

The pattern-matching-oriented programming language Egison

2013/3/4

Satoshi Egi

Profile of my language Egison

Paradigm	Pattern-matching-oriented, Pure functional
Author	Satoshi Egi
License	MIT
Version	3.2.23 (2014/02/27)
First Released	2011/5/24
Filename Extension	.egi
Implemented in	Haskell (about 3,400 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)** * : World first
 - 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, ...

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 _ <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
  <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 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 _ <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 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
      <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 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-or-King>]
[<cons <card _ $m>
  <cons <card _, m>
    <cons <card _, $n>
      <cons <card _, n>
        <cons _ <nil>>>>>
<Two-Pair>]
[<cons <card _ $n>
```

Matches with any card

Matches with any suit

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>>
```

Same number with \$m

Same number with \$n

Matches with any suit

Matches with any card

The pattern for two pair

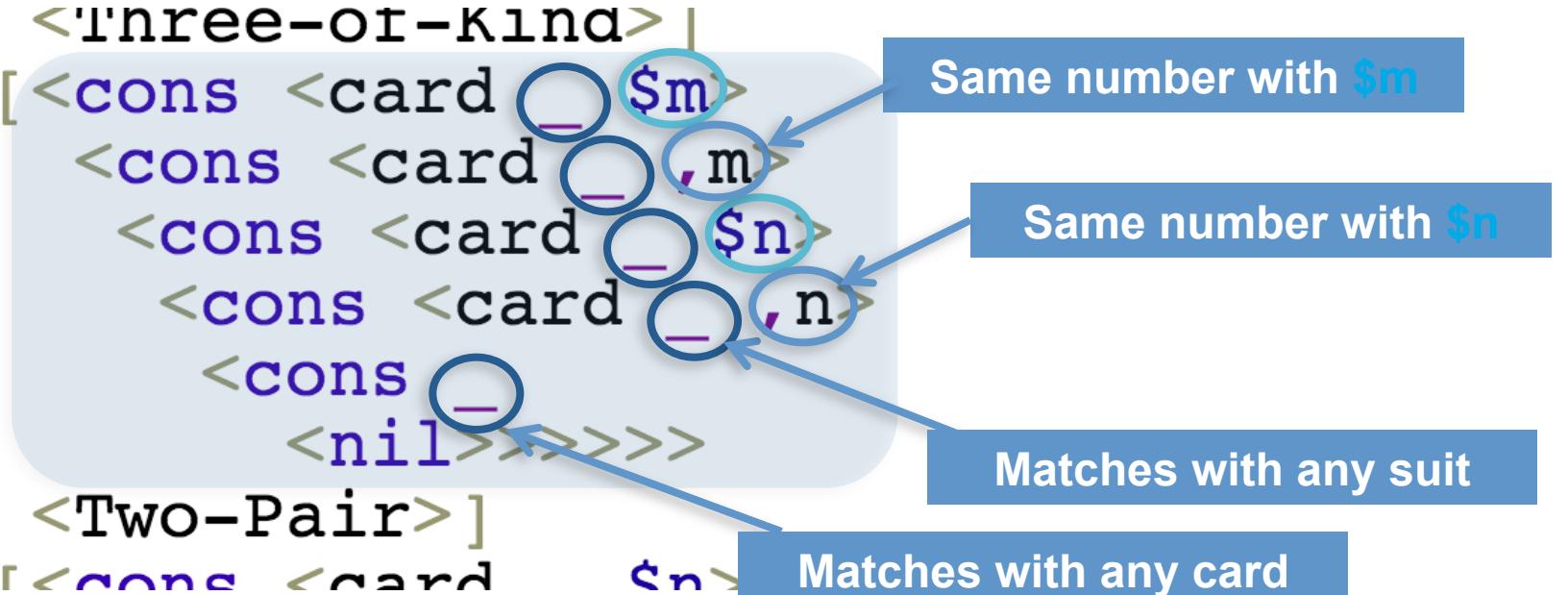
```
<Three-or-King> |  
[<cons <card $m>>  
<cons <card m>>  
<cons <card $n>>  
<cons <card n>>  
<cons <nil>>>>>  
<Two-Pair>]  
<cons <card $n>>
```

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 _ ,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 _ ,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>]))))
```

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
- **Implement 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
 - **Implement 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

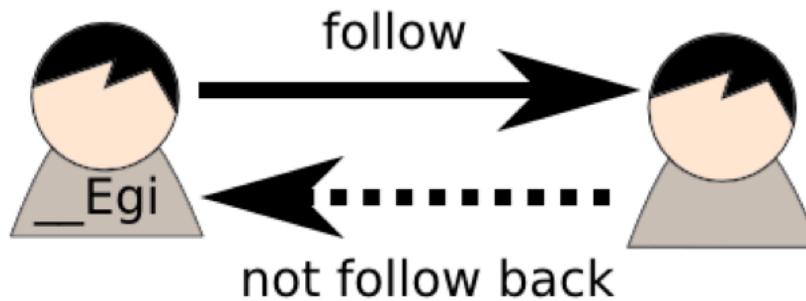
- **Implement new interesting applications**

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”.



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]
[ [
```

Egison version

- Very Simple
 - No where clauses
 - No subquery

Joining 4 tables

```
(match-all [user follow follow user]
[ [
```

Egison version

- Very Simple
 - No where clauses
 - No subquery

Joining 4 tables

```
(match-all [user follow follow user]
[ [
```

Egison version

- Very Simple
 - No where clauses
 - No subquery

Joining 4 tables

```
(match-all [user follow follow user]
[ [
```

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]
  [ [
```

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]
  [ [
```

not

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 <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

Egison version

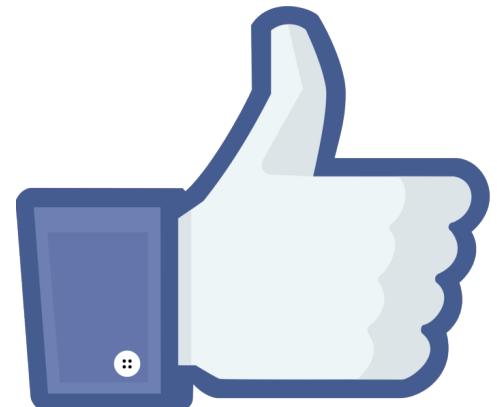
- Very Simple
 - No where clauses
 - No subquery

Joining 4 tables

```
(match-all [user follow follow user]
  [ [
```

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!



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

- **Implement 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

- **Implement new interesting applications**

e.g.

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

Ultimate goal: We'd like to implement a mathematician

A program that automatically proposes **new notions**, **new hypothesis** and **new proofs** is the one of the ultimate dream of Computer Science and humanity.



Effort for propagation

Egison web site

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

The screenshot shows the Egison website homepage as it would appear in a web browser. The header includes a navigation bar with links for Egison, Getting started, Tutorials, Manual, and Contact, along with social sharing buttons for Twitter and GitHub. The main content features a large banner with the heading "Brand-new Programming Experiences" and a subtext about pattern-matching against sets, graphs, XMLs, and other data types. A "View Demonstrations" button is prominently displayed. Below the banner, there are three circular navigation dots. The main content area is divided into several sections: "What's New" (mentioning Egison 3.2.13), "Download" (link to Mac package), "Tutorials" (links to "Egison in one minute", "Egison in thirty minutes", and "Egison in one day"), "Code and Documentations" (links to GitHub, presentation, and paper), and "Latest Articles". At the bottom, there are sections for "Pattern-matching-oriented" (with a note about being pattern-matching-oriented), "Demonstrations" (mentioning pattern-matching against ...), and "FAQ" (with a link to "Why new language?").

Egison - Programming Lan X

www.egison.org

Egison Getting started Tutorials Manual Contact

Tweet 12 Star 20

Brand-new Programming Experiences

You can directly represent pattern-matching against sets, graphs, XMLs or any other data types.

View Demonstrations

○ ● ○

What's New

Egison 3.2.13 is the latest version.

- 2014-01-24: Egison package for Mac is released!
- 2013-11-15: The creator started to work in [Rakuten Institute of Technology](#).

[View More on Twitter](#)

Download

- [Egison-3.2.13-20140204.pkg \(for Mac\)](#)

Tutorials

- [Egison in one minute](#)
- [Egison in thirty minutes](#)
- [Egison in one day](#)

Code and Documentations

[Code on GitHub](#) [Presentation](#) [Paper \(Draft\)](#)

Latest Articles

Pattern-matching-oriented

Egison is the **pattern-matching-oriented** programming

Demonstrations

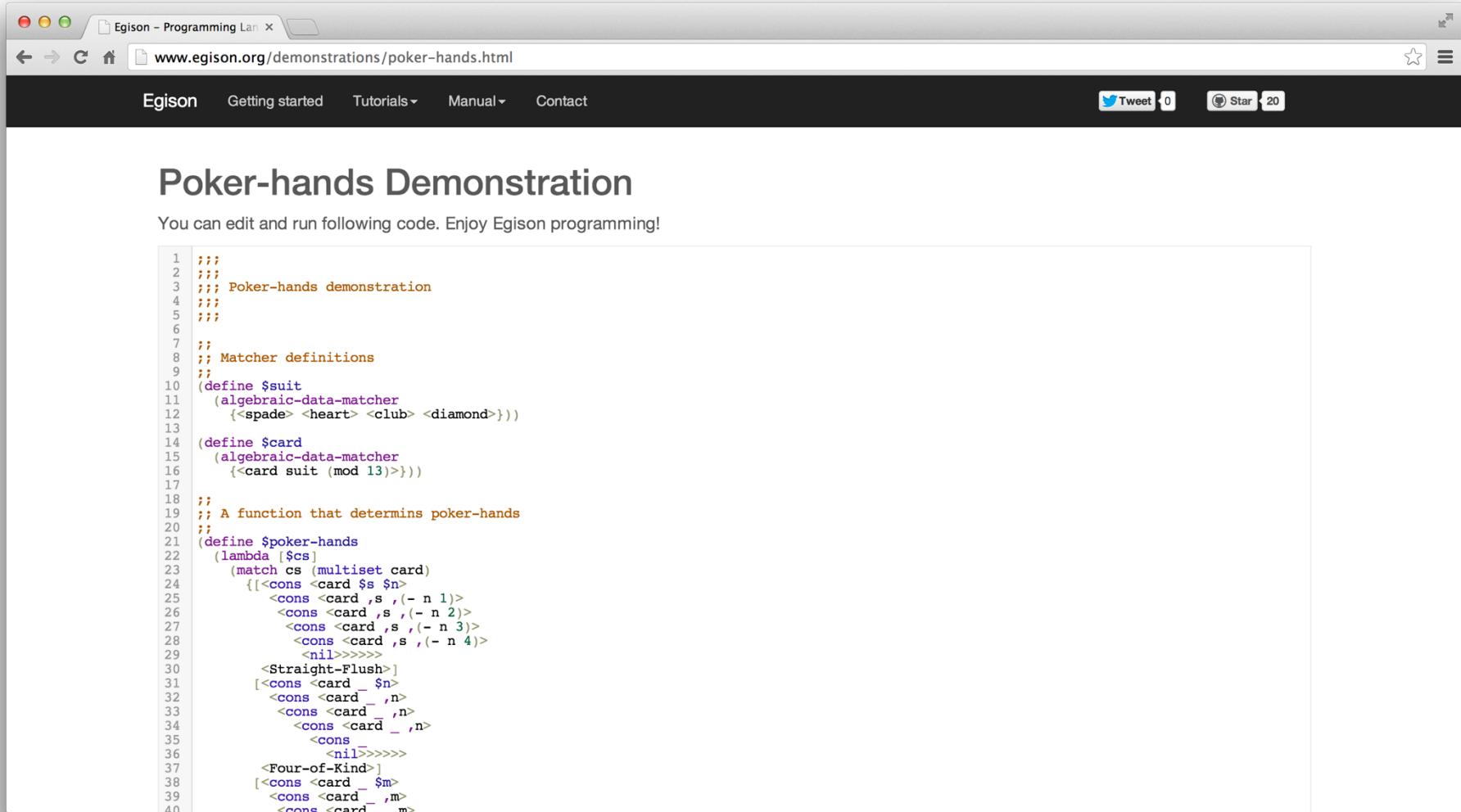
Pattern-matching against ...

FAQ

- Why new language?

Online interpreter

We can try Egison online right now!

A screenshot of a web browser window titled "Egison - Programming Lan". The URL is "www.egison.org/demonstrations/poker-hands.html". The page content is titled "Poker-hands Demonstration" and includes a message: "You can edit and run following code. Enjoy Egison programming!". Below this is a large block of Egison code. The browser interface shows standard navigation buttons, a search bar, and social sharing links for Twitter and GitHub.

```
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'

```
Terminal — ssh — 131x36
10:03:27
EGISON /home/egi% egison-tutorial
Egison Tutorial for Version 3.2.6 (C) 2013-2014 Satoshi Egi
http://www.egison.org
Welcome to Egison Tutorial!
=====
List of sections in the tutorial
1: Calculate numbers
2: Basics of functional programming
3: Define your own functions
4: Basic of pattern-matching
5: Pattern-matching against infinite collections
6: Writing scripts in Egison
=====
Choose a section to learn.
(1-6): 5
=====
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 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] [2 5]
>
Do you want to procced next? (Y/n):
=====
We can enumerate all two combinations of natural numbers as follow.

Examples:
 (define $two-combs (match-all nats (list integer) [<join _ (& <cons $x _> <join _ <cons $y _>>) > [x
 (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

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

Thank you for reaching this page!

Links

- <http://www.egison.org> (Our web site)
- <https://github.com/egison/egison> (GitHub)
- https://twitter.com/Egison_Lang (Twitter)
- <http://hackage.haskell.org/package/egison> (Hackage)