Skip-Gram モデルを用いたプログラミング言語の文法推定に向けて

松永 智將　山崎 徹郎　Daniel Perez　千葉 滋

Skip-gram モデルを用いて獲得したベクトル表現を基に，既存のソースコード群からプログラミング言語の文法を推定する手法について述べる．近年では，オープンソースソフトウェアが増えたことで学習データとして大量のプログラミング言語のソースコードを活用できるようになった．これらのソースコード群から文法を獲得できれば，コーディングスタイル検査器等の応用先が考えられる．本研究では，自然言語処理の分野でよく用いられる Skip-gram モデルと，CYK 法に基づく生成規則の集合の更新手法によってプログラミング言語の文法を推定する方法を検討する．

We describe a method for inferring the grammar of programming languages using the Skip-gram model. Recently, we became able to use a lot of source codes as training data because the number of open-source softwares has increased. If we can generate the grammar from these codes, we can apply it to analysis tools such as coding style checker. In this paper, we consider a method for inferring grammar using the Skip-gram model and the CYK parsing algorithm.

1　はじめに

プログラミング言語の文法を既存のソースコードから獲得することは可能だろうか．近年では，GitHub[†1] 等に公開されるオープンソースソフトウェアが増えたことで，学習データとして大量のソースコードが活用できるようになった．これらのソースコード群から文法を推定することができれば，コーディングスタイル検査器などの静的解析ツールへの応用が見込める．本研究では既存のソースコード群を用いて，プログラミング言語の文法を獲得するための汎用的な手法について検討する．

2　プログラミング言語の文法推定

形式文法は，BNF や EBNF を用いて図 1 のように「生成規則の集合」で表現することができる． 生成規則の左辺に表れるような記号を非終端記号と呼び，左辺の非終端記号は右辺の記号列に置き換えることができる．生成規則の右辺は，非終端記号または終端記号からなる記号列である．ここで終端記号とは，生成規則による置換をそれ以上行うことができない文字を指す．開始記号と呼ばれる特定の非終端記号から始めて，生成規則による置換を繰り返すことによって生成できる文字列の集合を言語と呼ぶ．

$$
\begin{array}{lcl}
Eq & \rightarrow & \text{“}=\text{”} \\
Lp & \rightarrow & \text{“(”} \\
Rp & \rightarrow & \text{“)”} \\
Op & \rightarrow & \text{“}+\text{”} \\
Var & \rightarrow & [\text{“}a\text{”} - \text{“}z\text{”}] + \\
Expr & \rightarrow & Var \\
 & | & Var\ Op\ Expr \\
 & | & Lp\ Expr\ Rp \\
Stmt & \rightarrow & VarEqExpr
\end{array}
$$

図 1　EBNF で記述された文法例

自然言語処理の分野では，自然言語の文法推定の研究が多く行われている．自然言語の文法はプログラミング言語の文法と比べて，非終端記号の数がわずかで

Grammar Induction for Programming Languages using Skip-Gram model

Tomomasa Matsunaga, 東京大学大学院情報理工学系研究科創造情報学専攻, Graduate School Information and Science Technology, The University of Tokyo.

†1 https://github.com

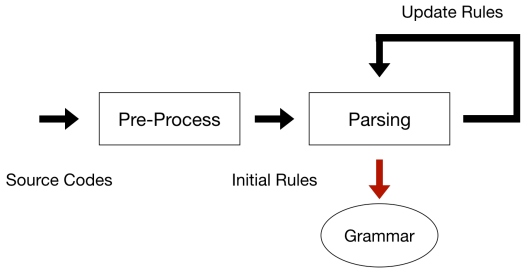

図 2　本システムの概略図

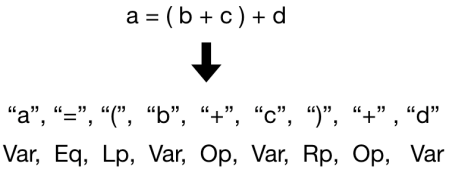

図 3　文字列からトークンとトークンの型情報への変換

ある等の違いがある．

ある言語のソースコード群から非終端記号と生成規則に対応する構造を獲得することができれば，その言語の文法を推定することができたと見なせる．プログラミング言語の場合は，言語設計者が定めた規則に従う文字列を使用者が記述するため，通常はトップダウンに文法が決定される．ソースコードから文法を得るために，ボトムアップに文法を解釈すると，文法とは交換可能な記号と連続して出現しやすい規則の集まりであると考えられる．交換可能な記号列の集合は文法における非終端記号に対応し，それらの連続出現度合いによって生成規則の右辺の並びが決定できる．

3　Skip-gram モデルを用いた文法の推定法

本論文で提案する手法では，はじめに「生成規則の集合」の初期値を決定し，その「生成規則の集合」を更新していくことで最終的な文法を獲得する．本システム全体の概略図を図 2 に示す．図 2 のように，本手法は初期の「生成規則の集合」を作成するまでの前処理 (3.1 節) と，その時点の「生成規則の集合」を用いて構文解析を行うことで「生成規則の集合」を更新する処理 (3.2 節) の二段階に分かれる．「生成規則の集合」を更新する回数はパラメータとして指定し，指定回更新された「生成規則の集合」を最終的な文法とする．

3.1　「生成規則の集合」の初期値を決定するための前処理

本節では，データセットから「生成規則の集合」の初期値を決定するための処理について述べる．データセットは特定のプログラミング言語で記述された複数のファイルから成り，前処理はデータセット内の全てのファイルに対して行われる．3.1.1 節から 3.1.6 節で述べる処理を順に行うことで,「生成規則の集合」の初期値を決定する．

3.1.1　字句解析器を用いた文字列からトークン列への変換

はじめに，データセット内の各ファイルに記述された文字列を字句解析器を用いてトークン列に分割する．ここで使用する字句解析器は，学習の対象とするプログラミング言語の字句解析器をそのまま用いる．図 3 のように,「字句解析時に何であると認識されたか」を示す型情報が各トークンに対して得られる (以降，トークンの型と呼ぶこととする)．ここでは，図 3 の a, b のように，型が同じトークンであっても別々のトークンとして分割される．トークンの型がスペース，改行またはコメントのいずれかであると認識されたトークンは，この段階で取り除く．

3.1.2　出現ファイル数に基づくトークンの置換

次に，データセット内に出現する各トークンの出現ファイル数を数え上げる．出現ファイル数を数え上げる対象はトークンの型ではなくトークン自身である．出現回数がデータセットに用いたファイル数の 5%に満たないトークンを，トークンの型で置き換える．図 4 の a, b, c, d のような変数名は，大半のファイルには同じ名前で現れないので，トークン自身をトークンの型と同じ Var に置換する．この処理は,「一般的に用いられていない出現度の低いトークン」を「同一のトークンの型を持つ場合には同一のトークンとして扱う」ことを目的として行っている．そのため，同じトークン名が一つのファイル内で多く用いられる場合にそのトークンを残してしまうのは目的に沿わない

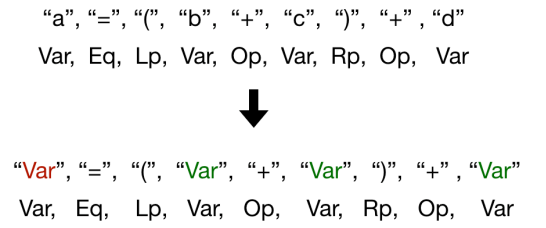

図 4　トークンの出現ファイル数に基づくトークンの置き換え処理

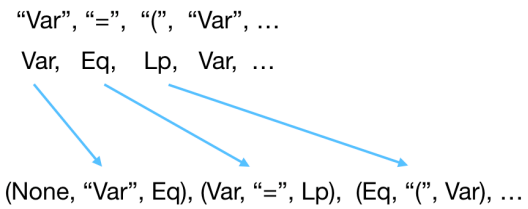

図 5　各トークンの三つ組への変換

ので，総出現回数ではなく出現ファイル数によって置き換えを行なっている．ここで，置き換えの有無に関わらずトークンの型はトークンと別に保持を続ける．

3.1.3　トークンから「トークンの三つ組」への変換

前節で置換したトークン列の各トークンを，以下の三点を合わせた三つ組に変換する (図 5)．

(i)　注目しているトークンの左隣のトークンの型

(ii)　注目しているトークン

(iii)　注目しているトークンの右隣のトークンの型

以降,「トークンと左右のトークンの型による三つ組」を単に「トークンの三つ組」を呼ぶこととする．ファイル内で最初に現れるトークンの左隣にはトークンがないので，字句解析器によって与えられる型とは別に $None$ というトークンの型を導入して三つ組へ変換することとする．同様に，ファイル内で最後のトークンも右隣のトークンの型として $None$ を指定することで三つ組に変換する．以降の処理は，ここで置き換えた「トークンの三つ組」の列に対して行う．

3.1.4　「トークンの三つ組」に対する二種類のベクトル表現の獲得

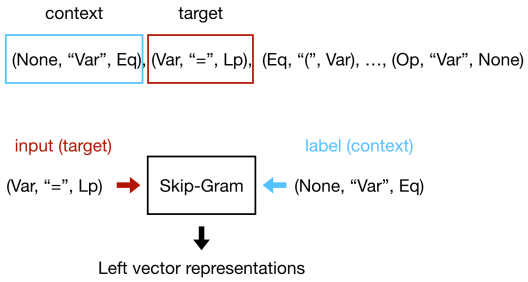

図 6　Skip-gram モデルによる左側の文脈のみを考慮した「トークンの三つ組」のベクトル表現の獲得

Neural network のモデルの一つである Skip-gram モデル [2] を用いて各「トークンの三つ組」に対してベクトル表現を与える．

通常の Skip-gram モデルでは，注目している単語に対してその左右に出現する単語をラベルとするような教師あり学習を行う．この学習によって得られる neural netowork の重みベクトルを，単語に対するベクトル表現として扱う．本手法では，各「トークンの三つ組」に対して二種類のベクトル表現を与える．一種類目のベクトル表現を得るために，注目している「トークンの三つ組」に対して，その左隣に出現する「トークンの三つ組」をラベルとして与えて学習を行う (図 6)．これにより，左側の文脈のみを考慮した「トークンの三つ組」のベクトル表現が得られる．二種類目のベクトル表現は，考慮する文脈の向きを反対にして，各「トークンの三つ組」に対してその右隣に出現する「トークンの三つ組」をラベルとした学習を行う．

3.1.5　「トークンの三つ組」のベクトルの距離によるクラスタリング

ここまでの処理によって，各トークンの三つ組に対して「左側の文脈のみを考慮したベクトル表現」と「右側の文脈のみを考慮したベクトル表現」が与えられている (図 7)．図 8 のように,「左側の文脈のみを考慮したベクトル表現」の距離でトークンの三つ組をクラスタリングし，同様に「右側の文脈のみを考慮したベクトル表現」の距離でトークンの三つ組をク

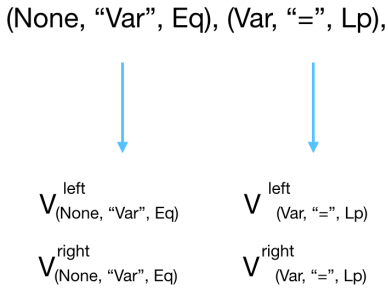

図 7　トークンの三つ組に対応する 2 種類のベクトル表現

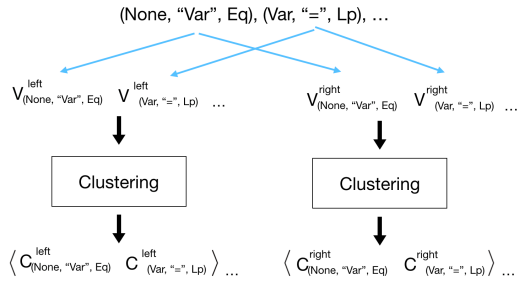

図 8　「トークンの三つ組」を「左側の文脈のみを考慮したベクトル表現」の距離と「右側の文脈のみを考慮したベクトル表現」の距離で別々にクラスタリング

ラスタリングする．ここで，クラスタリング手法には X-means 法 [3] を用いた．X-means 法とは，初期クラスタ数を決めて k-means 法を適用することで得られるクラスタを 2 分割する，という処理を再帰的に行うアルゴリズムである．

3.1.6　「生成規則の集合」の初期値の決定

本手法では推定する文法として文脈自由文法を仮定し，チョムスキー標準形で表されるような「生成規則の集合」を獲得する．チョムスキー標準形では文脈自由文法は図 9 のような規則のみで表現される．ここで，A, B, C は非終端記号であり，α は終端記号である．全ての文脈自由文法はチョムスキー標準形で書き表すことができることが知られている．

「トークンの三つ組」に与えられた二つのクラスタを基に「生成規則の集合」の初期値を決定する．本手法では，チョムスキー標準形の生成規則 (図 9 の式 1) に対応する生成規則で「生成規則の集合」を表現する．本手法における終端記号は「トークンの三つ組」なので，式 1 の右辺は「トークン列の三つ組」になる．式 1 のような生成規則の左辺に現れる非終端記号を，本手法では 3.1.5 節で得た二種類のクラスタの名前の組み合わせてによって表す．式 3 に本手法で初期値として与える生成規則の例を示す

$$A \rightarrow \alpha \quad (1)$$
$$A \rightarrow B\ C \quad (2)$$

図 9　チョムスキー標準形の表記法

$$\langle C_t^l C_t^r \rangle \rightarrow t \quad (3)$$

ここで，t は「トークンの三つ組」を表し，C_t^l は，「トークンの三つ組」を「左側の文脈のみを考慮したベクトル表現」の距離でクラスタリングをした結果，t が属しているクラスタを表す (以降，t が属する左側クラスタと呼ぶこととする)．同様に，C_t^r は「トークンの三つ組」を「右側の文脈のみを考慮したベクトル表現」の距離でクラスタリングをした結果，t が属しているクラスタを表す (以降，t が属する右側クラスタと呼ぶこととする)．データセットに現れる全ての「トークンの三つ組」に対して式 3 のような生成規則を生成し，これらを「生成規則の集合」の初期値として扱う．

3.2　文法の更新

最終的な文法を獲得するために，3.1 節の前処理によって得られた初期値から始めて,「生成規則の集合」の更新を指定回数行う.「生成規則の集合」の更新のために必要となる主な処理は，現在の生成規則で構文解析を行うことで「連続する二つの非終端記号」の出現回数を数え上げることである．

「生成規則の集合」の更新時に行う構文解析には CYK 法 [1] を用いた．CYK 法は，入力文字列内の全ての隣り合う部分列の組み合わせに対して，生成規則を適用可能か考慮することで構文解析を行う手法である．ここで，生成規則が適用可能な場合に，生成規則の右辺を左辺の非終端記号で置き換えることを「還元する」という．CYK 法では，チョムスキー標準形

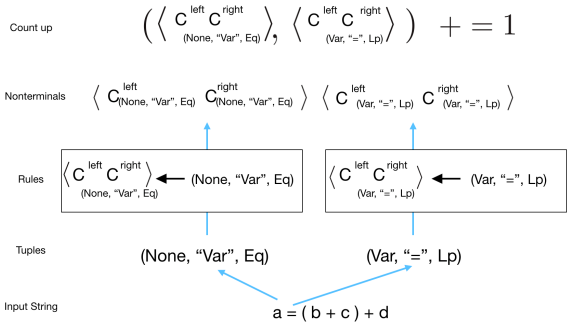

図 10　非終端記号の組の数え上げ

で記述された文脈自由文法を動的計画法に基づいて構文解析することができる．

3.2.1　初期状態の「生成規則の集合」の更新過程

「生成規則の集合」の更新過程を説明するために，初回の「生成規則の集合」の更新方法を述べる．初期状態では，出現し得る「トークンの三つ組」全てについて式 3 のような生成規則が定義されているため，各「トークンの三つ組」は一つの非終端記号に変換できる (図 10)．この非終端記号の列に対して,「隣り合う非終端記号の組」の出現回数を数え上げる．

「隣り合う非終端記号の組」の出現回数が数えられたら，出現回数の平均値を計算する．こうして得た出現回数の平均値よりも多く出現する「隣り合う非終端記号の組」全てに対して，以下の式 4 のような新しい生成規則を作る．

$$\langle C^l_{t_i} C^r_{t_{i+1}} \rangle \quad \rightarrow \quad \langle C^l_{t_i} C^r_{t_i} \rangle \ \langle C^l_{t_{i+1}} C^r_{t_{i+1}} \rangle \quad (4)$$

ここで,「トークンの三つ組」の列の i 番目の「トークンの三つ組」を t_i とし，t_i が属する左側クラスタを $C^l_{t_i}$ とする．また，t_i が属する右側クラスタを $C^r_{t_i}$ とする．式 3 と同様に，右辺の非終端記号は「トークンの三つ組」が属する左側クラスタと右側クラスタの組み合わせで表される．また，式 4 では左辺の非終端記号が以下の二つの組み合わせで表現される．

(i)　左側の「トークンの三つ組」が属する左側クラスタ

(ii)　右側の「トークンの三つ組」が属する右側クラスタ

新しく作られた式 4 のような「生成規則の集合」が，初期状態を更新した後の「生成規則の集合」となる．

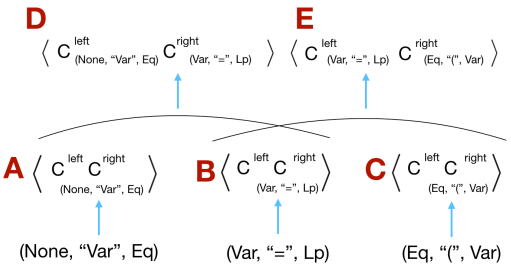

図 11　「トークンの三つ組」の列の還元

3.2.2　「生成規則の集合」の更新

「生成規則の集合」の更新時には,「トークンの三つ組」の列を生成規則で還元することで，図 11 の A ～E のような非終端記号が得られる．ある非終端記号を得るまでに還元された「トークンの三つ組」の列を,「その非終端記号が表す区間」と呼ぶこととする．また，以下の二点を満たす非終端記号 X と非終端記号 Y の組を「隣り合う非終端記号の組」とする．

(i)　X と Y が表す区間が重ならない

(ii)　「X が表す区間の最右端のトークンの三つ組」と「Y が表す区間の最左端のトークンの三つ組」が「トークンの三つ組の列」において隣接する

図 11 の「A と B」や「A と E」は「隣り合う非終端記号の組」であり,「A と C」や「B と E」,「D と E」は「隣り合う非終端記号の組」**ではない**．

「生成規則の集合」の更新処理は，二回目以降も初回と同様に以下の手順が繰り返し行われる．

1. データセットの全ファイルに対して構文解析を行う．
   (a)　3.1.3 節で得られた三つ組みの列を CYK 法で構文解析する．
   (b)　構文解析の過程で登場する「隣り合う非終端記号の組」の出現回数を全て数え上げる．
2. 「隣り合う非終端記号の組」の出現回数の平均値を計算する．
3. 手順 2 で計算した平均値より多く登場する「隣り合う非終端記号の組」全てに対して，右辺がそのような組み合わせとなる生成規則を生成する (後述)．
4. 手順 3 で得られた「生成規則の集合」を更新後

の「生成規則の集合」とする．

5. 更新回数が指定回数に満たなければ，手順 1 へ戻る．

通常の CYK 法とは異なり，本手法では考慮する文字列の長さに上限を決めて構文解析を打ち切って学習を行なっている．

上記の処理の手順 3 では，次のようにして新しい生成規則を作る．トークンの三つ組」の列の i 番目の「トークンの三つ組」を t_i とし，t_i が属する左側クラスタを $C^l_{t_i}$，t_i が属する右側クラスタを $C^r_{t_i}$ とする．j, k についても i と同様に定義する．ここで，隣り合う非終端記号の組を $\langle C^l_{t_i} C^r_{t_k} \rangle$ と $\langle C^l_{t_{k+1}} C^r_{t_j} \rangle$ とすると，作られる生成規則は

$$\langle C^l_{t_i} C^r_{t_j} \rangle \quad \rightarrow \quad \langle C^l_{t_i} C^r_{t_k} \rangle \ \langle C^l_{t_{k+1}} C^r_{t_j} \rangle \quad (5)$$

である．それぞれの非終端記号は「トークンの三つ組」の列を還元して得られるものである．本手法の生成規則の作り方では，非終端記号 $\langle C^l_{t_i} C^r_{t_k} \rangle$ は，還元前の「トークンの三つ組」の列の最左端 t_i が属する左側クラスタと最右端 t_k が属する右側クラスタからなる．したがって，新しく作られる生成規則の左辺の非終端記号も，以下の二つの組み合わせで表現される．

(i) 左辺の非終端記号が表す区間で一番左の「トークンの三つ組」が属する左側クラスタ

(ii) 左辺の非終端記号が表す区間で一番右の「トークンの三つ組」が属する右側クラスタ

こうしてより長い「トークンの三つ組」の列を還元して得られる非終端記号を作っていく．

3.3 獲得した文法による構文解析

本手法で獲得した文法の終端記号は，通常の字句解析器によって得られるトークンではなく，そのトークンに一連の処理を施した「トークンの三つ組」である．そのため，獲得した文法で実際に入力文字列を構文解析する際には，学習時と同様にあらかじめ入力文字列を「トークンの三つ組」に変換する必要がある．以下に，新しく与えられる入力文字列を「トークンの三つ組」の列に変換するための処理を示す．

1. 3.1.1 節で用いた字句解析器と同じ字句解析器を用いて，入力文字列からトークン列への変換処理を行う．学習時と同様に，各トークンに対するトークンの型の情報も付与される．
2. 3.1.2 節と同様にトークンの置換処理を行う．ここでは，「データセット内に一度も現れなかったトークン」または「データセット内で出現ファイル数が 5%に満たなかったトークン」をトークンの型で置き換える．
3. 3.1.3 節と同じ方法で，トークン列を「トークンの三つ組」の列に変換する．

上記の処理によって入力文字列から得られる「トークンの三つ組」の列を，獲得した文法を基に CYK 法で構文解析する．

4 「生成規則の集合」の獲得と構文解析結果

本手法を用いて，実際に文法の学習を行なった．今回は Ruby 言語を対象としてプログラミング言語の文法の獲得を試みた．本章では，学習時に用いた各種パラメータ値と，獲得した「生成規則の集合」で構文解析をした結果の例を示す．

4.1 Ruby 言語を対象とした文法の獲得

学習データを集めるために，Ruby 言語のオープンソースソフトウェアとして GitHub 上に公開されているリポジトリから，スター数の多い順に 100 リポジトリ取得した．これらのリポジトリから，Ruby 言語で記述されたファイルのみを抜き出して学習データとして使用した．実際に学習に用いられた Ruby 言語のファイル数は 44,742 である．

3.1.1 〜 3.1.3 節で述べた手順によって，上記のデータセットの各ファイルの文字列を「トークンの三つ組」の列に変換した．ここで，字句解析器として Ruby 言語の標準ライブラリに含まれている Ripper クラスを用いた．Ripper によって与えられるトークンの型が *on_sp*，*on_nl*，*on_ignored_nl* または *on_comment* のいずれかであるようなトークンはスペース，改行，コメントを表すので，字句解析を行った段階でこれらを取り除いた．出現ファイル数が全体の 5%に満たないトークンをトークンの型で置き換えたところ，*class*，*require*，*describe*(RSpec[†2]

†2 Ruby 言語のテスト用フレームワーク．`http://rspec.info`

で多く用いられているようである) など，一般的に広く用いられているトークンは置換されずに残った．これらの処理によって得られるデータセット内の「トークンの三つ組」は 5,780 種類となった．

次に,「トークンの三つ組」に対して「左側の文脈のみを考慮したベクトル表現」と「右側の文脈のみを考慮したベクトル表現」を与えた．この二種類のベクトル表現のどちらを学習する場合にも，Skip-gram モデルのウィンドウサイズは 1 で epoch 数を 100 回として学習を行なった．また，どちらのベクトル表現も次元数は 100 次元として埋め込みをした．

「左側の文脈のみを考慮したベクトル表現」の距離でトークンの三つ組をクラスタリングする際，X-means 法のクラスタ数の初期値として「トークンの三つ組」の種類数の半数である 2,890 を指定した．その結果，実際に得られたクラスタ数も 2,890 であった．「右側の文脈のみを考慮したベクトル表現」の距離でトークンの三つ組をクラスタリングする場合も同様に 2,890 を指定して同じクラスタ数に分割された．

「トークンの三つ組」の種類数が 5,780 であったため,「生成規則の集合」の初期値も 5,780 個の生成規則が生成された．この「生成規則の集合」の更新時には，CYK 法の深さ (非終端記号が表す区間として考慮する最長の長さ) を 10 として学習を行った．更新回数は上限を 100 回として学習を行なったが，実際にはその時点での「生成規則の集合」で更新を行なった結果が同じ「生成規則の集合」となった時点で，それ以上は更新が行われないので処理を打ち切っている．

4.2 獲得した文法による構文解析

実際に獲得した文法でいくつかのソースコードを対象に構文解析を行った．現在，獲得した文法を解析し，本論文で提案した手法の改善を進めている．例えば，獲得した文法を用いて図 12 を構文解析すると成功する．これは，Ruby 言語としても正しい文法である．また，図 13 のソースコードを獲得した文法で構文解析すると，*end* が抜けているので構文解析に失敗する．しかし，図 14 のソースコードは Ruby 言語の文法としては正しくないが，今回獲得した文法では構文解析に成功してしまう．今後，手法の改善を継続的に行う必要がある．

```
class ClassName < SuperClassName
    def self.foo
        a = b + (c + d)
    end
end
```

図 12　獲得した文法での構文解析に成功する Ruby 言語の文法として正しいソースコード例

```
class ClassName < SuperClassName
    def self.foo
        a = b + (c + d)
end
```

図 13　獲得した文法での構文解析に失敗する Ruby 言語の文法として正しくないソースコード例

```
class ClassName < SuperClassName
    self.foo
        a = b + (c + d)
    end
end
```

図 14　獲得した文法での構文解析に成功する Ruby 言語の文法として正しくないソースコード例

5 まとめ

既存のソースコード群からプログラミング言語の文法を獲得するための手法について検討をした．本手法では，Skip-gram モデルによって獲得したベクトル表現を基にした「生成規則の集合」の初期値の決定と，CYK 法に基づく「生成規則の集合」の更新によって最終的な文法を獲得した．本論文で述べた文法獲得手法の改善を今後続けていく．

参 考 文 献

[1] Earley, J.: An Efficient Context-free Parsing Algorithm, *Commun. ACM*, Vol. 13, No. 2(1970),

pp. 94–102.

[2] Mikolov, T., Sutskever, I., Chen, K., Corrado, G., and Dean, J.: Distributed Representations of Words and Phrases and Their Compositionality, *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, USA, Curran Associates Inc., 2013, pp. 3111–3119.

[3] Pelleg, D. and Moore, A.: X-means: Extending K-means with Efficient Estimation of the Number of Clusters, *In Proceedings of the 17th International Conf. on Machine Learning*, Morgan Kaufmann, 2000, pp. 727–734.