Perl 実習マニュアル

第零版
2014 年 10 月 3 日（金）

目次

第 1 章　Perl の基礎

1.1　プログラム

1.1.1　文書と言語

文字を並べることによって何かを記述したものは、「文書」(document) と呼ばれます。

文書を作るためには、記述したいことを意味として持つように、文字を並べていく必要があります。そして、そのためには、文字をどのように並べればどのような意味になるかということを定めた規則が必要になります。そのような規則は、「言語」(language) と呼ばれます。

人間に読んでもらうことを第一の目的とする文書を書く場合は、日本語や中国語やアラビア語のような、「自然言語」(natural language) と呼ばれる言語が使われます。自然言語というのは、人間の社会の中で自然発生的に形成された言語のことです。

言語には、自然言語のほかに、人間が意図的に設計することによって作られた、「人工言語」(artificial language) と呼ばれるものもあります。人間ではなくてコンピュータに読んでもらうことを第一の目的とする文書を書く場合は、通常、自然言語ではなく人工言語が使われます。

1.1.2　プログラムとプログラミング

コンピュータに何らかの動作を実行させるためには、実行してほしいことがどんな動作なのかということを記述した文書をコンピュータに与える必要があります。そのような文書は、「プログラム」(program) と呼ばれます。

プログラムを作成するためには、プログラムを書くという作業だけではなくて、プログラムの構造を設計したり、プログラムの動作をテストしたり、その不具合を修正したりするというような、さまざまな作業が必要になります。そのような、プログラムを作成するために必要となるさまざまな作業の全体は、「プログラミング」(programming) と呼ばれます。

1.1.3　プログラミング言語

プログラムというのも文書の一種ですから、それを書くためには何らかの言語が必要になります。プログラムを書く場合には、プログラムを書くことだけを目的として作られた人工言語を使うのが普通です。そのような、プログラムを書くための専用の言語は、「プログラミング言語」(programming language) と呼ばれます。

プログラミング言語には、たくさんのものがあります。例を挙げると、Fortran、COBOL、Lisp、Pascal、Basic、C、AWK、Smalltalk、ML、Prolog、Perl、PostScript、Tcl、Java、Ruby、…… というように、枚挙にいとまがないほどです。

1.1.4　スクリプト言語

プログラミング言語を設計するときには、どのようなことを得意とする言語を作るのかという方針を立てる必要があります。複数の方針を立ててプログラミング言語を設計することも可能です。しかし、プログラムがコンピュータによって実行されるときの効率を向上させるという方針と、プログラムが人間にとって書きやすくて読みやすいものになるようにするという方針とは、トレードオフの関係にあります。つまり、それらの二つの方針を両立させることは、とても困難なことなのです。

プログラミング言語の中には、書きやすさや読みやすさよりも実行の効率を優先させて設計されたものもあれば、それとは逆に、効率よりも書きやすさや読みやすさを優先させて設計されたものもあります。後者の方針で設計されたプログラミング言語は、「スクリプト言語」(scripting language) と呼ばれます[1]。スクリプト言語としては、sed、awk、Perl、Python、Ruby、PHP などがよく使われています。

プログラミング言語で書かれた文書は「プログラム」と呼ばれるわけですが、スクリプト言語で書かれた文書は、「スクリプト」(script) と呼ばれることもあります。

1.1.5　この文章について

この文章（「Perl 実習マニュアル」）は、Perl というスクリプト言語を使って、プログラムというものの書き方について説明する、ということを目的とするチュートリアルです。

Perl というのは、Larry Wall さんという人によって設計されたスクリプト言語です。

[1] スクリプト言語は、「軽量言語」(lightweight language) と呼ばれることもあります。

Perl の開発が始まったのは 1987 年ごろのことで、Perl は、それよりも遅れて開発が始まった Python、Ruby、PHP などのスクリプト言語に多大な影響を与えています。

1.2 インタプリタ

1.2.1 言語処理系

コンピュータというものは異質な二つの要素から構成されていて、それぞれの要素は、「ハードウェア」(hardware) と「ソフトウェア」(software) と呼ばれます。ハードウェアというのは物理的な装置のことで、ソフトウェアというのはプログラムなどのデータのことです。

コンピュータは、さまざまなプログラミング言語で書かれたプログラムを理解して実行することができます。しかし、コンピュータのハードウェアが、ソフトウェアの助力を得ないで単独で理解することのできるプログラミング言語は、ハードウェアの種類によって決まっているひとつの言語だけです。

ハードウェアが理解することのできるプログラミング言語は、そのハードウェアの「機械語」(machine language) と呼ばれます。機械語というのは、人間にとっては書くことも読むことも困難な言語ですので、人間が機械語でプログラムを書くということはめったにありません。

人間にとって書いたり読んだりすることが容易なプログラミング言語で書かれたプログラムをコンピュータに理解させるためには、そのためのプログラムが必要になります。そのような、人間が書いたプログラムをコンピュータに理解させるためのプログラムのことを、「言語処理系」(language processor) と呼びます（「言語」を省略して、単に「処理系」と呼ぶこともあります)。

言語処理系には、「コンパイラ」(compiler) と「インタプリタ」(interpreter) と呼ばれる二つの種類があります。コンパイラというのは、人間が書いたプログラムを機械語に翻訳するプログラムのことで、インタプリタというのは、人間が書いたプログラムがあらわしている動作をコンピュータに実行させるプログラムのことです。

1.2.2 Perl の言語処理系

Perl には、perl という言語処理系があります。少し紛らわしいかもしれませんが、先頭の p が大文字になっている Perl というのがプログラミング言語の名前で、小文字になっている perl というのが Perl の言語処理系の名前です。

perl は、Perl Foundation という非営利団体が中心となって、オープンソースとして開発されている言語処理系です。

Linux のディストリビューションには、デフォルトで perl が含まれています。また、Mac OS X にも、最初から perl が含まれています。

perl を Windows の上で使うためには、Windows 版の perl をダウンロードして、自分でインストールする必要があります。Windows 版の perl としては、たとえば次のようなものがあります。

ActivePerl	`http://www.activestate.com/activeperl`
Strawberry Perl	`http://strawberryperl.com/`

ちなみに、perl は、Perl のプログラムを仮想的なハードウェアの機械語に翻訳してから実行しますので、純粋なコンパイラでも純粋なインタプリタでもなくて、それらを合体させたようなプログラムです。

1.2.3 プログラムの入力

プログラムをファイルに保存したり、すでにファイルに保存されているプログラムを修正したりしたいときは、「テキストエディター」(text editor) と呼ばれるソフトを使います（テキストエディターは、単に「エディター」(editor) と呼ばれることもあります)。

それでは、何らかのテキストエディターを使って、次のプログラムを入力して、`hello.pl` というファイルに保存してください。

プログラムの例 `hello.pl`

```
print "Hello, world!\n"
```

このプログラムは、実行すると、「Hello, world!」という言葉を出力して、さらに改行を出力する、という動作をします。

Perl で書かれたプログラムをファイルに保存する場合、ファイル名の拡張子は、`.pl` にします。

1.2.4　プログラムの実行

Perl で書かれたプログラムは、

```
perl パス名
```

というコマンドによって実行することができます。コマンドライン引数として指定するのは、プログラムが保存されているファイルのパス名です。

それでは、先ほど入力したプログラムを perl に実行させてみましょう。コマンドを入力するためのアプリ（Linux や Mac OS ならばターミナル、Windows ならばコマンドプロンプト）を起動して、プログラムのファイルがあるフォルダをカレントフォルダにして、

```
perl hello.pl
```

というコマンドを入力してみてください。そうすると、perl によってプログラムが実行されて、

```
Hello, world!
```

という言葉が出力されるはずです。

1.2.5　エラー

プログラムの中には、何らかの間違いが含まれていることがあります。そのような、プログラムの中に含まれている間違いは、「エラー」(error) と呼ばれます。

言語処理系は、入力されたプログラムがエラーを含んでいる場合、そのエラーについてのメッセージを出力します。そのような、エラーについてのメッセージは、「エラーメッセージ」(error message) と呼ばれます。

それでは、エラーを含んでいる次の Perl のプログラムをファイルに保存してください。

プログラムの例　error.pl

```
print "Hello, world!\n
```

先ほどのプログラムとの相違点は、行末にあった二重引用符 (") が抜けているということです。このプログラムを実行させると、perl は、次のようなエラーメッセージを出力します。

```
Can't find string terminator '"' anywhere before EOF at error.pl line 1.
```

このエラーメッセージは、ファイルの終わりまで読んだにもかかわらず、二重引用符が依然として閉じられていない、ということを述べています。ちなみに、このエラーメッセージの中に出てくる EOF という単語は、ファイルの終わりを意味する end of file の略称です。

1.3　文と式

1.3.1　プログラムの構造

プログラムという文書は、構文的な部品から構成されます。そして、プログラムを構文的に構成している部品には、いくつかの階層があります。もっとも下にある階層は文字で、もっとも上にある階層はプログラム全体です。

Perl というプログラミング言語の場合、プログラムを構成している部品の中間的な階層として、「文」(statement) と呼ばれるものや、「式」(expression) と呼ばれるものがあります。

第 1.2 節で、

```
print "Hello, world!\n"
```

というプログラムを紹介しましたが、このプログラムの全体は、1 個の式になっています。また、この式の中にある、

```
"Hello, world!\n"
```

という部分も、1 個の式です。このように、式という部品は、その中にさらに式が含まれていることもあります。つまり、式という部品は入れ子にすることができる、ということです。式だけではなくて、文も、入れ子にすることができます。

Perl では、すべての式は、文でもあります。式がそのまま文になったものは、「単純文」(simple statement) と呼ばれます。

1.3.2 式と評価と値

プログラムという文書は、コンピュータによって実行されるべき動作を記述したものですから、コンピュータは、プログラムという文書を実行することができます。

同じように、文というプログラムの部品も、コンピュータの何らかの動作をあらわしています。ですから、コンピュータは、文というプログラムの部品を実行することができます。

さらに、式というプログラムの部品も、コンピュータの何らかの動作をあらわしています。ですから、コンピュータは、式というプログラムの部品を実行することができます。ただし、式の場合は、「実行する」(execute) とは言わずに、「評価する」(evaluate) と言うのが普通です。

式を評価すると、その結果として一つのデータが得られます。式を評価することによって得られるデータは、その式の「値」(value) と呼ばれます。

「評価する」という言葉は、「値を求める」というニュアンスを含んでいます。式を実行することを、「実行する」と言わずに「評価する」と言うのは、「式の実行というのは、単なる動作の実行ではなくて、値を求めるという志向性を持った動作の実行である」という意識が強く働いているからです。

1.3.3 リテラル

特定のデータを生成するという動作を記述した式は、「リテラル」(literal) と呼ばれます。リテラルを評価すると、それによって生成されたデータが、そのリテラルの値になります。

リテラルにはさまざまな種類があるのですが、ここでは、整数と文字列のデータを生成するリテラルについて、ごく簡単に説明しておきたいと思います（リテラルについては、第 2.1 節で、さらに詳細に説明します)。

整数のデータを生成するリテラルは、「整数リテラル」(integer literal) と呼ばれます。`0` から `9` までの数字を並べることによってできる列は、それによってあらわされる整数のデータを生成する整数リテラルです。たとえば、`2800` というのは、整数リテラルの一例です。この整数リテラルを評価すると、2800 という整数のデータが生成されて、そのデータが値として得られます。

文字列のデータを生成するリテラルは、「文字列リテラル」(string literal) と呼ばれます。文字列を二重引用符 (`"`) で囲んだものは、その文字列のデータを生成する文字列リテラルです。たとえば、`"namako"` というのは、文字列リテラルの一例です。この文字列リテラルを評価すると、`namako` という文字列のデータが生成されて、そのデータが値として得られます。

第 1.2 節で紹介したプログラムの中に書かれている、

```
"Hello, world!\n"
```

という式も、文字列リテラルです。この文字列リテラルの中に書かれている `\n` という部分は、改行を意味しています。

1.3.4 文の列

Perl のプログラムの中には、文を、いくつでも並べて書くことができます。プログラムの中に文がいくつか並んでいる場合、コンピュータはそれらの文を、原則として、先頭から末尾に向かって 1 回ずつ実行していきます。

文の列の中に単純文が含まれている場合、それらの単純文の末尾には、セミコロン (`;`) を書く必要があります。

それでは、実際に、2 個以上の文から構成される文の列を書いて、それがどのように実行されるかを試してみましょう。

プログラムの例 `sequence.pl`

```
print "Kaname Madoka\n";
print "Akemi Homura\n";
print "Miki Sayaka\n";
```

実行例

```
Kaname Madoka
Akemi Homura
Miki Sayaka
```

1.4 REPL

1.4.1 REPL の基礎

第 1.2 節では、ファイルに保存されているプログラムを Perl の処理系に実行させる方法について説明したわけですが、この節では、プログラムをファイルに保存してから実行させるのではなくて、キーボードから直接、プログラムを処理系に入力して実行させる方法について説明したいと思います。

スクリプト言語の処理系の多くは、「REPL」と呼ばれる機能を持っています。REPL というのは、read-eval-print loop の略称で、次の三つの動作を延々と繰り返す、という処理系の動作のことです。

(1) プログラムを読み込む (read)。
(2) 読み込んだプログラムを実行する (eval)。
(3) 結果を出力する (print)。

ですから、REPL を使うことによって、プログラムをキーボードから直接入力して、それを処理系に実行させる、ということができます。

1.4.2 perl の REPL

perl の REPL は、

```
perl -de 0
```

というコマンドを入力することによって、起動することができます。

それでは、実際に perl の REPL を起動してみてください。そうすると、

```
DB<1>
```

というものが表示されるはずです。これが、perl の REPL が出力するプロンプトです。

1.4.3 REPL の終了

REPL は、q という文字を入力することによって終了させることができます。

それでは、REPL を終了させてみましょう。q という文字を入力して、そののちエンターキーを押してみてください。すると、REPL が終了して、Linux や Mac OS の場合はターミナルのプロンプトが、Windows の場合はコマンドプロンプトのプロンプトが表示されるはずです。

1.4.4 文の実行

perl の REPL は、文を読み込んで、その文を実行する、ということを繰り返します。

それでは、文を REPL に入力してみましょう。たとえば、

```
print "Congratulations!"
```

というように文を入力して、そののちエンターキーを押してみてください。すると、入力した文が実行されて、

```
Congratulations!
```

と出力されます。

1.4.5 式の値の出力

perl の REPL に対して単純文として式を入力すると、その式が評価されることになります。しかし、perl の REPL は、単純文として入力された式の値を出力しません。ですから、REPL を使って式の値を調べたいときは、そのための文を入力する必要があります。たとえば、

```
print 式
```

という文を入力すると、式の値が出力されます。また、print と書く代わりに、p という文字を使って、

```
p 式
```

という形のものを入力しても、やはり式の値が出力されます。ただし、この p は、REPL の中だけでしか使えません。

それでは、REPL を使って、整数リテラルの値を調べてみましょう。たとえば、

```
p 2800
```

というように、pと空白と整数リテラルを入力して、そののちエンターキーを押してみてください。すると、

```
2800
```

と出力されます。整数リテラルを評価すると、それによって生成された整数のデータが値として得られますので、2800という整数リテラルを評価した結果が、2800と出力されたわけです。

次に、REPL を使って、文字列リテラルの値を調べてみましょう。たとえば、

```
p "namako"
```

というように、pと空白と文字列リテラルを入力して、そののちエンターキーを押してみてください。すると、

```
namako
```

と出力されます。整数リテラルを評価すると、それによって生成された文字列のデータが値として得られますので、"namako"という文字列リテラルを評価した結果が、namakoと出力されたわけです。

1.5 空白と改行と注釈

1.5.1 空白と改行

空白という文字（スペースキーを押したときに入力される文字）と改行という文字（エンターキーを押したときに入力される文字）は、Perl のプログラムの意味に影響を与えません。たとえば、

```
print "Hello, world!"
```

というプログラムは、

```
print      "Hello, world!"
```

と書いたとしても同じ意味になりますし、

```
print
"Hello, world!"
```

と書いたとしても同じ意味になります。

ただし、文字列リテラルの中に空白を挿入した場合は、その空白を含んだ文字列のデータが生成されます。たとえば、

```
print "H e l l o ,  w o r l d !"
```

というプログラムは、

```
H e l l o ,  w o r l d !
```

という文字列を出力します。

名前の途中には、空白や改行を挿入することができません。ですから、printという名前を、

```
p r i n t
```

と書くことはできません。

Perl では、改行を挿入することによって文をいくつかの行に分けることができるわけですが、REPL に文を入力する場合、途中に改行を挿入することはできません。

1.5.2 注釈

プログラムを書いているとき、それを読む人間（プログラムを書いた人自身もその中に含まれます）に伝えたいことを、そのプログラムの一部分として書いておきたい、ということがしばしばあります。プログラムの中に書かれたそのような文字列は、「注釈」(comment) と呼ばれます。

注釈は、プログラムを処理するプログラムが、「ここからここまでは注釈だ」ということを認識することができるように、注釈を書くための文法にしたがって書く必要があります。

Perl では、「この部分は注釈です」ということを言語処理系に知らせるために、井桁 (`#`) という文字を使います。プログラムの中に井桁を書くと、Perl の言語処理系は、その直後から改行までの部分を注釈とみなして無視します。

それでは、REPL を使って、井桁から改行までの部分が本当に無視されるかどうかを確かめてみましょう。次のプログラムを REPL に入力してみてください。

```
print "Hello, world!" # I am a comment.
```

この場合、入力したプログラムの中にある「I am a comment.」という部分は、井桁と改行のあいだに書かれていますので、Perl のインタプリタはその部分を注釈とみなして無視します。

逆に、井桁を書かなかった場合は、どうなるでしょうか。次のプログラムを REPL に入力してみてください。

```
print "Hello, world!" I am a comment.
```

井桁を書かなかった場合、インタプリタは、書かれたものをすべて解釈しようとしますので、エラーメッセージが表示されることになります。

1.5.3　コメントアウトとアンコメント

プログラムを作成したり修正したりしているとき、その一部分を一時的に無効にしたい、ということがしばしばあります。そのような場合、無効にしたい部分を削除してしまうと、それを復活させるのに手間がかかりますので、削除するのではなくて、注釈にすることによって無効にするという手段が、しばしば使われます。記述の一部分を注釈にすることによって、それを無効にすることを、その部分を「コメントアウトする」(comment out) と言います。逆に、コメントアウトされている部分を復活させることを、その部分を「アンコメントする」(uncomment) と言います。

第 2 章　式

2.1　リテラル

2.1.1　リテラルの基礎

「リテラル」というものについては、すでに第 1.3.3 項で簡単に説明しましたが、そこでの説明は概略にすぎないものでしたので、この節では、リテラルについて、もう少し詳細に説明したいと思います。

特定のデータを生成するという動作を記述した式は、「リテラル」(literal) と呼ばれます。たとえば、`"suzume"` のような、文字列を二重引用符で囲んだものは、リテラルの一種です。

リテラルを評価すると、それによって生成されたデータが、その値として得られます。たとえば、`"suzume"` というリテラルを評価すると、それによって生成された `suzume` という文字列のデータが、その値として得られます。

なお、この文章のこれから先の部分では、誤解のおそれがない場合、「○○のデータ」のことを単に「○○」と呼ぶことがあります。たとえば、文字列そのものではなくて文字列のデータのことを指しているということが文脈から明らかに分かる場合には、文字列のデータのことを単に「文字列」と呼ぶことがあります。

2.1.2　整数リテラル

整数のデータを生成するリテラルは、「整数リテラル」(integer literal) と呼ばれます。

整数リテラルは、次の 4 種類に分類することができます。

- 10 進数リテラル (decimal integer literal)
- 16 進数リテラル (hexadecimal integer literal)
- 8 進数リテラル (octal integer literal)
- 2 進数リテラル (binary integer literal)

これらの整数リテラルの相違点は、その名前が示しているとおり、整数を表現するための基数です。つまり、それぞれの整数リテラルは、10、16、8、2 のそれぞれを基数として整数を表現します。

10 進数リテラルは、 481 とか 3007 というような、数字だけから構成される列（ただし、先頭の数字は 0 以外でないといけません）です。たとえば、 481 という 10 進数リテラルを評価すると、481 というプラスの整数が値として得られます。

16 進数リテラルと 2 進数リテラルは、基数を示す接頭辞を先頭に書くことによって作られます。基数を示す接頭辞は、16 進数は 0x 、2 進数は 0b です（ x と b は大文字でもかまいません）。たとえば、 0xff 、 0b11111111 は、いずれも、255 という整数を生成します。

8 進数リテラルは、先頭に数字の 0 を書くことによって作られます。たとえば、 0377 は、255 という整数を生成します。

2.1.3 浮動小数点数リテラル

ひとつの数値を、数字の列と小数点の位置という二つの要素で表現しているデータは、「浮動小数点数」(floating point number) と呼ばれます。

浮動小数点数のデータを生成するリテラルは、「浮動小数点数リテラル」(floating point literal) と呼ばれます。

.003 、 41.56 、 723. というような、ドット (.) という文字を 1 個だけ含んでいる数字の列は、浮動小数点数のデータを生成するリテラルになります。この場合、ドットは小数点の位置を示します。

浮動小数点数を生成するリテラルとしては、

a e b

という形のものを書くことも可能です（ e は大文字でもかまいません）。a のところには、ドットを 1 個だけ含むかまたは含まない数字の列を書くことができて、b のところには、数字の列または左側にマイナスのある数字の列を書くことができます。この形のリテラルを評価すると、

$a \times 10^b$

という浮動小数点数が生成されます。

リテラル	意味
3e8	3×10^8
6.022e23	6.022×10^{23}
6.626e-34	6.626×10^{-34}

2.1.4 マイナスの数値を生成する式

マイナス (-) という文字を書いて、その右側に整数または浮動小数点数のリテラルを書くと、その全体は、マイナスの数値を生成する式になります。たとえば、 -56 という式はマイナスの 56 という整数を生成して、 -0xff はマイナスの 255 という整数を生成して、 -8.317 はマイナスの 8.317 という浮動小数点数を生成します。

マイナスの数値を生成するこのような式の先頭に書かれるマイナスという文字は、リテラルの一部分ではなくて、第 2.2 節で説明することになる「演算子」と呼ばれるものです。

2.1.5 文字列リテラル

文字列のデータを生成するリテラルは、「文字列リテラル」(string literal) と呼ばれます。

文字列リテラルは、次の 2 種類に分類することができます。

- 二重引用符文字列リテラル (double-quote string literal)
- 一重引用符文字列リテラル (single-quote string literal)

2.1.6 二重引用符文字列リテラル

二重引用符文字列リテラル (double-quote string literal) は、二重引用符 (") で文字列を囲むことによって作られます。たとえば、 "namako" は、二重引用符文字列リテラルです。

二重引用符文字列リテラルは、二重引用符で囲まれた中にある文字列を生成します。たとえば、 "namako" という二重引用符文字列リテラルは、 namako という文字列を生成します。

2.1.7 エスケープシーケンス

文字の中には、たとえばビープ音や改ページのように、そのままでは文字列リテラルの中に書くことができない特殊なものがあります。

そのような特殊な文字を含んだ文字列を作りたいときに使われるのが、「エスケープシーケンス」(escape sequence) と呼ばれる文字列です。エスケープシーケンスは、必ず、バックスラッシュ(\) という文字で始まります[1]。

エスケープシーケンスには、次のようなものがあります。

\a	ビープ音	\f	改ページ
\t	水平タブ	\b	バックスペース
\'	一重引用符	\"	二重引用符
\n	改行	\r	キャリッジリターン
\\	バックスラッシュ	\c*X*	コントロール文字 (Ctrl+*X*)
ooo	8 進数 *ooo* に対応する文字	\x*hh*	16 進数 *hh* に対応する文字

二重引用符文字列リテラルの中にエスケープシーケンスが含まれていた場合、そのエスケープシーケンスは、それが意味している文字に置き換わります。

```
  DB<1> p "namako\numiushi\nkurage"
namako
umiushi
kurage
```

2.1.8 一重引用符文字列リテラル

一重引用符文字列リテラル (single-quote string literal) は、一重引用符 (') で文字列を囲むことによって作られます。たとえば、'namako' は、一重引用符文字列リテラルです。

二重引用符文字列リテラルと同様に、一重引用符文字列リテラルも、一重引用符で囲まれた中にある文字列を生成します。たとえば、'namako' という一重引用符文字列リテラルは、namako という文字列を生成します。

二重引用符文字列リテラルと、一重引用符文字列リテラルとの相違点は、バックスラッシュが特別扱いされるかどうかです。二重引用符文字列リテラルの中に含まれているバックスラッシュは、エスケープシーケンスの 1 文字目として解釈されます。ただし、バックスラッシュから始まる文字列をエスケープシーケンスとして解釈することができなかった場合、バックスラッシュは、それ自身として解釈されます。それに対して、一重引用符文字列リテラルの場合、その中に含まれているバックスラッシュは特別扱いされることなく、無条件にそれ自身として解釈されます。ですから、一重引用符文字列リテラルは、Windows のパス名のような、バックスラッシュを何個も含む文字列を生成したいときに便利です。

```
  DB<1> p 'c:\prog\tex\bin'
c:\prog\tex\bin
```

ただし、「一重引用符文字列リテラルの中にあるバックスラッシュはそれ自身として解釈される」ということには、例外もあります。それは、\' と\\という二つの場合です。\' は、一重引用符を意味するエスケープシーケンスと解釈されて、\\は、バックスラッシュを意味するエスケープシーケンスと解釈されます。

```
  DB<1> p 'philosopher\'s stone'
philosopher's stone
  DB<2> p 'c:\\prog\\tex\\bin'
c:\prog\tex\bin
```

2.2 演算子

2.2.1 演算子の基礎

Perl では、頻繁に必要となる単純な動作は、「演算子」(operator) と呼ばれるものを書くことによって記述することができます。

演算子を使いたいときは、演算子と式とを組み合わせた式を書きます。大多数の演算子は、それと式とを組み合わせた式の構造によって、次の 2 種類に分類することができます。

- 二項演算子 (binary operator)
- 単項演算子 (unary operator)

[1] バックスラッシュは、日本語の環境では円マーク (¥) で表示されることがあります。

2.2.2 二項演算子

「二項演算子」(binary operator) と呼ばれるグループに所属している演算子は、

式 二項演算子 式

という構造の式を作ります。

二項演算子を含む式を評価すると、原則的には、まず演算子の左右の式が評価されて、それらの式の値に対して、二項演算子があらわしている動作が実行されて、その結果が式全体の値になります。

2.2.3 算術演算子

数値に対する計算をあらわしている演算子は、「算術演算子」(arithmetic operator) と呼ばれます。

二項演算子でかつ算術演算子であるような演算子としては、次のようなものがあります。

a + b　a と b とを足し算（加算）する。

a - b　a から b を引き算（減算）する。

a * b　a と b とを掛け算（乗算）する。

a / b　a を b で割り算（除算）する。整数の範囲で割り切れない場合は小数点以下も求める。

a % b　a を b で除算したあまりを求める。

a ** b　a の b 乗（べき乗）を求める。

```
  DB<1> p 30+7
37
  DB<2> p 30-7
23
  DB<3> p 30*7
210
  DB<4> p 30/7
4.28571428571429
  DB<5> p 30%7
2
  DB<6> p 3**4
81
```

2.2.4 文字列の連結

.（ドット）という演算子は、文字列を連結します。

. は、左右に書かれた式の値を連結して、それらをひとつの文字列にします。

```
  DB<1> p "kitsune" . "udon"
kitsuneudon
```

もしも、. の左右に書かれた式の値が文字列ではないデータだった場合は、そのデータが文字列に変換されたのちに連結が実行されます。

```
  DB<1> p "string" . 999
string999
  DB<2> p 999 . "string"
999string
  DB<27> p 999 . 777
999777
```

2.2.5 優先順位

ひとつの式の中に 2 個以上の演算子が含まれている場合、その式はどのように解釈されるのでしょうか。たとえば、

```
2+3*4
```

という式は、

```
2+3 *4
```

という構造なのでしょうか。それとも、

```
2+3*4
```

という構造なのでしょうか。

この問題は、個々の演算子が持っている「優先順位」(precedence) と呼ばれるものによって解決されます。

優先順位というのは、演算子が左右の式と結合する強さのことだと考えることができます。優先順位が高い演算子は、それが低い演算子よりも、より強く左右の式と結合します。

`*` と `/` と `%` は、`+` と `-` よりも高い優先順位を持っています。ですから、

```
2+3*4
```

という式は、

```
2+3*4
```

という構造だと解釈されます。

```
DB<1> p 2+3*4
14
```

さらに、`**` は、`*` と `/` と `%` よりも高い優先順位を持っています。ですから、

```
2*3**4
```

という式は、

```
2*3**4
```

という構造だと解釈されます。

```
DB<1> p 2*3**4
162
```

2.2.6　結合規則

ひとつの式の中に同じ優先順位を持っている 2 個以上の演算子が含まれている場合、その式はどのように解釈されるのでしょうか。たとえば、

```
10-5+2
```

という式は、

```
10-5+2
```

という構造なのでしょうか。それとも、

```
10-5+2
```

という構造なのでしょうか。

この問題は、同一の優先順位を持つ演算子が共有している「結合規則」(associativity) と呼ばれる性質によって解決されます。

結合規則には、「左結合」(left-associativity) と「右結合」(right-associativity) という二つのものがあります。左結合というのは、左右の式と結合する強さが左にあるものほど強くなるという性質で、右結合というのは、それが右にあるものほど強くなるという性質です。

`+`、`-`、`*`、`/`、`%` の結合規則は、左結合です。したがって、

```
10-5+2
```

という式は、

```
10-5+2
```

という構造だと解釈されます。

```
DB<1> p 10-5+2
7
```

`**` の結合規則は、右結合です。したがって、

```
2**3**4
```

という式は、

```
2**3**4
```

という構造だと解釈されます。

```
  DB<1> p 2**3**4
2.41785163922926e+024
```

2.2.7 丸括弧

ところで、2 と 3 とを足し算して、その結果と 4 とを掛け算したい、というときは、どのような式を書けばいいのでしょうか。先ほど説明したように、+ と * とでは、* のほうが優先順位が高くなっていますので、

```
2+3*4
```

と書いたのでは、期待した結果は得られません。

演算子の優先順位や結合規則に縛られずに、自分が望んだとおりに式を解釈してほしい場合は、ひとまとまりの式だと解釈してほしい部分を、丸括弧 (()) で囲みます。そうすると、丸括弧で囲まれている部分は、演算子の優先順位や結合規則とは無関係に、ひとまとまりの式だと解釈されます。

```
  DB<1> p (2+3)*4
20
  DB<2> p (2*3)**4
1296
  DB<3> p 10-(5+2)
3
  DB<4> p (2**3)**4
4096
```

2.2.8 単項演算子

「単項演算子」(unary operator) と呼ばれるグループに所属している演算子は、

```
単項演算子 式
```

という構造の式、または、

```
式 単項演算子
```

という構造の式を作ります。式の左側に置かれる単項演算子は「前置演算子」(prefix operator) と呼ばれ、式の右側に置かれる単項演算子は「後置演算子」(postfix operator) と呼ばれます。

ほとんどの二項演算子は、単項演算子よりも低い優先順位を持っていますが、べき乗を求める二項演算子 (**) は、単項演算子よりも高い優先順位を持っています。

2.2.9 符号の反転

- という前置演算子は、数値の符号（プラスかマイナスか）を反転させる算術演算子です。

```
  DB<1> p -(3+5)
-8
  DB<2> p -(3-5)
2
  DB<3> p -3**2
-9
  DB<4> p (-3)**2
9
```

2.3 関数呼び出し

2.3.1 関数

Perl では、動作させることのできるもののことを「関数」(function) と呼びます。

関数を動作させることを、それを「呼び出す」(call) と言います。そして、関数を呼び出すという動作をあらわす式は、「関数呼び出し」(function call) と呼ばれます。

2.3.2 引数と戻り値

関数は、自分が動作を開始する前に、自分を呼び出した者からデータを受け取ることができます。関数が受け取るデータは、「引数」(argument) と呼ばれます（「引数」は「ひきすう」と読みます)。関数は、引数を何個でも受け取ることができます。

関数は、自分の動作が終了したのちに、自分を呼び出した者にデータを返すことができます。関数が返すデータは、「戻り値」(return value) と呼ばれます。関数は、引数は何個でも受け取ることができるのに対して、返すことができる戻り値は 1 個だけです。

2.3.3 サブルーチンと組み込み関数

Perl では、「サブルーチン定義」(subroutine definition) と呼ばれる文をプログラムの中に書くことによって、関数を自由に定義することができます（サブルーチン定義の書き方については、第 7 章で説明することにしたいと思います)。プログラムの中にサブルーチン定義を書くことによって定義された関数は、「サブルーチン」(subroutine) と呼ばれます。

「サブルーチン」の対義語は、「組み込み関数」です。「組み込み関数」(built-in function) というのは、Perl の処理系に組み込まれている関数のことです。組み込み関数は、サブルーチン定義を書かなくても、関数呼び出しを書くだけで呼び出すことができます。

たとえば、Perl の処理系には、`length` という関数が組み込まれています。この組み込み関数は、1 個の文字列を引数として受け取って、その長さ（文字の個数）を求めるという動作をして、その結果を戻り値として返します。たとえば、`umiushi` という文字列を引数として `length` に渡したとすると、`length` は、7 という整数を戻り値として返します。

2.3.4 関数呼び出しの書き方

関数呼び出しは、

関数名 (式, ···)

と書きます。この中の「関数名」というところには、呼び出したい関数の名前を書きます。

関数呼び出しを評価すると、名前で指定された関数が呼び出されて、丸括弧の中に書かれた式の値が、その関数に引数として渡されます。そして、その関数が返した戻り値が、関数呼び出しの値になります。

`length` という関数を呼び出して、`umiushi` という文字列を引数として渡す関数呼び出しは、

```
length("umiushi")
```

と書くことができます。この式を評価すると、その値として 7 という整数が得られます。

```
  DB<1> p length("umiushi")
7
```

関数呼び出しの中の丸括弧は、省略することができます。ですから、`length` を呼び出す関数呼び出しは、

```
length "umiushi"
```

と書くこともできます。

第 1.2 節で、

```
print "Hello, world!\n"
```

というプログラムを紹介しましたが、このプログラムは、`print` という名前の関数を呼び出す関数呼び出しです。`print` は、引数として受け取ったデータを出力する組み込み関数です。

2.3.5 小数点以下の切り捨て

Perl では、除算をしたいときは `/` という演算子を使えばいいわけですが、整数の範囲での除算をしたい、というときには、この演算子だけでは不十分です。なぜなら、整数の範囲では割り切れない場合、`/` は、小数点以下も求めてしまうからです。

Perl で、整数の範囲での除算をしたいときは、`/` を使って除算をしたのち、`int` という組み込み関数を呼び出す必要があります。`int` は、引数として数値を受け取って、その小数点以下を切り捨てた結果を返す組み込み関数です。

```
  DB<1> p 30/7
4.28571428571429
```

```
  DB<2> p int(30/7)
4
```

2.3.6 文字列の長さ

文字列を構成している文字の個数は、その文字列の「長さ」(length) と呼ばれます。

length という組み込み関数は、引数として文字列を受け取って、その長さを戻り値として返します。

```
  DB<1> p length("umiushi")
7
```

2.3.7 部分文字列の取り出し

文字列の一部分になっている文字列は、「部分文字列」(substring) と呼ばれます。

substr という組み込み関数は、文字列から部分文字列を取り出して、取り出した部分文字列を戻り値として返します。

substr は、次の 3 個の引数を受け取ります。

1 個目　文字列
2 個目　取り出す部分文字列の先頭の位置（先頭の文字を 0 番目と数える）
3 個目　取り出す部分文字列の長さ

```
  DB<1> p substr("ABCDEFG", 2, 3)
CDE
```

3 個目の引数を省略すると、指定された位置から末尾までの部分文字列を返します。

```
  DB<1> p substr("ABCDEFG", 2)
CDEFG
```

2.3.8 部分文字列の位置

index という組み込み関数は、引数として 2 個の文字列を受け取って、1 個目の文字列の中に部分文字列として含まれている 2 個目の文字列を、左から右に向かって検索して、最初に見つかった位置を戻り値として返します。2 個目の文字列が部分文字列として含まれていない場合は、−1 を返します。

```
  DB<1> p index("ABCDEFG", "CDE")
2
  DB<2> p index("ABCDEFG", "XYZ")
-1
```

3 個目の引数としてプラスの整数を渡すと、その位置から検索を開始します。

```
  DB<1> p index("ABCDEFGABCDEFG", "CDE", 3)
9
```

2.3.9 文字コード

ord という組み込み関数は、引数として文字列を受け取って、その先頭の文字の文字コードを戻り値として返します。

```
  DB<1> p ord("A")
65
```

chr という組み込み関数は、引数としてプラスの整数を受け取って、その整数を文字コードとする文字だけから構成される文字列を戻り値として返します。

```
  DB<1> p chr(65)
A
```

2.4 変数

2.4.1 変数の基礎

Perl のプログラムは、しばしば、データを入れることのできる箱を扱います。そのような箱は、「変数」(variable) と呼ばれます。

変数という箱にデータを入れることは、変数にデータを「代入する」(assign) と言われます。

変数は、それに与えられた名前によって識別されます。変数に与えることのできる名前は、「変数名」(variable name) と呼ばれます。

変数名は、式として評価することができます。変数名を評価すると、その変数名が与えられている変数に格納されているデータが、その値として得られます。

2.4.2 変数の種類

Perl の変数には、次のような種類があります。

- スカラー変数 (scalar variable)
- 配列 (array)
- ハッシュ変数 (hash variable)

スカラー変数というのは、数値または文字列を格納することのできる変数のことです。スカラー変数は、ただ単に「変数」と呼ばれることもあります。

この節で説明するのは、スカラー変数だけです。配列については第 5.1.3 項で、ハッシュ変数については第 6.1.2 項で説明することにしたいと思います。

2.4.3 識別子

変数名は、

ファニー文字	識別子

という構文を持っています。

「ファニー文字」(funny character) というのは、変数の種類を識別するための文字のことで、変数のそれぞれの種類は、次のようなファニー文字によって識別されます。

`$` スカラー変数

`@` 配列

`%` ハッシュ変数

「識別子」(identifier) というのは、次のような規則によって作られる文字の並びのことです。

- 識別子を作るために使うことのできる文字は、半角の英数字とアンダースコア (_) です。
- 識別子の先頭の文字として、数字を使うことはできません。
- 識別子の長さは、1 文字以上 251 文字以下でないといけません。

識別子として使うことのできるものの例としては、次のようなものがあります。

```
a  A  namako  _  _a  a8
```

他方、次のようなものを識別子として使うことはできません。

`m&a` 使うことのできない文字を含んでいる。

`8a` 先頭の文字が数字。

小文字と大文字とは区別されますので、たとえば `a` と `A` は、異なる識別子とみなされます。

スカラー変数のファニー文字はドルマーク (`$`) ですから、スカラー変数の変数名は、

```
$namako
```

というような、識別子の左側にドルマークを書いたものになります。

2.4.4 代入演算子

変数にデータを代入したいときは、「代入演算子」(assignment operator) と呼ばれる演算子が使われます。

代入演算子は、すべて二項演算子で、ほとんどすべての演算子よりも低い優先順位を持っています。

2.4.5 単純代入演算子

= という演算子は、「単純代入演算子」(simple assignment operator) と呼ばれます。これは、代入演算子のうちで、もっとも単純な動作をするものです。

= を含む式は、

変数名 = 式

と書きます。この形の式を評価すると、変数名を名前として持つ変数に、式の値が代入されます。= を含む式の値は、変数に代入されたデータです。

それでは、単純代入演算子を使って変数にデータを代入して、そののち変数の内容を調べてみましょう。

```
DB<1> $a = 33

DB<2> p $a
33
```

代入演算子の前と後ろには、この例のように 1 個の空白を書くのが普通ですが、それらの空白は絶対に必要というわけではありません。

スカラー変数には、文字列を代入することもできます。

```
DB<1> $a = "namako"

DB<2> p $a
namako
```

2.4.6 複合代入演算子

+=、-=、*=、/=、%= などの演算子は、「複合代入演算子」(compound assignment operator) と呼ばれます。

複合代入演算子を含む式は、= を含む式と同じように、

変数名 複合代入演算子 式

と書きます。この形の式を評価すると、変数名を名前として持つ変数の内容と式の値に対して演算が実行されて、その結果が変数に代入されます。複合代入演算子を含む式の値は、変数に代入されたデータです。

ところで、「変数の内容と式の値に対して演算が実行されて」と書きましたが、ここで実行される「演算」というのは、いったい何なのでしょうか。

複合代入演算子は、すべて、イコール (=) の左側に何らかの二項演算子を書いた形になっています。変数の内容と式の値に対して実行される演算は、イコールの左側に書かれた二項演算子があらわしている演算です。つまり、☆が二項演算子だとすると、

a ☆= b

という式は、

a = a ☆ b

という式と同じ意味になるということです。たとえば、

```
$a += 8
```

という式は、

```
$a = $a + 8
```

という式と同じ意味になります。

それでは、複合代入演算子を使って、変数の内容を変化させてみましょう。

```
DB<1> $a = 7

DB<2> $a += 8

DB<3> p $a
```

```
15
```

2.4.7 インクリメント演算子とデクリメント演算子

代入演算子を使うことによって、変数の内容を変化させることができるわけですが、変数の内容を変化させる演算子は、代入演算子だけではありません。「インクリメント演算子」(increment operator) と呼ばれる ++ という演算子と、「デクリメント演算子」(decrement operator) と呼ばれる -- という演算子も、変数の内容を変化させます。

変数に格納されている数値を 1 だけ増加させることを、変数を「インクリメントする」(increment) と言います。そして、変数に格納されている数値を 1 だけ減少させることを、変数を「デクリメントする」(decrement) と言います。

インクリメント演算子は、変数をインクリメントする演算子で、デクリメント演算子は、変数をデクリメントする演算子です。

インクリメント演算子とデクリメント演算子は、どちらも単項演算子で、それぞれ、前置演算子と後置演算子の両方があります。つまり、次のような 4 個の演算子があるということです。

- 前置インクリメント演算子 (prefix increment operator)
- 後置インクリメント演算子 (postfix increment operator)
- 前置デクリメント演算子 (prefix decrement operator)
- 後置デクリメント演算子 (postfix decrement operator)

2.4.8 前置インクリメント演算子

前置インクリメント演算子を含む式は、

```
++変数名
```

と書きます。この形の式を評価すると、変数名を名前として持つ変数がインクリメントされます。前置インクリメント演算子を含む式の値は、インクリメントされた後の変数の内容です。

```
  DB<1> $a = 7

  DB<2> p ++$a
8
  DB<3> p $a
8
```

2.4.9 後置インクリメント演算子

後置インクリメント演算子を含む式は、

```
変数名++
```

と書きます。前置インクリメント演算子と後置インクリメント演算子とで違うのは、式の値です。後置インクリメント演算子を含む式の値は、インクリメントされる前の変数の内容です。

```
  DB<1> $a = 7

  DB<2> p $a++
7
  DB<3> p $a
8
```

2.4.10 前置デクリメント演算子

前置デクリメント演算子を含む式は、

```
--変数名
```

と書きます。この形の式を評価すると、変数名を名前として持つ変数がデクリメントされます。前置デクリメント演算子を含む式の値は、デクリメントされた後の変数の内容です。

```
  DB<1> $a = 7

  DB<2> p --$a
6
  DB<3> p $a
6
```

2.4.11 後置デクリメント演算子

後置デクリメント演算子を含む式は、

変数名--

と書きます。デクリメントされる前の変数の内容が式の値になります。

```
  DB<1> $a = 7

  DB<2> p $a--
7
  DB<3> p $a
6
```

2.4.12 変数の展開

二重引用符文字列リテラルの中に変数名を書いたものを評価すると、その変数名は、その名前で指定された変数の内容に展開されます。

それでは、REPL を使って試してみましょう。

```
  DB<1> $a = "namako"

  DB<2> p "I am a $a."
I am a namako.
```

このように、二重引用符文字列リテラルの中では、ドルマーク ($) は変数名の先頭と解釈されますので、ドルマークを含む文字列を、二重引用符文字列リテラルを使って求めたい場合は、ドルマークの左側にバックスラッシュ(\) を書く必要があります。

```
  DB<1> p "I am a \$a."
I am a $a.
```

一重引用符文字列リテラルの中では、変数名は変数の内容に展開されません。ですから、一重引用符文字列リテラルを使えば、ドルマークの左側にバックスラッシュを書かなくても、ドルマークを含む文字列を求めることができます。

```
  DB<1> p 'I am a $a.'
I am a $a.
```

2.5 標準入力からの読み込み

2.5.1 標準入力とは何か

プログラムは、「標準入力」(standard input) と呼ばれるものからデータを読み込むことができます。標準入力は、デフォルトではキーボードに割り当てられていますが、プログラムを起動するコマンドによって、任意のファイルに変更することができます。

2.5.2 標準入力からデータを読み込む式

Perl では、

```
<STDIN>
```

という式は、標準入力から 1 行の文字列を読み込んで、その文字列を自分の値にします。ですから、

```
$a = <STDIN>
```

という式を評価すると、標準入力から読み込んだ文字列が$aという変数に代入されます。

プログラムの例 stdin.pl

```
print "文字列を入力してください。: ";
$a = <STDIN>;
print "あなたが入力した文字列は「" . $a . "」です。\n";
```

実行例

```
文字列を入力してください。: Tomoe Mami
あなたが入力した文字列は「Tomoe Mami
```

」です。

2.5.3 改行の除去

先ほどのプログラムを実行した結果から分かるとおり、<STDIN> の値として得られる文字列は、その末尾が改行になっています。もしも文字列の末尾にある改行が不要な場合は、それを除去する必要があります。

chomp という組み込み関数は、文字列の末尾にある改行を除去します。たとえば、

```
chomp($a)
```

という式を評価することによって、$a という変数に格納されている文字列の末尾にある改行を除去することができます。

プログラムの例 chomp.pl

```
print "文字列を入力してください。: ";
$a = <STDIN>;
chomp($a);
print "あなたが入力した文字列は「" . $a . "」です。\n";
```

実行例

```
文字列を入力してください。: Momoe Nagisa
あなたが入力した文字列は「Momoe Nagisa」です。
```

2.6 コンテキスト

2.6.1 コンテキストの基礎

第 1.3.2 項で説明したように、式を評価すると、その式の値が得られるわけですが、Perl では、式の値というのは、「コンテキスト」(context) と呼ばれるものによって変化します。

コンテキストというのは、式が書かれている文脈のことです。つまり、評価される式がどのような式の一部分として書かれているのかということです。

2.6.2 数値コンテキスト

式を数値として評価するコンテキストは、「数値コンテキスト」(numeric context) と呼ばれます。

数値コンテキストで、文字列が代入されている変数の名前や文字列リテラルを評価すると、その文字列があらわしている数値が、値として得られます。

算術演算子の前後に書かれた式は、数値コンテキストで評価されます。

```
  DB<1> p 700 + "65"
765
```

2.6.3 文字列コンテキスト

式を文字列として評価するコンテキストは、「文字列コンテキスト」(string context) と呼ばれます。

文字列コンテキストで、数値が代入されている変数の名前や数値リテラルを評価すると、その数値を文字列に変換した結果が、値として得られます。

文字列を連結する演算子の前後に書かれた式は、文字列コンテキストで評価されます。

```
  DB<1> p 700 . "65"
70065
```

プログラムの例 sanbai.pl

```
print "数値を入力してください。: ";
$a = <STDIN>;
chomp($a);
print $a . "の 3 倍は" . $a*3 . "です。\n";
```

実行例

```
数値を入力してください。: 7
7 の 3 倍は 21 です。
```

第 3 章　選択

3.1　選択の基礎

3.1.1　選択とは何か

第 1.3.4 項で説明したように、Perl のプログラムの中には、文を、いくつでも並べて書くことができて、コンピュータはそれらの文を、原則として、先頭から末尾に向かって 1 回ずつ実行していきます。

ですから、いくつかの文を書くことによって、まずこの動作を実行して、次にこの動作を実行して、次にこの動作を実行して……というように、複数の動作を直線的に実行していくという動作を記述することができるわけですが、コンピュータに実行させたい動作は、必ずしも一直線に進んでいくものばかりとは限りません。しばしば、そのときの状況に応じて、いくつかの動作の候補の中からひとつの動作を選んで実行するということも、必要になります。

「いくつかの動作の候補の中からひとつの動作を選んで実行する」という動作は、「選択」(selection) と呼ばれます。

3.1.2　条件

コンピュータは、運を天に任せて動作を選択するわけではありません。選択は、何らかの判断にもとづいて実行されます。

成り立っているか、それとも成り立っていないか、という判断の対象は、「条件」(condition) と呼ばれます。

条件が成り立っていると判断されるとき、その条件は「真」(true) であると言われます。逆に、条件が成り立っていないと判断されるとき、その条件は「偽」(false) であると言われます。

3.1.3　真偽値

真を意味するデータと、偽を意味するデータは、総称して「真偽値」(Boolean) と呼ばれます。

Perl は、「真偽値コンテキスト」(Boolean context) と呼ばれるコンテキストでは、式を評価した結果が次のようなデータだった場合、そのデータを、偽という真偽値に変換します。

- 数値のゼロ。
- ゼロ (`0`) という 1 個の文字だけから構成される文字列。
- 空文字列（長さが 0 の文字列）。
- 未定義値（「定義されていない」ということを意味する特殊なデータ）。

そして Perl は、式を評価した結果がこれら以外のデータだった場合、そのデータを、真という真偽値に変換します。

3.2　比較演算子

3.2.1　比較演算子の基礎

二つのデータのあいだに何らかの関係があるという条件が成り立っているかどうかを調べる二項演算子は、「比較演算子」(comparison operator) と呼ばれます。

比較演算子の優先順位は、加算や乗算などの演算子よりも低くなっています。

比較演算子を含む式を評価すると、演算子の左右にある式が評価されて、それらの式の値のあいだに関係が成り立っているかどうかという判断が実行されます。そして、コンテキストが真偽値コンテキストの場合、関係が成り立っているならば真、成り立っていないならば偽が、式全体の値になります。

比較演算子を含む式を文字列コンテキストで評価すると、関係が成り立っている場合は `1` という文字列、成り立っていない場合は空文字列が、値として得られます。

3.2.2　数値の大小関係

次の比較演算子を使うことによって、数値の大小関係について調べることができます。

a > b　　a は b よりも大きい。
a < b　　a は b よりも小さい。
a >= b　　a は b よりも大きいか、または a と b とは等しい。
a <= b　　a は b よりも小さいか、または a と b とは等しい。

```
DB<1> p 8 > 5
1
DB<2> p 5 > 8

DB<3> p 5 > 5

DB<4> p 5 >= 5
1
```

3.2.3　文字列の大小関係

大小関係があるのは、数値と数値とのあいだだけではありません。文字列と文字列とのあいだにも大小関係があります。

辞書の見出しは、「辞書式順序」(lexicographical order) と呼ばれる順序で並べられています。文字列と文字列とのあいだの大小関係は、辞書式順序で文字列を並べたときに、後ろにあるものは前にあるものよりも大きい、という関係です。

次の比較演算子を使うことによって、文字列の大小関係について調べることができます。

a gt b　　a は b よりも大きい (greater than)。
a lt b　　a は b よりも小さい (less than)。
a ge b　　a は b よりも大きいか、または a と b とは等しい (greater than or equal to)。
a le b　　a は b よりも小さいか、または a と b とは等しい (less than or equal to)。

```
DB<1> p "stay" gt "star"
1
DB<2> p "star" gt "stay"

DB<3>
```

3.2.4　数値が等しいかどうか

二つの数値が等しいかどうかということは、次の比較演算子を使うことによって調べることができます。

a == b　　a と b とは等しい。
a != b　　a と b とは等しくない。

```
DB<1> p 5 == 5
1
DB<2> p 5 == 8

DB<3>
```

3.2.5　文字列が等しいかどうか

二つの文字列が等しいかどうかということは、次の比較演算子を使うことによって調べることができます。

a eq b　　a と b とは等しい (equal to)。
a ne b　　a と b とは等しくない (not equal to)。

```
DB<1> p "star" eq "star"
1
DB<2> p "star" eq "stay"

DB<3>
```

3.3 if 文

3.3.1 if 文の基礎

何らかの条件が成り立っているかどうかを調べて、その結果にもとづいて二つの動作のうちのどちらかを実行したい、というときは、「if 文」(if statement) と呼ばれる文を書きます。

if 文は、

```
if (式) ブロック1 else ブロック2
```

と書きます。

if 文の中には、「ブロック」(block) と呼ばれるものを書く必要があります。ブロックというのは、文の列を中括弧 ({}) で囲んだもののことです。たとえば、

```
{ print "namako"; print "umiushi"; print "hitode"; }
```

というようなものがブロックです。

if 文を実行すると、まず最初に、丸括弧の中に書かれた式が真偽値コンテキストで評価されます。そして、その式の値が真だった場合は、ブロック $_1$ が実行されます（その場合、ブロック $_2$ は実行されません)。式の値が偽だった場合は、ブロック $_2$ が実行されます（その場合、ブロック $_1$ は実行されません)。

つまり、 if 文は、「もしも条件が真ならばブロック $_1$ を実行して、そうでなければブロック $_2$ を実行する」という動作を意味しているわけです。

それでは、 if 文を REPL に入力してみましょう。

```
  DB<1> if (8 > 5) { print "namako"; } else { print "hitode"; }
namako
```

この場合、丸括弧の中に書かれた式の値は真ですので、namako が出力されて、hitode は出力されません。

```
  DB<1> if (5 > 8) { print "namako"; } else { print "hitode"; }
hitode
```

この場合、丸括弧の中に書かれた式の値は偽ですので、hitode が出力されて、namako は出力されません。

次のプログラムは、整数を読み込んで、それが偶数なのか奇数なのかということを調べて、その結果を出力します。

プログラムの例 evenodd.pl

```
print "整数を入力してください。: ";
$n = <STDIN>;
chomp($n);
print $n . "は";
if ($n%2 == 0) {
    print "偶数";
} else {
    print "奇数";
}
print "です。\n";
```

実行例

```
整数を入力してください。: 14
14 は偶数です。

整数を入力してください。: 15
15 は奇数です。
```

3.3.2 インデント

プログラムの中にブロックを書く場合は、通常、先ほどのプログラムのように、改行や空白を使って、

```
{
        文
        文
        ●
        ●
        ●
}
```

というような形にブロックを整形します。

このように、何個かの空白を先頭に挿入することによって文を右にずらすことを、文を「インデントする」(indent) と言います。

一定の個数の空白でブロックの中の文をインデントすると、ブロックの範囲が一目瞭然になりますので、プログラムが読みやすくなります。また、ブロックが何重にも入れ子になっている場合、入れ子の深さも、どれだけインデントされているかということによって視覚化されます。

3.3.3 else 以降を省略した if 文

二つの動作のうちのどちらかを選択するのではなくて、ひとつの動作を実行するかしないかを選択したい、ということもしばしばあります。そのような場合は、else 以降を省略した if 文を書きます。つまり、

```
if (式) ブロック
```

という形の if 文を書くわけです。この形の if 文を実行すると、式の値が真の場合はブロックが実行されますが、式の値が偽の場合は何も実行されません。

REPL を使って試してみましょう。

```
  DB<1> if (8 > 5) { print "namako"; }
namako
  DB<2> if (5 > 8) { print "namako"; }

  DB<3>
```

次のプログラムは、分を単位とする時間の長さを読み込んで、それを「何時間何分」という形式の文字列に変換した結果を出力します。ただし、「何分」という端数が出ない場合は、「何時間」という形式の文字列を出力します。

プログラムの例 mtohm.pl

```
print "時間の長さを分で入力してください。: ";
$m = <STDIN>;
chomp($m);
print $m . "分は" . int($m/60) . "時間";
if ($m%60 != 0) {
    print $m%60 . "分";
}
print "です。\n";
```

実行例

```
時間の長さを分で入力してください。: 160
160 分は 2 時間 40 分です。

時間の長さを分で入力してください。: 180
180 分は 3 時間です。
```

「何時間」の部分の右側に「何分」の部分を連結するという動作は、実行するかしないかを選択することになりますので、このように、else 以降を省略した if 文を使って書くことができます。

3.3.4 多肢選択

選択の対象となる動作が 3 個以上あるような選択は、「多肢選択」(multibranch selection) と呼ばれます。

多肢選択を記述したいときは、if 文の中に、

elsif（式）ブロック

という形のものをいくつか書きます。この中の elsif というのは、else if を縮めたもので、「そうではなくてもしも……ならば」ということを意味しています。

たとえば、2 個の条件による多肢選択は、

if（式 $_1$）ブロック $_1$ elsif（式 $_2$）ブロック $_2$ else ブロック $_3$

という形の if 文を書くことによって記述することができます。この形の if 文を実行すると、まず式 $_1$ が評価されます。もしもその値が真だったならば、ブロック $_1$ が実行されます。その場合、式 $_2$ は評価されません。式 $_1$ の値が偽だったならば、式 $_2$ が評価されます。もしもその値が真だったならば、ブロック $_2$ が実行されます。式 $_2$ の値も偽だったならば、ブロック $_3$ が実行されます。

次のプログラムは、数値を読み込んで、それがプラスなのかマイナスなのかゼロなのかということを出力します。

プログラムの例 sign.pl

```
print "数値を入力してください。: ";
$a = <STDIN>;
chomp($a);
print $a . "は";
if ($a > 0) {
    print "プラス";
} elsif ($a < 0) {
    print "マイナス";
} else {
    print "ゼロ";
}
print "です。\n";
```

実行例

```
数値を入力してください。: 5
5 はプラスです。

数値を入力してください。: -5
-5 はマイナスです。

数値を入力してください。: 0
0 はゼロです。
```

3.4 論理演算子

3.4.1 論理演算子の基礎

処理の対象が真偽値で、処理の結果も真偽値であるような動作をあらわしている演算子は、「論理演算子」(logical operator) と呼ばれます。

Perl には、次のような論理演算子があります。

a and b　a かつ b である。

a or b　a または b である。

not a　a ではない。

これらの演算子は、代入演算子よりも低い優先順位を持っています。そして、これらの演算子のあいだでは、not よりも and のほうが低い優先順位を持っていて、and よりも or のほうが低い優先順位を持っています。

3.4.2 論理積演算子

and は、「論理積演算子」(logical AND operator) と呼ばれます。これは、二つの条件が両方とも成り立っているかどうかを判断したいとき、つまり、A かつ B という条件が成り立っているかどうかを判断したいときに使われる論理演算子で、次のような動作をします。

真　and　真　　⟶　真
真　and　偽　　⟶　偽
偽　and　真　　⟶　偽
偽　and　偽　　⟶　偽

and の右側に書かれた式は、必ず評価されるとは限りません。右側の式が評価されるのは、左側の式の値が真だったときだけです。

3.4.3　論理和演算子

or は、「論理和演算子」(logical OR operator) と呼ばれます。これは、二つの条件のうちの少なくとも一つが成り立っているかどうかを判断したいとき、つまり、A または B という条件が成り立っているかどうかを判断したいときに使われる論理演算子で、次のような動作をします。

真　or　真　　⟶　真
真　or　偽　　⟶　真
偽　or　真　　⟶　真
偽　or　偽　　⟶　偽

プログラムの例　leapyear.pl

```
print "年を西暦で入力してください。: ";
$y = <STDIN>;
chomp($y);
print $y . "年はうるう年";
if ($y%4 == 0 and $y%100 != 0 or $y%400 == 0) {
    print "です。\n";
} else {
    print "ではありません。\n";
}
```

実行例

```
年を西暦で入力してください。: 2080
2080 年はうるう年です。

年を西暦で入力してください。: 2100
2100 年はうるう年ではありません。

年を西暦で入力してください。: 2400
2400 年はうるう年です。
```

and の場合と同じように、 or の右側に書かれた式も、必ず評価されるとは限りません。右側の式が評価されるのは、左側の式の値が偽だったときだけです。

3.4.4　論理否定演算子

not は、「論理否定演算子」(logical negation operator) と呼ばれます。これは、真偽値を反転させたいとき、つまり、A ではないという条件が成り立っているかどうかを判断したいときに使われる論理演算子で、次のような動作をします。

not 真　　⟶　偽
not 偽　　⟶　真

第 4 章　繰り返し

4.1 繰り返しの基礎

4.1.1 繰り返しとは何か

コンピュータに実行させたい動作は、必ずしも、一連の動作をそれぞれ一回ずつ実行していけばそれで達成される、というものばかりとは限りません。しばしば、ほとんど同じ動作を何回も何十回も何百回も実行しなければ意図していることを達成できない、ということがあります。

「同じ動作を何回も実行する」という動作は、「繰り返し」(iteration) と呼ばれます。

この章では、繰り返しというのはどのように記述すればいいのか、ということについて説明します。

4.1.2 繰り返しを記述するための文

繰り返しは、繰り返したい回数と同じ個数の文を書くことによって記述することも可能ですが、そのような書き方だと、繰り返しの回数に比例してプログラムが長くなってしまいます。ですから、多くのプログラミング言語は、繰り返しを簡潔に記述することができるようにする機能を持っています。

Perl では、繰り返しを簡潔に記述するための文として、次のようなものがあります。

- `while` 文 (`while` statement)
- `for` 文 (`for` statement)
- `foreach` 文 (`foreach` statement)

これらの文のうちで、この章で紹介するのは、`while` 文と `for` 文の二つだけです。`foreach` 文については、第 5.4 節で説明することにしたいと思います。

4.2 `while` 文

4.2.1 `while` 文の基礎

`while` 文は、

```
while（式）ブロック
```

と書きます。

`while` 文は、次のような動作をあらわしています。

(1) 丸括弧の中に書かれた式を真偽値コンテキストで評価する。その値が偽だった場合、`while` 文の動作は終了する。
(2) 式の値が真だった場合は、ブロックを実行する。
(3) (1) に戻って、ふたたび同じ動作を実行する。

4.2.2 無限ループ

繰り返しというのは、何らかの条件が成り立っているあいだだけ実行して、その条件が成り立たなくなったときに終了する、というのが普通です。しかし、永遠に終わらない繰り返しというものを記述することも可能です。永遠に終わらない繰り返しは、「無限ループ」(infinite loop) と呼ばれます。

`1` という整数リテラルは、真偽値コンテキストで評価すると、その値は常に真ですから、`while` 文の丸括弧の中にそれを書くと、その `while` 文は無限ループになります。

プログラムの例 `infinite.pl`

```
while (1) {
    print "Ctrl-C で終了します。\n";
}
```

このプログラムを実行すると、「`Ctrl-C` で終了します。」という文字列の出力が繰り返されます。実行したまま放置すると、繰り返しは無限に続きます。このプログラムを終了させたいときは、Ctrl-C、つまりコントロールキーを押しながら C のキーを押す、という操作をします。

4.2.3 自然数の出力

0、1、2、3、4、・・・という数は、「自然数」(natural number) と呼ばれます（0 を除外する場合もあります)。

それでは、すべての自然数を出力するプログラムを、 while 文を使って書いてみましょう。

すべての自然数を出力するためには、出力する自然数を入れる変数が必要です。そこで、 $i という名前の変数を使うことにしましょう。

あらかじめ初期値として $i に 0 を代入しておいて、

(1) $i の内容を出力する。

(2) $i をインクリメントする。

という動作を繰り返すと、すべての自然数が出力されることになります（ただし、本当にすべての自然数を出力するためには、無限の時間が必要です)。

プログラムの例 natural.pl

```
$i = 0;
while (1) {
    print $i . " ";
    $i++;
}
```

このプログラムも無限ループですので、Ctrl-C で終了させない限り、実行は永遠に終わりません。

4.2.4 有限の回数の繰り返し

while 文の丸括弧の中に 1 という整数リテラルを書くと、その繰り返しは無限ループになります。それに対して、 while 文の丸括弧の中に、繰り返しを開始する時点では値が真だけれども、繰り返しを続行していくと、いつかは値が偽になる、という式を書くと、その繰り返しは有限の回数で終了することになります。

たとえば、すべての自然数を出力する先ほどのプログラムを改造して、 while 文の丸括弧の中の式を、 1 という整数リテラルから、

```
$i <= 100
```

という式に書き換えたとすると、 $i という変数の内容が、0、1、2、3、4、・・・、100 と増加していくあいだは式の値が真ですが、101 になったところで式の値が偽になりますので、自然数は 100 までしか出力されないということになります。

次のプログラムは、自然数 n を読み込んで、0 から n までの自然数を出力します。

プログラムの例 finite.pl

```
print "自然数を入力してください。: ";
$n = <STDIN>;
$i = 0;
while ($i <= $n) {
    print $i . " ";
    $i++;
}
print "\n";
```

実行例

```
自然数を入力してください。: 20
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

4.3 for 文

4.3.1 for 文の基礎

for 文は、

for（式 1; 式 2; 式 3）ブロック

と書きます。

for 文は、次のような動作をあらわしています。

(1) 式 $_1$ を評価する。
(2) 式 $_2$ を真偽値コンテキストで評価する。その値が偽だった場合、for 文の動作は終了する。
(3) 式 $_2$ の値が真だった場合は、ブロックを実行する。
(4) 式 $_3$ を評価する。
(5) (2) に戻って、ふたたび同じ動作を実行する。

単純文と while 文を組み合わせることによって、for 文があらわしている動作を記述すると、次のようになります。

```
式1;
while (式2) {
  文の列
  式3;
}
```

4.3.2 while 文と for 文の使い分け

while 文と for 文は、どちらも繰り返しを記述するための文ですが、それらはどのように使い分ければいいのでしょうか。

while 文というのが繰り返しを記述するための汎用的な文であるのに対して、for 文というのは、ある特定のタイプの繰り返しを記述することを意識して作られた文です。

for 文が対象としている特定のタイプの繰り返しというのは、繰り返しのタイプとしては、きわめて頻繁に出現するものです。それは、次のような三つの動作によって制御される繰り返しです。

- 変数に初期値を代入する。
- 変数の内容を調べて、繰り返しを続行するかどうかを判断する。
- 変数の内容を変化させる。

たとえば、次の繰り返しも、この三つの動作によって繰り返しが制御されています。

```
$i = 0;
while ($i <= 100) {
    print $i . " ";
    $i++;
}
```

すなわち、$i = 0 という式で変数に初期値を代入して、$i <= 100 という式で、変数の内容を調べて、繰り返しを続行するかどうかを判断して、$i++ という式で、変数の内容を変化させているわけです。

この繰り返しを、for 文を使って記述すると、次のようになります。

```
for ($i = 0; $i <= 100; $i++) {
    print $i . " ";
}
```

while 文を使った場合と for 文を使った場合を比べてみると、for 文を使った場合のほうが読みやすくなっているはずです。その理由は、このようなタイプの繰り返しを while 文を使って記述した場合、それを制御する三つの動作が分散して書かれることになるのに対して、for 文を使って記述した場合は、それらの動作が一箇所にまとまって書かれることになるからです。

ですから、while 文と for 文は、for 文が対象としているタイプの繰り返しは for 文で記述して、それ以外のタイプの繰り返しは while 文で記述する、というように使い分けるのがいいということになります。

次のプログラムは、第 4.2.4 項で紹介した、自然数 n を読み込んで、0 から n までの自然数を出力するプログラムを、for 文を使って書き直したものです。

プログラムの例 for.pl

```
print "自然数を入力してください。: ";
$n = <STDIN>;
for ($i = 0; $i <= $n; $i++) {
```

```
    print $i . " ";
}
print "\n";
```

次のプログラムは、入力された数値を超えない範囲で、2^0、2^1、2^2、2^3、……を出力します。

プログラムの例 poweroftwo.pl

```
print "上限を入力してください。: ";
$n = <STDIN>;
for ($b = 1; $b <= $n; $b *= 2) {
    print $b . " ";
}
print "\n";
```

実行例

```
上限を入力してください。: 50000
1 2 4 8 16 32 64 128 256 512 1024 2048 4096 8192 16384 32768
```

第5章 リストと配列

5.1 リストと配列の基礎

5.1.1 リストとは何か

Perl では、式をコンマで区切って並べて、その全体を丸括弧で囲んだもの、つまり、

(式，式，･･･，式)

という形のものは、「リスト」(list) と呼ばれます。

「リストコンテキスト」(list context) と呼ばれるコンテキストでリストを評価すると、その値として、「リスト値」(list value) と呼ばれるデータが得られます。リスト値というのは、リストを構成しているそれぞれの式を先頭から順番に評価することによって得られたデータを、式が並んでいるのと同じ順序で並べてできる列のことです。たとえば、

(5，3，8，7，6)

というリストをリストコンテキストで評価すると、その値として、5、3、8、7、6 という数値をこの順序で並べたリスト値が得られます。

```
  DB<1> p (5, 3, 8, 7, 6)
53876
```

リスト値を構成しているそれぞれのデータは、そのリスト値の「要素」(element) と呼ばれます。

数値コンテキストと文字列コンテキストは、総称して「スカラーコンテキスト」(scalar context) と呼ばれます。

スカラーコンテキストでリストを評価すると、リストを構成しているそれぞれの式が先頭から順番に評価されるというところはリストコンテキストの場合と同じですが、その結果として得られるのは、リスト値ではなくて、最後の式の値だけです。

たとえば、スカラー変数にデータを代入する代入演算子の右側の式はスカラーコンテキストで評価されますので、そこにリストを書くとどうなるかを試してみましょう。

```
  DB<1> $a = (5, 3, 8, 7, 6)

  DB<2> p $a
6
```

この場合、評価されたリストの最後の式は 6 という整数リテラルですので、$a というスカラー変数には 6 という整数が代入されます。

今度は、関数呼び出しから構成されるリストを、スカラーコンテキストで評価してみましょう。

```
  DB<1> $a = ((print "ebi"), (print "tako"), (print "ika"))
ebitakoika
```

このように、スカラーコンテキストでリストを評価した場合も、リストを構成しているそれぞれの式は、先頭から順番に評価されます。

5.1.2 範囲演算子

Perl では、「範囲演算子」(range operator) と呼ばれる演算子を使うことによって、指定された範囲の中にあるすべてのデータから構成されるリスト値を生成することができます。

`..` というのが、範囲演算子です。この演算子は、その左側には範囲の下限を求める式、右側には範囲の上限を求める式を書きます。そして、その全体をリストコンテキストで評価すると、その範囲の中にあるすべての要素から構成されるリスト値が値として得られます。

```
  DB<1> p 0..9
0123456789
  DB<2> p "a".."z"
abcdefghijklmnopqrstuvwxyz
```

5.1.3 配列とは何か

リスト値を格納することのできる変数は、「配列」(array) と呼ばれます。

第 2.4.3 項で説明したように、変数名の先頭には、変数の種類を識別するために、「ファニー文字」と呼ばれる文字を書く必要があります。配列という変数の種類を識別するファニー文字は、アットマーク (`@`) です。

代入演算子を使うことによって、配列にリスト値を代入することができます。また、配列の変数名をリストコンテキストで評価すると、そこに格納されているリスト値が値として得られます。

```
  DB<1> @a = (5, 3, 8, 7, 6)

  DB<2> p @a
53876
```

5.1.4 配列の展開

スカラー変数の場合と同じように、二重引用符文字列リテラルの中に配列の変数名を書くと、そのリテラルが評価されるときに、その変数名は、そこに格納されているリスト値に展開されます。そのとき、リスト値の要素と要素とのあいだには、1 個の空白が挿入されます。

```
  DB<1> @a = (5, 3, 8, 7, 6)

  DB<2> p "@a"
5 3 8 7 6
```

5.1.5 添字

配列というのは、リスト値を構成しているそれぞれの要素を格納するための箱が一列に並んだものだと考えることができます。配列を構成しているそれぞれの箱は、その配列の「要素」(element) と呼ばれます。

配列のそれぞれの要素には、配列の先頭から順番に、0 から始まる番号が与えられています。ですから、要素の番号を指定することによって、配列の特定の要素を指定することができます。

番号を指定することによって配列の特定の要素を指定したいときは、

```
$変数名 [式]
```

という構文を持つ記述を書きます。この中の「変数名」のところには、配列の名前からファニー文字のアットマークを取り除いた部分を書きます。そして、角括弧 (`[]`) の中に書くのは、指定したい要素の番号を求める式です。たとえば、

```
$a[3]
```

という記述を書くことによって、`@a` という配列が持っている、3 という番号を持つ要素を指定することができます。

番号を指定することによって配列の特定の要素を指定する記述の中にある、

```
[式]
```

という部分は、「添字」(index) と呼ばれます。

番号を指定することによって配列の要素を指定する記述を式として評価すると、その値として、指定された要素の内容が得られます。

```
  DB<1> @a = (5, 3, 8, 7, 6)

  DB<2> p $a[3]
7
```

5.1.6 配列の要素への代入

配列の特定の要素にデータを代入したいときは、代入演算子の左側に、番号を指定することによって配列の要素を指定する記述を書きます。

```
  DB<1> @a = (5, 3, 8, 7, 6)

  DB<2> $a[2] = "namako"

  DB<3> p "@a"
5 3 namako 7 6
```

配列の要素への代入は、すでに存在する要素にしかできないというわけではありません。末尾の要素の番号よりも大きな番号を添字で指定してデータを代入すると、その番号までの要素が新しく作られます。その場合、それまで末尾だった要素と、データを代入した要素とのあいだの要素には、「未定義値」(undefined value) と呼ばれるものが代入されます。未定義値は、数値コンテキストでは 0 に、文字列コンテキストでは空文字列に、真偽値コンテキストでは偽に変換されます。

```
  DB<1> @a = (5, 3, 8, 7, 6)

  DB<2> $a[8] = "umiushi"

  DB<3> p "@a"
5 3 8 7 6    umiushi
```

5.1.7 リスト値と配列の長さ

リスト値を構成している要素の個数は、そのリスト値の「長さ」(length) と呼ばれます。同じように、配列を構成している要素の個数も、その配列の「長さ」と呼ばれます。

配列の変数名をスカラーコンテキストで評価すると、その値として、その配列の長さが得られます。

```
  DB<1> @a = "a".."z"

  DB<2> p $a = @a
26
```

式を強制的にスカラーコンテキストで評価したいときは、scalar という組み込み関数を使うこともできます。この関数は、引数を強制的にスカラーコンテキストで評価して、その値を戻り値として返します。

```
  DB<1> @a = "a".."z"

  DB<2> p scalar(@a)
26
```

5.1.8 リスト値の連結

リストの中に、数値または文字列を求める式ではなくて、リスト値を求める式を書くと、そのリスト値は、全体がひとつの要素になるのではなくて、個々の要素に展開されて、それらの要素がリスト値の一部分になります。つまり、そのようなリストの値は、要素としてリスト値を含むリスト値ではなくて、数値または文字列から構成されるリスト値だということです。たとえば、

```
(5, 3, 8, (2, 9, 0, 1, 4), 7, 6)
```

というリストをリストコンテキストで評価したときに得られるのは、長さが 6 のリスト値ではなくて、長さが 10 のリスト値です。

何個かのリスト値を連結したいときは、この原理を応用することができます。

```
  DB<1> @a = (5, 3, 2)

  DB<2> @b = (0, 4, 7)

  DB<3> @c = (9, 6, 1)

  DB<4> @d = (@a, @b, @c)

  DB<5> p "@d"
5 3 2 0 4 7 9 6 1
```

5.1.9 スカラー変数の名前のリスト

単純代入演算子 (=) の左側には、スカラー変数の名前から構成されるリストを書くことができます。その場合、=の右側に書かれた式はリストコンテキストで評価されて、その値として得られたリスト値を構成するそれぞれの要素が、それぞれの変数に代入されます。そして、それぞれの変数に代入されたデータから構成されるリスト値が、=を含む式の値になります。

```
  DB<1> p ($a, $b, $c, $d, $e) = (5, 3, 8, 7, 6)
53876
  DB<2> p "$a $b $c $d $e"
5 3 8 7 6
```

代入されるリスト値の長さが変数名のリストよりも長い場合、余った要素は無視されます。

```
  DB<1> p ($a, $b, $c) = (5, 3, 8, 7, 6)
538
  DB<2> p "$a $b $c"
5 3 8
```

代入されるリスト値の長さが変数名のリストよりも短い場合、余った変数には未定義値が代入されます。

```
  DB<1> ($a, $b, $c, $d, $e) = (9, 9, 9, 9, 9)

  DB<2> p ($a, $b, $c, $d, $e) = (5, 3, 8)
538
  DB<3> p "$a $b $c $d $e"
5 3 8
```

5.1.10 コマンドライン引数

Perl で書かれたプログラムを perl に実行させるコマンドには、プログラムのパス名の右側に、空白で区切られた単語を何個でも書くことができます。プログラムのパス名の右側に書かれたそれらの単語は、「コマンドライン引数」(command line argument) と呼ばれます。

コマンドライン引数は、@ARVG という名前の配列に格納されますので、その配列を使うことによって、プログラムの中でコマンドライン引数を利用することができます。

プログラムの例 commandline.pl

```
print "@ARGV\n";
```

実行例

```
$ perl commandline.pl namako umiushi hitode kamenote
namako umiushi hitode kamenote
```

5.2 リスト値や配列を処理する組み込み関数

5.2.1 リスト値の要素の連結

この節では、リスト値や配列を処理する組み込み関数を紹介したいと思います。

まず最初は、join という組み込み関数です。

join は、引数として文字列とリスト値を受け取って、リスト値を構成しているそれぞれの要素を、文字列で区切って並べることによってできる文字列を戻り値として返します。

```
  DB<1> p join(", ", (5, 3, 8, 7, 6))
5, 3, 8, 7, 6
```

5.2.2 配列への要素の追加

push という組み込み関数は、引数として配列とデータを受け取って、その配列の末尾にそのデータを追加します。戻り値は、追加したあとの配列の長さです。

```
  DB<1> @a = (5, 3, 8, 7, 6)

  DB<2> p push(@a, 2)
6
  DB<3> p "@a"
5 3 8 7 6 2
```

2個目の引数としてリスト値を渡すと、そのリスト値を構成しているすべての要素が配列に追加されます。

```
  DB<1> @a = (5, 3, 8, 7, 6)

  DB<2> p push(@a, (9, 1, 2, 0, 4))
10
  DB<3> p "@a"
5 3 8 7 6 9 1 2 0 4
```

unshift という組み込み関数は、引数として配列とデータを受け取って、その配列の先頭にそのデータを追加します。戻り値は、追加したあとの配列の長さです。

```
  DB<1> @a = (5, 3, 8, 7, 6)

  DB<2> p unshift(@a, 2)
6
  DB<3> p "@a"
2 5 3 8 7 6
```

5.2.3 配列の要素の削除

pop という組み込み関数は、引数として配列を受け取って、その配列の末尾の要素を削除して、その要素に格納されていたデータを戻り値として返します。

```
  DB<1> @a = (5, 3, 8, 7, 6)

  DB<2> p pop(@a)
6
  DB<3> p "@a"
5 3 8 7
```

shift という組み込み関数は、引数として配列を受け取って、その配列の先頭の要素を削除して、その要素に格納されていたデータを戻り値として返します。

```
  DB<1> @a = (5, 3, 8, 7, 6)

  DB<2> p shift(@a)
5
  DB<3> p "@a"
3 8 7 6
```

5.2.4 リスト値を逆順にする

reverse という組み込み関数は、引数としてリスト値を受け取って、それを構成しているそれぞれの要素を逆順に並べ替えることによってできるリスト値を戻り値として返します。

```
  DB<1> p reverse "a".."z"
zyxwvutsrqponmlkjihgfedcba
```

5.2.5 リスト値のソート

sort という組み込み関数は、引数としてリスト値を受け取って、それを構成しているそれぞれの要素を、辞書の順序で並べ替えることによってできるリスト値を戻り値として返します。

```
  DB<1> @a = sort ("set", "fun", "cup", "cat", "can")

  DB<2> p "@a"
can cat cup fun set
  DB<3> @a = sort (6, 30, 8, 200, 4)

  DB<4> p "@a"
200 30 4 6 8
```

数値から構成されるリストを、小さな数値から大きな数値へという順序で並べ替えたいときは、1 個目の引数として、

```
{ $a <=> $b }
```

というブロックを渡して、2 個目の引数としてリストを渡します。

```
  DB<1> @a = sort { $a <=> $b } (6, 30, 8, 200, 4)

  DB<2> p "@a"
4 6 8 30 200
```

$a と $b の順番を逆にした、

```
{ $b <=> $a }
```

というブロックを 1 個目の引数として渡すと、大きな数値から小さな数値へという順序になります。

```
  DB<1> @a = sort { $b <=> $a } (6, 30, 8, 200, 4)

  DB<2> p "@a"
200 30 8 6 4
```

5.2.6 部分配列の置き換え

配列の連続する一部分は、その配列の「部分配列」(subarray) と呼ばれます。

splice という組み込み関数は、部分配列の内容を別のリスト値に置き換えます。

splice は、次の 4 個の引数を受け取ります。

1 個目　配列
2 個目　部分配列の先頭になる要素の番号
3 個目　部分配列の長さ
4 個目　リスト値

splice の戻り値は、置き換わる前の部分配列に格納されていたリスト値です。

```
  DB<1> @a = "a".."g"

  DB<2> p splice(@a, 2, 4, (5, 3, 7, 6))
cdef
  DB<3> p "@a"
a b 5 3 7 6 g
```

部分配列の長さと、リスト値の長さは、同じでなくてもかまいません。リスト値のほうが長い場合は新しい要素が挿入され、リスト値のほうが短い場合は不要な要素が削除されます。

```
  DB<1> @a = "a".."g"

  DB<2> p splice(@a, 2, 4, (5, 3, 7, 6, 1, 0, 9, 8))
cdef
  DB<3> p "@a"
a b 5 3 7 6 1 0 9 8 g
  DB<4> @a = "a".."g"

  DB<5> p splice(@a, 2, 4, (5, 3))
cdef
  DB<6> p "@a"
a b 5 3 g
```

4 個目の引数を省略すると、指定された部分配列が削除されます。

```
  DB<1> @a = "a".."g"

  DB<2> p splice(@a, 2, 4)
cdef
  DB<3> p "@a"
a b g
```

3個目と4個目の引数を省略すると、2個目の引数で指定された要素から末尾の要素までが削除されます。

```
  DB<1> @a = "a".."g"

  DB<2> p splice(@a, 2)
cdefg
  DB<3> p "@a"
a b
```

2個目と3個目と4個目の引数を省略すると、すべての要素が削除されます。

```
  DB<1> @a = "a".."g"

  DB<2> p splice(@a)
abcdefg
  DB<3> p scalar(@a)
0
```

5.3 配列のスライス

5.3.1 配列のスライスの基礎

配列から取り出した要素を並べることによってできる配列を、もとの配列の「スライス」(slice)と呼びます。

5.3.2 配列のスライスの記述

配列のスライスは、

配列変数名 [式$_1$, 式$_2$, ･･･, 式$_n$]

という記述を書くことによって求めることができます。この中の「配列変数名」のところには、スライスを求めたい配列の名前を書きます。そして、式$_1$から式$_n$までのところには、配列から取り出したい要素の番号を求める式を書きます。たとえば、

```
@a[2, 0, 19]
```

という記述は、@aという名前の配列から、2番、0番、19番という番号の要素を取り出して、それらの要素をその順番で並べたスライスを求めます。

```
  DB<1> @a = "a".."z"

  DB<2> p @a[2, 0, 19]
cat
```

配列のスライスは、範囲演算子を使った、

配列変数名 [下限.. 上限]

という記述を書くことによっても求めることができます。この場合は、下限から上限までの番号を持つ要素から構成されるスライスが得られます。

```
  DB<1> @a = "a".."z"

  DB<2> p @a[14..20]
opqrstu
```

5.3.3 配列のスライスへの代入

代入演算子の左側に配列のスライスの記述を書いて、右側にリスト値を求める式を書くと、リスト値が配列のスライスに代入されます。

```
DB<1> @a = 0..10

DB<2> @a[6, 8, 1, 4] = "a".."d"

DB<3> p "@a"
0 c 2 3 d 5 a 7 b 9 10
```

配列のスライスの長さよりも代入するリスト値の長さのほうが短い場合は、リスト値の末尾に未定義値の要素が追加されます。

```
DB<1> @a = 0..10

DB<2> @a[6, 8, 1, 4] = "a".."c"

DB<3> p "@a"
0 c 2 3  5 a 7 b 9 10
```

配列のスライスの長さよりも代入するリスト値の長さのほうが長い場合、余分な要素は無視されます。

```
DB<1> @a = 0..10

DB<2> @a[6, 8, 1, 4] = "a".."g"

DB<3> p "@a"
0 c 2 3 d 5 a 7 b 9 10
```

配列のスライスから求めたリスト値を、同じ配列の異なるスライスに代入することによって、要素の内容を入れ替える、ということも可能です。

```
DB<1> @a = "a".."g"

DB<2> @a[1, 5] = @a[5, 1]

DB<3> p "@a"
a f c d e b g
```

5.4 foreach 文

5.4.1 foreach 文の基礎

foreach 文は、すでに第 4.1.2 項で説明したように、while 文や for 文と同じように、繰り返しを記述するための文です。

foreach 文を使うことによって記述することができるのは、リスト値または配列について、それを構成しているそれぞれの要素に対する処理を繰り返す、という動作です。

5.4.2 リスト値の繰り返し

リスト値を構成しているそれぞれの要素に対する繰り返しを実行する foreach 文は、

```
foreach スカラー変数名 リスト ブロック
```

と書きます。この中の「スカラー変数名」のところには、スカラー変数の名前を書きます。そして、「リスト」のところには、リストを書きます。

このような foreach 文は、リストの値として得られたリスト値を構成しているそれぞれの要素について、変数名をその要素の名前にしてブロックを実行する、という繰り返しをあらわしています。

```
DB<1> foreach $a (5, 3, 8, 7, 6) { print "[$a]" }
[5][3][8][7][6]
```

foreach 文で使われるスカラー変数の名前は、スカラー変数ではなくてリスト値の要素に与えられます。ですから、次のような foreach 文はエラーになります。

```
foreach $a (5, 3, 8, 7, 6) { $a = 4 }
```

foreach 文は、範囲演算子を使って、

```
foreach スカラー変数名 (下限.. 上限) ブロック
```

と書くこともできます。

```
  DB<1> foreach $a (10..20) { print "[$a]" }
[10][11][12][13][14][15][16][17][18][19][20]
```

5.4.3　配列の繰り返し

配列を構成しているそれぞれの要素に対する繰り返しを実行する foreach 文は、

```
foreach スカラー変数名 (配列変数名) ブロック
```

と書きます。この中の「スカラー変数名」のところには、スカラー変数の名前を書きます。そして、「配列変数名」のところには、配列の変数名を書きます。

このような foreach 文は、配列を構成しているそれぞれの要素について、変数名をその要素の名前にしてブロックを実行する、という繰り返しをあらわしています。

```
  DB<1> @a = (5, 3, 8, 7, 6)

  DB<2> foreach $a (@a) { print "[$a]" }
[5][3][8][7][6]
```

次のプログラムは、5 個の文字列を読み込んで、そののちそれらを出力します。

プログラムの例　foreach.pl

```
@a = ();
for ($i = 1; $i <= 5; $i++) {
    print $i . "個目の文字列を入力してください。: ";
    $a = <STDIN>;
    chomp($a);
    push(@a, $a);
}
foreach $a (@a) {
    print $a . " ";
}
print "\n";
```

実行例

```
1 個目の文字列を入力してください。: namako
2 個目の文字列を入力してください。: umiushi
3 個目の文字列を入力してください。: isoginchaku
4 個目の文字列を入力してください。: hitode
5 個目の文字列を入力してください。: fujitsubo
namako umiushi isoginchaku hitode fujitsubo
```

配列の繰り返しを実行する foreach 文のスカラー変数名は、配列の要素の名前として使われますので、そこにデータを代入すると、そのデータは配列の要素に代入されます。

```
  DB<1> @a = (5, 3, 8, 7, 6)

  DB<2> foreach $a (@a) { $a *= $a }

  DB<3> p "@a"
25 9 64 49 36
```

第 6 章　ハッシュ

6.1　ハッシュの基礎

6.1.1　ハッシュとは何か

Perl には、任意の個数のデータから構成されるデータが、リスト値のほかにもう一種類あります。それは、「ハッシュ」(hash) と呼ばれるデータです。

ハッシュを構成しているそれぞれのデータは、そのハッシュの「要素」(element) と呼ばれます。

ハッシュの 1 個の要素は、2 個のデータから構成されるペアになっています。ペアの一方を「キー」(key)、もう一方を「値」(value) と呼びます。

リスト値の要素とは違って、ハッシュの要素は、順序というものを持っていません。したがって、それらを番号で識別することはできません。ハッシュでは、それぞれの要素は、番号ではなくキーによって識別されます。ですから、ひとつのハッシュが、同一のキーを持つ要素を 2 個以上持つ、ということはできません。

6.1.2 ハッシュ変数

ハッシュを格納することのできる変数は、「ハッシュ変数」(hash variable) と呼ばれます。

第 2.4.3 項で説明したように、変数名の先頭には、変数の種類を識別するために、「ファニー文字」と呼ばれる文字を書く必要があります。ハッシュ変数という変数の種類を識別するファニー文字は、パーセント (`%`) です。

6.1.3 ハッシュ変数の初期化

ハッシュ変数は、リスト値を代入することによって初期化することができます。

ハッシュ変数を初期化するためのリストは、

```
(キー，値，キー，値，･･･)
```

というように、ハッシュを構成するそれぞれの要素のキーと値を求める式を並べて書きます。たとえば、

```
%a = ("namako", 5, "hitode", 3, "umiushi", 7)
```

と書くことによって、キーが `namako` で値が 5 の要素、キーが `hitode` で値が 3 の要素、キーが `umiushi` で値が 7 の要素、という 3 個の要素から構成されるハッシュで、`%a` というハッシュ変数を初期化することができます。

リストを書くとき、要素の区切りとして、コンマの代わりに太い矢印 (`=>`) を使うこともできます。

```
  DB<1> @a = (5 => 3 => 8 => 7 => 6)

  DB<2> p "@a"
5 3 8 7 6
```

コンマと太い矢印は混在させることも可能ですので、コンマと太い矢印を使い分けることによって、ハッシュ変数を初期化するためのリストを読みやすくすることができます。つまり、

```
(キー => 値，キー => 値，キー => 値，･･･)
```

という形のリストを書くわけです。ハッシュ変数を初期化する先ほどの式を、この書き方で書き直すと、

```
%a = ("namako" => 5, "hitode" => 3, "umiushi" => 7)
```

というようになります。先ほどの書き方よりも、どれがキーでどれが値なのかということが分かりやすくなっているはずです。

ハッシュ変数の変数名をリストコンテキストで評価すると、そこに格納されているハッシュを構成しているそれぞれの要素を、キー、値、キー、値、……という順序で並べることによってできるリスト値が値として得られます。

```
  DB<1> %a = ("namako" => 5, "hitode" => 3, "umiushi" => 7)

  DB<2> @a = %a

  DB<3> p "@a"
namako 5 hitode 3 umiushi 7
```

6.1.4 ハッシュ変数の要素を指定する記述

ハッシュ変数は、ハッシュを構成しているそれぞれの要素を格納するための箱が集まったものだと考えることができます。ハッシュ変数を構成しているそれぞれの箱は、そのハッシュ変数の「要素」(element) と呼ばれます。

ハッシュ変数の特定の要素は、キーを指定することによって指定することができます。

キーを指定することによってハッシュ変数の特定の要素を指定したいときは、

```
$変数名{式}
```

という構文を持つ記述を書きます。この中の「変数名」のところには、ハッシュ変数の名前からファニー文字のパーセントを取り除いた部分を書きます。そして、中括弧 ({}) の中に書くのは、指定したい要素のキーを求める式です。たとえば、

```
$a{"namako"}
```

という記述を書くことによって、%a というハッシュ変数が持っている、namako というキーを持つペアが格納されている要素を指定することができます。

ハッシュ変数の要素を指定する記述を式として評価すると、その値として、指定された要素に格納されているペアの値が得られます。

```
  DB<1> %a = ("namako" => 5, "hitode" => 3, "umiushi" => 7)

  DB<2> p $a{"namako"}
5
```

6.1.5　ハッシュ変数の要素への代入

代入演算子の左側にハッシュ変数の要素を指定する記述を書いて、右側に式を書くと、その式の値が、キーで指定されたハッシュ変数の要素に対して、ペアの値として代入されます。

```
  DB<1> %a = ("namako" => 5, "hitode" => 3, "umiushi" => 7)

  DB<2> $a{"namako"} = 88

  DB<3> p $a{"namako"}
88
```

ハッシュ変数の要素への代入は、すでに存在する要素にしかできないというわけではありません。まだ存在していないキーを指定してデータを代入すると、そのキーとデータから構成されるペアが格納される要素が新しく作られます。

```
  DB<1> %a = ()

  DB<2> $a{"kurage"} = 777

  DB<3> p $a{"kurage"}
777
```

6.2　ハッシュを処理する組み込み関数

6.2.1　ハッシュのすべてのキーを求める

この節では、ハッシュを処理する組み込み関数を紹介したいと思います。

まず最初は、keys という組み込み関数です。

keys は、引数としてハッシュを受け取って、それを構成しているすべての要素のキーから構成されるリスト値を戻り値として返します。

```
  DB<1> %a = ("namako" => 5, "hitode" => 3, "umiushi" => 7)

  DB<2> @a = keys(%a)

  DB<3> p "@a"
namako hitode umiushi
```

ハッシュの要素は順序というものを持っていませんので、keys が返すリスト値の中でキーが並ぶ順序は、期待したとおりになるとは限りません。

ハッシュを構成している要素の個数は、keys を呼び出す関数呼び出しをスカラーコンテキストで評価することによって求めることができます。

```
  DB<1> %a = (0 => 7, 1 => 4, 2 => 0, 3 => 9, 4 => 8, 5 => 3)

  DB<2> $a = keys(%a)

  DB<3> p $a
6
```

foreach 文と keys を使うことによって、ハッシュを構成しているすべての要素について何かを実行する、ということができます。たとえば、

```
foreach $key (keys(%hash)) {
    print $key . " => " . $hash{$key} . "\n";
}
```

という foreach 文を書くことによって、%hash というハッシュ変数に格納されているハッシュのすべての要素を出力することができます。

次のプログラムは、何組かのキーと値を読み込んで（キーとして end が入力されると、読み込みを終了します）、そののち、読み込んだすべてのキーと値を出力します。

プログラムの例 keys.pl

```
%hash = ();
$key = "";
while ($key ne "end") {
    print "キーを入力してください（end で終了）: ";
    $key = <STDIN>;
    chomp($key);
    if ($key ne "end") {
        print "値を入力してください: ";
        $value = <STDIN>;
        chomp($value);
        $hash{$key} = $value;
    }
}
foreach $key (keys(%hash)) {
    print $key . " => " . $hash{$key} . "\n";
}
```

実行例

```
キーを入力してください（end で終了）: Japan
値を入力してください: Tokyo
キーを入力してください（end で終了）: Italy
値を入力してください: Rome
キーを入力してください（end で終了）: Ethiopia
値を入力してください: Addis Ababa
キーを入力してください（end で終了）: end
Italy => Rome
Japan => Tokyo
Ethiopia => Addis Ababa
```

6.2.2 ハッシュのすべての値を求める

values という組み込み関数は、引数としてハッシュを受け取って、それを構成しているすべての要素の値から構成されるリスト値を戻り値として返します。

```
  DB<1> %a = ("namako" => 5, "hitode" => 3, "umiushi" => 7)

  DB<2> @a = values(%a)

  DB<3> p "@a"
5 3 7
```

keys の場合と同じように、ハッシュの要素は順序というものを持っていませんので、values が返すリスト値の中で値が並ぶ順序も、期待したとおりになるとは限りません。

6.2.3 ハッシュに要素が存在するかどうかを調べる

exists という組み込み関数は、ハッシュの要素を指定する記述を引数として受け取って、指定された要素が存在するかどうかを示す真偽値（存在するならば 1、存在しないならば空文字列）を戻り値として返します。

```
  DB<1> %a = ("namako" => 5, "hitode" => 3, "umiushi" => 7)

  DB<2> p exists($a{"kurage"})
```

```
  DB<3> p exists($a{"hitode"})
1
```

次のプログラムは、1 個の文字列を読み込んで、その文字列を構成している文字の度数分布を出力します。

プログラムの例　frequency.pl

```
print "文字列を入力してください: ";
$s = <STDIN>;
chomp($s);
%f = ();
for ($i = 0; $i < length($s); $i++) {
    $c = substr($s, $i, 1);
    if (exists($f{$c})) {
        $f{$c} += 1;
    } else {
        $f{$c} = 1;
    }
}
foreach $c (keys(%f)) {
    print $c . " => " . $f{$c} . "\n";
}
```

実行例

```
文字列を入力してください: namamuginamagomenamatamago
e => 1
n => 3
a => 8
m => 6
u => 1
g => 3
t => 1
i => 1
o => 2
```

6.2.4　ハッシュの要素の削除

delete という組み込み関数は、ハッシュの要素を指定する記述を引数として受け取って、指定された要素を削除して、その要素の値を戻り値として返します。指定された要素が存在しない場合は、何もしないで、未定義値を返します。

```
  DB<1> %a = ("namako" => 5, "hitode" => 3, "umiushi" => 7)

  DB<2> p delete($a{"namako"})
5
  DB<3> @a = %a

  DB<4> p "@a"
hitode 3 umiushi 7
```

6.2.5　ハッシュから要素を一組ずつ取り出す

each という組み込み関数は、引数としてハッシュを受け取って、そこから一組の要素を取り出して、そのキーと値から構成されるリスト値を戻り値として返します。この関数は、過去にどの要素を取り出したかということを覚えていて、呼び出すたびに、まだ取り出されていなかった要素を取り出します。

第 6.2.1 項で、foreach 文と keys を使うことによってハッシュを構成しているすべての要素について何かを実行する方法を説明しましたが、それと同じことは、while 文と each を使うことによっても実行することができます。たとえば、

```
while (($key, $value) = each(%hash)) {
    print $key . " => " . $value . "\n";
}
```

というforeach文を書くことによって、%hashというハッシュ変数に格納されているハッシュのすべての要素を出力することができます。

第6.2.1項で紹介した、何組かのキーと値を読み込んで、そののち、読み込んだすべてのキーと値を出力するプログラムは、eachを使うことによって、次のように書くこともできます。

プログラムの例　each.pl

```
%hash = ();
$key = "";
while ($key ne "end") {
    print "キーを入力してください（end で終了）: ";
    $key = <STDIN>;
    chomp($key);
    if ($key ne "end") {
        print "値を入力してください: ";
        $value = <STDIN>;
        chomp($value);
        $hash{$key} = $value;
    }
}
while (($key, $value) = each(%hash)) {
    print $key . " => " . $value . "\n";
}
```

第7章　サブルーチン

7.1　サブルーチンの基礎

7.1.1　サブルーチンについての復習

第2.3.3項で説明したように、Perlでは、「サブルーチン定義」(subroutine definition)と呼ばれる文をプログラムの中に書くことによって、関数を自由に定義することができます。プログラムの中にサブルーチン定義を書くことによって定義された関数は、「サブルーチン」(subroutine)と呼ばれます。

この章では、サブルーチンについて説明したいと思います。

7.1.2　サブルーチン定義の場所

文は、基本的には、並んでいる順序のとおりに実行されます。ですから、文が並んでいる順序は、プログラムの動作を左右することになります。

サブルーチン定義も文の一種ですので、文を書くことができる場所ならばどこにでも書くことができます。しかし、それが実行される順番は、それが書かれた場所とは無関係です。なぜなら、Perlの処理系は、普通の文を実行するよりも前にサブルーチン定義を実行するからです。したがって、プログラムの下のほうに書かれたサブルーチン定義によって定義されたサブルーチンを、それよりも上に書かれた文から呼び出す、ということも可能です。

7.1.3　サブルーチン定義の書き方

サブルーチンを定義するサブルーチン定義は、

sub 識別子 ブロック

と書きます。「識別子」のところには、名前としてサブルーチンに与えたい識別子を書きます。そして、「ブロック」のところには、サブルーチンにしたい動作をあらわすブロックを書きます。

たとえば、次のサブルーチン定義を書くことによって、「Hello, world!」という文字列を出力するサブルーチンを作って、worldという名前をそれに与えることができます。

```
sub world { print "Hello, world!"; }
```

7.1.4　サブルーチン呼び出し

サブルーチンを呼び出す式は、「サブルーチン呼び出し」(subroutine call)と呼ばれます。

引数を受け取らないサブルーチンを呼び出すサブルーチン呼び出しは、

&サブルーチン名

と書きます[1]。「サブルーチン名」のところには、名前としてサブルーチンに与えた識別子を書きます。たとえば、

```
&world
```

というサブルーチン呼び出しを書くことによって、world という名前のサブルーチンを呼び出すことができます。

それでは、REPL を使って試してみましょう。

```
DB<1> sub world { print "Hello, world!"; }

DB<2> &world
Hello, world!
```

次のプログラムは、「こんにちは、世界。」と出力する world というサブルーチンを定義して、それを呼び出します。

プログラムの例　world.pl

```
&world;

sub world {
    print "こんにちは、世界。\n";
}
```

実行例

```
こんにちは、世界。
```

7.1.5　サブルーチンを定義するという機能は何のためにあるのか

Perl は、サブルーチンを定義するという機能を持っています。そのような、名前によって呼び出すことのできる動作を定義するという機能は、Perl に限らず、ほとんどすべてのプログラミング言語が備えているものです。プログラミング言語は、いったい何のために、このような機能を備えているのでしょうか。

人間にとって、複雑なものを理解するというのは、容易なことではありません。ですから、複雑なものを作るときには、それを人間にとって理解しやすいものにする工夫をすることが、とても大切です。

複雑なものを人間にとって理解しやすいものにする上で重要なことは、少数の部品の組み合わせによって全体を構築するということです。個々の部品は、もしもそれ自体が複雑なものである場合は、それもまた、少数の部品の組み合わせによって構築される必要があります。

つまり、単純な部品を組み合わせることによって少し複雑な部品を作って、少し複雑な部品を組み合わせることによってさらに複雑な部品を作って、……というように、部品を階層的に組み合わせることによって構築されたものは、それがどれだけ複雑なものであっても人間にとって理解しやすい、ということです。

プログラムを書く場合も、もしもそれが複雑なものである場合は、それを人間にとって理解しやすいものにする工夫が必要になります。部品を階層的に組み合わせて構築することによって、人間にとって理解しやすいものを作ることができるという原理は、プログラムの場合にも有効です。プログラムを構築するための部品というのは、名前によって呼び出すことのできる動作です。

名前によって呼び出すことのできる動作を定義するという機能がプログラミング言語に備わっているのはいったい何のためなのか、という問題の解答は、人間にとって理解しやすいプログラムを書くことができるようにするため、ということになります。

7.2　引数と戻り値

7.2.1　引数が格納される配列

引数を渡してサブルーチンを呼び出すと、それらの引数は、@_ という名前の配列に格納されます。

[1] 多くの場合、アンパサンド (&) は省略することができるのですが、このチュートリアルでは常にアンパサンドを書くことにします。

たとえば、次のサブルーチン定義は、受け取ったすべての引数を、コンマで区切って出力します。

```
sub print_args { print join(", ", @_); }
```

それでは、REPL を使って試してみましょう。

```
  DB<1> sub print_args { print join(", ", @_); }

  DB<2> &print_args(5, 3, 8, 7, 6)
5, 3, 8, 7, 6
```

次のプログラムの中で定義されている stars というサブルーチンは、引数として受け取った個数のアスタリスクを出力します。

プログラムの例　stars.pl

```
print "星の個数を入力してください。: ";
$n = <STDIN>;
&stars($n);

sub stars {
    for ($i = 1; $i <= @_[0]; $i++) {
        print "*";
    }
    print "\n";
}
```

実行例

```
星の個数を入力してください。: 40
****************************************
```

7.2.2 引数の順序

先ほど定義した print_args の動作からも分かるとおり、@_ に格納されるリストは、サブルーチン呼び出しの中で並んでいるそれぞれの式の値を、それと同じ順序で並べたものです。たとえば、

```
&namako(24, 33, 81)
```

というサブルーチン呼び出しで、namako というサブルーチンを呼び出したとすると、@_ には、

```
(24, 33, 81)
```

というリストが格納されることになります。

次のプログラムの中で定義されている rect というサブルーチンは、文字列と縦の個数と横の個数を引数として受け取って、その文字列を長方形の形に並べたものを出力します。

プログラムの例　rect.pl

```
print "文字列を入力してください。: ";
$s = <STDIN>;
chomp($s);
print "横の個数を入力してください。: ";
$w = <STDIN>;
print "縦の個数を入力してください。: ";
$h = <STDIN>;
&rect($s, $w, $h);

sub rect {
    for ($i = 1; $i <= @_[2]; $i++) {
        for ($j = 1; $j <= @_[1]; $j++) {
            print @_[0];
        }
        print "\n";
    }
}
```

実行例

```
文字列を入力してください。: Perl
横の個数を入力してください。: 12
縦の個数を入力してください。: 3
PerlPerlPerlPerlPerlPerlPerlPerlPerlPerlPerlPerl
PerlPerlPerlPerlPerlPerlPerlPerlPerlPerlPerlPerl
PerlPerlPerlPerlPerlPerlPerlPerlPerlPerlPerlPerl
```

7.2.3　return 文

戻り値を返すサブルーチンを定義したいときは、サブルーチン定義のブロックの中に、「return 文」(return statement) と呼ばれる文を書きます。

return 文は、

```
return 式;
```

と書きます。この文は、その中の式を評価して、その値を戻り値として返して、サブルーチンを終了させる、という動作をあらわしています。

次のプログラムの中で定義されている square というサブルーチンは、1 個の数値を引数として受け取って、その 2 乗を戻り値として返します。

プログラムの例　square.pl

```
print "数値を入力してください。: ";
$a = <STDIN>;
chomp($a);
print $a . "の 2 乗は" . &square($a) . "です。\n";

sub square {
    return @_[0] * @_[0];
}
```

実行例

```
数値を入力してください。: 9
9 の 2 乗は 81 です。
```

n が自然数だとするとき、n 自身を除いた n のすべての約数の和が n と等しいならば、n は、「完全数」(perfect number) と呼ばれます。たとえば、6 は、$1+2+3=6$ ですから、完全数です。

次のプログラムの中で定義されている perfect というサブルーチンは、1 個の自然数を引数として受け取って、それが完全数ならば真、そうでなければ偽を戻り値として返します。

プログラムの例　perfect.pl

```
print "自然数を入力してください。: ";
$n = <STDIN>;
chomp($n);
print $n . "は完全数";
if (&perfect($n)) {
    print "です。\n";
} else {
    print "ではありません。\n";
}

sub perfect {
    $m = 0;
    for ($i = 1; $i < @_[0]; $i++) {
        if (@_[0] % $i == 0) {
            $m += $i;
        }
    }
    return @_[0] == $m;
}
```

実行例

```
自然数を入力してください。: 28
28 は完全数です。
```

```
自然数を入力してください。: 30
30 は完全数ではありません。
```

7.3 スコープ

7.3.1 スコープの基礎

名前と、その名前が与えられている対象とのあいだの関係が保たれる、プログラムの上での範囲は、その名前の「スコープ」(scope) と呼ばれます。

プログラミング言語の多くは、名前のスコープに関する規則を定めています。Perl も例外ではありません。

7.3.2 レキシカルなスコープ

プログラムの全域よりも狭い、限定されたスコープは、「レキシカルなスコープ」(lexical scope) と呼ばれます。

Perl では、ブロックの中だけというレキシカルなスコープを持つ名前が与えられた変数は、「レキシカル変数」(lexical variable) と呼ばれます。

レキシカル変数は、my という関数を使うことによって作ることができます。

my を呼び出す式は、

```
my 変数名
```

と書きます。そうすると、引数として my に渡した変数名を名前とするレキシカル変数が作られます。たとえば、

```
my $a;
```

という文を書くことによって、$a という名前を持つレキシカル変数を作ることができます。

my に渡す引数としては、スカラー変数の名前から構成されるリストを書くこともできます。たとえば、

```
my ($a, $b, $c);
```

という文を書くことによって、$a、$b、$c という 3 個のレキシカル変数を作ることができます。

my を呼び出す式の右側に代入演算子と式を書くことによって、レキシカル変数を作るだけではなくて、さらにその変数に初期値を代入する、ということもできます。たとえば、

```
my $a = 33;
```

という文を書くことによって、$a という名前を持つレキシカル変数を作って、そこに初期値として 33 を代入することができます。同じように、

```
my ($a, $b, $c) = @array;
```

という文を書くことによって、$a、$b、$c という 3 個のレキシカル変数を作って、それらの変数に、@array に格納されているリスト値のそれぞれの要素を初期値として代入することができます。

プログラムの例 local.pl

```
&subroutine1;

sub subroutine1 {
    my $a = "私は subroutine1 のレキシカル変数の\$a です。";
    print $a . "\n";
    &subroutine2;
    print $a . "\n";
}

sub subroutine2 {
    my $a = "私は subroutine2 のレキシカル変数の\$a です。";
    print $a . "\n";
}
```

実行例

```
私は subroutine1 のレキシカル変数の$a です。
私は subroutine2 のレキシカル変数の$a です。
私は subroutine1 のレキシカル変数の$a です。
```

7.3.3　引数をレキシカル変数に代入する

第 7.2.1 項で説明したように、Perl では、サブルーチンが受け取った引数は、@_ という名前の配列に格納されます。

受け取った引数を使うときには、配列に格納された状態のままで、$_[0] とか $_[1] というように書いてもいいのですが、それだと、プログラムが読みにくくなってしまいます。サブルーチンが受け取った引数は、それがどのような引数なのかということが分かるような名前のレキシカル変数に代入するほうがいいでしょう。

次のプログラムは、第 7.2.2 項で紹介した、文字列を長方形の形に並べたものを出力するプログラムを、rect というサブルーチンが受け取った引数をレキシカル変数に代入するように改良したものです。

プログラムの例　rect2.pl

```
print "文字列を入力してください。: ";
$s = <STDIN>;
chomp($s);
print "横の個数を入力してください。: ";
$w = <STDIN>;
print "縦の個数を入力してください。: ";
$h = <STDIN>;
&rect($s, $w, $h);

sub rect {
    my ($string, $width, $height) = @_;
    for (my $i = 1; $i <= $height; $i++) {
        for (my $j = 1; $j <= $width; $j++) {
            print $string;
        }
        print "\n";
    }
}
```

7.3.4　for 文のレキシカル変数

for 文で、繰り返しを制御するために使う変数に初期値を代入するときに、my を使ってその変数をレキシカル変数にしておくと、その変数は、for 文の中だけというレキシカルなスコープを持つことになります。

プログラムの例　myfor.pl

```
&subroutine;

sub subroutine {
    my $i = "私は for 文の外の\$i です。";
    print $i . "\n";
    for (my $i = 0; $i <= 10; $i++) {
        print $i . " ";
    }
    print "\n";
    print $i . "\n";
}
```

実行例

```
私は for 文の外の$i です。
0 1 2 3 4 5 6 7 8 9 10
私は for 文の外の$i です。
```

このプログラムの実行結果が示しているように、外側のスコープで使われている名前と同じ名前を内側のスコープで使うと、内側のスコープでは、外側のスコープで使われている名前が隠さ

れることになります。

7.3.5 foreach 文のレキシカル変数

for 文と同じように、foreach 文でも、リスト値または配列の要素に与えられるスカラー変数名を my に渡すと、その変数名は、foreach 文の中だけというレキシカルなスコープを持つことになります。

プログラムの例 myforeach.pl

```
&subroutine;

sub subroutine {
    my $a = "私は foreach 文の外の\$a です。";
    print $a . "\n";
    foreach my $a (0..10) {
        print $a . " ";
    }
    print "\n";
    print $a . "\n";
}
```

実行例

```
私は foreach 文の外の$a です。
0 1 2 3 4 5 6 7 8 9 10
私は foreach 文の外の$a です。
```

7.3.6 グローバルなスコープ

プログラムの全域というスコープは、「グローバルなスコープ」(global scope) と呼ばれます。

Perl では、my を使わないで変数に与えられた名前は、グローバルなスコープを持つことになります。そのような、それに与えられた名前がグローバルなスコープを持つ変数は、「グローバル変数」(global variable) と呼ばれます。

ブロックの内部も、グローバルなスコープの一部分になります。ですから、ブロックの中でも、グローバル変数に与えられた名前を書くことによって、その変数の内容を求めたり、その変数にデータを設定したりすることができます。

プログラムの例 global.pl

```
$global = "私はグローバル変数です。";
print "$global\n";
&subroutine;
print "$global\n";

sub subroutine {
    print "$global\n";
    $global = "違うデータを設定してみました。";
    print "$global\n";
}
```

実行例

```
私はグローバル変数です。
私はグローバル変数です。
違うデータを設定してみました。
違うデータを設定してみました。
```

7.3.7 レキシカルなスコープという規則のメリット

ところで、「ブロックの中で my 宣言によって変数に与えられた名前は、そのブロックの中だけというスコープを持つ」という規則には、いったいどのようなメリットがあるのでしょうか。

もしも、「いかなる変数名もグローバルなスコープを持つ」という規則が定められていたとするとどうなるか、ということについて考えてみましょう。その場合、変数に名前を与えるときには、その名前がすでに別の場所で使われていないか、ということに細心の注意を払う必要があります。うっかりと同一の名前を複数の場所で使ってしまうと、思わぬ不具合が発生しかねません。

つまり、「ブロックの中で my によって変数に与えられた名前は、そのブロックの中だけというスコープを持つ」という規則は、「ブロックを書くときには、そのブロックの外でどのような名前が使われているかということを、まったく気にする必要がない」というメリットを、プログラムを書く人に与えてくれているのです。

7.4　再帰

7.4.1　再帰とは何か

この節では、「再帰」(recursion) と呼ばれるものについて説明したいと思います。

再帰というのは、全体と同じものが一部分として含まれているという性質のことです。再帰という性質を持っているものは、「再帰的な」(recursive) と形容されます。

ここに、1 台のカメラと 1 台のモニターがあるとします。まず、それらを接続して、カメラで撮影した映像がモニターに映し出されるようにします。そして次に、カメラをモニターの画面に向けます。すると、モニターの画面には、そのモニター自身が映し出されることになります。そして、映し出されたモニターの画面の中には、さらにモニター自身が映し出されています。このときにモニターの画面に映し出されるのは、再帰という性質を持っている映像、つまり再帰的な映像です。

また、先祖と子孫の関係も再帰的です。なぜなら、先祖と子孫との中間にいる人々も、やはり先祖と子孫の関係で結ばれているからです。

7.4.2　基底

再帰という性質を持っているものは、全体と同じものが一部分として含まれているわけですが、その構造は、内部に向かってどこまでも続いている場合もあれば、どこかで終わっている場合もあります。

再帰的な構造がどこかで終わっている場合、その中心には、その内部に再帰的な構造を持っていない何かがあります。そのような、再帰的な構造の中心にあって、その内部に再帰的な構造を持っていないものは、その再帰的な構造の「基底」(basis) と呼ばれます。

先祖と子孫の関係では、親子関係というのが、その再帰的な構造の基底になります。

7.4.3　サブルーチンの再帰的な定義

サブルーチンは、再帰的に定義することが可能です。サブルーチンを再帰的に定義するというのは、定義される当のサブルーチンを使ってサブルーチンを定義するということです。再帰的な構造を持っている概念を取り扱うサブルーチンは、再帰的に定義するほうが、再帰的ではない方法で定義するよりもすっきりした記述になります。

サブルーチンを再帰的に定義する場合は、それが循環に陥ることを防ぐために、基底について記述した選択肢を作っておくことが必要になります。

7.4.4　階乗

n が 0 またはプラスの整数だとするとき、n から 1 までの整数をすべて乗算した結果、つまり、

$$n \times (n-1) \times (n-2) \times \cdots \times 1$$

という計算の結果は、n の「階乗」(factorial) と呼ばれて、$n!$ と書きあらわされます。ただし、$0!$ は 1 だと定義します。

たとえば、$5!$ は、

$$5 \times 4 \times 3 \times 2 \times 1$$

という計算をすればいいわけですから、120 ということになります。

階乗という概念は、再帰的な構造を持っています。なぜなら、階乗は、

$$\begin{cases} & 0! = 1 \\ n \geq 1 \text{ ならば} & n! = n \times (n-1)! \end{cases}$$

というように再帰的に定義することができるからです。

階乗を求めるサブルーチンも、再帰的に定義することができます。次のプログラムは、階乗を求める `factorial` というサブルーチンを再帰的に定義しています。

プログラムの例 factorial.pl

```
print "整数を入力してください。: ";
$n = <STDIN>;
chomp($n);
print $n . "の階乗は" . &factorial($n) . "です。\n";

sub factorial {
    my $n = $_[0];
    if ($n == 0) {
        return 1;
    } elsif ($n >= 1) {
        return $n * &factorial($n-1);
    }
}
```

実行例

```
整数を入力してください。: 5
5 の階乗は 120 です。
```

7.4.5 フィボナッチ数列

第 0 項と第 1 項が 1 で、第 2 項以降はその直前の 2 項を足し算した結果である、という数列は、「フィボナッチ数列」(Fibonacci sequence) と呼ばれます。フィボナッチ数列の第 0 項から第 12 項までを表にすると、次のようになります。

n	0	1	2	3	4	5	6	7	8	9	10	11	12
第 n 項	1	1	2	3	5	8	13	21	34	55	89	144	233

フィボナッチ数列というのは再帰的な構造を持っている概念ですので、その第 n 項 (F_n) は、

$$\begin{cases} & F_0 = 1 \\ & F_1 = 1 \\ n \geq 2 \text{ ならば} & F_n = F_{n-2} + F_{n-1} \end{cases}$$

というように再帰的に定義することができます。

フィボナッチ数列の第 n 項を求めるサブルーチンも、再帰的に定義することができます。次のプログラムは、フィボナッチ数列の第 n 項を求める fibonacci というサブルーチンを再帰的に定義しています。

プログラムの例 fibonacci.pl

```
print "整数を入力してください。: ";
$n = <STDIN>;
chomp($n);
print "フィボナッチ数列の第" . $n . "項は" .
    &fibonacci($n) . "です。\n";

sub fibonacci {
    my $n = $_[0];
    if ($n == 0) {
        return 1;
    } elsif ($n == 1) {
        return 1;
    } elsif ($n >= 2) {
        return &fibonacci($n-2) + &fibonacci($n-1);
    }
}
```

実行例

```
整数を入力してください。: 7
フィボナッチ数列の第 7 項は 21 です。
```

7.4.6　最大公約数

n がプラスの整数で、m が 0 またはプラスの整数だとするとき、n と m の両方に共通する約数のうちで最大のもののことを、n と m の「最大公約数」(greatest common measure, GCM) と呼びます（m が 0 の場合は、n と m の最大公約数は n だと定義します)。たとえば、54 と 36 の最大公約数は 18 です。

n と m の最大公約数は、「ユークリッドの互除法」(Euclidean algorithm) と呼ばれる次のような再帰的な手順を実行することによって求めることができます。

- m が 0 ならば、n が、n と m の最大公約数である。
- m が 1 以上ならば、n を m で除算したときのあまりを求めて、その結果を r とする。そして、m と r の最大公約数を求めれば、その結果が n と m の最大公約数である。

次のプログラムは、2 個のプラスの整数の最大公約数を求める gcm というサブルーチンを、ユークリッドの互除法を使って定義しています。

プログラムの例　gcm.pl

```
print "整数を入力してください。: ";
$n = <STDIN>;
chomp($n);
print "整数を入力してください。: ";
$m = <STDIN>;
chomp($m);
print $n . "と" . $m . "の最大公約数は" .
    &gcm($n, $m) . "です。\n";

sub gcm {
    my ($n, $m) = @_;
    if ($m == 0) {
        return $n;
    } elsif ($m >= 1) {
        return gcm($m, $n % $m);
    }
}
```

実行例

```
整数を入力してください。: 54
整数を入力してください。: 36
54 と 36 の最大公約数は 18 です。
```

第 8 章　ファイル

8.1　オープンとクローズ

8.1.1　オープンとクローズの基礎

この章では、ファイルに対する操作を実行するプログラムについて説明したいと思います。

ファイルからデータを読み込んだり、ファイルにデータを書き込んだりするためには、それに先立って、そのための準備をする必要があります。ファイルに対する読み書きのための準備をすることを、ファイルを「オープンする」(open) と言います。

ファイルに対して読み書きを実行したときは、それが終わったのち、その後始末をする必要があります。ファイルに対する読み書きが終わったのちにその後始末をすることを、ファイルを「クローズする」(close) と言います。

8.1.2　ファイルをオープンする組み込み関数

ファイルは、open という組み込み関数を呼び出すことによってオープンすることができます。

open を呼び出す式は、

open 変数名 , 式

と書きます。この中の「変数名」というところには、「ファイルハンドル」(file handle) と呼ばれ

るものに与えるスカラー変数の名前を書いて、「式」というところには、オープンしたいファイルのパス名を求める式を書きます。

open は、ファイルをオープンして、「ファイルハンドル」と呼ばれるものを作ります。ファイルハンドルというのは、プログラムの中からファイルを操作するために使われる変数の一種です。

たとえば、

```
open $handle, "namako.txt"
```

という式を評価したとしましょう。この場合、open は、カレントディレクトリにある namako.txt というファイルをオープンします。そして、そのファイルを操作するために使われるファイルハンドルを作って、それに対して $handle という変数名を与えます。

8.1.3 ファイルをクローズする組み込み関数

ファイルは、close という組み込み関数を呼び出すことによってクローズすることができます。

close を呼び出す式は、

```
close 変数名
```

と書きます。この中の「変数名」というところには、クローズしたいファイルのファイルハンドルに与えられている変数名を書きます。たとえば、

```
close $handle
```

という式を評価することによって、$handle という変数名が与えられているファイルファンドルによって操作されるファイルをクローズすることができます。

8.1.4 open の戻り値

open は、戻り値として、ファイルのオープンに成功した場合は 0 以外の数値、失敗した場合は未定義値を返します（未定義値は、真偽値コンテキストでは偽に変換されます）。

さらに、open は、ファイルのオープンに失敗した場合、その理由を示すメッセージを、$! という名前の変数に代入します。ですから、

```
if (open $handle, $pathname) {
    通常の処理
} else {
    print "$pathname: $!\n";
}
```

という if 文を書いておけば、ファイルのオープンに失敗した場合には、失敗した理由を示すメッセージが出力されることになります。

次のプログラムは、コマンドライン引数でパス名を渡すと、そのパス名で指定されたファイルをオープンします。そして、オープンに成功した場合は、「○○は正常にオープンされました。」というメッセージを出力します。オープンに失敗した場合は、$! に代入されたメッセージを出力します。

プログラムの例 open.pl

```
$pathname = $ARGV[0];
if (open $handle, $pathname) {
    print "$pathname は正常にオープンされました。\n";
    close $handle;
} else {
    print "$pathname: $!\n";
}
```

実行例

```
$ perl open.pl open.pl
open.pl は正常にオープンされました。
$ perl open.pl namako.txt
namako.txt: No such file or directory
```

8.2　ファイルからの読み込み

8.2.1　ファイルからデータを読み込む式

ファイルからデータを読み込みたいときは、

<変数名>

という形の式を書きます。この中の「変数名」のところには、open が作ったファイルハンドルに与えられている変数名を書きます。

ファイルからデータを読み込む式を、リストコンテキストで評価すると、その値として、ファイルの内容を構成しているそれぞれの行を要素とするリスト値が得られます（改行は行の末尾に残されます）。たとえば、

```
@a = <$handle>
```

という式を評価すると、$handle というファイルハンドルによって操作されるファイルの内容を構成しているそれぞれの行を要素とするリスト値が、@a という配列に代入されます。

次のプログラムは、コマンドライン引数でパス名を渡すと、そのパス名で指定されたファイルの内容を出力します。

プログラムの例　read.pl

```
$pathname = $ARGV[0];
if (open $handle, $pathname) {
    @a = <$handle>;
    print @a;
    close $handle;
} else {
    print "$pathname: $!\n";
}
```

テキストファイルの例　puellae.txt

```
Kaname Madoka
Akemi Homura
Miki Sayaka
Tomoe Mami
```

実行例

```
$ perl read.pl puellae.txt
Kaname Madoka
Akemi Homura
Miki Sayaka
Tomoe Mami
```

8.2.2　行単位での読み込み

ファイルからデータを読み込む式を、リストコンテキストではなくて、スカラーコンテキストで評価すると、ファイルから 1 行だけが読み込まれて、その行がその式の値になります（改行は行の末尾に残されます）。

ファイルからの読み込みは、「読み込み位置」(read position) と呼ばれる位置から開始されます。ファイルがオープンされると、その時点で、読み込み位置はファイルの先頭に置かれます。そして、読み込みが実行されると、読み込み位置は、読み込んだ部分の直後へ移動します。

ファイルの末尾という位置は、「ファイルの終わり」(end of file) と呼ばれます。読み込み位置がファイルの終わりにあるときに、ファイルからデータを読み込む式をスカラーコンテキストで評価すると、その値として、未定義値が得られます。

未定義値は真偽値コンテキストでは偽に変換されますので、

```
while ($line = <$handle>) {
    $line を処理する。
}
```

というような while 文を書くことによって、ファイルの終わりに到達するまで、ファイルから 1 行ずつ読み込んでそれを処理する、という動作を記述することができます。

次のプログラムは、コマンドライン引数でパス名を渡すと、そのパス名で指定されたファイルの内容を構成しているそれぞれの行の先頭に大なり (>) と空白を挿入した結果を出力します。

プログラムの例 quote.pl

```
$pathname = $ARGV[0];
if (open $handle, $pathname) {
    while ($line = <$handle>) {
        print "> $line";
    }
    close $handle;
} else {
    print "$pathname: $!\n";
}
```

実行例

```
$ perl quote.pl puellae.txt
> Kaname Madoka
> Akemi Homura
> Miki Sayaka
> Tomoe Mami
```

8.3 ファイルへの書き込み

8.3.1 オープンモード

ファイルにデータを書き込む場合も、ファイルからデータを読み込む場合と同じように、まず最初に open を使ってそのファイルをオープンしないといけません。そして、書き込みが終わったのち、 close を使ってそのファイルをクローズする必要があります。

ファイルにデータを書き込む場合、そのファイルをオープンするために open を呼び出す式は、

open 変数名 , オープンモード , 式

と書きます。この中の「変数名」というところには、ファイルハンドルに与える変数名を書いて、「式」というところには、オープンしたいファイルのパス名を求める式を書きます。そして、「オープンモード」というところには、「オープンモード」(open mode) と呼ばれる文字列を生成する文字列定数を書きます。

「オープンモード」というのは、どのような処理を実行するためにファイルをオープンするのかということを指定するための文字列のことです。ファイルに対して書き込みを実行したいときは、 > または >> という文字列を、オープンモードとして指定する必要があります。

> と >> との相違点は、ファイルがすでに存在していた場合に、すでに書き込まれているデータを削除するかしないかということです。 > は、すでに書き込まれているデータを削除した上で、データを書き込みます。 >> は、すでに書き込まれているデータを削除しないで、その末尾にデータを追加します。

たとえば、

```
open $handle, ">", "namako.txt"
```

という式を評価したとしましょう。この場合、 open は、カレントディレクトリにある namako.txt というファイルを、それに対して書き込みを実行するためにオープンします。そして、そのファイルを操作するために使われるファイルハンドルを作って、それに対して $handle という変数名を与えます。

8.3.2 print によるファイルへのデータの書き込み

print という組み込み関数は、標準出力にデータを出力することができるだけではなくて、ファイルにデータを書き込むということもできます。

ファイルにデータを書き込みたいとき、 print を呼び出す式は、

print 変数名 式

と書きます。この中の「変数名」というところには、データを書き込むファイルを操作するため

のファイルハンドルに与えられている変数名を書きます。そして、「式」というところには、ファイルに書き込みたいデータを求める式を書きます。変数名と式とのあいだを区切っているのが、コンマではなくて空白だという点に注意してください。たとえば、

```
print $handle $a
```

という式を書くことによって、$handle という変数名が与えられているファイルハンドルによって操作されるファイルに対して、$a という変数の内容を書き込むことができます。

次のプログラムは、1 個目のコマンドライン引数としてパス名、2 個目のコマンドライン引数として文字列を渡すと、そのパス名で指定されたファイルに、その文字列と改行を書き込みます。

プログラムの例　writefile.pl

```
$pathname = $ARGV[0];
$s = $ARGV[1];
if (open $handle, ">", $pathname) {
    print $handle "$s\n";
    close $handle;
} else {
    print "$pathname: $!\n";
}
```

実行例

```
$ perl writefile.pl hoge.txt isoginchaku
$ cat hoge.txt
isoginchaku
```

8.4　ファイルとディレクトリに対する操作

8.4.1　この節について

この節では、ファイルやディレクトリ（フォルダー）を操作する組み込み関数のうちのいくつかを紹介したいと思います。

8.4.2　ディレクトリの作成

mkdir という組み込み関数は、引数としてパス名を受け取って、そのパス名を持つディレクトリを作成します。たとえば、

```
mkdir "namako"
```

という式で mkdir を呼び出すことによって、namako という名前のディレクトリをカレントディレクトリに作成することができます。

8.4.3　ディレクトリの削除

rmdir という組み込み関数は、引数としてパス名を受け取って、そのパス名で指定されたディレクトリを削除します。たとえば、

```
rmdir "namako"
```

という式で rmdir を呼び出すことによって、カレントディレクトリにある namako という名前のディレクトリを削除することができます。

8.4.4　ファイルの削除

unlink という組み込み関数は、引数としてパス名を受け取って、そのパス名で指定されたファイルを削除します。たとえば、

```
unlink "umiushi.txt"
```

という式で unlink を呼び出すことによって、カレントディレクトリにある umiushi.txt という名前のファイルを削除することができます。

8.4.5　ファイルの名前の変更

rename という組み込み関数は、

```
rename パス名1, パス名2
```

というように、引数として二つのパス名を受け取って、パス名 $_1$ で指定されたファイルの名前をパス名 $_2$ に変更します。たとえば、

```
rename "kurage.txt", "hitode.txt"
```

という式で rename を呼び出すことによって、カレントディレクトリにある kurage.txt というファイルの名前を hitode.txt に変更することができます。

8.4.6 ファイルの情報の取得

stat という組み込み関数は、引数としてパス名を受け取って、そのパス名で指定されたファイルの情報から構成されるリスト値を戻り値として返します。

stat が返すリスト値は、次のような要素から構成されます（ただし、ファイルシステムのタイプによってはサポートされない要素もあります）。

(1) ファイルシステムのデバイス番号
(2) inode 番号
(3) ファイルモード（タイプとパーミッション）
(4) そのファイルに対するハードリンクの個数
(5) ファイルの所有者のユーザー ID（数値）
(6) ファイルの所有者のグループ ID（数値）
(7) デバイス識別子（特殊ファイルのみ）
(8) ファイルの大きさ（バイト数）
(9) 最終アクセス時刻（エポック時からの秒数）
(10) 最終更新時刻（エポック時からの秒数）
(11) inode が変更された時刻（エポック時からの秒数）
(12) ファイルの読み書きに適したブロックサイズ（バイト数）
(13) 実際に使用されているブロック数

エポック時というのは、1970 年 1 月 1 日午前 0 時 0 分 0 秒のことです。

8.4.7 最終アクセス時刻と最終更新時刻の変更

utime という組み込み関数は、次の要素から構成されるリスト値を引数として受け取って、ファイルの最終アクセス時刻と最終更新時刻を変更します。

(1) 最終アクセス時刻（エポック時からの秒数）
(2) 最終更新時刻（エポック時からの秒数）
(3) 変更の対象となるファイルのパス名

次のプログラムは、コマンドライン引数で指定されたファイルの最終アクセス時刻と最終更新時刻を現在の時刻に変更します。

プログラムの例 touch.pl

```
$now = time;
utime ($now, $now, $ARGV[0]);
```

このプログラムの中で使われている time というのは、エポック時から現在までの秒数を戻り値として返す組み込み関数です。

8.5 ディレクトリからの読み込み

8.5.1 ディレクトリからの読み込みの基礎

特定のディレクトリ（フォルダー）の中にあるすべてのファイルやディレクトリを処理したいという場合には、そのディレクトリから、その中にあるファイルやディレクトリの一覧を読み込む必要があります。

ディレクトリから一覧を読み込む方法は、ファイルからデータを読み込む方法によく似ています。ただし、そのために使う組み込み関数は、同じではありません。ディレクトリから一覧を読み込む場合に使われるのは、次のような組み込み関数です。

opendir	ディレクトリをオープンする。
closedir	ディレクトリをクローズする。
readdir	ディレクトリから一覧を読み込む。

8.5.2 ディレクトリのオープン

ディレクトリは、opendir という組み込み関数を呼び出すことによってオープンすることができます。

opendir を呼び出す式は、

```
opendir 変数名 , 式
```

と書きます。この中の「識別子」というところには、「ディレクトリハンドル」(directory handle) と呼ばれるものに与えるスカラー変数の名前を書いて、「式」というところには、オープンしたいディレクトリのパス名を求める式を書きます。

open は、ディレクトリをオープンして、「ディレクトリハンドル」と呼ばれるものを作ります。ディレクトリハンドルというのは、プログラムの中からディレクトリを操作するために使われる変数の一種です。

たとえば、

```
opendir $handle, "hitode"
```

という式を評価したとしましょう。この場合、opendir は、カレントディレクトリにある hitode というディレクトリをオープンします。そして、そのディレクトリを操作するために使われるディレクトリハンドルを作って、それに対して $handle という変数名を与えます。

8.5.3 ディレクトリのクローズ

ディレクトリは、closedir という組み込み関数を呼び出すことによってクローズすることができます。

closedir を呼び出す式は、

```
closedir 変数名
```

と書きます。この中の「変数名」というところには、クローズしたいディレクトリのディレクトリハンドルに与えられている変数名を書きます。たとえば、

```
closedir $handle
```

という式を評価することによって、$handle という変数名が与えられているディレクトリファンドルによって操作されるディレクトリをクローズすることができます。

8.5.4 open の戻り値

opendir は、open と同じように、戻り値として、ディレクトリのオープンに成功した場合は 0 以外の数値、失敗した場合は未定義値を返します（未定義値は、真偽値コンテキストでは偽に変換されます）。

さらに、opendir は、open と同じように、ディレクトリのオープンに失敗した場合、その理由を示すメッセージを、$! という名前の変数に代入します。

次のプログラムは、コマンドライン引数でパス名を渡すと、そのパス名で指定されたディレクトリをオープンします。そして、オープンに成功した場合は、「○○は正常にオープンされました。」というメッセージを出力します。オープンに失敗した場合は、$! に代入されたメッセージを出力します。

プログラムの例 opendir.pl

```
$pathname = $ARGV[0];
if (opendir $handle, $pathname) {
    print "$pathname は正常にオープンされました。\n";
    closedir $handle;
```

```
} else {
    print "$pathname: $!\n";
}
```

実行例

```
$ mkdir hitode
$ perl opendir.pl hitode
hitode は正常にオープンされました。
$ perl opendir.pl kurage
kurage: No such file or directory
```

8.5.5 ディレクトリから一覧を読み込む組み込み関数

ファイルからデータを読み込みたいときは、readdir という組み込み関数を使います。

readdir を呼び出す式は、

readdir 変数名

と書きます。この中の「変数名」のところには、opendir が作ったディレクトリハンドルに与えられている変数名を書きます。

readdir を呼び出す式をリストコンテキストで評価すると、その値として、ディレクトリの中にあるものの名前を要素とするリスト値が得られます。たとえば、

```
@a = readdir $handle
```

という式を評価すると、$handle というディレクトリハンドルによって操作されるディレクトリの中にあるものの名前を要素とするリスト値が、@a という配列に代入されます。

次のプログラムは、コマンドライン引数でパス名を渡すと、そのパス名で指定されたディレクトリの一覧を出力します。

プログラムの例 readdir.pl

```
$pathname = $ARGV[0];
if (opendir $handle, $pathname) {
    @a = readdir $handle;
    print "@a\n";
    closedir $handle;
} else {
    print "$pathname: $!\n";
}
```

実行例

```
$ perl readdir.pl .
. .. readrir.pl
```

この実行結果から分かるように、readdir が読み込むディレクトリの一覧の中には、カレントディレクトリ (.) と親ディレクトリ (..) も含まれています。

8.5.6 名前単位での読み込み

readdir を呼び出す式を、リストコンテキストではなくて、スカラーコンテキストで評価すると、ファイルから 1 個の名前だけが読み込まれて、その名前がその式の値になります。そして、すべての名前が読み込まれたのちに readdir を呼び出すと、戻り値として、未定義値が返ってきます。

次のプログラムは、コマンドライン引数でパス名を渡すと、そのパス名で指定されたディレクトリの中にあるすべてのものについて、その名前と大きさ（バイト数）を出力します。

プログラムの例 filesize.pl

```
$pathname = $ARGV[0];
if (opendir $handle, $pathname) {
    while ($dir = readdir $handle) {
        print "$dir: ";
        @s = stat $pathname . "/" . $dir;
        print "$s[7]\n";
```

```
    }
    closedir $handle;
} else {
    print "$pathname: $!\n";
}
```

実行例

```
$ perl filesize.pl .
.: 0
..: 0
filesize.pl: 252
```

8.5.7 ファイルテスト演算子

ディレクトリの中にあるものを処理するとき、その多くの場合に、処理しようとしているものがファイルなのかディレクトリなのかということによって異なる処理をする必要があります。

Perl では、「ファイルテスト演算子」(ile test operator) と呼ばれる演算子を使うことによって、パス名が示しているものがファイルなのかディレクトリなのかということを判定することができます。

ファイルテスト演算子にはさまざまなものがあるのですが、ファイルなのかディレクトリなのかということを判定するために使われるのは、次の二つです。

`-f` ファイルならば真、そうでなければ偽。

`-d` ディレクトリならば真、そうでなければ偽。

ファイルテスト演算子は前置演算子ですので、たとえば、

```
-f $pathname
```

という式を書くことによって、`$pathname` をパス名とするものがファイルなのかどうかを調べることができます。

次のプログラムは、コマンドライン引数でパス名を渡すと、そのパス名で指定されたディレクトリの中にあるすべてのものについて、その名前と、ファイルなのかディレクトリなのかということを出力します。

プログラムの例 `filetest.pl`

```
$pathname = $ARGV[0];
if (opendir $handle, $pathname) {
    while ($dir = readdir $handle) {
        print "$dir: ";
        $pathname2 = $pathname . "/" . $dir;
        if (-f $pathname2) {
            print "ファイル";
        } elsif (-d $pathname2) {
            print "ディレクトリ";
        }
        print "\n";
    }
    closedir $handle;
} else {
    print "$pathname: $!\n";
}
```

実行例

```
$ mkdir namako
$ perl filetest.pl .
.: ディレクトリ
..: ディレクトリ
filetest.pl: ファイル
namako: ディレクトリ
```

8.5.8 再帰的なディレクトリの処理

サブルーチンを再帰的に定義することによって、指定されたディレクトリだけではなくて、そのディレクトリの中にあるディレクトリも処理することができます。

次のプログラムは、コマンドライン引数でパス名を渡すと、そのパス名で指定されたディレクトリの中にあるすべてのディレクトリについて、その中にあるファイルのパス名を出力します。

プログラムの例 recursivedir.pl

```
&recursivedir($ARGV[0]);

sub recursivedir {
    my $pathname = $_[0];
    if (opendir my $handle, $pathname) {
        while (my $dir = readdir $handle) {
            if ($dir ne "." and $dir ne "..") {
                my $pathname2 = $pathname . "/" . $dir;
                if (-f $pathname2) {
                    print "$pathname2\n";
                } elsif (-d $pathname2) {
                    &recursivedir($pathname2);
                }
            }
        }
        closedir $handle;
    } else {
        print "$pathname: $!\n";
    }
}
```

第 9 章　正規表現

9.1　正規表現の基礎

9.1.1 正規表現とは何か

文字列を処理するときには、しばしば、その中に含まれている特定のパターン（構造）を探し出す必要が生じます。そして、パターンを探し出すという処理を記述するためには、そのパターンそのものを記述する方法が必要になります。

文字列を使うことによって、文字列のパターンを記述するということができます。文字列を使って文字列のパターンを記述するための言語としては、「正規表現」(regular expression) と呼ばれるものがよく使われます。「正規表現」というのは言語の名前ですが、この言語を使って作られた文字列のことも、「正規表現」と呼ばれます。

正規表現によってあらわされるパターンと文字列とが一致することを、正規表現（またはそれがあらわしているパターン）と文字列とが「マッチする」(match) と言います（名詞は「マッチング」(matching) です）。

プログラミング言語の多くは、その一部分として正規表現を含んでいます。正規表現は、プログラミング言語ごとの違いがほとんどありませんので、一度覚えてしまえば、別のプログラミング言語のために覚え直す必要は、ほとんどありません。

9.1.2 正規表現の基礎の基礎

1 個の特定の文字というパターンをあらわす正規表現は、たいていの場合、その文字そのものです。たとえば、A という文字は、A という正規表現によってあらわされます。

特定のパターンのうしろに特定のパターンが続いているという構造のことを「連接」(sequence) と呼びます。連接は、正規表現と正規表現とをパターンの順番のとおりに並べることによって表現することができます。たとえば、pi という正規表現は、p という文字のうしろに i という文字が続いているというパターン、つまり pi という文字列をあらわします。

ですから、たいていの場合、特定の文字列というパターンをあらわす正規表現は、それとまったく同じ文字列になります。たとえば、kamome という文字列は、kamome という正規表現によってあらわされます。

9.1.3　マッチするかどうかを調べる演算子

正規表現によってあらわされるパターンとマッチする部分文字列が文字列の中に含まれているかどうかを調べたいときは、次のどちらかの演算子を使います。

`=~`　マッチする部分文字列が存在するならば真、存在しないならば偽。
`!~`　マッチする部分文字列が存在しないならば真、存在するならば真。

これらの演算子を含む式は、

式　演算子　正規表現

と書きます。この中の「式」のところには、パターンとマッチする部分文字列が存在するかどうかを調べる対象となる文字列を求める式を書いて、「正規表現」のところには、調べたいパターンをあらわす正規表現を書きます。ただし、正規表現は、

/ 正規表現 /

というように、その前後にスラッシュ(`/`) を書く必要があります。

これらの演算子が求める真偽値は、文字列コンテキストでは、真は `1` という文字列、偽は空文字列に変換されます。

```
  DB<1> p "namako" =~ /ma/
1
  DB<2> p "namako" =~ /ga/

  DB<3>
```

9.1.4　正規表現オプション

正規表現を使って文字列を検索する場合、デフォルトでは、次のような規約のもとで検索が実行されます。

- 文字列を先頭から末尾へ向かって検索して、最初にマッチした時点で検索を終了する。
- 英字の大文字と小文字を区別する。
- 検索の対象が複数行の文字列だとしても、途中に含まれている改行の直前を行末、直後を行頭とみなさない。

これとは異なる規約のもとで検索を実行したい場合は、その規約を記述した文字列を使います。そのような文字列は、「正規表現オプション」(regular expression option) と呼ばれます。

正規表現オプションは、次の文字から構成される文字列です（文字を並べる順序に意味はありません）。

`g`　文字列の全体をグローバル (`global`) に検索する。
`i`　大文字と小文字を区別しない (`ignore case`)。
`m`　検索の対象が複数行 (`multiline`) の文字列の場合、途中に含まれている改行の直前を行末、直後を行頭とみなす。

たとえば、`gi` または `ig` という文字列は、大文字と小文字を区別しないで文字列の全体を検索するという意味を持つ正規表現オプションです。

正規表現オプションは、正規表現の右側に書くスラッシュのさらに右側に書きます。

```
  DB<1> p "NaMaKo" =~ /ma/

  DB<2> p "NaMaKo" =~ /ma/i
1
```

9.2　メタ文字

9.2.1　メタ文字の基礎

正規表現を構成するそれぞれの文字は、基本的には、自分自身というパターンを意味しています。しかし、特定の文字列ではなくて、いくつかの文字列とマッチするようなパターンを表現するためには、自分自身ではない特別な意味をいくつかの文字に与える必要があります。

正規表現を書くために使われる、自分自身ではなくて何か別のことを意味している文字は、「メタ文字」(metacharacter) と呼ばれます。

どの文字をメタ文字として使うのかというのは、プログラミング言語ごとに多少の違いがありますが、

```
\ . [ ] - ^ $ * + ? { } ( ) |
```

というような文字が使われていて、それぞれのメタ文字の意味も、ほぼ統一されています。

9.2.2 文字クラス

文字の集合は、「文字クラス」(character class) と呼ばれます。正規表現は、指定された文字クラスに属する任意の文字というパターンを表現することができます。

9.2.3 すべての文字

すべての文字の集合という文字クラスに属する任意の文字というパターンは、ドット (.) というメタ文字によってあらわされます。たとえば、 ma.iko という正規表現は、 ma と iko とのあいだに 1 個の任意の文字がある、というパターンをあらわしています。

```
  DB<1> p "mariko" =~ /ma.iko/
1
  DB<2> p "makiko" =~ /ma.iko/
1
  DB<3> p "machiko" =~ /ma.iko/

  DB<4>
```

9.2.4 文字の列挙

文字を列挙することによって文字クラスを指定したいときは、角括弧 ([]) というメタ文字を使います。文字クラスに属する文字を角括弧の中に列挙したものは、その文字クラスに含まれる任意の文字というパターンをあらわす正規表現になります。たとえば、 ma[mkr]iko という正規表現は、 ma と iko とのあいだに、 m 、 k 、 r のうちのいずれかが 1 個だけある、というパターンをあらわしています。

```
  DB<1> p "mariko" =~ /ma[mkr]iko/
1
  DB<2> p "makiko" =~ /ma[mkr]iko/
1
  DB<3> p "magiko" =~ /ma[mkr]iko/

  DB<4>
```

9.2.5 文字コードの範囲

文字を列挙することによって文字クラスを指定するのではなくて、文字コードの範囲を指定することによって文字クラスを指定する、ということも可能です。

文字コードの範囲で文字クラスを指定したいときは、マイナス (-) というメタ文字を使います。角括弧の中に、

[文字 $_1$] - [文字 $_2$]

という形のものを書くと、それは、文字 $_1$ から文字 $_2$ までという文字コードの範囲に含まれるすべての文字を列挙したことと同じ意味になります。たとえば、 [a-z] という正規表現は、任意の英字の小文字というパターンをあらわします。

```
  DB<1> p "mariko" =~ /ma[a-z]iko/
1
  DB<2> p "magiko" =~ /ma[a-z]iko/
1
  DB<3> p "maRiko" =~ /ma[a-z]iko/

  DB<4>
```

9.2.6　文字クラスに属さない文字

文字クラスを指定する方法には、それに属する文字について記述するという方法のほかに、それに属さない文字について記述するという方法もあります。

それに属さない文字を記述することによって文字クラスを指定したいときは、サーカムフレックス (^) というメタ文字を使います。左角括弧の直後にサーカムフレックスを書くと、文字クラスは、それに属する文字によって記述されるのではなくて、それに属さない文字によって記述されることになります。たとえば、[^r] という正規表現は、r という文字を除いたすべての文字から構成される文字クラスに属する任意の文字、というパターンをあらわします。

```
  DB<1> p "mamiko" =~ /ma[^r]iko/
1
  DB<2> p "mariko" =~ /ma[^r]iko/

  DB<3>
```

サーカムフレックスとマイナスとを組み合わせることも可能です。

```
  DB<1> p "maRiko" =~ /ma[^a-z]iko/
1
  DB<2> p "mariko" =~ /ma[^a-z]iko/

  DB<3>
```

9.2.7　文字クラスの略記法

文字クラスのうちで、しばしば使われるいくつかのものについては、次のような略記法があります。

略記法	もとの正規表現	説明
\d	[0-9]	数字
\D	[^0-9]	数字以外
\w	[0-9A-Za-z]	英数字
\W	[^0-9A-Za-z]	英数字以外
\s	[\t\n\r\f]	ホワイトスペース
\S	[^ \t\n\r\f]	ホワイトスペース以外

9.2.8　パターンの繰り返し

メタ文字を使うことによって、同じパターンがいくつも繰り返されている、というパターンをあらわす正規表現を作ることも可能です。

0 回以上の繰り返しを表現したいときは、アスタリスク (*) というメタ文字を使います。何らかの正規表現の直後にアスタリスクを書いたものは、その正規表現によってあらわされたパターンが 0 回以上繰り返されたもの、というパターンを意味する正規表現になります。たとえば、m* という正規表現は、長さが 0 以上の m の列、というパターンをあらわします。

```
  DB<1> p "mammmmmmmiko" =~ /mam*iko/
1
  DB<2> p "maiko" =~ /mam*iko/
1
```

同じように、[mk]* という正規表現は、m または k から構成される、長さが 0 以上の文字列とマッチします。

```
  DB<1> p "mamkmmkkmkmiko" =~ /ma[mk]*iko/
1
```

1 回以上の繰り返しとはマッチするけれども、繰り返しの回数が 0 回のときはマッチしないようにしたい、というときは、アスタリスクの代わりにプラス (+) というメタ文字を使います。たとえば、m+ という正規表現は、長さが 1 以上の m の列、というパターンをあらわします。

```
  DB<1> p "mammmmmmmiko" =~ /mam+iko/
1
  DB<2> p "maiko" =~ /mam+iko/

  DB<3>
```

0 回と 1 回の繰り返しだけとマッチする正規表現を書きたい、というときは、クエスチョンマーク (?) というメタ文字を使います。たとえば、 m? という正規表現は、 m の 0 回または 1 回の繰り返しというパターンをあらわします。

```
  DB<1> p "maiko" =~ /mam?iko/
1
  DB<2> p "mamiko" =~ /mam?iko/
1
  DB<3> p "mammiko" =~ /mam?iko/

  DB<4>
```

繰り返しの回数を整数で指定したいときは、中括弧 ({}) というメタ文字を使います。たとえば、 [0-9]{4} という正規表現は、4 桁の数字というパターンをあらわします。

```
  DB<1> p "ma8325iko" =~ /ma[0-9]{4}iko/
1
  DB<2> p "ma832iko" =~ /ma[0-9]{4}iko/

  DB<3> p "ma83257iko" =~ /ma[0-9]{4}iko/

  DB<4>
```

繰り返しの対象となる正規表現の範囲は、繰り返しをあらわす記述の直前にあるものだけ、という点に注意してください。つまり、 mi+ という正規表現で繰り返しの対象になるのは、 mi ではなくて、 i だということです。

9.2.9 グループ化

いくつかの正規表現の列をひとつの正規表現にすることを、いくつかの正規表現の「グループ化」(group) と呼びます。

丸括弧 (()) というメタ文字でいくつかの正規表現を囲むと、それらの正規表現はグループ化されます。たとえば、 (mi) という正規表現は、 m という正規表現と i という正規表現をグループ化します。

=~ という演算子は、正規表現の中にグループ化された正規表現が含まれていて、その正規表現にマッチする部分文字列を発見した場合、グループ化された正規表現にマッチした部分文字列を求めます。

```
  DB<1> p "xxxxx387xxxxx" =~ /([0-9]+)/
387
```

グループ化された正規表現を含む正規表現に、 g という正規表現オプションを付加すると、グループ化された正規表現にマッチしたすべての部分文字列を連結した文字列が得られます。

```
  DB<1> p "xxxxx387xxxxx546xxxxx" =~ /([0-9]+)/g
387546
```

このような式をリストコンテキストで評価すると、その値として、マッチしたすべての部分文字列から構成されるリストが得られます。

```
  DB<1> @a = "xxxxx387xxxxx546xxxxx" =~ /([0-9]+)/g

  DB<2> p "@a"
387 546
```

正規表現の中に、グループ化された正規表現が 2 個以上含まれている場合は、それらのグループ化された正規表現にマッチした部分文字列を連結した文字列が得られます。

```
  DB<1> p "xxxxx387xxxxx546xxxxx" =~ /([0-9]+)x*([0-9]+)/
387546
```

このような式も、リストコンテキストで評価すると、その値として、グループ化された正規表現にマッチした部分文字列から構成されるリストが得られます。

```
  DB<1> @a = "xxxxx387xxxxx546xxxxx" =~ /([0-9]+)x*([0-9]+)/

  DB<2> p "@a"
387 546
```

第 9.2.8 項の最後のところで、「繰り返しの対象となる正規表現の範囲は、繰り返しをあらわす記述の直前にあるものだけ、という点に注意してください」と書きましたが、「直前にあるもの」の範囲は、いくつかの正規表現をグループ化することによって、いくらでも拡大することが可能です。たとえば、(mi)+ という正規表現は、mi というパターンを 1 回以上繰り返したもの、というパターンをあらわします。

```
  DB<1> p "mamimimiko" =~ /ma(mi)+ko/
mi
  DB<2> p "mamiiiko" =~ /ma(mi)+ko/

  DB<3>
```

9.2.10　パターンの選択

二つのパターンのうちのどちらか、というパターンの選択を表現したいときは、縦棒 (|) というメタ文字を使います。

縦棒を使って選択を記述したいときは、

正規表現の列 | 正規表現の列

という形の正規表現を書きます。そうすると、縦棒の左右に書かれたそれぞれの正規表現の列があらわしているパターンのどちらか、という意味の正規表現になります。たとえば、a|b という正規表現は、a または b のどちらか、というパターンをあらわします。選択の対象がさらに選択であってもかまいませんので、a|b|c という正規表現を書くことによって、a または b または c のいずれか、というパターンをあらわすことも可能です。

縦棒による選択の範囲は、その直前と直後の正規表現だけではなくて、そのさらに左や右にも及ぶ、という点に注意してください。つまり、ab|cd という正規表現は、abd または acd というパターンをあらわしているのではなくて、ab または cd というパターンをあらわしているのだということです。

選択の対象となる正規表現の範囲を狭く限定したいときは、その範囲を、丸括弧 (()) を使ってグループ化します。たとえば、a(b|c)d という正規表現は、abd または acd というパターンをあらわします。同じように、ma(ri|sa)ko という正規表現は、mariko または masako というパターンをあらわします。

```
  DB<1> p "mariko" =~ /ma(ri|sa)ko/
ri
  DB<2> p "masako" =~ /ma(ri|sa)ko/
sa
  DB<3> p "madoko" =~ /ma(ri|sa)ko/

  DB<4>
```

9.2.11　アンカー

パターンが文字列の中の特定の位置にあるときだけマッチする正規表現を書きたいときは、文字列の中の特定の位置にマッチする正規表現を使います。そのような正規表現は、「アンカー」(anchor) と呼ばれます。

たとえば、サーカムフレックス (^) というメタ文字は、文字列の先頭という位置とマッチするアンカーです。

```
  DB<1> p "123xxxxxxx" =~ /^123/
1
  DB<2> p "xxx123xxxx" =~ /^123/

  DB<3>
```

ドルマーク ($) というメタ文字は、文字列の末尾という位置とマッチするアンカーです。

```
  DB<1> p "xxxxxxx123" =~ /123$/
1
  DB<2> p "xxx123xxxx" =~ /123$/

  DB<3>
```

m という正規表現オプションが正規表現に付加されている場合、サーカムフレックスは文字列

の先頭だけではなくて行の先頭にもマッチして、ドルマークは文字列の末尾だけではなくて行の末尾にもマッチします。

```
  DB<1> p "xxx\n123xxxx" =~ /^123/

  DB<2> p "xxx\n123xxxx" =~ /^123/m
1
  DB<3> p "xxx123\nxxxxx" =~ /123$/

  DB<4> p "xxx123\nxxxxx" =~ /123$/m
1
```

9.2.12 エスケープ

ところで、メタ文字自身をあらわす正規表現、つまり特定のメタ文字だけとマッチする正規表現は、どのように書けばいいのでしょうか。たとえば、ドット (.) というメタ文字とマッチする正規表現というのは、いったいどう書くのでしょう。

メタ文字は、自分自身という意味を持っていませんので、メタ文字自身をあらわす正規表現を書くためには、メタ文字が持っている本来の意味を、その文字自身という意味に変更しないといけません。文字が持っている本来の意味を別の意味に変更することを、文字を「エスケープする」(escape) と言います。

文字をエスケープしたいときは、バックスラッシュ(\ ) というメタ文字を使います。バックスラッシュというのは、その直後に書かれた文字をエスケープするという意味を持つメタ文字です。

バックスラッシュでメタ文字をエスケープすると、その文字の意味は、メタ文字としての意味から、その文字自身という意味に変更されます。たとえば、ドット (.) というのはメタ文字ですので、ドット自身ではなくてメタ文字としての意味を持っています。しかし、\. というように、バックスラッシュの右側にドットを書くと、その 2 文字は、ドット自身をあらわす正規表現になります。

```
  DB<1> p "mariko" =~ /ma.iko/
1
  DB<2> p "mariko" =~ /ma\.iko/

  DB<3> p "ma.iko" =~ /ma\.iko/
1
```

ちなみに、バックスラッシュ自身もメタ文字ですから、バックスラッシュ自身をあらわす正規表現は、\\ と書く必要があります。

また、スラッシュ(/) という文字は、メタ文字ではありませんが、文字列が正規表現だということを示すための特別な文字ですので、正規表現の中でスラッシュ自身を記述するためには、\/ というように、バックスラッシュを使ってエスケープしないといけません。

```
  DB<1> p "ma/iko" =~ /ma\/iko/
1
```

9.2.13 後方参照

開始タグと終了タグで作られた HTML の要素にマッチする正規表現というのは、どのように書けばいいのでしょうか。つまり、

<要素型名>内容</要素型名>

というパターンを正規表現でどう書くか、という問題です。開始タグの要素型名と終了タグの要素型名とは、同一でないといけないわけですが、それをどのように記述すればいいのでしょうか。

この問題を解決するためには、左にある正規表現にマッチした部分文字列をあらわす正規表現、というものを書く必要があります。そのような正規表現は、「後方参照」(backreference) と呼ばれます。

後方参照を書くためには、まず、パターンにマッチした部分文字列を記録する必要があります。そのために必要なことは、正規表現のグループ化です。正規表現をグループ化すると、その正規表現にマッチした部分文字列は、後方参照のために記録されるのです。

記録された部分文字列をあらわす正規表現は、

\1、\2、\3、……

と書きます。1、2、3、……という数字は、グループ化された正規表現を左から右へ数えたときの番号です。たとえば、(.)\1 という正規表現は、2 個の同じ文字が連続するというパターンをあらわしています。

```
  DB<1> p "abcdefghijklmn" =~ /(.)\1/

  DB<2> p "abcdefgghijklmn" =~ /(.)\1/
g
```

同じように、(.)(.)\2\1 というパターンは、esse のような、「2 個の同じ文字が連続していて、それらの左にある文字と右にある文字も同じである」というパターンをあらわします。

```
  DB<1> p "abcdefgghijklmn" =~ /(.)(.)\2\1/

  DB<2> p "abcdefggfhijklmn" =~ /(.)(.)\2\1/
fg
```

ですから、HTML の要素をあらわす正規表現は、

```
<([a-z]+)>[^<]*<\/\1>
```

と書くことができます。

```
  DB<1> p "<title>Frozen</title>" =~ /<([a-z]+)>[^<]*<\/\1>/
title
  DB<2> p "<title>Frozen</table>" =~ /<([a-z]+)>[^<]*<\/\1>/

  DB<3>
```

9.3　部分文字列の置換

9.3.1　部分文字列の置換の基礎

パターンにマッチした部分文字列を別の文字列に置き換えるという処理を、部分文字列の「置換」(replace) と呼びます。

部分文字列を置換したいときは、

変数名 =~ s/正規表現/文字列/

という形の式を書きます。この中の「変数名」のところには、部分文字列の置換の対象となる文字列が格納されている変数の名前を書いて、「正規表現」のところには、置換したい部分文字列にマッチするパターンをあらわす正規表現を書いて、「文字列」のところには、パターンにマッチした部分文字列を置き換える文字列を書きます。

この形の式を評価すると、部分文字列の置換が実行されて、その結果として得られた文字列が、もとの変数に代入されます。

```
  DB<1> $a = "mikako"

  DB<2> $a =~ s/ka/yo/

  DB<3> p $a
miyoko
```

9.3.2　すべての部分文字列の置換

パターンにマッチするすべての部分文字列の置換を実行するためには、g という正規表現オプションを正規表現に付加する必要があります。g を付加しなかった場合は、左から右へ検索していって、さいしょにパターンとマッチした部分文字列だけが置換されます。

部分文字列の置換の場合、正規表現オプションは、

s/正規表現/文字列/

という記述の末尾に書きます。たとえば、

```
$a =~ s/ma/be/g
```

という式は、$a という変数に格納されている文字列に含まれている ma という部分文字列を、すべて、be に置換します。

```
  DB<1> $a = "namamuginamagomenamatamago"

  DB<2> $a =~ s/ma/zeru/

  DB<3> p $a
nazerumuginamagomenamatamago
  DB<4> $a =~ s/ma/zeru/g

  DB<5> p $a
nazerumuginazerugomenazerutazerugo
```

9.3.3 マッチした部分文字列を使った置換

正規表現にマッチした部分文字列を、その部分文字列に置換する文字列の一部分として使いたい、ということがしばしばあります。

そのような場合は、`$&` という名前の変数を使います。この変数には、正規表現にマッチした部分文字列が発見された場合、その部分文字列が格納されます。たとえば、

```
$a =~ s/[0-9]+/'$&'/g
```

という式は、`[0-9]+` という正規表現とマッチしたすべての部分文字列を、それを引用符で囲んだ文字列に置換します。

```
  DB<1> $a = "xxxxx372xxxxx583xxxxx"

  DB<2> $a =~ s/[0-9]+/'$&'/g

  DB<3> p $a
xxxxx'372'xxxxx'583'xxxxx
```

`$&` のような、特殊な用途で使われる変数は、「特殊変数」(special variable) と呼ばれます。

9.3.4 後方参照による置換

後方参照は、置換でも使うことができます。

後方参照は、正規表現の中では、

```
\1、\2、\3、……
```

という正規表現を書くことによって記述するわけですが、置換する文字列の中で、マッチした部分文字列を使いたいときは、

```
$1、$2、$3、……
```

という特殊変数を使います。1、2、3、……という数字は、グループ化された正規表現を左から右へ数えたときの番号です。たとえば、

```
$a =~ s/<([^>]*)>/'$1'/g
```

という式は、小なり (`<`) で始まって大なり (`>`) で終わるすべての部分文字列を、それを引用符で囲んだ文字列に置換します。

```
  DB<1> $a = "xxxxx<namako>xxxxx<umiushi>xxxxx"

  DB<2> $a =~ s/<([^>]*)>/'$1'/g

  DB<3> p $a
xxxxx'namako'xxxxx'umiushi'xxxxx
```

9.4 正規表現を扱う組み込み関数

9.4.1 この節について

Perl の処理系には、正規表現を扱う演算子だけではなくて、正規表現を扱う関数も組み込まれています。この節では、そのような組み込み関数を紹介したいと思います。

9.4.2 配列からの抽出

grep という組み込み関数は、それぞれの要素に文字列が格納されている配列から、何らかの正規表現と一致する部分文字列を持つものだけを抽出します。

grep は、引数として、正規表現 r と、抽出の対象となる配列 a を受け取って、a の要素に格納されている文字列のうちで、r とマッチする部分文字列を持つものだけから構成されるリスト値を戻り値として返します。正規表現は、スラッシュ(/) で囲む必要があります。

```
DB<1> @a = ("xxj", "xjx", "hhj", "jxx", "bbj", "hjh")

DB<2> @b = grep(/j$/, @a)

DB<3> p "@b"
xxj hhj bbj
```

9.4.3 文字列の分割

split という組み込み関数は、何らかのパターンを持つ部分文字列を区切り文字として、文字列をいくつかの部分に分割します。

split は、引数として、正規表現 r と、分割の対象となる文字列 s を受け取って、r とマッチした部分文字列を区切り文字列とみなして s を分割して、それぞれの部分から構成されるリスト値を戻り値として返します。正規表現は、スラッシュ(/) で囲む必要があります。

```
DB<1> @a = split(/:/, "ika:tako:ebi:uni")

DB<2> p "@a"
ika tako ebi uni
```

split に対して、3 個目の引数としてプラスの整数を渡すと、split は、その整数を分割の最大数として文字列を分割します。たとえば、3 個目の引数として 2 を渡すと、split は、正規表現とマッチする部分文字列を左から右へ検索して、最初に見つかった部分文字列の左右で文字列を 2 分割します。

```
DB<1> @a = split(/:/, "ika:tako:ebi:uni", 2)

DB<2> p "@a"
ika tako:ebi:uni
```

split に対して、1 個目の引数として、正規表現ではなくて空文字列を渡すと、split は、自身を個々の文字に分割して、1 文字の文字列から構成されるリスト値を返します。

```
DB<1> @a = split("", "isoginchaku")

DB<2> p "@a"
i s o g i n c h a k u
```

参考文献

[結城,2006] 結城浩、『新版・Perl 言語プログラミングレッスン・入門編』、ソフトバンククリエイティブ、2006、ISBN 978-4-7973-3680-1。

索引