Programming Languages: Application and Interpretation

Version 5.3.4

May 8, 2013

This is the documentation for the software accompanying the textbook **Programming Languages: Application and Interpretation** (PLAI). The full book can be found on the Web at:

http://www.cs.brown.edu/~sk/Publications/Books/ProgLangs/

This package contains the following languages:

1 PLAI Scheme

```
#lang plai
```

PLAI Scheme is derived from the `scheme` language. In addition, it includes the `define-type` and `type-case` forms and testing support.

1.1 Defining Types: `define-type`

```
(define-type type-id variant ...)

variant = (variant-id (field-id contract-expr) ...)
```

Defines the datatype *`type-id`*. A constructor *`variant-id`* is defined for each variant. Each constructor takes an argument for each field of its variant.

The value of each field is checked by its associated *`contract-expr`*. A *`contract-expr`* may be an arbitrary predicate or a contract.

In addition to the contructors, a `define-type` expression also defines:

- a predicate *`type-id?`* that returns `true` for instances of the datatype, and `false` for any other value,
- for each variant, a predicate *`variant-id?`* that returns `true` when applied to a value of the same variant and `false` for any other value,
- for each field of each variant, an accessor *`variant-id-field-id`* that returns the value of the field, and
- for each field of each variant, a mutator *`set-variant-id-field-id!`* that set the value of the field.

1.2 Deconstructing Data Structures: `type-case`

```
(type-case datatype-id expr
   branch ...)

branch = (variant-id (field-id ...) result-expr ...)
       | (else result-expr ...)
```

Branches on the datatype instance produced by *`expr`*, which must be an instance of *`datatype-id`* (previously defined with `define-type`) Each *`branch`* extracts the values of the fields, and binds them to *`field-id`* ....

If a branch is not specified for each variant, you may use an `else` branch to create a catch-all branch. An `else` branch must be the last branch in the sequence of branches. `type-case` signals a compile-time error if all variants are not covered and the `else` branch is missing. Similarly, `type-case` signals a compile-time error if an `else` branch is unreachable because a branch exists for all variants.

1.3 Testing Infrastructure

PLAI Scheme provides the following syntactic forms for testing.

```
(test result-expr expected-expr)
```

If `result-expr` and `expected-expr` evaluate to the same value, `result-value`, the test prints the following expression:

`(good result-expr result-value expected-value location)`.

If they do not evaluate to the same value, the test prints

`(bad result-expr result-value expected-value location)`.

If evaluating `result-expr` signals an error, the test prints

`(exception result-expr exception-message <no-expected-value> location)`

If evaluating `expected-expr` signals an error, the test prints

`(pred-exception result-expr exception-message <no-expected-value> location)`

If the printout begins with `good`, then it is printed to `(current-output-port)`; otherwise it is printed to `(current-error-port)`.

```
(test/pred result-expr pred?)
```

Similar to `test`, but instead of supplying an expected value, the predicate `pred?` is applied to `result-expr`.

If evaluating `pred?` signals an error, the test prints

`(pred-exception result-expr exception-message <no-expected-value> location)`

The syntax of `pred?` is considered `expected-value` for the purposes of test reporting.

```
error : procedure?
```

Like `scheme`'s `scheme:error`, but generates exceptions that are caught by `test/exn`.

```
(test/exn result-expr error-message)
```

This test succeeds if the expression evaluates to a call to `error`. Moreover, the error message contained in the exception must contain the string *`error-message`*. Note that `test/exn` only succeeds if the exception was explicitly raised by the user.

For example, the following test succeeds:

```
(test/exn (error "/: division by zero") "by zero")
```

The error message is `"/: division by zero"`, and `"by zero"` is a substring of the error message. However, the following test fails:

```
(test/exn (/ 25 0) "by zero")
```

Although the expression raises an exception and the error string contains `"by zero"`, since the error was not explicitly raised by user-written code, the test fails.

The evaluation of *`error-message`* is considered *`expected-value`* for the purposes of test reporting.

```
(test/regexp result-expr error-message-regexp)
```

This test is similar to `test/exn`,but the error message is matched against a regular expression instead.

The evaluation of *`error-message-regexp`* is considered *`expected-value`* for the purposes of test reporting.

1.3.1 Test Flags

```
(abridged-test-output [abridge?]) → void?
  abridge? : boolean? = false
```

When this flag is set to `true`, the test forms never prints *`result-expr`* or *`location`*.

```
(plai-catch-test-exn [catch?]) → void?
  catch? : boolean? = true
```

When this flag is set to `true`, exceptions from tests will be caught. By default, exceptions are caught.

```
(halt-on-errors [halt?]) → void?
  halt? : boolean? = true
```

This flag determines whether the program immediately halts when a test fails. By default, programs do not halt on failures.

```
(print-only-errors [print?]) → void?
  print? : boolean? = true
```

When this flag is set to `true`, only tests that fail will be printed. By default, the results of all tests are printed.

```
(test-inexact-epsilon epsilon) → void?
  epsilon : number?
```

When testing inexact values for equality, `test` permits them to differ by *`epsilon`*. The default value of *`epsilon`* is `0.01`.

```
(plai-ignore-exn-strings ignore?) → void?
  ignore? : boolean?
```

If this flag is set to `true`, when testing for exceptions with `test/exn` and `test/regexp`, the message of the exception is ignored. By default, `test/exn` and `test/regexp` only succeed when the message of the exception matches the supplied string or regular expression.

```
plai-all-test-results
```

This variable is the list of all tests that have been run so far, with the most recent test at the head.

2 GC Collector Scheme

```
#lang plai/collector
```

GC Collector Scheme is based on PLAI Scheme. It provides additional procedures and syntax for writing garbage collectors.

2.1 Garbage Collector Interface

The GC Collector Scheme language provides the following functions that provide access to the heap and root set:

```
(heap-size) → exact-nonnegative-integer?
```

Returns the size of the heap. The size of the heap is specified by the mutator that uses the garbage collector. See `allocator-setup` for more information.

```
(location? v) → boolean?
  v : any/c
```

Determines if *v* is an integer between `0` and `(- (heap-size) 1)` inclusive.

```
(root? v) → boolean?
  v : any/c
```

Determines if *v* is a root.

```
(heap-value? v) → boolean?
  v : any/c
```

A value that may be stored on the heap. Roughly corresponds to the contract `(or/c boolean? number? procedure? symbol? empty?)`.

```
(heap-set! loc val) → void?
  loc : location?
  val : heap-value?
```

Sets the value at *loc* to *val*.

```
(heap-ref loc) → heap-value?
  loc : location?
```

Returns the value at *loc*.

```
(get-root-set id ...)
```

Returns the current roots as a list. Local roots are created for the identifiers *id* as well.

```
(read-root root) → location?
  root : root?
```

Returns the location of *root*.

```
(set-root! root loc) → void?
  root : root?
  loc : location?
```

Updates the root to reference the given location.

```
(procedure-roots proc) → (listof root?)
  proc : procedure?
```

Given a closure stored on the heap, returns a list of the roots reachable from the closure's environment. If *proc* is not reachable, the empty list is returned.

```
(with-heap heap-expr body-expr ...)

  heap-expr : (vectorof heap-value?)
```

Evaluates each of the *body-expr*s in a context where the value of *heap-expr* is used as the heap. Useful in tests:

```
(test (with-heap (make-vector 20)
        (init-allocator)
        (gc:deref (gc:alloc-flat 2)))
      2)
```

```
(with-roots roots-expr expr1 expr2 ...)

  roots-expr : (listof location?)
```

Evaluates each of *expr1* and the *expr2*s in in a context with the result of *roots-expr* as additional roots.

This function is intended to be used in test suites for collectors. Since your test suites are not running in the

```
#lang plai/mutator
```

language, `get-root-set` returns a list consisting only of the roots it created, not all of the other roots it normally would return. Use this function to note specific locations as roots and set up better tests for your GC.

```
(test (with-heap (make-vector 4)
                 (define f1 (gc:alloc-flat 1))
                 (define c1 (gc:cons f1 f1))
                 (with-roots (list c1)
                             (gc:deref
                              (gc:first
                               (gc:cons f1 f1)))))
      1)
```

2.2 Garbage Collector Exports

A garbage collector must define the following functions:

```
(init-allocator) → void?
```

`init-allocator` is called before all other procedures by a mutator. Place any requisite initialization code here.

```
(gc:deref loc) → heap-value?
  loc : location?
```

Given the location of a flat Scheme value, this procedure should return that value. If the location does not hold a flat value, this function should signal an error.

```
(gc:alloc-flat val) → location?
  val : heap-value?
```

This procedure should allocate a flat Scheme value (number, symbol, boolean, closure or empty list) on the heap, returning its location (a number). The value should occupy a single heap cell, though you may use additional space to store a tag, etc. You are also welcome to pre-allocate common constants (e.g., the empty list). This procedure may need to perform a garbage-collection. If there is still insufficient space, it should signal an error.

Note that closures are flat values. The environment of a closure is internally managed, but contains references to values on the heap. Therefore, during garbage collection, the environment of reachable closures must be updated. The language exposes the environment via the `procedure-roots` function.

```
(gc:cons first rest) → location?
  first : location?
  rest : location?
```

Given the location of the `first` and `rest` values, this procedure must allocate a cons cell on the heap. If there is insufficient space to allocate the cons cell, it should signal an error.

```
(gc:first cons-cell) → location?
  cons-cell : location?
```

If the given location refers to a cons cell, this should return the first field. Otherwise, it should signal an error.

```
(gc:rest cons-cell) → location?
  cons-cell : location?
```

If the given location refers to a cons cell, this should return the rest field. Otherwise, it should signal an error.

```
(gc:set-first! cons-cell first-value) → void?
  cons-cell : location?
  first-value : location?
```

If `cons-cell` refers to a cons cell, set the head of the cons cell to `first-value`. Otherwise, signal an error.

```
(gc:set-rest! cons-cell rest-value) → void?
  cons-cell : location?
  rest-value : location?
```

If `cons-cell` refers to a cons cell, set the tail of the cons cell to `rest-value`. Otherwise, signal an error.

```
(gc:cons? loc) → boolean?
  loc : location?
```

Returns `true` if `loc` refers to a cons cell. This function should never signal an error.

```
(gc:flat? loc) → boolean?
  loc : location?
```

Returns `true` if `loc` refers to a flat value. This function should never signal an error.

3 GC Mutator Scheme

```
#lang plai/mutator
```

The GC Mutator Scheme language is used to test garbage collectors written with the §2 "GC Collector Scheme" language. Since collectors support a subset of Scheme's values, the GC Mutator Scheme language supports a subset of procedures and syntax. In addition, many procedures that can be written in the mutator are omitted as they make good test cases. Therefore, the mutator language provides only primitive procedures, such as `+`, `cons`, etc.

3.1 Building Mutators

The first expression of a mutator must be:

```
(allocator-setup collector-module
                 heap-size)

heap-size = exact-nonnegative-integer
```

The *`collector-module`* form specifies the path to the garbage collector that the mutator should use. The collector must be written in the GC Collector Scheme language.

The rest of a mutator module is a sequence of definitions, expressions and test cases. The GC Mutator Scheme language transforms these definitions and statements to use the collector specified in `allocator-setup`. In particular, many of the primitive forms, such as `cons` map directly to procedures such as `gc:cons`, written in the collector.

3.2 Mutator API

The GC Mutator Scheme language supports the following procedures and syntactic forms:

```
if
```

Just like Racket's `if`.

```
and
```

Just like Racket's `and`.

```
or
```

Just like Racket's `or`.

`cond`

Just like Racket's `cond`.

`case`

Just like Racket's `case`.

`define-values`

Just like Racket's `define-values`.

`let`

Just like Racket's `let`.

`let-values`

Just like Racket's `let-values`.

`let*`

Just like Racket's `let*`.

`set!`

Just like Racket's `set!`.

`quote`

Just like Racket's `quote`.

`begin`

Just like Racket's `begin`.

```
(define (id arg-id ...) body-expression ...+)
```

Just like Racket's `define`, except restricted to the simpler form above.

```
(lambda (arg-id ...) body-expression ...+)
(λ (arg-id ...) body-expression ...+)
```

Just like Racket's `lambda` and λ, except restricted to the simpler form above.

```
error : procedure?
```

Just like Racket's `error`.

```
add1 : procedure?
```

Just like Racket's `add1`.

```
sub1 : procedure?
```

Just like Racket's `sub1`.

```
zero? : procedure?
```

Just like Racket's `zero?`.

```
+ : procedure?
```

Just like Racket's `+`.

```
- : procedure?
```

Just like Racket's `-`.

```
* : procedure?
```

Just like Racket's `*`.

`/ : procedure?`

Just like Racket's `/`.

`even? : procedure?`

Just like Racket's `even?`.

`odd? : procedure?`

Just like Racket's `odd?`.

`= : procedure?`

Just like Racket's `=`.

`< : procedure?`

Just like Racket's `<`.

`> : procedure?`

Just like Racket's `>`.

`<= : procedure?`

Just like Racket's `<=`.

`>= : procedure?`

Just like Racket's `>=`.

`symbol? : procedure?`

Just like Racket's `symbol?`.

`symbol=? : procedure?`

Just like Racket's `symbol=?`.

`number? : procedure?`

Just like Racket's `number?`.

`boolean? : procedure?`

Just like Racket's `boolean?`.

`empty? : procedure?`

Just like Racket's `empty?`.

`eq? : procedure?`

Just like Racket's `eq?`.

```
(cons hd tl) → cons?
  hd : any/c
  tl : any/c
```

Constructs a (mutable) pair.

```
(cons? v) → boolean?
  v : any/c
```

Returns `#t` when given a value created by `cons`, `#f` otherwise.

```
(first c) → any/c
  c : cons?
```

Extracts the first component of *c*.

```
(rest c) → any/c
  c : cons?
```

Extracts the rest component of *c*.

```
(set-first! c v) → void?
  c : cons?
  v : any/c
```

Sets the `first` of the cons cell *c*.

```
(set-rest! c v) → void?
  c : cons?
  v : any/c
```

Sets the `rest` of the cons cell *c*.

```
empty
```

The identifier `empty` is defined to invoke `(gc:alloc-flat '())` wherever it is used.

```
print-only-errors : procedure?
```

Behaves like PLAI's `print-only-errors`.

```
halt-on-errors : procedure?
```

Behaves like PLAI's `halt-on-errors`.

Other common procedures are left undefined as they can be defined in terms of the primitives and may be used to test collectors.

Additional procedures from `scheme` may be imported with:

```
(import-primitives id ...)
```

Imports the procedures *id* ... from `scheme`. Each procedure is transformed to correctly interface with the mutator. That is, its arguments are dereferenced from the mutator's heap and the result is allocated on the mutator's heap. The arguments and result must be `heap-value?`s, even if the imported procedure accepts or produces structured data.

For example, the GC Mutator Scheme language does not define `modulo`:

```
(import-primitives modulo)

(test/value=? (modulo 5 3) 2)
```

3.3 Testing Mutators

GC Mutator Scheme provides two forms for testing mutators:

```
(test/location=? mutator-expr1 mutator-expr2)
```

`test/location=?` succeeds if *mutator-expr1* and *mutator-expr2* reference the same location on the heap.

```
(test/value=? mutator-expr scheme-datum/quoted)
```

`test/value=?` succeeds if *mutator-expr* and *scheme-datum/expr* are structurally equal. *scheme-datum/quoted* is not allocated on the mutator's heap. Futhermore, it must either be a quoted value or a literal value.

```
(printf format mutator-expr ...)

format = literal-string
```

In GC Mutator Scheme, `printf` is a syntactic form and not a procedure. The format string, *format* is not allocated on the mutator's heap.

3.4 Generating Random Mutators

```
(require plai/random-mutator)
```

This PLAI library provides a facility for generating random mutators, in order to test your garbage collection implementation.

```
(save-random-mutator  file
                      collector-name
                     [#:heap-values heap-values
                      #:iterations iterations
                      #:program-size program-size
                      #:heap-size heap-size])     → void?
  file : path-string?
  collector-name : string?
  heap-values : (cons heap-value? (listof heap-value?))
              = (list 0 1 -1 'x 'y #f #t '())
  iterations : exact-positive-integer? = 200
  program-size : exact-positive-integer? = 10
  heap-size : exact-positive-integer? = 100
```

Creates a random mutator that uses the collector *collector-name* and saves it in *file*.

The mutator is created by first making a random graph whose nodes either have no outgoing edges, two outgoing edges, or some random number of outgoing edges and then picking a random path in the graph that ends at one of the nodes with no edges.

This graph and path are then turned into a PLAI program by creating a `let` expression that binds one variable per node in the graph. If the node has no outgoing edges, it is bound to a `heap-value?`. If the node has two outgoing edges, it is bound to a pair and the two edges are put into the first and rest fields. Otherwise, the node is represented as a procedure that accepts an integer index and returns the destination node of the corresponding edge.

Once the `let` expression has been created, the program creates a bunch of garbage and then traverses the graph, according to the randomly created path. If the result of the path is the expected heap value, the program does this again, up to *iterations* times. If the result of the path is not the expected heap value, the program terminates with an error.

The keyword arguments control some aspects of the generation of random mutators:

- Elements from the *heap-values* argument are used as the base values when creating nodes with no outgoing edges. See also `find-heap-values`.
- The *iterations* argument controls how many times the graph is created (and traversed).
- The *program-size* argument is a bound on how big the program it is; it limits the number of nodes, the maximum number of edges, and the length of the path in the graph.
- The *heap-size* argument controls the size of the heap in the generated mutator.

```
(find-heap-values input) → (listof heap-value?)
  input : (or/c path-string? input-port?)
```

Processes *input* looking for occurrences of `heap-value?`s in the source of the program and returns them. This makes a good start for the *heap-values* argument to `save-random-mutator`.

If *input* is a port, its contents are assumed to be a well-formed PLAI program. If *input* is a file, the contents of the file are used.

4 GC Collector Language, 2

```
#lang plai/gc2/collector
```

GC Collector Scheme is based on PLAI. It provides additional procedures and syntax for writing garbage collectors.

4.1 Garbage Collector Interface for GC2

The GC Collector Scheme language provides the following functions that provide access to the heap and root set:

```
(heap-size) → exact-nonnegative-integer?
```

Returns the size of the heap. The size of the heap is specified by the mutator that uses the garbage collector. See `allocator-setup` for more information.

```
(location? v) → boolean?
  v : any/c
```

Determines if *v* is an integer between `0` and `(- (heap-size) 1)` inclusive.

```
(root? v) → boolean?
  v : any/c
```

Determines if *v* is a root.

```
(heap-value? v) → boolean?
  v : any/c
```

A value that may be stored on the heap. Roughly corresponds to the contract `(or/c boolean? number? procedure? symbol? empty?)`.

```
(heap-set! loc val) → void?
  loc : location?
  val : heap-value?
```

Sets the value at *loc* to *val*.

```
(heap-ref loc) → heap-value?
  loc : location?
```

Returns the value at *loc*.

```
(get-root-set)
```

Returns the current `root?`s as a list. This returns valid roots only when invoked via the mutator language. Otherwise it returns only what has been set up with `with-roots`.

```
(read-root root) → location?
  root : root?
```

Returns the location of *root*.

```
(set-root! root loc) → void?
  root : root?
  loc : location?
```

Updates *root* to refer to *loc*.

```
(make-root name get set) → root?
  name : symbol?
  get : (-> location?)
  set : (-> location? void?)
```

Creates a new root. When `read-root` is called, it invokes *get* and when `set-root!` is called, it invokes *set*.

For example, this creates a root that uses the local variable `x` to hold its location:

```
(let ([x 1])
  (make-root 'x
             (λ () x)
             (λ (new-x) (set! x new-x))))
```

```
(with-heap heap-expr body-expr ...)

  heap-expr : (vectorof heap-value?)
```

Evaluates each of the *body-expr*s in a context where the value of *heap-expr* is used as the heap. Useful in tests:

```
(test (with-heap (make-vector 20)
        (init-allocator)
        (gc:deref (gc:alloc-flat 2)))
      2)
```

```
(with-roots (root-var ...) expr1 expr2 ...)

  roots-expr : (listof location?)
```

Evaluates each of *expr1* and the *expr2*s in in a context with additional roots, one for each of the *root-var*s. The get-root-set function returns these additional roots. Calling read-root on one of the newly created roots returns the value of the corresponding *root-var* and calling set-root! mutates the corresponding variable.

This form is intended to be used in test suites for collectors. Since your test suites are not running in the

```
#lang plai/gc2/mutator
```

language, get-root-set returns a list consisting only of the roots it created, not all of the other roots it normally would return. Use with-roots to note specific locations as roots and set up better tests for your GC.

```
(test (with-heap (make-vector 4)
                 (define f1 (gc:alloc-flat 1))
                 (define r1 (make-root 'f1
                                       (λ () f1)
                                       (λ (v) (set! f1 v))))
                 (define c1 (gc:cons r1 r1))
                 (with-roots (c1)
                             (gc:deref
                              (gc:first
                               (gc:cons r1 r1)))))
      1)
```

4.2 Garbage Collector Exports for GC2

A garbage collector must define the following functions:

```
(init-allocator) → void?
```

init-allocator is called before all other procedures by a mutator. Place any requisite initialization code here.

```
(gc:deref loc) → heap-value?
  loc : location?
```

Given the location of a flat value, this procedure should return that value. If the location does not hold a flat value, this function should signal an error.

```
(gc:alloc-flat val) → location?
  val : heap-value?
```

This procedure should allocate a flat value (number, symbol, boolean, closure or empty list) on the heap, returning its location (a number). The value should occupy a single heap cell, though you may use additional space to store a tag, etc. You are also welcome to pre-allocate common constants (e.g., the empty list). This procedure may need to perform a garbage-collection. If there is still insufficient space, it should signal an error.

Note that closures are flat values. The environment of a closure is internally managed, but contains references to values on the heap. Therefore, during garbage collection, the environment of reachable closures must be updated. The language exposes the environment via the `procedure-roots` function.

```
(gc:cons first rest) → location?
  first : root?
  rest : root?
```

Given two roots, one for the *`first`* and *`rest`* values, this procedure must allocate a cons cell on the heap. If there is insufficient space to allocate the cons cell, it should signal an error.

```
(gc:first cons-cell) → location?
  cons-cell : location?
```

If the given location refers to a cons cell, this should return the first field. Otherwise, it should signal an error.

```
(gc:rest cons-cell) → location?
  cons-cell : location?
```

If the given location refers to a cons cell, this should return the rest field. Otherwise, it should signal an error.

```
(gc:set-first! cons-cell first-value) → void?
  cons-cell : location?
  first-value : location?
```

If *`cons-cell`* refers to a cons cell, set the head of the cons cell to *`first-value`*. Otherwise, signal an error.

```
(gc:set-rest! cons-cell rest-value) → void?
  cons-cell : location?
  rest-value : location?
```

If *cons-cell* refers to a cons cell, set the tail of the cons cell to *rest-value*. Otherwise, signal an error.

```
(gc:cons? loc) → boolean?
  loc : location?
```

Returns `#true` if *loc* refers to a cons cell. This function should never signal an error.

```
(gc:flat? loc) → boolean?
  loc : location?
```

Returns `#true` if *loc* refers to a flat value. This function should never signal an error.

```
(gc:closure code-ptr free-vars) → location?
  code-ptr : heap-value?
  free-vars : (listof root?)
```

Allocates a closure with *code-ptr* and the free variables in the list *free-vars*.

```
(gc:closure-code-ptr loc) → heap-value?
  loc : location?
```

Given a location returned from an earlier allocation check to see if it is a closure; if not signal an error. if so, return the *code-ptr* for that closure.

```
(gc:closure-env-ref loc i) → location?
  loc : location?
  i : exact-nonnegative-integer?
```

Given a location returned from an earlier allocation, check to see if it is a closure; if not signal an error. Uf so, return the *i* th variable in the closure (counting from 0).

```
(gc:closure? loc) → boolean?
  loc : location?
```

Determine if a previously allocated location holds a closure. This function will be called only with locations previous returned from an allocating function or passed to `set-root!`. It should never signal an error.

5 GC Mutator Language, 2

```
#lang plai/gc2/mutator
```

The GC Mutator Scheme language is used to test garbage collectors written with the §2 "GC Collector Scheme" language. Since collectors support a subset of Racket's values, the GC Mutator Scheme language supports a subset of procedures and syntax. In addition, many procedures that can be written in the mutator are omitted as they make good test cases. Therefore, the mutator language provides only primitive procedures, such as `+`, `cons`, etc.

5.1 Building Mutators for GC2

The first expression of a mutator must be:

```
(allocator-setup collector-module
                 heap-size)

heap-size = exact-nonnegative-integer
```

collector-module specifies the path to the garbage collector that the mutator should use. The collector must be written in the GC Collector Scheme language.

The rest of a mutator module is a sequence of definitions, expressions and test cases. The GC Mutator Scheme language transforms these definitions and statements to use the collector specified in `allocator-setup`. In particular, many of the primitive forms, such as `cons` map directly to procedures such as `gc:cons`, written in the collector.

5.2 Mutator API for GC2

The GC Mutator Scheme language supports the following procedures and syntactic forms:

```
if
```

Just like Racket's `if`.

```
and
```

Just like Racket's `and`.

```
or
```

Just like Racket's `or`.

`cond`

Just like Racket's `cond`.

`case`

Just like Racket's `case`.

`define-values`

Just like Racket's `define-values`.

`let`

Just like Racket's `let`.

`let-values`

Just like Racket's `let-values`.

`let*`

Just like Racket's `let*`.

`set!`

Similar to Racket's `set!`. Unlike Racket's `set!`, this `set!` is syntactically allowed only in positions that discard its result, e.g., at the top-level or in a `begin` expression (although not as the last expression in a `begin`).

`quote`

Just like Racket's `quote`.

`begin`

Just like Racket's `begin`.

```
(define (id arg-id ...) body-expression ...+)
```

Just like Racket's `define`, except restricted to the simpler form above.

```
(lambda (arg-id ...) body-expression ...+)
(λ (arg-id ...) body-expression ...+)
```

Just like Racket's `lambda` and λ, except restricted to the simpler form above.

```
error : procedure?
```

Just like Racket's `error`.

```
add1 : procedure?
```

Just like Racket's `add1`.

```
sub1 : procedure?
```

Just like Racket's `sub1`.

```
zero? : procedure?
```

Just like Racket's `zero?`.

```
+ : procedure?
```

Just like Racket's `+`.

```
- : procedure?
```

Just like Racket's `-`.

```
* : procedure?
```

Just like Racket's `*`.

`/ : procedure?`

Just like Racket's `/`.

`even? : procedure?`

Just like Racket's `even?`.

`odd? : procedure?`

Just like Racket's `odd?`.

`= : procedure?`

Just like Racket's `=`.

`< : procedure?`

Just like Racket's `<`.

`> : procedure?`

Just like Racket's `>`.

`<= : procedure?`

Just like Racket's `<=`.

`>= : procedure?`

Just like Racket's `>=`.

`symbol? : procedure?`

Just like Racket's `symbol?`.

`symbol=? : procedure?`

Just like Racket's `symbol=?`.

`number? : procedure?`

Just like Racket's `number?`.

`boolean? : procedure?`

Just like Racket's `boolean?`.

`empty? : procedure?`

Just like Racket's `empty?`.

`eq? : procedure?`

Just like Racket's `eq?`.

```
(cons hd tl) → cons?
  hd : any/c
  tl : any/c
```

Constructs a (mutable) pair.

```
(cons? v) → boolean?
  v : any/c
```

Returns `#t` when given a value created by `cons`, `#f` otherwise.

```
(first c) → any/c
  c : cons?
```

Extracts the first component of *c*.

```
(rest c) → any/c
  c : cons?
```

Extracts the rest component of *c*.

```
(set-first! c v) → void
  c : cons?
  v : any/c
```

Sets the `first` of the cons cell *c*.

Calls to this function are allowed only in syntactic positions that would discard its result, e.g., at the top-level or inside a `begin` expression (but not in the last expression in a `begin`). Also, this function appear only in the function position of an application expression.

So, in order to pass around a version of this function, you must write something like this `(λ (c v) (begin (set-first! c v) 42))`, perhaps picking a different value than `42` as the result.

```
(set-rest! c v) → void
  c : cons?
  v : any/c
```

Sets the `rest` of the cons cell *c*, with the same syntactic restrictions as `set-first!`.

```
empty
```

The identifier `empty` is defined to invoke `(gc:alloc-flat '())` wherever it is used.

```
print-only-errors
```

Behaves like PLAI's `print-only-errors`.

```
halt-on-errors
```

Behaves like PLAI's `halt-on-errors`.

Other common procedures are left undefined as they can be defined in terms of the primitives and may be used to test collectors.

Additional procedures from `scheme` may be imported with:

```
(import-primitives id ...)
```

Imports the procedures `id ...` from `scheme`. Each procedure is transformed to correctly interface with the mutator. That is, its arguments are dereferenced from the mutator's heap and the result is allocated on the mutator's heap. The arguments and result must be `heap-value?`s, even if the imported procedure accepts or produces structured data.

For example, the GC Mutator Scheme language does not define `modulo`:

```
(import-primitives modulo)

(test/value=? (modulo 5 3) 2)
```

5.3 Testing Mutators with GC2

GC Mutator Scheme provides two forms for testing mutators:

```
(test/location=? mutator-expr1 mutator-expr2)
```

`test/location=?` succeeds if *mutator-expr1* and *mutator-expr2* reference the same location on the heap.

```
(test/value=? mutator-expr datum/quoted)
```

`test/value=?` succeeds if *mutator-expr* and *datum/expr* are structurally equal. *datum/quoted* is not allocated on the mutator's heap. Futhermore, it must either be a quoted value or a literal value.

```
(printf format mutator-expr ...)

format = literal-string
```

In GC Mutator Scheme, `printf` is a syntactic form and not a procedure. The format string, *format* is not allocated on the mutator's heap.

5.4 Generating Random Mutators for GC2

```
(require plai/gc2/random-mutator)
```

This PLAI library provides a facility for generating random mutators, in order to test your garbage collection implementation.

```
(save-random-mutator  file
                      collector-name
                     [#:heap-values heap-values
                      #:iterations iterations
                      #:program-size program-size
                      #:heap-size heap-size])      → void?
  file : path-string?
  collector-name : string?
  heap-values : (cons heap-value? (listof heap-value?))
              = (list 0 1 -1 'x 'y #f #t '())
```

```
iterations : exact-positive-integer? = 200
program-size : exact-positive-integer? = 10
heap-size : exact-positive-integer? = 100
```

Creates a random mutator that uses the collector *collector-name* and saves it in *file*.

The mutator is created by first making a random graph whose nodes either have no outgoing edges, two outgoing edges, or some random number of outgoing edges and then picking a random path in the graph that ends at one of the nodes with no edges.

This graph and path are then turned into a PLAI program by creating a `let` expression that binds one variable per node in the graph. If the node has no outgoing edges, it is bound to a `heap-value?`. If the node has two outgoing edges, it is bound to a pair and the two edges are put into the first and rest fields. Otherwise, the node is represented as a procedure that accepts an integer index and returns the destination node of the corresponding edge.

Once the `let` expression has been created, the program creates a bunch of garbage and then traverses the graph, according to the randomly created path. If the result of the path is the expected heap value, the program does this again, up to *iterations* times. If the result of the path is not the expected heap value, the program terminates with an error.

The keyword arguments control some aspects of the generation of random mutators:

- Elements from the *heap-values* argument are used as the base values when creating nodes with no outgoing edges. See also `find-heap-values`.
- The *iterations* argument controls how many times the graph is created (and traversed).
- The *program-size* argument is a bound on how big the program it is; it limits the number of nodes, the maximum number of edges, and the length of the path in the graph.
- The *heap-size* argument controls the size of the heap in the generated mutator.

```
(find-heap-values input) → (listof heap-value?)
  input : (or/c path-string? input-port?)
```

Processes *input* looking for occurrences of `heap-value?`s in the source of the program and returns them. This makes a good start for the *heap-values* argument to `save-random-mutator`.

If *input* is a port, its contents are assumed to be a well-formed PLAI program. If *input* is a file, the contents of the file are used.

6 Web Application Scheme

```
#lang plai/web
```

The Web Application Scheme language allows you to write server-side Web applications for the PLT Web Server.

For more information about writing Web applications, see: *Web Applications in Racket*.

When you click on the Run button in DrRacket, your Web application is launched in the Web server.

The application is available at *http://localhost:8000/servlets/standalone.rkt*.

The Web Application Scheme language will automatically load this URL in your Web browser.

You may use `no-web-browser` to prevent the browser from being launched and `static-files-path` to serve additional static files.

6.1 Web Application Exports

A Web application must define a procedure `start`:

```
(start initial-request) → response?
  initial-request : request?
```

The initial request to a Web application is serviced by this procedure.