Syntax as a Tool of Thought

Abstract

Programming languages allow computational problems to be expressed in a way humans can understand. Each language strives to complete a specific goal, and no language can be perfectly replicated by any other due to the inherent differences that exist from their distinct domains. Performance is an important metric to guide programmers in choosing a language for their project, but syntax is more important in communicating solutions to problems. Performance-focused languages like C and Fortran do an excellent job at expressing low-level details about the hardware or precise algorithmic elements, while high-level languages like Python are much better at explaining solutions that can be spoken aloud in plain English. There are elements within languages that enhance their ability to solve problems, including symmetrical aspects of the language that encourage high amounts of reuse over different areas and language-specific elements that are unique to its own style. Plaintext source code is only a single way of transcribing a programming problem, but statement graphs are another method popular to functional programming languages that introduce new ways of approaching and visualizing problems. JGPL is a new programming language built from the idea that exploits the abstraction of syntax in communication and uses the underlying concepts of statement graphs to allow programmers to extend upon assembly code to create fluid syntactical representations of unique domain-specific code.

Introduction

This paper is split into two parts: the first takes a look at programming languages in a general sense, analyzing the capabilities of specific languages in case studies and viewing the concepts that structure source code, while the second displays a new programming language built to address a problem introduced in this paper.

Programming languages are tools for expressing thoughts to a machine as well as to other programmers, but there are different methods that enhance the productivity of a language. The complexity of a language requires it maintain a defined scope as any large change could deprecate user programs, but language designers can employ different methods to minimize the cost of changes. To the end users, programs are diverse and uniquely adhere around project requirements, so the decision on which language to choose to best solve a problem comes down to many different factors. Performance can be important, but as a qualifier that is at least somewhat applicable to most projects, there are better metrics that can decide which languages to choose from. In fact, requiring performance from a language

could also signal the presence of other submetrics that point to more applicable qualities like preciseness in design, security, and reliability, each stemming from the same source that performance does. Furthermore, performance does not always indecisively point to languages like C; it is important to realize the context these languages were developed in and the environment and community they were meant to support. Python and C are often at odds with each other due to how differently they express the same ideas, but the context reveals that not only were they developed for completely different purposes and different audiences, but they also are incapable of representing the same programs effectively even when given line-for-line translations.

Abstraction and symmetry are two qualities which improve the usability of programming languages. Abstraction describes the reduction of complex concepts down to simple keywords in an attempt to divert the focus of the program from concrete but repetitive ideas to reusable building blocks and overlapping subparts. Symmetry in languages places little enforcement on rigid design patterns, encouraging creative experimentation with language syntax to create programs that work in ways the language designers may not have intended to the benefit of solving problems in new kinds of ways. While “power” is an adjective used to describe languages which employ these creative concepts, it is not the most useful quantifier due to all Turing-complete languages having the same level of computability; a much better relative term to describe programs is “elegancy”, which is explored with case studies in multiple languages and programs that exhibit it.

However, a lot can be learned from a language given just the simplest “Hello, World!” program it can create. Java, Python3, Python2, C, PostgreSQL, and ARM Assembly are compared and contrasted with only the syntax they provide, which reveals the design concepts the languages prioritize and the extent to which the languages enforce ideas on the user. In terms of design, the simple programs can build up to more complex variants that are describable with “statement graphs”, which are graphical depictions of the execution of the program. Functional programming languages blur the lines between what a “function” and “procedure” is, and utilize symmetrical design with first-class citizenship to create a unique perspective to programming. Coroutines can be created through statement graphs with another shift in perspective, yielding ideas like for-loops being anonymous coroutines and complex control-flow being the result of reducible expressions.

Finally, the new programming language JGPL is a culmination of the ideas expressed in the paper. It focuses on syntax as a tool of thought, with the formation of tokens into expressions being variable in a functional perspective. Basic ideas like assignment and iteration are not builtin concepts, but are entirely creatable and extendable by the user, allowing them to create their own means of execution with low-level and high-level design available. The biggest benefit JGPL presents is its fluidity; by letting the user import new syntax styles with a single statement, they can have different source files within their

program reflect the notation a problem set requires instead of adapting everything under a generalized umbrella. JGPL is better than other programming languages when the problem being solved requires a unique notation: while other programming languages necessitate a consistent syntax across all source files, JGPL allows users to import different syntactic styles with a single statement, allowing the representation of a much more diverse set of problems than normal programming languages would allow. Domain-specific programming languages are just as important as general-purpose languages, as they allow the representation of unique problems with distinct syntax. In this regard, JGPL can be visualized as a tool for generating domain-specific languages, as the syntax of different JGPL programs is completely user-defined with language structures being abstracted from standard intermediate code. Finally, the communicability of programming problems is important for all programmers to study, and a focus on improving the legibility, usefulness, and elegancy of code should be a priority among software teams.

Language Updates

Programming languages have the tendency to get complex very quickly. Even simple compilers can take months to develop, and introductory students may need to study a language for years before they can create usable programs with it. Depending on the complexity of the language's grammar, changes made to the language may be very difficult; a language which strives to add a certain feature may hit a roadblock when it comes to implementing changes without conflicting with user code. In this case, there are two ways a language can then avoid conflicts in design: by creating an entirely new dialect to house the changed language, or by shoehorning the feature into the language through complicated syntax. Python3 is an example of a language which created a new dialect in order to introduce a type-hinting system, support for asynchronous programming, several new keywords, and semantic changes for existing syntactical constructs that Python2 did not have [1]. Large-scale modifications to the language like this would have made backwards-compatibility difficult to achieve, and it can be tough for language developers to justify changes without also requiring an update in the language's major version to signal to end-users that a quick version update to their project is not possible. For example, Figure 1 depicts a valid Python3 code segment that requests input from the user in the form of their name and age, formats it as a string, and outputs it to the console.

```
1  name = input('Please input your name: ')
2  age = input('Please input your age: ')
3  format = name + ' ' + age
4  print(format)
```

Figure 1: Python program with user-defined variable "format"

This program executes as intended in Python3, but ambiguity would be introduced if a new reserved keyword was to be added. Consider a situation where future Python developers want to turn `format` into a reserved keyword like `break` or `None`. This would violate the backwards compatibility of the language, as the code could not be compiled due to the creation of a variable with the same name as a reserved keyword on line 3. In Python3's case, the language designers decided that it would implement the changes it desired without considering the programs made in Python2, and it would be up to the developers of outdated programs to update their own programs without relying on the Python interpreter to do it for them.

Alternatively, language designers have another option they may perform when it comes to implementing a new idea. In the previous scenario, `format` may have played a pivotal role in the language design, but the designers would have to consider variables named the same as their new keyword. If they wanted to skirt around the naming conflicts while still implementing their new feature, they could instead shoehorn their feature into a situation that would not conflict with the language's syntax. An example of this is C++'s lambda expressions introduced in C++11. The vast majority of C++ is backwards compatible with much older versions (with certain exceptions of keywords and syntax which become implemented and deprecated over time), but the upsides of low user version modification comes with its own downsides in the realm of complex syntax. The code example given in Figure 2 depicts a program that uses lambda expressions to sort a list of Human objects in order of increasing age, with the lambda expression shown on line 14.

```
1  typedef struct {
2      std::string name;
3      int age;
4  } Human;
5
6  // Creates a list of humans with random names and ages. The number of humans
7  // created is equal to <capacity>.
8  Human* create_arr(int capacity) {
9      // ...
10 }
11
12 int main(void) {
13     Human *humans = create_list(10); // Creates a list of 10 random humans.
14     sort(humans, humans + 10, [](const Human& a, const Human& b) -> bool {
15         return a.age < b.age;
16     });
17 }
```

Figure 2: C++ lambda expressions

C++ language designers had a goal to implement lambda expressions because of their concision and usefulness in completing short inline tasks, but the difficulty was how to fit in

these expressions into the syntax [2]. For this example, C++ lambda expressions are depicted in the form `[](type varname) -> type { code; }`. The brackets beginning the expression were an excellent choice to hint to the compiler that the tokens following the brackets depicted a lambda expression; in the language definition, brackets could only normally be seen following a variable identifier in order to denote that the identifier was an array of values, so brackets on their own would be an impossibility resulting in a compiler error before C++11. Therefore, there would be no issue of backwards-compatibility conflicts in the same way that Python had, since code could not have broken as a result of the update. This worked out fine in C++'s case, as the syntax change performed is relatively readable, but the more changes that are made to the language, the more difficult it becomes to find neat areas to shoehorn clean updates.

To these ends, it becomes imperative for language designers to anticipate changes before they occur by reserving keywords that they intend to be used later, introducing rules that will be performed in the case of conflicts, and creating shortcuts they can exploit in the future should the need arise. Although, another consequence of this ideology is the requirement for a flexible compiler: a compiler built to interpret a program of one type should be easy to update when the programming landscape changes. The MATLAB compiler is an example of one which can be considered limiting in its design decisions in this regard due to it containing a large amount of context-specific syntactical objects and frequent changes in syntactic structure across programs, making it difficult at times to create a flexible compiler that always consistently interprets programs correctly [3]. Because of this, MATLAB does not guarantee backwards compatibility due to the constraints it would put on their design. C++'s solution to lambda expressions was concise and elegant relative to its existing grammar because the placement of brackets does not carry with it any significant message, but the same cannot be said for MATLAB which is catered towards a different audience who may not gel with similar design decisions, thus warranting the need for recreation of portions of the grammar over the shoehorning of features inside it.

Performance

When tasked with solving a computing problem, there are many qualifiers which must be considered in choosing the best language to implement the solution. Problems may be visual in nature, requiring graphics to convey messages to the user with inputs being received, or they may be highly computational, focusing more on the completion of an operation above the interactivity of the user. The end users of the program can influence the decision on the type of program to create; if the end users are other programmers, then the only interaction involved may be a command-line interface or piped messages through a kernel, while technically challenged users are much more likely to prefer clickable assets,

large text, and concise on-screen instructions informing them of ways they can utilize the software. These aspects will influence the decision of the programming language to choose. An application which contains only buttons and text boxes as user input would be much better suited for Java than it would be for C, and an application designed for use on a website should consider being implemented in Javascript, HTML, and CSS due to the vast amount of support they have among service providers.

To these ends, performance is a large metric that influences the decision of a programming language. When comparing two identical programs that each contain the same features with one written in a compiled language versus another in an interpreted language (i.e., one written in C versus another in Python), compiled languages tend to outperform interpreted ones on most tasks. For example, Figure 3 depicts a segment of a program meant to categorize the contents of an image with the help of three different convolutional neural network architectures. The program was created first in Python and then translated line-for-line into an equivalent C program without changing the underlying algorithm.

```python
# Language: Python

for i in range(len(inputsnp.shape[0])):
    for j in range(len(weightsnp.shape[1])):
        outputnp[i][j] = biasnp[j]
        for k in range(len(weightsnp)):
            outputnp[i][j] += inputsnp[i][k] * weightsnp[k][j]
```

```c
// Language: C

for (int i = 0; i < inputsnp->shape[0]; i++) {
    for (int j = 0; j < weightsnp->shape[1]; j++) {
        outputnp[i][j] = biasnp[j];
        for (int k = 0; k < weightsnp->shape[0]; k++)
            outputnp[i][j] += inputsnp[i][k] * weightsnp[k][j];
    }
}
```

Figure 3: Line-by-line translation of a Python matrix-multiplication algorithm into C

The purpose of this translation was to compare the performance of the two languages on an equivalent program. The program itself was an implementation of three popular image-recognition convolutional neural network architectures, including VGG-16, Alexnet, and LeNet. The performance of the two programs on each architecture is shown in Figure 4.

	VGG-16 (s)	Alexnet (s)	LeNet (s)
Python	12633.41	1371.79	0.56

C	1328.21	150.89	0.35

Figure 4: Performance of a translated program written in Python and C on three different neural-network architectures

This experiment shows that a program written in C vastly outperforms an identical program written in Python. There are dozens of reasons that explain this: Python is interpreted, meaning machine-translation happens as the program is run compared to C which is compiled ahead of time; Python is dynamically typed, requiring the machine to do a lot of referencing to determine the definition of a keyword [4]; and the instruction set the Python virtual-machine operates under has a very high level of abstraction, meaning a single bytecode instruction may translate to multiple hardware machine-code instructions [5]. These observations also apply to other languages besides C and Python, with compiled programs like Go and Haskell still performing better than Ruby or even Java [6].

Language Usage

The performance comparison introduces an important consideration about the decision of a language. Speed will always be a concern to projects when choosing the best language for their project, so projects may feel inclined to implement their solutions in C just for the performance benefits, but this is not what is reflected in the real world: based on number of appearances in GitHub projects, the languages Python, Ruby, and Javascript appear to be more popular than C [7]. Regardless of the obvious speed benefits C offers, it seems that programmers tend to gravitate towards the concise, abstract, and easy-to-implement languages instead.

The paper at [8] identified hundreds of unique features that characterize programming languages, each of which are distinguishable under 5 umbrella attributes including usability (learnability, operability, user error protection), cost (implementation cost, licensing cost), product (ownership), supplier (language support, platform), and maintainability (modularity, reusability, testability). Across this spread of attributes, speed appears to play a relatively minor role in the decision of a language; after all, languages which run at light speed but have no outside support would require time-consuming maintenance to be done by the language users, while languages that are slow but easy to write in would result in completed programs made in a fraction of the time. In fact, a similarity that both Python and Ruby share is their readability and concision; a paper [9] found that programs with source code that ranks high in maintainability (a metric generated from Software Improvement Group's quality model) are correlated with a high issue-resolution speed, meaning that problems submitted from end-users are quickly resolved when the design of the program encourages readability. Furthermore, learnability and ease of use are both factors that are significant for a team to consider. Programming

languages that are excessively complex encourage learners to either delay their inclusion in a project until after they master the language or start programming at an intermediate level and risk incorporating poor code into their program's design.

In some ways, programming languages can be compared to spoken languages. Individuals tend to use the same language they were born with, with multilingual people shifting their choice in language depending on geography or populations they interact with. Programmers act in a similar way, with programmers choosing to either start new projects based on what they already know or choosing a language with existing libraries well-suited for the task. While Python may have started to become popular in the artificial intelligence discipline due to its concision and closeness in syntax to human-readable plain English, the continued use of Python in AI is at least somewhat attributable to the abundance of libraries built for the task, including NLTK, Tensorflow, and skikit-learn. The growing popularity of programming over the years also correlates with the increasing usage of languages that mimic human language. Languages that dominated the environment in the 60s like Fortran and COBOL had archaic syntax and required utmost precision and focus on the programmer's part to ensure that the program was designed correctly because of the architecture it was intended to be run on. Computers in the 1960s could not sacrifice any amount of performance, so programmers needed to write instructions in the same way computers would interpret them. Figure 5 demonstrates this concept with a function written in C that concatenates two character arrays, compared with a significantly more concise equivalent program written in Python.

```c
// Language: C

// This function returns a new string that is the combination of <a> and <b>.
// The resulting array is dynamically allocated; remember to free it.
char* concat(char* a, char* b) {
    int a_length = 0;
    while (a[a_length] != '\0')
        a_length++;

    int b_length = 0;
    while (b[b_length] != '\0')
        b_length++;

    char *c = malloc(sizeof(char) * (a_length + b_length + 1));
    for (int i = 0; i < a_length; i++)
        c[i] = a[i];
    for (int i = 0; i < b_length; i++)
        c[i + a_length] = b[i];

    return c;
}

```

```c
int main(void) {
    char *str1 = "Hello";
    char *str2 = "World";

    char *str3 = concat(str1, str2); // str3 contains "HelloWorld"

    free(str3);
}
```

```python
# Language: Python

def concat(a, b):
    return a + b

if __name__ == '__main__':
    str1 = 'Hello'
    str2 = 'World'

    str3 = concat(str1, str2)
```

Figure 5: Dynamic concatenation of two strings in C and Python

In C, concatenating two strings and returning the result from a function requires much more code than it would in Python. C first appeared in 1972 in order to implement the Unix operating system, so memory management and explicit data types were necessities given the task they were trying to complete [10]. Additionally, C was meant to run on tiny computers, with the earliest DEC PDP-11 minicomputer containing only 24K bytes of memory. Comparatively, Python first appeared in 1991 with the target audience being computer users who were not already computer programmers or software developers [11]. It was always designed with ease of use in mind, and usability favors abstraction over concretion, meaning redundant, complicated features that performed close to the hardware were hidden from the user. With Python, memory management became implicit, performed behind the scenes instead of explicitly stated by the user with malloc and free. The effect of this is a much different experience for the user as essential features are now automated; programs are capable of being written much faster, a virtual machine lends itself to better error tracing and logging, and the programmer can place a larger focus on processes and functions above individual instructions. Although, this comes with its own consequence of speed: Python programs need to handle garbage collection on its own which consumes resources, virtualization requires dynamic translation of instructions and maintenance of a human-readable stack trace, and programs need to predict the types and usage of variables due to the inability of the compiler to know for sure the context in which structures will be used before the program is run. C and Python both seem to be at ends with each other, but this makes choosing a language more of an important decision.

For this reason, programming languages and spoken languages are similar to each other. English and Japanese share a lot of similarities in the thoughts they are capable of conveying, but there are distinct differences between each language that limit their success in communicating specific ideas. A haiku in Japanese is centered around the 5-7-5 combination of Japanese characters, where each symbol is carefully chosen based on not only its meaning at face value but also hidden meanings in the placement of metasymbols within the characters and contextual artifacts throughout the entire haiku. The closest English equivalency would be to replace symbols with syllables and to translate the very limited collection of concepts into words and phrases. During translation, the general idea of the poem may be successfully conveyed, but the exact sentiment and message the poem hoped to address may be impossible to translate with perfect accuracy.

Programming languages share this similarity. Algorithms are independent from language and are capable of being shown in different languages—as shown previously with the matrix-multiplication example depicted in Figure 3—but this does not mean it is correct to depict any program in any language. Figure 3 featured identical program segments translated between programming languages, but the programs they were taken from could not be more different from each other. The matrix-multiplication algorithm shown would have very poor performance on dynamically-typed interpreted languages, as any code with frequent loops would. Each iteration of the loop would require the language to determine the type of the objects being operated on, which would significantly decrease the runtime performance of the program. The "pythonic" way of performing a matrix multiplication would be to utilize Python's numpy module's `matmul` function, which multiplies two input matrices together dramatically quicker than it would through a 3D for-loop like the one shown in Figure 3. This can be summarized in Figure 6, where the direct conversion between a source language and a target language without regard for semantics is "transcription", while the conversion with proper consideration for best practices and the underlying intent of the language is "translation".

Original	Transcription	Translation
お腹が空いた	My stomach is empty	I am hungry

`int arr[5];`
`for (int i = 0; i < 5; i++)`
`  arr[i] = i * 2;`
	`arr = [None] * 5`
	`for i in range(5):`
	`  arr[i] = i * 2`
		`arr = [(i * 2) for i in range(5)]`

Figure 6: Transcription vs. translation of languages		

In Python, there is nothing inherently incorrect about transcription; the result is fully executable code that can be run on any platform. The problem is that it does not take into consideration the ideals Python represents; Python is not built for the same purpose as C,

so the same program will not run optimally in a different environment. This is a sentiment that influences a lot of decision making on programming language use: problems are capable of being solved in many different ways, but the optimal way to solve a problem requires careful consideration of language's implementation itself.

Syntax as a Metric

Performance along with the list of attributes listed at [8] have been considered a fairly crucial metric when choosing the best language to program a project in, but they do not focus on the representation of the problem. For example, a programmer can make a solid defense to use Python for a natural-language processing project instead of C if only because Python has an abundance of libraries that support it, but this does not give any explanation of how the design of the language facilitates the task. External support for a language is important because it ensures bugs that out of the programmer's control are minimized and repaired, but support does not affect the code the developer created to solve the problem. Therefore, the rest of this paper will consider only one defining qualifier to distinguish languages: syntax. Syntax describes the placement of tokens in the source code solution of a problem defined in a programming language. Python's syntax can be viewed as concise, closely related to human-speakable dialogue, while C's syntax is verbose, mechanical, and concrete. Assuming that all other factors about a language are equivalent—including the number of external libraries present, the amount of support the language has on different architectures, and the overall performance an average program is capable of reaching—the syntax of the language is the culmination of the language's innate ability to represent a problem. To these ends, asking the best language to implement a natural-language processing program in is more of a matter of analyzing the syntax of the language. What about the design of Python encourages NLP tasks? Why does the concrete, low-level nature of C not support NLP as well as Python's abstract, high-level design does? What makes Python uniquely suited for this problem above all other languages?

A common misconception suggested by programmers is that older programs like Fortran and COBOL would become obsolete compared to Python and Ruby as soon as storage and execution time become irrelevant. After all, high-level languages are much better at interpreting programs that align closely with human-describable language, compared to the machine-like instructions that must be provided to their older counterparts. The problem with this view is that it implies all problems can be explained in the plain English format Python excels in. Consider the code segment in Figure 7, which depicts a simple summation of the values contained within a linked-list data structure of node objects.

```
1 total = 0                    # Set "total" equal to 0
2 node = list.first            # Set "node" equal to the first element of the list
3 while node is not None:      # While node is not None
```

```
4       total += node.value     # Increment total by the node's value
5       node = node.next        # Set node to the next node
```

Figure 7: Summation of values contained within a linked-list, with plain English comments after each statement

The comments following the lines of code are similar to what a teacher would say when describing the program to a student. Notice the similarities between the plain English text and the actual source code depicted on the left: the Python language prioritized "readability" and "learnability" within its design, so it uses simple syntax with real-world connections. An English-speaker would especially appreciate line 3 whose English description exactly aligns with the corresponding source code, which would not be the case for other languages like C or Java. This example makes it evident that the goal of Python is to create human-readable code, but this does not properly characterize all kinds of computing problems. Equation 1 contains the sine Taylor series, which is a method of calculating the sine of an input value *x* in radians, where the accuracy of the result increases as the upper bound of *n* increases.

$$sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)!} \quad (1)$$

If this equation were to be represented in a language like Python, it would first need to be translated to a plain English equivalency as before. The easiest way to do this would be to speak Equation 1 out loud: it may look simple enough, as the equation consists of nothing more than a summation, exponentials, and a factorial, but the sentence produced would be convoluted, long, and difficult to follow without the accompanying visual aid of Equation 1:

> "The sine of an input value *x* equals the summation of a resulting value, where the summation has a variable *n* wrap between zero and a sufficiently-high upper bound. The summation value consists of a fraction: the numerator is equal to negative one to the power of *n*, times x to the power of two times *n* plus one; the denominator is equal to the factorial of two times *n* plus one."

Translating the plain English description of the equation into equivalent Python code is straightforward enough with the help of the math library in Figure 8.

```
1  import math
2
3  def sin(x):
4      UPPER_BOUND = 5
```

```
    return sum([
        math.pow(-1, n) * math.pow(x, 2 * n + 1) / math.factorial(2 * n + 1)
        for n in range(UPPER_BOUND)
    ])
```

Figure 8: Python implementation of the Sine Taylor Series

Notice that each statement of the implementation shown above can be directly traced to the English description provided earlier, but a significant amount of readability has been lost in translation; this kind of function would be difficult to understand the meaning of if Equation 1 was not accompanied with it as a visual aid. With this example, it becomes clear that all programs cannot be represented with a single ubiquitous language. Problems relating to mathematics are capable of being represented in plain English equivalencies and further translated into Python, but it would be best to keep them in their original representations.

In the case of mathematical expressions, APL is a language which is built to properly interpret them. APL was developed by Kenneth Iverson in 1960 as a notational tool, meant to describe unique computational algorithms to students with a scope more specific to computing than what regular mathematical notation could offer. Iverson received the 1979 ACM Turing Award for his contributions to developing APL and his accompanying paper, Notation as a Tool of Thought [12], is a foundational contribution to the topic of computer programming languages. APL is famous for its unique character set with symbols beyond those offered in standard ASCII having important purposes. For example, below is an implementation of the Chinese Remainder Theorem—a method of finding solutions to modulo expressions—in APL:

```
crt←{m|⍵+.×⍺(⊣×⊢|∘⊃{0=⍵:1 0 ◊ (⍵∇⍵|⍺)+.×0
1,¯1,-⌊⍺÷⍵}⍣≡⍺÷⍨m←×/⍺}
```

This may look like complete nonsense to a programmer unfamiliar with APL, but refer back to how Equation 1 would appear to a layman in math. Each symbol has a precise meaning that applies an operation to the final result, similarly to what an actual mathematical equation would perform. In the context of computer software, the expression is similarly structured and executable in a format computers could interpret. Alternatively, the same theorem can be implemented in Java in 30 lines of code, but this suffers the same problems expressed earlier with the sine Taylor series: what benefits in a language with a closer analogue to English suffers in immediate understandability. In order to effectively achieve those 30 lines of code, shortcuts must be made and readability suffers in and of itself. APL may be excessively concise and archaic to all but highly-experienced programmers, but if programmers are sufficiently trained in the language and the only factor to the language decision depends only on the syntax (as our assumption previously made), then APL is a

strong contender for the best way to represent this problem. The best language for any task is its ability to represent the problem and its solution clearly, quickly, and effectively.

Abstraction

Significantly complex programs with millions of lines of code would benefit from hiding some details about implementation from both the end user and other developers. The less a programmer has to memorize, the more they can focus on improving their program to work in new contexts to meet new requirements. Abstraction is the "selective emphasis on detail" [13], shifting the focus of a computer operation from a rigid manipulation of known data between two understood forms to a general description of how concepts can be transformed. This is best demonstrated with Figure 9, which shows two classes with functions associated with them.

```
1  struct dog {
2      char *name;
3  };
4
5  struct cat {
6      char *name;
7  };
8
9  void make_dog_sound(struct dog* d) {
10     printf("%s says: Bark!\n", d->name);
11 }
12
13 void make_cat_sound(struct cat* c) {
14     printf("%s says: Meow!\n", c->name);
15 }
16
17 int main(void) {
18     struct dog d{"Sophie"};
19     struct cat c{"Mittens"};
20
21     makeDogSound(&d);
22     makeCatSound(&c);
23 }
```

Figure 9: Two C structs with associated functions

There is no ambiguity in this implementation, where any creation of a `dog` or `cat` (lines 18, 19) as well as any invocation of `make_dog_sound` or `make_cat_sound` (lines 21, 22) is clearly and easily traceable. A programmer wanting to discover the meaning of a function need only look at the names of the functions and structs being used. This demonstrates concretion, which C has a very strong grasp of compared to other languages. All types are explicitly defined and traceable at compile time, and the exact meaning of a program is

dependent on the machine instructions being performed. The obvious downside of this program design is its repetition; `dog` and `cat` have identical contents, and the operation of the procedures can be decomposed into a more basic, all-encompassing super-function. To exploit the principles of abstraction, we can create explicit combinations of these qualities to improve the code's usability, shown in Figure 10.

```
1  struct dog {
2      struct animal a;
3  };
4
5  struct cat {
6      struct animal a;
7  };
8
9  struct animal {
10     char *name;
11     char *sound;
12     void *subtype;
13 };
14
15 void make_sound(struct animal* a) {
16     printf("%s says: %s!\n", a->name, a->sound);
17 }
18
19 struct animal* create_animal(char* name, char* sound) {
20     struct animal *a = malloc(sizeof(struct animal));
21
22     char *cpy_name = malloc(strlen(name));
23     a->name = cpy_name;
24
25     char *cpy_sound = malloc(strlen(sound));
26     a->sound = cpy_sound;
27
28     return a;
29 }
30
31 struct animal* create_dog(char* name) {
32     struct animal *a = create_animal(name, "Bark");
33
34     struct dog *d = malloc(sizeof(struct dog));
35     d->a = a;
36
37     a->subtype = d;
38     return a;
39 }
40
41 struct animal* create_cat(char* name) {
42     struct animal *a = create_animal(name, "Meow");
43
44     struct cat *c = malloc(sizeof(struct cat));
```

```
45      c->a = a;
46
47      a->subtype = c;
48      return a;
49  }
50
51  int main(void) {
52      struct animal* a1 = create_dog("Sophie");
53      struct animal* a2 = create_cat("Mittens");
54
55      make_sound(a1);
56      make_sound(a2);
57
58      struct dog* dog_cast = (struct dog*)a1->subtype;
59
60      // ... {deallocate memory}
61  }
```

Figure 10: Added abstraction to C types, with repeated functionality combined

With this change, the `dog` and `cat` subtypes have had their similarities combined while keeping their differences separate. Each subtype structure has a reference to their base type (lines 2, 6) which itself has a reference to its implementing type (line 12). This implies a contract between `dog`, `cat`, and `animal`: the functionality of `dog` and `cat` must at least encompass the functionality of `animal`. In return, the subtypes can be used in any context where the supertype is required, thus allowing the combination of the previously-redundant `make_dog_sound` and `make_cat_sound` methods into a singular `make_sound` method. Furthermore, a generic animal can be converted back into its subtype through standard casts (line 58), regaining the specialized functionality that the specific type described. The usability of this code can be extended further with the use of the "factory" design pattern—which delegates the tasks of creating objects to a separate function—by combining the `create_dog` and `create_cat` methods into a single method that handles generation of subtypes.

Figure 10 showed an example of situations where repetition in code is avoided with abstraction. Previously in Figure 9, two separate code locations specify how an animal makes a sound, which is an operation that all types of animals share. If the program represented a database for a zoo, and thus required the addition of hundreds of different types of animals, then the modification of the `make_[x]_sound` function would become redundant and prone to errors. The combination of these different types into an enclosing supertype simplifies the modification of this operation as well. Although, this is not the only usage of abstraction; another large purpose is to hide the details of a specific function. Figure 11 provides another example.

```
1  struct animal {
2      char *name;
3      char *sound;
4      void *subtype;
5
6      int position;
7      int speed;
8
9      // Moves the animal <speed> units forward
10     void (*move)(struct animal*);
11 };
12
13 // Transports <creature> forwards so that their <position> is greater than or
14 // equal to <destination>
15 void transport(struct animal* creature, int destination) {
16     while (creature->position < destination) {
17         creature->move(creature);
18     }
19 }
```

Figure 11: Creation of a new function without knowing the implementation of a sub-function

The `animal` structure has been modified to account for its position (an integral value, perhaps representing its linear distance from a startpoint), speed, and a function describing how it can move itself forward. While every `animal` can make a sound the same way in the context of our program, each `animal` instead has a different unique style of locomotion; dogs and cats can walk, horses can gallop, birds can fly, and caterpillars can inch. It would be impossible to combine these distinct methods of transportations under a single method, so the animal structure is adapted with an abstract method that specifies a requirement that must be fulfilled. The `move` method can do anything it needs as long as it somehow moves the animal forward by a standard amount. In this way, abstract methods typically specify preconditions and postconditions while intentionally hiding the concrete methods completing those conditions. The benefit of this is apparent with the `transport` method (line 15): every animal is capable of completing the requirements specified within it the same exact way so we can utilize the abstract method `move`, which has implementation specified by concrete subtypes, to accomplish its goal.

The disadvantage of this approach to programming is the same as its benefit: the implementation details are hidden from the programmer. It is no longer possible to understand the exact operations being performed on line 17. The programmer is able to garner the result of the operation and can understand how the object may be different before and after it is run, but they have no way of knowing what exact hardware processes are performed inside. To modify the movement style of an abstract animal would require identifying its type and locating the definition of its `move` method. Programmers

attempting to understand the program at a concrete level will have to look in more places to do so. Python is a prime example of the extent of abstraction—which was a central concept considered during the development of its design—depicted in Figure 12 with an example.

```
1  print(sum([5, 3, 1, 7, 2, 9])) # 27
2
3  def product(items):
4      total = 1
5      for item in items:
6          total *= item
7      return total
8
9  sum = product
10
11 print(sum([5, 3, 1, 7, 2, 9])) # 1890
```

Figure 12: Replacing a language-defined method with a user-defined alternative

In this example, `sum` is a function defined by Python to return the result of the items in an iterable object added together. Because Python is extensible and abstract by nature with very few reserved keywords, we can replace the method with our own version (line 9). The function is being used the same way on line 1 and 11, but the user-defined version computes the product instead of the sum. Overriding keywords in this way is not traditionally used in Python, but it has a benefit in replacing the implementation of operations that should be used in new ways that Python is normally incapable of handling. A simple example of this is shown in Figure 13; a programmer may find it strange that the `sum(int)` throws a `TypeError` since numbers are not iterable, so they can correct this by overriding the builtin function with their own version that handles the edge case.

```
1  print(sum(5)) # TypeError
2
3  def new_sum(items):
4      import numbers
5      if isinstance(items, numbers.Number):
6          return items
7      else:
8          total = 0
9          for item in items:
10             total += item
11         return total
12
13 sum = new_sum
14
15 print(sum(5)) # 5
```

Figure 13: Replacing a language-defined method with a user-defined alternative

The drawbacks of keyword-overriding in this example are the same as with the C example: when using the newly-replaced `sum` keyword in another location in the program, the programmer is unsure about what exactly goes on within the method. Preconditions and postconditions may be enough to complete an execution of a program, but still more must be known to understand how the program completes its execution. For a program which is performance-intensive or precision-critical, knowing the steps involved in how a simple sum is computed can be more important than the benefit presented by implementation hiding. In a different file at another point in the program, a programmer may rely on the assumption that sum will throw a `TypeError` if given a non-iterable number and create their own measures to prevent that, not knowing that the replaced function runs just fine without it. Extending this concept to Figure 11, a programmer working with the abstract `move` method may not know that one concrete implementation for a subclass (say, an elephant) is very time-intensive, and as such requires an alternative method to be run instead for that specific case. Whether or not this is good programming design is outside of the scope of this paper but the fact the language encourages this is noteworthy.

A final note about abstraction is the ability of a language to express concrete ideas given certain syntactic constructs that enable overriding. Here is a code snippet in C taken from NVIDIA's collection of CUDA samples:

```
*((copy_t*)(shmem_warp_stream_ptr + SHMEM_STRIDE * i) + laneId) =
    *((copy_t*)(src_gmem_warp_stream_ptr + GLOBAL_MEM_STRIDE * i) + laneId);
```

This line was found in the middle of a function designed to run in parallel on a GPU utilizing a special technology called Tensor Cores to compute the matrix multiplication and addition of three input matrices. It may be hard to explain initially how this line works, but it can be easily decomposed into simpler subproblems by recognizing patterns:

- `copy_t` must represent a type, so `(copy_t*)(value)` casts some value.
- `*(value)` retrieves the result stored in a pointer when the asterisk does not represent multiplication, so the entire statement can be visualized as `*(pointer) = *(pointer)`, assigning the value stored at one pointer to the value stored in another.
- You can only multiply numbers together, but you can add two numbers, two pointers, or a number and a pointer together.

By applying these rules, we can reinterpret the complex statement as a much simpler alternative:

```
Set the value at [(type)(pointer + number * number) + number] to
    the value at [(type)(pointer + number * number) + number)
```

This enhancement to readability can only be performed because C does not allow operator overloading, and because C has a very rigid set of rules that must be applied to operands. Even without viewing a compiler or IDE with syntax highlighting, the exact hardware instructions being performed remain clear to any spectator, but the same cannot be said for Python. Operator overloading is a concept that benefits programmers who want to exploit abstraction. The message that an addition operator attempts to convey is not rigid and can apply to many different concepts. Adding two positive numbers yields a number larger in magnitude than either input, but adding two user-defined classes like spreadsheets or buildings does not need to have a strict logical meaning standardized in the real world. This has the potential to increase the readability of code by reducing programmatic statements into concepts. A more practical example of operator overloading is shown in Figure 14.

```
1  class Matrix:
2      def __init__(self, rows, cols):
3          self.rows = rows
4          self.cols = cols
5          self.vals = [[i + cols * j for i in range(cols)] for j in range(rows)]
6
7      def __mul__(self, other):
8          new_mat = Matrix(self.rows, other.cols)
9          for i in range(self.rows):
10             for j in range(other.cols):
11                 new_mat.vals[i][j] = 0
12                 for k in range(self.cols):
13                     new_mat.vals[i][j] += self.vals[i][k] * other.vals[k][j]
14         return new_mat
15
16 a = Matrix(5, 10)
17 b = Matrix(10, 3)
18 c = a * b
```

Figure 14: Replacing the builtin multiplication operation for a custom class in Python

In this example, a function is created (line 7) that replaces the multiplication operation so that the multiplication between two matrices yields a new matrix instead of a `TypeError`. Since matrices (and other user-defined classes) have no intrinsic meaning in Python, the language allows developers to create their own definitions of operations, extending this to more builtin operations beyond `__mul__` like `__len__` (the "length" of an object), `__repr__` (a displayable version of an object), and `__hash__` (a numerical representation of an object). Each of these functions are used throughout Python in different ways (for instance, `__radd__` is used when computing the summation of a list of user-defined objects during Python's `sum` function), and they marginally simplify the visual

complexity of the program as long as the function's usage is obvious enough. Despite this, the abstractive problem persists: for a casual onlooker, the definition of line 18 in the above program is a mystery. Even worse, since objects `a` and `b` do not explicitly include a type declaration, the multiplication of two objects with unknown types from within somewhere else in the program may even confuse a programmer (multiplication typically occurs between numbers, but what if the objects are matrices?). The consequence of operator overloading—and abstraction in general—presents only as large a benefit as the knowledge of the programmers involved of the program being developed.

Elegancy

A common adjective programmers use to describe languages is "power"; in essence, powerful languages are those which enable the user to do a lot of different things. The term should be considered more informal than a substantive indicator of usability, where "Turing completeness" is a much more applicable term regarding how well a system can be used to solve computational problems. Although, a very large number of systems can be said to be Turing complete, and therefore capable of solving any computational problem, including the physical card game *Magic: The Gathering* [14] and Java generics [15]. The logical conclusion of this sentiment is that every Turing complete programming language—of which most are, besides formatting languages like HTML or Markdown—is capable of solving any kind of computational problem, so what other metrics besides "readability" can be used to describe a language? In this case, "elegant" is a better adjective that describes the simplicity, expressiveness, and overall ingenuity of a segment of code.

One helpful indicator of elegancy is the amount of symbols required to describe an idea. Real-world languages tend to avoid redundancy when creating sentences, preferring to keep sentences relatively concise and expressing ideas with especially extreme importance using words of large magnitude—like "horrible" or "excellent"—to add emphasis instead of repeating words of tiny magnitude like "really really really good". Programming languages are similar in that especially verbose sections of code are considered "bad design" when compared to elegant alternatives.

The book *Beautiful Code* [16] is a collection of case studies written by 38 different computer scientists with each chapter describing a unique solution to a niche computing problem. The kinds of solutions presented span a diverse range of programming domains, including developing Python's dictionary data structure and multidimensional iterators in numpy to designing extendible modules in a Perl bioinformatics library that keep it useable years after its development in situations it was never intended to be used in. The first chapter describes an especially terse program of 30 lines developed by software engineer Rob Pike in 1998 that implements a regular expression matcher on a subset of patterns, given in Figure 15:

```c
/* match: search for regexp anywhere in text */
int match(char *regexp, char *text) {
    if (regexp[0] == '^')
        return matchhere(regexp+1, text);
    do {    /* must look even if string is empty */
        if (matchhere(regexp, text))
            return 1;
    } while (*text++ != '\0');
    return 0;
}

/* matchhere: search for regexp at beginning of text */
int matchhere(char *regexp, char *text) {
    if (regexp[0] == '\0')
        return 1;
    if (regexp[1] == '*')
        return matchstar(regexp[0], regexp+2, text);
    if (regexp[0] == '$' && regexp[1] == '\0')
        return *text == '\0';
    if (*text != '\0' && (regexp[0] == '.' || regexp[0] == *text))
        return matchhere(regexp+1, text+1);
    return 0;
}

/* matchstar: search for c*regexp at beginning of text */
int matchstar(int c, char *regexp, char *text) {
    do { /* a * matches zero or more instances */
        if (matchhere(regexp, text))
            return 1;
    } while (*text != '\0' && (*text++ == c || c == '.'));
    return 0;
}
```

Figure 15: Regex matcher in 30 lines of C code

The ingenuity of the program is evident on even a glance, exploiting the pointer arithmetic of C and frequent recursion to search for combinations of patterns in the text. The length of the example is miniscule compared to its applicability in a variety of situations; longer, more verbose code may gain in some amount of readability at a glance of the code, but a deeper look at the steps the program takes to efficiently locate a match in the text communicates meaningful ideas that cannot easily be replicated in longer variants. The size of the program also perfectly captures the simplicity of the operation, fitting neatly on a single screen and broken into digestible parts that are each easy to understand on their own.

In the same way program segments are elegant, entire programming languages can be elegant too. Concision in implementation is not as large as a factor, but concision in the resulting code created with the language is. Although, keeping in mind that different

languages are built for different tasks (and thus, languages like APL and Python are incomparable in some respects due to their differing goals and use cases), there are features in any programming language that are staples in their ability to uniquely solve any problem. For example, C has two excellent design factors that are indispensable in defining elegant code written within it: pointer arithmetic and preprocessor directives. The former was already shown in Figure 15, where the incrementation of the `text` and `regexp` pointers enable the program to describe complex ideas in little code, but the latter can be shown in Figure 16 with another example.

```
#if (CUDART_VERSION >= 9020)

__host__ void ndarray_to_half_arr(half* A, half* B, ndarray* h_A, ndarray* h_B,
                                  int m_global, int k_global, int n_global)
{
    // ...
}

#elif defined(USE_GPU_SIMULATOR)

__host__ void transfer_arr(half* dest, float* src, int dest_rows, int dest_cols,
                           int src_rows, int src_cols)
{
    // ...
}

__host__ void ndarray_to_half_arr(half* A, half* B, ndarray* h_A, ndarray* h_B,
                                    int m_global, int k_global, int n_global)
{
    // ...

    transfer_arr(A, h_A->arr, m_global, k_global, h_A->shape[0], h_A->shape[1]);
    transfer_arr(B, h_B->arr, k_global, n_global, h_B->shape[0], h_B->shape[1]);

    // ...
}

#else

__global__ void transfer_arr_kernel(half* dest, float* src, int dest_rows,
                                    int dest_cols, int src_rows, int src_cols)
{
    // ...
}

__host__ void ndarray_to_half_arr(half* A, half* B, ndarray* h_A, ndarray* h_B,
                                  int m_global, int k_global, int n_global)
{
    // ...

```

```
41      transfer_arr_kernel<<<1,1>>>(A_re, A_arr, m_global, k_global,
42                                   h_A->shape[0], h_A->shape[1]);
43      transfer_arr_kernel<<<1,1>>>(B_re, B_arr, k_global, n_global,
44                                   h_B->shape[0], h_B->shape[1]);
45
46      // ...
47  }
48
49  #endif
```

Figure 16: Snippet from a hardware-accelerated matrix-multiplication algorithm meant to run on GPUs

This code was written in CUDA for C, designed to preprocess matrices before they were multiplied and added together. The two input matrices, named A and B, must be converted into a special half-precision floating-point data type, but the way this gets completed can vary. The preprocessor macros allow a different version of the program to be compiled based on a condition. The first macro (lines 1-9) only compiles if the version of CUDA being used is greater than or equal to 9.2; the next version (lines 9-28) only compiles if CUDA's version is less than 9.2 and is being run on a GPU simulator instead of actual hardware; the final version (lines 28-49) only compiles if CUDA's version is both less than 9.2 and being run on hardware. Three different versions of this program exist for very specific reasons: conversion between standard floating-point and half-precision values can be performed on the CPU with newer versions of CUDA, but it must be done directly on the GPU if the CUDA version is old. Additionally, if the program is being run on a simulator, anything run on the GPU will take a very long time and should use special code developed by the simulator instead whenever possible. Finally, this program will not work if the macros were replaced with if-statements and branched dynamically due to compiler-only operations (half-precision conversion and simulator-only functions) not existing during compilation in different circumstances. Because of this, the program design is comparatively more compact than it would be in a language Java with different conditionally-compiled source files and complex GNU Makefile productions to control linking, and the entire implementation is elegant by not requiring the organization of an external variable to control the branching with an if-statement nor the requirement that a simulator be downloaded for users who would never utilize it due to having a newly-updated GPU.

Alternatively, Python has its own syntactic design elements that make it unique. Most of these involve abstraction and fluidity in identifier meaning, but a much more concise example can demonstrate the different builtin string variants. A Python string is a collection of symbols placed with quotes, and they can be created in different ways:

```
normal_str = 'hello world'
print(normal_str) # hello world
```

Formatted strings simplify the process of creating complex string results, usually for the purpose of displaying variables to the screen, and come in different flavors depending on use case (with bracket notation, format-specifiers, and f-strings shown):

```
name = 'Sophie'
pi = math.pi

normal_str =  'Name: ' + name + ', value: ' + str(round(pi, 3))
f_str_1    =  'Name: {}, value: {}' % (name, round(pi, 3))
f_str_2    =  'Name: %s, value: %.3f' % (name, pi)
f_str_3    = f'Name: {name}, value: {round(pi, 3)}'

print(normal_str) # Name: Sophie, value: 3.142
print(f_str_1)    # Name: Sophie, value: 3.142
print(f_str_2)    # Name: Sophie, value: 3.142
print(f_str_3)    # Name: Sophie, value: 3.142
```

Two adjacent string literals can also be appended, which can be utilized along with backslash line-extension to create simplistic multiline strings, useful for storing long error messages within a source file without having it extend past a character limit:

```
basic_example = 'hello'      'world'
multiline_str = 'hello' \
                'world'

print(basic_example) # helloworld
print(multiline_str) # helloworld
```

Raw strings ignore escape characters, perfect for regex strings and debug messages:

```
import re

test_str = 'this   is a   very   long  sentence'
regex_normal =  '[\\s][\\s]+' # Backslash needs to be escaped here
regex_raw    = r'[\s][\s]+'   # Raw strings ignore escape characters

print(test_str) # this   is a   very    long  sentence
print(re.sub(regex_normal, ' ', test_str)) # this is a very long sentence
print(re.sub(regex_raw,    ' ', test_str)) # this is a very long sentence
```

Unicode strings are used by default in Python3, but they allow representation of non-ASCII characters in Python2:

```
print  'El Niño' # SyntaxError: Non-ASCII character
print u'El Niño' # El Niño
```

Finally, byte strings represent characters as a machine-readable `bytes` object (which is internalized as a sequence of octets) instead of a human-readable str (which is a sequence of unicode characters), skipping the encoding process and letting the user represent low-level binary data like structs:

```
normal_str =  '\xff\xf8\x00\x00\x00\x00\x00\x00'
byte_str   = b'\xff\xf8\x00\x00\x00\x00\x00\x00'

print(normal_str) # ÿø
print(byte_str)   # b'\xff\xf8\x00\x00\x00\x00\x00\x00'
```

Although, none of these string representations allow the programmer to write especially "new" programs; formatted strings are nice to have and literal appending is useful, but these are easily accomplished with either user-defined functions or basic string addition. Furthermore, C's pointer arithmetic only saves the user from needing an index variable to serve as an extra level of indirection, and does not actually solve any special problems it otherwise would be unable to. Despite this, the builtin syntax-enhanced version in both cases add a bit of clarity to user purpose. Instead of adding more backslashes into a regex string, a simple 'r' can be prepended to preserve the standardized regex syntax of the pattern. For this reason, pythonic strings are elegant just like C preprocessor directives: the usefulness they present are special to the language they exist in, and conjure up new programs written in ways that are unique only to a specific language.

Symmetry

Symmetry is a term often used to describe geometry that is equivalent when viewed from different perspectives. In mathematics, objects which remain unchanged after being reflected, translated, or rotated can be described as symmetric [18]. At a minimum, "the very least we need for symmetry is the possibility of making a change and some aspect that is immune to this change" [19]. This idea can extend towards programming languages to describe how a single concept can be utilized in many different ways, with symmetrical syntactic constructs having loose constraints in order to encourage their use in unique ways and asymmetrical constructs imposing rigid restrictions on what the programmer is capable of doing.

The concept of symmetry allows programmers to abstract an idea to the extreme, reducing an unspecific idea to a multitude of forms. Although, languages that are turing complete cannot be described as entirely symmetrical or asymmetrical due to the relativeness of the term. An example of a deceptively asymmetrical language is the esolang Lenguage [23], whose execution depends only on the *size* of the program in bytes instead of the actual content of the file. This size is converted into a binary number which is further interpreted as machine code by a virtual machine, meaning a "Hello, World!" program requires a file with 1.75×10^{100} ASCII characters. Despite this tremendous restriction in programming ability, readability, and overall computability (the example program given would require 1.75×10^{76} yottabytes, while the total amount of data created, captured, or replicated in the world by 2018 was less than a single yottabyte [20]), the language imposes very little restrictions on what can and cannot be done besides that, with users proving that an

implementation of the video game Pong [21] and a simple UNIX shell [22] are both possible with Lenguage. Therefore, symmetry is only a useful quantifier to describe individual concepts within programming languages, and certain languages demonstrate exemplary usages of symmetry throughout their design.

In object-oriented languages, first-class citizens are significant sources for symmetry. In the Structure and Interpretation of Computer Programs [24], first-class citizens are defined as language elements which can be:

- Named by variables
- Passed as arguments
- Returned as the results of procedures
- Included in data structures

Elements that encompass these features are capable of being used in the same way as any other first-class citizen. For example, C functions are first-class citizens because you can store a function in a variable, pass it as an argument, return it from a separate function, and include it within a stuct, while Java methods are not first-class citizens because they cannot be stored in variables. Furthermore, Python and Smalltalk have first-class classes in the form of metaclasses, where descriptions of classes can be stored in variables with instances dynamically instantiated at runtime. Languages which have some degree of first-class citizenship encourage programmers to use their features just like any other feature, which leads to interesting programs as a result.

In terms of syntactical constructs rather than values, symmetry can be expressed in both the ability of a construct to accept a wide range of statements as well as the reuse of constructs throughout the language. The standard C-style for-loop gives an excellent example of this practice:

```
for (variable instantiation; condition; variable increment) {
    // ...
}
```

The obvious intent of the loop is to process a fixed-width collection of elements. For example, a program to add each element in an array of size 10 can be given by:

```
int *array = create_array(10); // Returns an array with 10 random elements
int sum = 0;
for (int index = 0; index < 10; index++) {
    sum += array[index];
}
```

The design of the loop is flexible and can yield different iteration mechanisms; iteration can be terminated early with a more complex condition, the loop can be offset with a different variable instantiation, and the index can increment by a value greater than 1 to skip elements within the list, to name a few. Although, while the design of the loop strongly

suggests it be used in this fashion, it makes no explicit requirements that it *must* be used in this way; in fact, we can decompose the for-loop to an even barer example:

```
for (start_statement; condition_expression; loop_statement) {
    // ...
}
```

`start_statement` is run once at the start of iteration, `loop_statement` runs after the enclosed code runs, and `condition_expression` is run after `loop_statement` and must be reducible to a value. The beauty of this design is that *any* form of expression is capable of being placed within the loop, no matter how convoluted it may seem. Figure 17 shows one such program which implements a depth-first traversal of an adjacency matrix, where the entire logic of the program is within the header (`start_statement`, `condition_expression`, and `loop_statement`, between lines 18 and 40), and no code is within the loop body (line 41).

```
 1  #define SIZE 7
 2
 3  int main(void) {
 4      int nodes[SIZE][SIZE] = {
 5          {0, 1, 0, 0, 0, 0, 1},
 6          {1, 0, 1, 0, 1, 0, 0},
 7          {0, 1, 0, 0, 0, 0, 0},
 8          {0, 0, 0, 0, 0, 1, 0},
 9          {0, 1, 0, 0, 0, 1, 0},
10          {0, 0, 0, 1, 1, 0, 1},
11          {1, 0, 0, 0, 0, 1, 0}
12      };
13
14      int order[SIZE] = {0};
15      int length = 1;
16      for
17      (
18          // start_statement
19          int current_node = 0,
20              inner_index = 0,
21              visited[SIZE] = {0},
22              queue[SIZE] = {0},
23              queue_index = 0,
24              zero = 0,
25              overflow = 0;
26
27           // condition_expression
28           queue_index >= 0;
29
30           // loop_statement
31           visited[current_node] = 1,
32           current_node = (nodes[current_node][inner_index] &&
33                           !visited[inner_index]) ?
```

```c
            zero + 0 * ((overflow ? overflow = 0 :
            (order[length++] = inner_index)) +
                (queue[queue_index++] = current_node) + (zero = inner_index) +
                (inner_index = 0)) :
            (++inner_index == SIZE) ?
                queue[--queue_index] + (inner_index = 0) + (overflow = 1) * 0 :
                current_node + (overflow = 0)
    ) { }

    // Displays: [ 0 1 2 4 5 3 6 ]
    printf("[ ");
    for (int i = 0; i < length; i++)
        printf("%d ", order[i]);
    printf("]\n");

    return 0;
}
```

Figure 17: Breadth-first traversal of an adjacency matrix in C, entirely performed within a loop header

The largest criticism syntax-lenient programs have is evident in Figure 17: it has the potential to encourage terrible design. A problem which *can* be done poorly a certain way should be anticipated by the language developers and persuaded to be accomplished a different, more efficient, and more readable way. This program is nearly impossible to read (and much more time consuming to write than a normal graph-traversal algorithm would have been), but it demonstrates an extreme view of an idea languages should consider. If the `loop_statement` field was required to be an incrementation of a variable declared in the `start_statement` field, then programmers that delegate incrementation to an external function would be forced into rewriting their code to comply with language standards. By making a language adhere to the most general case, creativity and expressibility can flourish in a source file to the benefit (or detriment) of a team.

Python is a more modern example of symmetry in action. While C can make the case for allowing symmetry for performance, Python has instead opted to encourage symmetry in design strictly for the interest of expressibility and ease of programming at the impairment of performance. Python is well-known for having a handful of builtin types that are used all throughout the language design, including the list, tuple, and dictionary. Lists and tuples differ in their runtime mutability, but each are used extensively throughout the language in different ways.

The most visible example is given with classes. Python classes are essentially symbol tables containing identifiers and values. Since a lot of different elements within Python are first-class citizens including functions and objects, the symbol table for a class can be stored internally as a dictionary-like object, shown in Figure 18.

```
1  class User:
2      def __init__(self, name, number):
3          self.name = name
4          self.number = number
5
6      def test(self):
7          print('Test!')
8
9  inst = User('Sophie', 100)
10 print(inst.__dict__)   # {'name': 'Sophie', 'number': 100}
11 print(User.__dict__)   # {..., 'test': <function User.test at 0x7f69060c5e50>, ...}
12 print(type(inst.__dict__)) # <class 'dict'>
13 print(type(User.__dict__)) # <class 'mappingproxy'>
```

Figure 18: Python code demonstrating that instance and class variables are stored within dictionary-like containers

The instance symbol table (line 12) is stored as the same type of dictionary end users have regular access to, but the class symbol table (line 13) is a `mappingproxy`—an immutable dictionary with the added restriction that keys must be strings in order to improve performance. In this case, the symmetry in the language helps the user contextualize its execution. In a language so enveloped in abstraction, this bit of clarity explains why classes act the way they do. With this added context, examine the example program shown in Figure 19, depicting a `Car` class with a `drive` method, which prints one of many different statements depending on the condition of different instance attributes.

```
1  class Car:
2      # Parameters:
3      #   -tires: list(str)
4      #   -driver: str
5      #   -steering_wheel: str
6      #   -airbag: str
7      #   -status: str (either 'perfect', 'good', 'fine', 'bad', 'terrible')
8      def __init__(self, tires, driver, steering_wheel, airbag, status):
9          self.tires = tires
10         self.driver = driver
11         self.steering_wheel = steering_wheel
12         self.airbag = airbag
13         self.status = status
14
15     # Returns a string representing the car's action.
16     # Restrictions before you can drive:
17     #   -Must have at least 4 tires
18     #   -Driver, steering wheel, and airbag must be present
19     #   -Status must be either 'perfect', 'good', or 'fine'
20     def drive(self):
21         if len(self.tires) < 4:
22             print('Not enough tires!')
```

```
23          elif self.driver is None:
24              print('No driver available!')
25          elif self.steering_wheel is None:
26              print('No steering wheel available!')
27          elif self.airbag is None:
28              print('No airbag available!')
29          else:
30              if self.status in ('perfect', 'good', 'fine'):
31                  print('The car is driving!')
32              elif self.status in ('bad', 'terrible'):
33                  print('The car is in poor shape!')
34              else:
35                  print('Unknown state specified!')
```

Figure 19: Sample Python program depicting a Car object with several attributes being accessed within a method named drive.

Without knowing how Python stores attributes, the repetition in the code segment is apparent to any developer. Python is an example of a language which tends to be unnecessarily verbose in some cases in order to keep the language symmetrical, with this example using the term `self` 13 separate times. Although, recognizing that `self` is a stand-in for a dictionary makes the mindset behind the design much more understandable. The overall complexity of pythonic classes can be reduced drastically taking this into consideration; a class method is simply a string (key, function name) associated with a pointer to executable code (value), where the first parameter is a substitute for a dictionary representing the instance. This simplicity can even aid the programmer into considering new solutions to class problems, now that they have a basis for relating the problem to something they understand.

Another example of symmetry in Python is their use of tuples and dictionaries in functions. Due to functions existing in classes through a dictionary, and dictionaries only having one value for any given key, functions do not allow overloading. This means only a single function of any given name can exist within a scope, leading to errors like the one shown in Figure 20.

```
1  class User:
2      def test(self, number):
3          print('Test, one parameter', number)
4
5      def test(self):
6          print('Test, no parameters')
7
8  user = User()
9  user.test()  # Test, no parameters
10 user.test(5) # TypeError: test() takes 1 positional argument but 2 were given
```

Figure 20: Function overloading errors

During execution, Python scans the `User` class from top to bottom, inserting into its symbol table items it finds on the way, but it does not check if an item already has an entry before insertion. This prevents users from organizing functions in terms of parameters, which is a common practice in some other object-oriented languages like Java and C#. To get around this, one potential solution is to use two special arguments instead, shown in Figure 21.

```
1  class User:
2      def test(self, *args, **kwargs):
3          if len(args) == 0:
4              print('Test, no parameters')
5          else:
6              print('Test, one parameter', args[0])
7
8  user = User()
9  user.test()  # Test, no parameters
10 user.test(5) # Test, one parameter 5
```

Figure 21: Function overloading with special arguments

The asterisk syntax is unique to this one instance, but the arguments are simple to comprehend due to them having a shared usage in standard Python: `kwargs` is a dictionary of named parameters (where the key is a string and the value is the passed value), and `args` is a tuple of the other unnamed arguments. There are a few rules that the parameters follow in regards to placement within the argument list and organization of incoming passed variables, but it is a smooth learning curve for programmers attempting to use it. The programmer could take this idea and further delegate differently-named sub-functions for the purpose of organization if they desired, treating the base function as a front-end wrapper for end users as would be the case in Java or C#.

The emphasis in these examples lies on their simplicity: due to them not introducing new, complicated syntactical constructs unique to the purpose they provide and instead using data structures familiar with the programmers with already frequent use, programmers can combine these concepts with existing code bases and libraries that manipulate dictionaries and tuples. To summarize the effect symmetry has on code, it minimizes the level of detail programmers need to memorize in order to accomplish strange things with the language. Standardizing all actions developers take by enforcing strict rules they must follow dampens their productivity, as they focus more on adherence to a ruleset than fluid representation of a solution to their unique problem.

Hello, World!

By this point, the importance of syntax in communication should be thoroughly conveyed. The mere placement of tokens on a document has a tremendous effect in the communication of ideas, and the capability of a language dictates the kind of problems it is built to solve. Two languages which are capable of performing the same tasks are not necessarily interchangeable, and every aspect of a language's design should be scrutinized. Even miniscule details in a language can convey enormous messages about the purpose it is meant to solve; to demonstrate this, "Hello, World!" programs will be analyzed and dissected across different languages, starting with Java's:

```
public class Program {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

The first program that many Java programmers will write is the one shown above. Despite its simple purpose, it comes with it a collection of keywords new programmers may not learn until months into in their programming career. Although, the touchstones of Java development should become immediately apparent with even the first line; everything in Java is contained within classes. Even though no Program instance is being created, a class must be defined to contain the static code. This compartmentalization can also be seen with the actual print statement, where the `out` variable is seen as a member of the `System` class. Additionally, command-line arguments must be supplied and the return type must be void (if the program needs to yield an exit code, then another statement would be used instead of returning the value), which in some ways implies safety and explicitness in design. Java wants programmers to be aware that code creates contracts that must be followed with no compromise, which will be evident to more experienced programmers implementing interfaces and extending abstract classes with strict rules on syntax. Finally, the argument list in the parameter being of type `String` with empty brackets afterwards shows the absence of pointers in Java, with `String`'s coalesced into their own class object and arrays offering more safety in restriction than pointers would allow. Beginner programmers who develop their own code may also notice that the `String` is the only capitalized type out of the common types they are most likely to employ in their programs when starting out, foreshadowing its differences in primitiveness (compared to the `int`, `double`, or `char`), as well as the array being the only multi-type storage structure besides the class that is defined with unique syntax (whereas types like the `ArrayList` are instantiated alike any other object). These insights can be shown in direct contrast to the tenants of Python, demonstrated efficiently within their own "Hello, World!" program:

```
print('Hello, World!')
```

This Python3 program makes an effort to be as concise as possible with only a single line of code compared to Java's five. Python does not waste any space in conveying an idea as simplistic as displaying content to the screen, and obviously prioritizes removing boilerplate code and meaningless compartmentalization compared to Java. This streamlines the creation of code, but also notes that not every feature of the language should be utilized when not needed; Java insists programmers recognize that command-line arguments exist and are supplied regardless of whether or not they will be used, while Python suggests that the programmer will implement these measures when they want to (through a separate imported `sys` module containing the arguments within). This can be further shown through the concept of interfaces in Java—a collection of method declarations that denote operations an object must be capable of completing—which does not exist in Python. Java-styled abstraction can only be completed in Python through importing a special module that provides method decorators, but this comes with it its own set of complications. From a design perspective, shifting certain features behind explicit import statements implies that those features are not intrinsic to the language. In other words, programmers who want to learn Python may consider learning Java-styled abstraction to be secondary to other features built directly into the language like list comprehension or argument extraction, similar to how Python's "numpy" module is externally-defined but also fundamental to learn for programmers looking to utilize the language. Of course, this is an idea that is wholly independent on the language, as even Java introductory courses will utilize import statements to bring in complex storage types very early in any programmer's journey. The consequence of Python incorporating so much within the language design in terms of syntax—with lists, sets, tuples, and dictionaries being intrinsically defined and supporting their own unique methods of instantiation—is shown as soon as it becomes necessary to define what is "necessary" for beginners to learn. Regardless, another idea expressed previously was that of the importance of keywords. Python3's "Hello, World!" program introduces a new complication when compared with Python2's version:

```
print 'Hello, World!'
```

The removal of parenthesis does not apply to user defined functions, as Python2 instead elevates the `print` identifier from a regular function to an intrinsic keyword with its own special rules and requirements. This keyword takes either zero or one values as arguments which are displayed to the console when executed. The problem with making `print` a special keyword instead of a regular function is that it separates it from the standardized meaning of a function as a discrete operation given data. Functions are operations coalesced under an identifier, which accept values as inputs that are changed with results yielded; a keyword does not carry with it the same connotations as a standard function

does due to its inherent uniqueness. Even if it is nearly identical to a regular function under the hood, it carries with it new semantic qualities that change the perception of its usage. Additionally, the keyword does require changes to be made in how the operation is performed. A regular function uses parameters as both operands to be modified and flags to determine the operation being performed. Similar to how an input to an abstract matrix multiplication function can be either another matrix or a scalar value, the algorithm being performed is determined by the operands being supplied. A programmer is capable of separating this matrix-multiplication example into two distinct sub-operations to add clarity to the program, but it is often good design to abstract the function into branching paths depending on the context. In this way, the `print` keyword does not allow a branching path because it does not accept parameters. The purpose of the keyword is always to place the printable value of a parameter directly into the standard output stream, and it is impossible to modify this operation without adding more code. By comparison, Python3 lets the user flush the output stream, define what arguments should be separated and followed by, and replace the output stream with an entirely new stream (for example, the standard error stream or a user-created logger) all through the use of passed optional arguments into the print function. The arguments allow the meaning of the keyword to be less strict with more customizability, while Python2 cannot say the same. Nevertheless, another concept Python's two programs demonstrate is the idea of builtin functionality, which is elaborated further through C's "Hello, World!" program:

```
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
}
```

C differs from the other languages listed so far in that it does not assume the user always wants to display content to the console, so it moves the common input/output functions into an external library for the user to manually import on their own. This behavior is similar to Java in a sense due to Java automatically importing the "java.lang" module in all programs [25], which contains the aforementioned `System` class along with object wrappers of each primitive type, a collection of the most common errors programmers will encounter, and interfaces user-defined classes may want to implement. This is similar in behavior to C automatically prepending an `#include` of `stdio.h`, `string.h`, and `stdlib.h` to every source file, so what is the benefit of C requiring manual inclusion? While C and Java run-time performances are unaffected by import statements, the usage of imports are necessary to avoid naming collisions in files. For example, if the user defines their own class named `ArrayList` and then imports `java.util.ArrayList`, the compiler will raise an error indicating its confusion on which type to instantiate when an object of type `ArrayList` is requested to be created. C is especially prone to this error due to a lack of namespaces which requires functions to be named entirely unique, causing

some programmers to adapt their own naming conventions to avoid collisions (like C's `pthread` library prepending "pthread_" to every member, resulting in function names like `pthread_create` and `pthread_join`). By avoiding automatic imports can C not only encourage the user to create functions without worrying about collisions, but also to override the functionality presented by an identifier. An example of this from a Python perspective was shown in Figure 13, where the predefined `sum` keyword was replaced with a user-defined alternative that accounts for non-enumerable values, but the concept can extend to C by, for instance, replacing the `printf` function with an alternate version that adds new functionality by introducing a new format specifier. Although, the benefit of automatic imports is ultimately debatable, as the boilerplate code it removes is oftentimes negligible and the push towards abstraction can lead to confusion.

Besides this, a few other observations can be made about C's "Hello, World!" program. First, the main function shown is not as rigid as Java's: the return type is specified to be an `int` but it could also be `void` (the presence of an explicit `return` statement at the end of the function is also optional), and the parameters could optionally include a spot for command-line arguments. This initially conflicts with the quality of "exactness" C's design emanates. As a language, C requires programmers be confident in their programming abilities, to the benefit of creating performance-critical code but to the detriment of producing hard-to-detect bugs. Although, just from the four lines of code presented above, Java can be suggested to value exactness and precision more than C does. In terms of language design, both C and Java offer the same level of "abstraction" due to Java's explicit hierarchy of class-extensions and interfaces and C's usage of preprocessor directives, typedefs, unions, and first-class functions, so its impossible to rank the two in terms of how "concrete" programs created with them are. The virtualization of Java code may make the execution of a program marginally more abstract due to it running interpreted code through a virtual machine rather than the target hardware itself like C does, but the layout and design of source code created in each language does not imply abstraction or concretion any differently. Therefore, to separate the designs of the two languages, it can be said that Java is more explicit than C is to some extent: when implementing interfaces, passing data to functions, or executing arbitrary code, Java prioritizes the semantic correctness of the code to a greater extent than C. Java will not let programmers cast an arbitrary variable to a different type unless it can know to a reasonable extent that the cast is realistic (for example, casting an abstract `List` to a concrete `ArrayList` may fail at runtime despite it making logical sense enough to not raise any errors during compilation). On the other hand, C does not make this restriction, with the user freely being capable of converting structures of one type to any other type through a series of dereferences, conversions into the abstract `void*` type, and casts into the final target type. The "Hello, World!" programs above each demonstrate these intrinsic differences between the languages, illustrating that C's design suggests nothing more than the readability of the

program to the *machine* along with the trust in the programmer's ability to deliver correct instructions to the machine, while Java's design prefers the both the readability and conscious accountability of the programmer through contracts to follow and preconditions/postconditions to meet.

So far, the only languages whose "Hello, World!" programs have been considered were each general-purpose languages, which is a descriptor that does not apply to a language like PostgreSQL:

```
CREATE OR REPLACE PROCEDURE main()
LANGUAGE plpgsql
AS $$
DECLARE
BEGIN
    RAISE NOTICE 'Hello, World!';
END;
$$;

CALL main();
```

PostgreSQL is most often used as a backend for managing relational databases, offering procedures and traditional programming language objects like loops and conditionals to structure database queries. Since the language is event-oriented and built to service requests for data, and therefore having many different "start points" compared to a program made in Java, Python, or C, code is often formatted across scripts that execute previously-initialized procedures. The concept of a "main" function does not typically apply in the same sense as it would for the other languages shown, but it is best replicated as it has been above with a "startup" procedure which can be used to invoke other procedures, create new databases and supply initial data, or otherwise initialize the rest of the program. In this way, the form the program takes is visually distinct compared to other languages which require a defined main start-point from which the rest of the program beings from.

Block-structured languages like PostgreSQL place a larger emphasis on the location and organization of statements, where certain categories of statements like assignment typically appear in their own sub-block within a larger block surrounded by explicit "BEGIN" and "END" labels. Structured language design does not yield any unique programs to be created or problems to be solved in the same way that Java-styled abstraction does not uniquely solve problems that were otherwise unsolvable in C, but it gained popularity in the 1960s as a measure to make source code more readable after the Böhm–Jacopini theorem [26] and open letter by Edsger W. Dijkstra [27] noted the benefits of organizing programs into definite structures as well as the harmfulness and primitiveness of the "goto" statement in designing readable programs. The organization of statements takes some amount of agency away from the programmer for the benefit of yielding safer, clearer programs, in the same way that Java's removal of explicit pointer types in lieu of classes shifted the focus of programming towards "security through deviation"—completing a

straightforward task in a roundabout way—rather than taking a direct, unsafe approach that could cost the source code in readability. Böhm, Jacopini, and Dijkstra proved with their publications that no amount of computability would be lost with the structured paradigm; problems may take more code to solve or be less intuitive, but the result would be safer and easier to manage from a programming perspective. The PostgreSQL "Hello, World!" program above emphasizes this same way of thinking by allowing explicit indications of attributes like the programming language, the beginnings and end of the procedure, and variable declarations.

The final example in this section will show the "Hello, World!" program from ARM Assembly, a low-level set of instructions running the code that languages like C get compiled into:

```
.global main

message:
    .asciz "Hello, World!\n"
    .align 4

main:
    LDR R0, =message
    BL printf

    MOV R7, #1
    SWI 0
```

It is important to remember that all programs shown in this section perform the same action, with the only difference between each being the way they depict their instructions. Each language has their own compilers which execute a vastly different sequence of instructions but the end goal of each program is the same. With this in mind, it is easy to imagine with these simple examples that programming languages can be easily compared to one another; if a language can be replicated by another, then why bother learning more than one? Assembly is the ultimate programming language to answer this, as all software designed to be run on general-purpose hardware can be written in assembly. It is a superset comprising all other languages because of the inevitability of programs tracing back into assembly. Similar to how a proof that a language can simulate one Turing machine means it is equivalent in power to any other Turing machine created, any language that builds programs that run on general-purpose computers can be written in assembly. For example, Python, whose interpreter is written in C (which itself translates C source instructions into assembly) must therefore be fully representable within assembly; Python cannot introduce instructions that do not eventually translate into assembly because it would otherwise not be capable of running. Despite this observation, assembly provides an extreme example of why programming languages are diversified towards different tasks and syntaxes. The design of ARM Assembly is not meant to be entirely readable by humans in the same way Python is, so it would be terrible at representing problems pertaining to natural-language

processing or statistical analysis (two abstract problems which require high readability and understandability by end users). On the other hand, the breakneck speed computing capabilities are advancing at makes a case that languages like assembly should fall into disuse as general-purpose languages, left behind in favor of more readable languages like C. Proponents of this idea argue that any performance hit caused by using higher-level languages would be negligible compared to the boost in productivity gained and that assembly should only be used sparingly in performance-critical or storage-restrained environments, but this neglects a point mentioned previously: every language is built to address a purpose, and the problems any one language is built to solve cannot always be replicated in any other language.

Referring back to the ARM Assembly code above, what other programming language requires variables to specify alignment? Furthermore, the final two lines load an exit code into the `R7` register and invoke a system call to terminate the program; can this operation be sufficiently communicated with a C-styled `exit(0)` call? Maintaining our assumption that performance and libraries are ignorable and that syntax is the most important factor in deciding language use, ARM Assembly's most outstanding feature is its ability to communicate how hardware responds to instructions. The Python code shown before was utterly incapable of describing how a computer responds to the instruction, while the C code only provided a bit more insight on the import of an external library and the indication that newlines must be included for it to get displayed. While the ARM Assembly does include an external function `printf` whose definition is abstracted from a top-level perspective to the reader, it is explicit about the steps required to create a string, pass a parameter, and branch and link to the function. For many problems, this level of granularity would be unnecessary and detract from the solution, but problems that are close to the hardware (in which the alignment of strings and the branching to labels is important) would require the explicitness that ARM Assembly provides.

Functions and Procedures

Source code has a tendency to become repetitive. Abstraction is a tool that can be used to reduce repetition within code by grouping statements that achieve a similar purpose together and separating the statements that are unique to a given task, as well as combining a collection of statements under an alias for the benefit of the programmer to refer to it as. Although, the syntactical structure within popular languages like C and Java limit the level of compartmentalization that can be performed.

Procedural programming languages each have some realization of a sequence of code with a single entry point that performs a task given inputs. Of this sequence, "functions" define sequences of code that can be explicitly reduced to a value while "procedures" define sequences which cannot. This separation is purely semantic as procedures can modify

global values in different parts of a program or passed-in parameters yielding “side effects” but cannot be considered equivalent to a returnable value. Furthermore, procedures can return values as long as the resulting value is not considered to be semantically equal to the steps performed.

A simple function and procedure in C are given in Figure 22. The function is semantically reducible to the operation that transforms a series of inputs into an output while the procedure is not:

```c
// Function: returns the sum of the elements in an array.
int sum(int* arr, int length) {
    int total = 0;
    for (int index = 0; index < length; index++)
        total += arr[index];
    return total;
}

// Procedure: tests the capabilities of the sum function.
int test(int* arr, int length, int expected_sum) {
    int calc_sum = sum(arr, length);
    if (calc_sum != expected_sum) {
        printf("Error: sum incorrect\n");
        return 1;
    }
    else {
        printf("Passed: valid sum\n");
        return 0;
    }
}
```

Figure 22: Functions and procedures

Any location that `sum` is used in this program is equivalent to a for-loop iterating through each element of the parameter array performing a summation operation on the data. The result of using the `sum` keyword is semantically equivalent to the use of the sequence of statements. Furthermore, while the test procedure does return an error code denoting the success or failure of the steps being performed, the purpose of executing the procedure is not to generate an error code but rather to run a series of instructions. The procedure is not a transformation on data in the same way the function is; the procedure uses an identifier to easily represent the series of steps being performed, but the purpose of the procedure is not necessarily to yield new data.

Programming languages like Python have contributed to the difficulty in properly defining the differences between the terms. Python allows the user to define sequences of code associated with an invokable identifier, but Python will always return a value from invoked code regardless of the existence of an explicit `return` statement, shown in Figure 23:

```
1  def display_usage():
2      print('Usage: python3 prog.py <file.txt>')
3      print('Params:')
4      print('  <file.txt>: (str) Input file containing a list of English words')
5
6  # ...
7
8  result = display_usage()
9  print(result) # "None"
```

Figure 23: Pythonic functions

The `display_usage` identifier is associated with a sequence of print statements, but no return value was specified. Line 8 stores the result of the function which is automatically set to `None` instead of throwing a syntax error like C or Java would do. This explains why our previous definition of a "procedure" allowed return values like error codes and side effects and was more of a human-defined concept rather than intrinsic explanation of syntax-defined operative steps. There are many more situations in which these definitions may be tested (for instance, functions which invoke procedures, or functions which are determinant on global flags that decide conditional branches), so the separation between function and procedure needs to be left intentionally limited.

Since a function is a natural extension of a transformation on input data, programming languages which are built entirely upon transformations are called "functional". These languages evolved from formal mathematical systems like lambda calculus, as a means of expressing nonlinear translations and reductions on branching trees of code [28], and general recursive functions, which perform operations on tuples of natural numbers to return a single natural number [29]. Lisp, one of the more popular functional programming languages, was based around recursive functions and is characterized by branching collections of operations used to modify linked-lists of data, where the code itself is even organized as a nested list. Figure 24 demonstrates a Lisp implementation of Project Euler's problem 1 [30], which finds the sum of all multiples of 3 and 5 within an upper bound. In the graphical depiction, nodes that introduce right angles represent functions to be reduced, while nodes that are in sequence are statements that should be run nominally.

```
1  (defun mulsum (num)
2      (let ((sum 0))
3      (loop for i from 1 to (- num 1)
4          do (if (or
5              (zerop (mod i 3))
6              (zerop (mod i 5)))
7                  (incf sum i)))
8      sum))
9  (print (mulsum 1000))
```

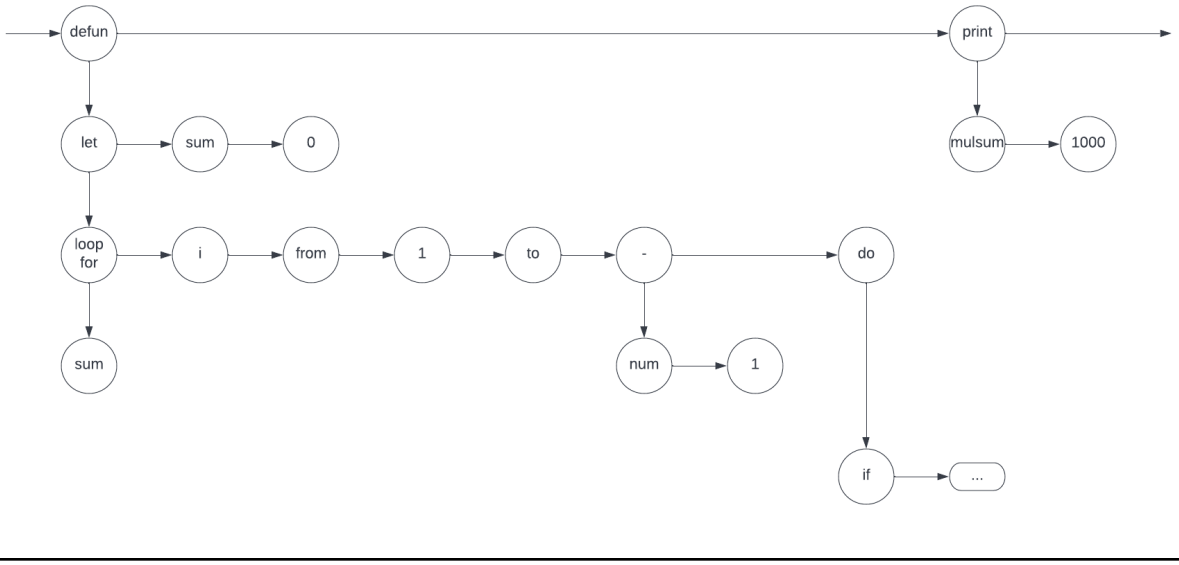

Figure 24: Lisp as lists

Depicting code in this manner blurs the distinction between function and procedure even further. Nodes which are “reduced” must be evaluated before the nodes containing them can be executed, so the result of a “reduction” can be seen as the returned value from a function. Therefore, the subtraction operator node in the figure can be seen as a traditional function, representing a transformation on two input parameters, but the “loop for” node does not return any standard result making it fit the definition of a procedure. Despite this, all values in our figure take the same super form as a node, where nodes can all be evaluated to return data (the evaluation of a number simply returns itself, while the evaluation of a more complex identifier like `mulsum` depends on external data structures like maps to correlate keywords with nodes). With this new visualization, it is possible to view the “loop for” node as simply another transformation on data; its returned value is just the culmination of a sequence of subnodes. Under this distinction, it can be argued that functional languages no not contain any procedures, but both viewpoints are defensible and matter only on the viewpoint of the subject due to the human-defined nature of the terms.

Coroutines

While the dichotomy between function and procedure does not matter for any reason besides visualization of code, the functional perspective that all syntactic objects are functions introduces a new important coding style. Functional code fits especially well within the graph-like structure shown previously, but Figure 25 shows how procedural code can be represented this way as well. Making a distinction between nodes which are

reducible to a value (functions, circles) and nodes which are not (procedures, squares) can more accurately mirror the operation of a language like C.

```c
int mulsum(int num) {
    int sum = 0;
    for (int i = 1; i < num; i++)
        if (i % 3 == 0 || i % 5 == 0)
            sum += i;
    return sum;
}

int main() {
    int sum = mulsum(1000);
    printf("%d\n", sum);
}
```

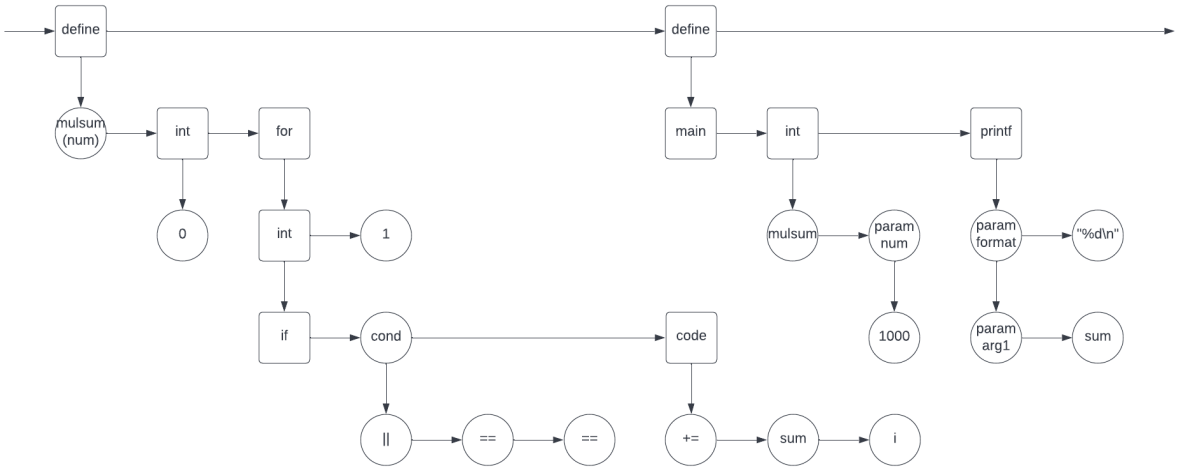

Figure 25: C as a graph (shortened)

Graphs used to represent code in this way also have the ability to introduce new control-flow structures like the coroutine, which is a special procedure that has a single initial entry point but multiple re-entry points that it can return to at different parts in the program. An example of a coroutine implemented in C# can be shown in Figure 26 along with its statement graph. In practice, coroutines are semantically identical to a procedure which maintains its state; variables within its scope are saved on exit and reloaded upon entry, and yield statements are marked with labels, stored within variables, and branched to upon re-entry of the procedure. Notice that the coroutine in the statement graph has multiple entry points as well as exactly two continuation points (where the procedure either exits the entire function when yield is met, or continues to the next statement if the procedure was invoked just prior).

```
1  static IEnumerator CountDown(int num) {
2      Console.Write("Begin ");
3      while (num > 0) {
4          yield return null; // "yield" exits and returns to this point later
5          Console.Write(num + " ");
6          num--;
7      }
8      Console.WriteLine("End");
9  }
10
11 public static void Main(string[] args) {
12     IEnumerator coroutine = CountDown(3);
13     coroutine.MoveNext(); // "Begin"
14     coroutine.MoveNext(); // "3 "
15     coroutine.MoveNext(); // "2 "
16     coroutine.MoveNext(); // "1 End"
17 }
```

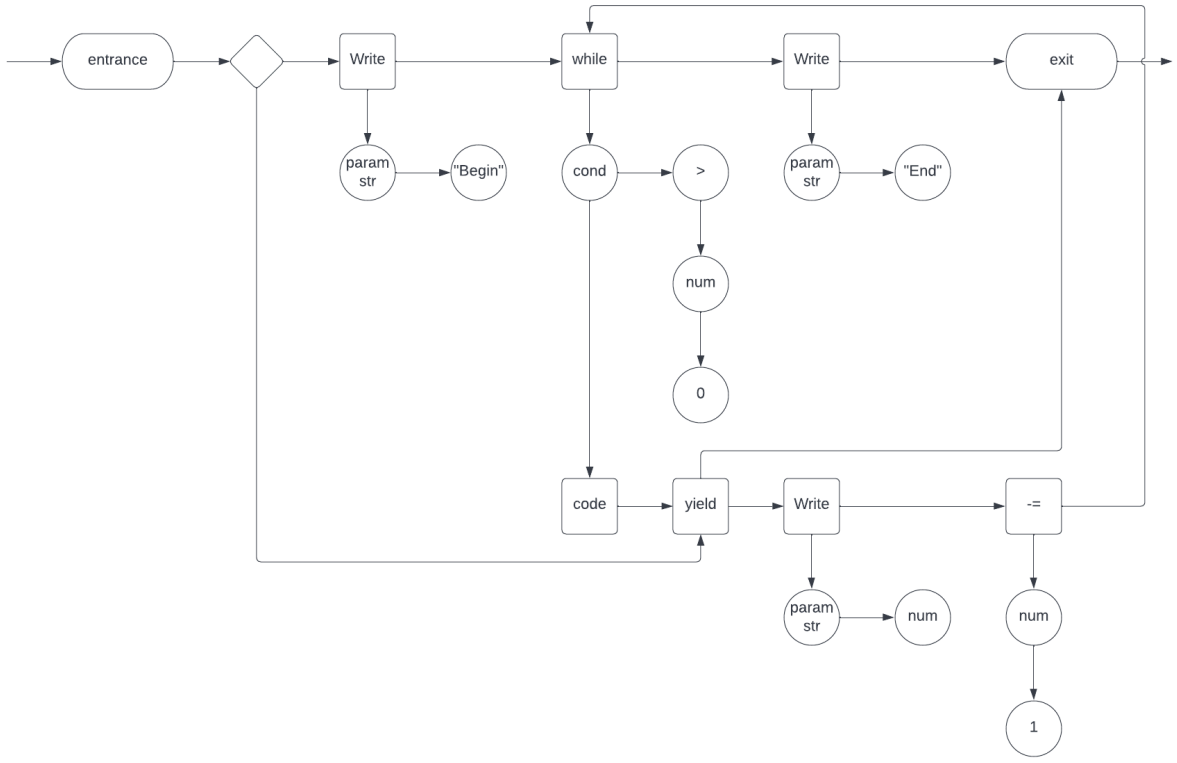

Figure 26: C# coroutine with added control-flow

This coroutine example introduces an important observation; if a coroutine is simply a collection of code with labels that are branched upon re-entry, then a for-loop can be considered to be an anonymous coroutine. C-styled for-loops have an “initial” starting point (the initialization statement), code that gets executed, and a condition that triggers either a natural exit of the loop or the deviation to a yield statement. The compiler then would

either re-invoke the anonymous loop procedure if it was yielded, or would continue to the next statement if the loop naturally exited. Figure 27 shows this process in full, with the same pattern of multiple entry points and exactly two continuation points as before. The compiler is implied to re-enter the anonymous coroutine immediately after it exits if the return value of `cond` is false.

```
1  for (int i = 0; i < 3; i++) {
2      printf("%d\n", i);
3  }
```

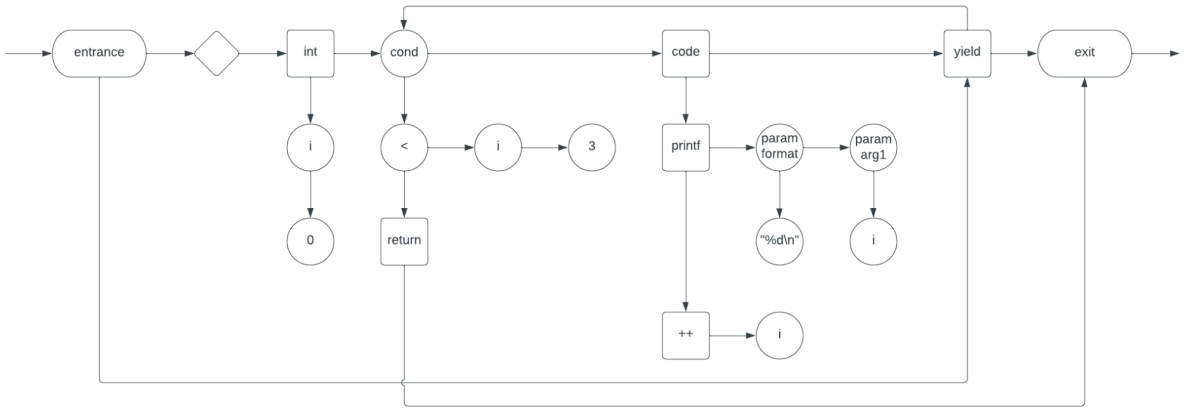

Figure 27: For-loop as an anonymous coroutine

Depicting programs as statement graphs has the tendency to get convoluted quickly, but it presents an elegant way of imagining the control flow of the program. The path that the program would take while imagining the loop as an anonymous coroutine seems wasteful at first as it involves several more branches than are necessary, but it will help to contextualize other code written with a similar mindset. In any case, consistently being capable of representing complex processes with abstract mentalities like this can lead to new programming ideas to be created.

JGPL

Statement graphs are one of many different ways of depicting programming languages, but the versatility of the graph allows us to imagine a new way of visualizing source code. Most procedural languages are very consistent in their ability to represent unified collections of statements, where brackets, colons and indentation, or other syntactic patterns collect sequences of code; invokable identifiers point to predetermined sequences (in the case of procedures and functions); and operators are used as abstract representations for identifiers (with operator overloading). In regards to invokable identifiers, the range of characters allowed to create these identifiers in C-like languages is relatively limited, only

using alphanumeric characters and underscores. This reduces the semantic meaning of a collection of statements into a single keyword for the benefit of concision but oftentimes at the cost of real-world equivalency. For example, the Python program below reads a file containing a collection of lists of numbers, where each line of the file has between 1 and 10 integer values, and finds the sum of each number within the file:

```
data = open('file.txt')
total = 0
for line in data.readlines():
    for num in line.split():
        total += int(num)
data.close()
```

Python was already noted as being one of the more English-aligned programming languages, whose design favored programming problems which could be describable using plain English, but there are still some flaws with the code given. For instance, the program features a very common coding pattern of opening a file, reading every line from it, and closing the file; this could be partly replicated with Python's `with` block to simplify the opening and closing, but it does not offer a way of reading the lines of the file as well. A function would not be a valid substitute for this problem because the problem requires a predefined startup and closing action to be performed with a user-defined intermediate action (the code executed during the reading of the file), and functions can only offer startup actions. Another problem with this code segment is the use of the identifier `split`. While familiar to other programmers, it does not make the code sound more natural when spoken aloud. If the goal of this exercise is generating pure-English code, then a better alternative would be to replace the line with the phrase "for each number within the line separated by spaces". As the pythonic syntax is limited in its ability to represent phrases like this, it opts instead to prioritize consistency by using the space character as a separator between syntactic units. I created a new programming language, named JGPL, in order to address these concerns.

JGPL was built on the basis of recognizing the inherent repetition within low-level source code. As mentioned in an earlier section, assembly languages are capable of representing code made in any other language meant to run on general-purpose hardware. Therefore, allowing programmers to write code directly with assembly would allow them to express virtually any idea they may have. Although, normal assembly programming languages intentionally limit the level of control programmers have in order to both optimize performance above problem solving and represent problems closer to hardware. Similar to how C has no concept of lists in the same way that Python does, ARM Assembly has no concept of a for-loop, requiring programmers to instead implement them manually with labels and conditional branches. The purpose of assembly languages is typically to generate code that is easily readable by the machine, but JGPL expresses a new idea of assembly that is built for the benefit of expressing solutions to problems.

JGPL separates distinct statements between lines, where each new line of code represents a new statement. Nearly everything in the language is a function or procedure, with the code that is not being clearly identifiable with a leading tilde. The simplest program you can make is a print statement made up of only intermediate code:

```
~PRINT "Hello, World!"
```

JGPL has 25 intermediate instructions that source programs get translated into, including assignment, arithmetic, branching, and object management. An interpreter reads these intermediate instructions at runtime to execute the program, but the user is not limited to only writing in intermediate code. Functions and procedures can be created through the use of the `func` statement, grouping together indented collections of code underneath an alias. To demonstrate the power of this concept, basic human-readable arithmetic and assignment statements are not a builtin feature of JGPL and must be implemented manually, as shown in Figure 28.

```
1  func <type var_type> <identifier id> = <value val>:
2      ~INSERT @id, @var_type
3      ~ASSIGN @id, @val
4
5  func <int op1> + <int op2>: int
6      int temp
7      ~IADD temp, op1, op2
8      return temp
9
10 main:
11     int a = 5       // Invokes function 1
12     int b = 7       // Invokes function 1
13     int c = (a + b) // Invokes function 2, then function 1
```

Figure 28: JGPL function declarations

There are only three builtin statements that do not begin with a tilde: `func`, which creates a new procedure or function, `block`, which describes an indented block of code, and `return`, which replaces a branched function with a value at runtime. Everything else, including variable declarations, loops, and complex data structures, are user defined. Asperands before variable identifiers when used in intermediate code signify redirection and is often used alongside parameters to inform the compiler that a variable holds the name or pointer to another variable. While C-like languages disallow spaces in invokable code identifiers, JGPL embraces it by considering arbitrary collections of tokens to be considered the identifier for a procedure or function. After the compilation of line 1, any instance of an assignment statement replaced with a branch to line 2.

Due to the potential complexity of determining which production rule to reduce to, JGPL follows strict requirements to minimize ambiguity. The following code can be seen as ambiguous under normal circumstances:

```
1  func display <value val>:
2          ~PRINT @val
3
4  func <int op1> + 5:
5          display "a"
6
7  func <int op1> + <int op2>:
8          display "b"
9
10 main:
11         display (1 + 5)
```

Figure 29: Ambiguity in JGPL

Both functions defined on lines 4 and 7 could apply to the invocation on line 11. To resolve ambiguity, JGPL chooses reductions based on the following rules:

1. Branches of reductions that are shorter are preferred over those that are longer. In the above example, line 11 matches with the function on line 4 because only one reduction is necessary.
2. Parentheses group tokens together underneath a value. Parentheses are reduced to functions before the procedures that contain them are reduced. If a token is encountered without being seen with an accompanied parenthesis, then the token is assumed to represent the entire value, and no further reduction will be performed with that token.

These restrictions increase the number of parenthesis into the program like Lisp, but it benefits in ease of computation and readability to an extent. By predefining a collection of default functions into a “standard library”, we can create programs that employ traditional syntactic structures familiar to C-like languages. A more complicated program is given below which demonstrates object-oriented design with lists:

```
1  func <list ls> [ <int index> ] = <value result>:
2      ~ATTRIBUTE @ls, @index, @result
3
4  func [ <int a> , <int b> , <int c> ]: list
5      list ls
6      ~OBJECT @ls
7      ~ATTRIBUTE @ls, size, 3
8      ls[0] = a
9      ls[1] = b
10     ls[2] = c
```

```
11      return l
12
13  func <list x> · <list y>: int
14      int result = 0
15      result += x[0] * y[0]
16      result += x[1] * y[1]
17      result += x[2] * y[2]
18      return result
19
20  main:
21      list listA = [1, 4, 6]
22      list listB = [8, 9, 2]
23      display (listA · listB) // 56
```

Figure 29: Dot products with lists

Objects within JGPL follow a similar convention as Python, where an object is simply an identifier connected to a dictionary. Objects are assignable with the `~OBJECT` command, which creates a new dictionary and stores it within the identifier (line 6), and attributes are assignable within the object with the `~ATTRIBUTE` command (line 7). Another benefit of the relaxed method of procedure definition is the usage of nonstandard characters used to represent procedures, as shown on lines 13 and 23, which use the mathematical dot operator to represent a dot product operation in a more standard way than the English word "dot" would. In this sense, the procedure can be imagined as an operator overload in a way comparable to C++ or C#.

In C-like languages, blocks are bounded areas of scope that traditionally execute code on entry or iteration alongside the code enclosed within. A big limitation of these languages is their inability to allow the user to define their own block. An area this would be useful is in implementation of C#'s `using` block, which ensures the correct initialization and disposal of an object alongside limiting its use within a defined scope, shown below:

```
using (StreamReader sr = new StreamReader("file.txt")) {
    while (sr.Peek() >= 0)
        Console.WriteLine(sr.ReadLine());
}
```

The usefulness of the statement is small as the only explicit statement the block negates the need of is an invocation of `IDisposable`'s `Dispose()` method. Despite this, the benefit it offers is semantic, as the programmer can be confident that the variable declared in the header does not come accompanied with any resource leaks immediately after the block. A language which did not feature a block similar to C#'s would not be incapable of solving any sort of unique file-related problem, but it would be negatively affected by being incapable of expressing file-like patterns in as elegant of a way. JGPL attempts to address this issue by making blocks a new type of procedure by utilizing the same statement-graph viewpoint

mentioned earlier. After following the design patterns mentioned earlier, if-statements and for-loops are not automatically defined within the program and can instead be defined by the user:

```
1  func <int var1> ge <int var2>: bool
2      int temp
3      ~GE temp, var1, var2
4      return temp
5
6  func <int var1> lt <int var2>: bool
7      int temp
8      ~LT temp, var1, var2
9      return temp
10
11 block if <bool b>:
12     ~BREQ b, 0, if_end
13     ~EXCON
14     ~LABEL if_end
15
16 main:
17     display "Type a number: "
18     int i = (int input)
19     if (i ge 0):
20         display i
21         display " is a positive number!\n"
22     if (i lt 0):
23         display i
24         display " is a negative number!\n"
```

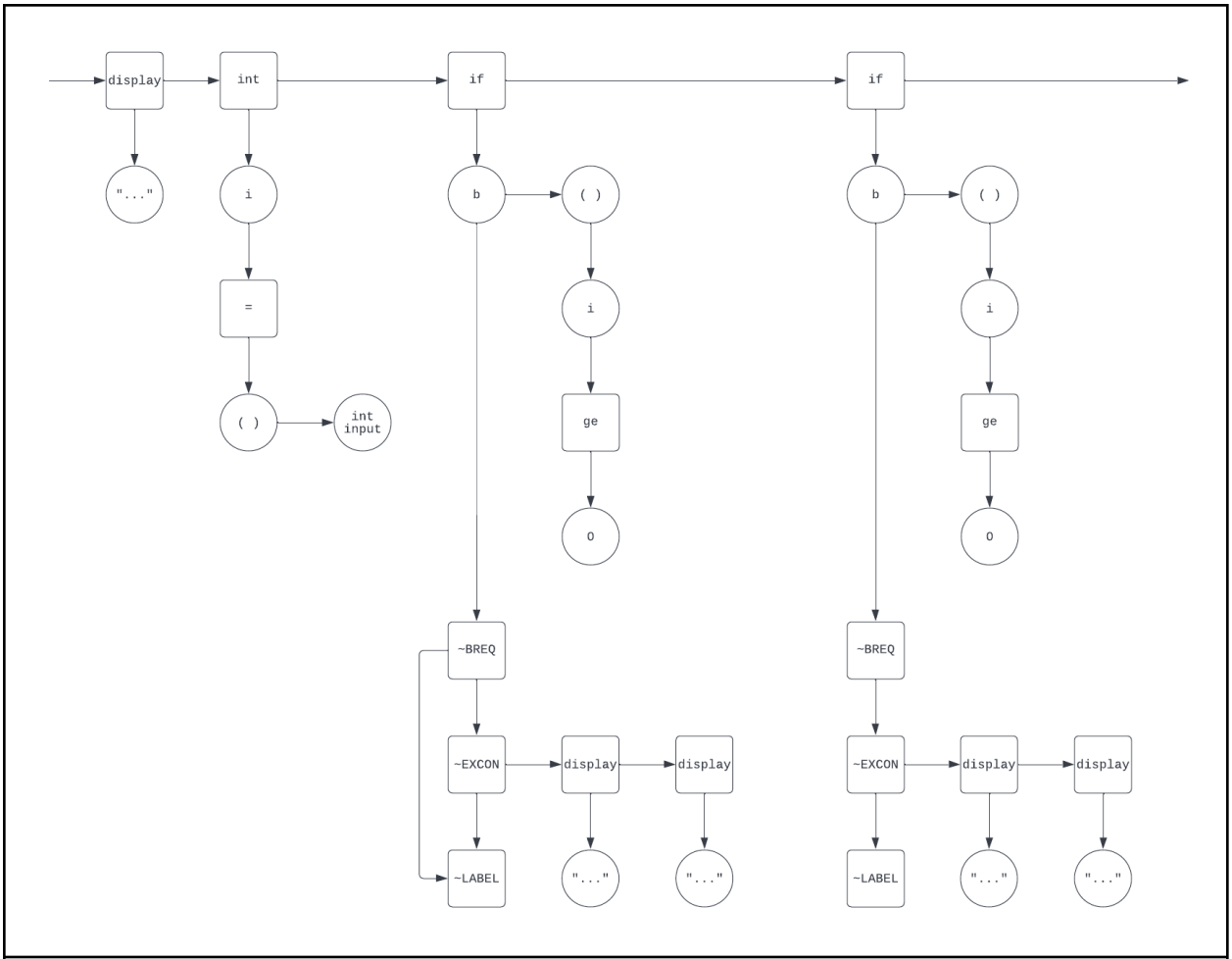

Figure 30: User-defined if-statements

The nodes in the statement graph are organized sequentially such that the nodes which feature the least number of reductions (or are further down the branches of the tree) are executed first. A few new intermediate codes are introduced with this example, including `~LABEL` (which creates a local label that can be branched to), `~BREQ` (which branches to a local label if two supplied values are equivalent), and `~EXCON` (which executes the contents associated with a block). The ability of an arbitrary procedure to execute nested code with a simple `~EXCON` call is comparable to the description of a coroutine given earlier; the code similarly contains a single entry point based on the graph, has conditional branching that optionally triggers nested labels or executions of block contents, and can exit with return statements. Extending upon this idea can create for-loops with more specific English-sounding wording in Figure 31, solving the problem that Python introduced earlier (line 17). The loop stores an identifier within a parameter instead of the value the identifier contains, essentially using pointer indirection to set the value of the index variable from a different scope.

```
block increase <identifier for_id> from <int start> to <int end> by <int incr>:
    ~ASSIGN @for_id, @start
    ~LABEL start_loop
    ~BRGE @for_id, end, end_loop
    ~EXCON
    ~IADD @for_id, @for_id, @incr
    ~BR start_loop
    ~LABEL end_loop

main:
    display "Enter the starting value: "      // 3
    int start = (int input)
    display "Enter the ending value: "        // 15
    int end = (int input)
    display "Enter the increment: "           // 2
    int incr = (int input)
    increase index from start to end by incr: // 3 5 7 9 11 13
        display index
        display " "
    display "\n"
```

Figure 31: For-loop implementation

Since typed assignment statements are not builtin to JGPL, the language is dynamically typed in a similar sense that Python is. This helps to resolve runtime ambiguity in statements where the only difference in determining the correct reduction in a variable is its type, but compile-time ambiguity is easily solved using cast functions:

```
func <int value>: string
    return value
```

Cast functions simply interpret a value of one type as another type. The function above signals to the compiler that all integers can be converted into strings with a single reduction; since the concept of reductions are used again, the language can use the same two rules in determining functions to choose from as stated before. Logic can be added to the function if required, but hierarchical abstraction can easily be implemented with only a return statement. For example, since integers and strings can both be considered subtypes of the supertype “value”, then the following productions can be generated to simplify repetitive code:

```
func int: type
    return "int"

func string: type
    return "string"

func <int value>: value
    return value
```

```
 9
10 func <string value>: value
11     return value
12
13 func <id type> <id identifier> = <value val>:
14     ~INSERT identifier, type
15     ~ASSIGN identifier, val
16
17 main:
18     int var1 = 5
19     string var2 = "Hello, World!\n"
```

Figure 32: Types

The JGPL project itself consists of a compiler and interpreter, which are both developed in Python and uploaded to GitHub [30]. The compiler first splits a source file into tokens given the simple syntactic rules JGPL initially defines, with tokens classified as either an `id`, `number`, `terminal` (non-alphabetic symbols), `indent`, `newline`, or `string` (text between quotes). Tokens are collected together into nested groups based on parentheses, statements based on newlines, and blocks based on indents. Then, the compiler traverses through each block starting at the top-level block through each sub-block in a tree-like path, identifying production definitions that are contained within each block. The compiler matches tokens with their respective productions, building a tree for each statement and reducing the leaves of the tree first. In order to simplify the production-identification process, the root of the tree will always be a procedure (a production that does not return a value) while non-roots will always be functions (a production that does return a value).

Once each statement has an identifiable production it can point to, it reiterates through the program and writes productions into an intermediate code file. These files must only contain intermediate code (which begin with a tilde "~" in normal source files), and the compiler accomplishes this by recursively iterating through productions and storing them as a collection of intermediate codes. Reductions of productions (in other words, function invocations) are converted into `ASSIGN`'s (assignments of parameters) and `FUNC`'s (which serve a multi-purpose of branching to a function pointer while keeping its previously-executed code index stored on a stack and putting the production's return value in a variable). Finally, the compiler writes each intermediate code into a compiled output file. The interpreter is a separate program which simply locates the `main` label (manually defined by the user to be the starting point of the code) and executes intermediate codes until the end of the file is reached, keeping track of the currently-executed code as an index (program counter) into a linear array of codes.

Syntax Variability

One of the benefits of writing code in a language with flexible syntax is its ability to adhere to problems. In JGPL, syntax styles can be easily imported in the same way an external module could be, allowing select source files to adopt design patterns specific to the task it needs to solve, similar in practice to how importing C#'s LINQ library allows programmers to use query-based retrieval of enumerable data using standard C# method-based syntax as well as SQL-like syntax. For example, consider a program that requires the usage of sets in Figure 33.

```
// set.jg: include the file into the program by passing it as a parameter to the
compiler. Commented ellipses "..." represents code that was hidden from the figure for
space.

// Returns an empty set, ready to be added to
func new set: set
    // ...

// Returns the null set, which is a special empty set
func null set: set
    // ...

// Returns the universal set containing every value
func universal set: set
    // ...

// Inserts the value into the set
func insert <value val> into <set storage>:
    // ...

// Returns a set that contains elements from both sets
func <set op1> union <set op2>: set
    // ...

// Returns a set that only contains elements found in both sets.
func <set op1> intersect <set op2>: set
    // ...

// Returns true if the value exists within the set, false otherwise
func <value val> belongs to <set op>: bool
    // ...
```

```
// main.jg

main:
    set new_set = (new set)
    set uni_set = (universal set)
    set nul_set = (null set)

```

```
    insert "a" into new_set
    insert "b" into new_set

    display (new_set intersect uni_set) // "{a, b}"
    display (uni_set union nul_set)     // "universal set"
    if ("a" belongs to new_set):
        display "True!"                 // "True!"
```

Figure 33: Set usage with standard alphabetic syntax

The code shown in `main.jg` depicts a style familiar to programmers of object-oriented languages, but the code would seem improper to mathematicians or computer science researchers studying set theory. Sets come equipped with their own special notation for representing special sets and operations between sets, but JGPL's representation of procedures can allow this notation to be possible as shown in Figure 34.

```
// set.jg

// Returns an empty set, ready to be added to
func new set: set
    // ...

// Returns the null set, which is a special empty set
func ∅: set
    // ...

// Returns the universal set containing every value
func μ: set
    // ...

// Inserts the value into the set
func insert <value val> into <set storage>:
    // ...

// Returns a set that contains elements from both sets
func <set op1> U <set op2>: set
    // ...

// Returns a set that only contains elements found in both sets.
func <set op1> ∩ <set op2>: set
    // ...

// Returns true if the value exists within the set, false otherwise
func <value val> ∈ <set op>: bool
    // ...
```

```
// main.jg

main:
    set new_set = (new set)
```

```
 5     set uni_set = μ
 6     set nul_set = ∅
 7
 8     insert "a" into new_set
 9     insert "b" into new_set
10
11     display (new_set ∩ uni_set) // "{a, b}"
12     display (uni_set ∪ nul_set) // "μ"
13     if ("a" ∈ new_set):
14         display "True!"         // "True!"
```

Figure 34: Set usage with special notation

With the changes made, the code within main.jg describes the problem it was created to solve a little better. These small notational differences reflect a core idea this paper was created to address: programmers should strive to seek the best ways of representing solutions to niche problems. A simple import of an external file can introduce better ways of denoting solutions to problems; the set notation is a more natural expression of set theory that regular English words cannot properly describe. A lot of nuance is left out when domain-specific notation is removed from source code, and JGPL attempts to reintroduce this syntax.

Since JGPL's main feature is the reduction of syntax from user-defined production rules, abstraction is made easy by allowing the user to define their own program design contracts to follow. The next example shown in Figure 35 will demonstrate how a Python-like iterator for a list (defined in Figure 29) is created with the help of non-compiled documentation. First, the properties for the object to be iterated through is organized:

```
list properties:
       size: int
       <number>: value
```

Then, the abstract properties for the iterator is determined. The properties include attributes the iterator should contain as well as blocks/functions with return values the iterator should define. Since JGPL features symmetry in object design by making all objects be implemented with Python dictionaries, abstract properties do not need to be explicitly declared through source code. The properties an iterator iter would need to define at a minimum would be:

```
func(bool) [abstract]: <iter> is done
func(value) [abstract]: get next element from <iter>
block: for <identifier> in <iter>
```

Next, the concrete properties for a list iterator is determined. Each of these properties would need to be represented within source code, and they must fulfill the functionality outlined by the abstract iterator above.

```
collection: list
index: int
func(list_iter): iterate(<list>)
func(iter): <list_iter>
func(bool): <list_iter> is done
func(value): get next element from <list_iter>
```

Now that the requirements are outlined, the code can be created. Figure 35 shows an implementation of the list iterator and a demonstration of it working.

```
1  // iter.jg
2
3  // Creates an iterator from the list
4  func iterate { <list collection> }: list_iter
5      list_iter iterator = (create object)
6      set collection in iterator to collection
7      set index in iterator to 0
8      return iterator
9
10 // list_iters are convertible to abstract iterators
11 func <list_iter iterator>: iter
12     return iterator
13
14 // Returns true if the iterator has completed, false otherwise.
15 func <list_iter iterator> is done: bool
16     collection = (get collection from iterator)
17     return ((get size from collection) eq (get index from iterator))
18
19 // Returns the next element from the iterator and advances the iterator forward.
20 func get next element from <list_iter iterator>: value
21     // Get attributes from iterator
22     collection = (get collection from iterator)
23     index = (get index from iterator)
24
25     // Retrieve the item at the position
26     item = (collection[index])
27
28     // Advance the index
29     set index in iterator to (index + 1)
30     return item
31
32 // Iterates through each element within an abstract iterator.
33 block for <identifier id> in <iter iterator>:
34     ~LABEL start_loop
35
36     if (iterator is done):
37         ~BR end_loop
38
39     value = (get next element from iterator)
40     ~ASSIGN @id, value
41     ~EXCON
42     ~BR start_loop
43
```

```
44      ~LABEL end_loop
```

```
1  // main.jg
2
3  // Displays "582"
4  main:
5      list values = [5, 8, 2]
6      for value in (iterate{values}):
7          display value
```

Figure 35: Set usage with special notation

As a final demonstration of the versatility of JGPL, the query syntax of SQL will be implemented allowing programmers to use SQL-like `SELECT` and `INSERT` statements. The following examples show the creation process for developing the extension.

First, the requirements for the extension are outlined. At a minimum, the code should be capable of retrieving and filtering records from a table by using SQL-like statements, outlined below:

```
CREATE TABLE <name> (name type, name type, ...)
INSERT INTO <table> VALUES (name type, name type, ...)
SELECT <schema> FROM <table> WHERE <condition>
```

Parameters are identifiable as triangle brackets which identify types that need to be implemented. Additionally, the syntax of the statements need to be slightly adjusted to account for reserved language symbols (like parenthesis, which are used in reductions). Types are outlined next, with a determination if the type is new or an alias for a different type and functions and attributes defined with the type:

- `table_schema`: (alias, `list`) The names of each column of the table.
    - Must be capable of checking if one schema equals another.
- `record`: (alias, `map`) Connects each value in the schema with a specific value. The mapping allows us to create sub-records that apply to sub-schemas based on `SELECT` statements.
    - Should be capable of being reduced by a schema, where an input schema asks as a mask to remove certain items from the map.
- `record_comparison`: (new) The comparison to be performed on rows within the table. For this reduced SQL module, all comparisons take the form of “schema relation value”, where relation is an aliased type based off of a string and can be either “`gt`”, “`lt`”, or “`eq`”.
    - Needs to be able to check if a record_comparison instance applies to a given record.
- `table`: (new) The main storage class for tables. Essentially a schema along with a

list of records.

- Must implement each of the required SQL functions detailed before, including `CREATE`, `INSERT`, and `SELECT`. Additionally, the result of a `SELECT` should be a table itself, which will allow nested `SELECT` statements.

After a rough outline with requirements like this has been created, the source code can begin construction. Figure 36 shows the complete module implemented, assuming that submodules listed in previous figures containing useful code (like lists, loops, and conditionals) were imported previously:

```
// sql.jg

// ===== table_schema properties ===== //

func *: table_schema
    schema = (create list {size=1})
    schema[0] = "*"
    return schema

func <table_schema schema>: list
    return schema

func <list schema>: table_schema
    return schema

func <table_schema op1> equals <table_schema op2>: bool
    bool equivalent = false
    int size1 = (get size from op1)
    int size2 = (get size from op2)
    if (size1 eq size2):
        equivalent = true
        increase index from 0 to size1 by 1: // C-styled for-loop
            item1 = (op1[index])
            item2 = (op2[index])
            equivalent = (equivalent && (item1 eq item2))
    return equivalent

// ===== record properties ===== //

func reduce <record rec> by <table_schema schema>: record
    map reduced_record = (create map) // A map is a list with objects as keys
    for item in (iterate{schema}):
        if (item is in rec):
            reduced_record[item] = (rec[item])
    return reduced_record

// ===== record_comparison properties ===== //

func gt: relation
```

```
40      return "gt"
41
42  func lt: relation
43      return "lt"
44
45  func eq: relation
46      return "eq"
47
48  func create comparison                      // Long functions
49       {id = <identifier id>,                 // can be extended
50       operator = <relation operator>,        // across lines
51       value = <value val>}: record_comparison
52
53      record_comparison obj = (create object)
54      set id in obj to id
55      set operator in obj to operator
56      set val in obj to val
57      return obj
58
59  func <identifier id> <relation operator> <value val>: record_comparison
60      return (create comparison {id=id, operator=operator, value=value})
61
62  func <record_comparison cmp> applies to <record rec>: bool
63      identifier id = (get id from cmp)
64      bool result = false
65      if (id is in rec):
66          value record_value = (rec[id])
67          value comparison_value = (get val from cmp)
68          relation rel = (get operator from cmp)
69          if (rel eq "gt"):
70              result = (record_val gt comparison_value)
71          if (rel eq "lt"):
72              result = (record_val lt comparison_value)
73          if (rel eq "eq"):
74              result = (record_val eq comparison_value)
75      return result
76
77  // ===== table properties ===== //
78
79  func CREATE TABLE { SCHEMA = <table_schema schema> }: table
80      table tbl = (create object)
81      set schema in tbl to schema
82      set records in tbl to (create list {size=0})
83      return tbl
84
85  func INSERT INTO <table tbl>
86       {<string key1>, <string key2>, <string key3>}
87       VALUES {<value val1>, <value val2> <value val3>}:
88
89      // Similar to list creation with brackets [], maps can be created with
90      // curly braces {}
91      records = (get records from tbl)
```

```
    append ({key1=val1, key2=val2, key3=val3}) to records

func INSERT INTO <table tbl> VALUES <list value_list>:
    table_schema schema = (get schema from tbl)
    map value_map = (create map)
    increase index from 0 to (get size from value_list) by 1:
        value_map[ (schema[index]) ] = (value_list[index])

    records = (get records from tbl)
    append value_map to records

func iterate { <table tbl> }: list_iter
    records = (get records from tbl)
    return iterate{records}

func SELECT <table_schema schema>
     FROM <table tbl>
     WHERE <record_comparison comparison>: table

    list valid_records = (create list {size=0})
    for record in (iterate{tbl}):
        if (comparison applies to record):
            append record to valid_records // Appends and increases list size

    if (schema ne *):
        int size = (get size from valid_records)
        list reduced_records = (create list{size=size})
        for record in (iterate{valid_records}):
            masked_record = (reduce record by schema)
            append masked_record to reduced_records
        valid_records = reduced_records
    if (schema eq *):
        schema = (get schema from tbl)

    table new_tbl = (create object)
    set schema in new_tbl to schema
    set records in new_tbl to valid_records
    return new_tbl

func display <table tbl>:
    display (get schema from tbl)
    for record in (iterate{tbl}):
        display record
```

```
// main.jg

main:
    table tbl = (CREATE TABLE {SCHEMA = (["name", "age", "gender"])})
    INSERT INTO tbl VALUES (["Luz", 14, "F"])
    INSERT INTO tbl VALUES (["Hunter", 16, "M"])
    INSERT INTO tbl VALUES (["Gus", 12, "M"])

```

```
 9    display (SELECT * FROM tbl WHERE ("age" gt 13))
10        // ["name", "age", "gender"]
11        // ["Luz", 14, "F"]
12        // ["Hunter", 16, "M"]
13
14    display (SELECT (["name"]) FROM tbl WHERE ("gender" eq "M"))
15        // ["name"]
16        // ["Hunter"]
17        // ["Gus"]
18
19    display (SELECT (["age"]) FROM tbl WHERE ("name" eq "Sophie"))
20        // ["age"]
21
22    table subquery = (SELECT (["name", "age"]) FROM tbl WHERE ("gender" eq "M"))
23    display (SELECT (["name"]) FROM subquery WHERE ("age" lt 15))
24        // ["name"]
25        // ["Gus"]
```

Figure 35: SQL-like syntax module

Mass-retrieval of data from a database can be accomplished using simple iteration and condition-checking, but doing it instead in a structured way with queries can reduce the amount of boilerplate code that would need to be created. From a semantic standpoint, JGPL associates a `SELECT` statement with an iteration through a sequence of records and a reduction of records through the use of a schema as a mask. It further breaks down each of these concepts into subconcepts through a branching tree structure until each of the leaves of the tree are concrete statements definable in the intermediate instruction set. This SQL example is the culmination of the abstraction/symmetry concept mentioned earlier in the paper: keywords represent concepts rather than concretions, and concept reuse leads to creative expressions of code. The first-class nature of nodes within this tree perspective allows the program to interpret language reductions as definitions to abstract tokens, giving a meaning to an otherwise meaningless string of tokens.

SQL can be viewed as one of the best ways to represent data retrieval from a database. Furthermore, it is one of the more "English" languages used, to an even greater extent than Python is. The query on line 14 of main.jg is:

```
SELECT (["name"]) FROM tbl WHERE ("gender" eq "M")
```

Following the exercise in an earlier chapter with the sine Taylor series, the verbal description of the query is nearly identical to the code that is actually written: "it selects the `name` column of a row from `tbl` where the `gender` column equals `m`". SQL removes the boilerplate text in the spoken description (deleting redundant words like "column" and articles like "the" and "a") and streamlines the code to remove implied descriptors (like "of a row from", which is already implied since `SELECT` statements always execute on singular

rows at a time). Therefore, with a bit of work, the actual SQL code can be seen as a direct translation of spoken language.

JGPL capitalizes on this idea by recognizing that SQL is a better way of representing the retrieval of data from iterable objects based on rules, compared to regular procedural or object-oriented code. The pythonic approach to solving this problem would utilize lambda expressions and separated functions for each part of the query, which ends up extending the program size and adds unnecessary complexity to the code. Furthermore, the problem with pythonic keyword representation remains, where relatively nontrivial functionality (like the comparison between a column in the table with a constant value) needs to be represented with a single keyword, while JGPL instead represents it with multiple keywords contained within a parenthesis. JGPL is a distinct improvement on this genre of problem solving: it represents the problem with code that enhances its readability and understandability, allowing the user to decide how expressibility can be gained through syntax that improves their program design.

Conclusion

English and Spanish have more in common than they have differences, but is a perfect translation of any text guaranteed? Language mostly attempts to communicate thoughts, emotions, and events, and most of this is accomplished through the utilization of concrete concepts that have direct parallels to other languages. Even so, the means by which a language can convey a message is aided by symbolism, allusion, allegory, colloquialisms, euphemisms, and other abstract concepts that rely on decades of context for a reader to understand. Furthermore, poetic devices like onomatopoeia, rhyming, rhythm, and alliteration are distinct to a language scheme even without regard for culture, making subjects that are saturated with imagery difficult for non-native speakers to understand. Some poems may be boring, repetitive, or simplistic, but still convey powerful emotions unique to a time period or location, impossible to translate without history lessons and lectures on culture and art.

Programming languages lie in this same vein. A programming language is a medium for expressing a solution to a problem which may be steeped in context. A program that is built to implement an algorithm relies not only on data structures and basic syntactic features like iteration and functions, but it requires organization of individual tokens in a sequence that appears pleasing to the viewer. Everything about a language, including the amount of whitespace that programmers feel inclined to add at the start of each line to properly align segments of code, affect the way a solution is perceived. Too many complex concepts densely packed on a page can make a solution seem unnecessarily complicated (unless the solution requires complexity?) but the superfluous inclusion of overly simplistic formatting can drag out the program much longer than it needs to (unless the audience demands

thoroughness and explanatory code?). It is difficult to find the perfect balance of organization for a project, but each of these factors decidedly impact the success of the source code.

This paper browsed an overview of what makes programming an artform. While abstraction and elegancy are useful tools in creating effective programs in a short timespan, we also considered why they are good for communicating the solutions of problems they were meant to solve. JGPL's approach to problem solving is to adapt to the problem it needs to solve; programmers should not compromise by generalizing solutions across a consistent syntax because the same syntax cannot describe every problem. Programmers should encourage the flexibility of their programs and the adaptability of their design; everyday problems are not rigid, so neither should their solutions be.

References

[1] Malloy, B. A., & Power, J. F. (2017). Quantifying the transition from python 2 to 3: An empirical study of python applications. 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). https://doi.org/10.1109/esem.2017.45

[2] Järvi, J., & Freeman, J. (2008). C++ lambda expressions and closures. Science of Computer Programming, 75(9), 762–772. https://doi.org/10.1016/j.scico.2009.04.003

[3] Joisha, P. G., Kanhere, A., Banerjee, P., Shenoy, U. N., & Choudhary, A. (2000). Handling context-sensitive syntactic issues in the design of a front-end for a Matlab compiler. Proceedings of the International Conference on APL-Berlin-2000 Conference - APL '00. https://doi.org/10.1145/570475.969784

[4] Tratt, L. (2009). Chapter 5 dynamically typed languages. Advances in Computers, 77, 149–184. https://doi.org/10.1016/s0065-2458(09)01205-4

[5] Brunthaler, S. (2009). Virtual-machine abstraction and optimization techniques. Electronic Notes in Theoretical Computer Science, 253(5), 3–14. https://doi.org/10.1016/j.entcs.2009.11.011

[6] Nanz, S., & Furia, C. A. (2015). A comparative study of programming languages in Rosetta Code. 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. https://doi.org/10.1109/icse.2015.90

[7] Bissyande, T. F., Thung, F., Lo, D., Jiang, L., & Reveillere, L. (2013). Popularity, interoperability, and impact of programming languages in 100,000 Open source projects. 2013 IEEE 37th Annual Computer Software and Applications Conference. https://doi.org/10.1109/compsac.2013.55

[8] Farshidi, S., Jansen, S., & Deldar, M. (2021). A decision model for programming language ecosystem selection: Seven industry case studies. Information and Software Technology, 139, 106640. https://doi.org/10.1016/j.infsof.2021.106640

[9] Bijlsma, D., Ferreira, M. A., Luijten, B., & Visser, J. (2011). Faster issue resolution with higher technical quality of software. Software Quality Journal, 20(2), 265–285. https://doi.org/10.1007/s11219-011-9140-0
[10] Ritchie, D. (2003). The development of the C language*. Bell Labs. Retrieved December 15, 2022, from https://www.bell-labs.com/usr/dmr/www/chist.html
[11] Venners, B. (2003, January 13). The Making of Python | A Conversation with Guido van Rossum, Part I. artima. Retrieved December 15, 2022, from https://www.artima.com/articles/the-making-of-python
[12] Iverson, K. E. (1979). Notation as a tool of thought. ACM Turing Award Lectures, 1979. https://doi.org/10.1145/1283920.1283935
[13] Shaw, M. (1984). Abstraction techniques in modern programming languages. IEEE Software, 1(4), 10–26. https://doi.org/10.1109/ms.1984.229453
[14] Churchill, A., Biderman, S., & Herrick, A. (2019). Magic: The gathering is Turing complete. arXiv preprint arXiv:1904.09828.
[15] Grigore, R. (2017). Java generics are Turing complete. Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. https://doi.org/10.1145/3009837.3009871
[16] Oram, A. (2007). Beautiful code. O'Reilly.
[17] Prechelt, L. (2000). An empirical comparison of seven programming languages. Computer, 33(10), 23–29. https://doi.org/10.1109/2.876288
[18] Zhao, L. (2008). Patterns, symmetry, and symmetry breaking. *Communications of the ACM*, *51*(3), 40-46.
[19] ROSEN, J. (1989). Symmetry at the foundations of Science. Symmetry 2, 13–15. https://doi.org/10.1016/b978-0-08-037237-2.50007-6
[20] Reinsel, D., Gantz, J., & Rydning, J. (2018, November). Report - the digitization of the world - from edge to core. Platina. Retrieved December 15, 2022, from https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf
[21] Lestrozi. (2022, July 25). BF Pong. GitHub. Retrieved December 15, 2022, from https://github.com/lestrozi/pipevm/blob/main/tests/bfponggen.py
[22] Lestrozi. (2022, July 25). BF Unix Shell. GitHub. Retrieved December 15, 2022, from https://github.com/lestrozi/pipevm/blob/main/tests/sh.bf
[23] Temkin, D. (2017). Language without code: Intentionally unusable, uncomputable, or conceptual programming languages. Journal of Science and Technology of the Arts, 9(3), 83. https://doi.org/10.7559/citarj.v9i3.432
[24] Abelson, H., Sussman, G. J., & Sussman, J. (1985). Structure and interpretation of computer programs. The MIT Press.
[25] Using package members. Using Package Members (The Java™ Tutorials > Learning the Java Language > Packages). (n.d.). Retrieved December 15, 2022, from https://docs.oracle.com/javase/tutorial/java/package/usepkgs.html

[26] Böhm, C., & Jacopini, G. (1966). Flow diagrams, Turing machines and languages with only two formation rules. Communications of the ACM, 9(5), 366–371. https://doi.org/10.1145/355592.365646
[27] Dijkstra, E. W. (1968). A case against the GO TO statement. University of Texas at Austin. Retrieved December 15, 2022, from https://www.cs.utexas.edu/users/EWD/ewd02xx/EWD215.PDF
[28] Church, A. (1932). A set of postulates for the foundation of Logic. The Annals of Mathematics, 33(2), 346. https://doi.org/10.2307/1968337
[29] Dean, W. (2020, April 23). Recursive functions. Stanford Encyclopedia of Philosophy. Retrieved December 15, 2022, from https://plato.stanford.edu/entries/recursive-functions/
[30] https://github.com/justinmgarrigus/JGPL