プログラミング言語Egisonのコンパイラの開発

江木聡志

2012/2/4

自己紹介

- 江木聡志
- 所属
  - 東京大学大学院情報理工学研究科コンピュータサイエンス専攻修士2年 萩谷研究室
  - Http://hagi.is.s.u-tokyo.ac.jp/~egi/
- 研究分野
  - プログラミング言語
- Twitter
  - @__Egi
- 得意な言語
  - Egison, Haskell, Scheme

優れたプログラミング言語とは

- アルゴリズム記述の際の冗長性を排除
  - 人間が頭の中では当たり前に行っているのにも関わらず，プログラムでは当たり前に書けない処理があれば，そこが冗長性のある部分
- 解決されてきた冗長性
  - 関数型言語：似た関数の記述
  - オブジェクト指向：似たデータ型の定義の記述
  - ガベージコレクション：メモリ管理の記述
  - Egison：パターンマッチ（データの分解）の記述

既存のパターンマッチの問題点

- 既存のプログラミング言語では集合やマルチセットなど正規形を持たないデータのパターンマッチを自然に表現できない
  - 正規形とは同じデータに対する1つの標準的な表現のこと
  - 既存の言語では，集合やマルチセットなどをパターンマッチする際，いちいちリストに変換してパターンマッチしなければならない
- 正規形を持たないデータ型に対してのパターンマッチの一般的な方法をモジュール化する方法が必要！！

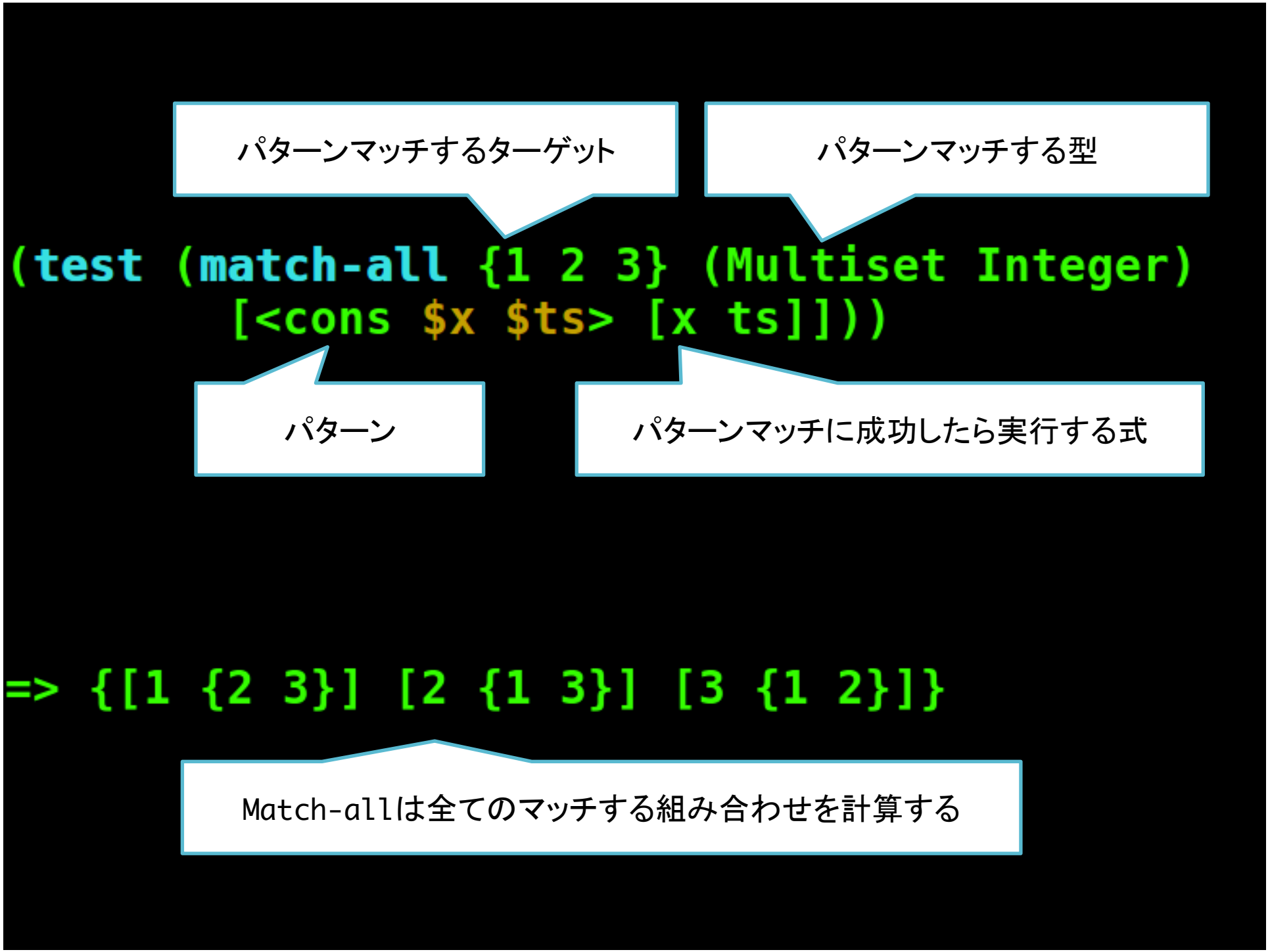
Egisonとは

- 江木が設計、実装したプログラミング言語
  - 純粋関数型言語
  - 強力なパターンマッチ機能が特徴

- インタプリタがHaskellで実装されていて、Hackageを用いて公開されている
- 専用のEmacs major modeが用意されている

Egisonのパターンマッチ

- 複数の結果を持つパターンマッチ
  - 複数の形を持つ正規形を持たないデータに対するパターンマッチには必須
- パターンマッチの際に，同じパターン内で先に束縛された変数の値を参照できる（non-left-linear）
  - 意味のあるパターンを表現するのに必須
- パターンマッチの際の探索を制御できる
  - パターンマッチの際に探索において，明らかに無駄な探索をしないようにすること必要
- …他にもいろいろ


パターンマッチするターゲット
パターンマッチする型
(test (match-all {1 2 3} (Multiset Integer)
[<cons $x $ts> [x ts]]))
パターン
パターンマッチに成功したら実行する式
=> {[1 {2 3}] [2 {1 3}] [3 {1 2}]}
Match-allは全てのマッチする組み合わせを計算する

```
(test (match-all {1 2 3} (List Integer)
        [<cons $x $ts> [x ts]]))

=> {[1 {2 3}]}
```

リストとして
パターンマッチ

```
(test (match-all {1 2 3} (Multiset Integer)
        [<cons $x $ts> [x ts]]))

=> {[1 {2 3}] [2 {1 3}] [3 {1 2}]}
```

マルチセットとして
パターンマッチ

```
(test (match-all {1 2 3} (Set Integer)
        [<cons $x $ts> [x ts]]))

=> {[1 {2 3}] [2 {1 3}] [3 {1 2}]
    [1 {2 3 1}] [2 {1 3 2}] [3 {1 2 3}]}
```

集合として
パターンマッチ

Egisonのパターンマッチ

- 複数の結果を持つパターンマッチ
  - 複数の形を持つ正規形を持たないデータに対するパターンマッチには必須
- パターンマッチの際に，同じパターン内で先に束縛された変数の値を参照できる意味のあるパターンを表現するのに必須
- パターンマッチの際の探索を制御できる
  - パターンマッチの際に探索において，明らかに無駄な探索をしないようにすること必要
- …他にもいろいろ

```
(test (match-all {2 8 7 2 7} (Multiset Integer)
        [<cons $m <cons ,m _>> m]))

=> {2 7}
```

パターンの左側のパターン変数に束縛された値を参照できる

```
(test (match {5 2 1 3 4} (Multiset Integer)
        {[<cons $n
            <cons ,(- n 1)
              <cons ,(- n 2)
                <cons ,(- n 3)
                  <cons ,(- n 4)
                    <nil>>>>>>
          <ok>]
         [_ <ko>]}))

=> <ok>
```

パターンの左側で得られた束縛を環境に追加して評価される

Egisonのパターンマッチ

- 複数の結果を持つパターンマッチ
  - 複数の形を持つ正規形を持たないデータに対するパターンマッチには必須
- パターンマッチの際に，同じパターン内で先に束縛された変数の値を参照できる（non-left-linear）
  - 意味のあるパターンを表現するのに必須
- パターンマッチの際の探索を制御できる
  - パターンマッチの際に探索において，明らかに無駄な探索をしないようにすること必要
- …他にもいろいろ

```
(define $fullhouse
  (lambda [$xs]
    (match xs (Multiset Integer)
      {[<cons $m
          <cons ,m
            <cons ,m
              <cons $n
                !<cons ,n
                   !<nil>>>>>>
          <true>]
        [_ <false>]})))
```

ここまでパターンマッチに成功したらこれ以降はバックトラックしない

ターゲットのコレクションが3つと同じ要素を持ち、
さらに残り2つの要素も同じであればパターンマッチに成功する

```haskell
fullhouse :: [Int] -> Bool
fullhouse xs = fullhouse2 $ sort xs

fullhouse2 :: [Int] -> Bool
fullhouse2 (x1:(x2:(x3:(x4:[x5]))))
  | (x1 == x2) && (x1 == x3) && (x4 == x5) = True
  | (x1 == x2) && (x3 == x4) && (x4 == x5) = True
  | otherwise = False
fullhouse2 _ = False
```

Egisonのパターンマッチ

- 複数の結果を持つパターンマッチ
  - 複数の形を持つ正規形を持たないデータに対するパターンマッチには必須
- パターンマッチの際に，同じパターン内で先に束縛された変数の値を参照できる（non-left-linear）
  - 意味のあるパターンを表現するのに必須
- パターンマッチの際の探索を制御できる
  - パターンマッチの際に探索において，明らかに無駄な探索をしないようにすること必要
- …他にもいろいろ

```
(test (match-all {{1 2 3 4 5} {4 5 1} {6 1 7 4}}
        (List (Multiset Integer))
        [<cons <cons $n _>
          <cons <cons ,n _>
           <cons <cons ,n _>
            <nil>>>>>
         n]))
```

ネストした型のパターンマッチも簡単に行える

```
=> {1 4}
```

```
(define $min
  (lambda [$ns]
    (match ns (List Integer)
      {[<cons $n <nil>> n]
       [<cons $n $rs>
        (let {[$r (min rs)]}
          (match (compare-integer n r) Order
            {[<less> n]
             [_ r]}))]})))

(define $gcd
  (lambda [$ns]
    (match ns (Set Integer)
      {[<cons $n <nil>> n]
       [<cons (& ,(min ns) $m)
              $rs>
        (gcd {m @((remove-all Integer)
                  (map (lambda [$r] (mod r m)))
                  rs)
                 0})]})))
```

正規形を持たないデータは別々の場所でそれぞれ別々の型として扱われることが多い

⟨pattern⟩ ⟶

_
| $v
| <⟨pattern constructor⟩ ⟨pattern⟩*>
| [⟨pattern⟩*]
| (? ⟨predicate⟩ ⟨pattern⟩*)
| (& ⟨pattern⟩*)
| (| ⟨pattern⟩*)
| !⟨pattern⟩

,⟨expression⟩ = (? equal? ⟨expression⟩)

Egisonの型の定義

- パターンとデータを引数に取り，全ての可能な束縛を計算するマッチ関数なるものを型ごとにユーザは記述する
  - 例．Multisetのマッチ関数は．パターン<cons $x $xs>，ターゲット{1 2 3}の場合，以下のような結果を返す
    - `{[x 1] [xs {2 3}]},{[x 2] [xs {1 3}]},`
      `{[x 3] [xs {1 2}]}`
- マッチ関数の記述のために以下の関数を定義する
  - Var-match: パターンが変数パターンである場合にパターンマッチを行う関数
  - Inductive-match: パターンが帰納的に構成されたパターンである場合にパターンマッチを行う関数
  - その他predicate-patternで使うための述語

```
(define $Multiset
  (lambda [$a]
    (type
      {[$var-match (lambda [$tgt] {tgt})]
       [$inductive-match
        (deconstructor
          {[nil []
            {[{} {[]}]
             [_ {}]]}]
          [cons [a (Multiset a)]
           {[$tgt (map (lambda [$t] [t ((remove a) tgt t)])
                       tgt)]}]
          [join [(Multiset a) (Multiset a)]
           {[$tgt (map (lambda [$ts]
                         [ts ((remove-collection a) tgt ts)])
                       (subcollections tgt))]}]})]
       [$equal? (lambda [$val $tgt]
                  (match [val tgt] [(Multiset a) (Multiset a)]
                    {[[<nil> <nil>] <true>]
                     [[<cons $x $xs> <cons ,x ,xs>] <true>]
                     [[_ _] <false>]}))]})))
```

Multisetは型aを受け取って型を返す関数

(Multiset Int) : 整数のマルチセット
(Multiset (Multiset Int)) :
　整数のマルチセットのマルチセット

```
(define $Multiset
  (lambda [$a]
    (type
      {[$var-match (lambda [$tgt] {tgt})]
       [$inductive-match
        (deconstructor
          {[nil []
            {[{} {[]}]
             [_ {}]}]
           [cons [a (Multiset a)]
            {[$tgt (map (lambda [$t] [t ((remove a) tgt t)])
                        tgt)]}]
           [join [(Multiset a) (Multiset a)]
            {[$tgt (map (lambda [$ts]
                          [ts ((remove collection a) tgt ts)])
                        (subcollections tgt))]}]}
          [$equal? (lambda [$val $tgt]
                     (match [val tgt] [(Multiset a) (Multiset a)]
                       {[[<nil> <nil>] <true>]
                        [[<cons $x $xs> <cons ,x ,xs>] <true>]
                        [[_ _] <false>]}))]})))
```

パターンコンストラクタnilは引数を取らない

1つ目の引数は型aとして，2つ目は(Multiset a)として再帰的にパターンマッチ

1つ目のと2つ目の引数を両方とも(Multiset a)として再帰的にパターンマッチ

```
(define $Multiset
  (lambda [$a]
    (type
      {[$var-match (lambda [$tgt] {tgt})]
       [$inductive-match
        (deconstructor
          {[nil []
            {[{} {[]}]
             [_ {}]]}]
          [cons [a (Multiset a)]
           {[$tgt (map (lambda [$t] [t ((remove a) tgt t)])
                       tgt)]}]
          [join [(Multiset a) (Multiset a)]
           {[$tgt (map (lambda [$ts]
                         [ts ((remove-collection a) tgt ts)])
                       (subcollections tgt))]}]})]
       [$equal? (lambda [$val $tgt]
                  (match [val tgt] [(Multiset a) (Multiset a)]
                    [[<nil> <nil>] <true>]
                    [[<cons $x $xs> <cons ,x ,xs>] <true>]
                    [_ _] <false>]))]})))
```

ターゲットが{1 2 3}である場合

{}を返す

{[1 {2 3}] [2 {1 3}] [3 {1 2}]}を返す

{[{} {1 2 3}] [{1} {2 3}] [{2} {1 3}]
{[3 {1 2}] [{1 2} 3] [{1 3} 2] [{2 3}
1] [{1 2 3} {}]}を返す

```
(test (match-all {1 2 3} (Multiset Integer)
        [<cons $x $xs> [x xs]]))

=> {[1 {2 3}] [2 {1 3}] [3 {1 2}]}
```

1つの要素と残りのコレクションとに分割するパターンマッチ

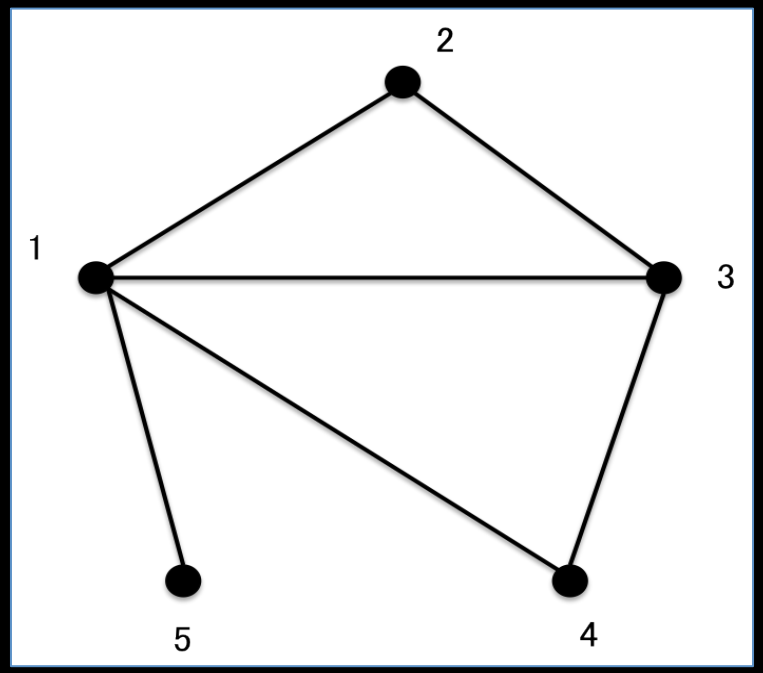
```
(test (match-all {1 2 3} (Multiset Integer)
        [<join $xs $ys> [xs ys]]))

=> {[{} {1 2 3}] [{3} {1 2}] [{2} {1 3}] [{2 3} {1}]
    [{1} {2 3}] [{1 3} {2}] [{1 2} {3}] [{1 2 3} {}]}
```

2つのコレクションに分割するパターンマッチ

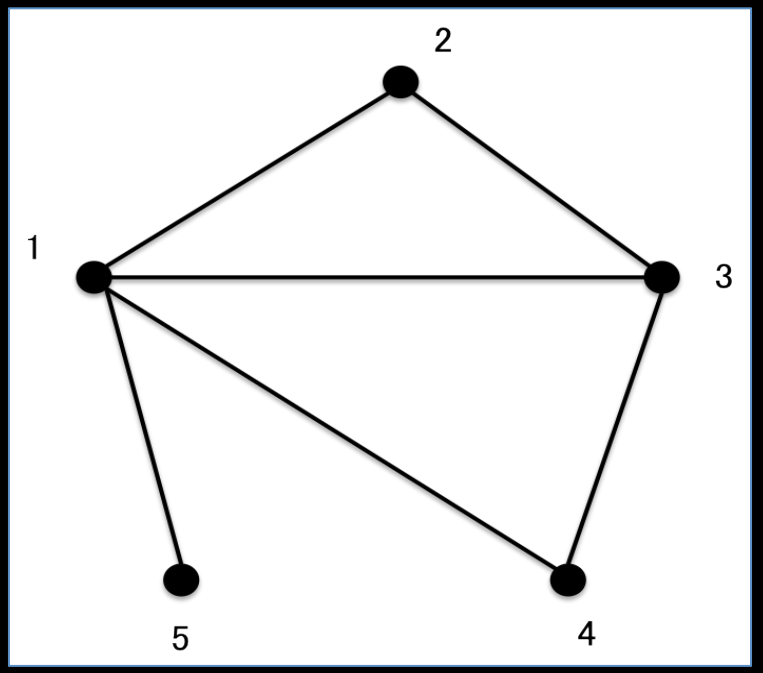
応用例

- ポーカーの役判定
  - 集合のパターンマッチが直接表現できるおかげで簡潔に記述できる
  - 麻雀も簡単に書ける
- グラフのパターンマッチ
  - グラフはノードとエッジの多重集合として表現できる
- 電子回路の変換
  - 端子への入力は多重集合として表現できる
- …

```
(define $g {<node 1 {2 3 4 5} {2 3 4 5}>
            <node 2 {1 3} {1 3}>
            <node 3 {1 2 4} {1 2 4}>
            <node 4 {1 3} {1 3}>
            <node 5 {1} {1}>})
```

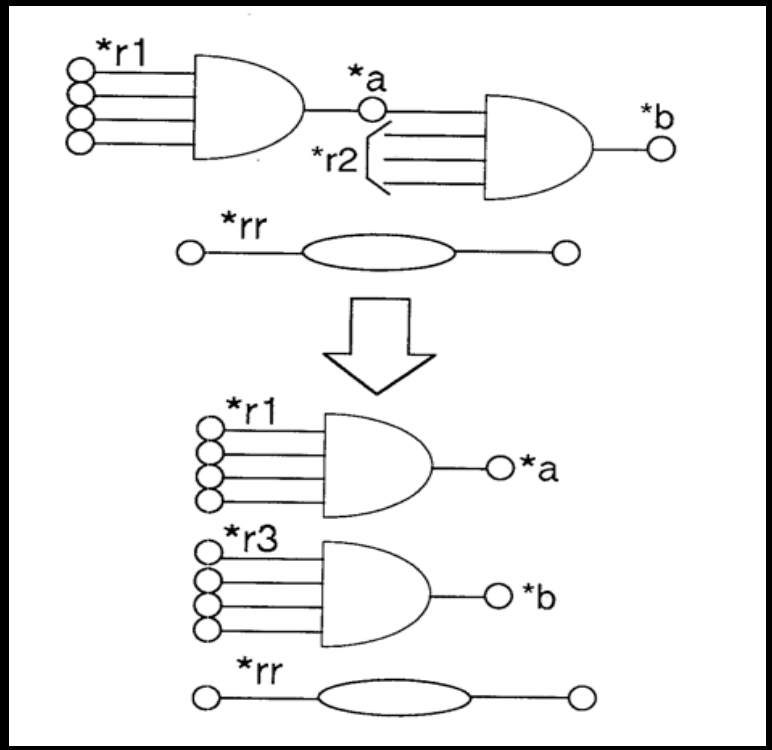


```
(test (match-all g Graph
        [<cons <node $n1 <cons $n2 _> _>
          <cons <node ,n2 <cons $n3 _> _>
           <cons <node ,n3 <cons ,n1 _> _>
            _>>>
         [n1 n2 n3]]))
```




```
=> {[1 2 3] [1 3 2] [1 3 4] [1 4 3] [2 1 3] [2 3 1]
    [3 1 2] [3 1 4] [3 2 1] [3 4 1] [4 1 3] [4 3 1]}
```

```
(define andAnd
  (lambda ($c)
    (match c Circuit
      {[<cons <and-gate $a $r1>
         <cons <and-gate $b <cons $a $r2>>
          $rr>>
        <cons <and-gate a r1>
         <cons <and-gate b (union r1 r2)>
          rr>>]
       [_ c]})))
```



Egisonの現状

- インタプリタを実装済み
  - Hackageを使ってフリーで配布中
  - インストール方法
    - http://hagi.is.s.u-tokyo.ac.jp/~egi/egison/manual/setting-up-egison.html
- マニュアルを公開中
  - 日本語でも英語でも公開中
- 修士論文はEgisonについて書かれた
- 現在、Egisonistは5人

Egisonの学術的な貢献

- 複数の結果を持つパターンマッチをnon-linearなパターンにも対応させた
  - 左から順にパターンマッチが行われる
  - 左側で束縛されたパターン変数の値を参照できる
- そのパターンマッチに探索を制御する仕組みを追加した
  - カットパターン
  - これも左側からパターンマッチが行われることを利用している

プロモーション

- プログラミングコンテスト
  - Egisonを使ってコンテストに参加してもらう
    - Google Code Jam 2012に参加してもらう
    - 上位者には賞品(賞金)が！？
  - Egisonプログラミングコンテストを開催
- Egisonのワークショップを開く
- EgisonのTwitterアカウント
  - @Egison_Lang

使える言語Egisonを目指す

- プログラミング言語として最低限の機能は全て実装する
  - システムへのアクセス(シェルコマンド，ファイル，ネットワーク)
  - Egisonのためのパッケージシステム(面白いソフトやライブラリをユーザにも開発，共有してもらう)
- 有名になる
  - 多くのひとに使ってもらうため
- コンパイラを作る
  - 速い処理系にするため