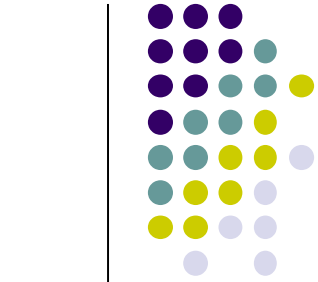
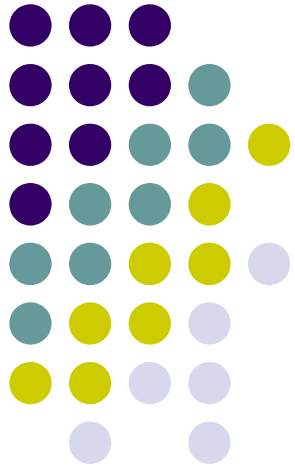
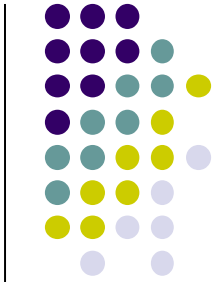
Describiendo Sintaxis y Semántica



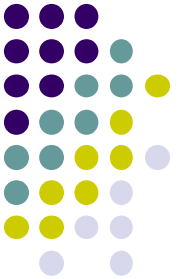


Sintaxis y Semántica



- La descripción de un lenguaje de programación implica dos aspectos principales:
  - **Sintaxis**: Describe la forma o estructura de expresiones, comandos y unidades de programa. Se compone de un conjunto de reglas que determinan qué construcciones son correctas.
  - **Semántica**: Describe el significado de las expresiones, los comandos y las unidades de programa. Define cómo las construcciones del lenguaje deben ser interpretadas y ejecutadas.
- Ejemplo – IF en lenguaje C
  - Sintaxis: `if (<expresión>) <instrucción>`
  - Semántica: si el valor actual de la expresión fuera verdadero, la instrucción incorporada será seleccionada para ejecución.

Sintaxis y Semántica - Ejemplo

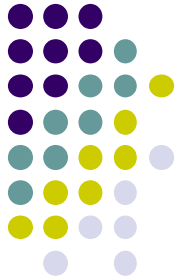


- Analice el siguiente código en lenguaje C y verifique si existen errores:

```
int j=0, cuenta, V[10];
float i@; cuenta = '0'
for (j=0, j<10; j++
{
   V[j] = cuenta++;
}
```

- El compilador tiene la responsabilidad de reportar errores!. Es necesario algún recurso para identificarlos.

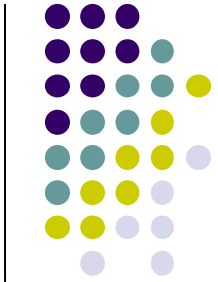
Describiendo la sintaxis



- Lenguajes, sean naturales o artificiales, son conjuntos de secuencias de caracteres de algún alfabeto, donde:
  - Una **sentencia** es una secuencia de caracteres sobre un alfabeto;
  - Un **lenguaje** es un conjunto de sentencias;
  - Un **lexema** es la unidad sintáctica de menor nivel en un lenguaje (ejemplo: *, sum, begin);
  - Un **token** es una categoría de lexemas (ejemplo: identificador, números, caracteres, etc.);

- Un programa puede ser visto como una **secuencia de lexemas**.

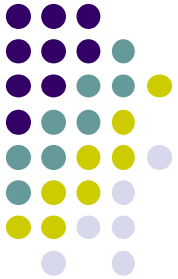
Describiendo la sintaxis



- Ejemplo:

  index = 2 * count + 17;

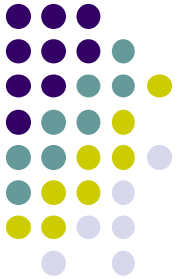
Describiendo la sintaxis



- Ejemplo:
  index = 2 * count + 17;

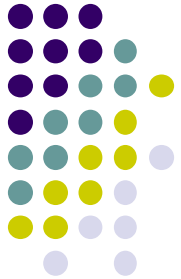
Lexemas	Tokens
index	identificador
=	signo_asignación
2	int_literal
*	mult_op
cont	identificador
+	suma_op
17	int_literal
;	punto_y_coma

Describiendo la sintaxis



- Los lenguajes se pueden definir formalmente de dos maneras:
- **Reconocedores**:
  - Un dispositivo de reconocimiento lee una cadena de entrada de un lenguaje y decide si ésta pertenece o no al lenguaje;
  - Ejemplo: Analizador sintáctico de un compilador.
- **Generadores**:
  - Dispositivo que genera una sentencia del lenguaje cada vez que se acciona;
  - Se puede determinar si la sintaxis de una determinada secuencia esta correcta comparándola con la estructura de un generador.

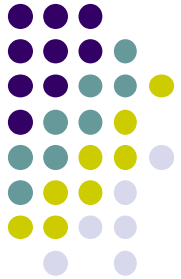
Métodos formales para describir sintaxis



- **Sintaxis** → definida formalmente a través de una gramática.
- **Gramática** → conjunto de definiciones que especifican una secuencia válida de caracteres.

- Dos clases de gramáticas son útiles en la definición formal de las gramáticas:
  - Gramáticas libres de contexto;
  - Gramáticas regulares.

Métodos formales para describir sintaxis

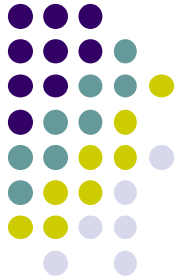


- **Forma de Backus-Naur (1959) – BNF**
  - Inventada por John Backus para describir el Algol 58; modificada por Peter Naur para describir el Algol 60. Se considera una gramática libre de contexto.

- La BNF es la forma más popular de describir concisamente la sintaxis de un lenguaje.

- Aunque simple es capaz de describir la gran mayoría de las sintaxis de los lenguajes de programación.

Forma de Backus-Naur (BNF)



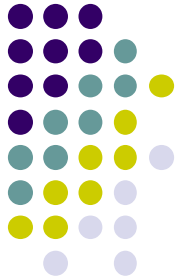
- En la BNF, las abstracciones se utilizan para representar clases de estructuras sintácticas.

```
<asignación> -> <var> = <expresión>
```

(LHS) (RHS)

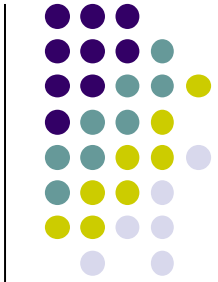
- Una abstracción se define a través de una regla o una producción. Esta está formada por:
  - **Lado izquierdo (LHS)** - abstracción que se va a definir (símbolo no terminal).
  - **Lado derecho (RHS)** - definición de la abstracción, compuesta por símbolos, lexemas y referencias a otras abstracciones.
- Símbolos y lexemas son denominados símbolos terminales.

Forma de Backus-Naur (BNF)



- <> indica un **no terminal** (término que necesita ser expandido);
- Símbolos no cercados por <> son **terminales**;
  - Estos son representativos por si. Ejemplo: if, while, (, =
- El símbolo → significa **es definido como;**
- El símbolo | significa **or** y es usado utiliza para separar alternativas.

Forma de Backus-Naur (BNF)



- Una descripción BNF o Gramática de un lenguaje es definida por un **conjunto de reglas**.

- Símbolos no terminales pueden tener **más de una definición** distinta, representando dos o más formas sintácticas posibles en el lenguaje.

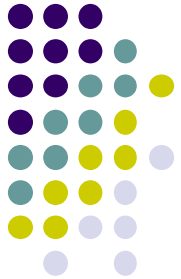
  - Regras diferentes:

```
<inst_if> → if (<expr_logica>) <inst>
<inst_if> → if (<expr_logica>) <inst> else <inst>
```

  - Misma Regla

```
<inst_if> → if (<expr_logica>) <inst>
                    | (<expr_logica>) <inst> else <inst>
```

Forma de Backus-Naur (BNF)



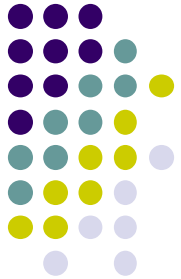
- Una regla es recursiva si el LHS aparece en el RHS.

- Ejemplo: definición de listas de identificadores:

  <ident_lista> → identificador
  | identificador, <ident_lista>

- <ident_lista> es definido como un único símbolo o un símbolo seguido de coma y de otra instancia de <ident_lista>.

Gramáticas y Derivaciones

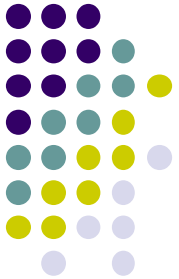


- BNF es un dispositivo generativo para definir lenguajes.

- Las sentencias del lenguaje son generadas a través de secuencias de aplicación de reglas, iniciándose por el símbolo no terminal de la gramática llamado **símbolo de inicio**.

- Una generación de sentencia se denomina **derivación**.

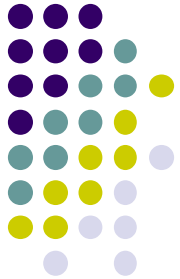
Gramáticas y Derivaciones



- Ejemplo de gramática de un lenguaje simple:

```
<programa> -> begin <lista_inst> end
<lista_inst> -> <inst>; <lista_inst>
              | <inst>
<inst> -> <var> = <expresion>
<var> -> A | B | C
<expresion> -> <var> + <var>
             | <var> - <var>
             | <var>
```

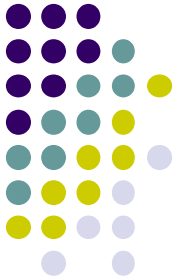
Gramáticas y Derivaciones



- Ejemplo de derivación de un programa en el lenguaje especificado:

```
<programa> => begin <lista_inst> end
           => begin <inst> ; <lista_inst> end
           => begin <var> = <expresion> ; <lista_inst> end
           => begin A = <expresion> ; <lista_inst> end
           => begin A = <var> + <var> ; <lista_inst> end
           => begin A = B + <var> ; <lista_inst> end
           => begin A = B + C ; <lista_inst> end
           => begin A = B + C ; <inst> end
           => begin A = B + C ; <var> = <expresion> end
           => begin A = B + C ; B = <expresion> end
           => begin A = B + C ; B = <var> end
           => begin A = B + C ; B = C end
```

Gramáticas y Derivaciones

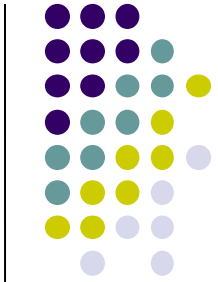


- Cada una de las cadenas de derivación, incluyendo <programa>, se llama de **forma sentencial**.

- En el ejemplo anterior, el no terminal substituído es siempre el de la extrema izquierda. Las derivaciones que usan este orden se denominan **derivaciones a la extrema izquierda** (leftmost derivations).
  - Es posible realizar la derivación a la extrema derecha;

- La derivación prosigue hasta que la forma sentencial no contenga ningún no terminal.

Gramáticas y Derivaciones



- Un ejemplo más de gramática

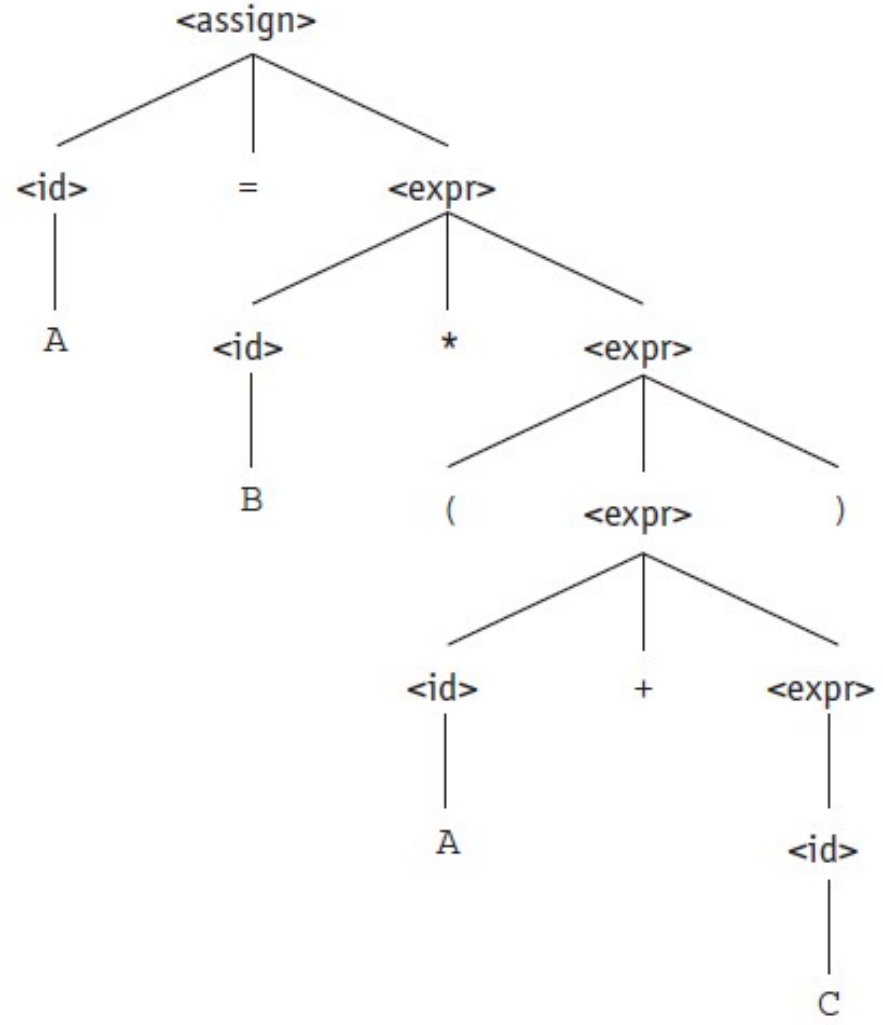
```
<asignación> → <id> = <expr>
<id> → A | B | C
<expr> → <id> + <expr>
              | <id> * <expr>
              | ( <expr> )
              | <id>
```

- Derivando: A=B*(A+C)

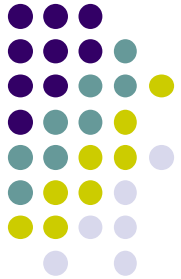
```
<asignación> => <id> = <expr>
             => A = <expr>
             => A = <id> * <expr>
             => A = B * <expr>
             => A = B * ( <expr> )
             => A = B * ( <id> + <expr> )
             => A = B * ( A + <expr> )
             => A = B * ( A + <id> )
             => A = B * ( A + C )
```

Árbol de análisis sintáctico

- Representación jerárquica
- de una derivación



Ambigüedad en gramáticas



- Una gramática es ambigua si genera una forma sentencial con dos o más árboles de análisis sintáctico distintos.
  - Si la gramática genera una sentencia con más de una derivación a la izquierda.
  - Si la gramática genera una sentencia con más de una derivación a la derecha.
- Cada derivación con una gramática no ambigua tiene un único árbol de análisis sintáctico, a pesar de ella poder ser representada por derivaciones diferentes (izquierda o derecha).

<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <expr>
| <expr> * <expr>
| (<expr>)
| <id>