Gura 言語マニュアル

Updated: November 23, 2013

Official site: http://www.gura-lang.org/

序文

リストなどで表現される複数のデータに対してある演算や変換操作を施し、結果を別の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。これ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。

このアイデアを実際に動くものにするため、当初は既存のスクリプト言語を拡張する形を模索していました。しかし、「暗黙的マッピング」は一部の関数のみに働くというのではその真価を発揮しません。数値演算はもちろん、文字列操作やイメージ描画、GUI やネットワークアクセスにいたるまで全ての処理にわたって使えるようになり、プログラミングスタイルの中に溶け込んで初めてその存在意義を有します。つまり、既にあるものの拡張ではなく、まったく新しい世界の創出が必要だったのです。そのためにスクリプト言語をスクラッチから作成することにしました。

新たなスクリプト言語を創生するにあたり、留意したのは以下の点です。

なじみのある文法を取り入れること

私は、既存の言語と同じことをするのなら、いたずらに新しい文法を作る必要はないと考えています。文法や記号の割り振り方を決める際も、できる限りなじみのある言語にあわせるようにしました。ブレース記号でブロックを作るようにしたので、スクリプトを書いた時の全体の雰囲気はJavaかJavaScriptに似ているように見えるかもしれません。モジュールによる名前空間の扱い方や関数の命名などは Python から影響を受けています。

手軽に使えて実用的であること

プログラミング言語は、身の回りにある課題を解決することができて初めて存在意義を持ちます。そして、課題を解決するために多くの手間が必要になるようでは実用的とはいえません。そこで、グラフィカルユーザインターフェースや各種ファイル処理などを言語本体に付属させ、ごく普通のユーザが実現したいことを即座にプログラミングできる環境を提供します。

このような方針のもと、すべての関数やメソッドに「暗黙的マッピング」のポリシーをいきわたらせたたスクリプト言

語 Gura を開発し、2011 年 3 月 15 日に最初のバージョンを SourceForge.JP にプロジェクト登録しました。

プログラミング言語というものの不思議なところは、言語を作成した本人が、即座にその言語におけるエキスパートになっているわけではないことです。これは例えば、あるゲームを考案したとして、ルールは自分で考えたものであっても、そのルールのもとで勝利に導くコツや定石を知るには、実際に競技をして慣れていかなければいけないのと同じです。私の場合も「暗黙的マッピング」という今までにないルールを伴った Gura プログラミングを知るには、自分で実際に多くのスクリプトを作成して試す必要がありました。そして、この言語を使ってみて実感したことは、「暗黙的マッピング」をはじめとする Gura のさまざまなフィーチャーはプログラミングの現場において大きな有効性を持っているということです。

一ユーザとして、私はこのスクリプト言語をあなたにお勧めします。

著者

2012 年 6 月 26 日

目次

1. このリファレンスについて 8
2. 実行方法 9
2.1. 実行ファイル 9
2.2. 対話モード 9
2.3. スクリプトファイル 9
2.4. コンポジットファイル 10
2.5. コマンドラインオプション 10
2.6. テンプレートエンジン 10
3. スクリプトの構成要素 12
3.1. 数値リテラル 12
3.2. 文字列リテラル 12
3.3. バイナリリテラル 13
3.4. 識別子 13
3.5. リスト 14
3.6. イテレータ 14
3.7. マトリクス 15
3.8. ブロック 15
3.9. 辞書 16
3.10. Quoted 値 16
3.11. シンボル値 16
3.12. 関数 16
3.13. アトリビュート 17
3.14. 演算子 17
3.15. コメント 17
3.15.1. ラインコメントとブロックコメント 17
3.15.2. マジックコメント 17
4. クラスとインスタンス 19
4.1. 概要 19
4.2. メンバアクセス 19
4.3. 定義基本データ型 19
4.4. オブジェクト型 20
5. 演算子 21
5.1. 組み込み演算子 21
5.2. 論理演算について 24
5.3. 文字列フォーマット 24
5.4. 代入演算子 25

5.4.1. シンボルへの代入 25
5.4.2. インデクスアクセスによる代入 25
5.4.3. 関数の代入 26
5.4.4. 複数シンボルへの一括代入 26
5.5. 演算子のオーバーロード 27
6. 関数 28
6.1. 関数の呼び出し 28
6.1.1. 構成要素 28
6.1.2. 関数インスタンス 28
6.1.3. 引数指定 29
6.1.4. 引数のリスト展開 30
6.1.5. **名前つき引数指定と引数の辞書展開** 30
6.1.6. アトリビュート指定 31
6.1.7. ブロック指定 31
6.1.8. 引数リストの省略 32
6.1.9. スコープ 32
6.1.10. **レキシカルスコープとダイナミックスコープ** 33
6.1.11. **ブロック式とスコープ** 35
6.2. 関数バインダ 36
6.3. 関数定義 37
6.3.1. 構成要素 37
6.3.2. 関数名 37
6.3.3. 引数定義リスト 38
6.3.4. 関数のアトリビュート定義 38
6.3.5. ブロック定義 39
6.3.6. ヘルプ文字列 41
6.4. 関数定義の例 41
6.5. 関数の戻り値 42
6.6. 暗黙的マッピング 42
6.7. 関数呼び出しの連結関係 43
6.8. 名前なし関数 44
7. 制御構文 45
7.1. 条件分岐 45
7.2. 繰り返し 45
7.2.1. `repeat` 関数 45
7.2.2. `while` 関数 45
7.2.3. `for` 関数 46
7.2.4. `cross` 関数 46
7.2.5. 繰り返し中のフロー制御 47

7.2.6. 繰り返し関数によるリストの生成 47
7.2.7. 繰り返し関数によるイテレータの生成 48
7.3. 例外処理 48
8. 暗黙的マッピング 49
8.1. 実装のきっかけ 49
8.2. コンセプト 49
8.3. 適用ルール 50
8.4. ケーススタディ 51
8.4.1. 演算子と暗黙的マッピング 51
8.4.2. 文字列出力との組み合わせ 51
8.4.3. ファイル入力との組み合わせ 51
8.4.4. パターンマッチングとの組み合わせ 52
9. メンバマッピング 53
9.1. ケーススタディ 53
10. ユーザ定義クラス 55
10.1. class 関数 55
10.2. 基本的なクラス定義 55
10.3. コンストラクタ関数についての詳細 56
10.4. クラスメソッドとインスタンスメソッド 56
10.5. メンバアクセス権 57
10.6. 継承 58
10.7. 特別なメソッド 58
10.8. 構造体のユーザ定義 60
10.9. 既存のクラスへのメソッド追加 60
11. モジュール 61
12. リストとイテレータ 63
12.1. 概要 63
12.2. 有限イテレータと無限イテレータ 63
12.3. イテレータ操作とブロック式 63
12.4. リストの生成 63
12.5. 要素操作ダイジェスト 64
12.6. ユーザ定義イテレータ 64
12.6.1. 繰り返し関数によるイテレータ定義 65
12.7. 汎用イテレータ関数によるイテレータ定義 66
13. 数学に関する機能 68
13.1. 複素数計算 68
13.2. 統計処理 68
13.3. 順列 68
13.4. 行列演算 68

13.5. 式の微分演算 68
14. パス名の操作 70
14.1. Gura におけるパス名 70
14.1.1. ファイルシステム内のパス 70
14.1.2. インターネットの URI パス 70
14.1.3. アーカイブファイル内のパス 70
14.2. ディレクトリ操作 71
15. ストリーム 72
15.1. 概要 72
15.2. ストリームの種類 72
15.3. ストリームの生成 73
15.4. コーデックの指定 73
15.5. 標準入出力 73
15.6. プロセス実行と標準入出力 74
15.7. テキストアクセスとバイナリアクセス 74
15.8. ストリーム間のデータコピー 75
15.9. スクリプトファイルの実行 75
15.9.1. アーカイブ中のスクリプトファイル 75
15.9.2. HTTP 上のスクリプトファイル 76
16. イメージ 77
16.1. 概要 77
16.2. ブランクイメージを生成する 77
16.3. ストリームからのイメージデータ読み込み 78
16.4. ストリームへのイメージデータ書き込み 78
16.5. イメージ加工 79
16.6. グラフィック描画 79
16.7. ディスプレイ出力 79
17. テンプレートエンジン 80

1. このリファレンスについて

本リファレンスはスクリプト言語 Gura の実行方法や文法、基本データ型、データ処理機構について説明したものです。実装している関数やメソッドの詳細や、同梱されているモジュールの説明は「**Gura ライブラリリファレンス**」を参照してください。

2. 実行方法

2.1. 実行ファイル

Gura の Windows 用実行ファイルは gura.exe と guraw.exe、Linux 用実行ファイルは gura です。
Windows 用の guraw.exe は、コマンドプロンプトを出さない実行ファイルです。wxWidgets や Tcl/Tk、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。

2.2. 対話モード

gura.exe または gura をスクリプトファイルを指定せずに実行すると、以下のような出力の後で対話モードに入り、ユーザからの入力待ちになります。

```
Gura 0.4.0 [MSC v.1600, May 16 2013] copyright (c) 2011- Y.Saito
>>>
```

プロンプト >>> に続いて Gura のスクリプトと改行を入力すると、スクリプトを実行してその結果を表示します。

```
>>> 3 + 4
7
>>> println('Hello world')
Hello world
```

対話モードを抜けるときは、キーボードで Ctrl+C を入力するか、スクリプト `sys.exit()` を実行してください。

2.3. スクリプトファイル

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 .gura または .guraw です。これらは Windows 環境でそれぞれ gura.exe と guraw.exe に関連づけられています。
スクリプトファイルには、何の宣言もなく先頭から Gura のプログラムを記述することができます。しかし、Linux など UNIX 系の OS で実行するならば、shebang を先頭行に記述しておくと便利です。以下は、Windows、Linux 環境ともに動作する Hello world スクリプトです。

```
#!/usr/bin/env gura
println('Hello world')
```

OS のシェルに shebang を認識させるには、ファイルの改行コードが LF になっている必要があります。ファイルの改行コードが CR-LF となっていると、シェルがエラーを出力します。
スクリプトファイル中に ASCII 文字でない日本語などの文字を含む場合は、UTF-8 文字コードでファイルを保存してください。その他の文字コードで記述する場合は、コマンドラインから -d オプションで文字コードを指定するか、以下のようにマジックコメントを記述します。

```
#!/usr/bin/env gura
# coding: shift_jis
println('こんにちは、世界')
```

2.4. コンポジットファイル

コンポジットファイルは Gura スクリプトファイルやその他のファイルを zip 形式で圧縮したもので、サフィックスには.gurc または.gurcw を付与します。コンポジットファイルを使うと、複数のスクリプトファイルや、イメージファイルを必要とするプログラムを単一のファイルにまとめて扱えるので、配布するときに便利です。

コンポジットファイルは ZIP 形式のファイルを扱う任意のツールを使って作成することができます。ただし、多くのツールはファイルのサフィックスによってファイル形式を決定するので、いったんサフィックスを .zip にしてファイルを作成し、その後ファイル名を変更します。以下は、ツールに 7z を使った作成例です。

```
> 7z a hoge.zip hoge.gura image1.png image2.png
> ren hoge.zip hoge.gurc
```

Gura パッケージに同梱されている gurcbuild モジュールを使えば、外部プログラムを使わずにコンポジットファイルを作成することができます。以下は、上記と同じファイル構成を持ったコンポジットファイルを作成する Gura スクリプトです。

```
#!/usr/bin/env gura
import(gurcbuild)
gurcbuild.build(['hoge.gura', 'image1.png', 'image2.png'])
```

この方法で作成したコンポジットファイルは、ファイルの先頭に shebang が追加され、実行属性がつくので、UNIX 系 OS で実行可能ファイルとして扱うことができます。

2.5. コマンドラインオプション

以下のコマンドライン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などを評価した後、対話モードに入ります。
-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。モジュールは複数指定することができます。これは、スクリプト中で import 関数を実行したのと同じ効果を持ちます。
-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。
-d encoding	スクリプトのエンコーディングを指定します。デフォルトはUTF-8です。
-T template	テンプレートファイルに埋め込まれたスクリプトを評価し、標準出力に結果を出力します。
-v	バージョン番号を表示します。

2.6. テンプレートエンジン

コマンドラインオプション -T で、テンプレートファイルを指定することができます。テンプレートファイルとは、通常のテキスト文書の中に Gura のスクリプトが埋め込まれたファイルで、動的にコンテンツを変化させるときなど

に使用します。テンプレートファイルを評価すると、埋め込まれた Gura のスクリプトを評価し、その結果を標準出力に出力します。

3. スクリプトの構成要素

3.1. 数値リテラル

数値リテラルには、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。虚数 j を表すときは "1j" と記述してください。"j" という表記は変数などのシンボル名と解釈されます。同じ理由で –j という数値は "-1j" と記述します。

以下は有効な数値リテラル表記例です。

```
43    3.1415    23e5      32j    1j     -3j
```

3.2. 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。

文字列リテラル内ではバックスラッシュ "\" に続けて以下のエスケープ文字を使うことができます。

エスケープ文字	説明
\\	バッククオート
\'	シングルクオーテーション
\"	ダブルクオーテーション
\a	ベル
\b	バックスペース
\f	改ページ
\r	キャレッジリターン
\n	改行
\t	タブ
\v	垂直タブ
\0	Null文字
\x*hh*	16進数*hh* をキャラクタコードとして持つ文字

文字列をシングルクオーテーションで囲むと、文字列中にシングルクオーテーションが現れる場合、これをエスケープ文字にする必要があります。また、ダブルクオーテーションで囲むと、文字列中のダブルクオーテーションをエスケープ文字にしなければいけません。それ以外に両者の間で違いはありませんが、Gura スクリプトの記述ではシングルクオーテーションの使用を推奨します。

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。以下のように、複数行にわたる文字列の表記が可能になります。

```
'''ABCD
EFGH
IJKL
'''
```

最初の行をそろえたい場合は、以下のようにクオーテーション直後の改行コードをエスケープして記述します。

```
'''\
ABCD
EFGH
IJKL
'''
```

文字列の前にプレフィックス "r" または "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。

プレフィックス "r" と "R" の違いは、複数行にわたる文字列表記において、最初に現れる改行コードの扱いにあります。以下のようにプレフィックス "r" を指定すると:

```
r'''
ABCD
EFGH
IJKL
'''
```

これは文字列 '\nABCD\nEFGH\nIJKL\n' として評価されます。それに対し、以下のようにプレフィックス "R" を指定すると、文字列 'ABCD\nEFGH\nIJKL\n' のように最初の改行コードが取り除かれます。

```
R'''
ABCD
EFGH
IJKL
'''
```

複数行文字列表記は、通常のテキストを文字列としてスクリプト中に埋め込む用途を想定しているので、プレフィックス "R" 指定の表記を使うとより自然にテキストを扱えます。

3.3. 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。

3.4. 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、日本語などをふくんだ識別子も使用することができます。

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。

3.5. 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。またリストの要素がリストになっている、入れ子の状態の場合、内部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。ブレース記号を使うと、上のリスト表記を以下のように書くことができます。

```
@{}
@{1, 2, 3, 4, 5}
@{'Hello', 2, 3, 'World'}
@{{1, 2, 3}, 4, 5, {6, 7, {8, 9, 10}}}
```

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
```

このとき、無限イテレータが要素に含まれるとエラーになります。

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。

3.6. イテレータ

括弧記号 "(" および ")" で囲んだ領域はイテレータになります。イテレータは iterator クラスのインスタンスです。イテレータ中には 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。

イテレータを評価すると、内部の要素を順に評価し、その結果を要素としてもつイテレータオブジェクトを返します。以下は有効なイテレータ表記の例です。

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

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
```

要素中に無限イテレータが含まれていてもエラーにはなりません。

要素がひとつだけの場合は要素の後ろにカンマ "," をつけます。たとえば 3 という数値を要素に持つイテレータを表記しようとして "(3)" と書くと、これはイテレータにはならず単なる数値リテラルとして扱われてしまいます。これを "(3,)" と記述することでイテレータになります。

3.7. 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)}}
```

3.8. 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したりする際に使われます。

3.9. 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。

```
%{key => value, key => vakue, ..}
```

2. キーと値を角括弧 "[" と "]" またはブレース記号 "{" および "}" で囲みます。

```
%{[key, value], [key, value], ..}
%{{key, value}, {key, value}, ..}
```

3. キーと値を一次元的に交互にならべて表記します。

```
%{key, value, key, value, ..}
```

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。

3.10. 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。

3.11. 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。

3.12. 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、その場合でも中身のない括弧を記述し、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。

3.13. 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。

3.14. 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。

3.15. コメント

3.15.1. 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 */ */ */
```

3.15.2. 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。

以下は shebang とマジックコメントを指定したファイルの先頭の記述例です。

```
#!/usr/bin/env gura
# coding: shift_jis
```

Emacs の文字コード指定としても認識させたい場合は以下のように記述します。

```
#!/usr/bin/env gura
# -*- coding: shift_jis -*-
```

4. クラスとインスタンス

4.1. 概要

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。

4.2. 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 this 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 ', this) }
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 ', this) }
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。

4.3. 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

4.4. 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

5. 演算子

5.1. 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?	x が false または nil の場合はfalse、それ以外の場合はtrueに変換します。
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。 fraction + fraction 和を fraction 型で返します。 fraction + number 和を fraction 型で返します。 number + fraction 和を fraction 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。
	fraction – fraction	差を fraction 型で返します。
	fraction – number	差を fraction 型で返します。
	number – fraction	差を fraction 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。
	fraction * fraction	積を fraction 型で返します。
	fraction * number	積を fraction 型で返します。
	number + fraction	積を fraction 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 timedelta 型で返します。
	number * timedelta	積を timedelta 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。
	fraction / fraction	除算結果を fraction 型で返します。
	fraction / number	除算結果を fraction 型で返します。
	number / fraction	除算結果を fraction 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。 X が nil のとき、y の値を返します。また、y が nil のとき、x の値を返します。 それ以外の型の値を渡すとエラーになります。
x & 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。 x または y が nil のとき、nil 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。

5.2. 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。

ある変数の値が nil 以外の有効値であるか確認する場合は注意が必要です。たとえば、変数 x が list インスタンスか nil 値をとる可能性があり、list インスタンスの場合のみある処理をさせるために以下のようなコードを書いたとします。

```
x = [1, 2, 3]
x && println('x is a valid value')       // NG
```

これですと暗黙的マッピングが働いてしまい、x の各要素ごとに && 演算子が適用されるので、期待した動作になりません。この例の場合は、以下のように ? 演算子を使って boolean 値に変換してから真偽の判断を行います。

```
x = [1, 2, 3]
x? && println('x is a valid value')      // OK
```

5.3. 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 0x20）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。

5.4. 代入演算子

5.4.1. シンボルへの代入

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。

5.4.2. インデクスアクセスによる代入

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。インデクスとして扱えるデータ型は、インスタンスの種類によって異なります。list インスタンスは数値のみを扱い、他の型が指定されるとエラーになります。dict インスタンスは string、number または symbol 型のオブジェクトをインデクスに指定できます。

インデクスは、複数指定することができます。

```
obj[5, 6, 7, 8, 9] = 10
```

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。上の例は以下のように記述することができます。

```
obj[5..9] = 10
```

5.4.3. 関数の代入

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。

5.4.4. 複数シンボルへの一括代入

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。以下の例は、変数 a, b, c にそれぞれ 1, 2, 3 を定義します。

```
[a, b, c] = [1, 2, 3]
```

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よりも多いと、エラーになります。

定義する値としてイテレータを指定することもできます。上の例は以下のように記述することができます。

```
[a, b, c] = 1..3
```

代入対象の数がイテレータ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、代入対象の数がイテレータ要素よりも多いと、エラーになります。

定義する値に無限イテレータを指定すると、代入対象の数だけ代入処理を行います。例を以下に示します。

```
[a, b, c] = 1..
```

この書式は、C 言語の enum 宣言のように使うことができます。

定義する値がリストまたはイテレータ以外の場合、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、以下の例は変数 a, b, c にそれぞれ 3 を定義します。

```
[a, b, c] = 10
```

5.5. 演算子のオーバーロード

operator クラスのメソッド assign()を使うと、演算子の処理内容を追加または上書きすることができます。このメソッドは以下のように実行します。

- 単項演算子の定義: operator(op).assign(type) {|value| …}
- 二項演算子の定義: operator(op).assign(type_l, type_r) {|value_l, value_r| …}

引数 op は、演算子シンボルをバッククオート ` でシンボル化したものを指定します。引数 type、type_l および type_r は、型名シンボルの先頭にバッククオート ` をつけたものを指定します。ブロック内には演算子が評価されたときに実行する内容を記述します。このとき、演算対象の値がブロックパラメータとして渡されます。ブロックの最終的な評価値をその演算子の結果として扱います。

以下は、単項演算子 - に文字列を与えたとき、順序を逆にした文字列を返すようにする例です。

```
operator(`-).assign(`string) {|x| x[-(1..)].join()}
```

実行例を以下に示します。

```
>>> -'hello world'
'dlrow olleh'
```

6. 関数

6.1. 関数の呼び出し

6.1.1. 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。

6.1.2. 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。

6.1.3. 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。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、その場合はイテレータからリスト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、その引数はオプショナル引数として扱われ、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()
```

オプショナル引数には nil 値を与えることもできます。この記述は、オプショナル引数の後に続く引数に値を指定したい場合に便利です。

```
func(nil, 2, 3)
```

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。

6.1.4. 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。

6.1.5. 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。

6.1.6. 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。

6.1.7. 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。

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|}
```

6.1.8. 引数リストの省略

ブロック式をとる関数呼び出しの場合、引数が必要ないときは引数リストの記述を省略することができます。関数 repeat は、引数に繰り返し回数を指定しますが、これ以下のように引数を省略して実行すると無限ループになります。

```
repeat () {
    // some process
}
```

このとき、引数の括弧をとりのぞいて以下のように記述することができます。

```
repeat {
    // some process
}
```

アトリビュート :symbol_func をつけて定義された関数は、関数シンボルを指定するだけでその関数を評価します。このようなふるまいをする関数には、return, break, continue があります。

6.1.9. 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。

6.1.10. 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。

6.1.11. 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)
```

あるスコープ内で、変数にいったんローカル変数として代入操作をすると、以降はアトリビュート:local をつけなくてもローカル変数として扱われます。

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。

6.2. 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') 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')
```

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。

6.3. 関数定義

6.3.1. 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
- ヘルプ文字列

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。

6.3.2. 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。

6.3.3. 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。

引数定義に型指定がなければ、任意の型の値を受け取ることができます。これには nil 値も含まれます。型指定がされると nil 値はエラーになりますが、ケースによっては無効値として引数に渡したいことがあります。そのような場合は引数のアトリビュートに ":nil" を指定します。

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。

6.3.4. 関数のアトリビュート定義

関数定義の引数リストの後、コロンに続いて角カッコでシンボルのリストを列記すると、その関数がオプションでうけとるアトリビュートの定義になります。

```
f():[a,b,c] = {
    if (__args__.isset(`a)) { … }
    if (__args__.isset(`b)) { … }
    if (__args__.isset(`c)) { … }
}
```

関数のアトリビュート指定は、主に関数のふるまいを静的に決定したいときに使います。

6.3.5. 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。

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)
}
```

6.3.6. ヘルプ文字列

関数インスタンスのプロパティ help にヘルプ文字列を登録することができます。

```
f() = {
    println('Hello, World!')
}
f.help = R'''
This function just prints out Hello, World.
'''
```

6.4. 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)
```

6.5. 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、引数が必要ない場合、引数リストの括弧を省略して実行することができます。

6.6. 暗黙的マッピング

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、this 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 this 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 [this, this, this] 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 [this, this, this] ではなく this 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。

6.7. 関数呼び出しの連結関係

ブロックの終端ブレース '}' 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-catch 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レース '}' 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。

6.8. 名前なし関数

関数 function を使うと、名前なし関数を生成することができます。関数 function の一般式は以下の通りです。

```
function(`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、function 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)}
function (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) = {
    function() { n += 1}
}
cnt = new_counter(2)
printf('%d\n', cnt())
printf('%d\n', cnt())
printf('%d\n', cnt())
```

7. 制御構文

Gura の実行要素はすべて関数であり、制御構文という特別な要素は存在しません。しかし、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。ブロックを使っているので、外見は Java や C++などと差異が見つからないかもしれません。

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。

7.1. 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}
```

7.2. 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。

7.2.1. `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、引数で指定した回数だけ `block` の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。

ブロックを評価するとき、ブロックパラメータを `|idx:number|` という形式で渡します。`idx` は 0 から始まるループカウンタです。

7.2.2. `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、引数で指定した式が条件を満たす間だけ block の処理を繰り返します。
ブロックを評価するとき、ブロックパラメータを |idx:number| という形式で渡します。idx は 0 から始まるループカウンタです。

7.2.3. 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 block の処理を繰り返します。
ブロックを評価するとき、ブロックパラメータを |idx:number| という形式で渡します。idx は 0 から始まるループカウンタです。
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。
イテレータ要素をひとつずつ評価するには、イテレータの each メソッドを使う方法もあります。以下に for 関数を使った場合と、イテレータの each メソッドを使った場合の例を示します。

```
tbl = [1, 2, 3, 4, 5]
for (x in tbl) { … }
tbl.each() {|x| … }
```

イテレータの each メソッドを使った記述の方が簡潔にかけることが多いですが、二つ以上のイテレータを同時に評価したりする場合は for 関数を使うと便利です。

7.2.4. 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 block の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のと同じです。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。

ブロックを評価するとき、ブロックパラメータを |idx:number, i0:number, i1:number, …| という形式で渡します。idx は 0 から始まるループカウンタで、それに続く i0、i1、… は指定したイテレータそれぞれのインデクス値です。

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
```

7.2.5. 繰り返し中のフロー制御

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、繰り返し関数の戻り値は 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。

7.2.6. 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。

7.2.7. 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ブロック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生成します。

この機能を使って、ユーザ定義のイテレータを作成することができます。詳細は「イテレータ」の章を参照ください。

7.3. 例外処理

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

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

ひとつの try に対して 1 個以上の任意の数の catch 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 catch 関数にエラー内容を順番に渡していきます。catch 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-catch シーケンスを終了します。もしいずれの catch 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。

catch 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 catch 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。

catch 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
```

catch 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?)
```

8. 暗黙的マッピング

8.1. 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。

8.2. 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。

8.3. 適用ルール

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。

8.4. ケーススタディ

8.4.1. 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()

8.4.2. 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。

8.4.3. 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。

8.4.4. 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 = readlines('hoge.h')
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。

9. 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演算子によって切り替えます。

モード	演算子	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。

9.1. 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 構造体を作った後、Fruit 構造体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', this.name, this.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()
```

10. ユーザ定義クラス

10.1. class 関数

ユーザ定義のクラスを作成するには class 関数を使います。class 関数の一般式は以下のとおりです。

```
class(superclass?:function) {block?}
```

10.2. 基本的なクラス定義

下のスクリプトは、A という名前のクラスを作成する例です。

```
A = class {
    Hello() = {
        println('Hello, ')
    }
}
```

class 関数のブロック内で定義される関数は「メソッド」と呼ばれ、このクラスのインスタンスを操作するための関数として働きます。

変数 A には、クラス A のインスタンスを生成するための関数が代入されます。この関数のことを、クラス A の「コンストラクタ関数」と呼びます。クラス A のインスタンスを生成してメソッドを呼び出す例を以下に示します。

```
a = A()
a.Hello()
```

コンストラクタ関数は、ブロックをとることができます。コンストラクタ関数をブロックをつけて評価すると、|obj| という形式でブロックパラメータを渡してブロックを評価します。obj は生成したインスタンスです。この場合、ブロックで最後に評価した値が、コンストラクタ関数の戻り値になります。ブロックを使うと、上の例は以下のように書くことができます。

```
A() {|a|
    a.Hello()
}
```

この表記では、生成したインスタンスはブロックの評価が終わった時点で消滅します。インスタンスの寿命を限定するときに便利です。

コンストラクタ関数は、インスタンスを生成するだけでなくインスタンスの内部状態を初期化する役目ももっています。以下のように __init__ という名前のメソッドを定義すると、コンストラクタを実行した際、インスタンス生成の後にこのメソッドの内容を実行します。

```
B = class {
    __init__(name:string) = {
        this.name = name
    }
    Hello() = {
```

```
        println('Hello, ', this.name)
    }
}
```

メソッド __init__ には引数を指定することができ、コンストラクタ関数も同じ引数リストを持ちます。上のクラス B を生成する例を以下に示します。

```
b = B('Gura')
```

ところで、上の例において this という名前の変数がメソッド内部で使われています。これはメソッドが属しているインスタンス自身への参照になっています。メソッド__init__では this.name に値を代入していますが、これは B のインスタンス内の変数 name への代入になります。メソッド Hello における this.name の値参照は、同じく B インスタンスの name 変数を参照しています。今後、クラスのインスタンス内で定義される変数を「プロパティ」と呼ぶことにします。

変数 this を使ってメソッドを呼び出すこともできます。以下に例を示します。

```
C = class {
    __init__(name:string) = {
        this.name = name
    }
    Hello() = {
        println('Hello, ', this.DuplicateName(4))
    }
    DuplicateName(n:number) = {
        this.name * n
    }
}
```

10.3. コンストラクタ関数についての詳細

この節では、クラスとコンストラクタ関数の生成について詳しく見ていきます。以下に例をあげます。

```
D = class {}
```

これは D という名前のクラスを生成している例ですが、詳しく見ると二つの処理が行われています。ひとつは、「D という名前のクラス」の作成であり、もう一つは「D という名前の関数」の作成です。

まず class 関数を実行すると、class 型のデータを生成して返します。このとき、class 関数自体はクラス名に関する情報を与えられていませんから、生成する class 型データは名前なしクラスになります。

クラスに名前がつけられるのは、代入演算子 = を評価するときです。この演算子は、右辺が class 型のデータで、また名前がついていない場合、左辺のシンボル値をもとにこのクラスに名前をつけます。さらに、演算子 = はこのクラスを生成するコンストラクタ関数を作成し、シンボル D に割り当てます。

10.4. クラスメソッドとインスタンスメソッド

メソッドの定義をするとき、引数リストの括弧に続いてアトリビュート :static をつけると、そのメソッドはクラスメソッドになります。

クラスメソッドは、クラス名の名前空間内に作成した通常関数としてふるまいます。呼び出すときはクラス名とドット記号 "." に続いてメソッド名と引数リストをつけます。

クラスメソッドの一般名を表記するときは *class*.*method*() のようにクラス名とメソッド名を "." でつなげて表します。これは実際の呼び出し方法のときの記述と同じです。

それに対し、インスタンスメソッドの一般名は *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。

10.5. メンバアクセス権

メソッドはデフォルトでは外部に対して公開されています、宣言のアトリビュートに:private をつけると隠ぺいされます。

インスタンス変数はデフォルトでは外部に対して隠ぺいされています。public を使ってシンボルを列挙するか、代入時にアトリビュート:public をつけると外部に公開されます。

引数指定で:priviledged アトリビュートをつけてわたされた変数に対するメンバアクセスでは、隠ぺいされた変数やメソッドにもアクセスできるようになります。

```
D = class {
    public {
        namePublic
    }
    __init__() = {
        this.namePublic = 'public'
        this.namePublic2:public = 'public by attribute'
        this.namePrivate = 'private'
    }
    funcPublic() = {
    }
    funcPrivate():private = {
    }
}
d = D()
println(d.namePublic)      // OK
println(d.namePublic2)     // OK
d.funcPublic()             // OK
println(d.namePrivate)     // error
d.funcPrivate()            // error

func1(d:D) = {
    println(d.namePrivate) // error
}
func2(d:D:priviledged) = {
    println(d.namePrivate) // OK
}
```

10.6. 継承

クラスを継承する場合は、引数 superclass にスーパークラスのコンストラクタ関数を指定します。省略したときは、Gura のルートクラス object をスーパークラスとします。スーパークラスのコンストラクタに渡す引数は、メソッド __init__ のブロック引数に記述します。

```
Person = class {
    __init__(job:string, name:string, age:number) = {
        this.job = job
        this.name = name
        this.age = age
    }
    Print() = {
        println(this.job, ' : ', this.name, ' : ', this.age)
    }
}
Teacher = class(Person) {
    __init__(name:string, age:number) = {|'teacher', name, age|
    }
}
Student = class(Person) {
    __init__(name:string, age:number) = {|'student', name, age|
    }
}
```

10.7. 特別なメソッド

定義するメソッドの中には、すでに出てきた __init__ メソッドを含め、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。

__getitemx__()

インスタンスに対して中身が空のインデクス参照をしたときに呼ばれるメソッドです。

例えば、foo[] という式が評価されると __getitemx__ が呼ばれます。

__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。

__setitemx__(value)

インスタンスに対して中身が空のインデクス代入をしたときに呼ばれるメソッドです。value には代入値が渡されます。

例えば、foo[] = 3 という式が評価されると __setitemx__ が呼ばれ、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。

また、以下のメソッドを定義すると、オペレータをオーバーライドすることができます。

メソッド	オーバーライドするオペレータ
__pos__(a)	+a
__neg__(a)	-a
__invert__(a)	~a
__not__(a)	!a
__add__(a, b)	a + b
__sub__(a, b)	a - b
__mul__(a, b)	a * b
__div__(a, b)	a / b
__eq__(a, b)	a == b
__ne__(a, b)	a != b
__ge__(a, b)	a >=b

__le__(a, b)	a <= b
__cmp__(a, b)	a <=> b
__or__(a, b)	a | b
__and__(a, b)	a & b
__xor__(a, b)	a ^ b
__shl__(a, b)	a << b
__shr__(a, b)	a >> b
__seq__(a, b)	a .. b
__seqinf__(a)	a..

10.8. 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。アトリビュート :loose を指定すると、引数すべてがオプショナルになります。

10.9. 既存のクラスへのメソッド追加

代入演算子を使い、クラス宣言をした後に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: ', this)
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: ', this)
```

11. 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（*.gura）と、C++で記述してビルドしたバイナリモジュール（*.gurd）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（.gura または .gurd）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、シンボルを現在の名前空間にとりこむと、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ります。モジュール内で定義されているシンボル名がユニークなときだけこの表記を利用してください。

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)
```

import 関数をアトリビュート :binary をつけて実行すると、バイナリモジュールのみをインポート対象にします。これは、同じ名前のスクリプトモジュールとバイナリモジュールを用意しておき、スクリプトモジュールから対応するバイナリモジュールをインポートするときに使用します。この機構により、基本機能をバイナリモジュールで提供しておき、それをスクリプトモジュールで拡張することが可能になります。

import 関数をアトリビュート :mixin_type をつけて実行すると、現在の名前空間にモジュール中の型シンボルをとりこみます。

12. リストとイテレータ

12.1. 概要

リストは、任意の要素を集めたものです。数値や文字列、オブジェクトなどをカンマで区切り、ブラケットで囲むと、それはリストになります。

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。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。リストは代表的なコンテナのひとつです。

リストとイテレータは、操作方法が非常によく似ています。また、リストからイテレータに変換したり、イテレータからリストに変換したりすることは、ごく普通に行われる操作です。このため、両者の違いは普段あまり意識する必要がないかもしれません。しかし、要素としてのデータの存在期間が問題になるとき、注意が必要になります。

Gura におけるリストや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、これら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。

12.2. 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。

12.3. 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。

12.4. リストの生成

イテレータをブラケットで囲むと、イテレータを展開した結果得られる要素を持つリストになります。複数のイテレータをカンマで区切ってブラケットで囲むと、それぞれのイテレータを展開して要素に追加します。

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}
```

12.5. 要素操作ダイジェスト

リストの要素操作をするメソッドを使ったスクリプトとその実行結果を示します。メソッドの詳細については「Gura ライブラリリファレンス」を参照ください。

x が [A, B, C, D, E, F, G, H, I, J] というリストである場合:

スクリプト	実行結果
x.head(3):list	[A, B, C]
x.tail(3):list	[H, I, J]
x.offset(3):list	[D, E, F, G, H, I, J]
x.skip(2):list	[A, D, G, J]
x.fold(3):list	[[A, B, C], [D, E, F], [G, H, I], [J]]
x.reverse():list	[J, I, H, G, F, E, D, C, B, A]
x.shuffle():list	[H, D, F, E, I, B, G, A, J, C] (結果はランダム)

x が [A, B, C, D] というリストである場合:

スクリプト	実行結果
x.round(14):list	[A, B, C, D, A, B, C, D, A, B, C, D, A, B]
x.pingpong(10):list	[A, B, C, D, C, B, A, B, C, D]
x.combination(3):list	[[A, B, C], [A, B, D], [A, C, D], [B, C, D]]
x.permutation(3):list	[[A, B, C], [A, B, D], [A, C, B], [A, C, D], [A, D, B], [A, D, C], [B, A, C], [B, A, D], [B, C, A], [B, C, D], [B, D, A], [B, D, C], [C, A, B], [C, A, D], [C, B, A], [C, B, D], [C, D, A], [C, D, B], [D, A, B], [D, A, C], [D, B, A], [D, B, C], [D, C, A], [D, C, B]]

12.6. ユーザ定義イテレータ

新しいデータ型に対して独自のイテレータを定義したい場合は、以下の関数を使うことができます。

- 繰り返し関数 repeat、while、for および cross
- 汎用イテレータ関数 iterator

12.6.1. 繰り返し関数によるイテレータ定義

繰り返し関数 repeat は指定回数だけブロックを評価します。以下は "Hello" という文字列を 10 回画面に表示する例です。

```
repeat (10) {
    println('Hello')
}
```

Gura は繰り返し関数に:iter というアトリビュートをつけることで、イテレータを生成することができます。上の例で、repeat にアトリビュート:iter をつけたコードを以下に示します。

```
x1 = repeat (10):iter {
    println('Hello')
}
```

これはいったいどういう意味を持つのでしょうか。まず、repeat 関数を評価した時点では、ブロックの内容は実行されず、画面には何も表示されません。ここで行われているのは、「"Hello" を 10 回表示するイテレータ」を生成し、それを変数 x1 に代入することです。実際に評価を行うには、以下のように each メソッドを実行します。

```
x1.each() {}
```

イテレータというのは、通常、値を順次返すものを指しますので、上の例は「イテレータ」と呼ぶには少々難がありそうです。繰り返し関数によるイテレータ生成では、繰り返しブロック内の最後に評価された式の値が、イテレータの要素になります。以下で生成しているのは 10 未満の 2 の倍数を返すイテレータです。

```
x2 = repeat (5):iter {|i|
    i * 2
}
```

生成したイテレータをリスト化して内容を確かめてみます。

```
>>> [x2]
[0, 2, 4, 6, 8]
```

通常の繰り返し処理と同じように、break や continue を使ってフローを制御することもできます。
上と同じ処理をbreakを使って書いた例を以下に示します。ループ回数を大きく設定し、指定の回数になったら break でぬけるようにしています。

```
x3 = repeat (100000):iter {|i|
    (i == 5) && break
    i * 2
}
```

評価結果は以下のようになります。

```
>>> [x3]
[0, 2, 4, 6, 8]
```

同じく continue を使った例を考えてみます。10 までの数値を作り、条件に合致しない数値の場合は continue でスキップします。

```
x4 = repeat (10):iter {|i|
    (i % 2 == 1) && continue
    i * 2
}
```

これの評価結果は以下のようになります。

```
>>> [x4]
[0, nil, 2, nil, 4, nil, 6, nil, 8, nil]
```

期待した数値列の間に nil 値が入ってしまいました。これは continue を評価した際のループの評価値が nil になるので、これが要素として扱われるためです。

アトリビュート:xiter を指定すると、要素から nil 値をとりのぞくことができます。書きなおした例を以下に示します。

```
x5 = repeat (10):xiter {|i|
    (i % 2 == 1) && continue
    i * 2
}
```

これの評価結果は、以下のように期待どおりのものになります。

```
>>> [x5]
[0, 2, 4, 6, 8]
```

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)。

```
x6 = repeat (2):iter {|i|
    repeat (2):iter {|j|
        repeat (3):iter {|k|
            [i, j, k]
        }
    }
}
```

評価結果は以下のようになります。

```
>>> [x6]
[[0, 0, 0], [0, 0, 1], [0, 0, 2], [0, 1, 0], [0, 1, 1], [0, 1, 2], [1, 0,
0], [1, 0, 1], [1, 0, 2], [1, 1, 0], [1, 1, 1], [1, 1, 2]]
```

12.7. 汎用イテレータ関数によるイテレータ定義

関数 iterator を使うと、イテレータや値を結合し、任意のデータ列を返すイテレータを定義することができま

す。

以下は [3, 1, 4, 1, 5, 9, 3] というデータ列を返すイテレータです。

```
x7 = iterator(3, 1, 4, 1, 5, 9, 3)
```

イテレータを要素にすると、そのイテレータの内部の要素を展開します。例を以下に示します。

```
x8 = iterator(1..5, 3, 9, 2, 8..3)
```

評価結果は以下の通りです。

```
>>> [x8]
[1, 2, 3, 4, 5, 3, 9, 2, 8, 7, 6, 5, 4, 3]
```

13. 数学に関する機能

13.1. 複素数計算

四則演算、マトリクス演算に対応しています。

13.2. 統計処理

合計・分散値・平均値・標準偏差を算出します。

13.3. 順列

`list#permutation`および`list#combination`メソッドにより、順列および組み合わせによる要素抽出を行います。

13.4. 行列演算

`matrix`クラスを使い、以下の行列演算ができます。

- 加算・減算・乗算
- 逆行列
- 転置行列

行列の要素には、実数および複素数を入れることができます。

13.5. 式の微分演算

式そのものを微分することができます。合成式の微分は以下の公式に則って式を導き出しています。

$\{f(x) + g(x)\}' = f'(x) + g'(x)$
$\{f(x) - g(x)\}' = f'(x) - g'(x)$
$f(g(x))' = f'(u)g'(x)$
$\{f(x)g(x)\}' = f'(x)g(x) + f(x)g'(x)$
$\{f(x) / g(x)\}' = \{f'(x)g(x) - f(x)g'(x)\} / g(x)^2$
$\{f(x)^{g(x)}\}' = f'(x)g(x)f(x)^{g(x)-1} + g'(x)\{\log f(x)\}f(x)^{g(x)}$

微分式を得るには、`function` クラスの `diff` メソッドを使います。以下のように数式からなる関数を定義し、`diff` メソッドを実行すると、微分式からなる関数を返します。関数の定義内容は `expr` プロパティで確認できます。

```
>>> f(x) = math.sin(x) ** 2
f(x)
>>> g = f.diff()
g(x)
>>> g.expr
`(math.cos(x) * 2 * math.sin(x))
```

得られる結果の例を以下に示します。

式	微分結果
x ** 2	2 * x
x ** 3	3 * x ** 2
x ** 4	4 * x ** 3
a ** x	math.log(a) * a ** x
math.sin(x)	math.cos(x)
math.cos(x)	-math.sin(x)
math.tan(x)	1 / math.cos(x) ** 2
math.exp(x)	math.exp(x)
math.log(x)	1 / x
math.log10(x)	1 / (x * math.log(10))
math.asin(x)	1 / math.sqrt(1 - x ** 2)
math.acos(x)	(-1) / math.sqrt(1 - x ** 2)
math.atan(x)	1 / (1 + x ** 2)
math.sqrt(x)	1 / (2 * math.sqrt(x))
math.sin(x) ** 2	math.cos(x) * 2 * math.sin(x)
math.sin(x ** 2)	math.cos(x ** 2) * (2 * x)
math.log(math.sin(x))	(1 / math.sin(x)) * math.cos(x)
x ** 2 * math.sin(x)	2 * x * math.sin(x) + x ** 2 * math.cos(x)
math.sin(x) / x ** 2	(math.cos(x) * x ** 2 - math.sin(x) * (2 * x)) / x ** 2 ** 2
3 ** 2 * x	2 * math.log(3) * 3 ** 2 * x
math.log((x ** 2 + 1))	(1 / (x ** 2 + 1)) * (2 * x)
(x - 1) ** 2 * (x - 2) ** 3 / (x - 5) ** 2	((2 * (x - 1) * (x - 2) ** 3 + (x - 1) ** 2 * (3 * (x - 2) ** 2)) * (x - 5) ** 2 - (x - 1) ** 2 * (x - 2) ** 3 * (2 * (x - 5))) / (x - 5) ** 2 ** 2
math.sin(2 * x - 3)	math.cos(2 * x - 3) * 2
math.cos(x) ** 2	-math.sin(x) * 2 * math.cos(x)
(2 * x - 1) ** 3	6 * (2 * x - 1) ** 2
math.sqrt(x ** 2 + 2 * x + 3)	(1 / (2 * math.sqrt(x ** 2 + 2 * x + 3))) * (2 * x + 2)
1 / x	(-1) / x ** 2
math.exp(x) + math.exp(-x)	math.exp(x) - math.exp(-x)
math.exp(x) - math.exp(-x)	math.exp(x) + math.exp(-x)
(math.sin(x + 2) + x + 2) * (math.sin(x + 3) + x + 3)	(math.cos(x + 2) + 1) * (math.sin(x + 3) + x + 3) + (math.sin(x + 2) + x + 2) * (math.cos(x + 3) + 1)
math.sin(math.sin(x ** 2 / 3))	math.cos(math.sin(x ** 2 / 3)) * (math.cos(x ** 2 / 3) * ((2 * x * 3) / 9))

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、多くの場合ハードディスクやソリッドストレージデバイスなどのファイルやディレクトリの名前を指します。しかし、Gura における「パス名」はそれよりも適用範囲が広く、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。以下に Gura で扱えるパスの種類をまとめます。

- ファイルシステム内のパス
- インターネットの URI パス
- アーカイブファイル内のパス

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、データの読み書きをするストリームの生成や、要素一覧の取得などを行う仕組みを提供します。また、モジュールをインポートすることで、まったく新しいプロトコルによるパス名の解釈や実際の資源操作を追加することもできます。

14.1.1. ファイルシステム内のパス

ファイルシステム上のファイルやディレクトリの例を以下に示します。

```
/home/yamada/work
C:\Windows\Media\chimes.wav
```

パス名を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。どちらを使って記述しても、実際にファイルをオープンしたりディレクトリを指定したりする際に、現在動作している OS に応じて適切に変換が行われます。バックスラッシュはエスケープする必要があるので、スラッシュ "/" を使って記述した方がすっきりとすることが多いでしょう。

14.1.2. インターネットの URI パス

インターネットの URI パスの例を以下に示します。

```
http://sourceforge.jp/
ftp://foo.hoge.com/dir1/dir2/
```

URI パスは、インターネットの通信プロトコル名に続いてコロン、スラッシュと資源へのパス名が記述されます。Gura は、通信プロトコルに対応するモジュールをインポートすることでこれらのパスにアクセスができるようになります。

14.1.3. アーカイブファイル内のパス

アーカイブファイルの中のパスの例を以下に示します。

```
hoge.zip/foo/bar.txt
footool.tar.gz/src/main.c
```

```
hoge.zip/
```

hoge.zip や footool.tar.gz はそれぞれ ZIP 形式および tar 形式でファイルをまとめたアーカイブファイルです。これらのアーカイブファイルに対応したモジュールをインポートすることで、アーカイブファイル名に続けて内部のパスを指定することができます。

言うまでもなく、アーカイブファイル自体はファイルシステムまたはインターネット上の資源として存在します。デフォルトではファイルシステム上にあるアーカイブファイルを扱えますが、必要なモジュールをインポートすることでインターネット上のアーカイブファイルを指定し、さらにその内部のパスを指定することができます。以下に例を示します。

```
/home/yamada/work/hoge.zip/foo/bar.txt
http://sourceforge.jp/hoge.zip/foo/bar.txt
ftp://foo.hoge.com/dir1/dir2/hoge.zip/foo/bar.txt
```

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。

ディレクトリ内の要素をサーチするため、以下の関数を用意しています。詳細は「Gura ライブラリリファレンス」を参照ください。

関数	説明
path.dir	パス名で指定したディレクトリに含まれるファイルまたはディレクトリをサーチします
path.walk	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ます
path.glob	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

15. ストリーム

15.1. 概要

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

15.2. ストリームの種類

ストリームは以下のものに大別されます。

- ソースストリーム
- デスティネーションストリーム
- フィルタストリーム

ソースストリームはデータの入力元になるストリームです。`stream#read` をはじめとするデータ読み込みメソッドを使ってデータを取得します。

デスティネーションストリームはデータの出力先になるストリームです。`stream#write` などのデータ書き込みメソッドを使ってデータを出力します。

フィルタストリームは、ソースストリームまたはデスティネーションストリームにアタッチするストリームです。フィルタストリームに読み込み操作を行うと、アタッチしたストリームからデータを読み込み、それを加工した結果を返します。また、フィルタストリームに書き込み操作を行うと、書き込みデータを加工し、その結果をストリームに書き込みます。

15.3. 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。

また、open 関数で明示的にオープンしなくても、stream 型の値を指定した引数に文字列を渡すと、自動的に stream 型にキャストがされます。例えば、テキストファイルから行ごとに文字列を読み込む関数 readlines は、最初の引数に stream を受け取るので、open 関数を使って以下のように実行します。

```
lines = readlines(open('hoge.txt'))
```

型キャストを使うと、以下のように記述できます。

```
lines = readlines('hoge.txt')
```

15.4. コーデックの指定

Gura にあらかじめ組み込まれている文字コーデックを以下に示します。

モジュール	コーデック
codecs.basic	us-ascii, utf-8, utf-16, base64
codecs.iso8859	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	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。

15.5. 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。

15.6. プロセス実行と標準入出力

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 = binary()
os.stdout = buff.stream()
os.exec('program')
```

標準入出力を切り替える処理は比較的頻繁に行われますが、このとき変数の設定を上記のように直接行うと記述が煩雑になります。これを解決するため os.redirect という関数が用意されています。以下は上の処理をこの関数を使って書きなおした例です。

```
buff = binary()
os.redirect(nil, buff) {
    os.exec('program')
}
```

切り替えた効果がブロックの範囲内に限定されるので、処理が分かりやすくなります。

15.7. 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#copyto(stream:stream):map:reduce
stream#copyfrom(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。

15.8. ストリーム間のデータコピー

入力用ストリームから出力用ストリームにデータをコピーするための関数として copy が用意されています。

copy 関数の最も簡単な使用方法は、コピー元のファイル名とコピー先のファイル名を指定して内容をコピーするというものです。以下に例を示します。

```
copy('src.txt', 'dest.txt')
```

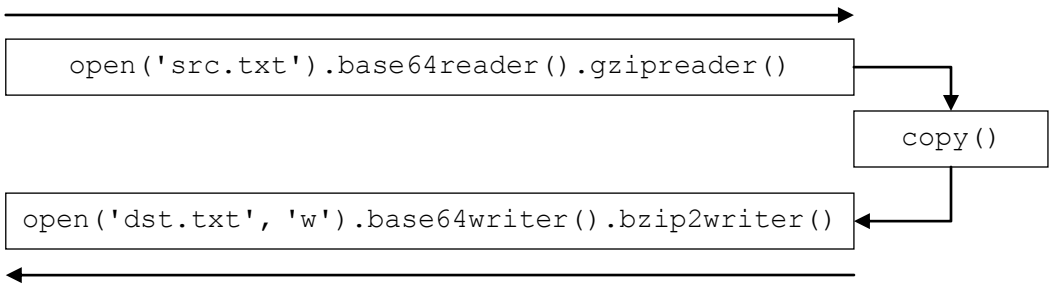
内部の処理では、ファイル名がストリームに変換されるので、上の処理は以下のように記述したのと同じです。

```
copy(open('src.txt', 'r'), open('dest.txt', 'w'))
```

最初の例の方が簡単ですが、二番目に示したものの方がより柔軟性に富んだ処理をすることができます。例えば、ファイル src.txt に base64 でエンコードされた gzip 形式のデータが格納されているとしましょう。このデータを展開し、今度は bzip2 形式で圧縮して再び base64 でエンコードした結果を dst.txt に格納する処理は以下のように記述することができます。

```
copy(open('src.txt').base64reader().gzipreader(),
     open('dst.txt', 'w').base64writer().bzip2writer())
```

データの流れを模式的に表すと以下のようになります。



15.9. スクリプトファイルの実行

15.9.1. アーカイブ中のスクリプトファイル

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が、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 zip archive.zip/hello.gura
```

"-i" 引数により zip モジュールをインポートし、ZIP アーカイブ内のスクリプトファイルを指定して実行しています。

スクリプトファイルのサフィックスが .gurc または .gurcw のファイルをコンポジットファイルと呼び、これらにはスクリプトファイルや他のファイルを梱包することができますが、これはZIPアーカイブ内のストリームにアクセスする機構を使って実現されています。Gura はコンポジットファイルのサフィックスを見つけると、自動的に zip モジュールをインポートし、アーカイブファイル内へのアクセスができるようにします。

15.9.2. HTTP 上のスクリプトファイル

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.gura
```

16. イメージ

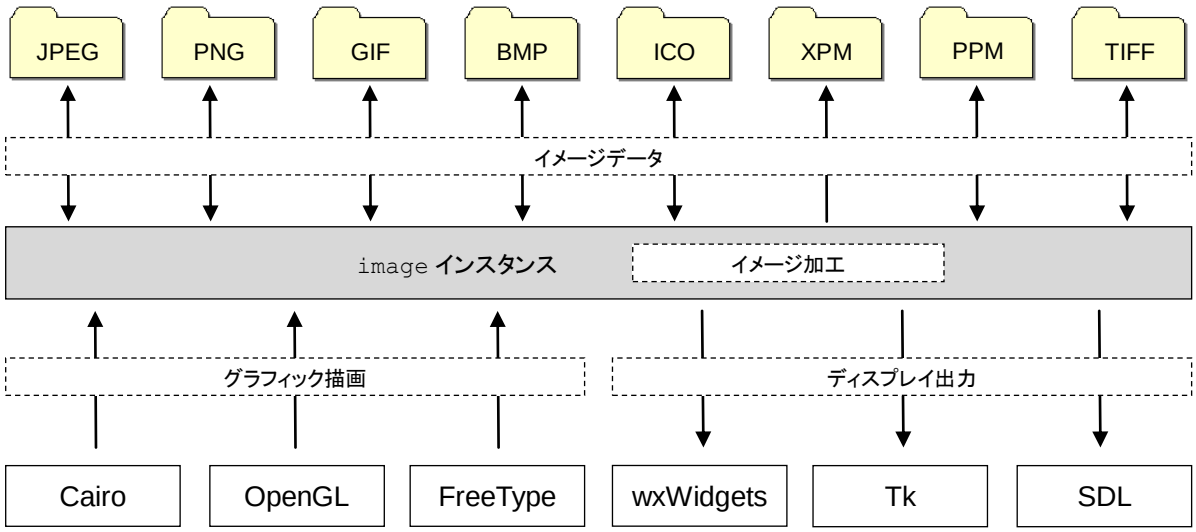
16.1. 概要

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、イメージデータ入出力・フィルタ処理・グラフィック描画・ディスプレイ出力などがあり、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、Gura のイメージインスタンスを Cairo や OpenGL などのグラフィック描画ライブラリの描画対象にす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。



16.2. 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. ストリームからのイメージデータ読み込み

ストリームからイメージデータを読み込むときは、image 関数を以下の形式で呼び出します。

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

引数 stream は、イメージデータを読み込むストリームです。この引数に文字列を渡すと、それをパス名として解釈して stream 型にキャストし、イメージデータを読み込みます。

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
msico	Windowsアイコンファイル	msico
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。

16.4. ストリームへのイメージデータ書き込み

ストリームにイメージデータを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:w, 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。この引数に文字列を渡すと、それをパス名として解釈して stream 型にキャストし、イメージデータを書きこみます。

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、「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。

16.5. イメージ加工

image クラスには、以下の操作を行うメソッドが用意されています。

メソッド	操作
image#crop	イメージ切り出し
image#flip	左右・上下反転
image#paste	イメージ貼り付け
image#reducecolor	減色処理
image#resize	サイズ変更
image#rotate	任意の角度の回転
image#thumbnail	サムネイル画像生成

16.6. グラフィック描画

二次元グラフィックを描画したいときは、ライブラリ Cairo をサポートするモジュール cairo が便利です。

三次元グラフィックライブラリ OpenGL をサポートするモジュール opengl を使うと、Z バッファを使った高度な三次元グラフィック描画ができます。

テキストを扱いたいだけであれば、ライブラリ FreeType をサポートする freetype モジュールで手軽にテキストをイメージに埋め込むことができます。

16.7. ディスプレイ出力

GUI を構築する wxWidgets モジュール wx と、Tcl/Tk モジュール tk を用意しています。

また、高速な画面表示を可能にする SDL (Simple Direct Layer) のモジュール sdl が用意されています。

17. テンプレートエンジン

テンプレートエンジンを使うと、任意のテキスト文字列の中に Gura スクリプトを埋め込み、スクリプトの実行結果を文字列中に挿入することができます。テンプレートエンジンを起動には以下の方法があります。

コマンドライン

オプション–T を使ってテンプレートを記述したテキストファイルを指定すると、その内容を評価します。

関数呼び出し

メソッド `string#template()`、`stream#template()` または関数 `template()` を使い、文字列またはストリーム中に記述されているテンプレート文字列に対して評価を行います。

Gura スクリプトは `"${"` と `"}"` にはさんで記述します。この内部は、通常の Gura スクリプトとして扱われるので、改行などを含んでいてもかまいません。通常のテキストの中に `"${"` という文字の並びがあり、これをスクリプトでなく通常文書として扱う場合は `"$${"` と記述します。スクリプト中の空白や改行は結果に影響を与えません。

評価結果を出力するときのルールは以下の通りです。

- 結果が文字列のとき、その内容を出力します。
- リストやイテレータの場合、その要素を文字列に変換して結合した結果を出力します。
- それ以外の `nil` 以外の要素は、文字列に変換されて出力されます。
- 行の先頭からスクリプト開始の `"${"` の間に空白が存在し、スクリプトの結果が複数行にわたる場合は、先頭の空白分がすべての行の前に追加されます（オートインデント機能）
- 最後に現れる改行コードはとりのぞかれます

例を以下に示します。

テンプレート	結果
`Hello ${'gura'.capitalize()} World`	`Hello Gura World`
`Hello ${3 + 4 * 2} World`	`Hello 11 World`
`AB ${['1st', '2nd', '3rd']} CD`	`AB 1st2nd3rd CD`

`AB ${['1st\n', '2nd\n', '3rd\n']} CD`	`AB 1st`
	`2nd`
	`3rd CD`

`  ${['1st\n', '2nd\n', '3rd\n']}`	`  1st`
	`  2nd`
	`  3rd`

`Hello ${`
`'gura'.capitalize()`
`} World`
	`Hello Gura World`

評価結果が `nil` の場合は何も出力されません。このふるまいは、評価結果が空の文字列のときと同じですが、`"${"` と `"}"` の直後にある改行コードをとりのぞく点が異なります。例を以下に示します。

テンプレート	結果
`Hello ${''} World`	`Hello World`

Line1 ${''} Line2	Line1 Line2
Hello ${nil} World Line1 ${nil} Line2	Hello World Line1 Line2

"${" と "}" の中で最後に記述されている関数が常にブロックをとる関数で、スクリプトにはブロックが記述されていない場合、そのスクリプトの後から "${end}" が現れるまでの文字列がそのブロックの内容として扱われます。これにより、if-elsif-else などの制御構文やループなどを記述することができます。

テンプレート	結果

${for (i in 1..5)}
 ${if (i < 2)}
  ${i} is less than two
 ${elsif (i < 4)}
  ${i} is less than four
 ${else}
  ${i} is greater or equal to four
 ${end}
${end}
	1 is less than two
	2 is less than four
	3 is less than four
	4 is greater or equal to four
	5 is greater or equal to four

オプショナルにブロックをとる関数で ${end} までの文字列をそのブロック内容として扱いたい場合は以下のように空のブロックを関数に渡します。

テンプレート	結果
${range(3) {}} Hello World ${end}	Hello World Hello World Hello World

ブロックパラメータを指定する場合は、以下のようにブロックパラメータのみを記述したブロックを関数に渡します。

テンプレート	結果
${range(3) {|i|}} ${i} ${end}	0 1 2