Vol.2014-NL-216 No.22
Vol.2014-SLP-101 No.22
2014/5/23

自動プログラミングへ向けた問題解答コーパスの収集と考察

小田 悠介[1,a)]　ニュービッグ グラム[1,b)]　サクティ サクリアニ[1,c)]　戸田 智基[1,d)]　中村 哲[1,e)]

概要：プログラミングは人手による作業が主であり, 最終的なソースコードそのものを生成する自動プログラミングシステムは実用化されていない. このような自動プログラミングシステムを学習, 評価するためには, 実際のプログラミングに関するタスクを切り出したコーパスの収集が必要である. 我々はタスクとして「自然言語による仕様文が与えられたとき, その仕様に対応する関数を出力する」という問題を考え, 仕様文・ソースコード例からなるパラレルコーパスを人手により収集した. また, 得られたコーパスから自動プログラミングのために必要な言語処理の技術を考察した.

1. はじめに

IT 産業の拡大により, 様々な場面でコンピュータプログラムが用いられるようになった. プログラムは定型的な処理をコンピュータにより自動化できる優れた手法である一方で, プログラム自体の開発は人の手に委ねられているため, これを開発するための人的資源や開発時間を確保する必要がある. 本来コンピュータプログラムとは, 要求される仕様をコンピュータが解釈実行可能なデータ構造として再現したものであり, 仕様が適切に記述された文書が与えられれば, その情報を用いてソースコード, あるいはプログラム自体の自動的な構築が可能であると考えられる. このような処理をここでは**自動プログラミング**と呼ぶ. 自動プログラミングシステムが実現されれば, 従来の人手によるプログラム開発の一部を機械的な処理に置き換えることができ, 開発に必要な資源や費用を直接削減することにつながる.

仕様の面からプログラムを直接記述する方法としては, 仕様の記述に特化した言語や枠組みを用いて仕様に相当する文書を作成し, これを変換することでプログラムを得る方法がある. しかし, これらの手法を用いて仕様を記述しようとする者は本来の目的とは別に仕様の記述方法に関する知識を習得しなければならず, 専門的な技術が必要であることに変わりはない. このため本稿では, 多くの者に追加学習が不要であると考えられる自然言語によって仕様が記述された文書を入力とし, この文書に対応するプログラミング言語の構文木ないしソースコードを出力するシステムを対象とする.

自然言語を対象とする以上, システムを学習, 評価するためにはデータが必要となる. 本稿ではこのようなデータの収集に取り組む. 具体的には, Web 上の数学に関する問題集を解く際に書かれたソースコードを対象とし, これにソースコード内に書かれた処理の説明をコメントとして追加したものを問題解答コーパスとして収集する.

また, 本稿では自動プログラミングを文字列から木構造への翻訳タスク (string-to-tree 翻訳) の一種として捉えた場合, どのような既存手法が適用可能か, 通常の翻訳に対してどのような問題が新たに発生するのか, 問題を解決するために必要な技術は何か, などを収集されたコーパス内のデータを用いて考察した.

2. 関連技術

自動プログラミング, または**ソースコード生成**と呼ばれる技術は古くから研究されており, 研究や実用例は数多く報告されている.

最も限定された例としては, C++言語のテンプレートなどに代表される**生成的プログラミング**と呼ばれるもので, プログラミング言語自体の言語仕様の一部としてソースコードに相当するデータを生成する仕組みを導入したものである. これを用いれば同様の構造を持つソースコードを単一のモデルから生成することができるため, 人手により記述されるソースコードの量は減少する. 一方で, 技術者に対してプログラミング言語や生成的プログラミングに関する知識を要求する点では, 生成的プログラミングは通常のプログラミングの延長線上にあり, 本質的には同様のも

[1] 奈良先端科学技術大学院大学
Nara Institute of Science and Technology
[a)] oda.yusuke.on9@is.naist.jp
[b)] neubig@is.naist.jp
[c)] ssakti@is.naist.jp
[d)] tomoki@is.naist.jp
[e)] s-nakamura@is.naist.jp

© 2014 Information Processing Society of Japan

のであると考えられる.

特定のプログラミング言語を対象とせず, プログラムの動作仕様のみを記述する場合, Z, LOTOS, UML などのプログラムの仕様記述のみに特化した**仕様記述言語**を用いることが考えられる. この場合, 仕様記述言語による仕様書を機械的な変換によってソースコードやプログラムにする手法 [1], [2] が考えられる. しかし, 仕様を記述するために仕様記述言語自体の言語仕様を技術者が理解していなければならず, 使用する言語や方法論に関する専門的な知識を持たない者には作業が難しい.

これら仕様の記述に特定の技術を必要とする手法に対し, 自然言語によって書かれた文章から対応するソースコードを推定する手法が考えられる. この場合は仕様記述言語などとは異なり, 入力されるデータが直接プログラムとして実装されるべき仕様にどのように結びついているかが明確ではない. このため, 自然言語の文から仕様に関する表現を推定, 抽出する処理が必要となる. 先行研究では, 組み合わせ範疇文法 (Combinatory Categorical Grammar: CCG) [3] と談話表示構造 (Discourse Representation Structure: DRS) [4] の組み合わせにより文から仕様に相当する式表現を抽出し, 得られた式を変換することによって Python 言語のソースコードを生成する手法を提案している. [5] ただしこの手法では最終的な式の生成を用意された辞書を用いた変換に頼っているため, システム自体が仕様書とソースコードの間の対応関係を直接学習しているわけではない. また, 複数の文の間の照応関係の推定精度や文自体の曖昧性によってソースコードをうまく生成できない場合があると述べられており, 自動プログラミングの枠組みでも自然言語における意味解析が課題となっていると言える.

自動プログラミングと逆の処理を行う技術として**自動コメント付与**が挙げられる. これはソースコードを解析し, 文ごとなどの単位で処理に対応する適切なコメントを追加する処理である. 例えば, 関数名など処理内容を直接表していると考えられるソースコード上の意味的な情報を参考にしながら, ソースコードの構文構造に対応するコメント文を生成する手法が提案されている. [6] 自動コメント付与の場合, 入力となるソースコードは形式言語であり, 字句解析, 構文解析, 意味解析の各段階では完全に一意な解析結果を得ることができる. このため, ソースコードの解析結果から定型的な処理によってコメントを生成する手法でもあまり問題にならない. これに対し自然言語を入力とした自動プログラミングの場合, 入力側に様々な曖昧性やノイズが含まれるため, 解析により一意な結果を得ることは難しい. このため, 自動コメント付与と同様の枠組みを直接適用することはできない.

3. 自動プログラミングにおける入出力の関係

本節では, 本稿における自動プログラミングの入出力の関係について説明する.

まず入力データとして想定するのは, 先行研究 [5] と同様に人手で書かれた自然言語による文とする. 自然言語からプログラムを生成するという問題設定は, 使用者に特定のプログラミング言語などに関する知識を要求することなくシステムの利用が可能になるという点で重要である. これ以降, 入力データの文を**仕様書**と呼ぶこととする.

システムから出力されるデータは実用的には特定の言語で記述されたソースコード, または実際に動作可能なプログラムの形をとることになるが, 自動プログラミングの手法を考える上ではプログラミング言語の構文木を出力と考えればよい. これはプログラミング言語の場合, 構文木に対応するソースコードをほぼ一意に求めることができるため, 構文木さえ得ることができればソースコードへの変換は容易に実現できるためである.

入出力間の制約として, 使用者が出力のプログラムに要求する仕様はすべて仕様書に含まれているものとし, 仕様書を解釈しても現れることのない暗黙的な仕様は存在しないものとする. 例えば「与えられた 100 個の整数のうち大きいものを 10 個選択する」という問題の場合, 使用者がプログラムの出力としてソートされた結果を望んでいたとしても, 仕様書にソートに関する事柄が含まれていないため出力側で勝手に付加するような処理はしない, ということである. 逆に, 仕様書に現れる仕様はすべて出力のプログラム中で実装されている必要があるものとする.

4. 問題解答コーパスの収集

モデルの学習および評価は実際に人の手で書かれた仕様書とソースコードに対して行うのが望ましい. これを行うため, 我々は Web 上に公開されているプログラミングの問題集に対する回答のソースコードを人手により作成し, これに日本語の説明文を追加したデータをコーパスとして収集している.

4.1 対象とするプログラミング言語

収集するソースコードを記述する言語は原理的には何を選択しても問題ないが, 今回我々は Python 言語を採用した. これは先行研究 [5] との比較が可能であることも挙げられるが, それ以外にも以下に述べるような理由がある.

- 一般的によく用いられる手続き型の構造を持つ言語である.
- 比較的容易に手に入る資料が多い.
- 仕様を記述するための仕組みや構文解析のためのライブラリを標準で備えている.
- 図示するときに分かりやすい.

4.2 収集方法

実際の作業として, 3 名の学生に Web 上の数学に関する

© 2014 Information Processing Society of Japan

問題集である Project Euler [*1] 上に公開されている問題を実際に Python 言語を使用して解かせ, 完成したソースコードにコメントとして説明を書かせた. ソースコードの作成にあたっては, それぞれの処理の記述に一貫性を持たせるために次の制約を設けることとした.

- 一つの問題に対して, その問題を解く関数を作成する.
- 関数は return のみで制御を抜けることとし, yield や例外送出は使用しないものとする.
- 標準関数と math ライブラリ内の関数以外は使用しないものとする.

コメントは Python 言語の Docstring 記法に従い, 関数ごとの処理の説明を関数の最初の文として書かせた. また, 関数内に出現する変数のうち関数の引数であるものは関数自体の外観の一部を構成すると考えられるので, コメント中に必ず全ての引数名を含ませることとした.

Project Euler に公開されている問題は数百問であり, 以上によって得られる関数と仕様書の対も同程度の量にしかならない. このため, 解いた問題に対してその問題の解法に含まれる部分問題を考え, 部分問題を解くコードも同様に作成するように指示した. 例えば「1000 未満の自然数で, 3 の倍数となるものの和を求める」という問題があった場合,「1000 未満の自然数の集合を求める」「ある自然数 x が 3 の倍数であるかどうか調べる」「与えられた自然数の集合 x に含まれる要素の総和を求める」などが部分問題として考えられる. このようにすることで, 一つの問題から複数の関数を得ることができ, コーパスのデータをより充実させることができる.

以上の課題により, 現在までに 18 個の問題から 262 個の仕様書と Python 言語による関数定義の対が得られた. これらのデータのうち代表的なものを図 1 に示す.

```python
def f0142(a):
  """a が 0 と等しければ True, そうでなければ False を返す ."""
  return a == 0
```

```python
def f0008(n, a):
  """n が a で割り切れなくなるまで n を a で割り , その結果を返す ."""
  while n % a == 0:
    n /= a
  return int(n)
```

```python
def f0145(a):
  """ 整数 a が素数であれば True を , そうでなければ False を返す ."""
  if a < 2:
    return False
  elif a == 2:
    return True
  else:
    for i in range(2, int(a**0.5) + 1):
      if a % i == 0:
        return False
    return True
```

図 1 収集されたコーパスの例
Fig. 1 Example of collected corpus.

4.3 コーパスの分類

収集された関数は, 関数内に出現する return 文の位置や個数によって以下に示す 3 種類に分類した. これらの分類は以下でそれぞれ述べるように問題としての難しさが異なると考えられ, 段階的に対応してゆく必要があると考えられる.

4.3.1 単一の return 文からなる関数

最初の分類は, 図 1 の最初の例に示すような, 関数の内容が唯一つの return 文により構成されるものである. このように記述された関数は手続き型言語よりは関数型言語の特徴に近い. これ以外の分類のように文が複数ある場合はその記述順などを考慮しなければならないが, この分類に属する関数はそのような問題はなく, 純粋に仕様書と処理内容のどの部分が対応するかのみを考えればよい.

4.3.2 複数の文と単一の return 文からなる関数

次の分類は図 1 の 2 番目の例に示すような, まず何らかの処理を行う文が続き, 最後に return 文が一つだけ登場する形のものである. この場合, 仕様書と処理内容の対応を取るだけでは, 変数など内部状態の扱いに関して問題が起こる. 図 5 の例では最初の while 文で変数 n が変更され, return 文が変更された n を利用している. このためこれら 2 文の間には順序関係があり, return 文の前に必ず while 文がないと仕様書の要件を満たすことができない.

4.3.3 複数の return 文を含む関数

上記のいずれにも属さない関数として, 図 1 の一番下の例のように, 関数定義の複数箇所に return 文が出現するものがある. この関数は簡単な素数判定を実装しているが, 最初の 2 つの条件分岐によってアルゴリズムを実行するための事前条件に合致しないものを除外している. また繰り返し文の中でも特定の条件に合致しなかった場合に即座に関数を終了するようになっている. このように return 文が複数ある場合, 単一の return 文の場合に比べて関数全体の処理順序に一貫性がなくなってしまい, 仕様書に記述された内容と対応を取るのが難しくなると考えられる.

表 1 に収集した関数を実際に分類した結果を示す. 単一 return 文の 2 分類に比べて複数 return 文の分類が少ないが, これはコーパスの収集に使用した問題集が数学関係であることと, 部分問題を抽出するというタスクを追加していることに関係があると考えられる.

[*1] https://projecteuler.net/

© 2014 Information Processing Society of Japan

表 1　コーパスの分類
Table 1　Classification of corpus.

分類	個数
単一 `return`	90
複数文+単一 `return`	161
複数 `return`	11
計	262

5. 機械翻訳としての自動プログラミング

3 節で述べた入出力の制約により, 仕様書中のあるフレーズと出力となる構文木の部分的な構造との間に一定の対応関係が得られると考えられる. よってこの問題設定の上で最初に考えることは, 仕様書とプログラムの対からこれらの対応関係をルールとして抽出することである. このルールを得ることができれば, あとはルール同士を適切な方法でつなぎ合わせることによって最終的な出力を生成することができると考えられる. この考え方は機械翻訳の枠組みで捉えることができ, 特に原言語に文字列, 目的言語に木構造を想定するため, **文字列から木構造への翻訳** (**string-to-tree 翻訳**) に属する問題の一種であると考えられる.

先行研究 [5] では自然言語による入力に直接解析を行うことで仕様に相当する表現を得ているのに対し, 機械翻訳の枠組みで自動プログラミングを捉える場合, 仕様書の一部と特定のソースコードの間に直接対応関係を見出そうとする. このアプローチでは, データさえ大量に用意することができれば既存の機械翻訳の手法を用いて自動的にルールを獲得することができ, 先行研究のように仕様とソースコードの対応関係をあらかじめ用意しておく必要がなくなると考えられる.

自然言語同士を入出力とする string-to-tree 翻訳では, 原言語側のトークン列から目的言語側の構文木への変換を行い, 得られた構文木から目的言語のトークン列を生成することで翻訳を行う. 本節では, 既存の string-to-tree 翻訳の手法を仕様書からプログラムへの変換に用いた場合にどのような結果が得られるのかを考察する.

string-to-tree 翻訳の枠組みでは, まず学習用に用意された対訳コーパスから原言語側のトークン列と目的言語側の構文木を生成し, この情報から原言語側のフレーズと目的言語側の部分的な構文木を対応付けた**同期ルール**を学習する. また, 各ルールやルール同士の組み合わせに関する重要度を表す重み係数も同時に学習する. 新たな原言語文が与えられたときは学習された同期ルールを用いて原言語文を再現できるような組み合わせを探索し, 発見されたルールの組み合わせの中で結合重みが最も大きくなるような候補から得られる構文木を翻訳結果として出力する.

これらの学習の前処理として, 原言語と目的言語の双方に対応する字句解析器, および目的言語の構文解析器が必要となる. 自然言語に対する字句解析や構文解析には必ず解析誤りが発生するため, 入出力の双方に自然言語を想定する通常の機械翻訳では, これらの解析誤りによる翻訳精度への影響が問題となる. 一方, 自動プログラミングの場合は目的言語側が形式言語であり, 2 節で述べたように字句解析誤りや構文解析誤りは原理的には発生しない. このため, 目的言語側の解析誤りによる翻訳精度への影響はないと考えてよい.

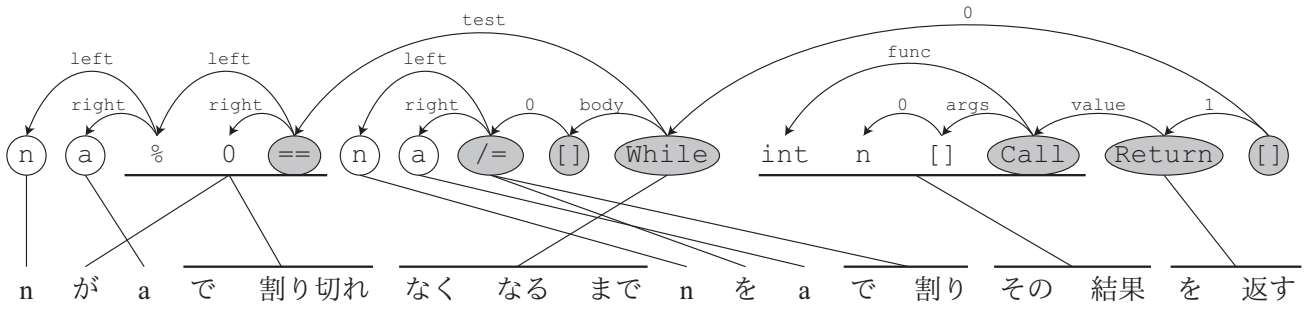
本稿では string-to-tree 翻訳の同期ルールの獲得に用いられる Galley-Hopkins-Knight-Marcu アルゴリズム (GHKM アルゴリズム) [7] によって, 仕様書中のフレーズとプログラミング言語の部分的な構文木との間の同期ルールを得ることを考える. GHKM アルゴリズムは原言語のトークン列と目的言語の構文木の葉ノードとの間の対応関係（アライメント）を参考にしながら, フレーズから部分木への同期ルールを抽出するアルゴリズムである. もともとの手法は構文木側のアライメント対象として葉ノードしか考慮していないが, これに内部ノードを含めることでトークン列と数式の対応関係を得る手法が提案されており [8], 本稿ではこちらの手法に基づくことにする.

GHKM アルゴリズムではフレーズに対するアライメントが他の部分木にまたがないような最小の部分木を**最小ルール**として抽出し, これを組み合わせることでより詳細なルールを獲得する. 最小ルールは**フロンティアノード**と呼ばれるノードを基準に構文木を切り出すことで得られる. 以下では 4 節で得たコーパスに対して実際にルールの抽出を行い, その上で発生する問題点について述べる.

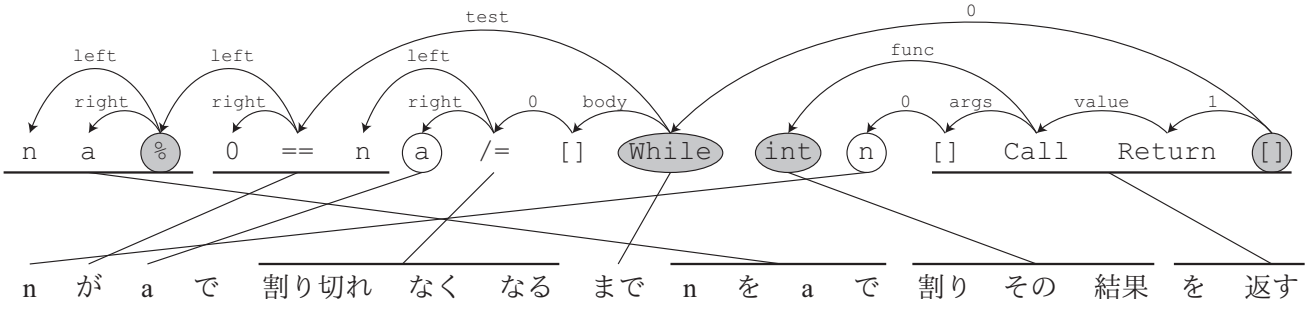
5.1 仕様書と構文木のアライメント

GHKM アルゴリズムを適用する前に, 仕様書とプログラミング言語の構文木との間にアライメントを得る必要がある. 本稿では, 仕様書の文を MeCab[9] により分割したトークン列と, Python 言語自身が標準装備する抽象構文木ライブラリ (`ast`) を用いて生成されたソースコードの構文木の各ノードとの間に, オラクルとしての人手によるアライメント, および教師なし学習による自動アライメントを行った. 人手によるアライメントは, 原言語側のすべての語を目的言語側の少なくとも一つのノードと関連付けること, 複数のノードと対応付ける場合はすべてのノードが構文木の上で接続されていることの 2 点を制約条件とした. 後者の条件は, 以降の処理で同期ルールを抽出する際にルールあたりのノード数を抑える意図がある. 教師なしアライメントに関しては多対多アライメントツールである pialign[10] を用い, コーパスから複数の `return` 文を含む 11 データを除いたもののうち, 構文木のノード数が 20 個以下となる仕様書と構文木の対 143 個に対してアライメントの学習を行った. ここで, pialign はフレーズ同士のアライメントを学習するため, 目的言語側の木構造のすべてのノードを一

© 2014 Information Processing Society of Japan



(a) 人手アライメント



(b) 教師なしアライメント

図 2 収集されたコーパスに対する構文木とアライメント, およびフロンティアノード
Fig. 2 Syntax tree, alignments, and frontier nodes for collected corpora.

列に並べておく必要がある. この順序は構文木上の帰りがけ順 (post-order) とした. これは, 倒置などを考えない場合の日本語の語順も構文木に対して帰りがけ順となっているためである.

図 2 に, 図 1 の 2 番目の例に対する人手アライメントと教師なしアライメントの結果を示す. 構文木は両者とも帰りがけ順でノードを並べたものを示した. また各言語のトークンの上下に引かれた横線はフレーズとしてアライメントされたことを表し, 線上のすべてのトークンに対してアライメントが存在することを表す.

仕様書の記述時に変数名を含めるよう強制しているため, 教師なし学習でも変数名に対するアライメントは比較的成功している. ただし図 2 の場合は, 意味的には直接対応しない位置の変数に対して関連付けがされてしまっているため問題がある. また, 帰りがけ順により偶然近い位置に並べられたノードが同じフレーズに含まれてしまう場合があり, 図 2 の例では「が」に関連付けられた 3 個のノードのうち 1 個が別の部分木に属していることが挙げられる.

他には, `While` ノードに対する「まで」, `/=`ノードに対する「割り切れ なく なる」などのように, コーパス中の共起頻度が高いために, 意味的に逆となるフレーズが対応付けられてしまう例も見られる.

自動アライメントを用いる場合はこれらの問題に対処しなければならない. まず最初に, 現在使用しているコーパスではアライメントの学習にはデータが少なすぎると考えられ, 単純にデータの増加による精度の向上は見込めると考えられる. また, 少量の人手アライメント結果を学習データに混ぜ, 教師あり学習によるアライメント [11] を適用することも考えられる. さらには, 図 2 の「が」の例のように木構造に対して一貫性のないアライメントを避けるために, 構文木の情報を用いてフレーズ候補を制限する必要もあると考えられる.

5.2 フロンティアノードの選択

仕様書と構文木との間にアライメントが得られたら, ルール抽出の基準となる構文木上のフロンティアノードを選択する. フロンティアノードとはそのノードを根とする部分木が構文木の他の一部とアライメントを共有していないようなノードのことであり, これらのノードが最小ルール同士の境界となる. ここでは先行研究 [8] に従った手法によりフロンティアノードかどうかを判定する.

まず構文木の各ノード v に対し, $\gamma(v)$ を v を根とする部分木内のノードに関連付けられているトークンのインデックス集合, $\delta(v)$ を v の子孫でも先祖でもないノードに関連付けられているトークンのインデックス集合と定義する. また $closure(I)$ を, インデックス集合 I の最小値から最大値までのすべての整数の集合とする. たとえば $I = \{3, 5\}$ の場合は $closure(I) = \{3, 4, 5\}$ となる.

このときフロンティアノードとは,

$$closure(\gamma(v)) \cap \delta(v) = \varnothing \quad (1)$$

© 2014 Information Processing Society of Japan

Vol.2014-NL-216 No.22
Vol.2014-SLP-101 No.22
2014/5/23

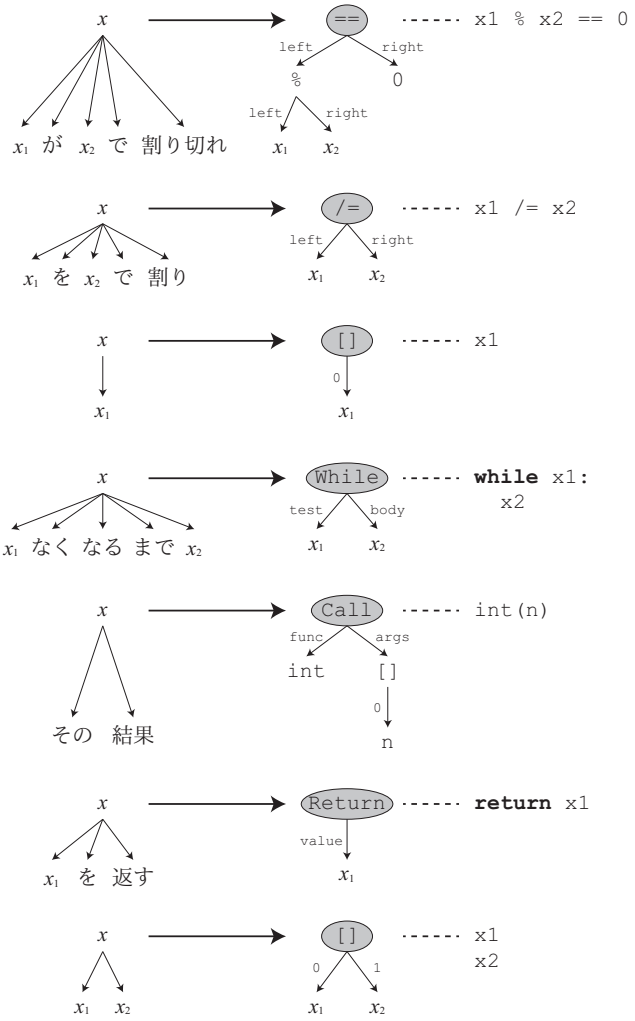


図 3 人手アライメントから得られる GHKM アルゴリズムの最小ルール

Fig. 3 Minimal rules of GHKM algorithm generated by hand-made alignment.

を満たすようなノード v であると定義する.

図 2 の楕円で囲まれたノードは, そのノードがフロンティアノードであることを表す.

5.3 最小ルールの抽出

得られたフロンティアノードを基に, 目的言語側の構文木から最小ルールとなる部分木の抽出を行う. ここでは最小ルールの部分木を, 変数名以外のフロンティアノードを根 v_r とし, 他のフロンティアノードか葉で終端するような部分木とする. 図 2 中の灰色で塗りつぶされたノードは最小ルールの根の候補となるノードである. このようにして抽出された部分木の終端にあるフロンティアノード $v_1, \cdots, v_n$ をすべて変数 $x_1, \cdots, x_n$ で置き換えることでルールを得る. 部分木に対応する原言語側のルールは, 目的言語側の根に対応するフレーズ $closure(v_r)$ から, 変数で置き換えられたフロンティアノードに対応するフレーズ $closure(v_1), \cdots, closure(v_n)$ をそれぞれ変数 $x_1, \cdots, x_n$ で置き換えたものである.

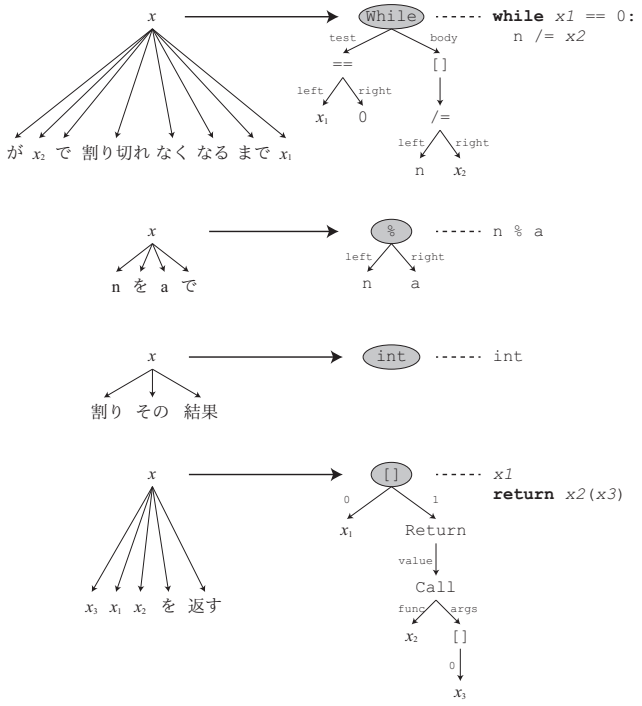


図 4 教師なしアライメントから得られる GHKM アルゴリズムの最小ルール

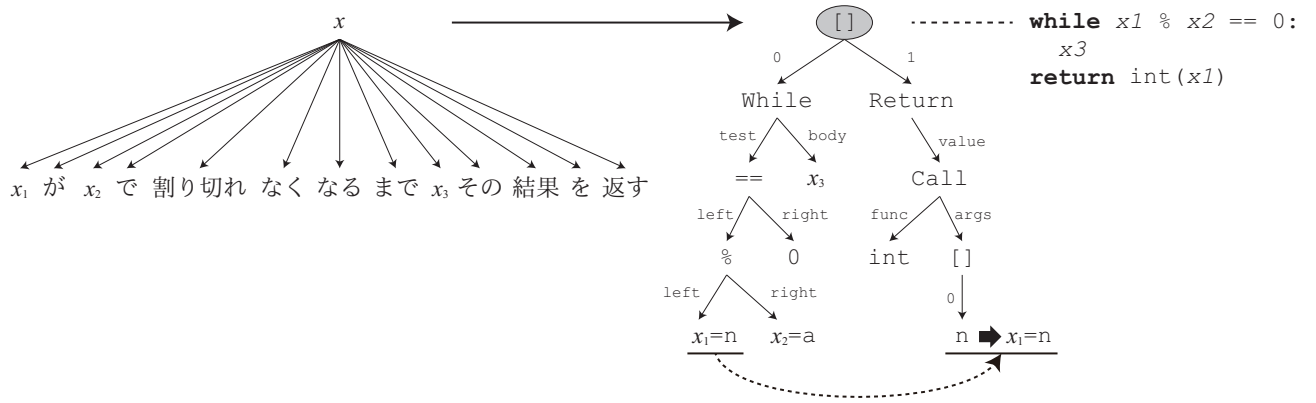
Fig. 4 Minimal rules of GHKM algorithm generated by unsupervised alignment.

図 3, 4 に, 図 2 のフロンティアノードから得られるすべての最小ルールを示す. 最も左にある日本語を含む木が原言語側のルール, 中央の木が目的言語側のルールである. 構文木の右に破線で結んだのは構文木に直接対応する Python 言語のソースコードである.

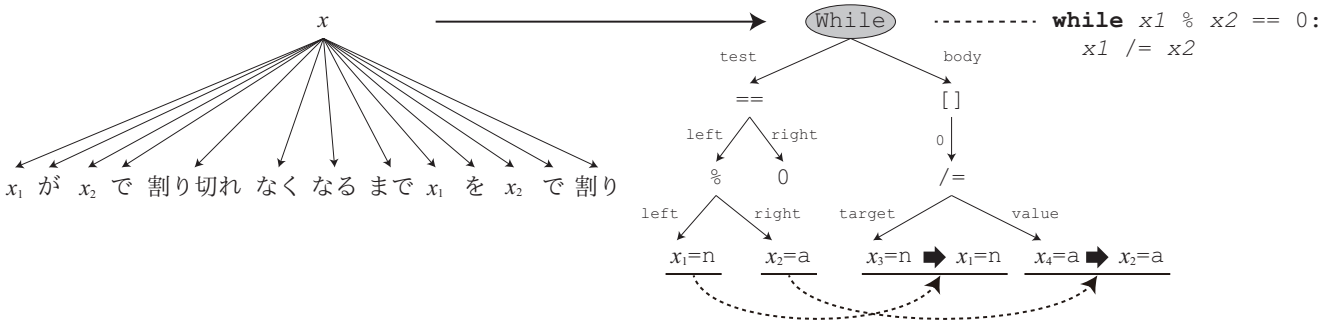
人手アライメントから得られた最小ルールは原言語側と目的言語側の意味的なつながりをよく表している. ただし「その 結果」に対する `int(n)` のように, 変数名が同期ルール内に残ってしまっているものについては単独ではルールとして用いることはできない. これは, 変数名自体は処理の内容に関係なく自由に決めることができるため, ルール側で変数名を固定してしまうとルール同士の組み合わせ時に他の変数との整合性が取れなくなってしまうためである. しかし, この残った変数については, 後に述べる組合せルールの抽出時に一定の範囲内で解決できる可能性がある.

教師なしアライメントにより得られる最小ルールは人手アライメントよりも数が少なく, 比較的大きな構造を持つものとなっている. これはアライメントが部分木をまたぐなど一貫性がないため, 大きな構造でないとうまくルールを抽出できないためである. また原言語側と目的言語側の変数の対応を見ても, 原言語側で `while` 文の処理内容を表しているはずの x_1 が目的言語側では条件式の項になっているなど, 意味的に間違っているものを抽出してしまっている. これはアライメントの時点で意味的な対応関係のな

© 2014 Information Processing Society of Japan



(a) 変数名の置き換え



(b) 変数の統合

図 5 同期ルールの組み合わせによる変数の変換
Fig. 5 Variable transformation by combining synchronous rules.

いもの同士が関連付けられてしまっているために起こる現象である.

5.4 組み合わせルールの作成

得られた最小ルールを元の構文木の構造に従って組み合わせることで, より大きな同期ルールを得ることができる. このようにして得たルールを**組み合わせルール**と呼ぶ. ここで組み合わせルールを作る際, 通常の string-to-tree 翻訳には現れなかった問題として構文木上の変数の扱いがある. ここで変数とは, 元の構文木に含まれるプログラミング言語としての変数を表すノードと, 同期ルール内に出現する変数の 2 種類が考えられるが, 変数名と言った場合には前者を, 単に変数と言った場合は後者を指すものとする.

まず, 最終的な同期ルールの中に変数名が残っている場合が挙げられる. この場合, 前述のようにその同期ルールを直接使用することができなくなってしまうため, 構文木中に残った変数名を消去する可能性について考える必要がある. ここでは一つの解決策として, 同期ルールを抽出する際, 構文木上の変数で置き換えられたノードが変数名を表していた場合, 元の変数名と置き換えられた変数との対応を記録しておく. 同期ルール同士を組み合わせる際, 構文木上に残る変数名とすでに対応関係が記録されている変数名を比較し, 対応関係がある場合は置き換えられた変数と同じ変数を代わりに挿入することで, 変数名を構文木から取り除くことができると考えられる. 図 5(a) に, 組み合わせルールから変数名を除去する例を示す.

また, 通常の string-to-tree 翻訳では同期ルールを組み合わせる際に目的言語側の構文木に含まれる変数に重複が発生しないよう書き換えを行うが, 同じ変数名が複数の同期ルール内の変数と対応している場合, 組み合わせルール内に同じ変数名を表す変数が複数発生してしまう問題がある. これに関しては, 該当する変数をひとつの同じ変数として統合してしまうことで, 元の変数名に対して一貫したルールを獲得することができると考えられる. 図 5(b) に, 元の変数名を共有する 2 組の変数を統合する例を示す.

これらの変数の変換で変数名を取り除くことができるのは大きな同期ルールを得られる場合のみであり, 変数名が残ったままの小さな同期ルールを利用するには別の手法が必要となる.

5.5 tree-to-string 翻訳の枠組みを直接適用することによる問題点

以上で string-to-tree 翻訳の枠組み, 特に GHKM アルゴリズムによる同期ルールの抽出を行うとどのような結果を得ることが期待できるかを述べた. しかし通常の自然言語同士に対する翻訳とは異なり, プログラミング言語の構文

© 2014 Information Processing Society of Japan

木には単なる構文情報以外にも考慮するべき問題点がいくつか挙げられる. 以下では特に重要な 2 個の問題点について述べる.

5.5.1 構文木に複数回出現する変数

プログラム側で同じ変数が仕様書内での出現回数よりも多く使用されている場合, 仕様書中の変数名と構文木中のノードの間にアライメントを取のが難しくなるという問題が発生する. 例えば図 1 の 2 番目の例で, 仕様書の文が「`n` が `a` で割り切れなくなるまで `a` で割り」となっていた場合, 対応する構文木上に `n` が 2 回登場するにも関わらず仕様書には 1 回しか登場しないこととなる. この場合の `n` に対するアライメントの取り方は, 全ての対応する構文木上のノードに対してアライメントを取るか, いずれか一つのノードに対して取るかの 2 種類となる. 前者を選んだ場合, 変数名に関するアライメントが離れた位置の構文木にまたがることとなるため, コーパスから抽出される最小ルールが巨大化することとなり, 得られるルールの汎用性が低下してしまう. また既存の string-to-tree 翻訳の枠組みでは複数個所で同時に木構造の置換を行うことはできないため, 一度に複数の変数を書き換えるようなルールを学習するこの方法は望ましくない. 後者を選んだ場合, 構文木に登場する変数のうちいくつかはアライメントが取られず残ることとなり, 最小ルール中に変数名が残ってしまう問題が発生する.

5.5.2 構文の順序

プログラム中に現れる文同士は通常その記述された順序に意味がある. たとえば 4 節で述べたように, 図 1 の 2 番目の例では, `while` 文と `return` 文を入れ替えてしまうと仕様書で要求されたものとは異なる動作を示す関数となってしまう. しかし通常の string-to-tree 翻訳では, 本稿の例でいう `[]` ノードのような並列構造を持つような構文木を想定しておらず, ルールを並列に組み合わせる際の順序まで制御することはできない. この問題は 4 節で挙げたコーパスの分類のうち, 単一 `return` 文のみによる関数に分類されたもの以外のすべての例で問題となるため, 解決する手法が必要である.

6. おわりに

本稿では自動プログラミングとして自然言語による仕様書からプログラムへの変換を問題設定とし, この学習・評価のためのコーパスを収集した. また問題を string-to-tree 翻訳の枠組みで捉え, GHKM アルゴリズムに基づく同期ルールの抽出を行ったときにどのような結果が得られるかを示した. 今後は, これらの手法で実際に得られるルールを用いて実際に仕様書からプログラムへの変換が可能かどうかを検証する. また, 5 節で述べた単純に string-to-tree 翻訳の枠組みを適用した場合に発生する問題を解決する手法についても今後考える.

参考文献

[1] Niaz, I. A. and Tanaka, J.: Code Generation From Uml Statecharts, *in Proc. 7th IASTED International Conf. on Software Engineering and Application (SEA 2003), Marina Del Rey* (2003).

[2] Long, Q., Liu, Z., Li, X. and Jifeng, H.: Consistent code generation from UML models, *Software Engineering Conference, 2005. Proceedings. 2005 Australian*, pp. 23–30 (2005).

[3] Steedman, M. and Baldridge, J.: Combinatory Categorial Grammar (2007).

[4] Kamp, H. and Reyle, U.: *From discourse to logic: Introduction to modeltheoretic semantics of natural language, formal logic and discourse representation theory*, No. 42 (1993).

[5] Vadas, D. and Curran, J. R.: Programming With Unrestricted Natural Language, *Proceedings of the Australasian Language Technology Workshop*, pp. 191–199 (2005).

[6] Sridhara, G., Hill, E., Muppaneni, D., Pollock, L. and Vijay-Shanker, K.: Towards automatically generating summary comments for java methods, *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pp. 43–52 (2010).

[7] Galley, M., Hopkins, M., Knight, K. and Marcu, D.: What's in a Translation Rule?, *HLT-NAACL*, pp. 273–280 (2004).

[8] Li, P., Liu, Y. and Sun, M.: An Extended GHKM Algorithm for Inducing λ-SCFG, *Twenty-Seventh AAAI Conference on Artificial Intelligence* (2013).

[9] Kudo, T., Yamamoto, K. and Matsumoto, Y.: Applying Conditional Random Fields to Japanese Morphological Analysis., *EMNLP*, Vol. 4, pp. 230–237 (2004).

[10] Neubig, G., Watanabe, T., Sumita, E., Mori, S. and Kawahara, T.: An Unsupervised Model for Joint Phrase Alignment and Extraction, *The 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies (ACL-HLT)*, Portland, Oregon, USA, pp. 632–641 (2011).

[11] Riesa, J., Irvine, A. and Marcu, D.: Feature-Rich Language-Independent Syntax-Based Alignment for Statistical Machine Translation, *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing* (2011).

© 2014 Information Processing Society of Japan