Chapter 2

Esoteric Languageを作る

2-1 きらめく星空のような言語、Starry

ここからは本書オリジナルの（esotericな）プログラミング言語を作成する。まずは星空のような見た目をしたスタック型言語、Starryだ。

Esoteric Languageの設計と実装

Starryの設計に移る前に、「（esotericな）プログラミング言語を設計するにはどうすればいいか」という話をしておこう。

まず、プログラミング言語というものは構文（syntax、シンタックス）と意味論（semantics、セマンティクス）からできている。

構文は、そのプログラミング言語がどのような構造をしているかということだ。例えばRubyのソースコードは「1」「"hello"」のようなリテラルや、「if」「while」のような予約語、「(」「)」「=」「<<」のような記号といったさまざまな要素から構成される。もっとも、これらの要素は自由に並べていいわけではなく、例えば「a + << 1」というソースコードはRubyの正しいプログラムとは見なされないわけだ。このようなルールのことを構文と言う。

一方、それらの規則に従って並べた要素がどういう意味として解釈されるのか、それが意味論だ。例えばC言語では「1+2」というプログラムは「整数1と整数2の足し算を行った結果」という意味になるが、Rubyでは「1+2」は（すでに説明したように）「2という整数オブジェクトを引数に、1という整数オブジェクトの+メソッドを呼び出した結果」という意味になる。

そういうわけで、オリジナルなプログラミング言語を設計したいと思えば、構文と意味論を決めることが必要だ。まあ、仕様が比較的単純なEsoteric Languageに限ってしまえば、「『見た目』と『中身』を決めることが必要だ」と言い換えてしまってもいい。

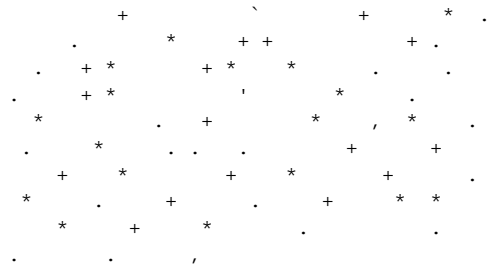
「見た目」から考えるか、「中身」から考えるかはあなたの自由だ。どちらでも好きなほうを先に決めればいい。これまでに取り上げたEsoteric Languageの中では、Whitespace

はきっと見た目から先に決まった言語だろう。一方Brainf*ckはたぶん「チューリング・マシンをモデルにした言語を設計する」という「中身」の側が先に決まった言語なのではないかと筆者は想像している。

Starry言語の設計

本節で紹介するStarryは、「星空みたいに見える言語はどうだろう」という、見た目のほうのアイデアから生まれた。アスキーアートで、「+」や「*」で星空を表現したものがあるが、あれがもしも言語だったら？　と考えたわけだ。

手元のメモには以下のような星空が描かれている（これは単なるコンセプトであって、Starryとして意味のあるプログラムではない）。



見た目が決まれば、次に考えるのは中身だ。付録の「Esoteric Language傑作選」を読んでもらえばわかるが、Esoteric Languageの「見た目」が実にバラエティに富んでいるのに対し、「中身」のバリエーションはそれほど多くはない。中でももっとも多いのがWhitespaceのように、スタックを計算領域として用いる言語だ。ある意味、伝統的といってもいいかもしれない。

多くの言語が採用しているということは、それだけスタック型言語が作りやすいということでもあるのだろう。Starryは最初に作る言語だから、とりあえずスタック型にしてみることにしよう。

▌命令を決めよう

見た目と中身が決まったところで、どのような命令を用意するか考えてみよう。スタック型の言語ということだから、用意する命令は

- スタックを扱う命令 : push, dup, swap, rotate, pop
- 四則演算 : + - * / %
- 文字・数値の入出力命令
- 分岐命令

くらいあればいいだろうか。

それぞれの命令についてもう少し詳しく説明しよう。

push命令は数値を１つスタックにプッシュする（数値以外は考えないことにしよう）。dup命令はスタックの一番上の値を複製する。swap命令はスタックの上から２つの値を入れ替える。rotate命令はスタックの上から３つの値を回転する（一番上の値を３番目に押し込む、と言ってもいい。スタックが「| x y z」となっていたなら、rotate後は「| z x y」のようになる）。pop命令はスタックの一番上の値を取り除く。

四則演算と、入出力命令はWhitespaceのそれと同じだ（ただしWhitespaceのようなヒープがないので、入力命令は読み込んだ値をスタックに積むことにする）。

分岐命令はWhitespaceのようにラベルを定義するとか、Brainf*ckのように２種類の対になる記号で表現するなどいろいろと考えられるが、これは構文をどれくらい複雑にできるかによって選択肢が変わってくるので、後で決めることにしよう。

構文を決めよう

次は、上のような命令をStarryでどう表現することにするかを決めよう。Starryの構文で決まっていることは以下だ。

1.「+」「*」「.」とスペースのみから構成される。
2. 見た目が星空っぽい。

1.では「なるべくシンプルな仕様のほうが言語として格好いい」という思想から４つの文字だけを構成要素として挙げたが、場合によっては「'」「`」「,」など、「.」と見た目が似ている文字を使うことにしてもいいかもしれない。

2.は言語のコンセプトなので絶対に譲れない点だ。「星空らしさ」をどう定義するかは難

しいところだが、「たくさんの空白の中に記号が点在している」とすればそこそこそれらしくなるだろう。

だがよく考えてみると、「星空っぽくしか書けない言語にする」のか、「星空っぽく書くことができる言語」にするのかは検討の余地がある。前者は「Starryの正しいプログラムはすべて星空っぽく見える」という意味で、後者は「Starryのプログラムの中には星空っぽく見えないものもあるが、プログラマに頑張って星空っぽくしてもらう」という意味だ。コンセプトが自動的に達成される仕様にするのか、それともプログラマの努力目標にするのか、という違いである。実際、Shakespeare（付録参照）では意味をなさない文章もプログラムとして許されるが、「Shakespeareプログラムはできるだけ文学的であるべきである」という指針を掲げることでプログラムの美しさを保っている。

言語の設計が楽なのはもちろん、プログラマの助けを借りるほうだ。Starryではプログラムが必ず星空っぽくなることは目指さないことにする。そのかわり、同じ処理をいろいろな書き方で行えるような柔軟性を取り入れることで、プログラマが記号をより星空らしくばらまけるようにすればいいだろう。

星空らしさを考慮するうえで重要なのは空白の比率だ。記号が2つ以上連続するのは見た目的に美しくないので、すべての記号は必ず1つ以上の空白で区切られるようにしたい。さらに、プログラマが密度を調整できるように、空白の個数にはいくつかの選択肢があるのが望ましい。

では、こういうのはどうだろうか。

1. 決められた記号とスペース以外の文字はすべて無視される。だから改行は好きなところに入れていい。
2. それぞれの記号によって、命令のカテゴリが異なる。例えば「+」はスタック操作、「*」は四則演算。
3. 命令の種類は、記号の前にいくつの空白があるかによって指定する。例えば「*」の前にいくつの空白があるかによって、+ - * / % のどの演算を行うかが決まる。
4. 空白の置き方の自由度を上げるため、空白の数を命令の数で割った「余り」で命令の種類を決める。

具体的に書くと、

```
「*」は足し算
「 *」は引き算
「  *」はかけ算
「   *」は割り算
「    *」は余り
「     *」は足し算
「      *」は引き算
「       *」はかけ算
「        *」は割り算
「         *」は余り
「          *」は足し算
             :
             :
```

のようになる。空白が０個から４個の場合だけあればすべての命令が書けるが、もっとたくさんの空白を入れる書き方も許される、というわけだ。こうすれば１つの記号で複数の命令を表すことができるし、プログラマが見た目を自由に調整することもできてよさそうだ。

意味論を決めよう

上の方針に従って、実際にそれぞれの命令の書き方を決めてみよう。用意する命令は

- スタックを扱う命令：push, dup, swap, rotate, pop
- 四則演算：+ - * / %
- 文字・数値の入出力命令
- 分岐命令

とのことだった。命令を４種類に分類すると「+」「*」「.」の３文字では表現しきれなさそうだが、適宜「,」「`」「'」も取り入れることにすれば問題ないだろう。

まず考えるべきは、「どの命令がよく使われるか？」ということだ。コンセプトに従うなら、プログラム中で一番多い記号は「.」や「,」ではなく、「*」や「+」でなくてはならない（そうでないとなんだか寂しい星空になってしまう）。

どんなプログラムを書くかにもよるが、大まかな使用頻度はおそらく「スタック命令＞四則演算＞入出力＞分岐命令」のようになるだろう。ということで、ここではスタック命令に「+」を、四則演算に「*」を割り当てることにしよう。四則演算は、

「*」 直前の空白の数を５で割った余りが０なら足し算、１なら引き算、２ならかけ算、３なら割り算、４なら余り

と定義すればいいだろう。

スタック命令も同じようにすればいい…と思いきや、それだとpush命令で「何という数値をプッシュするのか」が指定できない。なので、スタック命令は余りをとるのではなく、

「+」 空白の数が１ならdup命令、２ならswap命令、３ならrotate命令、４ならpop命令。５以上なら、空白の数から５を引いた数値をプッシュする

としよう。余りをとらないため、スタック命令の書き方はそれぞれ１種類しかない。dup命令なら「 +」だ。空白が０個のときはどうしようか。何か命令を割り当ててもいいが、そうすると記号が２つ以上続いてしまうので、見た目的によろしくない。「空白０個のあとに+がきたらエラー」としておこう。

残りは入出力と分岐だが、Brainf*ckで入出力に使われる記号、「,」と「.」がまだ割り当てられていない。入出力にこれらの記号を使うことにすれば、Brainf*ckプログラマ（が世界中にどれくらいいるのかわからないが）にとって覚えやすくていいだろう。なので、

「.」 空白の数を２で割った余りが０なら数値出力、１なら文字出力
「,」 空白の数を２で割った余りが０なら数値入力、１なら文字入力

としよう。

残った文字は「`（バッククオート）」と「'（シングルクオート）」だ。これらは英語の文章で単語をくくるために対になって使われる記号なので、Brainf*ckの「[」と「]」のように分岐とループを兼ねた命令を表現するのにぴったりだ。

だが、Brainf*ck方式を採用するとすると、「`」や「'」の前の空白の数には意味を持たせないことになり、ちょっともったいない。では、Whitespaceのようにラベルを使う方式はどうだろう？　つまり、

「`」 この位置がラベルになる（命令としては特に何もしない）
「'」 スタックトップが０以外なら、空白の数が同じ「`」までジャンプする

とするのだ。見た目こそBrainf*ckの命令に似ているが、「必ず正しい入れ子になっていなければならない」という制限がないし、1つの「`」に対して複数の「'」を設置することもできてよさそうだ。

Whitespaceのジャンプ命令と同じく、「'」は自分より後ろの「`」にもジャンプできることにしよう。また、Whitespaceではラベルの重複を許していたが（後に出てきたほうは単に無視される）、これはなんとなく気持ち悪いので、Starryでは空白の数が同じラベルが2つ以上あったらエラーにすることにする。

Hello, Starry world!

仕様が決まったところで、Hello Worldがどのように書けるか試してみよう。まず、もっとも単純に考えるなら、「Hello, world!」という文字列を出力するには

```
push 72, char_out, push 101, char_out, push 108, char_out,
push 108, char_out, push 111, char_out, push 44, char_out,
push 32, char_out, push 119, char_out, push 111, char_out,
push 114, char_out, push 108, char_out, push 100, char_out,
push 33, char_out
```

のような命令列を作ればいいから、StarryによるHello Worldプログラムは以下のようになる（「|」はそこに改行があることを示している）。

```
                                                    + .|
                                                       |
                    + .                                |
                                          + .          |
                                                       |
          + .                                          |
                                      + .              |
                 + .                          + .      |
                                                       |
                       + .                             |
                                                + .    |
                                                       |
                       + .                             |
                                                + .    |
                                                       |
              + .                       + .            |
```

これは正しいStarryプログラムだが、残念なことに、まったくもって星空には見えない。もう少し努力が必要そうだ。

どうも、あまり大きな数字を使うと記号と記号の間が開きすぎてよくないようだ。スタックと四則演算を使って、プッシュする数値が小さくて済むようにしてみよう。例えば最初の「H」を出力する部分は以下のようにもできる（「|」の右側はスタックの様子だ）。

```
push 7     | 7
push 10    | 7 10
*          | 70
push 2     | 70 2
+          | 72
char_out   |
```

これは、Starryプログラムでは以下のようになる。

```
              +                +  *       +     * + .
```

なかなかいい感じの星空になりそうだ。だが、この調子で「Hello, world!」の全文を表示するプログラムを解説していると長くなりそうなので、それは練習問題に譲るとしよう。かわりに略式のHello World、「Hi」を表示するプログラムを上の方針で書いてみると以下のようになる。

```
push 7
push 10
*
push 2
+
char_out

push 10
push 10
*
push 5
+
char_out
```

これをStarry化したものはリスト1のようになる（「|」は改行）。小さくとも、なかなか可愛い星空ではないだろうか？

リスト1:「Hi」という文字列を表示するStarryプログラム

```
           +               + |
 *       +     * .           |
   +               +  *      |
   +     * .
```

実装してみよう

では、Starryのインタプリタを実装してみよう。同じスタック型言語ということで、Whitespaceの処理系のコードがいろいろ流用できるはずだ。

実装方針についても、Whitespaceのときと同じように、まずプログラムの全体を「[:push, 1]」のような形式に変換し、それから実行するのがわかりやすいだろう。ただしWhitespaceではcompile.rbとvm.rbに処理を分けたが、Starryの文法はWhitespaceより単純なので、今回はファイルを分けず、クラスStarryにすべて書いてしまうことにする。

処理系のひな形は以下のようになるだろう。

```
# coding: utf-8

class Starry

  class ProgramError < Exception; end

  def self.run(src)
    new(src).run
  end

  def initialize(src)
    @insns = parse(src)
    @stack = []
    @labels = find_labels(@insns)
  end

  def run
```

```
    # あとで実装する
  end

  private

  def parse(src)
    # あとで実装する
  end

  def find_labels(insns)
    # あとで実装する
  end

end

Starry.run(ARGF.read)
```

基本的にWhitespace::VMクラスと同じなのだが、parseというメソッドが増えている。

parseメソッドは、Starryプログラムを文字列で受け取り、「[:push, 1]」のような表現に変換したものを返す。このようにソースコードを処理系が実行できる中間形式に変換するものを「パーサ」と呼ぶ※1。

※1：英語では parser。「パーサ」という読みと「パーザ」という読みがあるが、どちらかというと濁らないほうが一般的なようだ。

▌パーサの実装

まずはparseメソッドを実装しよう。四則演算だけ実装してみたのが以下だ。

```
OP_CALC = [:+, :-, :*, :/, :%]      # (1)

def parse(src)
  insns = []                        # (2)

  spaces = 0                        # (3)
  src.each_char do |c|              # (4)
    case c
    when " "
      spaces += 1                   # (5)
    when "*"
      op = OP_CALC[spaces % OP_CALC.size]  # (6)
      insns << [op]                        # (7)
```

```
      spaces = 0                              # (8)
    end
  end

  insns
end
```

まず準備として、四則演算の命令の一覧をOP_CALCという変数に保存している(1)。

parseメソッドでは、最初にパース結果を入れるinsnsという配列を用意する(2)。次に空白の数を数えるための変数spacesを用意し(3)、src.each_charを使って1文字ずつ順番に見ていく(4)。空白を見つけた場合はspacesを1増やす(5)。

「*」を見つけた場合は、空白の数をOP_CALC.size、つまり5で割った余りを求めて(%)、対応する命令をopに代入している(6)。四則演算の命令は引数をとらないので、insnsには[op]を追加する(7)。命令を1つ読み終わったので、spacesは0にしておく(8)。

入出力命令も同じように実装できる。

```
OP_CALC = [:+, :-, :*, :/, :%]
OP_OUTPUT = [:num_out, :char_out]
OP_INPUT = [:num_in, :char_in]

def parse(src)
  #(略)
  src.each_char do |c|
    case c
    when " "
      spaces += 1
    when "*"
      op = OP_CALC[spaces % OP_CALC.size]
      insns << [op]
      spaces = 0
    when "."
      op = OP_OUTPUT[spaces % OP_OUTPUT.size]
      insns << [op]
      spaces = 0
    when ","
      op = OP_INPUT[spaces % OP_INPUT.size]
      insns << [op]
      spaces = 0
```

```
      end
    end

    insns
  end
```

だが、命令を選ぶ部分が重複しているのが気になるので、ここをメソッドにまとめよう。以下ではselectというメソッド名にしてみた。

```
# (略)

def parse(src)
  # (略)
  src.each_char do |c|
    case c
    when " "
      spaces += 1
    when "*"
      insns << select(OP_CALC, spaces)
      spaces = 0
    when "."
      insns << select(OP_OUTPUT, spaces)
      spaces = 0
    when ","
      insns << select(OP_INPUT, spaces)
      spaces = 0
    end
  end

  insns
end

def select(ops, n)
  op = ops[n % ops.size]
  [op]
end
```

parseメソッドがすっきりした。

スタック命令はちょっと違う。空白が5個以上ならすべてpush命令になるため、空白が5個未満ならselectメソッドを使い、それ以上ならpush命令にする。

```
OP_STACK = [:__dummy__, :dup, :swap, :rotate, :pop]
#（略）

def parse(src)
  #（略）
  src.each_char do |c|
    case c
    when " "
      spaces += 1
    when "+"
      raise ProgramError, "0個の空白のあとに+が続きました" if spaces == 0
      if spaces < OP_STACK.size
        insns << select(OP_STACK, spaces)
      else
        insns << [:push, spaces - OP_STACK.size]
      end
      spaces = 0
    #
    #（略）
    #
    end
  end

  insns
end
```

上では「5個未満」と書いたが、実際には「5」ではなくOP_STACK.sizeを使うのがいいだろう。こうしておけば、言語仕様を変更して命令を追加したくなったときに、一箇所だけ変更すれば済むからだ。

空白が0個だった場合はProgramErrorだ。このため、OP_STACKの0番目は決して参照されない。ここでは「この値は使われませんよ」という意味で、:__dummy__というシンボルを置いている。

ジャンプ命令の実装は特に難しいことはない。命令の引数にはラベルを識別できるような値を置いておく。空白の個数を入れておくことにしよう。

```
#（略）

def parse(src)
```

```
    #（略）
    src.each_char do |c|
      case c
      #
      #（略）
      #
      when "`"
        insns << [:label, spaces]
        spaces = 0
      when "'"
        insns << [:jump, spaces]
        spaces = 0
      end
    end

    insns
  end
```

「`」はlabel命令、「'」はjump命令という名前にした。だが、Whitespaceのjump命令のように無条件でジャンプするのではなく、スタックの一番上の値が0でなかった場合のみジャンプすることに注意。もしもっと説明的な名前を付けたければ、jump_if_not_zero命令とか、jump_nonzero命令などのようになるだろう。

ラベルの探索

すべての命令がパースできるようになったので、次はfind_labelsメソッドを実装しよう。…といっても、これはWhitespace::VMのメソッドがほぼそのまま流用できる。

```
  def find_labels(insns)
    labels = {}
    insns.each_with_index do |(insn, arg), i|
      if insn == :label
        raise ProgramError, "ラベル#{arg}が重複しています" if labels[arg]
        labels[arg] = i
      end
    end
    labels
  end
```

Whitespaceの場合と違うのは、空白の数が同じラベルが複数あったらエラーにするようにしたことだ。

runメソッドの実装

では、それぞれの命令を実装していこう。runメソッドの構造はWhitespace::VMのそれとほぼ同じだ。違うのは、Whitespaceではexit命令で終わらないとエラーにしていたことくらいだ。pushメソッド、popメソッドもそのまま使うことができる。

```
def run
  pc = 0
  while pc < @insns.size
    insn, arg = *@insns[pc]

    case insn
      # あとで実装する
    else
      raise "[BUG] 知らない命令です(#{insn})"
    end
    pc += 1
  end
end

# (略)

def push(item)
  unless item.is_a?(Integer)
    raise ProgramError, "整数以外(#{item})をプッシュしようとしました"
  end
  @stack.push(item)
end

def pop
  item = @stack.pop
  raise ProgramError, "空のスタックをポップしようとしました" if item.nil?
  item
end
```

まずスタック命令の実装。これはWhitespaceとほとんど変わらない。唯一違うrotate命令は、スタックの上3つを並べ替える。スタックが「| x y z」となっているのを、「| z x y」のようにするのだった。

```
      case insn
      when :push
```

```
    push(arg)
  when :dup
    push(@stack[-1])
  when :swap
    y, x = pop, pop
    push(y)
    push(x)
  when :rotate
    z, y, x = pop, pop, pop
    push(z)
    push(x)
    push(y)
  when :pop
    pop
```

四則演算はWhitespaceそのままだ。

```
  when :+
    y, x = pop, pop
    push(x + y)
  when :-
    y, x = pop, pop
    push(x - y)
  when :*
    y, x = pop, pop
    push(x * y)
  when :/
    y, x = pop, pop
    push(x / y)
  when :%
    y, x = pop, pop
    push(x % y)
```

入出力は、Whitespaceがヒープに書き込んでいたのに対し、スタックに積むようにしたところが違う。

```
  when :num_out
    print pop
  when :char_out
    print pop.chr
```

```
        when :char_in
          push($stdin.getc.ord)
        when :num_in
          push($stdin.gets.to_i)
```

ジャンプ命令はスタックから値をポップし、0以外ならジャンプを試みる。

```
        when :label
          # ラベルの位置はすでに調べてあるので、何もしない
        when :jump
          if pop != 0
            pc = @labels[arg]
            raise ProgramError, "ジャンプ先(#{arg.inspect})が見つかりません"
if pc.nil?
          end
```

これで命令の実装は完了だ。

完成

完成したStarry処理系の全体をリスト2に載せた。

リスト2: 完成したStarry処理系

```
# coding: utf-8

class Starry

  class ProgramError < Exception; end

  OP_STACK = [:__dummy__, :dup, :swap, :rotate, :pop]
  OP_CALC = [:+, :-, :*, :/, :%]
  OP_OUTPUT = [:num_out, :char_out]
  OP_INPUT = [:num_in, :char_in]

  def self.run(src)
    new(src).run
  end

  def initialize(src)
    @insns = parse(src)
```

```
  @stack = []
  @labels = find_labels(@insns)
end

def run
  pc = 0
  while pc < @insns.size
    insn, arg = *@insns[pc]

    case insn
    when :push
      push(arg)
    when :dup
      push(@stack[-1])
    when :swap
      y, x = pop, pop
      push(y)
      push(x)
    when :rotate
      z, y, x = pop, pop, pop
      push(z)
      push(x)
      push(y)
    when :pop
      pop

    when :+
      y, x = pop, pop
      push(x + y)
    when :-
      y, x = pop, pop
      push(x - y)
    when :*
      y, x = pop, pop
      push(x * y)
    when :/
      y, x = pop, pop
      push(x / y)
    when :%
      y, x = pop, pop
      push(x % y)

    when :num_out
      print pop
    when :char_out
      print pop.chr
    when :char_in
      push($stdin.getc.ord)
```

```
      when :num_in
        push($stdin.gets.to_i)

      when :label
        # ラベルの位置はすでに調べてあるので、何もしない
      when :jump
        if pop != 0
          pc = @labels[arg]
          raise ProgramError, "ジャンプ先(#{arg.inspect})が見つかりません"
if pc.nil?
        end

      else
        raise "[BUG] 知らない命令です(#{insn})"
      end
      pc += 1
    end
  end

  private

  def parse(src)
    insns = []

    spaces = 0
    src.each_char do |c|
      case c
      when " "
        spaces += 1
      when "+"
        raise ProgramError, "0個の空白のあとに+が続きました" if spaces == 0
        if spaces < OP_STACK.size
          insns << select(OP_STACK, spaces)
        else
          insns << [:push, spaces - OP_STACK.size]
        end
        spaces = 0
      when "*"
        insns << select(OP_CALC, spaces)
        spaces = 0
      when "."
        insns << select(OP_OUTPUT, spaces)
        spaces = 0
      when ","
        insns << select(OP_INPUT, spaces)
        spaces = 0
      when "`"
        insns << [:label, spaces]
```

```
        spaces = 0
      when "'"
        insns << [:jump, spaces]
        spaces = 0
      end
    end

    insns
  end

  def select(ops, n)
    op = ops[n % ops.size]
    [op]
  end

  def find_labels(insns)
    labels = {}
    insns.each_with_index do |(insn, arg), i|
      if insn == :label
        raise ProgramError, "ラベル#{arg}が重複しています" if labels[arg]
        labels[arg] = i
      end
    end
    labels
  end

  def push(item)
    unless item.is_a?(Integer)
      raise ProgramError, "整数以外(#{item})をプッシュしようとしました"
    end
    @stack.push(item)
  end

  def pop
    item = @stack.pop
    raise ProgramError, "空のスタックをポップしようとしました" if item.nil?
    item
  end

end

Starry.run(ARGF.read)
```

サンプルコードとして、「H」という文字を表示するStarryプログラムを再掲する。できた処理系で実行して、ちゃんと動作するか確かめてほしい。