Compiladores
(IF688)

Leopoldo Teixeira
lmt@cin.ufpe.br | @leopoldomt

Contato

- Slack: **https://if688.slack.com/signup**
  Faça o login com seu email do CIn.

- E-mail: **lmt@cin.ufpe.br**
  Sala C012

Relembrando…

- o que é um compilador?
- como se divide um compilador?
- quais as fases de um compilador?

Linguagem

- Aurélio: *o uso da palavra articulada ou escrita como meio de expressão e comunicação entre pessoas*.
- Insuficientemente preciso para desenvolvimento matemático de uma teoria de linguagens

Que elementos definem uma linguagem?

Símbolos

- Também chamados de caracteres ou átomos
  - Entidade abstrata básica, não definida formalmente
  - Considerado como unidade atômica, não importando sua particular representação visual (*representações gráficas indivisíveis*)
    - Ex.: *a*, *abc*, *begin*, *if*, 5, 1024, 2.017e4

Alfabeto

- Definição: Conjunto finito de símbolos
  - conjunto infinito não é alfabeto
- Exemplo: conjunto **Σ** dos dígitos hexadecimais:
  - **Σ** = {0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f}
  - Cadeias que podem ser construídas a partir dos símbolos desse alfabeto correspondem aos numerais hexadecimais:
    - *123*, *a0b56*, *fe5dc*, *b*, *abc*, *55efff* ...

O que seria o alfabeto de uma linguagem de programação?

Programação

- O alfabeto de uma linguagem de programação é o conjunto de todos os símbolos usados na construção de programas, incluindo:
  - letras
  - dígitos
  - caracteres especiais como “>”, “/”, etc...
  - espaço ou “branco”

Palavra

- Sequência finita de símbolos do alfabeto
- O comprimento de uma palavra α é dado por | α |
    - número natural que designa a quantidade de *símbolos*

Cadeia vazia

- O conceito de cadeia vazia é especialmente importante na teoria das linguagens formais.

- Denota-se por ε a cadeia formada por uma quantidade nula de símbolos, isto é, a cadeia que não contém nenhum símbolo.

- Formalmente: | **ε** | = 0

Concatenação

- Duas cadeias, sejam elas elementares ou não, podem ser anexadas, formando uma só cadeia, através da concatenação.

- Operação binária sobre palavras: forma nova palavra como justaposição da primeira com a segunda

- **αβ** denota a concatenação de duas cadeias **α** e **β**

Propriedades

- No caso da cadeia vazia **ε** (elemento neutro em relação ao operador de concatenação) são válidas as seguintes relações:

- **αε** = **εα** = **α**

- |**αε**| = |**εα**| = |**α**|

Concatenação Sucessiva

- Ou Concatenação Sucessiva de uma Palavra (com ela mesma)
  - w^n
  - **onde n é o número de concatenações sucessivas**
- indutivamente, a partir da operação de concatenação
  - $w^0 = \varepsilon$
  - $w^n = ww^{n-1}$, para $n > 0$

Linguagem Formal

Uma linguagem formal é um **conjunto**, finito ou infinito, de **cadeias** de **comprimento finito**, formadas pela **concatenação** de elementos de um **alfabeto finito e não-vazio**.

Conjunto de todas as palavras

- Se **Σ** é um alfabeto
- Como definir o Conjunto de Todas as Palavras possíveis deste alfabeto?

Fechamento Reflexivo e Transitivo

O fechamento reflexivo e transitivo de um alfabeto **Σ** é definido como o conjunto (infinito) que contém todas as possíveis cadeias que podem ser construídas sobre o alfabeto dado, incluindo a cadeia vazia.

Formalmente, o fechamento reflexivo e transitivo de um conjunto **Σ** é definido como:

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \ldots = \bigcup_{i=0}^{\infty} \Sigma^i$$

Conjunto de todas as palavras

Coleção de todas as cadeias, de qualquer comprimento, que possam ser formadas por concatenação a partir dos símbolos do alfabeto.

- Σ^* = conjunto de todas as palavras possíveis sobre Σ
- $\Sigma^+ = \Sigma^* - \{\varepsilon\}$

Linguagem Formal

- Formulada de maneira mais rigorosa com o auxílio da operação de fechamento reflexivo e transitivo:
- sendo uma linguagem qualquer coleção de cadeias sobre um determinado alfabeto **Σ**, e como **Σ*** contém todas as possíveis cadeias sobre **Σ**, então toda e qualquer linguagem **L** sobre um alfabeto **Σ** sempre poderá ser definida como sendo um subconjunto de **Σ***, ou seja, **L** ⊆ **Σ***.

Definição de uma linguagem

- Especificação da sintaxe:
  - gramática livre de contexto
    BNF (Backus-Naur Form)
- Especificação da Semântica:
  - normalmente informal (textual)
  - formal: uso de semântica operacional, denotacional, de ações, etc.

Gramáticas

Exemplo: if-else

```
if ( expression ) statement else statement
```

Exemplo: if-else

```
if ( expression ) statement else statement
```

stmt → **if (** *expr* **)** *stmt* **else** *stmt*

Gramática Livre de Contexto

- Um conjunto de ***tokens***, ***símbolos terminais***
- Um conjunto de símbolos ***não-terminais***
- Um conjunto de ***produções***
    - cada produção consiste de um não-terminal, uma seta, e uma sequencia de tokens e/ou não terminais
- Um não terminal designado como símbolo ***inicial***

Exemplo 1

exp → *exp* + *exp*

exp → *exp* - *exp*

exp → *digit*

digit → **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

Derivamos palavras de uma gramática G a partir do seu símbolo inicial e repetidamente substituindo não-terminais pelo corpo de uma produção

A linguagem gerada por G chama-se L(G). Inclui todas as strings que podemos obter através de derivações em G.

Derivamos palavras de uma gramática G a partir do seu símbolo inicial e repetidamente substituindo não-terminais pelo corpo de uma produção

A linguagem gerada por G chama-se L(G). Inclui todas as strings que podemos obter através de derivações em G.

Exemplo

Para a gramática G abaixo:

exp → *exp* + *exp* | *exp* - *exp* | *digit*
digit → **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

L(G) = {0, 1, ..., 0+1, 0+2, ..., 1-1, ...}

Outro Exemplo…

call → **id (** *optparams* **)**

optparams → *params* | ε

params → *params* **,** *param* | *param*

Como poderíamos evitar o uso do símbolo ε?

Removendo ε

call → **id ( )** | **id (** *params* **)**

params → *params*, *param* | *param*

Defina uma gramática para a linguagem de parênteses.
E.g., (), ()(), ((())), ()((())), etc.

Respostas

$A \rightarrow (\ A\ )\ |\ A\ A\ |\ ()$

$A \rightarrow (\ A\ )\ |\ A\ A\ |\ \varepsilon$

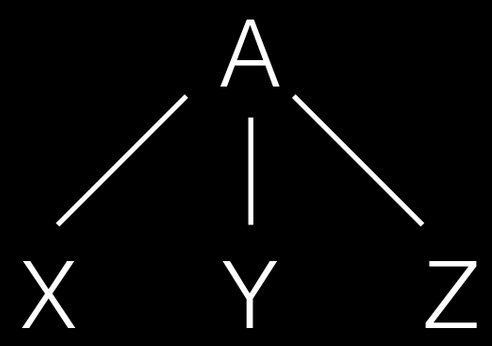
Parsing

- Problema de pegar uma string de terminais e verificar como derivá-la a partir do símbolo inicial da gramática;
    - caso não seja possível, reportar erros de sintaxe.
- Processo de procurar uma *parse-tree* para uma dada sequência de terminais.

Parse Trees

- Mostra graficamente como o símbolo inicial de uma gramática deriva uma string da linguagem.

- Para uma produção A → XYZ



Parse Tree

- A raiz é o símbolo inicial
- Cada folha é um terminal ou ε
- Cada nó interior é um não-terminal
- Se A é um não-terminal e $X_1, X_2,...,X_n$ são labels de filhos deste nó, tem que haver uma produção $A \rightarrow X_1 X_2 ... X_n$

Exemplo

exp → *exp* **+** *exp* | *exp* **-** *exp* | *digit*

digit → **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

Exemplo

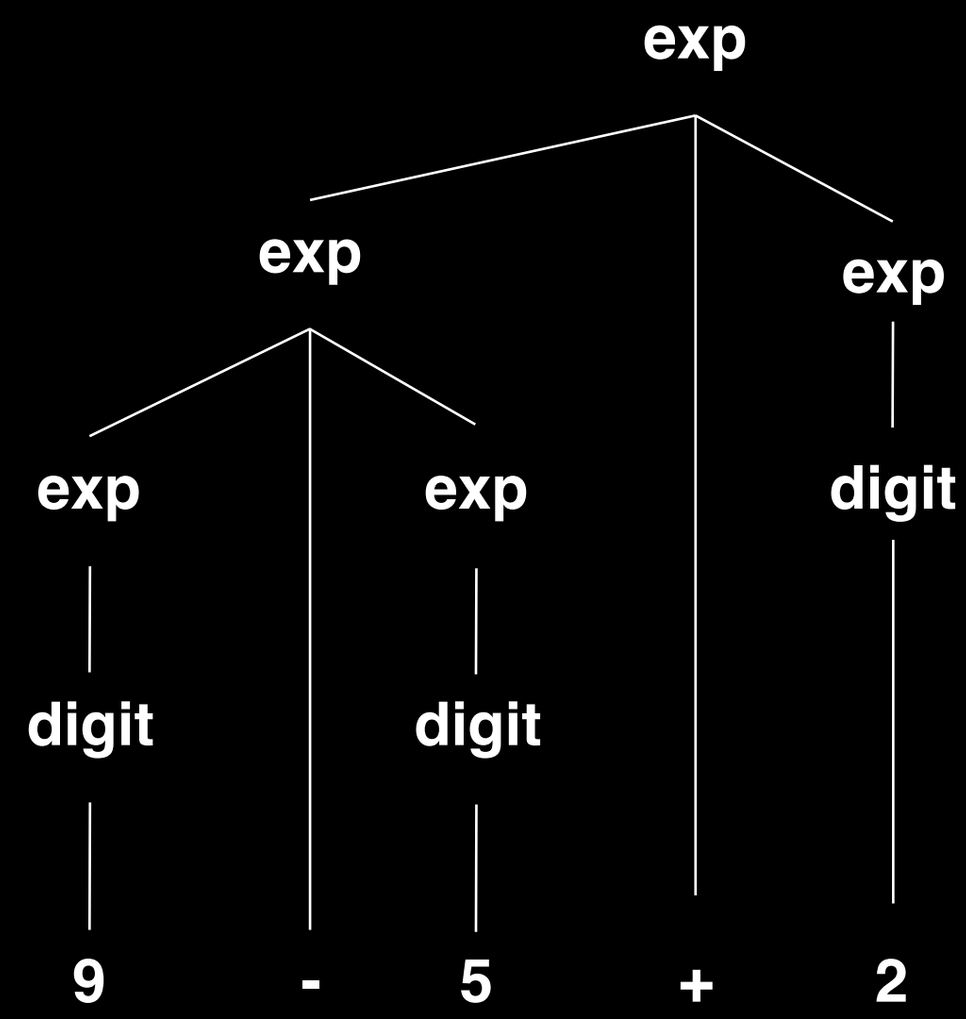
Árvore sintática para 9 – 5 + 2

exp → *exp* **+** *exp* | *exp* **-** *exp* | *digit*

digit → **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

Exemplo

Árvore sintática para 9 – 5 + 2



exp → *exp* **+** *exp* | *exp* **-** *exp* | *digit*

digit → **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

O que torna uma gramática ambígua?

Ambiguidade

- Uma gramática é dita **ambígua** quando gera mais de uma *parse-tree* para a mesma string.
- Problema: interpretação pode ser diferente de acordo com estrutura derivada

Ambiguidade

exp → *exp* + *exp*
| *exp* **-** *exp*
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

Ambiguidade

9 – 5 + 2

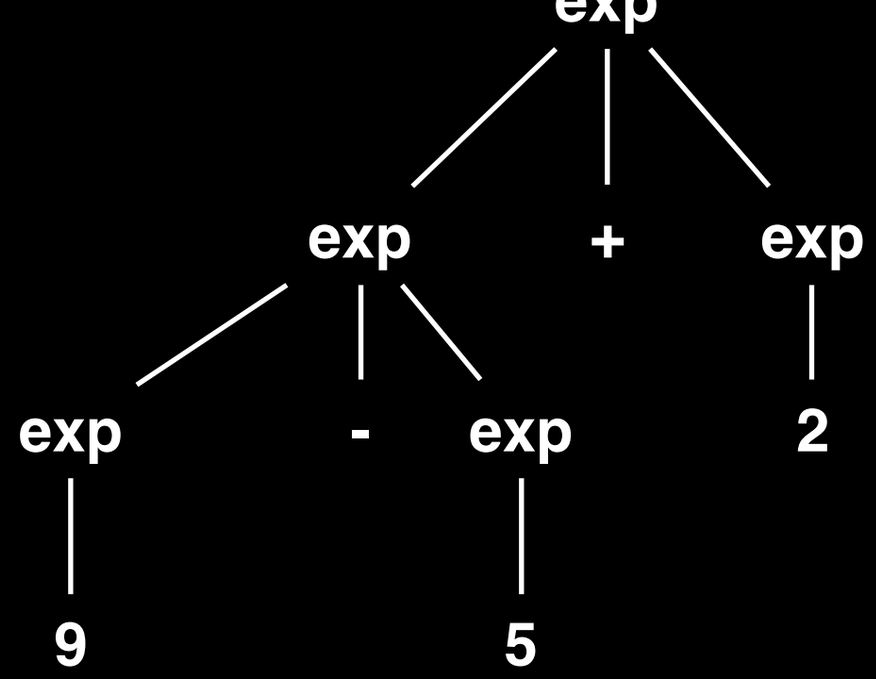
exp → *exp* + *exp*

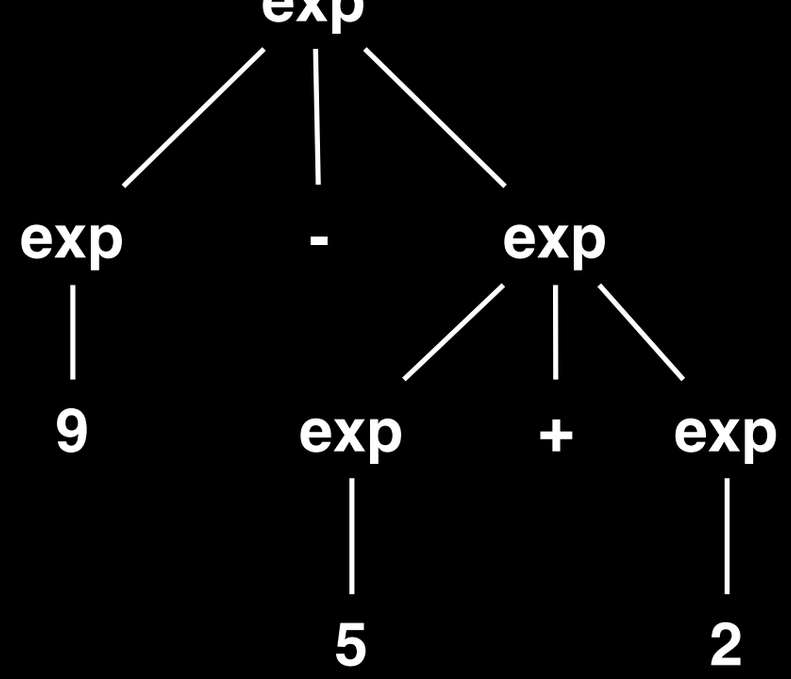
| *exp* - *exp*

| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

Exemplo: Duas *parse trees*

9 – 5 + 2





Como Eliminar Ambiguidade

- Reescrever gramática (mais comum)

- Usar gramáticas ambíguas com informações adicionais sobre como resolver ambigüidades

Exemplo 1

exp → *exp* + *digit*

exp → *exp* **-** *digit*

exp → *digit*

digit → **0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9**

Associatividade de Operadores

- Na maioria das linguagens de programação, os operadores **+**, **–**, ***** e **/** associam à esquerda
  - Exemplo: 9 – 5 + 2 equivale a (9-5)+2
- Atribuição em C e exponenciação associam à direita
  - Exemplo: a = b = c equivale a a = (b = c)

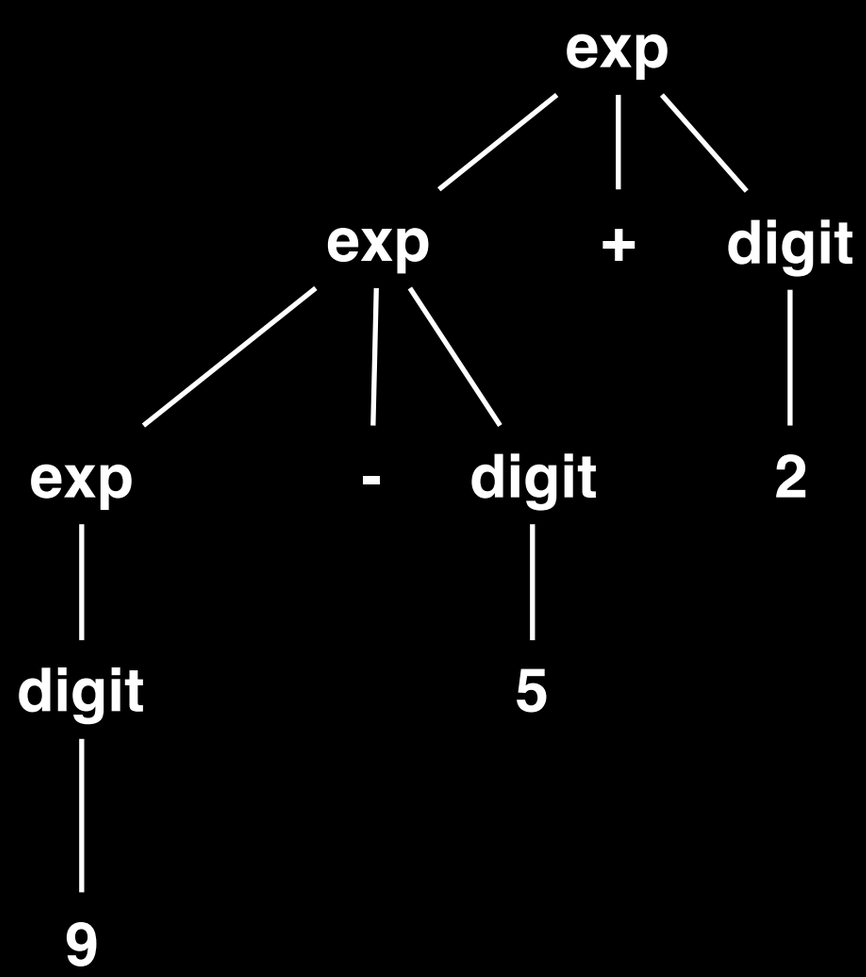
Associatividade à direita

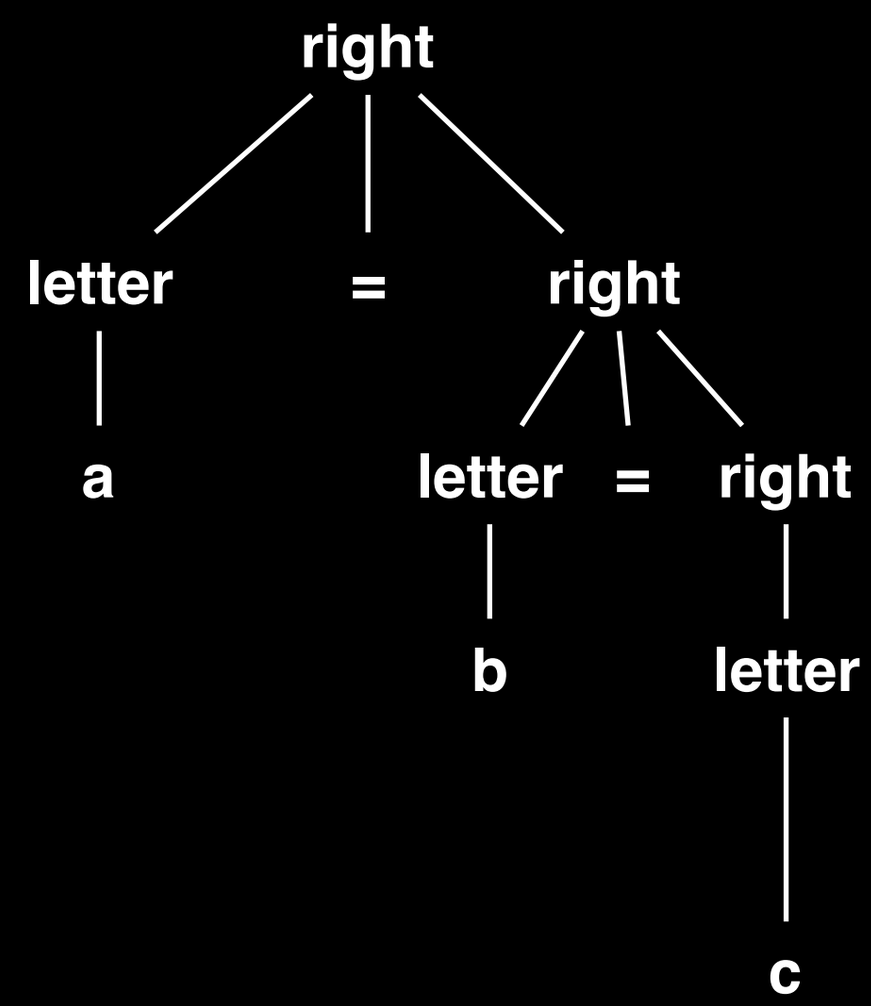
right → *letter* = *right* | *letter*

letter → **a** | **b** | … | **z**

Contraste *left* vs. *right*

- 9 – 5 + 2 vs. a = b = c





Precedência de operadores

- Considere a expressão 9 + 5 * 2
- Há duas interpretações possíveis
  - (9 + 5) * 2
  - 9 + (5 * 2)
- No entanto…
  - Multiplicação tem precedência sobre adição

Insuficiente…

exp → *exp* + *digit* | *exp* - *digit* |
exp / digit | *exp * digit* |
digit

digit → **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

Como resolver precedência de operadores?

Precedência de operadores

- Reescrever a gramática
- Usar não-terminais para diferenciar diferentes níveis de precedência
  - + e -
  - * e /
- Um outro não-terminal resolve as unidades básicas (dígitos)

Unidade básica de expressões

factor → ***digit*** **|** **(** ***expr*** **)**

digit → **0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9**

Precedência de operadores

term → *term* * *factor* | *term* / *factor* | *factor*

factor → *digit* | (*expr*)

digit → **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

Precedência de operadores

expr → ***expr*** + ***term*** | ***expr*** – ***term*** | ***term***

term → *term* * *factor* | *term* / *factor* | *factor*

factor → *digit* | **(** *expr* **)**

digit → **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

Outro Exemplo

stmt → **id** = *expression* **;**
| **if (** *expression* **)** *stmt*
| **if (** *expression* **)** *stmt* **else** *stmt*
| **while (** *expression* **)** *stmt*
| **do** *stmt* **while (** *expression* **) ;**
| **{** *stmts* **}**

stmts → *stmts stmt*
| ε

Exercício

- Modifique a gramática abaixo para que expressões aritméticas associem a esquerda

string → *string* **+** *string*
| *string* **-** *string*
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

Exercício

- Considere a gramática

$$S \rightarrow S\,S+ \mid S\,S* \mid \mathbf{a}$$

- Mostre como a palavra aa+a* pode ser gerada por esta gramática
- Construa uma parse-tree para esta string
- Que linguagem esta gramática gera?

Exercício

- Qual a linguagem gerada pelas gramáticas abaixo?
- Quais destas gramáticas são ambíguas?

a) $S \rightarrow \mathbf{0}\ S\ \mathbf{1} \mid \mathbf{0}\ \mathbf{1}$

b) $S \rightarrow +\ S\ S \mid -\ S\ S \mid \mathbf{a}$

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

d) $S \rightarrow \mathbf{a}\ S\ \mathbf{b}\ S \mid \mathbf{b}\ S\ \mathbf{a}\ S \mid \varepsilon$

e) $S \rightarrow \mathbf{a} \mid S + S \mid S\ S \mid S\ * \mid (\ S\ )$

Exercício

- Construa gramáticas livres de contexto não ambíguas para as seguintes linguagens:
  - expressões aritméticas em notação pós-fixa
  - listas de identificadores separados por vírgula, associativas à esquerda
  - listas de identificadores separados por vírgula, associativas à direita
  - expressões aritméticas de inteiros e identificadores, com as quatro operações básicas +, -, *, /

Exercício

- Construa gramática livre de contexto para numerais romanos, até o valor de 4 mil
  - **https://en.wikipedia.org/wiki/Roman_numerals**

Tradução dirigida por sintaxe

Compilar expressões aritméticas infixas para pós-fixas

Notação pós-fixada

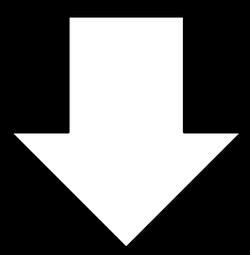
- Se E é uma variável ou constante, sua notação pós-fixada é ela mesma;
- Se E é uma expressão da forma E_1 **op** E_2, então sua notação pós fixada é E_1' E_2' **op**,
  - onde E_1' e E_2' são as notações pós-fixadas de E_1 e E_2;
- Se E é uma expressão com parênteses **(** E **)**, então sua notação pós-fixada é a mesma notação pós-fixada de E.

Exemplo

- (9-5)+2 → 95-2+
- 9-(5+2) → 952+-
- (9-(5+2))*3 → 952+-3*
- 9-(5+2)*3 → 952+3*-

Exemplo

$expr \rightarrow expr_1 + term$



traduza $expr_1$;
traduza term;
imprima **+** ;

Conceito: Atributo

- Um valor associado a um construtor do programa.
- Exemplos:
  - Tipo em uma expressão;
  - Número de instruções geradas;
  - Localização da primeira instrução gerada por um construtor;

Conceito: Esquema de tradução (dirigida pela sintaxe)

- Notação para associar trechos de programa a produções de uma gramática
- Os trechos de programa são executados quando a produção é usada durante a análise sintática
- O resultado da execução desses trechos de programa, na ordem criada pela análise sintática, produz a tradução desejada do programa fonte

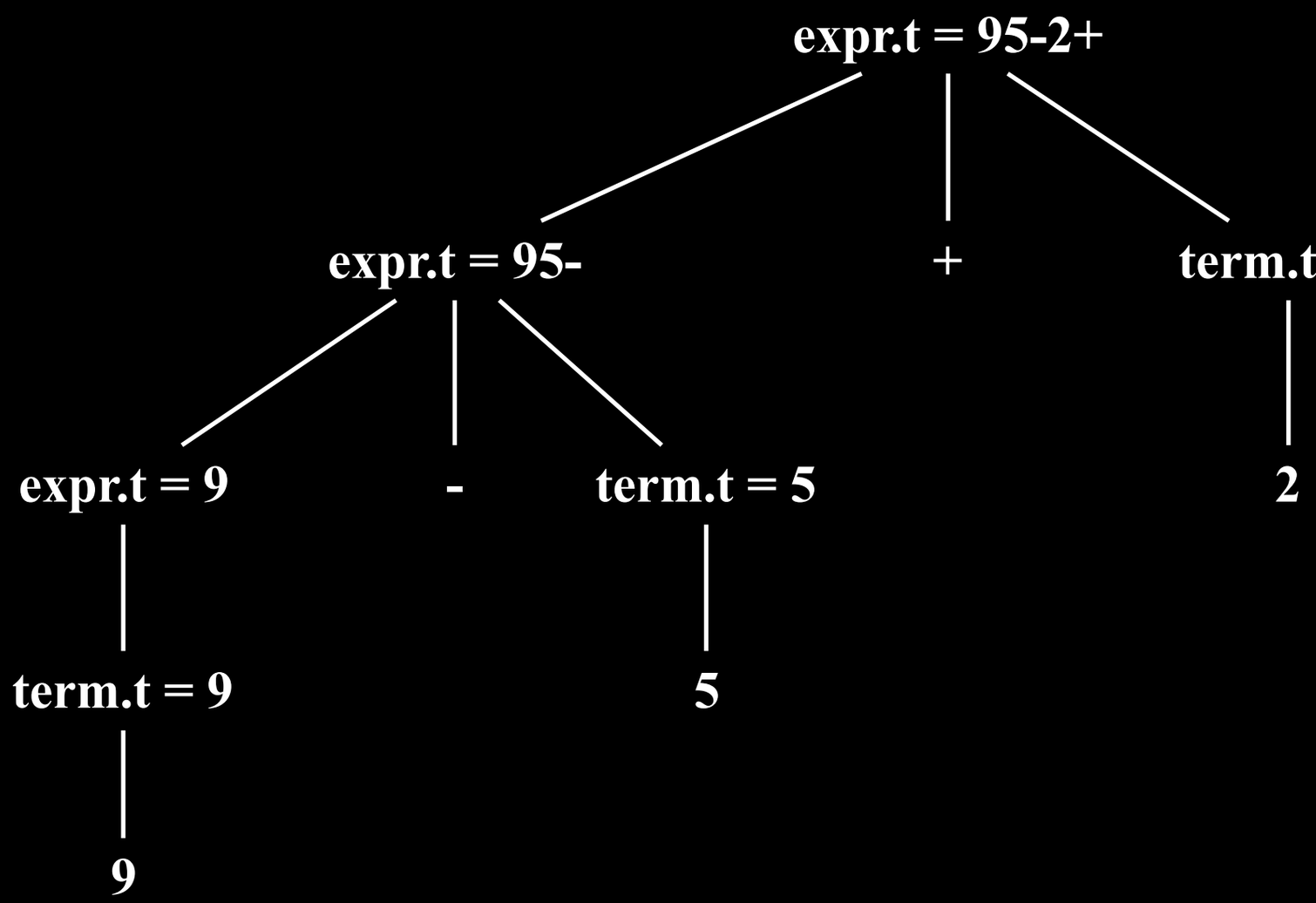
Atributos em Gramáticas

- Atributos permitem associar valores e tipos com expressões por meio da gramática
- Definição dirigida por sintaxe associa
  - a cada símbolo um conjunto de atributos;
  - a cada produção, regras semânticas para computar os valores dos atributos

Avaliando atribuitos

1. Dada uma string x, construa uma parse tree para ela;
2. Aplique as regras semânticas para avaliar os atributos em cada nó.

Exemplo



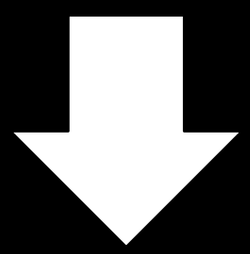
Tipos de Atributos

- Atributos **sintetizados**: seus valores são obtidos a partir dos *filhos* de um determinado nó;
  - Podem ser calculados através de uma travessia bottom-up;

- Atributos **herdados**: têm seus valores definidos a partir do *próprio nó*, de seus *pais* ou seus *irmãos*.

Exemplo

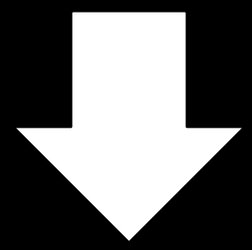
$expr \rightarrow expr_1$ **+** term



traduza $expr_1$;
traduza term;
trate **+** ;

Exemplo

$$\text{expr} \rightarrow \text{expr}_1 \; \mathbf{+} \; \text{term}$$



$$expr.t = expr_1.t \parallel term.t \parallel \text{‘+’}$$

Exemplo – Definição dirigida por sintaxe

Produção	***Regra semântica***
$expr \rightarrow expr_1 + term$	$expr.t = expr_1.t \parallel term.t \parallel$ '+'
$expr \rightarrow expr_1 - term$	$expr.t = expr_1.t \parallel term.t \parallel$ '-'
$expr \rightarrow term$	$expr.t = term.t$
$term \rightarrow \mathbf{0}$	$term.t =$ '0'
$term \rightarrow \mathbf{1}$	$term.t =$ '1'
…	
$term \rightarrow \mathbf{9}$	$term.t =$ '9'

Travessias

- Utilizadas para avaliação de atributos e especificar a execução de código em esquemas de tradução (*próximos slides*)
- Travessia em profundidade (*depth-first*)

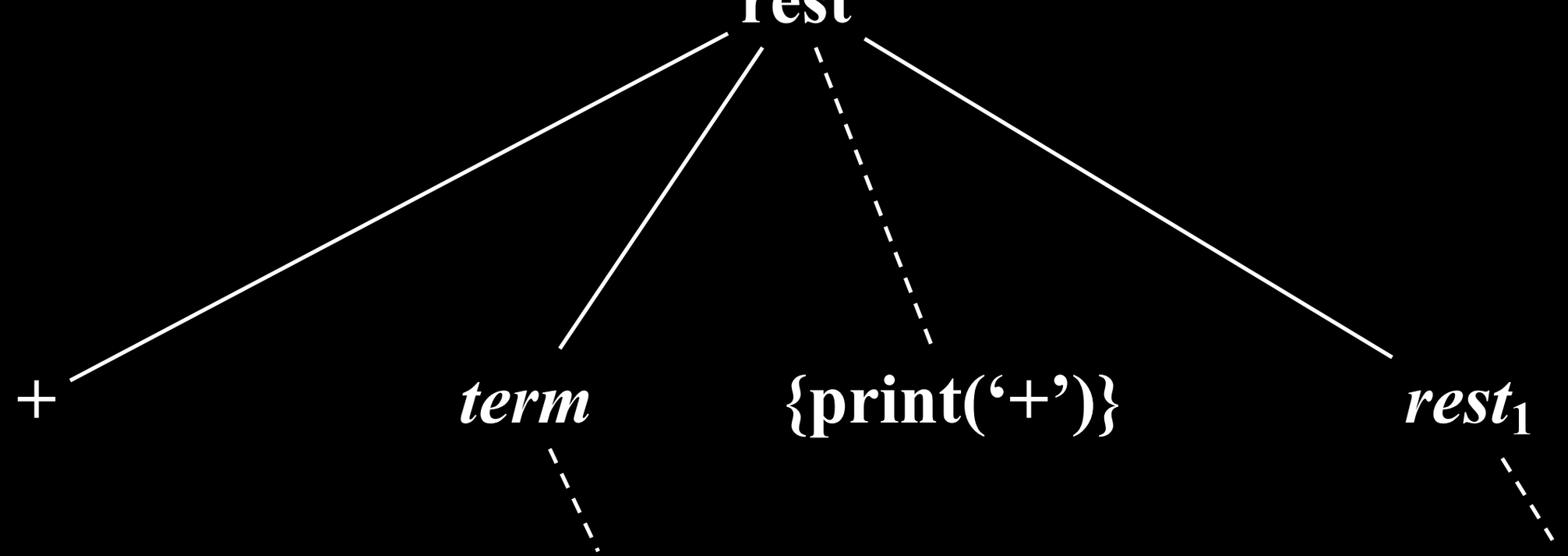
```
procedure visit (node N) {
   for (each child C of N, from left to right) {
      visit(C);
   }
 evaluate semantic rules at node N;
}
```

Esquemas de tradução

- Gramática livre de contexto com fragmentos de programas (**ações semânticas**) embutidos no lado direito das produções.
- Semelhante à definição dirigida por sintaxe, mas com a ordem de avaliação das regras semânticas explicitada.

Exemplo 1

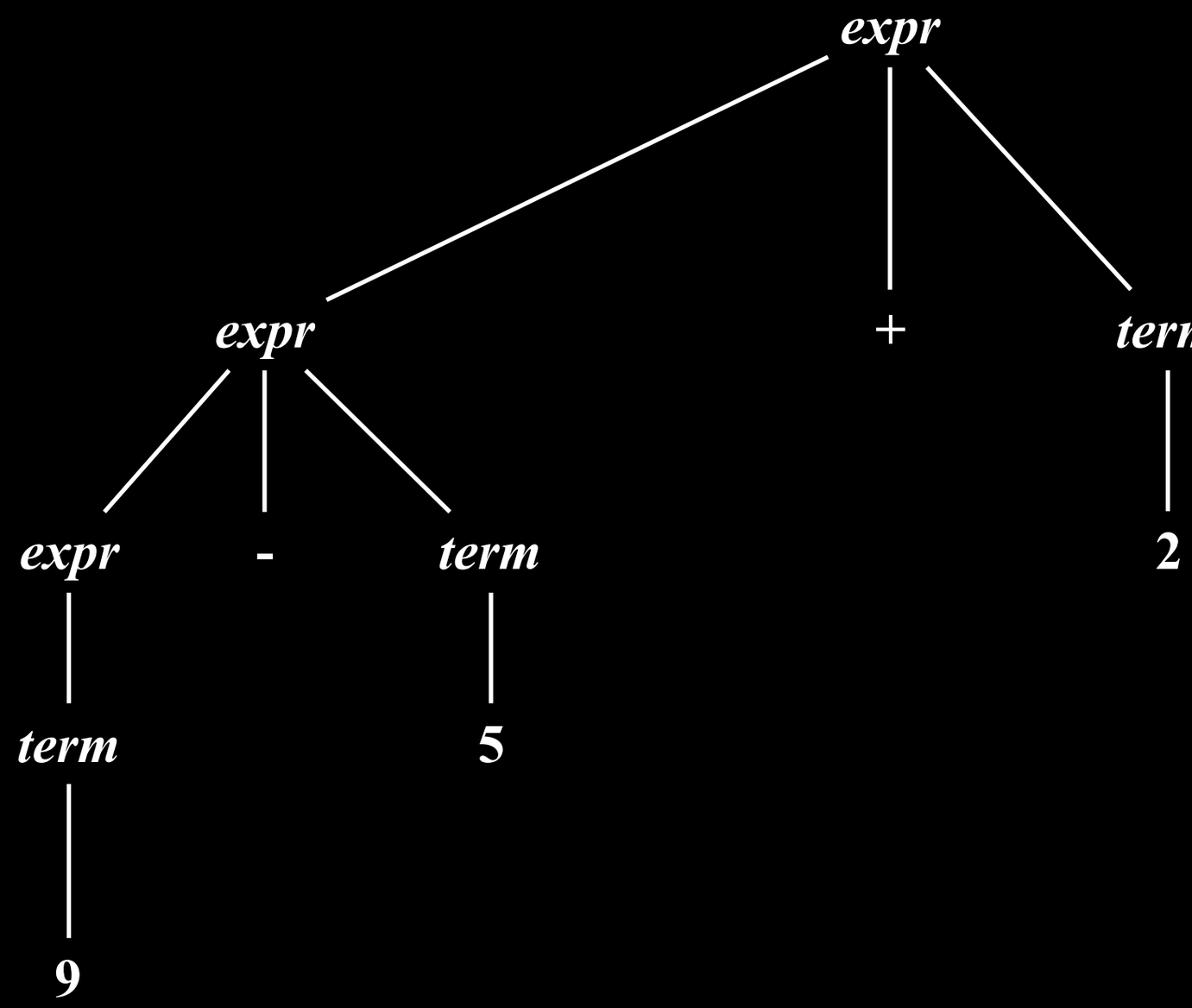
- *rest* → + *term* {print (‘+’) } $rest_1$



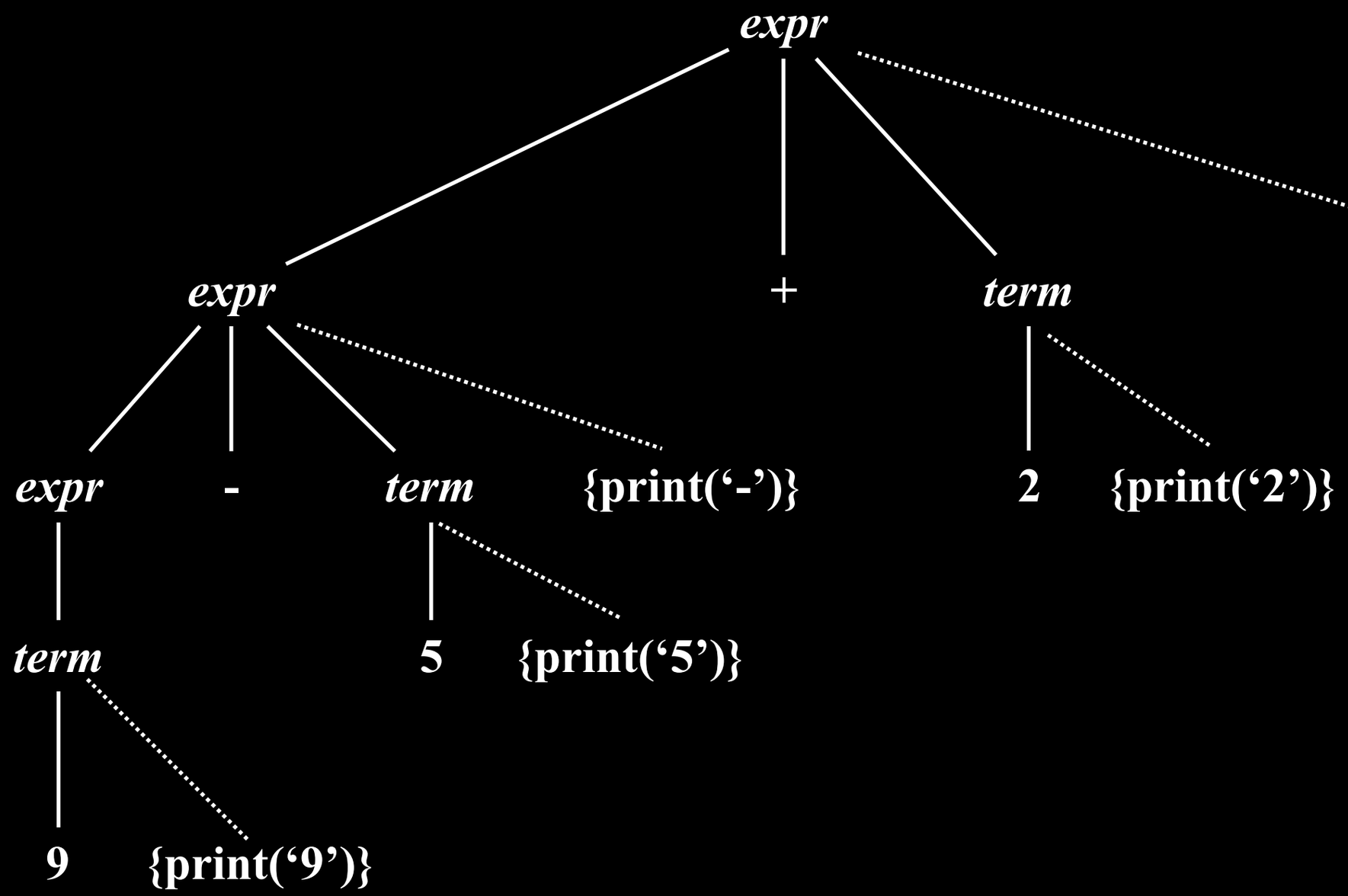
Esquema de tradução infixa → pós-fixa

expr ⟶ *expr* + *term* { print (‘+’) }
expr ⟶ *expr* - *term* { print (‘-’) }
expr ⟶ *term*
term ⟶ **0** { print (‘0’) }
term ⟶ **1** { print (‘1’) }
…
term ⟶ **9** { print (‘9’) }

Traduzindo 9-5+2



Ações traduzindo 9-5+2 em 95-2+



Parsing

- Processo de determinar como uma string de terminais pode ser gerada por uma gramática
- *Conceitualmente* é a construção da parse tree
- Mas a *parse tree* pode **não** ser efetivamente construída durante a compilação.

Parsing

- Para gramáticas livres de contexto sempre é possível construir um parser com complexidade O(n^3) para fazer o parsing de n tokens.

- Na prática, o parsing de linguagens de programação normalmente pode ser feito linearmente.

- Travessia linear da esquerda para a direita, olhando um token de cada vez.

Top-down ou bottom-up parsers

- Refere-se à ordem em que os nós da *parse tree* são criados.

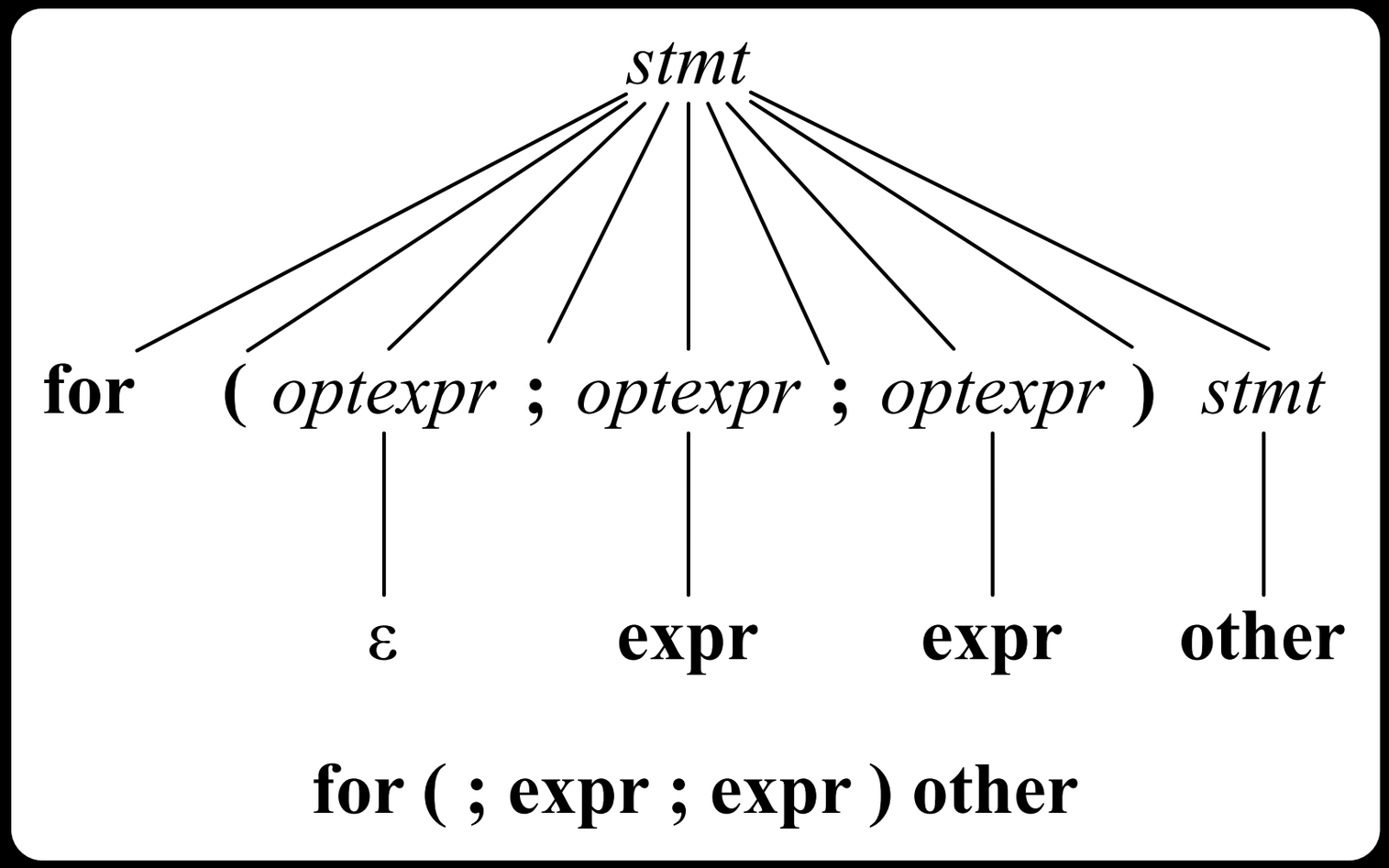
- Top-down: mais fáceis de escrever “à mão”

- Bottom-up: suportam uma classe maior de gramáticas e de esquemas de tradução; são frequentemente usados/gerados pelas ferramentas de geração automática de parsers.

Exemplo

stmt → **expr ;**
| **if (expr)** *stmt*
| **for (** *optexpr* **;** *optexpr* **;** *optexpr* **)** *stmt*
| **other**
optexpr → **expr**
| ε

Exemplo


stmt
for (optexpr ; optexpr ; optexpr) stmt
ε
expr
expr
other
for (; expr ; expr) other

Construindo um parser top-down

1. Para cada nó n, com um não-terminal A, selecione uma das produções de A e construa os filhos de n para os símbolos à direita da produção.

2. Encontre o próximo nó para o qual uma sub-árvore deve ser construída.

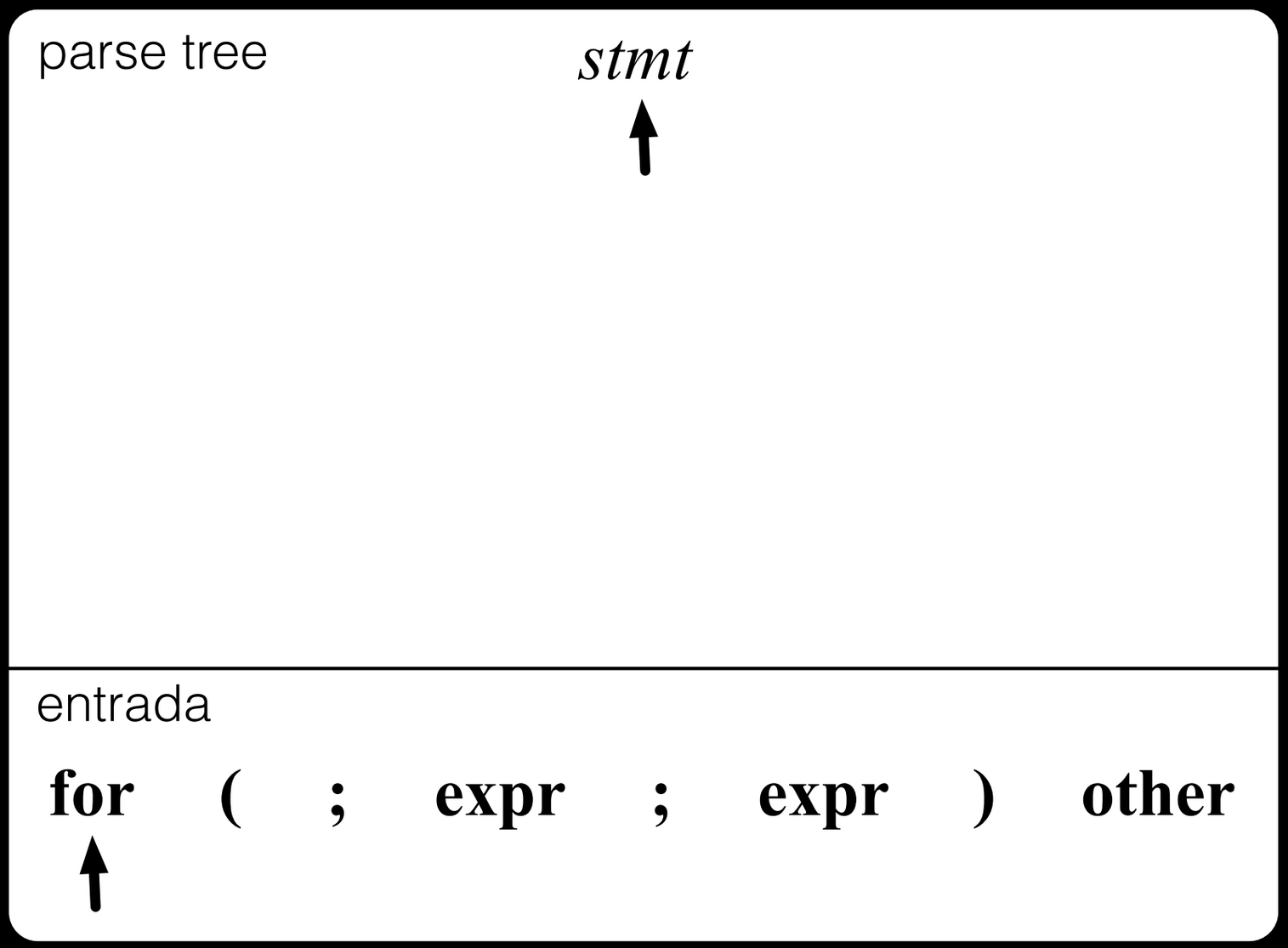
Construindo um parser top-down

- Para algumas gramáticas basta uma única travessia da esquerda para a direita da string de entrada.

- *Token* corrente é chamado de *lookahead symbol*.

- Exemplo:
  for (; expr ; expr) other

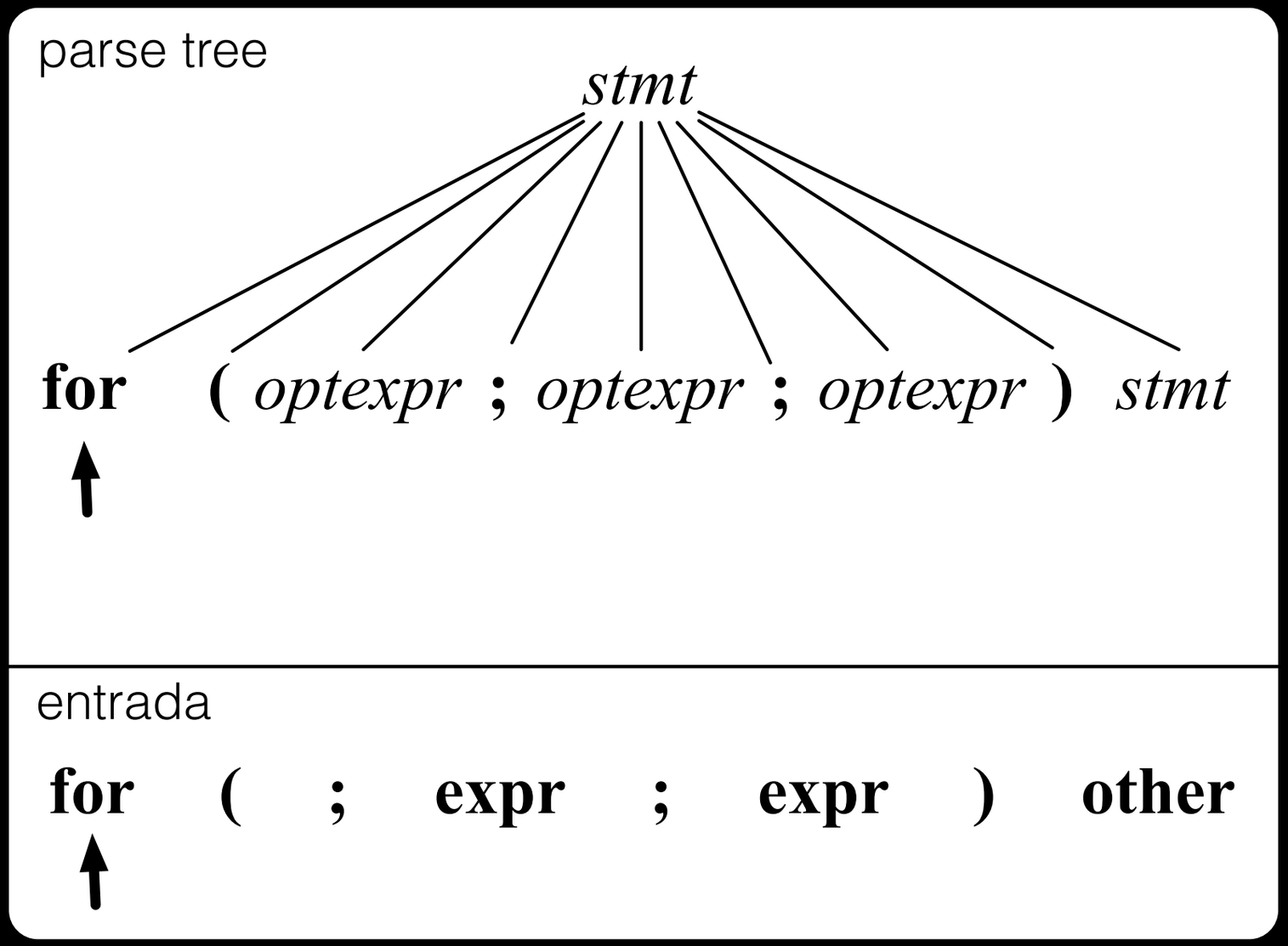
Exemplo



qual produção pode derivar uma string começando com lookahead?

Exemplo

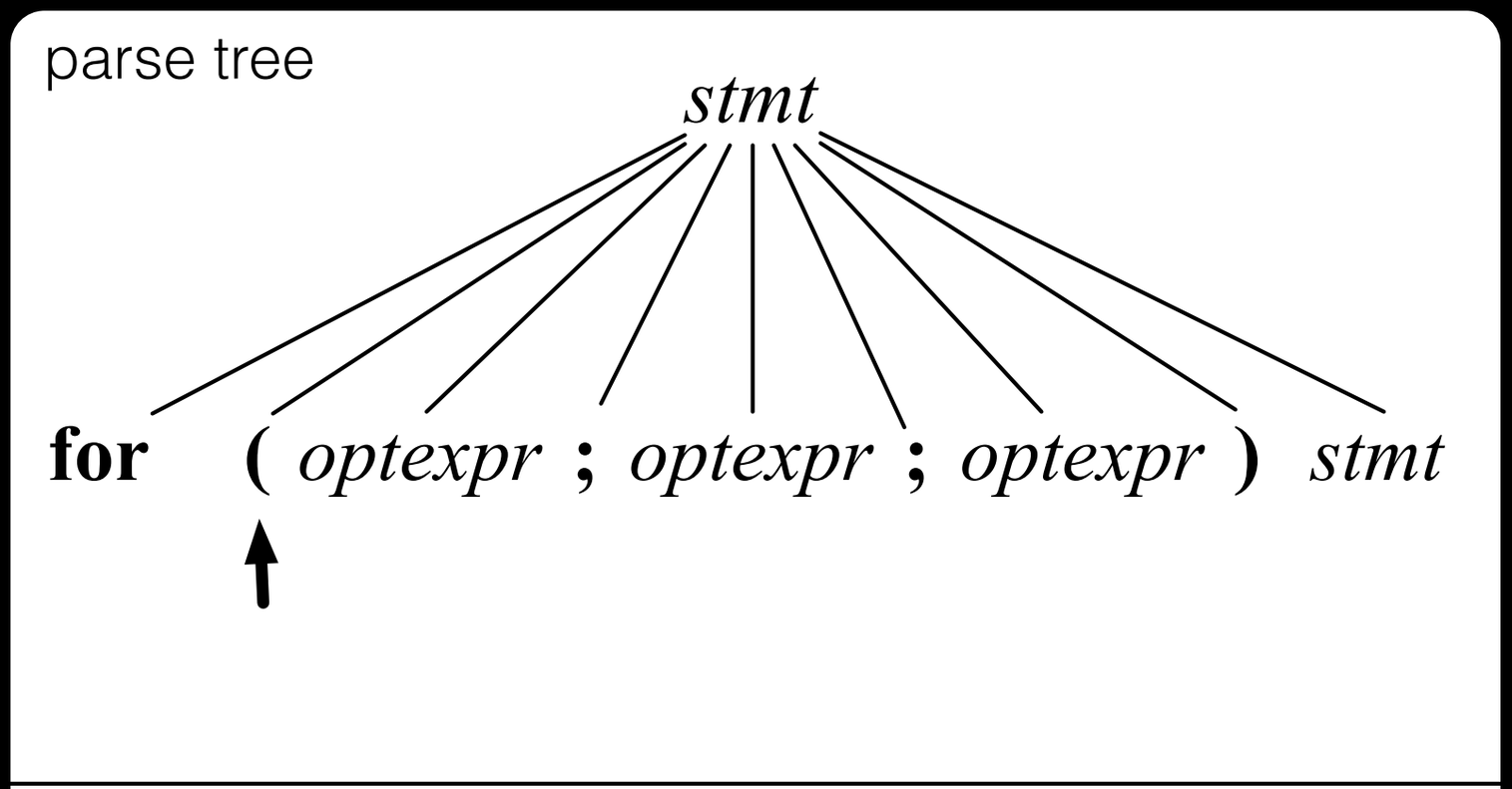
parse tree



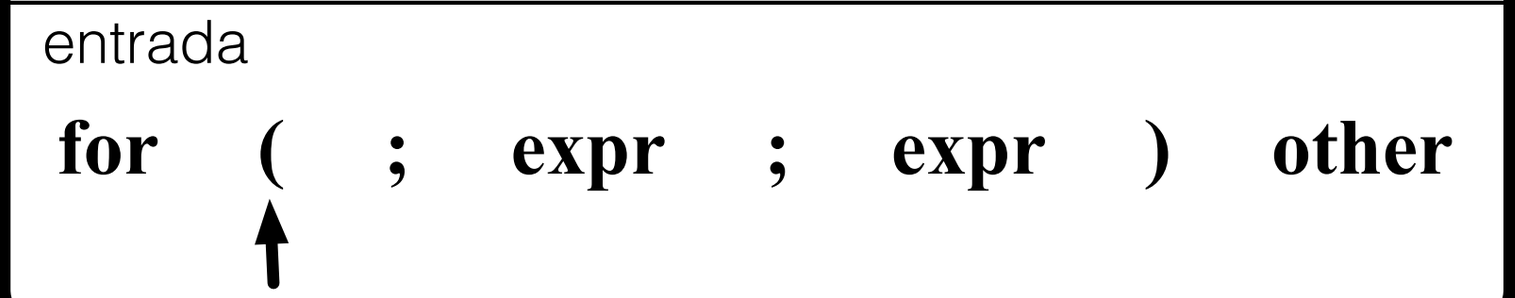
no caso, produção na árvore casa com *lookahead*

Exemplo

parse tree



entrada



e agora, pra onde *lookahead* vai apontar?

Backtracking

- A escolha de uma produção pode exigir tentativa-e-erro, voltando para tentar novas alternativas possíveis.

- *Predictive-parsing*: parsing em que não ocorre *backtracking*.

Recursive descent parsing

- Método de análise sintática top-down em que um conjunto de procedimentos recursivos é usado para processar a entrada.
- Cada procedimento está associado a um símbolo não-terminal da gramática.
- *Predictive parsing* é um caso especial de *recursive descent parsing* em que o símbolo *lookahead* determina sem ambiguidades o procedimento a ser chamado para cada não-terminal.

predictive parsing

stmt → **for (** *optexpr* **;** *optexpr* **;** *optexpr* **)** *stmt*

match (**for**); *match* (‘**(**‘);
optexpr (); *match* (‘**;**‘);
optexpr (); *match* (‘**;**‘);
optexpr (); *match* (‘**)**‘); *stmt* ();

predictive parsing

```
void stmt( ) {
   switch (lookahead) {
      case expr:    match(expr); match(';'); break;
      case if:      match(if); match('(');
                    match(expr);match(')'); stmt(); break;
      case for:     match(for); match('(');
                    optexpr(); match(';');
                    optexpr(); match(';'); optexpr();
                    match(')'); stmt(); break;
      case other:   match(other); break;
      default:      report("syntax error");
   }
}
```

predictive parsing

```
void match (terminal t) {
  if (lookahead == t)  lookahead = nextTerminal;
  else                 report (“syntax error”);
}

void optexpr( ) {
  if (lookahead == expr) match (expr) ;
}
```

Predictive parsing - Problema

- Recursão à esquerda leva a loop em predictive parsers:

$expr \rightarrow expr + term \mid term$

Predictive parsing - Problema

- Recursão à esquerda leva a loop em predictive parsers:

$expr \rightarrow expr + term \mid term$

$A \rightarrow A\alpha \mid \beta$

Predictive parsing - Solução

- Reescrever produções tornando-as recursivas à direita:

  $A \rightarrow A\alpha \mid \beta$

  reescrever para

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

- Exemplo: expressão “$\beta\alpha\alpha\alpha\alpha$”

Reescrita manual pode ser muito trabalhosa! Alguns tipos de parsers permitem definir regras de precedência e resolução de ambiguidade em produções.

Reescrevendo a gramática de expressões

expr → *expr* + *term* { print (‘+’) }
expr → *expr* - *term* { print (‘-’) }
expr → *term*
term → **0** { print (‘0’) }
term → **1** { print (‘1’) }
…
term → **9** { print (‘9’) }

Reescrevendo a gramática de expressões

$$A \rightarrow Aa \mid Ab \mid c$$

deve ser reescrita como

$$A \rightarrow cR$$
$$R \rightarrow aR \mid bR \mid \varepsilon$$

Reescrevendo a gramática de expressões

$A \rightarrow Aa \mid Ab \mid c$

deve ser reescrita como

$A \rightarrow cR$

$R \rightarrow aR \mid bR \mid \varepsilon$

expr → *expr* + *term*	{ print (‘+’) }
expr → *expr* - *term*	{ print (‘-’) }
expr → *term*	
term → **0**	{ print (‘0’) }
…	
term → **9**	{ print (‘9’) }

Reescrevendo a gramática de expressões

$A \rightarrow Aa \mid Ab \mid c$

deve ser reescrita como

$A \rightarrow cR$

$R \rightarrow aR \mid bR \mid \varepsilon$

expr → *expr* + *term*	{ print (‘+’) }
expr → *expr* - *term*	{ print (‘-’) }
expr → *term*	
term → **0**	{ print (‘0’) }
…	
term → **9**	{ print (‘9’) }

A = expr
a = + term {print(‘+’)}
b = - term {print(‘-’)}
c = term

Reescrevendo...

expr→ term rest
rest → + term {print (‘+’) } rest
rest → - term {print (‘-’) } rest
rest → ε
term→ 0 {print (‘0’) }
…
term→ 9 {print (‘9’) }

Implementando...

```
void expr () {
   term(); rest();
}
```

expr → term rest

Implementando…

```
void term () {
  if (lookahead is a digit) {
        t = lookahead;
        match(lookahead);
        print(t);
  }
  else report("syntax error");
}
```

```
term  → 0 {print ('0') }
…
term  → 9 {print ('9') }
```

Implementando…

```
void rest() {
  if (lookahead == '+') {
        match('+'); term(); print('+'); rest();
  }
  else if (lookahead =='-') {
        match('-'); term(); print('-'); rest();
  }
  else { }
}
```

rest	→	+ term {print ('+') } rest
rest	→	- term {print ('-') } rest
rest	→	ε

Código

Exercício

- Estenda a implementação do parser visto em aula para lidar com multiplicação e divisão, considerando a precedência

- Pense: como conseguir tratar qualquer tipo de número (não apenas single-digits) e espaços em branco?

  - se tiver alguma ideia, implemente!