プログラミング言語 Egison 入門

Egison 開発チーム

2021 年 5 月 24 日

はじめに

Egison は，「人間の頭の中の認識を，コンピュータ向けに翻訳することなしに，表現できる言語を作りたい」という動機で作られたプログラミング言語である．より幅広いアルゴリズムの簡潔な記述を可能にする新しいプログラミング言語は，既知のコンピュータ科学の問題のプログラミングを簡単にするだけでなく，従来プログラミングが難しいために敬遠されていた問題を扱いやすくし，コンピュータ科学の扱う範囲を広げる可能性を持つため重要である．

Egison に実装されているこれらのアイデアのうち，最初に実装された機能は本書の第一部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．本書の第一部は，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．

このパターンマッチを応用して，Egison には数式処理システムが実装されている．この数式処理システムには，任意のユーザー定義関数に対してテンソルの添字記法が適用できるという特徴がある．この機能によって，数学のなかでも特に計算が大変な微分幾何という分野に現れる計算を紙の上の数式に近い形でプログラムとして記述できる．そのほかにも実験的な新しい機能が Egison 上の数式処理システムには実装されており，第二部はこれらについて紹介する．

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や数式処理システム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．

2021 年 5 月 13 日

江木 聡志

目次

はじめに ii

第 1 章 プログラミング Egison とは 1

1.1 プログラミング言語全体の中での Egison の位置づけ 1

1.2 第 I 部 パターンマッチ指向プログラミング 3

1.3 第 II 部 数式処理システムとしての Egison 5

第 I 部 パターンマッチ指向プログラミング 7

第 2 章 Egison 早巡り 8

2.1 matchAll 式によるパターンマッチ 8

2.2 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 10

2.3 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 11

2.4 マッチャーによるパターンの拡張性と多相性 11

2.5 matchAllDFS 式によるパターンマッチ結果の順序の制御 12

2.6 and パターン・or パターン・not パターン 13

2.7 ループ・パターン 14

2.8 シーケンシャル・パターン 15

2.9 パターン関数によるパターンのモジュール化 17

2.10 マッチャー合成による新しいマッチャーの生成 17

第 3 章 パターンマッチ指向プログラミング入門 19

3.1 パターンマッチによるリスト・プログラミング 19

3.2 多重集合プログラミング 21

3.3 タプル・パターンによるデータの比較 24

3.4 ループ・パターンによる再帰的パターン 25

第 4 章 パターンマッチ指向プログラミングの効果 31

4.1 SAT ソルバーの実装 31

4.2 2 種類のループの分離 33

第 5 章 Egison パターンマッチの仕組み 34
5.1 パターンマッチ・アルゴリズムの概略 34
5.2 matchAll・matchAllDFS による探索木のトラバース 36
5.3 and パターン・or パターン・not パターンの実装 38
5.4 パターン関数の実装 39

第 6 章 マッチャーを定義しよう 41
6.1 マッチャーの定義の基礎 41
6.2 eq マッチャーの定義 43
6.3 multiset マッチャーの定義 44
6.4 soretedList マッチャーの定義 45
6.5 原始ワイルドカードによる最適化 45
6.6 assocMultiset マッチャー 46

第 7 章 関数型プログラミング言語としてのいくつかの機能 48
7.1 無名パラメーター関数 48
7.2 無名 matchAll 関数・無名 match 関数 49
7.3 中置演算子の定義 49
7.4 IO 入出力 51
7.5 seq 式 53

第 8 章 Sweet Egison - Egison パターンマッチの Haskell ライブラリ 55
8.1 Sweet Egison の使い方 55
8.2 Sweet Egison の仕組み 57
8.3 Sweet Egison の最適化 58
8.4 マッチャーとパターンの型 60
8.5 Sweet Egison のマッチャー定義 60
8.6 ユーザによる探索戦略の追加 61

第 II 部 数式処理システムとしての Egison 63

第 9 章 数式処理システム入門 64
9.1 未定義変数 = シンボル 64
9.2 関数適用のシンボル化 65
9.3 数式の簡約 65
9.4 二次方程式を解くプログラム - 数式を処理するアルゴリズムの記述 66

9.5 微分するプログラム - 数式に対するパターンマッチ 67
9.6 ベクトルや行列の計算 - 添字記法 69
9.7 数式の出力形式 69

第 10 章 数式データの扱い 71
10.1 数式データの内部表現 71
10.2 バッククオート（‘）による式展開の制御 72
10.3 シングルクオート（’）による関数適用の制御 73
10.4 withSymbols 式によるローカルシンボルの宣言 74
10.5 数式データに対するパターンマッチ 74

第 11 章 テンソル計算 77
11.1 テンソルとは 77
11.2 テンソルの添字記法 78
11.3 添字記法をプログラミングに導入するためのアイデア 79
11.4 スカラー仮引数とテンソル仮引数 80
11.5 三種類の添字 82
11.6 添字の簡約規則 83
11.7 テンソル関数の適用 84
11.8 省略された添字の補完 84
11.9 スカラー仮引数とテンソル仮引数の実装 85
11.10 反転スカラー仮引数 86
11.11 generateTensor 式によるテンソルの生成 86
11.12 テンソルの宣言 87
11.13 添字付加のために便利な構文 88
11.14 テンソルの対称性・歪対称性の宣言 89

第 12 章 関数シンボル 93
12.1 関数シンボルによるシンボリックな関数の引数の省略 93
12.2 関数シンボルを成分にもつテンソル 94
12.3 関数シンボルの内部表現とパターンマッチ 94
12.4 関数シンボルの実装方法 95

第 13 章 Egison で計算する微分幾何 96
13.1 微分幾何と Egison 96
13.2 リーマン曲率テンソルの計算 97
13.3 曲率形式の計算 - 微分形式を使った計算（1） 99
13.4 ホッジ・ラプラシアンの計算 - 微分形式を使った計算（2） 102

13.5 その他の微分形式についての演算子 - 内部積・リー微分 103

第 III 部 Egison の今後 104

第 14 章 Egison 開発の目標と方針 105
14.1 処理系の開発 105
14.2 アプリケーションの提示・開発 109
14.3 Egison の先にある研究 111

付録 A パターンマッチ指向プログラミング問題集 114

付録 B 数式処理システムとしての Egison の演習問題 118

参考文献 120

索引 121

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．

1.1　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．

また，このパターンマッチ機能を活かして Egison の上に数式処理システムが実装されている．数式処理システムとは，$x + x = 2x$ や $(x + y)^2 = x^2 + 2xy + y^2$ のようにシンボリックな計算ができるプログラミング言語のことをいう．Egison を使えば数式に対するパターンマッチが簡単に定義できるため，数式処理システムが他の言語にくらべて簡単に実装できる．この数式処理システムの簡潔な実装による拡張性を活かして，Egison にはいくつか数式処理システムとして新しい機能が実装されている．特にテンソルについての計算を記述するための機能が発展しており，従来ユーザーが定義することが困難であった微分幾何学の演算子を簡単に定義できるようになっている．

■新しい表記法の追求と科学研究　既存の言語では，我々の頭の中のイメージをそのまま表現できないことがある．このような不都合は，我々が頭の中でおこなっている抽象化を，言語では表現できないときに生じる．このような抽象化を表現可能にする新しい構文を発見して，言語に組み込むことができれば，この問題は解決される．実際，プログラミング言語について，このような抽象化をひとつ見つけて，作りはじめたのが Egison である．

そもそも，すべてのイメージを表現できるような言語を作ることが本当にできるのかという問題は非常に難しい．というのは，我々の頭の中のイメージは，対象に対する我々の理解が深まると同時に変化するものであるからである．その変化と同時に，我々にとって便利な表現も自然と変わっていく．

このことを実感するために，ある極端な例を考えてみる．

数の概念はすでに知っているけれども，たし算やかけ算の概念は知らないひとがいるとする．そのひとにとっては，10 進法は理解不能な概念であり，10 進表記もまったく理解不能な表記である．しかし，たし算やかけ算の概念とその性質をある程度知っているひとにとっては，10 進表記は非常に便利で，直感的な数の表現である．

たし算・かけ算を知らないというのは極端な例に思うかもしれない．しかし，我々がまだ知らない重要な概念は科学史をふりかえってみるとまだまだあってもおかしくない．このような概念が発見され，自然に対する我々の認識が深まれば，それを表現するために新しい表記法が言語に生まれる．逆に，このようにして生まれた新しい表記法は我々が自然をより深く理解することを助け，その結果，さらなる新しい表記法が生まれる．このように，表記法の進化と科学の発展は表裏一体の関係にある．

ゆえに，表記法について研究することには大きな意義があるだろう．「新しい表記法を見つ

けるための一般的な方法はあるのだろうか？」，「よりよい表記法が満たす一般的法則はあるのだろうか？」，「表記法の表現力を測る一般的な方法はあるのだろうか？」など興味深い問題はいくつも存在する．これらの問題の答えが見えたとき，まったく新しい世界観が生まれると筆者は期待している．新しいプログラミング言語を作ることをとおして，これらの問題の答えに近づきたいというのが，Egison を作った動機である．

■独自のプログラミング言語を用意する理由　Egison に実装されている新しいプログラミング言語の機能を，独自のプログラミング言語 Egison にではなく，既存のプログラミング言語を拡張して実装するという手段もある．このような手段を選ばず，独自のプログラミング言語 Egison を作る理由は，そのほうが新しい機能をすばやく確実に実装できるためである．既存のプログラミング言語に新しく発明した機能を追加するには，その新機能が既存の機能と競合し，言語の実装の広範囲に影響を与える場合があるために，新機能本体の実装以外のための大規模な調査や実装が必要になってしまう場合がある．実際，Egison のパターンマッチとテンソルの添字記法は両者ともそのような機能である．たとえば，Haskell に Egison のパターンマッチを実装しようとしたら，パターンマッチ機能の実装に加えて型システムの設計と実装もせねばならない．また，Haskell にテンソルの添字記法を実装する場合も，シンボリックな添字を扱うためのシンボル計算の機能や，シンボリックな添字をもつテンソルを扱うためのデータ構造の設計や型システムの設計・実装も必要となる．たいして，自身で発明した新機能を実装するために設計した独自のプログラミング言語を用意すれば，新機能の実装を最低限の労力で実装できる．ただし，独自のプログラミング言語を用意することには，既知の機能を再実装する必要があったり，新規ユーザーを得るのが難しいというデメリットもある．

2019 年からは既存のメジャーなプログラミング言語に Egison の機能を実装することにも取り組み始めている．他言語への Egison の機能の移植については，14.1.2 節で言及する．

1.2　第 I 部 パターンマッチ指向プログラミング

Egison が提唱する新しいプログラミング・パラダイムであるパターンマッチ指向プログラミングについて，第 I 部は解説する．

1.2.1　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．

```
def 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．

```
def 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．

```
def 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．

1.2.2　第 I 部の構成

第 I 部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簡単な機能については，登場したときに軽く解説するにとどめた．

第 I 部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，実際にプログラミングするときに便利な，無名パラメータ関数や IO 入出力するための機能について解説する．第 8 章では，Egison のパターンマッチを Haskell で使うためのライブラリ Sweet Egison の使い方を紹介する．

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 5.1 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 章を読むことができる．

1.3　第 II 部 数式処理システムとしての Egison

数式処理システムとして Egison がもつ機能を第 II 部は紹介する．

1.3.1　数式処理システムとしての Egison とその特徴

数式処理システムとは，$x + x = 2x$ のようにシンボリックな計算ができるプログラミング言語のことである．このようなシンボリックな計算は，我々が日頃おこなっている計算の大きな部分を占める．実際，中学校以降の数学で現れる計算のほとんどはシンボリックな計算を含む．そのため，シンボリックな計算をできるプログラミング言語を作ることは重要である．

数式処理システムを作ることがむずかしい原因は，シンボリックな変数を含む数式が一つの定まった形を持たない非自由データ型であることである．たとえば，$(x + y)^2$ という数式は，

$x^2 + 2xy + y^2$ という形の数式とも同値である．Egison の非自由データ型に対しても適用可能なパターンマッチ機能を使うと数式の書き換え処理を簡潔に実装できるために数式処理システムを，既存のプログラミング言語よりもかなり少ないコード量で実装することができる．このような考えをもとに 2016 年に Egison で数式処理システムは実装された．この数式処理システムは Egison 処理系と統合されており，Egison 上で使うことができる．

この数式処理システムの実装は，ほかの数式処理システムの実装にくらべてシンプルである．そのために機能拡張がしやすいというメリットがある．実際，このおかげで，Egison 独自の重要な特徴であるテンソルの添字記法をプログラミングへの導入についても，簡単に実装することができた．テンソルの添字記法をプログラミング言語に導入する手法は，この数式処理システムを使って微分幾何の計算をするプログラムのサンプルを作っている最中に発案され，数週間で実装された．

プログラミング言語に数式処理システムの機能を実装することには，実用的なものが作りやすいという利点がある．数式処理システムは，手で計算するより速ければ喜ばれるケースが多く，通常のプログラミング言語とくらべて，実行速度がシビアに求められない．実行速度よりも，普段手で書いている数学記法をプログラムでも記述できることのほうが数学の研究者にとってはうれしいことが多い．そのため，数式処理システムの実装は，新しい記法のプログラミングへの導入を目指す Egison にとって格好の課題といえる．

1.3.2　第 II 部の構成

第 II 部の最初に数式処理システムの使い方について解説したあと，Egison のもつ数式処理システムとしての機能を解説する．

第 II 部の構成は以下の通りである．第 9 章では，数式処理システムの特徴であるシンボリックな計算の方法と，具体的なプログラムの例を紹介する．第 10 章では，Egison 内部で数式データがどう表現されているのか解説する．第 11 章では，テンソルの添字記法を使ったプログラムの記述の方法を解説する．第 12 章では，関数シンボルについて解説する．第 13 章では，ここまで解説した機能の実践的な応用として，微分幾何の計算をするプログラムを紹介する．

第 I 部

パターンマッチ指向プログラミング

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（2.1 節，2.2 節，2.3 節，2.4 節，2.5 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（2.6 節，2.7 節，2.8 節，2.9 節，2.10 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．

2.1 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，2.4 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．

2.2 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，2.8 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．

```
def 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．

```
def 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．

2.3 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．（2.2 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．

2.4 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"]
```

2.5 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 [7] 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．

```
def 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]
```

2.6　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．

```
def 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)]
```

2.7 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．

```
def 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．

```
def 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．

```
1 (loop $i (1, 3) (_ ++ $x_i :: ...) _)
2 _ ++ $x_1 :: (loop $i (2, 3) (_ ++ $x_i :: ...) _)
3 _ ++ $x_1 :: _ ++ $x_2 :: (loop $i (3, 3) (_ ++ $x_i :: ...) _)
4 _ ++ $x_1 :: _ ++ $x_2 :: _ ++ $x_3 :: (loop $i (4, 3) (_ ++ $x_i :: ...) _)
5 _ ++ $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 3.4 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 [5] で解説されている．

2.8 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 4.1 節でも現れる．

```
def 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．

2.9 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．

```
def 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]]
```

2.10 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 1.2 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．

```
def 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．

```
def 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．

```
def 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．

```
def 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 3.4 節でこのツリーに対するパターンマッチを紹介する．

```
def 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．

3.1　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．

3.1.1　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．

```
def 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．

```
def 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．

```
def 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．

```
def 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．

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

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

3.1.2 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．

```
def 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．

```
def 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．

```
def 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 2.5 節で紹介した concat関数がある．マッチャー合成（第 2 章 2.10 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．

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

3.2 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．

3.2.1 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．

```
def 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．

3.2.2　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] “[p_1，p_2，…，p_n]” というパターンは，“$p1$: p_2 : … : p_n : []” に展開される糖衣構文である．

```
def 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]
```

[*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．

```
-- "Three of a kind"
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．

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

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

3.3 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 2.10 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．

```
def 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 2.8 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 4.1 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．

```
def 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]
```

3.4 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でてくる．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．

3.4.1 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．

```
def 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]]
```

3.4.2 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 2.10 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．

```
def 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．

```
def 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．

```
def 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．

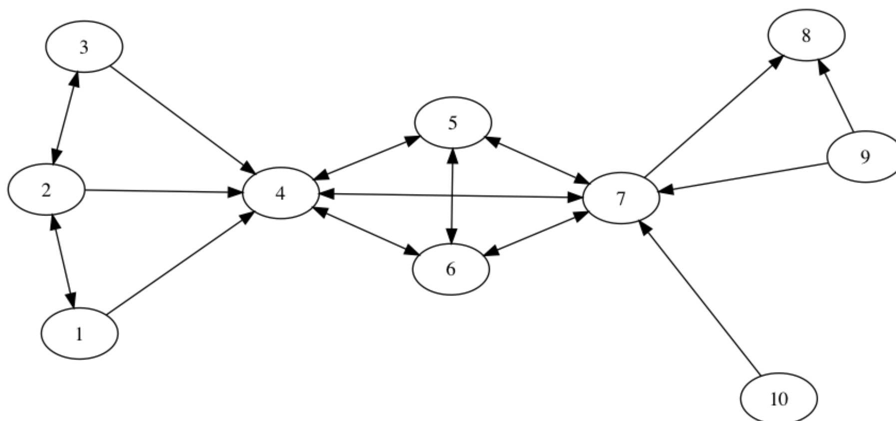

図 3.1: graphDataの描画

3.4.3 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．

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

def 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 6.1 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 2.10 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．

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

def 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．

```
def 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（3.4.2 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．

4.1　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，$p =$ `False`，$q =$ `True`，$r =$ `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．

4.1.1　Davis-Putnam アルゴリズム

ここでは Davis-Putnam アルゴリズム [8] 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．

```
def dp vars cnf :=
  match (vars, cnf) as (multiset integer, multiset (multiset integer)) with
  | (_, []) -> True
  | (_, [] :: _) -> False
  -- 1-literal rule
  | (_, ($l :: []) :: _) -> dp (delete (abs l) vars) (assignTrue l cnf)
  -- pure literal rule (positive)
  | ($v :: $vs, !((#(neg v) :: _) :: _)) -> dp vs (assignTrue v cnf)
  -- pure literal rule (negative)
  | ($v :: $vs, !((#v :: _) :: _)) -> dp vs (assignTrue (neg v) cnf)
  -- otherwise
  | ($v :: $vs, _) ->
    dp vs
      ((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 [8] でみることができる．

4.1.2 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．

```
def 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 3.3 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．

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

4.2　2 種類のループの分離

4.1 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，5.1 節で概略を説明する．5.2 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，5.2 節以降を読み飛ばして第 6 章に進んでも構わない．

5.1　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 6.3 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，5.2 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 6.2 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.

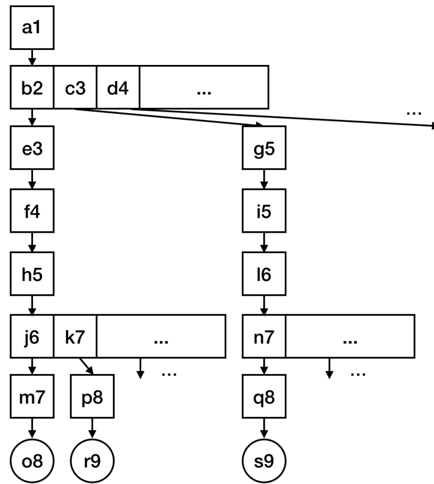

図 5.1: matchAllDFS式の探索木

```
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．

5.2 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

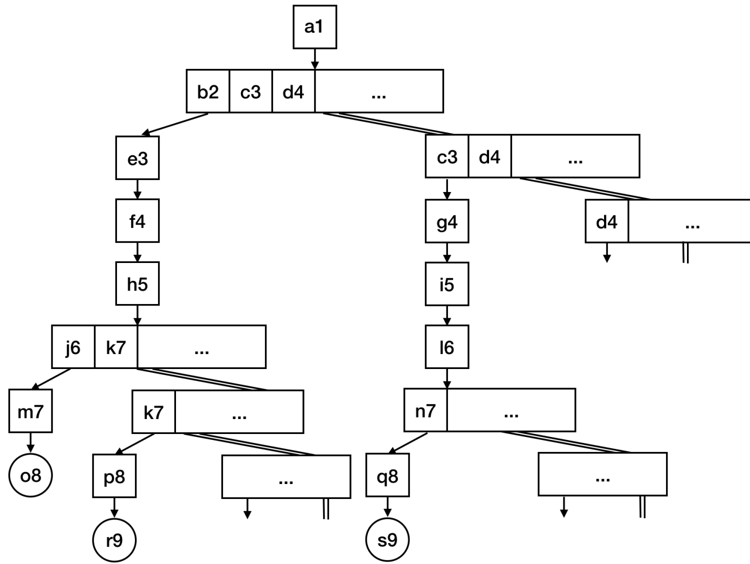

図 5.2: matchAll式の探索木

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．

5.3 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 2.6 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
```

5.4　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．

```
def 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．

6.1　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 def 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 5.1 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．

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

6.2 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 def 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，6.5 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*
```

6.3 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  def 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．

6.4 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．

```
def 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)
  ...
```

このようにネストしたパターンに対して直接パターンマッチのアルゴリズムを定義することによる最適化のことを，**パターン・フュージョン**（*pattern fusion*）と呼ぶ．

6.5 原始ワイルドカードによる最適化

パターンがワイルドカードを含む場合，ワイルドカードにマッチするターゲットの計算を省くことにより，パターンマッチの処理を高速化できる．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，Egison ではこの最適化をおこなえる．本節は，6.3 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．このような最適化のことを，**ワイルドカード最適化**（*wildcard optimization*）と呼ぶ．

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．

```
matchAll [1, 2, 3, 4, 5] as multiset eq with
| $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 6.3 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．

```
def 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)
  ...
```

6.4 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．

```
def 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)
  ...
```

6.6 assocMultiset マッチャー

Egison には，多重集合に対するマッチャーとして，assocMultisetが用意されている．本節はその使い方と定義を紹介する．

多重集合は，要素のその個数の連想リストとして表現できる．assocMultisetマッチャーは，このように連想リストとして表現された多重集合に対するマッチャーである．たとえば，以下のプログラムは 1 を 4 個，2 を 3 個，3 を 1 個もつ多重集合に対してパターンマッチをしている．ターゲットの多重集合に 3 個以上現れる要素を取り出すパターンが表現されている．

```
1 matchAll [(1, 4), (2, 3), (3, 1)] as assocMultiset eq with
2   | $x ^ #3 :: $rs -> (x, rs)
3 -- [(1, [(1, 1), (2, 3), (3, 2)]), (2, [(1, 4), (3, 2)])]
```

第 7 章

関数型プログラミング言語としてのいくつかの機能

本章は，関数型のプログラムの記述を便利にするための Egison の機能をいくつか紹介する．

7.1　無名パラメーター関数

無名パラメーター関数（anonymous parameter function）を使うと引数に名前をつけずに関数を定義できる．ラムダ式を使うと関数の命名を省略できる．引数の命名まで省略できる無名パラメーター関数は，これをさらにおしすすめたものと考えることができる．たとえば，第一引数を 10 倍して第二引数と足し合わせる無名パラメーター関数は以下のように定義できる．

```
2#(%1 * 10 + %2) 1 2
-- 12
```

#の前の 2は 2 引数の関数を定義していることを意味する．関数のボディの中で現れる %1と %2はそれぞれ第一引数と第二引数を表している．

無名パラメータ関数には，引数にタプルをとるものとリストをとるものの 2 つの変種がある．この 2 つの変種は#の前に指定する引数の数を括弧で囲むことによって定義する．#の前の数を ()で囲むとタプルを引数に取る無名パラメーター関数を定義できる．

```
(2)#(%1, %2) (10, 20)
-- (10, 20)
```

#の前の数を []で囲むとリストを引数に取る無名パラメーター関数を定義できる．

```
[2]#(%2, %1) [10, 20]
-- (20, 10)
```

7.2 無名 matchAll 関数・無名 match 関数

関数定義のとき，仮引数をすぐにパターンマッチすることが多い．するとその仮引数の名前がそのパターンマッチのターゲットとしてしか役に立たない．この問題を解決するために無名matchAll関数と無名match関数がある．

まず，無名matchAll関数を紹介する．\にスペースを入れずにmatchAllを続けることで無名matchAll関数は定義される．このときっパターンマッチのターゲットは省略される．この無名matchAll関数の引数がターゲットとなるからである．

```
\matchAll as list something with
  | $hs ++ $ts -> (hs, ts)
```

無名match関数は，無名matchAll関数のmatchAllをmatchに変えるだけで定義できる．

```
\match as list something with
  | nil -> True
  | _ -> False
```

7.3 中置演算子の定義

Egison はユーザーが新しく中置演算子を定義する機能を提供している．中置演算子を宣言するには，infix，またはinfixl，infixrを使う．infix，infixl，infixrはそれぞれ結合性のない演算子，左結合の演算子，右結合の演算子を定義するのに使う．infix，infixl，infixrは共通して三つの引数をとる．第一引数には，これから定義する中置演算子が関数であるのか，パターンコンストラクタであるのか指定するために，expressionかpatternというキーワードをとる．第二引数には，これから定義する中置演算子の優先度を整数値でとる．第三引数には，これから定義する中置演算子の名前をとる．

中置演算子として使う関数を定義する例として論理積&&を定義すると以下のようになる．

```
infixr expression 5 &&

def (&&) a b := match (a, b) as (eq, eq) with
  | (#True, #True) -> True
  | _              -> False
```

中置演算子として使うパターンコンストラクタを定義する例としてジョイン・パターン++を定義すると以下のようになる．

```
infixl pattern 7 ++

def list a := matcher
```

```
| $ ++ $ as (list a, list a) with
  ...
```

<table>
<tr><th>演算子</th><th>演算子の内容</th><th>優先度</th></tr>
<tr><td>^</td><td>べき乗</td><td>8</td></tr>
<tr><td>*</td><td>掛け算</td><td>7</td></tr>
<tr><td>/</td><td>割り算</td><td>7</td></tr>
<tr><td>%</td><td>割り算の余り</td><td>7</td></tr>
<tr><td>.</td><td>テンソルの掛け算</td><td>7</td></tr>
<tr><td>+</td><td>足し算</td><td>6</td></tr>
<tr><td>-</td><td>引き算</td><td>6</td></tr>
<tr><td>::</td><td>コンス</td><td>5</td></tr>
<tr><td>++</td><td>アペンド</td><td>5</td></tr>
<tr><td>=, <= >=, <, ></td><td>比較</td><td>4</td></tr>
<tr><td>&&</td><td>論理積</td><td>3</td></tr>
<tr><td>||</td><td>論理和</td><td>2</td></tr>
<tr><td>$</td><td>関数適用</td><td>0</td></tr>
</table>

表 7.1: 中置演算子一覧（関数）

<table>
<tr><th>演算子</th><th>演算子の内容</th><th>優先度</th></tr>
<tr><td>^</td><td>べき乗</td><td>8</td></tr>
<tr><td>*</td><td>掛け算</td><td>7</td></tr>
<tr><td>/</td><td>割り算</td><td>7</td></tr>
<tr><td>+</td><td>足し算</td><td>6</td></tr>
<tr><td>::</td><td>コンス</td><td>5</td></tr>
<tr><td>++</td><td>アペンド</td><td>5</td></tr>
<tr><td>&</td><td>and パターン</td><td>3</td></tr>
<tr><td>|</td><td>or パターン</td><td>2</td></tr>
</table>

表 7.2: 中置演算子一覧（パターン）

7.4 IO 入出力

遅延評価を基本の評価戦略とする Egison は，副作用をもつプログラムを記述するための特別な構文をもつ．Egison は静的型システムを持つ言語ではないが，Haskell と同じような方法で副作用をもつプログラムを記述する．

7.4.1 main 関数

たとえば，”Hello world!” という文字列を出力するプログラムは以下のように書ける．

```
def main args := write "Hello world!\n"
```

コマンドラインオプション-tがない場合は，Egison は mainの関数を実行する．上記のプログラムは以下のように実行できる．

```
$ cat hello.egi
def main args := write "Hello world!\n"
$ egison hello.egi
Hello world!
```

writeは Egison に組み込みで実装されている IO 関数である．writeは第一引数に文字列をとり，その文字列を標準出力に印字する．IO 関数には writeは以外にもいくつもある．図 7.3 にその一覧をまとめた．

mainの第一引数にはコマンドライン引数がはいる．コマンドライン引数は文字列として mainの第一引数に渡される．

```
$ cat args.egi
def main args := write (show args)
$ egison args.egi hello world 1
["hello", "world", "1"]
```

Egison でつくったスクリプトをコマンドにしたい場合はシェバン（shebang）を使えばよい．

```
$ cat args
#!/usr/local/egison

def main args := write (show args)
$ args hello world 1
["hello", "world", "1"]
```

7.4.2 do 式

do式を使うと，複数の IO 関数を一つにまとめ，順番に実行することができる．do式は Haskell の do記法に対応している．たとえば，以下のプログラムはユーザーの入力を一行読み取り，それを出力する．

```
def main arg := do
  let line := readLine ()
  write line
```

do式と再帰関数を組み合わせることができる．以下はインタプリタの REPL（Read-Eval-Print Loop）の Eval をぬいたプログラムである．

```
def main arg := repl

def repl := do
  write "> "
  flush ()
  let line := readLine ()
  write line
  flush ()
  repl
```

Haskell と同様に，オフサイドルールを使わない do記法の構文もサポートしている．

```
do { print "foo" ; print "bar" ; print "baz" }
```

Egison は静的型システムを持つ言語ではないため，do式は多相的ではない．Haskell の do式は，リストモナドや Maybe モナド，State モナド，IO モナドなどさまざまなモナドに対して使うことができるが，Egison の do式は IO モナドに対してしか使うことができない．

7.4.3 io 関数

io関数は Haskell の unsafePerformanceIO関数に対応する組み込み関数である．io関数を使うと IO 関数でない通常の関数のなかで IO 関数を呼び出せるようになる．これは printf デバッグをしたりするのに便利である．

以下のプログラムは二つのサイコロをふりそれらの目の合計値を返す．

```
def dice := io (rand 1 6)

dice + dice
```

7.5 seq 式

seq式は，プログラムを部分的に遅延評価ではなく正格評価したいときに使う．遅延評価は便利であるが，ときに，メモリを無駄に確保し効率が悪いことがある．部分的にプログラムを正格評価することで，プログラムの効率が向上することがある．seq式は第一引数のプログラムを正格評価したあと，第二引数のプログラムを評価する．たとえば，foldlは以下のように seq式を使うと効率が向上することが知られている．

```
def foldl $fn $init $ls :=
  match ls as list something with
    | [] -> init
    | $x :: $xs ->
      let z := fn init x
        in seq z (foldl fn z xs)
```

関数名	関数の内容
`return` x	x を IO 関数の結果として返す．
`openInputFile` $path$	パス $path$ のファイルを開き，そのファイルへの入力ポートを返す．
`openOutputFile` $path$	パス $path$ のファイルを開き，そのファイルへの出力ポートを返す．
`closeInputPort` $port$	入力ポート $port$ を閉じる．
`closeOutputPort` $port$	出力ポート $port$ を閉じる．
`readChar`	標準入力から一文字読み取り，その文字を返す．
`readLine`	標準入力から一行読み取り，その文字列を返す．
`writeChar` c	標準出力に文字 c を書き込む．
`write` s	標準出力に文字列 s を書き込む．
`readCharFromPort` $port$	入力ポート $port$ から一文字読み取り，その文字を返す．
`readLineFromPort` $port$	入力ポート $port$ から一行読み取り，文字列として返す．
`writeCharToPort` c $port$	文字 c を出力ポート $port$ に書き込む．
`writeToPort` s $port$	文字列 s を出力ポート $port$ に書き込む．
`isEof`	標準入力の一文字目が EOF であったら `True`，それ以外の場合は `False` を返す．
`isEofPort` $port$	入力ポート $port$ の一文字目が EOF であったら `True`，それ以外の場合は `False` を返す．
`flush`	標準出力に書き込み命令はされているもののバッファリングされている内容を即座に標準出力に出力する．
`flushPort` $port$	出力ポート $port$ に対して書き込み命令はされているもののバッファリングされている内容を即座に $port$ に出力する．
`readFile` $path$	パス $path$ のファイルの内容を文字列として返す．
`rand` n_1 n_2	整数 n_1 から n_2 までの整数のどれかをランダムで返す．
`f.rand` f_1 f_2	浮動小数点数 n_1 と n_2 の間の浮動小数点数をランダムで返す．
`newIORef`	変更可能な変数への参照を生成して返す．
`writeIORef` ref x	参照 ref が参照する変更可能な変数の値を x にする．
`readIORef` ref	参照 ref が参照する変更可能な変数の値を返す．
`readProcess` $path$ $args$ s	パス $path$ の実行ファイルに文字列のリスト $args$ を引数，標準入力に文字列 s を与えて実行し，その標準出力を文字列として返す．

表 7.3: IO 関数一覧

第 8 章

Sweet Egison - Egison パターンマッチの Haskell ライブラリ

Egison パターンマッチの実装には，本書でここまで紹介してきた Egison の言語実装のほかに，既存言語上の Domain Specific Language（DSL）として実装したライブラリ実装もある．既存言語のライブラリ実装には，deep embedding（インタプリタを埋め込むにより DSL を実現する手法）による実装である miniEgison 系と，shallow embedding（コンパイラを埋め込むことにより DSL を実現する手法）による実装である Sweet Egison 系の二種類がある．本章は，これらの実装のなかでもっとも高速な Sweet Egison の Haskell 実装の使い方を紹介する．Egison と異なり Haskell は静的型システムをもつ．そのため，Sweet Egison を使ったパターンマッチはコンパイル時に型検査がなされる．

8.1 Sweet Egison の使い方

Sweet Egison は Hackage を通して Haskell ライブラリとして公開されている．本節は Sweet Egison をすでにインストールしているものとして使い方を解説する．

Sweet Egison には Egison と同様に `matchAll`式がある．`matchAll`は引数として探索戦略とターゲットマッチャー，マッチ節のリストをとる．Sweet Egison では`matchAll`は Haskell の関数として実装されており，`as`や `with`のようなキーワードは挟まない．

```
matchAll dfs [1, 2, 3] (List Something)
  [[mc| $x : $xs -> (x, xs) |]]
-- [(1, [2, 3])]
```

上記の `matchAll`式は探索戦略として深さ優先探索 `dfs`が指定されている．Sweet Egison ではユーザが新しい評価戦略を定義できる．これは Egison にはない Sweet Egison の機能である．探索戦略の定義の方法は 8.6 節で解説する．上記の `matchAll`式のマッチャーは `(List Something)`である．8.5 節で改めて解説するように，Sweet Egison ではマッチャーは Haskell のデータとして表現さ

れる．マッチ節は mcクアジ・クォーター（quasi-quoter）を使って表現される．クアジ・クォーターは Template Haskell が提供する機能のひとつで，[mc|と|]で囲まれた部分を文字列として受け取り，Haskell プログラムを返す関数を定義することで，ユーザは Haskell メタプログラミングすることができる．マッチ節がどのような Haskell プログラムに展開されるのかは，8.3 節で紹介する．コンス・パターンは Haskell と同様にコロンひとつで表現する．

ライブラリに最初から定義されている探索戦略には，深さ優先探索 dfsと幅優先探索 bfsがある．深さ優先探索 dfsは，無限にあるパターンマッチの結果をすべて列挙できないことがあるが，高速である．

```
take 10 (matchAll dfs [1..] (Set Something)
  [[mc| $x : $y : _ -> (x, y) |]])
-- [(1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (1, 6), (1, 7), (1, 8), (1, 9), (1, 10)]
```

幅優先探索 bfsは，無限にあるパターンマッチの結果をすべて列挙できるが，深さ優先探索 dfsにくらべて実行効率がよくない．

```
take 10 (matchAll bfs [1..] (Set Something)
  [[mc| $x : $y : _ -> (x, y) |]])
-- [(1, 1), (1, 2), (2, 1), (1, 3), (2, 2), (3, 1), (1, 4), (2, 3), (3, 2), (4, 1)]
```

matchAll式がマッチ節のリストを引数にとるのは，複数のマッチ節を扱うためである．Egison と同様に “matchAll t m $[c_1,c_2,...]$” というプログラムは，“matchAll t m $[c_1]$ ++ matchAll t m $[c_2]$ ++ ...” と同値である．

```
matchAll [1, 2, 3] (List Something)
  [[mc| $x : $xs -> (x, xs) |]
  ,[mc| _ : $x : $xs -> (x, xs) |]]
-- [(1, [2, 3]), (2, [3])]
```

Sweet Egison では，Egison と同様に#を使って値パターンを表現する．値パターンを使って非線形パターンを表現できる．

```
matchAll [1, 5, 2, 4] (Multiset Eql)
  [[mc| $x : #(x + 1) : _ -> (x, x + 1) |]]
-- [(1, 2), (4, 5)]
```

最初のパターンマッチの結果だけを返す matchも Sweet Egison には実装されている．matchAll を使って matchは以下のように定義される．

```
match s tgt m cs = head (matchAll s tgt m cs)
```

matchを使ってポーカーの役判定は図 8.1 のように記述できる．Sweet Egison では，ターゲットのデータ型とマッチャーの両方でデータコンストラクタを使うために，データコンストラクタの名前がかぶりやすい．そのため，マッチャーのほうに CardMのように末尾にマッチャーの Mをつけて，名前の衝突を回避することがある．

```
data Suit = Spade | Heart | Club | Diamond deriving (Eq)
data Card = Card Suit Integer

poker :: [Card] -> String
poker cs =
  matchDFS cs (Multiset CardM)
    [[mc| [card $s $n, card #s #(n-1), card #s #(n-2), card #s #(n-3), card #s #(n-4)] ->
      "Straight flush" |],
     [mc| [card _ $n, card _ #n, card _ #n, card _ #n, _] -> "Four of a kind" |],
     [mc| [card _ $m, card _ #m, card _ #m, card _ $n, card _ #n] -> "Full house" |],
     [mc| [card $s _, card #s _, card #s _, card #s _, card #s _] -> "Flush" |],
     [mc| [card _ $n, card _ #(n-1), card _ #(n-2), card _ #(n-3), card _ #(n-4)] -> "
    Straight" |],
     [mc| [card _ $n, card _ #n, card _ #n, _, _] -> "Three of a kind" |],
     [mc| [card _ $m, card _ #m, card _ $n, card _ #n, _] -> "Two pair" |],
     [mc| [card _ $n, card _ #n, _, _, _] -> "One pair" |],
     [mc| _ -> "Nothing" |]]
```

図 8.1: ポーカーの役判定をする Haskell プログラム

8.2 Sweet Egison の仕組み

Sweet Egison の実装のアイデアを説明するために，まず以下の matchAll式がどのような Haskell プログラムに変換されるのかみる．cons $x _は$x : _と同値のパターンである．Sweet Egison では中置演算子は組み込みで:と ++のみが実装されている．説明をわかりやすくするために，本節では前置記法でコンス・パターンを記述する．

```
matchAll dfs [1, 2, 3] (Multiset Something) [[mc| cons $x _ -> x |]]
```

上記のプログラムは以下のようなリスト・モナドについての do式に展開される．

```
(\ (matcher, target) -> do
   (x, _) <- cons matcher target
   let (_, _) = consM matcher target
   return x)
 (Multiset Something, [1, 2, 3]) -- [1, 2, 3]
```

展開後のプログラムにあらわれる consと consMはそれぞれネクスト・ターゲット，ネクスト・マッチャーを計算するための関数である．consと consMは Haskell で定義された関数である．consと consMは，マッチャーとターゲットを引数に取る．マッチャーを引数にとる理由はパターンの多相性を実現するためである．

```
cons (Multiset Something) [1, 2, 3] -- [(1, [2, 3]), (2, [1, 3]), (3, [1, 2])]
consM (Multiset Something) [1, 2, 3] -- (Something, Multiset Something)
```

```
matchAll dfs [1, 2, 3] (Multiset Something) [[mc| cons $x (cons $y _) -> (x, y) |]]
-- [(1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3,2)]
```

```
(\ (matcher, target) -> do
   (x, target') <- cons matcher target
   let (_, matcher') = consM matcher target
   (y, _) <- cons matcher' target'
   let (_, _) = consM matcher' target'
   return (x, y))
 (Multiset Something, [1, 2, 3])
```

値パターンを含むパターンの展開もみる．

```
matchAll dfs [1, 2, 3] (Multiset Eql)
  [[mc| cons $x (cons #(x * 2) _) -> (x, y) |]]
-- [(1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3,2)]
```

valuePatは Haskell で定義された関数であり，さきほど登場した cons関数と同様に Haskell で定義された関数である．

```
(\ (matcher, target) -> do
   (x, target') <- cons matcher target
   let (_, matcher') = consM matcher target
   (target'', _) <- cons matcher' target'
   let (matcher'', _) = consM matcher' target'
   valuePat (x * 2) matcher'' target''
   return (x, y))
  (Multiset Something, [1, 2, 3])
```

8.3 Sweet Egison の最適化

実際の Sweet Egison は節で解説した展開よりも，最適化のために，もうすこし複雑な展開をする．特にワイルドカード最適化のために特別な工夫がある．本節ではこれらの工夫を紹介する．

8.3.1 ワイルドカードのパターンマッチの最適化

6.5 節で述べたワイルドカード最適化を Sweet Egison でもおこなうことができる．ワイルドカード最適化とは，ワイルドカードを含むパターンについて，ワイルドカードにマッチするターゲットの計算を省くことにより，パターンマッチの処理を高速化することであった．本節は，6.5

節と同様に，多重集合に対するコンス・パターンの第二引数がワイルドカードであった場合の最適化の例を使って Sweet Egison でこの最適化を使う方法を説明する．

```
matchAll dfs [1, 2, 3] (Multiset Something) [[mc| $x : _ -> x |]]
-- [1, 2, 3]
```

上記の matchAll式はワイルドカード最適化のために以下のように展開される．

```
(\ (matcher, target) -> do
   (x, _) <- cons (GP, WC) matcher target
   let (_, _) = consM matcher target
   return x)
 (Multiset Something, [1, 2, 3])
```

2 行目の cons関数が第一引数にパターンに対するパターンを追加で引数にとっている．パターンに対するパターンは，ワイルドカードとそれ以外のパターンを区別する．以下，PPはパターンに対するパターン（Patterns for Patterns），WCはワイルドカード（Wildcard），GPは一般のパターン（General Patterns）の略である．

```
data PP a = WC | GP
```

cons関数は第一引数として追加でこのパターンに対するパターンを追加で引数にとる．以下のように consパターンの第二引数がワイルドカードである場合は，その第二引数に対するネクスト・ターゲットとして undefinedを返す．

```
cons (GP, WC) (Multiset Something) [1, 2, 3]
-- [(1, undefined), (2, undefined), (3, undefined)]
```

consパターンの第二引数がワイルドカード以外である場合は，ネクスト・ターゲットとしてコレクションを計算して返す．

```
cons (GP, GP) (Multiset Something) [1, 2, 3]
-- [(1, [2, 3]), (2, [1, 3]), (3, [1, 2])]
```

8.3.2 パターン・フュージョン

6.4 節で解説した最適化であるパターン・フュージョンは，パターンのパターンマッチがでいない Sweet Egison ではユーザーが追加することができない．リストに対するパターンマッチで頻出するジョイン・コンス・パターンについてのみ組み込みでパターン・フュージョンが定義されている．Sweet Egison はジョイン・コンス・パターン p_1 ++ p_2 : p_3 を joinCons p_1 p_2 p_3 に変換する．

```
matchAll dfs [1, 2, 3] (List Something) [[mc| _ ++ $x : _ -> x |]]
-- [1, 2, 3]
```

たとえば，上記の matchAll式は以下のように展開される．

```
(\ (matcher, target) -> do
  (_, x, _) <- joinCons (WC, GP, WC) matcher target
  let (_, _, _) = joinConsM matcher target
  (List Something, [1, 2, 3])
```

8.4 マッチャーとパターンの型

マッチャーの型はクラスメソッドのない型クラスを使って表現される．

```
class Matcher m tgt
```

この型クラスを使って，たとえば Somethingマッチャーは下記のように定義される．データ Somethingをもつ型 Somethingを定義し，型 Somethingのデータは型 aのマッチャーであるという関係をインスタンス宣言で表現している．本来は，データ Somethingに Matcher aのような型を直接つけることが理想であるが，これは Haskell の型システムの制約のためできない．

```
data Something = Something

instance Matcher Something a
```

パターンは，8.3 節で述べたように，パターンに対するパターンと，マッチャー，ターゲットを引数にとり，ネクスト・ターゲットを返す関数として定義される．

```
type Pattern ps im it ot = ps -> im -> it -> [ot]
```

パターンの多相性を実現するためには，パターンを型クラスに属する関数として定義する．たとえば，コレクションに対するパターンは以下のような型クラスを使って定義される．

```
class CollectionPattern m t where
  type ElemM m
  type ElemT t
  nil :: Pattern () m t ()
  nilM :: m -> t -> ()
  cons :: Pattern (PP (ElemT t), PP t) m t (ElemT t, t)
  consM :: m -> t -> (ElemM m, m)
  join :: Pattern (PP t, PP t) m t (t, t)
  joinM :: m -> t -> (m, m)
  joinCons :: Pattern (PP t, PP (ElemT t), PP t) m t (t, ElemT t, t)
  joinConsM :: m -> t -> (m, ElemM m, m)
```

8.5 Sweet Egison のマッチャー定義

多重集合のマッチャーは以下のように定義される．

```
newtype Multiset m = Multiset m

instance Matcher m t => Matcher (Multiset m) [t]

instance Matcher m t => CollectionPattern (Multiset m) [t] where
  type ElemM (Multiset m) = m
  type ElemT [t] = t
  cons (_, WC) (Multiset _) xs = map (\x -> (x, undefined)) xs
  cons _       (Multiset _) xs = matchAll dfs xs (List Something)
    [[mc| $hs ++ $x : $ts -> (x, hs ++ ts) |]]
  consM (Multiset m) _ = (m, Multiset m)
```

8.6 ユーザによる探索戦略の追加

バックトラッキングによる探索アルゴリズムを記述することに使えるモナドをバックトラッキング・モナドという．代表的なバックトラッキング・モナドとしてリスト・モナドが知られている．リスト・モナドは以下のようにバックトラッキングを記述することに使える．リスト・モナドを使って 1 から 3 の整数のペアを列挙している．

```
do ns <- [1..3]
   x <- ns
   y <- ns
   return (x, y)
-- [(1,1),(1,2),(1,3),(2,1),(2,2),(2,3),(3,1),(3,2),(3,3)]
```

リスト・モナドは探索木を深さ優先探索の順番で探索する．そのため，探索木が無限に大きい場合，すべての結果を列挙できないことがある．

```
do ns <- [1..]
   x <- ns
   y <- ns
   return (x, y)
-- [(1,1),(1,2),(1,3),(1,4),(1,5),...]
```

ここで，深さ優先探索に対するリスト・モナドのように，深さ優先探索以外の探索戦略についても対応するモナドが存在するのかという疑問が生じる．Spivey によって幅優先探索について，リストのリストをベースのデータ構造にするモナドが対応することが発見されている [9]．このモナドは n 要素目のリストに，探索木の深さ n の場所にあるノードを保持する．

バックトラッキング・モナドは，`fromList`と `toList`という関数をもつ．これらはモナドのベースとなるデータ構造とリストとの間の変換関数である．深さ優先探索のモナドの場合は，ベースとなるデータ構造がリストであるため，どちらも `id`関数として実装できる．幅優先探索のモナド

の場合は，ベースとなるデータ構造がリストのリストであるため，toList関数が concatになったりする．バックトラッキング・モナドの実装の詳細は，backtracking という名前の Haskell ライブラリとして Hackage で公開されているのでそちらを参照してほしい [3].

バックトラッキング・モナドを使うと探索戦略を多相的に切り替えることができる．以下 dfsと bfsはそれぞれ深さ優先探索と幅優先探索のモナドのベースとなるデータ構造を初期化するための関数である．dfsと bfsを置き換えるだけで，探索戦略を制御できる．

```
toList (do
  ns <- dfs [1..]
  x <- fromList ns
  y <- fromList ns
  return (x, y))
-- [(1,1),(1,2),(1,3),(1,4),...]
```

```
toList (do
  ns <- bfs [1..]
  x <- fromList ns
  y <- fromList ns
  return (x, y))
-- [(1,1),(1,2),(2,1),(1,3),...]
```

matchAllの第一引数に渡されるのはこの探索戦略ごとのモナドを初期化するための関数である．

```
matchAll strategy [1..] (Set Something) [[mc| $x : $y : _ -> (x, y) |]]
```

matchAllの変換結果に適切に，この初期化関数と fromListと toListを挿入すれば，探索戦略を制御することができる．

```
toList
  ((\ (matcher, target) -> do
     (x, target') <- fromList (cons matcher target)
     let (_, matcher') = consM matcher target
     (y, _) <- fromList (cons matcher' target')
     let (_, _) = consM matcher' target'
     return (x, y))
   (strategy (Set Something, [1..])))
```

深さ優先探索や幅優先探索のためのモナドは Haskell で定義されている．ユーザーが新しく別の探索戦略のためのモナドを追加することも可能である．この機能は Egison 本体にはない Sweet Egison ならではの機能である．

第 II 部

数式処理システムとしての Egison

第 9 章

数式処理システム入門

数式処理システムとは，シンボリックな計算をサポートするプログラミング言語のことをいう．数式処理システムは，未束縛の変数をシンボルとして扱う．数式処理システムを使うと，たとえば，$x + x \to 2x$ や $(x + y)^2 \to x^2 + 2xy + y^2$ のような計算ができる．Egison もこのようなシンボリックな計算をサポートしている．本章は数式処理システムを使うとどのような計算ができるのかみていく．

9.1 未定義変数 = シンボル

数式処理システムは，未束縛の変数をシンボルとして扱う．そのため，未定義の変数をプログラム中で使ってもエラーにならない．

```
x -- x
```

数式処理システムとしての機能をもつ Egison には，シンボル同士の足し算や掛け算が組み込みで定義されている．そのため，たとえば，$x + x \to 2x$ や $(x + y)^2 \to x^2 + 2xy + y^2$ のような計算ができる．

```
x + x -- 2x
(x + y)^2 -- x^2 + 2 * x * y + y^2
```

Egison は数式を自動で積和標準形に展開する．積和標準形とは，掛け算が内側に，足し算が外側になるような形の数式のことである．そのため，$(x+y)^2$ は $x^2+2xy+y^2$ のように展開される．

出力の数式の項の因子として現れるシンボルの順番は，ユーザーが入力した数式に現れる順番と同じになるように実装されている．たとえば，上記の $(x + y)^2$ の x と y を入れ替えて $(y + x)^2$ を計算すると 2 項目は $2yx$ となる．

```
(y + x)^2 -- y^2 + 2 * y * x + x^2
```

9.2　関数適用のシンボル化

sqrt xや exp xのようにシンボルを引数にとる可能性がある関数について，関数適用を止めて，その式をシンボリックな値としてそのまま返すようにライブラリで定義されている．

```
sqrt 4 -- 2
sqrt x -- sqrt x
exp 1 -- e
exp x -- exp x
```

sqrt 2のように評価を進められない関数適用についても，関数適用を止めて，その式をそのまま返すようにライブラリで定義されている．$\sqrt{8} \to 2\sqrt{2}$ のような簡約もライブラリで定義されている．

```
sqrt 2 -- sqrt 2
sqrt 8 -- 2 * sqrt 2
```

これらの定義については，10.3 節で紹介するシングルクオートを使って定義される．

9.3　数式の簡約

$\sqrt{x} \cdot \sqrt{x} = x$ や $sin^2\theta + cos^2\theta = 1$ のような数式を紙の上で扱うときにおこなう簡約が，Egison を含む多くの数式処理システムには実装されている．

```
sqrt x * sqrt x -- x
(sin θ)^2 + (cos θ)^2 -- 1
```

Egison では，このような簡約のための機能は，Sweet Egison(Egison のパターンマッチ機能を提供する Haskell ライブラリ) を使って Haskell により言語に組み込みで実装されている．

上記以外にも，複素数などについての簡約ルールも実装されている．

```
i^2 -- -1
w^3 -- 1
w + w^2 -- -1
```

簡約ルールは https://github.com/egison/egison/blob/master/hs-src/Language/Egison/Math/Normalize.hs で定義されている．

数式の簡約の自動化は非常に難しく，Egison では簡潔なアルゴリズムしか用意されていない．たとえば，多項式の約分にはグレブナー基底についての理論に基づいたアルゴリズムが有用であるが，Egison には実装されていない．そのため，Egison は複雑な多項式の分数の適切に簡約できないことがある．

Wolfram 言語 (Mathematica) は，このような高度な数式の簡約ができる．“-M mathematica” オプションを追加して Egison 処理系を実行すると Wolfram 言語で読み込めるかたちで数式を出

力する．この出力を Wolfram 言語の処理系に入力すれば，Wolfram 言語の高度な簡約機能を利用できる．

```
$ egison -M mathematica
> x + sqrt 2
#mathematica|x + Sqrt[2]|#
```

9.4 二次方程式を解くプログラム - 数式を処理するアルゴリズムの記述

本節では，二次方程式を解くアルゴリズムをプログラムとして書いていく．以下のように二次方程式を引数にとり，解を返す関数 qFを定義していく．

```
qF (x ^ 2 + x + 1) x
-- ((-1 + i * sqrt 3) / 2, (-1 - i * sqrt 3) / 2)

qF (x ^ 2 + b * x + c) x
-- ((- b + sqrt (b^2 - 4 * c)) / 2, (- b - sqrt (b^2 - 4 * c)) / 2)

qF (a * x ^ 2 + b * x + c) x
-- ((- b + sqrt (b^2 - 4 * a * c)) / (2 * a), (- b - sqrt (b^2 - 4 * a * c)) / (2 * a))

qF (a * x ^ 2 + 2 * b * x + c) x
-- ((- b + sqrt (b^2 - a * c)) / a, (- b - sqrt (b^2 - a * c)) / a)
```

qFは解の公式 $\frac{-b\pm\sqrt{b^2-4ac}}{2a}$ をそのまま書くだけでも定義することができる．以下のプログラムは qF'という補助関数を使って qFを定義している．qF'は第一引数に二次の係数，第二引数に一次の係数，第三引数に定数項をとる．たとえば，qF' 1 0 2は 2 次方程式 $x^2+2=0$ の解を返す．coefficientsは多項式とシンボルを引数にとり，そのシンボルについて係数のリストを返す関数である．qF'は単純に $\frac{-b+\sqrt{b^2-4ac}}{2a}$ と $\frac{-b-\sqrt{b^2-4ac}}{2a}$ からなるタプルを返す関数として定義されている．

```
def qF f x :=
  match coefficients f x as list mathExpr with
    | [$a_0, $a_1, $a_2] -> qF' a_2 a_1 a_0

def qF' a b c :=
  ( ((- b) + sqrt (b ^ 2 - 4 * a * c)) / 2 * a
  , ((- b) - sqrt (b ^ 2 - 4 * a * c)) / 2 * a )
```

しかし，上記の記述ではアルゴリズムっぽさがないため，今回は平方完成によりこの解の公式を求めるアルゴリズムを書いていく．平方完成とは，x^2+bx+c のような二次式を $(x+b/2)^2-b^2/4+c$ のように変形する操作のことである．

```
1 def qF' a b c :=
2   match (a, b, c) as (mathExpr, mathExpr, mathExpr) with
3     | (#1, #0, _) -> (sqrt (- c), - sqrt (- c))
4     | (#1, _, _) ->
5       (2)#((- (b / 2)) + %1, (- (b / 2)) + %2)
6         (withSymbols [x, y]
7           qF (substitute [(x, y - b / 2)] (x ^ 2 + b * x + c)) y)
8     | (_, _, _) -> qF' 1 (b / a) (c / a)
```

平方完成を使ってを二次方程式を解くアルゴリズムを記述すると qF'は 3 つのマッチ節からなる関数となる．3 行目のマッチ節は，$x^2 + c = 0$ のような形の二次方程式を処理する．このマッチ節は，$\sqrt{c}$ と $-\sqrt{c}$ という結果を返す．4-7 行目のマッチ節は，$x^2 + bx + c = 0$ のような二次の係数が 1 である形の二次方程式を処理する．このマッチ節が平方完成の操作をするこの関数の本体である．substituteは Egison のライブラリ関数である．substituteは第一引数で指定された代入を第二引数の式に対しておこなう．ここでは第一引数で，(x, y - b / 2)という代入が指定されているため，$x = y - b/2$ という代入を式 $x^2 + bx + c$ に対しておこなう．同時に複数の代入を指定するために，substituteの第一引数はリストを引数に取るようになっている．$x = y - b/2$ という代入をするとこの方程式は 3 行目のマッチ節でマッチする y についての二次方程式に変形される．そのため，qF関数にこの変形された方程式を渡せば，y についてこの二次方程式の解を得られる．この y についての解に $\frac{-b}{2}$ を加えることにより，もとの x についての解を得ることができる．withSymbols式は，局所シンボルを生成する構文である．第一引数のリストで指定されたシンボル（上記の場合，xと y）を第二引数の式の中でシンボルとして使うことができる．withSymbols式のこの機能のおかげで，もしプログラムの別の箇所で，xや yにたとえば具体的な整数が束縛されていたとしても，withSymbols式の内部では，その影響を考えなくてよくなる．8 行目のマッチ節は，二次の係数が 1 でない二次方程式を処理する．二次の係数で方程式を割ることにより，二次の係数が 1 の方程式に変換し，ふたたび qF関数にわたす処理をしている．

二次方程式より一段と複雑になるが，三次方程式・四次方程式を解くプログラムを書いてみると面白い．これらのプログラムは Egison のソースコードの sample/math/algebra ディレクトリ以下に公開されている．

9.5 微分するプログラム - 数式に対するパターンマッチ

Egison には微分計算をするための関数 d/dがライブラリ関数として実装されている．この関数は第一引数の関数を第二引数のシンボルについて微分した結果を返す．関数の名前を d/dのようにすることで数式に近い形で微分を表現できている．Egison は/を変数名に使うことを許している．

```
d/d x x -- 1
d/d (x^2) x -- 2 * x
d/d (exp x) x -- exp x
```

```
d/d (log x) x -- 1 / x
d/d (x * log x) x -- log x + 1
d/d (1 / log x) x -- -1 / (x * (log x)^2)
```

本節はこの関数の実装を紹介する.

d/dは以下のように定義される.

```
1  def d/d f x :=
2    match f as mathExpr with
3      -- シンボルの微分
4      | #x -> 1
5      | ?isSymbol -> 0
6      -- 関数適用の微分
7      | #'exp $g -> exp g * d/d g x
8      | #'log $g -> 1 / g * d/d g x
9      | #'sqrt $g -> 1 / (2 * sqrt g) * d/d g x
10     | #'(^) $g $h -> f * d/d (log g * h) x
11     | #'cos $g -> (- sin g) * d/d g x
12     | #'sin $g -> cos g * d/d g x
13     -- 定数の微分
14     | #0 -> 0
15     | _ * #1 -> 0
16     -- 項の微分
17     | #1 * $fx ^ $n -> n * fx ^ (n - 1) * d/d fx x
18     | $a * $fx ^ $n * $r -> a * d/d (fx ^' n) x * r + a * fx ^' n * d/d r x
19     -- 多項式の微分
20     | poly $ts -> sum (map 1#(d/d %1 x) ts)
21     -- 商の微分
22     | $p1 / $p2 ->
23       let p1' := d/d p1 x
24           p2' := d/d p2 x
25        in (p1' * p2 - p2' * p1) / p2 ^ 2
```

第一引数の f を数式としてパターンマッチしている．4-5 行目はシンボルの微分を定義している．もし，fが xそのものであったら，1 を返し，fが x以外のシンボルであったら 0 を返すように定義されている．7-12 行目はいくつかの基本的な関数の適用について合成関数の微分 $\frac{df(g(x))}{dx} = \frac{df(g(x))}{dg(x)}\frac{dg(x)}{x}$ を記述している．たとえば，7 行目は指数関数の微分をしている．このマッチ節の本体は $\frac{de^{g(x)}}{dx} = e^{g(x)}\frac{dg(x)}{dx}$ を表現している．また，8 行目は対数関数の微分をしている．このマッチ節の本体は $\frac{dlog(g(x))}{dx} = \frac{1}{g(x)}\frac{dg(x)}{dx}$ を表現している．14-15 行目は定数項に対する微分を定義している．定数項は微分すると 0 になる．17-18 行目は，ここまでに微分を定義した項を掛け合わせた項についての微分を定義している．x^n は nx^{n-1} に微分されることと，fg は $f'g + fg'$ に微分されることを利用して定義している．ここでも合成関数の微分の公式が使われている．20 行目は多項式の微分が定義されている．ここまでで単項式についての微分が定義されて

いるので，それらを足し合わせたものについては，それぞれの項について微分して足し合わせてやればよい．22-24 行目は多項式を分母と分子にもつ式に対する微分が定義されている．商の微分の公式を使って定義されている．

9.6　ベクトルや行列の計算 - 添字記法

Egison の数式処理システムとしての特徴に，ベクトルや行列，そしてそれらを一般化したテンソルを扱う計算を簡潔に記述できるというものがある．これは二十世紀初頭に微分幾何や相対性理論の研究の過程で発明されたテンソルの添字記法をプログラミング言語の機能として Egison がサポートしているためである．

添字記法は，テンソルに添字を付加することによって，さまざまな種類のテンソルの掛け算を表現する．たとえば，ベクトル同士の掛け算には，テンソル積・アダマール積・内積の三種類があるが，これらは以下のように添字を使い分けることによって表現される．

```
[| x1, x2 |]_i .[| y1, y2 |]_j -- [| [| x1 * y1, x1 * y2 |], [| x2 * y1, x2 * y2 |] |]
    _i_j
[| x1, x2 |]_i .[| y1, y2 |]_i -- [| x1 * y1, x2 * y2 |]_i
[| x1, x2 |]~i .[| y1, y2 |]_i -- x1 * y1 + x2 * y2
```

上記の例のように，Egison ではベクトルは成分を `[| |]`で囲むことにより表現する．また行列は，ベクトルのベクトルとして表現する．関数型プログラミングの視点で添字記法は，テンソルの掛け算関数のパラメータとして，テンソルだけでなく添字も追加で渡すことによって一つの関数に複数の役割を持たしていると説明できる．

添字記法を言語機能としてもつプログラミング言語はほかにも存在する．そのなかで Egison の特徴は，関数型プログラミングとうまく調和するかたちで添字記法を組み込んでいるところである．そのおかげで，高階関数と添字記法を組み合わせて使うことができる．たとえば，$g_{i_1 j_1} g_{i_2 j_2} \ldots g_{i_n j_n}$ のような数式を `foldl`関数を使って以下のように記述することができる．

```
foldl . 1 (map (\x -> g_[i_x]_[j_x]) [1..n])
```

添字記法は Egison の重要な言語機能であるため，第 11 章と第 13 章で詳しく紹介する．

9.7　数式の出力形式

“`-M`” オプションで出力形式を指定することができる．“`egison -M latex`” を実行すると，LaTeX 形式で結果を出力する．

```
$ egison -M latex
Egison Version 3.7.14 (C) 2011-2018 Satoshi Egi
https://www.egison.org
Welcome to Egison Interpreter!
```

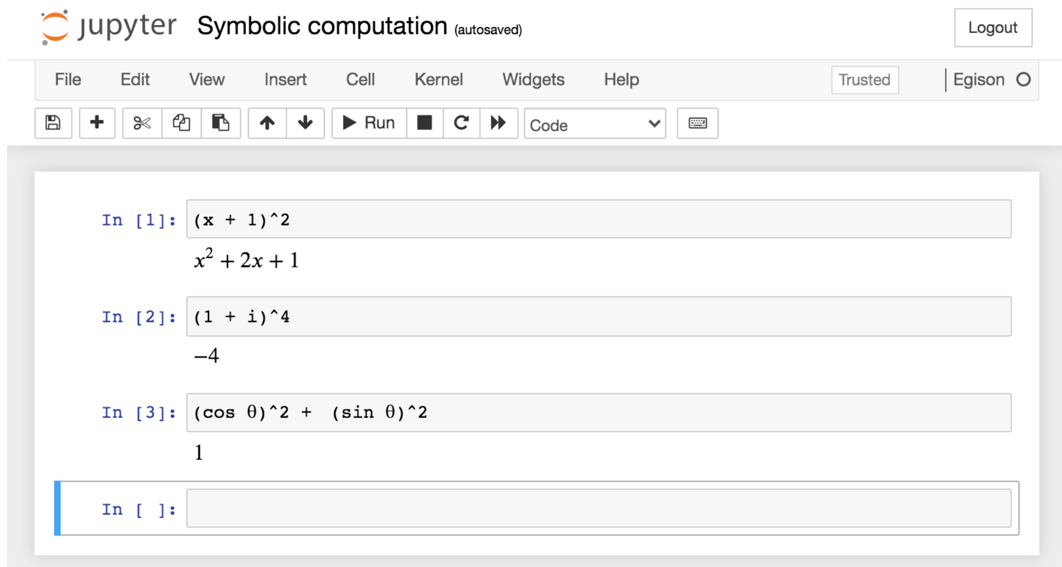

図 9.1: Jupyter Notebook 上で Egison を実行する様子

```
> (+ (sqrt a) b)
#latex|\sqrt{a} + b|#
```

“egison -M mathematica” を実行すると，Mathematica 形式で結果を出力する．

```
$ egison -M mathematica
Egison Version 3.7.14 (C) 2011-2018 Satoshi Egi
https://www.egison.org
Welcome to Egison Interpreter!
> (+ (sqrt a) b)
#mathematica|Sqrt[a] + b|#
```

LaTeX 出力は Jupyter Notebook とも連携している．Jupyter Notebook の Egison プラグインを利用すれば，図 13.1，図 13.4 のようにインタラクティブに計算結果が数式として表示される．

第 10 章

数式データの扱い

本章は，シンボリックな計算を扱うための Egison の機能を紹介する．

10.1　数式データの内部表現

数式は組み込みデータ型として Egison 処理系内部で Haskell の代数的データ型を用いて直接実装されている．fromMathExprという組み込み関数を使うと，この数式データを Egison 上の代数的データ型に変換することができる．

```
fromMathExpr  (x - sqrt 2)
-- Div (Plus [Term 1 [(Symbol x [], 1)], Term -1 [(Apply sqrt [Div (Plus [Term 2 []])
--     (Plus [Term 1 []])], 1)]]) (Plus [Term 1 []])
```

本節は，数式データを表現するこの内部の代数的データ型について説明する．

上記の出力の代数的データ型からもわかるように，数式データ型は，分母と分子の両方に積和標準形の多項式をとる分数として定義されている．Divがこの分数を生成するためのデータコンストラクタで，Plusが多項式を生成するためのデータコンストラクタとなっている．そのため，たとえば分数同士の足し算の結果は一つの分数にまとめられる．

```
a / b + x / y
-- (a * y + x * b) / (b * y)
```

Plusの引数がリストであることからもわかるように，多項式は，項のリストからなるデータ型として定義されている．そして，項は係数と因子のリストからなるデータ型として定義されている．項は Termデータコンストラクタを使って生成される．Termは第一引数に係数，第二引数に因子のリストをとる．係数には整数，因子にはシンボルなどをとることができる．たとえば，3 * x^2 * sqrt 2という項は，係数として 3，因子のリストとして [x^2, sqrt 2]をとる．因子にはシンボル以外に，$\sqrt{2}$ のような項もとることもできる．因子のリストは，底（x^2 の場合 x）と冪指数（x^2 の場合 2）の連想リストとなっている．冪指数には自然数しかとれないという制限がある．

```
fromMathExpr (3 * x^2 * sqrt 2)
```

```
-- Div (Plus [Term 3 [(Symbol x [], 2), (Apply sqrt [Div (Plus [Term 2 []]) (Plus [Term 1
     []])], 1)]]) (Plus [Term 1 []])
```

上記の例のように，因子は Symbolや Applyといったデータコンストラクタを使って生成される．Symbolデータコンストラクタは，第一引数にシンボル自身を，第二引数に添字のリストをとる．Symbolデータコンストラクタの第一引数にシンボル自身にしているのは，fromMathExprの結果を扱いやすくするためである．シンボルを表現するのデータ構造のなかにシンボル自身があると定義が無限に循環してしまうため，Haskell によるシンボルの内部表現は Symbolデータコンストラクタの第一引数は文字列をとっている．添字については，テンソルについて解説する第 11 章で説明する．Applyデータコンストラクタは，第一引数に関数名のシンボルを，第二引数に引数の数式のリストをとる．その他に因子のデータコンストラクタに Quoteと Functionがある．Quoteデータコンストラクタについては，10.2 節で紹介する．Functionデータコンストラクタについては，第 12 章で紹介する．

fromMathExprの逆に Egison 上の代数的データ型として表現された数式を Egison 処理系内部の数式データに変換する toMathExpr関数も用意されている．toMathExpr関数を使うと，Egison 上の代数的データ型として数式を処理したあとに，これらのデータを Egison 内部の数式データに戻すことができる．

```
toMathExpr (fromMathExpr  (x - sqrt 2))
-- x - sqrt 2
```

ただし，fromMathExprと toMathExprのような組み込み関数は，Egison ライブラリで定義されている mathExprマッチャーを使えば，ユーザーが触る必要はほぼない．mathExprマッチャーの実装に fromMathExprと toMathExprは使われている．

数は複雑なデータ型である．自然数までであれば代数的データ型としてエレガントに定義できる．しかし，自然数の足し算の逆関数引き算を考えると，負の数という概念が得られる．さらに足し算を繰り返すことによって得られる掛け算の逆を考えると，有理数という概念が得られる．さらに掛け算を繰り返すことによって得られるべき乗という操作の逆を考えると，平方根や虚数，それらを含む代数的数の概念が得られる．

10.2 バッククオート（‘）による式展開の制御

Egison は数式を自動で積和標準形に展開すると 9.1 で述べたが，この展開をバッククオート（`）により制御することができる．バッククオートに続く式は，ひとつのシンボルのように扱われる．

```
`(x + 1)^2
--  `(x + 1)^2

`(x + 1) + `(x + 1)
--  2 * `(x + 1)
```

```
`(x + 1) + (x + 1)
-- `(x + 1) + x + 1
```

バッククオートが役に立つのは，主に下記のような多項式の割り算を含む計算のときである．

```
(x + 1)^2 / (x + 1)
-- (x^2 + 2 * x + 1) / (x + 1)

`(x + 1)^2 / `(x + 1)
-- (x + 1)
```

バッククオートによりクオートされた項は，Egison 内部の数式データのなかで因子として扱われる．`Quote`は，`Symbol`や`Apply`と同列のデータコンストラクタとして定義されている．

```
fromMathExpr (`(x + 1)^2)
-- Div (Plus [Term 1 [(Quote (Div (Plus [Term 1 [(Symbol x [], 1)], Term 1 []]) (Plus [
    Term 1 []])), 2)]]) (Plus [Term 1 []])fromMathExpr `(x + 1)^2
```

10.3　シングルクオート（'）による関数適用の制御

Egison はシンボルを関数として適用するとシンボリックな結果を返す．例えば，未定義の変数`f`についての関数適用は以下のように動作する．

```
f a -- f a
```

定義済みの関数の適用についても上記のようにシンボリックな結果を返したいことがある．たとえば，`sqrt`関数については以下のように動作してほしい．

```
sqrt 4 -- 2
sqrt 5 -- sqrt 5
```

このような動作を実現するために，Egison には組み込み構文としてシングルクオート（`'`）が用意されている．シングルクオートには変数が続き，たとえその変数が定義済みであったとしても，その変数の名前のシンボルを返す．シングルクオートを関数の前に付加すれば，定義済みの関数に対してもシンボリックな関数適用を表現できる．

```
def f x := x + 1

f a -- a + 1
'f a -- f a
```

シングルクオートは平方根を計算する`sqrt`の実装以外にも，三角関数`sin`や`cos`の実装などで Egison ライブラリ内で使われている．

```
fromMathExpr x
-- Div (Plus [Term 1 [(Symbol x [], 1)]]) (Plus [Term 1 []])
fromMathExpr 'x
-- Div (Plus [Term 1 [(Symbol x [], 1)]]) (Plus [Term 1 []])
```

10.4 withSymbols 式によるローカルシンボルの宣言

未定義の変数はシンボルとして扱われるという仕様は，すでに定義済みの変数をユーザーがシンボルとして扱おうとするミスを誘発しやすい．この問題を解決するために，withSymbols式というローカルシンボルを宣言するための構文が Egison には用意されている．withSymbols式は，第一引数に変数のリストをとり，これらの変数は第二引数の式の評価時にシンボルとして扱われる．

```
def i := 10

i -- 10
withSymbols [i] i -- i
i + withSymbols [i] i -- 10 + i
```

withSymbols式で宣言されたローカルなシンボルは，未定義変数から生成されるグローバルなシンボルとは独立したシンボルとして処理される．

```
j + withSymbols [j] j -- j + j
```

10.5 数式データに対するパターンマッチ

本節は，数式データに対して用意されているパターンを紹介する．数式に対するマッチャーmathExprは Egison ライブラリに実装されている．mathExprの定義は lib/math/expression.egi で確認できる．また数式に対するパターンを，より数式に近いかたちで記述するためにいくつかの糖衣構文が組み込みで実装されている．本節では，これらの糖衣構文も紹介する．数式に対するパターンマッチは，9.5 節で実演されている．パターンの具体的な使用例として，9.5 節のプログラムを参照してほしい．

10.5.1 因子に対するパターンマッチ

因子に対するパターンコンストラクタは，symbol・apply・quote・funcの 4 つがある．symbolはシンボルに，applyは 9.2 節で紹介したシンボリックな関数適用に，quoteは 10.2 節で紹介したバッククオートされた式に，funcは第 12 章で解説する関数シンボルと呼ばれるオブジェクトにパターンマッチするためのパターンコンストラクタである．

symbolパターンコンストラクタの第一引数はシンボル自身に，第二引数は添字にパターンマッチする．添字については，テンソルについて解説する第 11 章で改めて紹介する．

```
match x as mathExpr with
  | symbol $s _ -> s
-- x
```

シンボルに対しては symbolパターンコンストラクタを使ってパターンマッチすることもできるが，実際には値パターンを使ってパターンマッチすることが多い．

```
match x as mathExpr with
  | #x -> "Matched"
  | _ -> "Not matched"
-- "Matched"
```

applyパターンコンストラクタの第一引数は関数名のシンボルに，第二引数は引数の数式のリストにパターンマッチする．引数のパターンマッチには，list mathExprマッチャーが使われるため，リストに対するパターンを使うことができる．

```
match ('f x 1 y)  as mathExpr with
  | apply $f ($x :: $xs) -> (f, x, xs)
-- (f, x, [1, y])
```

quoteパターンコンストラクタは引数に数式に対するパターンをとる．このパターンは mathExpr マッチャーを使ってクオートの中身の式とパターンマッチされる．

```
match `(x + 1) as mathExpr with
  | quote $e -> e
-- x + 1
```

10.5.2　項に対するパターンマッチ

項に対するパターンマッチには，termパターンコンストラクタを使う．termパターンコンストラクタの第一引数は係数に，第二引数は因子の連想リストにパターンマッチする．因子の連想リストのパターンマッチには，6.6 節で紹介した assocMultiset マッチャーが使われる．

```
matchAll 3 * x^2 * y  as mathExpr with
  | term $a ($x :: $xs) -> (a, x, xs)
-- [(3, x, [(x, 1), (y, 1)]), (3, y, [(x, 2)])]
```

termパターンコンストラクタには，より数式に近いパターンを記述するための糖衣構文がある．

```
matchAll 3 * x^2 * y  as mathExpr with
  | $a * $x * $xs -> (a, x, xs)
-- [(3, x, x * y), (3, y, x^2)]
```

```
matchAll 3 * x^2 * y  as mathExpr with
  | $a * $x^#2 * $xs -> (a, x, xs)
-- [(3, x, y)]
```

10.5.3　多項式に対するパターンマッチ

有理式に対するパターンマッチには，polyパターンコンストラクタを使う．polyパターンコンストラクタは引数に項のリストにマッチするパターンをとる．

```
matchAll x + y + 1  as mathExpr with
  | poly ($x :: $xs) -> (x, xs)
-- [(x, [y, 1]), (y, [x, 1]), (1, [x, y])]
```

polyパターンコンストラクタにも，より数式に近いパターンを記述するための糖衣構文がある．

```
matchAll x + y + 1  as mathExpr with
  | $x + $xs -> (x, xs)
-- [(x, y + 1), (y, x + 1), (1, x + y)]
```

10.5.4　有理式に対するパターンマッチ

有理式に対するパターンマッチには，divパターンコンストラクタを使う．divパターンコンストラクタの第一引数は分子の多項式に，第二引数は分母の多項式にパターンマッチする．

```
matchAll (a + b) / c  as mathExpr with
  | div $x $y -> (x, y)
-- [(a + b, c)]
```

divパターンコンストラクタにも，より数式に近いパターンを記述するための糖衣構文がある．

```
matchAll (a + b) / c  as mathExpr with
  | $x / $y -> (x, y)
-- [(a + b, c)]
```

第 11 章

テンソル計算

Egison にはテンソル計算を便利に記述するために開発された機能が実装されている．本章はこれらの機能を紹介していく．

11.1　テンソルとは

テンソルとは，ベクトルや行列を一般化したものである．ベクトルは一階のテンソル，行列は二階のテンソルである．ベクトルは一次元配列，行列は二次元配列としてプログラムでは表現されることが多い．n 階のテンソルは，n 次元配列により表現される．

以下はそれぞれベクトル，行列，2 階のテンソルの例を数式で表現したものである．

$$\begin{pmatrix} a_1 & a_2 & a_3 \end{pmatrix}$$

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix}$$

$$\begin{pmatrix} \begin{pmatrix} a_{111} & a_{112} & a_{113} \\ a_{121} & a_{122} & a_{123} \end{pmatrix} \begin{pmatrix} a_{211} & a_{212} & a_{213} \\ a_{221} & a_{222} & a_{223} \end{pmatrix} \begin{pmatrix} a_{311} & a_{312} & a_{313} \\ a_{321} & a_{322} & a_{323} \end{pmatrix} \end{pmatrix}$$

上記の数式は Egison では下記のように表現される．Egison ではテンソルの成分を `[| |]`で囲むことにより，テンソルを表現する．行列はベクトルのベクトルとして，2 階のテンソルは行列のベクトルとして表現される．これは C 言語などの多次元配列の表記法に似ている．

```
[| a_1, a_2, a_3 |]

[| [| a_11, a_12, a_13 |],
   [| a_21, a_22, a_23 |] |]

[| [| [| a_111, a_112, a_113 |],
      [| a_121, a_122, a_123 |] |],
   [| [| a_211, a_212, a_213 |],
      [| a_221, a_222, a_223 |] |],
```

```
[| [| a_311, a_312, a_313 |],
   [| a_321, a_322, a_323 |] |] |]
```

11.2 テンソルの添字記法

テンソル計算は，ベクトルについてのテンソル積・アダマール積・内積の組み合わせでほとんど表現できる．テンソル積は，ベクトルのすべての成分の組み合わせからなる行列を返す．アダマール積は，対応する成分同士のみからなるベクトルを返す．内積は，アダマール積の結果の成分を足し合わせた結果のスカラー値を返す．

$$\begin{pmatrix} a_1 & a_2 \end{pmatrix} \otimes \begin{pmatrix} b_1 & b_2 \end{pmatrix} = \begin{pmatrix} a_1 b_1 & a_1 b_2 \\ a_2 b_1 & a_2 b_2 \end{pmatrix}$$

$$\begin{pmatrix} a_1 & a_2 \end{pmatrix} \circ \begin{pmatrix} b_1 & b_2 \end{pmatrix} = \begin{pmatrix} a_1 b_1 & a_2 b_2 \end{pmatrix}$$

$$\begin{pmatrix} a_1 & a_2 \end{pmatrix} \cdot \begin{pmatrix} b_1 & b_2 \end{pmatrix} = a_1 b_1 + a_2 b_2$$

たとえば，行列 A と行列 B の掛け算は，A の行と B の列についてはテンソル積，A の列と B の行については内積をとる演算と解釈できる．

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{pmatrix}$$

ベクトルについては，掛け算の方法は，テンソル積・アダマール積・内積の 3 つだけだが，行列や行列よりさらに高階なテンソルとなると掛け算の方法が無数にある．これらの掛け算のすべてに名前をつけるのは大変である．そのため，数学者はシンボリックな添字をテンソルに付加し，その添字をパラメーターにしてこれらの掛け算を表現する．たとえば，それぞれのベクトルに異なるシンボルが付加されている場合はテンソル積に，それぞれのベクトルの同じ位置に同じシンボルが付加されている場合はアダマール積に，それぞれのベクトルの上下に同じシンボルが付加されている場合は内積になる．添字には上添字と下添字の二種類がある．この位置関係によってアダマール積となるか内積となるか区別する．このようにシンボリックな添字を使って，テンソルのかけあわせ方を指定する記法は，添字記法と呼ばれている．下記の数式は Egison で 9.6 節で示したプログラムで表現できる．

$$\begin{pmatrix} a_1 & a_2 \end{pmatrix}_i \begin{pmatrix} b_1 & b_2 \end{pmatrix}_j = \begin{pmatrix} a_1 b_1 & a_1 b_2 \\ a_2 b_1 & a_2 b_2 \end{pmatrix}_{ij}$$

$$\begin{pmatrix} a_1 & a_2 \end{pmatrix}_i \begin{pmatrix} b_1 & b_2 \end{pmatrix}_i = \begin{pmatrix} a_1 b_1 & a_2 b_2 \end{pmatrix}_i$$

$$\begin{pmatrix} a_1 & a_2 \end{pmatrix}^i \begin{pmatrix} b_1 & b_2 \end{pmatrix}_i = a_1 b_1 + a_2 b_2$$

行列のかけ算も添字記法をつかってうまく表現できる．内積をとる A の列と B の行について，それぞれ上下を逆にし，同じシンボルにすれば，行列の掛け算は表現できる．

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}^i{}_j \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}^j{}_k = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{pmatrix}^i{}_k$$

11.3 添字記法をプログラミングに導入するためのアイデア

添字記法のプログラミングへの導入が難しい理由は，演算子ごとにシンボリックな添字のルールが異なることにある．特にテンソルの足し算と掛け算では，添字のルールが異なる．例を紹介する．u と v をそれぞれベクトルとする．

- $u_i + v_i$ はベクトルを返す．対して，$u_i v_i$ は無効な式である．（$u_i v_i$ は上下添字を自由に入れ替えることが可能なユークリッド空間では内積を返す．しかし，このような例外はここでは考えない．）
- $u^i + v_i$ は無効である．対して，$u^i v_i$ は内積を返す．
- $u_i + v_j$ は無効である．対して，$u_i v_j$ はテンソル積を返す．

このように演算子ごとに異なるシンボリックな添字のルールがある．そのため，テンソルを扱う関数を定義するたびに，シンボリックな添字のルールを指定する必要がある．これはわずらわしい作業である．それだけではなく，添字ルールを記述するための新しい構文が必要になる場合もある．このような新しい構文はプログラミング言語を複雑にする．

Egison は，シンボリックな添字のルールを簡略化することによって，この問題を解決する．添字のルールの簡略化によって，有効な数式の解釈は変わらない．ただし，無効であった数式が有効になる場合がある．例えば，$v_i + v_j$ は無効な式であったが，新しい添字ルールでは有効な式になる．テンソルの足し算についての簡略化された添字ルールを図 11.1a で示す．添字ルールを緩めることによって，多くのテンソルについての演算をテンソルの足し算の変種として捉えることができるようになる．例えば，テンソルの掛け算は，足し算と同じ形で成分を掛け合わせたあとに縮約（contraction）という追加の処理をする演算と捉えられる（図 11.1b）．テンソルの縮約とは，同じシンボルの上添字と下添字があるときに，対角成分を足し合わせる操作のことをいう．このような見方をすると，テンソルについての関数を定義するときの添字ルールの記述をへらすことができる．

$$\begin{pmatrix} a \\ b \end{pmatrix}_i + \begin{pmatrix} c \\ d \end{pmatrix}_i = \begin{pmatrix} a+c \\ b+d \end{pmatrix}_i$$

$$\begin{pmatrix} a \\ b \end{pmatrix}_i + \begin{pmatrix} c \\ d \end{pmatrix}_j = \begin{pmatrix} a+c & a+d \\ b+c & b+d \end{pmatrix}_{ij}$$

$$\begin{pmatrix} a \\ b \end{pmatrix}_i + \begin{pmatrix} c \\ d \end{pmatrix}^i = \begin{pmatrix} a+c & a+d \\ b+c & b+d \end{pmatrix}_i^{\ i}$$

(a) 足し算の添字ルール

$$\begin{pmatrix} a \\ b \end{pmatrix}_i \begin{pmatrix} c \\ d \end{pmatrix}_i = contract \begin{pmatrix} ac \\ bd \end{pmatrix}_i = \begin{pmatrix} ac \\ bd \end{pmatrix}_i$$

$$\begin{pmatrix} a \\ b \end{pmatrix}_i \begin{pmatrix} c \\ d \end{pmatrix}_j = contract \begin{pmatrix} ac & ad \\ bc & bd \end{pmatrix}_{ij} = \begin{pmatrix} ac & ad \\ bc & bd \end{pmatrix}_{ij}$$

$$\begin{pmatrix} a \\ b \end{pmatrix}_i \begin{pmatrix} c \\ d \end{pmatrix}^i = contract \begin{pmatrix} ac & ad \\ bc & bd \end{pmatrix}_i^{\ i} = ac + bd$$

(b) 掛け算の添字ルール

図 11.1: 簡略化された添字ルール

11.4　スカラー仮引数とテンソル仮引数

テンソルの添字記法のプログラミングへの導入は，(i) スカラー仮引数とテンソル仮引数という 2 種類の仮引数の概念の導入と，(ii) 適切な添字の簡約ルールによってなされる．本節からこれらの概念を解説を始める．

まず，**スカラー仮引数**（*scalar parameter*）と**テンソル仮引数**（*tensor parameter*）という 2 種類の仮引数の概念を説明する．Egison にはスカラー仮引数とテンソル仮引数という二種類の仮引数があり，これらは仮引数の先頭に`$`と `%`を付加することで区別する．仮引数の名前の先頭に`$`が付いていればスカラー仮引数，`%`が付いていればテンソル仮引数となる．スカラー仮引数とテンソル仮引数は，引数としてテンソルが与えられたとき動作が異なる．スカラー仮引数にテンソルが引数として与えられると，テンソルの成分ごとに関数の処理がマップされ，スカラー仮引数にはテンソルの成分が渡される．たいして，テンソル仮引数にテンソルが引数として与えられると，通常の仮引数とどうようにテンソルがそのまま仮引数に渡される．ここまで登場した関数の仮引数には`$`も`%`も付加されていなかったが，どちらも付加されていない場合はテンソル仮引数として扱われる．

```
(\$u $v -> (u, v)) [| a, b |]_i [| x, y |]_j
-- [| [| (a, x), (a, y) |],
--    [| (b, x), (b, y) |] |]_i_j

(\%u %v -> (x, y)) [| a, b |]_i [| x, y |]_j
-- ([| a, b |]_i, [| x, y |]_j)
```

スカラー仮引数とテンソル仮引数は，**スカラー関数**（*scalar function*）と**テンソル関数**（*tensor function*）という 2 種類の関数を定義するためにそれぞれ使われる．スカラー関数は，テンソル

```
def min $x $y := if x < y then x else y
```

(a) min関数の定義

$$min(\begin{pmatrix}1\\2\\3\end{pmatrix}_i, \begin{pmatrix}10\\20\\30\end{pmatrix}_j) = \begin{pmatrix}min(1,10) & min(1,20) & min(1,30)\\min(2,10) & min(2,20) & min(2,30)\\min(3,10) & min(3,20) & min(3,30)\end{pmatrix}_{ij} = \begin{pmatrix}1 & 1 & 1\\2 & 2 & 2\\3 & 3 & 3\end{pmatrix}_{ij}$$

(b) 異なる添字をもつベクトルにたいする min関数の適用

$$min(\begin{pmatrix}1\\2\\3\end{pmatrix}_i, \begin{pmatrix}10\\20\\30\end{pmatrix}_i) = \begin{pmatrix}min(1,10) & min(1,20) & min(1,30)\\min(2,10) & min(2,20) & min(2,30)\\min(3,10) & min(3,20) & min(3,30)\end{pmatrix}_{ii} = \begin{pmatrix}min(1,10)\\min(2,20)\\min(3,30)\end{pmatrix}_i = \begin{pmatrix}1\\2\\3\end{pmatrix}_i$$

(c) 同じ添字をもつベクトルにたいする min関数の適用

図 11.2: スカラー関数の例として min関数の定義と適用例

を引数にとったとき，成分ごとに処理をマップする関数のことをいう．たとえば，“+” や，“-”，“*”，“/”，“∂/∂” はスカラー関数である．テンソル関数は，テンソルを引数にとったとき，テンソルをテンソルとしてそのまま処理する関数のことをいう．たとえば，行列式を計算する関数や，テンソル同士のかけ算のための関数は，テンソルの成分ごとに自動でマップされては記述できないため，テンソル関数として定義する必要がある．図 11.2 と図 11.3 でスカラー関数とテンソル関数の定義の例を説明する．

図 11.2 は，スカラー関数の例として，min関数を定義とその適用例を紹介している．仮引数の先頭に “$” が付加されており，テンソルが引数として渡された場合，成分ごとに関数が適用される．成分ごとに関数が適用された結果，図 11.2b のように関数はテンソル積と同じ形で適用される．テンソルの添字の簡約ルールはシンプルである．図 11.2c のようにスカラー関数を適用した結果，同じシンボルの添字が 2 つ以上付加されたテンソルが生成された場合，その対角成分からなるテンソルに簡約されるというルールだけである．添字簡約のルールの詳しい仕様は次節で論じる．

図 11.3 は，テンソル関数の例として，テンソルのかけ算をする “.” 関数の定義と適用例を紹介している．“%” が先頭に付加されている仮引数はテンソル仮引数となり，テンソルが引数として渡された場合，テンソルとしてそのまま渡される．図 11.3b のように，Egison は上添字と下添字の両方を組み込みでサポートしている．上添字と下添字で同じシンボルが使われていた場合は，関数 contractWithを使って縮約することができる．

上添字は “~”，下添字は “_” を使ってプログラム上で表現される．たとえば，図 11.3b の一つ目の計算例は以下のように Egison で表現される．

```
[| 1, 2, 3 |]~i . [| 10, 20, 30 |]_i
-- 140
```

```
def (.) %t1 %t2 := contractWith (+) (t1 * t2)
```

(a) “.” 関数の定義

$$\begin{pmatrix}1\\2\\3\end{pmatrix}^{i} \cdot \begin{pmatrix}10\\20\\30\end{pmatrix}_{i} = contractWith(+, \begin{pmatrix}10 & 20 & 30\\20 & 40 & 60\\30 & 60 & 90\end{pmatrix}^{i}_{i}) = 10 + 40 + 90 = 140$$

$$\begin{pmatrix}1\\2\\3\end{pmatrix}_{i} \cdot \begin{pmatrix}10\\20\\30\end{pmatrix}_{i} = contractWith(+, \begin{pmatrix}10\\40\\90\end{pmatrix}_{i}) = \begin{pmatrix}10\\40\\90\end{pmatrix}_{i}$$

$$\begin{pmatrix}1\\2\\3\end{pmatrix}_{i} \cdot \begin{pmatrix}10\\20\\30\end{pmatrix}_{j} = contractWith(+, \begin{pmatrix}10 & 20 & 30\\20 & 40 & 60\\30 & 60 & 90\end{pmatrix}_{ij}) = \begin{pmatrix}10 & 20 & 30\\20 & 40 & 60\\30 & 60 & 90\end{pmatrix}_{ij}$$

(b) “.” 関数の適用

図 11.3: テンソル関数の例として “.” 関数の定義と適用例

11.5 三種類の添字

テンソルの添字記法を扱うために Egison には三種類の添字が導入されている．最初の二つは前節で紹介した上添字と下添字で~と_で表現される．三つ目は上下添字と呼ばれる添字で~_で表現される．上下添字は，同一のシンボルをもつ上添字と下添字をもつテンソルに対してスカラー関数を適用した際に現れる．

```
[| 1 2 3 |]_i * [| 10 20 30 |]_i -- [| 10, 40, 90 |]_i
[| 1 2 3 |]~i * [| 10 20 30 |]~i -- [| 10, 40, 90 |]~i
[| 1 2 3 |]~i * [| 10 20 30 |]_i -- [| 10, 40, 90 |]~_i
```

前節でも登場した contractWith関数は上下添字について対角成分をリストに変換し第一引数の関数で畳み込む．

```
contractWith (+) [| 10, 40, 90 |]_i -- [| 10, 40, 90 |]_i
contractWith (+) [| 10, 40, 90 |]~i -- [| 10, 40, 90 |]~i
contractWith (+) [| 10, 40, 90 |]~_i -- 130
```

contractWith関数は組み込み関数 contractを使って定義されたライブラリ関数である．組み込み関数 contractは上下添字の付加されたスロットの成分のリストを返す．

```
contract [| 1, 2, 3 |]~_i -- [1, 2, 3]
contract [| [| 11, 12 |], [| 21, 22 |] |]~_i_j -- [[| 11, 12 |]_j, [| 21, 22 |]_j]
contract [| [| 11, 12 |], [| 21, 22 |] |]_i~_j -- [[| 11, 21 |]_j, [| 12, 22 |]_i]
```

```
E({A, xs}) =
  if e(xs) = [] then
    {A, xs})
  elsif e(xs) = [{k,j}, …] & p(k,xs) = p(j,xs) then
    E({diag(k, j, A), remove(j, xs))
  elsif e(xs) = [{k,j}, …] & p(k,xs) != p(j,xs) then
    E({diag(k, j, A), update(k, 0, remove(j, xs)))
```

図 11.4: 添字簡約の擬似コード

```
contract [| [| 11, 12 |], [| 21, 22 |] |]~_i~_j -- [11, 12, 21, 22]
```

11.6　添字の簡約規則

本節は，添字の簡約ルールを擬似コードを使って説明する．11.4 節で説明したように，テンソルをスカラー関数を適用したときは常にテンソル積のかたちで成分ごとに処理がマップされるため，ひとつのテンソルについての簡約ルールのみを定義すればよい．

図 11.4 は添字簡約をする擬似コードである．`E(A,xs)`は添字付きテンソルの簡約をする関数である．`A`はテンソルの成分からなる配列であり，`xs`は `A`に付加されている添字のリストを表す．たとえば，`E(A,xs)`は添字として“`~i_j_i`”を持つテンソルに対して以下のように動作する．上添字，下添字，上下添字を表すフラグとしてそれぞれ `1`，`-1`，`0`を使っている．

```
E({[|[|[|1,2|],[|3,4|]|]
    ,[|[|5,6|],[|7,8|]|]|]
 ,[{i,1}, {j,-1}, {i,-1}]}) =
{[|[|1,3|],[|6,8|]|], [{i,0}, {j,-1}]}
```

擬似コード中で使われている補助関数を説明する．`e(xs)`は `xs`中に現れる同一のシンボルからなる添字のペアを見つける関数である．`diag(k, j, A)`は `k`番目と `j`番目のスロットについて `A`の対角成分からなるテンソルを計算する関数である．`remove(k, xs)`はリスト `xs`から `k`番目の要素を取りのぞく関数である．`p(k, xs)`は連想リスト `xs`から `k`番目の値を取得する関数である．`update(k, p, xs)`は連想リスト `xs`の `k`番目の値を `p`に更新する関数である．これらの補助関数は以下のように動作する．

```
e([{i, 1}, {j, -1}, {i, -1}])        = [{1,3}]
diag(1, 2, [|[|11,12|],[|21,22|]|]) = [|11,22|]
p(2, [{i, 1}, {j, -1}])             = -1
remove(2, [{i, 1}, {j, -1}])        = [{i, 1}]
update(2, 0, [{i, 1}, {j, -1}])     = [{i, 1}, {j, 0}]
```

11.7 テンソル関数の適用

添字付きのテンソルがテンソル関数に渡されるとき，そのテンソルの添字ごとテンソル関数に適用される．図 11.3a で定義した “.” 関数でこの挙動を確認する．図 11.3b では，“.” 関数を適用をしたときなされる展開が段階的に表示されている．contractWithの内側に添字付きのテンソルがあらわれるが，これはテンソルの添字ごとテンソル関数に渡されることで可能になっている．この挙動のおかげで以下のようにテンソル関数についてテンソルの添字記法を使って関数適用できるようになっている．

```
[|1,2,3|]~i . [|10,20,30|]_i -- 140
[|1,2,3|]_i . [|10,20,30|]_j -- [|[|10,20,30|],[|20,40,60|],[|30,60,90|]|]_i_j
[|1,2,3|]_i . [|10,20,30|]_i -- [|10,40,90|]_i
```

テンソル関数の中で引数として渡されたテンソルに追加で添字を付加したいことがある．追加で新しい添字を付加するためには，以下のように...をテンソルと添字の間に挟めば良い．この機能は，11.13 節で紹介する subrefsと suprefsという組み込み関数を使った糖衣構文として実装されている．この機能の使用例は第 13 章で紹介する．

```
A..._i_j
```

11.8 省略された添字の補完

添字が省略されたテンソルが関数に適用された場合，Egison は自動で添字を補完する．本節はこの補完の規則を説明する．

以下，A，Bを 2 階のテンソル，つまり行列であるとする．添字を省略した状態でスカラー関数に A，Bを同時に適用すると，下記のように同じ組み合わせの添字が補完される．

```
A + B -- A_t1_t2 + B_t1_t2
```

関数適用の前に “!” が付加されていた場合は，下記のように違う添字が補完される．

```
A !+ B -- A_t1_t2 + B_t3_t4
```

添字が省略されたテンソルが，テンソル関数に適用された場合は，補完はおこなわれない．

上記の添字の補完規則には実は必然性がある．添字が省略された場合の添字補完規則を上記のように設定すると，微分形式のための演算子を簡潔に定義できる．微分形式の演算子の定義の具体例は，第 13 章で紹介する．あとで第 13 章でも述べるように，“!” が使われる必要があるのは，微分形式の演算子の定義の中だけになる．そのため，基本的にユーザーは “!” については意識しなくてよい．

11.9　スカラー仮引数とテンソル仮引数の実装

本節はスカラー仮引数とテンソル仮引数の実装方法について説明する．テンソル仮引数は，テンソルをテンソルとしてそのまま扱うため，通常の仮引数として実装できる．たいして，スカラー仮引数は tensorMapという組み込み関数を使ったテンソル仮引数しかあらわれない以下のようなプログラムに変換できる．

```
\ $x $y -> ...
-- => \ %x %y ->
--      tensorMap (\ %x -> tensorMap (\ %y -> ...) y))
--                x
```

名前が示すとおり，tensorMapは第一引数の関数を第二引数のテンソルの成分ごとに適用してテンソルを返す関数である．ただ，第一引数の関数に第二引数のテンソルの成分を適用した結果がテンソルである場合，その結果のテンソルの添字は，tensorMap全体の結果のテンソルの添字の末端に移動するというテンソル特有の仕様がある．

図 11.2 で定義した min関数を使って上記の挙動を確認する．min関数は引数のテンソルを以下のように処理する．

```
min [|1,2,3|]_i [|10,20,30|]_j -- [|[|1,1,1|],[|2,2,2|],[|3,3,3|]|]_i_j
min [|1,2,3|]_i [|10,20,30|]_i -- [|1,2,3|]_i
```

一つ目の式の結果のテンソルの添字が “_i_j” となっている．もし，tensorMapが単純に関数をテンソルの成分に適用するだけであったら，結果のテンソルは，[|[|1 1 1|]_j [|2 2 2|]_j [|3 3 3|]_j|]_iのようになっているはずである．しかし，上で説明したとおり，tensorMapは内側のテンソルの添字を外側のテンソルの最後に移動する．このために結果のテンソルの添字が “_i_j” となる．この仕組みがテンソルの添字記法を使ってスカラー関数を適用することを可能にしている．

スカラー仮引数は上記のように実装できるが，実際は効率化のための工夫を施した変換がなされる．上記の方法だと，対角化するために対角成分しか必要ない場合でも対角成分以外の成分ももつテンソルを生成してしまう．Egison はテンソルの成分の評価も遅延するため，計算コストはそれほどかからないが，無駄にメモリを確保してしまうため，これはあまりよくない．そのために，2 引数のスカラー関数は tensorMap2という組み込み関数を使った式に変換する．

```
\ $x $y -> ...
-- => \ %x %y ->
--      tensorMap2 (\ (%x, %y) -> ...)
--                x y
```

tensorMap2は 2 引数関数と二つのテンソルを引数にとり，それぞれの成分について第一引数の関数を適用しテンソルを返す．tensorMap2は，結果のテンソルに必要な成分だけを計算するように実装されている．

11.10 反転スカラー仮引数

偏微分をするために使われる∂/∂関数はスカラー関数である．しかし，∂/∂関数はただのスカラー関数ではない．∂/∂関数は第二引数のテンソルの添字の上下を反転する．たとえば，`(∂/∂ Γ~i_j_k x~l)`は，添字として "`~i_j_k_l`" をもつ四階のテンソルを返す．

∂/∂のようなスカラー関数を定義するために，Egison には**反転スカラー仮引数**（*inverted scalar parameters*）が組み込もで用意されている．反転スカラー仮引数は，仮引数の先頭に "`*$`" を付加することによって表現される．反転スカラー仮引数を含むプログラムは下記の例のように変換される．`flipIndices`は，その引数のテンソルのすべての添字の上下を反転させる組み込み関数である．上下添字は反転させても上下添字のままである．∂/∂の第二引数は反転スカラー仮引数を使って定義される．

```
def ∂/∂ $f *$x := ...
-- => def ∂/∂ %f %x := tensorMap (\ %f -> tensorMap (\ %x -> ...) (flipIndices x)) f
```

以下は，∂/∂の使用例である．

```
∂/∂ [|(r * (sin θ)),(r * (cos θ))|]_i [|r,θ|]_j
-- [|[|(sin θ),(r * (cos θ))|],[|(cos θ),(-1 * r * (sin θ))|]|]_i~j
∂/∂ [|(r * (sin θ)),(r * (cos θ))|]_i [|r,θ|]_i
-- [|(sin θ),(-1 * r * (sin θ))|]~_i
```

11.11 generateTensor 式によるテンソルの生成

`generateTensor`式はテンソルの初期化をするために便利な組み込み構文である．`generateTensor`式は第一引数にリストを引数にとる関数，第二引数に生成したいテンソルのサイズをとる．たとえば，すべての成分が 1 であるサイズが $(3, 4)$ の行列を生成するには以下のように `generateTensor`式を書く．

```
generateTensor [2]#1 [3, 4]
-- [| [| 1, 1, 1, 1 |], [| 1, 1, 1, 1 |], [| 1, 1, 1, 1 |] |]
```

無名 `match`関数と組み合わせると短い記述でいろいろなテンソルを生成できる．たとえば単位行列を生成するには以下のようにすればよい．

```
generateTensor (\match as list integer with
                 | [$n, #n] -> 1
                 | _ -> 0)
               [3, 3]
-- [| [| 1, 0, 0 |], [| 0, 1, 0 |], [| 0, 0, 1 |] |]
```

11.12　テンソルの宣言

テンソルを変数に束縛する方法を解説する．まずテンソルは通常の値と同様に def式や let式で定義することができる．

```
def A := [| [| 11, 12 |], [| 21, 22 |] |]

A -- [| [| 11, 12 |], [| 21, 22 |] |]
```

数学や物理では同じ変数名で添字の上下の型が違うテンソルにそれぞれ別の値を入れることがよくある．Egison でもその慣習に対応するために，添字の上下の型ごとに違うテンソルを定義できるようにしている．Egison は，変数を参照するとき，変数名と添字の型の両方を使う．変数定義のときに，変数名の後ろに添字の型を続けて記述することによって添字の型を指定する．

```
def g__ := [| [| 2, 2 |], [| 2, 2 |] |]
def g~~ := [| [| 1 / 2, 1 / 2 |], [| 1 / 2, 1 / 2 |] |]

g_1_1 -- 2
g~1~1 -- 1 / 2
```

添字の型を指定してテンソルが束縛されている変数に参照するには，**ダミーシンボル**（*dummy symbol*）を使うのが便利である．ダミーシンボルは，#で表現され，すべてユニークなシンボルとして扱われる．ダミーシンボルは Egison に組み込みで実装されている機構である．ダミーシンボルを使うと，上記で定義した g~~と g__を以下のように参照できる．

```
g_#_# -- [| [| 2, 2 |], [| 2, 2 |] |]_#_#
g~#~# -- [| [| 1 / 2, 1 / 2 |], [| 1 / 2, 1 / 2 |] |]~#~#
```

シンボルを使っても添字の型を指定してテンソルを参照することもできるが，ダミーシンボルを使う方が定義済みの変数をシンボルとして使おうとする間違いが起きにくい．

```
g_i_j -- [| [| 2, 2 |], [| 2, 2 |] |]_i_j
g~i~j -- [| [| 1 / 2, 1 / 2 |], [| 1 / 2, 1 / 2 |] |]_i_j
```

ダミーシンボルはテンソルの添字として使う以外の用途はないように考えているが，ふつうのシンボルのように扱うことができる．以下は二つのダミーシンボルを足し合わせている．二つのダミーシンボルを足し合わせると，以下のようにまとめられないが，これはこの二つのダミーシンボルがそれぞれ異なるシンボルとして扱われるからである．

```
# + # -- # + #
```

数式でテンソルを定義するときのように，左辺の変数の添字としてシンボルを書くこともできる．

```
def A_i_j := [| [| 11, 12 |], [| 21, 22 |] |]_i_j
def B_i_j := [| [| 11, 12 |], [| 21, 22 |] |]_j_i

A_#_# -- [| [| 11, 12 |], [| 21, 22 |] |]_#_#
B_#_# -- [| [| 11, 21 |], [| 12, 22 |] |]_#_#
```

この機能は以下のような糖衣構文として Egison では実装されている．

```
def A_i_j := ...
-- => def A__ := withSymbols [i, j]
--                  transpose [i, j] ...
```

transpose関数は，テンソルの成分を第一引数で指定された順番に転置する組み込み関数である．

同じ変数名と添字の型のテンソルが環境にない場合は，添字の型を末尾から順に取り除いて参照をおこなう．たとえば，下記の例のように T~と T_が定義されている状態で，T~#_#を参照すると，T~_が環境にないため，末尾の添字の型をひとつ取り除いて T~を参照する．T~は定義済みであるため，その値が参照される．

```
def T~ := [| [| 11, 12 |], [| 21, 22 |] |]
def T_ := [| [| -11, -12 |], [| -21, -22 |] |]

T~#_# -- [| [| 11, 12 |], [| 21, 22 |] |]
```

11.13 添字付加のために便利な構文

階数がパラメーター n によって変わるテンソル A について，$A_{i_1 \dots i_n}$ のように添字を付加したいことがある．このために subrefsと subrefs!という組み込み関数が Egison には用意されている．これらの関数は第一引数にテンソルに第二引数に添字のリストを付加する．subrefsは第一引数のテンソルの既存の添字に追加で第二引数に添字のリストを付加するのに対して，subrefs!は既存の添字を消して新たに添字を付加し直す．

```
subrefs! A (map 1#i_%1 [1..n])
subrefs  A (map 1#i_%1 [1..n])
```

$A_{i_1 \dots i_n}$ のような形のテンソルは数式でよく現れるため，上記の subrefs!と subrefsを使ったプログラムについては以下のような糖衣構文が用意されている．

```
A_(i_1)...(i_n)
A..._(i_1)...(i_n)
```

上記の内容は $A^{i_1 \cdots i_n}$ のような上添字をもつテンソルにも用意されており，このために suprefsと suprefs!という組み込み関数が Egison には用意されている．

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{12} & a_{22} & a_{23} \\ a_{13} & a_{23} & a_{33} \end{pmatrix}$$

(a) 対称なテンソルの例

$$\begin{pmatrix} 0 & a_{12} & a_{13} \\ -a_{12} & 0 & a_{23} \\ -a_{13} & -a_{23} & 0 \end{pmatrix}$$

(b) 歪対称なテンソルの例

図 11.5: テンソルの対称性と歪対称性

`subrefs`や `suprefs`のように既存の添字に追加で添字を付加する機能は，第 13 章で紹介する微分形式についての関数を定義するときに使われる．

11.14 テンソルの対称性・歪対称性の宣言

テンソルが**対称**であるとは，図 11.5a の例のように，上三角の成分と下三角の成分が一致することをいう．テンソルが**歪対称**であるとは，図 11.5b の例のように，上三角の成分と下三角の成分の符号が反転することをいう．テンソルが歪対称であるとき，対角成分は 0 となる．

数学や物理で現れるテンソルの多くが対称性をもつ．たとえば，13.2 節で計算するリーマン曲率テンソル R_{abcd} は $R_{abcd} = -R_{abdc}$，$R_{abcd} = -R_{bacd}$，$R_{abcd} = R_{cdab}$ という複数の対称性をもつ．数学者や物理学者はこれらの対称性を意識しながら適宜計算を省略する．Egison でも対称性を意識した計算の最適化を実装したいが，プログラミング言語でこの最適化をするには，明示的にテンソルの対称性を宣言する必要がある．Egison はテンソルの対称性を宣言するための記法が実装されている．本節は，この記法の使い方と実装を解説する．

11.14.1 テンソルの対称性を宣言する記法

リーマン曲率テンソルには，上記の対称性に加えて第一ビアンキ恒等式 $R_{abcd}+R_{acdb}+R_{adbc} = 0$ とよばれる対称性もある．この第一ビアンキ恒等式の表現として，$R_{a[bcd]} = 0$ という表記がある．Egison に実装されているテンソルの対称性を宣言する記法は，この表記法に触発されて得られたものである．

具体例をみせながら，その記法を紹介する．Egison でリーマン曲率テンソル R_{abcd} は下記のように定義できる．右辺に複雑な式があるが，本節では無視していただいて問題ない．左辺だけに注目してほしい．

```
def R_a_b_c_d := withSymbols [i, j]
```

```
  g_a_i . (∂/∂ Γ~i_b_d x~c - ∂/∂ Γ~i_b_c x~d +
           Γ~j_b_d . Γ~i_j_c - Γ~j_b_c . Γ~i_j_d)
```

$R_{abcd} = -R_{abdc}$ という歪対称性を宣言するには，以下のように添字_a_bを角括弧 []で囲む．

```
def R[_a_b]_c_d := withSymbols [i, j]
  g_a_i . (∂/∂ Γ~i_b_d x~c - ∂/∂ Γ~i_b_c x~d +
           Γ~j_b_d . Γ~i_j_c - Γ~j_b_c . Γ~i_j_d)
```

上記のように R_{abcd} を定義すると，たとえば R_{21cd} を参照したとき，R_{12cd} の計算結果の符号を反転した値を返すようになる．対称性を複数同時に宣言することもできる．$R_{abcd} = -R_{abdc}$ と $R_{abcd} = -R_{bacd}$ を同時に宣言するには，以下のように添字_a_bと_c_dの両方を角括弧 []で囲む．

```
def R[_a_b][_c_d] := withSymbols [i, j]
  g_a_i . (∂/∂ Γ~i_b_d x~c - ∂/∂ Γ~i_b_c x~d +
           Γ~j_b_d . Γ~i_j_c - Γ~j_b_c . Γ~i_j_d)
```

こうすると，たとえば R_{2121} を参照したとき，R_{1212} の計算結果を返すようになる．

$R_{abcd} = R_{cdab}$ のような複雑な対称性も記述できる．この対称性を記述するためには添字_a_bと_c_dをそれぞれひとまとまりで扱うために丸括弧 ()で囲む．そして，これらを波括弧{}で囲む．角括弧 []が歪対称性を宣言するために使われていたのに対し，波括弧{}は対称性を宣言するために使われる．

```
def R{(_a_b)(_c_d)} := withSymbols [i, j]
  g_a_i . (∂/∂ Γ~i_b_d x~c - ∂/∂ Γ~i_b_c x~d +
           Γ~j_b_d . Γ~i_j_c - Γ~j_b_c . Γ~i_j_d)
```

三つの対称性 $R_{abcd} = -R_{abdc}$，$R_{abcd} = -R_{bacd}$，$R_{abcd} = R_{cdab}$ を同時に宣言することもできる．

```
def R{[_a_b][_c_d]} := withSymbols [i, j]
  g_a_i . (∂/∂ Γ~i_b_d x~c - ∂/∂ Γ~i_b_c x~d +
           Γ~j_b_d . Γ~i_j_c - Γ~j_b_c . Γ~i_j_d)
```

まとめると，Egison は対称性の宣言には三種類の括弧を使い分ける．波括弧{}は対称性を宣言するために使う．角括弧 []は歪対称性を宣言するために使う．丸括弧 ()は複数の添字をひとまとまりにするために使う．

対称なテンソルの添字は隣り合っていることが多いため，数学や物理で現れるテンソルの対称性は，この記法でほぼ宣言することができる．:=の右辺のプログラムを全く変更せずに対称性を宣言できるところはこの記法のよいところである．

11.14.2　対称性の宣言によってテンソルの定義が短くなる例

さきほど，この手法の利点として，:=の右辺のプログラムを全く変更する必要がないと述べたが，:=の右辺のプログラムが簡略化できるケースもある．たとえば，下記のようにテンソルの各ブ

ロックごとに異なる式を使って値が定義されるテンソルがそのようなケースに該当する．

```
def R'_i_j_k_l :=
  generateTensor
    (\match as list integer with
       | [#1, #1, _, _] -> 0
       | [_, _, #1, #1] -> 0
       | [#1, $b, #1, $d] -> -1 * p^2 * δ~(b - 1)_(d - 1)
       | [$a, #1, #1, $d] ->      p^2 * δ~(a - 1)_(d - 1)
       | [#1, $b, $c, #1] ->      p^2 * g_(b - 1)_(c - 1)
       | [$a, #1, $c, #1] -> -1 * p^2 * g_(a - 1)_(c - 1)
       | [#1, $b, $c, $d] -> -1 * p * ∇J_(b - 1)_(c - 1)~(d - 1)
       | [$a, #1, $c, $d] ->      p * ∇J_(a - 1)_(c - 1)~(d - 1)
       | [$a, $b, #1, $d] -> -1 * p * ∇J~(d - 1)_(a - 1)_(b - 1)
       | [$a, $b, $c, #1] ->      p * ∇J_(c - 1)_(a - 1)_(b - 1)
       | [$a, $b, $c, $d] -> R_(a - 1)_(b - 1)_(c - 1)~(d - 1)
                             + -1 * p^2 * J_(b - 1)_(c - 1) * J_(a - 1)~(d - 1)
                             +      p^2 * J_(a - 1)_(c - 1) * J_(b - 1)~(d - 1)
                             +  2 * p^2 * J_(a - 1)_(b - 1) * J_(c - 1)~(d - 1))
    [5, 5, 5, 5]
```

対称性により条件分岐を大きく減らすことができる．

```
def R'{[_i_j][_k_l]} :=
  generateTensor
    (\match as list integer with
       | [#1, #1, _, _] -> 0
       | [#1, $b, #1, $d] -> -1 * p^2 * δ~(b - 1)_(d - 1)
       | [#1, $b, $c, $d] -> -1 * p * ∇J_(b - 1)_(c - 1)~(d - 1)
       | [$a, $b, $c, $d] -> R_(a - 1)_(b - 1)_(c - 1)~(d - 1)
                             + -1 * p^2 * J_(b - 1)_(c - 1) * J_(a - 1)~(d - 1)
                             +      p^2 * J_(a - 1)_(c - 1) * J_(b - 1)~(d - 1)
                             +  2 * p^2 * J_(a - 1)_(b - 1) * J_(c - 1)~(d - 1))
    [5, 5, 5, 5]
```

11.14.3 実装

テンソルの対称性を宣言する記法の実装方法について解説する．この記法は generateTensor式を使う糖衣構文として実装されている．本節では，この変換を具体例をいくつか紹介することによって説明する．

まず対称性を宣言する波括弧{}がどう展開されるか説明する．以下のプログラムの...の箇所にはなにか適当なテンソルを返すプログラムがはいる．

```
def X{_i_j} := ...
```

上記の形のプログラムは，以下のようなプログラムに展開される．

```
def X_i_j :=
 let tmpX_i_j := ...
   generateTensor
     (\i j -> if i > j then tmpX_j_i
              else          tmpX_i_j)
     (tensorShape tmpX_#_#)
```

tmpXという新たな変数を導入し，定義したいテンソルを代入する．X自体は generateTensor式を使って定義する．この generateTensor式は，下半分の成分については，上半分の成分を参照するように定義されている．Egison はテンソルの成分の評価を遅延するため，tmpXの下半分の成分は計算されることはない．そのため，テンソル成分の計算コストが半分になる．

歪対称性を宣言する角括弧 []も同じように展開される．

```
def X[_i_j] := ...
```

下半分の成分の符号が上半分の成分に対して反転していることと，対角成分が 0 になるように generateTensor式の内部が変更されている．

```
def X_i_j :=
  let tmpX_i_j := ...
    generateTensor
      (\i j -> if i > j      then -tmpX_j_i
               else if i = j then 0
               else               tmpX_i_j)
      (tensorShape tmpX_#_#)
```

第 12 章

関数シンボル

名前が f，引数が x, y と決まっているが，中身は未定義の関数 $f(x, y)$ は，紙の上では f と引数を省略して表記されることが多い．このシンボリックな関数の引数を省略する表記をプログラミングにも導入するための機能が**関数シンボル**（*function symbol*）である．関数シンボル自体の動機はシンプルであるが，関数シンボルは Egison 特有の言語機能であるため，一つの独立した章を使ってその実装方法も含め紹介する．

12.1　関数シンボルによるシンボリックな関数の引数の省略

関数シンボルが特に力を発揮するのは，微分を扱う計算のときである．以下では `x`，`y`を引数にとる関数シンボル `f`を定義している．そして，その関数シンボル `f`を `x`や `y`について微分している．`f|x`は，関数 `f`を `x`で偏微分した結果の関数を表す．`f|x|y` は，関数 `f`を `x`で偏微分した後，さらに `y`で偏微分した結果の関数を表す．“`|`” は，“`_`” や “`~`” と同様 Egison 組み込みの記号である．以下に関数シンボルの微分を標準的な Egison 出力と LaTeX 形式の出力の両方の表示例を示す．“`∂/∂`” は Egison で定義されているライブラリ関数である．

```
def f := function (x, y)

∂/∂ f x -- f|x
∂/∂ (∂/∂ f x) y -- f|x|y
```

`f`は `x`と `y`についての関数であるので，以下のように `z`で微分すると `0`になる．

```
∂/∂ f z -- 0
```

物理の計算では，t, x, y, z の 4 つの引数をとる関数がよく登場する．もし関数シンボルのような仕組みがないと，式が長くなり，読み難くなるため，関数シンボルの仕組みはプログラムの読み書きしやすくする．

合成関数の微分にも対応していることが，Egison の関数シンボルの重要な特徴である．たとえば，関数シンボルは，下記の例のように合成関数の微分にも対応できる．`function`式の仮引数が変

数名 ($x や$y) ではなく式をとるのは，成関数の微分に対応するためである．さきほどと同様に f を，$f(x,y)$ の省略であるとする．また，$x = r\cos\theta, y = r\sin\theta$ という関係式もあるとする．このとき，f を r で微分すると，$\frac{\partial f}{\partial x}\cos\theta + \frac{\partial f}{\partial y}\sin\theta$ という結果を返している．

```
def x := r * cos θ
def y := r * sin θ
def f := function (x, y)

∂/∂ f r
-- f|x * cos θ + f|y * sin θ
```

12.2 関数シンボルを成分にもつテンソル

数学や物理では，テンソルの成分として関数がでてくることがある．たとえば，ベクトル場やテンソル場を表現するときにそのような場面がある．このような場面に対応するため，テンソルの成分として関数シンボルを使うことを Egison は許している．以下の例のようにテンソルの成分が関数シンボルであった場合，テンソルを束縛している変数名に成分の位置の添字を付加した名前をもつ関数シンボルが生成される．

```
def g_i_j := generateTensor
      (\match as list eq with
         | [$n, #n] -> function (x, y, z)
         | _ -> 0)
      [3, 3]

g_i_j
-- [| [| g_1_1, 0, 0 |], [| 0, g_2_2, 0 |], [| 0, 0, g_3_3 |] |]_i_j∂∂

/ g_i_j x
-- [| [| g_1_1|x, 0, 0 |], [| 0, g_2_2|x, 0 |], [| 0, 0, g_3_3|x |] |]_i_j
```

12.3 関数シンボルの内部表現とパターンマッチ

関数シンボルは Functionデータコンストラクタを使って Egison 内部では表現されている．Functionデータコンストラクタは三つの引数をとる．第一引数には関数シンボルの名前のシンボルがはいる．第二引数には関数シンボルの引数のシンボルがはいる．第三引数には関数シンボルの引数の値がはいる．12.1 節で紹介した合成関数の微分を実現するためには，この三つの引数がすべて必要である．

```
def x := r * cos θ
def y := r * sin θ
```

```
def f := function (x, y)

fromMathExpr f
-- Div (Plus [Term 1 [(Function f [x, y] [r * cos θ, r * sin θ], 1)]]) (Plus [Term 1
     []])
```

funcパターンコンストラクタの第二引数

```
match f as mathExpr with
  | func $name $argnames $args $r -> (name, argnames, args, r)
```

```
def ∂/∂ $f *$x :=
  match f as mathExpr with
    ...
    | func _ $argnames $args ->
      sum (map2 (\s r -> (userRefs f [s]) * ∂/∂ r x) argnames args)
    ...
```

12.4 関数シンボルの実装方法

関数シンボルの実装で注意すべきことは，関数シンボルは自身の名前を保持することである．そのために Egison インタプリタは現在定義しようとしている変数の名前を管理している．また，12.2 節で解説したテンソルの成分に現れる関数シンボルをサポートするためには，現在テンソルのどの位置の成分を定義しているのか管理する必要がある．そのために Egison インタプリタは現在定義しようとしているテンソルの成分の位置も管理している．これらは Egison インタプリタの実装の evalの関数の追加引数として管理されている．

第 13 章

Egison で計算する微分幾何

本章は，数式処理システムとしての Egison の実践的な応用例として微分幾何への応用を紹介する．Egison を使うと，さまざまな多様体の曲率が定義の数式に近いプログラムを記述するだけで計算できたり，外微分やホッジ作用素のような微分形式のための作用素もプログラムで簡潔に定義できる．本章の解説は読者に微分幾何の知識を仮定している．微分幾何に興味を持つ読者には，小林昭七による『曲線と曲面の微分幾何』（裳華房）がおすすめである．

13.1　微分幾何と Egison

微分幾何は手計算がとくに大変な分野である．例えば，曲率が自明ではないもっとも単純な多様体は球面 S^2 であるが，そのリーマン曲率テンソルを手計算すると，初めての場合，30 分くらいかかる．続けて，その次に単純な多様体であるトーラス T^2 のリーマン曲率テンソルを計算すると，またしても 30 分以上はかかる．実際の研究のためには，より高次元の複雑な多様体の曲率を計算することが多い．このような場合，計算に一ヶ月以上かかることもある．そのため，微分幾何は数式処理システムがもっとも応用しやすい分野でもある．

数式処理システムを使えば，S^2 のリーマン曲率テンソルは，球面座標系，リーマン計量，クリストッフェル記号，リーマン曲率テンソルの定義をプログラムとして記述すれば，数秒で計算できる．T^2 については，S^2 のリーマン曲率テンソルを計算をするプログラムを書いた後であれば，座標系の設定以外の部分は使いまわせるため，数分の作業で計算できる．より高次元の複雑な多様体についても，計量の設定を変更するだけで，計算することができる．

Egison は，9.5 節で解説した微分演算子，第 11 章で解説したテンソルのための言語機能，第 12 章で解説した関数シンボルを組み合わせて，従来の数式処理システムよりも数学の記法に近い形で微分幾何の計算をプログラムとして記述できる．また，11.8 節で触れたように，外微分やホッジ作用素のような微分形式のための作用素もプログラムで簡潔に定義するための仕組みも提供している．本章では，これらの機能を実演し，微分幾何の研究・教育に Egison が役に立つことを紹介する．

13.2 リーマン曲率テンソルの計算

本節は，球面について，リーマン計量や，クリストッフェル記号，リーマン曲率テンソル，リッチ曲率，スカラー曲率といった様々な微分幾何的な量を Egison を使って計算するプログラムを解説する．図 13.1 は，Jupyter Notebook 上で球面のリーマン曲率テンソルを計算する Egison プログラムとその実行結果の出力である．さまざまな幾何学的不変量は，曲率から計算される．本節で解説するプログラムの計量の部分だけを変えれば，あらゆる多様体の曲率を計算できる．そのため，本節のプログラムを理解すれば，実際の微分幾何の研究にも Egison が使える．本節は，このプログラムの各箇所について順番に説明していくことによって進める．

13.2.1 球座標系 - [1]-[2]

球座標系は，r と θ，ϕ の 3 つのパラメーターからなる．r は原点からの距離をあらわす．θ は z 軸との角度，緯度をあらわす．ϕ は xy 平面における x 軸からの回転量，経度をあらわす．球座標系の r の値を固定すれば，球面の座標系を得ることができる．球座標系とデカルト座標系は，$x = r\cos\theta\sin\phi, y = r\cos\theta\cos\phi, z = r\sin\theta$ という関係式で結ばれている．以上が，図 13.1 の [1] と [2] で表現されている内容である．

13.2.2 接ベクトルとリーマン計量 - [3]-[7]

球面上の θ，ϕ によって定まる各点の上の接空間の基底となるベクトルをまず計算している．[3] でそのようなベクトルを計算し，[4] でそのベクトルを表示している．[4] の出力の 1 行目と 2 行目が基底となるベクトルの成分である．

リーマン計量は，各点における接ベクトルの大きさや向きの情報をもつ行列として表現できる．[5] でベクトルの内積を計算すことによって基底ベクトルの大きさが計算されている．[6] では，[5] で定義したリーマン計量の値を表示している．球座標系においては，座標から自然に定まる基底の大きさが，各点で異なる．たとえば，経度（ϕ）方向の基底ベクトルの大きさが，緯度（θ）によって変わる．この情報は，`g_i_j`の $2,2$ 成分の $r^2\sin^2\theta$ にあらわれている．

[7] では，逆行列を計算する関数 `M.inverse`を使って，上添字がつくリーマン計量を求めている．リーマン計量には下添字がつく g_{ij} と上添字がつく g^{ij} の 2 種類がある．

13.2.3 クリストッフェル記号とリーマン曲率テンソル - [8]-[12]

[8] と [9] は，第一種クリストッフェル記号と第二種クリストッフェル記号の数式による定義をプログラムとして記述したものである．数式に非常に近いかたちでプログラムとして表現されている．直感的には，クリストッフェル記号 $\Gamma^i{}_{jk}$ は，基底ベクトル e_j を e_k 方向に動かしたときの e_i 方向への変化の具合を表現する．

```
In [1]: def x := [| θ, φ |]

In [2]: def X := [| r * sin θ * cos φ, r * sin θ * sin φ, r * cos θ |]

In [3]: def e_i := ∂/∂ X x_i

In [4]: e_#
```

$$\begin{pmatrix} rcos(\theta)cos(\varphi) & rcos(\theta)sin(\varphi) & -rsin(\theta) \\ -rsin(\theta)sin(\varphi) & rsin(\theta)cos(\varphi) & 0 \end{pmatrix}_{\#}$$

```
In [5]: def g_i_j := generateTensor (\[a, b] -> e_a~i . e_b_i) [2, 2]

In [6]: def g~i~j := M.inverse g_#_#

In [7]: (g_#_#, g~#~#)
```

$$(\begin{pmatrix} r^2 & 0 \\ 0 & r^2 sin(\theta)^2 \end{pmatrix}_{\#\#}, \begin{pmatrix} \frac{1}{r^2} & 0 \\ 0 & \frac{1}{r^2 sin(\theta)^2} \end{pmatrix}^{\#\#})$$

```
In [8]: def Γ_i_j_k := (1 / 2) * (∂/∂ g_i_k x~j + ∂/∂ g_i_j x~k - ∂/∂ g_j_k x~i)

In [9]: def Γ~i_j_k := withSymbols [m] g~i~m . Γ_m_j_k

In [10]: (Γ_1_#_#, Γ_2_#_#)
```

$$(\begin{pmatrix} 0 & 0 \\ 0 & -r^2 sin(\theta)cos(\theta) \end{pmatrix}_{\#\#}, \begin{pmatrix} 0 & r^2 sin(\theta)cos(\theta) \\ r^2 sin(\theta)cos(\theta) & 0 \end{pmatrix}_{\#\#})$$

```
In [11]: def R~i_j_k_l := withSymbols [m]
          ∂/∂ Γ~i_j_l x~k - ∂/∂ Γ~i_j_k x~l + Γ~m_j_l . Γ~i_m_k - Γ~m_j_k . Γ~i_m_l

In [12]: (R~1_2_#_#, R~2_1_#_#)
```

$$(\begin{pmatrix} 0 & sin(\theta)^2 \\ -sin(\theta)^2 & 0 \end{pmatrix}_{\#\#}, \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}_{\#\#})$$

```
In [13]: def Ric_i_j := withSymbols [m] sum (contract R~m_i_m_j)

In [14]: Ric_#_#
```

$$\begin{pmatrix} 1 & 0 \\ 0 & sin(\theta)^2 \end{pmatrix}_{\#\#}$$

```
In [15]: def scalarCurvature := withSymbols [i, j] g~i~j . Ric_i_j

In [16]: scalarCurvature
```

$$\frac{2}{r^2}$$

図 13.1: 球面のリーマン曲率テンソルの計算

$$\Gamma_{ijk} = \frac{1}{2}(\frac{\partial g_{ij}}{\partial x^k} + \frac{\partial g_{ik}}{\partial x^j} - \frac{\partial g_{kj}}{\partial x^i}) \qquad \Gamma^i{}_{jk} = g^{im}\Gamma_{mjk}$$

リーマン曲率テンソルも，[11] のように，その数式による定義をそのままプログラムとして記述できている．直感的には，リーマン曲率テンソル $R^i{}_{jkl}$ は，e_k と e_l によって作られる閉路に

沿って e_j を動かしたときの e_i 方向への変化の具合を表現する．

$$R^i{}_{jkl} = \frac{\partial \Gamma^i{}_{jl}}{\partial x^k} - \frac{\partial \Gamma^i{}_{jk}}{\partial x^l} + \Gamma^m_{jl}\Gamma^i{}_{mk} - \Gamma^m_{jk}\Gamma^i{}_{ml}$$

13.2.4　リッチ曲率とスカラー曲率 - [13]-[16]

[13]-[14] でリッチ曲率を，[15]-[16] でスカラー曲率を計算している．それぞれ下記の数式に対応している．

$$Ric_{ij} =^m_{imj} \qquad S = g^{ij} Ric_{ij}$$

13.2.5　計量を変えてみる

本節で解説したプログラムの計量の部分を変えるだけで，いろいろな多様体のクリストッフェル記号やリーマン曲率テンソルを計算できる．

たとえば，トーラス T^2 のリーマン曲率テンソルを計算するには，球座標系をトーラスの座標系に変更すればよい．トーラスの座標系はたとえば，図 13.2 の [1]-[2] のように設定できる．バッククォート (“`”) が，この式の中で使われている．10.2 節で解説したように，バッククォートのついた数式は，Egison 処理系に一つのシンボルと同じように扱われる．ここでは，バッククォートを適切に挿入することより，計算プロセスや，最終的な計算結果が単純になっている．

また，計量を以下のように定義される Schwarzschild 計量に変えれば，原点を中心に一つだけ質量をもつ天体がある宇宙についての一般相対性理論の計算ができる．この計算結果は，「時空の曲がり＝重力」ということをあらわしている．

```
def g_i_j := [| [| `(c^2 * r - 2 * G * M) / c^2 * r, 0, 0, 0 |]
                [| 0, -1 * c^2 * r / `(c^2 * r - 2 * G * M), 0, 0 |]
                [| 0, 0, -1 * r^2, 0 |]
                [| 0, 0, 0, -1 * r^2 * (sin `$\theta$`)^2 |] |]
```

13.3　曲率形式の計算 - 微分形式を使った計算（1）

11.7 節や，11.8 節，11.13 節で言及したように，Egison のテンソル関連の機能は微分形式を使った計算を簡潔に表現できるように設計されている．微分形式を使うと，テンソルに関する演算をより抽象的に記述できるようになる．

本節では，微分形式の計算の例として，曲率形式の計算を紹介する．曲率形式は，リーマン曲率テンソルを一般化したもので，曲率形式の公式 $\Omega^i{}_j = d\omega^i{}_j + \omega^i{}_k \wedge \omega^k{}_j$ というリーマン曲率テンソ

```
In [1]: def x := [| θ, φ |]
In [2]: def X := [| `(a * cos θ + b) * cos φ, `(a * cos θ + b) * sin φ,  a * sin θ |]
In [3]: def e_i := ∂/∂ X x_i
In [4]: e_#
```

$$\begin{pmatrix} -a\sin(\theta)\cos(\varphi) & -a\sin(\theta)\sin(\varphi) & a\cos(\theta) \\ -(a\cos(\theta)+b)\sin(\varphi) & (a\cos(\theta)+b)\cos(\varphi) & 0 \end{pmatrix}_{\#}$$

```
In [5]: def g_i_j := generateTensor (\[a, b] -> e_a~i . e_b_i) [2, 2]
In [6]: def g~i~j := M.inverse g_#_#
In [7]: (g_#_#, g~#~#)
```

$$\left(\begin{pmatrix} a^2 & 0 \\ 0 & (a\cos(\theta)+b)^2 \end{pmatrix}_{\#\#}, \begin{pmatrix} \frac{1}{a^2} & 0 \\ 0 & \frac{1}{(a\cos(\theta)+b)^2} \end{pmatrix}^{\#\#}\right)$$

```
In [8]: def Γ_i_j_k := (1 / 2) * (∂/∂ g_i_k x~j + ∂/∂ g_i_j x~k - ∂/∂ g_j_k x~i)
In [9]: def Γ~i_j_k := withSymbols [m] g~i~m . Γ_m_j_k
In [10]: (Γ_1_#_#, Γ_2_#_#)
```

$$\left(\begin{pmatrix} 0 & 0 \\ 0 & (a\cos(\theta)+b)a\sin(\theta) \end{pmatrix}_{\#\#}, \begin{pmatrix} 0 & -(a\cos(\theta)+b)a\sin(\theta) \\ -(a\cos(\theta)+b)a\sin(\theta) & 0 \end{pmatrix}_{\#\#}\right)$$

```
In [11]: def R~i_j_k_l := withSymbols [m]
          ∂/∂ Γ~i_j_l x~k - ∂/∂ Γ~i_j_k x~l + Γ~m_j_l . Γ~i_m_k - Γ~m_j_k . Γ~i_m_l
In [12]: (R~1_2_#_#, R~2_1_#_#)
```

$$\left(\begin{pmatrix} 0 & \frac{(a\cos(\theta)+b)\cos(\theta)}{a} \\ \frac{-(a\cos(\theta)+b)\cos(\theta)}{a} & 0 \end{pmatrix}_{\#\#}, \begin{pmatrix} 0 & \frac{-a\cos(\theta)}{(a\cos(\theta)+b)} \\ \frac{a\cos(\theta)}{(a\cos(\theta)+b)} & 0 \end{pmatrix}_{\#\#}\right)$$

```
In [13]: def Ric_i_j := withSymbols [m] sum (contract R~m_i_m_j)
In [14]: Ric_#_#
```

$$\begin{pmatrix} \frac{a\cos(\theta)}{(a\cos(\theta)+b)} & 0 \\ 0 & \frac{(a\cos(\theta)+b)\cos(\theta)}{a} \end{pmatrix}_{\#\#}$$

```
In [15]: def scalarCurvature := withSymbols [i, j] g~i~j . Ric_i_j
In [16]: scalarCurvature
```

$$\frac{2\cos(\theta)}{(a\cos(\theta)+b)a}$$

図 13.2: トーラスのリーマン曲率テンソルの計算

ルの公式よりも簡潔な形の式を使って計算できる．上記の公式にあらわれる ω は接続形式と呼ばれ，微分形式において第二種クリストッフェル記号に対応するものである．

```
In [1]: def x := [| θ, φ |]

In [2]: def g_i_j := [| [| r^2, 0 |], [| 0, r^2 * (sin θ)^2 |] |]_i_j

In [3]: def g~i~j := [| [| 1 / r^2, 0 |], [| 0, 1 / (r^2 * (sin θ)^2) |] |]~i~j

In [4]: def Γ_i_j_k := (1 / 2) * (∂/∂ g_i_k x~j + ∂/∂ g_i_j x~k - ∂/∂ g_j_k x~i)

In [5]: def Γ~i_j_k := withSymbols [m] g~i~m . Γ_m_j_k

In [6]: def ω~i_j := Γ~i_j_#

In [7]: def d %t := !(flip ∂/∂) x t

In [8]: infixl expression 7 ∧

In [9]: def (∧) %x %y := x !. y

In [10]: def Ω~i_j := withSymbols [k]
           antisymmetrize (d ω~i_j + ω~i_k ∧ ω~k_j)

In [11]: Ω~1_2
```

$$\begin{pmatrix} 0 & \frac{sin(\theta)^2}{2} \\ \frac{-sin(\theta)^2}{2} & 0 \end{pmatrix}$$

```
In [12]: Ω~2_1
```

$$\begin{pmatrix} 0 & \frac{-1}{2} \\ \frac{1}{2} & 0 \end{pmatrix}$$

図 13.3: 曲率形式の計算

13.3.1　ウェッジ積・曲率形式

図 13.3 は S^2 の曲率形式を Egison で計算したプログラムである．[1]-[5] で球面座標系のパラメータと，リーマン計量，クリストッフェル記号を設定している．[6] で接続形式 ω を第二種クリストッフェル記号を使って定義している．[7] で外微分演算子を定義している．[8]-[9] でウェッジ積を定義している．ウェッジ積は中置演算子として定義されている．[10]-[12] で曲率形式を計算して出力している．

13.3.2　オイラー形式とオイラー数

オイラー類は，積分したらオイラー数が得られる特性類として有名な特性類である．以下のプログラムは，トーラス T^2 のオイラー形式を計算している．

```
def eulerForm := (1 / (2 * π)) * (Ω~1_2 - Ω~2_1)

eulerForm -- [| [| 0, (cos θ) / (* 2 π)) |], [| ((-1 * (cos θ)) / (* 2 π)), 0 |] |]
```

```
In [1]: def N := 2

In [2]: def x := [|r, θ|]

In [3]: def g_i_j := [| [| 1, 0 |], [| 0, r^2 |] |]_i_j

In [4]: def g~i~j := [| [| 1, 0 |], [| 0, 1 / r^2 |] |]~i~j

In [5]: def d %A := !(flip ∂/∂) x A

In [6]: def hodge %A :=
          let k := dfOrder A in
            withSymbols [i, j]
              (sqrt (abs (M.det g_#_#))) *
              (foldl (.) ((ε' N k)_(i_1)..._(i_N) . A..._(j_1)..._(j_k))
                         (map 1#g~(i_%1)~(j_%1) [1..k]))

In [7]: def δ %A :=
          let k := dfOrder A in
            -1^(N * (k + 1) + 1) * (hodge (d (hodge A)))

In [8]: def Δ %A :=
          match dfOrder A as integer with
            | #0 -> δ (d A)
            | #N -> d (δ A)
            | _  -> d (δ A) + δ (d A)

In [9]: def f := function (r, θ)

In [10]: Δ f
```

$$\frac{-\frac{\partial^2 f}{\partial \theta^2} - r\frac{\partial f}{\partial r} - r^2\frac{\partial^2 f}{\partial r^2}}{r^2}$$

図 13.4: 極座標のホッジ・ラプラシアンの計算

積分のための関数が Egison には実装されていないため，手計算でこれを積分した結果を記す．積分すると T^2 のオイラー数である 0 が得られる．

$$\chi(T^2) = \int_{T^2} d\theta d\phi \frac{\cos\theta}{2\pi} = \int_0^{2\pi} d\theta \cos\theta = [\sin\theta]_0^{2\pi} = \sin 2\pi - \sin 0 = 0$$

13.4　ホッジ・ラプラシアンの計算 - 微分形式を使った計算（2）

本節では，微分形式を使った計算をプログラムする例を紹介する．

図 13.4 は，極座標のラプラシアンをホッジ・ラプラシアンの公式 $\Delta = d\delta + \delta d$ を使って計算する Egison プログラムとその実行結果である．本節の前半は，このプログラムの解説をしながら進む．

13.4.1　外微分・ホッジ作用素

図 13.4 の前半 [1]-[4] は，極座標系のパラメーターと計量を設定している．その後，外微分・ホッジ作用素・外微分の随伴作用素という微分形式の基本的な演算子を [5]-[8] で定義している．[6] のホッジ作用素の定義には以下の公式を使っている．この定義のなかで使われている subrefs は，第二引数のシンボルのリストを，第一引数のテンソルの下添字として順番に付加する組み込み関数である．

$$*A = \sqrt{\det|g|} \cdot \epsilon_{i_1 \ldots i_n} \cdot A_{j_1 \ldots j_k} \cdot g^{i_1 j_1} \cdots g^{i_k j_k} \cdot e^{i_{k+1}} \wedge \ldots \wedge e^{i_n}$$

13.4.2　ホッジ・ラプラシアン

微分形式を使うと，ラプラシアンは $\Delta = d\delta + \delta d$ と座標系に依存しないように定義できる．この公式を使って表現されるラプラシアンはホッジ・ラプラシアンと呼ばれる．[8] においてこの公式が Egison で表現されている．[8] の定義において，0 形式と 2 形式の場合が特別扱いされている．これは，2 次元座標系における 2 形式にたいして d が，0 形式にたいして δ が，厳密には未定義であるためである．

[9] で定義した関数シンボルに，[10] でホッジ・ラプラシアンを適用している．13.2 節で解説したリーマン曲率テンソルと同様，ホッジ・ラプラシアンについても計量を変えるだけで，いろいろな座標系のラプラシアンを計算できる．

13.5　その他の微分形式についての演算子 - 内部積・リー微分

内部積やリー微分も，Egison で簡潔に定義できることを紹介しておく．これらの演算子は，たとえば，流体力学の質量保存の法則やナビエ・ストークス方程式の微分形式による表現をするときに使う．内部積は以下のように定義できる．

```
def ι %X %Y :=
  withSymbols [i] dfOrder Y * X...~i . (antisymmetrize Y..._i)
```

リー微分は以下のように定義できる．

```
def Lie %X %Y :=
  match dfOrder Y as integer with
    | #0 -> ι X (d Y)
    | #N -> d (ι X Y)
    | _  -> ι X (d Y) + d (ι X Y)
```

第 III 部

Egison の今後

第 14 章

Egison 開発の目標と方針

本章では，今後の Egison 開発についてどのようなことを考えているのか，「処理系の開発」・「アプリケーションの提示」・「Egison の先にある研究」と三つのテーマにわけて紹介する．

14.1　処理系の開発

Egison 処理系の開発については，

1. 独立したプログラミング言語としての Egison 処理系の開発
2. Egison の機能の他言語への移植

の 2 つを進めている．14.1.1 節で，Egison 本体の実装について，14.1.2 節で多言語への移植について述べる．

14.1.1　Egison 本体

Egison 本体には，Egison コミュニティで発明された言語機能をフルに実装されている．言語機能は豊富であるが，実行速度に関してはあまり気にせず実装されており，インタプリタ実装しかもたない．これは，新しい言語機能をスムーズに実装できるようにするためである．インタプリタは Haskell で実装されている．

独自の新機能の追求

プログラミング言語に今まで実装されたことがない新しい機能を Egison に実装し動いたときが，Egison 開発で一番楽しい瞬間である．新しい言語機能のアイデアは実際に現実的な問題について手を動かしてプログラムを書いているときに生まれる．ループ・パターンや，シーケンシャル・パターン，テンソルの対称性の宣言，関数シンボルなどの機能はそのようにして生まれた．今後もどんどん新しい言語機能を発案・実装していきたいと考えている．

静的型システムやコンパイラの実装

静的型システムやコンパイラのような既存の言語機能を Egison に組み込むことも重要な課題である．既存の言語機能を Egison に組み込むためには，新しい工夫が必要になることが多い．たとえば，静的型システムを Egison に組み込むには，パターンやマッチャーについて適切な型規則を設計する必要がある．

Egison に静的型システムを導入する研究は進んでおり，河田旺さんによる Typed Egison [2] と Formalized Egison [1] 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．Typed Egison は Egison version 3 をベースにしている．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．Egison 本体に静的型システムを導入するには，比較的大きな手間がかかるため，まだ着手されていない．Egison Version 5 の課題である．

Egison パターンマッチをコンパイル手法についても研究は進んでいる．2020 年 3 月に小川広水さんによって Egison パターンマッチをバックトラッキング・モナドを使った Haskell プログラムに変換する手法が発案され，Sweet Egison[4] という Haskell ライブラリとして実装された．ただ，Egison 本体については，新機能をすばやく実装できる proof of concept のプログラミング言語ということもあり，コンパイラを実装することはまだ先になりそうである．

■Egison Version 4 のリリース　2020 年 4 月にリリースした Egison version 4 から，多くのユーザーにとって馴染みやすいように文法を一新した．Egison Version 4 の新しい文法は中置記法（:=，++，::，+など）をサポートしたおかげで，必要な括弧の数が種類も減っており，視認性が高まっている．以下は，素数の無限リストから双子素数を抽出するパターンマッチを記述したプログラムを Egison version 3 と Egison version 4 で書きくらべたものである．

Listing 14.1: Egison version 3

```
(define $twin-primes
  (match-all primes (list integer)
    [<join _ <cons $p <cons ,(+ p 2) _>>> [p (+ p 2)]]))

(take 5 twin-primes) ; => {[3 5] [5 7] [11 13] [17 19] [29 31]}
```

Listing 14.2: Egison version 4

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

take 5 twinPrimes -- => [(3, 5), (5, 7), (11, 13), (17, 19), (29, 31)]
```

新文法の親しみやすさは，特に数式を書いたときに発揮される．リーマン曲率テンソルの公式という微分幾何で有名な公式を Egison version 3 と Egison version 4 で書きくらべてみる．リーマン曲率テンソルの公式は以下のような数式では表現される．

$$R^{i}{}_{jkl} = \frac{\partial \Gamma^{i}{}_{jl}}{\partial x^{k}} - \frac{\partial \Gamma^{i}{}_{jk}}{\partial x^{l}} + \Gamma^{m}_{jl}\Gamma^{i}{}_{mk} - \Gamma^{m}_{jk}\Gamma^{i}{}_{ml}$$

数式の記述には中置記法が多用されるため，中置記法をサポートした新文法のほうが数式に近い記述になる．以下は，実際にリーマン曲率テンソルの公式を Egison version 3 と Egison version 4 で書きくらべたものである．

Listing 14.3: Egison version 3

```
(define $R~i_j_k_l
  (with-symbols {m}
    (+ (- (∂/∂ Γ~i_j_l x~k) (∂/∂ Γ~i_j_k x~l))
       (- (. Γ~m_j_l Γ~i_m_k) (. Γ~m_j_k Γ~i_m_l)))))
```

Listing 14.4: Egison version 4

```
def R~i_j_k_l := withSymbols [m]
  ∂/∂ Γ~i_j_l x~k - ∂/∂ Γ~i_j_k x~l + Γ~m_j_l . Γ~i_m_k - Γ~m_j_k . Γ~i_m_l
```

14.1.2 他のプログラミングへの Egison 言語機能の移植

2019 年からは既存のメジャーなプログラミング言語に Egison の機能を実装することもはじまった．Egison の言語機能を既存プログラミング言語へ移植することには，

1. 移植先のプログラミング言語のユーザーが気軽に Egison の機能を試すことができるようになる．
2. 移植先のプログラミング言語の処理系で実行されるために，ナイーブな実装の Egison 本体よりも，プログラムの実行速度が格段に高速になる．

というメリットがある．ただし，14.1.1 で述べたとおり，既存言語に Egison の言語機能を移植するためには，既存言語の他の機能との兼ね合わせを考慮する必要があるために，すべての機能が移植できないことがある．たとえば，現在のところ，ループ・パターンの移植はまだできていない．また，既存言語の仕様や実装に詳しくなる必要があるために，これらのメリットを享受するためには比較的大きな開発コストがかかるというデメリットもある．そのため，Egison 本体も並行して開発されている．

Egison パターンマッチの Scheme ライブラリ実装（実装済み）

2018 年 12 月に Scheme という言語自体がシンプルでかつ強力なメタプログラミング機能（言語を拡張するための言語機能）をもつ関数型プログラミング言語に Egison のパターンマッチを提供するライブラリを実装した．言語の拡張機能（メタプログラミングのための機能）が強力でかつ言語仕様が簡潔な言語をほど，言語機能の移植が進めやすい．そのため，最初の移植先言語として，Scheme が選ばれた．このライブラリの実装については論文を執筆して Scheme Workshop 2019 で発表した．このライブラリ実装は Scheme 処理系によってコンパイルされるために，Egison のパターンマッチを使ったプログラムを，Egison 処理系よりも 50 倍近く高速に実行できる．Scheme 処理系には Gauche を選んだ．Deep embedding と呼ばれる手法を用いて，このライブラリは実装されている．

Egison パターンマッチの Haskell ライブラリ実装その 1（miniEgison として実装済み）

上記の Egison パターンマッチの Scheme 移植が完成した直後から，同様の手法で Egison パターンマッチを Haskell に移植するための Haskell ライブラリの開発を開始した．Scheme が動的型付けプログラミング言語であるのにしたいして，Haskell は静的型システムをもつ言語であり，型システムを意識した移植の手法を考える必要があることが実装の難所だった．この実装は 2019 年 9 月頃に完成した．

Egison パターンマッチの Haskell ライブラリ実装その 2（Sweet Egison として実装済み）

miniEgison には，miniEgison を使って書いたプログラムと同等のプログラムを Haskell で通常の関数型プログラミングスタイルで書いたプログラムをくらべたときに miniEgison によるプログラムのほうが数倍遅いという問題があった．この問題を解決するために，shallow embedding という手法で，Sweet Egison という Haskell ライブラリを 2020 年 3 月から新しく実装しなおした．Sweet Egison は，Egison パターンマッチを使って記述されたプログラムを，同等の意味の関数型プログラミングスタイルで記述されたプログラムに変換して実行する．そのため，Egison パターンマッチを用いた簡潔なプログラムの記述と通常の Haskell プログラムと同等の実行速度を併せ持つ．

Egison パターンマッチを GHC 拡張として実装（未実装）

ライブラリにより DSL として言語機能を実装すると，冗長な記述をしないといけない箇所がでてくることが多い，Egison パターンマッチを実装した Haskell ライブラリである miniEgison や Sweet Egison にもこの問題があり，マッチ節やマッチャー定義の記述に少し冗長な記述が必要になる箇所がある．この問題を解決するには，Haskell の標準的なコンパイラである GHC を直接編集して組み込みの構文として言語機能を実装すればよい．今後，GHC を直接拡張して Egison のパターンマッチを実装する試みをする予定である．ただ実装するだけでなく，GHC に取り込まれ

るためには GHC コミュニティとコミュニケーションをとる必要がある．GHC に新機能を提案するための仕組みとして GHC Proposal というものがある．ある程度実装がすすんだらドキュメントをまとめてそこで提案したい．

システム・プログラミング言語 Rust（未実装）

ユーザーがメモリの管理を操作できるプログラミング言語への Egison パターンマッチの実装は面白い課題である．現在のところ，以下のような疑問をもっている．

- メモリを効率的に操作することにより，効率的なパターンマッチ機能が実装できるか？
- メモリを効率的に操作するために，パターンマッチ機能を制限する必要があるか？
- パターンマッチ機能を制限したとして応用はあるか？

メタプログラミング機能が豊富といわれる Rust への実装を考えている．

14.2　アプリケーションの提示・開発

Egison の提供する言語機能のアプリケーションは，多くのプログラマにとって自明ではない．Egison の応用例を提示・開発することは，世のプログラマに Egison が実用に役に立つことに納得してもらい，さらに新しい応用例を考えてもらうために重要である．

14.2.1　アルゴリズムの実装（SAT ソルバー）

さまざまなアルゴリズムを Egison で実装してみることは，Egison の機能が役に立つ場面が明確になったり，新しい機能の発明につながることが多い．第 4 章で SAT ソルバーのアルゴリズムの 1 つである Davis-Putnam アルゴリズムの実装を紹介したが，より複雑であるが効率的な CDCL アルゴリズムも Egison で実装している．より効率的なアルゴリズムを実装しようとするとメモリの使い方まで管理したくなり，14.1.2 節で言及したように，Rust のようなシステム・プログラミング言語で Egison のパターンマッチを使いたいニーズも生まれる．

14.2.2　数式処理システム

本書の第 II 部で紹介した数式処理システムは，Egison パターンマッチの重要な応用である．数式に対するパターンマッチ・エンジンが簡潔に実装できるために，数式処理システムを少ない手間で実装することができた．そのために，数式処理システムの拡張も簡単で，テンソルの添字記法のサポートや関数シンボルなど新しい機能の実装を簡単に実験できた．

14.2.3　Egison のパターンマッチを応用した定理証明記述言語

将来はコンピュータ上で数学の証明は記述し検査されるようになる．現在これがされていないのは，コンピュータ上で数学の証明を紙の上のように簡潔に表現できないためである．Egison はプログラミング言語としてアルゴリズムの表現を簡潔にする機能を開発したが，同様に証明の記述を簡潔にすることもできるはずである．たとえば，Egison のパターンマッチは証明のための条件分岐を記述するために役に立つはずである．そのような考えをもとに，Egison のパターンマッチをもつ証明支援システムを設計している [6].

14.2.4　クエリ言語（パターンマッチの応用）

Egison のパターンマッチはさまざまな種類のデータベースのクエリ言語として使うことができる．例として，ソーシャルネットワークを管理するデータベースに対して，ユーザー名"Egison_Lang"のユーザからフォローされているが，このユーザをフォローし返していないユーザ一覧を取得するクエリを考える．このクエリは，matchAll式を使って記述できる．リレーショナル・データベースのテーブルを集合として，パターンマッチしている．

```
1 matchAll (users, follows, users) as (set user, set follow, set user) with
2   ((and (Name #"Egison_Lang") (ID $uid)) : _,
3    (and (FromID #uid) (ToID $fid)) : !((and (FromID #fid) (ToID #uid)) : _),
4    (and (ID #fid) (Name $fname)) : _) -> (fid, fname)
```

この matchAll式は，ユーザテーブル（users）と，フォローテーブル（follows），ユーザテーブルのタプルに対してパターンマッチする．それぞれのテーブルは集合としてパターンマッチされる．2 行目はユーザテーブルに対するパターンマッチを記述している．Nameと IDは，それぞれのレコードのフィールドの値を取得するためのパターン・コンストラクタである．2 行目のユーザテーブルに対するパターンは，ユーザ名が"Egison_Lang"であるユーザの ID をパターン変数$uidに束縛する．3 行目はフォローテーブルに対するパターンマッチを記述している．FromIDと ToIDは，フォロワーとフォロイーを取得するためのパターン・コンストラクタである．FromIDのユーザが ToIDのユーザをフォローしている．ユーザ IDuidのユーザをフォロー返ししていないユーザは not パターンを使ってパターンマッチされている．4 行目はユーザ IDfidのユーザの名前を取得するために，ユーザテーブルのパターンマッチをしている．結果として，タプル (fid, fname)を返している．

Egison パターンマッチにはクエリの記述が簡潔であるという利点もある．たとえば，上記と同等のクエリを SQL で記述すると複雑になる．非線形パターンのかわりに WHERE節のなかに条件を記述し，not パターンのかわりにサブクエリを使うためである．パターンマッチ指向によるクエリの記述は左から右に一度で読めるが，SQL のクエリはそのように読むことができない．

```
1 SELECT DISTINCT ON (user.name) user.name
```

```
2  FROM user AS user1, follow AS follow1, user AS user2
3  WHERE user1.name = 'Egison_Lang' AND follow1.from_id = user1.id AND user2.id = follow1.
     to_id
4    AND NOT EXISTS
5      (SELECT '' FROM follow AS follow2
6         WHERE follow2.from_id = follow1.to_id AND follow2.to_id = user1.id)
```

14.2.5 機械学習（テンソル添字記法の応用）

テンソルを扱う機械学習のアルゴリズムは，添字記法を使って簡潔に表現できる．その例として，Egison によるニューラル・ネットワークの実装を考えている．ニューラル・ネットワークの実装には下記のような数式が現れる．x_i は前の層から現在の層への入力，y_j は現在の層から次の層への出力，w_j^i は重み，b_j はバイアスを表す．

$$y_j = w_j^i x_i + b_j$$

1 階のテンソルであるベクトルや 2 階のテンソルである行列までしか登場しないニューラル・ネットワークのアルゴリズムの場合，微分幾何の計算にくらべると添字記法の恩恵は薄れるが，行列の転置のミスなどがなくなるため，添字記法は役に立つ．ニューラル・ネットワークのアルゴリズムには，ステンシル計算や，平均プーリングやマックス・プーリングといったプーリング処理のような数学のテンソル計算では現れない処理が必要になるため，このような処理を表現する方法について研究する必要がある．

14.3 Egison の先にある研究

Egison を開発している間に考える始めるようになった解決することがむずかしそうな大きな未解決問題を紹介する．

14.3.1 二次元以上のデータの表現

改行やインデントを使って，二次元的にプログラムを視覚化することができるが，基本的にプログラムは一次元の文字列データである．プログラムは構文木により内部的にはツリー構造によって表現される．プログラムを表現する文字列は，このツリー構造を深さ優先の順番で並べたものである．そのため，ツリーの形に落とし込めないデータやプログラムの構造は，きれいにプログラムとして表現できない．

ツリーの形に落とし込めないデータには，たとえばグラフがある．グラフを一次元の文字列に落とし込むと，二次元の紙面の上に図示したグラフとくらべて，どうしても読みにくいものになる．その他にもオセロ盤や碁盤も一次元の文字列に落とすのがむずかしいデータである．オセロの挟むという概念や，碁の囲むという概念は，直感的には簡単な概念であるのにプログラム上で表

現するのはむずかしい．

ツリーの形にきれいに落とし込めないプログラムの構造には，複数の引数をとって複数の結果を返す関数の適用がある．一般的なプログラムでは，たいていの関数は複数の引数をとるが一つの返り値を返す．そのため，$f(g(x), h(y))$ のように，ある関数の結果を別の関数の結果にわたすような関数適用のネストがきれいに書ける．一つの引数をとり，複数の結果を返す関数の適用もツリーの形にきれいに落とし込める．実はこの構文がパターンマッチである．パターン`$x :: $y :: _`は一つの引数をとって二つの返り値を返す関数`::`の適用をネストしたものと考えることができる．複数の引数をとって複数の結果を返す関数のネストした適用は，これらのようにツリーの形にきれいに落とし込めない．

複数の引数をとって複数の結果を返す関数が頻出する分野に量子アルゴリズムがある．量子アルゴリズムの記述のために，量子回路が記述されることが多いが，基本的に量子回路への q ビットの入力と出力は同数である．量子アルゴリズムの紹介には，量子回路の図が付随することが多いが，この図のほうがプログラムより理解しやすい．

これらの問題を解決するには，ツリー構造以外を使って内部的にプログラムを表現する，文字列以外を使ってプログラムを記述するなどといった根本的な変化が必要であるようにみえる．プログラムを二次元の図として記述する手法には，ヴィジュアル・プログラミングという分野の手法が知られている．このような視点で新しいプログラミング言語を考えることを面白そうに感じている．

14.3.2 数以外のユーザー定義データ型もシンボリックに処理できる数式処理システム

Egison を含む多くの数式処理システムは，数の計算をシンボリックに扱うことができる．しかし，シンボリックに扱いたい対象は数だけではない．リストや，ツリー，グラフなどもっとさまざまなユーザ定義データ型をシンボリックに扱いたいこともある．Egison 上の数式処理システムは数のシンボリックな扱いが組み込みで実装されており，任意のデータ型にシンボリックな計算を拡できるようになっていない．そこで，シンボリックな計算が柔軟にできるもっと一般的な数式処理システムを作れたら面白いと感じている．

14.3.3 複数のプログラミング・スタイルの表現力の差をくらべるための理論

ことなるプログラミング・スタイルで記述された複数のプログラムの読みやすさ・書きやすさを客観的にくらべる手法は確立されていない．このような手法がないことは，新しい記法の開発があまり活発でない原因の一つであると考えられる．また，もしこのような手法があれば Egison プログラムの読みやすさ・書きやすさを客観的に示すことができ，Egison を広めやすくなると考えている．そのようなわけで，あるプログラムの読みやすさ・書きやすさを測る手法について考察している．本節では，この問題に関する現時点の考えを書く．

プログラムの読みやすさ・書きやすさとは，プログラムの理解のしやすさと同じである．プログラムの理解のしやすさを定義するには，プログラムを理解するとはどういうことか定義する必要がある．プログラムを理解するとはどういうことか定義できれば，あとはそれにかかる計算量がプログラムの理解のしやすさということになる．

プログラムの理解には無数の段階がある．たとえば，マージソートを実装したプログラムについて，

- リストを引数にとってリストを返すプログラム
- 順序に沿って並べ直したリストを返すプログラム
- リストの長さを n としたとき，時間計算量 $n\log(n)$ で結果を返すプログラム

などいろいろな視点から解析し理解できる．これらのプログラムの性質は，プログラムの型として表現でき，型が合っているか検査したり，型を推論できる場合があることが知られている．

そこで，プログラムの理解のしやすさを計測するために型検査に必要なステップ数を使えるのではと考えている．新しいプログラミング・スタイルの優位性を示すには，そのプログラミング・スタイルによってある特定のプログラムの性質が示しやすくなることを示す．実際，計算量のような複雑なプログラムの性質については，Egison のパターンマッチを用いたプログラムは，通常の関数型のプログラムよりも，少ない手間で検査・推論できそうな直感がある．

付録 A

パターンマッチ指向プログラミング問題集

パターンマッチ指向プログラミングを体験していただくために，クイズ形式で問題を用意した．

member 関数

下記の空白をうめて member関数を完成させよ．

```
1  def member x xs :=
2    match xs as list eq with
3    | [          1          ] -> True
4    | _ -> False
5
6  member 1 [1,2,3,4]
7  -- True
8
9  member 5 [1,2,3,4]
10 -- False
```

concat 関数

下記の空白をうめて concat関数を完成させよ．

```
1  def concat xss :=
2    matchAllDFS xss as [          1          ] with
3    | [          2          ] -> [          3          ]
4
5  concat [[1,2],[3],[4,5]]
6  -- [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．

```
1 take 4 (matchAll primes as list integer with
2         | [          1          ] -> (p, p + 2, p + 6, p + 8))
3 -- [(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．

```
1 take 6 (matchAll primes as list integer with
2         | [          1          ] -> (p, p + 6))
3 -- [(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．

```
1 def intersect xs ys :=
2   matchAllDFS (xs, ys) as [          1          ] with
3   | [          2          ] -> [          3          ]
4
5 intersect [1,2,3,4] [2,4,5]
6 -- [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．

```
1 def difference xs ys :=
2   matchAllDFS (xs, ys) as [          1          ] with
3   | [          2          ] -> [          3          ]
4
5 difference [1,2,3,4] [2,4,5]
6 -- [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.

```
1 def doubles xs :=
2   matchAllDFS xs as [     1     ] with
3   | [     2     ] -> [     3     ]
4 
5 doubles [1,2,3,2,1,1,4,5,3]
6 -- [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.

```
1 def tails xs :=
2   matchAll xs as list something with
3   | [     1     ]
4   -> ts
5 
6 tails [1,2,3,4]
7 -- [[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.

```
1 def equalMultiset xs ys :=
2   match (xs, ys) as (list eq, multiset eq) with
3   | ([], [])
4   -> True
5   | [     1     ]
6   -> equalMultiset xs' ys'
7   | _
8   -> False
9 
10 equalMultiset [1,1,2] [2,1,1]
11 -- True
12 
13 equalMultiset [1,1,2] [1,2,2]
```

```
14 -- 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.

```
1 def equalMultiset xs ys :=
2   match (xs, ys) as (list eq, multiset eq) with
3   | [          2          ]
4   -> True
5   | _
6   -> False
```

ジョーカーの存在を考慮するポーカーの役判定

3 章 3.2.2 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.

```
1 def card := matcher
2   | card $ $ as (suit, mod 13) with
3     | Card $s $n -> [(s, n)]
4     | Joker -> [          1          ]
5   | $ as something with
6     | $tgt -> [tgt]
7
8 poker [Card Spade 5, Card Spade 6, Joker, Card Spade 8, Card Spade 9]
9 -- "Straight flush"
10 poker [Card Spade 5, Card Diamond 5, Joker, Card Club 5, Card Heart 7]
11 -- "Four of a kind"
```

付録 B

数式処理システムとしての Egison の演習問題

本付録の問題にあるプログラムをいくつか書いてみることによって数式処理システムとしての Egison を使いこなせるようになる．プログラムを書くむずかしさよりも，数学の知識を求められる問題が多いがぜひ挑戦してみてほしい．

三次方程式と四次方程式の解

9.4 節の内容を発展させて，三次方程式と四次方程式の解を求めるプログラムを書け．三次方程式と四次方程式の解を求めるアルゴリズムには，それぞれカルダノの方法，フェラーリの方法が知られている．

1 の n 乗根

1 の n 乗根は代数的に求めることができることが知られている．いくつかの n について 1 の n 乗根を計算するプログラムを書いてみよ．1 の 5 乗根，7 乗根，9 乗根，17 乗根については Egison レポジトリにサンプルコードが用意されている．

リーマン曲率の計算

13.2 節のプログラムを参考にして，円柱のリーマン曲率，S^3 のリーマン曲率を計算するプログラムを書け．

球座標系のホッジ・ラプラシアン

13.2 節の極座標のホッジ・ラプラシアンを計算するプログラムを参考にして，球座標系のホッジ・ラプラシアンを計算するプログラムを書け.

参考文献

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

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

[3] egison/backtracking: Backtracking monad in Haskell. https://github.com/egison/backtracking, 2020. [Online; accessed 2021-05-06].

[4] egison/sweet-egison: Haskell library for non-deterministic pattern matching. https://github.com/egison/sweet-egison, 2020. [Online; accessed 2021-05-22].

[5] 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.

[6] S. Egi. Pattern-match-oriented proof writing language. In *Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming*, pages 223–224, 2020.

[7] 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.

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

[9] J. M. Spivey. Algebras for combinatorial search. *Journal of Functional Programming*, 19(3-4):469–487, 2009.

索引

#, 10
$, 80
++, 9
--, 9
., 81
::, 9
?, 10
%, 80
~_, 82

algebraicDataMatcher, 18
and パターン, 13
as, 9

contract, 82

matchAll, 8
matchAllDFS, 12

not パターン, 13

or パターン, 13

primes, 10

with, 9

値パターン, 10
関数シンボル, 93
原始データパターン, 42
原始パターンパターン, 42
コレクション, 11
コンス・パターン, 9
シーケンシャル・パターン, 16
述語パターン, 10
ジョイン・パターン, 9
スカラー仮引数, 80
スカラー関数, 80
添字付き変数, 14
タプル・パターン, 17
ダミーシンボル, 87
テンソル仮引数, 80
テンソル関数, 80
反転スカラー仮引数, 86
パターン関数, 17
パターンのアドホック多相性, 11
パターン・フュージョン, 45
パターンマッチ指向, 3
変数パターン, 17
マッチャー, 9
無名パラメーター関数, 48
ループ・パターン, 14
ワイルドカード最適化, 45