handout #28: Bindex: An Introduction to Naming
• Handout #30: Extending Bindex
• Handout #31: Valex: Primitives and Desugaring

Working Together:
Reminder: if you worked with a partner on a previous problem set and want to work with a partner on this assignment, you should try to find a different partner. If your schedule makes this impossible, please consult Lyn.

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 the due date. The packet should include:

1. a team header sheet (see the end of this assignment for the 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 solution to Problem 1;
3. your pencil-and-paper solutions to Problems 2a and 2b;
4. your final versions of Simprex.ml, SimprexEnvInterp.ml, and SimprexSubstInterp.ml for Problems 2c and 2d.
5. your pencil-and-paper desugaring rule for classify for Problem 3.
6. your final version of Valex.ml for Problems 3 and 4.

Each team should also submit a single softcopy (consisting of your final ps5 directory) to the drop directory ~cs251/drop/p3/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 the following commands in Linux in the account of the team member being used to store the code.
cd /students/username/cs251

cp -R ps5 ~cs251/drop/ps5/username/
Group Problem 1 [20]: Abstract Syntax Trees

Background

Fig. ?? shows the abstract syntax tree (AST) for the following BINDEX averaging program:

\[
\text{bindex (a b)} \\
\text{(bind c (+ a b)} \\
\text{(/ c 2))}
\]

Suppose we annotate each node of the abstract syntax tree with a triple \((fv, env, val)\) of the following pieces of information:

1. \(fv\): the set of free variables of the program or expression rooted at the node.
2. \(env\): The environment in which the node would be evaluated if the program were run on the actual parameters \(a = 3\) and \(b = 8\). (Write environments as sets of bindings of the form \(key \rightarrow value\).)
3. \(val\): The value that would result from evaluating the node in the environment \(env\).

Fig. ?? shows the AST for the averaging program annotated with this information. The name \(e0\) abbreviates the environment \(\{a \mapsto 3, b \mapsto 8\}\) and \(e1\) abbreviates the environment \(\{a \mapsto 3, b \mapsto 8, c \mapsto 11\}\).

Your Task

In this problem, your task is to draw a similar annotated AST for the following BINDEX program:

\[
\text{bindex (a b c)} \\
\text{(* (bind d (* a c)} \\
\text{  (bind e (~ d b)} \\
\text{    (/ (* b d) (+ e a)))} \\
\text{  (bind e (bind b (* 8 a)} \\
\text{    (~ b c))} \\
\text{    (+ e b))))}
\]

You should annotate each node of the abstract syntax tree with the same three pieces of information used in the average example above. In this case, assume that the program is run on the actual parameters \(a = 2\), \(b = 3\), and \(c = 5\).

Note: for this problem, you will need to use a large sheet of paper and/or to write very small. It is strongly recommended that you write the solution using pencil (not pen, so you can erase) and paper. Don’t waste your time attempting to format it on a computer with a drawing program.
Figure 1: An AST for the \texttt{avg} program.

Figure 2: \texttt{avg} program AST annotated with free variable, environment, and value information.
Group Problem 2 [50]: Simprex

Rae Q. Cerf of Toy-Languages-‖Us likes the sigma construct from Handout #30, but she wants something more general. In addition to expressing sums, she would also like to express numeric functions like factorial and exponentiation that are easily definable via simple recursion. The functions \( f \) that Rae wants to define all have the following form:

\[
  f(n) = \begin{cases} 
    z, & \text{if } n \leq 0 \\
    c(n, f(n-1)), & \text{if } n > 0
  \end{cases}
\]

Here, \( z \) is an integer that defines the value of \( f \) for any non-positive integer value and \( c \) is a binary combining function that combines \( n \) and the value of \( f(n-1) \) for any positive \( n \). Expanding the definition yields:

\[
  f(n) = c(n, c(n-1, c(n-2, \ldots c(2, c(1, z))))).
\]

For example, to define the factorial function, Rae uses

\[
  \begin{align*}
  z_{\text{fact}} &= 1 \\
  c_{\text{fact}}(n, a) &= n \times a.
  \end{align*}
\]

To define the exponentation function \( b^n \), Rae uses

\[
  \begin{align*}
  z_{\text{expt}} &= 1 \\
  c_{\text{expt}}(n, a) &= b \times a.
  \end{align*}
\]

In this case, \( c_{\text{expt}} \) ignores its first argument, but the fact that \( c_{\text{expt}} \) is called \( n \) times is important. As another example, Rae defines the sum of the squares of the integers between 1 and \( n \) using

\[
  \begin{align*}
  z_{\text{sos}} &= 0 \\
  c_{\text{sos}}(n, a) &= n^2 + a.
  \end{align*}
\]

Rae designs an extension to BINDEX named SIMPREX that adds a new \texttt{simprec} construct for expressing her simple recursions:\(^1\)

\[
  \texttt{(simprec } E_{\text{zero}} \texttt{ } (I_{\text{num}} \texttt{ } I_{\text{ans}} \texttt{ } E_{\text{combine}}) \texttt{ } E_{\text{arg}} \texttt{)}
\]

Evaluates \( E_{\text{arg}} \) to the integer value \( n_{\text{arg}} \) and \( E_{\text{zero}} \) to the integer value \( n_{\text{zero}} \). If \( n_{\text{arg}} \leq 0 \), returns \( n_{\text{zero}} \). Otherwise, returns the value

\[
  c(n_{\text{arg}}, c(n_{\text{arg}} - 1, c(n_{\text{arg}} - 2, \ldots c(2, c(1, n_{\text{zero}}))))).
\]

where \( c \) is the binary combining function specified by \( (I_{\text{num}} \texttt{ } I_{\text{ans}} \texttt{ } E_{\text{combine}}) \). This denotes a two-argument function whose two formal parameters are named \( I_{\text{num}} \) and \( I_{\text{ans}} \) and whose body is \( E_{\text{combine}} \). The \( I_{\text{num}} \) parameter ranges over the numbers from \( n_{\text{arg}} \) down to 1, while the \( I_{\text{ans}} \) parameter ranges over the answers built up by \( c \) starting at \( n_{\text{zero}} \). The scope of \( I_{\text{num}} \) and \( I_{\text{ans}} \) includes only \( E_{\text{combine}} \); it does not include \( E_{\text{zero}} \) or \( E_{\text{arg}} \).

Here are some sample SIMPREX programs:

\(^1\)This is closely related to the notion of \textbf{primitive recursive functions} defined in the theory of computation.
After completing her design, Rae is called away to work on another problem. Toy-Languages-Us is impressed with your CS251 background, and has hired you to implement the Simprex language, starting with a version of the Sigmex implementation. Your first week on the job, you are asked to complete the following tasks that Rae has specified in a memo she has written about finishing her project.

a. [10] Rae’s memo contains the following Simprex test programs. Give the results of running each of the programs on the argument 3. Show your work so that you may get partial credit if your answer is incorrect.

i. \( \text{Simprex (a) (Simprex 0 (b c (+ 2 c)) a)} \)

ii. \( \text{Simprex (x) (Simprex 0 (n sum (+ n (* x sum)) 4))} \)

iii. \( \text{Simprex (y) (Simprex 0 (a b (+ b (sigma c 1 a (* a c))) y)} \)

iv. \( \text{Simprex (n) (Simprex (Simprex (* n (- n 3)) (q r r) (* n n)) (c d (+ d (Simprex 0 (x sum (+ sum (- (* 2 x) 1))) c))) (Simprex -5 (a b (+ 1 b)) (* n n)))} \)

b. [10] Rae’s memo also contains the Simprex expression in Fig. ???. You should (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.

a. [22] You should implement the Simprex language by completing the following tasks, which Rae has listed in her memo:

i. [2] Extend the definition of sexpToExp in Simprex.ml to correctly parse Simprex expressions.

ii. [2] Extend the definition of expToSexp in Simprex.ml to correctly unparsse Simprex expressions.

iii. [3] Extend the definition of freeVarsExp in Simprex.ml to correctly determine the free variables of a Simprex expression.

iv. [6] Extend the definition of eval in SimprexEnvInterp.ml to correctly evaluate Simprex expressions using the environment model.

v. [3] Extend the definition of subst in Simprex.ml to correctly perform substitutions into Simprex expressions.

vi. [6] Extend the definition of eval in SimprexSubstInterp.ml to correctly evaluate Simprex expressions using the substitution model.
Figure 3: SIMPREX expression for part 2b.
Notes for part c:

- Before you begin the programming in this problem, you should study Appendix A on Bindex.
- In Simprec.ml, the exp type is defined to be:

```
and exp =
    Lit of int (* integer literal with value *)
  | Var of var (* variable reference *)
  | BinApp of binop * exp * exp (* primitive application with rator, rands *)
  | Bind of var * exp * exp (* bind name to value of defn in body *)
  | Sigma of var * exp * exp * exp (* name * lo * hi * body *)
  | Simprec of exp * var * var * exp * exp
      (* zero-exp * n-var * ans-var * comb-exp * arg-exp *)
```

The s-expression notation \((\text{simprec } E_{\text{zero}} \ (I_{\text{num}} \ I_{\text{ans}} \ E_{\text{combine}}) \ E_{\text{arg}})\) is represented in OCAML as:

```
Simprec (<exp for \(E_{\text{zero}}\>),
        <string for \(I_{\text{num}}\>),
        <string for \(I_{\text{ans}}\>),
        <exp for \(E_{\text{combine}}\>),
        <exp for \(E_{\text{arg}}\>)
```

For example, the expression \((\text{simprec } 1 \ (x \ a \ (* \ x \ a)) \ n)\) is represented in OCAML as:

```
Simprec(Lit 1, "x", "a", BinApp(Mul, Var "x", Var "a"), Var "n")
```

- In subparts (iv) and (vi), full credit will only be given for definitions that do not use explicit recursion. Instead, use the higher-order list functions in the ListUtils module. Partial credit will be awarded for correct definitions that use explicit recursion.
- You can test your functions for this part by using the testing functions illustrated in Fig. ??:

c. [8] Rae ends her memo with the observation that sigma is no longer a necessary construct in SIMPREX because it can be desugared into the simprec construct. In particular, she notes that the following `sexpToExp` clause in Simprec.ml,

```
| Seq [Sym "sigma"; Sym name; lox; hix; bodyx] ->
  Sigma (name, sexpToExp lox, sexpToExp hix, sexpToExp bodyx)
```

can be replaced by a clause of the following form:

```
| Seq [Sym "sigma"; Sym name; lox; hix; bodyx] ->
  let loVar = StringUtils.fresh "lo"
  and hiVar = StringUtils.fresh "hi"
  and loDecVar = StringUtils.fresh "lodec"
  and ansVar = StringUtils.fresh "ans" in
  Bind(loVar, sexpToExp lox,
       Bind(hiVar, sexpToExp hix,
            Bind(loDecVar, BinApp(Sub, Var loVar, Lit 1),
                 Simprec(Ezero,
                         name,
                         ansVar,
                         Ecombine,
                         Earg))))
```
# Simprex.freeVarsExpString "(simprec a (b c (+ b (* c d))) e)";;
- : Simprex.S.elt list = ["a"; "d"; "e"]

# Simprex.substString "(simprec a (b c (+ b (* c (/ a d)))) d)"
   "((a (+ b c)) (d (- b c)))";;
(simprec (+ b c) (b.0 c.1 (+ b.0 (* c.1 (/ (+ b c) (- b c)))))) (- b c)
- : unit = ()

# SimprexEnvInterp.runString "(simprex (x) (simprec 0 (n sum (+ n sum)) x))" [10];;
- : int = 55

# SimprexEnvInterp.runFile "fact.spx" [5];;
- : int = 120

# SimprexEnvInterp.runFile "expt.spx" [3;4];;
- : int = 81

# SimprexEnvInterp.runFile "sos.spx" [4];;
- : int = 30

# SimprexEnvInterp.repl();;

simprex> (simprec 0 (n sum (+ n sum)) 5)
15

simprex> (#quit)
done
- : unit = ()

(* SimprexSubstInterp can be tested similarly to SimprexEnvInterp. *)

Figure 4: Testing functions for the SIMPREX implementation.
As a puzzle, she has left it for you to figure out what the Ocaml expressions $E_{\text{zero}}$, $E_{\text{comb}}$, and $E_{\text{arg}}$ must be so that the new clause implements the correct behavior for $\sigma$.

Notes for part d:

- Rae’s code is commented out in Simprex.ml. You should begin this problem by removing the comments and instead commenting out the former $\sigma$ clause.
- The three Binds are used to avoid evaluating the given expressions $\text{lo}$, $\text{hi}$, and the new expression $\text{BinApp(Sub, Var loVar, Lit 1)}$ more than once.
- The fresh variables $\text{loVar}$, $\text{hiVar}$, $\text{lodecVar}$, and $\text{ansVar}$ are used to prevent unwanted variable capture in the problem.
- The problem is much easier to solve if you assume that the $\text{lo}$ expression evaluates to the integer 1. In this case, $\text{lodecVar}$ is bound to the value 0 and can be ignored. Half credit will be awarded if you correctly solve the problem making this assumption. Please indicate (via a comment in your code) that you are making this assumption.
- Solving the general problem (i.e., without the assumption that $\text{lo}$ evaluates to 1) is challenging. Do not invest too much time solving the general problem unless you like challenges.

Group Problem 3 [15]: Extending Valex with New Primitive Operators

The Valex language implementation is designed to make it easy to add new primitive operators to the language. In this problem, you are asked to add the following four primitive operators to Valex.

\[(\text{abs } n)\]
Returns the absolute value of the integer $n$. E.g.

\begin{verbatim}
  valex> (abs -17)
  17

  valex> (abs 42)
  42
\end{verbatim}

\[(\text{sqrt } n)\]
If $n$ is a non-negative integer, returns the integer square root $n$. The integer square root of a non-negative integer is the largest integer $i$ such that $i^2 \leq n$. Signals an error if $n$ is negative. E.g.

\begin{verbatim}
  valex> (sqrt 25)
  5

  valex> (sqrt 35)
  5

  valex> (sqrt 36)
  6
\end{verbatim}

\[(\text{between } \text{lo } \text{hi})\]
Assume $\text{lo}$ and $\text{hi}$ are integers. If $\text{lo} \leq \text{hi}$, returns a list of integers from $\text{lo}$ to $\text{hi}$, inclusive. Otherwise, returns the empty list.
valex> (between 1 20)
(list 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20)

valex> (between 3 7)
(list 3 4 5 6 7)

valex> (between 7 3)
#e ; Might be displayed as (list) in some Valex implementations

(reverse xs)
Assume xs is a list. Returns a list whose elements are the elements of xs in reversed order.
E.g.

valex> (reverse (list 1 2 3))
(list 3 2 1)

valex> (reverse (between 3 7))
(list 7 6 5 4 3)

valex> (reverse (list))
#e ; Might be displayed as (list) in some Valex implementations

All four primitives above can be added to VALEX by adding new Primop entries to the primop list in ps5-group/Valex.ml. Each primitive can be added with just a line or two of code. Study the other primitives to see how this is done.

Notes:

- Be careful to change the VALEX implementation in the ps5-group directory, not the one in the valex directory!

- To use any parts of the VALEX interpreter, you must first execute the following in OCAML:
  ```
  #cd "/students/username/cs251/ps5-group"
  #use "load-valex.ml"
  ```

- An easy way to test your new primitives is to test them interactively in a read-eval-print loop launched by invoking ValexEnvInterp.repl(). Alternatively, you can you can extend the test entries in ValexTestEntries.valexEntries (which is located in the file ValexInterpTest.ml) to include programs containing the primitives.
Group Problem 4 [20]: 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 VALEX and is very impressed by the cond construct. He decides that it would be neat to extend VALEX 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:

```valex
(valex (grade)
  (classify grade
    ((90 100) 'A')
    ((80 89) 'B')
    ((70 79) 'C')
    ((60 69) 'D')
    (otherwise 'F'))))
```

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

In general, the classify construct has the following form:

```valex
(classify E
disc
  ((E_lo, E_hi, ) E_body)
  ...
  ((E_lo_n, E_hi_n, ) E_body_n)
  (otherwise E_dflt))
```

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

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

```valex
; Program 2
(valex (a b c d)
  (classify (* c d)
    ((a (- (/ (+ a b) 2) 1)) (* a c))
    (((+ (/ (+ a b) 2) 1) b) (* b d))
    (otherwise (- d c))))

; Program 3
(valex (a)
  (classify a
    ((0 9) a)
    (((/ 20 a) 20) (+ a 1))
    (otherwise (/ 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.
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 some examples of the resulting value returned by Program 3 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>

**Your Task**

Dexter has asked you to implement the `classify` construct in VALEX as syntactic sugar. You should begin by writing on paper desugaring rules that desugar `classify` into other VALEX constructs; turn in these rules with your hardcopy submission. Then you should implement your rule(s) by extending the `desugarRules` function in `ps5-group/Valex.ml` with clauses for `classify`.

**Notes:**

- Be careful to change the VALEX implementation in the `ps5-group` directory, not the one in the `valex` directory!

- Your desugaring should only evaluate $E_{disc}$ 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 `StringUtils.fresh` to create a fresh variable.

- You may want to treat differently the cases where $E_{disc}$ is an identifier and when it is not an identifier.

- For testing the desugaring of your `classify` construct, use one of the following two approaches:

1. Invoke the `Valex.desugarString` function on a string representing the expression you want to desugar. For example:

   ```ml
   # Valex.desugarString "(&& (|| a b) (|| c d))"
   (if (if a #t b) (if c #t d) #f)
   - : unit = ()
   
   # Valex.desugarString "(list 1 2 3)";;
   (prep 1 (prep 2 (prep 3 #e)))
   - : unit = ()
   ```

2. Use `ValexEnvInterp.repl()` to launch a read-eval-print loop and use the `#desugar` directive to desugar an expression. For example:
# ValexEnvInterp.repl();

valex> (#desugar (&& (|| a b) (|| c d)))
(if (if a #t b) (if c #t d) #f)

valex> (#desugar (list 1 2 3))
(prep 1 (prep 2 (prep 3 #e)))

- There are several ways to test the evaluation of your desugared classify construct:

  1. The test entries in ValexTestEntries.valexEntries (in the file ValexInterpTest.ml) include a few programs that contain classify. These will be automatically tested when you invoke testEnvInterp(). Just because your implementation passes the existing test cases does not necessarily mean it is completely correct. You may want to add more test entries to increase your confidence.

  2. You can use ValexEnvInterp.runString to interactively evaluate programs containing classify.

  3. You can interactively evaluate expressions containing classify in a read-eval-print loop launched via ValexEnvInterp.repl().

The first approach is recommended since you only have to type in each program once rather than every time you want to test it.
Appendix A: BINDEX Implementation

In Problem 2, you are asked to extend the BINDEX language. Before attempting the programming parts of this problem, you should study the code for the implementation of the BINDEX language, which can be found in `~/cs251/bindex` after you perform `cvs update -d`. There are three files to study: `Bindex.ml`, `BindexEnvInterp.ml`, and `BindexSubstInterp.ml`.

To use any of the functions defined within files in the `bindex` directory, you should first execute the following directives in OCAML:

```ocaml
#cd "~/students/username/cs251/bindex"
#use "load-bindex.ml"
```

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

```ocaml
# open Bindex;;
# setToList (freeVarsExp (stringToExp "(bind c (+ a b) (* c d))"));;
- : Bindex.S.elt list = ["a"; "b"; "d"]

# subst1 (stringToExp "a") (stringToExp "b") (stringToExp "c") (stringToExp "d");;
- : Bindex.exp = Bind ("a.1", BinApp (Add, BinApp (Add, Var "a", Var "b"), BinApp (Add, Var "b", Var "c")), BinApp (Add, Var "a.1", Var "c"))

# StringUtils.print (expToString (subst1 (stringToExp "a")) (stringToExp "a"));;
(bind a.2 (+ (+ b c) (+ b c)) (* a.2 a.2))

# BindexEnvInterp.run (Pgm(["a", "b"], BinApp(Add, Var "a", Var "b"))) [3;7];;
- : int = 10

# BindexEnvInterp.runString "((bind a (+ a a) (+ a (* a a))))" [5];;
- : int = 30

# BindexEnvInterp.runFile "avg.bdx" [3;7];;
(* Assume that the file avg.bdx contains an averaging program *)
- : int = 5

# BindexEnvInterp.repl();;

bindex> (+ 1 2)
3

bindex> (bind a (+ 1 2) (+ a (* a a)))
12

bindex> (#args a 3) (b 4) (c 5))

bindex> (+ a (* b c))
23

bindex> (#quit)
done
```

15
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 your 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 [15]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem 2 [50]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem 3 [15]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem 4 [20]</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>