

Compiladores

Aula 11

Análise Sintática

Análise Sintática Descendente Recursiva

Prof. Dr. Luiz Eduardo G. Martins

UNIFESP



Análise Sintática Descendente Recursiva

Análise Sintática:

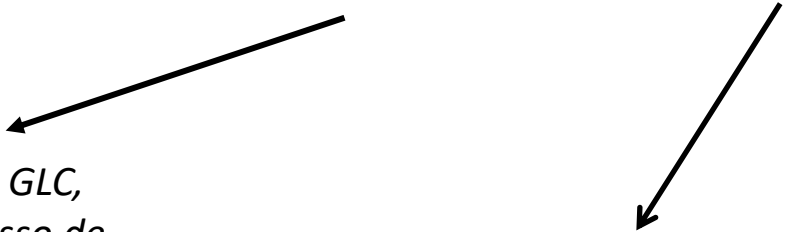
- Verifica se as sentenças do programa fonte são válidas para a linguagem em questão
- A análise sintática depende da análise léxica
- O analisador sintático é frequentemente chamado de *PARSER*
- Durante a análise sintática o *parser* “pede” para o *scanner* os *tokens* correspondentes aos lexemas do 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
- Iremos nos concentrar inicialmente na técnica chamada **Descendente Recursiva**, que é simples e flexível
- Adequada para implementação “manual” do *parser*

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 processar a cadeia de símbolos do lado direito da regra*

Análise Sintática Descendente Recursiva

- O conceito geral da **análise sintática descendente recursiva** é o seguinte:
 - A regra gramatical para um símbolo **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, uma opção “case” p/ cada terminal (começando no lado esquerdo das regras de produção), e um procedimento eat() para consumir 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()
{
    switch (tok)
    {
        case BEGIN : eat (BEGIN); S(); L(); break;

        case IF     : eat (IF); E(); eat (THEN); S(); eat (ELSE); S(); break;

        case PRINT  : eat (PRINT); eat(ID); break;

        default: ERRO();
    }
}
```

```
S → BEGIN S L
S → IF E THEN S ELSE S
S → PRINT ID
```

Análise Sintática Descendente Recursiva

- *Parser* para G1:

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

        case END    : eat (END); break;

        default: ERRO();
    }
}
```

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

Análise Sintática Descendente Recursiva

- *Parser* para G1:

```
void E()  
{  
    switch (tok)  
    {  
        case ID: eat (ID); eat(=); eat(NUM);  
  
        default: ERRO();  
  
    }  
}
```

$E \rightarrow ID = NUM$

Análise Sintática Descendente Recursiva

- *Parser* para G1:

```
void eat(int t)          int getToken()          int main()
{
    if (tok == t)        // chama o analisador léxico e obtém
        advance();       // o próximo token do programa fonte
    else ERRO();
}

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

Análise Sintática Descendente Recursiva

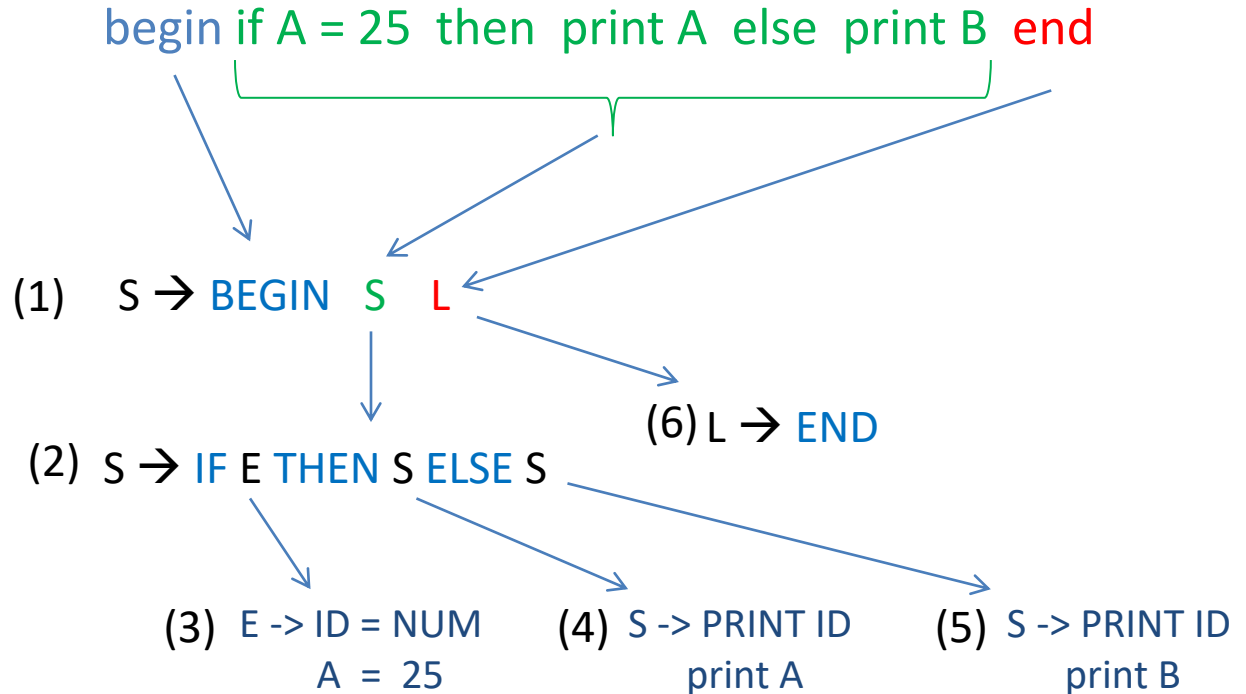
- Vamos usar a G1 para fazer a análise sintática do seguinte programa:

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

Análise Sintática Descendente Recursiva

- Derivação descendente à esquerda:

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



Análise Sintática Descendente Recursiva

- 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* (pois as regras não começam com símbolos terminais)

Análise Sintática Descendente Recursiva

- Uma 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

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

(G2 em BNF)

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

(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 == "-"))
    {
        eat(tok);
        termo();
    }
}
```

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

Continue a implementação... para termo(), fator() e main().

Considere o uso de eat(), advance(), ERRO() e getToken(), da mesma forma que em G1.

Análise Sintática Descendente Recursiva

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

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

termo → fator {mult fator}

Análise Sintática Descendente Recursiva

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

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

fator → (exp) | NUM

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

Análise Sintática Descendente Recursiva

- Bibliografia consultada

LOUDEN, K. C. **Compiladores: princípios e práticas.**

São Paulo: Pioneira Thompson Learning, 2004

MERINO, M. **Notas de Aulas - Compiladores,**

UNIMEP, 2006.