Compiladores 2025/26

Análise Sintática

Raul Barbosa

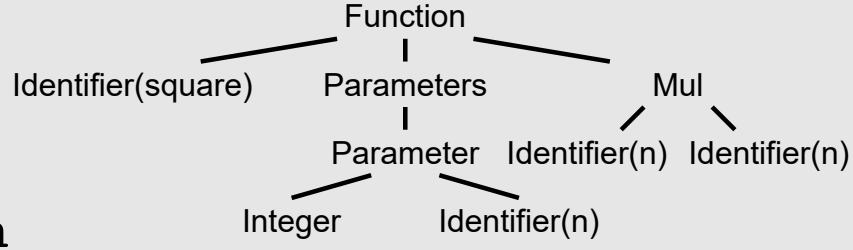
Departamento de Engenharia Informática
Faculdade de Ciências e Tecnologia
Universidade de Coimbra

Análise sintática (*parsing*)

Objetivo: Determinar se o programa respeita a sintaxe da linguagem, *i.e.*, o correto ordenamento dos símbolos lexicais.

- Entrada: Sequência de símbolos lexicais (os *tokens* da fase anterior).
- Saída: Árvore sintática do programa, se não houver erros sintáticos.
- Exemplo:

```
square(integer n) = n * n
```



- Cada nó interior é uma operação e os seus filhos são os argumentos.

As expressões regulares não têm expressividade suficiente.

- Exemplo: A sintaxe dos parêntesis equilibrados $\{ (^i)^i, i \geq 0 \}$

Expressar a sintaxe de linguagens de programação

A análise sintática é a segunda fase de compilação.

- Nem todas as sequências de símbolos lexicais são válidas.
- Em geral, a sintaxe das linguagens de programação é especificada através de gramáticas livres de contexto.
- A tarefa principal do "parser" é distinguir entre sequências sintaticamente válidas e inválidas.
- Dada uma sequência s de símbolos e uma gramática G, determinar se $s \in L(G)$ com eficiência.

Gramáticas livres de contexto

Terminais, não-terminais, um símbolo inicial e produções.

1. Os terminais T são os nomes dos símbolos léxicos, isto é, as categorias dos *tokens* que vêm da análise lexical. Por exemplo, são terminais os símbolos `(`, `)`, `=`, `if`, `else`, *etc.*
2. Os não-terminais NT são variáveis sintáticas, usadas nas produções, que representam conjuntos de sequências.
3. Um símbolo não-terminal é designado por símbolo inicial da gramática, que denota a linguagem gerada pela gramática. Convencionalmente é o não-terminal listado em primeiro lugar.
4. As produções são regras de substituição $\mathrm{NT} \rightarrow (\mathrm{NT} \cup \mathrm{T})*$ nas quais o "lado esquerdo" pode ser substituído pelo "lado direito".

Exemplo: gramática de expressões aritméticas

$$
\begin{array}{lrll}
(1) & E & \rightarrow & id \\
(2) & & | & E + E \\
(3) & & | & E - E \\
(4) & & | & E * E \\
(5) & & | & E / E \\
(6) & & | & (E)
\end{array}
$$

Derivação

1. Uma derivação de qualquer frase começa com o símbolo inicial.
2. Substitui-se um qualquer não-terminal pelo lado direito de uma das produções desse não-terminal.
3. Repete-se o passo 2. até que haja apenas terminais.

Exemplo: $E \underset{(2)}{\rightarrow} E + E \underset{(1)}{\rightarrow} id + E \underset{(4)}{\rightarrow} id + E * E \underset{(1)}{\rightarrow} id + id * E \underset{(1)}{\rightarrow} id + id * id$

Árvores de derivação

$$
\begin{array}{lll}
(1) & E \rightarrow & id \\
(2) & \mid & E + E \\
(3) & \mid & E - E \\
(4) & \mid & E * E \\
(5) & \mid & E / E \\
(6) & \mid & (E)
\end{array}
$$

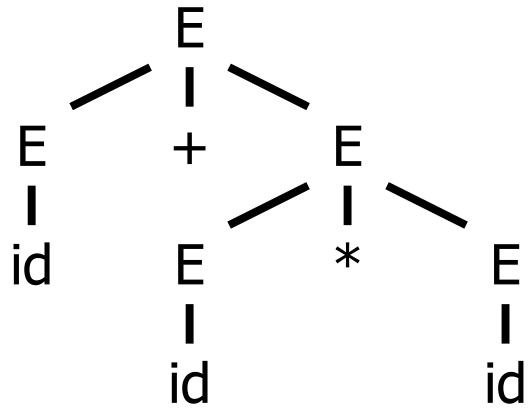
Uma derivação pode ser representada na forma de árvore.

- A raiz da árvore é o símbolo inicial da gramática
- Ao aplicar a produção $X \rightarrow Y_1 \cdots Y_n$ acrescenta-se ao nó X os filhos $Y_1 \cdots Y_n$
- As folhas da árvore são os terminais (podemos ler *em ordem*).

Derivação

E
$E + E$
$id + E$
$id + E * E$
$id + id * E$
$id + id * id$

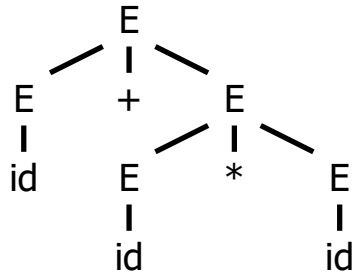
Árvore de derivação



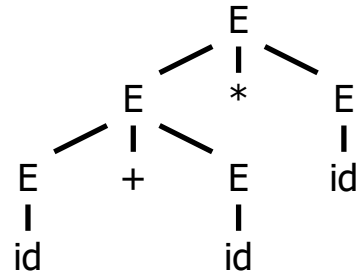
Ambiguidade

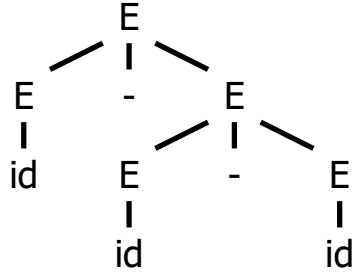
Uma gramática G é ambígua se alguma frase pertencente a $L(G)$ tiver mais do que uma árvore de derivação possível.

- É indesejável porque se deixaria ao acaso como compilar.

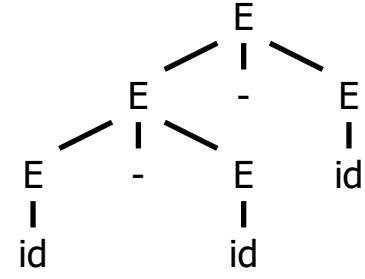


A frase $id + id * id$ tem duas árvores de derivação (precedência ambígua).





A frase $id - id - id$ também (associatividade ambígua).



Eliminar ambiguidades

Para se tornar esta gramática não-ambígua é necessário:

- Estabelecer níveis de precedência de operadores, mais elevado para $( )$, intermédio para $* /$ e mais baixo para $+ -$
- Impor uma associatividade preferencial (à esquerda, no caso dos operadores aritméticos, por convenção)
- Para tal, introduz-se não-terminais para Termos e Fatores

Gramática ambígua

$$
\begin{array}{rcl}
E & \rightarrow & id \\
 & | & E + E \\
 & | & E - E \\
 & | & E * E \\
 & | & E / E \\
 & | & (E)
\end{array}
$$

Gramática não-ambígua

$$
\begin{array}{rcl}
E & \rightarrow & E + T \\
 & | & E - T \\
 & | & T \\
T & \rightarrow & T * F \\
 & | & T / F \\
 & | & F \\
F & \rightarrow & id \\
 & | & (E)
\end{array}
$$

Fim de ficheiro

Convenciona-se que o eof (fim de ficheiro) se representa com o terminal \$ que pode facilmente ser incluído na gramática:

$$
\begin{array}{lcl}
S & \rightarrow & E\ \$ \\
E & \rightarrow & E + T \\
 & | & E - T \\
 & | & T \\
T & \rightarrow & T * F \\
 & | & T / F \\
 & | & F \\
F & \rightarrow & id \\
 & | & (\ E\ )
\end{array}
$$

Encontrar uma derivação

Análise sintática (*parsing*)

Dada uma sequência s de símbolos léxicos e uma gramática G, encontrar uma derivação em G que produza s.

Análise sintática descendente recursiva:

- A árvore é construída a partir da raiz na direção das folhas.
- A derivação é realizada pela esquerda.
- Este algoritmo funciona por tentativa e erro (*backtracking*):
  - Começar na raiz com o símbolo inicial;
  - Para cada nó não-terminal, expandir uma das suas produções;
  - Avançar caso os terminais expandidos sejam os da sequência s;
  - Caso contrário, voltar atrás e tentar expandir outra produção.

Este algoritmo é ineficiente e, para algumas gramáticas, no pior caso não termina.

Recursividade à esquerda

Uma gramática é recursiva à esquerda se permitir a algum não-terminal gerar-se a si mesmo, recursivamente, como o símbolo mais à esquerda.

Um analisador sintático descendente pode entrar em ciclo infinito, expandindo regras sem gerar terminais que permitam avançar.

- Por exemplo, começando com E e tentando gerar *id*...
- Expande a regra $E \rightarrow E + T$ e está de novo em E a tentar gerar *id*...

Pode reescrever-se a gramática usando recursividade à direita:

$$\begin{array}{rcl} A & \rightarrow & A\,\alpha \\ & | & \beta \end{array}$$

$$\begin{array}{rcl} A & \rightarrow & \beta\,A' \\ A' & \rightarrow & \alpha\,A' \\ & | & \epsilon \end{array}$$

Eliminação da recursividade à esquerda

Gramática original e gramática equivalente após transformação:

$$
\begin{array}{lcl}
E & \rightarrow & E + T \\
 & | & E - T \\
 & | & T \\
T & \rightarrow & T * F \\
 & | & T / F \\
 & | & F \\
F & \rightarrow & id \\
 & | & (E) \\
\end{array}
$$

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & + T\ E' \\
 & | & - T\ E' \\
 & | & \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & * F\ T' \\
 & | & / F\ T' \\
 & | & \epsilon \\
F & \rightarrow & id \\
 & | & (E) \\
\end{array}
$$

O caso particular das gramáticas preditivas

$$
\begin{aligned}
E &\rightarrow T\ E' \\
E' &\rightarrow +\ T\ E' \\
&\mid -\ T\ E' \\
&\mid \epsilon \\
T &\rightarrow F\ T' \\
T' &\rightarrow *\ F\ T' \\
&\mid /\ F\ T' \\
&\mid \epsilon \\
F &\rightarrow id \\
&\mid (\ E\ )
\end{aligned}
$$

Uma gramática preditiva é analisável sem tentativa e erro.

- *Backtrack-free grammars* ou *predictive grammars.*
- Considera-se o não-terminal atual e um símbolo de *lookahead*.
- Esse símbolo é suficiente para determinar a produção correta.
- Estas gramáticas caraterizam-se por:
  - não terem recursividade à esquerda,
  - serem não-ambíguas e
  - para cada não-terminal, terem terminais distintos como primeiro símbolo de cada produção.

As gramáticas preditivas prestam-se à análise sintática descendente recursiva.

Analisador sintático descendente recursivo

$$
\begin{aligned}
E &\rightarrow T\ E' \\
E' &\rightarrow +\ T\ E' \\
&\mid -\ T\ E' \\
&\mid \epsilon \\
T &\rightarrow F\ T' \\
T' &\rightarrow *\ F\ T' \\
&\mid /\ F\ T' \\
&\mid \epsilon \\
F &\rightarrow id \\
&\mid (\ E\ )
\end{aligned}
$$

```
enum Token {ADD, SUB, MUL, DIV, ID, LPAR, RPAR, EOF};

enum Token token;

void advance() { token = nextToken(); }

void match(enum Token t) {
  if(t==token) advance(); else error();
}

int main(void) {
  token = nextToken();
  E();
  match(EOF);
}

void E() { T(); EPrime(); }

void EPrime() {
  switch(token) {
    case ADD: match(ADD); T(); EPrime(); break;
    case SUB: match(SUB); T(); EPrime(); break;
    case RPAR: case EOF: break;
    default: error();
  }
}
```

```
void T() { F(); TPrime(); }

void TPrime() {
  switch(token) {
    case MUL: match(MUL); F(); TPrime(); break;
    case DIV: match(DIV); F(); TPrime(); break;
    case ADD: case SUB: case RPAR: case EOF: break;
    default: error();
  }
}

void F() {
  switch(token) {
    case ID: match(ID); break;
    case LPAR: match(LPAR); E(); match(RPAR); break;
    default: error();
  }
}
```

Análise sintática preditiva

- Um analisador descendente recursivo é também chamado *preditivo* quando não requer tentativa e erro.
- É a extensão natural de um DFA à análise sintática.
- Um único *lookahead token* indica qual a produção a usar.
- A pilha das chamadas recursivas preserva os contextos relativos aos lados direitos da produções.
- Para cada não-terminal, cada produção começa à direita por um terminal diferente.
- Aplicável a gramáticas preditivas (nem sempre é o caso).

Fatorização à esquerda

É o processo de isolar prefixos comuns num conjunto de produções.

Acrescentando chamadas de funções à gramática anterior:

$$
\begin{aligned}
F &\rightarrow id \\
&\mid id\ (\ Args\ ) \\
Args &\rightarrow E\ Arg \\
Arg &\rightarrow ,\ E\ Arg \\
&\mid \epsilon
\end{aligned}
$$

É necessário separar o prefixo comum de F dos sufixos distintos:

$$
\begin{aligned}
F &\rightarrow id\ OptArgs \\
OptArgs &\rightarrow (\ Args\ ) \\
&\mid \epsilon \\
Args &\rightarrow E\ Arg \\
Arg &\rightarrow ,\ E\ Arg \\
&\mid \epsilon
\end{aligned}
$$

Exercício

Exemplo clássico de ambiguidade em linguagens de programação:

$$
\begin{array}{rcl}
Stmt & \rightarrow & if\ E\ then\ Stmt \\
 & | & if\ E\ then\ Stmt\ else\ Stmt \\
 & | & OtherStmt
\end{array}
$$

Existem duas derivações distintas para a seguinte frase:

if E_1 then if E_2 then $OtherStmt_1$ else $OtherStmt_2$

É necessário eliminar a ambiguidade, uma vez que as duas derivações resultam em comportamentos distintos.

Resolução da ambiguidade

Estabelece-se uma regra relativamente a qual if controla cada else:

$$
\begin{aligned}
Stmt &\rightarrow \; if\ E\ then\ Stmt \\
&\quad\ |\ \; if\ E\ then\ WithElse\ else\ Stmt \\
&\quad\ |\ \; OtherStmt \\
WithElse &\rightarrow \; if\ E\ then\ WithElse\ else\ WithElse \\
&\quad\ |\ \; OtherStmt
\end{aligned}
$$

Cada else associa-se ao if mais interior não fechado.

Compiladores 2025/26

Análise Sintática

Raul Barbosa

Departamento de Engenharia Informática
Faculdade de Ciências e Tecnologia
Universidade de Coimbra