UNIOESTE – UNIVERSIDADE ESTADUAL DO OESTE DO PARANÁ

CENTRO DE ENGENHARIAS E CIÊNCIAS EXATAS

CURSO DE CIÊNCIA DA COMPUTAÇÃO

**RELATÓRIO TÉCNICO ACERCA DO ANALISADOR LÉXICO DA LINGUAGEM *PASCYTHON***

Vinícius de Oliveira Dias

FOZ DO IGUAÇU

2023

Vinícius de Oliveira Dias

**RELATÓRIO TÉCNICO ACERCA DO ANALISADOR LÉXICO DA LINGUAGEM *PASCYTHON***

Trabalho submetido à disciplina de Compiladores da Universidade Estadual do Oeste do Paraná, Curso de Ciência da Computação - Campus de Foz do Iguaçu, como requisito parcial para conclusão da disciplina.

Orientadora: Prof.ª Camille Frazao Bordini

FOZ DO IGUAÇU

2023

# RESUMO

O presente estudo apresenta um relatório técnico detalhado sobre o analisador léxico desenvolvido para a linguagem de programação procedural e não orientada a objetos denominada "Pascython". O analisador léxico desempenha um papel fundamental na interpretação e compreensão de programas escritos em Pascython, uma linguagem desenvolvida pelos autores.

Este relatório aborda os principais aspectos do analisador léxico, incluindo sua função na análise de código-fonte, sua estrutura e funcionamento interno. Além disso, serão apresentados exemplos práticos de como o analisador léxico é utilizado no contexto da linguagem Pascython, destacando sua importância na identificação de tokens e na preparação do código-fonte para análises posteriores.

**Palavras-chave**: Analisador léxico, analisador de código-fonte, Pascython.

**LISTA DE FIGURAS**

[Figura 1: Abertura do analisador léxico 8](#_Toc146563520)

[Figura 2: Menu de opções. 8](#_Toc146563521)

[Figura 3: Leitura do arquivo texto. 9](#_Toc146563522)

[Figura 4: análise completa. 9](#_Toc146563523)

[Figura 5: Opção "2" para entrada por teclado. 9](#_Toc146563524)

[Figura 6: encerrar o programa. 10](#_Toc146563525)

**SUMÁRIO**

[RESUMO 3](#_Toc146563502)

[1 INTRODUÇÃO 6](#_Toc146563503)

[2 DESCRIÇÃO DO FUNCIONAMENTO 7](#_Toc146563504)

[2.1 INSTRUÇÕES DE USO 7](#_Toc146563505)

[3 DESCRIÇÃO DA LINGUAGEM PASCYTHON 11](#_Toc146563506)

[3.1 TRATAMENTO DE ERROS 15](#_Toc146563507)

[4 DOCUMENTAÇÃO DAS FUNÇÕES 16](#_Toc146563508)

[5 PROCESSO DE DESENVOLVIMENTO 20](#_Toc146563509)

[6 CONCLUSÃO 22](#_Toc146563510)

[7 REFERÊNCIAS 23](#_Toc146563511)

# 1 INTRODUÇÃO

Este relatório apresenta o resultado do trabalho desenvolvido pela equipe como parte do Trabalho 1 da disciplina de Compiladores. O projeto consistiu na implementação de um *software* que simula a fase de análise léxica de um compilador, com foco na linguagem de programação Pascython, desenvolvido especificamente para o projeto. A análise léxica é uma etapa crucial no processo de compilação, na qual o código-fonte é dividido em lexemas e *tokens* para posterior processamento.

O *software* desenvolvido desempenha um papel fundamental na linguagem Pascython, permitindo a verificação da correta estruturação dos códigos-fonte e a identificação de erros. Ele é capaz de processar arquivos de texto gerados a partir do código-fonte Pascython, analisando os *tokens* presentes em cada lexema. Caso ocorram erros no código-fonte, o *software* é projetado para identificar o tipo de erro e a posição exata onde ocorreu. Isso auxilia os desenvolvedores na rápida correção de problemas, aprimorando a qualidade e a confiabilidade de seus programas.

O relatório técnico que segue descreverá em detalhes o funcionamento do *software* de análise léxica, incluindo informações sobre como utilizá-lo de forma autônoma. Além disso, será apresentada uma visão geral da linguagem Pascython, com todas as expressões regulares e autômatos utilizados para o reconhecimento de *tokens*, juntamente com a documentação das principais funções do código-fonte. Também será abordado o processo de desenvolvimento, incluindo configurações, bibliotecas, ferramentas auxiliares e a IDE (Ambiente Integrado de Desenvolvimento) utilizada.

Este trabalho representou um desafio significativo, exigindo a aplicação de conhecimentos teóricos em compiladores, linguagens de programação e análise léxica na prática. A seguir, apresentaremos em detalhes todas as etapas do desenvolvimento, os desafios encontrados e as soluções implementadas para atender às especificações do projeto.

# 2 DESCRIÇÃO DO FUNCIONAMENTO

O *software* analisador léxico desempenha um papel fundamental na linguagem de programação Pascython, permitindo verificar a correta estruturação de códigos-fonte e identificar erros. Este *software* tem a capacidade de processar arquivos texto gerados a partir do código-fonte Pascython e analisar os *tokens* presentes em cada lexema desse arquivo.

Caso ocorram erros no código-fonte, o *software* é projetado para identificar o tipo de erro e em qual lexema ele ocorreu. Isso ajuda a corrigir problemas rapidamente, melhorando a qualidade e a confiabilidade de seus programas.

O analisador léxico é flexível em relação à entrada de dados. Ele pode processar códigos a partir de arquivos de texto ou receber a entrada diretamente do teclado, permitindo que os desenvolvedores escolham a maneira mais conveniente de fornecer o código-fonte para análise.

A seguir serão descritas as instruções de uso do analisador.

## 2.1 INSTRUÇÕES DE USO

As instruções de uso a seguir são descritas para a IDE *Visual Studio Code* (VS), onde o código-fonte foi desenvolvido. A interface também é exibida no terminal da plataforma.

1. realizar a instalação da IDE. Muitos tutorais de instalação estão disponíveis on-line;
2. instalações do Bison e Flex;

- baixar e instalar o Flex (<https://sourceforge.net/projects/gnuwin32/files/flex/2.5.4a-1/>);

- baixar e instalar o Bison ([https://sourceforge.net/projects/gnuwin32/files/flex/2.5.4ª -1/](https://sourceforge.net/projects/gnuwin32/files/flex/2.5.4a-1/))

- configurar as variáveis de ambiente.

1. abrir os arquivos de programa no VS:

- abrir a aba “*File*”;

- selecionar a opção “*Add Folder to Workspace*...”;

- selecionar o arquivo do código-fonte;

1. para executar, abrir a aba “*Terminal*”;
2. selecionar a opção “*New Terminal*”;
3. caso haja mais de um arquivo no *workspace*, selecionar o arquivo;
4. no terminal, digitar os seguintes comandos. Após digitar cada comando, aperte a tecla “*Enter*”:

- flex .\pascython.l

- gcc \*.c -o “.\main.exe”

- .\main.exe

d

Agora, o *software* do analisador possui as seguintes funcionalidades:

1. página de abertura (Figura 1). Para prosseguir basta apertar qualquer tecla do teclado;

Figura 1: Abertura do analisador léxico

Tela de computador com texto preto sobre fundo branco

Descrição gerada automaticamente

Fonte: foto do autor (2023).

1. menu de opções (Figura 2). Basta digitar o número identificador para selecionar a opção desejada;

Figura 2: Menu de opções.

Interface gráfica do usuário, Texto

Descrição gerada automaticamente

Fonte: foto do autor (2023).

1. a opção “1” (Figura 3) trata de ler o arquivo texto com o código-fonte escrito em linguagem Pascython para análise. Basta escrever o caminho do seu arquivo. Exemplo: “nome\_arquivo.txt”.

Figura 3: Leitura do arquivo texto.

Forma

Descrição gerada automaticamente

Fonte: foto do autor (2023).

1. apenas é impresso na tela as opções erradas, a não ser que o usuário selecione a opção que imprima na tela a análise completo (Figura 4):

Figura 4: análise completa.

Forma

Descrição gerada automaticamente

Fonte: foto do autor (2023).

1. na opção “2” (Figura 5), é possível digitar várias linhas de código para análise (digite uma linha de código por vez). Para sair dessas opções, é preciso digitar “*exit*”:

Figura 5: Opção "2" para entrada por teclado.

Interface gráfica do usuário, Texto

Descrição gerada automaticamente

Fonte: foto do autor (2023).

1. a opção “3” (Figura 6) encerra o programa:

Figura 6: encerrar o programa.

Interface gráfica do usuário, Texto

Descrição gerada automaticamente

Fonte: foto do autor (2023).

# 3 DESCRIÇÃO DA LINGUAGEM PASCYTHON

A linguagem Pascython é uma fusão das características da linguagem C e Python, criando assim uma abordagem única no desenvolvimento de *software*. Uma de suas particularidades notáveis é a eliminação do uso do ponto e vírgula ";" ao final de cada comando, simplificando a escrita de código e tornando-o mais conciso e legível. Abaixo, serão detalhadas suas principais características.

Permite comandos condicionais dos tipos “if”, “else”, “switch”, “case”:

Trecho de Código 1: comandos condicionais.

*/\*Comandos condicionais\*/*

IF (if)

ELSE (else)

SWITCH (switch)

CASE (case)

Fonte: autor (2023).

Permite palavras reservados do tipo “return”, “remove”, “typedef”, “include”, “define”:

Trecho de Código 2: palavras-reservadas.

*/\*Palavras reservadas\*/*

RETURN (return)

REMOVE (remove)

TYPEDEF (typedef)

INCLUDE (include)

DEFINE (define)

Fonte: autor (2023).

Permite símbolos separadores do tipo "{}", "[]", e "()", o que facilita a organização e estruturação de blocos de código:

Trecho de Código 3: símbolos separadores.

*/\*Símbolos separadores\*/*

PARENTESES [()]

CHAVES [{}]

COLCHETES [[]]

Fonte: autor (2023).

Permite comandos de repetição do tipo "for", "while" e "do", oferecendo flexibilidade no controle de fluxo de programas:

Trecho de Código 4: comandos de repetição.

*/\*Funções de repetição\*/*

FOR (for)

WHILE (while)

DO (do)

Fonte: autor (2023).

Contém funções de entrada e saída de dados do tipo "printf", "scanf", "fprintf", "fscanf", que possibilitam a interação com o usuário e a manipulação de arquivos:

Trecho de Código 5: funções de entrada e saída de dados.

*/\*Funções de entrada/saída de dados\*/*

PRINTF (printf)

SCANF (scanf)

FSCANF (fscanf)

FPRINTF (fprintf)

Fonte: autor (2023).

Além disso, contém funções prontas do tipo "sizeof", "malloc", "strlen", "strcmp", "strcpy", que simplificam tarefas comuns de manipulação de memória e strings:

Trecho de Código 6: funções extras.

*/\*Funções extras\*/*

SIZEOF (sizeof)

MALLOC (malloc)

STRLEN (strlen)

STRCMP (strcmp)

STRCPY (strcpy)

Fonte: autor (2023).

Os tipos de dados existentes são "int", "char", "float", "double", "long", "short", "void" e "struct", oferecendo uma ampla gama de escolhas para representar diferentes tipos de dados:

Trecho de Código 7: tipos de dados.

*/\*Tipos de dados\*/*

STRUCT (struct)

FLOAT (float)

INT (int)

DOUBLE (double)

CHAR (char)

VOID (void)

LONG (long)

SHORT (short)

Fonte: autor (2023).

Contém os operadores "+", de adição, "-", de subtração, "\*", de multiplicação, "/", de divisão, e "%", para cálculos de resto, permitindo operações matemáticas em expressões:

Trecho de Código 8: operadores.

*/\*Operadores\*/*

ADD ("+")

SUB ("-")

DIV ("/")

MULT ("\*")

RESTO ("%")

Fonte: autor (2023).

Permite o uso de operadores lógicos como "&&" (AND), "||" (OR) e "!" (NOT), tornando possível a criação de expressões lógicas complexas:

Trecho de Código 9: operadores lógicos.

*/\*Símbolos lógicos\*/*

AND (&&)

OR ("||")

NOT ("!")

Fonte: autor (2023).

Contém os símbolos de atribuição, como "+=", "-=", "/=", "\*=", e "%=", que simplificam operações de atribuição e atualização de variáveis:

Trecho de Código 10: símbolos de atribuição.

*/\*Símbolos de atribuição\*/*

ATRIB ("=")

ATRIB\_ADD ("+=")

ATRIB\_SUB ("-=")

ATRIB\_DIV ("/=")

ATRIB\_MULT ("\*=")

ATRIB\_RESTO ("%=")

Fonte: autor (2023).

Além disso, disponibiliza os símbolos de comparação do tipo "EQUALS" (==), "GREATER-EQUAL" (>=), "LESS-EQUAL" (<=), "NOT-EQUAL" (!=), "GREATER-THAN" (>) e "LESS-THAN" (<) para avaliar e comparar valores:

Trecho de Código 11: símbolos de comparação.

*/\*Símbolos de comparação\*/*

EQUAL ("==")

MAIOR\_EQUAL (>=)

MENOR\_EQUAL (<=)

DIFERENTE (!=)

MAIOR (">")

MENOR ("<")

Fonte: autor (2023).

O alfabeto corresponde às letras maiúsculas e minúsculas, abrangendo de 'a' até 'z', permitindo a criação de identificadores descritivos:

Trecho de Código 12: alfabeto.

*/\*Alfabeto\*/*

LETRA [a-zA-Z\_]

Fonte: autor (2023).

Os dígitos variam de 0 a 9, e números são formados por 1 ou nenhum "+" ou "-", indicando números positivos ou negativos, seguidos por pelo menos 1 dígito e, opcionalmente, um ponto ".", seguido por 1 ou mais dígitos:

Trecho do Código 13: números.

*/\*Números\*/*

DIGITO [0-9]

NUMERO ("+"|"-")?{DIGITO}+|(("+"|"-")?{DIGITO}+.{DIGITO}+)

Fonte: autor (2023).

Os identificadores, ou variáveis, podem ser formados por uma letra inicial ou por letras seguidas de dígitos, proporcionando a flexibilidade necessária para nomear variáveis de acordo com a conveniência do programador.

Trecho de Código 14: identificadores.

IDENTIFICADOR {LETRA}({LETRA}|{DIGITO})\*

Fonte: autor (2023).

## 3.1 TRATAMENTO DE ERROS

O tratamento de erros é uma parte fundamental no desenvolvimento de qualquer linguagem de programação. Na linguagem "ERROVAR", os erros estão relacionados a variáveis e identificadores que têm um início inapropriado, começando por um dígito. Esse tipo de erro é identificado e tratado para garantir que as variáveis sigam as convenções de nomenclatura adequadas.

Por outro lado, temos o tratamento de erro chamado "ERROARROBA", que lida com a presença do símbolo "@" em locais onde não é reconhecido pela linguagem. Esse símbolo é considerado inválido e é tratado para evitar conflitos ou interpretações incorretas no código.

Finalmente, o tratamento de erro "ERRONUM" é destinado a lidar com números que contenham pontos de maneira inadequada. Nesse contexto, não é permitido que um dígito seja seguido imediatamente por um ponto, seguido por uma letra. Esse erro é identificado e corrigido para garantir que números sejam formatados corretamente, evitando ambiguidades ou erros de interpretação.

Trecho de Código 14: tratamento de erros.

ERROVAR {DIGITO}+{LETRA}+({DIGITO}|{LETRA})\*

ERROARROBA ({LETRA}|{DIGITO})("@")({LETRA}|{DIGITO})

ERRONUM {DIGITO}+.{LETRA}\*

Fonte: autor (2023).

# 4 DOCUMENTAÇÃO DAS FUNÇÕES

O método yyin() utilizado é uma função que normalmente é usada em conjunto com analisadores léxicos gerados pelo Flex. Sua principal função é configurar a fonte de entrada para o analisador léxico, no caso do projeto, um arquivo texto contendo as linhas de códigos. Ao chamar yyin(), será definido o contexto em que o analisador léxico vai operar, determinando de onde ele extrairá os dados a serem analisados.

O método yylex() é o coração do analisador léxico gerado pelo Flex. Ele desempenha o papel fundamental de ler os caracteres da fonte de entrada configurada com yyin() e transformá-los em *tokens* com base nas regras definidas no arquivo de especificações Flex. Sempre que yylex() é chamado, o analisador léxico lê o próximo caractere ou sequência de caracteres da fonte de entrada e aplica as regras correspondentes para identificar e produzir *tokens*.

O método yyrestart() é usado para reiniciar o analisador léxico gerado pelo Flex. Ele é especialmente útil quando você deseja alterar a fonte de entrada durante a análise léxica. Ao chamar yyrestart(), é possível especificar uma nova fonte de entrada com yyin() e reiniciar a análise a partir do início dessa nova entrada.

O método noyywrap() está relacionado à manipulação do final do arquivo (EOF) pelo analisador léxico gerado pelo Flex. Quando chamado, noyywrap() indica ao analisador léxico que ele não deve realizar uma ação especial quando atingir o final do arquivo de entrada. Ele faz com que o analisador continue a processar várias fontes de entrada em sequência, sem interrupções ao encontrar o final de um arquivo. O analisador simplesmente continua a analisar a próxima entrada sem a necessidade de parar ou sinalizar o fim do processo.

Trecho de Código 16: arquivo principal do analisador (pascython.l)

switch (op) {

            case 1:

                printf("Digite o nome do arquivo.\n");

                scanf("%[^\n]%\*c", str);

                yyin = fopen(str, "r");

                if(yyin == NULL){ printf("Arquivo incorreto, digite novamente...\n"); break;}

                yyrestart(yyin);

                yylex();

                opcao\_print();

                scanf("%d%\*c", &*op*);

                if(op == 1) analise\_lexica(tab);

                else if(op != 2) printf("Opcao invalida, retornando ao menu...\n");

                free\_list(tab);

                count = 1;

                break;

            case 2:

                printf("\nDigite alguma linha de codigo para verificar erros:\n");

                yyrestart(stdin);

                yylex();

                count = 1;

                break;

            case 3:

                break;

            default:

                printf("Valor invalido!\n");

        }

Fonte: autor (2023).

No código fornecido, são utilizados métodos relacionados ao analisador léxico gerado pelo Flex. Se o usuário escolher a opção 1, o programa solicitará ao usuário que digite o nome de um arquivo a ser analisado. Em seguida, ele usa a função fopen() para abrir esse arquivo para leitura ("r") e atribui o ponteiro de arquivo resultante a yyin. Isso faz com que o analisador léxico leia a partir desse arquivo. Em seguida, o método yyrestart(yyin) é chamado para reiniciar o analisador léxico a partir do início do arquivo.

Depois disso, o analisador léxico é acionado com yylex(), o que inicia a análise léxica do código no arquivo especificado. Após a análise, há uma chamada para opcao\_print(), seguida de liberação de memória com free\_list(tab).

Se o usuário escolher a opção 2, o programa redefine o analisador léxico para ler da entrada padrão (teclado) com yyrestart(stdin). Isso permite que o usuário insira código-fonte diretamente no programa. A análise é realizada com yylex() e não há ação adicional após a análise nesse caso.

As chamadas de yylex() apresentadas estão relacionadas às regras de reconhecimento de *tokens* em um analisador léxico gerado pelo Flex. O analisador léxico examina o código-fonte de entrada e, com base nas regras definidas, identifica lexemas e os associa a tokens específicos.

Na primeira chamada de exemplo abaixo, {RETURN} indica que a regra será aplicada quando o analisador encontrar o lexema "RETURN" no código-fonte. Quando essa regra é acionada, o código dentro das chaves é executado. Nesse caso, uma entrada é adicionada à tabela de símbolos contendo informações sobre o lexema "RETURN," o token "keyword\_return," e o número da linha onde foi encontrado.

Trecho de Código 17: *token* {RETURN}.

{RETURN}            {

                        tab = insert\_tabela(tab, yytext, "keyword\_return", "\0", count);

                    }

Fonte: autor (2023).

Na segunda chamada, {ERRONUM} representa uma regra para identificar lexemas que correspondem a erros relacionados a números não reconhecidos pela linguagem. Quando essa regra é ativada, duas ações são executadas: primeiro, uma entrada é inserida na tabela de símbolos para registrar o erro, incluindo o lexema, o token "erro\_num," uma descrição do erro e o número da linha. Em seguida, uma mensagem de erro é impressa usando printf(), que inclui informações detalhadas sobre o lexema, o token, a descrição do erro e a linha onde o erro ocorreu.

Trecho de Código 18: *token* {ERRONUM}.

{ERRONUM}           {

                        tab = insert\_tabela(tab, yytext, "erro\_num", "numero nao reconhecido pela linguagem", count);

                        printf("Lexema: %s\nToken: %s\nErro: %s\n\n", yytext, "erro\_num", "numero nao reconhecido pela linguagem");

                    }

Fonte: autor (2023).

Essas regras são utilizadas para reconhecer e tratar casos específicos no código-fonte, registrando informações relevantes na tabela de símbolos e fornecendo mensagens de erro detalhadas quando necessário. Isso é essencial para uma análise léxica precisa e a detecção eficiente de erros em um programa.

Trecho de Código 19: função analise\_lexica(tab).

void analise\_lexica(Tabela\* *tab*){

    if(*tab* != NULL){

        if(strlen(*tab*->tipo\_erro) != 0) printf("Lexema: %s\nToken: %s\nErro: %s\nLinha: %d\n\n", *tab*->nome, *tab*->token, *tab*->tipo\_erro, *tab*->line);

        else printf("Lexema: %s\nToken: %s\nLinha: %d\n\n", *tab*->nome, *tab*->token, *tab*->line);

        analise\_lexica(*tab*->prox);

    }

}

Fonte: autor (2023).

A função opcao\_print() desempenha um papel interativo, onde o programa pergunta ao usuário se ele deseja imprimir a análise completa do texto analisado. Esta função é acionada quando o usuário escolhe a opção de análise. Ela permite ao usuário decidir se deseja visualizar detalhes da análise léxica, como os lexemas, *tokens*, eventuais erros e as linhas onde esses elementos foram encontrados.

Quanto às funções analise\_lexica(tab) e free\_list(tab), elas são responsáveis pela gestão da tabela de símbolos (tab). A função analise\_lexica(tab) percorre essa tabela e imprime informações sobre cada elemento, incluindo lexema, token, descrição de erro (se houver) e a linha correspondente. Isso fornece uma visão detalhada da análise léxica, facilitando a depuração do código.

No geral, essas funções trabalham em conjunto para garantir que a análise léxica seja precisa, fornecendo ao usuário a opção de visualizar os resultados detalhados e garantindo que a memória seja gerenciada de forma eficiente durante o processo. Isso contribui para a confiabilidade e robustez do analisador léxico.

# 5 PROCESSO DE DESENVOLVIMENTO

No processo de desenvolvimento do analisador léxico, diversas etapas foram realizadas, permitindo a criação de uma ferramenta para processar código-fonte na linguagem Pascython. Estas etapas abrangem desde a instalação das ferramentas necessárias até os testes finais do programa. Neste tópico, serão exploradas cada uma dessas etapas.

No projeto, a equipe utilizou a linguagem C e integrou o Flex para identificar lexemas. O Flex permitiu reconhecer palavras-chave, identificadores e números de forma eficiente. Além disso, foi adotada a IDE Visual Studio Code (VS Code) para edição e gerenciamento de código. A combinação dessas ferramentas possibilitou o desenvolvimento eficaz de um analisador léxico preciso. Essas escolhas contribuíram para um processo de desenvolvimento mais suave e a criação de software de alta qualidade.

O primeiro passo consistiu na instalação do analisador léxico Flex e na configuração das variáveis de ambiente para que o Flex pudesse ser facilmente utilizado. Isso estabeleceu a base para a análise léxica do código-fonte.

Para dar suporte ao desenvolvimento do programa, bibliotecas da linguagem C, como <stdio.h>, <string.h> e <stdlib.h>, foram importadas. Além disso, ao longo do desenvolvimento, bibliotecas personalizadas foram criadas para atender às necessidades específicas do projeto.

A próxima foi a definição de cada tipo de *token* que compõe a linguagem Pascython. Isso incluiu a identificação de palavras-chave, identificadores, números e outros elementos fundamentais que fazem parte da gramática da linguagem.

Para facilitar a interação com o usuário e o gerenciamento das informações, duas bibliotecas auxiliares foram desenvolvidas. A "interface.h" concentrou todas as funções de interação com o usuário, enquanto a "tabela.h" tratou da inserção e impressão de informações em uma estrutura de dados baseada em Lista Encadeada Simples.

Uma etapa central do projeto envolveu a criação das regras para cada tipo de *token* definido anteriormente. Essas regras foram essenciais para o reconhecimento dos elementos do código-fonte e a inserção das informações correspondentes na estrutura de dados.

O programa principal (main()) foi desenvolvido para coordenar todas as ações executadas pelo analisador léxico Flex. Ele desempenhou um papel fundamental na orquestração do processo de análise léxica.

Após a implementação do programa, foram realizados testes para garantir o funcionamento correto do analisador léxico. Vários ajustes necessários foram feitos nessa fase referente à implementação para melhorar a precisão e a confiabilidade do programa.

A etapa final envolveu um teste abrangente e completo do analisador léxico para verificar se ele atendia a todas as especificações e funcionava de maneira consistente. Esse teste final garantiu que o programa estivesse pronto para ser usado em cenários reais.

Essas etapas representam o ciclo de desenvolvimento que culminou na criação de um analisador léxico eficiente e preciso para a linguagem Pascython. Cada passo desempenhou um papel crucial na construção dessa ferramenta, e o processo de teste e ajuste final garantiu sua qualidade e confiabilidade.

# 6 CONCLUSÃO

O desenvolvimento do analisador léxico para a linguagem de programação Pascython representou uma oportunidade valiosa para aplicar conceitos teóricos em um ambiente prático e desafiador. Ao longo deste projeto, a equipe enfrentou uma série de tarefas complexas relacionadas à análise léxica, implementação de regras de reconhecimento de *tokens* e tratamento de erros. Este trabalho proporcionou uma compreensão mais profunda dos processos internos de um compilador e das nuances envolvidas na criação de ferramentas de análise de código.

O *software* desenvolvido mostrou-se eficaz na identificação de lexemas e *tokens* em código-fonte Pascython, demonstrando a capacidade de processar arquivos de texto e fornecer informações detalhadas sobre cada elemento analisado. A flexibilidade do programa, que permite a análise a partir de arquivos ou entrada direta do usuário, aumentou sua utilidade e conveniência.

Durante o processo de desenvolvimento, vários desafios técnicos e de implementação foram enfrentados, que exigiram um aprofundamento ainda maior dos conceitos aprendidos em sala de aula e da ferramenta Flex. A criação de bibliotecas personalizadas para gerenciar a interação com o usuário e a tabela de símbolos simplificou a complexidade do código e melhorou a legibilidade. Além disso, a integração do Flex como analisador léxico mostrou-se uma escolha acertada, proporcionando um mecanismo eficiente para o reconhecimento de lexemas.

O processo de desenvolvimento, que incluiu desde a instalação das ferramentas necessárias até os testes finais, refletiu a abordagem metodológica e organizada da equipe. Os testes rigorosos garantiram a precisão e confiabilidade do analisador, preparando-o para uso em situações do mundo real.

Em resumo, o projeto proporcionou uma valiosa experiência na criação de um analisador léxico eficiente para a linguagem Pascython. Foi aprendida a importância da análise léxica como uma etapa crítica no processo de compilação e a necessidade de lidar com diversos cenários de erros. O conhecimento adquirido neste projeto será aplicável em futuros desafios relacionados a compiladores e processamento de linguagens de programação, consolidando o entendimento e habilidades nesse campo da equipe.

# 7 REFERÊNCIAS

Frazao Bordini, Camile. Slide de aula sobre "**Análise Léxica**". 2023. Curso de Ciência da Computação, Universidade Estadual do Oeste do Paraná.