SEVEN
publicações acadêmicas

Natural language, computational thinking, and cognitive development

Crossref doi
https://doi.org/10.56238/sevened2023.006-061

Marcelo Magalhães Foohs
Doctor in Informatics in Education. Associate Professor,
Department of Specialized Studies
Federal University of Rio Grande do Sul
ORCID: https://orcid.org/0000-0002-4735-0732
E-mail: 00145282@ufrgs.br

ABSTRACT
In this article, we explore the programming language as a manifestation of computational thinking, from the translation of abstractions into codes, so that computers can understand them. Some key concepts, such as syntax, abstraction, and problem-solving, are addressed, highlighting their relationship to higher psychological functions. It analyzes the synergistic interaction between computational thinking and natural language, examining how this symbiosis can influence textual cohesion and coherence. A didactic sequence that makes use of this synergy is presented, with summative evaluation criteria, in which remediation is the key piece for the transposition of linguistic narratives to the digital world, employing the Scratch platform as a catalyzing tool. This aims not only to enrich students' creative expression, but also to effectively integrate natural language, computational thinking, and educational technology.

Keywords: Computational thinking, Natural language, Higher psychological functions, Programming, Educational technology.

1 INTRODUCTION

In the Brazilian educational landscape, official tests have highlighted worrying gaps in the development of crucial skills for students. They are not mere statistics, but reveal profound deficiencies in textual production, in the interpretation of information and in the resolution of complex problems. These gaps reflect a larger challenge: the lack of an integrated approach that unites the powers of language, the logic of computational thinking, and higher cognitive functions.

This chapter sets out to explore the multifaceted intersection between programming language, natural language, and higher cognitive functions, aiming not only to identify these gaps, but to offer visionary solutions to fill them. Based on the theoretical foundations of renowned scholars such as Ingedore Koch (2020), with his in-depth analysis of textual linguistics, and on Vygotsky's (2001) ideas about higher cognitive functions, this work seeks to establish an intimate connection between apparently distinct but intrinsically linked concepts.

Programming language, far from being just a tool for communication between humans and computers, is the means by which we translate abstractions, logic, and algorithms into a form that computers not only understand, but execute. It is the code of the computational mind, allowing programmers to express their logical vision and devise solutions to complex problems.

In this journey of understanding, key aspects will be explored thoroughly: syntax, abstraction, and problem-solving. Its intrinsic relationship with higher psychological functions will be highlighted, analyzing how the synergistic interaction between computational thinking and natural language can influence textual cohesion and coherence.

In addition, an educational approach is proposed that integrates these elements effectively, using a didactic sequence that is based on this symbiosis. Remediation plays a key role in transposing linguistic narratives to the digital world, including summative assessment criteria and employing the Scratch platform as a catalytic tool. The goal is not only to enrich students' creative expression, but to effectively integrate natural language, computational thinking, and educational technology for students' cognitive development.

This chapter is not only a theoretical exposition, but an invitation to reflect on how the synergy between these domains can shape a promising educational future for Brazil. We are faced with a unique opportunity: the opportunity to redefine the educational paradigm by empowering our students to innovate, create and solve complex problems effectively for the challenges of the 21st century.

2 METHODOLOGICAL PROCEDURES

The methodological approach adopted in this chapter follows a structured process, aiming to provide a comprehensive and consistent understanding of the topics presented. We have divided the structure of the chapter into distinct sections, each focusing on specific and interconnected elements of computational thinking and its relationship to natural language.

The chapter follows a logical organization, beginning with a comprehensive introduction to computational thinking and its relationship to programming language and natural language. From there, each subsection was structured to explore a key concept of computational thinking, relating it directly to practical examples in Python language. The methodology adopted involved a sequence of demonstrations, using the programming language as a vehicle to illustrate and connect the concepts addressed.

The choice of examples was based on their ability to elucidate theoretical concepts in a clear and accessible way. Each example was explained in detail, emphasizing the relationships between computational thinking and natural language elements, in order to make the content more tangible and applicable.

The progressive construction of the examples, from subsection to subsection, allows a gradual and in-depth exploration of the themes, promoting the progressive understanding of the content and highlighting the interactions between the concepts presented.

Throughout the development, we constantly seek to contextualize the examples in the educational scenario, demonstrating how these concepts can be applied in the learning process, aiming to strengthen cognitive skills and the integral development of students.

This methodology was adopted with the purpose of offering a didactic structure, facilitating the understanding and applicability of the concepts presented, in addition to promoting an integrated view between computational thinking, natural language and cognitive development.

3 PROGRAMMING LANGUAGE: COMPUTER MIND CODE

Programming language is the means by which humans translate their abstractions, logic, and algorithms in such a way that computers can understand and execute them. Essentially, it is the code of the computational mind, allowing programmers to express their logical vision and create solutions to complex problems through computational thinking strategies.

The definition of computational thinking, developed jointly by the *International Society for Technology in Education* (ISTE) and the [1]Computer Science Teachers Association (CSTA) ([2]OPERATIONAL..., 2011), offers a comprehensive and precise description of the topic, which includes the following characteristics: 1. Problem formulation; 2. Logical organization and data analysis; 3. Data representation through abstractions; 4. Automation of solutions through algorithmic thinking; 5. Identification, analysis and implementation of efficient solutions; 6. Generalization and transference to a variety of problems.

Next, we will explore how the programming language becomes a tool for the manifestation of computational thinking.

3.1 SYNTAX AND LOGICAL STRUCTURING

Just like natural language, where grammatical and semantic rules structure our communications, the programming language also operates with a specific syntax. This syntax is the structural foundation that allows programmers to express algorithms, control structures, and programming logic in a way that is organized and understandable to computers. Let's explore how this manifests itself in Python, a language known for its clarity and expressiveness.

The syntax in Python is notable for its simplicity and readability. It uses an approach that values the clarity of the code, allowing you to express complex operations in a straightforward way. In the following, we will present examples that demonstrate how logical structuring and specific syntax in Python facilitate the expression of algorithms and mathematical operations in a concise manner.

[1] International Society for Technology in Education (ISTE).
[2] Computer Science Teachers Association (CSTA).

In the first example, Python syntax, such as the use of the "=" assignment operator, the "+" addition operator, and the "print" function for display, allows you to clearly and directly express a simple mathematical sum operation.

```
# Sum of two numbers
number1 = 10
number2 = 5
soma = number1 + number2
print("The sum is:", sum)
```

In the second example, we use Python syntax to calculate the average of three numbers. Logical structuring and specific syntax make it possible to perform this arithmetic operation in a concise and understandable manner.

```
# Calculation of the average of three numbers
number1 = 15
number2 = 20
number3 = 10
Average = (number1 + number2 + number3) / 3
print("The average is:", average)
```

Finally, we demonstrate the use of conditional structure in Python to check whether a number is odd or even. The specific syntax for decision structures allows you to control the flow of the program according to logical conditions.

```
# Odd or even number check
number = 7
if number % 2 == 0:
print("The number is even.")
else:
print("The number is odd.")
```

These examples illustrate how syntax in Python enables the structured organization of algorithms and programming logic, allowing you to express operations in an efficient and readable manner.

3.2 ABSTRACTION AND MODULARIZATION

One of the fundamental facets of computational thinking lies in the ability to abstract complex concepts into simpler, reusable components. The programming language offers tools for this abstraction, allowing the creation of functions and objects that represent abstract ideas or operations. Let's look at some examples to understand this concept more deeply.

Abstraction, in this context, allows you to encapsulate specific logics or operations in reusable structures, such as functions or objects, promoting a more simplified and abstract understanding of the problem. In a first example, we have a simple function in Python that calculates the square of a number. The "calcular_quadrado" function abstracts the mathematical operation required to find the square of a number, making it a modular and reusable unit.

```
# Function for calculating the square of a number
def calcular_quadrado(number):
return number ** 2
# Calling the function and printing the result
number = 8
print("The square is:", calcular_quadrado(number))
```

In the following second example, we demonstrate the creation of a Python class, representing a product. This class abstracts a product's common properties and behaviors, such as name and price, making it easier to create specific instances and reuse code.

```
# Creating a Python class to represent a product
Product class:
def __init__(self, nome, preco):
self.nome = nome
self.preco = preco
def exibir_produto(self):
print(f"Product: {self.name}, Price: ${self.price}")
# Creating an instance of the Product class and viewing its details
product1 = Product("Pen", 2.50)
produto1.exibir_produto()
```

Additionally, we present a function that checks whether a number is positive, negative, or zero. This function encapsulates the verification logic and returns an abstract result based on the given number.

```
# Function to check if a number is positive, negative, or zero
def verificar_numero(number):
if number > 0:
return "Positive"
ELIF number < 0:
return "Negative"
else:
return "Zero"
```

```
# Calling the function and printing the result
num = -7
print("The number is:", verificar_numero(num))
```

These examples highlight how abstraction and modularization in Python allow you to encapsulate logic and operations in reusable units, promoting code reuse and making it easier to understand complex concepts in simpler, more abstract terms.

3.3 PROBLEM-SOLVING AND LOGIC

The programming language's ability to express problem-solving through sequences of logical instructions is critical. By creating algorithms and control structures, such as loops and conditionals, programmers translate their logical reasoning ability into computational actions. Let's explore some examples that highlight this relationship between problem solving and logic in Python.

Logic in programming allows the creation of sequences of commands that guide the behavior of the program according to specific conditions, reflecting human logical reasoning in computational actions. In a first example, we demonstrate the use of a conditional structure to identify the largest number between two values. The logic of the "if-else" structure allows for the comparison of variables and decision-making based on logical conditions.

```
# Identification of the largest number between two values
number1 = 15
number2 = 20
If number1 > number2:
print("The largest number is:", number1)
else:
print("The largest number is:", number2)
```

In the second example below, we use a "for" loop to print the even numbers from 1 to 10. Using the repeating structure allows you to perform a repetitive action based on logical conditions.

```
# Loop for to print the even numbers from 1 to 10
for i in range(1, 11):
if i % 2 == 0:
print(i, end=" ")
```

Finally, we present a conditional framework that verifies the age for access to restricted content. This condition-based verification demonstrates the rationale behind conditional decisions in programs.

```
# Age verification for access to restricted content
age = 17
if age >= 18:
```

```
print("Access granted to restricted content.")
else:
print("You are not old enough to access this content.")
```

These examples aim to illustrate how the programming language allows you to express problem-solving logic through structured algorithms, using conditional and repetition structures to make decisions and perform actions in an automated and logical manner.

4 COMPUTATIONAL THINKING, HIGHER PSYCHOLOGICAL FUNCTIONS, AND EDUCATION

In addition to serving as a code for the computational mind, the programming language, structured by computational thinking strategies, is intrinsically linked to higher psychological functions, and can promote their development and improvement. Higher psychological functions, as conceived by Vygotsky (2001) and Vygotsky, Luria and Leontiev (2010), which include conceptual thinking, voluntary memory, attention, problem solving, planning and self-regulation, imagination and creativity, are important for the cognitive development of the student and for coping with complex challenges in the digital age.

That said, some examples in Python will be briefly presented below, which exemplify the relationship between computational thinking strategies and higher psychological functions. In addition, it is important to point out that Vygotsky (2001, p. 65), in his book *Thought and Language*, draws attention to the fundamental role of education in the relationship between learning and cognitive development:

> Our second series of investigations focused on the temporal relationships between teaching processes and the development of the psychological functions that correspond to them. We have found that teaching often precedes development. The child acquires certain habits and qualifications in a given domain before he learns to apply them consciously and deliberately. There is never a complete parallel between the course of teaching and the development of the corresponding functions.

It is interesting to note how this quote from Vygotsky (2001) reinforces the importance of education in this process, highlighting the relationship between learning and cognitive development. The following Python language examples can serve as a bridge between the theoretical concepts of higher psychological functions and their practical application in the educational context. Each example presented offers a tangible demonstration of how computational thinking strategies intertwine with fundamental cognitive skills such as problem-solving, planning, and self-regulation, promoting not only theoretical understanding but also the concrete application of these concepts in the educational setting.

4.1 CONCEPTUAL THINKING

Conceptual thinking involves the ability to understand and manipulate abstract concepts. In programming, students apply computational thinking strategies when dealing with concepts such as variables, functions, and control structures. Let's explore some examples that demonstrate the practical application of this thinking in the Python language.

Conceptual thinking in programming refers to the ability to work with abstract concepts in order to solve problems and create solutions using logical structures and specific operations. In the following first example, we present a function that calculates the volume of a cube based on the size of the given side. Here, the manipulation of the geometric concept of volume by means of the function abstracts the mathematical formula, evidencing conceptual thinking in the direct application of mathematical formulas.

```
# Function to calculate the volume of a cube
def calcular_volume_cubo(side):
return lado ** 3
# Calling the function and printing the result
lado_cubo = 5
print("The volume of the cube is:", calcular_volume_cubo(lado_cubo))
```

In the second example, we have a function that checks whether a number is positive or not. The manipulation of the concept of positivity by means of a function abstracts logic from verification, showing conceptual thinking in the application of conditional logic.

```
# Creating a function to check if a number is positive
def verificar_positivo(number):
if number > 0:
return True
else:
return False
# Checking if a number is positive and printing the result
num = -7
if verificar_positivo(num):
print("The number is positive.")
else:
print("The number is not positive.")
```

Finally, we introduce the use of lists in Python to store elements and access them through indexes. This manipulation of the concept of lists exemplifies how students can work with abstract data structures to manipulate information in an organized manner.

```
# Use of lists to store elements and manipulate data
lista_frutas = ['Apple', 'Banana', 'Orange', 'Strawberry']
print("The third fruit on the list is:", lista_frutas[2])
```

These examples aim to illustrate how conceptual thinking manifests itself in programming, allowing students to manipulate and apply abstract concepts to solve problems and create solutions in Python.

4.2 VOLUNTARY MEMORY

Voluntary memory refers to the ability to store and access information consciously and intentionally. In programming, the organization and structuring of information are fundamental for the construction of efficient algorithms. Let's explore below examples that illustrate how programming encourages the use of voluntary memory, enabling learners to store and access information consciously.

Voluntary memory in programming is related to the ability to store and retrieve information in an intentional way, using specific data structures and concepts. In this first example, we demonstrate the use of a Python dictionary to store contacts and their associated emails. This structure allows conscious access to information through keys, exemplifying voluntary memory in the organization and direct access to data.

```
# Storing contacts in a dictionary
agenda_contatos = {
'John': 'joao@email.com',
'Maria': 'maria@email.com',
'Carlos': 'carlos@email.com'
}
# Accessing a contact's email
print("John's email:", agenda_contatos['John'])
```

In the second example, we have variables that store data about the state of a game, such as score, lives, and current level. This intentional manipulation of variables to update and display information evidences voluntary memory in programming.

```
# Using variables to store data from a game
Score = 1500
lives = 3
nivel_atual = 5
# Updating and displaying the gamerscore's score
Score += 500
print("New Score:", Score)
```

Finally, we present a function that calculates the total monthly expenses based on a list of values. The intentional use of the function to store calculations exemplifies how voluntary memory is applied in the reuse of solutions to solve similar problems.

```
# Using a function to store monthly expense calculations
def calcular_despesas_mensais(expenses):
total_despesas = sum(expenses)
return total_despesas
# Calling the function and displaying the total monthly expenses
despesas_janeiro = [1000, 1500, 800, 2000]
total = calcular_despesas_mensais(despesas_janeiro)
print("Total expenses in January:", total)
```

These examples demonstrate how voluntary memory manifests itself in programming, allowing learners to intentionally store and access information to solve problems and organize data in Python. In addition, it is important to point out that, just like other higher psychological functions, voluntary memory can be developed with regular practice and exercise.

The practice of consciously manipulating information in programming not only strengthens technical skills but also has a significant impact on learners' cognitive development. By practicing data organization and problem-solving through programming, students not only improve their critical thinking skills but also strengthen volunteer memory, a crucial skill in many aspects of life.

Specifically in the educational context, this practice of developing voluntary memory through programming can be highly beneficial. It is not limited to improving students' technical skills, but also contributes to improving their ability to learn and solve problems in a variety of disciplines.

By integrating volunteer memory development into programming as part of the educational curriculum, we are not only empowering students to deal with future technological challenges, but also strengthening their overall cognitive skills. This type of educational approach not only prepares students for the digital age but also equips them with essential cognitive tools to tackle varied challenges throughout their academic and professional lives.

4.3 ATTENTION AND FOCUS ON PROGRAMMING

Attention is the ability to focus on a specific task while avoiding distractions. In programming, this aspect is crucial, requiring sustained attention to understand and solve complex problems. Let's explore below examples that illustrate how programming strengthens the ability to sustain attention and focus.

The ability to maintain attention is essential in solving programming problems. Learners need to focus on logic, break down challenges into manageable chunks, and develop strategies to find

solutions. In this first example, we have a program that checks if a number is prime. Attention is required to understand the logic behind the algorithm, following each step of the iteration to check if the number is divisible by some number other than 1 and itself.

```
# Creating a program that checks if a number is prime
def verifica_primo(number):
IF number > 1:
for i in range(2, numero):
if (number % i) == 0:
return False
return True
return False
# Checking if a number is prime
num = int(input("Enter a number to check if it's prime: "))
if verifica_primo(num):
print(num, "is a prime number.")
else:
print(num, "is not a prime number.")
```

In the second example, we have a binary search algorithm on an ordered list. Attention is crucial to understanding the logic behind binary search, following each iteration of the loop, and understanding how the algorithm efficiently finds the desired item in the list.

```
# Crafting a binary search program on an ordered list
def busca_binaria(list, item):
low = 0
high = len(list) - 1
while baixo <= alto:
middle = (low + high) // 2
Kick = list[middle]
if chute == item:
return half
if chute > item:
High = Medium - 1
else:
low = middle + 1
return None
# Using binary search
```

```
minha_lista = [1, 3, 5, 7, 9]
print("Number 5:", busca_binaria(minha_lista, 5))
```

These examples show how attention is key in programming. Developing this skill not only helps students solve complex problems in programming, but also strengthens their ability to focus and concentrate, which are essential skills for the educational process. Programming not only challenges them cognitively but also prepares them to maintain attention on challenging tasks, contributing positively to their educational development.

4.4 TROUBLESHOOTING

Problem-solving is an essential skill for finding effective solutions to complex challenges. In programming, this skill is constantly exercised, driving strategic thinking and creativity in the search for innovative solutions. Let's explore some examples that illustrate this ability to solve problems effectively in Python.

Programming provides an environment conducive to the development of problem-solving. By tackling algorithmic challenges, learners not only implement solutions but also enhance their ability to think creatively and efficiently to solve problems. In the first example presented below, we have a sequential search algorithm on a list. Problem-solving is applied by creating an efficient algorithm to find a specific item in the list, iterating through each element until it is found.

```
# Example 1: Implementing a Sequential Search Algorithm on a List
def busca_sequencial(list, item):
for i in range(len(lista)):
if lista[i] == item:
return i
return None
# Using sequential search
minha_lista = [6, 2, 8, 5, 3, 9]
print("Number 5:", busca_sequencial(minha_lista, 5))
```

In the second example, we have a program that calculates the factorial of a number. Here, problem solving is reflected in the creation of an efficient algorithm to calculate the factorial, applying the mathematical logic necessary for this task.

```
# Developing a program that calculates the factorial of a number
def calcular_fatorial(number):
if numero == 0 or numero == 1:
return 1
else:
```

```
factorial = 1
for i in range(2, numero + 1):
fatorial *= and
return fatorial
# Calculating the Factorial of a Number
n = 5
print("The factorial of", n, "is", calcular_fatorial(n))
```

These examples highlight how problem-solving in programming goes beyond simply implementing code. The development of this skill provides students with the ability to face challenges in a strategic manner, a valuable skill for the educational process. By integrating problem-solving through programming into the educational curriculum, we are not only teaching the programming language but also strengthening students' ability to approach complex problems creatively and effectively. This skill goes beyond the technical domain, being a valuable tool for cognitive development and the ability to find innovative solutions in various areas.

4.5 PLANNING AND SELF-REGULATION

Planning and self-regulation are fundamental skills for programmers, as they enable the development of efficient plans and the ability to adjust the behavior of the code according to those plans. Next, we'll explore examples that illustrate how these concepts apply in programming using the Python language.

In programming, planning means not only creating code that performs a specific task, but also anticipating possible scenarios and errors that may arise during execution. Self-regulation refers to the ability to make adjustments, corrections, and improvements to the code, ensuring predictable and efficient behavior. In the first example presented, we created a simulated login system where you can verify that the username and password match the registration. The application of planning occurs by considering the possibility of error when inserting a non-existent user, and it is essential to predict and treat this situation.

```
# Example 1: Implementing an Error-Handling Login System
usuarios_registrados = { user1': 'senha1', 'user2': 'senha2'}
def fazer_login(user, password):
try:
if usuarios_registrados[user] == Senha:
return True
else:
return False
```

```
except KeyError:
print("User not found!")
return False
# Trying to log in
login = fazer_login (user1', 'senha1')
print("Login:", login)
```

In the second example, we illustrate a program that validates user input to ensure that it is a number between 1 and 100. Here, self-regulation occurs by incorporating mechanisms to handle invalid inputs, ensuring that the program functions properly even in the face of potential errors.

```
# Developing a program to validate data entry
def validar_entrada(number):
try:
if int(numero) > 0 and int(numero) <= 100:
return True
else:
return False
except ValueError:
print("Please enter a valid number.")
return False
# Validating User Input
entrada_valida = validar_entrada('50')
print("Valid ticket:", entrada_valida)
```

These examples show how planning and self-regulation in programming go beyond simple coding. By integrating these skills into the educational context, in addition to learning to write code, students also develop the ability to anticipate scenarios, plan solutions, and correct mistakes, fundamental skills for solving complex problems in various areas of life and learning.

4.6 IMAGINATION AND CREATIVITY

Imagination and creativity are fundamental in the world of programming, providing the ability to conceive innovative ideas and explore various possibilities. In the practice of programming, students have the opportunity to exercise their imagination, creating programs and projects that go beyond simple logical functioning. Let's explore below some examples to illustrate how programming can encourage imagination and creativity using the Turtle library in Python.

In programming, imagination translates into the ability to visualize solutions or concepts, while creativity manifests itself in the way these ideas are applied in a unique and innovative way. In this

first example below, we use the Turtle library to draw a colorful kaleidoscopic pattern. The goal is to highlight how students can express their imagination visually, creating something aesthetically appealing based on programming logic.

```
# Creating a Drawing with the Turtle Library
from turtle import *
speed(10)
bgcolor("black")
colors = ["red", "orange", "yellow", "green", "blue", "purple"]
for x in range(360):
pencolor(colors[x % 6])
width(x / 100 + 1)
forward(x)
left(59)
done()
```

In the second example, we've created a simple game of clicking on the screen. Here, students' creativity comes into play as they develop the logic behind an interactive game, exploring not only visual drawing ability, but also the user's interaction with the program.

```
# Creating a simple game with the Turtle library
from turtle import *
speed(0)
bgcolor("black")
color("white")
hideturtle()
def desenhar_borda():
penup()
goto(-140, 140)
pendown()
for side in range(4):
forward(280)
right(90)
left(90)
penup()
goto(0, 0)
pendown()
setheading(0)
```

```
desenhar_borda()
write("Click inside the area to start the game.", align="center", font=("Arial", 16, "normal"))
def click(x, y):
goto(x, y)
dot(20)
onclick
done()
```

These examples illustrate how coding can be a fertile field for students' imagination and creativity. By introducing this approach into education, we are not only developing technical skills, but also fostering the ability to solve problems, innovate, and express ideas. These skills are essential for forming responsible and engaged citizens in today's society, enabling them to find creative solutions to complex challenges and to contribute meaningfully to the world we live in.

5 COMPUTATIONAL THINKING AND NATURAL LANGUAGE DEVELOPMENT

The interplay between computational thinking and natural language enhancement represents a deep and enriching symbiosis. Not only does it strengthen language comprehension, but it also directly influences the way we structure, interpret, and utilize language in our daily communication. By applying the principles of computational thinking, we immerse ourselves in a world of logic, abstraction, and problem-solving that has a significant impact on improving human language.

The contribution of computational thinking to natural language is not just limited to textual expression. It extends to how we organize our thoughts, structure our ideas, and communicate information in a cohesive and coherent way. Textual cohesion and coherence, identified by Beaugrande and Dressler (1981) as central elements in natural language, is a key point to be explored. By unraveling how computational thinking intertwines with these elements, we delve not only into the analysis of linguistic structures, but also into understanding how computational logic can enhance the clarity, effectiveness, and efficiency of written and spoken communication.

In the course of this section, we will take a closer look at how the principles of computational thinking intertwine with natural language, looking at how their applications can enrich understanding and communication in the linguistic context.

5.1 COHESION: THE LOGICAL INTERACTION OF TEXTUAL ELEMENTS

Textual cohesion refers to the way elements of a text logically connect together. In the world of programming, computational thinking is a powerful catalyst for cohesion, requiring programmers to organize their code in a logical and structured way. This practice is not only applicable to

programming, but is also valuable in natural language, as the ability to structure ideas in a logical and sequential manner is strengthened through computational thinking, resulting in greater textual coherence and clarity.

Exploring this relationship further, let's go into some examples that illustrate the interconnection between computational thinking and natural language, focusing on the cohesion of textual elements. Cohesion plays a crucial role in both programming and written communication, ensuring clarity and understanding of what is expressed. In the first example below, the "calcular_media" function is created to calculate the average of a list of numbers. Logic is essential: you add up the elements of the list, calculate the number of elements and, finally, calculate the average.

```
# Example of a function to calculate the average of a list of numbers
def calcular_media(list):
soma = sum(list)
quantity = len(list)
Average = Sum / Quantity
return media
# List of numbers
numbers = [10, 15, 12, 18, 20]
# Calling the function to calculate the average
media_calculada = calcular_media(numbers)
# Displaying the average
print("The average of the numbers is:", media_calculada)
```

In the second example presented below, a sentence is manipulated using Python. The sentence is broken down into words, and then the order of those words is reversed, demonstrating a logical textual manipulation.

```
# Example of text manipulation in Python
sentence = "Computational thinking strengthens textual cohesion."
words = sentence.split() # Dividing the sentence into words
reverse = ' '.join(words[::-1]) # Reversing word order
# Displaying the sentence reversed
print("Inverted sentence:", reverse)
```

In the third example presented below, a dictionary is used to organize information about a student, showing how logical structure is applied in organizing data.

```
# Example of organizing information in a Python dictionary
student = {
"name": "Mary",
```

```
"Age": 22,
"course": "Computer Science",
"Notes": [8.5, 9.0, 7.8, 9.5]
}
# Displaying student information in an organized manner
print(f"{student['name']}, {student['age']} years, from {student['course']}, grades: {student['grades']}")
```

These examples suggest that computational thinking, by requiring a logical and organized structure, strengthens the understanding of the use of cohesion in both programming and natural language. Cohesion is a vital skill not only for code, but also for textual clarity and comprehension in our everyday communication, playing a crucial role in developing individuals who are able to contribute effectively to society.

5.2 COHERENCE: SEMANTIC HARMONY IN THE TEXT

Textual coherence involves the construction of an overall and logical meaning in a text. Computational thinking, by stimulating the organization and logical connection of concepts, can assist in the creation of coherent texts in natural language. Programmers, when developing algorithms and logical structures, develop the ability to maintain semantic coherence in their codes. This ability is transferable to natural language, resulting in texts in which ideas connect in logical and meaningful ways.

Let's look at some examples of this interconnection between computational thinking and natural language, with a focus on the coherence of textual elements. In the first example, the 'e_palindromo' function checks whether a word is a palindrome, i.e. whether it is read the same way backwards. Logical coherence lies in the comparison between the original word and its inverted version, reflecting the need for a coherent structure to establish meaning in natural language.

```
# Palindrome Function Example
def e_palindromo(word):
return word == word[::-1]
# Checking if a word is a palindrome
result = e_palindromo("recognize")
If result:
print("It's a palindrome.")
else:
print("It's not a palindrome.")
```

In the second example, the 'ordenar_numeros' function arranges a list of numbers in ascending order. Logical coherence lies in using the ordering method, ensuring that the numbers are in a logical sequence. This reflects the need for cohesive and coherent organization to convey ideas logically in natural language.

```
# Example function sort numbers
def ordenar_numeros(list):
return sorted(list)
# Sorting a list of numbers
numbers = [8, 3, 5, 1, 9, 4]
lista_ordenada = ordenar_numeros(numbers)
print("Sorted list:", lista_ordenada)
```

In the third example, the 'fibonacci' function generates the Fibonacci sequence based on the number of terms specified. Logical coherence lies in the mathematical recursion used to generate the sequence, ensuring that each number is the coherent result of the sum of the previous two. This reflects the coherent structure required to convey complex information in natural language.

```
# Fibonacci Function Example
def fibonacci(n):
if n <= 1:
return n
else:
return fibonacci(n - 1) + fibonacci(n - 2)
# Generating the Fibonacci sequence
terms = 10
sequencia = [fibonacci(i) for i in range(termos)]
print("Fibonacci sequence:", sequence)
```

By highlighting the logical coherence present in Python code examples, it reinforces the relevance of computational thinking in the development of coherence in both programming and natural language. This connection illustrates how the practice of computational thinking can strengthen the understanding and application of the elements of cohesion and coherence in human communication.

6 PROPOSAL OF A DIDACTIC SEQUENCE WITH THE USE OF SCRATCH

The synergy between natural language and computational thinking, as discussed earlier, shows great promise in the development of higher psychological functions. Natural language, used for human expression and communication, and computational thinking, with its abilities of decomposition, abstraction, pattern recognition, and algorithms, have the potential to reinforce each other. This, in

turn, amplifies creativity, problem-solving, and understanding of complex concepts. In this context, a didactic sequence centered on remediation is proposed, using it as an axis to transform narratives constructed in natural language into interactive digital narratives, employing computational thinking strategies and the Scratch platform.

Remediation, as proposed by Bolter and Grusin (2000), refers to the process by which new media incorporate and reconfigure the characteristics of previous media. It is a concept that underlines the continuous interaction between different forms of communication and their mutual influences, resulting in new forms of expression. From an educational perspective, remediation can be seen as an opportunity to integrate natural language and computational thinking, expanding pedagogical possibilities.

6.1 VISUAL PROGRAMMING LANGUAGE SCRATCH

Scratch is a visual programming platform developed by MIT (Massachusetts Institute of Technology) to teach programming concepts in an accessible and fun way. It is designed to be used by children, young people, and programming beginners, offering an intuitive interface based on blocks of code, which makes it more attractive compared to traditional programming languages.

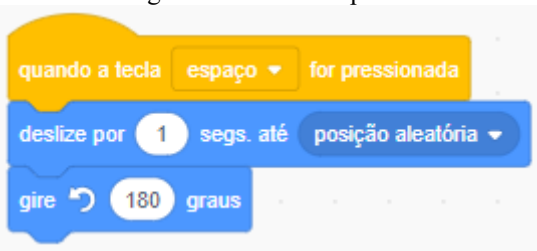
Scratch allows users to create interactive projects, such as animations, games, stories, and more, without the need to write complex code. Its graphical interface utilizes colorful command blocks that fit together like puzzle pieces, making it easy to create sequences of actions. Command blocks represent different operations such as movement, appearance, sound, controls, and variables. For example, there are blocks for moving characters, playing sounds, changing colors, and controlling the flow of the program.

Figure 1 – Blocks to move and change color.

Source - Own

In addition, Scratch allows you to create actors and backgrounds to compose scenes. Actors can be customized with their own images, sounds, and animations. The Scratch canvas is divided into two main areas: the programming area, where blocks are dragged to create scripts, and the stage, where projects are executed. Projects in Scratch can respond to events such as clicks and keystrokes. This allows programs to react to the user's actions.

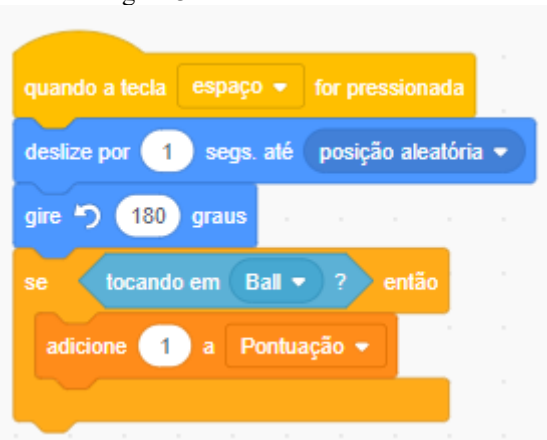
Figure 2 – Event Response



Source – Own

Scratch also allows the use of variables to store information and control the behavior of programs. In the example below, Score is the variable that is modified when the character touches the ball.

Figure 3 – Variable Modification



Source – Own

Finally, Scratch offers a series of visual and interactive resources that help users understand the principles of programming in a practical and engaging way, promoting creativity and logical thinking.

6.2 PROPOSAL FOR DIDACTIC SEQUENCE

This didactic sequence proposal aims to amplify the benefits of the synergy between computational thinking, natural language and higher psychological functions. Considering the characteristics of Scratch, it is natural to anticipate that, during the remediation process, the language, initially written in accordance with standard norms, will undergo substantial transformations. This will

result in the adoption of a linguistic variant more appropriate to the new digital environment of narrative, thus promoting the concept of pedagogy of linguistic variation, as highlighted by Faraco (2015). This, in turn, supports the importance of a pedagogical approach to the Portuguese language that is fully aware of the sociolinguistic communicative context. From this perspective, the following didactic sequence is proposed:

1. Introduction to the concept of remediation and computational thinking: Introduce students to the concept of remediation, highlighting its relationship with the evolution of media and the interaction between natural language and computational thinking. Introduce the fundamental principles of computational thinking, such as: decomposition, abstraction, pattern recognition, and algorithms.
2. Natural Language Narrative Analysis: Encourage students to explore traditional and/or authorial narratives, identifying narrative elements, characters, plot, and key points. Analyze the structures and techniques of storytelling, taking into account the psycholinguistic and emotional aspects of the characters. Introduce the scripting process as an integral part of transposing natural language text into an interactive digital narrative, using Scratch.
3. Scripting and transposition to interactive digital narratives: Explain the importance of scripting in the transformation of natural language narratives to interactive digital narratives in the Scratch environment. Present the Scratch software as a tool for building interactive digital narratives. Assist students in translating the elements of linguistic narratives into computational language, adapting and recreating the story interactively.
4. Development of interactive narratives: Guide students in the creation of their own projects in Scratch, integrating elements of the adapted linguistic narrative. Actively stimulate the application of computational thinking, to solve challenges and create interactivity in the narrative, taking into account the previously developed script.

Considering that the proposed didactic sequence integrates three essential components: natural language text, remediation process and interactive digital narrative, and that its main objective is the cognitive development of students, some criteria for the final evaluation will be presented below:

1. Fidelity to the original narrative: Evaluate the fidelity in transposing the fundamental elements of the natural language narrative to the digital format, ensuring the cohesion of the story during the adaptation.
2. Creativity in adaptation and innovation: Evaluate creativity in reinterpreting and adapting narrative into an interactive form, taking into account the innovations and resources used to make digital storytelling unique and engaging.

3. Efficiency in the use of Scratch software: Evaluate the ability of students to efficiently use Scratch features to create interactivity and captivate the target audience, ensuring a fluid and attractive experience.
4. Proficient application of computational thinking: Evaluate the sound application of computational thinking principles, such as: decomposition, abstraction, pattern recognition, and algorithms, during the development of interactive digital storytelling, demonstrating understanding and skill in applying these principles.

The integration of remediation with the proposed pedagogical approach offers a valuable opportunity for the enrichment of the educational experience, uniting human expression through natural language with the creative power of computational thinking. This didactic sequence provides a solid framework for the exploration of these concepts, encouraging the creation of interactive digital narratives that not only broaden the understanding and application of computational thinking, but also enrich students' creative expression. This process, therefore, promotes the cognitive development of students in a comprehensive way.

That said, it is essential to highlight how this didactic sequence proposal, with the use of Scratch, goes beyond the integration between natural language and computational thinking. It becomes a key pillar in strengthening students' capacities to understand and apply essential skills for problem-solving and creative expression. By entering the world of Scratch, students are not only learning how to use a tool; are immersed in a visual programming language that allows the construction of interactive narratives. This hands-on approach provides them with a significant opportunity to:

1. Think algorithmically: The block structure in Scratch teaches students how to think in logical sequences of actions. By organizing commands in a logical way to create desired interactions, they develop the ability to break down problems into smaller, solvable steps.
2. Abstract and recognize patterns: While creating projects in Scratch, students identify visual and functional patterns in the programming blocks. This helps in understanding concepts such as repetition, conditionals, and loops, empowering them to abstract these patterns for application in different contexts.
3. Solve problems creatively: The Scratch platform fosters creativity by allowing students to express their ideas interactively. They are challenged to find original solutions to implement functionalities, promoting creativity in problem solving.
4. Enhance communication and expression skills: By transforming narratives into interactive projects, students are encouraged to express their ideas clearly and cohesively, applying natural language concepts in the construction of digital stories.

In this way, this approach is not limited to learning programming. It provides a unique opportunity for students to develop foundational skills for computational thinking while promoting

creative expression and problem-solving in innovative and meaningful ways. This integration of remediation, natural language, and computational thinking not only enriches the educational experience but also prepares students to meet the challenges and explore the opportunities of the modern world.

7 CONCLUSION: A SYNERGY AND A SYMPHONY

In our immersion into the depths of the synergy between natural language and computational thinking in this chapter, we explore the intricate pathways that intertwine the expressive richness of human communication with the structured logic of computational thinking. The proposal of a didactic sequence involving the remediation and exploration of Scratch represents more than a simple dive into visual programming; It is a profound journey into educational potential that expands the conventional horizons of the classroom.

We begin our journey with an invitation to a linguistic metamorphosis. Entering the universe of computational thinking, we present the fundamental pillars: decomposition, abstraction, pattern recognition, and algorithms. These principles are not just tools for solving logical challenges; They are instruments that enhance the capacity for analysis, creation and innovation. And it is at the intersection of these principles with natural language that the foundations of our educational proposal lie.

Our starting point for the proposed didactic sequence was the thorough analysis of natural language narratives, inviting students to explore the complexities of human expression. This dive allows you to identify narrative elements, understand emotional and psycholinguistic structures, being the first chords of this educational symphony. This thorough analysis lays the groundwork for the next step: the journey of screenwriting, intertwined with the potential of Scratch. Here, students not only translate, but reinvent narratives, adapting them to the emerging digital landscape, thus nurturing the richness of linguistic variation suggested by Faraco (2015). This natural transition allows the universe of natural language to merge harmoniously with the innovative and creative possibilities provided by the digital environment. Scratch, in this panorama, outlines a path where creativity and computational thinking converge harmoniously. Through its colorful blocks, students transform ideas into visual interactions, applying the principles of computational thinking in a practical and engaging way. Every line of code in Scratch is a verse, and every project is a unique work, a digital expression of remediated narratives.

By exploring the development of interactive storytelling, we not only encourage the application of computational thinking, but also the flourishing of creativity. Students become protagonists, creators of their own digital stories, where each challenge overcome is a harmonious note in the melody of learning.

The final evaluation is not a judgment, but an invitation to reflection. Fidelity to the original narrative, creativity in adaptation, efficiency in the use of Scratch and proficient application of computational thinking are criteria that seek not only to measure, but to illuminate progress, recognizing not only the result, but the path taken by the learners.

In this world where the interaction between natural language and computational thinking is essential, our pedagogical proposal proves to be a unique educational soundtrack. It invites educators to become conductors, students to become composers, transforming the classroom into a stage of expression, learning, and discovery.

This chapter is not only a point of arrival, but a starting point for an educational approach that transcends paradigms. The synergy between natural language, computational thinking and education is the symphony that resonates in building citizens of the future, creative, innovative and endowed with the unique ability to translate thoughts into digital language.

Ultimately, this chapter presents itself not only as a contribution to research, but as a practical and valuable tool for educators seeking to enrich their pedagogical practices. The proposal is not just theory; it is an invitation to transformation, to the creation of educational experiences that echo beyond the confines of classrooms, shaping minds for the challenges and opportunities of the 21st century. May this educational symphony inspire, delight and guide the steps of all those involved in the art of educating.

REFERENCES

BEAUGRANDE, Robert-Alain de; DRESSLER, Wolfgang Ulrich W. Introduction to text linguistics. Tübingen, Germany: Max Niemeyer, 1981.

BOLTER, Jay David; GRUSIN, Richard. Remediation: Understanding new media. Cambridge, Massachusetts: MIT Press, 2000.

FARACO, Carlos Alberto. Norma culta brasileira: construção e ensino. *In*: ZILLES, Ana Maria Stahl; FARACO, Carlos Alberto (org.). Pedagogia da variação linguística: língua diversidade e ensino. São Paulo: Parábola Editorial, 2015. p. 19-30.

KOCH, Ingedore. Introdução à linguística textual. São Paulo: Contexto, 2020.

OPERATIONAL Definition of Computational Thinking – for K-12 Education. [*S. l.*]: NSF; ISTE; CSTA, 2011. Disponível em: https://cdn.iste.org/www-root/Computational_Thinking_Operational_Definition_ISTE.pdf. Acesso em: 28 set. 2023.

VIGOTSKY, Lev Semenovich. Pensamento e Linguagem. São Paulo, SP: Ícone, 2001. *E-Book*. Edição eletrônica: Ed. Ridendo Castigat Mores. Disponível em: www.jahr.org. Acesso em: 26 out. 2023.

VIGOTSKY, Lev Semenovich; LURIA, Alexander Romanovich; LEONTIEV, Alex N. Linguagem, desenvolvimento e aprendizagem. Tradução: Maria da Pena Villalobos. 11. ed. São Paulo, SP: Ícone, 2010. (Coleção Educação Crítica).