Compiladores: Análise Léxica

Ciência da Computação
Universidade de Itaúna

Definição

- É a primeira fase de um compilador.
- É responsável pela leitura da entrada como um fluxo de caracteres convertendo-os num fluxo de *tokens* a ser analisado pelo analisador sintático (*parser*).
- Também chamados de *scanners*, entregam juntamente com cada *token* o seu *lexema* e os atributos associados.

Definição

- Os *tokens* usualmente são conhecidos pelo seu lexema (sequência de caracteres que compõe um único *token*) e atributos adicionais.
- Os *tokens* podem ser entregues ao *parser* como tuplas na forma <a, b,…, n > assim a entrada:
  a = b + 3
- poderia gerar as tuplas:
  <**id**, a> <=,> <**id**, b> <+,> <**num**, 3>
- note que alguns *tokens* não necessitam atributos adicionais.

Objetivos

- Remoção de espaços em branco e comentários.
- Identificação de constantes.
- Identificação de palavras-chave (construções da linguagem).
- Reconhecimento dos identificadores.
- Separação do *parser* da representação da entrada.

Remoção de Espaços em Branco e Comentários

- Espaços em branco (*whitespaces*) são os brancos, tabulações e quebras de linha.
- Comentários são textos inseridos no programa para melhorar seu entendimento.
- Se o analisador léxico "elimina" os espaços em branco e os comentários, isto é, não os envia para o *parser*, então:
  - o *parser* não precisa efetuar seu tratamento (o que representa alguma otimização) e
  - gramática não precisa considerar sequer sua existência (tal inclusão é bastante trabalhosa).

Identificação de Constantes

- Embora dígitos possam ser repassados separadamente para o *parser*, sabemos de antemão que dígitos agrupados tem significado distinto.
- Simplificando a gramática do *parser* o analisador léxico pode identificar e agrupar os dígitos encontrados como unidades autônomas entregando ao *parser* um *token* especial (p.e. **NUM** ou **VAL**) e seu valor.

Identificação de Palavras-Chave

- A grande maioria das linguagens utiliza cadeias fixas de caracteres como elementos de identificação de construções particulares da linguagem (comandos de decisão, seleção, repetição, pontuação etc.).
- Tais cadeias são as chamadas palavras-chave da linguagem e também devem ser identificadas pelo analisador léxico.

Reconhecimento de Identificadores

- Variáveis, vetores, estruturas, classes etc. são usualmente distinguidas entre si através de identificadores, ou seja, de nomes arbitrários designados pelo programador.
- Os identificadores usualmente possuem regras de formação e devem ser distinguidos das palavras-chave da linguagem (tal tarefa é bastante simplificada quando as palavras-chave são reservadas).

Separação do *Parser* da Entrada

- Como o analisador léxico efetua as seguintes operações:
  - leitura caracteres da entrada;
  - determinação dos lexemas e seus atributos;
  - entrega dos *tokens* (lexemas + atributos) para o *parser*.
- Então o *parser* não precisa conhecer detalhes sobre a entrada (origem, alfabeto, símbolos especiais, operadores complexos).

Tipos de *Tokens*

- As linguagens de programação usualmente distinguem certos tipos ou classes de *tokens*:
  - palavras-chave
  - operadores
  - identificadores
  - constantes
  - literais
  - símbolos de pontuação

Tokens, Padrões e Lexemas

- Um mesmo *token* pode ser produzido por várias cadeias de entrada.
- Tal conjunto de cadeias é descrito por uma regra denominada *padrão*, associada a tais *tokens*.
- O padrão reconhece as cadeias de tal conjunto, ou seja, reconhece os *lexemas* que pertencem ao padrão de um *token*.

Tokens, Padrões e Lexemas

- A declaração C seguinte:

  int k = 123;

- Possui várias subcadeias:
  - *int* é o lexema para um *token* tipo **palavra-reservada**.
  - = é o lexema para um *token* tipo **operador**.
  - *k* é o lexema para um *token* tipo **identificador**.
  - *123* é o lexema para um *token* do tipo número **literal** cujo atributo valor é 123.
  - ; é o lexema para um *token* tipo **pontuação**.

Padrões

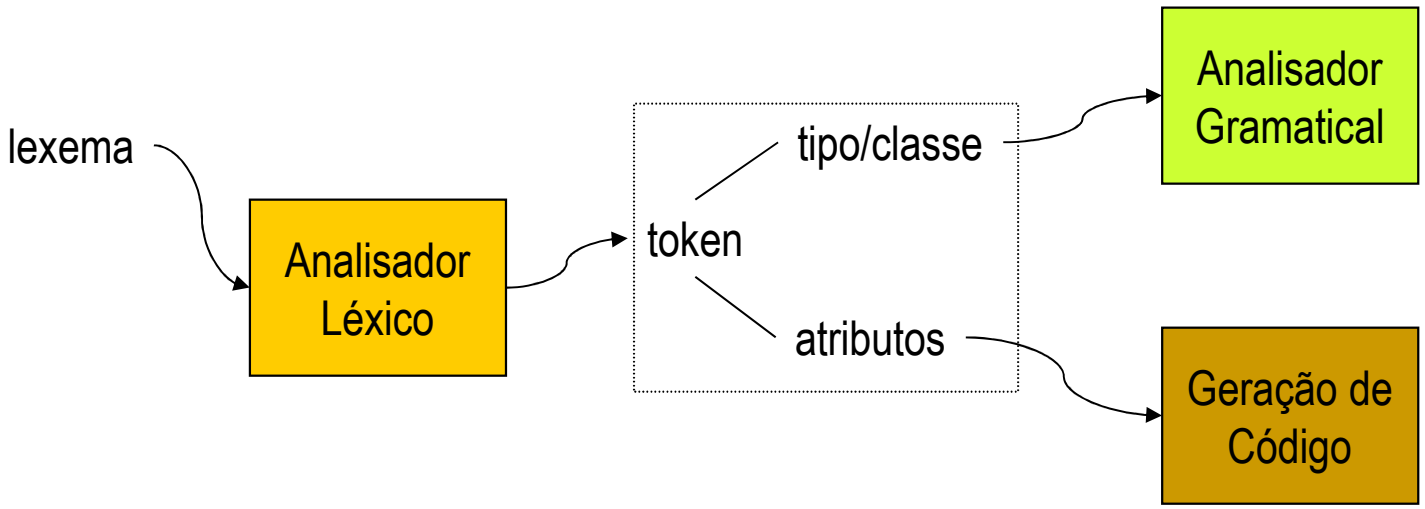
- São regras que descrevem o conjunto possível de lexemas que podem representar uma certa classe ou tipo de *token*.
- Em alguns casos o padrão é extremamente simples e restritivo:
  - o padrão para o *token* **float** é a própria cadeia "float", assim como **int**, **char**, **double** ou **void**.
- Em outros é um conjunto de valores:
  - o padrão para operadores relacionais em C é o conjunto {<, <=, >, >=, == e !=}

Padrões

- Usualmente os padrões são convenções determinadas pela linguagem para formação de classes de *tokens*:
  - **identificadores**: letra seguida por letras ou dígitos.
  - **literal**: cadeias de caracteres delimitadas por aspas.
  - **num**: qualquer constante numérica.

Atributos dos *Tokens*

- Quando um lexema é reconhecido por um padrão, a analisador léxico deve providenciar outras informações adicionais (os atributos) conforme o padrão.



Atributos dos *Tokens*

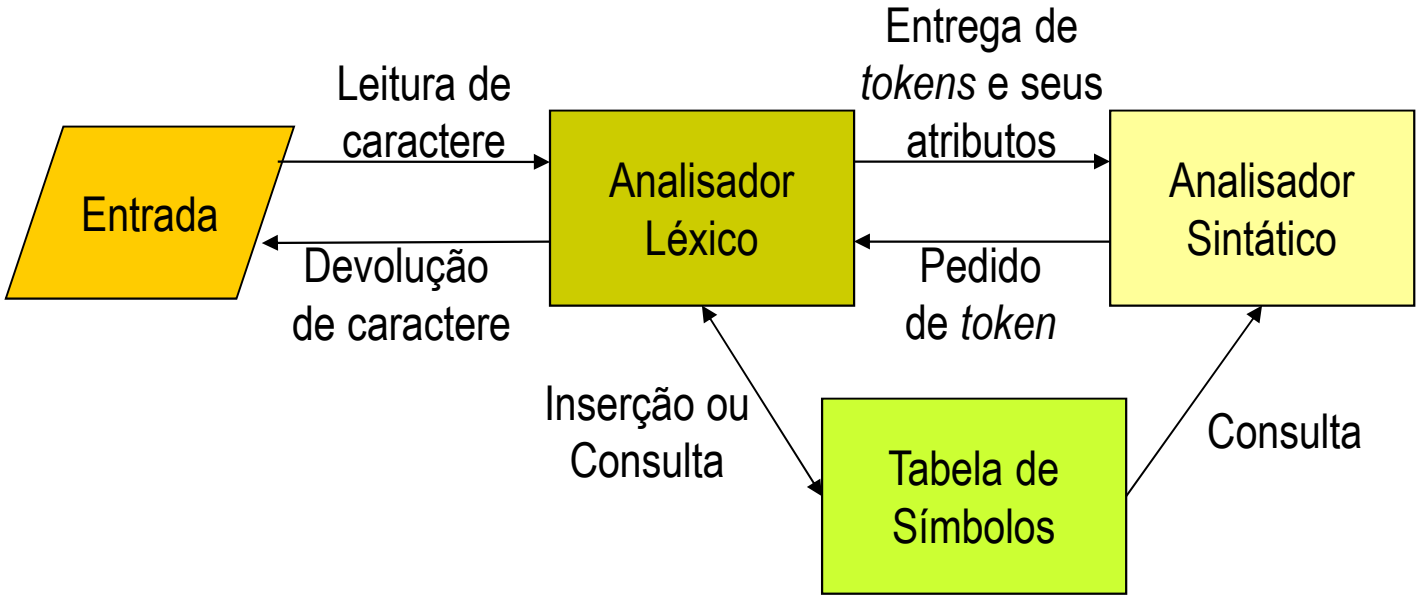
- Para a análise gramatical (sintática) basta conhecer-se o tipo do *token* (p.e. **NUM**).
- Já na fase de tradução (geração de código) o valor do *token* (seu atributo) é essencial.
- Na prática os *tokens* tem um único atributo que é um apontador para sua entrada na tabela de símbolos, onde as demais informações são armazenadas.

Estrutura Funcional

- Podem ser implementados num único passo (que efetua tanto a leitura da entrada como a separação dos *tokens* e seus atributos).
- Podem ser implementados em dois passos: um de varredura ou *scanning* (que efetua a leitura da entrada) e outro de análise léxica propriamente dita (que separa os *tokens* e seus atributos).

Interação do Analisador Léxico e o *Parser*

- Um analisador léxico interage com a entrada e com o *parser* (analisador sintático ou gramatical):



Divisão entre Análise Léxica e Análise Gramatical

Justifica-se tal divisão pois:

- Permite a simplificação do projeto de uma destas duas fases. Por exemplo:
  - É mais simples a extração de comentários e brancos pelo analisador léxico que sua incorporação na gramática.
- Leva a um projeto global de linguagem mais simples.

Divisão entre Análise Léxica e Análise Gramatical

- Permite tratar com maior eficiência as operações realizadas sobre a entrada através do uso de técnicas especializadas.
- Maior portabilidade pois peculiaridades do alfabeto de entrada, anomalias da entrada e representações especiais são isoladas na análise léxica.

Características do Funcionamento

- Em algumas situações o analisador léxico não consegue determinar o *token* apenas com o caractere corrente.
- Operadores relacionais do tipo <, <=, >, >=, != e == exigem a leitura de um caractere adicional.
- Quando tal caractere é o '=' então o par de caracteres lido é o lexema do *token*, caso contrário tal caractere deve ser "devolvido" a entrada pois pertence ao próximo *token*.

Características do Funcionamento

- O analisador léxico e o *parser* formam um par produtor-consumidor.
- Os *tokens* podem ser guardados num *buffer* até que sejam consumidos permitindo que tanto o analisador léxico quanto o *parser* pudessem ser executados em paralelo.
- Usualmente o *buffer* armazena apenas um *token* (simplificando a implementação) de forma que o analisador léxico opere apenas quando o *buffer* estiver vazio (operação por demanda).

Características de Funcionamento

- É muito conveniente que a entrada seja buferizada, i.e., que a leitura da entrada seja fisicamente efetuada em blocos enquanto que logicamente o analisador léxico consome apenas um caractere por vez.
- A buferização facilita os procedimentos de devolução de caracteres.

Erros Léxicos

- Poucos erros podem ser detectados durante a análise léxica dada a visão restrita desta fase, como mostra o exemplo:

  fi (a == f(x)) *outro_cmd*;

  - *fi* é palavra chave **if** grafada incorretamente?
  - *fi* é um identificador de função que não foi declarada faltando assim um separador (';') entre a chamada da função *fi* e o comando seguinte (*outro_cmd*)?

Erros Léxicos

- Ainda assim o analisador léxico pode não conseguir prosseguir dado que a cadeia encontrada não se enquadra em nenhum dos padrões conhecidos.
- Para permitir que o trabalho desta fase prossiga mesmo com a ocorrência de erros deve ser implementada uma estratégia de recuperação de erros.

Recuperação de Erros Léxicos

- Ações possíveis:

  (1) remoção de sucessivos caracteres até o reconhecimento de um *token* válido (modalidade pânico).

  (2) inserção de um caractere ausente.

  (3) substituição de um caractere incorreto por outro correto.

  (4) transposição de caracteres adjacentes.

Recuperação de Erros Léxicos

- Tais estratégias poderiam ser aplicadas dentro de um escopo limitado (denominado erros de distância mínima).

  … while (a<100) { fi (a==b) break else a++; } …

- Estas transformações seriam computadas na tentativa de obtenção de um programa sintaticamente correto (o que não significa um programa correto, daí o limitado número de compiladores experimentais que usam tais técnicas).

Enfoques de Implementação

- Existem 3 enfoques básicos para construção de um analisador léxico:
  - Utilizar um gerador automático de analisadores léxicos (tal como o compilador LEX, que gera um analisador a partir de uma especificação).
  - Escrever um analisador léxico usando um linguagem de programação convencional que disponha de certas facilidades de E/S.
  - Escrever um analisador léxico usando linguagem de montagem.

Enfoques de Implementação

- As alternativas de enfoque estão listadas em ordem crescente de complexidade e (infelizmente) de eficiência.
- A construção via geradores é particularmente adequada quando o problema não esbarrar em questões de eficiência e flexibilidade.
- A construção manual é uma alternativa atraente quando a linguagem a ser tratada não for por demais complexa.

Construção de um Analisador Léxico Simples

- Construiremos um analisador léxico que:
  - elimine espaços em branco e
  - reconheça números compostos de vários dígitos.
- Será utilizada a linguagem C implementando-se a função *lexan*().
- As funções padrão *getchar()* e *ungetch()* (<stdio.h>) serão responsáveis pelas operações *buferizadas* na entrada.

Construção de um Analisador Léxico Simples

- O código sugerido para um analisador léxico simples é dividido em três partes (arquivos):
  - Variáveis e constantes globais (global.h)
  - Analisador Léxico (lexan.c)
  - Programa de Teste (lexantest.c)

```
/***********
global.h
***********/

#define NONE   -1
#define NUM    256

int clinha;
int tokenval;
int lookahead;
```

Construção de um Analisador Léxico Simples

```
/***********
lexan.c
***********/
#include <stdio.h>
#include <ctype.h>
#include "global.h"
int clinha=1;
int tokenval=NONE;

int lexan() {
  int t;
  while(1) {
    t=getchar();
    if(t==' '||t=='\t') ; //
    else if(t=='\n')
        clinha++;
    else if(isdigit(t)) {
        ungetc(t, stdin);
        scanf("%d", &tokenval);
        return NUM;
      } else {
        tokenval=NONE;
        return t;
      }
  }
}
```

Construção de um Analisador Léxico Simples

```
/***********
lexantest.c
***********/
#include <stdio.h>
#include "global.h"

main() {
  lookahead=lexan();
  while(lookahead!=EOF) {
    if(lookahead==NUM) {
      printf("%d\n",tokenval);
    } else {
      printf("%c\n",lookahead);
    }
    lookahead=lexan();
  }
}
```

Construção de um Analisador Léxico Simples

- Observando o código podemos notar que:
  - A função *lexan* retorna os *tokens* como inteiros.
  - Caso o *token* sejam um número (**NUM**) a variável global *tokenval* contem seu valor.
  - A constante **NUM** é definida como 256 pois tal valor não corresponde a qualquer outro caractere válido (0 - 255 no ASCII).
  - A variável global *clinha* mantêm uma contagem do número de linha que está sendo correntemente processada.

Construção de um Analisador Léxico Simples

- Para compilar os arquivos separadamente:
  - gcc -c lexan.c
  - gcc -c lexantest.c
  - gcc lexan.o lexantest.o -o lexantest
- Para compilar todos os arquivos simultaneamente:
  - gcc lexan.c lexantest.c -o lexantest

Especificação de *Tokens*

- Os padrões de especificação de *tokens* são na verdade especificações de conjuntos de cadeias válidas.
- Tais conjuntos de cadeias podem ser mais precisamente expressos através da notação oferecida pelas *expressões regulares*.
- Para compreendermos melhor a notação das *expressões regulares* devemos primeiro apresentar alguns conceitos básicos.

Alfabetos

- *Alfabeto* ou *classe de caracteres* define um conjunto finito de símbolos. Exemplos comuns são:
  - letras do alfabeto ocidental: {a, b, c, d, … , z}
  - dígitos arábicos: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }
  - alfabeto binário: {0, 1}
- Um *alfabeto* pode mesclar diferentes tipos de símbolos tais como os alfabetos de computadores ASCII e EBCDIC.

Cadeias

- Uma *cadeia* é uma sequência finita de símbolos retirados de um dado *alfabeto*.
- O termo *palavra* pode ser usado como sinônimo para *cadeia*.
- Se uma cadeia s possui n símbolos, dizemos ser n o comprimento desta cadeia, que se denota por $|s|$.
- Uma cadeia s vazia (onde $|s|=0$) é denotada por ε.

Cadeias

- *Prefixo* de *s* : cadeia obtida com os *m* primeiros caracteres da cadeia *s* (*m* <= |*s*|).
- *Sufixo* de *s* : cadeia obtida com os *m* últimos caracteres da cadeia *s* (*m* <= |*s*|).
- *Subcadeia* de *s* : cadeia obtida com a remoção de um *prefixo* e um *sufixo* da cadeia *s*.
- *Subsequência* de *s* : cadeia obtida com a remoção de um ou mais símbolos de *s*.

Cadeias

- A *concatenação* de cadeias é uma importante operação onde "atrelamos" uma certa cadeia *x* a outra cadeia *y* formando uma nova cadeia denotada *xy* :

  x = lo, y = bo

  xy = lobo e yx = bolo

  s ε = ε s = s, pois ε é o elemento identidade (elemento neutro) da concatenação.

Cadeias

- A *exponenciação* de cadeias pode ser pensada como uma espécie de produto conforme a definição:

  $s^0 = \varepsilon$

  $s^i = s^{i-1}\, s$

- Como:

  $s^1 = s$

- Assim:

  $s^2 = ss$, $s^3 = sss$, $s^4 = ssss$, …

Gramáticas

- Para definir-se uma gramática é necessário:
  - Um conjunto de símbolos que constituirão as cadeias desta gramática, denominado alfabeto ou alfabeto terminal.
  - Um conjunto de regras que permita distinguir quais sentenças de cadeias pertencem a gramática e quais não, onde são necessárias:
    - regras de transformação das cadeias (produções)
    - um conjunto de símbolos não-terminais
    - um símbolo de partida

Gramáticas

- Portanto uma gramática é uma quádrupla:
  $G = \{\Sigma, V, P, S\}$
  onde:
  - Σ : alfabeto de símbolos *terminais*
  - V : conjunto de símbolos *não-terminais*
  - P: conjunto de *produções*
  - S: símbolo de *partida* ou *axioma*

Classificação de Chomsky

Gramática Tipo 3 (Regular)

- Quando todas as suas produções são da forma:
  - $A \rightarrow \in$ ou $A \rightarrow a$ ou $A \rightarrow aB$
    onde A e B são símbolos *não-terminais* e *a* é um *símbolo terminal*.
  - Exemplo I: números binários inteiros sem sinal
    $B \rightarrow 0 \mid 1 \mid 0B \mid 1B$
  - Exemplo II: identificadores típicos
    Identificador $\rightarrow$ aFim | .. | zFim
    Fim $\rightarrow$ aFim | .. | zFim | 0Fim | .. | 9Fim
    Fim $\rightarrow$ a | b | c | .. | z | 0 | 1 | 2 | .. | 9 | $\in$

Classificação de Chomsky

Gramática Tipo 2 (Livre de Contexto)

- Quando todas as produções tem a forma:
  - $A \rightarrow x$

    onde A é um *não-terminal* e x pertence a $V(V \cup \Sigma)^*$.
  - Exemplo: expressões aritméticas simples

    Expr → Termo | Expr + Termo | Expr - Termo

    Termo → Fator | Termo * Fator | Termo / Fator

    Fator → Identificador | Número
  - Geralmente as linguagens de programação são gramáticas tipo 2, ou seja, livres de contexto.

Classificação de Chomsky

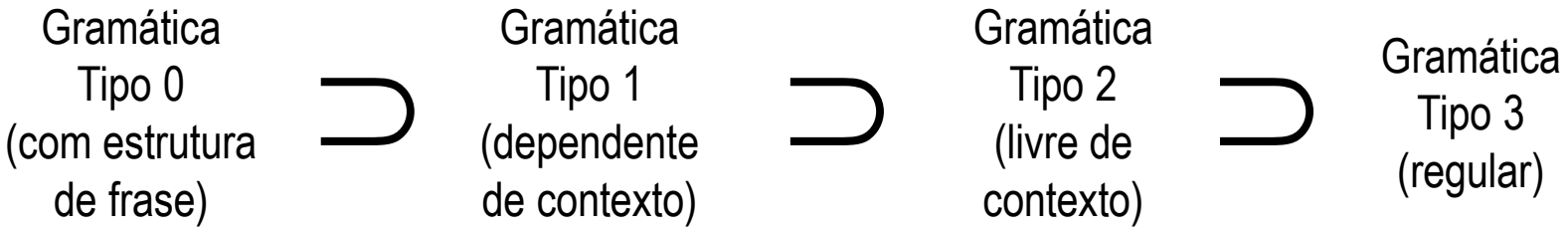
Gramática Tipo 1 (Dependente de Contexto)

- Quando todas as produções tem a forma:
  - $xAy \rightarrow xzy$
    onde A é um *não-terminal* e x e y pertence a $(V \cup \Sigma)^*$ e z pertence a $(V \cup \Sigma)^+$.
  - Significa que só podemos re-escrever A como z se A estiver dentro do contexto x e y.
  - Geralmente as linguagens naturais (idiomas) são gramáticas tipo 1, ou seja, dependentes de contexto e tendem a ser bastante complexas.

Classificação de Chomsky

Gramática Tipo 0 (Com Estrutura de Frase)

- É quando a linguagem é tão geral que não pode ser restrita a um conjunto de regras de produção.



Linguagens

- A definição de linguagem é muito ampla:

  "É um conjunto qualquer de cadeias sobre um alfabeto fixo"

- Assim sendo são linguagens:
  - $\varnothing$ (conjunto vazio)
  - $\{\in\}$ (conjunto com apenas a cadeia vazia)
  - Todos os programas sintaticamente corretos de uma linguagem de programação qualquer.
  - Todas as sentenças gramaticalmente corretas de um dado idioma.

Linguagens

- Sob certos aspectos as linguagens podem ser tratadas como conjuntos de cadeias.
- A definição de uma linguagem não inclui qualquer definição de significado as suas cadeias.
- A atribuição de significado às cadeias de uma linguagem requer métodos especiais que serão vistos durante o estudo da fase de análise sintática dos compiladores.

Operações sobre Linguagens

- Das várias operações que podem ser realizadas sobre linguagens destacamos:
  - União
  - Concatenação
  - Exponenciação
  - Fechamento Kleene ou Transitivo
  - Fechamento Positivo

Operações sobre Linguagens

- Dada duas linguagens L e M:
  - *União* de L e M:

    $L \cup M = \{ s \mid s \text{ está em } L \textbf{ ou } s \text{ está em } M \}$
  - *Concatenação* de L e M:

    $L\,M = \{ st \mid s \text{ está em } L \textbf{ e } t \text{ está em } M \}$
  - *Exponenciação* de L:

    $L^0 = \{\in\}$

    $L^i = L^{i-1}L$

    $L^1 = L,\ L^2 = LL, L^3 = LLL \ldots$

Operações sobre Linguagens

- *Fechamento Kleene ou Transitivo de L*:

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

denota **zero** ou mais concatenações de L.

- *Fechamento Positivo de L*:

$$L^+ = \bigcup_{i=1}^{\infty} L^i$$

denota **uma** ou mais concatenações de L.

Exemplos

- Seja *L* o conjunto de maiúsculas e minúsculas do alfabeto ocidental e D o conjunto de dígitos arábicos:
  L = {A, B, .., Z, a, b, .. z }
  D = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }
  - Como uma cadeia *s* pode ser formada por um único símbolo ($|s|=1$) então *L* e *D* podem ser consideradas como linguagens formadas pelas cadeias de um símbolo de tais alfabetos.
  - *L* e *D* são linguagens ditas finitas.

Exemplos

(1) $A = L \cup D$, conjunto de letras e dígitos.

(2) $B = LD$, conjunto de cadeias compostas de uma letra seguida por um dígito.

(3) $C = DL$, conjunto de cadeias compostas de um dígito seguido por uma letra.

(4) $E = L^4$, conjunto de cadeias compostas por quatro letras

(5) $F = LD^2$, conjunto de cadeias iniciadas por uma letra e seguida de dois dígitos.

(6) $G = (LD)^2 = LDLD$, conjunto de cadeias compostas de letra-dígito-letra-dígito.

Exemplos

(7) $H = L^*$, conjunto de todas as cadeias formadas por zero ($\in$) ou mais letras.

(8) $I = D^+$, conjunto de todas as cadeias formadas por um ou mais dígitos.

(9) $J = L(L \cup D)$, conjunto de cadeias compostas de letra-letra ou letra-dígito.

(10) $K = L(L \cup D)^*$, conjunto de cadeias iniciadas por letra e seguida de zero ou mais letras ou dígitos.

(11) $M = L(L \cup D)^+$, conjunto de cadeias iniciadas por letra e seguida de um ou mais letras ou dígitos.

Expressões Regulares

- Notação que permite exprimir precisamente o formato de um conjunto de cadeias.
- Se *r* é uma *expressão regular* então *L(r)* é a linguagem produzida por tal expressão.
- A linguagem formada por um conjunto de cadeias produzidas por uma expressão regular é dita *conjunto regular*.
- Uma *expressão regular* pode ser composta de *expressões regulares* mais simples.

Expressões Regulares

- letra (letra | dígito)*
  É uma *expressão regular* que especifica identificadores C, Java ou Pascal onde:
  - o operador '|' significa ou (união).
  - o operador '*' significa zero ou mais instâncias (fechamento Kleene ou Transitivo).
  - os parêntesis permitem especificar a sequência de avaliação da expressão.
  - a justaposição de letra com a expressão parentizada significa concatenação.

Expressões Regulares

- As regras que definem as *expressões regulares* sobre um alfabeto Σ são:

  (1) $\in$ é uma *expressão regular* que denota $\{\in\}$, ou seja, o conjunto que contêm a cadeia vazia.

  (2) sendo *a* um símbolo de Σ então *a* é uma *expressão regular* que denota $\{a\}$, isto é, o conjunto formado pela cadeia *a*. Rigorosamente um símbolo *a* é diferente de uma cadeia *a* que é diferente de uma *expressão regular a*.

Expressões Regulares

(3) sendo *r* e *s expressões regulares* que denotam as linguagens *L(r)* e *L(s)* são equivalentes:

a) $(r) \mid (s) \equiv L(r) \cup L(s)$

b) $(r)(s) \equiv L(r)L(s)$

c) $(r)^* \equiv (L(r))^*$

d) $(r) \equiv L(r)$

- A definição de *expressão regular* é recursiva (pois usa a si mesmo como parte da definição).

Expressões Regulares

- Dado o alfabeto: $\Sigma = \{ a, b \}$
- As *expressões regulares* sobre Σ produzem:

ab	{ab}
a|b	{a, b}
a(a|b)	{aa, ab}
(a|b)(a|b)	{aa, ab, ba, bb}
a^*	{∈ , a, aa, aaa, a…a}
$(a\|b)^*=(a^*b^*)^*$	{∈ , a, b, aa, ab, ba, bb, aaa, … }
$a\|a^*b$	{a, b, ab, aab, aaab, a…ab}

Expressões Regulares

- Para escrevermos *expressões regulares* devemos considerar a seguinte precedência de seus operadores:

  (0) Expressões parentizadas.

  (1) Operadores unários * e $^+$ (fechamento Kleene e positivo) que são associativos à esquerda.

  (2) Concatenação (justaposição).

  (3) Operador | (união) que também é associativo à esquerda.

Expressões Regulares

- Assim, segundo as convenções de precedência:
  (a)|((b)(c))* equivale a *a|b*c*
- Tal *expressão regular* produz o seguinte conjunto:
  {a, c, bc, bbc, bbbc, b…bc}
- Se duas expressões regulares denotam a mesma linguagem então dizemos que são equivalentes.

Propriedades Algébricas das Expressões Regulares

• r | s = s | r	| é comutativa
• r | (s | t) = (r | s) | t	| é associativa
• (rs)t = r(st)	concatenação é associativa
• r(s | t)=rs | rt (s | t)r=sr | tr	concatenação se distribui sobre |
• ∈ r=r r∈ =r	∈ é o elemento identidade da concatenação
• r*=(r | ∈)*	relação entre ∈ e *
• r**=r*	* é idempotente
• r*=r+ | ∈ r+=rr*	relação entre fechamentos positivo e Kleene

Definições Regulares

- Podemos dar nomes a *expressões regulares*.
- Tais nomes podem ser usados na construção de outras *expressões regulares*.
- Um conjunto de *expressões regulares* sobre Σ compõe uma *definição regular* :

$$d_1 \rightarrow r_4$$
$$d_2 \rightarrow r_2$$
$$:$$
$$d_n \rightarrow r_n$$

Exemplo de Definições Regulares

- Identificadores C, Pascal ou Java:

  letra → A | B | .. | Z | a | b | .. | z
  dígito → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
  identificador → letra(letra | dígito)*

- Números sem sinal em C, Pascal ou Java:

  dígito → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
  dígitos → dígito dígito*
  fração_opc → . dígitos | ∈
  expoente_opc → (E(+ | - | ∈)dígitos) | ∈
  num → dígitos fração_opc expoente_opc

Simplificações Notacionais

- Como algumas construções são freqüentes são introduzidas algumas simplificações:
  - Operador unário $^+$ denota uma ou mais ocorrências, daí se a *expressão regular r* denota uma linguagem *L(r)* então r^+ denota $(L(r))^+$.
  - Operador unário ? denota zero ou uma ocorrência assim: $r? = r \mid \in = L(r) \cup \{\in\}$.
  - Colchetes denotam classes de caracteres tais como: [A-Z] ou [A-Za-z] ou [0-9].

Simplificações Notacionais

- Assim poderíamos re-escrever as *definições regulares*:
  - Identificadores:
    [A-Za-z][A-Za-z0-9]*
  - Números sem sinal:
    dígitos → (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)+
    fração_opc → (. dígitos)?
    expoente_opc → ((E(+ | -)?dígitos)?
    num → dígitos fração_opc expoente_opc

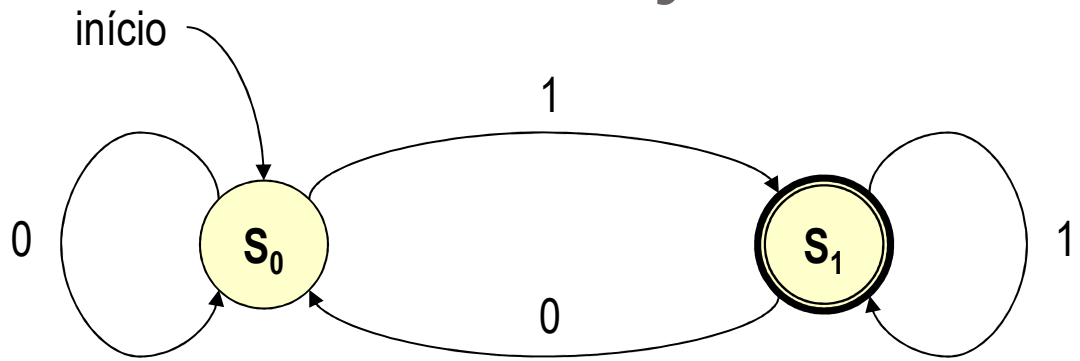
Reconhecimento de Tokens

- A especificação de *tokens* pode ser feita através de gramáticas ou de *expressões regulares*.
- O reconhecimento dos *tokens* é fundamental para um analisador léxico dado seu papel em obter os *tokens* da entrada bem como seus atributos.

Diagramas de Conway

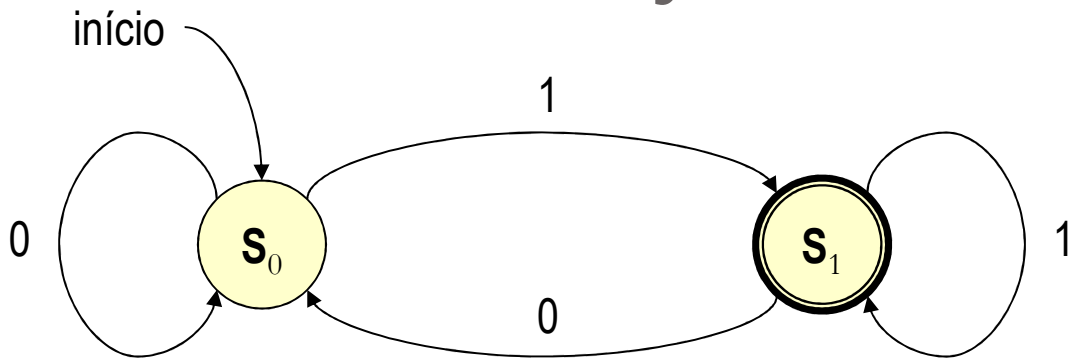
- Os diagramas de Conway ou de transição permitem especificar as ações de um analisador léxico tal como num fluxograma estilizado.
- Permitem controlar as informações a medida que a entrada é lida pelo analisador léxico.
- São determinísticos, ou seja, em qualquer ponto representado cada caminho é único.
- Diagramas de Conway = Autômatos Finito Determinísticos

Diagramas de Conway



- Os círculos representam *estados* distintos.
- Os arcos representam as *transições* ou *lados* entre estados provocadas pelo caractere indicado.
- Existe um *estado inicial* denotado pelo arco *início*.
- Existe um ou mais estados especiais denominados estados finais (círculo de borda dupla).

Diagramas de Conway



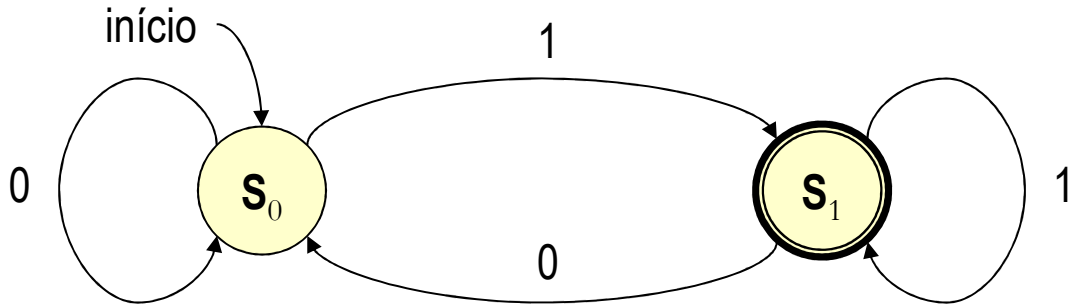
Dada as cadeias $c_1 = 0010$ e $c_2 = 1001$ temos:

- Para c_1:
  $S_0 \rightarrow 0 \rightarrow S_0 \rightarrow 0 \rightarrow S_0 \rightarrow 1 \rightarrow S_1 \rightarrow 0 \rightarrow S_0$
  Esta cadeia não "chega" num estado final.
- Para c_2:
  $S_0 \rightarrow 1 \rightarrow S_1 \rightarrow 0 \rightarrow S_0 \rightarrow 0 \rightarrow S_0 \rightarrow 1 \rightarrow S_1$
  Esta cadeia "chega" num estado final.

Autômatos de Estados Finitos

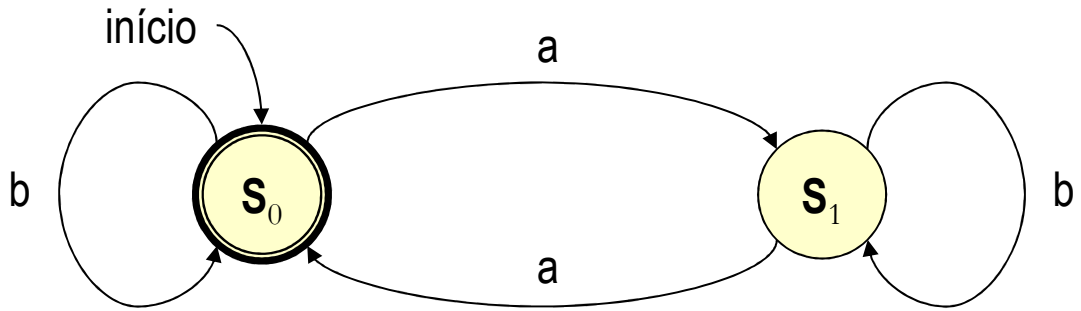
- Um diagrama de Conway é uma forma conveniente de representarmos uma máquina abstrata denominada "Autômatos de Estados Finitos" ou abreviadamente AF.
- Dizemos que um AF *aceita* ou *reconhece* uma certa cadeia *c* contendo n símbolos (n>=0) quando é conduzida a um *estado final* por meio das transições provocadas por todos os caracteres contidos na cadeia c.

Autômatos de Estados Finitos



- O AF acima descrito trata cadeias contendo apenas os símbolos 0 e 1, aceitando aquelas que:
  - sejam não vazias $|c|>0$ e
  - representem números binários ímpares (terminados pelo dígito 1).

Autômatos de Estados Finitos



- O outro AF acima trata cadeias contendo apenas os símbolos *a* e *b*, aceitando aquelas que:
  - sejam vazias |c|>= 0 ou
  - representem sequências contendo um número par de símbolos *a* (e um número qualquer de símbolos *b*).

Autômatos de Estados Finitos

- Definimos um AF como uma quíntupla ordenada:

  $AF = \{S, S_0, F, A, g\}$

  onde:

  S: conjunto de estados, onde $S \neq \emptyset$ e finito.

  S_0: estado inicial, sendo que $S_0 \in S$

  F: conjunto de estados finais, sendo $F \subset S$

  A: alfabeto de entrada (conj. finito de símbolos)

  g: aplicação de transição $S \times A \rightarrow S$

Autômatos de Estados Finitos

- O AF para números binários ímpares visto anteriormente e define então como:

  $(\{S_0,S_1\}, S_0, \{S_1\}, \{0,1\}, g)$

  sendo g:

  $g(S_0,0) \rightarrow S_0$

  $g(S_0,1) \rightarrow S_1$

  $g(S_1,0) \rightarrow S_0$

  $g(S_1,1) \rightarrow S_1$

Tipos de Autômatos de Estados Finitos

- Existem os AF Não-Determinísticos (AFN) e os AF Determinísticos (AFD):
  - Num AFN um mesmo símbolo pode rotular duas diferentes transições para fora de um mesmo estado, assim como $\in$ pode rotular uma transição.
  - Num AFD só pode existir uma transição rotulada com um dado símbolo para fora de um mesmo estado e não se aceitam transições rotuladas por $\in$.

Implementação de AFs

- Um AF Determinístico pode ser facilmente implementado utilizando-se comandos de seleção tipo *switch* (C ou Java).
- Também podem ser usados vetores mapeando a aplicação de transição.
- Um AFD genérico pode ser construído de forma que sua especificação seja obtida de um arquivo de definição.

Implementação de AFs

```
// Fonte Java
public class BasicAF {
public static void main(String a[]) {
    int p=0;
    int state=0;
    while (p<a.length()) {
     switch(state) {
       case 0:
        switch(a.charAt(p)) {
          case '0':
            state=0;  break;
          case '1':
            state=1;  break;
        }
        break;
       case 1:
        switch(a.charAt(p)) {
          case '0':
            state=0; break;
          case '1':
            state=1; break;
        }
        break;
     }
     p++;
    }
    if (state==1)
      System.out.println(”Aceita");
    else
      System.out.println("Nao
    aceita");
} }
```

Análise Léxica como AFs

- Considerando agora um conjunto de diagramas de transição temos que:
  - ordenar os estados de partida na ordem em que devem ser testados;
  - quando uma cadeia não é aceita, esta deve ser reaplicada considerando o próximo estado de partida;
  - quando uma cadeia é aceita retorna-se ao primeiro estado de partida para iniciar a procura de um novo padrão.

Análise Léxica como AFs

- para evitar redundância, vários diagramas podem ser re-escritos como um único mas tal tarefa geralmente não é simples;
- rotinas de tratamento de erros léxicos podem ser inseridas na implementação dos AFs para simplificar os diagramas de transição;
- situações esperadas com maior frequência devem ser posicionadas no início da sequência.

Análise Léxica como AFs

- Ainda na especificação de AFs podemos usar:
  - * para indicar uma retração da entrada (devolução de um caractere a cadeia de entrada).
  - **outro** para indicar qualquer outro caractere além dos especificados pelos demais lados.
  - chamadas de funções para indicar o processamento de ações especiais quando algum estado de aceitação é atingido.

Construção de AFs a partir de Gramáticas Regulares

- Dada as produções de uma gramática tal:

  $A \rightarrow x\ B$

  $B \rightarrow y \mid z$

  onde:

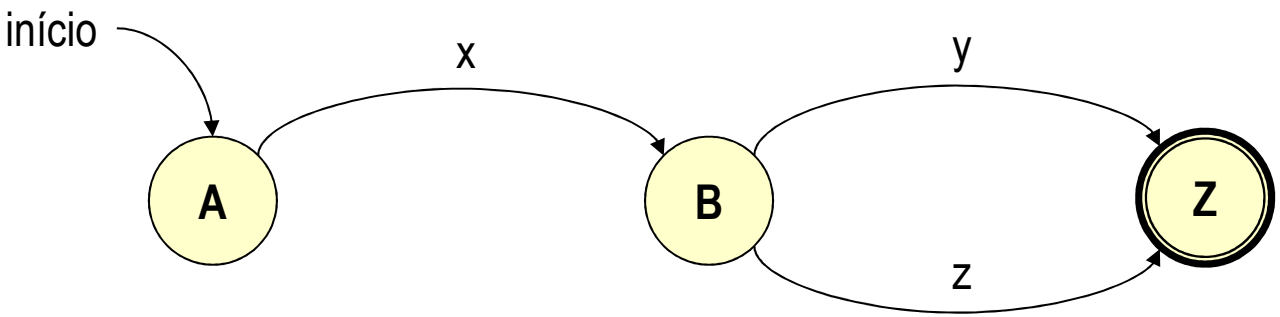
  - A e B são símbolos *não-terminais*
  - x, y, z são símbolos *terminais*
  - A é o símbolo de partida
- Tal gramática pode ser usada como especificação de um AF.

Construção de AFs a partir de Gramáticas Regulares

- A conversão pode ser feita do seguinte modo:
  - cria-se um estado distinto para cada símbolo *não-terminal*;
  - o estado inicial é aquele indicado pelo símbolo de partida;
  - um estado adicional correspondente ao estado final é criado para indicar-se conformidade com a gramática original;

Construção de AFs a partir de Gramáticas Regulares

- as transições são obtidas da seguinte maneira:
  - A → x B equivale a uma transição rotulada como x do estado A para o estado B.
  - A → x equivale a uma transição rotulada como x do estado A para o estado final.



Construção de AFs a partir de Expressões Regulares

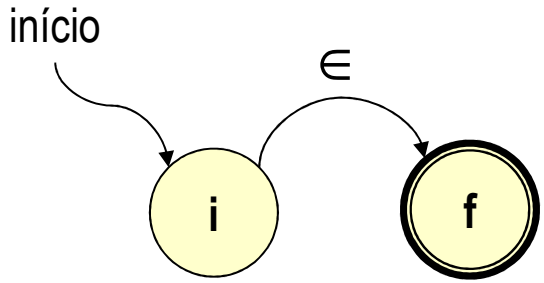
- Expressões regulares também podem ser utilizadas como especificação de AFs.
- Uma expressão regular pode ser transformada num AF através da técnica de construção de Thompson.
  Entrada: uma expressão regular r sobre o alfabeto Σ.
  Saída: um AF N que aceita $L(r)$ ou $N(r)$.

Construção de AFs a partir de Expressões Regulares

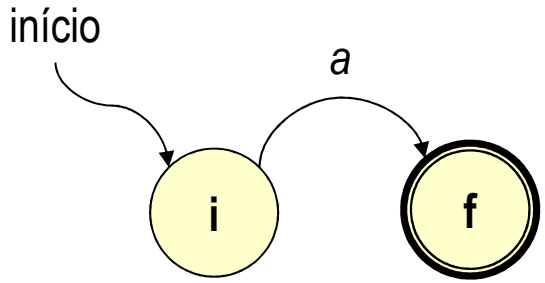
- Dada uma expressão regular *r*, esta é dividida em suas expressões constituintes.
- Para cada sub-expressão criamos um AF que possui propriedades especiais:
  - possui apenas um estado final
  - nenhum lado entra no estado inicial
  - nenhum lado deixa o estado final
- Os AFs são combinados indutivamente até obtermos um AF que corresponde a toda expressão *r*.

Construção de AFs a partir de Expressões Regulares

- Para $\in$ construímos o AFN:

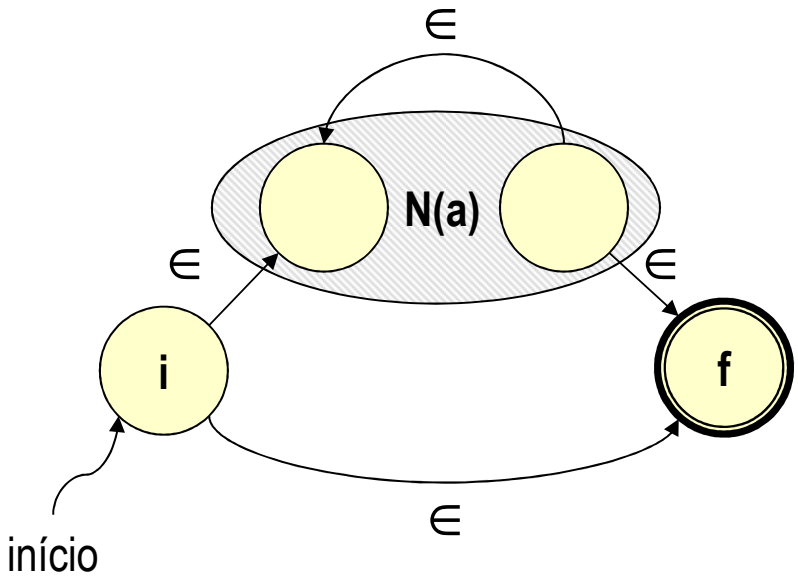


- Para *a* em Σ construímos o AFN:

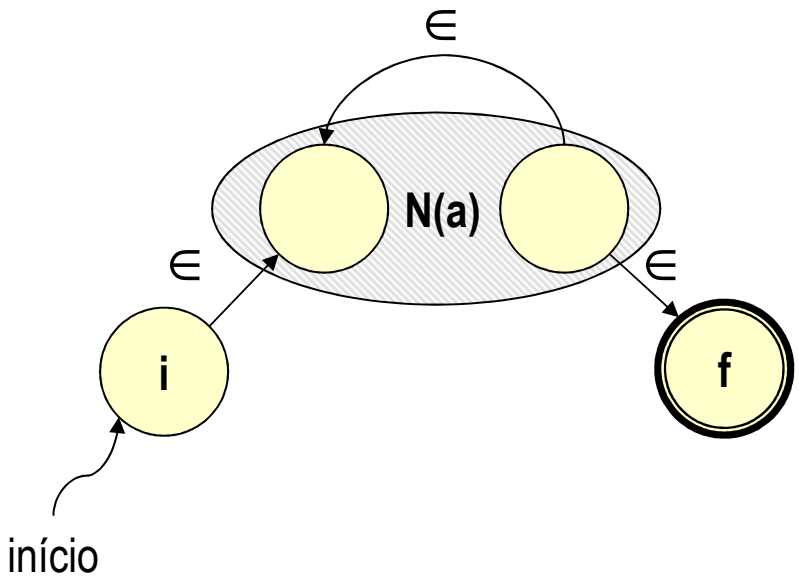


Construção de AFs a partir de Expressões Regulares

- Para a^* construímos o AFN:

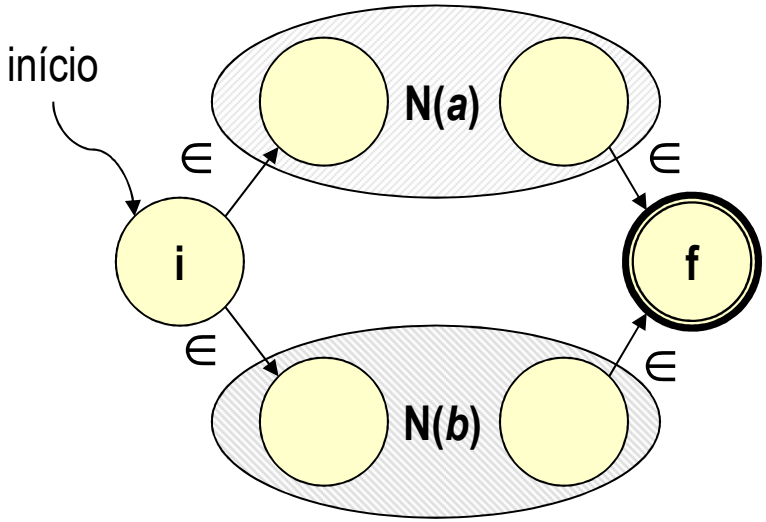


- Assim para a^+ construímos o AFN:



Construção de AFs a partir de Expressões Regulares

- Se N(a) e N(b) são AFs para expressões regulares a e b, então para expressão $a \mid b$:



- Se N(a) e N(b) são AFs para expressões regulares a e b, então para expressão $a\ b$: