

プログラミング言語Egisonのループ構文による拡張

江木聡志(東京大学)

添字付き変数

```
$x_1
$y_2_1
$z_(+ 1 1)_3_(* 3 2)
```

Egisonでは変数名の後に'_'とそれに続けて式を書く
と添字付き変数となります。添字はいくつでも付ける
ことができます。添字の式は評価されると整数を返
す式でないといけません。

例1. 組み合わせの計算

コレクションをリストとしてパターンマッチし、consと
joinを組み合わせたパターンでマッチすると、そのコ
レクションからいくつかの要素を取り出す組み合わ
せをとることができます。

以下は2つの要素を取り出す組み合わせを返す例
です。

```
(test (match-all {1 2 3 4 5} (List Integer)
  [<join _ <cons $a_1 <join _ <cons $a_2 _>>>
   [a_1 a_2]]))
=> {[1 2] [1 3] [1 4] [1 5] [2 3]
    [2 4] [2 5] [3 4] [3 5] [4 5]}
```

また、以下のようにすると3つの要素を取り出す組み
合わせを返す式になります。

```
(test (match-all {1 2 3 4 5} (List Integer)
  [<join _ <cons $a_1 <join _ <cons $a_2 <join _ <cons $a_3 _>>>>
   [a_1 a_2 a_3]]))
=> {[1 2 3] [1 2 4] [1 2 5] [1 3 4] [1 3 5]
    [1 4 5] [2 3 4] [2 3 5] [2 4 5] [3 4 5]}
```

取り出す要素の数が固定なら、上記の例の通り1つ
の簡潔なパターンで表現できます。

しかし、パラメータの値によって取り出す要素の数を
変えるようなパターンを書こうとすると特別な工夫な
しではそれが不可能なことに気付きます。

```
<join _ <cons $a_1
<join _ <cons $a_2
...
<join _ <cons $a_i _>
...
<join _ <cons $a_n>>...>
```

ループ構文を用いるとこのような添字だけが変わる
式が任意の回数続く式を表現することができます。

以下は上の式をループ構文を使って表現したもので
す。

```
(loop $l $i (between 1 n) <join _ <cons $a_i l>> _)
```

このパターンを使うと引数で指定されたコレクショ
ンから、もう1つの引数で指定された数の要素を抽
出した組み合わせを返す関数を簡潔に定義できま
す。

```
(define $combination
  (lambda [$xs $k]
    (match-all xs (List Something)
      [(loop $l $i (between 1 k) <join _ <cons $a_i l>> _)
       [@(loop $l $i (between 1 k) {a_i @l} {})]
      ])))

(test (combination {1 2 3 4 5} 3))

=> {[1 2 3] [1 2 4] [1 2 5] [1 3 4] [1 3 5]
    [1 4 5] [2 3 4] [2 3 5] [2 4 5] [3 4 5]}

(test (combination {1 2 3 4 5 6 7 8} 2))

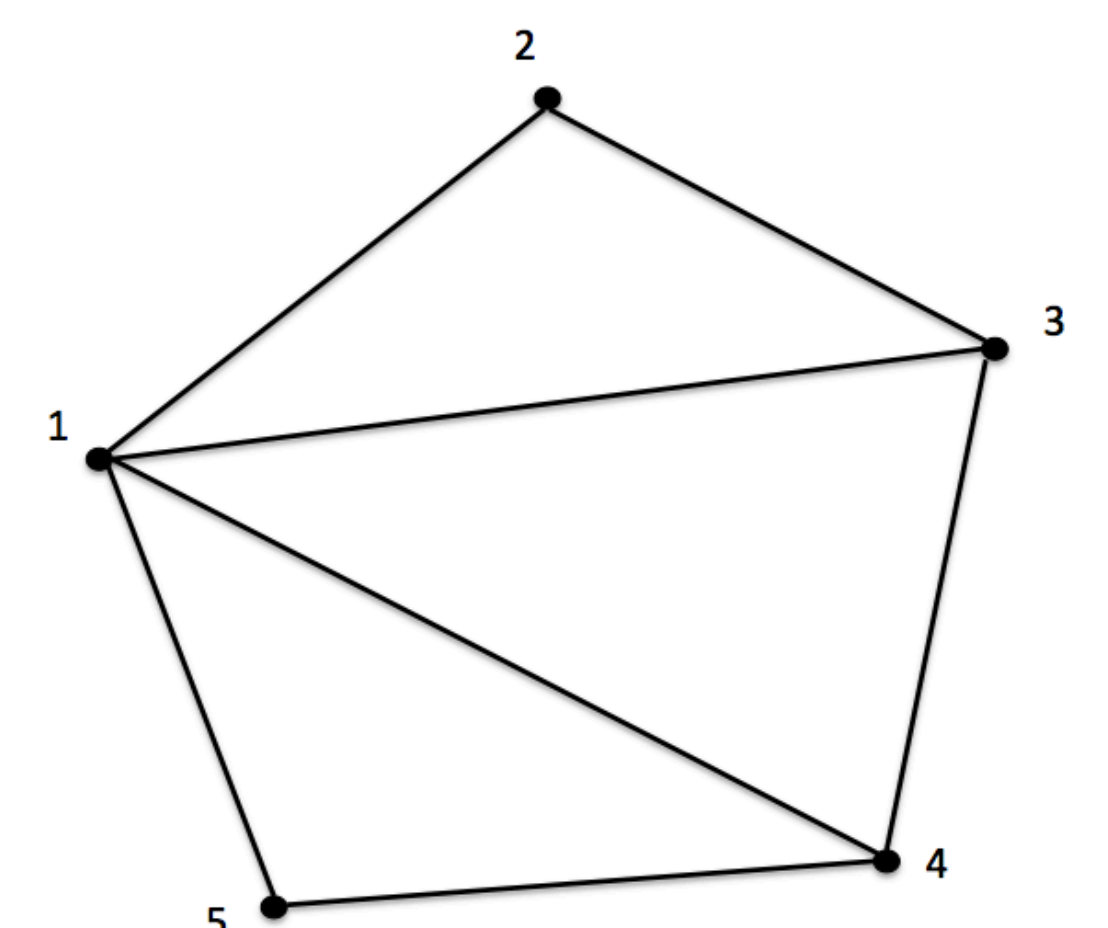
=> {[1 2] [1 3] [1 4] [1 5] [1 6] [1 7] [1 8]
    [2 3] [2 4] [2 5] [2 6] [2 7] [2 8] [3 4]
    [3 5] [3 6] [3 7] [3 8] [4 5] [4 6] [4 7]
    [4 8] [5 6] [5 7] [5 8] [6 7] [6 8] [7 8]}
```

例2. グラフのパターンマッチ

ループ構文を使うと、グラフのノードの個数によって
パターンの長さが変わるハミルトン閉路やハミルトン
パスのパターンが1つのパターンで簡潔に表現でき
ます。

```
(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 5} {1 3 5}>
  <node 5 {1 5} {1 5}>})
```

グラフはノードとそこから伸びる
エッジの組み合わせのマルチ
セットとして表現されます。



```
(define $hamilton-cycle
  (lambda [$g]
    (let [{n (size g)}]
      (match-all g Graph
        [<cons <node $a_1 <cons $a_2 _> _>
         (loop $l $i (between 3 n)
           <cons <node ,a_(- i 1) <cons $a_i _> _>
             l>
           <cons <node ,a_n <cons ,a_1 _> _>
             _>)]
         [@(loop $l $i (between 1 n) {a_i @l} {})]
        ])))

(test (hamilton-cycle g))

=> {[1 2 3 4 5] [2 3 4 5 1] [3 4 5 1 2] [4 5 1 2 3] [5 1 2 3 4]}
```

```
(define $hamilton-path
  (lambda [$g]
    (let [{n (size g)}]
      (match-all g Graph
        [<cons <node $a_1 <cons $a_2 _> _>
         (loop $l $i (between 3 n)
           <cons <node ,a_(- i 1) <cons $a_i _> _>
             l>
           <cons <node ,a_n _> _>
             <nil>>)]
         [@(loop $l $i (between 1 n) {a_i @l} {})]
        ])))

(test (hamilton-path g))

=> {[1 2 3 4 5] [2 1 3 4 5] [2 3 1 4 5] [2 3 4 1 5] [2 3 4 5 1] [3 2 1 4 5]
    [3 4 5 1 2] [4 3 2 1 5] [4 5 1 2 3] [4 5 1 3 2] [5 1 2 3 4] [5 1 4 3 2]}
```