2015年10月27日

プログラミング入門

型

横浜国立大学

倉光君郎

型

型(type)は、プログラミング言語の発展とともに進化してきた理論と技法である。今日、ソフトウェア信頼性やソフトウェアのセキュリティなどの重要な基盤となっている。
ここでは、まず型とは何か学んでみよう。

値と種類

前回は、簡単な数式を試してみた。その中でいくつか変わったことに気づいたかもしれない。
例えば、7/2 を評価したとき、3.5 にならなく、3となる。

```
>>>
7/2
<<<
3
```

なぜ？

コンピュータ上では、自然数 と 実数が区別されているためである。ちなみに、自然数のことを整数、実数のことを浮動小数点数と呼ぶ。

基本型

型は、このような値の種類を区別するために使われる。プログラミング言語で表現するデータ構造の分だけ、型の種類がある。
まず最初に覚える型は次のとおりである。

- int – 整数値を表す型
- boolean – 論理値を表す型
- double – (倍精度の)浮動小数点数を表す型

また、プログラミング言語によって、サポートし

ている型の種類は異なる。Konoha は、Java 言語をベースにした型を採用している。ただし、他の言語では、例えば、boolean 型を bool 型と書いたり、それぞれ少し名称が異なる場合がある。しかし、上記の型の概念は一般的であり、ほとんど全ての言語で普遍的にあらわれるものである。

整数

整数は、コンピュータ上で表現される値の中でもっとも基本的な値である。コンピュータ内の2進数の表現と対応しており、8ビット,16ビット, 32ビット, 64ビットと処理の単位にあわせて大きさがある。

- `byte` - 8ビット整数
- `short` - 16ビット整数
- `int` - 32ビット整数

- `long` - 64ビット整数

32ビット整数は、2^{32} までの数値を表現できる。10進数なら、0 〜 4294967295に相当する。ただし、符号を2の補数表現で表現するため、通常は-2147483648 から $+$2147483647の範囲である。

整数オーバーフロー

32ビット整数は、日常レベルで十分に大きな値を扱うことができるが、より大きな数値を扱うことができない。範囲を超えてしまうとどうなるか？

```
>>>
2147483647+1
<<<
-2147483648
```

このように負の数に反転してしまう。これを整数オーバフロー (integer overflow) と呼ぶ。これはプログラミング言語では、標準的なふるまいであり、ユーザは反転しないように注意することになっている。プログラミングをするときは、常に自分の扱っている数値の上限を意識しなければならない。
より大きな整数を扱いときは、64 ビット長の long を使う。ただしこちらも大きさには限界がある。ソフトウェアで工夫すれば、無限長の整数を表現することができる。

boolean 型: 論理値

論理値は、真(true)、偽(false) を表す2値である。boolean 型で型付けされている。
次のようなリテラルで表現される。

```
true
false
```

プログラミング言語によっては、boolean 型が存在しない言語もある。代わりに、整数値の1 と 0 で代用することがある。しかし、概念として論理

値をもたない言語は存在しない。

double 型: 浮動小数点数

浮動小数点数は、IEEE754規格にもとづく、コンピュータ内の小数点数の表現である。多くのプログラミング言語において、次の2種類の型が用意されている。

- float 型 - 単精度浮動小数点数
- double 型 - 倍精度浮動小数点数

重要なのは、double 型である。float 型は、実用上、十分な精度が得られないことも多いため、

double 型が一般的に利用される。Konoha は、float 型と書いても double 型と解釈し直す。浮動小数点数は、整数と区別するため、明示的に 0.0 のように小数点以下記述する。もしくは、e もしくは E を含めることで指数表現することもできる。

```
3.14
314e-2
```

浮動小数点数は、整数とおなじく、四則演算と比較演算を使うことができる。ビット操作演算は用

意されていない。

```
>>>
7.0 / 2.0
<<<
3.50
```

浮動小数小数点数は、いわゆる実数とは異なる。
特に、丸め誤差などの注意が必要である。

演算子

演算子は、各種の演算をあらわす記号のことである。四則演算の＋や-は演算子である。
型ごとに、以下のような演算子が定義されている。

- 四則演算 + - * / %
- ビット演算 << >> & | ^ ~
- 比較演算 == != < <= > >=
- 論理演算 && || !

四則演算

整数は、四則演算、ビット演算、比較演算を行うことができる。

- 加算 `x + y`
- 減算 `x - y`
- 乗算 `x * y`
- 除算 `x / y`
- 余算 `x % y`

ビット演算

ビット演算子とは、ビット単位のデータ操作を行う演算子であり、マルチメディア処理等でよく利用される。

- AND `x & y`
- OR `x | y`
- XOR `x ^ y`
- **左シフト** `x << y`
- **右シフト** `x >> y`

- **補数 `~x`**

これらの演算子は、`x` と `y` に整数で型付けされた式をとり、評価すれば整数値となる。評価すると、整数になる式は整数型で型付けされているという。

比較演算子

比較演算子は、整数の大小を評価するときに使う。

- `x == y` – 等しい
- `x != y` – 等しくない
- `x < y` – より小さい
- `x > y` – より大きい
- `x <= y` – 以下
- `x >= y` – 以上

等価$(==)$と代入$(=)$はよく似ているが、一文字違えば別の演算子である。（間違えないように）

比較演算の結果

比較演算子は、その評価値がより興味深い。四則演算やビット演算と異なり、評価値は整数ではない。論理値と呼ばれる、真(true)、偽(false) を表す2値である。

```
>>>
1 < 2
<<<
true
```

論理演算

論理演算子は、論理式 x, y に対して、次のように定義されている。もちろん、論理演算による式も 論理式となる。

- `x && y` − x かつ y
- `x || y` − x または y
- `!x` − x でない

条件演算

条件演算は、三項演算子ともよばれ、論理式といっしょに用いて条件式を構成する。

＜論理式＞ ? ＜式1＞ : ＜式2＞

＜論理式＞が真(true)のとき、＜式1＞が評価され、そうでなければ＜式2＞が評価される。たとえば、次の条件式は x と y のうち大きな数値がえられる。

```
x > y ? x : y
```

型強制とキャスト

プログラミング言語では、整数と浮動小数点数、論理値をそれぞれ異なる型で区別していることを理解したと思う。もし両者を混在して利用したらどうなるのだろうか？

型エラー

例えば、整数値と論理値を加算すると、演算子のパラメータの型が一致しない。このような状況を「型エラー」と呼び、評価することができない。

```
>>>
1 + true
(<stdio>:1) [error] [型エラー] 二項演算子の型(パラメータ)が一致しません
```

型強制

同じ方針でいくのなら、int **型と**double **型の加算も型エラーになるはずである。ところが、こちらは正しく動作してくれる。**

```
>>>
1 + 2.0
<<<
3.0
```

これは、型強制と呼ばれる自動変換が働いてくれたためである。
int 型とdouble 型の加算は、仮に型が異なっていても評価してくれた方が直感的である。プログラミング言語は、型エラーをさけるため、int 型の数値をdouble 型に変換してから double 型どうしの加算として評価している。

型変換(キャスト)

なお、自動変換ではなく、ユーザが明示的に型変換を行いたい場合は、次のように式の前に(型名)で囲んだキャスト演算子を用いる。

```
>>>
(double)1
<<<
1.0
```

変数と型

プログラミングの世界では、変数に型付けすべきかしないべきか、長らく大きな論争となってきた。

- 静的型付け派 – 変数に型付けすべき
- 動的型付け派 – 変数に型付けすべきでない

静的型付けは、C/C++言語, Java言語, C#言語など主流プログラミング言語で採用されている。一方、動的型付けは、JavaScript, Ruby, Python,

PHP など、スクリプト言語とよばれる書きやすさ重視する言語で愛用されている。

動的型付けによる変数

前回、変数を使ったときは、変数に型付けすることなく使用した。

```
x = 0
```

つまり、変数 x にはどのような型の値でも代入可能である。今回、紹介するのは、変数に型付けする方法である。

変数宣言

変数に型付けすることを変数宣言という。次のように 型、変数名、そして初期値のように書く。

```
int x = 0;
```

これで変数 x は、int 型で型付けされた。もう int 型の値しか代入することはできない。

変数宣言と型エラー

試しに違うboolean 型のtrueを代入してみよう。

```
>>>
x = true
(<stdio>:2) [error] 型エラー: int
```

型が必要です.

このように変数の型付けは、変数操作に対して型が正しいかどうか検査できるようにする。これを型検査とよぶ。

型検査のおかげで、ユーザはより間違いを発見しやすくなる。一方、常に型を意識して書く必要があり、多少、手間が増える。
ちなみに、Konoha **は、教育プログラミング言語なので、両者を体験できるように、静的型付けも動的型付けも両方とも採用している。ただし、静的型付けを推奨している。**