



Judson Santos Santiago

# Análise Ascendente



Compiladores

# Introdução

- A **análise descendente** é empregada em muitos compiladores
  - MSVC++, Clang e GCC (C, C++, Objective-C, Fortran, Ada, e Go)
    - São compiladores **escritos à mão** usando descida recursiva
- A **análise ascendente** possui a vantagem de permitir a construção de analisadores sintáticos de forma automática
  - Muitas linguagens usam (**ou já usaram**) **geradores de analisadores**
    - Ruby, PHP, Haskell, Cobol, Perl
    - **C, C++, Objective-C, Fortran, Ada, e Go**

# Análise Ascendente

- A análise ascendente corresponde à construção de uma árvore de derivação a partir das folhas até a raiz

*expr*  *expr* + *term*  
| *term*  
*term*  *term* \* *fact*  
| *fact*  
*fact*  ( *expr* )  
| *id*

Análise  
Ascendente  
*id* \* *id*

*id* \* *id*

*fact* \* *id*  
|  
*id*

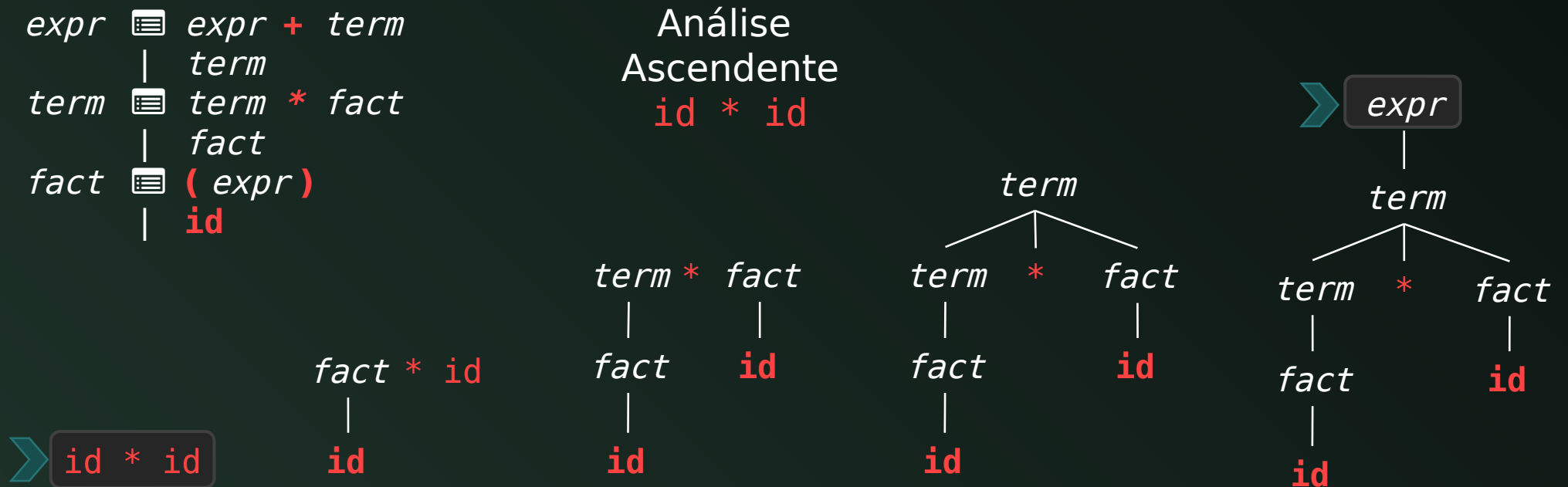
*term* \* *fact*  
|      |  
*fact*   *id*  
|  
*id*

*term*  
/    |    \  
*term* \* *fact*  
|      |  
*fact*   *id*  
|  
*id*

*expr*  
|  
*term*  
/    |    \  
*term* \* *fact*  
|      |  
*fact*   *id*  
|  
*id*

# Análise Ascendente

- O processo de análise pode ser pensado como a **redução** de uma cadeia de entrada para o símbolo inicial da gramática



# Análise Ascendente

- Em cada passo da redução, uma subcadeia **casando com o lado direito** de uma produção é **substituída pelo não-terminal** na cabeça desta produção
  - As principais decisões da análise ascendente em cada passo:
    - Determinar quando reduzir
    - Determinar a produção a ser utilizada




```
expr  [ ] expr + term
      | term
term  [ ] term * fact
      | fact
fact  [ ] ( expr )
      | id
```

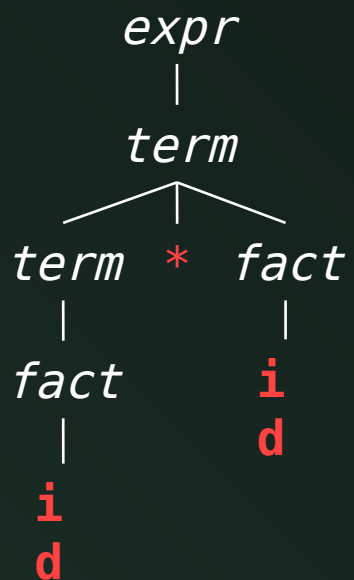
## Sequência de Reduções

```
expr
term
term * fact
term * id
fact * id
id * id
```

# Análise Ascendente

- Uma **redução** é o inverso de um passo de **derivação**
  - O método **ascendente** constrói uma **derivação** mais à direita invertida

*expr*  *expr* + *term*  
| *term*  
*term*  *term* \* *fact*  
| *fact*  
*fact*  ( *expr* )  
| *id*



## Descendent

*e*  
*expr*  
*term*  
*term* \* *fact*  
*term* \* *id*  
*fact* \* *id*  
*id* \* *id*

## Ascendent

*e*  
*id* \* *id*  
*fact* \* *id*  
*term* \* *id*  
*term* \* *fact*  
*term*  
*expr*

# Análise Ascendente

- Existem **vários métodos** de análise ascendente
  - LR (Left-right Rightmost derivation)
  - SLR (Simple LR)
  - GLR (Generalized LR)
  - LALR (Look-Ahead LR)
- Todos eles são analisadores **Shift-Reduce**
  - A análise Shift-Reduce é um método de **análise ascendente** comumente utilizada nos geradores automáticos
    - Yacc, Bison, etc.



# Analizador Shift-Reduce

- Um **handle** é uma **cadeia que casa com o corpo de uma produção** e cuja redução para o não-terminal da cabeça representa um passo da derivação mais à direita invertida

```

exp  ::= expr +
r    ::= term
      | term
ter  ::= term *
m    ::= fact
      | fact
fac  ::= ( expr )
t    ::= id



```

Entrada	Handle	Produção de Redução
$id_1 * id_2$	$id_1$	$fact \Rightarrow id$
$fact * id_2$	$fact$	$term \Rightarrow fact$
$term * id_2$	$id_2$	$fact \Rightarrow id$
$term * fact$	$term * fact$	$term \Rightarrow term * fact$
$term$	$term$	$expr \Rightarrow term$
$expr$		








# Analizador Shift-Reduce

- O analisador sintático **shift-reduce** utiliza uma pilha e um buffer

$expr$    $expr +$   
                   $term$   
          |  $term$   
 $term$    $term *$   
                   $fact$   
          |  $fact$   
 $fact$    $( expr )$   
          |  $id$

O handle sempre  
aparece no topo  
da pilha

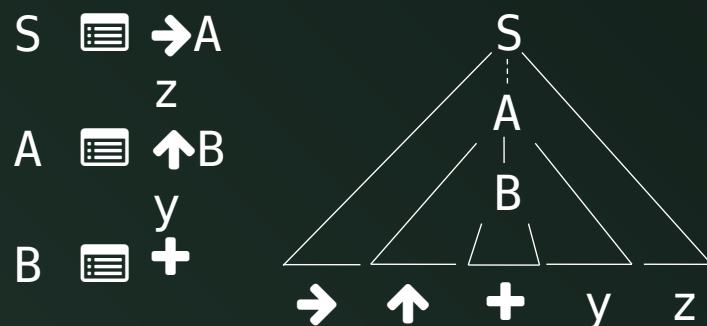
Pilha	Entrada	Ação
\$	$id_1 * id_2$ \$	shift
\$ $id_1$	$* id_2$ \$	reduce ( $fact$  $id$ )
\$ $fact$	$* id_2$ \$	reduce ( $term$  $fact$ )
\$ $term$	$* id_2$ \$	shift
\$ $term *$	$id_2$ \$	shift
\$ $term * id_2$	\$	reduce ( $fact$  $id$ )
\$ $term *$ $fact$	\$	reduce ( $term$  $term * fact$ )
\$ $term$	\$	reduce ( $expr$  $term$ )
\$ $expr$	\$	accept

# Analizador Shift-Reduce

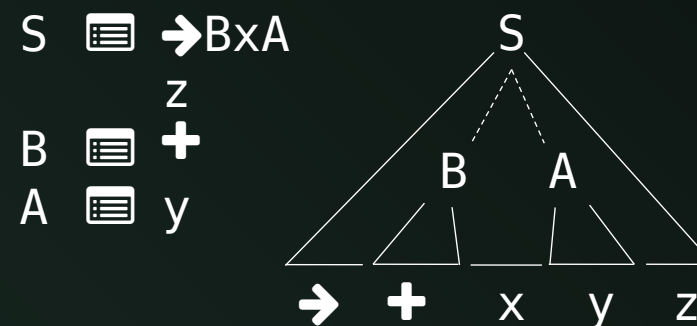
- Existem 4 ações possíveis para o analisador shift-reduce:
  - **Shift**: transfere o próximo símbolo da entrada para o topo da pilha
  - **Reduce**: decide por qual não-terminal a cadeia que está no topo da pilha deve ser reduzida
  - **Accept**: anuncia o término bem sucedido da análise sintática
  - **Error**: chama uma rotina de recuperação de erro ao descobrir um erro na sintaxe das cadeias de entrada

# Analizador Shift-Reduce

- O **handle** sempre aparece no **topo da pilha do reconhecedor**, nunca em seu interior



Pilha	Entrada
\$ → ↑ +	yz\$
\$ → ↑ B	yz\$
\$ → ↑ By	z\$
\$ → A	z\$



Pilha	Entrada
\$ → +	xyz\$
\$ → B	xyz\$
\$ → Bxy	z\$
\$ → BxA	z\$

# Exercício

Mostre os passos de reconhecimento do analisador shift-reduce, indicando o handle em cada passo, para as seguintes entradas e gramáticas:

1.  $S \rightarrow 0S1 \mid 01$

a) 000111

2.  $S \rightarrow SS+ \mid SS^* \mid a$


a)  $SSS+a^*+$

b)  $SS+a^*a+$

c)  $aaa^*a++$

# Conflitos Shift-Reduce

- Existem gramáticas para as quais a análise shift-reduce não pode ser aplicada:
  - Nessas gramáticas, conhecendo todo o conteúdo da pilha e a próxima entrada, não é possível decidir entre:
    - Empilhar e avançar (shift)
    - Ou reduzir (reduce)

```
stmt   if expr then stmt
      | if expr then stmt else
      | stmt
      | other
```

Pilha	Entrada
\$... if expr then stmt	else ... \$

# Conflitos Shift-Reduce

- A gramática do exemplo anterior não é LR(1)
  - É preciso **olhar mais de um token a frente** para decidir
- Gramáticas LR(1):
  - O L indica que a entrada é **varrida da esquerda para a direita**
  - O R indica que é feita uma **derivação mais à direita invertida**
  - O "1" diz respeito a **quantidade de símbolos à frente**
- A análise pode ser adaptada para **resolver certos conflitos**
  - O conflito pode ser resolvido sempre pela transferência, por exemplo

# Conflitos Reduce-Reduce

- Outro tipo de conflito ocorre quando há **mais de uma produção casando com um handle**
  - Nessas gramáticas, conhecendo todo o conteúdo da pilha e a próxima entrada, **não é possível decidir** entre qual produção reduzir (reduce)

```
inst  📄 id ( parameter_list )
      | expr := expr
parameter_list  📄 parameter_list, parameter
                | parameter
parameter       📄 id
expr            📄 id ( expr_list )
                | id
expr_list       📄 expr_list, expr
                | expr
```

Pilha	Entrada
\$... id <sub>1</sub> ( id <sub>2</sub>	, id <sub>3</sub> ) ... \$

**id<sub>2</sub>** deve ser reduzido para  
um *parameter* ou uma *expr*?

Depende se **id<sub>1</sub>** é uma  
função ou um arranjo.



# Analísadores LR

- O tipo mais comum de analisadores sintáticos ascendentes é baseado em reconhecedores LR(k)
  - Na prática,  $k = 1$  é suficiente
  - O valor ótimo de  $k$  é igual a 1
- Intuitivamente, para uma gramática ser LR é suficiente que um analisador shift-reduce seja capaz de reconhecer o handle quando este aparecer no topo da pilha

# Resumo

- Os reconhecedores LR:
  - São capazes de reconhecer praticamente todas as construções sintáticas definidas nas linguagens de programação modernas
  - É um método de análise eficiente em espaço e tempo
  - Detecta erros sintáticos tão logo eles apareçam na cadeia de entrada
  - As gramáticas reconhecidas por métodos LR representam um superconjunto das que podem ser reconhecidas com métodos LL
- A principal desvantagem:
  - Sua construção a mão é muito trabalhosa, melhor usar uma ferramenta