CS251 Programming Languages **Handout # 19**
Prof. Lyn Turbak **Saturday, February 23**
Wellesley College

Problem Set 4

Due: Saturday, March 2, 2002

Reading: The final version of Local Binding (Handout #15) and IBEX (Handout #18).

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

Submission:

- Problems 1, 3(a) and 3(d) are pencil-and-paper problems that only needs to appear in your hardcopy submission.
- For Problems 2 and 3, your softcopy submission should include a copy of your entire `ps4` directory.
- Your hardcopy submission for Problem 2 should be your final version of the file `primops.scm`.
- Problem 3(b), your hardcopy submission should be your final version of `env-eval.scm`.
- Problem 3(c), your hardcopy submission should be your final version of `subst-eval.scm`.
- Problem 3(e), your hardcopy submission should be your final version of `desugar.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:

```
;; 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 [30]: `bindpar` and `bindseq`

a. [**3**] 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.

```
;; 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**] When running the program from part a, how many times are each of the five binary operators `+`, `-`, `*`, `div`, and `mod` performed in (1) a call-by-value interpreter and (2) a call-by value interpreter?

c. [**8**] Redo parts a and b, except using a version of the program in which every `bindseq` has been replaced by `bindpar`.

d. [**4**] Write the result of desugaring the programs from both part a and part c 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 α-renaming bound variables when necessary.

e. [**10**] Figure 1 shows a clause for handling `bind` within the `subst-cbn` function of the call-by-name 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 Figure 1. In a call-by-*value* substitution model interpreter, it turns out that only the `else` branch of this `if` would ever be taken. Explain why this is so.
3. Give a BINDEX *program* (not 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.

```
((bind? exp)
 (let* ((name (bind-name exp))
        (defn (bind-defn exp))
        (body (bind-body exp))
        ;; BODY-FVS is the set of variables in the body
        ;; that appear free outside the BIND
        (body-fvs (set-difference (free-vars (bind-body exp))
                                  (set-singleton (bind-name exp))))

        ;; NEW-ENV is ENV with any binding for NAME removed
        (new-env (env-remove (list (bind-name exp)) env))

        ;; CAPTURABLES is the set of all new free vars that
        ;; are introduced into the copy of the BIND body
        ;; returned by SUBST.
        (capturables
         (foldr set-union
                (set-empty)
                (map (lambda (fv)
                       (let ((probe (env-lookup fv new-env)))
                         (if (unbound? probe)
                             (set-singleton fv)
                             (free-vars probe))))
                     body-fvs)))

        )
   (if (set-member? name capturables)
       ; Then clause
       (let ((new-name (name-not-in name capturables)))
         (make-bind new-name
                    (subst-cbn env defn)
                    (subst-cbn new-env (rename1 name new-name body))))
       ; Else clause
       (make-bind name
                  (subst-cbn env defn)
                  (subst-cbn new-env body)))))
```

Figure 1: Clause for handling `bind` within the `subst-cbn` function of the call-by-name BINDEX and IBEX interpreter.

Problem 2 [15]: Extending IBEX with string operations

Strings

The IBEX language implementation is designed to make it fairly easy to add new primitive dataypes 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:

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

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

;; Extend TYPE-OF with a new STRING type.
(define type-of
  (lambda (val)
    (cond ((mini-integer? val) 'int)
          ((mini-boolean? val) 'bool)
          ((mini-symbol? val) 'sym)
          ((mini-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:

```
(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:

(strlen *str*)
Returns the length (number of characters in) the string *str*.

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

(str+ *str1 str2*)
Returns the string that has all the characters of `str1` followed by all of those of `str2`. For example, `(str+ "ab" "bcd")` yields `"abbcd"`.

(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 `substr:index-out-of-bounds` and whose value is the offending index. For example:

```
(substr 1 1 "abcdef") ; Returns "a"
(substr 2 5 "abcdef") ; Returns "bcde"
(substr 3 2 "abcdef") ; Returns ""
(substr 0 5 "abcdef") ; Throws exception substr:index-out-of-bounds 0
(substr 2 7 "abcdef") ; Throws exception substr:index-out-of-bounds 7
```

Your Task

Your task is to extend the IBEX interpreter with the above four string operations by adding appropriate bindings to the `primop-env` environment in the file `primops.scm`. Each binding associates a name with a **primitive operator descriptor** created by invoking `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.

Notes:

- To use any parts of the IBEX interpreter, you must first evaluate the following in Scheme:

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

- You will need to use Scheme string operations in your implementation. For documentation on these, consult the section on strings in the *Revised*^{5} *Report on the Algorithmic Language Scheme (R5RS)*. You can access *R5RS* on-line from the CS251 home page.
- You can test your implementation by invoking `(test-strings)`.

Problem 3 [55]: Extending IBEX **with** `loop`

The `loop` *construct*

Due to your extensive experience with IBEX in CS251, you have been elected head of the IBEX Users Group, a worldwide consortium of IBEX programmers. At your most recent consortium meeting, there was much grumbling from attendees about the lack of expressiveness of IBEX. As one dissatisfied IBEX programmer put it, "Sure, `sigma` 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 IBEX if you will help with the implementation.

Ida quickly designs the following IBEX `loop` construct:

$$
\begin{array}{l}
\texttt{(loop ((}sv_1\ E_{init_1}\ E_{update_1}\texttt{)} \\
\qquad\qquad\quad \vdots \\
\qquad\quad \texttt{(}sv_n\ E_{init_n}\ E_{update_n}\texttt{))} \\
\qquad\ E_{test} \\
\quad E_{body}\texttt{)}
\end{array}
$$

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 a true 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 `loop` is the updater expression, the test expression, and the body expression. The scope does *not* include the initializer expressions.

For example, the following IBEX program calculates the factorial of `n`:

```
(program (n)
  (loop ((i n (- i 1))
         (prod 1 (* i prod)))
        (> i 0)
    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.

t	`i`	`prod`
0	5	1
1	4	5
2	3	20
3	2	60
4	1	120
5	0	120

As another example, here are an IBEX program that calculates the nth Fibonacci number, and an iteration table that summarizes the iteration for $n = 6$.

```
(program (n)
  (loop ((i 0 (+ 1 i))
         (fib_i 0 fib_{i+1})
         (fib_{i+1} 1 (+ fib_i fib_{i+1})))
        (< i n)
    fib_i))
```

t	i	fib_i	fib_i+1
0	0	0	1
1	1	1	1
2	2	1	2
3	3	2	3
4	4	3	5
5	5	5	8
6	6	8	13

Note that when evaluating the updater expressions $\texttt{fib}_{i+1}$ and $\texttt{(+ fib}_i\ \texttt{fib}_{i+1}\texttt{)}$ to determine the state for time $t + 1$, *both* of these expressions are evaluated with respect to the values of the state variables $\texttt{fib}_{i+1}$ and $\texttt{fib}_{i+1}$ 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. Parta (a) and (d) are pencil-and-paper problems; parts (b), (c), and (e) require extending the version of the IBEX interpreter in `~/cs251/ps4`. To use any parts of the IBEX interpreter, you must first evaluate the following in a Scheme interpreter:

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

Parts (a) – (d) are independent and can be done in any order. The code for part (e) can be written independently of the other parts, but testing it requires the completion of one of parts (b) or (c). The abstract syntax you need for manipulating `loop` expressions is summarized in Figure 2.

(**make-loop** *vars inits updates test body*)
Constructs and returns a new `loop` expression with state variables *vars*, initializers *inits*, updaters *updates*, test expression *test*, and body expression *body*.

(**loop-vars** *loop-node*)
Returns a Scheme list of the state variables in *loop-node*.

(**loop-inits** *loop-node*)
Returns a Scheme list of the initializer expressions in *loop-node*.

(**loop-updates** *loop-node*)
Returns a Scheme list of the updater expressions in *loop-node*.

(**loop-test** *loop-node*)
Returns the test expression of *loop-node*.

(**loop-body** *loop-node*)
Returns the body expression of *loop-node*.

(**loop?** *node*)
Returns `#t` if *node* is a `loop` expression, and `#f` otherwise.

Figure 2: Abstract syntax for the `loop` expression.

a. [5]: Variable Scoping

Consider the following (contrived) IBEX expression:

```
(loop ((a a (+ a 1))
       (b b (- b a)))
      (<= a b)
  (loop ((a b (* a 2))
         (b 0 (+ b (loop ((a a (div a 2))
                          (b 1 (* b a)))
                         (= a 0)
                      b))))
        (> a b)
    (+ a b)))
```

Copy this program onto a sheet of paper, and draw a line between every variable reference and the variable declaration to which it refers. Indicate free variables by drawing circles around them.

b. [12]: Environment Model Evaluation

In this problem, you are to extend the `env-eval` function in `env-eval.scm` to correctly specify the evaluation of the `loop` construct in the environment model. You should do this by fleshing out the three expressions E_1, E_2, and E_3 in the following skeleton:

```
((loop? exp)
 (let  ((vars (loop-vars exp))
        (inits (loop-inits exp))
        (updates (loop-updates exp))
        (test (loop-test exp))
        (body (loop-body exp)))
   (env-eval body
             (iterate E1
                      E2
                      E3))))
```

The above skeleton uses the following higher-order `iterate` function. The `iterate` function is similar to `generate`, except that rather than returning a list of iterated values, it returns the last value of an iteration (the one for which the `done?` predicate is true):

```
(define iterate
  (lambda (seed next done?)
    (if (done? seed)
        seed
        (iterate (next seed) next done?))))
```

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?
- Be sure to use `mini-bool-to-scheme-bool` to convert an IBEX boolean to a Scheme boolean.
- Use the environment operations specified in Handout 13 to manipulate environments.
- You can test your `loop` clause for `env-eval` by evaluating `(test-loop-env-eval)`, which uses `env-eval` to evaluate a suite of benchmarks containing `loop`.
- The `desugar` function used in `env-run` has already been extended to handle `loop`.

c. [8]: Substitution Model Evaluation

In this problem, you are to extend the `subst-eval` function in `subst-eval.scm` to correctly specify the evaluation of the `loop` construct in the substitution model. You should do this by fleshing out the three expressions E_1, E_2, and E_3 in the following skeleton:

```
((loop? exp)
 (let  ((vars (loop-vars exp))
        (inits (loop-inits exp))
        (updates (loop-updates exp))
        (test (loop-test exp))
        (body (loop-body exp)))
   (let ((state E1))
     (if (mini-bool-to-scheme-bool
           (subst-eval (subst state test)))
          (subst-eval (make-loop vars
                                 E2
                                 updates
                                 test
                                 body))
          E3))))
```

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.
- In your solution, you will need to use the `subst` function in `subst.scm`, which has already been extended to handle `loop` appropriately. Note that `subst` depends on functions in `rename.scm` and `free-vars.scm`, both of which files have already been extended to handle `loop` appropriately.
- You can test your `loop` clause for `subst-eval` by evaluating `(test-loop-subst-eval)`, which uses `subst-eval` to evaluate a suite of benchmarks containing `loop`.
- The `desugar` function used in `subst-run` has already been extended to handle `loop`.

d. [10]: Desugaring `least` into `loop`

Ida notes that many iteration constructs can be desugared into an appropriate `loop` expression. As an example, she invents the following `least` construct, whose abstract syntax is manipulated by the functions in Figure 3.

```
(least var body)
```

(make-least *var body*)
Constructs and returns a new `least` expression with variables *vars* (which should be a symbol) and body expression *body*.

(least-var *least-node*)
Returns the variable of *loop-node*.

(least-body *least-node*)
Returns the body of *loop-node*.

(least? *node*)
Returns `#t` if *node* is a `least` expression, and `#f` otherwise.

Figure 3: Abstract syntax for the `least` expression.

Rather than explaining the meaning of the `least` construct, she instead shows you how the `desugar` function can be extended to desugar `least` into `loop`:

```
;; DESUGAR clause for LEAST
((least? exp)
   (let ((var (least-var exp))
         (body (least-body exp)))
     (make-loop (list var)
                (list (make-literal 0))
                (list (make-primapp '+
                                    (list (make-varref var)
                                          (make-literal 1))))
                (make-primapp 'not (list (desugar body)))
                (make-varref var))))
```

i. [**3**] Based on the above desugaring, give an English description for the meaning of `(least var body)`. Your description should be very concise.

ii. [**4**] What are the values of the following expressions using `least`?

- `(least x (> (* x x) 100))`

-
  ```
  (least i (>= (* i (least j (<= (div 100 (+ j 1))
                                 i)))
               80))
  ```

iii. [**3**] Briefly explain the key advantage of implementing `least` as syntactic sugar rather than as a core IBEX construct (like `if`, `bind`, or `loop`).

e. [20]: Desugaring `sigma` into `loop`

Inspired by Ida's `least` construct, you decide to extend IBEX with the `(sigma` E_{lo} E_{hi} E_{body}`)` construct from Problem Set 3. Rather than implementing `sigma` "from scratch", as you did in Problem Set 3, you instead implement it as syntactic sugar by rewriting all `sigma` expressions into expressions using `loop`.

You should do this problem in two parts:

- Write a desugaring rule (like those in Handout #18), that specifies how to rewrite the expression `(sigma` E_{lo} E_{hi} E_{body}`)` into an expression that uses `loop` in addition to any kernel IBEX constructs. The desugared expression should evaluate each of E_{lo} and E_{hi} 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.
- Extend the `desugar` function in the file `desugar.scm` so that it implements your desugaring rule from part (1) and correctly desugars `sigma` into `loop`.

  Notes:

  - To create "fresh" variables in your implementatation, you should use the `name-not-in` function described in Figure 4 and defined in `rename.scm`.
  - You can test your desugaring by evaluating `(test-sigma-desugaring` *`run`*`)`, where `run` is one of `env-run` or `subst-run`. (You only need one of these to be working to test your desugaring.)
  - Feel free to define any auxiliary functions that you find helpful.

(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 4: Specification for the `name-not-in` function.

Extra Credit 1 [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 $E_{disc}$
  (($E_{lo_1}$ $E_{hi_1}$) $E_{body_1}$)
              ⋮
  (($E_{lo_n}$ $E_{hi_n}$) $E_{body_n}$)
  (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 $((E_{lo_i}\ E_{hi_i})\ E_{body_i})$ 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
(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.

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 IBEX as syntactic sugar. You should begin by writing one or more desugaring rules that desugar `classify` into other IBEX constructs. Then you should implement your rule(s) by extending the `desugar` function in `desugar.scm` with a clause for `classify`. You should use the abstract syntax functions in Figure 5 to manipulate `classify` expressions. 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 the `name-not-in` function described in Figure 4 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.

The hardcopy submission for this problem should include your final version of the file `desugar.scm`.

(**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 5: Abstract syntax for `classify`.

Problem Set Header Page
Please make this the first page of your hardcopy submission.

CS251 Problem Set 4

Due Saturday, March 2

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.*

Part	**Time**	**Score**
General Reading		
Problem 1 [30]		
Problem 2 [15]		
Problem 3 [55]		
Extra Credit [30]		
Total		