HaoriBricks: ブロック玩具に学ぶ日本語文章生成ライブラリ

佐藤理史
名古屋大学大学院工学研究科

1 はじめに

我々は、2013年にコンピュータを用いて小説を自動生成する研究を開始し、2015年には『第３回日経星新一賞』に二つの作品を応募するまでに至った[1]。応募した作品は、選考の初期の段階で撥ねられたようであるが、ウェブを通じて公開したところ[1]、好意的な反応が得られた。我々のグループは、引き続き『第４回日経星新一賞』にも、作品を応募した[2]。

星新一賞は、一万字以下のショートショートを対象とした文学賞である。我々が制作した作品は三千字程度であり、プロの作家から、その短さでは賞を取るのは難しいとの指摘を受けている。賞を目指すため、より長い小説を作る必要があるが、その実現には多くの課題が存在する。

小説の生成は、概念的には、プロット生成と文章生成に大きく分けることができる。文章生成における課題は、「いかにして長い文章を簡便に作るか」である。既存の文章を再利用（切り貼り）する方法は、現在の技術では自然な文章を作ることが難しく、かつ、剽窃の問題をクリアすることが難しい。そのため、文章を構成する部品はすべて、制作者の手で入力する必要がある。それらの部品は、そのままの形で使用するのであればプレーンテキスト（文字列データ）で構わないが、文章を生成する過程でなんらかの加工が必要となる場合は、適切なアノテーション（情報付加）が不可欠である。

我々のグループがこれまで制作した作品で用いた部品の多くはプレーンテキストで記述されており、統語的変形等の操作が必要なもののみ、Haori [3]が解釈できる形式で記述されている。これは、部品をプレーンテキストで入力する場合と比べて、Haori形式で入力するコストが非常に大きい（数十倍）であることに起因する。同時に、Haoriは、文の統語構造から表層文を生成するツールとして設計されたため、「文を部品から組み立てる」機能に乏しい。このため、作成した部品の再利用が容易ではない。

[1] http://kotoba.nuee.nagoya-u.ac.jp/sc/gw/

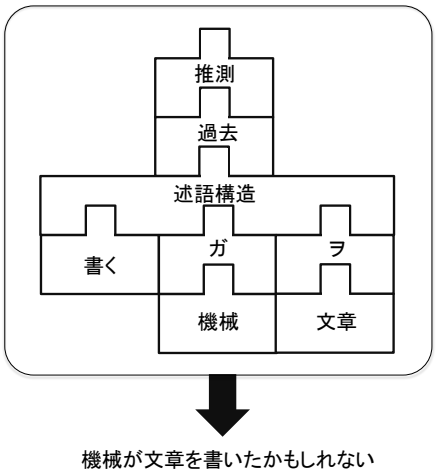

図 1: HaoriBricksのコンセプト図

このような背景より、日本語文章の自動生成を支援する新たなツールとしてHaoriBricksを設計し、実装を進めている。HaoriBricksの基本アイディアは、ブロック玩具から借用している（図1参照）。すなわち、文章を、部品（bricks）を組み合わて作成するというアイディアである。

2 実行例

HaoriBricksは、Rubyのライブラリ（モジュール）として実装されている。HaoriBricksを用いた日本語テキストの生成例を図2に示す。

1. HaoriBricksの基本データ構造は、各種のBrickクラスのインスタンスで、ライブラリには、Brick（インスタンス）を作り出したり組み合わせたりする、さまざまなメソッドが定義されている。
2. 「ヲ(·)」は、格助詞「を」に対応するBrickを作成し、そのBrickに引数のBrickを接続する。引数が文字列の場合は、辞書を引いてBrickに変換することも行なう。すなわち、「ヲ(’文章’)」は、格助詞「を」に対応するBrickに、「文章」という語に対応するBrickを接続した構造（マクロBrick）を作成する。

```
x = 述語構造('使う', ヲ('HaoriBricks'))
=> #<Haori::Brick::Tree:0x007fa8219d9290>

haori(x)
=> "HaoriBricksを使う"

def 自動生成する (*children)
  述語構造(スル(複合('自動', '生成')), *children)
end
=> nil

y = 自動生成する(読点(テ形(x)), ヲ('文章'))
=> #<Haori::Brick::Tree:0x007fa822086ed0>

haori(y)
=> "HaoriBricksを使って、文章を自動生成する"

haori(意志形(連用修飾要素追加(y, ニ('大量'))))
=> "HaoriBricksを使って、文章を大量に自動生成しよう"
```

図 2: HaoriBricksを用いたテキストの生成例

3. 「述語構造(·)」は、述語構造を表すBrickを作成し、引数のBricksを接続する。第一引数が述語、残りの引数をその支配下の構造である。
4. 「haori(·)」は、Brickから表層文字列を生成するためのメソッドである。
5. マクロBrickは、(単なる) Rubyのメソッドである。すなわち、ユーザーが新たなマクロBrickを定義することができる。たとえば、「自動生成する(·)」というマクロBrickは、述語を「自動生成する」に固定した述語構造を作り出すことができる。新たなマクロBrickを定義すれば、ライブラリが提供する他のBricksと全く同じように利用できるようになる。
6. 「複合(·)」は、複合語を作り出すBrickを作成する。
7. 「スル(·)」は、派生語尾スルに対応するBrickを作成し、引数のBrickをこれに接続する。
8. 「テ形(·)」は、活用形をタ系連用テ形に変換するBrickを作成し、引数のBrickをこれに接続する。
9. 「読点(·)」は、読点を付与するBrickを作成し、引数のBrickをこれに接続する。
10. 「連用修飾要素追加(·)」は、第一引数の述語構造に、第二引数の連用修飾要素を追加するBrickを作成する。
11. 「意志形(·)」は、活用形を意志形に変換するBrickを作成し、引数のBrickをこれに接続する。

このように、HaoriBricksは、小さな部品を組み合わせて、より大きな部品を定義したり、それらを組み合わせて文（や文章）を作成することができる。

3 基本設計

HaoriBricksは、次のように設計されている。

1. Brickは、一つの突起といくつかのスロット（突起を嵌める部分）を持つ。
2. 内容語を表すBrickは、スロットを持たない。
3. 機能語を表すBrickは、スロットを一つ持つ。
4. 述語構造を表すBrickは、任意個のスロットを持つ。
5. あるBrickの突起を、別のBrickのスロットに嵌めることにより、二つのBricks を組み合わせる（接続する）ことができる。
6. Brickの背後には、統語構造が隠蔽されている。
7. 統語構造を持つBrick以外に、統語構造を持たないBrickも存在する。このようなBrickは、支配下のBricksの統語構造を操作するためのBrickである。たとえば、活用形を変更するBrick、連用修飾要素を追加するBrickなどが、これに相当する。
8. マクロBrickは、Bricksの組み合わせ情報のみを保持する。つまり、Bricksを接続した時点では、背後にある統語構造を操作しない。
9. Brickの表層文字列化は、(1) スロットに接続されている配下のBricksを含めた全体を統語構造に変換した後、(2)その統語構造を表層文字列に変換する。この統語構造は、Haori [3]が採用していた構造（Lexal、Bal、JBT）[2]を若干修正したものである。
10. つまり、HaoriBricksは、Haoriの統語構造を簡便に組み立てるためのBrickラッパである。

4 主要なBrickクラス

主要なBrickクラスには、次のものがある。

1. `Word`
   内容語に対応するBrick。背後に語彙情報を持つ。語彙情報（品詞、活用型、表記形など）は辞書からの供給される。
2. `Particle`
   助詞に対応するBrick。背後に語彙情報を持つ。スロットを一つ持つ。
3. `Tree`
   木構造を作るためのBrick。任意個のスロットを持ち、第一スロットを木のルートノード、以降のスロットを子木とする構造を作り出す。
4. `Boundary`
   句読点などを挿入するためのBrick。HaoriBricksが仮定する文法では、語と語（より正確に

[2]Lexalは語彙、Balは文節、JBTは文節依存木を表すデータ構造である。

は隣り合う言語単位）の間に、「境界」という実体が存在すると考える。たとえば、「これは例です」では、以下に示す「-」のところに、見えない境界が存在すると考える。

- これ - は - 例 - です -

一方、句読点や括弧などは、「見える境界」と考える。見える境界を「-[、]-」のように書くとすると、「これは、例です。」を次のように解釈する。

- これ - は -[、]- 例 - です -[。]-

つまり、HaoriBricksでは、テキストを、境界と語彙（言語単位）が交互に並んだ列とみなす。

5. `Param`
   統語構造に対するパラメータを指定するためのBrick。典型的なパラメータには、活用形、表記形（ひらがな表記、漢字表記）などがある。
6. `Trans`
   統語構造に対する変形操作を指定するためのBrick。現時点では、述語構造に対する補足要素追加と連用修飾要素追加がある。

5 表層文字列化

すでに3節で述べたように、Brickの表層文字列化は、次の２ステップからなる。

1. 配下のBricksを含めて、統語構造に変換する。
2. 得られた統語構造を表層文字列に変換する。

5.1 統語構造への変換

それぞれのBrickは、背後の統語構造をどのように組み立てるかを記述したプログラムを保持している。Brickの統語構造への変換は、配下のBricksを統語構造に変換したのち、このプログラムを起動して、Brick全体に対する統語構造を生成する。これらのプログラムは、以下のようなに種類に大別できる。

1. Lexalを作成する（`Word`）
2. Lexalを作成し、それに配下構造を接続した構造を作る（`Particle`など）
3. JBTを作成する（`Tree`）
4. 配下構造へパラメータの値を反映させる（`Param`）
5. 配下構造を変形する（`Trans`）

これらのうち、最初の三種類は、単に（ボトムアップに）統語構造を組み立てるものである。残りの二種類は、いわゆる統語的変形をサポートするために、配下の統語構造になんらかの操作を適用するものである。

文生成ツールにおいて、このような統語的変形をサポートすることは、入力の省力化という観点において非常に重要である。たとえば、図2では、「自動生成する」というマクロBrick（メソッド）を定義したが、もし、活用形が外から操作できないのであれば、すべての活用形に対してメソッドを定義するか、あるいは、活用形を引数に取れるようなメソッドを定義する必要が生じる。これに対して、HaoriBricksは、活用形を外から操作する機能、連用修飾要素を外から追加する機能を提供するので、図2に示すように「自動生成する」というマクロBrick（メソッド）を一つ定義すれば十分である。

図3に、Bricksから生成される統語構造の一例を示す。なお、この構造からは、「『文章』を生成する。」というテキストが生成される。

5.2 統語構造からの表層文字列生成

統語構造からの表層文字列生成は、次の２ステップで実行される。

1. 語彙と境界の線形化
   統語構造から、境界と語彙が交互に並ぶ列（Boundary-Unit Sequence, Bus）を作成する。統語構造（Lexal, Bal, JBT）は、いずれも`:left_boundary`と`:right_boundary`の二つのパラメータをとることができ、これらが指定されている場合は、語彙間に設定される境界に、その情報を引き継ぐ。
2. 文字列生成と連結
   Busのそれぞれの要素を文字列化して、それらを連結する。なお、直前の要素に特定の活用形を要求する語彙（たとえば、派生語尾「れる」は未然形を要求する）が存在する場合は、この文字列生成の時点で要求情報が伝達され、反映される。

6 基本セット

HaoriBricksでは、ブロック玩具にならってBrick-Setという形で提供することを予定しており、現在、基本セット(`Haori::Set::Basic`)の実装を進めている。基本セットには、おおよそ、次のようなBricksを含めることを予定している。

1. 文章構造を表すBricks
   文章、段落
2. 文の構造を表すBricks
   文、述語構造、修飾構造、並列構造*
3. 節に関連するBricks
   主要な節形式*

```
#<Haori::SS::JBT:0x007fda9295d250
 @type=:predicate, @param={:right_boundary=>"。"}, @force={},
 @root= #<Haori::SS::Bal:0x007fda9295d980
  @param={}, @force={},
  @mp= #<Haori::SS::MP:0x007fda9295d9f8
   @param={}, @force={},
   @list= [#<Haori::SS::Lexal:0x007fda9295da70
     @param={}, @force={},
     @lemma= #<Haori::KB::Lexicon::Entry:0x007fda93253828
       @hash={:id=>"生成", :u=>"語", :lex=>"生成", :lc=>"名詞"}>>,
    #<Haori::SS::Lexal:0x007fda9295d840
     @param={}, @force={},
     @lemma= #<Haori::KB::Lexicon::Entry:0x007fda93253a80
       @hash= {:u=>"派生語尾", :lc=>"動詞", :ctype=>"サ変動詞型", :lex=>"する", :id=>"スル"}>>]>,
  @fp=#<Haori::SS::FP:0x007fda9295d8b8 @force={}, @list=[], @param={}>>
 @children= [#<Haori::SS::JBT:0x007fda9295d390
   @type=:tree, @param={}, @force={},
   @root= #<Haori::SS::Bal:0x007fda9295d570
    @param={}, @force={},
    @mp= #<Haori::SS::MP:0x007fda9295d5e8
     @param={}, @force={},
     @list= [#<Haori::SS::Lexal:0x007fda9295d700
       @param={:left_boundary=>"『", :right_boundary=>"』"}, @force={},
       @lemma= #<Haori::KB::Lexicon::Entry:0x007fda932534e0
         @hash={:id=>"文章", :u=>"語", :lex=>"文章", :lc=>"名詞"}>>]>,
    @fp= #<Haori::SS::FP:0x007fda9295d480
     @param={}, @force={},
     @list= [#<Haori::SS::Lexal:0x007fda9295d750
       @param={}, @force={},
       @lemma= #<Haori::KB::Lexicon::Entry:0x007fda93252ce8
         @hash={:lc=>"格助詞", :pc=>"50", :lex=>"を", :id=>"格ヲ", :u=>"語"}>>]>>,
   @children=[]>]>
```

図 3: 「句点(述語構造(スル('生成'), ヲ(二重括弧('文章'))))」から生成される統語構造

4. 文節に関連するBricks
   主要な助詞（格助詞、副助詞、接続助詞、終助詞）
5. 述語に関連するBricks
   複合述語（複合動詞の類）、助動詞、判定詞
6. 語構成に関連するBricks
   派生、接辞、複合
7. 記号に関するBricks
   句読点、括弧の類
8. パラメータを変更するBricks
   活用形、接続型、表記形
9. 統語的変形を行なうBricks
   補足要素追加、連用修飾要素追加、副助詞追加*、提題化*、従属節追加*

これらのリストのうち、*をつけたものの仕様が、現時点では固まっていない。副助詞の追加は、格助詞の前後のどちらにも挿入できる場合（「〜だけに」、「〜にだけ」）があり、かつ、格助詞の消失を伴う場合もある。これらを明示的に指定する形式とデフォールトの設計が悩ましい。残りの四つ（並列構造、節形式、提題化、従属節追加）は、それらの統語構造をどのように表現すべきかが自明ではない。

基本セットでは、語彙に対応したBricksを提供することを基本とし、抽象的なBricksは導入しない予定である。たとえば、図1にある「過去」や「推測」といったBricksは導入せず、当面は、「活用形(:タ形)」や「かもしれない」といったより直接的なBricksで代用する。

謝辞 本研究は、JSPS 科学研究費基盤研究 (B) 「文章の読解と産出のための言語処理技術」(課題番号 15H02748) の助成を受けている。

参考文献

[1] 佐藤理史. コンピュータが小説を書く日—AI作家に「賞」は取れるか. 日本経済新聞出版社, 2016.

[2] 松山諒平, 佐藤理史, 松崎拓也. 人狼ログからの小説の自動生成. 言語処理学会第23回年次大会発表論文集, 2017.

[3] 佐藤理史. 「文生成器を作る」とはどういうことか. 言語処理学会第21回年次大会発表論文集, pp. 1080–1083, 2015.