初学者の学習支援のための Python と Ruby 間の翻訳

2011SE286 渡邉 将匡 2014SE003 赤羽 里帆 2014SE020 廣瀬 隼大
指導教員：横山 哲郎

1 はじめに

今日の我が国では，IoT, 第四次産業，AI, ビッグデータといった言葉をよく耳にするようになり，IT 業界について，社会に広く浸透してきている．それに伴い，かつてよりもプログラミング言語が一般の人々にも認知されている．また，産業界での大型の IT 関連投資が続いていることや，昨今の情報セキュリティに対するニーズの増大を契機に，IT 人材の不足が課題となっている．しかし，我が国の労働人口は減少が見込まれており，今後 IT 需要が拡大する一方で，IT 人材不足はより一層深刻化する可能性がある [1]．このことから日本政府は今後十分な IT 人材を確保する為，プログラミング言語の義務教育化政策など，国の成長戦略に IT に関わる事項を盛り込んでおり [2]，今後多くの人がプログラミング言語を学習することが必要となる．

今日では，多くのプログラミング言語が使われている．そのため，開発者が使いたいコードを Web で発見したとしても，それらが使いたい言語で利用できるとは限らない．もしも，別の言語で書かれたコードを，自分の使いたい言語で利用する場合，何らかの手段で言語を翻訳する必要がある．この時，コードの完全な変換ができるツールが存在すればよいが，ツールで翻訳ができない部分がコード中にある場合は手動で翻訳しなければならない．その場合では，対象のプログラミング言語を新しく，ある程度習得する必要がある．しかし，新しいプログラミング言語を学習するにあたって，学習時間や学習費用，学習する上でかかる労力等の多くのコストがかかる．プログラミング言語を学習したいと考えている人の中には, その学習時間を十分に確保できない人たちもいる．そこで我々は，初学者の学習にかかるコストを削減することによって，初学者の学習支援をするソフトウェアを作成する．

我々の研究では，構文が似ている 2 言語を使用し，習得済みの言語からもう一方へと翻訳することによって，構文の学習を支援する．今回の研究で用いる言語は Python と Ruby であり，ここでの初学者は Python が習得済みであることを想定している．Python は世界標準 MIT 教科書でプログラミングのイントロダクションに使われており，プログラミング言語使用率もとても高い [5] ことから，Python を習得している者は多いと考えられる．また, Ruby は Python と似ている構文を持っており, 軽量スクリプト言語の中でかなり使用率も高いが，Python よりは低い citetiobe ため，Python を既に習得しつつ，Ruby を習得していない者は多いと考えらる．また，Python と Ruby は基本的な概念や使用用途において非常に類似している．しかし，様々な差異が存在するため [4]，そのようなコードを翻訳し，比較することで初学者の効率的な学習に繋がると考え，今回はこの 2 言語を採択した．

本稿で開発する翻訳ソフトウェアは，両言語の既存の標準ライブラリを使用して構文木へと変換を行い，その構文木を翻訳する．得られた構文木は，プリティプリンタを用いてコードへと変換することができる．そのため，プリティプリンタを組み合わせることで学習支援を行うことができる．また，我々の翻訳の範囲は，南山大学のプログラミング基礎の授業の範囲である．なぜなら，合格基準の 1 つに，高水準プログラミング言語の基本的な文法とその意味を理解している [6] があるため，初学者がプログラミング言語の構文を習得する範囲としては十分だと考えた．

事前に動きがわかっているコードを翻訳し，翻訳前と翻訳後のコードを比較することによって，初学者がコードの意味を理解しやすくなることが期待される．また, Python は WEB 系の言語であり，ネット上に多くのコードが存在するため，本稿の範囲内であれば，翻訳ソフトウェアを使用することにより，初学者にとって効率的な学習に繋げることが出来る．また，2 つのコードを比較して学習することにより，その差異で構文を学習できるため，効率が上がる．例えば，if 構文における Python の `elif` と Ruby の `elsif` の差や，二項演算における Python の/と//の除算演算子と Ruby の/の除算演算子の差がわかるため，該当部分以外が同じであれば，その差異だけを学習すれば構文が学習できる．

2 関連研究

ここでは，本ソフトウェアに関連する技術や研究を記述する．

2.1 字句解析

字句解析とは，OS の読み込みによって主記憶装置に読み込まれたソースコードの文字列を 1 字 1 字調べながら，ソースコードを字句に分割する工程である．ここで字句とは，プログラムとして意味を持つ，最小単位の文字列のことである．入力は文字列となり，出力は字句の列となる．

2.2 構文解析

構文解析とは，ソースコードが文法的に正しいかどうかチェックする工程である．字句解析によって得られた字句の列が入力となり，構文木が出力となる．ここで構文木とは，字句を要素とする木構造のデータである．構文解析を行う手法はいくつか存在するが, 本稿のソフトウェアでは, 構文解析手法の一つである LALR 構文解析を用いて構文解析を行っている．

3 設計

この章では，本研究におけるソフトウェアの設計について記述する．なお，本ソフトウェアの要求は学習支援であり，翻訳を行うことではない．また我々は，本ソフトウェアを実装する際，構造化開発を用いた．なぜならば，本ソフトウェアの開発が小規模な開発だと判断したためである．また，本ソフトウェアを，オブジェクト指向における 1 つのオブジェクトだと判断したため，機能に着目して開発を行う構造化開発が適していると判断した．

3.1 本ソフトウェアの概要

本ソフトウェアは，具象構文が出来るだけ似た形になるように，プログラミング言語の翻訳を行うソフトウェアである．なぜならば 1 章より，比較学習のために，ソースコードとターゲットコードが似た形であることが好ましいためである．そのため，我々は翻訳規則を一般化する際に，左辺と右辺の終端記号の差が最小になるように一般化し，ソースコードの意味を表示的意味論的に保存するだけではなく，自然言語的にも保存するよう心がけた．本研究では翻訳する言語として Python と Ruby を使用する．そこで，これらの言語の翻訳範囲は南山大学のプログラミング基礎の授業の範囲とした．その範囲を EBNF 記法を用いて制定した．制定した翻訳の範囲を図 3.1 に示す．

3.2 翻訳の例

本稿における，Python と Ruby の構文の差となる部分の翻訳の例を，実際のコードを使用して図 3.2 に示す．本稿では，If 文を例として翻訳について説明する．

if 文の構文の差として，条件式の後ろの:と `then` の差，`elif` と `elsif` の差，`end` の有無が挙げられる．これは，Python と Ruby で，条件式と中身の文を区切る字句が異なっており，C 言語での `else if` を表す字句が異なっており，構文の終わりを示す方法が異なっていることを示している．

次に，二項演算の差として，除算演算子の種類数の差が挙げられる．Python では/と//の除算演算子が存在し，/は商を浮動小数点型で返す演算であり，//は商の小数点以下を切り捨てた結果を返す演算である．Ruby の/演算子は，項が 2 つとも整数型である場合は商を整数型で返し，どちらか一方に浮動小数点型が含まれていた場合は商を浮動小数点型で返す．よって，/演算子は項のどちらかを浮動小数点型にすればよく，//演算子は，Ruby の/演算子の商の小数点以下を切り捨てればよい．そのため，翻訳結果として `to_f` メソッドと `floor` メソッドが付与される．

最後に，比較演算の差として，Python は数学に近い形で記述できるが，Ruby はそのように記述できないという差が挙げられる．そのため，図 3.2 のコードを翻訳した結果として，比較演算を，比較演算子の左右の項だけの比較演算に分割し，それらを `and` 演算子で論理結合した式が現れている．

$single\_input ::= simple\_stmt \mid compound\_stmt\ NEWLINE$
$stmt ::= simple\_stmt \mid compound\_stmt$
$simple\_stmt ::= small\_stmt\ NEWLINE$
$small\_stmt ::= (expr\_stmt \mid flow\_stmt)$
$expr\_stmt ::= test\,(augassign\,test \mid {}' =' test)$
$augassign ::= ('+=' \mid '-=')$
$flow\_stmt ::= break\_stmt \mid return\_stmt$
$break\_stmt ::= '\texttt{break}'$
$return\_stmt ::= '\texttt{return}'[testlist]$
$testlist ::= test\,(',' test) * [',']$
$compound\_stmt ::= if\_stmt \mid while\_stmt \mid$
$for\_stmt \mid funcdef$
$funcdef ::= '\texttt{def}'\ NAME\ parameters\ ':'\ suite$
$parameters ::= '(' [typedargslist] ')'$
$typedargslist ::= (tfpdef\ (',' tfpdef) *$
$tfpdef ::= NAME$
$while\_stmt ::= '\texttt{while}'\ test\ ':'\ suite$
$for\_stmt ::= '\texttt{for}'\ atom\ '\texttt{in}'\ test\ ':'\ suite$
$if\_stmt ::= '\texttt{if}'\ test\ ':'\ suite$
$('\texttt{elif}'\ test\ ':'\ suite) * ['\texttt{else}'\ ':'\ suite]$
$test ::= and\_test\,('\texttt{or}'\ and\_test) *$
$and\_test ::= not\_test\,('\texttt{and}'\ not\_test) *$
$not\_test ::= '\texttt{not}' not\_test \mid comparison$
$comparison ::= expr\,(comp\_op\,expr) *$
$comp\_op ::= '<' \mid '>' \mid '==' \mid '>=' \mid '<=' \mid$
$'!='$
$expr ::= term\,(('+' \mid '-') term) *$
$term ::= factor(('*' \mid '/' \mid '//') factor) *$
$factor ::= ('+' \mid '-') factor \mid power$
$power ::= atom\_expr\,['**' factor]$
$atom\_expr ::= atom\,trailer *$
$atom ::= '[' [testlist] ']' \mid NAME \mid NUMBER \mid$
$STRING+$
$trailer ::= '(' [arglist] ')'$
$arglist ::= argument(',' argument) *$
$argument ::= test\,['=' test]$
$suite ::= simple\_stmt \mid NEWLINE\ INDENT\ stmt+$
$DEDENT$

図 3.1: Python の翻訳範囲

我々は図 3.2 の翻訳例などから，翻訳の規則を一般化し，再帰降下法によって制定した．以下に，構文の差となる部分の一般化した翻訳規則を記述する．

3.3 構文木

Python と Ruby は似ている構文を持つが，構文木はデータ構造から異なっている．Python の ast モジュールによって定義される構文木は，クラスによるデータ構造を持つが，Ruby の Rippre クラスによって定義される構文木では，データ構造は S 式で表現されている．そのために，Ruby の構文木はリスト型のデータ構造となる．さらに，ast モジュールによる Python の構文木と Rippre モジュールによる Ruby の構文木は，括弧を表すノードの有無や，図 3.4 に示すような，構文によるノードの種類数などの差が存在する．

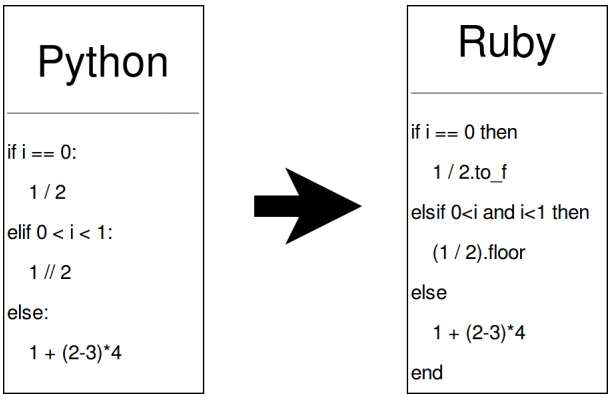

図 3.2: コードの翻訳

$$\begin{array}{l}\mathcal{T}[\![\texttt{'if'}\ test_1 \\ \texttt{':'}\ suite_1 \\ (\ \texttt{'elif'}\ test_2 \\ \texttt{':'}\ suite_2\ )* \\ [\ \texttt{'else'}\ suite_3]\!]^{\texttt{Python}}\end{array} = \begin{array}{l}\texttt{'if'}\ \mathcal{T}[\![test_1]\!]^{\texttt{Python}} \\ \texttt{'then'}\ \mathcal{T}[\![suite_1]\!]^{\texttt{Python}} \\ (\ \texttt{'elsif'}\ \mathcal{T}[\![test_2]\!]^{\texttt{Python}} \\ \texttt{'then'}\ \mathcal{T}[\![suite_2]\!]^{\texttt{Python}}\ )* \\ [\ \texttt{'else'}\ \mathcal{T}[\![suite_3]\!]^{\texttt{Python}}\ ] \\ \texttt{'end'}\end{array}$$

図 3.3: if 文の翻訳規則

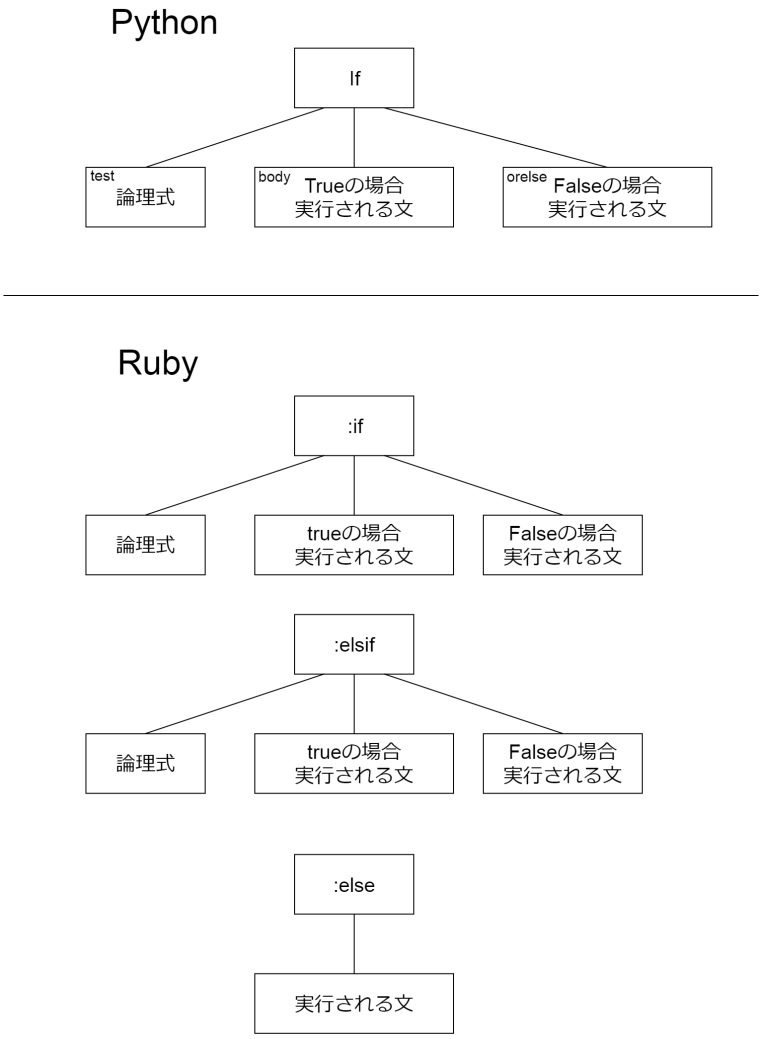

図 3.4: if 構文木

4 実装

ここでは，3 章で説明した構文木と具象構文の翻訳の実装について記述する．

4.1 if 文の翻訳

if 文の構文木を翻訳するためのコードは図 4.1 となり，関数 `func_if` が制御モジュールの実装となる．入力は，翻訳前の構文木の `If` ノードを表す `ast.If` 型の `i_node` と，関数 `sp_func_if` から直接呼び出されたかどうかを表す `boolean` 型の `sp_flag` となる．`sp_flag` には初期値として `False` が入力されており，通常の呼出しでは `i_node` のみに実引数が代入されるように実装されている．`sp_flag` が `True` であれば `elsif` ノードに，`False` であれば `if` ノードに翻訳される．これは，Python に Ruby の `elsif` に相当するノードが存在しないため，Python の `elif` 句が必ず `if` 句の下に来ることを利用して翻訳しているためである．

図 4.1 の関数 `trans_if` は変換モジュールの実装となる．入力は，関数 `func_if` の変数 `i_node` と，翻訳後の構文木を表すリスト型の `t_tree` である．Python では + 演算子でリストを連結することができるので，+=演算子で `If` ノードの子ノードを翻訳した結果を連結している．ただし，子ノードを翻訳した結果が `t_tree` の要素 1 つにならなければいけないため，子ノードを翻訳した結果を要素に持つリストを一時的に作成し，`t_tree` に連結している．

子ノードのデータは，それぞれ `i_node.test`, `i_node.body`, `i_node.orelse` に保存されている．`i_node.test` には条件式が，`i_node.body` には条件式が真だった場合に実行される文が保存されているため，`i_node.test` は式を翻訳する `func_expr` で，`i_node.body` は文を翻訳する `func_sentence` で翻訳している．`i_node.orelse` には，条件式が偽だった場合に実行される文が保存されているが，Ruby の `elsif` ノードとなる可能性がある `If` ノードが保存されている場合もあるため，どちらであるか判断しつつ翻訳を行う `sp_func_if` を呼び出している．

図 4.1 の関数 `sp_func_if` は下位モジュールの実装となる．入力は関数 `trans_if` の `i_node.orelse` を表すリスト型の `else_body` である．`else_body` が `elif` 句であった場合，`else_body` の要素は 1 つであり，かつ `else_body` の要素の型は `ast.If` 型になる．それらの条件が真だった場合は実引数を `else_body[0]`，`True` として `func_if` を呼び出すことで，`else_body` を `elsif` ノードに翻訳する．条件式を `and` 演算子で論理結合しない理由は，`else_body` の要素数が 0 だった場合，存在しない要素を呼び出そう

としてエラーが発生し，その対処のためにコードが冗長になってしまうためである．条件式が偽であった場合，else_body は else 句の文となる．しかし，else 句が記述されていなかった場合，else ノードを生成してはならない．そのため，else_body の要素数が 0 かどうかで else ノードを生成しないかどうかを決定している．else ノードを生成しない場合は，子ノードの要素が空であることを示す None を，else ノードを生成する場合は，else_body を翻訳した結果を要素に持つ else ノードがそれぞれ生成される．

```
1  def func_if(i_node, sp_flag=False):
2      t_tree = []
3      if sp_flag:
4          t_tree = [':elsif']
5      else:
6          t_tree = [':if']
7      return trans_if(i_node, t_tree)
8
9
10 def trans_if(i_node, t_tree):
11     t_tree += [func_expr(i_node.test)]
12     t_tree += [func_sentence(i_node.body)]
13     t_tree += [sp_func_if(i_node.orelse)]
14     return t_tree
15
16
17 def sp_func_if(else_body):
18     t_tree = []
19     if len(else_body) == 1:
20         if isinstance(else_body[0], ast.If):
21             t_tree += [func_if(else_body[0],
                   True)]
22         else:
23             t_tree += [':else', [func_sentence(
                   else_body)]]
24     elif len(else_body) == 0:
25         t_tree = None
26     else:
27         t_tree += [':else', [func_sentence(
               else_body)]]
28     return t_tree
```

図 4.1: if 文の構文木の翻訳

5 おわりに

3 章の設計により, 学習を支援するために, 南山大学のプログラミング基礎の範囲の Python のコードの構文木を，似た形で Ruby の構文木へと翻訳するソフトウェアを実装した．また，本ソフトウェアとプリティプリンタを組み合わせることで，比較学習のためのコードを瞬時に入手することができ，Web 上でのコードも図 3.1 の範囲内，すなわち南山大学のプログラミング基礎の範囲内であれば，翻訳を行うことが可能であるため，Ruby のコードを探す場合にかかる時間的コストとユーザーにかかる労力を削減することが可能となる．また，その範囲内であれば，コードの質は一定に保たれる．3.2 節の翻訳規則より，2 つのコードの差は比較的少なく，構文を全て覚えるよりも少ない情報量を覚えるだけで済む．さらに，南山大学のプログラミング基礎の合格基準 [6] より，本ソフトウェアの翻訳の範囲を学習すれば，基礎的な構文は学習できる．

参考文献

[1] 経 済 産 業 省：News Release, http://www.meti.go.jp/press/2016/06/20160610002/20160610002.pdf. (2016).

[2] 内 閣 官 房：成 長 戦 略（ 素 案 ）, http://www.kantei.go.jp/jp/singi/keizaisaisei/skkkaigi/dai11/siryou1-1.pdf. (2013).

[3] Cass, S.: The 2017 Top Programming Languages - IEEE Spectrum, IEEE Spectrum: Technology, Engineering, and Science News, available from 〈https://spectrum.ieee.org/computing/software/the-2017-top-programming-languages〉(accessed 2017-10-4).

[4] 清水崇之, 小川翔二郎, 松原俊一, Duerst, M.: Python から Ruby へとプログラムの 自動変換を図るシステムの構築，情報処理学会全国大会講演論文集，Vol.73, No.1, pp1333-1334 (2011).

[5] TIOBE: TIOBE Index |TIOBE - The Software Quality Company, TIOBE - The Software Quality Company, available from 〈https://www.tiobe.com/tiobe-index/〉(accessed 2017-11-20).

[6] 南山大学：公開用 HTML，南山大学シラバスデータベースシステム，入手先 〈https://porta.nanzan-u.ac.jp/syllabus/html/2017_40003099.html〉（ 参 照 2017-11-20）.

[7] 中田育男：コンパイラ, 産業図書 (1981)