□□□□□□□□□□

□3□ (B)

□□□□□□

ポーランド電卓が土台

`;` で、実行中に語彙を追加

れば、電卓は言語に

自分で定義したワードを、同じループで実行

☒☒☒

ろに書き、値はすべてスタック経由

は、空白で区切ったワードの列

ユーザ定義に、扱いの差は無し

	内容
	`3 4 +` のように演算子を後ろに書く
	引数も返り値もすべてスタック経由
	空白区切りの「ワード」が並んでいるだけ
	組み込みとユーザ定義を区別しない

解析とディスパッチだけ。処理系の題材として小さい

□□□□□□□□□

とスタックは第2回のまま利用

、第1回のディスパッチと同じ形

卓を、スタックと辞書で再構成

作ったもの	今日の役割
`@command` と `run()`	ワード辞書とディスパッチ
ディスパッチテーブル	組み込みワードの実装
`split` によるトークン化	字句解析
逆ポーランド電卓	今日の出発点

図

読む

、演算子なら上から2つ取って計算

: `7` 。先に pop されるのは `3`

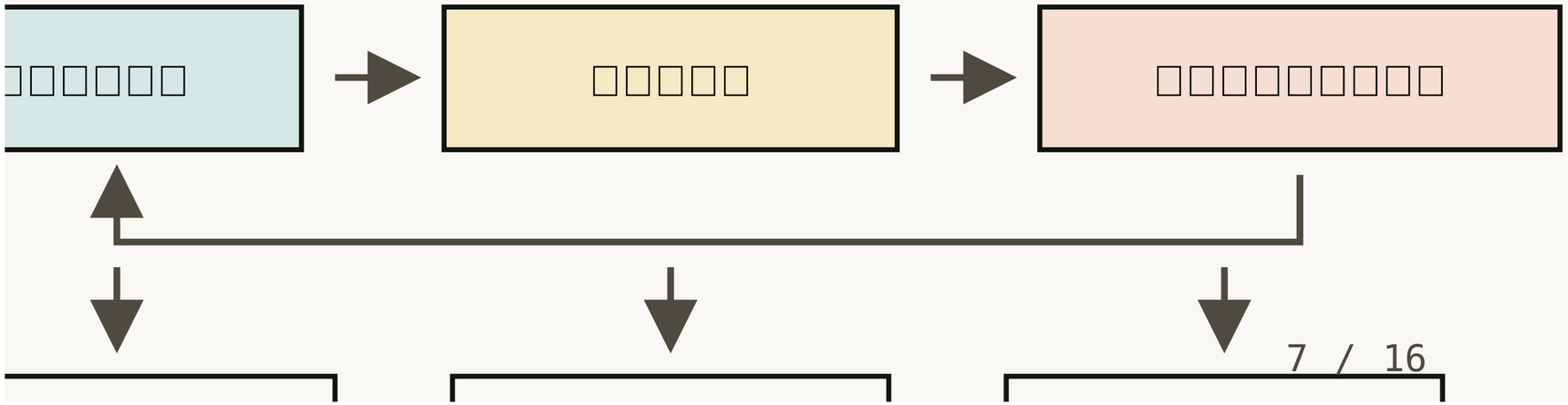
	スタック
	`[3]`
	`[3, 4]`
	`[7]`
	`[7, 2]`
	`[14]`

□□□□□□□□

あれば、その関数を呼ぶ

数として積む

なら、そこで停止



□□□□□□□□

普通の関数をワードに変換

れるのが右辺

は、スタックを直接操作

```
ack):
 stack.pop(), stack.pop()
ppend(fn(a, b))



(operator.add), "-": binop(operator.sub),
```

□□□□□

、大文字にそろえて表を参照

整数も、数として積む

さは、次の `: ... ;` から

```
(stack, code):
in code.split():
oken.upper()
in WORDS:
RDS[up](stack)
oken.lstrip("-").isdigit():
ack.append(int(token))

ise ValueError(f"□らないワード: {token}")
```

□□□□□□

□□□

までが、新しいワードの本体

きたら、保存した列を評価

ードは、別のワードからも呼び出し可能

```
;
      ( 25 )

;
UBLE DOUBLE ;
      ( 20 )
```

定義の扱いに差は無し。ここが Forth の面白さ

□□□□□□□□□

辞書は、動いている最中に書き換え可能

`SQUARE DUP * ;` も、同じことを実行

ら、新しい名前が言語の一部に

```
 = lambda s: s.append(s.pop() ** 2)
SQUARE が使える
```

ではスタックマシン。`dis` でバイトコードを見ると、今日の処理に似た命令列が出現

□□□□□□

タは、字句解析とディスパッチで動作

理系は、スタックと辞書で構成

書を書き換えれば、言語自体を延長可能

`EN` まで実装できれば、小さいながらも言語処理系

□□□□□□□□□

スタック操作を動作

`□□ ... ;` の登録

L を付けて対話可能に

```
```

けのインタプリタ (`+ - * /`)

ク操作ワード (`. DUP DROP SWAP OVER ROT`)

定義ワード `: □□ ... ;`

ープ (REPL)

`> =` / `IF ... ELSE ... THEN`

□□□□□□□

開く」から課題を開き、「ドライブにコピーを保存」

回のスタック操作を移し、1から `TODO` を埋める

消さない

のテストケースで採点するので消しても害はない