CS131 Final Project: Critiquing Programming Languages

Shinn Taniya and Erin Burke

Apr. 13, 2022

1 Overview

This document contains a set of guiding questions and readings to help you prepare for the final project deliverable: a written paper on the impacts of succinctness in programming languages. Your responses to these questions can form the basis of your final paper.

2 Pedagogical Transparency

One goal of this project, on top of thinking critically about PL design and implications, is to explore the value of writing *about* our fields of study, in this case, programming languages.

Read over this very short (2 page) summary of a recent panel discussion entitled "Writing in CS: Why and How?" from SIGCSE 2018's Technical Symposium on Computer Science Education [3]. The PDF is provided on the project page on Sakai.

Question. 1 In a few sentences, based on this reading, reflect on the impact of writing within computer science education from your own point of view.

Answer. 1 We believe that writing is essential within the computer science education. This is because as the reading touches upon, computer science students require communication and critical thinking skills that can be nurtured through writing assignments. Any computer science student will agree that programmers must write clearly and concisely for everything from comments in their code to documentation, as well as for their ability to write code. Hence, interdisciplinary curriculum that merge writing and computer science such as the Writing Enriched Curriculum (WEC) concept touched on by Bruce Maxwell and Philip Barry is especially effective as it allows faculty to construct statements regarding effective types of writing and an evaluation metric for 'good writing' for each type. Writing, therefore, serves as both an end and a means of

expression for programmers, allowing them to conceptualize material, consolidate concepts, and synthesize information from various sources.

3 Succinctness is Power: Summarize and Reflect

3.1 What to do

Read Paul Graham's essay "Succinctness is Power" which you can find on the internet and answer the following questions. You may want to skim the questions first so you know what you are looking for.

3.2 Questions to Answer

Question. It is important to cite our sources. Figure out how to cite Graham's essay using BibTeX. Take a look at the citation for Mia Minnes article above and in the `citations.bib` file.

Answer. To access the citation for Graham's essay, click here: [2].

Question. Who do you think is the intended audience of this essay?

Answer. We believe that there are two intended audiences of this essay. The first is software engineers who design programming languages. Graham's central focus is how succinctness of a language is directly correlated to its power, and how a smaller source code is the purpose of high-level languages. Hence, by understanding topics relating to design choice, specifically focusing on how it affects language size, Graham assists software engineers in language design.

The second intended audience is developers who have experience using multiple languages. Many of the key ideas and examples Graham uses involve experience with and knowledge of programming languages. However, he does use several non-technical examples to illustrate his point, such as when he compares code readability per line to an installment payment plan.

Question. It is important to define our terms. What does Graham mean by 'succinctness'?

Answer. Graham defines succinctness of a programming language as the size of it's source code. Graham also takes into account that many different metrics can be used to measure the size of code, the most common of which being the number of lines. However, languages have different policies about how much code can be written in one line. He argues that a more effective metric for program size is the number of elements, defined as anything that would be a distinct node in a representation of the source code tree. Graham makes the case that this tree representation's size is "proportionate to the amount of work you have to do to write or read the program" [2], which is a more specific defi-

nition of succinctness as opposed to simply program size.

Question. What are some synonyms for 'succinctness'?

Answer. The synonyms for 'succinctness' we provide are based on Graham's definition of the term as defined above. Synonyms: compactness, conciseness, syntactical optimization.

Question. It is important to define our terms. What does Graham mean by 'power'?

Answer. He argues that "if smaller source code is the purpose of high-level languages, and the power of something is how well it achieves its purpose, then the measure of the power of a programming language is **how small it makes your programs**" [2]. This directly supports the key argument in his paper the succinctness = power. He also claims that existing tests that compare the power of programming languages use problems that aren't long or complex enough to produce meaningful results. Graham argues that a more appropriate test of a language's power is "**how well you can discover and solve new problems**, not how well you can use it to solve a problem someone else has already formulated" [2].

Question. What are some synonyms for 'power'?

Answer. The synonyms for 'power' we provide are based on Graham's definition of the term as defined above. Synonyms: ability, capacity, capability, potentiality, competency, applicability, efficacy, adequacy.

Question. In CS131, we spent some time thinking about abstraction. Do you think abstraction and succinctness are related? If so, how? If not, why not? Can you name any differences between the two concepts? Can you come up with any similarities?

Answer. Abstraction and succinctness are definitely related. As we defined in CS131, abstraction is the process of removing physical, spatial, or temporal details or attributes in programming languages to focus on details of greater importance. Hence, the process of making a program succinct is a form of abstraction as it attempts to hide any unnecessary implementation from the source code so that users can see a shorter version that only includes critical information. However, these two terms are not synonymous as an abstraction is more comprehensive, focusing on a coherent collection of ideas to represent the source code, whereas succinctness focuses on brevity. It is important to clarify that in his definition of 'succinctness', Graham mentions how a developer who has found repeated patterns that could be simplified has **discovered a new abstraction**" [2]. However, this abstraction is more syntactical in nature, rather than the syntactical + semantic use of 'abstraction' traditionally.

Question. How does Graham propose to measure succinctness?

Answer. Graham takes into account that many different metrics can be used to measure succinctness, the most common of which being the number of lines. However, languages have different policies about how much code can be written in one line. He argues that a more effective metric for program size is the number of elements, defined as anything that would be a distinct node in a representation of the source code tree. Graham makes the case that this tree representation's size is "proportionate to the amount of work you have to do to write or read the program" [2], which is a more specific definition of succinctness as opposed to simply program size.

Question. Come up with at least two other proposed methods to measure succinctness. Feel free to be as creative as you want here. You don't have to actually do / implement them, just propose some ideas and imagine how you would carry them out on pieces of code. It might help to think about how you might compare the succinctness of two pieces of code from the same or different languages.

Answer.

1. Another way to measure succinctness could be to take pieces of code from different languages that are programmed to perform the same task. Then, you measure the productivity of each of the different languages by taking the number of lines of code the program has divided by the time it takes for the program to compile. Whichever language has the best ratio of (lines of code)/ productivity can be deemed the most succinct.

2. Another metric to evaluate succinctness could be to measure the amount of bytes of machine code it takes for the language in question to implement a standard token. This token can be something like an arithmetic operation such as (+). For example, if we were to measure the succinctness of Python, we will take the amount of bytes it takes for a low-level language such as Assembly to implement (+). Then, we will code the same functionality in Python (without using external libraries) and count the number of bytes. By comparing the number of bytes it took to implement (+) between Python and Assembly, we can make an observation about Python's succinctness.

Question. What impact does succinctness have on readability of code?

Answer. As Graham argues, when discussing the readability of code, the readability of the whole program matters more than the readability of an individual line of code. Furthermore, Graham argues that succinctness is a factor (mathematically) in readability, so it is meaningless to argue how the **goal of**

a language is readability, not succinctness [2]. To a certain extent, we agree with Graham's argument. Making code succinct, involves making source code smaller so that you are extracting only the important information from the source code. However, as Graham briefly touches on, we believe that there are situations where a language is 'too succinct'. If the language was excessively compact where syntactical information is abstracted to a high-level, this would heavily limit the readability of that code.

Question. What impact does succinctness have on writability of code?

Answer. Similar to our aforementioned discussion of the impact of succinctness on code readability, succinctness has an impact on the writability of code. Programs such as Python are incredibly easy to write as users can reason about the code similar to how they would reason about writing an essay using a 'natural language'. However, there is a trade-off between the writeability of the code vs. the succinctness of the language. If we make a language increasingly succinct, users will be unable to reason critically about their code as the syntactical rules will be overly abstract to understand.

Question. What impact does succinctness have on maintainability of code?

Answer. Succinctness has a impact on maintainability of code as languages that are easier to understand will allow developers to more easily identify bugs and propose solutions to fix them. We believe that an appropriately succinct language (one that is not overly abstract or contains too much unnecessary information) results in a lower learning curve, allowing developers to maintain code through correcting errors and improving performance.

Question. What impact does succinctness have on accessibility of programming?

Answer. Succinctness has a large impact on accessibility of programming. By designing languages to have a less complicated structure, software engineers are effectively lowering the learning curve for their users. With a more compact structure, it would also be easier for software engineers to build in accessibility features so that people with disabilities will experience less complications while programming. For example, Myna is a vocal user interface that allows users to program purely based by voice. Because it's syntactical structure is similar to natural languages, its users find it intuitive to program in.

Question. What impact does succinctness have on learnability of programming?

Answer. Learnability in programming languages is the practice of designing languages that can be easily read/worked with such that there are no significant learning barriers for the user. As established above, since we argued that

succinctness has positive impacts on readablitiy/ writeability/ maintainability/ accessibility, it is evident that succinctness helps with learnability as well. By removing the syntactical complexities/ unnecessary information from the source code, succinct languages allow its users to learn the language more efficiently.

Question. What impact does succinctness have on teachability of programming?

Answer. As we established above, since succinctness allows users to better extract solely the critical information from the program and reason about it, it helps to minimize obstacles in the learning process. Hence, as evident in languages such as Python, the language will have a larger community base that allows users to interact with each other and offer further optimizations to the language. Hence, we can argue that succinctness eases the learning curve of languages, which in turn has a positive effect on the language's teachability as it nurtures an interactive community of skillfull developers to teach each other critical skills.

Question. What is your personal experience with succinctness in programming languages? This could come from CS courses, personal projects, internships, reading or writing about programming, grutoring, or anywhere else in your experience.

Answer. As CS majors at HMC, we can speak on various experiences with succinctness in programming languages. For example, in *CS105 Computer Systems*, we both struggled with understanding/ analyzing the low-level symbolic code in assembly language. Since assembly language usually has one statement per machine instruction, it's structure is concrete and complex. In contrast to this, high-level languages such as Python were much easier to grasp, allowing us to write code that was less error prone and maintain. Furthermore, using Python, we have been able to understand and implement complex processes such as machine learning models which we would have been impractical to understand in low-level languages.

Question. At the end of Graham's article, he refers to some other sources and languages. (Some of the links are broken, so you may need to do a little searching of your own for some of them depending on what catches your interest). Choose one of these resources and in a few sentences, summarize it's relevance to Graham's argument. Make sure to figure out how to cite it!

Answer. In the paper titled "Lisp as an Alternative to Java" from Intelligence's Volume 11 Issue 4 [1], author Erann Gat talks about the advantages of the language Lisp. According to Gat, Lisp offers the same advantages that Java does including automatic memory management, dynamic object-oriented programming, and portability. Furthermore, it is superior to Java in runtime, programming effort, and variability of results. This paper is relevant to Gra-

ham's argument as the study found that Lisp programs were significantly lower in development time compared to that of Java although the users in the Lisp control group were less experienced programmers. Furthermore, Gat found that the runtime performance of Lisp was significantly better than Java. Hence, applying Graham's argument, it is evident that Lisp is has comparative advantages over Java as the succinct language framework allows to optimize performance and runtime.

Question. We covered many technical topics in CS131. Think of one topic we covered and relate it to the notion of succinctness.

Answer. An example of a technical topic in CS131 that relates to the notion of succinctness is pattern matching in Haskell. Because Haskell is a functional language, we were able to employ pattern matching techniques to make our programs more succinct. Pattern matching, defined as the process of checking whether a specific sequence of characters/ tokens/ data exists among the given data, allowed us to check an argument against different forms. Hence, through pattern matching, we were able to greatly simplify our code without losing critical information or slowing runtime processes by identifying specific types of repeated expressions.

Question. We covered several cultural / societal / broader impact topics in CS131 (during HW 1 and module 9 for instance) . Think of one cultural / societal / broader impact topic we covered and relate it to the notion of succinctness.

Answer. Since we argued that succinctness has positive effects on readability/ writability/ maintability/ accessibility/ learnability/ teachability, it follows that succinctness has a net positive impact on socio-politico-cultura structures. Especially because succinct languages have a lower learning curve, it has less barriers for people with learning disabilities. Furthermore, given that it has a lesser learning curve, it will allow for the development of a community support system where users can interact and assist each other in discussing critical issues. This contrasts with languages that are considered 'elitist' which have a high learning curve and restricts users without resources from learning it.

Question. What is a potential benefit of succinctness in programming languages?

Answer. As discussed above, there are many benefits of succinctness in programming languages including positive effects on readability/ writability/ maintability/ accessibility/ learnability/ teachability of code. To avoid repetition in our responses, please refer to the above answers for specific benefits of succinctness.

Question. What is a potential harmful effect of succinctness in programming languages?

Answer. As briefly discussed in the question asking about the impact of succinctness on the writeability of code, if we place increasing emphasis on making languages succinct, there could be situations where we are over-simplifying critical complexities. Thus, the more complexity we try to hide, the bigger the chances of it to leak details that are significant. Another potential harmful effect is the notion that a succinct language is always preferable. This is where we disagree with Graham as in the sub-section titled Readbility, Graham explains that succinct code is always preferable despite increasing the readability-per-line. However, while working with languages such as PHP, we found that succinctness is not always attractive in a programming language. Although PHP is regarded as one of the easiest programming languages to master, it's horrible design with inconsistent rules of coding, abbreviations, and algorithms is the reason for it's unpopularity. PHP is a good example of how a language that focuses solely on succinctness and ease-to-learn may not be the most optimal language. We believe that in addition to succinctness, 'powerful' languages have robust and consistent frameworks that allow them to be effective in various applications.

Question. Are there any other comments, ideas, reflections, or observations you would like to make about succinctness?

Answer. Although there are some potential harmful effects of succinctness, in general, I agree with Graham that succinctness in languages are beneficial. This is because when we start off programming, its easy to write complicated functions that have interesting outputs. However, the problem is that it's impossible to change at that point because it becomes impossible to understand all of the processes. By designing languages that are succinct, its users are able to avoid unnecessary complexities and understand the central components of the programs.

4 APL: A Programming Language

One of the most terse and succinct programming languages out there is APL. (It has a more modern descendent called J in case you are curious to check that out).

For this section, you should

- Read the article "The APL Programming Language Source Code" by Leonard J. Shustek, available from the Computer History Museum at

  computerhistory.org/blog/the-apl-programming-language-source-code/

- Explore the tab labelled "Hi" on tryapl.org. You can click the examples to automatically enter them into the interpreter, or you can pull up the APL keyboard using the button on the top right. See what each of the examples does.

- On tryapl.org, click the "Learn" tab, then under "APL Basics" click through the three basic tutorials

  1. APL Expressions
  2. Arrays
  3. Functions

  You can just keep clicking the "Next" button, which will automatically show you some explanation of what is happening and run some APL code in the interpreter for you. As you are clicking through, take note of anything that seems peculiar, familiar, or interesting.

- Read this blog post from CodeBurst on solving the Fizz Buzz problem in APL:

  codeburst.io/fizzbuzz-in-apl-a193d1954b4b

  Try to understand each step, and feel free to try it out in the APL interpreter on tryapl.org. You can go directly to the final solution in the interpreter by following this link.

4.1 Questions to Answer

Question. What does APL stand for?

Answer. APL (named after the book A Programming Language) is an array-oriented programming language

Question. Who invented APL? Anything notable about this person?

Answer. APL was developed in the 1960s by Kenneth E. Iverson. He was a Canadian computer scientist who was honored with the Turing Award in 1979

for the development of APL.

Question. When was APL invented?

Answer. APL was first invented by Iverson in 1957 as a mathematical notation, not as a computer programming language. APL then was transformed into a programming language in November 27, 1966.

Question. Who do you think was the typical user of APL?

Answer. Since APL was initially designed as a mathematical notation, it uses symbols that are closer to standard mathematics than programming. Hence, one typical user of APL could have been mathematicians/ researchers/ engineers who wanted to learn programming using familiar concepts.

Another typical user could have been developers who wanted a succinct yet efficient language to master. Since APL was both very readable and compact, it had a smaller learning curve. Furthermore, APL applications emphasized interactivity, which provided a huge productivity increase compared to the batch-job processing more typical at the time. Hence, even for a typical developer, APL was an attractive language.

Question. What was the original purpose of the APL language? That is, why was it invented in the first place?

Answer. APL was initially developed as a mathematical notation. Although other matrix-oriented symbol systems existed, including the concise tensor notation invented by Einstein, it was tailored more towards mathematical analysis and less towards synthesis of algorithms. Specifically, APL was invented as a consistent language for describing operations on arrays, a notation a machine could interpret.

Question. How was APL received? Back up your answer with evidence from the article.

Answer. As the article mentions, **APL became popular when IBM introduced "APL\360" for their System/360 mainframe computer** [4]. Hence, APL was only popularized after APL became a programming language and was adopted into IBM's mainframe computer. After that, APL thrived in the 70's and 80's as an ultra-high productivity language for solving a problem rapidly and accurately.

Question. Why do you think APL didn't stick around?

Answer. I believe that APL didn't stick around because better languages started emerging. For example, languages such as R are probably better at performing mathematical computation and software such as Excel provided spread-

sheets that far out-perform APL arrays. Furthermore, APL relied on special symbols to represent its built-in functions, but these symbols are no longer available on the standard keyboard.

Question. What are some of the claimed benefits of APL?

Answer. Some of the claimed benefits of APL include 1. APL is an interactive language returning answers immediately, 2. APL is easy to learn and use, 3. APL code takes less time to write and debug than other high-level languages.

Question. According to any of the sources you looked at, what is the "power" that APL gives the user? Back up your answer with evidence from one of the sources.

Answer. As the article describes, using APL, the programmer can **sit and an electromechanical typewriter linked to a timeshared computer, could type APL statements and get an immediate response. Programs could be defined, debugged, run, and saved on a computer that was simultaneously being used by dozens of other people** [4]. Another power granted to the user is that **APL\360 was a conversational language that provided fast response and efficient execution for as many as 50 simultaneous users** [4]. Finally, applications that employed APL emphasized interactivity, which provided a huge productivity boost compared to the batch-job processing that was typical at the time.

Question. What do you think Paul Graham would think of APL? Why?

Answer. Graham would praise APL as it is probably one of the most succinct languages ever created. Since Graham emphasizes the benefits of shorter programs including quicker development time and less code to maintain and debug, Graham would praise APL's productivity and brevity.

Question. What do you think Ada Lovelace would have thought of APL? Why?

Answer. Lovelace, who was a mathematicians would have praised APL as well. Lovelace is recognized as the first programmer as along with Charles Babbage, they completed the Analytical Engine. Although Babbage was the one who invented the Analytical Engine, it's capabilities were limited to mathematical calculations. It was Lovelace who described how codes could be created for the Engine to handle letters and symbols along with numbers and theorized a method for the Engine to repeat a series of instructions. Hence, the evolution of the Engine shares an obvious similarity with APL as APL was also initially created as a mathematical system and was later transformed into a programming language through the use of symbols and abstractions. Hence, Lovelace would love APL as she could relate to the experiences of Iverson and would be amazed with the capacity of APL as it far surpasses the sophistication of the

Analytical Engine.

Question. What do you think Grace Hopper would have thought of APL? Why?

Answer. As we know, Hopper was involved in the creation of UNIVAC, the first all-electronic digital computer. She invented the first computer compiler, a program that translates written instructions into codes that computers read directly. Since Hopper's work precedes Iverson's, she would be proud about how her work contributed to the development of APL. Hopper, who developed the first compiler A-0, which translated mathematical code into machine-readable code, is sure to recognize the mathematical genius of Iverson. Hopper, who's central motivation was to construct high-level languages so that she can assist in expanding the community of computer users, would have recognized APL as an amazing language as its brevity and efficiency made major impacts on programming accessibility.

Question. What do you think Alonzo Church would have thought of APL? Why?

Answer. From CS131, we know that Church's main contribution was lambda calculus within which it was possible to define a class of functions that coincide, arguably, with the intuitively computable functions. It is undeniable that APL influenced functional programming - programming with mathematical functions. Since the foundation of APL is mathematical notation, Church would have respected the genius of Iverson in establishing a language where mathematics forms the cornerstone of the syntactical and semantic design.

Question. Are the any similarities between APL and Haskell? Do you think knowing Haskell first influences the way you might experience APL?
Answer. This relates to our answer to the previous question. Haskell was developed after APL, and was definitely influenced by APL's design. The core idea behind both of these languages is that they are both examples of an algebra of programming. Using these algebraic tools, programmers can derive or calculate a correct program to solve a problem by combining standard higher-order functions according to mathematical rules. Knowing Haskell first influences the way we experience APL as APL is an even more abstract language than Haskell.

Question. Any similarities between APL and any other programming, natural, or fictional language that you can think of?

Answer. According to online resources, APL led to the rise of K. K is a proprietary array processing programming language that serves as the foundation for kdb+, an in-memory, column-based database. Since K has elements of Scheme, a functional programming language that shares many similarities with Lisp, the fact that APL led to the rise of K corroborates our earlier arguments describing

how APL influenced functional programming.

Question. What do you think of APL in terms of readability, writability, maintainability, teachability, learnability?

Answer. APL is at a very high-level of abstraction, making it well-suited for concise formulations of algorithms. Hence, APL enjoys the positive impacts of succinctness as described in our responses to Graham's including readability, writability, maintainability, teachability, and learnability.

Question. What do you personally think of APL now that you know a bit more about it?

Answer. We believe that APL is an incredible language as it remains one of the most powerful, consistent, and concise languages ever devised. The core features of APL including its interactive design, its ease-to-learn, optimal readability/writeability make it well-suited as a communication mode between developers and computers.

Question. What do you make of Paul Graham's claim that "succinctness is power" with respect to APL?

Answer. As we argued above, we believe that APL is one of the best evidences to support Graham's claim as APL's succinctness provides many advantages. APL's user-oriented notation which consists of a set of symbols with a very simple set of syntactical rules for putting them together make it easy for developers to put components together to understand the processing of data.

Question. The previous section asked you to think of ways to measure succinctness. How do you think that would that work out with APL?

Answer. Our first evaluation method to measure succinctness is better suited to measure succinctness for APL. Using this evaluation metric, APL can easily shown to be succinct than other programming languages as the APL symbol set is the equivalent of many words in describing algorithms and procedures. Hence, since often one or a few APL symbols can have the same result as several lines of code in another programming language (with no obvious decrease in efficiency), by using the (lines of code)/productivity as a metric would show that APL is a succinct language.

Question. What other observations do you have about APL?

Answer. If we had spare time, we would love to learn APL as it would allow us to develop shorter programs that enable us to think more about the problem we're trying to solve than how to express it to a computer.

5 Java Enterprise FizzBuzz

5.1 What to do

1. Read this blog post about Java Enterprise FizzBuzz.

   https://brian.copeland.bz/2020/02/fizz-buzz-enterprise-edition/

2. Explore the repository of the Java FizzBuzz Enterprise edition for at least 10 minutes. (E.g. pretend you are going to be an intern at SeriousCompany and you need to get familiar with this project source code.)

   https://github.com/EnterpriseQualityCoding/FizzBuzzEnterpriseEdition

5.2 Questions to Answer

Question. Who wrote this code?

Answer. The repository has 31 contributors who are java programmers who are frustrated with the coding practices and standards common in large enterprise engineering teams. They were likely aggravated with how verbose the java language is and how their productivity was measured by how many lines of code they produce rather than the quality of that code.

Question. Who is the intended audience of this code?

Answer. Java FizzBuzz is intended to be a parody of verbose programming languages and excessively tedious standards of enterprise code for fellow frustrated programmers. It could also be viewed as a critique to managers who enforce these standards in industry.

Question. Locate the file `Constants.java`. Explain this file. What do you make of it?

Answer. The file is located here. The file `Constants.java` includes a public class Constants that has constants declared within it. Constants are variables whose value cannot change once it has been assigned. Since Java doesn't have built-ion support for constants, the file uses the keyword "final" in front of the variable declaration to signify that it is a constant. These constants help to make the program more easily read and understood by others as certain keywords such as Fizz and Buzz are assigned to be the strings "Fizz" and "Buzz" respectively.

Question. Locate the file `NewLineStringReturner.java`. As far as you can understand, what is this piece of code doing? What do you make of it?

Answer. The file is located here. The `NewLineStringReturner.java` class simply returns a line break. The class is then used by the `NewLinePrinter.java`

class which implements the line break output mechanism. The string that signifies a line break is declared in the `Constants.java` file as LINE_SEPARATOR.

Question. What do you think the existence of this repository says about the practice of programming and the Java programming language?

Answer. The existence and many online discussion forums about this repository suggests that many programmers are frustrated with the current paradigms in software development that focuses on large scale, maintainable systems over clear, concise code.

Question. What is your own personal opinion of Java? What personal experiences have influenced they way you think about Java compared to other languages?

Answer. I only used Java during CS60, but it's my least favorite of the most popular programming languages (i.e. Python, JavaScript, C++, C). Especially after learning Python, Java's verbose syntax was tedious. However, after learning Java, I actually liked C++ - because the language was so much more efficient, it was more rewarding to learn of all of the weirder features.

Question. In an earlier section, you were asked to come up with a way to measure succinctness. How would that apply to this repository?

Answer. This repository parodying enterprise Java code standards by creating a repository that is essentially the opposite of succinct.

Question. Compare and contrast the FizzBuzz APL implementation with this Java FizzBuzz implementation.

Answer. The Java FizzBuzz APL implementation is incredibly robust. It is clear that the Fizz Buzz Enterprise Edition (FBEE) spreads the logic required to implement the FizzBuzz coding task across dozens of files. This goes against Graham's suggestion of making code succinct as FBEE is not succinct at all. However, FBEE offers much more functions than traditional fizz buzz such as printing numbers other than 3 or 5, altering the notion of "division", rigorous testing, and features a Contributor Code of Conduct. Compared to this, APL's implementation of FizzBuzz is incredibly succinct and only has the capability to perform the necessary computation.

6 Power and Succinctness in Other PLs

We just looked at two extremes: a super terse language and a verbose satire. Let's think about more typical examples now.
Question. Choose any 5 programming languages and rank them by your perceived level of succinctness. (You can include APL and Java if you like)

Answer. Ranking the languages from least to most succinct.

1. C#.
2. Java.
3. Python.
4. Haskell.
5. APL.

Question. For those same 5 programming languages, rank them by your perceived level of power.

Answer. Ranking the languages from least to most power.

1. C#.
2. Haskell.
3. APL.
4. Java.
5. Python.

Question. What criteria are you using to determine the level of succinctness?

Answer. To determine the level of succinctness, we are using our personal experiences in using the language as a gauge. To be more specific, we are comparing the syntactical rules declared for each language and are remembering if we felt as though some languages required unnecessary symbols and terms for implementation. For example, languages such as C# and Java require unnecessary syntactical terms whereas APL can express all of the information using a few symbols.

Question. What criteria are you using to determine the level of power in a PL?

Answer. It was difficult to come up with a good evaluation metric for determining the level of power in a PL. Since the definition of 'power' can be

depending on what we hope to optimize, depending on our definition, our rankings could be entirely different. We decided to define 'power' as the ease to learn the language and the number of functionalities it has to offer. Using this metric, Python and Java are the most 'powerful' as they have an easy learning curve and great libraries. Haskell is an incredibly powerful language as it is a statically typed functional language. C# is incredibly difficult to learn and has long development times which is why we ranked it at the bottom.

7 Connecting with Prior CS131 Material

7.1 Material from Critiquing PLs

On Sakai, there are several places where we provided material or asked you to interact with material related to broader impacts of PLs. These inclduded

1. HW1 from Module 1
    - C is Manly Python is for n00bs
    - Grace Hopper's Keynote Talk
    - Natural Languages and Their Character Sets / Directionality
    - PLs and Their Character Sets / Keywords / Directionality
2. From Module 9
    - Accessibility
    - Community and Identity
    - Design
    - Abstraction
    - Symbolic Reasoning
    - Language
    - Ko's 10 forms of PLs
    - Blackwell's Metaphor's in Java
    - Dijkstra's thoughts on Teaching CS

Question. Having gone through Graham's "Succinctness is Power", learned a little about APL, and explored a Java satire, choose any one (or more!) of the above components to connect to the ideas of succinctness and power and programming languages. Write at least one paragraph making these connections.

Answer. Applying Graham's "Succinctness is Power" to APL, we were able to dissect the advantages and disadvantages of APL's design. Examining APL, it's brevity and efficiency corroborated Graham's claim that given that most programmers have similar line-of-code-per-hour productivity rate, so languages that require fewer lines of code are more effective. Also, by learning about the mathematical foundation which APL was built on, we were able to understand how it relates to the works of Lovelace, Hopper, Church. Hence, this idea of how succinctness is power directly ties to good practices in language design such that the language becomes more accessible to a wider range of users. We explored this idea further by examining the Java FizzBuzz implementation as we got to see an example of how making a language intentionally robust and complex is not helpful, further corroborating Graham's claims. Hence, through exploring these distinct topics, the underlying theme was comparing the efficiency/ brevity of different languages and how that affects their readability/ writeability/ maintability/ accessibility/ learnability/ teachability.

7.2 Technical Material from CS131

In CS 131, we covered a lot of technical PLs principles:

1. Functional Programming
2. Functional Data Structures
3. Interpreting and Evaluation
4. Parsing
5. Functional Abstractions
6. Type Systems
7. Lambda Calculus

Question. Having gone through Graham's "Succinctness is Power", learned a little about APL, and explored a Java satire, choose any one (or more!) of the above technical PLs topics to connect to the ideas of succinctness and power and programming languages. Write at least one paragraph making these connections.

Answer. Having gone through Graham's "Succinctness is Power", we were able to take his claims and compare them with ideas of functional abstractions we learned in CS131. In functional abstraction, the details of the algorithms to accomplish the function are not visible to the consumer of the function. This is evident in languages such as APL that uses symbols to offer high-level abstractions of central pieces needed to perform essential tasks. This also is directly tied to the arguments that Graham provides in his claim that succinctness is power as Graham's notion of succinctness involves extracting only the significant components of low-level languages and designing higher-level languages so that users can write code that is less error prone and maintain. Through APL, we also saw how the idea of functional abstractions was formed where the language relied on defining simple objects and a few axioms that relate them to see what sort of results the axioms yield. Finally, by exploring Java satire through the Fizz Buzz Enterprise Edition (FBEE), we grew to appreciate languages such as Haskell more. Haskell is able to represent these abstracts through a typeclass along with a set of laws that well-behaved instances must obey. These functional abstractions in Haskell lead to code that is more readable, significantly improving development time. Hence, by answering these initial questions, we were able to connect the information we learned through these articles to the topics we covered in CS131 which we thought was super cool.

References

[1] Erann Gat. Point of view: Lisp as an alternative to java. *Intelligence*, 11(4):21–24, dec 2000.

[2] Paul Graham. Succinctness is power, May 2002.

[3] Mia Minnes, Bruce A. Maxwell, Stephanie R. Taylor, and Phillip Barry. Writing in CS: why and how? In Tiffany Barnes, Daniel D. Garcia, Elizabeth K. Hawthorne, and Manuel A. Pérez-Quiñones, editors, *Proceedings of the 49th ACM Technical Symposium on Computer Science Education, SIGCSE 2018, Baltimore, MD, USA, February 21-24, 2018*, pages 402–403. ACM, 2018.

[4] Leonard J. Shustek. The apl programming language source code, Jun 2021.