Freshmen Courses for Informatics Education Incorporating Blended Learning

Yasushi Kuno
Univ. of Electro-Communications
y-kuno@uec.ac.jp

Hironori Egi
Univ. of Electro-Communications
hiro.egi@uec.ac.jp

Noriko Akazawa
Univ. of Electro-Communications
akazawai@uec.ac.jp

Sumito Takeuchi
Univ. of Electro-Communications
s-take@uec.ac.jp

Michiko Sasakura
Univ. of Electro-Communications
sasakura@uec.ac.jp

Makiko Kimoto
Univ. of Electro-Communications
kimoto.makiko@uec.ac.jp

KEYWORDS

blended learning, flipped classroom, report-based evaluation, programming exam with automatic scoring

ABSTRACT

University of Electro-Communications (UEC) is a national college located in the suburbs of Tokyo, and is specialized in informatics, science, and engineering. The task of our informatics course for the freshmen year is to smoothly prepare students toward advanced technical curriculum in the sophomore year and further. To accomplish the task with limited class hours, we have designed and implemented a systematic blended learning curriculum and methods using Moodle LMS. In our method, students study class contents using text and videos in advance (flipped classroom), and the majority of class hours are used for exercises (mostly using computers). All learning materials are distributed using Moodle, and students can use them from home as well as in our computer rooms. Students are required to submit a report each week, and those reports are scored by the lecturer and teaching assistants with appropriate feedback. End-term exams are also carried out using Moodle with automatic scoring. Final marks are calculated by 50:50 sum of report score and exam score. As a result, students' satisfaction was high. Further, with our programming course, most students could acquire fundamental programming skills.

1 INTRODUCTION

University of Electro-Communications (UEC) is a college located in the suburbs of Tokyo, and is specialized in informatics, science, and engineering. In the undergraduate curriculum, we provide three clusters (departments), cluster 1 through 3. Cluster 1 covers informatics, and cluster 3 corresponds to science and engineering. Cluster 2 is the "boundary" or "fusion" cluster, including robotics, sensor & control, and so on. As a result, our students need to be fluent with informatics.

In Japanese colleges and universities, science and engineering (including informatics) 4th (final) grade undergraduate students must experience research activities and write a paper to graduate. Therefore, they have to acquire advanced knowledge and skills needed for research by their 3rd grade, which is not an easy task.

Service Management Congress on DT&EL, Suderburg, Germany, May 27-29, 2019.
.

We are designing and implementing the informatics curriculum for the freshmen year, whose task is a smooth entrance of our students into more advanced materials covered in the sophomore year and further. Those materials include algorithms & data structures, numerical analysis, computer architecture, operating systems and so on.

However, students also have to take mathematics, physics & chemistry (with experiments), and some other courses in their freshmen year. Therefore, we have only two informatics course of 15 weeks (90 minutes per week) each, namely "computer literacy" in the first semester and "fundamental programming" in the second semester (the former and the latter half of freshmen year).

The above is rather limited hours considering what students have to acquire by the sophomore year. To overcome the problem, we have designed and implemented the above two courses with extensive use of IT (information technology) in blended learning settings, which we explain in this paper.

Rest of this paper is as follows. In Chapter 2, we discuss the general design policy of our courses (our two courses have many design choices in common). In Chapter 3 and 4, we explain our design and experiences on "computer literacy" ("CL" below) and "fundamental programming" ("FP" below), respectively. In Chapter 5, we review related works. In Chapter 6, we present discussions and conclusion.

2 GENERAL DESIGN POLICY FOR TWO COURSES

2.1 General course settings

UEC have about 800 students in a grade. However, the number of students registered to CL or FP courses differ from above, because some students pass the subject due to qualified courses in their previous schools (other colleges), and some students failed in the previous year take the same course again. The number of students showing up to the end-term exam is approximately 770.

There are 13 classes of approximately 60 students each, with one lecturer and two graduate teaching assistants for each class. Two computer rooms are used, so there can be at most two classes at the same time. There are 15 weeks of class hours (90 minutes each).

80 minutes end-term exam is also performed in the ordinary class hour in 16th or 17th week. We fully use CBT (computer based test) to reduce scoring cost. To prevent problem leakage, We provide seven problem sets of similar but not identical problems; classes

performing the exam on the same or neighboring hour share the same set of problems.

2.2 Problems and solutions

Firstly, we must make clear what problems to attack with our course design; the following is our list.

(1) Limited time — As noted above, 15 weeks with 90 minutes each is allocated for each course, and we have to get along with this.
(2) Need for exercises — Students will have very few experiences on college level informatics materials. For concrete understandings of those materials, experience through exercises is mandatory.
(3) The large difference among students — Some of our students are interested in informatics since their youth, and are fluent with it, while others are just novices. Additionally, although "information study" subject is mandatory in Japanese high school curriculum, actual contents and levels largely differ from school to school, leading to again large differences in even among "novices."
(4) Novices dropping out — The gap between required skill to pass and students' actual skills tend to be large in novices, so novices have a tendency to dropping out. However, most of our students are novices, because "information study" is not considered important in Japan (it is rarely used in a college entrance exam, and subject that is not used in college entrance exam often regarded as unimportant).
(5) Skilled student getting bored — For our college, development of skilled students throughout the freshmen year is very important. However, if we make contents of the course easy to prevent (4), then skilled students will get bored and lose interest in the subject.
(6) Difficulty in programming education — Programming is notorious for its difficulty in teaching; we have seen many students dropping out. FP curriculum has to deal with this difficulty somehow.

For (1) and (2), flipped classroom (student study supplied materials in advance to the class hours) can be the solution. For the purpose, we prepared textbooks which explain all materials in detail, and lecture videos for every week.

In order that those materials actually be used, we additionally did the followings.

- We have included all materials in a Moodle course, and used them fully in the class hours. Students can log in to the course from their home, and they will see the same materials, leading to a smooth transition from classroom learning to home learning.
- Students have to submit their assignment report every week through the Moodle course. Additionally, they are also required to submit "activity report" at the end of each class hour. Form of these reports are similar, so that they can "practice" report submission in every class hours, and also get used to Moodle operation.
- Video lectures are actually placed on YouTube, so that they can comfortably be viewed with smart phones. They are split into a short movie of around 5 minutes, so that viewing one of the video does not take long.
- We asked the lecturer to avoid lectures during class hours and proceed to exercise as soon as possible; this urge students the need for studying in advance.

To overcome (3), we have prepared many exercise problems for each week, and ordered students to "choose one or more exercises, practice them, and write a report." There is also a large difference in the difficulty of those exercises. Therefore, students fluent in the topics choose difficult exercises and answer many in their report, while novices will choose easy ones and answer only a few in their report.

The above method seems not fair at first glance, but we believe this scheme is a "must" in our situation. Consider the following: muscular workout requires appropriate weight to be effective; if too heavy, it will cause damage, and if too light, it will be useless.

Likewise, the difficulty of an exercise should be balanced with the students' ability; if too difficult or too easy, it will be useless. We saw many classrooms in which only one exercise is presented and all the students must do that; we suspect that that exercise will be too difficult or too easy for most of the students and thus ineffective. We avoid such problem.

People also might suspect: "hey, all student will choose the easiest exercise for their luxuriation." In our observation, most student choose exercises appropriate for their ability. Perhaps it is because skilled students like informatics and so want to attack difficult exercises, and for novices easy exercise will be appropriate anyway. There do exist lazy students who choose easy exercises in spite of their ability, and they tend to score less in the end-term exam.

Solutions to (4) and (5) are tightly connected to the above scheme (students choose which exercise to solve). As noted previously, students are required to submit a report on every week. Contents of these reports are how they solved the problems, plus findings from the experience of solving the problems; the difficulty of the problems are not accounted for in scoring.

Professors are accustomed to review and grade reports with respect to their quality (sentences, logics, presentations and so on), and we are asking for such grading. Therefore, novices and skilled students in informatics are evaluated on equal foot. Students also know the fact, so novices are not discouraged.

Additionally, our criteria for grading is that, ordinary reports, which is the majority, receives score B, extremely superior (very few) reports receive A, and reports with some clear failure (do not satisfy some stated condition) receives C. This criteria is very coarse and thus reports can be graded rather quickly.

Yet, a class with 60 students and 15 weeks result in 900 reports in total, which is a huge amount. To reduce the burden of opening 60 documents every week, we concatenated 60 reports into single PDF document and added watermark with a student ID to every page, changing colors among students. Professors just have to skim through a PDF document of approximately 120 (week of the short report) to 600 (week of the long report) pages, which was manageable enough.

Many students complain that their reports receiving B in spite of their effort, but we repeatedly explain that A is limited to really

excellent reports. Instead we are asking professors to return one-line feedback comments to their reports if possible (along with TA); actual existence or elaboration of feedback comments vary among classes.

We assign 3 points to B report, 4 to A, and 2 to C. There are two "integrated exercise" weeks, in which not many topics are presented and students attack slightly complex problems (or group works), and those receives double points (b = 6, a = 8, c = 4). The result is that all normal grade (B or b) results in approximately full score (50 points) for the reports. With A or a, additional points can be earned, up to 59 (one need 60 points to pass the course). Remaining scores come form end-term exam, whose full score is also 50. The total of these scores (maximum 100) is used for grading.

With this scoring, one is likely to pass the course if they submit a normal report every week, which is encouraging to novices — solution to (4). Note that there are also good points for skilled students, because good total score requires good points from the end-term exam, so they will not complain about either — solution to (5).

As for (6), we are going to discuss the topics in Chapter 4, but we are using some of CL hours as preparation to FP. For example, our students experience assembly language simulator in "principles of computers" topic, and simple JavaScript programming in "software development" topic included in CL.

3 COMPUTER LITERACY (CL)

3.1 CL course design policy

As we search on the net, there seem to be many colleges having a course named "Computer Literacy." However, the content of our CL is distinguishing in that it mainly focuses on Unix, LaTeX and HTML (Figure 1), which will be needed in the sophomore year and later.

The difficulty in our CL course design is that large portion of our contents are command line based (Unix, Shell) and markup language based (LaTeX, HTML / CSS), while students are accustomed to GUIs (Windows, Explorer) and GUI applications (Word, Excel, PowerPoint). If they feel like: "Hey, what heck is this old-fashioned, complex, tedious and seemingly useless tools?" then we will certainly fail.

Solutions of our predecessor (course design used until the school year 2016) were to "treat them as knowledge to be acquired by students." Students learn that knowledge through lectures, memorize them, dump them onto an exam sheet, and forget — it is seemingly understandable to students, because many school contents were likewise. However with such experiences, students will never use that those knowledge to actual tasks in their college life, which is undesirable.

Our solution is already described in the previous chapter — lots of experiences through exercises in the class hours, and actual use of that knowledge through weekly report assignment (students have to use command lines or markup languages because exercise problem states as such).

3.2 Curriculum for CL2018

Figure 1 shows our CL curriculum for the school year 2018. #1 is the guidance, but a difficulty in authentification and issue of safe passwords are noted, and students craft their own "safe and also memorizable" password and change login password. Most students have not heard about touch typing, so we introduce the touch method here. We provide measurement page, and those reached to 150 or 100 characters / minutes by the end of the semester will receive 3 or 2 bonus points respectively.

Table 1: Weekly curriculum of CL2018

week	topics
#1	What is computers?, Passwords, Touch Typing
#2	Principles and Functionality of Internet
#3	Nexworks and Security
#4	Principle of Computers (assembly programming)
#5	File System, File Manipulation
#6	Text files and Text editors
#7	Computer Systems and OS
#8	Unix Filters, Shell scripts
#9	Markup and text formatters (LaTeX)
#10	Graphics, figures and tables (LaTeX)
#11	Academic Literacy (Integrated Exercise)
#12	HTML / CSS and Web page creation
#13	Web and Information Architecture
#14	Web site design and construction in team (Integrated Exercise)
#15	Software development and Tests (JS programming)

#2 includes the first experience of Unix command line, but major topics is on packet switching, network protocols and Internet; students try `ping` command to measure packet round-trip time, and paper-based exercise on error recovery protocol (the sender write messages onto multiple memo sheet "packet" and send one by one to the receiver, and intervening "network" occasionally introduce transmission errors — the sender and the receiver have to come up of some error recovery method to attain error free transmission).

#3 includes topics of cryptography and PKI, and students examine PKI certificate on their Web browser (although they must have used Web browsers many thousands of times, scarcely any student have not experienced those matter in the browser). Then we set up a college e-mail account on our IMAP mail client (Thunderbird), and view RFC822 format message for their own mail messages (which is also their first experience).

#4 contains the topic "principle of computers." However, those topics are difficult to be understood with ordinary lectures. Therefore, we use an assembly language programming experience here. We designed and developed a small accumulator (1-register) machine architecture and its simulator running on a Web page. Students experience simple assembly language programming with the simulator. Figure 1 shows a screenshot of the simulator, with the result of "compare X and Y value, then store larger to Z" program shown. Our intent was that students will understand the principles of the computer through the exercise, and the exercise can be a good introduction to programming by itself.

#5-#8 are the various aspects and topics on Unix system, and students exercise Unix commands and tools.

#9-#11 are focused on LaTeX report writing. Starting from #9, students are required to write their reports with LaTeX and submit

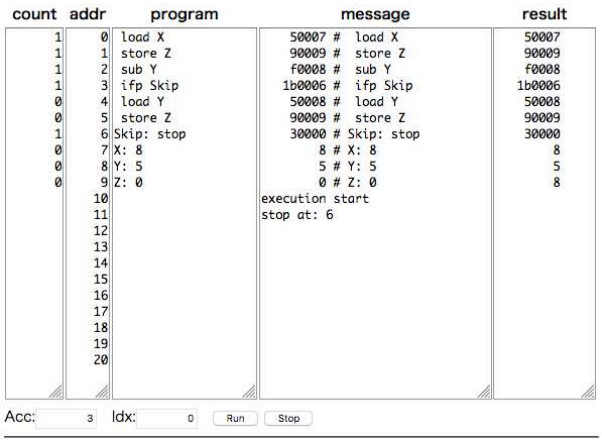

Figure 1: A small computer simulator

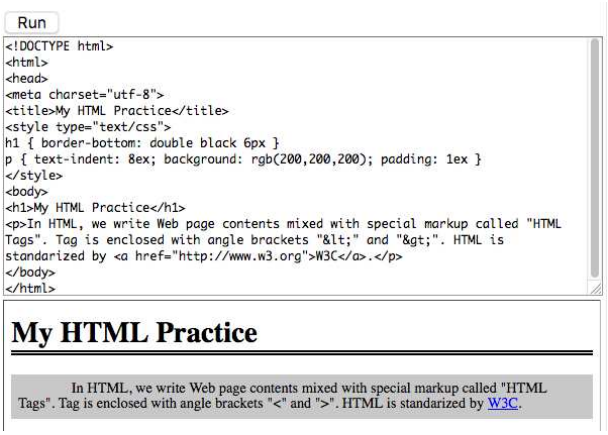

Figure 2: HTML practice page

resulting PDF. #9 introduces LaTeX, and in #10, basics of graphics (pixel graphics, vector graphics) and various image files are covered, along with LaTeX figures and tables. #11 is an integrated exercise; topics of academic literacy are briefly introduced, then the students are gathered to a group of 3-5 people to discuss an academic literacy-based theme they have chosen.

#12-#14 are focused on Web page design and construction with HTML and CSS. #12 introduces HTML / CSS using HTML practice page (Figure 2). The page allows students to type and modify HTML / CSS in the upper area, and "Run" button immediately shows the result display in the lower area; CSS styling, character entry (for special characters) and link tag can be used as in the ordinary HTML file. Note that we teach both HTML and CSS from the start, because students have high motivation in decorating their pages; they were interested in a various presentation which can be specified through CSS. #13 covers topics of inline / background images in a page, intra-site links, and structure of Web sites (linear, hierarchy and so on). #14 is again an integrated practice; students forms groups and design / develop a site on some specific theme (chosen by themselves) containing multiple pages.

Finally, #15 is an independent topic of software development; this topic was placed in the middle in the school year 2017 with Web site construction being the last topic, but students complained about heavy integrated exercise just before the exam week, so we moved this topic to the last week. This topic includes a notion of high-level language (compared to assembly language, which is low level), simple JavaScript programming, and concept of software product development (which is quite different from simple programming) along with the notion of software testing. In this week, we use two practice web pages, namely JavaScript execution page and test case execution page. In the former, one can type and execute a simple JavaScript program. In the latter, one enters a JavaScript function plus several simple test cases (set of input parameters and expected return value) and run them to experience how the unit test looks like.

3.3 Experiences of CL2018

Here we report our experience for the school year 2018. Week #1-#3 were relatively easy, and students seem to enjoy them. As for assembly programming in #4, many novice students report that they have enjoyed their programming experiences. As assembly language instructions (corresponding to CPU instructions of "small computer") are very primitive and simple, students had little difficulty in grasping and using them; some students with previous programming experiences were a bit confused because assembly language does not have nested if or while statements.

However, Unix topics in #5-#8 were dark sides; although students could submit the report as in the other weeks, chosen exercises were relatively easy ones and an only small number of student (perhaps with previous Unix experiences) became fluent with file / directory hierarchy, Unix commands, process handling, and Emacs editing. We feel these topics are difficult to master in several weeks.

From #9, LaTeX sections start. Perhaps because we have explained that LaTeX is the majority in science / engineering academic community and convenience LaTeX math formulas, there was a little complaint in learning LaTeX. #10 mainly focuses on exercise on graphics (write PPM image and / or PostScript with a text editor); students were impressed that text can actually be sources for graphics. The focus of #11 is group discussion; many students noted that they have not experienced such discussion before, and it was interesting (apparently this is the weak side of traditional school education in Japan). These reports must be formatted with LaTeX so that students get accustomed to it; by the end of the semester, several students seem to start using LaTeX for reports in other courses.

From #12, HTML / CSS sections start. As students have already learned one markup language (namely LaTeX), learning another seems of little difficulty. Additionally, the practice page is proved to be a powerful tool for learning the topic. In #13, students are requested to construct multiple passes or pages with separate image files referred to within, so practice pages cannot be used. However, as students see that copying HTML source from the practice page input area to text editor results in the correct HTML file, they had

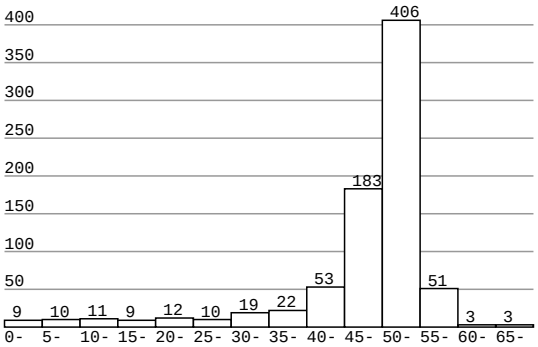

Figure 3: Histogram for CL report score (n=801)

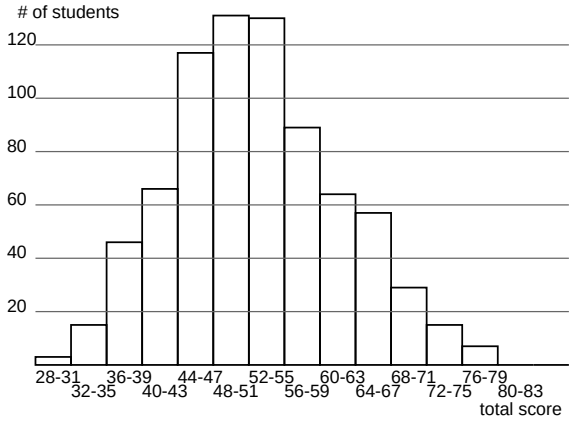

Figure 4: Histogram for CL exam score (n=769)

little problem in transition. In #14, Web site planning, design and development in teams are the mandatory exercise, and most students enjoyed their original Web site construction. They also noted about the difficulty of team development (load bias, communication problems and so on), which we hoped them to experience.

Finally, #15 is mainly focused toward programming in a high-level language (JavaScript in this case), although other topics are also covered. We asked which one of the assembly and JavaScript to prefer in the inquiry, and as a result, some prefer assembly and the other prefer JavaScript. This topic seems to be a good preparation for FP in the second semester.

3.4 Evaluation method for CL2018

As noted previously, 50 points come from assignment reports, and the remaining 50 points come from the end-term exam of 80 minutes. For the exam, we used the following types of problems.

- 5-2 problems (34 problems) — Five sentences regarding a chosen topic are provided, and students choose two correct sentences from the list. The score will be 2 if both choices are correct, 1 if one choice is correct, and 0 otherwise.
- Split-Paper (SP) tests (8 problems) — Students are asked to construct correct answers by choosing lines from choice set and reordering them as necessary. The score will be 2 if the resulting answer is correct, 1 if a single difference from the correct answer is found, and 0 otherwise.

We use 5-2 test instead of a simple multiple choice test or YES-NO test, because interference among five statements shows a problem more difficult. As to the SP tests, we explain them in depth in the next chapter; for CL course, various kind of lines — lines from LaTeX source, HTML source, PPM image text, and program text are used.

As exams are carried out in a normal class setting (in two computer rooms), we prepared seven problem sets to avoid problem leakage. The corresponding problems for each set are similar.

For a total of 42 problems, the full score will be 84 points; we scale them appropriately, and calculate sum with report score (plus typing bonus of at most 3 points) for the final mark.

Table 2: [CL] Q. Was the course useful to you?

Positive	Weak Positive	Neutral	Weak Negative	Negative	No Answer
164	230	191	72	39	9
23.0%	33.5%	26.8%	10.1%	5.5%	1.3%

Table 3: [CL] Q. Have you learned much in the course?

Positive	Weak Positive	Neutral	Weak Negative	Negative	No Answer
405	243	33	18	5	10
56.7%	34.0%	4.6%	2.5%	0.7%	1.4%

3.5 Results of CL2018

Figure 3 shows accumulated points of report scores (as noted previously scores over 59 is forced to 59 for grading). As expected, frequency around 50 points (all B / b) is highest, meaning that most students submitted the ordinary-level report.

Figure 4 shows a histogram for the exam, with the full score being 84. From the distribution, we consider that the test score appropriately measures students' performance. We also examine students' score for each problem to investigate where to improve in the course (not shown here due to space limitation).

Table 2 and 3 are a summary of inquiry in the last class hour (#15). These results indicate that students recognize the class as useful, and they have learned much from the course.

4 FUNDAMENTAL PROGRAMMING (FP)

4.1 FP course design policy

In contrast to CL, the goal of FP is clear and can be agreed upon by most people — just acquire programming skills. However, this simple goal is notoriously difficult, as noted before.

Based on our previous experiences, we choose the following seven policies to overcome the problem (difficulty of programming education). We explain them below.

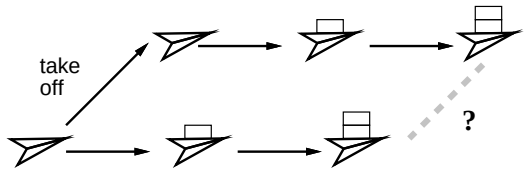

Figure 5: Concept of take off

Policy 1: Take off first. The term "take off" means that one can express his / her idea as programming code and run it. It sounds like a final goal and not a policy, but it is not so.

In many programming classes, the teacher first explains not only overviews but also many details (accurate lexical or syntax rules and so on) of the programming language before exercises. It is perhaps because many programming textbooks are written in such a way. With such method, students get to exercise of writing their own program, they get stuck because so much knowledge was already presented, and they cannot know which of this knowledge are important and which are not (the lower route of Figure 5).

In our scheme, we present a short but complete running program and explain them. Following is the first program in our FP course, a Ruby method to compute the area of a triangle.

```
def triarea(w, h)
  s = (w * h) / 2.0
  return s
end
```

Then we explain the meaning of the code from top to the bottom, but it won't be long because the program is so short (we avoid generalization as much as possible here).

After students actually entered and run the above code, first exercise problems are presented. They are: "sum of two numbers," "volume of corn," "the inverse of a number" and so on. It is not difficult for students to solve those problems, and this means "express his / her idea as programming code and run it." Then, we add new knowledge one by one, always making sure the "took off" status of the students continuing (upper route of Figure 5).

"Take off first" approach has huge benefits, namely: (1) if something goes wrong, computer automatically tell the student as such because the program will not run, so students can be confident with their code, and (2) students are highly motivated because looking at one's own program running is a pleasure.

Policy 2: Varied levels of exercises. This is the solution to the problem (3) of section 2.2.

Policy 3: Take precedence on practice. To keep the status of "took off," practice is important. To reserve enough class hours for practice, the flipped classroom is employed, as in CL.

Policy 4: Encourage novices. This is the solution to the problem (4) of section 2.2.

Policy 5: Clearly indicate the goal of acquiring programming skills. Many programming course exams include problem other than writing programs, e.g., test syntax knowledge, program comprehension, and so on. Therefore, many students without actual programming skills pass the programming course thanks to these kinds of problems. However, they will get stuck when they really have to create a program to solve their problems. It is as if a doctor saying: "I got the doctor license thanks to easy problem, but I really do not have the skill to treat patients."

Therefore, in the end-term exam of our FP course, all of the problems are program construction, and we announce our students as such from the start, and students struggle to acquire the skill. Previously, program construction exams were difficult for a large number of students because a skilled person has to grade them. However, we developed Split-Paper (SP) tests for programming performance evaluation, and these tests can be automatically scored. We discuss the issue of evaluation further in section 4.4.

Policy 6: There is no single "correct" code in programming. In our experience, high school students have the tendency of believing that there is always "a single, correct answer" to the problem, and they "search" for that single solution. However in programming, code with the identical outcome can be written in multiple ways, and students should realize the fact.

We repeatedly tell our students the above issue, and presents multiple "correct" sample answers to a problem many times. Additionally, we speak to the student as: "There is multiple correct ways to write the code, so YOU have to decide among those choices yourself; develop your own policy on which way to choose."

Policy 7: You should produce your code out of your brain. As in Policy 1 (take off policy), students should express their ideas as program code. However, for the first few weeks, those "ideas" are provided as exercise problems, and they code the solution. However, people learn programming best when they are programming their own idea, what they want to do themselves.

Therefore, we include "free programming" problem stating "create a program which you feel interested" on many occasions. Additionally, we use two "Integrated Exercise" week for the creation of pictures (images) and movies (frame animations) of students' preference. They are appropriate because we can easily come up of our own idea with pictures and movies.

4.2 Curriculum for FP2018

Figure 4 shows our FP curriculum for the school year 2018. We use Ruby for 10 weeks, and use C language for the remaining 5 weeks. Ruby was chosen because it is a suitable language for "Take off first" scheme; code can simply contain target method (function or subroutine in Ruby term), no extra declaration, includes or surrounding module necessary.

However in UEC, programming curriculum of the sophomore year is based on C language, so we cannot avoid C language for our course. In the programming community, it is often said that learning a 2nd language is not so much of a burden but leads to lots of benefits. In our case, C language (in addition to Ruby) can provide (1) experiences in a statically typed language, and (2) knowledge that primitive elements of programming languages (if, while, array, struct) are similar or identical.

Table 4 shows our FP curriculum for the school year 2018. #1 includes a brief guidance which is immediately followed by the explanation of the first example (`triarea` above), how to use Ruby, and then first exercises explained above. For all exercises we use

Table 4: Weekly curriculum of FP2018

week	topics R: Ruby; C: C language
#1R	Introduction to Programming; Numerical Errors
#2R	Control Structures; Numerical Integration
#3R	Control Structures(2); Arrays and its usage
#4R	Procedure and Abstraction; Recursion
#5R	2-Dim Arrays; Records; Image Representation
#6R	Drawing a Picture (Integrated Exercise)
#7R	Sorting Algorithms; Computational Complexity
#8R	Complexity (2); Random Numbers/Algorithms
#9R	Object-Orientation
#10R	Dynamic Data Structures; Encapsulation
#11C	Introduction to C Language; Solving $f(x) = 0$
#12C	Various C Types; Dynamic Programming
#13C	Manipulating Strings; 2-Dim Arrays in C
#14C	Record Types in C; Dynamic Data Structures (2)
#15C	Team Development (Integrated Exercise)

the `irb` command (read-eval-print loop in term of programming language community), so that we can invoke methods by directly specifying input data and automatic printouts of return value; in this way we can postpone tedious, lengthy and difficult input / output altogether.

The latter half of #1 includes topics of numerical error present in floating point calculation (real numbers), and exercise observe various kind of numerical errors appearing in computation. Such topics are not usually included in the introductory programming course. However, the topic can be used with simple straight line code (no loops or branches), so we can include it here. The intention is that skilled students are going to investigate these advanced topics with interests; they will get bored if such topics do not exist. Additionally, we wanted all students to know an important fact that "computation with computer is not at all 100%-accurate," to which many students expressed surprise.

#2 and #3 focus on if statements and loops, and then combines them to construct more complex control structures. Introduction to arrays is also included. #4 focuses on procedures. As noted above, we use Ruby procedure (method) from the beginning, so here we review the concept of the procedure (parameters, return values, variable scopes), and also provide materials on global variables and recursive procedure.

As stated in the "Policy 7" section, we would like to include exercises on drawing pictures (with code), so #5 and #6 is focused to this topic. In #5, we introduce concept of 2-dim arrays, records, and combine them to represent an image. We construct an arbitrary image on the image data structure in memory, and can use simple Ruby method to write out as a PPM format image (PPM is a simple image format used on Unix). Followings are the code to create image data structure and `writeimage` method.

```
Pixel = Struct.new(:r, :g, :b)
$img = Array.new(200) do Array.new(300) do
         Struct.new(255, 255, 255) end end
def writeimage(filename)
  open(finename, 'wb') do |f|
    f.puts("P6\n300 200\n255")
    $img.each do |a| a.each do |p|
      f.write(p.to_a.pack('ccc')) end end
  end
end
```

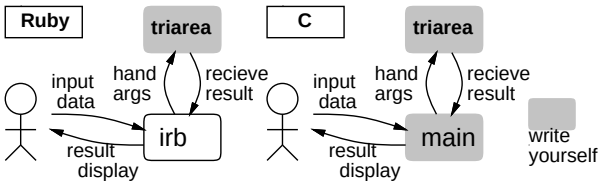

Figure 6: Roles of irb command and main function in C

In #5, we introduce the above data structure, with the exercise of drawing a line and filling various shapes of various colors (in prior to the exercise, a sample program to fill tow circles is presented and explained). #6 is an integrated exercise, and report assignment is "to create a picture which you feel beautiful." We also provide sample method to fill triangle, thick line, oval, and arbitrary convex polygon; students are expected to combine them to construct their code.

#7 through #10 is used to present various CS ideas to students. Those topics are relatively independent, so the difficulty in understanding one of them will not affect the learning of later weeks.

#11 is the initial week for C programming. For a smooth transition, we presente Figure 6 to urge similarity among Ruby and C, and again start from the C version of `triarea` (area of a triangle) example. We prepared many exercise problems identical to Ruby section, and specified the number of problems to be reported as "ten" (in contrast to "one" in other weeks) to urge sufficient practicing. Additionally, the latter half contains topics on solving $f(x) = 0$ equation (with enumeration, binary search, and Newton's methods). This part is for skilled students, just as in numerical error part of #1. All of the following weeks (except for the integrated exercise week) contains advanced topics for the same purpose.

In #12, major focus is on an array. However, as C language handles array access as pointer operation, concepts of address and pointers are also covered. Exercises are mostly straightforward array manipulation. As an advanced topic, dynamic programming is included.

In #13, string (array of character) is introduced, and basic string operation (string length, character replacement and the like) are cast as exercises. As an advanced topic, pattern matching with recursive function is included. Handling of 2-dim arrays in C is also included to meet practical needs.

In #14, struct (record) type is introduced with several sample functions manipulating 24bit RGB color data (1 byte for each color). Exercises are also on the same RGB color structure. As advanced topics, dynamic memory allocation and linked list construction are provided.

#15 is the final week with integrated exercise. Exercise problem is to generate frame animation (actually a series of PPM files, which can be combined to GIF animation using Unix command). Additionally, the topic of organizing a C program into multiple source files (including header files) is explained, and animation

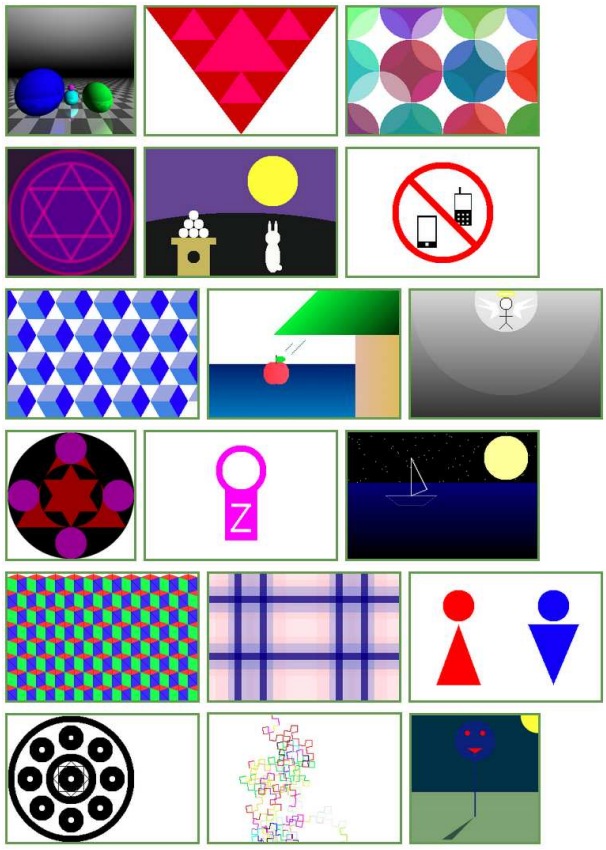

Figure 7: Some pictures in FP2018 report #6

exercise must be carried out by a team of 2 to 3 students. These topic prepare our students toward larger program and team development.

4.3 Experiences of FP2018

Here we report experiences for the school year 2018. Week #1-#3 (introduction part) went smoothly. In #4, many students wrote that simple recursion (such as factorial) was easy to understand. However, for more difficult recursion (such as permutation), included as an advanced topic, exercise was attempted by an only a small portion of the students.

When the topic was changed to pictures in #5-#6, many students felt it as difficult, but along with the exercises they understood gradually, and could submit many interesting pictures (Figure 7 shows some of them) with integrated exercise report.

Middle section #7-#10 contains various relatively independent topics, and preferable topics seemed to differ among student by student. In #7 (topics of sorting), many student had difficulties in manipulating arrays; we felt that exercises in array introduction week (#3) were not sufficient, and planning to improve them. In #8 (topics of complexity and random algorithm), many students seemed to enjoy writing simulations such as "coin toss game" or "dice roll game." Object orientation (#9) is a seemingly difficult topic, but many students have commented that such kind of "packaging" will be useful in practical software development. #10 is the final week for Ruby, and we thought that dynamic data structure (single liked list) will be difficult. However, many students could solve several basic exercises.

From #11, C section starts. Students had little problems with #11 because exercises are generally easy and identical to corresponding Ruby version, except for advanced $f(x) = 0$ part.

However in #12, when address and pointer are introduced and array access defined as pointer arithmetic plus dereference, many students had trouble with understanding what is explained. Yet, the could solve easy exercises based on Ruby experience (semantics of `a[i]` is the same in Ruby and C after all). However, understanding accurate semantics in C is important to deal with more complex C programming.

Another problem is that students are not very fluent with array handling; perhaps exercises on arrays are not enough in earlier weeks (around #3). In #13, students are required to act upon C string, or array of characters, and again lack of skill with array manipulation was the problem. On the other hand, structure in #14 was not much problem because we restricted the topic to simple operation only. On this part (#12-#14), "advanced" topics were not problematic because only skilled students have attempted corresponding exercises.

Finally, #15 is the second integrated exercise of team task to develop an animation generating program. As students are already used to picture generation in #6, what is new was principles of animation plus team development, both of which were not easy but doable. Also, note that the complexity of C program varies according to what kind of animation to generate. For example, minor modification of supplied sample programs is not very difficult. Therefore, this exercise seemed to adapt well against difference in students' levels.

4.4 Evaluation method for FP2018

As stated in Policy 5 of section 4.1, the goal of the course is program construction skills, so we wanted all of our end-term exam problem to be program construction problems. Additionally, we wanted to use automatic scoring because we use CBT (computer-based test) and the number of students is large.

Many CBT sites use Multiple-Choice (MC) tests, in which a list of choices is presented to the examinee, and he / she chose one of the choices as the answer. Fill-in-the-Hole (FH) tests can be considered as a variant, in which in which there are several "holes" in the problem sentences and examinee answer the words appropriate for those holes. MC tests are popular because they can be scored by program easily. However, regarding the evaluation of programming skill, appropriateness of MC tests are doubtful. We have previously experienced the case in which examinee who passed the MC-type programming skill tests could not actually write a program from the scratch.

Constructed-Response (CR) tests are alternatives to MC tests, and are also widely used. In CR tests, the examinee writes answer essays, math proofs, or program codes in answering area (or type them in for CBT). CR tests on programming tasks are widely

Figure 8: A SP test for #1 practice

used when one wants to asses examinees' programming skills accurately, because the task of "writing program down" is identical to the actual programming task. Therefore, we wanted to use the CR test for our end-term test. However, there is a problem — automatic scoring of program code was difficult (there are several attempts, but they are not widely used yet).

As a solution to the problem, we have developed the Split-Paper (SP) test. Figure 8 is a screenshot of the SP practice problem used in our course. Stated problem is: "Construct a Ruby method which accepts x and returns $x^4 - x$ as the result," and the answer is filled halfway. Students drag lines from the choice are (left area) to the code area (right area), to construct the answer program (indentation is applied automatically). The upper area shows the list of symbols (Japanese KATKANA characters in our case) corresponding to the choice lines; this list is collected by the custom Moodle module and used for scoring. For the students to get accustomed to SP test, we provided two practice SP tests for all the #2-#14 weeks.

For the end-term exam, we used an 80-minute CBT on Moodle, with all problems being SP program construction. As in CL, we prepared seven sets with similar problems. The number of problems was 28. We uniformly gave 2 points for a correct answer (the answer that matches one of the correct programs), and 1 point with one difference from either of the correct programs; we used edit distance to compute this difference. Exam scores are scaled and added with report scores for the final mark.

4.5 Results of FP2018

Figure 9 shows the accumulated points of report score. As in CL, frequency around 50 points (all B / b) is highest; most students submitted an ordinary report.

Figure 10 shows the distribution of exam score for the 2017 and 2018 school year. Those distributions, especially the one for the 2018 school year (which is fairly close to normal distribution)

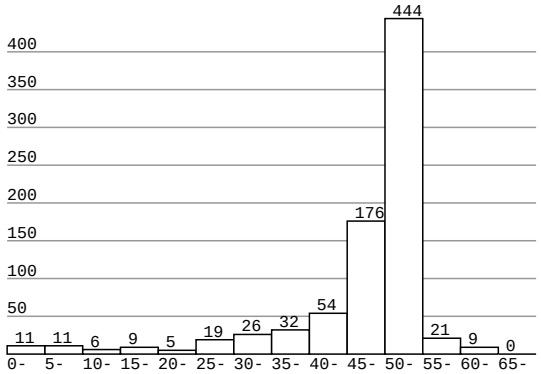

Figure 9: Histogram for FP report scores (n=854)

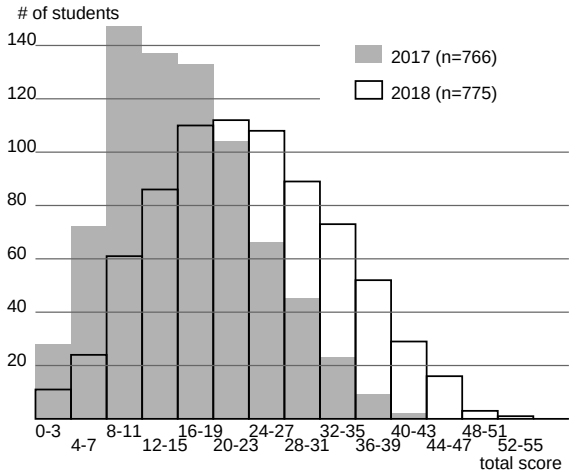

Figure 10: Hisogram for FP2017 and FP2018 exam scores

suggests that the SP format programming test is an appropriate method for grading students.

Improvement in 2018 was due to (1) improvement of problem set difficulty and (2) improvement of curriculum and teaching materials/methods, as we expected.

In 2018, 95% of the students earned 8 (corresponding to 4 problems) or more points, and 77% earned 16 (corresponding to 8 problems) or more points. As we consider it difficult to obtain SP test scores if one cannot write the corresponding program, we assumed that many students in our classes actually obtained abilities to write simple programs, which is the goal with our FP course.

Table 5 and 6 are a summary of inquiry in the last class hour (#15). As in CL, students seem to recognize the class as useful, and they have learned much from the course.

5 RELATED WORKS

There are many attempt toward effective learning on both computer education (as in CL) and programming education (as in FP).

Among them, those targeted toward self-direction are close to our approach. Isomöttönen et al. (Isömöttonen, V., Tirronen, V.,

Table 5: [FP] Q. Was the course useful to you?

Positive	Weak Positive	Neutral	Weak Negative	Negative	No Answer
186	331	145	20	12	15
26.2%	46.7%	20.5%	2.8%	1.7%	2.1%

Table 6: [FP]Q. Have you learned much in the course?

Positive	Weak Positive	Neutral	Weak Negative	Negative	No Answer
279	283	92	26	15	14
39.4%	39.9%	13.0%	3.7%	2.1%	2.0%

2013) describe their self-directed approach to basic programming courses. In their course, practice session, independent work days and review session are repeated to enhance self-directed learning. In their approach, the lecturer has to read many lines of code, and review session focus each students' code; both are not practical in our case of 800 students. Instead of the review session, our FP textbook provides detailed explanations of exercise problems with sample answers at the beginning of next week's chapter, and students study the material by themselves.

Osborne (Osborne, L. J., 2006) reports course targeted to group work and report writing in advance to technical courses, in order to develop a habit of learning. In our courses, several group works are included, and the student must submit the report each week; the net effect of these design might be similar to the above case.

Ott et al. (Ott, C., Robins, A., Shephar, K., 2016) state the importance of feedbacks in computer science education and propose rules toward good feedback practice. Although the volume of direct feedback to individuals is small in our courses (due to large class size), we also provide feedback mechanism in response to the submitted report.

Automatic evaluation of programming skills plays an important role in our FP design, because only with such measure, we can implement "fully program construction" end-term exam with the least grading cost. However, Multiple-Choice (MC) tests are not suitable for the purpose. Simskin et al. (Simskin, M. G., Kuechler, W. L., 2004) compares MC tests against CR test in detail and concludes that MC test which measures ability same as CR test (write a program on a paper) can be constructed, yet construction is very difficult. We share the same view. On the other hand, our SP tests problem can be made with a small cost; we just create a correct program, split them line by line, shuffle them and add some extra choice lines.

There are other researchers investigating programming tests of similar format (Parsons, D., Haden, P., 2006); they use the term "Parsons problems" or "code mangler tests." Denny et al. (Denny, P., Luxton-Reilly A., Simon B., 2008) and Cheng et al. (Cheng, N., Harrington, B., 2017) both investigated correlation of SP-like test score with CR tests, and report moderate correlation. One of the authors has obtained a similar result (Kuno, Y., Nakayama, Y., Kakuda, H., 2019). Therefore, we consider that our approach of grading FP course with SP end-term tests is reasonable.

6 DISCUSSION AND CONCLUSION

We described the design and experiences of CL and FP courses for the freshmen year. Both courses have limited class hours (15 weeks of 90 minutes each), and we have to prepare students toward advanced informatics materials in the sophomore year. Contents for the courses are introductory programming for FP, and various informatics-related topics (networks, principles of computers, software development, HTML/CSS and LaTeX) for CL.

There are many contents in both courses, and exercise is mandatory to learn them in spite of limited class hours. Moreover, there is a large difference in students' background, so we have to avoid novices' dropping out and experts' boredom at the same time.

Our solution to the above problems is use of flipped classrooms, many exercise problems from which student choose a few to report (for every week), and grading scheme which covers both novices (all B report results in 50 points for report part) and experts (good final grade requires a good score in the end-term exam). To implement those solutions as a whole, blended learning settings with Moodle LMS is extensively used (distribution of materials, practice pages, report acceptance / returning, fully CBT end-term exams).

That design seems successfull, as many students commented that studying not restricted to campus is a good point, and the majority of students earnded close to 50 points in report score.

Difficulty in lerarning programming is also a grave problem to be overcome in the FP course. To attack the problem, we used policies "take off first," "varying levels of exercises," "take precedence on practice," "encourage novices," "goal as acquiring programming skills," "no single correct program," and "code out of your brain." These also seem successful, and the majority of our students acquired skills to write at least simple programs. Use of SP tests in the end-term exam with automatic scoring was successfull and useful at least for our class settings.

As a final remark, lots of feedback from the students (mostly through activity reports and assignment reports in every week) have been valuable for us to continuously improve our course design and contents settings, which is an important benefit of our blended learning design.

REFERENCES

Cheng, N., Harrington, B. (2017). *The Code Mangler: Evaluating Coding Ability Without Writing Any Code.* Proc. SIGCSE'17, pp. 123-128.

Denny, P., Luxton-Reilly A., Simon B. (2008). *Evaluating a New Exam Question: Parsons Problems.* Proc. Fourth Intl. Workshop on Computing Education Research 2008 (ICER'08), pp. 113-124.

Isömöttonen, V., Tirronen, V. (2013). *Teaching Programming by Emphasizing Self-Direction: How Students React to the Active Role Required of Them?.* ACM TOCE, vol. 13, no. 2, article 6.

Kuno, Y., Nakayama, Y., Kakuda, H. (2019). *Appropriateness of Split-Paper Test Scores as Programming Performance Metrics.* in submission.

Osborne, L. J. (2006). *Thinking, Speaking, and Writing for Freshmen.* Proc. SIGCSE'06, pp. 112-116.

Ott, C., Robins, A., Shephar, K. (2016). *Translating Principles of Effective Feedback for Studentsinto the CS1 Context.* ACM TOCE, vol. 16, no. 1, article 1.

Parsons, D., Haden, P. (2006). *Parsons Programming Puzzles: A Fun and Effective Learning Tool for First Programming Courses.* Proceedings of the 8th Australasian Conference on Computing Education (ACE'06), vol. 52, pp. 157-163.

Simskin, M. G., Kuechler, W. L. (2004). *Multiple-Choice Tests and Student Understanding: What Is the Connection?.* Decision Science Journal of Innovative Education, vol. 3, no. 1.