Universidade Federal de Uberlândia
Faculdade de Computação

Gramática Livre de Contexto (GLC)

(material baseado no livro [Aho et al., 2008])

Curso de Bacharelado em Ciência da Computação
GBC071 - Construção de Compiladores
Prof. Luiz Gustavo Almeida Martins

Fase de Análise (*Front-end*)

- A análise é organizada em torno da **sintaxe da linguagem de programação** a ser compilada
  - **Sintaxe** descreve a **forma apropriada** dos programas
  - **Semântica** define o **significado** dos programas

- Especificação das **regras precisas** que descrevem a estrutura sintática de programas bem formados é feita por uma **Gramática Livre de Contexto (GLC)**
  - Pode usar **Forma de Backus (*Backus-Naur Form* - BNF)**

- Técnica de compilação orientada pela gramática é conhecida como **tradução dirigida por sintaxe**
  - Analisador sintático que controla o processo de tradução
  - Demanda devido a limitações de memória para os 1os compiladores

Gramática Livre de Contexto (GLC)

- Descrevem naturalmente a **estrutura hierárquica** das construções de linguagens de programação

- **Ex:**
  - Comando ***if-then-else***:

    If (*expressão* **)** *comandos* **else** *comandos*

  - Produção da GLC:

    cmd → **if (** *expr* **)** *cmd* **else** *cmd*

Gramática Livre de Contexto (GLC)

- Benefícios de usar uma GLC no projeto:
  - Provê uma **especificação sintática precisa e fácil de entender** para as linguagens de programação
  - Permite a **construção automática de um analisador** sintático eficiente
    - Possibilita a detecção de ambiguidades sintáticas
  - Facilita a **tradução correta** de programas e a **detecção de erros**
  - Permite o **desenvolvimento iterativo e incremental** de uma linguagem
    - Novas construções podem ser incluídas para realizar novas tarefas

Gramática Livre de Contexto (GLC)

- Uma GLC *G* pode ser representada por uma quadrupla:

$$G = (V, T, P, S)$$

  - ***T*** = conjunto de símbolos terminais
    - Símbolos elementares da linguagem (***tokens***)
  - ***V*** = conjunto de símbolos não terminais
    - Representa as cadeias de terminais (**variáveis sintáticas**)
  - ***P*** = conjunto de produções (regras de transição)
    - Especifica as formas de uma construção da linguagem
    - **Cabeça** da produção (lado esquerdo) deve ter um não-terminal
    - **Corpo** da produção (lado direito) é uma sequência de terminais e não terminais
  - ***S*** = símbolo inicial da gramática (**símbolo sentencial**)
    - Deve ser um **símbolo não terminal**

Exemplo de Gramática

- Gramática para formar **expressões com somas e subtrações:**

G1 = ({*expr, term*}, {ID, NUM,OP, (,)}, P, *expr*)

- *expr* → *expr* **OP** *term* | *term*
- *term* → **ID** | **NUM** | **(** *expr* **)**

Exemplo de Gramática

- Gramática para definir a **chamada de funções:**

G2 = ({*call, optparams, params, param, exp*},
{**id**, **const**, ε}, P, *call*)

- *call* → **id (** *optparams* **)**
- *optparams* → *params* | ε
- *params* → *params* **,** *param* | *param*
- *param* → **const** | **id** | *exp*

Notação BNF

- ***Backus-Naur Form* (BNF):**
  - **Notação textual para especificar as produções** das gramáticas
  - Desenvolvida por **John Backus** para especificar a linguagem Algol 60
  - **Peter Naur** introduziu posteriormente algumas simplificações

- **Algumas características da notação:**
  - Operador **::=** indica uma produção da gramática
  - Operador **|** indica alternativas de expansão (**união**)
  - Operador ***** indica repetição (**fecho de Kleene**)
  - Colchetes (**[** e **]**) indicam opcionalidade do conteúdo
  - Colchetes agudos (**<** e **>**) delimitam os **símbolos não-terminais**
  - Quaisquer outros símbolos são considerados **símbolos terminais**

- **Ex:** *<expr> ::= <expr>* **+** *<digito>* | **(** *<expr>* **)** | *<digito>*

Derivações

- Uma gramática deriva cadeias:
  - Começa pelo símbolo inicial ***S*** (**primeira forma sentencial**)
  - Aplica substituições aos não terminais até restar **apenas símbolos terminais** (**última forma sentencial**)
  - Cada forma sentencial na sequência pode ser obtida da anterior
    - **Substituição de uma variável pelo corpo de uma de suas produções**

- O conjunto das cadeias de terminais que podem ser derivadas a partir do símbolo inicial formam **a linguagem** da gramática

- **Análise sintática** consiste em tentar descobrir **como derivar uma cadeia de terminais** (***tokens***)
  - Deve informar os **erros de sintaxe** da cadeia, se não aceita
  - **Ex:** *expr* $\Rightarrow$* 2 + 5 - 3 *call* $\Rightarrow$* soma(x , y)

Derivações

- As derivações de uma gramática permitem:
  - **Reconhecer a estrutura do programa**
    - Verificar se uma dada **sentença de *tokens* (terminais) é aceita pela linguagem** gerada pela gramática (***parsing***)
    - **Objetivo principal**
  - **Identificar as operações elementares** que devem ser realizadas em cada sentença da linguagem
  - **Determinar a ordem de execução** dessas operações elementares

Derivações

- **Exemplo:** considere a instrução ***a = a + 2 * b;***
  - **Cadeia de tokens:**

    <id,1> <=> <id,1> <+> <const,2> <> <id,2> <;>*

  - **Identificação das operações envolvidas:**
    - **Atribuição:** *<lado esq> <=> <lado dir> <;>*
    - **Soma (lado direito):** *<op esq> <+> <op dir>*
    - **Multiplicação (operando direito):** *<op esq> <*> <op dir>*
  - **Ordenação da execução:** multiplicação, soma e atribuição

- A **construção da árvore de derivação** da sentença facilita essas tarefas

Árvores de Derivação

- Mostra de **forma representativa** como uma cadeia é derivada a partir do símbolo inicial *S*
  - Adota uma estrutura de dados do tipo **árvore**
  - Cada produção provoca uma ramificação da árvore
  - Nós folha de uma árvore de derivação formam a cadeia gerada
    - Representam a cadeia de *tokens* (**símbolos terminais**)
    - Leitura é feita da esquerda para a direita

- **Análise ou reconhecimento:** processo de **encontrar a árvore de derivação** para uma determinada cadeia de terminais

Árvores de Derivação

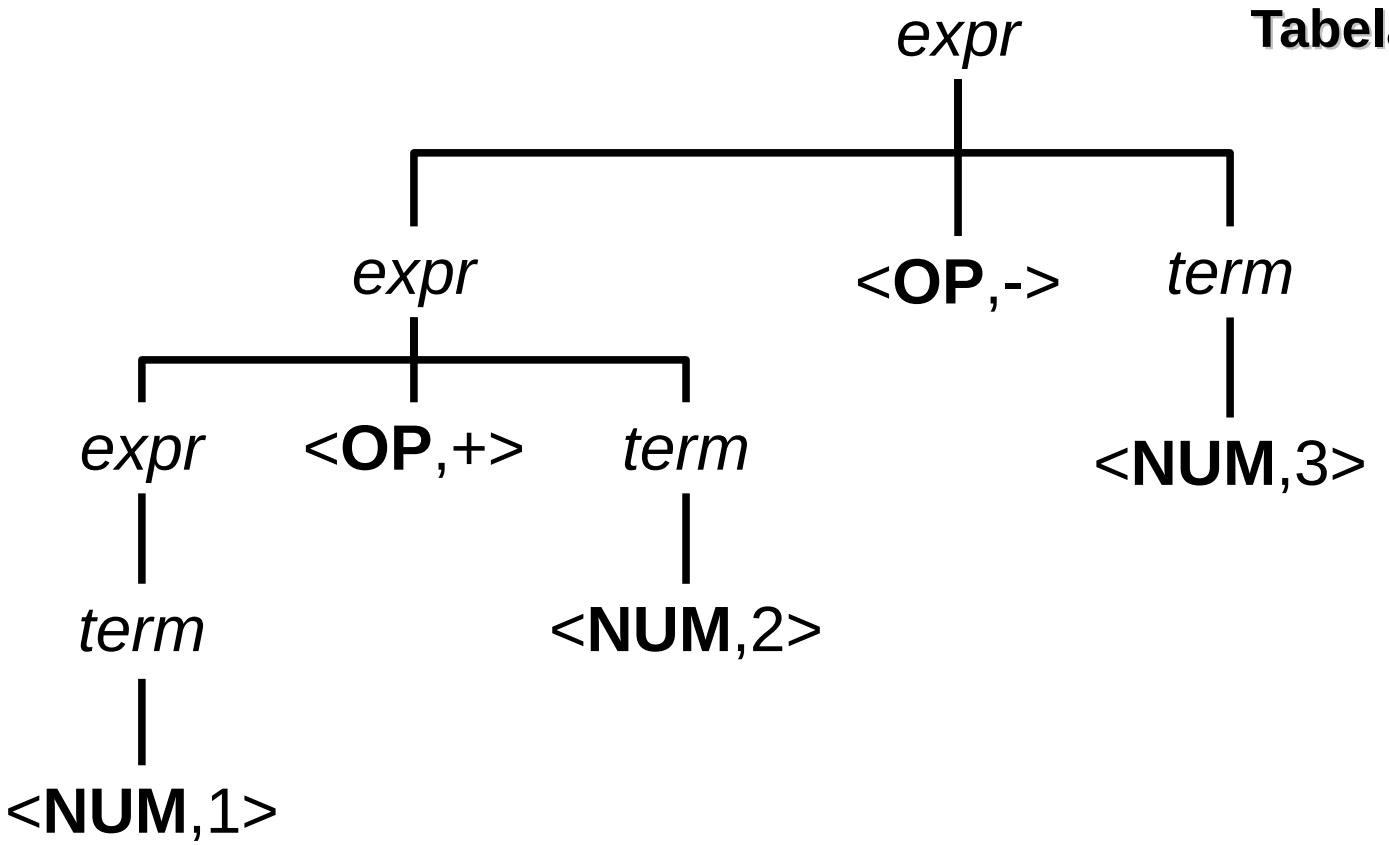
- Formalmente, uma árvore de derivação é uma árvore com as seguintes propriedades:
  - Nó raiz é rotulado pelo símbolo inicial *S*
  - Cada nó folha é rotulado por um terminal ou por ε
  - Cada nó intermediário é rotulado por um não terminal
  - Se ***A*** é o não terminal rotulando algum nó da árvore e X_1, X_2, …, X_n são os rótulos dos filhos de ***A***, então existe uma produção $A \rightarrow X_1 X_2 \ldots X_n$
    - **Caso especial:** se $A \rightarrow \varepsilon$ é uma produção, um nó rotulado com ***A*** pode ter um único filho rotulado com **ε**

Árvores de Derivação

- **Exemplo:** *2+5-3*

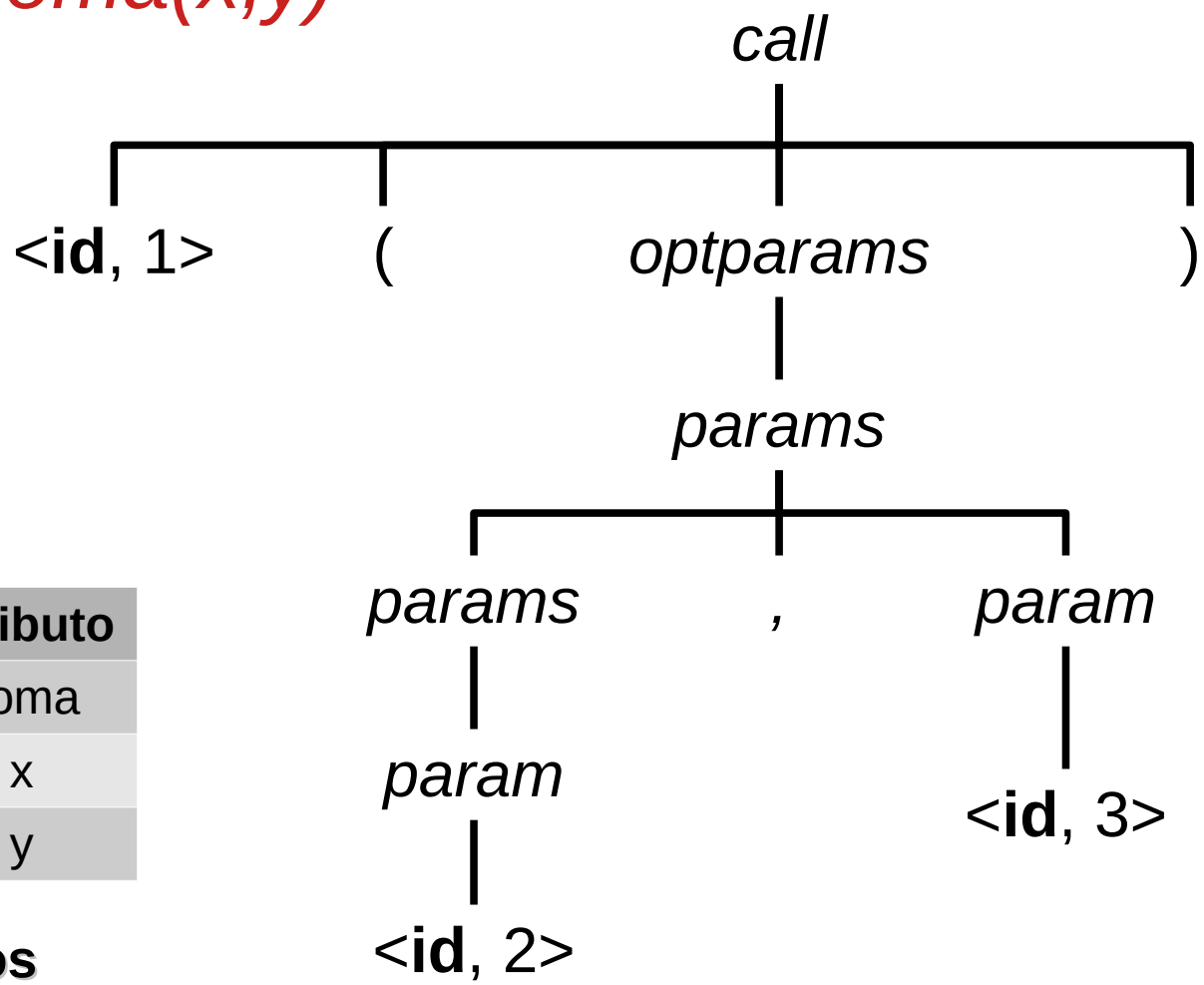
Posição	Token	Atributo	Tipo
1	NUM	2	INT
2	NUM	5	INT
3	NUM	3	INT

Tabela de Símbolos



Árvores de Derivação

- **Exemplo:** *soma(x,y)*



Posição	Token	Atributo
1	ID	soma
2	ID	x
3	ID	y

Tabela de Símbolos

Ambiguidade

- Reconhecimento de uma cadeia envolve **encontrar sua sequência de derivações**
  - **Problema:** pode haver diferentes combinações das produções para derivar uma mesma sentença

- **Ambiguidade: mais de uma árvore de derivação** por sentença
  - Cada árvore pode indicar **um significado diferente para sentença**
  - **Ex:** Diferente ordem de execução para as operações

- No projeto de compiladores devemos:
  - Especificar **gramáticas não ambíguas**

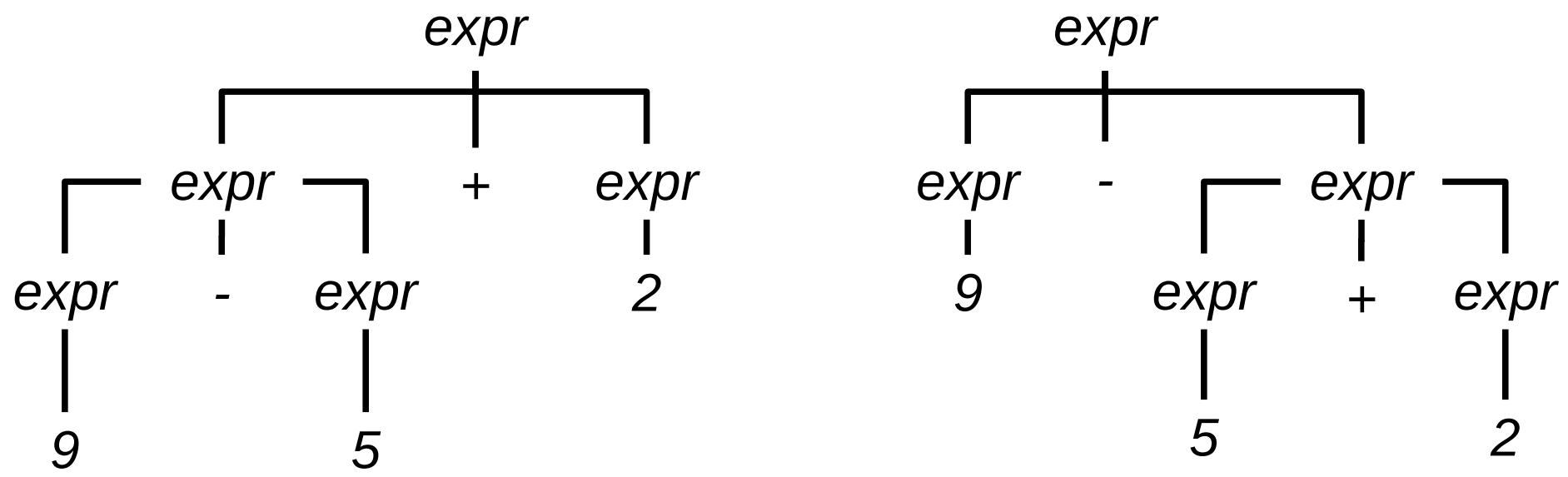
    E/OU
  - Usar **gramáticas ambíguas com regras adicionais** para tratar as ambiguidades

Ambiguidade

- **Ex:** Considere as produções de um gramática:

  expr → *expr* + *expr* | *expr* - *expr* | 0 | 1 | 2 | … | 9

- *9 - 5 + 2* pode gerar as árvores de derivação:



Associatividade de Operadores

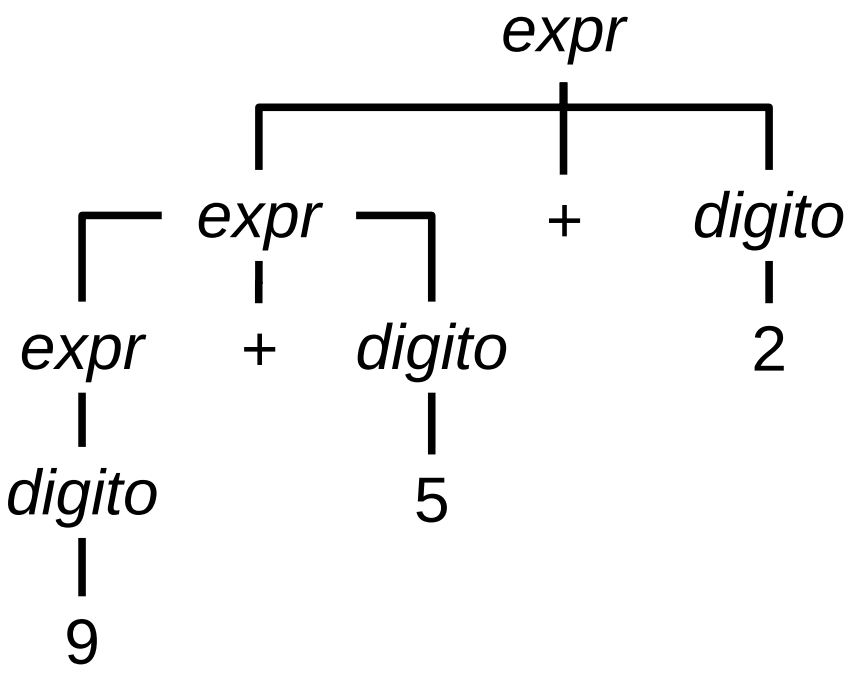
- Quando um operando está associado a mais de um operador, são necessárias **convenções para decidir qual aplicar:**
  - **Associatividade à esquerda:** ocorre quando o operando é associado ao **operador da esquerda**
    - **Ex:** operadores aritméticos
  - **Associatividade à direita:** ocorre quando o operando é associado ao **operador da direita**
    - **Ex:** exponenciação e atribuição

Associatividade de Operadores

- **Exemplo:**

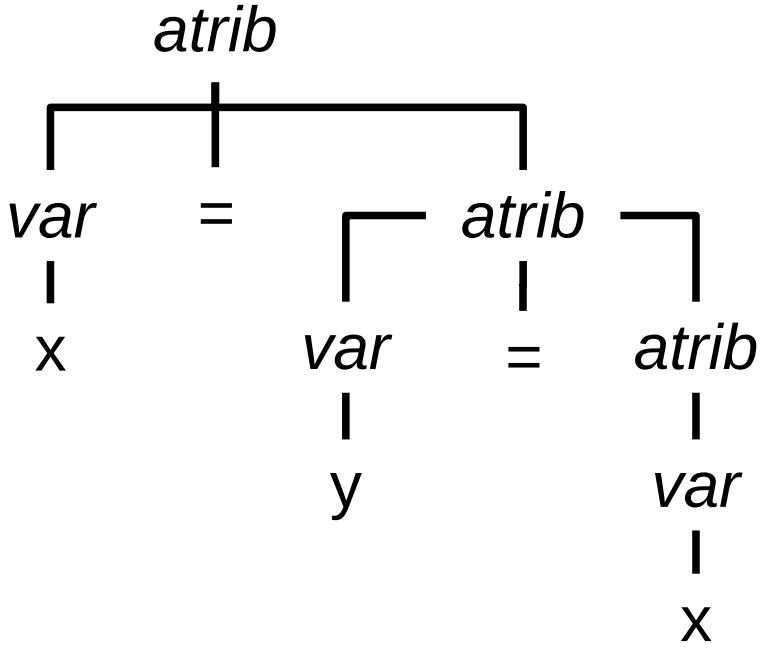
Associatividade à esquerda:

expr → *expr* + *digito* | *digito*
digito → 0 | 1 | … | 9



Associatividade à direita:

atrib → *var* = *atrib* | *var*
var → a | b | … | z

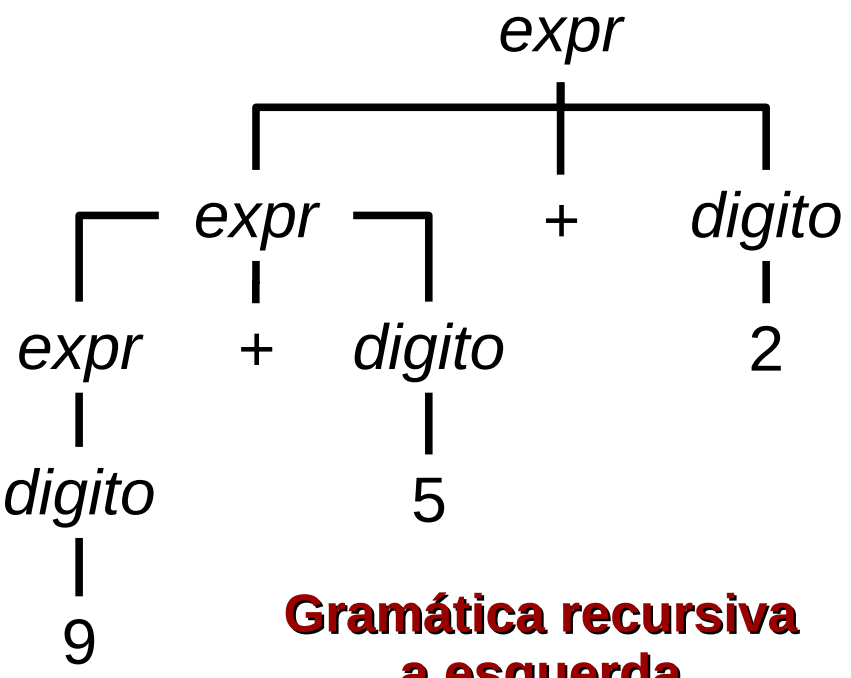


Associatividade de Operadores

- **Exemplo:**

Associatividade à esquerda:

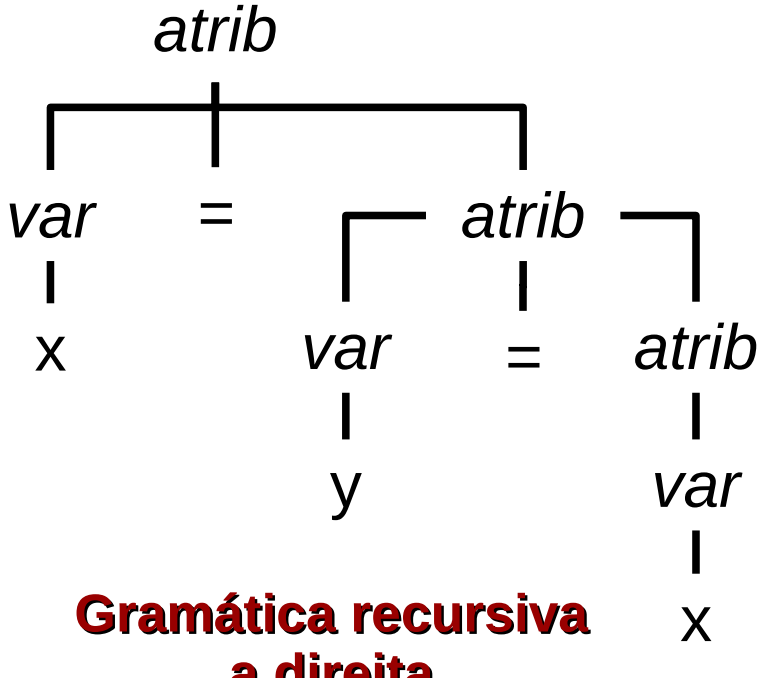
expr → *expr* + *digito* | *digito*
digito → 0 | 1 | … | 9



Gramática recursiva a esquerda

Associatividade à direita:

atrib → *var* = *atrib* | *var*
var → a | b | … | z



Gramática recursiva a direita

Precedência de Operadores

- Associatividade só se aplica a **ocorrências de operadores com a mesma precedência**
  - Não resolve ambiguidade de cadeias com diferentes precedências
  - **Ex:** 9 + 5 * 2

- Exige **regras que definam a precedência** dos operadores

- Uma gramática pode ser construída a partir de uma **tabela com a associatividade e a precedência** dos operadores
  - Cada **nível de precedência** é representado por **um não terminal**
    - Organizada da menor para a maior precedência
  - Usa um **não terminal extra** para as **unidades básicas**
    - Constantes, variáveis, parênteses, etc.

Precedência de Operadores

- **Exemplo:** considere as 4 operações aritméticas básicas

Operadores	Precedência	Associatividade	Não-Terminal
+ e -	Menor	esquerda	*expr*
* e /	Maior	esquerda	*termo*

Precedência de Operadores

- **Exemplo:** considere as 4 operações aritméticas básicas

Operadores	Precedência	Associatividade	Não-Terminal
+ e -	Menor	esquerda	*expr*
* e /	Maior	esquerda	*termo*

Não-terminal extra: *fator*

- Representa as expressões que **não podem ser desmembradas**
- **Ex:** parênteses, operações unárias, constantes, variáveis

Precedência de Operadores

- **Exemplo:** considere as 4 operações aritméticas básicas

Operadores	Precedência	Associatividade	Não-Terminal
+ e -	Menor	esquerda	*expr*
* e /	Maior	esquerda	*termo*

Não-terminal extra: *fator*

- Representa as expressões que **não podem ser desmembradas**
- **Ex:** parênteses, operações unárias, constantes, variáveis

- Produções da gramática:
  - *expr* → *expr* + *termo* | *expr* - *termo* | *termo*
  - *termo* → *termo* * *fator* | *termo* / *fator* | *fator*
  - *fator* → *digito* | (*expr*)

Aninhamento de instruções

- Considere a GLC para instruções em C:
  - **Palavras-chave** permitem reconhecer os comandos
  - **id** representa qualquer identificador
  - Produções para ***expr*** omitidas para simplificar

Produções da gramática:

cmd → **id** = *expr ;* |
if (*expr*) *cmd* |
if (*expr*) *cmd* **else** *cmd* |
while (*expr*) *cmd* |
do *cmd* **while** (*expr*) ; |
{ *bloco* }

bloco → *bloco* cmd | ε

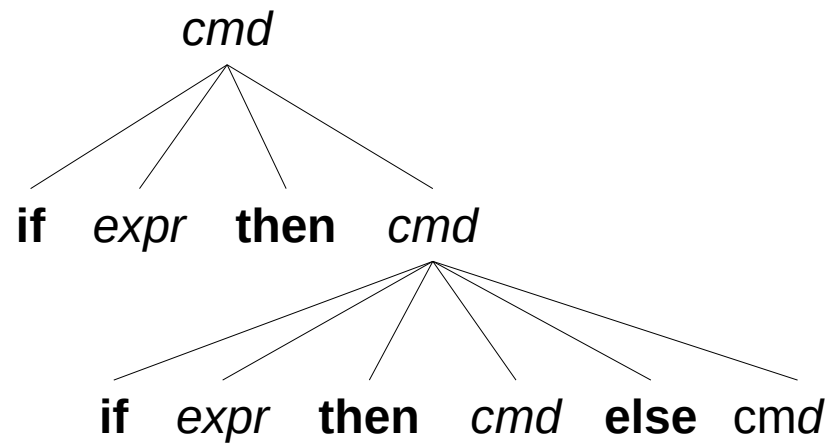
OBS: uso do "**;**" apenas em produções que não terminam com ***cmd***

Aninhamento de instruções

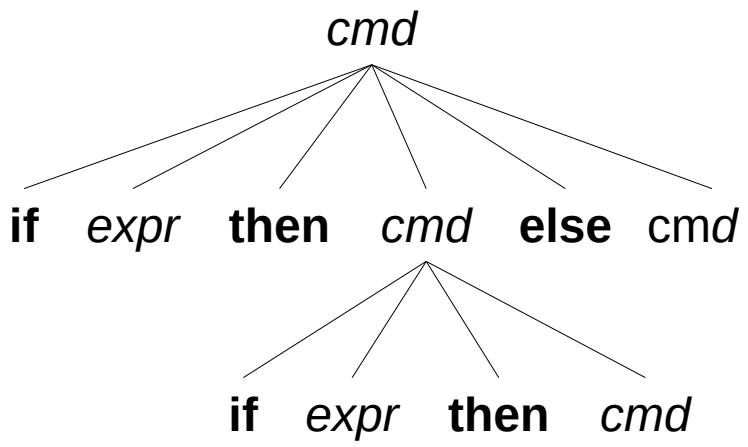
- **Problema:** aninhamentos entre ***if-then*** e ***if-then-else*** permitem diferentes árvores de derivação para a mesma sentença:

if *E1* **then** *S1* **if** *E2* **then** *S1* **else** *S2*

Árvore Derivação 1



Árvore Derivação 2



- **Árvore de derivação 1** é a preferida pelas linguagens:
  - **Regra:** casar o **else** com o **then** livre mais próximo

Aninhamento de instruções

- Regra pode ser incorporada diretamente na gramática:

Fonte: [Aho, 2008]

Gramática original (versão simplificada):

cmd → **if** *expr* **then** *cmd*
| **if** *expr* **then** *cmd* **else** *cmd*
| **outros**

Gramática modificada:

cmd → *cmd_casado*
| *cmd_livre*

cmd_casado → **if** *expr* **then** *cmd_casado* **else** *cmd_casado*
| **outros**

cmd_livre → **if** *expr* **then** *cmd*
| **if** *expr* **then** *cmd_casado* **else** *cmd_livre*

Exercícios

1. Considerando as GLCs a seguir:

   a) $S \rightarrow SS+ \mid SS* \mid a$

   b) $S \rightarrow 0S1 \mid 01$

   c) $S \rightarrow SS \mid S+S \mid S* \mid (S) \mid a \mid b$

   d) $S \rightarrow aSbS \mid bSaS \mid \varepsilon$

   e) $S \rightarrow S(S)S \mid \varepsilon$

Responda:

- Que linguagem essas gramáticas geram?
- Quais gramáticas são ambíguas? Justifique sua resposta apresentando duas árvores de derivação para uma mesma cadeia da linguagem

Exercícios

2.Construa GLCs não ambíguas para as linguagens:

a) Expressões aritméticas na notação pós-fixada

b) Listas associativas à esquerda de identificadores separados por vírgulas

c) Repita o item *b* considerando associatividade à direita

d) Expressões aritméticas de inteiros e identificadores com os 4 operadores binários (+, -, * e /)

- **Inteiro** é uma cadeia de 1 ou mais dígitos: ***digito****
- **Identificador** é uma cadeia formada por uma letra seguida por letras, dígitos ou "_" : ***letra* (*digito* | *letra* | _)***

e) Repita o item *d* com os operadores unários (+ e -)

Referências Bibliográficas

- Oliveira, G.M.B. material didático da disciplina Linguagens Formais e Autômatos, UFU

- Aho, A.V.; Lam, M.S.; Sethi, R.; Ullman, J.D. Compiladores: Princípios, técnicas e ferramentas, 2ª ed., Pearson, 2008

- Alexandre, E.S.M. Livro de Introdução a Compiladores, UFPB, 2014

- Aluisio, S. material da disciplina “Teoria da Computação e Compiladores”, ICMC/USP, 2011