CS251 Programming Languages **Handout # 32**
Prof. Lyn Turbak **March 27, 2007**
Wellesley College ***Revised March 27, 2007***

Problem Set 6
Due: 11:59pm Wednesday, April 4

Revisions:

Mar 27:

- In the pset submission description, (a) you should also turn in testing transcripts for parts 1c and 1d and (b) change `Simp.ml` to `BindexPartialEval.ml` for Problem 2.
- The notes for Problem 1 neglected to mentioned that you should perform `#use "load-simprex.ml"` to load the SIMPREX implementation.
- The notes for Problem 2 neglected to mentioned that you should use `BindexEnvInterp.binApply` to apply an operator to two integers when partially evaluating a binary application.

Overview:
The purpose of this assignment is to give you practice with reasoning about the BINDEX and VALEX 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 #30: Bindex: An Introduction to Naming
- Handout #31: Extending Bindex
- Handout #33: Valex: Type Checking 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 11:59pm 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 solutions to Problems 1a and 1b;
3. your final versions of `Simprex.ml`, `SimprexEnvInterp.ml`, and `SimprexSubstInterp.ml` for Problems 1c and 1d;
4. your testing transcripts for Problems 1c and 1d;
5. your final version of `BindexPartialEval.ml` for Problem 2;
6. your pencil-and-paper desugaring rule for `classify` for Problem 4;
7. your final version of `Valex.ml` for Problems 3 and 4.

Each team should also submit a single softcopy (consisting of your final `ps6` 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 ps6 ~cs251/drop/ps6/username/
```

Group Problem 1 [45]: SIMPREX

Rae Q. Cerf of Toy-Languages-Я-Us likes the `sigma` construct from Handout #31, 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 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

$$z_{fact} = 1$$
$$c_{fact}(n, a) = n \times a.$$

To define the exponentation function b^n, Rae uses

$$z_{expt} = 1$$
$$c_{expt}(n, a) = b \times a.$$

In this case, c_{expt} ignores its first argument, but the fact that c_{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

$$z_{sos} = 0$$
$$c_{sos}(n, a) = n^2 + a.$$

Rae designs an extension to BINDEX named SIMPREX that adds a new `simprec` construct for expressing her simple recursions:[1]

> `(simprec` E_{zero} `(`I_{num} I_{ans} $E_{combine}$`)` E_{arg}`)`
> Evaluates E_{arg} to the integer value n_{arg} and E_{zero} to the integer value n_{zero}. If $n_{arg} \leq 0$, returns n_{zero}. Otherwise, returns the value
>
> $$c(n_{arg},\ c(n_{arg}-1,\ c(n_{arg}-2,\ \ldots\ c(2,\ c(1,\ n_{zero}))))).$$
>
> where c is the binary combining function specified by `(`I_{num} I_{ans} $E_{combine}$`)`. This denotes a two-argument function whose two formal parameters are named I_{num} and I_{ans} and whose body is $E_{combine}$. The I_{num} parameter ranges over the numbers from n_{arg} down to 1, while the I_{ans} parameter ranges over the answers built up by c starting at n_{zero}. The scope of I_{num} and I_{ans} includes only $E_{combine}$; it does not include E_{zero} or E_{arg}.

Here are some sample SIMPREX programs:

```
;; Program that calculuates the factorial of n
(simprex (n) (simprec 1 (x a (* x a)) n))

;; Exponentiation program raising base b to power p
(simprex (b p) (simprec 1 (n ans (* b ans)) p))

;; Program summing the squares of the numbers from 1 to x
(simprex (x) (simprec 0 (y z (+ (* y y) z)) x))
```

[1] This is closely related to the notion of **primitive recursive functions** defined in the theory of computation.

After completing her design, Rae is called away to work on another problem. Toy-Languages-Я-Us is impressed with your CS251 background, and has hired you to implement the SIMPREX language, starting with a version of the SIGMEX implementation. (SIGMEX is the name of the language that results from extending BINDEX with the `sigma` construct from Handout #31.) 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. [1] `(simprex (a) (simprec 0 (b c (+ 2 c)) a))`

ii. [2] `(simprex (x) (simprec 0 (n sum (+ n (* x sum))) 4))`

iii. [3] `(simprex (y) (simprec 0 (a b (+ b (sigma c 1 a (* a c)))) y))`

iv. [4]
```
(simprex (n)
        (simprec (simprec (* n (- n 3)) (q r r) (* n n))
                 (c d (+ d (simprec 0 (x sum (+ sum (- (* 2 x) 1))) c)))
                 (simprec -5 (a b (+ 1 b)) (* n n))))
```

b. **[10]** Rae's memo also contains the SIMPREX expression in Fig. 1. You should (1) circle every free variable reference occurrence and (2) draw a line from every bound variable reference occurrence to the binding occurence of that reference.

```
(bind  a  (simprec  a

                    (  a  b  (+  b  (sigma  b  a  b

                                            (*  a  b  ))))

                    b  )

  (simprec (sigma  c  a  b  (+  a  (*  b  c  )))

           (  b  c  (bind  a  (*  a  b  )

                              (*  b  (+  a  c  ))))

           (-  a  c  )))
```

Figure 1: SIMPREX expression for part 2b.

c. **[20]** Rae has created a skeleton implementation of SIMPREX by modifying the files for the SIGMEX (= BINDEX + `sigma`) implementation to contain stubs for the `simprec` construct. Her modified files, which are named `Simprex.ml`, `SimprexEnvInterp.ml`, and `SimprexSubstInterp.ml`, can be found in your `ps6-group` directory.

Finish the implementation of the SIMPREX language by completing the following six tasks, which Rae has listed in her memo:

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

ii. **[2]** Extend the definition of `expToSexp` in `Simprex.ml` to correctly unparse `simprec` expressions.

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

iv. **[5]** Extend the definition of `eval` in `SimprexEnvInterp.ml` to correctly evaluate `simprec` expressions using the environment model.

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

vi. **[5]** Extend the definition of `eval` in `SimprecSubstInterp.ml` to correctly evaluate `simprec` expressions using the substitution model.

Notes for part c:

- Before you begin the programming in this problem, you should study Appendix A on BINDEX.
- Perform `#use "load-simprex.ml"` to load the SIMPREX implementation.
- 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 * num-var * ans-var * comb-exp * arg-exp *)
```

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

```
Simprec (<exp for E_zero>,
         <string for I_num>,
         <string for I_ans>,
         <exp for E_combine>,
         <exp for E_arg>)
```

  For example, the expression `(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 should test your functions for this part by using the tests similar to the ones illustrated in Fig. 2. You should include at least the test cases shown in the figure, but should also develop some test cases of your own. Turn in a transcript of your test cases for this part.

```
# 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 2: Sample tests for the SIMPREX implementation.

d. [**5**] 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 `Simprex.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(E_zero,
                           name,
                           ansVar,
                           E_comb,
                           E_arg))))
```

As a puzzle, she has left it for you to figure out what the OCAML expressions E_{zero}, E_{comb}, and E_{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 part by removing the comments and instead commenting out the former `sigma` clause.
- The three `Bind`s are used to avoid evaluating the given expressions `lo`, `hi`, and the new expression `BinApp(Sub, Var loVar, Lit 1)` more than once.
- The fresh variables `loVar`, `hiVar`, `lodecVar`, and `ansVar` are used to prevent unwanted variable capture in the problem.
- The problem is much easier to solve if you assume that the `lo` expression evaluates to the integer 1. In this case, `lodecVar` is bound to the value 0 and can be ignored. Partial 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 `lo` evaluates to 1) is challenging. Do *not* invest too much time solving the general problem unless you like challenges.
- You should test your desugaring by executing some sample programs that use the `sigma` construct.

Group Problem 2 [25]: Partial Evaluation

Avoiding Magic Constants

It is good programming style to avoid "magic constants" in code by explicitly calculating certain constants from others. For instance, consider the following two BINDEX programs for converting years to seconds:

```
; Program 1
(bindex (years)
  (* 31536000 years))

; Program 2
(bindex (years)
  (bind seconds-per-minute 60
    (bind minutes-per-hour 60
      (bind hours-per-day 24
        (bind days-per-year 365 ; ignore leap years
          (bind seconds-per-year (* seconds-per-minute
                                    (* minutes-per-hour
                                       (* hours-per-day
                                          days-per-year)))
            (* seconds-per-year years)))))))
```

The first program uses the magic constant 31536000, which is the number of seconds in a year.[2] The second program shows how this constant is calculated from simpler constants. By showing the process by which `seconds-per-year` is calculated, the second program is a more robust and well-documented software artifact. Calculated constants also have the advantage that they are easier to modify. Although the numbers in the above program aren't going to change, there are many so-called "constants" built into a program that change over its lifetime. For instance, the size of word of computer memory, the price of a first-class stamp, and the rate for a certain tax bracket

[2]It is worth noting that this number is approximately $\pi \times 10^7$. So a century is approximately $\pi \times 10^9$ seconds, which means that π seconds is approximately one nano-century!

are all numbers that could be hard-wired into programs but which might need to be updated in future version of the software.

However, magic constants can have performance advantages. In the above programs, the program with the magic constant performs one multiplication, while the other program performs four multiplications. If performance is critical, the programmer might avoid the clearer style and instead opt for magic constants.

Partial Evaluation

Is there a way to get the best of both approaches? Yes! We can write our program in the clearer style, and then automatically transform it to the more efficient style via a process known as **partial evaluation**. Partial evaluation transforms an input program into a **residual** program that has the same meaning by performing computation steps that would otherwise be performed when running the program. Any computation steps that can be performed during partial evaluation are steps that do not need to be performed when the residual program is run later. In most cases, the residual program has better run-time performance than the original program.

For instance, we can use partial evaluation to systematically derive the first program above from the second. We begin via a step known as **constant propagation**, in which we substitute the four constants at the top of the second program into their references to yield:

```
(bindex (years)
  (bind seconds-per-minute 60
    (bind minutes-per-hour 60
      (bind hours-per-day 24
        (bind days-per-year 365 ; ignore leap years
          (bind seconds-per-year (* 60 (* 60 (* 24 365)))
            (* seconds-per-year years)))))))
```

Next, we eliminate the now-unnecessary first four bindings via a step known as **dead code removal**:

```
(bindex (years)
  (bind seconds-per-year (* 60 (* 60 (* 24 365)))
    (* seconds-per-year years)))
```

We can now perform the three multiplications involving manifest integers in a step known as **constant folding**:

```
(bindex (years)
  (bind seconds-per-year 31536000
    (* seconds-per-year years)))
```

Finally, another round of constant propagation and dead code removal yields the first program:

```
(bindex (years)
  (* 31536000 years))
```

It is not possible to eliminate bindings whose definition ultimately depends on the program parameters. Nevertheless, it is often possible to partially simplify such definitions. For example, consider:

```
(bindex (a)
  (bind b (* 3 4)
    (bind c (+ a (- 15 b))
      (bind d (/ c b)
        (* d c)))))
```

The transformation techniques described above can simplify this program to:

```
(bindex (a)
  (bind c (+ a 3)
    (bind d (/ c 12)
      (* d c))))
```

In this example, `(+ a (- 15 b))` cannot be replaced by a number (because the value of `a` is unknown), but it can be simplified to the **residual expression** `(+ a 3)`. Similarly, `(/ c b)` is transformed to the residual expression `(/ c 12)` and `(bind b ...)` is transformed to the residual expression

```
(bind c (+ a 3)
  (bind d (/ c 12)
    (* d c)))
```

Your Task

In this problem, your task is to write a function `partialEval` that performs partial evaluation on a BINDEX program. Given a BINDEX program, `partialEval` should return another BINDEX program that has the same meaning as the original program, but which also satisfies the following properties:

1. The program should not contain any `bind` expressions in which a variable is bound to an integer literal.
2. The program should not contain any binary applications in which an arithmetic operator is applied to two integer literals. There are two exceptions to this property: the program may contain binary applications of the form `(/ n 0)` or `(% n 0)`, since performing these applications would cause an error in the partial evaluation process.

It is possible to write separate functions that perform the constant propagation, constant folding, and dead-code elimination steps, but it is tricky to get them to work together to perform all simplifications. It turns out that it is much more straightforward to perform all three kinds of simplification at the same time in a single walk over the expression tree.

By analogy with `BindexEnvInterp.run` and `BindexEnvInterp.eval`, partial evaluation can be peformed by a pair of functions:

val `partialEval: Bindex.pgm -> Bindex.pgm`
Returns a partially evaluated version of the given BINDEX program.

val `peval: Bindex.exp -> int Env.env -> Bindex.exp`
Given a BINDEX expression *exp* and a partial evaluation environment *env*, returns the partially evaluated version of *exp*. The partial evaluation environment contains name/value bindings for names whose integer values are known.

Your goal is to implement simplification by fleshing out these two function definitions in the file `BindexPartialEval.ml`.

Note that there is a correspondence between `run`/`eval` in `BindexEnvInterp` and `partialEval`/`peval`. `peval` is effectively a version of `eval` that evaluates as much of an expression as it can based on the "partial" environment information it is given. Because bindings for some names may be missing in the environment, `peval` cannot always evaluate every expression to the integer it denotes and in some cases must instead return a residual expression that will determine the value when the program is executed. Because of this, `peval` must always return an expression rather than an integer; even in the case where it can determine the value of an expression, that value must be expressed as an integer literal node, not an integer.

Notes

- Perform `#use "load-peval.ml"` to load the partial evaluator.
- Use `BindexEnvInterp.binApply` to apply an operator to two integers.
- Divisions and remainders whose second operands are zero must be left in the program. Such programs will encounter divide-by-zero errors when they are later executed. For example,

```
(bindex (a)
  (bind b (* 3 4)
    (bind c (/ b (- 12 b))
      (* c b))))
```

  should be transformed to:

```
(bindex (a)
  (bind c (/ 12 0)
    (* c 12)))
```

- In some cases it would be possible to perform more aggressive simplification if you took advantage of algebraic properties like the associativity and commutativity of addition and multiplication. To simplify this problem, *you should not use any algebraic properties of the arithmetic operators.* For example, you should not transform `(+ 1 (+ a 2))` into `(+ 3 a)`, but should leave it as is. You should not even perform "obvious" simplifications like `(+ 0 a)` $\Rightarrow$ `a`, `(* 1 a)` $\Rightarrow$ `a`, and `(* 0 a)` $\Rightarrow$ `0`. Although the first two of these simplification are valid, the last is unsafe in the sense that it can change the meaning of a program. For instance, `(* 0 (/ a b))` cannot be simplified to `0`, because it does not preserve the meaning of the program in the case where `b` is `0` (in which case evaluating the expression should give an error).
- You may assume that the programs given to your simplifier do *not* contain the `sigma` or `simprex` constructs from Problem 1.
- You can use the `testPartialEval` function (which takes a string representation of a BINDEX program) to test your partial evaluator. For example:

```
# testPartialEval "(bindex () (+ 1 2))";;
(bindex () 3)
- : unit = ()

# testPartialEval "(bindex (a)
                     (bind b (* 3 4)
                       (bind c (/ b (- 12 b))
                         (* c b))))";;
(bindex (a) (bind c (/ 12 0) (* c 12)))
- : unit = ()

# testPartialEval "(bindex (a)
                     (+ (* (+ 1 2) a)
                        (+ (* 3 4)
                           (+ (* 0 a)
                              (+ (* 1 a)
                                 (+ 0 a))))))";;
(bindex (a) (+ (* 3 a) (+ 12 (+ (* 0 a) (+ (* 0 a) (+ 0 a))))))
- : unit = ()
```

- You can also test your partial evaluator by evaluating `test()`. This applies your partial evaluator to all the test entries in the list `testEntries` in the file `BindexPartialEvalTest.ml`. The entries in this list are by no means exhaustive. You are strongly encouraged to add more entries to this list.

Group Problem 3 [10]: 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.

(abs *n*)
Returns the absolute value of the integer *n*. E.g.

```
valex> (abs -17)
17

valex> (abs 42)
42
```

(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.

```
valex> (sqrt 25)
5

valex> (sqrt 35)
5

valex> (sqrt 36)
6
```

(between *lo hi*)
Assume *lo* and *hi* are integers. If $lo \leq hi$, returns a list of integers from *lo* to *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 `ps6-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 `ps6-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/ps6-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 (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:

```
(classify E_disc
  ((E_lo1  E_hi1) E_body1)
     ⋮
  ((E_lon  E_hin) E_bodyn)
  (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 $((E_{lo_i}\ E_{hi_i})\ E_{body_i})$ from top to bottom until one matches. The value matches a clause if it lies in the range between V_{lo_i} and V_{hi_i}, inclusive, where V_{lo_i} is the value of E_{lo_i}, and V_{hi_i} is the value of E_{hi_i}. When the first matching clause is found, the value of the associated expression E_{body_i} 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 the `classify` construct in action:

```
; 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.

a	b	c	d	result
10	20	3	4	30
10	20	3	6	120
10	20	3	5	2

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.

a	result
0	0
5	5
10	11
20	21
25	5
30	4

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 `ps6-group/Valex.ml` with clauses for `classify`.

Notes:

- Be careful to change the VALEX implementation in the `ps6-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:

```
# 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 exisiting 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

Problems 1 and 2 involve the BINDEX language (or extensions thereof). Before attempting the programming parts of these problems, 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:

```
#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:

```
# open Bindex;;

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

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

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

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

# BindexEnvInterp.runString "(bindex (a) (bind b (* a a) (+ a b)))" [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
```

Group Problem Header Page
Please make this the first page of your hardcopy submission for group problems.

CS251 Problem Set 6 Group Problems
Due 11:59pm Wednesday, April 4

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.*

Part	**Time**	**Score**
General Reading		
Problem 1 [45]		
Problem 2 [25]		
Problem 3 [10]		
Problem 4 [20]		
Total		