Perl

Maximilian egger

Specialisation Seminar

Supervisor:
Prof. Dr. Cezary Kaliszyk
Department of Computer Science
University of Innsbruck

Innsbruck, July 15, 2023

Abstract

This paper presents a comprehensive examination of the Perl programming language. Starting with its inception in 1987 by Larry Wall, the paper traces the evolution of Perl into a powerful and versatile tool for text manipulation, system administration, and web development. The language's unique syntax, characterized by its expressive yet unconventional design, is explored, highlighting its ability to handle complex tasks with ease.

In addition to its syntax, the paper explores Perl's distinguishing features, including default variables and robust support for regular expressions. These features enable developers to efficiently process and manipulate text data, making Perl a favored choice for tasks requiring sophisticated pattern matching.

However, the paper also acknowledges certain limitations of Perl. The principle of "There's More Than One Way To Do It" (TIMTOWTDI) can lead to code readability challenges, especially for those unfamiliar with multiple coding styles. Furthermore, Perl's declining popularity compared to languages like Python and its interpretation overhead raise concerns about performance and industry adoption.

To provide a balanced view, the paper conducts a comparative analysis of Perl with C and Python. While Perl excels in certain text-processing tasks, C outperforms it in computationally intensive operations, and Python's simplicity and broad library support contribute to its widespread adoption.

In conclusion, this paper serves as a valuable resource for developers seeking to understand the strengths and weaknesses of Perl. By offering insights into its unique syntax, distinguishing features, and comparative performance, the paper equips readers to make informed decisions when choosing a programming language for their specific projects.

1 Introduction

The *Perl* programming language was first introduced in 1987 by Larry Wall, with the focus on the most important principle of language design: "To make easy jobs easy and hard things not impossible"[8, xvii]. The Perl language prioritizes practicality, emphasizing ease of use, efficiency, and comprehensive functionality over aesthetic qualities like minimalism or elegance[9]. Often called the "Practical Extraction and Report Language"[7, 5], Perl revolutionized the manipulation and extraction of information from text input, surpassing other available languages at the time. Its optimization for manipulating and scanning text files made it a go-to tool and is a key-factor in its continuing relevance today. Over the years, the open-source language has evolved with the support of its vibrant developer community, expanding its capabilities from simple scripts to building web services and large applications utilizing different programming paradigms such as *Object-Oriented Programming* and *Functional Programming.*

This paper tries to give the readers a brief overview of the Perl programming language. It begins by exploring the origins of Perl and continues to examine the evolution of

Perl into the versatile tool it has become today. Subsequently, a brief explanation of the Perl language is presented, emphasizing its unusual yet expressive syntax and rich, community-driven library ecosystem. The discussion proceeds to highlight the distinguishing features of Perl, including its expressiveness, usage of context, default variables and its adherence to the principle of offering multiple solutions to problems, strong text manipulation tools. However, the paper also addresses the limitations of Perl, including potential challenges in code maintenance and the learning curve associated with its syntax. Finally, a comparative analysis with Python and C is provided, offering insights into the unique strengths and weaknesses of each language and aiding developers in making informed choices for their projects.

2 History

Perl, an expressive and versatile programming language, has a storied history that spans over three decades. Developed in the late 1980s by Larry Wall, Perl emerged as a response to the limitations of existing Unix scripting languages. Wall's aim was to design a language that prioritized practicality, focusing on enabling developers to accomplish both simple and complex tasks effectively[8, xvii–xx].

Wall identified a specific problem prevalent among system administrators at the time – the lack of scripting languages with robust text processing and manipulation capabilities[1][7, 5].
Leveraging his background in linguistics, he sought to create a language that excelled in these areas[2, 3]. Wall's innovation provided a powerful solution for system administrators, enabling them to effortlessly extract and manipulate information from text files[7, 8]. The impact of Perl was significant, rapidly gaining popularity within the Unix user community. Its prowess in text processing made it a preferred tool for log file analysis, report generation, and data extraction tasks. Beyond system administration, Perl found applications in diverse fields such as web development and bioinformatics[1].

Over the years, Perl underwent iterative evolution, marked by major releases that introduced new features and enhancements. The language garnered recognition for its expressive syntax, extensive library ecosystem, and its adherence to the principle of offering multiple solutions to problems. Perl's flexibility and robust text manipulation capabilities played pivotal roles in its continuous growth and adoption[7, 6-8][2, 4-5].
Critical to Perl's evolution and continued relevance was its active and dedicated community of developers. This community contributed to the language's expansion, developing numerous modules and libraries that extended its functionality. The spirit of collaboration and knowledge sharing within the Perl community fostered a dynamic en-

[1]Larry Wall interview — Linux Journal `https://www.linuxjournal.com/article/3394`, visited 09.06.2023

vironment, facilitating the language's ongoing development and adaptation[2]. Perl has remained resilient, embracing modern programming paradigms such as Object-Oriented Programming and Functional Programming, while retaining its core strengths in text processing[2, 141-162][6, 272-307].

Its enduring impact is a testament to Larry Wall's visionary approach and the collective efforts of the Perl community. Evidence of Perl's continued usage can be found in the statistics of contributions to the Comprehensive Perl Archive Network (CPAN). While the numbers may be decreasing slowly, the fact that developers continue to contribute to CPAN showcases the enduring relevance of Perl as a programming language. Perl remains a valuable resource, offering a vast collection of modules and libraries that are actively maintained and utilized by developers today[3].

3 The Perl Language

Perl distinguishes itself from other programming languages by adopting a design that mirrors human communication rather than being rooted in strict mathematical concepts. Its syntax, while unique and often considered unconventional, offers powerful features and flexibility.
In this chapter, readers will be given a short summary of the Perl syntax and some of its unique characteristics and distinctions from other programming languages many are familiar with.

3.1 Variable Names and Sigils

In Perl, variable names always have a leading *sigil*, which indicate their container type. *Scalar variables*, representing single values, use the dollar sign ($) sigil. *Array or List variables*, use the at sign (@) sigil. Hash variables, representing key-value pairs, use the percent sign (%) sigil.

```
my $scalar = 42;
my @list = (1, 2, 3);
my %hash = ('key1', 'value1', 'key2', 'value2');
```

[2]Comprehensive Perl Archive Network: http://www.cpan.org/misc/cpan-faq.html#How_does_the_CPAN_work, visited 09.06.2023

[3]CPAN Tester Statistics:https://stats.cpantesters.org/statscpan.html#milestones, visited 09.06.2023

3.2 Context

In Perl, the concept of context refers to how values or expressions are evaluated and used in different situations. The context influences the behavior and interpretation of these values. Perl has two primary contexts: scalar context and list context.
It is essential to understand the concept of context in Perl to write effective and accurate code.

Scalar Context:
In scalar context, a variable or expression is evaluated to produce a single scalar value. For example, when assigning an array to a scalar variable, only the last element of the array is stored in the scalar variable. Here's an example:

```
my @numbers = (1, 2, 3);
my $last_number = @numbers;
# $last_number will be assigned the value 3
```

List Context:
In list context, a variable or expression is evaluated to produce a list of values. For instance, when assigning an array to a list of variables, each element of the array is assigned to the corresponding variable.

```
my @names = ('Alice', 'Bob', 'Charlie');
my ($first, $second, $third) = @names;
# $first will be 'Alice', $second will be 'Bob', $third will be '
    Charlie'

#Invalid! We need to use List context!
my %hash1{("key1", "key2", "key3")} = ("val1", "val2", "val3");
say $hash1{"Max"};

#Correct statement using List Context:
my %hash1;
@hash1{("Max", "Milla", "Tobi")} = ("Egger", "Maskov", "Krissmer")
    ;
say $hash1{"Max"};
```

Context influences various operations and built-in functions in Perl. For example, the behavior of the reverse function changes depending on the context. In scalar context, it concatenates the elements of an array into a single string, while in list context, it reverses the order of the elements.

3.3 Accessing Variables

The sigil used when accessing a variable in Perl varies depending on the intended operation. For instance, when declaring an array as @values, accessing its first element as a single value is done using $values[0]. On the other hand, retrieving a list of values from the array is achieved using @values[@indices]. The choice of sigil determines the contextual interpretation when working with variables. Here are some examples:

```
my @values = (1, 2, 3);
my $scalar = $values[0];#Retrieves a scalar
my @list = @vales[0,1]
my @empty_list = @values[0];  #Retrieves a single element list
```

List are evaluated to their final element or their length in the scalar context:

```
my @values = (1, 2, 3);
$length = @values;       #Evaluates to the length of the array
$val = @values[(0,1)];   #Evaluates to the value of the final index
```

3.4 Coercion

A variable in Perl has both a *container-type* - *scalar, array or hash* - and a value-type. Variables are either *numeric literals*, which are represented as integers or double-precision floating-point values or *string literals*. Perl is a loosely typed language, and therefore the value-type of variables can change over time. The container-type of a variable, however, cannot change.

In Perl, when an operation is performed on a variable or value that expects a specific type, but the actual value has a different type, *coercion* occurs. Perl attempts to automatically convert the value to the expected type to facilitate the operation. If you use a string in a numeric context, Perl will attempt to convert the string to a numeric value if possible.

```
my $num = "10";          #Value type: string
$num = $num + 5;         #Coercion from string to number
print $num;              #Output: 15

$num = $num + "a"        #Can't convert "a" to a number => 0
print $num               #Output: 10
```

3.5 Branching Directives

Like any other modern programming language, Perl also offers branching directives. They can either be declared in the conventional way, in which the condition is followed by the instruction - also called *prefix if*.

```
if ($value1 > $value2) {
    print 'value1 is larger than value 2';
}
```

Perl, however, also offers the possibility of using the *postfix if*, where the condition comes after the instruction. This due to Perl roots in linguistics, because it allows to express conditions in a way that mirrors how we naturally phrase statements in spoken language.

```
print 'value1 is larger than value 2' if value1 > value2;
```

3.6 Looping Directives

In Perl, developers can choose between a multiple styles for writing *for-loops* and *while-loops*.

For-loops
The syntax for writing *for-loops* in Perl is short and concise and allows the developer to create loops with or without counting variables or to iterate through elements of a list.

```
#For-loop without counting variable
for (1..10) {
    print "x";
}
#For-loop with counting variable
for my $i (1..10) {
    print "$i ";
}
#Foreach-loop for iterating over lists
my @fruits = ("apple", "banana", "orange");
foreach my $fruit (@fruits) {
    print "$fruit ";
}
```

While-loops
While-loops in Perl should look and feel familiar for most developers. But Perl offers an alternative: The *until* statement is a conditional construct in Perl that executes a block of code only if a specified condition is false. It is the opposite of the *while* statement.

```
my $count = 1;
until ($count > 5) {
    print "$count ";
    $count++;
}

my $count = 1;
until ($count > 5) {
    print "$count ";
    $count++;
}
```

4 Distinguishing features of Perl

Perl boasts a set of distinguishing features that contribute to its uniqueness and versatility as a programming language. Firstly, its distinctive yet expressive syntax enables developers to write concise and powerful code, accommodating a multitude of coding styles. Secondly, Perl's robust support for regular expressions empowers advanced pattern matching and sophisticated text manipulation, making it a potent tool for handling complex data processing tasks. Moreover, Perl is renowned for its default variables, which provide convenient access to frequently used data and simplify code development. Lastly, Perl's guiding principle of "There's More Than One Way To Do It" (TIMTOWTDI) embraces the flexibility of multiple solutions to a problem, encouraging developers to explore creative and innovative approaches. We will briefly cover the features mentioned and give a small introduction on the concepts.

4.1 Syntax

Perl's syntax is probably its most distinguishing and also most powerful feature. In the previous chapter, we provided a brief overview of Perl syntax, highlighting its flexibility and unique characteristics. However, delving into every detail of Perl's syntax would be beyond the scope of this paper. Nevertheless, it is worth mentioning a couple of additional examples that demonstrate Perl's rich syntax. One such feature is the ability to access array elements using negative indices, allowing easy retrieval of elements from the end of the array without explicit calculations. Another intriguing aspect is Perl's treatment of strings when used with the ++ operator, enabling string increment operations. These are just a glimpse of the many distinctive syntax elements that contribute to Perl's power and versatility, making it a language well-suited for a wide range of programming tasks[2][7].

4.2 Default Variables

Perl's default variables are a unique and powerful feature that sets it apart from other programming languages. These variables are automatically available for use without explicit declaration, providing convenient access to commonly used data within Perl programs. They are denoted by special punctuation symbols, such as $_, $@, $., and others, and they carry specific meanings based on the context in which they are used.

4.2.1 $_ - The default scalar variable

It serves as the default target for many built-in functions and operations, eliminating the need for declaring and assigning variables in certain contexts. Functions and operators use this variable as a default, if no parameter is explicitly used. In loops, the default scalar variable is a convenient placeholder that references the current element being processed in the loop. Additionally, when dealing with open files, **$_** represents the current line, allowing seamless file processing without requiring an explicit variable to hold the line content.

```perl
$_ = 'My name is Maximilian';
say;
#Output: "My name is Maximilian"

print "#$_ " for 1 .. 3;
#Output: "#1 #2 #3"

my @fruits = ("apple, banana, melon");
print "$_" for @fruits;
#Output: "apple, banana, melon"
```

4.2.2 @_ - The default array variable

In contrast to many other programming languages, Perl subroutines do not explicitly declare their arguments. Instead, Perl uses the default array variable @_ to capture the arguments passed to the subroutine during the function call. By accessing the elements of @_ directly within the subroutine code, Perl enables more flexible and concise function definitions, allowing subroutines to handle varying numbers of arguments seamlessly.

```perl
sub print_arguments {
    my ($first_arg, $second_arg) = @_;

    say "$first_arg $second_arg";
}

print_arguments("Hello", "World");
#Output: "Hello World"
```

4.2.3 $. - Current line number variable

The **$.** variable in Perl is a special built-in variable that represents the current line number in a file being processed. It provides a convenient way to keep track of the line number while reading or processing files in Perl scripts. This feature is particularly useful in tasks that involve parsing or filtering data from text files.

```perl
while (<$file>) {
    print "$. $_";
}
#Outputs each line of the open file with the current
#line number in front
```

4.3 Regular Expressions

Regular Expressions in Perl are a powerful and fundamental feature that has played a significant role in making Perl a dominant force in text processing and pattern matching. Regular expressions, commonly known as regex, provide a concise and expressive syntax for describing complex text patterns. In Perl, regular expressions are seamlessly integrated into the language, enabling developers to perform advanced string manipulations, data extraction, and pattern matching with ease[3, 283-285].
While Perl didn't create regular expressions, it certainly played a crucial role in popularizing them among developers. The flexibility and efficiency offered by Perl's regular expressions attracted programmers dealing with text processing tasks, and its expressive syntax became a model for regular expression support in other programming languages. Many languages nowadays offer regular expression packages that are "Perl compatible"[3, 90-91].

In Perl, you can use regular expressions by using the powerful built-in support for regex through the matching operator **m//** (the **m** can be omitted for simplicity) and the substitution operator s///. The m// operator is used to match a regex pattern against a string, while the s/// operator is used for search and replace operations with regex. Using the binding operator = we can apply the regex defined in the second operand to the string of the first operand.

```
my $name = 'Chatfield';
say 'Found a hat!' if $name =~ /hat/;
#Output: "Found a hat!"
```

Moreover, Perl compatible regular expressions provide several quantifiers, such as *****, **+**, **?**, and **{}**, which offer different ways to define the repetition:

- The * quantifier matches zero or more occurrences of the preceding element.
- The + quantifier matches one or more occurrences of the preceding element.
- The ? quantifier matches zero or one occurrence of the preceding element (making it optional).
- The {} quantifiers, like {n} or {n, m}, allow you to specify exact or range-based repetitions, respectively.

```
my $string = 'Brussle';
say 'True' if $string =~ /m?ussle/;     #Output: True
$string = 'mmmmssle';
say 'True' if $string =~ /mu**ssle/;    #Output: True
$string = 'ussle';
say 'False' if $string !~ /m+ussle/;    #Output: False
```

Perl compatible regular expressions offer many more features than matching, substitution and quantifiers, but it would again break the scope of this paper to go into more detail.

4.4 TIMTOWTDI - There's More Than One Way To Do It

Unlike many other programming languages that promote a single prescribed approach to problem-solving, Perl embraces a contrasting philosophy known as TIMTOWTDI, which stands for "There's More Than One Way To Do It". It reflects Perl's philosophy of providing multiple, diverse solutions to a problem, allowing developers the freedom to choose the approach that best fits their coding style and preferences. Perl embraces the idea that different developers may have distinct perspectives and ways of tackling a task, and rather than imposing a single rigid solution, it encourages creativity and flexibility. This philosophy is deeply embedded in Perl's design and is evident in various aspects of the language, such as its expressive and adaptable syntax, powerful regular expression support, and numerous built-in functions. The TIMTOWTDI principle empowers Perl programmers to explore various coding techniques and discover innovative solutions that suit their specific needs[2, 3].

5 Limitations

While Perl is a powerful and versatile programming language, it does come with certain limitations that developers should be aware of. Understanding these limitations is essential for making informed decisions when choosing a programming language for a specific project.

5.1 Too many ways to do it?

The "There's More Than One Way To Do It" (TIMTOWTDI) principle in Perl, while promoting flexibility and creativity, can also lead to some challenges and potential issues in certain contexts. With numerous ways to accomplish the same task, Perl code can become difficult to read and understand, especially for developers who are not familiar with the various coding styles. This can pose challenges for team collaboration and code maintenance over time. TIMTOWTDI can lead to inconsistencies in codebases, as different developers may choose different approaches for similar tasks. Establishing coding standards and maintaining consistency across projects can become challenging. For beginners or developers transitioning to Perl from other languages, the abundance of options may initially be overwhelming. The learning curve can be steeper due to the need to understand various syntaxes and approaches for accomplishing tasks. When encountering issues in Perl code, it may be more challenging to identify the root cause, as there could be multiple ways to implement a certain feature or algorithm. Overall, while TIMTOWTDI offers creative possibilities, developers must carefully consider trade-offs and employ best practices to mitigate potential issues related to code readability, maintainability, and consistency[5, 212-213].

5.2 Perl is scripting language

As a scripting language, Perl offers powerful text-processing capabilities and quick development cycles, making it an excellent choice for automating tasks and handling diverse data formats. However, its nature as an interpreted language means that Perl code cannot be easily obfuscated to hide its intent, making it more transparent and accessible. Additionally, due to the interpretation process, Perl programs tend to be slower compared to well-optimized programs written in compiled languages like C or C++[4].

5.3 Code readability

Code readability in Perl can sometimes be challenging, especially when default variables and one-liners are involved. Default variables like $_, which are automatically used by certain functions and loops, can make code terse but less self-explanatory, as it may not be immediately clear which variable is being operated on. Similarly, Perl's one-liners, though powerful for quick text processing tasks, can quickly become difficult to comprehend as they often pack multiple operations into a single line. This conciseness, while efficient for short scripts, can lead to reduced readability and maintenance issues in larger projects. If one would like an example of the difficulty to read concise, Perl scripts, please refer to the list of Perl one-liners and try to interpret them.

5.4 Popularity

Over the years, Perl has experienced a decline in popularity compared to other programming languages. One contributing factor to this trend may be the emergence of newer languages with a focus on simplicity, performance, and modern paradigms. While Perl remains a powerful language with its strengths in text processing and scripting tasks, its complex syntax and steep learning curve may deter newcomers and developers seeking more approachable alternatives[4].
The dwindling popularity of Perl presents a significant challenge due to the network effect, where the value and utility of a language increase with the number of users and contributors. As a language's community grows, it fosters an ecosystem of libraries, frameworks, and tools, enriching the development experience and making it more attractive to new developers. Conversely, a declining user base may lead to a reduced focus on language development, fewer updates, and a shrinking pool of resources and expertise. This, in turn, could create a self-reinforcing cycle, making it less appealing for developers to invest in Perl projects.

[4]TIOBE Index: `https://www.tiobe.com/tiobe-index/` visited 13.07.2023

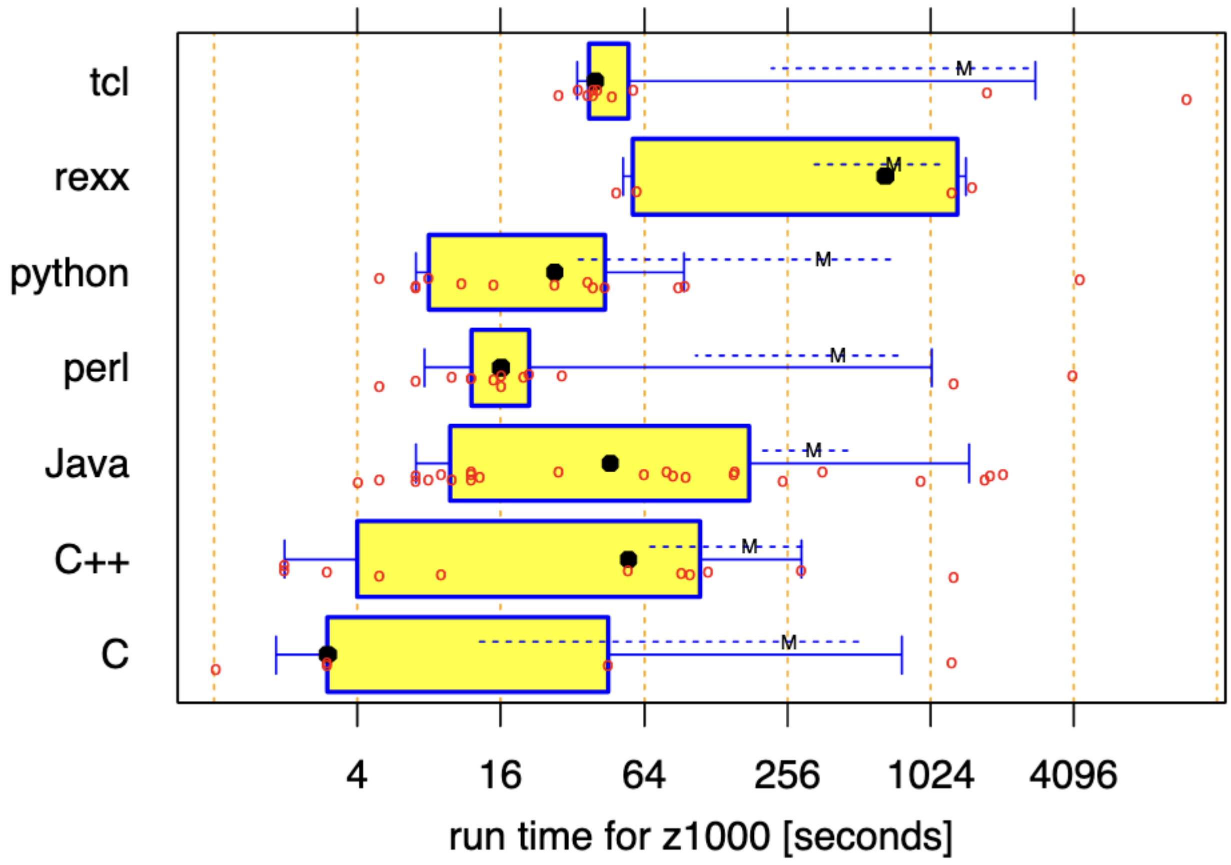

Figure 1: Program run time on the z1000 data set on logarithmic axis

Source: Prechelt, L. (2000)[4]

6 Comparison to C and Python

This section provides a comparative analysis of Perl with two other widely used programming languages: C and Python. Each language possesses unique characteristics, strengths, and trade-offs that make them suitable for different types of projects and programming paradigms. By exploring the key differences and similarities between Perl, C, and Python, this section aims to help developers make informed decisions when selecting the most appropriate language for their specific programming requirements.
We will compare the programming languages by looking at the paper "*An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl for a search/string-processing program*" published by Lutz Prechelt in the year 2000[4]. The paper tries to compare the programming languages by giving a number of developers specialized in different programming languages the same task.

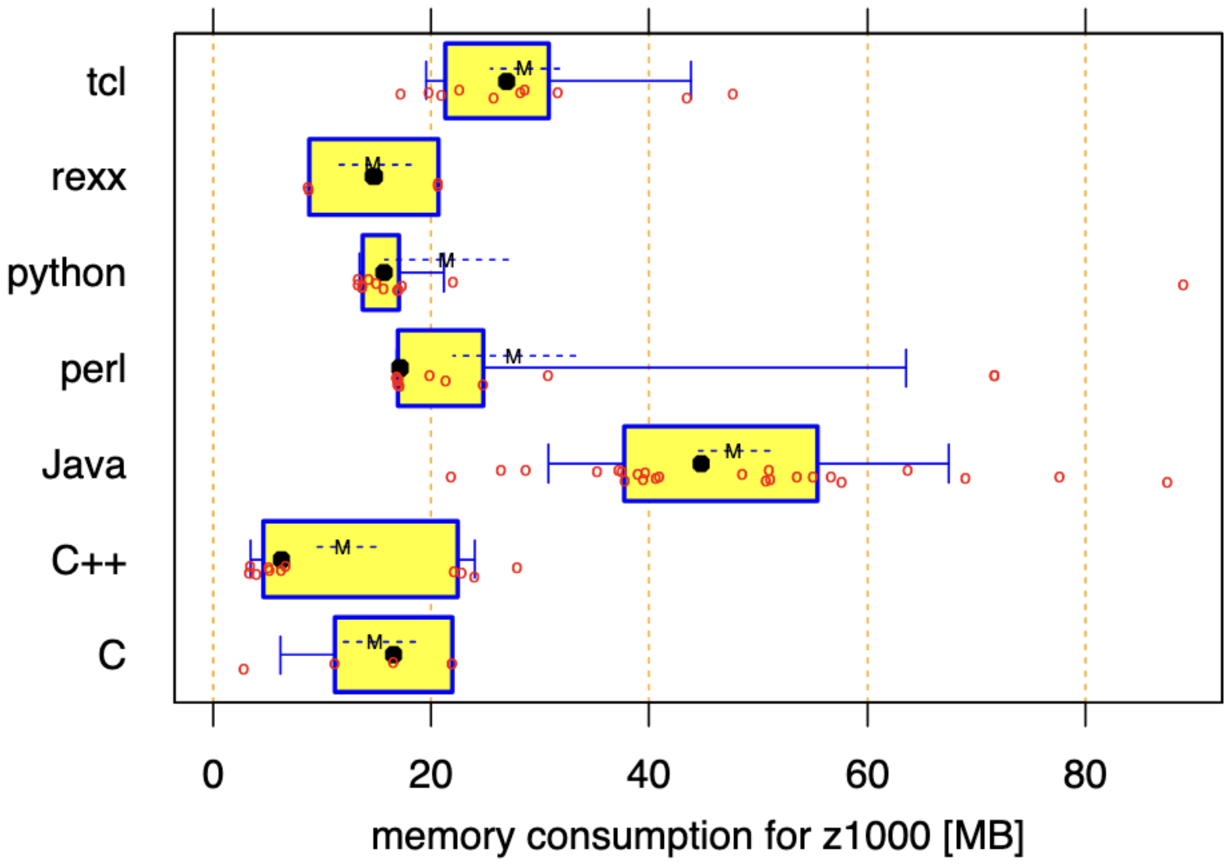

Figure 2: Program run time on the z1000 data set on logarithmic axis

Source: Prechelt, L. (2000)[4]

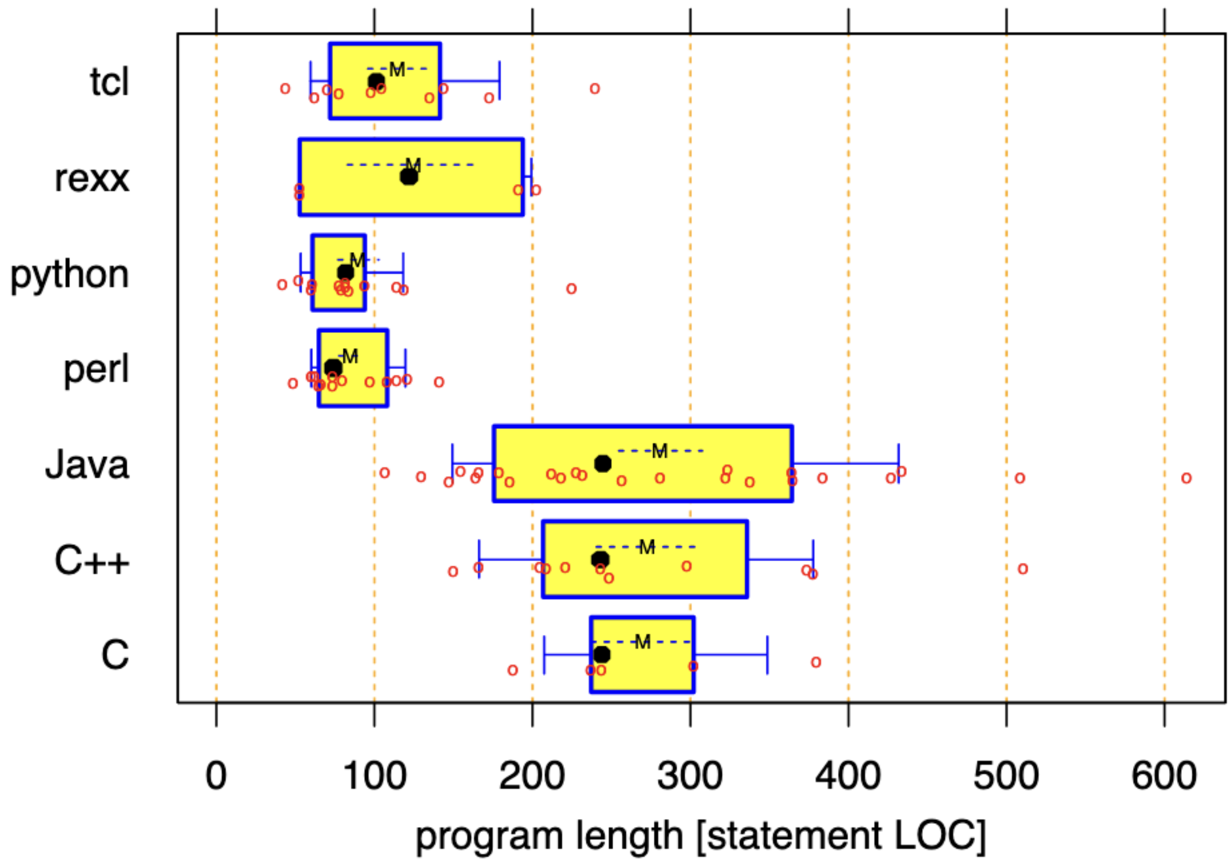

Figure 3: Program run time on the z1000 data set on logarithmic axis

Source: Prechelt, L. (2000)[4]

6.1 Comparison to C

Upon reviewing figures 1-3, several conclusions can be drawn regarding the performance and coding complexity of the compared programming languages. Firstly, the optimized C program demonstrates superior performance compared to Perl programs, indicating that C is well-suited for computationally intensive tasks and offers efficient execution. Additionally, it becomes apparent that working with C might demand a higher level of expertise, as its optimization and memory management require a deeper understanding of low-level concepts. Moreover, the figures suggest that accomplishing a simple task in C may require a larger number of lines of code compared to Perl, highlighting the language's potential for increased development effort for straightforward tasks. These observations underscore the trade-offs and considerations developers should consider when choosing between C and Perl for different types of projects.

6.2 Comparison to Python

After analyzing figures 1-3, we notice that Perl and Python perform almost the same in terms of speed and memory consumption and require approximately the same amount of lines of code. However, Python is vastly more popular (rank 1 on the TIOBE index, compared to Perl's rank 27 [5]), and there are several reasons why. First, Python has a simpler and more readable syntax, making it easier for beginners to learn and use. Second, Python has strong libraries for important areas like data science and machine learning, which makes it highly valuable in those fields. Third, many industries have adopted Python as their go-to language, making it widely used and supported. Lastly, the more people use Python, the more valuable it becomes due to the network effect, where its benefits grow with its popularity. All these factors contribute to Python's widespread acceptance and explain why it has become more popular than Perl despite similar performance.

7 Conclusion

In conclusion, Perl stands as a language with distinctive and powerful syntax, offering developers a wide range of creative solutions for various tasks. It continues to excel in system administration and text manipulation tasks, leveraging its strengths in regular expressions and text processing capabilities. Moreover, its community remains dedicated, providing ongoing support and development for the language. However, Perl's future growth potential is hindered by the lack of widespread industry adoption, resulting in limited interest from new developers. While Perl retains its effectiveness for

[5]TIOBE Index: `https://www.tiobe.com/tiobe-index/` visited 13.07.2023

specific use cases, the overall trajectory suggests that its popularity will continue to decline. Nonetheless, Perl will continue to serve as a valuable tool for those who appreciate its strengths and rely on its capabilities in their domains.

References

[1] Beginner's Introduction to Perl — perl.com. `https://www.perl.com/pub/2008/04/23/a-beginners-introduction-to-perl-510.html/`. [Accessed 09-Jun-2023].

[2] Chromatic. *Modern Perl 4e*. Pragmatic Programmers, Raleigh, NC, Nov. 2015.

[3] J. E. Friedl. *Mastering regular expressions*. " O'Reilly Media, Inc.", 2006.

[4] L. Prechelt. An empirical comparison of c, c++, java, perl, python, rexx and tcl. *IEEE Computer*, 33(10):23–29, 2000.

[5] L. Prechelt. Are scripting languages any good? a validation of perl, python, rexx, and tcl against c, c++, and java. *Adv. Comput.*, 57:205–270, 2003.

[6] L. Rosenfeld and A. B. Downey. *Think Perl 6*. O'Reilly Media, May 2017.

[7] R. Schwartz, b. d. foy, and T. Phoenix. *Learning Perl*. O'Reilly Media, 2011.

[8] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl*. O'Reilly Media, Sebastopol, CA, 3 edition, July 2000.

[9] L. Wall et al. The perl programming language, 1994.