

The Programming Language Egison

Vol.01 Feb/18/2014

Satoshi Egi

Rakuten Institute of Technology

<http://rit.rakuten.co.jp/>

Profile of my language Egison

Paradigm	Pattern-matching-oriented, Pure functional
Author	Satoshi Egi
License	MIT
Version	3.2.21 (2014/02/18)
First Released	2011/5/24
Filename Extension	.egi
Implemented in	Haskell (about 2,500 lines)

Two aspects of programming languages

- **Human-centric aspect**
 - Represent human's intuition directly
- **Computer-centric aspect**
 - Represent how computers run directly

Two aspects of programming languages

- **Human-centric aspect**
 - Represent human's intuition directly
- **Computer-centric aspect**
 - Represent how computers run directly

Evolution of languages from human-centric aspect

- **Fortran(1957)**
 - Modular programming*
- **Lisp(1958)**
 - First class functions*
 - Garbage collector*
- **Scheme(1975)**
 - Lexical scope with first class functions*
- **Haskell(1990)**
 - Purely functional programming
 - Strong static typing
- **Egison(2011)**
 - Pattern-matching against unfree data types*
e.g. list, multiset, set, graph, tree, ...

* : World first

The first sample code 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 <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 _ <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 _ <nil>>>>>
        <One-Pair>]
      [<cons _ <cons _ <cons _ <cons _ <cons _ <nil>>>>>
        <Nothing>] }))))
```

The first sample code 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 <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 <card _ ,n>
  <cons <card _ ,n>
  <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 _ <cons _ <cons _ <nil>>>>>
  <One-Pair>]
[<cons _ <cons _ <cons _ <cons _ <cons _ <cons _ <nil>>>>>
  <Nothing>]})}))
```

Match as a set of cards

The first sample code 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 <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
                <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

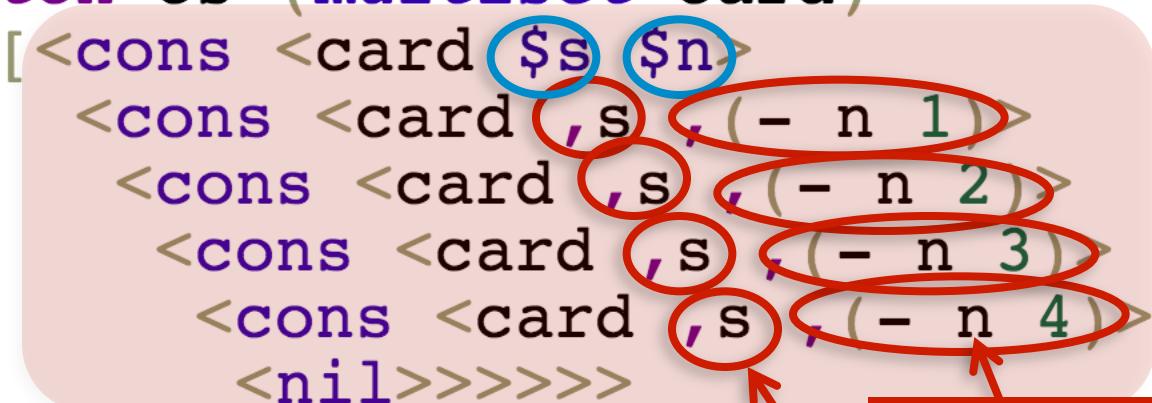
```
(define $poker-hands
  (lambda [$cs]
    (match cs (multiset card)
      {[<cons <card $s $n>
        <cons <card ,s . (- n 1)>
        <cons <card ,s . (- n 2)>
        <cons <card ,s . (- n 3)>
        <cons <card ,s . (- n 4)>
        <nil>>>>>
        <Straight-Flush>]
      [<cons <card _ $n>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



We can use non-linear patterns

The first sample code 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 <card _ ,n>
        <cons _ <nil>>>>>
      <Four-of-Kind>]
      [<cons <card $m>
        <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 _ <nil>>>>>
  <One-Pair>]
  [<cons _ <cons _ <cons _ <cons _ <nil>>>>>
  <Nothing>] })))
```

Pattern for two pair

The pattern for two pair

```
<Three-or-King>]
[<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>]  


Same number with $m



Same number with $n



Matches with any suit



Matches with any card


```

The pattern for two pair

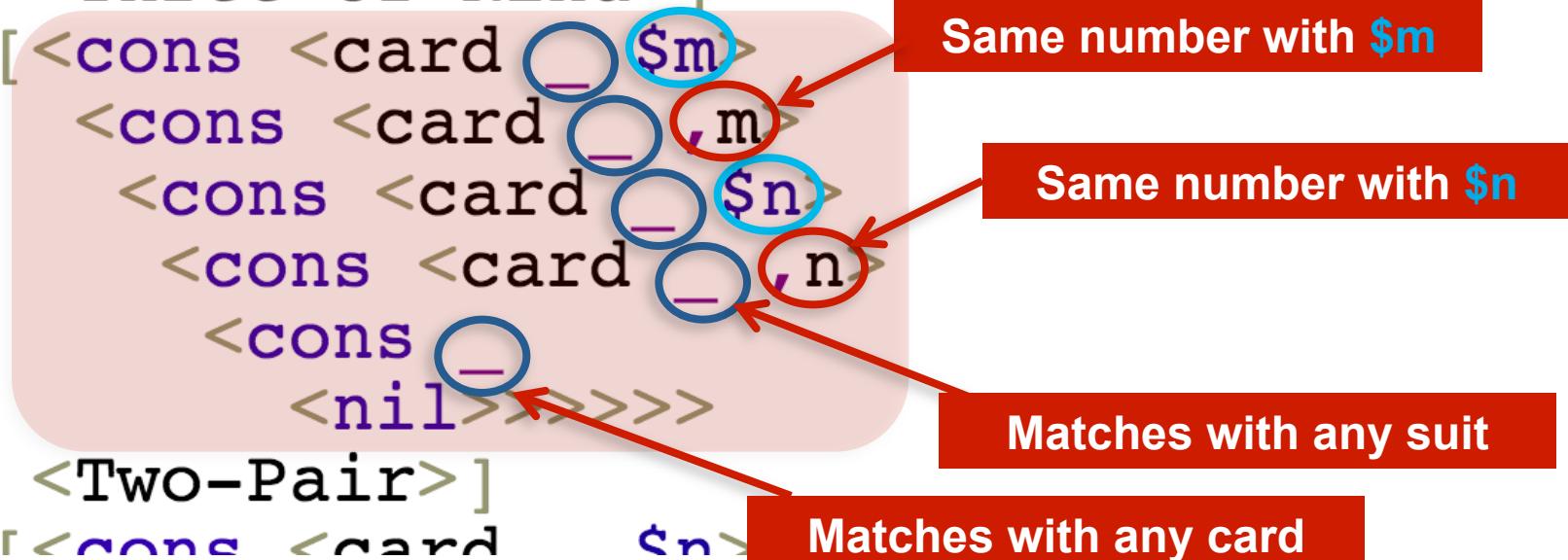
```
<Three-of-Kind> |  
[<cons <card $m>  
  <cons <card m>  
    <cons <card $n>  
      <cons <card n>  
        <cons <nil>>>>>>  
<Two-Pair>]  
<cons <card $n>
```

Same number with \$m

Same number with \$n

Matches with any suit

Matches with any card



Non-linear patterns have very
strong power

The first sample code

```
(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 <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 <card _ ,n>
        <cons <card _ ,n>
        <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 _ <nil>>>>>
      ] <One-Pair>]
      [<cons _ <cons _ <cons _ <cons _ <cons _ <nil>>>>>
      ] <Nothing>])))
```

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

The first sample code

```
(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 <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 _ <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 _ <nil>>>>>
      ] <One-Pair>]
      [<cons _ <nil>>>>>
      ] <Nothing>]))))
```

Isn't it great?



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

Our next plans

- **Provide new elegant ways to access data**
 - Create the most elegant query language
 - Able to access lists, sets, graphs, trees or any other data in a unified way
- **Provide new elegant ways for data analysis**
 - Provide a way to access various algorithm and data structures in a unified way
- **Implementing new interesting applications**
 - e.g.
 - Natural language processing, New programming languages, Mathematical expression handling
 - ...**(I'd like various new challenges)**

Our next plans

- 
- Stage1**
- **Provide new elegant ways to access data**
 - Create the most elegant query language
 - Able to access lists, sets, graphs, trees or any other data in a unified way
- Stage2**
- **Provide new elegant ways for data analysis**
 - Provide a way to access various algorithm and data structures in a unified way
 - **Implementing new interesting applications**
 - e.g.
 - Natural language processing, New programming languages, Mathematical expression handling
 - ...**(I'd like various new challenges)**

Our next plans

- **Provide new elegant ways to access data**

Stage1

- Create the most elegant query language
- Able to access lists, sets, graphs, trees or any other data in a unified way

- **Provide new elegant ways for data analysis**

Stage2

- Provide a way to access various algorithm and data structures in a unified way

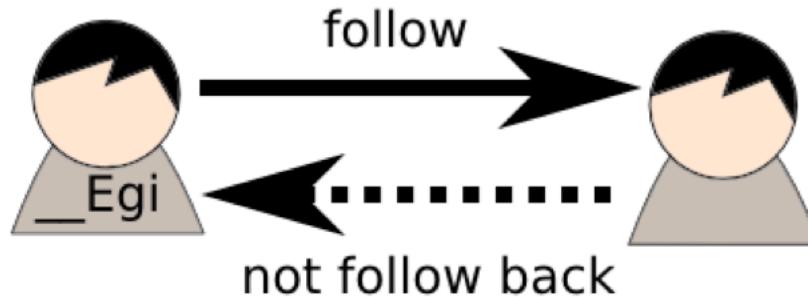
- **Implementing new interesting applications**

e.g.

- Natural language processing, New programming languages, Mathematical expression handling
- ...**(I'd like various new challenges)**

Query example

- Query that returns twitter users who are followed by “_Egi” but not follow back “_Egi”.



SQL version

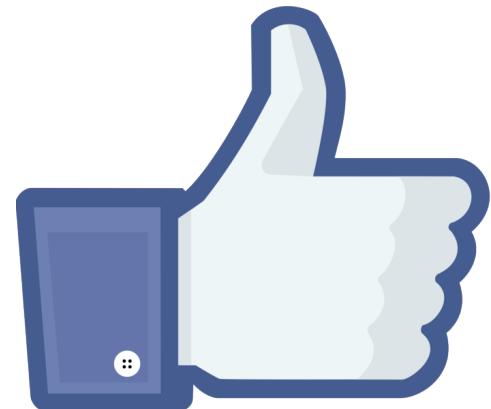
- Complex and difficult to understand
 - Complex where clause contains “NOT EXIST”
 - Subquery

```
SELECT DISTINCT ON (twitter_user4.screen_name) twitter_user4.screen_name
  FROM twitter_user AS twitter_user1,
       follow AS follow2,
       twitter_user AS twitter_user4
 WHERE twitter_user1.screen_name = '__Egi'
   AND follow2.from_uid = twitter_user1.uid
   AND twitter_user4.uid = follow2.to_uid
   AND NOT EXISTS
     (SELECT ''
      FROM follow AS follow3
      WHERE follow3.from_uid = follow2.to_uid
        AND follow3.to_uid = twitter_user1.uid)
 ORDER BY twitter_user4.screen_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> _>]
return <user fid fname>
```

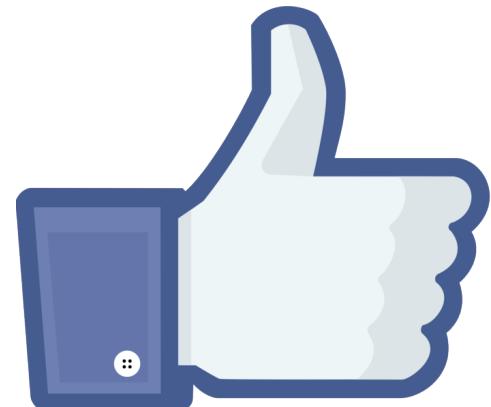


Egison version

- 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>_>]
return <user fid fname>
```

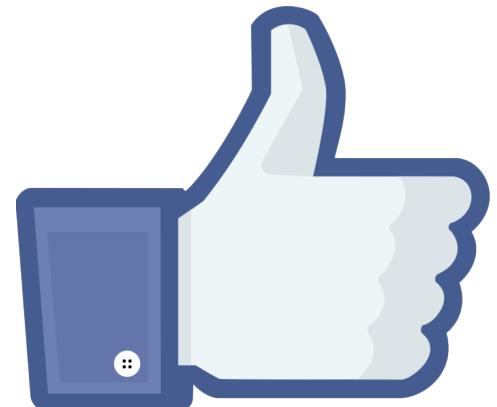


Egison version

- Very Simple
 - No where clauses
 - No subquery

Joining 4 tables

```
match-all [user follow follow user]
[<cons <user $uid , " __Egi"> _> 1. Get id of "__Egi"
 <cons <follow ,uid $fid> _>
 ^<cons <follow ,fid ,uid> _>
 <cons <user ,fid $fname> _>]
return <user fid fname>
```



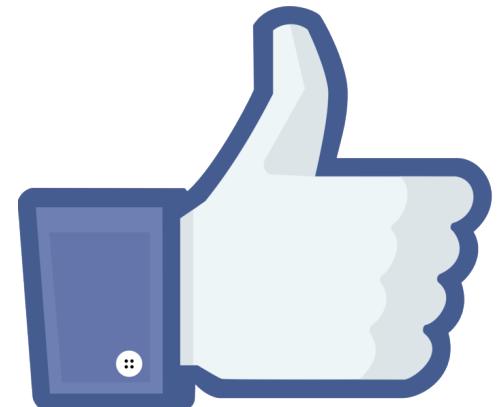
Egison version

- 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> _>]
return <user fid fname>
```

1. Get id of “__Egi”
2. Followed by ‘uid’



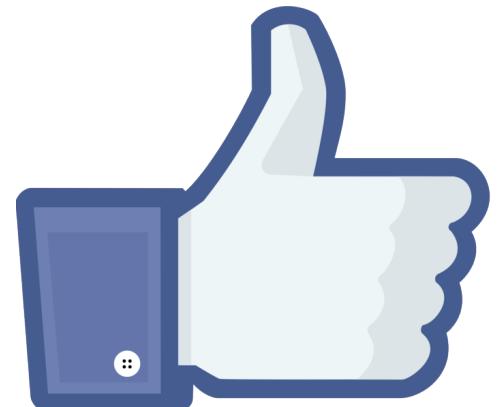
Egison version

- 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> _>]
return <user fid fname>
```

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



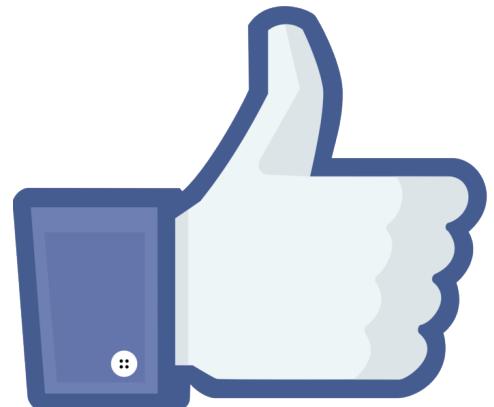
Egison version

- Very Simple
 - No where clauses
 - No subquery

Joining 4 tables

```
match-all [user follow follow user]
[<cons <user $uid , " __Egi"> _>
<cons <follow ,uid $fid> _>
not ^<cons <follow ,fid ,uid> _>
<cons <user ,fid $fname> _>]
return <user fid fname>
```

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



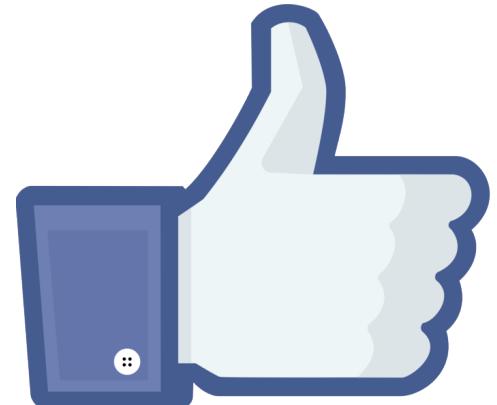
Egison version

- 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> _>]
return <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

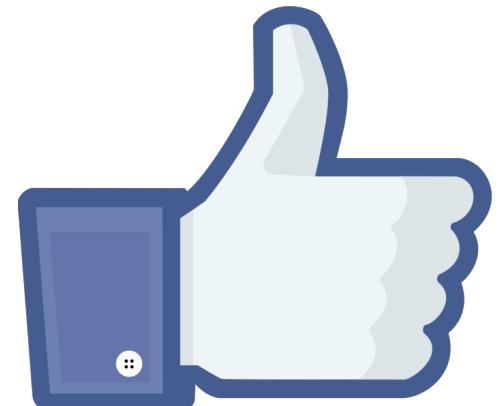
Joining 4 tables

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



However...

- **Difficult to use for many, yet**
 - Our syntax is simple but unusual.



GUI frontend

- We'll provide GUI for intuitive data access
 - Data access for even non-engineers

Very Easy!



The plan after the next

- **Provide new elegant ways to access data**

Stage1

- Create the most elegant query language
- Able to access lists, sets, graphs, trees or any other data in a unified way

- **Provide new elegant ways for data analysis**

Stage2

- Provide a way to access various algorithm and data structures in a unified way

- **Implementing new interesting applications**

e.g.

- Natural language processing, New programming languages, Mathematical expression handling
- ...**(I'd like various new challenges)**

For that...

- Human-centric aspect
 - Represent human's intuition directly
- Computer-centric aspect
 - Represent how computers run directly

The plans always in the mind

- **Provide new elegant ways to access data**

Stage1

- Create the most elegant query language
- Able to access lists, sets, graphs, trees or any other data in a unified way

- **Provide new elegant ways for data analysis**

Stage2

- Provide a way to access various algorithm and data structures in a unified way

- **Implementing new interesting applications**

e.g.

- Natural language processing, New programming languages, Mathematical expression handling
- ...**(I'd like various new challenges)**

Ultimate goal

- The ultimate goal is to create auto theory constructor

Automatically extract beautiful rules
from messy world



Effort for propagation

Egison web site

Please visit the web site. <http://www.egison.org>

The screenshot shows the Egison website homepage. At the top, there's a navigation bar with links for "Egison", "Getting started", "Tutorials", "Manual", and "Contact". On the right of the navigation bar are social sharing buttons for Twitter and GitHub, and a star icon indicating 20 stars.

The main content area features a large banner with the heading "Brand-new Programming Experiences" and a subtext: "You can directly represent pattern-matching against sets, graphs, XMLs or any other data types." Below this is a blue button labeled "View Demonstrations".

Below the banner, there are three sections: "What's New", "Download", and "Code and Documentations".

- What's New:** States that Egison 3.2.13 is the latest version. It lists two recent releases:
 - 2014-01-24: Egison package for Mac is released!
 - 2013-11-15: The creator started to work in Rakuten Institute of Technology.A "View More on Twitter" button is present.
- Download:** Provides a download link for "Egison-3.2.13-20140204.pkg (for Mac)".
- Code and Documentations:** Links to "Code on GitHub", "Presentation", and "Paper (Draft)".

On the right side, there are sections for "Latest Articles" and "FAQ".

At the bottom, there are footer links for "Pattern-matching-oriented", "Demonstrations", and "FAQ".

Online interpreter

We can try Egison online right now!

Egison – Programming Lan

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

Egison Getting started Tutorials Manual Contact

Poker-hands Demonstration

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

```
1  ;;;
2  ;;;
3  ;;; Poker-hands demonstration
4  ;;;
5  ;;;
6
7  ;;
8  ;; Matcher definitions
9  ;;
10 (define $suit
11   (algebraic-data-matcher
12     {<spade> <heart> <club> <diamond>}))
13
14 (define $card
15   (algebraic-data-matcher
16     {<card suit (mod 13)>}))
17
18 ;;
19 ;; A function that determines poker-hands
20 ;;
21 (define $poker-hands
22   (lambda [scs]
23     (match cs (multiset card)
24       {[<cons <card $s $n>
25         <cons <card ,s ,(- n 1)>
26         <cons <card ,s ,(- n 2)>
27         <cons <card ,s ,(- n 3)>
28         <cons <card ,s ,(- n 4)>
29         <nil>>>>>
30         <Straight-Flush>]
31       [<cons <card _ $n>
32         <cons <card _ ,n>
33         <cons <card _ ,n>
34         <cons <card _ ,n>
35         <cons
36           <nil>>>>>
37         <Four-of-Kind>]
38       [<cons <card _ $m>
39         <cons <card _ ,m>
40         <cons <card _ ,m>
```

Installer of Egison and ‘egison-tutorial’

- We've prepared a package for Mac users
 - <http://www.egison.org/getting-started.html>



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

Get following commands!

- egison
- egison-tutorial

'egison-tutorial'

```
EGISON /home/egi% egison-tutorial
Egison Tutorial for Version 3.2.3 (C) 2013-2014 Satoshi Egi
http://www.egison.org
Welcome to Egison Tutorial!
=====
List of tutorials.
1: Lv1 - Calculate numbers
2: Lv2 - Basics of functional programming
3: Lv3 - Define your own functions
4: Lv4 - Basic of pattern-matching
5: Lv5 - Pattern-matching against infinite collections
6: Lv6 (preparing) - Pattern-matching against graphs
7: Lv7 (preparing) - Modularize patterns
8: Lv8 (preparing) - Define your own matchers
=====
Please select a section to learn.
(1-8): 5
=====
We can write a pattern-matching against infinite collections.
Please note that Egison really enumerate all possible cases.
e.g.
  (take 10 (match-all nats (set integer))
  =====
> (take 10 (match-all nats (set integer))
{[1 1] [1 2] [2 1] [1 3] [2 2] [3 1] [1 4] [2 3] [3 2] [4 1]}
> (take 20 (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]
>
Do you want to proceed next? (Y/n):
=====
We can enumerate all two combinations of natural numbers as follow.
e.g.
  (define $two-combs (match-all nats (list integer) [<join _ (& <cons $x_> <join _ <cons $y_>>)
  (take 100 two-combs)
  =====
>
```

You'll get Egison
easily!



Next plan: Extend other languages

- **Ruby** <https://github.com/egisatoshi/egison-ruby>

Non-linear patterns

Non-linear patterns are the most important feature of our pattern-matching 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
```

Next plan: Extend other languages

- **Haskell (planning)**
 - We'd like to start something by the end of year.

Thank you for reaching this page!