# SchLint

*Debugging tools for MIT Scheme*

Dylan Sherry dsherry@mit.edu
Joe Henke jdhenke@mit.edu
Owen Derby oderby@mit.edu
March 31, 2013

## Introduction

With this project, we set out to expand the debugging facilities available for MIT Scheme. In particular, we wanted to detect as many bugs as possible before runtime, track executed code during runtime for later review, and determine where returned values originated from and what processes modified them. Such advanced debugging utilities provide powerful tools to the programmer, enabling faster iteration by catching bugs sooner and making it easier to find bugs/problems.

To this end, we propose **SchLint**, a simple tool for advanced debugging of MIT Scheme programs. The name is a combination of Scheme and "lint," reflecting one of it's primary purposes as a static analysis tool. With SchLint, we provide tools for all three debugging capabilities outlined above. Our purpose is not to replace any of the existing debugging features of Scheme. In particular, if there are bugs which are caught already easily detected (e.g. syntax errors), we do not address them. Rather, SchLint can be viewed as an augmentation to the standard debugging process in Scheme. It will warn you about possible problems with your code, and aid you in triaging errors you may encounter.

We continue by giving an overview of how SchLint is constructed. Then we proceed into the details of the two major sub-systems of SchLint: 1) the lint-like static analysis and program flow abstraction and 2) the provenance tracking system for scheme objects.

## System Overview

SchLint is implemented as a custom MIT Scheme interpreter running within standard MIT Scheme. Instead of starting from scratch, we built SchLint from an existing interpreter[1]. This interpreter provides a REPL environment where the evaluation step occurs in two distinct phases. First, the raw s-expression is analyzed and compiled into a combinator. A combinator is a function which takes one argument, an environment. A combinator may call other combinators. Second, the combinator is called with the current environment to complete the evaluation, and returns a value.

Within this guest interpreter framework, the static analysis is implemented by instrumenting the analyze procedure, to statically analyze the code as it's compiled. To properly detect bugs in the code, SchLint builds a special kind of graph, called a Control Flow Graph (CFG), to model the program code. Once this graph is constructed, bugs can be detected easily by searching for

[1] http://groups.csail.mit.edu/mac/users/gjs/6.945/psets/ps04/code/

certain properties of the graph. After construction of the CFG, the code can be executed[1]. During execution, the CFG is traversed, and the execution is tracked as a trace over a series of edges of the graph. This trace can be queried later, to check for errors or to see code coverage.

Provenance tracking was implemented by slightly changing what sort of object the evaluation passes around internally. Normally, the interpreter passes around values (or lists, or other primitive or user-defined datatypes) while evaluating an expression, and returns that after the evaluation ends. To allow for tracking of arbitrary provenance, we abstracted objects into **cells**. A cell represents a data object and a list of tags. Tags are used to record provenance. With the introduction of cells, everything passes around cells behind the scenes of the interpreter, and the REPL simply extracts the value from the result of an evaluation before returning to the user.

# Lint and Scheme

The term “lint” originates from a program out of Bell Labs for statically checking for suspect code in C programs[2]. Nowadays, programs exist which implement lint-like behaviour for most almost every programming language, although none exist for MIT Scheme. With static analysis of supplied code, our linter detects

1. Unused procedures: procedures which are defined but never evaluated.
2. Unreachable procedures: procedures which are defined but can never be evaluated because of where or how they are defined.
3. Undefined procedures: procedures for which there exist one or more calls, but are not defined within the environment of those calls.

To perform this static analysis, we represent the program as a Control Flow Graph (CFG). A CFG attempts to capture all the possible valid execution traces through a program. By capturing what’s allowed, it becomes easy to check if a proposed trace is not allowed. A CFG captures procedure definitions and call sites, which is sufficient to model flow of execution through a program. Using this CFG, we are able to statically detect common bugs in a program and then track how the program is executed at runtime.

## *Graph Abstraction*

To start, we created a simple directed graph library to support building a CFG while abstracting out the details of what constitutes a graph. A graph is simply a collection of node objects and edge objects. Each node object corresponds to a node in the graph, and can have an arbitrary data object associated with it. Each edge object represents a directed edge in the graph, originating at a single node and ending at a single node. Each edge object has a reference to a source node object and destination node object and can have an arbitrary data object associated with it. Node objects maintain a list of all incoming edge objects (edge objects which have that node as a destination) and a list of all outgoing edge objects (edge objects which have that node as a source).

This graph library provides a minimalistic API for creating and manipulating these objects.

---

[1] Because this is implemented as an interpreter, the analysis and execution is actually interleaved. As illustrated by the examples later on, the conceived use case is defining all necessary procedures and structures first, then checking for bugs, then executing.

[2] https://en.wikipedia.org/wiki/Lint_%28software%29

1. A new graph object can be created, and edge and node objects can be created and added to it.
2. The list of all node and edge objects (separately) can be retrieved from the graph, and the existence of a node or edge object can be checked.
3. Incoming and outgoing edge objects can be retrieved given a node object. Conversely, given an edge object, the source and destination node objects can be retrieved.
4. The data associated with edge and node objects can be retrieved and modified.
5. Edge and node objects can be removed from the graph.

### *Control Flow Graph*

We use our graph library to represent the CFG. Nodes in the CFG correspond to procedures in the program under analysis. A CFG encapsulates two unique relations, which are represented as edges:

1. Definitions: "*A* defines *B*" iff the definition of procedure *B* is within the definition of procedure *A*.
2. Applications: "*A* calls *B*" iff there is an application of procedure *B* within the definition of procedure *A*.

Every procedure has a single incoming definition edge. Top-level procedures aren't defined by another procedure, so a fake node, called **root**, is added to be the "procedure" which defines all top-level procedures. A call edge ending at procedure *A* exists for every application of *A*. This means there may be multiple call edges from one node to another (when *B* invokes *A* multiple times) and (multiple) call edges from a node to itself (*A* recursively calls itself).

To construct the CFG, the code is analyzed in a depth-first manner. As the interpreter analyzes the code, it keeps track of the scope in which new definitions will occur by passing around a reference to the current "parent" node, which is initially the root node. As it encounters "define" statements which define a procedure, a new node is added to the graph and a definition edge is added from the current parent node to the new node. The definition expression is then analyzed, with the new node as the parent.

As applications of procedures are encountered, a new call edge should be added from the current parent node to the invoked procedure. Before the edge can be added, however, the node corresponding to the invoked procedure needs to be found in the graph. Given the symbol identifying the procedure and the parent node, the task is to find the corresponding node, if any exists, by searching upwards from the parent. We only need consider the subset of nodes in the graph which are callable from an object in the parent's scope (which is where the application is occurring). Figure 1 gives an illustration of this distinction. We have 3 possible results from our search

1. The node exists and is callable from the current scope.
2. The invoked procedure is defined in the default REPL environment (primitives, SchLint reserved procedures and other procedures defined before Schlint REPL is started)
3. The node does does not exist or is not callable from the current scope.

In the first and second cases, a call edge is added from the parent node to the found node. In the third case, a special "undefined" node is created (or used if already existing), which indicates that such a procedure has been invoked, but not defined, and a call edge is added to it. This allows us to properly track calls to procedures which may be defined later on. When the

procedure is defined in the future, the defined node simply assumes all call edges to this temporary one.

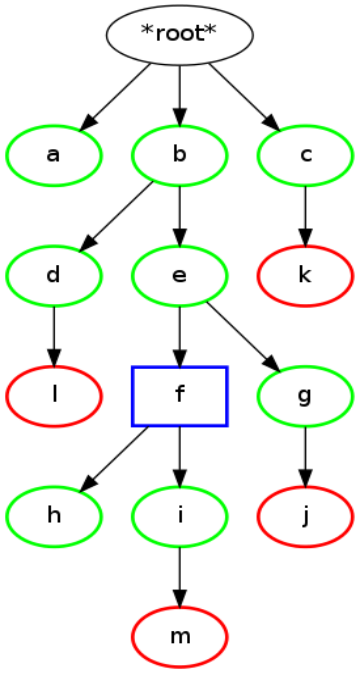



**Figure 1:** Illustration of callable (green) and un-callable (red) nodes in the CFG from the scope of f.

## *Detecting Problems*

We can use the CFG to statically detect three common bugs in code:

1. Unused Definitions: A procedure which is defined in a non-global scope but which is never called. We can detect this from the CFG because it will be a node or sub-tree which has a define edge, but no call edges to it.
2. Undefined Reference: An invocation of a procedure which doesn't exist. We can detect this by searching for any "undefined"-type nodes.
3. Dead Code: Procedures which are defined but all references to them are lost, such as defining a new procedure with the same name. We can detect dead code by searching the graph for sub-graphs which we can not reach from the root, by following either call or define edges.

## *Tracking Executions*

Aside from statically catching bugs, we use the CFG to track program executions during runtime. We treat executions as a property of call edges, and for each call edge we add an execution event/entry every time that call is actually made at runtime. Each execution event records the arguments for that call and the value it returned. The effect of this is that for any call edge in the CFG, at any point in the runtime, we can stop and see exactly how many times it was actually invoked, with which arguments, and what was returned. Further, we can see which call edges were actually used/traversed and which remained untouched. Finally, we can order the execution events by the time they occurred, and use that to form a very detailed stack-trace

for the entire program. This can be very useful to see how and what a program executed, even if it doesn't produce any errors (which is usually the only time you get a stack trace).

## *Examples*

Here we provide some brief demonstrations of our static analysis system detecting some common problems. For each listing, we run the static analysis by loading in SchLint (via the command (load "load")) and initializing it (via the command (init)), then importing the listing, and finally calling the analysis procedure (check-cfg). For each example, we provide the listing code, the output from the analysis, and the CFG generated (for reference).

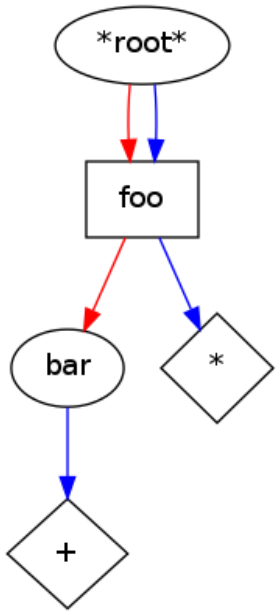
input:

```
(define (foo x)
  (define (bar y)
    (+ y 1))
  (* x x))
(foo 3)
```

output:

```
1 ]=> (check-cfg)
"function bar is unused"
;Value: #f
```

generated CFG:




**Listing 1**: Detecting unused code

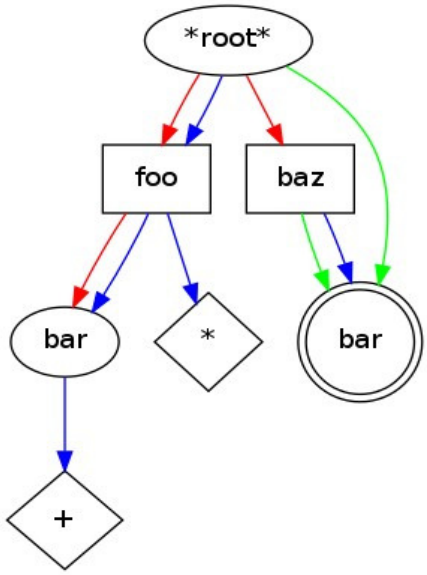
input:

```
(define (foo x)
  (define (bar y)
    (+ y 1))
  (bar (* x x)))
(define (baz x)
  (bar x))
(foo 3)
```

output:

```
1 ]=> (check-cfg)
"baz calls undefined function bar"
;Value: #f
```

generated CFG:




**Listing 2**: Detecting undefined references.

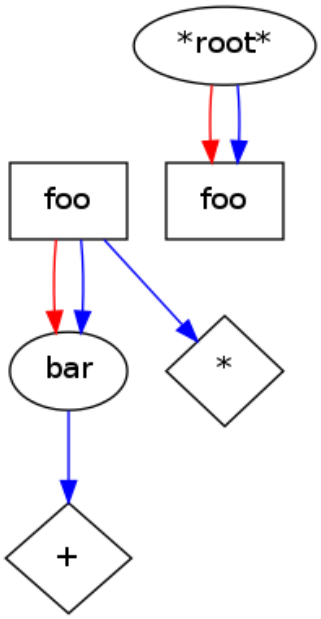
input:

```
(define (foo x)
  (define (bar y)
    (+ y 1))
  (bar (* x x)))
(define (foo x) 2)
(foo 3)
```

output:

```
1 ]=> (check-cfg)
"function foo is unreachable"
"function bar is unreachable"
;Value: #f
```

generated CFG:




**Listing 3**: Detecting dead code.

# Provenance Tracking

## *The Idea*

Provenance tracking refers to seeing what pieces of data contributed to the formation of other pieces of data. From a debugging standpoint, it can immediately answer I have a problem with my output. Which of these inputs need I concern myself with?

More specifically, any object can be given any number of tags. To propagate these tags, objects created from the output of primitive procedures are assigned the union of the tags of the arguments to the primitive procedure. The only other way to propagate tags is through tagging functions themselves. If a function has tags, the object returned from each function call also has that function's tags.

## *The Interface*

We've provided the ability in our guest interpreter to tag objects and see to which objects they then contribute.

### *Creating Tags*

Tags can be attached to any object in any scope.

```
(define a 2)

;;; To user defined variables
(add-tag a 'apples)
```

```
;;; Within functions
(define (foo x)
  (define b 3)
  (add-tag b 'bananas)
  (+ x b))

;;; To primitive functions
(add-tag + 'bar)

;;; Even to constant objects
;;; Note: These are transient objects
(add-tag 1 'baz)
; Value: 1
```

### *Checking Tags*

Now we can check to see how tags have been modified for a particular variable or object after some computation.

```
;;; Check variables
(get-tags a)
; Value: (apples)

(define c a)

;;; Definitions propagate tags
(get-tags c)
; Value: (apples)

;;; Function calls propagate tags
(get-tags (foo a))
; Value: (apples inner bar)

;;; Assigning to transient objects work
;;; but are exactly that - transient
(get-tags (add-tag 1 'one-time))
; Value: (one-time)

(get-tags 1)
; Value: ()
```

## *Implementation*

In summary, we built off of PS4, abstracted the notion of an object to also hold provenance information and ensured the provenance information propagated through newly created objects.

### *PS4 Recap*

We built off the guest interpreter provided in PSET 4 - Compiling to Combinators.

This provided a REPL environment where the evaluation step occurs in two distinct phases. First, the raw expression is compiled into a combinator using analyze. A combinator is a function which takes one argument, an environment. A combinator may call other combinators. Second, the combinator is called with the provided environment, returning a value in scheme. Thus, we have defined our evaluation procedure.

### *Modifying eval's meaning*

The fundamental change we've implemented is to the eval procedure paradigm. It is still coded to be the same thing, however it is now defined semantically to return a cell. A cell represents an abstraction of any kind of scheme object. We've defined a cell to be the following structure.

```
(define-structure cell value tags)
```

So a cell holds two datatypes, but could easily be extended to hold more.
The first component is value. This is an MIT Scheme value and these are the primitives upon which the user of this guest interpreter can build.

The second component is tags. A tag can be any object. In practice they are typically symbols as they are distinguished by eq?. Tags are either added explicitly to an object via add-tag or when the object is created through a primitive procedure, in which case it's tags are set to the union of the arguments to the primitive procedure.

This required removing the assumption that the output value of eval was an MIT Scheme object from the code. This mostly affected all the analyze procedures and the mechanism for variable and environment management. The print step in the REPL environment was also modified to only print the value of the cell that was ultimately returned from eval.

### *Tail Recursion*

Special care was taken to avoid breaking tail recursion within the guest interpreter. If within definitions of analyze, a combinator was called and then tags were added to that result, it would break tail recursion. To avoid this, it is determined if the current function call is in a tail recursive position. If so, it first adds the tags which are to be applied to a queue of pending tags, then makes an appropriate tail recursive tail without adding tags. When a non-recursive call is found, it then adds the queued tags to the result. Top level calls are guaranteed to not be in tail recursive positions and so will inherit the pending tags upon the final return.

## *Design Decisions*

### *Functions*

What tags should be propagated to newly defined compound procedures? We decided none. The only real alternative would be to use the union of all variable accesses made within the function to outside the environment frame created in the function, but those accesses don't occur when the function is defined. Furthermore, any output from the function which was affected by these external variable access will still reflect these sources in their tags.

Because primitive procedure definitions were not interpreted by the guest procedure, it is treated as a black box and it is assumed all arguments contribute to the output. In fact, this may not necessarily be true though, but we could not find a general solution to this.

What if a user explicitly tags a variable which is itself a function? We decided that these tags should in fact be propagated with the output of any successive function calls, to answer the question which objects were affected by this function?

#### *Conditionals*

Should the tags of the predicate of a conditional be propagated to the data created in the consequence or alternative? We decided no. This seemed too indirect a connection to include with provenance information, as we intended to represent who contributed to the values of a variables creation, not necessarily just the fact that it was created.

#### *Doubly Linked Tags*

Something which was not implemented but we feel would be a great addition to this system is to be able to answer, to whom did I contribute? Currently, any object's tags can be retrieved, but once an object is tagged, one can't easily see what objects now have those tags. It is unclear what such functionality would return. Perhaps any variables bound to objects who has those tags and are defined in the current scope, but it should be investigated further.

### *Notes on Abstractions*

#### *Abstracting Values*

We found that even though all references to values are now replaced with cells, the only places from the original ps4 code where a cell must be unpacked down to its value was when applying a primitive procedure, conditional predicates or generic dispatch predicates for analyze. Given this very limited rep-exposure of a value/cell, we feel that choosing this abstraction was a good choice and perhaps even further clarifies the distinction between the interpreter objects, MIT Scheme Objects used to run the interpreter itself and the primitive MIT scheme objects which the interpreter may access.

#### *Abstracting Environments*

In the original ps4 code, useful abstractions were defined to abstract away the internals of how an environment was represented, but were not actually used. We replaced all instances of environment rep exposure by using the function calls specifically design to operate on environments. This way, if the structure or internals of an environment must ever be changed, only these operation need be changed.

## Conclusion

Debugging in Scheme can be a difficult process, especially if the problematic bug is subtle or involves multiple components. We set out to improve this experience with MIT Scheme. Starting from the ps4 code as a baseline, we constructed SchLint, a collection of useful debugging tools for MIT Scheme.

1. By combining a simple graph API with the compiler interpreter, we were able to abstract the flow of a program into a control flow graph. With this control flow graph, we are able to statically detect bugs involving undefined procedures, uncalled procedures, and dead procedures.
2. Further, we use the generated control flow graph to track program execution at runtime, to provide increased visibility and verbosity when examining how a program ran and why it returned the value it did.
3. Finally, to enhance the ability to determine what inputs affected which outputs, we implemented provenance tracking, which allows inputs and functions to be tagged, so you can quickly determine the entire provenance of an output simply by examining its tags.

A copy of this document and the code (included in the appendix) can be found online at https://github.com/jdhenke/introspect

```
;;; graph.scm - our directed graph sub-system used to store data

;;; JDH - a main goal is to separate external interface
;;;       from internal implementation details.
;;;       Therefore: objects created with this API should only be operated on
;;;                  by functions within this API
;;; Basically, this is an ADT

;;; Internal Representation of objects

(define-structure graph nodes)
(define-structure node incoming-edges outgoing-edges data)
(define-structure edge src-node dest-node data)

;;; is this separation useful?
(define (graph-add-node graph node)
  (set-graph-nodes! graph (append (graph-nodes graph) `(,node))))

(define (node-add-incoming-edge node edge)
  ; TODO: check for double adding edge? these should be sets. implement sets?
  (set-node-incoming-edges! node (append (node-incoming-edges node) `(,edge))))

(define (node-add-outgoing-edge node edge)
  ; TODO: check for double adding edge? these should be sets. implement sets?
  (set-node-outgoing-edges! node (append (node-outgoing-edges node) `(,edge))))


;;; Organized by CRUD operations

;;; Create

;;; Returns a directed graph, which may be acted upon by this API
(define (create-graph)
  (make-graph '()))

;;; Adds a new node to the graph
;;; Returns the newly created node
;;; Can optionally specify an arbitrary data object
;;; to be attached to this node
(define (add-node graph #!optional node-data)
  (let ((node (make-node '() '() node-data)))
    (graph-add-node graph node)
    node))

;;; Adds a new edge to the graph; returns constructed edge
;;; The graph represents a directed edge from `from-node` to `to-node`
;;; Can optionally specify an arbitrary data object
;;; to be attached to this edge
;;; NOTE: can create multiple edges between the same two nodes
(define (add-edge from-node to-node #!optional edge-data)
  (let ((edge (make-edge from-node to-node edge-data)))
    (node-add-outgoing-edge from-node edge)
    (node-add-incoming-edge to-node edge)
    edge))

;;; Read

;;; Returns true if a node exists in graph
(define (exists-node? g n)
  (let loop ((nodes (get-nodes g)))
    (cond ((null? nodes) #f)
          ((eq? n (car nodes)) #t)
          (else (loop (cdr nodes))))))

;;; Returns true if an edge exists from `from-node` to `to-node`
(define (exists-edge? from-node to-node)
  (there-exists? (node-outgoing-edges from-node)
            (lambda (n)
              (eq? (edge-dest-node n) to-node))))

;;; Gets the data object attached to `node`
;;; returns `#f` if nothing has been attached
```

```
(define (get-node-data node)
  (node-data node))

;;; Gets the data object attached to `edge`
;;; returns `#f` if nothing has been attached
(define (get-edge-data edge)
  (edge-data edge))

;;; Returns list of incoming edges to node
(define (get-incoming-edges node)
  (node-incoming-edges node))

;;; Returns list of outgoing edges from node
(define (get-outgoing-edges node)
  (node-outgoing-edges node))

;;; Returns all the nodes in the graph
(define (get-nodes graph)
  (graph-nodes graph))

;;; Returns all the edges in the graph
(define (get-edges graph)
  (append-map get-outgoing-edges (get-nodes graph)))

;;; Update

;;; set-node-data! and set-edge-data! are automagically created by scheme!

;;; Delete

;;; Removes node from graph
;;; Also removes any edges to or from it
(define (remove-node! graph node)
  (set-graph-nodes! graph
                    (delq node (graph-nodes graph)))
  (for-each (lambda (e)
              (remove-edge! e))
            (node-outgoing-edges node))

  (for-each (lambda (e)
              (remove-edge! e))
            (node-incoming-edges node)))

;;; Removes edge from graph
;;; Leaves the nodes
(define (remove-edge! edge)
  (let ((src-node (edge-src-node edge))
        (dest-node (edge-dest-node edge)))
    (set-node-outgoing-edges! src-node (delq edge (node-outgoing-edges src-node)))
    (set-node-incoming-edges! dest-node (delq edge (node-incoming-edges dest-node)))))

;;; Prints all nodes in a graph
(define (pp-graph graph)
  (define (pp-list l)
    (if (not (null? l))
        (begin (pp (car l)) (pp-list (cdr l)))))
  (pp-list (get-nodes graph))
  (pp-list (get-edges graph)))
```

```
(load "graph.scm")
(load "utility.scm")

;;; Basic node/edge creation/deletion test for graph API
;;; Run using `(load "graph-tests.scm")`

(define g (create-graph))
(define n1 (add-node g))
(define n2 (add-node g))

(assert (exists-node? g n1) "adding node 1")

(assert (exists-node? g n2) "adding node 2")
(assert (not (eq? n1 n2)) "nodes are distinct")

(assert (not (exists-edge? n1 n2)) "adding edge 1")
(assert (not (exists-edge? n2 n1)) "adding edge 2")

(define e1 (add-edge n1 n2))
(assert (exists-edge? n1 n2) "adding edge 1 in correct direction")
(assert (not (exists-edge? n2 n1)) "adding edge 1 in correct direction (2)")

(add-edge n2 n1)
(assert (exists-edge? n1 n2) "adding edge 2 maintains old edge")
(assert (exists-edge? n2 n1) "adding edge 2 creates new edge" )

(remove-edge! e1)
(assert (not (exists-edge? n1 n2)) "remove edge 1")
(assert (exists-edge? n2 n1) "removing edge 1 keeps edge 2")


(remove-node! g n1)
(assert (= 1 (length (graph-nodes g))) "removes at least *a* node")
(assert (eq? n2 (car (graph-nodes g))) "node 2 remains")
(assert (not (eq? n1 (car (graph-nodes g)))) "node 1 does not remain")
(assert (exists-node? g n2) "removing node 1 leaves node 2")
(assert (not (exists-node? g n1)) "removing node 1 works")
(assert (not (exists-edge? n1 n2)) "removing node 1 keeps node 2")
(assert (not (exists-edge? n2 n1)) "removing node 1 removes edge 2")

'passed
```

```
;;; ui.scm - our control flow graph (cfg for short) front-end

;;; Given our generic graph library, we need an interface to interact with the
;;; control flow graphs we generate for a given program. This should implement
;;; procedures to make it more natural to construct a control flow graph.

;;; Define some data types that will be used
;;; These will be used to:
;;;  - differentiate between eachother
;;;  - store additional data for function calls
;;;    - these are stored as a list of executions
(define-structure function-def)
(define-structure possible-function-def)
(define-structure function-call executions)
(define-structure execution inputs output)

;;; Constants
;; descriptor type indicator
(define *desc* '*desc*)
;; function types
(define *root* '*root*) ;uniqe type for the non-existant node used to unite all
                        ;global/source nodes into a peer-group
(define *global* 'global)
(define *normal* 'non-global)
(define *undefined* 'undefined)
(define *primitive* 'primitive)

;;; We associate a description list with each node in our cfg. These descriptors
;;; indicate which type the function is (global or not) and the name associated
;;; with it.
(define (cfg:make-descriptor type name)
  (list *desc* type name))
(define (cfg:node-type node)
  (assert (node? node) "need a node object")
  (let ((desc (get-node-data node)))
    (assert (cfg:descriptor? desc) "node data not proper description")
    (cadr desc)))
(define (cfg:node-name node)
  (assert (node? node) "need a node object")
  (let ((desc (get-node-data node)))
    (assert (cfg:descriptor? desc) "node data not proper description")
    (caddr desc)))
(define (cfg:descriptor? desc)
  (and (list? desc) (= (length desc) 3) (eq? (car desc) *desc*)))
(define (cfg:global-node? node)
  (eq? (cfg:node-type node) *global*))
(define (cfg:undefined-node? node)
  (eq? (cfg:node-type node) *undefined*))
(define (cfg:primitive-node? node)
  (eq? (cfg:node-type node) *primitive*))
(define (cfg:root-node? node)
  (eq? (cfg:node-type node) *root*))
(define (cfg? cfg)
  (and (pair? cfg) (graph? (cfg:get-graph cfg)) (node? (cfg:get-root cfg))))


;;; cfg (private) helper procedures

;;; given a node defining the local scope and a function name, find any node
;;; within the current scope called the given name. Possible returns multiple
;;; undefined nodes if they were possibly defined in different scopes but have
;;; the same name.
(define (cfg:find-defined-node p-node f)
  (define (filter-for-node nodes)
    (filter (lambda (n)
              (eq? (cfg:node-name n) f))
            nodes))
  (filter-for-node
   (cfg:get-nodes-by-edge p-node
                          (lambda (e)
                            (or (function-def? e)
                                (possible-function-def? e))))))
```

```
;;; add 'function' of given 'type', with reference to current scope defined by
;;; p-node in the given 'cfg'
(define (cfg:add-function cfg p-node type function)
  (assert (node? p-node) "need a node object")
  ;; helper function to merge several undefined nodes into one new (defined) node.
  (define (merge-nodes nodes new-node)
    (for-each
     (lambda (node)
       (for-each
        (lambda (e)
          (let ((etype (get-edge-data e)))
            (cond
             ;; remove old possible-function-def edges
             ((possible-function-def? etype) (remove-edge! e))
             ;; transfer call edges to new-node
             ((function-call? etype)
              (begin
                (add-edge (edge-src-node e) new-node etype)
                (remove-edge! e)))
             ;; no other edge types should exist
             (else (assert #f "shouldn't have function defs here!")))))
          (get-incoming-edges node))
        ;; remove old node
        (remove-node! (cfg:get-graph cfg) node)
        )
        nodes))

  (let ((nodes (cfg:find-defined-node p-node function))
        (new-node (add-node (cfg:get-graph cfg) (cfg:make-descriptor type function))))
    (cond
     ;; no node named `function defined in this scope, add new node
     ((null? nodes)
      new-node)
     ;; replace undefined node with definition
     ((cfg:undefined-node? (car nodes))
      (begin (merge-nodes nodes new-node) new-node))
     ;; otherwise, handle case of redefining previously defined node. Do
     ;; this by removing the define edge to the old definition and creating
     ;; a new node for the new definition.
     (else
      (begin (assert (= (length nodes) 1) "should only find at most 1 previous function definitio
n")
             ;;(pp `("replacing definition for " ,(cfg:node-name (car nodes))))
             (let ((def-edge (find (lambda (e)
                                     (and (function-def? (get-edge-data e))
                                          (eq? (edge-dest-node e) (car nodes))))
                                   (node-outgoing-edges p-node))))
               (assert (not (eq? def-edge #f)))
               (remove-edge! def-edge)
               new-node))))))

;;; add an undefined function 'function' to the given 'cfg'
(define (cfg:add-undefined-function cfg caller function)
  (let ((nodes (cfg:find-defined-node caller function)))
    (if (null? nodes)
        ;; If no placeholder node exists in this scope, create it
        (let ((node (add-node (cfg:get-graph cfg)
                              (cfg:make-descriptor *undefined* function))))
          ;; recursively add possible definition edges from every ancestor to new
          ;; node, to encode all the possible valid define edges if this undefined
          ;; node is ever defined in the future.
          (let lp ((parent caller))
            (add-edge parent node (make-possible-function-def))
            (if (cfg:root-node? parent)
                node
                (lp (cfg:defined-by parent)))))
        ;; return existing one
        (begin (assert (= (length nodes) 1)) (car nodes)))))
```

```
(define (cfg:add-primitive-function cfg function)
  (add-node (cfg:get-graph cfg) (cfg:make-descriptor *primitive* function)))

(define (cfg:get-graph cfg)
  (car cfg))
(define (cfg:get-root cfg)
  (cadr cfg))

(define (cfg:add-define-edge f sub-f)
  (assert (node? f) "need a node object")
  (assert (node? sub-f) "need a node object")
  (let*  ((edges (get-incoming-edges sub-f))
          (edge (find (lambda (e)
                        (function-def? (get-edge-data e)))
                      edges)))
    (assert (eq? edge #f) `(,sub-f " has already been defined"))

    (add-edge f sub-f (make-function-def))))

(define (cfg:add-call-edge caller callee)
  (assert (node? caller) "need a node object")
  (assert (node? callee) "need a node object")
  (add-edge caller callee (make-function-call '())))

;;; Given a function node, return the node for the function which defined it.
(define (cfg:defined-by func)
  (assert (node? func) "need a node object")
  (let* ((edges (get-incoming-edges func))
         (edge (find (lambda (e)
                       (function-def? (get-edge-data e)))
                     edges)))
    (assert (not (eq? edge #f)) "should have single incoming define edge")
    (edge-src-node edge)))

;;; Given a function node, return the node of every function it defines.
(define (cfg:get-nodes-by-edge func edge-pred)
  (assert (node? func) "need a node object")
  (let ((edges (get-outgoing-edges func)))
    (map edge-dest-node
         (filter (lambda (e) (edge-pred (get-edge-data e)))
                 edges))))

;;; Given a function node, return the node of every function it defines.
(define (cfg:get-defines func)
  (cfg:get-nodes-by-edge func function-def?))

;;; Given a function node, return the node of every function it defines.
(define (cfg:get-possible-defines func)
  (cfg:get-nodes-by-edge func possible-function-def?))

;;; Given a function node, find the nodes of all other functions within the same
;;; namespace.
(define (cfg:get-peers func)
  (assert (node? func) "need a node object")
  (let* ((parent (cfg:defined-by func))
         (peers (cfg:get-defines parent)))
    (filter (lambda (p) (not (eq? p func))) peers)))


;;; given a node defining the local scope and a function name, find
;;; appropriate node, recursively looking upward through scopes
(define (cfg:find-callable-node p-node f)
  (define (filter-for-node nodes)
    (find (lambda (n)
            (eq? (cfg:node-name n) f))
          nodes))
  (let lp ((peers (cfg:get-defines p-node))
           (parent p-node))
    (let ((func (filter-for-node peers)))
      (cond ((cfg:root-node? parent) func) ; If parent is root, return what we have
            ((not (eq? func #f)) func) ; if we found it
```

```
            (else (let ((parent-parent (cfg:defined-by parent)))
                    (lp (cfg:get-defines parent-parent) parent-parent)))))))

(define (cfg:find-primitive-function cfg f)
  (define (assigned-and-bound? symb)
    (and (symbol? symb)
         (eq? 'normal
              (environment-reference-type generic-evaluation-environment symb))))
  (define (filter-for-node nodes)
    (find (lambda (n)
            (eq? (cfg:node-name n) f))
          nodes))
  (let ((func (filter-for-node (filter (lambda (n) (cfg:primitive-node? n))
                                       (get-nodes (cfg:get-graph cfg))))))
    (cond ((not (eq? func #f)) func)
          ((assigned-and-bound? f) (cfg:add-primitive-function cfg f))
          (else #f))))

(define (cfg:add-execution edge exec)
  (assert (edge? edge))
  (assert (execution? exec))
  (let ((fc (get-edge-data edge)))
    (assert (function-call? fc))
    (set-function-call-executions!
     fc
     (append (function-call-executions fc) `(,exec)))))

;;; cfg public/functions. Everything you need to construct a proper cfg!!

;;; utilities
;;; Given cfg, return all nodes which satisfy the predicate pred.
(define (filter-nodes cfg pred)
  (filter pred (graph-nodes (cfg:get-graph cfg))))

;; Is node in nodes? return true if so, false otherwise.
(define (in? node nodes)
  ;;(pp `(,node 'in ,nodes))
  (not (eq? (memq node nodes) #f)))

;;; Create a cfg object and return it.
(define (create-cfg)
  (let ((cfg (create-graph)))
    ;;; We employ the use of a dummy root node to make our cfg a well-defined
    ;;; tree.
    (let ((root (add-node cfg (cfg:make-descriptor *root* *root*))))
      (list cfg root))))

;;; Functions are declared in one of two types of scopes: At the top level
;;; (global scope), or within the scope of another function. We need to handle
;;; these two cases separately.

;;; given a cfg object and a function name defined at the top level, add the
;;; function to the cfg as a source node. Return the newly created node.
(define (define-global-func cfg f)
  (let* ((root (cfg:get-root cfg))
         (func (cfg:add-function cfg root *global* f)))
    ;;(pp `("add definition from " ,root " to " ,func))
    (cfg:add-define-edge root func)
    func))

;;; Given a cfg object, the name of a parent function and the name of the
;;; sub-function which is defined within the scope of hte parent, add the new
;;; sub-function 'sub-f' to the 'cfg' and add a directed edge from 'parent' to
;;; 'sub-f'. Return the node of the sub-function.
(define (define-sub-function cfg parent sub-f)
  (let ((sub-f (cfg:add-function cfg parent *normal* sub-f)))
    ;;(pp `("add definition from " ,parent " to " ,sub-f))
    (cfg:add-define-edge parent sub-f)
    sub-f))

;;; We also care about functions which call other functions, so we add a
;;; different type of edge for invocations than we do for defines.
```

```
;;; Add a function call to a cfg
;;; Given the cfg, calling function node and the called function name, add a
;;; directed edge from caller to callee. Will create a new "undefined" node as a
;;; placeholder until it is defined for any currently undefined nodes
(define (add-function-call cfg caller callee)
  (let ((ce-f (cfg:find-callable-node caller callee)))
    (if (eq? ce-f #f) ; callee not defined, add place holder
        (begin (set! ce-f (cfg:find-primitive-function cfg callee))
               (if (eq? ce-f #f) ; callee not defined, add place holder
                   (set! ce-f (cfg:add-undefined-function cfg caller callee)))))
    ;;(pp `("add function call from " ,caller " to " ,ce-f))
    (cfg:add-call-edge caller ce-f)))

;;; Handle special case of calling global function.
(define (add-global-call cfg callee)
  (add-function-call cfg (cfg:get-root cfg) callee))

;;; Add an execution of the given edge with provided inputs and outputs
(define (add-execution edge inputs outputs)
  (let ((exec (make-execution inputs outputs)))
    (cfg:add-execution edge exec)
    exec))

;;; Return all executions of a given edge
(define (get-executions edge)
  (function-call-executions (get-edge-data edge)))

;;; Print a cfg graph
(define (pp-cfg graph)
  (pp-graph (cfg:get-graph graph)))

(define (node-string node)
  (let ((ns (obj->string node)))
    (string-append "nd" (substring ns 7 (- (string-length ns) 1)))))

(define (cfg->dot cfg #!optional file_path)
  (define (convert)
    (write-string "digraph {") (newline)
    (write-string "subgraph {") (newline)
    (write-string " rank=same; ")
    (for-each (lambda (n)
                (write-string (node-string n))
                (write-string "; "))
              (filter-nodes cfg cfg:global-node?))
    (write-string "}") (newline)
    (for-each (lambda (e)
                (let ((snode (edge-src-node e))
                      (dnode (edge-dest-node e))
                      (etype (get-edge-data e)))
                  (write-string (node-string snode))
                  (write-string "->")
                  (write-string (node-string dnode))
                  (write-string " [color=")
                  (cond
                   ((function-def? etype) (write "red"))
                   ((function-call? etype) (write "blue"))
                   (else (write "green")))
                  (write-string "];") (newline)))
              (get-edges (cfg:get-graph cfg)))
    (for-each (lambda (n)
                (write-string (node-string n))
                (write-string " [label=")
                (write (obj->string (cfg:node-name n)))
                (write-string ", shape=")
                (cond
                 ;((cfg:root-node? n) (write "point"))
                 ((cfg:global-node? n) (write "box"))
                 ((cfg:undefined-node? n) (write "doublecircle"))
                 ((cfg:primitive-node? n) (write "diamond"))
                 (else (write "ellipse")))
                (write-string "];") (newline))
```

```
            (get-nodes (cfg:get-graph cfg)))
   (write-string "}") (newline)
   'ok)
 (if (default-object? file_path)
     (convert)
     (with-output-to-file file_path convert)))
```

```
;;; Unit testing
(define *g* (create-cfg))
(assert (cfg? *g*))

;;; simple example program
;;;
;;;(define (baz) #f)
;;;(define (foo)
;;;  (define (bar)
;;;    (baz))
;;;  (bar))

(define *cfg* (create-cfg))
(define *baz (define-global-func *cfg* 'baz))
(define *foo (define-global-func *cfg* 'foo))
(define *bar (define-sub-function *cfg* *foo 'bar))
(define bar->baz (add-function-call *cfg* *bar 'baz))
(define foo->bar (add-function-call *cfg* *foo 'bar))
(add-execution foo->bar '() '())
(add-execution bar->baz '() '())
(add-execution bar->baz '() '())


(assert (eq? (cfg:defined-by *baz) (cfg:get-root *cfg*)) "baz not in global scope")
(assert (eq? (cfg:defined-by *foo) (cfg:get-root *cfg*)) "foo not in global scope")
(assert (eq? (cfg:defined-by *bar) *foo) "bar not in foo's scope")

(assert (equal? (cfg:get-defines *foo) `(,*bar)) "foo doesn't define bar")
(assert (null? (cfg:get-defines *baz)) "baz defines something?")
(assert (null? (cfg:get-defines *bar)) "bar defines something?")

(assert (equal? (cfg:get-peers *foo) `(,*baz)) "baz isn't foo's peer")
(assert (equal? (cfg:get-peers *baz) `(,*foo)) "foo isn't baz's peer")
(assert (null? (cfg:get-peers *bar)) "bar has peer")

(assert (= (length (get-executions foo->bar)) 1))
(assert (= (length (get-executions bar->baz)) 2))


;;; simple example of adding a call before it's defined
;;;(define (foo)
;;;  (define (bar)
;;;    (baz))
;;;  (bar))
;;;(define (baz) #f)
(define *cfg2* (create-cfg))
(define *foo2 (define-global-func *cfg2* 'foo))
(define *bar2 (define-sub-function *cfg2* *foo2 'bar))
(add-function-call *cfg2* *bar2 'baz)
(add-function-call *cfg2* *foo2 'bar)
(define *baz2 (define-global-func *cfg2* 'baz))

(assert (eq? (cfg:defined-by *baz2) (cfg:get-root *cfg2*)) "baz not in global scope")
(assert (eq? (cfg:defined-by *foo2) (cfg:get-root *cfg2*)) "foo not in global scope")
(assert (eq? (cfg:defined-by *bar2) *foo2) "bar not in foo's scope")

(assert (equal? (cfg:get-defines *foo2) `(,*bar2)) "foo doesn't define bar")
(assert (null? (cfg:get-defines *baz2)) "baz defines something?")
(assert (null? (cfg:get-defines *bar2)) "bar defines something?")

(assert (equal? (cfg:get-peers *foo2) `(,*baz2)) "baz isn't foo's peer")
(assert (equal? (cfg:get-peers *baz2) `(,*foo2)) "foo isn't baz's peer")
(assert (null? (cfg:get-peers *bar2)) "bar has peer")


;;; simple example of adding a call before it's defined
;;;(define (foo)
;;;  (define (bar)
;;;    (baz))
;;;  (bar))
;;;(define (bar)
;;;  (define (baz) #f)
```

```
;;;     (baz))
(define *cfg3* (create-cfg))
(define *foo3 (define-global-func *cfg3* 'foo))
(define *bar3 (define-sub-function *cfg3* *foo3 'bar))
(add-function-call *cfg3* *bar3 'baz)
(add-function-call *cfg3* *foo3 'bar)
(define *bar3_2 (define-global-func *cfg3* 'bar))
(define *baz3 (define-sub-function *cfg3* *bar3_2 'baz))
(add-function-call *cfg3* *bar3_2 'baz)


'passed
```

```
;;; search depth first starting from the start node. Pred is a predicate mapping
;;; from edge to a boolean value, indicating whether to traverse it or not.
(define (visit-nodes start pred)
  (define (get-edges node)
    (filter pred (node-outgoing-edges node)))
  (let visit-sub-nodes ((start start)
                        (visited-nodes '()))
    (if (in? start visited-nodes)
        visited-nodes
        (let lp ((visited-nodes (append `(,start) visited-nodes))
                 (edges (get-edges start)))
          (if (null? edges) visited-nodes
              (lp (visit-sub-nodes (edge-dest-node (car edges)) visited-nodes)
                  (cdr edges)))))))

;;; Given a cfg, find any dead/unreachable code. Returns a list of
;;; dead/unreachable nodes. Returns an empty list if none are found.  Dead code
;;; is considered to be any sub-graph of the cfg which is not reachable from the
;;; root node by following any edges.
(define (find-dead-code cfg)
  (let ((alive-nodes (visit-nodes (cfg:get-root cfg) (lambda (n) #t)))) ;; visit all edges
    (remove (lambda (n)
              (or (in? n alive-nodes)
                  (cfg:primitive-node? n))) (graph-nodes (cfg:get-graph cfg)))))

;;; Return list of any undefined procedures in the cfg. Returns an empty list if
;;; none are found. An undefined procedure is any procedure which is called from
;;; a node defined in the graph, but there is no valid definition to be
;;; called. Put another way, this detects any calls which would result in a
;;; runtime error complaining that some procedure is not defined.
(define (find-undefined-procedures cfg)
  (filter-nodes cfg cfg:undefined-node?))

;;; Return list of nodes for any unused procedures in cfg. An unused procedure
;;; is any procedure which is defined outside of the global scope and which is
;;; never called. Since it is not in the global scope, we can conclude that it
;;; will never be called. Notice that the subtle difference between this and
;;; dead code, which can never be called/will never be executed: unused code
;;; could be called with a slight modification to the source code. Returns an
;;; empty list if all code is used.
(define (find-unused-code cfg)
  (let ((called-nodes ;; collect all nodes which are called
         (apply append
          (map
           (lambda (n) (visit-nodes n (lambda (e) (function-call?
                                                   (get-edge-data e)))))
                   (filter-nodes cfg cfg:global-node?)))))
    (remove (lambda (n) (or (cfg:root-node? n) ;; don't return root node
                            (cfg:global-node? n)
                            (cfg:primitive-node? n)
                            (in? n called-nodes)))
            (graph-nodes (cfg:get-graph cfg)))))

(define (analyze-cfg cfg)
  (let ((dead-nodes (find-dead-code cfg))
        (undefined-nodes (find-undefined-procedures cfg))
        (unused-nodes (find-unused-code cfg))
        (problem #f))
    (if (not (null? dead-nodes))
        (begin
          (for-each (lambda (n)
                      (write-line (string-append
                                   "function " (obj->string (cfg:node-name n))
                                   " is unreachable")))
                    dead-nodes)
          (set! problem #t)))
    (if (not (null? undefined-nodes))
        (begin
          (for-each (lambda (n)
                      (for-each (lambda (e)
                                  (write-line
                                   (string-append
```

```
                                    (obj->string (cfg:node-name (edge-src-node e)))
                                    " calls undefined function "
                                    (obj->string (cfg:node-name n)))))
                                (filter (lambda (e)
                                          (function-call? (get-edge-data e)))
                                        (get-incoming-edges n))))
                    undefined-nodes)
          (set! problem #t)))
    (if (not (null? unused-nodes))
        (begin
          (for-each (lambda (n)
                      (write-line (string-append "function "
                                                 (obj->string (cfg:node-name n))
                                                 " is unused")))
                    unused-nodes)
          (set! problem #t)))
    (if (not problem)
        (begin (write-string "No problems found.") (newline)))
    (not problem)))
```

```
;;;; Separating analysis from execution.
;;;   Generic analysis, but not prepared for
;;;   extension to handle nonstrict operands.

;;; EVALUATION
;;; Takes place in two separate phases:
;;;   1) analyze compiles the expression into a combinator
;;;   2) run the combinator with the given environment to yield the answer cell
;;;
;;;   exp - raw expression read in
;;;   env - environment in which to execute this
;;;   tail? - boolean value if the expression being evaluated is in tail position.
;;;   returns - cell of answer
(define (eval exp env tail?)
  ((analyze exp rootnode) env tail?))

;;; ANALYZE
;;; A function which, when given an expression returns a combinator
;;; A combinator is a function which given and environment produces
;;; The result of evaluating the expression in the environment
;;; Returns the cell of the answer
(define analyze
  (make-generic-operator 2 'analyze
    (lambda (exp parent-node)
      (cond ((application? exp)
             (analyze-any-application exp parent-node))
            (else
             (error "Unknown expression type"
                    exp))))))

(define (analyze-self-evaluating exp parent-node)
  (let ((return (default-cell exp)))
    (lambda (env tail?)
      (if tail? (apply-delayed-work return))
      return)))

(defhandler analyze analyze-self-evaluating self-evaluating? any?)

(define (analyze-quoted exp parent-node)
  (let ((qval (default-cell (text-of-quotation exp))))
    (lambda (env tail?)
      (if tail? (apply-delayed-work qval))
      qval)))

(defhandler analyze analyze-quoted quoted? any?)

(define (analyze-variable exp parent-node)
  (lambda (env tail?)
    (let ((return (get-variable-cell exp env)))
      (if tail? (apply-delayed-work return))
      return)))

(defhandler analyze analyze-variable variable? any?)

(define (analyze-if exp parent-node)
  (let ((pproc (analyze (if-predicate exp) parent-node))
        (cproc (analyze (if-consequent exp) parent-node))
        (aproc (analyze (if-alternative exp) parent-node)))
    (lambda (env tail?)
      (if (true? (pproc env #f)) (cproc env tail?) (aproc env tail?)))))

(defhandler analyze analyze-if if? any?)

(define (analyze-lambda exp parent-node)
  (let ((vars (lambda-parameters exp))
        (bproc (analyze (lambda-body exp) parent-node)))
    (lambda (env tail?)
      (default-cell (make-compound-procedure vars bproc env)))))

(defhandler analyze analyze-lambda lambda? any?)

(define (analyze-any-application exp parent-node)
```

```
  (let ((destination-name (operator exp)))
    ;; add a call edge
    (if *verbose* (begin (pp "Adding edge to/from") (pp exp) (pp parent-node)))
    (let ((edge (if (rootnode? parent-node)
                    (add-global-call *g* destination-name)
                    (add-function-call *g* parent-node destination-name)))
          (combinator (analyze-application exp parent-node)))
      (lambda (env tail?)
        (if tail?
            (begin (enqueue *pending-execs* (list edge (operands exp)))
                   (combinator env tail?))
            (let ((return-value (combinator env tail?)))
              (if *verbose* (begin (pp "Adding execution") (pp exp) (pp edge)
                                   (pp (operands exp)) (pp return-value)))
              (add-execution edge (operands exp) return-value)
              return-value))))))

(define (analyze-application exp parent-node)
  (define (analyze-tmp exp)
    (analyze exp parent-node))
  (let ((fproc (analyze (operator exp) parent-node))
        (aprocs (map analyze-tmp (operands exp))))
    (lambda (env tail?)
      (let* ((proc-cell (fproc env #f))
             (proc-tags (cell-tags proc-cell)))
        (if tail?
            (begin (enqueue *pending-tags* proc-tags)
                   (execute-application
                    proc-cell
                    (map (lambda (aproc) (aproc env #f)) aprocs) #t))
            (add-cell-tags!
             (execute-application
              proc-cell
              (map (lambda (aproc) (aproc env #f)) aprocs)
              #f)
             (cell-tags proc-cell)))))))

(defhandler analyze analyze-application escaped-apply? any?)

(define execute-application
  (make-generic-operator 3 'execute-application
    (lambda (proc-cell args-cells tail?)
      (error "Unknown procedure type" proc-cell))))

(defhandler execute-application
  (lambda (proc-cell args-cells tail?)
    (let* ((proc (cell-value proc-cell))
           (vars (procedure-parameters proc))
           (body (procedure-body proc))
           (proc-env (procedure-environment proc)))
      (let ((new-env (extend-environment vars args-cells proc-env)))
        (body new-env tail?))))
  compound-procedure?)

(defhandler execute-application
  (lambda (proc-cell args-cells tail?)
    (let ((return (apply-primitive-procedure proc-cell args-cells)))
      (if tail? (apply-delayed-work return))
      return))
  strict-primitive-procedure?)

(define (analyze-sequence exps parent-node)
  (define (sequentially proc1 proc2)
    (lambda (env tail?) (proc1 env #f) (proc2 env #f)))
  (define (loop first-proc rest-procs)
    (cond ((null? rest-procs) first-proc)
          ((= 1 (length rest-procs))
           (lambda (env tail?) (first-proc env #f) ((car rest-procs) env #t)))
          (else (loop (sequentially first-proc (car rest-procs))
                      (cdr rest-procs)))))
  (if (null? exps) (error "Empty sequence"))
  (define (analyze-tmp exp)
```

```
    (analyze exp parent-node))
  (let ((procs (map analyze-tmp exps)))
    (loop (car procs) (cdr procs))))

(defhandler analyze
  (lambda (exp parent-node)
    (analyze-sequence (begin-actions exp) parent-node))
  begin? any?)


(define (analyze-assignment exp parent-node)
  (let ((var (assignment-variable exp))
        (vproc (analyze (assignment-value exp) parent-node)))
    (lambda (env tail?)
      (let ((cell (vproc env #f)))
        (set-variable-cell! var cell env)
        (default-cell 'ok)))))

(defhandler analyze analyze-assignment assignment? any?)

(define (analyze-definition exp parent-node)
  (let ((var (definition-variable exp))
        (value (definition-value exp))
        (parent-node parent-node))
    (if (lambda? value)
        ;; Defining a procedure
        (if (rootnode? parent-node)
            (set! parent-node (define-global-func *g*
                                (definition-variable exp)))
            (set! parent-node (define-sub-function *g* parent-node
                                (definition-variable exp)))))
    (let ((vproc (analyze value parent-node)))
      (lambda (env tail?)
        (let ((cell (vproc env #f)))
          (define-variable! var cell env)
          (default-cell 'ok))))))

(defhandler analyze analyze-definition definition? any?)

;;; Macros (definitions are in syntax.scm)

(defhandler analyze (lambda (exp node) (analyze (cond->if exp) node))
  cond? any?)

(defhandler analyze (lambda (exp node) (analyze (let->combination exp) node))
  let? any?)

;;; Special forms to get tags
(define (analyze-get-tags exp parent-node)
  (begin
    (if *verbose* (begin (pp "analyze-get-tags") (pp exp)))
    (let ((var-obj (analyze (tag-var exp) parent-node)))
      (lambda (env tail?)
        (let* ((cell (var-obj env #f))
               (return (make-cell (cell-tags cell) (cell-tags cell))))
          (if tail? (apply-delayed-work return))
          return)))))

(define (analyze-add-tag exp parent-node)
  (begin
    (if *verbose* (begin (pp "analyze-add-tag") (pp exp)))
    (let ((aobj (analyze (tag-var exp) parent-node))
          (atag (analyze (tag-tag exp) parent-node)))
      (lambda (env tail?)
        (let* ((cell (aobj env #f))
               (tag-cell (atag env #f))
               (tag (cell-value tag-cell))
               (return (add-cell-tag! cell tag)))
          (if tail? (apply-delayed-work return))
          return)))))

(defhandler analyze analyze-get-tags get-tags? any?)
```

```
(defhandler analyze analyze-add-tag add-tag? any?)

(define (analyze-ignore exp parent-node)
  (lambda (env tail?)
    (set! hook/repl-eval default-repl-eval)
    (let ((ret (default-cell (default-repl-eval (cadr exp)
                    generic-evaluation-environment 'sussman-explain-me?))))
      (set! hook/repl-eval our-repl-eval)
      (if tail? (apply-delayed-work ret))
      ret)))

(defhandler analyze analyze-ignore
  (lambda (exp)
    (and (list? exp)
         (eq? 'ignore (car exp)))))
```

```
;;; -*- Mode:Scheme -*-

(declare (usual-integrations))

;;; Self-evaluating entities

(define (self-evaluating? exp)
  (or (number? exp)
      (eq? exp #t)
      (eq? exp #f)
      (string? exp)))   ; Our prompt (viz., "EVAL==> ") is a string.

;;; Variables

(define (variable? exp) (symbol? exp))

(define (same-variable? var1 var2) (eq? var1 var2))  ;; Nice abstraction

;;; Special forms (in general)

(define (tagged-list? exp tag)
  (and (pair? exp)
       (eq? (car exp) tag)))

;;; Quotations

(define (quoted? exp) (tagged-list? exp 'quote))

(define (text-of-quotation quot) (cadr quot))

;;; Assignment--- SET!

(define (assignment? exp) (tagged-list? exp 'set!))
(define (permanent-assignment? exp) (tagged-list? exp 'set!!))

(define (assignment-variable assn) (cadr  assn))
(define (assignment-value    assn) (caddr assn))

;;; Definitions

(define (definition? exp) (tagged-list? exp 'define))

(define (definition-variable defn)
  (if (variable? (cadr defn))                    ;;   (DEFINE  foo      ...)
      (cadr  defn)
      (caadr defn)))                             ;;   (DEFINE (foo ...) ...)

(define (definition-value defn)
  (if (variable? (cadr defn))                    ;;   (DEFINE  foo        ...)
      (caddr defn)
      (cons 'lambda                              ;;   (DEFINE (foo p...) b...)
            (cons (cdadr defn)                   ;; = (DEFINE  foo
                  (cddr  defn)))))               ;;     (LAMBDA (p...) b...))

;;; LAMBDA expressions

(define (lambda? exp) (tagged-list? exp 'lambda))
(define (lambda-parameters lambda-exp) (cadr lambda-exp))
(define (lambda-body lambda-exp)
  (let ((full-body (cddr lambda-exp)))
    (sequence->begin full-body)))


(define declaration? pair?)

(define (parameter-name var-decl)
  (if (pair? var-decl)
      (car var-decl)
      var-decl))

(define (lazy? var-decl)
  (and (pair? var-decl)
```

```
       (memq 'lazy (cdr var-decl))
       (not (memq 'memo (cdr var-decl)))))

(define (lazy-memo? var-decl)
  (and (pair? var-decl)
       (memq 'lazy (cdr var-decl))
       (memq 'memo (cdr var-decl))))

(define (sequence->begin seq)
  (cond ((null? seq) seq)
        ((null? (cdr seq)) (car seq))
        ((begin? (car seq)) seq)
        (else (make-begin seq))))

(define (make-begin exp) (cons 'begin exp))
```

```
;;; If conditionals

(define (if? exp) (tagged-list? exp 'if))

(define (if-predicate exp) (cadr exp))

(define (if-consequent exp) (caddr exp))

(define (if-alternative exp)
  (if (not (null? (cdddr exp)))
      (cadddr exp)
      'the-unspecified-value))

(define (make-if pred conseq alternative)
  (list 'IF pred conseq alternative))


;;; COND Conditionals

(define (cond? exp) (tagged-list? exp 'cond))

(define (clauses cndl) (cdr cndl))
(define (no-clauses? clauses) (null? clauses))
(define (first-clause clauses) (car clauses))
(define (rest-clauses clauses) (cdr clauses))
(define (else-clause? clause) (eq? (predicate clause) 'else))

(define (predicate clause) (car clause))

(define (actions clause)
  (sequence->begin (cdr clause)))

(define (cond->if cond-exp)
  (define (expand clauses)
    (cond ((no-clauses? clauses)
           (list 'error "COND: no values matched"))
          ((else-clause? (car clauses))
           (if (no-clauses? (cdr clauses))
               (actions (car clauses))
               (error "else clause isn't last -- INTERP" exp)))
          (else
           (make-if (predicate (car clauses))
                    (actions (car clauses))
                    (expand (cdr clauses))))))
  (expand (clauses cond-exp)))
```

```
;;; BEGIN expressions (a.k.a. sequences)

(define (begin? exp) (tagged-list? exp 'begin))
(define (begin-actions begin-exp) (cdr begin-exp))

(define (last-exp? seq) (null? (cdr seq)))
(define (first-exp seq) (car seq))
(define (rest-exps seq) (cdr seq))
(define no-more-exps? null?)            ; for non-tail-recursive vers.

;;; LET expressions

(define (let? exp) (tagged-list? exp 'let))
(define (let-bound-variables let-exp)
  (map car (cadr let-exp)))
(define (let-values let-exp) (map cadr (cadr let-exp)))
(define (let-body let-exp) (sequence->begin (cddr let-exp)))
(define (let->combination let-exp)
  (let ((names (let-bound-variables let-exp))
        (values (let-values let-exp))
        (body (let-body let-exp)))
    (cons (list 'LAMBDA names body)
          values)))

;;; Procedure applications -- NO-ARGS? and LAST-OPERAND? added

;;; Procedures we don't want to add to our graph
(define (escaped-apply? exp)
  (any (lambda (proc-name) (tagged-list? exp proc-name)) escaped-procs))

(define (application? exp)
  (pair? exp))

(define (no-args? exp)                          ;; Added for tail recursion
  (and (pair? exp)
       (null? (cdr exp))))

(define (args-application? exp)                 ;; Changed from 5.2.1
  (and (pair? exp)
       (not (null? (cdr exp)))))


(define (operator app) (car app))
(define (operands app) (cdr app))

(define (last-operand? args)                    ;; Added for tail recursion
  (null? (cdr args)))

(define (no-operands? args) (null? args))
(define (first-operand args) (car args))
(define (rest-operands args) (cdr args))
```

```
;;; Another special form that will be needed later.

(define (amb? exp)
  (and (pair? exp) (eq? (car exp) 'amb)))

(define (amb-alternatives exp) (cdr exp))

;;; Our syntactic cases
(define (add-tag? exp) (tagged-list? exp 'add-tag))
(define (get-tags? exp) (tagged-list? exp 'get-tags))

(define (tag-var exp) (cadr exp))
(define (tag-tag exp) (caddr exp))
```

```
;;; -*- Mode:Scheme -*-

(declare (usual-integrations))

;;; Structure to abstract the idea of a scheme object from out guest interpreter
;;; Allows us to add any information we want to any object
(define-structure cell value tags)

;;; Creates a cell if only the value is known.
;;; This allows a single place to change conventions if other
;;; metadata is unspecified about the object.
(define (default-cell value)
  (make-cell value '()))


;;; Useful helper function to retrieve the cell associated
;;; with a variable in an environment.
;;; Returns the cell found, #f no variable exists.
(define (get-cell var env)
  (let plp ((env env))
    (if (eq? env the-empty-environment)
        #f
        (let scan
            ((vars (environment-variables env))
             (cells (environment-cells env)))
          (cond ((null? vars) (plp (environment-parent env)))
                ((eq? var (car vars)) (car cells))
                (else (scan (cdr vars) (cdr cells)))))))))

;;; Looks for cell of var in env
;;; If not found, finds real scheme value associated with variable
;;; and wraps in default cell
(define (get-variable-cell var env)
  (let ((cell (get-cell var env)))
    (if cell
        cell
        (let ((new-cell (default-cell (lookup-scheme-value var))))
          (let loop ((env env))
            (if (eq? (environment-parent env) the-empty-environment)
                (define-variable! var new-cell env)
                (loop (environment-parent env))))
          new-cell))))

;;; Uses get-variable-cell to find cell, if it exists.
;;; If found, replaces cell contents with given ones
(define (set-variable-cell! var cell env)
  (let ((current-cell (get-variable-cell var env)))
    (set-cell-value! current-cell (cell-value cell))
    (set-cell-tags! current-cell (cell-tags cell))))


;;; Note - Predicates will receive cell as arguments
;;;        NOT values

(define the-unspecified-value (list 'the-unspecified-value))

(define (true? cell)
  (if (cell-value cell) true false))

(define (false? cell)
  (if (cell-value? cell) false true))

;;; Primitive procedures are inherited from Scheme.

(define (strict-primitive-procedure? cell)
  (procedure? (cell-value cell)))

(define (apply-primitive-procedure proc-cell args-cells)
  (let* ((proc (cell-value proc-cell))
         (value (apply proc (map cell-value args-cells)))
         (all-tags (apply append (map cell-tags args-cells))))
    (let loop ((all-tags all-tags)
```

```
                (tags '()))
      (if (null? all-tags)
          (make-cell value tags)
          (let inner-loop ((tags-left tags))
            (if (null? tags-left)
                (loop (cdr all-tags)
                      (cons (car all-tags) tags))
                (if (eq? (car all-tags) (car tags-left))
                    (loop (cdr all-tags)
                          tags)
                    (inner-loop (cdr tags-left)))))))))

;;; Compound procedures

(define (make-compound-procedure vars bproc env)
  (vector 'compound-procedure vars bproc env))

(define (compound-procedure? cell)
  (let ((obj (cell-value cell)))
    (vector-compound-procedure? obj)))

;;; Introduced this becuase the REPL loop uses this for pretty printing
(define (vector-compound-procedure? obj)
    (and (vector? obj)
         (eq? (vector-ref obj 0) 'compound-procedure)))

(define (procedure-parameters p) (vector-ref p 1))
(define (procedure-body p) (vector-ref p 2))
(define (procedure-environment p) (vector-ref p 3))

;;; An ENVIRONMENT is a chain of FRAMES, made of vectors.

(define (extend-environment variables cells base-environment)
  (if (list? variables)
      (inner-extend-environment variables cells base-environment)
      (let loop ((vars variables)
                 (cells cells)
                 (out-vars '())
                 (out-cells '()))
        (if (pair? vars)
            (loop (cdr vars)
                  (cdr cells)
                  (cons (car vars) out-vars)
                  (cons (car cells) out-cells))
            ;;; case where vars is actually a variable
            (let* ((new-vars (cons vars out-vars))
                   (new-cell (add-cell-tags! (default-cell (map cell-value cells))
                                             (apply append (map cell-tags cells))))
                   (new-args-cells (cons new-cell out-cells)))
              (inner-extend-environment new-vars new-args-cells base-environment))))))

(define (inner-extend-environment variables cells base-environment)
  (if (fix:= (length variables) (length cells))
      (vector variables cells base-environment)
      (if (fix:< (length variables) (length cells))
          (error "Too many arguments supplied" variables cells)
          (error "Too few arguments supplied" variables cells))))

;;; These are useful in maintaing the abstraction of an environment

(define (environment-variables env) (vector-ref env 0))
(define (environment-cells env) (vector-ref env 1))
(define (environment-parent env) (vector-ref env 2))

(define (set-environment-variables! env vars)
  (vector-set! env 0 vars))
(define (set-environment-cells! env cells)
  (vector-set! env 1 cells))

(define the-empty-environment '())
```

```
;;; Extension to make underlying Scheme values available to interpreter

(define (lookup-scheme-value var)
  (lexical-reference generic-evaluation-environment var))

(define (define-variable! var cell env)
  (if (eq? env the-empty-environment)
      (error "Unbound variable -- DEFINE" var) ;should not happen.
      (let scan
          ((vars (environment-variables env))
           (cells (environment-cells env)))
        (cond ((null? vars)
               (set-environment-variables! env (cons var (environment-variables env)))
               (set-environment-cells! env (cons cell (environment-cells env))))
              ((eq? var (car vars))
               (set-car! cells cell))
              (else
               (scan (cdr vars) (cdr cells)))))))
```

```
;;;; Read-eval-print loop for extended Scheme interpreter

(declare (usual-integrations write write-line pp eval))

(define write
  (make-generic-operator 1 'write
    (access write user-initial-environment)))

(define write-line
  (make-generic-operator 1 'write-line
    (access write-line user-initial-environment)))

(define pp
  (make-generic-operator 1 'pretty-print
    (access pp user-initial-environment)))

(define (procedure-printable-representation procedure)
  `(compound-procedure
    ,(procedure-parameters procedure)
    ,(procedure-body procedure)
    <procedure-environment>))

(defhandler write
  (compose write procedure-printable-representation)
  vector-compound-procedure?)

(defhandler write-line
  (compose write-line procedure-printable-representation)
  vector-compound-procedure?)

(defhandler pp
  (compose pp procedure-printable-representation)
  vector-compound-procedure?)


(define (read) (prompt-for-command-expression "eval> "))

(define the-global-environment 'not-initialized)

(define default-repl-eval 'undefined)

(define (our-repl-eval input default/env default/repl)
  ;;; Technically nothing at the top level is never a site of tail-recursion,
  ;;; but with the way the repl is configured, the evaluation by eval is in tail
  ;;; position.
  (cell-value (eval input the-global-environment #t)))

(define (init)
  (set! the-global-environment
        (extend-environment '() '() the-empty-environment))
  (reset-cfg)
  ;;; Only run once! Otherwise we risk losing our reference to initial repl!
  (if (eq? default-repl-eval 'undefined)
      (set! default-repl-eval hook/repl-eval))
  (go)
)

(define (go)
  (set! hook/repl-eval our-repl-eval)
  "entered SchLint"
)

(define (exit)
  (set! hook/repl-eval default-repl-eval)
  "exited SchLint"
)
```

```
;;;;           Most General Generic-Operator Dispatch

(declare (usual-integrations))

;;; Generic-operator dispatch is implemented here by a discrimination
;;; list, where the arguments passed to the operator are examined by
;;; predicates that are supplied at the point of attachment of a
;;; handler (by DEFHANDLER).

;;; To be the correct branch all arguments must be accepted by the
;;; branch predicates, so this makes it necessary to backtrack to find
;;; another branch where the first argument is accepted if the second
;;; argument is rejected.  Here backtracking is implemented using #f
;;; as a failure return, requiring further search.  A success is
;;; consummated by calling the WIN procedure.

;;; The discrimination list has the following structure: it is a
;;; possibly improper alist whose "keys" are the predicates that are
;;; applicable to the first argument.  If a predicate matches the
;;; first argument, the cdr of that alist entry is a discrimination
;;; list for handling the rest of the arguments.  If a discrimination
;;; list is improper, then the cdr at the end of the backbone of the
;;; alist is the default handler to apply (all remaining arguments are
;;; implicitly accepted).
```

```
(define (make-generic-operator arity #!optional name default-operation)
  (let ((record (make-operator-record arity)))
    (define (find-branch tree arg win)
      (let loop ((tree tree))
        (cond ((pair? tree)
               (or (and ((caar tree) arg)
                        (win (cdar tree)))
                   (loop (cdr tree))))
              ((null? tree) #f)
              (else tree))))
    (define (identity x) x)
    (define (find-handler arguments)
      (let loop ((tree (operator-record-tree record))
                 (args arguments))
        (find-branch tree (car args)
          (if (pair? (cdr args))
              (lambda (branch) (loop branch (cdr args)))
              identity))))
    (define (operator . arguments)
      (if (not (acceptable-arglist? arguments arity))
          (error:wrong-number-of-arguments
           (if (default-object? name) operator name) arity arguments))
      (apply (find-handler arguments) arguments))

    (set-operator-record! operator record)
    (if (not (default-object? name))
        (set-operator-record! name record))

    (set! default-operation
      (if (default-object? default-operation)
          (named-lambda (no-handler . arguments)
            (error "Generic operator inapplicable:"
                   (if (default-object? name) operator name)
                   arguments))
          default-operation))
    (assign-operation operator default-operation)

    operator))

(define *generic-operator-table*
  (make-eq-hash-table))
```

```
(define (get-operator-record operator)
  (hash-table/get *generic-operator-table* operator #f))

(define (set-operator-record! operator record)
  (hash-table/put! *generic-operator-table* operator record))

(define (make-operator-record arity) (cons arity '()))
(define (operator-record-arity record) (car record))
(define (operator-record-tree record) (cdr record))
(define (set-operator-record-tree! record tree) (set-cdr! record tree))

(define (acceptable-arglist? lst arity)
  (let ((len (length lst)))
    (and (fix:<= (procedure-arity-min arity) len)
         (or (not (procedure-arity-max arity))
             (fix:>= (procedure-arity-max arity) len)))))

(define (assign-operation operator handler . argument-predicates)
  (let ((record (get-operator-record operator))
        (arity (length argument-predicates)))
    (if record
        (begin
          (if (not (<= arity (procedure-arity-min
                              (operator-record-arity record))))
              (error "Incorrect operator arity:" operator))
          (bind-in-tree
           argument-predicates
           handler
           (operator-record-tree record)
           (lambda (new)
             (set-operator-record-tree! record new))))
        (error "Assigning a handler to an undefined generic operator"
               operator)))
  operator)

(define defhandler assign-operation)
```

```
(define (bind-in-tree keys handler tree replace!)
  (let loop ((keys keys) (tree tree) (replace! replace!))
    (if (pair? keys)
        ;; There are argument predicates left
        (let find-key ((tree* tree))
          (if (pair? tree*)
              (if (eq? (caar tree*) (car keys))
                  ;; There is already some discrimination list keyed
                  ;; by this predicate: adjust it according to the
                  ;; remaining keys
                  (loop (cdr keys)
                        (cdar tree*)
                        (lambda (new)
                          (set-cdr! (car tree*) new)))
                  (find-key (cdr tree*)))
              (let ((better-tree
                     (cons (cons (car keys) '()) tree)))
                ;; There was no entry for the key I was looking for.
                ;; Create it at the head of the alist and try again.
                (replace! better-tree)
                (loop keys better-tree replace!))))
        ;; Ran out of argument predicates
        (if (pair? tree)
            ;; There is more discrimination list here, because my
            ;; predicate list is a proper prefix of the predicate list
            ;; of some previous assign-operation.  Insert the handler
            ;; at the end, causing it to implicitly accept any
            ;; arguments that fail all available tests.
            (let ((p (last-pair tree)))
              (if (not (null? (cdr p)))
                  (warn "Replacing a default handler:" (cdr p) handler))
              (set-cdr! p handler))
            (begin
              ;; There is no discrimination list here, because my
              ;; predicate list is not the proper prefix of that of
              ;; any previous assign-operation.  This handler becomes
              ;; the discrimination list, accepting further arguments
              ;; if any.
              (if (not (null? tree))
                  (warn "Replacing a handler:" tree handler))
              (replace! handler))))))
```

```
#|
;;; Demonstration of handler tree structure.
;;; Note: symbols were used instead of procedures

(define foo (make-generic-operator 3 'foo 'foo-default))
(pp (get-operator-record foo))
(3 . foo-default)

(defhandler foo 'two-arg-a-b 'a 'b)
(pp (get-operator-record foo))
(3 (a (b . two-arg-a-b)) . foo-default)

(defhandler foo 'two-arg-a-c 'a 'c)
(pp (get-operator-record foo))
(3 (a (c . two-arg-a-c) (b . two-arg-a-b)) . foo-default)

(defhandler foo 'two-arg-b-c 'b 'c)
(pp (get-operator-record foo))
(3 (b (c . two-arg-b-c))
   (a (c . two-arg-a-c) (b . two-arg-a-b))
   . foo-default)

(defhandler foo 'one-arg-b 'b)
(pp (get-operator-record foo))
(3 (b (c . two-arg-b-c) . one-arg-b)
   (a (c . two-arg-a-c) (b . two-arg-a-b))
   . foo-default)

(defhandler foo 'one-arg-a 'a)
(pp (get-operator-record foo))
(3 (b (c . two-arg-b-c) . one-arg-b)
   (a (c . two-arg-a-c) (b . two-arg-a-b) . one-arg-a)
   .
   foo-default)

(defhandler foo 'one-arg-a-prime 'a)
;Warning: Replacing a default handler: one-arg-a one-arg-a-prime

(defhandler foo 'two-arg-a-b-prime 'a 'b)
;Warning: Replacing a handler: two-arg-a-b two-arg-a-b-prime

(defhandler foo 'three-arg-x-y-z 'x 'y 'z)
(pp (get-operator-record foo))
(3 (x (y (z . three-arg-x-y-z)))
   (b (c . two-arg-b-c) . one-arg-b)
   (a (c . two-arg-a-c) (b . two-arg-a-b-prime) . one-arg-a-prime)
   .
   foo-default)
|#
```

```
;;; search depth first starting from the start node. Pred is a predicate mapping
;;; from edge to a boolean value, indicating whether to traverse it or not.
(define (visit-nodes start pred)
  (define (get-edges node)
    (filter pred (node-outgoing-edges node)))
  (let visit-sub-nodes ((start start)
                        (visited-nodes '()))
    (if (in? start visited-nodes)
        visited-nodes
        (let lp ((visited-nodes (append `(,start) visited-nodes))
                 (edges (get-edges start)))
          (if (null? edges) visited-nodes
              (lp (visit-sub-nodes (edge-dest-node (car edges)) visited-nodes)
                  (cdr edges)))))))

;;; Given a cfg, find any dead/unreachable code. Returns a list of
;;; dead/unreachable nodes. Returns an empty list if none are found.  Dead code
;;; is considered to be any sub-graph of the cfg which is not reachable from the
;;; root node by following any edges.
(define (find-dead-code cfg)
  (let ((alive-nodes (visit-nodes (cfg:get-root cfg) (lambda (n) #t)))) ;; visit all edges
    (remove (lambda (n)
              (or (in? n alive-nodes)
                  (cfg:primitive-node? n))) (graph-nodes (cfg:get-graph cfg)))))

;;; Return list of any undefined procedures in the cfg. Returns an empty list if
;;; none are found. An undefined procedure is any procedure which is called from
;;; a node defined in the graph, but there is no valid definition to be
;;; called. Put another way, this detects any calls which would result in a
;;; runtime error complaining that some procedure is not defined.
(define (find-undefined-procedures cfg)
  (filter-nodes cfg cfg:undefined-node?))

;;; Return list of nodes for any unused procedures in cfg. An unused procedure
;;; is any procedure which is defined outside of the global scope and which is
;;; never called. Since it is not in the global scope, we can conclude that it
;;; will never be called. Notice that the subtle difference between this and
;;; dead code, which can never be called/will never be executed: unused code
;;; could be called with a slight modification to the source code. Returns an
;;; empty list if all code is used.
(define (find-unused-code cfg)
  (let ((called-nodes ;; collect all nodes which are called
         (apply append
          (map
           (lambda (n) (visit-nodes n (lambda (e) (function-call?
                                                   (get-edge-data e)))))
                   (filter-nodes cfg cfg:global-node?)))))
    (remove (lambda (n) (or (cfg:root-node? n) ;; don't return root node
                            (cfg:global-node? n)
                            (cfg:primitive-node? n)
                            (in? n called-nodes)))
            (graph-nodes (cfg:get-graph cfg)))))

(define (analyze-cfg cfg)
  (let ((dead-nodes (find-dead-code cfg))
        (undefined-nodes (find-undefined-procedures cfg))
        (unused-nodes (find-unused-code cfg))
        (problem #f))
    (if (not (null? dead-nodes))
        (begin
          (for-each (lambda (n)
                      (write-line (string-append
                                   "function " (obj->string (cfg:node-name n))
                                   " is unreachable")))
                    dead-nodes)
          (set! problem #t)))
    (if (not (null? undefined-nodes))
        (begin
          (for-each (lambda (n)
                      (for-each (lambda (e)
                                  (write-line
                                   (string-append
```

```
                                    (obj->string (cfg:node-name (edge-src-node e)))
                                    " calls undefined function "
                                    (obj->string (cfg:node-name n)))))
                                (filter (lambda (e)
                                          (function-call? (get-edge-data e)))
                                        (get-incoming-edges n))))
                    undefined-nodes)
          (set! problem #t)))
    (if (not (null? unused-nodes))
        (begin
          (for-each (lambda (n)
                      (write-line (string-append "function "
                                                 (obj->string (cfg:node-name n))
                                                 " is unused")))
                    unused-nodes)
          (set! problem #t)))
    (if (not problem)
        (begin (write-string "No problems found.") (newline)))
    (not problem)))
```

```
(define a 2)
(add-tag a 'aa)
(assert (equal? (get-tags a) '(aa)) "basic tag")

(assert (equal? (get-tags (add-tag 1 'temporary)) '(temporary)) "tagging self-evaluating works")
(add-tag 1 'temp)
(assert (not (equal? (get-tags 1) '(temp))) "tagging self-evaluating sticks...")

(add-tag + 'bar)
(assert (equal? (get-tags (+ 1 2)) '(bar)) "tagging primitive procedure")
(assert (equal? (get-tags (+ a a)) '(bar aa)) "checking multiple tags")

;; a tail-recursive list summing procedure
(define (loop lis sum-so-far)
  (cond ((null? lis)
         sum-so-far)
        (else
         (loop (cdr lis)
               (+ sum-so-far (car lis))))))

;; a friendly wrapper that supplies an initial running sum of 0
(define (list-sum lis)
  (loop lis 0))
(assert (equal? (get-tags (list-sum (list a 2))) '(aa bar)))

;; Test that provenance tracking hasn't broken tail recursion...
(define (foo a) (if (> a 20000) a (foo (+ a 1))))
(foo 0)

'passed
```

```
;;; utility.scm - collection of useful utility functions

(declare (usual-integrations))

(define (identity x) x)

(define (any? x) #t)

(define ((compose f g) x) (f (g x)))


;;; This is to keep the Scheme printer from going into an infinite
;;; loop if you try to print a circular data structure, such as an
;;; environment

(set! *unparser-list-depth-limit* 10)
(set! *unparser-list-breadth-limit* 10)

;;; assert, courtesy of GJS
;;; This is part of paranoid programming.
(define (assert p #!optional error-comment irritant)
  (if (not p)
      (begin
        (if (not (default-object? irritant))
            (pp irritant))
        (error
         (if (default-object? error-comment)
             "Failed assertion"
             error-comment)))))

(define (obj->string obj)
  (with-output-to-string
    (lambda () (write obj))))

(define (empty-queue? q)
  (null? q))

(define (enqueue x q)
  (set! q (append q (list x))))

(define (dequeue)
  (let ((x (car q)))
    (set! q (cdr q))
    x))
```

```
(load "utils" user-initial-environment)
(load "ghelper" user-initial-environment)
(load "syntax" user-initial-environment)
(load "rtdata" user-initial-environment)

(define generic-evaluation-environment
  (extend-top-level-environment user-initial-environment))

(load "graph")
(load "cfg")
(load "checker")

(load "schlint" user-initial-environment)
(load "analyze" generic-evaluation-environment)
(load "repl" generic-evaluation-environment)

(ge generic-evaluation-environment)
```