

Construção de compiladores

Prof. Daniel Lucrédio

Departamento de Computação - UFSCar

1º semestre / 2015

Aula 3


Análise sintática

Introdução

Contexto

- Compilação = conversa (unilateral)
 - Linguagem humana tem:

Vocabulário + Gramática

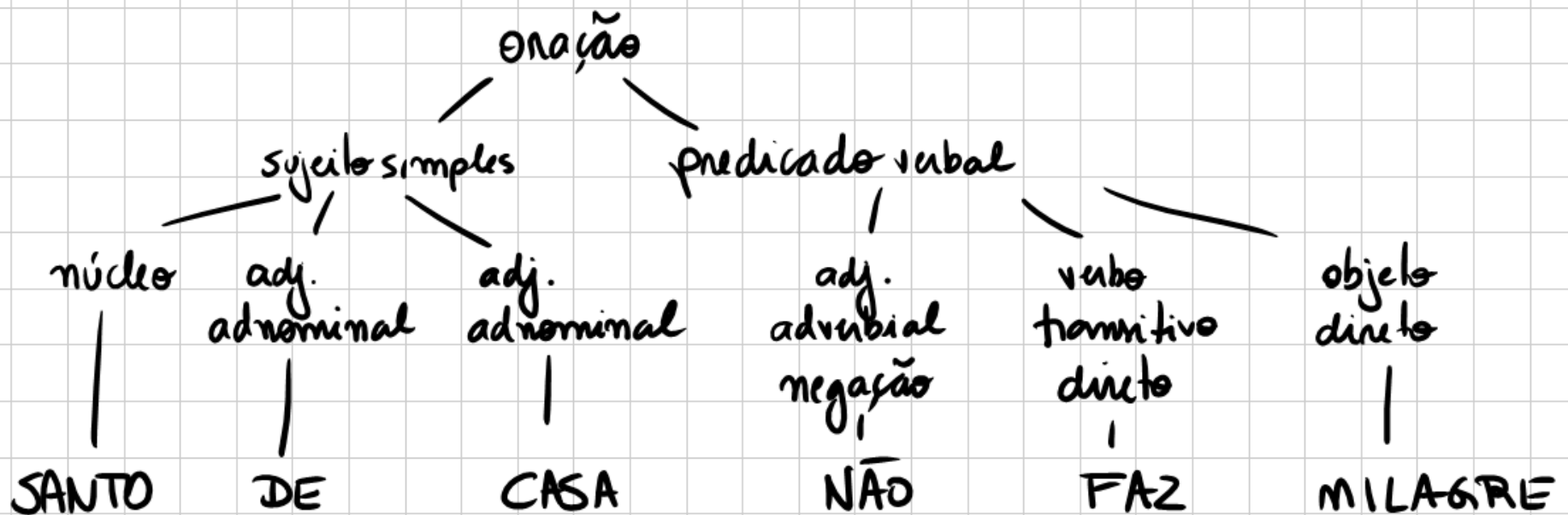
- 
- Nomes das coisas
 - É o que mais facilmente emerge na consciência dos locutores
 - Quem tem bebês sabe como eles aprendem a falar

- Ações
- Composição
- Conceitos complexos

- Seguir esse modelo natural é muito útil (como visto na aula anterior)

Objetivo

- Objetivo da análise sintática
 - Reconhecer a estrutura das frases
 - Todos nós sabemos fazer (ou deveríamos)

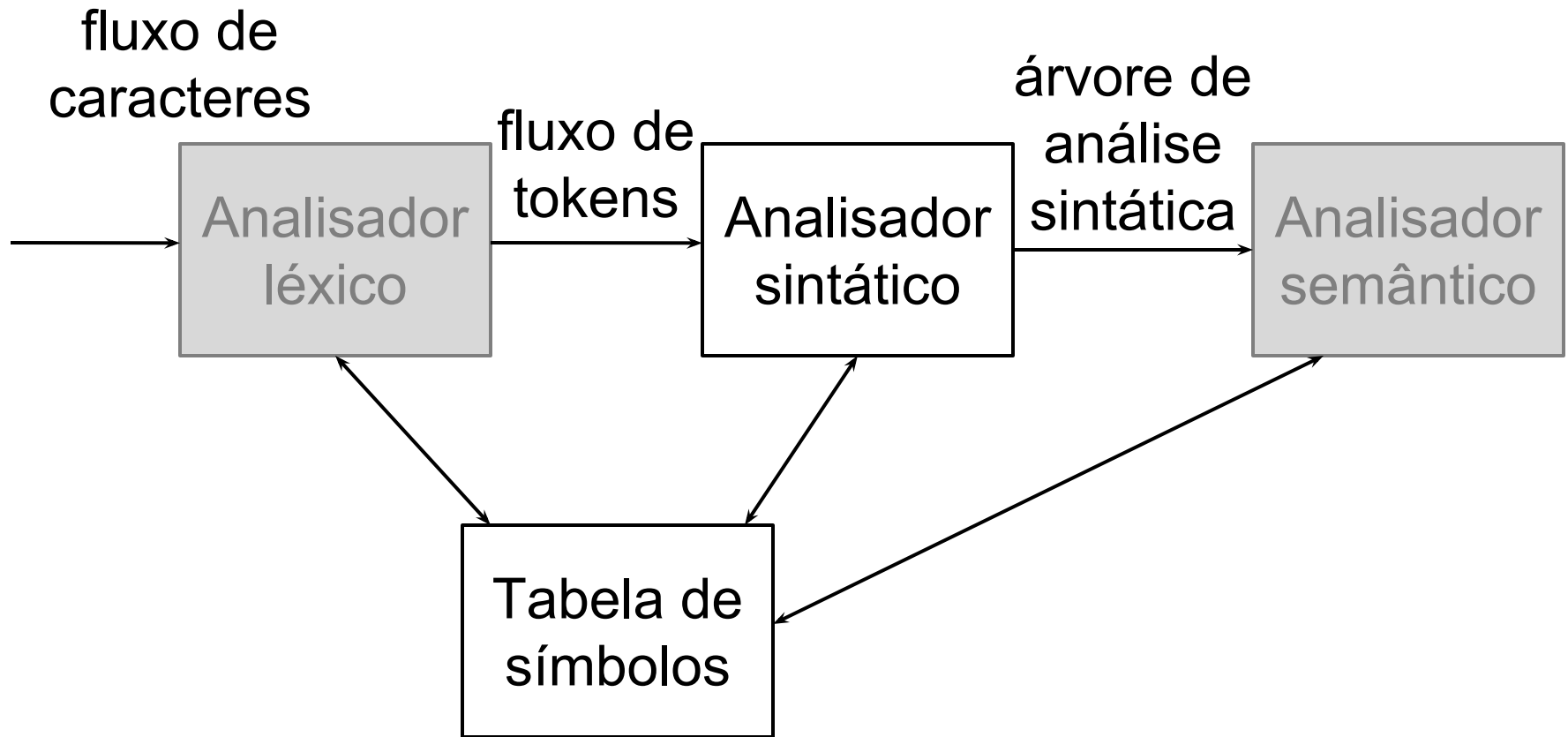


Em compiladores

- O objetivo é o mesmo
 - Frases = programas/modelos
 - Estrutura = linguagem de programação/modelagem
- Humanos são exímios processadores de linguagem
 - Está na nossa natureza
- Mas computadores são mais exatos (determinísticos)
- Eles precisam de um ALGORITMO que
 - Dado um fluxo de palavras
 - E uma definição da linguagem
 - Organize as palavras em uma estrutura coerente com a linguagem

Contexto

- Fluxo de palavras vem do analisador léxico

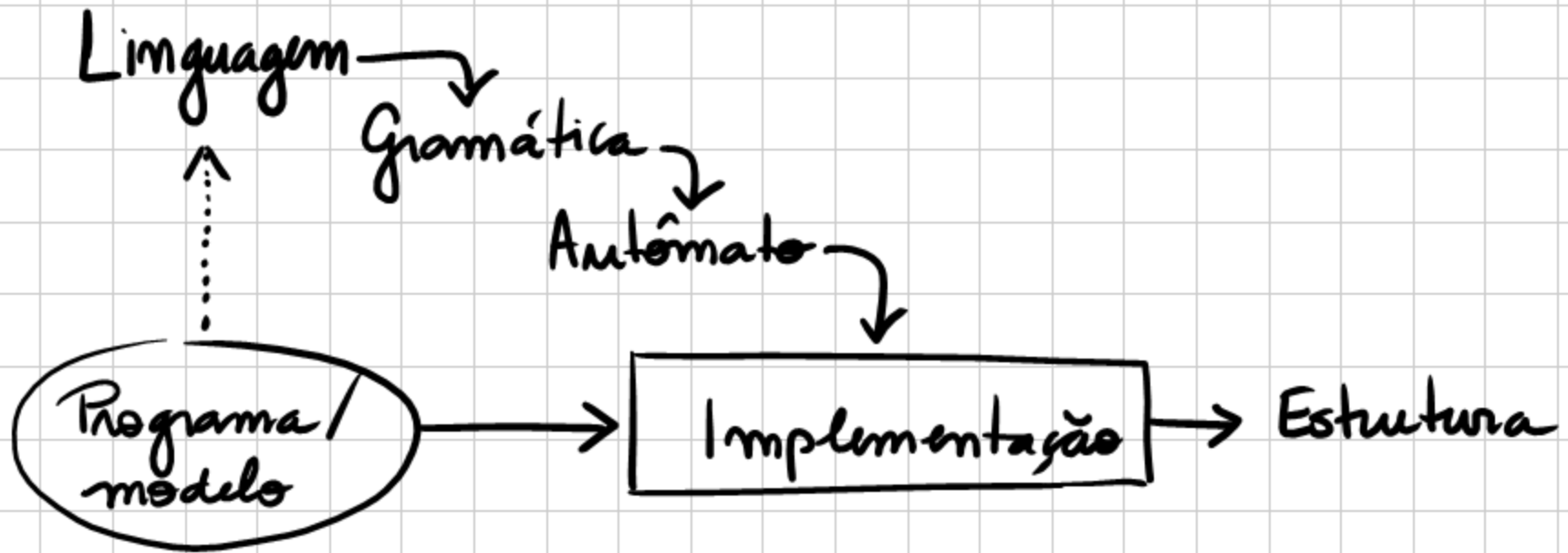


Obs: dentro da análise (front-end) do processo análise-síntese

Contexto

- Definição da linguagem
 - Deve ser uma definição precisa e formal
 - A ponto de ser utilizada por um algoritmo
 - Deve descrever a estrutura sintática da linguagem
- Novamente, a ciência da computação vem em auxílio
 - Em LFA vimos tal formalismo
 - Gramáticas
 - Com uma gramática podemos definir estruturas de cadeias de símbolos
 - E mais: para cada gramática, existe um modelo formal de máquina reconhecedora!
 - Autômatos
 - Autômatos significam IMPLEMENTAÇÃO!

Compiladores x LFA



Compiladores x LFA

- Quais são os tipos de gramáticas?
 - Hierarquia de Chomsky

Hierarquia	Gramáticas	Linguagens	Autômato mínimo
Tipo-0	Recursivamente Enumeráveis ou irrestritas	Recursivamente Enumeráveis	Máquinas de Turing
Tipo-1	Sensíveis ao contexto	Sensíveis ao contexto	MT com fita limitada
Tipo-2	Livres de contexto	Livres de contexto	Autômatos de pilha
Tipo-3	Regulares (Expressões regulares)	Regulares	Autômatos finitos

Compiladores x LFA

- Qual tipo de linguagem escolher?
- Escolha mais óbvia:
 - Tipo-0: Linguagens recursivamente enumeráveis
 - Gramáticas irrestritas
 - Máquina de Turing
- Vantagens:
 - Máquina de Turing é simples de implementar
 - Linguagens RE cobrem virtualmente tudo o que é necessário para a maioria das aplicações
- Desvantagens:
 - Gramáticas irrestritas são difíceis de conceber
 - E transformar em uma MT

Compiladores x LFA

- Próxima opção:
 - Tipo-1: Linguagens sensíveis ao contexto
 - Gramáticas sensíveis ao contexto
 - Máquina de Turing com fita limitada
- Mesmas vantagens e desvantagens do tipo-0
- No outro extremo:
 - Tipo-3: Linguagens regulares
 - Gramáticas regulares
 - Autômatos finitos
- Vantagens: simples implementação, gramáticas simples de conceber e converter em autômatos
- Desvantagens: não cobre as necessidades das linguagens
 - Não há recursividade / capacidade de “contar”

Compiladores x LFA

- Sobrou:
 - Tipo-2: Linguagens livres de contexto
 - Linguagens livres de contexto
 - Autômatos com uma pilha
- Vantagens:
 - Autômatos com pilha são de fácil implementação
 - Linguagens livres de contexto são (relativamente) fáceis de conceber e (relativamente) fáceis de converter em um autômato
- Desvantagens:
 - Algumas construções da maioria das linguagens exige sensibilidade ao contexto
 - Mas é possível contornar!!

Gramáticas livre de contexto

- Por estes motivos, as gramáticas livres de contexto são a melhor opção
 - Existem algoritmos para análise sintática baseados em seus princípios
 - Existem técnicas para projetar gramáticas livres de contexto
 - Existem técnicas para adicionar sensibilidade ao contexto
 - Em compiladores são chamadas de ANÁLISE SEMÂNTICA

Análise sintática

- Nesta parte (fundamental) da disciplina
 - Veremos alguns dos algoritmos para análise sintática
 - LL(k), LL(*), LR/LALR, GLR/Universal
 - Veremos que cada um tem suas características e necessidades específicas
 - Cada um tem vantagens e desvantagens
 - É importante conhecer a ESSÊNCIA que CARACTERIZA cada técnica, mesmo que não consiga implementá-las em detalhes
 - É importante saber os seus desdobramentos, para poder no futuro tomar decisões conscientes

Análise sintática

- Mas antes, vamos estudar o formalismo
 - O que é análise sintática baseada em gramáticas livres de contexto?
 - Tentaremos entender primeiro como fazer “de cabeça”
 - Depois começaremos a explorar as técnicas

Gramáticas livres de contexto

Gramáticas livres de contexto

$A \rightarrow 0A1$

$A \rightarrow B$

$B \rightarrow \#$

Regras de substituição ou produções

Lado esquerdo ou **cabeça**:
sempre um único símbolo. Esses
símbolos são chamados de
variáveis ou **não-terminais**

Lado direito ou **corpo**: uma
cadeia de símbolos. Podem ter
variáveis e outros símbolos,
chamados de **terminais**

Uma das variáveis é designada como a variável ou símbolo inicial.
É a variável que aparece do lado esquerdo da primeira regra.
(Neste exemplo, **A** é o símbolo inicial)

Gramáticas livres de contexto

- Definição formal
- $G = (V, T, P, S)$
 - V = conjunto de variáveis
 - T = conjunto de terminais
 - P = conjunto de produções
 - S = símbolo inicial
- Ex:
 - $G_{\text{palíndromos}} = (\{P\}, \{0, 1\}, A, P)$
 - $A = \{$
 - $P \rightarrow \varepsilon$
 - $P \rightarrow 0$
 - $P \rightarrow 1$
 - $P \rightarrow 0P0$
 - $P \rightarrow 1P1$
 - $\}$

Gramáticas livres de contexto

- Em compiladores, os terminais são os tokens
 - Mais especificamente, os TIPOS dos tokens
 - <id, “var1”>
 - id é usado na análise sintática
 - “var1” é ignorado
- Os não-terminais definem normalmente as construções da linguagem
 - De alto nível (programa, função, bloco)
 - De baixo nível (comandos, expressões)

Gramáticas livres de contexto

Programa \rightarrow ListaComandos

ListaComandos \rightarrow Comando ListaComandos

ListaComandos \rightarrow Comando

Comando \rightarrow ComandoIf

Comando \rightarrow ComandoAtrib

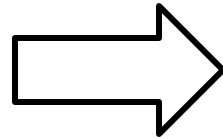
ComandoIf \rightarrow TK_IF Expr TK_THEN Comando

ComandoIf \rightarrow TK_IF Expr TK_THEN Comando
ELSE Comando

ComandoAtrib \rightarrow id TK_ATRIB Expr

...

Gramáticas livres de contexto

$$A \rightarrow 0A1$$
$$A \rightarrow B$$
$$B \rightarrow \#$$

$$A \rightarrow 0A1 \mid B$$
$$B \rightarrow \#$$

Sempre que houver mais de uma produção para uma mesma variável, podemos agrupá-las com o símbolo “|”.

Gramáticas livres de contexto

Programa \rightarrow ListaComandos

ListaComandos \rightarrow Comando ListaComandos
 | Comando

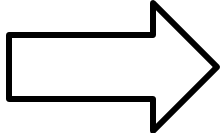
Comando \rightarrow ComandoIf | ComandoAtrib

ComandoIf \rightarrow TK_IF Expr TK_THEN Comando
 |
 TK_IF Expr TK_THEN
 Comando ELSE Comando

ComandoAtrib \rightarrow id TK_ATRIB Expr

...

Gramáticas livres de contexto

$A \rightarrow 0B1$
 $B \rightarrow \# \mid \%$  $A \rightarrow 0 (\# \mid \%) 1$

Quando uma regra só é utilizada dentro de outra, é possível criar uma subregra anônima, utilizando parênteses

Gramáticas livres de contexto

Programa \rightarrow ListaComandos

ListaComandos \rightarrow Comando ListaComandos
 | Comando

Comando \rightarrow ComandoIf | (id TK_ATRIB
 Expr)

ComandoIf \rightarrow TK_IF Expr TK_THEN Comando
 |
 TK_IF Expr TK_THEN
 Comando ELSE Comando

...

Gramáticas livres de contexto

- Como uma gramática descreve uma linguagem?
- Duas formas:
 - Inferência recursiva
 - Derivação
- Ex: Gramática para expressões aritméticas
 - $V = \{E, I\}$
 - $T = \{+, *, (,), a, b, 0, 1\}$
 - $P =$ conjunto de regras ao lado
 - $S = E$

$$\begin{array}{lcl} E & \rightarrow & I \\ & | & E + E \\ & | & E * E \\ & | & (E) \\ I & \rightarrow & a \\ & | & b \\ & | & Ia \\ & | & Ib \\ & | & I0 \\ & | & I1 \end{array}$$

Gramáticas livres de contexto

- Inferência recursiva
 - Dada uma cadeia (conjunto de símbolos terminais)
 - Do corpo para a cabeça
- Ex: $a^*(a+b00)$
 - $a^*(a+b00) \Leftarrow a^*(a+100) \Leftarrow a^*(a+10) \Leftarrow a^*(a+1) \Leftarrow a^*(a+E) \Leftarrow a^*(1+E) \Leftarrow a^*(E+E) \Leftarrow a^*(E) \Leftarrow a^*E \Leftarrow 1^*E \Leftarrow E^*E \Leftarrow E$

Gramáticas livres de contexto

- Derivação
 - Dada uma cadeia (conjunto de símbolos terminais)
 - Da cabeça para o corpo
- Ex: $a^*(a+b00)$
 - $E \Rightarrow E^*E \Rightarrow I^*E \Rightarrow a^*E \Rightarrow a^*(E) \Rightarrow a^*(E+E) \Rightarrow a^*(I+E) \Rightarrow a^*(a+E) \Rightarrow a^*(a+I) \Rightarrow a^*(a+I0) \Rightarrow a^*(a+I00) \Rightarrow a^*(a+b00)$
- Símbolo de derivação: \Rightarrow *
- Derivação em múltiplas etapas: \Rightarrow
 - $E \Rightarrow a^*(E^*)$
 - $a^*(E+E) \Rightarrow a^*(a+I00)$
 - $E \Rightarrow a^*(a+b00)$

Gramáticas livres de contexto

- Derivações mais à esquerda
 - Sempre substituir a variável mais à esquerda
 - Notação: \Rightarrow_{lm}^* , \Rightarrow_{lm}
- Derivações mais à direita
 - Sempre substituir a variável mais à direita
 - Notação: \Rightarrow_{rm}^* , \Rightarrow_{rm}

Árvores de análise sintática

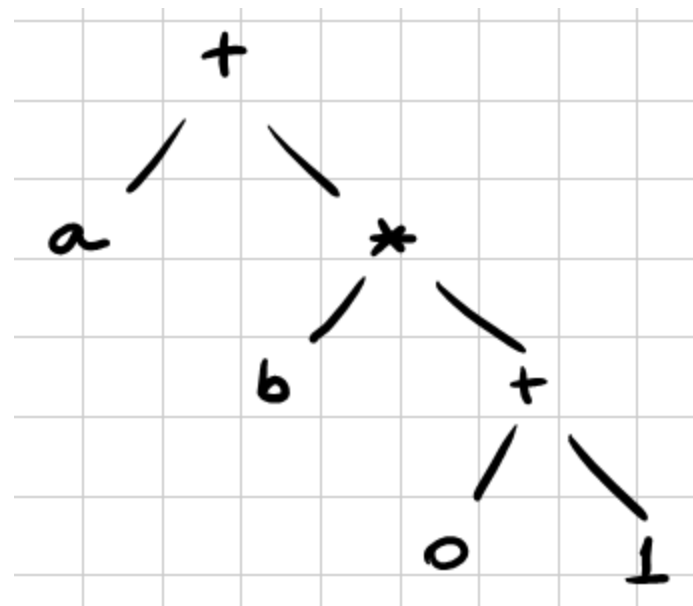
Árvores de análise sintática

- Representação visual para derivações
 - Em formato de árvore
- Mostra claramente como os símbolos de uma cadeia de terminais estão agrupados em subcadeias
- Permitem analisar alguns aspectos da linguagem e ver o processo de derivação / inferência recursiva
- Ex: $a+b*(0+1)$

$$\begin{array}{ccccccc} E & \rightarrow & I & & & & \\ & | & E & + & E & & \\ & | & E & * & E & & \\ & | & (E) & & & & \\ I & \rightarrow & a & | & b & | & 0 & | & 1 \end{array}$$

Árvores de sintaxe abstrata

- As árvores de análise sintática são completas
 - Representam completamente a derivação
 - São também conhecidas como árvores de derivação ou *parse trees*
- Mas nem sempre é necessário utilizar toda a informação
- Ex: $a+b*(0+1)$



Árvores de sintaxe abstrata

- É uma árvore simplificada
- Contém a informação necessária para o compilador, e nada mais
 - Em contraste com a sintaxe concreta
- Omite (abstrai) detalhes
 - Pois muitas vezes as regras gramaticais incluem não-terminais somente como mecanismos auxiliares para:
 - Repetição
 - Opcionalidade

Árvores de sintaxe abstrata

- **Ex:**

$\text{Comando} \rightarrow \text{ComandoIf} \mid (\text{id TK_ATRIB Expr})$

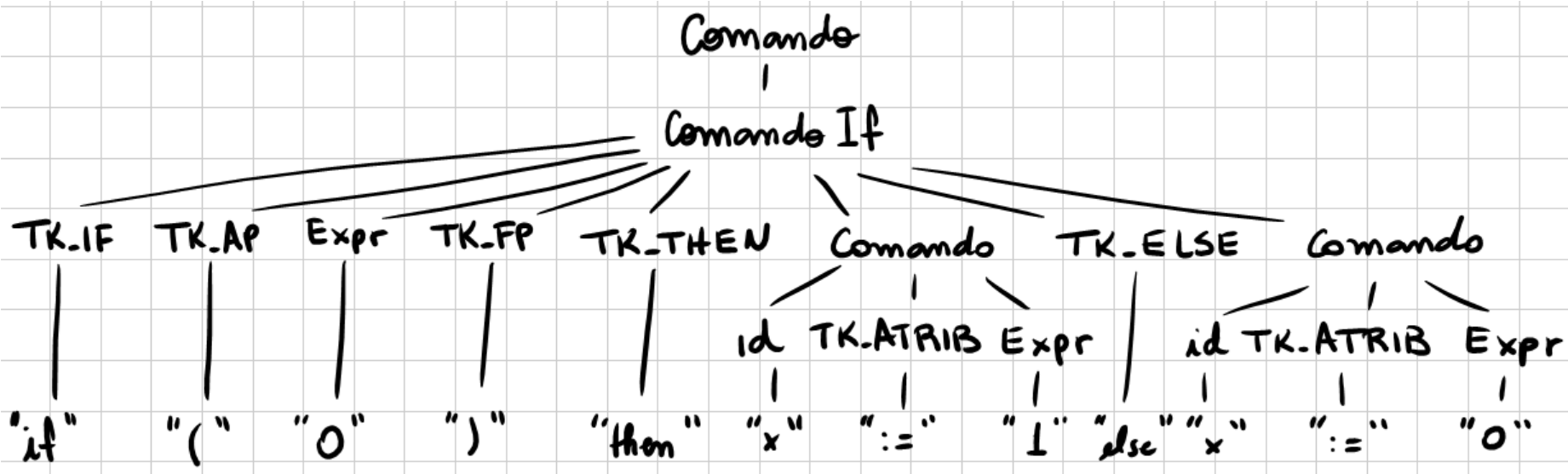
$\text{ComandoIf} \rightarrow \text{TK_IF TK_AP Expr TK_FP TK_THEN Comando} \mid$
 $\text{TK_IF TK_AP Expr TK_FP TK_THEN Comando ELSE Comando}$

$\text{Expr} \rightarrow \text{TK_0} \mid \text{TK_1}$

Árvores de sintaxe abstrata

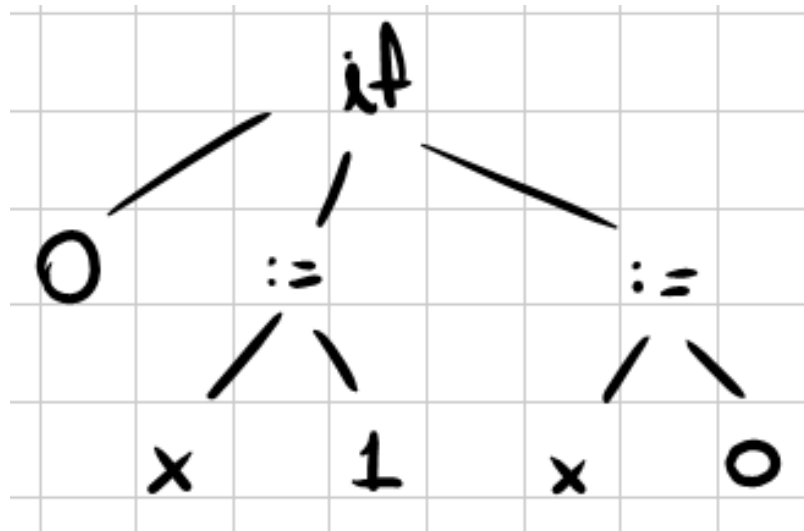
- Fazendo a árvore de análise sintática (concreta) da cadeia:

if(0) then x := 1 else x:=0



Árvores de sintaxe abstrata

- Na verdade, o que queremos representar é:



- Pois é isso o que importa para o compilador
 - O resto é “detalhe”

Árvores de sintaxe abstrata

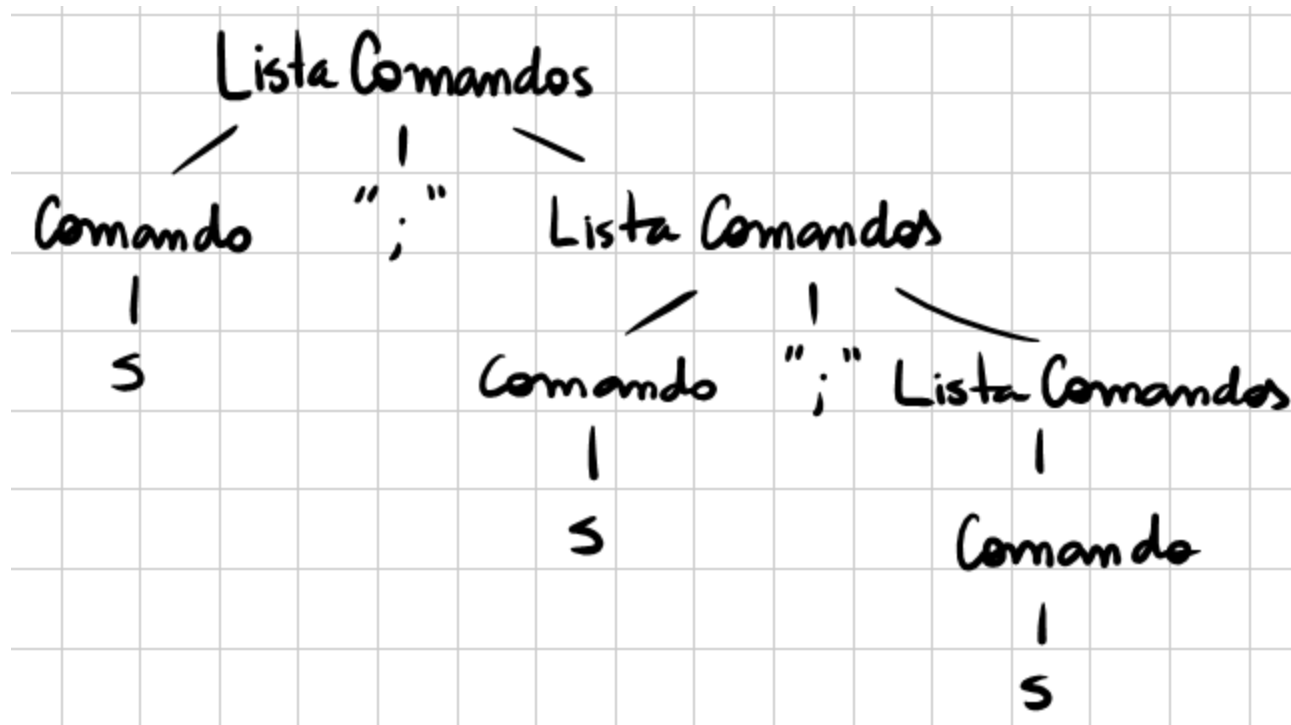
- Outro exemplo:
 - Sequência de declarações separadas por ponto-e-vírgula:

$$\begin{aligned} \text{ListaComandos} &\rightarrow \text{Comando} \text{ ';' } \text{ListaComandos} \\ &\quad | \text{Comando} \end{aligned}$$
$$\text{Comando} \rightarrow c$$

Árvores de sintaxe abstrata

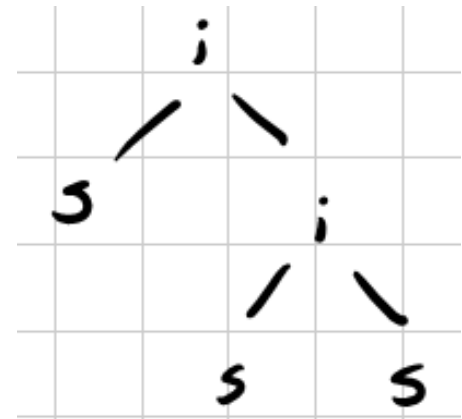
- Fazendo a árvore de análise sintática da cadeia:

S ; S ; S

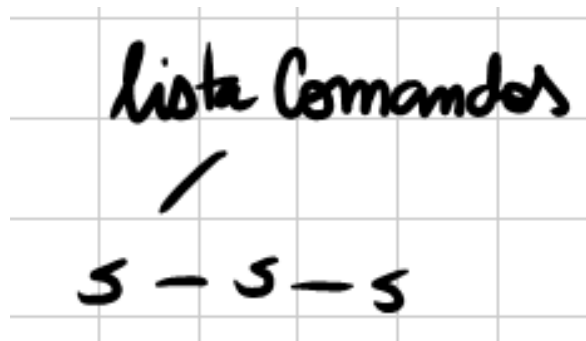


Árvores de sintaxe abstrata

- Na verdade, poderíamos representar assim:



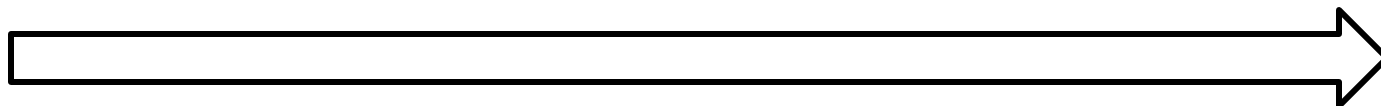
- Ou, melhor ainda:



Árvores de sintaxe abstrata

- Na verdade, a sintaxe abstrata pode ser
 - QUALQUER ESTRUTURA DE DADOS
 - Como uma lista, no exemplo anterior
- Pode ser, por exemplo, um grafo direcionado
 - Apesar de que na prática, quase sempre é uma árvore
 - Em alguns contextos, é chamada de METAMODELO

Árvore de
análise
sintática



Mais fácil de construir
Mais difícil de usar

Mais difícil de construir
Mais fácil de usar

***Estrutura (árvore)
de sintaxe abstrata***

Exemplo

- Gramática simplificada da linguagem ALGUMA

```
VARIAVEL : ('a'..'z'|'A'..'Z')
          ('a'..'z'|'A'..'Z'|'0'..'9')*;
TIPO_VAR : 'INTEIRO' | 'REAL';
programa : ':' 'DECLARACOES' listaDeclaracoes ':'
          'ALGORITMO' listaComandos;
listaDeclaracoes : declaracao listaDeclaracoes |
                  declaracao;
declaracao : VARIAVEL ':' TIPO_VAR;
listaComandos : comando listaComandos | comando;
comando : comandoEntrada | comandoSaida;
comandoEntrada : 'LER' VARIAVEL;
comandoSaida : 'IMPRIMIR' VARIAVEL;
```


Exemplo

:DECLARACOES

argumento:INTEIRO

fatorial:INTEIRO

:ALGORITMO

% Calcula o fatorial de um número
inteiro

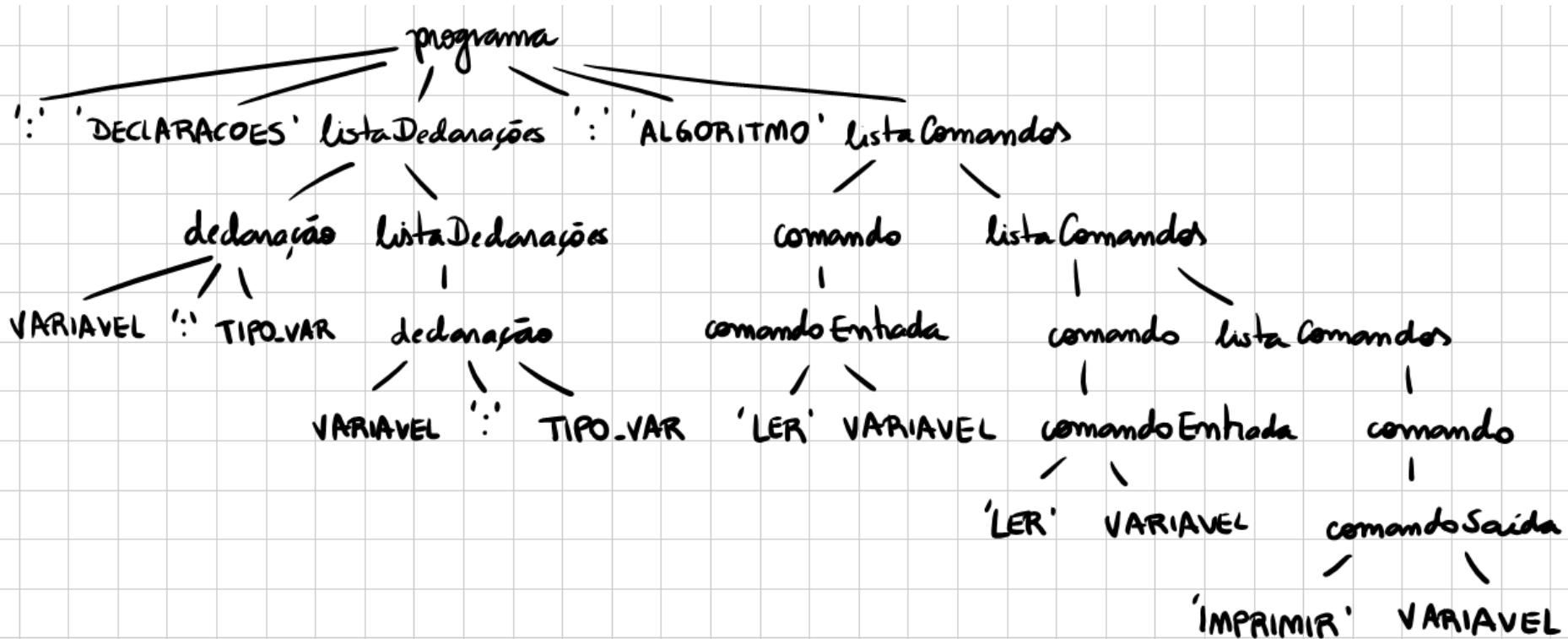
LER argumento

LER fatorial

IMPRIMIR fatorial

Exemplo

- Árvore de análise sintática (sintaxe concreta)



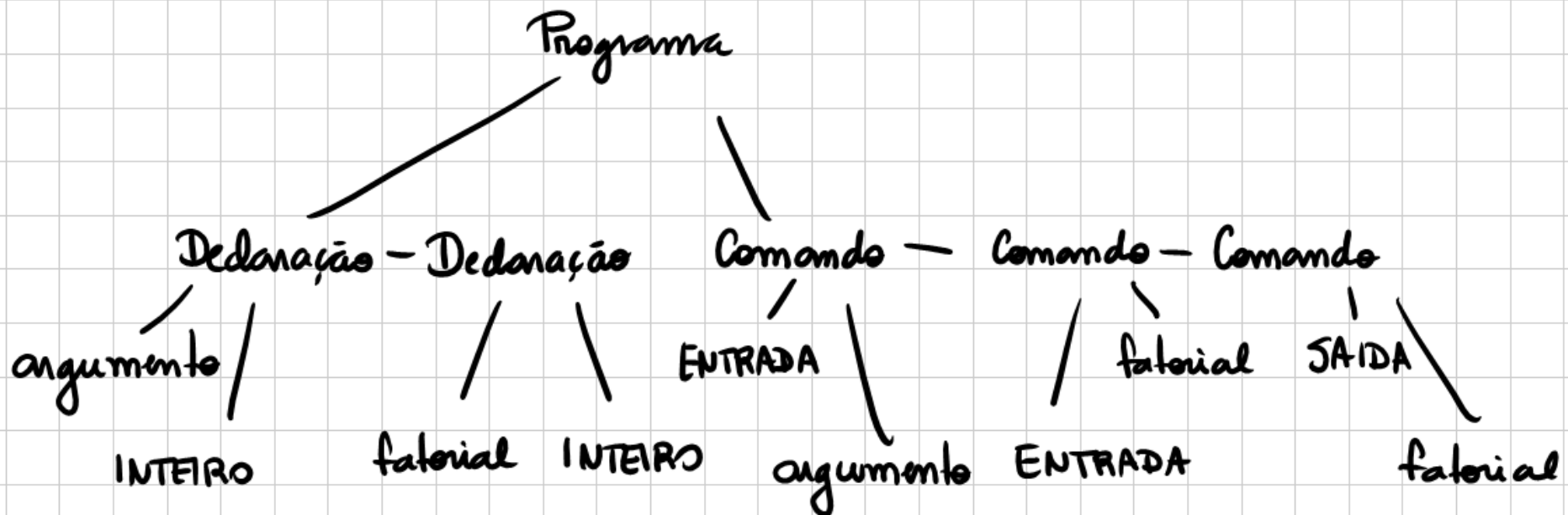
Exemplo

- Estrutura de dados de sintaxe abstrata:

```
class Programa {
    Declaracao[] declaracoes;
    Comando[] comandos;
}
class Declaracao {
    String nomeVar;
    TipoVar tipo;
}
class Comando {
    TipoComando tipo;
    String variavel;
}
enum TipoVar { INTEIRO, REAL }
enum TipoComando { ENTRADA, SAIDA }
```

Exemplo

- Árvore de sintaxe abstrata



Sintaxe abstrata vs concreta

- Resumindo
- A sintaxe concreta é necessária
 - É através dela que o compilador analisa a estrutura
 - Essencial para repetições, condicionais, etc
 - Mas apenas num primeiro momento
- Uma vez que a análise sintática é concluída
 - Podemos “jogar fora” a sintaxe concreta
 - Mas precisamos de outra estrutura de dados
 - Mais limpa, sem tantos detalhes, mais fácil de trabalhar
 - Trata-se da sintaxe abstrata
- Podemos definir qualquer estrutura de dados para a sintaxe abstrata
 - Normalmente, é uma árvore, sem tantos detalhes

Ambiguidade

Ambiguidade

- Considere as seguintes frases (verídicas), extraídas de um sistema de pedidos de um almoxarifado de um banco
 - “Armário para funcionário de aço”
 - “Cadeira para gerente sem braços”
- Quem é de aço? O armário ou funcionário?
 - “(Armário para funcionário) de aço”
- Quem não tem braços? A cadeira ou o gerente?
 - “(Cadeira para gerente) sem braços”
- O problema é a ambiguidade

Ambiguidade

- Outro exemplo, gramática à direita
 - Encontre derivações mais à esquerda para a cadeia $a + b^*a$
- Respostas:
- $E \Rightarrow E + E \Rightarrow I + E \Rightarrow a + E \Rightarrow a + E * E \Rightarrow a + I * E \Rightarrow a + b * E \Rightarrow a + b * I \Rightarrow a + b * a$
- $E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow I + E * E \Rightarrow a + E * E \Rightarrow a + I * E \Rightarrow a + b * E \Rightarrow a + b * I \Rightarrow a + b * a$

$$\begin{aligned}E &\rightarrow I \\E &\rightarrow E + E \\E &\rightarrow E * E \\E &\rightarrow (E) \\I &\rightarrow a \\I &\rightarrow b\end{aligned}$$

Ambiguidade

- A diferença entre as árvores e as derivações mais à esquerda é significativa
 - Dependendo de qual árvore usar, o gerente pode ficar sem braços
 - Cadeira para ____
 - ____ sem braços
 - Dependendo de qual derivação à esquerda usar, a adição pode ocorrer antes da multiplicação
 - $a + \underline{\quad}$
 - $\underline{\quad} * a$

Ambiguidade

- Gramáticas são usadas para dar estrutura para programas, documentos, etc
 - Supõe-se que essa estrutura é única
 - Caso não seja, podem ocorrer problemas
- Nem toda gramática fornece estruturas únicas
 - Ambiguidade
 - Algumas vezes é possível reprojeter a gramática para eliminar a ambiguidade
 - Em outras vezes, isso é impossível
 - Ou seja, existem linguagens “inerentemente ambíguas”
 - Isto é: toda gramática para esta linguagem será fatalmente ambígua

Ambiguidade

- O que caracteriza ambiguidade
 - A existência de duas ou mais árvores de análise sintática para pelo menos uma cadeia da linguagem
- Formalmente:
 - Uma CFG $G = (V, T, P, S)$ é ambígua se existe pelo menos uma cadeia w em T^* para o qual podemos encontrar duas árvores de análise sintática diferentes, cada qual com uma raiz identificada como S e um resultado w .
 - Se TODAS as cadeias tiverem no máximo uma árvore de análise sintática, a gramática é não-ambígua

Ambiguidade

- Também pode-se pensar na ambiguidade em termos de derivações
- Teorema: Para cada gramática $G = (V, T, P, S)$ e cadeia w em T^* , w tem duas árvores de análise sintática distintas se e somente se w tem duas derivações mais à esquerda distintas a partir de S
 - Corolário: Se para uma gramática $G = (V, T, P, S)$, e uma cadeia w em T^* , for possível encontrar duas derivações mais à esquerda distintas, G é ambígua
- O mesmo vale para derivações mais à direita

Ambiguidade

- Eliminando a ambiguidade
 - Problemas
- Primeiro: saber se uma gramática é ambígua é um problema indecidível, ou seja, descobrir que uma gramática é ambígua depende de análise, exemplos e um pouco de sorte!
- Segundo: existem linguagens inerentemente ambíguas, ou seja, TODA CFG será ambígua
- Terceiro: mesmo para uma linguagem que não é inerentemente ambígua, não existe um algoritmo para remover a ambiguidade

Eliminando ambiguidade

- Existem algumas técnicas bem conhecidas, para alguns casos de ambiguidade
- Primeira técnica: forçar a precedência de terminais introduzindo novas regras
- Segunda técnica: modificar ligeiramente a linguagem
- Terceira técnica: “ajustar” diretamente o analisador

Eliminando ambiguidade

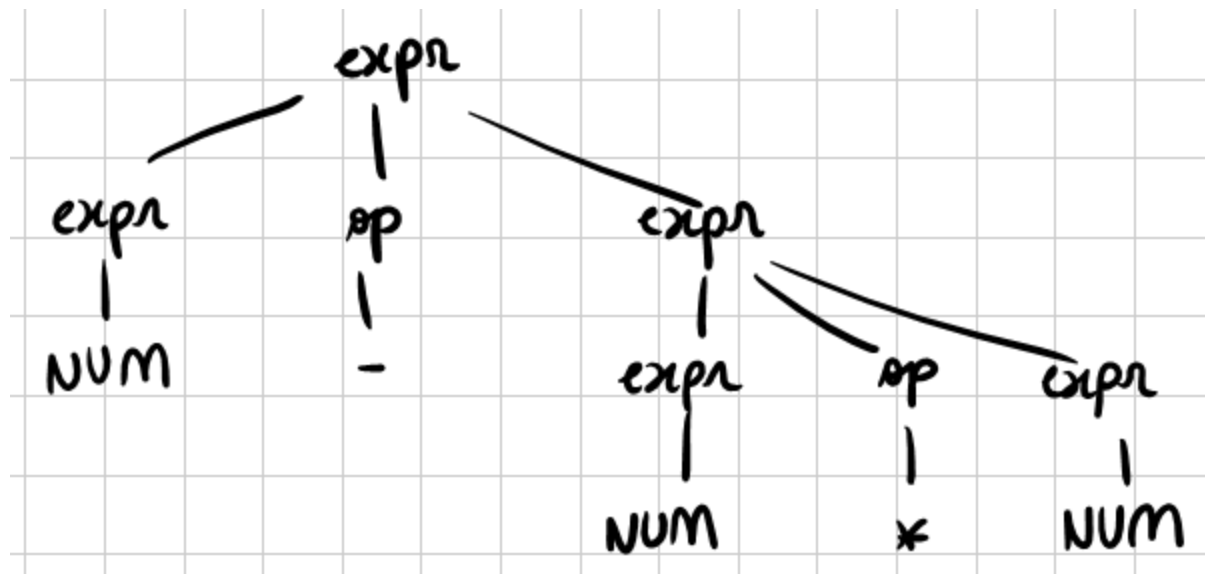
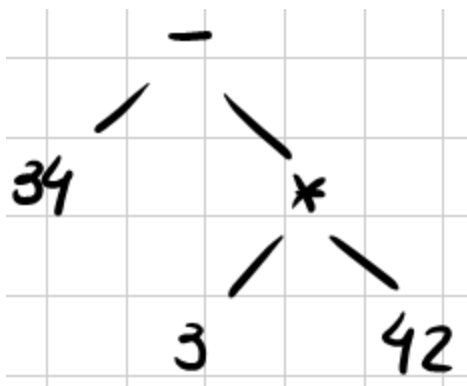
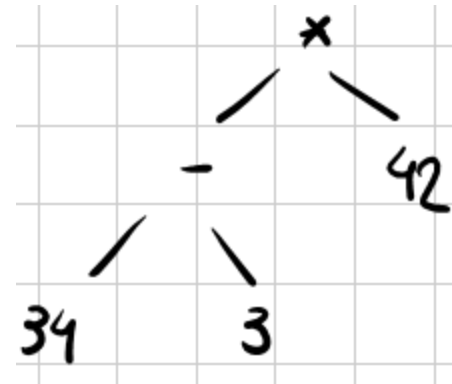
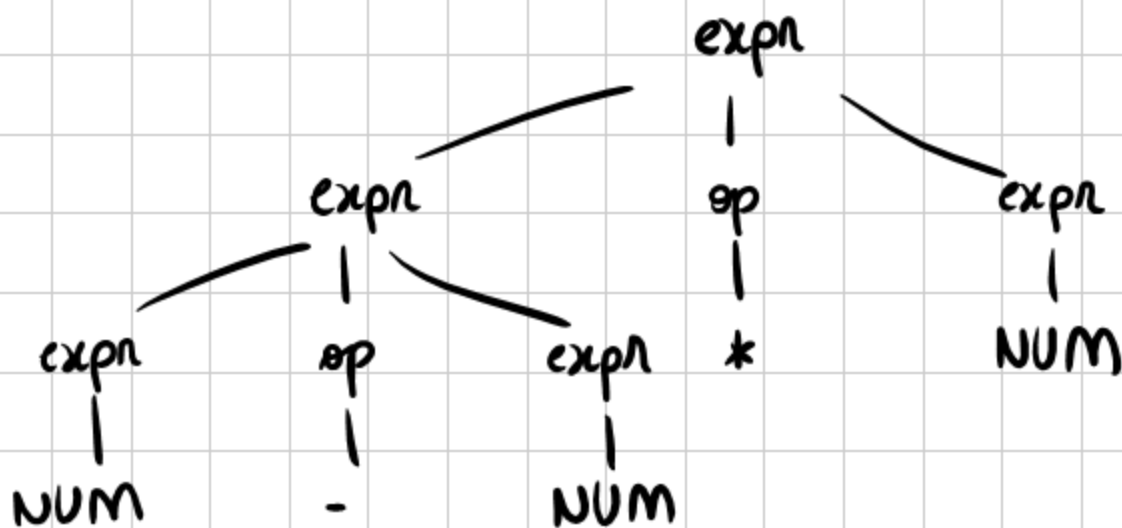
- Exemplo clássico: expressões aritméticas
 - Na prática é o único exemplo em que é possível remover a ambiguidade “facilmente”

$\text{expr} \rightarrow \text{expr op expr} \mid ' (' \text{expr} ') ' \mid \text{NUM}$

$\text{op} \rightarrow + \mid - \mid *$

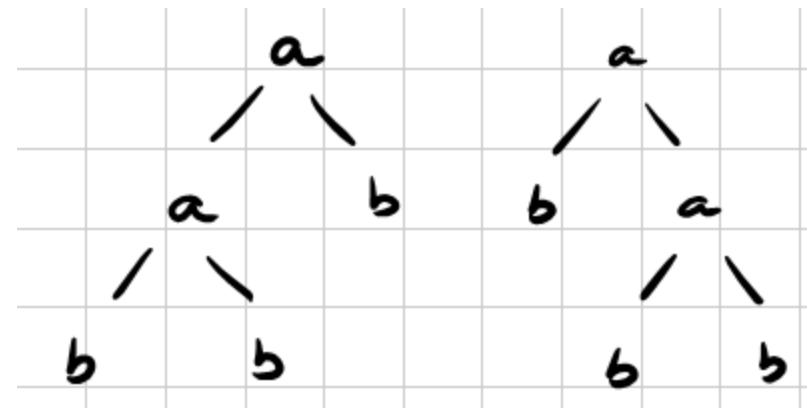
- É fácil demonstrar que existe mais de uma árvore de análise sintática para a cadeia $34 - 3 * 42$
 - E que também irão resultar em sintaxes abstratas diferentes

Eliminando ambiguidade



Eliminando ambiguidade

- Existe um ponto de ambiguidade
 - Associatividade
- Causado por uma recursividade dupla (à direita e à esquerda)
 - $S \rightarrow S \text{ 'qualquer terminal' } S \mid \dots$
- Exemplo mais genérico
 - $S \rightarrow SaS \mid b$
 - Cadeia = babab



Eliminando ambiguidade

- Nestes casos, é preciso remover a recursividade de um dos lados
- Para “forçar” a associatividade à esquerda

$$S \rightarrow SaS \mid b \quad \Rightarrow \quad S \rightarrow Sab \mid b$$

- Para “forçar” a associatividade à direita

$$S \rightarrow SaS \mid b \quad \Rightarrow \quad S \rightarrow baS \mid b$$

Eliminando ambiguidade

- No exemplo das expressões

$\text{expr} \rightarrow \text{expr op expr} \mid ' (' \text{ expr } ') ' \mid \text{NUM}$

$\text{op} \rightarrow + \mid - \mid *$

- Forçando associatividade à esquerda:

$\text{expr} \rightarrow \text{expr op} (' (' \text{ expr } ') ' \mid \text{NUM}) \mid ' (' \text{ expr } ') ' \mid \text{NUM}$

$\text{op} \rightarrow + \mid - \mid *$

- Para melhorar a legibilidade, vamos inserir uma outra regra:

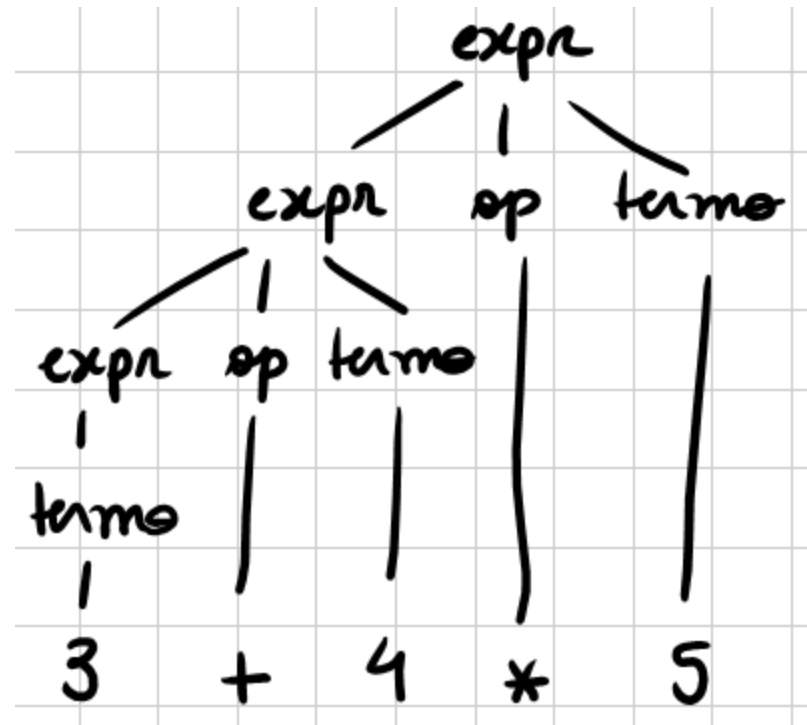
$\text{expr} \rightarrow \text{expr op termo} \mid \text{termo}$

$\text{termo} \rightarrow ' (' \text{ expr } ') ' \mid \text{NUM}$

$\text{op} \rightarrow + \mid - \mid *$

Eliminando ambiguidade

- Já removemos a ambiguidade!
- Mas tente criar mais de uma árvore para:
 - $3 + 4 + 5$
 - $3 * 4 + 5$
 - $3 + 4 * 5$
- O que há de errado com o último exemplo?
 - Uma criança aprendendo matemática não veria o erro
 - Resposta: matemáticos decretaram uma ordem “certa” para as operações



Eliminando ambiguidade

- Ao remover a ambiguidade
 - Eliminamos a flexibilidade de diferentes precedências
 - Antes, era possível “escolher” qual operador tinha maior precedência
 - A gramática era flexível
 - Agora, todos os operadores têm a mesma precedência
 - Ou seja, vale a ordem em que aparecem na cadeia

Eliminando ambiguidade

- Precisamos portanto definir a precedência
- Na nossa convenção matemática, * tem maior precedência sobre + e –
- Resolvemos isso criando diferentes classes de operadores e uma cascata de regras

`expr` \rightarrow `expr op1 termo` | `termo`

`termo` \rightarrow `termo op2 fator` | `fator`

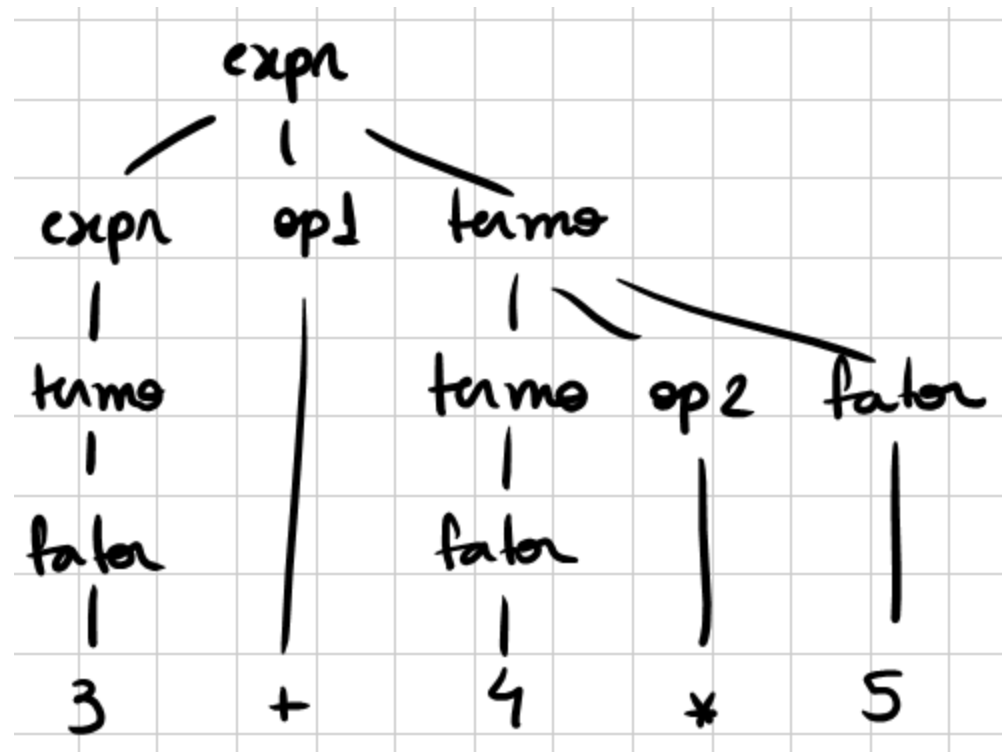
`fator` \rightarrow `' (' expr ') '` | `NUM`

`op1` \rightarrow `+` | `-`

`op2` \rightarrow `*`

Eliminando ambiguidade

- Testando agora:
 - $3 + 4 + 5$
 - $3 * 4 + 5$
 - $3 + 4 * 5$



Associatividade e precedência

- Regra genérica
- $S \rightarrow S \ a \ T \mid S \ b \ T \mid S \ c \ T \mid S \ d \ T \mid T$
- $T \rightarrow x$
- Associatividade = a,b à esquerda e c,d à direita
- Precedência = $a < b < c < d$
 - Temos quatro classes de precedência, precisamos de quatro regras distintas
 - Para cada uma, inserimos a recursão conforme a associatividade (esquerda ou direita)
- $S \rightarrow S \ a \ S1 \mid S1$
- $S1 \rightarrow S1 \ b \ S2 \mid S2$
- $S2 \rightarrow S3 \ c \ S2 \mid S3$
- $S3 \rightarrow T \ d \ S3 \mid T$
- $T \rightarrow x$

Associatividade e precedência

- Exercício
- Defina uma gramática para expressões aritméticas, com os operadores: $+$, $-$, $*$, $/$, $\%$ (módulo) e $^$ (potência)
- Precedência:
 - $+, - < *, /, \% < ^$
- Associatividade
 - Todos são associativos à esquerda, exceto o operador de potência
- As expressões não utilizam parêntesis

Associatividade e precedência

- Primeiro passo: gramática ambígua

$\text{expr} \rightarrow \text{expr op expr} \mid \text{NUM}$

$\text{op} \rightarrow + \mid - \mid * \mid / \mid \% \mid ^$

- Segundo passo: separando os operadores em três classes de precedência

$\text{op1} \rightarrow + \mid -$

$\text{op2} \rightarrow * \mid / \mid \%$

$\text{op3} \rightarrow ^$

Associatividade e precedência

- Terceiro passo: forçando associatividade e precedência

$\text{expr} \rightarrow \text{expr op1 termo} \mid \text{termo}$

$\text{termo} \rightarrow \text{termo op2 fator} \mid \text{fator}$

$\text{fator} \rightarrow \text{NUM op3 fator} \mid \text{NUM}$

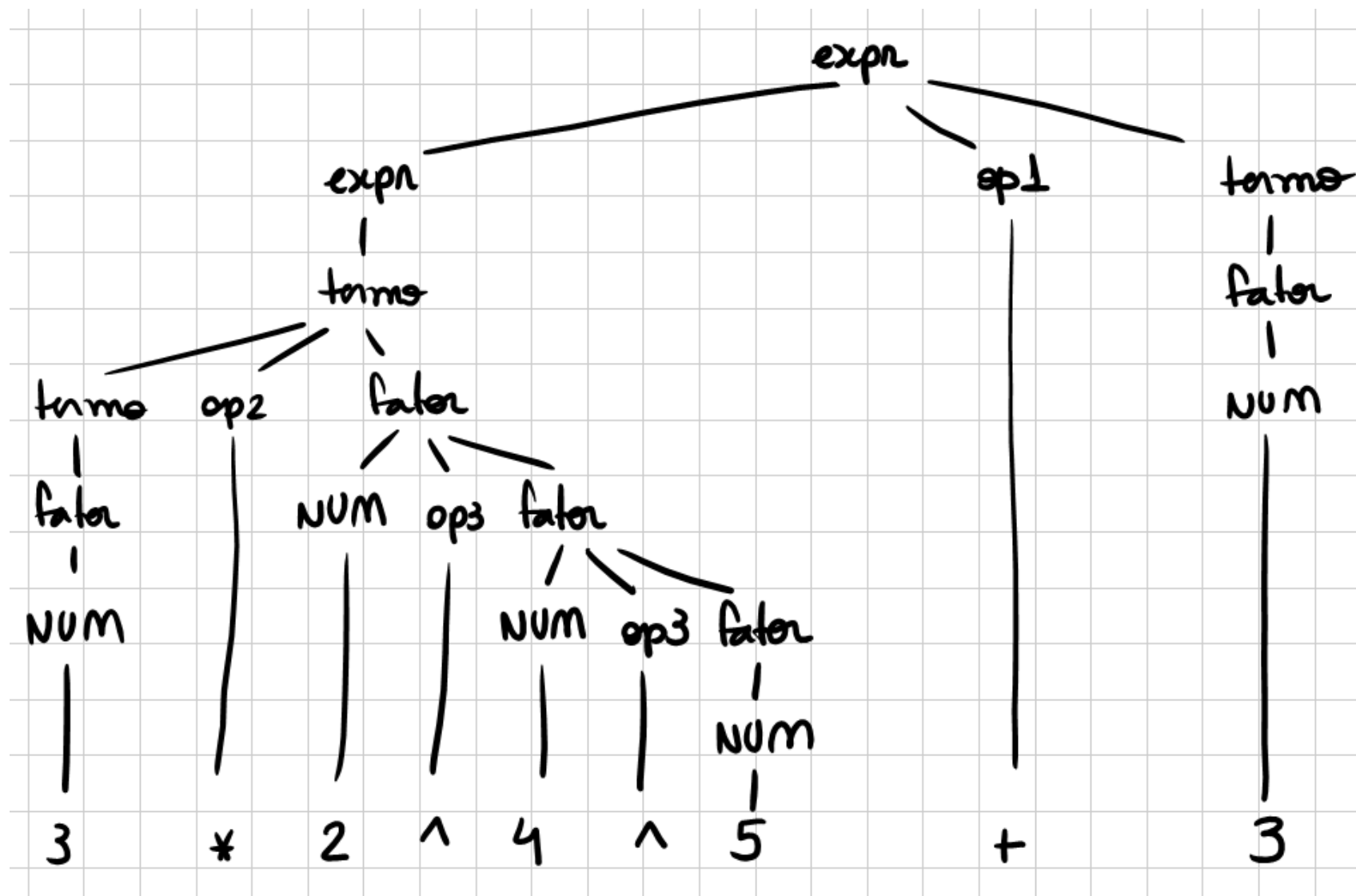
$\text{op1} \rightarrow + \mid -$

$\text{op2} \rightarrow * \mid / \mid \%$

$\text{op3} \rightarrow ^$

- Testando: $3 * 2 ^ 4 ^ 5 + 3$

Associatividade e precedência



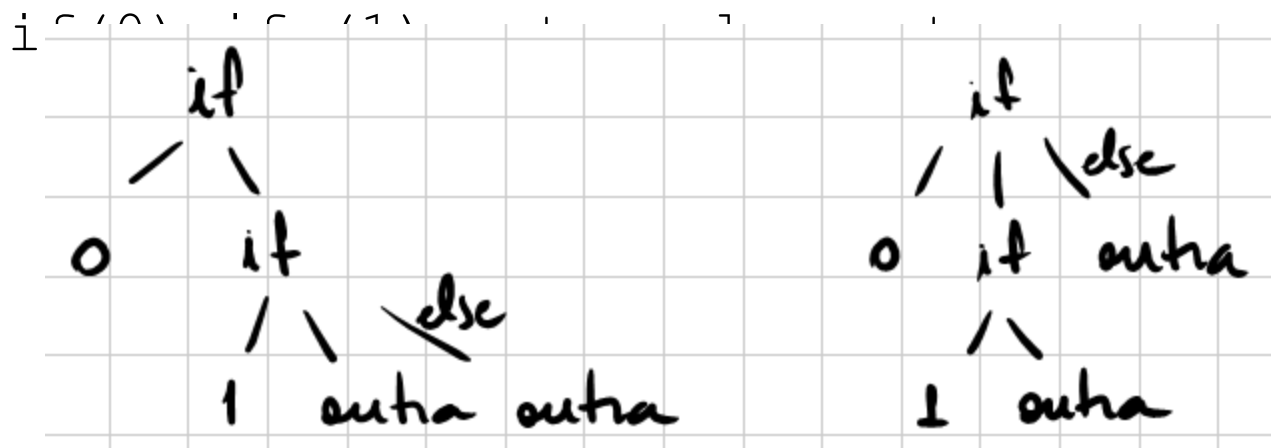
Eliminando ambiguidade

- Segunda técnica: modificando ligeiramente a linguagem

declaração \rightarrow if-decl | outra

$$\text{if-decl} \rightarrow \text{if (exp) declaração} \mid \text{if (exp) declaração else declaração}$$
$$\text{exp} \rightarrow 0 \mid 1$$

- Verifique que há duas árvore para a seguinte cadeia



Eliminando ambiguidade

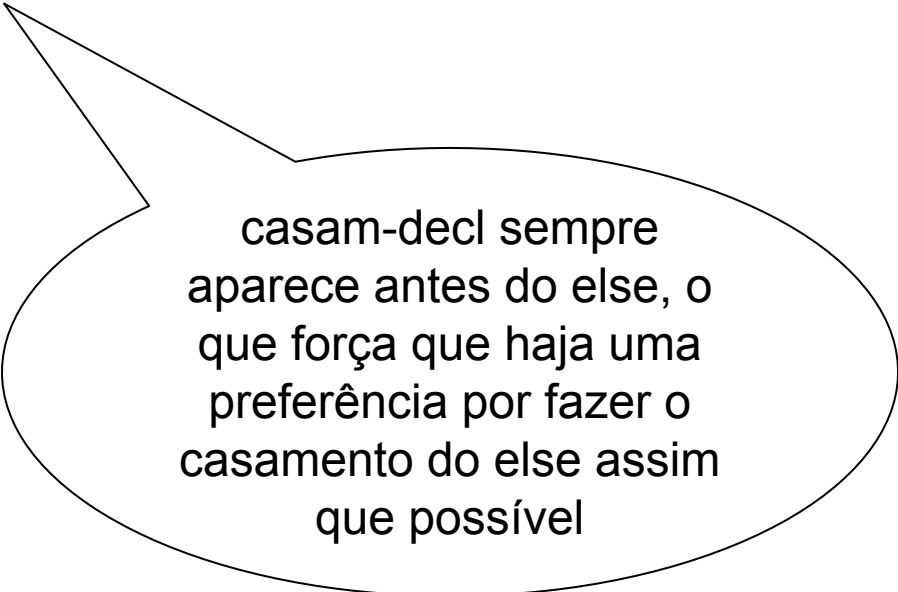
- Neste caso, é mais difícil modificar a gramática

`declaração → casam-decl | sem-casam-decl`

`casam-decl → if (exp) casam-decl else
casam-decl | outra`

`sem-casam-decl → if (exp) declaração | if
(exp) casam-decl else sem-casam-decl`

`exp → 0 | 1`



casam-decl sempre
aparece antes do else, o
que força que haja uma
preferência por fazer o
casamento do else assim
que possível

Eliminando ambiguidade

- Outra opção: inserir uma construção “endif”

`declaração → if-decl | outra`

`if-decl → if (exp) declaração endif | if
(exp) declaração else declaração endif`

`exp → 0 | 1`

- Agora não há mais dúvida

`if(0) if (1) outra else outra endif endif`

Eliminando ambiguidade

- Terceira técnica: inserir regras “extras” diretamente no analisador

`declaração → if-decl | outra`

`if-decl → if (exp) declaração | if (exp)`

`declaração else declaração`

`exp → 0 | 1`

- Neste exemplo, é possível dizer para o analisador ser “ganancioso”
 - Ou seja, sempre buscar a regra que faz o casamento com mais tokens
 - É uma política que a maioria dos analisadores (ANTLR, YACC) já segue
 - Recomendado quando modificar a gramática aumenta a complexidade

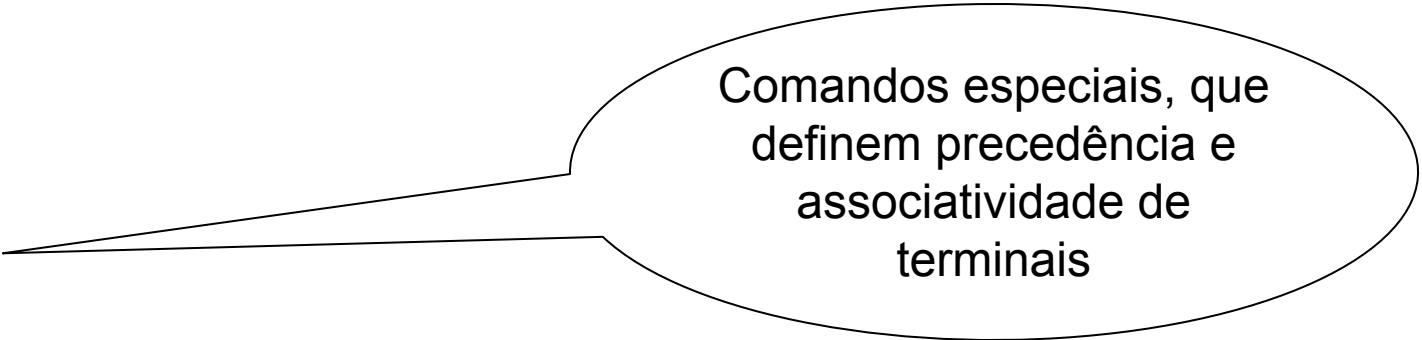
Eliminando a ambiguidade

- Outro exemplo dessa técnica - YACC
 - É possível definir a precedência e associatividade dos terminais
 - Considere o seguinte exemplo de gramática:

```
%left '+'
```

```
%left '*'
```

```
E ::= E + E | E * E | NUM
```



Comandos especiais, que
definem precedência e
associatividade de
terminais

Eliminando a ambiguidade


- Dada a entrada $3 + 4 * 5$
- Após ler o símbolo “4”, o YACC está na seguinte configuração (*veremos depois como ele faz análise*)
 - $E + E <\text{YACC está aqui}> * 5$
- Nesse momento ele precisa decidir entre fazer a inferência (reduzir $E + E$ para E) ou continuar lendo
- Ele então olha para o terminal mais à direita do seu lado esquerdo (+), e o terminal mais à esquerda do seu lado direito (*)
 - Neste caso, $*$ tem precedência sobre $+$, então a decisão é não inferir neste momento, e sim continuar lendo:
 - $E + E * <\text{YACC}> a \rightarrow E + E * a <\text{YACC}> \rightarrow E + E * E \rightarrow E + E \rightarrow E$

Ambiguidade

- Remover a ambiguidade nem sempre é possível
- Não há algoritmo
 - Como aquele que remove não-determinismo em autômatos finitos
- Alguns exemplos são clássicos
 - Expressões aritméticas
 - If-then-else
- Assim como as suas soluções
- Na prática, você vai resolver as ambiguidades (não-determinismos/conflitos) de acordo com o algoritmo de análise sintática
 - Algoritmos LL tem uma certa forma
 - Algoritmos LR tem outra forma
 - Por isso é importante conhecer estes algoritmos

Recursividade à esquerda

Recursividade à esquerda

- Uma gramática é recursiva à esquerda se houver um não-terminal A tal que haja uma derivação
 - $A \Rightarrow Ax$
- Alguns algoritmos não conseguem lidar com gramáticas recursivas à esquerda
 - É necessário remover
- Regra simples:
 - $A \xRightarrow{\quad} A\alpha \mid \beta$

 - $A \rightarrow \beta R$
 - $R \rightarrow \alpha R \mid \varepsilon$

Recursividade à esquerda

- Obs: É diferente de quando vimos o caso da associatividade dos operadores
 - Naquele exemplo, o objetivo é eliminar a ambiguidade
 - As mudanças alteravam as derivações possíveis para remover a ambiguidade
- $A \rightarrow SaS \mid b \rightarrow A \rightarrow baS \mid b$
- A solução aqui é mais genérica
 - Não remove a ambiguidade!

Recursividade à esquerda

- Existem três tipos de recursividade à esquerda
- Recursão imediata em apenas uma produção
 - $A \rightarrow A\alpha \mid \beta$
- Recursão imediata em mais de uma produção
 - $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$
- Recursão é não-imediata
 - $A \rightarrow B\beta \mid \dots$
 - $B \rightarrow C\gamma \mid \dots$
 - $C \rightarrow A\delta \mid \dots$

Recursividade à esquerda

- **Recursão imediata em apenas uma produção**
- Antes
 - $A \rightarrow A\alpha \mid \beta$
- Depois
 - $A \rightarrow \beta R$
 - $R \rightarrow \alpha R \mid \varepsilon$
- Ex:
 - Antes:
 - $\text{expr} \rightarrow \text{expr '+' termo} \mid \text{termo}$
 - Depois
 - $\text{expr} \rightarrow \text{termo expr2}$
 - $\text{expr2} \rightarrow \text{'+' termo expr2} \mid \varepsilon$

Recursividade à esquerda

- **Recursão imediata em mais de uma produção**
- Primeiro, agrupe as produções da seguinte forma
 - $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$
 - Onde nenhum β_i começa com A , e nenhum α_i é ε
- Substitua as produções de A por
 - $A \rightarrow \beta_1 R \mid \beta_2 R \mid \dots \mid \beta_n R$
 - $R \rightarrow \alpha_1 R \mid \alpha_2 R \mid \alpha_3 R \mid \dots \mid \alpha_m R \mid \varepsilon$
- Ex:
 - Antes
 - $\text{expr} \rightarrow \text{expr '+' termo} \mid \text{expr '-' termo} \mid \text{termo} \mid \text{constante}$
 - Depois
 - $\text{expr} \rightarrow \text{termo expr2} \mid \text{constante expr2}$
 - $\text{expr2} \rightarrow \text{'+' termo expr2} \mid \text{'-' termo expr2} \mid \varepsilon$

Recursividade à esquerda

- Recursão não-imediata
 - Situação menos comum
 - Algoritmo um pouco mais complicado (mas nem tanto)
 - Não veremos aqui na disciplina
- Se algum dia você se deparar com uma situação assim
 - Procure o livro do dragão!

Fatoração à esquerda

Fatoração à esquerda

- Útil para deixar uma gramática adequada para análise sintática preditiva
- Quando a escolha entre duas produções não é clara
 - Pode-se tentar reescrever as produções para atrasar a decisão até que haja entrada suficiente para tomar a decisão
- Ex:
 - comando \rightarrow if (expr) then cmd else cmd
 - comando \rightarrow if (expr) then cmd
- Mediante um token “if”, um analisador preditivo (que tenta prever a regra) não sabe o que fazer

Fatoração à esquerda

- Fatoração é simples:

- Antes:

- $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3 \mid \dots \mid \alpha\beta_n$

- Depois:

- $A \rightarrow \alpha R$

- $R \rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \mid \dots \mid \beta_n$

- Exemplo:

- Antes:

- $\text{comando} \rightarrow \text{if (expr) then cmd else cmd} \mid \text{if (expr) then cmd}$

- Depois:

- $\text{comando} \rightarrow \text{if (expr) then cmd comandoElse}$

- $\text{comandoElse} \rightarrow \text{else cmd} \mid \epsilon$

Neste exemplo:

$$\alpha = \text{if (expr) then cmd}$$

$$\beta_1 = \text{else cmd}$$

$$\beta_2 = \epsilon$$

EBNF e diagramas sintáticos

EBNF

- Na prática existem algumas notações que facilitam a escrita de gramáticas
- Principalmente no caso de recursividade
 - Recursividade é quase sempre usado para representar uma lista
 - Ex:
 - $A \rightarrow Aa \mid a$ (um ou mais)
 - $A \rightarrow Aa \mid \varepsilon$ (zero ou mais)
- Outro exemplo comum é opcionalidade
 - $A \rightarrow a \mid \varepsilon$ (zero ou um)
- Tais notações são chamadas de EBNF
 - Ou BNF estendida

EBNF

- Usaremos aqui a notação do ANTLR

$A \rightarrow x? = A \rightarrow x \mid \varepsilon$

$A \rightarrow x^* = A \rightarrow xA \mid \varepsilon$ (ou $A \rightarrow Ax \mid \varepsilon$)

$A \rightarrow x+ = A \rightarrow xA \mid x$ (ou $A \rightarrow Ax \mid x$)

- Exs:

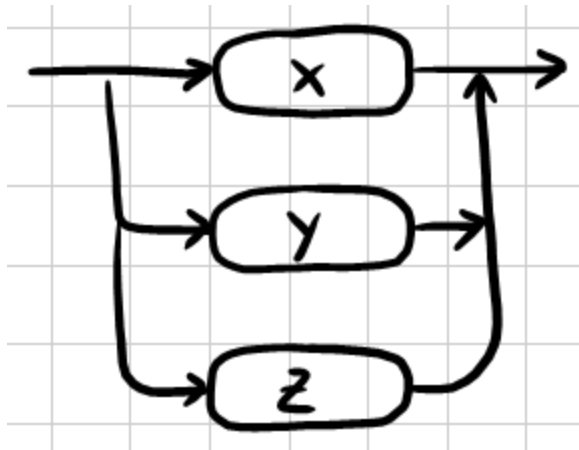
`expr : termo (op1 termo)+`

`if-decl : 'if' '(' expr ')' 'then' cmd
('else' cmd)?`

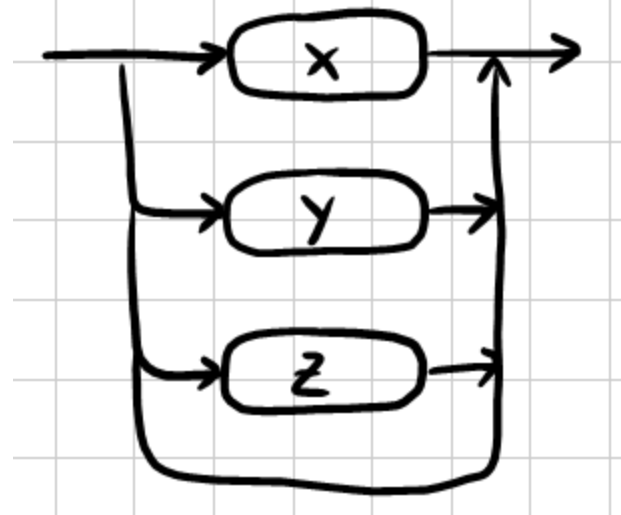
Diagramas sintáticos

- Ajudam a visualizar as regras

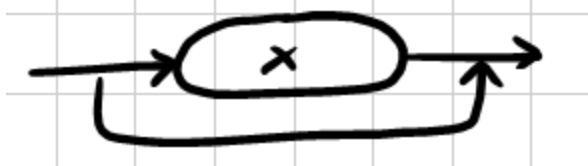
$(x \mid y \mid z)$



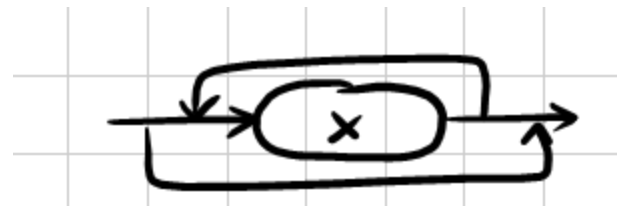
$(x \mid y \mid z) ?$



$x ?$



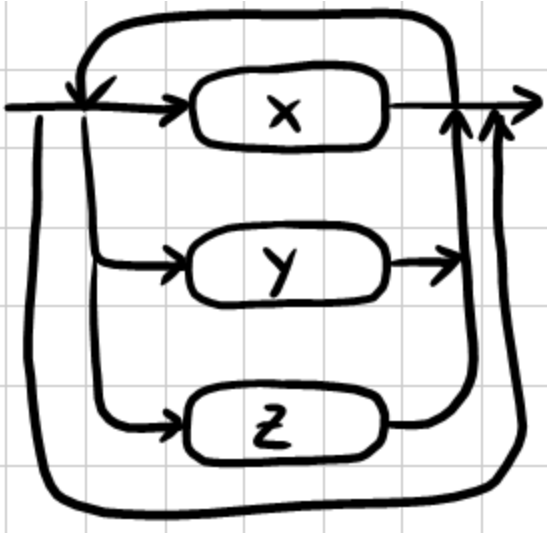
x^*



Diagramas sintáticos

- Ajudam a visualizar as regras

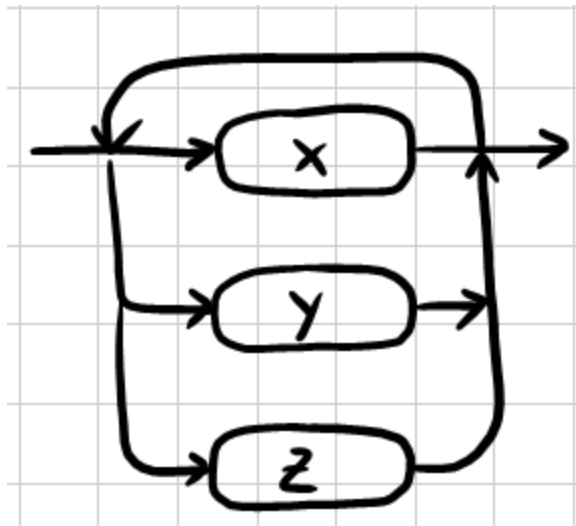
$(x \mid y \mid z)^*$



x^+

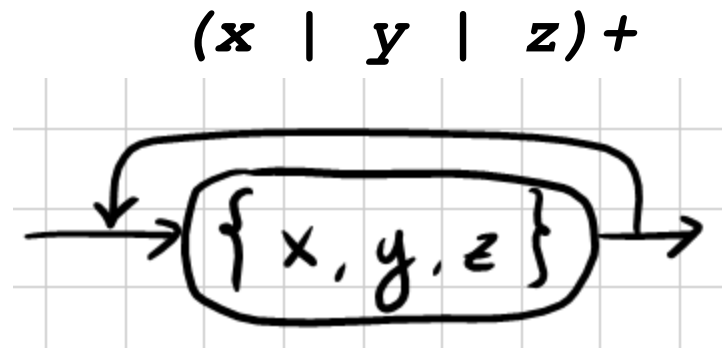
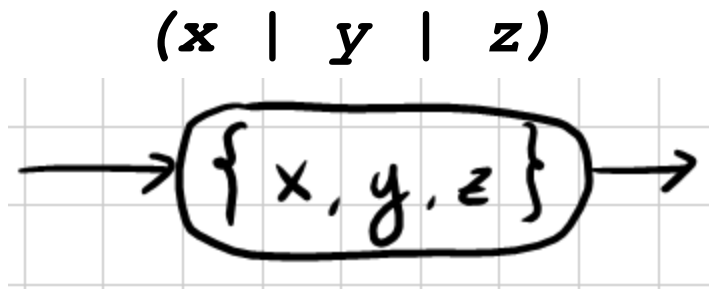


$(x \mid y \mid z)^+$



Diagramas sintáticos

- Ajudam a visualizar as regras



Diagramas sintáticos

- Desenhe os diagramas sintáticos para as seguintes regras

☐ **NUMINT** : { '+' | '-' } ? { '0' .. '9' } + ;

☐ **NUMREAL** : { '+' | '-' } ? { '0' .. '9' } + { '.' { '0' .. '9' } + } ? ;

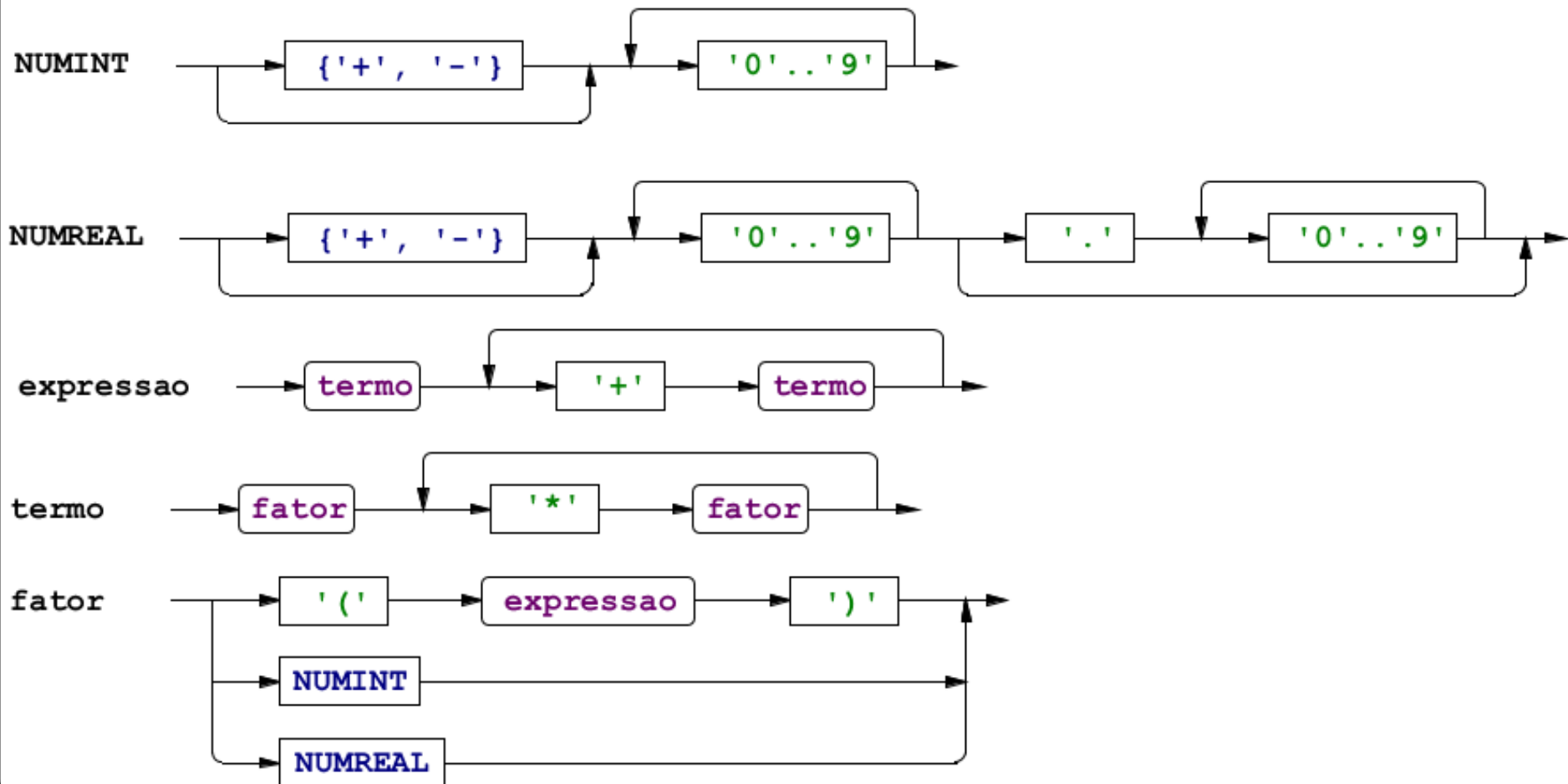
☐ **expressao** : **termo** { '+' **termo** } + ;

☐ **termo** : **fator** { '*' **fator** } + ;

☐ **fator** : '(' **expressao** ')' | **NUMINT** | **NUMREAL** ;

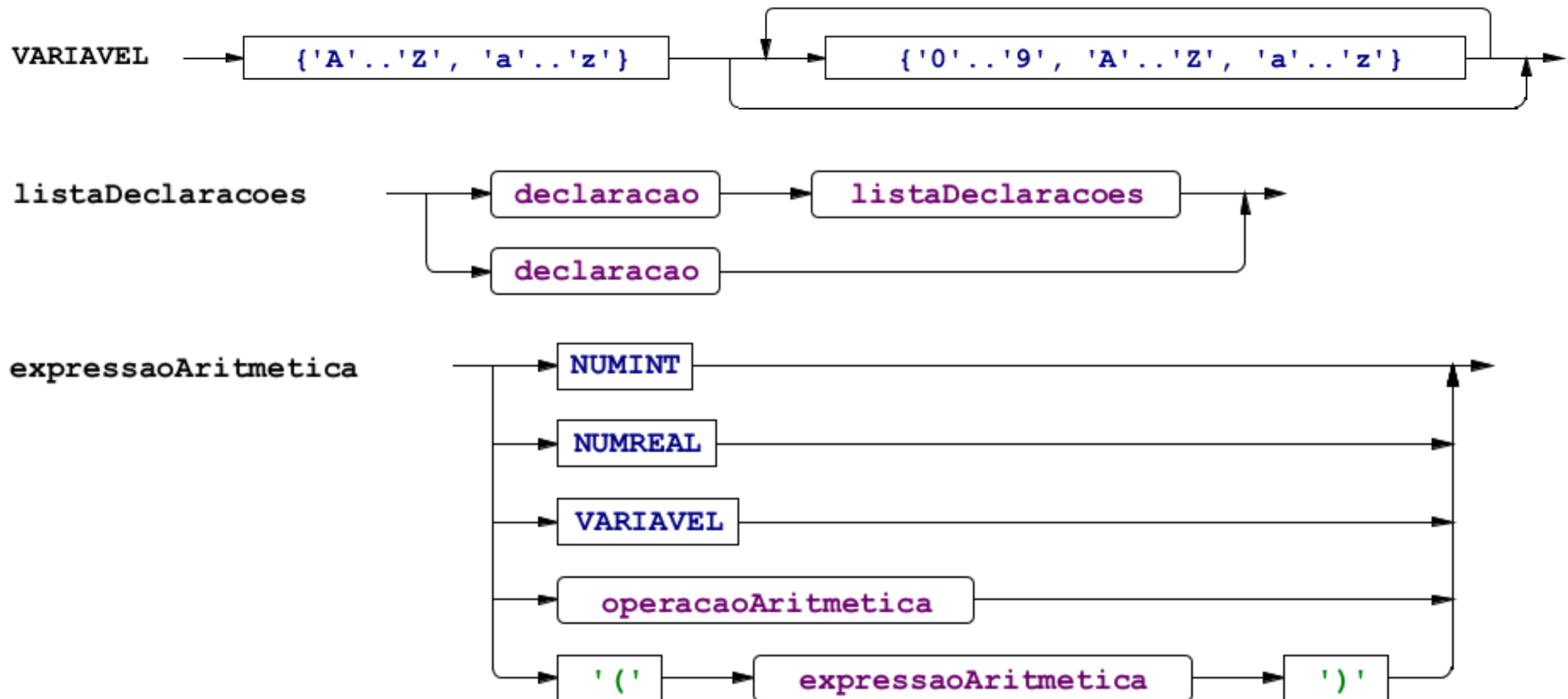
Diagramas sintáticos

- Resposta



Diagramas sintáticos

- Escreva as regras para os seguintes diagramas sintáticos



Diagramas sintáticos

- Resposta

```
VARIAVEL : ('a'..'z' | 'A'..'Z')  
          ('a'..'z' | 'A'..'Z' | '0'..'9')*;
```

```
listaDeclaracoes : declaracao listaDeclaracoes  
                  | declaracao;
```

```
expressaoAritmetica : NUMINT | NUMREAL |  
                    VARIAVEL | operacaoAritmetica | '('  
                    expressaoAritmetica ')';
```

Fim