IPA

2011 年度 未踏 IT 人材発掘・育成事業 採択案件評価書

1. 担当PM

原田 康徳 PM
（日本電信電話株式会社 NTT コミュニケーション科学基礎研究所 主任研究員）

2. 採択者氏名

チーフクリエータ：江木 聡志
（東京大学大学院 情報理工学系研究科
コンピュータ科学専攻 萩谷研究室 修士 2 年）

3. 委託金支払額

1,792,000 円

4. テーマ名

プログラミング言語 Egison のコンパイラの開発

5. 関連Webサイト

http://hagi.is.s.u-tokyo.ac.jp/~egi/egison/index-j.html

6. テーマ概要

強力なパターンマッチ機能をもつプログラミング言語 Egison のコンパイラを提案する。

Egison は提案者が設計、開発したプログラミング言語である。(http://hagi.is.s.u-tokyo.ac.jp/~egi/egison/index-j.html) Egison を使うと、正規形を持たないデータ、例えば、集合や多重集合などといったコレクションや、また環や群とい

った代数構造などのパターンマッチを直感的に表現することができる。

正規形というのは、同じデータを表現する 1 つの決まった標準的な形のことをいう。

これらの正規形を持たないデータ型のデータのパターンマッチを行うには、一度これらのデータを正規形を持つデータ型として捉え直して、パターンマッチを行うことが必要である。

例えば、既存のプログラミング言語では集合のパターンマッチを行うさい、これをリストとして捉え直してパターンマッチを行う。

多くのプログラマは、この煩雑な作業を当たり前のことだと認識しているが、これは実はかなりのプログラミングの際の潜在的な精神的なストレスになっている。

Egison では、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を用意することによってこの問題を解決した。

Egison のインタプリタは既に実装されていて、公開されている。Egison は Hackage(Haskell で書かれたソフトウェアのためのパッケージシステム)のパッケージとして配布されていて、Haskell の環境さえ整っていれば、誰でも無料で簡単にインストールすることができる。(http://hackage.haskell.org/package/egison)インタプリタのソースコードは、GitHub で公開されている。(https://github.com/egisatoshi/egison)

本提案の目標は、この Egison のコンパイラを作成することによって、この強力な表現能力を持ったプログラミング言語の実用に耐えうる処理系を実現することである。

7. 採択理由

このような新しいプログラミング言語の提案は、よい処理系を作ることも重要であるが、言語の特徴を広く認知させ、作者の想像を超えた使われ方をしてもらうことが、より重要である。

わかりやすいマニュアル、実施例など作成が必要である。応用を探るワークショップを開催してもよいかもしれない。

この言語の機能は 10 年後には世の中に当たり前になっているようなものを目指してほしい。

8. 開発目標

Egison とは、本プロジェクトのクリエータが 2010 年 3 月にアイデアを得て開発を始め、2011 年 5 月に最初のリリースをしたプログラミング言語である。

Egison の特徴はその強力なパターンマッチの記述力にある。Egison では 1 つの定まった形を持たないデータ型(正規形を持たないデータ型)についてのパターンマッチを直感的に表現することができる。Egison ではプログラマが集合や多重集合のような、

正規形を持たないデータ型のパターンマッチの方法を定義することが可能であり、それを使用したプログラムを直感的に記述することができる。

本プロジェクトの目的は、Egison を開発者やその知り合いに留まらずに広く一般に使える言語にまで完成度を引き上げ、Egison をより多くのユーザに普及させることを通して、パターンマッチの重要性を世に知らしめることである。

9. 進捗概要

(1) 言語仕様の改良

Egison の特徴であるパターンマッチの記述を強化するための言語仕様の拡張、改善を行った。また実用に耐えうる言語にするためにプリミティブ関数やライブラリ関数の追加などを行った。

プロジェクト期間中に行った一番大きな改良は、データごとのパターンマッチの方法の記述方法の仕様の改良である。2012 年 7 月 14 日にリリースされたバージョン 2.2.0 からこの改良は加えられた。

ユーザがデータ型ごとにパターンマッチの方法を記述することによって、Egison の複雑なパターンマッチは実現されている。そのデータ型定義の表現が簡潔になり、かつ強力になった。それによって、より詳細にパターンマッチの方法を定義できるようになった。

型定義の仕様は Egison のバージョンが上がるたびにしばしばあった。しかし最も顕著な仕様の改良をしたのは後述のワークショップの前後である。以下はバージョン 1.2.3(ワークショップ以前)の型定義である。

```
(define $Multiset
  (lambda [$a]
    (type
      {[$var-match (lambda [$tgt] {tgt})]
       [$inductive-match
        (destructor
          {[nil []
            {[{} {[]}]
             [_ {}]}]
           [cons [a (Multiset a)]
            {[$tgt (map (lambda [$t] [t ((remove a) tgt t)])
                        ((unique a) tgt))]}]
           [join [(Multiset a) (Multiset a)]
            {[$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>]}))]})))
```

図 1 Egison バージョン 1.2.3 での型定義

Multiset は型 a を受け取って型 a のマルチセットを返す関数として定義されている。例えば(Multiset Integer)は整数のマルチセットの型となる。

このころの型定義は var-match, inductive-match, equal という 3 つの関数を定義することで型ごとのパターンマッチの方法を定義していた。

var-match はパターンが変数である場合にパターンマッチに使われる。inductive-match はパターンが帰納的に構成されたものである場合に使われる。equal?はパターンがバリューパターンである場合に使われる。

var-match は引数としてターゲットを受け取り、パターン変数に束縛されうる値のコレクションを返す関数となっている。この例の場合だとターゲットと同じ値だけからなるコレクションを返すようになっている。以下は剰余環のデータ型定義の例である。

```
(define $Mod
  (lambda [$m]
    (type
      {[$var-match (lambda [$tgt] {(mod tgt m)})]
       [$equal? (lambda [$val $tgt] (= (mod val m) (mod tgt m)))]})))
```

図 2 剰余環のデータ型定義

mod 型は整数 m を受け取り、その整数 m の剰余環の型を返す関数として定義されている。例えば(Mod 13)は 13 の剰余環である。var-match はターゲットの値を m で割った剰余を返すようになっており、パターン変数にはその値が束縛される。

inductive-match は destructor 式を用いて定義される。destructor 式は destructor

マッチ節のコレクションを引数にとる。destructor マッチ節は 3 つの要素をとる。

1 つ目は、パターンコンストラクタの名前である。Multiset の場合では、nil, cons, join がある。

2 つ目は、パターンコンストラクタの引数のパターンをパターンマッチする型である。パターンコンストラクタの引数のパターンについて再帰的にパターンマッチが行われる。nil パターンコンストラクタは引数を取らないので、空のタプルがそこに入る。cons パターンコンストラクタの場合は、1 つ目の引数は型 a として、2 つ目の引数は型(Multiset a)として再帰的にパターンマッチが行われるので[a (Multiset a)]となる。join の場合は、1 つ目の引数も 2 つ目の引数も両方(Multiset a)としてパターンマッチが行われるので、[(Multiset a) (Multiset a)]となる。

3 つ目は、パターンコンストラクタの引数のパターンを再帰的にパターンマッチする際のターゲットを返すプリミティブマッチ節のコレクションである。2 つ目の引数で指定された型で再帰的にパターンマッチが行われる。

例として、図 1 に示した Multiset の定義の cons のプリミティブマッチ節をみる。プリミティブマッチ節は、プリミティブパターン(primitive pattern)と式のタプルとして表される。マッチ式のターゲットが、プリミティブパターンにマッチしたら、そのプリミティブマッチ節の式を評価した値が、再帰的に行われる次のパターンマッチのターゲットになる。図 1 の場合だと、型が(Multiset Integer)で、パターンが<cons $x $xs>という形のインダクティブパターンで、ターゲットが{1 2 3}というコレクションだった場合、inductive-match 関数を使って、パターンマッチの処理が行われる。その際、inductive-match 関数の cons のデコンストラクトマッチ節にマッチし、さらに、その中の唯一のプリミティブマッチ節にマッチします。

このプリミティブマッチ節の式を評価すると、{[1 {2 3}] [2 {1 3}] [3 {1 2}]}がその結果として返る。これのそれぞれ 1 つ目の要素を Integer として、2 つ目の要素を(Multiset Integer)として、再帰的にパターンマッチを行われる。

プリミティブパターンマッチでは、パターンマッチに成功する場合、可能な束縛結果が一意的に決まるようなパターンマッチしか行えない。プリミティブパターンマッチでは、人がプリミティブに行えるデコンストラクトを行えるようになっている。プリミティブパターンは図 3 に示す 7 つの要素から構成される。

```
primpat ::= _
          | patvar
          | < cons primpat ... &t;
          | [ primpat ... ]
          | {}
          | { primpat .primpat }
          | { .primepat primepat }
```

図 3 プリミティブパターンの要素

"_"は、ワイルドカードである。ターゲットが何でもパターンマッチが成功する。

pat-var はパターン変数である。ターゲットが何でもパターンマッチに成功し、値をその変数に束縛する。

<cons prime-pat ...>は、ターゲットが cons で指定されたデータコンストラクタで構成されたインダクティブデータである場合にパターンマッチを行う。データコンストラクタの引数については、それに対して再帰的にパターンマッチを行う。

[prime-pat ...]は、ターゲットがタプルの場合にパターンマッチを行う。タプルの中身の要素については、それに対して再帰的にパターンマッチを行う。

{}は、エンプティパターン(empty pattern)と呼ばれ、ターゲットが空のコレクションの場合にパターンマッチを行う。

{prime-pat . prime-pat}は、コンスパターン(cons pattern)と呼ばれ、ターゲットが 1 つ以上の要素を含むコレクションである場合にリストとしてパターンマッチを行い、1 つ目の prime-pat とターゲットの先頭の要素を、2 つ目の prime-pat とターゲットの残りの要素コレクションに対してパターンマッチを行う。

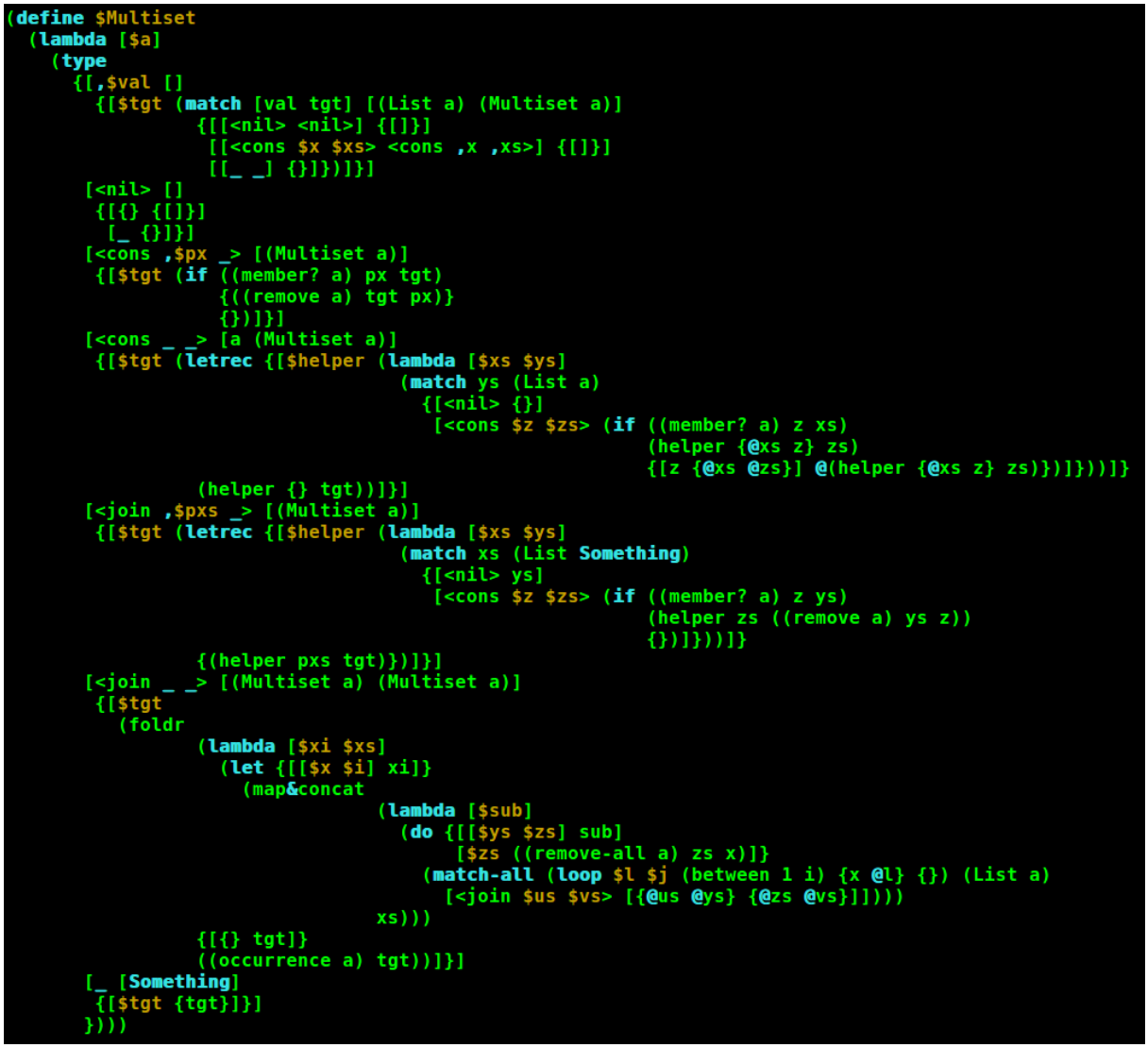
{.prime-pat prime-pat}は、スノックパターン(snoc pattern)と呼ばれ、ターゲットが 1 つ以上の要素を含むコレクションである場合にリストとしてパターンマッチを行い、2 つ目の prime-pat とターゲットの一番後ろの要素を、1 つ目の prime-pat とターゲットの残りの要素コレクションに対してパターンマッチを行う。

以下はリストの型定義の例である。リストの型定義ではタプルパターン以外の全てのプリミティブパターンが使われている。nil のプリミティブパターンマッチではエンプティパターンが、cons のプリミティブパターンマッチではコンスパターンが、snoc のプリミティブパターンマッチではスノックパターンが使われている。

```
(define $List
  (lambda [$a]
    (type
      {[$var-match (lambda [$tgt] {tgt})]
       [$inductive-match
        (destructor
          {[nil []
            {[{} {[]}]
             [_ {}]}]
           [cons [a (List a)]
            {[{$x .$xs} {[x xs]}]
             [_ {}]}]
           [snoc [a (List a)]
            {[{.$xs $x} {[x xs]}]
             [_ {}]}]
           [join [(List a) (List a)]
            {[$tgt (let {[$loop-fn
                          (lambda [$ts]
                            (match ts (List a)
                              {[<nil> {[{} {}]}]
                               [<cons $x $xs>
                                {[{} ts]
                                 @(map (lambda [$as $bs] [{x @as} bs])
                                       (loop-fn xs))}]}]))]}
                     (loop-fn tgt))]}]
           [nioj [(List a) (List a)]
            {[$tgt (let {[$loop-fn
                          (lambda [$ts]
                            (match ts (List a)
                              {[<nil> {[{} {}]}]
                               [<snoc $x $xs>
                                {[{} ts]
                                 @(map (lambda [$as $bs] [{@as x} bs])
                                       (loop-fn xs))}]}]))]}
                     (loop-fn tgt))]}]})]
       [$equal? (lambda [$val $tgt]
                  (match [val tgt] [(List a) (List a)]
                    {[[<nil> <nil>] <true>]
                     [[<cons $x $xs> <cons ,x ,xs>] <true>]
                     [[_ _] <false>]}))]}))))
```

図 4 リストの型定義

以上がバージョン 1.2.3 の時点での Egison 型定義の仕様である。次にバージョン 2.4.1（ワークショップ以降）の仕様を見ていく。図 5 はバージョン 2.4.1 の型定義である。



```
(define $Multiset
  (lambda [$a]
    (type
      {[,$val []
        {[$tgt (match [val tgt] [(List a) (Multiset a)]
                 {[[<nil> <nil>] {[]}]
                  [[<cons $x $xs> <cons ,x ,xs>] {[]}]
                  [[_ _] {}]})]}]
       [<nil> []
        {[{} {[]}]
         [_ {}]}]
       [<cons ,$px _> [(Multiset a)]
        {[$tgt (if ((member? a) px tgt)
                   {((remove a) tgt px)}
                   {})]}]
       [<cons _ _> [a (Multiset a)]
        {[$tgt (letrec {[$helper (lambda [$xs $ys]
                                   (match ys (List a)
                                     {[<nil> {}]
                                      [<cons $z $zs> (if ((member? a) z xs)
                                                         (helper {@xs z} zs)
                                                         {[z {@xs @zs}] @(helper {@xs z} zs)})]}))]}
                 (helper {} tgt))]}]
       [<join ,$pxs _> [(Multiset a)]
        {[$tgt (letrec {[$helper (lambda [$xs $ys]
                                   (match xs (List Something)
                                     {[<nil> ys]
                                      [<cons $z $zs> (if ((member? a) z ys)
                                                         (helper zs ((remove a) ys z))
                                                         {})]}))]}
                 {(helper pxs tgt)})]}]
       [<join _ _> [(Multiset a) (Multiset a)]
        {[$tgt
          (foldr
                 (lambda [$xi $xs]
                   (let {[[$x $i] xi]}
                     (map&concat
                                (lambda [$sub]
                                  (do {[[$ys $zs] sub]
                                       [$zs ((remove-all a) zs x)]}
                                    (match-all (loop $l $j (between 1 i) {x @l} {}) (List a)
                                      [<join $us $vs> [{@us @ys} {@zs @vs}]])))
                                xs)))
                 {[{} tgt]}
                 ((occurrence a) tgt))]}]
       [_ [Something]
        {[$tgt {tgt}]}]
       }))))
```

図 5 Egison バージョン 2.3.1 での型定義

図 1 の Multiset の型定義より長いが、それはこちらの型定義のほうがより詳細にパターンマッチの方法を定義しているからである。パターン内にバリューパターンが含まれる場合、こちらのほうがかなり早く動作する。

図 5 のコードの意味を説明する。

Multiset は型 a を受け取り、その型 a のマルチセットの型を返すように定義されている。これは前のバージョンと同じである。

型定義の要である destructor 式の仕様が変わったのがこの改良の肝である。Type 構文が過去のバージョンの destructor 式の役目を果たしている。

過去の仕様では destructor マッチ節の 1 つ目の要素は、パターンコンストラクタの名前であった。しかし、新しいバージョンではここにパターンのパターンを記述するようになっている。これにより、var-match と equal?をそれぞれ別々に定義する必要がなく

なり、構文がシンプルになっている。なぜなら var-match はパターンがパターン変数である場合の関数であったが、その役割は最後の destructor マッチ節が果たしている（パターン変数は単一の_にマッチする）。equal?はパターンがバリューパタンである場合に使われる関数であったが、それは最初の destructor マッチ節がその役割を果たしている。新しいバージョンでは equal?関数はライブラリに図 6 のように定義されていて、お互い片方をバリューパターンとしてマッチ可能なら、同じ値という動作になる。

```
(define $match?
  (lambda [$a]
    (lambda [$x $y]
      (match x a
        {[,y #t]
         [_ #f]}))))

(define $=
  (lambda [$a]
    (lambda [$x $y]
      (and ((match? a) x y) ((match? a) y x)))))
```

図 6 equal?関数の定義

次に重要な変更は、新たなパターンの追加である。not パターン、loop パターン、macro パターンが新たにプロジェクト期間中に追加されたパターンである。

- not パターン

not パターンは^pat という形をとる。ターゲットが pat にマッチしない場合にマッチするパターンである。図 7 はパターンマッチのターゲットのコレクションに 2 度同じ要素が現れない場合にマッチするパターンを書いたものである。

```
(match-all {1 2 1 4 2 1} (Multiset Integer)
  [<cons $x ^<cons ,x _>> x])
-> {4}
```

図 7 not パターンの例

整数のマルチセットとしてターゲットをマッチしているので、図 7 の例のパターンは変数 x に同じ要素が 2 度現れない要素を束縛する。ターゲットのコレクションの要素のうち、4 だけが 1 個しか含まれていない要素なので 4 だけがマッチする。

図 8 は not パターンを使って 4-queen の問題を解くプログラムを書いたものである。

```
(define $four-queen
  (match-all {1 2 3 4} (Multiset Integer)
    [<cons $a_1
      <cons (& ^,(- a_1 1) ^,(+ a_1 1)
               $a_2)
       <cons (& ^,(- a_1 2) ^,(+ a_1 2)
                ^,(- a_2 1) ^,(+ a_2 1)
                $a_3)
        <cons (& ^,(- a_1 3) ^,(+ a_1 3)
                 ^,(- a_2 2) ^,(+ a_2 2)
                 ^,(- a_3 1) ^,(+ a_3 1)
                 $a_4)
         <nil>>>>>>
     [a_1 a_2 a_3 a_4]]))

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

図 8 not パターンを使った 4-Queen

図 8 の最終行にみられる出力では、1 行目は 2 列目に、2 行目は 4 列目に、3 行目は 1 列目に、4 行目は 3 列目にクイーンを置く配置と、1 行目は 3 列目に、2 行目は 1 列目に、3 行目は 4 列目に、4 行目は 2 列目にクイーンを置く配置の 2 つの解を出力している。図 9 は 1 つ目の答えを視覚化したものである。

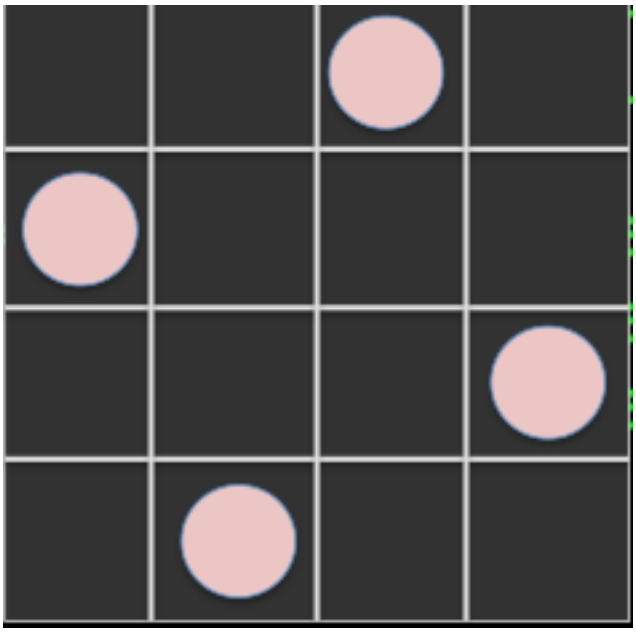


図 9 4-Queen の解の視覚化

- loop パターン

loop パターンは、パラメータの値によって繰り返しの数が異なるパターンを記述するための機構である。loop パターンを使えば図 8 のプログラムは図 10 のように簡略化できる。

```
(define $four-queen
  (match-all {1 2 3 4} (Multiset Integer)
    [<cons $a_1
           (loop $l $i {2 3 4}
                 <cons (loop $l1 $i1 (between 1 (- i 1))
                             (& ^,(- a_i1 (- i i1))
                                ^,(+ a_i1 (- i i1))
                                l1)
                             $a_i)
                       l>
                 <nil>)>
     [a_1 a_2 a_3 a_4]]))
```

図 10 loop パターンを使った 4-Queen

loop パターンは 1 引数目にループ変数と呼ばれる変数（図 10 では"$l"）、2 引数目に添字変数と呼ばれる変数（"$i"）、3 引数目に添字変数が動く範囲であるコレクション（"{2 3 4}"）、4 引数目に添字変数が添字集合を動いている間にループ変数が取る

値(″<cons″ から″l>″まで)、5 引数目に添字変数が添字集合の要素を全て取り終わったあとにループ変数が取る値(″<nil>″)をとる。

Egison には添字付き変数という概念がある。変数名に含まれる″_″以降は式として評価されて整数を返すものとなり、変数名とその値との組み合わせで 1 つの変数となる。添字付き変数の存在のおかげで、loop パターンを用いて任意の回数のパターン変数を含むパターンの繰り返しを表現できる。図 11 は loop パターンが、loop がなくなるまで展開されていく例である。

```
(loop $l $i {1 2} <join _ <cons $a_i l>> _)
->
<join _ <cons $a_1 (loop $l $i {2} <join _<cons $a_i l>> _)>>
->
<join _ <cons $a_1 <join _ <cons $a _2 (loop $l $i {} <join _<cons $a_i l>>
_)>>>>
->
<join _ <cons $a_1 <join _ <cons $a _2 _>>>>
```

図 11 loop パターンの例

loop パターンを使うと、先ほどの 4-queen は図 12 のように簡単に n-queen に一般化できる。loop パターンはこのようなことをするために考えられた機構である。

```
(define $n-queen
  (lambda [$n]
    (match-all (between 1 n) (Multiset Integer)
      [<cons $a_1
             (loop $l $i (between 2 n)
                   <cons (loop $l1 $i1 (between 1 (- i 1))
                               (& ^,(- a_i1 (- i i1))
                                  ^,(+ a_i1 (- i i1))
                                  l1)
                               $a_i)
                         l>
                   <nil>)>
       [@(loop $l $i (between 1 n)  {a_i @l} {})]])))
```

図 12 n-Queen

loop パターンの他の応用例としては例えばグラフのパターンマッチがある。例えばハミルトン閉路にマッチするパターンではグラフのノードの数に応じてパターンの繰り返しの回数が変わってくるので loop パターンが必須である。

- macro パターン

macro パターンはパターンをモジュール化し再利用するための機構である。Egison ではパターンの左側で行われた束縛をその右側をパターンマッチする際に参照することが可能であるので、スコープの関係で関数を用いてこのようなモジュール化ができない。そこでマクロを用意した。

図 13 は麻雀の役判定のために、macro パターンを用いて順子と刻子のパターンを表現したものである。

```
(define $shuntsu
  (macro [$s $pat]
    <cons $`s <cons ,(+ `s 1) <cons ,(+ `s 2) pat>>>))

(test (match-all {1 3 5 6 2 4} (Multiset Integer)
        [(shuntsu %m (shuntsu %n _)) [m n]]))

-> {[1 4] [4 1]}

(define $kohtsu
  (macro [$s $pat]
    <cons $`s <cons ,`s <cons ,`s pat>>>))

(test (match-all {1 3 5 5 2 5} (Multiset Integer)
        [(shuntsu %m (kohtsu %n _)) [m n]]))

-> {[1 5]}
```

図 13 macro パターンを用いた順子と刻子のパターン

図 13 の例では macro の引数に%が先頭についた変数を渡している。これは macro 変数と呼ばれる変数である。これは macro の中身で展開され、"`"(バッククォート)で打ち消される。つまり、1 つ目の例の shuntsu の中身の$`s は$m に展開される。麻雀の役判定には順子と刻子のパターンを複数書く必要があるが、この macro パターンを用いると大幅に簡略化できる。

- その他

プリミティブ関数の充実化も行った。IO プリミティブを充実させたことにより、Egison

でアルゴリズムだけでなくファイルの処理などのプログラムも書けるようになった。図 14 は Egison でファイルのコピーのプログラムを書いたものである。

```
(define $main
  (lambda [$: $argv]
    (match argv (List String)
      {[<cons $file1 <cons $file2 <nil>>>
        (do {[[$: $port1] (open-input-file : file1)]
             [[$: $port2] (open-output-file : file2)]}
          (letrec {[$copyLoop (lambda [$:]
                                (do {[[$: $line] (read-line-from-port : port1)]}
                                  (if (eof? line)
                                    :
                                    (do {[$: (write-string-to-port : port2 line)]
                                         [$: (write-char-to-port : port2 '\n')]}
                                      (copyLoop :))))))]}
            (do {[$: (copyLoop :)]
                 [$: (close-output-port : port2)]
                 [$: (close-input-port : port1)]}
              :))))]
       [_ <argv-error>]})))
```

図 14 ファイルのコピー

また Egison のバージョン 2.3 から多次元配列を扱えるようにした。これによって Egison で将棋盤やルービックキューブのパターンマッチなども行えるようになり、Egison で表現できる範囲が大きく広がった。

(2) コンパイラの作成

簡易的なものではあるがコンパイラを作成し、実行可能なバイナリファイルを出力できるようになった。図 15 は Egison で記述した Hello World!と、標準出力に出力するプログラムをコンパイルして実行した例である。egisonc というコマンドを用いてコンパイルは行われる。

```
/home/egi/egison2/sample/io% cat hello.egi
(define $main
  (lambda [$: $argv]
    (do {[$: (print : "Hello world!")]}
      :)))
/home/egi/egison2/sample/io% egisonc hello.egi
[1 of 1] Compiling Main             ( _tmp.hs, _tmp.o )
Linking hello ...
/home/egi/egison2/sample/io% ./hello
Hello world!
```

図 15 Egison コンパイラ

図 15 の hello.egi は、サンプルコードとしてプログラムと一緒に配布されている。

プロジェクト開始時の Egison はバージョン 0.4.0 だったが、現在の Egison のバージョンは 2.4.1 である。そのことからもわかるようにプロジェクト期間中に Egison は 2 度もメジャーアップデートした。

(3) ワークショップの開催

2012 年 7 月 7 日に秋葉原で第 1 回 Egison ワークショップを行った[1]。13:00-19:00 に渡るワークショプで 20 人弱もの参加者が集まった。ワークショップ開催前後で Twitter 上において大きく話題になった。図 16 はワークショップ開催前のツイートの一部を集めたものである。

図 16 ワークショップ開催前のツイート

図 17 はワークショップが終わったあとのツイートの一部を集めたものである。

[1] http://hagi.is.s.u-tokyo.ac.jp/~egi/egison/workshop/1.html

図 17 ワークショップ開催後のツイート

Egison の未来に期待する多くのツイートを見つけることができる。

ワークショップのプログラムは以下のようなものであった。

- 13:00～14:30 は Egion 講習の時間とした。開発者であるクリエータが Egison のデモプログラムを動かしながら参加者に Egison について解説を行った。
- 14:30～16:30 は各自が Egison の演習を行った。Egison 公式サイトにある Egisonist 検定を解いてもらった。
- 16:30～18:00 は本クリエータではない 4 人の Egison プログラマに講演を行っていただいた。

1 人目の話者は本多健太郎氏で、「Egison で QM 法」という題目で講演してくれた。QM 法というのは、論理式を圧縮するためのアルゴリズムであり、電子回路の最適化を行うために用いられる。Egison はマルチセットを直感的に扱えるため、この QM 法のアルゴリズムも短くシンプルに書ける。

2 人目の話者は中村宇佑氏で、「Egison で文法解析（PEG）」という題目で発表してくれた。Egison の強力なパターンマッチ機能をもちいて文法解析のパターンマッチをする話をしてくれた。Egison の拡張のアイデアについても話してくれた。

3 人目の話者は川又生吹氏で「Egison で麻雀」という題目で発表してくれた。麻雀の役判定はまさにマルチセットのパターンマッチであるので、Egison で書くと非常にシンプルに書くことができる。

4 人目の話者は平井洋一氏で「パターンのパターンマッチ」という題目で話してくれた。Egison でパターン型を定義し、パターンのパターンマッチの方法を定義すること

でパターンのパターンマッチをしたプログラムを動かしてくれた。

• 18:00～19:00 は Egison についてメタに考えるグループディスカッションを行った。

ワークショップを行った結果、当初の目的の通り、Egison を理解する人が増えた。ワークショップが終わった後、積極的に Egison を使ってくれてバグ報告をしてくれたり、Egison のパッチを書いてくれたり、ライブラリを編集してくれたりする方々が現れた。現在は彼らも GitHub の Egison のソースコードを自由に編集できるようにしている。以下は GitHub で彼らがたくさんの Issue を投げてくれているもののスナップショットをとったものである。ワークショップが終わってからこの 2 月で 41 もの Issue があがっている。

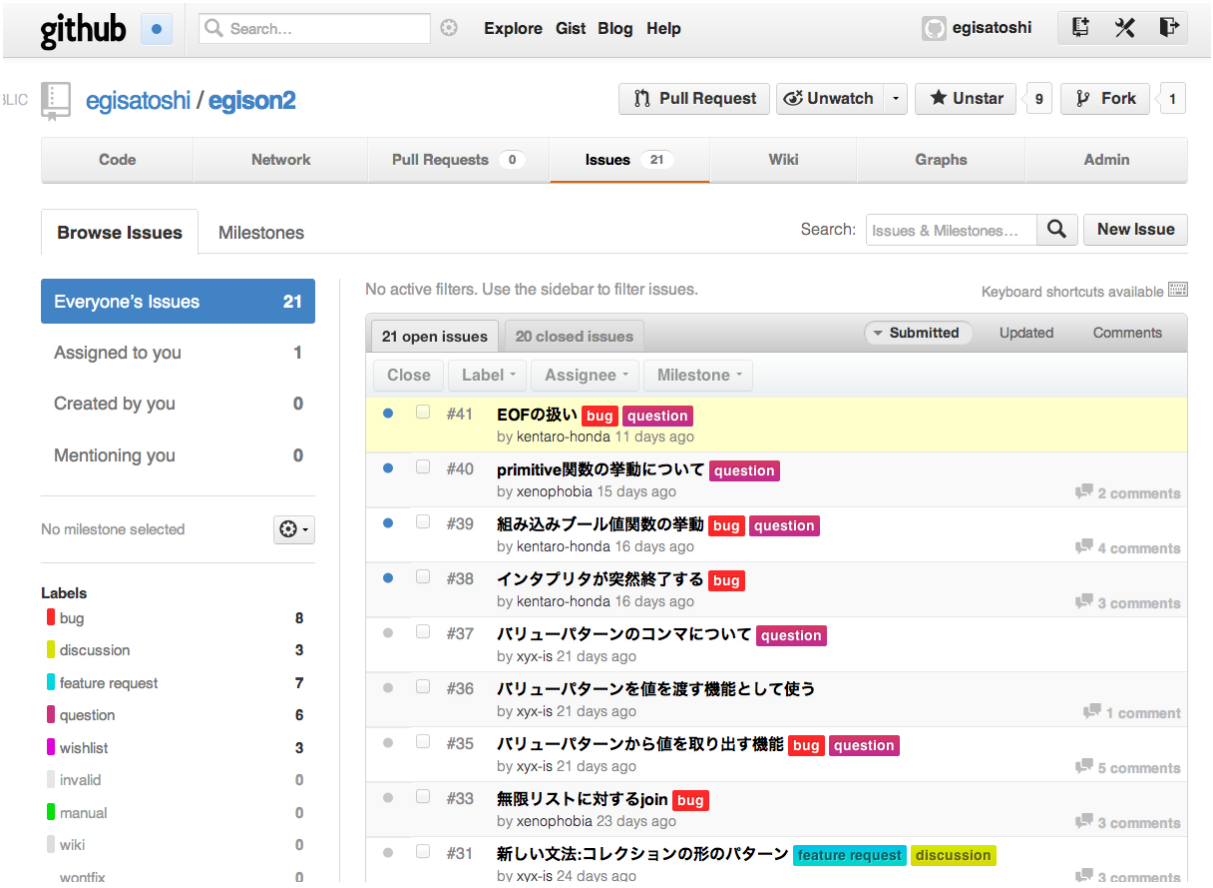


図 18 GitHub 上での Egison の盛上り

ワークショップの参加者の一人が熱烈な Egison ファン(Egisonist)になってくれたことが、ワークショップの一番大きな収穫であったかもしれない。彼はワークショップの開催以前から Egison に惚れ込み、熱心に Egison の勉強をして、Egison のバグ報告までしてくれた(図 19)。

図 19 Egisonist からのバグ報告

ワークショップが終わった後も Egison でプログラムを書くことを続けてくれた（図 20）。

図 20 Egisonist の反応

さらにはワークショップが終わった 2 週間後に Haskell のプログラム内でも Egison が使えるように Egison-Quote というパッケージを作って公開してくれた（図 21）。

図 21 Egison-Quote の登場

図 22 はその Egison-Quote を用いて Haskell プログラムのなかに Egison コードを埋め込んだ例である。

```haskell
{-# Language TemplateHaskell, QuasiQuotes #-}
module Main where

--import Language.Haskell.TH
import Language.Egison.Quote

combination3_2 :: [(Int, Int)]
combination3_2 = [egison| (match-all {1 2 3} (Multiset Integer)
                            [<cons $x <cons $y _>> [x y]])
                          :: [(Int, Int)] |]

main :: IO ()
main = do
  let ps = combination3_2
  putStrLn $ show ps
```

図 22 Egison-Quote の例

Haskellプログラム内で[egison| …|]と囲うとそのなかにEgisonコードを記述することができ、指定した Haskell の型で結果を抽出することができる。彼は現在 Egison 開発コミュニティに属しており、開発を継続してくれている。

10. プロジェクト評価

開発当初から、かなりの完成度で動作していたが、2 度のメジャーバージョンアップにより、言語上の機能もかなり追加され、プログラミング言語として完成度が高まった. このプロジェクトは単に新しいプログラミング言語が増えただけにはとどまらない、非常に大きな貢献がある。それは、パターンマッチを自ら記述できるという新しい分野を切り開いた点である。単体の言語の完成度が高まることで、言語に込められたメッセージ(パターンマッチを記述する)という部分が広まり、その結果、これを別の言語に実装される可能性もでてくる。その最初のステップとして、Haskell から Egison のパターンマッチ式を呼び出す機能を実装する人も現れた。核の部分のオリジナル度が非常に高いため、このような広がりは今後ますます増えるだろう。

11. 今後の課題

新しいプログラミング言語は普及するまでに時間が相当かかるので、辛抱強くこれに付き合ってゆくしかない。ワークショップは定期的に開催することが重要で、そこで

発表することを目標として応用が広がってゆく。