

# Compiladores

## Análise Sintática

### Análise Sintática Descendente Recursiva

Prof. Dr. Luiz Eduardo G. Martins  
(adaptado por Profa Dra Ana Carolina Lorena)

UNIFESP

# Análise Sintática

- Verifica se o programa fonte é uma cadeia válida para a linguagem em questão
- A análise sintática depende da análise léxica
- O analisador sintático é frequentemente chamado de *PARSER*
-

# Análise Sintática

- Durante a análise sintática o *parser* “pede” para o *scanner* os *tokens* correspondentes aos lexemas do programa que está sendo analisado
- A análise sintática verifica se a ordem desses *tokens* está de acordo com a GLC

# Análise Sintática Descendente Recursiva

- Existem diversas técnicas para se implementar um analisador sintático
- **Análise sintática descendente**: analisa cadeia de marcas pelo acompanhamento dos passos de uma derivação à esquerda
  - Nome vem da forma como a árvore sintática é percorrida, em pré-ordem (raiz para folhas)

# Análise Sintática Descendente

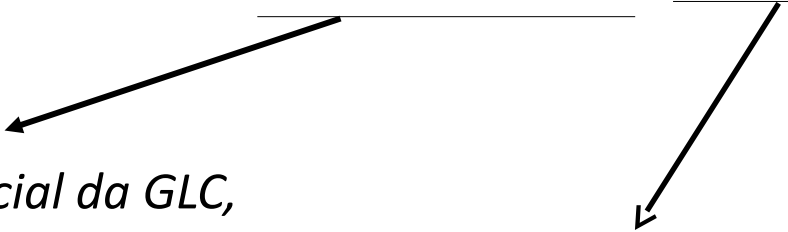
- Há duas formas de construir analisadores descendentes:
  - **Com retrocesso**: testa diferentes possibilidades de análise de entrada, retrocedendo se alguma possibilidade falhar
    - ℓ Mais poderosos, mas lentos (tempo exponencial)
  - **Preditivos**: tenta prever a construção seguinte na entrada com base em uma ou mais marcas de verificação à frente
    - ℓ Foco

# Análise Sintática Descendente Preditiva

- Tipos que estudaremos:
  - **Descendentes recursivos**: versáteis e usados na construção manual de analisadores sintáticos
    - ℓ Para linguagens simples e suficientemente formalizadas
  - **LL(1)**: esquema simples com pilha explícita
    - ℓ Mais poderosos, mas mais complexos

# Análise Sintática Descendente Recursiva

- Dada uma linguagem, definida por uma Gramática Livre de Contexto (GLC), **parte-se do símbolo inicial da GLC**
  - Com sucessivas derivações descendentes, busca-se alcançar sentenças válidas de *tokens*
- Por que o nome Analisador Descendente Recursivo ?



*Começa pelo símbolo inicial da GLC,  
**descendo** até os demais (processo  
de derivação descendente)*

*Cada símbolo não-terminal da GLC  
corresponde a uma função  
**recursiva**, que é responsável por  
verificar se os tokens da entrada  
correspondem ou não àquele não-  
terminal*

# Análise Sintática Descendente Recursiva

- O conceito geral da **análise sintática descendente recursiva** é o seguinte:
  - A regra gramatical para um **A não-terminal** é vista como uma definição de procedimento para reconhecer um **A**
  - O lado direito da regra gramatical para **A** especifica a estrutura do código para esse procedimento
    - A sequência de terminais corresponde a casamentos com a entrada
    - A sequência de não-terminais corresponde a ativações de outros procedimentos
    - As escolhas correspondem a alternativas dentro do código (declarações *case* ou *if*)



# Análise Sintática Descendente Recursiva

- Considere a gramática G1:

$S \rightarrow \text{BEGIN } S L$

$S \rightarrow \text{IF } E \text{ THEN } S \text{ ELSE } S$

$S \rightarrow \text{PRINT ID}$

$L \rightarrow ; S L$

$L \rightarrow \text{END}$

$E \rightarrow \text{ID} = \text{NUM}$

Um analisador recursivo descendente p/ essa linguagem possui um procedimento p/ cada não-terminal, e uma opção “case” p/ cada terminal  
Código nos próximos slides

$\Sigma = \{ \text{ID}, \text{NUM}, \text{IF}, \text{THEN}, \text{ELSE}, \text{BEGIN}, \text{END}, \text{PRINT}, ;, , = \}$

# Análise Sintática Descendente Recursiva

- *Parser* para G1:

```
void S()  
{
```

Vê marca seguinte de entrada (um símbolo de verificação à frente)

```
    switch (tok)
```

```
{
```

```
    case BEGIN : casa(BEGIN); S(); L(); break;
```

```
    case IF : casa(IF); E(); casa(THEN); S(); casa(ELSE); S(); break;
```

```
    case PRINT : casa(PRINT); casa(ID); break;
```

```
    default: ERRO();
```

```
}
```

$S \rightarrow \text{BEGIN } S \ L$

$S \rightarrow \text{IF } E \ \text{THEN } S \ \text{ELSE } S$

$S \rightarrow \text{PRINT } ID$

Casa marca seguinte com seu parâmetro  
(adianta entrada em caso de sucesso e declara erro caso contrário)

# Análise Sintática Descendente Recursiva

- *Parser* para G1:

```
void L()
{
    switch (tok)
    {
        case ; : casa(;); S(); L(); break;

        case END : casa(END); break;

        default: ERRO();
    }
}
```

$L \rightarrow ; S L$ $L \rightarrow \text{END}$
---

# Análise Sintática Descendente Recursiva

- *Parser* para G1:

```
void E()  
{  
    casa(ID); casa(=); casa(NUM);  
}
```

$E \rightarrow ID = NUM$

# Análise Sintática Descendente Recursiva

- *Parser* para G1:

```
void casa(int t)
{
    if (tok == t)
        avance();
    else ERRO();
}
```

```
int getToken()
{
    // chama o analisador léxico e obtém
    // o próximo token do programa fonte
}
```

```
int main()
{
    avance()

    S();
    return 0;
}
```

```
void avance()
{
    tok= getToken();
}
```

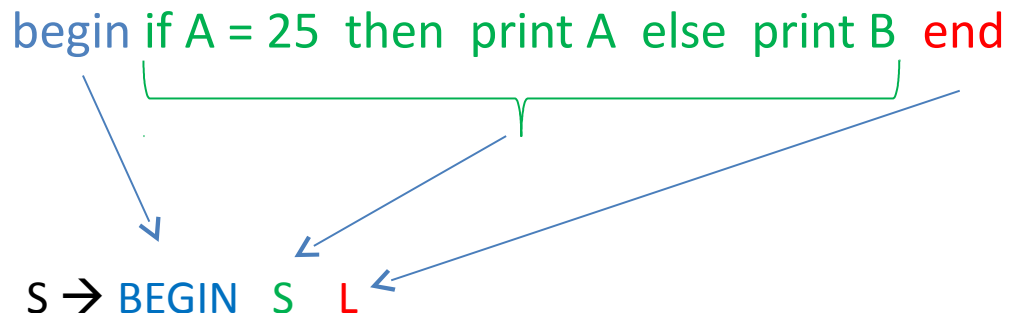
```
void ERRO()
{
    printf("ERRO na Análise Sintática: %d %d", line, tok);
}
```

# Análise Sintática Descendente Recursiva

- Vamos usar a G1 para fazer a análise sintática do seguinte programa:

```
begin
  if A = 25
  then print A
  else print B
end
```

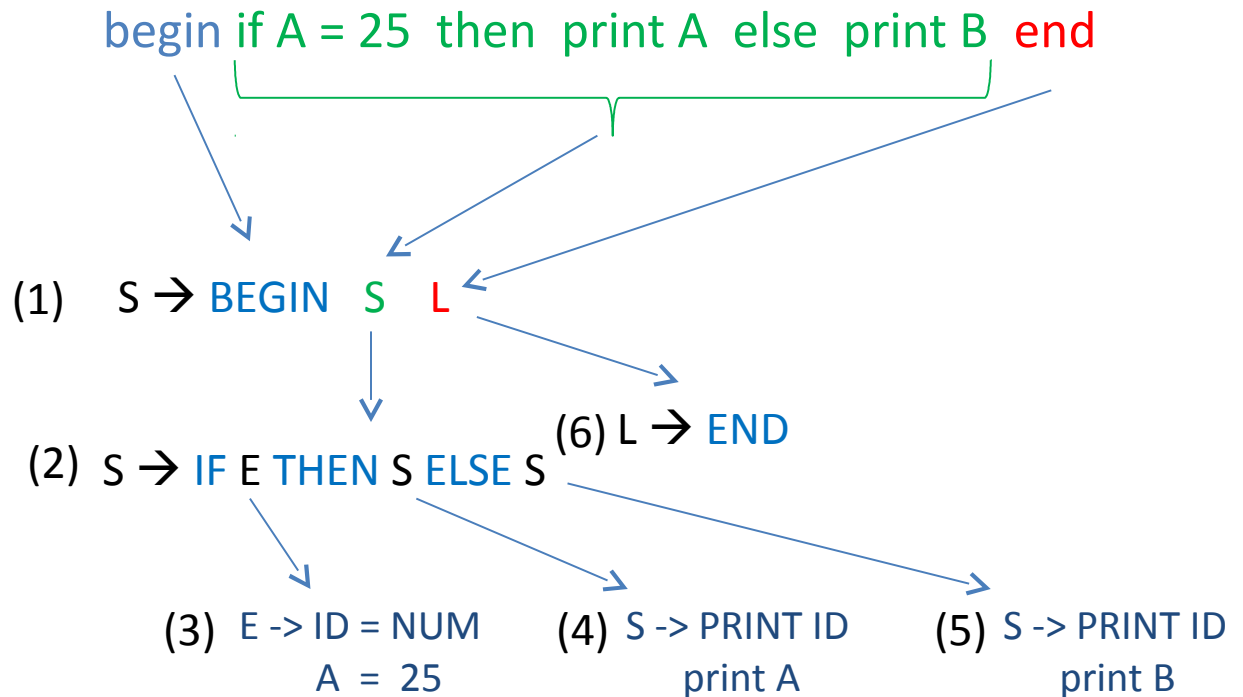
```
begin if A = 25 then print A else print B end
BEGIN IF ID = NUM THEN PRINT ID ELSE PRINT ID END
```



# Análise Sintática Descendente Recursiva

- Derivação descendente à esquerda:

- (1)  $S \rightarrow \text{BEGIN } S \text{ L}$
- (2)  $S \rightarrow \text{IF } E \text{ THEN } S \text{ ELSE } S$
- (3)  $S \rightarrow \text{PRINT ID}$
- (4)  $L \rightarrow ; S \text{ L}$
- (5)  $L \rightarrow \text{END}$
- (6)  $E \rightarrow \text{ID} = \text{NUM}$



# Análise Sintática Descendente Recursiva

- **Exercício 1:** Verifique se a cadeia abaixo faz parte da linguagem gerada pela gramática G1

$S \rightarrow \text{BEGIN } S L$   
 $S \rightarrow \text{IF } E \text{ THEN } S \text{ ELSE } S$   
 $S \rightarrow \text{PRINT ID}$   
 $L \rightarrow ; S L$   
 $L \rightarrow \text{END}$   
 $E \rightarrow \text{ID} = \text{NUM}$

```
begin
  if X = 1000 then
    print X
  else
    print Y;
    print Z;
    print W
end
```

- Faça o teste de mesa para essa cadeia, usando o *parser* da G1



# Análise Sintática Descendente Recursiva

- Agora considere a gramática G2 dada a seguir:

$\text{exp} \rightarrow \text{exp soma termo} \mid \text{termo}$

$\text{soma} \rightarrow + \mid -$

$\text{termo} \rightarrow \text{termo mult fator} \mid \text{fator}$

$\text{mult} \rightarrow * \mid /$

$\text{fator} \rightarrow (\text{exp}) \mid \text{NUM}$

$\Sigma = \{ (, ), +, -, *, /, \text{NUM} \}$

# Análise Sintática Descendente Recursiva

- A tentativa de construir um procedimento descendente recursivo para *exp* levaria ao seguinte código:

```
void exp()  
{  
    exp(); soma(); termo();  
}
```

*exp* → *exp soma termo* | *termo*

- Mas isso levaria a dois problemas:
  - Chamada recursiva infinita de *exp()*
  - Não temos como saber a escolha a ser feita entre *exp soma termo* e *termo*

# Análise Sintática Descendente Recursiva

- A alternativa é reescrevermos a G2 no formato EBNF (*Extended BNF*)
- A EBNF permite substituir as estruturas recursivas por estruturas de repetição e opcionalidade
  - { estrutura } : as chaves correspondem ao \* das expressões regulares
  - [ estrutura ] : os colchetes correspondem a ? das expressões regulares
  - Chaves e colchetes são metacaracteres da EBNF

# Análise Sintática Descendente Recursiva

- Reescrevendo G2 em EBNF

exp  $\rightarrow$  exp soma termo | termo  
soma  $\rightarrow$  + | -  
termo  $\rightarrow$  termo mult fator | fator  
mult  $\rightarrow$  \* | /  
fator  $\rightarrow$  (exp) | NUM

(G2 em BNF)

exp  $\rightarrow$  termo {soma termo}  
soma  $\rightarrow$  + | -  
termo  $\rightarrow$  fator {mult fator}  
mult  $\rightarrow$  \* | /  
fator  $\rightarrow$  (exp) | NUM

(G2 em EBNF)

- A implementação de um parser descendente recursivo normalmente é feita com base em uma gramática escrita em EBNF

# Análise Sintática Descendente Recursiva

- Então a implementação do *parser* para G2 ficaria:

```
void exp()
{
    termo();
    while ((tok == "+") || (tok == "-"))
    {
        casa(tok);
        termo();
    }
}
```

$exp \rightarrow termo \{soma\ termo\}$

**Exercício 2:** Continue a implementação... para termo(), fator() e main(). Considere o uso de casa(), avance(), ERRO() e getToken().

# Análise Sintática Descendente Recursiva

- Então a implementação do *parser* para G2 ficaria:

```
void termo()
{
    fator();
    while ((tok == "*" ) || (tok == "/"))
    {
        casa(tok);
        fator();
    }
}
```

termo → fator {mult fator}

# Análise Sintática Descendente Recursiva

- Então a implementação do *parser* para G2 ficaria:

```
void fator()
{
    switch (tok)
    {
        case ( : casa(); exp(); casa());
        case NUM : casa(NUM);
        default : ERRO();
    }
}
```

fator → (exp) | NUM

```
int main()
{
    avance();
    exp();
    return 0;
}
```

# Análise Sintática Descendente Recursiva

- Código gerando a árvore sintática:

```
árvore exp()  
{  
    árvore temp, novatemp;  
    temp = termo();  
    while ((tok == "+") || (tok == "-"))  
    {  
        novatemp = criaNóOp(tok);  
        casa(tok);  
        filhoEsq(novatemp) = temp;  
        filhoDir(novatemp) = termo();  
        temp = novatemp;  
    }  
    return temp;  
}
```

$exp \rightarrow termo \{soma\ termo\}$



# Análise Sintática Descendente Recursiva

- Considere a gramática G3:

- $\text{if-decl} \rightarrow \text{if (exp) decl} \mid$   
 $\text{if (exp) decl else decl}$

- Tem duas regras começando com if, qual escolher?

- Em EBNF:

- ℓ  $\text{if-decl} \rightarrow \text{if (exp) decl} [\text{else decl}]$

- ℓ Opção é traduzida em teste no código

# Análise Sintática Descendente Recursiva

- Parser para G3:

```
void if-decl()
{
    casa(if);
    casa();
    exp();
    casa());
    decl();
    if (tok == else)
    {
        casa(tok);
        decl();
    }
}
```

if-decl  $\rightarrow$  if (exp) decl [else decl]

Efeito é de adiar a decisão de reconhecer a parte opcional else

Solução corresponde também à regra de eliminação de ambiguidade pelo aninhamento mais próximo

# Análise Sintática Descendente Recursiva

- Código gerando a árvore sintática:

```
árvore if-decl()  
{  
    árvore temp;  
    casa(if);  
    casa();  
    temp = criaNóDecl(if);  
    testeFilho(temp) = exp();  
    casa();  
    thenFilho(temp) = decl();  
    if (tok == else)  
    {  
        casa(tok);  
        elseFilho(temp) = decl();  
    }  
    else  
        elseFilho(temp) = NULL;  
    return temp;  
}
```

if-decl  $\rightarrow$  if (exp) decl [else decl]

# Análise Sintática Descendente Recursiva

- Bibliografia consultada

Capítulo 4: LOUDEN, K. C. **Compiladores: princípios e práticas**. São Paulo: Pioneira Thompson Learning, 2004  
MERINO, M. **Notas de Aulas - Compiladores**, UNIMEP, 2006.