# 簡易インタプリタの実装

## スクリプト言語演習

佐藤 正平、草野 翼、伊澤 侑祐、大久保 寛

Scripting Languages / EECS / 07

# 電卓から小さな言語まで、4 段階で組み立てる

| 段階 | できるようになること | 中心となる道具 | 課題 |
|---|---|---|---|
| 1. 計算する | 数式を計算する電卓 | スタック、関数を入れた辞書 | 1 |
| 2. 増やす | 使えるワードを増やす | `WORDS` に関数を足す | 2 |
| 3. 定義する | 新しいワードを自分で作る | `: 名前 ... ;` | 3, 4 |
| 4. 対話する | 入力と実行を繰り返す | REPL | 5 |

- 使う道具は、第 6 回のリスト・辞書・`for`・`if` だけ
- 電卓に「新しい命令を定義する仕組み」を加えると、言語になる
- 各スライドの右側に、コードの読み方を書いてある

# 1970 年代から実用されてきた小さな言語 Forth

## 全体像

```
3 4 + 2 * .
```

- Charles H. Moore が開発した、スタックを使って計算する言語
- 宇宙機・天文台・組み込み機器の制御に使われてきた
- 命令 (**ワード**) を空白で区切って並べるだけ。文法がとても小さい
- 実行中に、新しいワードを定義できる

読み方: 3 と 4 を積み、+ で足して 7、2 を積み、* で掛けて 14、. で表示する

# 演習の地図: 骨組みは用意済み、★だけ自分で書く

## 全体像

```
forth_eval(stack, text)          用意済み: ワードを左から 1 つずつ取り出す
  ├── read_definition(tokens, i)   ★課題 3  「: 名前 本体 ;」を読む
  ├── eval_user(stack, word)       ★課題 4  自作ワードの本体を実行する
  └── evaluate(stack, text)        ★課題 1  電卓: WORDS を引いて実行する
                                 ★課題 2  WORDS のワードを増やす
repl(lines)                      ★課題 5  1 行ずつ評価する対話ループ
```

- ループの外枠は用意してある。**空いている所を 1 行ずつ**埋めていく
- うまく動かないときは `trace("3 4 +")` で、1 語ごとのスタックを見る

# スタック: 最後に積んだものから取り出すリスト

## 1. 計算する

```
stack = []
stack.append(3)      # [3]
stack.append(4)      # [3, 4]
top = stack.pop()    # top は 4
print(stack)         # [3]
```

**コードの読み方**

- スタックは、ただのリスト。皿を積み重ねるイメージ
- `append`: 一番上に積む (push)
- `pop`: 一番上を取り出す (pop)。取り出した値は、リストから消える
- `stack[-1]`: 一番上を、取り出さずに見る

# 逆ポーランド記法: 演算子を数の後ろに書く

## 1. 計算する

| 読んだワード | 行うこと | スタック |
|---|---|---|
| 3 | 3 を積む | [3] |
| 4 | 4 を積む | [3, 4] |
| + | 2 つ取り出し、和を積む | [7] |
| 2 | 2 を積む | [7, 2] |
| * | 2 つ取り出し、積を積む | [14] |

- `3 4 + 2 *` は、普通の書き方の `(3 + 4) * 2`
- かっこも、計算の順番の決まりも要らない。左から読むだけ
- スタック全体が、計算途中の状態を表す

# 引き算と割り算は、取り出す順序に注意

1. 計算する

```
stack = [10, 3]
b = stack.pop()
a = stack.pop()
stack.append(a - b)
```

**コードの読み方**

1. 10 3 - の時点で、スタックは [10, 3]
2. 最初の pop で 3 が出る。これが右側の値 b
3. 次の pop で 10 が出る。これが左側の値 a
4. a - b は 10 - 3 で 7

- 後から積んだ値が先に出る。b と a を逆にすると、3 - 10 になる

# ワードの表: 名前から関数を引く辞書

## 1. 計算する — 課題 1

```
WORDS = {}

def add(stack):
    b, a = stack.pop(), stack.pop()
    stack.append(a + b)

WORDS["+"] = add
```

**コードの読み方**

- `WORDS`:「ワードの名前 → 関数」の表 (普通の辞書)
- `add` は、スタックを受け取り、上から 2 つ取り出して、足して積む
- `WORDS["+"] = add`: 関数を `"+"` という名前で表に入れる。`()` は付けない
- 第 6 回の `OPS` と同じ形。課題 1 では `- * /` を同じ形で足す

# evaluate: 表にあれば呼び、数なら積む I



```
def evaluate(stack, text):
    for token in text.split():
        if token in WORDS:
            WORDS[token](stack)
        elif token.lstrip("-").isdigit():
            stack.append(int(token))
        else:
            raise ValueError(token)
```

**コードの読み方**

1. `split()` で、入力を空白で区切り、1 つずつ見る
2. 表 `WORDS` にあれば、その関数を取り出して、スタックを渡して呼ぶ
3. 数なら、`int` で数にして積む (`lstrip("-")` は `-5` のような負の数のため)
4. どちらでもなければ、エラーにする

# evaluate: 表にあれば呼び、数なら積む II

## 1. 計算する — 課題 1

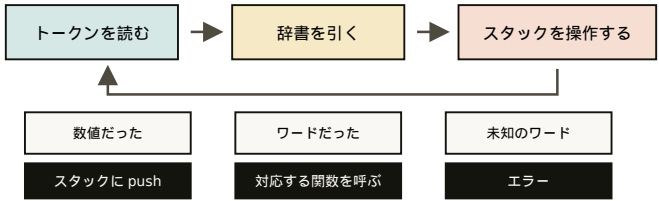


Figure: evaluate の流れ

# ワードを増やしても、evaluate は変えない

## 2. 増やす — 課題 2

| ワード | 意味 | スタックの変化 |
|---|---|---|
| . | 一番上を表示して取り除く | (a → ) |
| DUP | 一番上を複製 | (a → a a) |
| DROP | 一番上を捨てる | (a → ) |
| SWAP | 上 2 つを交換 | (a b → b a) |

```
def dup(stack):
    stack.append(stack[-1])

WORDS["DUP"] = dup
```

- (a b → b a) は、左が実行前、右が実行後のスタック
- stack[-1] を積み足すと、一番上の複製になる
- 表に入れるだけで、5 DUP * が 25 になる

# : から ; までを、新しいワードとして保存する

## 3. 定義する

```
: SQUARE DUP * ;
5 SQUARE .
```

| 部分 | 役割 |
|---|---|
| `:` | 定義の始まり |
| `SQUARE` | 新しいワードの名前 |
| `DUP *` | 本体。呼ばれたときに実行する |
| `;` | 定義の終わり |

- 読み方:「SQUARE は、DUP して * することだ」と覚えておく
- 定義を読んでいる間は、本体を実行しない
- 本体はワードのリストのまま、USER_WORDS に保存する

# read_definition: ; の位置を探して切り出す

## 3. 定義する — 課題 3

```
tokens = [':', 'SQUARE', 'DUP', '*', ';', '5', 'SQUARE']
位置       0       1       2     3    4    5     6
```

```
j = tokens.index(";", i)
name = tokens[i + 1].upper()
body = tokens[i + 2 : j]
return name, body, j + 1
```

**コードの読み方** (i は : の位置、ここでは 0)

1. tokens.index(";", i): ; の位置を探す。ここでは 4
2. name: : の次。大文字にして'SQUARE'
3. body: 名前と ; の間。['DUP', '*']
4. j + 1: 定義の次の位置 5。そこから続きを読む

# 保存した本体を実行すれば、自作ワードが動く

## 3. 定義する — 課題 4

```
def eval_user(stack, word):
    forth_eval(stack, USER_WORDS[word])
```

**コードの読み方**

1. `USER_WORDS["SQUARE"]` は、保存した本体 `['DUP', '*']`
2. その本体を、そのまま `forth_eval` に渡して実行する
3. `5 SQUARE` では、5 を積み、`DUP` で複製、`*` で掛けて 25

```
: DOUBLE DUP + ;
: QUADRUPLE DOUBLE DOUBLE ;
5 QUADRUPLE .
```

- `5 QUADRUPLE .` は 20 を表示する
- 自作ワードの中で、別の自作ワードも使える。`forth_eval` がまた `eval_user` を呼ぶ

# REPL:「読む・実行する・表示する」を繰り返す

## 4. 対話する — 課題 5

```
stack = []
while True:
    line = input(">> ")
    if line.strip().lower() == "bye":
        break
    try:
        forth_eval(stack, line)
        print(stack)
    except Exception as e:
        print("error:", e)
```

**コードの読み方**

1. `stack = []` は、ループの外で作る。だから、前の行の結果が残る
2. `input` で 1 行読む。`bye` ならやめる
3. `forth_eval` で実行して、スタックを表示する
4. エラーが出たら `error:` と表示して、ループは続ける

- 課題 5 では、`input()` の代わりに行のリストを受け取る `repl(lines)` を書く (採点できるようにするため)

# 状態をループの外に置くと、対話が次の行へ続く

## 4. 対話する

```
>> 3 4 +
[7]
>> 2 *
[14]
>> : SQUARE DUP * ;
[14]
>> 5 SQUARE
[14, 25]
>> FOO
error: 知らないワード: FOO
```

- `3 4 +` で `[7]`、続けて `2 *` で `[14]`。前の結果にそのまま続く
- `: SQUARE DUP * ;` は何も積まず、`[14]` のまま。定義だけが残る
- エラーが起きた行は `error: ...` と表示して、ループは続ける

# まとめ: 4 段階をつなぐと、小さな言語処理系になる

| 段階 | 入力から起きること | 書いたもの |
|---|---|---|
| 1. 計算する | 数を積むか、表の関数を呼ぶ | `evaluate`、`- * /` |
| 2. 増やす | 表に関数を足すと、使えるワードが増える | `. DUP DROP SWAP` |
| 3. 定義する | ワードの列に名前を付けて保存し、実行する | `read_definition`、`eval_user` |
| 4. 対話する | 状態を保ったまま入力を繰り返す | `repl` |

- 課題は「例題セルを実行 → `TODO` の 1 行を埋める → 確認セルが `OK`」
- 基本課題 (課題 1〜5) で 80 点。発展は、比較のワードと `IF ... THEN` (各 10 点)

# 付録: 講義では扱わない補足

発展的な内容や復習。時間があるときや、質問が出たときに使う。

- 発展: 比較と条件分岐

# 発展: 比較と条件分岐

付録

```
: ABS DUP 0 < IF 0 SWAP - THEN ;
-5 ABS .
```

- `-5 ABS .` は 5 を表示する
- 比較のワード `< > =`: 真は `-1`、偽は `0` (`WORDS` に入れる)
- `IF ... ELSE ... THEN`: 用意された `split_if` で、真のときに実行する部分と、偽のときに実行する部分に分ける
- `forth_eval` の `TODO` の所に、`:` の場合と同じ形で `IF` の場合を足す

ワードと条件分岐がそろうと、小さな言語でも役に立つプログラムが書ける。