Introdução ao parsing

- As linguagens de programação têm regras para aquilo que constitui um programa bem formado.

- Essas regras são normalmente especificadas numa notação especial chamada BNF (Backus-Naur Form) ou EBNF (extended BNF)

John Backus → FORTRAN
(1ª linguagem de programação de alto nível)

Peter Naur → ALGOL 60
(percursor de Pascal, C)

- As regras constituem a gramática da linguagem.

- O objectivo de um parser é verificar que a sequência de tokens obtida pela análise lexical ~~pode ser~~ é válida de acordo com a gramática.

- As regras de uma linguagem de programação são geralmente definidas de forma recursiva.

- Exemplo:

  - se S_1 e S_2 são instruções, e
    E for uma expressão, então

    if (E) S_1 else S_2

    é uma instrução.

- O exemplo anterior não pode ser especificado com a notação das expressões regulares.

- Mas pode ser especificado do seguinte modo:

  stmt → if (expr) stmt else stmt

onde stmt significa a classe de instruções e expr significa a classe de expressões.

Formalmente, uma gramática independente de contexto (ou simplesmente, uma gramática) tem 4 coisas:

1) Um conjunto de símbolos terminais
2) Um conjunto de símbolos não terminais.
3) Um símbolo inicial
4) Um conjunto de regras

- Os símbolos terminais são os tokens.
- Os símbolos não terminais são variáveis sintáticas que denotam conjuntos de strings, e impõem uma estrutura hierárquica na linguagem. (No exemplo anterior, stmt e expr são símbolos não terminais)
- O símbolo inicial, cujo conjunto de strings denota a linguagem definida pela gramática.
- As regras especificam o modo como os símbolos terminais e não terminais podem ser combinados.

Exemplo: uma gramática para expressões aritméticas simples.

expr → expr op expr
expr → (expr)
expr → − expr
expr → id
op → +
op → *
op → −
op → /
op → ↑

Símbolos Terminais: id + − * / ↑ ( )

" não terminais: expr op

Símbolo inicial: expr

Regras:

Derivações

- Podemos pegar no lado esquerdo de uma regra e substituí-la pelo que está no lado direito dessa regra. A esta substituição dá-se o nome de derivação.

- Ex: $expr \Rightarrow -expr \Rightarrow -(expr) \Rightarrow -(id)$

  $-(id)$ foi derivado a partir de expr. Significa isso que $-(id)$ é uma instância particular de expr.

- Notação:

  $\Rightarrow$ ⇝ deriva num passo

  $\overset{*}{\Rightarrow}$ ⇝ deriva em zero ou mais passos.

  $\overset{+}{\Rightarrow}$ ⇝ deriva em um ou mais passos.

$expr \overset{*}{\Rightarrow} -(id)$

- Dada uma gramática G e um símbolo inicial S

  - $L(G)$ ⇝ linguagem gerada por G.

- Uma sequência de símbolos terminais w pertence a $L(G)$ se e só se $S \overset{+}{\Rightarrow} w$

- A string w é uma frase de G.

- Para verificar se um programa é sintaticamente válido, necessitamos de mostrar que esse programa (que é uma sequência de tokens) pode ser derivado a partir do símbolo inicial da gramática da linguagem de programação.

  - Um programa em C é sintáticamente correcto se poder ser gerado a partir da gramática do C.

- Como fazer essa verificação?
  → é essa a tarefa do parsing.

Considere a seguinte gramática:

$$
\begin{aligned}
E \rightarrow\ & E + E \\
\mid\ & E * E \\
\mid\ & (E) \\
\mid\ & -E \\
\mid\ & id
\end{aligned}
$$

→ está na notação EBNF.

"|" ~ "ou"

- A string $-(id + id)$ é uma frase da gramática. Porquê? Porque pode ser derivada a partir de E, o símbolo inicial da gramática.

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow (-id+E)$$
$$\Rightarrow -(id + id)$$

- Podíamos ter feito outra derivação. Ex:

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(E + id)$$
$$\Rightarrow -(id + id)$$

↑ diferença está aqui

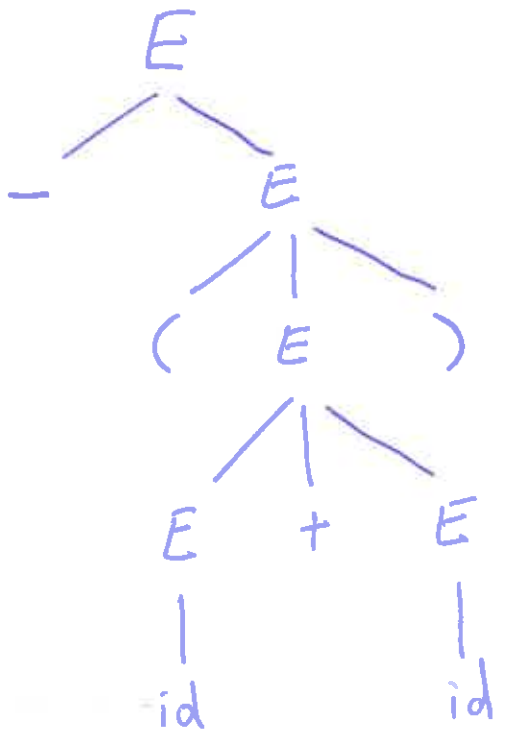
- Quando substituimos sempre o símbolo não terminal mais à esquerda, dizemos que fazemos uma derivação mais à esquerda.

- Situação análogo para o caso de usarmos sempre o símbolo não terminal mais à direita → derivação mais à direita.

- Árvore sintática (Parse tree) → é uma representação gráfica de uma derivação, embora não especifique a ordem pela qual a derivação foi feita.

  - raíz → símbolo inicial
  - nós interiores → não terminais
  - folhas → terminais / tokens

- Árvore de parsing para $-(id + id)$ de acordo com a gramática dada.



- Se uma frase tiver mais do que uma árvore de parsing, a gramática diz-se <u>ambígua</u>.

- Normalmente elimina-se a ambiguidade transformando a gramática, ou com a introdução de uma regra para resolver as situações ambíguas.

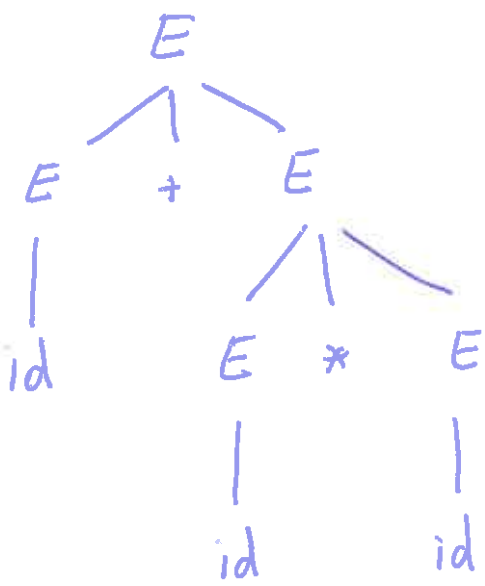
  — Uma linguagem de programação não pode ser ambígua, pois isso implicaria que um programa pudesse ser interpretado de maneiras distintas.

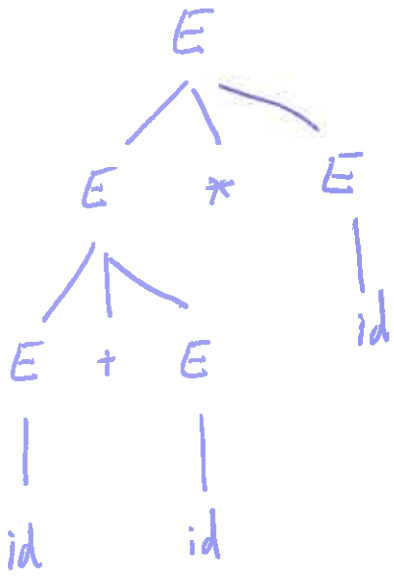
- A gramática que vimos há pouco é ambígua. Por exemplo, a frase id + id * id pode dar origem a duas árvores distintas.

$$
\begin{aligned}
E &\Rightarrow E + E \\
&\Rightarrow id + E \\
&\Rightarrow id + E * E \\
&\Rightarrow id + id * E \\
&\Rightarrow id + id * id
\end{aligned}
$$

$$
\begin{aligned}
E &\Rightarrow E * E \\
&\Rightarrow E + E * E \\
&\Rightarrow id + E * E \\
&\Rightarrow id + id * E \\
&\Rightarrow id + id * id
\end{aligned}
$$

Nota: são ambas derivações mais à esquerda.



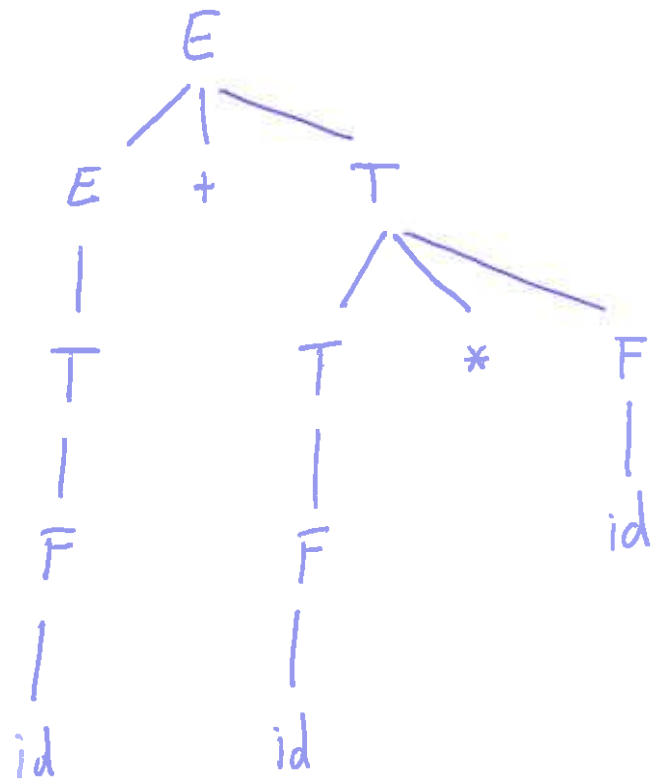


Podíamos eliminar a ambiguidade transformando a gramática para:

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid -E \mid id$$



Outro exemplo típico de ambiguidade: "dangling else"

```
stmt → if ( expr ) stmt
     | if ( expr ) stmt else stmt
     | ...
```

A gramática é ambígua. A seguinte frase tem 2 árvores de parsing.

```
if (E1) if (E2) S1 else S2
```

Duas interpretações possíveis.

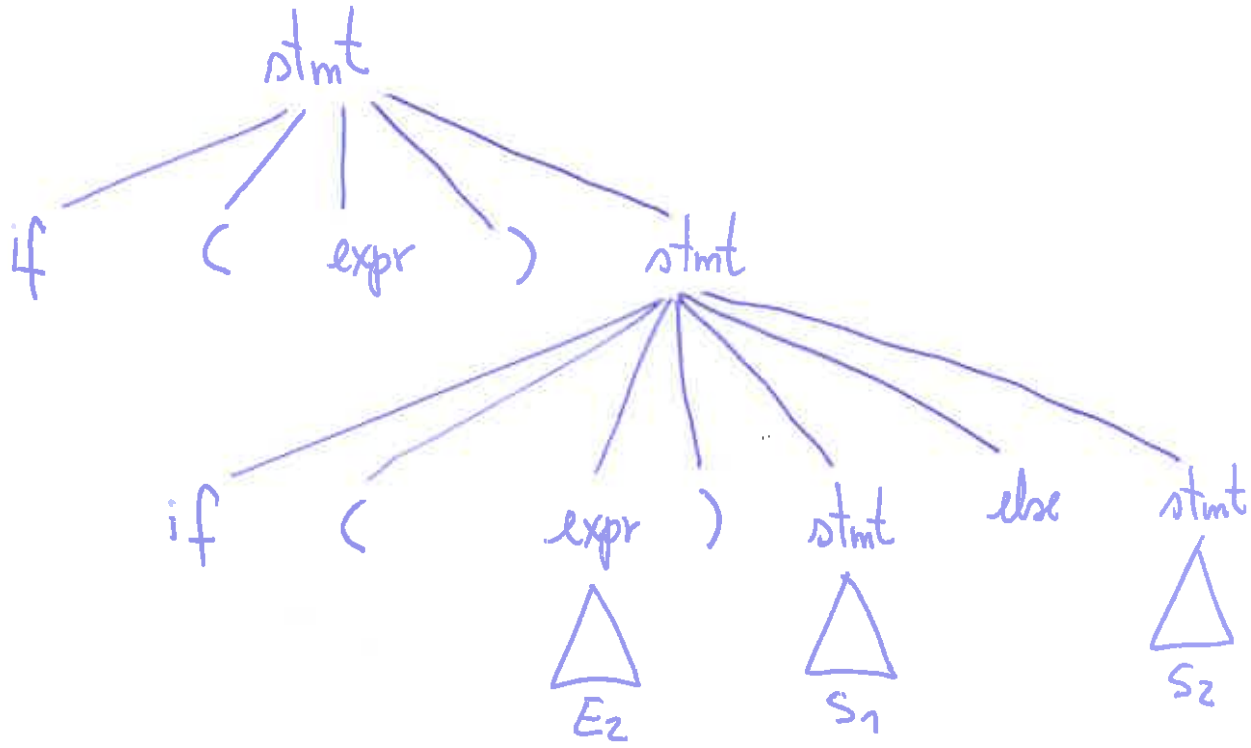
①

```
if (E1)
    if (E2)
    |   S1
    else
        S2
```

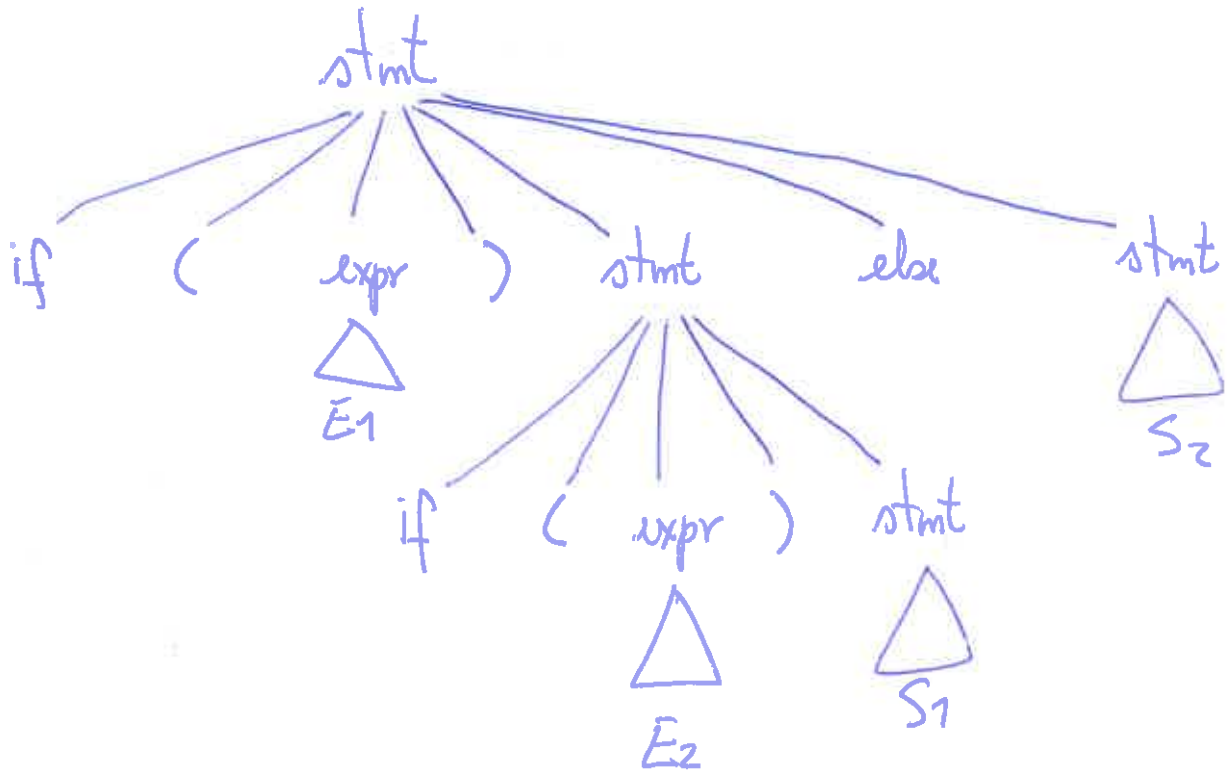
②

```
if (E1)
|   if (E2)
|       S1
else
    S2
```

①



②



Na maioria das linguagens de programação, a primeira árvore é preferida.

Pode-se eliminar a ambiguidade alterando a gramática. Ver este exemplo no livro do dragão.