Universidad Distrital Francisco José de Caldas

Design and Implementation of a Simple Compiler in Python for Generating Personalized Avatars

Juan David Guevara García - 20202020001
judguevarag@udistrital.edu.co
Javier Alejandro Álvarez Marín - 20202020028
jaalvarem@udistrital.edu.co

Bogotá, Colombia
July 8, 2025

July 10, 2025

*Abstract This paper presents a practical exercise in computer science aimed at developing a conceptual understanding of compiler functionality without delving into low-level implementation details. To achieve this, we designed and implemented a Python-based command interpreter capable of rendering digital avatars. The primary objective is to facilitate the internalization of core compiler principles through hands-on experience. Furthermore, this technology could have potential for broader applications, such as enhancing user interaction in e-commerce platforms, or serving as an educational tool to support the teaching of programming and compiler concepts.

Keywords: Compiler Design, Command Interpreter, Python

GitHub Repository: `https://github.com/Jaalx/finalCienciasTres`

Contents

Introduction

1 Background

Compilers are essential tools in computer science that translate high-level programming languages into machine-executable code. Despite their ubiquity, their internal operation often remains abstract to students and novice programmers[1, 2]. Traditional compiler courses focus heavily on theory and formal grammar, which can make the learning curve steep and less engaging[3].

To address this, our project proposes a practical and visual approach to understanding the compilation process. Inspired by formal language theory—specifically context-free grammars—we designed an educational compiler implemented in Python [4]. Instead of generating machine code, the compiler interprets a simple command language and renders personalized digital avatars as output, making compiler stages visually traceable and conceptually tangible [5].

2 Problem Statement

While many avatar generation tools and compiler teaching aids exist, they rarely intersect. Most avatar libraries require direct programmatic control and do not support custom scripting languages. On the other hand, compiler education platforms often lack engaging or visual components, relying purely on textual outputs.

The core problem we address is the lack of a user-friendly and visual tool that integrates compiler principles into a tangible and motivating output—such as real-time avatar generation—using simplified syntax and grammar.

3 Aims and Objectives

Aim: To develop a simple compiler in Python that interprets high-level, avatar-specific commands and renders digital avatars in response.

Objectives:

- Define a domain-specific language (DSL) for avatar customization.
- Implement lexical, syntactic, and semantic analyzers to validate input instructions.
- Integrate a graphical user interface (GUI) for user interaction.
- Use the `py_avataaars` library to render avatars based on validated instructions.
- Provide meaningful error messages for incorrect syntax or invalid attribute combinations.
- Demonstrate the pedagogical value of the tool for teaching compiler concepts.

4 Solution Approach

Our solution involves creating a simplified compiler architecture comprising three key stages:

- **Lexical Analysis:** Identifies tokens from user input using regular expressions.
- **Syntactic Analysis:** Ensures the structure of the input matches a predefined context-free grammar.
- **Semantic Analysis:** Validates the logical compatibility of verbs and attributes (e.g., "teñir cabello" is valid, but "teñir ropa" is not).

Once the input passes all validation stages, it is passed to the rendering module[6], which uses `py_avataaars` to generate a customized SVG avatar. The user interface, built with Tkinter, facilitates command input and provides real-time visual feedback[7].

5 Summary of Contributions and Achievements

This project makes the following contributions:

- A domain-specific mini-language for avatar customization.
- A complete Python-based compiler pipeline (lexical, syntactic, and semantic).
- A GUI tool that transforms user instructions into personalized avatars.
- A reusable educational resource for teaching compilation concepts.

Additionally, we successfully implemented error handling that distinguishes between syntax and semantic issues, reinforcing learning through feedback.

Methodology

6 Lexical Analyzer Implementation

The lexical analyzer is responsible for scanning the input commands and breaking them down into individual tokens. It uses regular expressions to identify keywords, attributes, colors, styles, and other language components.

The implementation in Python is shown below:

```python
import re

class Token:
    def __init__(self, type_, value):
        self.type_ = type_
        self.value = value

    def __repr__(self):
        return f"Token({self.type_}, {self.value})"

class LexicalAnalyzer:
    @staticmethod
    def lex(code: str):
        tokens = []

        token_specification = [
            ("KEYWORD", r'\b(inicio|final)\b'),
            ("VERBO", r'\b(te ir|ajustar|a adir|expresar)\b'),
            ("ATRIBUTO", r'\b(cabello|ropa|boca|ojos|cejas|accesorio)\b'),
            ("COLOR_PIEL", r'\b(negra|bronceada|amarilla|p lida|clara|trigue a|oscura)\b'),
            ("COLOR_CABELLO", r'\b(negro|cobrizo|rubio|rubio_dorado|casta o|casta o_oscuro|rosado_pastel|platinado|rojo|gris)\b'),
            ("COLOR_ROPA", r'\b(negro|azul|azul2|gris|gris_claro|blanco|rojo|rosado|pastel_azul|pastel_verde|pastel_naranja|pastel_rojo|pastel_amarillo|heather)\b'),
            ("COLOR_SOMBRERO", r'\b(negro|azul|azul2|gris|gris_claro|blanco|rojo|rosado|pastel_azul|pastel_verde|pastel_naranja|pastel_rojo|pastel_amarillo|heather)\b'),
            ("ESTILO_CABELLO", r'\b(calvo|parche|sombrero|hiyab|turbante|gorro1|gorro2|gorro3|gorro4|voluminoso|bob|mo o|largo_rizado|largo_ondulado|dreadlocks_largos|frida|afro|afro_cinta|medio_largo|mia|rapado_lados|liso|liso2|mech n|dreadlocks1|dreadlocks2|esponjado|mullet|corto_rizado|corto_plano|corto_redondo|corto_ondulado|corte_lados|cesar|cesar_lado)\b'),
            ("TIPO_ROPA", r'\b(blazer_camisa|blazer_su ter|cuello_su ter|camiseta_gr fica|hoodie|overol|camiseta_cuello_redondo|cuello_scoop|cuello_v)\b'),
            ("EXPRESION", r'\b(neutral|preocupado|incr dulo|comiendo|frunci ndo|triste|gritando|serio|sonriente|lengua|brillo|v mito|feliz)\b'),
            ("ACCESORIO", r'\b(gafas_redondas|gafas_sol|ninguno)\b'),
            ("SEPARADOR", r';'),
            ("SKIP", r'[ \t\n]+'),
            ("MISSMATCH", r'.'),
        ]

        tok_regex = '|'.join(f'(?P<{name}>{pattern})' for name, pattern in token_specification)

        for mo in re.finditer(tok_regex, code):
            kind = mo.lastgroup
            value = mo.group()
            if kind == "SKIP":
                continue
            elif kind == "MISSMATCH":
                raise RuntimeError(f"Caracter inesperado: {value}")
            else:
                tokens.append(Token(kind, value))
        return tokens
```

Listing 1: Lexical Analyzer Code

7 Syntactic Analyzer Implementation

The syntactic analyzer validates the structure of the parsed tokens based on a predefined context-free grammar. It uses a top-down parsing strategy and a recursive descent approach.

The following simplified grammar was used to define valid input:

```
<PROGRAMA>          -> "inicio" <INSTRUCCIONES> "final"
<INSTRUCCIONES>     -> <INSTRUCCION> ";" <INSTRUCCIONES> | <INSTRUCCION> ";"
<INSTRUCCION>       -> <VERBO> <ATRIBUTO> <VALOR>
<VERBO>             -> "teñir" | "ajustar" | "añadir" | "expresar"
<ATRIBUTO>          -> "cabello" | "ropa" | "boca" | "ojos" | "cejas" | "accesorio"
<VALOR>             -> <COLOR_CABELLO> | <COLOR_ROPA> | <ESTILO_CABELLO> | <TIPO_ROPA> | <EX
```

Here is the Python implementation:

```python
class ParserError(Exception):
    pass

class SyntacticAnalyzer:
    def __init__(self, tokens):
        self.tokens = tokens
        self.pos = -1
        self.current_token = None
        self.advance()

    def advance(self):
        self.pos += 1
        if self.pos < len(self.tokens):
            self.current_token = self.tokens[self.pos]
        else:
            self.current_token = None

    def parse(self):
        self.program()

    def match(self, expected_type):
        if self.current_token and self.current_token.type_ == expected_type:
            self.advance()
        else:
            raise ParserError(f"Se esperaba {expected_type}, pero se encontr
    {self.current_token}")

    def program(self):
        self.match("KEYWORD")  # 'inicio'
        self.instructions()
        self.match("KEYWORD")  # 'final'
        if self.current_token is not None:
            raise ParserError(f"Tokens inesperados despu s de 'final': {self.
    current_token}")

    def instructions(self):
        self.instruction()
        while self.current_token and self.current_token.type_ == "SEPARADOR":
            self.match("SEPARADOR")
            if self.current_token and self.current_token.type_ == "VERBO":
                self.instruction()

    def instruction(self):
        self.match("VERBO")
        self.match("ATRIBUTO")
        self.value()

    def value(self):
        valid_types = {
            "COLOR_PIEL", "COLOR_CABELLO", "COLOR_ROPA", "COLOR_SOMBRERO",
            "ESTILO_CABELLO", "TIPO_ROPA", "EXPRESION", "ACCESORIO"
        }
        if self.current_token and self.current_token.type_ in valid_types:
            self.advance()
        else:
            raise ParserError(f"Valor inesperado: {self.current_token}")
```

Listing 2: Syntactic Analyzer Code

This module ensures that each command follows the expected grammar and structure. If invalid patterns are detected, the parser raises detailed errors pointing to the position and type of mistake.

8 Semantic Analyzer Implementation

The semantic analyzer ensures logical consistency between the verbs and attributes used in the instructions. It performs a validation to confirm whether each verb is applied to an allowed attribute and builds a configuration dictionary used to render the avatar.

If semantic inconsistencies are detected (e.g., using `teñir` with `ropa`), the analyzer raises an appropriate error.

```python
class SemanticAnalyzer:
    def __init__(self, tokens):
        self.tokens = tokens
        self.avatar_config = {
            "cabello": None,
            "ropa": None,
            "boca": None,
            "ojos": None,
            "cejas": None,
            "accesorio": None
        }

        self.verbo_atributos_permitidos = {
            "te ir": {"cabello"},
            "ajustar": {"ropa"},
            "a adir": {"accesorio"},
            "expresar": {"boca", "ojos", "cejas"}
        }

    def analyze(self):
        i = 0
        while i < len(self.tokens):
            token = self.tokens[i]

            if token.type_ == "SEPARADOR":
                i += 1
                continue

            if token.type_ == "KEYWORD":
                i += 1
                continue

            if token.type_ == "VERBO":
                if i + 2 >= len(self.tokens):
                    raise ValueError("Instrucci n incompleta cerca de: " + str(token))

                verbo = self.tokens[i].value
                atributo = self.tokens[i + 1].value
                valor = self.tokens[i + 2].value

                if atributo not in self.avatar_config:
                    raise ValueError(f"Atributo desconocido: {atributo}")

                if atributo not in self.verbo_atributos_permitidos.get(verbo, set()):
                    raise ValueError(f"El verbo ’{verbo}’ no puede usarse con el atributo ’{atributo}’")

                self.avatar_config[atributo] = valor
                i += 3
                continue

            raise ValueError(f"Valor inesperado: {token}")

        return self.avatar_config
```

Listing 3: Semantic Analyzer Code

An example of successful semantic analysis:

```
inicio
teñir cabello castaño_oscuro;
ajustar ropa hoodie;
expresar boca sonriente;
final
```

And an invalid instruction:

```
inicio
teñir ropa azul;
final
```

This will trigger an error because the verb `teñir` cannot be used with the attribute `ropa`.

Results

This chapter presents the results obtained from testing the compiler modules and their integration with the avatar rendering system. We evaluate the correctness and behavior of each component by applying valid and invalid input sequences.

9 Validation of Compiler Modules

To verify the functionality of the system, we conducted unit tests on the lexical, syntactic, and semantic analyzers.

9.1 Case 1: Valid Instructions

The following input was used:

```
inicio
teñir cabello castaño_oscuro;
ajustar ropa hoodie;
añadir accesorio gafas_redondas;
expresar boca sonriente;
expresar ojos feliz;
final
```

All analyzers successfully processed the instructions. The semantic analyzer generated the following avatar configuration[3]:

- cabello: castaño_oscuro
- ropa: hoodie
- accesorio: gafas_redondas
- boca: sonriente
- ojos: feliz

The avatar was correctly rendered using the `py_avataaars` library.

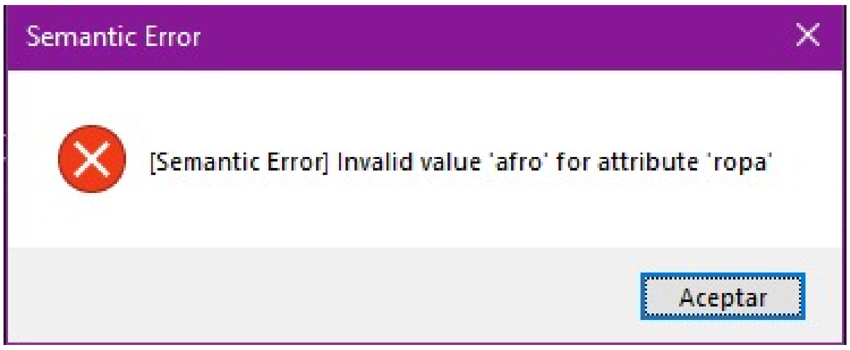
9.2 Case 2: Invalid Semantic Instruction

Invalid input:

```
inicio
teñir ropa afro;
final
```

The lexical and syntactic analyzers accepted this input. However, the semantic analyzer returned the following error:

[Semantic Error] Invalid value 'afro' for attribute 'ropa'



```
[Semantic Error] Invalid value 'afro' for attribute 'ropa'
```

Figure 1: Generated avatar from Example 1 input.

This shows the effectiveness of semantic validation.

10 Example of Avatar Generation

Below is a successful example demonstrating the integration of the full pipeline, from user command input to avatar rendering.

10.1 Avatar Example 1

Input:

```
inicio
ajustar piel trigue a;
te ir cabello rojo;
ajustar cabello afro;
te ir ropa pastel_azul;
ajustar ropa hoodie;
a adir accesorio gafas_sol;
expresar cejas fruncidas;
expresar ojos sorprendidos;
expresar boca gritando;
final
```

Listing 4: Avatar Compilation Input Example

Description:
This input generates an avatar with:

- Afro hairstyle with red color
- Trigueña skin tone
- Hoodie clothing in pastel blue

- Sunglasses
- Furrowed eyebrows, surprised eyes, and shouting mouth

Output:

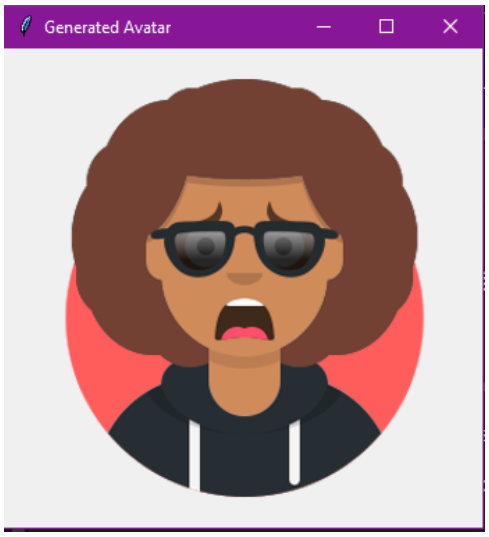


Figure 2: Generated avatar from Example 1 input.

11 Summary

All modules—lexical, syntactic, semantic, and graphical—performed as expected. The system correctly parsed valid inputs and rejected incorrect ones with appropriate error messages. The avatar rendering confirmed that the pipeline from command input to visualization is fully functional and user-friendly.

Discussion and Analysis

This project demonstrates how compiler theory can be applied in a creative and interactive context to enhance educational outcomes. By abstracting away low-level implementation details, students can focus on understanding core compilation phases—lexical, syntactic, and semantic—through visual feedback.

The integration of a graphical interface and avatar rendering introduces an engaging way to explore abstract concepts like grammar rules, token sequences, and attribute validation. The tests showed that the system handles valid and invalid inputs appropriately, confirming the effectiveness of the analyzers.

One observed limitation is the lack of real-time feedback on individual phases during command typing. Future improvements could include interactive error highlighting and suggestions.

Overall, the approach offers both didactic value and creative engagement, promoting deeper learning of compiler structures in a visual, beginner-friendly format.

Conclusions

This project successfully demonstrated the design and implementation of a simplified compiler system in Python focused on the generation of personalized avatars[4, 7, 6]. Through the use of lexical, syntactic, and semantic analysis, user-defined commands could be validated and translated into visual representations using the `py_avataaars` library.

The modular design allowed for clear separation of concerns across the analysis stages, and the system effectively identified structural and logical errors in the input. The GUI component contributed to accessibility and user experience, allowing immediate feedback and visualization.

The results from testing showed that the system performs consistently with both correct and incorrect inputs, confirming the robustness of the implemented analyzers. Overall, this project provided a functional and educational tool that reinforces key concepts of compiler theory through interactive and visual methods.

References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Addison-Wesley, 2006. [Online]. Available: https://www.pearson.com/en-us/subject-catalog/p/compilers-principles-techniques-and-tools/P200000003444/9780136067054

[2] D. Grune, H. E. Bal, C. J. Jacobs, and K. G. Langendoen, *Modern Compiler Design*, 2nd ed. Springer, 2012. [Online]. Available: https://link.springer.com/book/10.1007/978-1-4471-2300-0

[3] M. González and J. Salas, “Design and implementation of an educational compiler for learning grammatical structures,” *Revista de Ingeniería y Tecnología*, vol. 16, no. 2, pp. 45–56, 2018. [Online]. Available: https://example-journal.org/article/educational-compiler

[4] M. Lutz, *Learning Python*, 5th ed. O’Reilly Media, 2013. [Online]. Available: https://learning.oreilly.com/library/view/learning-python-5th/9781449355722/

[5] G. van Rossum and F. L. D. Jr., “The python language reference manual,” https://docs.python.org/3/reference/index.html, accessed: 2025-07-08.

[6] P. S. Foundation, “Tkinter - python interface to tcl/tk,” https://docs.python.org/3/library/tkinter.html, accessed: 2025-07-08.

[7] PyPI, “py_avataaars: A python library for generating avatars,” https://pypi.org/project/py-avataaars/, accessed: 2025-07-08.