解析表現文法によるプログラム中の欠損を許した構文解析とそのツールの提案

解析表現文法によるプログラム中の欠損を許した構文解析とそのツールの提案
A Proposal of a Parsing Method and its Tool that Treats Syntactically Incomplete Programs Using Parsing Expression Grammars

山能 佑介 *
Yūsuke Yamanō
法政大学大学院 情報科学研究科 情報科学専攻
Email: yusuke.yamano.3n@stu.hosei.ac.jp

***Abstract*—Visual programming, especially block-based programming, is used for beginners in programming. Particularly, there are systems that programmers freely convert between a block-based language and a general text-based language. However, there is a problem that the conversion can be done only when there is no syntax error in the text program. This is because programs with syntax error cannot be parsed and therefore cannot be converted into abstract syntax trees or other forms of representation. In order to solve this problem, we propose a tool that can automatically generate a grammar that allows a program with missing parts. The tool generates a parsing expression grammar that can parse such incomplete program by providing the syntax of the programming language. At the same time, the tool generates a template for manipulating concrete syntax trees that are created in the process of parsing with the generated parser, and supports conversion from concrete syntax trees. The template is that for a syntax-directed parsing program for the concrete syntax tree, which supports the conversion from the concrete syntax tree to the abstract syntax tree or other program representations such as blocks. As a result, it is easy to introduce a system that can convert between text-based languages and block-based languages, even if the program has some missing parts in the process of editing.**

1. はじめに

プログラミング初学者に対して用いられる学習環境として, ビジュアルプログラミング, 特にブロックを用いたビジュアルプログラミングがある. その中でも学習のためブロック型プログラミングと, 一般的なプログラミング言語によるテキスト型プログラミングの間で変換が行えるシステムも存在する [1].

このようなシステムでは, ブロック型言語とテキスト型言語の間で互いに変換が行われる. しかし, このような変換は構文に誤りがない場合にのみしか行えないという問題がある. これは図 1 のように, 構文誤りのあるテキスト型プログラムでは構文解析ができないため抽象構文木が作れず, テキスト型プログラムからブロック型プログラムへ変換できないためである. 一方で, プログラマは編集途中に, 欠けたままのブロック型プログラムのコードを変換することがあるが, この場合はテキスト部分が一部欠けたようなテキスト型プログラムを出力する変換を行うことが適当である. このようにして作られた, 欠けのあるテキスト型プログラムは普通の構文解析では処理できない. そこで, 先行研究 [2] では PEG [3] を利用し, 一部のコードが欠けたプログラムに対して構文解析が行えるよう, 欠損した箇所に仮のトークンを埋め込む「拡大文法」と呼ばれる手法を提案した. しかし先行研究では拡大文法を手作業で生成していたため様々なプログラミング言語に対応することが難しく, また仮のトークンを埋め込むことで構文誤りを解消するため, 厳密には構文解析しようとするもとのプログラムとは構文が異なるプログラムを解析することになる.

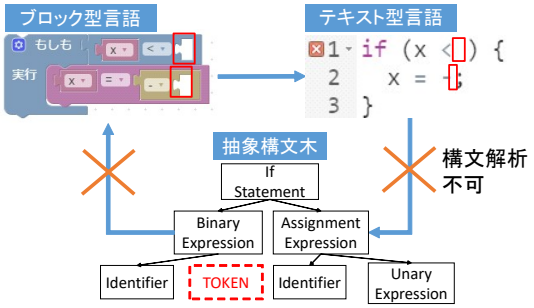

図 1. 構文誤りにより変換できないプログラム

これらを解決するため, 本研究では欠損のあるプログラムを許容する構文を自動生成するツールを提案する. 本ツールでは, プログラミング言語の構文を与えることで, 欠損のあるプログラムでも構文解析可能な解析表現文法を生成する. また同時に, 生成した構文での構文解析の過程で作られる具象構文木を走査するためのスタブを生成し, 具象構文木からの他の表現への変換を支援する. このスタブは, もとのプログラミング言語に与えられえる具象構文木に従う構文主導解析プログラムの雛形であり, 具象構文木から抽象構文木や他のプログラム表現 (ブロック等) への変換プログラムの記述を容易にする. その結果, 編集途中等で構文的に欠損のあるプログラムでも, テキスト型言語やブロック型言語の間で変換できるシステムを効率的に導入可能となる.

以下, 本論文では, 2 節ではまず本研究で用いる, 文法の一種である PEG と手法である拡大文法について説明する. 3 節では拡大文法の自動生成や変換のための構文木について説明する. 4 節では前節をもとに具体的な実装について述べる. 5 節では本ツールをいくつかの例で適用し結果について見る. 6 では前節までをもとに本ツールについて考察し, また関連研究について説明する.

2. 準備

2.1. Parsing Expression Grammar

Parsing Expression Grammar (解析表現文法, 以下 PEG) は, 文脈自由文法や正規文法などに代わる新たな形式文法である. PEG は, 文脈自由文法と異なり構文の定義から曖昧さを排除しており, 代わりに優先度付き選

*Supervisor: Prof. Akira Sasaki

択を導入している. このため, PEG はプログラミング言語など機械指向の構文記述を行いやすい.

構文規則について, 非終端記号を A, 解析表現を e とすると $A \leftarrow e$ と表現される. PEG の解析表現を e とすると, 任意の既存の解析表現を e_1, e_2, e_3 としたとき, e は次の解析表現のいずれかとなる.

$e ::=$	ϵ	空文字
	α	任意の終端記号
	A	任意の非終端記号
	$e_1 e_2$	連接式
	e_1 / e_2	優先度付き選択式
	$e*$	0 又は 1 以上の繰り返し式
	$!e$	否定先読み

連接式は e_1 を呼び出した後に e_2 を呼び出すことを表す. 優先度付き選択式は, 入力文字列が e_1 と一致するかまず試み, 一致しなかった場合は e_2 を呼び出すことを表す. この解析表現が PEG の大きな特徴のひとつであり, 文脈自由文法などではどちらか一方を選択する, という形である. 0 又は 1 以上の繰り返し式は, 0 又は 1 以上の個数 e が続くとして入力文字列を消費し続ける. PEG の繰り返しは貪欲であり一致し続ける限り入力を消費する. 否定先読みは e に一致しないときにのみ成功し, 入力文字列は消費しない.

これに加え, 次のような syntax sugar を定義している.

$. = a/b/c/...$	任意の終端記号
$e_1? = e_1/\epsilon$	省略可能式
$e+ = e(e*)$	1 以上の繰り返し式
$\&e = !(!e)$	肯定先読み
$[abc] = a/b/c$	文字クラス

また, PEG により定義した構文規則は再帰下降構文解析器で実装できる. このときバックトラック, つまり複数の選択肢を試み失敗の場合手戻りする可能性がある. このため, ナイーブな PEG の実装では入力される文字列の大きさに従い, 最悪の場合では構文解析に指数時間の計算量が必要となる. この問題は Packrat Parsing [4] の方法を利用することで回避できる. これは, 構文解析の結果をメモ化し, 線形時間で解析する手法である.

2.2. 拡大文法

我々の先行研究 [2] では, テキスト型言語とブロック型言語の変換の際に, プログラム中に文法的な欠損を含む場合でも構文解析可能にするための手法として, 拡大文法を提案した. ブロック型言語とテキスト型言語を併用する場合, テキスト型プログラムからブロック型プログラムへの変換を行う際にはプログラムに構文誤りがない状態でなければ変換はできない. またブロックが欠けたような状態では, ブロック型プログラムからテキスト型プログラムへの変換は可能だが構文解析の行えないプログラムへと変換してしまう.

先行研究ではこのような構文誤りを含むテキスト型プログラムでも構文解析できるよう, 対象のテキスト型言語の文法を拡大した文法を定義し構文誤りを修正できるようにした. これを拡大文法と呼ぶ.

拡大文法では各構文において一意に識別できるようなトークン列を「キーフレーズ」と定義し, それ以外のトークンを省略可能とした構文を用意し, 元の構文に統合する. if 文の推定を行う拡大文法の例を図 2 に示す.

```
/* 通常の構文 */
IfStatement
  = IfToken _ "(" Expression _ ")" _ Statement
/* 拡大文法適用後 */
IfStatement
  = IfToken _ "("? Expression? _ ")"? _ Statement?
/* 統合した構文 */
IfStatement
  = IfToken _ "(" Expression _ ")" _ Statement
  / IfToken _ "("? Expression? _ ")"? _ Statement?
```

図 2. if 文による拡大文法の例

この構文 `IfStatement` の例では, `IfToken` がキーフレーズとなる. よって拡大文法適用後の構文では, このキーフレーズ以外に省略可能を表す?の接尾辞を付与している. これにより, `Expression` など他の非終端記号がプログラム中で欠けている場合, それを読み飛ばして構文解析する. 統合後の構文には, 元の構文の後に優先度付き選択式を使用して拡大文法が付与されている.

先行研究では拡大文法を利用して構文解析し, 欠損している箇所に対して補完を行う. その補完したプログラムを構文解析し抽象構文木を得ることで, 欠損を含むようなテキスト型プログラムからブロック型プログラムへの変換を可能にした.

3. 提案手法

本研究では先行研究における拡大文法を一般化し, 汎用的に扱う手法を提案する. 先行研究のように欠損を含むようなプログラムに構文誤りを解消するようトークンを埋め込むのではなく, 内部的に構文木を生成することで, 汎用的な操作とプログラム間の双方向変換を維持することを狙いとする.

3.1. 提案手法の概要

テキスト型言語と他のプログラム表現 (ブロック型言語等) を併用するとき, テキスト型言語からプログラム表現への変換を行うにはテキストのプログラムに構文誤りがない状態でなければ行うことができないという問題がある. これは, テキスト型言語を構文解析し抽象構文木を構成し, その抽象構文木から他のプログラム表現に変換するためである. 先行研究ではこれを拡大文法と呼ばれる手法で解決した. しかし先行研究では, この拡大文法の生成は手作業で行うため, 様々なテキスト型言語を導入する場合困難である. また, 欠損のあるプログラムを拡大文法で解析した後の処理について, 欠損箇所に構文誤りを回避するような仮のトークンを埋める. そのため, 厳密には変換前と変換後のプログラムは異なる.

本研究では拡大文法の自動生成を行い, 欠損した箇所の情報は拡大文法で構文解析した際の構文木に含める.

まず, 拡大文法の生成の概要を説明する. テキスト型言語の構文規則を入力とし, 出力としてその構文の拡大文法と, この拡大文法で構文解析した際に生成される構文木の変換系を得る. そのために次のような手法を採る.

1) ある汎用言語の構文規則 G から, 対応する構文木 T への変換規則 $M\langle G, T\rangle$ と, 拡大文法を適用した構文規則 G' を生成する.
2) $M\langle G, T\rangle$ と G' から, 欠けを含む構文木 T' への変換規則 $M\langle G', T'\rangle$ を自動生成する.
3) $M\langle G', T'\rangle$ を処理する処理系を開発する.

まず, 特定のテキスト型言語の構文規則 G を用意する. 次に構文規則 G から, G で構文解析した際に生成され

る対応した構文木 T との変換規則 $M\langle G,T\rangle$ と，拡大文法を適用し欠損のあるプログラムを許容する構文規則 G' を得る．G' の欠損する可能性のある箇所の情報と，$M\langle G,T\rangle$ を利用することで，変換規則 $M\langle G',T'\rangle$ を生成する．最後に，この生成した $M\langle G',T'\rangle$ を処理する処理系を開発する．

説明した手法の概念図を図 3 に示す．矢印に振られた

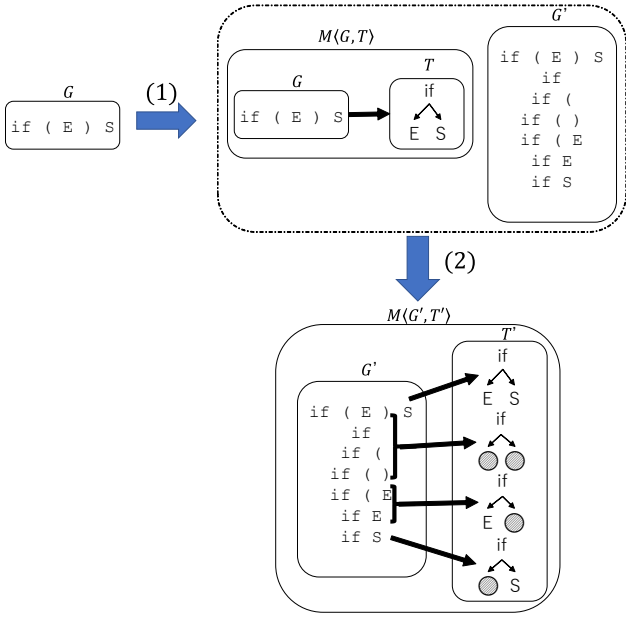

図 3. 拡大文法生成の概念図

番号は手法の項目と対応する．(3) については，本研究では既存の parser generator を用いるため，考慮しない．

テキスト型言語のプログラムと他のプログラム表現は図 4 のようになる．生成したテキスト型言語の拡大文

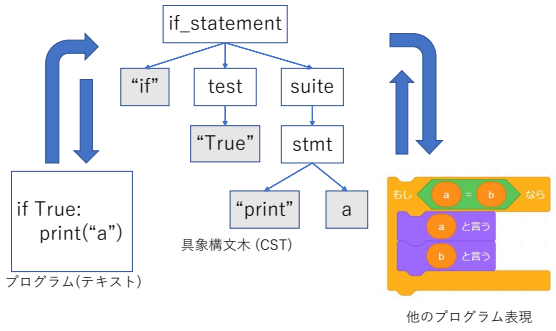

図 4. テキスト型言語から他のプログラム表現への変換

法を利用することで，欠損を含む CST を生成できる．またこの構文木を走査することで，もとのテキスト型プログラムへ復元可能である．ここで他のプログラム表現への変換について考える．テキスト型プログラムから構文木への変換は，これまで生成した変換規則により容易に可能となった．しかし他のプログラムへの変換は容易ではない．構文木から他のプログラム表現へ変換を行う変換規則を生成する必要がある．そこで，構文木から他のプログラム表現へ変換を容易にするため，構文木を走査するためのスタブを生成する．このスタブを他のプログラム表現を生成するよう変更を加えるようにすることで，他のプログラム表現への対応を容易にする．

次節から，拡大文法の生成や具象構文木，スタブについて説明する．以下では次の図 5 の構文の例を利用して説明する．簡単な if 文の例であり，if 文，式，文のみで構成される．なお _ は任意の個数のスペースとする．

```
IfStatement
  = "if" _ "(" _ Expression _ ")" _ Statement
Expression
  = "True" / "False"
Statement
  = "print"
```

図 5. PEG による if 文の例

3.2. 拡大文法の生成

先行研究では，既存の構文の受理する範囲を拡大し，テキストに欠損のあるような状態でも受理するような文法である「拡大文法」を提案した．先行研究ではテキスト型言語の文法から拡大文法を手作業で生成した．本研究ではこれを自動で生成する．

拡大文法の自動生成について説明する．入力としてプログラミング言語の PEG を入力することで，拡大文法に拡張した PEG を出力する．まず，記述された PEG を解析し構文の Abstract Syntax Tree (AST) を出力する PEG を用意する．

`IfStatement` から生成される AST の例は図 6 のようになる．`Rule` はそれぞれの構文規則を表し，この例

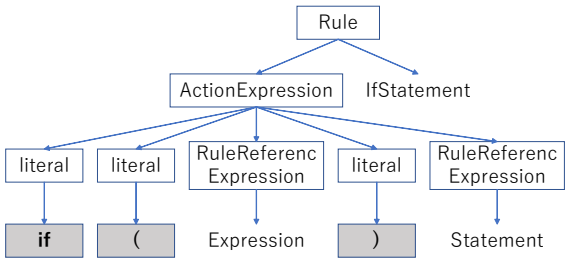

図 6. 与えた構文の AST の一部 (IfStatement)

では `IfStatement` のみを抜き出している．`literal` は文字列であり，`RuleReferenceExpression` は他の構文規則を参照する非終端記号である．この AST から，キーフレーズをまず設定する．拡大文法の定義より，キーフレーズは欠損のある構文でも必ず存在し，他の構文に設定したキーフレーズと一致しないものである．この構文では文字列 `of` がキーフレーズである．よって，それ以外の文字列`"("`や非終端記号 `Expression` などは省略可能となる．拡大文法にした `IfStatement` は図 7 のようになる．

```
IfStatement
  = "if" _ "("? _ Expression? _ ")"? _ Statement
```

図 7. 拡大文法を適用した IfStatement の例

また，非終端記号の中でも省略したくないものについて，事前にそのような構文を登録しておくことで明示的にキーフレーズとした．詳しくは実装で述べる．

これにより，構文的に欠損のあるプログラムでも構文解析可能な PEG を生成する．

3.3. 具象構文木

生成した拡大文法を用いることで，入力文字列に対する構文解析の過程で構文木を生成する．この構文木を走

査することで, プログラム中の欠損の情報を保持するとともに, またテキスト型プログラムや他のプログラム表現へと変換することを可能にする. 本ツールでは, 拡大文法に対する構文解析器は, そのまま構文解析の結果の表現である構文解析木, すなわち具象構文木 (Concrete Syntax Tree, CST) を生成する.

以下では, 図 8 のプログラムを例として CST の説明を行う. このプログラムを構文解析すると, 図 9 のようなCST となる. ここで, CST は S 式の形式で表現している. CST には各構文規則の名前や文字列が格納されている. このような CST を走査し, プログラムを変換していく. 次に, 拡大文法を利用した欠損のあるプログラムとその CST を見る. 欠損のあるプログラムを図 10, そのプログラムの CST を図 11 に示す. 図 10 のプログラムは, 編集途中などで閉じ括弧や文が欠損したプログラムである. 図 11 の欠損した箇所を見る. 具象構文木で, 本来何らかの構文の要素ある箇所が欠けた場合は欠けたという情報 `extend` を埋めている. それに加えて文字列の場合その文字列を, 構文規則の参照の場合その構文規則名を付与している. これにより欠損を含むプログラムでも, 具象構文木に欠損した箇所とその欠けた構文を保持することができる.

```
if (True) print
```

図 8. if 文のプログラム

```
(IfStatement
    ("if") ("(") (Expression "True") (")")
    (Statement "print"))
```

図 9. if 文のプログラムの CST

```
if (True
```

図 10. 欠損のある if 文のプログラム

```
(IfStatement
    ("if") ("(") (Expression "True")
    (extend (")")) (extend (Statement)))
```

図 11. 欠損のある if 文のプログラムの CST

3.4. 変換プログラムのためのスタブ

拡大文法に基づく構文解析の過程では, 具象構文木を生成する. この具象構文木を容易に扱うため, 拡大文法の生成と同時に, 具象構文木を走査するためのスタブを生成する. これは, 具象構文に対する構文主導解析プログラムの雛形であり, 具象構文木から抽象構文木や他のプログラム表現 (ブロック等) への変換を支援する.

If の言語の例において生成されるスタブの疑似コードを図 12 に示す. 生成したスタブでは, 再帰的に `walk` メソッドを実行することで構文木全体を走査する. `IfStatement` メソッド等はそれぞれ構文規則と対応している. このメソッドを編集することで, 他のプログラム表現への変換を行う処理を追加することができる.

また, 本ツールではこのスタブを, 元のテキスト型プログラム表現に戻すアンパーザ (unparser) のプログラムとして, 自動生成する. これにより, ツール使用者はこのプログラムを参考にして, 使用者の目的とするプログラム表現へ変換するコードへ改変することを期待する. 詳しくは実装で述べる.

```
class Stub {
  walk() { /* ... */ }
  IfStatemet() { /* ... */ }
  Expression() { /* ... */ }
  /* ... */
}
```

図 12. 生成されるスタブの例

4. 実装

4.1. 拡大文法を適用した構文生成のアルゴリズム

拡大文法生成の具体的なアルゴリズムや実装について説明する.

本ツールでは PEG.js [5] のサンプルに存在する PEG の文法をもとに, テキスト型言語の文法を構文解析する. PEG.js は JavaScript 向けの parser generator である. PEG による文法定義を行うことで, その構文解析器を生成する. サンプルの PEG の文法を利用し, 拡大文法にしたい文法を構文解析し AST を得る. 拡大文法を得るために, まずキーフレーズを収集する. キーフレーズは, 構文を一意に特定するためのトークン列である. よって他の構文内で設定したキーフレーズと一致しない, つまり重複のないように選択する必要があるため, 最初に AST を走査し収集する必要がある. 各構文規則について, それぞれ走査を行う. 最初に見つけた文字列をその構文の仮のキーフレーズとして登録する. 他の構文規則も走査し, 仮のキーフレーズに重なりがある場合, それらを仮のキーフレーズから除去する. そうして残った候補をキーフレーズと設定する.

二度目の走査で, キーフレーズではない記号に対して省略可能を表す`"?"`を付与する. これと同時に, 拡大文法を適用する記号を収集する. 構文規則の走査が終了した際に, 生成される拡大文法に, 収集した情報を利用して具象構文木を生成するための semantic action を付与する. これにより, 構文解析の過程で具象構文木を生成する拡大文法が生成される.

また, 非終端記号の中でも省略したくないものについて, 事前にそのような非終端記号を登録しておくことで明示的に省略しないとした. これは一見非終端記号であっても, `operator = "+" / "-"`のような, キーフレーズと差異のないものも含まれるためである.

4.2. 生成されるスタブ (unparser)

4.1 で述べた構文解析器の生成と同時にスタブを生成する. このスタブは構文解析の過程で生成される具象構文木を走査するためのものである. 本ツールではこのスタブを, 具象構文木から元の構文解析前のテキストへと戻す unparser の形で生成する. これにより, ツール使用者がこの unparser を参考にすることで他のプログラム表現への変更を用意にすることを期待する.

生成される unparser の例は図 13 のようになる. 生成されるスタブでは再帰的に具象構文木を走査し他のプログラム表現へと変換する. `walk` メソッドは次の構文木へと辿るためのメソッドである. 走査する構文はそれ

```
class Unparser {
  walk(cst) {
     if (cst == '_') return '_';
     if (typeof cst == 'string') return cst;
     const name = cst.name;
     const func = cst => this['_' + name](cst);
     return func(cst);
  }
  _if_statemet() {
    if (typeof cst.body[1] == 'object') {
      if ('extend' in cst.body[1])
        return cst.body[1] = '_';
    }
    /* ... */
    return ['if', this.walk(cst.body[1]), ... ].join();
   }
  /* ... */
}
```

図 13. unparser の例

ぞれメソッドと対応しており, walk メソッドは引数の CST の名前を利用しメソッドを実行している. 構文のメソッドでは, テキスト型プログラムを返すとともに, プログラムに欠損のある場合の処理をおこなっている. ここで欠損箇所に仮のトークンを埋め込むなどの処理をおこない, プログラムを変換可能にする.

5. 実験

実験では, 簡単なテキスト型言語からいくつかのプログラム表現へ変換できるかどうかを確認した. また, 生成されたスタブをどの程度の行数変更することで変換可能となったかを確認した. 簡単なテキスト型言語として図 14 の例を利用した.

```
if_statement = "if" __ test ":" __ suite
test = "True"
suite = "print"
__ = [ ]+
```

図 14. 実験で利用した構文

5.1. unparser

本ツールでは生成されるスタブは unparser の形をとる. そのためスタブを変更することなく, 具象構文木を読み込ませることでもとのテキスト型プログラムへと戻る.

以下に if 文の例に対していくつかの箇所を欠損させた結果を示す. 矢印左側が元となるテキスト型プログラムであり, 矢印右側が unparser により変換したプログラムである. 復元されたプログラムについて, 欠損している箇所には _ を埋め込んでいる.

1) `if True: print` ⟹ `if True: print`
2) `if` ⟹ `if ____`
3) `if True` ⟹ `if True___`
4) `if True:` ⟹ `if True: _`
5) `if :` ⟹ `if _:__`
6) `if : print` ⟹ `if _: print`
7) `if print` ⟹ `if ___print`
8) `if True print` ⟹ `if True_ print`

(1) は欠損のないプログラムであり, もとのプログラムと unparser が変換したプログラムが一致していることから, unparser が機能していることがわかる. この if 文の例ではキーフレーズは"if"であるため, もっとも欠損がある (2) でも構文解析し unparser による変換ができていることがわかる. (2) 以外の欠損を含むプログラムについても, 欠損箇所に正しく _ が埋め込まれ unparser による変換がなされている.

5.2. AST

生成されたスタブを編集し, 他のプログラム表現のひとつとして AST に変換する例を見る. 以下に if 文の例を AST に変換した結果を示す.

1) `if True: print` ⟹
   `(If (Bool (True)) (Print))`
2) `if` ⟹
   `(If (Missing test) (Missing suite))`
3) `if True` ⟹
   `(If (Bool (True)) (Missing suite))`
4) `if True:` ⟹
   `(If (Bool (True)) (Missing suite))`
5) `if :` ⟹
   `(If (Missing test) (Missing suite))`
6) `if : print` ⟹
   `(If (Missing test) (Print))`
7) `if print` ⟹
   `(If (Missing test) (Print))`
8) `if True print` ⟹
   `(If (Bool (True)) (Print))`

欠損のある箇所には, (Missing ラベル名) といった形で欠損の情報とラベル名を保持している. (1) から, まず完全な文のときに正しい AST が出力されていることがわかる. (4, 5, 6) から, 非終端記号が欠損した場合にも Missing で補いつつ変換できている. (3, 7, 8) より, 文字列":"が欠損した場合も構文解析は行えており, 期待した通り同一の AST が生成されている.

5.3. ブロック言語

ブロック言語に変換する例を見る. 本研究では JavaScript 向けブロック言語の Blockly [6] に対応する XML 表現に変換した. 入力したプログラムと出力された XML の例を図 15 に示す. プログラム 1 では if 文を

```
/* 入力するプログラム1 */
if
/* 出力されるブロック(XML)1 */
<xml xmlns="https://developers.google.com/blockly/xml">
  <block type="controls_if">
  </block>
</xml>

/* 入力するプログラム2 */
if True:
/* 出力されるブロック(XML)2 */
<xml xmlns="https://developers.google.com/blockly/xml">
  <block type="controls_if">
    <value name="IF0">
      <block type="logic_boolean">
        <field name="BOOL">TRUE</field>
      </block>
    </value>
  </block>
</xml>
```

図 15. 生成されるブロック (XML 形式) の例

表すブロック 1 個が出力されている. このような if 文のブロックのみでは出力されるテキスト型プログラムに構文誤りが発生するが, 拡大文法により構文解析が可能になっていることがわかる. プログラム 2 の出力では if 文ブロックに真偽値ブロックが接続されている. 要素 value

は真偽値ブロックが接続されているときのみ if 文ブロックに現れる要素である. このような要素が存在の有無も, 走査について記述せずともスタブに記述されたコードの欠損の確認を利用することで容易に実装できる.

5.4. 編集したスタブの行数

前節まででいくつかの例を用いて実験を行い, 拡大文法やスタブの動きを確認した. それぞれのスタブについて編集した行数を確認し, 他のプログラム表現への変換規則を作る際のスタブの影響を見る.

表 1. 編集したスタブの追加，削除，総行数

スタブの種類	追加行数	削除行数	総行数
unparser	0	0	73
AST	49	29	93
ブロック型言語	31	33	71

表 1 は実験で使用したスタブの編集した行数である. unparser は本ツールでは拡大文法生成時に出力されるため, 編集した行はない. AST, ブロック型言語の削除行数については, 同程度であったことがわかる. これはキーフレーズでない文字列 (":") や空白文字を走査するメソッドなど, この 2 例では利用しない類似した機能を削除したためである. AST の追加行数が多いことについて, AST を扱うためのクラスを新たに書き加えたことが原因である. AST を扱うためのクラスが 46 行のため, スタブを変更するために追加した行は 3 行となる. ブロック型言語において追加行数が多くなる要因は, 主に XML を生成するための文字列が追加行数の要因である.

6. 考察

6.1. 議論

本研究では, 文法を与えることで拡大文法の自動生成し, その文法に対する構文解析器から具象構文木を得るとともに, さらにそのような具象構文木を構文主導解析するためのスタブを生成するツールを開発した. 本研究の重要な点の一つは, 変換系を実装するにあたって, 開発者が用いるスタブが有用であるかである. 5 節より if 文のみで構成される文法について, 拡大文法が適用されそして構文解析が成功したときに具象構文木を生成するような文法を自動生成できるようになった. また, 5.4 節ではスタブの編集行数について確認した. 不要なコードを削除した上で半分以上のコードは利用されており, スタブが構文主導解析プログラムの雛形として効率化に寄与したといえる.

拡大文法の自動生成については, 本ツールの範囲では変換できない例がいくつかある. 先行研究でも述べられており, 例えば拡大文法にすることで空文字を受理してしまうような場合である. これは, 先行研究 [7] のようなより厳密なキーフレーズの定式化などを利用することで改善すると考えられる. また, 拡大文法の先行研究では, 拡大文法の適用は最大限行うというのが方針であった. これは, 構文中のキーフレーズ以外はすべて省略可能にするということである. 本研究では, あらかじめ指定することで意図的に省略しない構文規則を定義できるようにした. これにより, 構文要素によっては, あえて拡大文法を適用しないという方針をとることが可能となった.

6.2. 関連研究

伝統的な誤り訂正の方法として, フレーズレベル回復やエラーリカバリなどがある. これは, 非終端記号を読み替えて構文解析を継続したり, よく知られたエラーについて適切なエラーメッセージを生成し解析を続けたりをおこなう. 本研究では拡大文法を利用するため, そのような誤りを含めたまま構文解析を可能にするということが特徴である.

Pencil Code [8] はブロック型言語を用いて, 汎用言語である JavaScript や Cofeescript などを学習できるプログラミング環境である. クリックすることでエディタをブロックとテキストのもので切り替えることができる. ブロックに接続する箇所が欠けている場合, その箇所はテキストエディタでは__で表現され, 不完全なプログラムでも両者の変換が可能となっている. しかし, テキストからブロックへの変換は構文誤りがない状態でなければ変換することはできない.

7. おわりに

本研究では構文的に欠けのあるコードでも構文解析できるようにする, 拡大文法を自動で適用する手法とそのツールを提案した. プログラミング言語の文法を与えることで, 拡大文法を適用した文法を得ることができる. またこの文法から生成した構文解析器は構文解析とともに具象構文木を得る. さらに, この具象構文木を構文主導解析するためのスタブを生成する.

本ツールを, if 文のみで構成される文法に適用し, 拡大文法を生成した. このときに生成されるスタブを編集し, いくつかのプログラム表現への変換系を作成した. 作成した変換系から, 本ツールで生成されるスタブの有用性を確認した.

今後の課題として, より厳密に拡大文法について考える必要がある. 本ツールの範囲では拡大文法に失敗する構文が存在する. このため, 拡大文法の定義の定義の見直しや定式化などにより, より高い精度の自動変換が行えることが望まれる.

参考文献

[1] M. Homer, and J. Noble, “Combining tiled and textual views of code,” in *2014 Second IEEE Working Conference on Software Visualization*, 2014, pp. 1–10.

[2] 山梨裕矢, 佐々木晃, “構文誤りを含むプログラムのブロック言語表現への変換手法”, 情報処理学会論文誌プログラミング（PRO）, vol. 12, no. 3, pp. 6–6, Sep. 2019.

[3] B. Ford, “Parsing expression grammars: A recognition-based syntactic foundation,” *SIGPLAN Not.*, vol. 39, no. 1, p. 111122, Jan. 2004.

[4] B. Ford, “Packrat parsing: Simple, powerful, lazy, linear time, functional pearl,” *SIGPLAN Not.*, vol. 37, no. 9, p. 3647, Sep. 2002.

[5] R. Futago-za, “Peg.js parser generator for javascript,” https://pegjs.org/.

[6] F. Neil, N. Quynh, S. Ellen, and F. Mark, “Blockly — google developers,” https://developers.google.com/blockly.

[7] 佐藤開智, 佐々木晃, “解析表現文法による不完全なプログラムの構文解析手法の提案”, 情報処理学会論文誌プログラミング（PRO）, vol. 13, no. 3, pp. 16–16, June 2020.

[8] D. Bau, D. A. Bau, M. Dawson, and C. S. Pickens, “Pencil code: Block code for a text world,” in *Proceedings of the 14th International Conference on Interaction Design and Children*, ser. IDC ’15. New York, NY, USA: Association for Computing Machinery, 2015, p. 445448.