

CET 058 – Compiladores

Prof. Mathias Santos de Brito

Ementa

- Introdução ao Estudo dos Compiladores
- Linguagens de Programação
- Tradutores e Compiladores
- Análise Léxica
- Análise Sintática
- Geração de Código Intermediário
- Otimização de Código
- Gerência de Memória
- Geração de Código Objeto



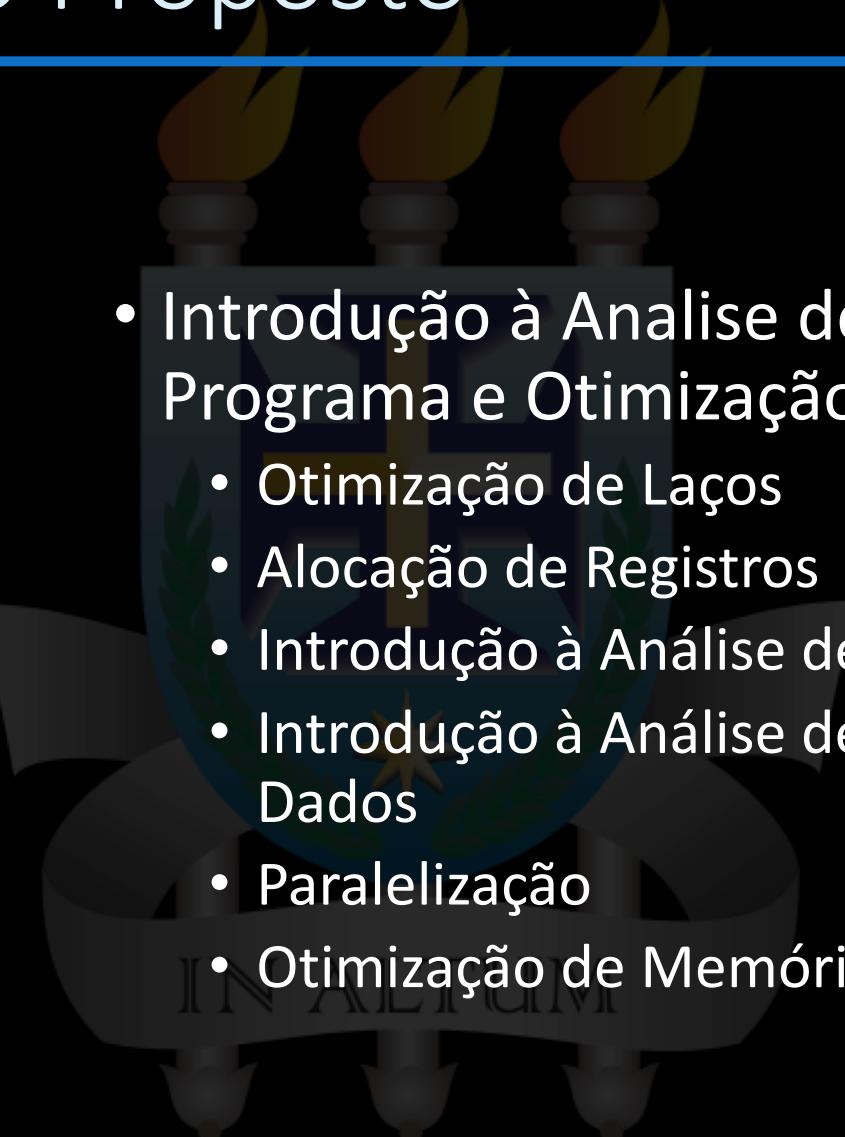
Conteúdo Programático - Atual

- Linguagens de Programação
 - Evolução das Linguagens de Programação
- Análise Léxica
 - Gramática e Linguagens Regulares (revisão)
 - Especificação, Implementação e tabela de símbolos
- Análise Sintática
- Gramáticas Livres-de-Contexto (revisão)
 - Análise descendente (top-down)
 - Análise Redutiva (bottom-up)
 - Recuperação de Erros
- Tradução dirigida por Sintaxe
- Geração de Código Intermediário
- Linguagens Intermediárias
 - Construção da Tabela de Símbolos
 - Geração de Código para Comando de Atribuição, Expressões Lógicas, Comandos de Controle e Backpatching
- Otimização de Código
- Otimização de Código Intermediário
 - Otimização de Código para Expressões Aritméticas
- Gerência de Memória
- Geração de Código Objeto

Conteúdo Programático - Proposto

Estrutura Baseada na Disciplina MIT 6035 do Prof. Martin Rinard
<http://6.035.scripts.mit.edu/fa18/schedule.html>

Conteúdo Programático Proposto

- 
- Análise Léxica
 - Expressões Regulares
 - Gramáticas Livres-de-Contexto (revisão)
 - Análise descendente (top-down)
 - Análise Redutiva (bottom-up)
 - Análise Sintática
 - Representações Intermediárias
 - Geração de Código (não otimizado)
 - Introdução à Analise de Programa e Otimização.
 - Otimização de Laços
 - Alocação de Registros
 - Introdução à Análise de Programa
 - Introdução à Análise de Fluxo de Dados
 - Paralelização
 - Otimização de Memória

Compiladores é uma das disciplinas
mais complexas da Ciência da
Computação.

Um desafio para ambos Professor e Estudante

Avaliações

- Provas – Peso total 30%
- Prova 1 – 01/04/2019
- Prova 2 – 29/05/2019
- Projeto – Peso Total 70%
- 4 Etapas
 - Análise Léxica – 24/04/2019
 - Análise Sintática – 29/05/2019
 - Otimização – 27/06/2019

Projeto e Organização da Disciplina

- Ferramentas a serem utilizadas
 - Flex e BISON
 - Talvez LLVM
- Organização da Disciplina e Comunicação
 - <https://github.com/CET058>
- Repositório de Suporte
 - <https://github.com/CET058/2019.1>
- Contato por e-mail através de msbrito@uesc.br com título iniciando em [CET058]
 - Atenção e-mails sem esse título poderão demorar de ser respondidos.



Análise Léxica

O que é?

“É a primeira Fase de um Compilador. Sua tarefa principal é a de ler os caracteres de entrada e produzir uma sequência de *tokens* que o *parser* utiliza para a análise sintática”

Alfredo V. Aho; Compiladores: Princípios, Técnicas e Ferramentas

Objetivos

- Ler o arquivo de entrada e escaneia os caracteres.
- Agrupa-os em Lexemas e produz um *token* como saída.
- Pode remover espaços e comentários do código-fonte.
- Expandir Macros encontradas no código-fonte

Tokens e Lexemas

- *Tokens* são um conjunto de caracteres que possui um significado
- Podem ser:
 - identificadores
 - Constantes
 - Operadores
 - Etc.
- Um **Lexema** é o conjunto de caracteres que forma o *token*.
- Um lexema pode ser composto por um ou mais caracteres
- Um *token* pode ser composto por mais de um lexema.

Um exemplo... Quais *Tokens* podemos extrair?

```
#include <stdio.c>

unsigned int fib(unsigned int n) {
    if (n == 0) {
        return 0;
    }
    if (n == 1) {
        return 1;
    }
    return fib(n-2) + fib(n-1);
}

int main() {
    for(i=0; i<10; i++) {
        printf("%d", fib(i));
    }
}
```

Antes de Iniciar a análise sintática devemos extrair os *tokens*, tente encontrar alguns *tokens* e classifique-os...

<id, “n”> <number, “0”> <if, >

Exemplos

Token	Padrão	Exemplos de Lexemas
if		if
relação		<, >, <=, >=, <>, >, >=
id		Idade, altura
Número		1, 3.52, 5.4345E6

Problemas Durante o Processo

- Ambiguidades – Quando definimos a nossa linguagem podemos ter que lidar com ambiguidades do gênero:
 - Exemplos:
 - int vs print
 - > vs >=
- Lookahead
 - É o nome da técnica usada para identificar se a leitura do *token* atual encerrou baseado na leitura de caracteres seguintes, definindo assim o termo do *token* atual bem como o início do próximo *token*.

Antes da teoria... Prática!

Expressões Regulares

Análise Léxica

Antes da Teoria... Prática!

- Antes de mergulhar nos detalhes das expressões regulares e suas formalidades, vamos brincar com uma ferramenta que usa expressões regulares.
- Vamos fazer alguns exercícios e entender como as expressões regulares funcionam na prática.

Como extrair os *tokens*?

- Podemos utilizar expressões regulares para extrair os *tokens* de um *stream* de caracteres.
- Expressões regulares são extremamente úteis em diversas área da computação e suas aplicações.
 - Administração de Redes
 - Análise de Dados
 - Etc...
- Diversas ferramentas usam expressões regulares e muitas linguagens oferecem bibliotecas para processar *strings* usando expressões regulares.
 - sed, awk, perl
- Um problema potencial são as diferentes sintaxes para representar uma Expressão Regular.

Brincando com o Awk

Análise Léxica

Sintaxe para Expressões Regulares em AWK

- ^ caractere no começo do string.
- \$ caractere no final do string
- . Qualquer caractere simples incluindo nova linha
- [...] Lista de caracteres podendo usar intervalos como a-z, A-Z, 0-9
- | usado para indicar alternativas, como um OU.
- * o símbolo o regular anterior pode se repetir.
- + como o anterior mas deve ocorrer pelo menos uma vez
- ? O símbolo anterior deve ocorrer uma vez ou nenhuma.
- Ex para detecção de um identificador.
[_a-zA-Z][_a-zA-Z0-9]*

Teoria: Linguagens e Expressões Regulares

Revisão de Teoria da Computação...

Alfabetos e Cadeias

- Expressões Regulares são uma excelente ferramenta para especificar padrões que podem ocorrer em uma **cadeia** de caracteres.
 - Nos referimos diariamente à cadeia de caracteres como strings. Os dois conceitos são equivalentes.
- Ao conjunto de caracteres válidos em uma **cadeia** chamamos de **alfabeto**.
 - Ex. o conjunto $A = \{0,1\}$ é o alfabeto binário.
 - \emptyset denota o conjunto vazio

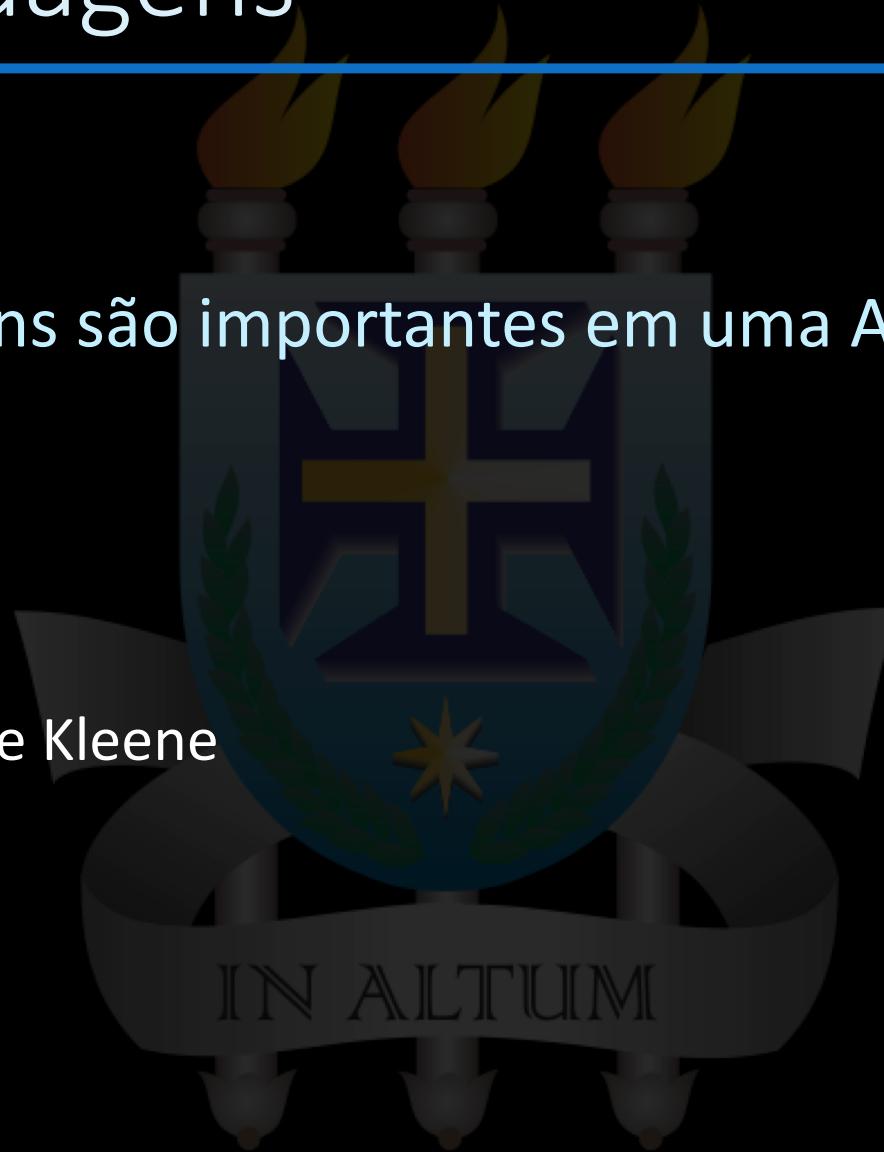
Linguagens Regulares

“Uma **linguagem** é qualquer conjunto contável de cadeias de um alfabeto fixo.”

“ \emptyset , o conjunto vazio, ou $\{\epsilon\}$ o conjunto contendo apenas a cadeia vazia são linguagens sob essa definição”

Operações Sobre Linguagens

- Três operações sobre linguagens são importantes em uma Análise Léxica elas são:
 - União
 - Concatenação
 - Kleene-Star ou Fecho de Kleene
 - Kleene-Plus ou Fecho Positivo de Kleene



- A união é uma operação bem conhecida por nós, vejamos:
 - “Dada duas linguagens L e M a sua união é composta por todas as cadeias formadas a partir de uma cadeia da primeira linguagem e uma cadeia da segunda linguagem em todas as formas possíveis.” (AHO, 1995; p. 76)
 - Símbolos - U na matemática ou | em sistemas UNIX
 - “União de $L \cup M = \{s | s \text{ está em } L \text{ ou } s \text{ está em } M\}$ ” (AHO, 1995; p. 76)
- Dada uma Linguagem $L = \{a, b, \dots, z, A, B, \dots, Z\}$ e $D = \{0, 1, \dots, 9\}$
- “ $L \cup D =$ O conjunto de Letras e Dígitos – Estritamente falando a linguagem com 62 cadeias de tamanho um, cada uma tendo uma letra ou dígito”. (AHO, 1995; p. 77)

Concatenação

- “A concatenação é formada por uma cadeia da primeira linguagem e uma cadeia da segunda linguagem em todas as formas possíveis e concatenando-as.” (AHO, 1995; p. 76)
 - Símbolo - . ou implícito
 - $LM = \{st | s \text{ está em } L \text{ e } t \text{ está em } M\}$
 - A operação de concatenação pode também ser escrita como $L.M$
- Dada uma Linguagem $L = \{a, b, \dots, z, A, B, \dots, Z\}$ e $D = \{0, 1, \dots, 9\}$
- “ LD é o conjunto de 520 cadeias de tamanho 2 cada um consistindo em uma letra seguida por um dígito.” (AHO, 1995; p. 77)

Fecho de Kleene

- O Fecho de Kleene, representado por L^* “é o conjunto de cadeias obtidas concatenando L zero ou mais vezes.” (AHO, 1995; p. 76)
 - $L^0 = \{\epsilon\}$
- Dada uma Linguagem $L = \{a, b, \dots, z, A, B, \dots, Z\}$
- “ L^* é o conjunto de todas as cadeias de letras, incluindo ϵ a cadeia vazia.” (AHO, 1995; p. 77)

Fechamento positivo de Kleene

- O Fecho positivo de Kleene, representado por L^+ é semelhante ao fecho de kleene com a diferença que ϵ não é possível.
 - Símbolo + sobrescrito, normal em UNIX.
- Dada uma Linguagem $D = \{0, 1, \dots 9\}$
- “ $D^+ =$ é o conjunto de todas as cadeias de um ou mais dígitos.”
(AHO, 1995; p. 77)

Expressões Regulares

- Como vimos anteriormente as expressões regulares são um ferramenta útil para definirmos um padrão que estamos buscando em uma cadeia de caracteres.
- A notação de expressões regulares que utilizamos tanto no awk como no Flex, se diferenciam um pouco da notação formal.
- Vejamos alguns exemplos.

Expressões Regulares

- Suponhamos que *letra_* seja o conjunto de todas as letras do alfabeto mais o caractere underscore e *digito* representando qualquer dígito de 0 à 9.
 - Tomemos por base a seguinte expressão regular abaixo:
 - $\text{letra_}(\text{letra_}|\text{digito})^*$
 - A barra vertical, como vimos é união e *letra_* está sendo concatenada com a união entre parêntese.
 - Esta expressão regular representa o conjunto dos identificadores da linguagem C.
- Adaptado de (AHO, 1995; p. 77)

Expressões Regulares: Regras Base

1. “ ϵ é uma expressão regular, ou seja, a linguagem cujo único elemento é a cadeia vazia.”
 2. “Se a é um símbolo pertencente a Σ , então \mathbf{a} é uma expressão regular, e $L(\mathbf{a}) = \{a\}$, ou seja, a linguagem com uma cadeia, de tamanho um, com a em sua única posição. Observe que por convenção, usamos itálico para símbolos e negrito para sua expressão regular correspondente.”
- (AHO, 1995; p. 77)

Expressões Regulares

- “As expressões regulares são construídas recursivamente a partir de expressões regulares menores.
- “**Indução:** existem 4 partes na indução, por meio das quais expressões regulares maiores são construídas a partir das menores...”
 1. $(r)|(s)$ é uma expressão regular denotando a linguagem $L(r) \cup L(s)$.
 2. $(r)(s)$ é uma expressão regular denotando a linguagem $L(r)L(s)$.
 3. $(r)^*$ é uma linguagem denotando $(L(r))^*$
 4. (r) é um linguagem denotando $L(r)$. Esta última regra diz que podemos acrescentar pares de parêntesis em torno das expressões sem alterar a linguagem que eles denotam.
- (AHO, 1995; p. 77)

Expressões Regulares: Precedência

- a) O operador unário $*$ possui precedência mais alta e é associativa à esquerda.
 - b) A concatenação tem a segunda maior prioridade e é associativa à esquerda.
 - c) | Tem a precedência mais baixa, e é associativa à esquerda.
- (AHO, 1995; p. 77)

Expressões Regulares

- Ex. “Considere $\Sigma = \{a, b\}$
 1. A expressão regular $a|b$ denota a linguagem $\{a, b\}$
 2. $(a|b)(a|b)$ denota $\{aa, ab, ba, bb\}$, a linguagem de todas as cadeias de tamanho dois do alfabeto Σ . Outra expressão regular para a mesma linguagem é $aa|ab|ba|bb$
 3. a^* denota as linguagens consistindo em todas as cadeias de zero ou mais *as*, ou seja $\{\epsilon, a, aa, aaa, \dots\}$
 4. $(a|b)^*$ denota o conjunto de todas as cadeias consistindo em zero ou mais instâncias de *a* ou *b*, ou seja todas as cadeias de *as* e *bs*; $\{\epsilon, a, b, aa, ab, ba, bb, aaa, \dots\}$. Outra expressão regular para a mesma linguagem é $(a^*b^*)^*$
 5. $a|a^*b$ denota a linguagem $\{a, b, ab, aab, aaab, \dots\}$, ou seja a cadeia *a* e todas as cadeias consistindo em zero ou mais *as* e terminado em *b*.”
- (AHO, 1995; p. 77-78)

Expressões Regulares

- Uma linguagem que pode ser definida por uma expressão regular é chamada de *conjunto regular*.
- Duas expressões regulares são equivalentes se elas denotam o mesmo conjunto regular e as descrevemos como $r = s$
- As expressões regulares possuem uma diversidade de expressões regulares:
- (AHO, 1995; p. 78)

Expressões Regulares: Leis Algébricas

LEI	DESCRIÇÃO
$r s = s r$	é comutativo
$r (s t) = (r s) t$	é associativo
$r(st) = (rs)t$	A concatenação é associativa
$r(s t) = r s r t; (s t)r = s r t r$	A concatenação distribui entre
$\epsilon r = r \epsilon = r$	ϵ é o elemento identidade para concatenação
$r^* = (r \epsilon)^*$	ϵ é garantido em um fechamento
$r^{**} = r^*$	* é igual potência

FIGURA 3.7 Leis algébricas para expressões regulares.

(AHO, 1995; p. 78)

Definições Regulares

- Para maior conveniência, podemos dar nome às expressões regulares, como já vimos anteriormente com o exemplo do identificador em C.
 - “Se Σ é um alfabeto de símbolos básicos, então uma definição regular é uma sequência de definições da forma.
$$\begin{aligned}d_1 &\rightarrow r_1 \\d_2 &\rightarrow r_2 \\&\dots \\d_n &\rightarrow r_n\end{aligned}$$
1. Cada d_i é um novo símbolo, não em Σ e não o mesmo que qualquer outro d_i .
 2. Cara r_i é uma expressão regular envolvendo símbolos do alfabeto $\Sigma \cup \{d_1, d_2, \dots, d_{n-i}\}$.
- (AHO, 1995; p. 78)

Definições Regulares

- Exemplo:

$$letra_ \rightarrow A | B | C | \dots | Z | a | b | c | \dots | z | _$$
$$digito \rightarrow 0 | 1 | 2 | \dots | 9 |$$
$$id \rightarrow letra_ (letra | digito)^*$$


Exercícios

- Fazer exercícios da seção 3.3 do livro (AHO, 1995; p. 78)



Flex – Gerador de Analisador Léxico

Sintaxe para formação de expressões regulares.

PADRÃO	CASA COM...
[0-9]	all the digits between 0 and 9
[0+9]	either 0, + or 9
[0, 9]	either 0, ‘,’ or 9
[0 9]	either 0, ‘ ’ or 9
[-09]	either - , 0 or 9
[-0-9]	either – or all digit between 0 and 9
[0-9]+	one or more digit between 0 and 9
[^a]	all the other characters except a
[^A-Z]	all the other characters except the upper case letters
a{2, 4}	either aa, aaa or aaaa
a{2, }	two or more occurrences of a
a{4}	exactly 4 a's i.e, aaaa
.	any character except newline
a*	0 or more occurrences of a
a+	1 or more occurrences of a
[a-z]	all lower case letters
[a-zA-Z]	any alphabetic letter
w(x y)z	wxz oPwyz

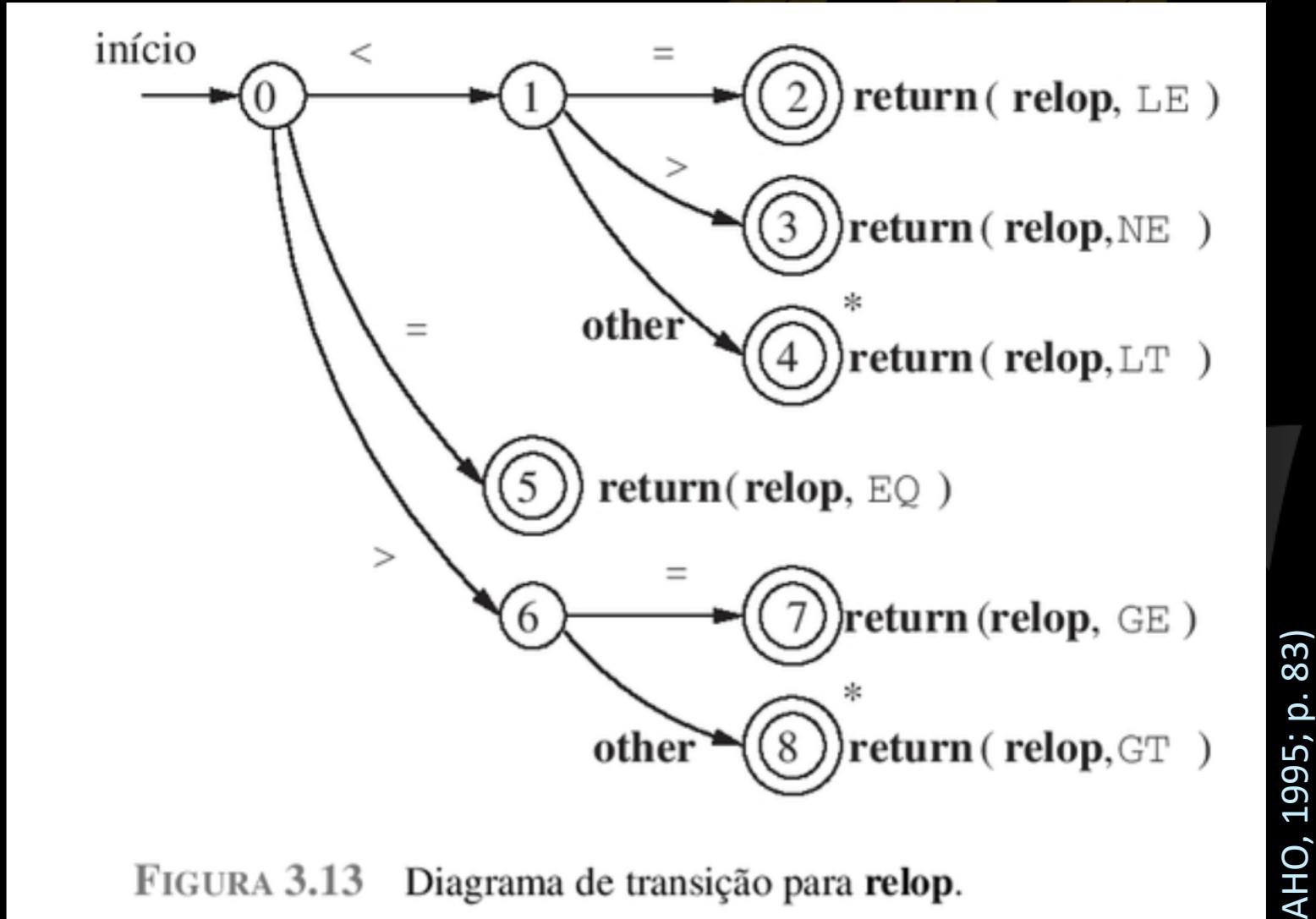
Atenção

- Note que, dentro de classes de caracteres, operadores de expressões regulares perdem o seu significado e se tornam literais.
- Com a exceção dos caracteres abaixo que precisam ser escapados.
 - \ caractere de escape
 - - intervalo
 - [] operadores de classe de caracteres
 - ^ no início da classe.

Diagramas de Transição

- Um passo inicial na análise léxica é a construção de **Diagramas de Transição** a partir dos padrões definidos.
 - Dentre os nós no conjunto de estados definimos um estado inicial e um ou mais estados finais.
- Estes diagramas são compostos por nós e arestas, onde um nó representa um estado e as arestas uma condição de transição.
- Esses diagramas já são velho conhecidos de vocês, e vocês os estudaram na disciplina Teoria da Computação como uma forma de representar Autômatos, no nosso caso estamos lidando com Autômatos Finitos Determinísticos (AFD ou DFA em Inglês) nos exemplos a seguir.

Um exemplo de Diagrama de Transição

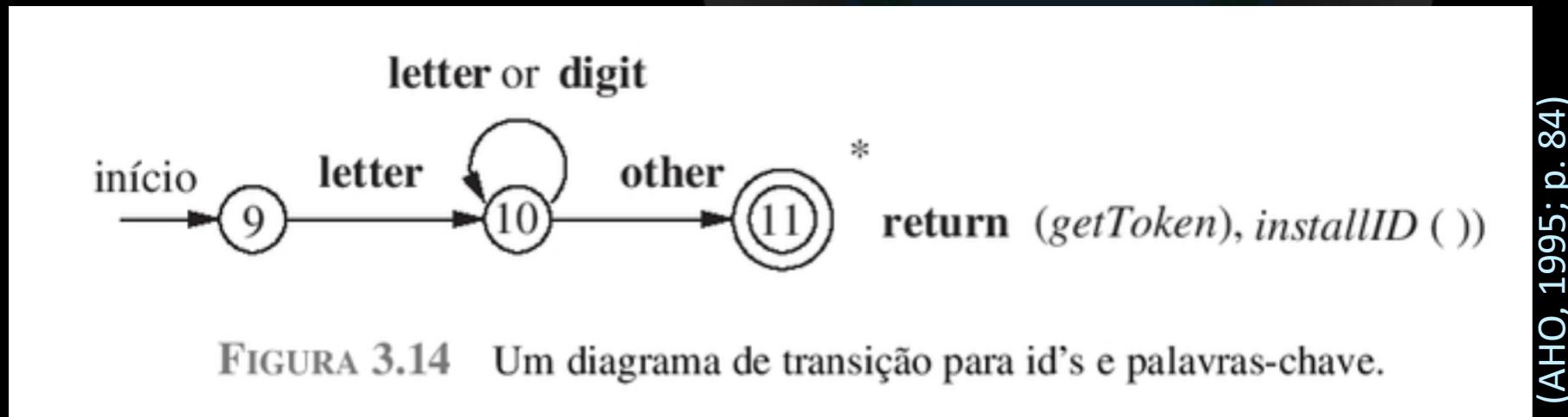


Fonte: AHO, Alfred V.; SETHI, Ravi; ULLMAN, Jeffrey D. Compiladores: princípios, técnicas e ferramentas. Rio de Janeiro: Guanabara Koogan, c1995. 344p. ISBN 8527703122 (broch.)

(AHO, 1995, p. 83)

Ambiguidades, Identificadores e Palavras Reservadas

- O problema de ambiguidade entre identificadores e palavras reservadas constitui um problema adicional na análise Léxica.
- Pois palavras chaves podem ser uma substring, sufixo ou prefixo de um identificador. Analise o Diagrama abaixo!

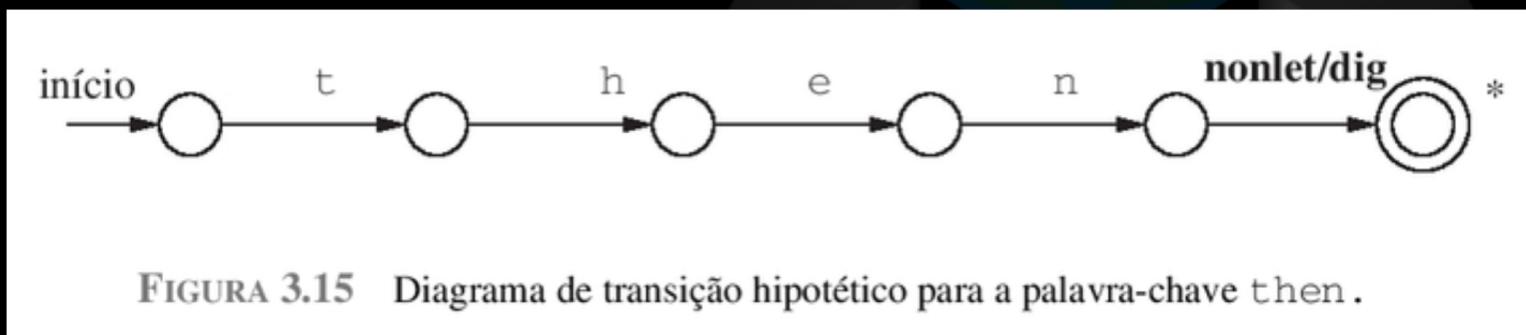


Fonte: AHO, Alfred V.; SETHI, Ravi; ULLMAN, Jeffrey D. Compiladores: princípios, técnicas e ferramentas. Rio de Janeiro: Guanabara Koogan, c1995. 344p. ISBN 8527703122 (broch.)

(AHO, 1995, p. 84)

Ambiguidades, Identificadores e Palavras Reservadas

- Temos duas soluções básicas para este problema:
 1. Adicionar as palavras-reservadas na tabela de símbolos e marca-las como palavra-reservada. Ao reconhecer um lexema do tipo ID este é colocado na tabela de símbolos se ainda não existe, ou retornando uma referência ao símbolo na tabela.
 2. Adicionar um diagrama específico para cada palavra-chave. Analise o diagrama abaixo: para “then” e “thenextvalue”.



Ambiguidades, Identificadores e Palavras Reservadas

- Em relação à técnica 2, não precisamos submeter a análise para diferentes diagramas.
- Podemos escrever um só diagrama que contemplem todos os diagramas necessários para o reconhecimento dos lexemas.

Ambiguidades, Identificadores e Palavras Reservadas

- Fazer exercícios da página 88 e 89 do livro (AHO, 1995)



Expressões Regulares AFD/DFA e AFN/NFA

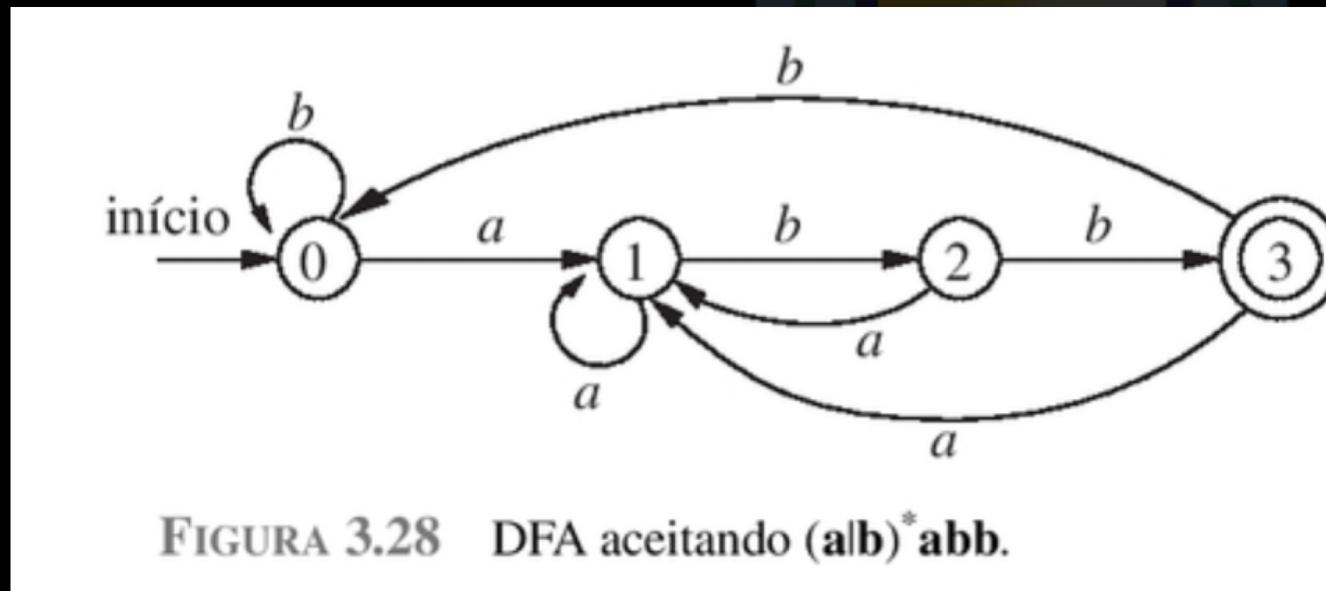
- Expressões Regulares é uma notação para definir padrões, porém a sua análise e simulação em computadores são realizados através de autômatos finitos.
- A grosso modo a diferença entre autômatos finitos determinísticos e não-determinísticos é o fato de que o segundo pode conter diferentes arestas saindo de um estado e pode inclusive ter uma aresta rotulada com o símbolo ϵ , vazio.

Expressões Regulares AFD/DFA e AFN/NFA

- Ambos os tipos de autômatos podem ser utilizados para representar uma expressão regular.
 - Em geral AFDs são mais fáceis de simular do que AFN.
- A conversão de uma AFN para uma AFD, porém, pode ser muito complexa, caso essa complexidade seja maior que a simulação do AFN, este último pode ser a escolha correta.

Usando AFD

- Crie um AFD que aceite a linguagem $(a|b)^*abb$



Usando AFD

- Façam os Exercícios das Páginas 96 e 97 do livro (AHO, 1995)



Tabela de Símbolos

- A tabela de símbolos armazena

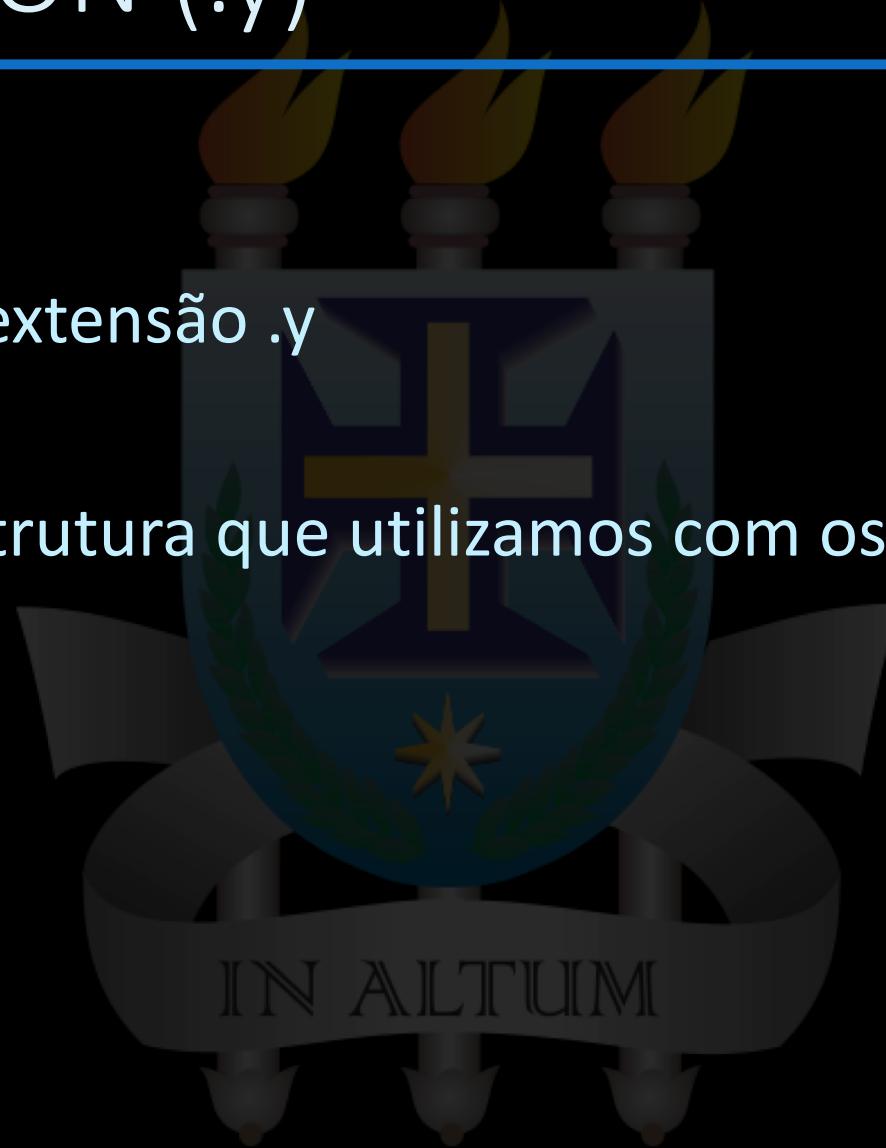


Bison: Gerador de Parsers para análise Sintática

- Bison é um gerador de *parser* para linguagens livres de contexto.
- Bison em modo POSIX é compatível com YACC
- Bison pode gerar parsers do tipo LALR, canonical LR, IELR e GLR.
- Bison pode ser utilizado com o *flex* para realizar a extração de tokens.

Arquivos fontes de BISON (.y)

- Os fontes de BISON possuem extensão .y
- A estrutura lembra muito a estrutura que utilizamos com os nossos fontes em FLEX.
- Vamos dar uma olhada...



Estrutura de um Fonte BISON

```
1 %{
2
3 /* Código C, use para #include, variáveis globais e constantes
4 * este código ser adicionado no início do arquivo fonte em C
5 * que será gerado.
6 */
7
8 %}
9
10 /* Declaração de Tokens no formato %token NOME_DO_TOKEN */
11
12
13 %%
14 /* Regras de Sintaxe */
15 %%
16
17 /* Código C geral, será adicionado ao final do código fonte
18 * C gerado.
19 */
```

~

BISON e Flex

- Para conectarmos o BISON com o FLEX algumas coisas devem ser observadas.
- Os TOKENS serão definidos no código BISON, que irá gerar um arquivo .h (header C), o qual deveremos incluir no fonte do FLEX.
- Cada regra no FLEX deverá retornar o TOKEN.
- O fonte FLEX não terá mais uma função **main** visto que quem irá invocar a função de **parser** do analisador é o BISON.

Um exemplo de .l para ser integrado ao BISON

```
1 /* SEÇÃO DE DEFINIÇÕES */
2
3 %{
4     #include "parser.tab.h"
5 %}
6
7 /* Definições de Nomes */
8 NUMBER          [0-9] +
9 /* FIM DA SEÇÃO DE DEFINIÇÕES */
10
11
12 /*Seção de Regras*/
13 %%
14
15 %%
16 /* Fim da seção de Regras */
17
18
19
20 /*Seção de Código do Usuário*/
21
22 int yywrap() { }
```

Gerando o parser

- Até aqui, utilizávamos o flex para gerar nosso analisador léxico.
- Precisamos nessa etapa construir também o nosso parser.
- Para isso devemos utilizar o comando **bison**, a sequência de comandos necessárias para a geração do parser são:

```
$ bison -d parser.y
```

```
$ flex analisador.l
```

```
$ gcc -o parser.x parser.tab.c lex.yy.c -lfl (ou -ll no mac)
```

Criando um Makefile

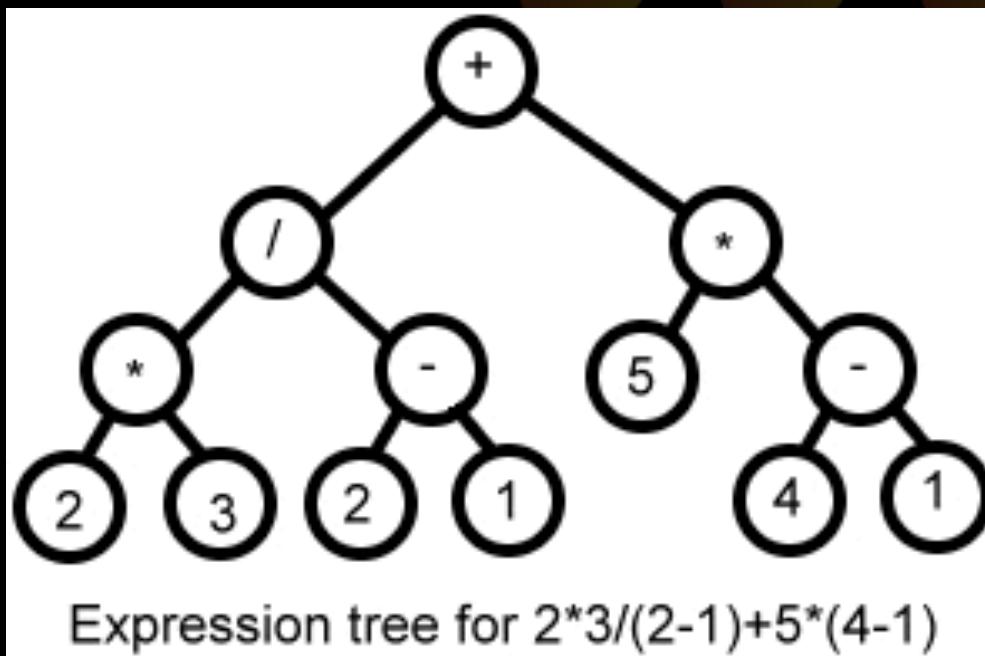
- Para facilitar o processo de geração do parser podemos criar um arquivo Makefile, assim basta digitarmos make, para que todos os comandos necessários sejam executados. Abaixo um exemplo de Makefile.

```
1 compile: parser.y analisador.l  
2         bison -d parser.y  
3         flex analisador.l  
~ 4         gcc -o parser.x parser.tab.c lex.yy.c -lfl  
+ 5
```

Vamos à prática...

- Liguem os computadores no Linux
- Verifiquem se o comando **bison** está instalado corretamente
- Clonem ou atualizem o nosso repositório, você irá encontrar lá um novo diretório chamado **bison**
- Entre no diretório `parser_de_expressão`
 - Note que você já tem um arquivo `Makefile` para facilitar o processo de compilação
- Vamos à prática... Olho na tela...

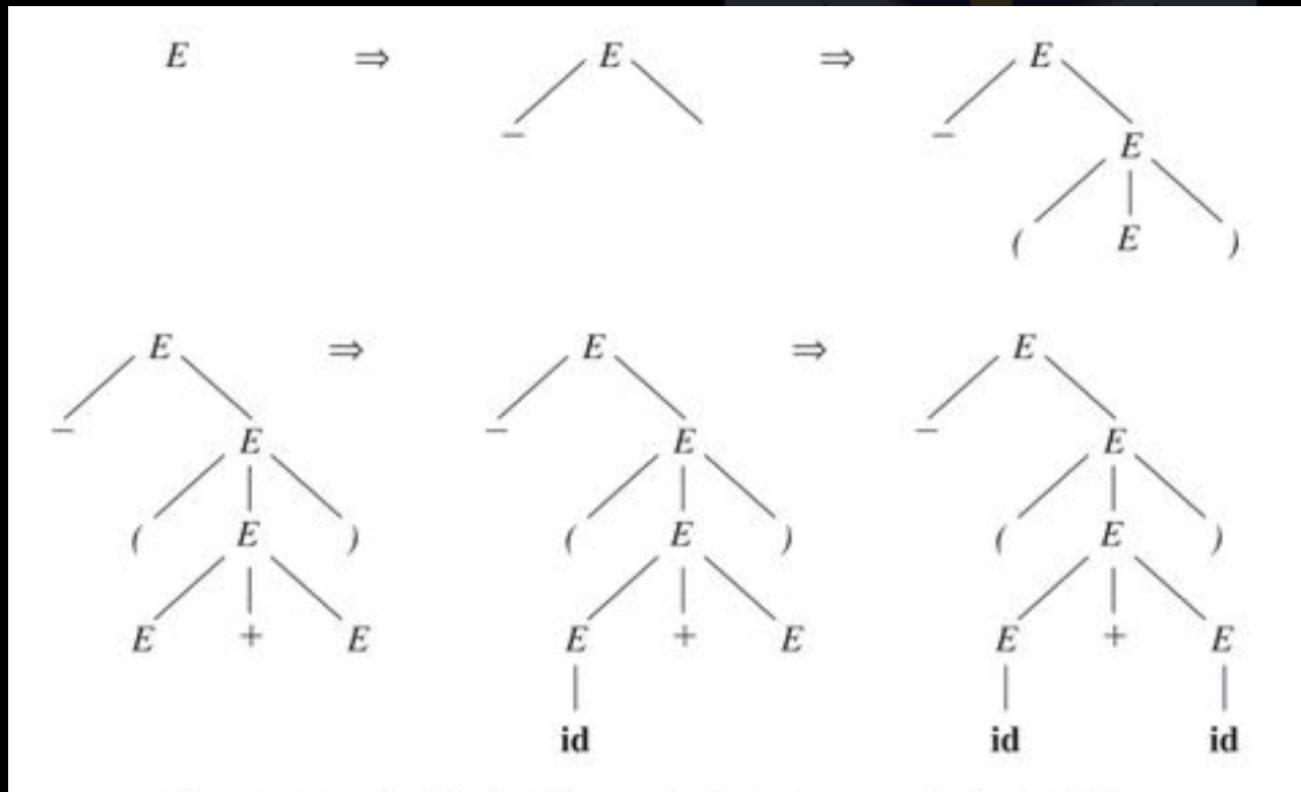
Árvore de Derivação para Expressão



- Saída \rightarrow [ADD [DIV [MUL [2] [3]] [SUB [2][1]]][MUL [5] [SUB [4][1]]]

Árvore de Derivação para Expressão

$-(id + id)$



Árvore de Derivação

$\text{if } E_1 \text{ then if } E_2 \text{ then } S_1 \text{ else } S_2$

