Scheme + Machine Learning

Panicz Maciej Godek

godek.maciek@gmail.com

https://github.com/panicz/writings/tree/master/talks/mlgdansk

ML Gdańsk, 24.04.2017

Agenda

- Lisp syntax
- genetic programming framework
- functional programming
- neural networks
- Emacs + Geiser
- humanoid robots

Agenda

- Lisp syntax
- genetic programming framework
- functional programming
- neural networks
- Emacs + Geiser
- humanoid robots

Agenda

- Lisp syntax
- genetic programming framework
- functional programming
- neural networks
- Emacs + Geiser
- humanoid robots

Agenda

- Lisp syntax
- genetic programming framework
- functional programming
- neural networks
- Emacs + Geiser
- humanoid robots

Agenda

- Lisp syntax
- genetic programming framework
- functional programming
- neural networks
- Emacs + Geiser
- humanoid robots

Agenda

- Lisp syntax
- genetic programming framework
- functional programming
- neural networks
- Emacs + Geiser
- humanoid robots

Agenda

- Lisp syntax
- genetic programming framework
- functional programming
- neural networks
- Emacs + Geiser
- humanoid robots

Lisp vs. convention

conventional	**Lisp**
`2 + 2`	`(+ 2 2)`
`f(X, Y)`	`(f X Y)`
`[1, 2, 3]`	`'(1 2 3)`
`[X, Y, Z]`	`` `(,X ,Y ,Z) ``
`[X, Y, Z]`	`(list X Y Z)`
`['X', 'Y', 'Z']?`	`'(X Y Z)`
`a = X; b = Y; ...`	`(let* ((a X) (b Y)) ...)`
`a, b = X, Y; ...`	`(let ((a X) (b Y)) ...)`

Lisp vs. convention

conventional	**Lisp**
`2 + 2`	`(+ 2 2)`
`f(X, Y)`	`(f X Y)`
`[1, 2, 3]`	`'(1 2 3)`
`[X, Y, Z]`	`` `(,X ,Y ,Z) ``
`[X, Y, Z]`	`(list X Y Z)`
`['X', 'Y', 'Z']?`	`'(X Y Z)`
`a = X; b = Y; ...`	`(let* ((a X) (b Y)) ...)`
`a, b = X, Y; ...`	`(let ((a X) (b Y)) ...)`

Lisp vs. convention

conventional	**Lisp**
`2 + 2`	`(+ 2 2)`
`f(X, Y)`	`(f X Y)`
`[1, 2, 3]`	`'(1 2 3)`
`[X, Y, Z]`	`` `(,X ,Y ,Z) ``
`[X, Y, Z]`	`(list X Y Z)`
`['X', 'Y', 'Z']?`	`'(X Y Z)`
`a = X; b = Y; ...`	`(let* ((a X) (b Y)) ...)`
`a, b = X, Y; ...`	`(let ((a X) (b Y)) ...)`

Lisp vs. convention

conventional	**Lisp**
`2 + 2`	`(+ 2 2)`
`f(X, Y)`	`(f X Y)`
`[1, 2, 3]`	`'(1 2 3)`
`[X, Y, Z]`	`` `(,X ,Y ,Z) ``
`[X, Y, Z]`	`(list X Y Z)`
`['X', 'Y', 'Z']?`	`'(X Y Z)`
`a = X; b = Y; ...`	`(let* ((a X) (b Y)) ...)`
`a, b = X, Y; ...`	`(let ((a X) (b Y)) ...)`

Lisp vs. convention

conventional	**Lisp**
`2 + 2`	`(+ 2 2)`
`f(X, Y)`	`(f X Y)`
`[1, 2, 3]`	`'(1 2 3)`
`[X, Y, Z]`	`` `(,X ,Y ,Z) ``
`[X, Y, Z]`	`(list X Y Z)`
`['X', 'Y', 'Z']?`	`'(X Y Z)`
`a = X; b = Y; ...`	`(let* ((a X) (b Y)) ...)`
`a, b = X, Y; ...`	`(let ((a X) (b Y)) ...)`

Lisp vs. convention

conventional	**Lisp**
`2 + 2`	`(+ 2 2)`
`f(X, Y)`	`(f X Y)`
`[1, 2, 3]`	`'(1 2 3)`
`[X, Y, Z]`	`` `(,X ,Y ,Z) ``
`[X, Y, Z]`	`(list X Y Z)`
`['X', 'Y', 'Z']?`	`'(X Y Z)`
`a = X; b = Y; ...`	`(let* ((a X) (b Y)) ...)`
`a, b = X, Y; ...`	`(let ((a X) (b Y)) ...)`

Lisp vs. convention

conventional	**Lisp**
`2 + 2`	`(+ 2 2)`
`f(X, Y)`	`(f X Y)`
`[1, 2, 3]`	`'(1 2 3)`
`[X, Y, Z]`	`` `(,X ,Y ,Z) ``
`[X, Y, Z]`	`(list X Y Z)`
`['X', 'Y', 'Z']?`	`'(X Y Z)`
`a = X; b = Y; ...`	`(let* ((a X) (b Y)) ...)`
`a, b = X, Y; ...`	`(let ((a X) (b Y)) ...)`

Lisp vs. convention

conventional	**Lisp**
`2 + 2`	`(+ 2 2)`
`f(X, Y)`	`(f X Y)`
`[1, 2, 3]`	`'(1 2 3)`
`[X, Y, Z]`	`` `(,X ,Y ,Z) ``
`[X, Y, Z]`	`(list X Y Z)`
`['X', 'Y', 'Z']?`	`'(X Y Z)`
`a = X; b = Y; ...`	`(let* ((a X) (b Y)) ...)`
`a, b = X, Y; ...`	`(let ((a X) (b Y)) ...)`

Lisp vs. convention

conventional	**Lisp**
`2 + 2`	`(+ 2 2)`
`f(X, Y)`	`(f X Y)`
`[1, 2, 3]`	`'(1 2 3)`
`[X, Y, Z]`	`` `(,X ,Y ,Z) ``
`[X, Y, Z]`	`(list X Y Z)`
`['X', 'Y', 'Z']?`	`'(X Y Z)`
`a = X; b = Y; ...`	`(let* ((a X) (b Y)) ...)`
`a, b = X, Y; ...`	`(let ((a X) (b Y)) ...)`

Crossing-over

```
(define (cross-over daddy mommy)
  (assert (= (length daddy) (length mommy)))
  (let* ((position (random (length daddy)))
         (sperm (take daddy position))
         (ovum (drop mommy position)))
    `(,@sperm ,@ovum)))

(cross-over '(a b c d e) '(v w x y z))
 ===> '(a b c y z)
```

Crossing-over

```
(define (cross-over daddy mommy)
  (assert (= (length daddy) (length mommy)))
  (let* ((position (random (length daddy)))
         (sperm (take daddy position))
         (ovum (drop mommy position)))
    `(,@sperm ,@ovum)))

(cross-over '(a b c d e) '(v w x y z))
 ===> '(a b c y z)
```

Crossing-over

```
(define (cross-over daddy mommy)
  (assert (= (length daddy) (length mommy)))
  (let* ((position (random (length daddy)))
         (sperm (take daddy position))
         (ovum (drop mommy position)))
    `(,@sperm ,@ovum)))

(cross-over '(a b c d e) '(v w x y z))
 ===> '(a b c y z)
```

Crossing-over

```
(define (cross-over daddy mommy)
  (assert (= (length daddy) (length mommy)))
  (let* ((position (random (length daddy)))
         (sperm (take daddy position))
         (ovum (drop mommy position)))
    `(,@sperm ,@ovum)))

(cross-over '(a b c d e) '(v w x y z))
 ===> '(a b c y z)
```

Crossing-over

```
(define (cross-over daddy mommy)
  (assert (= (length daddy) (length mommy)))
  (let* ((position (random (length daddy)))
         (sperm (take daddy position))
         (ovum (drop mommy position)))
    `(,@sperm ,@ovum)))

(cross-over '(a b c d e) '(v w x y z))
 ===> '(a b c y z)
```

Crossing-over

```
(define (cross-over daddy mommy)
  (assert (= (length daddy) (length mommy)))
  (let* ((position (random (length daddy)))
         (sperm (take daddy position))
         (ovum (drop mommy position)))
    `(,@sperm ,@ovum)))
```

```
(cross-over '(a b c d e) '(v w x y z))
 ===> '(a b c y z)
```

Crossing-over

```
(define (cross-over daddy mommy)
  (assert (= (length daddy) (length mommy)))
  (let* ((position (random (length daddy)))
         (sperm (take daddy position))
         (ovum (drop mommy position)))
    `(,@sperm ,@ovum)))

(cross-over '(a b c d e) '(v w x y z))
 ===> '(a b c y z)
```

Crossing-over

```
(define (cross-over daddy mommy)
  (assert (= (length daddy) (length mommy)))
  (let* ((position (random (length daddy)))
         (sperm (take daddy position))
         (ovum (drop mommy position)))
    `(,@sperm ,@ovum)))

(cross-over '(a b c d e) '(v w x y z))
 ===> '(a b c y z)
```

Social ceremony

```
(define (procreate population social-status)
   (let* ((census (map (lambda (specimen)
                         `(,(social-status specimen) . ,specimen))
                       population))
          (social-ladder (sort census
                               (lambda ((a . _) (b . _))
                                  (> a b))))
          (population (map (lambda ((status . specimen))
                             specimen)
                           social-ladder))
          (size (length population))
          (males (biased-random-indices size))
          (females (shuffle (biased-random-indices size)))
          (offspring (map (lambda (man woman)
                            (cross-over
                              (list-ref population man)
                              (list-ref population woman)))
                          males females)))
      (map (on-average-once-in size (mutate not)) offspring)))
```

Social ceremony

```
(define (procreate population social-status)
   (let* ((census (map (lambda (specimen)
                         `(,(social-status specimen) . ,specimen))
                       population))
          (social-ladder (sort census
                               (lambda ((a . _) (b . _))
                                 (> a b))))
          (population (map (lambda ((status . specimen))
                             specimen)
                           social-ladder))
          (size (length population))
          (males (biased-random-indices size))
          (females (shuffle (biased-random-indices size)))
          (offspring (map (lambda (man woman)
                            (cross-over
                              (list-ref population man)
                              (list-ref population woman)))
                          males females)))
     (map (on-average-once-in size (mutate not)) offspring)))
```

Social ceremony

```
(define (procreate population social-status)
   (let* ((census (map (lambda (specimen)
                         `(,(social-status specimen) . ,specimen))
                       population))
          (social-ladder (sort census
                               (lambda ((a . _) (b . _))
                                  (> a b))))
          (population (map (lambda ((status . specimen))
                             specimen)
                           social-ladder))
          (size (length population))
          (males (biased-random-indices size))
          (females (shuffle (biased-random-indices size)))
          (offspring (map (lambda (man woman)
                            (cross-over
                              (list-ref population man)
                              (list-ref population woman)))
                          males females)))
     (map (on-average-once-in size (mutate not)) offspring)))
```

Social ceremony

```
(define (procreate population social-status)
   (let* ((census (map (lambda (specimen)
                         `(,(social-status specimen) . ,specimen))
                       population))
          (social-ladder (sort census
                               (lambda ((a . _) (b . _))
                                  (> a b))))
          (population (map (lambda ((status . specimen))
                             specimen)
                           social-ladder))
          (size (length population))
          (males (biased-random-indices size))
          (females (shuffle (biased-random-indices size)))
          (offspring (map (lambda (man woman)
                            (cross-over
                              (list-ref population man)
                              (list-ref population woman)))
                          males females)))
     (map (on-average-once-in size (mutate not)) offspring)))
```

Social ceremony

```
(define (procreate population social-status)
   (let* ((census (map (lambda (specimen)
                         `(,(social-status specimen) . ,specimen))
                       population))
          (social-ladder (sort census
                               (lambda ((a . _) (b . _))
                                  (> a b))))
          (population (map (lambda ((status . specimen))
                             specimen)
                           social-ladder))
          (size (length population))
          (males (biased-random-indices size))
          (females (shuffle (biased-random-indices size)))
          (offspring (map (lambda (man woman)
                            (cross-over
                              (list-ref population man)
                              (list-ref population woman)))
                          males females)))

     (map (on-average-once-in size (mutate not)) offspring)))
```

Social ceremony

```
(define (procreate population social-status)
   (let* ((census (map (lambda (specimen)
                         `(,(social-status specimen) . ,specimen))
                       population))
          (social-ladder (sort census
                               (lambda ((a . _) (b . _))
                                  (> a b))))
          (population (map (lambda ((status . specimen))
                             specimen)
                           social-ladder))
          (size (length population))
          (males (biased-random-indices size))
          (females (shuffle (biased-random-indices size)))
          (offspring (map (lambda (man woman)
                            (cross-over
                              (list-ref population man)
                              (list-ref population woman)))
                          males females)))

     (map (on-average-once-in size (mutate not)) offspring)))
```

Social ceremony

```
(define (procreate population social-status)
   (let* ((census (map (lambda (specimen)
                         `(,(social-status specimen) . ,specimen))
                       population))
          (social-ladder (sort census
                               (lambda ((a . _) (b . _))
                                  (> a b))))
          (population (map (lambda ((status . specimen))
                             specimen)
                           social-ladder))
          (size (length population))
          (males (biased-random-indices size))
          (females (shuffle (biased-random-indices size)))
          (offspring (map (lambda (man woman)
                            (cross-over
                              (list-ref population man)
                              (list-ref population woman)))
                          males females)))
     (map (on-average-once-in size (mutate not)) offspring)))
```

Social ceremony

```
(define (procreate population social-status)
   (let* ((census (map (lambda (specimen)
                         `(,(social-status specimen) . ,specimen))
                       population))
          (social-ladder (sort census
                               (lambda ((a . _) (b . _))
                                  (> a b))))
          (population (map (lambda ((status . specimen))
                             specimen)
                           social-ladder))
          (size (length population))
          (males (biased-random-indices size))
          (females (shuffle (biased-random-indices size)))
          (offspring (map (lambda (man woman)
                            (cross-over
                              (list-ref population man)
                              (list-ref population woman)))
                          males females)))
     (map (on-average-once-in size (mutate not)) offspring)))
```

Social ceremony

```
(define (procreate population social-status)
   (let* ((census (map (lambda (specimen)
                         `(,(social-status specimen) . ,specimen))
                       population))
          (social-ladder (sort census
                               (lambda ((a . _) (b . _))
                                  (> a b))))
          (population (map (lambda ((status . specimen))
                             specimen)
                           social-ladder))
          (size (length population))
          (males (biased-random-indices size))
          (females (shuffle (biased-random-indices size)))
          (offspring (map (lambda (man woman)
                            (cross-over
                              (list-ref population man)
                              (list-ref population woman)))
                          males females)))
      (map (on-average-once-in size (mutate not)) offspring)))
```

Social ceremony

```
(define (procreate population social-status)
   (let* ((census (map (lambda (specimen)
                         `(,(social-status specimen) . ,specimen))
                       population))
          (social-ladder (sort census
                               (lambda ((a . _) (b . _))
                                  (> a b))))
          (population (map (lambda ((status . specimen))
                             specimen)
                           social-ladder))
          (size (length population))
          (males (biased-random-indices size))
          (females (shuffle (biased-random-indices size)))
          (offspring (map (lambda (man woman)
                            (cross-over
                              (list-ref population man)
                              (list-ref population woman)))
                          males females)))
     (map (on-average-once-in size (mutate not)) offspring)))
```

Lottery

```
(define (biased-random-indices size)
   (if (= size 0)
       '()
       `(,(random size)
         . ,(biased-random-indices (- size
                                      1)))))
```

Lottery

```
(define (biased-random-indices size)
   (if (= size 0)
      '()
      `(,(random size)
        . ,(biased-random-indices (- size
                                     1)))))
```

Lottery

```
(define (biased-random-indices size)
   (if (= size 0)
      '()
      `(,(random size)
        . ,(biased-random-indices (- size
                                     1)))))
```

Shuffling

```
(define (shuffle l)
  (match (length l)
    (0 '())
    (1 l)
    (n (let ((left right (split-at l
                                   (random n))))
         (if (= (random 2) 1)
           `(,@(shuffle right)
             ,@(shuffle left))
           `(,@(shuffle left)
             ,@(shuffle right)))))))
```

Shuffling

```
(define (shuffle l)
  (match (length l)
    (0 '())
    (1 l)
    (n (let ((left right (split-at l
                                   (random n))))
         (if (= (random 2) 1)
           `(,@(shuffle right)
             ,@(shuffle left))
           `(,@(shuffle left)
             ,@(shuffle right)))))))
```

Shuffling

```
(define (shuffle l)
  (match (length l)
    (0 '())
    (1 l)
    (n (let ((left right (split-at l
                                   (random n))))
         (if (= (random 2) 1)
           `(,@(shuffle right)
             ,@(shuffle left))
           `(,@(shuffle left)
             ,@(shuffle right)))))))
```

Shuffling

```
(define (shuffle l)
  (match (length l)
    (0 '())
    (1 l)
    (n (let ((left right (split-at l
                                   (random n))))
         (if (= (random 2) 1)
           `(,@(shuffle right)
             ,@(shuffle left))
           `(,@(shuffle left)
             ,@(shuffle right)))))))
```

Shuffling

```
(define (shuffle l)
  (match (length l)
    (0 '())
    (1 l)
    (n (let ((left right (split-at l
                                   (random n))))
         (if (= (random 2) 1)
           `(,@(shuffle right)
             ,@(shuffle left))
           `(,@(shuffle left)
             ,@(shuffle right)))))))
```

Shuffling

```
(define (shuffle l)
  (match (length l)
    (0 '())
    (1 l)
    (n (let ((left right (split-at l
                                   (random n))))
         (if (= (random 2) 1)
           `(,@(shuffle right)
             ,@(shuffle left))
           `(,@(shuffle left)
             ,@(shuffle right)))))))
```

Shuffling

```
(define (shuffle l)
  (match (length l)
    (0 '())
    (1 l)
    (n (let ((left right (split-at l
                                   (random n))))
         (if (= (random 2) 1)
           `(,@(shuffle right)
             ,@(shuffle left))
           `(,@(shuffle left)
             ,@(shuffle right)))))))
```

Mutations

```
(define ((on-average-once-in n action) arg)
  (assert (and (integer?  n) (> n 0)))
  (if (= (random n) 0)
      (action arg)
      arg))

(define ((mutate how) specimen)
  (let* ((n (random (length specimen)))
         (mutation (how (list-ref specimen
                                  n))))
    (alter #;element-number n #;in specimen
           #;with mutation)))
```

Mutations

```
(define ((on-average-once-in n action) arg)
  (assert (and (integer?  n) (> n 0)))
  (if (= (random n) 0)
      (action arg)
      arg))

(define ((mutate how) specimen)
  (let* ((n (random (length specimen)))
         (mutation (how (list-ref specimen
                                  n))))
    (alter #;element-number n #;in specimen
           #;with mutation)))
```

Mutations

```
(define ((on-average-once-in n action) arg)
  (assert (and (integer?  n) (> n 0)))
  (if (= (random n) 0)
      (action arg)
      arg))

(define ((mutate how) specimen)
  (let* ((n (random (length specimen)))
         (mutation (how (list-ref specimen
                                  n))))
    (alter #;element-number n #;in specimen
           #;with mutation)))
```

Mutations

```
(define ((on-average-once-in n action) arg)
  (assert (and (integer?  n) (> n 0)))
  (if (= (random n) 0)
      (action arg)
      arg))

(define ((mutate how) specimen)
  (let* ((n (random (length specimen)))
         (mutation (how (list-ref specimen
                                  n))))
    (alter #;element-number n #;in specimen
           #;with mutation)))
```

Mutations

```
(define ((on-average-once-in n action) arg)
  (assert (and (integer?  n) (> n 0)))
  (if (= (random n) 0)
      (action arg)
      arg))

(define ((mutate how) specimen)
  (let* ((n (random (length specimen)))
         (mutation (how (list-ref specimen
                                  n))))
    (alter #;element-number n #;in specimen
           #;with mutation)))
```

Mutations

```
(define ((on-average-once-in n action) arg)
  (assert (and (integer?  n) (> n 0)))
  (if (= (random n) 0)
      (action arg)
      arg))

(define ((mutate how) specimen)
  (let* ((n (random (length specimen)))
         (mutation (how (list-ref specimen
                                  n))))
    (alter #;element-number n #;in specimen
           #;with mutation)))
```

Mutations

```
(define ((on-average-once-in n action) arg)
  (assert (and (integer?  n) (> n 0)))
  (if (= (random n) 0)
      (action arg)
      arg))

(define ((mutate how) specimen)
  (let* ((n (random (length specimen)))
         (mutation (how (list-ref specimen
                                  n))))
    (alter #;element-number n #;in specimen
           #;with mutation)))
```

Mutations

```
(define ((on-average-once-in n action) arg)
  (assert (and (integer?  n) (> n 0)))
  (if (= (random n) 0)
      (action arg)
      arg))

(define ((mutate how) specimen)
  (let* ((n (random (length specimen)))
         (mutation (how (list-ref specimen
                                  n))))
    (alter #;element-number n #;in specimen
           #;with mutation)))
```

Natural selection

```
(define (evolve population #;towards criterion
                   #;for iterations)
  (assert (and (integer?  iterations)
               (>= iterations 0)))
  (if (<= iterations 0)
      population
      (evolve (procreate population criterion)
              #;towards criterion
              #;for (- iterations 1))))
```

Natural selection

```
(define (evolve population #;towards criterion
                   #;for iterations)
  (assert (and (integer?  iterations)
               (>= iterations 0)))
  (if (<= iterations 0)
      population
      (evolve (procreate population criterion)
              #;towards criterion
              #;for (- iterations 1))))
```

Natural selection

```
(define (evolve population #;towards criterion
                   #;for iterations)
  (assert (and (integer?  iterations)
               (>= iterations 0)))
  (if (<= iterations 0)
     population
      (evolve (procreate population criterion)
              #;towards criterion
              #;for (- iterations 1))))
```

Natural selection

```
(define (evolve population #;towards criterion
                   #;for iterations)
  (assert (and (integer?  iterations)
               (>= iterations 0)))
  (if (<= iterations 0)
     population
     (evolve (procreate population criterion)
             #;towards criterion
             #;for (- iterations 1))))
```

Making history

```
(define (optimize dimension population-size
                  iterations criterion)
  (let* ((population (generate-population
                      population-size
                      dimension))
         (modern-society (evolve population
                                 #;towards criterion
                                 #;for iterations)))
    (argmax criterion modern-society)))

(define (generate-population size dimension)
  (generate-list size
                 (lambda ()
                   (generate-specimen dimension))))
```

Making history

```
(define (optimize dimension population-size
                  iterations criterion)
  (let* ((population (generate-population
                      population-size
                      dimension))
         (modern-society (evolve population
                                 #;towards criterion
                                 #;for iterations)))
    (argmax criterion modern-society)))

(define (generate-population size dimension)
  (generate-list size
                 (lambda ()
                   (generate-specimen dimension))))
```

Making history

```
(define (optimize dimension population-size
                  iterations criterion)
  (let* ((population (generate-population
                      population-size
                      dimension))
         (modern-society (evolve population
                                 #;towards criterion
                                 #;for iterations)))
    (argmax criterion modern-society)))

(define (generate-population size dimension)
  (generate-list size
                 (lambda ()
                   (generate-specimen dimension))))
```

Making history

```
(define (optimize dimension population-size
                  iterations criterion)
  (let* ((population (generate-population
                      population-size
                      dimension))
         (modern-society (evolve population
                                 #;towards criterion
                                 #;for iterations)))
    (argmax criterion modern-society)))

(define (generate-population size dimension)
  (generate-list size
                 (lambda ()
                   (generate-specimen dimension))))
```

Making history

```
(define (optimize dimension population-size
                  iterations criterion)
  (let* ((population (generate-population
                      population-size
                      dimension))
         (modern-society (evolve population
                                 #;towards criterion
                                 #;for iterations)))
    (argmax criterion modern-society)))

(define (generate-population size dimension)
  (generate-list size
                 (lambda ()
                   (generate-specimen dimension))))
```

Making history

```
(define (optimize dimension population-size
                  iterations criterion)
  (let* ((population (generate-population
                      population-size
                      dimension))
         (modern-society (evolve population
                                 #;towards criterion
                                 #;for iterations)))
    (argmax criterion modern-society)))

(define (generate-population size dimension)
  (generate-list size
                 (lambda ()
                   (generate-specimen dimension))))
```

Genetics

```
(define (generate-specimen dimension)
  (generate-list dimension
      (lambda () (= (random 2) 0))))

(define (generate-list n generator)
  (assert (and (integer?  n) (>= n 0)))
  (if (= n 0)
    '()
    `(,(generator)
      .  ,(generate-list (- n 1) generator))))
```

Genetics

```
(define (generate-specimen dimension)
  (generate-list dimension
      (lambda () (= (random 2) 0))))

(define (generate-list n generator)
  (assert (and (integer?  n) (>= n 0)))
  (if (= n 0)
    '()
    `(,(generator)
      .  ,(generate-list (- n 1) generator))))
```

Genetics

```
(define (generate-specimen dimension)
  (generate-list dimension
      (lambda () (= (random 2) 0))))

(define (generate-list n generator)
  (assert (and (integer? n) (>= n 0)))
  (if (= n 0)
    '()
    `(,(generator)
      . ,(generate-list (- n 1) generator))))
```

Genetics

```
(define (generate-specimen dimension)
  (generate-list dimension
      (lambda () (= (random 2) 0))))

(define (generate-list n generator)
  (assert (and (integer?  n) (>= n 0)))
  (if (= n 0)
    '()
    `(,(generator)
      .  ,(generate-list (- n 1) generator))))
```

Genetics

```
(define (generate-specimen dimension)
  (generate-list dimension
      (lambda () (= (random 2) 0))))

(define (generate-list n generator)
  (assert (and (integer?  n) (>= n 0)))
  (if (= n 0)
    '()
    `(,(generator)
      .  ,(generate-list (- n 1) generator))))
```

Genetics

```
(define (generate-specimen dimension)
  (generate-list dimension
      (lambda () (= (random 2) 0))))

(define (generate-list n generator)
  (assert (and (integer?  n) (>= n 0)))
  (if (= n 0)
    '()
    `(,(generator)
      .  ,(generate-list (- n 1) generator))))
```

SAT problem

```
(define (satisfied?  formula #;under valuation)
  (match formula
    ((’and .  clauses)
     (every (lambda (clause)
              (satisfied?  clause #;under valuation))
            clauses))
    ((’or .  clauses)
     (any (lambda (clause)
            (satisfied?  clause #;under valuation))
          clauses))
    ((’not clause)
     (not (satisfied?  clause #;under valuation)))
    (_
     (assert (symbol?  formula))
     (lookup formula #;in valuation))))
```

SAT problem

```
(define (satisfied?  formula #;under valuation)
  (match formula
    ((’and .  clauses)
     (every (lambda (clause)
              (satisfied?  clause #;under valuation))
            clauses))
    ((’or .  clauses)
     (any (lambda (clause)
            (satisfied?  clause #;under valuation))
          clauses))
    ((’not clause)
     (not (satisfied?  clause #;under valuation)))
    (_
     (assert (symbol?  formula))
     (lookup formula #;in valuation))))
```

SAT problem

```
(define (satisfied?  formula #;under valuation)
  (match formula
    ((’and .  clauses)
     (every (lambda (clause)
              (satisfied?  clause #;under valuation))
            clauses))
    ((’or .  clauses)
     (any (lambda (clause)
            (satisfied?  clause #;under valuation))
          clauses))
    ((’not clause)
     (not (satisfied?  clause #;under valuation)))
    (_
     (assert (symbol?  formula))
     (lookup formula #;in valuation))))
```

SAT problem

```
(define (satisfied?  formula #;under valuation)
  (match formula
    ((’and .  clauses)
     (every (lambda (clause)
              (satisfied?  clause #;under valuation))
            clauses))
    ((’or .  clauses)
     (any (lambda (clause)
            (satisfied?  clause #;under valuation))
          clauses))
    ((’not clause)
     (not (satisfied?  clause #;under valuation)))
    (_
     (assert (symbol?  formula))
     (lookup formula #;in valuation))))
```

SAT problem

```
(define (satisfied?  formula #;under valuation)
  (match formula
    ((’and .  clauses)
     (every (lambda (clause)
              (satisfied?  clause #;under valuation))
            clauses))
    ((’or .  clauses)
     (any (lambda (clause)
            (satisfied?  clause #;under valuation))
          clauses))
    ((’not clause)
     (not (satisfied?  clause #;under valuation)))
    (_
     (assert (symbol?  formula))
     (lookup formula #;in valuation))))
```

SAT problem

```
(define (satisfied?  formula #;under valuation)
  (match formula
    ((’and .  clauses)
     (every (lambda (clause)
              (satisfied?  clause #;under valuation))
            clauses))
    ((’or .  clauses)
     (any (lambda (clause)
            (satisfied?  clause #;under valuation))
          clauses))
    ((’not clause)
     (not (satisfied?  clause #;under valuation)))
    (_
     (assert (symbol?  formula))
     (lookup formula #;in valuation))))
```

SAT problem

```
(define (satisfied?  formula #;under valuation)
  (match formula
    ((’and .  clauses)
     (every (lambda (clause)
              (satisfied?  clause #;under valuation))
            clauses))
    ((’or .  clauses)
     (any (lambda (clause)
            (satisfied?  clause #;under valuation))
          clauses))
    ((’not clause)
     (not (satisfied?  clause #;under valuation)))
    (_
     (assert (symbol?  formula))
     (lookup formula #;in valuation))))
```

SAT problem

```
(define (satisfied?  formula #;under valuation)
  (match formula
    ((’and .  clauses)
     (every (lambda (clause)
              (satisfied?  clause #;under valuation))
            clauses))
    ((’or .  clauses)
     (any (lambda (clause)
            (satisfied?  clause #;under valuation))
          clauses))
    ((’not clause)
     (not (satisfied?  clause #;under valuation)))
    (_
     (assert (symbol?  formula))
     (lookup formula #;in valuation))))
```

SAT problem

```
(define (satisfied?  formula #;under valuation)
  (match formula
    ((’and .  clauses)
     (every (lambda (clause)
              (satisfied?  clause #;under valuation))
            clauses))
    ((’or .  clauses)
     (any (lambda (clause)
            (satisfied?  clause #;under valuation))
          clauses))
    ((’not clause)
     (not (satisfied?  clause #;under valuation)))
    (_
     (assert (symbol?  formula))
     (lookup formula #;in valuation))))
```

SAT problem

```
(define (satisfied?  formula #;under valuation)
  (match formula
    ((’and .  clauses)
     (every (lambda (clause)
              (satisfied?  clause #;under valuation))
            clauses))
    ((’or .  clauses)
     (any (lambda (clause)
            (satisfied?  clause #;under valuation))
          clauses))
    ((’not clause)
     (not (satisfied?  clause #;under valuation)))
    (_
     (assert (symbol?  formula))
     (lookup formula #;in valuation))))
```

Dictionary lookup

```
(define (lookup key #;in mapping)
  (let* ((((name value) . remaining) mapping))
    (if (eq?  name key)
        value
        (lookup key remaining))))

(lookup 'y '((x 1) (y 2) (z 3)))
===> 2
```

Dictionary lookup

```
(define (lookup key #;in mapping)
  (let* ((((name value) . remaining) mapping))
    (if (eq? name key)
        value
        (lookup key remaining))))

(lookup 'y '((x 1) (y 2) (z 3)))
===> 2
```

Dictionary lookup

```
(define (lookup key #;in mapping)
  (let* ((((name value) . remaining) mapping))
    (if (eq?  name key)
        value
        (lookup key remaining))))

(lookup 'y '((x 1) (y 2) (z 3)))
===> 2
```

Dictionary lookup

```
(define (lookup key #;in mapping)
  (let* ((((name value) . remaining) mapping))
    (if (eq?  name key)
        value
        (lookup key remaining))))

(lookup 'y '((x 1) (y 2) (z 3)))
===> 2
```

Dictionary lookup

```
(define (lookup key #;in mapping)
  (let* ((((name value) . remaining) mapping))
    (if (eq?  name key)
        value
        (lookup key remaining))))

(lookup 'y '((x 1) (y 2) (z 3)))
===> 2
```

Dictionary lookup

```
(define (lookup key #;in mapping)
  (let* ((((name value) . remaining) mapping))
    (if (eq?  name key)
        value
        (lookup key remaining))))

(lookup 'y '((x 1) (y 2) (z 3)))
===> 2
```

SAT problem continued

```
(satisfied?  '(and p q)
             '((p #true) (q #true)))
(satisfied?  '(and p q)
             '((p #true) (q #false)))
(satisfied?  '(and p (not p)) '((p #true)))
(satisfied?  '(and p (not p)) '((p #false)))
(satisfied?  '(and (or x1 (not x3))
                   (or x2 x3 (not x1))) ???)
```

SAT problem continued

```
(satisfied?  '(and p q)
             '((p #true) (q #true)))
(satisfied?  '(and p q)
             '((p #true) (q #false)))
(satisfied?  '(and p (not p)) '((p #true)))
(satisfied?  '(and p (not p)) '((p #false)))
(satisfied?  '(and (or x1 (not x3))
                   (or x2 x3 (not x1))) ???)
```

SAT problem continued

```
(satisfied?  '(and p q)
             '((p #true) (q #true)))
(satisfied?  '(and p q)
             '((p #true) (q #false)))
(satisfied?  '(and p (not p)) '((p #true)))
(satisfied?  '(and p (not p)) '((p #false)))
(satisfied?  '(and (or x1 (not x3))
                   (or x2 x3 (not x1))) ???)
```

SAT problem continued

```
(satisfied?  '(and p q)
             '((p #true) (q #true)))
(satisfied?  '(and p q)
             '((p #true) (q #false)))
(satisfied?  '(and p (not p)) '((p #true)))
(satisfied?  '(and p (not p)) '((p #false)))
(satisfied?  '(and (or x1 (not x3))
                   (or x2 x3 (not x1))) ???)
```

SAT problem continued

```
(satisfied?  '(and p q)
             '((p #true) (q #true)))
(satisfied?  '(and p q)
             '((p #true) (q #false)))
(satisfied?  '(and p (not p)) '((p #true)))
(satisfied?  '(and p (not p)) '((p #false)))
(satisfied?  '(and (or x1 (not x3))
                   (or x2 x3 (not x1))) ???)
```

SAT problem continued

```
(satisfied?  '(and p q)
             '((p #true) (q #true)))
(satisfied?  '(and p q)
             '((p #true) (q #false)))
(satisfied?  '(and p (not p)) '((p #true)))
(satisfied?  '(and p (not p)) '((p #false)))
(satisfied?  '(and (or x1 (not x3))
                   (or x2 x3 (not x1))) ???)
```

Atomic formulas

```
(define (atomic-formulas proposition)
  (match proposition
    ((operator . clauses)
     (delete-duplicates (append-map
                         atomic-formulas
                         clauses)))
    (_
     (assert (symbol? proposition))
     `(,proposition))))

(atomic-formulas '(and (or x1 (not x3))
                       (or x2 x3 (not x1))))
===> (x1 x3 x2)
```

Atomic formulas

```
(define (atomic-formulas proposition)
  (match proposition
    ((operator . clauses)
     (delete-duplicates (append-map
                         atomic-formulas
                         clauses)))
    (_
     (assert (symbol? proposition))
     `(,proposition))))

(atomic-formulas '(and (or x1 (not x3))
                       (or x2 x3 (not x1))))
===> (x1 x3 x2)
```

Atomic formulas

```
(define (atomic-formulas proposition)
  (match proposition
    ((operator . clauses)
     (delete-duplicates (append-map
                         atomic-formulas
                         clauses)))
    (_
     (assert (symbol? proposition))
     `(,proposition))))

(atomic-formulas '(and (or x1 (not x3))
                       (or x2 x3 (not x1))))
===> (x1 x3 x2)
```

Atomic formulas

```
(define (atomic-formulas proposition)
  (match proposition
    ((operator . clauses)
     (delete-duplicates (append-map
                         atomic-formulas
                         clauses)))
    (_
     (assert (symbol? proposition))
     `(,proposition))))

(atomic-formulas '(and (or x1 (not x3))
                       (or x2 x3 (not x1))))
===> (x1 x3 x2)
```

Atomic formulas

```
(define (atomic-formulas proposition)
  (match proposition
    ((operator . clauses)
     (delete-duplicates (append-map
                         atomic-formulas
                         clauses)))
    (_
     (assert (symbol? proposition))
     `(,proposition))))

(atomic-formulas '(and (or x1 (not x3))
                       (or x2 x3 (not x1))))
===> (x1 x3 x2)
```

Atomic formulas

```
(define (atomic-formulas proposition)
  (match proposition
    ((operator .  clauses)
     (delete-duplicates (append-map
                         atomic-formulas
                         clauses)))
    (_
     (assert (symbol?  proposition))
     `(,proposition))))

(atomic-formulas '(and (or x1 (not x3))
                       (or x2 x3 (not x1))))
===> (x1 x3 x2)
```

Atomic formulas

```
(define (atomic-formulas proposition)
  (match proposition
    ((operator .  clauses)
     (delete-duplicates (append-map
                         atomic-formulas
                         clauses)))
    (_
     (assert (symbol?  proposition))
     `(,proposition))))

(atomic-formulas '(and (or x1 (not x3))
                       (or x2 x3 (not x1))))
===> (x1 x3 x2)
```

Atomic formulas

```
(define (atomic-formulas proposition)
  (match proposition
    ((operator .  clauses)
     (delete-duplicates (append-map
                         atomic-formulas
                         clauses)))
    (_
     (assert (symbol?  proposition))
     `(,proposition))))

(atomic-formulas '(and (or x1 (not x3))
                       (or x2 x3 (not x1))))
===> (x1 x3 x2)
```

Conjunctive Normal Form

```
(define (number-of-satisfied-subformulas
                    #;of cnf #;for chromosome)
  (let* ((variables (atomic-formulas cnf))
         (valuation (map list variables
                         chromosome))
         (('and .  or-clauses) cnf))
    (count (lambda (subformula)
             (satisfied? subformula
                         #;under valuation))
           or-clauses)))
```

Conjunctive Normal Form

```
(define (number-of-satisfied-subformulas
                    #;of cnf #;for chromosome)
  (let* ((variables (atomic-formulas cnf))
         (valuation (map list variables
                         chromosome))
         (('and . or-clauses) cnf))
    (count (lambda (subformula)
             (satisfied? subformula
                         #;under valuation))
           or-clauses)))
```

Conjunctive Normal Form

```
(define (number-of-satisfied-subformulas
                    #;of cnf #;for chromosome)
  (let* ((variables (atomic-formulas cnf))
         (valuation (map list variables
                                 chromosome))
         (('and . or-clauses) cnf))
    (count (lambda (subformula)
             (satisfied? subformula
                         #;under valuation))
           or-clauses)))
```

Conjunctive Normal Form

```
(define (number-of-satisfied-subformulas
                    #;of cnf #;for chromosome)
  (let* ((variables (atomic-formulas cnf))
         (valuation (map list variables
                                 chromosome))
         (('and .  or-clauses) cnf))
    (count (lambda (subformula)
             (satisfied? subformula
                         #;under valuation))
           or-clauses)))
```

Conjunctive Normal Form

```
(define (number-of-satisfied-subformulas
                    #;of cnf #;for chromosome)
  (let* ((variables (atomic-formulas cnf))
         (valuation (map list variables
                                 chromosome))
         (('and .  or-clauses) cnf))
    (count (lambda (subformula)
             (satisfied? subformula
                         #;under valuation))
           or-clauses)))
```

Applying genetic strategy

```
(define (solve-SAT formula/cnf population iterations)
  (let* ((dimension (length (atomic-formulas
                             formula/cnf)))
         (measure (lambda (chromosome)
                    (number-of-satisfied-subformulas
                     #;of formula
                     #;for chromosome))))
    (optimize dimension population iterations measure)))
```

Applying genetic strategy

```
(define (solve-SAT formula/cnf population iterations)
  (let* ((dimension (length (atomic-formulas
                                formula/cnf)))
         (measure (lambda (chromosome)
                    (number-of-satisfied-subformulas
                        #;of formula
                        #;for chromosome))))
    (optimize dimension population iterations measure)))
```

Applying genetic strategy

```
(define (solve-SAT formula/cnf population iterations)
  (let* ((dimension (length (atomic-formulas
                              formula/cnf)))
         (measure (lambda (chromosome)
                    (number-of-satisfied-subformulas
                        #;of formula
                        #;for chromosome))))
    (optimize dimension population iterations measure)))
```

Applying genetic strategy

```
(define (solve-SAT formula/cnf population iterations)
  (let* ((dimension (length (atomic-formulas
                              formula/cnf)))
         (measure (lambda (chromosome)
                    (number-of-satisfied-subformulas
                        #;of formula
                        #;for chromosome))))
    (optimize dimension population iterations measure)))
```

Matrix operations

```
(define (transpose M)
  (apply map list M))

(define (M*2 A B)
  (let ((B^T (transpose B)))
    (map (lambda (rA)
           (map (lambda (cB)
                  (sum (map * rA cB)))
                B^T))
         A)))

(define (M* M . MM)
  (fold-left M*2 M MM))
```

Matrix operations

```
(define (transpose M)
  (apply map list M))

(define (M*2 A B)
  (let ((B^T (transpose B)))
    (map (lambda (rA)
           (map (lambda (cB)
                  (sum (map * rA cB)))
                B^T))
         A)))

(define (M* M . MM)
  (fold-left M*2 M MM))
```

Matrix operations

```
(define (transpose M)
  (apply map list M))

(define (M*2 A B)
  (let ((B^T (transpose B)))
    (map (lambda (rA)
           (map (lambda (cB)
                  (sum (map * rA cB)))
                B^T))
         A)))

(define (M* M . MM)
  (fold-left M*2 M MM))
```

Matrix operations

```
(define (transpose M)
  (apply map list M))

(define (M*2 A B)
  (let ((B^T (transpose B)))
    (map (lambda (rA)
           (map (lambda (cB)
                  (sum (map * rA cB)))
                B^T))
         A)))

(define (M* M . MM)
  (fold-left M*2 M MM))
```

Matrix operations

```
(define (transpose M)
  (apply map list M))

(define (M*2 A B)
  (let ((B^T (transpose B)))
    (map (lambda (rA)
           (map (lambda (cB)
                  (sum (map * rA cB)))
                B^T))
         A)))

(define (M* M . MM)
  (fold-left M*2 M MM))
```

Matrix operations

```
(define (transpose M)
  (apply map list M))

(define (M*2 A B)
  (let ((B^T (transpose B)))
    (map (lambda (rA)
           (map (lambda (cB)
                  (sum (map * rA cB)))
                B^T))
         A)))

(define (M* M . MM)
  (fold-left M*2 M MM))
```

Matrix operations

```
(define (transpose M)
  (apply map list M))

(define (M*2 A B)
  (let ((B^T (transpose B)))
    (map (lambda (rA)
           (map (lambda (cB)
                  (sum (map * rA cB)))
                B^T))
         A)))

(define (M* M . MM)
  (fold-left M*2 M MM))
```

Neural network

```
(define ((neural-network . layers) input)
  (fold-left
    (lambda (feed (activate weights))
      (let* ((biased-input `((1 . ,feed)))
             ((output) (M* biased-input
                           weights)))
        (map activate output)))
    input
    layers))

(define (sigmoid x)
  (/ 1 (+ 1 (exp (- x)))))
```

Neural network

```
(define ((neural-network . layers) input)
  (fold-left
    (lambda (feed (activate weights))
      (let* ((biased-input `((1 . ,feed)))
             ((output) (M* biased-input
                           weights)))
        (map activate output)))
    input
    layers))

(define (sigmoid x)
  (/ 1 (+ 1 (exp (- x)))))
```

Neural network

```
(define ((neural-network . layers) input)
  (fold-left
    (lambda (feed (activate weights))
      (let* ((biased-input `((1 . ,feed)))
             ((output) (M* biased-input
                           weights)))
        (map activate output)))
    input
    layers))

(define (sigmoid x)
  (/ 1 (+ 1 (exp (- x)))))
```

Neural network

```
(define ((neural-network . layers) input)
  (fold-left
    (lambda (feed (activate weights))
        (let* ((biased-input `((1 . ,feed)))
               ((output) (M* biased-input
                             weights)))
          (map activate output)))
    input
    layers))

(define (sigmoid x)
  (/ 1 (+ 1 (exp (- x)))))
```

Neural network

```
(define ((neural-network . layers) input)
  (fold-left
    (lambda (feed (activate weights))
        (let* ((biased-input `((1 . ,feed)))
               ((output) (M* biased-input
                             weights)))
          (map activate output)))
    input
    layers))

(define (sigmoid x)
  (/ 1 (+ 1 (exp (- x)))))
```

Neural network

```
(define ((neural-network . layers) input)
  (fold-left
    (lambda (feed (activate weights))
        (let* ((biased-input `((1 . ,feed)))
               ((output) (M* biased-input
                             weights)))
          (map activate output)))
    input
    layers))

(define (sigmoid x)
  (/ 1 (+ 1 (exp (- x)))))
```

Neural network

```
(define ((neural-network . layers) input)
  (fold-left
    (lambda (feed (activate weights))
        (let* ((biased-input `((1 . ,feed)))
               ((output) (M* biased-input
                             weights)))
          (map activate output)))
    input
    layers))

(define (sigmoid x)
  (/ 1 (+ 1 (exp (- x)))))
```

Neural network

```
(map (neural-network `(,sigmoid
                       (( 0.80109 0.43529)
                        (-0.46122 0.78548)
                        ( 0.97314 2.10584)
                        (-0.39203 -0.57847)))
                    `(,identity
                      ((-0.23680)
                       (-0.81546)
                       ( 1.03775))))
     '((23 75 176)
       (25 67 180)
       (28 120 175)))
===> ((0.798) (0.801) (-0.014))
```

Neural network

```
(map (neural-network `(,sigmoid
                       (( 0.80109 0.43529)
                        (-0.46122 0.78548)
                        ( 0.97314 2.10584)
                        (-0.39203 -0.57847)))
                    `(,identity
                      ((-0.23680)
                       (-0.81546)
                       ( 1.03775))))
     '((23 75 176)
       (25 67 180)
       (28 120 175)))
===> ((0.798) (0.801) (-0.014))
```

Resources

A Pamphlet against R

`https://github.com/panicz/pamphlet/`

SLAYER framework

`https://bitbucket.org/panicz/slayer`

Resources

A Pamphlet against R

`https://github.com/panicz/pamphlet/`

SLAYER framework

`https://bitbucket.org/panicz/slayer`