Nimです、こんばんは

14:00〜15:10 セッションs5b

自己紹介

浅田睦葉

- 🎓 筑波大学情報学群情報科学類 1年
- 📚 『プログラミングNim』（インプレスR&D, 2021）著者
- 📌 mock-up 開発
- 🐣 Twitter: @momeemt

自己紹介

- Nimが好きで3年半くらい書いています

自己紹介

- Nimが好きで3年半くらい書いています
- メタプログラミングやNimコンパイラに興味があります

自己紹介

- Nimが好きで3年半くらい書いています
- メタプログラミングやNimコンパイラに興味があります
- 組込みプログラミングは初学者なのでお手柔らかにお願いいたします 🙇‍♂️

自己紹介

- Nimが好きで3年半くらい書いています
- メタプログラミングやNimコンパイラに興味があります
- 組込みプログラミングは初学者なのでお手柔らかにお願いいたします 

▲ 表紙がえらいかわいい2021

▲ 表紙がえらいかわいい2022

Nimとは？

- Andreas Rumpf 氏が2008年から開発を続けているシステムプログラミング言語

Nimとは？

- Andreas Rumpf 氏が2008年から開発を続けているシステムプログラミング言語
- 複数の言語をバックエンド言語に持つ静的型付けのコンパイラ言語

Nimとは？

- Andreas Rumpf 氏が2008年から開発を続けているシステムプログラミング言語
- 複数の言語をバックエンド言語に持つ静的型付けのコンパイラ言語

```nim
from strutils import parseInt

proc fizzbuzz (num: int): string =
  if num mod 15 == 0:
    result = "FizzBuzz"
  elif num mod 3 == 0:
    result = "Fizz"
  elif num mod 5 == 0:
    result = "Buzz"
  else:
    result = $num

when isMainModule:
  stdout.write "Number: "
  let input = stdin.readLine.parseInt
  for i in 1 .. input:
    echo fizzbuzz(i)
```

▲ FizzBuzz

Nimとは？

- Andreas Rumpf 氏が2008年から開発を続けているシステムプログラミング言語
- 複数の言語をバックエンド言語に持つ静的型付けのコンパイラ言語

```nim
from strutils import parseInt

proc fizzbuzz (num: int): string =
  if num mod 15 == 0:
    result = "FizzBuzz"
  elif num mod 3 == 0:
    result = "Fizz"
  elif num mod 5 == 0:
    result = "Buzz"
  else:
    result = $num

when isMainModule:
  stdout.write "Number: "
  let input = stdin.readLine.parseInt
  for i in 1 .. input:
    echo fizzbuzz(i)
```

▲ FizzBuzz

```nim
type
  UserObj = object
    id: int
    name: string
  User = ref UserObj

var user = User(id: 1, name: "momeemt")

# ref 0x10425b050 --> [id = 1, name = 0x10425c060"momeemt"]
echo user.repr

# (id: 1, name: "momeemt")
echo user[]
```

▲ 構造体とその参照

Nimで出来ること

1. Web開発

- JavaScriptとの相性が良く、Webバックエンドフレームワークの開発も盛ん

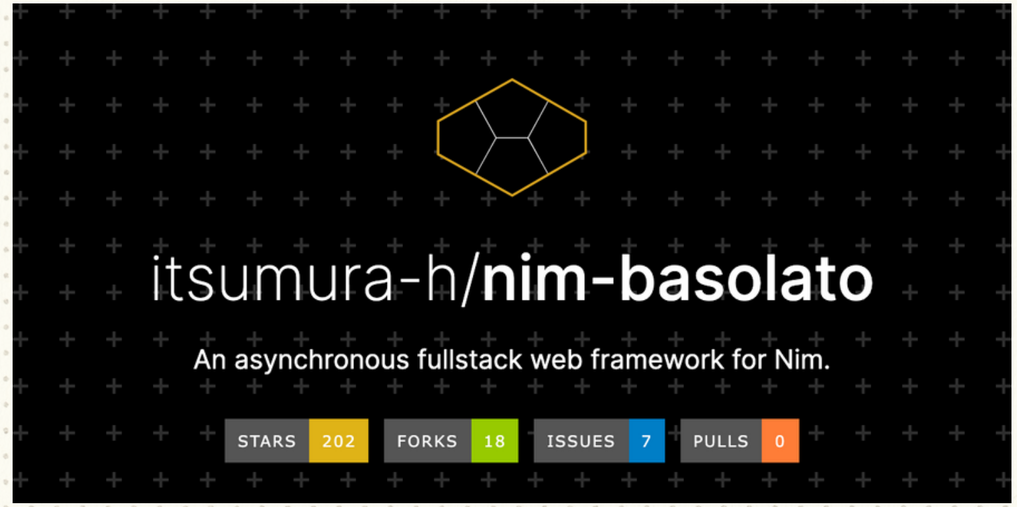
▲ Webフロントエンドフレームワーク

▲ Webバックエンドフレームワーク

Nimで出来ること

1. Web開発

・日本人作者のWebバックエンドフレームワーク Basolato



▲ フルスタックフレームワーク

・高性能な非同期HTTPサーバー
・非同期クエリビルダ allographer
・開発支援 CLIツール

... etc

Nimで出来ること

2. CLIツール

・メタプログラミングを活かしたオプション解析ライブラリ cligen が非常に優秀

▲ オプション解析ライブラリ

Nimで出来ること

2. CLIツール

・オプション解析ライブラリ cligen が非常に優秀

▲ オプション解析ライブラリ

```nim
import cligen

proc command (foo = false, bar = false): int =
  echo (foo, bar)
  return 0

when isMainModule:
  dispatch(command, short = {"bar":'B'})
```

プロシージャを渡すだけで◎

Nimで出来ること

2. CLIツール

・オプション解析ライブラリ cligen が非常に優秀

▲ オプション解析ライブラリ

```nim
import cligen

proc command (foo = false, bar = false): int =
  echo (foo, bar)
  return 0

when isMainModule:
  dispatch(command, short = {"bar":'B'})
```

プロシージャを渡すだけで◎

↓

メタプログラミングによる恩恵

Nimで出来ること

3. DevOps

- NimのサブセットであるNimScriptがNimVM上で動作する
  - クロスプラットフォームで動作
  - Nimでサポートされる強力なメタプログラミングをそのまま扱える
  - taskが便利

Nimで出来ること

3. DevOps

- NimのサブセットであるNimScriptがNimVM上で動作する
  - クロスプラットフォームで動作
  - Nimでサポートされる強力なメタプログラミングをそのまま扱える
  - taskが便利

▲ こちらの記事がおすすめ

Nimで出来ること

4. OS開発

- 生ポインタ操作が可能でスタンドアロンなバイナリを生成できるのでOSを開発できる

Nimで出来ること

4. OS開発

- 生ポインタ操作が可能でスタンドアロンなバイナリを生成できるのでOSを開発できる
  - Nimで開発された実験的なカーネル（2013-）
    - https://github.com/dom96/nimkernel

Nimで出来ること

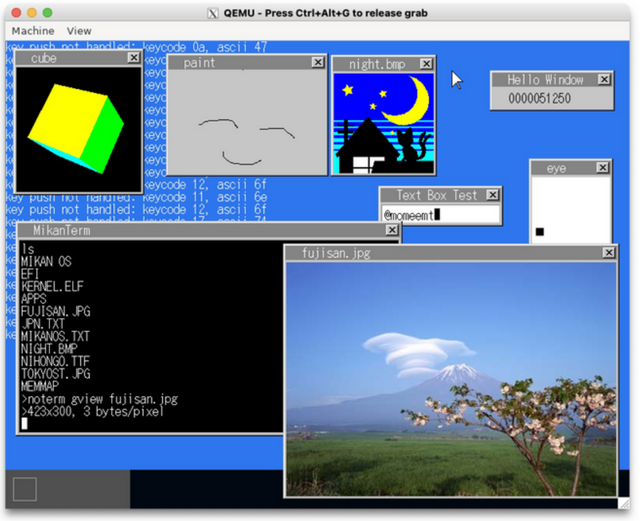
4. OS開発

- 生ポインタ操作が可能でスタンドアロンなバイナリを生成できるのでOSを開発できる
  - Nimで開発された実験的なカーネル（2013-）
    - https://github.com/dom96/nimkernel
  - NimでOS開発したいねという話 by sksatさん（2017）
    - main関数をブートローダーから呼び出してQEMUで動作させた

Nimで出来ること

4. OS開発

- 生ポインタ操作が可能でスタンドアロンなバイナリを生成できるのでOSを開発できる
  - Nimで開発された実験的なカーネル（2013-）
    - https://github.com/dom96/nimkernel
  - NimでOS開発したいねという話 by sksatさん（2017）
    - main関数をブートローダーから呼び出してQEMUで動作させた
  - NimによるmikanosのFFIの移植 by Double-oxygeNさん (2021)
  - NimによるmikanosのFFIの移植 by momeemt（2021）
    - @uchan_nos さんが開発した教育用OS mikanosを FFIで呼び出して叩いてインクリメンタルに置き換え

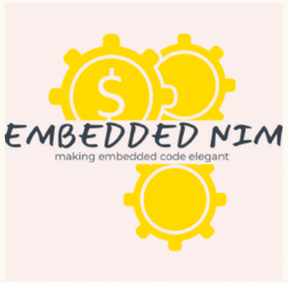


▲ 動作したがNim onlyではない

Nimで出来ること

5. 組込み開発

- スタンドアロンなバイナリが生成できるので理屈ではNimでも組込みプログラミングができる
  - C/C++とのFFIが洗練されており導入は他言語に比べればしやすい



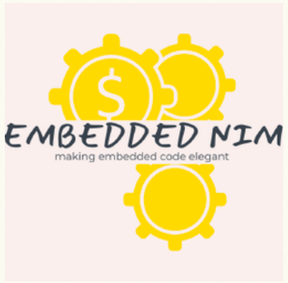
◀ @EmbeddedNim

- Raspberry PiのNimラッパーやドライバなど組込み関連のライブラリを多く管理している
- https://github.com/EmbeddedNim

Nimで出来ること

5. 組込み開発

- スタンドアロンなバイナリが生成できるので理屈ではNimでも組込みプログラミングができる
  - C/C++とのFFIが洗練されており導入は他言語に比べればしやすい



◀ @EmbeddedNim

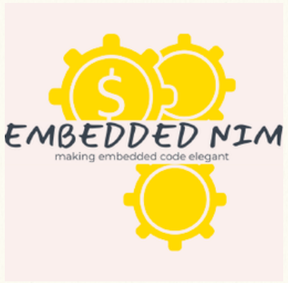
- Raspberry PiのNimラッパーやドライバなど組込み関連のライブラリを多く管理している
- https://github.com/EmbeddedNim

- ArduinoにNimを導入する実験は何人かの先人により行われている
  - Nim言語:AVR:Arduino Uno/Nano用にコンパイルしたときのメモ 2019/01
    - http://mpu.seesaa.net/article/463862262.html
  - Nim on Arduino
    - https://disconnected.systems/blog/nim-on-adruino/

Nimで出来ること

5. 組込み開発

- スタンドアロンなバイナリが生成できるので理屈ではNimでも組込みプログラミングができる
  - C/C++とのFFIが洗練されており導入は他言語に比べればしやすい



◀ @EmbeddedNim

- Raspberry PiのNimラッパーやドライバなど組込み関連のライブラリを多く管理している
- https://github.com/EmbeddedNim

- ArduinoにNimを導入する実験は何人かの先人により行われている
  - Nim言語:AVR:Arduino Uno/Nano用にコンパイルしたときのメモ 2019/01
    - http://mpu.seesaa.net/article/463862262.html
  - Nim on Arduino
    - https://disconnected.systems/blog/nim-on-adruino/

- C99が出力される + Rustほど高い学習コストを要しないので良いと考える意見もある

基本文法

```
1 from strutils import parseInt
2
3 proc fizzbuzz (num: int): string =
4   if num mod 15 == 0:
5     result = "FizzBuzz"
6   elif num mod 3 == 0:
7     result = "Fizz"
8   elif num mod 5 == 0:
9     result = "Buzz"
10  else:
11    result = $num
12
13 when isMainModule:
14  stdout.write "Number: "
15  let input = stdin.readLine.parseInt
16  for i in 1 .. input:
17    echo fizzbuzz(i)
```

▲ FizzBuzzで記述したプログラムを中心に説明します

基本文法

Nimを書けるようになりましょう！

```
proc name (param1: Type1, ..., paramN: TypeN): ReturnType =
  body
```

プロシージャ（procedure）

- 他言語における関数/メソッド/手続き が プロシージャ

基本文法

Nimを書けるようになりましょう！

```
proc name (param1: Type1, ..., paramN: TypeN): ReturnType =
  body
```

プロシージャ（procedure）

- 他言語における関数/メソッド/手続き が プロシージャ
- result変数 が特徴的
  - return文と異なり関数の実行を終了しない
  - たとえば動的配列を返す場合に便利

基本文法

Nimを書けるようになりましょう！

```
proc name (param1: Type1, ..., paramN: TypeN): ReturnType =
  body
```

プロシージャ（procedure）
- 他言語における関数/メソッド/手続き が プロシージャ
- result変数 が特徴的
  - return文と異なり関数の実行を終了しない
  - たとえば動的配列を返す場合に便利

```
proc square (arr: seq[int]): seq[int] =
  for elem in arr:
    result.add elem * elem

echo @[1, 2, 3].square
```

一時変数を用意しなくても、
result変数に直接戻り値を構成できる

基本文法

Nimを書けるようになりましょう！

UFCS（統一関数呼び出し構文）
- 元々はD言語が持つ糖衣構文
- Nimはクラスが無く関数呼び出し構文とメソッド呼び出し構文を同一視できる

```
proc greeting (name: string) =
  echo "Hello ", name, "!"

const EventName = "SWEST24"

EventName.greeting
EventName.greeting()
greeting(EventName)
greeting EventName
```

基本文法

Nimを書けるようになりましょう！

seq[T]とは

動的配列型で、実行時に配列サイズを変更可能
要素が増えるたびにポインタを確保している

```
echo @[1, 2, 3, 4, 5]
echo @['a', 'b', 'c']
echo @[true, false]
echo @["Nim", "C", "C++"]
```

Tは 型引数 で任意の型を入れることができる
seq[T]型は ジェネリクス と呼ばれる抽象型

基本文法

Nimを書けるようになりましょう！

seq[T]型とは

動的配列型: 実行時に配列サイズを変更可能
要素が増えるたびにポインタを確保している

```
echo @[1, 2, 3, 4, 5]
echo @['a', 'b', 'c']
echo @[true, false]
echo @["Nim", "C", "C++"]
```

Tは 型引数 で任意の型を入れることができる
seq[T]型は ジェネリクス と呼ばれる抽象型

array[I; T]型とは

静的配列型: コンパイル時に配列サイズが確定

```
echo [1, 2, 3, 4, 5]
echo ['a', 'b', 'c']
echo [true, false]
echo ["Nim", "C", "C++"]
```

Tは 型引数 で任意の型を入れることができる
Iも 型引数 で整数のコンパイル時定数が入る

基本文法

Nimを書けるようになりましょう！

```nim
from strutils import parseInt

proc fizzbuzz (num: int): string =
  if num mod 15 == 0:
    result = "FizzBuzz"
  elif num mod 3 == 0:
    result = "Fizz"
  elif num mod 5 == 0:
    result = "Buzz"
  else:
    result = $num

when isMainModule:
  stdout.write "Number: "
  let input = stdin.readLine.parseInt
  for i in 1 .. input:
    echo fizzbuzz(i)
```

if文 / if式

- Nimにおいて最も基礎的な条件分岐構文

基本文法

Nimを書けるようになりましょう！

```nim
from strutils import parseInt

proc fizzbuzz (num: int): string =
  if num mod 15 == 0:
    result = "FizzBuzz"
  elif num mod 3 == 0:
    result = "Fizz"
  elif num mod 5 == 0:
    result = "Buzz"
  else:
    result = $num

when isMainModule:
  stdout.write "Number: "
  let input = stdin.readLine.parseInt
  for i in 1 .. input:
    echo fizzbuzz(i)
```

if文 / if式

- Nimにおいて最も基礎的な条件分岐構文
- 厳格に **bool値** のみを受け取る

基本文法

Nimを書けるようになりましょう！

```nim
from strutils import parseInt

proc fizzbuzz (num: int): string =
  if num mod 15 == 0:
    result = "FizzBuzz"
  elif num mod 3 == 0:
    result = "Fizz"
  elif num mod 5 == 0:
    result = "Buzz"
  else:
    result = $num

when isMainModule:
  stdout.write "Number: "
  let input = stdin.readLine.parseInt
  for i in 1 .. input:
    echo fizzbuzz(i)
```

if文 / if式

- Nimにおいて最も基礎的な条件分岐構文
- 厳格に **bool値** のみを受け取る

```nim
if condition1:
  body1
elif condition2:
  body2
else:
  body3
```

▲if文

```nim
var data = if condition1:
             body1
           elif condition2:
             body2
           else:
             body3
```

▲if式

式の場合は必ず値を返す必要がある

基本文法

Nimを書けるようになりましょう！

when文 / when式

- コンパイル時における条件分岐構文
- 条件に満たすブロックのみコンパイルする
- 式の場合は必ず値を返す必要がある

```
when staticCondition:
  body1
else:
  body2
```

▲ when文

```
const OS = when defined(windows): "Windows"
           elif defined(macosx): "Mac"
           else: "Unix"
```

▲ when式

基本文法

Nimを書けるようになりましょう！

for文とは

```
for Identifier in Iterator:
  body
```

- Nimにおいて最も基礎的なループ文
- イテレータから値を取り出して識別子に代入し、プログラムを実行する
- イテレータから値を取り出し終えると終了する

基本文法

Nimを書けるようになりましょう！

イテレータとは

- ある集合構造を繰り返し、値を取り出すことができるインターフェース
- イテレータは第一級オブジェクト

```
 1 iterator fizzbuzz (limit: int): string =
 2   var index = 1
 3   while index <= limit:
 4     yield if index mod 15 == 0: "FizzBuzz"
 5           elif index mod 3 == 0: "Fizz"
 6           elif index mod 5 == 0: "Buzz"
 7           else: $index
 8     index += 1
 9
10 for text in fizzbuzz(100):
11   echo text
```

モジュールとパッケージ

洗練されたモジュールシステムとパッケージディレクトリ

```
import ModuleName
```

- モジュールシステムによりプログラムを機能によって分割できる
- 情報の隠蔽やファイルを個別にコンパイル可能にする

モジュールとパッケージ

洗練されたモジュールシステムとパッケージディレクトリ

```
import ModuleName
```

- モジュールシステムによりプログラムを機能によって分割できる
- 情報の隠蔽やファイルを個別にコンパイル可能にする

- アスタリスク（*）でマークすることで他のモジュールに公開
- そうでないシンボルは非公開になる

```
1 var
2   a = 0
3   b* = 1
4
5 let
6   c* = 2
7
8 proc f* () =
9   discard
```

モジュールとパッケージ

洗練されたモジュールシステムとパッケージディレクトリ

```
import MyModule except symbol1
from MyModule import symbol2
from MyModule import nil
```

- except節によって特定のシンボルを除く
- from節によって特定のシンボルのみをインポートする
- from節によって nil をインポートすると接頭辞を強制することができる
  - MyModule.symbol3

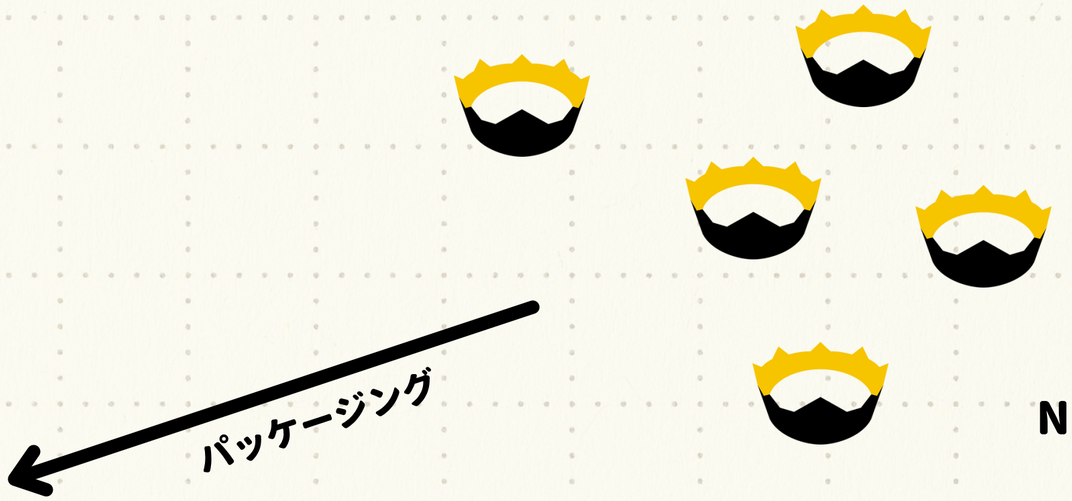
モジュールとパッケージ

洗練されたモジュールシステムとパッケージディレクトリ

Nimble (ビルドツール兼パッケージマネージャー)
Nimble Package Directory

モジュールとパッケージ

洗練されたモジュールシステムとパッケージディレクトリ



Nimで書かれたモジュール

Nimble (ビルドツール兼パッケージマネージャー)
Nimble Package Directory

```
$ nimble install <package-name>
```

パッケージのインストール

モジュールとパッケージ

Nimbleファイル

```
# Package
version       = "0.1.0"
author        = "Mutsuha Asada"
description   = "swest24"
license       = "MIT"
srcDir        = "src"
installExt    = @["nim"]
bin           = @["swest24"]

# Dependencies
requires "nim >= 1.6.6"
```

- パッケージのメタデータ、依存するパッケージとそのバージョン、タスクなどを記述する
- NimScriptが記述できる

モジュールとパッケージ

ビルドとパッケージ公開

```
$ nimble build
```

- Nimbleファイルが存在し、それにより規定されるディレクトリ構造を取るNimモジュールの集合を**パッケージ**と呼んでいる
- Nimbleパッケージはnimbleコマンドによってビルド/実行できる

モジュールとパッケージ

ビルドとパッケージ公開

```
$ nimble build
```

- Nimbleファイルが存在し、それにより規定されるディレクトリ構造を取るNimモジュールの集合を**パッケージ**と呼んでいる
- Nimbleパッケージはnimbleコマンドによってビルド/実行できる

```
$ nimble publish
```

- 開発したバイナリ/ライブラリは次のコマンドにより開始される対話により公開できる

Nimの型

整数型 / 浮動小数点数型

符号付き整数	符号なし整数	浮動小数点数
int型	uint型	float型
int8型	uint8型	float32型
int16型	uint16型	float64型
int32型	uint32型	
int64型	uint64型	

Nimの型

真偽値 (bool型)

関係演算子	論理演算子
<, <=	and
>, >=	or
==, !=	xor
	not

Nimの型

文字型

```
1 const a: char = 'a'
```

- シングルクォーテーションに囲まれたリテラルは char型 に型付けされる
- 実体は8bitのunsignedな型
- Unicode文字を扱うことはできない

Nimの型

文字列型

```
const
  greeting: string = "Hello"
  alphabet: string = """
a
  b
    c
      d
    e
  f
g
"""
```

- ダブルクォーテーションで囲まれた
  リテラルが string型 に型付けされる
- 3つのダブルクォーテーションで囲まれた
  改行を含む文字列として扱われる

- string型はミュータブルなので
  文字を追加したり書き換えられる
- seq[char]と同じように扱える

Nimの型

列挙型

```nim
type Fruits = enum
  orange, apple, grape, muscat, banana, strawberry
```

- 指定した値から構成される新しい型を定義できる
- 内部的にそれぞれの値が整数値として管理される
  - orangeは0、muscatは3、strawberryは5であると解釈される
- デフォルトでは整数型のように順序を持つ
- ユーザーが値に対して独自の値を割り当てることができる

Nimの型

部分範囲型

```nim
type Natural = range[0..high(int)]
```

- 序数を持つ型の範囲を定めて新しく型を定義できる
  - subrange型と呼ばれる
- 範囲外の値にはコンパイル時に検出可能であれば
  コンパイルエラーを、そうでなければ実行時エラーを
  投げる

Nimの型

構造体型

```nim
type User = object
  id: int
  name: string
  password: string

var User1 = User(id: 1, name: "momeemt", password: "password")
```

- 構造化型は同時に複数の値を保持することができ、無制限にネストすることができる
- T(fieldA: valueA, fieldB: valueB, ...) という構文で構築できる
  - 暗黙にプロシージャが定義されるわけではない

構造体の性質

ディープコピー

```
type User = object
  name: string

let momeemt = User(name: "momeemt")
let momeemt_copy = momeemt
```

momeemt_copyにはmomeemtがディープコピーされる

構造体の性質

ディープコピー

```
type User = object
  name: string

let momeemt = User(name: "momeemt")
let momeemt_copy = momeemt
```

momeemt_copyにはmomeemtがディープコピーされる

シャローコピー

```
type User = ref object
  name: string

let momeemt = User(name: "momeemt")
let momeemt_copy = momeemt
```

momeemt_copyとmomeemtの実体は同じ

構造体の性質

ディープコピー

```
type User = object
  name: string

let momeemt = User(name: "momeemt")
let momeemt_copy = momeemt
```

momeemt_copyにはmomeemtがディープコピーされる

シャローコピー

```
type User = ref object
  name: string

let momeemt = User(name: "momeemt")
let momeemt_copy = momeemt
```

momeemt_copyとmomeemtの実体は同じ

生ポインタ

```
type User = ptr object
  name: string

let momeemt = cast[User](User.sizeof.alloc)
momeemt.name = "momeemt"

GCunref momeemt.name
dealloc momeemt
```

ptr[T]型はGCの対象から外れ、自力でメモリ管理する

プラグマ

コンパイラに補足情報や実行命令を与える

プラグマを利用することでメタプログラミングにおける特殊な命令を実行できる
また、出力されるCプログラムに対して干渉できる

```
{.pragmaName.}
{.pragmaName2: "arg".}
```

プラグマの呼び出し

プラグマ

noSideEffectプラグマ

```
proc noSideEffectAdd(a, b:int): int {.noSideEffect.} =
  result = a + b
```

付与したプロシージャやイテレータが副作用を持たないことをコンパイル時に検査する

プラグマ

noSideEffectプラグマ

```
proc noSideEffectAdd(a, b:int): int {.noSideEffect.} =
  result = a + b
```

付与したプロシージャやイテレータが副作用を持たないことをコンパイル時に検査する

```
func noSideEffectAdd(a, b:int): int =
  result = a + b
```

funcはnoSideEffectプラグマが付与されたプロシージャの糖衣構文

FFI

Cのコードを Nim で実行する

```
int shareAdd (int a, b) {
  return a + b
}
```

ソースコードファイル / 共有ヘッダからシンボルを読み込める

```
proc shareAdd(a, b: cint) {.header:"path".}
```

headerプラグマに対象となるファイルの(絶対|相対)パスを渡す

FFI

Cのコードを Nim で実行する

```
{.push header:"<stdio.h>".}

# ...

{.pop.}
```

標準ライブラリから直接読み込むこともできる

FFI

Nimのコードをcで実行する

```
proc shareAdd(a, b: int): int {.exportc.} =
  resule = a + b
```

```
$ nim c --noMain --noLinking --header:main.h main.nim
```

Nimプログラムからmain関数・リンクなしでヘッダーファイルを生成する

FFI

Nimのコードをcで実行する

```
proc shareAdd(a, b: int): int {.exportc.} =
  resule = a + b
```

```
$ nim c --noMain --noLinking --header:main.h main.nim
```

Nimプログラムからmain関数・リンクなしでヘッダーファイルを生成する

```
#include <stdio.h>
#include "main.h"

int main () {
  NimMain();
  int ans = shareAdd(46, 54)
  printf("%d\n", ans);
  return 0;
}
```

Nimコンパイラと関連づけてコンパイルを行う

```
$ gcc -o main.exe -Inimcache -I"path" nimcache/*.c main.c
```

ARC / ORC

ARC: Nimにおける決定論的な参照カウンタ

```
var someNumbers = @[1, 2]
var other = someNumbers
someNumbers.add 3
```

3行目でsomeNumbersがreallocされるので、otherがダングリングポインタにならないようにする

ARC / ORC

ARC: Nimにおける決定論的な参照カウンタ

```
var someNumbers = @[1, 2]
var other = someNumbers
someNumbers.add 3
```

3行目でsomeNumbersがreallocされるので、otherがダングリングポインタにならないようにする

Nimは通常、ディープコピーすることでダングリングポインタを回避
→ 右辺値をmoveすることで someNumbers からの参照を無効にする

ARC / ORC

ARC: Nimにおける決定論的な参照カウンタ

```
var someNumbers = @[1, 2]
var other = someNumbers
someNumbers.add 3
```

3行目でsomeNumbersがreallocされるので、otherがダングリングポインタにならないようにする

Nimは通常、ディープコピーすることでダングリングポインタを回避
→ 右辺値をmoveすることで someNumbers からの参照を無効にする
→ これはC+11における move semantics と同等

ARC / ORC

ARC: Nimにおける決定論的な参照カウンタ

```
var someNumbers = @[1, 2]
var other = someNumbers
someNumbers.add 3
```

3行目でsomeNumbersがreallocされるので、otherがダングリングポインタにならないようにする

Nimは通常、ディープコピーすることでダングリングポインタを回避
→ 右辺値をmoveすることで someNumbers からの参照を無効にする
→ これはC+11における move semantics と同等

Rustは所有権によりsomeNumbersを無効にしてアクセスするとコンパイルエラーを発生させる
→ Nimは参照のみ無効にすることを選んだ。3行目のアクセスは有効で someNumbers == @[3] となる

ARC / ORC

ARC: Nimにおける決定論的な参照カウンタ

Nimは明示的なmoveを嫌ったので、ARCは次の条件下において暗黙にmoveする

ARC / ORC

ARC: Nimにおける決定論的な参照カウンタ

Nimは明示的なmoveを嫌ったので、ARCは次のような状況で暗黙にmoveする

```
var value = fn()
```

- **fn() の結果を参照するのは value のみなので、value に直接 move される**

ARC / ORC

ARC: Nimにおける決定論的な参照カウンタ

Nimは明示的なmoveを嫌ったので、ARCは次のような状況で暗黙にmoveする

```
var value = fn()
```

- **fn() の結果を参照するのは value のみなので、value に直接 move される**

```
var value1 = fn1()
var value2 = fn2(value1)
```

- **value1がfn2プロシージャの呼び出し以外に使われていないとき、**
  fn1() を fn2 に move して fn2 の戻り値を value2 に move することができる

ARC / ORC

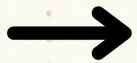
ARC: Nimにおける決定論的な参照カウンタ

ARCはスコープベースのメモリ管理を行う
値が不要になる時をコンパイル時に解析して、デストラクタを自動的に挿入する

```
proc main =
  let mystr = stdin.readLine()

  case mystr
  of "hello":
    echo "Nice to meet you!"
  of "bye":
    echo "Goodbye!"
    quit()
  else:
    discard

main()
```



```
var mystr
try:
  mystr = readLine(stdin)
  case mystr
  of "hello":
    echo ["Nice to meet you too!"]
  of "bye":
    echo ["Goodbye!"]
    quit(0)
  else:
    discard
finally:
  `=destroy`(mystr)
```

ARC / ORC

ORC: ARCベースの循環参照コレクタ（非決定論的）

ARCは参照カウンタなので循環参照を処理できない
ORCはARCベースの循環参照コレクタで将来的にはGCをこちらに切り替えることを目指している

- **ORCはスコープベースのローカルな参照の追跡を行うことができる**

ジェネリクス

型引数を使って抽象的なシンボルを定義する

```
type
  Matrix [T] = seq[seq[T]]

func `+` [T] (mat1, mat2: Matrix[T]): Matrix[T] =
  result = mat1
  for y in 0..mat1.high:
    for x in 0..mat1[0].high:
      result[y][x] = mat1[y][x] + mat2[y][x]

const
  mat1 = @[@[1, 2, 3], @[4, 5, 6]]
  mat2 = @[@[4, 5, 6], @[1, 2, 3]]

echo mat1 + mat2
```

・任意の数値型でインスタンスかできる行列型を定義する

ジェネリクス

型引数を使って抽象的なシンボルを定義する

```
type
  Matrix [T] = seq[seq[T]]

func `+` [T] (mat1, mat2: Matrix[T]): Matrix[T] =
  result = mat1
  for y in 0..mat1.high:
    for x in 0..mat1[0].high:
      result[y][x] = mat1[y][x] + mat2[y][x]

const
  mat1 = @[@[1, 2, 3], @[4, 5, 6]]
  mat2 = @[@[4, 5, 6], @[1, 2, 3]]

echo mat1 + mat2
```

- 任意の数値型でインスタンスかできる行列型を定義する

↓

- Tに任意の型が入るので欠陥がある

ジェネリクス

型クラスの導入

```
1 type
2   Matrix [T: SomeNumber] = seq[seq[T]]
3
4 func `+` [T: SomeNumber] (mat1, mat2: Matrix[T]): Matrix[T] =
5   result = mat1
6   for y in 0..mat1.high:
7     for x in 0..mat1[0].high:
8       result[y][x] = mat1[y][x] + mat2[y][x]
```

- SomeNumberは整数型、浮動小数点数型の型クラス
- TにSomeNumber以外の値が渡ると呼び出し元でエラーを検出する

ジェネリクス

static[T]型によってコンパイル時定数に依存する型を作る

```
type
  Matrix [N, M: static int, T] = array[N, array[M, T]]

func `+` [N, M: static int, T] (mat1, mat2: Matrix[N, M, T]): Matrix[N, M, T] =
  for y in 0 ..< N:
    for x in 0 ..< M:
      result[y][x] = mat1[y][x] + mat2[y][x]

func `*` [L, M, N: static int, T] (mat1: Matrix[L, M, T],
                                   mat2: Matrix[M, N, T]): Matrix[L, N, T] =
  for y in 0 ..< L:
    for x in 0 ..< N:
      for n in 0 ..< M:
        result[y][x] = mat1[y][n] + mat2[n][x]

let
  mat1 = [[1, 2], [3, 4]]
  mat2 = [[5, 6], [7, 8]]
  mat3 = [[1, 2, 3], [4, 5, 6]]

echo mat1 + mat2
echo mat2 * mat3
```

テンプレート

衛生的な仕組みを兼ね備える抽象構文木の置換メカニズム

```
template unless (condition: bool, body: untyped): untyped =
  if not condition:
    body

let number = 100
if number mod 10 == 0:
  echo $number, " is divisible by 10."

unless number mod 12 == 0:
  echo $number, " is not divisible by 12."
```

条件式がfalseを返す場合にブロック内が実行される条件分岐

テンプレート

衛生的な仕組みを兼ね備える抽象構文木の置換メカニズム

```
1 template `!=` (left, right: untyped): untyped =
2   not (left == right)
3
4 echo 12 != 25
```

すべての型は==演算子を実装するだけで良い

テンプレート

インラインプラグマとの違い

```
1 template `!=` (left, right: untyped): untyped =
2   not (left == right)
3
4 echo 12 != 25
```

テンプレート:　　意味解析時に展開される
インラインプラグマ: インライン関数指定（__inline）をして出力するがインライン化するかどうかは
バックエンド言語に委ねられる

テンプレート

インラインプラグマとの違い（テンプレートの場合）

・Nimプログラムのエントリポイントであるの内に展開される

```nim
template `**` (left, right: untyped): untyped =
  var res = 1
  for i in 0 ..< int(right):
    res *= left
  res

echo 5 ** 2
```

```c
N_LIB_PRIVATE N_NIMCALL(void, NimMainModule)(void) {
  // 略
}
```

テンプレート

インラインプラグマとの違い（インラインプラグマの場合）

・N_INLINEが付与されてコンパイルされている

```
proc `**` (left, right: int): int {.inline.} =
  result = 1
  for i in 0 ..< right:
    result *= left

echo 5 ** 2
```

```
static N_INLINE(NI, starstar___book_1)(NI left, NI right) {
  // 略
}
```

テンプレート

untyped型とtyped型

- untyped型は遅延評価が行われる

```
template myVar (name: untyped, value: string): untyped =
  var name = value

myVar(greeting, "Hello")
echo greeting
```

テンプレート

untyped型とtyped型

- untyped型は遅延評価が行われる

```
template myVar (name: untyped, value: string): untyped =
  var name = value

myVar(greeting, "Hello")
echo greeting
```

- ただし、構文上不正なプログラムは抽象構文木に変換できない

```
template myTemplate (statement: untyped): untyped =
  echo "myTemplate!"

myTemplate:
  if name {
    echo "hi!"
  }
```

テンプレート

untyped型とtyped型

・typed型は呼び出し時に意味解析が行われる

```
template cycleIndex (elem: typed, index: int): untyped =
  elem[index mod elem.len]

let
  arr1 = @[1, 2, 3, 4, 5]
  arr2 = @['a', 'b', 'c']
  arr3 = [true]

echo arr1.cycleIndex(10)
echo arr2.cycleIndex(100)
echo arr3.cycleIndex(1000)
```

テンプレート

テンプレートが持つ衛生性

・テンプレートはグローバルに公開されるプログラムとスコープを持つプログラムを分けて生成する

```
template forLearningHygiene: untyped =
  type MyType = object
  var myVar = 100

  proc square (num: int): int =
    result = num * num

forLearningHygiene()

# ok
echo square(10)

# undeclared identifier: ...
var myType = MyType()
echo myVar
```

テンプレート

衛生性を管理する

- injectプラグマによりグローバルに公開でき、gensymプラグマによりスコープに閉じ込められる

```
template forLearningHygiene: untyped =
  let myLet {.inject.} = "Hi!"
  proc square (num: int): int {.gensym.} =
    result = num * num

forLearningHygiene()

# ok
echo myLet

# undeclared identifier: ...
echo square(10)
```

マクロ

抽象構文木を受け取り、抽象構文木を返す言語機能

```
proc debug [T] (name: string, value: T) =
  echo name, ": ", value

var
  greeting = "hello!"
  height = 172

debug("greeting", greeting)
debug("height", height)
```

▲ プロシージャで実装したデバッグユーティリティ

マクロ

抽象構文木を受け取り、抽象構文木を返す言語機能

```
import std/macros

macro debug(arg: untyped): untyped =
  result = newStmtList(
    newCall("write", newIdentNode("stdout"), newLit(arg.repr)),
    newCall("write", newIdentNode("stdout"), newLit(": ")),
    newCall("writeLine", newIdentNode("stdout"), arg)
  )

var
  greeting = "hello!"
  height = 172

debug(greeting)
debug(height)
```

▲ マクロで実装したデバッグユーティリティ

マクロ

マクロを展開するマクロで構造を見破る

```
expandMacros:
  debug(greeting)
```

▲ expandMacrosマクロを使う

マクロ

マクロを展開するマクロで構造を見破る

```
expandMacros:
  debug(greeting)
```

```
write(stdout, "greeting")
write(stdout, ": ")
writeLine(stdout, [greeting])
```

▲ 生成されるプログラム (デバッグ出力)

マクロ

コンパイル時関数実行 (CTFE) をサポート

- 一部制限があるが、プロシージャ、イテレータ、テンプレート、マクロを**コンパイル時に実行できる！**

```
macro debug(arg: untyped): untyped =
  result = newStmtList(
    newCall("write", newIdentNode("stdout"), newLit(arg.repr)),
    newCall("write", newIdentNode("stdout"), newLit(": ")),
    newCall("writeLine", newIdentNode("stdout"), arg)
  )
```

マクロ

受け取る抽象構文木に対して制限をかける

・コンパイル時にユーザーカスタムな意味解析エラー（コンパイルエラー）を出力できる

```
macro debug(args: varargs[untyped]): untyped =
  result = newStmtList()
  for arg in args:
    expectKind(arg, nnkIdent)
    result.add newCall("write", newIdentNode("stdout"), newLit(arg.repr))
    result.add newCall("write", newIdentNode("stdout"), newLit(": "))
    result.add newCall("writeLine", newIdentNode("stdout"), arg)
```

▲ 受け取る引数は全て識別子（nnkIdent）でなければならない

マクロ

受け取る抽象構文木に対して制限をかける

・コンパイル時にユーザーカスタムな意味解析エラー（コンパイルエラー）を出力できる

```
macro debug(args: varargs[untyped]): untyped =
  result = newStmtList()
  expectMinLen(args, 1)
  for arg in args:
    expectKind(arg, nnkIdent)
    result.add newCall("write", newIdentNode("stdout"), newLit(arg.repr))
    result.add newCall("write", newIdentNode("stdout"), newLit(": "))
    result.add newCall("writeLine", newIdentNode("stdout"), arg)
```

▲ 受け取る引数は1つ以上でなければならない

マクロ

quoteプロシージャを使ったより直感的なマクロ構築

```
macro debug2(args: varargs[untyped]): untyped =
  result = newStmtList()
  for arg in args:
    let name = arg.repr
    result.add quote do:
      stdout.writeLine(`name` & ": " & $`arg`)
```

▲生成されるプログラムをプログラムとして記述できる

マクロに関連するプラグマ

compileTime プラグマ

```
proc helper(node: NimNode): NimNode =
  result = node

proc helper(node: NimNode): NimNode {.compileTime.} =
  result = node
```

▲ NimNodeを受け取る/返すプロシージャは暗黙にcompileTimeプラグマを持つ

マクロに関連するプラグマ

error プラグマ

```
proc procWithBug() {.error: "When called, it detects compile-time errors.".} =
  echo "Hello Nim!"

# Error: When called, it detects compile-time errors.
procWithBug()
```

▲ シンボルを呼び出すとコンパイルエラーが発生する

ユーザー定義プラグマ

```
macro counter (name: untyped): untyped =
  result = newIdentNode($name & "CalledCounter")

macro calledCounter (body: untyped): untyped =
  let name = body.name
  var body = body
  body[^1].add nnkInfix.newTree(
    newIdentNode("+="),
    newIdentNode($name & "CalledCounter"),
    newLit(1)
  )
  result = newStmtList(
    nnkVarSection.newTree(
      nnkIdentDefs.newTree(
        newIdentNode($name & "CalledCounter"),
        newEmptyNode(),
        newLit(0)
      )
    ),
    body
  )

proc fib (n: int): int {.calledCounter.} =
  if n == 0: result = 1
  elif n == 1: result = 1
  else: result = fib(n-1) + fib(n-2)
```

・テンプレート/マクロはプラグマとして利用できる

ユーザー定義プラグマ

```
macro counter (name: untyped): untyped =
  result = newIdentNode($name & "CalledCounter")

macro calledCounter (body: untyped): untyped =
  let name = body.name
  var body = body
  body[^1].add nnkInfix.newTree(
    newIdentNode("+="),
    newIdentNode($name & "CalledCounter"),
    newLit(1)
  )
  result = newStmtList(
    nnkVarSection.newTree(
      nnkIdentDefs.newTree(
        newIdentNode($name & "CalledCounter"),
        newEmptyNode(),
        newLit(0)
      )
    ),
    body
  )

proc fib (n: int): int {.calledCounter.} =
  if n == 0: result = 1
  elif n == 1: result = 1
  else: result = fib(n-1) + fib(n-2)
```

・テンプレート/マクロはプラグマとして利用できる

```
calledCounter:
  proc fib (n: int): int =
    if n == 0: result = 1
    elif n == 1: result = 1
    else: result = fib(n-1) + fib(n-2)
```

▲プラグマとして呼び出されたマクロは
上のように展開される

項書き換えマクロ

コンパイル時最適化を目的にパターンマッチに基づいて抽象構文木を置換する機能
最適化を定義してコンパイルパイプラインを拡張するインターフェース

```
template optMulMatrix {
  `*`(a, [[1, 0, 0], [0, 1, 0], [0, 0, 1]])
} [L: static int] (a: array[L, array[3, int]]): array[L, array[3, int]] = a
```

▲ 単位行列との行列積は計算を省略する最適化

引数制約式におけるパターンマッチ演算子

- `|` 演算子 ... 順序付きの選択肢の作成
- `{}` 演算子 ... 式を引数にバインドする
- `~` 演算子 ... パターンの否定
- `*` 演算子 ... 式を平坦化する
- `**` 演算子 ... 式を逆ポーランド記法の順序で収集する

項書き換えマクロ

`|` 演算子　順序付きの選択肢を作成する

```
template optLiteral {-10|0|10} (): untyped = 2

let num = -10
echo num # 2
```

項書き換えマクロ

`{}`演算子　式を引数にバインドする

```
template optLiteral {(-10|0|10){x}} (x: untyped): untyped = x * x

let
  num1 = -10
  num2 = 0
  num3 = 10
  num4 = 5

echo num1 # 100
echo num2 # 0
echo num3 # 10
echo num4 # 5
```

項書き換えマクロ

`~`演算子　パターンの否定

```
template optBool {a = (~a){b} and (~a){c} and (~a){d}} (a, b, c, d: bool) =
  a = b
  if a: a = c
  if a: a = d

var
  a = false
  b = true
  c = true
  d = false

a = b and c and d
echo a
```

項書き換えマクロ

`*`演算子　式をフラットにする

```
var called = 0

proc `~` (nums: varargs[int]): int =
  for num in nums:
  if num mod 2 == 0:
    result += num
  inc called

template optEven { `~` * a } (a: int): untyped =
~a
echo 5 ~ (2 ~ 7) ~ 4
echo "called: ", called
```

項書き換えマクロ

`**`演算子　式を逆ポーランド記法の順序で収集する

```
proc `~<` (left, right: int): int =
  if left < right:
    result = right
  else:
    result = left

macro optMax { `~<` ** a } (a: int): untyped =
  echo a.astGenRepr
  result = newLit(0)

echo 5 ~< (2 ~< 7) ~< 4
```

抽象構文木からユーザー最適化を注入できる

```
nnkArgList.newTree(
  newLit(5),
  newLit(2),
  newLit(7),
  newSymNode("~<"),
  newSymNode("~<"),
  newLit(4),
  newSymNode("~<")
)
```

caseマクロ

```
{.experimental: "caseStmtMacros".}

import std/[macros, options]

macro `case`(n: Option): untyped =
  result = newTree(nnkIfStmt)
  let selector = n[0]
  expectLen(n, 3)
  for i in 1 ..< n.len:
    let it = n[i]
    case it.kind
    of nnkOfBranch:
      expectLen(it, 2)
      expectKind(it[0], {nnkCall, nnkIdent})
      if it[0].kind == nnkCall:
        expectIdent(it[0][0], "Some")
        let cond = nnkDotExpr.newTree(selector, newIdentNode("isSome"))
        result.add newTree(nnkElifBranch, cond, nnkStmtList.newTree(
          nnkVarSection.newTree(
            nnkIdentDefs.newTree(
              it[0][1],
              newEmptyNode(),
              nnkDotExpr.newTree(
                selector,
                newIdentNode("get")
              )
            )
          ), it[1])
        )
      elif it[0].kind == nnkIdent:
        expectIdent(it[0], "None")
        let cond = nnkDotExpr.newTree(selector, newIdentNode("isNone"))
        result.add newTree(nnkElifBranch, cond, it[1])
    else:
      error "custom 'case' for Option[T] cannot handle this node", it
```

- case文を任意の具体型に対して拡張できる実験的機能
  - {.experimental: "caseStmtMacros".}

caseマクロ

```
{.experimental: "caseStmtMacros".}

import std/[macros, options]

macro `case`(n: Option): untyped =
  result = newTree(nnkIfStmt)
  let selector = n[0]
  expectLen(n, 3)
  for i in 1 ..< n.len:
    let it = n[i]
    case it.kind
    of nnkOfBranch:
      expectLen(it, 2)
      expectKind(it[0], {nnkCall, nnkIdent})
      if it[0].kind == nnkCall:
        expectIdent(it[0][0], "Some")
        let cond = nnkDotExpr.newTree(selector, newIdentNode("isSome"))
        result.add newTree(nnkElifBranch, cond, nnkStmtList.newTree(
          nnkVarSection.newTree(
            nnkIdentDefs.newTree(
              it[0][1],
              newEmptyNode(),
              nnkDotExpr.newTree(
                selector,
                newIdentNode("get")
              )
            )
          ), it[1])
        )
      elif it[0].kind == nnkIdent:
        expectIdent(it[0], "None")
        let cond = nnkDotExpr.newTree(selector, newIdentNode("isNone"))
        result.add newTree(nnkElifBranch, cond, it[1])
    else:
      error "custom 'case' for Option[T] cannot handle this node", it
```

- case文を任意の具体型に対して拡張できる実験的機能
  - {.experimental: "caseStmtMacros".}
- ユーザー定義パターンマッチを実装できる
  - 左は Option[T] 型に対して機能する

```
case opt1
of Some(a):
  echo a
of None:
  echo "none"

case opt2
of Some(a):
  echo a
of None:
  echo "none"
```

▲ Rustライクなパターンマッチ

forループマクロ

```
import std/macros

proc multipleAST (loopast: NimNode, nest: int): NimNode =
  if nest == 0:
    result = loopast[^1]
  else:
    result = nnkStmtList.newTree(
      nnkForStmt.newTree(
        loopast[^(nest+2)],
        loopast[^2][1],
        multipleAST(loopast, nest-1)
      )
    )

macro multiple (loop: ForLoopStmt): untyped =
  result = nnkForStmt.newTree(
    loop[0],
    loop[^2][1],
    multipleAST(loop, loop.len-3)
  )
```

・イテレータにForLoopStmt型を受け取るマクロを置くことでfor文全体を書き換える機能

forループマクロ

```
import std/macros

proc multipleAST (loopast: NimNode, nest: int): NimNode =
  if nest == 0:
    result = loopast[^1]
  else:
    result = nnkStmtList.newTree(
      nnkForStmt.newTree(
        loopast[^(nest+2)],
        loopast[^2][1],
        multipleAST(loopast, nest-1)
      )
    )

macro multiple (loop: ForLoopStmt): untyped =
  result = nnkForStmt.newTree(
    loop[0],
    loop[^2][1],
    multipleAST(loop, loop.len-3)
  )
```

・イテレータにForLoopStmt型を受け取るマクロを置くことでfor文全体を書き換える機能

・左は多重ループに展開するマクロを実装している

```
for x, y, z in multiple(1..2):
  echo x, " ", y, " ", z
```

▲3重ループに展開される

まとめ

- NimはC/C++/Objective-C/JavaScriptをバックエンド言語に持つコンパイル言語です
- 静的型付け言語であり、ガベージコレクタ（GC）を持ちます
- Web開発・CLI開発・デスクトップアプリ開発・OS開発・組込み開発など幅広く適用できます
- Nimはビルドツール兼パッケージマネージャーであるNimbleを使って開発を行います
- 数値型や文字列型はもちろん、列挙型、構造体型、メタ型、コンパイル時定数型、ジェネリクスなど十分な型の表現力を持ちます
- 洗練されたFFIにより、非常に簡単にCやC++、JavaScript、Objective-Cと相互運用できます
- テンプレートは衛生性を持った抽象構文木の置換機能です
- マクロは抽象構文木を受け取り、新たな抽象構文木を返却する言語機能です
- 現在Nimには新しいGCの導入が進んでおり、決定論的な参照カウンタであるARCとそれをベースにした循環参照コレクタであるORCが検討されています。
- 項書き換えマクロにより、ユーザー定義最適化を記述してコンパイルパイプラインを拡張できます
- caseマクロにより、型に対するユーザー定義パターンマッチを記述できます

参考文献

- Nim Manual
  - https://nim-lang.org/docs/manual.html
- Nim Experimental Features
  - https://nim-lang.org/docs/manual_experimental.html
- Nim Compiler User Guide
  - https://nim-lang.org/docs/nimc.html
- std/macros
  - https://nim-lang.org/docs/macros.html
- Introduction to ARC/ORC in Nim
  - https://nim-lang.org/blog/2020/10/15/introduction-to-arc-orc-in-nim.html
- Move semantics for Nim
  - https://youtu.be/yA32Wxl59wo