# Applying cooperative learning with role division to learn recursion

**YungYu ZHUANG*, Hong-Wei CHEN, Jen-Hang WANG & Gwo-Dong CHEN**
*Department of Computer Science and Information Engineering, National Central University, Taoyuan City, Taiwan*
*yungyu@ncu.edu.tw

**Abstract:** Nowadays, most programming languages support multiple programming paradigms, enabling the choices between them for different application scenarios and even using them together, for example, object-oriented programming for system modeling, event-driven programming for graphical user interface, and functional programming for wrapping actions. However, learning different programming paradigms is quite difficult, especially when they are given as a whole. Recursion is a typical example since it originated in functional programming but was also included in most structured programming languages. As Dijkstra pointed out, learners who are familiar with structured programming tend to think about the control flow of recursion and get confused by how the recursion can be implemented. To address this issue, we developed a methodology along with system software to help novices think of recursive solutions from the definition rather than the implementation.

**Keywords:** Programming, programming paradigm, recursion, cooperative learning, role division

## 1. Introduction

Learning programming is learning how to appropriately use programming paradigms to implement software and organize program code (Floyd, 1979), for example, learning how to use objects, events, and functions. For a specific application scenario, programmers need to appropriately use the programming paradigms supported in the programming language they are using to implement software features. In other words, programming paradigms are the methodology to organize programs for specific application scenarios and the way to implement parts of a software system, which enables programmers to express their ideas with concrete program code. Without knowledge of programming paradigms, writing a good program to fit the system requirement is impossible.

Programming paradigms can be supported with either language constructs or design patterns. Language constructs are the essential elements in a programming language to form a programming paradigm, basically supported by reserved keywords and language syntax. For example, the “`if-else`” and “`for`” structured programming and “`class`” for object-oriented programming; programmers can easily use them for desired functionalities. On the other hand, design patterns are developed based on existing language constructs to achieve functionalities that are not directly supported by the language. In other words, following the developed patterns to write a piece of code to implement functionality such as “`for`” or “`class`.” In languages that do not directly support event-driven programming, for example, C++ and Java, programmers can implement events with the Observer pattern (Gamma et al., 1993), while in a language like C#, programmers may simply use the language constructs named events and delegates.

Modern programming languages tend to support multiple programming paradigms to meet different programmers’ needs and even allow programmers to use several paradigms together. For example, languages like C++ and Python support both procedural programming and object-oriented programming and allow programming with functional flavor. Scala supports object-oriented programming and functional programming. In fact, mainstream languages in the industry, such as C language, also support functional-like things in addition to procedural programming, for example, recursion. Supporting multiple programming paradigms in a single programming language is a trend.

However, learning multiple programming paradigms at the same time is quite difficult. Learners may get confused about what programming paradigms actually are, how they are supported, and when they can be applied. We also found such a situation in classrooms where students have difficulty switching between programming paradigms and using them correctly. As Dijkstra mentioned, students usually struggle against learning recursion because they consider recursion based on their understanding of structured programming rather than from the viewpoint of definition (Dijkstra, 1999). Once students start trying to think and implement recursion with structured programming, i.e., sequence, branch, and loop (Dahl, Dijkstra, & Hoare, 1972; Nassi & Shneiderman, 1973), they will get confused about the execution of every recursive step since recursion cannot be natively represented with structured programming and flowcharts. Although it is possible to convert a recursive call into an iteration with appropriate symbols and conditions, the representation will become iterative style---the recursive call will be represented by repetitive steps rather than a comprehensive step. Actually, students must recognize that recursion is a definition and use it without tracing each step. Therefore, how to help novices learn how to use programming paradigms is an important issue.

The observation led us to think about how to help students think and learn individual programming paradigms. So far as we know, most research activities on improving programming learning discuss programming as a whole or stay in discussing structured programming. In this paper, we take the recursion in functional programming as an example to point out how different programming paradigms and the problems learners may face when learning programming. To address this issue, we developed a methodology to help novices learn from the definition of recursion. We designed a system to let learners focus on the definition and benefit from cooperative learning with role division.

## 2. Literature review

In this section, we discuss existing research activities on programming paradigms, cooperative learning, and recursion to clarify the research gap, i.e., the importance and difficulty of learning recursion.

### *2.1 Programming paradigm*

Programming paradigms are how to implement and organize program code, which often determines whether programmers use a language (Floyd, 1979). There have been many programming paradigms developed for resolving different application scenarios. For example, structured programming was suggested to improve the structure of program code (Dahl, Dijkstra, & Hoare, 1972). Procedural programming was discussed to divide program code into numbers of procedures or subroutines. Functional programming was developed to make code declarative and reduce side effects (Hughes, 1989). Object-oriented programming was proposed to model a system with message sending protocol (Goldberg & Robson, 1983). Aspect-oriented programming was developed to modularize crosscutting concerns (Kiczales et al., 1997). Reactive programming follows the idea of data-flow programming to simplify further the usage of events in programs (Zhuang, 2019). Programming paradigms play an essential role in practicing the separation of concerns, which lets programmers focus on implementing and modifying the code of a particular concern (Dijkstra, 1974). The support of the programming paradigm is the most crucial feature in programming language design.

#### *2.1.1 Structured programming and procedural programming*

Structured programming is an early milestone in the history of programming languages. It is to encourage programmers to represent their algorithms with the three types of decomposition: concatenation (sequence), selection (branch), and repetition (loop) (Dahl, Dijkstra, & Hoare, 1972). Structured programming also encourages programmers to consider block and scope and represent them in the form of flowcharts (Chapin, 1970; Nassi & Shneiderman, 1973).

Procedural programming suggests considering a program as a series of procedure calls, which can be regarded as a further step toward modularization upon imperative programming and structured programming. It emphasizes the usage of modular procedures to express computational steps in the algorithm to implement. Nowadays, structured programming and procedural programming are supported in most imperative programming languages, such as C and C++.

#### *2.1.2 Functional programming*

Functional programming has its origin in lambda calculus and was introduced in early programming languages for artificial intelligence, such as Lisp. It naturally supports recursion without transferring it into repetition and can easily represent recursive ideas in computer science. Although the mainstream languages in the industry nowadays are imperative, functional programming elements like recursion and lambda are partially introduced, i.e., programmers can write imperative programs with "functional flavor."

On the other hand, pure functional programming languages are continuously used in academia and now going popular. Several functional programming languages, such as Haskell and Scala, have a large community and are used in industry.

### *2.2 Cooperative learning and functional roles*

Cooperative learning has shown its effectiveness in education (Roger & Johnson, 1994; Slavin, 1980), including programming learning (Chu & Hwang, 2010; Hwang et al., 2012). Those research activities have reported cooperative learning can increase student achievement. With different kinds of cooperative learning approaches, students' learning performance may be improved (Sharan, 1980). In cooperative learning, positive interdependence can be considered a critical element (Roger & Johnson, 1994), which means students in a cooperative lesson have to ensure not only themselves but also group members can learn the assigned material well. In other words, for group success, all group members need to devote efforts and make their contributions to the joint effort. In such group training and learning, compared with classifying all group members as a single category, giving group members different functional roles is more effective for group growth and production (Benne & Sheats, 1948; Benne & Sheats, 2020).

### *2.3 Recursion*

Recursion is not only an important thinking method but also a representative approach used in computer science. Many important problems and algorithms in computer science are based on recursion, for example, the traversal of trees and graphs. Although the concept of recursion is quite natural, it is always a big challenge for novice programmers. One of the reasons might be that recursion has its root in functional languages like Lisp. At the same time, it was introduced in imperative languages like C. Dijkstra also observed that students who are familiar with Fortran tend to have difficulty in learning recursion since Fortran does not support recursion. Kahney concluded the problem of learning recursion comes from an incorrect understanding of recursion (Kahney, 1983). As Rinderknecht mentioned, many recursion teachings focus on control flow rather than definition (Rinderknecht, 2014).

Although the difficulty of learning recursion is not a new issue, there were rarely research activities on improving it from the viewpoint of definition. Recent research, RecurTutor, emphasized the necessity of practice and showed convincing experimental results (Hamouda et al., 2018). However, according to the findings in the above research results, how to afford a better understanding of recursion from the viewpoint of definition should be more critical. This observation motivated us to design a different learning method for understanding recursive programming.

## 3. Methodology

To help novices learn the concepts in different programming paradigms, such as recursion, we propose applying cooperative learning with role division to programming learning. We developed a methodology along with a concrete system to let learners focus on the definition of recursion without thinking about the elements in other programming paradigms.

### *3.1 The definition of recursion*

The recursive solution to a problem is to consider and resolve a smaller instance of the same problem (Roberts, 1986). In programming, recursion is performed by letting functions call themselves with smaller data sizes, i.e., reducing the problem size to resolve. Once the functions are called with the base cases that cannot be reduced anymore, the problem can be easily resolved. Therefore, we can define a recursion solution to a given problem as the following three elements: base case, reduction, and rewrite.

Here we use the example of calculating factorial numbers in mathematics to explain the three elements. As a frequently used number in mathematics, the factorial of a non-negative integer `n` is denoted by `n!`, which is the product of all positive integers less than or equal to `n`. For example, factorial three can be calculated as follows:

```
3! = 3 * 2 * 1
```

And actually, it is equal to `3 * (3-1)!`:

```
3! = 3 * 2!
```

For the problem of calculating the factorial number of a given `n`, we can consider the three elements for the recursive solution based on its definition. According to the definition of factorial numbers, the simplest case without any calculation is the case of zero, and the factorial number of zero is one:

```
0! = 1
```

Hence, we can set the base cases of our recursive solution to them, try to reduce the given `n` to one or zero, and rewrite the rules to combine the calculation as shown in Table 1. Suppose we use a function named `fact(n)` to represent the calculation of factorial `n`. The three steps of thinking the recursive solution will be considering the following three questions in order:

1. How to handle the base case(s) for the given problem?
2. Can the problem be reduced to the same problem(s) with a smaller problem size?
3. How to rewrite the problem in the form of the same problem(s) with a smaller problem size?

Table 1. *The three elements in the recursion definition*

| Element | Meaning | Example |
|---|---|---|
| Base case | find out the extreme cases of the problem that can be simply calculated or got | `0! = 1` |
| Reduction | move one step forward by reducing the problem size | `(3 - 1)!` |
| Rewrite | rewrite the given problem statement with a combination of smaller ones | `3! = 3 * 2!` |

### *3.2 Division of roles*

In order to let learners focus their attention on the recursion definition itself, the task of thinking of a recursive solution is allotted to three roles: base case finder, problem size reducer, and problem statement rewriter, as shown in Figure 1. By dividing the task of thinking of a recursive solution into three parts, each learner can concentrate on only the assigned part without considering the whole problem once. The role division can relieve the difficulty and help learners start by thinking about a part of the problem.

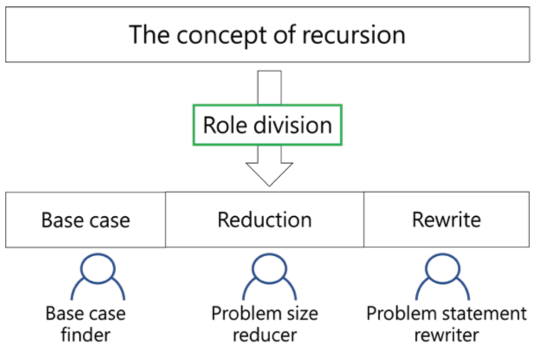

*Figure 1.* The role division is based on the three elements in the recursion definition.

### *3.3 Learning design*

The learning design of our methodology is shown in Figure 2. Students in a classroom are first grouped to learn a given problem, where three students per group. After the teaching activities given by the lecturer, students start to think about a part of the problem based on the roles assigned to them, i.e., base case finder, problem size reducer, and problem statement rewriter. When students finish their work, our system will compose their thinking result to compose the solution to the given problem. After completing the given problem, students need to exchange their roles for the next problem to learn how to think about different parts of the problem. Such a design is to help learners think of the solution according to the definition without thinking about control flow.

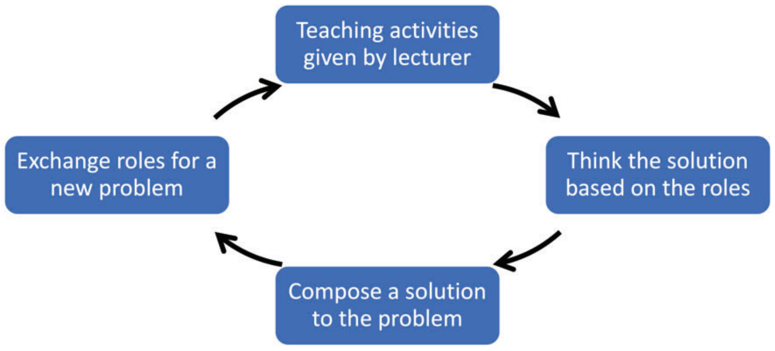
Teaching activities given by lecturer
Think the solution based on the roles
Compose a solution to the problem
Exchange roles for a new problem

*Figure 2.* The learning design of our methodology.

### *3.4 System implementation*

Based on our proposed learning design, we developed a learning system software to support learners. The system is implemented in C# programming language with TCP/IP socket connection. Each group member can launch a client on the computer individually and switch to the mode for the assigned role. Node.js and WPF (Windows Presentation Foundation) frameworks are used for the server and client user interface. The object language generated by the system is Python, which will be shown on the client user interface to help learners verify their thinking results.

## 4. Conclusions and future work

We took recursion as an example to discuss the problem novices may face when they learn various programming paradigms. A methodology using cooperative learning with role division was developed along with a concrete system to help learners focus on the definition of recursion. We have conducted an experiment based on our methodology and system interface compared with learning in an integrated development environment. We are analyzing the experimental results to give a detailed report in a journal paper. In the future, we plan to extend our system to support more complicated recursive problems such as backtracking usage. Currently, our system can only cover typical recursive problems due to the implementation of our system. Showing the effectiveness of our methodology in learning other programming paradigms is also included in our future work.

## Acknowledgements

This work was supported by the National Science and Technology Council, Taiwan [Grant No.: MOST 110-2511-H-008-003-MY2].

## References

Benne, K. D., & Sheats, P. (1948). Functional roles of group members. *Journal of social issues*, *4*(2), 41-49. https://doi.org/10.1111/j.1540-4560.1948.tb01783.x

Benne, K. D., & Sheats, P. (2020). Functional roles of group members. In *Shared Experiences in Human Communication* (pp. 155-163). Routledge.

Chapin, N. (1970). Flowcharting with the ANSI standard: A tutorial. *Acm computing surveys (csur)*, *2*(2), 119-146.

Chu, H.-C., & Hwang, G.-J. (2010). Development of a project-based cooperative learning environment for computer programming courses. *International Journal of Innovation and Learning*, *8*(3), 256-266. https://doi.org/10.1504/IJIL.2010.035029

Dahl, O.-J., Dijkstra, E. W., & Hoare, C. A. R. (1972). *Structured programming*. Academic Press Ltd.

Dijkstra, E. W. (1974). *E.W. Dijkstra Archive: On the role of scientific thought (EWD447)*. Retrieved Dec 23 from https://www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD447.html

Dijkstra, E. W. (1999). *E.W. Dijkstra Archive: Computing Science: Achievements and Challenges (EWD1284)*. Retrieved Dec 30 from https://www.cs.utexas.edu/users/EWD/transcriptions/EWD12xx/EWD1284.html

Floyd, R. W. (1979). The paradigms of programming. *Commun. ACM*, *22*(8), 455–460. https://doi.org/10.1145/359138.359140

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1993). Design patterns: Abstraction and reuse of object-oriented design. European Conference on Object-Oriented Programming,

Goldberg, A., & Robson, D. (1983). *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc.

Hamouda, S., Edwards, S. H., Elmongui, H. G., Ernst, J. V., & Shaffer, C. A. (2018). RecurTutor: An interactive tutorial for learning recursion. *ACM Transactions on Computing Education (TOCE)*, *19*(1), 1-25.

Hughes, J. (1989). Why Functional Programming Matters. *The Computer Journal*, *32*(2), 98-107. https://doi.org/10.1093/comjnl/32.2.98

Hwang, W.-Y., Shadiev, R., Wang, C.-Y., & Huang, Z.-H. (2012). A pilot study of cooperative programming learning behavior and its relationship with students' learning performance. *Computers & Education*, *58*(4), 1267-1281. https://doi.org/https://doi.org/10.1016/j.compedu.2011.12.009

Kahney, H. (1983). What do novice programmers know about recursion. Proceedings of the SIGCHI conference on Human Factors in Computing Systems,

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., & Irwin, J. (1997). Aspect-oriented programming. European conference on object-oriented programming,

Nassi, I., & Shneiderman, B. (1973). Flowchart techniques for structured programming. *SIGPLAN Not.*, *8*(8), 12–26. https://doi.org/10.1145/953349.953350

Rinderknecht, C. (2014). A survey on teaching and learning recursive programming. *Informatics in Education-An International Journal*, *13*(1), 87-120.

Roberts, E. (1986). *Thinking recursively*. J. Wiley.

Roger, T., & Johnson, D. W. (1994). An overview of cooperative learning. *Creativity and collaborative learning*, 1-21.

Sharan, S. (1980). Cooperative Learning in Small Groups: Recent Methods and Effects on Achievement, Attitudes, and Ethnic Relations. *Review of Educational Research*, *50*(2), 241-271. https://doi.org/10.3102/00346543050002241

Slavin, R. E. (1980). Cooperative Learning. *Review of Educational Research*, *50*(2), 315-342. https://doi.org/10.3102/00346543050002315

Zhuang, Y. (2019). A lightweight push-pull mechanism for implicitly using signals in imperative programming. *Journal of Computer Languages*, *54*, 100903. https://doi.org/https://doi.org/10.1016/j.cola.2019.100903