UNIVERSIDADE ESTADUAL DO OESTE DO PARANÁ
GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ALIANA WAKASSUGUI DE PAULA E SILVA

RELATÓRIO TÉCNICO

ANALISADOR SINTÁTICO DA LINGUAGEM Ç

2024

1 Resumo

A análise sintática é a fase onde um texto de entrada (como um código fonte da linguagem reconhecida pelo compilador) é organizado em estruturas. Neste relatório, será descrito o projeto de um compilador que está sendo desenvolvido para a disciplina de Compiladores, onde um analisador léxico já foi elaborado na etapa anterior e, agora, um analisador sintático foi desenvolvido para continuar o seu progresso. O projeto foi elaborado com a ajuda do *Flex Windows (Lex and Yacc)*. Neste relatório, entende-se por 'programa' o arquivo de texto contendo o código-fonte escrito na linguagem fictícia Ç, que é processada pelo compilador em projeto. O objetivo é, além de conseguir fazer as etapas do *Scanner* (Analisador Léxico), responsável por incluir as palavras-chaves do programa em tabelas de símbolos e de palavras reservadas, ser capaz de usá-las para forçar o programador a estabelecer uma estrutura formal e pré-definida da linguagem, seguindo regras e sentenças sintáticas. Por fim, o trabalho é útil para compreender a arquitetura de um compilador e a forma como as linguagens formais e os autômatos finitos são importantes e facilitadores para essa construção.

2 Introdução

Um *Parser* (analisador sintático) é a parte do compilador que, em conjunto com o scanner, estrutura o programa de entrada em uma árvore sintática. Ele decompõe os blocos de código em unidades menores, até cada um se tornar, dentro da ramificação da árvore, em tokens que o scanner delimitou. O esquema visual disso é representado na Figura 1.

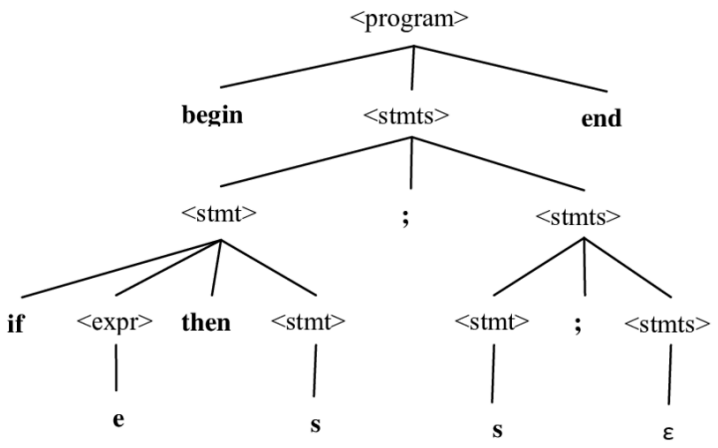

Figura 1 - Representação de uma árvore sintática e suas derivações. Fonte: PONCE et al. (2014).

A decomposição do programa é feita por meio de Gramáticas, em geral, Livres de Contexto, regras que estabelecem as possíveis formações das linhas de código. Por

exemplo, através da gramática da Figura 2, escrita em uma notação textual, é possível reconhecer sentenças que contém operações aritméticas, como:

2 + 3
4 * (5 + 6)
7 - 8 / 2

```
<expressao> ::= <termo> | <expressao> "+" <termo> | <expressao> "-" <termo>
<termo>     ::= <fator> | <termo> "*" <fator> | <termo> "/" <fator>
<fator>     ::= "(" <expressao> ")" | <numero>
<numero>    ::= <digito> | <numero> <digito>
<digito>    ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

Figura 2 - Gramática que reconhece operações aritméticas. Fonte: OPENAI, 2024.

A seguir, é descrito com mais detalhes como o sistema funciona e como foi desenvolvido.

3 Funcionamento do Software

Para desenvolver o projeto, foi usado o sistema auxiliar *Flex Windows (Lex and Yacc)*, que ajuda a automatizar etapas muito complexas e minuciosas. De forma resumida, ele faz o "grosso" da análise léxica e sintática, a parte de *pattern matching* de tokens de um programa e a organização de cada token em uma estrutura padronizada. O que precisou ser de fato projetado foram as normas gramaticais que a linguagem deve seguir e as estruturas auxiliares no processo.

3.1 Análise Léxica

3.1.1 A Linguagem Ç

Essa linguagem reconhece e utiliza os seguintes padrões ou classes de tokens relevantes para a compilação:

I. Literais:

`Identificador, Inteiro, Real, Caracter, Cadeia, Filename`

II. Tipos de Dados:

`Int, Float, Char, Void, String`

III. Operações Aritméticas:

`Soma, Multiplicação, Divisão, Subtração, Recebe`

IV. Operações Relacionais:

`Igual, Diferente, Maior, Menor, Maior ou Igual, Menor ou Igual`

V. Operações Lógicas:

`E, Ou, Not`

VI. Delimitadores:

`Abre Parêntesis, Fecha Parêntesis, Abre Chaves, Fecha Chaves, Vírgula, Ponto e Vírgula, Dois Pontos`

VII. Palavras reservadas de seleção e repetição:

`If, Else, While, For, Do, Switch, Case`

VIII. Comandos:

`Break, Return, Scan, Print, Define, Include`

IX. Outras palavras reservadas:

`Comentário, Default`

Para cada uma delas, existe uma regra de reconhecimento de expressão regular utilizando os operadores de concatenação, união, repetição, entre outros[1]. Abaixo são citadas todas elas, incluindo as intermediárias, seguidas do devido trecho em linguagem LEX, usada em software.

I. Literais:

[1] Licenciatura em Engenharia Informática – DEI/ISEP. Linguagens de Programação 2006/07: Ficha 1 - Introdução ao FLEX e expressões regulares. 2006.

LETRA = um caractere de ‘a’ até ‘z’ e de ‘A’ até ‘Z’.
DIGITO = um caractere de ‘0’ até ‘9’.
IDENTIFICADOR = uma Letra ou ‘_’ seguido de Letra ou ‘_’ ou Digito.
NATURAL_SEM_ZERO = um caractere de ‘1’ até ‘9’.
NATURAL = um Natural Sem Zero seguido de 0 ou mais Digitos ou apenas um ‘0’.
INTEIRO = um ‘-’ opcional seguido de um Natural.
REAL = um Inteiro seguido de ‘.’ e 0 ou mais Digitos opcionais.
CARACTER = qualquer caractere dentro de aspas simples ou nada dentro de aspas simples.
CADEIA = qualquer sequência de 0 ou mais caracteres, exceto as aspas duplas e a quebra de linha, dentro de aspas duplas.
FILENAME = um ou mais caracteres de ‘a’ até ‘z’, de ‘A’ até ‘Z’, ‘_’,‘/’, ‘-’ ou ‘.’ seguidos de “.(extensão)”.

```
LETRA               [a-zA-Z]
DIGITO              [0-9]
IDENTIFICADOR       (_|{LETRA})(_|{LETRA}|{DIGITO})*
NATURAL_SEM_ZERO    [1-9]
NATURAL             (({NATURAL_SEM_ZERO}{DIGITO}*)|0)
INTEIRO             [-]?{NATURAL}
REAL                {INTEIRO}([.]{DIGITO}*)?
CARACTER            ('.')|''
CADEIA              \"[^\"\n]*\"
FILE_NAME           [a-zA-Z0-9_/\.-]+(\.h|\.cpp|\.c|\.txt|\.js|\.py)
```

II. Tipos de Dados:

INT = sequência exata de caracteres “int”.
FLOAT = sequência exata de caracteres “float”.
CHAR = sequência exata de caracteres “char”.
VOID = sequência exata de caracteres “void”.
STRING = sequência exata de caracteres “string”.

```
INT                 int
FLOAT               float
CHAR                char
VOID                void
STRING              string
```

III. Operações Aritméticas:

SOMA = caractere ‘+’.
MULTIPLICACAO = caractere ‘*’.

```
DIVISAO = caractere '/'.
SUBTRACAO = caractere '-'.
RECEBE = caractere '='.
```

```
SOMA                "+"
MULTIPLICACAO       "*"
DIVISAO             "/"
SUBTRACAO           "-"
RECEBE              "="
```

IV. Operações Relacionais:

```
IGUAL = caracteres "==".
DIFERENTE = caracteres "!=".
MAIOR = caractere '>'.
MENOR = caractere '<'.
MAIOR_IGUAL = caracteres ">=".
MENOR_IGUAL = caracteres "<=".
```

```
IGUAL               "=="
DIFERENTE           "!="
MAIOR               ">"
MENOR               "<"
MAIOR_IGUAL         ">="
MENOR_IGUAL         "<="
```

V. Operações Lógicas:

```
E_LOGICO = caracteres "&&".
OU_LOGICO = caracteres "||".
NOT_LOGICO = caractere '!'.
```

```
E_LOGICO            "&&"
OU_LOGICO           "||"
NOT_LOGICO          "!"
```

VI. Delimitadores:

```
BRANCO = espaços em branco ou tabulações.
EOL = quebras de linha.
ABRE_P = caractere '('.
FECHA_P = caractere ')'.
ABRE_CH = caractere '{'.
FECHA_CH = caractere '}'.
```

```
VIRGULA = caractere ‘,‘.
PONTO_VIRGULA = caractere ‘;‘.
DOIS_PONTOS = caractere ‘:‘.
```

```
BRANCO              [ \t]+
EOL                 [\n\r]
ABRE_P              "("
FECHA_P             ")"
ABRE_CH             "{"
FECHA_CH            "}"
VIRGULA             ","
PONTO_VIRGULA       ";"
DOIS_PONTOS         ":"
```

VII. Palavras reservadas de seleção e repetição:

```
IF = sequência exata de caracteres “if”.
ELSE = sequência exata de caracteres “else”.
WHILE = sequência exata de caracteres “while”.
FOR = sequência exata de caracteres “for”.
DO = sequência exata de caracteres “do”.
SWITCH = sequência exata de caracteres “switch”.
CASE = sequência exata de caracteres “case”.
```

```
IF                  if
ELSE                else
WHILE               while
FOR                 for
DO                  do
SWITCH              switch
CASE                case
```

VIII. Comandos:

```
BREAK = sequência exata de caracteres “break”.
RETURN = sequência exata de caracteres “return”.
SCAN = sequência exata de caracteres “scan”.
PRINT = sequência exata de caracteres “print”.
DEFINE = sequência exata de caracteres “#define”.
INCLUDE = sequência exata de caracteres “#include”.
```

```
BREAK               break
RETURN              return
SCAN                scan
PRINT               print
DEFINE              #define
```

```
INCLUDE             #include
```

IX. Outras palavras reservadas:

```
COMENTARIO = sequência de 0 ou mais caracteres precedidos por "//".
DEFAULT = sequência exata de caracteres "default".
```

```
COMENTARIO          "//"(.*)
DEFAULT             default
```

Após as definições das classes, cada uma delas é tratada conforme seu tipo: palavra reservada ou símbolo, e é inserida na sua respectiva tabela.

Um exemplo de código reconhecido por esta linguagem seria o trecho:

```
#include "BIBLIOTECA_H"

int soma(int a, int b) {
      return a + b
}
```

O resultado da análise léxica deste código, após ser aceito, são as palavras detectadas e armazenadas nas tabelas, invisível ao programador na linha de comando. Uma representação das tabelas de símbolos e de palavras reservadas consequentes deste programa são mostradas abaixo (Tabela 1 e 2).

ID	Lexema	Categoria
97	a	Identificador
98	b	Identificador
432	soma	Identificador
953	"BIBLIOTECA_H"	Cadeia

Tabela 1 - Tabela de Símbolos. Fonte: autoria própria.

ID	40	41	43	44	123	125	331	672	775
Lexema	(	)	+	,	{	}	int	return	#include

Tabela 2 - Tabela de Palavras Reservadas. Fonte: autoria própria.

3.1.2 Tratamento de Erros

Quando ocorre algum erro léxico no programa, este é exibido na tela, mostrando em qual linha aconteceu e uma descrição do erro. Os erros reconhecidos pelo scanner são:

I. Erro 1 - reconhecimento de palavras reservadas mal formadas, como:

```
if0()
@include
```

II. Erro 2 - identificadores mal formados, que não seguem o padrão solicitado, como:

```
int 1x, variavel_*
```

III. Erro 3 - operadores mal formados, duplicados ou "compostos", como:

```
x ++ y
if(a !=== 0)
```

3.2 Análise Sintática

Enquanto o scanner faz o trabalho de alimentar as tabelas com os tokens, o parser se preocupa em capturar sentenças completas, e não apenas sequências pequenas de caracteres. Como dito, ele faz isso por meio de gramáticas. A seguir, são listadas as gramáticas, de forma resumida e sem considerar a declaração dos terminais (tokens), usadas no software.

```
<programa> ::= <sentencas_include> <sentencas_define> <funcoes>

<sentencas_include> ::= <sentenca_include> <sentencas_include>
                      | ε

<sentenca_include> ::= "#include" "<" <nome_arquivo> ">"
                     | "#include" <cadeia>

<sentencas_define> ::= <sentenca_define> <sentencas_define>
                     | ε

<sentenca_define> ::= "#define" <identificador> <valor_define>

<valor_define> ::= <expressao>

<funcoes> ::= <funcao> <funcoes>
            | ε
```

```
<funcao> ::= <tipo_dado> <identificador> "(" <parametros_funcao> ")"
<corpo>

<parametros_funcao> ::= <parametro_funcao>
                      | <parametro_funcao> "," <parametros_funcao>
                      | ε

<parametro_funcao> ::= <tipo_dado> <identificador>

<corpo> ::= "{" <comandos> "}"
          | ε

<comandos> ::= <comando> <comandos>
             | <comando>

<comando> ::= <decl_var>
            | <atribuicao>
            | <sentenca_if>
            | <sentenca_while>
            | <sentenca_for>
            | <sentenca_do>
            | <sentenca_scan>
            | <sentenca_print>
            | <chamada_funcao>
            | <sentenca_return>
            | <sentenca_switch>
            | “break”
            | ε

<decl_var> ::= <tipo_dado> <identificador> <decl_vars>
            | <tipo_dado> <atribuicao> <decl_vars>;

<decl_vars> ::= "," <identificador> <decl_vars>
            | "," <atribuicao> <decl_vars>
            | ε;

<atribuicao> ::= <identificador> "=" <expressao>
            | <identificador> "=" <atribuicao>
            | <identificador> "=" <chamada_funcao>;

<expressao> ::= <operacao>
            | <operando>;

<operacao> ::= <operando> <op> <operando>
            | <not_logico> <operando>;
```

```
<operando> ::= <literal>
            | "(" <operacao> ")"
            | <operacao>
            | "(" <operando> ")";

<op> ::= <op_aritmetico>
       | <op_relacional>
       | <op_logico>;

<sentenca_if> ::= "if" "(" <expressao> ")" <corpo>
                | "if" "(" <expressao> ")" <corpo> "else" <corpo>;

<sentenca_while> ::= "while" "(" <expressao> ")" <corpo>;

<sentenca_for> ::= "for" "(" <parametros_for> ")" <corpo>;

<parametros_for> ::= <parametro_decl_var> ";" <parametro_expressao> ";"
<parametro_atribuicao>;

<parametro_decl_var> ::= <decl_var>
                       | ε;

<parametro_expressao> ::= <expressao>
                        | ε;

<parametro_atribuicao> ::= <atribuicao>
                         | ε;

<sentenca_do> ::= "do" <corpo> "while" "(" <expressao> ")";

<sentenca_scan> ::= "scan" "(" <parametros_scan> ")";

<parametros_scan> ::= <identificador>
                    | <identificador> "," <parametros_scan>;

<sentenca_print> ::= "print" "(" <parametros_print> ")";

<parametros_print> ::= <cadeia> <parametros_print>
                     | <identificador> <parametros_print>
                     | ";" <parametros_print>
                     | ε;

<sentenca_return> ::= "return" <expressao>;

<sentenca_switch> ::= "switch" "(" <identificador> ")" "{" <cases> "}";

<cases> ::= <sentenca_case> <cases>
```

```
        | ε;

<sentenca_case> ::= "case" <literal> ":" <comandos>
                | "default" ":" <comandos>;

<chamada_funcao> ::= <identificador> "(" <argumentos> ")";

<argumentos> ::= <expressao>
             | <expressao> "," <argumentos"
             | ε;

<tipo_dado> ::= "int"
             | "float"
             | "char"
             | "void"
             | "string";

<literal> ::= <inteiro>
             | <real>
             | <caracter>
             | <cadeia>
             | <identificador>;

<op_aritmetico> ::= <soma>
                  | <subtracao>
                  | <multiplicacao>
                  | <divisao>;

<op_relacional> ::= <igual>
                  | <diferente>
                  | <maior>
                  | <menor>
                  | <maior_igual>
                  | <menor_igual>;

<op_logico> ::= <e_logico>
             | <ou_logico>
             | <e_logico>;
```

Quando cada regra é encontrada, seu valor é inserido na árvore de derivação na ordem de "exploração", ou seja, na ordem de chegada. Por exemplo, para esse programa na linguagem Ç abaixo, a árvore gerada seria a apresentada logo a seguir.

```
#include "BIBLIOTECA_H"

int soma(int a, int b) {
```

```
                    return a + b
                }
```

Árvore de Derivação:

```
Árvore de Derivação:
+- programa
|  +- sentencas_include
|  |  +- sentenca_include
|  |  |  -- #include
|  |  |  -- "BIBLIOTECA_H"
|  |  -- sentencas_include
|  -- sentencas_define
|  +- funcoes
|  |  +- funcao
|  |  |  -- int
|  |  |  -- soma
|  |  |  -- (
|  |  |  +- parametros_funcao
|  |  |  |  +- parametro_funcao
|  |  |  |  |  -- int
|  |  |  |  |  -- a
|  |  |  |  -- ,
|  |  |  |  +- parametros_funcao
|  |  |  |  |  +- parametro_funcao
|  |  |  |  |  |  -- int
|  |  |  |  |  |  -- b
|  |  |  -- )
|  |  |  +- corpo
|  |  |  |  -- {
|  |  |  |  +- comandos
|  |  |  |  |  +- comando
|  |  |  |  |  |  +- sentenca_return
|  |  |  |  |  |  |  -- return
|  |  |  |  |  |  |  +- expressao
|  |  |  |  |  |  |  |  +- operacao
|  |  |  |  |  |  |  |  |  +- operando
|  |  |  |  |  |  |  |  |  |  +- literal
|  |  |  |  |  |  |  |  |  |  |  -- a
|  |  |  |  |  |  |  |  |  +- op
|  |  |  |  |  |  |  |  |  |  -- +
|  |  |  |  |  |  |  |  |  +- operando
|  |  |  |  |  |  |  |  |  |  +- literal
|  |  |  |  |  |  |  |  |  |  |  -- b
|  |  |  |  -- }
|  |  -- funcoes
```

As folhas da árvore são sempre os terminais/tokens os quais o programa contém, e quanto mais alto o nível da árvore, mais genérica é aquela regra.

3.3 Execução

Para compilar um código nesta linguagem, é preciso tê-lo em um arquivo de texto e seguir os passos:

I. Redirecionar-se à pasta “src”, encontrada no caminho:

```
cd C:\Users\...\Trabalho\src
```

II. Compilar o parser em “analise_sintatica.y”, o scanner em “analise_lexica.l” e os arquivos .c no diretório pai:

```
yacc -d analise_sintatica.y

lex analise_lexica.l

gcc (Get-ChildItem -Recurse -Path "C:\Users\...\Trabalho" -Filter "*.c").FullName
-o gçç
```

III. Por fim, para testar o programa, basta chamar o executável “gçç.exe” seguido do nome do arquivo (redirecionar para a pasta onde ele se encontra, se necessário):

```
.\gçç ..\codigos_teste\teste_yacc.txt
```

Porém, na pasta do projeto já haverá o mesmo executável disponível para rapidamente testar um novo programa, sem a necessidade de executar os passos anteriores.

Depois da execução, os possíveis erros serão exibidos na caixa de comando, ou, as tabelas de tokens e uma mensagem de “árvore gerada” será exibida em caso de sucesso. O arquivo texto com a árvore de derivação gerada será criada na pasta “arvores_geradas”, com o primeiro nome igual ao nome do programa original.

Em caso de recompilação do mesmo arquivo, a árvore já criada para ele será atualizada (sobrescrita).

5 Construção do Software

O software foi construído na IDE Virtual Studio Code, na linguagem C e com o auxílio do sistema Flex Windows (Lex and Yacc), como mencionado. As principais bibliotecas usadas, além das de autoria própria, foram `stdio.h`, `string.h`, `stdlib.h`, `stdarg.h` e `libgen.h`.

O arquivo “analise_lexica.l” contém, principalmente, todas as regras criadas para o reconhecimento da linguagem. Ele contém a inclusão da biblioteca “hash_table.h”, as expressões regulares válidas para cada classe de *token* que executa as funções primordiais para armazenar as palavras (lexemas) em tabelas *hashing*[2], para abrir e ler o arquivo de código fonte desejado e para executar a função de análise léxica pré-definida pelo programa, “yylex()”.

No arquivo “hash_table.h” estão todos os métodos para tratar o armazenamento dos lexemas em tabelas dinâmicas. Ele possui a estrutura de uma tabela hashing para guardar as palavras reservadas[3] e os símbolos[4] da linguagem, uma para cada; possui funções de impressão, inicialização, inserção e busca em uma tabela.

Estrutura de uma tabela de símbolos e de palavras reservadas:

```
typedef struct simbolos {
    Simbolo **tabela;
} TabelaSimbolo;

typedef struct reservadas {
    Reservada **tabela;
} TabelaReservada;
```

Principais métodos para utilizar a tabela (impressão, inicialização e inserção):

```
// Imprime todas os símbolos e suas informações
// Entrada: ponteiro para uma TabelaSimbolo
// Saída: nenhuma
void imprimirTabelaSimbolo(TabelaSimbolo *T);

// Inicia a tabela de simbolo com seus valores iguais a NULL
// Entrada: nenhuma
// Saída: ponteiro para uma TabelaSimbolo
TabelaSimbolo* iniciarTabelaSimbolo();

// Insere um lexema na tabela de simbolos
// Entrada: ponteiro para string do lexema, ponteiro para uma TabelaSimbolo
// Saída: nenhuma
void inserirSimbolo(char *lexema, int categoria, TabelaSimbolo *T);
```

Estes mesmos métodos existem para a tabela de palavras reservadas.

[2] Estruturas de dados que mapeiam chaves a valores, permitindo acesso rápido e eficiente aos dados.
[3] Palavras reservadas são identificadores especiais que têm um significado específico e são usadas para definir a estrutura e a lógica dos programas.
[4] Símbolos são caracteres ou combinações de caracteres que operam diretamente sobre valores, expressões ou definem a estrutura sintática do código.

No arquivo “analise_sintatica.y” é descrito como devem ser as estruturas sintáticas de um programa na linguagem Ç. Ao cair em uma regra, um ou mais valores associados àquela sintaxe são inseridos numa pilha, conforme a análise é sendo feita e a gramática vai ficando mais alta e genérica, esses valores são desempilhados e um novo valor associado a ele é empilhado. Esse processo é feito de baixo (folhas) para cima (raíz). Os principais métodos para que isso ocorra são apresentados a seguir.

Estrutura de uma árvore:

```
typedef struct No {
    char valor[MAX];
    int num_filhos;
    struct No** filhos;
} No;
```

Métodos de empilhar, desempilhar, inserir na árvore, associar parente e gerar um arquivo com a árvore:

```
// Empilha um nó na pilha de filhos
// Entrada: pilha em questão e nó a ser empilhado
// Saída: nenhuma
void empilharFilho(PilhaFilhos** pilha, No* no);

// Desempilha um nó da pilha de filhos
// Entrada: pilha em questão
// Saída: nó desempilhado
No* desempilharFilho(PilhaFilhos** pilha);

// Associa filhos a um nó pai
// Entrada: ponteiro ao nó pai, pilha de nós e número de filhos
// Saída: nenhuma
void associarFilhos(No* pai, PilhaFilhos** pilha, int num_filhos);

// Insere um nó na árvore
// Entrada: pilha de nós, valor do nó e seu número de filhos
// Saída: nenhuma
void inserir(PilhaFilhos** pilha, char* valor, int num_filhos);

// Salva arvore em arquivi dado
// Entrada: ponteiro para a árvore, caminho para o diretório do arquivo e o
nome do arquivo
// Saída: nenhuma
// Pós-condição: o arquivo será salvo no caminho dado, com o nome
"arvore_NOME.txt"
void salvarArvoreEmArquivo(No* arvore, const char* caminhoDiretorio, const
char* nomeArquivoOriginal);
```

6 Conclusão

Este projeto teve como objetivo o desenvolvimento de um analisador sintático para a linguagem fictícia Ç, parte de um compilador em construção para a disciplina de Compiladores. O trabalho foi uma continuação do analisador léxico desenvolvido previamente, abordando a análise sintática necessária para estruturar o código fonte em uma árvore sintática. O uso do Flex Windows (Lex and Yacc) facilitou a automatização das etapas de reconhecimento e organização dos tokens, enquanto o desenvolvimento das gramáticas que definem a estrutura da linguagem foi a chave para a eficácia do analisador sintático.

A partir dos conceitos de gramáticas livres de contexto, foi possível estabelecer as regras que regem a formação das sentenças da linguagem, garantindo que o código escrito respeite a sintaxe definida e possa ser processado corretamente pelo compilador. O projeto permitiu a compreensão da importância da análise sintática na construção de compiladores, mostrando como ela pode ajudar na detecção de erros no código, antes mesmo da execução.

O trabalho também evidenciou a relevância das ferramentas Lex e Yacc para a automação de tarefas complexas, além da importância das gramáticas na definição da estrutura das linguagens de programação. A análise sintática, aliada à análise léxica, é essencial para transformar o código fonte em uma representação compreensível para o compilador, contribuindo para o avanço da construção de compiladores completos. Com isso, o projeto não apenas cumpriu seus objetivos acadêmicos, mas também proporcionou um aprendizado significativo sobre a construção de sistemas de compilação e o tratamento de linguagens formais.

Referências

[1] SILVA, Rômulo César. **Introdução a Linguagens Formais e Autômatos**. Unioeste, Junho de 2020.

[2] SCALA DOCUMENTATION. **Pattern Matching**. Disponível em: https://docs.scala-lang.org/tour/pattern-matching.html. Acesso em: 13 ago. 2024.

[6] LICENCIATURA EM ENGENHARIA INFORMÁTICA – DEI/ISEP. Linguagens de Programação 2006/07: **Ficha 1 - Introdução ao FLEX e expressões regulares**. 2006.

[7] ONLINEGDB. Disponível em: https://www.onlinegdb.com/. Acesso em: 13 ago. 2024.

SILVA, Rômulo César. **Hashing**. Unioeste, Junho de 2016.

UNIVERSIDADE DO VALE DO RIO DOS SINOS. CIÊNCIAS EXATAS E TECNOLÓGICAS – Curso de Informática. **COMPILADORES I**. Disciplina: Compiladores I. Professor responsável: Fernando Santos Osório. Semestre: 2006/2. Web: http://inf.unisinos.br/~osorio/compil.html.

NIEMANN, Tom. **LEX & YACC Tutorial**. Disponível em: http://epaperpress.com/lexandyacc/. Acesso em: 13/08/2024.

PONCE, H. et al. **Domain-specific languages for command and control systems**. 2014. Disponível em: https://www.researchgate.net/publication/265218184_DOMAIN-SPECIFIC_LANGUAGES_FOR_COMMAND_AND_CONTROL_SYSTEMS. Acesso em: 2 dez. 2024.

WIKIPÉDIA. **Análise sintática (computação)**. Disponível em: https://pt.wikipedia.org/wiki/An%C3%A1lise_sint%C3%A1tica_(computa%C3%A7%C3%A3o). Acesso em: 3 dez. 2024.