Construção de Compiladores

Prof. Dr. Daniel Lucrédio
DC - Departamento de Computação
UFSCar - Universidade Federal de São Carlos

Tópico 03 - Análise Sintática Introdução

Referências bibliográficas

Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. Compiladores: Princípios, Técnicas e Ferramentas (2a. edição). Pearson, 2008.

Kenneth C. Louden. Compiladores: Princípios E Práticas (1a. edição). Cengage Learning, 2004.

Terence Parr. The Definitive Antlr 4 Reference (2a. edição). Pragmatic Bookshelf, 2013.

Contexto

- Linguagem humana tem:

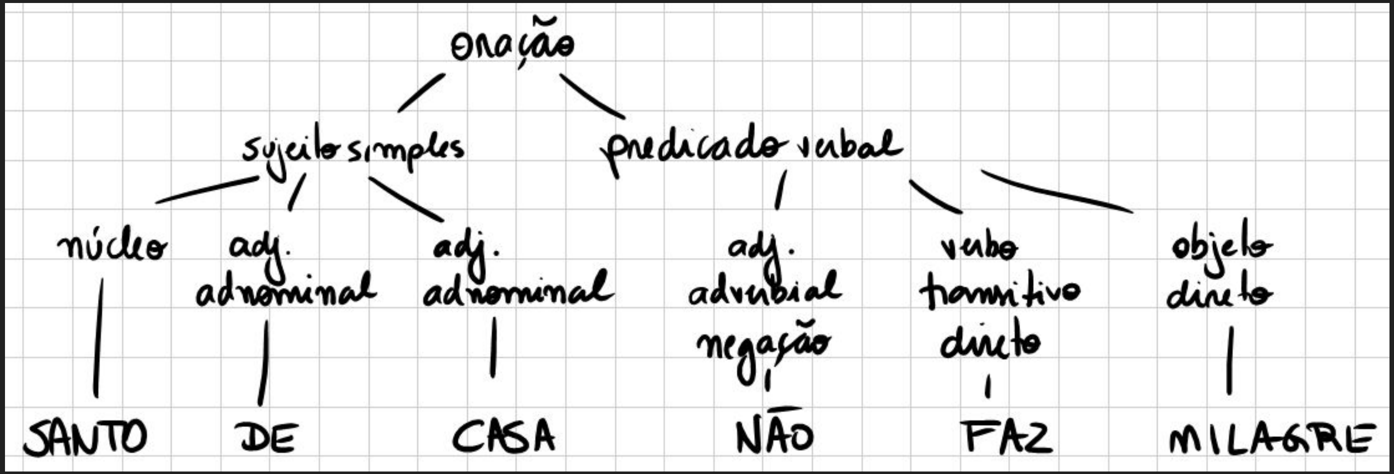
Vocabulário + Gramática

- Nomes das coisas

- Ações
- Composição
- Conceitos complexos

Objetivo

- Objetivo da análise sintática
  - Reconhecer a estrutura das frases

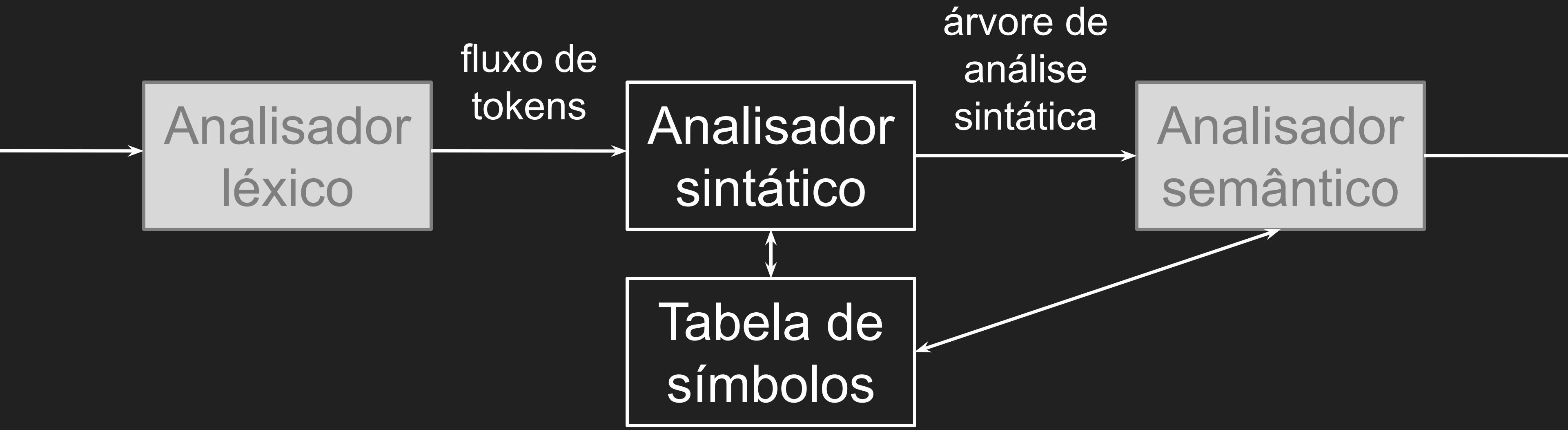


Em compiladores

- 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 Livres de Contexto
  - Permitem definir estruturas de cadeias de símbolos
    - (Exceto pelos nomes, como já visto na introdução)
  - Implementação simples associada (PDA)

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/não-terminais** e **terminais**

A variável que aparece do lado esquerdo da primeira regra é designada **variável inicial 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

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

$\Rightarrow$

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

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

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

$\Longrightarrow$

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

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

}

Análise sintática … recordando

- Inferência recursiva
  - Dada uma cadeia (conjunto de símbolos terminais)
  - Vamos 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

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

Análise sintática … recordando

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

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

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

- $E \rightarrow I \mid E + E \mid E * E \mid (E)$
- $I \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$

- 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: $\Rightarrow_{rm}$, $\overset{*}{\Rightarrow}_{rm}$

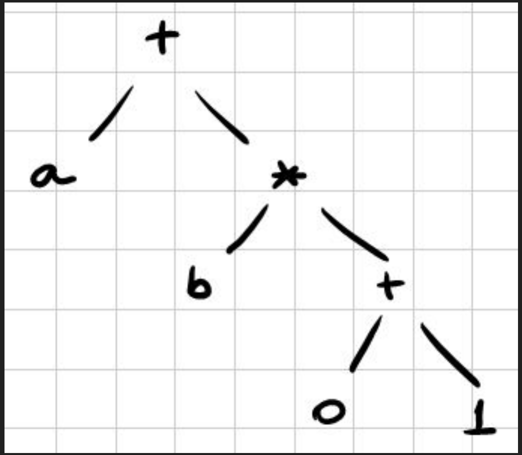
Árvores de análise sintática

- 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
  - $E \rightarrow I \mid E + E \mid E * E \mid (E)$
  - $I \rightarrow a \mid b \mid 0 \mid 1$

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:

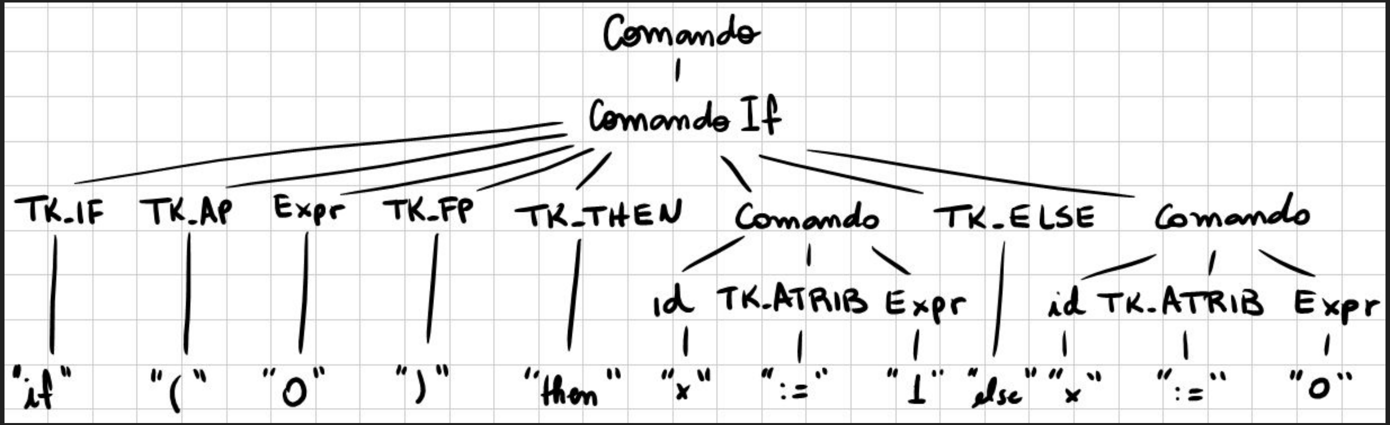
```
Comando → ComandoIf | (id TK_ATRIB Expr)

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

Árvores de sintaxe abstrata

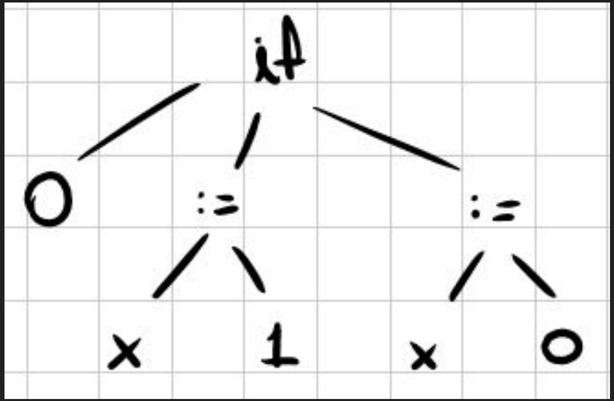
- Árvore de análise sintática (concreta) da cadeia:

```
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”

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;
```

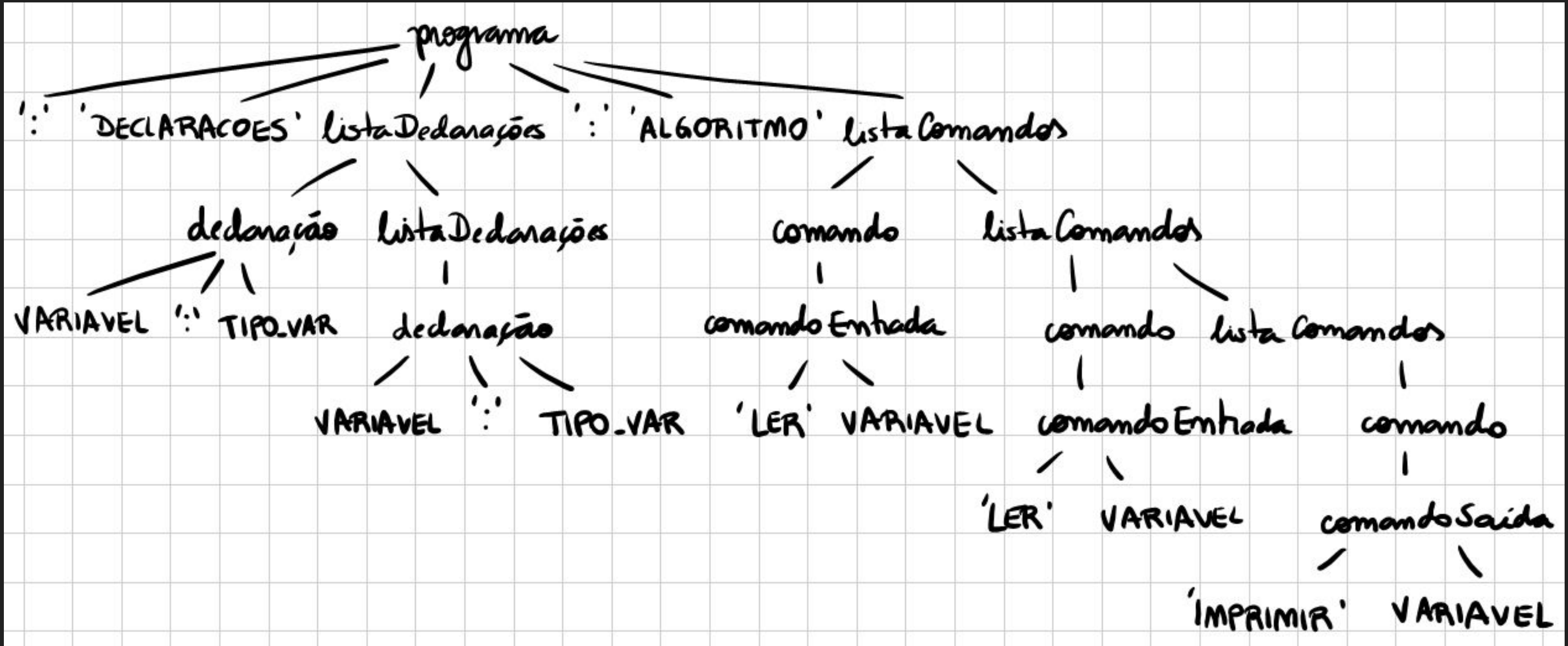
Exemplo

```
: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 }
```

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

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

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

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

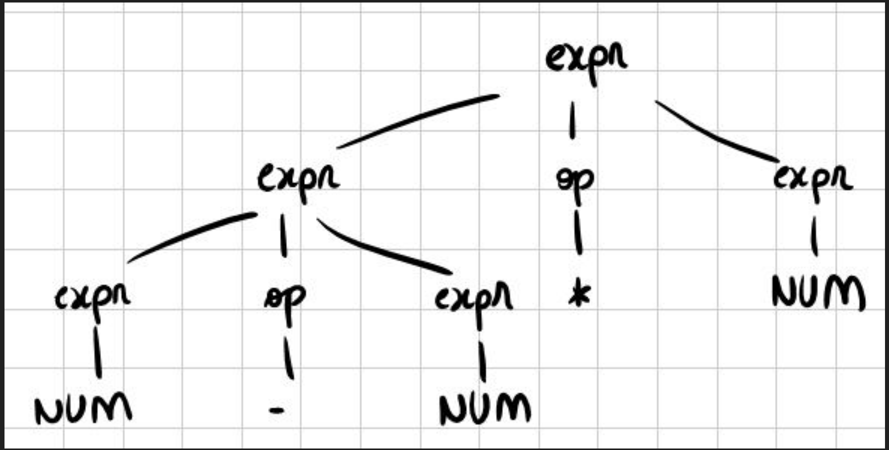
- Exemplo clássico: expressões aritméticas

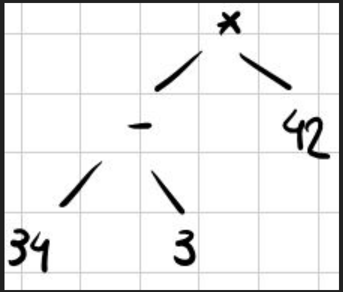
  expr → expr op expr | '(' expr ')' | NUM

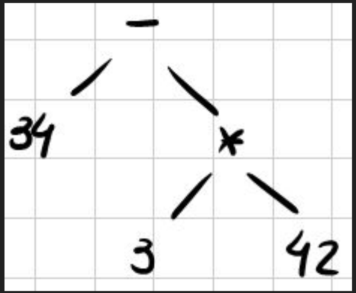
  op → + | - | *

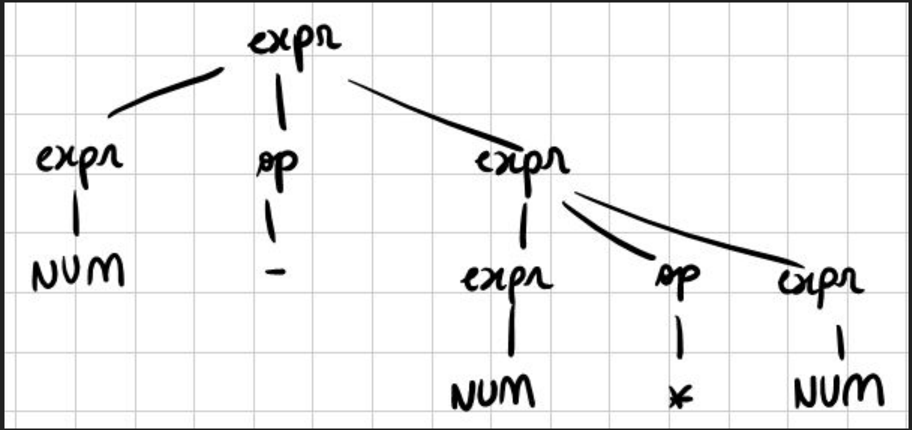
- É fácil demonstrar que existe mais de uma árvore de análise sintática para a cadeia 34 – 3 * 42
  - Também resultam em sintaxes abstratas diferentes

Eliminando ambiguidade



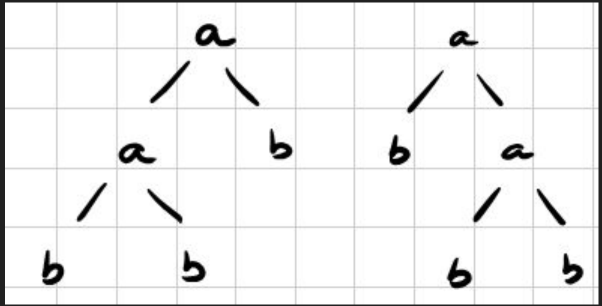






Eliminando ambiguidade

- 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

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

- Para “forçar” a associatividade à esquerda

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

- Para “forçar” a associatividade à direita

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:

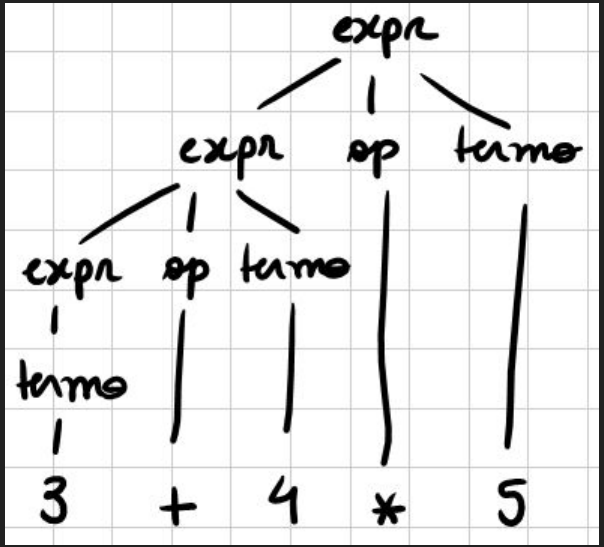
  expr → expr op termo | termo

  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

- Agora, todos os operadores têm a mesma precedência
  - Ou seja, vale a ordem em que aparecem na cadeia

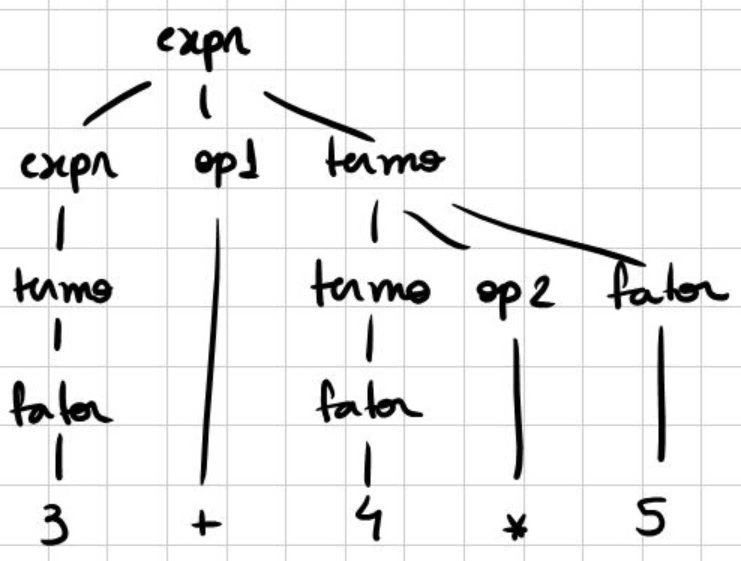
Eliminando ambiguidade

- 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)

$S \rightarrow S\ a\ S1 \mid S1$

$S1 \rightarrow S1\ b\ S2 \mid S2$

$S2 \rightarrow S3\ c\ S2 \mid S3$

$S3 \rightarrow T\ d\ S3 \mid T$

$T \rightarrow x$

Associatividade e precedência

- Exercício
  - Defina uma gramática para expressões aritméticas com 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

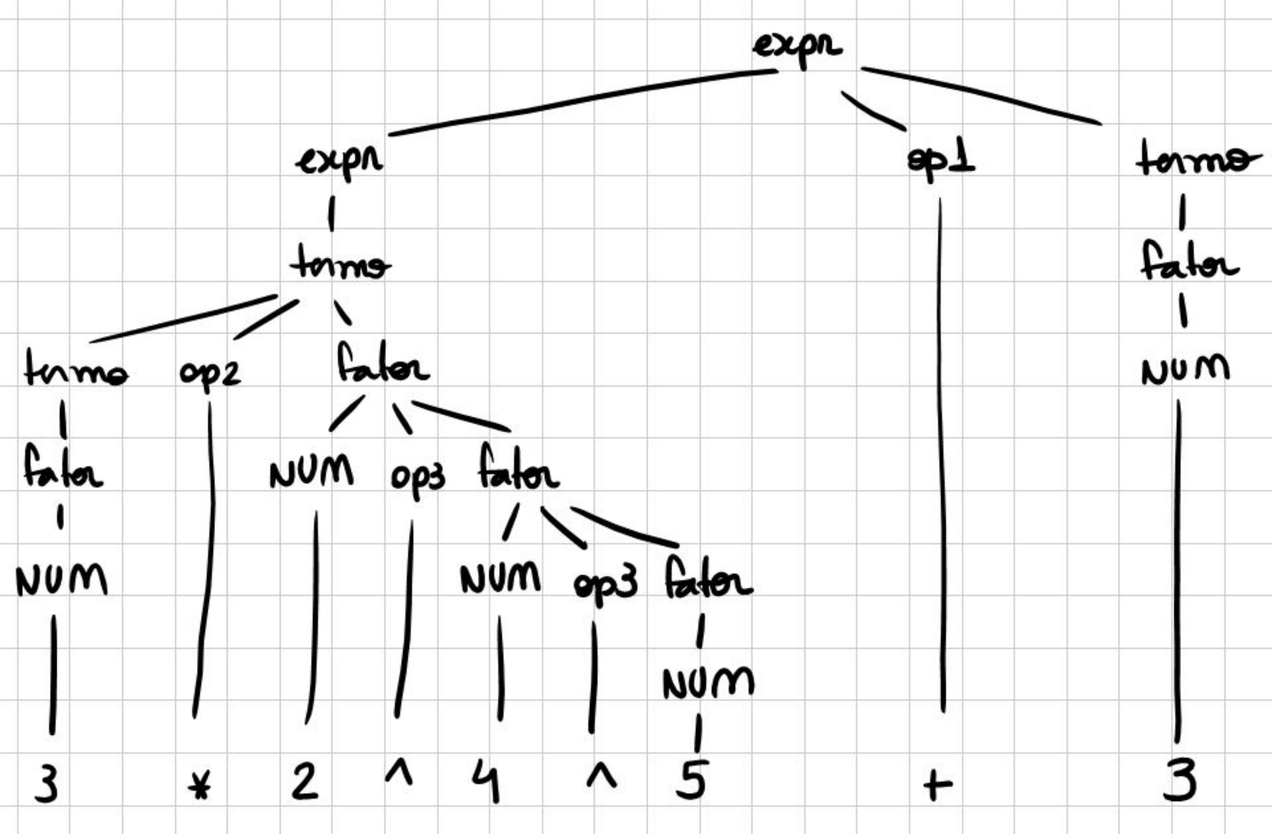
    op1 → + | -

    op2 → * | / | %

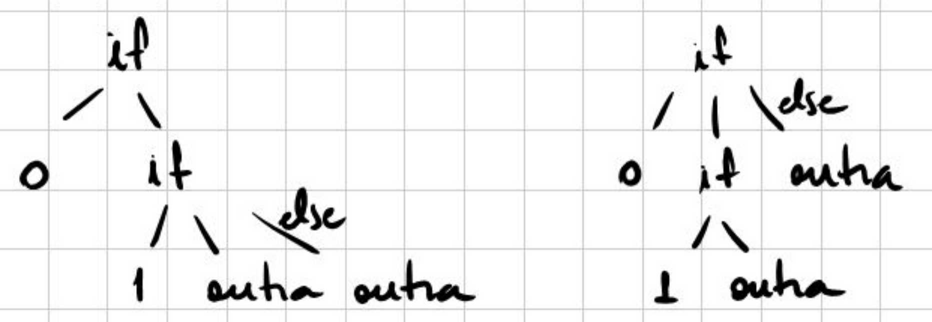
    op3 → ^

- 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:

Comandos especiais, que definem precedência (ordem crescente) e associatividade de terminais

```
%left '+'
%left '*'
E ::= E + E | E * E | NUM
```

Resumo: ambiguidade

- Nem sempre é possível remover a ambiguidade
- 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
  - $A \overset{*}{\Rightarrow} Ax$
- 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

- 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
    - $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$
  - Recursão não-imediata
    - $A \rightarrow B\beta \mid \ldots$
    - $B \rightarrow C\gamma \mid \ldots$
    - $C \rightarrow A\delta \mid \ldots$

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
  - $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$
  - Onde nenhum β_i começa com A, e nenhum α_i é ε
- Substitua as produções de A por
  - $A \rightarrow \beta_1 R \mid \beta_2 R \mid \ldots \mid \beta_n R$
  - $R \rightarrow \alpha_1 R \mid \alpha_2 R \mid \alpha_3 R \mid \ldots \mid \alpha_m R \mid \varepsilon$
- 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:
    - $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3 \mid \ldots \mid \alpha\beta_n$
  - Depois:
    - $A \rightarrow \alpha R$
    - $R \rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots \mid \beta_n$
- Exemplo:
  - Antes:
    - comando → if (expr) then cmd else cmd |
      if (expr) then cmd
  - Depois:
    - comando → if (expr) then cmd comandoElse
    - comandoElse → else cmd | ε

EBNF

EBNF - Extended Backus-Naur Form

- 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

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

$$A \to x^* = A \to xA \mid \varepsilon \text{ (ou } A \to Ax \mid \varepsilon)$$

$$A \to x+ = A \to xA \mid x \text{ (ou } A \to Ax \mid x)$$

- Exs:

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

Fim