Pedagogical Implications of Parser Combinators in Programming Languages Courses: A Comparative Study*

Abbas Attarwala[1], Pablo Raigoza[2]
[1]Computer Science Department
California State University
Chico, CA 95973
aattarwala@csuchico.edu
[2]Computer Science Department
Cornell University
Ithaca, NY 14850
pr428@cornell.edu

Abstract

This paper recounts the experience of teaching parser combinators in a programming language course using OCaml at both Boston University and California State University, Chico. The main focus is on how parser combinators are introduced when teaching parsing to students who are new to functional programming. Techniques such as boxes and color coding are employed to simplify the understanding of the concepts. Furthermore, teaching course evaluation data are presented to compare course outcomes, contrasting semesters when parser combinators were not used with those when they were incorporated into the teaching. Reflections and feedback from students provide insight into the effectiveness of these teaching methods. Additionally, a two-tailed Welch t-test is conducted on the teaching course evaluation data to assess the impact of using parser combinators.

1 Introduction

This paper discusses the benefits of teaching parser combinators in a third-year programming language course. The box representations for the parser and the color coding for the parser combinators are introduced and explored, drawing on experiences from Boston University (BU) and California State University, Chico (CSU Chico). A common challenge that I[1] have observed is that students initially create ad hoc parsers based on the context-free grammars (CFG) provided for their projects. However, when project requirements evolve and a revised CFG is introduced in later parts of the project, these initial parsers do not scale well. As a result, students often face difficulties and must completely rewrite their parsing code to accommodate the new requirements. I propose that parser combinators could be a solution, potentially enabling students to develop parsers that are more adaptable and scalable. To evaluate this hypothesis, this paper compares and analyzes my teaching course evaluation data across various semesters, focusing on student performance and adaptability in courses taught with and without parser combinators.

In this paper, we define a parser as a function that takes a string as input and produces an output that is either a tuple consisting of the parsed value and the remaining unconsumed part of the string, or an indication that the parsing has failed. A parser combinator is a higher-order function that takes one or more functions or parsers as input and returns a new parser as output. It allows for the construction of complex parsers by combining simpler components in a modular and reusable way.

2 Literature Review

In our exploration of teaching methodologies for parser combinators, we have identified a notable gap in the existing literature. Although there is extensive documentation on the technical advantages and applications of parser combinators, their pedagogical aspects have been largely overlooked.

Parser combinators [10], offer functional programmers a clean and flexible method for constructing parsers. This flexibility is attributed to the abstraction they provide, distancing the programmer from complex parsing machinery. However, [10] also presents a trade-off: This abstraction comes with the cost of executing the combinators and the functions that build them, often necessitating repetitive execution. [6] mentions that parser combinators are used in parsing sequences generated by CFG, in specialized data formats like JSON and YAML, and markup languages such as XML and HTML. Their paper also

[1] First person in this paper refers to Abbas Attarwala

illustrates the use of parser combinators in programming language processing, specifically in identifying syntax errors.

Furthermore, [8, 9] emphasize the balance the parser combinators maintain between flexibility and abstraction. Parser combinators enable the creation of parsers in a style that remains close to the CFG. Highlighting the role of higher-order functions [7], along with [5, 10], points out the strengths in developing combinator libraries, particularly parser combinators. They underline the beauty of these abstractions in functional programming, but also note the scarcity of literature on their practical, maintainable, and scalable use.

Our research provides valuable insights into the pedagogical effectiveness of parser combinators. By conducting a comparative study of course evaluations from semesters with and without their use, we aim to demonstrate their impact on student learning. While the current literature thoroughly explores the technical strengths and applications of parser combinators, our research examines their pedagogical value, offering a different perspective in the realm of functional programming education.

3 Parser Combinators

In the Summer of 2020, while teaching CS 320 at BU my students built an interpreter for a stack-based programming language. They were provided with the initial CFG and a set of operational semantics. As the project progressed, new features were added, such as nested conditional statements. Initially, students created parsers using regular expressions or some complex parsing involving a stack, but these were ad hoc and struggled to adapt to the evolving grammar, leading to significant rewriting and frustration for both students and myself.

To address this, I integrated parser combinators into the curriculum, beginning at BU in the Fall of 2020 and continuing through Fall 2022 at CSU Chico. Before introducing this concept, I engaged students with a simple OCaml exercise involving string parsing to demonstrate the practical benefits of parser combinators. In this simple example, I ask my students to write an OCaml code to parse the first three characters of a string but only if it begins with ‘a’, followed by ‘b’ followed by ‘c’. The code in OCaml is shown in Listing 1. I provide my students with `getFirstCharacter` function which accepts a `string` and returns back a `option` tuple of the extracted first character and the unconsumed string.

```
let parse s =
  match (getFirstCharacter s) with
  | None -> None
  | Some (firstC, rest) -> if firstC = 'a' then
      (match (getFirstCharacter rest) with
      | None -> None
```

```
7       | Some (secondC, rest) -> if secondC = 'b' then
8           (match (getFirstCharacter rest) with
9           | None -> None
10          | Some (thirdC, rest) -> if thirdC = 'c' then
11              Some (true, rest)
12              else None)
13          else None)
14      else None
```

Listing 1: Parsing code without using parser combinators

Students quickly realize that while the initial code works, it's not scalable and becomes cluttered, especially with numerous error checks for parsing characters other than 'a', 'b', or 'c'. To address these issues, I guide them through refactoring the code using parser combinators.

I define a parser as a function that accepts a `string` and returns an `('a, string) option` type in OCaml. The `option` type indicates that the function returns `None` if parsing fails, or `Some` if parsing is successful. In the case of success, the `Some` tuple contains two elements: the parsed value (represented by the type variable `'a`) and the remaining unconsumed string. I represent parsers as boxes: the input is a `string` and the output is an `('a, string) option` as seen in Figure 2. Another way to think about parser combinators is like a glue that combines two parsers to create a new parser. The `»` operator is a parser combinator, commonly referred to as the sequencing operator, which I implement during my lecture. It not only links two parsers—`p1` and `p2`—to form a new parser `p3` (`let p3 = p1 » p2`), but also establishes a dependency where `p2` runs only if `p1` succeeds, passing the unconsumed string from `p1` to `p2`. The type of `»` is defined as `'a parser ->'b parser-> 'b parser`. Following OCaml's operator naming conventions, the operator `»` associates to the left. Figure 1 shows the refactored code that parses the first three characters of a string, specifically 'a', 'b', and 'c', using this method.

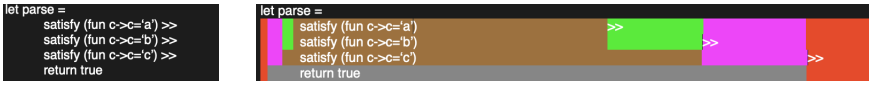

Figure 1: Refactored OCaml code using parser combinator. The same code on the right is color coded to represent each parser. The red parser for instance is a sequence of the purple parser followed by the grey parser. satisfy, `»`, and return are parser combinators.

I demonstrate to my students that `satisfy` is a combinator that verifies if a string starts with a specific character. For example, `satisfy (fun c -> c = 'a')` returns a parser that checks whether the string begins with 'a'. If this check passes, `satisfy (fun c -> c = 'b')` checks for 'b', followed by

`satisfy (fun c -> c = 'c')` checking for 'c'. This sequential checking, facilitated by the » operator, simplifies error handling. The » operator inherently handles errors, so if any parser in the sequence fails, the entire parsing process fails. Unlike the explicit error checks in Listing 1, parser combinators allow students to focus on parsing the required elements without worrying about extensive error handling for intermediate steps.

In Figure 1, I also use color coding for each parser, which students refer to in Figure 2 to visualize how the » operator unpacks the unconsumed string and feeds it to the next combinator in the chain. In the color-coded diagram, the first » in green sequences the two brown parsers to create the green parser. The second » in purple sequences the green parser with the next brown parser to create the purple parser. Finally, the third » in red sequences the purple parser with the grey parser to create the red parser. The red parser returns true if the string begins with 'a', followed by 'b', followed by 'c'.

In Figure 2, I provide a color-coded visual representation of the parser sequence from Figure 1. The colors in the visualization match those used in the OCaml code in Figure 1. The red parser in Figure 2 is a sequence consisting of the purple parser followed by the grey parser. The purple parser itself is a sequence of the green parser followed by the rightmost brown parser. Finally, the green parser is a sequence of two brown parsers. The process begins with the red parser, which takes the input string "abcxyz". This string is then passed through the purple parser to the green parser, and finally to the leftmost brown parser, represented by `satisfy (fun c -> c = 'a')`. The leftmost » that creates the green parser takes the unconsumed string "bcxyz" from the first brown parser and passes it to the second brown parser. The green parser's output is the same as the second brown parser's output. The next » that creates the purple parser extracts "cxyz" from the green parser and feeds it to the third brown parser. The purple parser's output matches the third brown parser's output. Finally, the third » that creates the red parser takes "xyz" from the purple parser and passes it to the grey parser. The `return true` parser adds `true` to the tuple and places the unconsumed string in the second position. The `true` indicates that the parsing has succeeded. Students are encouraged to consider what abstract syntax tree can be returned at this point instead of `true`. The grey parser's output is also the red parser's output. From the red parser's perspective, it processes the input string "abcxyz" and returns the tuple `Some (true, ''xyz'')`.

To accommodate different parsing requirements, such as allowing zero or more spaces between characters, I provide an extensive library of parser combinators, including the `many0` combinator. This combinator takes a parser as input and runs it zero or more times, allowing it to parse sequences where a particular pattern may occur multiple times or not at all. This example effec-

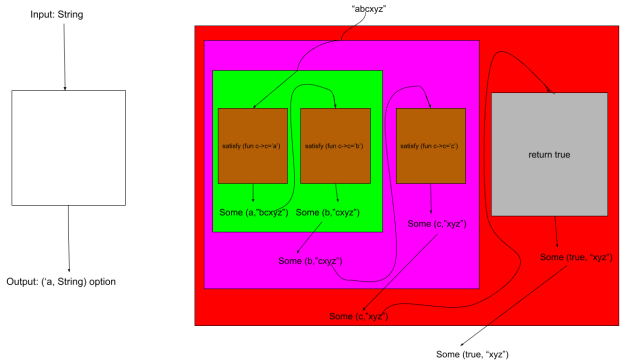

Figure 2: On the left is a box representation of a parser. On the right each color box represents a parser. For instance the purple parser is a sequence of the green parser followed by the right most brown parser.

tively demonstrates to students the ease of adapting parser combinators to new requirements. The new code that accommodates this is shown in Listing 2.

```
let space_parser =
satisfy (fun c -> c = ' '))

let parse_with_zero_or_more_spaces =
  satisfy (fun c -> c = 'a') >>
  many0 space_parser >>
  satisfy (fun c -> c = 'b') >>
  many0 space_parser >>
  satisfy (fun c -> c = 'c') >>
  many0 space_parser >>
  return true
```

Listing 2: OCaml code to parse strings with zero or more spaces between the letter ‘a’ and ‘b’; between ‘b’ and ‘c’ and after ‘c’.

In programming language courses, especially those focused on interpreter creation and extensive parsing, the inclusion of parser combinators is crucial. These combinators not only provide practical exposure to functional programming principles like higher-order functions and immutability but also enhance the readability and maintainability of code. This aligns well with the demands of modern, agile software development, offering flexibility and ease of use for rapid prototyping and adapting to evolving project requirements.

4 Teaching Course Evaluation Data

In Table 1, I present the evaluations for my Summer 2020 programming languages course at BU, which marked my first time teaching an OCaml course without parser combinators, over six weeks via Zoom. The following year, Summer 2021, I introduced parser combinators into the curriculum, dedicating 1.5 weeks to them and incorporating them into half of the assignments, as also detailed in Table 1. The latest evaluations from my Summer 2023 course at California State University, Chico, shown in Table 2, continue to reflect the use of parser combinators over the same six-week Zoom format, allowing direct comparison to the 2020 course without them. I have excluded evaluations from Fall and Spring semesters due to the different 16-week format and mixed in-person/online teaching during the pandemic.

Questions	Summer 2020			Summer 2021		
	N	SD	Mean	N	SD	Mean
The extent to which you found the class intellectually challenging:	16	.79	4.5	15	.96	4.13
The extent that assignments furthered your understanding of course content:	16	1.11	4.38	15	.5	4.47
The instructor's ability to present the material is:	16	.58	4.69	15	1.02	4.6
The instructor's overall rating is:	16	.77	4.69	15	.34	4.87

Table 1: Course Evaluation data from Summer 2020 and Summer 2021 teaching at BU on a scale of 1 (poor) to 5 (superior).

Some feedback from students in the Summer of 2020 at BU:

1. *I think we could have definitely had a little bit more time for the last few assignments as they are harder.*

2. *Problem sets were interesting and challenging.*

Some feedback from students in the Summer of 2021 at BU:

1. *Professor Attarwala was incredible at teaching this class! 320 with him was the most engaging remote class I've been in during the pandemic. His color coding, visualizations, and reinforcements really drilled in the material.*

2. *Professor has a great way of explaining concepts. His enthusiasm is definitely infectious and his use of visual aids especially the virtual blackboard with color-coded notation keeps me excited.*

Questions	N	SD	Mean
The course increased my knowledge of the subject matter:	20	.94	4.55
The assignments helped me understand the material:	20	.94	4.55
The instructor presented in an understandable manner:	20	.93	4.65
How do you rate the overall quality of teaching:	19	.54	4.79

Table 2: Course Evaluation data from Summer 2023 at CSU Chico on a scale of 1 (poor) to 5 (superior).

Some feedback from students in the Summer of 2023 at CSU Chico:

1. *His teaching style allows me to really understand concepts and I love how he visualizes concepts.*

2. *I very much enjoyed the prof. drawing on the board gave very good visuals pointers for the current material that was talked about.*

An analysis of the teaching evaluations in Table 1 and Table 2 reveals a subtle, but informative, trend. The introduction of parser combinators slightly affected the numerical ratings (especially of "The extent to which you found the class intellectually challanging" i.e., it decreased slightly at BU, however a comparable question at CSU Chico suggest that it increased again), but student testimonials emphasize the success of my visual and color-coded teaching methods in enhancing comprehension. When assessing the statement, "The instructor's ability to present the material is:", the numerical ratings decreased during the two semesters in which parser combinators were introduced. Conversely, for "The instructor's overall rating", the numerical ratings increased in the semesters that included teaching with parser combinators. This contrast between quantitative and qualitative feedback highlights the complexity of evaluating teaching effectiveness. While parser combinators increased course difficulty, effective teaching practices ensured positive learning experiences. Future investigations will employ my statistical frameworks [1, 2] to rigorously assess the benefits of parser combinators in programming education.

5 Impact of Parser Combinators on Teaching Effectiveness

In our examination of teaching outcomes, we anticipated that incorporating parser combinators—a notably complex topic—into the programming course curriculum would challenge students intellectually and improve their comprehension of the course material. In our research, the null hypothesis is stated as there is no difference in teaching effectiveness with the integration of parser combinators, and the alternative hypothesis, which anticipated a discernible

impact, whether positive or negative. To investigate these hypotheses, we conducted a two-tailed Welch's t-test with an alpha level set at 5%, which indicates the threshold for rejecting the null hypothesis, across four dimensions of teaching effectiveness from the teaching course evaluations, i.e., (1) The extent to which you found the class intellectually challanging; (2) The extent that assignments furthered your understanding of course content; (3) The instructor's ability to present the material and (4) The instructor's overall rating.

In our analysis, we used Welch's two-tailed t-test and not the Student's t-test. The latter assumes homogeneity of variances across compared groups, the data presented in Section 4 suggests variability in this respect. Therefore, applying the Student's t-test might result in misleading outcomes. Welch's t-test is more appropriate for our data, as it does not require equal variances, as supported by the literature [3, 4]. It offers a more accurate analysis by adjusting degrees of freedom based on the sample sizes and variances of the groups compared. More formally here is how the t-statistic and the degree of freedom are calculated for the Welch's t-test:

$$t\_statistic = \frac{\mu_1 - \mu_2}{\sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}} \qquad Deg\ Freedom = \frac{\left(\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}\right)^2}{\frac{(\sigma_1^2/n_1)^2}{n_1 - 1} + \frac{(\sigma_2^2/n_2)^2}{n_2 - 1}}$$

μ_1, μ_2 are the means of the two groups, σ_1^2, σ_2^2 are their variances, and n_1, n_2 are the sample sizes. Table 3 displays the t-statistic, degrees of freedom, and p-value for each of the four evaluation questions. The comparison is between courses that included parser combinator instruction at BU in Summer 2021 and those that did not in Summer 2020. The table also presents t-statistics, degrees of freedom, and p-values for the same four questions, comparing the Summer 2023 parser combinator courses at CSU Chico with the non-parser combinator courses at BU in Summer 2020.

While statistical significance was not achieved in the results, it is remarkable that student evaluations consistently rated highly across all semesters, including those following the introduction of parser combinators. This also suggests a potential ceiling effect due to the high baseline of teaching performance (see Table 1 when the course was taught without using parser combinators at BU). Qualitatively, student feedback recognized and valued the increased depth and rigor that parser combinators brought to the course. This feedback aligns with my educational objectives of developing analytical skills and equipping students for the intricacies of real-world programming tasks.

The absence of statistically significant differences might be interpreted as an indicator of unchanged teaching effectiveness; however, it may also highlight the robustness of instructional quality in the face of introducing more complex subject matter. Future studies could explore different teaching methods or

examine the long-term effects of incorporating advanced computational concepts into the curriculum. This research could provide valuable information on optimizing educational improvement strategies.

Evaluation Item	With Parser Combinators BU in Summer of 2021			With Parser Combinators CSU Chico in Summer of 2023		
	T-Stat	DF	P-Value	T-Stat	DF	P-Value
The extent to which you found the class intellectually challenging	1.167	27.19	0.2532	-0.173	33.89	0.8634
The assignments helped me understand the material	-0.294	21.13	0.7716	-0.488	29.49	0.6289
The instructor presented in an understandable manner	0.299	21.90	0.7675	0.158	32.30	0.8756
How do you rate the overall quality of teaching	-0.851	20.92	0.4045	-0.437	26.25	0.6658

Table 3: Results from Welch's test comparing teaching evaluations of BU Summer 2021 and CSU Chico Summer 2023 against BU Summer 2020.

6 Conclusion

In conclusion, the integration of parser combinators into the curriculum has been crucial in enhancing both code readability and adaptability, offering students a different application of functional programming principles in the context of parsing strings that is used very often in designing interpreter and compilers. The teaching method I used, which included color-coded diagrams, seems to have helped make parser combinators easier to understand. It is not clear if every student preferred this way, but overall, their feedback did not get worse, even with this challenging topic added to the course.

Although statistical tests did not produce significant results, this should not overshadow the pedagogical benefits observed. The consistency of high student evaluations, even with the incorporation of this advanced topic, suggests that the educational quality was maintained at its usual high standard.

In addition, students have reported that the challenge of engaging with such high-level material has been a rewarding experience. This qualitative feedback highlights the value of integrating such advanced topics into the curriculum, serving as a catalyst for developing critical thinking and problem solving skills.

Acknowledgement

We gratefully acknowledge the use of OpenAI's ChatGPT for proofreading, grammatical checks, and other text editing tasks.

References

[1] Abbas Attarwala. "Live coding in the classroom: Evaluating its impact on student performance through ANOVA and ANCOVA". In: *2023 International Conference on Intelligent Education and Intelligent Research (IEIR)*. IEEE. 2023, pp. 1–6.

[2] Abbas Attarwala and Kun Tian. "A statistical framework for measuring the efficacy of peer review on students' performance". In: *2023 International Conference on Intelligent Education and Intelligent Research (IEIR)*. IEEE. 2023, pp. 1–7.

[3] Marie Delacre, Daniël Lakens, and Christophe Leys. "Why psychologists should by default use Welch's t-test instead of Student's t-test". In: *International Review of Social Psychology* 30.1 (2017), pp. 92–101.

[4] Ben Derrick, Deirdre Toher, and Paul White. "Why Welch's test is Type I error robust". In: *The quantitative methods for Psychology* 12.1 (2016), pp. 30–38.

[5] Jeroen Fokker. "Functional parsers". In: *Advanced Functional Programming: First International Spring School on Advanced Functional Programming Techniques Båstad, Sweden, May 24–30, 1995 Tutorial Text 1*. Springer. 1995, pp. 1–23.

[6] Mikhail Kuznetsov and Georgii Firsov. "Syntax Error Search Using Parser Combinators". In: *2021 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (ElConRus)*. IEEE. 2021, pp. 490–493.

[7] Jamie Willis and Nicolas Wu. "Design patterns for parser combinators (functional pearl)". In: *Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell*. 2021, pp. 71–84.

[8] Jamie Willis and Nicolas Wu. "Design patterns for parser combinators in scala". In: *Proceedings of the Scala Symposium*. 2022, pp. 9–21.

[9] Jamie Willis and Nicolas Wu. "Garnishing parsec with parsley". In: *Proceedings of the 9th ACM SIGPLAN International Symposium on Scala*. 2018, pp. 24–34.

[10] Jamie Willis, Nicolas Wu, and Matthew Pickering. "Staged selective parser combinators". In: *Proceedings of the ACM on Programming Languages* 4.ICFP (2020), pp. 1–30.