Linguagens de Programação

Sintaxe e Semântica

Andrei Rimsa Álvares
andrei@cefetmg.br

Sumário

- Introdução
- Sintaxe
  - Gramáticas
  - Gramáticas BNF
  - Construção de Gramáticas
  - Estrutura Léxica
- Gramáticas e Prolog

INTRODUÇÃO

Linguagens de Programação

Introdução

- Como descrever uma linguagem de programação?
  - **Sintaxe:** como se vê o programa, sua forma e estrutura
    - A sintaxe pode ser definida através de **gramáticas formais**
  - **Semântica:** o que os programas fazem, seus comportamentos e significados
    - A semântica é mais difícil de se definir, mas existem formalismos para isso

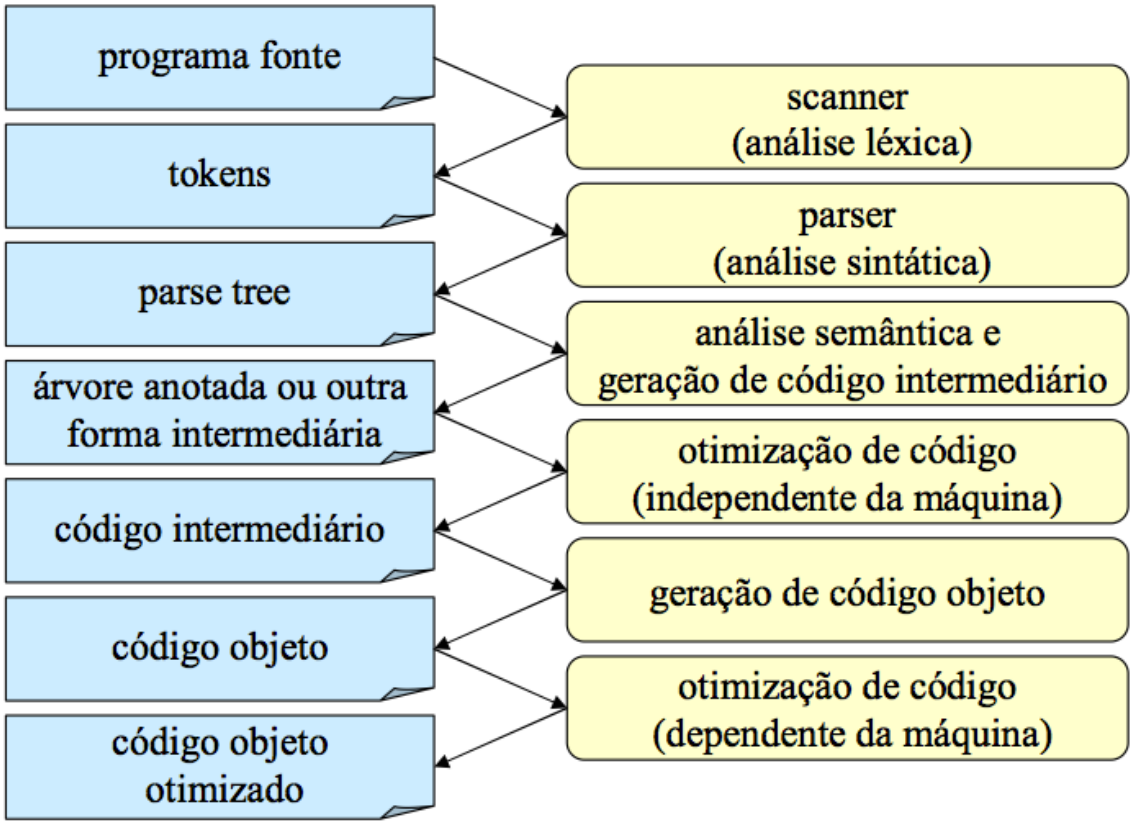
Sintaxe + Semântica: em conjunto definem uma linguagem

Linguagens

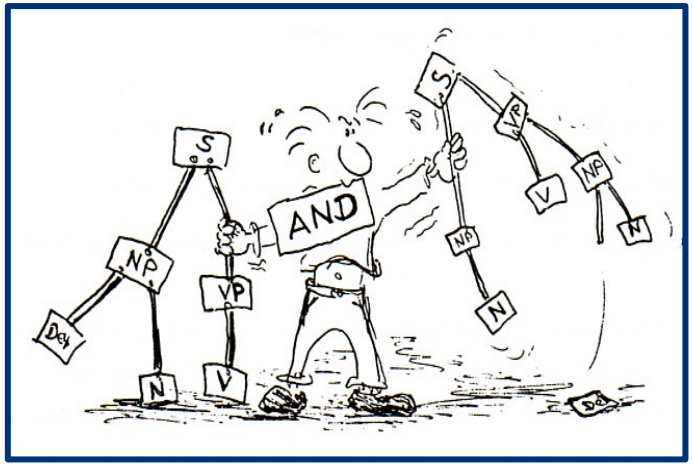
- As linguagens podem ser
  - **Reconhecidas** por um dispositivo capaz de ler texto de entrada sobre um alfabeto e decidir se pertence ou não à linguagem
    - **Ex.:** análise sintática feita por um compilador
  - **Geradas** por um dispositivo capaz de gerar sentenças de uma linguagem
    - **Ex.:** pode-se determinar se uma sentença é sintaticamente correta comparando-a com a estrutura do gerador

Fase de Compilação

- Um processo de compilação pode ser dividido nas seguintes fases:



SINTAXE
GRAMÁTICAS



Uma Gramática para Português

- Exemplo de formação de sentenças (simples) em português
  - **Sentença:** sintagma nominal + verbo + sintagma nominal

    `<S> ::= <NP> <V> <NP>`
  - **Sintagma nominal:** artigo + substantivo

    `<NP> ::= <A> <N>`
  - **Verbos:**

    `<V> ::= adora | odeia | come`
  - **Artigos:**

    `<A> ::= o | a | um | uma`
  - **Substantivos:**

    `<N> ::= cachorro | gato | rato`

Funcionamento da Gramática

- A gramática é um **conjunto de regras** que diz como se deve construir uma **árvore de derivação** (*parse tree*)

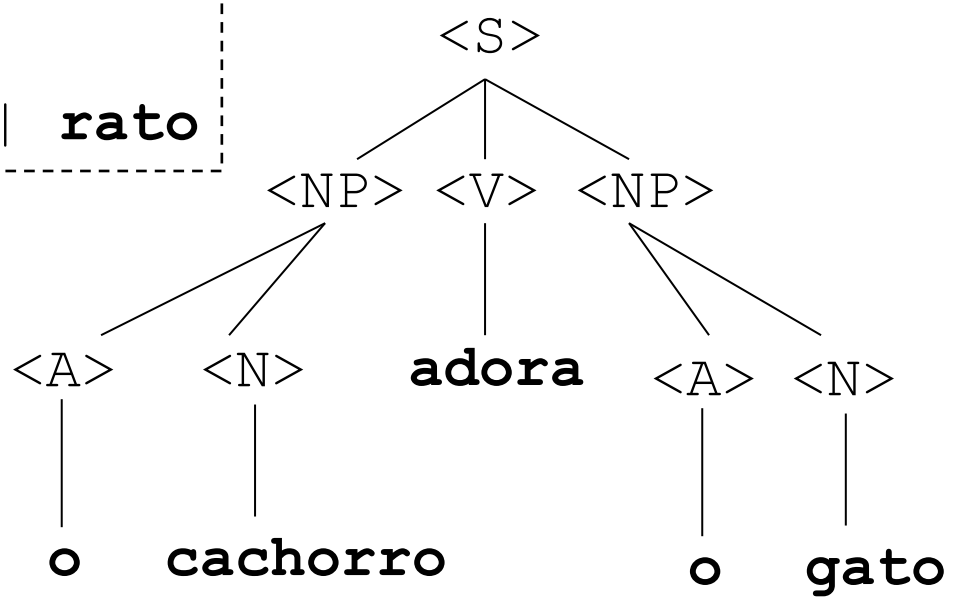
- O **símbolo de partida** (nesse exemplo <S>) deve ser colocado na raiz da árvore

- As **regras** da gramática dizem como os nós podem ser expandidos (derivação) em qualquer ponto da árvore
  - **Ex.:** a regra <S> **::=** <NP> <V> <NP> diz que se pode adicionar os nós <NP>, <V> e <NP>, nessa ordem, como filhos de <S>

Árvore de Derivação

- Exemplo de derivação para a gramática em português para o texto: **o cachorro adora o gato**

```
<S>  ::= <NP> <V> <NP>
<NP> ::= <A> <N>
<V>  ::= adora | odeia | come
<A>  ::= o | a | um | uma
<N>  ::= cachorro | gato | rato
```



Uma Gramática para uma LP

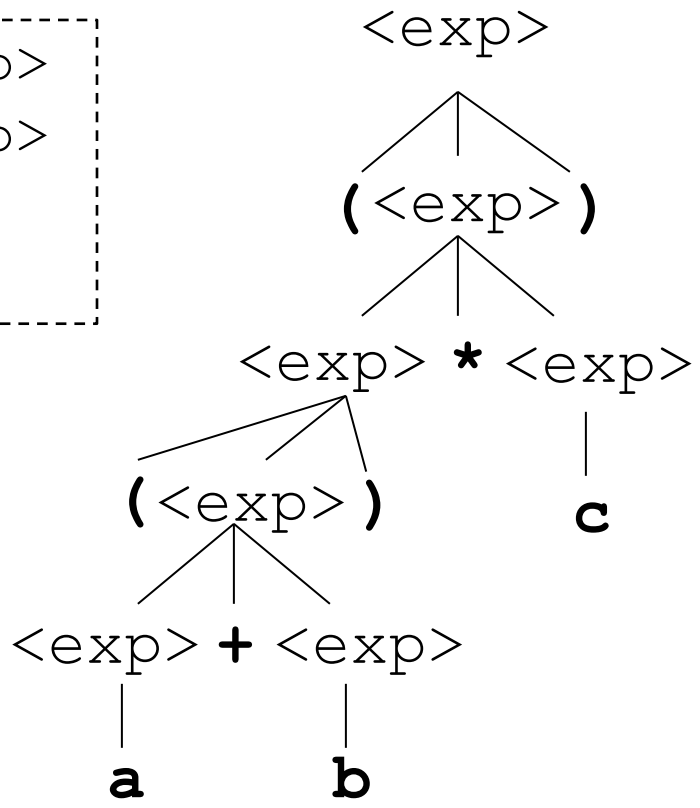
- Exemplo de definição de uma expressão de uma linguagem de programação, tal que uma expressão pode ser
  - uma soma de duas expressões
  - um produto de duas expressões
  - uma subexpressão com parênteses
  - uma das variáveis **a**, **b** ou **c**

```
<exp> ::= <exp> + <exp>
<exp> ::= <exp> * <exp>
<exp> ::= ( <exp> )
<exp> ::= a | b | c
```

Árvore de Derivação

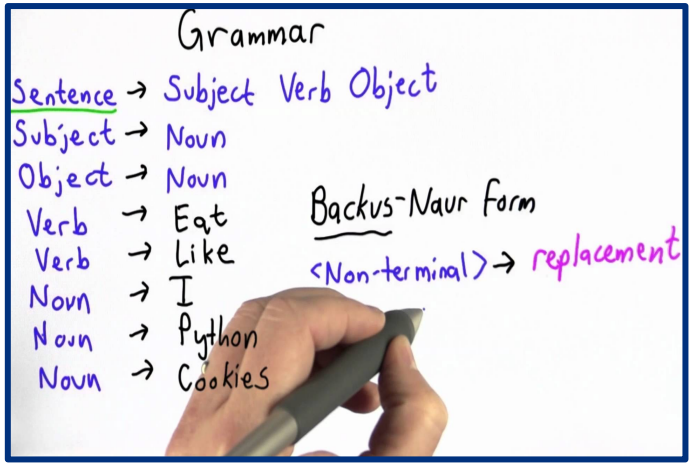
- Exemplo de derivação para a gramática de expressões de uma linguagem de programação para a expressão: `((a + b) * c)`

```
<exp> ::= <exp> + <exp>
<exp> ::= <exp> * <exp>
<exp> ::= ( <exp> )
<exp> ::= a | b | c
```



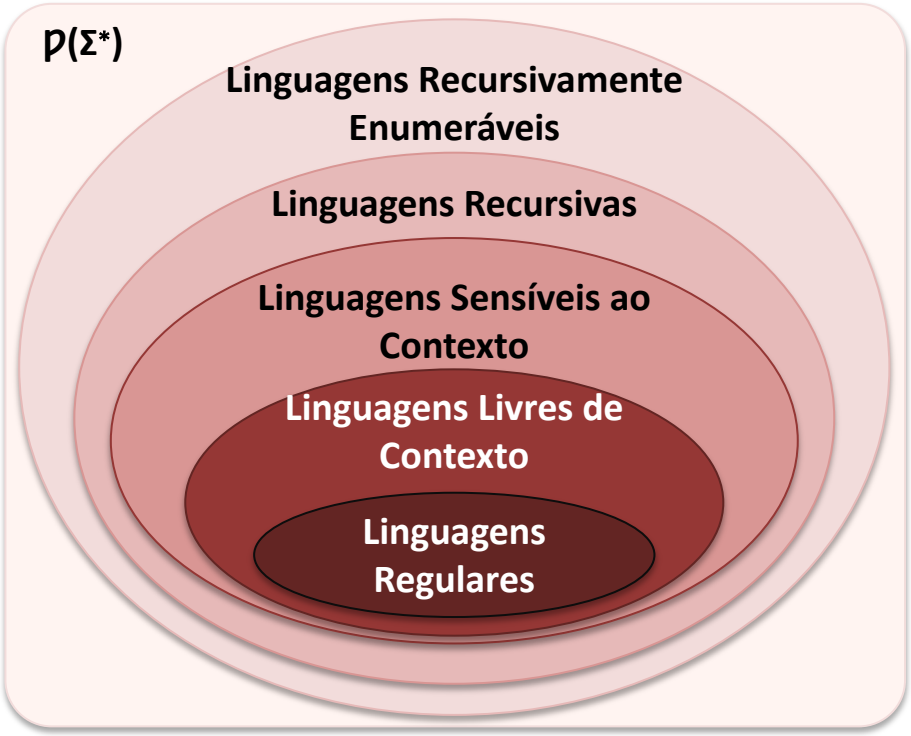
SINTAXE

GRAMÁTICA BNF (BACKUS-NAUR FORM)



Gramáticas Livres de Contexto

- Linguagens de programação pertencem à classe de **linguagens livres de contexto** (LLC) segundo a hierarquia de Chomsky (1950)

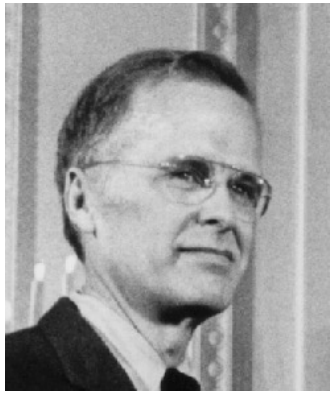


Hierarquia de Chomsky

- LLCs foram desenvolvidas para descrever a sintaxe de linguagens naturais
- LLCs podem ser geradas por gramáticas livres de contexto

Gramáticas BNF

- BNF (*Backus-Naur Form*) é uma gramática livre de contexto inventada por John Backus para descrever Algol 58

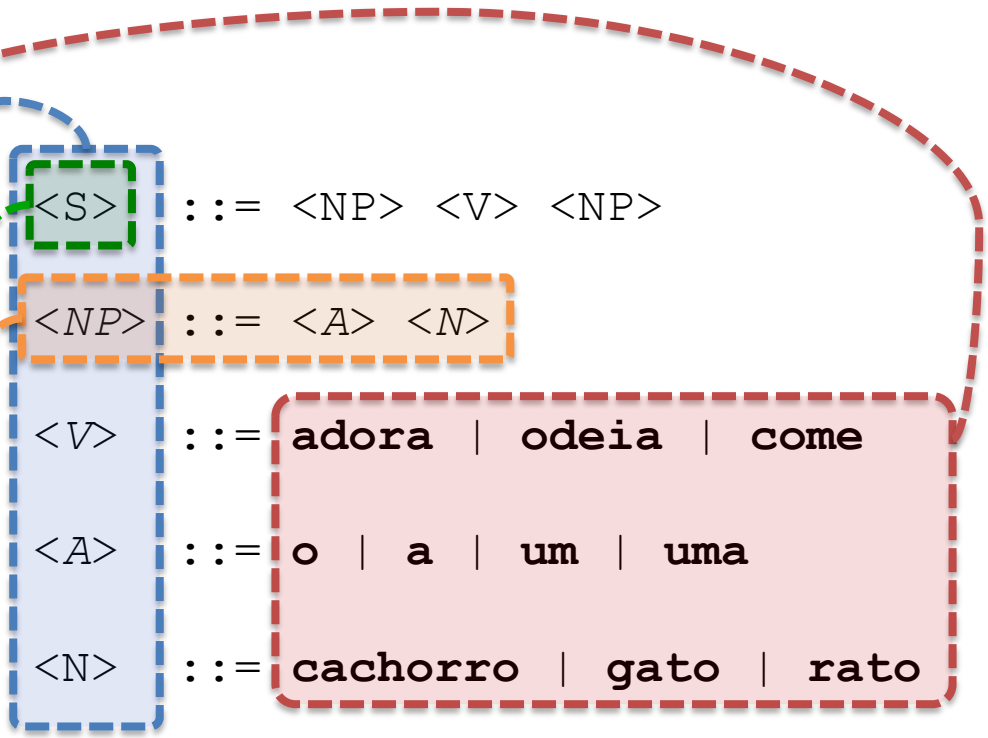


Esta notação é comumente utilizada para definir a sintaxe de linguagens de programação

Gramáticas BNF

- Uma gramática BNF é formada por quatro partes

1) Um conjunto de *tokens* (símbolos terminais)
2) Um conjunto de símbolos não-terminais
3) Um símbolo de partida
4) Um conjunto de produções (regras)



Gramáticas BNF

- Os ***tokens*** são as menores unidades de sintaxe
  - Cadeias de um ou mais caracteres (ex.: identificador, palavras reservadas, operadores, símbolos da linguagem)
  - São atômicos, ou seja, não são compostos de partes menores

- Os **símbolos não-terminais** são pedaços maiores de sintaxe
  - São cadeias colocadas entre chaves chevron (ex.: `<NP>`)
  - Não ocorrem literalmente no texto do programa
  - A gramática informa como eles podem ser expandidos em outros símbolos não-terminais e *tokens*

Gramáticas BNF

- O **símbolo de partida** é um símbolo não-terminal particular que forma a raiz de qualquer árvore de derivação para a gramática

- As **produções** são as regras de construção da árvore, tal que cada uma possui um ***left-hand side***, um separador **::=** e um ***right-hand side***

***Left-hand side* (*LHS*):** possui apenas um símbolo não-terminal

***Right-hand side* (*RHS*):** sequência de um ou mais símbolos não-terminais ou *tokens*

  - Uma produção permite uma única possibilidade de construir uma árvore de derivação: permite que um símbolo não-terminal no LHS seja expandido de acordo com o RHS, em ordem, como filhos da árvore

Forma Abreviada

- Quando se usa mais de uma produção com o mesmo *left-hand side*, pode-se usar uma forma abreviada

- A gramática BNF pode definir o *left-hand side*, o separador **::=** e então a lista de todos os possíveis *right-hand side* separados pelo símbolo especial | (barra em pé)

```
<exp> ::= <exp> + <exp>
        | <exp> * <exp>
        | ( <exp> )
        | a | b | c
```

Símbolo Não-Terminal Vazio

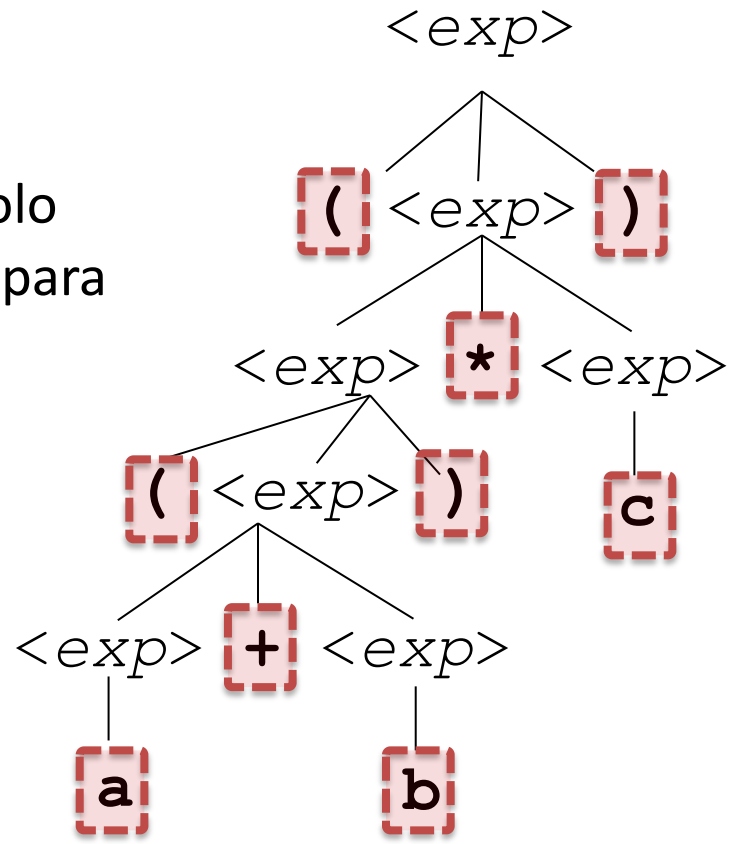
- BNF prevê um símbolo não-terminal especial <empty> ou ε que deve ser usado quando se deseja gerar nada

- Por exemplo, a gramática a seguir define um comando **if-then** típico com a parte **else** opcional

```
<if-stmt>   ::= if <expr> then <stmt> <else-part>
<else-part> ::= else <stmt> | <empty>
```

Árvore de Derivação

- Para construir uma árvore de derivação, coloque o símbolo de partida na raiz
- Expanda os nós em filhos para todo símbolo não-terminal, usando qualquer produção para esse não-terminal na gramática
  - Termine quando todas as folhas forem *tokens*
- Leia as folhas da esquerda para direita para obter a string que foi derivada da árvore



BNF Estendida (EBNF)

- A BNF estendida (EBNF) provê sintaxe adicional para simplificar algumas tarefas da gramática

  - `{x}`: zero ou mais repetições de `x`
  - `[x]`: opcional (ex.: `x | <empty>`)
  - `( )`: agrupamento de comandos
  - `|` : usado em qualquer lugar para escolha entre alternativas
  - `'|'`: aspas entre tokens para distinguir de metassímbolos da própria gramática

Exemplo EBNF

- Exemplos de regras usando BNF estendida (EBNF)

```
<if-stmt> ::= if <expr> then <stmt-list> [ else <stmt-list> ]
<stmt-list> ::= { <stmt> ; }
<thing-list> ::= { ( <stmt> | <declaration> ) ; }
<mystery1> ::= a[1]
<mystery2> ::= 'a[1]'
```

Compiladores

- O que foi feito é chamado de ***parsing***: tentar encontrar uma árvore de derivação para um determinado texto de entrada

- Isso é o que o compilador faz para todo programa que você tenta compilar: tenta construir uma árvore de derivação para seu programa usando a gramática da linguagem utilizada

Na disciplina de construção de compiladores você irá aprender algoritmos para fazer isso eficientemente

Definição de Linguagens

- Gramáticas são utilizadas para definir a **sintaxe** de linguagens de programação
- A linguagem definida por uma gramática é o **conjunto de todas as cadeias** que podem ser derivadas por alguma árvore de derivação para aquela gramática
- Normalmente, esse conjunto é infinito (embora a gramática seja finita)

Como construir gramáticas?

SINTAXE
CONSTRUÇÃO DE GRAMÁTICAS

Linguagens de Programação

Construção de Gramáticas

- A dica mais importante na construção de gramáticas é **dividir e conquistar**

- Considere a linguagem de declaração de identificadores em Java
  - Contém **um tipo**, **lista de nomes de variáveis separadas por vírgula** e **terminada com ;** (ponto e vírgula)
  - Cada variável pode, opcionalmente, ser seguida de uma expressão de inicialização

- Exemplos

```
float a;
```

```
boolean a, b, c;
```

```
int a=1, b, c=1+2;
```

Construção de Gramáticas

- É fácil definir a declaração se for adiada a definição da lista de variáveis separadas por vírgula com seus inicializadores

```
<var-dec> ::= <type-name> <declarator-list> ;
```

- A definição dos tipos (primitivos) é bastante simples

```
<type-name> ::= boolean | byte | short | int |
                long | char | float | double
```

Não foram considerados tipos construídos, como classes, interfaces e arranjos

Como fazer a regra <declarator-list>?

Construção de Gramáticas

- Agora, pode-se adiar a definição das variáveis com inicializadores e fazer apenas a lista de variáveis separadas

```
<declarator-list> ::= <declarator>
                    | <declarator> , <declarator-list>
```

Listas são implementadas com recursividade

  - Ou na forma EBNF

```
<declarator-list> ::= <declarator> { , <declarator> }
```

Como fazer a regra <declarator>?

Construção de Gramáticas

- Por fim, basta criar a regra para definição das variáveis com seus respectivos inicializadores

```
<declarator> ::= <variable-name>
                 | <variable-name> = <expr>
```

  - Ou na forma EBNF

```
<declarator> ::= <variable-name> [ = <expr> ]
```

Gramáticas e Linguagens

- Para uma mesma linguagem, pode-se construir diferentes gramáticas
  - Ex.: linguagem que contém uma ou mais variáveis (a, b, c) separadas pelo símbolo – (menos)

G_1: <subexp> ::= a | b | c | <subexp> - <subexp>

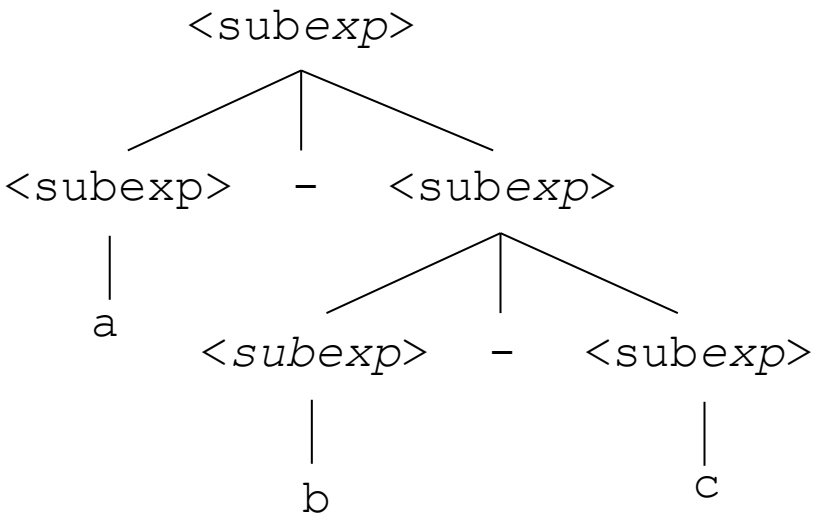
G_2: <subexp> ::= <var> - <subexp> | <var>
<var> ::= a | b | c

G_3: <subexp> ::= <subexp> - <var> | <var>
<var> ::= a | b | c

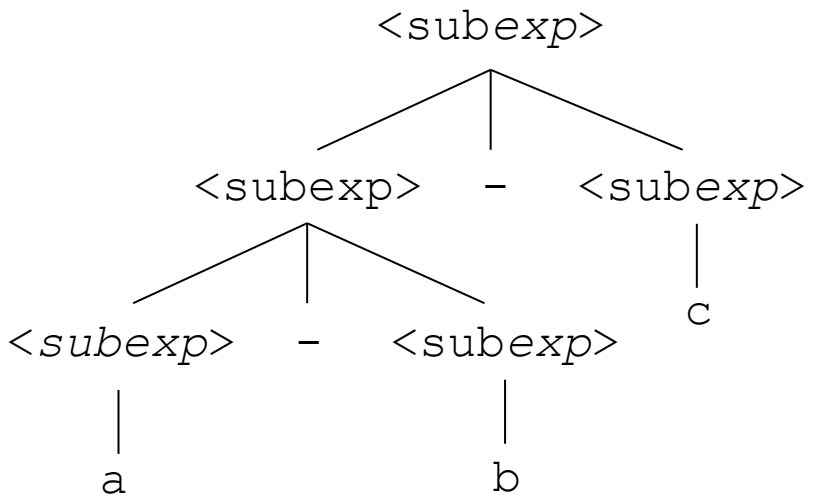
Qual delas usar?

Gramáticas Ambíguas

- Uma gramática é **ambígua** se gera uma forma sentencial que possui duas ou mais árvores de derivação
  - Ex.: a sentença: `a-b-c` usando a gramática `G1`



Uma árvore de derivação para G_1

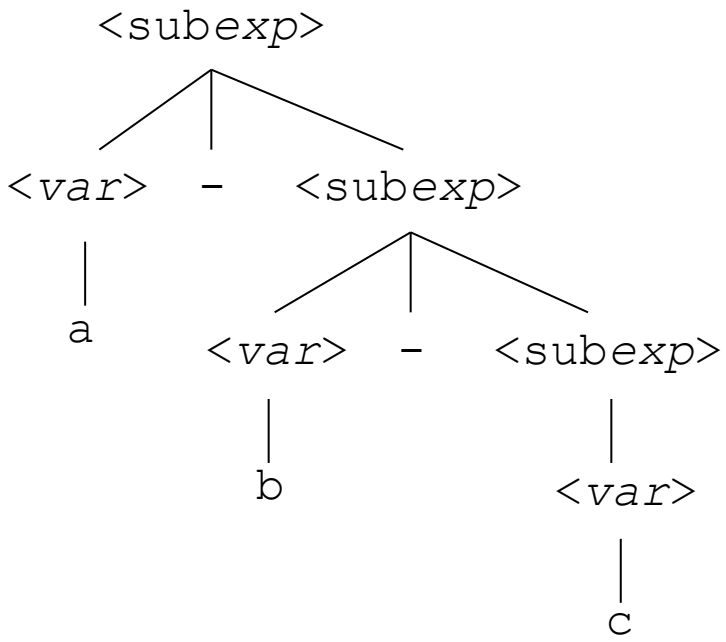


Uma outra árvore de derivação para G_1

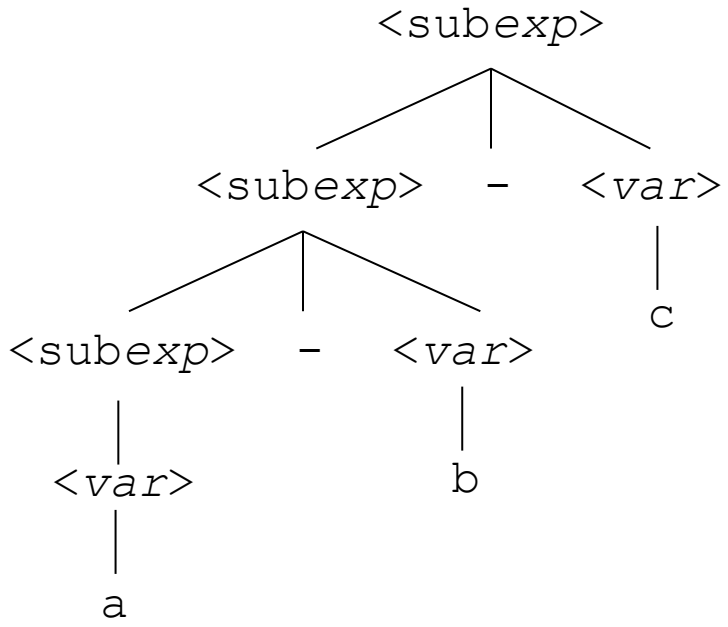
Qual o problema de se ter uma gramática ambígua?

Gramáticas "Equivalentes"

- O objetivo é gerar gramáticas cuja estrutura da árvore de derivação corresponda à semântica do texto gerado
  - Isso torna a construção da gramática mais difícil: estamos interessados na estrutura da gramática e não só nas derivações



Árvore de derivação para G_2



Árvore de derivação para G_3

Operadores

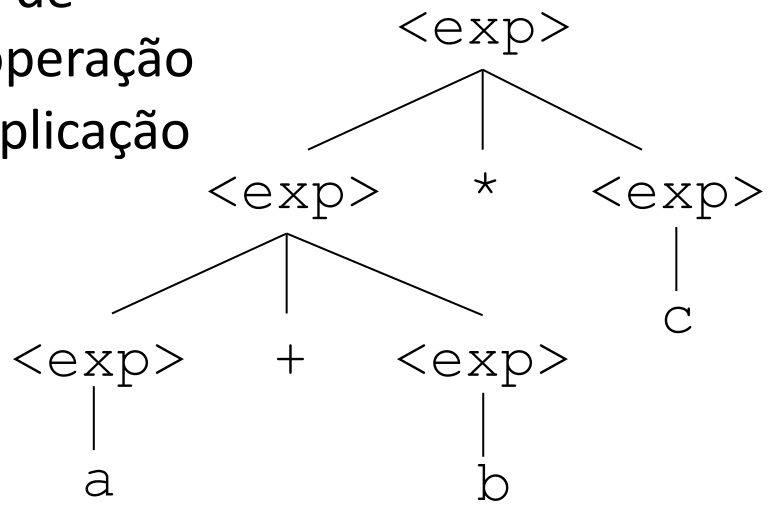
- Sintaxe especial para operações simples frequentemente utilizadas, como adições (`+`), subtrações (`-`), multiplicações (`*`) e divisões (`/`)
  - A palavra **operador** refere tanto ao símbolo usado para especificar as operações quanto ao significado da operação
- **Operandos** são as entradas para um operador: como `1` e `2` na expressão `1+2`
  - Operadores **unários** tomam um operando (ex.: `-1`)
  - Operadores **binários** tomam dois operandos (ex.: `1+2`)
  - Operadores **ternários** tomam três operandos (ex.: `a?b:c`)
- Na maioria das linguagens, operadores binários usam notação in-fixa (`1+2`), enquanto operadores unários pré-fixa (`-1`) ou pós-fixa (`a++`)

Problemas com Precedência

- Considere a seguinte gramática para expressões aritméticas usando **parênteses**, operadores + e *, e as variáveis **a**, **b** e **c**

```
G4: <exp> ::= <exp> + <exp>
            | <exp> * <exp>
            | ( <exp> )
            | a | b | c
```

- Essa gramática gera a seguinte árvore de derivação para a+b*c que efetua a operação de adição antes da operação de multiplicação



Precedência de Operadores

- A precedência de operadores é aplicada quando a ordem de avaliação não é decidida completamente pelo uso de parênteses

- Cada operador tem um **nível de precedência** e aqueles com precedência mais alta devem ser avaliados antes daqueles de menor precedência (como se tivesse parênteses)

- As linguagens de programação colocam a multiplicação com maior precedência que a adição (+) de forma que a semântica seja a de:

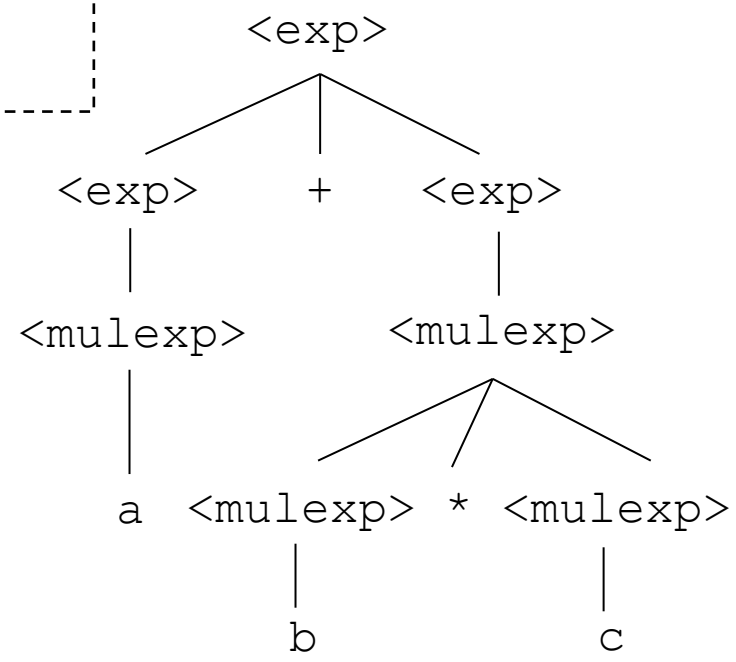
```
a+b*c = a+(b*c)
```

Precedência Correta

- Para consertar o problema da precedência, pode-se modificar a gramática de forma a forçar que a operação de multiplicação (*) seja colocada após o operador de adição (+) na árvore de derivação

```
G5: <exp>    ::= <exp> + <exp> | <mulexp>
    <mulexp> ::= <mulexp> * <mulexp>
                 | ( <exp> )
                 | a | b | c
```

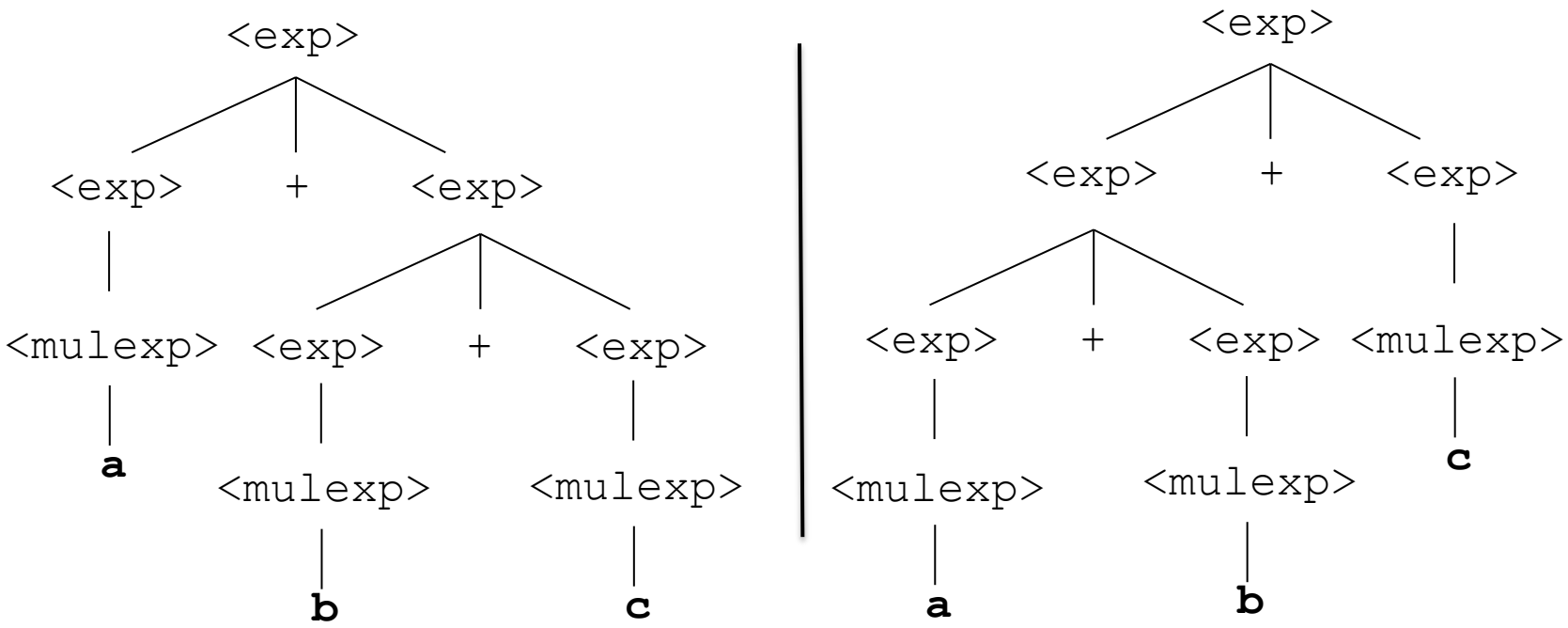
- A nova gramática gera a mesma linguagem da anterior, mas com a diferença que a árvore de derivação para `a+b*c` exibe a precedência correta



Esta gramática ainda tem algum problema?

Problemas com Associatividade

- A gramática G_5 gera as duas árvores seguintes para `a+b+c`, tal que a primeira não é a forma convencional de associatividade para o operador de adição



Cuidado: a gramática G_5 ainda é ambígua

Associatividade de Operadores

- Associatividade é aplicada quando a ordem de avaliação não é decidida por parênteses nem por precedência
  - Operadores de **associatividade à esquerda** agrupam da esquerda para a direita: `a+b+c+d = ((a+b)+c)+d`
  - Operadores de **associatividade à direita** agrupam da direita para a esquerda: `a+b+c+d = a+(b+(c+d))`

- A maioria dos operadores de uma linguagem são associativos à esquerda, mas existem exceções

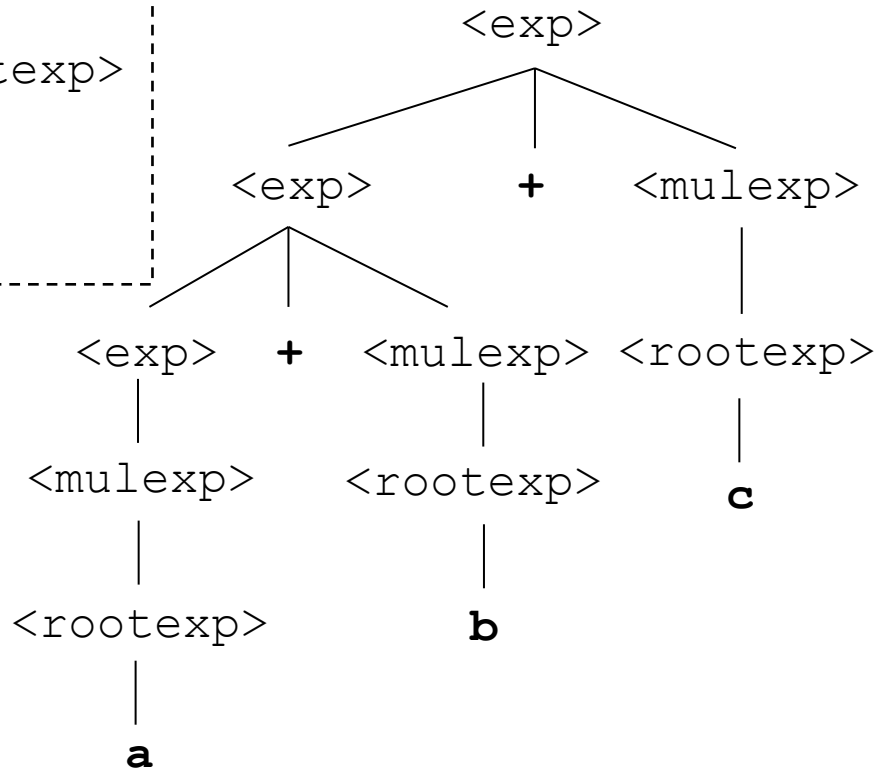
– C++	– ML
`a<<b<<c` (associativos à esquerda)	`3-2-1` (associativos à esquerda)
`a=b=0` (associativos à direita)	`1::2::nil` (associativos à direita)

Associatividade Correta

- Para resolver o problema de associatividade, pode-se modificar a gramática para que as árvores de `+ s` cresçam para baixo e esquerda (também para `* s`)

```
G6: <exp>       ::= <exp> + <mulexp>
                      | <mulexp>
    <mulexp>    ::= <mulexp> * <rootexp>
                      | <rootexp>
    <rootexp>   ::= ( <exp> )
                       | a | b | c
```

- A nova gramática gera a mesma linguagem da anterior, mas com a diferença que a árvore de derivação para `a+b+c` exibe associatividade correta



Exercício

- Para a gramática G_6, como faria para

```
G6: <exp>      ::= <exp> + <mulexp>
                     | <mulexp>
    <mulexp>   ::= <mulexp> * <rootexp>
                     | <rootexp>
    <rootexp>  ::= ( <exp> )
                     | a | b | c
```

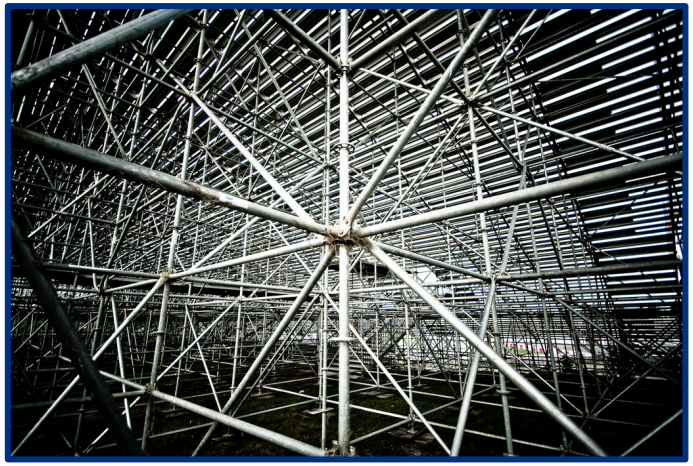
1) Adicionar o operador `&` (E aritmético) com **menor precedência** que a soma com **associatividade à esquerda**?
2) Adicionar o operador `**` (potência) com **maior precedência** que a multiplicação com **associatividade à direita**?

Solução do Exercício

- Gramática modificada G_7 com os operadores `&` (com menor precedência que a soma e associatividade à esquerda) e `**` (com maior precedência que a multiplicação e associatividade à direita)

```
G7: <exp>       ::= <exp> & <addexp>
                      | <addexp>
    <addexp>    ::= <addexp> + <mulexp>
                      | <mulexp>
    <mulexp>    ::= <mulexp> * <powerexp>
                      | <powerexp>
    <powerexp>  ::= <rootexp> ** <powerexp>
                      | <rootexp>
    <rootexp>   ::= ( <exp> )
                      | a | b | c
```

SINTAXE
ESTRUTURA LÉXICA



Tokens

- ***Tokens*** são pedaços de programa atômicos, ou seja, não se considera que são formados por outras partes menores
- Exemplos de *tokens* são: **identificadores** (`count`), **palavras reservadas** (`if`), **operadores** (`==`), **símbolos** (`;`), **constantes** (`123.4`)
- Contudo, programas em sua forma textual são apenas sequências de caracteres armazenadas em arquivos

Como dividir tal sequência de dados em uma sequência de *tokens*?

Estrutura Léxica vs. Sintática

- Até agora, gramáticas foram usadas apenas para definir a **estrutura sintática**: como que um programa pode ser construído a partir de uma sequência de *tokens*

- Também é preciso definir a **estrutura léxica**: como que um programa em forma de texto pode ser dividido em *tokens*

Uma Gramática Para a Todos Governar

- Pode-se usar apenas uma gramática para definir tanto a estrutura sintática quanto léxica, usando caracteres como *tokens*

```
<if-stmt>    ::= if <white-space> <expr> <white-space>
                 then <white-space> <stmt>
                      <white-space> <else-part>
<else-part> ::= else <white-space> <stmt> | <empty>
```

Abordagem pouco prática: espaços em branco e comentários deixam a gramática menos legível

Uma Gramática Separada

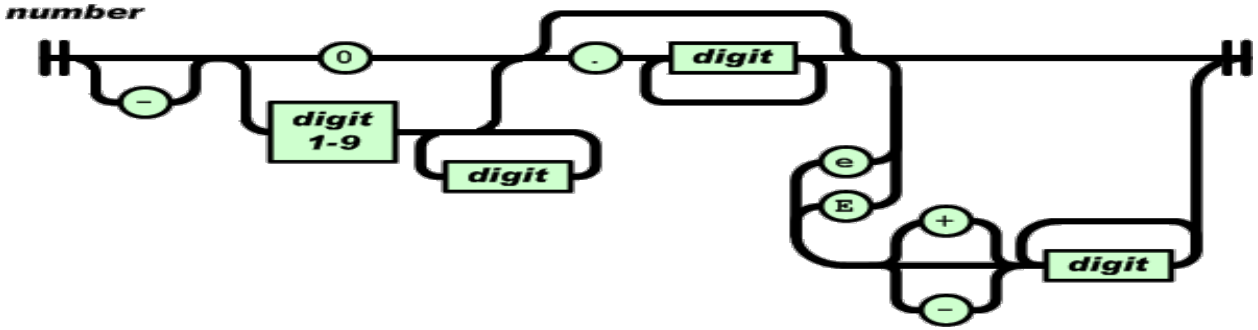
- Usualmente existem duas gramáticas separadas
  1) Uma para descrever como construir uma sequência de *tokens* a partir de uma sequência de caracteres
  2) Outra para descrever como construir uma árvore de derivação para uma sequência de *tokens*

- Exemplo

```
<program-file>    ::= <end-of-file>
                        | <element> <program-file>
<element>         ::= <token> | <one-white-space> | <comment>
<one-white-space> ::= <space> | <tab> | <end-of-line>
<token>           ::= <identifier> | <operator> | <constant> | …
```

Um Passe de Compilação Separado

- Um ***scanner*** lê o arquivo de entrada e o divide em uma sequência de *tokens* de acordo com a primeira gramática
- O *scanner* ignora espaços em branco e comentários
- O ***parser*** então constrói a árvore de derivação a partir do fluxo de *tokens* de acordo com a segunda gramática



Exemplo de número em JSON: token numérico em ponto flutuante

Um Pouco de História

- As primeiras linguagens não separavam a estrutura léxica da sintática
  - Os primeiros dialetos de Fortran e Algol permitiam espaços em qualquer lugar, mesmo no meio das palavras reservadas
  - Outras linguagens como PL/I permitiam que palavras reservadas fossem usadas como identificadores
- Outras linguagens possuem estrutura léxica engessada (formato é fixado), tal que as posições da coluna são significativas
  - Por exemplo: um comando por linha
- As linguagens modernas na sua grande maioria são livres de formato, ou seja, suas posições de coluna são ignoradas

Conhecem alguma linguagem moderna não livre de formato?

GRAMÁTICAS E PROLOG

Linguagens de Programação

Implementação de Gramática em Prolog

Cuidado: Prolog não suporta recursão à esquerda

- A gramática...

```
R1: <exp>      ::= <mulexp> + <exp>
                 | <mulexp>
    <mulexp>   ::= <rootexp> * <mulexp>
                 | <rootexp>
    <rootexp>  ::= ( <exp> )
                 | <num>
```

Modificada por recursão à direita

Trocou <var> por <num>

... pode ser implementada em Prolog como

```
exp --> mulexp, ['+'], exp.
exp --> mulexp.
mulexp --> rootexp, ['*'], mulexp.
mulexp --> rootexp.
rootexp --> ['('], exp, [')'].
rootexp --> num.
num([X|L], L) :- number(X).
```

Como você poderia:

1) Adicionar o operador `&` (E aritmético) com **menor precedência** que a soma com **associatividade à direita?**
2) Adicionar o operador `**` (potência) com **maior precedência** que a multiplicação com **associatividade à direita**?

Outra Gramática em Prolog

- A nova gramática pode ser implementada em Prolog

```
exp --> sumexp, ['&'], exp.
exp --> sumexp.
sumexp --> mulexp, ['+'], sumexp.
sumexp --> mulexp.
mulexp --> powerexp, ['*'], mulexp.
mulexp --> powerexp.
powerexp --> rootexp, ['**'], powerexp.
powerexp --> rootexp.
rootexp --> ['('], exp, [')'].
rootexp --> num.
num([X|L], L) :- number(X).
```

R_2:

```
<exp> ::= <sumexp> & <exp>
                  | <sumexp>
<sumexp> ::= <mulexp> + <sumexp>
                  | <mulexp>
<mulexp> ::= <powerexp> * <mulexp>
                  | <powerexp>
<powerexp> ::= <rootexp> ** <powerexp>
                  | <rootexp>
<rootexp> ::= ( <exp> )
                  | <num>
```

Gramática do Português em Prolog

- A gramática do Português pode ser escrita em Prolog

```
sentenca --> expr_nominal, predicado.
expr_nominal --> artigo, nome.
expr_nominal --> artigo, nome, expr_preposicional.
predicado --> verbo.
predicado --> verbo, expr_nominal.
predicado --> verbo, expr_nominal, expr_preposicional.
expr_preposicional --> preposicao, expr_nominal.
nome --> [menino] ; [menina] ; [pato] ;
         [telescopio] ; [musica] ; [pena].
preposicao --> [ate] ; [com].
verbo --> [e] ; [viu] ; [esta] ; [toca] ;
          [conta] ; [surpreende].
artigo --> [o] ; [a] ; [um] ; [uma].
```

Como contar quantas palavras tem em uma sentença?

Embutindo Computações nas Cláusulas

- É possível embutir computações nas cláusulas da gramática. Por exemplo, para contar as palavras numa sentença

```
sentenca(W) --> expr_nominal(W1), predicado(W2), {W is W1 + W2}.
expr_nominal(W) --> artigo, nome, {W is 2}.
expr_nominal(W) --> artigo, nome, expr_preposicional(WP),
                    {W is WP + 2}.
predicado(W) --> verbo, {W is 1}.
predicado(W) --> verbo, expr_nominal(WN), {W is WN + 1}.
predicado(W) --> verbo, expr_nominal(WN), expr_preposicional(WP),
                 {W is 1 + WN + WP}.
expr_preposicional(W) --> preposicao, expr_nominal(WN),
                          {W is 1 + WN}.
nome --> [menino] ; [menina] ; [pato] ;
         [telescopio] ; [musica] ; [pena].
preposicao --> [ate] ; [com].
verbo --> [e] ; [viu] ; [esta] ; [toca] ;
          [conta] ; [surpreende].
artigo --> [o] ; [a] ; [um] ; [uma].
```

É possível computar o resultado de uma expressão na gramática R_1?

Calculando o Valor das Expressões

- Embutindo as computações das expressões na gramática R_1

```
exp(N) --> mulexp(N1), [+], exp(N2), {N is N1 + N2}.
exp(N) --> mulexp(N).
mulexp(N) --> rootexp(N1), [*], mulexp(N2),
              {N is N1 * N2}.
mulexp(N) --> rootexp(N).
rootexp(N) --> ['('], exp(N), [')'].
rootexp(N) --> num(N).
num(X, [X|L], L) :- number(X).
```

Gramáticas na Hierarquia de Chomsky

- Como implementar as seguintes gramáticas em Prolog?
  - $L_1 = \{a^* b^*\}$

```
P → AB
A → aA
    | λ
B → bB
    | λ
```

```
p --> a, b.
a --> ['a'], a.
a --> [].
b --> ['b'], b.
b --> [].
```

Gramáticas na Hierarquia de Chomsky

- Como implementar as seguintes gramáticas em Prolog?
  - $L_2 = \{ a^n b^n \mid n \geq 0 \}$

```
P → aPb
      | λ
```

```
p --> ['a'], p, ['b'].
p --> [].
```

Gramáticas na Hierarquia de Chomsky

- Como implementar as seguintes gramáticas em Prolog?
  - $L_3 = \{ a^n b^n c^n \mid n \geq 0 \}$

```
P  → aPbC | λ
Cb → Cb
Cc → cc
```

Cuidado: neste caso não é uma conversão direta!

```
abc --> as(N), bs(N), cs(N).

as(0) --> [].
as(M) --> [a], as(N), {M is N + 1}.

bs(0) --> [].
bs(M) --> [b], bs(N), {M is N + 1}.

cs(0) --> [].
cs(M) --> [c], cs(N), {M is N + 1}.
```

Gramáticas na Hierarquia de Chomsky

- Como implementar as seguintes gramáticas em Prolog?
  - $L_4 = \{ a^n b^m a^n b^m \mid n, m \geq 0 \}$

P	→	XY
X	→	aXA
Y	→	BYb
AB	→	BA
XB	→	bX
AY	→	Ya
XY	→	λ

```
abab --> an(N), bm(M), an(N), bm(M).

an(0) --> [].
an(NN) --> [a], an(N), {NN is N + 1}.

bm(0) --> [].
bm(MM) --> [b], bm(M), {MM is M + 1}.
```

Cuidado: também não é uma conversão direta!

Dica: é possível gerar gramáticas não livres de contexto!

ISSO É TUDO, PESSOAL!