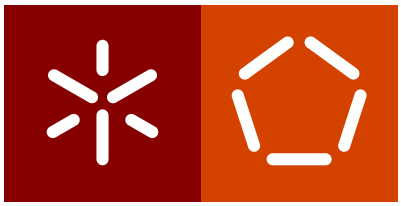
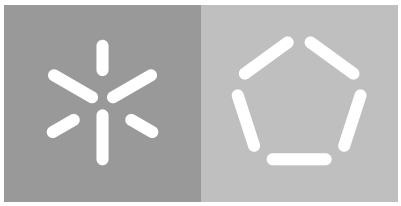


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Júlio Miguel de Sá Lima Magalhães Alves

Properties that better describe a programming language

January 2024



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Júlio Miguel de Sá Lima Magalhães Alves

Properties that better describe a programming language

Master dissertation
Master Degree in Informatics Engineering

Dissertation supervised by
Pedro Rangel Henriques
Alvaro Costa Neto

January 2024

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

Júlio Miguel de Sá Lima Magalhães Alves

Acknowledgments

Ao longo de todo o meu percurso académico, contei com o apoio de várias pessoas, a quem quero agradecer.

À professora, Maria João Varanda Pereira, pela orientação e disponibilidade, não só na realização do meu primeiro artigo científico, como também na realização desta dissertação.

Aos meus orientadores, Professor Doutor Pedro Rangel Henriques e Professor Alvaro Costa Neto, por terem sido incansáveis durante este ano e por me transmitirem sempre os melhores conselhos. O vosso apoio foi absolutamente fundamental.

Aos meus amigos, pelo companheirismo e pelos bons momentos que partilhamos, que ajudaram a tornar esta caminhada mais fácil.

À minha família, por estarem do meu lado incondicionalmente.

Aos meus padrinhos e às minhas tias, Elisabete e Assunção, por todo o carinho e por serem sempre um ombro amigo.

Aos meus primos, André e Mafalda, por serem pilares fundamentais no meu crescimento pessoal.

À minha namorada, Maria, por me mostrar todos os dias que é possível ser e fazer melhor, mesmo quando não parece possível.

E, finalmente, aos meus pais e à minha irmã, que são os melhores exemplos que tenho na vida. Agradeço-lhes pelos sacrifícios que fizeram para me proporcionar as melhores oportunidades possíveis, pelo apoio incondicional em todos os momentos e por serem os meus maiores alicerces. Nada do que alcancei teria sido possível sem o vosso amor e apoio inabalável.

ABSTRACT

This document reports the development of a Master's Thesis, included in the second year of the Master's Degree in Informatics Engineering at Universidade do Minho in Braga, Portugal.

The main goal for this project was to identify which characteristics influence the recognition and identification of a programming language, considering both its typical source code elements and its linguistic style. In other words, which elements contribute the most to the characterization of a language? How many structural elements of a language may be modified without losing its identity?

In order to achieve these goals, a comprehensive bibliographic research was made, ranging from basic concepts such as programming languages and how they work, to several state-of-the-art studies that have been conducted in the same context of this project. Complementary to this research, a set of programming languages was also chosen as a study subject, which resulted in a detailed review and categorization of their characteristics.

After the definition of a general approach, a survey was developed and conducted to gather programmers' answers on how they identify and recognize programming languages. In addition to the survey, a machine learning model was also used to evaluate how these two facets (human *versus* machine) compared to each other. This dual approach provided insights into which syntactic and semantic elements have a greater influence on the identity of a programming language.

This Master's project resulted in an overall picture of programming languages' characteristics and the relative influence they have on both programmers' and AI-driven recognition. This result may serve as support for language engineers and project managers who wish to reduce attrition when defining or designing new languages for a project, domain, or context.

Keywords: Programming Languages, Programming Language characterization, Programming Language design, Programming Language identification

RESUMO

Este documento é referente ao desenvolvimento de um projeto de Mestrado, incluído no segundo ano do Mestrado em Engenharia Informática da Universidade do Minho em Braga, Portugal.

O objetivo principal deste projeto é identificar quais características influenciam o reconhecimento e a identificação de uma linguagem de programação, especificamente analisando o código fonte e o seu estilo linguístico. Por outras palavras, quais elementos contribuem mais para a caracterização de uma linguagem? Especificamente, quantos dos elementos estruturais de uma linguagem podem ser modificados sem que esta perca a sua identidade?

Para atingir os objetivos deste projeto, foi realizada uma pesquisa bibliográfica exaustiva, que abrange desde conceitos básicos, como linguagens de programação e seu funcionamento, até vários estudos de ponta realizados no mesmo contexto deste projeto. Complementarmente a esta pesquisa, um conjunto de linguagens de programação foi escolhido como objeto de estudo, resultando numa revisão detalhada e categorização de suas características.

Após a definição de uma abordagem geral, foi desenvolvida e conduzida uma pesquisa para reunir respostas de programadores sobre como eles identificam e reconhecem linguagens de programação. Além da pesquisa, um modelo de *machine learning* também foi utilizado para avaliar como essas duas facetas (homem *versus* máquina) se comparavam entre si. Esse duplo enfoque forneceu *insights* sobre quais elementos sintáticos e semânticos têm maior influência na identificação de uma linguagem de programação.

Este projeto de mestrado resultou em uma visão geral das características das linguagens de programação e da influência relativa que elas exercem tanto no reconhecimento de programadores quanto no reconhecimento impulsionado por IA. Este resultado pode servir como suporte para engenheiros de linguagens e gerentes de projetos que desejam reduzir atrito ao definir ou projetar novas linguagens para um projeto, domínio ou contexto.

Palavras-Chave: Linguagens de programação, Caracterização de linguagens de programação, Design de linguagens de programação, Identificação de linguagens de programação

CONTENTS

LIST OF FIGURES

1

INTRODUCTION

In this first chapter of the dissertation, the thesis is introduced through its context. Furthermore, objectives are detailed, the development approach is specified and the document structure is presented.

1.1 CONTEXT

Computers have evolved to be capable of recognizing and translating to machine code sets of sentences written according to formal rules, which enables them to execute the tasks they are being asked to do. These sets of sentences are called *programming languages*. Every programming language has its own syntax and semantics rules that may convey similarities between each other, but that also makes each language unique (Khasnabish et al., 2014). Being formal languages, these rules have to be strictly followed in order to construct valid programs.

Each programming language has been developed with different goals in mind, with some being more specific for certain fields of application, such as artificial intelligence or web development, while many others are denominated as being of general purpose. It can be said that behind the design of a programming language lies a philosophy of problem solving, usually suited for the intended area of application (Klein et al., 2011). It is also evident that all languages have different characteristics, that can become obstacles not only to the development process, but also in teaching. Due to each language's nature, it may be harder for programmers and students to understand how to properly apply the language rules to meet their needs (Neto et al., 2021). The study of these differences and the identification of languages' main characteristics are not common practices, making them very important subjects to be tackled.

It is possible to identify some interesting questions that deserve further research: What indicates that two programming languages are different if what distinguishes them is not clear? What characterizes and identifies a language? Is it a syntax driven feature, such as the opening and closing symbols of code blocks? Could it be how a variable is declared? If a programming language is incrementally modified, at what point does it lose its identity?

1.2 OBJECTIVES

This Master's Degree thesis has the following objectives:

- To find what syntactic and semantic characteristics are most relevant for a language's identity from programmers' point-of-view;
- To evaluate how changes in each of these syntactic and semantic characteristics affect programmers' identification of the language, and how disruptive they are;
- To establish an identity comparison method between languages, quantifying its similarities and differences;
- To elaborate a panorama about several active languages, relating to its similarities and differences;
- To develop a programming language identity analysis tool.

1.3 RESEARCH HYPOTHESIS

Programming languages' identities might be recognized in a source code snippet by evaluating their characteristics, both theoretical and practical, as ranked in importance by programming languages users.

1.4 DEVELOPMENT APPROACH

The development of this thesis followed an iterative methodology throughout an academic year, composed of the following steps:

- Bibliographic study concerning the definition and characterization of a large set of programming languages that must be initially chosen. That phase shall also include the search for bibliography describing works related to the one here proposed;
- Analysis of different changes that can be made to each language and how disruptive they are to the language's identification and characterization;
- Comparison of the previous phase outcomes when applied to other languages;
- Proposal of a characterization method to identify each programming language;
- Design and implementation of a tool to automate such identification approach;
- Construction of a panorama highlighting similarities and differences between the previously chosen languages;

- Results evaluation and discussion.

1.5 DOCUMENT STRUCTURE

The document is composed of five different chapters with the following contents:

INTRODUCTION In the first part of the document, the theme is contextualized and this work's objectives are stated. Moreover, it also contains the development approach with every step to be followed;

STATE OF THE ART In this chapter, crucial topics such as programming languages processing, different programming languages and their characteristics and machine learning approaches are introduced;

PROPOSED APPROACH In this chapter, solutions for both the study and the application that has been developed, in order to help visualize better the problem in hands;

DEVELOPMENT In this chapter, the methods applied to the study are discussed and its results analyzed;

CONCLUSION The fifth and last chapter contains what was concluded during the realization of this document and a brief presentation on what the future work should be.

2

STATE OF THE ART

Before understanding what are the defining characteristics of a programming language, it is crucial to comprehend what a programming language is, how it is defined and how it is processed. This chapter will begin by explaining what a programming language is, followed by an analysis of different studies related to the definition and identification of programming languages through different techniques.

2.1 PROGRAMMING LANGUAGES PROCESSING

Programming languages are defined by sets of rules, that together form a well-constructed grammar. These rules define both its structure—syntax rules— and meaning—semantic rules. These instructions, also known as code, can be used to create programs that perform specific tasks, solve problems, and interact with other systems.

Programming languages main purpose is to be used to write algorithms, that are not more than sets of instructions a computer can understand and execute, in order to solve problems and aid task performing. A lot of programming languages have been developed, creating a diverse set of different syntax, structure and features. While these characteristics contribute to their suitability for certain areas, other factors like performance, community support, and available libraries also influence their practical applicability.

The main way to define the syntax of a programming language is by using a grammar. A formal grammar is composed of a set of rules that, if followed, create valid text for a language. To formally specify a grammar, four elements are needed: (Aho, 2007)

- Finite set of tokens - The elementary symbols of the language defined by the grammar;
- Finite set of non-terminal symbols - Represent a set of strings of terminals;
- Finite set of grammar rules - Consisting of a token and a sequence of terminals and/or non-terminal. Its purpose is to specify one of the written forms of a construct;
- Start symbol - From where all the previous components descend.

The process of translating human-readable source code written in a high-level programming language into machine code or an intermediate code that can be executed by a computer is called compilation. Programming languages can also be interpreted. Interpretation refers to the process of executing a program written in a high-level programming language directly without prior compilation. An interpreter reads and translates the source code into machine code or an intermediate code line by line, executing each statement as it goes along.

Compilation has a few steps divided in two phases, front-end and back-end. (Aho, 2007).

2.1.1 *Front-end*

The front-end phase of compilation, which can also be called the analysis phase, determines if a program is syntactically, lexically or semantically badly formed. At this phase, the lexical, syntactical and semantics analysis are done.

Lexical Analysis

A compiler's first process when examining a program is the lexical analysis (Aho, 2007). In this phase, it captures all the characters in the source code and groups them in sequences called *lexemes*. A lexeme is a unit of lexical meaning that correlates to a set of forms that a single word can take, as in, the words *sing*, *sings* and *sang* are all forms of the same lexeme, represented by the verb *to sing*.

At the end of this process, all lexemes become *tokens* with a name and a value, that, if no errors are encountered, move on to the next phase: the syntax analysis.

Syntax Analysis

The syntax can be described as being the structural form of the language (Floyd, 1964), which means, it defines how a set of symbols is used to create valid source codes.

In this second phase, which can also be called *parsing*, the grammatical structure is transformed into a tree containing the tokens generated by the lexical analyzer. This tree represents the order and the correlation between the statements of the source code, by creating a hierarchical organization via its structure. Usually, the interior nodes represent operations and their children represent the arguments. The order of the operations is also taken into account so that a correct program is built.

Semantics Analysis

Semantics is, simply put, the meaning of the language (Floyd, 1964), being what gives some limited and well-defined concept of sense to a program. A language's semantic definition specifies which sentences in a program are valid commands from a meaningful point of

view. The main job of semantics is to determine which steps must be given in order for the computer to be able to run a program in that particular language. Semantics is the process through which valid strings in a programming language syntax are assigned computational meaning.

In this compiling phase, information gathered in the previous phases is used to guarantee the source program has semantic consistency with the language. It is at this phase that, for example, the type checking is done and valid combinations of values are verified.

2.1.2 *Back-end*

The back-end phase, also called the synthesis phase, gathers the information assembled by the front-end to build the target program. This phase is constituted by the intermediate code generation, machine-independent code optimization, code generation and machine-dependent code optimization.

Due to the nature of this research, this phase will not be deepened as it doesn't relate to the end goal.

2.2 PROGRAMMING LANGUAGES CHARACTERISTICS

As acknowledged within the programming community, each programming language exhibits distinct characteristics and specifications. In the following subsections, different programming languages will be approached, while stating their main characteristics. The selection of these programming languages was based on the survey participants' familiarity, considering that they are included in their university curriculum. It is crucial to emphasize that the primary objective of this study is to explore the retrocompatible, and common, features inherent in each programming language. Consequently, there is a heightened emphasis on earlier versions of each language.

2.2.1 *C*

C was developed by Dennis Ritchie in the 1970s as a general-purpose, structured programming language. It wasn't created with any particular application field in mind, working well for both commercial and academic applications. Some of its features are: portability, flexibility, effectiveness, efficiency, reliability and interactivity.

Every component of the C programming language has to be used in a specific way or sequence. Some of the defining characteristics of C are the following (Programming, 2016):

- Mandatory `main` function. The source code will not compile without it since it defines the entry point for execution;
- Use of brackets to enclose the contents of a code block;
- Semi-colons are applied to indicate the end of a statement;
- Variables and constants must be explicitly declared using the `type identifier` pattern;
- Functions are the fundamental structural elements of the source code;
- Function signatures must have a return type, followed by its name and a pair of parenthesis grouping all arguments;
- Functions may return no value. In this case, the return type of the function must be `void`;
- The fixed-length array is the only data collection construct built into the language and its declaration follows the basic pattern. Lengths and dimensions are defined using square brackets after the identifier;
- Pointers to indirectly access memory addresses. A pointer is a variable that holds an address to a memory location and is declared like any other variable, with the only difference being the need for an asterisk prefix before the identifier.

2.2.2 C++

C++ was developed by Bjarne Stroustrup in the late 1970s. It is also a general-purpose programming language, with some of its main application areas being: operating systems construction, games development, compilers implementation and advanced computation.

C++ aimed to mix Simula's facilities for program organization and C's efficiency and flexibility on systems programming. As an extension of C, C++ contains all of its characteristic syntactic and semantic constructs, encapsulating all of its features. However, C++ introduced *Object-Oriented Programming (OOP)* concepts such as classes, references, access control, properties and methods, bringing C's structured form into a new paradigm (Stroustrup, 1996). Some of its main characteristics are:

- C++ imported all the basic definitions for syntactic and semantic elements from C. Therefore, characteristics such as the entry point for execution, code block notation, declaration syntax and so on were either unaltered or simply augmented;
- The concept of classes was the most important aspect that C++ implemented on top of C. Classes are data types defined by the programmer, composed of variables and

methods that can manipulate them. Each variable and method has can have different kinds of accesses, as they can be private, public or protected;

- Classes are declared using the `class` keyword, in a similar way to structures (`struct`);
- Properties are declared inside the class definition using the standard pattern;
- Methods may be declared either inside the class definition or, if declared outside, using a specific notation (`class::method`);
- The standard library added several data structures, such as dynamic vectors, linked lists, queues etc;
- Some other additions to C's mechanisms are also present, such as operator overloading, `new` and `delete` commands, templates, virtual functions etc.

2.2.3 *C#*

C# was created by Microsoft, with its first version being dated from 2000. It is a modern, general-purpose programming language. Some of its main application areas are the development of Windows and Web applications.

C# is an evolution of C++, simplifying many of its features whilst maintaining its power. It was built as an OOP Language, meaning a program cannot be created without building classes and its correspondent fields and methods inside.

As an evolution of C++, it was meant to solve some of its problems, like pointers usage and memory allocation issues, which helped programmers tremendously. It reduced problem solving complexity and time spent solving problems that derived from working directly with memory (Abolrous, 2007).

Some of C# characteristics are the following:

- The need of a `Main` function, just like its predecessor C++;
- As a successor of C++, it maintains the same rules to delimit code blocks, with brackets being used to delimit code blocks and using semi-colons to end;
- When declaring a variable, one can explicitly state what the variable type will be or can use the keyword `var`, and the variable type will be infered by the compiler;
- As C# is an OOP language, classes are a very important structural element, however, it can also be said that methods are very important as well;
- C# offers a variety of different data structures, such as arrays, lists, stacks, queues, sets and dictionaries;

- Notion of directives - To indicate that a specific *namespace* will be used. Identified by the keyword `using`, followed by the desired namespace identifier;
- Reference types - Similar to pointers and references in C++. These reference types are Classes, Interfaces, Delegate, Objects and Strings;
- Convert value types to reference types and vice versa - By using *boxing* and *unboxing*, respectively. *Boxing* is done by assigning the value type variable to a variable of type object. *Unboxing* is done by casting the reference type variable.

2.2.4 *Java*

Java was developed by a team lead by James Gosling at Sun Microsystems. It was launched in 1995. When Java was created, it was meant to be general-purpose, concurrent, class-based, object-oriented language, simple enough to be learned quickly. Java was inspired by C and C++. Its main application areas are desktop and mobile applications, big data and web development.

Java is strong and statically typed. This means every variable and expression must have a type and that type is known at compile time. Types also limit what values a variable can have or an expression can produce.

Some of Java characteristics are the following (Gosling et al., 1996; Jav):

- As Java was inspired by C and C++, having a `main` function is obligatory, brackets are used to delimit code blocks and semi-colons to end statements;
- All variables must be declared with a type;
- A variable can be declared as `final` - This means it can only be assigned once and will always have the same value, once assigned;
- Just like C#, although classes are a crucial structural element, methods also have a vital importance on the language structure;
- Java also offers a big variety of data structures, that are arrays, lists, stacks, queues, sets and maps;
- Types can be divided into two categories. These categories are primitive types, such as `boolean` and numeric types, and reference types such as classes, interfaces and array types. Values in reference types relate to objects, which may be arrays or instances of a class type;
- The special type `null` - `null` has no name and can be assigned or cast to any reference type and its reference is the only possible value of an expression of `null` type;

- Values on types - Primitive types will always hold a primitive value of that primitive type whereas variables of reference types are more complex. If a variable is of a class of a certain type, it can hold a null reference, a reference to an instance of that class or any of its subclasses. In cases where the variable is of interface type, it can also hold a null reference or a reference to any class that implements the interface;
- *Boxing* and *Unboxing* - In Java, it's only possible to convert expressions of primitive type to corresponding expressions of reference type and vice versa;
- Conversions - Such as identity conversion, widening and narrowing primitive conversion, unchecked conversion and capture conversion;
- Usage of other packages - To use other packages, one must use the keyword `import` followed by the name of the package.

2.2.5 *Python*

Python was developed by Guido van Rossum and released in 1991. It is a general purpose programming language and its main application areas are Web applications, general software, data science and machine learning.

Python is an interpreted, interactive and OOP language but also supports other programming paradigms such as procedural and functional. Some of Python characteristics are the following:Pyt

- It isn't obligatory to have a `main` function;
- In Python, instead of using braces to delimit code blocks, these are defined by indentation;
- In Python, semi-colons aren't required to mark the end of a statement. To delimit the end of a statement, a new line is enough;
- Variables are declared when a value is given to them, without needing to declare their type;
- To define a function, its name must be preceded by the keyword `def`. After the function name, a pair of parenthesis must be used and inside them, the user can indicate a set of arguments for the function;
- Despite being an OOP language, Python main structural element are functions;
- When using Python, the user can use different data structures like lists, tuples, sets, dictionaries, strings and range.

2.2.6 *Haskell*

Haskell was developed in 1990 by a team of scientists, including Simon Peyton-Jones, Paul Hudak Erik Meijer, Philip Wadler and Lennart Augustsson. It is a general purpose language and its mainly used in academic studies, however, it is also used in industry. Differently from all other languages previously discussed, Haskell is a purely functional programming language meaning, instead of telling the computer what to do, like in imperative programming languages, the programmer tells the computer what something is. Haskell is also lazy, meaning that computations are not performed unless explicitly instructed in the program.

Haskell is statically typed, allowing multiple possible errors to be detected at compile time and, to add to this, its type system has type inference, which doesn't force the user to label every piece of code with a type as it will find that by itself.

Some of Haskell characteristics are (Lipovaca, 2011):

- Haskell doesn't need an entrypoint;
- Code blocks are delimited by indentation, just like Python;
- To define the end of a statement, a new line is used;
- To declare a variable in Haskell, the user must use either the keyword `let` or `var`, followed by the variable name. Variables are imutable by default, however, if the keyword `var` is used, it allows the variable to be mutable;
- Due to Haskell strong inference mechanism, it is not needed to specify the variables type;
- To declare a function type, the user declares its name, followed by the :: operator, which can be read as "type of", and all its parameters. These are separated by -> characters, with the last parameter being the return type;
- Functions are the main structural element of the language;
- The data structures offered by Haskell are lists, tuples, sets, maps, strings and range;
- Pattern matching - A very distinct characteristic Haskell has in comparison to other languages is that, when defining a function, they are defined using pattern matching. This allows the compiler to know how to deal with the information when a specific situation occurs and it also leads to simpler and more readable code.;
- Guards - instead of having multiple lines of code to write a nested `if...then...else` condition, Haskell has what is called guards. Guards are indicated by a pipe character (`|`) followed by a boolean expression and what will be used in case the boolean

expression is true. Should the boolean expression be false and the function goes to the next guard, and so on;

- `where` - In some other programming languages, when the user wants to store a variable for future use, he does exactly that and the variable is ready to be used whenever needed. However, in Haskell that's not possible so, to circumvent this limitation, it offers the keyword `where`. With `where`, the user can declare variables to be used inside a function and not worry about having to calculate its value every time, with its biggest limitation being the fact that `where` scope is limited to the function it was declared;
- `case` - Haskell also has a `switch` like statement, with the help of the keyword `case` and its used as `case <expression> of <pattern> -> <result>`;
- Haskell stands out from the other languages discussed in this text by placing a significant emphasis on recursion, a characteristic that sets it apart in terms of programming paradigm compared to the rest.

2.3 Machine Learning

When programmers create software, they typically don't account for all potential outcomes, leaving their programs open to mistakes. When pursuing a new interest or activity, similar situations arise. Nevertheless, the pathway to improvement involves encountering setbacks and deriving valuable insights from them.

It is feasible to show that programs, like humans, may improve when allowed to learn from their experiences. It was with this intention that Machine Learning started to be developed, as it is advantageous that computers can learn like people, achieving higher levels of proficiency and personalization (Mitchell, 1997). These improvements can be applied to many areas, with them being as different as medicine and advertising.

In accordance with learning paradigms, Machine Learning techniques are often categorized into three main groups: *Supervised Learning*, *Unsupervised Learning*, and *Reinforcement Learning*.

2.3.1 *Supervised Learning*

In this technique of Machine Learning, the computer receives some input-output pairs and learns how to map them (Stuart Russell, 2010). In supervised learning, the computer is given a training set that, as its name suggests, serves to train its software on how to map the input-output pairs. In order to verify its results, a test set is created and fed to the computer, where its accuracy is evaluated.

A model—in Machine Learning, a model is a process that has been trained to recognize certain types of patterns—is considered good when it generalizes well. In order for a model to generalize well, it is not desirable to construct a one hundred per cent accurate model because it suggests overfitting—meaning that the model is only capable of predicting the data it was given, and if given new data, it won't be able to predict as well. The exact opposite is called underfitting, when a model isn't accurate enough with the current data, which also indicates that it won't perform well when given new data.

2.3.2 *Unsupervised Learning*

In contrast to supervised learning, unsupervised learning does not obligate the user to provide input-output pairings, since the model will infer patterns from the input data, even though it does not get explicit feedback about its results. It usually uses clustering to accomplish its goal (Stuart Russell, 2010).

To categorize data into distinct groups, clustering algorithms employ a certain distance or dissimilarity measure. Formally, the distance between two points (d(x,y), with x and y being two different points) is considered to be a function that satisfies the following conditions:

1. $d(x, y) \geq 0, \forall x, y;$
2. $d(x, x) = 0, \forall x;$
3. $d(x, y) = d(y, x), \forall x, y;$
4. $d(x, y) + d(y, z) \geq d(x, z), \forall x, y, z;$

The distance can be measured using, as an example, the *Euclidian* or *Manhattan* distance. While distance must satisfy all the conditions above, dissimilarity must only satisfy the first three, with the fourth being optional. (Kanade et al., 2021)

2.3.3 *Reinforcement Learning*

As previously mentioned, individuals acquire new skills through a process of trial and repetition, persisting until proficiency is achieved. This happens because feedback is constantly given to the individual, whether it is positive or negative feedback. As an example learning how to play football. When the ball is shot to the goal, the player who made the shot will instantly know if it was good or bad, depending on if it was a goal or a miss. If it was a goal, the player will repeat the same shot; if it was a miss, the player will try to adjust it for the next time. A similar process occurs in reinforcement learning when the model learns from a succession of reinforcements. While in Supervised Learning, the user

guides the model throughout the whole process, in Reinforcement Learning, the model is not continually guided by explicit reinforcements.

2.4 Programming Languages Identity and Recognition

With the evolution of programming and the appearance of so many new programming languages, it becomes necessary to be able to distinguish them for various purposes, such as learning or storage in applications like GitHub. In this section, multiple ways of identifying a programming language that exists at the moment will be presented.

2.4.1 *Image Based Classification*

Image-based classification in Bonifro et al. (2021) was able to identify a very limited amount of different programming languages, making it attractive to try to evolve this method of classification.

In the paper by Bonifro et al. (2021), it was intended to discover if it was possible to identify the programming language used in snippet images, among, at least, more than the languages available at the moment, without any *a priori* knowledge. The other question that was to be answered is what makes code visually recognizable.

Bonifro et al. (2021) conducted an experiment using classifiers like *Convolutional Neural Networks (CNN)* and Transfer Learning. Three different CNN architectures were used, each being pre-trained for image recognition, with two having around thirty layers and the other having eight layers.

To start the model training, the classification layer of each CNN was replaced with another with one hundred forty-nine output neurons and then a two-step training procedure was applied to all CNN. In the first step, so that the training only affects the new head, the weights of the body are frozen. The reasoning behind it is to maintain the features previously learned by the CNN and be able to make predictions about the new classification task.

After some training, the second step is started. At this point, the weights are all unfrozen, making way for training updates all over the architecture.

A precision and recall of 92% were achieved and, with this great result in mind, evidence on what identifies a language the most was also gathered, being symbolic characters like punctuation, arithmetic operators and parentheses contribute the most to the visual recognition of a programming language.

2.4.2 *Algorithmic Classification*

The main purpose of this type of classification is to identify the programming language in which a program was written, without any starting information besides the code itself.

Klein et al. (2011) developed research on algorithms to recognize and identify correctly code from as many programming languages as possible, being the code given the only available information.

After gathering a large database of source code, with as much representation as possible from each programming language, the algorithm was trained with code fragments in identified languages.

In this research, statistical analysis and a grammatical approach were experimented with, however, the grammatical approach couldn't be fully implemented. Since comments and strings in any language might contain arbitrary information, it is crucial to disregard them while training and identifying the algorithm. The algorithm applies a simple heuristic that is, comments and strings are parts that will contain natural language, so, naturally, will often have spaces between alphabetical words. Should a sequence of alphabetical-only words be separated by spaces and ending in a space or newline and its *words property* will be at a line of code defined for this purpose only. To begin, the algorithm starts by identifying lines most likely to contain a comment or string, which will be called *candidate lines*. The position of the string that matched the *words property* is also recorded for each *candidate line* in addition to the line number, with this being called *capture*. In case a *candidate line* has more than one *capture*, the *capture* with more words is used. After every *candidate line* was analyzed, cases where the lines matched are removed from future consideration.

Regarding the statistical analysis, pieces of source code were evaluated based on specific features and the results obtained are compared to those of each language in the database. The features evaluated were brackets, to evaluate the prevalence of different kinds of brackets, the first and last word in each line of code, to find common prefixes and characters that are commonly used in a line end, keywords, operators, punctuation and comments and strings. Afterwards, for each language and feature a score is calculated and the higher the score, the higher the probability the language is a correct match to the source code.

The program ranks languages from most to least likely and using 25 source code files the results were as follows:

- 48% of the fragments language were correctly identified;
- 12% of the fragments language were identified as the second most likely language;
- 12% of the fragments language were identified as the third most likely language;
- 4% of the fragments language were identified as the fourth most likely language;

- 12% of the fragments language were not identified as one of the five most likely languages;

2.4.3 *Syntax Oriented Classification*

Syntax highlighters identify the languages in question with different techniques. Some ask the user to indicate the name of the programming language, which makes it a very manual process, others use the success rate of the highlighting process to determine the language and some highlighters generalize the highlighting so that it works with all languages.

The absence of a genuinely automatic and reliable system underscores the imperative to develop one.

Zevin and Holzem (2017) believed the programming language identification should be evaluated on its accuracy, popular programming language support, extensibility in a reasonable amount of effort and time, independence of file extension and the time required for identification.

The chosen approach by Zevin and Holzem (2017) was to automatically derive a single grammar from training data. This was the preferred approach, not only because it would be impractical to obtain a grammar for every language and, as older languages evolve and newer ones appear, the maintenance cost of the multiple grammars would be very high. What makes this approach possible is that programming languages share a common syntax structure.

After building and running the grammar, the most representative productions are selected to be used as features by the language classifier allowing the classifier to be trained.

In the end, this method achieved an accuracy of 0.99, measured with thousands of source files, supports the 29 most popular programming languages, is fully automated, doesn't rely on the file extension and averaged 0.1 seconds per identification.

3

PROPOSED APPROACH

In this chapter, a proposal for the project will be presented and explained.

3.1 STUDY DIAGRAM

As this Master's project has a strong theoretical component, a study diagram has been developed, as can be seen on Figure 1.

In order to achieve the main goal of this Master's work, to start this study, it is crucial to choose what programming language characteristics should be studied and after determining what characteristics are most important, select the programming languages to be analyzed.

Due to the abstract nature of determining what characteristics identify a programming language, a survey was created. This survey involved different programming languages' code snippets and the participants were asked to, not only guess what programming language is present in each snippet, but also to state what made them give that answer. Afterwards, the results were analyzed, stored and conclusions were drawn.

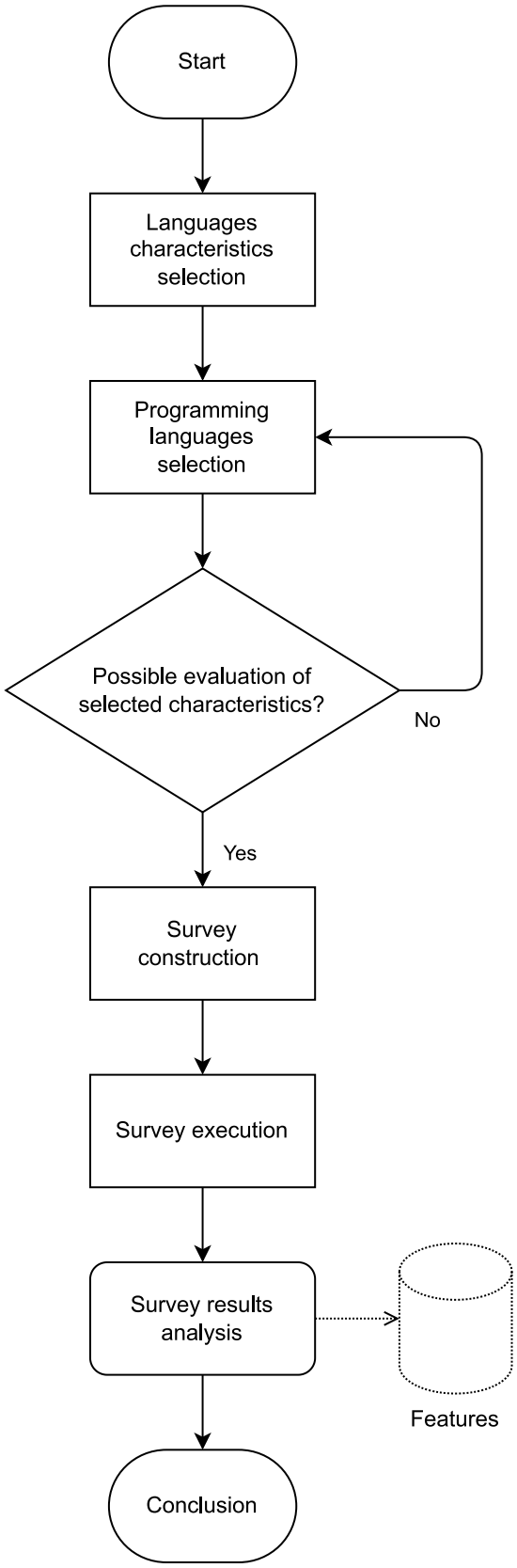


Figure 1: Theoretical study execution diagram.

3.2 SYSTEM DIAGRAM

The system developed to implement the conclusions drawn from the study, has the behaviour depicted in Figure 2:

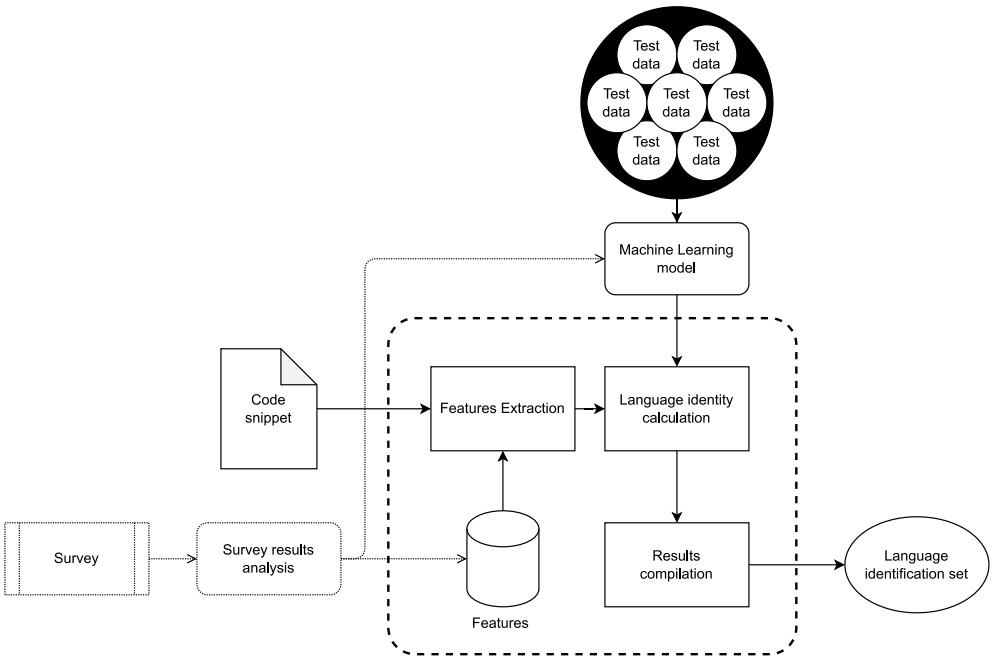


Figure 2: Architectural diagram for the system implemented.

After extracting the relevant features from the survey conducted, they are stored in a proper repository and are also used to train a Machine Learning model. From that basilar step on, the behavior of the identification system desired is described by the following steps:

1. A code snippet will be given to the system;
2. After receiving the snippet, its code features will be extracted. The features to be extracted are known from the results of the survey;
3. Having the snippet features extracted, an algorithm to calculate the language identity is used, with the assistance of the Machine Learning model, which has been previously trained with other code snippets and also been fed the features extracted from the survey;
4. The results are compiled and a Language Identification set is presented to the user.

Actually, the system projected above and planned at the beginning of the Master's work has not been developed, due to time and hardware constraints. However, instead of a direct implementation of the system, automatized recognition tests have been made using a

pre-trained machine learning model, available on https://huggingface.co/huggingface/CodeBERTa-language-id.

4

DEVELOPMENT

In this chapter, the different phases of the project's execution will be approached and every step of their development will be explained.

4.1 SURVEY

As per Alves et al. (2023), a crucial part of this Master's work consisted of developing and implementing a survey. The survey was sent to people who have knowledge of programming languages, whether experienced software engineers or university students. With this survey, it was expected to gather the different perspectives a programming language user can have, regarding the identification of programming languages.

4.1.1 *Survey Construction*

As stated previously, a survey was built to gather different opinions. While planning the survey, it became evident that certain aspects couldn't be ignored:

- Length – As expected, this survey has been distributed to as many people as possible, so caution was had not to create a lengthy survey, to avoid discouraging possible respondents and get the more accurate answers;
- Snippets to include – To complement the previous aspect, for the survey to achieve a greater range of answers and the most complete results, a strategy was elaborated to know what kind of snippets to show.

It is also important to mention that, while not directly applied, the general concepts behind Value-Focused Thinking Neto et al. (2022) founded the rationale throughout the construction of the survey, as was the case with previous studies Neto et al. (2021).

It was also decided that the languages to be studied on the survey are the languages characterized on 2.2.

4.1.2 *Survey Structure*

It was important to test different scenarios, such as language identification, language comparison, and language breakpoint, which led to structure the survey into four sections, with the other section responsible for assessing the respondents' languages knowledge.

Since the survey contemplated multiple programming languages and the target audience may have various degrees of knowledge of these languages, it becomes critical to understand how to weigh one's answers. For this reason, the survey started with a question on the respondents' familiarity with the multiple languages, as shown in Figure 3.

Figure 3: Survey's first question, designed to evaluate the respondents' knowledge levels of each language.

The respondent could give one of five answers to each language, to evaluate how knowledgeable he or she is of each, ranging from complete ignorance to profound knowledge.

The next section of the survey evaluated if the respondent could identify each language on a sequence of snippets while being asked to justify his or her answer. The purpose of this section was to understand the reasoning behind the identification of a programming language and to comprehend if multiple people use the same thought process, namely if in the snippet there is a distinctive feature. An example of a snippet shown on this section can be seen on Listing 4.1

```c
int main()
{
    if( n <= 1 || (n = atoi(argv[1])) <= 0 ) n = 8;
    int hist[n];
    solve(n, 0, hist);
    return 0;
}
```

Listing 4.1: C language snippet used in the second section of the survey.

Using the same strategy as for the first question, for each language the user can give one of four answers: Doesn't know, it sure isn't, maybe and absolutely is.

For the following section, the identification of programming languages took a different route. In this section, the user was presented with two code snippets written in different languages that solve the same problem. The reasoning behind this section was that, when facing two very similar code snippets, confusion would be induced in the respondents, forcing them to give more complete justifications on what languages were presented. It would also provide vital information on what characteristics make a language standout.

As per the previous section, and said previously, it was not only asked for the user to identify the programming language of each snippet but also to explain the thinking behind the given answer. The following listings, Listing 4.2 and Listing 4.3, were shown in one of the questions present in this section.

```java
class FileIOTest {
    public static void main(String[] args) throws Exception {
        var lines = Files.readAllLines(Paths.get("input.txt"));
        Files.write(Paths.get("output.txt"),lines);
    }
}
```

Listing 4.2: Java snippet used in the third section of the survey. This snippet was presented in the same question as listing 4.3 for comparison.

```c#
class FileIOTest
{
    public static void Main(string[] args)
    {
        var lines = File.ReadLines("input.txt");
        File.WriteAllLines("output.txt",lines);
    }
}
```

Listing 4.3: C# snippet used for comparison with the one presented in listing 4.2, in the third section of the survey.

The last section presented respondents with a starting code snippet in one programming language that is progressively modified until it becomes a substantially different language.

This section was given this structure because, it was believed, before distributing the survey, that respondents wouldn't say the language stopped being identifiable on the first iteration of modifications, which led to making the modifications progressive. This would, in theory, determine the exact breaking point of language identification and what characteristic is the most important.

With this being a particularly important question, but also one very difficult to gather results from, a strategy must be implemented in the different iterations to better evaluate what characteristic is more important to each language. As an example, in the snippet shown in Figure 4, the first iteration modified the data types. In the second iteration, the modification was made to the standard functions (`printf` in this instance). For the final iteration, the way a function is declared is changed, removing the data type to be returned and removing the necessity to state the data type of the function arguments (dynamic typing).

```c
int main(int argc, char *argv[])
{
    char t[255]="alphaBETA";
    str_toupper(t);
    printf("uppercase: %s\n",t);
    str_tolower(t);
    printf("lowercase: %s\n",t);
    return 0;
}

```

(a) Original version.

```c
int main(int argc, string argv)
{
    string t="alphaBETA";
    str_toupper(t);
    printf("uppercase: %s\n",t);
    str_tolower(t);
    printf("lowercase: %s\n",t);
    return 0;
}
```

(b) Different basic type (`string`).

```c
int main(int argc, string argv)
{
    string t="alphaBETA";
    str_toupper(t);
    write("uppercase: ",t);
    str_tolower(t);
    write("lowercase: ",t);
    return 0;
}
```

(c) Different standard functions.

```c
main(argc, argv)
{
    t="alphaBETA";
    str_toupper(t);
    printf("uppercase: ",t);
    str_tolower(t);
    printf("lowercase: ",t);
    return 0;
}
```

(d) Different type declaration (dynamic).

Figure 4: Four progressively altered snippets of C code used to establish what is the breaking point for the identification of a programming language.

4.2 SURVEY RESULTS ANALYSIS

The survey was available between April 12, 2023, and April 30, 2023. The results were analyzed using answers taken from a class of fourth-year software engineering students. A small number of individuals, including university professors and software developers, were also invited to respond; however, the poll was not made public, since the results and input from students had to be first carefully analyzed and discussed to make required modifications for a broader dissemination.

The respondents were chosen based on their knowledge of computer programming languages. The bulk of responders were Master's Degree students enrolled in a Languages Engineering programme at the University of Minho (UMinho). The questionnaire was completed by 44 participants, 39 of them were students. The other five individuals were professors from the Department of Informatics at UMinho.

To begin the examination of the findings, it was necessary to start with the first question, which demonstrates how well the respondents know the six chosen languages. The most well-known languages were JAVA, C, and PYTHON, with 95%, 88%, and 93% of respondents claiming to be at least competent enough to use them effectively. These findings indicate that inquiries about JAVA, C, and PYTHON can provide a better grasp of what distinguishes these languages. In comparison, only 47%, 45%, and 31% of respondents said they were at least competent in HASKELL, C++, and C#, respectively.

4.2.1 *Second Section: Language Identification*

Regarding `C`, respondents successfully identified it, stating the main reasons to its identification were:

- Variable declaration syntax;
- Use of pointers;
- Syntax features like semicolons, brackets to delimit code blocks and functions signature;

Like `C`, `Java` was also correctly identified, with the main reasons being:

- Methods access modifiers;
- Methods signatures;
- Variables types;
- The usage of `System.out.println`;

The language featured in the third sample was C#. As can be seen from the first question, respondents were unfamiliar with this language and there was a constant lack of confidence in determining whether the language was Java or C#. There were nine people who were certain it was Java, and nine who were certain it was C#. Twenty-four people indicated it may be Java, while twenty-one suggested it could be C#. There were fourteen people who thought it may be C++. However, it was agreed that it was neither Python nor C.

The respondents indicated a significant difficulty in identifying traits that allowed them to differentiate between Java and C#. The respondents who correctly identified it as C# stated that the adoption of the PascalCase naming convention on identifiers convinced them. Given that it reflects a writing style rather than a language feature, it is acceptable to assume that the identification was a result of the language's conventions, rather than its formal definition.

The fourth sample was an excerpt of C++ code. The audience agreed that this sample was not written in C, Java, nor Python. There was some doubt if it might be C#, because respondents identified this language as a derivation of C's syntax. Nonetheless, there was a high degree of agreement in recognizing the language as C++ because of:

- The usage of the `std` library;
- `Unsigned long long` type;
- The method or member function call notation.

Python was presented in the fifth snippet of this section. It was clear to the audience that this was Python because no other language had a good response and Python was chosen as the correct language by one hundred percent of the users. What prompted their responses was:

- Block syntax and indentation;
- The use of `def` when defining a function;
- Type inference.

For the last question on this section, respondents were finally presented with a Haskell code snippet. Once again, the respondents were certain it couldn't be any other language other than Haskell. The reasons given for this were:

- Functional programming style;
- Point free syntax, a style of writing Haskell code that avoids explicit mention of the arguments of a function;
- Function signatures;
- The complete overall difference to the other languages.

4.2.2 *Third Section: Language Comparison*

Java vs C#

Two comparisons were performed between these two languages since Java and C# are very similar, and certain challenges were expected. The respondents accurately identified the languages in the first comparison, citing the variation in the naming convention as the main contribution to the identification (as they did in the previous section). Respondents also cited the way exceptions are handled in Java as a distinguishing feature. However, there was significant ambiguity on the C# snippet in the second comparison, with respondents being mostly divided between C#, C++, and Java. There were also other responses that raised the possibility for it to be C or another language. This indecision is due to respondents not being able to associate the types ulong and uint to a language. Java's identification was very straightforward to the respondents, stating the use of the keyword final as the main reason for identification.

C vs C++

In this comparison, the respondents were consistent, with virtually no problems to the correct identification of each language. However, only two reasons were given: the standard output functions (printf for C and cout for C++) and the use of namespace on C++.

C vs Java

Once again, there was no doubt among the respondents on what language was present in each snippet. Respondents stated that the standard output functions (printf and System.out.println) were the main reasons for their correct identification. There were also a few answers pointing to the way arrays are declared as a contributing factor.

4.2.3 *Fourth Section: Language Identification Breaking-point*

Respondents for C were convinced that the language lost its identity after the first excerpt. The change that prompted this perspective was the replacement of char * with a new type string, which is frequently used in other languages. With C++, the replies also leaned for stating that the language could not be C++ after the first excerpt. The change in how libraries are imported, from namespace and #include <libraryName> to with, was the defining change that led to the breaking point.

Respondents agreed on the first sample being C# and were open to the notion of it being C# on the second snippet. Because the output function was changed from the first to the second sample, it is reasonable to conclude that it is not a very distinguishing feature of

the language. However, it became clear to the respondents after the third snippet that the language was not `C#`. From this snippet on, the way code blocks are constructed changed from brackets to indentation, like in `Python`.

Curiously, the exact same reaction was obtained with `Java`. On the second snippet, the code blocks syntax was modified from curly brackets to indentation, triggering the respondents to not identify the language as being `Java`.

Respondents were receptive to `Haskell` and `Python` being the original languages until the function declaration syntax was changed. The change in `Haskell` was to replace the normal syntax `functionName :: argument -> argument -> result` with `functionName(argument, argument)`. `Python`'s change was minor, consisting of only changing the phrase `def` to `function`.

4.3 Machine Learning Model Application

As an addition to this experiment, given that the survey only allows access to a limited number of code snippets and was not answered by a significant amount of people, a Machine Learning model was used to complement the results found through the survey. As this model was found on the web, the languages it has been trained on are not the exact same as the ones in study.The model has been trained on six languages, with only two (Java and Python) being the same as the languages tested on the survey. The other languages are Go, JavaScript, Ruby and PHP.

Comparisons were made between the survey findings and the results produced using the model on specific code fragments. It must be taken into account that, despite not all six surveyed languages were trained by the model, it is expected that C#, C++, C and, obviously, Java, code snippets are given a high resemblance to Java by the model, due to all these languages being descendants of C and having similar syntactical features. Obviously, it is also expected that the model correctly identifies Python code snippets, as the model was trained on it.

Haskell stands out as a programming language that doesn't rely on traditional code block structures like many other languages do. When evaluating code snippets, Haskell's unique approach can offer valuable insights into how different languages are identified. For instance, if the model assigns a high score to Java, it suggests that the presence or absence of code blocks might not be as critical in distinguishing programming languages.

All the snippets from the second, third and fourth sections of the survey were tested on the Machine Learning Model, in order to evaluate its capacity of identification and how both results were similar.

4.3.1 *Second section: Language identification*

The expectations were met for the snippets of this section. The model returned very high scores to the snippets that the survey respondents identified as C, C#, C++, and Java, ranging from 99.8% to 100%. As a result, it is reasonable to believe that the model had little uncertainty about which language these samples resembled the most. The model was also on par with the respondents on the snippet that was written in Python, with a score of 100%.

The survey respondents were certain that the Haskell snippet was written in Haskell, and they were not open to the possibility of it being written in another language. The model, on the other hand, was less certain of the language, and gave JavaScript a score of 80.8%, Python 12.7%, Ruby 4.6%, and PHP 1.2%. Java and Go received scores of nearly 0%.

4.3.2 *Third section: Language Comparison*

`Java` *vs* `C#`

In the first comparison, the respondents were unanimous in identifying the languages, and so was the model, with the Java excerpt receiving a score of 100% and C# receiving 99.8%. For the respondents, the second comparison between these languages raised some doubts, unlike the model, which repeated the same scores for each language.

`C` *vs* `C++`

In this particular case, the respondents successfully identified the languages as C and C++ while the model, despite still in accordance with what was expected, presented some uncertainty. The excerpt written in C was given a 91% score to Java and 9% to JavaScript. As for the one written in C++, a bigger discrepancy was found. While Java kept the highest score with 56%, JavaScript also received a high score with 40%, while Ruby received 3%.

`C` *vs* `Java`

With this comparison, the expectations for the model results were met. Like the respondents, there was no doubt for the model. The C snippet received a score of 99.6% in Java and the Java snippet received, with no surprise, a score of 100%.

4.3.3 *Fourth Section: Language Identification Breaking-point*

For the C cases, the model maintained the same position that the snippets were Java up until the last snippet, which represented an abrupt change of results, with JavaScript getting a score of 68.2%, Java a score of 17.5% and Python 10%.

Just like in the C case, the C++ breaking-point analysis only got a change in the model results for the last snippet, with all the previous snippets having Java as the undisputed language. In the last snippet, however, Ruby became the language with the highest score with 41.6%, followed by Java with 27.3% and JavaScript with 24.7%.

Regarding C#, the model couldn't identify any change in the snippets that made it doubt which language it was, with all the snippets receiving scores of nearly 100%.

The model response to Java snippets, unlike in the previous examples, actually changed on the third snippet, going from a 100% score in Java to 12%, which can be seen on the appendix 14c. In the last snippet, the model gave an even higher score to JavaScript, with the language now receiving a score of 99%.

In the Haskell case, the model wasn't able to find a breaking-point in the language identification, with all the snippets receiving increasingly higher scores in Java, going from 93.8% in the first snippet to 99.2% in the last one.

For Python, the model also identifies a breaking point on the third snippet, going from a score of 99.8% in Python to a score of 97% in JavaScript for the last two snippets.

4.4 Machine Learning Model Exploration

The following sections present a series of tests conducted on the languages the model has been trained on, as a means to evaluate what characteristics identify a language. Obviously, a structure had to be developed to get the most out of the tests. To start, and to have a control group, every language had tests made to an untouched code snippet written in the language under study. Afterwards, tests were divided in modifications regarding:

1. Code blocks
2. End of statement
3. Type annotation
4. Function or method signature
5. Function or method call
6. Naming style
7. Standard Library

8. Entry Point

4.4.1 *Python*

As can be seen in Figure 5, starting on the original code snippet, the model identified Python with a very high degree of confidence in almost all modifications.

From all the conducted experiments (Appendix D.1), the ones that made the model change opinions were:

- Code block Symbol Delimitation: The model provided a wildly different set of results, giving Python a score of 3.6%, as seen in column named *Symbol Delim.*, with JavaScript receiving 90.5% and Ruby 5.6%;
- Code Block Suffix keyword: In this situation, the model was certain the language in question was Ruby, giving it a score of 99.6%, with Python score being negligible, as seen in column named *Suffix Keyword*;
- Code block prefix and suffix keywords: This modification gave very similar results as the previous modification, with Ruby being the language with the highest score, being 99.8%, with Python score being, once again, negligible, as seen in column named *Prefix & Suffix Keyword*;
- End of statement Symbol Based: Despite the model still considering Python as the most probable language, giving it a score of 65.1%, as seen in column named *Symbol Based*, JavaScript received a higher score, in comparison with the original code snippet, with it being 27.7%. Less significant but also relevant, Ruby was identified as the third most probable language, with a score of 6.1%.

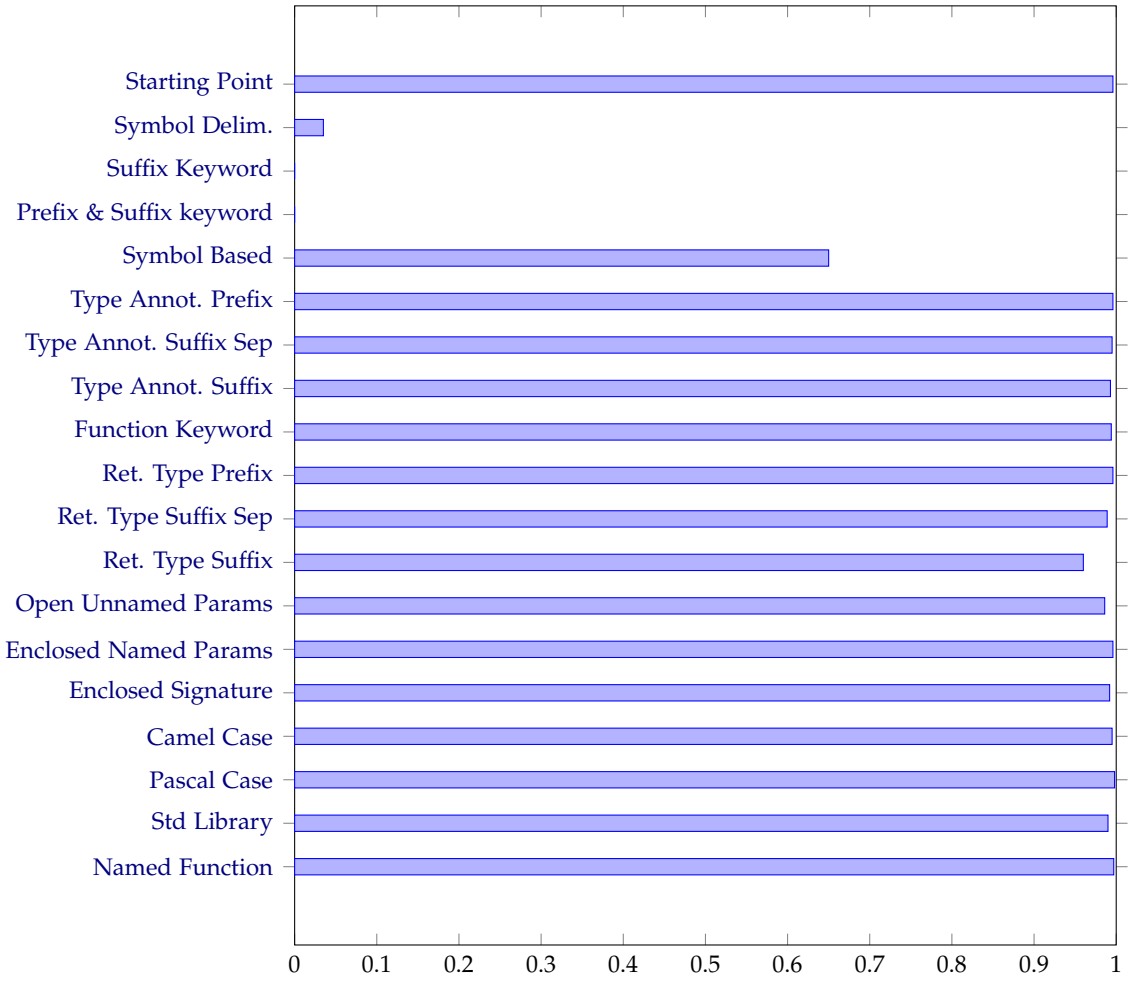


Figure 5: Python score on each test.

4.4.2 *Go*

As can be seen in Figure 6, on the starting point, the model successfully identified the language as Go, however, it only gave Go 65%, with JavaScript receiving a score of 34.6%.

From all the conducted experiments, that can be seen on the appendix D.2, the ones that made the model change opinions were:

- Code Block Indentation: The model didn't give any language unanimous voting, with Python being the language with the highest score of 41.5%, JavaScript getting 26.4%, Go 21%, as seen in column named *Indentation*, and the last language with a relatively high score being Ruby with 10.9%;
- Code block suffix keyword: In this case, the same happened, with no language having more than 50%. This time Ruby is the language with the highest score, receiving 42.6%,

JavaScript 38.5%. Go was the third-placed language, getting 18.5%, as seen in column named *Suffix Keyword*;

- Code block prefix and suffix keyword: This time, the model identified a language with a very high degree of certainty, with Ruby having a score of 93.5% and the second placed language, JavaScript, receiving just 5.5%. As seen in column named *Prefix & Suffix Keyword*, Go score was negligible;
- End of Statement symbol based: Once again, the model identified a language with a very high degree of certainty, with JavaScript receiving a score of 97.8%. Go is the second placed language and only receiving a score of 1.7%, as seen in column named *Symbol Based*;
- Definition keyword: JavaScript is the language with the highest score, with it being 78.6%. Go was the second placed language, with a score of 20%, as seen in column named *Definition Keyword*;
- Return type prefix: The same result order as the last case was obtained, however, the score of the top placed languages changed. In this occasion, JavaScript has a score of 86%. Go score went down to 13.6%, as seen in column named *Ret. Type Prefix*;
- Return type suffix with separator: A very similar result has the case where the modification was adding the `def` keyword, with JavaScript having a score of 74.7%. Go achieved a score of 23.5%, as seen in column named *Ret. Type Suffix Sep.*;
- Return type suffix: This time, the similarity in results is with the case where the modification was on the return type prefix, with JavaScript having a score of 83.5%. Go received a score of 16%, as seen in column named *Ret. Type Suffix*;
- Enclosed signature: In this case, JavaScript received a score of 84.4%. Go was once again the second placed language, with a score of 15.2%, as seen in column named *Enclosed Signature*;
- Standard library: With this modification, JavaScript received a score of 83.3%. Go received a score of 16.2%, as seen on column named *Std Library*.

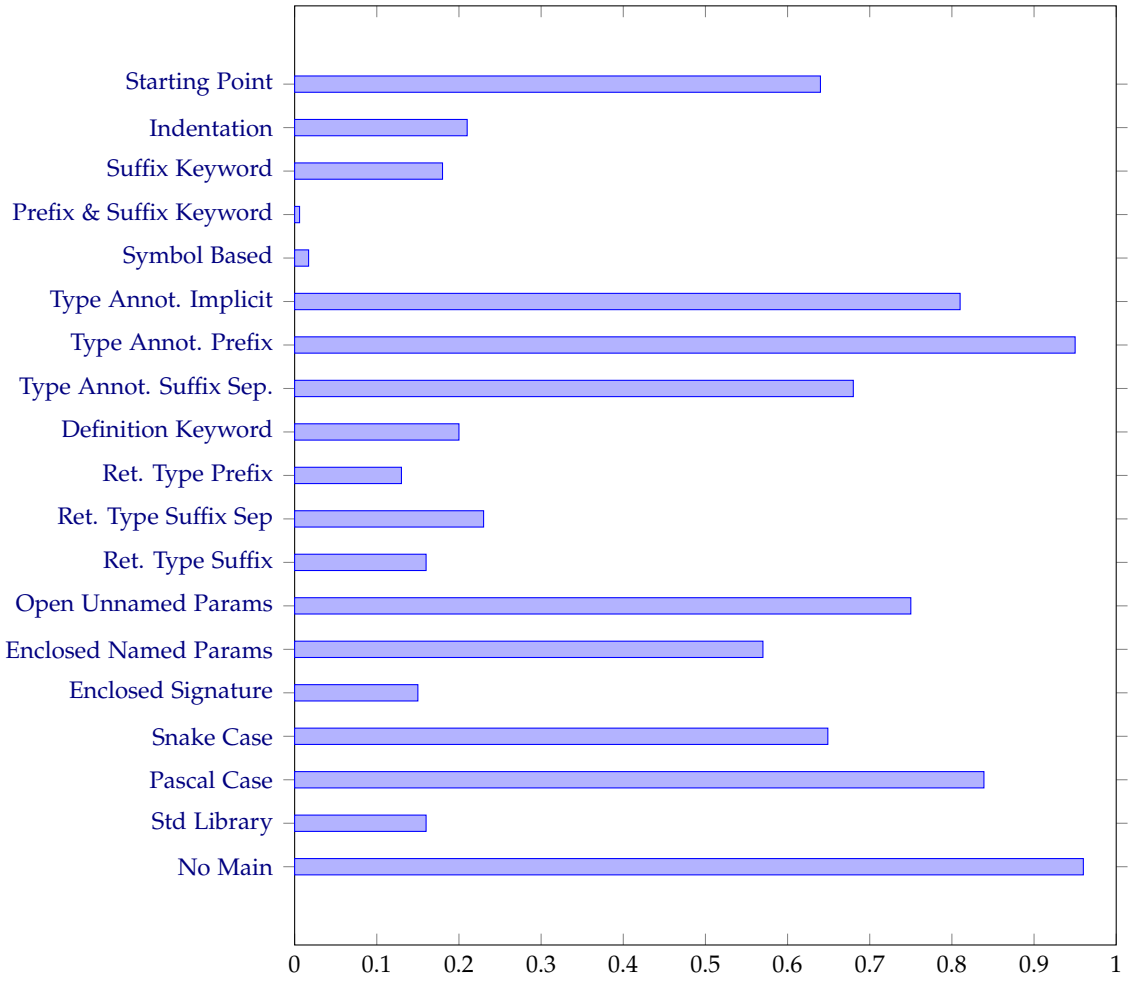


Figure 6: GO score on each test made

4.4.3 *JavaScript*

As can be seen in Figure 7, from the get-go, the model successfully identified the language as being JavaScript, giving it a score of 97%.

Unlike the other languages, in no situation JavaScript left the first place. However, in some cases, its score dropped significantly. The cases were:

- Indentation: JavaScript dropped to 68.2%, as seen in column named *Indentation*, with Java increasing its score to 28.9% and Python to 2%;
- Suffix keyword on code block: This time, JavaScript got a score of 67.5%, as seen in column named *Suffix Keyword*, Java 26.3% and Ruby 6%, as seen on column named *Suffix Keyword*;

- Prefix and suffix keyword on code block: Once again, JavaScript score dropped significantly, this time to 70.5%, as seen in column named *Prefix & Suffix Keyword*, with Ruby raising to 15.6% and Java to 13.4%;
- Type annotation prefix: This was the case in which JavaScript score dropped the most, going down to 63.2%, as seen in column named *Type Annot. Prefix*, and Java going up to 36.2%.

All the tests made can be seen on the appendix D.4.

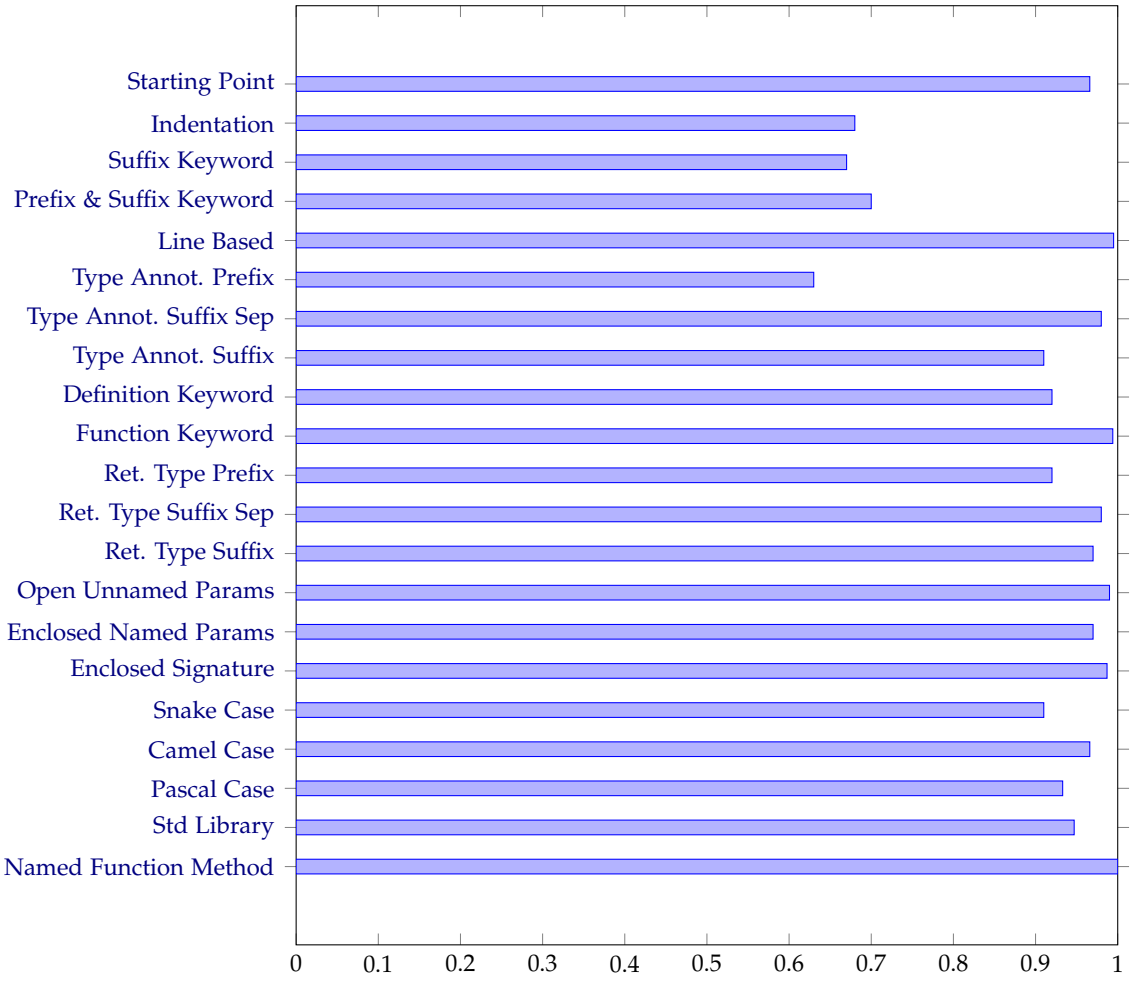


Figure 7: JavaScript score on each test made

4.4.4 *Ruby*

As can be seen in Figure 8, the model, since the beggining, had very high confidence when identifying the code as being Ruby, giving it a score of 100%.

In complement to this affirmation, there were only two cases in which the model stopped identifying the snippets as being Ruby. The cases were:

- Code Block Symbol delimitation: In this snippet, the model gave JavaScript a score of 83.65%. As seen in column named *Symbol Delim.*, Ruby got a score of 16%;
- Indentation: This time, the language with the highest score was Python, with a score of 58.4%. Ruby got a score of 41%, as seen in column named *Indentation*.

All the tests made can be seen on the appendix D.5.

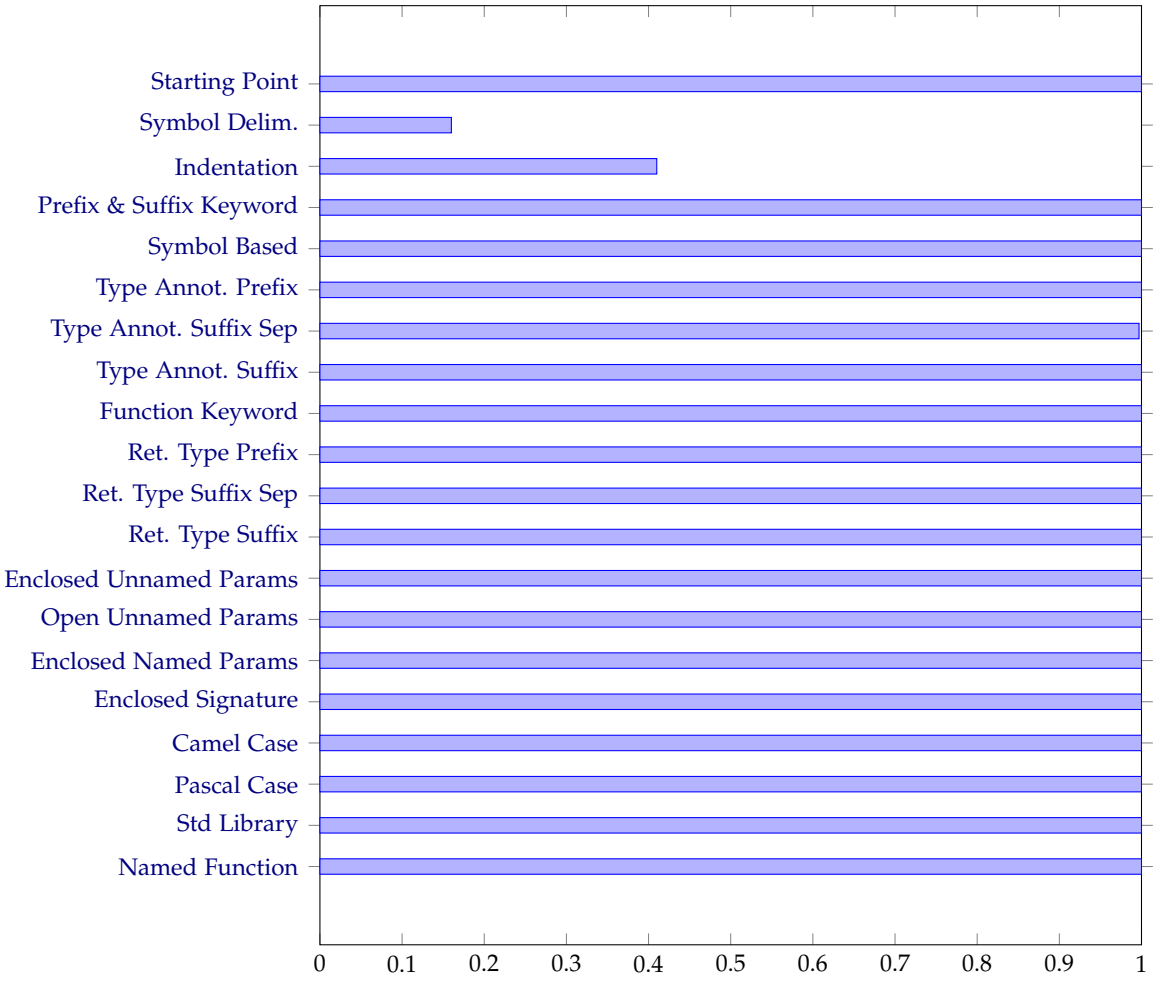


Figure 8: Ruby score on each test made

4.4.5 *PHP*

As can be seen in Figure 9, with PHP, the model revealed some uncertainty, giving the language 72% on the starting point snippet.

After all the modifications, that can be seen on the appendix D.6, there were some cases in which the model stopped identifying the snippet as being PHP:

- Prefix and suffix keyword on code block: In this snippet, the model considered it was more probable the snippet had been written on JavaScript, as it gave the language the highest score of 48.8%, followed by PHP with 35.5%, as seen in column named *Prefix & Suffix keyword*, and Ruby with 15.4%;
- Type Annotation suffix with separator: Once again, JavaScript was the language with the highest score, getting 61%. PHP was second place with 37.6%, as seen in column named *Type Annot. Suffix Sep*;
- Enclosed named parameters: This time, the same placement was obtained, with JavaScript getting a even higher score of 67.7%. PHP received 31.6%, as seen in column named *Enclosed Named Params*;
- Enclosed signature: JavaScript score once again increased to 74.4%. Inversely, PHP score decreased to 25.2%, as seen in column named *Enclosed Signature*;
- Camel case: Despite the same placement as the previous cases, JavaScript decreased to 59.1%. PHP increased to 40.4%, as seen in column named *Camel Case*;
- Standard library: A very similar result as the previous case was obtained, with JavaScript receiving a score of 58%. PHP achieved a score of 41.4%, as seen in column named *Std Library*.

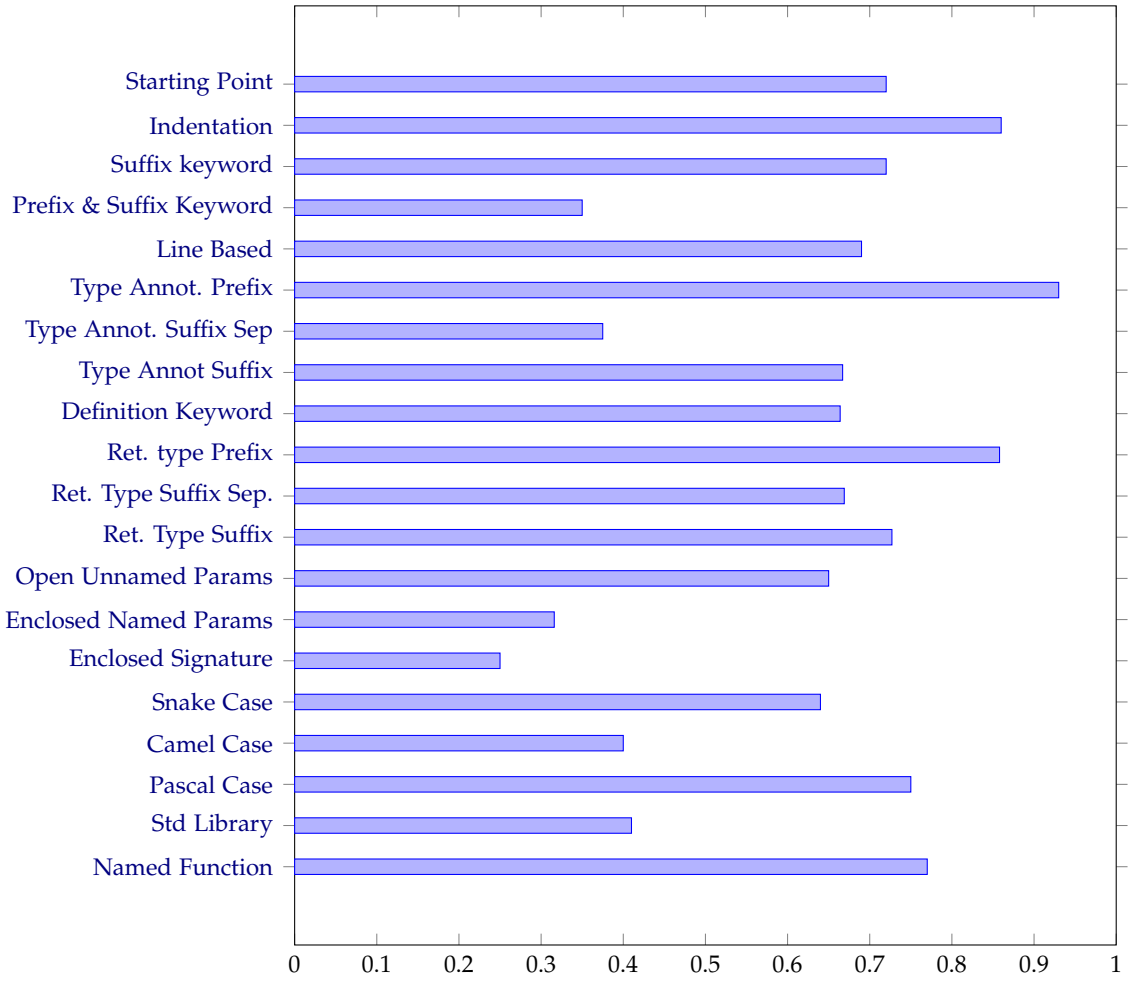


Figure 9: PHP score on each test made

4.4.6 *Java*

As can be seen in Figure 10, the model had no doubt in identifying Java as the language present on the starting point snippet, receiving a score of 100%.

As was already seen previously, the model always had some cases in which modifications to the original code snippet resulted in a change of results. However, in Java, no matter which modification was made, the model always gave it a score of 100%. The tests made can be seen on the appendix D.3. For this reason, a new set of tests was conducted to try and identify something that disrupted the identification of the language, as can be seen on appendix D.3.1.

In this new set of tests, only one case provided a different set of results. Unlike all other cases, in which only one modification was made in relation to the original code snippet, this snippet suffered significant modifications. The modifications were made concerning

the standard library methods, the classes, access modifiers, and types. These modifications resulted in the model classifying the snippet as JavaScript code, with a score of 97.4%. Java, as seen in column named *Acc., Cl., Fns, T.*, only received a score of 2.4%.

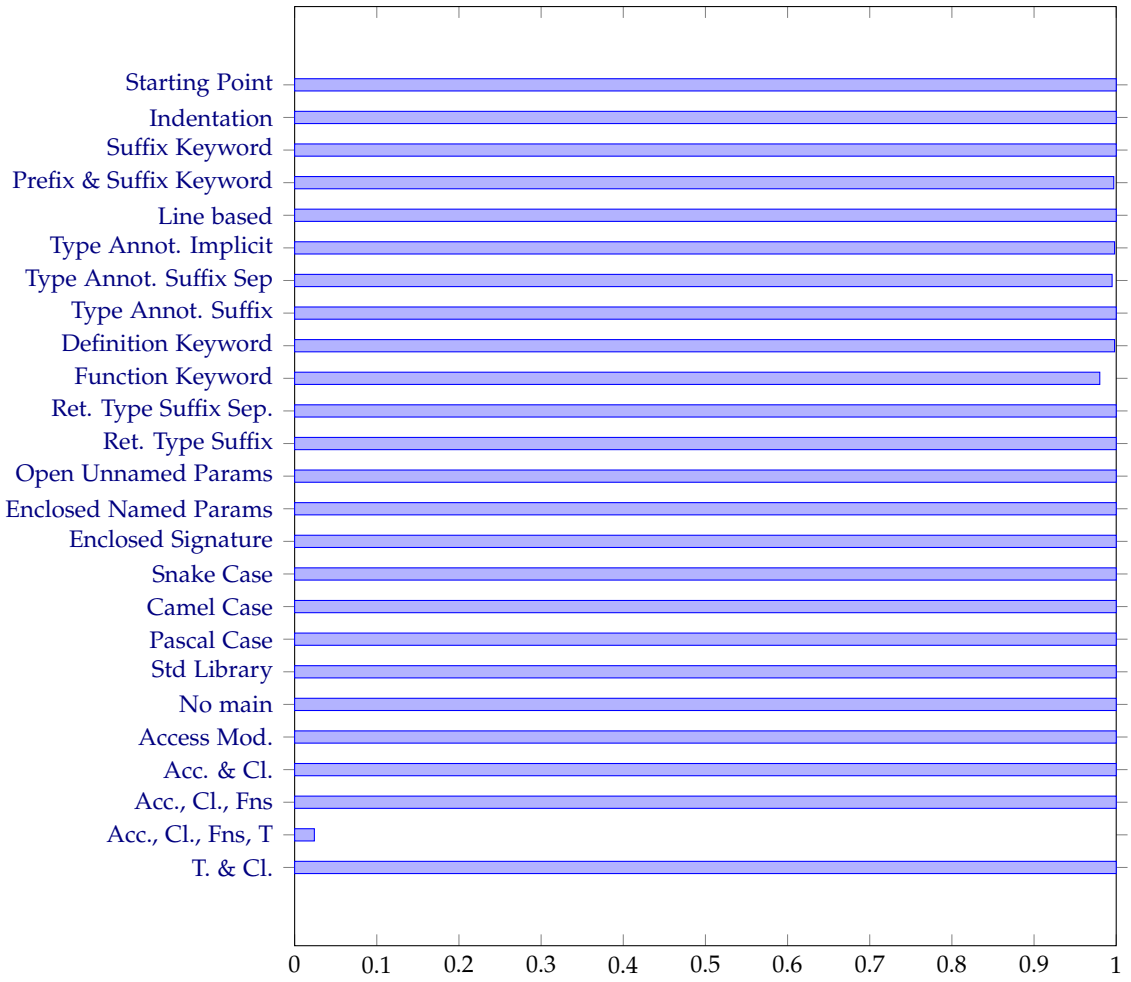


Figure 10: Java score on each test made

4.5 Results analysis

After gathering all the results, it is possible to verify that, in all tested languages, there were specific modifications that resulted in the model not being able to identify the snippet as being written in the original language.

The results indicate that the model's ability to identify programming languages from code snippets is sensitive to code structure, keywords, and symbols. Some modifications led to shifts in language predictions; in some cases, the model considered alternative languages with varying degrees of confidence.

Apart from Java and PHP, all languages suffered a significant drop in recognition when the modifications were made to the code block syntax, namely when adding a symbol delimitation or using indentation to delimit the code block. The code block modifications made to symbol delimitation, indentation, and prefix-and-suffix keywords all had a significant impact on language identification. Out of all the modifications, these were, undoubtedly, the ones that resulted in a more drastic difference. So, it is possible to infer that the code block structure has a great impact when identifying a programming language.

On the other hand, and contrary to all expectations, Java kept being identified without any uncertainty, whenever only one type of modification to the original code snippet was made. All these modifications can be seen on the appendix D.3. For this reason, it became crucial to expand the tests and, instead of only making one modification at a time, several modifications were increasingly compounded, as per the new section D.3.1 on the appendix. However, even then, Java was still given 100%. It wasn't until four modifications were made that a completely different result was obtained and the model reduced Java to a near insignificant score. In this particular case, the modifications involved the removal of access modifiers, classes, standard functions, and types. As it is clear, the language was stripped of a lot of its elements.

Java's case may be explained by its notable differences from the other languages known by the model. It is very verbose and purely object-oriented, unlike the other languages. This probably lead to a bias when identifying Java versus other languages.

It's essential to understand that programming language identification based on code snippets can be challenging, especially when snippets are short and lack unique identifiers. The model's sensitivity to specific code modifications highlights the importance of carefully crafting code features for accurate language identification. Also, the model has only been trained in six languages and its results were based solely on them.

5

CONCLUSION

The main goal of this Master's work was to determine which characteristics identify different programming languages. Although programmers can identify programming languages, most of the time, that challenge is made simpler due to the context in which they are involved.

As it is obvious, to be able to identify a programming language, it is crucial to understand what is a programming language. For this reason, an explanation of how programming languages are defined and processed, and the different phases of it, is given at the beginning of this dissertation.

Then, once it became clear the role of programming languages, it was also important to review the research already done on programming language identification, namely using Machine Learning. Given that Machine Learning is present on the approaches discovered in the literature reviewed, it was also important to discuss, and understand, the different learning paradigms.

Once the groundwork was established, an introduction to a selection of programming languages was done, enumerating their most important characteristics, to, later on, be able to know what characteristics must be tested on each language.

Before starting the tests, some planning had to be done, to achieve the best possible results. This led to the development of a study plan and a system diagram. The study diagram provided a clearer view of the direction that needed to be followed, to not lose focus on what was being tested. The system diagram was developed thinking on how an optimal tool to solve the Master's problem stated initially should look like. This tool should take the knowledge rules derived from the survey results, and incorporate them into a pre-trained machine-learning model. This would allow a user to provide a code snippet and receive a language identification set, where the model would give its verdict on what languages are the most probable to be present in the code snippet.

The fourth chapter addresses the most important aspect of this work. It begins with a discussion of the implemented survey, which resulted in a paper Alves et al. (2023). The development process of the survey is described, followed by an explanation of the survey's final structure. The section concludes with an analysis of the survey results.

Since the survey results were inconclusive, an alternative method had to be employed. As mentioned earlier, this alternative method involved using a machine learning model. The motivation behind this decision was to utilize a machine learning model equipped with the knowledge of multiple individuals, thereby eliminating the time constraints associated with a survey, where waiting for a sufficient number of responses is necessary.

However, it was crucial to assess if the model was capable of replacing individuals. So, the last sections of Chapter 4 were dedicated to exposing and discussing the exploration done by submitting the survey questions to the model. The outcomes produced by the AI model were compared with the results obtained in the survey.

After proving its capabilities, a new set of tests was made on the languages on which the model was trained.

With all this being said, it was possible to draw a set of conclusions to be made, using the good and bad results obtained from it.

Despite the survey not giving good enough results, it showed a lot of potential to learn the thought process used by a real person, when tasked with identifying a programming language. Nonetheless, it became evident that when trying to identify the most relevant characteristic of a programming language through modification of a code snippet until a language is no longer recognizable, these modifications can't be progressive. Progressive modifications can lead to confusion among respondents and introduce overhead for all evaluated characteristics after the first. When introducing a second modification, it isn't only assessing how relevant that characteristic is, but it sums up the relevance of both characteristics. This can cause false positives. This can also be difficult to implement in a way respondents understand what is trying to be achieved.

In a positive regard, it was possible to check the importance of code blocks when identifying a programming language, and also how difficult it is to identify a key characteristic in Java. This is a critical objective that was attained during the Master's work conducted and guarantees a good starting point for future iterations.

5.1 Future work

Identifying programming languages is very subjective and full of nuances, which makes it unfeasible to find a definitive answer to this question on a single iteration. However, with this project, the groundwork has already been done and it is possible to determine what should be the following steps:

- **Redo the survey** – Although its results proved inconclusive, the survey is a very important method to help identify the characteristics of a programming language, as it gives first-hand answers from real people, main targets of this work. A new version of the survey should fix the discussed problems, namely in the last section; be

implemented in a structure in which the respondents only have to answer questions of languages they claimed to know; be more extensive; and be distributed to a wider range of respondents;

- **Creation of a machine learning model based on the survey findings** – This was a planned step that couldn't be completed due to multiple constraints. Nonetheless, it became clear in the second part of this project that an AI-powered model can effectively identify programming languages, given its training process. The model to be constructed should be trained with the biggest number of programming languages possible, to get the most complete results;
- **Test with bigger code snippets** - Given that the snippets used were as minimal as possible to remove noise, the identification of the languages was made easier. Introducing bigger and more diverse snippets should create a layer of difficulty and, hopefully, give a more complete answer.

BIBLIOGRAPHY

Java documentation. https://docs.oracle.com/en/java/. Accessed: 2023-10-30.

Python documentation. https://www.python.org/doc/. Accessed: 2023-10-30.

Sam A Abolrous. *Learn C-Sharp - Includes the C-Sharp 3.0 Features*. Wordware Pub, 2007. ISBN 9781598220353; 1598220357.

Alfred V.; Monica S. Lam; Ravi Sethi; Jeffrey D. Ullman Aho. Compilers: principles, techniques & tools. 2007.

Júlio Alves, Alvaro Costa Neto, Maria João Varanda Pereira, and Pedro Rangel Henriques. Characterization and Identification of Programming Languages. In Alberto Simões, Mario Marcelo Berón, and Filipe Portela, editors, *12th Symposium on Languages, Applications and Technologies (SLATE 2023)*, volume 113 of *Open Access Series in Informatics (OASIcs)*, pages 13:1–13:13, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-291-4. doi: 10.4230/OASIcs.SLATE.2023.13. URL https://drops.dagstuhl.de/opus/volltexte/2023/18527.

Francesca Del Bonifro, Maurizio Gabbrielli, Antonio Lategano, and Stefano Zacchiroli. Image-based many-language programming language identification. *PeerJ Computer Science*, 7, 2021.

Robert W. Floyd. The syntax of programming languages. 1964.

James Gosling, William N. Joy, and Guy L. Steele. The java language specification. 1996.

Vincent Kanade, Varun Mallmann-trenn, and Frederik Mathieu. Clustering algorithms. *Wireless RF Energy Transfer in the Massive IoT Era*, 2021.

Jyotiska Nath Khasnabish, Mitali Sodhi, Jayati Deshmukh, and Gopalakrishnan Srinivasaraghavan. Detecting programming language from source code using bayesian learning techniques. In *MLDM*, 2014.

David Klein, Kyle Murray, and Simon Weber. Algorithmic programming language identification. *ArXiv*, abs/1106.4064, 2011.

Miran Lipovaca. *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press, 2011. ISBN 1593272839; 9781593272838.

Tom Mitchell. Machine learning. 1997.

Alvaro Costa Neto, Cristiana Araújo, Maria João Varanda Pereira, and Pedro Rangel Henriques. Programmers' affinity to languages. In *ICPEC*, 2021.

Alvaro Costa Neto, Cristiana Araújo, Maria João Varanda Pereira, and Pedro Rangel Henriques. Value-focused investigation into programming languages affinity. In *ICPEC*, 2022.

Easy Programming. *C Programming Language The ULtimate Beginner's Guide*. CreateSpace Independent Publishing Platform, 2016.

Bjarne Stroustrup. *A History of C++: 1979–1991*, page 699–769. Association for Computing Machinery, New York, NY, USA, 1996. ISBN 0201895021.

Peter Norvig Stuart Russell. *Artificial Intelligence: A Modern Approach*. Prentice Hall Series in Artificial Intelligence. Prentice Hall, 3rd edition, 2010. ISBN 0136042597; 9780136042594.

Shaul Zevin and Catherine Holzem. Machine learning based source code classification using syntax oriented features. *ArXiv*, abs/1703.07638, 2017.

A

CODING LANGUAGE EXAMPLES

In this chapter, there are various programming languages code snippets displayed, where the main objective is to try to identify what language is present in the snippet. After identifying the language, it is expected that be explained what made one say it was that language.

A.1 EXAMPLE 1

```
int main(int n, char **argv)
{
  if (n <= 1 || (n = atoi(argv[1])) <= 0) n = 8;
  int hist[n];
  solve(n, 0, hist);
}
```

In this fragment, it is a displayed a C code snippet. Due to the use of function atoi and use of pointers, one can say confidently it is C.

A.2 EXAMPLE 2

```
private static void runTest(LoopTest loopTest) {
        List<Integer> values = new ArrayList<>();
        for (int i = loopTest.start ; i <= loopTest.stop ; i += loopTest.increment ) {
            values.add(i);
            if ( values.size() >= 10 ) {
                break;
            }
        }
        System.out.printf("%-45s %s%s%n", loopTest.comment, values, values.size()==10 ? " (loops
     forever)" : "");
    }
```

In this snippet, due to the use of System.out.printf, it is easily identified as Java, instead of other languages like C#. Other characteristics like using "new ArrayList<>()" and the method "size()" also help identify this snippet as Java.

A.3 Example 3

```
  public void Shuffle()
  {
        var random = new Random();
        for (int i = 0; i < cards.Count; i++)
        {
            int r = random.Next(i, cards.Count);
            var temp = cards[i];
            cards[i] = cards[r];
            cards[r] = temp;
        }
  }
```

In this snippet, it is C# because of naming a method in PascalCase "Next()", using Count in the for loop condition instead of length(), like it is used in Java and also because of a coding convention for C# programmers where the bracket is placed in a new line.

A.4 Example 4

```

entropy
  :: (Ord a, Floating c)
  => [a] -> c
entropy =
  sum .
  map (negate . ((*) <*> logBase 2)) .
  (map =<< flip (/) . sum) . map genericLength . group . sort

main :: IO ()
main = print $ entropy "1223334444"
```

For those familiar with Haskell, it becomes evident that's the language present for a variety of reasons, namely the function signature, from the use of :: to specify the function type, to the way the arguments are given, the characteristic type IO(), the use of $ to replace parenthesis, points to make a pipeline and common functions like map and flip .

A.5 Example 5

```
size_t popcount(unsigned long long n) {
  return std::bitset<CHAR_BIT * sizeof n>(n).count();
}
```

The use of characteristic types like size_t and unsigned long long makes it clear we are viewing a C++ snippet. Other thing that shows this is a C++ snippet is the presence of "std::bitset".

A.6 Example 7

```
def qsort(list):
    if not list:
        return []
    else:
        pivot = list[0]
        less = [x for x in list[1:]   if x <  pivot]
        more = [x for x in list[1:] if x >= pivot]
        return qsort(less) + [pivot] + qsort(more)
```

The presence of def when declaring a function, using : and indentation (although recommended for every language) to declare code blocks and the way list comprehension is made, makes it clear this is a Python code snippet.

A.7 Example 8

```
func main() {
    printGrid("puzzle:", puzzle)
    if s := solve(puzzle); s == "" {
        fmt.Println("no solution")
    } else {
```

```
        printGrid("solved:", s)
    }
}
```

The use of func when declaring a method, the use of := and the existence of fmt.Println indicates us this is a GO code snippet.

A.8 Example 9

```
fun main(args: Array<String>) {
    val s = "0123456789"
    val n = 3
    val m = 4
    val c = '5'
    val z = "12"
    var i: Int
    println(s.substring(n, n + m))
    println(s.substring(n))
    println(s.dropLast(1))
    i = s.indexOf(c)
    println(s.substring(i, i + m))
    i = s.indexOf(z)
    println(s.substring(i, i + m))
}
```

The use of val for read only variables, println without System.out like it is in Java, the way types are given to variables and the use of fun when creating a new function, tells us that this is a Kotlin code snippet.

B

CODE COMPARISON EXAMPLES

In this chapter, the objective is to, between two code snippets of functions that do the same thing in different languages, identify which language is present in each snippet and why.

B.1 COMPARISON 1

```
class FileIOTest {

    public static void main(String[] args) throws Exception{
        var lines = Files.readAllLines(Paths.get("input.txt"));
        Files.write(Paths.get("output.txt"),lines);
    }
}
```

```
class FileIOTest
{

    public static void Main(string[] args)
    {
        var lines = File.ReadLines("input.txt");
        File.WriteAllLines("output.txt",lines);
    }
}
```

The use of PascalCase when naming methods, including for the main method and the placement of brackets in new lines in the second snippet and the fact that exceptions are thrown in the method signature in the first snippet, tells us that the the first snippet is of Java and the second is C#.

B.2 COMPARISON 2

```
#include <stdio.h>

main(){
    int k;
    for(k=10;k>=1;k--){
        printf("%d\n",k);
    }
}
```

```
using namespace std;
#include <iostream>

main(){
    int k;
    for(k=10;k>=1;k--){
        cout << k << endl;
    }
}
```

The use of using namespace and cout « (..) « endl; in the second code snippet tells us this is C++ and, the use of printf in the first code snippet tells us this is a C code snippet.

B.3 COMPARISON 3

```
int sessionAttendees[5] = {75,63,103,87};
int totalAttendees = 0;
int i;

for( i = 0; i < 5; i++)
{
    if(sessionAttendees[i] > 0)
        totalAttendees += sessionAttendees[i];
}
printf("%d",totalAttendees);
```

```
int[] sessionAttendees = new int[]{75,63,103,87};
int totalAttendees = 0;

for( int i = 0; i < sessionAttendees.length ; i++)
{
    if(sessionAttendees[i] > 0)
        totalAttendees += sessionAttendees[i];
}
System.out.println(totalAttendees);
```

The use of printf and the way arrays are declared in the first snippet, tells us this is a C code snippet and, likewise, the way arrays are declared in the second snippet, the use of length in the for loop and System.out.println tells us this is a JAVA snippet.

B.4 COMPARISON 4

```
public static ulong Fib(uint n) {
    return (n < 2)? n : Fib(n-1) + Fib (n-2);
}
```

```
public static long recFibN(final int n) {
    return (n < 2) ? n : recFibN(n-1) + recFibN (n-2);
}
```

In the first snippet, it is visible the use of the types ulong and uint, that are characteristic of C#. In the second snippet, the use of final indicates this is a Java snippet.

C

CODE MODIFICATION EXAMPLES

In this chapter, the objective is to identify what modification leads to not identifying the language.

C.1 C

```c
int main(int argc, char *argv[])
{
    char t[255]="alphaBETA";
    str_toupper(t);
    printf("uppercase: %s\n",t);
    str_tolower(t);
    printf("lowercase: %s\n",t);
    return 0;
}
```

(a) Original version.

```c
int main(int argc, string argv)
{
    string t="alphaBETA";
    str_toupper(t);
    printf("uppercase: %s\n",t);
    str_tolower(t);
    printf("lowercase: %s\n",t);
    return 0;
}
```

(b) Different basic type (`string`).

```c
int main(int argc, string argv)
{
    string t="alphaBETA";
    str_toupper(t);
    write("uppercase: ",t);
    str_tolower(t);
    write("lowercase: ",t);
    return 0;
}
```

(c) Different standard functions.

```c
main(argc, argv)
{
    t="alphaBETA";
    str_toupper(t);
    printf("uppercase: ",t);
    str_tolower(t);
    printf("lowercase: ",t);
    return 0;
}
```

(d) Different type declaration (dynamic).

Figure 11: Four progressively altered snippets of C code used to establish what is the breaking point for the identification of a programming language.

C.2 C++

```cpp
#include <iostream>
#include <string>

using namespace std;
int main() {
  string foo("_upperCas3Me!!");
  str_toupper(foo);
  cout << foo << endl;
  str_tolower(foo);
  cout << foo << endl;
  return 0;
}
```

(a) Original version.

```
with iostream
with string

with std;
int main() {
  string foo("_upperCas3Me!!");
  str_toupper(foo);
  cout << foo << endl;
  str_tolower(foo);
  cout << foo << endl;
  return 0;
}
```

(b) Different import keyword

```
with iostream
with string

with std;
int main() {
  string foo("_upperCas3Me!!");
  str_toupper(foo);
  write(foo);
  str_tolower(foo);
  write(foo);
  return 0;
}
```

(c) Different standard functions.

```
with iostream
with string

with std;
int main() {
  string foo = "_upperCas3Me!!";
  str_toupper(foo);
  write(foo);
  str_tolower(foo);
  write(foo);
  return 0;
}
```

(d) Different type declaration (dynamic).

```
with iostream
with string

with std;
int main():
  string foo = "_upperCas3Me!!";
  str_toupper(foo);
  write(foo);
  str_tolower(foo);
  write(foo);
  return 0;
```

(e) Different code block delimitation

Figure 12: Five progressively altered snippets of C++ code used to establish what is the breaking point for the identification of a programming language.

C.3 C#

```
static void Main(string[] args)
{
    string x = "foo";
    x += "bar";
    System.Console.WriteLine(x);
}

```

(a) Original version.

```
static void Main(string[] args)
{
    string x = "foo";
    x += "bar";
    write(x);
}
```

(b) Different standard functions.

```
static void Main(string[] args)
    string x = "foo";
    x += "bar";
    write(x);


```

(c) Different code block delimitation

```
function Main(string[] args)
    x = "foo";
    x += "bar";
    write(x);

```

(d) Different type declaration and function keyword.

Figure 13: Four progressively altered snippets of C# code used to establish what is the breaking point for the identification of a programming language.

C.4 JAVA

```java
public static String repeat(String str, int
     times) {
    StringBuilder sb = new StringBuilder(
    str.length() * times);
    for (int i = 0; i < times; i++)
        sb.append(str);
    return sb.toString();
}
```

(a) Original version.

```java
public static String repeat(String str, int
     times)
    StringBuilder sb = new StringBuilder(
    str.length() * times);
    for (int i = 0; i < times; i++)
        sb.append(str);
    return sb.toString();

```

(b) Different code block delimitation

```
function repeat(str, times)
    StringBuilder sb = new StringBuilder(
    str.length() * times);
    for (int i = 0; i < times; i++)
        sb.append(str);
    return sb.toString();

```

(c) Different method declaration

```
function repeat(str, times)
    sb = createString(numberOfChars(str) *
    times);
    for (int i = 0; i < times; i++)
        sb.add(str);
    return sb.stringify();
```

(d) Different type declaration and standard functions

Figure 14: Four progressively altered snippets of Java code used to establish what is the breaking point for the identification of a programming language.

C.5 HASKELL

```haskell
releaseFork :: Int -> Fork -> STM ()
releaseFork i fork = putTMVar fork i

```

(a) Original version.

```haskell
releaseFork :: Int -> Fork -> STM ()
releaseFork i fork = putTMVar(fork(i))
```

(b) Different function composition

```haskell
releaseFork(Int i, Fork fork)
releaseFork i fork = putTMVar(fork(i))
```

(c) Different method declaration

```haskell
releaseFork(Int i, Fork fork)
releaseFork = putTMVar(fork(i))
```

(d) Different function definition

Figure 15: Four progressively altered snippets of HASKELL code used to establish what is the breaking point for the identification of a programming language.

C.6 PYTHON

```
def move2front_encode(strng, symboltable):
    sequence, pad = [], symboltable[::]
    for char in strng:
        indx = pad.index(char)
        sequence.append(indx)
        pad = [pad.pop(indx)] + pad
    return sequence

```

(a) Original version.

```
function move2front_encode(string strng,
    array symboltable):
    array sequence, pad = [], symboltable
    [::]
    for char in strng:
        int indx = pad.index(char)
        sequence.append(indx)
        array pad = [pad.pop(indx)] + pad
    return sequence
```

(b) Different type and function declaration

```
function move2front_encode(string strng,
    array symboltable){
    array sequence, pad = [], symboltable
    [::]
    for char in strng {
        int indx = pad.index(char)
        sequence.append(indx)
        array pad = [pad.pop(indx)] + pad
    }
    return sequence
}
```

(c) Different code block delimitation

```
function move2front_encode(string strng,
    array symboltable){
    array sequence, pad = [], symboltable.
    reverse()
    foreach(char char: strng) {
        int indx = pad.index(char)
        sequence.append(indx)
        array pad = [pad.pop(indx)] + pad
    }
    return sequence
}
```

(d) Different functions

Figure 16: Four progressively altered snippets of PYTHON code used to establish what is the breaking point for the identification of a programming language.

D

MACHINE LEARNING MODEL TESTS

As a way to determine what characteristics influence the recognition of a programming language, and to test the machine learning model, several tests were made.

These tests consisted in, starting with a code snippet written in the original programming language and doing modifications to it. The modifications are made on the code blocks, end of statement, types annotation, function or method signature, function or method call, naming style, standard library, and entry point.

D.1 PYTHON

This is the starting point.

```
class Helloer:
  def say_hi(self, n):
    print(f'Hi #{n}')
    return n

my_name = input()
h = Helloer()
h.say_hi(my_name)
```

Modification on the the symbol delimitation

```
class Helloer {
  def say_hi(self, n) {
    print(f'Hi #{n}')
    return n
  }
}

my_name = input()
h = Helloer()
h.say_hi(my_name)
```

Modification on the suffix keyword

```
class Helloer
  def say_hi(self, n)
    print(f'Hi #{n}')
    return n
  end
end

my_name = input()
h = Helloer()
h.say_hi(my_name)
```

Modification on both the prefix and suffix keyword

```
class Helloer
begin
  def say_hi(self, n)
  begin
    print(f'Hi #{n}')
    return n
  end
end

my_name = input()
h = Helloer()
h.say_hi(my_name)
```

Modification symbol based

```
class Helloer:
  def say_hi(self, n):
    print(f'Hi #{n}');
    return n;

my_name = input();
h = Helloer();
h.say_hi(my_name);
```

Modification type annotation prefix

```
class Helloer:
  String say_hi(Helloer self, String n):
    print(f'Hi {n}')
    return n

String my_name = input()
Helloer h = Helloer()
h.say_hi(my_name)
```

Modification type annotation suffix with separator

```
class Helloer:
  say_hi(self: Helloer, n: String): String
    print(f'Hi {n}')
    return n

my_name: String = input()
h: Helloer = Helloer()
h.say_hi(my_name)
```

Modification type annotation suffix

```
class Helloer:
  say_hi(self Helloer, n String) String:
    print(f'Hi {n}')
    return n

my_name String = input()
h Helloer = Helloer()
h.say_hi(my_name)
```

Modification function keyword

```
class Helloer:
  function say_hi(self, n):
    print(f'Hi {n}')
    return n

my_name = input()
h = Helloer()
h.say_hi(my_name)
```

Modification return type prefix

```
class Helloer:
  String say_hi(self, n):
    print(f'Hi {n}')
    return n

my_name = input()
h = Helloer()
h.say_hi(my_name)
```

Modification return type suffix with separator

```
class Helloer:
  say_hi(self, n): String:
    print(f'Hi {n}')
    return n

my_name = input()
h = Helloer()
h.say_hi(my_name)
```

Modification return type suffix

```
class Helloer:
  say_hi(self, n) String:
    print(f'Hi {n}')
    return n

my_name = input()
h = Helloer()
h.say_hi(my_name)
```

Modification open unnamed parameters

```
class Helloer:
  def say_hi(self, n):
    print f'Hi {n}'
    return n
```

```
my_name = input
h = Helloer
h.say_hi my_name
```

Modification enclosed named parameters

```
class Helloer:
  def say_hi(self, n):
    print(s: f'Hi {n}')
    return n

my_name = input()
h = Helloer()
h.say_hi(n: my_name)
```

Modification enclosed Signature

```
class Helloer:
  def say_hi(self, n):
    (print f'Hi {n}')
    return n

my_name = (input)
h = (Helloer)
(h (say_hi my_name))
```

Modification camel case

```
class Helloer:
  def sayHi(self, n):
    print(f'Hi {n}')
    return n

myName = input()
h = Helloer()
h.sayHi(myName)
```

Modification Pascal Case

```
class Helloer:
  def SayHi(self, N):
    print(f'Hi {N}')
    return N

MyName = input()
H = Helloer()
H.SayHi(MyName)
```

Modification standard library

```
class Helloer:
  def say_hi(self, n):
    write_line(f'Hi {n}')
    return n

my_name = read_line()
h = Helloer()
h.say_hi(my_name)
```

Modification named function or method

```
class Helloer:
  def say_hi(self, n):
    print(f'Hi {n}')
    return n

def main:
  my_name = input()
  h = Helloer()
  h.say_hi(my_name)
```

D.2 Go

Go starting point

```
package main
import ("fmt")

func sayHi(n string) string {
```

```
    fmt.Println("Hi", n)
    return n
}

func main() {
    var myName string
    fmt.Scanln(&myName)
    sayHi(myName)
}
```

Modification on indentation

```
package main
import ("fmt")

func sayHi(n string) string :
    fmt.Println("Hi", n)
    return n

func main() :
    var myName string
    fmt.Scanln(&myName)
    sayHi(myName)
```

Modification suffix keyword

```
package main
import ("fmt")

func sayHi(n string) string {
    fmt.Println("Hi", n)
    return n
end

func main() {
    var myName string
    fmt.Scanln(&myName)
    sayHi(myName)
end
```

Modification prefix and suffix keyword

```
package main
import ("fmt")

func sayHi(n string) string
begin
  fmt.Println("Hi", n)
  return n
end

func main()
begin
  var myName string
  fmt.Scanln(&myName)
  sayHi(myName)
end
```

Modification symbol based

```
package main
import ("fmt")

func sayHi(n string) string {
  fmt.Println("Hi", n);
  return n;
}

func main() {
  var myName string;
  fmt.Scanln(&myName);
  sayHi(myName);
}
```

Modification type annotation implicit

```
package main
import ("fmt")

func sayHi(n) {
  fmt.Println("Hi", n)
  return n
}

func main() {
  myName = ""
```

```
    fmt.Scanln(&myName)
    sayHi(myName)
  }
```

Modification type annotation prefix

```
  package main
  import ("fmt")

  func string sayHi(string n) {
    fmt.Println("Hi", n)
    return n
  }

  func void main() {
    string myName
    fmt.Scanln(&myName)
    sayHi(myName)
  }
```

Modification type annotation suffix with separator

```
  package main
  import ("fmt")

  func sayHi(n: string): string {
    fmt.Println("Hi", n)
    return n
  }

  func main(): void {
    myName: string
    fmt.Scanln(&myName)
    sayHi(myName)
  }
```

Modification definition keyword

```
  package main
  import ("fmt")

  def sayHi(n string) {
```

```
    fmt.Println("Hi", n)
    return n
}

def main() {
    var myName string
    fmt.Scanln(&myName)
    sayHi(myName)
}
```

Modification return type prefix

```
package main
import ("fmt")

string sayHi(n string) {
    fmt.Println("Hi", n)
    return n
}

void main() {
    var myName string
    fmt.Scanln(&myName)
    sayHi(myName)
}
```

Modification return type suffix with separator

```
package main
import ("fmt")

sayHi(n string): string {
    fmt.Println("Hi", n)
    return n
}

main(): void {
    var myName string
    fmt.Scanln(&myName)
    sayHi(myName)
}
```

Modification return type suffix

```
package main
import ("fmt")

sayHi(n string) string {
  fmt.Println("Hi", n)
  return n
}

main() void {
  var myName string
  fmt.Scanln(&myName)
  sayHi(myName)
}
```

Modification open unnamed parameters

```
package main
import "fmt"

func sayHi(n string) string {
  fmt.Println "Hi", n
  return n
}

func main() {
  var myName string
  fmt.Scanln &myName
  sayHi myName
}
```

Modification enclosed named parameters

```
package main
import ("fmt")

func sayHi(n string) string {
  fmt.Println(s:"Hi", p: n)
  return n
}

func main() {
  var myName string
  fmt.Scanln(s: &myName)
```

```
    sayHi(n: myName)
}
```

Modification enclosed signature

```
package main
import ("fmt")

func sayHi(n string) string {
  (fmt (Println "Hi", n))
  return n
}

func main() {
  var myName string
  (fmt (Scanln &myName))
  (sayHi myName)
}
```

Modification snake case

```
package main
import ("fmt")

func say_hi(n string) string {
  fmt.Println("Hi", n)
  return n
}

func main() {
  var my_name string
  fmt.Scanln(&my_name)
  say_hi(my_name)
}
```

Modification Pascal Case

```
package main
import ("fmt")

func SayHi(N string) string {
  fmt.Println("Hi", N)
```

```
    return N
}

func main() {
    var MyName string
    fmt.Scanln(&MyName)
    SayHi(MyName)
}
```

Modification Standard library

```
package main
import ("std")

func sayHi(n string) string {
    std.Writeln("Hi", n)
    return n
}

func main() {
    var myName string
    std.Readln(&myName)
    sayHi(myName)
}
```

Modification Entry Point First Global scoped line

```
package main
import ("fmt")

func sayHi(n string) string {
    fmt.Println("Hi", n)
    return n
}

var myName string
fmt.Scanln(&myName)
sayHi(myName)
```

D.3 JAVA

Java starting point

```
public class Helloer {
  public String sayHi(String n) {
    System.out.println("Hi " + n);
    return n;
  }
}

public class Main {
  public static void main(String[] args) {
    String myName = System.console().nextLine();
    Helloer h = new Helloer();
    h.sayHi(myName);
  }
}
```

Modification on indentation

```
public class Helloer:
  public String sayHi(String n):
    System.out.println("Hi " + n);
    return n;

public class Main:
  public static void main(String[] args):
    String myName = System.console().nextLine();
    Helloer h = new Helloer();
    h.sayHi(myName);
```

Modification suffix keyword

```
public class Helloer
  public String sayHi(String n)
    System.out.println("Hi " + n);
    return n;
  end
end

public class Main
  public static void main(String[] args)
    String myName = System.console().nextLine();
```

```
    Helloer h = new Helloer();
    h.sayHi(myName);
  end
end
```

Modification Prefix and Suffix keyword

```
public class Helloer
begin
  public String sayHi(String n)
  begin
    System.out.println("Hi " + n);
    return n;
  end
end

public class Main
begin
  public static void main(String[] args)
  begin
    String myName = System.console().nextLine();
    Helloer h = new Helloer();
    h.sayHi(myName);
  end
end
```

Modification end of statement line based

```
public class Helloer {
  public String sayHi(String n) {
    System.out.println("Hi " + n)
    return n
  }
}

public class Main {
  public static void main(String[] args) {
    String myName = System.console().nextLine()
    Helloer h = new Helloer()
    h.sayHi(myName)
  }
}
```

Modification type annotation implicit

```
public class Helloer {
  public sayHi(n) {
    System.out.println("Hi " + n);
    return n;
  }
}

public class Main {
  public static main(args) {
    myName = System.console().nextLine();
    h = new Helloer();
    h.sayHi(myName);
  }
}
```

Modification type annotation suffix with separator

```
public class Helloer {
  public sayHi(n: String): String {
    System.out.println("Hi " + n);
    return n;
  }
}

public class Main {
  public static main(args: String[]): void {
    myName: String = System.console().nextLine();
    h: Helloer = new Helloer();
    h.sayHi(myName);
  }
}
```

Modification type annotation suffix

```
public class Helloer {
  public sayHi(n String) String {
    System.out.println("Hi " + n);
    return n;
  }
}

public class Main {
  public static main(args String[]) void {
```

```
    myName String = System.console().nextLine();
    h Helloer = new Helloer();
    h.sayHi(myName);
  }
}
```

Modification definition keyword

```
public class Helloer {
  public def sayHi(String n) {
    System.out.println("Hi " + n);
    return n;
  }
}

public class Main {
  public static def main(String[] args) {
    String myName = System.console().nextLine();
    Helloer h = new Helloer();
    h.sayHi(myName);
  }
}
```

Modification function keyword

```
public class Helloer {
  public function sayHi(String n) {
    System.out.println("Hi " + n);
    return n;
  }
}

public class Main {
  public static function main(String[] args) {
    String myName = System.console().nextLine();
    Helloer h = new Helloer();
    h.sayHi(myName);
  }
}
```

Modification return type suffix with separator

```
public class Helloer {
  public sayHi(String n): String {
    System.out.println("Hi " + n);
    return n;
  }
}

public class Main {
  public static main(String[] args): void {
    String myName = System.console().nextLine();
    Helloer h = new Helloer();
    h.sayHi(myName);
  }
}
```

Modification return type suffix

```
public class Helloer {
  public sayHi(String n) String {
    System.out.println("Hi " + n);
    return n;
  }
}

public class Main {
  public static main(String[] args) void {
    String myName = System.console().nextLine();
    Helloer h = new Helloer();
    h.sayHi(myName);
  }
}
```

Modification open unnamed parameters

```
public class Helloer {
  public String sayHi(String n) {
    System.out.println "Hi " + n;
    return n;
  }
}

public class Main {
  public static void main(String[] args) {
    String myName = System.console.nextLine;
```

```
    Helloer h = new Helloer;
    h.sayHi myName;
  }
}
```

Modification enclosed named parameters

```
public class Helloer {
  public String sayHi(String n) {
    System.out.println(s: "Hi " + n);
    return n;
  }
}

public class Main {
  public static void main(String[] args) {
    String myName = System.console().nextLine();
    Helloer h = new Helloer();
    h.sayHi(n: myName);
  }
}
```

Modification enclosed signature

```
public class Helloer {
  public String sayHi(String n) {
    (System (out (println ("Hi " + n))));
    return n;
  }
}

public class Main {
  public static void main(String[] args) {
    String myName = (System (console nextLine));
    Helloer h = (Helloer new);
    (h (sayHi myName));
  }
}
```

Modification snake case

```
public class helloer {
```

```
  public String say_hi(String n) {
    System.out.println("Hi " + n);
    return n;
  }
}

public class main {
  public static void main(String[] args) {
    String my_name = System.console().nextLine();
    helloer h = new helloer();
    h.say_hi(my_name);
  }
}
```

Modification camel case

```
public class helloer {
  public String sayHi(String n) {
    System.out.println("Hi " + n);
    return n;
  }
}

public class main {
  public static void main(String[] args) {
    String myName = System.console().nextLine();
    helloer h = new helloer();
    h.sayHi(myName);
  }
}
```

Modification pascal case

```
public class Helloer {
  public String SayHi(String N) {
    System.out.println("Hi " + N);
    return N;
  }
}

public class Main {
  public static void main(String[] Args) {
    String MyName = System.console().nextLine();
    Helloer H = new Helloer();
```

```
    H.SayHi(MyName);
  }
}
```

Modification standard library

```
public class Helloer {
  public String sayHi(String n) {
    writeLine("Hi " + n);
    return n;
  }
}

public class Main {
  public static void main(String[] args) {
    String myName = readLine();
    Helloer h = new Helloer();
    h.sayHi(myName);
  }
}
```

Modification first global scoped line

```
public class Helloer {
  public String sayHi(String n) {
    System.out.println("Hi " + n);
    return n;
  }
}

String myName = System.console().nextLine();
Helloer h = new Helloer();
h.sayHi(myName);
```

D.3.1 *New set of tests*

Modifications on access modifier

```
class Helloer {
  String sayHi(String n) {
```

```
    System.out.println("Hi " + n);
    return n;
  }
}

class Main {
  void main(String[] args) {
    String myName = System.console().nextLine();
    Helloer h = new Helloer();
    h.sayHi(myName);
  }
}
```

Modifications on access modifier and classes

```
String sayHi(String n) {
  System.out.println("Hi " + n);
  return n;
}

void main(String[] args) {
  String myName = System.console().nextLine();
  sayHi(myName);
}
```

Modifications on access modifier, classes and functions

```
String sayHi(String n) {
  print("Hi " + n);
  return n;
}

void main(String[] args) {
  String myName = nextLine();
  sayHi(myName);
}
```

Modifications on access modifier, classes, functions and types

```
sayHi(n) {
  print("Hi " + n);
  return n;
```

```
}

main(args) {
  myName = nextLine();
  sayHi(myName);
}
```

Modifications on types and classes

```
public sayHi(n) {
  System.out.println("Hi " + n);
  return n;
}

public static main(args) {
  myName = System.console().nextLine();
  sayHi(myName);
}
```

D.4 JAVASCRIPT

Javascript starting point

```
class Helloer {
  sayHi(n) {
    console.log("Hi " + n);
    return n;
  }
}

let myName = readline();
let h = new Helloer();
h.sayHi(myName);
```

Modification on indentation

```
class Helloer:
  sayHi(n):
    console.log("Hi " + n);
    return n;
```

```
let myName = readline();
let h = new Helloer();
h.sayHi(myName);
```

Modification suffix keyword

```
class Helloer
  sayHi(n)
    console.log("Hi " + n);
    return n;
  end
end

let myName = readline();
let h = new Helloer();
h.sayHi(myName);
```

Modification prefix and suffix keyword

```
class Helloer
begin
  sayHi(n)
  begin
    console.log("Hi " + n);
    return n;
  end
end

let myName = readline();
let h = new Helloer();
h.sayHi(myName);
```

Modification end of statement line based

```
class Helloer {
  sayHi(n) {
    console.log("Hi " + n)
    return n
  }
}
```

```
let myName = readline()
let h = new Helloer()
h.sayHi(myName)
```

Modification type annotation prefix

```
class Helloer {
  string sayHi(string n) {
    console.log("Hi " + n);
    return n;
  }
}

string myName = readline();
Helloer h = new Helloer();
h.sayHi(myName);
```

Modification type annotation suffix with separator

```
class Helloer {
  sayHi(n: string): string {
    console.log("Hi " + n);
    return n;
  }
}

myName: string = readline();
h: Helloer = new Helloer();
h.sayHi(myName);
```

Modification type annotation suffix

```
class Helloer {
  sayHi(n string) string {
    console.log("Hi " + n);
    return n;
  }
}

myName string = readline();
h Helloer = new Helloer();
h.sayHi(myName);
```

Modification definition keyword

```
class Helloer {
  def sayHi(n) {
    console.log("Hi " + n);
    return n;
  }
}

let myName = readline();
let h = new Helloer();
h.sayHi(myName);
```

Modification function keyword

```
class Helloer {
  function sayHi(n) {
    console.log("Hi " + n);
    return n;
  }
}

let myName = readline();
let h = new Helloer();
h.sayHi(myName);
```

Modification return type prefix

```
class Helloer {
  string sayHi(n) {
    console.log("Hi " + n);
    return n;
  }
}

let myName = readline();
let h = new Helloer();
h.sayHi(myName);
```

Modification return type suffix with separator

```
class Helloer {
  sayHi(n): string {
    console.log("Hi " + n);
    return n;
  }
}

let myName = readline();
let h = new Helloer();
h.sayHi(myName);
```

Modification return type suffix

```
class Helloer {
  sayHi(n) string {
    console.log("Hi " + n);
    return n;
  }
}

let myName = readline();
let h = new Helloer();
h.sayHi(myName);
```

Modification open unnamed parameters

```
class Helloer {
  sayHi(n) {
    console.log "Hi " + n;
    return n;
  }
}

let myName = readline;
let h = new Helloer;
h.sayHi myName;
```

Modification enclosed named parameters

```
class Helloer {
  sayHi(n) {
```

```
    console.log(s: "Hi " + n);
    return n;
  }
}

let myName = readline();
let h = new Helloer();
h.sayHi(n: myName);
```

Modification enclosed signature

```
class Helloer {
  sayHi(n) {
    (console (log ("Hi " + n)));
    return n;
  }
}

let myName = (readline);
let h = (Helloer new);
(h (sayHi myName);
```

Modification snake case

```
class helloer {
  say_hi(n) {
    console.log("Hi " + n);
    return n;
  }
}

let my_name = readline();
let h = new helloer();
h.say_hi(my_name);
```

Modification camel case

```
class helloer {
  sayHi(n) {
    console.log("Hi " + n);
    return n;
  }
```

```
}

let myName = readline();
let h = new helloer();
h.sayHi(myName);
```

Modification pascal case

```
class Helloer {
  SayHi(N) {
    console.log("Hi " + N);
    return N;
  }
}

let MyName = readline();
let H = new Helloer();
H.SayHi(MyName);
```

Modification standard library

```
class Helloer {
  sayHi(n) {
    print("Hi " + n);
    return n;
  }
}

let myName = scan();
let h = new Helloer();
h.sayHi(myName);
```

Modification named function or method

```
class Helloer {
  sayHi(n) {
    console.log("Hi " + n);
    return n;
  }
}

function main() {
```

```
  let myName = readline();
  let h = new Helloer();
  h.sayHi(myName);
}
```

D.5 RUBY

Ruby starting point

```
class Helloer
  def say_hi(n)
    puts "Hi #{n}"
    n
  end
end

my_name = gets
h = Helloer.new
h.say_hi(my_name)
```

Modification on symbol delimitation

```
class Helloer {
  def say_hi(n) {
    puts "Hi #{n}"
    n
  }
}

my_name = gets
h = Helloer.new
h.say_hi(my_name)
```

Modification on indentation

```
class Helloer:
  def say_hi(n):
    puts "Hi #{n}"
    n
```

```
my_name = gets
h = Helloer.new
h.say_hi(my_name)
```

Modification Prefix and Suffix keyword

```
class Helloer
  def say_hi(n)
  begin
    puts "Hi #{n}"
    n
  end
end

my_name = gets
h = Helloer.new
h.say_hi(my_name)
```

Modification End of statement symbol based

```
class Helloer
  def say_hi(n)
    puts "Hi #{n}";
    n;
  end
end

my_name = gets;
h = Helloer.new;
h.say_hi(my_name);
```

Modification Type Annotation Prefix

```
class Helloer
  String say_hi(String n)
    puts "Hi #{n}"
    n
  end
end

String my_name = gets
Helloer h = Helloer.new
```

```
h.say_hi(my_name)
```

Modification Type Annotation Suffix with separator

```
class Helloer
  say_hi(n: String): String
    puts "Hi #{n}"
    n
  end
end

my_name: String = gets
h: Helloer = Helloer.new
h.say_hi(my_name)
```

Modification Type Annotation Suffix

```
class Helloer
  say_hi(n String) String
    puts "Hi #{n}"
    n
  end
end

my_name String = gets
h Helloer = Helloer.new
h.say_hi(my_name)
```

Modification Function Keyword

```
class Helloer
  function say_hi(n)
    puts "Hi #{n}"
    n
  end
end

my_name = gets
h = Helloer.new
h.say_hi(my_name)
```

Modification Return Type Prefix

```
class Helloer
  String say_hi(n)
    puts "Hi #{n}"
    n
  end
end

my_name = gets
h = Helloer.new
h.say_hi(my_name)
```

Modification Return Type Suffix with separator

```
class Helloer
  say_hi(n): String
    puts "Hi #{n}"
    n
  end
end

my_name = gets
h = Helloer.new
h.say_hi(my_name)
```

Modification Return Type Suffix

```
class Helloer
  say_hi(n) String
    puts "Hi #{n}"
    n
  end
end

my_name = gets
h = Helloer.new
h.say_hi(my_name)
```

Modification Enclosed unnamed parameters

```
class Helloer
  def say_hi(n)
```

```
    puts("Hi #{n}")
    n
  end
end

my_name = gets()
h = Helloer.new()
h.say_hi(my_name)
```

Modification open unnamed parameters

```
class Helloer
  def say_hi(n)
    puts "Hi #{n}"
    n
  end
end

my_name = gets
h = Helloer.new
h.say_hi my_name
```

Modification enclosed named parameters

```
class Helloer
  def say_hi(n)
    puts(s: "Hi #{n}")
    n
  end
end

my_name = gets()
h = Helloer.new()
h.say_hi(n: my_name)
```

Modification enclosed signature

```
class Helloer
  def say_hi(n)
    (puts "Hi #{n}")
    n
  end
```

```
end

my_name = (gets)
h = (Helloer new)
(h (say_hi my_name))
```

Modification camel case

```
class Helloer
  def sayHi(n)
    puts "Hi #{n}"
    n
  end
end

myName = gets
h = Helloer.new
h.sayHi(myName)
```

Modification pascal case

```
class Helloer
  def SayHi(n)
    puts "Hi #{n}"
    n
  end
end

MyName = gets
H = Helloer.new
H.SayHi(MyName)
```

Modification standard library

```
class Helloer
  def say_hi(n)
    write_line "Hi #{n}"
    n
  end
end

my_name = read_line
```

```
h = Helloer.new
h.say_hi(my_name)
```

Modification named function or method

```
class Helloer
  def say_hi(n)
    puts "Hi #{n}"
    n
  end
end

def main()
  my_name = gets
  h = Helloer.new
  h.say_hi(my_name)
end
```

D.6 PHP

PHP starting point

```
class Helloer {
  function sayHi($n) {
    echo "Hi $n";
    return n;
  }
}

$my_name = fgets(STDIN);
$h = new Helloer();
$h->sayHi(my_name);
```

Modification on indentation

```
class Helloer:
  function sayHi($n):
    echo "Hi $n";
    return n;
```

```
$my_name = fgets(STDIN);
$h = new Helloer();
$h->sayHi(my_name);
```

Modification on suffix keyword

```
class Helloer
  function sayHi($n)
    echo "Hi $n";
    return n;
  end
end

$my_name = fgets(STDIN);
$h = new Helloer();
$h->sayHi(my_name);
```

Modification on prefix and suffix keyword

```
class Helloer
begin
  function sayHi($n)
  begin
    echo "Hi $n";
    return n;
  end
end

$my_name = fgets(STDIN);
$h = new Helloer();
$h->sayHi(my_name);
```

Modification on end of statement line based

```
class Helloer {
  function sayHi($n) {
    echo "Hi $n"
    return n
  }
}

$my_name = fgets(STDIN)
```

```
$h = new Helloer()
$h->sayHi(my_name)
```

Modification on type annotation prefix

```
class Helloer {
  string sayHi(string $n) {
    echo "Hi $n";
    return n;
  }
}

string $my_name = fgets(STDIN);
Helloer $h = new Helloer();
$h->sayHi(my_name);
```

Modification on type annotation suffix with separator

```
class Helloer {
  sayHi($n: string): string {
    echo "Hi $n";
    return n;
  }
}

$my_name: string = fgets(STDIN);
$h: Helloer = new Helloer();
$h->sayHi(my_name);
```

Modification on type annotation suffix

```
class Helloer {
  sayHi($n string) string {
    echo "Hi $n";
    return n;
  }
}

$my_name string = fgets(STDIN);
$h Helloer = new Helloer();
$h->sayHi(my_name);
```

Modification definition keyword

```
class Helloer {
  def sayHi($n) {
    echo "Hi $n";
    return n;
  }
}

$my_name = fgets(STDIN);
$h = new Helloer();
$h->sayHi(my_name);
```

Modification return type prefix

```
class Helloer {
  string sayHi($n) {
    echo "Hi $n";
    return n;
  }
}

$my_name = fgets(STDIN);
$h = new Helloer();
$h->sayHi(my_name);
```

Modification return type suffix with separator

```
class Helloer {
  sayHi($n): string {
    echo "Hi $n";
    return n;
  }
}

$my_name = fgets(STDIN);
$h = new Helloer();
$h->sayHi(my_name);
```

Modification return type suffix

```
class Helloer {
  sayHi($n) string {
```

```
    echo "Hi $n";
    return n;
  }
}

$my_name = fgets(STDIN);
$h = new Helloer();
$h->sayHi(my_name);
```

Modification open unnamed parameters

```
class Helloer {
  function sayHi($n) {
    echo "Hi $n";
    return n;
  }
}

$my_name = fgets STDIN;
$h = new Helloer;
$h->sayHi my_name;
```

Modification enclosed named parameters

```
class Helloer {
  function sayHi($n) {
    echo(s: "Hi $n");
    return n;
  }
}

$my_name = fgets(s: STDIN);
$h = new Helloer();
$h->sayHi(n: my_name);
```

Modification enclosed signature

```
class Helloer {
  function sayHi($n) {
    (echo "Hi $n");
    return n;
  }
```

```
}

$my_name = (fgets STDIN);
$h = (Helloer new);
($h (sayHi my_name));
```

Modification snake case

```
class helloer {
  function say_hi($n) {
    echo "Hi $n";
    return n;
  }
}

$my_name = fgets(STDIN);
$h = new helloer();
$h->say_hi(my_name);
```

Modification camel case

```
class helloer {
  function sayHi($n) {
    echo "Hi $n";
    return n;
  }
}

$myName = fgets(STDIN);
$h = new helloer();
$h->sayHi(myName);
```

Modification pascal case

```
class Helloer {
  function SayHi($N) {
    echo "Hi $N";
    return N;
  }
}

$MyName = fgets(STDIN);
```

```
$H = new Helloer();
$H->SayHi(MyName);
```

Modification standard library

```
class Helloer {
  function sayHi($n) {
    write_line "Hi $n";
    return n;
  }
}

$my_name = read_line(STDIN);
$h = new Helloer();
$h->sayHi(my_name);
```

Modification named function or method

```
class Helloer {
  function sayHi($n) {
    echo "Hi $n";
    return n;
  }
}

function main() {
  $my_name = fgets(STDIN);
  $h = new Helloer();
  $h->sayHi(my_name);
}
```