Construção de compiladores

Profs. Mário César San Felice (e Helena Caseli,
Murilo Naldi, Daniel Lucrédio)
Departamento de Computação - UFSCar
1º semestre / 2018
Tópico 3 - Introdução à Análise Sintática

Análise sintática

Introdução

Contexto

- Linguagem humana tem:

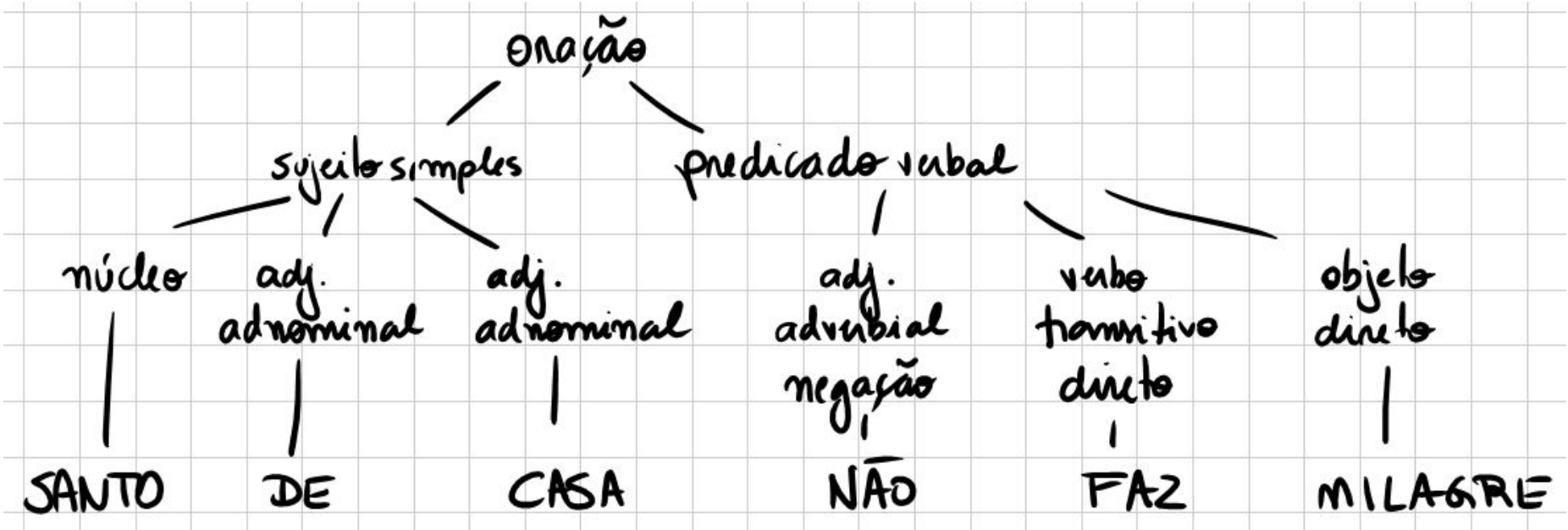
Vocabulário + Gramática

- Nomes das coisas
- Ações
- Composição
- Conceitos complexos

- Hoje começamos a tratar a gramática

Objetivo

- Objetivo da análise sintática
  - Reconhecer a estrutura das frases

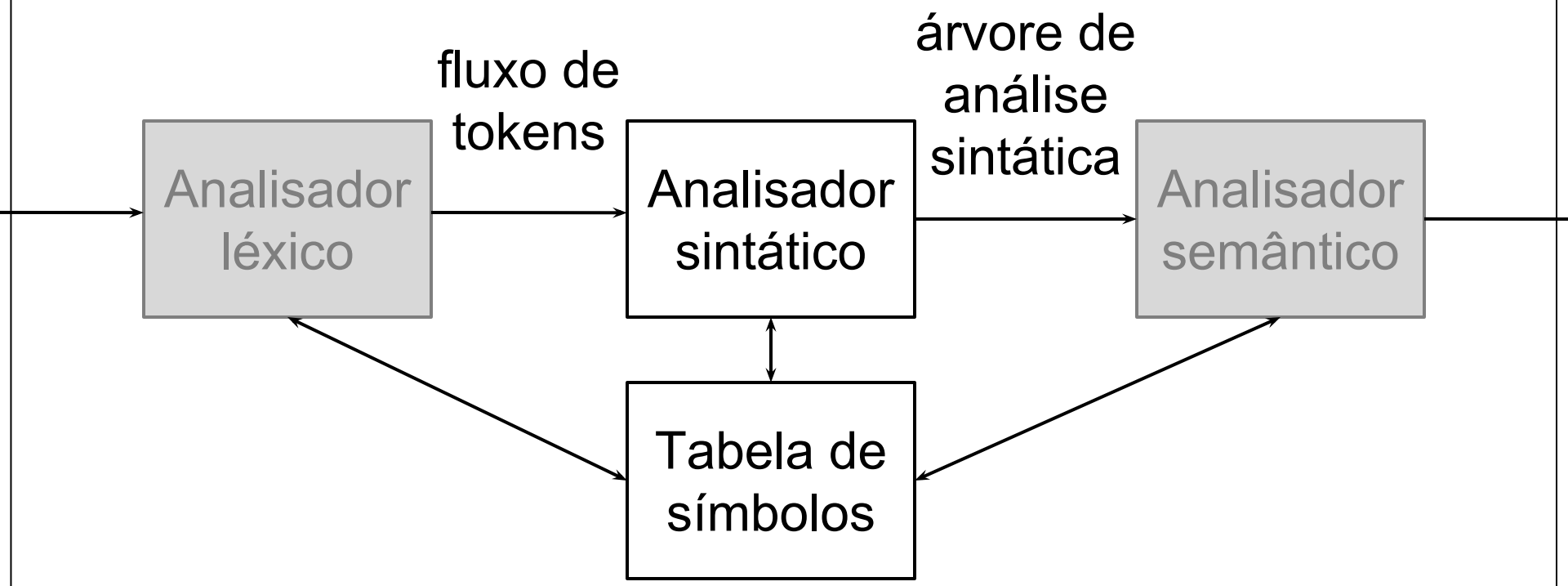


Em compiladores

- O objetivo é o mesmo
  - Frases = programas
  - Estrutura = linguagem de programação
- Humanos são exímios processadores de linguagem
- Computadores 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)

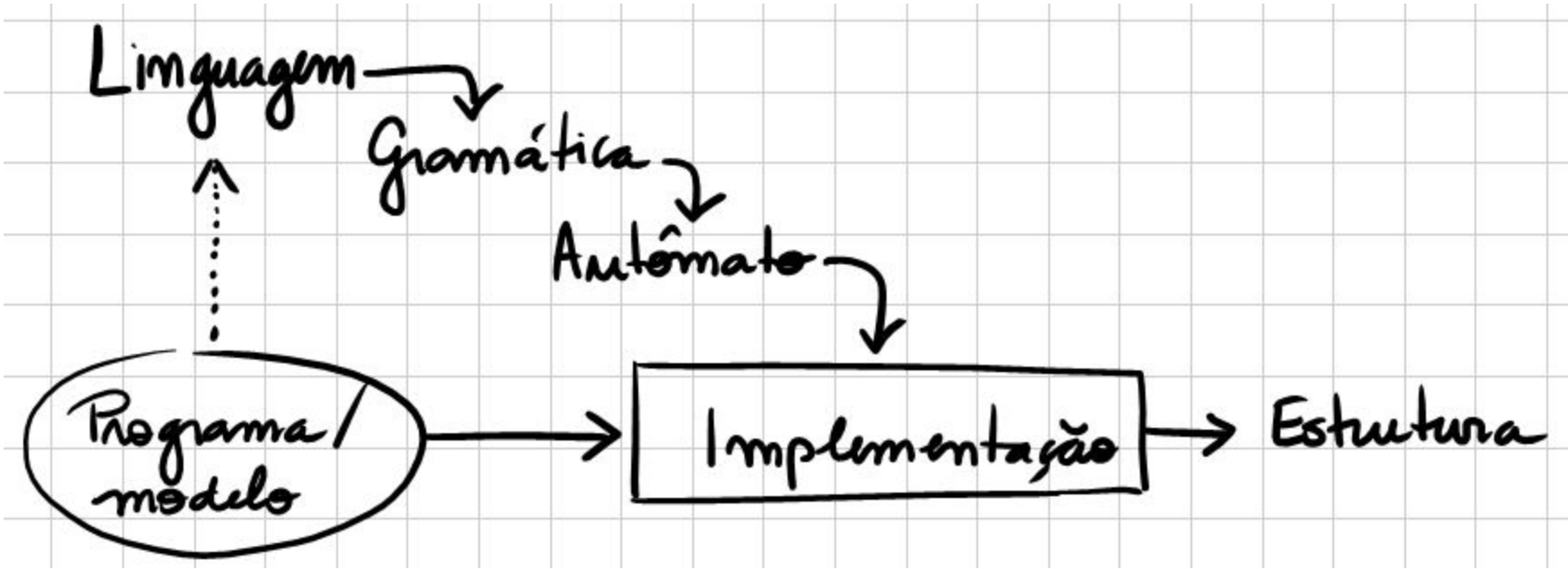
Contexto

- Definição da linguagem
  - Deve ser precisa e formal
  - Deve descrever a estrutura sintática

- A teoria da computação vem em auxílio
  - Gramáticas vistas em LFA
  - Permitem definir estruturas de cadeias de símbolos

  - E mais, para cada gramática existe
    - Um modelo formal de máquina reconhecedora!
      - Autômatos significam IMPLEMENTAÇÃO!

Compiladores x LFA



Compiladores x LFA

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

Hierarquia	Gramáticas	Linguagens	Autômato
Tipo-0	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	Expressões regulares	Regulares	Autômatos finitos

Compiladores x LFA

- Qual tipo de linguagem escolher?
  - Escolha óbvia
    - Tipo-0: Linguagens recursivamente enumeráveis
    - Gramáticas irrestritas
    - Máquinas de Turing
- Vantagem:
  - Linguagens RE cobrem tudo o que é necessário
- Desvantagens:
  - Gramáticas irrestritas são difíceis de conceber
    - E de transformar em uma Máquina de Turing

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

Compiladores x LFA

- No outro extremo:
  - Tipo-3: Linguagens regulares
  - Gramáticas regulares
  - Autômatos finitos

- Vantagens:
  - Gramáticas simples de conceber
    - E de converter em um autômato

- Desvantagens:
  - Não cobre as necessidades das linguagens
  - Não há recursividade / capacidade de “contar”
    - Regra S → (S) é proibida

Compiladores x LFA

- Sobrou:
  - Tipo-2: Linguagens livres de contexto
  - Gramáticas livres de contexto
  - Autômatos com uma pilha

- Vantagens:
  - Gramáticas (relativamente) fáceis de conceber
  - E de converter em um autômatos com pilha

- Desvantagens:
  - Algumas construções da maioria das linguagens exige sensibilidade ao contexto
    - Mas é possível contornar!!

Gramáticas livre de contexto

- As gramáticas livres de contexto são a melhor opção, pois existem:
  - Técnicas para projetar essas gramáticas
  - Algoritmos para análise sintática baseados em seus princípios
  - Técnicas para adicionar sensibilidade ao contexto
    - Em compiladores corresponde a ANÁLISE SEMÂNTICA

Análise sintática

- Nesta parte da disciplina veremos:
  - Alguns dos algoritmos para análise sintática
    - LL(k), LL(*), LR/LALR, GLR/Universal
  - Características, necessidades específicas, vantagens e desvantagens de cada algoritmo

- É importante conhecer a ESSÊNCIA de cada técnica
  - Ainda que não a implementemos em detalhes
  - Para poder tomar decisões de projeto consistentes

Análise sintática

- Mas antes, vamos estudar o formalismo
- O que é análise sintática baseada em gramáticas livres de contexto?
- Primeiro vamos entender 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**: um único símbolo. Esses símbolos são chamados de **variáveis** ou **não-terminais**

Lado direito ou **corpo**: uma cadeia de símbolos. Pode ter variáveis e **terminais**

A variável que aparece do lado esquerdo da primeira regra é designada **variável ou símbolo inicial**. (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:
  - $G_{palíndromos}$ = ({L},{0,1},P,L)

P = {

$L \rightarrow \varepsilon$

$L \rightarrow 0$

$L \rightarrow 1$

$L \rightarrow 0L0$

$L \rightarrow 1L1$

}

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

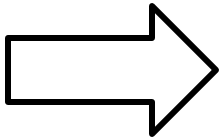
- Os não-terminais definem normalmente as construções da linguagem
  - De alto nível (programa, função, bloco)
  - De baixo nível (comandos, expressões)

Gramáticas livres de contexto

```
Programa → ListaComandos
ListaComandos → Comando ListaComandos
ListaComandos → Comando
Comando → ComandoIf
Comando → ComandoAtrib
ComandoIf → TK_IF Expr TK_THEN Comando
ComandoIf → TK_IF Expr TK_THEN Comando
                  TK_ELSE Comando
ComandoAtrib → id TK_ATRIB Expr
...
```

Gramáticas livres de contexto

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



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

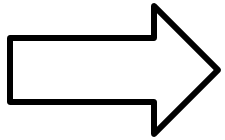
Se houver mais de uma produção para uma mesma variável, podemos agrupá-las com o símbolo “|”.

Gramáticas livres de contexto

```
Programa → ListaComandos

ListaComandos → Comando ListaComandos |
                Comando

Comando → ComandoIf | ComandoAtrib

ComandoIf → TK_IF Expr TK_THEN Comando |
            TK_IF Expr TK_THEN Comando
              ELSE Comando

ComandoAtrib → id TK_ATRIB Expr

...
```

Gramáticas livres de contexto

$A \rightarrow 0B1$
$B \rightarrow \# \mid \%$



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

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

Gramáticas livres de contexto

```
Programa → ListaComandos
ListaComandos → Comando ListaComandos |
                Comando
Comando → ComandoIf | (id TK_ATRIB Expr)
ComandoIf → 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}; S=E;
  - P = {
    - E → I | E + E | E * E | (E)
    - I → a | b | Ia | Ib | I0 | I1

  }

Gramáticas livres de contexto

- E → I | E + E | E * E | (E)
- I → a | b | Ia | Ib | I0 | I1

- Inferência recursiva
  - Dada uma cadeia (conjunto de símbolos terminais)
  - Do corpo para a cabeça

- Ex: a*(a+b00)
  - a*(a+b00) ⇐ a*(a+I00) ⇐ a*(a+I0) ⇐ a*(a+I) ⇐ a*(a+E) ⇐ a*(I+E) ⇐ a*(E+E) ⇐ a*(E) ⇐ a*E ⇐ I*E ⇐ E*E ⇐ E

Gramáticas livres de contexto

- E → I | E + E | E * E | (E)
- I → a | b | Ia | Ib | I0 | I1

- Derivação
  - Dada uma cadeia (conjunto de símbolos terminais)
  - Da cabeça para o corpo

- Ex: a*(a+b00)
  - E ⇒ E*E ⇒ I*E ⇒ a*E ⇒ a*(E) ⇒ a*(E+E) ⇒ a*(I+E) ⇒ a*(a+E) ⇒ a*(a+I) ⇒ a*(a+I0) ⇒ a*(a+I00) ⇒ a*(a+b00)

Gramáticas livres de contexto

- E → I | E + E | E * E | (E)
- I → a | b | Ia | Ib | I0 | I1

- Símbolo de derivação: $\Rightarrow$

- Derivação em múltiplas etapas: $\overset{*}{\Rightarrow}$
  - $E \overset{*}{\Rightarrow} a*(E)$
  - $a*(E+E) \overset{*}{\Rightarrow} a*(a+I00)$
  - $E \overset{*}{\Rightarrow} a*(a+b00)$

Gramáticas livres de contexto

- Derivações mais à esquerda
  - Sempre substituir a variável mais à esquerda
  - Notação: $\Rightarrow_{lm}$, $\overset{*}{\Rightarrow}_{lm}$

- Derivações mais à direita
  - Sempre substituir a variável mais à direita
  - Notação: $\Rightarrow_{rm}$, $\overset{*}{\Rightarrow}_{rm}$

Árvores de análise sintática

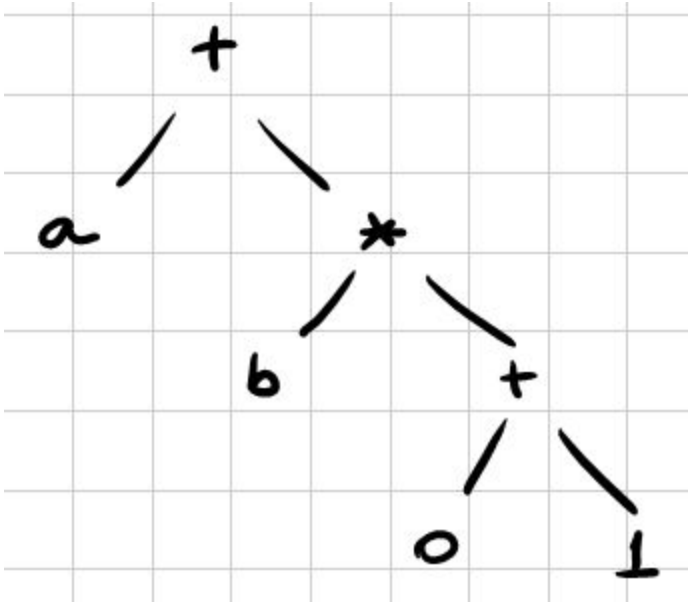
Árvores de análise sintática

- Representação visual para derivações
- Mostra claramente como os símbolos de uma cadeia de terminais estão agrupados em subcadeias
- Permite analisar alguns aspectos da linguagem e ver o processo de derivação / inferência recursiva
  - Ex: a+b*(0+1)
- Produções
  - E → I | E + E | E * E | (E)
  - I → a | b | 0 | 1

Árvores de análise sintática

- Também conhecidas por:
  - Árvores de derivação ou *parse trees*

- Elas representam completamente a derivação
  - 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
- Omite (abstrai) detalhes pois
  - Muitas vezes as regras gramaticais incluem não-terminais somente como mecanismos auxiliares

Árvores de sintaxe abstrata

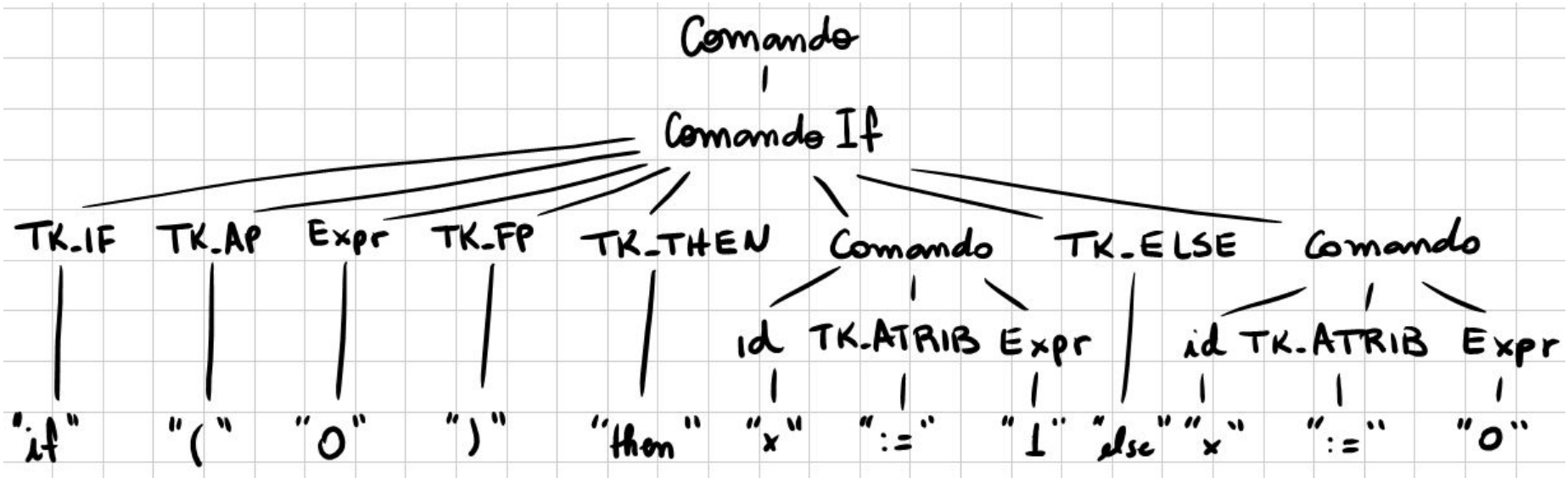
- Ex:

```
Comando → ComandoIf | (id TK_ATRIB Expr)

ComandoIf → TK_IF TK_AP Expr TK_FP
              TK_THEN Comando |
            TK_IF TK_AP Expr TK_FP
              TK_THEN Comando ELSE
              Comando

Expr → TK_0 | TK_1
```

Árvores de sintaxe abstrata

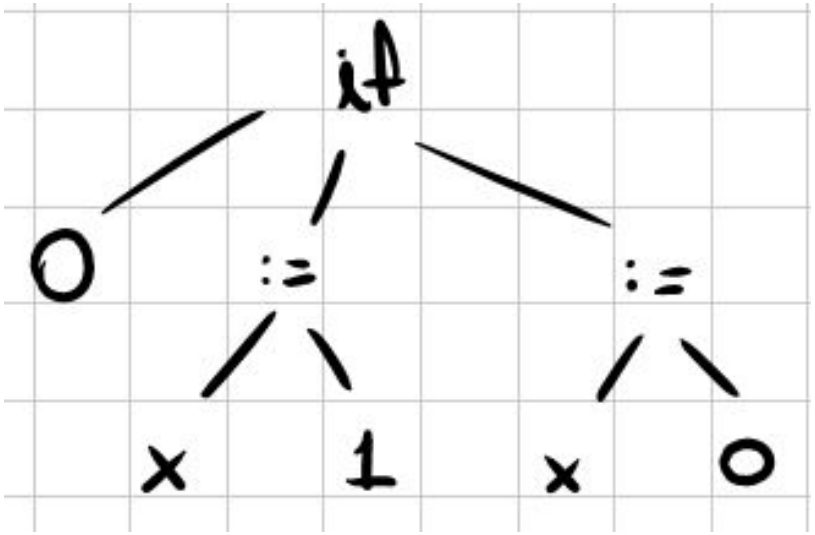
- Árvore de análise sintática (concreta) da cadeia:

```
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

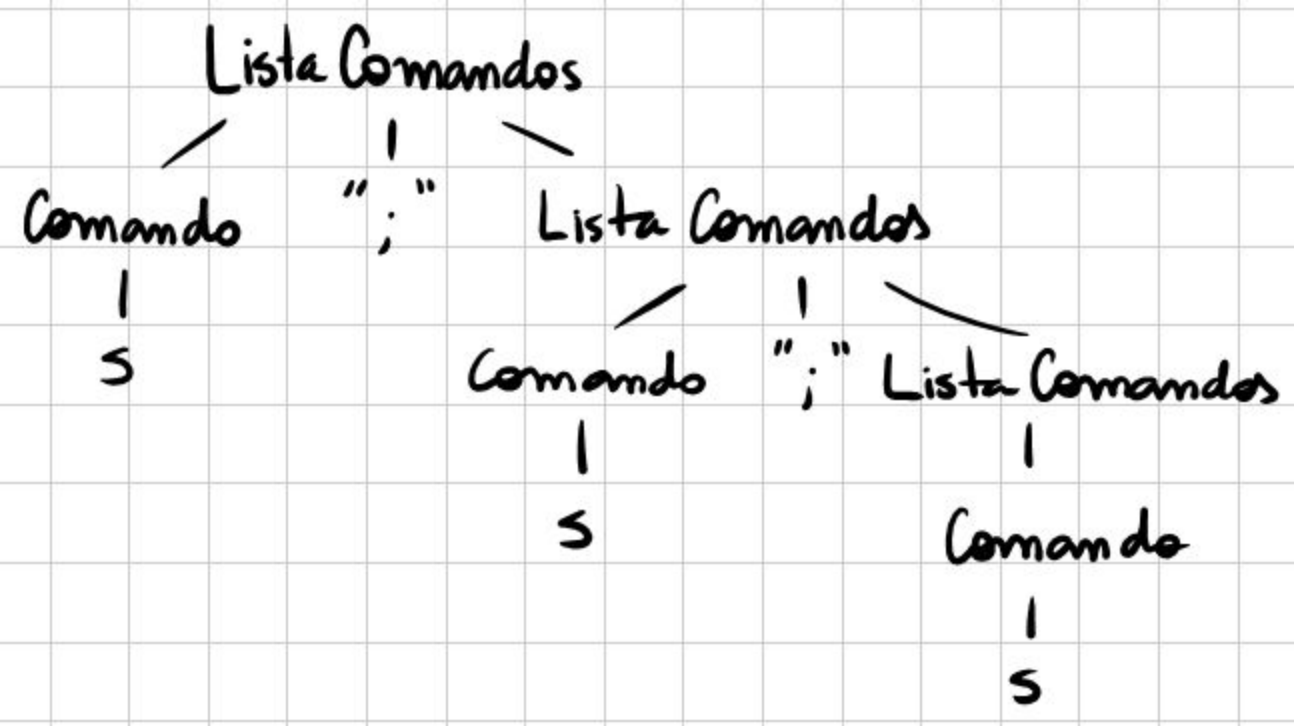
- Outro exemplo:
  - Sequência de declarações separadas por ponto-e-vírgula:

```
ListaComandos
            → Comando ';' ListaComandos |
              Comando

Comando     → s
```

Árvores de sintaxe abstrata

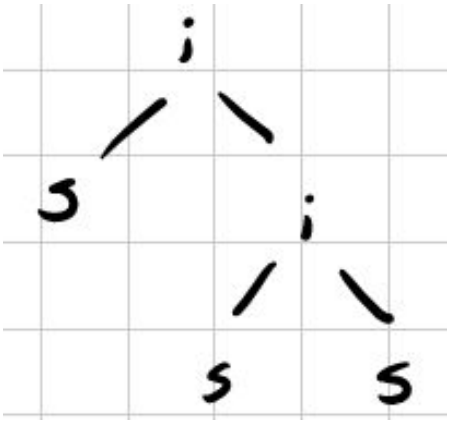
- Fazendo a árvore de análise sintática da cadeia:

```
s;s;s
```

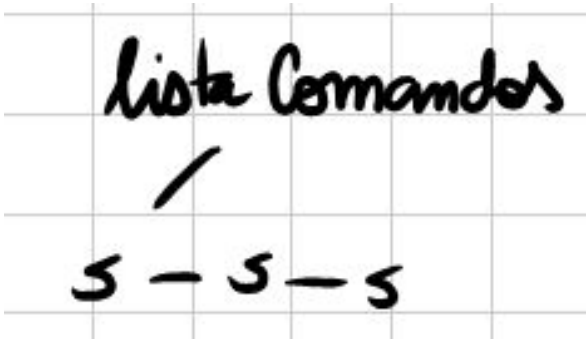


Árvores de sintaxe abstrata

- Na verdade, poderíamos representar assim:



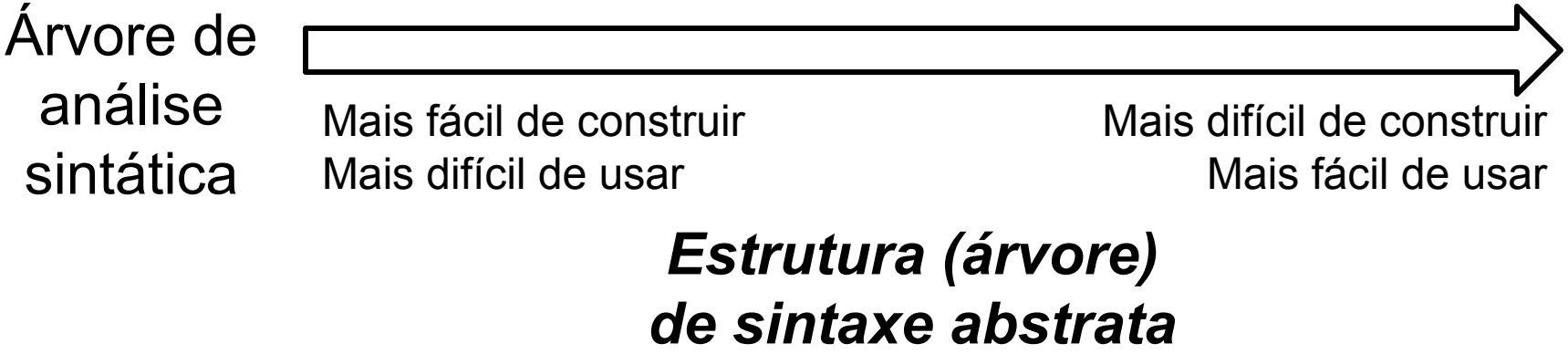
- Ou, melhor ainda:



Árvores de sintaxe abstrata

- Na verdade, a sintaxe abstrata pode ser
  - QUALQUER ESTRUTURA DE DADOS
    - Como a lista do exemplo anterior
    - Ou um grafo direcionado
  - Em alguns contextos, é chamada METAMODELO

- Na prática, quase sempre é uma árvore



Exemplo

- Gramática simplificada da linguagem ALGUMA

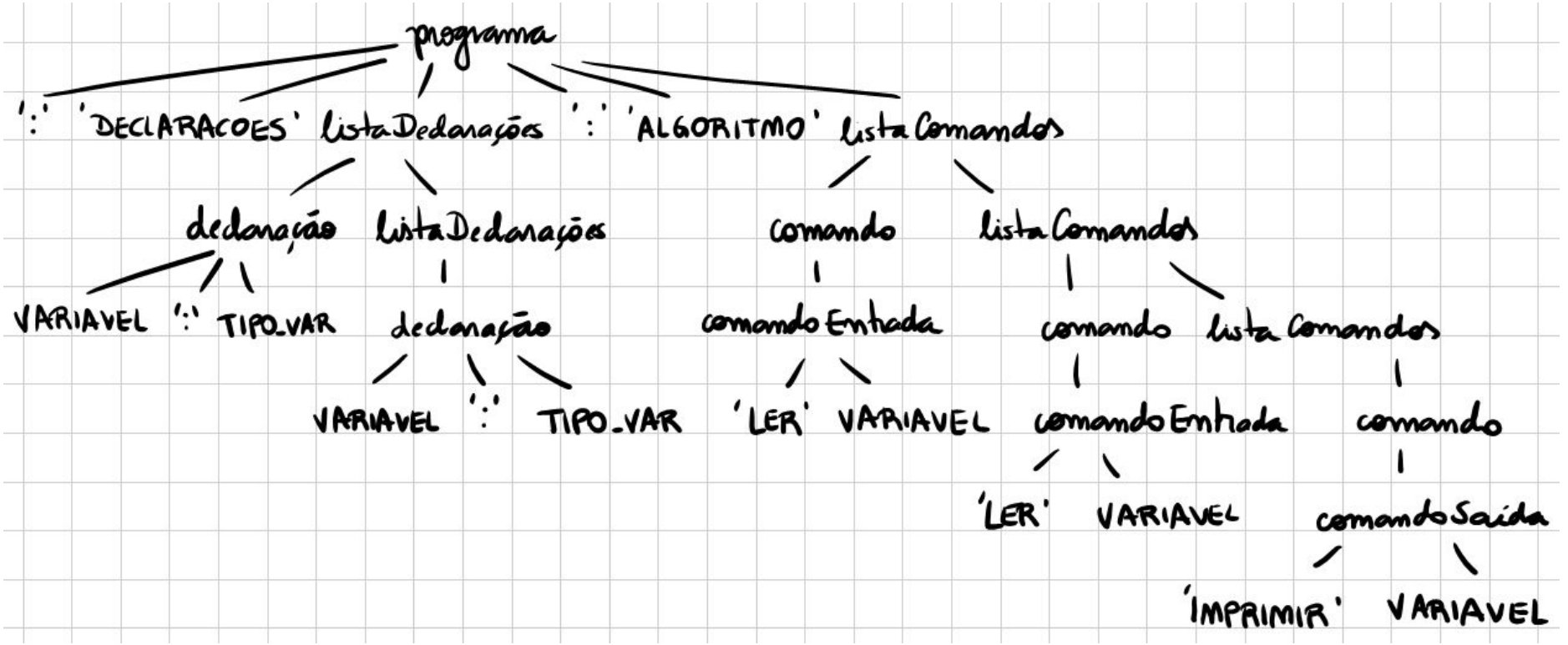
```
VARIAVEL : ('a'..'z'|'A'..'Z`)('a'..'z'|'A'..'Z'|'0'..'9')*;
TIPO_VAR : 'INTEIRO' | 'REAL';

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)



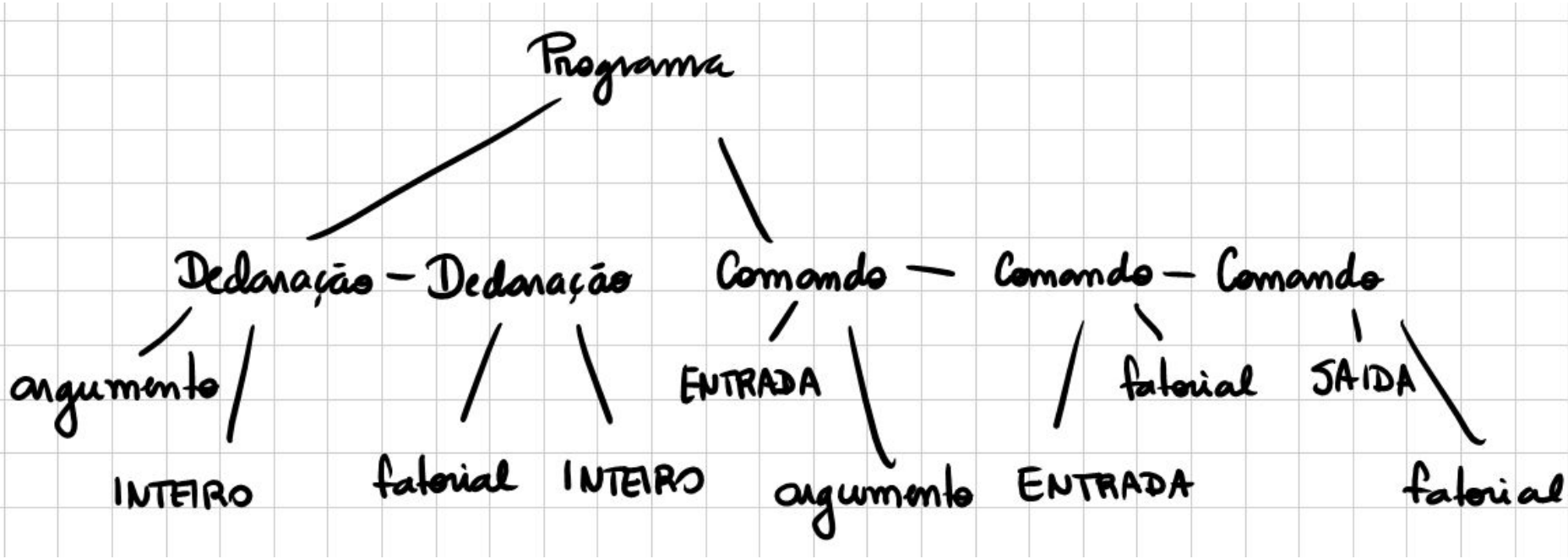
Exemplo

- 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



Resumo: abstrata vs concreta

- A sintaxe concreta é necessária
  - É através dela que o compilador analisa a estrutura
  - Essencial para validar repetições, condicionais, etc

- Uma vez que a análise sintática é concluída
  - Podemos “jogar fora” a sintaxe concreta
  - Mas precisamos de outra estrutura de dados
    - Mais limpa e fácil de trabalhar
    - Chamada de sintaxe abstrata

- Podemos usar diversas estruturas de dados ela
  - 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”

Ambiguidade

- Outro exemplo (gramática à direita)
  - Encontre derivações mais à esquerda para a cadeia a + b*a

$E \rightarrow I$
$E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow (E)$
$I \rightarrow a$
$I \rightarrow b$

- Respostas:
  - $E \Rightarrow E + E \Rightarrow I + E \Rightarrow a + E$
    $\Rightarrow a + E * E \Rightarrow a + I * E \Rightarrow a + b * E$
    $\Rightarrow a + b * I \Rightarrow a + b * a$

  - $E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow I + E * E$
    $\Rightarrow a + E * E \Rightarrow a + I * E \Rightarrow a + b * E$
    $\Rightarrow a + b * I \Rightarrow a + b * a$

Ambiguidade

- A diferença entre as árvores e as derivações mais à esquerda é significativa
  - Dependendo dela, o gerente pode ficar sem braços
    - Cadeira para ___
    - ___ sem braços
  - Dependendo da derivação à esquerda que usar, a adição pode ocorrer antes da multiplicação
    - a + ___
    - ___ * a

Ambiguidade

- Gramáticas são usadas para dar estrutura a programas, documentos, etc
  - Supõe-se que essa estrutura é única
  - Caso não seja, podem ocorrer problemas

- Nem toda gramática fornece estruturas únicas
  - Algumas vezes é possível reprojetar a gramática para eliminar a ambiguidade
  - Em outras vezes, isso é impossível
    - Existem linguagens “inerentemente ambíguas”
    - Ou seja, toda gramática para esta linguagem será ambígua

Ambiguidade

- O que caracteriza ambiguidade
  - A existência de duas ou mais árvores de análise sintática para uma mesma cadeia da linguagem

- Formalmente:
  - Uma CFG G = (V,T,P,S) é ambígua se existe pelo menos uma cadeia w em T* para a 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

- Podemos relacionar ambiguidade com as 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

Eliminando ambiguidade

- Dificuldades:
  - Saber se uma gramática é ambígua é um problema indecidível
    - Descobrir que uma gramática é ambígua depende de análise, exemplos e sorte!
  - Existem linguagens inerentemente ambíguas
    - TODA CFG para essas linguagens será ambígua
  - Mesmo para uma linguagem que não é inerentemente ambígua
    - Não existe algoritmo para remover a ambiguidade

Eliminando ambiguidade

- Existem técnicas 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

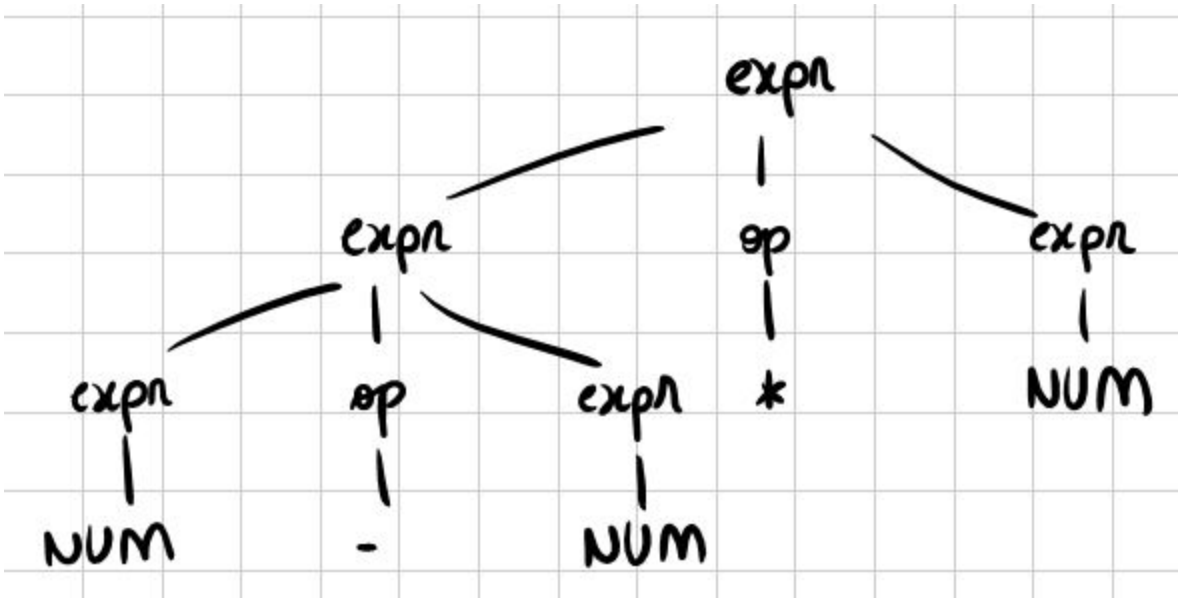
- Exemplo clássico: expressões aritméticas

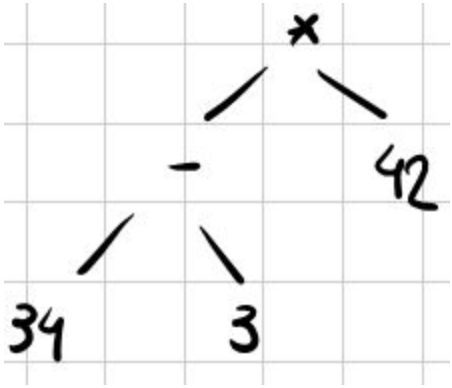
  expr → expr op expr | '(' expr ')' | NUM

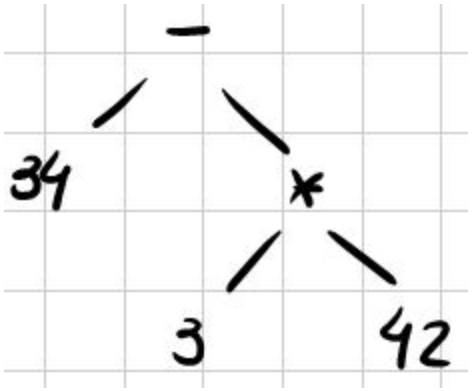
  op → + | - | *

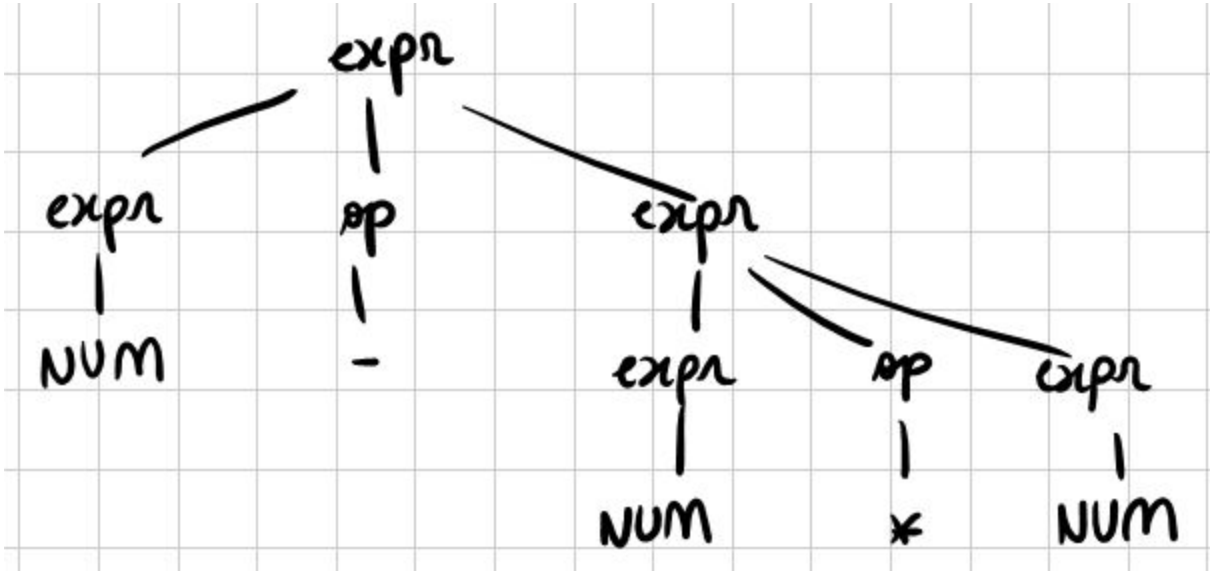
- É fácil demonstrar que existe mais de uma árvore de análise sintática para a cadeia 34 – 3 * 42
  - Também resultam em sintaxes abstratas diferentes

Eliminando ambiguidade







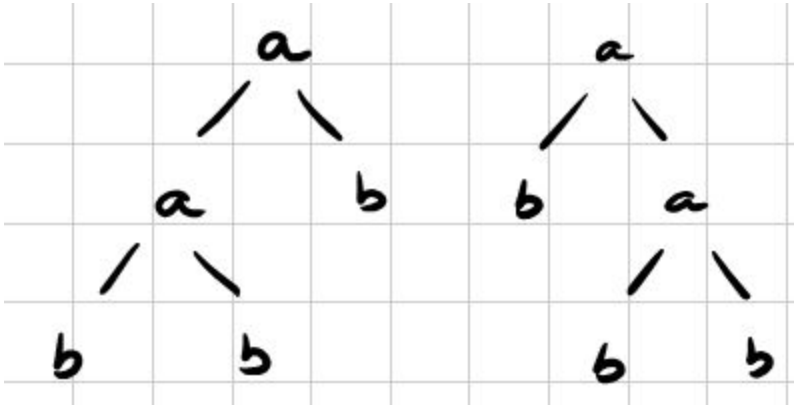


Eliminando ambiguidade

- Existe um ponto de ambiguidade
  - Associatividade

- Causado por uma recursividade dupla (à direita e à esquerda)
  - S → S ‘qualquer terminal’ S | ...

- Exemplo mais genérico
  - S → SaS | b
  - Cadeia = babab



Eliminando ambiguidade

- Nestes casos, é preciso remover a recursividade de um dos lados

- Para “forçar” a associatividade à esquerda

  $S \rightarrow SaS \mid b$ ➡ $S \rightarrow Sab \mid b$

- Para “forçar” a associatividade à direita

  $S \rightarrow SaS \mid b$ ➡ $S \rightarrow baS \mid b$

Eliminando ambiguidade

- No exemplo das expressões

  expr → expr op expr | '(' expr ')' | NUM

  op → + | - | *

- Forçando associatividade à esquerda:

  expr → expr op ('(' expr ')' | NUM) | '(' expr ')' | NUM

  op → + | - | *

- Para melhor legibilidade, vamos inserir outra regra:

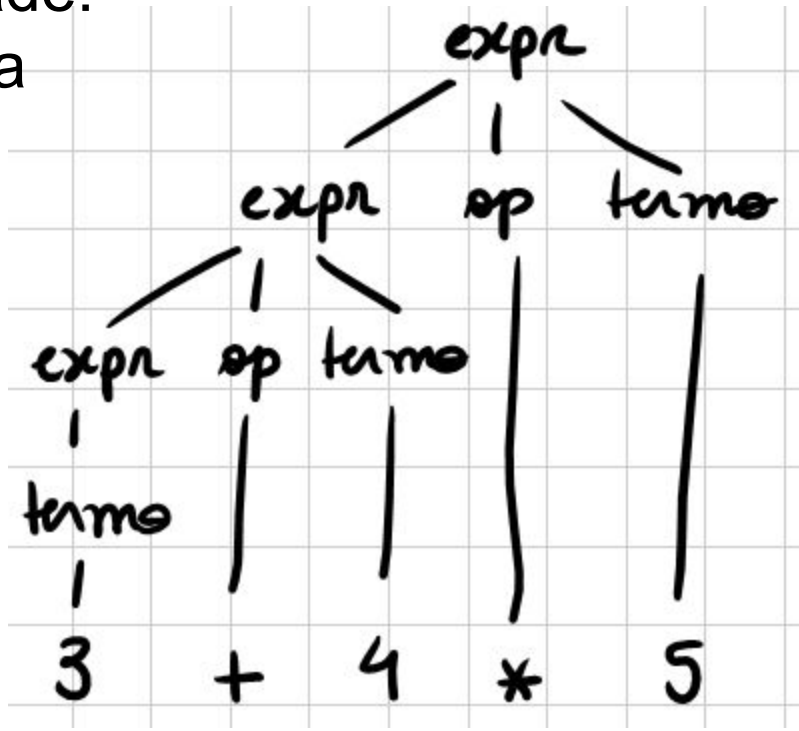
  expr → expr op termo | termo

  termo → '(' expr ')' | NUM

  op → + | - | *

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?
  - Matemáticos decretaram uma ordem “certa” para as operações

Eliminando ambiguidade

- Ao remover a ambiguidade
  - Eliminamos a flexibilidade de 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
  - Uma cascata de regras

expr → expr op1 termo | termo

termo → termo op2 fator | fator

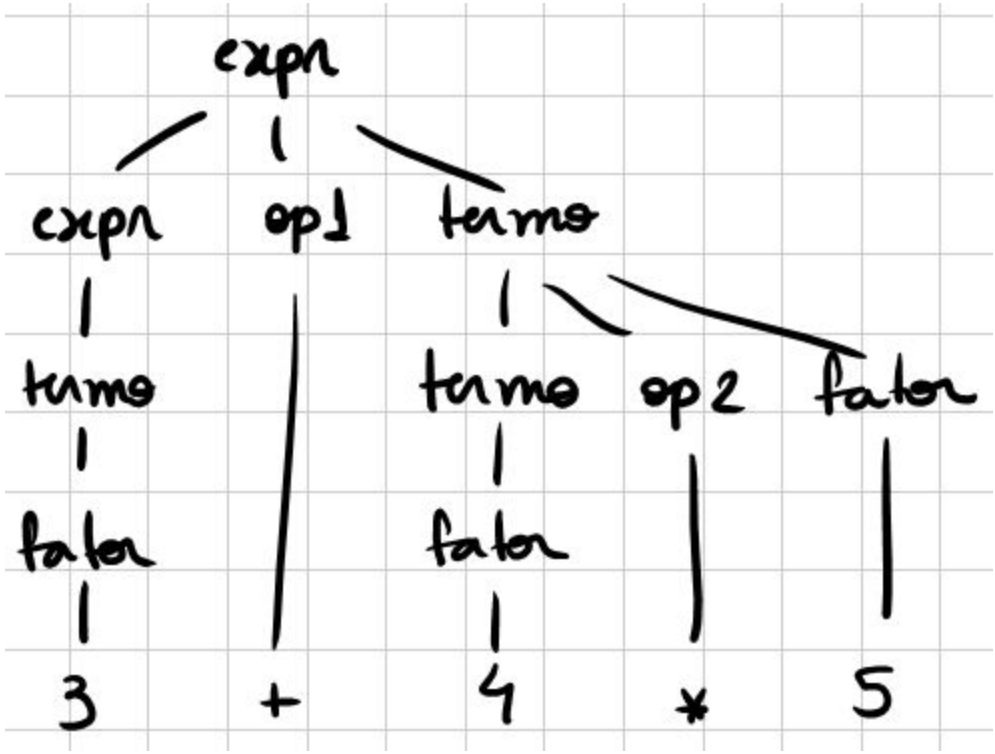
fator → '(' expr ')' | NUM

op1 → + | -

op2 → *

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
    - Em cada uma, inserimos a recursão conforme a associatividade (esquerda ou direita)

  $S \rightarrow S\ a\ S1 \mid S1$ $\qquad$ $S1 \rightarrow S1\ b\ S2 \mid S2$

  $S2 \rightarrow S3\ c\ S2 \mid S3$ $\qquad$ $S3 \rightarrow T\ d\ S3 \mid T$

  $T \rightarrow x$

Associatividade e precedência

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

Associatividade e precedência

- Primeiro passo:
  - Gramática ambígua

    expr → expr op expr | NUM

    op → + | - | * | / | % | ^

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

    op1 → + | -

    op2 → * | / | %

    op3 → ^

Associatividade e precedência

- Terceiro passo:
  - Forçando associatividade e precedência

    expr → expr op1 termo | termo

    termo → termo op2 fator | fator

    fator → NUM op3 fator | NUM

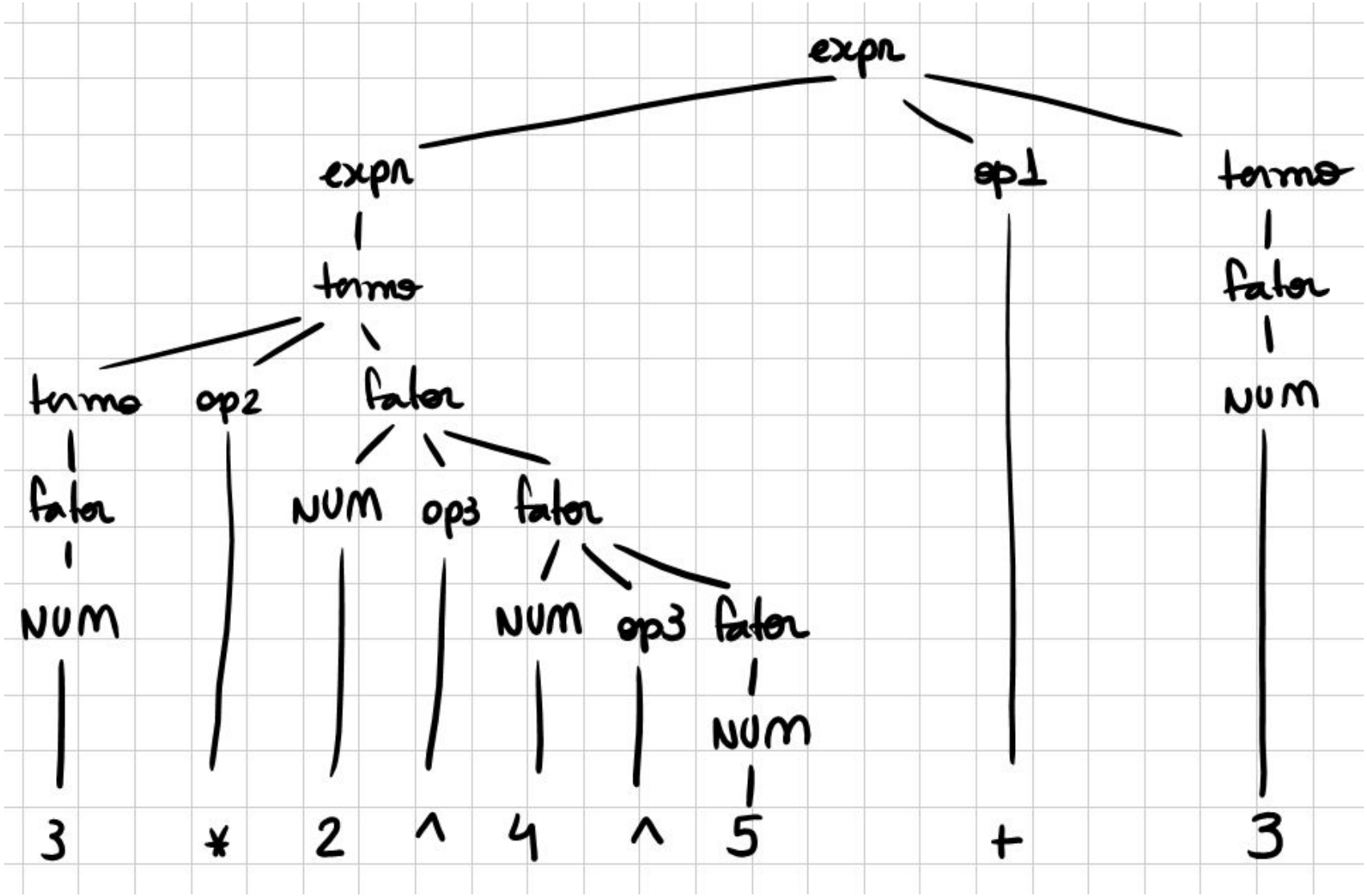
    op1 → + | -

    op2 → * | / | %

    op3 → ^

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

Associatividade e precedência



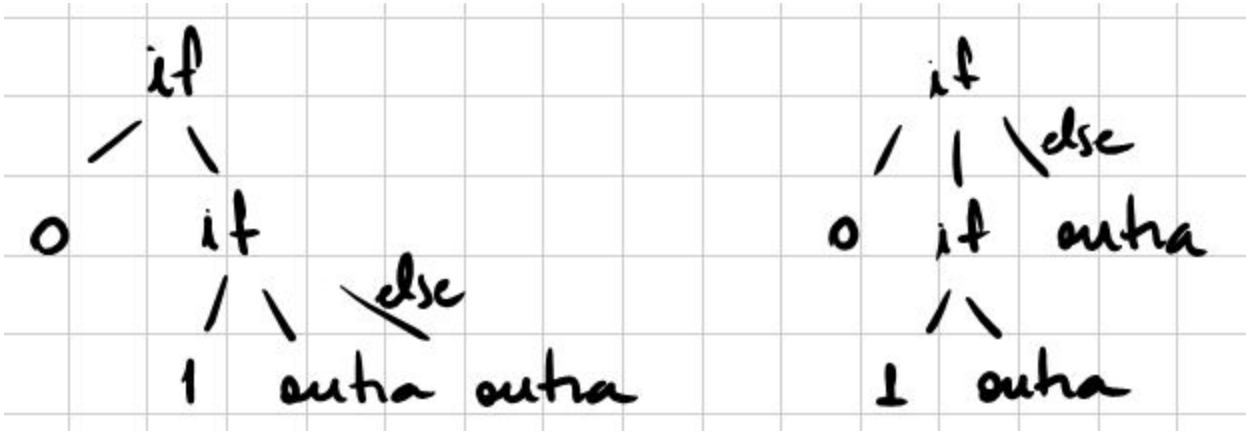
Eliminando ambiguidade

- Segunda técnica:
  - Modificar ligeiramente a linguagem

```
declaração → if-decl | outra
if-decl → if (exp) declaração |
    if (exp) declaração else declaração
exp → 0 | 1
```

- Verifique que há duas árvore para a seguinte cadeia

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



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

somente **casam-decl** 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

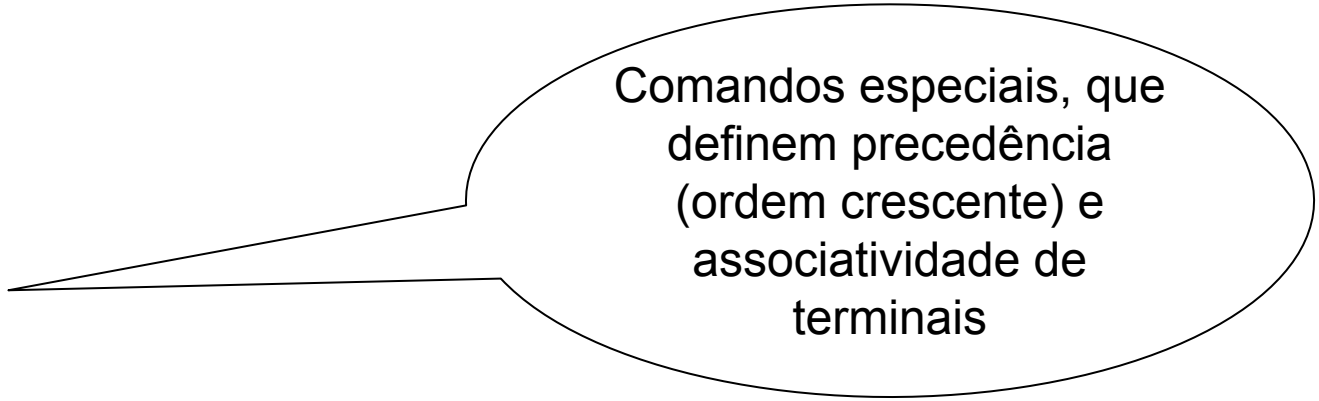
- Terceira técnica:
  - Inserir regras “extras” diretamente no analisador

```
declaração → if-decl | outra
if-decl → if (exp) declaração | if
  (exp) declaração else declaração
exp → 0 | 1
```

- Podemos 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 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
```

Eliminando ambiguidade

- Dada a entrada 3 + 4 * 5
- Após ler o símbolo “4”, o YACC está na configuração
  - E + E <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 (+)
  - O terminal mais à esquerda do seu lado direito (*)

Eliminando ambiguidade

- Ele então olha para:
  - O terminal mais à direita do seu lado esquerdo (+)
  - O terminal mais à esquerda do seu lado direito (*)

- Como * tem precedência sobre +
  - A decisão é não inferir neste momento
  - E continuar lendo

$$E + E * \langle YACC \rangle\ a \rightarrow E + E * a\ \langle YACC \rangle$$
$$\rightarrow E + E * E$$
$$\rightarrow E + E$$
$$\rightarrow E$$

Resumo: ambiguidade

- Nem sempre é possível remover a ambiguidade
- Não há algoritmo
  - Como aquele que remove não-determinismo em autômatos finitos

- Alguns exemplos (e suas soluções) são clássicos
  - Expressões aritméticas
  - If-then-else

- Resolver ambiguidades (não-determinismos/conflitos) também depende do algoritmo de análise sintática
  - Algoritmos LL tem uma certa forma
  - Algoritmos LR tem outra forma

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
  - Nesses casos, é necessário remover a recursão à esquerda
- Regra simples:
  - $A \rightarrow A\alpha \mid \beta$

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

Recursividade à esquerda

- Obs: É diferente de quando vimos o caso da associatividade dos operadores
  - Naquele exemplo, o objetivo era 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
    - $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid ... \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid ... \mid \beta_n$
  - Recursão não-imediata
    - $A \rightarrow B\beta \mid \ldots$
    - $B \rightarrow C\gamma \mid \ldots$
    - $C \rightarrow A\delta \mid ...$

Rec. imediata em uma produção

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

- Ex:
  - Antes:
    - expr → expr ‘+’ termo | termo
  - Depois
    - expr → termo expr2
    - expr2 → ‘+’ termo expr2 | ε

RI em mais de uma produção

- Primeiro, agrupe as produções da seguinte forma
  - $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid ... \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid ... \mid \beta_n$
  - Onde nenhum β_i começa com A, e nenhum α_i é ε
- Substitua as produções de A por
  - $A \rightarrow \beta_1 R \mid \beta_2 R \mid ... \mid \beta_n R$
  - $R \rightarrow \alpha_1 R \mid \alpha_2 R \mid \alpha_3 R \mid ... \mid \alpha_m R \mid \varepsilon$
- Ex:
  - Antes
    - expr → expr ‘+’ termo | expr ‘-’ termo | termo | constante
  - Depois
    - expr → termo expr2 | constante expr2
    - expr2 → ‘+’ termo expr2 | ‘-’ termo expr2 | ε

Recursão não-imediata

- Situação menos comum
- Algoritmo um pouco mais complicado
- Não veremos na disciplina

- Se algum dia se deparar com uma situação assim
  - Procure no livro do dragão!

Fatoração à esquerda

Fatoração à esquerda

- Quando a escolha entre duas produções não é clara
  - Pode-se reescrever as produções para adiar a decisão até haver entrada suficiente para a decisão

- Útil para deixar uma gramática adequada para análise sintática preditiva
  - Ex:
    - comando → if (expr) then cmd else cmd
    - comando → 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:
    - $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3 \mid ... \mid \alpha\beta_n$
  - Depois:
    - $A \rightarrow \alpha R$
    - $R \rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \mid ... \mid \beta_n$
- Exemplo:
  - Antes:
    - comando → if (expr) then cmd else cmd | if (expr) then cmd
  - Depois:
    - comando → if (expr) then cmd comandoElse
    - comandoElse → else cmd | ε

Neste exemplo:
α = if (expr) then cmd
β_1 = else cmd
β_2 = ε

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 usada 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

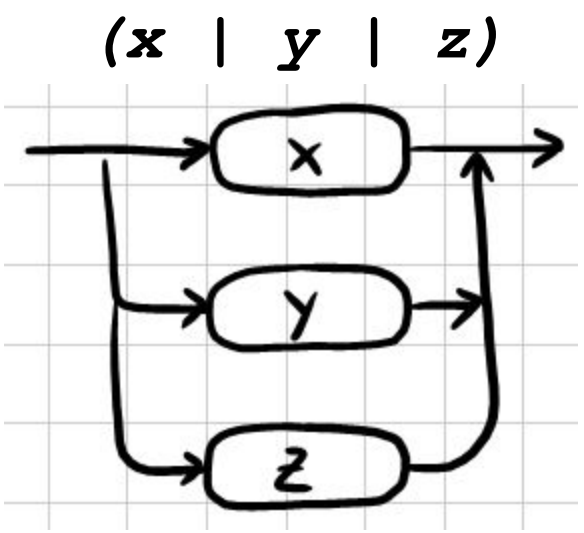
```
A → x? = A → x | ε
A → x* = A → xA | ε (ou A → Ax | ε)
A → x+ = A → xA | x (ou A → Ax | x)
```

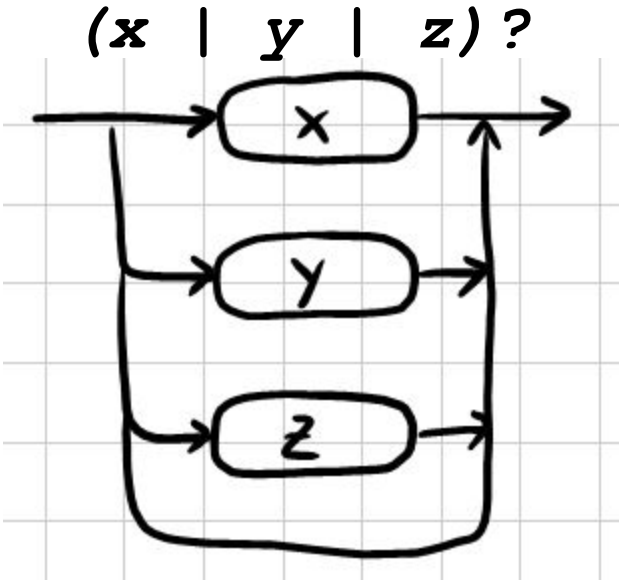
- Exs:

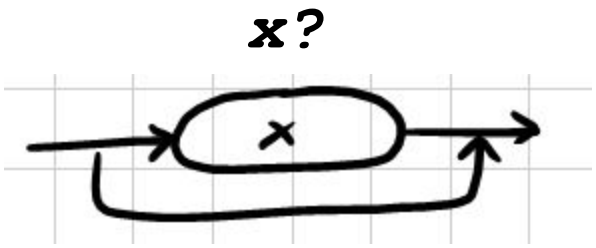
```
expr : termo (op1 termo)*
if-decl : 'if' '(' expr ')' 'then' cmd
   ('else' cmd)?
```

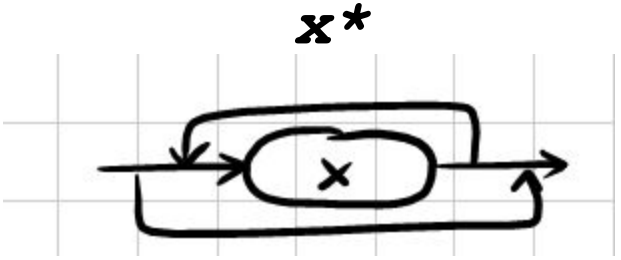
Diagramas sintáticos

- Ajudam a visualizar as regras





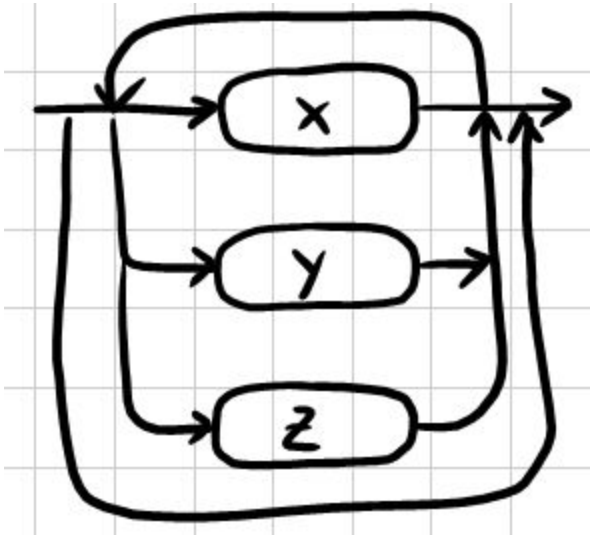




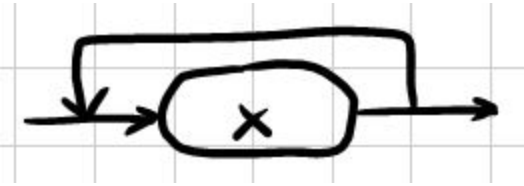
Diagramas sintáticos

- Ajudam a visualizar as regras

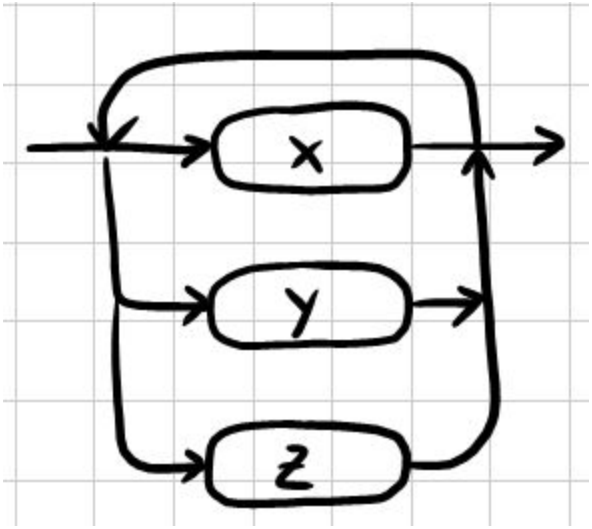
(x | y | z)*



x+

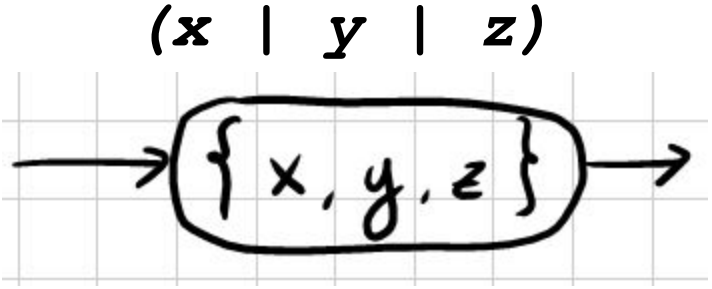


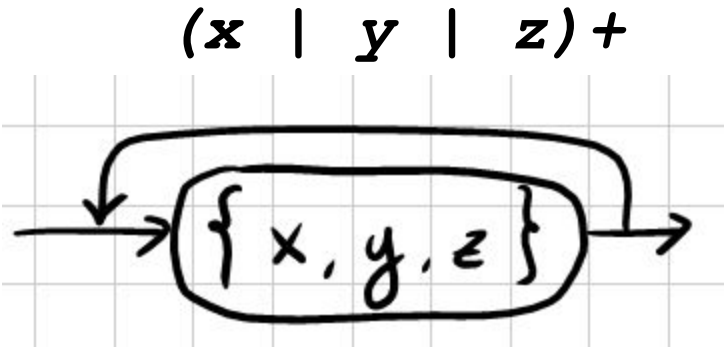
(x | y | z)+



Diagramas sintáticos

- Ajudam a visualizar as regras





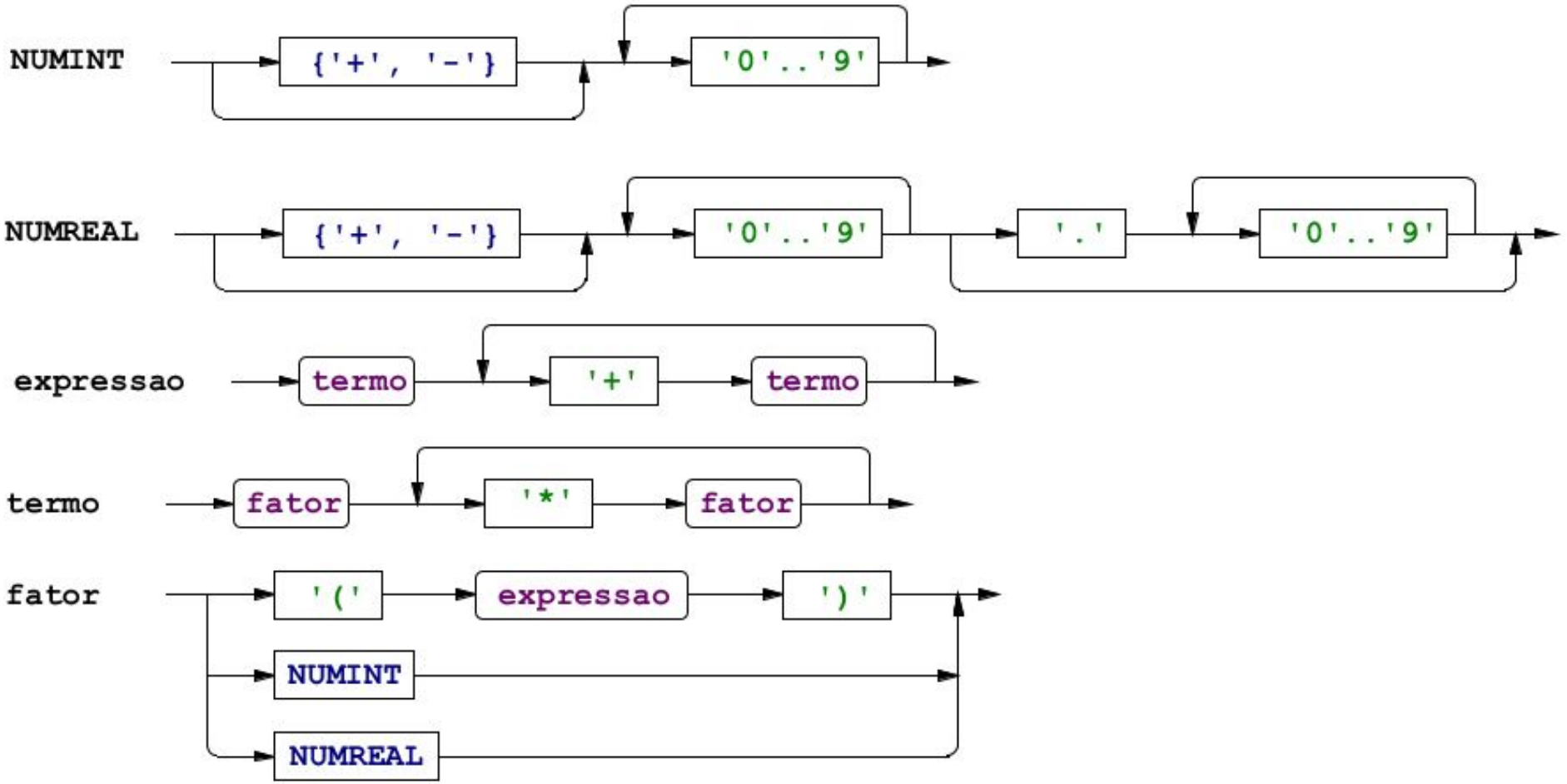
Diagramas sintáticos

- 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;
```

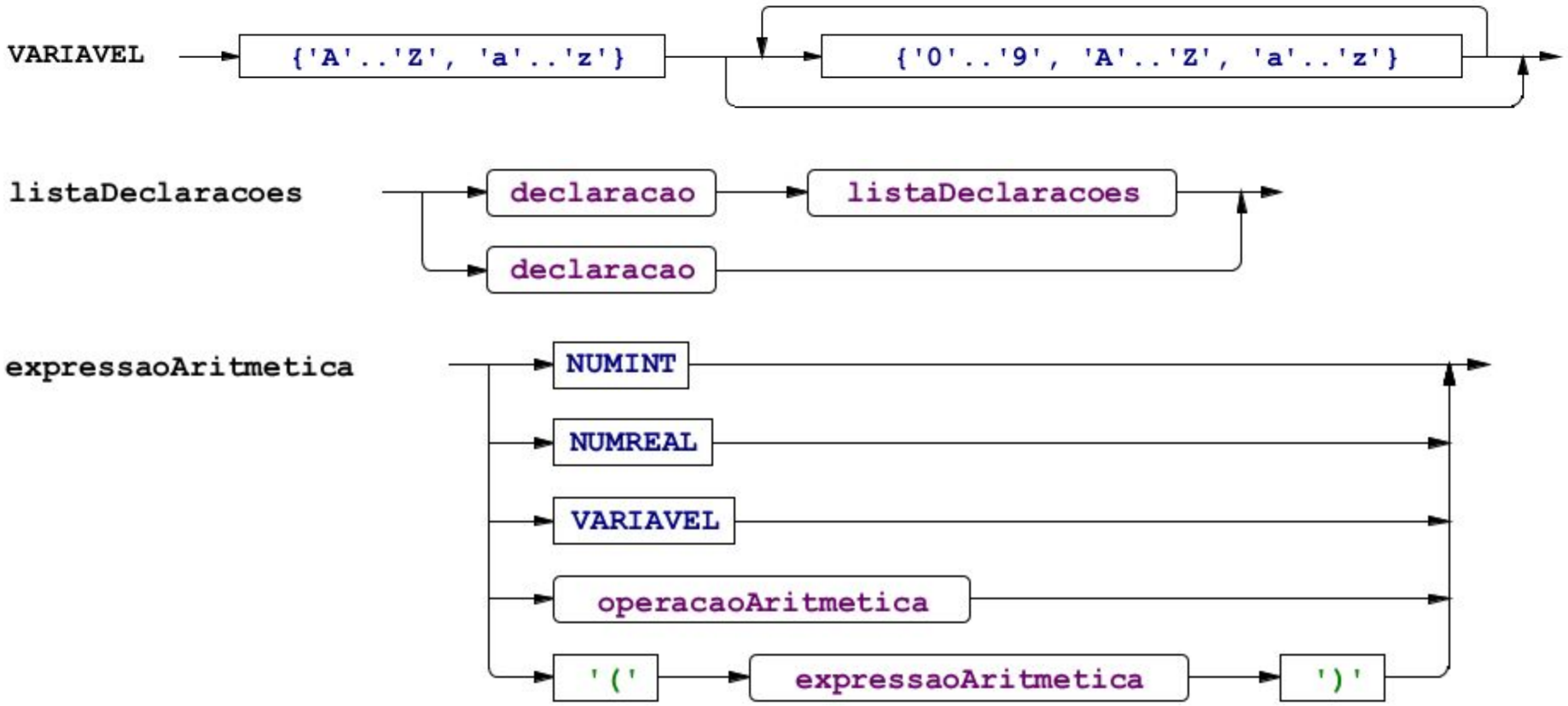
Diagramas sintáticos

- Resposta



Diagramas sintáticos

- Escreva as regras para os seguintes diagramas



Diagramas sintáticos

- Respostas

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

listaDeclaracoes : declaracao
   listaDeclaracoes | declaracao;

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

Fim