The Pattern-Matching Oriented Programming Language Egison

Satoshi Egi

27 September, 2013

My Profile

- Satoshi Egi (江木 聡志)
  - Site : http://www.egison.org/~egi/
  - Github : https://github.com/egisatoshi/
  - Twitter : @__Egi
  - Activity :
    - The inventor of programming language Egison
      - Site : http://www.egison.org
      - Source : https://github.com/egisatoshi/egison3
      - Twitter : @Egison_Lang
  - Award
    - Certified as a Super Creator, IPA, October 2012

My Programming Experience

- UNIX Shell (October, 2007 – November, 2007)
  - I implemented UNIX Shell using C in two months after I first touched terminal.
  - source : https://github.com/egisatoshi/egsh
- Scheme Interpreter (March, 2008)
  - I implemented interpreter of Scheme using Scheme in a day when I first touched Scheme and functional programming language.
- Scheme Compiler (October, 2008 – March, 2009)
  - I implemented compiler of Scheme in Scheme.
  - source : https://github.com/egisatoshi/scheme-compiler
- Egison (March, 2010 – present)
  - I have designed and implemented Egison in Haskell.
  - source : https://github.com/egisatoshi/egison3
- Ruby, Rails, Erlang, PostgreSQL, Emacs Lisp, OCaml, VHDL, Prolog, PHP, …

Outline of Presentation

- Programming Language Egison, so far
  - What is Egison?
  - What Egison can do?
    - Four important features of Egison
  - Comparison with Other Work
- Future of Egison

The Programming Language Egison

- **World's first** programming language that can directly represent **pattern-matching** against sets.
  - Proposed new paradigm "**pattern-matching-oriented**" for the first time.
  - Open source software (MIT license)
  - Version3.0.10 (July, 2013)
  - Released first version in May, 2011
    - Development started from March, 2010
  - I have designed and implemented Egison.
    - Now, Egison community has 15 members.

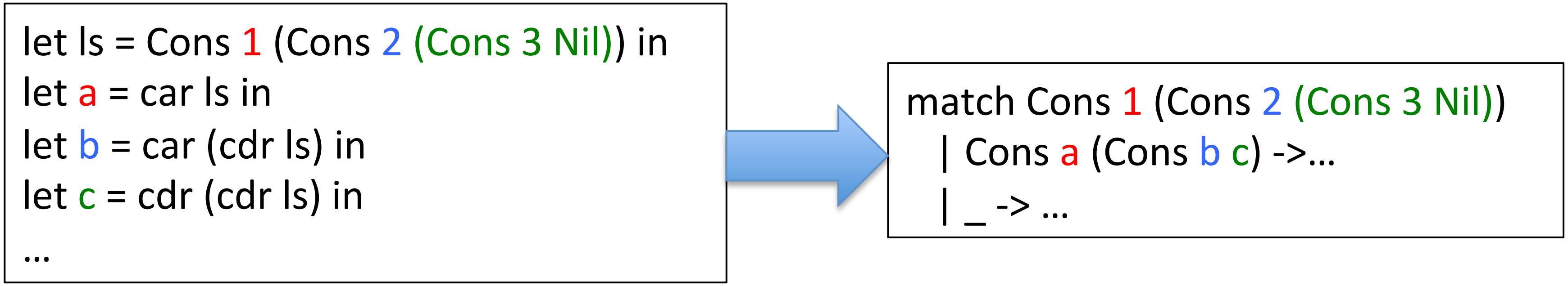
What is Pattern-Matching?

- Pattern-Matching before Egison
- Pattern-Matching of Egison

Syntax of programming languages to represent “destruction of data” and “conditional branches” based on the result of destruction in a simple way.

Pattern-Matching before Egison

- We can represent destruction of “fixed” data like array with a simple pattern.
  - Nested destructor applications are replaced with pattern-matching.
  - Example. Pattern-matching of a list

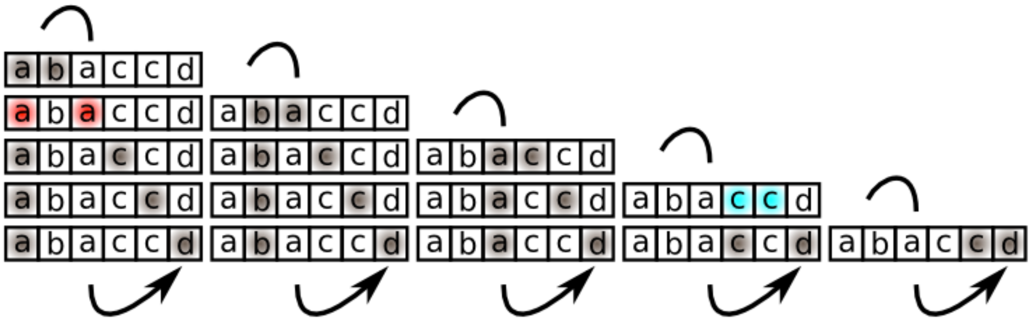


```
let ls = Cons 1 (Cons 2 (Cons 3 Nil)) in
let a = car ls in
let b = car (cdr ls) in
let c = cdr (cdr ls) in
…
```

```
match Cons 1 (Cons 2 (Cons 3 Nil))
  | Cons a (Cons b c) ->…
  | _ -> …
```

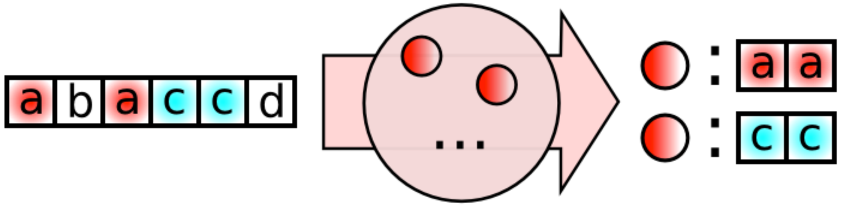
Pattern-Matching of Egison

- We can pattern-match data which has no “standard form” like sets.
  - If we regard a collection {1, 2, 3} as a set, {1, 3, 2} and {3, 2, 1} are same collection with {1, 2, 3}.
  - **Nested loops** are replaced with pattern-matching.



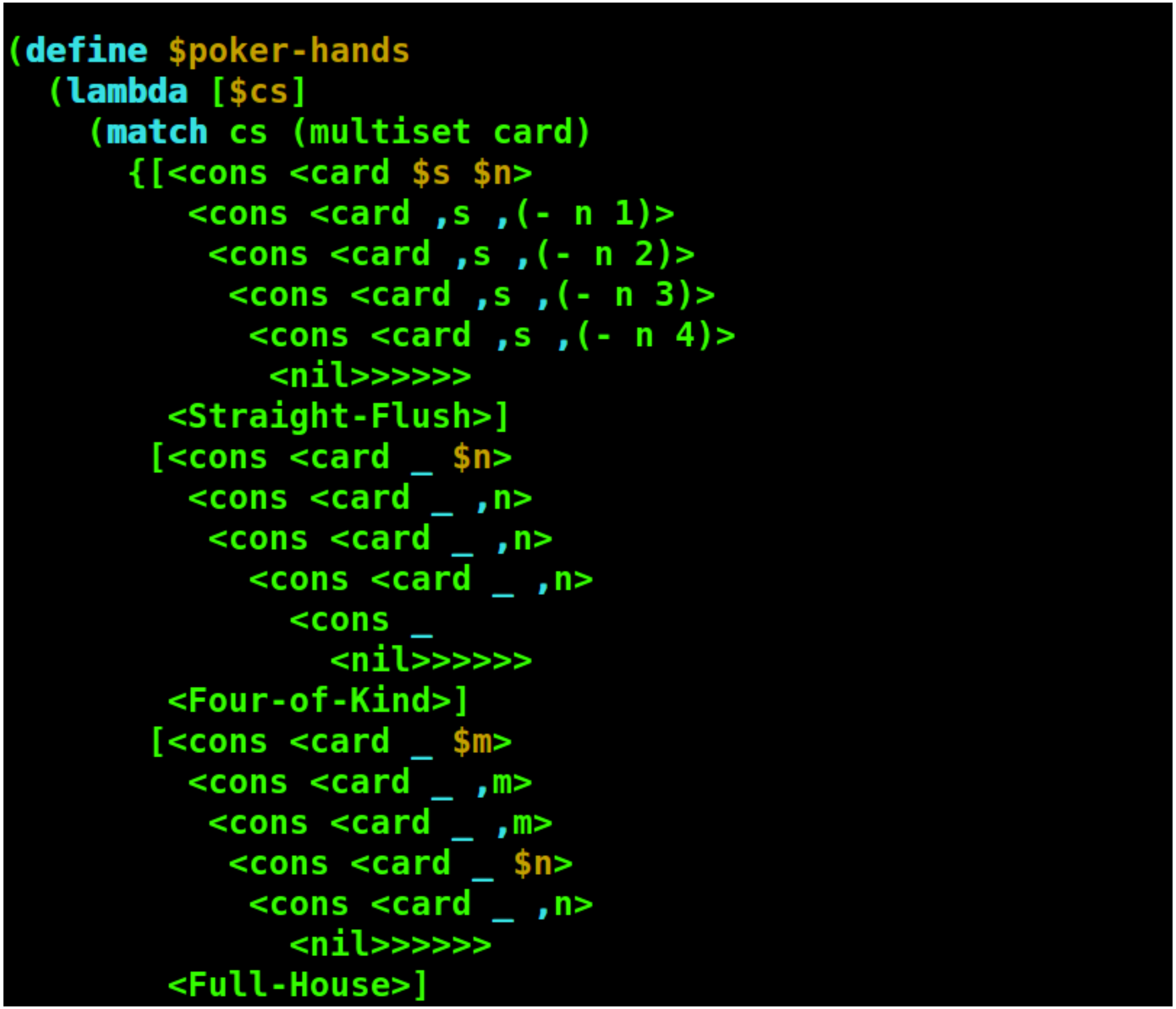
```
for (…) {
  for (…) {
    if (xs[i] == xs[j]) return xs[i]
    …
  }
}
```

is replaced with



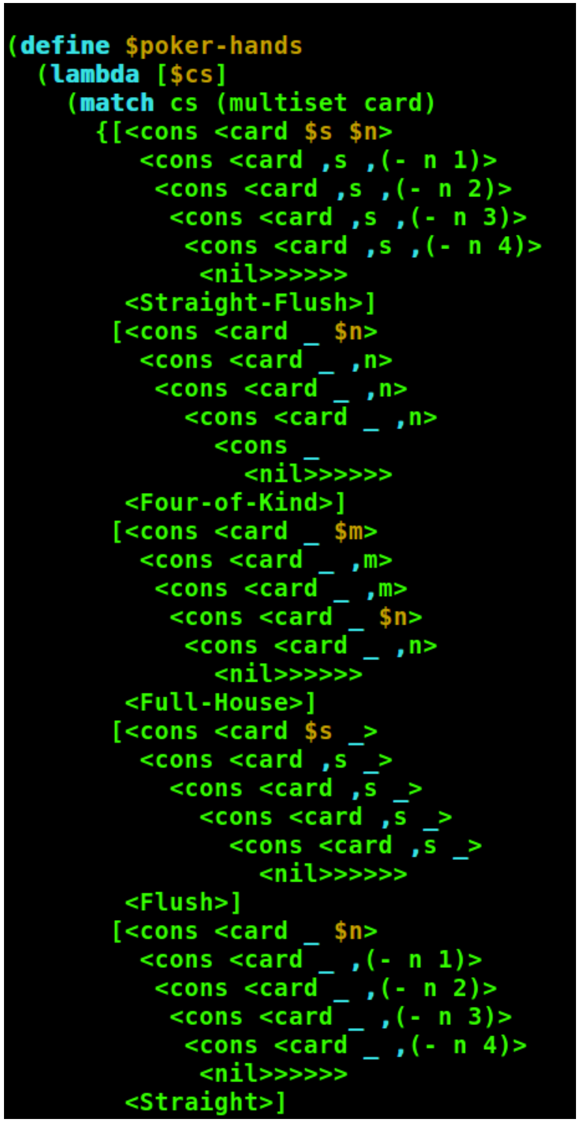
```
(match-all xs (multiset integer)
  [<cons $x <cons ,x _>> x])
```

Demo of Egison

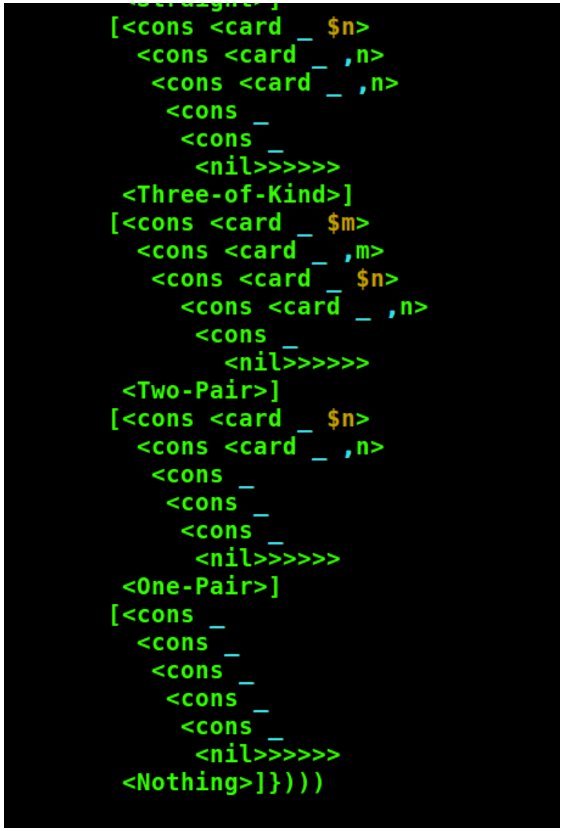


```
(define $poker-hands
  (lambda [$cs]
    (match cs (multiset card)
      {[<cons <card $s $n>
         <cons <card ,s ,(- n 1)>
          <cons <card ,s ,(- n 2)>
           <cons <card ,s ,(- n 3)>
            <cons <card ,s ,(- n 4)>
             <nil>>>>>>
        <Straight-Flush>]
       [<cons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

Demo 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>]})))
```

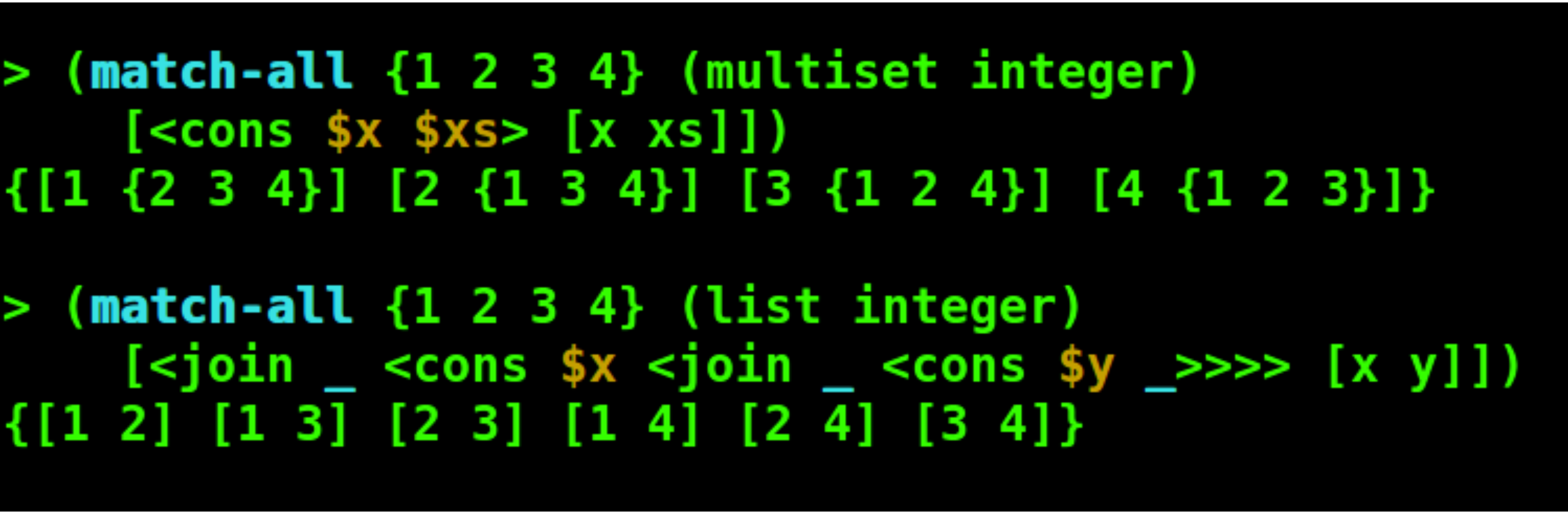
All hands are represented in a single pattern!

Features of Egison Pattern-Matching

1. Multiple results
2. Non-linear patterns
3. Modularization of matchers
4. Modularization of patterns

Pattern-Matching with Multiple Results

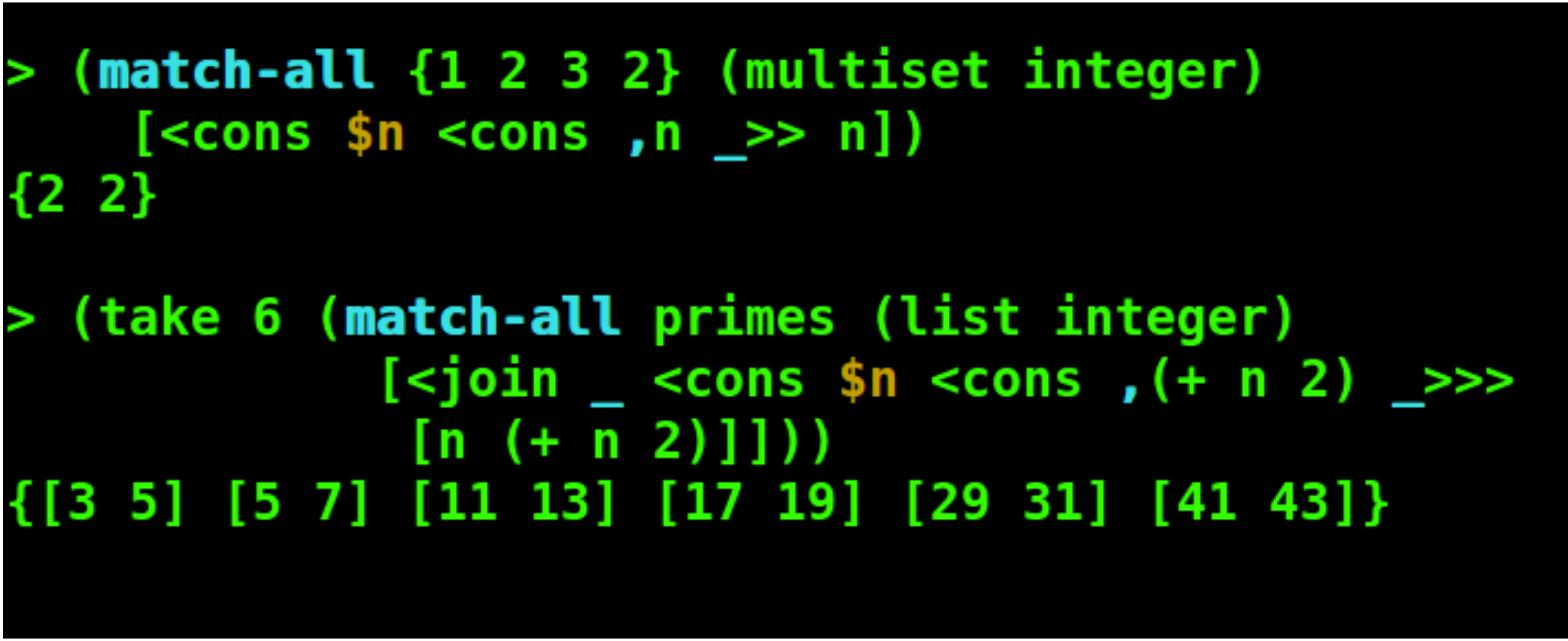
- Egison can deal with multiple results of pattern-matching.



```
> (match-all {1 2 3 4} (multiset integer)
    [<cons $x $xs> [x xs]])
{[1 {2 3 4}] [2 {1 3 4}] [3 {1 2 4}] [4 {1 2 3}]}

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

Non-Linear Pattern Matching

- A variable can appear more than once in a single pattern.

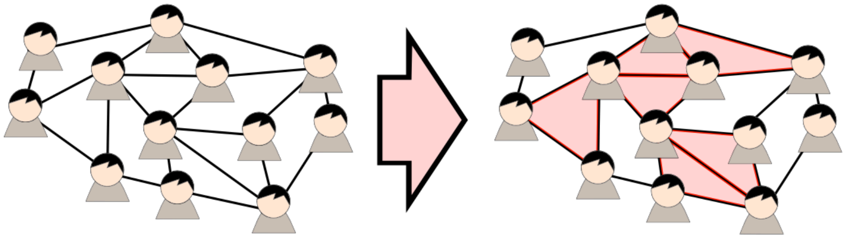


```
> (match-all {1 2 3 2} (multiset integer)
    [<cons $n <cons ,n _>> n])
{2 2}

> (take 6 (match-all primes (list integer)
            [<join _ <cons $n <cons ,(+ n 2) _>>>
             [n (+ n 2)]]))
{[3 5] [5 7] [11 13] [17 19] [29 31] [41 43]}
```

Programmer can define Matchers

- We can modularize a way of pattern-matching for each data type.
  - list, multiset, set
  - compact-list
  - mod
  - graph
- We can pattern-match against not only sets.

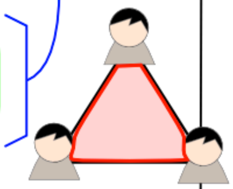
Pattern-Matching against Graphs



Get all combinations a friend of a friend is a friend

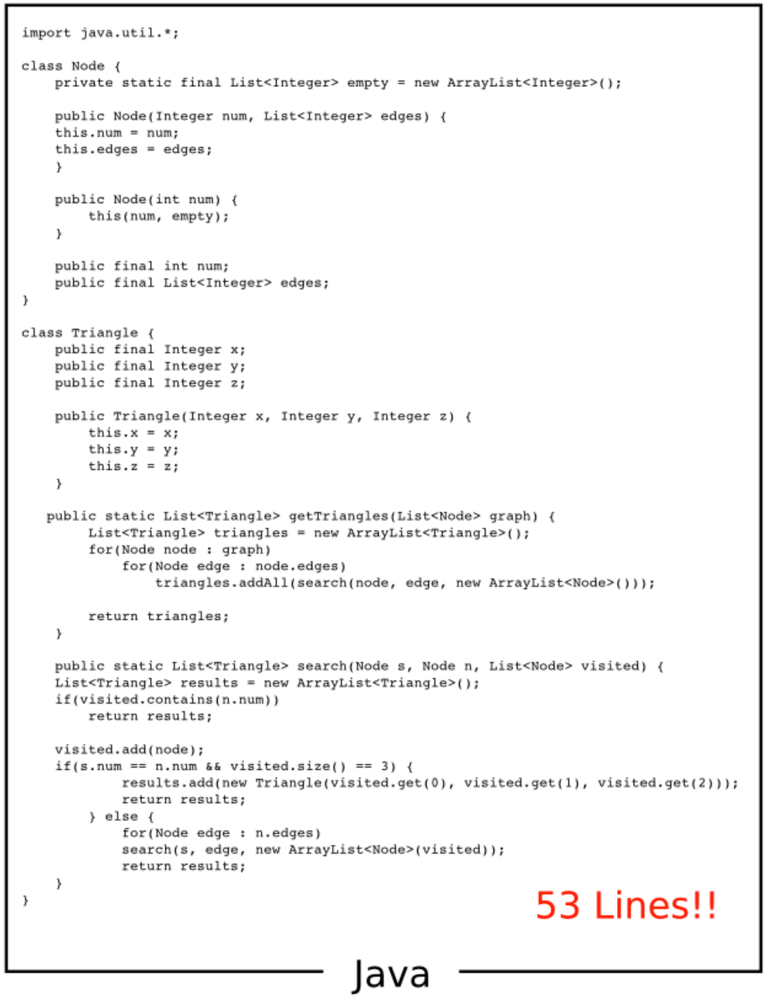
```
type nodeInfo = <node integer (multiset node)>

graph = (multiset nodeInfo)

triangles g = match-all g graph
  <cons <node $n_1 <cons $n_2 _>>
   <cons <node ,n_2 <cons $n_3 _>>
    <cons <node ,n_3 <cons ,n_1 _>>
     _>>> :
  {n_1 n_2 n_3}
```

Pattern that matches with a circuit length 3



Only 10 lines!!

Egison



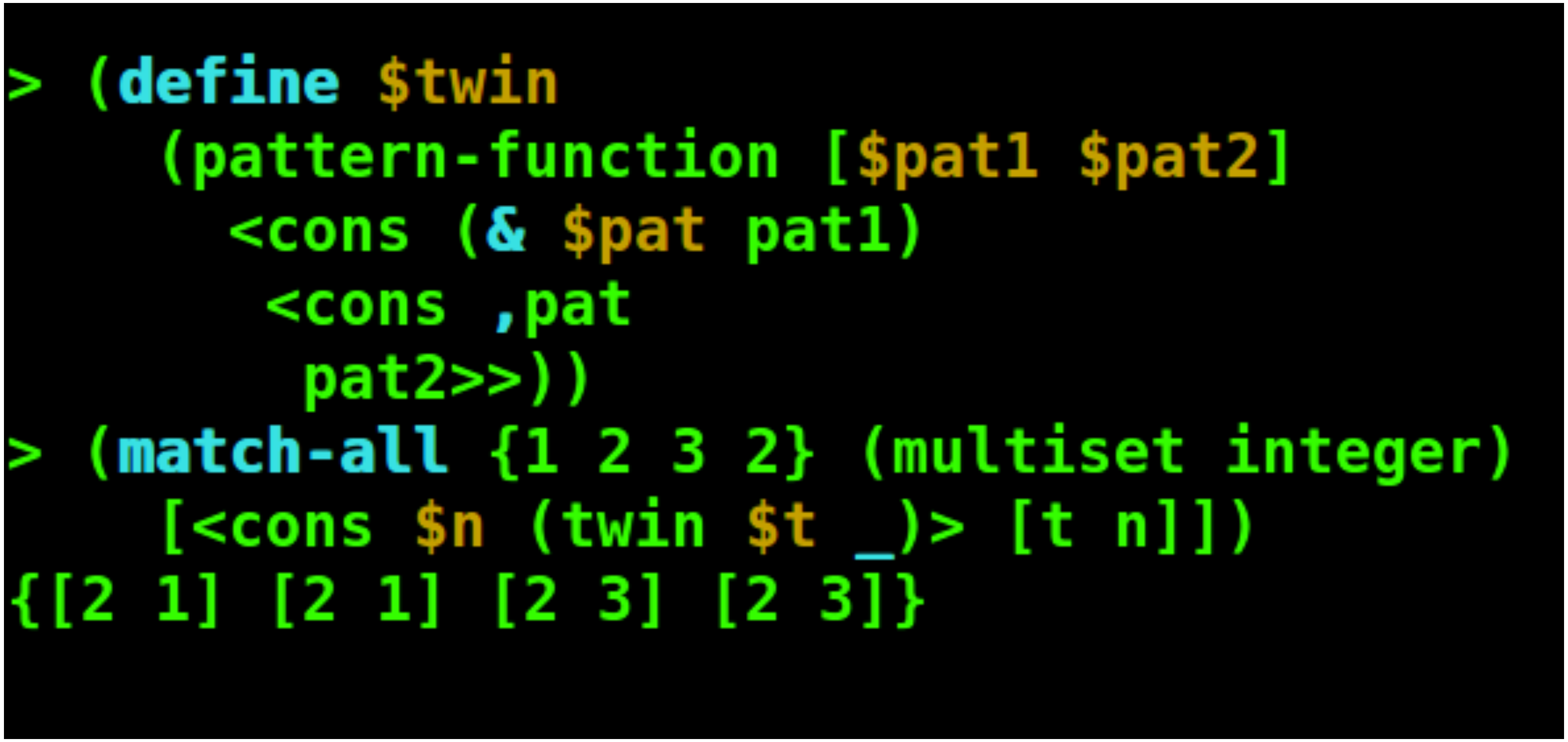
```
import java.util.*;

class Node {
    private static final List<Integer> empty = new ArrayList<Integer>();

    public Node(Integer num, List<Integer> edges) {
    this.num = num;
    this.edges = edges;
    }

    public Node(int num) {
        this(num, empty);
    }

    public final int num;
    public final List<Integer> edges;
}

class Triangle {
    public final Integer x;
    public final Integer y;
    public final Integer z;

    public Triangle(Integer x, Integer y, Integer z) {
        this.x = x;
        this.y = y;
        this.z = z;
    }

   public static List<Triangle> getTriangles(List<Node> graph) {
        List<Triangle> triangles = new ArrayList<Triangle>();
        for(Node node : graph)
            for(Node edge : node.edges)
                triangles.addAll(search(node, edge, new ArrayList<Node>()));

        return triangles;
    }

    public static List<Triangle> search(Node s, Node n, List<Node> visited) {
    List<Triangle> results = new ArrayList<Triangle>();
    if(visited.contains(n.num))
        return results;

    visited.add(node);
    if(s.num == n.num && visited.size() == 3) {
            results.add(new Triangle(visited.get(0), visited.get(1), visited.get(2)));
            return results;
        } else {
            for(Node edge : n.edges)
            search(s, edge, new ArrayList<Node>(visited));
            return results;
    }
}
```

53 Lines!!

Java

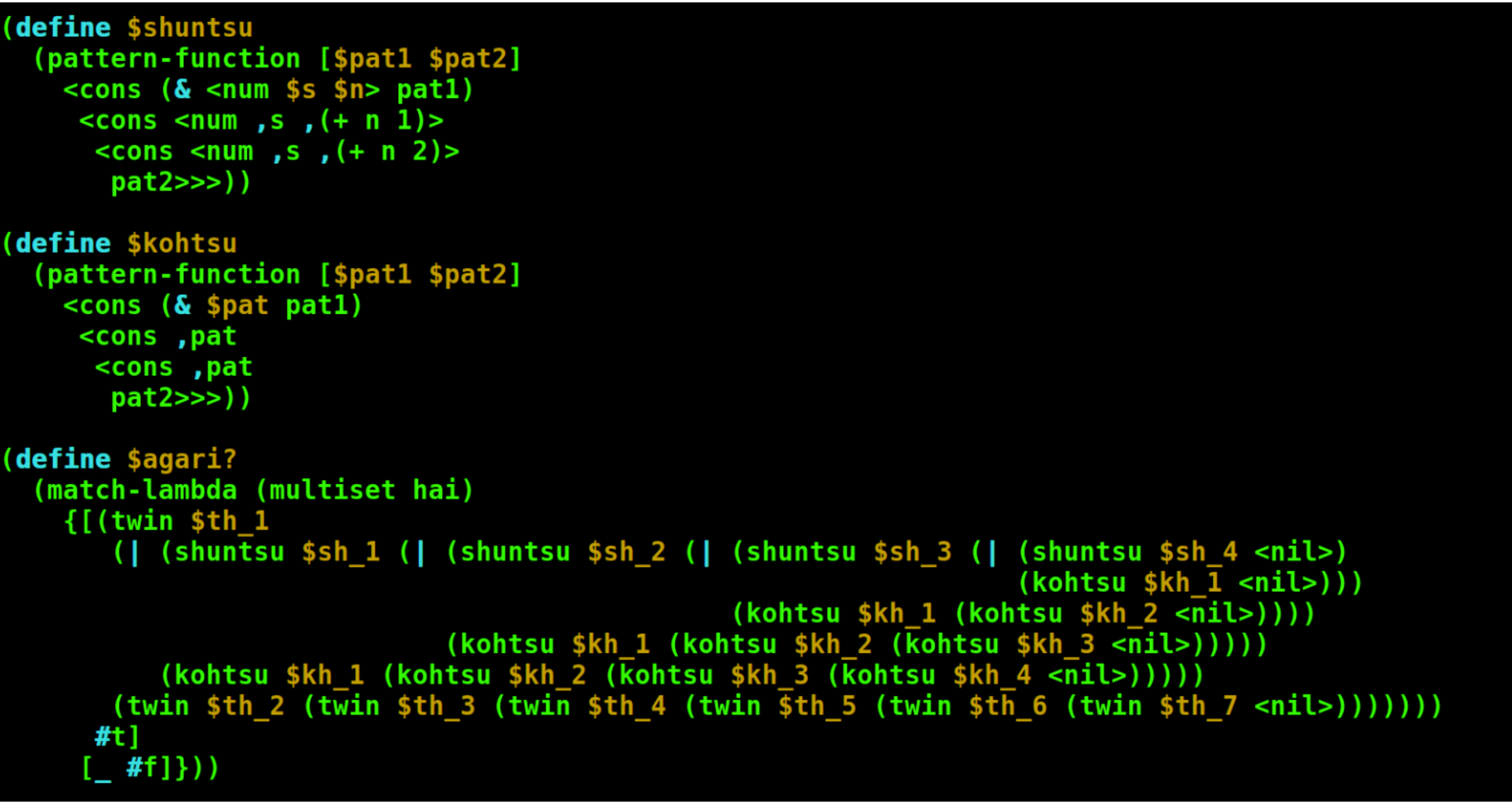
Modularization of Patterns

- We can modularize useful patterns.
- Patterns have lexical scoping.



```
> (define $twin
    (pattern-function [$pat1 $pat2]
      <cons (& $pat pat1)
       <cons ,pat
        pat2>>))
> (match-all {1 2 3 2} (multiset integer)
    [<cons $n (twin $t _)> [t n]])
{[2 1] [2 1] [2 3] [2 3]}
```

Demo of Mahjong



```
(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>>>))

(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]}))
```

Comparison with Other Work

	Views [1]	Active Patterns [2]	First Class Patterns [3]	Egison
Multiple Results	Not supported	Not supported	Perfect	Perfect
Non-Linear Patterns	Not supported	Partially supported	Not supported	Perfect
Matcher Definition	Partially supported	Partially supported	Partially supported	Perfect

[1] P. Wadler. Views: A way for pattern matching to cohabit with data abstraction. In Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, page 313. ACM, 1987.

[2] M. Erwig. Active patterns. Implementation of Functional Languages, pages 21–40, 1996.

[3] M. Tullsen. First Class Patterns. Practical Aspects of Declarative Languages, pages 1–15, 2000.

Future of Egison

- Egison is **one of the most innovative ideas** in the history of Computer Science and Information Technology.
- Pattern-matching of Egison has wide area of application.

Application of Egison

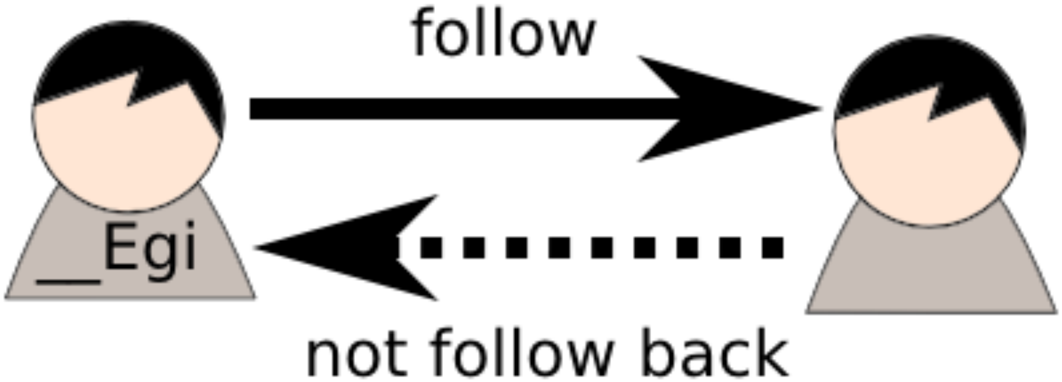
- Big data analysis with the strong expressive power of pattern-matching.
- High performance computing by automatic parallelization of programs.

Big Data Analysis

- It is natural to represent database access as pattern-matching against sets.
  - Egison Query Language
    - We can represent a complex query in a simple way with EgisonQL.
      - No complex where clauses
      - No subquery (query in query)
  - Other Database like GraphDB
    - No standard query language yet, as Relational Database.
    - We can get standard with pattern-matching of Egison.

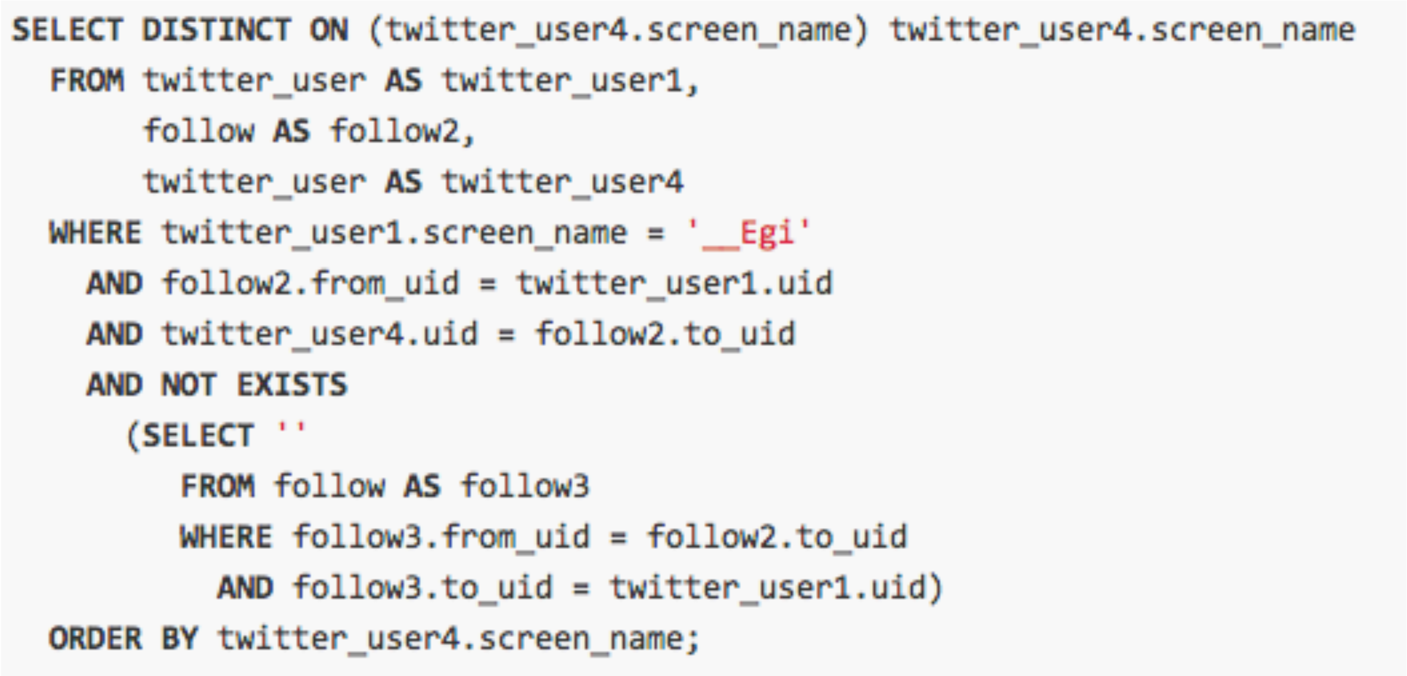
Demo of EgisonQL

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



SQL Version

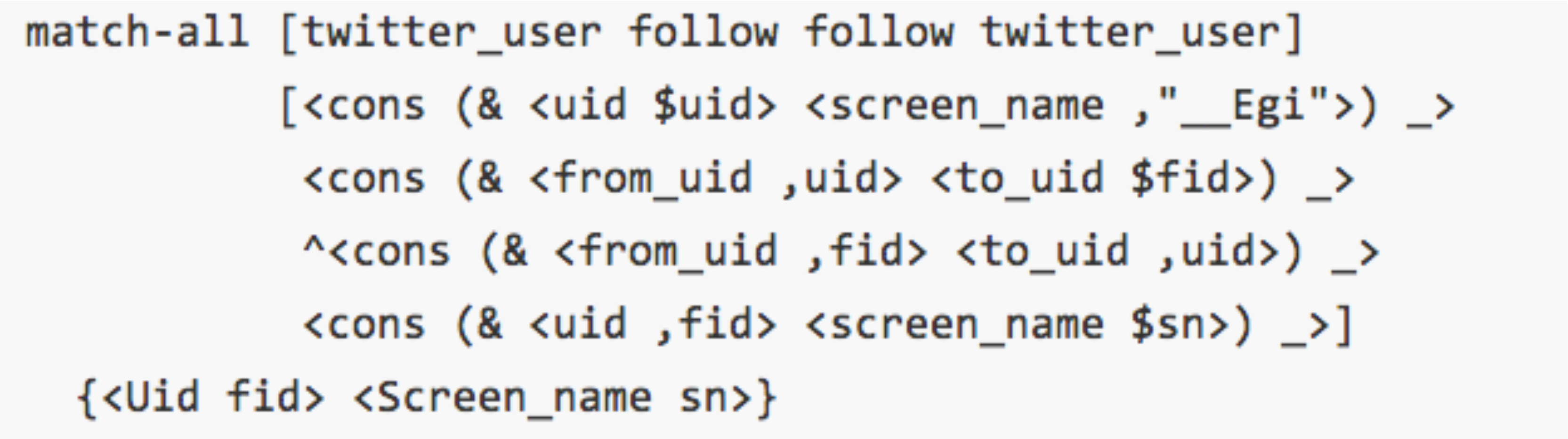
- Complex and difficult to understand
  - Complex where clause contains “NOT EXIST”
  - Subquery



```
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;
```

EgisonQL Version

- Very Simple
  - No where clauses
  - No subquery



```
match-all [twitter_user follow follow twitter_user]
          [<cons (& <uid $uid> <screen_name ,"__Egi">) _>
           <cons (& <from_uid ,uid> <to_uid $fid>) _>
           ^<cons (& <from_uid ,fid> <to_uid ,uid>) _>
           <cons (& <uid ,fid> <screen_name $sn>) _>]
  {<Uid fid> <Screen_name sn>}
```

Application for E-Commerce (1)

- Any analysis can be done immediately.
  - Recommendation
    - Make users notice what they want
  - Trend analysis
    - What is selling well now?
    - Why the trend occurred?

Application for E-Commerce (2)

- I'd like to create intuitive GUI for analysis.
  - Even a **non-engineer** can do analysis easily.
    - People who know a market really well, but can't write a program can analyze the market.
  - Even **users** themselves can analyze themselves easily.
    - Users can know items that they will want even before they knew the items.

High Performance Computing

- Not only easy to write. Egison can run faster!
- Automatic Parallel Computing
  - Egison is purely functional programming language.
  - Parallelizable loops are written using pattern-matching in Egison.
  - Pattern-Matching process of Egison is automatic parallelizable.
    - More detail, please read "Pattern-Matching Mechanism of Egison" (http://www.egison.org/manual3/mechanism.html)
- More abstract a programming language is, easier to analyze a program and more able to optimize it.

References

[1] P. Wadler. Views: A way for pattern matching to cohabit with data abstraction. In Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, page 313. ACM, 1987.

[2] M. Erwig. Active patterns. Implementation of Functional Languages, pages 21–40, 1996.

[3] M. Tullsen. First Class Patterns. Practical Aspects of Declarative Languages, pages 1–15, 2000.

Appendix. Why I Created Egison

- I'd like to formalize reasoning and run it on computer.
  - A program that automatically proposes hypothesis of number theory and geometry and prove them
- I'd like to formalize recognition of human to represent on computer.
  - Existing languages cannot treat important data in mathematics as sets, directly.
  - It is impossible to formalize reasoning on such representation.
- I created the programming language that can treat data like sets, intuitively.