# AIMA for Chicken Scheme

Peter Danenberg <pcd@roxygen.org>

August 13, 2012

## Contents



# 1 AIMA

## 1.1 aima

**Module** aima

**Description** AIMA contains functions common to agents and environments.

**Exports**

- compose-environments
- debug?
- debug-print
- default-steps
- make-debug-environment
- make-step-limited-environment

- `make-performance-measuring-environment`
- `random-seed`
- `randomize!`
- `simulate`

## 1.2 debug?

**Parameter** `#t`

**Description** Should we print debugging information to stdout?

```
(define debug? (make-parameter #t))
```

## 1.3 debug-print

**Procedure** `(debug-print key value)` → `unspecified`
`(debug-print key value out)` → `unspecified`

**Description** Print key-value pairs if the parameter 'debug?' is true.

**Parameters**

| | |
|---|---|
| `key` | The key to print |
| `value` | The value to print |
| `out` | The port to print to |

```
(define debug-print
  (case-lambda
    ((key value) (debug-print key value #t))
    ((key value out) (if (debug?) (format out "~a: ~a~%" key value)))))
```

## 1.4 random-seed

**Parameter** `#f`

**Description** 'random-seed' is passed to 'randomize!' during 'simulate'.

```
(define random-seed (make-parameter #f))
```

## 1.5 randomize!

**Parameter** `randomize`

**Description** 'randomize!' is called before simulation and is seeded with 'random-seed'.

```
(define randomize! (make-parameter randomize))
```

## 1.6 simulate

**Procedure** `(simulate environment)` → `#f`
`(simulate environment randomize! random-seed)` → `#f`

**Description** Run an environment to completion; an environment is complete when it returns false.

**Parameters**

| | |
|---|---|
| `environment` | The environment to simulate |
| `randomize!` | Function to seed the random-number generator for reproducible results |
| `random-seed` | Seed to seed the random-number generator |

```
(define simulate
  (case-lambda
    ((environment) (simulate environment (randomize!) (random-seed)))
    ((environment randomize! random-seed)
     (if random-seed (randomize! random-seed))
     (loop ((while (environment)))))))
```

## 1.7 compose-environments

**Procedure** `(compose-environments . environments)` → `environment`

**Description** Compose environments into a single environment suitable for 'simulate'.

'compose-environments' effectively 'ands' over its constituent environments every step.

**Parameters**

| | |
|---|---|
| `environments` | The environments to be composed |

```
(define (compose-environments . environments)
  (lambda ()
    (every identity (map (lambda (environment) (environment)) environments))))
```

## 1.8 make-performance-measuring-environment

**Procedure** `(make-performance-measuring-environment measure-performance score-update!)` → `environment`

**Description** Make an environment that updates a score according to a performance measure.

**Parameters**

| | |
|---|---|
| `measure-performance` | A nullary procedure which measures performance |
| `score-update!` | A function which receives the performance measure and updates the score accordingly |

```
(define (make-performance-measuring-environment
         measure-performance
         score-update!)
  (lambda () (score-update! (measure-performance))))
```

### 1.9 default-steps

**Parameter** 1000

**Description** Default number of steps for the step-limited environment

```
(define default-steps (make-parameter 1000))
```

### 1.10 make-step-limited-environment

**Procedure** `(make-step-limited-environment)` → `environment`
`(make-step-limited-environment steps)` → `environment`

**Description** Make an environment that stops simulation after a certain number of steps.

**Parameters**

| | |
|---|---|
| `steps` | The number of steps after which to stop simulating |

```
(define make-step-limited-environment
  (case-lambda
    (() (make-step-limited-environment (default-steps)))
    ((steps)
     (let ((current-step 0))
       (lambda ()
         (set! current-step (+ current-step 1))
         (< current-step steps))))))
```

### 1.11 make-debug-environment

**Syntax** `(make-debug-environment object make-printable-object)` → `environment`

**Description** Make an environment that prints debugging information (according to 'debug?').

**Parameters**

| | |
|---|---|
| `object` | The object to debug |
| `make-printable-object` | A function which optionally transforms the object before printing |

```
(define-syntax
  make-debug-environment
  (er-macro-transformer
```

```
    (lambda (expression rename compare)
      (let ((%print (rename 'debug-print)))
        (match expression
               ((_ object) `(lambda () (,%print ',object ,object)))
               ((_ object make-printable-object)
                `(lambda ()
                   (,%print ',object (,make-printable-object ,object)))))))))
```

# 2 AIMA-Vacuum

## 2.1 `aima-vacuum`

**Module** `aima-vacuum`

**Description** 'aima-vacuum' has agents and environments for chapter 2: Intelligent Agents.

**Exports**

- `agent-score`
- `agent-score-set!`
- `agent-location`
- `agent-location-set!`
- `agent-program`
- `agent-program-set!`
- `clean`
- `clean?`
- `copy-world`
- `cycle`
- `cycle?`
- `connect!`
- `default-n-nodes`
- `direction->move`
- `dirty`
- `dirty?`
- `display-world`
- `display-pdf`
- `down`
- `down?`

- left
- left?
- location-status
- location-status-set!
- location-neighbors
- location-neighbors-set!
- make-agent
- make-graph
- make-graph-world
- make-linear-world
- make-location
- make-node
- make-performance-measure
- make-preferential-depth-first-world
- make-randomized-graph-agent
- make-reflex-agent
- make-simple-reflex-agent
- make-stateful-reflex-agent
- make-stateful-graph-agent
- make-score-update!
- make-unknown-location
- make-world
- move->direction
- random-start
- reverse-move
- right
- right?
- simulate-graph
- simulate-graph/animation
- simulate-penalizing-vacuum
- simulate-vacuum
- unknown
- unknown?
- up
- up?

- world-location
- world-location-set!
- write-world-as-pdf
- write-world-as-dot
- write-world-as-gif