Why is programming difficult?

Proposal for learning programming in "small steps" and a prototype tool for detecting "gaps"

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

[1] Sagami Kouyoukan High School
[2] Meisei University
[3] Osaka Gakuin University
[4] Osaka Electro-Communication University

Abstract. In this article, we propose a model for an understanding process that learners can use while studying programming. We focus on the "small step" method, in which students learn only a few concepts for one program to avoid having trouble with learning programming. We also analyze the difference in the description order between several C programming textbooks on the basis of the model. We developed a tool to detect "gaps" (a lot of concepts to be learned in a program) in programming textbooks.

1 Introduction

For students in the digital age it is important to understand that electronic equipment works with a computer and how it works. It is desirable for them to write programs so that they understand how a computer works according to the given instructions in a program. However, a lot of students drop out of introductory programming courses.

Why do students have difficulties with programming, although it is a creative activity and could be fun for them? In our class we often find that students have lost their way when they study particular contents that have many things to learn. That students are lost when they study a particular content with many things to learn. Therefore, we suppose that students are lost when they study a particular content with many things to learn (we say that such programs have "gaps"), so we developed a tool that detects new learning elements between the exercises in programming textbooks.

In this paper we will show the "small step" method which, by reducing the amount of new concepts, helps the students understand programming and the problems in existing textbooks.

We explain the tool and report on the results of applying the tool to C programming textbooks.

2 "Small steps" in programming learning

There is lots of research on the difficulties for novices in learning to program[1, 2]. Novice programmers often face difficulties already in basic courses[3]. There is a lot of novices think that programming is difficult and requires too much knowledge and too many skills to be learned all at once and right from the beginning[4, 5].

Skinner[6] developed teaching machines and programmed instruction that presents material structured stepwise in a logical sequence. His work affects self-study materials for CAI, e-learning systems, and so on.

We focus on the principle that Skinner stated:

> "In acquiring complex behavior the student must pass through a carefully designed sequence of steps, often of considerable length. Each step must be so small that it can always be taken, yet in taking it the student moves somewhat closer to fully competent behavior". [6]

We call this principle "small steps". This principle is efficient in studying mathematics and a foreign language. Our goal is to also apply this principle to programming learning and help novice students to the learning of programming smoothly.

Our hypothesis is that if there are a lot of new syntax elements in one program, students may feel high gaps and find it difficult to understand further programs.

3 Problems in programming textbooks having "gaps"

Teachers or authors of programming textbooks tend to add lots of new concepts (we call them "gaps") in each exercise because they know various concepts of programming. As a result, they make big "gaps" between exercises, and learners have difficulty solving them.

Fig. 1 shows an example of a big gap in a C-course exercise. First the teacher gives a simple example that displays a string constant. Next he/she gives an example that displays a variable number. She or he thinks it is an easy task because it contains only one concept "displays a number that is calculated from a variable".

However, there are lots of new concepts in the second program.

- declaration of variable: `int n`
- data types: `int`
- substitute a constant for a variable: `n = 10`
- arithmetic operation: `n+20`
- formatting: `%d`
- multi argument of a function: `printf("%d",n+20)`

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

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

Fig. 1. Example of a big gap

It is not easy to detect new concepts even if we only show a two-line program. We focus on these gaps that teachers cannot identify. In particular, we focus on repetition (looping) structures with which lots of students have difficulty.

Repetition structures in C are much more complicated. Fig. 2 shows examples of two kinds of repetition: "for" and "while" structures.

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

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

Fig. 2. Example of repetition in C(for, while)

Both programs in Fig. 2 show that the instruction repeats "display a string Hello" 10 times. To understand these programs, learners should know "variables", "data type (int)", "variables initialization", "assignment", "compare operators", "Boolean values" and "increment operator". Many concepts are necessary to understand these simple structures.

General-purpose programming languages like C have multiple expressions to express repetition structures. For example "for", "while" and "do-while". Therefore, teachers tend to give various examples and exercises to students, but these are too much to master, so learners may be confused when, for example, a "for" program contains "if".

Moreover, a nested structure would bring confusion to the students.

Which should be the first structure when teaching repetition structures? Some teachers teach "for" first because a number of repetitions can be indicated clearly with the fixed numbers. However, as shown above, learners must know many concepts in order to understand "for(i=0 ; i<10 ; i++)". Therefore they cannot understand it and rely on rote memorization. On the other hand, teachers wonder why learners forget it so fast.

The "small steps" teaching approach would resolve this problem in the following ways.

- **Changing order:** For example, we should teach "conditions", "increment operator" and "if", before we teach "while". After students understand all of them, we can teach "for".

– **Adding a new program:** For example, if the teacher shows students the program shown in Fig. 3, before he/she shows them the program on the right side of Fig. 1, they only need to learn an arithmetic operation, so the gap is lowered.

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

Fig. 3. Example of adding a new program

4 Analysis of C language textbooks

To verify the hypothesis described above we analyzed programs that appear in C language textbooks.

4.1 Analyzing textbooks for introductory programming

As examples of textbooks for introductory programming we analyzed five textbooks[8–12] focused on the learning order of concepts. Table 1 shows the order of each textbook.

Table 1. Analysis of five textbooks

A	B	C	D	F
variables	variables	variables	variables	variables
...	types	...	assignments	types
assignments	assignments	assignments	types	assignments
expressions	arrays	(arithmetic) expressions	(arithmetic) expressions	arrays
...	strings	casts	casts	(arithmetic) expressions
types	expressions	(comparison) expressions	...	(compound) expressions
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

This result shows two problems with the order of learning control structures.

The first problem is “teaching everything”. Some textbooks show many concepts that do not need to be learned by novice students to explain language specifications. For example, textbook B explains arrays at the introductions of the types and all operators of C language.

The second problem is “disorder”. Some textbooks teach in an order that does not match with the structure of understanding. For example, textbooks B ,C and D introduce “while” statements after “for” statements. To understand “for” statements, the understanding of “if” statements (“condition”) and “while” statements (“repetition”) are needed, but students do not learn repetition at this time.

Textbook A introduces conditional expressions of “if” statements in a weird order; it starts with `x % 5`. This example contains two concepts: “behavior of ‘if’ statements” and “interpretation of integer values as Boolean values (yes or no)”. Table 2 shows the alternative order of conditional expressions. It would be easier if students start learning with a comparison expression. It is easy to grasp an image of boolean values (comparison is answerable by yes or no). And only then students should learn the fact that “all integer values have the meanings yes or no (Boolean) in C language”.

Table 2. Order of comparison expression in textbook A and the alternative order

No.	Textbook A	Alternative order
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

Taking these points into account when the teacher uses a textbook to teach programming, they can help students understand smoothly by changing the order of programs or adding supplemental programs that are not in the textbook.

The order of programs in a textbook reflects the philosophy of the author and is the standard order when using the textbook. We are interested in whether the standard order and the “ideal” order are the same. If students and teachers learn in the ideal order, they must feel that it is “easy to learn” and “easy to teach”.

5 Tools for detecting “gaps” in programming textbooks

As we described before, most textbooks have gaps. Students and teachers feel that gaps are difficult. We propose to change the order of programs in textbooks or provide other programs to fill these gaps. However, there are some problems.

– It takes a long time to find the gaps in textbooks.
– If a teacher tries to fill the gap, he/she is not sure whether the gap is really lowered.

Our hypothesis is that if there are a lot of new syntax elements in one program, students may feel high gaps and may find it difficult to understand further programs.

To solve these problems, we developed a tool named "De-gapper", which detects new syntax elements of each program in programming textbooks.

5.1 Overview of the tool

De-gapper requires a set of program files as an input, arranged according to occurrence in the textbook. For each program file, De-gapper checks and reports if there are learing concepts appearing for the first time in the textbook. Fig. 4 shows an example of input program files and output report of De-gapper.

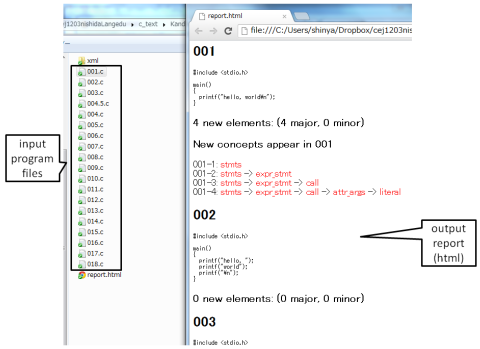

Fig. 4. An example of input and output of De-gapper

If there are too many new learning concepts for one program, the program is expected to be too difficult to be understood by students.

This tool is useful for a teacher teaching programming with a textbook. De-gapper tells him/her in advance which parts of the programs in the textbook are difficult, and he/she can prepare supplemental explanations for the difficult parts.

It is also useful for the author of the programming textbook. De-gapper lists all learning concepts that appear in the textbook. The author can check whether all learning concepts are explained in the textbook.

5.2 Implementation

De-gapper consists of a "parser" and a "checker". The parser parses all program files in the textbook and outputs syntax trees as XML files (called "XML tree

files"). Fig. 5 shows a sample input and output of the parser. The snipped parts of Fig. 5 contain tags for the entire program and function definitions. Since this paper will discuss programs that have just one function, we will omit these tags.

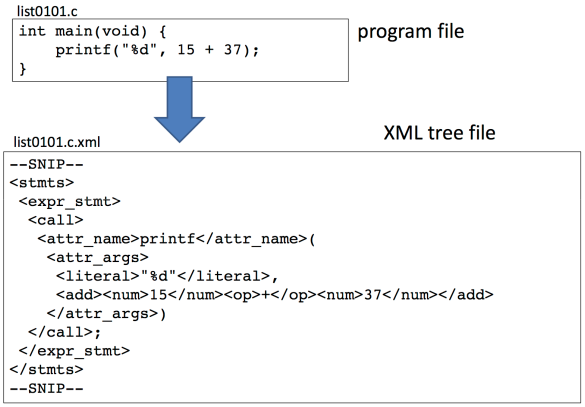

Fig. 5. Example of input and output of the parser

The checker scans all XML tree files in the order of occurrence in the textbook. For each XML tree file, the checker detects the paths of tag hierarchy. For example, the checker will detect the following paths in the XML tree file in Fig. 5:

```
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
```

De-gapper names these concepts "`list0101-1`" to "`list0101-7`". Note that they are ordered by path string, not by occurence in the program.

If there are paths that have not appeared yet in previously checked XML tree files, the paths are detected as "new learning concepts" of the program corresponding to the XML tree file.

Current implementation can parse C and Java program files. But it is easy to make it workable for other languages if we add parsers for these languages.

5.3 Examples

Example 1 - Small Example: Suppose that there is a textbook whose first program is `list0101.c`, shown in Fig. 5. De-gapper detects all paths of the program as new learning concepts. These can be explained as:

– `list0101-1`: The first example of a list of statements.
– `list0101-2`: The first example of an expression statement.
– `list0101-3`: The first example of a function call in a statement.
– `list0101-4`: The first example of an additive expression in a function call.
– `list0101-5`: The first example of a number in an additive expression.
– `list0101-6`: The first example of a `+` operator in an additive expression.
– `list0101-7`: The first example of a literal in a function call.

If the next program in the textbook is the one shown in Fig. 6, De-gapper detects 11 new learning concepts named `list0102-1` to `list0102-11`. These can be explained as:

– `list0102-1` to `list0103-3`: The first example of a variable declaration.
– `list0102-4`: The first example of a variable in a function call.
– `list0102-5`: The first example of an assignment.
– `list0102-6`: The first example of a variable in the left member of an assignment.
– `list0102-7`: The first example of an additive expression in the right member of an assignment. (at `vy=vx+10;`)
– `*list0102-8`: The first example of a number in an additive expression (at `vy=vx+10;`).
– `*list0102-9`: The first example of a `+` operator in an additive expression (at `vy=vx+10;`).
– `list0102-10`: The first example of a variable in an additive expression (at `vy=vx+10;`).
– `list0102-11`: The first example of an immediate number in the right member of an assignment(at `vx=57;`).

In these concepts, `list0102-8` and `list0102-9`, annotated with `*`, are not completely "new" learning concepts for the following reasons:

– Students learn `list0102-7`, meaning that they can write additive expressions on the left member of an assignment.
– Additive expressions are already learned in `list0101-4`.
– They also learned `list0101-5` and `list0101-6`. They already know that they can write `+` operators or numbers in additive expressions.
– Thus, they would imagine easily that they can write `+` operators or numbers in an additive expression on the left member of an assignment.

We will call such new concepts "minor" new concepts, and the rest "major" new concepts.

Example 2 - The K & R book: We will show another example. We applied De-gapper to "The C programming language"[7], from 1.1 to 1.6. The result is shown in Table 3.

De-gapper found that program No. 3 had extremely high gap with 26 major new concepts. Many supplemental programs are needed. However, we do not

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
```

Fig. 6. Example of the program that contains variables first

discuss this program in this section (This gap is apparent even if we do not use De-gapper). We will focus on the programs that follow No. 3. From No. 4 to No. 11, most programs have quite low gaps, with 1 to 3 new concepts. Number 5 has a little bit of a high gap with 6 new concepts. Fig. 7 shows program No. 5 (also No. 4 to check gaps).

This is the first example of a "for" statement. Number 5 has the following major new concepts:

```
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
```

When students learn the "for" statement, the following concepts are mandatory:

- `005-1`: the "for" statement itself
- `005-6`: the initializer part of statement
- `005-2`: the condition part of statement
- `005-23`: the increment part of statement

We focus on the remaining parts: `005-3` and `005-12`. These are interpreted as:

- `005-3`: The first example of an immediate number in a comparison expression: `fahr <= 300` .

Table 3. New concepts in the K & R book from 1.1 thru 1.6

No.	Section	# of new concepts (major/minor)	The first example of:
1	1.1	4/0	`printf("Hello, world\n");`
2	1.1	0/0	multiple `printf`
3	1.2	26/4	`int` variables and "while" statement
4	1.2	1/2	`float` variables
5	1.3	6/22	"for" statement
6	1.4	1/0	`#define`
7	1.5.1	2/0	`c=getchar();` (receiving return value of function call)
8	1.5.1	3/4	`c=getchar()` in loop condition of "while" statement
9	1.5.2	3/1	increment prefix `++`
10	1.5.2	3/1	"for" statement with an empty statement
11	1.5.3	3/5	"if" statement and character literal
12	1.5.4	10/24	else part and `||` operator
13	1.6	15/17	arrays and `&&` operator

– `005-12`: The first example of an arithmetic expression in an argument of a function call: `printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));`

These concepts are not the essential parts of a "for" statement. Although the authors may intend to write program No. 5 as a transformation of No. 4 ("while" statement), it is not a pure transformation; No. 4 contains the variables `lower`, `upper`, `step` and `celsius` to define ranges or keep a calculation result, while No. 5 does not contain them. The pure transformation would be like Fig. 8 named "No. 4.5". If there was this program between No. 4 and No. 5, the number of the new concept would be 5:

```
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` is a new concept that does not appear in No. 5, which contains no compound statement in the "for" statement. Aside from `004.5-13`, the remaining four new concepts correspond to the mandatory concepts of "for" statements.

This example shows that the program in which De-gapper indicates many learning concepts does actually have difficult points.

Through these examples, De-gapper is expected to have the following benefits:

– De-gapper can find gaps from many programs automatically.
– De-gapper helps teachers find the gaps hidden in textbooks and gives them chances to "degap" (to prepare supplemental programs between gaps).
– After inserting supplemental programs, De-gapper can re-evaluate the gaps to check if the gaps are lowered.

No. 4

```
#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. 5

```
#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));
}
```

Fig. 7. First example of "for" statement in K & R book, in comparison with the previous "while" statement program.

6 Summary and future work

We proposed a "small step" teaching method for programming learning and a tool for detecting "gaps". We plan to develop the method and tool in the following ways:

- **Weighting new concepts:**
  The number of new concepts in a program is not always proportional to the "height" of gaps, i.e., how difficult the program appears to the students? We will have some experimental classes to measure gaps that De-gapper has calculated and the actual gaps that students felt.
- **Correspondence between new concepts and program parts:**
  In current implementation, the contents of new concepts are indicated by XML paths. They should be interpreted into understandable natural language. In addition, current implementation does not show which part of the program corresponds to the paths.
  We plan to add a feature that automatically highlights which part of a program are new concepts or shows corresponds parts by hovering the mouse cursor over XML paths.

Acknowledgment

This work was supported by JSPS KAKENHI Grant Number 22500828.

```
#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);
  }
}
```

Fig. 8. A proposal to prepare a supplemental program as the first example of a "for" statement in K & R book.

References

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. Paul Gross, Kris Powers. Evaluating assessments of novice programming environments. ACM, ICER'05, pp.99–110, 2005.
3. Essi Lahtinen, Kirsti Ala-Mutka, Hannu-Matti Jarvinen. A study of the difficulties of novice programmers. ACM, SIGCSE Bulletin, Vol.37, Issue 3, pp.14–18, 2005.
4. Shuhaida Mohamed Shuhidan, Margaret Hamilton, Daryl D'Souza. Understanding novice programmer difficulties via guided learning. ACM, ITiCSE '11, pp.213–217, 2011.
5. James Spohrer, Eliot Soloway. Novice mistakes: Are the folk wisdoms correct? ACM, CACM, Vol.29, No.7, pp.624–632, 1986.
6. Burrhus Frederic Skinner. Teaching Machines. Science, Vol.128,pp969-977,1958.
7. Brian Kernighan,Dennis Ritchie. C Programming Language (2nd Edition). Prentice Hall PTR,1988.
8. BohYoh Shibata. *Meikai C Gengo 2nd ed, Nyumonhen*. SoftBank Creative, 2004.(in Japanese)
9. Haruhiko Hayashi. *Shin C Gengo Nyumon 2nd ed, Beginner Hen*. SoftBank Creative, 2003.(in Japanese)
10. Mana Takahashi. *Yasashii C 2nd ed*. SoftBank Publishing, 2003.(in Japanese)
11. Stephen Kochan. *Programming in C 3rd ed*. Sams Publishing, 2005.
12. Greg Perry. *Absolute beginner's guide to C 2nd ed*. Sams Publishing, 1994.