The Egison Programming Language

Vol.01 Mar/28/2014

Satoshi Egi
Rakuten Institute of Technology
http://rit.rakuten.co.jp/

楽天 Rakuten

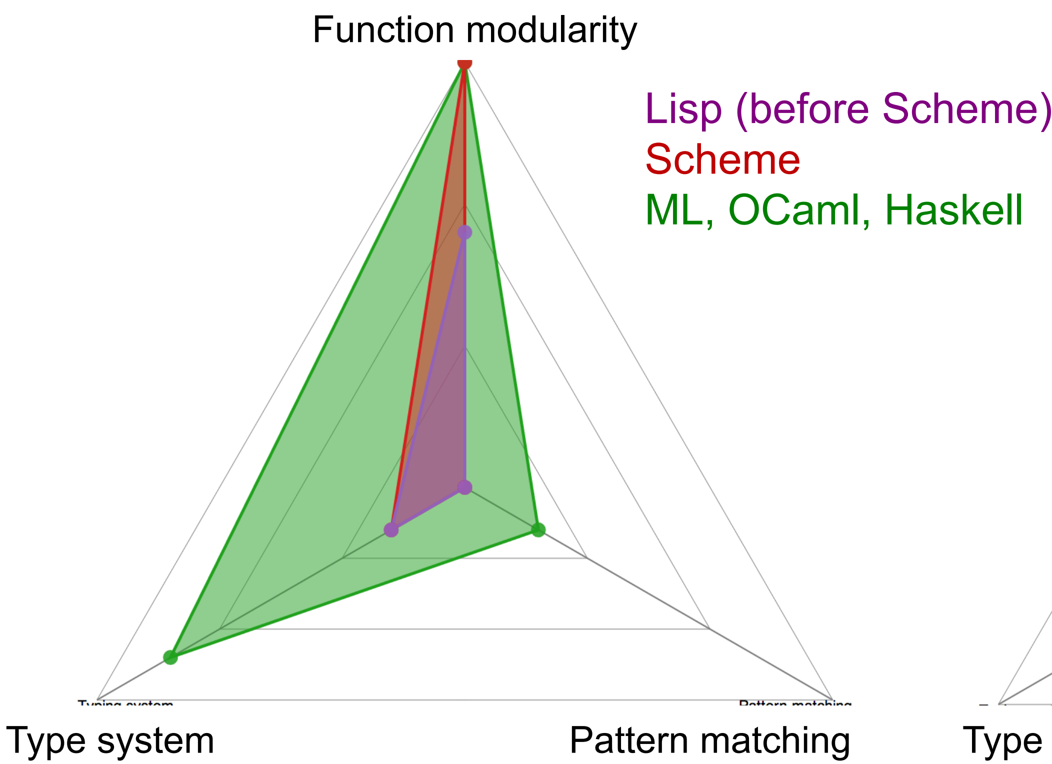
Profile of Egison

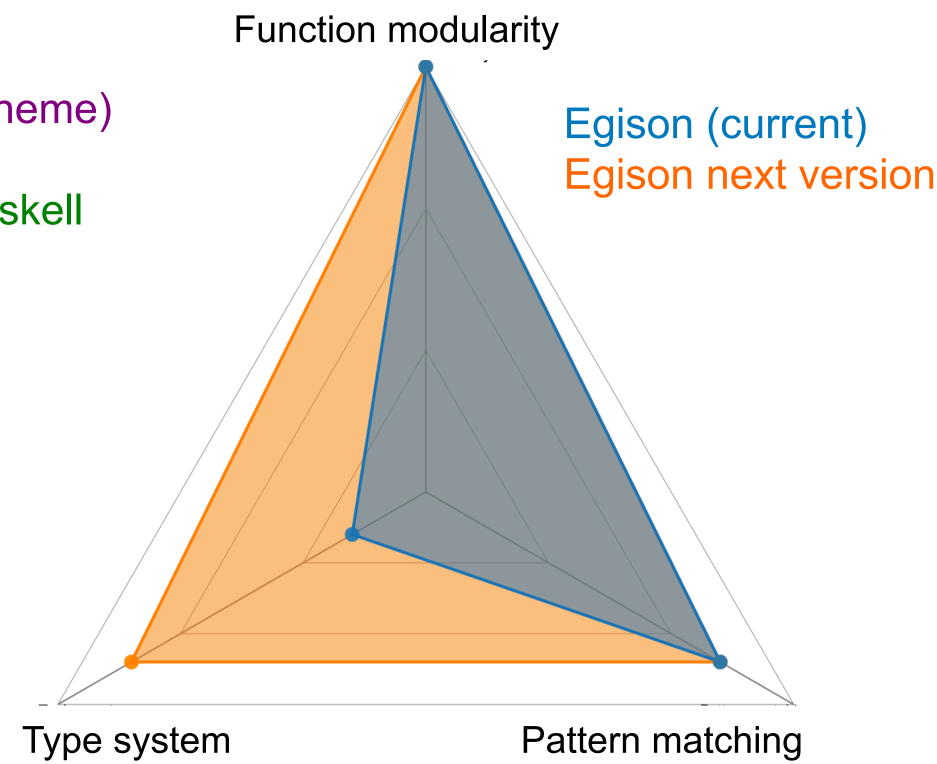
Egison is the programming language I’ve created.

Paradigm	**Pattern-matching-oriented**, Pure functional
Author	Satoshi Egi
License	MIT
Version	3.3.4 (2014/03/28)
First Released	2011/5/24
Filename Extension	.egi
Implemented in	Haskell (about 3,400 lines)

Motivation

I'd like to create a programming language that directly represents **human's intuition**.





Egison in one minute

Egison is the world's first programming language that realized **non-linear pattern-matching with backtracking**.

```
pairs = []
(1..n).each do |i|
  (i..n).each do |j|
    if xs[i] == xs[j]
      pair = xs[i]
    end
  end
end
```

Ruby

Enumerate the elements of the collection 'xs' that appear more than twice

```
(match-all xs (multiset integer)
  [<cons $x <cons ,x _>> x])
```

Egison

Quick Tour

The ‘match-all’ expression

Target

Matcher

```
(match-all {1 1 2 3 2} (multiset integer)
  [<cons $x <cons ,x _>> x])
```

Pattern

Expression executed when pattern matching succeed

```
;=>{1 1 2 2}
```

Result

Meaning: Pattern match against the “target” as the “matcher” with the “pattern” and return all results of pattern matching.

The ‘match-all’ expression

Target

```
(match-all {1 1 2 3 2} (multiset integer)
  [<cons $x <cons ,x _>> x])

;=>{1 1 2 2}
```

Pattern-match against the target data {1 1 2 3 2}

The ‘match-all’ expression

Target

Matcher

```
(match-all {1 1 2 3 2} (multiset integer)
  [<cons $x <cons ,x _>> x])

;=>{1 1 2 2}
```

Pattern-match against the target data {1 1 2 3 2} as the multiset of integers

The ‘match-all’ expression

Target

Matcher

```
(match-all {1 1 2 3 2} (multiset integer)
  [<cons $x <cons ,x _>> x])
```

Pattern

```
;=>{1 1 2 2}
```

Pattern-match against the target data {1 1 2 3 2} as the multiset of integers with the pattern <cons $x <cons ,x _>>

The ‘match-all’ expression

Target

Matcher

```
(match-all {1 1 2 3 2} (multiset integer)
  [<cons $x <cons ,x _>> x])
```

Pattern

Expression executed when pattern matching succeed

```
;=>{1 1 2 2}
```

Pattern-match against the target data {1 1 2 3 2} as the multiset of integers with the pattern <cons $x <cons ,x _>> and return the value bound to x.

The ‘match-all’ expression

Target

Matcher

```
(match-all {1 1 2 3 2} (multiset integer)
  [<cons $x <cons ,x _>> x])
```

Pattern

Expression executed when pattern matching succeed

```
;=>{1 1 2 2}
```

Result

Pattern-match against the target data {1 1 2 3 2} as the multiset of integers with the pattern <cons $x <cons ,x _>> and return the value bound to x.

The ‘cons’ pattern constructor

Divide a collection into an element and a collection of rest of elements.

```
(match-all {1 2 3} (list integer)
  [<cons $x $xs> [x xs]])
;=>{[1 {2 3}]}

(match-all {1 2 3} (multiset integer)
  [<cons $x $xs> [x xs]])
;=>{[1 {2 3}] [2 {1 3}] [3 {1 2}]}

(match-all {1 2 3} (set integer)
  [<cons $x $xs> [x xs]])
;=>{[1 {1 2 3}] [2 {1 2 3}] [3 {1 2 3}]}
```

The ‘cons’ pattern constructor

Divide a collection into an element and a collection of rest of elements.

```
(match-all {1 2 3} (list integer)
  [<cons $x $xs> [x xs]])
;=>{[1 {2 3}]}

(match-all {1 2 3} (multiset integer)
  [<cons $x $xs> [x xs]])
;=>{[1 {2 3}] [2 {1 3}] [3 {1 2}]}

(match-all {1 2 3} (set integer)
  [<cons $x $xs> [x xs]])
;=>{[1 {1 2 3}] [2 {1 2 3}] [3 {1 2 3}]}
```

The meaning of ‘cons’ changes for each matcher

The nested ‘cons’ pattern constructor

Extracting two elements using the ‘cons’ patterns.

```
(match-all {1 2 3} (list integer)
  [<cons $x <cons $y _>> [x y]])
;=>{[1 2]}

(match-all {1 2 3} (multiset integer)
  [<cons $x <cons $y _>> [x y]])
;=>{[1 2] [1 3] [2 1] [2 3] [3 1] [3 2]}

(match-all {1 2 3} (set integer)
  [<cons $x <cons $y _>> [x y]])
;=>{[1 1] [1 2] [2 1] [1 3] [2 2] [3 1] [2 3] [3 2] [3 3]}
```

Non-linear patterns

We can deal with the multiple occurrences of the same variables in a pattern.

```
(match-all {1 1 2 3 2} (list integer)
  [<cons $x <cons ,x _>> x])
;=>{1}
```

Two same head elements

```
(match-all {1 1 2 3 2} (multiset integer)
  [<cons $x <cons ,x _>> x])
;=>{1 1 2 2}
```

Elements that appear twice

Not-patterns

Patterns that match if the pattern does not match.

```
(match-all {1 1 2 3 2} (multiset integer)
  [<cons $x <cons ,x _>> x])
;=>{1 1 2 2}
```

One more ‘x’

```
(match-all {1 1 2 3 2} (multiset integer)
  [<cons $x ^<cons ,x _>> x])
;=>{3}
```

No more ‘x’

The 'join' pattern constructor

Divide a collection into two collections.

```
(match-all {1 2 3} (list integer)
  [<join $xs $ys> [xs ys]])
;=>{[{} {1 2 3}] [{1} {2 3}] [{1 2} {3}] [{1 2 3} {}]}

(match-all {1 2 3} (multiset integer)
  [<join $xs $ys> [xs ys]])
;=>{[{} {1 2 3}] [{1} {2 3}] [{2} {1 3}] [{3} {1 2}]
[{1 2} {3}] [{1 3} {2}] [{2 3} {1}] [{1 2 3} {}]}

(match-all {1 2 3} (set integer)
  [<join $xs $ys> [xs ys]])
;=>{[{} {1 2 3}] [{1} {1 2 3}] [{2} {1 2 3}] [{3} {1 2
3}] [{1 2} {1 2 3}] [{1 3} {1 2 3}] [{2 1} {1 2 3}]
[{2 3} {1 2 3}] [{3 1} {1 2 3}] [{1 2 3} {1 2 3}] [{3
2} {1 2 3}] [{1 3 2} {1 2 3}] [{2 1 3} {1 2 3}] [{2 3
1} {1 2 3}] [{3 1 2} {1 2 3}] [{3 2 1} {1 2 3}]}
```

Playing with the ‘join’ pattern constructor

Enumerate all two combination of elements.

```
(match-all {1 2 3 4 5} (list integer)
  [<join _ <cons $x <join _ <cons $y _>>>>
   [x y]])
;=>{[1 2] [1 3] [2 3] [1 4] [2 4] [3 4] [1 5]
[2 5] [3 5] [4 5]}
```

Demonstrations

The first application of Egison

```
(define $poker-hands
  (lambda [$cs]
    (match cs (multiset card)
      {[<cons <card $s $n>
         <cons <card ,s ,(- n 1)>
          <cons <card ,s ,(- n 2)>
           <cons <card ,s ,(- n 3)>
            <cons <card ,s ,(- n 4)>
             <nil>>>>>>
        <Straight-Flush>]
       [<cons <card _ $n>
         <cons <card _ ,n>
          <cons <card _ ,n>
            <cons <card _ ,n>
              <cons _
                <nil>>>>>>
        <Four-of-Kind>]
       [<cons <card _ $m>
         <cons <card _ ,m>
          <cons <card _ ,m>
           <cons <card _ $n>
            <cons <card _ ,n>
              <nil>>>>>>
        <Full-House>]
       [<cons <card $s _>
         <cons <card ,s _>
           <cons <card ,s _>
             <cons <card ,s _>
               <cons <card ,s _>
                 <nil>>>>>>
        <Flush>]
       [<cons <card _ $n>
         <cons <card _ ,(- n 1)>
          <cons <card _ ,(- n 2)>
           <cons <card _ ,(- n 3)>
            <cons <card _ ,(- n 4)>
             <nil>>>>>>
        <Straight>]
       [<cons <card _ $n>
         <cons <card _ ,n>
          <cons <card _ ,n>
           <cons _
            <cons _
             <nil>>>>>>
        <Three-of-Kind>]
       [<cons <card _ $m>
         <cons <card _ ,m>
          <cons <card _ $n>
            <cons <card _ ,n>
              <cons _
                <nil>>>>>>
        <Two-Pair>]
       [<cons <card _ $n>
         <cons <card _ ,n>
          <cons _
           <cons _
            <cons _
             <nil>>>>>>
        <One-Pair>]
       [<cons _
         <cons _
          <cons _
           <cons _
            <cons _
             <nil>>>>>>
        <Nothing>]})))
```

The first application of Egison

```
(define $poker-hands
  (lambda [$cs]
    (match cs (multiset card)
      {[<cons <card $s $n>
         <cons <card ,s ,(- n 1)>
          <cons <card ,s ,(- n 2)>
           <cons <card ,s ,(- n 3)>
            <cons <card ,s ,(- n 4)>
             <nil>>>>>>
        <Straight-Flush>]
       [<cons <card _ $n>
         <cons <card _ ,n>
          <cons <card _ ,n>
            <cons <card _ ,n>
              <cons _
                <nil>>>>>>
        <Four-of-Kind>]
       [<cons <card _ $m>
         <cons <card _ ,m>
          <cons <card _ ,m>
           <cons <card _ $n>
            <cons <card _ ,n>
              <nil>>>>>>
        <Full-House>]
       [<cons <card $s _>
         <cons <card ,s _>
           <cons <card ,s _>
             <cons <card ,s _>
               <cons <card ,s _>
                 <nil>>>>>>
        <Flush>]
       [<cons <card _ $n>
         <cons <card _ ,(- n 1)>
          <cons <card _ ,(- n 2)>
           <cons <card _ ,(- n 3)>
            <cons <card _ ,(- n 4)>
             <nil>>>>>>
        <Straight>]
       [<cons <card _ $n>
         <cons <card _ ,n>
          <cons <card _ ,n>
           <cons _
            <cons _
             <nil>>>>>>
        <Three-of-Kind>]
       [<cons <card _ $m>
         <cons <card _ ,m>
          <cons <card _ $n>
            <cons <card _ ,n>
              <cons _
                <nil>>>>>>
        <Two-Pair>]
       [<cons <card _ $n>
         <cons <card _ ,n>
          <cons _
           <cons _
            <cons _
             <nil>>>>>>
        <One-Pair>]
       [<cons _
         <cons _
          <cons _
           <cons _
            <cons _
             <nil>>>>>>
        <Nothing>]})))
```

Match as a set of cards

The first application of Egison

```
(define $poker-hands
  (lambda [$cs]
    (match cs (multiset card)
      {[<cons <card $s $n>
         <cons <card ,s ,(- n 1)>
          <cons <card ,s ,(- n 2)>
           <cons <card ,s ,(- n 3)>
            <cons <card ,s ,(- n 4)>
             <nil>>>>>>
        <Straight-Flush>]
       [<cons <card _ $n>
         <cons <card _ ,n>
          <cons <card _ ,n>
            <cons <card _ ,n>
              <cons _
                <nil>>>>>>
        <Four-of-Kind>]
       [<cons <card _ $m>
         <cons <card _ ,m>
          <cons <card _ ,m>
           <cons <card _ $n>
            <cons <card _ ,n>
              <nil>>>>>>
        <Full-House>]
       [<cons <card $s _>
         <cons <card ,s _>
           <cons <card ,s _>
             <cons <card ,s _>
               <cons <card ,s _>
                 <nil>>>>>>
        <Flush>]
       [<cons <card _ $n>
         <cons <card _ ,(- n 1)>
          <cons <card _ ,(- n 2)>
           <cons <card _ ,(- n 3)>
            <cons <card _ ,(- n 4)>
             <nil>>>>>>
        <Straight>]
       [<cons <card _ $n>
         <cons <card _ ,n>
          <cons <card _ ,n>
           <cons _
            <cons _
             <nil>>>>>>
        <Three-of-Kind>]
       [<cons <card _ $m>
         <cons <card _ ,m>
          <cons <card _ $n>
           <cons <card _ ,n>
        <Tw
       [<co
         <c
          <
            <cons _
             <nil>>>>>>
        <One-Pair>]
       [<cons _
         <cons _
          <cons _
           <cons _
            <cons _
             <nil>>>>>>
        <Nothing>]})))
```

Pattern for straight flash

The pattern for straight flush


```
(define $poker-hands
  (lambda [$cs]
    (match cs (multiset card)
      {[<cons <card $s $n>
         <cons <card ,s ,(- n 1)>
          <cons <card ,s ,(- n 2)>
           <cons <card ,s ,(- n 3)>
            <cons <card ,s ,(- n 4)>
             <nil>>>>>>
        <Straight-Flush>]
       [<cons <card _ $n>
         <cons <card _ ,n>
```

The pattern for straight flush

```
(define $poker-hands
  (lambda [$cs]
    (match cs (multiset card)
      {[<cons <card $s $n>
         <cons <card ,s ,(- n 1)>
          <cons <card ,s ,(- n 2)>
           <cons <card ,s ,(- n 3)>
            <cons <card ,s ,(- n 4)>
             <nil>>>>>>
        <Straight-Flush>]
       [<cons <card _ $n>
         <cons <card _ ,n>
```

Same suit with $s

The pattern for straight flush


```
(define $poker-hands
  (lambda [$cs]
    (match cs (multiset card)
      {[<cons <card $s $n>
         <cons <card ,s ,(- n 1)>
          <cons <card ,s ,(- n 2)>
           <cons <card ,s ,(- n 3)>
            <cons <card ,s ,(- n 4)>
             <nil>>>>>>
        <Straight-Flush>]
       [<cons <card _ $n>
         <cons <card _ ,n>
```

Numbers are serial from $n

Same suit with $s

The pattern for straight flush


```
(define $poker-hands
  (lambda [$cs]
    (match cs (multiset card)
      {[<cons <card $s $n>
         <cons <card ,s ,(- n 1)>
          <cons <card ,s ,(- n 2)>
           <cons <card ,s ,(- n 3)>
            <cons <card ,s ,(- n 4)>
             <nil>>>>>>
        <Straight-Flush>]
       [<cons <card _ $n>
         <cons <card _ ,n>
```

Numbers are serial from $n

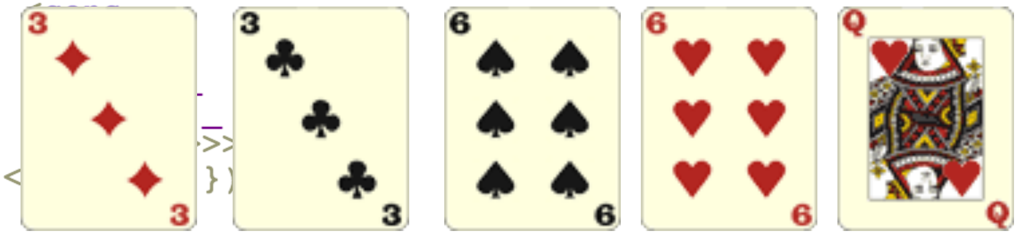
Same suit with $s

We can write any expression after ‘,’

The first application of Egison

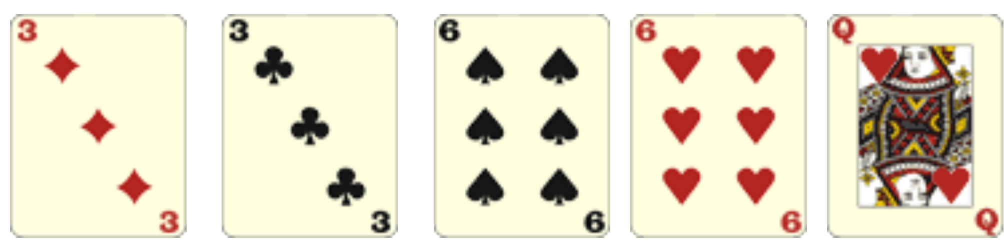
```
(define $poker-hands
  (lambda [$cs]
    (match cs (multiset card)
      {[<cons <card $s $n>
         <cons <card ,s ,(- n 1)>
          <cons <card ,s ,(- n 2)>
           <cons <card ,s ,(- n 3)>
            <cons <card ,s ,(- n 4)>
             <nil>>>>>>
        <Straight-Flush>]
       [<cons <card _ $n>
         <cons <card _ ,n>
          <cons <card _ ,n>
            <cons <card _ ,n>
              <cons _
                <nil>>>>>>
        <Four-of-Kind>]
       [<cons <card _ $m>
         <cons <card _ ,m>
          <cons <card _ ,m>
           <cons <card _ $n>
            <cons <card _ ,n>
              <nil>>>>>>
        <Full-House>]
       [<cons <card $s _>
         <cons <card ,s _>
           <cons <card ,s _>
             <cons <card ,s _>
               <cons <card ,s _>
                 <nil>>>>>>
        <Flush>]
       [<cons <card _ $n>
         <cons <card _ ,(- n 1)>
          <cons <card _ ,(- n 2)>
           <cons <card _ ,(- n 3)>
            <cons <card _ ,(- n 4)>
             <nil>>>>>>
        <Straight>]
       [<cons <card _ $n>
         <cons <card _ ,n>
          <cons <card _ ,n>
           <cons _
            <cons _
             <nil>>>>>>
        <Three-of-Kind>]
       [<cons <card _ $m>
         <cons <card _ ,m>
          <cons <card _ $n>
            <cons <card _ ,n>
              <cons _
                <nil>>>>>>
        <Two-Pair>]
       [<cons <card _ $n>
         <cons <card _ ,n>
          <cons _
           <cons _
            <cons _
             <nil>>>>>>
        <One-Pair>]
       [<cons _
```

Pattern for two pair



The pattern for two pair

```
 <Three-of-Kind>]
[<cons <card _ $m>
  <cons <card _ ,m>
   <cons <card _ $n>
    <cons <card _ ,n>
     <cons _
       <nil>>>>>>
 <Two-Pair>]
[<cons <card _ $n>
```



The pattern for two pair

```
 <Three-of-Kind>]
[<cons <card _ $m>
  <cons <card _ ,m>
   <cons <card _ $n>
    <cons <card _ ,n>
     <cons _
      <nil>>>>>>
 <Two-Pair>]
[<cons <card   $n>
```

Matches with any suit

Matches with any card

The pattern for two pair

```
<Three-of-Kind>]
[<cons <card _ $m>
  <cons <card _ ,m>
    <cons <card _ $n>
      <cons <card _ ,n>
        <cons _
          <nil>>>>>>
<Two-Pair>]
[<cons <card   $n>
```

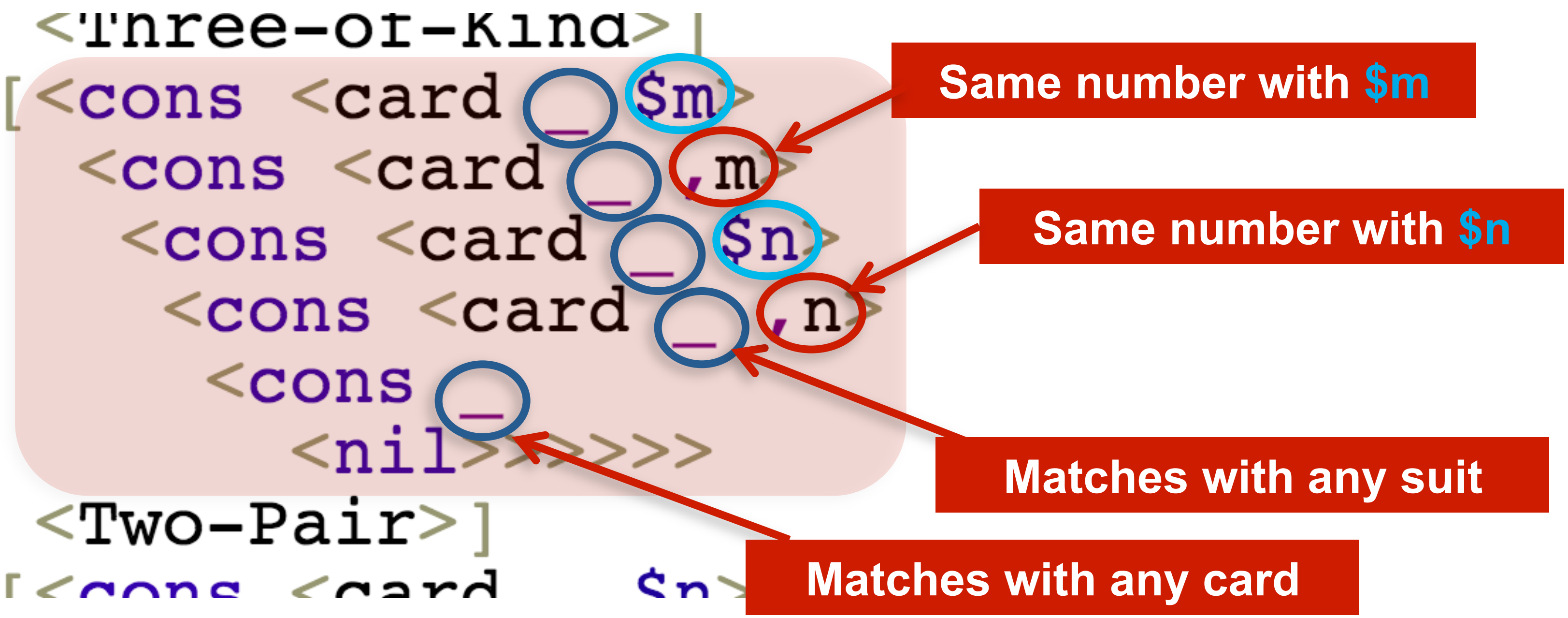
Same number with $m

Same number with $n

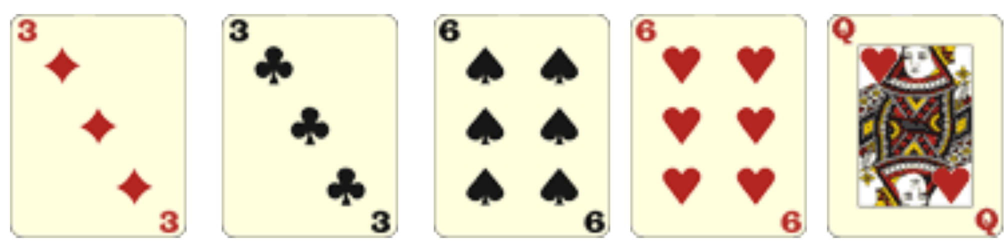
Matches with any suit

Matches with any card

The pattern for two pair



Non-linear patterns have very strong power



The first application of Egison

```
(define $poker-hands
  (lambda [$cs]
    (match cs (multiset card)
      {[<cons <card $s $n>
        <cons <card ,s ,(- n 1)>
         <cons <card ,s ,(- n 2)>
          <cons <card ,s ,(- n 3)>
           <cons <card ,s ,(- n 4)>
            <nil>>>>>>
        <Straight-Flush>]
       [<cons <card _ $n>
        <cons <card _ ,n>
         <cons <card _ ,n>
           <cons <card _ ,n>
             <cons _
               <nil>>>>>>
        <Four-of-Kind>]
       [<cons <card _ $m>
        <cons <card _ ,m>
         <cons <card _ ,m>
          <cons <card _ $n>
           <cons <card _ ,n>
             <nil>>>>>>
        <Full-House>]
       [<cons <card $s _>
        <cons <card ,s _>
          <cons <card ,s _>
            <cons <card ,s _>
              <cons <card ,s _>
                <nil>>>>>>
        <Flush>]
       [<cons <card _ $n>
        <cons <card _ ,(- n 1)>
         <cons <card _ ,(- n 2)>
          <cons <card _ ,(- n 3)>
           <cons <card _ ,(- n 4)>
            <nil>>>>>>
        <Straight>]
       [<cons <card _ $n>
         <cons <card _ ,n>
          <cons <card _ ,n>
           <cons _
            <cons _
             <nil>>>>>>
        <Three-of-Kind>]
       [<cons <card _ $m>
         <cons <card _ ,m>
          <cons <card _ $n>
            <cons <card _ ,n>
              <cons _
                <nil>>>>>>
        <Two-Pair>]
       [<cons <card _ $n>
         <cons <card _ ,n>
          <cons _
           <cons _
            <cons _
             <nil>>>>>>
        <One-Pair>]
       [<cons _
         <cons _
          <cons _
           <cons _
            <cons _
             <nil>>>>>>
        <Nothing>]})))
```

Non-linear patterns enables to represent all hands in a single pattern

The first application of Egison

```
(define $poker-hands
  (lambda [$cs]
    (match cs (multiset card)
      {[<cons <card $s $n>
         <cons <card ,s ,(- n 1)>
          <cons <card ,s ,(- n 2)>
           <cons <card ,s ,(- n 3)>
            <cons <card ,s ,(- n 4)>
             <nil>>>>>>
        <Straight-Flush>]
       [<cons <card _ $n>
         <cons <card _ ,n>
          <cons <card _ ,n>
            <cons <card _ ,n>
              <cons _
                <nil>>>>>>
        <Four-of-Kind>]
       [<cons <card _ $m>
         <cons <card _ ,m>
          <cons <card _ ,m>
           <cons <card _ $n>
            <cons <card _ ,n>
              <nil>>>>>>
        <Full-House>]
       [<cons <card $s _>
         <cons <card ,s _>
           <cons <card ,s _>
             <cons <card ,s _>
               <cons <card ,s _>
                 <nil>>>>>>
        <Flush>]
       [<cons <card _ $n>
         <cons <card _ ,(- n 1)>
          <cons <card _ ,(- n 2)>
           <cons <card _ ,(- n 3)>
            <cons <card _ ,(- n 4)>
             <nil>>>>>>
        <Straight>]
       [<cons <card _ $n>
         <cons <card _ ,n>
          <cons <card _ ,n>
           <cons _
            <cons _
             <nil>>>>>>
        <Three-of-Kind>]
       [<cons <card _ $m>
         <cons <card _ ,m>
          <cons <card _ $n>
            <cons <card _ ,n>
              <cons _
                <nil>>>>>>
        <Two-Pair>]
       [<cons <card _ $n>
         <cons <card _ ,n>
          <cons _
           <cons _
            <cons _
             <nil>>>>>>
        <One-Pair
       [<cons _
         <cons _
          <cons _
           <cons _
            <cons _
             <nil
        <Nothing>]})))
```

Egisonists can write this code in 2 minutes!

Non-linear patterns enables to represent all hands in a single pattern

Java version

Just finding a pair of cards is already complex.

```
public static boolean hasPair(Card[] cards) {
  for (int i = 0; i <= 4 ;i++) {
    for (int j = i + 1 ; j <= 4 ; j++) {
     if (cards[i] == (cards[j])) {
       return true;
     }
     return false;
    }
  }
}
```

I found a poker-hand evaluator in Java more than 200 lines of code.

http://www.codeproject.com/Articles/38821/Make-a-poker-hand-evalutator-in-Java

More complex example, Mahjong

```
;;
;; Pattern modularizations
;;
(define $twin
  (pattern-function [$pat1 $pat2]
    <cons (& $pat pat1)
     <cons ,pat
      pat2>>))

(define $shuntsu
  (pattern-function [$pat1 $pat2]
    <cons (& <num $s $n> pat1)
     <cons <num ,s ,(+ n 1)>
      <cons <num ,s ,(+ n 2)>
       pat2>>>))

(define $kohtsu
  (pattern-function [$pat1 $pat2]
    <cons (& $pat pat1)
     <cons ,pat
      <cons ,pat
       pat2>>>))

;;
;; A function that determins whether the hand is finished or not.
;;
(define $agari?
  (match-lambda (multiset hai)
    {[(twin $th_1
         (| (shuntsu $sh_1 (| (shuntsu $sh_2 (| (shuntsu $sh_3 (| (shuntsu $sh_4 <nil>)
                                                                    (kohtsu $kh_1 <nil>)))
                                                  (kohtsu $kh_1 (kohtsu $kh_2 <nil>))))
                              (kohtsu $kh_1 (kohtsu $kh_2 (kohtsu $kh_3 <nil>)))))
            (kohtsu $kh_1 (kohtsu $kh_2 (kohtsu $kh_3 (kohtsu $kh_4 <nil>)))))
         (twin $th_2 (twin $th_3 (twin $th_4 (twin $th_5 (twin $th_6 (twin $th_7 <nil>)))))))
       #t]
      [_ #f]}))
```

楽天 Rakuten

More complex example, Mahjong

```
;;
;; Pattern modularizations
;;
(define $twin
  (pattern-function [$pat1 $pat2]
    <cons (& $pat pat1)
     <cons ,pat
      pat2>>))
```

Two same tiles


```
(define $shuntsu
  (pattern-function [$pat1 $pat2]
    <cons (& <num $s $n> pat1)
     <cons <num ,s ,(+ n 1)>
      <cons <num ,s ,(+ n 2)>
       pat2>>>))
```

Three consecutive tiles

```
(define $kohtsu
  (pattern-function [$pat1 $pat2]
    <cons (& $pat pat1)
     <cons ,pat
      <cons ,pat
       pat2>>>))
```

Three same tiles

```
;;
;; A function that determins whether the hand is finished or not.
;;
(define $agari?
  (match-lambda (multiset hai)
    {[(twin $th_1
        (| (shuntsu $sh_1 (| (shuntsu $sh_2 (| (shuntsu $sh_3 (| (shuntsu $sh_4 <nil>)
                                                                 (kohtsu $kh_1 <nil>)))
                                                (kohtsu $kh_1 (kohtsu $kh_2 <nil>))))
                               (kohtsu $kh_1 (kohtsu $kh_2 (kohtsu $kh_3 <nil>)))))
           (kohtsu $kh_1 (kohtsu $kh_2 (kohtsu $kh_3 (kohtsu $kh_4 <nil>))))))
        (twin $th_2 (twin $th_3 (twin $th_4 (twin $th_5 (twin $th_6 (twin $th_7 <nil>)))))))
      #t]
     [_ #f]}))
```

More complex example, Mahjong

```
;; Pattern modularizations
(define $twin
  (pattern-function [$pat1 $pat2]
    <cons (& $pat pat1)
     <cons ,pat
      pat2>>))
```

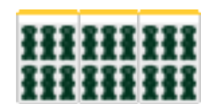
Two same tiles


```
(define $shuntsu
  (pattern-function [$pat1 $pat2]
    <cons (& <num $s $n> pat1)
     <cons <num ,s ,(+ n 1)>
      <cons <num ,s ,(+ n 2)>
       pat2>>>))
```

Three consecutive tiles

```
(define $kohtsu
  (pattern-function [$pat1 $pat2]
    <cons (& $pat pat1)
     <cons ,pat
      <cons ,pat
       pa
```

Three same tiles



Seven twins or one twin + four shuntsu or kohtsu

```
;; A function that determins whether the hand is finished or not.
(define $agari?
  (match-lambda (multiset hai)
    {[(twin $th_1
        (| (shuntsu $sh_1 (| (shuntsu $sh_2 (| (shuntsu $sh_3 (| (shuntsu $sh_4 <nil>)
                                                                 (kohtsu $kh_1 <nil>)))
                                                (kohtsu $kh_1 (kohtsu $kh_2 <nil>))))
                              (kohtsu $kh_1 (kohtsu $kh_2 (kohtsu $kh_3 <nil>)))))
           (kohtsu $kh_1 (kohtsu $kh_2 (kohtsu $kh_3 (kohtsu $kh_4 <nil>))))))
       (twin $th_2 (twin $th_3 (twin $th_4 (twin $th_5 (twin $th_6 (twin $th_7 <nil>)))))))
      #t]
     [_ #f]}))
```

More complex example, Mahjong

```
;;
;; Pattern modularizations
;;
(define $twin
  (pattern-function [$pat1 $pat2]
    <cons (& $pat pat1)
     <cons ,pat
      pat2>>))
```

Two same tiles


```
(define $shuntsu
  (pattern-function [$pat1 $pat2]
    <cons (& <num $s $n> pat1)
     <cons <num ,s ,(+ n 1)>
      <cons <num ,s ,(+ n 2)>
       pat2>>>))
```

Three consecutive tiles

```
(define $kohtsu
  (pattern-function [$pat1 $pat2]
    <cons (& $pat pat1)
     <cons ,pat
      <cons ,pat
       pa
```

Three same tiles

Seven twins or one twin + four shuntsu or kohtsu

```
;;
;; A function that determins whether the hand is finished or not.
;;
(define $agari?
  (match-lambda (multiset hai)
    {[(twin $th_1
        (| (shuntsu $sh_1 (| (shuntsu $sh_2 (| (shuntsu $sh_3 (| (shuntsu $sh_4 <nil>)
                                                                (kohtsu $kh_1 <nil>)))
                                               (kohtsu $kh_1 (kohtsu $kh_2 <nil>))))
                             (k
           (kohtsu $kh_1 (kohts
        (twin $th_2 (twin $th_3
      #t]
     [_ #f]}))
```

Pattern modularization makes programming more simple!

One More Exciting Demonstration

Collections

```
{@{1 2} 3 4};=>{1 2 3 4}
{1 @{2 3} 4};=>{1 2 3 4}
{1 2 @{3 4}};=>{1 2 3 4}

(take 3 {1 2 3 4 5});=>{1 2 3}

(filter odd? {1 2 3 4 5});=>{1 3 5}
(filter even? {1 2 3 4 5});=>{2 4}

(map (+ $ 10) {1 2 3 4 5});=>{11 12 13 14 15}
(map (* $ 2) {1 2 3 4 5});=>{2 4 6 8 10}

(foldl + 0 {1 2 3 4 5});=>15
(foldl * 1 {1 2 3 4 5});=>120
```

Infinite collections

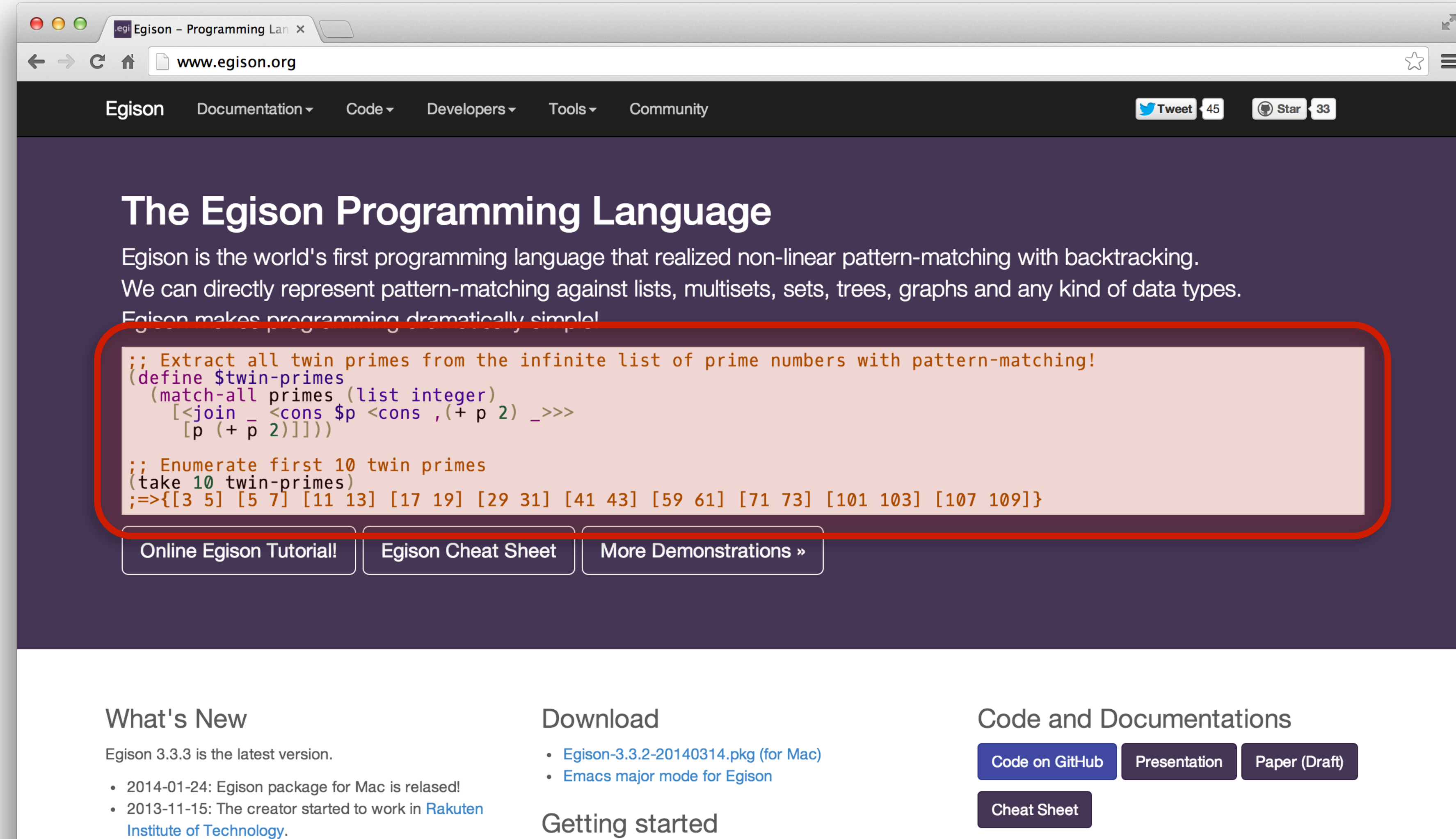
```
(define $ones {1 @ones})
(take 10 ones);=>{1 1 1 1 1 1 1 1 1 1}

(define $nats {1 @(map (+ $ 1) nats)})
(take 10 nats);=>{1 2 3 4 5 6 7 8 9 10}

(define $evens (map (* $ 2) nats))
(take 10 evens);=>{2 4 6 8 10 12 14 16 18 20}

(define $primes (filter prime? nats))
(take 10 primes);=>{2 3 5 7 11 13 17 19 23 29}
```

Beautiful example on elementary mathematics

```
;; Extract all twin primes with pattern-matching!
(define $twin-primes
  (match-all primes (list integer)
    [<join _ <cons $p <cons ,(+ p 2) _>>>
     [p (+ p 2)]]))

;; Enumerate first 10 twin primes
(take 10 twin-primes)
;=>{[3 5] [5 7] [11 13] [17 19] [29 31] [41 43] [59 61]
[71 73] [101 103] [107 109]}

;; Enumerate first 100 twin primes
(take 30 twin-primes)
;=>{[3 5] [5 7] [11 13] [17 19] [29 31] [41 43] [59 61]
[71 73] [101 103] [107 109] [137 139] [149 151] [179
181] [191 193] [197 199] [227 229] [239 241] [269 271]
[281 283] [311 313] [347 349] [419 421] [431 433] [461
463] [521 523] [569 571] [599 601] [617 619] [641 643]
[659 661]}
```

Pattern matching against an infinite collection

This sample is on the homepage of Egison



Egison design policy

Beautiful and Elegant
-> Simple

Visions

What we will be able to do with Egison

- **Access data in new elegant ways**
  - The most elegant query language
  - Able to access lists, sets, graphs, trees or any other data in a unified way
- **Analyze data in new elegant ways**
  - Provide a way to access various algorithm and data structures in a unified way
- **Implement new interesting applications**

  e.g.
  - Natural language processing, New programming languages, Mathematical expression handling, Image processing

What we will be able to do with Egison

Stage1

- **Access data in new elegant ways**
  - The most elegant query language
  - Able to access lists, sets, graphs, trees or any other data in a unified way

Stage2

- **Analyze data in new elegant ways**
  - Provide a way to access various algorithm and data structures in a unified way
- **Implement new interesting applications**

  e.g.
  - Natural language processing, New programming languages, Mathematical expression handling, Image processing

What we will be able to do with Egison

Stage1

- **Access data in new elegant ways**
  - The most elegant query language
  - Able to access lists, sets, graphs, trees or any other data in a unified way

Stage2

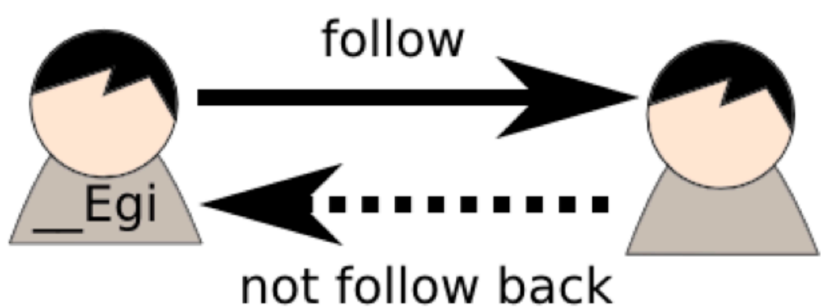
- **Analyze data in new elegant ways**
  - Provide a way to access various algorithm and data structures in a unified way
- **Implement new interesting applications**

  e.g.

  - Natural language processing, New programming languages, Mathematical expression handling, Image processing

Query example

- Query that returns twitter users who are followed by "__Egi" but not follow back "__Egi".



User:

id	integer
name	string

Follow:

from_id	integer
to_id	Integer

SQL version

- Complex and difficult to understand
  - Complex where clause contains “NOT EXIST”
  - Subquery

```
SELECT DISTINCT ON (user4.name) user4.name
  FROM user AS user1,
       follow AS follow2,
       user AS user4
  WHERE user1.name = '__Egi'
    AND follow2.from_id = user1.id
    AND user4.id = follow2.to_id
    AND NOT EXISTS
      (SELECT ''
         FROM follow AS follow3
         WHERE follow3.from_id = follow2.to_id
           AND follow3.to_id = user1.id)
  ORDER BY user4.name;
```

Egison version

- Very Simple
  - No where clauses
  - No subquery

```
(match-all [user follow follow user]
  [[<cons <user $uid ,"__Egi"> _>
    <cons <follow ,uid $fid> _>
    ^<cons <follow ,fid ,uid> _>
    <cons <user ,fid $fname> _>]
   <user fid fname>])
```

Egison version

- Very Simple
  - No where clauses
  - No subquery

Joining 4 tables

```
(match-all [user follow follow user]
  [[<cons <user $uid ,"__Egi"> _>
    <cons <follow ,uid $fid> _>
    ^<cons <follow ,fid ,uid> _>
    <cons <user ,fid $fname> _>]
   <user fid fname>])
```

Egison version

- Very Simple
  - No where clauses
  - No subquery

Joining 4 tables

1. Get id of "__Egi"

```
(match-all [user follow follow user]
  [[<cons <user $uid ,"__Egi"> _>
    <cons <follow ,uid $fid> _>
    ^<cons <follow ,fid ,uid> _>
    <cons <user ,fid $fname> _>]
   <user fid fname>])
```

Egison version

- Very Simple
  - No where clauses
  - No subquery

Joining 4 tables

```
(match-all [user follow follow user]
  [[<cons <user $uid ,"__Egi"> _>
    <cons <follow ,uid $fid> _>
    ^<cons <follow ,fid ,uid> _>
    <cons <user ,fid $fname> _>]
   <user fid fname>])
```

1. Get id of “__Egi”

2. Followed by ‘uid’

楽天 Rakuten

Egison version

- Very Simple
  - No where clauses
  - No subquery

Joining 4 tables

```
(match-all [user follow follow user]
  [[<cons <user $uid ,"__Egi"> _>
    <cons <follow ,uid $fid> _>
   ^<cons <follow ,fid ,uid> _>
    <cons <user ,fid $fname> _>]
   <user fid fname>])
```

not

1. Get id of “__Egi”

2. Followed by ‘uid’

3. But not follow back

Egison version

- Very Simple
  - No where clauses
  - No subquery

Joining 4 tables

```
(match-all [user follow follow user]
  [[<cons <user $uid ,"__Egi"> _>
    <cons <follow ,uid $fid> _>
   ^<cons <follow ,fid ,uid> _>
    <cons <user ,fid $fname> _>]
   <user fid fname>])
```

1. Get id of “__Egi”
2. Followed by ‘uid’
3. But not follow back
4. Get name of ‘fid’

not

楽天 Rakuten

Egison version

- Very Simple
  - No where clauses
  - No subquery

Joining 4 tables

not

```
(match-all [user follow follow user]
  [[<cons <user $uid ,"__Egi"> _>
    <cons <follow ,uid $fid> _>
   ^<cons <follow ,fid ,uid> _>
    <cons <user ,fid $fname> _>]
   <user fid fname>]])
```

1. Get id of "__Egi"
2. Followed by 'uid'
3. But not follow back
4. Get name of 'fid'
Return the results

Egison version

- Very Simple
  - No where clauses
  - No subquery

Joining 4 tables

```
(match-all [user follow follow user]
  [[<cons <user $uid ,"__Egi"> _>
    <cons <follow ,uid $fid> _>
   ^<cons <follow ,fid ,uid> _>
    <cons <user ,fid $fname> _>]
   <user fid fname>])
```

not

1. Get id of "__Egi"
2. Followed by 'uid'
3. But not follow back
4. Get name of 'fid'

Return the results

We can run this query against data in SQLite!

GUI frontend

- **We'll provide GUI for intuitive data access**
  - **Data access** for even **non-engineers**
  - Engineers can concentrate on **data analysis**

Very Easy!

What we will be able to do with Egison

Stage1

- **Access data in new elegant ways**
  - The most elegant query language
  - Able to access lists, sets, graphs, trees or any other data in a unified way

Stage2

- **Analyze data in new elegant ways**
  - Provide a way to access various algorithm and data structures in a unified way
- **Implement new interesting applications**

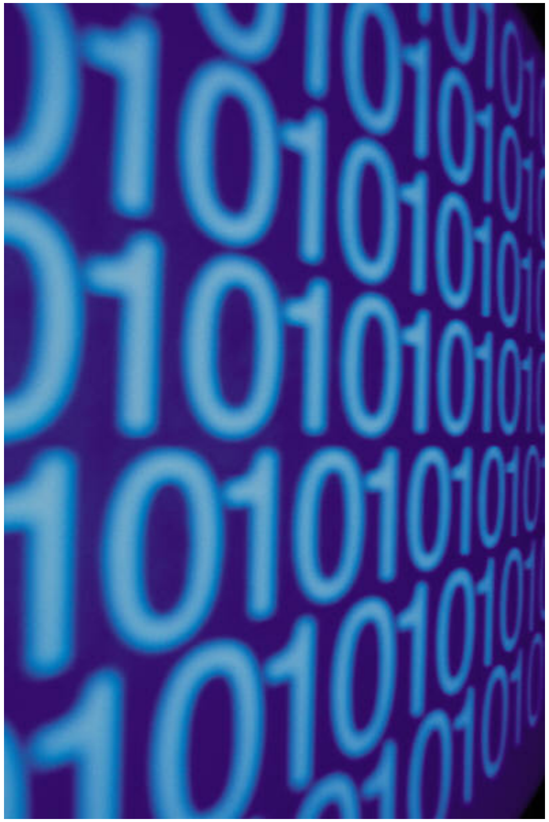
  e.g.

  - Natural language processing, New programming languages, Mathematical expression handling, Image processing

Database in the next age

In future, databases will be embedded in programming languages and hidden.

We will be able to handle databases directly and easily as arrays and hashes in existing languages.



The pattern-matching of Egison will play a necessary role for this future.

The other funny plans

- **Access data in new elegant ways**
  - The most elegant query language
  - Able to access lists, sets, graphs, trees or any other data in a unified way

Stage1

- **Analyze data in new elegant ways**
  - Provide a way to access various algorithm and data structures in a unified way

Stage2

- **Implement new interesting applications**

  e.g.

  - Natural language processing, New programming languages, Mathematical expression handling, Image processing

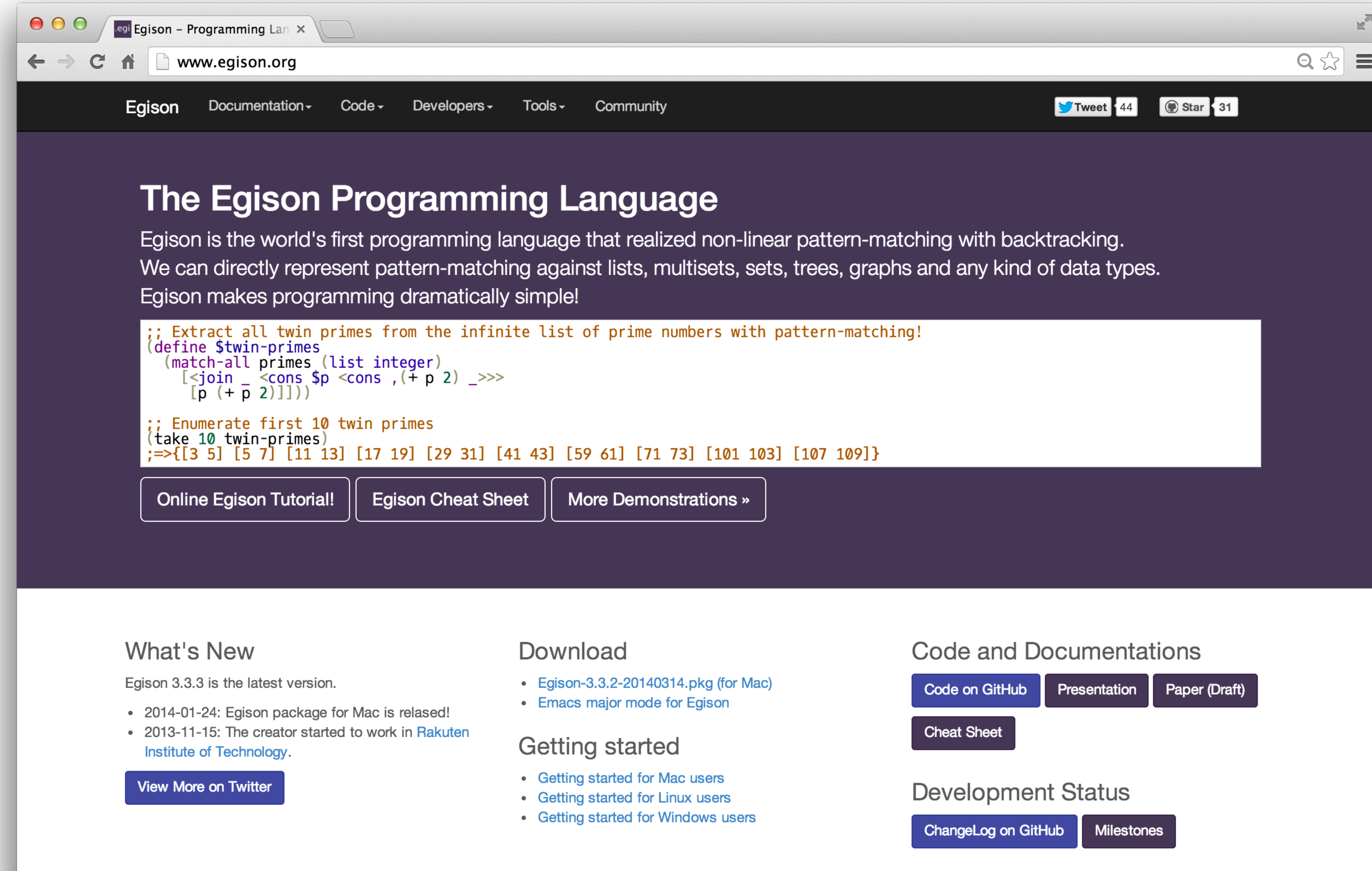
Egison has wide range of applications

- Data mining
  - Work as the most elegant query language
- Natural Language Processing
  - Enable to handle complex syntax structures intuitively as humans do in their mind
- New Programming Languages
- Mathematical expression handling
  - Enable to handle complex structures easily
  - Enable to handle various mathematical notion directly

Egison is an inevitable and necessary innovation in the history of computer science

The Current Situation

Egison website



Online demonstrations

Egison – Programming Lan ×

www.egison.org/demonstrations/poker-hands.html

Egison Documentation Code Developers Tools Community Tweet 1 Star 31

Poker-hands Demonstration

You can edit and run following code. Enjoy Egison programming!

```
;;;
;;;
;;; Poker-hands demonstration
;;;
;;;

;;
;; Matcher definitions
;;
(define $suit
  (algebraic-data-matcher
    {<spade> <heart> <club> <diamond>}))

(define $card
  (algebraic-data-matcher
    {<card suit (mod 13)>}))

;;
;; A function that determins poker-hands
;;
(define $poker-hands
  (lambda [$cs]
    (match cs (multiset card)
      {[<cons <card $s $n>
         <cons <card ,s ,(- n 1)>
          <cons <card ,s ,(- n 2)>
           <cons <card ,s ,(- n 3)>
            <cons <card ,s ,(- n 4)>
             <nil>>>>>>
        <Straight-Flush>]
       [<cons <card _ $n>
         <cons <card _ ,n>
          <cons <card _ ,n>
            <cons <card _ ,n>
              <cons _
                <nil>>>>>>
        <Four-of-Kind>]
       [<cons <card _ $m>
         <cons <card _ ,m>
          <cons <card _ ,m>
           <cons <card _ $n>
```

Installer of Egison and ‘egison-tutorial’

- Egison can be installed on Mac, Windows and Linux.
  - We’ve prepared a package for Mac
  - Download it from http://www.egison.org

Install me Egison and please try ‘egison-tutorial’!

Get following commands!

- egison
- egison-tutorial

‘egison-tutorial’

Egison – Programming Lan… × | Online Egison Tutorial ×

try.egison.org

```
Online Egison Tutorial
Egison Tutorial Version 3.3.3 (C) 2013-2014 Satoshi Egi
http://www.egison.org
Welcome to Egison Tutorial!
==============================
1: Calculate numbers
2: Basics of functional programming
3: Basics of pattern-matching
4: Pattern-matching against infinite collections
==============================
** Information **
1. To procceed the next in the tuturial, please press just 'Enter'.
2. You can input a multi-line expression using 'Shift+Enter'.
*****************
Choose a section to learn.
(1-4): 4
====================
We can write a pattern-matching against infinite lists even if that has infinite results.
Note that Egison really enumurate all pairs of two natural numbers in the following example.

Examples:
  (take 10 (match-all nats (set integer) [<cons $m <cons $n _>> [m n]]))
====================
> (take 10 (match-all nats (set integer) [<cons $m <cons $n _>> [m n]]))
{[1 1] [1 2] [2 1] [1 3] [2 2] [3 1] [1 4] [2 3] [3 2] [4 1]}
> (take 30 (match-all nats (set integer) [<cons $m <cons $n _>> [m n]]))
{[1 1] [1 2] [2 1] [1 3] [2 2] [3 1] [1 4] [2 3] [3 2] [4 1] [1 5] [2 4] [3 3] [4 2] [5 1] [1 6] [2
[6 1] [1 7] [2 6] [3 5] [4 4] [5 3] [6 2] [7 1] [1 8] [2 7]}
>
```

You’ll get Egison easily!

‘egison-tutorial’

Egison – Programming Lan

Online Egison Tutorial

try.egison.org

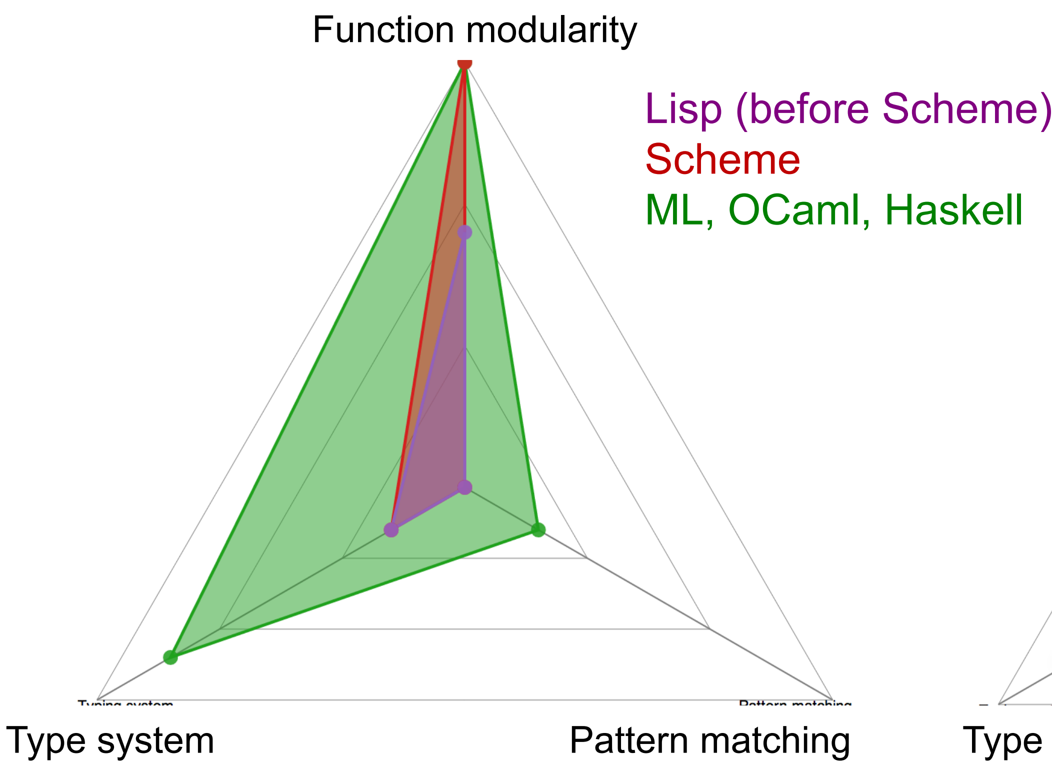
```
Online Egison Tutorial
Egison Tutorial Version 3.3.3 (C) 2013-2014 Satoshi Egi
http://www.egison.org
Welcome to Egison Tutorial!
==============================
1: Calculate numbers
2: Basics of functional programming
3: Basics of pattern-matching
4: Pattern-matching against infinite collections
==============================
** Information **
1. To procceed the next in the tuturial, please press just 'Enter'.
2. You can input a multi-line expression using 'Shift+Enter'.
*****************
Choose a section to learn.
(1-4): 4
====================
We can write a pattern-matching against infinite lists even if that has infinite results.
Note that Egison really enumurate all pairs of two natural numbers in the following example.

Examples:
  (take 10 (match-all nats (set integer) [<cons $m <cons $n _>> [m n]]))
====================
> (take 10 (match-all nats (set integer) [<cons $m <cons $n _>> [m n]]))
{[1 1] [1 2] [2 1] [1 3] [2 2] [3 1] [1 4] [2 3] [3 2] [4 1]}
> (take 30 (match-all nats (set integer) [<cons $m <cons $n _>> [m n]]))
{[1 1] [1 2] [2 1] [1 3] [2 2] [3 1] [1 4] [2 3] [3 2] [4 1] [1 5] [2 4] [3 3] [4 2] [5 1] [1 6] [2
[6 1] [1 7] [2 6] [3 5] [4 4] [5 3] [6 2] [7 1] [1 8] [2 7]}
>
```

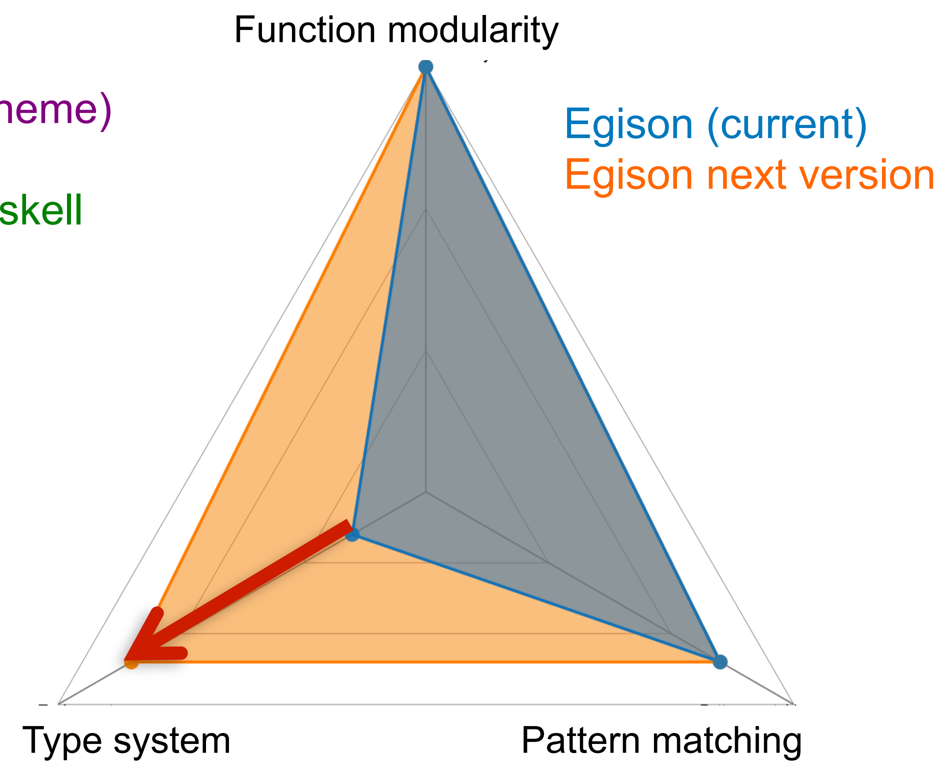
We can try it online, too!

Next Plans

Egison as the programming language

Make Egison **the perfect programming language**.





Extending other languages

- **Ruby** https://github.com/egison/egison-ruby

Non-linear patterns

Non-linear patterns are the most important feature of our pattern-mathcing system. Patterns which don't have `_` ahead of them are value patterns. It matches the target when the target is equal with it.

```
match_all([1, 2, 3, 2, 5]) do
  with(Multiset.(_a, a, *_)) do
    a #=> [2,2]
  end
end
```

```
match_all([30, 30, 20, 30, 20]) do
  with(Multiset.(_a, a, a, _b, b)) do
    [a, b] #=> [[30,20], ...]
  end
end
```

```
match_all([5, 3, 4, 1, 2]) do
  with(Multiset.(_a, (a + 1), (a + 2), *_)) do
    a #=> [1,2,3]
  end
end
```

- **Python (planning)**
- **Haskell (planning)**

Poker hands in Ruby

```
def poker_hands cs
  match(cs) do
    with(Multiset.(_[_s, _n], _[s, (n + 1)], _[s, (n + 2)], _[s, (n + 3)], _[s, (n + 4)])) do
      "Straight flush"
    end
    with(Multiset.(_[_, _n], _[_, n], _[_, n], _[_, n], _)) do
      "Four of kind"
    end
    with(Multiset.(_[_, _m], _[_, m], _[_, m], _[_, _n], _[_, n])) do
      "Full house"
    end
    with(Multiset.(_[_s, _], _[s, _], _[s, _], _[s, _], _[s, _])) do
      "Flush"
    end
    with(Multiset.(_[_, _n], _[_, (n + 1)], _[_, (n + 2)], _[_, (n + 3)], _[_, (n + 4)])) do
      "Straight"
    end
    with(Multiset.(_[_, _n], _[_, n], _[_, n], _, _)) do
      "Three of kind"
    end
    with(Multiset.(_[_, _m], _[_, m], _[_, _n], _[_, n], _)) do
      "Two pairs"
    end
    with(Multiset.(_[_, _n], _[_, n], _, _, _)) do
      "One pair"
    end
    with(Multiset.(_, _, _, _, _)) do
      "Nothing"
    end
  end
end
```

Database support

We will extend support for high speed data storages as the backend of Egison.

MySQL

redis

Cassandra

I appreciate your request for support !

Thank you!

Please visit our website!
http://www.egison.org
Follow us in Twitter @Egison_Lang

Let’s talk and collaborate with us!
satoshi.egi@mail.rakuten.com