「文生成器を作る」とはどういうことか

佐藤理史
名古屋大学大学院工学研究科電子情報システム専攻

1 はじめに

この短い論考では、「日本語の文生成器を作るということは、どういうことなのか」について、いくつかの視点から考察する。私の知る限り、日本語の文生成器として広く使われているツールは存在しない。日本語の文生成器の存在自体、ほとんど知られていない。

我々のグループは、現在、ショートショートを自動生成するシステムの作成に取り組んでいる [1]。そのようなシステムの実現のためには、多様な文の生成を可能とする文生成器が必要であることは疑う余地がなく、これを実現することが喫緊の課題となっている。このことが、本論考の背景となっている。

2 文生成は文解析の逆変換ではない

まず、文生成を次のようにみなすことから出発する。

> 文生成とは、何らかの表現から、表層文 (文字列) を作り出す操作 (処理) を指す。

最初に確認すべきことは、次のことである。

> 文生成は、文解析の逆変換ではない。

表層文から文解析器によって作り出される解析結果 (たとえば、文節依存構造) は、通常、表層文 (文字列) を生成するのに十分な情報を完全に (明示的に) 有している。図 1 に、(a) 表層文と、(b) それを Juman+KNP によって解析した結果 (簡略化したもの) を示す。この図の (b) から (a) を作る操作は、私が考える文生成ではない。

文解析は、(a) から (b) を作り出す操作である。その操作において、外部知識が参照され、そこから情報が取り込まれる。(b) は (a) よりも豊かな情報を有し、(a) を作り出すのに十分な情報を有している。すなわち、文解析の逆変換である (b) から (a) を作る操作は、容易かつ自明である。

では、何をもって文生成とみなせばよいのか。そのヒントは、いくつかのところにある。

```
(a) 文解析器を作る

(b) + 1D <係:文節内><体言>
    文 ぶん 文 名詞 普通名詞 * *
    + 2D <係:文節内><体言>
    生成 せいせい 生成 名詞 サ変名詞 * *
    + 3D <体言><係:ヲ格>
    器 うつわ 器 名詞 普通名詞 * *
    を を を 助詞 9 格助詞 * *
    + -1D <用言:動><レベル:C>
    作る つくる 作る 動詞 * 子音動詞ラ行 基本形
```

図 1: (a) 表層文と (b) 文解析結果

第一のヒントは、文解析器にある。文解析器 (形態素解析器と構文解析器) が広く用いられる理由は、文字列としての文に、より豊かな情報を付加してくれるからである。解析器は、いわば、外部知識から情報を注入する役割を果たしている。たとえば、図 1 の (a) には、「文」が名詞で「作る」が動詞であるという情報は存在しない。解析結果の (b) には存在する。解析器は、入力を、より情報量の多い出力に変換するからこそ有用であり、使われるのである。

これと同じことが生成器に対しても成り立つだろう。すなわち、生成器も、入力を、より情報量の多い出力に変換する操作となっていなければならない。そうでなければ、そんな生成器は不要ということになるだろう。

もう一つのヒントは、英語の生成器にある。テキスト生成の標準的な構成は、Document Planning、Microplanning、Surface Realization のパイプライン構成である [2]。これらのモジュールのうち、Surface Realization が狭義の文生成に対応し、よく知られた英語用ツールに RealPro がある。このツールは、Deep Syntactic Structure (深い構文構造) と呼ばれる構造を、英語の表層文 (文字列) に変換する。

ここで、その入出力関係だけを見ると何も見えてこない。しかし、英語の表層文の一歩手前では、すべての語の表層文字列と語順が定まった状態が存在するはずである。それは、いわば、図 1 の (b) に相当するものである。つまり、文生成の主要部は、深い構文構造から (b) に相当する構造 (これを浅い構文構造と呼ぼう) に変換する処理である。

Copyright(C) 2015 The Association for Natural Language Processing.
All Rights Reserved.

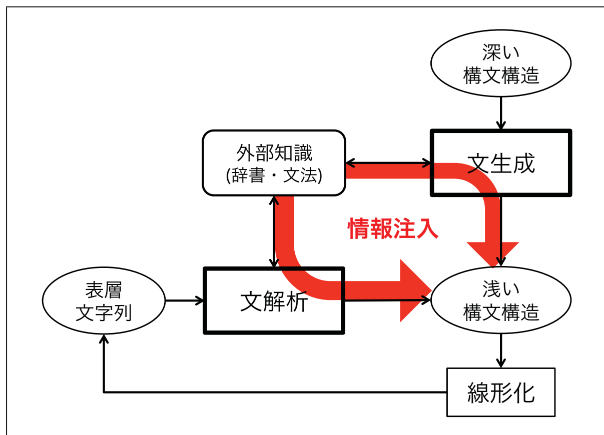

図 2: 文解析と文生成

このように考えると、文解析と文生成を図 2 のような図式として整理することができる。どちらも、外部知識から情報を注入する処理であり、その出力は、事実上、浅い構文構造である。文解析の出力が浅い構文構造 (つまり、表層文字列を復元できる構造) である限り、文生成は文解析の逆変換にはなり得ない。

3　基本的な道具立て

次に、文生成器を構成する基本的な道具立てについて考えよう。

いま、「食べる」という文字列を生成することを考える。この文字列だけを生成するのであれば、この文字列を直接指定するのが最も妥当な方法である。

しかし、ある場合には「食べる」を生成し、別の場合には「食べた」を生成したいのであれば、それをパラメータによって制御するのが常套手段である。たとえば、次のように。

gen(食べる, { 活用型:母音動詞, 活用形:辞書形 })
　⇒ 食べる
gen(食べる, { 活用型:母音動詞, 活用形:タ形 })
　⇒ 食べた

これは、活用型と活用形をパラメータ化し、辞書形の文字列から他の活用形の文字列を生成できるようにすることに他ならない。「活用形を抽象化した」という言い方もできる。当然ながら、入力には「食べる」のタ形がどのような文字列かという情報が存在しないので、「食べた」を作り出すためには、外部知識を参照する必要がある。

次に、あるパラメータの値を明示的に指定しなかった場合、デフォールト値を用いることを考えよう。たとえば、活用形のデフォールト値を「辞書形」と定める。すると、次のような生成が可能となる。

gen(食べる, { 活用型:母音動詞 })
　⇒ gen(食べる, { 活用型:母音動詞, 活用形:辞書形 })
　⇒ 食べる

デフォールトを適切に設定することにより、生成のために必要な入力の記述を削減することができる。

最後に、語に関する情報を、辞書という形であらかじめ記述しておくことを考える。この辞書 (外部知識) とうまく連携すれば、生成のために必要な記述を、表面上は削減することができる。たとえば、「食べる」という語に対して、「活用型:母音動詞」という情報が辞書から供給されれば、あとは、活用形だけ指定すればよい。つまり、次のような生成が可能となる。

gen(食べる, { 活用形:タ形 })
　⇒ gen(食べる, { 活用型:母音動詞, 活用形:タ形 })
　⇒ 食べた

上に示した、(i) パラメータ化、(ii) デフォールトの設定、(iii) 外部知識との連携、の 3 つを、文生成器を実現するための基本的な道具立てとして採用する。

4　パラメータの依存関係と階層

語には、いくつかの抽象化のレベルを設定することができる。ここでは、実際の文に現れる形を「表層形」と呼ぼう。表層形として、次のようなものが考えられる。

食べる、食べた、たべる、たべた

次に、活用形を抽象化 (パラメータ化) したものを「書字形」と呼ぼう。上記の表層形は、次のように整理される。

書字形	表層形
食べる	食べる (辞書形)、食べた (タ形)
たべる	たべる (辞書形)、たべた (タ形)

最後に、書字形の表記を抽象化したものを「語彙素」と呼ぼう。上記の書字形は、次のように整理される。

語彙素	書字形
食べる	食べる (標準表記)、たべる (かな表記)

上記の関係は、次のような式で表現できる。

$$\text{書字形} = f_1(\text{語彙素}, \text{表記}) \quad (1)$$

$$\text{表層形} = f_2(\text{書字形}, \text{活用形}) \quad (2)$$

前の式を後の式に代入すると、次の式が得られる。

$$\text{表層形} = f_2(f_1(\text{語彙素}, \text{表記}), \text{活用形}) \quad (3)$$

Copyright(C) 2015 The Association for Natural Language Processing.
All Rights Reserved.

この式は、「語彙素と表記と活用形から、表層形が計算できる」ことを意味する。

今回作成する文生成器では、文生成に必要な情報を、すべて、「属性名:属性値」として表現する。これが、パラメータと呼んでいるものの実体である。上記の関係は、パラメータ間に依存関係を定義することに他ならない。

パラメータ間の依存関係のうち、主要な関係を抜き出したものが、パラメータの階層である。先の例であれば、次のような階層が設定されていることになる。(右の方がより抽象度が高い)

$$表層形 \rightarrow 書字形 \rightarrow 語彙素$$

文生成のある種の側面は、「より抽象度の高い記述から、表層形を生成すること」にある。これを、表層実体化 (realization) と呼ぶ。表層実体化のためには、最終的に、それに必要なパラメータの値をすべて定める必要がある。文生成の本質は、「表層実体化に必要なパラメータの値のうち、入力として明示的に与えられていない値を、何らかの方法で決定すること」である。それらは、デフォールト値で決まることもあれば、文の他の要素からの要請からによって決まることもある。

5 文生成器を作るとはどういうことか

文生成器を作るということは、結局のところ、式 (3) のような関数を設計することである。これは、「文の表層形を、それを合成する関数とパラメータ (引数) に分解すること」に他ならない。

ここでの基本的な道具立ては、先に示したように、(i) パラメータ化、(ii) デフォールトの設定、(iii) 外部知識との連携、の 3 つである。これに加えて、パラメータを適切に階層化することがおそらく必要である。

6 日本語の文生成

それでは、実際の日本語文の生成器をどのように設計すればよいだろうか。

現在作成中の文生成器では、日本語文生成処理の統一的なデータ構造として、日本語文節木 (JBT) を採用する。この木構造は、文節をノードとする木構造で、親子関係は文節の依存関係を表すものとする。

JBT のノードである文節は、文節属性リスト (BAL) で表現する。次の例は、「太郎が重い荷物を軽々と運ぶ」を生成する JBT である。BAL は、Ruby の Hash として記述する。

```
[ { string: '運ぶ' },
  [ { string: '太郎が', order: 1 } ],
  [ { string: '荷物を', order: 2 },
    [ { string: '重い' } ] ],
  [ { string: '軽々と', order: 3 } ] ]
```

JTB に含まれるすべての BAL において、表層形 (string) と姉妹ノードにおける順番 (order) の 2 つが属性の値が定まれば、その JTB に対する表層文字列を生成することができる。なお、姉妹ノードが存在しないノードにおいては、順番の指定は不要である。

このような枠組みにおいて、文生成とは、「JTB に含まれる、それぞれの BAL の string と order の値を決定すること」である。

7 文節の段階的抽象化

幸いなことに、文節の表層形 (string) と姉妹ノードにおける順番 (order) は、比較的独立に考えることができる。ここでは、抽象的な記述から、文節の表層形を求めることに焦点を絞ろう。このためは、文節の定義 (モデル) が必要となる。

ここで、文節のモデルを考える方向として、少しずつ抽象化する方向 (ボトムアップ) で考える。なぜなら、生成できない文が存在するのはツールとして致命的であるからである。

いま、文節は、主要部と (後続する) 機能部から構成されるものとみなそう。ここで、機能部は、0 個以上の助詞の列とする。このような文節モデルの導入により、文節の表層形を求める問題は、主要部の表層形を求める問題と、機能部の表層形を求める問題に分解できる。

まず、抽象度の低いレイヤとして、主要部や機能部を形態素列で指定するレイヤを導入する。「列」は並び順も指定することを意味する。ここで抽象化されるのは、活用形である。活用形は、明示的に指定されなければ、後続形態素によって定まるデフォールトを用いる。記法については細かく説明しないが、次のような生成が可能である。「運ぶ」「レル」などの形態素の詳細記述は、辞書に定義してあることが前提である[1]。

```
{ muw: '運ぶ/レル/テいる/ナイ' }
  ⇒ 運ばれていない
{ muw: '運ぶ/レル/ナイ/テいる' }
  ⇒ 運ばれないでいる
```

実は、「ナイ」のテ形には「なくて」と「ないで」の 2 種類が存在するが、「テいる」が接続するのは「ない

[1] 文生成器が準拠する文法は、益岡・田窪文法 [3] である。「レル」と「ナイ」は接辞、「テいる」はテ形複合動詞として扱うので、ここに示す 2 例は、いずれも文節の主要部である。

Copyright(C) 2015 The Association for Natural Language Processing.
All Rights Reserved.

で」のみである。活用形の抽象化では、このような制約を生成器に組み込む必要がある。

主要部では、形態素の並び順は明示的に指定するものとするが、機能部では、並び順を抽象化した (生成器が自動的に並び順を決定する) レイヤも導入する。

```
{ p: ’ダケ副/ガ格’ } ⇒ だけが
{ p: ’ガ格, ダケ副’ } ⇒ だけが
```

この例の場合、「がだけ」という並び順は許されなので、並び順は一意に定まる。

より抽象度の高いレイヤとして、「文節の機能」に基づくレイヤを導入する。文節の機能としては、補足語、連体修飾、連用修飾などを設定する。これらに加えて、節末となる文節では、補足節、連体節、副詞節、並列節などの節タイプも、文節の機能とみなす。

たとえば、イ形容詞の連体修飾には、基本形 (イ形) とタ形が存在する。さらに、「大きい」「小さい」などのいくつか形容詞には、ナ形が存在する[2]。このため、これらをパラメータ化する必要がある。「大きい」の連体修飾のデフォルトは「ナ形」とする。

```
{ mp: ’大きい’, f: ’連体修飾’ }
  ⇒ 大きな
{ mp: ’大きい’,
  f: { type: ’連体修飾’, form: ’イ形’ } }
  ⇒ 大きい
{ mp: ’大きい’,
  f: { type: ’連体修飾’, form: ’タ形’ } }
  ⇒ 大きかった
```

節末文節は、かなりやっかいである。たとえば、補足節では、形式名詞等の挿入が必要となるが、「起きるのが」を 1 文節とみなすか 2 文節とみなすかを定めなければならない。現在の私の考えは、通常の文節以外に、「機能文節」という特殊なタイプの文節を設定し、通常の文節に「組み入れることができるようにする」というものである。つまり、「起きるのが」は 2 文節とみなすが、文節木 (JBT) においては、「起きるのが」をあたかも 1 文節のようにみなす。詳細は、稿を改めて述べるつもりだが、次のような生成を認める。

```
{ mp:’起きる’,
  f: { type: ’補足節’, case: ’ヲ格’ } }
  ⇒ 起きることを
{ mp:’起きる’,
  f: { type: ’補足節’, func_noun: ’の’,
       case: ’ガ格’ } }
  ⇒ 起きるのが
```

副詞節についても、同様の考え方を適用する。ここでは、節を受ける名詞を、形式名詞的に扱う。

[2] これらのナ形は、ふつう、連体詞として扱われることが多いが、「目の大きな少女」のように補足語 (格要素) を取りうるので、形容詞とみなした方がよい。

```
{ mp: ’実行する’,
  f: { type: ’副詞節’, func_noun: ’ため’ } }
  ⇒ 実行するため
```

文節の機能は、おおよそ、主要部の末尾の活用形、機能部 (助詞)、および、文節に組み入れる機能文節を規定する。以下に、主要部の末尾の活用形を統制する例を示す。

```
{ mp: ’読む’,
  f: { type: ’副詞節’, conj: ’ナガラ接’ } }
  ⇒ 読みながら
{ mp: ’終わる’,
  f: { type: ’副詞節’, func_noun: ’後’ } },
  ⇒ 終わった後
```

以上のような例で、おそらく想像がつくと思われるが、「文節の機能」という抽象化の導入には、次のような作業が必要である。

1. 文節の機能タイプを設定する。
2. 各タイプが取りうる (表層) 形式を列挙する。
3. それらの生成を制御するパラメータ集合を設計する。

この作業は、日本語の文法を生成用に再編成することと、ほとんど等価である。

8 まとめ

本論考の結論は以下のとおりである。

> 文生成器を作ることは、文の表層形を、それを合成する関数とパラメータに分解することに他ならない。それは、日本語の文法を生成用に再編成することと、ほとんど等価である。

この作業は、絶望的に難しいのか。あるいは、真摯に取り組めば乗り越えられる難易度なのか。願わくば後者であってほしいものである。

謝辞 本研究は、JSPS 科研費 24300052、および、中山隼雄科学技術文化財団の研究助成に受けて実施した。

参考文献

[1] 高木大生, 佐藤理史, 松崎拓也. プロットと背景知識を用いた短編小説の自動生成. 情報処理学会第 77 回全国大会講演論文集, 2015.

[2] Ehud Reiter and Robert Dale. *Building Natural Language Generation Systems.* Cambridge University Press, 2000.

[3] 益岡隆志, 田窪行則. 基礎日本語文法―改訂版―. くろしお出版, 1992.

Copyright(C) 2015 The Association for Natural Language Processing.
All Rights Reserved.