

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

Facultad de Ciencias

Integrantes:

Adrián Aguilera Moreno
Sebastián Alejandro Gutierrez Medina



Compiladores

Tarea 05

1. (2pts.) Demuestre que la siguiente gramática pertenece a la clase **LALR** pero no a la clase **SLR**.

$$E \rightarrow Aa \mid bAc \mid dc \mid bda \quad A \rightarrow d$$

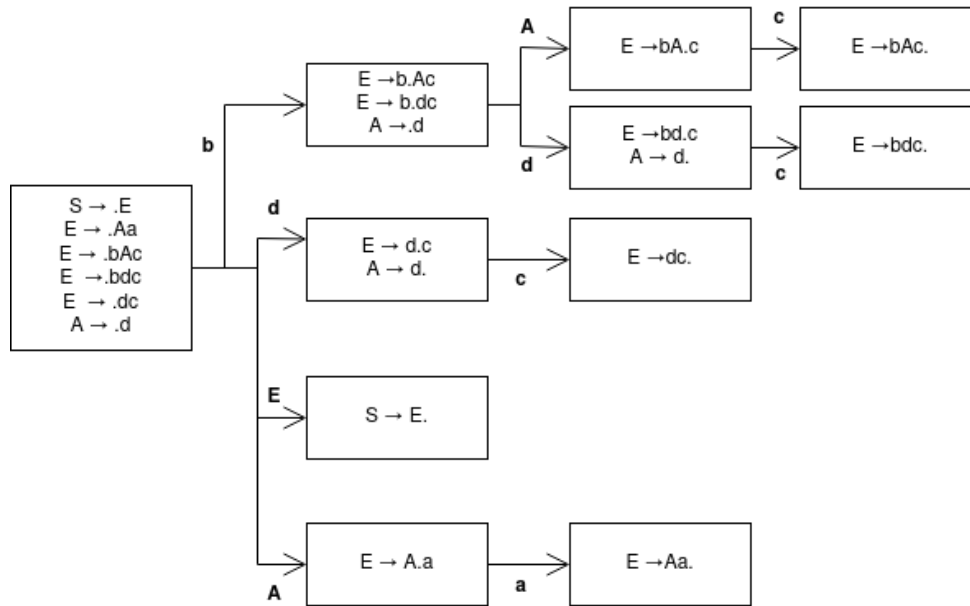
(1pt.) Además analice la cadena bdc mostrando la secuencia de acciones del parser.

Solución: Para este problema, basta dar la tabla **LALR** de la gramática y probar que la gramática es $LR(0)$ con conflictos *shift/reduce*, así es fácil ver que no pertenece a **SLR**.

Primero mostremos que la gramática no pertenece a **SLR**. Observemos que

- (0) $S \rightarrow E$
- (1) $E \rightarrow Aa$
- (2) $E \rightarrow bAc$
- (3) $E \rightarrow bdc$
- (4) $E \rightarrow dc$
- (5) $A \rightarrow d$

Luego, tenemos que **LR(0)** es



dónde I_0 es el estado inicial; I_1 es la transición por E ; I_2 es la transición por A ; I_3 es la transición por b ; I_4 es la transición por d . De igual manera, I_5 es la transición desde I_2 por a ; I_6 es la transición desde I_3 por A ; I_7 es la transición desde I_3 por d ; I_8 es la transición desde I_4 por c ; I_9 es la transición desde I_6 por c ; I_{10} es la transición desde I_7 por c .

Luego, analicemos la tabla siguiente y observemos que existen al menos 2 conflictos *shift/reduce* y por tanto nuestra gramática no corresponde a la clase **SLR**:

Estado	Acción					GoTo	
	a	b	c	d	\$	E	A
0		s3		s4		1	2
1					acc		
2	s5						
3				s7			6
4	r5		s8/r5				
5					r1		
6			s9				
7	r5		s10/r5				
8					r5		
9					r2		
10					r74		

Del análisis **LR(0)** anterior, encontremos el análisis para **LR(1)**:

I_0 :

$$\begin{aligned}
 S &\rightarrow .E, \{a, \$\} \\
 E &\rightarrow .Aa, \{a, \$\} \\
 E &\rightarrow .bAc, \{a, \$\} \\
 E &\rightarrow .dc, \{a, \$\} \\
 E &\rightarrow .bda, \{a, \$\} \\
 A &\rightarrow .d, \{a, \$\}
 \end{aligned}$$

I_1 :

$$\begin{aligned}
 S &\rightarrow E., \{\$ \} \\
 E &\rightarrow A.a, \{a\} \\
 E &\rightarrow .Aa, \{a, \$\} \\
 E &\rightarrow .bAc, \{a, \$\} \\
 E &\rightarrow .dc, \{a, \$\} \\
 E &\rightarrow .bda, \{a, \$\} \\
 A &\rightarrow .d, \{a, \$\}
 \end{aligned}$$

I_2 :

$$\begin{aligned}
 E &\rightarrow b.Ac, \{a, \$\} \\
 E &\rightarrow .bAc, \{a, \$\} \\
 E &\rightarrow .dc, \{a, \$\} \\
 E &\rightarrow .bda, \{a, \$\} \\
 A &\rightarrow .d, \{a, \$\}
 \end{aligned}$$

I_3 :

$$\begin{aligned}
 E &\rightarrow bd.a, \{a, \$\} \\
 E &\rightarrow .bda, \{a, \$\} \\
 A &\rightarrow .d, \{a, \$\}
 \end{aligned}$$

I_4 :

$$\begin{aligned}
 E &\rightarrow bda., \{a, \$\} \\
 A &\rightarrow .d, \{a, \$\}
 \end{aligned}$$

I_5 :

$$\begin{aligned} E &\rightarrow .dc, \{a, \$\} \\ E &\rightarrow .bda, \{a, \$\} \\ A &\rightarrow .d, \{a, \$\} \end{aligned}$$

I_6 :

$$\begin{aligned} E &\rightarrow d.c, \{a, \$\} \\ A &\rightarrow d., \{a, \$\} \end{aligned}$$

I_7 :

$$\begin{aligned} E &\rightarrow dc., \{a, \$\} \\ A &\rightarrow d., \{a, \$\} \end{aligned}$$

Minimizando estados tenemos que

I_1 :

$$\begin{aligned} S &\rightarrow E., \{\$ \} \\ E &\rightarrow A.a, \{a\} \\ E &\rightarrow .Aa, \{a, \$\} \\ E &\rightarrow .bAc, \{a, \$\} \\ E &\rightarrow .dc, \{a, \$\} \\ E &\rightarrow .bda, \{a, \$\} \\ A &\rightarrow .d, \{a, \$\} \end{aligned}$$

I_2 :

$$E \rightarrow b.Ac, \{a, \$\}$$

I_3 :

$$E \rightarrow bd.a, \{a, \$\}$$

I_4 :

$$E \rightarrow bda., \{a, \$\}$$

I_5 :

$$\begin{aligned} E &\rightarrow d.c, \{a, \$\} \\ A &\rightarrow d., \{a, \$\} \end{aligned}$$

Por lo anteriores pasos, es suficiente para probar que (en efecto) nuestra gramática pertenece a **LALR**.

Por último, veamos que la cadena bdc es producida por la gramática en el parser **LALR** como se muestra a continuación:

$$\begin{aligned} S &\rightarrow E \\ &\rightarrow bAc \\ &\rightarrow bdc. \end{aligned}$$

2. (2.5pts.) La siguiente gramática genera expresiones en notación polaca inversa, es decir los argumentos preceden al operador:

$$E \rightarrow E E op \mid id \quad op \rightarrow + \mid - \mid * \mid /$$

Suponer que cada *id* (identificadores en mayúsculas) tiene un atributo sintético **name** que es una cadena y los símbolos *E* y *op* tienen un atributo **val** que también es una cadena.

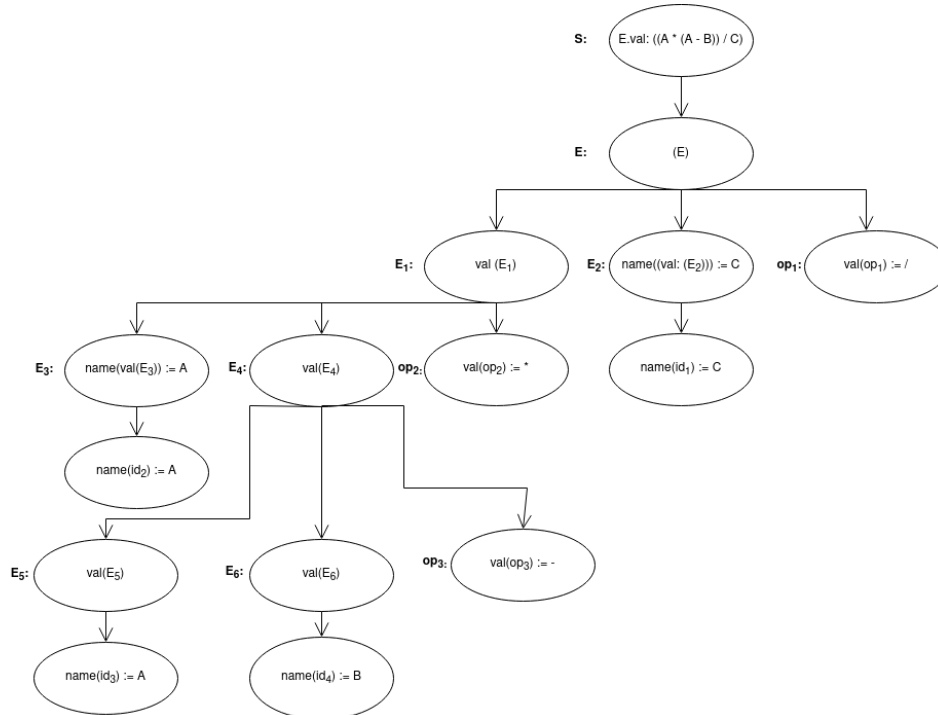
Diseña una gramática con atributos para organizar el atributo **val** de la raíz del parse tree para guardar la traducción de la expresión en notación infija (utiliza los paréntesis necesarios). Explica la idea que usas para definir las funciones semánticas.

Por ejemplo, si las hojas del parse tree (de izquierda a derecha) son *A A B - * C /* entonces la raíz debe tener como atributo **val** la cadena $((A * (A - B))/C)$.

Solución:

Producciones	Reglas Semánticas
$S \rightarrow (E)$	$S.val := E.val$
$E \rightarrow ((E_1) (E_2) op)$	$E.val := E_1.val op E_2.val$
$E \rightarrow id$	$E.name := id.name$
$op \rightarrow +$	$op.val := +$
$op \rightarrow -$	$op.val := -$
$op \rightarrow *$	$op.val := *$
$op \rightarrow /$	$op.val := /$

A continuación se muestra el árbol para organizar **val** respecto a $((A(AB-)*C)/)$:



Cómo podemos notar, $S \rightarrow E = ((A * (A - B))/C)$ que es generado por las reglas semánticas definidas. La idea es ir aplicando recursión de hojas a raíz, y siguiendo las reglas de semánticas. De esta manera pasamos de notación polaca inversa a sufija.

3. (2.5pts.) Extender la siguiente gramática con atributos para la regla $E \rightarrow E_1 * E_2$ y obtener el código de tres direcciones para la expresión $x = a[i][j] * b[i][j]$ donde a y b son arreglos de tamaño 2×3 y 2×2 respectivamente y cada uno de ellos almacena enteros cuyo tamaño es 4.

$S \rightarrow \text{id} = E ;$	$\{ \text{gen}(top.get(\text{id.lexeme}))' = E.addr; \}$
$ L = E ;$	$\{ \text{gen}(L.array.base'[L.addr]') = E.addr; \}$
$E \rightarrow E_1 + E_2$	$\{ E.addr = \text{new Temp}();$ $\text{gen}(E.addr' = E_1.addr' + E_2.addr); \}$
$ \text{id}$	$\{ E.addr = top.get(\text{id.lexeme}); \}$
$ L$	$\{ E.addr = \text{new Temp}();$ $\text{gen}(E.addr' = L.array.base'[L.addr]'); \}$
$L \rightarrow \text{id} [E]$	$\{ L.array = top.get(\text{id.lexeme});$ $L.type = L.array.type.elem;$ $L.addr = \text{new Temp}();$ $\text{gen}(L.addr' = E.addr' * L.type.width); \}$
$L \rightarrow$	$L_1 [E] \{ L.array = L_1.array;$ $L.type = L_1.type.elem;$ $t = \text{new Temp}();$ $L.addr = \text{new Temp}();$ $\text{gen}(t' = E.addr' * L.type.width);$ $\text{gen}(L.addr' = L_1.addr' + t); \}$

Table 1: Gramatica Extendida

$S \rightarrow id = E;$	$\{ \text{gen}(top.get(id.lexeme))' = E.addr; \}$
$ L = E;$	$\{ \text{gen}(L.array.base'[L.addr]') = E.addr; \}$
$E \rightarrow E_1 + E_2$	$\{ E.addr = \text{new Temp}();$ $\text{gen}(E.addr' = E_1.addr' + E_2.addr); \}$
$ E_1 * E_2$	$\{ E.addr = \text{new Temp}();$ $\text{gen}(E.addr' = E_1.addr' * E_2.addr); \}$
$ id$	$\{ E.addr = top.get(id.lexeme); \}$
$ L$	$\{ E.addr = \text{new Temp}();$ $\text{gen}(E.addr' = L.array.base'[L.addr]'); \}$
$L \rightarrow id[E]$	$\{ L.array = top.get(id.lexeme);$ $L.type = L.array.type.elem;$ $L.addr = \text{new Temp}();$ $\text{gen}(L.addr' = E.addr' * L.type.width); \}$
$ L_1[E]$	$\{ L.array = L_1.array;$ $L.type = L_1.array.type.elem;$ $t = \text{new Temp}();$ $\text{gen}(t' = E.addr' * L.type.width);$ $\text{gen}(L.addr' = L_1.addr' + t); \}$

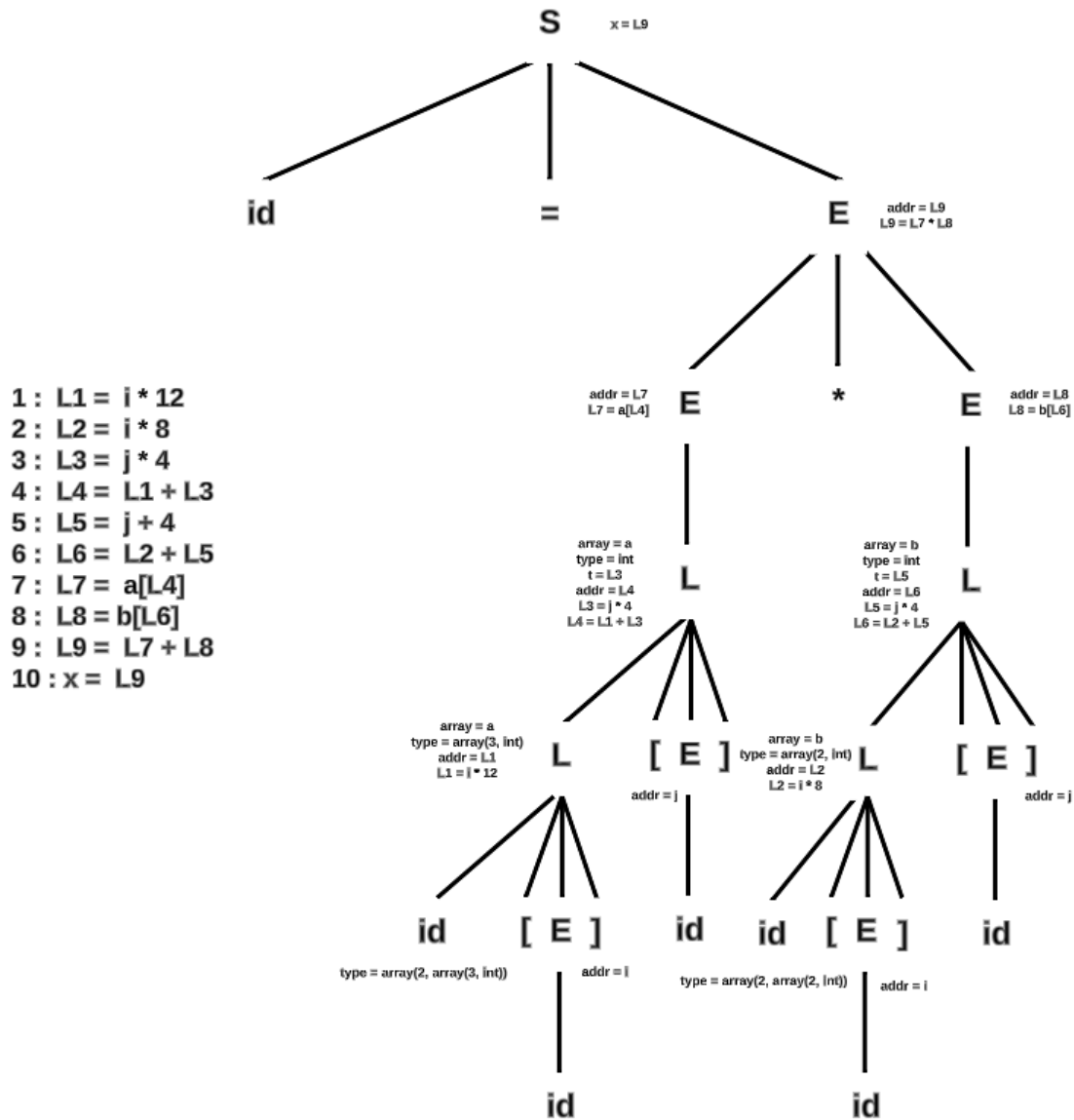


Figure 1: Código tres direcciones

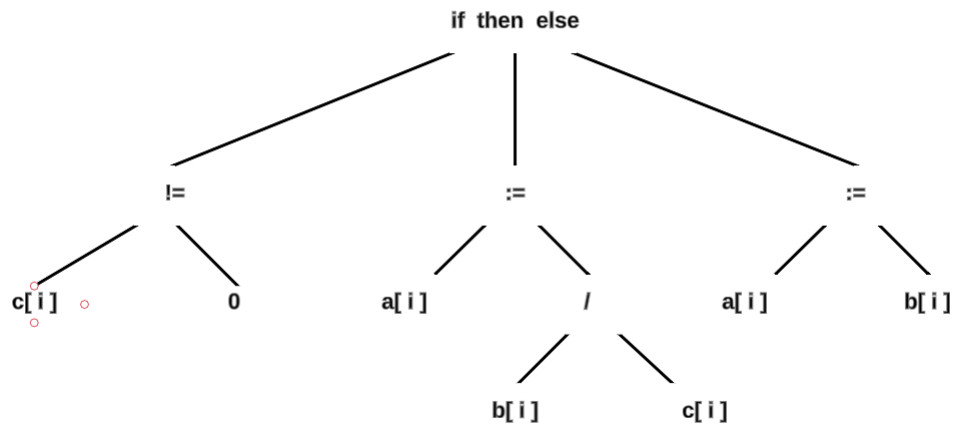
4. (2pts.) Considera el siguiente fragmento de código:

```
if ( c[i] != 0 )
then
  a[i] := b[i] / c[i];
else
  a[i] := b[i];
```

Obtener las representaciones intermedias correspondientes a 1) árbol de sintaxis abstracta; 2) gráfica de control de flujo y 3) código de tres direcciones. Explica tus respuestas.

Discutir (ampliamente) las ventajas que consideras para cada representación.

Arbol de Sintaxis Abstracta

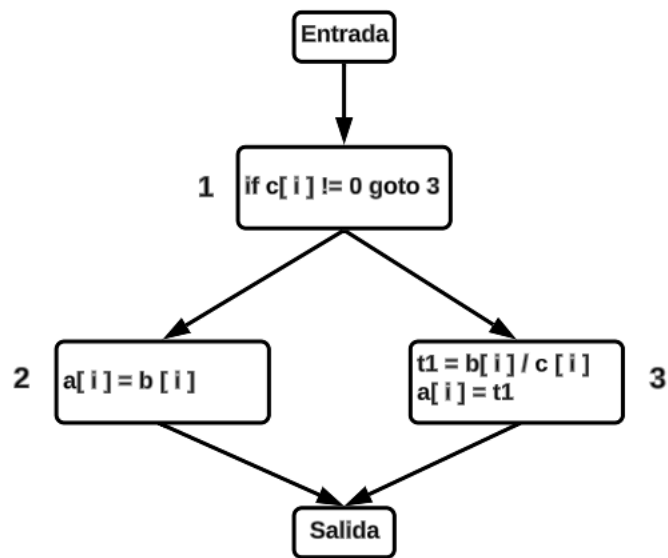


Grafica de flujo

Codigo de Tres Direcciones

```

1 : if c[i] != 0 goto 3
2 : a[i] = b[i]
3 : t1 = b[i] / c[i]
4 : a[i] = t1
  
```



Árbol de sintaxis abstracta

1. Representación estructural: Genera una representación estructural del código, capturando la jerarquía y las relaciones entre diferentes construcciones del lenguaje. Preserva la estructura sintáctica y semántica del código.
2. Independencia del lenguaje: Es independiente del lenguaje y se puede utilizar como una representación intermedia en compiladores para varios lenguajes de programación.
3. Facilidad de manipulación: Permite recorrer y manipular fácilmente mediante algoritmos y transformaciones. Facilita el análisis, optimización y transformación del código fuente.
4. Detección de errores: Ayuda a detectar y reportar errores de sintaxis, errores de tipo y otros problemas semánticos durante el proceso de compilación.

Gráficos de flujo

1. Análisis de flujo de control: Proporciona una representación clara y visual del flujo de control dentro de un programa. Ayuda a analizar las rutas de ejecución del programa, detectar bucles, identificar código inaccesible y realizar transformaciones de programa relacionadas con el flujo de control.
2. Optimización de rendimiento: Puede ser utilizado para optimizar el código, como identificar oportunidades para desenrollar bucles, fusionar bucles o mover código invariante al bucle.
3. Depuración y perfilado: Ayuda en la depuración y el perfilado del programa al proporcionar información sobre el flujo de control, lo que permite comprender el comportamiento del programa e identificar cuellos de botella de rendimiento.

Código de tres direcciones

1. Representación de bajo nivel: Es una representación de bajo nivel que simplifica construcciones complejas de lenguajes de alto nivel en operaciones básicas con direcciones explícitas. Abstrae los detalles específicos del lenguaje y se centra en operaciones esenciales.
2. Optimización de código: También sirve como una representación intermedia que permite diversas optimizaciones de código, como la reducción de constantes, la eliminación de subexpresiones comunes y la asignación de registros.
3. Generación de código: Puede ser utilizada como entrada para generar código máquina, apuntando a arquitecturas específicas o máquinas virtuales.
4. Modularidad: Facilita transformaciones y optimizaciones modulares en operaciones individuales o bloques de código sin afectar la estructura completa del programa.

5. (Hasta 1.5pt extra). Explica lo que es una gráfica de control de flujo. ¿Quién fue Frances Allen?