7　文の構造と文法理論

我々が日常話をしている言葉や文の中に、何らかの構造 (特に前節で述べたような構造) があることを普通意識する事は少ない。しかし、頭の中にある言語処理機構ではこれを駆使していることは想像に難くない。

たとえば、次の英語の文章を読んで、意味を取ってみよう。

> The paper discusses the theory underlying the system, the representations and algorithms used in the implementation, the semantic constraints which support the heuristics necessary to achieve tractable learning, the limitations of the current theory and the implications of this work for language acquisiton research.

一見複雑そうに見えるこの文を解釈するには、課題 9 でやったように、どの語句とどの語句が関係しているか（どの句や語が構成素となってより大きな句を構成しているか）を考えなくてはならない。このような操作は、母語である日本語を読む場合でも我々は頭の中で行なっているのだが、無意識的に行なっているために気がつかないのかも知れない。[16]

7.1　言語研究の対象と文法理論の妥当性

- Chomsky(生成文法の創始者) による研究対象としての「言語の能力」の分類
  - 言語能力 (linguistic competence):
    頭の中に存在する知識としての文法
  - 言語運用 (linguistic performance):
    実際に言語を使用する際のいろいろな知識。状況に即した適切な表現の用い方にかかわる知識など。
- 「言語は知識の計算的なシステムである」: 有限の形式的な規定により、無限の文をつくり出す事ができる体系

  文法理論とは、そのような形式的な規定、すなわち規則の体系
- 文法理論の妥当性
  - 観察的妥当性
    観測される事実に対応するかどうか。事実をもれなく、しかも事実のみを記述するかどうか。
    例 : 日本語の記述として、「任意の文字を並べる」(過剰な記述)、「名詞の後に助詞「が」をつけたものと動詞をこの順に並べる」(過少な記述)

[16] 構造は以下：

NP(The paper) VP(V(discusses) NP(the theory VP-MOD(underlying the system)), NP(the representations and algorithms VP-MOD(used in the implementation)), NP(the semantic constraints RC(which support NP(the heuristics AP(necessary to achieve tractable learning)))), NP(the limitations of the current theory) and NP(the implications PP(of this work) PP(for language acquisiton research))).

– 記述的妥当性
過剰でも過少でもない理論だけでは不十分。心理的実在性や、言語学的に意味のある一般化がなされているかどうか、が問題。
例：英語の五文型の考察 ⇒ 主語と動詞句との構成

– 説明的妥当性
言語が習得されるということから考えて、幼児がどのような初期状態から、文法を習得するかをよりよく説明できるか。

7.2 古典的句構造文法

言語の文が無限に生成できること、またそれが「名詞句」や「動詞句」というような意味的もしくは文法的なまとまりをもつことに注目して、文の構造を記述する。

そのための第一歩として、5.3 節で検討した句の構造を、「形式言語理論」の記法を用いて記述することを考える。以下は、句の構造を記述するための「規則 (rule)」と呼ばれる (書き換え規則、ともいう)。簡単に言えば、句として認められるようなものだけを過不足なく記述するための道具である。

(37) $\alpha \longrightarrow \beta$

ここで α はその文法で使用する範疇 (category) 記号である。範疇とは、文、名詞句、動詞句というような、語や句の分類に用いられる品詞の、より一般的な概念である。

β は α がどのような単語や範疇の並びから構成されるかを記述するものである。木構造で言えば、あるノード (A とする) に注目し、そのノードを親ノードとするすべての子ノードを $B_1, \ldots, B_n$ とすると

$$A \rightarrow B_1\ B_2\ \ldots\ B_n$$

が一つの書き換え規則に対応する。[17]

例えば、日本語には次のような文法規則 (の集合) が考えられる。(範疇名はここでは適当である。後で修正する。)

[17] この書き換え規則は、A を分解してみると $B_1\ B_2\ \ldots\ B_n$ に分けることができる、とも読めるし、逆に $B_1\ B_2\ \ldots\ B_n$ から A が組み立てられる、とも読める。

表 7.1 日本語の文法規則を書き換え規則で表わした例

文	→	動詞句
動詞句	→	動詞
動詞句	→	助詞句　動詞句
動詞句	→	副詞句　動詞句
名詞句	→	名詞
名詞句	→	連体句　名詞句
助詞句	→	名詞句　助詞
副詞句	→	副詞
連体句	→	形容詞
連体句	→	名詞句　連体助詞

これ以外に、辞書に当たる以下のような書き換え規則も必要である。

表 7.2 日本語の辞書に相当する書き換え規則の例

名詞	→	太郎
名詞	→	花子
名詞	→	本
形容詞	→	黒い
助詞	→	が
助詞	→	を
副詞	→	ゆっくり
連体助詞	→	の
動詞	→	読んだ
動詞	→	見た

これらによって、例えば「太郎が本を読んだ」や「花子が太郎の本を読んだ」という文がどのような構造をしているか、を解析することができる。つまり、上の文法規則をどのように適用していけば、文がどのような語、そして語の集まりである句から構成されているかを調べて表すことができる。この構造を図で表したものを解析木とよぶ、ということは 5 章で述べた。

今あげた文法規則では、単純なものであったが、実際にこれを文解析に用いようとすると、単純過ぎて、実際の文を解析するには十分ではない。例えば、英語によく見られるように同じ言葉でもいくつかの品詞に対応していたり (例えば 'run' は動詞と名詞とがある)、同じ品詞であってもより細かく分類する必要があったり (「読んで」と「読んだ」では後に続きうる語が異なる)、同音異義語を区別する必要があったり (「いった」では「言った」と「行った」という違いがあり、例えば「電車で行った」はよいが「電車で言った」はおかしい)、というような問題がある。

従って、単語と品詞とを対応させるだけではなく、単語が持っているいろいろな形態的、意味的、文法的な情報を組み込んだ文法規則を作る必要がある。機械翻訳などで実際に文解析に用いられている文法規則の数は数千と言われている。なお、最近の研究では、単語

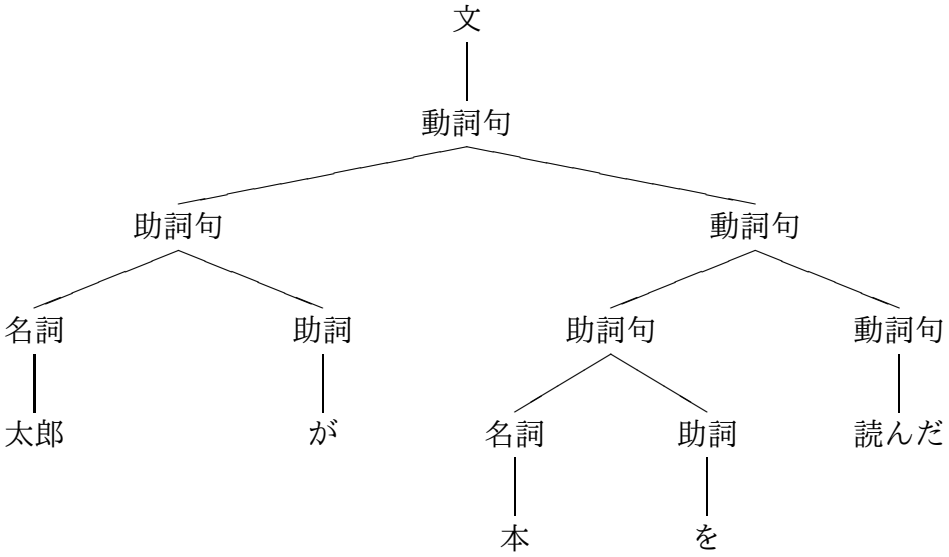


図 7.1 「太郎が本を読んだ」の構文木

に持たせるべき情報を体系化し、その情報間の関係を規則化することで、かなり少ない数の文法規則で記述できることが知られている。

7.3 古典的句構造文法による構文解析の例

ここでプログラミング言語 Prolog を用いた日本語の構文解析のプログラムの一つを紹介する。

Prolog では、古典的句構造文法を拡張した DCG (Definite Clause Grammar, 限定節文法) をサポートしている。つまり、DCG 形式で文法規則を書くと、その文法規則にもとづいて構文解析を行うプログラムを作成したことになるのである。

ここで、DCG 形式の文法規則とは、以下に示すように、書き換え規則に基づく形式の文法規則である:

```
文 --> 動詞句.
動詞句 --> 動詞.
動詞句 --> 助詞句, 動詞句.          動詞句 --> 副詞句, 動詞句.
名詞句 --> 名詞.                    名詞句 --> 連体句, 名詞句.
助詞句 --> 名詞句, 助詞.            副詞句 --> 副詞.
連体句 --> 形容詞.                  連体句 --> 名詞句, 連体助詞.
名詞 --> [太郎].                    名詞 --> [花子].
名詞 --> [本].
形容詞 --> [黒い].                  連体助詞 --> [の].
助詞 --> [が].                      助詞 --> [を].
副詞 --> [ゆっくり].
動詞 --> [読んだ].                  動詞 --> [見た].
```

このDCGのプログラムと表7.1、7.2にあげた書き換え規則と見比べてみると以下の違いがある:

- 文法規則に対して: → を-->で置き換え、→ の右辺にある範疇名同士をコンマ , で区切り、最後にピリオド . を付け加えてある
- 辞書に対して: → を-->で置き換え、→ の右辺にある語を [と] でくくり[18] 、最後にピリオド . を付け加えてある

これで原理上は構文解析器ができたのだが、それを実行させると

```
| ?-文 ([太郎, の, 本, を, 読んだ],[]).
yes
```

となり、「太郎の本を読んだ」という文が「認識」されたのはわかるが、どのように「解析」されたかは不明である。

そこで、以下のように「引数」を加えることを考える（これができるのが、DCG による「拡張」の一つ)。

```
文(文(X)) --> 動詞句(X).
動詞句(動詞句(X)) --> 動詞(X).
動詞句(動詞句(X,Y)) --> 助詞句(X), 動詞句(Y).
動詞句(動詞句(X,Y)) --> 副詞句(X), 動詞句(Y).
名詞句(名詞句(X)) --> 名詞(X).
名詞句(名詞句(X,Y)) --> 連体句(X), 名詞句(Y).
助詞句(助詞句(X,Y)) --> 名詞句(X), 助詞(Y).
副詞句(副詞句(X)) --> 副詞(X).
連体句(連体句(X)) --> 形容詞(X).
連体句(連体句(X,Y)) --> 名詞句(X), 連体助詞(Y).
名詞(名詞(太郎)) --> [太郎].
名詞(名詞(花子)) --> [花子].
名詞(名詞(本)) --> [本].
形容詞(形容詞(黒い)) --> [黒い].
連体助詞(連体助詞(の)) --> [の].
助詞(助詞(が)) --> [が].
助詞(助詞(を)) --> [を].
副詞(副詞(ゆっくり)) --> [ゆっくり].
動詞(動詞(読む)) --> [読んだ].
動詞(動詞(見る)) --> [見た].
```

規則は以下の形を持つ:

文法範疇 A(...) --> 文法範疇 B(...) .

[18] [と] でくくられたものは Prolog ではリストを表わす。

もしくは

文法範疇 A(...) --> 文法範疇 B(...), 文法範疇 C(...) .

Prolog では、英字の大文字から始まる名前は変数を表わす。また、括弧の中にはどのように指定してもよいが、ここでは構文木を書くことを考え、文法範疇がどのような要素から作られているかを記録するようにしてある。つまり、最初の規則では、『動詞句』を構成するものをX とすると、文 (X) が『文』を構成するものとして記録するようにしてある (同じ変数X を使っていることに注意)。同様に、三番目の規則は『助詞句』(その構成素をX とする) と『動詞句』(その構成素をY とする) から『動詞句』が構成されることを規定しているが、左側の『動詞句』を構成するものとして動詞句 (X,Y) を記録している、というわけである。

辞書は以下のように書かれている。

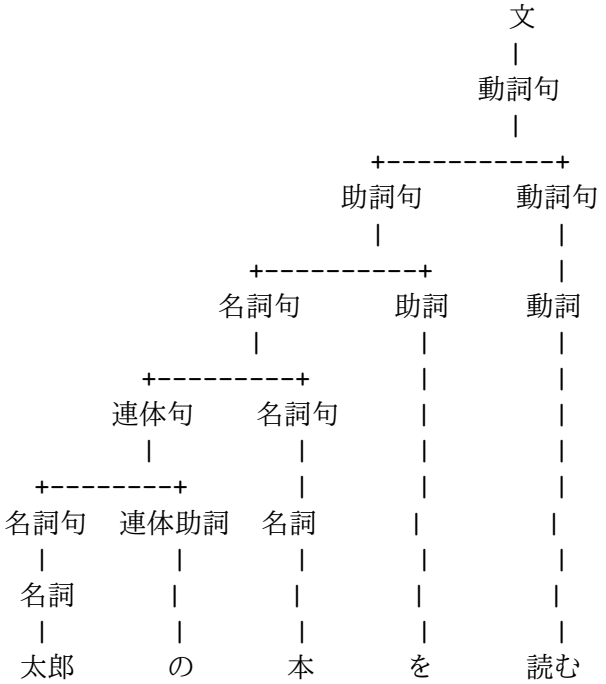
文法範疇 (...) --> [単語] .

ここでも括弧の中は自由であるが、構文木を書く都合から、どういう単語か、ということとその文法範疇とを指定してある。

上のように直した文法規則をもちいて、構文解析をしてみたのが以下である。さっきと比べ、引数の数が一個ふえている。そして、第一引数の変数 (X) に、構文解析の結果（構文木) が作られている。

```
| ?- 文 (X,[太郎, の, 本, を, 読んだ],[]).
X = 文 (動詞句 (助詞句 (名詞句 (連体句 (名詞句 (名詞 (太郎)), 連体助詞 (の)), 名詞句 (名詞
(本))), 助詞 (を)), 動詞句 (動詞 (読む)))) ?
yes
```

この結果を、木表示プログラムによって構文木を書かせたものが以下：



このように、DCG は簡便な構文解析器となりうるが、日本語を扱う場合には一つの大きな問題がある。それは、日本語では、次のようなパタンの文法規則が多く現れる、という

ことである:

例 1: $\alpha \rightarrow \alpha \quad \beta$ (例： 動詞句 $\rightarrow$ 動詞句　助動詞)

例 2: $\alpha \rightarrow \beta \quad \gamma$ (例： 文 $\rightarrow$ 名詞句　動詞句)

$\beta \rightarrow \alpha \quad \delta$ (例： 名詞句 $\rightarrow$ 文　名詞句)

これがなぜ問題を引き起こすか、というのは、Prolog の動作原理と関係するので、詳細は省略する。Prolog を学んだことがあるものは、考えて欲しい。

7.4 Prolog を用いた構文解析の実習

Prolog について講義するとすれば、1ヶ月くらいはかかってしまうので、その詳細には触れない。単に道具としてみなし、以下の例にしたがって試してほしい。ここで、?- は Prolog のプロンプトであり、このプロンプトの後に入力する。また、[と] の中はファイル名を一重引用符’でくくったものであり、一般にはファイルパスを書く (したがって、Prolog を立ち上げた時のカレントディレクトリにファイルがない場合には、相対パスや絶対パスによってファイル名を指定する)。

以下のことを注意しておく。

- 変数には半角英大文字、それ以外には原則として半角英小文字か日本語 (全角文字) を使うこと。また、全角の空白を入力しないように注意すること。
- プログラム名 (例えば、`translate_bup`) や文法範疇名と丸括弧の間にはスペースを置かないこと。
- Prolog の命令を実行すると何かメッセージが出て来るかもしれないが、エラーのメッセージでない限り無視すること。例えば以下は「警告」のメッセージであってエラーではない (warning が何を意味する単語か、くらいの知識は持っていて欲しい)。

```
{Warning: abolish(user:parsed_history/4) - no matching predicate}
```

- Prolog への命令には必ず最後に . (半角のピリオド) の入力を忘れないこと。
- 長いこと待っていても、何も出力がない場合は、入力の間違いか (カッコがあっていない、最後のピリオドを忘れているなど)、もしくは無限ループに陥っている可能性がある。それには C-c (コントロール C) を入力する。すると、以下のようなメッセージが表示されるので、a を入力すると Prolog のプログラムを中止させることができる。

```
Prolog interruption (h for help)?
```

- Prolog を終了するには、`halt.` を入力し、ENTER キーを押す。(最後のピリオドを忘れないこと)

1. ファイルを用意し、DCG 形式の文法規則を書く。ここで、ファイルの名前は任意であるが、以下ではparse.pl を、DCG 形式の文法規則が収められているファイルの名前とする。プログラムを日本語 (EUC コード) で書くこともできるが、Linux に限定されている (Windows ではできない)。また、

```
setenv SP_CTYPE euc
```

としておく必要がある (.cshrc に書き込んでおくと良い— .cshrc をいじっていなければこうなっているはず)。

注意: Windows 用の Sicstus Prolog では日本語は扱えないので、Windows で作業する場合には、半角英数文字だけを使って規則やプログラムを書くこと。

2. sicstus Prolog を起動し、bupTR.pl というファイルに入っているtranslate_bup というプログラムを用いて、parse.pl に入っている DCG 形式の文法規則を一括して BUP 形式の文法規則に変換する。以下の例では、bpTemp.pl を、parse.pl の規則を BUP 形式に変換した結果を収めるためのファイルとする。このファイル名も任意であるが、以下ではこの名前を用いることにする。

例
```
% sicstus
SICStus 3.12.2 (x86-linux-glibc2.3): Sun May 29 11:59:09 CEST 2005
Licensed to sccs.chukyo-u.ac.jp
?- ['bupTR.pl'].
?- translate_bup('parse.pl', 'bpTemp.pl').
?- halt.
```

最後の halt. の入力により sicstus Prolog が終了する。

3. エディタで、bpTemp.pl の==> をすべて--> に変換する。Linux の場合、sed コマンドを使えば、エディタを立ち上げなくても変換できる。

例
```
sed 's/==>/-->/' < bpTemp.pl > jparser.pl
```

4. sicustus を起動し、bup.pl と jparser.pl とtreeJdraw.pl に入っているプログラムを読み込む。

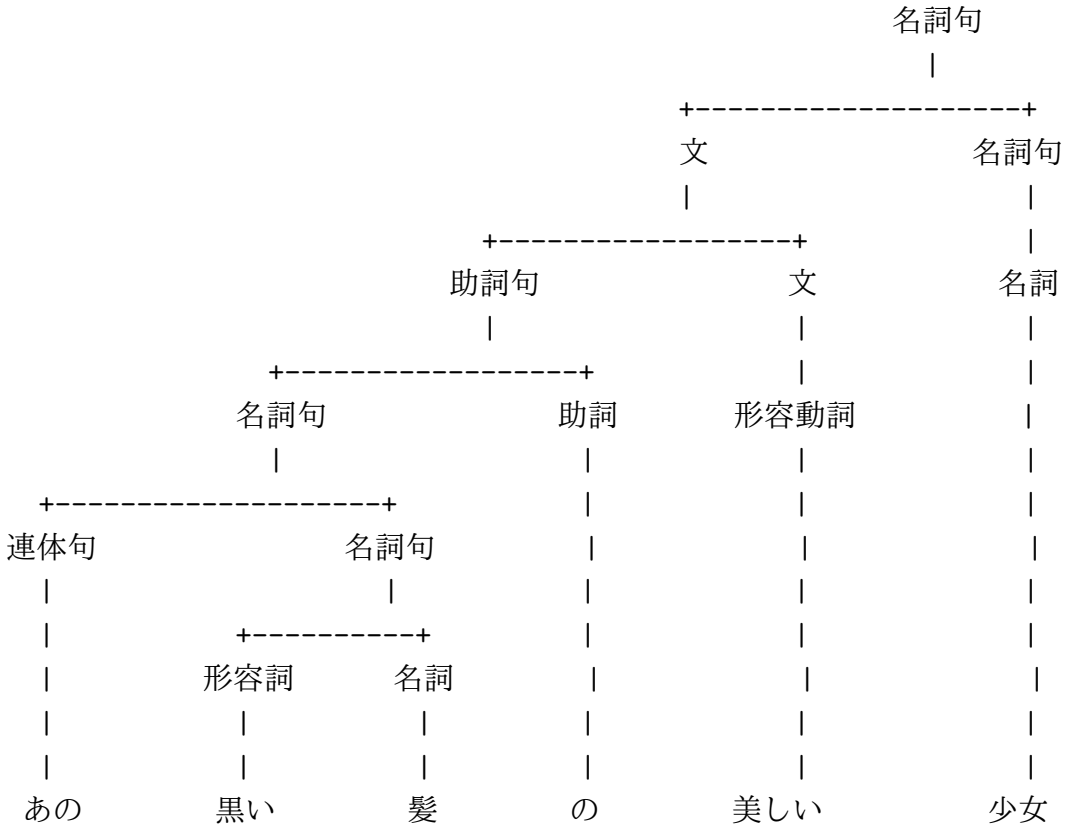
例
```
?- ['bup.pl','jparser.pl','treeJdraw.pl'].
```

5. 以下のようにparse というプログラムに適当な引数を与えて実行する。ここで、プログラムの構文木を出力して最後にX = 名詞句 (文 (助詞句 . . . 名詞 (少女)) ? というようなメッセージを出して止まった時に、; を入力すると、Prolog は別な解を出力しようとする (なければ、no と出力される)。; ではなく単にリターンを押せば、そこで

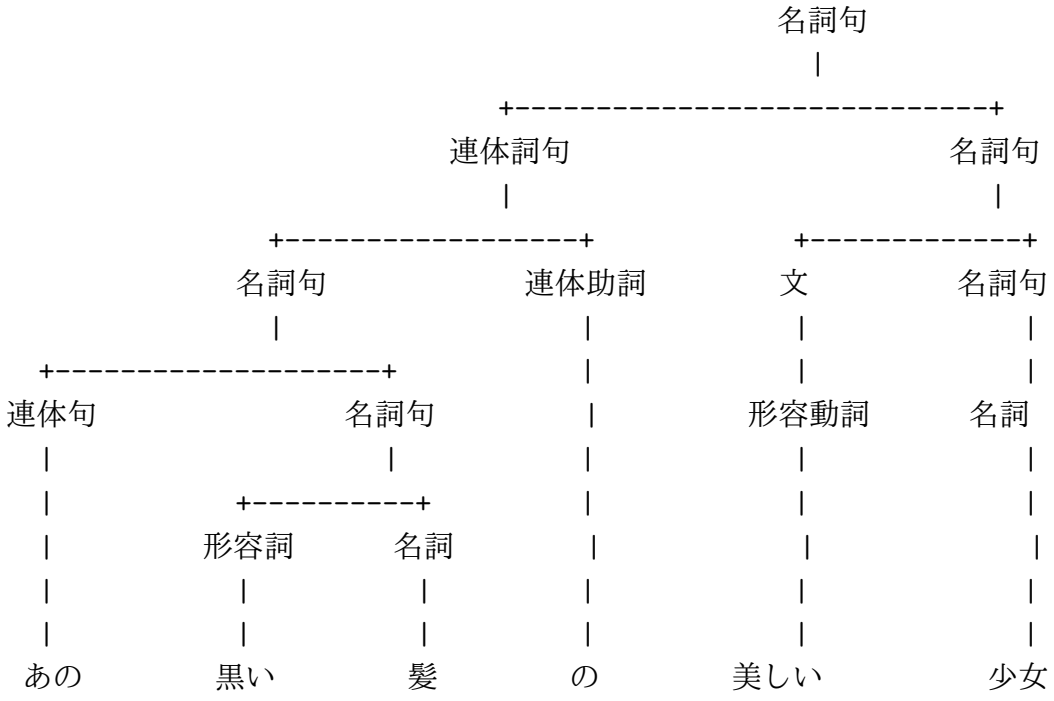
プログラムの実行が一旦終了となり、次の命令を受け付けるプロンプト (?-) が出力されて、命令の待ち状態となる。

なお、以下に例を示すが、parse というプログラムは、最初に文法範疇名 (単にA のような変数でもよい)、次に例に示すように[ ] の中に単語をカンマで区切って並べ、最後に[X] のようなものを与えて呼び出す。二番目の単語列が一番目で指定された文法範疇を構成するものとして文法規則が書けていれば、三番目の引数で指定された X という変数にその構文木の構造がセットされる。以下の例では、その構造の情報を draw というプログラムに渡して、構文木を描かせている。

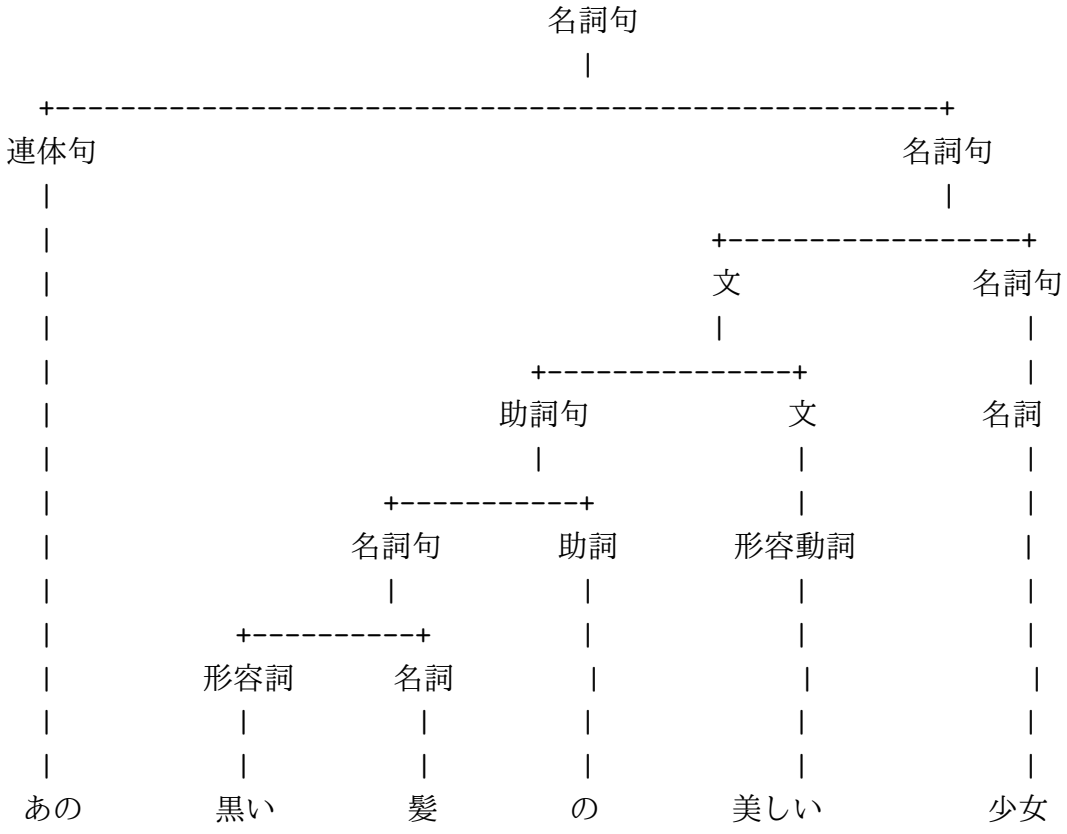
例　?- parse(名詞句,[あの, 黒い, 髪, の, きれいな, 少女],[X]),draw(X).



X = 名詞句 (文 (助詞句 (名詞句 (連体句 (あの), 名詞句 (形容詞 (黒い), 名詞 (髪))), 助詞 (の)), 文 (形容詞 (美しい))), 名詞句 (名詞 (少女))) ? ;



X = 名詞句(連体詞句(名詞句(連体句(あの),名詞句(形容詞(黒い),名詞(髪))),
連体助詞(の)),名詞句(文(形容詞(美しい)),名詞句(名詞(少女)))) ? ;



X = 名詞句(連体句(あの),名詞句(文(助詞句(名詞句(形容詞(黒い),名詞(髪)),
助詞(の)),文(形容動詞(美しい))),名詞句(名詞(少女)))) ?