Judson Santos Santiago

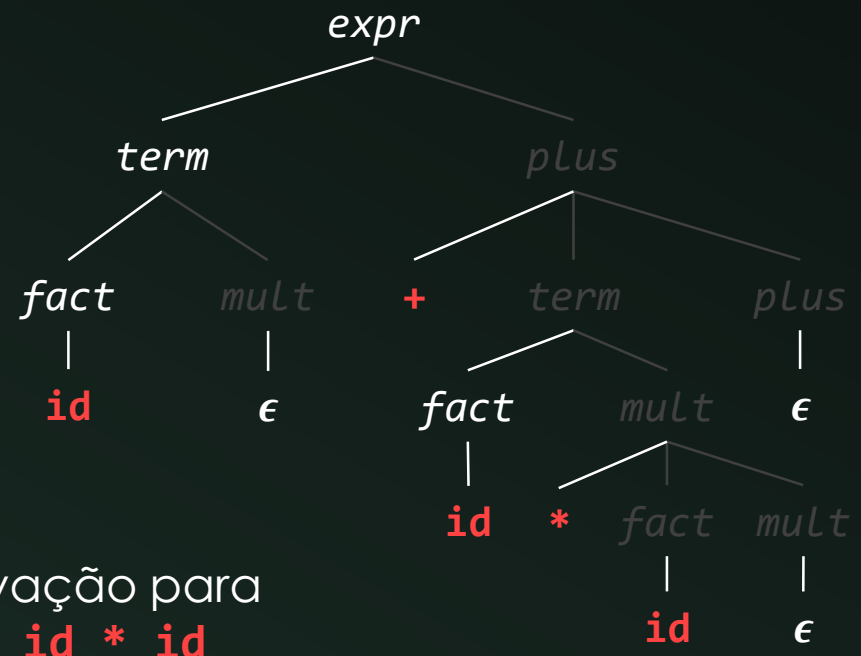
Análise Descendente

Compiladores

Introdução

- A análise descendente constrói uma árvore de derivação
  - De cima para baixo
  - Da esquerda para direita
  - Produz uma derivação mais à esquerda

```
expr   → term plus
plus   → + term plus
       | ϵ
term   → fact mult
mult   → * fact mult
       | ϵ
fact   → (expr)
       | id
```

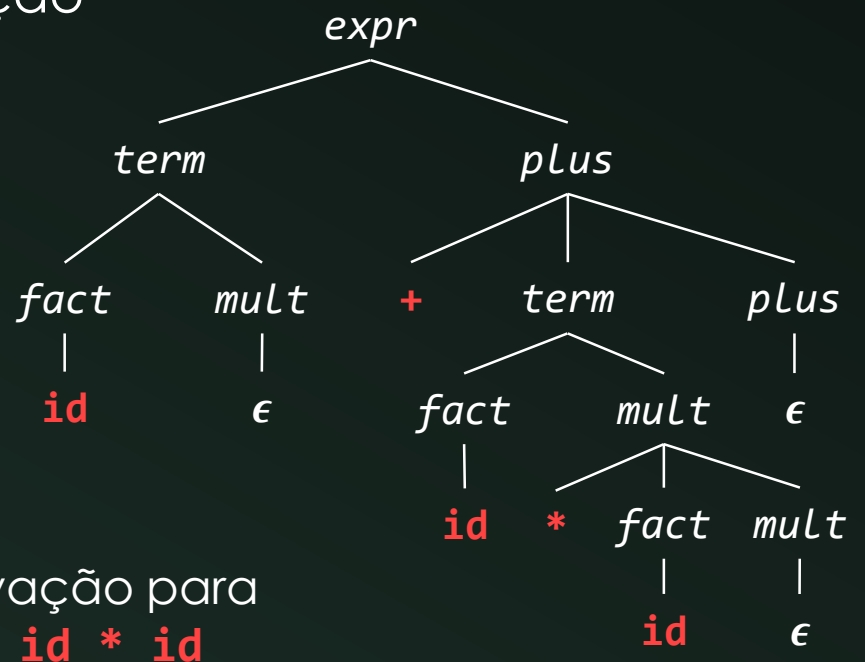


Derivação para
id + id * id

Introdução

- As principais tarefas da análise descendente são:
  - Determinar a produção a ser aplicada em cada derivação
  - Casar os símbolos terminais da produção

```
expr   → term plus
plus   → + term plus
       | ϵ
term   → fact mult
mult   → * fact mult
       | ϵ
fact   → (expr)
       | id
```



Derivação para
id + id * id

Introdução

- Essas tarefas podem ser realizadas através:
  - Análise sintática de descida recursiva
    - Seleciona produção com base em tentativa e erro
    - Pode precisar retroceder na cadeia de entrada
  - Análise sintática preditiva
    - Um caso especial da descida recursiva
    - Decide a partir do próximo símbolo da entrada
    - Conjuntos FIRST devem ser disjuntos

Descida Recursiva

- O método de análise sintática de descida recursiva consiste em:
  - Um conjunto de funções
    - Uma para cada símbolo não-terminal A

```
void A() {
    escolha uma produção-A, A → X1 X2 ... Xk;
    para (i = 1 até k) {
        se (Xi é um não-terminal)
            chama função Xi();
        senão se (Xi é igual ao símbolo da entrada)
            avance na entrada para o próximo símbolo;
        senão
            erro();
    }
}
```

Descida Recursiva

- A análise sintática de descida recursiva
  - Começa pela função que representa o símbolo inicial
  - Obtém sucesso se consumir toda a cadeia de entrada

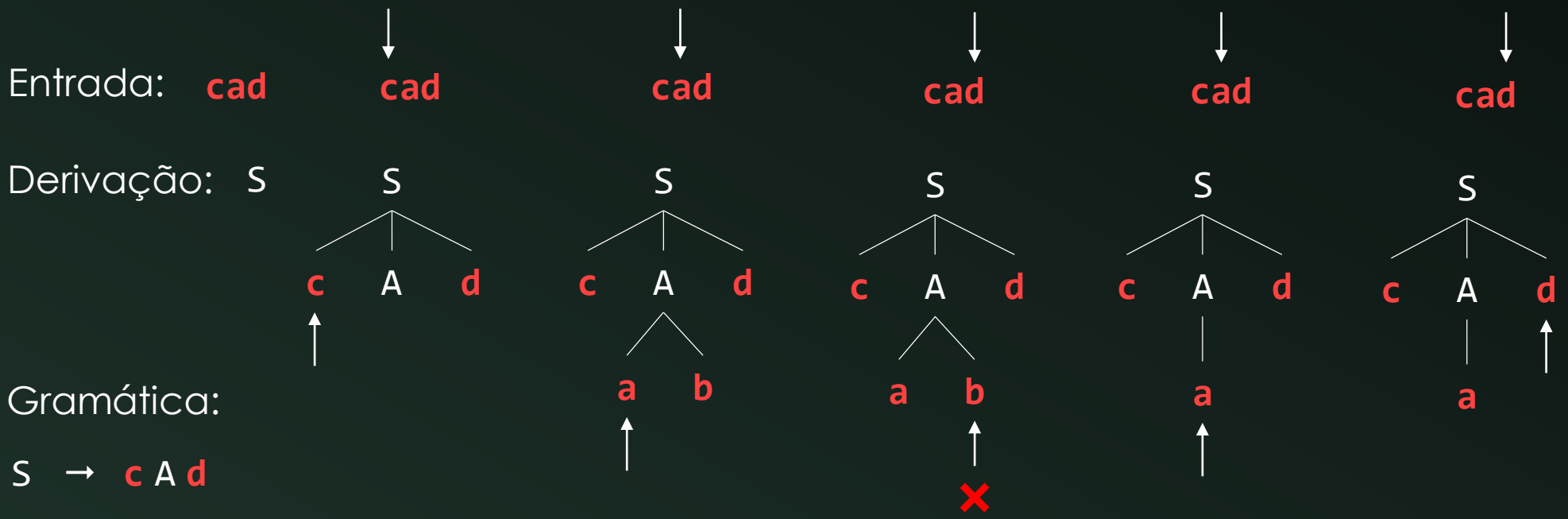
  - A escolha da produção é não-determinística:

    `escolha uma produção-A, A → X1 X2 ... Xk;`

    - Pode ser necessário retroceder para escolher outra produção
    - O erro só acontece depois de todas as produções serem testadas
    - Um apontador para a entrada precisa ser guardado na função

Descida Recursiva

- Para construir uma árvore de derivação descendente para **cad**



FIRST e FOLLOW

- Analisadores descendentes mais eficientes podem ser obtidos com o auxílio de duas funções:
  - FIRST
  - FOLLOW
- As funções são vinculadas a uma gramática e permitem:
  - Escolher que produção aplicar com base nos símbolos da entrada
  - Gerar tokens de sincronismo para a recuperação de erros
    - Quando o tratamento de erros utilizar o modo pânico

FIRST

- FIRST(α) é o conjunto de símbolos terminais que iniciam as cadeias derivadas a partir de α
  - Sendo α qualquer cadeia de símbolos da gramática
  - Se $\alpha \overset{*}{\Rightarrow} \epsilon$ então ϵ também está em FIRST(α)

S → A a
A → c γ

A ⇒ c γ
Portanto c está em FIRST(A)

Se FIRST(α) e FIRST(β) são conjuntos disjuntos, então não há dúvidas na escolha da produção em
A → α | β

FIRST

- Como calcular o FIRST de todos os símbolos da gramática?
  - Aplique as regras até que não seja mais possível acrescentar símbolos terminais ou ϵ a nenhum dos conjuntos FIRST:
    1. Se **a** é um terminal, então `FIRST(a) = {a}`
    2. Se X é um não-terminal e $X \rightarrow Y_1\ Y_2\ Y_3\ Y_4\ \ldots\ Y_k$ é uma produção:
       - Adicione o símbolo **a** em `FIRST(X)` se, para algum `i`, **a** estiver em $FIRST(Y_i)$ e ϵ estiver em todos os $FIRST(Y_1)$ ... $FIRST(Y_{i-1})$, ou seja, $Y_1 \ldots Y_{i-1} \Rightarrow \epsilon$
       - Adicione ϵ a FIRST(X) se ϵ está em FIRST(Y_i) para todo `i = 1,2, ..., k`
    3. Se $X \rightarrow \epsilon$ é uma produção, então acrescente ϵ a `FIRST(X)`

FIRST

- Considerando a gramática abaixo:

```
expr   → term plus
plus   → + term plus
       | ϵ
term   → fact mult
mult   → * fact mult
       | ϵ
fact   → (expr)
       | id
```

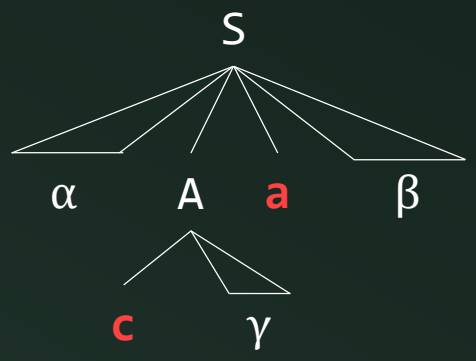
```
FIRST(id) = { id }
FIRST(() = { ( }
FIRST()) = { ) }
FIRST(+) = { + }
FIRST(*) = { * }

FIRST(fact) = { (, id }
FIRST(term) = { (, id }
FIRST(expr) = { (, id }
FIRST(plus) = { +, ϵ }
FIRST(mult) = { *, ϵ }

FIRST(term plus) = { (, id }
FIRST(+ term plus) = { + }
FIRST(ϵ) = { ϵ }
FIRST(fact mult) = { (, id }
FIRST(* fact mult) = { * }
FIRST((expr)) = { ( }
```

FOLLOW

- FOLLOW(A) é o conjunto de terminais que podem aparecer imediatamente à direita de A em alguma forma sentencial
  - Sendo A um não-terminal
  - Se A for o símbolo mais a direita em alguma forma sentencial, então $ estará em `FOLLOW(A)`



$S \Rightarrow \alpha A \mathbf{a} \beta$

Portanto **a** está em `FOLLOW(A)`

FOLLOW

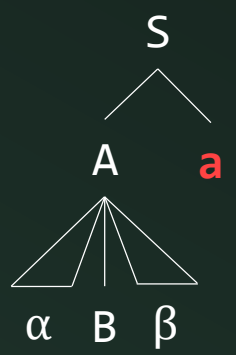
- Como calcular FOLLOW para todos os não-terminais?
  - Aplique as regras até que nada mais possa ser acrescentado a nenhum dos conjuntos `FOLLOW`:

    1. Coloque $ em `FOLLOW(S)`, onde S é o símbolo inicial da gramática
    2. Se houver uma produção `A → α B β`, então tudo em `FIRST(β)`, exceto ϵ, está em `FOLLOW(B)`
    3. Se houver uma produção `A → α B`, ou uma produção `A → α B β`, onde o `FIRST(β)` contém ϵ, então tudo em `FOLLOW(A)` está em `FOLLOW(B)`

FOLLOW

- Essa terceira regra precisa de um exemplo:

3. Se houver uma produção `A → α B`, ou uma produção `A → α B β`, onde o `FIRST(β)` contém ϵ, então tudo em `FOLLOW(A)` está em `FOLLOW(B)`



`FOLLOW(A)`

`S → Aa`
`A → αBβ`

Substituindo `A` por `αBβ` em `S`
`S → αBβa`

`FOLLOW(B)` quando $\beta \Rightarrow \epsilon$

FOLLOW

- Considerando a gramática abaixo:

```
expr   → term plus
plus   → + term plus
       | ϵ
term   → fact mult
mult   → * fact mult
       | ϵ
fact   → (expr)
       | id
```

```
FIRST(id) = { id }
FIRST(() = { ( }
FIRST()) = { ) }
FIRST(+) = { + }
FIRST(*) = { * }
```

```
FIRST(fact) = { (, id }
FIRST(term) = { (, id }
FIRST(expr) = { (, id }
FIRST(plus) = { +, ϵ }
FIRST(mult) = { *, ϵ }
```

```
FOLLOW(expr) = { $, ) }
FOLLOW(plus) = { $, ) }
FOLLOW(term) = { +, $, ) }
FOLLOW(mult) = { +, $, ) }
FOLLOW(fact) = { *, +, $, ) }
```

Exercício

1. Calcule os conjuntos FIRST e FOLLOW da gramática abaixo:

```
S -> SS+
   | SS-
   | a
```

```
FIRST(a) = { a }
FIRST(+) = { + }
FIRST(-) = { - }
```

```
FIRST(S) = { a }
FIRST(SS+) = { a }
FIRST(SS-) = { a }
```

```
FOLLOW(S) = { $, a, +, - }
```

Resumo

- A análise descendente constrói a árvore de derivação
  - De raiz para as folhas
  - Pode ser feita pelos métodos:
    - Descida recursiva (pode haver retrocesso)
      - Uma função para cada não-terminal
    - Reconhecimento preditivo (sem retrocesso)
      - Pode usar recursão ou uma tabela de análise sintática com uma pilha
  - As funções FIRST e FOLLOW auxiliam
    - Na construção dos analisadores
    - Na detecção e recuperação de erros