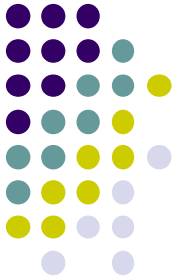
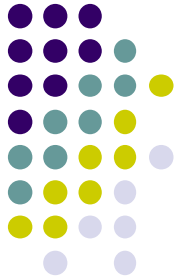


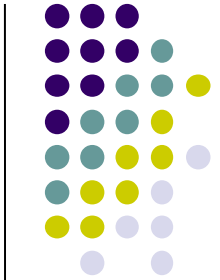
Capítulo 4
Análisis Léxico y Sintáctico

Análisis Sintáctico



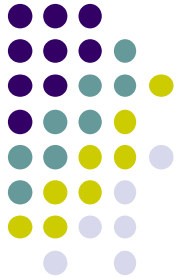
- Los analizadores sintácticos construyen árboles de análisis (parse trees) para los programas de datos.
  - En algunos casos, el parse tree es construido implícitamente (apenas el resultado de recorrer el árbol es generado);
- Objetivos del Análisis Sintáctico:
  - Encontrar errores de sintaxis; para cada uno, producir un mensaje de diagnóstico y recuperarse;
  - Producir árboles de análisis sintáctico completo, o al menos recorrer la estructura del árbol, para una entrada sintácticamente correcta (parse trees).

Análisis Sintáctico



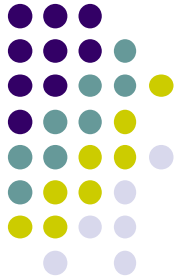
- Los analizadores sintácticos son clasificados de acuerdo con la dirección en la cual ellos construyen el árbol de análisis:
  - Arriba-Abajo (top-down): el árbol es construido de la raíz hacia las hojas;
  - Abajo-Arriba (bottom-up): el árbol es construido de las hojas hacia la raíz.
- Todos los algoritmos de análisis comúnmente utilizados operan bajo la obligación de que ellos nunca ven al frente más que un símbolo en el programa de entrada.

Analizadores Arriba-Abajo



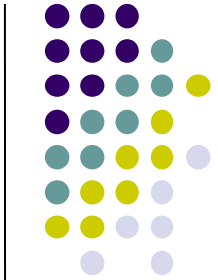
- Realizan una derivación **más a la izquierda** (leftmost derivations).
- Dada una forma sentencial, que es parte de una derivación más a la izquierda, la tarea del analizador es encontrar la siguiente forma sentencial en esa derivación.
- Un **analizador descendente recursivo** es una versión codificada de un analizador sintáctico basado directamente en la descripción BNF del lenguaje.
  - Alternativa para la recursión: tabla de análisis para implementar las reglas de la BNF.

Analizadores Abajo-Arriba



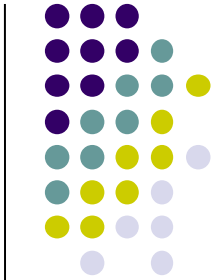
- Realizan una derivación **más a la derecha** (*rightmost* derivations).
- Dada una forma sentencial a la derecha α, el analizador debe determinar cual sub-cadena de α es el lado derecho de alguna regla en la gramática que debe ser reducida para producir la forma sentencial dada en la derivación más a la derecha.

Análisis Descendente Recursiva



- Un analizador descendente recursivo consiste de una colección de funciones (muchas de las cuales son recursivas).
- La implementación de estas funciones describe naturalmente las reglas gramaticales de BNF.
- El analizador descendente recursivo tiene una función para cada no-terminal de la gramática.

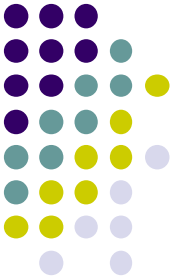
Análisis Descendente Recursiva (Implementación)



Gramática:

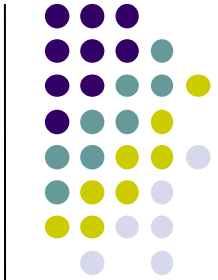
```
<expr> -> <term> + <term>
        | <term> - <term>
        | <term>
<term> -> <factor> * <factor>
        | <factor> / <factor>
        | <factor>
<factor> -> ident
          | numero
          | ( <expr> )
```

Análisis Descendente Recursiva (Implementación)



```
...
/*
<expr> -> <term> + <term>
        | <term> - <term>
*/
int expr(FILE *code_file, int next_token, NextChar *next_char)
{
 printf("Enter <expr>\n");
 next_token = term(code_file, next_token, next_char);
 while (next_token == ADD_OP || next_token == SUB_OP)
 {
       next_token = lex(code_file, next_char);
       next_token = term(code_file, next_token, next_char);
 }
 printf("Exit <expr>\n");
 return next_token;
 }
 ...
```

Análisis Descendente Recursiva (Implementación)



Entrada:

```
(suma + 47) / total
```

Análisis Descendente Recursiva (Implementación)

- **Entrada:** `(suma + 47) / total`
- **Salida:**

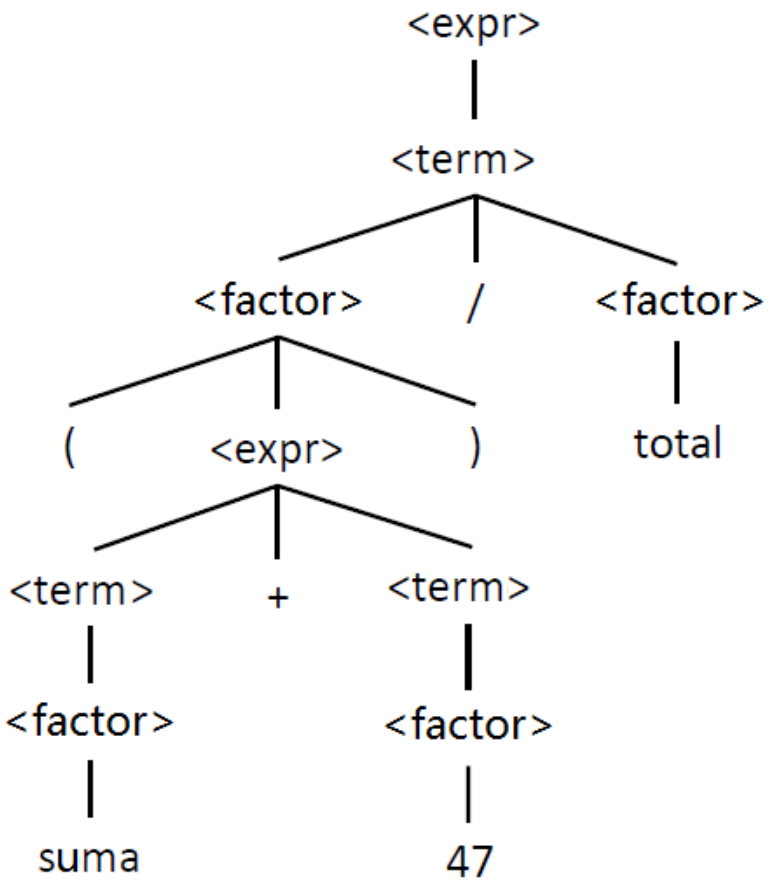
```
Token: 25, Lexema: (
Enter <expr>
Enter <term>
Enter <factor>
Token: 11, Lexema: suma
Enter <expr>
Enter <term>
Enter <factor>
Token: 21, Lexema: +
Exit <factor>
Exit <term>
Token: 10, Lexema: 47
Enter <term>
Enter <factor>
...
```

```
...
Token: 26, Lexema: )
Exit <factor>
Exit <term>
Exit <expr>
Token: 24, Lexema: /
Exit <factor>
Token: 11, Lexema: total
Enter <factor>
Token: -1, Lexema: EOF
Exit <factor>
Exit <term>
Exit <expr>
```

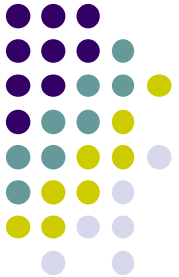
Análisis Descendente Recursiva (Implementación)

- **Salida:**

```
•Token: 25, Lexema: (
•Enter <expr>: 25
•Enter <term>: 25
•Enter <factor>: 25
•Token: 11, Lexema: suma
•Enter <expr>: 11
•Enter <term>: 11
•Enter <factor>: 11
•Token: 21, Lexema: +
•Exit <factor>: 11
•Exit <term>: 21
•Token: 10, Lexema: 47
•Enter <term>: 10
•Enter <factor>: 10
•Token: 26, Lexema: )
•Exit <factor>: 10
•Exit <term>: 26
•Exit <expr>: 11
•Token: 24, Lexema: /
•Exit <factor>: 25
•Token: 11, Lexema: total
•Enter <factor>: 11
•Token: -1, Lexema: EOF
•Exit <factor>: 11
•Exit <term>: -1
•Exit <expr>: 25
```



Recursive-Descent Parsing (cont.)



- The LL Grammar Class
  - The Left Recursion Problem
    - If a grammar has left recursion, either direct or indirect, it cannot be the basis for a top-down parser
      - A grammar can be modified to remove left recursion

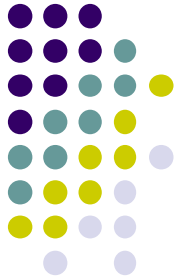
      For each nonterminal, A,

      1. Group the A-rules as $A \rightarrow A\alpha_1 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$ where none of the β's begins with A
      2. Replace the original A-rules with

         $A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$

         $A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon$

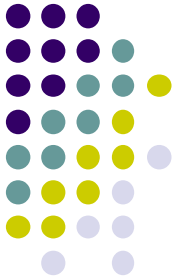
Recursive-Descent Parsing (cont.)



- The other characteristic of grammars that disallows top-down parsing is the lack of pairwise disjointness
  - The inability to determine the correct RHS on the basis of one token of lookahead
  - Def: FIRST(α) = {a | α =>* aβ }

    (If α =>* ε, ε is in FIRST(α))

Recursive-Descent Parsing (cont.)



- Pairwise Disjointness Test:
  - For each nonterminal, A, in the grammar that has more than one RHS, for each pair of rules, $A \rightarrow \alpha_i$ and $A \rightarrow \alpha_j$, it must be true that

    $FIRST(\alpha_i) \cap FIRST(\alpha_j) = \phi$
- Examples:

  $A \rightarrow a \mid bB \mid cAb$

  $A \rightarrow a \mid aB$