Vol.2013-CE-118 No.21
2013/2/9

プログラミング初学者の段階的な理解モデルの検討とツールによる支援

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

概要：初学者がプログラミングを学ぶ際の学習について、段階的に少しずつ学習を進めるスモールステップの概念に着目し 2 つの提案を行う。ひとつは、繰返しなどの基本的な概念について、段階的に理解を進めるための理解構造の提案である。もうひとつは、教科書や授業等の例題が適切な難易度で提示されているかを検証するツールの提案である。実際にいくつかの市販の入門書を対象に調査し、説明の順序と理解構造を対比した。また、掲載された例題について、一度に多くの新出概念が出現していないかどうかを検証した。

Study on a “small steps” learning model for novice programmers and a supporting tool for detecting “learning gaps”

Abstract: In this paper, we make two proposals for novice programming learners. One is a model for an understanding process that learners can use while studying programming elements, e.g. loop structure. We focus on the “small step” method, in which students learn few concepts per one program to avoid having trouble with learning programming. The other is a tool named “De-gapper ”, which detects new syntax elements of each program in programming textbooks. We think “gaps”; a lot of concepts to be learned in a program prevent learners from understanding programming. Therefore we analyze the difference in the description order between several C programming textbooks on the basis of our model with De-gapper .

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上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、習得しなければいけないことが多く、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]。一方、教える側は自分が大きな困難を感じずにプログラミングができるようになっている場合が多いので、簡単に思えるところで、学生たちがつまずく理由の理解が難しくなる。

そこで我々は、「一度に学ぶ内容が多すぎ、初学者の理解を超えている」ことがつまずきの原因のひと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スモールステップの考え方は学校教育などで取り入れられているが、我々はこの考え方をプログラミングに適用することにした。そして、学習内容がスモールステップになっていることを確認するために、教科書の例題や授業で使用されるサンプルプログラムを評価する差分分析ツール De-gapper を開発した。

本稿ではスモールステップの考えに基づいたプログラミングの学習モデルの提案を行う。また、その学習を支援する De-gapper の概要と、De-gapper を授業で用いられている入門書に適用した結果を報告する。また、De-gapper を使うことでのメリットについて述べていく。

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

[1] 神奈川県立相模向陽館高等学校
Sagami Kouyoukan High School
[2] 大阪電気通信大学
Osaka Electro-Communication University
[3] 明星大学
Meisei University
[4] 大阪学院大学
Osaka Gakuin University
[a] hohuku@amy.hi-ho.ne.jp

© 2013 Information Processing Society of Japan

コースを終えた学生の 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 [2] 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、多くの学生が得点が低く、理解ができていないことがわかった [1]。

このようにさまざまな研究がなされているが、初学者に対するプログラミングの教育はいまだ試行錯誤が繰り返され、決定的と言える教育法は確立されていない。[12] [13]

学習者に合わせて少しずつ学ぶことができるコンピュータ支援教材（CAI）や e-ラーニングなどに影響を与えた心理学者 Skinner は、教育目標の到達に向かって既習事項を復習しながら、学習内容を段階的に配置して提示するプログラム学習理論を提唱した [5]。

たとえば、２次不等式を公式を使って解き方を教えるのは簡単であるが、丸暗記で覚えたものは意味を理解せず、定着しづらい。そこで、高校数学では、（１）２次関数のグラフが描ける（２）２次方程式の解と２次関数の関係を理解する（３）２次関数のグラフを使って、視覚的に２次不等式を解く、という手順を踏み教えることが多い。このような手順を踏み、別の単元との関連付けることで、理解の定着は大いに増す。（１）については「$y = ax^2$ のグラフが描ける」（２）については「２次方程式が解ける」という前提条件が必要になる。「２次方程式を解く」には、「因数分解ができる」「代入できる」「解の公式が書ける」などの前提条件が必要となり、学習内容が段階的に配置され、理解構造になっている。

この理論は日本ではスモールステップという名前で、親しまれ、数学や語学など積み上げ学習を必要とする教科を中心に様々な教育現場で活用されている。そこで、我々はスモールステップの考え方をプログラミング学習に適用することを考えた。

ete the task correctly (even when syntax errors were ignored).”[3]

3. スモールステップの考え方

Winslow は、「初学者はスキルが少なく、知識も関連づいていないため、プログラムを 1 行ごとに解読をするしかないが、プログラミングの熟達者はプログラムを意味のある小さな塊（チャンク）として捉えることができ、一度に多くの量のプログラムを読むことができる」[4] と述べている。

プログラムの中に一度に多くの初出事項が出てくると、初学者の短期記憶能力を超えてしまい、プログラムを扱うことができずに理解が困難になる可能性が考えられる。これを改善するためには、理解に合わせて細かいステップを設定し、学んだことを何度も復習しながら、少しずつ新しい概念を含めるスモールステップの考え方が有効である。例題ごとのプログラムの初出事項を制限し、学習のステップを細かく配置することで、初学者にとって学びやすくすることが期待できる。**図 1** に、スモールステップの概念図を示す。

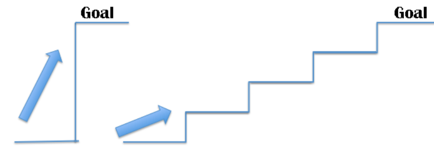

図 1　スモールステップによる学習

4. スモールステップを実現する手立て

プログラミングを教える教員や入門書の著者は多くの知識があるので、多くのことを教えてしまいがちである。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 2 に、C 言語での例題の飛躍の例を示す。最初に文字列を表示する左の例題を扱った後で、教員は数も表示できることを示すつもりで（教師にとっては簡単な）右の例題を示している。左の例題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の表示 1 個だけである。

しかし実際には、右の例題には以下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`
- 変数への定数の代入: `n = 10`
- 算術演算: `n+20`
- 書式指定: `%d`
- 関数の複数の引数: `printf("",n+20)`

```
printf("Hello");
```

```
int n = 10;
printf("%d", n+20);
```

図 2　複数の概念が出現する例題の例

たった 2 行のプログラムでも、どのような概念が新たに出現しているかを明確に認識することは容易ではない。私たちはこのような、著者や教員が意図しない例題間のギャップに着目した。

次に、主に初学者がつまずきやすい「繰返し」に焦点を当てて分析する。

C での繰返しは複雑である。**図 3** に「for」と「while」を使った繰返しの例を示す。

このプログラムでは、画面に「Hello」を 10 回出力している。これらのプログラムを理解する前提としては、「変数」「数値のデータ型 (int)」「変数の初期化」「値の代入」

© 2013 Information Processing Society of Japan

```
int i;
for (i = 0; i < 10; i++) {
    printf("Hello");
}
```

```
int i = 0;
while (i < 10) {
    printf("Hello");
    i++;
}
```

図 3 C の繰返しの例 (for, while)

「比較演算子」「真偽値」「インクリメント」の概念をあらかじめ理解している必要がある。

C などの汎用言語では、繰返しの構文として「for」「while」「do…while」など、複数の構文が用意されていることが多い。したがって、「1 つの構文が定着しないうちに、次の構文を学習すること」、「構文の選択肢の多さ」、「理解する必要のある前提」など初学者にとって極めて負担が多い。そこで、教える側は多くの例題や演習をさせるが、演習の内容が多いので学習者はかえって混乱する。

また、繰返しでは「繰返しと if の組合せ」、「複数の繰返し」など急に内容が高度になり、さらに難しさに拍車をかけている。

繰返しを教える際に、固定回数の繰返しが教えやすいということで「for」から教える考え方もある。しかし、「for(i=0; i<10; i++)」はひとまとまりになった形で一度に多くの概念が出てくるため、個々の仕組みがわかりにくい。

for などの繰返しをスモールステップで理解するためには、学習を細かい小さなステップに分割する必要がある。繰返しの構文の理解構造を**図 4** に示す。それぞれの項目の下には、その概念を理解するために必要な事前知識が示されている。

while を理解するためには、ループの意味、条件式、変数の理解が必要である。条件式を理解するために if 文で比較演算などを事前に学習する。また、while 文を学習する前に、変数の代入、インクリメントなどを学習しておく必要がある。テキストもこのような順番で記述してあるとスモールステップが実現でき理解しやすい。

具体的には、「条件式」「インクリメント」「if」を教えてから、「while」を教える。そして学生がすべて理解した後、「for」を教える。このような順序にすることでスモールステップが実現できる。

また、前述の図 2 の例は、**図 5** の左の例を示してから右の例を示すことで、学生は書式指定子を既に学んでいるので、ギャップを小さくできる。

5. C の教科書の分析

仮説を確認するために、私たちは C の教科書で使われている例題を分析することにした。

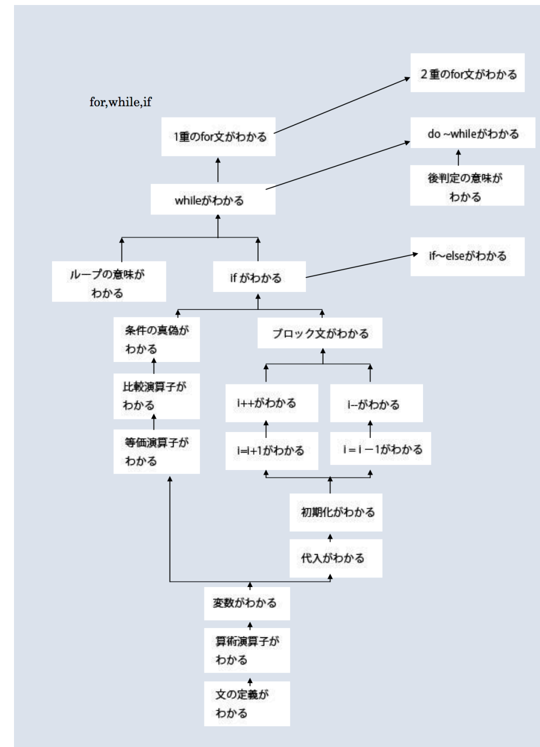

図 4 繰返し構文の理解構造

```
int n = 10;
printf("%d", n);
```

```
int n = 10;
printf("%d", n+20);
```

図 5 ギャップの小さい例題を追加する例

5.1 入門書の分析

入門書の例として、5 冊の教科書 [7][8][9][10][11] を学習の順番に注目して分析した。**表 1** に、それぞれの教科書の順番を示す。

表 1 5 冊の教科書の分析

A	B	C	D	F
変数	変数	変数	変数	変数
…	型	…	型	代入
代入	代入	代入	型	代入
式	配列	算術式	算術式	配列
…	文字列	キャスト	キャスト	算術式
型	式	条件式	…	複合式
if	if	if	for	if
switch	for	switch	while	while
do	while	for	do	do
while	do	while	if	for
for	switch	do	switch	switch

結果を見ると、for を中心とする制御構造の出現順序に関する 2 つの違いが観察された。ひとつは、理解構造を積み重ねていく学習から外れた概念を扱っていることである。具体的には、学習の流れから見て必須ではない概念を、言

語仕様の説明のために扱ってしまっている。たとえば、B ではデータ型の説明において配列を扱っており、演算子についても C の演算子をすべて説明している。

もうひとつは、説明の順序が理解構造の積み重ねと合致していないものである。たとえば、B、C、D では for の後で while を扱っており、for を学習するために必要な「条件比較」if と「繰返し」を理解するための while が、for の前に扱われていないので、学生は理解して学ぶことができない可能性がある。

```
if (vx % 5)
  puts("その数は 5 で割りきれません.");
```

図 6 教科書 A での if 文を使った最初の例題

また、A では、**図 6** に示すように条件分岐 if の条件式が「x%5」のような形から説明されており、難解である。ここでは、「if の条件式の意味」と「真偽値としての整数値の解釈」という 2 つの概念が同時に出現している。**表 2** の代替案のように、比較演算子を用いた一般的な条件比較の概念を学んだ後に、C 言語に特有の「真偽値と 0 の関係」を学んでいくほうが理解しやすいと思われる。その後、「C 言語におけるすべての整数値は真偽値を持つ」ということを学ぶことができる。

表 2 比較演算子と条件分岐の説明順の例

No.	教科書 A	代替案
3-1	if (x% 5)	if (x1 == x2)
3-2	if (x% 2)	if (x1 > x2)
3-3	if (x% 5) else	if (x1 == x2) else
3-4	if (x% 2) else	if (x != 0) else
3-5	if (x) else	if ((x%2) !=0)
3-6	if (x1 == x2) else	if (x%2) else

また、B では**図 7** に示すように繰返し while の条件式が「dt[i] ! = -1」から始まる。等価演算子「!=」「i 番目の配列の要素」「while の条件式の意味」「配列の繰返し」という 4 つの概念が新しく扱われている。表 2 の代替案のように、if などで等価演算子「!=」を学んだ後、配列を伴わない条件式で while を扱い、その後配列を扱うことができる。

```
while(dt[i] != -1){
  printf("%d\n", dt[i]);
  i++;
}
```

図 7 教科書 B での while 文を使った最初の例題

このような点を意識することにより、市販のテキストを使う場合にも、言語の理解構造に合わせて説明順序を入れ替えたり、他のプログラムを補足することで、学習者がスムーズに理解できる授業が可能になる可能性がある。

実際にテキストで学ぶ学習者や授業で使う教授者にとって、「学びやすい」または「教えやすい」順序がそれと同じかどうかが本研究の興味である。

6. 教科書のギャップ（飛躍）を見つけるツール

先に述べたように、ほとんどの教科書にはギャップが存在し、学生や教える側はギャップで難しさを感じる。私たちは、言語の理解構造に合わせて説明順序を入れ替えたり、他のプログラムを補足することで、ギャップを埋めることを提案する。しかし、いくつかの問題がある。

- 教科書のギャップを見つけるのには時間がかかる。
- ギャップがなくなったことを確認するのは簡単でない。

これらの問題を解決するために、教科書の中のプログラムの新しい構文要素を発見する De-gapper というツールを開発した。

6.1 ツールの概要

De-gapper に教科書のプログラムを読み込むとき、教科書に記載されている順番で一度に読み込む必要がある。それぞれのプログラムを De-gapper はチェックし、初出の概念が現れたら表示する。**図 8** に De-gapper でのプログラムの入力とレポートの出力の様子を記す。

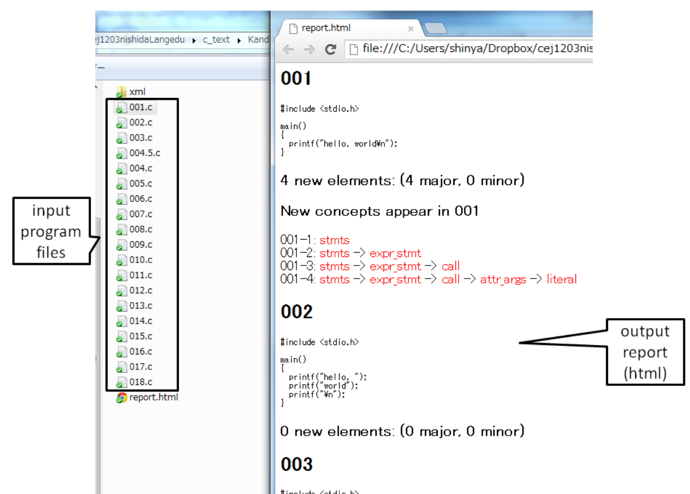

図 8 De-gapper の入力と出力の例

もし、1 つのプログラムの中にあまりに多くの新しい概念が含まれているなら、そのプログラムは学生にとって難しいと考えられる。

このツールは教科書を使ってプログラミングを教える教員の役に有用である。De-gapper を使うと教科書のどの部分が難しいか教えてくれ、教える側は難しい部分について補った説明をすることができる。また、宿題やテストを出す際にも教えていない概念を出題していないか確認するこ

とができる。

このツールは教科書の著者にも有用である。De-gapper は教科書の中に含まれるすべての概念を書き出すので、著者は教科書の中にすべての概念が説明されているか確認することができる。

6.2 実装

De-gapper は “parser” と “checker” という 2 つのプログラムから構成されている。parser は教科書のすべてのプログラムファイルを分析し、XML ファイルの構文木（以下、XML 木）として書き出す。**図 9** に parser の入力と出力例を示す。図 9 に示した部分はプログラムのすべてと関数定義のタグを含んでいない。ここで論じているのが、ただ 1 つの関数なので、私たちはそれ以外のタグを省略することにする。

checker はすべての XML 木ファイルを教科書に書かれている順番で分析する。それぞれの XML 木のファイルごとに、タグの階層構造のパスを分析する。たとえば、checker は図 9 の XML 木ファイルを次のように表示する。

```
list0101-1: stmts
list0101-2: stmts -> expr_stmts
list0101-3: stmts -> expr_stmts -> call
list0101-4: stmts -> expr_stmts -> call -> attr_args -> add
list0101-5: stmts -> expr_stmts -> call -> attr_args -> add -> num
list0101-6: stmts -> expr_stmts -> call -> attr_args -> add -> op
list0101-7: stmts -> expr_stmts -> call -> attr_args -> literal
```

この例で、De-gapper はプログラムに含まれる “list0101-1” から “list0101-7” の概念を出力している。これらはプログラムに表示される順番ではなく、タグの階層構造のパス順になっている。

もし、前に分析した XML 木ファイルにまだ現れていないパスがあれば、そのパスは XML 木ファイルに応じてそのプログラムの「新しい概念」として検出される。

現在は C と Java に対応しているが、別の言語に対応した parser を加えれば、他の言語にも簡単に対応することができる。

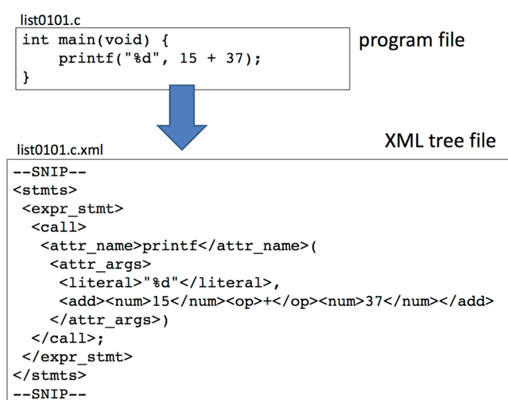

図 9 parser の入力と出力の例

7. 分析例

7.1 例 1- Small Example:

分析例として図 9 に示した “list0101.c” を最初のプログラムとする教科書を取り上げる。De-gapper はプログラムのすべてのパスを新しい概念として検出する。それは次のようになる。

- `list0101-1`: 文の最初の例
- `list0101-2`: 式文の最初の例
- `list0101-3`: 文の関数呼出の最初の例
- `list0101-4`: 関数呼出の中の加法式の最初の例
- `list0101-5`: 加法式の中の数の最初の例
- `list0101-6`: 加法式の中の + 演算子の最初の例
- `list0101-7`: 関数呼出の中の書式指定子の最初の例

この教科書の次の例題は**図 10** に示す “list0102.c” である。このとき、De-gapper は次にように “list0102-1” から “list0102-11” という項目で新しい概念を検出する。

- `list0102-1` から `list0103-3`: 変数の宣言の最初の例
- `list0102-4`: 関数呼出の中の変数の最初の例
- `list0102-5`: 代入の最初の例
- `list0102-6`: 代入の左辺が変数である最初の例
- `list0102-7`: 代入の右辺が加法式である最初の例 (`vy=vx+10;`)
- `*list0102-8`: 加法式の中が数字である最初の例 (`vy=vx+10;`).
- `*list0102-9`: 加法式の中の+ 演算子の最初の例 (`vy=vx+10;`).
- `list0102-10`: 加法式の中の変数の最初の例 (`vy=vx+10;`).
- `list0102-11`: 代入式の右辺に数字が数字である最初の例 (`vx=57;`).

新しい概念の中で “list0102-8” や “list0102-9” のように ＊が先頭についているものは、完全なる新しい概念とはみなさない。なぜならば、加法式はすでに `list0101-4` で学習しており、`list0101-5` と `list0101-6` に示す通り、加法式の中で+演算子や数を使うことも学習済みである。よって、代入式の左辺ではあるが、加法式の中で+演算子や数を使えることは容易に類推することができる。

このような新しいが類推できる概念を “minor” の新しい概念、残りを “major” の新しい概念と呼ぶことにする。

7.2 例 2 - K & R :

別の例として「プログラミング言語 C」[6] の 1.1 から 1.6 節までを De-gapper で処理した。結果を**表 3** に示す。

De-gapper を使うことで No.3 は 26 個の major の新しい概念が存在し、極めて多くのギャップを持つことがわ

© 2013 Information Processing Society of Japan

list0102.c

```
#include  <stdio.h>
int main(void) {
    int  vx, vy;
    vx = 57;
    vy = vx + 10;
    printf("vx = %d¥n", vx);
    printf("vy = %d¥n", vy);
}
```

```
list0102-1: decls
list0102-2: decls -> decl
list0102-3: decls -> decl -> attr_declarators -> declarator
list0102-4: stmts -> expr_stmt -> call -> attr_args -> var
list0102-5: stmts -> expr_stmt -> let
list0102-6: stmts -> expr_stmt -> let -> attr_left -> var
list0102-7: stmts -> expr_stmt -> let -> attr_right -> add
*list0102-8: stmts -> expr_stmt -> let -> attr_right -> add -> num
*list0102-9: stmts -> expr_stmt -> let -> attr_right -> add -> op
list0102-10: stmts -> expr_stmt -> let -> attr_right -> add -> var
list0102-11: stmts -> expr_stmt -> let -> attr_right -> num
```

図 10 2 つ目の例題（変数が初めて登場する）

No.	節	新しい概念 (major/minor)	最初の例:
1	1.1	4/0	`printf("Hello, world\n");`
2	1.1	0/0	複数の `printf`
3	1.2	26/4	`int` 変数 and ”while” 文
4	1.2	1/2	`float` 変数
5	1.3	6/22	”for” 文
6	1.4	1/0	`#define`
7	1.5.1	2/0	`c=getchar();` (関数呼出しの戻り値)
8	1.5.1	3/4	`c=getchar()` ”while” 文での条件式における
9	1.5.2	3/1	インクリメント `++`
10	1.5.2	3/1	空文を使った for 文
11	1.5.3	3/5	if 文と文字定数
12	1.5.4	10/24	else と `||` 演算子
13	1.6	15/17	配列と `&&` 演算子

表 3 K & R book の 1.1 から 1.6 までの新しい概念

かった。この例題の前には多くの補完するプログラムが必要であるが、このギャップは De-gapper を使わなくても明らかであるため、ここではそれについて論じないこととする。ここでは、新しい概念の数が 1 から 3 というように、プログラムのギャップの数が少なく、判断が難しいと考えられる No4. から No.11 までに焦点をあてて考える。

No.5 は新しい概念の数が 6 とややギャップが高い。**図 11** にプログラム No.4 と No.5 を示す。No.5 は for 文の最初の例題であり、次のような major な新しい概念がある。

```
005-1: stmts -> for
005-2: stmts -> for -> attr_condition -> comp
005-3: stmts -> for -> attr_condition -> comp -> num
005-6: stmts -> for -> attr_initializer -> let
005-12: stmts -> for -> attr_loop -> expr_stmt -> call
       -> attr_args -> mul
005-23: stmts -> for -> attr_increment -> let
```

```
#include <stdio.h>
/* print Fahrenheit-Celsius table
 for fahr = 0, 20, ..., 300; floating-point version */
main()
{
  float fahr, celsius;
  float lower, upper, step;

  lower = 0; /* lower limit of temperatuire scale */
  upper = 300; /* upper limit */
  step = 20; /* step size */

  fahr = lower;

  while (fahr <= upper) {
    celsius = (5.0/9.0) * (fahr-32.0);
    printf("%3.0f %6.1f¥n", fahr, celsius);
    fahr = fahr + step;
  }
}
```

No.4

```
#include <stdio.h>
/* print Fahrenheit-Celsius table */
main()
{
  int fahr;
  for (fahr = 0; fahr <= 300; fahr = fahr + 20)
    printf("%3d %6.1f¥n", fahr, (5.0/9.0)*(fahr-32));
}
```

No.5

図 11 K & R 本の while 文のプログラムと、次に出てくる最初の for 文の例

No.5 のように for 文を学ぶときは、次の概念を扱わざるを得ない。

- 005-1: “for” 文自体
- 005-6: 文の初期化の部分
- 005-2: 文の条件の部分
- 005-23: 文の増分の部分

残りの新しい概念、005-3 と 005-12 は次の通りであるが、これらは for 文の本質ではない。

- 005-3: 条件式の中の数値の最初の例: `fahr <= 300` .
- 005-12: 関数呼出しの引数内の算術式の最初の例: `printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));`

教科書の著者は No.4（while 文）を書き換えたプログラムとして No.5 を書きがちであるが、新出の概念は for 文の本質的な部分に限るべきであると考える。また、No.4 は計算する範囲の変数 `lower`, `upper`, `step`, `celsius` を定義しているのに対して、No.5 はそれらは使われていない。これらを考慮し、No.4 と同一の内容を for 文で書き換えたプログラム例は**図 12** のようになる。もし、この例題が No.4 と No.5 の間にあったなら、No.4 から増える新しい概念は以下の 5 個に留まる。

```
004.5-1: stmts -> for
004.5-2: stmts -> for -> attr_condition -> comp
004.5-5: stmts -> for -> attr_increment -> let
004.5-10: stmts -> for -> attr_initializer -> let
004.5-13: stmts -> for -> attr_loop -> compound_statement
```

`004.5-13` は for 文のブロックに複数の文が含まれない No.5 にない新しい概念である。`004.5-13` は別として、残りの 4 つの新しい概念は for 文の本質的な概念に相当する。

ここで示した例のように、De-gapper は、確かに難しいと思われる場所については、学ぶ概念が多いと指摘していることがわかる。以上のことにより、De-gapper を使うと

Vol.2013-CE-118 No.21
2013/2/9

```
#include <stdio.h>
/* print Fahrenheit-Celsius table
 for fahr = 0, 20, ..., 300; floating-point version */
main()
{
  float fahr, celsius;
  float lower, upper, step;

  lower = 0; /* lower limit of temperatuire scale */
  upper = 300; /* upper limit */
  step = 20; /* step size */

  for (falr = lower ; fahr <= upper; fahr = fahr + step) {
    celsius = (5.0/9.0) * (fahr-32.0);
    printf("%3.0f %6.1f¥n", fahr, celsius);
  }
}
```

図 12 K&R 本の for 文の最初の例として提案する補完的なプログラム

次のことが期待できる。

- De-gapper を使うと多くのプログラムから自動的にギャップを発見することができる。
- De-gapper は教える側が教科書に隠れたギャップを見つける手伝いをし、degap（ギャップの間に補うプログラムを用意する）の機会を与える。
- 補うプログラムを入れた後で、De-gapper はギャップが小さくなったか再確認することができる。

8. まとめと今後の課題

我々はスモールステップの考え方によるプログラミング学習とギャップを見つけるツールを提案した。また、授業をする教員が手作業だと見落としがちな部分を、ツールを用いて発見できることを確認した。今後は、学習法とツールを発展させることを考えている。

プログラム中の新しい概念の数はギャップの高さに常に比例するわけではない。De-gapper でギャップの分析結果と学生が感じている実際のギャップを比較する実験を行い、ギャップ測定の精度を上げていくことが必要である。

現在の実装では、新しい概念の内容は XML のパスとして表示される。しかし、一般的に利用してもらうためには、それを理解しやすい自然言語で表示する必用がある。また、現在の実装ではパスとプログラム文の対応が示されていない。将来的には XML パスと新しい概念、プログラムの該当部が関連して表示されるようなインタフェースの改善を行う予定である。

さらに、学生の理解度を調査することで、理解構造図に出てくる構文要素の関係や階層構造を検証していく予定である。

参考文献

[1] Michael McCracken, Vicki Almstrum, Danny Diaz, et al. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. ACM, SIGCSE Bulletin, Vol.33, No.4, pp.125–180, 2001.

[2] Raymond Lister, Elizabeth Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Mostrom, Kate Sanders,Otto Seppalla, Beth Simon and LyndaThomas. A Multi-National Study of Reading and Tracing Skills in Novice Programmers. ACM, SIGSCE Bulletin, Vol.36, pp119-150, 2004.

[3] E Soloway, K Ehrlich, J Bonar and J Greenspan. What do novices know about programming?. Directions in human-computer interactions, Norwood, NJ, Ablex, pp.27-54,1982.

[4] Leon Winslow. Programming pedagogy - A psychological overview. SIGCSE Bulletin, Vol.28, pp17-22, 1996

[5] Burrhus Frederic Skinner. Teaching Machines. Science, Vol.128,pp969-977,1958.

[6] Brian Kernighan,Dennis Ritchie. C Programming Language (2nd Edition). Prentice Hall PTR,1988.

[7] 柴田望洋: 新版 明快 C 言語入門編, ソフトバンククリエイティブ, 2004.

[8] 林晴比古: 改訂新 C 言語入門 ビギナー編, ソフトバンククリエイティブ, 1998.

[9] 高橋麻奈: やさしい C 第 3 版, ソフトバンククリエイティブ, 2007.

[10] Stephen Kochan. *Programming in C 3rd ed.* Sams Publishing, 2005.

[11] Greg Perry. *Absolute beginner's guide to C 2nd ed.* Sams Publishing, 1994.

[12] J Bennedsen and E Caspersen. Optimists have more fun, but do they learn better? On the influence of emotional and social factors on learning introductory computer science. Computer Science Education, Vol.18, pp1-16, 2008

[13] A McGettrick, R Boyle, R Ibbett, J Lloyd, G Lovegrove and K Mander. Grand challenges in computing: Education - A summary. The Computer Journal, Vol. 48, pp42-48, 2005

© 2013 Information Processing Society of Japan