

Forth インタプリタの仕組み

スクリプト言語演習

伊澤侑祐

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
1. 読む	入力をワードへ分ける	<code>split</code>
2. 実行する	ワードに対応する処理を呼ぶ	スタック、辞書
3. 定義する	新しいワードを追加する	: 名前 ... ;
4. 対話する	入力と実行を繰り返す	REPL

第 2 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。

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

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。

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', '*']
```

- Python の字句解析より単純で、括弧や演算子の優先順位を扱わない
- 左から右へ読めば、実行順序もそのまま決まる
- ワード名は大文字にそろえると、大文字小文字を区別せずに済む

小さな構文なので、実行の仕組みに集中できる

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() # 右側の値 3
a = stack.pop() # 左側の値 10
stack.append(a - b)
```

10 3 - → 10 - 3 → 7

- 後から積んだ値が先に取り出される
 - $a - b$ と $b - a$ は結果が異なる
 - 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

```
import operator

def binop(fn):
    def run(stack):
        b, a = stack.pop(), stack.pop()
        stack.append(fn(a, b))
    return run
```

```
WORDS = {
```

```

    "+": binop(operator.add),
    "-": binop(operator.sub),
}

```

関数を辞書の値にすると、長い条件分岐を表の参照へ置き換えられる。

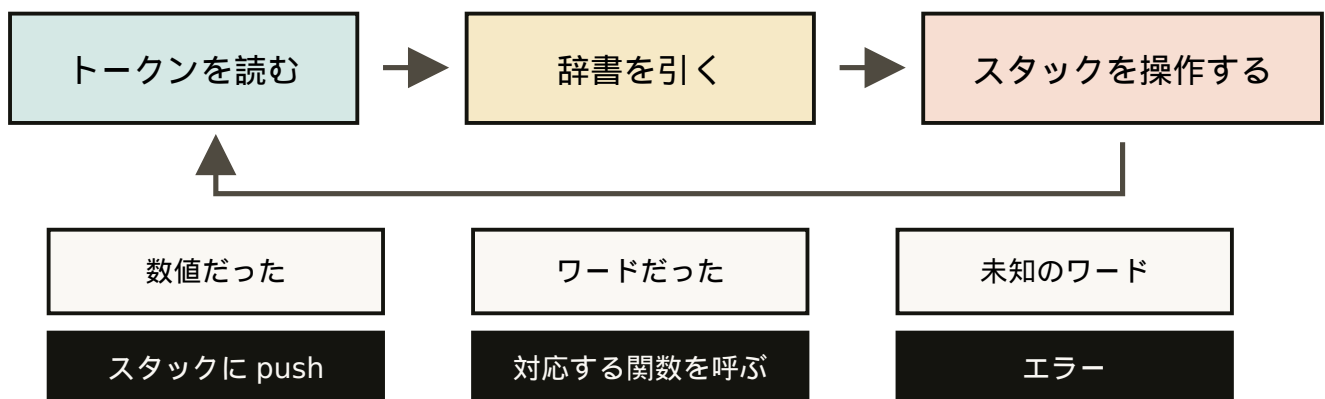
ディスパッチは、ワードの種類に応じて処理を選ぶ

2. 実行する

```

for token in code.split():
    word = token.upper()
    if word in WORDS:
        operation = WORDS[word]
        operation(stack)
    elif token.lstrip("-").isdigit():
        stack.append(int(token))
    else:
        raise ValueError(f"知らないワード: {token}")

```



ワードが増えても、辞書へ登録すればループ本体は変わらない。

スタック操作は、入る前と出た後で考える

2. 実行する

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)

(a b → b a) の表記は、左が実行前、右が実行後を表す。

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

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。

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

: SQUARE DUP * ;

5 SQUARE .

SQUARE を読む

↓ 辞書から本体 [DUP, *] を取得

DUP を実行

↓

* を実行

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 .

- QUADRUPLE の本体に DOUBLE を 2 回保存する
- 実行時には、通常のディスパッチで DOUBLE を引く
- 小さなワードを組み合わせて、大きな処理を作れる

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。

```
while True:  
    line = input(">> ")  
    if line.strip().upper() == "BYE":  
        break  
    # line を評価する  
    print(stack)
```

- Read: 1 行の入力を読む
 - Eval: 現在の辞書とスタックを使って評価する
 - Print: 結果またはスタックを表示する
 - Loop: 次の入力へ戻る
-

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]

スタックとワード辞書を毎回作り直さないため、前の入力結果と定義が残る。この「実行中に状態と語彙が変わる」性質が、動的な処理系の核になる。

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
1. 読む	文字列をワードの列へ分ける
2. 実行する	数を積むか、辞書の操作を呼ぶ
3. 定義する	ワード列に名前を付け、辞書へ加える
4. 対話する	状態を保ったまま入力を繰り返す

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。