プログラミング初歩一般化済み #4 変数、関数、名前空間

Mimir Yokohama

2020-07-23

目次

変数

変数をはじめよう

```
puts "Hello, world!"
```

というコードにプログラミングらしさを感じるのはなかなか難しいでしょう。プログラムといえばやはり何かしら動的な要素がほしいものです。

変数はそれを用意に実現します。変数を理解すればプログラムらしいコードが書けるようになります。

変数とは

変数は例えば Ruby なら次のように使います。

```
hell = "Hello"
```

これを「変数 `hell` に文字列 `Hello` を代入する」というふうに表現します。

Perl では変数に種類があり、`$`ではじまるスカラー変数、`@`ではじまるリスト変数、`%` ではじまるハッシュ変数、`&`で始まる関数の 4 つです。文字列はスカラー変数なので次のようになります。

```
$hell = "Hello";
```

もうちょっと楽しい感じにしてみましょう。変数に名前を入れて挨拶してみます。

```
$name = "John";
print("Hello, ${name}!\n");
```

Perl ではダブルクォート文字列リテラル中では`${...}`という形で変数名を指定して変数を文字列中に展開することができます。

変数は値に名前をつけるものです。ここでは `John` という文字列に`$name` という名前をつけました。これで、これが有効な範囲であればどこからでも`$name` という形で `John` を呼び出せます。

でもそれだけだと、John と書いても同じですね。しかし、変数は「変数」という名前の通り、その値は変わりうるものです。おっと、値が変わりうるという表現はあまり正しくありません。値は不変でなければ変数でなくてもかわりうるからです。正しくは、「その変数が何を指しているかが変わりうる」ですね。

では次の Perl コードを見てみましょう。

```
$name = $ARGV[0];
print("Hello, $name!\n");
```

今度は$name の値が$ARGV[0] となりました。スカラー変数の形をしています。

これは、$ARGV というスカラー変数ではなく、@ARGV というリスト変数になります。リスト変数は 0 個以上のスカラー値からなりますが、そのうちひとつだけを取り出すとスカラー値になりますので、リスト変数からひとつだけを取り出す場合は@ではなく$で始めます。

リスト変数は先頭から 0 で数え始めます。つまり、$ARGV[0] はリスト変数@ARGV の最初のスカラー値を意味します。そして、@ARGV というのは、コマンドライン引数を指します。

ここで例えば次のように起動します。

```
$ perl hello.pl Bobby
```

するとこうなります。

```
$ perl hello.pl Bobby
Hello, Bobby!
```

コマンドライン引数が変われば$ARGV[0] の値は変わり、そして$name の値も変わるので結果が違ってきます。

```
$ perl hello.pl Haruka
Hello, Haruka!
```

これはリテラルで書くだけでは表現できませんね。

変数はある程度の制約はありますが、値として表現できるものであればなんでも変数として名前をつける方法があるのが普通です。

一度名前をつけてしまえばそれが何を指しているかということは変わらない場合、定数というものを使うことができます。定数はその名の通り、指しているものが定まっています。この場合は定数にしてしまっても問題なさそうですね。

Ruby では定数は大文字ではじまります。変数は小文字ではじまるので、区別できます。

```
NAME=ARGV[0]
printf("Hello, %s!\n", NAME)
```

定数にしなければならない、というケースは比較的稀です。後述するスコープ上の制約がある場合もありますが、そうしたことを除けば定数でできることは変数でもできるのが普通です。

そこで定数を使うのは、間違ってその名前が指しているものを変更してしまわないためです。定数に対してそのようなことをすると、エラーになるか、もしくは警告されるので、問題にきづくことができます。定数を使うことで、「この名前は常に同じものを指している」という了解を得ることができます。

ただ、今行ったように定数でなければ困る、ということは基本ありませんし、そのために Perl には単純な定数の表現がありません。定数を使いたければ `constant` プラグマを使う必要があります。`constant` プラグマが導入されたのは Perl としては結構あとになってからですから、あまり必要と感じていなかったことがわかります。

関数

さて、変数を使うことにより値に名前をつけることができるようになりました。しかし、名前をつけたいものの値と呼べるかどうか微妙なものがあります。それが関数です。

関数、という言葉はそもそも値となりうるものを関数と呼ぶ、という考え方もあるのですが、ここでは単純に「呼び出し可能なコード群」というふうに捉えます。

ここでは「浮気なジョニー」という処理を考えてみましょう。浮気なジョニーは名前を与えられたら 3 回 “I love you” といいます。

```
print("I love you $name\n");
print("I love you $name\n");
print("I love you $name\n");
```

しかしジョニーは浮気性なのでいろんな女の子に “I love you” といいます。そのたびに 3 回も書いていたら大変ですね。

そこで、3 回 “I love you” という行為を flimsy(薄っぺらい) と名付けましょう。こんな感じです。

```
sub flimsy {
    print("I love you $name\n");
    print("I love you $name\n");
    print("I love you $name\n");
}
```

これが Perl のサブルーチンです。`sub` のあとに関数名を書くことで、一連の処理に名前をつけることができます。これでたくさんの女の子に “I love you” というのが簡単になりました。

```
sub flimsy {
    print("I love you $_[0]\n");
    print("I love you $_[0]\n");
    print("I love you $_[0]\n");
}

flimsy("Jessy");
flimsy("Anna")
flimsy("Bob")
```

Ruby の場合は「メソッド」と呼ばれるものになりますが、これはまた違う概念が入ってくるので今はやめておきましょう。ただ、関数というのは別にあり、こちらは常に値になります。ですから、変数で名前をつけることができます。

```
flimsy = lambda {|name| 3.times { printf("I love you %s\n", name) } }
```

JavaScript の場合も関数は常に値になるようになっており、変数に入れることができますし、変数に入れなくても変数と同じ扱いになります。

Perl は無名サブルーチンという形で関数を値にすることができます。

```
$flimsy = sub {
    print("I love you $_[0]\n");
    print("I love you $_[0]\n");
    print("I love you $_[0]\n");
}
```

ちょっとわかりにくいかもしれませんね。

名前空間

名前をつける

さて、ここまで変数も定数も関数も「名前をつける」という言い方をしました。

変数はよく「名前の書かれた箱の中に値を入れる」という表現がなされ、実際にスラングとして「変数 x に値を格納する」というような言い方をします。

しかし、実際はこれは便宜的な話であり、「名前をつける」ということであるというのをきちんと認識する必要があります。

スコープ

スコープの基本

佐藤さんには幼馴染の山田くんがいます。山田くんはおもしろいヤツであり、近所では有名です。小学校の頃は山田といえば彼のことでした。佐藤さんの学校には山田先生がいます。そして、佐藤さんはバンドをしていてフルートを吹きますが、佐藤さんのバンドにはファゴット担当の山田さんがいます。そして、佐藤さんは国語が好きで、憧れの人は国語の大家、山田大先生です。

佐藤さんは日々過ごす中で山田という名前の人が 4 人出てきます。しかし、みんな山田という名前です。まさか山田 A、山田 B と呼び分けるわけにもいきません。

しかし、ほとんどの場合「山田」といったときに誰を指すかがわからない、ということはありません。小学校時代の友達と話しているときは山田くん、学校では山田先生、バンド仲間となら山田さん、そして将来の話をするときや憧れの人の話をするときは山田大先生になるでしょう。

このように「名前の通用する範囲」を示すのが「スコープ」です。次の疑似コードを見てみましょう。

```
def ご近所会 {
  山田 = "山田くん"
  呼び出し
  おはなし
}
```

```
def 授業 {
  山田 = "山田先生"
  登校
  着席
}
```

ここでは同じ**山田**ですが、この**山田**が通用するのはこの関数の中だけです。関数の中で定義されていますが、これは関数が呼び出されたときに変数**山田**が上書きされるということではなく、そもそもこの変数**山田**はそれぞれの関数の中がスコープであり、そのスコープの外側ではこの変数自体が見えません。

次の Ruby コードで見てみましょう。

```ruby
yamada = nil  # 変数を定義していないとエラーになる

def school
  yamada = "山田先生"
  yamada     # 山田先生
end

yamada       # nil
school()
yamada       # nil
```

このように、`school` 関数の中の `yamada` 変数は `school` 関数内でのみ有効なのであり、その外側にある変数 `yamada` は同じ名前を持つ別の変数です。

「その名前が通用する範囲」がスコープです。

スコープの種類

スコープには 3 つの種類があります。

- グローバルスコープ
- レキシカルスコープ
- ダイナミックスコープ

順に見ていきましょう。

グローバルスコープはプログラム全体に通用する名前です。同一プログラム中のどこで参照したとしても、同じ名前であれば同じものを指します。グローバルスコープを持つ変数をグローバル変数と言いますが、グローバル変数の利用は様々な問題を引き起こすため、一般に好まれません。一方、定数はグローバルスコープを持つことが多く、変数と定数の使い分けのひとつとして「グローバルスコープを必要とするのであれば定数」という使い方がなされたりします。

グローバル変数を使用することは忌避されますが、これを「グローバル変数を使ってはならない」と解釈するのは適切ではありません。グローバル変数が問題となりうるのは、同一の名前を使うことが常に名前の衝突になるからであり、名前の利用ごとにプログラム全体に対して気を配らねばならないからです。ですから、再利

用される予定のない小さなコードでグローバル変数を使うことに支障があるわけではありません。このような場合、グローバル変数の利用によって短く書ける、あるいは明瞭になるのであればグローバル変数を使うべきです。

レキシカルスコープは、ローカルスコープ、スタティックスコープ、静的スコープ、字句的スコープなどとも言います。スコープが記述上確定するタイプのスコープです。

先程の「関数の中でのみ有効な名前」というのは、有効範囲が関数の中にあることは実行するまでもなく見かけ上で判断することができるため、レキシカルスコープを持つということになります。

ローカルスコープを持つ変数をローカル変数と言います。この言葉については注意が必要です。なぜならば、ローカル変数と言ったときにより限定された範囲のものであるという誤解が蔓延しているからです。しかしその誤解は明確な定義があるものではありません。例えば関数の中だけで有効な変数がローカル変数である、あるいはブロックの中だけで有効な変数がローカル変数であるといった説明がされたりしますが、それではローカル変数が定義しきれないため、そのような説明をするとローカル変数の説明が二転三転することになります。しかし、結局のところどちらも「関数内がスコープとなるレキシカルスコープ」「ブロック内がスコープとなるレキシカルスコープ」であり、レキシカルスコープ＝ローカルスコープを持つ変数がローカル変数であるという言い方で十分に説明ができます。そして、それ以上にローカル変数の意味を限定することはできません。

ダイナミックスコープは名前が何を指すかが実行時に決まるスコープです。動的スコープとも言います。

Perl では `local` 宣言がグローバル変数に対してネストされたダイナミックスコープを生成します。

次の Perl コードを見てみましょう。[*1]

```
$name = "Haruka";

sub call_me() {
  print $name, "\n";
}

sub call_him() {
  local $name = "Bobby";
  call_me;
}

call_me;
call_him;
call_me;
```

最初に定義されている`$name` はグローバル変数です。ですから、どこで参照したとしても同じものを参照することになります。

この変数を使用しているのは `call_me` 変数です。`call_him` 関数も `call_him` 関数が `call_me` 関数を呼んでいますので、3 回呼ばれることになります。

`call_him` 関数では `local` 宣言によってグローバル変数`$name` に対してダイナミックスコープを持つ値 `Bobby`

[*1] この Perl コードは直接グローバル変数を使うため、最新の perl では互換モードでないと動作しません。

を代入しています。call_him を呼び出すと、call_him 関数内でグローバル変数にネストしたダイナミックスコープを持つ値がセットされるため、ここから call_me 関数を呼び出されたとき、$name の値は Bobby になっています。

```
% perl ex1.pl
Haruka
Bobby
Haruka
```

これがグローバル変数を上書きしたわけではないことは、次に call_me 関数を呼び出したときに値が Haruka に戻っていることでわかります。では、local を外してみましょう。

```
$name = "Haruka";

sub call_me() {
  print $name, "\n";
}

sub call_him() {
  $name = "Bobby";
  call_me;
}

call_me;
call_him;
call_me;
```

今度はグローバル変数の値が書き換えられているため、3 回目の呼び出しでも値は変わったままです。

```
% perl ex1.pl
Haruka
Bobby
Bobby
```

では my 宣言によって call_him の$name をローカル変数にしてみましょう。

```
$name = "Haruka";

sub call_me() {
  print $name, "\n";
}

sub call_him() {
  my $name = "Bobby";
  call_me;
}
```

```
call_me;
call_him;
call_me;
```

今度は call_him 内の$name は call_him 関数内でのみ通用する名前となりましたから、call_me 関数には影響しません。そのため 3 回とも Haruka になります。

```
% perl ex1.pl
Haruka
Haruka
Haruka
```

多くの言語ではグローバルスコープを持つ名前とレキシカルスコープを持つ名前が使用されます。また、オブジェクト指向言語においては動的結合が重要な要素になるためダイナミックスコープを持つものが多く、結果としてそれがどのような名前であるかによって 3 種類のスコープが使い分けられています。

また、変数を常にダイナミックスコープにする言語もあります。いくつかの Lisp 系言語は通常の変数がダイナミックスコープを持ちます。

ほとんどの言語においてこれらのスコープを使い分ける必要はなく、名前の種類によって適切に使い分けられます。良い設計の言語であれば、そのようなスコープの違いはごく自然な感覚として受け入れられるようになっており、コードを書くときに名前とスコープの関係を意識する必要はあまりありません。

例えば Ruby はオープンクラスであり、クラスを何度でも変更することができますが、このときインスタンス変数を定義したクラス定義以外の場所でのクラス定義でもインスタンス変数を参照できます。

```ruby
# クラス Foo の定義です
class Foo
  def initialize
    # インスタンス変数の定義
    # このクラス定義の中で@name を定義しています
    @name = "John"
  end

  def hello
    # インスタンス変数のスコープはオブジェクトであり、
    # これはオブジェクトのもとになるクラスの定義なので、
    # 当然この中で (たとえレキシカルスコープでも) 見えます。
    printf("Hello, %s\n", @name)
  end
end

# ここはクラス定義の外です。
# ここでは先程のインスタンス変数は見えません。

puts "Are you ready?"
```

```ruby
foo = Foo.new

# もう一度クラス Foo を定義し、メソッド greeting を追加します。
class Foo
  def greeting
    # レキシカルスコープなら見えない位置です
    printf("Hey %s, How are you?", @name)
  end
end

foo.greeting    # Hey John, How are you?
```

foo.greeting というコードではレシーバオブジェクト foo がコンテキストであることを実行時にセットした上で greeting メソッドが呼び出されるため、オブジェクト foo のインスタンス変数が見えます。これは、インスタンスレベルで評価されれば常に見えます。

```ruby
foo.instance_eval {
  p @name
}
```

実は Ruby ではクラス変数はレキシカルスコープを持っているため、instance_eval でオブジェクトのコンテキストからクラス変数は見えません。

```ruby
class Foo
  def initialize
    @@name = "John"
  end
end

foo = Foo.new

foo.instance_eval {
  p @@name
}
```

```
%  ruby ex3.rb
ex3.rb:10: warning: class variable access from toplevel
Traceback (most recent call last):
    2: from ex3.rb:9:in `<main>'
    1: from ex3.rb:9:in `instance_eval'
ex3.rb:10:in `block in <main>': uninitialized class variable @@name in Object (NameError)
```

これは仕様だそうです。

名前空間の仕組み

スコープは名前の有効範囲を限定します。レキシカルスコープはコード上で、ダイナミックスコープは実行上である範囲でのみその名前が有効であるようにします。

このように名前が属する限定された領域を「名前空間」と言います。しかし、一般に名前空間と言った場合にはスコープのことを表しません。

名前空間というと、通常名前そのものがより上位の名前に所属しているものを言います。

例えば、「ミーミル町の山田」と「ミーミル高校の山田」は違うものとして識別可能です。そして、ミーミル町の話をしているときは「山田」といえば「ミーミル町の山田」として認識できますし、ミーミル高校の話をしているときならば「ミーミル高校の山田」であると認識できます。

これが名前空間です。もしこれがグローバルな名前であれば、「ミーミル町の山田」と「ミーミル高校の山田」は認識できますが、「山田」では何を指しているのかわかりません。

これは擬似コードで次のようなことを意味します。

```
namespace ミーミル町 {
  山田 = "山田くん"
  def ご近所会 {
    呼び出し
    おはなし
  }
}

namespace ミーミル高校 {
  山田 = "山田先生"
  def 授業 {
    登校
    着席
  }
}
```

これで**ミーミル町**名前空間の中で変数**山田**といえば常に**山田くん**を指すようになりました。

ではこれを名前空間の外側で言及したい場合はどうすれば良いでしょう？ 書き方は言語により様々ですが、一般的には完全修飾と呼ばれる一番もとになる名前空間から書けば表現することができます。

ミーミル町::山田

もっとも、namespace 直下のローカル変数のスコープが名前空間であるならば、外の名前空間からアクセスすることはできません。名前空間は「より広いスコープで衝突しない形で名前を使う」ためにあります。

Ruby ではモジュール、またはクラスが名前空間を分離します。スコープが名前空間の外まであるのはグローバルな名前、つまり定数だけです。グローバル変数はどの名前空間で定義したとしても名前空間に所属するようになっていないため、名前空間つきでグローバル変数を定義することができません。

Perl では package というものが名前空間になり、package 変数というものがあります。このあたりの仕組みは非常に直感に反する複雑なものであるため、省略します。

JavaScript はすべての名前がオブジェクトに属します。これはオブジェクト指向の話の中で説明します。

名前空間としてのオブジェクト指向

オブジェクト指向とは

オブジェクト指向という言葉は広く使われていますが、その定義が厳密に固定できるわけではありません。動的結合は必須要素であるとも考えられますが、そんなこともありません。

オブジェクト指向の特徴としてよく

- カプセル化
- 継承
- 多態性

の 3 つが挙げられますが、これらは別にオブジェクト指向言語に共通しているわけでもありません。

ただし、オブジェクト指向を表現するためにはオブジェクト名前空間か動的結合のどちらかを最低でも取り入れる必要があります。

オブジェクト名前空間

オブジェクト名前空間はオブジェクトごとに独立した名前を持っているというものです。

Ruby ではオブジェクト生成時に「コンストラクタ」として `initialize` というメソッドが呼ばれます。そして、インスタンス変数はオブジェクトがスコープになる変数で、`@`ではじまります。

```
# オブジェクトのベースになる「クラス」の定義
class Foo

  # コンストラクタ メソッド
  def initialize(name)
    @name = name
  end

  # インスタンス変数 @name を使うメソッド
  def hello
    printf("Hello, %s\n", @name)
  end
end

john = Foo.new("John")       # Foo クラスのオブジェクト john
haruka = Foo.new("Haruka")   # Foo クラスのオブジェクト haruka
```

```
john.hello      # hello というメソッドの名前自体がオブジェクトに属している
                # @name は john 固有の値なので "Hello, John"

haruka.hello    # こちらも同じようにメソッドの名前はオブジェクトに属している
                # こらちの@name の値は Haruka なので "Hello, Haruka"
```

より厳密にいうと Ruby は hello というメソッドはオブジェクトに属しているわけではなく、Foo クラスに属しています。そして、

1. オブジェクトの特異クラス
2. オブジェクトの特異クラスに include されているモジュール
3. オブジェクトのクラス
4. オブジェクトのクラスに include されているモジュール
5. オブジェクトのクラスのスーパークラス

といった順に探索され、オブジェクトのクラス (つまり Foo クラス) でインスタンスメソッドとして定義されている hello メソッドが呼ばれています。ですから、厳密には hello メソッドの名前空間はオブジェクト自身ではなく、オブジェクトのクラスの名前空間にあり、オブジェクトの名前空間としてアクセスすることで連鎖的にアクセスすることができています。

また、@name というインスタンス変数のほうはスコープがオブジェクトとなるものであり、オブジェクトに固有のものとなります。スコープがオブジェクトにあるということは、他のオブジェクトにおける同名のインスタンス変数は別の名前であり、逆に同オブジェクト内であればインスタンス変数は見える、ということになるためです。先程の説明の通り、インスタンス変数はダイナミックスコープを持つため、オブジェクトのコンテキスト内であればどこからでも見えます。

これは Ruby の説明であり、こうした仕様、挙動は言語により大きく異なります。

名前空間上のオブジェクト指向の価値

オブジェクト指向は非常に細かく名前空間を分離するという特徴があります。

例え極めて単純なオブジェクト指向 (例えば Lua にあるようなもの) であったとしても、名前の衝突を気にする必要があまりありません。

例えば、send や write といった動詞は、様々なケースにおいて使いたい動詞でしょう。これが手続き型言語である Perl では open はファイルを開くもの、と決まっており、別の何かを開く関数を定義しようとすると例えば socket_open や stream_open といった名前をつけていく必要があります。

しかしオブジェクト指向言語では、write, open, send, read といった汎用性のある動詞を、オブジェクトに所属させることでオブジェクトとして適切な意味で動作させることができます。Ruby で言うと File クラスの open メソッドはファイルを、Dir クラスはディレクトリを開き、TCPServer クラスでは TCP サーバーとして接続を待ち受け、TCPSocket クラスでは TCP クライアントとして TCP/IP 接続を行います。

もし新たに Box クラスを書くことになったとき、箱を開くメソッドも衝突を恐れずに open メソッドとして定義することができます。

これはとくにプログラムのサイズが大きくなったり、多くのライブラリを使用する場合に名前の衝突の可能性

はどんどん上がっていきますが、オブジェクト指向を使うことで自然な名前を使いながら衝突を避けることができます。

クロージャ

クロージャの基本

クロージャは関数です。

クロージャはその関数が定義されたときのコンテキストでスコープを閉じ込めます。

これはどういうことでしょうか。Ruby では `Proc` と呼ばれる関数オブジェクトがクロージャになっています。`Proc` オブジェクトを定義する方法はいくつかありますが、ここでは `Kernel.lambda` メソッドを使ってみましょう。

```
lambda {
  puts "Hello, closure"
}
```

`Proc` オブジェクトは `call` メソッドによりその関数を実行することができます。

```
closure = lambda {
  puts "Hello, closure"
}

closure.call
```

Ruby のローカル変数のスコープはブロックです。ブロックの外側ではメソッドがスコープになります。

```
name = "Haruka"

def get_name
  name = "Bobby"
end

name  # Haruka
```

さて、ではメソッドの中で定義された変数をブロックの中にあるクロージャで使ってみましょう。

```
def greeting
  name = "Bobby"
  labmda { printf("Hey, %s\n", name) }
end
```

このままでも使えるのですが、わかりやすく `return` してみましょう。`return` はメソッドの実行をそこで終了し、引数として与えられた値をメソッドの返り値にします。

```
myname = "Haruka"
```

```
def greeting
  name = "Bobby"
  return labmda { printf("Hey, %s\n", name) }
end

closure = greeting  # greeting の戻り値の Proc オブジェクトが入る
```

では呼び出してみましょう。

```
printf("This is %s\n", myname)
closure.call
```

```
% ruby ex5.rb
This is Haruka
Hey, Bobby
```

Proc オブジェクトの中では name というローカル変数が使用されていますが、これはメソッドの中がスコープであり、メソッドの外側で参照するとエラーになります。しかし、Proc オブジェクトはクロージャであるため、自分が定義されたメソッドのスコープを持っています。そのため、自分が定義されたときに有効なスコープ内で存在した変数 name を参照できるわけです。

JavaScript の関数もクロージャです。これを使うとこんなことができます。

```
var obj = (function() {
  var ruka = {
    "name" : "Haruka",
    "belong" : "Mimir Yokohama"
  }
  return ruka
})()

obj.name   #Haruka
```

少し複雑に見えるかもしれません。分解して見ていきましょう。

まず

```
( ... )
```

というのは、それをひとまとまりの値として表現するためのものです。ここでは

```
(
  function() {
    // ...
  }
)
```

となっていますから、これで匿名関数オブジェクトになります。

関数オブジェクトは次のように呼び出せます。

```
var func = function() { console.log("Hey!") }
func()
```

関数オブジェクトに () をつけることで呼び出せるわけですから、

```
(
  function() {
    // ...
  }
)()
```

とすれば定義したばかりで変数にも格納していない匿名関数オブジェクトを呼び出すことができます。

JavaScript において、関数オブジェクトの戻り値は関数内で return した値か、(何も return しなければ)undefined です。ここでは return ruka としていますから、関数内で定義された変数 ruka を返していますね。

JavaScript では宣言なしに定義された変数はグローバル変数になりますが、var キーワードによって宣言された場合ローカル変数になります。このローカル変数のスコープは関数です。JavaScript は少し特殊な点があり、ローカル変数は自身が宣言される関数全体に対してスコープを持ちます。普通は宣言された場所より後ろ側でのみ有効なスコープを持つのですが、JavaScript はどの位置で宣言したとしても関数全体が有効なスコープになります。

さて、ruka は変数にも格納されない匿名関数の中をスコープとする変数です。ですから単に

```
var obj = (function() {
  var ruka = {
    "name" : "Haruka",
    "belong" : "Mimir Yokohama"
  }
})
```

としてしまうと、この変数 ruka を参照することは決してできません。変数 obj の中身は関数オブジェクトであり、この関数オブジェクトを呼び出すことはできますが、その中で定義されているローカル変数にはアクセスできません。

もちろん、このローカル変数そのものを return するようにすればアクセスできます。

```
var obj = (function() {
  var ruka = {
    "name" : "Haruka",
    "belong" : "Mimir Yokohama"
  }
  return ruka
})
```

この場合、obj を呼び出すと匿名関数内のローカル変数 ruka が返りますから、そこからアクセスできます。

```
obj().name   // Haruka
```

動的な値だとこの意味がより明確になります。次のコードでは document.getElementById() を使って DOM エレメントを取得していますが、この DOM エレメントそのものには直接アクセスすることができません。ただし、return される関数オブジェクトを呼び出すことでこの DOM エレメントの値を操作することができます。

```
const formSetter = (function() {
  var textform = document.getElementById("TextLine")
  return function(str) {
    textform.value = str
  }
})()
```

最新の JavaScript ではこんなトリッキーなことをしなくて良いように let キーワードが追加され、ブロックローカルな変数を宣言することができます。

ネストしたローカルスコープ

Perl では自分を内包する外側にスコープがある変数はネストしたスコープ内でも有効です。

```
my $x = "A";
my $f = sub {
  $x = "B"
};

$f->();

print $x;
```

```
% perl ex7-1.pl
B
```

ただし、my 宣言は現在のスコープでローカルスコープを生成しますから、ネストしたスコープ内で my 宣言を行うと、その変数はネストしたスコープに閉じ込められます。

```
my $x = "A";
my $f = sub {
  my $x = "B"
};

$f->();

print $x;
```

```
% perl ex7-1.pl
A
```

別の言い方をすると、外側のスコープの変数は直接見えているので、内側のスコープから外側のスコープに値を持ち出すことができます。

Ruby の場合変数宣言がないため、これは常に適用されます。そのため

```
10.times do |i|
  x = i
end

puts x
```

とすると、定義されていない変数 x を参照するので例外が発生します。一方、

```
x = nil

10.times do |i|
  x = i
end

puts x
```

のようにその変数が外側のスコープで登場している場合、ネストしたスコープであるブロック内の値が持ち出され、x は 9 になります。(Integer#times は 0 からカウントします)

Ruby の場合、ローカル変数を宣言する方法がないため、外側のスコープで同じ名前の変数が使われるとネストしたスコープの中で外側のスコープの変数を破壊することになります。しっかり命名規則をもってネストしたスコープで衝突しないように注意する必要があります。

これは言い換えると、クロージャにおいてもクロージャが閉じ込めた変数のスコープがどの範囲かを気にする必要があるということでもあります。クロージャは自動的にネストしたスコープのローカル変数を生み出すわけではありません。

```
i = 0

succ = ->() { i += 1 }

i = 5

succ.()
succ.()
succ.()
i       # 8
```

もっとも、ネストしたスコープのローカル変数として変数をコピーした場合はその変数を閉じ込めることができます。

```
my $i = 0;

my $f = sub {
  my $i = $i;
  return sub { ++$i; };
```

```
}

my $succ = $f->();

my $i = 10;
$succ->(); # 1
$succ->(); # 2
$succ->(); # 3
$i         # 10
```

Ruby の場合、外側のスコープのローカル変数と同名のネストしたスコープのローカル変数を使うことはできないため、Ruby ではできないことです。

クロージャと関数

以前、関数とサブルーチンは異なるものである可能性があるということを示しました。

これは、明確に「クロージャであるものを関数と呼ぶ」という考え方があるためです。また、クロージャである関数を Lisp での呼び方を拝借して「ラムダ式」と呼ぶ場合もあります。

些細なことですが、Ruby は関数オブジェクトは `Proc` オブジェクトであり、`lambda` オブジェクトとは違いがあります。もっとも、`Kernel#lambda` によって作られるものは `Proc` オブジェクトであり、特別な属性を持つ `Proc` オブジェクトであるとみなすことができます。

ただし、実際にはメソッド呼び出し時のブロックも、関数コンストラクタ (`->() { ... }`という構文) も `lambda` `Proc` オブジェクトを生成するため、純粋な `Proc` オブジェクトを生成するには `Proc.new` を使うしかなく、滅多に使うことはありません。

変数の宣言

変数の宣言の仕方やその意味は言語によって異なります。

Perl

Perl の場合、宣言によって変数のスコープが生成されます。宣言しない場合、変数はグローバル変数になります。

非 strict 環境では変数宣言はオプショナルであり、しなくても構いません。その場合変数はグローバル変数となり、未宣言の変数の参照は文字列コンテキストでは空文字列、数値コンテキストでは `0` になります。

local

`local` 宣言はダイナミックスコープを持つ値を生成します。これはグローバル変数をオーバーライドするものであり、`local` で宣言されたスコープを抜けるとグローバル変数の値が見えるようになります。

local 変数は Perl4 から存在します。

my

`my` 宣言はレキシカルスコープを持つ変数を宣言します。Perl5 から導入されました。

スコープはブロックになります。

our

`our` 宣言はレキシカルスコープを持つ変数を宣言します。Perl5 から導入されました。

スコープは my 変数よりも広く、パッケージになります。

state

`state` 宣言はレキシカルスコープを持つ変数を宣言します。Perl 5.10 から導入されました。

state 変数のスコープはブロックですが、そのブロックにおいて最初期化されない、という特徴があります。例えば次のようなコードがあるとします。

```
use 5.10.0;
foreach $i (1..9)  {
    state $xs = $i;
    my $xm = $i;
    print $xs, ":", $xm, "\n";
}
```

`state` 宣言も `my` 宣言もこのループの中で繰り返し実行されます。しかし、`my` 宣言はループするたびに初期化されるのに対し、`state` 宣言は `1` が代入されたあと、再び初期化されることはなく、次のような結果になります。

```
% perl ex9.pl
1:1
1:2
1:3
1:4
1:5
1:6
1:7
1:8
1:9
```

JavaScript

JavaScript の場合、宣言によって変数のスコープが生成されます。宣言しない場合、変数はグローバル変数になります。

変数宣言はオプショナルであり、しなくても構いません。その場合変数はグローバル変数となり、未宣言の変数の参照は `undefined` になります。

var

var 宣言は関数をスコープとするレキシカル変数を宣言します。

宣言の位置にかかわらず、所属する関数全体で有効になります。

また、var はトップレベルで宣言した場合、window オブジェクトのプロパティの定義となります。

let

let 宣言はブロックをスコープとするレキシカル変数を宣言します。

var と違い、宣言された位置で初期化され、関数全体をスコープにするようなことはありません。

const

const 宣言はブロックをスコープとするレキシカル定数を宣言します。

定数としては珍しく、ブロックローカルなスコープを持ちます。スコープは let と同じです。

プロパティ

オブジェクトに属する値としてプロパティがあります。

プロパティを定義するには、オブジェクトをレシーバとしてプロパティの代入を行うだけです。宣言や定義は必要なく、オブジェクトはオープンであり、いつでもプロパティを代入でき、初めてプロパティが代入されたとき、オブジェクトにその名前のプロパティが追加されます。

オブジェクトの初期化時には this キーワードを使ってオブジェクトを指定できます。

また、オブジェクトリテラルで指定した場合、左辺が文字列であるか生のキーワードであるかによらず同名のプロパティが作成されます。

Ruby

Ruby には変数宣言がありません。宣言なしに代入することができ、その種別は名前によって決定され、スコープは種別によって決まります。

宣言がないため、ネストしたスコープを作ることはできません。

未定義の変数を参照した場合、NameError になります。ただし、これは必ず代入が実行されなければならないという意味ではなく、Ruby 処理系が変数として名前を認識していなくてはならないという意味になります。例えば

```
p a
```

とすると、変数 a が定義されていないため NameError になりますが、

```
false and a = 10
```

```
p a
```

の場合、`a = 10` が実行されることは決してないにもかかわらず、`nil` になります。

宣言がなく、名前が変数として認識される必要があることから、初期化は必須です。

ローカル変数

ローカル変数はアルファベット小文字、あるいは_から始まる名前です。

ブロックローカルな変数となります。

グローバル変数

グローバル変数は$で始まる名前です。

定数

定数は大文字で始まる名前です。

スコープはグローバルですが、名前空間はネストされるため、クラスやモジュール内で宣言された場合や、明示的に暮らすやモジュールの名前空間を指定された場合はクラスやモジュールの定数になります。

インスタンス変数

@で始まる変数はインスタンス変数です。

インスタンス変数はオブジェクトに所属し、ダイナミックスコープを持ちます。

クラス変数

@@で始まる変数はクラス変数です。

クラス変数はクラスにローカルスコープを持つ変数です。そのスコープは複雑ですが、`instance_eval` からクラス変数が見えないのは仕様とのことです。

Lua

Lua は変数宣言は省略できます。この場合、グローバル変数になります。

```
v = nil

function hi()
  v = "Hello"
end

print(v)
hi()
print(v)
```

`local` 宣言によってローカル変数にすることができます。

```
v = nil

function hi()
  local v = "Hello"
end

print(V)
hi()
print(v)
```

Zsh

Zsh では変数に型があり、その型によっては宣言が必要になります。

まず普通に変数を使うことはできます。

```
v=Hello
```

この場合グローバル変数になります。

```
hi() {
  v=Hello
}

hi
print $v
```

`typeset` によって宣言すると変数は関数ローカルなレキシカルスコープを持ちます。

```
hi() {
  typeset v=Hello
}

hi
print $v
```

`typeset` は変数を宣言するための共通のコマンドですが、分かりやすく `local` というコマンドもあります。

```
hi() {
  local v=Hello
}

hi
print $v
```

`typeset` の`-g` オプションを使うと明示的にグローバル変数にできます。

```
hi() {
  typeset -g v=Hello
}

hi
print $v
```

typeset の-x オプションを使うと環境変数になります。

```
hi() {
  typeset -x v=Hello
}

hi
print $v
```

変数定義が () で囲まれている場合、配列になります。

```
a=(a b c)
```

typeset -a によって明示的に配列を定義することもできます。

```
typeset -a a=(a b c)
```

typeset -A を使えば連想配列を定義できます。この書式は配列と同じで、連想配列を使うには typeset を使わねばなりません。

```
typeset -A a=(a Apple b Banana c Cherry)
```

typeset -i あるいは integer によって整数型の変数を宣言できます。arithmetic evalution では常に変数は数値型であるとみなされますが、Zsh の整数型は基数を持つことができるため、文字列との変換は単純なものではありません。

```
typeset -i i=100
```

typeset -E で浮動小数点数を定義できます。typeset -f は関数なので注意してください。

typeset -T でタイ変数を定義できます。これは、スカラー変数と配列変数を結びつけ、スカラー変数が配列を:でつないだものに見えるようにするものです。これは Zsh における$path と$PATH の関係と同じです。

例えば次の例では

```
typeset -T NAME name

name+=(joe john jonny)

print $NAME
```

```
% zsh ex16.zsh
joe:john:jonny
```

となります。

Zsh には他にも様々な変数型や属性があります。

Dart

Dart はオプショナルな型を持ちます。

通常、次のようにどのように変数であれ `var` によって宣言できます。

```
var name = "Haruka";
```

変数がどのような型を持つのかというのは型推論によって決定されます。しかし、明示的に型を指定して宣言することもできます。

```
String name = "Haruka";
```

Dart の型システムは非常に緩やかなので、宣言と矛盾した型の値を入れたとしても通ります。

```
int name = "Haruka";
```

静的型と動的型の大きな違いとして、静的型においては変数に型を宣言した以上、その名前はその型のために確保されるものであり、その名前に対して他の型の値は入らないという特徴があります。しかし、Dart は本質的には動的型の言語なので、そのような制約はありません。しかし、`var` 宣言では型推論によって型が確定されるなんらかの変数ということになります。

そこで、「途中が型が変わりうる変数」の宣言として `dynamic` があります。

```
dynamic myid = 1505;
myid = "Haruka";
```

また、全ての値が `Object` 型から派生しているため、`Object` 型であるとすれば全てをカバーできます。

```
Object name = "Haruka";
```

C

C は静的型付けの言語であり、変数は常に型を伴って宣言されねばなりません。つまり、いきなり

```
name = "Haruka";
```

とやるとエラーになります。

```
int year;
year = 2020;
```

のように予め変数に対して型を明示する必要があります。

さて、先の例では文字列だったのにここでは整数になりました。なぜなら C での文字列はそんなに簡単ではないからです。文字列=バイト列であるのであれば、char 配列に格納することで実現できます。

```
char name[7];
name[0] = "H";
name[1] = "A";
name[2] = "R";
name[3] = "U";
name[4] = "K";
name[5] = "A";
name[6] = "\0";
```

最近は文字列処理が得意な言語が多いのでピンとこないかもしれませんが、Perl は非常に文字列を得意とする言語です。Java はあまり文字列が得意ではありませんが、それでも当時としてはまだまともに扱えるほうです。

Go

Go は型推論を持つ静的型付けの言語です。

通常は次のように宣言します。

```
var name string
name = "Haruka"
```

C 言語よりは楽ですね。最近はパースが楽であるという理由で型の名前が後ろにくる言語が結構あります。

一方、暗黙の型を使う場合、:=という代入を行います。この場合、型推論が働くため宣言は不要です。

```
name := "Haruka"
```

しかし、この方法が使えるのはリテラルだけなので、そこまで楽になるわけではありません。

それなら Perl などに近い感覚だと思うかもしれませんが、実際は静的型なので結構な制約があります。それが顕著なのが Map(連想配列) です。

Go の Map は次のように宣言します。

```
var haruka = map[string]string{"Name":"Haruka", "Belong":"Mimir Yokohama"}
```

見ての通り、キーも値も型が限定されるため、自由に情報を追加することができません。このことから、Go で Map はあまり使わず、むしろ構造体をよく使います。

また、配列も型は固定ですし、さらに長さも固定です。

```
var entry[3] string
entry[0] = "Haruka"
entry[1] = "Bobby"
entry[2] = "Erina"
```

おまけ

Perl や JavaScript のように「変数のスコープの種類を宣言で使い分ける」という言語はかなり稀で、それらの多くは言語の進化の過程で変数のスコープの変更を余儀なくされた言語です。

おそらく、Perl は最初から `my` があれば `our` や `local` は登場しなかったでしょうし、JavaScript も最初から `let` があったならば `var` は登場しなかったはずです。

このようなケースにおいて「新しく登場したものに統一したほうが良い」と言えるかどうかは微妙です。もちろん、「導入するのが良い」と考えたからこそその機能が登場したわけですが、同時にそれは言語に求められるものが変化したからそうなったのだとも言え、その言語らしさが損なわれているのだとも言えます。Perl で言えば web の普及で Perl が使われるようになったために大規模なプログラムや分割パッケージが求められてシンプルに処理することができなくなり、JavaScript はより高機能で strict な言語であることが求められたために動的でミニマムな言語でいることができなくなりました。

一般にグローバル変数は「悪いものである」とされがちですが、実際には何の問題も生じないのであればグローバル変数を使うことは何も悪くなく、グローバル変数を使うことで簡潔・明快に書けるのであれば複雑な書き方でそれを避けるよりも使うほうが良いでしょう。

これは GOTO にも言えます。GOTO を避けることは宗教となり、なんとしても使ってはならないとされますが、それで GOTO を使えば自然に書けるものを非常に不自然なコードを書いてまで避けるのは滑稽です (GOTO がない代わりに他の帯域脱出の方法やコンテキストスイッチ方法が用意されている言語もあります)。

Perl の our 変数はグローバル変数を支障なく書きやすい、なかなか優れた仕組みであると言えます。グローバル変数による名前の衝突を、最小の変更で避けるための苦肉の策とも言えますが、当該パッケージ、つまりはそのファイルの中で衝突しなければ問題ないということになるため、our 変数を使って問題を生じることは稀です。