情報処理学会論文誌 Vol.55 No.1 45–56 (Jan. 2014)

De-gapper—プログラミング初学者の段階的な理解を支援するツール

長 慎也[1,a)] 保福 やよい[2,3] 西田 知博[4] 兼宗 進[3]

受付日 2013年3月18日, 採録日 2013年10月9日

概要：プログラミングを学習するにあたって，学習者は教材を参照しながら，プログラミングに必要な要素を少しずつ学んでいく．しかし，教材によっては，1つのプログラムにたくさんの学習内容が詰まっており，これが学習者のつまずきの原因となることがある．また，教える側はプログラミングを熟知しているがために，知らず知らずのうちに一度にたくさんの学習内容を詰め込んでしまうことがある．この問題を解決するため，教材にあるプログラムの構文要素から，一度にたくさんの学習内容が出てきていないかを自動的に判定するツール De-gapper を開発した．実際の教材に掲載された例題について，De-gapper が検出した学習内容の量と，目視で確認した学習内容の量とが一致しているかどうかを検証した．

キーワード：プログラミング学習，教員支援，教材作成支援，構文解析

De-gapper – Tool for Support Programming Learners' "Step-by-step" Learning

Shinya Cho[1,a)] Yayoi Hofuku[2,3] Tomohiro Nishida[4] Susumu Kanemune[3]

Received: March 18, 2013, Accepted: October 9, 2013

Abstract: In programming leraning, students learn programming elements step-by-step by referring learning materials. But some materials tend to put too many learning concepts into one program because teachers are familiar with programming and they expect students can understand these concepts. This may be a factor of students' frustration. To solve this problem, we developed a tool named "De-gapper" which analyzes syntax trees of programs in learing material and check if there are too many learing concepts. Using De-gapper, we verified whether De-gapper's outputs are correspond to the amount of learing concepts checked manually in actual teaching material.

Keywords: programming learning, teaching support, teaching material development support, syntax analysis

1. はじめに

プログラミングの学習は，それを通じてコンピュータの本質的な動きを理解させることにつながり，「情報の科学的な理解」だけでなく情報社会の基礎を理解させるうえでも役に立つと考えられる．

しかしプログラミングを学ぶ際には，習得しなければならないことが多く，学習の初期段階でドロップアウトすることも少なくない [1], [2]．

そこで我々は，「一度に学ぶ内容（学習項目）が多すぎて，初学者の理解を超えている」ことがつまずきの原因の1つになっている点に着目した．新しいことを少しずつ教えていく，という考え方は学校教育などで取り入れられており，この考え方はプログラミング教育にも適用されるべ

1 明星大学
Meisei University, Hino, Tokyo 191–8506, Japan
2 神奈川県立相模向陽館高等学校
Sagami Kouyoukan High School, Zama, Kanagawa 252–0003, Japan
3 大阪電気通信大学
Osaka Electro-Communication University, Neyagawa, Osaka 572–8530, Japan
4 大阪学院大学
Osaka Gakuin University, Suita, Osaka 564–8511, Japan
a) cho@eplang.jp

きである．

しかし，教える側としては「新しい学習項目を少しずつ教えている」つもりでも，自分自身が困難をあまり感じないために，つい一度にたくさんのことを教えてしまいがちである．そこで，教材が本当に「新しい学習項目を少しずつ教えている」かを客観的に確認できるようなツールが必要であると考えた．

本論文では，教科書の例題や授業で使用されるサンプルプログラム間に出現する構文要素の差分を抽出するツールDe-gapperを提案し，De-gapperを授業で用いられている教材に適用した結果を報告する．

2. 関連研究

初学者がプログラムを学ぶときにぶつかる困難さを述べている研究は数多くある．1980年代にはプログラミングのコースを終えた学習者の38%しか，繰返しを用いて平均を求めることができなかったという報告[3]や，初学者はプログラミングの基本的な概念が曖昧であり，入門的な問題解決能力が欠如しているという分析[1]などがなされている．

2004年にMcCrackenらは4大学の216名の初学者に対して，プログラミングのテストを行ったが，多くの学習者が得点が低く，理解ができていないことが分かった[2]．

そこで，学習者の理解の構造や度合いを分析し，効果的な学習方法を探るさまざまな試みがなされている．

文献[4], [5]では，読む，書く，模倣，トレースなどをプログラミングのスキルとして定義し，ペーパテストを行うことにより，それらのスキルの関連性を分析して，学習者の理解構造を明らかにしようとしている．

内藤らは，学習者が作成しているプログラムを直接モニタリングし，テスト項目をクリアしているかどうかを判定することにより進捗を分析して，大規模なクラスの演習をサポートするシステムを提案している[6]．

谷川らは，プログラムの関数呼び出しに着目し，ライブラリ関数の代わりに用意したスパイ関数によって呼び出しを記録することによって，関数呼び出しのパターンによる学生の習得項目の把握を行っている[7]．

これらのほかにも，学習者の状況をリアルタイムに分析し，それを指導に使おうという試みは多くなされている[8], [9], [10]．

授業の中において学習者の作成したプログラムを分析して指導に活かすことは重要であるが，進行中の授業の中で対応するために問題をかかえたすべての学習者に対応することは難しい．したがって，我々は，授業を行う前の段階として，学習者が無理なく理解できる教材をあらかじめ設計しておくことが非常に重要であると考え，教材中のプログラムを静的に分析する今回のツールの開発，提案を行った．

3. プログラミングの学習項目と飛躍

3.1 例題間の飛躍

教材に現れるプログラム（例題）1つにつき，たくさんの学習する内容（学習項目）が入っていると，結果として，例題と例題の間に大きなギャップ（飛躍）が生じ，学習者にとって理解が困難になる．**図 1** に，C言語での例題の飛躍の例を示す．例題 (a) は，文字列をそのまま表示する例題であり，C言語の初学者が見た場合でも容易に理解可能である．次に，初学者が例題 (b) を見た場合に，初学者にとって「新しい概念」はどの程度あるか，ということを考える．例題 (b) で教員が意図した新しい概念は，「変数を使って計算をした数値を表示する」という1個の概念だけである．

しかし実際には，例題 (a) にある概念，

- 関数の呼び出し：`printf()`
- 関数呼び出しに1つの引数を指定：`printf(xxx)`
- 文字列：`"Hello"`
- 文の区切り記号：`;`

に加え，例題 (b) には次のような多くの概念が入ってしまっている．

- 変数の宣言：`int n`
- 変数のデータ型：`int`
- 宣言時の変数の初期化：`int n = 10`
- 算術演算：`n+20`
- 書式指定：`%d`
- 関数呼び出しに2つの引数を指定：`printf("",xxx)`
- 関数呼び出しの引数に変数を指定：`printf("",n)`
- 関数呼び出しの引数内に式を記述：`printf("",n+20)`

この飛躍を小さくする（飛躍を埋める）には，**図 2** に示した例題 (a-2), (a-3) を示してから例題 (b) を示す．(a-2) を提示することにより，

- 書式指定：`%d`
- 関数呼び出しに2つの引数を指定：`printf("",10)`

という概念が学べ，(a-3) を提示することにより，

```
/* 例題 (a) */
printf("Hello");
```

```
/* 例題 (b) */
int n = 10;
printf("%d", n+20);
```

図 1 複数の概念が出現する例題の例
Fig. 1 Example of a big gap.

```
/* 例題 (a-2) */
printf("%d", 10);
```

```
/* 例題 (a-3) */
int n = 10;
printf("%d", n);
```

図 2 飛躍の小さい例題を追加する例
Fig. 2 Complementary example to fill the gap.

- 変数の宣言：`int n`
- 変数のデータ型：`int`
- 宣言時の変数の初期化：`int n = 10`
- 関数呼び出しの引数に変数を指定：`printf("",n)`

を学ぶことができるので，(b) で学ぶことは，

- 算術演算：`n+20`
- 関数呼び出しの引数内に式を記述：`printf("",n+20)`

となり，飛躍を小さくすることができる．

3.2 構文要素の分析で抽出可能な学習項目

1 章で述べたように，我々は構文要素の差分をとることにより，学習者が新たに学ばなければならない学習項目を抽出し，飛躍の検出を行うことを考えた．

学習項目には「プログラムは文字で書く」などの基本的な概念や，「命令（関数）がある」や「変数がある」など，プログラムを構成する要素そのものであり，プログラムの記述以前のものがある．

また，「プログラムは上から順に実行される」「f(x) という関数呼び出しは，x の中身を評価（計算）してから，f を呼び出す」「printf を使うと文字を表示できる」など，実行して分かるような，プログラムの評価戦略や組み込み関数の振舞いなどに関わる学習項目もある．

しかし，これらの学習項目は構文の解析によって抽出することは難しい．そこで，ここでは，ソースコードの字面から分析できるような学習項目を対象とする．この観点から分析すると，3.1 節であげた例の学習項目は以下のようになる．

図 1 **例題 (a) の学習項目：**

- 命令は英字で構成される名前を持つ．
- 文字は `"..."` で囲む．
- 文は「;」で終る．
- 命令は `f(p);` という形で書く．

図 2 **例題 (a-2) の新出学習項目：**

- 命令の引数は 2 個書くことができる．
- 複数個の引数は「,」で区切る．
- 引数に数を書くことができる．
- 命令は f(p); だけでなく f(p1,p2); とも書くことができる．

図 2 **例題 (a-3) の新出学習項目：**

- 変数は英字で構成される名前を持つ．
- 「変数名=値」という形で変数に値を入れることができる．
- 数値は `int` で表す．
- 変数は「型　変数名;」で宣言する．
- 変数を宣言するときに「型　変数名=値;」という形で値を指定できる．
- 命令の引数に変数を書くことができる．

図 1 **例題 (b) の新出学習項目：**

- 命令の引数として計算式を書くことができる．
- 式の中に変数を書くことができる．

このように分析を行えば，学習項目が 4 個であった例題 (a) から例題 (b) へは 12 個の新出項目があるが，例題 (a-2)，(a-3) をその間に入れることにより，それぞれのステップでの新出項目を 4，6，2 個と減らして，学習の飛躍を抑えられていることが分かる．

3.1 節であげた概念は構文要素として De-gapper で検出可能なものである．この概念の数は，4 個から始まり新出概念は 2，4，2 個の計 12 個であり，プログラムを教える教員が注意深く検討した 16 個の学習項目の 75%をカバーしている．また，ここであげた例題の追加を行っても，例題 (a-2) から (a-3) の間には 6 個の新出学習項目があり，この間が飛躍となる可能性があるが，機械的な構文の分析でも 4 個の新出概念を見つけ，比較的多いことが検知できている．したがって，機械的な分析であっても，新しい概念が多く，さらなる例題の追加をするか，丁寧な指導を行う必要がある箇所の指摘を行うことができる．

4. 教科書の飛躍を見つけるツールの設計

教科書のサンプルプログラムに，先に述べたような飛躍が含まれている場合は，他のプログラムを補足することで，飛躍を埋めることができる．しかし，飛躍を人間による目視で見つけるのには時間がかかったり，見落としたりする可能性がある．この問題を解決するために，教科書の中のプログラムに現れた新しい学習項目を機械的に発見するツールを開発することが必要であると考えた．

3 章で述べたように，プログラムの表記に関する学習項目については，字句解析や構文解析など，ソースコードの機械的な解析によって検出がある程度可能と考えられる．

授業に用いられる教科書などの教材に掲載されているプログラムを先頭から順に構文解析を行えば，それぞれのプログラムで出現した新しい構文要素が検出できる．このとき「新しい構文要素」が多く出現するプログラムを「飛躍のあるプログラム」と考えれば，機械的に飛躍の検出が可能であると考え，その解析を行うツール De-gapper を開発した．

4.1 概要

De-gapper は，教材に掲載されているプログラムの列（プログラム列）を入力とする．プログラム列は，学習者がそれぞれのプログラムを学習する順番，つまり，教材に記載されている順番に並んでいるものとする．

De-gapper は，プログラム列の各プログラムを順番どおり分析する．各プログラムの分析においては，そのプログラムのソースコードを構文解析し，新しい構文要素，つまり，これまでの分析においては出てこなかった構文要素を検出する．すべてのプログラムについて分析が終わると，

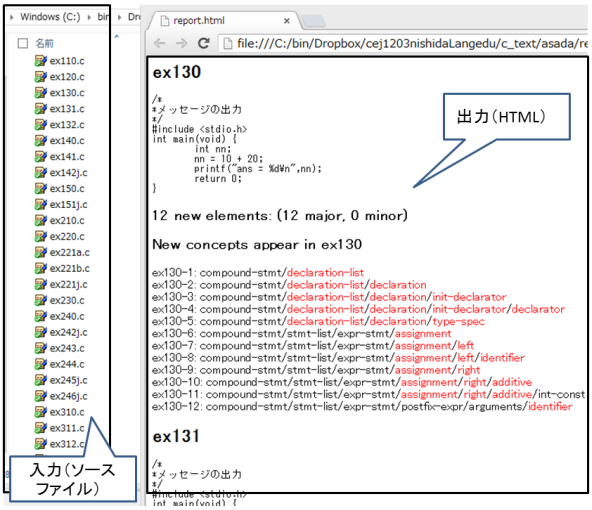

図 3 De-gapper の入力と出力の例
Fig. 3 Input and output examples of De-gapper.

各プログラムと，そのプログラムに現れた新しい構文要素の組を出力する．

図 3 に，De-gapper でのプログラムの入力と出力の例を示す．

4.2 実装

本節では，De-gapper における実際の分析方法を示す．

De-gapper は，プログラム列に含まれるそれぞれのプログラムを順番に分析する．各プログラムの分析においては，まず，プログラムのソースファイルを構文解析し，構文木を XML で表現したもの（XML 木）を記したファイル（XML 木ファイル）を生成する．次に，その XML 木に含まれる各要素のパス名を検出する．要素のパス名とは，ルート要素からその要素までの階層構造をたどったときに通る要素の列（パス）について，パスに含まれる要素の名前をルート要素から順に並べ，/ で連結したものである．ただし，“X-list”（X は任意の文字列）というタグ名の XML 木が，子要素を 0 個または 1 個しか含んでいない場合，パスから “X-list” が取り除かれる．これは，3 章で「命令の引数は 2 個書くことができる」などの学習項目があるとおり，複数並べられる構文要素が複数並べられていない場合，それらを複数並べられることを学習者は理解するとは限らないため，複数の要素が並んだときに初めて “X-list”（X が複数並んだもの）を提示することが適当である，という配慮のためである．

もし，これまでの分析，つまり，現在分析中のプログラムより前のプログラムについての分析で検出されていないパス名があれば，そのパス名はそのプログラムの「新しい構文要素」として検出される．

現時点の実装では，De-gapper が解析できる構文は，C 言語と Java である（ただし，C 言語のプリプロセッサ指令は解析前に除去している）が，構文解析を行って XML 木を生成する部分を変更すれば，他の言語にも簡単に対応することができる．C 言語の構文解析においては，K&R [11] の付録 A.13 に定義されている構文に準拠して XML 木を生成するが，構文および XML 木の生成方法に一部違いがある．詳細は付録 A.1 を参照されたい．

```
<expr-stmt>
  <function-call>
    <name>
      <ident>printf</ident>
    </name>
    <arguments>
      <lparen>(</lparen>
        <string>"Hello"</string>
      <rparen>)</rparen>
    </arguments>
  </function-call>
  <semicolon>;</semicolon>
</expr-stmt>
```

図 4 例題 (a) から生成される XML 木ファイル
Fig. 4 XML tree generated from example (a).

```
a-1: expr-stmt
a-2: expr-stmt/function-call
a-3: expr-stmt/function-call/arguments
a-4: expr-stmt/function-call/arguments/lparen
a-5: expr-stmt/function-call/arguments/rparen
a-6: expr-stmt/function-call/arguments/string
a-7: expr-stmt/function-call/name
a-8: expr-stmt/function-call/name/ident
a-9: expr-stmt/semicolon
```

図 5 例題 (a) のプログラムから検出される新しい構文要素
Fig. 5 Syntax elements detected from example (a).

4.3 実行例

本節では De-gapper の実際の実行例を示す．

4.3.1 例 1

ここでは，入力として，図 1 の例題 (a) と例題 (b) をこの順で与えた場合を考える．

図 4 に，例題 (a) から生成される XML 木ファイル（関数定義の内部のタグのみ）を示す．また，例題 (a) の XML 木のパス名は**図 5** のように検出される．なお，実際の De-gapper の出力においては，関数宣言に関連するパスがそれぞれのパスの前に付加されているが，本論文では，関数宣言が main のみであるようなプログラムだけを対象に議論しているため，これらのパスは省略して表記している．以下の図も同様である．

なお，これらはプログラム中に出現する順番ではなく，パス名を文字コード順に並べ替えたものになっている．これは，構文要素のパスどうしを比較するときに分かりやす

```
b-1: declaration
b-2: declaration/init-declarator
b-3: declaration/init-declarator/declarator
b-4: declaration/init-declarator/declarator/ident
b-5: declaration/init-declarator/initializer
b-6: declaration/init-declarator/initializer/equal
b-7: declaration/init-declarator/initializer/int-const
b-8: declaration/semicolon
b-9: declaration/type-spec
b-10: declaration/type-spec/int
b-11: expr-stmt/function-call/arguments/arg-expr-list
b-12: expr-stmt/function-call/arguments/arg-expr-list/additive
b-13: expr-stmt/function-call/arguments/arg-expr-list/additive/ident
b-14: expr-stmt/function-call/arguments/arg-expr-list/additive/int-const
b-15: expr-stmt/function-call/arguments/arg-expr-list/additive/plus
b-16: expr-stmt/function-call/arguments/arg-expr-list/comma
b-17: expr-stmt/function-call/arguments/arg-expr-list/string
```

図 6 例題 (b) のプログラムから検出される新しい構文要素
Fig. 6 New syntax elements detected from example (b).

```
/* const.c */
int main(void) {
  printf("%d", 57);
  printf("%d", 57 + 10);
}
```

```
/* var.c */
int main(void) {
  int vx,vy;
  vx= 57;
  vy= vx + 10;
  printf("vx = %d\n", vx);
  printf("vy = %d\n", vy);
}
```

図 7 De-gapper への入力例：const.c, var.c
Fig. 7 Input examples for De-gapper: const.c, var.c.

くするためである．また，同一のパス名は各プログラムにつき一度しか検出しない．

例題 (a) は，プログラム列の最初のプログラムであるので，図 5 にあげたすべてのパス名を新しい構文要素として検出する．それぞれのパス名には a-1 から a-9 までのラベルが付与される．

次に，例題 (b) の解析を行う．ここで検出される新しい構文要素は，「例題 (b) の XML 木から検出されるパス名のうち，これまでの分析で検出されていないもの，つまり，図 5 に含まれないもの」となる．具体的には**図 6** のようになる．

4.3.2 例 2

ここでは入力として**図 7** の const.c と var.c のプログラムをこの順で与えた場合を考える．const.c，var.c で検出される新しい構文要素を，それぞれ**図 8**，**図 9** に示す．

このうち，図 9 の var-17 と var-18 は「マイナーな」新しい構文要素として検出される．「マイナーな」とは，これまでに出現した構文要素に関する知識を組み合わせれば，容易に類推可能な構文要素を指す．De-gapper は，マイナーな構文要素については，どの構文要素から類推可能かを提示する．var-17 は，次のような考えに基づき，var-15 と const-7 から類推可能であると提示している．

- var-15 は，「右辺が加法式である代入式」を新しい構文要素としてあげている．
- さらに，var-17 は，「右辺が加法式である代入式」の加法式の中に「整数定数」がある，ということを新しい構文要素としてあげている．
- しかし，「加法式」はすでに const のプログラムで学習しており，さらに，const-7 で，「加法式」の中に「整数定数」があることも学習している．
- よって，var-17 は，var-15 と const-7 から類推可能である．

同様に var-18 は，「右辺が加法式である代入式」の加法式の中に「+」という演算子があることを示しているが，このことも const-8 で学習済みであることから類推可能であると判断している．

「マイナーな新しい構文要素」を厳密に定義すると次のようになる．A，B，C をそれぞれパス名として，A/B/C

```
const-1: stmt-list
const-2: stmt-list/expr-stmt
const-3: stmt-list/expr-stmt/function-call
const-4: stmt-list/expr-stmt/function-call/arguments
const-5: stmt-list/expr-stmt/function-call/arguments/arg-expr-list
const-6: stmt-list/expr-stmt/function-call/arguments/arg-expr-list/additive
const-7: stmt-list/expr-stmt/function-call/arguments/arg-expr-list/additive/int-const
const-8: stmt-list/expr-stmt/function-call/arguments/arg-expr-list/additive/plus
const-9: stmt-list/expr-stmt/function-call/arguments/arg-expr-list/int-const
const-10: stmt-list/expr-stmt/function-call/arguments/arg-expr-list/string
const-11: stmt-list/expr-stmt/function-call/arguments/lparen
const-12: stmt-list/expr-stmt/function-call/arguments/rparen
const-13: stmt-list/expr-stmt/function-call/name
const-14: stmt-list/expr-stmt/function-call/name/ident
const-15: stmt-list/expr-stmt/semicolon
```

図 8 const.c から検出される新しい構文要素
Fig. 8 New syntax elements detected from const.c.

```
var-1: declaration
var-2: declaration/init-declarator-list
var-3: declaration/init-declarator-list/comma
var-4: declaration/init-declarator-list/init-declarator
var-5: declaration/init-declarator-list/init-declarator/declarator
var-6: declaration/init-declarator-list/init-declarator/declarator/ident
var-7: declaration/semicolon
var-8: declaration/type-spec
var-9: declaration/type-spec/int
var-10: stmt-list/expr-stmt/assignment
var-11: stmt-list/expr-stmt/assignment/eq
var-12: stmt-list/expr-stmt/assignment/left
var-13: stmt-list/expr-stmt/assignment/left/ident
var-14: stmt-list/expr-stmt/assignment/right
var-15: stmt-list/expr-stmt/assignment/right/additive
var-16: stmt-list/expr-stmt/assignment/right/additive/ident
*var-17: stmt-list/expr-stmt/assignment/right/additive/int-const
var-17 is a Minor (a/b -> b/c) new concept; can be learned from:
var-15+const-7
*var-18: stmt-list/expr-stmt/assignment/right/additive/plus
var-18 is a Minor (a/b -> b/c) new concept; can be learned from:
var-15+const-8
var-19: stmt-list/expr-stmt/assignment/right/int-const
var-20: stmt-list/expr-stmt/function-call/arguments/arg-expr-list/ident
```

図 9 var.c から検出される新しい構文要素
Fig. 9 New syntax elements detected from var.c.

というパス名が新しい構文要素として検出されたときに，A/B というパス名の構文要素（$m1$）がすでに検出されていて（今解析しているプログラム自身で検出されている場合も含む），かつ，パス名が B/C で終わる構文要素（$m2$）がすでに検出されていれば，A/B/C はマイナーな新しい構文要素であるとする．$m1$ と $m2$ を，A/B/C がマイナーであることの「根拠」となる構文要素と呼ぶ．

var-17 の例では，
A=`stmt-list/expr-stmt/assignment/right`，
B=`additive`，C=`int-const` となる．なお，マイナーでない新しい構文要素は「メジャーな」新しい構文要素と呼ぶ．

5. 評価

5.1 評価方法

De-gapper が出力した新しい構文要素が，人手で検出した学習項目（以下，単に学習項目）とどの程度一致しているかを調べるために，著者の 1 人が実際の授業で使用した例題および演習のプログラム列を De-gapper に入力させた結果と，客観性を持たせるため，別の著者の 1 人が目視により分析した学習項目とを比較した．対象とした授業では文献 [12] をベースに例題や演習を作っており，今回は基本的な制御構造（条件分岐，繰返し）の学習までを範囲とした 53 個のプログラムを分析した．

前述のとおり，De-gapper で検出されるマイナーな新しい構文要素は，ほかから学習項目から類推可能と見なし，メジャーな新しい構文要素（以下，単に構文要素）と学習項目とを比較し，それらが「対応」しているかを検証した．

あるプログラムから検出された構文要素と学習項目とを比較し，構文要素と学習項目とが指摘しているソースコードの範囲が一致していれば，それらは「対応」しているとする．

ただし，「変数は『型　変数名;』で宣言する」というように，書式が明示的に書かれている学習項目は，それぞれ「型」「変数名」「;」に該当する 3 つの構文要素に対応するものとする．また，ある学習項目が指摘するソースコード上の範囲と同じ範囲を指摘する構文要素が複数ある場合，その学習項目はそれらすべての構文要素と対応するものとする．

5.2 構文要素と学習項目の個数・対応数

すべてのプログラムを通じて，構文要素は 140 個，学習項目は 78 個検出された．構文要素のうち 125 個と，学習項目のうち 70 個は，少なくとも 1 つ以上の学習項目および構文要素に対応した．

以下の節では，対応する学習項目がなかった構文要素と，対応する構文要素がなかった学習項目をあげる．

5.3 対応する学習項目がなかった構文要素

対応する学習項目がなかった構文要素とは，De-gapper は構文要素を検出をしたが，人手で検出した学習項目の中になかったものである．言い換えれば，ここであげられた構文要素は学習者にとって自明であると判断されたものである．このような構文要素が 15 個見つかった．

5.3.1 数学でも習う内容（4 個）

算術演算に関する構文要素のうち，加法式，等式，不等式や括弧つきの演算についての学習項目については，（初等中等教育における）数学で習う記法と同じであるという理由で，学習項目にはあげられなかった．「加法式の中に数値を書ける」「加法式に + を書ける」「加法式に − を書ける」「括弧のついた式の中に加法式が書ける」などの構文要素[*1]であった．

一方で，数学では学習しないか，記法が違うもの，たとえば ∗，/，%，!=，==，>= などの演算子の記法については学習項目にあげられていた．

5.3.2 以前のプログラムより単純な構文（3 個）

たとえば，`wa = a+b;` という文（代入式の右辺に加法式を書ける）を含むプログラムが提示された後に，`wa = c;` という文（代入式の右辺に変数 1 つのみを書ける）を含むプログラムが初めて提示された場合，De-gapper はこれを別の構文と見なす．人手では `wa = a+b;` というプログラムを理解していれば，`wa=c;` はより容易に理解するであろうと考え，学習項目としては検出されなかった．

5.3.3 同じように見えるものが，構文上区別されている場合に検出されるもの（8 個）

De-gapper の構文定義のうえでは else のない if 文（if-statement）と，else のある if 文（if-else-statement）を区別して扱っていた．このため「else のない if 文の条件式に論理積演算子（&&）が使える」「else のある if 文の条件式に論理積演算子（&&）が使える」という構文要素が別々に検出された．人手による分類では，これらのうち最初に出現したプログラムについてだけを学習項目としてあげていた．

5.4 対応する構文要素がなかった学習項目

対応する構文要素がなかった学習項目とは，人手によって検出されたが，De-gapper では検出されなかった学習項目である．このような学習項目が 8 個見つかった．

5.4.1 マイナーな構文要素として検出されたもの（7 個）

De-gapper が検出を行ったが，それがマイナーな学習項目に分類されたものが 7 個あげられた．

たとえば**図 10** にあげる ex141.c からは「変数に値を入れるとき，変数を使った計算式で入れることができる」という学習項目を検出した（図中の★）が，これはマイナーであると分類された．

```
#include <stdio.h>
int main(void) {
     int a, b, wa;
     a = 100;  b= 8;
     wa = a+b;  //★
     printf("%d + %d = %d\n",a, b, wa);
     return 0;
}
```

図 10 プログラム ex141.c
Fig. 10 Program ex141.c.

[*1] 構文要素は，De-gapper が検出したものであるので，実際にはパス名の形式で検出されているが，読みやすくするためにパス名から推測される学習項目の形式で表記する．本項以下同様とする．

```
#include <stdio.h>
int main(void) {
     int nn;
     nn = 10 + 20; //★
     printf("ans = %d\n",nn);
     return 0;
}
```

図 11 プログラム ex130
Fig. 11 Program ex130.

```
#include <stdio.h>
int main(void) {
     int a, b;
     a = 100;  b= 8;
     printf("%d + %d = %d\n",a, b, a+b /*★*/);
     return 0;
}
```

図 12 プログラム ex140
Fig. 12 Program ex140.

マイナーとされた根拠は，このプログラムより前に提示された**図 11** と**図 12** のプログラムに基づく．

ex130-13 と ex140-7 は，それぞれ「変数に値を入れるとき，数値の式の形で入れることができる」「命令の引数として計算式を書け，計算式の中に変数を書ける」という学習項目と対応していたが，人手による検出においては，これらの学習項目から「変数に値を入れるとき，変数を使った計算式で入れることができる」ことは必ずしも自明ではない，という判断がされた．

5.4.2 De-gapper が構文要素を検出しなかったもの（1個）

図 13 に示すプログラム（ex333j）からは，「switch において，case ラベルを case k:case l:case m ··· のようにまとめて書くことができる」という学習項目が提示されたが，これは De-gapper によってメジャーおよびマイナーな構文要素としてはあげられなかった唯一の学習項目であった．

switch 文の例は図 13 のプログラムより前に出現して，「switch 文の中に case がある」という構文要素はすでに検出されていた．この学習項目は，1 つの switch 文に同レベルに並んでいる case がいくつ並んでいるか，に関する指摘であった．De-gapper の現時点での実装は，case 文の並びの個数については検出を行っていないため，検出されなかった．

6. 考察

ここでは，De-gapper の目的である「飛躍の検出」をどれだけできたかを考察する．

```
int main(void) {
 int x;
 printf("月を入力してください \n");
 scanf("%d", &x);
 switch (x){
  case 1: case 3: case 5: case 7: case 8: case 10:
    printf("31 日 \n");
    break;
  case 4: case 6:case 9:case 11:
    printf("30 日 \n");
    break;

  default: printf("28 日 \n");
    break;
 }
 return 0;
}
```

図 13 プログラム ex333j
Fig. 13 Program ex333j.

6.1 構文要素数の推移による飛躍の推定

De-gapper が検出した構文要素と，人手で検出した学習項目は 1 対 1 で対応するとは限らないため，構文要素の個数がそのまま学習項目の個数になるわけではない．しかし，連続するプログラム列において，検出された構文要素の個数が急激に増えた場合，そこに飛躍がある可能性が高い．

そこで，前章の教科書のプログラムを先頭から見ていったときに，構文要素と学習項目の個数が変化する様子を**図 14** に示す．このグラフから，学習項目が増えている箇所，すなわち飛躍のある箇所で，構文要素の個数も増えていることが分かる．

De-gapper のユーザ（教員など）は，De-gapper が検出した構文要素の個数が多いところでは，解説を丁寧に行うなどの対応が必要であることを，前もって知ることが可能となる．

6.2 検出漏れ・過剰な検出への対策

De-gapper によって出力された構文要素と，学習項目を目視で確認した学習項目とを比べた結果，構文要素 140 個に対して 125 個は何らかの学習項目と対応しており，また，学習項目 78 個のうち 70 個は何らかの構文要素と対応し，構文要素と学習項目は，ともに 9 割程度が対応関係にあることが分かった．

ここでは，検出されなかった 1 割程度の学習項目を正しく検出し，過剰に検出された 1 割程度の構文要素の検出を抑えるための方策を議論し，飛躍の検出性能を高めることを考える．

6.2.1 構文定義をカスタマイズする

5.3.3 項で述べたように，見かけ上似ているものが，構文

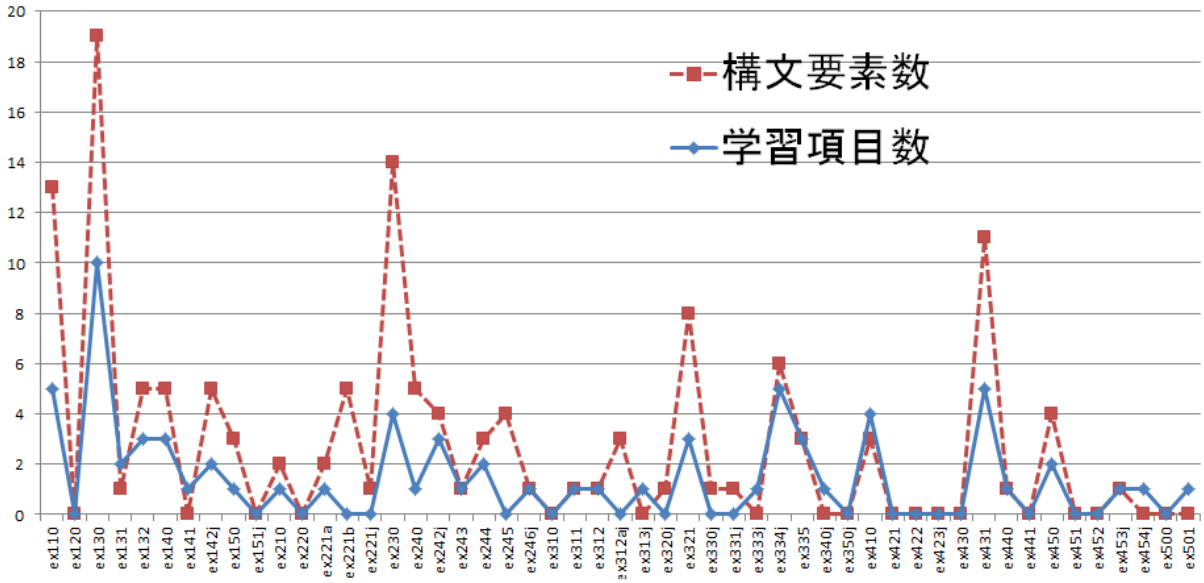

図 14 学習項目と構文要素の個数比較
Fig. 14 Comparison of learning elements with De-gapper output.

上は別の要素として定義されている場合，構文要素を過剰に提示してしまうことがある．

これは，今回の De-gapper の実装に組み込まれている構文定義によって起こる問題であり，実装を見直すことによって改善できる場合がある．

5.3.3 項の場合，「else のある if 文」「else のない if 文」という 2 つの構文ではなく，「if 文」という構文が 1 つだけあり，「if 文には else がつく場合とつかない場合がある」という定義をしておけば，冗長な提示を避けることができる．

また，5.4.2 項に示した，switch 文中の case の並び方についての学習項目を検出できなかった問題については，case が複数並んでいることを検出できるように構文定義を改めることが可能である．具体的には，「case n:」が複数並んだもの（case-list）という文を定義し，これを現在の実装の case 文の定義と置き換える．すると，case-list が含まれるパスは，case が 2 つ以上並んだときに初めて検出される．

6.2.2 学習者の前提知識を用いてフィルタを行う

5.3.1 項であげたような，「数学と同じ記法の式については検出しない」というように，学習項目としてあげるべきかそうでないかは，学習者がすでに持っている知識に依存する場合もある．このため，一部の構文要素については検出をしないようにするフィルタ機能があることが望ましい．

6.2.3 学習順序の入れ替えを提案する

5.3.2 項で述べたような，複雑な構文木を持ったプログラム (A) の後にそれより単純な構文木を持ったプログラム (B) が初めて提示される場合，(B) は自明に理解できるプログラムということができる．

ただし，これは (A) の内容が理解できていれば，という前提である．(A) の難しいプログラムを先に習い，(B) の易しいプログラムを後に習う，という順序では，(A) の時点で内容が理解できない可能性があり，本来なら (B)，(A) の順序で習うことが望ましい．

そこで，De-gapper の機能として，複雑な構文木が最初に提示され，それより後に単純な構文木が提示された場合，提示の順番を入れ替えることを提案するような機能を追加することが望ましい．

6.2.4 マイナーな構文要素の重み付けをする

マイナーな構文要素とは，「学習者にわざわざ教えなくても自明に理解するはずの構文要素」であるが，5 章であげたとおり，マイナーに分類された構文要素でも実際には教えることが望ましいものがあった．

ある構文要素がマイナーであるかどうかは，それが既存の構文要素から理解可能かどうかによっているが，既存の構文要素が，今学習しているプログラムよりどの程度前にあげられているか，また，その既存の構文要素がこれまでどの程度の頻度で使われているか，などによっても理解の難易度は変わる可能性がある．

また，マイナーな構文要素（A/B/C）は，基本的に 2 つの別の構文要素（A/B と，B/C で終わるもの）を根拠にしているが，それら自身がまたマイナーな構文要素である場合，それらを根拠とする構文要素をさらに順番にたどって，2 個を超える構文要素を根拠に理解をしなければならないことも考えられる．

これらのことを勘案して，マイナーな構文要素においても重み付けを行い，必要に応じて提示をする必要があると考えられる．

6.3 出力形式について

現在の De-gapper の出力は，ソートされた構文木のパス名だけを出力しているため，それぞれのパスがプログラム

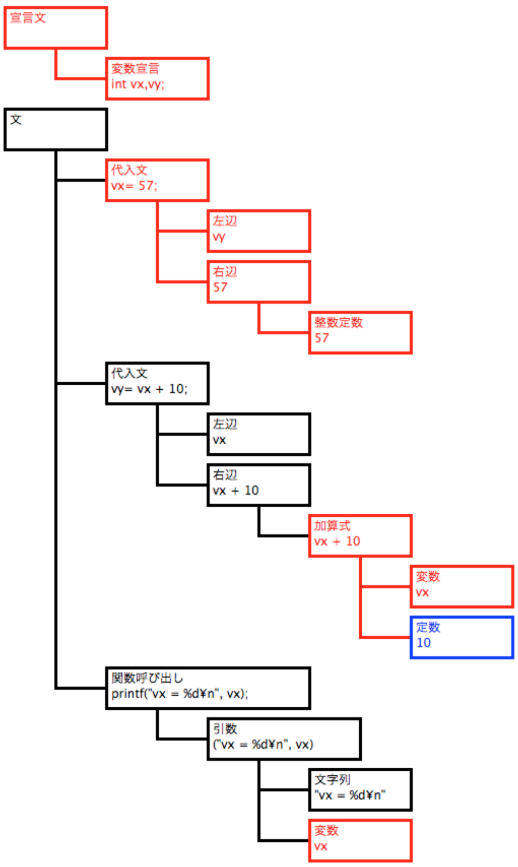

図 15 構文構造のグラフィカル表示
Fig. 15 Graphical presentation of syntax structure.

のどの部分に対応しているか分かりにくいし，構文解析の経験がない利用者がその結果を読み取ることは難しい．

そこで，利用者に提示する出力形式として，2 つの方法を考えている．

構文を視覚的に表示

1 つ目の出力形式は，**図 15** のように，構文のパスを木構造としてグラフィカルに表示し，検出部をマークする方法である．

図 15 は，図 7 の var.c の検出結果を表示したものである．const.c で検出済みのものも含めて，構文の階層構造を木として示し，メジャーな新しい構文要素を赤で，マイナーな新しい構文要素を青で示している．構文要素を表すノードにはあわせて，検出されたプログラムの該当部分も提示するようにしている．また，前節で考察した問題点である「学習者に見えない学習項目まで提示される」ことがないよう「変数宣言」以下の概念として「宣言並び」や「初期化宣言子」は表示しないようにしている．

この表示方法は，構文の構造が容易に把握できるようになっているので，飛躍を埋めるための中間の例題を作る際に，どのような構文要素を新出とすべきかを検討する場合に有用であると考える．

プログラムリストに埋め込んで表示

2 つ目の出力形式は，**図 16** のように，プログラムリストの中該当部分にそこで新出する概念をコメントとして埋め込むものである．

```
int main(void) {
 /* 宣言文-変数宣言 */
 int vx,vy;
 /* 代入文-左辺,右辺-整数定数 */
 vx= 57;
 /* 右辺-加算式-変数,(整数定数) */
 vy= vx + 10;
 /* 引数-変数 */
 printf("vx = %d\n", vx);
 printf("vy = %d\n", vy);
}
```

図 16 プログラムに埋め込んで表示
Fig. 16 Embedded presentation in source program.

図 16 は，図 15 と同様，図 7 の var.c の検出結果を表示したものである．検出した概念は該当する箇所の前にコメントとして構文の既出の部分を除いた階層構造のパスを日本語で表示している．なお，マイナーな新出の構文については ( ) を付けて表示している．また，プログラム自身の文字の色もメジャーな新しい構文要素は赤，マイナーな新しい構文要素は青となるようにしている．

この表示方法は，プログラムの中でどの部分が新出の概念であるかを容易に把握できるという利点を持っている．

7. まとめ

本論文では，プログラミング学習において，一度にたくさんの概念を学習させるプログラムがあると学習者がつまずく原因になると考え，そのような「飛躍」のあるプログラムを自動的に検出するツール De-gapper を提案した．De-gapper では，プログラムの表記から分かる学習項目だけを対象とした場合に，それらの学習項目のうち 9 割程度は機械的に発見でき，教材を設計する段階で飛躍を前もって検出するための有用なツールであることを確認した．

今後は，構文だけでは過剰に検出したり，検出できなかったりした学習項目を正しく検出できるよう，ツールを発展させることを考えている．

参考文献

[1] Lister, R., Adams, E., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Mostrom, J., Sanders, K., Seppalla, O., Simon, B. and Thomas, L.: A Multi-National Study of Reading and Tracing Skills in Novice Programmers, *SIGSCE Bulletin*, Vol.36, pp.119–150, ACM (2004).
[2] McCracken, M., Almstrum, V., Diaz, D., et al.: A Multi-national, Multi-institutional Study of Assessment of Pro-

gramming Skills of First-year CS Students, *SIGCSE Bulletin*, Vol.33, No.4, pp.125–180, ACM (2001).
[3] Soloway, E., Ehrlich, K., Bonar, J. and Greenspan, J.: What do Novices Know about Programming?, *Directions in Human-computer Interactions*, Norwood, NJ, Ablex, pp.27–54 (1982).
[4] Lopez, M., Whalley, J., Robbins, P. and Lister, R.: Relationships between Reading, Tracing and Writing Skills in Introductory Programming, *ICER '08*: *Proc. 4th International Workshop on Computing Education Research*, pp.101–112 (2008).
[5] 山本三雄, 関谷貴之, 山口和紀：プログラミングのスキル階層に関する研究, 情報処理学会研究報告, Vol.2010-CE-104, No.3, pp.1–25 (2010).
[6] 内藤広志, 齊藤 隆：プログラミング演習のための進捗モニタリングシステム，情報処理学会研究報告，Vol.2008-CE-093, No.5, pp.33–40 (2008).
[7] 谷川紘平，フォンディン ドン，原田史子，島川博光：C 言語関数呼出しの記録を用いた演習過程での習得項目の把握，電子情報通信学会論文誌 D，情報・システム，No.12, pp.2079–2089 (2012).
[8] 平澤智明，高野辰之，宮川 治，北澤由貴，古澤資栄，小澤諒：プログラミングの入門教育を対象とした概念学習システムの開発，情報処理学会研究報告，Vol.2010-CE-107, No.8, pp.1–8 (2010).
[9] 長谷川伸，松田承一，高野辰之，宮川 治：プログラミング入門教育を対象としたリアルタイム授業支援システム，情報処理学会論文誌，Vol.52, No.12, pp.3135–3149 (2011).
[10] 加藤利康，石川 孝：プログラミング演習支援システムにおける学習状況把握機能の提案，情報処理学会研究報告，Vol.2013-CE-120, No.2, pp.1–8 (2013).
[11] Kernighan, B. and Ritchie, D.: *C Programming Language* (*2nd Edition*), Prentice Hall PTR (1988).
[12] 浅井宗海：情報処理教育標準テキストシリーズ「C 言語」，実教出版 (1995).

付 録

A.1 De-gapper が解析する C 言語の構文

De-gapper が解析する C 言語の構文と，K&R [11] の付録 A.13 に示されている構文は次のような違いがある．

(1) 字句要素は，それぞれをタグで囲む．タグ名には，定義されている名称（identifier，integer-constant など）があればその名称を，予約語（`int` など）にはその予約語と同名を，記号には適当につけた名前（`+` には plus など）を用いる．ただし，制御構造に用いられる予約語（`if`，`else`，`while`，`for`，`do`，`switch`，`case`，`default`，`break`，`continue`，`return`）はタグで囲まない．

(2) 構文要素の名前に略称を用いる（statement を stmt，expression を expr など）．また，一部の式の名前から expression を省略する（addive-expressin は単に additive）．

(3) 繰返し構造はネストしない．たとえば，statement-list は，「statement がいくつか並んだもの」という構造を「statement-list と statement を 1 個ずつ並べたもの」と表現している．しかし，De-gapper では，statement-list の要素の内部には statement-list の要素をネストさせず statement の要素を同じ階層に複数並べた構造を生成させている．

(4) 関数定義（function-definition）の直下の要素である compound-statement と，それに含まれる statement-list は，それぞれ f-compound-statement と f-statement-list という名前の要素を生成し，他の compound-statement と区別する．

(5) 式の構文については，子要素が 1 つしかない場合は，その子要素のみを生成する．たとえば，`42` という式は，実際には expression であり，assignment-expression でもあり，conditional-expression でもあり，さらに下位の式の要素も含んでおり，素直に構文に従って要素を生成した場合は，これらの要素を大量にネストしたものが生成される．De-gapper では，これを単に `<int-const>42</int-const>` と生成する．

(6) 代入式（assignment-expression）は，左辺を `<left>...</left>`，右辺を `<right>...</right>` で囲む

(7) 後置式（postfix-expression）は，一次式（primary-expression）そのものか，それに関数の引数，後置インクリメント，後置デクリメントをつけたものであり，それぞれ function-call, postfix-increment, postfix-decrement という名前（タグ名）の要素を生成する．function-call については，関数名を `<name>...</name>`，引数を `<arguments>...</arguments>` で囲む．配列の要素アクセスと，構造体のメンバアクセスは現時点の実装では対応していない．

(8) 同じ優先順位の 2 項演算はネストしない．たとえば，前述のルールに従えば，`12+34+56` の XML 木は

```
<additive>
  <additive>12+34</additive>+56
</additive>
```

であるが，De-gapper は，これを

```
<additive>12+34+56</additive>
```

と生成する．

(9) selection-statement は，それぞれ if（else なし），if-else（else あり），switch に分ける．

(10) iteration-statement は，while 文，do 文，for 文のいずれかであり，それぞれ while, do, for という名前（タグ名）の要素を生成する．

(11) jump-statement は，continue 文，break 文，return 文のいずれかであり，それぞれ continue，break，return という名前の要素を生成する．goto 文は解析の対象から外す．

(12) labeled-statement は，case 文，default 文のいずれかであり，それぞれ case，default という名前の要素を生

成する．goto のラベル定義は解析の対象から外す．

長 慎也 （正会員）

2001 年早稲田大学大学院理工学研究科情報科学専攻修士課程修了．2005 年早稲田大学大学院理工学研究科にて博士（情報科学）の学位を取得．2006 年より一橋大学総合情報処理センター助手．2010 年より明星大学情報学部准教授．プログラミング教育，プログラミング言語の開発に関する研究に従事．2005 年情報処理学会山下記念研究賞受賞．電子情報通信学会，教育情報システム学会，ACM，IEEE 各会員．

保福 やよい （正会員）

1984 年御茶の水女子大学理学部数学科卒業．1984 年神奈川県立高等学校教員として勤務，現在に至る．2013 年大阪電気通信大学医療福祉工学研究科博士後期課程入学．

西田 知博 （正会員）

1991 年大阪大学基礎工学部情報工学科卒業．同大学大学院基礎工学研究科を経て，1996 年同大学情報処理教育センター助手．2000 年大阪学院大学情報学部講師．2010 年から同大学准教授．プログラミング教育および情報教育に関する研究に従事．ACM，電子情報通信学会，情報科教育学会各会員．

兼宗 進 （正会員）

1987 年千葉大学工学部電子工学科卒業．1989 年筑波大学大学院理工学研究科修士課程修了．2004 年筑波大学大学院ビジネス科学研究科博士課程修了．博士（システムズ・マネジメント）．企業勤務後，2004 年から一橋大学総合情報処理センター准教授，2009 年から大阪電気通信大学医療福祉工学部教授．2013 年から同大学総合情報学部教授．プログラミング言語，情報科学教育に興味を持つ．ACM，IEEE Computer Society 各会員．