Forth インタプリタの仕組み

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 / Dynamic Languages / 08

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. 読む	入力をワードへ分ける	`split`	用意済み
2. 実行する	ワードに対応する処理を呼ぶ	スタック、辞書 (第 7 回の電卓)	1, 2
3. 定義する	新しいワードを追加する	`: 名前 ... ;`	3, 3-b
4. 対話する	入力と実行を繰り返す	REPL	4

- 第 7 回の電卓に「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

- Charles H. Moore が 1960 年代後半に開発した_スタック指向言語_
  - スタックを介して計算を行う
- 名前は「第 4 世代言語 (fourth)」に由来する
- 宇宙機・天文台・組み込み機器の制御に実績がある
- 小さく・速く・対話的に動かすことを設計方針とする

実績	内容
宇宙機	NASA の探査機（Cassini など）の制御に利用
天文台	望遠鏡の制御・データ処理に利用
組み込み	数 KB の処理系で動く軽さから幅広く使われる

この言語の「核」を最小構成で実装する

Forth は、空白で区切ったワードを順に実行する

全体像

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

特徴	意味
逆ポーランド記法	演算子を値の後ろに書く
スタック指向	引数と結果をスタックで受け渡す
小さな構文	基本の入力はワードの列
拡張可能	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
- 字句解析器・状態・ディスパッチがあれば、最小の処理系を作れる

これまで使った道具が、そのまま言語処理系になる

全体像

これまでの道具	Forth での役割
文字列の分割	入力をワードへ分ける
Python のリスト	計算途中の値を保持する
関数を値に持つ辞書	ワードから処理を選ぶ
状態更新のループ	ワードを順番に実行する

新しい道具を大量に覚えるのではなく、既知の道具の組み方を変える。

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 3-b   自作ワードの本体を実行する
  └── evaluate(stack, text)        ★課題 1     第 7 回の電卓: 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 4     1 行ずつ評価する対話ループ
```

- 課題 1・2 は**第 7 回の課題 4・4-b と同じ**。第 7 回のコードをそのまま貼ってよい
- ループ本体を一から書くのではなく、**小さな関数を 1 つずつ**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

最初の字句解析は、空白で分けるだけ

1. 読む

```
code = "3 4 + 2 *"
tokens = code.split()

print(tokens)
# ['3', '4', '+', '2', '*']
```

コードの読み方

- `split()` は、空白で区切って、リストにする
- 括弧や、演算子の優先順位は考えなくてよい
- 左から右へ読めば、そのまま実行する順番になる
- 大文字小文字は区別しない (`forth_eval` が大文字にそろえる)

数は積み、演算子は上の値を使う

2. 実行する

読んだワード	行うこと	スタック
`3`	3 を push	`[3]`
`4`	4 を push	`[3, 4]`
`+`	2 つ pop し、和を push	`[7]`
`2`	2 を push	`[7, 2]`
`*`	2 つ pop し、積を push	`[14]`

- スタックはただのリスト。`append` が push、`pop` が pop
- スタック全体が、実行途中の状態を表す

pop の順序は、引き算と割り算で結果に現れる

2. 実行する

```
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取り出される。b と a を逆にすると、結果が変わる
- スタック図を書いてからコードにすると、順序を間違えにくい

ワード辞書は、名前から操作関数を引く表

2. 実行する

```
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
- 第 7 回と同じ書き方。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

2. 実行する — 課題 1

```
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. 入力を空白で区切り、1 つずつ見る
2. 表 WORDS にあれば、その関数を呼ぶ
3. 数なら、スタックに積む (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

2. 実行する — 課題 1

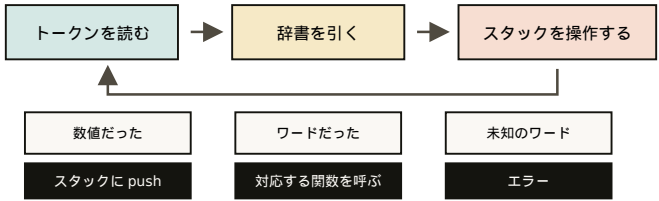

Figure: Forth のディスパッチループ

スタック操作は、入る前と出た後で考える I

2. 実行する — 課題 2

ワード	意味	スタックの変化
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)
OVER	2 番目を上へコピー	(a b → a b a)
ROT	3 番目を一番上へ移動	(a b c → b c a)

スタック操作は、入る前と出た後で考える II

2. 実行する — 課題 2

```
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] は、一番上の値。それを積み足すと複製になる
- 書いた関数は、WORDS["DUP"] = dup で表に入れる
- . (一番上を出力して取り除く) は、この回で新しく足すワード

ここまでは、固定された語彙を実行する電卓

2 段階目までを確認する

```
入力文字列
  ↓ split
ワードの列
  ↓ 左から 1 語ずつ読む
数なら push ──┐
              ├→ スタックを更新
ワードなら実行┘
```

この時点でも計算はできるが、使えるワードは Python 側で決めたものだけである。次は、Forth のプログラム自身からワードを増やす。

: 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 * ;
```

部分	役割
`:`	定義の開始
`SQUARE`	新しいワードの名前
`DUP *`	実行時に評価する本体
`;`	定義の終了

- 読み方:「SQUARE は、DUP して * することだ」と登録する
- 定義中は本体をまだ実行せず、ワードの列として保存する

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']
添字        0       1        2     3    4    5     6
```

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

コードの読み方 (i は : の位置、ここでは 0)

- tokens.index(";", i): ; の位置を探す。ここでは 4。無ければエラー
- tokens[i + 1]: 名前。'SQUARE'
- tokens[i + 2 : j]: 名前と ; の間が本体。['DUP', '*']
- j + 1: 定義の次の位置。ここでは 5

- 返すのは (名前を大文字にしたもの，本体，次の位置) の 3 つ組
- 読み取るだけで、定義は**実行しない**。登録は用意済みの forth_eval がする

保存した本体を評価すれば、自作ワードを実行できる

3. 定義する — 課題 3-b

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

```python
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

組み込みワードは Python 関数、自作ワードはワード列だが、どちらも「名前を辞書で引いて実行する」という入口は同じである。

自作ワードから別の自作ワードも呼べる

3. 定義する

```
: DOUBLE DUP + ;
: QUADRUPLE DOUBLE DOUBLE ;

5 QUADRUPLE .
```

- `DOUBLE` は、2 倍にするワード。`QUADRUPLE` は、それを 2 回使うので 4 倍
- `5 QUADRUPLE` は、5 → 10 → 20 と進み、`20` を表示する
- 実行時には、通常のやり方で `DOUBLE` を引く
- `forth_eval` が自分自身を呼ぶので、入れ子にもそのまま対応できる

語彙が増えるにつれて、電卓がプログラミング言語へ近づく。

REPL は「読む・実行する・表示する」を繰り返す

4. 対話する

REPL は Read – Eval – Print Loop の略である。

```
def repl_interactive():
    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 4 は、採点のため `input()` ではなく行のリストを受け取る `repl(lines)` を書く

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]
```

- 読み方: `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
- `5 SQUARE` で、5 を積んで 2 乗し、`[14, 25]`
- エラーが起きた行は `error: ...` と表示して、ループは続ける

4 段階をつなぐと、小さな言語処理系になる

まとめ

段階	入力から起きること	書いたもの
1. 読む	文字列をワードの列へ分ける	(用意済み)
2. 実行する	数を積むか、辞書の操作を呼ぶ	`evaluate`、`WORDS` (第 7 回の電卓)
3. 定義する	ワード列に名前を付け、辞書へ加える	`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`

Forth の実装は小さいが、字句解析・評価・定義・対話というインタプリタの主要部分をすべて含んでいる。

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
```

- 比較演算子 < > =: 真は -1、偽は 0 (WORDS に入れる)
- IF ... ELSE ... THEN: 対応する THEN を、入れ子の深さを数えて探す
- forth_eval の # 発展 2 の目印に、: の枝と同じ形で足す

語彙と制御構造が揃うと、小さな言語として実用的なプログラムが書ける。