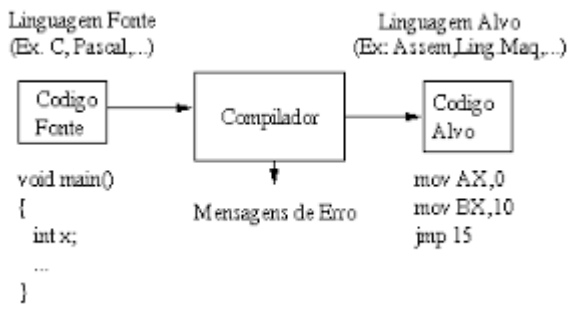
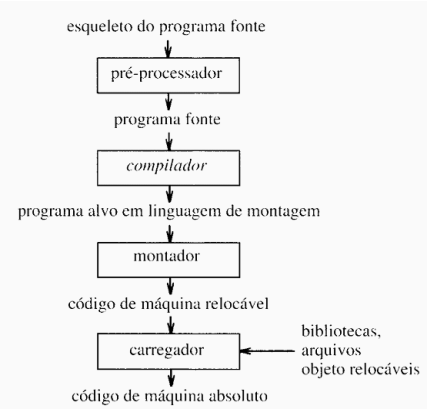
Guia para compiladores

Guilherme Dos Santos Modesto Teixeira

Introdução

Um compilador é um programa que traduz todo o código fonte de uma linguagem de programação de alto nível para código de máquina ou código intermediário antes de sua execução. Este processo gera um programa executável que pode ser executado independentemente do compilador. O compilador analisa, otimiza e traduz o código de uma só vez, e os erros são reportados antes da execução do programa.





Em primeira análise, vale ressaltar a diferença entre um interpretador e um compilador.

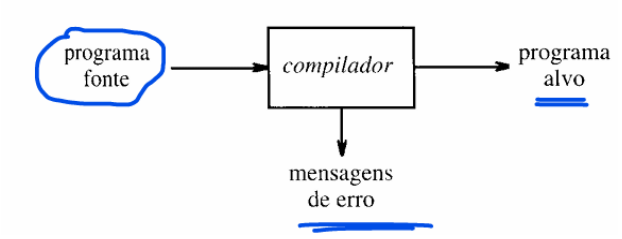
Um interpretador, por outro lado, traduz e executa o código fonte linha por linha ou declaração por declaração em tempo de execução. O código é executado diretamente pelo interpretador, não é criado um arquivo executável separado. Isso permite uma execução mais imediata e interativa, mas pode resultar em um desempenho mais lento comparado ao código compilado. Os erros são reportados enquanto o programa está sendo executado, o que pode facilitar a depuração em fases iniciais de desenvolvimento.

Etapas do compilador

No geral um compilador consiste nas etapas de análise e síntese, embarcadas no módulo do compilador.

A etapa de análise consiste em averiguar o código fonte, reconhecendo sua estrutura e o significado da linguagem de alto nível a ser traduzida. Nesse ponto verifica-se erros de digitação do código.

Na etapa de síntese ocorre a tradução do programa para assembly, código binário ou intermediário.



Um compilador é composto por duas partes principais: o front-end e o back-end. Cada uma dessas partes desempenha um papel distinto (análise e síntese) no processo de compilação.

- Front-End: Refere-se ao processo de análise léxica, sintática e semântica.
- Back-End: Refere-se ao processo de transformar o código gerado pelo Front-End em código executável.

Análise Léxica

A análise léxica (*scanner*) é o processo inicial da implementação de um compilador, responsável por ler (caracter a caracter) o código fonte como entrada e produzir uma sequência de **tokens** como saída.

O processo de análise léxica é fundamental, pois a estrutura de tokens serve como base para o analisador sintático (*parser*), que se concentra na estrutura gramatical do código sem se preocupar com detalhes de " baixo nível ", com eliminação e espaços em brancos.

Token é uma unidade básica (lógica) de uma linguagem que possui significado próprio (lexema), como palavras reservadas, identificadores, operadores e símbolos (+, -, >, <).

Exemplo

O código abaixo é escrito na linguagem de programação C.

```
#include <stdio.h>

int main()

{

   int result;

   result = 1;

   if (result != 0) {

      printf("Bad file handle\n");

   }

}
```

A sua documentação define os tokens da linguagem. De acordo com essa documentação, os tokens identificados no exemplo de código fornecido são:

Token	Tipo
#	Pré-processador
include	Palavra-chave
<stdio.h>	Identificador de arquivo de cabeçalho
int	Palavra-chave
main	Identificador
( ) ; { }	Símbolos
result	Identificador
=	Operador de atribuição
1	Constante numérica
if	Palavra-chave

!=	Operador relacional
0	Constante numérica
printf	Identificador
"Bad file handle\n"	Literal de string

Formalismo teórico da análise léxica

Os conceitos relacionados ao processo estão descritos abaixo:

- **Token**
- **Alfabeto (Σ)**
- **Cadeia ou palavra**
  - **Cadeia vazia (ε).**
  - **Comprimento de uma cadeia (n .**
  - **Potência de um alfabeto ($\sum^{k}$).**
  - **Fecho (Σ*).**
  - **Concatenação de cadeias.**
- **Linguagens Regulares**
  - **Expressões Regulares**
  - **Autômatos Finitos**
  - **Gramáticas Regulares**

Token

Como dito anteriormente, é uma unidade básica (lógica) de uma linguagem que possui significado próprio (lexema).

Alfabeto Σ

Conjunto de símbolos finitos.

Exemplos:

Alfabeto dos Símbolos Especiais

- Conjunto de símbolos: { !, @, #, $, %, ^, &, *, (,) }
- Representação: Σ = { !, @, #, $, %, ^, &, *, (,) }

Alfabeto Decimal

- Conjunto de símbolos: { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }
- Representação: Σ = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }

Alfabeto Binário

- Conjunto de símbolos: {0,1}
- Representação: Σ = { 0, 1 }

Cadeia

Sequência finita de símbolos de um alfabeto.

Exemplos:

- Σ = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }
- Cadeias:
    - 0123
    - 03456
    - 474674

Cadeia Vazia: Cadeia formada por zero símbolos e representada por ε

Comprimento de uma cadeia: Quantidade de símbolos (n) da cadeia de um alfabeto.

A cadeia 0123 tem 4 símbolos. n = 4

A cadeia vazia (ε) tem n = 0.

Potência de um alfabeto $\sum^{k}$

Potência é o conjunto de todas as cadeias de um alfabeto de tamanho K.

Exemplo:

Potência 2 $\sum^{2}$

Para o alfabeto: Σ = { !, @, #, $, %, ^, &, *, (,) } temos o conjunto de todas as cadeias de tamanho (k = 2).

{ !!, !@, !#, !$, !%, !^, !&, !, *!(, !), @!, @@, @#, @$, @%, @^, @&, @, @(, @)*, #!, #@, ##, #$, #%, #^, #&, #*, #), $(, $, $, #, %, &,),) }

Fecho

O fecho de Kleene de um alfabeto Σ é um conceito fundamental na teoria da computação e na linguagem formal. Ele representa o conjunto de todas as cadeias possíveis que podem ser formadas a partir dos símbolos de Σ, incluindo a cadeia vazia.

- $\sum^{0}$: Conjunto contendo apenas a cadeia vazia $\{\epsilon\}$.
- $\sum^{1}$: Conjunto de todas as cadeias de comprimento 1, que são exatamente os símbolos do alfabeto Σ.
- $\sum^{2}$: Conjunto de todas as cadeias de comprimento 2 formadas a partir dos símbolos de Σ.
- $\sum^{3}$: Conjunto de todas as cadeias de comprimento 3 formadas a partir dos símbolos de Σ.

Concatenação de cadeias

A concatenação de cadeias é uma operação fundamental.

Dadas duas cadeias x e y, onde x = x1x2….xn e y = y1y2….yn, a concatenação de x e y gera xy

Exemplo:

- x = abc
- y = def
- xy = abcdef
- Concatenação de x ou y com cadeia vazia é a própria cadeia (x ou y).

Linguagem Regular

Uma linguagem regular é um conjunto de cadeias (ou palavras) sobre um alfabeto que pode ser descrito por uma expressão regular, reconhecido por um autômato finito, ou gerado por uma gramática regular. As linguagens regulares são uma classe fundamental na teoria da computação e na teoria das linguagens formais.

Expressão Regular

Uma linguagem regular pode ser definida por uma expressão regular. Expressões regulares usam operadores como união (|), concatenação, e fecho de Kleene (*) para descrever conjuntos de cadeias.

Autômato Finito

Uma linguagem regular pode ser reconhecida por um autômato finito determinístico (DFA) ou não determinístico (NFA). Um autômato finito é uma máquina de estados que aceita ou rejeita cadeias de caracteres.

Gramática Regular

Uma linguagem regular pode ser gerada por uma gramática regular. Gramáticas regulares são um tipo especial de gramática formal onde as produções têm uma forma restrita.

Expressão Regular

Axiomas de uma expressão regular.

Concatenação

- Se r e s são expressões regulares, então rss é uma expressão regular que representa a concatenação de r e s.
- Exemplo: Se r = a e s = b, então rs = ab.

União (Alternância)

- Se r e s são expressões regulares, então r | s é uma expressão regular que representa a união das linguagens de r e s.
- Exemplo: Se r = a e s = b, então r | s = a | b (representa a linguagem {a, b}).

Fecho de Kleene

- Se r é uma expressão regular, então r^* é uma expressão regular que representa o fecho de Kleene de r, ou seja, zero ou mais ocorrências de r.
- Exemplo: Se r = a, então r^* representa a linguagem { ϵ, a, aa, aaa, ... }).

Exemplos

Exemplos de expressões regulares

$\Sigma = \{a, b\}$

1) UNIÃO

$(a|b) = \{a; b\}$

2) FECHO

$a^* = \{\varepsilon; a; aa; aaa \ldots\}$

Concatenações de a

Exemplos de expressões regulares

$\Sigma = \{a, b\}$

$a(a|b) = \{aa; ab\}$

Concatenação

a

b

$(a|b)^* = \{\varepsilon; a; b; aa;$
$aab; bb; ab; ba \ldots\}$

$a^* b = \{ b ; ab ; aab ;$
$aaab \ldots \}$

$a \mid \varepsilon = \{ \varepsilon , a \}$

$a \varepsilon = \{ a \}$

Exemplos de expressões regulares por uma linguagem regular

Desenvolva uma expressão regular sobre o alfabeto Σ = {a, b, c} que produza a linguagem L = {w | w possui *abc* como prefixo}.

Testar cadeias válidas para a linguagem L

$abc\,\varepsilon = abc$ ✓

$abca$ ✓ $abcc$ ✓

$abcaa$ ✓ $abcabc$ ✓

$abcb$ ✓

$abcbb$ ✓

$abc(a|b|c)^*$

$\varepsilon ; a ; b ; c$

$aa ; bb ; cc$

$ab ; ba ; ca \ldots$

Pode ser qualquer combinação de (a | b | c)*

Exemplos de expressões regulares por uma linguagem regular

Desenvolva uma expressão regular sobre o alfabeto Σ = {i, j, k} que produza a linguagem L = {w | w possui *kik* como sufixo}.

Testar cadeias válidas para a linguagem L

$kik\,\varepsilon = kik$ ✓

$ikik$ ✓ $ikkik$ ✓

$jkik$ ✓ $jikkik$ ✓

$kkik$ ✓ $kikkik$ ✓

$kijkik$ ✓

Qualquer ocorrência de (i | j | k)*

$(i|j|k)^* kik$

Exemplos de expressões regulares por uma linguagem regular

Conjunto de todas as palavras formadas pelo alfabeto { a,b }, tal que todas as palavras começam pelo caractere a e contêm ao menos um caractere b.

a^{*} ; ab ; abb ...

$aabb$; $a__b__ = a(a|b)^{*}b(a|b)^{*}$

aab

qualquer ocorrência de $(a|b)^{*}$

Subpalavra

Uma subpalavra (ou substring) é uma sequência contígua de caracteres que aparece dentro de uma cadeia maior. Por exemplo, na cadeia "abcde", "abc", "bcd", "cde", "ab", "bc", "cd", "de", "a", "b", "c", "d", "e" são todas subpalavras.

Exemplo para a linguagem L com Subpalavra xyx

Para desenvolver uma expressão regular que descreve a linguagem L sobre o alfabeto Σ={x,y,z} que contém todas as cadeias que possuem " xyx " como subpalavra, precisamos considerar todas as possíveis cadeias que podem ter "xyx" em qualquer posição. A expressão regular deve permitir qualquer sequência de caracteres antes e depois de " xyx ", ou seja, Σ* xyx Σ*.

Exemplos de expressões regulares por uma linguagem regular

Desenvolva uma expressão regular sobre o alfabeto Σ = {x, y, z} que produza a linguagem L = {w | w possui xyx como subpalavra}.

Testar cadeias válidas para a linguagem L

$\varepsilon xyx\varepsilon = xyx$ $\quad \varepsilon xyx\varepsilon$

$xxyxx$ $\quad yxyxy$

$xyxyxy$ $\quad (x|y|z)^{*}xyx(x|y|z)^{*}$

$xyzxyx\ xyz$

A expressão regular deve permitir qualquer sequência de caracteres antes e depois de "xyx".

[Link com mais exercícios resolvidos.]

Autômatos Finitos

Autômatos finitos são um modelo matemático usado para representar e analisar máquinas que têm um número finito de estados. Eles são amplamente utilizados na teoria da computação para descrever e reconhecer linguagens regulares. Existem três tipos principais de autômatos finitos: autômatos finitos determinísticos (DFA), autômatos finitos não determinísticos (NFA) e autômatos finitos com movimentos vazios (AFε) .

Autômato Finito Determinístico (DFA)

Um autômato finito determinístico é definido por uma quíntupla $(Q,\Sigma,\delta,q0,F)$ onde:

- Q: Um conjunto finito de estados.
- Σ: Um alfabeto finito de símbolos (conjunto de caracteres que o autômato pode ler).
- δ: A função de transição, $\delta:Q\times\Sigma\rightarrow Q$, que define a transição do autômato de um estado para outro com base no símbolo lido.
- q0: O estado inicial, $q0 \in Qq0$, onde o autômato começa a execução.
- F: Um conjunto de estados finais ou de aceitação, $F\subseteq QF$. O autômato aceita a cadeia se, após ler toda a cadeia, ele termina em um dos estados finais.

Propriedades do DFA

- Determinístico: Para cada par do estado e símbolo do alfabeto, há uma única transição possível.
- Exclusividade de Transições: Não há ambiguidade nas transições; um estado é um símbolo determinam um único próximo estado.

Exemplo

O DFA para a seguinte expressão regular: (a | b)(a | b).

Para ilustrar como um autômato finito pode reconhecer essa linguagem, vamos definir um DFA que aceita exatamente as cadeias abaixo:

aa, ab, ba, bb

Alfabeto: $\Sigma=\{a,b\}$

Conjunto de Estados: $Q=\{q0,q1,q2\}$

Estado Inicial: q0

Estado Final: $F=\{q2\}$

Funções de transição:

δ(q0,a)=q1

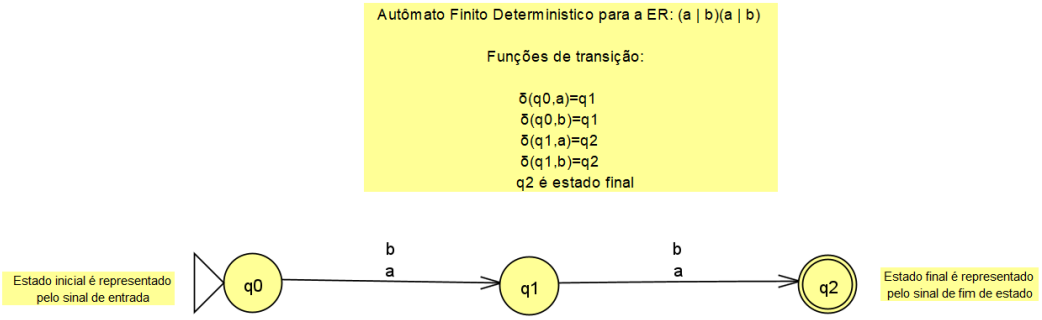
δ(q0,b)=q1

δ(q1,a)=q2

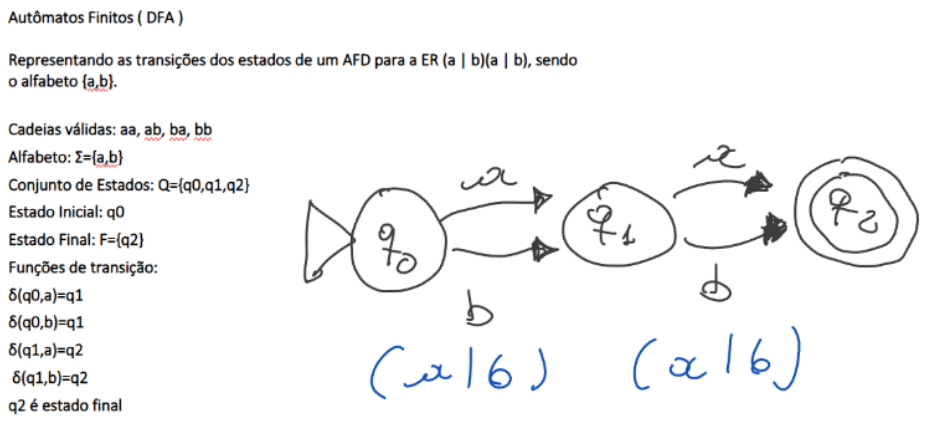
δ(q1,b)=q2

q2 é estado final

Representação no [software JFLAP](#).



Representação no papel



Autômato Finito Não Determinístico (NFA)

- Q: Um conjunto finito de estados.
- Σ: Um alfabeto finito de símbolos (conjunto de caracteres que o autômato pode ler).
- δ: A função de transição, $\delta: Q \times \Sigma \rightarrow 2^{Q}$, que define a transição do autômato de um estado para um subconjunto de estados, permitindo múltiplas transições para um símbolo lido.
- q0: O estado inicial, q0 ∈ Qq0, onde o autômato começa a execução.
- F: Um conjunto de estados finais ou de aceitação, F⊆QF. O autômato aceita a cadeia se, após ler toda a cadeia, ele termina em um dos estados finais.

Comparação entre DFA e NFA

- Transição no NFA é um conjunto de estados, diferente do DFA.

Exemplo

O NFA para a seguinte ER: (a | b)* (aa | bb)

Para ilustrar como um autômato finito pode reconhecer essa linguagem, vamos definir um NFA que aceita exatamente as cadeias abaixo:

aa, bb, aaa, abb, .. , qualquer ocorrência de (a | b)* no começo.

Alfabeto: Σ={a,b}

Conjunto de Estados: Q={q0,q1,q2,q3}

Estado Inicial: q0

Estado Final: F={q3}

Funções de transição:

δ(q0,a) = {q0,q1}
δ(q0,b) = {q0,q2}
δ(q1,a) = {q3}
δ(q2,b) = {q3}

Representação no JFLAP

O NFA para a seguinte ER: (a | b)* (aa | bb)

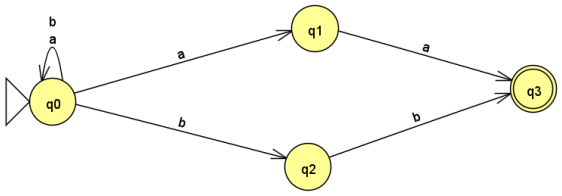
Alfabeto: Σ={a,b}

Conjunto de Estados: Q={q0,q1,q2,q3}

Estado Inicial: q0

Estado Final: F={q3}

Funções de transição:
δ(q0,a) = {q0,q1}
δ(q0,b) = {q0,q2}
δ(q1,a) = {q3}
δ(q2,b) = {q3}

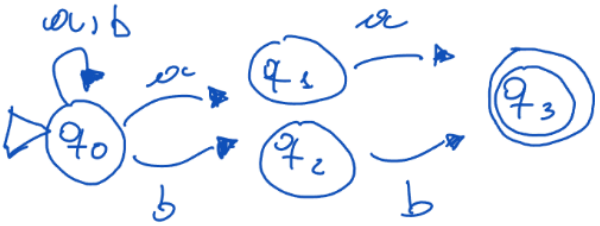


Representação no papel

Autômatos Finitos (DFA)

O NFA para a seguinte ER: (a | b)* (aa | bb).

Alfabeto: Σ={a,b}
Conjunto de Estados: Q={q0,q1,q2,q3}
Estado Inicial: q0
Estado Final: F={q3}
Funções de transição:
δ(q0,a) = {q0,q1} → Transição é um conjunto de estados
δ(q0,b) = {q0,q2}
δ(q1,a) = {q3}
δ(q2,b) = {q3}



[Link com mais exercícios resolvidos.]

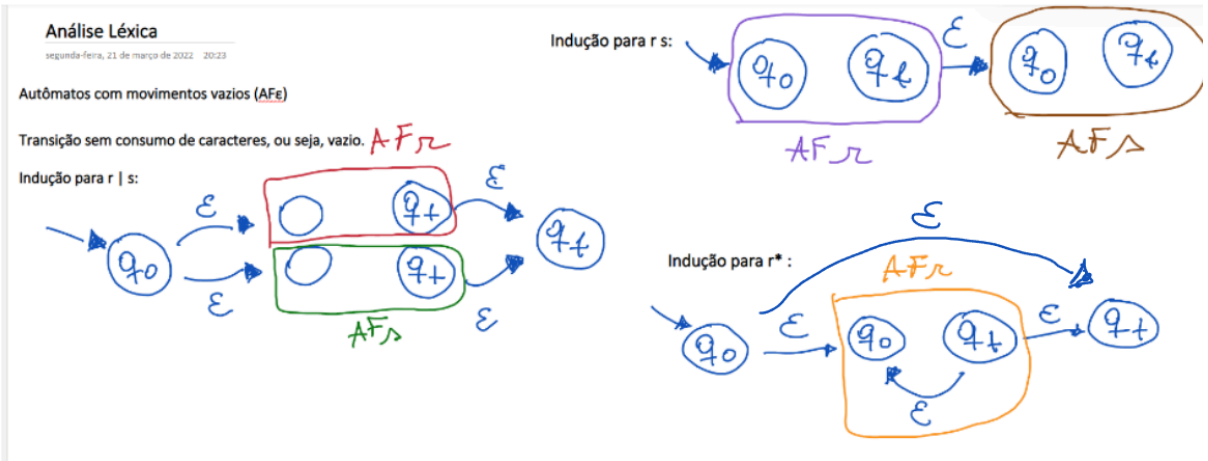
Autômato Finito com movimentos vazios (AFε)

Um Autômato Finito Não Determinístico com Movimentos Vazios (ε-NFA) é uma variação do autômato finito não determinístico (NFA) que permite transições que não consomem nenhum símbolo da cadeia de entrada. Essas transições são chamadas de movimentos vazios ou transições epsilon (ε).

O Algoritmo de Thompson é um método utilizado para converter uma expressão regular em um autômato finito não determinístico (NFA). Esse algoritmo é eficiente e fornece uma construção sistemática do NFA a partir de operações básicas de expressões regulares. A principal vantagem do Algoritmo de Thompson é que ele produz NFAs que são simples e possuem uma estrutura que facilita a conversão posterior para autômatos finitos determinísticos (DFAs).

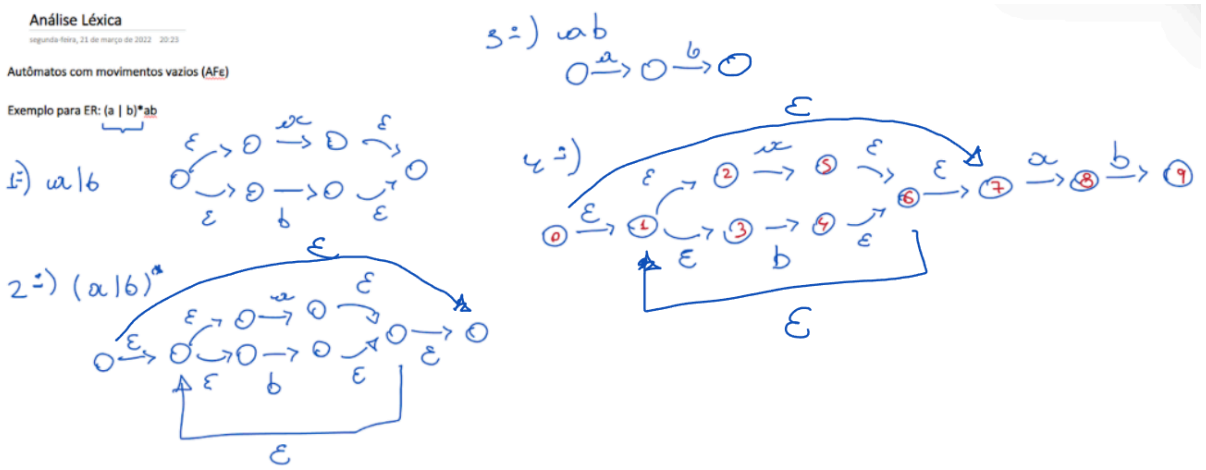
A indução no contexto do Algoritmo de Thompson refere-se à abordagem sistemática e recursiva utilizada para construir um autômato finito não determinístico (NFA) a partir de uma expressão regular.

Indução



Exemplo

Para ER (a|b)*ab.



Outro algoritmo usado é o de Subconjuntos, também conhecido como algoritmo de subset construction, é um método para converter um autômato finito não determinístico (NFA) em um autômato finito determinístico (DFA). Este processo é fundamental na teoria da computação, pois permite a utilização de DFAs, que são mais simples de implementar e têm um comportamento mais previsível que os NFAs.

O objetivo do Algoritmo dos Subconjuntos é transformar um NFA, que pode ter múltiplas transições para um estado dado e símbolo de entrada, em um DFA, que tem uma única transição para cada estado e símbolo de entrada.

Os conceitos de fecho-ε e movimentos são fundamentais para a teoria de autômatos finitos não determinísticos (NFA) e para a conversão de NFAs em autômatos finitos determinísticos (DFA) através do Algoritmo dos Sub-conjuntos. Vamos definir e explorar esses conceitos em detalhe.

Fecho-ε (q)

O fecho-ε de um estado q (ε-closure ou ε-fecho) é o conjunto de todos os estados que podem ser alcançados a partir de qqq utilizando transições ε (movimentos vazios). Uma transição ε permite que o autômato mude de estado sem consumir nenhum símbolo da cadeia de entrada. Formalmente, o fecho-ε de um estado q é definido como:

Fecho-ε(q)={p ∣ p é atingível a partir de q por uma sequência de transições ε}

Esse conjunto inclui o próprio estado q e todos os estados que podem ser alcançados a partir de q por qualquer número de transições ε.

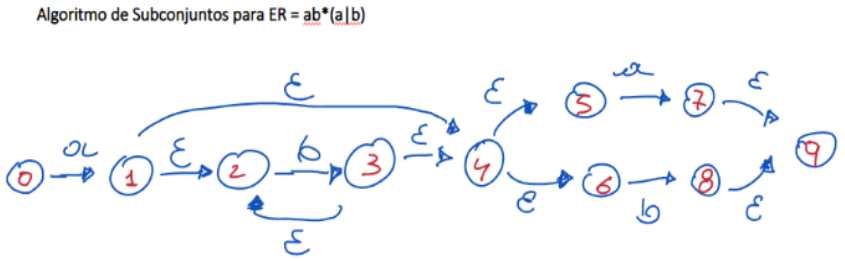
Fecho-ε (Q)

O fecho-ε de um conjunto de estados Q é o conjunto de todos os estados que podem ser alcançados a partir de qualquer estado em Q utilizando transições ε.

Movimento (Q, a)

O movimento de um conjunto de estados Q ao ler um símbolo a é o conjunto de todos os estados que podem ser alcançados a partir de qualquer estado em Q utilizando uma transição que consome o símbolo a.

Exemplo



fecho-ε(0) = {0} = A

fecho-ε(Mov(A, a)) = {1, 2, 4, 5, 6} = B
= 1

fecho-ε(Mov(A, b)) = { }
= { }

fecho-ε(Mov(B, a)) = {7, 9} = C
= {7}

ESTADO FINAL

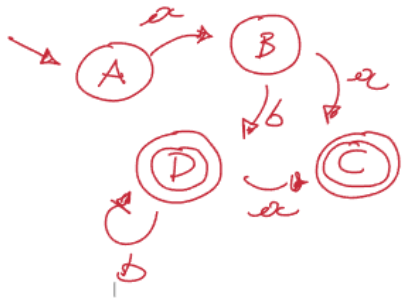
fecho-ε(Mov(B, b)) = {3, 4, 2, 5, 6, 8, 9} = D
= {3, 8}

fecho-ε(Mov(C, a)) = { }
= { }

fecho-ε(Mov(D, a)) = C
= {7}

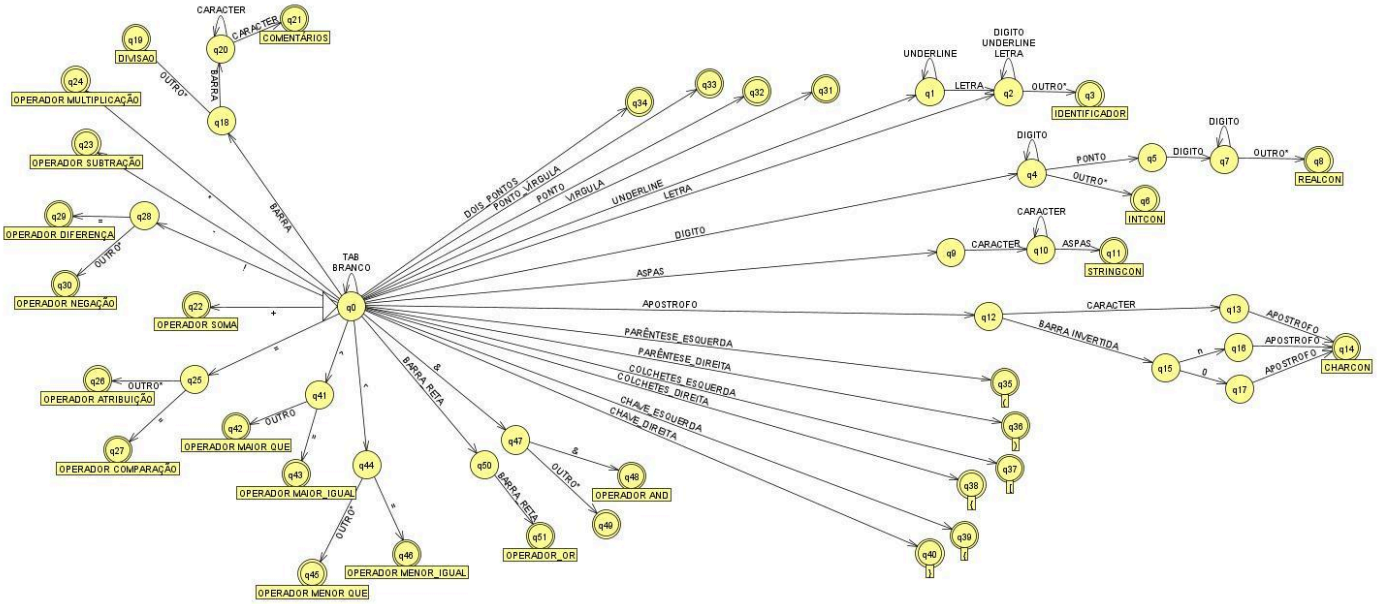
fecho-ε(Mov(D, b)) = D
= {3, 8}

Autômato Reduzido



[Link com exercícios]

Exemplo de autômato para implementação do módulo de um analisador léxico da linguagem DoBlock



Análise Sintática

A principal função do módulo de análise sintática é construir uma árvore de derivação ou uma árvore sintática abstrata a partir da sequência de tokens fornecida pelo analisador léxico. Essa árvore representa a estrutura gramatical do código fonte e é essencial para verificar a conformidade com a gramática da linguagem.

Gramática:

- Uma gramática formal é um conjunto de regras que descreve a estrutura de uma linguagem. É usada para definir quais sequências de símbolos formam sentenças válidas na linguagem. Em uma gramática, os símbolos são classificados em dois tipos principais: terminais e não-terminais.
    - Os símbolos terminais são os elementos básicos da linguagem que aparecem nas sentenças finais. Eles representam os "tokens" ou unidades mínimas que a linguagem pode reconhecer. Em uma gramática para uma linguagem de programação, os símbolos terminais geralmente correspondem a palavras-chave, identificadores, operadores e outros elementos que aparecem no código fonte.

      Exemplos de símbolos terminais: id, +, *, (,), números, palavras-chave como if, while, etc.

- Os símbolos não-terminais são utilizados para definir estruturas mais complexas e são usados nas regras da gramática para gerar sentenças. Eles representam categorias sintáticas que podem ser decompostas em outras categorias, sejam elas terminais ou não-terminais. Os não-terminais ajudam a descrever como os símbolos terminais podem ser combinados para formar construções válidas na linguagem.

  Exemplos de símbolos não-terminais: E (expressão), T (termo), F (fator), etc.

Regras de Produção

As regras de produção descrevem como os símbolos não-terminais podem ser substituídos por uma sequência de símbolos terminais e não-terminais.

Cada regra tem a forma: Não-terminal -> Sequência de símbolos

Árvore de Derivação:

- Uma árvore de derivação é uma representação hierárquica da aplicação das regras da gramática. O nó raiz representa o símbolo inicial da gramática, e os nós folha representam os tokens da entrada.

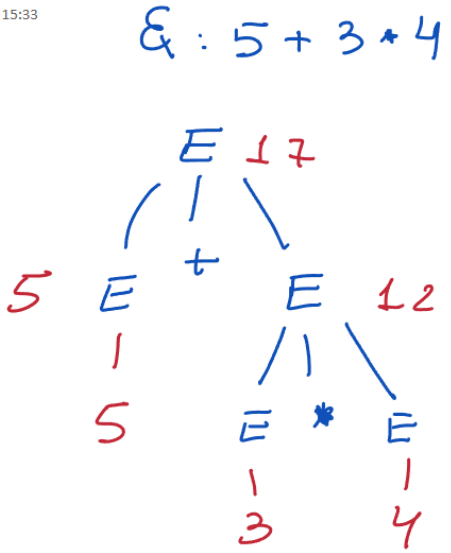
Exemplo

Considere a seguinte gramática para expressões aritméticas:

Gramática

sexta-feira, 2 de agosto de 2024 15:33

1. **E → E + E**
2. **E → E - E**
3. **E → E * E**
4. **E → E / E**
5. **E → 1 | 2 | 3 …**



Recursão à esquerda

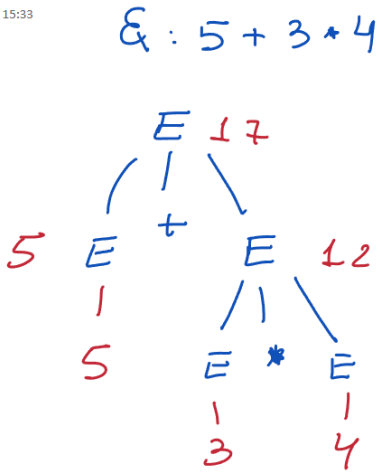
Gramáticas formais são fundamentais na definição da sintaxe de linguagens de programação. Elas especificam como os símbolos de uma linguagem podem ser combinados para formar sentenças válidas. No entanto, algumas gramáticas podem ser ambíguas, permitindo múltiplas derivações para a mesma sentença. Isso pode levar a interpretações diferentes, o que é indesejável em contextos como a compilação de programas.

Aqui, E representa uma expressão e os números 1, 2, 3, etc., são símbolos terminais. Esta gramática é ambígua porque uma mesma expressão pode ser derivada de múltiplas formas, resultando em diferentes árvores de derivação e, consequentemente, em diferentes interpretações.

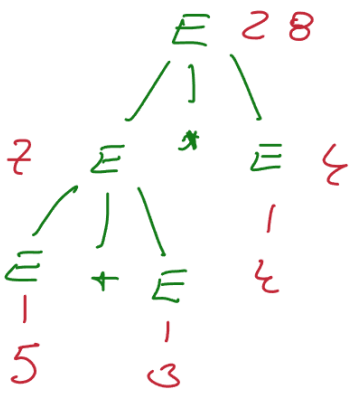
Gramática

sexta-feira, 2 de agosto de 2024 15:33

1. **E → E + E**
2. **E → E - E**
3. **E → E * E**
4. **E → E / E**
5. **E → 1 | 2 | 3 …**



Sintaticamente e Matematicamente correto

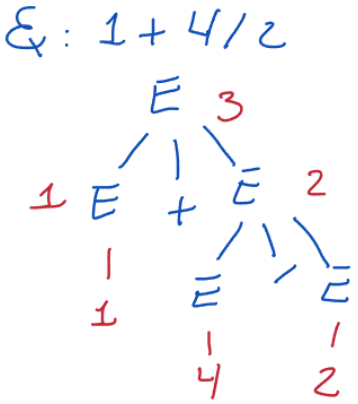


Sintaticamente correto, mas matematicamente incorreto

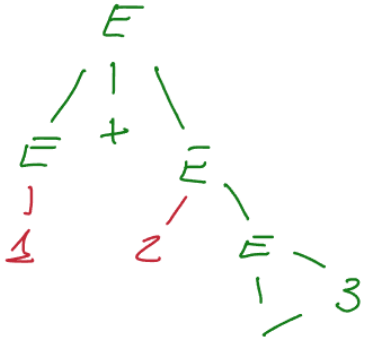
Gramática

sexta-feira, 2 de agosto de 2024 15:33

1. **E → E + E**
2. **E → E - E**
3. **E → E * E**
4. **E → E / E**
5. **E → 1 | 2 | 3 …**



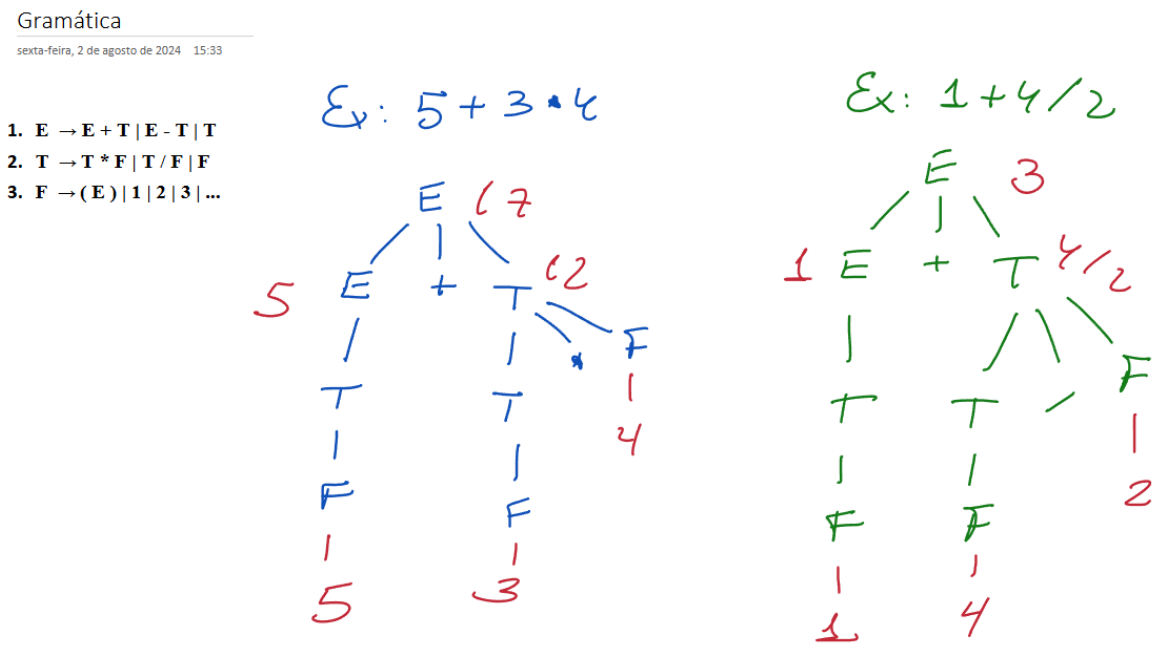
Sintaticamente e Matematicamente correto



Sintaticamente correto, mas matematicamente incorreto

Para resolver a ambiguidade, precisamos modificar a gramática para refletir a precedência correta dos operadores (multiplicação e divisão têm maior precedência do que adição e subtração) e associatividade.

E (expressão) lida com adição e subtração.
T (termo) lida com multiplicação e divisão.
F (fator) lida com números e expressões entre parênteses.



A nova gramática garante que a multiplicação e a divisão são avaliadas antes da adição e subtração, produzindo uma única árvore de derivação para qualquer expressão dada.

Recursão à esquerda

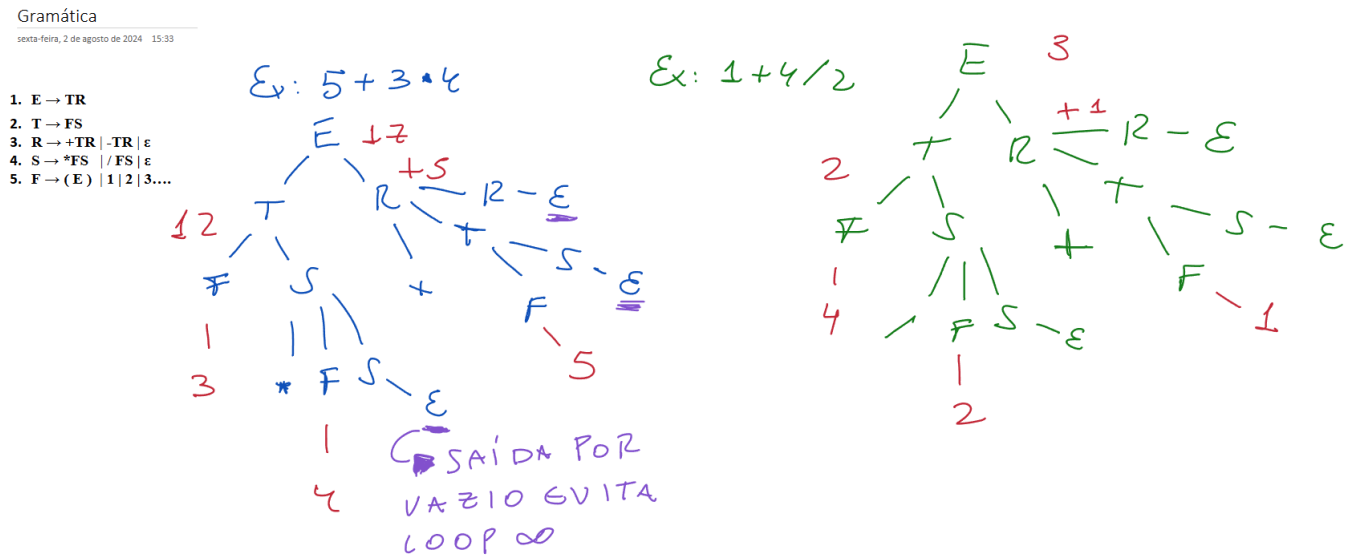
A definição da sintaxe é feita usando gramáticas formais. Uma gramática define as regras para construir expressões válidas na linguagem. No entanto, algumas gramáticas podem ter uma estrutura conhecida como recursão à esquerda.

A recursão à esquerda ocorre quando uma regra de produção de uma gramática pode se referir a si mesma no início da sua definição. Isso significa que a derivação de uma expressão pode começar com o mesmo símbolo não-terminal que está sendo definido, levando a loops infinitos em analisadores sintáticos.

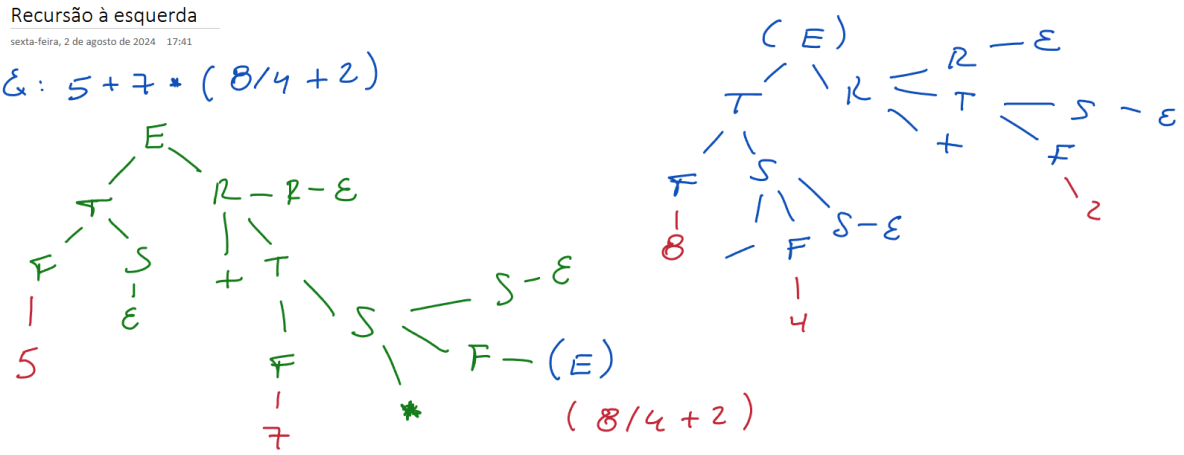
As regras **E → E + T** e **T → T * F** são exemplos de recursão à esquerda, porque **E** e **T** aparecem no início das suas próprias produções.

Para resolver a recursão à esquerda, reescrevemos a gramática para que as produções não comecem com o mesmo símbolo não-terminal. Em vez de permitir uma referência direta à esquerda, introduzimos novos símbolos não-terminais para representar as partes recursivas e não-recursivas da gramática.

1. **E → TR**
2. **T → FS**
3. **R → +TR | -TR | ε**
4. **S → *FS | / FS | ε**
5. **F → (E) | 1 | 2 | 3….**



Exemplo com (E)



Implementação prática de um compilador

O livro Compiladores: Princípios, Técnicas e Ferramentas retrata **TODO** o processo teórico e prático da implementação de um compilador.

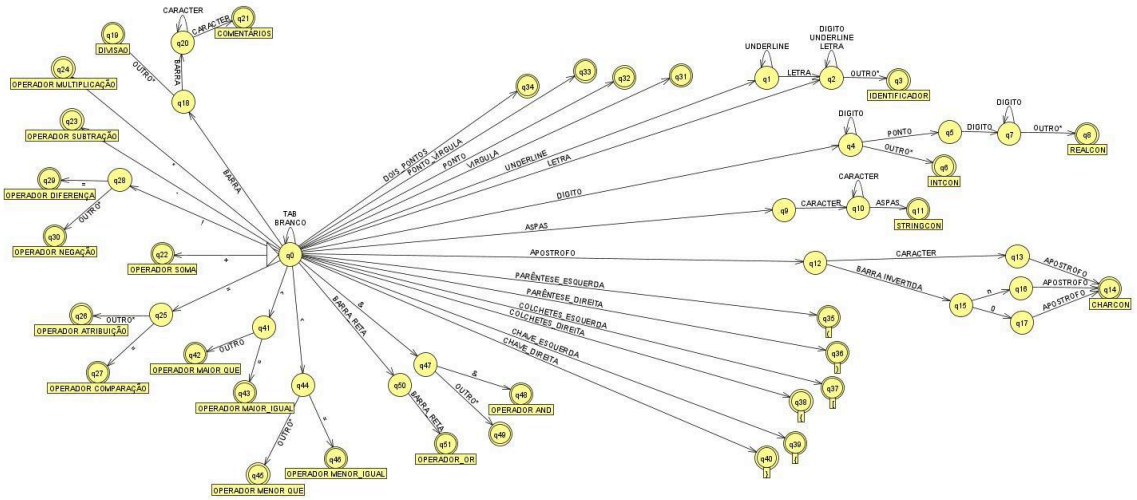
OBS:

- **Cada projeto de compilador tem suas especificações de produção, mas no geral o padrão das soluções estudadas na matéria estão descritas abaixo.**
- **Sugiro refatoração dos códigos e revisão das especificações do compilador alvo da matéria.**
- **Link para o projeto desse compilador**

Implementação prática da etapa de análise léxica

No geral o programador deve transcrever as transições dos estados para o código, ou seja, consiste em analisar os tokens da linguagem, criando uma estrutura de dados *typedef* para os respectivos tokens, uma função para manipular essa estrutura, para ler caracteres por caracteres, inserção de tokens na estrutura e outras funções de auxílio.

O primeiro passo para implementar um analisador léxico é fazer o autômato referente aos tokens da linguagem.



Estrutura do Token

```c
#define TAM_MAX_LEXEMA 31

enum TOKEN_CAT
{
    ID = 1,       // 1
    PAL_RESERV,   // 2
    SINAL,        // 3
    CONST_CHAR,   // 4
    CONST_FLOAT,  // 5
    CONST_INT,    // 6
    ID_CONST,     // 7
    LITERAL,      // 8
    OP_ARIT,      // 9
    OP_LOGIC,     // 10
    OP_RELAC,     // 11
    FIM_EXPR,     // 12
    FIM_PROG      // 13
};

enum PAL_RESERV
{
    MAIN = 1, // 1
    BLOCK,    // 2
    ENDBLOCK, // 3
    CONST,    // 4
    CHAR,     // 5
    INT,      // 6
    REAL,     // 7
    BOOL,     // 8
    WITH,     // 9
    DO,       // 10
    VARYING,  // 11
    FROM,     // 12
    TO,       // 13
    DOWNTO,   // 14
    WHILE,    // 15
    ENDWHILE, // 16
    FOR,      // 17
    IF,       // 18
    ELSEIF,   // 19
    ELSE,     // 20
    ENDIF,    // 21
    GOBACK,   // 22
    GETINT,   // 23
    GETCHAR,  // 24
    GETREAL,  // 25
    PUTINT,   // 26
    PUTREAL,  // 27
    PUTCHAR   // 28
}; // e etc..
```

```c
typedef struct
{
    enum TOKEN_CAT cat; // deve receber uma das constantes de enum TOKEN_CAT
    union
    {                                // parte variável do registro
        int codigo;                  // para tokens das categorias SINAL
        int indice;                  // indica o indice da tabela onde o Analex armazenou um literal(cadeira de caracter) para tokens da cat
        char lexema[TAM_MAX_LEXEMA]; // cadeia de caractares que corresponde o nome do token da cat. ID
        int valInt;                  // valor da constante inteira em tokens da cat. CONST_INT
        float valFloat;              // valor da constante flutuante em tokens da cat. CONST_FLOAT
        char caractere;              // caracter do token da cat. CONST_CHAR
    };
} TOKEN; // Tipo TOKEN

// Variáveis globais
extern TOKEN tk;
extern FILE *fd;
TOKEN AnaLex(FILE *);
```

- Esta estrutura representa um token e inclui um campo para a categoria do token (cat), e uma união para armazenar diferentes tipos de valores associados ao token, como código de sinal, índice de literal, lexema, valor inteiro, valor flutuante ou caractere.
- Declaração de variáveis globais: tk para armazenar o token atual e fd para o arquivo sendo analisado.
- Declaração do protótipo da função AnaLex, que será responsável por analisar lexicalmente um arquivo e retornar um token.

Definição de Constantes e Enumerações:

- TAM_MAX_LEXEMA define o tamanho máximo de um lexema (nome de token).
- enum TOKEN_CAT define categorias de tokens como identificadores, palavras reservadas, sinais, constantes de diferentes tipos (char, float, int), literais, operadores aritméticos, lógicos, relacionais, fim de expressão e fim de programa.
- enum PAL_RESERV define palavras reservadas específicas da linguagem, como MAIN, BLOCK, IF, WHILE, etc.
- enum SINAIS define sinais válidos na linguagem, como parênteses, chaves, colchetes e vírgula.
- enum OP_ARIT define operadores aritméticos como atribuição, adição, subtração, multiplicação e divisão.
- enum OP_RELAC define operadores relacionais como igualdade, diferença, menor ou igual, maior ou igual, menor que e maior que.
- enum OP_LOGIC define operadores lógicos como AND, OR e NOT.

Função AnaLex

A função principal do lexer, AnaLex, recebe um ponteiro para um arquivo (FILE *fd) e retorna um token (TOKEN). Dentro dessa função, várias variáveis são declaradas para armazenar o estado atual do analisador, tamanhos dos lexemas e dígitos, além de buffers para os próprios lexemas e dígitos.

estado: Variável que guarda o estado atual do autômato finito que implementa o analisador léxico.

tamL: Tamanho atual do lexema.

tamD: Tamanho atual dos dígitos numéricos.

lexema: Buffer para armazenar o lexema corrente.

digitos: Buffer para armazenar a sequência de dígitos numéricos corrente.

c: Caractere atual lido do arquivo.

token: Estrutura que representa o token reconhecido.

A função entra em um loop infinito que lê caracteres do arquivo e processa cada um conforme o estado atual do autômato finito:

Estado 0: Estado inicial.

- Se c for um espaço ou tabulação, permanece no estado 0.
- Se c for uma aspa simples ('), transita para o estado 9 e adiciona c ao lexema.
- Se c for uma aspa dupla ("), transita para o estado 15.
- Se c for uma barra (/), transita para o estado 18.
- Se c for um igual (=), transita para o estado 21.
- Se c for um sublinhado (_) ou uma letra (isalpha(c)), transita para o estado 1 e adiciona c ao lexema.
- Se c for um dígito (isdigit(c)), transita para o estado 4 e adiciona c ao digitos.

Os demais estados (1, 4, 9, 15, 18, 21, etc.) serão definidos para continuar o processamento específico de cada tipo de token, como identificadores, números, literais de caracteres, strings, operadores, comentários, etc. Cada estado tem a lógica específica para reconhecer o padrão de caracteres correspondente e formar os tokens.

Reconhecimento de palavras reservadas da linguagem

No contexto de um analisador léxico, este trecho de código é usado para identificar quando um lexema (uma sequência de caracteres identificada pelo analisador léxico) corresponde a uma palavra reservada da linguagem. As palavras reservadas são termos que têm um significado especial na linguagem e não podem ser usados como identificadores comuns (por exemplo, nomes de variáveis). Quando o analisador léxico identifica um lexema como uma palavra reservada, ele cria um token apropriado com uma categoria e um código específicos para uso nas fases subsequentes do compilador

```
case 1:
    if (c == '_' || isalpha(c) || isdigit(c))
    {
        estado = 1;
        lexema[tamL++] = c;
        lexema[tamL] = '\0';
    }
    else
    {
        estado = 3;

        ungetc(c, fd);

        strcpy(token.lexema, lexema);

        // verificacao de palavras reservadas
        if (strcmp("main", lexema) == 0)
        {
            token.cat = PAL_RESERV;
            token.codigo = MAIN;
        }
        else if (strcmp("block", lexema) == 0)
        {
            token.cat = PAL_RESERV;
            token.codigo = BLOCK;
        } // e etc..
```

O analisador léxico deve retornar os tokens presentes no arquivo txt.

```
// Programa exemplo em Do::Block - Alternativa 2
// Preencher uma matriz de inteiros com 1's acima
// da diagonal principal (inclusive)

block main

const int DIM_LIN = 10
const int DIM_COL = 10

int mat[DIM_LIN][DIM_COL]
int i, j
do
    do
   if (i >= j) m[l][c] = 1
   else m[l][c] = 0
   endif
varying j from 0 to DIM_COL-1
varying i from 0 to DIM_LIN-1

endblock
```

```
[========== INICIO - Analise lexica ==========]
<PR, block>
<PR, main>
<PR, const>
<PR, int>
<ID, DIM_LIN>
<OP_ARIT, ATRIBUICAO>
<CONST_INT, 10>
<PR, const>
<PR, int>
<ID, DIM_COL>
<OP_ARIT, ATRIBUICAO>
<CONST_INT, 10>
<PR, int>
<ID, mat>
<SN, ABRE_COL>
<ID, DIM_LIN>
<SN, FECHA_COL>
<SN, ABRE_COL>
<ID, DIM_COL>
<SN, FECHA_COL>
<PR, int>
<ID, i>
<SN, VIRGULA>
<ID, j>
<PR, do>
<PR, do>
<PR, if>
<SN, ABRE_PAR>
<ID, i>
<OP_RELAC, MAIOR_IGUAL>
<ID, j>
<SN, FECHA_PAR>
<ID, m>
<SN, ABRE_COL>
<ID, l>
<SN, FECHA_COL>
<SN, ABRE_COL>
<ID, c>
<SN, FECHA_COL>
<OP_ARIT, ATRIBUICAO>
<CONST_INT, 1>
<PR, else>
<ID, m>
<SN, ABRE_COL>
<ID, l>
<SN, FECHA_COL>
<SN, ABRE_COL>
<ID, c>
<SN, FECHA_COL>
<OP_ARIT, ATRIBUICAO>
<CONST_INT, 0>
<PR, endif>
<PR, varying>
<ID, j>
<PR, from>
<CONST_INT, 0>
<PR, to>
<ID, DIM_COL>
<OP_ARIT, SUBTRACAO>
<CONST_INT, 1>
<PR, varying>
<ID, i>
<PR, from>
<CONST_INT, 0>
<PR, to>
<ID, DIM_LIN>
<OP_ARIT, SUBTRACAO>
<CONST_INT, 1>
<PR, endblock>

FIM DO ARQUIVO

[========== FIM - Analise lexica ==========]
```

Implementação prática da etapa de análise sintática

A análise sintática descendente recursiva é uma técnica de análise sintática onde a estrutura gramatical do código é verificada através de um conjunto de funções recursivas. Cada função corresponde a uma produção da gramática da linguagem, e a análise é realizada descendo pela árvore sintática, verificando se a sequência de tokens segue as regras da gramática.

A análise sintática é uma fase fundamental no processo de compilação, responsável por validar a estrutura gramatical de um programa. Esta etapa segue a análise léxica, que transforma o código-fonte em tokens. O objetivo da análise sintática é garantir que a sequência de tokens siga as regras gramaticais da linguagem de programação.

Objetivo da Análise Sintática

O principal objetivo da análise sintática é verificar a conformidade do código com a gramática da linguagem. Isso envolve a identificação de estruturas de controle, declarações de variáveis, expressões e comandos, assegurando que estejam dispostos de maneira correta e coerente.

Técnicas de Análise Sintática

Existem várias técnicas e algoritmos para realizar a análise sintática. Os mais comuns são:

1. **Análise Ascendente (Bottom-Up Parsing)**: Constrói a árvore sintática a partir das folhas (tokens) para a raiz, aplicando reduções sucessivas.
2. **Análise Descendente (Top-Down Parsing)**: Constrói a árvore sintática a partir da raiz para as folhas, aplicando expansões sucessivas.

Estrutura de um Analisador Sintático

Um analisador sintático tipicamente possui funções que correspondem a regras gramaticais da linguagem. Cada função verifica a presença de estruturas específicas e consome tokens conforme necessário.

Função de Consumo

Uma função essencial é a de consumo de tokens (consome). Ela compara o token atual com o esperado e avança para o próximo token se corresponder, ou gera um erro sintático caso contrário. Isso garante que os tokens seguem a sequência esperada pela gramática.

Funções de Declaração

Funções como decl_list_var e decl_block_prot verificam declarações de variáveis e blocos, consumindo tokens que representam tipos de dados e identificadores. Essas funções também gerenciam o escopo das variáveis, crucial para a correta associação de identificadores.

Funções de Bloco Principal e Definição de Blocos

A função block_main verifica a estrutura do bloco principal do programa, que é um requisito para muitas linguagens de programação. Similarmente, block_def lida com a definição de blocos adicionais, assegurando que as declarações e comandos dentro deles seguem a gramática.

Funções de Expressão e Comando

Funções como expr, atrib e cmd são responsáveis por verificar expressões e comandos. Elas lidam com operadores aritméticos, relacionais e lógicos, bem como comandos condicionais e de loop.

Tratamento de Erros Sintáticos

O tratamento de erros é uma parte crucial do analisador sintático. Quando um token inesperado é encontrado, a função de erro sintático (errorSint) é chamada, geralmente fornecendo a linha e uma mensagem de erro detalhada. Isso ajuda na depuração do código-fonte, indicando precisamente onde a estrutura gramatical foi violada.

Tabela de símbolos

A tabela de símbolos é uma estrutura fundamental em compiladores e interpretadores de linguagens de programação. Ela é responsável por armazenar informações sobre identificadores (como variáveis, funções e outros símbolos) utilizados em um programa. Nesta implementação, a tabela de símbolos é gerenciada por uma série de funções e estruturas de dados. A seguir, apresentamos um detalhamento das principais funções e técnicas utilizadas.

Estrutura de Dados

A tabela de símbolos é representada pela estrutura TabSimb, que contém os seguintes campos:

- lexema: String que armazena o nome do identificador.
- tipo: Inteiro que indica o tipo do identificador (por exemplo, variável inteira, string, etc.).
- escopo: Inteiro que representa o escopo em que o identificador foi declarado.
- categoria: String que classifica o identificador (por exemplo, variável, função).
- zombie: Booleano que indica se o identificador está marcado como "zumbi".
    - Controlar se é parâmetro de função ou não.
- endereco: Inteiro que armazena o endereço do identificador na tabela.

A estrutura TabIdef é utilizada para gerenciar a tabela de símbolos. Ela contém:

- tamTab: O número atual de símbolos na tabela.
- tabela_simb: Um array de estruturas TabSimb que armazena os símbolos.

```c
// Definição da estrutura TabSimb
typedef struct
{
    char lexema[TAM_MAX_LEXEMA];
    int tipo;
    int escopo;
    char categoria[TAM_CATEGORIA];
    bool zombie;
    int endereco;
} TabSimb;

// Definição da estrutura TabIdef
typedef struct
{
    int tamTab;
    TabSimb tabela_simb[TAM_MAX_TAB];
} TabIdef;

// Declaração de variáveis externas
extern TabIdef tabela_idef;
extern TOKEN tk;
extern int escopoAtual;

// Declaração de funções
void Iniciar_tabela();
void Imprimir_tabela();
int Buscar_tabela(const char lexema[]);
int Buscar_escopo(char lexema[], int escopo);
int Insercao_tabela(char lexema[], int escopo, int tipo, char categoria[TAM_CATEGORIA], bool zombie);
int Remover_ultimo();

#endif
```

```c
// Inicializa a tabela de símbolos
void Iniciar_tabela() {
    tabela_idef.tamTab = 0;
}

// Imprime a tabela de símbolos
void Imprimir_tabela() {
    printf("\nTabela de Símbolos:\n");
    printf("-----------------------------------------------------------------------------------------\n");
    printf("| %-6s | %-10s | %-10s | %-6s | %-10s | %-6s | %-8s |\n",
           "Indice", "Lexema", "Tipo", "Escopo", "Categoria", "Zombie", "Endereco");
    printf("-----------------------------------------------------------------------------------------\n");

    for (int i = 0; i < tabela_idef.tamTab; i++) {
        printf("| %-6d | %-10s | %-10d | %-6d | %-10s | %-6d | %-8d |\n",
                i,
                tabela_idef.tabela_simb[i].lexema,
                tabela_idef.tabela_simb[i].tipo,
                tabela_idef.tabela_simb[i].escopo,
                tabela_idef.tabela_simb[i].categoria,
                tabela_idef.tabela_simb[i].zombie,
                tabela_idef.tabela_simb[i].endereco);
    }
    printf("-----------------------------------------------------------------------------------------\n");
}


// Busca um lexema na tabela de símbolos
int Buscar_tabela(const char lexema[]) {
    for (int i = 0; i < tabela_idef.tamTab; i++) {
        if (strcmp(tabela_idef.tabela_simb[i].lexema, lexema) == 0) {
            return i;  // Retorna o indice se encontrar o lexema
        }
    }
    return -1;  // Retorna -1 se não encontrar o lexema
}

// Busca um lexema em um escopo específico
int Buscar_escopo(char lexema[], int escopo) {
    for (int i = 0; i < tabela_idef.tamTab; i++) {
        if (strcmp(tabela_idef.tabela_simb[i].lexema, lexema) == 0 && tabela_idef.tabela_simb[i].escopo == escopo) {
            return i;
        }
    }
    return -1;
}

// Insere um novo símbolo na tabela de símbolos e testa se já foi declarado
int Insercao_tabela(char lexema[], int escopo, int tipo, char categoria[], bool zombie) {
    if (tabela_idef.tamTab >= TAM_MAX_TAB) {
        printf("[ERRO] Tabela de símbolos cheia.\n");
        return -1;
    }

    int pos = Buscar_escopo(lexema, escopo);
    if (pos != -1) {
        printf("[ERRO] Identificador '%s' já declarado no escopo atual.\n", lexema);
        return -1;
    }

    TabSimb novoSimbolo;
    strncpy(novoSimbolo.lexema, lexema, TAM_MAX_LEXEMA - 1);
    novoSimbolo.lexema[TAM_MAX_LEXEMA - 1] = '\0'; // Assegura que a string esteja terminada em null
    novoSimbolo.escopo = escopo;
    novoSimbolo.tipo = tipo;
    strncpy(novoSimbolo.categoria, categoria, TAM_CATEGORIA - 1);
    novoSimbolo.categoria[TAM_CATEGORIA - 1] = '\0'; // Assegura que a string esteja terminada em null
    novoSimbolo.zombie = zombie;
    novoSimbolo.endereco = tabela_idef.tamTab;

    tabela_idef.tabela_simb[tabela_idef.tamTab++] = novoSimbolo;
    return 0;
}

// Remove o último símbolo inserido na tabela de símbolos
int Remover_ultimo() {
    if (tabela_idef.tamTab > 0) {
        tabela_idef.tamTab--;
        return 0;
    } else {
        printf("[ERRO] Tabela de símbolos vazia.\n");
        return -1;
    }
}

```

Árvore Sintática

Uma árvore sintática é uma estrutura hierárquica que representa a estrutura de um programa conforme a gramática da linguagem de programação. Cada nó da árvore corresponde a uma construção sintática na linguagem, como uma expressão, um comando ou uma declaração. A árvore é gerada durante a análise sintática do código-fonte e é usada para verificar se o código está conforme as regras da linguagem e para realizar transformações e otimizações durante a compilação.

Essas funções são essenciais para depurar e visualizar a estrutura da árvore sintática durante o desenvolvimento do compilador. Elas ajudam a garantir que a árvore está sendo construída corretamente e que a estrutura do código-fonte está sendo representada de forma adequada. A capacidade de imprimir a árvore de forma hierárquica e formatada facilita a identificação de problemas e a compreensão da estrutura do código analisado.

```c
enum MOVIM
{
    AVANCA,
    MANTEM,
    RETROCEDE
};

// extern enum BOOL mostraArvore;
extern char TABS[200];

void errorLex(int contLinha, char caracter);
void errorSint(int contLinha, char caracter[]);
void PrintNodo(char info[], int movim);
void PrintNodoInt(int val, int movim);
```

[Código completo com árvore sintática e tabelada símbolos]

```c
void errorSint(int contLinha, char caracter[])
{
    printf("\nCaracter '%s' invalido na linha: %d\n", caracter, contLinha);
    exit(1);
}

void errorLex(int contLinha, char caracter)
{
    char c_str[2];
    c_str[0] = caracter;
    c_str[1] = '\0';
    printf("\nCaracter '%s' invalido na linha: %d\n", c_str, contLinha);
    exit(1);
}

void PrintNodo(char info[], int movim)
{

    if (movim == AVANCA)
    {
        printf("%s%s\n", TABS, info);
        strcat(TABS, "\t");
    }
    else if (movim == MANTEM)
    {
        printf("%s%s\n", TABS, info);
    }
    else if (movim == RETROCEDE)
    {
        TABS[strlen(TABS) - 1] = '\0';
    }
}

void PrintNodoInt(int val, int movim)
{

    if (movim == AVANCA)
    {
        printf("%s%d\n", TABS, val);
        strcat(TABS, "\t");
    }
    else if (movim == MANTEM)
    {
        printf("%s%d\n", TABS, val);
    }
    else if (movim == RETROCEDE)
    {
        TABS[strlen(TABS) - 1] = '\0';
    }
}
```

O analisador sintático deve retornar se a análise está correta, a tabela de símbolos e a árvore.

```
EXPRESS├âO OK

Tabela de S├¡mbolos:
-----------------------------------------------------------------------------------------
| ├ìndice | Lexema     | Tipo        | Escopo | Categoria | Zombie | Endere├°o |
-----------------------------------------------------------------------------------------
| 0       | DIM_LIN    | 1598900548  | 1      | var       | 0      | 0         |
| 1       | DIM_COL    | 1598900548  | 1      | var       | 0      | 1         |
| 2       | mat        | 7627117     | 1      | var       | 0      | 2         |
| 3       | i          | 105         | 1      | var       | 0      | 3         |
| 4       | j          | 106         | 1      | var       | 0      | 4         |
-----------------------------------------------------------------------------------------

[========== FIM - Analise sintatica ==========]
```

```
[========== INICIO - Analise sintatica ==========]
In├¡cio do Programa
        Declara├°├úo de Bloco de Prot├¦tipo
        In├¡cio do Bloco Main
                Declara├°├úo de Lista de Vari├íveis
                        Tipo
                        Declara├°├úo de Vari├ível
                Declara├°├úo de Lista de Vari├íveis
                        Tipo
                        Declara├°├úo de Vari├ível
                Declara├°├úo de Lista de Vari├íveis
                        Tipo
                        Declara├°├úo de Vari├ível
                Declara├°├úo de Lista de Vari├íveis
                        Tipo
                        Declara├°├úo de Vari├ível
                        Declara├°├úo de Vari├ível
                In├¡cio do Comando
                        Comando DO
                                In├¡cio do Comando
                                        Comando DO
                                                In├¡cio do Comando
                                                        Comando IF
                                                                Express├úo
                                                                        Express├úo Simples
                                                                                Termo
                                                                                        Fator
                                                                        Operador Relacional
                                                                        Express├úo Simples
                                                                                Termo
                                                                                        Fator
                                                                In├¡cio do Comando
                                                                        Atribui├°├úo
                                                                                Express├úo
                                                                                        Express├úo Simples
                                                                                                Termo
                                                                                                        Fator
                                                                                Express├úo
                                                                                        Express├úo Simples
                                                                                                Termo
                                                                                                        Fator
                                                                                Express├úo
                                                                                        Express├úo Simples
                                                                                                Termo
                                                                                                        Fator
                                                                Comando ELSE
                                                                        In├¡cio do Comando
                                                                                Atribui├°├úo
                                                                                        Express├úo
                                                                                                Express├úo Simples
                                                                                                        Termo
                                                                                                                Fator
                                                                                        Express├úo
                                                                                                Express├úo Simples
                                                                                                        Termo
                                                                                                                Fator
                                                                                        Express├úo
                                                                                                Express├úo Simples
                                                                                                        Termo
                                                                                                                Fator
                                                        Comando VARYING
                                                                Express├úo
                                                                        Express├úo Simples
                                                                                Termo
                                                                                        Fator
                                                                Express├úo
                                                                        Express├úo Simples
                                                                                Termo
                                                                                        Fator
                                                                                Operador Aritm├®tico
                                                                                Termo
                                                                                        Fator
                                        Comando VARYING
                                                Express├úo
                                                        Express├úo Simples
                                                                Termo
                                                                        Fator
                                                Express├úo
                                                        Express├úo Simples
                                                                Termo
                                                                        Fator
                                                                Operador Aritm├®tico
                                                                Termo
                                                                        Fator
EXPRESS├âO OK
```

Links importantes

[Teoria de compiladores]

[Compiladores para humanos]

[Compiladores para burros]

[Linguagem Compilada vs Interpretada | Qual é melhor?]

[Linguagens e autômatos]