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

* : World first

- **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, …

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 _
             <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 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 _
             <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 <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 _
             <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>
```

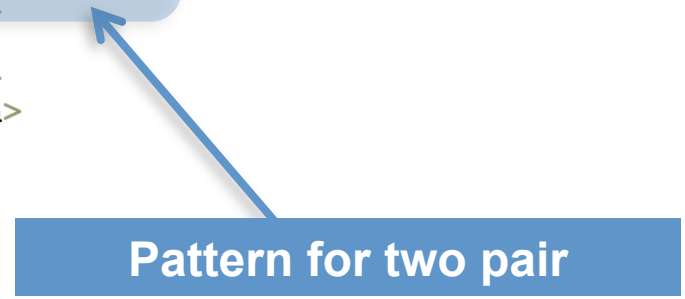
Numbers are serial from $n

Same suit with $s

We can use non-linear patterns

The first sample code of Egison

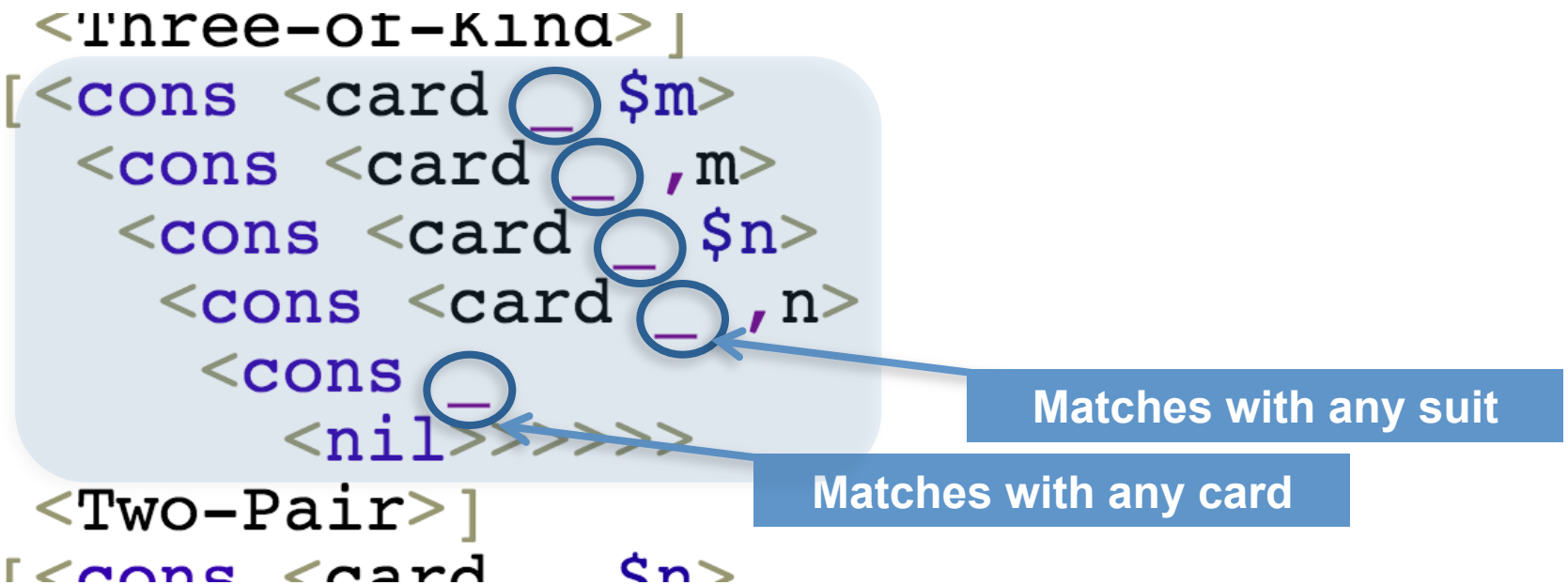
```
(define $poker-hands
  (lambda [$cs]
    (match cs (multiset card)
      {[<cons <card $s $n>
         <cons <card ,s ,(- n 1)>
          <cons <card ,s ,(- n 2)>
           <cons <card ,s ,(- n 3)>
            <cons <card ,s ,(- n 4)>
             <nil>>>>>>
        <Straight-Flush>]
       [<cons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



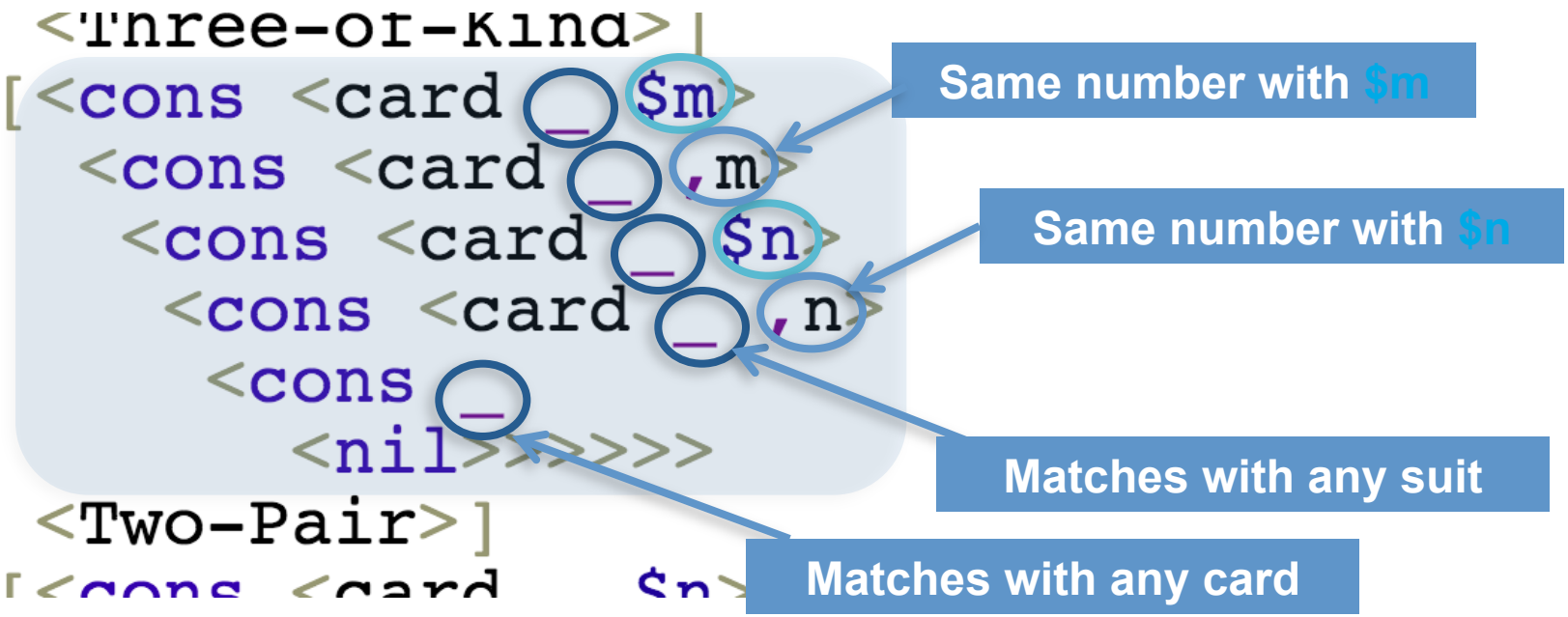
The pattern for two pair

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

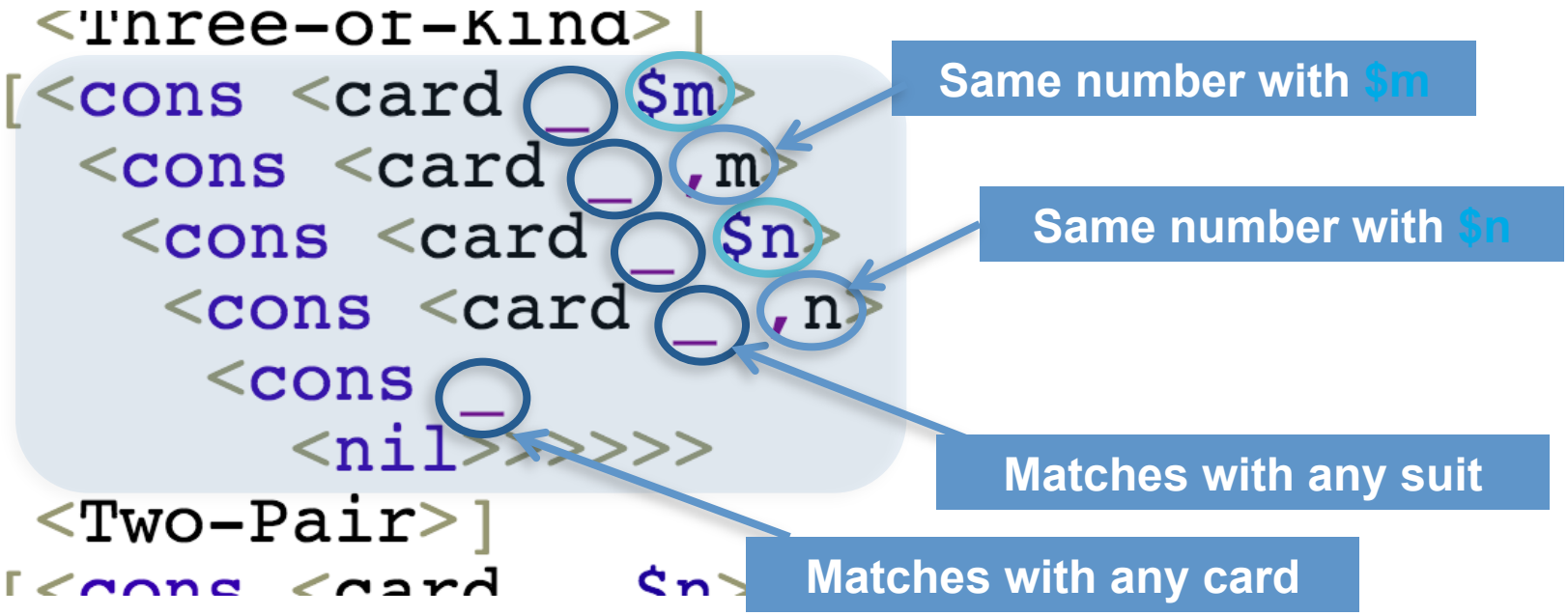
The pattern for two pair



The pattern for two pair



The pattern for two pair



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 _
            <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 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 _
            <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>]}))))
```

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

- **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

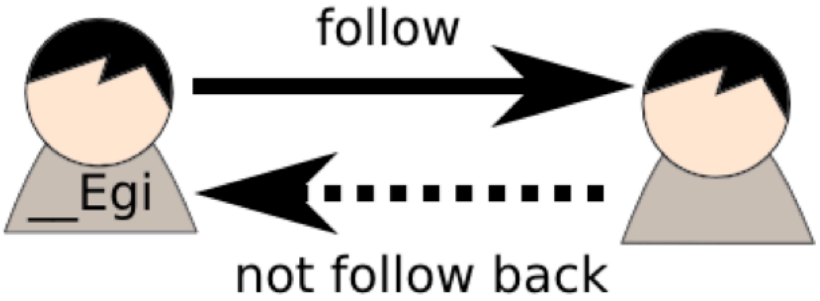
- **Provide new elegant ways for data analysis**
  - Provide a way to access various algorithm and data structures in a unified way

- **Implement new interesting applications**

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

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

```
(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”

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'

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>])
```

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 <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

Egison version

- Very Simple
  - No where clauses
  - No subquery

Joining 4 tables

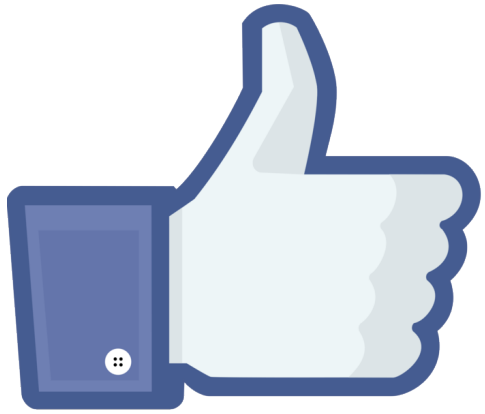
```
(match-all [user follow follow user]
  [[<cons <user $uid ,"__Egi"> _>
    <cons <follow ,uid $fid> _>
    ^<cons <follow ,fid ,uid> _>
    <cons <user ,fid $fname> _>]
   <user fid fname>])
```

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!

The plan after the next

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)**

For that…

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

The plans always in the mind

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)**

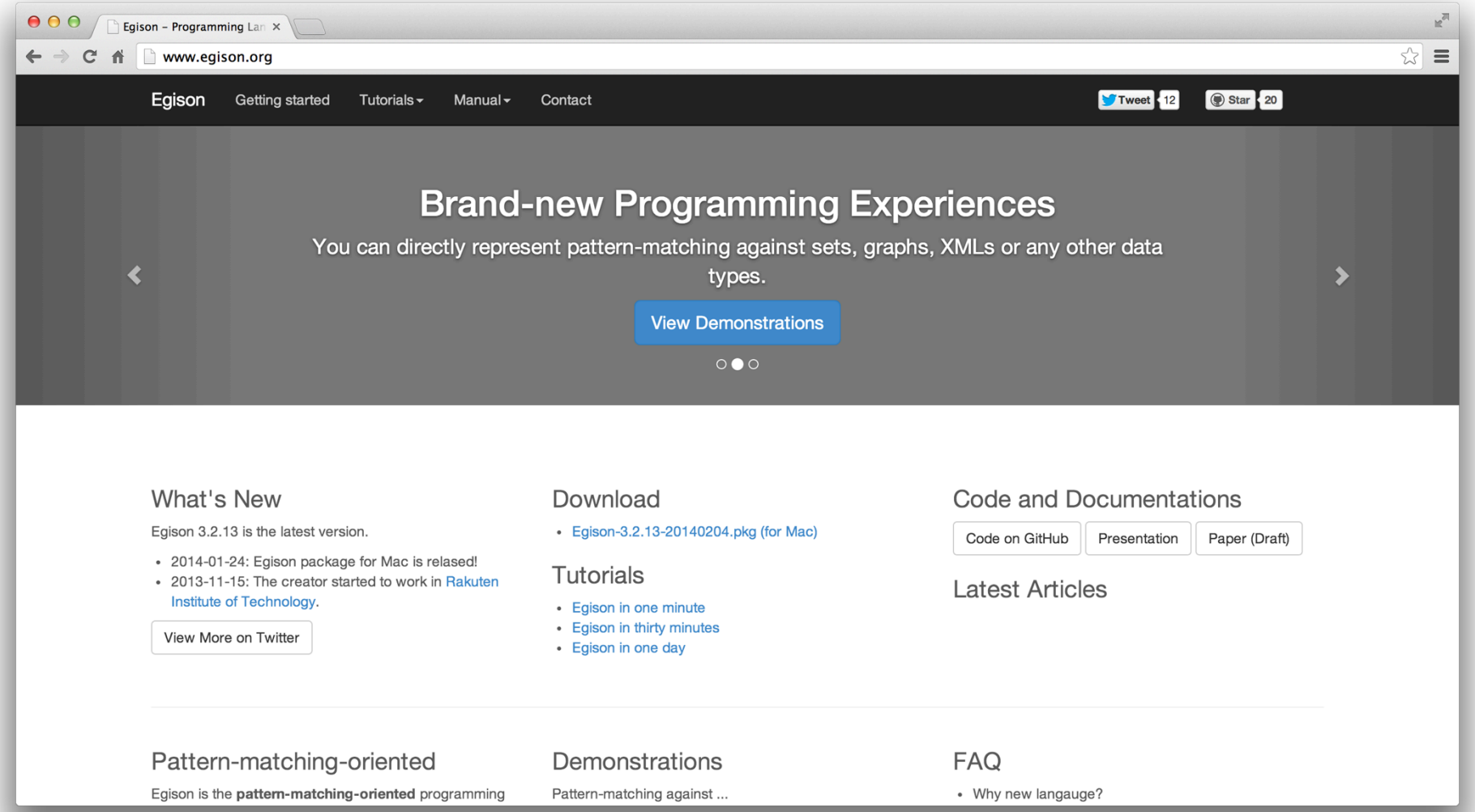
Ultimate goal: We'd like to implement a mathematician

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

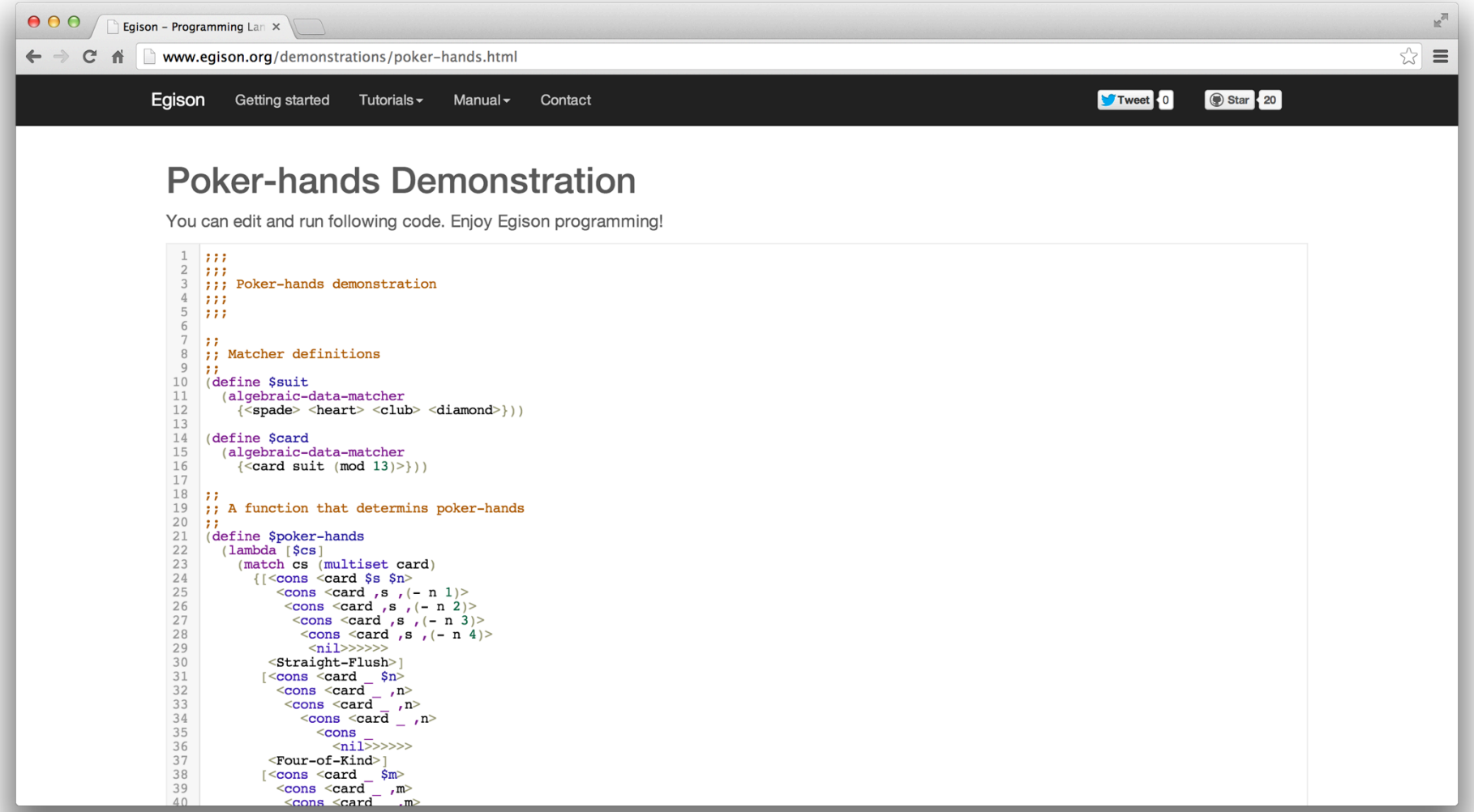
Egison web site

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



Online interpreter

We can try Egison online right now!



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 — 131×36

```
EGISON /home/egi% egison-tutorial                                    10:03:27
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-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
```

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)