CS251 Programming Languages **Handout # 16**
Prof. Lyn Turbak **Friday, February 16**
Wellesley College

Problem Set 3
Due: February 23, 2001

Reading: Carefully read the class notes on Simple Interpretation (Handout #13), and Simple Naming (Handout #15). Additionally, skim *SICP* 4.1, which describes a **metacircular interpreter** for Scheme. It is more complex than the simple interpreters we are studying right now, but in two more weeks we will be building up to an interpreter of this complexity.

Submission:

- Problems 1 and 2 are pencil-and-paper problems that only need to appear in your hardcopy submission.
- For Problems 3 and 4, your softcopy submission should include a copy of your entire `ps3` directory.
- Your hardcopy submission for Problem 3 should include the following files: `env-eval.scm`, `free-vars.scm`, `subst-eval.scm`, and `rename.scm`.
- Your hardcopy submission for Problem 4 should be the file `simplify.scm`.

Problem 0: Studying BINDEX
All of the problems on this problem set involve the BINDEX language discussed in class or extensions to this language. Before attempting the problems, you should study the code for the implementation of the BINDEX language, which can be found in `~/cs251/ps3` 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 BINDEX works.
To use any of the functions defined within files in the `ps3` directory, you should evaluate the following in Scheme:

```
(cd "~/cs251/ps3")
(load "load-bindex.scm")
```

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

```
;; Run the averaging program on the inputs 3 and 8
;; under the environment model
(env-run '(program (a b)
            (bind c (+ a b)
              (div c 2)))
         '(3 8))
5
```

```
;; Run the averaging program on the inputs 3 and 8
;; under the substitution model
(subst-run '(program (a b)
              (bind c (+ a b)
                (div c 2)))
           '(3 8))
 5

;; Calculate free variables of an expression.
(free-vars '(bind c (+ a b) (* c d)))
 (a b 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) '(3 5)) '(bind c (+ a b) (* c d)))
 (bind c (+ 3 b) (* c d))
```

Since BINDEX programs are represented as s-expressions, they can be named so that they are easily reusable:

```
(define avg
  '(program (a b)
     (bind c (+ a b)
       (div c 2))))

(env-run avg '(3 8))
 5

(env-run avg '(20 10))
 15
```

Of course, when writing Scheme programs that manipulate BINDEX programs you should only use the abstract syntax operators! For example:

```
(program-formals avg)
 (a b c)

(bind? (program-body avg))
 #t

(bind-name (program-body avg))
 c
```

Do not use any "raw" Scheme list operations to manipulate BINDEX programs! That is, you should not use `car` and `cdr` to access the components of a node, nor should you use `cons`, `list`, `append`, etc. to create nodes. (However, you may use these operations to manipulate lists of formal parameters, lists of binding names and definitions, etc.)

When experimenting with `env-run`, it is helpful to trace the function `env-run` to get a sense for how computation proceeds. For example:

```
(trace env-eval)

(env-run avg '(3 8))

[Entering #[compound-procedure 31 env-eval]
    Args: (bind c (+ a b) (div c 2))
          ((a 3) (b 8))]
[Entering #[compound-procedure 31 env-eval]
    Args: (+ a b)
          ((a 3) (b 8))]
[Entering #[compound-procedure 31 env-eval]
    Args: b
          ((a 3) (b 8))]
[8
      <== #[compound-procedure 31 env-eval]
    Args: b
          ((a 3) (b 8))]
[Entering #[compound-procedure 31 env-eval]
    Args: a
          ((a 3) (b 8))]
[3
      <== #[compound-procedure 31 env-eval]
    Args: a
          ((a 3) (b 8))]
[11
      <== #[compound-procedure 31 env-eval]
    Args: (+ a b)
          ((a 3) (b 8))]
[Entering #[compound-procedure 31 env-eval]
    Args: (div c 2)
          ((c 11) (a 3) (b 8))]
[Entering #[compound-procedure 31 env-eval]
    Args: 2
          ((c 11) (a 3) (b 8))]
[2
      <== #[compound-procedure 31 env-eval]
    Args: 2
          ((c 11) (a 3) (b 8))]
[Entering #[compound-procedure 31 env-eval]
    Args: c
          ((c 11) (a 3) (b 8))]
[11
      <== #[compound-procedure 31 env-eval]
    Args: c
          ((c 11) (a 3) (b 8))]
[5
      <== #[compound-procedure 31 env-eval]
    Args: (div c 2)
          ((c 11) (a 3) (b 8))]
[5
      <== #[compound-procedure 31 env-eval]
    Args: (bind c (+ a b) (div c 2))
          ((a 3) (b 8))]
;Value: 5
```

Similarly, when experimenting with `subst-run` and `eval-run`, it is helpful to trace the functions `subst` and `subst-eval`. For example:

```
(trace subst)

(trace subst-eval)

(subst-run avg '(3 8))

[Entering #[compound-procedure 28 subst]
    Args: ((a 3) (b 8))
          (bind c (+ a b) (div c 2))]
[Entering #[compound-procedure 28 subst]
    Args: ((a 3) (b 8))
          (div c 2)]
[Entering #[compound-procedure 28 subst]
    Args: ((a 3) (b 8))
          2]
[2
      <== #[compound-procedure 28 subst]
    Args: ((a 3) (b 8))
          2]
[Entering #[compound-procedure 28 subst]
    Args: ((a 3) (b 8))
          c]
[c
      <== #[compound-procedure 28 subst]
    Args: ((a 3) (b 8))
          c]
[(div c 2)
      <== #[compound-procedure 28 subst]
    Args: ((a 3) (b 8))
          (div c 2)]
[Entering #[compound-procedure 28 subst]
    Args: ((a 3) (b 8))
          (+ a b)]
[Entering #[compound-procedure 28 subst]
    Args: ((a 3) (b 8))
          b]
[8
      <== #[compound-procedure 28 subst]
    Args: ((a 3) (b 8))
          b]
[Entering #[compound-procedure 28 subst]
    Args: ((a 3) (b 8))
          a]
[3
```

```
      <== #[compound-procedure 28 subst]
    Args: ((a 3) (b 8))
          a]
[(+ 3 8)
      <== #[compound-procedure 28 subst]
    Args: ((a 3) (b 8))
          (+ a b)]
[(bind c (+ 3 8) (div c 2))
      <== #[compound-procedure 28 subst]
    Args: ((a 3) (b 8))
          (bind c (+ a b) (div c 2))]
[Entering #[compound-procedure 29 subst-eval]
    Args: (bind c (+ 3 8) (div c 2))]
[Entering #[compound-procedure 29 subst-eval]
    Args: (+ 3 8)]
[Entering #[compound-procedure 29 subst-eval]
    Args: 8]
[8
      <== #[compound-procedure 29 subst-eval]
    Args: 8]
[Entering #[compound-procedure 29 subst-eval]
    Args: 3]
[3
      <== #[compound-procedure 29 subst-eval]
    Args: 3]
[11
      <== #[compound-procedure 29 subst-eval]
    Args: (+ 3 8)]
[Entering #[compound-procedure 28 subst]
    Args: ((c 11))
          (div c 2)]
[Entering #[compound-procedure 28 subst]
    Args: ((c 11))
          2]
[2
      <== #[compound-procedure 28 subst]
    Args: ((c 11))
          2]
[Entering #[compound-procedure 28 subst]
    Args: ((c 11))
          c]
[11
      <== #[compound-procedure 28 subst]
    Args: ((c 11))
          c]
[(div 11 2)
      <== #[compound-procedure 28 subst]
    Args: ((c 11))
          (div c 2)]
[Entering #[compound-procedure 29 subst-eval]
    Args: (div 11 2)]
[Entering #[compound-procedure 29 subst-eval]
    Args: 2]
[2
      <== #[compound-procedure 29 subst-eval]
    Args: 2]
[Entering #[compound-procedure 29 subst-eval]
    Args: 11]
[11
      <== #[compound-procedure 29 subst-eval]
    Args: 11]
[5
      <== #[compound-procedure 29 subst-eval]
    Args: (div 11 2)]
[5
      <== #[compound-procedure 29 subst-eval]
    Args: (bind c (+ 3 8) (div c 2))]
;Value: 5
```

The file `bindex-examples.scm` contains a few simple programs to experiment with, but you are encouraged to write some of your own as well. The `bindex-examples.scm` file also contains an implementation of a simple test suite that will test an evaluator on a list of programs and examples and compare the results to the expected results. You are encourage to add new programs and test cases to your own local copy of `bindex-examples.scm`.

The `test-run` function is used to test a program evaluator (such as `subst-run` or `env-run`) on the test cases in the test suite. For example:

```
(test-run env-run)

Running inc on (3) gives 4. OK!
Running c2f on (100) gives 212. OK!
Running c2f on (0) gives 32. OK!
Running c2f on (-40) gives -40. OK!
Running calc on (20) gives 10. OK!
Running calc on (31) gives 15. OK!
Running avg on (3 8) gives 5. OK!
Running avg on (20 10) gives 15. OK!
Running test-bindseq on (3) gives 63. OK!
Running test-bindpar on (3) gives 37. OK!
Running test-bindseq2 on (3) gives 6. OK!
Running test-bindpar2 on (3) gives 149. OK!
Running test-ast on (5 4 2) gives 42. OK!
Running test-unbound on (1 2) gives (bindex-error unbound-variable c). OK!
Running test-div-by-0 on (2 1 0) gives (bindex-error division-by-zero (div 3 0)). OK!
Running test-mod-by-0 on (2 1 0) gives (bindex-error division-by-zero (mod 3 0)). OK!
Done.
```

The `test-run` function complains if the actual result does not match the expected result specified in test-suite. For instance, if we change the line

```
(list 'avg avg '(3 8) 5)
```

in `test-suite` to instead be the (incorrect) line

```
(list 'avg avg '(3 8) 6)
```

then running `(test-run env-run)` would indicate what it thinks is an error as follows:

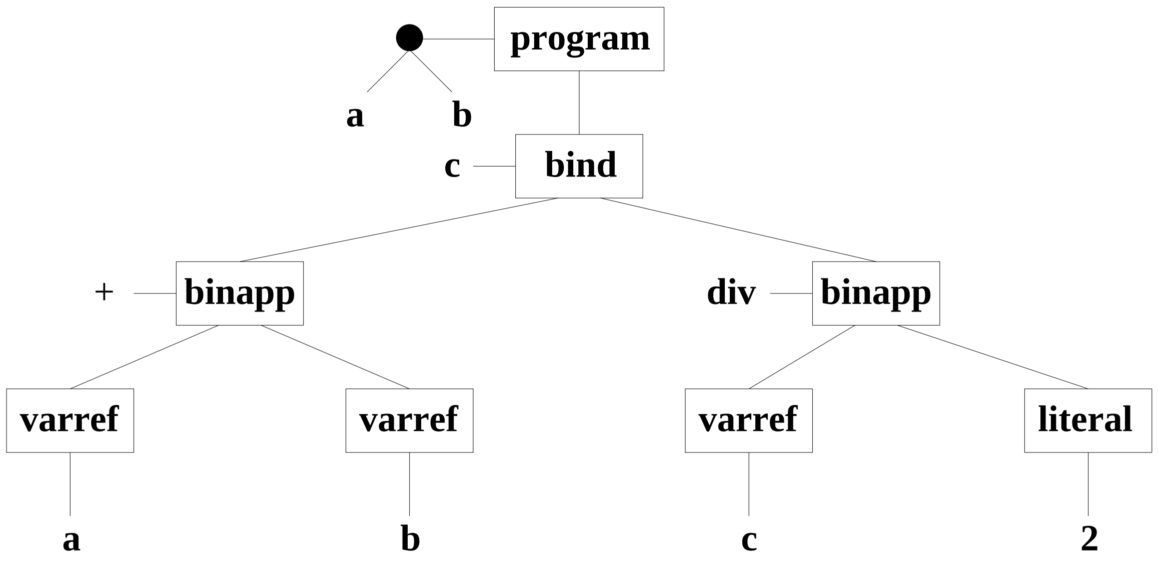
```
Running avg on (3 8) gives 5 ***ERROR!*** Expected 6
```

Problem 1 [20]: Abstract Syntax Trees

Consider the following BINDEX averaging program:

```
(program (a b)
  (bind c (+ a b)
    (div c 2)))
```

Here is the abstract syntax tree (AST) for this program:

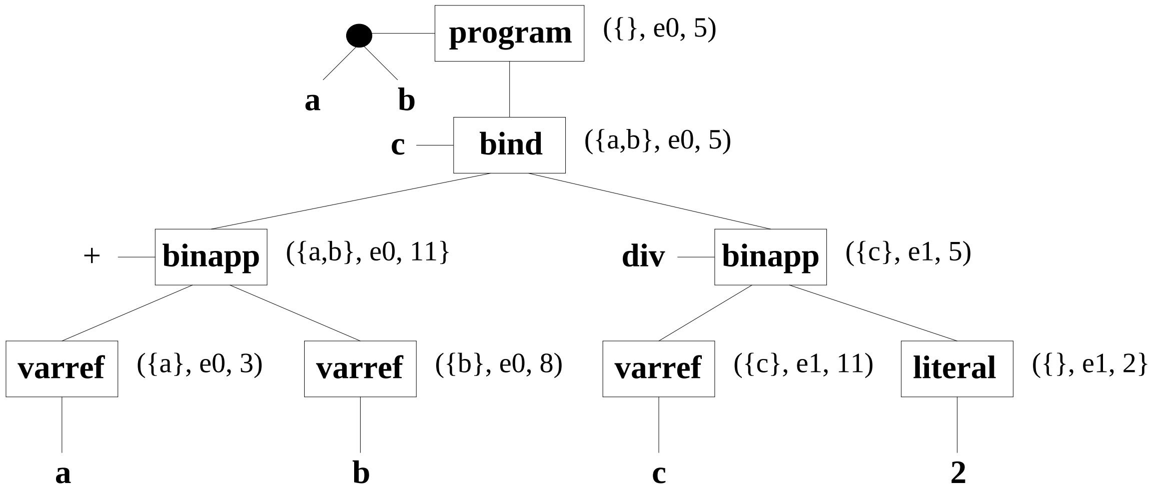


Note that the multiple parameters of the program are shown branching off a single solid node that stands for the sequence of parameters.

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 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* = *value*.)
3. *Val*: The value that would result from evaluating the node in the environment *Env*.

The following picture shows the AST for the averaging program annotated with this information. The name *e0* abbreviates the environment $\{a = 3, b = 8\}$ and *e1* abbreviates the environment $\{a = 3, b = 8, c = 11\}$.



In this problem, you are to draw a similar annotated AST for the following BINDEX program:

```
(program (a b c)
  (* (bind d (* a c)
       (bind e (- d b)
         (div (* b d) (+ e a))))
     (bind e (bind b (* 12 a)
               (- b c))
       (div 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.

Problem 2 [20]: bind*

An interesting aspect of interpreters is that they encourage exploring the space of design choices when implementing a language feature. As an example of this, we will study some design choices involved in extending BINDEX with `bind*`, a new binding construct that allows specifying mulitple name/definition bindings at the same time.

The concrete syntax for `bind*`, which is patterned after that of Scheme's `let` expression, is as follows:

```
(bind* (
         (name_1  defn_1)
           ⋮
         (name_n  defn_n)
        )
   body)
```

The `bind*` construct can specify any number of bindings between a name and a corresponding definition expression. Each such binding appears as a parenthesized list of name and definition; the list of all bindings is itself parenthesized. As with `bind` in BINDEX and `let` in Scheme, the `bind*` construct also specfies a body expression.

Abstractly, `bind*` expressions are manipulated by the following abstract syntax:

(make-bind* *names defns body*)
Returns a `bind*` node whose bindings are between the names in *names* and the corresponding expressions in *defns* and whose body is the expression *body*. It is an error if *names* and *defns* do not have the same length.

(bind*-names *node*)
Returns the names in the bindings of the `bind*` expression *node*.

(bind*-defns *node*)
Returns the definition expressions in the bindings of the `bind*` expression *node*.

(bind*-body *node*)
Returns the body expression of the `bind*` expression *node*.

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

Here is a concrete example of a program P using `bind*`:

```
(program (s t)
  (bind* ((s (+ (* 10 t) s))
          (t (+ (* 10 s) t)))
    (+ s t)))
```

What is the meaning of `bind*`? As with BINDEX's `bind` and Scheme's `let`, let's assume that the scope of all the bound names includes the body expression, and that the value of a `bind*` expression relative to an environment is the value of its body relative to the environment extended with bindings between the names and the values of their corresponding definitions.

However, this specification leaves an important question unanswered: does the scope of one name include the definition expressions of any of the other bindings? For instance, in the above example, does the `s` in `(* 10 s)` refer to the `s` in the previous binding or to the program parameter named `s`? This is a design decision that must be made when extending the BINDEX interpreter to handle `bind*`. It turns out there are many possibilities, including ones that may not seem obvious at first. For instance it is possible that the `t` in `(* 10 t)` could refer to the `t` in the binding below it rather than to the program parameter `t`!

Let's consider some of the possibilities in the context of the following skeleton of the definition of `env-eval` extended with a clause for `bind*`:

```
(define env-eval
  (lambda (exp env)
    (cond ···
          ((bind*? exp)
           (env-eval (bind*-body exp)
                     (fold2 (lambda (name defn e)
                              (env-bind name
                                        (env-eval defn env1)
                                        env2))
                            env
                            (bind*-names exp)
                            (bind*-defns exp))))
          ···)))
```

The meaning of the `bind*` construct depends on whether *fold2* is `foldl2` or `foldr2`[1], whether *env1* is `env` or `e`, and whether *env2* is `env` or `e`.

For each of the following combinations of assumptions, indicate the result of evaluating program P on the input values 1 and 2 under those assumptions. That is, what integer would be returned by `(env-eval` P `'(1 2))`? Show your work for partial credit. (This is a pencil and paper problem; you should be able to determine the answers without actually extending the BINDEX interpreter. *Hint:* Draw lots of pictures!)

fold2	*env1*	*env2*	value of P run on 1 & 2
foldl2	e	e	
foldl2	e	env	
foldl2	env	e	
foldl2	env	env	
foldr2	e	e	
foldr2	e	env	
foldr2	env	e	
foldr2	env	env	

[1]The `foldl2` and `foldr2` functions are versions of `foldl` and `foldr` that use a ternary operator to accumulate a result over a pair of lists starting with an initial value. They are defined the Simple Naming notes (Handout #15).

Problem 3 [30]: sigma

The `sigma` *Construct*

In this problem, you will extend the BINDEX interpreter to implement a new summation construct:

(sigma *var lo hi body*)
Assume that *var* is a variable name, *lo* and *hi* are expressions denoting integers, and *body* is an expression that may refer to *var*. Return the sum of *body* evaluated at all values of the index variable *var* ranging from *lo* up to *hi*, inclusive. This sum would be expressed in traditional mathematical summation notation as:

$$\sum_{var=lo}^{hi} body.$$

If the value of *lo* is greater than that of *hi*, the sum is 0.

Here are some examples of `sigma` in action:

```
(sigma k 1 6 k)
```
; 1 + 2 + 3 + 4 + 5 + 6 = 21

```
(sigma k 3 5 (* k k))
```
; $3^2 + 4^2 + 5^2 = 50$

```
(sigma k 3 2 (* k k))
```
; evaluates to 0

```
(sigma i 2 5
  (sigma j i 4
    (* i j)))
```
*; (2*2) + (2*3) + (2*4) + (3*3) + (3*4) + (4*4) = 55*

The `sigma` construct can be manipulated via the following abstract syntax operations:

(make-sigma *var lo hi body*)
Returns a `sigma` node whose index variable is *var*, whose lower bound is *lo*, whose upper bound is *hi*, and whose body expression is *body*.

(sigma-var *sigma-node*)
Returns the index variable of *sigma-node*.

(sigma-lo *sigma-node*)
Returns the lower bound expression of *sigma-node*.

(sigma-hi *sigma-node*)
Returns the upper bound expression of *sigma-node*.

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

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

Your Task

Your goal is to extend the BINDEX interpreter so that it appropriately handles the `sigma` construct. This involves extending several functions within the interpreter to handle `sigma`, as detailed below[2]:

a Extend the definition of `free-vars` in `free-vars.scm` to correctly determine the free variables of a `sigma` expression.

b Extend the definition of `env-eval` in `env-eval.scm` to correctly evaluate `sigma` expressions via the environment model.

c Extend the definition of `subst` in `subst.scm` to correctly perform substitutions into `sigma` expressions.

d Extend the definition of `subst-eval` in `subst-eval.scm` to correctly evaluate `sigma` expressions via the substitution model.

e Extend the definition of `rename` in `rename.scm` to correctly rename `sigma` expressions.

Notes

- Start your problem by loading the entire BINDEX interpreter for Problem Set 3 as described in Problem 0.
- Loading the bindex interpreter automatically loads the file `sigma.scm`, which contains the following nullary functions for testing your extensions: `test-sigma-free-vars`, `test-sigma-env-eval`, `test-sigma-subst`, `test-sigma-subst-eval`, and `test-sigma-rename`. The function `test-sigma` tests all of these.
- Implementing parts (c) and (e) requires you to perform a summation as the `sigma`-bound variable ranges over the integer values from the lower bound to the upper bound. There are many ways to implement this summation, but a particularly elegant approach is to use some of the higher-order list functions from `"~/cs251/util/list-ops.scm"`. These are automatically loaded when you load the BINDEX interpreter.

[2]Implementing `sigma` also requires modifying the `desugar` function. However, since we have not yet discussed desugaring, this has already been done for you.

Problem 4 [30]: Program Simplification

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
(program (years)
  (* 31536000 years))

; Program 2
(program (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. 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 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.

Program Simplification

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 **program simplification**. In program simplification, we rewrite a program into another one that has the same meaning by performing computation steps that would otherwise be performed when running the program. Any steps we can perform during simplification are steps that are avoided later; in most cases, this improves the run-time performance of the program.

For instance, we can use program simplification 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:

```
(program (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**:

```
(program (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**:

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

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

```
(program (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:

```
(program (a)
  (bind b (* 3 4)
    (bind c (+ a (- 15 b))
      (bind d (div c b)
        (* d c)))))
```

The simplification techniques described above can simplify this program to:

```
(program (a)
  (bind c (+ a 3)
    (bind d (div c 12)
      (* d c))))
```

Your Task

In this problem, your task is to write a function `simplify` that performs simplification on a BINDEX program by using the constant propagation, constant folding, and dead-code elimination steps illustrated above. Given a BINDEX program, `simplify` 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 `(div` *n* `0)` or `(mod` *n* `0)`, since these cannot be simplified by the constant folding 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 `env-run` and `env-eval`, simplification can be peformed by a pair of functions `simplify` and `simp`:

(simplify *pgm*)
Returns the simplified version of the given BINDEX program *pgm*.

(simp *exp env*)
Given a BINDEX expression *exp* and an simplification environment *env*, returns the simplified version of *exp*. The simplification environment contains name/value bindings for names whose values are known.

Your goal is to implement simplification by fleshing out the following skeleton for these two functions:

```
(define simplify
  (lambda (pgm)
    ;; flesh out these details
    ))

(define simp
  (lambda (exp env)
    (cond ((literal? exp)
           ;; code for handling literal case
           )
          ((varref? exp)
           ;; code for handling variable reference case
           )
          ((binapp? exp)
           ;; code for handling binary application case
           )
          ((bind? exp)
           ;; code for handling bind case
           )
          (else (error "SIMP: unrecognized expression: " exp))
          )))
```

The correspondence between `env-run`/`env-eval` and `simplify`/`simp` is not coincidental. Indeed, `simp` is effectively a version of `env-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, `simp` 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, `simp` 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

- To do this problem, you should flesh out the skeletons for the `simplify` and `simp` functions in `~/cs251/ps3/simplify.scm`.
- Loading `~/cs251/ps3/simplify.scm` automatically loads `~/cs251/ps3/simplify-test.scm`. This latter file contains a testing function for the simplifier, which you can invoke via `(test-simplify)`.
- 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,

  ```
  (program (a)
    (bind b (* 3 4)
      (bind c (div b (- 12 b))
        (* c b))))
  ```

  should be transformed to:

  ```
  (program (a)
    (bind c (div 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" identities like `(* 0 a)` $\Rightarrow$ `0`, `(* 1 a)` $\Rightarrow$ `a`, `(+ 0 a)` $\Rightarrow$ `a`. See the extra credit problem if you are interested in more aggressive simplification.
- You may assume that the programs given to your simplifier do *not* contain the `sigma` construct from Problem 3.

Extra Credit 1 [20]: Implementing Environment Operations Appendix A of Handout #13 presents a contract for environments. The following environment functions are said to be the core functions in the contract: `env-empty`, `env-bind`, `env-lookup`, `env-names`, and `unbound?`. Show that the remaining functions in the contract (`env-make`, `bindings->env`, `env-values`, `env->bindings`, `env-remove`, `env-extend`,`env-merge`) can be implemented in terms of the core functions.

Extra Credit 2 [up to 30]: Improving Simplification In order to constrain Problem 4, only very particular simplifications were allowed. But there are many other meaning-preserving simplifications that can be performed that make use of algebraic properties. Extend the `simplify` function to be more aggressive by implementing as many algebraic simplifications that you can think of.

You must be careful to justify that each of your simplifications preserves the meaning of the program. Many "obvious" simplifications can actually change the meaning of a program. For instance, `(* 0 (div 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).

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

CS251 Problem Set 3
Due Friday, February 23

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 [20]		
Problem 2 [20]		
Problem 3 [30]		
Problem 4 [30]		
Extra Credit 1 [20]		
Extra Credit 2 [30]		
Total		