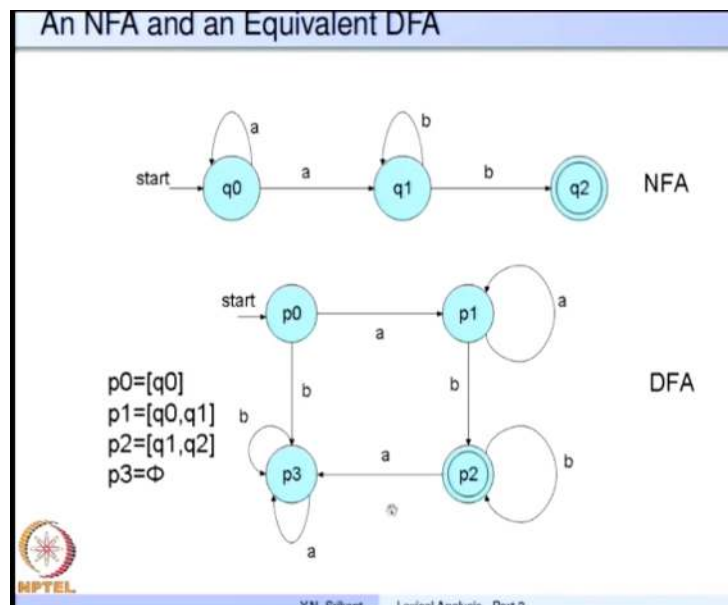


# Actividades Semana 5

## Mod - 02 Lec 03 - Parte 2

### Ejemplo 1:

- ❖ El estado inicial del DFA corresponde al conjunto  $\{q_0\}$  y estará representado por  $[q_0]$ .
- ❖ A partir de  $\delta([q_0], a)$ , los nuevos estados del DFA se construyen bajo demanda.
- ❖ Cada subconjunto de estados NFA es un posible estado DFA.
- ❖ Todos los estados de la DFA que contienen algún estado final como miembro serían estados finales de la DFA.

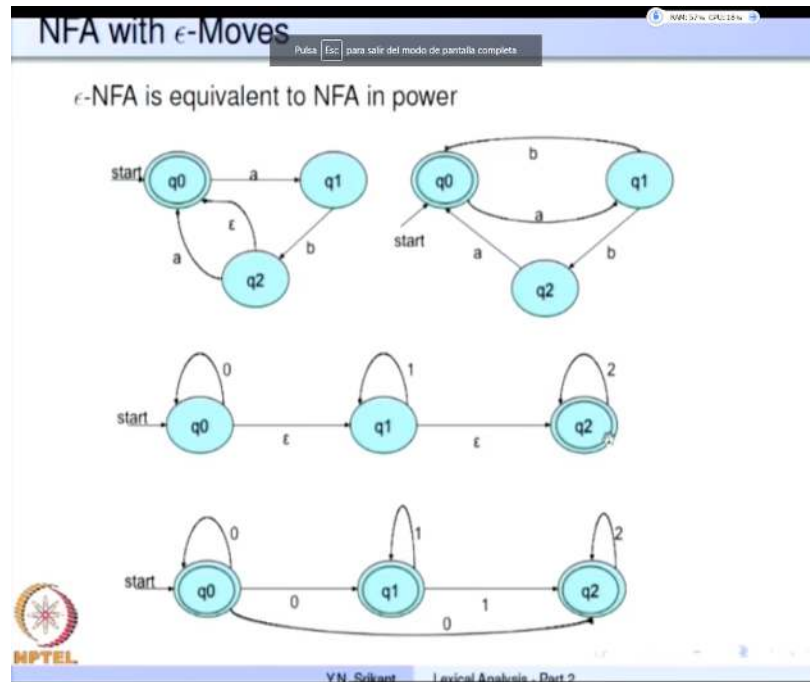


## Expresiones regulares:

Sea  $\Sigma$  un alfabeto. Los REs  $\Sigma$  y los idiomas que denotan (o generan) se definen a continuación.

- $\phi$  es un RE.  $L(\phi) = \phi$
- $\epsilon$  es un RE.  $L(\epsilon) = \epsilon$
- Para cada  $a \in \Sigma$ ,  $a$  es un RE.  $L(a) = \{a\}$
- Si  $r$  y  $s$  son REs que denotan los lenguajes  $R$  y  $S$ , respectivamente.

- $(rs)$  es un RE,  $L(rs) = R.S = \{xy \mid x \in R \wedge y \in S\}$
- $(r + s)$  es un RE,  $L(r + s) = R \cup S$
- $(L^*)$  se llama cierre de Kleene o cierre de  $L$

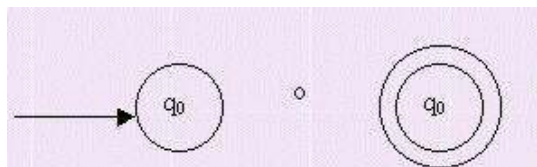


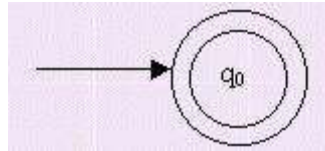
## Equivalencia entre Expresión Regular y AFD.

Teorema: dada una expresión regular existe un autómata finito que acepta el lenguaje asociado a esta expresión regular. Por tanto existe una equivalencia entre la expresión regular y el autómata finito determinístico.

Para ello tenemos como base los AFD asociados a las expresiones regulares siguientes:

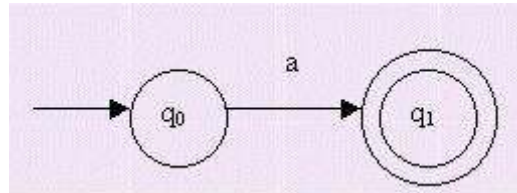
f su autómata es el que no tiene ningún estado, es decir,



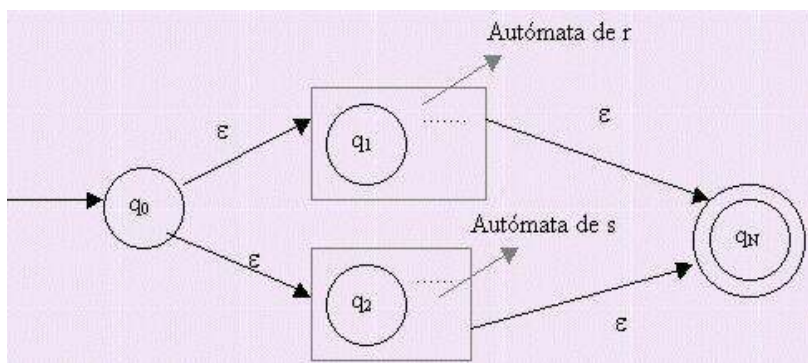


e su autómata asociado es

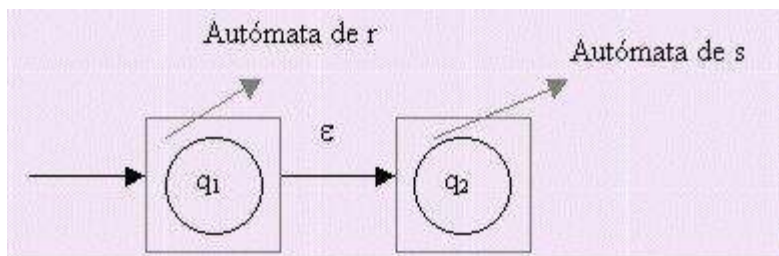
Si  $a \in A$  es una expresión regular su autómata asociado es



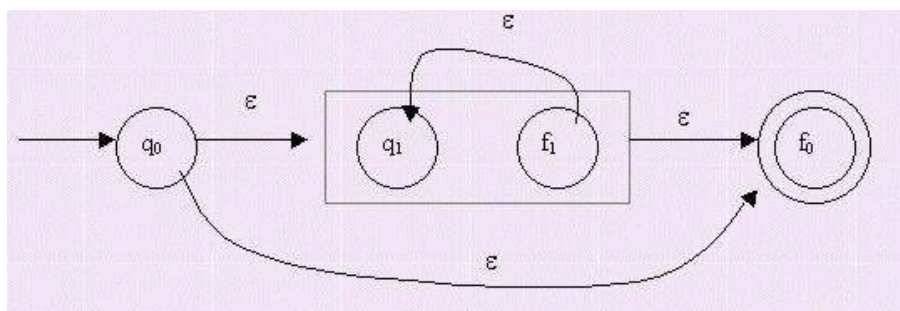
El autómata asociado a la expresión  $(r+s)$  es



El autómata asociado a la expresión  $(rs)$  es




El autómata asociado a la expresión  $r^*$  es



Conociendo estos autómatas podemos pasar a ver algunos ejemplos para construir AFD que acepte un lenguaje correspondiente a la expresión regular dada.

### Examples of Regular Expressions

- 1  $L = \text{set of all strings of 0's and 1's}$   
 $r = (0 + 1)^*$ 
  - How to generate the string 101 ?
  - $(0 + 1)^* \Rightarrow^4 (0 + 1)(0 + 1)(0 + 1)\epsilon \Rightarrow^4 101$
- 2  $L = \text{set of all strings of 0's and 1's, with at least two consecutive 0's}$   
 $r = (0 + 1)^*00(0 + 1)^*$
- 3  $L = \{w \in \{0, 1\}^* \mid w \text{ has two or three occurrences of 1, the first and second of which are not consecutive}\}$   
 $r = 0^*10^*010^*(10^* + \epsilon)$
- 4  $r = (1 + 10)^*$   
 $L = \text{set of all strings of 0's and 1's, beginning with 1 and not having two consecutive 0's}$
- 5  $r = (0 + 1)^*011$   
 $L = \text{set of all strings of 0's and 1's ending in 011}$

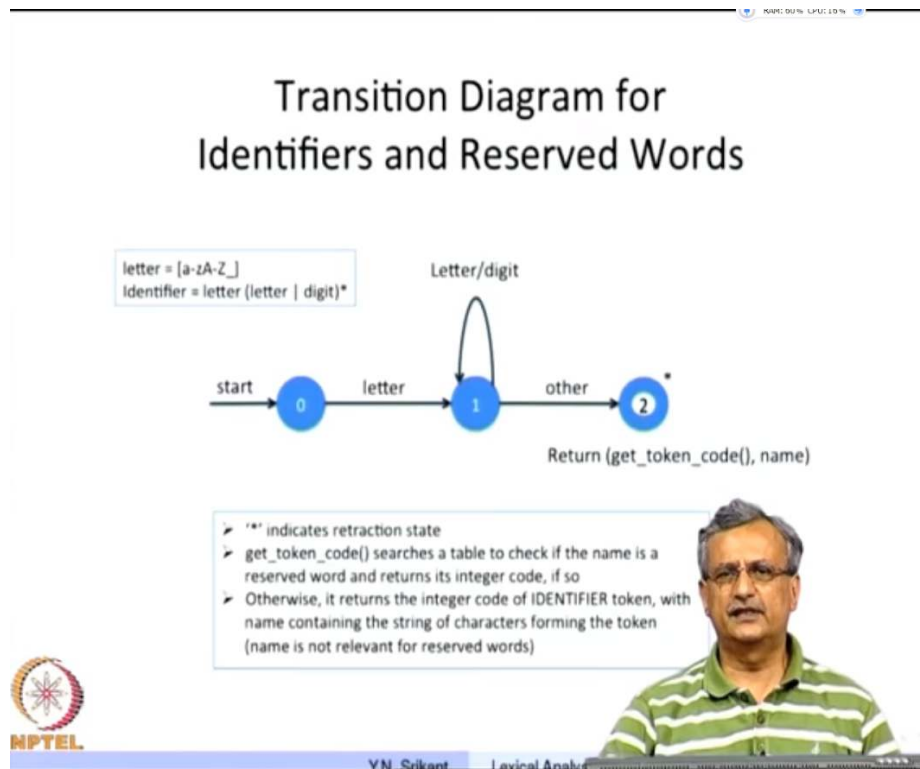


Y.N. Srikant Lexical Analysis - Part 2

## Diagramas de transición

Los diagramas de transición son DFA generalizados con las siguientes diferencias:

- ★ Los bordes pueden estar etiquetados con un símbolo, un conjunto de símbolos o una definición regular.
- ★ Algunos estados de aceptación pueden indicarse como estados de retracción, lo que indica que el lexema no incluye el símbolo que nos llevó al estado de aceptación.
- ★ Cada estado de aceptación tiene una acción adjunta, que se ejecuta cuando se alcanza ese estado. Normalmente, esta acción devuelve un token y su valor de atributo.
- ★ Los diagramas de transición no están destinados a la traducción automática, sino sólo a la traducción manual.




Un PDA  $M$  es un sistema  $(Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$ , dónde:

- ❖  $Q$  es un conjunto finito de estados
- ❖  $\Sigma$  es el alfabeto de entrada
- ❖  $\Gamma$  es el alfabeto de la pila
- ❖  $q_0 \in Q$  es el estado de inicio
- ❖  $z_0 \in \Gamma$  es el símbolo de inicio en la pila (inicialización)
- ❖  $F \subseteq Q$  es el conjunto de estados finales
- ❖  $\delta$  es la función de transición,  $Q \times \Sigma \cup \{\epsilon\} \times \Gamma$  a finitos subconjuntos de  $Q \times \Gamma^*$
- ❖ El símbolo más a la izquierda de  $\gamma_i$  será la nueva parte superior de la pila
- ❖ A en la función anterior  $\delta$  podría ser  $\epsilon$ , en cuyo caso, el símbolo de entrada no se utiliza y el cabezal de entrada no avanza.
- ❖ Para un PDA  $M$ , definimos  $L(M)$ , el lenguaje aceptado por  $M$  **por estado final**, como  $L(M) = \{w \mid (q_0, w, z_0) \xrightarrow{*} (p, \epsilon, \gamma), \text{ para algunos } p \in F \text{ y } \gamma \in \Gamma^*\}$

- ❖ Definimos  $N(M)$ , el lenguaje aceptado por  $M$  **por pila vacía**, como  $N(M) = \{w \mid (q_0, w, Z_0) \xrightarrow{*} (p, \epsilon, \epsilon), \text{ para algunos } p \in Q\}$
- ❖ Cuando la aceptación es por pila vacía, el conjunto de estados finales es irrelevante y, por lo general, establecemos  $F = \emptyset$

PDA - Examples (contd.)

- $L = \{ww^R \mid w \in \{a, b\}^+\}$   
 $M = (\{q_0, q_1, q_2\}, \{a, b\}, \{Z, a, b\}, \delta, q_0, Z, \{q_2\})$ , where  $\delta$  is defined as follows  
 $\delta(q_0, a, Z) = \{(q_0, aZ)\}$ ,  $\delta(q_0, b, Z) = \{(q_0, bZ)\}$ ,  
 $\delta(q_0, a, a) = \{(q_0, aa), (q_1, \epsilon)\}$ ,  $\delta(q_0, a, b) = \{(q_0, ab)\}$ ,  
 $\delta(q_0, b, a) = \{(q_0, ba)\}$ ,  $\delta(q_0, b, b) = \{(q_0, bb), (q_1, \epsilon)\}$ ,  
 $\delta(q_1, a, a) = \{(q_1, \epsilon)\}$ ,  $\delta(q_1, b, b) = \{(q_1, \epsilon)\}$ ,  
 $\delta(q_1, \epsilon, Z) = \{(q_2, \epsilon)\}$
- $(q_0, abba, Z) \vdash (q_0, bba, aZ) \vdash (q_0, ba, baZ) \vdash (q_1, a, aZ) \vdash (q_1, \epsilon, Z) \vdash (q_2, \epsilon, \epsilon)$
- $(q_0, aaa, Z) \vdash (q_0, aa, aZ) \vdash (q_0, a, aaZ) \vdash (q_1, \epsilon, aZ) \vdash \text{error}$
- $(q_0, aaa, Z) \vdash (q_0, aa, aZ) \vdash (q_1, a, Z) \vdash \text{error}$



## Actividades Semana 6

### Mod - 03 Lec 06 Análisis Sintáctico

#### No determinista y Determinista PDA

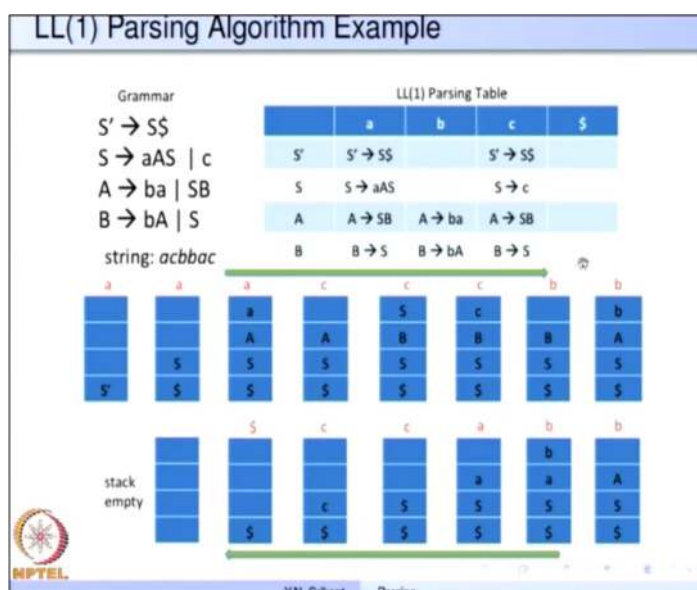
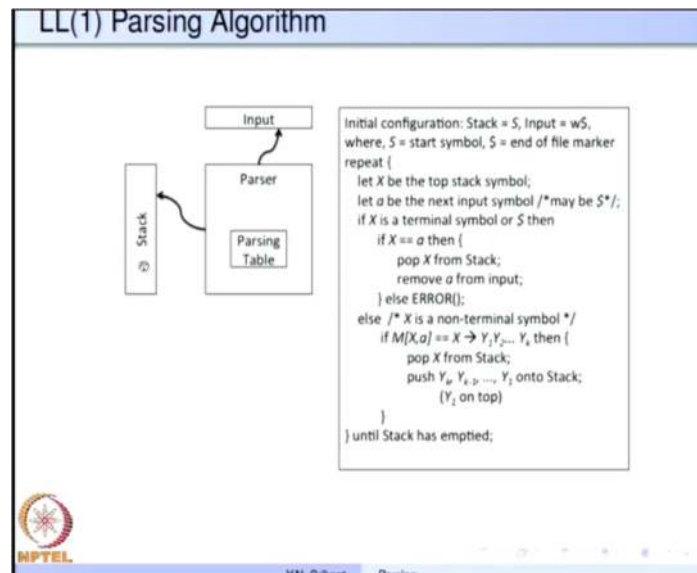
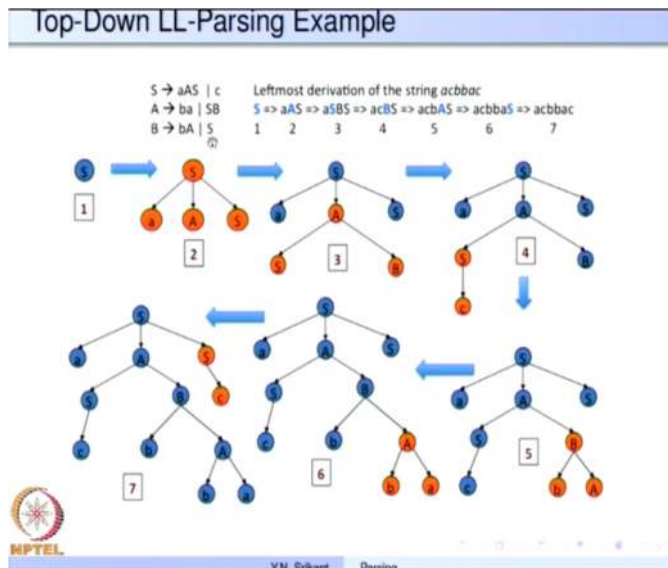
- ❖ Al igual que en el caso de NFA y DFA, PDA también tiene dos versiones: NPDA y DPDA.
- ❖ Sin embargo, NPDA es estrictamente más poderoso que el DPDA.
- ❖ Por ejemplo, el lenguaje,  $L = \{ww^R \mid w \in \{a, b\}^+\}$  solo puede ser reconocido por un NPDA y no por ningún DPDA.
- ❖ Al mismo tiempo, el lenguaje,  $L = \{wcw^R \mid w \in \{a, b\}^+\}$ , puede ser reconocido por un DPDA.

- ❖ En la práctica, necesitamos un DPDA, ya que tienen exactamente un movimiento posible en cualquier momento..
- ❖ Nuestros analizadores son todos DPDA.
- ❖ El análisis es el proceso de construir un árbol de análisis para una oración generada por una gramática determinada.
- ❖ Si no hay restricciones sobre el lenguaje y la forma de gramática utilizada, los analizadores de lenguajes sin contexto requieren  $O(n^3)$  tiempo ( $n$  siendo la longitud de la cadena analizada).
  - Algoritmo de Cocke-Younger-Kasami.
  - Algoritmo de Earley.
- ❖ Los subconjuntos de lenguajes libres de contexto normalmente requieren  $O(n)$  tiempo.
  - Análisis predictivo usando  $LL(1)$  gramática (método de análisis de arriba hacia abajo).
  - Mayús-Reducir el análisis sintáctico usando  $LR(1)$  gramática (método de análisis de abajo hacia arriba).

## **Análisis de arriba hacia abajo usando gramáticas LL**

- ❖ El análisis de arriba hacia abajo mediante el análisis predictivo, rastrea la derivación más a la izquierda de la cadena mientras se construye el árbol de análisis.
- ❖ Comienza desde el símbolo de inicio de la gramática y "predice" la siguiente producción utilizada en la derivación.
- ❖ Dicha "predicción" es ayudada por tablas de análisis (construidas fuera de línea).
- ❖ La siguiente producción que se utilizará en la derivación se determina utilizando el siguiente símbolo de entrada para buscar la tabla de análisis (símbolo de anticipación).
- ❖ Poner restricciones en la gramática asegura que ningún espacio en la tabla de análisis contenga más de una producción.
- ❖ En el momento de la construcción de la tabla de análisis, si dos producciones se vuelven elegibles para colocarse en el mismo espacio de la tabla de análisis, la gramática se declara no apta para predicciones.







## Gramáticas fuertes de LL (k)

- ❖ Sea la gramática dada  $G$ .
- ❖ La entrada se amplía con  $k$  símbolos,  $\$^k$ ,  $k$  es el avance de la gramática.
- ❖ Introducir un nuevo no terminal  $S'$ , y una producción,  $S' \rightarrow S\$^k$ , donde  $S$  es el símbolo de inicio de la gramática dada.
- ❖ Considere solo las derivaciones más a la izquierda y asuma que la gramática no tiene símbolos inútiles.
- ❖ Una producción  $A \rightarrow \alpha$  en  $G$  se llama fuerte  $LL(k)$  producción, si en  $G$ .  
 $S' \Rightarrow^* wA\gamma \Rightarrow w\alpha\gamma \Rightarrow^* wzy$   
 $S' \Rightarrow^* w'A\delta \Rightarrow w'\beta\delta \Rightarrow^* w'zy$   
 $|Z| = k, z \in \Sigma^*, w \text{ y } w' \in \Sigma^*, \text{ entonces } \alpha = \beta$
- ❖ Una gramática (no terminal) es fuerte  $LL(k)$  si todas sus producciones son fuertes  $LL(k)$ .
- ❖ Fuerte  $LL(k)$  Las gramáticas no permiten que se utilicen diferentes producciones del mismo no terminal incluso en dos derivaciones diferentes, si los primeros  $k$  símbolos de las cadenas producidas por  $\alpha\gamma$  y  $\beta\delta$  son las mismas.

- Example:  $S \rightarrow Abc|aAcb$ ,  $A \rightarrow \epsilon|b|c$   
 $S$  is a strong  $LL(1)$  nonterminal
  - $S' \Rightarrow S\$ \Rightarrow Abc\$ \Rightarrow bc\$$ ,  $bbc\$$ , and  $cbc\$$ , on application of the productions,  $A \rightarrow \epsilon$ ,  $A \rightarrow b$ , and,  $A \rightarrow c$ , respectively.  $z = b$ ,  $b$ , or  $c$ , respectively
  - $S' \Rightarrow S\$ \Rightarrow aAcb\$ \Rightarrow acb\$$ ,  $abcb\$$ , and  $accb\$$ , on application of the productions,  $A \rightarrow \epsilon$ ,  $A \rightarrow b$ , and,  $A \rightarrow c$ , respectively.  $z = a$ , in all three cases
  - In this case,  $w = w' = \epsilon$ ,  $\alpha = Abc$ ,  $\beta = aAcb$ , but  $z$  is different in the two derivations, in all the derived strings
  - Hence the nonterminal  $S$  is strong  $LL(1)$

$A$  is not strong  $LL(1)$

$S' \Rightarrow^* Abc\$ \Rightarrow \underline{b}c\$$ ,  $w = \epsilon$ ,  $z = b$ ,  $\alpha = \epsilon$  ( $A \rightarrow \epsilon$ )

$S' \Rightarrow^* Acb\$ \Rightarrow \underline{b}c\$$ ,  $w' = \epsilon$ ,  $z = b$ ,  $\beta = b$  ( $A \rightarrow b$ )

Aunque los lookaheads son los mismos ( $z = b$ ),  $\alpha \neq \beta$ , y por tanto, la gramática no es fuerte  $LL(1)$ .

A is not strong  $LL(2)$

$S' \Rightarrow^* Abc\$ \Rightarrow \underline{bc}\$, w = \epsilon, z = bc, \alpha = \epsilon (A \rightarrow \epsilon)$



$S' \Rightarrow^* aAc\$ \Rightarrow a\underline{bc}b\$, w' = a, z = bc, \beta = b (A \rightarrow b)$

Aunque los lookaheads son los mismos ( $z = bc$ ),  $\alpha \neq \beta$ , y por tanto, la gramática no es fuerte  $LL(2)$ .

## Mod - 03 Lec 07 Análisis Sintáctico

**FIRST and FOLLOW Computation Example**


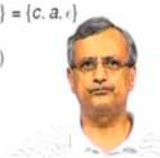
- Consider the following grammar  
 $S' \rightarrow S\$$ ,  $S \rightarrow aAS \mid c$ ,  $A \rightarrow ba \mid SB$ ,  $B \rightarrow bA \mid S$
- $FIRST(S') = FIRST(S) = \{a, c\}$  because  
 $S' \Rightarrow S\$ \Rightarrow c\$$ , and  $S' \Rightarrow S\$ \Rightarrow aAS\$ \Rightarrow abaS\$ \Rightarrow abac\$$
- $FIRST(A) = \{a, b, c\}$  because  
 $A \Rightarrow ba$ , and  $A \Rightarrow SB$ , and therefore all symbols in  $FIRST(S)$  are in  $FIRST(A)$
- $FOLLOW(S) = \{a, b, c, \$\}$  because  
 $S' \Rightarrow S\$$ ,  
 $S' \Rightarrow^* aAS\$ \Rightarrow aSB\$ \Rightarrow aSbAS\$$ ,  
 $S' \Rightarrow^* aSBS\$ \Rightarrow aSS\$ \Rightarrow aSaASS\$$ ,  
 $S' \Rightarrow^* aSS\$ \Rightarrow aSc\$$
- $FOLLOW(A) = \{a, c\}$  because  
 $S' \Rightarrow^* aAS\$ \Rightarrow aAaAS\$$ ,  
 $S' \Rightarrow^* aAS\$ \Rightarrow aAc\$$

VN Subramanian Director

**FIRST Computation: Algorithm Trace - 1**

- Consider the following grammar  
 $S' \rightarrow S\$$ ,  $S \rightarrow aAS \mid \epsilon$ ,  $A \rightarrow ba \mid SB$ ,  $B \rightarrow cA \mid S$
- Initially,  $FIRST(S) = FIRST(A) = FIRST(B) = \emptyset$
- Iteration 1
  - $FIRST(S) = \{a, \epsilon\}$  from the productions  $S \rightarrow aAS \mid \epsilon$
  - $FIRST(A) = \{b\} \cup FIRST(S) - \{\epsilon\} \cup FIRST(B) - \{\epsilon\} = \{b, a\}$   
 from the productions  $A \rightarrow ba \mid SB$   
 (since  $\epsilon \in FIRST(S)$ ,  $FIRST(B)$  is also included;  
 since  $FIRST(B) = \emptyset$ ,  $\epsilon$  is not included)
  - $FIRST(B) = \{c\} \cup FIRST(S) - \{\epsilon\} \cup \{\epsilon\} = \{c, a, \epsilon\}$   
 from the productions  $B \rightarrow cA \mid S$   
 ( $\epsilon$  is included because  $\epsilon \in FIRST(S)$ )

VN Subramanian Director

**FIRST Computation: Algorithm Trace - 2**

- The grammar is  
 $S' \rightarrow S\$$ ,  $S \rightarrow aAS \mid \epsilon$ ,  $A \rightarrow ba \mid SB$ ,  $B \rightarrow cA \mid S$
- From the first iteration,  
 $FIRST(S) = \{a, \epsilon\}$ ,  $FIRST(A) = \{b, a\}$ ,  $FIRST(B) = \{c, a, \epsilon\}$
- Iteration 2  
 (values stabilize and do not change in iteration 3)
  - $FIRST(S) = \{a, \epsilon\}$  (no change from iteration 1)
  - $FIRST(A) = \{b\} \cup FIRST(S) - \{\epsilon\} \cup FIRST(B) - \{\epsilon\}$   
 $= \{b, a, c, \epsilon\}$  (changed!)
  - $FIRST(B) = \{c, a, \epsilon\}$  (no change from iteration 1)

## Condiciones LL (1)

- ❖ Sea  $G$  una gramática libre de contexto
- ❖  $G$  es LL (1) si para cada par de producciones  $A \rightarrow \alpha$  y  $A \rightarrow \beta$ , la siguiente condición se cumple.
  - $\text{dirsymb}(\alpha) \cap \text{dirsymb}(\beta) = \emptyset$ , where  
 $\text{dirsymb}(\gamma) = \text{if } (\epsilon \in \text{first}(\gamma)) \text{ then } ((\text{first}(\gamma) - \{\epsilon\}) \cup \text{follow}(A)) \text{ else first}(\gamma)$  ( $\gamma$  stands for  $\alpha$  or  $\beta$ )
  - $\text{dirsymb}$  significa "conjunto de símbolos de dirección"
- ❖ Una formulación equivalente (como en el libro de ALSU) es como abajo
  - $\text{first}(\alpha, \text{follow}(A)) \cap \text{first}(\beta, \text{follow}(A)) = \emptyset$
- ❖ Construcción de la tabla de análisis sintáctico LL (1)
  - para cada producción  $A \rightarrow \alpha$
  - para cada símbolo  $s \in \text{dirsymb}(\alpha)$
  - /\*  $s$  puede ser un símbolo de terminal o  $\$$  \*/
  - add  $A \rightarrow \alpha$  to  $LLPT[A, s]$
  - Hacer que cada entrada indefinida de  $LLPT$  sea un error
  - para cada producción  $A \rightarrow \alpha$
  - para cada símbolo de terminal  $a \in \text{first}(\alpha)$
  - add  $A \rightarrow \alpha$  to  $LLPT[A, a]$
  - if  $\epsilon \in \text{first}(\alpha)$ {
  - para cada símbolo de terminal  $b \in \text{follow}(A)$
  - add  $A \rightarrow \alpha$  to  $LLPT[A, b]$
  - if  $\$ \in \text{follow}(A)$
  - add  $A \rightarrow \alpha$  to  $LLPT[A, \$]$
  - }
  - Hacer que cada entrada indefinida de  $LLPT$  sea un error
- ❖ Una vez completada la construcción de la tabla LL (1) (siguiendo cualquiera de los dos métodos), si algún espacio en la tabla LL (1) tiene dos o más producciones, entonces la gramática NO es LL (1)

Ejemplos simples de gramática LL (1):

- P1:  $S \rightarrow \text{if } (a) S \text{ else } S \mid \text{while } (a) S \mid \text{begin } SL \text{ end}$
- P2:  $SL \rightarrow S S'$
- P3:  $S' \rightarrow \epsilon$
- {if, while, begin, end, a, (, ), ., ;} are all terminal symbols
- Clearly, all alternatives of P1 start with distinct symbols and hence create no problem
- P2 has no choices
- Regarding P3,  $\text{dirsymb}(SL) = \{\epsilon\}$ , and  $\text{dirsymb}(\epsilon) = \{\text{end}\}$ , and the two have no common symbols
- Hence the grammar is LL(1)



**LL(1) Table Construction Example 1**

LL(1) Parsing Table for the original grammar

	if	id	else	a	\$
$S'$	$S' \rightarrow SS$			$S' \rightarrow SS$	
$S$	$S \rightarrow \text{if id } S$	$S \rightarrow \text{if id } S$	$S \rightarrow \text{if id } S$	$S \rightarrow a$	

Original Grammar

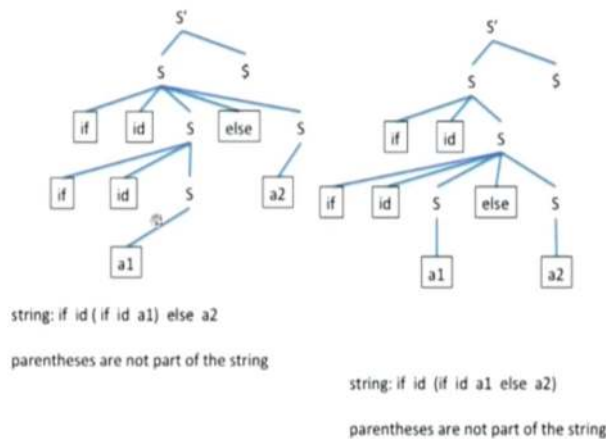
$S' \rightarrow SS$   
 $S \rightarrow \text{if id } S \mid \text{if id } S \text{ else } S \mid a$

Grammar is not LL(1)

tokens: if, id, else, a

$\text{dirsymb}(SS) = \{\text{if}, a\}$ ;  $\text{dirsymb}(a) = \{a\}$   
 $\text{dirsymb}(\text{if id } S) = \{\text{if}\}$   
 $\text{dirsymb}(\text{if id } S \text{ else } S) = \{\text{if}\}$

$\text{dirsymb}(\text{if id } S) \cap \text{dirsymb}(a) = \emptyset$   
 $\text{dirsymb}(\text{if id } S \text{ else } S) \cap \text{dirsymb}(a) = \emptyset$   
 $\text{dirsymb}(\text{if id } S) \cap \text{dirsymb}(\text{if id } S \text{ else } S) \neq \emptyset$



### LL(1) Table Construction Example 2

**Original Grammar**

$$S' \rightarrow SS$$

$$S \rightarrow \text{if id } S \mid \text{if id } S \text{ else } S \mid a$$

**LL(1) Parsing Table for modified grammar**

	if	else	a	\$
$S'$	$S' \rightarrow SS$		$S' \rightarrow SS$	
$S$	$S \rightarrow \text{if id } S \mid S1$		$S \rightarrow a$	
$S1$		$S1 \rightarrow \epsilon$ $S1 \rightarrow \text{else } S$		$S1 \rightarrow \epsilon$

**Left-Factored Grammar**

$$S' \rightarrow SS$$

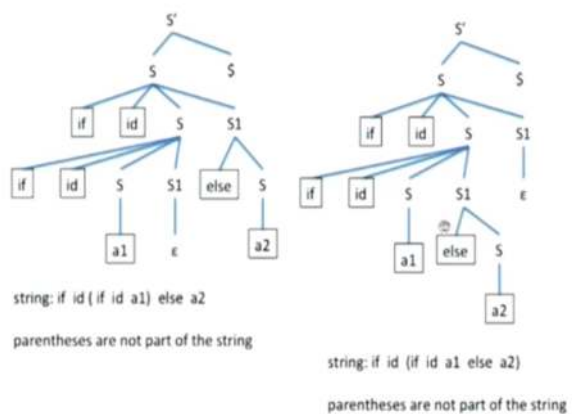
$$S \rightarrow \text{if id } S S1 \mid a$$

$$S1 \rightarrow \epsilon \mid \text{else } S$$

tokens: if, id, else, a

Grammar is not LL(1)

$\text{dirsymb}(\text{if id } S S1) \cap \text{dirsymb}(a) = \emptyset$   
 $\text{dirsymb}(\epsilon) \cap \text{dirsymb}(\text{else } S) \neq \emptyset$



### LL(1) Table Construction Example 3

**Original Grammar**

$$S' \rightarrow SS$$

$$S \rightarrow aAS \mid c$$

$$A \rightarrow ba \mid SB$$

$$B \rightarrow bA \mid S$$

Grammar is LL(1)

**LL(1) Parsing Table**

	a	b	c	\$
$S'$	$S' \rightarrow SS$		$S' \rightarrow SS$	
$S$	$S \rightarrow aAS$		$S \rightarrow c$	
$A$	$A \rightarrow SB$	$A \rightarrow ba$	$A \rightarrow SB$	
$B$	$B \rightarrow S$	$B \rightarrow ba$	$B \rightarrow S$	

$\text{first}(S) = \{a, c\}$   
 $\text{first}(A) = \{a, b, c\}$   
 $\text{first}(B) = \{a, b, c\}$

$\text{dirsymb}(aAS) \cap \text{dirsymb}(c) = \emptyset$   
 $\text{dirsymb}(ba) \cap \text{dirsymb}(SB) = \emptyset$   
 $\text{dirsymb}(bA) \cap \text{dirsymb}(S) = \emptyset$

$\text{follow}(S) = \{a, b, c, \$\}$   
 $\text{follow}(A) = \{a, c\}$   
 $\text{follow}(B) = \{a, c\}$

$\text{dirsymb}(SS) = \{a, c\}$   
 $\text{dirsymb}(aAS) = \{a\}$   
 $\text{dirsymb}(c) = \{c\}$

$\text{dirsymb}(ba) = \{b\}$   
 $\text{dirsymb}(SB) = \{a, c\}$   
 $\text{dirsymb}(bA) = \{b\}$   
 $\text{dirsymb}(S) = \{a, c\}$

## Eliminación de los símbolos inútiles

Now we study the grammar transformations, elimination of useless symbols, elimination of left recursion and left factoring

- ❖ Given a grammar  $G = (N, T, P, S)$ , a non-terminal  $X$  is useful if  $S \Rightarrow^* \alpha X \beta \Rightarrow w$ , where,  $w \in T^*$   
Otherwise,  $X$  is useless

- ❖ Two conditions have to be met to ensure that  $X$  is useful
  1.  $X \Rightarrow^* w, w \in T^*$  ( $X$  derives some terminal string)
  2.  $S \Rightarrow^* \alpha X \beta$  ( $X$  occurs in some string derivable from  $S$ )

## Mod-03 Lec-08 Análisis Sintáctico



### Análisis de Descenso Recursivo

- ❖ Estrategia de análisis de arriba hacia abajo.
- ❖ Una función / procedimiento para cada no terminal.
- ❖ Las funciones se llaman entre sí de forma recursiva, según la gramática.
- ❖ La pila de recursividad maneja las tareas de la pila del analizador LL (1).
- ❖ LL (1) condiciones que deben cumplirse para la gramática.
- ❖ Se puede generar automáticamente a partir de la gramática.
- ❖ La codificación manual también es fácil.
- ❖ La recuperación de errores es superior.

**An Example**

```
Grammar:  $S' \rightarrow S\$$ ,  $S \rightarrow aAS \mid c$ ,  $A \rightarrow ba \mid SB$ ,  $B \rightarrow bA \mid S$ 



/* function for nonterminal S' */
void main() { /* S' --> S$ */
    fS(); if (token == eof) accept();
        else error();
    }
/* function for nonterminal S */
void fS() { /* S --> aAS | c */
    switch token {
        case a : get_token(); fA(); fS();
                break;
        case c : get_token(); break;
        others : error();
    }
}
```



An Example (contd.)

```
void fA() { /* A --> ba | SB */
    switch token {
        case b : get_token();
                  if (token == a) get_token();
                  else error(); break;
        case a,c : fS(); fB(); break;
        others : error();
    }
}

void fB() { /* B --> bA | S */
    switch token {
        case b : get_token(); fA(); break;
        case a,c : fS(); break;
        others : error();
    }
}
```



YN Subant Desktop

## Generación Automática de Analizadores de RD

- ❖ El esquema se basa en la estructura de las producciones..
- ❖ La gramática debe satisfacer las condiciones LL (1).
- ❖ La función `get_token()` obtiene el siguiente token del analizador léxico y lo coloca en el token de variable global.
- ❖ La función `error ()` imprime un mensaje de error adecuado.
- ❖ En la siguiente diapositiva, para cada componente gramatical, se muestra el código que se debe generar.

Automatic Generation of RD Parsers (contd.)

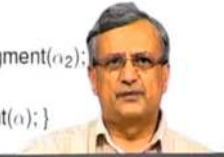

- 1.  $\epsilon$  :
- 2.  $a \in T$  : if (token == a) get\_token(); else error();
- 3.  $A \in N$  : fA(); /\* function call for nonterminal A \*/
- 4.  $\alpha_1 | \alpha_2 | \dots | \alpha_n$  :

```
switch token {
    case dirsym( $\alpha_1$ ): program_segment( $\alpha_1$ ); break;
    case dirsym( $\alpha_2$ ): program_segment( $\alpha_2$ ); break;
    ...
    others: error();
}
```

- 5.  $\alpha_1 \alpha_2 \dots \alpha_n$  :

```
program_segment( $\alpha_1$ ); program_segment( $\alpha_2$ );
program_segment( $\alpha_n$ );
```

- 6.  $A \rightarrow \alpha$  : void fA() { program\_segment( $\alpha$ ); }



## Análisis de abajo hacia arriba

- ❖ Comience en las hojas, construya el árbol de análisis en segmentos pequeños, combine los árboles pequeños para hacer árboles más grandes, hasta que se alcance la raíz.
- ❖ Este proceso se llama reducción de la oración al símbolo inicial de la gramática.
- ❖ Una de las formas de "reducir" una oración es seguir la derivación más a la derecha de la oración al revés.
  - El análisis sintáctico Shift-Reduce implementa dicha estrategia.
  - Utiliza el concepto de asa para detectar cuándo realizar reducciones.

## Análisis de Mayús-Reducir

- ❖ **Mango:** Mango de una forma de oración correcta  $\gamma$ , es una producción  $A \rightarrow \beta$  y un puesto en  $\gamma$ , donde la cadena  $\beta$  puede ser encontrado y reemplazado por A, para producir la forma oracional derecha anterior en una derivación más a la derecha de  $\gamma$ .  
Es decir, si  $S \Rightarrow_{rm}^* \alpha A w \Rightarrow_{rm} \alpha \beta w$ , entonces  $A \rightarrow \beta$  en la posición siguiente  $\alpha$  es un mango de  $\alpha \beta w$ .
- ❖ Un asa siempre aparecerá en la parte superior de la pila, nunca sumergida dentro de la pila.
- ❖ En el análisis sintáctico S-R, ubicamos el mango y lo reducimos por el LHS de la producción repetidamente, para llegar al símbolo de inicio.
- ❖ Estas reducciones, de hecho, trazan una derivación más a la derecha de la oración a la inversa. Este proceso se llama poda de mango.
- ❖ LR-Parsing es un método de análisis sintáctico con reducción de cambios.

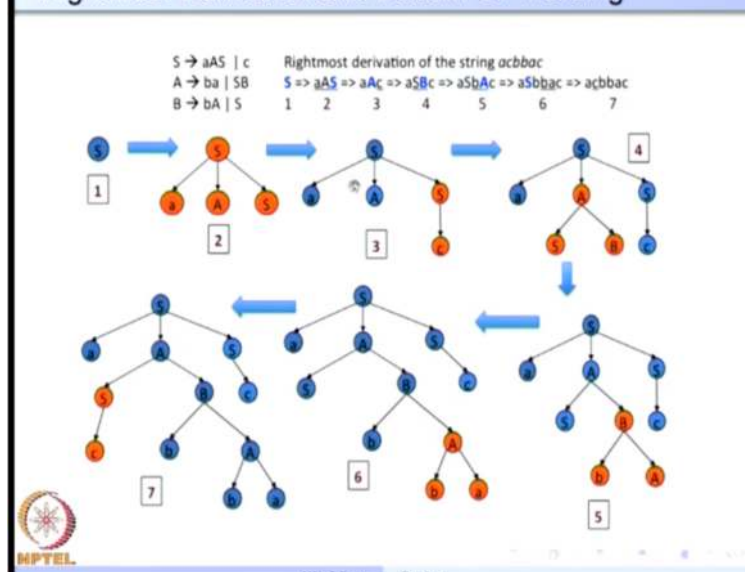


### Examples (contd.)

- $E \rightarrow E + E, E \rightarrow E * E, E \rightarrow (E), E \rightarrow id$   
For the string =  $id + id * id$ , two rightmost derivation marked with handles are shown below

$$\begin{aligned}
 E &\Rightarrow \underline{E + E} \quad (E + E, E \rightarrow E + E) \\
 &\Rightarrow \underline{E + E * E} \quad (E * E, E \rightarrow E * E) \\
 &\Rightarrow \underline{E + E * id} \quad (id, E \rightarrow id) \\
 &\Rightarrow \underline{E + id * id} \quad (id, E \rightarrow id) \\
 &\Rightarrow \underline{id + id * id} \quad (id, E \rightarrow id) \\
 E &\Rightarrow \underline{E * E} \quad (E * E, E \rightarrow E * E) \\
 &\Rightarrow \underline{E * id} \quad (id, E \rightarrow id) \\
 &\Rightarrow \underline{E + E * id} \quad (E + E, E \rightarrow E + E) \\
 &\Rightarrow \underline{E + id * id} \quad (id, E \rightarrow id) \\
 &\Rightarrow \underline{id + id * id} \quad (id, E \rightarrow id)
 \end{aligned}$$

### Rightmost Derivation and Bottom-UP Parsing



## Algoritmo de análisis Shift-Reduce

- ❖ ¿Cómo ubicamos un identificador en una forma de oración correcta?  
→ Un analizador LR usa un DFA para detectar la condición de que ahora hay un identificador en la pila.
- ❖ ¿Qué producción utilizar, en caso de que haya más de una con el mismo RHS?  
→ Un analizador LR utiliza una tabla de análisis similar a una tabla de análisis LL, para elegir la producción.
- ❖ Se usa una pila para implementar un analizador S-R, el analizador tiene cuatro acciones:
  1. **cambio:** el siguiente símbolo de entrada se desplaza a la parte superior de la pila.

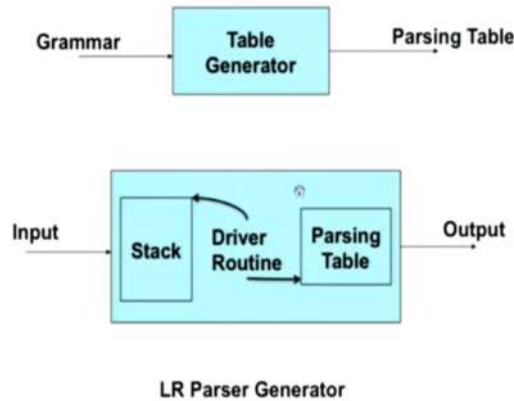
2. **reducir:** el extremo derecho del asa es la parte superior de la pila; ubica el extremo izquierdo del mango dentro de la pila y reemplaza el mango por el LHS de una producción adecuada.
3. **aceptar:** anuncia la finalización exitosa del análisis.
4. **error:** error de sintaxis, se llama a la rutina de recuperación de errores.

S-R Parsing Example 1		
§ marks the bottom of stack and the right end of the input		
Stack	Input	Action
§	acbbac§	shift
§ a	cbbac§	shift
§ ac	bbac§	reduce by $S \rightarrow c$
§ aS	bbac§	shift
§ aSb	bac§	shift
§ aSbb	ac§	shift
§ aSbba	c§	reduce by $A \rightarrow ba$
§ aSbA	c§	reduce by $B \rightarrow bA$
§ aSB	c§	reduce by $A \rightarrow SB$
§ aA	c§	shift
§ aAc	§	reduce by $S \rightarrow c$
§ aAS	§	reduce by $S \rightarrow aAS$
§ S	§	accept

S-R Parsing Example 2		
§ marks the bottom of stack and the right end of the input		
Stack	Input	Action
§	$id_1 + id_2 * id_3 §$	shift
§ $id_1$	$+ id_2 * id_3 §$	reduce by $E \rightarrow id$
§ $E$	$+ id_2 * id_3 §$	shift
§ $E +$	$id_2 * id_3 §$	shift
§ $E + id_2$	$* id_3 §$	reduce by $E \rightarrow id$
§ $E + E$	$* id_3 §$	shift
§ $E + E *$	$id_3 §$	shift
§ $E + E * id_3$	§	reduce by $E \rightarrow id$
§ $E + E * E$	§	reduce by $E \rightarrow E * E$
§ $E + E$	§	reduce by $E \rightarrow E + E$
§ $E$	§	accept

## Análisis LR

- ❖ LR (k): exploración de izquierda a derecha con derivación más a la derecha en reversa, siendo k el número de tokens de anticipación.  
→  $k = 0, 1$  son de interés práctico
- ❖ Los analizadores sintácticos LR también se generan automáticamente utilizando generadores de analizadores sintácticos.
- ❖ Las gramáticas LR son un subconjunto de CFG para los que se pueden construir analizadores LR.
- ❖ Las gramáticas LR (1) se pueden escribir con bastante facilidad para prácticamente todas las construcciones del lenguaje de programación para las que se pueden escribir CFG.
- ❖ El análisis sintáctico LR es el método de análisis sintáctico de reducción de cambios sin retroceso más general (conocido en la actualidad).
- ❖ Las gramáticas LL son un subconjunto estricto de las gramáticas LR; una gramática LL (k) también es LR (k), pero no viceversa.



## Configuración del analizador LR

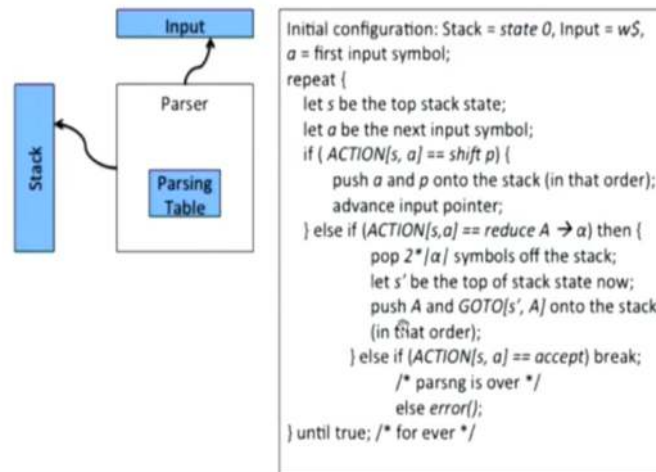
- ❖ Una configuración de un analizador LR es:

$(s_0 X_1 s_2 X_2 \dots X_m s_m, a_i a_{i+1} \dots a_n \$)$ , where,

**apilar**      **entrada inesperada**

$s_0, s_1, \dots, s_m$ , son los estados del analizador, y  $X_1, X_2, \dots, X_m$ , son símbolos gramaticales (terminales o no terminales).

- ❖ Inicio de la configuración del analizador:  $(s_0, a_1 a_2, \dots, a_n \$)$ , donde,  $s_0$  es el estado inicial del analizador, y  $a_1 a_2 \dots a_n$  es la cadena que se va a analizar.
- ❖ Dos partes en la tabla de análisis: ACCIÓN e IR
  - La tabla ACCIÓN puede tener cuatro tipos de entradas: **desplazamiento**, **reducción**, **aceptación** o **error**.
  - La tabla IR proporciona la siguiente información de estado que se utilizará después de un movimiento de reducción.



STATE	ACTION				GOTO		
	a	b	c	\$	S	A	B
0	S2		S3		1		
1				R1 acc			
2	S2	S6	S3		8	4	
3	R3	R3	R3	R3			
4	S2		S3		5		
5	R2	R2	R2	R2			
6	S7						
7	R4	R4	R4	R4			
8	S2	S10	S3		12		9
9	R5	R5	R5	R5			
10	S2	S6	S3		8	11	
11	R6	R6	R6	R6			
12	R7	R7	R7	R7			

1.  $S' \rightarrow S$   
2.  $S \rightarrow aAS$   
3.  $S \rightarrow c$   
4.  $A \rightarrow ba$   
5.  $A \rightarrow SB$   
6.  $B \rightarrow bA$   
7.  $B \rightarrow S$

1.  $S' \rightarrow S$
2.  $S \rightarrow aAS$
3.  $S \rightarrow c$
4.  $A \rightarrow ba$
5.  $A \rightarrow SB$
6.  $B \rightarrow bA$
7.  $B \rightarrow S$

# Actividades Semana 7

## Mod-03 Lec-09 Análisis Sintáctico

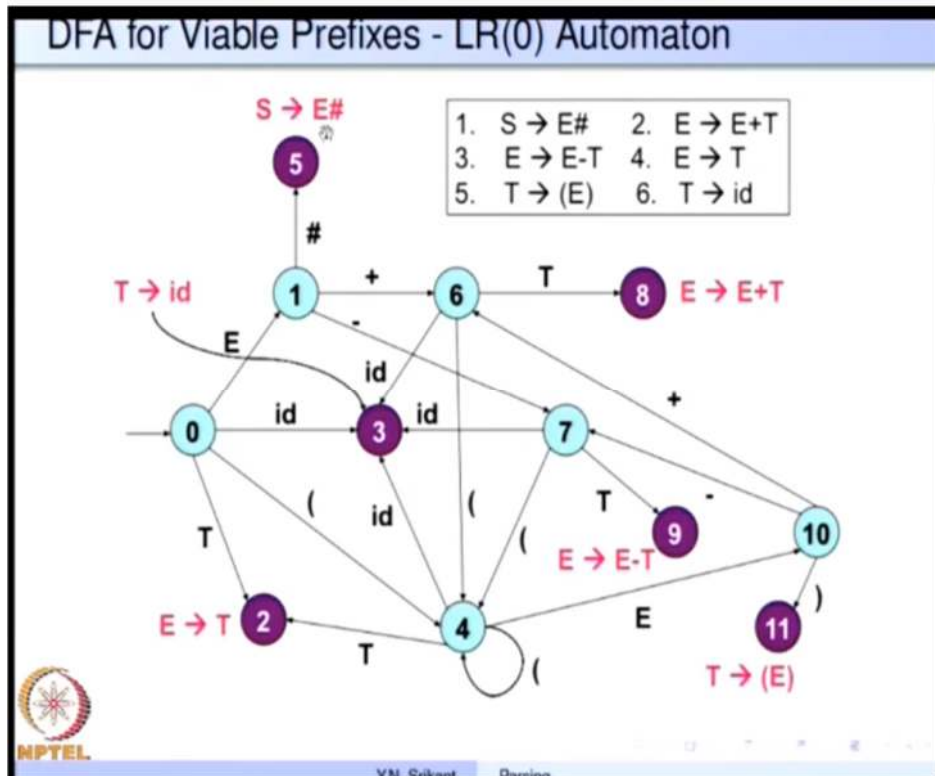
### Gramáticas LR

- ❖ Considere una derivación más a la derecha:  
 $S \Rightarrow_{rm} \phi B t \Rightarrow_{rm} \phi \beta t$ ,  
donde la producción  $B \rightarrow \beta$  ha sido aplicada.
- ❖ Se dice que una gramática es LR (k), si para cualquier cadena de entrada dada, en cada paso de cualquier derivación del extremo derecho, el identificador  $\beta$  se puede detectar examinando la cuerda  $\phi \beta$  y escaneando como máximo k símbolos de la cadena de entrada no utilizada.

- Example: The grammar,  $\{S \rightarrow E, E \rightarrow E + E \mid E * E \mid id\}$ , is not LR(2)
  - $S \Rightarrow^1 \underline{E} \Rightarrow^2 \underline{E + E} \Rightarrow^3 \underline{E + E * E} \Rightarrow^4 \underline{E + E * id} \Rightarrow^5 \underline{E + id * id} \Rightarrow^6 \underline{id + id * id}$
  - $S \Rightarrow^{1'} \underline{E} \Rightarrow^{2'} \underline{E * E} \Rightarrow^{3'} \underline{E * id} \Rightarrow^{4'} \underline{E + E * id} \Rightarrow^{5'} \underline{E + id * id} \Rightarrow^{6'} \underline{id + id * id}$

- ❖ En las dos derivaciones anteriores, la manija en los pasos 6 y 6' y en los pasos 5 y 5' es  $E \rightarrow id$ , y la posición está subrayada (con la misma anticipación de dos símbolos,  $id + y + id$ ).
- ❖ Sin embargo, las manijas en el paso 4 y en el paso 4' son diferentes ( $E \rightarrow id$  y  $E \rightarrow E + E$ ), aunque la anticipación de 2 símbolos es la misma ( $* id$ ), y la pila también es la misma ( $\phi = E + E$ ).
- ❖ Eso significa que el identificador no se puede determinar mediante la búsqueda anticipada.
- ❖ Un prefijo viable de una forma de oración  $\phi\beta t$ , donde  $\beta$  denota el identificador, es cualquier prefijo de  $\phi\beta$ . Un prefijo viable no puede contener símbolos a la derecha del identificador.
- ❖ Siempre es posible agregar símbolos terminales apropiados al final de un prefijo viable para obtener una forma de oración correcta.
- ❖ Los prefijos viables caracterizan los prefijos de formas oracionales que pueden aparecer en la pila de un analizador LR.
- ❖ **Teorema:** El conjunto de todos los prefijos viables de todas las formas oracionales correctas de una gramática es un lenguaje regular.
- ❖ El DFA de este lenguaje normal puede detectar identificadores durante el análisis LR.
- ❖ Cuando este DFA alcanza un "estado de reducción", el prefijo viable correspondiente no puede crecer más y, por lo tanto, indica una reducción.
- ❖ Este DFA puede ser construido por el compilador usando la gramática.
- ❖ Todos los analizadores LR tienen un DFA incorporado.
- ❖ Construimos una gramática aumentada para la que construimos el DFA.
  - Si  $S$  es el símbolo de inicio de  $G$ , entonces  $G'$  contiene todas las producciones de  $G$  y también una nueva producción  $S' \rightarrow S$ .

→ Esto permite que el analizador se detenga tan pronto como aparezca S' en la pila.



**Items and Valid Items**

- A finite set of *items* is associated with each state of DFA
  - An *item* is a marked production of the form  $[A \rightarrow \alpha_1 \cdot \alpha_2]$ , where  $A \rightarrow \alpha_1 \alpha_2$  is a production and  $\cdot$  denotes the mark
  - Many items may be associated with a production  
e.g., the items  $[E \rightarrow \cdot E + T]$ ,  $[E \rightarrow E \cdot + T]$ ,  $[E \rightarrow E + \cdot T]$ , and  $[E \rightarrow E + T \cdot]$  are associated with the production  $E \rightarrow E + T$
- An item  $[A \rightarrow \alpha_1 \cdot \alpha_2]$  is *valid* for some viable prefix  $\phi \alpha_1$ , iff, there exists some rightmost derivation  $S \Rightarrow^* \phi A t \Rightarrow \phi \alpha_1 \alpha_2 t$ , where  $t \in \Sigma^*$
- There may be several items valid for a viable prefix
  - The items  $[E \rightarrow E \cdot - T]$ ,  $[T \rightarrow \cdot id]$ , and  $[T \rightarrow \cdot (E)]$  are all valid for the viable prefix "E-" as shown below  
 $S \Rightarrow E\# \Rightarrow E - T\#$ ,  $S \Rightarrow E\# \Rightarrow E - T\# \Rightarrow E - id\#$ ,  
 $S \Rightarrow E\# \Rightarrow E - T\# \Rightarrow E - (E)\#$

## Elementos y estados válidos de LR (0) DFA

- ❖ Un ítem indica cuánta producción ya se ha visto y cuánto queda por ver.
- ❖  $[E \rightarrow E - T]$  indica que ya hemos visto una cadena derivable de "E-" y que esperamos ver a continuación, una cadena derivable de T.
- ❖ Cada estado de un DFA LR (0) contiene sólo aquellos elementos que son válidos para el mismo conjunto de prefijos viables.
  - Todos los elementos del estado 7 son válidos para los prefijos viables "E-" y "E -" (y muchos más).
  - Todos los elementos del estado 4 son válidos para el prefijo viable "(" (y muchos más).
  - De hecho, el conjunto de todos los prefijos viables para los que los elementos en un estado  $s$  son válidos es el conjunto de cadenas que pueden llevarnos del estado 0 (inicial) al estado  $s$ .
- ❖ Construir el DFA LR (0) usando conjuntos de elementos es muy simple.

**Closure of a Set of Items**

```

Itemset closure(I) { /* I is a set of items */
  while (more items can be added to I) {
    for each item  $[A \rightarrow \alpha.B\beta] \in I$  {
      /* note that B is a nonterminal and is right after the "." */
      for each production  $B \rightarrow \gamma \in G$ 
        if (item  $[B \rightarrow \gamma] \notin I$ ) add item  $[B \rightarrow \gamma]$  to I
    }
  }
  return I
}
    
```

State 0	State 1	State 7	State 2
$S \rightarrow E\#$	$S \rightarrow E\#$	$E \rightarrow E-.T$	$E \rightarrow T.$
$E \rightarrow E+.T$	$E \rightarrow E+.T$	$T \rightarrow (E)$	
$E \rightarrow E-.T$	$E \rightarrow E-.T$	$T \rightarrow .id$	
$E \rightarrow .T$			
$T \rightarrow (E)$			
$T \rightarrow .id$			

● indicates closure items

**GOTO set computation**

```

Itemset GOTO(I, X) { /* I is a set of items
  X is a grammar symbol, a terminal or a nonterminal */
  Let  $I' = \{[A \rightarrow \alpha.X\beta] \mid [A \rightarrow \alpha.X\beta] \in I\}$ ;
  return (closure(I'))
}
    
```

State 0	State 1	State 7
$S \rightarrow E\#$	$S \rightarrow E\#$	$E \rightarrow E-.T$
$E \rightarrow E+.T$	$E \rightarrow E+.T$	$T \rightarrow (E)$
$E \rightarrow E-.T$	$E \rightarrow E-.T$	$T \rightarrow .id$
$E \rightarrow .T$		
$T \rightarrow (E)$		
$T \rightarrow .id$		

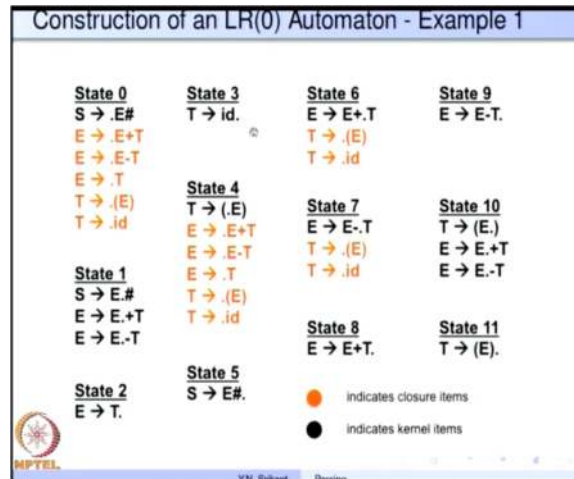
● indicates closure items

GOTO(0, E) = 1  
GOTO(1, -) = 7

- ❖ Si un artículo  $[A \rightarrow \alpha.B\delta]$  está en un estado (es decir, el conjunto de elementos  $I$ ), entonces, en algún momento en el futuro, esperamos ver en la entrada  $a$ , una cadena derivable de  $B\delta$ .
  - Esto implica una cadena derivable de B también.
  - Por lo tanto, agregamos un elemento  $[B \rightarrow \beta]$  de B, al estado (es decir, conjunto de elementos  $I$ ).
- ❖ Si  $I$  es el conjunto de elementos válido para un prefijo viable  $\gamma$ 
  - Todos los elementos del cierre ( $I$ ) también son válidos para  $\gamma$
  - GOTO ( $I, X$ ) son los elementos del conjunto válidos para el prefijo viable  $\gamma X$ .



- ★ Si  $[A \rightarrow \alpha, B\delta]$  (en el conjunto de elementos  $I$ ) es válido para el prefijo viable  $\phi\alpha$ , y  $B \rightarrow \beta$  es una producción, tenemos  $S \Rightarrow^* \phi A t \Rightarrow \phi \alpha B \delta t \Rightarrow^* \phi \alpha B x t \Rightarrow \phi \alpha \beta x t$  demostrando que el artículo  $[B \rightarrow \beta]$  (En el cierre de  $I$ ) es válido para  $\phi\alpha$ .
- ★ La derivación anterior también muestra que el artículo  $[A \rightarrow \alpha B\delta]$  en GOTO ( $I, B$ ) es válido para el prefijo viable  $\phi\alpha B$ .



## Cambiar y Reducir Acciones

- ❖ Si un estado contiene un elemento del formulario  $[A \rightarrow \alpha]$  ("reducir artículo"), luego una reducción por la producción  $A \rightarrow \alpha$  es la acción en ese estado.
- ❖ Si no hay "elementos reducidos" en un estado, entonces cambiar es la acción apropiada.
- ❖ Podría haber cambios-reducir conflictos o reducir-reducir conflictos en un estado.
  - Ambos elementos de desplazamiento y reducción están presentes en el mismo estado (conflicto S-R).
  - Hay más de un elemento de reducción en un estado (conflicto R-R).
  - Es normal tener más de un elemento de turno en un estado (no es posible que haya conflictos de turno).
- ❖ Si no hay conflictos S-R o R-R en ningún estado de un DFA LR (0), entonces la gramática es LR (0), de lo contrario, no es LR (0).

## Mod-03 Lec-10 Análisis Sintáctico

### Analizadores SLR (1)

- ❖ Si la gramática no es LR (0), intentamos resolver los conflictos en los estados usando un símbolo de anticipación.
- ❖ Ejemplo: la expresión gramática que no es LR (0) El estado que contiene los elementos  $[T \rightarrow F]$  y  $[T \rightarrow F * T]$  tiene conflictos S-R.
  - Considere la reducción de artículos  $[T \rightarrow F]$  y los símbolos en FOLLOW (T).
  - FOLLOW (T) =  $\{+, \}, \$\}$ , y reducción por  $T \rightarrow F$  se puede realizar al ver uno de estos símbolos en la entrada (mirar hacia adelante), ya que el cambio requiere ver  $*$  en la entrada.
  - Recuerde de la definición de FOLLOW (T) que los símbolos en FOLLOW (T) son los únicos símbolos que pueden seguir legalmente a T en cualquier forma de oración, y por lo tanto, la reducción por  $T \rightarrow F$  cuando se ve uno de estos símbolos, es correcto.
  - Si los conflictos S-R se pueden resolver usando el conjunto FOLLOW, se dice que la gramática es SLR (1).

**A Grammar that is not LR(0) - Example 2**

<b>State 0</b> $S \rightarrow E$ $E \rightarrow E+T$ $E \rightarrow T$ $T \rightarrow F^*T$ $T \rightarrow F$ $F \rightarrow (E)$ $F \rightarrow id$	<b>State 2</b> $E \rightarrow T$  <b>State 3</b> $T \rightarrow F^*T$ $T \rightarrow F$ Shift-reduce conflict	<b>State 5</b> $F \rightarrow id$  <b>State 6</b> $E \rightarrow E+T$ $T \rightarrow F^*T$ $T \rightarrow F$ $F \rightarrow (E)$ $F \rightarrow id$	<b>State 8</b> $F \rightarrow (E)$ $E \rightarrow E+T$  <b>State 9</b> $E \rightarrow E+T$
<b>State 1</b> $S \rightarrow E$ $E \rightarrow E+T$ Shift-reduce conflict	<b>State 4</b> $F \rightarrow (E)$ $E \rightarrow E+T$ $E \rightarrow T$ $T \rightarrow F^*T$ $T \rightarrow F$ $F \rightarrow (E)$ $F \rightarrow id$	<b>State 7</b> $T \rightarrow F^*T$ $T \rightarrow F^*T$ $T \rightarrow F$ $F \rightarrow (E)$ $F \rightarrow id$	<b>State 10</b> $E \rightarrow F^*T$  <b>State 11</b> $F \rightarrow (E)$

follow(S) =  $\{ \$ \}$ . Reduction on \$ and shift on +, eliminates conflicts

follow(T) =  $\{ \$, ), + \}$ , where \$ is EOF

Reduction on \$, ), and +, and shift on \*, eliminates conflicts

Grammar is not LR(0), but is SLR(1)

NPTEL

Y.N. Srikant Decision

### Construction of an SLR(1) Parsing Table

Let  $C = \{I_0, I_1, \dots, I_i, \dots, I_n\}$  be the canonical LR(0) collection of items, with the corresponding states of the parser being  $0, 1, \dots, i, \dots, n$ . Without loss of generality, let 0 be the initial state of the parser (containing the item  $[S' \rightarrow \cdot S]$ )

Parsing actions for state  $i$  are determined as follows

- If  $([A \rightarrow \alpha \cdot a \beta] \in I_i) \ \&\& \ ([A \rightarrow \alpha a \cdot \beta] \in I_j)$   
set  $ACTION[i, a] = \text{shift } j$  /\*  $a$  is a terminal symbol \*/
- If  $([A \rightarrow \alpha \cdot] \in I_i)$   
set  $ACTION[i, a] = \text{reduce } A \rightarrow \alpha$ , for all  $a \in \text{follow}(A)$
- If  $([S' \rightarrow \cdot S] \in I_i)$  set  $ACTION[i, \$] = \text{accept}$
- If  $([A \rightarrow \alpha \cdot A \beta] \in I_i) \ \&\& \ ([A \rightarrow \alpha A \cdot \beta] \in I_j)$   
set  $GOTO[i, A] = j$  /\*  $A$  is a nonterminal symbol \*/

All other entries not defined by the rules above are made *error*

### A Grammar that is not LR(0) - Example 3

Grammar  
 $S' \rightarrow S, S \rightarrow aSb, S \rightarrow \epsilon$

$\text{follow}(S) = \{\$, b\}$

	a	b	\$	S
0	S2	reduce $S \rightarrow \epsilon$	reduce $S \rightarrow \epsilon$	1
1			accept	
2	S2	reduce $S \rightarrow \epsilon$	reduce $S \rightarrow \epsilon$	3
3		S4		
4	reduce $S \rightarrow aSb$	reduce $S \rightarrow aSb$		

State 0:  $S' \rightarrow \cdot S$   
 State 1:  $S' \rightarrow S \cdot$   
 State 2:  $S \rightarrow \cdot aSb$   
 State 3:  $S \rightarrow a \cdot Sb$   
 State 4:  $S \rightarrow aS \cdot b$

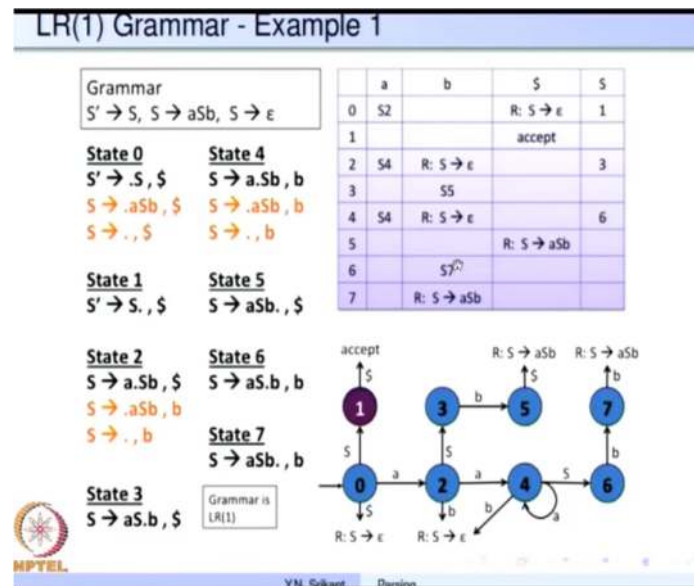
shift-reduce conflict in states 0, 2

indicates closure items  
 indicates kernel items

Grammar is not LR(0), but is SLR(1)

## El problema con los analizadores SLR (1)

- ❖ El proceso de construcción del analizador SLR (1) no recuerda suficiente contexto izquierdo para resolver conflictos.
  - En la gramática " $L = R$ " (diapositiva anterior), el símbolo '=' entró en follow (R) debido a la siguiente derivación:
 
$$S' \Rightarrow S \Rightarrow L = R \Rightarrow L = L \Rightarrow L = id \Rightarrow *R = id \Rightarrow \dots$$
  - La producción utilizada es  $L \rightarrow *R$
  - La siguiente derivación más a la derecha en reversa no existe (y por lo tanto la reducción por  $R \rightarrow L$  en '=' en el estado 2 es ilegal).  $id = id \leftarrow L = id \leftarrow R = id \dots$
- ❖ Generalización del ejemplo anterior
  - En algunas situaciones, cuando aparece un estado  $i$  en la parte superior de la pila, un prefijo viable  $\beta\alpha$  puede estar en la pila de modo que  $\beta A$  no puede ir seguido de "a" en ninguna forma de oración correcta.
  - Así, la reducción por  $A \rightarrow \alpha$  sería inválida en 'a'.
  - En el ejemplo anterior,  $\beta = \epsilon, \alpha = L$ , y  $A = R$ ; L no se puede reducir a R en "=", ya que conduciría a la secuencia de derivación ilegal anterior.

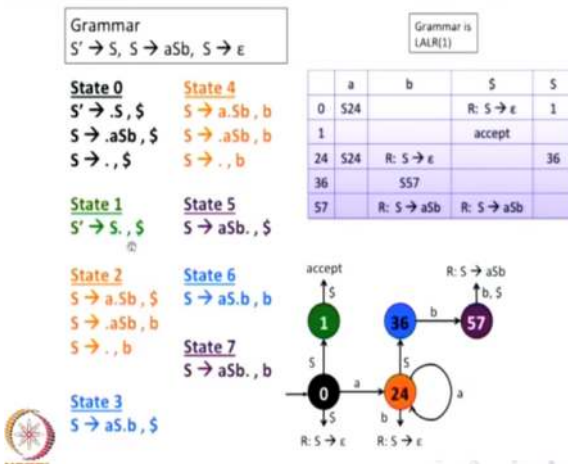


## Mod-03 Lec-11 Análisis Sintáctico

### Analizadores LALR (1)

- ❖ Los analizadores sintácticos LR (1) tienen una gran cantidad de estados
  - Para C, muchos miles de estados.
  - Un analizador SLR (1) (o LR (0) DFA) para C tendrá unos cientos de estados (con muchos conflictos).
- ❖ Los analizadores sintácticos LALR (1) tienen exactamente el mismo número de estados que los analizadores sintácticos SLR (1) para la misma gramática y se derivan de los analizadores sintácticos LR (1).
  - Los analizadores SLR (1) pueden tener muchos conflictos, pero los analizadores LALR(1) pueden tener muy pocos conflictos.
  - Si el analizador LR (1) no tuvo conflictos S-R, entonces el analizador LALR(1) derivado correspondiente tampoco tendrá ninguno.
  - Sin embargo, esto no es cierto con respecto a los conflictos R-R.
- ❖ Los analizadores sintácticos LALR(1) son tan compactos como los analizadores sintácticos SLR (1) y son casi tan potentes como los analizadores sintácticos LR (1).
- ❖ La mayoría de las gramáticas de los lenguajes de programación también son LALR(1), si son LR (1).

### LALR(1) Parser Construction - Example 1



### LALR(1) Parser Construction - Example 1 (contd.)

LR(1) Parser Table

	a	b	\$	S
0	S24		R: $S \rightarrow \epsilon$	1
1			accept	
2	S4	R: $S \rightarrow \epsilon$		3
3		SS		
4	S4	R: $S \rightarrow \epsilon$		6
5			R: $S \rightarrow aSb$	
6		S7		
7	R: $S \rightarrow aSb$			

LALR(1) Parser Table

	a	b	\$	S
0	S24		R: $S \rightarrow \epsilon$	1
1			accept	
24	S24	R: $S \rightarrow \epsilon$		36
36		SS7		
57	R: $S \rightarrow aSb$	R: $S \rightarrow aSb$		

## Construcción de analizadores sintácticos LALR (1)

- ❖ La parte principal de los elementos LR (1) (la parte que sigue a la omisión del símbolo de anticipación) es la misma para varios estados de LR (1) (los símbolos de anticipación serán diferentes).
  - Fusionar los estados con el mismo núcleo, junto con los símbolos de anticipación, y cambiarles el nombre.
- ❖ Las partes ACTION y GOTO de la tabla del analizador se modificarán.
  - Fusionar las filas de la tabla del analizador correspondientes a los estados fusionados, reemplazando los nombres antiguos de los estados por los nuevos nombres correspondientes para los estados fusionados.
  - Por ejemplo, si los estados 2 y 4 se fusionan en un nuevo estado 36, todas las referencias a los estados 2, 3 y 6 serán reemplazadas por 24, 24, 36 y 36, respectivamente.
- ❖ Los analizadores LALR (1) pueden realizar algunas reducciones más (pero no cambios) que un analizador LR (1) antes de detectar un error.

LALR(1) Parser Error Detection		
LR(1) Parser		
0	ab\$	shift
0 a 2	b\$	$S \rightarrow \epsilon$
0 a 2 5 3	b\$	shift
0 a 2 5 3 b 5	\$	$S \rightarrow aSb$
0 5 1	\$	accept
0	aa\$	shift
0 a 2	a\$	shift
0 a 2 a 4	\$	error
0	aab\$	shift
0 a 2	ab\$	shift
0 a 2 a 4	b\$	$S \rightarrow \epsilon$
0 a 2 a 4 5 6	b\$	shift
0 a 2 a 4 5 6 b 7	\$	error

LALR(1) Parser		
0	ab\$	shift
0 a 2 4	b\$	$S \rightarrow \epsilon$
0 a 2 4 5 3 6	b\$	shift
0 a 2 4 5 3 6 b 5 7	\$	$S \rightarrow aSb$
0 5 1	\$	accept
0	aa\$	shift
0 a 2 4	a\$	shift
0 a 2 4 a 2 4	\$	error
0	aab\$	shift
0 a 2 4	ab\$	shift
0 a 2 4 a 2 4	b\$	$S \rightarrow \epsilon$
0 a 2 4 a 2 4 5 3 6	b\$	shift
0 a 2 4 a 2 4 5 3 6 b 5 7	\$	$S \rightarrow aSb$
0 a 2 4 5 3 6	\$	error

## Características de los analizadores LALR (1)

- ❖ Si un analizador LR (1) no tiene conflictos S-R, entonces el analizador LALR (1) derivado correspondiente tampoco tendrá ninguno.
  - Los estados del analizador LR (1) y LALR (1) tienen los mismos elementos principales (las búsquedas anticipadas pueden no ser las mismas).
  - Si un estado  $s_1$  del analizador LALR (1) tiene un conflicto S-R, deben tener dos elementos  $[A \rightarrow \alpha., a]$  y  $[B \rightarrow \beta, a\gamma, b]$ .
  - Uno de los estados  $s_1'$ , a partir del cual se genera  $s_1$ , debe tener los mismos elementos básicos que  $s_1$ .
  - Si el artículo  $[A \rightarrow \alpha., a]$  está en  $s_1'$ , entonces  $s_1'$  también debe tener el elemento  $[B \rightarrow \beta, a\gamma, c]$  (la búsqueda anticipada no necesita ser b en  $s_1'$  puede ser b en algún otro estado, pero eso no nos interesa).
  - Estos dos elementos en  $s_1'$  todavía crean un conflicto S-R en el analizador LR (1).
  - Por lo tanto, la fusión de estados con un núcleo común nunca puede introducir un nuevo conflicto de S-R, porque el cambio depende sólo del núcleo, no de la anticipación.
  - Sin embargo, la fusión de estados puede introducir un nuevo conflicto R-R en el analizador LALR (1) aunque el analizador LR (1) original no tuviera ninguno.
  - Tales gramáticas son raras en la práctica.

- Aquí hay uno del libro de ALSU. Construya los juegos completos de elementos LR (1) como trabajo a domicilio:

$$\begin{aligned} S' &\rightarrow S \$, S \rightarrow aAd \mid bBd \mid aBe \mid bAe \\ A &\rightarrow c, B \rightarrow c \end{aligned}$$

- Dos estados contienen los elementos:

$$\{[A \rightarrow c., d], [B \rightarrow c., e]\} \text{ y } \{[A \rightarrow c., e], [B \rightarrow c., d]\}$$

- La fusión de estos dos estados produce el estado LALR (1):

$$\{[A \rightarrow c., d / e], [B \rightarrow c., d / e]\}$$

- Este estado LALR (1) tiene un conflicto de reducir-reducir

## **Recuperación de errores en analizadores LR: construcción del analizador**

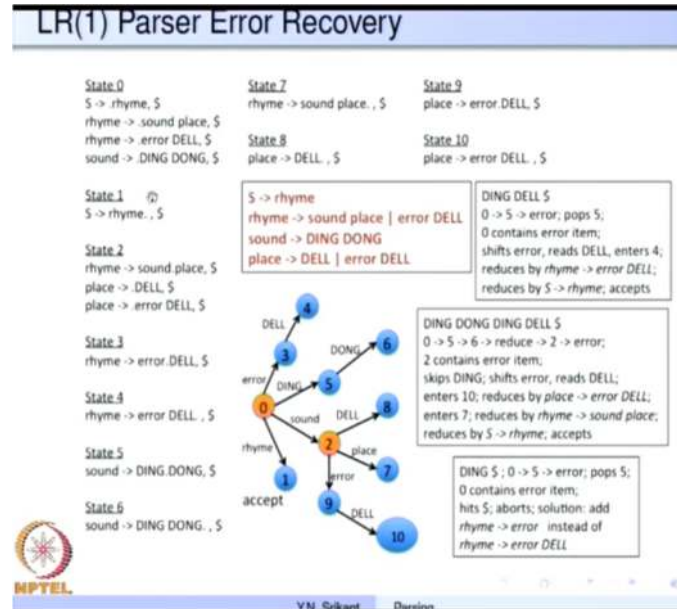
- ❖ El escritor del compilador identifica los principales no terminales, como los de programa, declaración, bloque, expresión, etc.
- ❖ Agrega a la gramática, producciones de error de la forma  $A \rightarrow error \alpha$ , Donde A es un importante no terminal y  $\alpha$  es una cadena adecuada de símbolos gramaticales (generalmente símbolos terminales), posiblemente vacía.
- ❖ Asocia una rutina de mensaje de error con cada producción de error.
- ❖ Construye un analizador LALR (1) para la nueva gramática con producciones de error.

## **Recuperación de errores en analizadores LR: funcionamiento del analizador**

- ❖ Cuando el analizador encuentra un error, escanea la pila para encontrar el estado superior que contiene un elemento de error del formulario  $A \rightarrow error \alpha$
- ❖ El analizador luego cambia un error de token como si hubiera ocurrido en la entrada.
- ❖ Si  $\alpha = \epsilon$ , reducido por  $A \rightarrow \epsilon$  e invoca la rutina de mensajes de error asociada con él.
- ❖ Si  $\alpha \neq \epsilon$ , descarta los símbolos de entrada hasta que encuentra un símbolo con el que el analizador puede continuar



- ❖ Reducción por  $A \rightarrow, error$   $\alpha$  ocurre en el momento apropiado Ejemplo: Si la producción de error es  $A \rightarrow, error$ ;;, y procede como arriba.
- ❖ La recuperación de errores no es perfecta y el analizador puede abortar al final de la entrada.



## LEX and YACC

### YACC Example

```
%token DING DONG DELL
%start rhyme
%%
rhyme : *sound place '\n'
      { printf("string valid\n"); exit(0); }
sound : DING DONG ;
place : DELL ;
%%
#include "lex.yy.c"

int yywrap(){return 1;}
yyerror( char* s)
{ printf("%s\n",s); }
main() {yyparse(); }
```

### LEX Specification for the YACC Example

```
%%
ding return DING;
dong return DONG;
dell return DELL;
[ ]* ;
\n|. return yytext[0];

Compiling and running the parser
lex ding-dong.l
yacc ding-dong.y
gcc -o ding-dong.o y.tab.c
ding-dong.o

Sample inputs      ||      Sample outputs
ding dong dell     ||      string valid
ding dell          ||      syntax error
ding dong dell$    ||      syntax error
```

## Forma de un archivo YACC

- ❖ YACC tiene un lenguaje para describir gramáticas libres de contexto.
- ❖ Genera un analizador LALR (1) para el CFG descrito.

- ❖ Forma de un programa YACC

```
%{declaraciones - opcional
%}
%%
reglas - obligatorio
%%

programas - opcional
```
- ❖ YACC utiliza el analizador léxico generado por LEX para hacer coincidir los símbolos terminales del CFG.
- ❖ YACC genera un archivo llamado y.tab.c

## Declaraciones y Reglas

- ❖ **Tokens:** %token nombre1 nombre2 nombre3, ...
- ❖ **Símbolo de Inicio:** %nombre de inicio
- ❖ **nombres** en reglas: letra (letra | dígito | . | \_)\* la letra puede ser minúscula o mayúscula
- ❖ **Valores de los símbolos y acciones:** Ejemplo

```
A : B
    { $$ = 1; }
    C
    { x = $2; y = $3; $$ = x+y; }
    ;
```

- Now, value of A is stored in \$\$ (second one), that of B in \$1, that of action 1 in \$2, and that of C in \$3.

- ❖ La acción intermedia en el ejemplo anterior se traduce en una -producción de la siguiente manera:

```
$ACT1 : /* empty */
    { $$ = 1; }
    ;
A : B $ACT1 C
    { x = $2; y = $3; $$ = x+y; }
    ;
```

- ❖ Las acciones intermedias pueden devolver valores

- ❖ Por ejemplo, el primer \$\$ del ejemplo anterior está disponible como \$2
- ❖ Sin embargo, las acciones intermedias no pueden referirse a los valores de los símbolos a la izquierda de la acción.
- ❖ Las acciones se traducen en código C que se ejecutan justo antes de que el analizador realice una reducción.

## **Análisis Léxico**

LA devuelve los números enteros como números simbólicos.

Los números de tokens son asignados automáticamente por YACC, comenzando desde 257, para todos los tokens declarados usando la declaración de **%token**.

Los tokens pueden devolver no solo números de token sino también otra información (por ejemplo, valor de un número, cadena de caracteres de un nombre, puntero a la tabla de símbolos, etc.).

Los valores adicionales se devuelven en la variable, **yylval**, conocida por los analizadores generados por YACC.

## **Symbol Values**

- ❖ Los tokens y los no terminales son símbolos de pila.
- ❖ Los símbolos de pila se pueden asociar con valores cuyos **tipos** se declaran en una declaración %union en el archivo de especificación YACC.
- ❖ YACC convierte esto en un tipo de unión llamado YYSTYPE.
- ❖ Con declaraciones de % token y % type, informamos a YACC sobre los tipos de valores que toman los tokens y los no terminales.
- ❖ Automáticamente, las referencias a \$1, \$2, **yylval**, etc., se refieren al miembro apropiado del sindicato (vea el ejemplo a continuación).

## **Recuperación de errores en YACC**

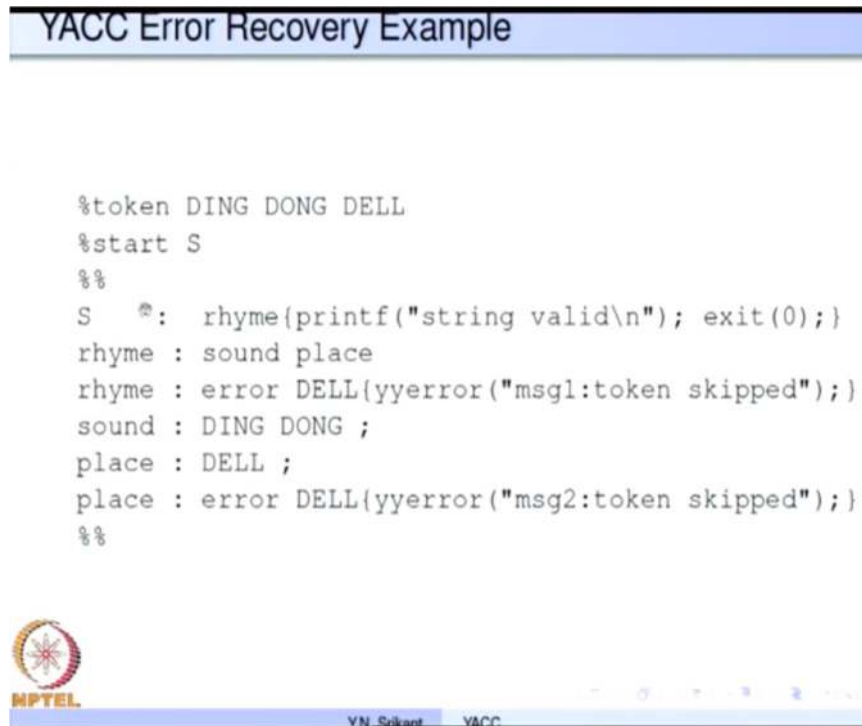
- ❖ Para evitar una cascada de mensajes de error, el analizador permanece en estado de error (después de ingresarlo) hasta que tres tokens se hayan transferido con éxito a la pila.

- ❖ En caso de que ocurra un error antes de esto, no se dan más mensajes y el símbolo de entrada (que causa el error) se elimina silenciosamente.
- ❖ El usuario puede identificar no terminales **mayores** como los de **programa**, **declaración**, o **bloque**, y agregue producciones de error para estos a la gramática.

Ejemplos:

*declaración* → **error** {acción 1}

*declaración* → **error** ‘,’ {acción 2}



## Actividades Semana 8

### Mod-04 Lec-12 Análisis Semántico

#### Análisis Semántico

- ❖ La consistencia semántica que no puede ser manejada en la etapa de análisis se maneja aquí.
- ❖ Los analizadores no pueden manejar las características sensibles al contexto de los lenguajes de programación.
- ❖ Son semánticas estáticas de los lenguajes de programación y pueden ser comprobadas por el analizador semántico.

- Las variables se declaran antes de su uso.
- Los tipos coinciden en ambos lados de las asignaciones.
- Los tipos de parámetros y el número coinciden en la declaración y el uso.
- ❖ Los compiladores sólo pueden generar código para comprobar la semántica dinámica de los lenguajes de programación en tiempo de ejecución.
  - Si se produce un desbordamiento durante una operación aritmética.
  - Si los límites de la matriz se cruzaran durante la ejecución.
  - Si la recursividad cruzara los límites de la pila.
  - Si la memoria del cabezal es insuficiente.
- ❖ La información de los tipos se almacena en la tabla de símbolos o en el árbol de sintaxis.
  - Tipos de variables, parámetros de función, dimensiones de la matriz, etc.
  - Se utiliza no sólo para la validación semántica sino también para las fases posteriores de la compilación.
- ❖ Si no es necesario que las declaraciones aparezcan antes de su uso (como en C++), el análisis semántico necesita más de una pasada.
- ❖ La semántica estática de PL puede especificarse utilizando gramáticas de atributos.
- ❖ Los analizadores semánticos pueden generarse semi automáticamente a partir de atributos.
- ❖ Las gramáticas de atributos son extensiones de las gramáticas sin contexto.

## Semántica Estática

```
int dot_prod(int x[], int y[]){
    int d, i; d = 0;
    for (i=0; i<10; i++) d += x[i]*y[i];
    return d;
}
main(){
    int p; int a[10], b[10];
    p = dot_prod(a,b);
}
```

❖ Muestras de comprobaciones semánticas estáticas en principal.

- Los tipos de p y el tipo de retorno de dot\_prod coinciden.
- El número y el tipo de los parámetros de dot\_prod son los mismos tanto en su declaración como en su uso.
- p es declarado antes de su uso, igual para a y b.


Static Semantics: Errors given by gcc Compiler

```
int dot_product(int a[], int b[]) {...}

1 main(){int a[10]={1,2,3,4,5,6,7,8,9,10};
2 int b[10]={1,2,3,4,5,6,7,8,9,10};
3 printf("%d", dot_product(b));
4 printf("%d", dot_product(a,b,a));
5 int p[10]; p=dotproduct(a,b); printf("%d",p);}
```

In function 'main':

- error in 3: too few arguments to fn 'dot\_product'
- error in 4: too many arguments to fn 'dot\_product'
- error in 5: incompatible types in assignment
- warning in 5: format '%d' expects type 'int', but argument 2 has type 'int \*'



YH Sukant Semantic Analysis

❖ Muestras de comprobaciones semánticas estáticas en dot\_prod.

- d e i se declaran antes de su uso.
- El tipo de d coincide con el tipo de retorno de dot\_prod.

- El tipo de  $d$  coincide con el tipo de resultado de  $*$ .
- Los elementos de las matrices  $x$  e  $y$  son compatibles con  $*$ .

## Semántica Dinámica

```
int dot_prod(int x[], int y[]){
    int d, i; d = 0;
    for (i=0; i<10; i++) d += x[i]*y[i];
    return d;
}
main(){
    int p; int a[10], b[10];
    p = dot_prod(a,b);
}
```

### ❖ Muestras de comprobaciones semánticas dinámicas en dot\_prod.

- El valor de  $i$  no excede el rango declarado de las matrices  $x$  e  $y$  (tanto inferior como superior).
- No hay desbordamientos durante las operaciones de  $*$  y  $+$  en  $d += x[i] * y[i]$ .

### ❖ Muestras de comprobaciones semánticas dinámicas en dot\_prod.

- El valor de  $i$  no excede el rango declarado de las matrices  $x$  e  $y$  (tanto inferior como superior).
- No hay desbordamientos durante las operaciones de  $*$  y  $+$  en  $d += x[i] * y[i]$ .

```
int fact(int n){
    if (n==0) return 1;
    else return (n*fact(n-1));
}
main(){int p; p = fact(10); }
```

### ❖ Muestras de comprobaciones semánticas dinámicas de hecho.

- La comprobación del programa no se desborda debido a la recursividad.



→ No hay desbordamiento debido a “\*” en  $n * fact(n - 1)$ .

## **Gramáticas de Atributos**

- ❖ Dejemos que  $G = (N, T, P, S)$  sea un CFG y dejemos  $V = N \cup T$ .
- ❖ Cada símbolo  $X$  de  $V$  tiene asociado un conjunto de atributos (denotados por  $X.a$ ,  $X.b$ , etc.).
- ❖ Dos tipos de atributos: heredados (denotados por  $Al(X)$ ) y sintetizados (denotados por  $AS(X)$ ).
- ❖ Cada atributo toma valores de un dominio especificado (finito o infinito), que es su tipo.
  - Los dominios típicos de los atributos son, enteros, reales, caracteres, cadenas, booleanos, estructuras, etc.
  - Se pueden construir nuevos dominios a partir de dominios dados mediante operaciones matemáticas como producto cruzado, mapa, etc.
  - formación: un mapa,  $N \rightarrow D$ , dónde,  $N$  y  $D$  son los dominios de los números naturales y los objetos dados, respectivamente.
  - estructura: un producto cruzado,  $A_1 \times A_2 \times \dots \times A_n$ , dónde  $n$  es el número de campos en la estructura, y  $A_i$  es el dominio del campo  $i^{th}$ .

## **Reglas de Cálculo de Atributos**

- ❖ Una producción  $p \in P$  tiene un conjunto de reglas de cálculo de atributos (funciones).
- ❖ Se prevén normas para el cálculo de:
  - Atributos sintetizados del LHS no terminal de  $p$ .
  - Atributos heredados de los no terminales RHS de  $p$ .
- ❖ Estas reglas pueden usar atributos de los símbolos de la producción de  $p$  solamente.
  - Las reglas son estrictamente locales para la producción  $p$  (sin efectos secundarios).

- ❖ Las restricciones de las reglas definen diferentes tipos de atributos gramaticales.
  - Gramáticas de atributo L, gramáticas de atributo S, gramáticas de atributo ordenadas, gramáticas de atributo absolutamente no circulares, gramáticas de atributo circulares, etc.

## **Atributos Sintetizados y Heredados.**

- ❖ Un atributo no puede ser a la vez sintetizado y heredado, pero un símbolo puede tener ambos tipos de atributos.
- ❖ Los atributos de los símbolos se evalúan sobre un árbol de análisis haciendo pasadas sobre el árbol de análisis.
- ❖ Los atributos sintetizados se calculan de abajo hacia arriba desde las hojas hacia arriba.
  - Siempre se sintetiza a partir de los valores de atributo de los hijos del nodo.
  - Los nodos de la hoja (terminales) tienen atributos sintetizados inicializados por el analizador léxico y no pueden ser modificados.
  - Un AG con sólo atributos sintetizados es una gramática de atributos S (SAG).
  - La YACC sólo permite SAGs.
- ❖ Los atributos heredados fluyen desde el padre o los hermanos hasta el nodo en cuestión.

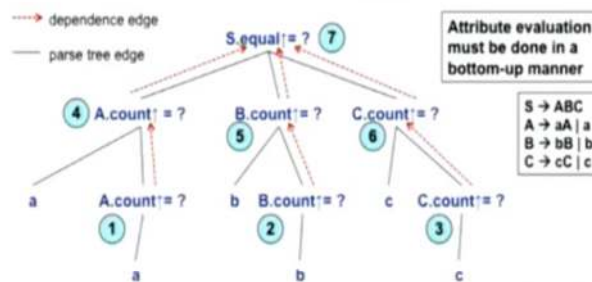
## Attribute Grammar - Example 1

- The following CFG  
 $S \rightarrow A B C$ ,  $A \rightarrow aA \mid a$ ,  $B \rightarrow bB \mid b$ ,  $C \rightarrow cC \mid c$   
generates:  $L(G) = \{a^m b^n c^p \mid m, n, p \geq 1\}$
- We define an AG (attribute grammar) based on this CFG to generate  $L = \{a^n b^n c^n \mid n \geq 1\}$
- All the non-terminals will have only synthesized attributes
  - $AS(S) = \{equal \uparrow: \{T, F\}\}$
  - $AS(A) = AS(B) = AS(C) = \{count \uparrow: integer\}$



Y.N. Srikant Semantic Analysis

## Attribute Grammar - Example 1 (contd.)



- 1  $S \rightarrow ABC \{ S.equal \uparrow := \text{if } A.count \uparrow = B.count \uparrow \ \& \ B.count \uparrow = C.count \uparrow \text{ then } T \text{ else } F \}$
- 2  $A_1 \rightarrow aA_2 \{ A_1.count \uparrow := A_2.count \uparrow + 1 \}$
- 3  $A \rightarrow a \{ A.count \uparrow := 1 \}$
- 4  $B_1 \rightarrow bB_2 \{ B_1.count \uparrow := B_2.count \uparrow + 1 \}$
- 5  $B \rightarrow b \{ B.count \uparrow := 1 \}$
- 6  $C_1 \rightarrow cC_2 \{ C_1.count \uparrow := C_2.count \uparrow + 1 \}$
- 7  $C \rightarrow c \{ C.count \uparrow := 1 \}$



Y.N. Srikant Semantic Analysis

## Gráfico de Dependencia de Atributos

- ❖ Que  $T$  sea un árbol de análisis generado por el CFG de un AG,  $G$ .
- ❖ El gráfico de dependencia de atributos (gráfico de dependencia para abreviar) para  $T$  es el gráfico dirigido,  $DG(T) = (V, E)$ , donde

$V = \{b \mid b \text{ es una instancia de atributo de algún nodo del árbol}\}$ , y

$E = \{(b, c) | b, c \in V, b \text{ y } c \text{ son atributos de símbolos gramaticales en la misma producción } p \text{ de } B, \text{ y el valor de } b \text{ se utiliza para calcular el valor de } c \text{ en una regla de cálculo de atributos asociada a la producción } p\}$

- ❖ An AG  $G$  is non-circular, iff for all trees  $T$  derived from  $G$ ,  $DG(T)$  is acyclic
  - Non-circularity is very expensive to determine (exponential in the size of the grammar).
  - Therefore, our interest will be in subclasses of AGs whose non-circularity can be determined efficiently.
- ❖ Assigning consistent values to the attribute instances in  $DG(T)$  is attribute evaluation.

## **Estrategia de Evaluación de Atributos**

- ❖ Construye el árbol de análisis.
- ❖ Construye el gráfico de dependencia.
- ❖ Realizar una clasificación topológica en el gráfico de dependencia y obtener un orden de evaluación.
- ❖ Evaluar los atributos según este orden utilizando las reglas de evaluación de atributos correspondientes adjuntas a las producciones respectivas.
- ❖ Múltiples atributos en un nodo del árbol de análisis pueden hacer que ese nodo sea visitado varias veces.
  - Cada visita resulta en la evaluación de por lo menos un atributo.

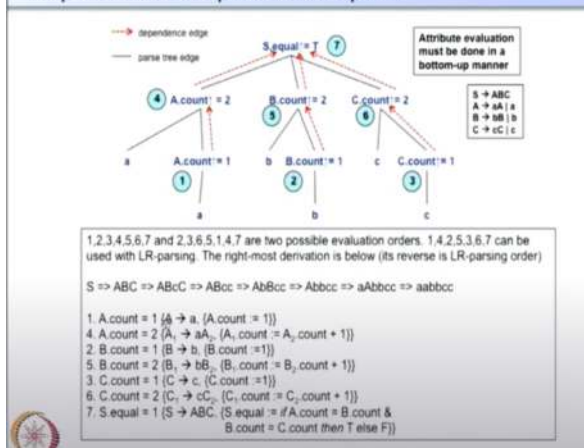
## Attribute Evaluation Algorithm

**Input:** A parse tree  $T$  with unevaluated attribute instances

**Output:**  $T$  with consistent attribute values

```
{ Let  $(V, E) = DG(T)$ ;
  Let  $W = \{b \mid b \in V \ \& \ indegree(b) = 0\}$ ;
  while  $W \neq \emptyset$  do
    { remove some  $b$  from  $W$ ;
       $value(b) :=$  value defined by appropriate attribute
        computation rule;
    for all  $(b, c) \in E$  do
      {  $indegree(c) := indegree(c) - 1$ ;
        if  $indegree(c) = 0$  then  $W := W \cup \{c\}$ ;
      }
    }
```

### Dependence Graph for Example 1



### Attribute Grammar - Example 2

- AG for the evaluation of a real number from its bit-string representation  
Example:  $110.101 = 6.625$
- $N \rightarrow L.R, L \rightarrow BL \mid B, R \rightarrow BR \mid B, B \rightarrow 0 \mid 1$
- $AS(N) = AS(R) = AS(B) = \{value \uparrow: real\}$ ,  
 $AS(L) = \{length \uparrow: integer, value \uparrow: real\}$ 
  1.  $N \rightarrow L.R \{N.value \uparrow := L.value \uparrow + R.value \uparrow\}$
  2.  $L \rightarrow B \{L.value \uparrow := B.value \uparrow, L.length \uparrow := 1\}$
  3.  $L_1 \rightarrow BL_2 \{L_1.length \uparrow := L_2.length \uparrow + 1;$   
 $L_1.value \uparrow := B.value \uparrow * 2^{L_2.length \uparrow} + L_2.value \uparrow\}$
  4.  $R \rightarrow B \{R.value \uparrow := B.value \uparrow\}$
  5.  $R_1 \rightarrow BR_2 \{R_1.value \uparrow := (B.value \uparrow + R_2.value \uparrow) / 2\}$
  6.  $B \rightarrow 0 \{B.value \uparrow := 0\}$
  7.  $B \rightarrow 1 \{B.value \uparrow := 1\}$

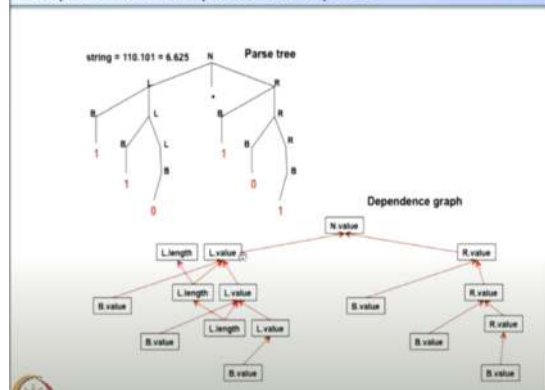
## Mod-04 Lec-13 Análisis Semántico

### Ejemplos

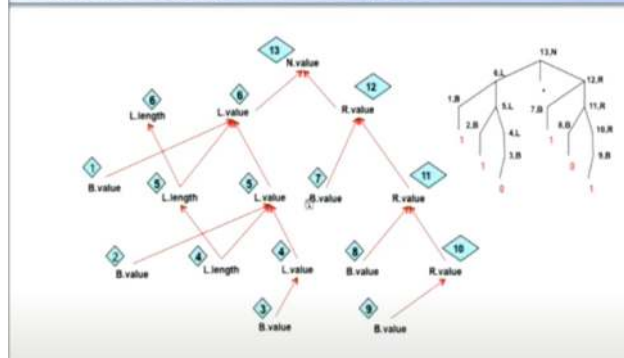
#### Attribute Grammar - Example 2

- AG for the evaluation of a real number from its bit-string representation  
Example:  $110.101 = 6.625$
- $N \rightarrow L.R, L \rightarrow BL \mid B, R \rightarrow BR \mid B, B \rightarrow 0 \mid 1$
- $AS(N) = AS(R) = AS(B) = \{value \uparrow: real\}$ ,  
 $AS(L) = \{length \uparrow: integer, value \uparrow: real\}$ 
  1.  $N \rightarrow L.R \{N.value \uparrow := L.value \uparrow + R.value \uparrow\}$
  2.  $L \rightarrow B \{L.value \uparrow := B.value \uparrow, L.length \uparrow := 1\}$
  3.  $L_1 \rightarrow BL_2 \{L_1.length \uparrow := L_2.length \uparrow + 1;$   
 $L_1.value \uparrow := B.value \uparrow * 2^{L_2.length \uparrow} + L_2.value \uparrow\}$
  4.  $R \rightarrow B \{R.value \uparrow := B.value \uparrow\}$
  5.  $R_1 \rightarrow BR_2 \{R_1.value \uparrow := (B.value \uparrow + R_2.value \uparrow) / 2\}$
  6.  $B \rightarrow 0 \{B.value \uparrow := 0\}$
  7.  $B \rightarrow 1 \{B.value \uparrow := 1\}$

#### Dependence Graph for Example 2



### Attribute Evaluation for Example 2 - 1



### Attribute Grammar - Example 3

- A simple AG for the evaluation of a real number from its bit-string representation

Example:  $110.1010 = 6 + 10/2^4 = 6 + 10/16 = 6 + 0.625 = 6.625$

- $N \rightarrow X.X, X \rightarrow BX \mid B, B \rightarrow 0 \mid 1$

- $AS(N) = AS(B) = \{value \uparrow: real\}$

$AS(X) = \{length \uparrow: integer, value \uparrow: real\}$

- $N \rightarrow X.X \{N.value \uparrow = X_1.value \uparrow + X_2.value \uparrow / 2^{X_1.length}\}$

- $X \rightarrow B \{X.value \uparrow = B.value \uparrow, X.length \uparrow = 1\}$

- $X_1 \rightarrow BX_2 \{X_1.length \uparrow = X_2.length \uparrow + 1; X_1.value \uparrow = B.value \uparrow * 2^{X_2.length} + X_2.value \uparrow\}$

- $B \rightarrow 0 \{B.value \uparrow = 0\}$

- $B \rightarrow 1 \{B.value \uparrow = 1\}$

### Attribute Grammar - Example 4

- An AG for associating type information with names in variable declarations

$AI(L) = AI(ID) = \{type \downarrow: \{integer, real\}\}$

$AS(T) = \{type \uparrow: \{integer, real\}\}$

$AS(ID) = AS(identifier) = \{name \uparrow: string\}$

- $DList \rightarrow D \mid DList ; D$

- $D \rightarrow T L \{L.type \downarrow = T.type \uparrow\}$

- $T \rightarrow int \{T.type \uparrow = integer\}$

- $T \rightarrow float \{T.type \uparrow = real\}$

- $L \rightarrow ID \{ID.type \downarrow = L.type \downarrow\}$

- $L_1 \rightarrow L_2 . ID \{L_2.type \downarrow = L_1.type \downarrow; ID.type \downarrow = L_1.type \downarrow\}$

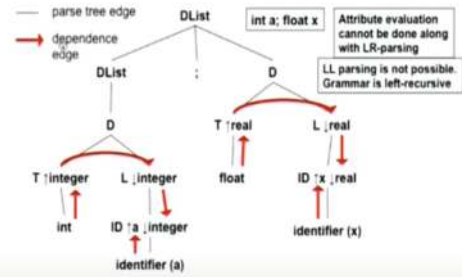
- $ID \rightarrow identifier \{ID.name \uparrow = identifier.name \uparrow\}$

Example: `int a,b,c; float x,y`

a,b, and c are tagged with type *integer*

x,y, and z are tagged with type *real*

### Attribute Evaluation for Example 4



- $DList \rightarrow D \mid DList ; D$
- $D \rightarrow T L \{L.type \downarrow = T.type \uparrow\}$
- $T \rightarrow int \{T.type \uparrow = integer\}$
- $T \rightarrow float \{T.type \uparrow = real\}$
- $L \rightarrow ID \{ID.type \downarrow = L.type \downarrow\}$
- $L_1 \rightarrow L_2 . ID \{L_2.type \downarrow = L_1.type \downarrow; ID.type \downarrow = L_1.type \downarrow\}$
- $ID \rightarrow identifier \{ID.name \uparrow = identifier.name \uparrow\}$

## Gramática de Atributos - Ejemplo 5

- Consideremos primero el CFG para un lenguaje simple

$S \rightarrow E$

$E \rightarrow E + T \mid T \mid \text{let } id = E \text{ in } (E)$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid \text{number} \mid id$





→ Sintetizada, o

→ Heredado, pero con las siguientes limitaciones: considerar una producción  $p: A \rightarrow X_1X_2\dots X_n$ . Let  $X_i.a \in Al(X_i)$ .  $X_i.a$  sólo se puede utilizar.

★ elementos de  $Al(A)$ .

★ elementos de  $Al(X_k)$  or  $AS(X_k)$ ,  $k = 1, \dots, i - 1$  (es decir, los atributos de  $X_1, \dots, X_{i-1}$ )

❖ Nos concentramos en los SAG, y 1 - pasar los LAG, en los que la evaluación de atributos puede combinarse con el análisis de LR, LL o RD.

**Input:** A parse tree  $T$  with unevaluated attribute instances

**Output:**  $T$  with consistent attribute values

void dfvisit( $n$ : node)

{ for each child  $m$  of  $n$ , from left to right do

{ evaluate inherited attributes of  $m$ ;

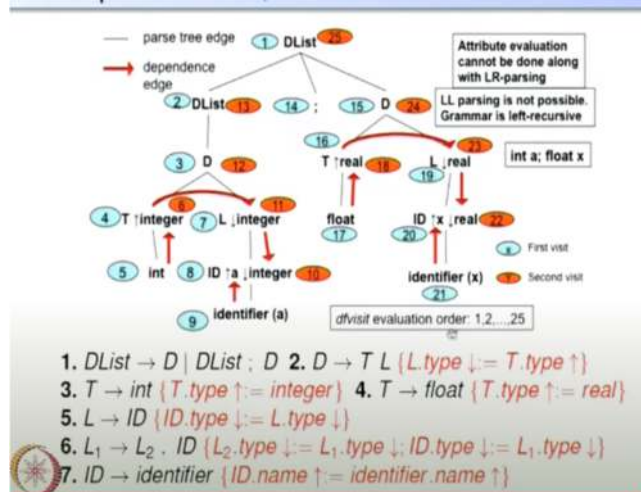
dfvisit( $m$ )

};

evaluate synthesized attributes of  $n$

}

#### Example of LAG - 1, Evaluation Order



## Mod-04 Lec-14 Análisis Semántico

### Gramática de Traducción Atribuida

❖ Aparte de las reglas de cálculo de atributos, se añade al AG algún segmento de programa que realiza el cálculo de salida o de algún otro árbol de efectos secundarios.

- ❖ Ejemplos: operaciones de tablas de símbolos, escritura de código generado en un archivo, etc.
- ❖ Como resultado de estos segmentos de código de acción, las órdenes de evaluación pueden verse restringidas.
- ❖ Tales restricciones se añaden al gráfico de dependencia de atributos como bordes implícitos.
- ❖ Estas acciones pueden ser añadidas tanto a los SAG como a los LAG (haciéndolos, SATG y LATG resp.).
- ❖ Nuestra discusión sobre el análisis semántico utilizará LATG (1 - paso) y SATG.

## Ejemplos:

**Example 1: SATG for Desk Calculator**

```

%%
lines: lines expr '\n' {printf("%g\n", $2);}
      | lines '\n'
      | /* empty */
      ;
expr : expr '+' expr {$$ = $1 + $3;}
/* Same as: expr(1).val = expr(2).val+expr(3).val */
      | expr '-' expr {$$ = $1 - $3;}
      | expr '*' expr {$$ = $1 * $3;}
      | expr '/' expr {$$ = $1 / $3;}
      | '(' expr ')' {$$ = $2;}
      | NUMBER /* type double */
      ;
    
```

**Example 2: SATG for Modified Desk Calculator**

```

%%
lines: lines expr '\n' {printf("%g\n", $2);}
      | lines '\n'
      | /* empty */
      ;
expr : NAME '=' expr {sp = symlook($1);
                      sp->value = $3; $$ = $3;}
      | NAME {sp = symlook($1); $$ = sp->value;}
      | expr '+' expr {$$ = $1 + $3;}
      | expr '-' expr {$$ = $1 - $3;}
      | expr '*' expr {$$ = $1 * $3;}
      | expr '/' expr {$$ = $1 / $3;}
      | '(' expr ')' {$$ = $2;}
      | NUMBER /* type double */
      ;
    
```

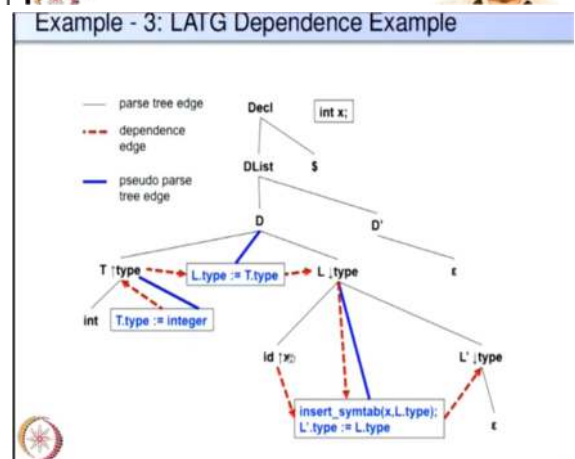
**Example 3: LAG, LATG, and SATG**

LAG (notice the changed grammar)

1.  $Decl \rightarrow DList$  2.  $DList \rightarrow D D'$  3.  $D' \rightarrow \epsilon$ ;  $DList$
4.  $D \rightarrow T L$  ( $L.type \downarrow := T.type \uparrow$ )
5.  $T \rightarrow int$  ( $T.type \uparrow := integer$ ) 6.  $T \rightarrow float$  ( $T.type \uparrow := real$ )
7.  $L \rightarrow ID L'$  ( $ID.type \downarrow := L.type \downarrow$ ;  $L'.type \downarrow := L.type \downarrow$ ;
8.  $L' \rightarrow \epsilon$ ;  $L$  ( $L.type \downarrow := L'.type \downarrow$ ;
9.  $ID \rightarrow identifier$  ( $ID.name \uparrow := identifier.name \uparrow$ )

LATG (notice the changed grammar)

1.  $Decl \rightarrow DList$  2.  $DList \rightarrow D D'$  3.  $D' \rightarrow \epsilon$ ;  $DList$
4.  $D \rightarrow T$  ( $L.type \downarrow := T.type \uparrow$ )  $L$
5.  $T \rightarrow int$  ( $T.type \uparrow := integer$ ) 6.  $T \rightarrow float$  ( $T.type \uparrow := real$ )
7.  $L \rightarrow id$  ( $insert\_syntab(id.name \uparrow, L.type \downarrow)$ ;  
 $L.type \downarrow := L.type \downarrow$ ;  $L'$
8.  $L' \rightarrow \epsilon$ ;  $L$  ( $L.type \downarrow := L'.type \downarrow$ ;  $L$



### Example 3: LAG, LATG, and SATG (contd.)

SATG

1.  $Decl \rightarrow DList\$$
2.  $DList \rightarrow D \mid DList ; D$
3.  $D \rightarrow T L \{ patchtype(T.type \uparrow, L.namelist \uparrow); \}$
4.  $T \rightarrow int \{ T.type \uparrow := integer \}$
5.  $T \rightarrow float \{ T.type \uparrow := real \}$
6.  $L \rightarrow id \{ sp = insert\_symtab(id.name \uparrow);$   
 $L.namelist \uparrow = makelist(sp); \}$
7.  $L_1 \rightarrow L_2, id \{ sp = insert\_symtab(id.name \uparrow);$   
 $L_1.namelist \uparrow = append(L_2.namelist \uparrow, sp); \}$

### Example 4: SATG with Scoped Names

1.  $S \rightarrow E \{ S.val := E.val \}$
2.  $E \rightarrow E + T \{ E(1).val := E(2).val + T.val \}$
3.  $E \rightarrow T \{ E.val := T.val \}$
- /\* The 3 productions below are broken parts  
of the prod.:  $E \rightarrow let\ id = E\ in\ (E) \text{ */}$
4.  $E \rightarrow L B \{ E.val := B.val; \}$
5.  $L \rightarrow let\ id = E \{ //scope\ initialized\ to\ 0;$   
 $scope++; insert(id.name, scope, E.val) \}$
6.  $B \rightarrow in\ (E) \{ B.val := E.val;$   
 $delete\_entries(scope); scope--; \}$
7.  $T \rightarrow T * F \{ T(1).val := T(2).val * F.val \}$
8.  $T \rightarrow F \{ T.val := F.val \}$
9.  $F \rightarrow (E) \{ F.val := E.val \}$
10.  $F \rightarrow number \{ F.val := number.val \}$
11.  $F \rightarrow id \{ F.val := getval(id.name, scope) \}$

## LATG for Sem. Analysis of Variable Declarations - 1

1.  $Decl \rightarrow DList\$$
2.  $DList \rightarrow D \mid D ; DList$
3.  $D \rightarrow T L$
4.  $T \rightarrow int \mid float$
5.  $L \rightarrow ID\_ARR \mid ID\_ARR, L$
6.  $ID\_ARR \rightarrow id \mid id [ DIMLIST ] \mid id BR\_DIMLIST$
7.  $DIMLIST \rightarrow num \mid num, DIMLIST$
8.  $BR\_DIMLIST \rightarrow [ num ] \mid [ num ] BR\_DIMLIST$

Note: array declarations have two possibilities

`int a[10,20,30]; float b[25][35];`

## LATG para Sem. Análisis de declaraciones variables - 2

- ❖ La gramática no es LL(1) y por lo tanto no se puede construir un analizador LL(1) a partir de ella.
- ❖ Asumimos que el árbol de análisis está disponible y que la evaluación de atributos se realiza sobre el árbol de análisis.

- ❖ Las modificaciones del CFG para convertirlo en LL(1) y los correspondientes cambios en el AG se dejan como ejercicios.
- ❖ Los atributos y sus reglas de cálculo para las producciones 1 - 4 son como antes y los ignoramos.
- ❖ Proporcionamos el AG sólo para las producciones 5 - 7; el AG para la regla 8 es similar al de la regla 7.
- ❖ El manejo de las declaraciones constantes es similar al de las declaraciones variables.

#### Identifier type information in the Symbol Table

Identifier type information record

name	type	etype	dimlist_ptr
------	------	-------	-------------

1. type: (simple, array)  
2. type = simple for non-array names  
3. The fields etype and dimlist\_ptr are relevant only for arrays. In that case, type = array  
4. etype: (integer, real, error type), is the type of a simple id or the type of the array element  
5. dimlist\_ptr points to a list of ranges of the dimensions of an array. C-type array declarations are assumed  
Ex. float my\_array[5][12][15]  
dimlist\_ptr points to the list (5, 12, 15), and the total number elements in the array is  $5 \times 12 \times 15 = 900$ , which can be obtained by traversing this list and multiplying the elements.

