Construção De Um Interpretador Em Python Para A Linguagem SCL Baseada na IEC 61131-3

1st Eliézer Carretin
Aluno da instituição
Fundação Hermínio Ometto, Uniararas
Araras, Brasil
eliezercarretin@alunos.fho.edu.br

2nd Iago Bernardino de Almeida
Aluno da instituição
Fundação Hermínio Ometto, Uniararas
Araras, Brasil
iagoalmeida@alunos.fho.edu.br

3rd Renato Luciano Cagnin
Docente da instituição
Fundação Hermínio Ometto, Uniararas
Araras, Brasil
renatocagnin@fho.edu.br

***Abstract*—The IEC 61131-3 standard defines a set of programming languages for Programmable Logic Controllers (PLCs), aiming to promote portability, interoperability, and standardization across industrial platforms. One of its textual languages, Structured Control Language (SCL), is particularly well-suited for expressing structured logic and mathematical operations. However, industrial environments such as Siemens TIA Portal often require full project compilation for testing even minimal code changes, which slows down debugging and iteration.**

This paper presents the development of a Domain-Specific Language (DSL) inspired by SCL, implemented in Python. The project includes the construction of a custom lexical analyzer, parser, semantic evaluator, and execution engine capable of interpreting and simulating control logic outside of proprietary industrial software. The language supports primitive types (`INT`, `REAL`, `BOOL`), variable declarations, arithmetic and logical expressions, conditional statements (`IF/ELSE`), loops (`WHILE`, `FOR`), and output via `PRINT`. The lexical rules are manually defined, and the parser is hand-written following a recursive descent strategy.

This solution enables fast validation and feedback of logical control blocks, making it highly effective for educational purposes, code prototyping, and early-stage development in industrial automation.

***Index Terms*—IEC 61131-3, SCL, DSL, Python Interpreter, Recursive Descent Parser, PLC Programming, Structured Text.**

I. Introdução

A **IEC 61131-3: Programmable Controllers – Part 3: Programming Languages** é a norma internacional que estabelece um conjunto padronizado de linguagens voltadas à programação de **Controladores Lógicos Programáveis (CLPs)**. Seu objetivo é reduzir a fragmentação dos ambientes de programação industrial, promovendo **portabilidade**, **interoperabilidade** e **padronização** entre diferentes fabricantes. A norma contempla linguagens gráficas e textuais, sendo a **Structured Text (ST)** — comercialmente conhecida como **Structured Control Language (SCL)** — uma das mais adequadas para aplicações com lógica estruturada e operações matemáticas complexas.

O artigo-base utilizado neste trabalho é de autoria de Wirth (1996) e trata da necessidade de padronização formal nas definições sintáticas de linguagens, utilizando a notação **EBNF (Extended Backus-Naur Form)** para representar gramáticas de forma clara e precisa [3]. Embora o artigo não trate especificamente de SCL ou CLPs, seus fundamentos gramaticais são amplamente aplicáveis à descrição da linguagem utilizada na norma IEC 61131-3. Este trabalho se apropria dessa base teórica para construir uma linguagem específica de domínio (DSL) inspirada na SCL.

O problema central abordado neste estudo é a dificuldade de validação isolada de blocos de código em ambientes industriais como o **TIA Portal**, que exige a compilação completa do projeto, mesmo para testes simples. Essa limitação dificulta prototipagens rápidas, depuração e ensino da lógica de controle.

Diante disso, propõe-se o desenvolvimento de um **interpretador leve e independente**, construído inteiramente em **Python**, que permita a análise, avaliação e execução simulada de trechos de código em SCL de forma modular. A solução inclui a implementação de um **analisador léxico artesanal**, que reconhece tokens a partir de expressões regulares, e de um **parser descendente recursivo**, responsável por validar a sintaxe conforme uma gramática definida manualmente com base na EBNF.

Ao contrário de outras abordagens que utilizam ferramentas automáticas como `PLY` ou `Lark`, este projeto adota uma estrutura manual e educacional. Os componentes principais do interpretador foram organizados nos arquivos `token_definitions.py` (definição de tokens), `scl_parser.py` (parser e execução), `fho_parser_base.py` (estrutura base abstrata) e `main.py` (execução do interpretador).

A linguagem resultante oferece suporte a: (i) **declaração e atribuição de variáveis** com tipos primitivos (`INT`, `REAL`, `BOOL`), (ii) **operações aritméticas e lógicas**, (iii) **estruturas de controle** como `IF`, `WHILE`, `FOR`, e (iv) comando de saída com `PRINT`. O interpretador simula a execução das instruções por meio de uma **tabela de símbolos** que armazena valores e tipos.

A proposta reforça o uso da IEC 61131-3 como base técnica, mas adapta sua aplicação a um contexto mais didático, leve e interpretável, com foco em **ensino, prototipagem** e **testes rápidos de lógica de controle**.

II. Contextualização e Objetivos da Aplicação

O desenvolvimento de aplicações industriais com Controladores Lógicos Programáveis (CLPs) apresenta desafios

constantes no que diz respeito à agilidade no ciclo de desenvolvimento, especialmente nas fases de prototipação, testes e validação de lógica de controle. Em plataformas como o *TIA Portal*, da Siemens, qualquer modificação em trechos de código escritos em Structured Control Language (SCL) exige a compilação completa do projeto, mesmo quando a intenção é validar apenas uma lógica pontual. Essa limitação compromete a eficiência no processo de depuração, revisão de código e aprendizado prático da linguagem.

Neste cenário, este trabalho propõe a criação de uma ferramenta que permita a análise e execução isolada de trechos de código inspirados na linguagem SCL, conforme definida na norma IEC 61131-3. A proposta consiste na construção de uma linguagem específica de domínio (DSL) simplificada, juntamente com um interpretador desenvolvido em Python, que funcione de forma totalmente autônoma em relação a ambientes industriais. Essa abordagem visa fornecer uma alternativa leve e acessível para validação e simulação de lógica de controle estruturada, tanto em contextos educacionais quanto de prototipagem rápida.

O escopo da aplicação está restrito a: (i) suporte a tipos de dados primitivos (`INT`, `REAL`, `BOOL`), (ii) declaração e atribuição de variáveis, (iii) operadores aritméticos e lógicos, (iv) estruturas de controle condicionais e iterativas (`IF`, `WHILE`, `FOR`) e (v) saída de dados via `PRINT`. A proposta não contempla integração com hardware real, manipulação de entradas e saídas físicas nem implementação de blocos funcionais completos, visando preservar a simplicidade e garantir a viabilidade do projeto dentro dos limites de tempo e recursos disponíveis.

A relevância deste contexto reside na crescente necessidade de ferramentas didáticas e modulares que apoiem a formação de profissionais e o desenvolvimento incremental de soluções em automação industrial. A possibilidade de testar lógicas de controle em um ambiente interpretado e portátil contribui diretamente para o aprendizado da linguagem e para a melhoria da qualidade do código.

A escolha metodológica é fundamentada na norma IEC 61131-3, que orienta a estrutura sintática da linguagem, e no uso de uma **gramática formal expressa em EBNF**, conforme discutido por Wirth [3]. A análise léxica foi implementada manualmente com base em expressões regulares, e a análise sintática foi construída por meio de um **parser descendente recursivo**, sem o uso de geradores automáticos de parser. Essa abordagem permite controle total sobre o fluxo de interpretação e facilita a incorporação de regras semânticas diretamente no processo de execução.

III. Metodologia

A metodologia deste trabalho parte dos fundamentos estruturais da norma IEC 61131-3, especialmente no que se refere à definição formal de linguagens de programação para CLPs e à linguagem textual Structured Text (ST), conhecida como Structured Control Language (SCL). Entretanto, para viabilizar a construção de uma ferramenta funcional, leve e didática em ambiente acadêmico, diversas adaptações foram necessárias em relação às abordagens formais e industriais normalmente utilizadas.

A seguir, são descritas as principais modificações metodológicas implementadas, bem como suas justificativas com base nas limitações e objetivos do novo contexto:

A. *Redução do escopo da linguagem*

A norma IEC 61131-3 contempla uma gama extensa de elementos sintáticos e semânticos, incluindo blocos de função (FBs), tipos compostos, acesso a hardware e operadores bit a bit. Neste projeto, a linguagem foi propositalmente reduzida a um subconjunto essencial, composto por tipos primitivos (`INT`, `REAL`, `BOOL`), estruturas de controle (`IF`, `WHILE`, `FOR`), operadores lógicos e aritméticos e comandos de atribuição. Essa simplificação foi adotada para garantir clareza didática, viabilidade de implementação e foco na lógica de controle estruturada, conforme o tempo e escopo disponíveis.

B. *Interpretação direta do código-fonte*

Ao invés de desenvolver um compilador que gere código intermediário ou binário, optou-se pela construção de um **interpretador** capaz de avaliar diretamente o código-fonte a partir de sua estrutura sintática. A execução é realizada por meio de uma árvore de análise e uma tabela de símbolos que armazena valores e tipos das variáveis. Essa escolha permite resposta imediata à execução de trechos de código e facilita o processo de depuração e aprendizagem incremental.

C. *Parser manual por análise descendente recursiva*

Diferentemente de soluções que utilizam ferramentas automatizadas como `PLY` ou `Lark`, este projeto adota uma abordagem manual de parsing baseada em **análise sintática descendente recursiva**. Essa escolha confere maior controle sobre o fluxo de execução e possibilita a implementação direta de verificações semânticas dentro das funções de parsing, além de facilitar a personalização da linguagem e o entendimento do processo por estudantes.

D. *Execução desacoplada de ambientes industriais*

A ferramenta foi desenvolvida para funcionar de maneira completamente autônoma, sem qualquer dependência de plataformas industriais como o TIA Portal. A execução é realizada em ambiente Python puro, o que permite simular a lógica de controle sem a necessidade de hardware ou licenças específicas. Essa independência torna o interpretador portátil, acessível e adequado a contextos educacionais, projetos de pesquisa e prototipagem rápida.

E. *Simplificação da análise semântica*

A verificação semântica implementada foi limitada à verificação de escopo e consistência de tipos durante as atribuições e avaliações de expressões. Recursos como inferência de tipos, estruturas compostas, validação de fluxo e chamadas de função não foram incluídos. Essa simplificação reduz a complexidade do sistema e atende aos principais objetivos de ensino e prototipação de lógica básica.

IV. Desenvolvimento Prático da Aplicação

O desenvolvimento da aplicação foi realizado de forma incremental e modular, respeitando as fases típicas da construção de uma linguagem interpretada: definição léxica, análise sintática, semântica básica e execução virtual. Todo o projeto foi implementado utilizando a linguagem **Python 3**, por sua legibilidade, ampla aceitação educacional e recursos nativos que facilitam o desenvolvimento de linguagens formais.

A seguir, descreve-se o passo a passo do desenvolvimento prático da aplicação:

A. *Definição dos tokens e tipos léxicos*

O primeiro passo consistiu na criação do arquivo `token_definitions.py`, responsável por definir os **tokens** utilizados pela linguagem DSL. Foram especificadas as categorias léxicas essenciais, como palavras-chave (`IF`, `WHILE`, `PRINT`, etc.), operadores (`:=`, `+`, `<>`), tipos de dados (`INT`, `REAL`, `BOOL`) e literais (`TRUE`, `FALSE`, números). Cada token é representado por um identificador constante, acompanhado de uma classe `Token` que armazena o tipo, valor, posição e linha de ocorrência.

B. *Implementação do analisador léxico*

No arquivo `scl_parser.py`, foi implementada a função `_get_next_token`, responsável por analisar o texto de entrada caractere a caractere, montando os tokens definidos anteriormente. A implementação reconhece padrões léxicos por meio de expressões regulares e lógica condicional explícita, dispensando o uso de bibliotecas externas como PLY ou Lark. Esse analisador também descarta espaços e comentários em estilo `//`.

C. *Construção do parser descendente recursivo*

A análise sintática foi implementada manualmente por meio de chamadas recursivas (parser descendente), também dentro do arquivo `scl_parser.py`. Cada regra da gramática é representada por uma função Python, como `statement`, `expression`, `if_statement`, etc. O parser consome os tokens gerados e valida a estrutura de comandos da DSL. Esse modelo facilita o controle de erros e a personalização da sintaxe.

D. *Execução interpretada com tabela de símbolos*

Ao mesmo tempo em que valida a sintaxe, o parser executa diretamente os comandos por meio de uma **tabela de símbolos**, implementada como um dicionário Python. Essa estrutura armazena variáveis, seus tipos e valores. Com isso, o interpretador simula o comportamento de uma CPU virtual, capaz de avaliar expressões, realizar atribuições, aplicar operadores lógicos e matemáticos, além de controlar o fluxo com comandos `IF`, `WHILE`, `FOR`.

E. *Arquitetura modular do projeto*

A aplicação foi organizada em quatro módulos principais:

- `token_definitions.py`: definição de tokens e classe `Token`.
- `scl_parser.py`: analisador léxico, parser e executor.
- `fho_parser_base.py`: classe abstrata com lógica de controle de fluxo por caractere (base técnica para evolução futura).
- `main.py`: ponto de entrada da aplicação, contendo o código de teste e execução.

F. *Disponibilidade do Código-Fonte*

O código-fonte completo do interpretador InterSCL, juntamente com a documentação e exemplos de uso, está disponível publicamente no repositório GitHub do projeto [1]. O acesso ao repositório permite a reprodutibilidade dos resultados apresentados e futuras contribuições para o desenvolvimento da ferramenta.

G. *Etapas de prototipagem e testes*

O desenvolvimento seguiu uma abordagem iterativa. A cada etapa implementada — seja na definição léxica, regras sintáticas ou operadores — um novo trecho de código em SCL era adicionado ao arquivo `main.py` para teste. Esses testes incluem: declarações de variáveis, operações com inteiros e reais, lógica condicional, laços `FOR` e `WHILE`, além da verificação da saída com o comando `PRINT`.

Os testes foram realizados de forma manual, verificando se os valores armazenados na tabela de símbolos condiziam com os resultados esperados. Casos de erro sintático e semântico também foram contemplados, com mensagens claras sendo emitidas para facilitar o diagnóstico.

V. Avaliação e Justificativa de Escolhas Técnicas

Durante o desenvolvimento do interpretador da linguagem DSL inspirada na SCL, as escolhas técnicas foram pautadas por critérios de viabilidade, clareza didática e independência de ferramentas externas. As decisões adotadas visam garantir que a aplicação seja acessível, compreensível e útil tanto para fins educacionais quanto para prototipagem prática.

A. *Escolha da linguagem Python*

A linguagem Python foi escolhida como base para a implementação por oferecer alta legibilidade, ampla disponibilidade em ambientes acadêmicos e suporte a paradigmas de programação estruturada e orientada a objetos. Além disso, sua sintaxe limpa facilita a implementação manual de analisadores léxicos e sintáticos, permitindo que os conceitos de compiladores e linguagens formais sejam explorados com profundidade.

[1] https://github.com/EliezerCarretin/InterSCL

B. *Parser manual e sem bibliotecas externas*

Ao invés de utilizar frameworks externos como `PLY` ou `Lark`, foi adotada uma abordagem manual baseada em análise descendente recursiva. Essa escolha permite controle total sobre o processo de parsing, facilita a implementação direta de verificações semânticas e evita a complexidade adicional imposta por bibliotecas geradoras de código. Além disso, essa estratégia é mais adequada para contextos educacionais, onde a compreensão do funcionamento interno do parser é fundamental.

C. *Estrutura modular e separação de responsabilidades*

A aplicação foi estruturada em módulos distintos: definição de tokens, parser e executor, classe base abstrata e script principal de testes. Essa separação de responsabilidades favorece a manutenção do código, facilita a adição de novos recursos e melhora a organização do projeto como um todo.

D. *Critérios de funcionalidade da aplicação*

Para considerar a aplicação funcional, foram estabelecidos os seguintes critérios:

- **Reconhecimento correto de tokens:** o analisador léxico deve identificar corretamente palavras-chave, operadores, delimitadores, identificadores e literais.
- **Validação sintática:** a aplicação deve rejeitar trechos de código com estruturas inválidas e emitir mensagens de erro informativas.
- **Execução coerente:** a lógica contida nos blocos de código (`IF`, `FOR`, `WHILE`) deve ser corretamente interpretada, com avaliação de expressões e controle de fluxo.
- **Tabela de símbolos ativa:** a aplicação deve manter um registro atualizado dos valores das variáveis durante a execução.
- **Saída visível ao usuário:** comandos como `PRINT` devem apresentar resultados coerentes com o código fornecido.

Esses critérios garantem que o interpretador possa ser utilizado como ferramenta prática de simulação, verificação e aprendizado, mesmo sem integração com CLPs reais ou plataformas industriais.

VI. Resultados

Após a implementação do interpretador da linguagem DSL inspirada na SCL, foram conduzidos diversos testes com trechos de código representativos de lógicas de controle comuns. O objetivo foi validar o comportamento da ferramenta quanto à análise léxica, sintática e semântica, bem como à execução correta dos comandos definidos.

A. *Exemplo prático de execução*

O código a seguir foi utilizado como cenário de teste principal, no arquivo `main.py`:

```
REAL a;
REAL b;
REAL c;
BOOL resultado_final;
INT i;

a := 10.0;
b := 20.0;
c := 5.0;
resultado_final := FALSE;

PRINT a;

IF (a + b * 2.0 > 40.0) AND (a <> b) THEN
    resultado_final := TRUE;
END_IF;

PRINT resultado_final;

FOR i := 1 TO 3 DO
    c := c + i;
    PRINT c;
END_FOR;

PRINT c;
```

B. *Análise dos resultados obtidos*

Durante a execução, o interpretador produziu a seguinte saída no terminal:

```
[SAÍDA SCL] 10.0
[SAÍDA SCL] True
[SAÍDA SCL] 6.0
[SAÍDA SCL] 8.0
[SAÍDA SCL] 11.0
[SAÍDA SCL] 11.0
```

Cada valor impresso corresponde corretamente ao resultado esperado a partir da lógica do código:

- O primeiro `PRINT a;` exibe `10.0`, valor inicial de `a`.
- A expressão lógica avaliada no bloco `IF` resulta em `TRUE`, conforme esperado.
- O laço `FOR` incrementa `c` de forma cumulativa com os valores de `i`, resultando em `6.0`, `8.0` e `11.0`.
- A variável `c`, ao final do laço, contém corretamente o valor acumulado final: `11.0`.

C. *Verificação da tabela de símbolos*

Ao término da execução, a estrutura interna do interpretador apresenta uma tabela de símbolos coerente, com tipos e valores atualizados corretamente para cada variável. Um exemplo simplificado de estado final:

```
{
  "a": {"type": "REAL", "value": 10.0},
  "b": {"type": "REAL", "value": 20.0},
  "c": {"type": "REAL", "value": 11.0},
  "resultado_final": {"type": "BOOL", "value": Tru
  "i": {"type": "INT", "value": 3}
}
```

D. *Conclusão dos testes*

Com base nos testes realizados, conclui-se que a aplicação atende a todos os critérios de funcionalidade definidos:

- Reconhecimento preciso de tokens;
- Validação sintática e emissão de erros em casos incorretos;
- Execução lógica conforme especificado;
- Atualização consistente da tabela de símbolos;
- Saída correta e imediata dos resultados.

Os resultados demonstram que o interpretador é plenamente capaz de simular trechos de código SCL de forma isolada e confiável, tornando-se uma ferramenta eficaz para aprendizado, prototipagem e verificação de lógica de controle.

VII. Conclusão

Este trabalho apresentou o desenvolvimento de um interpretador leve e funcional para uma linguagem específica de domínio (DSL) inspirada na Structured Control Language (SCL), conforme definida pela norma IEC 61131-3. A aplicação foi inteiramente desenvolvida em Python, utilizando um analisador léxico artesanal e um parser descendente recursivo, sem depender de ferramentas externas. O interpretador é capaz de simular trechos de lógica de controle com estruturas básicas como declaração de variáveis, atribuições, expressões aritméticas e lógicas, além de comandos de fluxo (`IF`, `WHILE`, `FOR`) e saída com `PRINT`.

A aplicação foi validada por meio de testes com diferentes estruturas e blocos de código, demonstrando comportamento compatível com a semântica esperada da linguagem SCL. A manutenção da tabela de símbolos e a execução coerente das instruções mostram que a ferramenta atende aos critérios de funcionalidade propostos.

Do ponto de vista educacional, a ferramenta se mostra promissora por permitir que estudantes e desenvolvedores testem, validem e compreendam a lógica de programação de CLPs de forma modular, sem necessidade de hardware industrial ou softwares proprietários.

Apesar de seu sucesso dentro do escopo proposto, o projeto possui limitações importantes. Não há suporte, por exemplo, para tipos compostos, estruturas aninhadas complexas, funções ou blocos de função (FBs), nem para operações de entrada/saída física. A análise semântica é restrita a escopo e tipagem básica, e não há integração com hardware nem interface gráfica.

Como trabalhos futuros, destacam-se as seguintes possibilidades:

- Inclusão de suporte a funções, blocos compostos e chamadas parametrizadas;
- Expansão da análise semântica com inferência de tipos e verificação de escopo mais refinada;
- Integração com CLPs reais ou simuladores industriais (como OpenPLC);
- Criação de uma interface gráfica para facilitar a inserção e visualização de código e resultados;
- Aplicação do interpretador em ambientes educacionais com validação em tempo real.

Conclui-se, portanto, que o projeto alcançou seu objetivo principal: criar uma ferramenta prática e educativa que simula a execução de lógicas industriais escritas em SCL, com independência de plataformas industriais e foco em portabilidade, didática e extensibilidade.

References

[1] International Electrotechnical Commission. **IEC 61131-3:** *Programmable controllers – Part 3: Programming languages,* 3rd ed., Geneva, Switzerland: IEC, 2013.

[2] WIRTH, Niklaus. *What can we do about the unnecessary diversity of notation for syntactic definitions?* 1996. Disponível em: https://people.inf.ethz.ch/wirth/Articles/EBNF.pdf. Acesso em: 5 maio 2025.

[3] WIRTH, Niklaus. *What can we do about the unnecessary diversity of notation for syntactic definitions?* 1996. Disponível em: https://people.inf.ethz.ch/wirth/Articles/EBNF.pdf. Acesso em: 5 maio 2025.