Rubyによる超絶技巧プログラミング

遠藤 侑介[a)]

概要：本稿は，実用性の観点を排除してプログラミング本来の美しさを追求する「超絶技巧プログラミング」を提案する．このプログラミングにおいては，実用上意味のない目的や制約を設定し，それを充足するプログラムを書く方法を開発する．さらに，Ruby における超絶技巧プログラミングの実例を，その実装技法の解説と共に 2 つのテーマに分けて紹介する．1 つは self-descriptive な Ruby プログラム，もう 1 つは使用可能文字を制限した Ruby プログラムを書く方法である．これらの事例を通して超絶技巧プログラミングの魅力を示し，また，プログラミング言語 Ruby の神秘性の一端を解き明かす．

キーワード：超絶技巧プログラミング, 難解プログラミング, quine, Ruby

Transcendental Programming in Ruby

Yusuke Endoh[a)]

Abstract: We propose *transcendental programming*, which quests for the inherent beauty of programming in disregard of usefulness. In our proposed programming, we set a non-practical goal and/or constraint, and then develop a technique for writing a program that satisfies them. In addition, we show two case studies of transcendental programming, with explanation of the implementation techniques. One is self-descriptive Ruby programs, and the other is how to write Ruby program under a constraint that requires some kinds of characters not be used. Through these studies, we show the fun of transcendental programming, and illustrate some of the subtleties of the Ruby programming language.

Keywords: transcendental programming, esoteric programming, quine, Ruby

1. はじめに

「美しいプログラム」という時，その「美しさ」は「機能美」を指していることが多い．例えば

- よく抽象化されていて無駄のない構造を持つ，可読性やメンテナンス性の高いプログラム
- 高度な観察に裏付けされた工夫による，実行性能のよいアルゴリズム
- 組み込みシステムやブラウザなど，実行環境に厳しい制約がある中で動作するプログラム

などが美しいと言われることが多い．ここで共通しているのは，いずれも実用性を追求しているということである．

しかし，人間が感じる美しさは実用性に基づいたものばかりではない．例えば簡単にできるはずのことを，敢えて手の込んだからくりの連鎖によって実現するルーブ・ゴールドバーグ・マシン[*1]や，紙を折ることで動植物や生活道具の形を作る折り紙などは，実用性とは無縁だが多くの人が美しさを感じる．プログラミングもこれらと同様に，実用性とは無縁の面白さ・美しさを持つものであるが，この点について議論されることは多くない．

そこで我々は，実用上意味のない目的・制約・テーマを定め，それを達成するプログラムを作成する超絶技巧プログラミングを提案する．これを通して，プログラミング本来の面白さ・美しさを追求したり，使用するプログラミング言語の神秘性を解き明かしたりすることができる．本稿ではさらに，我々がこれまで取り組んできた超絶技巧プログラミングの実例を紹介する．すべてを紹介することは紙面の都合上難しいため，今回は使用するプログラミング言語を Ruby[28] に限定し，さらにテーマを次の 2 つに限定

a) mame@tsg.ne.jp

*1 NHK 教育テレビの番組「ピタゴラスイッチ」に登場する「ピタゴラ装置」の名でも知られる．

```
                                alias|send\
                            ;$stdin=GC | "%p?"%def#
                        FALSE.gets();(8  |  64).chr+232424.
                  to_s(25)+", "+%w|w !   |   *"orlc". next<<012||
            (c).Yusuke end;"oh, 2009"    |    "stegano-X."[0,4].reverse
d,be="whydoes","crypto";:make.           |     %.mains..tr'eams',be.delete(d)
```

図 1　最悪な Hello, world!
Fig. 1　Hello, world! from hell

する．1 つは self-descriptive なプログラム，もう 1 つは使用する文字を制限したプログラムである．

本稿の残りの構成は以下の通りである．2 節では超絶技巧プログラミングの定義を与える．3 節では関連研究について触れる．4 節では self-descriptive な Ruby プログラムの事例とその実装技法を紹介する．5 節では使用可能文字を制限して Ruby プログラムを書く方法について議論する．

2. 超絶技巧プログラミング

本稿では，実用上意味のない目的を定め，その目的を達成するプログラムを無用に回りくどい方法で書いたり，実用上意味のない制約を見つけ，それを満たすプログラムを書く方法を模索したりする行為を超絶技巧プログラミングと呼ぶ．超絶技巧プログラミングは特定のプログラミング言語に特化した概念ではないが，実用的なプログラミング言語を用いることを前提としている．

Hello, world!プログラムを題材として超絶技巧プログラミングの具体例を示す．Ruby で Hello, world!を普通に書くと以下のようになる．

```
puts "Hello, world!"
```

これに対し，Ruby の多数の機能を経由して，なるべく回りくどく書いた Hello, world!を図 1 に示す．これを実行すると，"Hello, world!"を出力する．

```
$ ruby hello.rb
Hello, world!
```

どこで"Hello, world!"という文字列を作っているか，また，どの命令で出力しているか，一見ではわからない．このプログラムの動作については付録 A.2 で解説する．

「超絶技巧」という言葉は，Franz Liszt 作のピアノ曲 “Transcendental Ètudes（超絶技巧練習曲）”[22] に由来する．超絶技巧練習曲は非常に高度な演奏技術を必要とすることで有名な曲である．さらに Transcendental は元々「超越論的」「超自然的」という哲学的な意味を持つ単語である．この両方の意味において，ある種の高度な技術を必要とし，俗世での即物的な目的を持たない超絶技巧プログラミングはこの名前が適切であると考えた．

3. 関連研究

超絶技巧プログラミングは全く新しい概念ではない．例えば **International Obfuscated C Code Contest (IOCCC)** のいくつかのエントリはまさに超絶技巧プログラミングをしていると言える．IOCCC は故意に読みづらい C 言語プログラムを書き，その汚さを競うコンテストである．その目的は，プログラミング作法の重要性を反面教師的に訴えたり，C 言語の神秘性を示したりすることなどである．特定の言語に特化してはいるが，超絶技巧プログラミングに非常に近い思想を持つと言える．

Just Another Perl Hacker (JAPH) は，”Just Another Perl Hacker” という文字列を出力する Perl プログラムを多種多様な方法で記述するハッカーの嗜みである．ある面では超絶技巧プログラミングに似ているが，JAPH は Perl に特化している点に加え，目的・仕様が最初から決まっている点が大きく異なる．超絶技巧プログラミングは，意味のない目的・仕様を見つけ出すことに大きな意味があると我々は考える．

難解プログラミング言語 (esoteric programming language)[9] と呼ばれる，プログラムの読解が困難になるように設計されたプログラミング言語も多数存在する．Brainfuck[4]（機械語に近い 8 命令のみを持つ言語），Whitespace[31]（スペース，タブ，改行のみで記述する言語），Unlambda[30]（SKI コンビネータを直接記述する言語）などが有名である．当然，実用性を目指したものではなく，ハッカーの間で行われる遊びである．超絶技巧プログラミングはこれらと全く同じ精神を持つ．ただし超絶技巧プログラミングでは，一般的に用いられるプログラミング言語を用いることを前提としている．

コードゴルフ（ショートコーディングとも言う）[2], [5] は，仕様を満たすプログラムをできる限り短く書く競技で，いわばコルモゴロフ複雑性を実際に測定しようと挑戦する試みのようなものである．コードゴルフではプログラムの長さをバイト数単位で競うため，省略可能なインデントや改行がすべて取り除かれる．一般に「短くて簡潔なプログラムは美しい」と言われるが，コードゴルフのプログラムは同じ意味で「美しい」とは言えない．コードゴルフに必要な技術と超絶技巧プログラミングに必要な技術には

重複するものが多々あるが，コードゴルフはプログラムの長さを短くするための深い知識がひたすらに必要とされる一方，超絶技巧プログラミングでは「プログラムの長さ」のような所与の評価基準がないため，より自由な発想と広範な技術が求められるという違いがある．

より具体的な関連技術については，4 節と 5 節にて，各節のテーマに直接関連するものに絞って挙げる．ただし，超絶技巧プログラミングの関連研究はブログやメール等で散発的に議論されることが多いため，我々が関連研究を網羅的に把握できているわけではないことをあらかじめ断わっておく．広く調査して包括的にまとめたサーベイが必要である．

4. Case Study 1: Self-Descriptive

本節では，self-descriptive な Ruby プログラムというテーマに基づいたプログラム例を示す．

なお，本稿で説明する Ruby プログラムはすべてプログラミング言語 Ruby の標準実装である ruby で動作確認している．使用したバージョンは ruby 1.9.3p286 (2012-10-12 revision 37165) [x86_64-linux] である．

4.1 実例: FizzBuzz

FizzBuzz はプログラミングの学習でしばしば用いられる題材である．数字を 1 から順に出力するが，3 で割り切れる場合には数字の代わりに “Fizz” を，5 で割り切れる場合には “Buzz” を，両方で割り切れる場合には “FizzBuzz” を出力する．

超絶技巧プログラミングで self-descriptive な FizzBuzz を実装した一例を図 2 に示す．このプログラムは，“1” という数字の字形を模しており，プログラム自身が最初に出力する数字を表現している．

このプログラムを実行した一連の様子を図 3 に示す．図 2 のプログラムを実行すると，“2” という字形のアスキーアートが出力される．この出力文字列も Ruby プログラムとなっており，これを再度実行すると “Fizz” という字形のアスキーアートが出力される．以上のように，実行するたびに，次の FizzBuzz の出力の字形に整形されたプログラムを出力するプログラムとなっている．

```
          eval(s=s=
          %w@proc{|
          n|z=32.ch
          r;k="[#{n
          +=1}]";u=
":>==;<==?"[m=n**4
%-15,m+13]||"#{$f=
k}";d="Y.E.#{c=64.
chr}*'')";$f||d<<z
+k;t="eval(s=s=%w#
          {c+s=s[0,
          334]}#$f#
          ";25.time
          s{|y|m=u.
          bytes.map
          {|v|t<<s;
          (0..[62-v
          ,2].min).
          map{|x|"i
          f0zgl11p0
zghuhku744d8hzeg41qtfx7xs7t
wflr".to_i(36)[x+32+v*3-y/5
*44]<1?z*9:t.slice!(0,9)}<<
z}.join.rstrip;y>23&&m[-9,9
]=d;puts(m)}}[1]#pY.E.@*'')
```

図 2 self-descriptive FizzBuzz
Fig. 2 self-descriptive FizzBuzz

```
$ cat fizzbuzz.rb
        eval(s=s=
        %w@proc{|
        n|z=32.ch
        r;k="[#{n
        +=1}]";u=
":>==;<==?"[m=n**4
%-15,m+13]||"#{$f=
k}";d="Y.E.#{c=64.
chr}*'')";$f||d<<z
+k;t="eval(s=s=%w#
        {c+s=s[0,
        334]}#$f#
        ";25.time
        s{|y|m=u.
        bytes.map
        {|v|t<<s;
        (0..[62-v
        ,2].min).
        map{|x|"i
        f0zgl11p0
zghuhku744d8hzeg41qtfx7xs7t
wflr".to_i(36)[x+32+v*3-y/5
*44]<1?z*9:t.slice!(0,9)}<<
z}.join.rstrip;y>23&&m[-9,9
]=d;puts(m)}}[1]#pY.E.@*'')
$ ruby fizzbuzz.rb
eval(s=s=%w@proc{|n|z=32.ch
r;k="[#{n+=1}]";u=":>==;<==
?"[m=n**4%-15,m+13]||"#{$f=
k}";d="Y.E.#{c=64.chr}*'')"
;$f||d<<z+k;t="eval(s=s=%w#
                {c+s=s[0,
                334]}#$f#
                ";25.time
                s{|y|m=u.
                bytes.map
{|v|t<<s;(0..[62-v,2].min).
map{|x|"if0zgl11p0zghuhku74
4d8hzeg41qtfx7xs7twflr".to_
i(36)[x+32+v*3-y/5*44]<1?z*
9:t.slice!(0,9)}<<z}.join.r
strip;y>2
3&&m[-9,9
]=d;puts(
m)}}[2]#p
roc{|n|z=
32.chr;k="[#{n+=1}]";u=":>=
=;<==?"[m=n**4%-15,m+13]||"
#{$f=k}";d="Y.E.#{c=64.chr}
*'')";$f||d<<z+k;t="eval(s=
s=%w#{c+s=s[0,334]Y.E.@*'')
$ ruby fizzbuzz.rb | ruby
eval(s=s=%w@proc{|n|z=32.ch
r;k="[#{n+=1}]";u=":>==;<==
?"[m=n**4%-15,m+13]||"#{$f=
k}";d="Y.E.#{c=64.chr}*'')"
;$f||d<<z+k;t="eval(s=s=%w#
{c+s=s[0,                   334]}#$f# ";25.times{|y|m=u. bytes.map{|v|t<<s;
(0..[62-v                   ,2].min). map{|x|"if0zgl11p0 zghuhku744d8hzeg41
qtfx7xs7t                   wflr".to_ i(36)[x+32+v*3-y/5 *44]<1?z*9:t.slice
!(0,9)}<<                   z}.join.r strip;y>23&&m[-9,9 ]=d;puts(m)}}#proc
{|n|z=32.                   chr;k="[# {n+=1}]";u=":>==;< ==?"[m=n**4%-15,m+
13]||"#{$f=k}";d="Y.E.#{c=6                 4.chr}*''           )";$f||d<
<z+k;t="eval(s=s=%w#{c+s=s[                 0,334]}#$           f#";25.ti
mes{|y|m=u.bytes.map{|v|t<<                 s;(0..[62           -v,2].min
).map{|x|"if0zgl11p0zghuhku                 744d8hzeg           41qtfx7xs
7twflr".to_i(36)[x+32+v*3-y                 /5*44]<1?           z*9:t.sli
ce!(0,9)}                   <<z}.join .rstrip;y          >23&&m[-9
,9]=d;put                   s(m)}}pro c{|n|z=32          .chr;k="[
#{n+=1}]"                   ;u=":>==; <==?"[m=n          **4%-15,m
+13]||"#{                   $f=k}";d= "Y.E.#{c=          64.chr}*'
')";$f||d                   <<z+k;t=" eval(s=s=          %w#{c+s=s
[0,334]}#                   $f#";25.t imes{|y|m=u.bytes. map{|v|t<<s;(0..[6
2-v,2].mi                   n).map{|x |"if0zgl11p0zghuhk u744d8hzeg41qtfx7x
s7twflr".                   to_i(36)[ x+32+v*3-y/5*44]<1 ?z*9:t.slice!(0,9)
}<<z}.joi                   n.rstrip; y>23&&m[-9,9]=d;pu ts(m)}}proc{|n|z=3
2.chr;k="                   [#{n+=1}] ";u=":>==;<==?"[m= n**4%-15,Y.E.@*'') [3]
```

図 3 self-descriptive FizzBuzz の動作
Fig. 3 The execution of self-descriptive FizzBuzz

4.2 典型的な実装技法

self-descriptive FizzBuzz で用いている主な実装技法を説明する．(1) プログラム自身を文字列として得る自己複製，(2) 文字列を数字のアスキーアートに整形するためのフォント埋め込み，(3) アスキーアートの形状で動作するプログラムとして書く実行可能アスキーアート化である．

4.2.1 自己複製

そのプログラム自身を文字列として出力するプログラムを Quine[27] と呼ぶ．他に，自己複製プログラム，自己

出力プログラム，Print Me! などと呼ばれることもある．ハッカーの嗜みとして知られる有名なテーマである．

Ruby のように eval（文字列をプログラムとして実行する命令）のある言語では，以下のように比較的簡潔に Quine を書くことができる（eval がない場合の Quine や他の技法は [16] を参照せよ）．

```
1: eval s="
2:   s = 'eval s=' + s.inspect;
3:   puts(s)
4: "
```

ただし改行は説明のために入れたもので，実際には改行しない．このプログラムはほぼ全体が文字列リテラルであり，その文字列を eval する構造になっている．まず変数 s に全体の文字列を代入する．次にその文字列が eval され，文字列の中身の実行が始まる．inspect はオブジェクトを人間に読みやすい形式に変換する Ruby の組み込みメソッドであり，ここでは文字列の最初と最後にダブルクォートを追加する効果を持つ[*2]．これによって 2 行目では，元のプログラム全体を文字列として復元している．最後に 3 行目でこの文字列を出力する．

self-descriptive FizzBuzz では，後述するアスキーアート化のために異なる文字列リテラルを用いていたり，それに応じて inspect 以外の方法でプログラムを復元していたりする点が異なるが，基本的には同じ構造である．

4.2.2 フォント埋め込み

self-descriptive FizzBuzz は，プログラム自体を次に出力する文字列の形状に整形する必要がある．原理的には好きな方法で整形すればよいが，このプログラムが長くなるとプログラムが巨大化するため，アスキーアート自体も巨大化してしまい，美しくない．そこで，多倍長整数の基数変換を用いた，簡単なフォント埋め込みを述べる．

self-descriptive FizzBuzz には，最大 3×5 ピクセルのビットマップフォントが以下の 15 文字分含まれている．(210 バイト)

```
###  #   ###  ###  # #  ###  ###  ###  ###  ###  ###  ##
# #  ##    #    #  # #  #    #    # #  # #  #    #    # #  # #  ##  #
# #   #   ###  ###  ###  ###  ###   #   ###  ###  ###  ##   # #   #
# #   #   #      #    #  # #  #  #  # #    #  #    #    # #  # #  #  #
###  ### ###  ###    #  ###  ###   #   ###  ###  #    ##   ##  ##  #
```

を 1 に，空白を 0 に置き換え，210 ビットの 2 進数として解釈し，その値を 10 進数にすると 64 バイトで表現できる．

```
1516849080105099949603183465996815374383250973658966149057739167
```

Ruby は組み込みで多倍長整数をサポートしているので，これをこのまま整数のリテラルとして書くことができる．さらに Ruby では，文字列を整数に変換するメソッド to_i が 2 進数から 36 進数[*3] までサポートしており，これを活用して更に短くすることができる．(52 バイト)

```
"8hlduayacv7bl0a0h2m2d2ti20qewh0rrsjcmsgpr".to_i(36)
```

なお，この整数から指定ドットの情報を取り出すには，C 言語のようにビットシフトで取り出してもよいが，2 進数の n 桁目を取り出すメソッド Integer#[] が組み込みに存在するのでこれを用いた方が簡潔である．

4.2.3 実行可能アスキーアート化

Ruby は%記法という柔軟性の高いリテラルを持つ[*4]．アスキーアート化で重要なのは%w(...) という文字列の配列のリテラルである．%w(foo bar) と書いた場合，"foo"という文字列を第一要素に持ち，"bar"という文字列を第二要素に持つ配列を表す．

プログラムをアスキーアート化するには，まずプログラム全体を%w(...) で囲み，その内部に空白を自由に追加して整形する．そして，配列の各要素を順に結合した文字列を得るメソッド Array#join を適用すると，元のプログラム全体が文字列となる．最後にこの文字列を eval することで，元のプログラムが実行される．(図 4)

この際の注意点として，元のプログラムは空白とバックスラッシュを含まないように作成する必要がある．この制約は容易に回避できる．例えば空白文字を出力する必要があれば，整数からその ASCII コードを持つ文字に変換するメソッド Integer#chr を用い，32.chr として作成できる[*5]．どうしてもバックスラッシュが必要であれば，92.chr などでバックスラッシュ文字を生成し，これを元に本来実行したいプログラム文字列を生成し，再度 eval するようなプログラムに変換すればよい．ただし，経験上バックスラッシュを回避できなかったことはない．

```
eval %w(
pu
  ts
    'H
      i'
).join##
```
⇒ `eval "puts'Hi'"` ⇒ `puts'Hi'`

図 4　実行可能なアスキーアートの実行の様子
Fig. 4　The execution of an executable ASCII-art

4.3 self-descriptive の他の実例

更に，我々がこれまで行なってきた self-descriptive な超絶技巧プログラミングの実例を示す．

4.3.1 Qlobe

図 5 は self-descriptive な地球儀プログラムである．プログラム中に文字列リテラルで地球儀のアスキーアートが含まれている．実行すると自分自身を，地球儀の部分を 45 度[*6] 回転させて出力する．

実装は以下の通りである．まず，self-descriptive FizzBuzz

[*2] 改行やタブなどの制御文字がある場合はエスケープ付き文字に置き換えられるが，ここではそのような文字は使用していない．
[*3] 16 進数が"0"から"9"，"a"から"f"の文字で表現するように，36 進数は"0"から"z"で表現する．
[*4] この記法は Perl 由来であるため，この節の実装技法は Perl でも同様に使用可能であると考えられるが，実際に確認はしていない．
[*5] printf の"%c"を用いる方法や，アスキーアートの形状に仮定が置ける場合は特定の位置に文字列リテラルを置く方法もある．
[*6] 回転角はコマンドライン引数で 1 度単位で指定可能である．

図 5　Qlobe: 地球儀 Quine

Fig. 5　Qlobe: Quine of Rotating Earth

図 6　山手 Quine

Fig. 6　Yamanote-Line Quine (Yamanote-Line: a loop line operated by East Japan Railway)

と同様に自己複製を行う．また，zlib で圧縮された世界地図データがプログラムに埋め込まれており，これを用いて地球儀をレンダリングする．プログラム中に割り込む地球儀の文字列は，ショートカット演算子やセミコロンで回避している．

なお，このプログラムは本発表の後で，西尾によって Python に移植された [26]．

4.3.2　山手 Quine

図 6 は JR 山手線の各駅を順に出力するプログラムである．図 6 の状態では「東京」を表し，これを実行すると「有楽町」を出力する．更に実行を続けると，「新橋」，「浜松町」，……と進む．29 回の実行で元の「東京」に戻るので，このプログラムは Quine の亜種[*7] と言える．

4.3.3　15quzzle

図 7 は，15 パズルを表すプログラムである．コマンドライン引数に”u”，”d”，”l”，”r”のいずれかを与えて実行すると，パネルを上下左右に動かした自分自身を出力する．

このプログラムは各パネルの下に数字を表す行が挿入される．この行はショートカット演算子||によって無視している．しかしこの行が挿入される箇所は，（パズル上の）空白の位置によって変わるため，挿入されうる箇所全てに||を配置している．

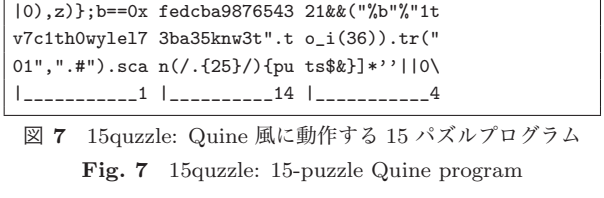

図 7　15quzzle: Quine 風に動作する 15 パズルプログラム

Fig. 7　15quzzle: 15-puzzle Quine program

4.4　関連研究

プログラム自身の形状に意味を持たせる試みは，IOCCC[18] に多数見られる．例えば 1988 年の大会の Westley によるエントリ [21] は，円の形状を持ち，円周率を計算するプログラムである．2011 年の大会の浜地 [12] や Hou[17] によるエントリは，それぞれ，ピクロスの形状をしたピクロスソルバ，電卓の形状をした電卓プログラムである．これらのプログラムの共通点は，プログラムの振る舞いを象徴する形状をしていることである．本節で紹介したプログラムのように，プログラムの振る舞いを象徴するだけでな

[*7] 狭義では，Quine は 1 回実行で自分自身を出力しなければならない．よって，このプログラムのように複数回の実行を経て元のプログラムに戻るプログラムは，Quine と区別して Multiquine などと呼ばれることがある．

く，出力自体も形状に意味のあるプログラムになっているような例は我々の知る限り多くない．

特筆すべき例外として，2000 年の大会の Yang によるエントリ [32] が挙げられる．これは，「あく」という形状のプログラムとなっていて，これを実行すると「そく」という形状に，次は「ざん」という形状に，そしてさらに実行すると「あく」に戻るというプログラムである．特に山手 Quine はこのプログラムに着想を得て作成した．

5. Cast Study 2: 文字制限

本節では，小文字アルファベットのみ[*8]を用いて任意の Ruby プログラムを書く，という超絶技巧プログラミングについて解説する．芸術と制約はしばしば合わせて語られる．厳しい制約によって不可能と思われることを実現するのは，一種の美しさを持つと言えよう．

実装戦略は，(1) 実行したいプログラム文字列を構築し，(2)eval する，という方針に基づく．Ruby では (2) は eval メソッドを呼ぶだけであるため，困難なのは (1) である．本節では「実行したい（文字制限されていない）プログラム」から「そのプログラム文字列を構築する（文字制限された）プログラム」へのエンコード方法を中心に説明する．

以下の説明では，比較的緩い文字制約の下で任意の Ruby プログラムを書く方法から始め，段階的に制約を強めていく．

制約 **1**　アルファベット（大文字含む）と数字のみで書く
制約 **2**　小文字アルファベットと数字のみで書く
制約 **3**　小文字アルファベットのみで書く

5.1 アルファベット（大文字含む）と数字のみで書く

最初に，本節の基本的なアイデアは [15] が示したものであることを断っておく．

アルファベットと数字のみで Ruby プログラムを書くにあたっては，(1) 文字列リテラルを使えない，(2) メソッド呼び出しができない，という 2 つの問題がある．以下，それぞれの解決方法を示す．

5.1.1 文字列リテラルを避ける

Ruby で文字列を構築するには通常，文字列リテラルを用いる．文字列リテラルの記法はダブルクォートを用いるため，当然今回の制約下では使用できない．Ruby には他に%記法の文字列リテラルがあるが，これも同様に記号を用いるため使用できない．

これを解決するには，まず文字列を返す組み込み関数を用いて種となる文字列を作り，この文字列に編集を行なって所望のプログラム文字列を構築する．

[15] では，種となる文字列として `String nil` を使用している．これは `nil` 値を文字列型に型変換するという意味で，空文字列となる．この種となる空文字列に，`String#concat` メソッドを用いて文字を 1 文字ずつ追加していく．

```
eval( (String nil).
        concat(112).
        concat(32).
        concat(49))
```

図 8　文字列リテラルを用いずに “p 1” をエンコードしたプログラム
Fig. 8　An encoded program for “p 1” with no string literals

図 8 は，“p 1” という 3 文字（ASCII コードは 112，32，49）からなるプログラムを上記の方法でエンコードしたものである．

5.1.2 メソッド呼び出し構文を避ける

上記のプログラムはメソッド呼び出しでピリオドや括弧を用いているため，当初の制約を満たしていない．

ここでは，`for` 文を用いることでメソッドを呼び出す．Ruby の `for` 文は `each` メソッドを呼び出す糖衣構文となっている．つまり，

```
for 変数 in コレクション do
  ...
end
```

は，

```
コレクション.each do |変数|
  ...
end
```

とほぼ同じ意味である[*9]（Ruby になじみのない読者は，`do ... end` は引数としてラムダ式を渡していると思えばよい）．このため，`for` 文を用いることで `each` という名前のメソッドを呼び出すことができる．

ただし，前述のエンコードでは `concat` という名前のメソッドを呼び出す必要がある．これを解決するために，open class と呼ばれる，既存クラスに後からメソッド定義を追加・再定義する機能を用いる．これを用いて `String` クラスに `each` メソッドを追加定義し，その中で `concat` メソッドを呼べばよい．この際，レシーバが `self` である場合には，レシーバを（ピリオドごと）省略できることに注意せよ．

以上をプログラムとしてまとめた例を図 9 に示す．

5.2 小文字アルファベットと数字のみで書く

次に，大文字アルファベットを用いない方法を述べる．前節の方法から大文字アルファベットを排除すればよい．図 9 の中で大文字アルファベットを使用しているのは，1

[*8] 空白文字（スペースおよび改行）も許容することにする．

[*9] 厳密には変数のスコープが異なる．前者では外側のスコープと同じであり，後者はブロックのために新しいスコープを作る．

```
 1: class String
 2:   def each
 3:     concat 112
 4:     concat  32
 5:     concat  49
 6:     eval self
 7:     exit
 8:   end
 9: end
10: for i in String nil do
11: end
```

図 9　アルファベットと数字のみで“p 1”をエンコードしたプログラム
Fig. 9　An encoded program for “p 1” with only alphabets and digits

```
public
def each
  clear
  concat 112
  concat  32
  concat  49
  eval self
  exit
end
for i in inspect do
end
```

図 10　小文字アルファベットと数字のみで“p 1”をエンコードしたプログラム
Fig. 10　An encoded program for “p 1” with only lower-case alphabets and digits

行目と 10 行目の String のみであり，これらを排除すれば十分である．

1 行目は，String クラスに each メソッドを定義するため，String クラスを open class で開く行である．代わりに，String クラスの親クラスである Object クラスにメソッドを定義する．ここで有用なのは，Ruby においてトップレベルで定義されたメソッドは Object クラスに所属するという言語仕様である．ただしデフォルトの状態では private メソッドとして定義されるため，public と書くことで[*10]public メソッドであることを明示する必要がある．

10 行目は，空文字列を生成するために String nil を用いている．これを排除するには，文字列を返す他のメソッド呼び出しに置き換えればよい．ここでは inspect メソッドを代替として用いる．このメソッドは空でない文字列を返すが，Object#each の最初で String#clear メソッドを呼べば文字列の内容を捨てて空にすることができる[*11]．

以上の方法で，図 9 から大文字を排除したプログラムを図 10 に示す．

5.3　小文字アルファベットのみで書く

最後に，数字を排除する方法を述べる．

基本的なアプローチは，所望の数字の長さの文字列を作り，String#size メソッドを呼び出すことである．図 11 にそのプログラム例を示す．まず String#clear を用いて，文字列の長さを 0 とする．次に String#concat に適当な数字を与えて，文字列の長さを 1 とする（適当な数字はここでは String#size で得ている）．所望の長さになるまで concat size を繰り返してもよいが，concat self を用いることで長さを 2 倍にすることができる．最後に

```
clear         # ""
concat size   # "\0"
concat self   # "\0\0"
concat self   # "\0\0\0\0"
concat self   # "\0\0\0\0\0\0\0\0"
size          # 2^3 = 8 が得られる
```

図 11　小文字アルファベットのみで数 8 を作る方法
Fig. 11　How to create number 8 with only lower-case alphabets

String#size によって所望の数字を得られる．

しかしこの方法は，String#clear を呼ぶため，これまでに構築してきたプログラム文字列を破壊してしまうという問題がある．これを解決するにはいくつか方法があるが，ここでは例外処理の構文を利用する方法を述べる．

Ruby における例外処理の構文 begin A ensure B end は，A を評価し，次に（A の評価中で例外が起きた場合でも起きなかった場合でも）B を評価し，最後に A の評価結果を返す．Java における try { A } finally { B }に似た構文であるが，B の評価中に A の返り値を保持して最終的な返り値とする点が異なる．

この構文を入れ子で用いて，文字列の先頭に任意の 1 文字を追加するイディオムを図 12 に示す．このイディオムではまず現在の文字列を複製する（2 行目）．この返り値は外側の begin ... end の返り値として保持される．4 行目に実行が移り，5 行目で所望の数字を作成して返す．この際，現在の文字列は破壊され，無意味な文字列となってしまう（空文字列ですらないことに注意せよ）．返された数字は内側の begin ... end の返り値として保持される．そして 7 行目に実行が移り，現在の文字列が空となる．以上で内側の begin ... end の評価がすべて完了し，生成された数字が 4 行目の concat に引数として渡される．現在の文字列は 7 行目の clear で空にされているため，文字列はこの数字の ASCII コード 1 文字からなる文字列とな

[*10] Ruby では public はキーワードではなく，その後定義されるメソッドを public とする，public という名前のメソッドである．
[*11] Java や C++のオブジェクト指向では，このように親クラス（Object）のメソッドから子クラス（String）のメソッドを呼び出すことは許されていないが，Ruby では許されている．メソッド探索が完全に実行時に行われるためである．

```
1: concat begin
2:   dup
3: ensure
4:   concat begin
5:     (所望の数字を作る)
6:   ensure
7:     clear
8:   end
9: end
```

図 12　文字列の先頭に所望の 1 文字を追加するイディオム
Fig. 12　An idiom that prepends any character to the string

る．ここで外側の `begin ... end` の評価が完了し，最初に保持しておいた元々の文字列を返す．この文字列は現在の文字列に追加される（1 行目）．以上のような仕組みで，このイディオムは元の文字列の先頭に任意の 1 文字を追加する．

以上で小文字アルファベットのみで任意の Ruby プログラムを書く方法が完成した．この方法に基づいて実際に任意のプログラム文字列をエンコードするプログラムと，この方法で作成した Hello, world!プログラムを付録 A.3 に示す．

5.4　関連研究

使用可能文字を制限してプログラムを書く試みには多様なバリエーションが存在する．Ruby では，記号のみ [20]，数字のみ*12[7]，アンダースコアのみ*12[8] といった関連研究が存在する．Perl では，記号のみ [1]，予約語のみ [29]，Javascript では，記号のみ [10]，顔文字のみ [11] がある．本節で述べた「小文字アルファベットのみ」の制限は，竹迫の発表 [29] に着想を得たものである．竹迫は予約語のみを用いており，本節より厳しい制限である．Ruby において純粋に予約語のみでプログラムを書く問題は未解決である．

Python ようにインデントに意味のある言語においては，改行を制限すること（ワンライナー）がしばしば厳しい制約となる*13．西尾がこの制限に取り組んでいる [24][25]．

表示可能な ASCII 文字に制限した機械語プログラムは，ソフトウェアセキュリティの分野で脆弱性の攻撃方法として研究されている [3]．文字制限という点では似ているが，実用上の動機が明確であり超絶技巧プログラミングではない*14．

最後に本節の内容に関する後日談を記す．本節の方法は Ruby 1.9 の機能を使用しているが，浜地は Ruby 1.8 *15 でも動作する方法を議論している [13]．また浜地は，文字種の少なさを競うコードゴルフの亜種競技を開催した [14]．西尾は Python で同様の使用文字制限に挑戦した [23]（小文字アルファベットと括弧とカンマのみ）．

*12 特定のライブラリのロードを要する．
*13 Haskell もインデントに意味があることで有名な言語だが，Haskell ではインデントを用いた構文が用いない構文への糖衣構文として提供されているため，特に困難はないと思われる．
*14 逆に，本節の内容を攻撃方法として応用することも考えにくい．記号文字をすべて排除する必要があるは考えにくいためである．
*15 ただし 1.8 系列は既に開発が停止しており使用は勧められない．

6.　おわりに

実用性の観点を排除してプログラミング本来の面白さ・美しさを追求する超絶技巧プログラミングを提案した．また，その実践例として，我々がこれまで取り組んできた超絶技巧プログラミングの実例とその実装技法について，2 つのテーマに絞って紹介した．1 つは self-descriptive な Ruby プログラムというテーマで，プログラムの形状がそのプログラムの振る舞いを表し，出力自体もプログラムになっているようなプログラムを扱った．もう 1 つは文字制約というテーマで，使用する文字を制限してプログラミングする方法を議論した．

しかし超絶技巧プログラミングはこの 2 つのテーマに限定したものではなく，あくまで例に過ぎないことを明記しておく．プログラムを書くという行為だけではなく，新しいテーマを見つける行為も含めて超絶技巧プログラミングである．

謝辞　いくつかのプログラムに対して重要な示唆を，さらに，本発表や本稿の構成・説明方法について数多くの実りある意見をくれた hirekoke 氏に深謝する．特に駅名を列挙する Quine というアイデアに対し，環状線である山手線を提案したのは彼女である．

また，本稿の初期版について有用なコメントをくれた浜地慎一郎氏，西尾泰和氏，竹迫良範氏に感謝する．

参考文献

[1] *Acme-EyeDrops*, CPAN. `http://search.cpan.org/dist/Acme-EyeDrops/lib/Acme/EyeDrops.pm`
[2] *Anarchy Golf*. `http://golf.shinh.org/`
[3] *Ascii shellcode*. `http://www.blackhatlibrary.net/Ascii_shellcode`
[4] *Brainfuck*, Wikipedia. `http://en.wikipedia.org/wiki/Brainfuck`
[5] *Code golf*. `http://codegolf.com/`
[6] *Don't repeat yourself*, Wikipedia. `http://en.wikipedia.org/wiki/Don%27t_repeat_yourself`
[7] 遠藤侑介: *1234567890_*, rubygems (2010). `http://rubygems.org/gems/1234567890_`
[8] 遠藤侑介: _, rubygems (2009). `http://rubygems.org/gems/_`
[9] *Esoteric programming language*, Wikipedia. `http://en.wikipedia.org/wiki/Esoteric_programming_language`
[10] はせがわようすけ: *jjencode*. `http://utf-8.jp/public/jjencode.html`
[11] はせがわようすけ: *aaencode*. `http://utf-8.jp/public/aaencode.html`

[12] 浜地慎一郎: The winning entry for the *Best solved puzzle* award, in the 20th International Obfuscated C Code Contest (2011).
http://www.ioccc.org/2011/hamaji/hamaji.c
[13] 浜地慎一郎: *downcase_quine.rb* (2012).
http://d.hatena.ne.jp/shinichiro_h/20120827
[14] 浜地慎一郎: *Hello broken keyboard* (2012).
http://d.hatena.ne.jp/shinichiro_h/20120915
[15] 浜地慎一郎: 任意の *Ruby* プログラムをアルファベットと数字のみにするプログラム (2008.11.09).
http://d.hatena.ne.jp/shinichiro_h/20081109
[16] 浜地慎一郎: *Quine* いろいろ (2008.11.02).
http://d.hatena.ne.jp/shinichiro_h/20081102
[17] Hou Qiming: The winning entry for the *Best self documenting program* award, in the 20th International Obfuscated C Code Contest (2011).
http://www.ioccc.org/2011/hou/hou.c
[18] *The International Obfuscated C Code Contest*.
http://www.ioccc.org/
[19] kikx: *Quine.grass* ネタバレ (2008).
http://d.hatena.ne.jp/kikx/20080914
[20] kurimura: 2008-08-24 のエントリ, kurimura の日記 (2008).
http://d.hatena.ne.jp/kurimura/20080824
[21] Merlyn LeRoy: The winning entry for the *Best layout* award, in 5th International Obfuscated C Code Contest (1988).
http://www.ioccc.org/1988/westley.c
[22] Franz List: *Transcendental Études* (1826, 1837, 1852).
[23] 西尾泰和: 「*Python* で記号なしプログラミング」(未完), 西尾泰和のはてなダイアリー (2012).
http://d.hatena.ne.jp/nishiohirokazu/20120906/1346938523
[24] 西尾泰和: *Python* でワンライナーを作成する際のノウハウ集, 西尾泰和のブログ (2006).
http://www.nishiohirokazu.org/blog/2006/08/python_12.html
[25] 西尾泰和: ワンライナーはダークサイド。改行をいれるべし, 西尾泰和のブログ (2007).
http://www.nishiohirokazu.org/blog/2007/08/post_323.html
[26] 西尾泰和: *Qlobe.py*, 西尾泰和のはてなダイアリー (2012).
http://d.hatena.ne.jp/nishiohirokazu/20120904/1346685131
[27] *Quine (computing)*, Wikipedia.
http://en.wikipedia.org/wiki/Quine_(computing)
[28] *Ruby programming language*.
http://www.ruby-lang.org/
[29] 竹迫良範: *ppencode*, Lightweight Language Day and Night (2005).
[30] *Unlambda*, Wikipedia.
http://en.wikipedia.org/wiki/Unlambda
[31] *Whitespace (programming language)*, Wikipedia.
http://en.wikipedia.org/wiki/Whitespace_(programming_language)
[32] Don Yang: The winning entry for the *Best Layout* award, in the 15th International Obfuscated C Code Contest (2000).
http://www.ioccc.org/2000/dhyang.c

付　　録

A.1 質疑応答

Q.（質問者不記録）小文字アルファベットのみで Hello, world!を出力するプログラム（図 A.3）の最後に書かれていた `o h` の意味は何か.

A. `yusuke end o h` という署名の一部で，コードとしての意味はほぼない．Ruby ではメソッドの探索は実行時に行われるので，存在しないメソッドの呼び出しを書いても，それが実際に実行されるまでコンパイルエラーのような例外は起きない．よって，その箇所に実行が到達するまでに `exit` すれば問題ない．ただし，`end` はブロックの終端を表すキーワードを兼ねている.

Q.（早稲田大数学科・石井さん）`Object#each` をオーバーライドしてしまうと，実行したいプログラムで `each` ができなくなるのではないか.

A. 元々 `Object#each` というメソッドは無く，これを定義しても子クラスの `each` メソッド（例えば `Array#each`）には影響を与えないので，問題が起きることは稀であろう．仮にメソッドを再定義する場合でも，`alias` を用いて元のメソッドに別名を付けておき，プログラム文字列の構築を終え次第，元の名前に再度 `alias` をすればよい．`remove_method` を用いて別名を消すことをもできる.

ただし，普通でないプログラム（例えば `caller` 関数を用いて，`eval` によって実行されているか，普通にトップレベルで実行されているかを判断し，この情報に依存して挙動を変えるようなプログラム）をエンコードする場合には，元のプログラム側で修正を行う必要があるだろう.

Q.（東大・大島さん）このように制限のない言語では，逆にバグの温床になるようなことはないのか.

A. ある．例えば，存在しないメソッドの呼び出しは typo である可能性が高いが，実行するまでそれに気づけないことはしばしば問題として挙げられる．これは `eval` やリフレクションによる動的なメソッド定義を許した副作用である．動的なメソッド定義には（例えば setter のような）単純なメソッド定義を繰り返さずに済む（Don't repeat yourself 原則 [6]）という利点があり，一長一短といえる.

また，既存クラスに後からメソッド定義を追加・再定義できる open class は非常に強力な機能であり，使い方を誤れば極めて危険である．しかし他人の作ったライブラリの挙動を，プログラム自体を変更せずに修正できる（この手法を monkey-patching と呼ぶ）というメリットがある.

言語の安全性と柔軟性はトレードオフであるが，Ruby はこのようなトレードオフにおいて，柔軟性を優先する傾向がある（このことは動的型付けであることからも伺える）.

Q.（質問者不記録）過去に見た一番ひどいプログラムは何か.

A. 超絶技巧プログラミングの成果を比較するのは極めて難しく，1 つを挙げることは避けたい．

IOCCC のエントリはいずれも「ひどい」プログラムばかりで目を見張るものがある．その中でも，我々が超絶技巧プログラミングを行うようになった大きなきっかけは「あくそくざん」[32] である．

ひどい「プログラム」そのものではなく，制約の下でのプログラムの「書き方」に感動した例として，kikx によるプログラミング言語 grass における Quine の書き方 [19] を挙げる．grass は de Bruijn index で書かれたラムダ式を w，W，v の 3 種の文字を使ってエンコードする言語であり，単純に Quine を書くと非常に長大になるが，kikx は極めて短いエンコード方法を提案し，1.5 kB 程度の Quine を実際に作成した．詳細は [19] を参照せよ．

A.2 最悪な Hello, world!の動作

まず，図 1 のプログラムを通常のインデントに直したものを図 A·1 に示す．

1 行目は Object#|メソッドを Object#send メソッドの別名として定義する．Ruby では|演算子はオブジェクトのメソッドであり，A | B は A.|(B) というメソッド呼び出しの糖衣構文として扱われる．Object#send はオブジェクトのメソッドを呼び出すメソッドで，A.send('foo', 'bar') は A.foo('bar') のような意味である（正確には，send メソッドでは private メソッドであっても呼び出せるという違いがある）．以上により，A | "foo"は A.foo を実行したのとほぼ同じ効果を持つ．

2 行目は GC クラスオブジェクトに対してメソッドを呼び出すが，その引数は"%p?" %（次以降の行の返り値）となっている．3 行目から 9 行目はメソッド定義のブロックになっており，メソッド定義の構文は nil 値を返す．"%p?" % nil は"nil?"に評価される．メソッド Object#nil?は，レシーバが nil の時に true を，そうでないときに false を返す．GC クラスオブジェクトは nil ではないため，結果として 2 行目はグローバル変数$stdin に false を代入する．

3 行目から 9 行目のメソッド定義を詳しく見ていく．このメソッド定義では，定数値 FALSE に gets メソッドを定義している．Ruby ではこのように既存のオブジェクトにメソッド定義を後から追加定義できる（たとえ組み込みクラスのオブジェクトであっても）．大文字の FALSE はあまり Ruby では用いられないが，false が代入された組み込みの定数である（初期の Ruby では FALSE のみがあった）．4 行目から 8 行目まではすべて文字列を連結している．4 行目は ASCII コードで 8|64 = 72 の文字からなる文字列，すなわち"H"となる．5 行目は 232324 を 25 進数表記した文字列，すなわち"ello"となる．6 行目はそのまま"，"である．7 行目はまず，%w|w !|で"w"と"!"の 2 要素からなる配列を構築する．次に"orlc".next で"orld"を得る．配列に対する乗算，つまり Array#*メソッドは，引数の文字列を用いて配列を連結する．よって%w|w !| * "orlc".next は"world"となる．これに 8 進数で 012 の文字，すなわち改行を追加する．8 行目の (c).Yusuke は，7 行目の終わりのショートカット演算子により評価が省略される．以上により，このメソッドは"Hello, world!\n"という文字列を返すことがわかる．

10 行目では，"stegano-X."[0,4] で，先頭の 4 文字からなる文字列"steg"を得る．これに対して String#reverse を適用し，"gets"とする．すなわち 10 行目全体は"oh, 2009".gets を呼び出すこととなる．しかし String クラスに gets という名前のメソッドは定義されていないため，Object#gets が呼び出される．このメソッドは private メソッドであるが，Object#send は前述の通り private メソッドであっても呼び出せる．Object#gets の仕様は複雑なので詳しくは省略するが，ここでの挙動は，$stdin.gets を呼び出し，その結果をグローバル変数$_に代入する．

11 行目は変数 d と be にそれぞれ文字列"whydoes"と"crypto"を代入する．

12 行目ではまず，%記法に注目する必要がある．%記法は%に続く文字が開き括弧の場合，対応する閉じ括弧までの文字列となるが，それ以外の記号の場合，同じ記号までの文字列となる．つまり%.mains. で"mains"という文字列を表す．この文字列に対し String#tr メソッドを呼び出す．このメソッドは 2 つの文字列を引数として受け取り，第一引数の文字列に含まれる文字を第二引数の文字列の文字に置き換える．第一引数は"eams"である．第二引数は be.delete(d) で，"crypto"から"whydoes"に含まれる文字を消した文字列，すなわち"crpt"となる．よって，"mains"の"m"を"p"に，"a"を"r"に，"s"を"t"に置き換え，結果的に"print"が得られる．よって 12 行目は全体として:make | "print"を評価することになる．これは 10 行目と同様，Object#print を無引数で呼び出す効果を持つ．Object#print は，無引数で呼び出された場合，グローバル変数$_の中身を標準出力に出力する．

以上により，"Hello, world!\n"が出力される．

A.3 小文字アルファベットのみで書いた Hello, world!

図 A·2 は，5 節で示した方法で任意のプログラムを実際にエンコードするプログラムである．

図 A·3 は小文字アルファベットのみで書かれた Hello, world! プログラムを示している．このプログラムは図 A·2 のエンコードプログラムの出力そのものではなく，なるべく行数が少なくなるよう，また，一行中の単語がなるべく等間隔で配置されるようレイアウトを調整している．

```
 1: alias|send
 2: $stdin=GC | "%p?" %
 3:   def FALSE.gets()
 4:     (8|64).chr +
 5:       232424.to_s(25) +
 6:       ", " +
 7:       %w|w !| * "orlc".next << 012 ||
 8:         (c).Yusuke
 9:   end
10: "oh, 2009" | "stegano-X."[0,4].reverse
11: d,be="whydoes","crypto";
12: :make.| %.mains..tr'eams',be.delete(d)
```

図 **A·1** 最悪な Hello, world!（インデント修正）

Fig. A·1 Hello, world! from hell (with normal indent)

```
code = "puts'Hello, world!'"

puts "public"
puts "def each"
puts "  clear"
code.each_byte.to_a.reverse_each do |x|
  puts "  concat begin"
  puts "    dup"
  puts "  ensure"
  puts "    clear"
  ("%b" % x)[0..-2].scan(/10*/) do |b|
    puts "    #{ "concat " * (b.size + 1) }size"
  end
  puts "    concat size" if x % 2 == 1
  puts "    concat begin"
  puts "      size"
  puts "    ensure"
  puts "      clear"
  puts "    end"
  puts "  end"
end
puts "  eval self"
puts "  exit"
puts "end"
puts "for x in inspect do"
puts "end"
```

図 **A·2** 小文字アルファベットのみで任意のプログラムをエンコードするプログラム

Fig. A·2 A program that encodes an arbitrary program with only lower-case alphabets

```
loop       do      break       public      end
catch      inspect       do      def      each
loop       do       break      clear       end
concat     begin      dup     ensure     clear
concat     concat    concat     concat    size
loop   do  break   concat   concat  size   end
loop   do  break   concat   concat  size   end
loop    do     break    concat     size    end
concat  begin   size  ensure  clear   end  end
concat     begin      dup     ensure     clear
concat concat concat concat concat concat size
loop    do     break    concat     size    end
concat  begin   size  ensure  clear   end  end
concat     begin      dup     ensure     clear
loop   do  break   concat   concat  size   end
concat     concat    concat     concat    size
concat        concat        concat        size
concat  begin   size  ensure  clear   end  end
concat     begin      dup     ensure     clear
loop   do  break   concat   concat  size   end
concat        concat        concat        size
loop   do  break   concat   concat  size   end
concat        concat        concat        size
concat  begin   size  ensure  clear   end  end
concat     begin      dup     ensure     clear
loop   do  break   concat   concat  size   end
loop   do  break   concat   concat  size   end
concat     concat    concat     concat    size
loop   do  break   concat   concat  size   end
concat  begin   size  ensure  clear   end  end
concat     begin      dup     ensure     clear
loop   do  break   concat   concat  size   end
concat        concat        concat        size
loop   do  break   concat   concat  size   end
loop   do  break   concat   concat  size   end
loop   do  break   concat   concat  size   end
loop    do     break    concat     size    end
concat  begin   size  ensure  clear   end  end
concat     begin      dup     ensure     clear
loop   do  break   concat   concat  size   end
loop   do  break   concat   concat  size   end
concat        concat        concat        size
loop   do  break   concat   concat  size   end
loop   do  break   concat   concat  size   end
concat        concat        concat        size
loop   do  break   concat   concat  size   end
concat  begin   size  ensure  clear   end  end
concat     begin      dup     ensure     clear
concat concat concat concat concat concat size
concat  begin   size  ensure  clear   end  end
concat     begin      dup     ensure     clear
concat        concat        concat        size
loop   do  break   concat   concat  size   end
concat        concat        concat        size
concat  begin   size  ensure  clear   end  end
concat     begin      dup     ensure     clear
loop   do  break   concat   concat  size   end
concat        concat        concat        size
loop   do  break   concat   concat  size   end
loop   do  break   concat   concat  size   end
loop   do  break   concat   concat  size   end
loop    do     break    concat     size    end
concat  begin   size  ensure  clear   end  end
```

```
concat     begin      dup     ensure     clear
loop   do  break   concat   concat  size   end
concat        concat        concat        size
loop   do  break   concat   concat  size   end
concat        concat        concat        size
concat  begin   size  ensure  clear   end  end
concat     begin      dup     ensure     clear
loop   do  break   concat   concat  size   end
concat        concat        concat        size
loop   do  break   concat   concat  size   end
concat        concat        concat        size
concat  begin   size  ensure  clear   end  end
concat     begin      dup     ensure     clear
loop   do  break   concat   concat  size   end
concat     concat    concat     concat    size
concat        concat        concat        size
loop    do     break    concat     size    end
concat  begin   size  ensure  clear   end  end
concat     begin      dup     ensure     clear
concat     concat    concat     concat    size
concat     concat    concat     concat    size
concat  begin   size  ensure  clear   end  end
concat     begin      dup     ensure     clear
concat     concat    concat     concat    size
loop   do  break   concat   concat  size   end
loop   do  break   concat   concat  size   end
loop    do     break    concat     size    end
concat  begin   size  ensure  clear   end  end
concat     begin      dup     ensure     clear
loop   do  break   concat   concat  size   end
loop   do  break   concat   concat  size   end
concat     concat    concat     concat    size
loop   do  break   concat   concat  size   end
loop    do     break    concat     size    end
concat  begin   size  ensure  clear   end  end
concat     begin      dup     ensure     clear
loop   do  break   concat   concat  size   end
loop   do  break   concat   concat  size   end
concat        concat        concat        size
concat        concat        concat        size
concat  begin   size  ensure  clear   end  end
concat     begin      dup     ensure     clear
loop   do  break   concat   concat  size   end
loop   do  break   concat   concat  size   end
concat        concat        concat        size
concat        concat        concat        size
loop    do     break    concat     size    end
concat  begin   size  ensure  clear   end  end
concat     begin      dup     ensure     clear
loop   do  break   concat   concat  size   end
loop   do  break   concat   concat  size   end
concat  concat   concat  concat   concat  size
concat  begin   size  ensure  clear   end  end
loop     do     break    eval     self     end
loop     do      exit     end      end      end
for       each       in       inspect       do




                              copyright mmxii
                                 yusuke end
                                           o
                                            h
```

図 **A·3** 小文字アルファベットのみで書かれた Hello, world!プログラム

Fig. A·3 Hello, world! program by using only lower-case alphabets