Gura 言語マニュアル

Updated: March 11, 2011

目次

1. この文書について 6
2. はじめに 6
3. 実行方法 7
4. スクリプトの構成要素 7
4.1. 数値リテラル 7
4.2. 文字列リテラル 7
4.3. バイナリリテラル 8
4.4. 識別子 8
4.5. リスト 8
4.6. マトリクス 9
4.7. ブロック 9
4.8. 辞書 9
4.9. Quoted 値 10
4.10. シンボル値 10
4.11. 関数 10
4.12. アトリビュート 10
4.13. 演算子 10
4.14. コメント 11
4.14.1. ラインコメントとブロックコメント 11
4.14.2. マジックコメント 11
5. クラスとインスタンス 11
5.1. メンバアクセス 11
5.2. 定義基本データ型 12
5.3. オブジェクト型 13
6. 演算子 13
6.1. 組み込み演算子 13
6.2. 論理演算について 16
6.3. 文字列フォーマット 16
6.4. 代入演算子 17
7. 関数 18
7.1. 関数の呼び出し 18
7.1.1. 構成要素 18
7.1.2. 関数インスタンス 19
7.1.3. 引数指定 19
7.1.4. 引数のリスト展開 20
7.1.5. 名前つき引数指定と引数の辞書展開 20

7.1.6. アトリビュート指定 21
7.1.7. ブロック指定 21
7.1.8. スコープ 22
7.1.9. レキシカルスコープとダイナミックスコープ 23
7.1.10. ブロック式とスコープ 25
7.2. 関数バインダ 26
7.3. 関数定義 27
7.3.1. 構成要素 27
7.3.2. 関数名 27
7.3.3. 引数定義リスト 28
7.3.4. 暗黙的マッピングと関数アトリビュート定義 28
7.3.5. ブロック定義 29
7.3.6. ブロック定義の例 29
7.3.7. 関数定義の例 31
7.3.8. 関数の戻り値 32
7.4. 関数呼び出しの連結関係 33
7.5. 名前なし関数 33
7.6. メソッド呼び出し 34
7.7. メソッド定義 34
8. 制御構文 34
8.1. 条件分岐 34
8.2. 繰り返し 35
8.2.1. repeat 関数 35
8.2.2. while 関数 35
8.2.3. for 関数 35
8.2.4. cross 関数 36
8.2.5. 繰り返しにおけるフロー制御 36
8.2.6. 繰り返し関数によるリストの生成 37
8.2.7. 繰り返し関数によるイテレータの生成 37
8.3. 例外処理 37
9. 暗黙的マッピング 38
9.1. 実装のきっかけ 38
9.2. コンセプト 39
9.3. 適用ルール 39
9.4. ケーススタディ 40
9.4.1. 演算子と暗黙的マッピング 40
9.4.2. 文字列出力との組み合わせ 40
9.4.3. ファイル入力との組み合わせ 41
9.4.4. パターンマッチングとの組み合わせ 41

10. メンバマッピング 41
10.1. ケーススタディ 42
11. ユーザ定義クラス 43
11.1. 構造体のユーザ定義 44
11.2. 型変換 44
12. モジュール 44
13. イテレータ 45
13.1. 有限イテレータと無限イテレータ 46
13.2. イテレータ操作とブロック式 46
13.3. リストの生成 46
13.4. イテレータの生成 46
13.4.1. 同じ値を指定の数だけ出力する 46
13.4.2. 乱数を指定の数だけ出力する 47
13.4.3. 指定の範囲内の数列を出力する 47
13.4.4. 範囲とサンプル数を指定して数列を出力する 47
13.5. 要素の抽出 47
13.5.1. 先頭から指定数だけ要素を抽出する 47
13.5.2. 後尾の指定数だけ要素を抽出する 48
13.5.3. 先頭から指定数だけ要素をとばす 48
13.5.4. 指定数ずつ要素を除外する 48
13.5.5. 要素から nil 値を除外する 49
13.5.6. 条件に合致する要素を抽出する 49
13.5.7. 条件に合致している間の要素を抽出する 49
13.5.8. 条件に合致してからの要素を抽出する 49
13.6. 要素順序の操作 50
13.6.1. 要素列をくりかえし巡回する 50
13.6.2. 要素列をソートする 50
13.6.3. 要素列を逆に走査する 50
13.6.4. 要素列を左右に走査する 50
13.6.5. 要素を指定した数ごとに折り返す 51
14. パス名の操作 51
14.1. Gura におけるパス名 51
14.2. ディレクトリ操作 52
14.2.1. パターン 52
14.2.2. 指定のディレクトリ内のサーチ 52
14.2.3. 再帰的なディレクトリサーチ 53
14.2.4. パターンによるサーチ 53
15. ストリーム 53
15.1. ストリームの生成 54

15.2. 標準入出力 54
15.3. テキストアクセスとバイナリアクセス 55
15.4. さまざまなストリーム読み込み 56
16. イメージ 56
16.1. ファイルからの読み込み 56
16.2. ブランクイメージを生成する 57
16.3. ファイルへの書き込み 57
16.4. イメージへの描画 58
16.5. 画面表示 58
16.6. イメージ加工 58
16.6.1. 反転 58
16.6.2. 回転 58
16.6.3. 拡大・縮小 58
16.6.4. サムネイル作成 59
16.6.5. 部分抽出 59

1. この文書について

スクリプト言語 Gura の言語仕様について説明します。内容は、Gura v0.1.0 の実装に基づきます。

2. はじめに

リストなどで表現される複数のデータに対してある処理を施し、変換した結果をリストに格納するという処理は頻繁に行われるものの一つです。たとえば、ある数列を数学的な関数にかけた結果を得てプロットしたり、データベースに格納された複数のレコードから情報を抽出して特定のフォーマットに変換したりする処理がこれに含まれます。

このような処理をするため、多くのプログラミング言語は繰り返し処理を行う制御構文を用意しています。これを使うと、リストの要素を順にとりだして処理をし、結果用のリストを生成することが可能になります。また関数型言語では、写像を行う高階関数を用意し、リスト要素に特定の関数を適用することで結果を得るアプローチがよくとられます。

いずれの方法にせよ、既存の言語において複数データを処理するには「繰り返す」という操作を明示的にプログラムすることが必要でした。しかし、たとえば一対一の写像を行う関数 f があった場合、これに n 個のデータを与えれば n 個の写像結果を求めていることは自明です。複数の要素を表すデータ構造であるリストやイテレータが引数として関数に与えられたとき、これを展開して繰り返し実行する機能をプログラム言語自体にとりいれてしまえば、データが複数になってもユーザ（プログラマ）は直接関数を呼び出すだけで望む結果が得られることになります。

このアイデアは写像すなわちマッピング処理を暗黙的に行うものなので、「暗黙的マッピング」と名づけました。Gura は、この「暗黙的マッピング」を実践するために誕生した新しいスクリプト言語です。

Gura は「暗黙的マッピング」のほかに、複数のオブジェクトに属するメンバに対して一括処理できる「メンバマッピング」や、イメージデータを統一的に扱える機構や抽象ストリーム処理、引数の型変換、モジュール拡張など、プログラミングを便利にするさまざまな工夫を取り入れています。あなたの日々の作業を助けるツールのひとつとして、Gura が役立つことを祈ってやみません。

3. 実行方法

Gura の Windows 用実行ファイルは gura.exe、Linux 用実行ファイルは gura です。Windows 用には、コマンドプロンプトを出さない実行ファイル guraw.exe も用意されています。Tcl/Tk や SDL などを使った GUI プログラムを実行するときは、こちらが便利です。

gura.exe または gura を引数をつけずに実行すると、プロンプト ">>>" が出てプログラムの入力待ちになります。この機能を、REPL (Read-eval-print loop) と呼びます。

引数にスクリプトファイルを指定すると、その内容を実行します。スクリプトファイルのサフィックスは ".az" です。

他に、以下のコマンドオプションを受け付けます。

オプション	説明
-c cmd	引数列に記述した cmd の内容をスクリプトとして実行します。
-t	スクリプトファイルの指定や-c オプションを使うと、そのスクリプト内容を実行した後プログラムを終了します。-t オプションを指定すると、これらの実行の後に REPL を起動します。
-i module[,…]	スクリプトを実行する前にインポートするモジュールを指定します。
-I dir	モジュールをサーチするディレクトリを指定します。
-C dir	スクリプトの実行前に、カレントディレクトリを指定のディレクトリに変更します。
-v	バージョン番号を表示します。

4. スクリプトの構成要素

4.1. 数値リテラル

数値には、実数と虚数の二種類があります。

実数は number 型のインスタンスで、内部表現はすべて浮動小数点数値として扱われます。実数の表記を正規表現で表すと以下のようになります。

```
-?[0-9]*(¥.[0-9]*)?([e|E][-+]?[0-9]+)?
```

虚数は complex 型のインスタンスです。実数を表す数値リテラルの直後に "j" をつけて表現します。

4.2. 文字列リテラル

文字列をシングルクオーテーション "'" またはダブルクオーテーション """ で囲むと、文字列リテラルになります。文字列リテラルは、string 型のインスタンスとして扱われ、内部表現は UTF-8 フォーマットになります。処理は常に文字単位で行われます。シングルクオーテーションとダブルクオーテーションは、内部にそれぞれダブルクオーテーション記号やシングルクオーテーション記号をエスケープすることなく含められること以外に違いはありません。

シングルクオーテーションまたはダブルクオーテーションを三つ連ねた記号で文字列を囲むと、文字列表記中に改行を含むことができます。

文字列の前に "r" を指定すると、文字列中にエスケープ記号 "¥" を含められるようになります。これは、エス

ケープ記号を頻繁に記述する正規表現パターンなどを記述する際に便利です。ただし、文字列の最後にエスケープ記号が表れる場合は、この記法が使えませんので注意してください。

文字列の前に "e" を指定すると、文字列中に "${…}" というパターンがあると、その内容をスクリプトと認識して評価し、評価結果に置き換えます。

文字列の "r" 指定は "e" 指定と併用できます。

4.3. バイナリリテラル

文字列リテラルの前に "b" を指定すると、バイナリリテラルになります。表記のルールは文字列リテラルと同じです。

バイナリリテラルは、binary 型のインスタンスとして扱われ、内部表現はバイトデータのバイナリ列になります。処理は常にバイト単位で行われます。

4.4. 識別子

識別子は、英文字・アンダースコア "_" ・ドル記号 "$" ・ アット記号 "@" または UTF-8 マルチバイト列の先頭バイトで始まり、これらの文字に加えて数字・UTF-8 マルチバイト列データが続く文字列です。UTF-8 を受け付けますので、日本語などの表記も可能です。

識別子を評価すると、現在のスコープの定義内容に置き換えられます。置き換えられる値は、その定義内容を変更しないかぎり不変です。

4.5. リスト

角括弧記号 "[" および "]" で囲んだ領域はリストになります。リストは list クラスのインスタンスです。リスト中には Gura で認識できる任意のデータを要素として記述することができます。要素間はカンマ "," で区切りますが、改行も要素間の区切りとして認識されます。これは、要素を複数の行に分けて記述する場合、行末のカンマを省略できることを意味します。

リストを評価すると、内部の要素を順に評価し、その結果を要素としてもつリストオブジェクトを返します。以下は有効なリスト表記の例です。

```
[]
[1, 2, 3, 4, 5]
["Hello", 2, 3, "World"]
[[1, 2, 3], 4, 5, [6, 7, [8, 9, 10]]]
```

リストは、角括弧のかわりに "@{" と "}" で囲んでも実現することができます。両者とも処理内容は同じです。また、要素がリストの場合、そのリストをブレース記号 "{" および "}" で囲んで表現することもできます。これらの表記を使うと、C 言語などにおける配列の初期化宣言と似た表現ができるので、両者のコード間でデータの移植を行うときなどに便利です。この表記は、実装上は "@" という名前の関数をブロック式つきで呼び出したものです。関数とブロック式については後述を参照ください。

リスト要素の中に、評価結果がイテレータになるものがあると、そのイテレータを展開したものを要素として追加します。

```
[1..10]      # [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] と等価
```

```
[1..3, 8..10]# [1, 2, 3, 8, 9, 10] と等価
```

評価結果がリストになる値の後に、角括弧で囲んだ一つ以上のインデクス数値を指定すると、要素の参照ができます。また、同じく評価結果がリストになる値の後に、角括弧で囲んだ一つ以上のインデクス数値を指定し、代入演算子 "=" に続けて代入値を指定すると、指定の要素の内容変更ができます。

4.6. マトリクス

"@@{" および "}" で囲んだ領域はマトリクスの生成になります。マトリクスは matrix クラスのインスタンスです。

マトリクスの要素は、行ごとに列要素をブレース記号 "{" および "}"、または各カッコ記号 "[" および "]" で囲って表します。要素は、評価をした後の値が matrix インスタンスに入ります。列要素の数はすべての行で同じでなければいけません。異なるものがあるとエラーになります。

以下は有効はマトリクス表記の例です。

```
@@{{2, 5, -1}, {1, 3, 1}, {3, -1, -2}}
@@{{math.cos(t), -math.sin(t)}, {math.sin(t), math.cos(t)}}
```

4.7. ブロック

ブレース記号 "{" および "}" で囲んだ領域をブロックと呼びます。リスト中には Gura で認識できる任意のデータを要素として記述することができます。要素間はカンマ "," で区切りますが、改行も要素間の区切りとして認識されます。これは、要素を複数の行に分けて記述する場合、行末のカンマを省略できることを意味します。

これまでの記述を読むとリストの説明と同じなのに気がつきますが、違いは評価時にあらわれます。ブロックを評価すると、リストと同じように内部の要素を順に評価していくのですが、一番最後に評価された要素の値がそのブロックの値になります。以下は有効なブロック表記の例です。

```
{}
{1, 2, 3, 4, 5}
{"Hello", 2, 3, "World"}
{{1, 2, 3}, 4, 5, {6, 7, {8, 9, 10}}}
{print(), x += 2}
```

ブロックは、関数を定義する際の手続きを記述したり、関数にブロック式を渡す際に使われます。

4.8. 辞書

"%{" と "}" で囲んだ領域は辞書の生成になります。辞書は dict クラスのインスタンスです。辞書定義の要素は、キーと値からなりますが、その表記方法は以下のように 3 通りあります。可読性の観点から、1 番目の表記を推奨します。

1. キーと値を辞書代入演算子 "=>" でつなげて表現します。辞書代入どおしの間は、カンマ "," または改行で区切ります。
2. キーと値を要素に持つリストまたはブロックにして表現します。リスト形式の場合、角括弧 "[" と "]" でこらの要素を囲みます。ブロックの場合はブレース記号 "{" および "}" で囲みます。
3. キーと値を一次元的に交互にならべて表記します。

辞書のキーには、数値、文字列、シンボルが使用できます。値は Gura が扱える任意のデータ型を割り当てられます。

辞書の表記は、実装上は "%" という名前の関数をブロック式つきで呼び出したものです。関数とブロック式については後述を参照ください。

評価結果が辞書になる値の後に、角括弧で囲んだ一つ以上のキーを指定すると、要素の参照ができます。また、同じく評価結果が辞書になる値の後に、角括弧で囲んだ一つ以上のキーを指定し、代入演算子 "=" に続けて代入値を指定すると、指定の要素の内容変更ができます。

4.9. Quoted 値

式の先頭にバッククオート "`" をつけると、式の評価が遅延され、Quoted 値として扱われます。Quoted 値は、expr クラスのインスタンスです。

4.10. シンボル値

識別子の先頭にバッククオート "`" をつけると、シンボル値として扱われます。シンボル値は、内部で一意な数値として扱われるので、値同士の比較が高速にできます。この性質を利用して、辞書のキーや列挙値などに使われます。

4.11. 関数

評価結果が関数インスタンスになる値の後に、括弧 "(", ")" で囲んだ引数リストをつけると関数表記として扱われます。引数は空であっても良いですが、その場合でも、いくつかの例外を除いて中身のない括弧を記述し、それが関数呼び出しであることを明示する必要があります。

クラスに属する関数を、特に「メソッド」と呼びます。

他の言語と比べてユニークな点として、Gura には制御構文やクラス定義などを表現するための特別なステートメントは存在しません。こういった処理もすべて関数として実装されています。

4.12. アトリビュート

コロン記号 ":" に識別子を続けたものをアトリビュートと呼びます。アトリビュートは、識別子や関数の引数リストの後に記述し、以下のような用途に使用します。

- 識別子に値を代入するときの型変換指定
- 関数定義の引数指定における型指定
- 関数呼び出しの引数リストの後に記述して、関数の挙動を指定
- 関数定義の引数リストの後に記述して、関数のデフォルトの挙動を指定

関数呼び出しで使われるアトリビュート指定は、関数の引数指定とよく似ています。両者の違いは、関数の引数が動的にその値を変えていくのに対して、アトリビュート指定では静的な指定になる点です。

4.13. 演算子

関数の特別な記述形式として演算子があります。演算子には、ひとつの引数のみをとる単項演算子と、二つの引数をとる二項演算子があります。

4.14. コメント

4.14.1. ラインコメントとブロックコメント

スクリプト中で、スラッシュ記号を二つつなげた記号 "//" またはシャープ記号 "#" が表れると、そこから行末までをコメントと見なします。これをラインコメントと呼びます。

スラッシュとアスタリスクをつなげた記号 "/*" からアスタリスクとスラッシュをつなげた記号 "*/" の間もコメントになります。これをブロックコメントと呼びます。ブロックコメント中は、改行を含むことができます。また、ブロックコメントの中に、他のブロックコメントをネストして記述することができます。

以下は有効なコメントの例です。

```
// line comment
# line comment again
x = 10 // line comment after some code
x = 10 # line comment after some code again
/* block comment */
/*
block comment
*/
/* /* /* nested comment */ */ */
```

4.14.2. マジックコメント

スクリプトファイルに ascii コード以外の文字を含む場合は、エンコーディング名をマジックコメントとして記述する必要があります。マジックコメントとは、スクリプトファイルの一行目または二行目に書かれるコメントで、"coding: XXXXXX" という書式を含みます。XXXXXX の部分はエンコーディング名で、例えば utf-8 や shift_jis のような文字列が入ります。

マジックコメントはラインコメントとして記述する必要があります。パーサーはまず一行目にラインコメントが記述されているかを調べマジックコメントの文字列を探します。もし一行目が shebang (UNIX のシェルスクリプトで使われる、#! で始まるスクリプト実行コマンドライン宣言) ならば、二行目にラインコメントが記述されているかを調べマジックコメントの文字列を探します。

5. クラスとインスタンス

Gura が扱うデータはすべてなんらかのクラスに属します。クラスは、基本データ型とオブジェクト型に大別されます。この区別はデータが占有するメモリ領域の管理のしかたの違いに基づきます。

クラスをもとに生成したデータをインスタンスと呼びます。インスタンスは、クラスが提供するメソッドや変数の定義を引き継ぎます。

クラス名と変数名は、別の名前空間に属します。つまり、クラス名と変数名が同じであってもかまいません。

5.1. メンバアクセス

インスタンスにドット "." をつけてメンバ変数を指定すると、変数の内容を参照できます。またメソッドを指定するとそのメソッドを実行します。メソッドの中では、自分自身を self という名前の変数で参照します。

メンバ変数を指定した後、代入演算子 "=" に続いて値を指定すると変数の内容を変更できます。また、メソッドも、通常の関数定義と同じように代入演算子 "=" でその処理内容を外部から定義することができます。
これは、既存のインスタンスに対してメソッドをあとから拡張できることを意味します。以下に例を示します。

```
str = ""
str.introduce() = { println('this string is ', self) }
str.introduce()
```

string 型のインスタンスにメソッド introduce を定義しています。この機能により、すでに存在するインスタンスの機能拡張が容易にできます。

クラスにメソッドを追加することもできます。その場合は、クラスへの参照を得るために関数 classref を使います。この関数の一般式は以下の通りです。

```
classref(type:expr):map
```

string クラスにメソッドを追加する例を以下に示します。

```
classref(`string).introduce() = { println('this string is ', self) }
str = ""
str.introduce()
```

クラスに追加したメソッドは、そのクラスのすべてのインスタンスで使えることになります。これは、メソッドを定義する前に生成したインスタンスに対しても同じです。この機能は、モジュールをインポートすることで既存クラスを拡張するときに使われます。正規表現モジュール re をインポートしたとき、string クラスに string#match などのメソッドを追加するのはその一例です。

5.2. 定義基本データ型

基本データ型はもっともプリミティブなデータ型です。基本データ型のデータは、値渡しで処理が行われます。言語に組み込まれている基本データ型には以下のものがあります。

データ型	説明
symbol	シンボル値を表すデータ型です。シンボル値とは識別子の前にバッククオート "`" をつけたものを指します。内部表現は 32bit 整数値になるので、シンボル同士の比較が高速にできます。
boolean	真偽値を表すデータ型です。真を表すboolean型の変数としてtrue，偽を表す変数としてfalseが定義されています。nil値も偽として扱われ、そのほかの値はすべて真となります。空のリストやゼロ数値も真とみなされるので注意してください。 number 型に変換すると、真は1、偽は0になります。
number	実数値を表すデータ型です
complex	複素数値を表すデータ型です

5.3. オブジェクト型

オブジェクト型のデータは、参照渡しで処理が行われます。言語に標準で組み込まれているオブジェクト型には以下のものがあります。

データ型	説明
function	関数
string	文字列
binary	バイナリデータ
list	リスト
matrix	行列
dict	辞書
stream	ストリーム
datetime	時刻
timedelta	時間差
iterator	イテレータ
expr	quoted値
environment	スコープ
error	エラー
image	画像
color	色データ
palette	パレット
codec	文字コーデック

6. 演算子

6.1. 組み込み演算子

Gura に組み込まれている演算子とその機能を以下にまとめます。

演算子	機能
+x	x が number 型、complex 型、または matrix 型のいずれかの場合、x の値自身を返します。 それ以外の型の値を渡すとエラーになります。
-x	x が number 型、complex 型、matrix 型、またはtimedelta 型のいずれかの場合、x の符号を反転した値を返します。 それ以外の型の値を渡すとエラーになります。
~x	x が number 型のとき、ビット反転した結果を number 型で返します。x は、演算に先立ち整数値に丸められます。 それ以外の型の値を渡すとエラーになります。
!x	x を真偽値とみなし、論理反転した結果を Boolean 型で返します。

x + y	以下の演算結果を返します。 number + number　和を number 型で返します。 complex + complex　和を complex 型で返します。 number + complex　和を complex 型で返します。 complex + number　和を complex 型で返します。 matrix + matrix　和を matrix 型で返します。 datetime + timedelta　和を datetime 型で返します。 timedelta + datetime　和を datetime 型で返します。 timedelta + timedelta　和を timedelta 型で返します。 string + string　結合した結果を string 型で返します。 binary + binary　結合した結果を binary 型で返します。 binary + string　結合した結果を string 型で返します。 sting + binary　結合した結果を string 型で返します。 string + any　any を文字列に変換し、結合した結果を string 型で返します。 any + string　any を文字列に変換し、結合した結果を string 型で返します。
x – y	以下の演算結果を返します。 number - number　差を number 型で返します。 complex - complex　差を complex 型で返します。 number - complex　差を complex 型で返します。 complex - number　差を complex 型で返します。 matrix - matrix　差を matrix 型で返します。 datetime - timedelta　差を datetime 型で返します。 datetime - datetime　差を timedelta 型で返します。 timedelta - timedelta　差を timedelta 型で返します。
x * y	以下の演算結果を返します。 number * number　積を number 型で返します。 complex * complex　積を complex 型で返します。 number * complex　積を complex 型で返します。 complex * number　積を complex 型で返します。 matrix * matrix　積を matrix 型で返します。 matrix * list　積を list 型で返します。 list * matrix　積を list 型で返します。 timedelta * number　積を datetime 型で返します。 number * timedelta　積を datetime 型で返します。 function * any　関数バインダになります。後述を参照ください。 string * number　number 個結合した結果を string 型で返します。 number * string　number 個結合した結果を string 型で返します。 binary * number　number 個結合した結果を binary 型で返します。 number * binary　number 個結合した結果を binary 型で返します。

x / y	以下の演算結果を返します。 number / number　除算結果を number 型で返します。 complex / complex　除算結果を complex 型で返します。 number / complex　除算結果を complex 型で返します。 complex / number　除算結果を complex 型で返します。 matrix / matrix　除算結果を matrix 型で返します。
x % y	以下の演算結果を返します。 number % number　余りを number 型で返します。 string % any　文字列フォーマット指定になります。後述を参照ください。
x ** y	以下の演算結果を返します。 number ** number　べき乗を number 型で返します。 complex ** complex　べき乗を complex 型で返します。 number ** complex　べき乗を complex 型で返します。 complex ** number　べき乗を complex 型で返します。
x == y	x と y が等しいときに true、それ以外はfalseを返します。
x != y	x と y が異なるときに true、それ以外はfalseを返します。
x > y	x が y よりも大きいときに true、それ以外は false を返します。
x < y	x が y よりも小さいときに true、それ以外は false を返します。
x >= y	x が y よりも大きいか等しいときに true、それ以外は false を返します。
x <= y	x が y よりも小さいか等しいときに true、それ以外は false を返します。
x <=> y	x が y よりも小さいときに -1、等しいときに 0、大きいときに 1 を返します。
x in y	**for 関数の引数中で使われたとき** イテレータ代入式として扱われます。詳細はfor関数の説明を参照ください。 **それ以外の場所で使われたとき** y がリストまたはイテレータの場合、x が y の要素のうちのひとつと等しいときに true 、それ以外はfalse返します。 y がそれ以外の型の場合、演算子 == と同じ結果を返します。すなわち、x と y が等しいときにtrue、それ以外はfalseを返します。
x | y	x と y が number 型のとき、ビットごとのOR演算をした結果を number 型で返します。x, y は、演算に先立ち整数値に丸められます。 x と y が boolean 型のとき、論理和を計算した結果を boolean 型で返します。すなわち、x と y が共に false ときは false、それ以外は true を返します。 それ以外の型の値を渡すとエラーになります。
x & y	x と y が number 型のとき、ビットごとのAND演算をした結果を number 型で返します。x, y は、演算に先立ち整数値に丸められます。 x と y が boolean 型のとき、論理積を計算した結果を boolean 型で返します。すなわち、x と y が共に true ときは true、それ以外は false を返します。 それ以外の型の値を渡すとエラーになります。

x ^ y	x と y が number 型のとき、ビットごとのXOR演算をした結果を number 型で返します。x, y は、演算に先立ち整数値に丸められます。 x と y が boolean 型のとき、排他論理和を計算した結果を boolean 型で返します。すなわち、x と y が同じ真偽値のときはfalse、それ以外は false を返します。 それ以外の型の値を渡すとエラーになります。
x << y	x と y がnumber 型のとき、x の値を y ビット左シフトした結果を number 型で返します。x, y は、演算に先立ち整数値に丸められます。 それ以外の型の値を渡すとエラーになります。
x >> y	x と y がnumber 型のとき、x の値を y ビット右シフトした結果を number 型で返します。x, y は、演算に先立ち整数値に丸められます。 それ以外の型の値を渡すとエラーになります。
x && y	x を偽値と判断したとき、false を結果として返します。y の評価は行いません。 x を真値と判断したとき、y を評価し、これも真値と判断すると y の値を返します。y を偽値と判断すると、false を結果として返します。
x || y	x を真値と判断したとき、x の値を返します。y の評価は行いません。 x を偽値と判断したとき、y を評価し、これを真値と判断すると y の値を返します。y も偽値と判断すると、false を結果として返します。
x = y	代入演算子です。 記号 "=" の前に演算子の記号をつけると、代入対象との演算を行った結果を定義します。例えば、"x += y" という式を評価すると、まず "x + y" の結果を求め、それを x に定義します。この形式をとる演算子には、"+="、"-="、"*="、"/="、"%="、"**="、"|="、"&="、"^="、"<<="、">>=" があります。 代入演算子の詳細については、後述の説明を参照ください。

6.2. 論理演算について

論理演算子 && や || は、左側の式の条件によって右側の式を評価するか否かが決まるので、条件分岐文として if 関数の代わりに使うことができます。

暗黙的マッピングと組み合わせて使うときは、& と | を使います。これは、list#filter メソッドなど、真偽値を要素に持つイテレータを引数にとる関数を使う際に重要になる特性です。

6.3. 文字列フォーマット

文字列とリストをパーセント記号 '%' でつなげると、文字列中のフォーマッタ指定に基づいてリストの内容を文字列に変換します。

書式の形式は %[flags][width][.precision]specifier のようになります。

[specifier] には以下のうちのひとつを指定します。

specifier	説明
d, i	10 進符号つき整数

u	10 進符符号なし整数
b	2 進数整数値
o	8 進符号なし整数
x, X	16 進符号なし整数
e, E	指数形式浮動小数点数（E は大文字で出力）
f, F	小数形式浮動小数点数（F は大文字で出力）
g, G	eまたはf形式の適した方（G は大文字で出力）
s	文字列
c	文字

[flags] には以下のうちのひとつを指定します。

flags	説明
+	プラス数値のとき、先頭に + 記号をつけます
-	左詰めで文字列を配置します
(空白)	プラス数値のとき、先頭に空白文字をつけます
#	2 進、8 進、16 進整数の変換結果に対しそれぞれ "0b", "0", "0x" を先頭につけます
0	桁数の満たない部分を 0 で埋めます

[width] には最小の文字幅を 10 進数値で指定します。文字列に変換した結果の長さがこの数値に満たないとき、残りの幅を空白文字（文字コード 32）で埋めます。長さがこの数値以上の場合は何もしません。[width] の位置に数値ではなくアスタリスク "*" を指定すると、最小の文字幅を指定する数値を引数から取得します。

[precision] は specifier によって意味が異なります。浮動小数点数に対しては、小数点以下の表示桁数の指定になります。

6.4. 代入演算子

代入演算子 "=" は二項演算子のような形式を持ちます。しかし、変数スコープの内容を変えるという副作用を持つ点で、他の演算子と異なります。

代入演算子を使って、変数、インデクス要素、関数に新しい値を定義できます。また、角括弧を使って複数の要素に代入処理をすることができます。

変数は、"symbol = value" という式を評価することで内容を変更することができます。このとき、symbol の後に型名を表すアトリビュートをつけると、値をその型に変換してから変数に代入します。例えば、"foo:string = 3" という式は、3 という number 型の値を string 型に変換してから foo という名前の変数に代入します。

インデクス要素は、"obj[index] = value" という式を評価することで内容を変更することができます。obj は、インデクスアクセスを提供する任意のインスタンスで、代表的なものとしてリストクラス list や辞書クラス dict のインスタンスがあります。角括弧 "[" および "]" の中には、インデクスになる値を指定します。

インデクスは、複数指定することができます。

インデクスにリストまたはイテレータを指定すると、それらの要素をインデクス値として扱います。

関数の一般式と手続き本体を代入演算子で結合すると、関数の定義になります。単純な例では、"func() = {…}" というような形式になります。関数の一般式と、その定義方法については後の章で説明します。

代入演算子の左側に、角括弧 "[" および "]" で囲んで代入対象（変数シンボルまたはインデクス要素）を列挙すると、各代入対象ごとに値を定義します。

定義する値がリストの場合、リスト要素に対応する位置の代入対象に値を定義します。例えば、"[a, b, c] = [1, 2, 3]" という式は、変数 a, b, c にそれぞれ 1, 2, 3 を定義します。代入対象の数がリスト要素よりも少ないと、代入対象の数だけ代入処理を行います。逆に、代入対象の数がリスト要素がよりも多いと、エラーになります。

定義する値がリスト以外の場合、角括弧内の代入対象にはすべて同じ値を定義します。例えば、"[a, b, c] = 3" という式は、変数 a, b, c にそれぞれ 3 を定義します。

7. 関数

7.1. 関数の呼び出し

7.1.1. 構成要素

関数インスタンスに、引数リストをつけて評価すると関数呼び出しになります。引数リストは、関数に渡す 0 個以上の値をカンマで区切り、括弧 "(" および ")" で囲ったものです。

関数インスタンスを得る最も一般的な方法は、関数インスタンスを割り当てた識別子を評価することです。例えば識別子 println は、文字列と改行コードを標準出力に出力する機能を持った関数インスタンスに割り当てられているので、これに引数リストをつけて以下のように評価します。

```
println("hello world")
```

関数呼出しを構成する要素は以下の通りです。

- 関数インスタンス
- 引数指定
- アトリビュート指定
- ブロック指定

Gura のライブラリリファレンスなどには、関数の呼び出し形式を表した一般式を掲載しています。一般式を見ると、その関数の名前、属しているクラス、受け取る引数の名前や型、受け付けるアトリビュート、またブロック指定の有無といった情報を得ることができます。一般式の例を以下に示します。

```
open(name:string, mode:string => "r", encoding:string => "utf-8"):map {block?}
```

例であげた関数は、名前が open で、引数として string 型の name, mode, encoding をとり、mode と encoding はデフォルト値を持ちます。また、:map というアトリビュートがデフォルトで指定されていて、ブロックをオプショナルに指定することができます。

以下、一般式の表記をもとに、関数呼び出し要素の詳細について説明します。

7.1.2. 関数インスタンス

関数呼出しができるのは識別子の評価で得られた関数インスタンスに限りません。例えば、関数インスタンスを返す関数 foo()があった場合、この関数インスタンスを以下のように直接呼び出すことができます。

```
foo()()
```

関数インスタンスがメソッドである場合は、関数インスタンスの中にレシーバインスタンスの参照が格納されます。以下の例について考察してみます。

```
f = "Hello".mid
```

`f` はメソッド `string#mid` の実行内容を含んだ関数インスタンスになりますが、レシーバインスタンスである `"Hello"` への参照も `f` の内部に格納されます。この定義を使った `f(1, 2)` という呼出は `"Hello".mid(1, 2)` と等価です。

7.1.3. 引数指定

一般式の中で、変数の名前のみが指定された引数は、任意の型の値を受け取ることを意味します。例えば、引数リスト中に単に `variable` と指定した引数があれば、その引数には数値、文字列や任意の型のインスタンス、また `nil` 値も渡すことができます。

変数の名前の後に、要素が空の角括弧 `"[]"` をついているとき、その引数はリストを受け取ります。`variable[]` と指定されている引数にリストを渡します。このような指定をした引数にはイテレータを渡すこともでき、その場合はリストからイテレータに型変換がされます。それ以外の型の値を指定すると、エラーになります。

変数が受け取るデータ型を指定した引数もあり、そのような場合は変数名の後に、コロン `':'` に続けて型指定がされます。例えば、`func(x:number, y:string)` という関数があったとき、最初の引数には `number` 型、二番目には `string` 型の値を渡します。それ以外の型の値が渡された場合は型変換を試み、それに失敗するとエラーになります。

一般式の変数リスト中、変数名の後に、`"?"` がついたものがあれば、その引数は省略が可能です。例えば、`func(x?, y?, z?)` という関数があったとき、以下のような呼び出しができます。

```
func(1, 2, 3)
func(1)
func()
```

Gura の関数の中には、可変長引数をとるものがあります。そういった関数の一般式は、可変な長さになる引数の指定には `"*"` または `"+"` という記号がつきます。

可変長引数の呼び出し形式を持つ好例は `printf` です。C 言語で有名な同名の関数に由来する、この関数の一般式は以下のようになります。

```
printf(format:string, values*):map:void
```

printf 関数を呼び出す際は、string 型の値を最初の引数として指定した後、任意の数の引数を渡すことができます。例を以下に示します。

```
printf("Hello world¥n")
printf("Current number: %d¥n", x)
printf("%d + %d = %d¥n", x, y, z)
```

記号 "*" がついた可変長引数は、0 個以上の引数を受け付けます。つまり、引数がひとつも指定されていなくてもエラーにはなりません。一方、記号 "+" がついたものは、1 個以上の引数を受付けます。これは、関数が少なくとも一個の引数を期待しており、指定がないとエラーになるという意味です。

引数指定で "=>" という記号に続いて設定値が指定されている場合、その引数はデフォルト値を持ちます。呼び出しの際、その引数の指定を省略すると、デフォルト値がかわりに使われます。例えば、func(x => "yes") という関数があり、引数指定を省略して func() のように呼び出した場合、引数 x の値は "yes" になります。

デフォルト値として表記されている内容は、関数の定義時の評価値ではなく、呼び出しごとに評価したものが関数本体に渡されます。これは、"=>" のあとに続く式が動的にその値を変える内容の場合、デフォルト値が変化するという意味になります。

引数指定で、変数名の前にバッククオートが "`" がついている場合、その引数に渡した要素は評価がされず、式のまま関数に渡されます。この機能は、if や while などのフロー制御関数で使われます。例えば、関数 while の一般式は以下のようになっています。

```
while (`cond) {block}
```

一般的に、引数に指定した要素は、まず評価がされてから関数に渡されます。しかし、上記の引数 cond に渡した式は、評価されることなく関数本体に渡されます。これを評価するのは、while 関数の内部です。この機構により、関数の形式でありながら構文のような働きをさせることが可能になります。

7.1.4. 引数のリスト展開

記号 "*" を引数の後につけると、その引数をリストとみなして、引数リストの要素に展開することができます。例えば x = [1, 2, 3] というリストがあったとき、func(x*) という呼び出しは func(1, 2, 3) と等価になります。

リスト展開は任意の数だけ指定することができ、通常の引数指定と混在することも可能です。x = [1, 2, 3]、y = [5, 6, 7] というように変数が代入されていたとき、func(x*, 4, y*) は func(1, 2, 3, 4, 5, 6, 7) という呼び出しになります。

7.1.5. 名前つき引数指定と引数の辞書展開

関数の一般式で、引数にはそれぞれシンボル名が定義づけられています。例えば、func(a, b, c) という一般式を持つ関数では、それぞれの引数のシンボル名は a、b、c となります。名前つき引数指定を使うと、これらのシンボル名を関数を呼び出しの際に明示的に指定することができます。名前つき引数指定は、引数のシンボル名と割り当てる値を辞書代入演算子 "=>" でつなげて表記します。以下の 3 つの呼び出しは等価です。

```
func(1, 2, 3)
func(a => 1, b => 2, c => 3)
func(b => 2, a => 1, c => 3)
```

名前つき引数指定で記述するシンボル名は、バッククオーテーション記号を省略できます。

名前つき引数指定は、引数の数が多かったり、それぞれの引数の意味が分かりづらいときに名前を明確に記述して可読性を高める用途に使われます。また、引数の多くがオプション指定が可能になっており、特定の引数のみ値を設定するときなどにも便利です。

記号 "%" を引数の後につけると、その引数を辞書とみなして、引数リスト中のキーワード引数要素に展開することができます。例えば `x = %{`foo => 3, `bar => 4}` という辞書があったとき、`func(x%)` という呼び出しは `func(foo => 3, bar => 4)` と等価になります。

辞書展開は任意の数だけ指定することができ、通常の引数指定と混在することも可能です。`x = %{`foo => 1, `bar => 2}`、`y = %{`hoge => 5}` というように変数が代入されていたとき、`func(x%, 4, y%)` は `func(foo => 1, bar => 2, 4, hoge => 5)` という呼び出しになります。

7.1.6. アトリビュート指定

引数リストの後に、コロン記号 ":" に続けてアトリビュートを指定することができます。アトリビュートによって、関数のふるまいをカスタマイズできます。

各関数が独自に提供するアトリビュート指定もあります。そのようなアトリビュートは、一般式で、コロン ":" の後に続いて角括弧 "[" および "]" に囲まれたシンボルのリストで表されます。例として、任意の値を `number` 型に変換する関数 `tonumber` の一般式を以下に示します。

```
tonumber(value):map:[nil,zero,raise,strict]
```

この関数は呼び出しの際、`:nil` や `:zero` といったアトリビュートを受け取り、それらの指定に応じた動作をします。

7.1.7. ブロック指定

引数リストとアトリビュートの後に、ブレース記号 "{" および "}" で囲まれた要素列をとる関数があります。この要素列をブロックと呼びます。ブロック式をとる関数の一般式は、引数宣言の後に `"{block}"` または `"{block?}"` のように表記されます。前者が指定されている場合、その関数は必ずブロックを指定する必要があります。後者の場合、呼び出し時のブロック指定はオプショナルになります。

ブロック内の要素をどのように評価するかは関数によって異なります。代表的なものとして、以下の評価方法があります。

- 手続きとみなして、連続して評価する。
- データ列とみなして、評価した結果をコンテナに蓄える

ブレース記号の直後に二つのバー記号 "|" で囲んだ引数列を記述すると、ブロック中で引数を受け取ることができます。これをブロック引数とよびます。

ブロック引数で渡される引数の数やデータ型は、ブロックを評価する関数によって異なります。例えば、繰り返

し処理をする repeat 関数はブロックの内容を指定回数だけ繰り返し評価しますが、このとき `|idx:number|` というブロック引数の形式で、0 から始まるループの回数をブロックに渡します。また、ファイルを行単位で読み込む `readlines` 関数にブロックをつけると一行ごとにブロックを評価しますが、このとき渡すブロック引数の形式は`|line:string, idx:number|` となり、`line` に一行分の文字列、`idx` に 0 から始まるインデクス番号を代入します。それぞれの関数呼び出しの例を以下に示します。

```
repeat (10) {|n| println(n)}
readlines('hoge.txt') {|line, idx| print(idx, ' ', line)}
```

ブロック引数の記述は、関数定義の引数リストの記述と同じです。引数の型名をアトリビュートで指定するとその型に変換して変数に代入しますし、可変長引数指定 `"*"` や辞書指定 `"%"` も使えます。ただ一点異なるのは、関数定義の引数リストはリストに宣言された分だけ引数を渡さないとエラーになるのに対し、ブロック引数は宣言がなければそのまま無視されるという点です。必要のない引数は省けますし、引数を受け取らなくてよい場合はブロック引数の記述そのものを省略できます。ただしそれとは逆に、関数が提供する引数の数よりも多くブロック引数を記述すると、エラーになります。

引数をひとつも指定しないで、ブロック式のみ指定して関数を呼び出す場合、引数リストの記述を省略することができます。例を示します。関数 `repeat` は、引数に繰り返し回数を指定しますが、これを省略すると無限ループになります。このとき、`repeat () { some_process()}` と記述するかわりに、空の引数リストを省略して `repeat {some_process()}` と記述することができます。
ブロック式をつけて関数を呼び出すとき、

ブロック式にはブロックの手続き本体と引数情報などが含まれていますが、これらの情報をそのまま他の関数に渡したいことがあります。ブロック式を含む `expr` 型の変数を `"{|"` および `"|}"` ではさみ以下のように記述することでブロックの内容を関数に渡すことができます。

```
block = `{|x| println(x)}
repeat(10) {|block|}
```

7.1.8. スコープ

変数や関数を定義する空間を複数持ち、それらへの参照を限定する仕組みをスコープと呼びます。スコープは、構造化プログラミングをするときに必須の概念で、適切に扱うと効率的なプログラム開発ができるようになります。C や Java など静的に変数に型つけをする言語では、変数宣言をしたコード上の位置がスコープ空間になります。一方、Gura は宣言なしで変数を使うことができるスクリプト言語なので、スコープ空間の生成の仕方がそれらと異なります。

Gura では、「環境」と呼ぶ構造によってスコープを実現されています。環境は、「フレーム」と呼ばれる層を積み重ねたフレームスタックを内部に持ちます。フレームは、フレームの性質を定義する属性と、変数、関数の実体や型名とシンボル値とを結びつける辞書を持っています。関数呼び出しをすると、新たな環境を生成してそれまで実行していた環境のフレームの参照を引き継いでフレームスタックを作り、その上に新しいフレームを積み重ねます。プログラムの中で変数や関数の参照を行うと、そのときに属している環境のフレームスタックを順に探索していきます。フレームスタックの最上位に配置したフレームの属性によって、このときの探索ルールが変わります。

スクリプトを実行すると、一つのフレームを持った環境が用意されます。このときの環境をルート環境、中に用意したフレームをルートフレームと呼びます。ルートフレーム内に定義した変数や関数は、そのスクリプト内の任意の位置から参照が可能です。

関数を呼び出すと、環境を用意して新たなフレームを一つ積み重ねます。このフレームを関数呼出フレームと呼びます。関数に渡した引数の内容は関数呼出フレーム内に定義されます。また、関数内部で評価した代入操作の結果もこのフレームに反映します。

関数呼出のたびに関数呼出フレームを積み重ねるので、関数はそれぞれ独立したフレームを持つことになります。変数や関数の代入操作は、この独立したフレームに対して行われ、外部に影響を与えることはありません。変数や関数の参照は、積み重ねたフレームを順に探索していきます。つまり、初めに独自のフレーム内を探索し、そこで見つからなければ一つ下のフレームという具合です。

あるシンボルが外部のフレームの定義内容を指している場合、そのシンボルを関数内部から一度でも参照すると、その後の同じシンボルへの代入は外部定義へのアクセスになります。以下の例を考えて見ます。

```
x = 3
func() = { x = x + 1 }
func()
```

関数 func を呼び出すと、まず x + 1 を評価するためにシンボル x の参照を行いますが、このとき参照されるのは関数外部のフレームで定義されている変数 x です。関数 func はシンボル x への代入をこの変数に割り付けますので、評価結果の x への代入は同じ変数へのアクセスになります。

参照をしないで外部への代入処理をするには、extern 関数を使うか、アクセスする変数に :extern アトリビュートをつけます。

extern 関数の一般式は以下の通りです。

```
extern(`syms+)
```

引数 syms には、アクセスする変数のシンボル名を列挙します。この関数を実行すると、syms で指定されたシンボル名を現在の環境のフレームスタックから探索し、見つかったものを現在の環境で書き込みできるよう設定します。指定のシンボルが見つからないとエラーになります。

シンボルに代入するときにアトリビュート:extern をつけると、そのシンボルに対して extern 関数と同じ処理してから代入を行います。書式は以下のとおりです。

```
symbol:extern = value
```

一度参照がされていれば、extern 関数や:extern アトリビュートを使う必要はないのですが、そのような場合でも明示することによってスコープの範囲を明確にすることができます。

7.1.9. レキシカルスコープとダイナミックスコープ

関数の外部参照のスコープは、プログラム中における関数の記述位置を基点にした、いわゆるレキシカルスコープになります。これにより、プログラムの見た目がそのままスコープの内外関係になるので、処理内容を把握するのが容易になります。以下のプログラムで、関数 f が表示する x は、呼出元である関数 caller 内部の x ではなく、プログラムの「見た目」どおりの「外側」にある x です。この結果は "root" を表示します。

```
x = 'root'
f() = println(x)
caller() = {
    x = 'caller'
    f()
}
g()
```

上記のレキシカルスコープが関数のデフォルトのふるまいになりますが、関数を定義するときにアトリビュート :dynamic_scope をつけると、その関数はダイナミックスコープで動作するようになります。以下の例は "caller" を表示します。

```
x = 'root'
f():dynamic_scope = println(x)
caller() = {
    x = 'caller'
    f()
}
g()
```

ダイナミックスコープはどのような場面で役立つのでしょうか。想定されるもののひとつは、引数に式を渡したときの評価です。

例として、式を引数に受け取り、その評価結果を表示する tester という関数の定義を考察します。以下のようなコードを書いたとしましょう。

```
tester(test:expr) = printf("result .. %s¥n", eval(test))
x = 1
tester(`(x + 2))
```

この場合、関数 tester は x + 2 という式を引数で受け取り、これを関数 eval で評価します。関数 eval はレキシカルスコープのルールに基づいて変数 x を参照し、結果を得ることができます。

しかし、関数 tester を以下のように呼出したらどうなるでしょうか。

```
tester(test:expr) = printf("result .. %s¥n", eval(test))
hoge() = {
    x = 1
    tester(`(x + 2))
}
hoge()
```

結論から言うと、これはエラーになります。変数 x は関数 hoge のローカル変数なので、関数 tester のレキシカルスコープの範囲にないからです。これを以下のようにダイナミックスコープに切り替えると、関数 tester の「外側」は呼び出し元である関数 hoge の環境になるので、期待どおりの結果を得られます。

```
tester(test:expr):dynamic_scope = printf("result .. %s¥n", eval(test))
hoge() = {
    x = 1
```

```
    tester(`(x + 2))
}
hoge()
```

一般的な用途では、関数呼出でダイナミックスコープを使うことはごくまれです。式を関数に渡す処理が必要になったとき、この機能を思い出してください。

7.1.10. ブロック式とスコープ

Gura は関数呼び出しの際にブロック式を渡すことができます。ブロック式はそれを評価する関数の中で関数インスタンスとして扱われますが、これをブロック関数と呼びます。ブロック関数内のスコープの性質は通常の関数とは少々異なります。以下、例をあげて考察していきます。

```
func() {block} = {
    block()
}
```

関数 `func` はブロック式をとります。ブロック式の内容は関数インスタンスとして変数 `block` に代入されるので、それを内部で呼び出しています。

ここで、以下のような処理を考えてみます。最後の `println` で表示される内容は 1 でしょうか、それとも 2 でしょうか。

```
x = 1
func() { x = 2 }
println(x)
```

Gura でこれを実行すると、2 が表示されます。上記の `func` のようにブロック式をとる関数は制御構文のように使われることが多く、実際 `if` や `while` などの関数はまさにその用途のために存在します。この観点からすると、ブロックの中に記述した `x = 2` という式は呼び出しもとのスコープに対する処理とするのが自然な発想といえます。

しかし、ブロックの内容は関数インスタンスとなって `func` に渡され、関数呼出で評価がされています。これを通常の関数呼出フレームで評価してしまうと、その内部における代入処理は外部のフレームに影響しません。

これを解決するため、ブロック関数の呼び出し時は通常の関数呼出フレームではなく「ブロック関数呼出フレーム」を使います。このフレームがスタックフレームの最上位に積まれていると、代入処理を評価したときにこのフレームではなくその下のフレームに対して処理を行うようになります。

場合によっては、ブロック式の中で有効な変数を使いたいことがあります。そのような時は、代入する識別子にアトリビュート `:local` をつけ、ローカル変数として宣言します。これは、アトリビュート `:extern` とは逆に、変数の代入操作を最上位のフレームに限定するものです。以下の例では、ブロック内の変数 `x` がローカル変数になり、2 を代入する操作はこのローカル変数に対して行われるので、1 を表示します。

```
x = 1
func() { x:local = 2 }
println(x)
```

一度ローカル変数として宣言されると、以降はアトリビュート`:local` をつけなくても同じ変数に対する操作に

なります。

ローカル変数は local 関数を使っても宣言することができます。local 関数の一般式は以下の通りです。

```
local(`syms+)
```

引数 syms には、ローカル変数として宣言する変数のシンボル名を列挙します。

7.2. 関数バインダ

関数インスタンスとリストを演算子 "*" でつなげると、リストの内容を引数リストにして関数を実行することができます。このときの演算子 "*" のふるまいを関数バインダと呼びます。例えば、func * [1, 2, 3] という式は func(1, 2, 3) という呼び出しと同じです。

ところで、関数バインダと同じ効果は、引数リスト中で "*" を使ったリスト展開でも得られます。上の例は func([1, 2, 3]*) と評価しても結果は同じです。しかし、関数バインダを使うと「引数となるリスト」のリストやイテレータをとって、複数の関数評価ができるようになります。例えば、x = [[1, 2, 3], [4, 5, 6], [7, 8, 9]] というリストがあったとします。これに対して func * x と評価すると、各要素を引数リストとみなして func(1, 2, 3)、func(4, 5, 6)、func(7, 8, 9) という呼び出しになります。右辺の内容として、リストのかわりにイテレータを渡すこともできます。

この機能が役立つケースのひとつは、CSV やデータベースアクセスで得られた結果を構造体に収める処理です。CSV を例題にとりあげて考察してみます。CSV ファイルは複数の文字列をカンマで区切って一行ずつ配置したテキストフォーマットで、列ごとに意味を持たせています。例えば、一列目に名前、二列目に性別、三列目に年齢を格納した以下のような CSV ファイル people.csv を考えてみます。

```
"Honma Chise","female",46
"Kawahata Nana","female",47
"Kikuchi Takao","male",35
"Iwai Michiko","female",36
"Kasai Satoshi","male",24
```

Gura では csv モジュールの関数 csv.read を使って複数の文字列を要素に持つリストを、一行ごとに生成するイテレータを得ることができます。上のファイルを使い、csv.read("people.csv"):iter を実行すると、以下のような要素を生成するイテレータを返します。

```
["Honma Chise", "female", "46"]
["Kawahata Nana", "female", "47"]
["Kikuchi Takao", "male", "35"]
["Iwai Michiko", "female", "36"]
["Kasai Satoshi", "male", "24"]
```

リストのままだと要素のアクセスがしづらいので、構造体を使うことを考えます。上のデータ構造を表現する構造体は、struct 関数を使って以下のように作ることができます。

```
Person = struct(name:string, gender:string, age:number)
```

Person は構造体を生成する関数インスタンスです。person = Person("Honma Chise", "female", 46) のように評価すると、構造体インスタンス person を作り、person.name に"Honma Chise"、

person.gender に"female"、person.age に 46 が入ります。この関数インスタンスと、前述の CSV アクセス関数を関数バインダで組み合わせると、CSV ファイルを読み込んで構造体に格納する処理は以下のように記述できます。

```
Person = struct(name:string, gender:string, age:number)
people = Person * csv.read("people.csv"):iter
```

peopleはPerson構造体インスタンスを要素に持つイテレータになります。後述するメンバマッピングを使うと、各フィールドは people:*name, people:*gender, people:*age というようにアクセスできます。

7.3. 関数定義

7.3.1. 構成要素

関数の一般式を記述して、代入演算子 "=" とそれに続く関数本体の式を記述すると、関数インスタンスを生成して識別子に関連付けます。

最も簡単な例として、引数をひとつもとらない関数 hoge の定義を考えます。この場合の一般式は hoge()となるので、関数定義は以下のように書けます。

```
hoge() = println('Hello world')
```

関数本体が複数の式から成る場合、式をカンマ "," で区切って列挙したものをブレース記号 "{" および "}" で囲んだブロック式で表記します。以下に例を示します。

```
hoge() = { println('first line'), println('second line') }
```

ブロック式の内容を複数の行に分けて記述することもできます。その際、行末はカンマと同じ意味を持つので、行ごとにカンマを書く必要はありません。上の例は以下のように書くことができます

```
func() = {
    println('first line')
    println('second line')
}
```

関数の一般式は以下の要素で構成されます。

- 関数名
- 引数定義リスト
- アトリビュート定義
- ブロック定義

以下、関数定義の一般式で指定される要素の詳細について説明します。

7.3.2. 関数名

関数定義で指定する関数名は、識別子として認識できる任意のシンボル名です。これは、変数名として扱えるものと同じです。

7.3.3. 引数定義リスト

引数定義リストは、0 個以上の引数定義を括弧 "(" および ")" で囲ったものです。引数定義の間はカンマ記号 "," で区切ります。

引数定義の最も簡単なものは、単に識別子を記述したものです。関数が呼び出されると、呼び出し時の引数位置に対応する識別子に値を代入し、関数を評価します。

識別子の後にアトリビュートをつけると、それを引数の型として扱います。異なる型の値をこの引数に渡すと、最初に型変換を試み、それに失敗するとエラーになります。

引数のアトリビュートに ":nomap" を指定すると、その引数にリストやイテレータが指定されても暗黙的マッピングで展開されないようになります。型指定のアトリビュートと ":nomap" は併記が可能です。

識別子の後に、辞書代入演算子 "=>" と値を指定すると、それが引数のデフォルト値になります。指定の位置の引数を省略すると、定義されたデフォルト値がかわりに変数に設定されます。

識別子の前にバッククオート "`" をつけると、その位置に指定した引数は、評価前の式が関数に渡されます。

識別子の後に対になった角括弧 "[]" をつけると、その引数はリストを受け取ります。関数呼び出しでイテレータがこの引数に渡されると、リストに変換されます。リストとして扱えない要素を渡すと型エラーになります。

識別子の後にクエスチョンマーク "?" をつけると、その引数はオプショナルになります。

識別子の後にアスタリスク "*" やプラス記号 "+" をつけると、可変長引数を受け付けるようになります。アスタリスクをつけた場合、引数は 0 個以上の値を受け付けます。つまり、対応する位置に引数がなくてもエラーにはなりません。一方、プラス記号をつけると、引数は 1 個以上の値を受け付けます。対応する位置に引数がひとつも指定されていないとエラーになります。

引数リストの中に、識別子に続いてパーセント記号 "%" が記述された要素があると、名前つき引数の内容がこの識別子に辞書として格納されます。

7.3.4. 暗黙的マッピングと関数アトリビュート定義

この節では暗黙的マッピングと関係のある関数アトリビュート定義について説明します。暗黙的マッピングの詳細は後の章を参照ください。

関数アトリビュートとして :map をつけると、その関数は暗黙的マッピングが有効であることを表します。

アトリビュート :void は、関数が常に nil 値を返すことを宣言するものです。通常、関数というものはある入力を受け取って何らかの処理をし、その結果として値を返します。戻り値が常に nil ということは、その関数の処理結果が戻り値としてでなく、なにがしかの状態または外部 I/O への働きとして現れることを示唆します。例えば、画面に文字列を表示する println 関数は :void アトリビュートをつけて定義されていますが、この関数の処理結果は標準出力 I/O へのアクセスという形で現れます。

この宣言は、暗黙的マッピングを適切に働かせるために重要です。暗黙的マッピングでは、引数にイテレータが渡されたとき、関数の処理を含めたイテレータを返すというルールがあります。つまり、引数にイテレータが入っていると、関数の処理が即座に行われないのです。例えば、println(1..10) という記述があったとき、ユーザは 1 から 10 までの数字を即座に表示することを期待しています。しかし、関数 println に渡されているのはイテレータなので、暗黙的マッピングのルールに基づくと、この呼び出しでは所定の処理を行うイテレータが返され、表示処理そのものは遅延されることになります。

しかし、アトリビュート `:void` がついていると、スクリプトはその関数が値を返す類のものでないことを知ることができます。これは、関数が内部でデータを「消費」していると見ることができますが、そういった処理のため、イテレータを渡しても即時実行するように動作を切り替えるわけです。

アトリビュート `:reduce` をつけた関数は、常に同じ実体を返すことを表します。このアトリビュートの用途として想定しているものの一つは、`self` 参照を返すメソッドの定義です。

クラス `Hoge` があり、メソッド `Hoge#foo(x)`と`Hoge#bar(y)` が実装されていると仮定します。このとき、これらのメソッドがインスタンスへの参照 `self` を戻り値として返すように作られていると、クラス `Hoge` のインスタンス `hoge` へのメソッド呼び出しを `hoge.foo(1).bar(2)` のように続けて記述する、いわゆるメソッドチェーンが可能になります。これは、プログラムを簡潔に表記するのに便利ですが、暗黙的マッピングのルールを適用したときに不都合が起こります。例えば、`hoge.foo([1, 2, 3])` のようにメソッドを呼び出すと、暗黙的マッピングによってリスト要素ごとの処理を行い、戻り値が `[self, self, self]` というリストになります。これは同じインスタンスへの参照を含むリストが呼び出しごとに生成されることになり、非効率的です。さらに、このような値が帰ってきてしまうと、前述のようなメソッドチェーンが記述できなくなります。

関数定義のときにアトリビュート `:reduce` をつけておくと、暗黙的マッピングで繰り返し処理を行う際、最初に評価した値を常に返すようになります。前の例で `hoge.foo([1, 2, 3])`という呼び出しがされても、この戻り値はリストではなく `self` になります。これにより、メソッドチェーン中に暗黙的マッピングを働かせて `hoge.foo([1, 2, 3]).bar(2)` というような記述が可能になります。

7.3.5. ブロック定義

関数にブロックを渡せるようにするには、引数定義リストとアトリビュート定義に続いて、ブロック要素を受け取る識別子をブレース記号 `"{"` および `"}"` で囲んだものを指定します。識別子は、慣例的に `block` という名前をつけることが多いですが、任意の名前をつけることができます。

ブロック式定義がない関数に、ブロックをつけて呼び出すとエラーになります。

逆に、ブロック式定義された関数は、呼び出しの際、必ずブロックを記述しなければいけません。ブロックを記述しないとエラーになります。ただし、ブロック式定義中の識別子の後にクエスチョンマーク `"?"` をつけると、そのブロックはオプショナルになります。関数は、ブロックなしでもありでも呼び出すことができるようになります。

ブロックは、関数インスタンスとして識別子に代入されます。ブロック式をオプショナル指定にした関数を、ブロックなしで呼び出すと、この識別子には `nil` が代入されます。

識別子の前にバッククオート `"`"` をつけると、ブロックは関数インスタンスでなく quoted 値として代入されます。

7.3.6. ブロック定義の例

ブロック定義をした関数宣言の例を以下に示します。

```
f(x:number) {block} = {
    block(x)
    block(x + 1)
    block(x + 2)
}
```

以下のように呼び出します。

```
f(2) {|x|
    print(x)
}
```

ブロックの引数は外部のスコープと独立しています。

```
x = 0
f(2) {|x|
    print(x)
}
println(x)
```

ブロック式に割り当てる名前は何でもかまいません。

```
f(x:number) {yield} = {
    yield(x)
    yield(x + 1)
    yield(x + 2)
}
```

ブロックをオプション指定で宣言したとき、ブロックをつけないで関数を呼び出すとブロック式のシンボルには`nil`が渡されます。

```
f_opt() {block?} = {
    if (block == nil) {
        println("not specified")
    } else {
        block()
    }
}
f_opt()
f_opt() {
    println("message from block")
}
```

ブロックは通常、それを実行している関数の「外側」の環境にアクセスできる変数スコープで動作します（「外側」とはレキシカルスコープのそれになりますが、ダイナミックスコープに切り替えることもできます)。関数外部の変数の値を変更することができます。

```
g() {block} = {
    block()
}
n = 2
g() {
    n = 5
}
```

```
printf("n = %d¥n", n) # n = 5
```

ブロックシンボルにアトリビュート :inside_scope をつけると、関数内部のスコープに切り替わり、関数の内部処理で設定される変数などにアクセスできるようになります。関数外部の変数は、参照できた値に対しての改変が可能です。

```
h() {block:inside_scope} = {
    m = "local in h()"
    block()
}
h() {
    printf("%s¥n", m)    # h()'s local variable m is accessible
}
n = 2
h() {
    n = 5
}
printf("n = %d¥n", n)    # n = 2
h() {
    n += 5
}
printf("n = %d¥n", n)    # n = 7
```

quoted value にしたブロックを設定した変数を {|..|} で囲って関数に渡すと、それがブロック本体として扱われます。ブロックパラメータも記述できます。例えば:

```
f() {block} = {
    block(1, 2, 3, 4)
}
```

という関数があった場合、以下のふたつの呼び出しは等価です。

```
block = `{|a, b, c, d|
    printf("%d %d %d %d¥n", a, b, c, d)
}
f() {|block|}

f() {|a, b, c, d|
    printf("%d %d %d %d¥n", a, b, c, d)
}
```

7.3.7. 関数定義の例

関数の引数には、オプション引数・デフォルト値・可変長引数を指定できます。

```
f1(a, b?, c?) = printf("%s, %s, %s¥n", a, b, c)
f1(2)                       # 2, nil, nil
```

```
f2(a, b => 10, c => "abc") = printf("%s, %s, %s¥n", a, b, c)
f2(2)                      # 2, 10, abc

f3(a, b, c*) = printf("%s, %s, %s¥n", a, b, c):nomap
f3(2, 3, 4, 5, 6, 7)       # 2, 3, [4, 5, 6, 7]
f3(2, 3)                   # 2, 3, []

f4(a, b, c+) = printf("%s, %s, %s¥n", a, b, c):nomap
f4(2, 3, 4, 5, 6, 7)       # 2, 3, [4, 5, 6, 7].
f4(2, 3)                   # error. c has to get at least one value.
```

関数呼び出しの際は、キーワード引数指定ができます。キーワードと値は、辞書演算子（=>）で対応づけます。

```
g1(a, b, c) = printf("%s, %s, %s¥n", a, b, c)
g1(2, b => 3, c => 4)              # 2, 3, 4
```

引数リストの中に、% を後尾につけたシンボルを加えておくと、引数リストに合致しないキーワード引数指定の組を辞書にした値をそのシンボルに割り当てます。

```
g2(a, b, dict%) = printf("%s, %s, %s¥n", a, b, dict)
g2(2, b => 3, c => 4, d => 5)       # 2, 3, %{c => 4, d => 5}
g2(2, 3, c => 4, d => 5)           # 2, 3, %{c => 4, d => 5}
```

引数宣言のシンボル名の先頭にバッククオートをつけると、未評価の式（quoted value）を値として渡せます。この機能を使って、制御構文を実現することができます。以下は、quoted value を使って、C の for ステートメントのような動作をする関数を作成している例です。

```
c_like_for(`init, `cond, `next):dynamic_scope {block:inside_scope} = {
    env = outers()
    env.eval(init)
    while (env.eval(cond)) {
        block()
        env.eval(next)
    }
}
n = 0
c_like_for (i = 1, i <= 10, i += 1) {
    n += i
}
printf("i = %d, sum = %d¥n", I, n)
```

7.3.8. 関数の戻り値

関数の本体で、一番最後に評価された式の値が関数の戻り値になります。

また、return 関数を使って戻り値を指定することもできます。一般式は以下のとおりです。

```
return(value?):symbol_func
```

return 関数を呼ぶと、関数の処理を中断して処理を呼び出しもとに戻します。このとき、引数 value を指定すると、その値を関数の戻り値として扱います。引数を省略すると、nil を戻り値とします。

この関数は :symbol_func アトリビュートをつけて定義されているので、文中に "return" というシンボルが記述されていると、return 関数の処理が行われます。

7.4. 関数呼び出しの連結関係

ブロックの終端ブラケット '}' の後、同じ行に関数呼び出しの式が続くと、二つ目の関数呼び出しは前の関数と連結関係を持つようになり、二つ目の関数が評価されるか否かは最初の関数の実行内容によって制御されます。例えば、一行の間に func1(){}func2(){} と記述すると、func1 が func2 の評価をするか否かを決めることができるようになります。関数はいくつでも連結することができます。

この機能を使う代表的な例が if-elsif-else シーケンスと、try-except シーケンスです。

例えば、if と elsif を使った条件文は以下のようになります。

```
if (cond1) { process1 } elsif (cond2) { process2 }
```

この文は、最初に if 関数を評価します。if 関数は引数 cond1 の結果を真値と判断すると自身のブロック内容 process1 を評価し、続く連結式を評価しません。逆に cond1 の結果を偽値と判断すると、連結されている elsif 関数の呼び出しを評価します。elsif 関数は引数 cond2 の結果を真値と判断すると自身のブロック内容 process2 を評価します。

同一行に書く必要があるのは終端ブラケット '}' と関数インスタンスの式の間だけなので、あとの要素は行を分けて記述することができます。前述の if-elsif の文は以下のように記述できます。

```
if (cond1) {
    process1
} elsif (cond2) {
    process2
}
```

連結関係を認識しない関数に連結式をつなげると、単に無視されて評価されません。

7.5. 名前なし関数

関数 lambda を使うと、名前なし関数を生成することができます。名前なし関数の概念と、この lambda という名前は LISP から来ています。関数 lambda の一般式は以下の通りです。

```
lambda(`args*) {block}
```

args に引数指定、block に関数本体のコードを記述します。引数指定は、通常の関数定義と同じ文法で記述することができます。

引数指定が必要ない場合、lambda のかわりに "&{…}" という形式を使って関数を生成することもできます。一般式は以下の通りです。

```
&{block}
```

どちらの形式でも、通常の関数定義にはない機能があります。それは、関数本体のコードの中に、先頭がドル

記号 "$" で始まる識別子があると、その識別子が出現した順に引数リストに追加するというものです。これは、"&{…}" の形式を使って簡易的に関数インスタンスを生成したいときに便利です。以下の 2 つの表記は同じ機能を持つ関数の定義になります。

```
&{println($foo, $bar)}
lambda (foo, bar) {println(foo, bar)}
```

名前なし関数は、クロージャを実現するのに使われます。

```
new_counter(n:number) = {
    lambda() { n += 1}
}
cnt = new_counter(2)
printf("%d¥n", cnt())
printf("%d¥n", cnt())
printf("%d¥n", cnt())
```

7.6. メソッド呼び出し

インスタンスメソッドは class#method()のようにクラス名とメソッド名を "#" でつなげたもので表記します。これはドキュメントやヘルプなど、メソッドのふるまいを説明する資料でのみ使われる表記方法です。実際の呼び出しでは、例えばインスタンスの変数名が obj だとすると、obj.method() のようになります。

7.7. メソッド定義

メソッドの定義は、class 関数のクラス宣言で行います。
代入演算子を使い、クラス宣言をした後から、インスタンスやクラスにメソッドを追加することもできます。

```
x = "hello"
x.hoge() = println("This string is: ", self)
x.hoge()
```

クラスにメソッドを追加する場合も同様です。以下は、文字列クラスにメソッド print を定義する例です。classref 関数は組込みクラスの参照を得る関数です。

```
classref(`string).hoge() = println("This string is: ", self)
```

8. 制御構文

Gura には制御構文という特別な要素は存在しません。が、他のプログラミング言語でおなじみの条件分岐や繰り返しといった処理によく似た形式で実行することができる関数を提供しています。この章ではそれらの関数の動作内容を見ていきます。あわせて Gura に特有の、リスト・イテレータ生成の方法も説明します。

8.1. 条件分岐

条件分岐を行う if-elsif-else シーケンスの一般式は以下のようになります。

```
if (`cond) {block} elsif (`cond) {block} elsif (`cond) {block} else {block}
```

ひとつの `if` に対して、0 個以上の任意の数の `elsif` を記述できます。`else` はひとつのみです。ブロック内に記述する式がひとつだけであっても、ブロックを囲むブレース記号 `"{"` および `"}"` は省略できないので注意してください。

if-elsif-else シーケンスを評価すると、条件に合致したブロックの評価値を全体の値として返します。この性質を使って、C 言語でおなじみの三項演算子、すなわち `result = flag? a : b` という形式を以下のように記述することができます。

```
result = if (flag) {a} else {b}
```

8.2. 繰り返し

繰り返しを実現する関数には `repeat`, `while`, `for` および `cross` があります。

Gura の繰り返し関数は、単にリピート処理をするだけではありません。ループが一回まわるごとに、評価した値をリストの要素として残していく機能を使うと、リストの生成をシンプルに記述できます。また、繰り返し処理をその場で評価せず、処理を内包したイテレータを生成するという機能もあるので、クロージャの生成機構としてふるまわせることも可能になります。

8.2.1. repeat 関数

`repeat` 関数の一般式は以下のようになります。

```
repeat (n?:number) {block}
```

`repeat` 関数は、引数で指定された回数だけ処理を繰り返します。引数は省略可能で、省略した場合無限ループになります。

8.2.2. while 関数

`while` 関数の一般式は以下のようになります。

```
while (`cond) {block}
```

`while` 関数は、引数で指定された式が条件を満たす間だけ処理を繰り返します。

8.2.3. for 関数

`for` 関数の一般式は以下のようになります。

```
for (`expr+) {block}
```

`for` 関数は、一つ以上のイテレータ代入式を引数にとり、イテレータが終了するまで処理を繰り返します。
イテレータ代入式の形式は以下のようになります。

```
symbol in iterator
[symbol1, symgol2 ..] in iterator
```

最初の形式では、イテレータの要素が *`symbol`* で表される変数に代入されます。もし要素がリストであれば、*`symbol`* に代入される値はそのリストそのものになります。二番目の形式では、イテレータの要素がリストであれ

ばリストの要素ごとに対応する位置にあるシンボルの変数に値を代入します。要素がリストでない場合、全てのシンボルの変数に同じ値が代入されます。

イテレータ代入式が二つ以上指定された場合、一回のループで引数中のイテレータを一つずつ評価していきます。こうして、いずれかのイテレータが終了するまで処理が繰り返されます。つまり、イテレータの要素数が異なるときは、ループの回数は一番短いイテレータの要素数にあわせられます。

8.2.4. cross 関数

cross 関数の一般式は以下のようになります。

```
cross (`expr+) {block}
```

cross 関数は、一つ以上のイテレータ代入式を引数にとり、イテレータが終了するまで処理を繰り返します。イテレータ代入式が一つのとき、処理内容は for 関数に一つの引数を渡したときと同じです。二つのイテレータ代入式を指定すると多重ループになり、一つ目のイテレータが外側、二つ目のイテレータが内側のループを構成します。イテレータ代入式を複数指定することも可能で、n 個の代入式を指定すると n 重の多重ループになります。

cross 関数の実行例を以下に示します。

```
>>> cross (x in ["Taro", "Hanako"], y in 1..3) { println(x, " ", y) }
Taro 1
Taro 2
Taro 3
Hanako 1
Hanako 2
Hanako 3
```

8.2.5. 繰り返しにおけるフロー制御

繰り返し関数を途中で抜けるために、break 関数が用意されています。この関数を評価すると、一番内側の繰り返し関数の処理を中断します。一般式は以下のとおりです。

```
break(value?):symbol_func
```

アトリビュート:symbol_func は、この関数が単独のシンボルで記述したときでも、関数呼び出しとして評価することを指定するものです。引数として value を渡すと、中断した繰り返し関数の戻り値をその値に設定します。省略すると、この戻り値は nil になります。

例を以下に示します。

```
for (str in strList) {
    if (str == "end") { break }
}
```

繰り返し処理の続きをスキップして先頭に戻るには continue 関数を使います。一般式は以下のとおりです。

```
continue(value?):symbol_func
```

この関数も break 関数と同じように、シンボルのみで関数呼び出しになります。引数として value を渡すと、ループのその回の評価値をその値に設定します。省略すると、その回の評価値は nil になります。

8.2.6. 繰り返し関数によるリストの生成

繰り返し関数 repeat, while, for, cross は、デフォルトでは一番最後のループで評価した値をその関数自体の戻り値とします。しかし、アトリビュート :list または :xlist を指定すると、ループごとの評価値を要素にもつリストを返すようになります。アトリビュート :list を指定すると、すべての評価値を要素に持つリストになります。アトリビュート:xlist では、評価値が nil になるものを要素から除外します。

8.2.7. 繰り返し関数によるイテレータの生成

繰り返し関数 repeat, while, for, cross は、デフォルトでは繰り返し条件に基づいて即座に block の内容を評価します。しかし、アトリビュート :iter または :xiter を指定すると、その場で評価することはせず、ループの内容を一度ずつ評価するイテレータを返すようになります。アトリビュート :iter を指定すると、すべての評価値を返すイテレータになります。アトリビュート:xiter では、評価値が nil になるものをスキップするイテレータを返します。

この機能を使った繰り返し関数の見た目や用途は、今まで見慣れたものとは大分異なります。以下の例を見てみましょう。

```
x = repeat (10):iter { println("Hello") }
```

この式を評価しても、画面には何も表示されません。ここでは「"Hello"を10回表示するイテレータ」を生成し、x に代入しています。イテレータを、例えば "x.next()" のように評価すると画面に "Hello" を表示します。

少し複雑な例として、多重ループのイテレータを生成してみます。以下は、1から3までの数値の3つの組み合わせを返すイテレータの例です（これと同じ処理は cross 関数を使うともっと簡単に実現できますが、多重ループの例としてとりあげています)。

```
iter = repeat (3):iter {|i|
    repeat (3):iter {|j|
        repeat (3):iter {|k|
            [i, j, k]
        }
    }
}
```

生成されたイテレータを iter.each():list のように評価すると、リストで結果を得られます。

8.3. 例外処理

例外処理を行う try-except シーケンスの一般式は以下のようになります。

```
try {block} except (error*:error) {block} except (error*:error) {block}
```

ひとつの try に対して 1 個以上の任意の数の except を記述することができます。

通常、スクリプトの実行中に例外が発生するとスクリプトが中断されます。しかし、try 関数のブロック中で発生

した例外はこの関数が捕捉し、それから後続する except 関数にエラー内容を順番に渡していきます。except 関数は、引数で指定されたエラーインスタンスと渡されたエラー内容を比較し、等しいと判断したときは自身のブロックの内容を実行し、この try-except シーケンスを終了します。もしいずれの except 関数の条件にも合致しないときは、通常どおりのエラー処理が行われます。

except 関数は、0 個以上のエラーインスタンスを引数にとることができます。引数がなにも指定されないと、それまでの except 関数で合致しなかった残りのすべての例外をその場で捕捉します。1 個以上指定された場合は、いずれかのエラーインスタンスに合致すれば捕捉することになります。

except 関数に渡すエラーインスタンスは以下のようなシンボル名で定義されています。

```
SyntaxError、ArithmeticError、TypeError、ZeroDivisionError、ValueError
SystemError、IOError、IndexError、KeyError、ImportError、AttributeError
StopIteration、RuntimeError、NameError、NotImplementedError、IteratorError
CodecError、CommandError、MemoryError、FormatError、ResourceError
```

except 関数のブロックは、|error:error| という形式のブロック引数を受け取ります。引数 error は実際に発生したエラーに対応するエラーインスタンスで、エラー種別やメッセージなどをメンバに含みます。ユーザは、この情報をもとに適切な処理を実装することができます。

関数 raise を使って、ユーザが意図的に例外を発生させることもできます。一般式は以下の通りです。

```
raise(error:error, msg:string => 'error', value?)
```

9. 暗黙的マッピング

9.1. 実装のきっかけ

数式 $y = x^2$ のグラフを描画する処理を考えてみます。座標値は、x に-5 から 5 までの数値を 1 きざみで代入したときの y の値を求め、各座標値に対応する画面位置にプロットすることにしましょう。

従来のプログラミング言語でこのような処理を行うには、ループ構文を記述して繰り返し処理するというのが常套手段でした。C 言語であれば、以下のようなプログラムを思い浮かべることができます。

```
const float x[] = {-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5 };
float y[11];
for (int i = 0; i < 11; i++) {
    y[i] = x[i] * x[i];
}
```

関数プログラミングを提唱している言語ならば、同じ処理をするのに高階関数を適用することを思いつくでしょう。LISP の場合、写像処理を行う map を使って以下のように記述できます。

```
(map (lambda (x) (* x x)) '(-5 -4 -3 -2 -1 0 1 2 3 4 5))
```

かなりエレガントに書くことができました。LISP に限らず、高階関数という概念はものごとを抽象的にとらえる強力な武器になります。しかし抽象的な思考というものは、得てしてその道の入門者にとってはとっつきづらいものです。そもそも、ここで実際に解決したいのは、x の数列に対応する x^2 の値を求めるという単純な課題です。以

下のような記述で、答えが求まらないものでしょうか。

```
x = [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5]
y = x * x
```

x * x という表記は、x にひとつの数値を受け取ることを期待しています。これに対して、x に数値のリストを与えたとき、暗黙的に写像すなわちマッピングを行うようにすれば、ユーザは繰り返し処理を意識することなく結果を得られるようになります。

「暗黙的マッピング」の実装はこのような発想からスタートしました。

9.2. コンセプト

Gura の演算子は、ほとんどすべて暗黙的マッピングが有効になります。これは、数式を構成する四則演算だけでなく、大小比較などの演算子を含みます。ユーザが書いた演算式は、そのまま数列を処理する機能を持つことになります。

演算子に加え、関数（組込み関数とユーザ定義関数）も、暗黙的マッピングの宣言をしていればこの機能が働くようになります。つまり、引数にデータ列を渡すと、データ列の要素ごとにくりかえし関数が実行されるのです。Gura が提供する組込み関数や標準モジュールの関数のほとんどは、暗黙的マッピングの宣言がされています。

9.3. 適用ルール

暗黙的マッピングは、"&&" と "||" を除くすべての演算子について適用されます。これらの演算子が除外されているのは、引数に評価前の式を受け取るものだからです。

暗黙的マッピングはまた、アトリビュート :map をつけて宣言された関数に対しても働きます。Gura が標準で提供する関数の多くは、この宣言をつけて提供されています。

Gura において、データ列を表現するデータ型はリストとイテレータです。リストやイテレータが、演算子や関数に渡されると、暗黙的マッピングが行われるようになります。

以下説明のため、データ型を 3 つのカテゴリに分類します。つまり、リスト、イテレータ、そしてそれ以外のスカラーです。

暗黙的マッピングによって得られる結果は、関数のアトリビュート指定や、引数のカテゴリがリスト、イテレータまたはスカラーのどれなのかによって異なります。デフォルトの動作では、引数にイテレータがひとつでも含まれると、結果はイテレータになります。以下に戻り値の条件をまとめます。

引数カテゴリ	戻り値
スカラーのみ	スカラーについて関数を実行し、結果を返します
スカラーかリスト (イテレータは無い)	リストの要素ごとに関数を実行し、その結果をリストとして返します
イテレータが含まれる	イテレータを結果として返します

暗黙的マッピング宣言された関数 func(a, b) の呼び出しを例にとって考察します。引数 a, b にスカラー、リスト、イテレータを渡したときの戻り値は以下のようになります。

```
func(scalar, scalar)          scalar
```

```
func(scalar, list)          list
func(list, list)            list
func(scalar, iterator)      iterator
func(iterator, list)        iterator
func(iterator, iterator)    iterator
```

戻り値の型を変えたい場合は、関数呼び出しでアトリビュートを指定します。暗黙的マッピングの戻り値を変更するアトリビュートの一覧を以下に示します。

アトリビュート	説明
:list	リストを返します。
:xlist	nil 値を要素から除外したリストを返します。
:set	重複する値を要素から除外したリストを返します。
:xset	nil 値と重複する値を要素から除外したリストを返します。
:iter	イテレータを返します。
:xiter	nil 値をスキップするイテレータを返します。

9.4. ケーススタディ

9.4.1. 演算子と暗黙的マッピング

Gura の演算子に暗黙的マッピングを適用した例を以下に示します。

```
>>> [1, 2, 3, 4] + [5, 6, 7, 8]
[6, 8, 10, 12]
>>> [1, 2, 3, 4] + 5
[6, 7, 8, 9]
>>> ([1, 2, 3, 4] + [5, 6, 7, 8]) / 2
[3, 4, 5, 6]
>>> [3, 8, 0, 4] < [4, 5, 3, 1]
[true, false, true, false]
```

演算子の暗黙的マッピングと、リスト・イテレータ操作とを組み合わせるといろいろな処理が簡潔に表現できます。

リスト x, y の内積を計算 ……(x * y).sum()
数値リスト x の中で、10 未満の要素をカウント ……(x < 10).count()
数値リスト x の中で、3 以上 10 以下の要素をカウント ……((3 <= x) & (x <= 10)).count()

9.4.2. 文字列出力との組み合わせ

暗黙的マッピング処理をさまざまなデータ入出力関数や処理関数と組み合わせると、制御構文を記述することなく多くの課題を解決することができます。以下に例をあげます。

```
>>> x = [1, 2, 3, 4]
>>> printf("result = %2d, %2d, %2d, %f¥n", x, x * x, x * x * x, math.sqrt(x))
```

```
result =  1,  1,  1, 1.000000
result =  2,  4,  8, 1.414214
result =  3,  9, 27, 1.732051
result =  4, 16, 64, 2.000000
```

x * x や math.sqrt(x) などの式で暗黙的マッピング処理が働いてリスト要素ごとの演算をしています。さらに関数 printf の実行でも、リストが引数として与えられたことによってやはりこの機能が作動し、要素ごとの表示処理をします。 printf は値を持たない関数なので、結果としてのリストは生成しません。

9.4.3. ファイル入力との組み合わせ

行番号をつけてファイルを表示するプログラムは以下のように書けます。

```
printf("%7d %s", (1..), open("hoge.txt").readlines())
```

1.. と stream#readlines はリストではなくイテレータを返します。1.. は 1 から始まる無限数列を表しますが、長さの異なるリストやイテレータが与えられた場合は短い方にあわせられるので表示する行数は stream#readlines が終了するまでになります。

9.4.4. パターンマッチングとの組み合わせ

以下は正規表現を使ってファイルから情報を抽出し、表示する例です。

```
import(re)
lines = open("hoge.h").readlines()
println(re.match(r"class (¥w+)", lines).skipnil():*group(1))
```

stream#readlines で生成したイテレータ lines を受け取った関数 re.match は、結果として re.match_t インスタンスを要素にするイテレータを返します。 関数 re.match は、パターンに合致しない場合は nil を返すので、イテレータのインスタンスメソッド iterator#skipnil を使って nil 値をスキップするイテレータを生成します。":*" は後述するメンバマッピングオペレータで、上の例ではイテレータの各要素に対して re.match_t#group メソッドを実行しています。

10. メンバマッピング

暗黙的マッピングは、関数の引数にリストやイテレータが渡されたときに、それらを展開して関数を評価する機能でした。メンバマッピングは、メンバアクセスのレシーバになった対象がリストやイテレータだったとき、その要素に対して一つずつメンバアクセス処理をするものです。

メンバマッピングには、マッピングの結果をリストとして得る map-to-list、マッピングの結果をイテレータとして得る map-to-iterator、そして暗黙的マッピングのルールに基づいて要素を走査する map-along という 3 つのモードがあります。モードはレシーバとメンバを結合する記号によって切り替えます。

モード	記号	説明
map-to-list	::	リスト中のオブジェクトごとにメンバを評価し、その結果をリストとして返します。 例えば、objs::method() は以下のコードと同じ結果になります。

		`for (obj in objs):list { obj.method() }`
map-to-iterator	`:*`	リスト中のオブジェクトごとにメンバを評価するイテレータを返します。例えば、`objs:*method()` は以下のコードと同じ結果になります。 `for (obj in objs):iter { obj.method() }`

map-along	`:&`	引数の値をもとに暗黙的マッピングを行いますが、このときレシーバであるリストの要素も順に走査していきます。例えば、`as, bs, cs` を何らかのリストと仮定すると、`objs:&method(as, bs, cs)` は以下のコードのように要素を走査します（結果は異なります)。
		`for (obj in objs, a in as, b in bs, c in cs) {`
		`  obj.method(a, b, c)`
		`}`
		この形式は、暗黙的マッピングとメンバマッピングがくみあわさった形と見ることができます。

10.1. ケーススタディ

簡単なクラスを宣言して、メンバマッピングの用例を見ていきます。

以下は、名前と値段を表示する `Print()` というメソッドを持った `Fruit` クラスを作った後、`Fruit` クラスのインスタンスのリスト `fruits` を生成しています。

```
Fruit = struct(name:string, price:number) {
    Print() = printf("name:%s  price:%d¥n", self.name, self.price)
}
fruits = @(Fruit) {
    { "apple", 100 }, { "orange", 80 }, { "grape", 120 }
}
```

`fruits` の要素について `Print` を実行するには、メンバマッピングを使って以下のように記述します。

```
fruits::Print()
```

値段の合計と平均を計算します。

```
printf("sum = %.1f, average = %.1f¥n",
          fruits::price.sum(), fruits::price.average())
```

一番長い名前にそろえて一覧表示します。一見簡単そうなこの処理は、制御構文を使うと意外と煩雑になります。メンバマッピング処理で簡潔な記述が可能になります。

```
printf("%-*s %d¥n",
          fruits::name::len().max(), fruits::name, fruits::price)
```

上と同じですが、イテレータとしてメンバマッピングを処理しています。要素数が多いときは、こちらの方が実行速度が速くなります。

```
printf("%-*s %d¥n",
```

```
        fruits:*name:*len().max(), fruits:*name, fruits:*price)
```

関数インスタンスを使って、値段が 100 円未満のものを表示します。

```
fruits.filter(&{$f.price < 100})::Print()
```

以下の例は、上と同じ処理を、暗黙的マッピングと組み合わせて処理しています。

```
fruits.filter(fruits:*price < 100)::Print()
```

値段や名前をキーにしてソートをします。

```
fruits.sort(&{$f1.price <=> $f2.price})::Print()
fruits.sort(&{$f1.name <=> $f2.name})::Print()
```

11. ユーザ定義クラス

class 関数を使うと、ユーザ定義のクラスを作成することができます。class 関数の一般式は以下のとおりです。

```
class(superclass?:function):[static] {block?}
```

class 関数は、作成したクラスのコンストラクタ関数を返します。

引数 superclass には、親クラスのコンストラクタ関数を指定します。省略すると、Gura のルートクラス object を親クラスとします。

block 中には、クラスに定義するメソッド定義やクラス変数の指定を記述します。メソッド定義の仕方は、通常の関数定義と同じです。メソッド中で自分が属しているインスタンスを参照するには、self 変数を使います。

定義するメソッドの中には、以下のように特殊な働きをするものがあります。

__init__(…)

コンストラクタ関数の定義をします。この関数で定義した引数やブロック式が、class 関数で返される関数インスタンスの引数になります。

__del__()

インスタンスが削除されるときに呼ばれるメソッドです。

__getprop__(symbol:symbol)

インスタンスに対してプロパティ参照をした際、指定のプロパティがインスタンス内で定義されていないときに呼ばれます。引数 symbol にプロパティ名が渡されるので、対応するプロパティ値を返します。

例えば、foo.bar という式が評価され、foo インスタンスの中にプロパティ bar が存在しないと __getprop__が呼ばれ、symbol に`bar が入ります。

__putprop__(symbol:symbol, value)

インスタンスに対してプロパティ代入をしたときに呼ばれるメソッドです。引数 symbol に設定するプロパティのシンボル、value に値が渡されます。このメソッドで代入処理をした場合は true、しなかった場合は false を返します。

例えば、foo.bar = 3 という式が評価されると __putprop__ が呼ばれ、symbol に`bar、value に

数値 3 が入ります。

`__getitem__(key)`

インスタンスに対してインデクス参照をしたときに呼ばれるメソッドです。引数 `key` には、キーとして指定された値が渡されます。

例えば、`foo["hoge"]` という式が評価されると `__getitem__` が呼ばれ、`key` に文字列 `"hoge"` が渡されます。

`__setitem__(key, value)`

インスタンスに対してインデクス代入をしたときに呼ばれるメソッドです。引数 `key` には、キーとして指定された値、`value` には代入値が渡されます。

例えば、`foo["hoge"] = 3` という式が評価されると `__setitem__` が呼ばれ、`key` に文字列 `"hoge"` が、`value` に数値 3 が入ります。

`__str__()`

インスタンスを文字列として評価するときに呼ばれるメソッドです。

11.1. 構造体のユーザ定義

Gura における構造体は、クラスの特殊な形式として実装されています。構造体は `struct` 関数で作成することができます。`struct` 関数の一般式は以下のとおりです。

```
struct(`args+):[loose] {block?}
```

`block` には `class` 関数の `block` と同じように、メソッド定義やクラス変数の定義を記述します。

11.2. 型変換

以下の操作をしたとき、データの型変換が行われます。

- 型指定された引数に値を渡すとき
- アトリビュートで型指定された変数に値を代入するとき

12. モジュール

モジュールは、関数やクラスを提供するファイルです。モジュールには、通常の Gura スクリプトで記述されたスクリプトモジュール（`*.az`）と、C++で記述してビルドしたバイナリモジュール（`*.azd`）があります。スクリプトをモジュールとして使用する場合、コード中に特別な記述をする必要はありません。

モジュールを現在実行しているスクリプト中にとりこむには、`import` 関数を使用します。`import` 関数は、引数としてモジュール名を受け取り、そのモジュール名にサフィックス（`.az` または `.azd`）をつけたファイルを指定のパスから探索します。探索パスは `sys` モジュール中の変数 `sys.path` に配列の形式で指定します。この変数の内容を書き換えると、モジュールの探索処理に反映されます。Windows 環境では、デフォルトで以下の順にモジュールを探索します（`gura.exe` が存在するディレクトリを `%GURA_DIR%` で表しています）。

1. カレントディレクトリ

2. %GURA_DIR%¥module
3. %GURA_DIR%¥module¥site

Linux 環境では以下のようになります（ディレクトリのプレフィックスが /usr/local になるか /usr になるかは、インストール時のコンフィグレーションによって決まります)。

1. カレントディレクトリ
2. /usr/local/lib/gura/ または /usr/lib/gura
3. /usr/local/lib/gura/site または /usr/lib/gura/site

import 関数の最も基本的な使い方は、単にモジュール名を引数として渡すものです。例えば、CSV フォーマットの読み書きをするモジュール csv をインポートするには、以下のようにします。

```
import(csv)
```

これで csv モジュールが読み込まれ、csv という名前でモジュール内のシンボルを参照できるようになります。例えば、csv モジュール内の read という関数を呼び出すには、csv.read(stream)のように記述します。

場合によっては、モジュール内のシンボルを現在の名前空間にとりこんで、モジュール名なしに参照したいこともあります。そのような場合は、import 関数の後にブロックを記述し、とりこむシンボル名を列挙します。例えば、csv モジュールの read および write 関数をとりこむには以下のように記述します。

```
import(csv) {read, write}
```

これで、プログラムからは read(stream) のように呼び出すことができます。モジュール内のシンボルをすべて取り込むこともでき、その場合はアスタリスク "*" をブロック内に記述します。以下は、opengl モジュールのすべてのシンボルをとりこむ例です。

```
import(opengl) {*}
```

ただし、この機能を使うとモジュールのシンボル名が現在の名前空間にすでにあるシンボル名と衝突してエラーになる可能性があるので注意してください。

import 関数に二つ目の引数を指定すると、モジュールを別名で取り込むことができます。この機能は、長い名前のモジュールを短い名前で参照する場合などに便利です。以下は、sqlite3 モジュールを sq という名前で参照する例です。

```
import(sqlite3, sq)
```

13. イテレータ

イテレータは、コンテナ内の要素を順に取得または評価するための機構です。多くのプログラミング言語で取り入れられている概念なので、すでにおなじみの方が多いでしょう。イテレータのもっとも一般的な用途は、繰り返し処理を行う for 構文などに渡して、コンテナ内の要素に順にアクセスするというものです。

Gura におけるイテレータの役割は、他の言語よりもずっと重要です。なぜなら、イテレータは暗黙的マッピングや、メンバマッピングに適用する基本的なデータだからです。そのため、Gura では豊富な種類のイテレータを容易しています。これらを組み合わせると、今まで制御構文で行っていた処理がもっと簡潔な記法で実現できる

ようになります。

13.1. 有限イテレータと無限イテレータ

イテレータには、有限イテレータと無限イテレータがあります。

有限イテレータは、走査に先立って要素の総数があらかじめ分かっているイテレータです。例えば、数列 `1..10` は代表的な有限イテレータです。

一方、無限イテレータは、要素の数が不明なものを指します。実際に走査を始めたら有限な個数で終了したという場合でも、あらかじめ要素数を知る手段が得られないものは無限イテレータと呼ばれます。無限数列 `1..` は代表的な無限イテレータです。

このような区別をつけるのは、イテレータ操作の中には要素数があらかじめ分かっていなければいけないものがあるからです。例えば、要素数を返す `iterator#count` メソッドや、要素を逆順に操作するイテレータを生成する `iterator#reverse` などがこれにあたります。無限イテレータにこれらの操作を行うとエラーになります。

また、イテレータをリストに変換するような操作を無限イテレータに適用すると、エラーになります。

13.2. イテレータ操作とブロック式

イテレータを返す関数は、オプションでブロック式を受け付けます。関数呼び出しの際にブロックが指定されると、イテレータの要素ごとに繰り返しブロックの内容を評価します。このとき、`|value, idx:number|` という形式でブロック引数が渡されます。`value` は要素の値、`idx` はループのインデクス数値です。

13.3. リストの生成

イテレータを返す関数にアトリビュート `:list` をつけて実行するとリスト生成をすることができます。例えば、指定の範囲の数列を出力する `range` 関数にリストを出力するよう指示するには以下のようにします。

```
range(10):list
```

アトリビュート `:list` といっしょにブロック式の指定がされると、ループごとのブロックの評価値を要素に持つリストが生成されます。以下の例は、二乗値を要素に持つリストの生成になります。

```
range(10):list {|x| x * x}
```

13.4. イテレータの生成

指定のルールに基づいて値を出力するイテレータを生成する関数が用意されています。

13.4.1. 同じ値を指定の数だけ出力する

指定の値を指定の数だけ出力するイテレータを生成します。一般式は以下のとおりです。

```
fill(n:number, value?) {block?}
```

引数 `n` に生成する個数、`value` に値を指定します。`value` には任意の型のデータを指定できます。`value` は省略可能で、省略した場合は `nil` 値を要素に持つイテレータになります。

13.4.2. 乱数を指定の数だけ出力する

指定の数だけ乱数を出力するイテレータを生成します。一般式は以下のとおりです。

```
rands(num?:number, range?:number) {block?}
```

引数 num に生成する個数、range に値の上限値を整数で指定します。出力する数値 x は 0 < x < range を満たす整数値になります。num が省略されると、無限に乱数を出力します。range が省略されると、出力する数値 x は 0 < x < 1 を満たす小数値になります。

13.4.3. 指定の範囲内の数列を出力する

開始値と終了値、および間隔を指定して連続する数列を出力するイテレータを生成します。一般式は以下のとおりです。

```
range(num:number, num_end?:number, step?:number):map {block?}
```

引数 num のみを指定すると、0 から num – 1 までの整数を出力します。
引数 num と num_end を指定すると、num から num_end – 1 までの整数を出力します。
引数 num, num_end および step を指定すると、num を開始値にして、step ごとに数値をインクリメントして num_end を超えない範囲までの数値を出力します。
連続した数値を出力するのに、オペレータ ".." を使うこともできます。"n..m" という形式では、n から m までの整数を出力するイテレータになります。また、"n.." と指定すると、n を始点にして、無限にインクリメントするイテレータになります。

13.4.4. 範囲とサンプル数を指定して数列を出力する

範囲とサンプル数を指定して数列を出力するイテレータを生成します。一般式は以下のとおりです。

```
interval(a:number, b:number, samples:number):map:[open,open_l,open_r] {block?}
```

引数 a に最小値、b に最大値を指定すると、サンプル数 samples 個だけ、[a, b] の範囲内で等間隔な数列を出力します。
アトリビュート:open, :open_l, :open_r を指定すると、範囲のオープン条件を指定できます。:open を指定すると、範囲指定が(a, b)に、:open_l では(a, b]、:open_r では[a, b)になります。

13.5. 要素の抽出

すでに存在するリストやイテレータから、条件に従って要素を抽出するメソッドが用意されています。

13.5.1. 先頭から指定数だけ要素を抽出する

先頭から指定の数だけ要素を抽出するイテレータを返します。テキストファイルの先頭から指定数だけ行を表示する UNIX コマンド head から名前をとっています。一般式は以下のとおりです。

```
list#head(n:number):map {block?}
iterator#head(n:number):map {block?}
```

引数 n に抽出する要素数を指定します。
例えば、[A, B, C, D, E, F, G] というリストを代入した変数 x があったとき、x.head(3) を実行すると以下のような要素列を出力するイテレータを生成します。

```
A B C
```

13.5.2. 後尾の指定数だけ要素を抽出する

要素列の後尾から指定の数だけ要素を抽出するイテレータを返します。テキストファイルの終端から指定数だけ行を表示する UNIX コマンド tail から名前をとっています。無限イテレータに対してこのメソッドを実行するとエラーになります。一般式は以下のとおりです。

```
list#tail(n:number):map {block?}
iterator#tail(n:number):map {block?}
```

引数 n に抽出する要素数を指定します。
例えば、[A, B, C, D, E, F, G] というリストを代入した変数 x があったとき、x.tail(3) を実行すると以下のような要素列を出力するイテレータを生成します。

```
E F G
```

13.5.3. 先頭から指定数だけ要素をとばす

head とは逆に、先頭の要素を指定数だけ除外した後の要素を持つイテレータを返します。一般式は以下のとおりです。

```
list#offset(n:number):map {block?}
iterator#offset(n:number):map {block?}
```

引数 n に除外する要素数を指定します。
例えば、[A, B, C, D, E, F, G] というリストを代入した変数 x があったとき、x.offset(3) を実行すると以下のような要素列を出力するイテレータを生成します。

```
D E F G
```

13.5.4. 指定数ずつ要素を除外する

指定数だけ要素を除外しながら要素列を走査するイテレータを返します。一般式は以下のとおりです。

```
list#skip(n:number):map {block?}
iterator#skip(n:number):map {block?}
```

引数 n に除外する要素数を指定します。
例えば、[A, B, C, D, E, F, G] というリストを代入した変数 x があったとき、x.skip(2) を実行すると以下のような要素列を出力するイテレータを生成します。

```
A D G
```

13.5.5. 要素から nil 値を除外する

要素列から nil 値を除外するイテレータを返します。一般式は以下の通りです。

```
list#skipnil():map {block?}
iterator#skipnil():map {block?}
```

13.5.6. 条件に合致する要素を抽出する

リストまたはイテレータから、条件に合致する要素を抽出するイテレータを生成します。一般式は以下のとおりです。

```
list#filter(criteria) {block?}
iterator#filter(criteria) {block?}
```

criteria には関数またはイテレータを指定できます。
関数は、一つの引数をとり boolean 値を返すものを指定します。filter 関数はリストまたはイテレータの要素をひとつずつ関数に渡し、その戻り値が true のときその要素を抽出します。
criteria にイテレータを指定すると、filter 関数は抽出対象のリストまたはイテレータと同時に criteria のイテレータを走査し、これが true 値のときに要素を抽出します。

13.5.7. 条件に合致している間の要素を抽出する

リストまたはイテレータから、条件に合致している間の要素を抽出するイテレータを生成します。一般式は以下のとおりです。

```
list#while(criteria) {block?}
iterator#while(criteria) {block?}
```

criteria には関数またはイテレータを指定できます。
関数は、一つの引数をとり boolean 値を返すものを指定します。while 関数はリストまたはイテレータの要素をひとつずつ関数に渡し、その戻り値が true の間だけ要素を抽出します。false になったら処理を終了します。
criteria にイテレータを指定すると、while 関数は抽出対象のリストまたはイテレータと同時に criteria のイテレータを走査し、これが true 値の間だけ要素を抽出します。false になったら処理を終了します。

13.5.8. 条件に合致してからの要素を抽出する

リストまたはイテレータから、条件に合致した時点からの要素を抽出するイテレータを生成します。一般式は以下のとおりです。

```
list#since(criteria) {block?}
iterator#since(criteria) {block?}
```

criteria には関数またはイテレータを指定できます。
関数は、一つの引数をとり boolean 値を返すものを指定します。since 関数はリストまたはイテレータの要素

をひとつずつ関数に渡し、その戻り値が true になった時点で抽出を開始します。
criteria にイテレータを指定すると、since 関数は抽出対象のリストまたはイテレータと同時に criteria のイテレータを走査し、これが true 値になった時点で抽出を開始します。

13.6. 要素順序の操作

要素を走査する順番や、抽出方法を変えるメソッドが用意されています。

13.6.1. 要素列をくりかえし巡回する

リストまたはイテレータの要素を順に走査し、最後に到達したら再び最初に戻るイテレータを生成します。一般式は以下のとおりです。

```
list#round(n?:number) {block?}
iterator#round(n?:number) {block?}
```

引数 n に走査結果で得られる要素の数を指定します。この引数を省略すると、無限に走査をくりかえす無限イテレータになります。
例えば、[A, B, C] というリストを代入した変数 x があったとき、x.round(10) を実行すると以下のような要素列を出力するイテレータを生成します。

```
A B C A B C A B C A
```

13.6.2. 要素列をソートする

要素列をソートした結果を走査するイテレータを生成します。一般式は以下の通りです。

```
list#sort(directive?, keys[]?):[stable]
iterator#sort(directive?, keys[]?):[stable]
```

13.6.3. 要素列を逆に走査する

要素列を逆から走査するイテレータを生成します。一般式は以下のとおりです。

```
list#reverse() {block?}
iterator#reverse() {block?}
```

例えば、[A, B, C, D, E, F] というリストを代入した変数 x があったとき、x.reverse() を実行すると以下のような要素列を出力するイテレータを生成します。

```
F E D C B A
```

13.6.4. 要素列を左右に走査する

リストまたはイテレータの要素を順に走査し、最後に到達したら逆向きに走査、再び最初に戻ったら順に走査を繰り返すイテレータを生成します。一般式は以下のとおりです。

```
list#pingpong(n?:number):[sticky,sticky_l,sticky_r] {block?}
```

```
iterator#pingpong(n?:number):[sticky,sticky_l,sticky_r] {block?}
```

引数 n に走査結果で得られる要素の数を指定します。この引数を省略すると、無限に走査をくりかえす無限イテレータになります。アトリビュート:sticky, :sticky_l, :sticky_r は先頭または終端で折り返しをするときに要素を 2 度繰り返すか否かを指定します。:sticky_l が先頭要素、:sticky_r が終端要素、:sticky が両端の要素に対する指定になります。

例えば、[A, B, C, D] というリストを代入した変数 x があったとき、x.pingpong(10) を実行すると以下のような要素列を出力するイテレータを生成します。

```
A B C D C B A B C D
```

x.pingpong(10):sticky を実行すると以下のような要素列を出力するイテレータを生成します。

```
A B C D D C B A A B
```

13.6.5. 要素を指定した数ごとに折り返す

連続した要素を、指定個数分ずつリストに収めたものを要素として返すイテレータを生成します。一般式は以下の通りです。

```
list#fold(n:number, nstep?:number):[iteritem] {block?}
iterator#fold(n:number):[iteritem] {block?}
```

引数 n に、ひとつのリストに収める要素の数を指定します。実行すると、n 個の要素を抽出してリストに収めたものを要素として返し、つぎの走査では続きの n 個の要素を抽出して再びリストにして返します。これを要素がなくなるまで繰り返します。残りの要素数が n 個に満たない場合は、ある分だけリストに収めます。

引数 nstep を指定すると、次の要素抽出に移るときのオフセット移動量を n 個ではなく nstep 個ずつに変更します（現在 list#fold メソッドのみ対応)。

イテレータが返す要素はリストですが、:iteritem アトリビュートを指定するとこれをイテレータにすることができます。

例えば、[A, B, C, D, E, F, G, H, I, J] というリストを代入した変数 x があったとき、x.fold(3) を実行すると以下のような要素列を出力するイテレータを生成します。

```
[A, B, C] [D, E, F] [G, H, I] [J]
```

14. パス名の操作

14.1. Gura におけるパス名

一般にパス名というとファイルシステム上のファイルやディレクトリの名前を指すことが多いです。しかし、Gura における「パス」とは、データが格納されている場所を表す抽象的な名前を言います。この中には、ファイルシステム上の資源はもちろん、それ以外にもネットワーク資源の名前や、アーカイブファイルの中身なども含まれます。以下はいずれも Gura で扱えるパス名の例です。

```
/home/yamada/work
```

```
C:¥Windows¥Media¥chimes.wav
http://sourceforge.jp/
hoge.zip/foo/bar.txt
footool.tar.gz/src/main.c
```

最初の二つは、ファイルシステム上のファイルやディレクトリを表す例です。パス名をスクリプトに記述する際のセパレータは、スラッシュ "/" またはバックスラッシュ "¥" を指定します。スクリプトの内部処理で、現在動作している OS に応じて適切に変換が行われます。

三つ目の例は、HTTP プロトコルによる URI を表しています。

最後の二つは、アーカイブファイルの中のファイルを表すものです。

Gura は、こういったパス名で指し示される資源、すなわちファイルやディレクトリに対し、読み書きや要素サーチなどを行う仕組みを提供します。また、拡張性を持たせるため、パス名の解釈や実際の資源操作は、モジュールによって行われます。実際、上の例であげたパス名はそれぞれ `fs`, `http`, `zipfile`, `tar` モジュールで解釈・処理されます。

14.2. ディレクトリ操作

パス名が指すストレージやプロトコルがディレクトリサーチに対応していれば、ファイルの一覧や検索が可能になります。

14.2.1. パターン

ファイル名をサーチするのに、UNIX シェルで使われるパターンマッチングを使うことができます。使用できるワイルドカードは以下のとおりです。

パターン	説明
*	1文字以上の文字列
?	任意の1文字
[]	括弧内に含まれる文字のいずれか

14.2.2. 指定のディレクトリ内のサーチ

ディレクトリを表すパス名を指定し、含まれるファイルまたはディレクトリをサーチし、各要素のパス名を返します。一般式は以下のとおりです。

```
path.dir(pathname?:string, pattern*:string)
                                :map:flat:[stat,icase,file,dir] {block?}
```

引数 `pathname` はパス名です。引数 `pattern` には、ファイルまたはディレクトリのベース名に対するパターンを 0 個以上指定します。この引数を省略すると、すべてのファイルまたはディレクトリをサーチします。

アトリビュート `:stat` をつけるとパス名ではなく詳細情報を含んだ `stat` 型オブジェクトを返します。`:icase` は、パターンマッチングの際に大文字と小文字の区別をなくすアトリビュートです。`:file` や `:dir` をつけると、サーチ対象をそれぞれファイルまたはディレクトリに限定できます。

ブロック式をつけると、各サーチ結果ごとにブロックが繰り返し評価されます。このとき、ブロックには

`|pathname:string, idx:number|` という形式で引数が渡されます。`pathname` はサーチ結果のパス名、`idx` はループのインデクス番号です。

14.2.3. 再帰的なディレクトリサーチ

パス名で指定したディレクトリを基点として含まれるファイルまたはディレクトリを再帰的にサーチし、各要素のパス名を返します。一般式は以下のとおりです。

```
path.walk(pathname?:string, maxdepth?:number, pattern*:string)
                                    :map:flat:[stat,icase,file,dir] {block?}
```

引数 `pathname` はパス名です。`maxdepth` には、サーチするディレクトリの深さを指定します。`0` を指定すると基点のディレクトリのみのサーチとなり、これは `path.dir` の動作と同じになります。省略すると、深さの制限がなくなります。

引数 `pattern` には、ファイルまたはディレクトリのベース名に対するパターンを0個以上指定します。この引数を省略すると、すべてのファイルまたはディレクトリをサーチします。

アトリビュート `:stat` をつけるとパス名ではなく詳細情報を含んだ `stat` 型オブジェクトを返します。`:icase` は、パターンマッチングの際に大文字と小文字の区別をなくすアトリビュートです。`:file` や `:dir` をつけると、サーチ対象をそれぞれファイルまたはディレクトリに限定できます。

ブロック式をつけると、各サーチ結果ごとにブロックが繰り返し評価されます。このとき、ブロックには `|pathname:string, idx:number|` という形式で引数が渡されます。`pathname` はサーチ結果のパス名、`idx` はループのインデクス番号です。

14.2.4. パターンによるサーチ

パターンに適合するファイルやディレクトリをサーチします。一般式は以下の通りです。

```
path.glob(pattern:string):map:flat:[stat,icase,file,dir] {block?}
```

引数 `pattern` にパターンを指定します。このパターンはディレクトリ名を含むことができ、パス名の途中のディレクトリ名にもワイルドカードを使えます。

アトリビュート `:stat` をつけるとパス名ではなく詳細情報を含んだ `stat` 型オブジェクトを返します。`:icase` は、パターンマッチングの際に大文字と小文字の区別をなくすアトリビュートです。`:file` や `:dir` をつけると、サーチ対象をそれぞれファイルまたはディレクトリに限定できます。

ブロック式をつけると、各サーチ結果ごとにブロックが繰り返し評価されます。このとき、ブロックには `|pathname:string, idx:number|` という形式で引数が渡されます。`pathname` はサーチ結果のパス名、`idx` はループのインデクス番号です。

15. ストリーム

Gura では、ストレージ中などにあるファイルを「ストリーム」という抽象化されたインターフェースを使って読み書きします。この仕組みにより、データの実体が実際にどこに格納されているか、また、どのようなプロトコルでアクセスするかを言語が判断し、適切なモジュールを使って処理を行います。例えば、ストリームを使って以下のよう

なファイルにアクセスすることができます。

- ディスクストレージ中のファイル
- HTTP プロトコルで取得するファイル
- アーカイブファイル中のファイル

15.1. ストリームの生成

ストリームを生成する代表的な関数は open です。一般式は以下のとおりです。

```
open(name:string, mode:string => "r", encoding:string => "utf-8"):map {block?}
```

引数 name に、ストリームを表すパス名を指定します。引数 mode はアクセス方法の指定で、読み込みのとき "r"、書き込みには "w"、追加は "a" を指定します。
引数 encoding には、ストリームの内容をテキストデータとして入出力するときに使用する文字コーデックの名前を指定します。デフォルトでは utf-8 が使用されます。文字コーデックの実際の処理は、以下に示すモジュールが提供します。

codecs.basic モジュールが提供するコーデック
us-ascii, utf-8, utf-16, base64,

codecs.iso8859 モジュールが提供するコーデック
iso-8859-1, iso-8859-2, iso-8859-3, iso-8859-4, iso-8859-5, iso-8859-6
iso-8859-7, iso-8859-8, iso-8859-9, iso-8859-10, iso-8859-11, iso-8859-12
iso-8859-13., iso-8859-14, iso-8859-15, iso-8859-16

codecs.japanese モジュールが提供するコーデック
euc-jp, cp932, shift_jis, ms_kanji, jis, iso-2022-jp

モジュールを追加することで、新たな文字コーデックに対応させることができます。

また、ストリームを期待している変数や関数の引数に対して文字列を指定すると、ストリームに型変換して渡されます。以下は、型変換を使って open 関数と等価な処理を行う例です。

```
file_stream:stream = "hoge.txt"
```

15.2. 標準入出力

コンソールに対する入出力処理もストリームとして扱います。標準入力・標準出力・標準エラー出力に対応するストリームが、sys モジュールと os モジュールでそれぞれ以下の変数で定義されています。

標準入力	sys.stdin	os.stdin
標準出力	sys.stdout	os.stdout
標準エラー出力	sys.stderr	os.stderr

これらのストリームは以下の用途で使用します。

- 関数 print、println、printf は出力先のストリームとして sys.stdout を参照します。
- 関数 os.exec は、起動した外部プロセスの標準出力の内容を os.stdout に、標準エラー出力の内容を os.stderr に出力します。

標準入出力を設定する変数の内容をほかのストリームインスタンスで置き換えると、入出力がそのストリームに切り替わります。これは、外部プロセスの出力内容をとりこむときなどに便利です。以下は、外部プロセスの標準出力の内容を binary 型のバッファにとりこむ例です。

```
buff = b""
os.stdout = buff.stream()
os.exec("program")
```

15.3. テキストアクセスとバイナリアクセス

ストリームで扱うデータは単なるバイト列です。この意味で言うとストリームにおけるデフォルトのアクセスフォーマットはバイナリデータであると考えることができます。ストリームで扱うデータをバイナリデータとして扱うか、テキストデータとして扱うかは、ストリームを操作するメソッドによって決まります。open 関数に渡すエンコーディング指定は、テキストアクセスのメソッドのみで有効になり、バイナリアクセスのメソッドには影響しません。つまり、ストリームをバイナリとして扱うことが分かっていれば、open 関数のエンコーディング指定は気にする必要はありません。

ストリームの内容をバイナリデータとして扱うメソッドには以下のものがあります。

```
stream#read(len?:number)
stream#write(buff:binary):reduce
stream#seek(offset:number, origin?:symbol):reduce
stream#tell()
stream#readto(stream:stream):map:reduce
stream#writefrom(stream:stream):map:reduce
stream#compare(stream:stream):map
```

ストリームの内容をテキストデータとして扱うメソッドには以下のものがあります。

```
stream#print(values*):map:void
stream#println(values*):map:void
stream#printf(format:string, values*):map:void
stream#readtext()
stream#readline():[chop]
stream#readlines(nlines:number):[chop] {block?}
```

その他にも、引数としてストリームを受け取る関数やメソッドがあり、それぞれストリームデータの扱いが異なります。例えば、JPEG ファイルの読み書きならばバイナリデータとして扱いますし、CSV ファイルならばテキストファイルとして見るでしょう。

15.4. さまざまなストリーム読み込み

Gura はほとんどのデータ入出力をストリームとして扱います。

これは、スクリプトファイルそのものも例外ではありません。例えば、ZIP アーカイブの中にあるスクリプトファイルを、展開することなく以下のように直接実行することができます。

```
gura –i zipfile archive.zip/hello.az
```

"-i *module*" は、スクリプト実行前にモジュールをあらかじめインポートするコマンドライン引数です。ここでは、モジュール zipfile をインポートして ZIP アーカイブのアクセスを可能にしています。

HTTP サーバ上にあるスクリプトファイルも、以下のように実行できます。

```
gura –i net.http http://aaa.bbb.ccc/hello.az
```

16. イメージ

かつてのコンピュータ操作はテキストのやりとりが中心でしたが、今ではグラフィカルユーザインターフェースによるものに完全に移行しました。また、Web ブラウザを中心としたインターネットアクセスをぬきにしては今日のプログラミング技術は語れません。そのような中にあって最も重要な位置を占めるのが、グラフィックイメージ（以下、単にイメージと呼びます）の操作です。

イメージ操作には、ファイル入出力・フィルタ処理・描画・画面表示などがあり、それぞれの処理においてライブラリが発表されています。このため、ライブラリの処理の間でイメージデータのやりとりをすることは頻繁に行われることの一つです。イメージデータは赤・緑・青の三原色と、透明度をあらわすアルファ値で表現され、これらの要素を順番にメモリの中に格納するフォーマットが一般にとられます。このとき、ライブラリによって格納するバイト順やアラインメントが異なっているために、変換処理などの煩雑な手続きが必要な場合が少なくありません。

そこで Gura は、イメージデータを言語の標準的なデータ型と位置づけ、イメージを操作するモジュール群はこのデータ型を中心に実装する方針をとりました。これにより、いろいろな処理をするライブラリ・モジュール間のデータ交換を自然な形で実装することができるようになります。

例えば標準の Tcl/Tk ライブラリで扱えるイメージフォーマットは GIF と PPM だけです。しかし、Gura に組み込まれた `tk` モジュールは Gura のイメージ型を扱うように実装されているため、Gura のモジュールが対応している PNG や JPEG などのイメージも Tk のキャンバスに表示できます。また、Cairo や OpenGL などのグラフィック描画ライブラリの描画対象をイメージに切り替えることができるので、イメージへの重ね描きをしたり、描画結果を任意のイメージフォーマットで出力したりすることができます。

16.1. ファイルからの読み込み

イメージファイルからイメージを読み込むときは、`image` 関数を以下の形式で呼び出します。

```
image(stream:stream, format?:symbol, imgtype?:string) {block?}
```

引数 `stream` は、イメージファイルを読み込むストリームです。

`format` はイメージインスタンスのデータ内部表現で、`RGB` 要素のみを持つ`rgb` かアルファ要素も含む`rgba` を指定します。省略すると、`rgba` が使われます。

imgtype には、"jpeg" や"png" というようにイメージタイプ名を文字列で指定します。この引数が省略されると、イメージファイルのヘッダ情報やファイル名のサフィックスからイメージタイプを識別します。

イメージファイルの読み込みをするには、対応するモジュールをあらかじめインポートしておく必要があります。モジュールとサポートするイメージファイルは以下のとおりです。

モジュール名	サポートするイメージファイル	イメージタイプ名
bmp	Windows BMPファイル	bmp
msicon	Windowsアイコンファイル	msicon
ppm	PPMファイル	ppm
jpeg	JPEGファイル	jpeg
png	PNGファイル	png
gif	GIFファイル	gif

モジュールを新規に開発することで、新しいイメージタイプに対応させることが可能です。

イメージタイプによっては、アニメーション GIF のように複数のイメージデータをひとつのファイルに格納していたり、イメージ特有のプロパティデータを持っているものがあります。これらの情報は、各モジュールが提供する関数やクラスで操作することができます。詳細は、モジュールのリファレンスを参照してください。

ブロック式をつけると、|img:image| という形式でブロック引数を渡してブロックを評価します。img は生成したイメージのインスタンスです。

16.2. ブランクイメージを生成する

以下の形式で image 関数を呼び出すと、ブランクのイメージインスタンスを生成します。

```
image(format:symbol, width:number, height::number, color?:color) {block?}
```

format はイメージインスタンスのデータ内部表現で、RGB 要素のみを持つ`rgb かアルファ要素も含む`rgba を指定します。省略すると、`rgba が使われます。

width、height にはイメージの幅と高さをそれぞれピクセル単位で指定します。

color は生成時に塗りつぶす色指定です。省略すると黒になります。

16.3. ファイルへの書き込み

ファイルへの書き込みを行うメソッド image#write が用意されています。一般式は以下のとおりです。

```
image#write(stream:stream, imgtype?:string):map:reduce
```

引数 stream は、イメージファイルを書き込むストリームです。

imgtype には、"jpeg" や"png" というようにイメージタイプ名を文字列で指定します。この引数が省略されると、イメージファイルのヘッダ情報やファイル名のサフィックスからイメージタイプを識別します。

イメージファイルの読み込みをするには、対応するモジュールをあらかじめインポートしておく必要があります。モジュールとサポートするイメージファイルは、「ファイルからの読み込み」の節を参照ください。

このメソッドは、一つの画像データのみの書き込みに対応しています。しかしイメージタイプによっては、アニメーション GIF のように複数のイメージデータをひとつのファイルに格納していたり、イメージ特有のプロパティデ

ータを持っているものがあります。こういったファイルの書き込みは、各モジュールが提供する関数やクラスを使うことで可能になります。詳細は、モジュールのリファレンスを参照してください。

16.4. イメージへの描画

二次元グラフィックライブラリ Cairo や三次元グラフィックライブラリ OpenGL のモジュールが用意されています。詳細は各モジュールのリファレンスを参照してください。

16.5. 画面表示

GUI を提供する Tcl/Tk や、高速な画面表示を可能にする SDL (Simple Direct Layer) のモジュールが用意されています。詳細は各モジュールのリファレンスを参照してください。

16.6. イメージ加工

イメージクラスには、回転や拡大・縮小を行うメソッドが用意されています。いずれも、もとのイメージデータは改変せず、新たなイメージインスタンスを生成して返します。

16.6.1. 反転

左右または上下にイメージを反転したイメージを生成して返します。一般式は以下の通りです。

```
image#flip(orient::symbol):map
```

引数 `orient` にシンボル `` `horz `` を指定すると左右反転、`` `vert `` を指定すると上下反転になります。`` `both `` を指定すると、左右および上下反転をします。これは時計周りに 180 度回転させたことと同じです。

16.6.2. 回転

指定の角度だけ回転したイメージを生成して返します。一般式は以下の通りです。

```
image#rotate(rotate:number, background?:color):map
```

引数 `rotate` に回転する角度を degree 単位で指定します。正の数は時計まわり、負の数は反時計回りを意味します。引数 `background` は、回転したときにできる余白を、塗りつぶす色を指定します。省略すると黒で塗りつぶします。

16.6.3. 拡大・縮小

指定の大きさに拡大または縮小したイメージを生成して返します。一般式は以下の通りです。

```
image#resize(width?:number, height?:number):map:[box]
```

`width` および `height` にリサイズ結果の大きさを指定します。どちらかを省略した場合、オリジナルの縦横比率を保つようにリサイズされます。アトリビュート `:box` を指定して、`width` のみを指定すると、縦横がいずれも `width` の正方形が指定されます。

16.6.4. サムネイル作成

指定の範囲に収まるようリサイズしたイメージを生成して返します。指定した範囲よりもイメージが小さい場合、元のイメージへの参照をそのまま返します。一般式は以下の通りです。

```
image#thumbnail(width?:number, height?:number):map:[box]
```

`width` および `height` にリサイズ結果の大きさを指定します。どちらかを省略した場合、オリジナルの縦横比率を保つようにリサイズされます。アトリビュート`:box` を指定して、`width` のみを指定すると、縦横がいずれも`width` の正方形が指定されます。

16.6.5. 部分抽出

指定の部分を抽出したイメージを返します。一般式は以下の通りです。

```
image#crop(x:number, y:number, width?:number, height?:number):map
```

引数 `x`、`y`、`width`、`height` に抽出する範囲を指定します。`width`、`height` を省略すると、イメージの右端および下端までの範囲を抽出します。