UNIVERSIDADE ESTADUAL DO OESTE DO PARANÁ
GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ALIANA WAKASSUGUI DE PAULA E SILVA

RELATÓRIO TÉCNICO

ANALISADOR LÉXICO DA LINGUAGEM Ç

2024

1 Resumo

O analisador léxico é uma das partes importantes de um compilador, é onde ocorre a varredura e a correção do código fonte em termos de vocabulário da linguagem. Cada linha é repartida entre várias subpalavras relevantes para a linguagem e, então, armazenadas. Este relatório descreve o projeto de um compilador que foi desenvolvido para a disciplina de Compiladores, em que um analisador léxico é elaborado com a ajuda do programa *Flex Windows (Lex and Yacc)*. O objetivo é conseguir ler um código fonte da linguagem criada, Ç, e então a compilar parcialmente através do analisador léxico implementado. O sistema irá varrer o código, incluir as palavras-chaves do programa em estruturas de dados e apontar erros léxicos caso existam. O trabalho é útil para compreender a forma como os programas de alto nível são traduzidos para outras linguagens, como as de montagem ou de máquina, e como cada processo é cuidadosamente construído. Além disso, é importante para exemplificar como os autômatos finitos e determinísticos e as expressões regulares se fazem presentes na teoria (e na prática) da computação.

2 Introdução

Um autômato finito determinístico é um modelo de máquina ou um dispositivo reconhecedor de determinada linguagem[1] e é uma peça de extrema importância para o processo de análise léxica de um código. Uma linguagem, dentro do assunto de autômatos, significa a palavra ou o conjunto de palavras que este modelo reconhece. Ele é projetado para validar somente as palavras de interesse e rejeitar palavras irrelevantes, e pode ser usado para encontrar padrões dentro de um universo de palavras, como um texto, ou seja, fazer um pattern matching[2]. Dessa forma, um autômato é utilizado para fazer o reconhecimento de palavras-chave pertencentes à uma linguagem de programação, a qual possui um dicionário próprio de palavras reservadas ao dispor do programador.

Entretanto, o pattern matching é feito inteiramente pelo programa *Flex Windows (Lex and Yacc)*, sendo necessário por parte do aluno estabelecer as regras de reconhecimento das palavras da linguagem criada. Além disso, o analisador léxico implementado armazena os padrões em uma estrutura de dados dinâmica, a fim de facilitar o trabalho da próxima peça do compilador: o analisador sintático. Ele fará o uso de todas as informações para o seu próprio tipo de verificação. Por fim, além do reconhecimento de padrões e da implementação de uma forma de armazenamento, também foi pensada na identificação e no apontamento de erros do código, levando em conta as regras que a linguagem aceita.

A seguir, é descrito como o sistema foi desenvolvido.

---

[1] Rômulo César Silva, 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: 13/08/2024.

3 Funcionamento do Software

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* e a função principal “main()”, que executa as funções primordiais para armazenar as palavras (lexemas) em tabelas *hashing*[3], para abrir e ler o arquivo de código fonte desejado e 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[4] e os símbolos[5] da linguagem, uma para cada; possui funções de impressão, inicialização, inserção e busca em uma tabela.

O arquivo “lex.yy.c” é aquele que contém toda a lógica de programação por trás da implementação dos autômatos usados nas expressões regulares, e é gerado pelo sistema automaticamente após a compilação do arquivo de extensão *lex*.

Por fim, “gçç.exe” é o arquivo gerado após a compilação do arquivo .l e dos arquivos .c referentes ao programa. É por meio dele que a “compilação” do código fonte em linguagem Ç é feita.

Para isso, é preciso ter um arquivo texto com o código na mesma pasta que o analisador léxico e, em seguida, abrir um terminal ou um prompt de comando neste mesmo diretório. Depois, deve-se executar o arquivo “gçç.exe” seguido do nome do arquivo de código, incluindo sua extensão, por exemplo:

```
C:\Users\Documents .\gçç.exe codigo_fonte.txt
```

Em seguida, o código será compilado e, caso houver erros, serão exibidas mensagens de aviso apontando o tipo de erro, a linha no arquivo onde o erro se encontra e o lexema errado em questão.

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

4 A Linguagem Ç

Essa linguagem reconhece os seguintes padrões ou classes de tokens:

Identificador, Inteiro, Real, Caracter, Cadeia, If, Else, While, For, Do, Return, Int, Float, Char, Void, String, Comentário, Soma, Multiplicação, Divisão, Subtração, Atribuição, Struct, Enum, Define e Include.

Para cada uma delas, foi feita uma regra de expressão regular utilizando os operadores de concatenação, união, repetição, entre outros[6]. Por exemplo, para uma classe de Identificador, tem-se a seguinte regra:

```
LETRA               [a-zA-Z]
DIGITO              [0-9]
IDENTIFICADOR       (_|{LETRA})(_|{LETRA}|{DIGITO})*
```

Isso significa que, por exemplo, a regra "[a-zA-Z]" identifica todo os caracteres entre os intervalos de 'a' até 'z' e de 'A' até 'Z', levando em conta a tabela ASCII, e é nomeado como "LETRA".

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

struct lista {
    string _cadeia0 = "ola mundo!"
    float real35 = 3.09, y = 2.

}
```

O resultado da compilação deste código seria as palavras detectadas e armazenadas nas tabelas, invisível ao programador na linha de comando. As palavras como "#include" ou "BIBLIOTECA_H" são separadamente validadas de acordo com a regra que mais se adequa a elas, sendo que cada regra é posta cautelosamente pensando na ordem ou prioridade de abrangência.

Na imagem abaixo, são listadas todas as regras usadas no programa.

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

```
13  LETRA               [a-zA-Z]
14  DIGITO              [0-9]
15  IDENTIFICADOR       (_|{LETRA})(_|{LETRA}|{DIGITO})*
16  NATURAL_SEM_ZERO    [1-9]
17  NATURAL             (({NATURAL_SEM_ZERO}{DIGITO}*)|0)
18  INTEIRO             [-+]?{NATURAL}
19  REAL                {INTEIRO}([.,]{DIGITO}*)?
20  CARACTER            ('.')|''
21  CADEIA              \"[^\"\n]*\"
22
23  IF                  if
24  ELSE                else
25  WHILE               while
26  FOR                 for
27  DO                  do
28  RETURN              return
29
30  INT                 int
31  FLOAT               float
32  CHAR                char
33  VOID                void
34  STRING              string
35
36  COMENTARIO          "//"(.*)
37  SOMA                "+"
38  MULTIPLICACAO       "*"
39  DIVISAO             "/"
40  SUBTRACAO           "-"
41  ATRIBUICAO          "="
```

Imagem 1: Expressões regulares das classes de token. Fonte: autoria própria.

```
43  STRUCT              struct
44  ENUM                enum
45  DEFINE              #define
46  INCLUDE             #include
47
48  BRANCO              [ \t]+
49  EOL                 [\n\r]
50  DELIMITADORES       ("("|")")|("{"|"}")|("["|"]"|,)
51  ERRO1_1             [!#@$%&*,.;<>:^~0-9=|]+({IF}|{ELSE}|{WHILE}|{FOR}|{DO}|{RETURN}|{INT}|{FLOAT}|{CHAR}|{VOID}|{STRUCT}|{ENUM})
52  ERRO1_2             ({IF}|{ELSE}|{WHILE}|{FOR}|{DO}|{RETURN}|{INT}|{FLOAT}|{CHAR}|{VOID}|{STRUCT}|{ENUM})[!#@$%&*,.;<>:^~0-9=|]+
53  ERRO1_3             [!#@$%&*,.;<>:^~0-9=|]+({IF}|{ELSE}|{WHILE}|{FOR}|{DO}|{RETURN}|{INT}|{FLOAT}|{CHAR}|{VOID}|{STRUCT}|{ENUM})[!#@$%&*,.;<>:^~0-9=|]+
54  ERRO1_4_1           ([!#@$%&*,.;<>:^~0-9=|]+({DEFINE}|{INCLUDE}))
55  ERRO1_4_2           (({DEFINE}|{INCLUDE})[!#@$%&*,.;<>:^~0-9=|]+)
56  ERRO1_4_3           ([!#@$%&*,.;<>:^~0-9=|]+({DEFINE}|{INCLUDE})[!#@$%&*,.;<>:^~0-9=|]+)
57  ERRO1_4             {ERRO1_4_1}|{ERRO1_4_2}|{ERRO1_4_3}
58  ERRO1_PALAVRA       {ERRO1_1}|{ERRO1_2}|{ERRO1_3}|{ERRO1_4}
59
60  ERRO2_1             [!#@$%&*,.;<>:^~0-9]+{IDENTIFICADOR}
61  ERRO2_2             {IDENTIFICADOR}[!#@$%&*,.;<>:^~]+
62  ERRO2_3             [!#@$%&*,.;<>:^~0-9]+{IDENTIFICADOR}[!#@$%&*,.;<>:^~]+
63  ERRO2_ID            {ERRO2_1}|{ERRO2_2}|{ERRO2_3}
64
65  ERRO3_SOMA          ({SOMA}|{MULTIPLICACAO}|{DIVISAO}|{SUBTRACAO})+{SOMA}({SOMA}|{MULTIPLICACAO}|{DIVISAO}|{SUBTRACAO})*
66  ERRO3_MULTIPLICAO   ({SOMA}|{MULTIPLICACAO}|{DIVISAO}|{SUBTRACAO})+{MULTIPLICACAO}({SOMA}|{MULTIPLICACAO}|{DIVISAO}|{SUBTRACAO})*
67  ERRO3_DIVISAO       ({SOMA}|{MULTIPLICACAO}|{DIVISAO}|{SUBTRACAO})+{DIVISAO}({SOMA}|{MULTIPLICACAO}|{DIVISAO}|{SUBTRACAO})*
68  ERRO3_SUBTRACAO     ({SOMA}|{MULTIPLICACAO}|{DIVISAO}|{SUBTRACAO})+{SUBTRACAO}({SOMA}|{MULTIPLICACAO}|{DIVISAO}|{SUBTRACAO})*
69  ERRO3_OP            ({ERRO3_SOMA}|{ERRO3_DIVISAO}|{ERRO3_SUBTRACAO}|{ERRO3_MULTIPLICAO})
70
71  ERRO4_CHAR          [@$¨_¹²³£¢¬§`^:ªº~]+
```

Imagem 2: Expressões regulares das classes de token e de reconhecimento de erros. Fonte: autoria própria.

4.1 Tratamento de Erros

Além de classes de tokens, a linguagem também captura quatro tipos de erros comuns cometidos em etapas da construção léxica, são eles:

a) Palavras reservadas mal formadas: palavras-chave que estão mal escritas, contendo algum caractere desconhecido ou indevido, exceto aquelas que são identificadores. Como, ao invés de “if”, estar “if,” ou “9if”;
b) Identificadores mal formados: identificadores que não respeitam a formação correta, que é alguma letra ou *underline* (_) seguido de várias letras, underlines ou dígitos. Por exemplo, ao invés de “identificador1” estar “1identificador”;
c) Operadores mal formados: caracteres reservados para a operação aritmética estarem mal escritos ou duplicados. Estar “**” ao invés de apenas “*” ou então “+-” ao invés de somente um operador;
d) Caracteres inválidos: o programador digita um caractere não reconhecido pela linguagem, como “@” ou “~”.

Todas as regras para o tratamento de erros estão contidas na imagem 2.

5 Métodos

Os principais métodos usados na análise léxica vêm da biblioteca criada “hash_table.h”, que contém as funções de lógica por trás do tratamento das tabelas.

O software começa inicializando a tabela de símbolos e a de palavras reservadas (imagens 3 e 4), atribuindo todos os campos da estrutura como NULL

```
15  typedef struct noS{
16      int id;
17      char *lexema;
18      Categoria categoria;
19  } Simbolo;
20
21  // Tabela Hash de Simbolos (identificadores, const, num)
22  // Ex: x, fun1, Pessoa, 5.12
23  typedef struct simbolos {
24      Simbolo **tabela;
25  } TabelaSimbolo;
```

Imagem 3: Estrutura da tabela de símbolos. Fonte: autoria própria.

```
28  typedef struct noR{
29      int id;
30      char *lexema;
31  } Reservada;
32
33  // Tabela Hash de Palavras Reservadas
34  // Ex: if, else, for, ;, {, }...
35  typedef struct reservadas {
36      Reservada **tabela;
37  } TabelaReservada;
```

Imagem 4: Estrutura da tabela de palavras reservadas. Fonte: autoria própria.

Depois disso, cada lexema reconhecido pelas expressões regulares são inseridos através da função de inserção, que primeiro calcula a posição onde deveria armazenar este valor (a entrada do cálculo Hashing é feito com a soma dos valores em decimal de cada caractere presente do lexema), faz uma busca pela tabela para verificar se o valor já não está presente e então o insere. Para o caso de símbolos, sua categoria também é inserida, podendo ser Identificador, Inteiro, Real, Caracter ou String.

Depois de todas as inserções, ainda é possível chamar a função de impressão do conteúdo das duas tabelas.

5.1 Processo de Construção

Para desenvolver o software, foi preciso instalar o programa auxiliar *Flex Windows (Lex and Yacc)* no computador, pois assim, pôde-se criar um arquivo *lex* contendo as regras da linguagem. Esse primeiro arquivo foi feito após toda a criação da biblioteca “hash_table.h” e sua implementação “hash_table.c”.

Depois dos programas prontos, foram compilados através dos comandos:

```
lex analise_lexica.l
gcc *.c -o gçç
```

Em seguida, dois arquivos texto foram criados, um contendo o código fonte sem nenhum erro léxico e o outro contendo erros. Os dois foram testados através da execução do programa gerado na compilação acima, por meio do comando:

```
.\gçç.exe errado.txt
```

Todos os programas foram feitos no ambiente de desenvolvimento integrado Visual Studio Code.

Além disso, a ferramenta OnlineGDB[7] também foi utilizada para testar exemplos de palavras reconhecidas ou não pela linguagem C (na qual o projeto foi baseado) devido a sua praticidade. Também são dados créditos aos materiais de auxílio para a implementação das estruturas de dados e das expressões regulares, presentes nas referências bibliográficas.

6 Conclusão

O desenvolvimento deste analisador léxico para a linguagem Ç permitiu uma compreensão mais profunda dos mecanismos fundamentais por trás da tradução de linguagens de alto nível em código de máquina. Por meio da implementação de autômatos finitos determinísticos e do uso de expressões regulares, foi possível construir uma ferramenta capaz de identificar e processar tokens específicos, além de reconhecer e tratar erros léxicos com precisão. O projeto, além de consolidar o conhecimento sobre análise léxica e técnicas de compilação, evidenciou a importância de uma base teórica sólida para a aplicação prática em sistemas computacionais. A experiência de utilizar ferramentas como Flex Windows e Visual Studio Code, bem como a integração com estruturas de dados dinâmicas, foi essencial para o sucesso do projeto e proporcionou um aprendizado significativo sobre os processos envolvidos na construção de compiladores. Ao final, o resultado alcançado não apenas atende aos objetivos propostos, mas também serve como um alicerce para futuras explorações no campo da ciência da computação.

[7] ONLINEGDB. Disponível em: <https://www.onlinegdb.com/>. Acesso em: 13/08/2024.

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.