Picat 実習マニュアル

第零版 alpha04

大黒学

Picat 実習マニュアル・第零版 alpha04
著者——大黒学

2017 年　9 月 25 日（月）　第零版 alpha04 発行

目次

第 1 章　Picat 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、Python、Haskell、Swift、……というように、枚挙にいとまがないほどです。

1.1.4　この文章について

この文章（「Picat 実習マニュアル」）は、Picat というプログラミング言語を使って、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。

1.2　言語処理系

1.2.1　言語処理系の基礎

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 Picat の言語処理系

Picat には、現在、Windows 版、macOS 版、Linux 版の言語処理系があります。

Picat の言語処理系は、`http://picat-lang.org/download.html` からダウンロードすることができます。

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、Picat の言語処理系を使って、Picat のプログラムを実行してみましょう。

まず、何らかのテキストエディターを使って、Picat のプログラムを入力して、それをファイルに保存します。 ちなみに、Picat のプログラムを保存するファイルに付ける拡張子は、 `.pi` です。

それでは、次のプログラムを入力して、`hello.pi` という名前のファイルに保存してください。

```
main => println('Hello, World!').
```

このプログラムは、`Hello, World!` という文字列を出力して、さらに改行を出力する、という動作をします。

1.2.4 プログラムの実行

Picat で書かれたプログラムは、

`picat` パス名

というコマンドによって、Picat の言語処理系に実行させることができます。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（拡張子の `.pi` は省略することができます)。

それでは、先ほど入力したプログラムを言語処理系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 macOS 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、

```
picat hello
```

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

```
Hello, World!
```

という文字列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、実行しようとしたプログラムにエラーが含まれていた場合、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 Picat 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.pi`

```
main => brintln('Hello, World!').
```

先ほどのプログラムとの相違点は、`println` という正しい名前が、`brintln` という間違った名前に変わっているというところです。このプログラムを Picat の言語処理系に実行させると、

言語処理系は、次のようなエラーメッセージを出力します。

```
*** Undefined procedure: brintln/1
```

このエラーメッセージは、brintln というのが何の名前なのか分からない、ということを述べています。

1.3 REPL

1.3.1 REPL の基礎

言語処理系は、「REPL」と呼ばれるものと、そうでないものとに分類することができます。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、という動作をする処理系のことです。

(1) プログラムまたはその断片を読み込む (read)。
(2) 読み込んだプログラムまたはその断片を実行する (eval)。
(3) 結果を出力する (print)。

ですから、REPL を使うことによって、プログラムまたはその断片をキーボードから直接入力して、それがどのように動作するかということを即座に確かめる、ということができます。

1.3.2 REPL の起動

Picat の言語処理系も、REPL として動作させることができます。
Picat の言語処理系は、コマンドライン引数を何も書かずに、

```
picat
```

というコマンドをシェルに入力することによって起動した場合、REPL として動作します。
それでは、実際に、Picat の言語処理系を REPL として動作するように起動してみてください。そうすると、

```
Picat>
```

というプロンプトが表示されるはずです。

1.3.3 質問

Picat の REPL に入力されるものは、「質問」(query) と呼ばれます。
たとえば、help という質問を Picat の REPL に入力すると、REPL の使い方が表示されます。
それでは、ヘルプを出力させてみましょう。help という質問を入力して、そののちエンターキーを押してみてください。

```
Picat> help
 cl(File)           -- compile and load the module File.pi
（中略）
yes
```

Picat の REPL は、質問が入力された場合、その質問に対応する動作を実行したのち、yes または no という回答を出力します。yes というのは質問を肯定する回答で、no というのは質問を否定する回答です。
たとえば、Picat の REPL は、5 = 5 という質問に対しては yes と回答し、5 = 8 という質問に対しては no と回答します。

```
Picat> 5 = 5
yes
Picat> 5 = 8
no
```

1.3.4 REPL の終了

Picat の REPL は、halt または exit という質問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、実際に REPL を終了させてみてください。REPL が終了すると、Linux や macOS 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ます。

1.3.5 コンパイルとロード

Picat の REPL は、ファイルに保存されているプログラムをコンパイルして、その結果としてできた機械語のプログラムを読み込む、ということもできます。REPL が機械語のプログラムを読み込むことを、プログラムを「ロードする」(load) と言います。

ファイルに保存されているプログラムをコンパイルしてロードしたいときは、

```
cl( ファイル名 )
```

という形の質問を REPL に入力します。この中の「ファイル名」のところには、コンパイルしてロードしたいプログラムが格納されているファイルの名前から、拡張子の .pi を取り除いたものを書きます。たとえば、

```
cl(hello)
```

という質問を入力することによって、hello.pi というファイルに格納されている Picat のプログラムを、コンパイルしてロードすることができます。

```
Picat> cl(hello)
Compiling:: hello.pi
hello.pi compiled in 4 milliseconds
loading...
yes
```

ロードしたプログラムは、main という質問を REPL に入力することによって実行することができます。

```
Picat> main
Hello, World!
yes
```

1.4 空白と改行と注釈

1.4.1 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、Picat のプログラムの意味に影響を与えません。たとえば、

```
main => println('Hello, World!').
```

というプログラムは、

```
main   =>   println(   'Hello, World!'   )   .
```

と書いたとしても同じ意味になりますし、

```
main
=>
println(
'Hello, World!'
).
```

と書いたとしても同じ意味になります。

ただし、一重引用符で囲まれた部分に空白を挿入した場合は、その空白はプログラムの意味に影響を与えます。たとえば、

```
main => println('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、空白も改行も挿入することはできません。ですから、println という名前を、

```
p r i n t l n
```

と書くことはできません。

1.4.2 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。

しかし、プログラミング言語の文法を無視してプログラムの中に注釈を書くと、たいていの場合、そのプログラムは実行することができなくなります。

次のプログラムは、Picat の文法を無視して Picat のプログラムの中に注釈を書いたものです。入力して、言語処理系に実行させてみましょう。

プログラムの例 comment.pi

```
main => I am a comment. ('Hello, World!').
```

注釈は、言語処理系が、「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。

Picat には、「この部分は注釈である」ということを言語処理系に認識してもらう方法が、二つあります。

そのうちのひとつは、パーセント (%) という文字を使うという方法です。プログラムの中に % を書くと、その直後から最初の改行までが注釈だと認識されます。たとえば、Picat の言語処理系は、

```
% I am a comment.
```

という記述を、注釈だと認識します。

それでは、パーセントから改行までの部分が本当に注釈だと認識されるかどうかを確かめてみましょう。次のプログラムを入力して、言語処理系に実行させてみてください。

プログラムの例 percent.pi

```
main => % I am a comment.
println('Hello, World!').
```

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、パーセントと改行のあいだに書かれていますので、Picat の言語処理系はその部分を注釈だと認識します。

プログラムの一部分を注釈として認識してもらう方法の二つ目は、スラッシュアスタリスク (/*) とアスタリスクスラッシュ(*/) でそれを囲むという方法です。たとえば、Picat の言語処理系は、

```
/* I am a comment. */
```

という記述を、注釈だと認識します。

それでは、スラッシュアスタリスクからアスタリスクスラッシュまでの部分が本当に注釈だと認識されるかどうかを確かめてみましょう。次のプログラムを入力して、言語処理系に実行させてみてください。

プログラムの例 slaster.pi

```
main => /* I am a comment. */ println('Hello, World!').
```

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、スラッシュアスタリスクとアスタリスクスラッシュのあいだに書かれていますので、Picat の言語処理系はその部分を注釈だと認識します。

改行を含んでいる注釈、つまり 2 行以上の注釈も、その全体を /* と */ で囲むことによって、注釈だと認識してもらうことができます。たとえば、Picat の言語処理系は、

```
/* I am a comment
which includes a newline. */
```

という記述を、注釈だと認識します。

1.4.3 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　式の基礎

2.1.1　式と評価と値

Picat のプログラムの中には、「式」(expression) と呼ばれるものを書くことができます。

式というのは、コンピュータによる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、式があらわしている動作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。

2.1.2　REPL による式の評価

式は、それ自体は質問ではありませんので、REPL に式を入力したとしても、REPL には、それをどうすればいいのか分かりません。

REPL に式を評価してほしいときは、

```
X = 式
```

という形の質問を入力します。たとえば、

```
X = 5+3
```

という質問を REPL に入力すると、REPL は、`5+3` という式を評価して、その値を出力します。

```
Picat> X = 3+5
X = 8
yes
```

2.1.3　式の構造

式というのは、プログラムというものを組み立てるための部品のようなものだと考えることができます。

多くの場合、式という部品は、より単純な式を組み合わせることによって作られています。たとえば、`5+3` というのはひとつの式ですが、この式は、より単純な式を組み合わせることによって作られています。

`5+3` という式の中にある `5` という部分は、5 という整数を求めるという動作をあらわしている式です。したがって、REPL に対して `X = 5` という質問を入力すると、`5` という式が評価されて、その値が出力されます。

```
Picat> X = 5
X = 5
yes
```

同じように、`3` という部分も、3 という整数を求めるという動作をあらわしている式です。つまり、`5+3` という式は、`5` という式と `3` という式を、`+` というものをあいだにはさんで結びつけることによってできているわけです。

どんな式でも、それ自体を、さらに複雑な式の部品にすることができます。たとえば、`5+3` という式を部品にして、`5+3-7` という式を作ることができます。式というのは、組み合わせることによっていくらでも複雑なものを作ることができるという、そんな性質を持っている部品なのです。

2.2 リテラル

2.2.1 リテラルの基礎

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、`481` のような、何個かの数字を並べることによってできる列は、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、`481` というリテラルを評価すると、それによって生成された 481 という整数のデータが、その値として得られます。

```
Picat> X = 481
X = 481
```

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、整数そのものではなくて整数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、整数のデータのことを単に「整数」と呼ぶことがあります。

2.2.2 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。

10 進数リテラル以外の整数リテラルは、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`、8 進数は `0o`、2 進数は `0b` です（`x` と `o` と `b` は大文字でもかまいません）。たとえば、`0xff`、`0o377`、`0b11111111` は、どれも、255 という整数を生成します。

```
Picat> X = 0xff
X = 255
yes
Picat> X = 0o377
X = 255
yes
Picat> X = 0b11111111
X = 255
```

2.2.3 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。

`0.003`、`41.56`、`723.0` というような、ドット (`.`) という文字の左右に数字の列を書いたものは、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。

```
Picat> X = 3.14
X = 3.14
```

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
`3.0e8`	3.0×10^8
`6.022e23`	6.022×10^{23}
`6.626e-34`	6.626×10^{-34}

```
Picat> X = 3.0e8
X = 300000000.0
```

2.2.4 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。

```
Picat> X = -56
X = -56
```

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 2.5 節で説明することになる「演算子」(operator) と呼ばれるものです。

2.3 アトム

2.3.1 アトムの基礎

Picat では、名前によって指示される個々の「ものごと」のことを「アトム」(atom) と呼びます。そして、アトムに与えられた名前は、「アトム名」(atom name) と呼ばれます。

アトム名は、「非引用名」(unquoted name) と「引用名」(quoted name) の 2 種類に分類することができます。

2.3.2 非引用名の作り方

非引用名は、次のような規則に従って作ることになっています。

- 非引用名を作るために使うことのできる文字は、英字、数字、アンダースコア (_) です。
- 非引用名の先頭の文字は、英字の小文字でないといけません。

非引用名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。

```
x  x8  namako  backToTheFuture  back_to_the_future
```

最後の例のように、アンダースコアは、複数の単語から構成される非引用名を作るときに、空白の代わりとして使うことができます。

英字の大文字と小文字は区別されますので、たとえば、 ab と aB は、それぞれを異なる非引用名として使うことができます。

非引用名として使うことのできないものの例としては、次のようなものがあります。

`nam@ko`	使うことのできない文字 (@) を含んでいる。
`8x`	先頭の文字が数字。
`Namako`	先頭の文字が英字の大文字。
`_namako`	先頭の文字がアンダースコア。

ちなみに、先頭の文字が英字の大文字またはアンダースコアになっている名前は、アトムの名前ではなくて、「変数」(variable) と呼ばれるものになります。変数については、第 3.5 節で説明することにしたいと思います。

2.3.3 引用名の作り方

引用名は、任意の文字から構成される文字列を一重引用符 (’) で囲むことによって作られます。
引用名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。

```
'X'  '8X'  'Namako'  '_namako'  'nam@ko'
```

非引用名を一重引用符で囲んだ引用名は、その中の非引用名と同じ名前だとみなされます。たとえば、 namako という非引用名と、 'namako' という引用名は、同じ名前だとみなされます。

2.3.4 エスケープシーケンス

文字の中には、たとえば改行や一重引用符のように、そのままでは引用名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む引用名を作りたいときには、「エスケープシーケンス」(escape sequence) と呼ばれる文字列が使われます。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。

\'	一重引用符	\"	二重引用符
\t	水平タブ	\n	改行
\r	キャリッジリターン	\\	バックスラッシュ

引用名の中に書かれたエスケープシーケンスは、それが意味している文字を引用名の中に組み込みます。
たとえば、 '\n\'\\' は、改行と一重引用符とバックスラッシュから構成される引用名です。

2.3.5 アトム名の評価

すべてのアトム名は、式として評価することができます。アトム名を評価することに得られる値は、そのアトム名が与えられているアトムです。

```
Picat> X = namako
X = namako
yes
```

2.3.6 単一文字アトム

1 個の文字から構成される非引用名、あるいは、1 個の文字または 1 個のエスケープシーケンスを一重引用符で囲んだ引用名を名前とするアトムは、「単一文字アトム」(single-character atom) と呼ばれます。
たとえば、次のような名前が与えられたアトムは、単一文字アトムです。

```
a  'A'  '\n'
```

2.4 関数呼び出し

2.4.1 関数

Picat では、式を書くことによって動作させることが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。

2.4.2 引数と項数

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、引数を何個でも受け取ることができます。関数が受け取る引数の個数は、その関数の「項数」(arity) と呼ばれます。

[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。

項数が異なる二つ以上の関数には、同一の名前を与えることができます。ですから、特定の関数に言及するときには、

関数名 / 項数

という形のものを書くことによって、関数名だけではなくて、さらに項数が併記されることもあります。たとえば、`hoge/3` というのは、名前が `hoge` で項数が 3 の関数という意味です。

2.4.3 戻り値

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。

関数を呼び出してそれに引数としてデータを渡すことを、関数をデータに「適用する」(apply) と言うこともあります。

2.4.4 ユーザー定義関数と組み込み関数

Picat では、「関数定義」(function definition) と呼ばれる記述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（関数定義の書き方については、第 4.5 節で説明することにしたいと思います)。プログラムの中に関数定義を書くことによって定義された関数は、「ユーザー定義関数」(user-defined function) と呼ばれます。

「ユーザー定義関数」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、Picat の言語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、関数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、Picat の言語処理系には、`abs` 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 個の数値を引数として受け取って、その絶対値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、−5 という数値を引数として `abs` に渡したとすると、`abs` は、5 という数値を戻り値として返します。

2.4.5 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。

`abs` という関数を呼び出して、−5 という数値を引数として渡す関数呼び出しは、

```
abs(-5)
```

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 5 という数値が得られます。

```
Picat> X = abs(-5)
X = 5
```

2.4.6 数値を処理する組み込み関数

組み込み関数のうちで、数値を処理の対象とするものとしては、次のようなものがあります。

`abs`(X)	X の絶対値を返す。
`sign`(X)	X の符号を判定して、その結果を返す。戻り値は、引数がプラスならば 1、マイナスならば −1、0 ならば 0。
`max`(X, Y)	X と Y の最大値（小さくないほう）を返す。
`min`(X, Y)	X と Y の最小値（大きくないほう）を返す。
`ceiling`(X)	X よりも小さくない最小の整数を返す。
`floor`(X)	X よりも大きくない最大の整数を返す。
`round`(X)	X の小数点以下を四捨五入した結果を返す。
`trancate`(X)	X の小数点以下を切り捨てた結果を返す。

pow(X, Y)	X の Y 乗を返す。X **Y と同じ。
exp(X)	自然対数の底の X 乗を返す。
log(X)	X の自然対数の値を返す。
log10(X)	X の常用対数（10 を底とする対数）の値を返す。
log2(X)	X の二進対数（2 を底とする対数）の値を返す。
log(B, X)	B を底とする X の対数の値を返す。
sqrt(X)	X の平方根を返す。
sin(X)	X のサイン（正弦）を返す。
cos(X)	X のコサイン（余弦）を返す。
tan(X)	X のタンジェント（正接）を返す。
asin(X)	X のアークサイン（逆正弦）を返す。
acos(X)	X のアークコサイン（逆余弦）を返す。
atan(X)	X のアークタンジェント（逆正接）を返す。
atan2(X, Y)	X と Y の符合を考慮して、Y/X のアークタンジェント（逆正接）を返す。
to_radians(X)	X を度からラジアンに変換した結果を返す。
to_degrees(X)	X をラジアンから度に変換した結果を返す。
random()	整数の乱数を返す。
random2()	環境に依存する種を使って、整数の乱数を返す。
rand_max()	整数の乱数の最大値を返す。
random(S)	整数の乱数を返す。さらに乱数発生器に種として S を設定する。
frand()	0.0 から 1.0 までの範囲で浮動小数点数の乱数を返す。
gcd(N, M)	整数 N と整数 M の最大公約数 (greatest common divisor) を返す。

三角関数のメソッドが扱う角度の単位は、ラジアンです。

式の中では、e と書くことによって自然対数の底を求めることができて、pi と書くことによって円周率を求めることができます。

2.4.7 アトムを処理する組み込み関数

組み込み関数のうちで、アトムを処理の対象とするものとしては、次のようなものがあります。

len(A)	アトム A のアトム名を構成している文字の個数を返す。アトム名が引用名の場合、最初と最後の一重引用符は数えない。
length(A)	len/1 と同じ。
chr(N)	整数 N を UTF-8 での文字コードとする単一文字アトムを返す。
ord(A)	単一文字アトム A の UTF-8 での文字コードを返す。
to_int(D)	数字の単一文字アトム D を整数に変換した結果を返す。
to_integer(D)	to_int/1 と同じ。

```
Picat> X = len('nam@ko')
X = 6
yes
Picat> X = chr(65)
X = 'A'
yes
Picat> X = ord('A')
X = 65
yes
Picat> X = to_int('5')
X = 5
yes
```

2.5 演算子

2.5.1 演算子の基礎

Picat の組み込み関数の中には、「演算」(operation) と呼ばれるものもあります。演算に与えられた名前は、「演算子」(operator) と呼ばれます。

演算は、関数呼び出しではなくて、演算を呼び出すための特殊な式によって呼び出されます。このチュートリアルでは、演算を呼び出すための式のことを「演算子式」(operator expression) と呼ぶことにしたいと思います。

演算は、関数であるとは限りません。「述語」(predicate) と呼ばれるものの中にも、「演算」と呼ばれるものがあります。述語については、第 3.2 節で説明することにしたいと思います。

2.5.2 単項演算と二項演算

演算には、項数が 1 のものと 2 のものがあります。

項数が 1 の演算は「単項演算」(unary operation) と呼ばれ、そのような演算に与えられた名前は「単項演算子」(unary operator) と呼ばれます。

項数が 2 の演算は「二項演算」(binary operation) と呼ばれ、そのような演算に与えられた名前は「二項演算子」(binary operator) と呼ばれます。

2.5.3 二項演算を呼び出す演算子式

二項演算を呼び出してそれに引数を渡す演算子式は、

式 $_1$	演算子	式 $_2$

と書きます。この形の演算子式を評価すると、式 $_1$ と式 $_2$ の値が引数として演算に渡されて、その演算の戻り値が演算子式の値になります。

2.5.4 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。

X `+` Y　X と Y とを足し算（加算）する。

X `-` Y　X から Y を引き算（減算）する。

X `*` Y　X と Y とを掛け算（乗算）する。

X `/` Y　X を Y 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。

X `//` Y　X を Y で割り算（除算）する。X と Y は整数でなければならない。整数の範囲で割り切れない場合、小数点以下は切り捨てられる。

X `mod` Y　X を Y で除算したあまりを求める。X と Y は整数でなければならない。

X `**` Y　X の Y 乗（べき乗）を求める。

```
Picat> X = 30+7
X = 37
yes
Picat> X = 30-7
X = 23
yes
Picat> X = 30*7
X = 210
yes
Picat> X = 30/7
X = 4.285714285714286
yes
Picat> X = 30//7
X = 4
yes
Picat> X = 30 mod 7
X = 2
yes
Picat> X = 3**4
```

```
X = 81
```

2.5.5 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 `//` と `mod` 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。

```
Picat> X = 2+3*4
X = 14
```

さらに、`**` は、`*` と `/` と `//` と `mod` 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。

```
Picat> X = 2*3**4
X = 162
```

2.5.6 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。

`+`、`-`、`*`、`/`、`//`、`mod` 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。

```
Picat> X = 10-5+2
X = 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。

```
Picat> X = 2**3**4
X = 241785163922925834941235 2
```

2.5.7　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。

```
Picat> X = (2+3)*4
X = 20
yes
Picat> X = (2*3)**4
X = 1296
yes
Picat> X = 10-(5+2)
X = 3
yes
Picat> X = (2**3)**4
X = 4096
```

2.5.8　単項演算を呼び出す演算子式

単項演算を呼び出してそれに引数を渡す演算子式は、

```
演算子 式
```

と書きます。この形の演算子式を評価すると、演算子の右側に書かれた式の値が引数として演算に渡されて、その演算の戻り値が演算子式の値になります。

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。

2.5.9　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。

```
Picat> X = -(3+5)
X = -8
yes
Picat> X = -(3-5)
X = 2
```

```
yes
Picat> X = -3**2
X = -9
yes
Picat> X = (-3)**2
X = 9
```

第 3 章　ゴール

3.1　ゴールの基礎

3.1.1　ゴールとは何か

Picat のプログラムの中には、動作をあらわしている記述として、2 種類のものを書くことができます。そのうちのひとつは、第 2 章で説明した「式」(expression) と呼ばれる記述です。そして、動作をあらわしているもうひとつの記述は、「ゴール」(goal) と呼ばれます。

ゴールと式は、どちらも、プログラムという記述を構成している部品だと考えることができます。式というのは何らかのゴールの中に書かないといけませんので、ゴールというのは式よりも大きな部品だということになります。

式によってあらわされている動作を実行することを、式を「評価する」(evaluate) と言うわけですが、それに対して、ゴールによってあらわされている動作を実行することを、ゴールを「解決する」(resolve) と言います。

Picat の REPL に対して入力することができるもののことを「質問」(query) と呼ぶわけですが、質問というのは、ゴールである必要があります。

3.1.2　真偽値

成り立っているか成り立っていないかということを判定することのできる対象のことを、「命題」(proposition) と言います。命題が成り立っているとき、その命題は「真」(true) であると言い、命題が成り立っていないとき、その命題は「偽」(false) であると言います。

真と偽は、総称して「真偽値」(Boolean value) と呼ばれます。

Picat のゴールは、何らかの命題をあらわしていると考えることができます。そして、ゴールを解決すると、そのゴールがあらわしている命題の真偽値が判明します。

Picat の REPL に質問を入力したときに REPL が出力する `yes` または `no` というのは、質問として入力したゴールがあらわしている命題の真偽値に対応しています。ゴールが真だった場合は `yes` が出力され、ゴールが偽だった場合は `no` が出力されます。

3.2　述語呼び出し

3.2.1　述語

Picat では、ゴールを書くことによって動作させることができるもののことを「述語」(predicate) と呼びます。

述語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、述語を呼び出すという動作をあらわすゴールは、「述語呼び出し」(predicate call) と呼ばれます。

3.2.2　引数と項数

関数と同じように、述語も、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。関数の場合と同じように、述語が受け取るデータも、「引数」(argument) と呼ばれます。

述語は、引数を何個でも受け取ることができます。述語が受け取る引数の個数は、その述語の「項数」(arity) と呼ばれます。

項数が異なる二つ以上の述語には、同一の名前を与えることができます。ですから、特定の述語に言及するときには、

述語名 / 項数

という形のものを書くことによって、述語名だけではなくて、さらに項数が併記されることもあります。たとえば、hoge/3 というのは、名前が hoge で項数が 3 の述語という意味です。

3.2.3　成功と失敗

述語の動作は、「成功する」(succeed) または「失敗する」(fail) という結果を出して終了します。

述語呼び出しを解決することによって判明する真偽値は、その述語呼び出しによって呼び出された述語の動作が成功した場合は真になって、失敗した場合は偽になります。

3.2.4　ユーザー定義述語と組み込み述語

Picat では、「述語定義」(predicate definition) と呼ばれる記述をプログラムの中に書くことによって、述語を自由に定義することができます（述語定義の書き方については、第 4 章で説明することにしたいと思います)。プログラムの中に述語定義を書くことによって定義された述語は、「ユーザー定義述語」(user-defined predicate) と呼ばれます。

「ユーザー定義述語」の対義語は、「組み込み述語」です。「組み込み述語」(built-in predicate) というのは、Picat の言語処理系に組み込まれている述語のことです。組み込み述語は、述語定義を書かなくても、述語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、Picat の言語処理系には、int/1 という述語が組み込まれています。この組み込み述語は、1 個のデータを引数として受け取って、それが整数ならば成功して、そうでなければ失敗します。

3.2.5　述語呼び出しの書き方

述語呼び出しは、

述語名 (式 , ···)

と書きます。この中の「述語名」というところには、呼び出したい述語の名前を書きます。

述語呼び出しを解決すると、名前で指定された述語が呼び出されて、丸括弧の中に書かれた式の値が、その述語に引数として渡されます。

int/1 という述語を呼び出して、5 という数値を引数として渡す述語呼び出しは、

```
int(5)
```

と書くことができます。3.2.4 で紹介したように、int/1 は、引数が整数ならば成功して、そうでなければ失敗する述語ですので、このゴールを解決すると、真偽値として真が得られます。

```
Picat> int(5)
yes
Picat> int(5.8)
no
```

項数が 0 の述語を呼び出す場合、述語呼び出しの丸括弧は省略することができます。たとえば、hoge/0 という述語を呼び出す、hoge() という述語呼び出しは、丸括弧を省略して、hoge と書いてもかまいません。

ちなみに、項数が 0 の関数を呼び出す場合は、述語の場合とは違って、関数呼び出しの丸括弧を省略することはできません。

3.2.6　常に成功する組み込み述語と常に失敗する組み込み述語

true/0 という組み込み述語は、常に成功します。

それとは逆に、false/0 という組み込み述語は、常に失敗します。fail/0 も、常に失敗する組み込み述語です。

```
Picat> true
yes
Picat> false
no
Picat> fail
no
```

3.2.7　数値の性質を判定する組み込み述語

次の組み込み述語は、引数として受け取った数値の性質を判定します。

even(N)　N が偶数ならば成功する。
odd(N)　N が奇数ならば成功する。
prime(N)　N が素数ならば成功する。

3.2.8 データの種類を判定する組み込み述語

次の組み込み述語は、引数として受け取ったデータの種類を判定します。

int(T)　T が整数ならば成功する。
integer(T)　int/1 と同じ。
float(T)　T が浮動小数点数ならば成功する。
real(T)　float/1 と同じ。
number(T)　T が数値ならば成功する。
atom(T)　T がアトムならば成功する。
digit(T)　T が数字の単一文字アトムならば成功する。
atomic(T)　T が数値またはアトムならば成功する。

3.2.9 データを出力する組み込み述語

次の組み込み述語は、引数として受け取ったデータを標準出力に出力します。

print(T)　T を標準出力に出力する。
println(T)　T を標準出力に出力して、そののち改行を標準出力に出力する。
write(T)　T を標準出力に出力する。
writeln(T)　T を標準出力に出力して、そののち改行を標準出力に出力する。

print/1 と println/1 は、引用名の一重引用符を出力しません。
それに対して、 write/1 と writeln/1 は、引用名の一重引用符も出力します。ただし、非引用名を囲んでいる一重引用符は出力しません。
これらの述語は、常に成功します。

```
Picat> print('nam@ko')
nam@ko
yes
Picat> write('nam@ko')
'nam@ko'
yes
```

3.3 述語である演算

3.3.1 演算子ゴール

第 2.5 節で、「演算」(operation) と呼ばれるものと「演算子」(operator) と呼ばれるものについて説明して、組み込み関数のうちで演算であるものを紹介しましたが、演算というのは、関数であるとは限りません。組み込み述語の中にも、演算であるものが存在します。
関数である演算の場合と同じように、述語である演算も、述語呼び出しではなくて、それを呼び出すための特殊なゴールによって呼び出されます。このチュートリアルでは、述語である演算を呼び出すためのゴールのことを「演算子ゴール」(operator goal) と呼ぶことにしたいと思います。

3.3.2 演算子ゴールの書き方

述語である二項演算を呼び出す演算子ゴールは、

| ゴール $_1$ | 演算子 | ゴール $_2$ |

または

| 式 $_1$ | 演算子 | 式 $_2$ |

と書きます。

そして、述語である単項演算を呼び出す演算子ゴールは、

演算子 ゴール

と書きます。

3.3.3　論理演算子

述語である演算の中には、真偽値が与えられたときに、それを処理することによって真偽値を求める、という動作をするものがあります。そのような演算は、「論理演算」(logical operation) と呼ばれます。そして、論理演算に与えられた名前は、「論理演算子」(logical operator) と呼ばれます。

3.3.4　論理積

二つの真偽値が与えられたときに、その両方が真ならば真、そうでなければ偽を求める、という論理演算は、「論理積」(logical conjunction) と呼ばれます。

コンマ (,) は、論理積を求める二項演算子です。

ゴール$_1$, ゴール$_2$

というゴールを解決すると、まず最初にゴール$_1$ が解決されて、それが真だったならば次にゴール$_2$ が解決されて、それも真だったならば、ゴールの全体が真になります。ゴール$_1$ が偽だった場合、ゴール$_2$ は解決されないで、ゴールの全体が偽になります。

```
Picat> true, true
yes
Picat> true, false
no
Picat> false, true
no
Picat> false, false
no
Picat> true, print(namako)
namako
yes
Picat> false, print(namako)
no
```

&& も、 , と同じ論理演算をあらわす演算子です。

3.3.5　論理和

二つの真偽値が与えられたときに、それらのうちの少なくともひとつが真ならば真、両方とも偽ならば偽を求める、という論理演算は、「論理和」(logical disjunction) と呼ばれます。

セミコロン (;) は、論理和を求める二項演算子です。

ゴール$_1$; ゴール$_2$

というゴールを解決すると、まず最初にゴール$_1$ が解決されて、それが偽だったならば次にゴール$_2$ が解決されて、それも偽だったならば、ゴールの全体が偽になります。ゴール$_1$ が真だった場合、ゴール$_2$ は解決されないで、ゴールの全体が真になります。

```
Picat> true; true
yes
Picat> true; false
yes
Picat> false; true
yes
Picat> false; false
no
Picat> false; print(namako)
namako
yes
Picat> true; print(namako)
yes
```

||も、;と同じ論理演算をあらわす演算子です。

3.3.6 否定

ひとつの真偽値が与えられたときに、それが真ならば偽、偽ならば真を求める、という論理演算は、「否定」(negation) と呼ばれます。

not は、否定を求める単項演算子です。

not ゴール

というゴールを解決すると、その中のゴールが解決されて、それが真だったならばゴールの全体が偽になって、偽だったならばゴールの全体が真になります。

```
Picat> not true
no
Picat> not false
yes
```

3.3.7 論理演算子の優先順位

論理演算子は、Picat の演算子のうちで最も優先順位が低い演算子です。したがって、それらの演算子の左右に書くものを丸括弧で囲む必要が生じることは、めったにありません。

論理演算子のあいだでの優先順位は、;と||が最も低くて、次に低いのが,と&&で、次に低いのがnotです。

;を含む演算子ゴールを,の左右に書く場合や、,または;を含むゴールをnotの右に書く場合は、丸括弧が必要になります。

```
Picat> false, false; true
yes
Picat> false, (false; true)
no
Picat> not true, false
no
Picat> not (true, false)
yes
```

3.4 関係演算子

3.4.1 関係演算子の基礎

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、「関係演算子」(relational operator) と呼ばれます。

関係演算子は、ゴールを作る演算子です。

式 $_1$ 関係演算子 式 $_2$

という形のゴールを解決すると、式 $_1$ と式 $_2$ 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、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、ゴールの結果になります。

関係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。

3.4.2 大小関係

次の関係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。

X > Y　　X は Y よりも大きい。

X < Y　　X は Y よりも小さい。

X >= Y　　X は Y よりも大きいか、または X と Y とは等しい。

X <= Y　　X は Y よりも小さいか、または X と Y とは等しい。

```
Picat> 8 > 5
yes
Picat> 5 > 8
no
```

```
Picat> 5 > 5
no
Picat> 5 >= 5
yes
```

3.4.3　データが等しいかどうか

二つのデータが等しいかどうかということは、次の関係演算子を使うことによって調べることができます。

T `==` U　　T と U とは等しい。
T `!==` U　　T と U とは等しくない。

```
Picat> namako == namako
yes
Picat> namako == umiushi
no
Picat> namako !== namako
no
Picat> namako !== umiushi
yes
```

3.4.4　数値が等しいかどうか

`==` は、二つの等しい数値が与えられたとしても、それらの一方が整数で他方が浮動小数点数だという場合には、それらは等しくないと判断します。

それに対して、`=:=` という関係演算子は、二つの数値が数値として等しいかどうかを調べます。

```
Picat> 5 == 5.0
no
Picat> 5 =:= 5.0
yes
```

3.5　変数

3.5.1　変数の基礎

Picat のプログラムは、「変数」(variable) と呼ばれるものを扱うことができます。

Picat における変数というのは、何らかのデータを保持することができるもののことだと考えることができます。保持するべきデータを変数に与えることを、変数をデータに「束縛する」(bind) と言います。

変数をデータに束縛すると、その変数はそれ以降、それに束縛されたデータと同一視されることになります。データに束縛される以前の状態にある変数は、「自由である」(free) と言われ、自由である変数は「自由変数」(free variable) と呼ばれます。

3.5.2　変数名の作り方

変数は、「変数名」(variable name) と呼ばれる名前によって識別されます。

変数名は、次のような規則に従って作ることになっています。

- 変数名を作るために使うことのできる文字は、英字、数字、アンダースコア (`_`) です。
- 変数名の先頭の文字は、英字の大文字、またはアンダースコアでないといけません。

変数名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。

```
X  X8  Namako  _namako  Back_to_the_Future
```

最後の例のように、アンダースコアは、複数の単語から構成される変数名を作るときに、空白の代わりとして使うことができます。

英字の大文字と小文字は区別されますので、たとえば、`_a` と `_A` は、それぞれを異なる変数名として使うことができます。

変数名として使うことのできないものの例としては、次のようなものがあります。

`Nam@ko` 使うことのできない文字 (`@`) を含んでいる。
`8X` 先頭の文字が数字。
`namako` 先頭の文字が英字の小文字。

3.5.3 匿名変数

1 個のアンダースコアのみから構成される変数名、すなわち_という変数名は、「匿名変数名」(anonymous variable name) と呼ばれます。そして、匿名変数を名前として持つ変数は「匿名変数」(anonymous variable) と呼ばれます。

匿名変数は、いかなるデータにも束縛されません。また、ひとつのプログラムの中にいくつかの匿名変数名を書いた場合、それぞれの匿名変数名は、別々の匿名変数の名前だとみなされます。

3.6 単一化

3.6.1 単一化の基礎

Picat では、二つのものを同じものにしようと試みることを、それらを「単一化する」(unify) と言います（unify の名詞形は unification)。

単一化というのは、あくまで「試みること」ですので、かならず成功するとは限りません。試みたけれども失敗する、という可能性もあります。

= は、左右の式を評価することによって得られた値を単一化する二項演算子です。単一化に成功した場合の結果は真で、失敗した場合の結果は偽です。

3.6.2 データとデータとの単一化

データとデータとの単一化は、それらが同じデータならば成功して、異なるデータならば失敗します。たとえば、

```
5+3 = 6+2
```

というゴールを解決すると、左右の式の値はどちらも 8 ですので、このゴールの真偽値は真になります。また、

```
5+3 = 6+3
```

というゴールを解決した場合、左の式の値が 8 であるのに対して右の式の値は 9 ですので、このゴールの真偽値は偽になります。

```
Picat> 5+3 = 6+2
yes
Picat> 5+3 = 6+3
no
```

3.6.3 自由変数とデータとの単一化

第 3.5.1 項で説明したように、保持するべきデータを変数に与えることを、変数をデータに「束縛する」(bind) と言います。

自由変数とデータとを単一化すると、その自由変数はそのデータに束縛されます。

```
Picat> X = 8
X = 8
yes
```

すでに何らかのデータに束縛されている変数、すなわち自由ではない変数は、その変数が束縛されているデータと同一視されますので、それ以外のデータとの単一化は失敗します。

```
Picat> X = 8, X = 5
no
```

3.6.4 自由変数と自由変数との単一化

単一化と束縛という二つの概念は、よく似ています。自由変数とデータとの単一化というのは、自由変数をそのデータに束縛することだと考えることもできます。しかし、単一化と束縛とは、異なる概念です。

単一化というのは、「二つのものを同じものにしようと試みること」ですから、二つの別々の自由変数を単一化するということもできます。その場合は、どちらの自由変数も何かに束縛されるわけではなくて、単一化されたのちも自由変数のままです。

自由変数と自由変数とを単一化すると、それらの自由変数は一体化されますので、一方をデータに束縛すると、同時に他方も同じデータに束縛されます。

```
Picat> X = Y, X = 8
X = 8
Y = 8
yes
```

3.6.5　匿名変数の単一化

第 3.5.3 項で説明したように、匿名変数は、いかなるデータにも束縛されません。しかし、匿名変数と何かとの単一化は、常に成功します。

```
Picat> _ = 8
yes
Picat> _ = 8, _ = 5
yes
```

3.6.6　変数が自由かどうかの判定

次の組み込み述語は、引数として受け取った変数が自由かどうかを判定します。

`var(`T`)`　　T が自由変数ならば成功する。

`nonvar(`T`)`　T が自由変数ではないならば成功する。

```
Picat> var(X)
yes
Picat> X = 8, var(X)
no
Picat> nonvar(X)
no
Picat> X = 8, nonvar(X)
X = 8
yes
```

第 4 章　述語と関数の定義

4.1　述語と関数の定義の基礎

4.1.1　定義

述語と関数は、「定義」(definition) と呼ばれるものをプログラムの中に書くことによって、自由に作り出すことができます。

定義は、Picat のプログラムという記述を構成する、最も大きな部品です。

述語を作り出す定義は「述語定義」(predicate definition) と呼ばれ、関数を作り出す定義は「関数定義」(function definition) と呼ばれます。

述語や関数には、名前としてアトム名を与えることができます。

4.1.2　ルール

定義は、1 個以上の「ルール」(rule) と呼ばれるものから構成されます。

述語定義を構成するルールは、基本的には、

頭部 => 本体 .

と書きます。「頭部」のところに書くのは、述語呼び出しとそれとが一致するかどうかを調べることになるパターンです。そして、「本体」のところに書くのは、定義される述語の動作となるゴールです。

述語呼び出しとルールの頭部とが一致した場合、そのルールの本体となっているゴールが解決されます。このことを、ルールを述語呼び出しに「適用する」(apply) と言います。

4.1.3 引数を受け取らない述語の定義

引数を受け取らない述語、すなわち項数が 0 の述語を定義する場合、ルールの頭部のところに書くパターンは、その述語に与える名前です。

次のプログラムは、名前が world で項数が 0 の述語を定義する 1 個のルールから構成されています。

プログラムの例 world.pi

```
world => println('What a wonderful world!').
```

このプログラムは、REPL を使ってコンパイルしてロードすることによって、実行することができます。

cl/1 という組み込み述語は、ファイルに格納されている Picat のプログラムをコンパイルしてロードします。この組み込み述語に渡す引数は、Picat のプログラムが格納されているファイルのファイル名から、 .pi という拡張子を取り除いた部分です。たとえば、

```
cl(world)
```

というゴールを解決することによって、 world.pi というファイルに格納されている Picat のプログラムをコンパイルしてロードすることができます。

```
Picat> cl(world)
Compiling:: world.pi
world.pi compiled in 0 milliseconds
loading...
yes
```

これで、 world/0 という述語が定義されましたので、述語呼び出しを質問として REPL に入力することによって、それを呼び出すことができます。

```
Picat> world
What a wonderful world!
yes
```

4.1.4 失敗する述語の定義

ルールの本体というのはゴールですから、それが解決されると、その結果として真偽値が得られます。ルールによって定義された述語は、そのルールの本体が解決された結果として、真が得られた場合は成功して、偽が得られた場合は失敗します。

次のプログラムの中で定義されている mistake/0 という述語は、メッセージを出力したのちに、失敗して終了します。

プログラムの例 mistake.pi

```
mistake => println('What a wonderful mistake!'), fail.
```

それでは、 mistake/0 を呼び出してみましょう。

```
Picat> mistake
What a wonderful mistake!
no
```

Picat には、失敗するという動作だけをする組み込み述語として、 false/0 と fail/0 という二つのものがあります。プログラムの中でこれらの述語を使うときは、真偽値の偽を求めることが目的の場合は false/0 を使って、述語の動作を失敗させることが目的の場合は fail/0 を使う、というように使い分けをするとよいでしょう。

4.1.5 コマンドで実行されるプログラム

第 1.2.4 項で説明したように、Picat で書かれたプログラムは、Picat の言語処理系を実行するコマンドの引数として、そのプログラムのパス名を指定することによって、実行することができます。

ただし、Picat のプログラムを直接的にコマンドで実行することができるようにするためには、そのプログラムの中で、 main/0 または main/1 という述語を定義しておく必要があります。

main/0 は、Picat で書かれたプログラムを、

```
picat パス名
```

というコマンドで実行した場合に呼び出される述語です。それに対して、main/1 は、Picat で書かれたプログラムを、

picat [パス名] [引数] [引数] ・・・

というコマンドで実行した場合に呼び出される述語です。

main/1 が受け取る引数については、第 5.6 節で説明することにしたいと思います。

4.2　引数を受け取る述語の定義

4.2.1　引数を受け取るためのパターン

第 4.1.2 項で説明したように、ルールの頭部のところに書くのは、述語呼び出しとそれとが一致するかどうかを調べることになるパターンです。

引数を受け取る述語の定義、すなわち項数が 1 以上の述語を定義する場合、ルールの頭部のところには、

[述語名]([パターン], ・・・)

という形のものを書きます。丸括弧の中には、いくつかのパターンをコンマで区切って並べます。丸括弧の中に書かれたパターンの個数が、定義される述語の項数になります。

パターンとして書くことができるものは、数値、アトム、変数名など、多種多様です。

述語呼び出しとルールの頭部とが一致した場合、そのルールはその述語呼び出しに適用されます。

次のプログラムは、引数が 50 ならば成功する、fifty/1 という述語を定義します。

プログラムの例　fifty.pi

```
fifty(50) => true.
```

実行例

```
Picat> fifty(50)
yes
Picat> fifty(51)
no
```

4.2.2　複数のルールから構成される述語定義

述語定義は、2 個以上のルールから構成されることもあります。

述語定義が 2 個以上のルールから構成されている場合、それらのうちで頭部が述語呼び出しと一致するものが、述語呼び出しに適用されます。

次のプログラムは、引数が、spring、summer、autumn、winter のいずれかならば成功する、season/1 という述語を定義します。

プログラムの例　season.pi

```
season(spring) => true.
season(summer) => true.
season(autumn) => true.
season(winter) => true.
```

実行例

```
Picat> season(autumn)
yes
Picat> season(night)
no
```

4.2.3　パターンとしての変数名

ルールの頭部の中には、パターンとして変数名を書くこともできます。その場合、変数名によって識別される変数は、自由変数として、それと一致した述語呼び出しの中のものと単一化されます。

変数名と一致したものがデータの場合、その変数名によって識別される変数は、一致したデータに束縛されます。

次のプログラムは、引数が 3 の倍数ならば成功する、 multhree/1 という述語を定義します。

プログラムの例 multhree.pi

```
multhree(N) => N mod 3 =:= 0.
```

実行例

```
Picat> multhree(12)
yes
Picat> multhree(14)
no
```

次のプログラムは、2 個目の引数が 1 個目の引数の 2 乗ならば成功する、 square/2 という述語を定義します。

プログラムの例 square.pi

```
square(X, Y) => Y = X*X.
```

実行例

```
Picat> square(3, 9)
yes
Picat> square(3, 8)
no
```

ルールの頭部の中に書かれた変数名と、述語呼び出しの中に書かれた変数名とが一致して、しかも述語呼び出しの中の変数名によって識別される変数が自由変数だという場合は、自由変数と自由変数とが単一化されることになります。

その場合、もしもルールの本体の解決によって、ルールの側の変数がデータに束縛されたならば、述語呼び出しの側の変数も、同時にそのデータに束縛されることになります。

たとえば、

```
square(3, A)
```

という述語呼び出しで square/2 を呼び出したとすると、ルールの中の X という変数は 3 に束縛されて、 Y という変数は A という変数と単一化されます。そののち、 Y は 9 に束縛されますので、それと同時に、 A も 9 に束縛されることになります。ですから、 square/2 を使うことによって、任意の数値の 2 乗を求めることができます。

実行例

```
Picat> square(3, A)
A = 9
yes
```

ただし、 square/2 を使って任意の数値の平方根を求める、ということはできません。たとえば、

```
square(A, 9)
```

という述語呼び出しで square/2 を呼び出した場合、 X と A が単一化されて、 Y が 9 に束縛されるわけですが、そののち、 X*X という式を評価する段階でエラーになります。なぜなら、この場合の X は自由変数のままですが、自由変数を算術演算の対象にすることはできないからです。

実行例

```
Picat> square(A, 9)
*** error(instantiation_error,* /2)
```

4.3 条件

4.3.1 条件の基礎

第 4.1.2 項で説明したように、ルールは、基本的には、

頭部 => 本体 .

と書くわけですが、頭部と => のあいだには、「条件」(condition) と呼ばれるものを書くこともできます。ただし、頭部と条件は、コンマで区切る必要があります。つまり、

頭部 , 条件 => 本体 .

という形のルールを書くこともできる、ということです。

条件は、1 個のゴールです。このゴールは、述語呼び出しと頭部とが一致した場合に解決されます。

条件を伴わないルールは、述語呼び出しと頭部とが一致した場合にはかならず述語呼び出しに適用されるわけですが、条件を伴うルールは、述語呼び出しと頭部とが一致したとしても、かならず述語呼び出しに適用されるとは限りません。適用されるのは、条件の解決が成功した場合だけです。条件の解決が失敗した場合、ルールは述語呼び出しに適用されません。

次のプログラムは、引数が 0 よりも大きいならばそれを出力して、引数が 0 以下ならば何も出力しないで失敗する、 priplus/1 という述語を定義します。

プログラムの例　priplus.pi

```
priplus(X), X > 0 => println(X).
```

実行例

```
Picat> priplus(500)
500
yes
Picat> priplus(-500)
no
```

4.3.2　条件の有用性

先ほど紹介した priplus/1 の定義は、条件を使わずに、

```
priplus(X) => X > 0, println(X).
```

と書くことも可能です。したがって、先ほどの述語定義は、条件というものの有用性を何も示していません。

条件が有用性を発揮するのは、2 個以上のルールによって述語を定義する場合です。

述語定義が 2 個以上のルールから構成されている場合、ルールの頭部と述語呼び出しとが一致しているかどうかということは、最も上に書かれているルールから、順番に下に向かって判定されていきます。そして、ルールが述語呼び出しに適用された場合、そのルールよりも下に書かれている、まだ判定されていないルールは、原則として、判定の対象から除外されます[1]。

条件を伴うルールは、その頭部と述語呼び出しが一致したとしても、条件の解決に失敗した場合は述語呼び出しに適用されませんので、そのルールの下に書かれているルールは、依然として判定の対象であり続けます。

次のプログラムは、1 個目の引数が数値で、2 個目の引数が自由変数の場合、1 個目の引数が 0 よりも大きいならば 2 個目の引数を plus というアトムに束縛して、1 個目の引数が 0 よりも小さいならば 2 個目の引数を minus というアトムに束縛して、そのどちらでもないならば 2 個目の引数を zero というアトムに束縛する、 sign/2 という述語を定義します。

プログラムの例　sign.pi

```
sign(X, S), X > 0 => S = plus.
sign(X, S), X < 0 => S = minus.
sign(_, S)        => S = zero.
```

実行例

```
Picat> sign(500, A)
A = plus
yes
Picat> sign(-500, A)
```

[1] まだ判定されていないルールが判定の対象から除外されないようにすることも可能です。その方法については、第 4.4 節で説明します。

```
A = minus
yes
Picat> sign(0, A)
A = zero
yes
```

sign/2 の定義を構成している 3 個目のルールは、_という変数名を使っています。これは、第 3.5.3 項で説明した、「匿名変数名」と呼ばれる変数名です。

ところで、sign/2 を定義するときに、条件を使わないで、X > 0 や X < 0 というゴールを本体に移動させると、どうなるのでしょうか。試してみましょう。

プログラムの例 sign2.pi

```
sign2(X, S) => X > 0, S = plus.
sign2(X, S) => X < 0, S = minus.
sign2(_, S) =>        S = zero.
```

実行例

```
Picat> sign2(500, A)
A = plus
yes
Picat> sign2(-500, A)
no
Picat> sign2(0, A)
no
```

この述語定義の場合、1 個目のルールが述語呼び出しに適用された段階で、2 個目と 3 個目のルールは判定の対象から除外されますので、1 個目の引数が 0 以下だった場合は、失敗して終わることになります。

4.4 バックトラック

4.4.1 バックトラックの基礎

第 4.3.2 項で説明したように、述語定義が 2 個以上のルールから構成されている場合、ルールの頭部と述語呼び出しとが一致しているかどうかということは、最も上に書かれているルールから、順番に下に向かって判定されていきます。そして、ルールが述語呼び出しに適用された場合、そのルールよりも下に書かれている、まだ判定されていないルールは、原則として、判定の対象から除外されます。

ルールが判定の対象から除外されるということについては、脚注で述べたとおり、除外されないようにすることも可能です。

ルールの頭部と本体のあいだには、=> というものを書くわけですが、その場所には、=> の左側にクエスチョンマークを加えた、?=> というものを書くこともできます。

=> ではなくて ?=> を使うと、そのルールが述語呼び出しに適用されたとしても、そのルールよりも下に書かれている、まだ判定されていないルールは、依然として判定の対象であり続けます。

ルールが述語呼び出しに適用されて、その本体の解決が成功したとすると、そのルールが => を使っている場合も、?=> を使っている場合も、同じように、そこで述語の動作は終了します。

ルールが述語呼び出しに適用されて、その本体の解決が失敗したとすると、そのルールが => を使っている場合は、そこで述語の動作は終了します。しかし、そのルールが ?=> を使っている場合は、もしも、そのルールの下にさらにルールが書かれているならば、それらのルールに対して、頭部と述語呼び出しとが一致するかどうかの判定が続行されます。

ルールの本体の解決が失敗したときに、それとは別のルールについて頭部と述語呼び出しとが一致するかどうかを判定するという処理に戻ることを、「バックトラックする」(backtrack) と言います。そして、?=> を使っているルールは、「バックトラック可能なルール」(backtrackable rule) と呼ばれます。

4.4.2 バックトラック可能なルールを使った述語定義

まず、次のプログラムの中で定義されている backtrack1/0 という述語を呼び出してみましょう。

プログラムの例 backtrack1.pi

```
backtrack1 => println('first rule'), fail.
backtrack1 => println('secand rule'), fail.
backtrack1 => println('third rule'), fail.
```

実行例

```
Picat> backtrack1
first rule
no
```

この述語の動作から分かるとおり、=> を使って書かれたルールが述語呼び出しに適用された場合、それよりも下に書かれているルールは、頭部と述語呼び出しとが一致するかどうかを判定する対象から除外されます。ですから、=> を使って書かれたルールの本体が解決に失敗した場合、そこで述語の動作は終了します。

次に、次のプログラムの中で定義されている backtrack2/0 という述語を呼び出してみましょう。

プログラムの例　backtrack2.pi

```
backtrack2 ?=> println('first rule'), fail.
backtrack2 ?=> println('secand rule'), fail.
backtrack2 ?=> println('third rule'), fail.
```

実行例

```
Picat> backtrack2
first rule
secand rule
third rule
no
```

backtrack2/0 は、backtrack1/0 のルールで使われている => を、?=> に変更したものです。呼び出した結果から分かるとおり、?=> を使って書かれたルールの本体が解決に失敗した場合、そのルールよりも下に、まだルールが残っているならば、そのルールの頭部と述語呼び出しとが一致するかどうかが判定され、一致したならば、そのルールが述語呼び出しに適用されます。

バックトラック可能なルールが述語呼び出しに適用された場合、そのルールの本体が解決に成功したとしても、まだ判定されないで残っているルールは、判定を待っている状態で温存されます。

さらに、次のプログラムの中で定義されている backtrack3/0 という述語を呼び出してみましょう。

プログラムの例　backtrack3.pi

```
backtrack3 ?=> println('first rule').
backtrack3 ?=> println('secand rule').
backtrack3 ?=> println('third rule').
```

この述語定義は、必ず成功するルールから本体が構成されていますので、述語呼び出しだけを質問として REPL に入力した場合、述語呼び出しに適用されるのは最初のルールだけです。

実行例

```
Picat> backtrack3
first rule
yes
```

しかし、コンマで接続された右側のゴールの解決が失敗した場合は、まだ判定されないで残っているルールにバックトラックします。そして、判定の対象として残されているルールが存在しなくなると、述語呼び出しの解決は失敗して終了します。

実行例

```
Picat> backtrack3, fail
first rule
secand rule
third rule
no
```

4.4.3 非決定的な述語

解答が 1 個だけとは限らない述語は、「非決定的である」(nondeterministic) と言われます。

非決定的な述語は、バックトラック可能なルールを使うことによって、定義することができます。

たとえば、次のプログラムの中で定義されている backtrack4/1 という述語は、 one 、 two 、 three という 3 個の解答を持つ述語です。

プログラムの例 backtrack4.pi

```
backtrack4(X) ?=> X = one.
backtrack4(X) ?=> X = two.
backtrack4(X) ?=> X = three.
```

REPL に対して、

```
backtrack4(A)
```

という質問を入力すると、

```
A = one ?
```

という解答が出力されて、そこで入力待ちになります。ここでセミコロン (;) を入力してエンターキーを押すと、次のルールにバックトラックして、

```
A = two ?
```

というように次の解答が出力されます。さらにセミコロンを入力していくことによって、すべての解答を求めることができます。

セミコロンを手で入力する代わりに、 fail/0 を使って次のルールにバックトラックさせることもできます。

実行例

```
Picat> backtrack4(A), println(A), fail
one
two
three
no
```

4.5 関数定義

4.5.1 関数定義の基礎

第 4.1.1 項で説明したように、述語と同様に、関数も、「関数定義」(function definition) と呼ばれるものをプログラムの中に書くことによって、自由に作り出すことができます。

関数定義も、述語定義と同様に、1 個以上の「ルール」(rule) から構成されます。

関数定義を構成するルールは、基本的には、

頭部 = 式 => 本体 .

と書きます。「頭部」のところに書くのは、関数呼び出しとそれとが一致するかどうかを調べることになるパターンで、「本体」のところに書くのは、定義される関数の動作となるゴールです。そして、「式」のところに書くのは、関数が返す戻り値を求める式です。

次のプログラムは、戻り値として常に 7 を返す、 seven/0 という関数を定義しています。

プログラムの例 seven.pi

```
seven = X => X = 7.
```

実行例

```
Picat> A = seven()
A = 7
yes
```

述語定義の場合とは違って、関数定義では、バックトラック可能なルール、つまり、頭部と本体のあいだに => ではなくて ?=> を書いたルールを使うことはできません。

4.5.2　関数事実

関数を定義するルールが、

頭部 = X => X = 式 .

という形になっている場合、このルールは、

頭部 = 式 .

と書くこともできます。この形のルールは、「関数事実」(function fact) と呼ばれます。

次のプログラムは、先ほどの seven/0 と同じ動作をする seven2/0 という関数を、関数事実を使って定義しています。

プログラムの例　seven2.pi

```
seven2 = 7.
```

実行例

```
Picat> A = seven2()
A = 7
yes
```

4.5.3　引数を受け取る関数の定義

引数を受け取る関数を定義したいときは、ルールの頭部のところに、

関数名 (パターン , ･･･)

という形のものを書きます。丸括弧の中に、パターンとして変数名を書いておくと、頭部と関数呼び出しとが一致した場合に、その変数名によって識別される変数は、自由変数として、それと一致した関数呼び出しの中のものと単一化されます。

次のプログラムは、1 個の数値を引数として受け取って、それを 10 倍した結果を戻り値として返す、 tentimes/1 という関数を定義しています。

プログラムの例　tentimes.pi

```
tentimes(X) = X * 10.
```

実行例

```
Picat> A = tentimes(8)
A = 80
yes
```

4.5.4　条件を伴うルールによる関数の定義

述語定義の場合と同じように、関数定義でも、条件を伴うルールを使うことができます。

関数定義では、条件を伴うルールは、

頭部 = 式 , 条件 => 本体 .

と書きます。

次のプログラムは、1 個の整数を引数として受け取って、それが偶数ならば even というアトムを、奇数ならば edd というアトムを戻り値として返す、 evenodd/1 という関数を定義しています。

プログラムの例　evenodd.pi

```
evenodd(N) = X, N mod 2 =:= 0 => X = even.
evenodd(_) = odd.
```

実行例

```
Picat> A = evenodd(6)
A = even
yes
Picat> A = evenodd(7)
```

```
A = odd
yes
```

4.6 再帰

4.6.1 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。

4.6.2 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。

4.6.3 述語と関数の再帰的な定義

述語と関数は、再帰的に定義することが可能です。

述語を再帰的に定義するというのは、定義される当の述語を使って述語を定義するということです。同じように、関数を再帰的に定義するというのも、定義される当の関数を使って関数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述語または関数は、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。

述語または関数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ルールを書いておくことが必要になります。

4.6.4 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。

階乗を求める述語も、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/2` という述語を再帰的に定義しています。

プログラムの例　factorial.pi

```
factorial(0, F)          => F = 1.
factorial(N, F), N >= 1 => factorial(N-1, F1), F = F1*N.
```

実行例

```
Picat> factorial(5, X)
X = 120
yes
```

4.6.5　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。

二つの整数の最大公約数は、「ユークリッドの互除法」(Euclidean algorithm) と呼ばれる方法を使えば、きわめて簡単に求めることができます。

ユークリッドの互除法は、二つの整数を n と m とするとき、次のように再帰的に記述することが可能です。

- 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、m と、n を m で除算したときのあまりと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（一方は 0 でもよい）の最大公約数を求める gcm/2 という関数を、ユークリッドの互除法を使って再帰的に定義しています。

プログラムの例　gcm.pi

```
gcm(N, 0) = N.
gcm(N, M) = G, M >= 1 => G = gcm(M, N mod M).
```

実行例

```
Picat> A = gcm(54, 36)
A = 18
yes
```

4.6.6　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 項を加算した結果である、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 項を求める述語も、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/2 という述語を再帰的に定義しています。

プログラムの例　fibonacci.pi

```
fibonacci(0, F)          => F = 1.
fibonacci(1, F)          => F = 1.
fibonacci(N, F), N >= 2 =>
    fibonacci(N-2, F2),
    fibonacci(N-1, F1),
    F = F2+F1.
```

実行例

```
Picat> fibonacci(7, A)
A = 21
yes
```

4.6.7 テーブリング

先ほど紹介した、再帰を使ってフィボナッチ数列の第 n 項を求める述語は、それほど実用的なものとは言えません。なぜなら、n が大きくなればなるほど、再帰的に述語を呼び出す回数が爆発的に増えていくからです。試しに、第 200 項を求めてみてください。おそらく、なかなか結果が出力されないと思います。

Picat は、「テーブリング」(tabling) と呼ばれる機能を持っています。これは、述語または関数を呼び出すことによって得られた結果を記憶しておいて、同じ呼び出しが起きた場合はその結果を利用する、という機能です。再帰を使って定義された述語または関数は、この機能を使うことによって、結果を求める時間を短縮することができます。

フィボナッチ数列の第 n 項を再帰的に求める述語は、テーブリングを使うことによって、結果が求まるまでの時間を大幅に短縮することができます。

テーブリングを利用したいときは、述語定義の直前に、`table` という単語を書きます。

再帰を使ってフィボナッチ数列の第 n 項を求める述語は、次のように定義を書くことによって、テーブリングを利用することができるようになります。

プログラムの例　`table.pi`

```
table
fibonacci(0, F)          => F = 1.
fibonacci(1, F)          => F = 1.
fibonacci(N, F), N >= 2 =>
    fibonacci(N-2, F2),
    fibonacci(N-1, F1),
    F = F2+F1.
```

実行例

```
Picat> fibonacci(200, A)
A = 453973694165307953197296969697410619233826
yes
```

テーブリングを使うと、それを使わない場合に比べてメモリーを余分に消費します。ですから、それを使うかどうかという決定をするためには、メモリーの消費という犠牲を払う価値が、結果を求める時間の短縮という効果に見合うものかどうか、ということについて判断する必要があります。

第 5 章　複合項

5.1　複合項の基礎

5.1.1 複合項とは何か

Picat は、「複合項」(compound term) と呼ばれるものを扱うことができます。複合項というのは、複数のデータから構成されるデータのことです。

変数は、単純なデータだけではなくて、複合項に束縛することもできます。

Picat では、単純なデータまたは複合項という概念を、「項」(term) と呼びます。

5.1.2 複合項の分類

複合項は、次の 4 種類のものに分類することができます。

- リスト (list)
- 配列 (array)
- マップ (map)

- ストラクチャー (structure)

5.2　リスト

5.2.1　リストとは何か

リスト (list) というのは、項を並べることによってできる列のことだと考えることができます。

リストを構成している個々の項は、そのリストの「要素」(element) と呼ばれます。そして、リストを構成している要素の個数は、そのリストの「長さ」(length) と呼ばれます。

5.2.2　リスト式

リストは、「リスト式」(list expression) と呼ばれる式を評価することによって生成することができます。

リスト式というのは、基本的には、0 個以上の式をコンマで区切って並べて、その全体を角括弧で囲んだもの、つまり、

[式, 式, ･･･]

という形の式のことです。リスト式を評価すると、その中の個々の式が評価されて、それらの式の値を、式と同じ順序で並べたリストが生成されて、そのリストが、そのリスト式の値になります。

たとえば、

```
[namako, 81, 7.63]
```

というリスト式を評価すると、その値として、namako というアトム、81 という整数、7.63 という浮動小数点数を、この順序で並べることによってできるリストが生成されて、そのリストが、このリスト式の値になります。

```
Picat> X = [namako, 81, 7.63]
X = [namako,81,7.63]
yes
```

5.2.3　空リスト

長さが 0 のリスト、つまり 0 個の要素から構成されるリストは、「空リスト」(empty list) と呼ばれます。

空リストは、リストであると同時にアトムでもあります。ですから、組み込み述語の atom/1 は、空リストはアトムであると判定します。

```
Picat> atom([])
yes
```

5.2.4　リストの頭部と尾部

空リスト以外の任意のリストは、「頭部」(head) と呼ばれる部分と「尾部」(tail) と呼ばれる部分に分解することができます。

リストの頭部というのは、リストを構成している要素のうちで、もっとも先頭にあるもののことです。そして、リストの尾部というのは、リストから頭部を取り除くことによってできるリストのことです。たとえば、

```
[a, b, c, d, e]
```

というリストは、a という頭部と、

```
[b, c, d, e]
```

という尾部に分解することができます。

5.2.5　頭部と尾部からのリストの生成

何らかのデータを頭部、何らかのリストを尾部とするリストを生成したいときは、

[式$_1$ | 式$_2$]

という形のリスト式を評価します。そうすると、その値として、式$_1$ の値を頭部、式$_2$ の値を尾部とするリストが得られます。

```
Picat> X = [a|[b, c, d, e]]
X = [a,b,c,d,e]
yes
```

5.2.6 リストを分解する単一化

リストを頭部と尾部に分解したいときは、

[変数名$_1$ | 変数名$_2$]

という形の式と、分解したいリストとを単一化します。そうすると、変数名$_1$ を名前とする変数がリストの頭部に束縛されて、変数名$_2$ を名前とする変数がリストの尾部に束縛されます。たとえば、

```
[H|T] = [a, b, c, d, e]
```

というゴールを解決すると、 H が a に束縛されて、 T が、

```
[b, c, d, e]
```

に束縛されることになります。

```
Picat> [H|T] = [a, b, c, d, e]
H = a
T = [b,c,d,e]
yes
```

長さが 1 のリスト、つまり要素の個数が 1 のリストを頭部と尾部に分解すると、尾部は空リストになります。

```
Picat> [H|T] = [a]
H = a
T = []
yes
```

5.2.7 範囲演算子

「範囲演算子」(range operator) と呼ばれる、 .. という演算子を使うことによって、一定の幅で増加していく数値の列（つまり等差数列）から構成されるリストを生成することができます。

.. は、 $I..J$ という式で呼び出すと、

I、$I+1$、$I+2$、$I+3$、$\cdots$、J

という数値の列から構成されるリストを生成します。

```
Picat> X = 23..34
X = [23,24,25,26,27,28,29,30,31,32,33,34]
yes
```

$I..S..J$ という式で呼び出すと、 .. は、I から始まって、S の幅で増加していって、J を超える直前で終わる数値の列から構成されるリストを生成します。

```
Picat> X = 0..8..100
X = [0,8,16,24,32,40,48,56,64,72,80,88,96]
yes
Picat> X = 0.1..0.1..1.0
X = [0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0]
yes
```

増加の幅としてマイナスの数値を指定することによって、減少していく数値の列から構成されるリストを .. に生成させることも可能です。

```
Picat> X = 100..-8..0
X = [100,92,84,76,68,60,52,44,36,28,20,12,4]
yes
```

5.2.8　文字列

Picat では、文字列は、単一文字アトムを要素とするリストとして扱われます。単一文字アトムというのは、第 2.3.6 項で説明したように、1 個の文字から構成される非引用名、あるいは、1 個の文字または 1 個のエスケープシーケンスを一重引用符で囲んだ引用名を名前とするアトムのことです。

組み込み述語の `write/1` と `writeln/1` は、引数としてリストを受け取った場合、それをリスト式の形で出力します。たとえ、受け取った引数が単一文字アトムを要素とするリスト、つまり文字列だったとしても、それは同じです。

```
Picat> write([n, a, m, a, k, o])
[n,a,m,a,k,o]
yes
```

それに対して、`print/1` と `println/1` は、基本的にはリストをリスト式の形で出力するのですが、引数が文字列だった場合は、それを文字列として出力します。

```
Picat> print([n, a, m, a, k, o])
namako
yes
```

5.2.9　文字列リテラル

文字列を生成する方法としては、リスト式を書くという方法のほかに、「文字列リテラル」(string literal) と呼ばれるリテラルを書くという方法もあります。

文字列を二重引用符で囲んだものは、その文字列を生成する文字列リテラルになります。たとえば、`"namako"` は、`namako` という文字列を生成する文字列リテラルです。

```
Picat> write("namako")
[n,a,m,a,k,o]
yes
Picat> print("namako")
namako
yes
```

第 2.3.4 項で紹介したエスケープシーケンスは、文字列リテラルの中でも使うことができます。

```
Picat> print("umiushi\n\"\\")
umiushi
"\
yes
```

5.2.10　インデックス

リストを構成しているそれぞれの要素は、「インデックス」(index) と呼ばれる番号によって識別することができます。

インデックスは、先頭の要素が 1 で、末尾に向かって 1 ずつ増えていきます。たとえば、

```
[namako, 81, 7.63]
```

というリストの場合、`namako` のインデックスは 1 で、81 のインデックスは 2 で、7.63 のインデックスは 3 です。

5.2.11　n 次元リスト

すべての要素がリスト以外のデータであるリストは、「1 次元リスト」(one-dimensional list) と呼ばれます。それに対して、要素としてリストを含んでいるリストは、「多次元リスト」(multidimensional list) と呼ばれます。

n を 2 以上の整数とするとき、要素として $(n-1)$ 次元リストを含んでいるリストは、「n 次元リスト」(n-dimensional list) と呼ばれます。たとえば、

```
[a, b, [c, d, e], f, [g, h], i]
```

というリストは 2 次元リストです。

リストが何次元リストであるかということを、そのリストの「次元」(dimension) と呼びます。

5.2.12 添字表記

リストが持っている特定の要素を、その要素のインデックスを使って指定したいときは、「添字表記」(subscript notation) と呼ばれるものを書きます。

添字表記は、基本的には、

変数名 [式]

と書きます。「変数名」のところには、リストに束縛されている変数の名前を書いて、「式」のところには、指定したい要素のインデックスを求める式を書きます。

添字表記は、式として評価することができます。添字表記を式として評価すると、その値として、インデックスで指定されたリストの要素が得られます。

```
Picat> X = [a, b, c, d, e], Y = X[3]
X = [a,b,c,d,e]
Y = c
yes
```

多次元リストにおいて、リストの要素の要素や、リストの要素の要素の要素を指定したいときは、

変数名 [式$_1$, 式$_2$, 式$_3$, ···]

という形の添字表記を書きます。式$_1$ で要素を指定して、式$_2$ で要素の要素を指定して、式$_3$ で要素の要素の要素を指定します。

```
Picat> X = [a, b, [c, d, e], f, [g, h], i], Y = X[3, 2]
X = [a,b,[c,d,e],f,[g,h],i]
Y = d
yes
```

5.3 リストを処理する組み込み関数

5.3.1 この節について

この節では、組み込み関数のうちで、リストを処理の対象とするものを紹介したいと思います。

5.3.2 リストの連結

++ という二項演算子は、引数として与えられた二つのリストを連結した結果を返します。

```
Picat> X = [5, 8, 3, 7] ++ [2, 0, 6, 1]
X = [5,8,3,7,2,0,6,1]
yes
```

5.3.3 リストの長さ

len/1 という組み込み関数は、引数として受け取ったリストの長さを返します。

```
Picat> X = len([a, b, c, d, e, f, g])
X = 7
yes
```

length/1 という組み込み関数も、 len/1 と同じ動作をします。

5.3.4 リストの分解

head/1 という組み込み関数は、引数として受け取ったリストの頭部を返します。引数が空リストの場合はエラーになります。

```
Picat> X = head([5, 8, 3, 7])
X = 5
yes
Picat> X = head([])
*** unresolved_function_call(head([]))
```

first という組み込み関数も、 head/1 と同じ動作をします。

tail/1 という組み込み関数は、引数として受け取ったリストの尾部を返します。引数が空リストの場合はエラーになります。

```
Picat> X = tail([5, 8, 3, 7])
X = [8,3,7]
yes
Picat> X = tail([])
*** unresolved_function_call(tail([]))
```

last/1 という組み込み関数は、引数として受け取ったリストの末尾の要素を返します。引数が空リストの場合はエラーになります。

```
Picat> X = last([5, 8, 3, 7])
X = 7
yes
Picat> X = last([])
*** unresolved_function_call(last([]))
```

5.3.5　リストからの部分リストの取り出し

リストの一部分を構成しているリストは、「部分リスト」(sublist) と呼ばれます。

slice/3 という組み込み関数は、slice(L, I, J) という式で呼び出すと、I 番目から J 番目までの要素で構成される部分リストをリスト L から取り出して返します。要素の番号は、先頭の要素を 1 番目と数えます。

```
Picat> X = slice([a, b, c, d, e, f, g, h, i], 4, 7)
X = [d,e,f,g]
yes
```

slice/2 という組み込み関数は、slice(L, I) という式で呼び出すと、I 番目から末尾までの要素で構成される部分リストをリスト L から取り出して返します。

```
Picat> X = slice([a, b, c, d, e, f, g, h, i], 4)
X = [d,e,f,g,h,i]
yes
```

5.3.6　リストからの要素の削除

delete/2 という組み込み関数は、delete(L, X) という式で呼び出すと、リスト L から、それに含まれる最初の X を削除することによって得られるリストを返します。

```
Picat> X = delete([2, 3, 1, 4, 3, 5, 3, 6], 3)
X = [2,1,4,3,5,3,6]
yes
```

delete_all/2 という組み込み関数は、delete_all(L, X) という式で呼び出すと、リスト L から、それに含まれるすべての X を削除することによって得られるリストを返します。

```
Picat> X = delete_all([2, 3, 1, 4, 3, 5, 3, 6], 3)
X = [2,1,4,5,6]
yes
```

remove_dups/1 という組み込み関数は、remove_dups(L) という式で呼び出すと、リスト L の要素のうちで重複のあるものを、一つを残してすべて削除することによって得られるリストを返します。

```
Picat> X = remove_dups([d, b, c, a, b, d, b, b, d, c])
X = [d,b,c,a]
yes
```

5.3.7　リストへの要素の挿入

insert/3 という組み込み関数は、insert(L, I, E) という式で呼び出すと、リスト L の I 番目の要素の直前に E を挿入することによって得られるリストを返します。要素の番号は、先頭の要素を 1 番目と数えます。

```
Picat> X = insert([5, 8, 3, 7], 3, a)
X = [5,8,a,3,7]
yes
```

insert_all/3 は、insert_all(L_1, I, L_2) という式で呼び出すと、リスト L_1 の I 番目の要素の直前に、リスト L_2 を構成しているすべての要素を挿入することによって得られるリストを返します。

```
Picat> X = insert_all([5, 8, 3, 7], 3, [a, b, c, d])
X = [5,8,a,b,c,d,3,7]
yes
```

5.3.8 リストの平坦化

リストを要素として含むリストを、リストを要素として含まないリストに変換することを、リストを「平坦化する」(flatten) と言います。

flatten/1 という組み込み関数は、flatten(L) という式で呼び出すと、リスト L を平坦化した結果を返します。

```
Picat> X = flatten([a, [b, [c, d, e], f], [g, h], i])
X = [a,b,c,d,e,f,g,h,i]
yes
```

5.3.9 リストの生成

new_list/1 という組み込み関数は、new_list(N) という式で呼び出すと、N 個の自由変数から構成されるリストを生成して返します。

```
Picat> X = new_list(5)
X = [_8264,_826c,_8274,_827c,_8284]
yes
```

new_list/2 という組み込み関数は、new_list(N, E) という式で呼び出すと、N 個の E から構成されるリストを生成して返します。

```
Picat> X = new_list(5, namako)
X = [namako,namako,namako,namako,namako]
yes
```

5.3.10 リストの並べ替え

reverse/1 という組み込み関数は、reverse(L) という式で呼び出すと、リスト L の順序を逆にしたリストを返します。

```
Picat> X = reverse([a, b, c, d, e, f, g])
X = [g,f,e,d,c,b,a]
yes
```

データの列を一定の規則にもとづいて並べ替えることを、データの列を「ソートする」(sort) と言います。多くの場合、データの列は大きさの順序にもとづいてソートされます。小さいものから大きいものへという順序は「昇順」(ascending order) と呼ばれ、大きいものから小さいものへという順序は「降順」(descending order) と呼ばれます。

sort/1 という組み込み関数は、sort(L) という式で呼び出すと、リスト L を昇順にソートした結果を返します。

```
Picat> X = sort([7, 4, 9, 0, 8, 3, 1, 6, 5, 2])
X = [0,1,2,3,4,5,6,7,8,9]
yes
```

sort_down/1 という組み込み関数は、sort_down(L) という式で呼び出すと、リスト L を降順にソートした結果を返します。

```
Picat> X = sort_down([7, 4, 9, 0, 8, 3, 1, 6, 5, 2])
X = [9,8,7,6,5,4,3,2,1,0]
yes
```

sort_remove_dups/1 という組み込み関数は、sort_remove_dups(L) という式で呼び出すと、リスト L を昇順にソートして、その要素のうちで重複のあるものを、一つを残してすべて削除することによって得られるリストを返します。

```
Picat> X = sort_remove_dups([3, 0, 2, 0, 3, 1, 2, 0, 3])
X = [0,1,2,3]
```

```
yes
```

sort_remove_dups/1 は、与えられたリストが十分に長い場合は、remove_dups/1 よりも高速に動作することが期待できます。

sort_down_remove_dups/1 という組み込み関数は、sort_down_remove_dups(L) という式で呼び出すと、リスト L を降順にソートして、その要素のうちで重複のあるものを、一つを残してすべて削除することによって得られるリストを返します。

```
Picat> X = sort_down_remove_dups([3, 0, 2, 0, 3, 1, 2, 0, 3])
X = [3,2,1,0]
yes
```

5.3.11　リストの要素の和と積

sum/1 という組み込み関数は、sum(L) という式で呼び出すと、リスト L のすべての要素の和を返します。

```
Picat> X = sum([30000, 7000, 800, 50, 4])
X = 37854
yes
```

prod/1 という組み込み関数は、prod(L) という式で呼び出すと、リスト L のすべての要素の積を返します。

```
Picat> X = prod([2, 3, 7, 100])
X = 4200
yes
```

5.3.12　リストの要素の平均

avg/1 という組み込み関数は、avg(L) という式で呼び出すと、リスト L の要素の平均を返します。

```
Picat> X = avg([3, 2, 4, 0, 7, 6, 9, 8])
X = 4.875
yes
```

5.3.13　リストの要素の最大値と最小値

max/1 という組み込み関数は、max(L) という式で呼び出すと、リスト L の要素の最大値を返します。

```
Picat> X = max([48, 37, 29, 73, 56])
X = 73
yes
```

min/1 という組み込み関数は、min(L) という式で呼び出すと、リスト L の要素の最小値を返します。

```
Picat> X = min([48, 37, 29, 73, 56])
X = 29
yes
```

5.3.14　大文字と小文字の変換

to_lowercase/1 という組み込み関数は、to_lowercase(S) という式で呼び出すと、文字列 S に含まれているすべての英字の大文字を小文字に変換した結果を返します。

```
Picat> X = to_lowercase("NAMAKO")
X = [n,a,m,a,k,o]
yes
```

to_uppercase/1 という組み込み関数は、to_uppercase(S) という式で呼び出すと、文字列 S に含まれているすべての英字の小文字を大文字に変換した結果を返します。

```
Picat> X = to_uppercase("namako")
X = ['N','A','M','A','K','O']
yes
```

5.3.15 アトムから文字列への変換

atom_chars/1 という組み込み関数は、atom_chars(A) という式で呼び出すと、アトム A を文字列に変換した結果を返します。

```
Picat> X = atom_chars(namako)
X = [n,a,m,a,k,o]
yes
```

atom_codes/1 という組み込み関数は、atom_codes(A) という式で呼び出すと、アトム A を構成しているそれぞれの文字を UTF-8 での文字コードに変換して、それらの文字コードのリストを返します。

```
Picat> X = atom_codes(namako)
X = [110,97,109,97,107,111]
yes
```

5.3.16 文字列から文字コードへの変換

to_codes/1 という組み込み関数は、to_codes(S) という式で呼び出すと、文字列 S を構成しているそれぞれの文字を UTF-8 での文字コードに変換して、それらの文字コードのリストを返します。

```
Picat> X = to_codes("namako")
X = [110,97,109,97,107,111]
yes
```

5.3.17 数値から文字列への変換

number_chars/1 という組み込み関数は、number_chars(N) という式で呼び出すと、数値 N を、それを 10 進数であらわす文字列に変換した結果を返します。

```
Picat> X = number_chars(9999)
X = ['9','9','9','9']
yes
```

to_binary_string/1 という組み込み関数は、to_binary_string(N) という式で呼び出すと、数値 N を、それを 2 進数であらわす文字列に変換した結果を返します。

```
Picat> X = to_binary_string(15)
X = ['1','1','1','1']
yes
```

to_oct_string/1 という組み込み関数は、to_oct_string(N) という式で呼び出すと、数値 N を、それを 8 進数であらわす文字列に変換した結果を返します。

```
Picat> X = to_oct_string(4095)
X = ['7','7','7','7']
yes
```

to_hex_string/1 という組み込み関数は、to_hex_string(N) という式で呼び出すと、数値 N を、それを 16 進数であらわす文字列に変換した結果を返します。

```
Picat> X = to_hex_string(65535)
X = ['F','F','F','F']
yes
```

to_radix_string/2 という組み込み関数は、to_radix_string(N, R) という式で呼び出すと、数値 N を、それを R 進数であらわす文字列に変換した結果を返します。R は、2 から 36 までの整数でないといけません。

```
Picat> X = to_radix_string(1679615, 36)
X = ['Z','Z','Z','Z']
yes
```

5.4　リストを処理する組み込み述語

5.4.1　この節について

この節では、組み込み述語のうちで、リストを処理の対象とするものを紹介したいと思います。

5.4.2　リストかどうかの判定

list/1 という組み込み述語は、list(T) というゴールで呼び出すと、項 T がリストならば成功します。

```
Picat> list([8, 3, 1, 0, 4])
yes
Picat> list(namako)
no
Picat> list("namako")
yes
```

空リストは、アトムであって、かつリストでもありますので、list/1 は、空リストはリストだと判定します。

```
Picat> list([])
yes
```

5.4.3　文字列かどうかの判定

string/1 という組み込み述語は、string(T) というゴールで呼び出すと、項 T が文字列ならば成功します。

```
Picat> string([n, a, m, a, k, o])
yes
Picat> string([3, 8, 4, 7, 2, 0])
no
```

5.4.4　リストに含まれている要素かどうかの判定

membchk/2 という組み込み述語は、membchk(T, L) というゴールで呼び出すと、T が L に含まれている要素ならば成功します。

```
Picat> membchk(2, [6, 5, 7, 2, 8])
yes
Picat> membchk(3, [6, 5, 7, 2, 8])
no
```

5.4.5　リストからの要素の取り出し

nth/3 という組み込み述語は、nth(I, L, E) というゴールで呼び出すと、リスト L の I 番目の要素が E ならば成功します。要素の番号は、先頭の要素を 1 番目と数えます。

```
Picat> nth(3, [a, b, c, d, e], c)
yes
Picat> nth(3, [a, b, c, d, e], d)
no
```

E を変数にすると、その変数は、リスト L の I 番目の要素に束縛されます。

```
Picat> nth(3, [a, b, c, d, e], E)
E = c
yes
```

nth/3 は、非決定的な述語です。I と E をともに変数にして呼び出すと、バックトラックするたびに、I は 1 から len(L) までのそれぞれの整数に束縛されて、E は I 番目の要素に束縛されます。

```
Picat> nth(I, [a, b, c, d, e], E), write([I, E]), fail
[1,a][2,b][3,c][4,d][5,e]
no
```

5.4.6　リストの連結

append/3という組み込み述語は、append(X, Y, Z)というゴールで呼び出すと、リストXの右側にリストYを連結した結果がZならば成功します。

```
Picat> append([a, b, c], [d, e, f], [a, b, c, d, e, f])
yes
```

Zを変数にすると、その変数は、リストXの右側にリストYを連結した結果に束縛されます。

```
Picat> append([a, b, c], [d, e, f], Z)
Z = [a,b,c,d,e,f]
yes
```

append/3は、非決定的な述語です。XとYをともに変数にして呼び出すと、バックトラックするたびに、XとYは、連結するとZになるそれぞれのリストに束縛されます。

```
Picat> append(X, Y, [a, b, c, d, e]), writeln([X, Y]), fail
[[],[a,b,c,d,e]]
[[a],[b,c,d,e]]
[[a,b],[c,d,e]]
[[a,b,c],[d,e]]
[[a,b,c,d],[e]]
[[a,b,c,d,e],[]]
no
```

append/4という組み込み述語は、append(W, X, Y, Z)というゴールで呼び出すと、リストWの右側にリストXを連結して、さらにその右側にリストYを連結した結果がZならば成功します。

```
Picat> append([a, b], [c, d], [e, f], [a, b, c, d, e, f])
yes
```

Zを変数にすると、その変数は、リストWの右側にリストXを連結して、さらにその右側にリストYを連結した結果に束縛されます。

```
Picat> append([a, b], [c, d], [e, f], Z)
Z = [a,b,c,d,e,f]
yes
```

append/4は、非決定的な述語です。W、X、Yをいずれも変数にして呼び出すと、バックトラックするたびに、WとXとYは、連結するとZになるそれぞれのリストに束縛されます。

```
Picat> append(W, X, Y, [a, b, c]), writeln([W, X, Y]), fail
[[],[],[a,b,c]]
[[],[a],[b,c]]
[[],[a,b],[c]]
[[],[a,b,c],[]]
[[a],[],[b,c]]
[[a],[b],[c]]
[[a],[b,c],[]]
[[a,b],[],[c]]
[[a,b],[c],[]]
[[a,b,c],[],[]]
no
```

append/4は、append/3を使って次のように定義されていると考えることができます。

```
append(W, X, Y, Z) => append(W, X, WX), append(WX, Y, Z).
```

5.4.7 リストの選択

select/3という組み込み述語は、select(X, L, R)というゴールで呼び出すと、リストLからXを1個だけ削除した結果がRならば成功します。

```
Picat> select(3, [6, 3, 5, 3, 7, 3, 2], [6, 5, 3, 7, 3, 2])
yes
Picat> select(3, [6, 3, 5, 3, 7, 3, 2], [6, 3, 5, 7, 3, 2])
yes
Picat> select(3, [6, 3, 5, 3, 7, 3, 2], [6, 3, 5, 3, 7, 3, 2])
no
Picat> select(3, [6, 3, 5, 3, 7, 3, 2], [6, 5, 7, 3, 2])
```

```
no
```

select/3 は、非決定的な述語です。R を変数にして呼び出すと、バックトラックするたびに、R は、L から X を 1 個だけ削除した結果に束縛されます。

```
Picat> select(3, [6, 3, 5, 3, 7, 3, 2], R), writeln(R), fail
[6,5,3,7,3,2]
[6,3,5,7,3,2]
[6,3,5,3,7,2]
no
```

X と R をともに変数にして呼び出すと、バックトラックするたびに、X は L から選択されたそれぞれの要素に束縛されて、R は選択された要素を L から削除したリストに束縛されます。

```
Picat> select(I, [6, 3, 5, 3, 7], R), writeln([I, R]), fail
[6,[3,5,3,7]]
[3,[6,5,3,7]]
[5,[6,3,3,7]]
[3,[6,3,5,7]]
[7,[6,3,5,3]]
no
```

5.5　リストを処理する述語または関数の定義

5.5.1　リストのみと一致するパターン

ルールの頭部の中には、空リスト以外のリストのみと一致するパターン、というものを書くことができます。

空リスト以外のリストのみと一致するパターンは、

[パターン$_1$ | パターン$_2$]

と書きます。このパターンは、パターン$_1$ がリストの頭部と一致して、パターン$_2$ がリストの尾部と一致した場合に、その全体がリストと一致します。パターンに変数名が含まれていた場合、その変数は、それに一致した項に束縛されます。

次のプログラムは、headtail(L) というゴールで呼び出すと、L が空リストではないリストならば、その頭部と尾部のそれぞれを出力する、headtail/1 という述語を定義します。

プログラムの例　headtail.pi

```
headtail([H|T]) =>
    print('head: '), writeln(H),
    print('tail: '), writeln(T).
```

実行例

```
Picat> headtail([a, b, c, d, e])
head: a
tail: [b,c,d,e]
yes
```

5.5.2　特定の長さのリストと一致するパターン

ルールの頭部の中には、特定の長さのリストと一致するパターン、というものを書くこともできます。

特定の長さのリストと一致するパターンは、

[パターン , パターン , ···]

と書きます。この形のパターンは、その中に書かれたそれぞれのパターンがリストのそれぞれの要素に一致した場合に、その全体がリストに一致します。パターンに変数名が含まれていた場合、その変数名は、それに一致した項に束縛されます。

次のプログラムは、three(L) というゴールで呼び出すと、L が長さが 3 のリストならば、そのリストのそれぞれの要素を出力する、three/1 という述語を定義します。

プログラムの例　three.pi

```
three([X, Y, Z]) =>
    print('1: '), writeln(X),
    print('2: '), writeln(Y),
    print('3: '), writeln(Z).
```

実行例

```
Picat> three([namako, umiushi, hitode])
1: namako
2: umiushi
3: hitode
yes
```

5.5.3 リストの再帰的な処理

リストは、次のような再帰的な構造を持っています。

- 空リストはリストである。
- 何らかの項を頭部とし、リストを尾部とするものは、リストである。

したがって、リストを処理する述語や関数の多くは、空リストを基底として再帰的に定義を書くことができます。

5.5.4 再帰を使ってリストを処理する述語の定義

それでは、再帰を使うことによってリストを処理する述語を定義してみましょう。

次のプログラムは、allzero(L) というゴールで呼び出すと、L が空リストであるか、またはすべての要素が 0 であるリストならば成功する、allzero/1 という述語を定義します。

プログラムの例 allzero.pi

```
allzero([])    => true.
allzero([0|T]) => allzero(T).
```

実行例

```
Picat> allzero([])
yes
Picat> allzero([0, 0, 0, 0, 0, 0])
yes
Picat> allzero([0, 0, 0, 0, 3, 0])
no
```

5.5.5 再帰を使ってリストを処理する関数の定義

次に、再帰を使うことによってリストを処理する関数を定義してみましょう。

次のプログラムは、countzero(L) という式で呼び出すと、リスト L に要素として含まれる 0 の個数を返す、countzero/1 という関数を定義します。

プログラムの例 countzero.pi

```
countzero([])    = 0.
countzero([0|T]) = countzero(T) + 1.
countzero([_|T]) = countzero(T).
```

実行例

```
Picat> X = countzero([8, 0, 3, 7, 0, 0, 2, 0, 0, 3, 0, 4])
X = 6
yes
```

次のプログラムは、deletezero(L) という式で呼び出すと、リスト L から、それに要素として含まれているすべての 0 を削除することによってできるリストを返す、deletezero/1 という関数を定義します。

プログラムの例 deletezero.pi

```
deletezero([])    = [].
deletezero([0|T]) = deletezero(T).
```

```
deletezero([H|T]) = [H|deletezero(T)].
```

実行例

```
Picat> X = deletezero([8, 0, 3, 7, 0, 0, 2, 0, 0, 3, 0, 4])
X = [8,3,7,2,3,4]
yes
```

次のプログラムは、countdown(N) という式で呼び出すと、

$[N,\ N-1,\ N-2,\ \cdots,\ 1]$

というリストを返す、countdown/1 という関数を定義します。

プログラムの例 countdown.pi

```
countdown(0) = [].
countdown(N) = [N|countdown(N-1)], N >= 1 => true.
```

実行例

```
Picat> X = countdown(20)
X = [20,19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1]
yes
```

5.5.6 素因数分解

n が 2 以上の整数で、n の約数が 1 と n 自身のほかには存在しないとき、そのような n を「素数」(prime number) と呼びます。たとえば、2、3、5、7、11、13、17、19 などは素数です。

n を 2 以上の整数とするとき、n の約数であって、かつ素数であるものを、n の「素因数」(prime factor) と呼びます。たとえば、3 と 7 と 11 は 14553 の素因数です。

2 以上の整数 n が与えられたときに、n の素因数の列で、それらの素因数の積が n と等しくなるものを求めることを、n の「素因数分解」(prime factor decomposition) と言います。たとえば、14553 という整数は、

$3, 3, 3, 7, 7, 11$

という列に素因数分解することができます。2 以上の任意の整数は、素因数分解することが可能です。

次のプログラムは、decomp(N) という式で呼び出すと、N を素因数分解した結果のリストを返す、decomp/1 という関数を定義します。

プログラムの例 decomp.pi

```
decomp(N) = decomp(N, 2).

decomp(1, _) = [].
decomp(N, P) = [P|decomp(N//P, P)], N mod P =:= 0 => true.
decomp(N, P) = decomp(N, P+1).
```

実行例

```
Picat> X = decomp(14553)
X = [3,3,3,7,7,11]
yes
```

decomp/1 は、decomp/2 という関数を呼び出すだけの関数です。

decomp/2 は、decomp(N, P) という式で呼び出すと、P よりも小さな素因数を持たない整数 N を素因数分解した結果のリストを返す、という関数です。2 よりも小さな素数というのは存在しませんので、decomp(N, 2) という式を評価することによって、2 以上の任意の整数 N を素因数分解した結果のリストを求めることができます。

5.6 コマンドライン引数

5.6.1 main についての復習

第 4.1.5 項で説明したように、Picat の言語処理系を実行するコマンドで Picat のプログラムを実行した場合は、そのプログラムの中で定義されている、main/0 または main/1 という述語が呼び出されます。

Picat のプログラムが、

picat パス名

というコマンドで実行された場合は、main/0 が呼び出されます。それに対して、Picat のプログラムが、

picat パス名 引数 引数 ・・・

というコマンドで実行された場合は、main/1 が呼び出されます。

5.6.2 main/1 が受け取る引数

main/1 は、引数として、コマンドライン引数を受け取ります。

Picat のプログラムを、

picat パス名 $引数_1$ $引数_2$ ・・・$引数_n$

というコマンドで実行したとすると、main/1 は、

[$引数_1$, $引数_2$, ・・・, $引数_n$]

というリストを引数として受け取ります。それぞれの引数は、文字列です。

たとえば、hoge.pi というファイルに保存されている Picat のプログラムを、

```
python hoge namako umiushi kurage hitode
```

というコマンドで実行したとすると、そのプログラムの中で定義されている main/1 は、

```
["namako", "umiushi", "kurage", "hitode"]
```

というリストを引数として受け取ることになります。

5.6.3 コマンドライン引数を出力するプログラム

次のプログラムは、コマンドライン引数を標準出力に出力します。

プログラムの例 comarg.pi

```
main => writeln('Usage: picat comarg arg1 arg2 ... argn').

main(X) => comarg(1, X).

comarg(N, [])    => true.
comarg(N, [H|T]) =>
    print(N),
    print(': '),
    println(H),
    comarg(N+1, T).
```

実行例

```
$ picat comarg
'Usage: picat comarg arg1 arg2 ... argn'
$ picat comarg namako umiushi kurage hitode
1: namako
2: umiushi
3: kurage
4: hitode
```

5.7　配列

5.7.1　配列とは何か

配列 (array) は、リストと同じように、項を並べることによってできる列です。

配列を構成している個々の項は、その配列の「要素」(element) と呼ばれます。そして、配列を構成している要素の個数は、その配列の「長さ」(length) と呼ばれます。

長さが 0 の配列、つまり 0 個の要素から構成される配列は、「空配列」(empty array) と呼ばれます。

リストと同じように、配列を構成しているそれぞれの要素も、「インデックス」(index) と呼ばれる番号によって識別することができます。配列のインデックスも、先頭の要素が 1 で、末尾に向かって 1 ずつ増えていきます。

5.7.2　リストと配列の相違点

リストと配列は、どちらも、項を並べることによってできる列ですが、両者のあいだには次のような相違点があります。

- リストは、その長さが長ければ長いほど、その長さを求めるために時間がかかるが、配列は、自身の長さを常にデータとして保持しているので、長さとは無関係に一定の時間でその長さを求めることができる。
- リストの要素へのアクセスは、右にある要素ほど時間がかかるが、配列の要素は、どの要素についても一定の時間でアクセスすることができる。

5.7.3　配列式

配列は、「配列式」(array expression) と呼ばれる式を評価することによって生成することができます。

配列式というのは、0 個以上の式をコンマで区切って並べて、その全体を中括弧で囲んだもの、つまり、

{式, 式, ··· }

という形の式のことです。配列式を評価すると、その中の個々の式が評価されて、それらの式の値を、式と同じ順序で並べた配列が生成されて、その配列が、その配列式の値になります。

たとえば、

```
{namako, 81, 7.63}
```

という配列式を評価すると、その値として、`namako` というアトム、81 という整数、7.63 という浮動小数点数を、この順序で並べることによってできる配列が生成されて、その配列が、この配列式の値になります。

```
Picat> X = {namako, 81, 7.63}
X = {namako,81,7.63}
yes
```

5.7.4　n 次元配列

すべての要素が配列以外のデータである配列は、「1 次元配列」(one-dimensional array) と呼ばれます。それに対して、要素として配列を含んでいる配列は、「多次元配列」(multidimensional array) と呼ばれます。

n を 2 以上の整数とするとき、要素として $(n-1)$ 次元配列を含んでいる配列は、「n 次元配列」(n-dimensional array) と呼ばれます。たとえば、

```
{a, b, {c, d, e}, f, {g, h}, i}
```

という配列は 2 次元配列です。

配列が何次元配列であるかということを、その配列の「次元」(dimension) と呼びます。

5.7.5　配列の添字表記

配列も、リストと同じように、添字表記を書くことによって、インデックスで要素を指定することができます。配列の添字表記の書き方は、リストと同じです。

```
Picat> X = {a, b, {c, d, e}, f, {g, h}, i}, Y = X[3, 2]
X = {a,b,{c,d,e},f,{g,h},i}
Y = d
yes
```

5.7.6 配列を処理する組み込み関数

to_array/1という組み込み関数は、to_array(L)という式で呼び出すと、リスト L を配列に変換した結果を返します。

```
Picat> X = to_array([a, b, c, d, e, f, g])
X = {a,b,c,d,e,f,g}
yes
```

それとは逆に、to_list/1という組み込み関数は、to_list(A)という式で呼び出すと、配列 A をリストに変換した結果を返します。

```
Picat> X = to_list({a, b, c, d, e, f, g})
X = [a,b,c,d,e,f,g]
yes
```

new_arrayという組み込み関数は、

new_array(D_1, D_2, $\cdots$, D_n)

という式で呼び出すと、自由変数から構成される、D_1、D_2、……、D_n を長さとする n 次元配列を生成して返します。ただし、n は最大 10 までです。

```
Picat> X = new_array(3, 2)
X = {{_8050,_8054},{_8044,_8048},{_8038,_803c}}
yes
```

第 5.3 節で紹介した、リストを処理する組み込み関数のうちで、次のものは、配列を処理するために使うこともできます。

```
++        len/1       length/1    first/1        last/1
slice/3   slice/2     reverse/1   sort/1         sort_down/1
sort_remove_dups/1    sort_down_remove_dups/1    sum/1
prod/1    avg/1       max/1       min/1
```

5.7.7 配列かどうかの判定

array/1という組み込み述語は、array(T)という式で呼び出すと、T が配列ならば成功します。

```
Picat> array({8, 3, 1, 0, 4})
yes
Picat> array([8, 3, 1, 0, 4])
no
```

5.7.8 配列からの要素の取り出し

第 5.4.5 項で紹介したnth/3という組み込み述語は、リストから要素を取り出すためだけではなく、配列から要素を取り出すためにも使うこともできます。

```
Picat> nth(3, {a, b, c, d, e}, c)
yes
Picat> nth(3, {a, b, c, d, e}, d)
no
Picat> nth(3, {a, b, c, d, e}, E)
E = c
yes
Picat> nth(I, {a, b, c, d, e}, E), write([I, E]), fail
[1,a][2,b][3,c][4,d][5,e]
no
```

5.8　マップ

5.8.1　マップとは何か

Picat では、「マップ」(map) と呼ばれるデータを扱うことができます。

マップは、リストや配列と同じように、多数のデータが集まってできているデータです。リストや配列は、多数のデータが順番を伴った列を構成しているわけですが、それに対してマップは、データが順番を伴わずに集まったものです。

Picat において「マップ」と呼ばれる種類のデータは、Picat 以外のプログラミング言語においては、必ずしも「マップ」と呼ばれるとは限りません。Pica のマップに相当するデータを指示する名前としては、「連想配列」(associative array)、「辞書」(dictionary)、「ハッシュ」(hash) などがあります。

マップを構成している個々のデータは、そのマップの「要素」(element) と呼ばれます。そして、マップを構成している要素の個数は、そのマップの「大きさ」(size) と呼ばれます。

マップの要素は、「キー値ペア」(key-value pair) と呼ばれる、二つの項のペアです。キー値ペアを構成している二つの項のそれぞれは、「キー」(key) と「値」(value) と呼ばれます。

キーは、個々の要素を識別するために使われる項です。したがって、1 個のマップの中に、同じキーを持つ要素が 2 個以上存在することはできません。それに対して、値は、要素の識別には使われませんので、1 個のマップの中に、同じ値を持つ要素が 2 個以上存在することも可能です。

5.8.2　マップの生成

マップは、`new_map/1` という組み込み関数を呼び出すことによって生成することができます。

`new_map/1` に渡す引数は、

キー = 値

という形でキー値ペアを記述した項を要素とするリストです。

```
Picat> X = new_map([namako=38, umiushi=81, kurage=47])
X = (map)[namako = 38,kurage = 47,umiushi = 81]
yes
```

`new_map/0` という組み込み関数は、空のマップを返します。

```
Picat> X = new_map()
X = (map)[]
yes
```

5.8.3　マップの大きさ

マップの大きさを調べたいときは、`size/1` という組み込み関数を呼び出します。

`size/1` は、`size(`M`)` という式で呼び出すと、マップ M の大きさを返します。

```
Picat> X = new_map([a=3, b=8, c=9, d=6, e=4]), Y = size(X)
X = (map)[c = 9,b = 8,d = 6,e = 4,a = 3]
Y = 5
yes
```

5.8.4　マップかどうかの判定

項がマップかどうかを判定したいときは、`map/1` という組み込み述語を呼び出します。

`map/1` は、`map(`T`)` というゴールで呼び出すと、項 T がマップならば成功します。

```
Picat> map(new_map([a=8, b=4, c=3]))
yes
Picat> map([3, 2, 4, 5, 7])
no
```

5.8.5　マップの要素の取得

マップから要素を取得したいときは、`get/2` という組み込み関数を呼び出します。

`get/2` は、`get(`M`, `K`)` という式で呼び出すと、マップ M に含まれている要素のうちで、K をキーとするものの値を返します。K をキーとする要素が存在しない場合はエラーになります。

```
Picat> X = new_map([a=54, b=38, c=67]), Y = get(X, b)
X = (map)[b = 38,a = 54,c = 67]
Y = 38
yes
Picat> X = new_map([a=54, b=38, c=67]), Y = get(X, d)
*** error(existence_error(key,d),get)
```

get/3という組み込み関数も、get(M, K, D)という式で呼び出すと、get/2と同じように、マップMに含まれている要素のうちで、Kをキーとするものの値を返します。ただし、get/2とは違って、Kをキーとする要素が存在しない場合は、エラーにはしないでDを返します。

```
Picat> X = new_map([a=54, b=38, c=67]), Y = get(X, c, none)
X = (map)[b = 38,a = 54,c = 67]
Y = 67
yes
Picat> X = new_map([a=54, b=38, c=67]), Y = get(X, d, none)
X = (map)[b = 38,a = 54,c = 67]
Y = none
yes
```

5.8.6 マップの要素の変更

マップの要素を変更したいとき、つまりキーに対応する値を変更したいときは、put/3という組み込み述語を呼び出します。

put/3は、put(M, K, V)というゴールで呼び出すと、マップMに含まれているKをキーとする要素について、その値をVに変更して、成功します。

```
Picat> X = new_map([a=72, b=43, c=65]), put(X, b, 100)
X = (map)[b = 100,a = 72,c = 65]
yes
```

5.8.7 マップへの要素の追加

マップの要素を変更するput/3は、マップに要素を追加したいときにも使うことができます。

put(M, K, V)というゴールでput/3を呼び出したとき、もしもKをキーとする要素がマップMの中に存在しなかったならば、この述語は、Kをキー、Vを値とする新しい要素をマップに追加します。

```
Picat> X = new_map([a=88, b=37, c=54]), put(X, d, 62)
X = (map)[b = 37,d = 62,a = 88,c = 54]
yes
```

5.8.8 キーがマップに存在するかどうかの判定

特定のキーを持つ要素がマップの中に存在しているかどうかを判定したいときは、has_key/2という組み込み述語を呼び出します。

has_key/2は、has_key(M, K)というゴールで呼び出すと、Kをキーとする要素がマップMの中に存在しているならば成功します。

```
Picat> X = new_map([a=29, b=51, c=77]), has_key(X, b)
X = (map)[b = 51,a = 29,c = 77]
yes
Picat> X = new_map([a=29, b=51, c=77]), has_key(X, d)
no
```

5.8.9 マップからリストへの変換

マップをリストに変換したいときは、map_to_list/1という組み込み関数を呼び出します。

map_to_list/1は、map_to_list(M)という式で呼び出すと、マップMを構成しているそれぞれのキー値ペアを、

キー = 値

という形で記述した項を要素とするリストを返します。

```
Picat> X = new_map([a=98, b=22, c=43]), Y = map_to_list(X)
X = (map)[b = 22,a = 98,c = 43]
```

```
Y = [b = 22,a = 98,c = 43]
yes
```

マップは、キーのみのリストや、値のみのリストに変換することもできます。

keys/1 という組み込み関数は、keys(M) という式で呼び出すと、マップ M を構成しているそれぞれのキー値ペアのキーのみを要素とするリストを返します。

```
Picat> X = new_map([a=98, b=22, c=43]), Y = keys(X)
X = (map)[b = 22,a = 98,c = 43]
Y = [b,a,c]
yes
```

values/1 という組み込み関数は、values(M) という式で呼び出すと、マップ M を構成しているそれぞれのキー値ペアの値のみを要素とするリストを返します。

```
Picat> X = new_map([a=98, b=22, c=43]), Y = values(X)
X = (map)[b = 22,a = 98,c = 43]
Y = [22,98,43]
yes
```

5.9　集合

5.9.1　集合とは何か

Picat では、「集合」(set) と呼ばれるデータを扱うことができます。

集合というのは、マップの一種です。マップは、それを構成しているすべてのキー値ペアについて、その値が not_a_value というアトムである場合、「集合」と呼ばれるものになります。

5.9.2　集合の生成

集合は、new_set/1 という組み込み関数を呼び出すことによって生成することができます。

new_set/1 に渡す引数は、キーを要素とするリストです。

```
Picat> X = new_set([a, b, c, d, e])
X = (map)[b,d,e,a,c]
yes
```

5.9.3　集合を扱う組み込み述語と組み込み関数

集合はマップの一種ですので、マップを扱うすべての組み込み述語や組み込み関数は、集合を扱うこともできます。

たとえば、has_key/2 は、has_key(S, K) というゴールで呼び出すと、K をキーとする要素が集合 S の中に存在しているならば成功します。

```
Picat> X = new_set([a, b, c, d, e]), has_key(X, d)
X = (map)[b,d,e,a,c]
yes
Picat> X = new_set([a, b, c, d, e]), has_key(X, w)
no
```

5.10　ストラクチャー

5.10.1　ストラクチャー

Picat では、「ストラクチャー」(structure) と呼ばれるデータを扱うことができます。

ストラクチャーは、リストや配列と同じように、項を並べることによってできる列です。

ストラクチャーがリストや配列と違っている点は、ストラクチャーは必ず何らかのアトムをその名前として持つ、ということです。ストラクチャーの名前は、「関数子」(functor) と呼ばれます。

ストラクチャーを構成している個々の項（関数子を除く）は、そのストラクチャーの「要素」(element) と呼ばれます。そして、ストラクチャーを構成している要素の個数は、そのストラクチャーの「項数」(arity) または「長さ」(length) と呼ばれます。

リストや配列と同じように、ストラクチャーを構成しているそれぞれの要素も、「インデックス」(index) と呼ばれる番号によって識別することができます。ストラクチャーのインデックス

も、先頭の要素が 1 で、末尾に向かって 1 ずつ増えていきます。

5.10.2 項構築子

ストラクチャーは、「項構築子」(term constructor) と呼ばれる式を評価することによって生成することができます。

項構築子というのは、

$ アトム (式 , 式 , ···)

という形の式のことです。項構築子を評価すると、その中の個々の式が評価されて、それらの式の値を式と同じ順序で並べた、先頭のアトムを関数子とするストラクチャーが生成されて、そのストラクチャーが、その項構築子の値になります。

たとえば、

```
$umiushi(namako, 81, 7.63)
```

という項構築子を評価すると、その値として、namako というアトム、81 という整数、7.63 という浮動小数点数を、この順序で並べた、umiushi というアトムを関数子とするストラクチャーが生成されて、そのストラクチャーが、この項構築子の値になります。

```
Picat> X = $umiushi(namako, 81, 7.63)
X = umiushi(namako,81,7.63)
yes
```

5.10.3 演算子によるストラクチャー

演算子を関数子とするストラクチャーを作る項構築子は、演算子に特有の書き方で書くことができます。

二項演算子を関数子とするストラクチャーを作る項構築子は、

$ 式 二項演算子 式

と書くことができます。

```
Picat> X = $30+7
X = 30 + 7
yes
```

同じように、単項演算子を関数子とするストラクチャーを作る項構築子は、

$ 単項演算子 式

と書くことができます。

```
Picat> X = $-(3+5)
X = - (3 + 5)
yes
```

5.10.4 n 次元ストラクチャー

すべての要素がストラクチャー以外のデータであるストラクチャーは、「1 次元ストラクチャー」(one-dimensional structure) と呼ばれます。それに対して、要素としてストラクチャーを含んでいるストラクチャーは、「多次元ストラクチャー」(multidimensional structure) と呼ばれます。

n を 2 以上の整数とするとき、要素として $(n-1)$ 次元ストラクチャーを含んでいるストラクチャーは、「n 次元ストラクチャー」(n-dimensional structure) と呼ばれます。たとえば、

```
$a(b, c(d, e, f), g(h, i, j), k)
```

というストラクチャーは 2 次元ストラクチャーです。

ストラクチャーが何次元ストラクチャーであるかということを、そのストラクチャーの「次元」(dimension) と呼びます。

5.10.5 ストラクチャーの添字表記

ストラクチャーも、リストや配列と同じように、添字表記を書くことによって、インデックスで要素を指定することができます。ストラクチャーの添字表記の書き方は、リストや配列と同じです。

```
Picat> X = $a(b, c(d, e, f), g(h, i, j), k), Y = X[3, 2]
X = a(b,c(d,e,f),g(h,i,j),k)
Y = i
yes
```

5.10.6　ストラクチャーを生成する関数

ストラクチャーは、new_struct/2 という組み込み関数を呼び出すことによって生成することも可能です。

new_struct/2 は、new_struct(A, N) という式で呼び出すと、アトム A を関数子とする、N 個の自由変数から構成されるストラクチャーを返します。

```
Picat> X = new_struct(umiushi, 8)
X = umiushi(_96e8,_96ec,_96f0,_96f4,_96f8,_96fc,_9700,_9704)
yes
```

new_struct/2 は、2 個目の引数が整数ではなくてリストだった場合、そのリストの要素から構成されるストラクチャーを返します。

```
Picat> X = new_struct(umiushi, [namako, 81, 7.63])
X = umiushi(namako,81,7.63)
yes
```

5.10.7　ストラクチャーの項数

ストラクチャーの項数を調べたいときは、arity/1 という組み込み関数を呼び出します。

arity/1 は、arity(S) という式で呼び出すと、ストラクチャー S の項数を返します。

```
Picat> X = $umiushi(a, b, c, d, e, f), Y = arity(X)
X = umiushi(a,b,c,d,e,f)
Y = 6
yes
```

length/1 という組み込み関数も、arity/1 と同じ動作をします。

```
Picat> X = $umiushi(a, b, c, d, e, f), Y = length(X)
X = umiushi(a,b,c,d,e,f)
Y = 6
yes
```

5.10.8　関数子の取得

ストラクチャーの関数子を求めたいときは、name/1 という組み込み関数を呼び出します。

```
Picat> X = $umiushi(a, b, c, d, e, f), Y = name(X)
X = umiushi(a,b,c,d,e,f)
Y = umiushi
yes
```

5.10.9　ストラクチャーかどうかの判定

項がストラクチャーかどうかを判定したいときは、struct/1 という組み込み述語を呼び出します。

struct/1 は、struct(T) というゴールで呼び出すと、項 T がストラクチャーならば成功します。

```
Picat> struct($umiushi(a, b, c, d, e, f))
yes
Picat> struct([a, b, c, d, e, f])
no
```

5.10.10　ストラクチャーからリストへの変換

ストラクチャーをリストに変換したいときは、to_list/1 という組み込み関数を呼び出します。

to_list/1 は、to_list(S) という式で呼び出すと、ストラクチャー S を構成しているそれぞれの要素から構成されるリストを返します。

```
Picat> X = $umiushi(a, b, c, d, e, f), Y = to_list(X)
X = umiushi(a,b,c,d,e,f)
```

```
Y = [a,b,c,d,e,f]
yes
```

5.11 高階述語と高階関数

5.11.1 高階述語と高階関数の基礎

引数として述語または関数を受け取る述語は、「高階述語」(higher-order predicate) と呼ばれます。同じように、引数として述語または関数を受け取る関数は、「高階関数」(higher-order function) と呼ばれます。

Picat では、述語そのものや関数そのものを引数として述語や関数に渡すということはできません。Picat において「高階述語」と呼ばれるのは、述語または関数の名前、あるいは述語呼び出しまたは関数呼び出しを引数として受け取る述語のことです。同じように、Picat において「高階関数」と呼ばれるのは、述語または関数の名前、あるいは述語呼び出しまたは関数呼び出しを引数として受け取る関数のことです。

5.11.2 関数を呼び出す組み込み関数

apply という組み込み関数は、関数を呼び出すという動作をします。

apply を呼び出す式の書き方には、2 通りのものがあります。ひとつは、

apply(A, T_1, T_2, $\cdots$, T_n)

という書き方です。この書き方で呼び出すと、apply は、アトム A を名前とする関数を呼び出して、T_1 から T_n までの項をその関数に引数として渡して、その関数が返した戻り値を、戻り値として返します。

```
Picat> X = apply(slice, [a, b, c, d, e, f, g, h, i], 3, 7)
X = [c,d,e,f,g]
yes
Picat> X = apply('+', 3, 5)
X = 8
yes
```

apply を呼び出す式の書き方の二つ目は、apply(S) という書き方です。この書き方で呼び出すと、apply は、ストラクチャー S を関数呼び出しとして評価して、その値を戻り値として返します。

```
Picat> X = apply($slice([a, b, c, d, e, f, g, h, i], 3, 7))
X = [c,d,e,f,g]
yes
Picat> X = apply($3+5)
X = 8
yes
```

5.11.3 述語を呼び出す組み込み述語

call という組み込み述語は、述語を呼び出すという動作をします。

apply の場合と同じように、call を呼び出すゴールの書き方にも、2 通りのものがあります。ひとつは、

call(A, T_1, T_2, $\cdots$, T_n)

という書き方です。この書き方で呼び出すと、call は、アトム A を名前とする述語を呼び出して、T_1 から T_n までの項をその述語に引数として渡して、その述語が成功したならば成功します。

```
Picat> call(append, [a, b, c, d], [e, f, g, h, i], X)
X = [a,b,c,d,e,f,g,h,i]
yes
Picat> call('>', 8, 5)
yes
Picat> call('>', 5, 8)
no
```

call を呼び出すゴールの書き方の二つ目は、call(S) という書き方です。この書き方で呼び

出すと、call は、ストラクチャー S を述語呼び出しとして解決して、その述語が成功したならば成功します。

```
Picat> call($append([a, b, c, d], [e, f, g, h, i], X))
X = [a,b,c,d,e,f,g,h,i]
yes
Picat> call($8>5)
yes
Picat> call($5>8)
no
```

次のプログラムは、filter(P, L) という式で呼び出すと、リスト L の要素のうちで、アトム P を名前とする述語にそれを引数として渡した場合に成功するもののみから構成されるリストを返す、filter/2 という関数を定義します。

プログラムの例　filter.pi

```
filter(_, [])    = [].
filter(P, [H|T]) = [H|filter(P, T)], call(P, H) => true.
filter(P, [_|T]) = filter(P, T).
```

実行例

```
Picat> X = filter(int, [d, 8, 4, a, 6, g, 7, h, m, 0, 5, e])
X = [8,4,6,7,0,5]
yes
Picat> X = filter(atom, [d, 8, 4, a, 6, g, 7, h, m, 0, 5, e])
X = [d,a,g,h,m,e]
yes
```

5.11.4　リストの写像

リストを構成しているすべての要素について、それらを何らかの関数で処理して、その関数が返した戻り値から構成されるリストを作る、という処理を、「写像」(mapping) と呼びます。

map/2 と map/3 という組み込み関数は、リストの写像を実行します。

map/2 は、map(A, L) という式で呼び出すと、リスト L を構成しているすべての要素について、アトム A を名前とする関数でそれらの要素を処理して、その関数が返した戻り値から構成されるリストを返します。

```
Picat> X = map(chr, [117, 109, 105, 117, 115, 104, 105])
X = [u,m,i,u,s,h,i]
yes
```

map/3 は、map(A, L_1, L_2) という式で呼び出すと、同じインデックスを持つリスト L_1 の要素とリスト L_2 の要素について、アトム A を名前とする関数でそれらの二つの要素を処理する、ということをすべての要素について実行して、その関数が返した戻り値から構成されるリストを返します。

```
Picat> X = map('+', [3, 8, 7, 6, 4], [7, 2, 3, 4, 6])
X = [10,10,10,10,10]
yes
```

5.11.5　リストの畳み込み

引数が 2 個の関数を使って、リストの頭部と、リストの尾部を再帰的に処理した結果とを処理する、という処理を、「畳み込み」(folding) と呼びます。

reduce/2 と reduce/3 という組み込み関数は、リストの畳み込みを実行します。

reduce/2 は、reduce(A, L) という式で呼び出すと、アトム A を名前とする関数を使って、L というリストに対して畳み込みを実行します。

```
Picat> X = reduce('++', ["namako", "umiushi", "kurage"])
X = [n,a,m,a,k,o,u,m,i,u,s,h,i,k,u,r,a,g,e]
yes
```

reduce/2 は、2 個目の引数として受け取ったリストの要素が 1 個だけだった場合は、その要素を戻り値として返します。そして、要素が 0 個だった場合は、エラーになります。

```
Picat> X = reduce('++', ["namako"])
X = [n,a,m,a,k,o]
yes
Picat> X = reduce('++', [])
*** unresolved_function_call(reduce(++,[]))
```

reduce/3 は、reduce(A, L, T) という式で呼び出すと、アトム A を名前とする関数を使って、[T|L] というリストに対して畳み込みを実行します。

```
Picat> X = reduce('++', ["umiushi", "kurage"], "namako")
X = [n,a,m,a,k,o,u,m,i,u,s,h,i,k,u,r,a,g,e]
yes
```

5.11.6　畳み込みによる関数の定義

リストを処理する関数のうちのある種のものは、再帰を使う代わりに、reduce/3 を使って定義することも可能です。

第 5.5.5 項で再帰を使って定義した関数のうちの、countzero/1 と deletezero/1 は、再帰ではなく reduce/3 を使って定義することもできます。

countzero/1 は、countzero(L) という式で呼び出すと、リスト L に要素として含まれる 0 の個数を返す、という関数です。

次のプログラムは、countzero/1 を、再帰ではなく reduce/3 を使って定義します。

プログラムの例　countzero2.pi

```
countzero(N, 0) = N+1.
countzero(N, _) = N.

countzero([])  = 0.
countzero([E]) = countzero(0, E).
countzero(L)   = reduce(countzero, L, 0).
```

実行例

```
Picat> X = countzero([8, 0, 3, 7, 0, 0, 2, 0, 0, 3, 0, 4])
X = 6
yes
```

deletezero/1 は、deletezero(L) という式で呼び出すと、リスト L から、それに要素として含まれているすべての 0 を削除することによってできるリストを返す、という関数です。

次のプログラムは、deletezero/1 を、再帰ではなく reduce/3 を使って定義します。

プログラムの例　deletezero2.pi

```
deletezero(X, 0) = X.
deletezero(X, Y) = X ++ [Y].

deletezero([])  = [].
deletezero([E]) = deletezero([], E).
deletezero(L)   = reduce(deletezero, L, []).
```

実行例

```
Picat> X = deletezero([8, 0, 3, 7, 0, 0, 2, 0, 0, 3, 0, 4])
X = [8,3,7,2,3,4]
yes
```

第 6 章　代入と選択と繰り返し

6.1　代入

6.1.1　代入とは何か

Picat においては、変数をデータに束縛すると、その変数はそのデータと同一視されることになり、その変数を別のデータに束縛するということはできなくなります。

```
Picat> X = 8, X = 5
no
```

しかし、Picat では、変数が束縛されているデータを変更するということも可能です。変数が束縛されているデータを変更することを、変数にデータを「代入する」(assign) と言います（assign の名詞形は assignment）。

6.1.2　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、 := という演算子を使います。

代入演算子を使うためのゴールは、基本的には、

変数名 := 式

と書きます。このゴールを解決すると、 := の左側の変数名によって指定された変数に、 := の右側の式の値が代入されます。

```
Picat> X = 8, X := 5
X = 5
yes
```

代入演算子の左側には、リストや配列やストラクチャーの添字表記を書くこともできます。添字表記を書くと、それによって指定された要素を代入演算子の左側の式の値によって置き換えた結果が、変数に代入されます。

```
Picat> X = [a, b, c, d, e], X[3] := namako
X = [a,b,namako,d,e]
yes
Picat> X = {a, b, c, d, e}, X[3] := namako
X = {a,b,namako,d,e}
yes
Picat> X = $hoge(a, b, c, d, e), X[3] := namako
X = hoge(a,b,namako,d,e)
yes
```

6.2　if-else 文

6.2.1　選択

「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。

Picat では、選択は、複数のルールから構成される述語または関数を定義することによって記述することができます。

しかし、場合によっては、ひとつのルールの中で選択を記述したい、ということもあります。Picat では、そのような場合のために、「if-else 文」(if-else statement) と呼ばれるゴールを書くことができるようになっています。

6.2.2　if-else 文の書き方

if-else 文は、基本的には、

if 条件 then 選択肢$_1$ else 選択肢$_2$ end

と書きます。「条件」、「選択肢$_1$」、「選択肢$_2$」のところには、それぞれ、何らかのゴールを書きます。

6.2.3　if-else 文の解決

if-else 文は、その全体がひとつのゴールですから、それを解決することができます。if-else 文を解決すると、まず、「条件」のところに書かれたゴールが解決されます。そして、それが真であるか偽であるかという結果によって、選択肢$_1$ または 選択肢$_2$ のどちらかが解決されます。真だった場合は 選択肢$_1$ で、偽だった場合は 選択肢$_2$ です。

```
Picat> if true then print(namako) else print(umiushi) end
```

```
namako
yes
Picat> if false then print(namako) else print(umiushi) end
umiushi
yes
```

if-else 文は、その全体がひとつのゴールですから、それを解決すると、その結果として真（成功）または偽（失敗）という結果が得られます。if-else 文の全体の真偽値は、選択肢$_1$ または 選択肢$_2$ を解決した結果によって決まります。解決したゴールが真だったならば全体も真になり、偽だったならば全体も偽になります。

```
Picat> if true then false else true end
no
Picat> if false then false else true end
yes
```

6.2.4 if-else 文のスタイル

REPL に if-else 文を入力する場合は、その途中で改行を入力することはできませんが、ファイルに保存するプログラムの中では、if-else 文の途中に改行を挿入することができます。

if-else 文の途中に改行を挿入する場合は、通常、次のようなスタイルでそれを書きます。

```
if 条件 then
    選択肢1
else
    選択肢2
end
```

次のプログラムは、1 個の整数を引数として受け取って、それが偶数ならば even と出力して、奇数ならば edd と出力する、printevenodd/1 という述語を定義しています。

プログラムの例 printevenodd.pi

```
printevenodd(N) =>
    if N mod 2 =:= 0 then
        println(even)
    else
        println(odd)
    end.
```

実行例

```
Picat> printevenodd(6)
even
yes
Picat> printevenodd(7)
odd
yes
```

6.2.5 条件が偽の場合は何もしない if-else 文

二つのゴールのうちのどちらか一方の解決を選択するのではなくて、ひとつのゴールを解決するか、それとも解決しないか、ということを選択する if-else 文を書くこともできます。

ひとつのゴールを解決するかしないかを選択する if-else 文は、

```
if 条件 then 選択肢 end
```

と書きます。

この形の if-else 文を解決すると、まず「条件」のところに書かれたゴールが解決されて、それが真だった場合は、「選択肢」のところに書かれたゴールが解決されます。偽だった場合は、何も解決されません。

```
Picat> if true then print(namako) end
namako
yes
Picat> if false then print(namako) end
```

```
yes
```

この形の if-else 文を解決した結果は、条件が真だった場合は、解決したゴールの結果が全体の結果になります。条件が偽だった場合は、全体の結果は真になります。

```
Picat> if true then true end
yes
Picat> if true then false end
no
Picat> if false then false end
yes
```

次のプログラムは、分を単位とする時間の長さを引数として受け取って、それを「何 h 何 m」という形式の文字列に変換した結果を出力する、 printhm/1 という述語を定義しています。

プログラムの例　printhm.pi

```
printhm(M) =>
    HM = number_chars(M//60) ++ "h",
    if not M mod 60 =:= 0 then
        HM := HM ++ number_chars(M mod 60) ++ "m"
    end,
    println(HM).
```

実行例

```
Picat> printhm(160)
2h40m
yes
Picat> printhm(180)
3h
```

「何 h 」の部分の右側に「何 m 」の部分を連結するゴールは、解決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、ひとつのゴールを解決するかしないかを選択する if-else 文を使って書くことができます。

6.2.6　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-else 文の中に、

```
elseif 条件 then
    選択肢
```

という形のものをいくつか書きます。この中の elseif 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、3 個の条件による多肢選択は、

```
if 条件1 then
    選択肢1
elseif 条件2 then
    選択肢2
elseif 条件3 then
    選択肢3
else
    選択肢4
end
```

という形の if-else 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-else 文を実行すると、結果が真になる条件が見つかるまで、条件$_1$、条件$_2$、条件$_3$ という順番で条件が解決されていきます。結果が真になる条件が見つかった場合は、その条件と同じ番号の選択肢が解決

されます（その場合、それ以降の条件は解決されません）。すべての条件の結果が偽だった場合は、選択肢$_4$ が解決されます。

次のプログラムは、数値を引数として受け取って、それがプラスならばplus と出力して、そうでなくてマイナスならばminus と出力して、そうでなければzero と出力する、printsign/1 という述語を定義しています。

プログラムの例 printsign.pi

```
printsign(X) =>
    if X > 0 then
        println(plus)
    elseif X < 0 then
        println(minus)
    else
        println(zero)
    end.
```

実行例

```
Picat> printsign(5)
plus
yes
Picat> printsign(-5)
minus
yes
Picat> printsign(0)
zero
yes
```

6.3 while 文

6.3.1 繰り返し

「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。

繰り返しは、その対象となる動作の記述を、繰り返したい回数と同じ個数だけ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。

Picat では、次の 3 種類のゴールのうちのどれかを使うことによって、繰り返しを簡潔に記述することができます。

- while 文 (while statement)
- do-while 文 (do-while statement)
- foreach 文 (foreach statement)

この節では、 while 文と do-while 文について説明します。 foreach 文については、第 6.4 節で説明することにしたいと思います。

6.3.2 条件による繰り返し

while 文と do-while 文は、条件による繰り返しを記述するためのゴールです。条件による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、何らかの動作を繰り返す、というタイプの繰り返しのことです。

6.3.3 while 文の書き方

while 文は、

while (条件) 繰り返しの対象 end

と書きます。「条件」と「繰り返しの対象」のところには、それぞれ、何らかのゴールを書きます。

6.3.4 while 文の解決

while 文は、その全体がひとつのゴールですから、それを解決することができます。while 文を解決すると、次のような動作が実行されます。

(1) 「条件」のところに書かれたゴールを解決する。その結果が偽だった場合、while 文の解決は終了する。
(2) 「条件」のところに書かれたゴールの結果が真だった場合は、「繰り返しの対象」のところに書かれたゴールを実行する。
(3) (1) に戻って、ふたたび同じ動作を実行する。

6.3.5　無限ループ

while 文を使うと、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になります。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。

たとえば、次の while 文を REPL に入力すると、kurage という文字列の出力が永遠に繰り返されます。

```
while (true) print(kurage) end
```

この while 文の解決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。

次の質問を REPL に入力すると、プラスの整数が 1 から順番に出力されていきます。

```
I = 1, while (true) println(I), I := I + 1 end
```

6.3.6　無限ループではない繰り返し

最初は真になっているけれども、何回も繰り返しをすると偽になる、という条件による繰り返しは、無限ループにはならず、有限回だけ繰り返して終了します。

たとえば、次の質問を REPL に入力すると、1 から 10 までの整数が順番に出力されます。

```
I = 1, while (I <= 10) println(I), I := I + 1 end
```

6.3.7　繰り返しの対象の結果が偽だった場合

while 文の解決は、「条件」のところに書かれたゴールの結果が偽だった場合だけではなくて、「繰り返しの対象」のところに書かれたゴールの結果が偽だった場合も、そこで終了します。その場合は、while 文の全体も偽になります。

```
Picat> I = 1, while (true) println(I), I := I + 1, I <= 3 end
1
2
3
no
```

6.3.8　while 文のスタイル

REPL に while 文を入力する場合は、その途中で改行を入力することはできませんが、ファイルに保存するプログラムの中では、while 文の途中に改行を挿入することができます。

while 文の途中に改行を挿入する場合は、通常、次のようなスタイルでそれを書きます。

```
while ( 条件 )
    繰り返しの対象
end
```

6.3.9　繰り返しを使って最大公約数を求める関数

while 文を使った関数定義の例として、二つの整数の最大公約数を求める関数を定義してみましょう。

第 4.6.5 項で説明したように、二つの整数の最大公約数は、「ユークリッドの互除法」(Euclidean algorithm) と呼ばれる方法を使うことによって求めることができます。第 4.6.5 項では、ユークリッドの互除法について再帰を使って説明しましたが、この方法は、次のように、繰り返しを使うことによって説明することもできます。

ステップ 1　N と M という変数のそれぞれを、与えられた二つの整数に束縛する。

ステップ 2　M が 0 ならば計算を終了する。
ステップ 3　N を M で除算して、そのあまりを R という変数に代入する。
ステップ 4　M を N に代入する。
ステップ 5　R を M に代入する。
ステップ 6　ステップ 2 に戻る。

この計算を実行していけば、計算が終了したときの N が、最初に与えられた二つの整数の最大公約数になっています。ステップ 2 からステップ 6 までは、

M が 0 ではないあいだ、ステップ 3 からステップ 5 までを繰り返す。

ということだと考えることができますので、その部分は、while 文を書くことによって記述することができます。

次のプログラムは、2 個のプラスの整数（一方は 0 でもよい）を引数として受け取って、それらの最大公約数を求める gcm/2 という関数を、ユークリッドの互除法と繰り返しを使って定義しています。

プログラムの例　gcm2.pi

```
gcm(N, M) = N =>
    while (not M =:= 0)
        R := N mod M,
        N := M,
        M := R
    end.
```

実行例

```
Picat> X = gcm(54, 36)
X = 18
yes
```

6.3.10　繰り返しを使ってフィボナッチ数列の第 n 項を求める関数

while 文を使った関数定義の二つ目の例として、フィボナッチ数列の第 n 項を求める関数を定義してみましょう。

フィボナッチ数列については、第 4.6.6 項で、この概念は再帰的な構造を持っていると説明しましたが、再帰的な構造を持っているものを扱う関数や述語は再帰的にしか定義することができない、というわけではありません。フィボナッチ数列の第 n 項は、次のように、繰り返しを使うことによって求めることもできます。

ステップ 1　N という変数を、求めたい項の番号 (n) に束縛する。
ステップ 2　$F2$、$F1$、F、I という変数のそれぞれを、1 に束縛する。
ステップ 3　I と N が等しいならば計算を終了する。
ステップ 4　$F2$ と $F1$ を加算した結果を F に代入する。
ステップ 5　$F1$ を $F2$ に代入する。
ステップ 6　F を $F1$ に代入する。
ステップ 7　I と 1 を加算した結果を I に代入する。
ステップ 8　ステップ 3 に戻る。

この計算を実行していけば、計算が終了したときの F が、フィボナッチ数列の第 n 項になっています。ステップ 3 からステップ 8 までは、

I が N よりも小さいあいだ、ステップ 4 からステップ 8 までを繰り返す。

ということだと考えることができますので、その部分は、while 文を書くことによって記述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/1 という関数を、繰り返しを使って定義しています。

プログラムの例　fibonacci2.pi

```
fibonacci(N) = F =>
    F2 = 1,
```

```
    F1 = 1,
    F = 1,
    I = 1,
    while (I < N)
        F := F2 + F1,
        F2 := F1,
        F1 := F,
        I := I + 1
    end.
```

実行例

```
Picat> X = fibonacci(7)
X = 21
yes
Picat> X = fibonacci(200)
X = 453973694165307953197296969697410619233826
yes
```

6.3.11　do-while 文の書き方

do-while 文は、

do 繰り返しの対象 while (条件)

と書きます。「繰り返しの対象」と「条件」のところには、それぞれ、何らかのゴールを書きます。

6.3.12　do-while 文の解決

do-while 文は、その全体がひとつのゴールですから、それを解決することができます。

while 文の解決と、do-while 文の解決との相違点は、最初に解決されるのが、「条件」のところに書かれたゴールなのか、「繰り返しの対象」のところに書かれたゴールなのか、ということです。

while 文の場合、最初に解決されるのは「条件」のところに書かれたゴールです。したがって、それが最初から偽の場合は、「繰り返しの対象」のところに書かれたゴールを 1 回も解決しないで終了することになります。

それに対して、do-while 文の場合、最初に解決されるのは、「繰り返しの対象」のところに書かれたゴールです。したがって、たとえ「条件」のところに書かれたゴールが最初から偽だったとしても、「繰り返しの対象」のところに書かれたゴールが 1 回も解決されないで終了する、ということはありません。

実行例

```
Picat> while (false) print(kurage) end
yes
Picat> do print(kurage) while (false)
kurage
yes
```

while 文と同じように、do-while 文も、その解決は必ず成功しますので、その結果は常に真です。

6.3.13　do-while 文のスタイル

while 文の場合と同じように、do-while 文も、REPL にそれを入力する場合は、その途中で改行を入力することはできませんが、ファイルに保存するプログラムの中では、その途中に改行を挿入することができます。

do-while 文の途中に改行を挿入する場合は、通常、次のようなスタイルでそれを書きます。

```
do
    繰り返しの対象
while ( 条件 )
```

6.4 foreach 文

6.4.1 複合項による繰り返し

foreach 文は、 while 文や do-while 文と同じように、繰り返しを簡潔に記述するためのゴールです。

while 文または do-while 文が、条件による繰り返しを記述するためのゴールであるのに対して、 foreach 文は、複合項による繰り返しを記述するためのゴールです。複合項による繰り返しというのは、複合項を構成しているそれぞれの項について何らかの動作を実行する、というタイプの繰り返しのことです。

6.4.2 foreach 文の書き方

foreach 文は、基本的には、

foreach (イテレーター) 繰り返しの対象 end

と書きます。「繰り返しの対象」のところには、何らかのゴールを書きます。

foreach 文の中に書かれる、「イテレーター」(iterator) というのは、

パターン in 式

という形を持つ記述のことです。この中の「パターン」のところには何らかのパターンを、「式」のところには複合項を求める何らかの式を書きます。

6.4.3 foreach 文の解決

foreach 文は、その全体がひとつのゴールですから、それを解決することができます。foreach 文を解決すると、次のような動作が実行されます。

(1) 「式」のところに書かれた式を評価する。

(2) 「式」のところに書かれた式の値として得られた複合項を構成しているそれぞれの項について、先頭の項から順番に、「パターン」のところに書かれたパターンとその項とが一致するかどうかを調べて、一致したならば、「繰り返しの対象」のところに書かれたゴールを実行する、ということを繰り返す。

パターンの中に自由変数の変数名が含まれていた場合、その自由変数は、一致した項と単一化されます。

```
Picat> foreach (X in [namako, umiushi, kurage]) println(X) end
namako
umiushi
kurage
yes
```

while 文と同じように、 foreach 文も、「繰り返しの対象」のところに書かれたゴールの結果が偽だった場合、解決はそこで終了します。その場合は、 foreach 文の全体も偽になります。

```
Picat> foreach (X in [a, b, c, d]) println(X), X !== b end
a
b
no
```

6.4.4 foreach 文のスタイル

REPL に foreach 文を入力する場合は、その途中で改行を入力することはできませんが、ファイルに保存するプログラムの中では、 foreach 文の途中に改行を挿入することができます。

foreach 文の途中に改行を挿入する場合は、通常、次のようなスタイルでそれを書きます。

```
foreach ( イテレーター )
    繰り返しの対象
end
```

次のプログラムは、N をプラスの整数とするとき、divisor(N) というゴールで呼び出すと、N のすべての約数を出力する、divisor/1 という述語を定義します。

プログラムの例　divisor.pi

```
divisor(N) =>
    foreach (I in 1..N)
        if N mod I =:= 0 then
            print(I),
            print(' ')
        end
    end,
    println('').
```

実行例

```
Picat> divisor(96)
1 2 3 4 6 8 12 16 24 32 48 96
yes
```

6.4.5　マップによる繰り返し

マップも複合項の一種ですから、foreach 文を使うことによって、その要素に対する繰り返しを記述することができます。

foreach 文を使ってマップによる繰り返しを記述するときに、パターンとして変数名を書いた場合、その変数は、

キー = 値

という形のキー値ペアと単一化されます。ですから、パターンとして、

変数名$_1$ = 変数名$_2$

という形のものを書けば、変数名$_1$ とキーが単一化されて、変数名$_2$ と値が単一化されることになります。

次のプログラムは、M をマップとするとき、printmap(M) というゴールで呼び出すと、M のすべての要素を出力する、printmap/1 という述語を定義します。

プログラムの例　printmap.pi

```
printmap(M) =>
    foreach (Key=Value in M)
        print('key: '),
        print(Key),
        print(', value: '),
        println(Value)
    end
```

実行例

```
Picat> printmap(new_map([namako=38, umiushi=81, kurage=47]))
key: kurage, value: 47
key: umiushi, value: 81
key: namako, value: 38
yes
```

6.4.6　多重イテレーターを持つ foreach 文

foreach 文の丸括弧の中には、「多重イテレーター」(multiple iterators) と呼ばれる、

イテレーター , イテレーター , ･･･

というように、イテレーターをコンマで区切って並べたものを書くこともできます。

多重イテレーターを持つ foreach 文は、多重の繰り返し、つまり、繰り返しの中に繰り返しがあるような繰り返しをあらわします。先頭のイテレーターはもっとも外側の繰り返しで、右にあるイテレーターほど、内側の繰り返しをあらわします。たとえば、

```
foreach (イテレーター1, イテレーター2)
    繰り返しの対象
end
```

という foreach 文は、

```
foreach (イテレーター1)
    foreach (イテレーター2)
        繰り返しの対象
    end
end
```

という foreach 文と同じ動作をします。

次のプログラムは、$Height$ と $Width$ をプラスの整数とするとき、

rectangle($Height$, $Width$)

というゴールで呼び出すと、垂直方向の個数が $Height$ で水平方向の個数が $Width$ であるような長方形になるようにアスタリスク (*) を出力する、rectangle/2 という述語を定義します。

プログラムの例 rectangle.pi

```
rectangle(Height, Width) =>
    foreach (I in 1..Height, J in 1..Width)
        print('*'),
        if J =:= Width then println('') end
    end
```

実行例

```
Picat> rectangle(3, 7)
*******
*******
*******
yes
```

6.4.7 条件を伴うイテレーター

イテレーターは、

```
パターン in 式, 条件
```

という形で書くこともできます。この中の「条件」のところには、何らかのゴールを書きます。

この形のイテレーターを持つ foreach 文を解決すると、複合項を構成しているそれぞれの項とパターンとが一致するたびに、条件が解決されます。そして、条件の結果が真だった場合にだけ、繰り返しの対象となるゴールが解決されます。

次のプログラムは、N をプラスの整数とするとき、divisor(N) というゴールで呼び出すと、N のすべての約数を出力する、divisor/1 という述語を、条件を伴うイテレーターを使って定義します。

プログラムの例 divisor2.pi

```
divisor(N) =>
    foreach (I in 1..N, N mod I =:= 0)
        print(I),
        print(' ')
    end,
    println('').
```

実行例

```
Picat> divisor(96)
1 2 3 4 6 8 12 16 24 32 48 96
yes
```

6.4.8　エラトステネスのふるい

2 から n までの範囲にあるすべての素数を求めるための手順としては、「エラトステネスのふるい」(sieve of Eratosthenes) と呼ばれるものがよく知られています。

エラトステネスのふるいは、次のような手順です。

(1) 2 から n までのすべての整数を並べた列を作る。
(2) 2 から $\sqrt{n}$ までのそれぞれの整数について、小さいものから順番に、それを I として、次の (3) を実行する。
(3) I が列の中にあるならば、その倍数のうちで列の中にあるものをすべて取り除く。ただし、I 自身は取り除かない。

この手順が終了すると、素数だけが列の中に残ります。

次のプログラムは、N をプラスの整数とするとき、`sieve(`N`)` というゴールで呼び出すと、エラトステネスのふるいを使って、2 から N までの範囲にあるすべての素数から構成される配列を求める、`sieve/1` という関数を定義しています。

プログラムの例　`sieve.pi`

```
sieve(N) = P =>
    S = create_sequence(N),
    eratosthenes(S),
    P = sequence_to_prime_list(S).

create_sequence(N) = S =>
    S = new_array(N),
    foreach (I in 2..N)
        S[I] := true
    end.

eratosthenes(S) =>
    foreach (I in 2..to_int(sqrt(len(S))))
        if S[I] == true then
            foreach (J in I+I..I..len(S))
                S[J] := false
            end
        end
    end.

sequence_to_prime_list(S) = P =>
    P = {},
    foreach (I in 2..len(S))
        if S[I] == true then
            P := P ++ {I}
        end
    end.
```

実行例

```
Picat> X = sieve(50)
X = {2,3,5,7,11,13,17,19,23,29,31,37,41,43,47}
yes
```

6.5　内包表記

6.5.1　内包表記とは何か

Picat では、リストまたは配列を生成する方法のひとつとして、「内包表記」(comprehension) と呼ばれるものがあります。

リストを生成する内包表記は「リスト内包表記」(list comprehension) と呼ばれ、配列を生成する内包表記は「配列内包表記」(array comprehension) と呼ばれます。

内包表記は、式の一種です。それを評価すると、リストまたは配列が生成されて、そのリストまたは配列が内包表記の値になります。

6.5.2 内包表記の書き方

リスト内包表記は、基本的には、

[式 : イテレーター]

と書きます。この中の「式」のところには何らかの式を書いて、「イテレーター」のところには、foreach 文の中に書くものと同じ形のイテレーターを書きます。

リスト内包表記は式の一種ですので、評価することができます。リスト内包表記の評価というのは、次のような動作です。

(1) イテレーターの中に書かれた、複合項を求める式を評価する。
(2) 式の値として得られた複合項を構成しているそれぞれの項について、先頭の項から順番に、イテレーターの中に書かれたパターンとその項とが一致するかどうかを調べて、一致したならば、「式」のところに書かれた式を評価する、ということを繰り返す。
(3) 「式」のところに書かれた式の値を、評価の順番のとおりに並べることによってできるリストを生成して、そのリストをリスト内包表記の値にする。

```
Picat> X = [I mod 2 : I in 0..10]
X = [0,1,0,1,0,1,0,1,0,1,0]
yes
```

配列内包表記の書き方は、リスト内包表記の書き方とほとんど同じです。違うのは、角括弧ではなくて中括弧で囲むというところだけです。

```
Picat> X = {I mod 2 : I in 0..10}
X = {0,1,0,1,0,1,0,1,0,1,0}
yes
```

6.5.3 多重イテレーターを持つ内包表記

foreach 文の場合と同じように、内包表記でも、多重イテレーターを使うことができます。

```
Picat> X = [[A, I] : A in [a, b, c], I in 1..3]
X = [[a,1],[a,2],[a,3],[b,1],[b,2],[b,3],[c,1],[c,2],[c,3]]
yes
```

6.5.4 条件を伴うイテレーターを持つ内包表記

条件を伴うイテレーター、つまり、

パターン in 式, 条件

という形のイテレーターは、foreach 文の中だけではなくて、内包表記の中にも書くことができます。

条件を伴うイテレーターを内包表記の中に書いた場合、複合項を構成しているそれぞれの項とパターンとが一致するたびに、条件が解決されます。そして、条件の結果が真だった場合にだけ、コロンの左側の式が評価されて、その値から構成されるリストまたは配列が生成されます。

```
Picat> X = [I : I in 1..50, (I mod 5 =:= 0; I mod 7 =:= 0)]
X = [5,7,10,14,15,20,21,25,28,30,35,40,42,45,49,50]
```

次のプログラムは、N をプラスの整数とするとき、divisor(N) という式で評価すると、N のすべての約数から構成されるリストを返す、divisor/1 という関数を、条件を伴うイテレーターを持つリスト内包表記を使って定義します。

プログラムの例 divisor3.pi

```
divisor(N) = [I : I in 1..N, N mod I =:= 0].
```

実行例

```
Picat> X = divisor(96)
X = [1,2,3,4,6,8,12,16,24,32,48,96]
yes
```

第 7 章　例外

7.1　例外の基礎

7.1.1　例外とは何か

Picat の述語と関数は、その実行中に不都合な事象が起きた場合に、「例外」(exception) を「投げる」(throw) という動作を実行することによって、プログラムの実行を中断させることができます。

たとえば、/ または // という演算子は、除算という動作を実行したとき、右側の数値（割る数）が 0 だった場合、例外を投げます。

例外は、何らかの項です。処理系に組み込まれている述語や関数が投げる例外は、多くの場合、例外を投げるに至った原因や、例外を投げた述語や関数の名前を要素とする、error というアトムを関数子とするストラクチャーです。

たとえば、/ は、0 による除算を実行した場合、

```
error(zero_divisor,/ /2)
```

というストラクチャーを例外として投げます。

Picat の処理系は、例外が投げられてプログラムの実行が中断した場合、まず *** と出力して、それに続けて例外を出力します。

```
Picat> X = 5/0
*** error(zero_divisor,/ /2)
Picat>
```

7.1.2　例外による述語または関数の終了

述語または関数が例外を投げると、その例外は、原則的には実行中の述語と関数をすべて終了させます。

それでは、次のプログラムを使って、例外は実行中の述語を終了させるということを確認してみましょう。

プログラムの例　exception.py

```
pred1(A, B) =>
    println('beginning of pred1'),
    pred2(A, B),
    println('ending of pred1').

pred2(A, B) =>
    println('beginning of pred2'),
    pred3(A, B),
    println('ending of pred2').

pred3(A, B) =>
    println('beginning of pred3'),
    println(A/B),
    println('ending of pred3').
```

このプログラムは、pred1、pred2、pred3 という三つの述語から構成されています。pred1 は pred2 を呼び出して、pred2 は pred3 を呼び出します。pred3 は、その中で除算を実行しますので、そこで例外が投げられる可能性があります。

このプログラムの pred1 を、2 回、呼び出してみましょう。1 回目は例外が投げられない引数を、2 回目は例外が投げられる引数を与えてみます。

実行例

```
Picat> pred1(5, 3)
beginning of pred1
beginning of pred2
beginning of pred3
1.666666666666667
ending of pred3
ending of pred2
ending of pred1
```

```
yes
Picat> pred1(5, 0)
beginning of pred1
beginning of pred2
beginning of pred3
*** error(zero_divisor,/ /2)
Picat>
```

このように、例外が投げられた場合は、pred3 も pred2 も pred1 も、その時点で実行が中断されるということが分かります。

7.2 例外の捕獲

7.2.1 例外処理

第 7.1.2 項で説明したように、例外が投げられると、原則的には実行中のすべての述語と関数が終了させられるのですが、場合によっては、ただ単に終了するのではなくて、発生した例外に対応した何らかの処理を実行したいということもあります。そのような、例外が投げられた場合に実行される処理は、「例外処理」(exception processing) と呼ばれます。

7.2.2 catch

例外処理を実行するためには、投げられた例外を「捕獲する」(catch) ということが必要になります。「捕獲する」というのは、述語または関数を終了させるという例外が投げられた場合の動作を停止させるということです。

例外を捕獲するためには、catch という組み込み述語を呼び出す必要があります。

catch を呼び出すゴールは、

catch(ゴール, パターン, 例外処理)

と書きます。この中の「ゴール」のところには例外が投げられるかもしれないゴールを書いて、「パターン」のところには捕獲したいパターンと一致するパターンを書いて、「例外処理」のところには例外処理を実行するゴールを書きます。

catch を呼び出すと、まず「ゴール」のところに書かれたゴールが解決されます。その途中で例外が投げられなかった場合は、catch の動作はそれで終了します。

もしも、ゴールの解決の途中で例外が投げられたならば、その例外とパターンとが一致するかどうかを調べて、一致したならば、その例外は捕獲されて、パターンの中の変数と、それに一致した項とを単一化したのち、例外処理のゴールを解決します。

```
Picat> catch(X = 5/0, E, println(E))
error(zero_divisor,+ / 2)
E = error(zero_divisor,+ / 2)
yes
```

Picat 2.0 では、0 で除算したときの例外を捕獲すると、例外の 2 個目の引数が / /2 ではなく + / 2 になりますが、その理由は不明です。

catch は、1 個目の引数として渡されたゴールを解決している途中で例外が投げられた場合、それまでの変数の束縛を無効にします。ですから、例外処理を実行するときには、すべての変数が自由変数になっています。

```
Picat> catch((X = 3, Y = 5/0), _, X = 7)
X = 7
yes
```

7.2.3 例外を捕獲する関数の定義の例

例外を捕獲する関数を、catch を使って定義してみましょう。

次のプログラムの中で定義されている divide という関数は、1 個目の引数を 2 個目の引数で除算して、その商を戻り値として返します。ただし、

```
error(zero_divisor, _)
```

というパターンと一致する例外が投げられた場合は、それを捕獲して、dividion by zero というアトムを戻り値として返します。

プログラムの例 divide.pi

```
divide(A, B) = X =>
    catch(X = A/B,
        error(zero_divisor, _),
        X = 'division by zero').
```

実行例

```
Picat> X = divide(5, 3)
X = 1.666666666666667
yes
Picat> X = divide(5, 0)
X = 'division by zero'
yes
```

7.3 例外を投げる組み込み述語

7.3.1 throw

自身の判断で例外を投げる述語や関数を定義したいときは、throw という組み込み述語を呼び出します。

throw を呼び出すゴールは、

```
throw 式
```

と書きます。この中の「式」のところには、何らかの式を書きます。

throw を呼び出すゴールを解決すると、その中に書かれた式が評価されて、その値が例外として投げられます。

```
Picat> throw 5+3
*** 8
```

7.3.2 例外を投げる述語の定義の例

第 4.6.4 項で、階乗を求める関数の定義を紹介しましたが、そのときに紹介した定義は、もしも引数としてマイナスの整数を受け取った場合は、失敗して終了する、というものでした。

次のプログラムは、n の階乗を求める述語の定義を、引数がマイナスだった場合は、

```
error(negative_argument(n), factorial/2)
```

という例外を投げるように改良したものです。

プログラムの例 factorial2.pi

```
factorial(0, F)         => F = 1.
factorial(N, F), N >= 1 => factorial(N-1, F1), F = F1*N.
factorial(N, _)         =>
    throw $error(negative_argument(N), factorial/2).
```

実行例

```
Picat> factorial(5, X)
X = 120
yes
Picat> factorial(-5, X)
*** error(negative_argument(-5),factorial/2)
Picat> catch(factorial(-5, X), E, X = 0)
X = 0
E = error(negative_argument(-5),factorial / 2)
yes
```

参考文献

[Kjellerstrand,2014] Håkan Kjellerstrand, “My First Look At Picat as a Modeling Language for Constraint Solving and Planning,” 2014.

[Zhou,2015] Neng-Fa Zhou, Håkan Kjellerstrand, Jonathan Fruhman, *Constraint Solving and Planning with Picat*, Springer, 2015, ISBN 978-3-319-25881-2.

[Zhou,2016] Neng-Fa Zhou and Jonathan Fruhman, *A User's Guide to Picat, Version 2.0*, 2016.

索引