プログラミング言語 Egison 入門

Egison 開発チーム

2020 年 5 月 1 日

はじめに

Egison は，より直感的にアルゴリズムを表現したいという動機で発案されたプログラミング言語のアイデアを実証するためにつくられた proof of concept のプログラミング言語である．Egison に実装されているこれらのアイデアのうち，もっとも重要な機能は本書の主題であるパターンマッチである．Egison のパターンマッチの特徴は，ユーザーにより拡張可能でかつ，非線形パターン（パターン変数に束縛した値を同じパターン内で参照するパターン）をバックトラッキングにより効率的に処理することである．本書は，Egison を通して提唱されているパターンマッチ指向プログラミングというプログラミング・パラダイムをできるだけコンパクトに紹介することを目的として執筆された．パターンマッチまわりの Egison の構文と，それらの構文を使ったプログラミングテクニックを紹介する．

本書がきっかけとなって，プログラミング言語，とくにパターンマッチの研究をする研究仲間が読者の中から現れてくれたらとてもうれしい．

2020 年 2 月 13 日
江木 聡志

目次

第 1 章

プログラミング Egison とは

本章では，Egison がどのような言語であるのか紹介する．Egison とそれが提唱しているパターンマッチ指向プログラミングについて，その大体のイメージをつかんでもらいたい．

1　プログラミング言語全体の中での Egison の位置づけ

プログラミング言語の機能には大きく分けて 2 種類ある．コンピュータを含むさまざまな機械の操作を簡潔に記述するための機能と，人間の頭のなかにある抽象的な概念をプログラムとして表現するための機能である．コンピュータは物理的なデバイスであるため，前者の機能は必須である．この機能には，たとえばキャッシュやメモリ管理ための操作を自動化する機能などが含まれる．後者の機能は，プログラマの頭のなかにあるアルゴリズムの認識を，コンピュータ向けに翻訳することなく記述できるようにすることを目指す．コンピュータに解かせたい問題のなかには，その解法に抽象的な概念が関係する問題が多くある．人間には簡単に理解できる抽象化でも，コンピュータが理解できる形で表現することは難しいことが多い．Egison は後者の機能の拡充に注力してつくられている．

アルゴリズムを簡潔に記述することを目指す主流の流派は，関数型プログラミング言語である．関数型プログラミング言語によるプログラムの記述の大きな特徴は，関数を他の組み込みデータ型と同じようにプログラマが扱えることである．具体的にいうと，関数を別の関数の引数として渡したり，関数を返す関数を定義することができる．そのおかげで，関数型プログラミング言語以外の言語ではモジュール化がむずかしい処理をモジュール化できることが多々ある．関数型プログラミングでは，物理的なデバイスであるコンピュータの操作は，多くの場合，コンパイラによってプログラマから隠される．

しかし，関数型プログラミング言語でも，頭のなかのアルゴリズムのイメージを，プログラムとして記述できるように翻訳する必要がある場合がある．非自由データ型を扱うアルゴリズムは，その典型的な例である．非自由データ型とは，同じデータにたいして複数の同値な表現形があるデータ型のことをいう．たとえば，多重集合（要素の重複を許す集合）は非自由データ型である．$\{a, a, b\}$ という多重集合は，$\{a, b, a\}$ や $\{b, a, a\}$ とも表現できるからである．ほかには，グラフや

数式なども非自由データ型である．

非自由データ型にたいするパターンマッチを可能にすることによって，簡潔に記述できるアルゴリズムの範囲が広げることを目指して Egison は開発された．Egison には非自由データ型のデータをパターンマッチするための機能が実装されている．そして，このパターンマッチを活かしたプログラミング・パラダイムであるパターンマッチ指向プログラミングを提唱している．

2　パターンマッチ指向プログラミングとは

本節では，いくつかの例をみることによって，パターンマッチ指向プログラミングとはどのようなプログラミング・パラダイムであるのかそのイメージを紹介する．

パターンマッチ指向プログラミングによってプログラムの記述が直感的になっていることのわかりやすい例に intersect関数の実装がある．intersectは 2 つのリストを引数にとり，それらのリストに共通する要素のリストを返す関数である．Egison を使って引数のリストをそれぞれ要素の順序を無視する集合としてパターンマッチすると，2 つのリストの共通要素にマッチするパターンを記述することにより，intersectを記述できる．プログラムの読み方は次章以降で紹介する．

```
intersect xs ys :=
  matchAllDFS (xs, ys) as (set eq, set eq) with
  | ($x :: _, #x :: _) -> x
```

対して，Egison のパターンマッチを使わずに関数型プログラミングのスタイルで Haskell で記述すると，リストに対する関数を組み合わせて共通要素を抜き出すための方法を記述する必要がある．

```
intersect xs ys =  filter (\x -> any (== x) ys) xs
```

このプログラムは，リスト内包表記を使って以下のように書き直すこともできる．

```
intersect xs ys =  [x | x <- xs, any (== x) ys]
```

Egison のパターンマッチを使ったプログラムは，2 つのリストの共通要素にマッチするパターンを記述しているだけであるのに対し，関数型プログラミングでは，どうやってその共通要素を取り出すかその方法をプログラマが考えて記述する．前者のように「何を計算したいか（what to do）」を記述するプログラミングスタイルは宣言型プログラミング，後者のように「どうのように計算するか（how to do）」を記述するプログラミングスタイルは手続き型プログラミングと呼ばれている．「何を計算したいか」から「どのように計算するか」が明らかである場合は，「何を計算したいか」を記述する宣言型プログラミングのほうがプログラムが読みやすく簡潔になる．intersectの例の場合は，Egison が集合のパターンマッチアルゴリズムをモジュール化できるおかげで，宣言的なプログラミングが可能になっている．

少し毛色の違う例として，concat関数をパターンマッチ指向プログラミングで定義する．リストのリストの要素にマッチするパターンを記述することにより，concatを定義できる．

```
concat xss :=
  matchAllDFS xss as list (list something) with
  | _ ++ (_ ++ $x :: _) :: _ -> x

concat [[1,2],[3],[4,5]]
-- [1,2,3,4,5]
```

さらにもう 1 つのパターンマッチ指向プログラミングの例として，unique関数を定義する．後方に自身と同じ値が含まれない要素にマッチするパターンを記述することにより uniqueを定義できる．

```
unique xs :=
  matchAllDFS xs as list eq with
  | _ ++ $x :: !(_ ++ #x :: _) -> x

unique [1,2,3,2,4]
-- [1,3,2,4]
```

次章以降で，上記のプログラムで使われている Egison の構文や，機能，プログラミングテクニックの解説をしていく．

3　本書の構成

本書は可能な限りコンパクトに，パターンマッチ指向プログラミングとそのための Egison の機能の解説をまとめることを目指した．そのため，本書は前提知識として，関数型プログラミング（Lisp 系言語や，OCaml，Haskell を使ったプログラミング）の知識を仮定している．既存の関数型言語にも存在する Egison の機能については，独立した章や節を設けず，登場したときに軽く解説するにとどめた．

本書の構成は以下の通りである．第 2 章では，パターンマッチに関係する Egison の構文を一通り紹介する．第 3 章では，パターンマッチ指向プログラミングで頻出するテクニックを紹介する．第 4 章では，Egison のパターンマッチを活かしたプログラミングスタイルであるパターンマッチ指向プログラミングとはなにか解説する．第 5 章では，Egison 内部のパターンマッチの仕組みも解説する．第 6 章では，ユーザーによるマッチャー定義の方法を説明する．第 7 章では，Egison のパターンマッチの実装を紹介する．

ユーザーとして Egison に興味がある読者は，第 2 章，第 3 章，第 4 章，第 5 章 1 節，第 6 章を順に読んでいくのがよい．Egison の設計・実装に興味がある読者は，第 2 章の前半部分を読んだ後，第 5 章，第 6 章，第 7 章を読むことができる．

第 2 章

Egison 早巡り

本章は，パターンマッチ指向プログラミングのための Egison の機能を一通り紹介する．本章を理解すれば，Egison のパターンマッチの仕様はほぼ一通り理解したことになるはずである．

本章の前半（1 節，2 節，3 節，4 節，5 節）では，パターンマッチのための構文を紹介する．Egison のパターンマッチは，非自由データ型に対するパターンマッチを実現するために，複数のマッチ結果をもつ非線形パターン（パターン変数に束縛した値を同じパターン内で参照するパターン）を効率的に処理できるように設計されている．本章の前半では，この機能がどのように構文として表現できるのか解説する．

本章の後半（6 節，7 節，8 節，9 節，10 節）では，さまざまな組み込みパターンを紹介する．そのうち後半で紹介されるループ・パターンや，シーケンシャル・パターンは Egison 以外のプログラミング言語にはまだ実装されていないパターンである．これらの組み込みパターンは，最初は特殊でアドホックにみえるかもしれない．これらの組み込みパターンの意味は，初見で完全に理解できなくても問題ない．本書の後半で，これらのパターンの実用例が紹介されたときに，本章にもどってきてほしい．

1　matchAll 式によるパターンマッチ

パターンマッチを記述するためのいくつかの構文を Egison は提供している．そのうちもっとも基本的な構文は `matchAll`式である．

```
matchAll [1,2,3] as list something with
| $x :: $ts -> (x, ts)
-- [(1,[2,3])]
```

`matchAll`式は，**ターゲット**（上記の場合，`[1,2,3]`），**マッチャー**（上記の場合，`list something`），1 つ以上の**マッチ節**（上記の場合，`$x :: $xs -> (x,xs)`）から構成される．マッチ節は，**パターン**（上記の場合，`$x :: $xs`）と**ボディ**（上記の場合，`(x,xs)`）からなる．`matchAll`式は，既存のプログラミング言語のマッチ式と同様に，ターゲットとパターンのパターンマッチを試みて，もしマッ

チしたらそのマッチ節のボディを評価する．

Egison の matchAll式の特徴は，(1) 結果としてリストを返すことと，(2) マッチャーという追加の引数をとることにある．(1) の特徴は，複数の結果をもつパターンマッチをサポートするための特徴である．matchAll式は複数のパターンマッチの結果すべてについてボディを評価して，その結果を集めてリストとして返す．上記の例のパターンに使われている::は**コンス・パターン**（*cons pattern*）と呼ばれる，リストを先頭の要素と残りのリストに分解するパターンコンストラクタである．そのため，上記の例の場合はパターンマッチの結果が一つであるので，長さが 1 のリストを返している．(2) の特徴は，パターンマッチアルゴリズムの拡張性とパターンの多相性を実現するための特徴である．マッチャーは Egison 以外の言語ではみられない独自のオブジェクトで，パターンマッチアルゴリズムを保持するためのオブジェクトである．マッチャーによるパターンの多相性については，4 節で扱う．--で始まる行はコメントである．本書において，プログラム直後のコメントはプログラムの実行結果を記述するものとする．

matchAll式の文法の詳細についてもう少し説明する．asと withというキーワードでマッチャーは囲まれる．matchAll式はマッチ節を複数とることができる．[*1]マッチ節の先頭には|がつく．[*2]|はマッチ節が複数行に渡る場合にプログラムの可読性を高める．マッチ節中のパターンとボディは->で区切られる．

```
matchAll ターゲット as マッチャー with
| パターン 1 -> ボディ 1
| パターン 2 -> ボディ 2
...
```

マッチ節が 1 つであるときは，下記のようにマッチ節の前の|を省略することができる．

```
matchAll ターゲット as マッチャー with パターン -> ボディ
```

複数の結果をもつパターンマッチの例を紹介する．下記の matchAll式のパターンで使われている ++は**ジョイン・パターン**（*join pattern*）と呼ばれる，リストを先頭部分のリストと残りのリストに分解するパターンコンストラクタである．ジョイン・パターンによる複数の分解のすべてについて，matchAll式はボディ式を評価する．

```
matchAll [1,2,3] as list something with
| $hs ++ $ts -> (hs, ts)
-- [([], [1, 2, 3]), ([1], [2, 3]), ([1, 2], [3]), ([1, 2, 3], [])]
```

[*1] matchAll t as m with c_1 c_2 ... は，(matchAll t as m with c_1) ++ (matchAll t as m with c_2) ++ ... と同値である．

[*2] マッチ節が複数ある場合，最初のマッチ節の前にも | は必須である．

2　値パターンと述語パターンによる非線形パターンの表現

matchAll式は，非線形パターンと組み合わせたときにその本領を発揮する．たとえば，以下の非線形パターンは，対象のコレクションが同値な要素のペアを含む場合にパターンマッチに成功する．

```
matchAll [1,2,3,2,4,3] as list integer with
| _ ++ $x :: _ ++ #x :: _ -> x
-- [2,3]
```

値パターン (*value pattern*) は非線形パターンを表現するために重要な役割を果たす．値パターンは，値パターンの中身とターゲットが等しいかどうかチェックする．値パターンの先頭には，#が付加される．#の後ろには任意の式を書くことができる．#に続く式は，その値パターンの左側に現れたパターン変数に束縛された値を参照しながら評価される．この動作を実現するため，Egisonのパターンは左から右に順番に処理されることが決まっている．その結果，`$x :: #x :: _`のようなパターンは妥当であるが，`#x :: $x :: _`は正しく動かない．（どうしてもパターンの右側に現れるパターン変数の値を参照したいという場合は，8 節で紹介するシーケンシャル・パターンが使える．）

ほかの非線形パターンの例として，双子素数のパターンマッチを紹介する．双子素数とは，差が 2 であるような素数のペアのことをいう．primesには素数の無限列が束縛されている．[*3]このmatchAll式は，素数の無限列からすべての双子素数を順番に抜き出す．

```
twinPrimes := matchAll primes as list integer with
  | _ ++ $p :: #(p + 2) :: _ -> (p, p + 2)

take 8 twinPrimes
-- [(3, 5), (5, 7), (11, 13), (17, 19), (29, 31), (41, 43), (59, 61), (71, 73)]
```

パターンマッチのときに，同値性よりも一般的な条件を使いたいことがある．**述語パターン** (*predicate pattern*) はそのために用意されている組み込みパターンである．述語パターンは，中身の述語にターゲットを適用した結果がTrueである場合，パターンマッチに成功するパターンである．述語パターンの先頭は，?からはじまり，?のあとには 1 引数の述語が続く．

```
twinPrimes := matchAll primes as list integer with
  | _ ++ $p :: ?(\q -> q = p + 2) :: _ -> (p, p + 2)
```

[*3] primesは Egison の組み込みライブラリで定義されている．

3　バックトラッキングによる効率的な非線形パターンマッチ

Egison 内部のパターンマッチアルゴリズムは，非線形パターンを効率的に処理するためにバックトラッキングを使う．

```
matchAll [1..n] as list integer with _ ++ $x :: _ ++ #x :: _ -> x
-- O(n^2) で [] を返す
matchAll [1..n] as list integer with _ ++ $x :: _ ++ #x :: _ ++ #x :: _ -> x
-- O(n^2) で [] を返す
```

上記の 2 つのマッチ式は，1 から n までの整数のリストから同じ要素の 2 つ組，3 つ組をそれぞれ抽出する．同じ要素の 2 つ組も 3 つ組もターゲットのリストには含まれていないため，これらの matchAll式は両方とも空リストを返す．2 つ目の matchAll式が評価されるとき，Egison 処理系は 2 つ目の#xのパターンマッチまで行き着かない．1 つ目の#xでパターンマッチが失敗するからである．（2 節で述べたように Egison のパターンマッチはパターンを左から右へ順番に処理する．）それゆえ，両方の matchAll式の時間計算量は同じである．Egison 内部のパターンマッチアルゴリズムについては，第 5 章で詳しく解説する．

4　マッチャーによるパターンの拡張性と多相性

パターンの拡張性以外のマッチャーがもたらすメリットに，**パターンのアドホック多相性**がある．パターンのアドホック多相性とは，::や ++などといったパターン・コンストラクタを，リストや多重集合などといった複数のマッチャーに対して，同じ名前で使うことができる性質のことである．パターンのアドホック多相性は，さまざまな非自由データ型のパターンマッチを許す Egison において非常に重要である．なぜなら，1 つのデータがプログラムの複数の箇所でそれぞれ違う非自由データ型としてパターンマッチされることが多々あるためである．たとえば，リストは多重集合，集合としてパターンマッチされることがある．パターンの多相性によって，パターンコンストラクタの名前の数を大幅に減らすことができる．

下記の matchAll式は，**コレクション**（*collection*）[1,2,3]をターゲットとして，それぞれ違うマッチャーを使って同じコンス・パターンでパターンマッチしている．ここでコレクションという言葉を使ったが，これは今までリストと呼んでいたもののことを指している．コレクションは，リストとしてパターンマッチされることもあれば，多重集合や集合としてパターンマッチされることがある．そのため，リストや多重集合，集合などとして扱われることがあるデータを総称して以降コレクションと呼ぶ．マッチャーがリストの場合，コンス・パターンは，先頭の要素と残りの要素のコレクションに分解する．マッチャーが多重集合の場合，コンス・パターンは，ある要素と残りの要素のコレクションに分解する．マッチャーが集合の場合，コンス・パターンは，ある要素とターゲットのコレクション自身に分解する．集合のこの挙動は，集合をすべての要素を無限個含むコレクションとしてとらえれば，自然な仕様と考えることができる．（$\infty - 1 = \infty$ と考える．）

```
matchAll [1,2,3] as list something with $x :: $xs -> (x,xs)
-- [(1,[2,3])]
matchAll [1,2,3] as multiset something with $x :: $xs -> (x,xs)
-- [(1,[2,3]),(2,[1,3]),(3,[1,2])]
matchAll [1,2,3] as set something with $x :: $xs -> (x,xs)
-- [(1,[1,2,3]),(2,[1,2,3]),(3,[1,2,3])]
```

パターンの多相性は特に値パターンを表現するときに便利である．コンス・パターンなどのコンストラクタパターンと同様に値パターンの挙動もマッチャーによって異なる．たとえば，`[1,2,3] == [2,1,3]`のようなコレクション同士の同値性は，コレクションをリストとしてみなすか多重集合としてみなすかによって変わってくる．しかし Egison では，パターンのアドホック多相性のおかげで，両者の同値性を同じパターンでチェックできる．値パターンの多相性によって，非自由データ型を扱うプログラムが大幅に読みやすくなる．

```
matchAll [1,2,3] as list integer with #[2,1,3] -> "Matched" -- []
matchAll [1,2,3] as multiset integer with #[2,1,3] -> "Matched" -- ["Matched"]
```

5　matchAllDFS 式によるパターンマッチ結果の順序の制御

matchAll式は可算無限個の結果すべてを列挙するように内部のパターンマッチアルゴリズムが設計されているが，その列挙の仕方は処理系によって決められている．場合によっては，この順序が重要であることがある．

典型的な例を紹介する．以下の matchAll式はすべての自然数のペアを列挙する．take関数を使って先頭 8 個のペアを取り出している．matchAllはパターンマッチの探索木をすべてのノードをたどるために幅優先探索する．[*4] その結果，パターンマッチの結果の順序は以下のようになる．

```
take 8 (matchAll [1..] as set something with
        | $x :: $y :: _ -> (x,y))
-- [(1,1),(1,2),(2,1),(1,3),(2,2),(3,1),(2,3),(3,2)]
```

上記の順序は，無限に大きい可能性がある探索木をすべてたどる場合に適している．しかし，この順序が好ましくない場面もある．パターンマッチの探索木を深さ優先探索する matchAllDFSは，このような場面のために用意されている．

```
take 8 (matchAllDFS [1..] as set something with
        | $x :: $y :: _ -> (x,y))
-- [(1,1),(1,2),(1,3),(1,4),(1,5),(1,6),(1,7),(1,8)]
```

[*4] この幅優先探索の詳細については，Egison パターンマッチの設計についての論文 [9] の 5.2 節にて詳しく解説されている．

たとえば，以下のようにパターンマッチにより concat関数を定義する場合，matchAllDFS式を使うことで正しい順番で結果のコレクションを生成できる．

```
concat xss := matchAllDFS xss as list (list something) with
  | _ ++ (_ ++ $x :: _) :: _ -> x
```

もし，matchAllDFSの代わりに matchAllを使ってしまうと，以下のように，引数のリストのリストの要素を交互に列挙してしまう．

```
take 10 (matchAll [[1..], map neg [1..]] as list (list something) with
         | _ ++ (_ ++ $x :: _) :: _ -> x)
--  [1,2,-1,3,-2,4,-3,5,-4,6]
```

6　and パターン・or パターン・not パターン

and パターンや *or* パターン，*not* パターンといった論理パターンは，パターンの表現力を広げるために重要な役目を果たす．and パターンは，2 つのパターンをとり，両方のパターンがマッチする場合，その and パターン自体がパターンマッチに成功する．or パターンは，2 つのパターンをとり，どちらか 1 つのパターンがマッチする場合，その or パターン自体がパターンマッチに成功する．not パターンは，1 つのパターンをとり，そのパターンのマッチに失敗した場合，not パターン自体のパターンマッチに成功する．

and パターンと or パターンの使用例として，三つ子素数を抽出するパターンマッチを紹介する．三つ子素数とは，$(p, p+2, p+6)$ または $(p, p+4, p+6)$ の形で表される素数の三つ組のことをいう．or パターン (#(p + 2) | #(p + 4))は，$p+2$ と $p+4$ の両方にマッチするために使われている．and パターン ((#(p + 2) | #(p + 4)) & $m)は，$p+2$ または $p+4$ にマッチした場合，その値を$mに束縛するために使われている．この and パターンの使い方は，Haskell に提供されている as パターンの使い方に似ている．

```
primeTriples := matchAll primes as list integer with
  | _ ++ $p :: ((#(p + 2) | #(p + 4)) & $m) :: #(p + 6) :: _
  -> (p, m, p + 6)

take 6 primeTriples -- [(5,7,11),(7,11,13),(11,13,17),(13,17,19),(17,19,23),(37,41,43)]
```

not パターンは，その名前が示すとおり，ターゲットがパターンとマッチしない場合にパターンマッチに成功する．not パターンの先頭には!が付加され，!の後に任意のパターンが続く．以下の matchAllは双子素数でない隣り合う素数のペアを列挙する．not パターン!#(p + 2)は，$p+2$ 以外の値にマッチするパターンを表現している．

```
take 10 (matchAll primes as list integer with
         | _ ++ $p :: (!#(p + 2) & $q) :: _ -> (p, q))
-- [(2,3),(7,11),(13,17),(19,23),(23,29),(31,37),(37,41),(43,47),(47,53),(53,59)]
```

7　ループ・パターン

ループ・パターン（*loop pattern*）はパターンの複数回の繰り返しを表現するための組み込みパターンである．ループ・パターンは正規表現のクリーネスター演算子（*）の拡張である．

ターゲットのコレクションの 2 つの要素の組み合わせを列挙するパターンマッチを考えることからはじめる．これは以下のような matchAll式で記述できる．

```
comb2 xs := matchAll xs as list something with
  | _ ++ $x_1 :: _ ++ $x_2 :: _ -> [x_1, x_2]

comb2 [1,2,3,4] -- [[1,2],[1,3],[2,3],[1,4],[2,4],[3,4]]
```

Egison はこの例の中の$x_1と$x_2のように，パターン変数に添字を付加することを許す．[*5]これらは**添字付き変数**と呼ばれ，x_1 や x_2 のような数式に対応する．_のあとに続く式は，添字と呼ばれ，整数に評価される必要がある．x_i_j_kのように任意個の添字を変数に付加することができる．添字付き変数$x_iに値が束縛されたとき，もし変数 xにまだ何も束縛されていなかった場合に，Egison 処理系は，連想配列を生成し，xに束縛する．この連想配列のキーは整数 iであり，それに対応する値は添字付き変数$x_iにマッチした値である．変数 xにすでに連想配列が束縛されている場合には，新しいキーとバリューのペアがこの連想配列に追加される．

上記の comb2の 2 を n に一般化してみよう．ここで，ループ・パターンを使うことができる．

```
comb n xs := matchAll xs as list something with
             | loop $i                 -- 添字変数
                    (1, n)             -- 添字範囲
                    (_ ++ $x_i :: ...) -- 繰り返しパターン
                    _                  -- 終端パターン
             -> map (\i -> x_i) [1..n]

comb 2 [1,2,3,4] -- [[1,2],[1,3],[2,3],[1,4],[2,4],[3,4]]
comb 3 [1,2,3,4] -- [[1,2,3],[1,2,4],[1,3,4],[2,3,4]]
```

ループ・パターンは，**添字変数**，**添字範囲**，**繰り返しパターン**，**終端パターン**を引数にとる．添字変数（上記の場合，$i）は，現在の繰り返し回数を保持する変数である．添字範囲（上記の場合，(1, n)）は，添字変数が動く範囲を指定するために使われる．添字範囲は，**開始値**と**終了値**のペアである．繰り返しパターン（上記の場合，(_ ++ $x_i :: ...)）は，添字変数が添字範囲のなかにいる間，繰り返されるパターンである．終端パターン（上記の場合，_）は，添字変数が添字範囲の外に動いたときに展開されるパターンである．繰り返しパターンの中では，**三点リーダーパターン** ... を使うことができる．繰り返しパターンや終端パターンは，三点リーダーパターンの場所に展

[*5] このため，Egison では snake_case によって命名された変数を使うことができない．

開される．三点リーダーパターンで繰り返しパターン展開されるとき，添字変数の値がインクリメントされる．たとえば，$n = 3$ のとき，上記のループ・パターンは，以下のように展開される．

```
(loop $i (1, 3) (_ ++ $x_i :: ...) _)
_ ++ $x_1 :: (loop $i (2, 3) (_ ++ $x_i :: ...) _)
_ ++ $x_1 :: _ ++ $x_2 :: (loop $i (3, 3) (_ ++ $x_i :: ...) _)
_ ++ $x_1 :: _ ++ $x_2 :: _ ++ $x_3 :: (loop $i (4, 3) (_ ++ $x_i :: ...) _)
_ ++ $x_1 :: _ ++ $x_2 :: _ ++ $x_3 :: _
```

上記のループ・パターンの繰り返し回数は定数だった．しかし，添字範囲の終了値を整数値でなくパターンにすることにより，ターゲットによりループ・パターンの繰り返し回数が変わるようにすることができる．添字範囲の終了値がパターンである場合，三点リーダーパターンは繰り返しパターンと終端パターンの両方に展開される．三点リーダーパターンが終端パターンに展開されたときの繰り返し回数が，添字範囲の終了値のパターンとパターンマッチされる．以下のループ・パターンは，ターゲットのコレクションの先頭部分を列挙する．

```
matchAll [1,2,3,4] as list something with
| loop $i (1, $n) ($x_i :: ...) _ -> map (\i -> x_i) [1..n]
-- [[],[1],[1,2],[1,2,3],[1,2,3,4]]
```

上記，2 通りの添字範囲の指定を紹介したが，これらは下記のように，より一般的な添字範囲の指定方法に展開される．`(1, n)`は下記の 3 つ目のケースに，`(1, $n)`は下記の 4 つ目のケースに対応する．

```
(開始値)                <=> (開始値, [開始値..], _)
(開始値, 終了値のリスト) <=> (開始値, 終了値のリスト _)
(開始値, 終了値)        <=> (開始値, [終了値] _)
(開始値, 終端パターン)   <=> (開始値, [開始値..], 終端パターン)
```

一般に，添字範囲は，開始値と終了値のリスト，終端パターンの 3 つからなる．開始値は省略できない．終了値のリストが省略された場合は，開始値から始まる無限リストが終了値のリストとなる．終了値のリストがリストでなく，整数値であった場合，その整数値だけを含むリストに変換される．終端パターンが省略された場合，ワイルドカードが補完される．たとえば，`(1, n)`は`(1, [n], _)`，`(1, $n)`は`(1, [1..], $n)`にそれぞれ展開される．

ループ・パターンは，ツリーやグラフのパターンマッチをするときに特によく使われる．第 3 章 4 節でそのような例を紹介する．形式的なループ・パターンの構文と意味論は，ループ・パターンの論文 [7] で解説されている．

8　シーケンシャル・パターン

Egison 処理系はパターンを左から右に順番に処理する．しかし，パターンの右側で束縛されるパターン変数の値を参照したいなど，この処理の順番を変えたいことがある．**シーケンシャル・パ**

ターン (*sequential pattern*) はそのために用意された組み込みパターンである．

シーケンシャル・パターンは，パターンマッチの処理の順番をユーザーが変えることを許す．シーケンシャル・パターンは，パターンのリストとして表現される．パターンマッチは，リストの先頭から順番に実行される．以下のシーケンシャルパターンは，リストの 3 つ目の要素，1 つ目の要素，2 つ目の要素の順番でターゲットのリストをパターンマッチする．

```
matchAll [2,3,1,4,5] as list integer with
| {     @    ::     @    :: $x :: _,
   (#(x + 1),       @    ),
                #(x + 2)}
-> "Matched" -- ["Matched"]
```

シーケンシャル・パターン中に現れる@は，**後回しパターン変数**（*later pattern variable*）と呼ばれる．後回しパターン変数に束縛されたターゲットは，シーケンシャル・パターンの次の要素のパターンでパターンマッチされる．複数の後回しパターン変数が現れた場合，次のシーケンスはそれらをまとめてタプルとしてパターンマッチする．

シーケンシャル・パターンのこの特徴は，パターンの離れた複数の部分についてまとめて not パターンを適用することを可能にする．たとえば，以下のシーケンシャル・パターンは，ターゲットのコレクションのペアが，共通の要素をちょうど 1 つだけ含む場合に，パターンマッチに成功する．シーケンシャル・パターンは，それぞれのコレクションに共通の要素があることをチェックした後に，それぞれの残りの要素のコレクションにこれ以上共通要素がないことをチェックすることを可能にする．シーケンシャル・パターンと not パターンの組み合わせは，数学的なアルゴリズムを記述するときに現れることがある．たとえば，第 4 章 1 節でも現れる．

```
singleCommonElem xs ys := match (xs, ys) as (multiset eq, multiset eq) with
| {($x :: @, #x :: @),
   !($y :: _, #y :: _)} -> True
| _ -> False
```

シーケンシャル・パターンと同等のことが matchAll式のネストにより表現できるのではと考えた読者がいるかもしれない．実は，少なくとも 2 つの理由でこれは不可能である．1 つ目の理由は，ネストした matchAll式は，Egison 内部の幅優先探索を壊すことに由来する．外側の matchAll式の 2 番めの結果は，外側の matchAll式 1 つ目の結果について内側の matchAll式の結果をすべて評価した後に計算される．2 つ目の理由は，後回しパターン変数がターゲットだけでなく，マッチャーの情報も保持することである．matchAllの引数のマッチャーが，関数の引数に由来するパラメーターであることがある．それゆえ，内側の matchAll式で使うべきマッチャーが構文的に導けないことがある．

9　パターン関数によるパターンのモジュール化

コンス・パターンやジョイン・パターンのようなパターン・コンストラクタは，第 6 章で解説されるようにマッチャーで定義される．これらのパターン・コンストラクタを組み合わせて，新しいパターン・コンストラクタをつくることができる．そのためには，パターンを引数にとってパターンを返す**パターン関数**（*pattern function*）を使う．

パターン関数を定義する構文はラムダ式の構文と似ているが，矢印に->の代わりに=>を使う点が異なる．以下で定義されているパターン関数 twinは，それぞれの第一引数が同じ値である場合にマッチするような，二重にネストしたコンス・パターンをモジュール化している．パターン関数の仮引数は**変数パターン**（*variable pattern*）と呼ばれる．この例の変数パターンは，pat1と pat2である．パターン関数のボディで，変数パターンの中身を参照するときは，~を変数パターンの先頭に付加する．これは変数パターンをパターン・コンストラクタと区別するためである．

```
twin := \pat1 pat2 => (~pat1 & $x) :: #x :: ~pat2
```

この twinパターン関数は，以下のように動作する．

```
match [1, 1, 2, 3] as list integer with
| twin $n $ns -> [n, ns]
-- [1, [2, 3]]
```

10　マッチャー合成による新しいマッチャーの生成

ここまでに登場したマッチャーは，Egison 唯一の組み込みマッチャー somethingを除いて，すべてユーザーが定義することができる．マッチャーを定義するには，第 6 章で解説する matcher式を使うことが基本であるが，既存のマッチャーを組み合わせて新しいマッチャーをつくることもできる．この方法で，たとえば，多重集合のタプルのマッチャーや多重集合の多重集合のマッチャーを定義することができる．

タプルに対するマッチャーは，マッチャーのタプルにより表現される．**タプル・パターン**はそのようなマッチャーを使ったパターンマッチのときに使われる．たとえば，以下は，第 1 章 2 節でも登場した intersect関数の定義である．2 つの集合のタプルに対するマッチャーを使って，コレクションの共通要素をパターンマッチで取り出している．

```
intersect xs ys := matchAll (xs,ys) as (set eq, set eq) with
  | ($x :: _, #x :: _) -> x
```

上記で使われている eqマッチャーは，同値性が定義されたデータ型[*6]のパターンマッチに使える

[*6] 現在の Egison には静的な型はないが，Haskell でいうと Eq型クラスに属するデータ型

ユーザー定義マッチャーである．eqマッチャーに対して，値パターンが使われた場合，同値性のチェックによりパターンマッチがおこなわれる．

また，タプル・マッチャーと，マッチャーを引数にとってマッチャーを返す関数を組み合わせると，さまざまな非自由データ型に対するマッチャーが定義できる．たとえば，グラフの各ノードを整数，グラフの辺を 2 つのノードのペアとして表現したとき，有向グラフを表すマッチャーは次のように辺の多重集合として定義できる．

```
graph := multiset (integer, integer)
```

隣接リストにより表現したグラフのマッチャーも定義できる．隣接リストによるグラフは，整数と整数の多重集合のタプルの多重集合として定義される．ここでも整数によりノードの ID を表現している．

```
adjacencyGraph := multiset (integer, multiset integer)
```

代数的データ型に対するマッチャーは matcher式によっても定義できるが，代数的データ型に対するマッチャーを簡単な記述で定義できる特別な構文 algebraicDataMatcher式を Egison は提供している．algebraicDataMatcher式は糖衣構文で，matcher式に脱糖される．algebraicDataMatcher式を使えば，たとえば二分木に対するマッチャーは以下のように定義できる．

```
binaryTree a := algebraicDataMatcher
  | bLeaf a
  | bNode a (binaryTree a) (binaryTree a)
```

代数的データ型に対するマッチャーと非自由データ型に対するマッチャーも組み合わせることができる．たとえば，任意の数の子供をもつツリーで子供の順番を無視するマッチャーは以下のように定義できる．第 3 章 4 節でこのツリーに対するパターンマッチを紹介する．

```
tree a := algebraicDataMatcher
  | leaf a
  | node a (multiset (tree a))
```

第 3 章

パターンマッチ指向プログラミング入門

本章は，第 2 章でみてきたパターンマッチのための構文やさまざまな組み込みパターンを使って，どのようなプログラミングができるのかを解説する．そのために，Egison パターンマッチを活用したプログラミングにおいて頻出するパターンを紹介していく．本章で Egison のパターンマッチを使ったプログラミングの具体的なイメージをつかんでいってもらいたい．

1　パターンマッチによるリスト・プログラミング

本節は，リスト・プログラミングが Egison のパターンマッチによってどのように変わるのか観察する．ここでリスト・プログラミングとは，関数型プログラミング言語を含む一般のプログラミング言語のリスト・ライブラリで提供されているような関数を定義するためのプログラミングのことをいう．`_ ++ $x :: _`という 2 引数目がコンス・パターンのジョイン・パターンは，リスト・プログラミングにおいて頻出する．そのためこのパターンに名前をつけ，**ジョイン・コンス・パターン**（*join-cons pattern*）と呼んでいる．本節で示すように，リストに対する関数の多くがジョイン・コンス・パターンを使って定義できる．

1.1　単一のジョイン・コンス・パターン — map 関数とその仲間

`list`マッチャーについて，パターン`_ ++ $x :: _`は，ターゲットのそれぞれの要素にマッチする．その結果，下記の`matchAll`式は，`xs`のそれぞれの要素にマッチし，それらに関数`f`を適用した結果を返す．これはまさに`map`関数の定義である．

```
map f xs := matchAll xs as list something with
  | _ ++ $x :: _ -> f x
```

この`matchAll`式を変更して，さまざまな関数を定義できる．たとえば，`filter`関数は述語パターンを挿入して定義できる．

```
filter pred xs := matchAll xs as list something with
  | _ ++ (?pred & $x) :: _ -> x
```

member関数は値パターンを挿入することにより定義できる．member関数は第一引数の要素が第二引数のコレクションに含まれているかどうか判定する述語である．member関数は，match式を使って定義する．match式は，car (matchAll ...)[*1]のエイリアスとして実装されている．この実装が可能であるのは，Egison が matchAll式を遅延評価するためである．

```
member x xs := match xs as list eq with
  | _ ++ #x :: _ -> True
  | _            -> False
```

delete関数は，member関数をすこし編集して実装できる．delete関数は，第一引数 xの最初の出現を第二引数のコレクション xsから取り除く．

```
delete x xs := match xs as list eq with
  | $hs ++ #x :: $ts -> hs ++ ts
  | _                -> xs
```

述語 anyと everyも同様に match式により実装できる．anyは第二引数のコレクションの要素のうちに第一引数の述語を満たすものがあるかどうか判定する述語である．everyは第二引数のコレクションの要素のすべてが第一引数の述語を満たすかどうか判定する述語である．

```
any pred xs := match xs as list something with
  | _ ++ ?pred :: _ -> True
  | _               -> False

every pred xs := match xs as list something with
  | _ ++ !?pred :: _ -> False
  | _                -> True
```

1.2 ジョイン・コンス・パターンのネスト — unique・concat 関数

複数のジョイン・コンス・パターンを組み合わせることで，さらに強力なパターンをつくることができる．その例として unique関数を紹介する．unique関数をパターンマッチ指向プログラミングにより定義すると以下のようになる．

```
unique xs := matchAllDFS xs as list eq with
  | _ ++ $x :: !(_ ++ #x :: _) -> x
```

not パターンが，xが xのあとにそれ以上現れないことを表現するために使われている．その結果このパターンは，それぞれの要素の最後の出現にマッチする．

[*1] carはコレクションの先頭要素を取り出す関数である．

```
unique [1,2,3,2,4] -- [1,3,2,4]
```

述語パターンを上手に使って，それぞれの要素の最初の出現からなるコレクションを返すunique関数を定義することも可能である．最初の出現だけにマッチするために，その要素より前に同じ要素が出現しないときにマッチするパターンを書く必要がある．Egison のパターンマッチはパターンを左から右に処理するため，そのようなパターンは，コンス・パターンとジョイン・パターンを単純に組み合わせるだけでは書けない．

```
unique2 xs := matchAllDFS xs as list eq with
  | $hs ++ (!?(\x -> member x hs) & $x) :: _ -> x

unique2 [1,2,3,2,4] -- [1,2,3,4]
```

もうひとつのもっとエレガントな方法に，シーケンシャル・パターンを使う方法がある．同じ意味のパターンを，ジョイン・パターンの第一引数に後回しパターン変数を使うシーケンシャル・パターンにより表現できる．

```
unique xs := matchAllDFS xs as list eq with
  |  {@               ++ $x :: _,
      !(_ ++ #x :: _)}
  -> x
```

ネストしたジョイン・コンス・パターンのもうひとつの例に，第 2 章 5 節で紹介した concat関数がある．マッチャー合成（第 2 章 10 節）とジョイン・コンス・パターンを組み合わせることにより，concatをパターンマッチ指向プログラミングにより定義できる．

```
concat xss := matchAllDFS xss as list (list something) with
  | _ ++ (_ ++ $x :: _) :: _ -> x
```

2　多重集合プログラミング

多重集合を直接パターンマッチできることは，Egison の大きな特徴である．多重集合のパターンマッチを活かしたプログラミングのコツを本節では紹介したい．これ以降の本書に出てくるパターンマッチはほぼ多重集合のパターンマッチである．

2.1　多重集合のコンス・パターン

multisetマッチャーのコンス・パターンは，要素の順序を無視してコレクションのパターンマッチをするのに便利である．数学のアルゴリズムを記述するとき，このような状況はよく現れる．

簡単な例からはじめる．連想リストに対する lookup関数は，単一のコンス・パターンで定義できる．単一の多重集合のコンス・パターンは，リストのジョイン・コンス・パターンと置き換える

こともできる．

```
lookup k ls := match ls as multiset (eq, something) with
  | (#k, $x) :: _ -> x
```

ただし，ネストした多重集合のコンス・パターンは，リストのジョイン・コンス・パターンと置き換えることができない．k 重にネストした多重集合のコンス・パターンは $P(n,k) = \frac{n!}{(n-k)!}$ 通りの k 個の要素のパーミュテーションを列挙するために使えるのに対し，k 重にネストしたリストのジョイン・コンス・パターンは $C(n,k) = \frac{n!}{k!(n-k)!}$ 通りの k 個の要素の組み合わせを列挙するために使える．

```
matchAll [1, 2, 3] as list something with
| _ ++ $x :: _ ++ $y :: _ -> (x, y)
-- [(1,2),(1,3),(2,3)]

matchAll [1, 2, 3] as multiset something with
| $x :: $y :: _ -> (x, y)
-- [(1,2),(1,3),(2,1),(2,3),(3,1),(3,2)]
```

上記と同等なプログラムは，従来の関数型プログラミングの一般的な方法では，リスト内包表記によって以下のように記述できる．tailsは，引数のリストの後方部分からなる部分リストを返す関数である．splitsは，引数のリストを先頭部分のリストと残りのリストに分解したものを返す関数である．

```
[ (x, y) | x : ts <- tails [1, 2, 3], y <- ts ]
-- [(1,2),(1,3),(2,3)]

[ (x, y) | (hs, x : ts) <- splits [1, 2, 3], y <- hs ++ ts ]
-- [(1,2),(1,3),(2,1),(2,3),(3,1),(3,2)]
```

多重集合のためのライブラリは，従来のプログラミング言語にも用意されているものの，多重集合として扱いたいコレクションをリストとして捉え直す必要がある場面は多い．実際，上記のリスト内包表記による $C(n,2)$ 個の要素の組み合わせの列挙では，splits関数と ++関数という 2 つのリストを処理するための関数を使っており，対象のコレクションをリストとして捉え直した処理を記述している．多重集合に対するパターンマッチは，このようなコレクションをリストとして捉え直す処理をパターンの定義（この例の場合，multisetマッチャーの::パターンの定義）に隠すことによって，多重集合として扱いたいコレクションを直接多重集合として扱える機会を広げる．本書のこれ以降の部分は，多重集合に対するさまざまなパターンを，第 2 章で紹介したパターンと組み合わせて記述していく．

2.2 ポーカーの役判定

多重集合に対する非線形パターンの応用例としてポーカーの役判定を紹介する．多重集合に対する非線形パターンにより，ポーカーのすべての役はそれぞれ 1 つのパターンとして表現できる．[*2]

```
poker cs :=
  match cs as multiset card with
  | card $s $n :: card #s #(n-1) :: card #s #(n-2) :: card #s #(n-3) :: card #s #(n-4) ::
      _
    -> "Straight flush"
  | card _ $n :: card _ #n :: card _ #n :: card _ #n :: _ :: []
    -> "Four of a kind"
  | card _ $m :: card _ #m :: card _ #m :: card _ $n :: card _ #n :: []
    -> "Full house"
  | card $s _ :: card #s _ :: card #s _ :: card #s _ :: card #s _ :: []
    -> "Flush"
  | card _ $n :: card _ #(n-1) :: card _ #(n-2) :: card _ #(n-3) :: card _ #(n-4) :: []
    -> "Straight"
  | card _ $n :: card _ #n :: card _ #n :: _ :: _ :: []
    -> "Three of a kind"
  | card _ $m :: card _ #m :: card _ $n :: card _ #n :: _ :: []
    -> "Two pair"
  | card _ $n :: card _ #n :: _ :: _ :: _ :: []
    -> "One pair"
  | _ :: _ :: _ :: _ :: _ :: [] -> "Nothing"
```

この poker関数は以下のように動く．

```
poker [Card Spade 5, Card Spade 6, Card Spade 7, Card Spade 8, Card Spade 9]
-- "Straight flush"
poker [Card Spade 5, Card Diamond 5, Card Spade 7, Card Club 5, Card Heart 5]
-- "Four of a kind"
poker [Card Spade 5, Card Diamond 5, Card Spade 7, Card Club 5, Card Heart 7]
-- "Full house"
poker [Card Spade 5, Card Spade 6, Card Spade 7, Card Spade 13, Card Spade 9]
-- "Flush"
poker [Card Spade 5, Card Club 6, Card Spade 7, Card Spade 8, Card Spade 9]
-- "Straight"
poker [Card Spade 5, Card Diamond 5, Card Spade 7, Card Club 5, Card Heart 8]
-- "Three of a kind"
```

[*2] 著者が Egison のマッチ式を設計したとき，ポーカーの役判定をするプログラムを簡潔に記述できることを必要条件として設計した．

```
poker [Card Spade 5, Card Diamond 10, Card Spade 7, Card Club 5, Card Heart 10]
-- "Two pair"
poker [Card Spade 5, Card Diamond 10, Card Spade 7, Card Club 5, Card Heart 8]
-- "One pair"
poker [Card Spade 5, Card Spade 6, Card Spade 7, Card Spade 8, Card Diamond 11]
-- "Nothing"
```

パターンコンストラクタ（card）は小文字から始まり，データコンストラクタ（Card）は大文字から始まるというルールが Egison にはある．

なお，上の例でカードのマッチャーは以下のように代数的データマッチャーとして定義されている．

```
suit := algebraicDataMatcher
  | spade
  | heart
  | club
  | diamond

card := algebraicDataMatcher
  | card suit (mod 13)
```

3　タプル・パターンによるデータの比較

複数のデータをくらべたいという場面は，プログラミングにおいて頻出する．このような場面で，とくに not パターンと一緒に使われるタプル・パターンが役に立つことがおおい．たとえば，difference関数は，第 2 章 10 節に登場した intersect関数の実装に not パターンを挿入することにより定義できる．

```
difference xs ys := matchAll (xs, ys) as (set eq, set eq) with
  | ($x :: _, !(#x :: _)) -> x
```

!($x :: _, #x :: _)のように，not パターンの挿入位置を変えることにより，2 つのコレクションに共通要素が 1 つもない場合にパターンマッチに成功するパターンも記述できる．

これらのパターンをシーケンシャル・パターンと組み合わせることにより，さらに複雑なパターンを記述することができる．シーケンシャル・パターンは，not パターンを，パターンの複数の離れた部分にまとめて適用することを可能にするからである．たとえば，第 2 章 8 節では，2 つのコレクションに共通要素が 1 つしかないことをチェックする singleCommonElem関数をパターンマッチにより定義した．シーケンシャル・not パターンは，数学のアルゴリズムを記述するときに必要となることがある．たとえば，第 4 章 1 節でで紹介する SAT ソルバーの実装でもシーケンシャル・not パターンは現れる．

```
singleCommonElem := match (xs, ys) as (multiset eq, multiset eq) with
```

```
                    | [($x :: @, #x :: @),
                       !($y :: _, #y :: _)] -> True
                    | _ -> False
```

シーケンシャル・パターンをループ・パターンと組み合わせることも可能である．たとえば，2つのリストの共通の先頭部分にマッチするパターンを，シーケンシャル・ループ・パターンにより記述できる．

```
match (xs, ys) as (list eq, list eq) with
| loop $i (1,$n)
    [($x_i :: @, #x_i :: @), [...]]
    !($y :: _, #y :: _)
-> map (\i -> x_i) [1..n]
```

4　ループ・パターンによる再帰的パターン

ループ・パターンは，パターンの繰り返しを表現するために使われる．ループ・パターンは，シンプルなパターンコンストラクタを組み合わせて複雑なパターンを構築したり，パターン変数の数がパラメーターによって変わるパターンを記述するときに役に立つ．そのような状況では，複雑な再帰を記述する必要がてでくる．ループ・パターンを使うと，それらの再帰をパターンに押し込め，直感的な記述ができる．本節では，そのような例を紹介していく．

4.1　N クイーン問題

本節は，N クイーン問題を解くプログラムをみせることにより，技巧的であるがトリッキーなループ・パターンの例を紹介する．N クイーン問題とは，$n \times n$ のチェス盤に n 個のチェスのクイーンの駒をお互いが攻撃しあえないように配置する問題である．チェスのクイーンは，縦横斜め何マス離れている駒でも攻撃することができる．将棋でいうと，飛車と角行を合わせた動きをチェスのクイーンはできる．

4 クイーン問題をとくプログラムからはじめよう．このプログラムでは，4 つのクイーンの配置をリストで表現する．リストの n 番目の要素は，n 行目のクイーンが何列目の位置にあるかを表現する．このとき，解は，コレクション `[1,2,3,4]`の要素の順番が並び替わったコレクションである必要がある．2 つのクイーンを同じ行や列に配置できないためである．それゆえ，コレクション `[1,2,3,4]`を整数の多重集合としてパターンマッチする．クイーン同士が斜めのラインを共有できないという条件は，$a_1 \pm 1 \neq a_2$，$a_1 \pm 2 \neq a_3$，$a_2 \pm 1 \neq a_3$，$a_1 \pm 3 \neq a_4$，$a_2 \pm 2 \neq a_4$，$a_3 \pm 1 \neq a_4$ という条件により表現する．

```
matchAll [1,2,3,4] as multiset integer with
  $a_1 ::
   (!#(a_1 - 1) & !#(a_1 + 1) & $a_2) ::
```

```
    (!#(a_1 - 2) & !#(a_1 + 2) & !#(a_2 - 1) & !#(a_2 + 1) & $a_3) ::
     (!#(a_1 - 3) & !#(a_1 + 3) & !#(a_2 - 2) & !#(a_2 + 2) & !#(a_3 - 1) & !#(a_3 + 1) &
      $a_4) ::
       [] -> [a_1,a_2,a_3,a_4]
-- [[2,4,1,3],[3,1,4,2]]
```

上記のプログラムを n クイーン問題のために一般化するには，二重にネストしたループ・パターンを使うことができる．外側のループ・パターンの添字変数 iが，内側のループ・パターン添字範囲にて，内側のループの繰り返し回数の違いを表現するために使われている．また，前の繰り返しパターンで束縛された値，たとえば，$a_jに束縛された値が，#(a_j - (i - j))や#(a_j + (i - j))のように参照されていることにも注意してほしい．このように，添字付きパターン変数の非線形性（パターンの左側で添字付きパターン変数に束縛した値が参照できること）がうまく機能している．

```
nQueens n :=
  matchAll [1..n] as multiset integer with
  | $a_1 ::
     (loop $i (2,n)
           ((loop $j (1, i - 1)
                  (!#(a_j - (i - j)) & !#(a_j + (i - j)) & ...)
                  $a_i) :: ...)
           [] -> map (\i -> a_i) [1..n]

nQueens 4 -- [[2,4,1,3],[3,1,4,2]]
```

4.2　ツリーのパターンマッチ

本節は，ツリーのパターンマッチをみせることにより，ループ・パターンの実例を紹介する．本節のツリーのノードは，XML のように任意の個数の部分ツリーを子供としてもつ．また，これらのサブツリーは多重集合として扱われる．このようなツリーに対するマッチャーは第 2 章 10 節にて定義した．このマッチャーを本節では使う．

プログラミング言語をカテゴリー分けしたツリーにたいしてパターンを書く．treeDataがそのカテゴリーツリーを定義している．たとえば，"Egison"は，"pattern-match-oriented"（パターンマッチ指向）カテゴリーと，"Functional programming"（関数型プログラミング）カテゴリーの"Dynamically typed"（動的型付け）サブカテゴリーに属している．データコンストラクタ Nodeと Leafは大文字から始まる．

```
treeData :=
  Node "Programming language"
    [Node "pattern-match-oriented" [Leaf "Egison"],
     Node "Functional language"
       [Node "Strictly typed" [Leaf "OCaml", Leaf "Haskell", Leaf "Curry", Leaf "Coq"],
```

```
      Node "Dynamically typed" [Leaf "Egison", Leaf "Lisp", Leaf "Scheme", Leaf "Racket
    "]],
    Node "Logic programming" [Leaf "Prolog", Leaf "Curry"],
    Node "Object oriented" [Leaf "C++", Leaf "Java", Leaf "Ruby", Leaf "Python", Leaf "
    OCaml"]]
```

下記の matchAll式は，ある言語が属するカテゴリーを列挙する．ツリーの末端は任意の深さでありうるため，添字範囲の終了値がパターン（下記の場合$n）であるループ・パターンが使われている．このループ・パターンの三点リーダーパターンは，繰り返しパターンの末尾でない場所に位置している．繰り返しの場所をユーザーが指定できることも，正規表現のクリーネスター演算子にはないループ・パターンの強みのひとつである．パターンコンストラクタ nodeと leafは小文字から始まる．

```
ancestors x t := matchAll t as tree string with
  | loop $i (1,$n)
      (node $c_i (... ::  _))
      (leaf #x)
  -> map (\i -> c_i) [1..n]

ancestors "Egison" treeData
-- [["Programming language", "pattern-match-oriented"], ["Programming language", "
    Functional language", "Dynamically typed"]]
```

あるサブカテゴリーに属する言語をすべて列挙するようなパターンを記述することも可能である．二重にネストしたループ・パターンを使えば，そのサブカテゴリーが任意の深さにあってもよいようにできる．以下のパターンは，特定のカテゴリーに属するすべての言語を列挙する．ツリーのなかで現れる順番と同じ順番で言語を列挙するために，matchAllDFSが使われている．

```
descendants x t := matchAllDFS t as tree string with
  | loop _ (1,_)
      (node _ (... ::  _))
      (node #x ((loop _ (1,_)
                  (node _ (... :: _))
                  (leaf $y)) :: _))
  -> y

descendants "Functional language" treeData
-- ["OCaml", "Haskell", "Curry", "Coq", "Egison", "Lisp", "Scheme", "Racket"]
```

ツリーのパターンマッチができる DSL（domain-specific language）としては，XML path がある．ユーザー定義可能な数少ないパターンコンストラクタとループ・パターンを組み合わせて幅広いパターンを記述できることが Egison の利点である．XML path は，たとえば ancestorコマンドなど，多くの組み込みコマンドを使ってパターンを記述する．

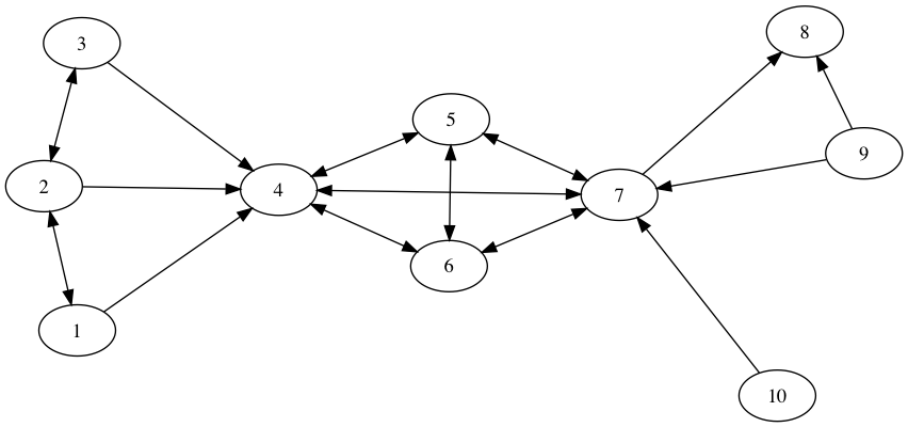

図 3.1　graphDataの描画

4.3　グラフのパターンマッチ

本節では，グラフに対するパターンマッチを紹介する．辺の集合として表現したグラフと，隣接リストとして表現したグラフの両方についてのパターンマッチを紹介する．

辺の集合としてのグラフ

本節は，辺の集合として表現したグラフに対してパターンマッチをおこなう．まず，以下のようなマッチャーとグラフを定義する．

```
graph := set edge
edge := algebraicDataMatcher
  | edge integer integer

graphData :=
  [ Edge 1 2, Edge 2 1, Edge 2 3,  Edge 2 4, Edge 3 4, Edge 4 5, Edge 4 6, Edge 4 7
  , Edge 5 4, Edge 5 6, Edge 5 7,  Edge 6 4, Edge 6 5, Edge 6 7, Edge 7 4, Edge 7 5
  , Edge 7 6, Edge 7 8, Edge 9 10, Edge 10 7 ]
```

パターンコンストラクタ edgeは小文字から始まり，データコンストラクタ Edgeは大文字から始まっている．図 3.1 は，上記のグラフを描画している．本節は，このグラフに対していくつかのパターンマッチを紹介する．

あるノード sから 2 辺でたどり着けるノードを列挙するパターンは以下のように書ける．

```
let s := 1 in
  matchAll graphData as graph with
  | edge (#s & $x_1) $x_2 :: edge #x_2 $x_3 :: _
  -> x
```

```
-- [1,3,4,5,6,7]
```

ノード sを始点とする辺でノード sとつながっているが，自身を始点とする辺がノード sに対してでていないノードを列挙するパターンは not パターンを使って以下のように書ける．

```
let s := 1 in
  matchAll graphData as graph with
  | edge #s $x :: !(edge #x #s :: _)
  -> x
-- [4]
```

ノード sからノード eへのパスすべてを列挙するパターンはループ・パターンを使って記述できる．Egison は，パターン中で let式を使うことを許している．この let式は，$x_1に sを束縛するために使われている．この let式のおかげで，ループ・パターンの最初の繰り返しパターンを特別扱いする必要がなくなっている．

```
let (s, e) := (1, 8) in
  matchAll graphData as graph with
  | let x_1 := s in
      loop $i (2, $n)
           (edge #x_(i - 1) $x_i :: ...)
           (edge #x_(n - 1) (#e & $x_n) :: _)
  -> map (\i -> x_i) [1..n]
-- [[1,4,7,8], ...]
```

サイズ n のクリーク（完全グラフになっている部分グラフ）を列挙するパターンは以下のように書ける．二重にネストしたループ・パターンを使う．

```
let n := 4 in
  matchAll graphData2 as graph with
  | edge $x_1 $x_2 :: loop $i (3, n)
                           (edge #x_1 $x_i :: loop $j (2, i - 1)
                                                   (edge #x_j #x_i :: ...)
                                                   ...)
                           _
  -> map (\i -> x_i) [1..n]
-- [[4,5,6,7],...]
```

本節は，有向グラフのパターンマッチを紹介した．上記の有向グラフに対するパターンを無向グラフに対するパターンに変更することは簡単で，Edge a bと Edge b aを同一視するようなマッチャーを使えばよい．そのような無向グラフに対するマッチャーは，第 6 章 1 節で紹介する unordered pair に対するマッチャーを使えば定義できる．

隣接グラフ

本節は，重み付き隣接リストにより表現したグラフのパターンマッチをおこなう．重み付き隣接リストに対するマッチャーは，マッチャー合成（第 2 章 10 節）により定義できる．下記のプログラム中の graphDataは隣接リストにより空路による都市間のネットワークを重み付き隣接リストにより表現している．2 つの都市間の移動にかかる時間が，重みとして整数により表現されている．

```
graph := multiset (string, multiset (string, integer))

graphData :=
  [("Berlin", [("New York", 14), ("London", 2), ("Tokyo", 14), ("Vancouver", 13)]),
   ("New York", [("Berlin", 14), ("London", 12), ("Tokyo", 18), ("Vancouver", 6)]),
   ("London", [("Berlin", 2), ("New York", 12), ("Tokyo", 15), ("Vancouver", 10)]),
   ("Tokyo", [("Berlin", 14), ("New York", 18), ("London", 15), ("Vancouver", 12)]),
   ("Vancouver", [("Berlin", 13), ("New York", 6), ("London", 10), ("Tokyo", 12)])]
```

以下のプログラム中のパターンは，ベルリンを出発して，すべての都市をまわって，ベルリンに戻るルートにマッチする．このパターンを使って，巡回セールスマン問題を解いている．非線形ループ・パターンを効果的に使っている．

```
trips :=
  let n := length graphData in
    matchAll graphData as graph with
    | (#"Berlin", (($s_1,$p_1) : _)) ::
        loop $i (2, n - 1)
          ((#s_(i - 1), ($s_i, $p_i) :: _) :: ...)
          ((#s_(n - 1), (#"Berlin" & $s_n, $p_n) :: _) :: [])
    -> sum (map (\i -> p_i) [1..n]), map (\i -> s_i) [1..n]

head (sortBy (\(_, x), (_, y) -> compare x y)) trips)
-- (["London", "New York", "Vancouver", "Tokyo"," Berlin"], 46)
```

ツリー（4.2 節）と同様に，グラフに対するパターンマッチのための DSL がいくつか存在する [11, 12, 1]．これらの DSL はグラフ・データベースに対するクエリー言語として開発されている．いくつかのユーザー定義パターン・コンストラクタと，ループ・パターンをはじめとする少数の組み込みパターンを組み合わせて多様なパターンを表現できることが，Egison のこれらの DSL にたいする利点である．

第 4 章

パターンマッチ指向プログラミングの効果

本章の目的は，パターンマッチ指向プログラミングがどのような場面でどのように役に立つか明確にすることである．前章までの例は，すべて単一の matchAllや matchで記述した例ばかりを紹介してきたが，本章ではより大きなプログラムの記述の中で Egison のパターンマッチがどのような役割を果たすのかみていく．

1　SAT ソルバーの実装

ある程度大きなプログラムを書く中で，パターンマッチ指向プログラミングの効果をみるために，SAT ソルバーを実装する．SAT ソルバーは，与えられた命題論理式が真になる論理変数の割り当てが存在するかどうか判定するプログラムである．SAT ソルバーが入力にとる論理式は，連言標準形（*conjunctive normal form*）であることが多い．論理式が連言標準形であるとは，論理式がリテラル（*literal*）の選言からなる節の連言になっていることをいう．リテラルとは，p または $\neg p$ という形をした論理式のことをいう．たとえば，$(p \vee q) \wedge (\neg p \vee r) \wedge (\neg p \vee \neg r)$ は，$p = \mathsf{False}$，$q = \mathsf{True}$，$r = \mathsf{True}$ という割り当てをすれば真になる連言標準形の論理式である．

1.1　Davis-Putnam アルゴリズム

ここでは Davis-Putnam アルゴリズム [10] という，SAT 問題を解くアルゴリズムのなかでもシンプルなアルゴリズムを実装する．本節の実装では，連言標準形の命題論理式をリテラルのコレクションのコレクションとして表現する．$\wedge$ と $\vee$ は可換な演算子であるため，このコレクションのコレクションは，リテラルの多重集合の多重集合としてパターンマッチできる．さらに，リテラルを整数として表現し，p の形のリテラルは正整数，$\neg p$ の形のリテラルは p に対応する正整数の -1 倍の負整数となるようにする．そうすると，これらの論理式に対するマッチャーは multiset (multiset integer)と定義できる．以下のプログラムは，Davis-Putnam アルゴリズム

の核の実装である．dp関数は，論理変数のリストと論理式を引数にとり，解がある場合は Trueを，そうでない場合は Falseを返す．

```
1  dp vars cnf :=
2    match (vars, cnf) as (multiset integer, multiset (multiset integer)) with
3    | (_, []) -> True
4    | (_, [] :: _) -> False
5    -- 1-literal rule
6    | (_, ($l :: []) :: _) -> dp (delete (abs l) vars) (assignTrue l cnf)
7    -- pure literal rule (positive)
8    | ($v :: $vs, !((#(neg v) :: _) :: _)) -> dp vs (assignTrue v cnf)
9    -- pure literal rule (negative)
10   | ($v :: $vs, !((#v :: _) :: _)) -> dp vs (assignTrue (neg v) cnf)
11   -- otherwise
12   | ($v :: $vs, _) ->
13     dp vs
14       ((resolveOn v cnf) ++ (deleteClausesWith v (deleteClausesWith (neg v) cnf)))
```

1 つ目のマッチ節（3 行目）は，入力された論理式が空だった場合，解を持つことを表現している．2 つ目のマッチ節（4 行目）は，入力された論理式が空節を含んでいた場合，解が存在しないことを表現している．3 つ目のマッチ節（6 行目）は，1-リテラル・ルール（*1-literal rule*）というルールを表現している．入力の論理式が 1 つのリテラルからなる節を持つとき，そのリテラルが Trueになる割り当てを即座にできる．4 つ目のマッチ節（8 行目）は，ある論理変数の否定がリテラルとして全く現れない場合，その論理変数に即座に Trueを割り当てることができることを表現している．たとえば，$(p \vee q) \wedge (\neg p \vee r) \wedge (\neg p \vee \neg r)$ は論理変数 q の否定を含まないため，q に Trueを割り当てることができる．5 つ目のマッチ節（10 行目）は，4 つ目のマッチ節と逆のことを表現している．このマッチ節は，ある論理変数の否定のみが入力の論理式に含まれる場合，その論理変数に Falseを割り当てることができることを表現している．最後のマッチ節（12-14 行目）は導出原理（*resolution principle*）を適用している．このマッチ節は，$p \vee C$ と $\neg p \vee D$ という形のすべての節のペアを列挙し（C と D はリテラルの選言とする），$C \vee D$ という形の節を生成する．

上記の dpの定義は，入力の論理式を簡約するためのすべてのルールを多重集合に対するパターンマッチをいかして記述している．同様のことをするために，伝統的な関数型プログラミングでは，複数のライブラリ関数を組み合わせたり，補助関数を定義する必要がある．Davis-Putnam アルゴリズムの OCaml による実装は [10] でみることができる．

1.2 導出原理

本節は，resolveOn関数の実装を紹介する．まずは，ナイーブな実装を紹介するところからはじめる．resolveOn関数は，matchAll式を使って以下のように定義できる．

```
resolveOn v cnf := matchAll cnf as multiset (multiset integer) with
```

```
| (#v :: $xs) :: (#(negate v) :: $ys) :: _ -> unique (filter (\c -> not (tautology c)) (
    xs ++ ys))
```

$p \vee C$ と $\neg p \vee D$ の形の節のペアを列挙するパターンは，多重集合の多重集合に対するパターンマッチにより記述できる．ただし，これだけでは，$C \vee D$ がトートロジーな節である場合（C がリテラル q を含みかつ D がリテラル $\neg q$ を含む場合そうなる）にもマッチしてしまう．そのために，filter関数と tautology関数を使って，そのような節を取り除いている．シーケンシャル・notパターン（第 3 章 3 節）を使って resolveOn関数を定義すると，パターンにそのような操作を取り込める．

```
resolveOn v cnf :=
  matchAll cnf as multiset (multiset integer) with
  | {(#v :: (@ & $xs)) :: (#(neg v) :: (@ & $ys)) :: _,
     !($l :: _, #(neg l) :: _)}
    ->  unique (xs ++ ys)
```

2　2 種類のループの分離

1 節で実装した SAT ソルバーは，パターンマッチ指向プログラミングによって取りのぞけないループを含んでいた．そのループとは，dp関数の再帰である．この再帰は，SAT 問題を解くための探索空間を狭めるために本質的なループである．この探索空間の縮小は，単純なバックトラッキング・アルゴリズムでは不可能なものである．たいして，その他のバックトラッキングにより実装できるループはすべてパターンの中に押し込められている．伝統的な関数型プログラミングでは，これら 2 種類のループのどちらも再帰を使って記述する必要があった．パターンマッチ指向プログラミングは，バックトラッキングにより実装できるループをパターンに閉じ込めることにより，時間的計算量を減らすための本質的なループのみの記述にプログラマが注力できるようにする．これにより，プログラムの記述しやすさ・読解しやすさが向上する．

第 5 章

Egison パターンマッチの仕組み

本章は，Egison 内部のパターンマッチの仕組みを解説する．まず，1 節で概略を説明する．2 節以降では，Egison パターンマッチを自身で実装するために必要な事項を解説する．単に Egison の使い方を知りたい読者は，2 節以降を読み飛ばして第 6 章に進んでも構わない．

1　パターンマッチ・アルゴリズムの概略

以下の matchAll式を実行したときにどのような処理がなされるのかみていくことにより，Egison 内部のパターンマッチ・アルゴリズムの概略をつかむ．

```
matchAll [2,8,2] as multiset eq with
| $m :: #m :: _ -> m
-- [2,2]
```

上記の matchAll式を受け取ると Egison 処理系は，以下のような**マッチング・ステート**（*matching state*）と呼ばれるオブジェクトを処理系内部で生成する．マッチング・ステートは，**マッチング・アトム**（*matching atom*）のスタックと，パターンマッチの途中結果（下記の場合，[]），matchAll式実行時の環境（下記の場合，env）からなる．マッチング・アトムは，パターン，マッチャー，ターゲットからなる 3 つ組である．初期マッチング・ステートは，matchAll式のパターン，マッチャー，ターゲットから構成されるマッチング・アトム 1 つからなるスタックから構成される．Egison 内部のパターンマッチ・アルゴリズムは，マッチング・ステートの簡約プロセスとして定義されている．簡約の結果，マッチング・アトムのスタックが空になるとパターンマッチに成功する．

```
MState [($m :: #m :: _, multiset eq, [2,8,2])]
       [] env
```

このマッチング・ステートは，次のステップで以下の 3 つのマッチング・ステートに簡約される．この簡約は multisetマッチャーのコンス・パターンの定義（第 6 章 3 節にて解説する）にもとづいて実行される．

```
MState [($m, eq, 2)
```

```
       ,(#m :: _, multiset eq, [8,2])]
       [] env

MState [($m, eq, 8)
       ,(#m :: _, multiset eq, [2,2])]
       [] env

MState [($m, eq, 2)
       ,(#m :: _, multiset eq, [2,8])]
       [] env
```

このように 1 つのマッチング・ステートを 1 ステップ簡約すると複数のマッチング・ステートが生成される．生成されるマッチング・ステートの数が 0 個であることも，無限個ある場合もある．これらのマッチング・ステートがどのような順番で簡約されるのかは，2 節で論じる．ここでは，1 つ目のマッチング・ステートの簡約をみる．1 つ目のマッチング・ステートは，次のステップで以下のように簡約される．スタックの先頭のマッチング・アトムのマッチャーが eqから somethingに変わる．この簡約は，eqマッチャーに定義されている．（第 6 章 2 節でその定義をみる．）

```
MState [($m, something, 2)
       ,(#m :: _, multiset eq, [8,2])]
       [] env
```

somethingは Egison 唯一の組み込みマッチャーであり，パターンマッチの途中結果に新しい束縛を追加する．ここでは，変数 mに 2を束縛している．

```
MState [(#m :: _, multiset eq, [8,2])]
       [(m, 2)] env
```

再び，multisetのコンス・パターンの定義にもとづき，マッチング・ステートが簡約される．

```
MState [(#m, eq, 8)
       ,(_, multiset eq, [2])]
       [(m, 2)] env

MState [(#m, eq, 2)
       ,(_, multiset eq, [8])]
       [(m, 2)] env
```

上記，1 つ目のマッチング・ステートは値パターン#mのパターンマッチに失敗して消える．（簡約された結果 0 個のマッチング・ステートを生成するともいうことができる．）2 つ目のマッチング・ステートは，先頭のマッチング・アトムが解決されて以下のようになる．

```
MState [(_, multiset eq, [8])]
       [(m, 2)] env
```

このマッチング・ステートも，先頭のマッチング・アトムが解決されて以下のようになる．

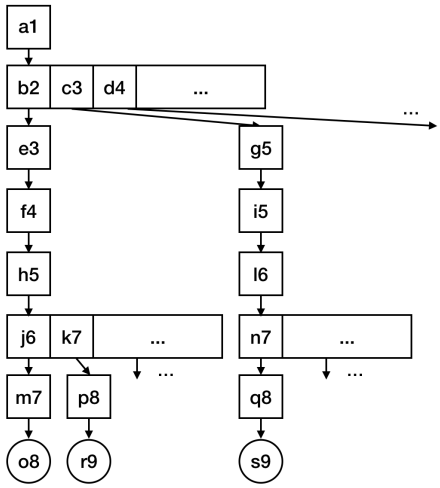

図 5.1　matchAllDFS式の探索木

```
MState []
       [(m, 2)] env
```

最終的にマッチング・アトムのスタックが上記のように空になると，このパターンマッチの途中結果が最終結果としてボディの評価に使われる．

2　matchAll・matchAllDFS による探索木のトラバース

本節では，matchAll式と matchAllDFS式を実行したときに内部でどのような探索木がトラバースされるのかを解説する．ある 1 つのマッチング・ステートを簡約すると 0 個から無限個までを含む複数のマッチング・ステートが生成される．そのため，Egison の内部のパターンマッチ・アルゴリズムは，初期マッチング・ステートを根とする木の探索と考えることができる．

まずは，以下のような matchAllDFS式の探索木をみる．

```
take 8 (matchAllDFS [1..] as set integer with
        | $m :: $n :: _ -> (m, n))
-- [(1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (1, 6), (1, 7), (1, 8)]
```

この matchAllDFS式の探索木は図 5.1 のようになる．図中の四角は，1 つのマッチング・ステートを表現している．図中の丸は，マッチング・アトムのスタックが空になったマッチング・ステート，パターンマッチの最終結果を表現している．matchAllDFS式はこの探索木を深さ優先探索する．

そのため，マッチング・ステートは，a1，b2，e3，f4，h5，j6，m7，o8，k7，p8，r9，... という順番で簡約されていく．マッチング・ステート h5 は無限個のマッチング・ステート（j6，k7，...）に簡約される．そのため，深さ優先探索では，マッチング・ステート c3 の簡約に行き着くこと

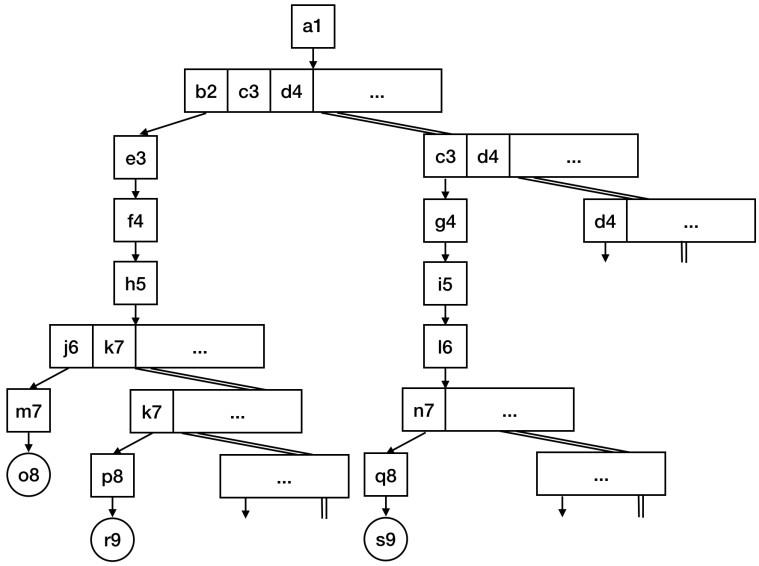

図 5.2　matchAll式の探索木

がない．このように matchAllDFSでは，すべてのマッチング・ステートをたどれず，可算無限のパターンマッチ結果をすべて列挙できない場合がある．

matchAll式は，可算無限のパターンマッチ結果をすべて列挙するために，この探索木に変形を加えて幅優先探索をおこなう．図 5.2 は，下記の matchAll式を実行したときの探索木である．

```
take 8 (matchAll [1..] as set integer with
        | $m :: $n :: _ -> (m, n))
-- [(1, 1), (1, 2), (2, 1), (1, 3), (2, 2), (3, 1), (1, 4), (2, 3)]
```

図 5.1 の探索木については，1 つのマッチング・ステートが 1 つのノードをあらわしていたが，図 5.2 の探索木については，一連なりのマッチング・ステートのリストを 1 つのノードとする．たとえば，a1 は 1 つのマッチング・ステートからなるノード，b2，c3，d4，... は無限個のマッチング・ステートからなる 1 つのノードである．このように探索木をとらえると，この探索木は二分木になっている．探索木が二分木であるため，すべてのノードの子が有限であるため，幅優先探索をすれば，すべてのノードをたどることができる．

図 5.1 の探索木を斜めにとらえることによって，図 5.2 の二分木への変形はできる．b2，c3，d4，... からなるノードに注目しよう．このノードの子は，e3 単独からなるノードと，c3，d4，... からなるノードである．一般に，あるノードの子は，そのノードの先頭のマッチング・ステートを簡約した結果と，そのノードの先頭の要素を取り除いたマッチング・ステートのリストとなる．

3　and パターン・or パターン・not パターンの実装

本節では，組み込みパターンの実装の例として，and パターン・or パターン・not パターン（第 2 章 6 節）の実装を説明する．

and パターンの実装からみていく．以下のような and パターンを含むパターンマッチを考える．

```
matchAll [1, 2, 3] as list integer with
| $n :: (_ :: _ & $rs) -> (n, rs)
-- [(1, [2, 3])]
```

このパターンマッチを処理する過程で，以下のようなマッチング・ステートの簡約にいきつく．

```
MState [(_ :: _ & $rs, [2, 3], list integer)]
       [(n, 1)] env
```

スタックの先頭のマッチング・アトムのパターンが and パターンであった場合，Egison 処理系は以下のように簡約する．

```
MState [(_ :: _, [2, 3], list integer)
       ,($rs, [2, 3], list integer)]
       [(n, 1)] env
```

and パターンのそれぞれの引数について，同じターゲットとマッチャーからつくったマッチング・アトムがスタックに追加される．

次に or パターンの実装をみる．以下のような or パターンを含むパターンマッチを考える．

```
matchAll [1, 1, 2] as list integer with
| $m :: ([] | #m :: _) -> m
-- [1]
```

このパターンマッチを処理する過程で，以下のようなマッチング・ステートの簡約にいきつく．

```
MState [(([] | #m :: _), [1, 2], list integer)]
       [(m, 1)] env
```

スタックの先頭のマッチング・アトムのパターンが or パターンであった場合，Egison 処理系は以下のように簡約する．

```
MState [([], [1, 2], list integer)]
       [(m, 1)]

MState [(#m :: _, [1, 2], list integer)]
       [(m, 1)] env
```

or パターンのそれぞれの引数について，同じターゲットとマッチャーからつくったマッチング・アトムをスタックの先頭にもつマッチング・ステートを生成する．

最後に not パターンの実装をみる．以下のような not パターンを含むパターンマッチを考える．

```
matchAll [2, 8, 2] as multiset integer with
| $m :: (!(#m :: _) & $rs) -> (m, rs)
-- [(8, [2,2])]
```

このパターンマッチを処理する過程で，以下のようなマッチング・ステートの簡約にいきつく．

```
MState [(!(#m :: _), [2, 2], multiset integer)
       ,($rs, [2, 2], multiset integer)]
       [(m, 8)] env
```

スタックの先頭のマッチング・アトムのパターンが not パターンであった場合，Egison 処理系は以下のように，スタックの先頭のマッチング・アトムから not パターンの not をのぞいたマッチング・アトムだけをスタックにもつマッチング・ステートを生成する．

```
MState [((#m :: _), [2, 2], multiset integer)
       [(m, 8)] env
```

このマッチング・ステートの簡約した結果，1 つもパターンマッチに成功するマッチング・ステートがなければ，以下のような先ほどのマッチング・ステートから not パターンをもつ先頭のマッチング・アトムをのぞいたマッチング・ステートを簡約する．

```
MState [($rs, [2, 2], multiset integer)]
       [(m, 8)] env
```

4　パターン関数の実装

本節は，パターン関数の実装を解説する．以下のようなパターン関数の適用を含むパターンマッチを考える．

```
twin := \p1 p2 => (~p1 & $x) :: #x :: ~p2

matchAll [1, 2, 1, 3] as multiset integer with
| $m :: twin $n _ -> (m, n)
-- [(2, 1), (2, 1), (3, 1), (3, 1)]
```

このパターンマッチを処理する過程で，以下のようなマッチング・ステートの簡約にいきつく．

```
MState [(twin $n _, [1, 1, 3], multiset integer)]
       [(m, 2)]
```

このようにスタックの先頭のマッチング・アトムのパターンが，パターン関数の適用であった場合，パターン関数が展開される．この展開と同時に，MNodeというほとんど MStateと同じ構造のデータにマッチング・アトムは変換される．MNodeは，パターンマッチの途中結果（下記の場合，

[]）と，パターン関数を定義したときの環境（下記の場合，env1），パターン関数の引数に束縛されたパターンの環境（下記の場合，[(p1, $n), (p2, _)]）をもつ．このように MStateがネストしたような構造をつくるのは，パターンの静的スコープを実現するためである．

```
MState [MNode [(~p1 & $x) :: #x :: ~p2, [1, 1, 3], multiset integer)]
              []
              env1
              [(p1, $n), (p2, _)]]
       [(m, 2)] env
```

このマッチング・ステートを簡約していくと，スタックの先頭のパターンがパターン関数の仮引数（下記の場合，p1）であるマッチング・ステートにいきつく．

```
MState [MNode [(p1, 1, integer),
               (#x :: ~p2, [1, 3], multiset integer)]
              [(x, 1)]
              env1
              [(p1, $n), (p2, _)]]
       [(m, 2)] env
```

パターン関数の引数に束縛されたパターンの環境（MNodeの 3 つ目の引数）から p1に束縛されているパターンを取り出し，マッチング・アトムのパターンをそのパターンに展開する．同時に，このマッチング・アトムを MNodeから MStateのマッチング・アトムに持ち上げる．

```
MState [($n, 1, integer),
        MNode [(#x :: ~p2, [1, 3], multiset integer)]
              [(x, 1)]
              env1
              [(p1, $n), (p2, _)]]
       [(m, 2)] env
```

第 6 章

マッチャーを定義しよう

本章は，ユーザーが自身でマッチャーを定義する方法を解説する．

1　マッチャーの定義の基礎

本節は，非自由データ型のなかでもっとも単純な unordered pair（要素の順番を無視するペア）のマッチャー定義をみていくことにより，マッチャー定義の方法の概略を解説する．

まずは，整数の unordred pair に対するマッチャー unorderedIntegerPairマッチャーの挙動を確認しよう．unorderedIntegerPairに対して，pairパターンを使うと以下のように 2 通りの要素の順番を両方パターンマッチする．

```
matchAll (1, 2) as unorderedIntegerPair with
| pair $x $y -> (x, y)
-- [(1,2),(2,1)]
```

そのおかげで，pairの第一引数がターゲットの 2 番目の要素である場合にも，パターンマッチに成功する．

```
matchAll (1, 2) as unorderedIntegerPair with
| pair #2 $y -> y
-- [1]
```

パターンがパターン変数であった場合には，ターゲットをそのまま束縛する．

```
matchAll (1, 2) as unorderedIntegerPair with
| $p -> p
-- [(1,2)]
```

上記のような動作をする unorderedIntegerPairマッチャーは以下のように定義できる．

```
1 unorderedIntegerPair := matcher
2   | pair $ $ as (integer, integer) with
3     | ($x, $y) -> [(x, y), (y, x)]
```

```
4   | $ as something with
5     | $tgt -> [tgt]
```

matcherは，マッチャーを生成するための組み込み構文である．

```
matcher
| 原始パターンパターン as ネクスト・マッチャー式 with
  | 原始データパターン -> ボディ
  ...
...
```

matcher式は，複数のマッチャー節（*matcher clause*）をとる．マッチャー節は，**原始パターンパターン**（*primitive pattern-pattern*），ネクスト・マッチャー式（*next matcher expression*），複数の原始マッチ節（*primitive match clause*）からなる．原始マッチ節は，**原始データパターン**（*primitive data pattern*）とボディからなる．

unorderedIntegerPairの 1 つ目のマッチャー節（2-3 行目）をみていく．まず，このマッチャー節の原始パターンパターンは pair $ $である．原始パターンパターンはパターンに対するパターンである．この原始パターンパターンは pairパターンにマッチし，このマッチャー節は pairパターンに対するパターンマッチの方法を定義している．pairのように，原始パターンパターン中にあらわれるパターンコンストラクタは，原始パターンパターンコンストラクタ（*primitive pattern-pattern constructor*）と呼ばれる．pairパターンの引数としてあらわれている$はパターン・ホール（*pattern hole*）と呼ばれる原始パターンパターンの構成要素で，任意のパターンがマッチする．パターン・ホールにマッチしたパターンは，ネクスト・パターン（*next pattern*）と呼ばれる．原始パターンパターンのパターンマッチは，一般の関数型プログラミング言語の代数的データ型に対するパターンマッチとほぼ同じようにおこなわれる．次に，このマッチャー節のネクスト・マッチャー式は，(integer, integer)である．これは，上記のパターン・ホールに束縛された 2 つのパターン（pairパターンの引数）がそれぞれ integerマッチャーを使って再帰的にパターンマッチされることを表現している．このマッチャー節は，1 つの原始マッチ節（3 行目）をとる．原始データパターンは，ターゲットとパターンマッチされる．原始データパターンのパターンマッチも，一般の関数型プログラミング言語の代数的データ型に対するパターンマッチとほぼ同じようにおこなわれる．原始マッチ節のボディは，ターゲットの分解結果を返す．(x, y)と(y, x)はネクスト・ターゲット（*next target*）と呼ばれ，ネクスト・パターンとネクスト・マッチャーを使って再帰的にパターンマッチされる．

2 つ目のマッチャー節（4-5 行目）の原始パターンパターンは，パターン・ホールである．パターン・ホールは任意のパターンにマッチするため，1 つ目のマッチャー節でマッチしなかったパターンはすべて 2 つ目のマッチ節で処理される．1 つ目のマッチャー節でマッチしないunorderedIntegerPairに対するパターンは，ワイルドカードかパターン変数のみであるので，このマッチャー節はこれらのパターンを処理する．このマッチャー節は，パターンとターゲットを変えず，ただマッチャーを somethingに変換するだけである．パターンとターゲットは somethingを

使ってパターンマッチされる．第 5 章 1 節でみたように somethingは，パターン変数に値を束縛することができる組み込みマッチャーである．

以下の unorderedPairのように，ペアの要素のデータ型に対するマッチャーを受け取る unordered pair に対するマッチャーを定義することができる．

```
matchAll (1, 2) as unorderedPair integer with
| pair $x $y -> (x, y)
-- [(1,2),(2,1)]
```

そのためには，unorderedPairをマッチャーを引数にとってマッチャーを返す関数として定義すればよい．pairパターンに対するマッチャー節のネクスト・マッチャー式で unorderedPairの引数である aを指定している．

```
unorderedPair a := matcher
  | pair $ $ as (a, a) with
    | ($x, $y) -> [(x, y), (y, x)]
  | $ as something with
    | $tgt -> [tgt]
```

2　eq マッチャーの定義

eqマッチャーも matcher式でユーザーが定義することができる．

```
1 eq := matcher
2   | #$val as () with
3     | $tgt -> if val == tgt then [()] else []
4   | $ as something with
5     | $tgt -> [tgt]
```

eqマッチャーの 1 つ目のマッチャー節（2-3 行目）は値パターンに対するマッチャー節である．#$valは値パターンにマッチする原始パターンパターンである．このような原始パターンパターンは原始値パターン（*primitive value pattern*）と呼ばれる．変数 valには値パターンの中身が束縛される．このマッチャー節の原始パターンパターンは，パターン・ホールを含んでいないため，ネクスト・マッチャー式は空タプルである．原始マッチ節で，値パターンの中身とターゲットの値が等しいかどうかチェックしている．

原始パターンパターンは，ここまでにでてきた 3 つの構成要素であるパターン・ホール，原始パターンパターンコンストラクタの適用，原始値パターンと，原始ワイルドカードからなる．

```
原始パターンパターン ::= $ | c 原始パターンパターン* | #$ID | _
```

原始ワイルドカードについては，5 節で用例を紹介する．

原始データパターンは，ワイルドカード，パターン変数，原始データパターンコンストラクタの適用からなる．

```
原始データパターン ::= _ | $ID | C 原始データパターン*
```

3 multiset マッチャーの定義

本節は，multisetマッチャーの定義を解説する．

```
1  multiset a := matcher
2    | [] as () with
3      | $tgt -> match tgt as (mutiset a) with
4                  | [] -> [()]
5                  | _ -> []
6    | $ :: $ as (a, multiset a) with
7      | $tgt -> matchAll tgt as list a with
8                  | $hs ++ $x :: $ts -> (x, hs ++ ts)
9    | #$val as () with
10     | $tgt -> match (val, tgt) as (list a, multiset a) with
11                 | ([], []) -> [()]
12                 | ($x :: $xs, #x :: #xs) -> [()]
13                 | (_, _) -> []
14   | $ as something with
15     | $tgt -> [tgt]
```

1 つ目と 4 つ目のマッチャー節はほぼ自明であるので，2 つ目と 3 つ目のマッチャー節をみていく．

2 つ目のマッチャー節（6-8 行目）をみていこう．原始プリミティブパターンは`$ :: $`で，ネクスト・マッチャー式は `(a, multiset a)`である．`a`は `multiset`の引数のマッチャーである．ネクスト・ターゲットは，`matchAll`式を使って定義されている．この `matchAll`式は，ターゲットが `[1,2,3]`である場合，`[(1,[2,3]),(2,[1,3]),(3,[1,2])]`を返す．このリストに含まれるそれぞれのネクスト・ターゲットは，ネクスト・パターンとネクスト・マッチャーを使って再帰的にパターンマッチされる．たとえば，ネクスト・ターゲット `1`と `[2,3]`は，ネクスト・マッチャー `a`と `multiset a`を使って，コンス・パターンの第 1 引数と第 2 引数のパターンとパターンマッチされる．

3 つ目のマッチ節をみていこう．このマッチャー節の原始パターンパターンは，原始値パターン`#$val`である．このマッチャー節は，値パターンを処理する．このマッチャー節は，値パターンの中身（`val`）とターゲット（`tgt`）が等しいかどうかくらべる．この `match`式の 1 つ目と 3 つ目のマッチ節は自明であるので説明を省略する．2 つ目のマッチ節のパターンは，`val`の先頭の要素を `tgt`から取り出す．そして，`val`と `tgt`から同じ要素を 1 つ抜いたコレクションを`$xs`と`#xs`というパターンを使って再帰的にパターンマッチしている．`#xs`のパターンマッチに，このマッチャー節自身が再帰的に呼び出されるが，コレクションの長さが 1 つずつ短くなっていくため，最終的に 1 つ目か 3 つ目のマッチ節どちらかにいきつく．

4　soretedList マッチャーの定義

二重にネストしたジョイン・コンス・パターンを使って $(p, p+6)$ という形の素数のペアを列挙するプログラムは，後方の素数のペアになるほど列挙に時間がかかるようになる．その理由は，二重にネストしたジョイン・コンス・パターンは，すべての素数の組み合わせを検査するためである．たとえば，このパターンは，$(3,5)$，$(3,7)$，$(3,11)$，$(3,13)$，$(3,17)$，$(3,19)$ のような素数のペアすべてを検査する．しかし，$(3,11)$ 以降の素数のペア，差が 6 より大きくなる最初の素数のペア，以降のペアについては，$(p, p+6)$ であるか検査する必要はない．

```
take 10 (matchAll primes as sortedList integer with
         | _ ++ $p :: (_ ++  #(p + 6) :: _) -> (p, p + 6))
-- [(5,11),(7,13),(11,17),(13,19),(17,23),(23,29),(31,37),(37,43),(41,47),(47,53)]
```

ソート済みリストに特化したマッチャーを使うことにより，この不必要な探索は避けることができる．通常の listマッチャーに，`$ ++ #$px : $`を原始パターンパターンにもつマッチャー節を追加すれば，ソート済みリストに特化したマッチャーをつくることができる．このマッチャー節が，$(p, p+6)$ という形の素数のペアにマッチする二重にネストしたジョイン・コンス・パターンの計算量を $O(n^2)$ から $O(n)$ に減らす．

```
sortedList a := matcher
  | $ ++ #$px :: $ as (sortedList a, sortedList a) with
    | \$tgt -> matchAll tgt as list a with
                | loop $i (1, $n)
                    ((?(\x -> x < px) & $h_i) :: ...)
                    (#px :: $ts)
                  -> (map (\i -> h_i) [1..n], ts)
  ...
```

5　原始ワイルドカードによる最適化

原始パターンパターン中で，原始ワイルドカードを使うことにより，ワイルドカードを特別扱いすることによって，ワイルドカードがパターン中に現れる特定のパターンの処理を高速化できる．本節は，3 節で紹介した multisetマッチャーについて，そのような最適化を紹介する．

本節で紹介する最適化の対象は，以下のようなコンス・パターンの第二引数がワイルドカードである場合である．この場合，対象のコレクションから要素を 1 つ除いた残りのコレクションを計算する必要はない．原始ワイルドカードを使うことにより，この計算を省くように multisetマッチャーに記述することができる．

```
1 matchAll [1, 2, 3, 4, 5] as multiset eq with
2 | $x :: _ -> x
```

```
-- [1, 2, 3, 4, 5]
```

このような最適化は，以下の 2-3 行目のマッチャー節を 3 節の multisetマッチャーの定義に追加すればできる．このマッチャー節の原始パターンパターンで原始ワイルドカードが使われている．パターン・ホールはコンス・パターンの第一引数だけであるので，ネクスト・マッチャーは aだけになり，ネクスト・ターゲットは tgtのそれぞれの要素であるために，tgtをそのまま返すようになっている．

```
multiset a := matcher
  | $ :: _ as a with
    | $tgt -> tgt
  | $ :: $ as (a, multiset a) with
    | $tgt -> matchAll tgt as list a with
                | $hs ++ $x :: $ts -> (x, hs ++ ts)
  ...
```

4 節の sortedListマッチャーも同様の最適化ができる．下記の 2-7 行目のマッチャー節は，ジョイン・パターンの第一引数を計算することを省略する役割を果たす．

```
sortedList a := matcher
  | _ ++ #$px :: $ as sortedList a
    | \tgt -> matchAll tgt as list a with
                | loop $i (1, _)
                    (?(\x -> x < px) :: ...)
                    (#px :: $ts)
                  -> ts
  | $ ++ #$px :: $ as (sortedList a, sortedList a)
    | \tgt -> matchAll tgt as list a with
                | loop $i (1, $n)
                    ((?(\x -> x < px) & $h_i) :: ...)
                    (#px :: $ts)
                  -> (map (\i -> h_i) [1..n], ts)
  ...
```

第 7 章

Egison パターンマッチの実装

本章は，Egison パターンマッチの実装を紹介する．Egison パターンマッチの実装については，5 個の実装と現在 2 本の論文が公開されている．本章はこれらの外部資料を読むヒントを提供する．

1　インタプリタ実装

Egison インタプリタ [5] は，本書で紹介した Egison のパターンマッチの全機能が実装された唯一の処理系である．このインタプリタは Haskell で実装されている．このインタプリタへの Egison パターンマッチ実装の方法については，第 5 章で紹介した．このインタプリタには，パターンマッチ以外に数式処理システムとしての機能も実装されている．

そのほかの Egison のインタプリタ実装には，河田旺さんによる Typed Egison [6] と Formalized Egison [3] がある．Typed Egison は，Egison インタプリタからパターンマッチに関する最小限の機能を切り出し，静的型システムを実装したものである．Formalized Egison は，Typed Egison の型安全性を証明するために Coq で実装されたインタプリタである．2020 年 2 月現在，証明を完成させるために開発を続けている最中である．

2　ほかの関数型言語へのライブラリ実装

Egison パターンマッチを使えるようにするためのライブラリが，Scheme と Haskell で提供されている．

Scheme については，Egison パターンマッチを Scheme マクロとして実装 [2] したライブラリが公開されている．この実装は，`match-all`式（Egison の `matchAll`に対応する）や `match`式を Scheme プログラムに変換した後 Scheme 処理系でコンパイルするため，Egison インタプリタにくらべて格段に速い．Egison パターンマッチを Scheme マクロとして実装する方法については論文 [8] にまとめられている．

Haskell メタプログラミングによる実装 [4] も公開されている．上記 Scheme マクロと同様に，`matchAll`式や `match`式を Haskell で実行できるプログラムに変換して，GHC（Haskell の標準的な

コンパイラ）でコンパイルして実行するため，Egison インタプリタにくらべて格段に速い．また GHC の型検査・型推論機能を使って，型検査と型推論もおこなうように実装されている．この Haskell ライブラリについては，現在論文を準備中である．

上記の Scheme 実装と Haskell 実装の両方について，ループ・パターンやシーケンシャル・パターンといった Egison 独自のパターンで実装されていないパターンがある．その理由は添字付き変数など，既存の言語の上に適切に実装するのが難しいプログラミング言語の機能を含んでいるためである．

付録 A

パターンマッチ指向プログラミング問題集

member 関数

下記の空白をうめて member関数を完成させよ.

```
member x xs :=
  match xs as list eq with
  | [          1          ] -> True
  | _ -> False

member 1 [1,2,3,4]
-- True

member 5 [1,2,3,4]
-- False
```

concat 関数

下記の空白をうめて concat関数を完成させよ.

```
concat xss :=
  matchAllDFS xss as [          1          ] with
  | [          2          ] -> [          3          ]

concat [[1,2],[3],[4,5]]
-- [1,2,3,4,5]
```

四つ子素数

四つ子素数を列挙するプログラムを記述せよ．四つ子素数とは，$(p, p+2, p+6, p+8)$ という形の素数の四つ組のことをいう．

```
take 4 (matchAll primes as list integer with
        | [        1        ] -> (p, p + 2, p + 6, p + 8))
-- [(5, 7, 11, 13), (11, 13, 17, 19), (101, 103, 107, 109), (191, 193, 197, 199)]
```

差が 6 の素数のペア

下記の空白をうめて素数のペア $(p, p+6)$ を抽出せよ．

```
take 6 (matchAll primes as list integer with
        | [        1        ] -> (p, p + 6))
-- [(5, 11), (7, 13), (11, 17), (13, 19), (17, 23), (23, 29)]
```

intersect 関数

下記の空白をうめて，2 つのコレクションの共通部分を抽出する intersect関数を完成させよ．

```
intersect xs ys :=
  matchAllDFS (xs, ys) as [        1        ] with
  | [        2        ] -> [        3        ]

intersect [1,2,3,4] [2,4,5]
-- [2,4]
```

difference 関数

下記の空白をうめて，第二引数のコレクションに含まれていない第一引数のコレクションの要素のみを抽出する difference関数を完成させよ．

```
difference xs ys :=
  matchAllDFS (xs, ys) as [        1        ] with
  | [        2        ] -> [        3        ]

difference [1,2,3,4] [2,4,5]
-- [1,3]
```

doubles 関数

コレクション中にちょうど 2 回現れる要素のみを抽出する doubles関数を完成させよ．

```
doubles xs :=
  matchAllDFS xs as [      1      ] with
  | [      2      ] -> [      3      ]

doubles [1,2,3,2,1,1,4,5,3]
-- [2,3]
```

tails 関数

tails関数をパターンマッチを使って定義せよ．ループ・パターンを使う回答とそうでない回答の 2 つを考えてみよ．

```
tails xs :=
  matchAll xs as list something with
  | [      1      ]
  -> ts

tails [1,2,3,4]
-- [[1, 2, 3, 4], [2, 3, 4], [3, 4], [4], []]
```

多重集合の同値性チェック

多重集合の同値性をチェックする 2 引数述語 equalMultisetをパターンマッチを使って定義してみよ．

```
equalMultiset xs ys :=
  match (xs, ys) as (list eq, multiset eq) with
  | ([], [])
  -> True
  | [      1      ]
  -> equalMultiset xs' ys'
  | _
  -> False

equalMultiset [1,1,2] [2,1,1]
-- True

equalMultiset [1,1,2] [1,2,2]
```

```
-- False
```

ループ・パターンを使って再帰関数を使わずに書いてみよ．

```
equalMultiset xs ys :=
  match (xs, ys) as (list eq, multiset eq) with
  | [          2          ]
  -> True
  | _
  -> False
```

ジョーカーの存在を考慮するポーカーの役判定

3 章 2.2 節のポーカーの役判定のためのパターンマッチ（poker関数）の実装を一切変更せずに，cardマッチャーの定義だけを変更して，ジョーカーの存在を考慮するポーカーの役判定をできるようにせよ．

```
card := matcher
  | card $ $ as (suit, mod 13) with
    | Card $s $n -> [(s, n)]
    | Joker -> [          1          ]
  | $ as something with
    | $tgt -> [tgt]

poker [Card Spade 5, Card Spade 6, Joker, Card Spade 8, Card Spade 9]
-- "Straight flush"
poker [Card Spade 5, Card Diamond 5, Joker, Card Club 5, Card Heart 7]
-- "Four of a kind"
```

参考文献

[1] The Neo4j Manual v2.3.3. https://neo4j.com/docs/stable/index.html, 2016. [Online; accessed 14-June-2018].

[2] egison/egison-scheme: Scheme Macros for Egison Pattern Matching. https://github.com/egison/egison-scheme, 2018. [Online; accessed 2019-09-21].

[3] akawashiro/formalized-egison. https://github.com/akawashiro/formalized-egison, 2019. [Online; accessed 2020-02-09].

[4] egison/egison-haskell: Template Haskell Implementation of Egison Pattern Matching. https://github.com/egison/egison-haskell, 2019. [Online; accessed 2019-09-21].

[5] egison/egison: The Egison Programming Language. https://github.com/egison/egison-haskell, 2019. [Online; accessed 2020-02-09].

[6] egison/typed-egison. https://github.com/egison/typed-egison, 2019. [Online; accessed 2020-02-09].

[7] S. Egi. Loop Patterns: Extension of Kleene Star Operator for More Expressive Pattern Matching against Arbitrary Data Structures. In *The Scheme and Functional Programming Workshop*, 2018.

[8] S. Egi. Scheme Macros for Non-linear Pattern Matching with Backtracking for Non-free Data Types. In *The Scheme and Functional Programming Workshop*, 2019.

[9] S. Egi and Y. Nishiwaki. Non-linear Pattern Matching with Backtracking for Non-free Data Types. In *Asian Symposium on Programming Languages and Systems*, pages 3–23. Springer, 2018.

[10] J. Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.

[11] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and Complexity of SPARQL. In *International semantic web conference*, 2006.

[12] M. A. Rodriguez. The Gremlin Graph Traversal Machine and Language. In *Proceedings of the 15th Symposium on Database Programming Languages*, 2015.

索引