聖徳大学研究紀要　聖徳大学　第28号　聖徳大学短期大学部　第50号　97-104（2017）

もう１つの関数プログラミングの方法

－ 思考法からプログラムへのアプローチ方法 －

山下 伸夫

How to think and how to write programs in functional programming

― An attempt to approach programs from thinking functionally ―

YAMASHITA, Nobuo

概要

型付λ計算を計算モデルとする関数プログラミングの思考法と，その思考のプログラムとして記述は型シグネチャを中心に展開できることを示す。関数プログラミングの基本的なアイデアは「プログラムは関数であり，計算は関数適用の値を求める」ことであって，その記述については「正しく構成された式は型付けされた値を表示する」という単純な原則に基づくものである。また，その動機は数学的思考における組織的で構造的な抽象化と汎用的な具象化の方法を手にすることである。本稿では具体的なプログラミングの課題を解決する思考プロセスをプログラムとして数学的記法を使って記述し，それを観察することによって，有用なプログラミングの概念を再発見，理解，利用できることを示す。特に型式（type expression）を分析し，埋め込みの特定領域言語を設計する関数プログラミングの思考法のエッセンスを抽出し記述できることを示す。

キーワード

宣言的プログラミング，関数プログラミング，関数プログラミング言語

Abstract

In this article, we illustrate how to think and how to describe programs in functional programming. Since problems expected to be solved by computer grow increasingly complex, we are always looking for ways to figure out structures of such problems. In functional programming, we have mathematical means of abstraction and instantiation. It is important for us to find useful concepts by writing our way of thinking as a program. So, we demonstrate importance of type analyzing of function types for building programs by function compositions and for designing DSL（domain specific language）.

Key words

declarative programming, functional programming, functional programming language

1　はじめに

本稿の目的は，関数プログラミングの経験のない読者が，型付λ計算を計算モデルとする関数プログラミングの思考の方法とその記述の方法とを理解することである。すなわち，関数の値（function value）および関数の型（function type）を中心に思考する方法を理解し，プログラムを構成する感覚を得ることである。

関数プログラミングに対する関心の高まりとともに，特定のプログラミング言語による関数プログラミングについてのテキストが増えている。そのようなテキストではプログラミング言語の関数プログラミングに関わる概念や機能が繰り返し説明されているので，読めば，「参照透明」「静的型付け」「高階関数」などプログラム記述言語を特徴付ける概念や機能の概要について説明できるようになるであろう。また，典型的な「高階関数」がどのような計算するかについても説明できるはずである。しかし，言語機能の特徴や長所を知っていても，関数プログラミングにおいて何をどのように考えるか，あるいは，その思考をどのように表現するかについて，よく判らないという感想を多く耳にする[1]。一部の有用な高階関数の API を知っていても，解決すべき個々の問題に対してどの関数をどう使うのか，あるいは自分で用意する場合，なにをどのように定義するかが判らず，プログラムが書けないという場面をよく見かける。

原因の１つは，関数プログラミングに関するテキストのほとんどが，読者がプログラミング経験を持つことを前提とし，その経験との対比によって関数プログラミングを説明することにある。前提となる現在主流のプログラミング技法と関数プログラミングの技法では，問題の捉え方そのものが大きく異なる。それゆえ，プログラマにとって既知の記述言語で記述したものとの対比を介して関数プログラミングにおける記述を見ても，差異は何であるかは理解できるが，いざプログラミングとなる

聖徳大学短期大学部総合文化学科・准教授

と，どこからスタートして，どのように思考を進めればよいかが判らないという事態が起きていると考えられる。

もちろん，対比で浮びあがるプログラム記述言語を特徴づける概念は多い。しかし，一方では，関数プログラミングで考えるときにはそのようには考えないということであったり，他方では，関数プログラミングで考えるときは自明であるようなものからの帰結であったりする。したがって，問題をどのように捉え，どのように解決し，それを記述するかという思考を理解し，プログラミングを実際に行うためには，対比は手掛かりになり難い。

では，直截に関数プログラミングをするとして，何を手掛りにすべきか。関数の型(function type)とその記述である型シグネチャ(type signature)であろうと筆者は考えている。関数プログラミングの動機は，数学的思考における組織的で構造的な抽象化と汎用的な具象化の方法を手にすることである。しかし，関数プログラミングに高度な数学が必要なわけではなく，基本的な論証をたどる論理力があればよい。関数プログラミングの基本的なアイデアは「プログラムは関数であり，計算は関数適用式で指定された値を求めること」という単純なものであり，その記述については「正しく構成された式は型付けされた値を表示する」という単純な原則に基づくだけだからである。

本稿では，命令プログラミングとの対比を際立たせる，関数プログラミング用の記述言語に特有の概念や用語の詳しい説明はせずとも関数プログラミングの思考法を直截に考察することにより，プログラムを構成していくことができることを示す。

2　関数プログラミング

プログラミングの技法は以下の２つに大別できる。

命令的方法　どのように計算すべきを示す手順を構成する。
宣言的方法　なにを計算すべきかという宣言を構成する。

実行効率が良い計算プロセスを実現するには，可能な限り計算機資源を効率的に使うように命令列を構成する必要がある。しかし，複雑で規模の大きな問題の解を正しいプログラムとして書くためには，なにを計算するかの論理構造を捕捉する必要がある。すなわち，論理構造を高水準の概念を使って簡潔に構成する必要がある。そのためには構成がより抽象的になる宣言的な方法によるアプローチが必要である。

関数プログラミングの基本的動機は，プログラミングを計算機ハードウェアの論理構成とは独立した，数学的な活動として行いたいということである[2]。関数は数学の対象であるから関数プログラミングではプログラマが行う活動を数学的な思考として行えるということであり，数学的思考における組織的で構造的な抽象化と汎用的な具象化の方法が手に入るということでもある。

数学的な抽象化と具象化の手法は，簡潔で，把握しやすく，モジュラリティが高く再利用しやすいコードを産む。これが，期待される恩恵につながるのである。

3　関数プログラムの記述

一般に高水準のプログラム記述言語には，以下の３つの仕組みがある[3]。

- **基本式**　言語が扱うもっとも単純な対象を表す
- **組み合わせ方法**　より単純な対象から複雑な対象に合成する
- **抽象の方法**　合成された複雑な対象に名前を付けて，１つの対象として扱う。

関数プログラミング言語は，関数プログラミングの思考を支援するプログラム記述言語であるから，少くとも，関数の定義を記述し，計算すべき式を記述する方法を与えるものでなければならない。

以降，プログラム記述は Haskell [4]の記法を用いる。記述言語として Haskell を使うのは，関数プログラミングの思考をそのまま記述できる表現力があるからであり，型式(type expression)の構文や関数定義構文が数学的記法と近いからである。

プログラム記述例は，関数プログラミングの思考法をプログラムとして数学的に記述する例を示すもので，何をどのように記述できるべきものかを示すためのものである。架空の言語を用いた疑似コードで示すこともできるのであるが，計算機上で計算を実行できなければリアリティがないので疑似コードは採用しない。

3.1　型

値を求めるのが関数プログラミングにおける目標であるが，その値を特徴づけるものとして，型(type)を導入すれば，関数の性質をより扱いやすくする。型は値の集合と考えられる。

型 τ は以下の帰納的定義により構成する。

$$\begin{aligned} \tau \quad ::= \quad & b \\ \mid \quad & T\ \tau_1 \cdots \tau_n \\ \mid \quad & \tau \rightarrow \tau \end{aligned}$$

１つめの生成規則において b は，整数の型(`Int` など)，浮動小数点の型(`Float` など)，論理値の型(`Bool`)，文字の型(`Char`)などの基底型である。２つめの生成規則において T は，型を１つ以上パラメータとする型構成子(type constructor)で，たとえば，(,) `Int Bool`（通常は(`Int`, `Bool`) と書くことが多い）は，(,)が型構成子，１つめの型パラメータが `Int` で，２つめの型パラメータが `Bool` であるような対(つい)の型である。また，[] `Char`（通常は［`Char`］と書くことが多い）は，[]が型構成子，

型パラメータが Char である。これは文字のリストすなわち，文字列型である。３つめの生成規則において → は関数を表し，その左側が関数の始域を表す型，右側が関数の終域を表す型である。

3.1.1 関数値（function value)と関数型（function type)

関数プログラミングで，もっとも重要なのは関数の値であり，その型である。型やその記述方法を考えることで，重要な考え方や記述言語にあると便利な性質が見えてくる。

数学では，関数 f はある集合 A の各要素に集合 B の要素を１つだけに対応させる１つの規則である。このとき集合 A を関数 f の始域(domain)，集合 B を終域(codomain)という。プログラミングでは，集合を型に，要素を値に読み替え，プログラムでは数学風の記法で

```
f :: A -> B
```

と書く。また，この記述のことを関数 f の型シグネチャという。

型の情報とその記述方法はプログラミングにおいては重要である。x が A 型の値を表していれば，関数 f を x に適用した結果の値は，式 f x で表し(数学では括弧を使って $f(x)$ と書くのが普通であるが，$sin\ x$ のように不要な括弧は書かないのと同じ記法を使う)，適用の結果として得られる値の型は B である。x が表している値の型が A ではなかったら，関数 f を x に適用することはできない。もし，プログラム中にそのような関数適用が記述されていれば，これはプログラムの誤りである。このようなプログラムの誤りを言語系が実行前の翻訳段階で見つけること(型検査)ができれば，予期せぬ動作するプログラムの実行を未然に防ぐことが可能である。

型の記述には関数の抽象的な仕様が含まれていると考えてよい。すなわち，型記述には含まれるプログラマの意図と，実装記述との整合性は，言語システムによってチェックされるべきである。

型シグネチャの::の右辺は，型式(type expression)である。正しく形成されている１つの型式は１つの型を表している。また，A が１つの型を表す型式であり，B が１つの型を表す型式であれば，A -> B という型式は１つの関数型(function type)を表している。関数の型があるということは，その要素である関数の値があるということになる。すなわち，関数は値として他のデータ型の値と同じように扱えることを示している。このことを以って，関数が第一級の対象であるという。

また，型の構成規則の３つめ，関数の型の構成規則において始域の型(→ の左側)や終域の型(→ の右側)が関数の型になりうる。すなわち，(A -> B) -> C，A -> (B -> C)，(A -> B) -> (C -> D) という型の関数が考えられるということである。このような始域が関数の型であったり，終域が関数である関数を高階関数という。関数プログラミングでの関心の中心は関数の組み立てであるから，特に終域が関数であるような高階関数，すなわち，関数を生成する関数が重要である。

型式において，-> は型上の二項演算子と見なせる。これを関数型構成演算子と呼ぶことにする。１つの型式において複数の関数型構成演算子があるときの曖昧性を避けるために括弧を使うが，関数型構成演算子に結合性を付与すると括弧を減らすことができる。終域が関数型である関数が重要であるとすれば，関数型構成演算子は右結合であるとするのが合理的である。たとえば，A -> (B -> C) は A -> B -> C と書いてよく，A -> B -> C とあれば A -> (B -> C) と解釈できる。

これと類似した議論は，関数適用の表記についてもできる。g :: A -> (B -> C) という型シグネチャをもつ関数 g があって，x が A 型の値を表し，Y が B 型の値を表すとしよう。このとき関数適用 g x が表す値の型は B -> C という関数型になる。したがって，g x は関数であるから y に適用でき，その適用結果の値は式 (g x) y で表される。このとき，関数適用は左結合であるとするのが合理的である。たとえば，(g x) yは g x y と書いてよく，g x y とあれば (g x) y と解釈できる。

3.2 関数定義の記述

関数定義は以下のように記述する。

```
stdBMI :: Float
stdBMI = 22.0

stdWeight :: Float -> Float
stdWeight = \ h -> stdBMI * h
```

上のスクリプトは，変数 stdBMI を浮動小数点数型 Float の値 22.0 に束縛し，変数 stdWeight を Float -> Float 型の関数の値 \ h -> stdBMI * h に束縛している。関数リテラル記法 \ x -> e は，λ 抽象式 $\lambda x.e$ を表す。関数の定義は，関数適用後の値を表す式を使って定義することもできる。

```
stdWeight :: Float -> Float
stdWeight h = stdBMI * h
```

さらに，stdWeight は stdBMI の値を左から掛ける関数なので，直截に

```
stdWeight :: Float -> Float
stdWeight = (stdBMI *)
```

と書ける。

４　関数プログラミングの実際

プログラミング技法やプログラミング言語を特徴づける概念の説明だけでは，実際にプログラミングを行う感覚，思考の進め方，あるいはもとっと感覚的いえば，関数プログラミングの「こころ」を伝えるのは難しい。ここでは，実際のプログラミングに沿って，関数プログラミングでは，どのように考え，それをどのように記述するかを説明する。

■　プログラミング課題　整数の四則演算式を計算する。

4.1　main の構成

このプログラムは，ユーザが整数の四則演算式を入力し，計算機がその結果を表示するというやりとりを繰り返すセッションを提供する。セッションはユーザ入力，計算機出力が交互に繰り返されるものと考えられる。そうすると，セッションへの入力文字列は，改行文字で終端した複数の入力文字列が１つに連結された文字列であり，セッションが構成する出力文字列は，改行文字で終端した複数の出力文字列を１つに連結した文字列である。

この入力文字列から出力文字列への変換を行う関数calculator :: String -> Stringを定義したとして，ユーザ入力は実行時に与られるものであるから，予めプログラムの中にcalculator［入力データ］のような適用式を書くことはできない。プログラム実行時に入力ストリームから入力文字列を読み込み，それを関数calculatorに渡たし，計算結果の文字列を出力ストリームに書き込む仕組みが提供されていなければならない。これを提供する関数を interact とすると，その型は (String -> String) -> IO () となるはずである。すなわち，interactは文字列変換関数から入出力アクションへの変換関数である。入出力アクションとは，実行時に入出力を行う

```
main :: IO ()
main = interact calculator
```

4.2　calculator の構成

文字列を改行文字で分解する関数 lines :: String -> [String]と複数の文字列を改行文字で終端して連結する関数unlines :: [String] -> Stringとを使えば，calculator は以下のように構成できる。

```
calculator :: String -> String
calculator = unlines . map calc . lines
```

ここで，unlines と map calc との間と，map calc とlines との間にあるドットは，関数合成演算子である。この演算子は左右のオペランド(ともに関数)を連結して，関数を１つ作る。右の関数を引数に適用した結果の値に左の関数を適用するような関数がでる。この関数合成演算子の型は (b -> c) -> (a -> b) -> (a -> c)である。左オペランドの関数の始域の型 b と右オペランドの関数の終域の型 b が一致していることがこの演算子の働きを示している。

```
lines      :: String     -> [String]
map calc   ::   ?        ->   ?
unlines    :: [String]   -> String
```

のように型を並べてみれば，map calc :: [String] -> [String]であることが見てとれる。また，map calc は関数 calc をリストの要素に適用した結果をリストにする関数であり，関数 calc :: String -> String は１回分の入力から出力への関数であることからも同じ結論になる。calc の仕事は，

1．文字列から四則演算式の構成を切り取りつつ演算式が表す値に変換
2．整数型の値を文字列に変換

である。type Expr = Intを宣言しておけば，１つめの関数を readExpr :: String -> Expr, ２つめの関数をshowValue :: Expr -> Stringとすると，calcは以下のように構成できる。

```
calc :: String -> String
calc = showValue . readExpr
```

これらの２つの関数のうち，showValueについては，Expr は Intの別名であるから，定義済みで提供される関数show :: Int -> Stringを使って以下のように定義できる。

```
showValue :: Expr -> String
showValue = show
```

4.3　四則演算式の具象構文

正しい入力としては，たとえば，2+3，2-3，2*3，2/3 なども含み，1 や23も含むものとする。さらに，2 +3*4/5 のような複雑なもの,(2+3) * (4-5) のような括弧を用いたものまで含める。

入力文字列が「四則演算式として正しく表すよう構成されたもの」であることを仮定するが，どのように構成されたものを正しいとするか，文法を規定しておく必要がある。通常このような文法は拡張 BNF 記法を用いて規定する。ここでは，四則演算式を以下のように規定することにする。

expression　::=　*additive*
additive　::=　*multitive* (*addop*　*multitive*)*
multitive　::=　*primary* (*mulop*　*primary*)*

primary ::= *number* | '(' *expression* ')'
addop ::= '+' | '-'
mulop ::= '*' | '/'
number ::= ['0' - '9']$^+$

具象構文にしたがう入力文字列を受けいれ，指定の型の値に変換する関数を構文解析器(パーサ)という。上述のように定義した具象構文の非終端記号 *expression* に対応するパーサ `pExpression` を構成できれば，`readExpr` はそのパーサを使って構成できるはずである。

4.4 パーサ言語

単純なパーサを構成して複雑なパーサを組み立てる枠組みを作る。すなわち，

- 基本パーサを構成する手段
- 単純なパーサを組み合わせて複雑なパーサを構成する演算

を提供する(最近の現代的なプログラミング言語なら，このようなライブラリが最初から手にはいるが，ここでは関数プログラミングの考え方を示すために一から作る)。

一般に，特定用途関数の集合を規定し，それに属する少数の基本関数と，特定用途関数集合上に閉じた少数の演算ができて，当該集合のすべての要素を生成できれば，その特定用途の「言語」(DSL(Domain Specific Language))を構成できたことになる。

4.4.1 パーサの型

まず，パーサの型から考える。トークン列から求める型の値(以降，解析値と呼ぶ)を構成する関数というのが最初の構想になる。

```
type Parser = [Token] -> Value
```

トークンの型と解析値の型をパラメータとすると，

```
type Parser t v = [t] -> v
```

となる。

しかし，このパーサではすべてを一度に変換することになり，部品パーサを構成して，その部品パーサを合成して全体を組み立てるには不向きである。パーサを組み立て部品とするには，パーサの解析結果は，解析値と入力列の残りの部分の対になるようにする。

```
type Parser t v = [t] -> (v,[t])
```

さらに，入力が文法を満していなかったり，文法によっては途中で複数の構文解析が可能な場合があることも考慮にいれて，適用すると解析結果のリストになるようなパーサ関数を考えるのがよい。

```
type Parser t v = [t] -> [(v,[t])]
```

パーサの解析結果が空のリストであれば，これは解析が失敗したことを表す。

パーサが必要となる課題の多くでは，入力は文字列 `String`，すなわち，文字のリスト [`Char`] であるので，トークンを文字に固定してしまっても汎用性を大きく損うことはない。結局パーサ関数の型は，以下のように解析値の型をパラメータとして以下のようにする。

```
type Parser a = String -> [(a, String)]
```

4.4.2 基本パーサの構成

入力文字列の先頭文字が指定した条件を満していれば，その文字を消費し，解析値をその文字とするパーサを構成する関数を定義する。加えて，先頭が指定した文字であったとき，その文字を消費し，それを解析値とするパーサを定義する。

```
pSat :: (Char -> Bool) -> Parser Char
pSat pred (c:cs) | pred c = [(c, cs)]
pSat _    _               = []

pChar  ::  Char -> Parer Char
pChar c = pSat (c ==)
```

`pSat` の定義に左辺に現れている `(c:cs)` は，パーサへの入力文字列に1文字以上の文字が含まれている場合，先頭文字を `c`，残りの文字列を `cs` とする(すなわち，局所変数 `c`，`cs` を導入し，それぞれを先頭文字と残りの文字列で束縛する)という表明である。さらに，`| pred c` の部分は，付帯条件を示していて，この場合は先頭文字 `c` が指定された述語関数 `pred` を満す，すなわち，`pred c` の値が `True` のときに，右辺の式を採用することを示している。`pSat` の2つめの定義左辺に現れている2つの `_` は，局所変数は導入しないことを表している。すなわち，1つめの定義で捕捉できない場合(すなわち，入力文字列の先頭が条件を満さない場合および入力文字列が空の場合)をすべて引き受けている。

4.4.3 パーサの変換演算

`Parser a`型のパーサを適用して，入力をいくらか消費し，変換結果がなにがしかの値になったとき，変換結果だけを関数 `f :: a -> b`で変換すれば，この構文解析は `Parser b`型のパーサを使ったのと同じことである。そこで，指定した関数 `f :: a -> b`を指定して，`Parser a` 型のパーサを別の `Parser b` 型のパーサに変換する演算(`Parser a -> Parser b`)を構成する関数 `pFmap` を定義する。すなわち，`pFmap`を関数 `f :: a -> b` に適用すると，`Parser a -> Parser a` という型のパーサ変換関数になり，この関数をパーサ `p :: Parser a` に適用すると，`Parser b`という別の型のパーサになる。さらに，`Parser b` 型のパーサは `String -> [(b, String)]` 型の関数のことであるから，この関数を入力 `s :: String` に適用すると，[(b,

String)] 型の値になる。リストをある種の集合と見なすと，pFmap f p sは，内包表記であらわすと

$$\{\ (f\ a,\ t) \mid (a,\ t) \in p\ s\ \}$$

という集合になると見なせる。同様にリスト内包表記(list comprehension)があれば，以下のように定義できる。

```
pFmap :: (a -> b) -> (Parser a -> Parser a)
pFmap f p  s = [ (f a, t) | (a, t) <- p  s ]
```

pFmapは，値を変換する a -> b 型の関数を使って，Parser a -> Parser b 型のパーサ変換関数を構成する関数であるが，視点を換えれば，a -> b 型の関数と Parser a 型のパーサとをオペランドとする二項演算と見ることもできる。この場合の演算は左オペランドの関数を使って，右オペランドのパーサを別のパーサに変換するというものである。実際，そのように考えて利用できれば便利であるから，pFmap に対応する演算子$$$も定義しておく。

```
($$$) :: (a -> b) -> Parser a -> Parser b
($$$) = pFmap
```

これで，パーサ変換演算を構成する関数 pFmap が定義できたわけであるが，上の議論で，関数 f :: a -> b は関数適用すると b 型の値になるわけだが，この b が b1 -> b2 という関数かもしれない。そうであるとすると，pFmap を f に適用すると，Parser (b1 -> b2) 型のパーサが構成される。これをもとにパーサ変換関数を構成する関数 pApp があると便利であるので定義しておく。

```
pApp :: Parser (a -> b) -> (Parser a -> Parser b)
pApp p q s
   = [ (f a, u) | (f, t) <- p s, (a, u) <- q t ]
```

s の先頭部分を消費して，f を手に入れ，残り t の先頭部分を消費して，a を手にいれる。

pFmap 関数の議論と同様に，pApp 関数は，Parser (a -> b) 型のパーサと Parser a 型のパーサとをオペランドとする二項演算，すなわち，左オペランドのパーサを使って，右オペランドのパーサを別のパーサに変換する演算を提供する。ここでも，pApp に対応する二項演算子***を定義しておく。

```
(***) :: Parser (a -> b) -> Parser a -> Parser b
(***) = pApp
```

4.4.4 パーサ選択とパーサ連接

2つのパーサの選択演算を実現する関数 pAlt は拡張 BNF 記法の選択'|'に対応する。

```
pAlt :: Parser a -> Parser a -> Parser a
pAlt p q s = p s ++ q s
```

ここで，++ はリストの連結演算子である。

パーサp :: Parser a とパーサq :: Parser b とを順に適用し，2つの値から新しい値を合成する関数 f :: a -> b -> cを適用するようなParser cは以下のように構成できる。

```
(f $$$ p) *** q
```

すなわち，f $$$ p は Parser (b -> c) 型のパーサであるから,(f $$$ p) *** q は Parser c 型のパーサになるということである。

$$$と***とは同じ優先順位，左結合性の演算子であるとして，括弧を省略して，

```
f $$$ p *** q
```

と表記することにする。

これを利用して2つのオペランドのパーサを順に適用して，得られた値を対構成子 (,) :: a -> b -> (a, b) で合成するパーサを構成する演算子&&&を以下のように定義しておく。

```
(&&&) :: Parser a -> Parser b -> Parser (a, b)
p &&& q = (,) $$$ p *** q
```

4.4.5 パーサの反復

拡張 BNF における反復$()^*$ に対応して指定したパーサを0回以上反復するパーサを構成する関数 pMany と1回以上反復するパーサ pMany1 とを用意する。pMany p は，入力を消費せず解析値を空のリストとするパーサ pUnit [] と p を1回以上反復するパーサ pMany1 p の選択である。pMany1 p は，パーサ p とパーサ pMany p を順に適用して,その解析値を前のパーサの解析値(型は a)と後のパーサの解析値(型は [a])とをリストの構成子関数 (:) :: a -> [a] -> [a] で合成するパーサである。(:) x xs は x をリスト xs の先頭に加えたリストである。これは，x : xs と書いても同じである。

```
pMany :: Parser a -> Parser [a]
pMany p = pUnit [] ||| pMany1 p

pMany1 :: Parser a -> Parser [a]
```

```
pMany1 p = (:) $$$ p *** pMany p
```

4.5 readExpr および pExpression の構成

readExpr s は，構文解析結果 pExpression s の内，入力文字列をすべて消費したときの解析値である。

```
type Expr = Int

readExpr :: String -> Expr
readExpr s
  = case [ x | (x, "") <- pExpression s ] of
      [e] -> e
      _   -> error "no deterministic parse"
```

pExpression は拡張 BNF 記法の記述を機械的にプログラムに書き換えること定義できる。開始記号 *expression* の生成規則は，

$$expression \quad ::= \quad additive$$

であるから，

```
pExpression  ::  Parser  Expr
pExpression = pAdditive
```

非終端記号 *additive* および *adop* の生成規則は，

$$additive \quad ::= \quad multitive \ (\ addop \quad multitive\ )^{*}$$
$$addop \quad ::= \quad '+' \ | \ '-'$$

であるから，それぞれに対応するパーサは

```
pAdditive :: Parser  Expr
pAdditive = mkEBOp
    $$$  pMultitive
    ***  pMany (pAddOp &&&  pMultitive)

type BOp =  Expr ->  Expr -> Expr

pAddOp :: Parser BOp
pAddOp = pPlus ||| pMinus
```

と定義できる。

パーサpMultitiveの解析値の型はExprであり，パーサpMany (pAddOp &&& pMultitive)の解析値は[(BOp, Expr)]型であるから，合成関数mkEBOpの型はExpr -> [(BOp, Expr)] -> Exprとなる。mkEBOpの仕事は，１つめパーサの解析値を初期値として，２つめのパーサの解析値であるリストを左畳み込むことである。畳むための関数ebop の型は Expr -> (BOp, Expr) -> Expr である。ebopは，二項演算 o を２つのオペランド e1, e2 に適用する関数である。

```
mkEBOp :: Expr -> [(BOp, Expr)] -> Expr
mkEBOp = foldl ebop
  where
    ebop e1 (o, e2) = o e1 e2
```

foldl :: (a -> b -> a) -> a -> [b] -> a は汎用の左畳み込み関数で，たとえば Haskell 2010 では仕様の一部として，以下のように定義され，標準ライブラリとして提供されている。

```
foldl  ::  (a -> b -> a) -> a -> [b] -> a
foldl  f  e  []      =  e
foldl  f  e  (x:xs) =  foldl f (f e  x) xs
```

したがって，関数 foldl f e :: [b] -> a は以下のような変換を行うものになる。

$$[x_1,\ x_2,\ \ldots,\ x_n] \Rightarrow (\ f \ldots (\ f\ (\ f\ e\ x_1)\ x_2)\ \ldots\ x_n)$$

同様に非終端記号 *multitive* および *mulop* の生成規則は，

$$multitive \quad ::= \quad primary \quad (\ mulop \quad primary\ )^{*}$$
$$mulop \quad ::= \quad '*' \ | \ '/'$$

であるから，それぞれに対応するパーサは，

```
pMultitive :: Parser Expr
pMultitive =  mkEBOp
    $$$ pPrimary
    *** pMany (pMulOp &&& pPrimary)

pMulOp :: Parser BOp
pMulOp = pTimes ||| pDivide
```

二項演算子のそれぞれを解析値とするパーサは，二項演算子を表すそれぞれの文字を解析値とするパーサから構成する。

```
pPlus, pMinus, pTimes, pDivide :: Parser BOp
pPlus   = const (+) $$$ pChar  '+'
pMinus  = const (-) $$$ pChar  '-'
pTimes  = const (*) $$$ pChar  '*'
```

```
pDivide = const div $$$ pChar  '/'
```

ここで，`const :: a -> (b -> a)` は，常に指定した値になる関数を構成する関数である。すなわち，`const (+)` は，なにに適用しても，`(+)` になる関数である。

最後に，

$$primary \quad ::= \quad number \mid \text{'('} \ expression \ \text{')'}$$
$$number \quad ::= \quad [\text{'0'}\text{-}\text{'9'}]^{+}$$

という生成規則から，対応するパーサを定義する。

```
pPrimary  ::  Parser  Expr
pPrimary = pNumber
       |||  ( between $$$  pChar   '('
                      ***  pExpression
                      ***  pChar   ')' )

between  ::  a -> b -> c -> b
between _ b _  =  b

pNumber :: Parser  Expr
pNumber =  readInt $$$  pMany1 (pSat isDigit)

readInt :: String -> Int
readInt = foldl f 0
 where
   f a c = 10 * a + digitToInt c
```

ここで，`readInt` は数字だけからなる文字列を `Int` 型の値に変換する関数で，前にも使った左畳み込み関数 `foldl` を使って上位桁から畳み込んでいる。この畳み込みはホーナー法の一種である。

5　まとめ

関数プログラミング入門者が最初に思う疑問は，なにをどう考えてプログラムを書けばよいかが判らないというものである。それが，関数プログラミングとはなにか，他の手法とどう違うのかという質問に形を変えて問われていると思われる［5］。関数プログラミングの際立つ特徴の記述は，そのままではプログラムをどう書くかには繋りにくい。

関数プログラミングの思考を表現したプログラムは，式と値だけの単純な構造で，プログラムの字面に現れない暗黙の状態を含まない。それゆえに意味を追いやすく，観察することで構造が見えやすく，抽象化しやすい。これは関数プログラミングの思考法が数学的思考法であり，プログラムの記述方法が参照透明な数学的記述方法であることに拠る。

静的型付け機構をもち，関数プログラミングのために設計された参照透明な言語で，さらに日本語で容易に手に入る「関数プログラミング」の入門書［6］［7］［8］［9］で使われているものは，Haskell しかない。それらのテキストの中で，本稿と同じ，関数の型シグネチャを手掛りに，定義を与えようとするものはない。

本稿では，特有の概念を対比によって説明することなく，素朴に型式を観察することで，高階関数をどのように考え，構成するかが見えることを示した。プログラムにある関数にはすべて型シグネチャが書かれている。実は，静的型付けを行う関数プログラミング用の記述言語系がもつ型システムは，強力な型推論機構が備わっているのが普通で，型シグネチャがなくても言語系が正しく推論してくれる。それゆえ，「型シグネチャを書く必要はない」と断言するテキスト［10］や，「型シグネチャは不要だが，ドキュメントとして機能するので書いておくと，読みやすくなる」と説明するテキストもある［11］。しかし，前節でみたように型シグネチャを書くことはプログラミングの一部であり，型推論機構がプログラマの意図と実装の齟齬をチェックしてくれからこそ，書く意味があり，ドキュメント以上のものである。関数の値そのものは表示させて見ることができないが，関数の型シグネチャには関数の仕様の概要が表現されているので，関数定義はこれを指針に進めるのが合理的である。

参考文献

［1］五味弘：はじめての Lisp 関数型プログラミング: ラムダ計算からリファクタリングまで一気にわかる，SoftwareDesign plus シリーズ，技術評論社（2016）.
［2］Backus，J.: Can Programming Be Liberated from the Von Neumann Style?: A Functional Style and Its Alge-bra of Programs，Commun. ACM，Vol. 21，No. 8，pp.613–641（1978）.
［3］Abelson，H.，Sussman，G. and Sussman，J.: Structure and Interpretation of Computer Programs，Electrical engineering and computer science series，MIT Press（1996）.
［4］Marlow，S.: Haskell 2010 Language Report，https://www.haskell.org/onlinereport/haskell2010/.
［5］住井英二郎：「関数型言語」に関する FAQ 形式の一般的説明，http://qiita.com/esumii/items/ec589d138e72e22ea97e（2015-2016）.
［6］重城良国：Haskell 教養としての関数型プログラミング，秀和システム（2017）.
［7］Richard，B.: Haskell による関数プログラミングの思考法，アスキー・ドワンゴ（2017）.
［8］Lipovaca，M.: すごい Haskell たのしく学ぼう!，オーム社（2012）.
［9］Graham，H.: プログラミング Haskell，オーム社（2009）.
［10］五十嵐淳：プログラミング in OCaml —関数型プログラミングの基礎から GUI 構築まで—，株式会社技術評論社（2007）.
［11］本間雅洋，類地孝介，逢坂時響：Haskell 入門—関数型プログラミング言語の基礎と実践—，株式会社技術評論社（2017）.