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**.

$$E \to Aa \mid bAc \mid dc \mid bda \qquad A \to d$$

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

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

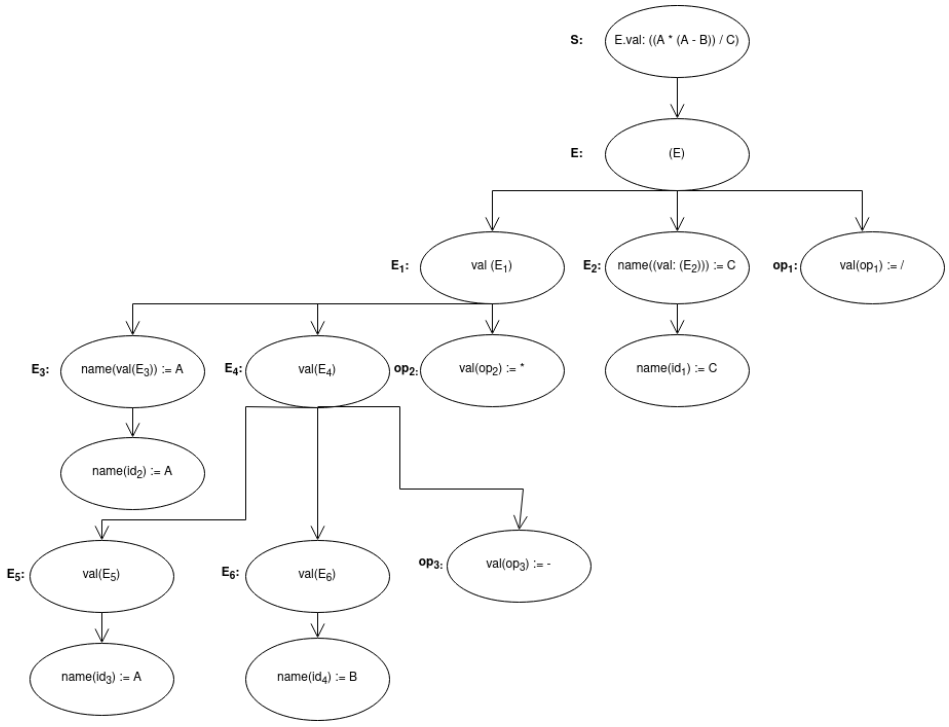
$$E \to E\ E\ op \mid id \qquad\qquad op \to + \mid - \mid * \mid /$$

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
$S \to (E)$	S.`val` := E.`val`
$E \to ((E_1)\ (E_2)\ op)$	E.`val` := E_1.`val` op E_2.`val`
$E \to id$	E.`name` := id.`name`
$op \to +$	op.`val` := $+$
$op \to -$	op.`val` := $-$
$op \to *$	op.`val` := $*$
$op \to /$	op.`val` := $/$

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



Cómo podemos notar, $S \to E = ((A * (A - B))/C)$ 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.

$S \rightarrow$	**id** = E ;	{ $gen($ $top.get($**id**$.lexeme)$ $'='$ $E.addr)$; }
|	L = E ;	{ $gen(L.array.base$ $'['$ $L.addr$ $']'$ $'='$ $E.addr)$; }
$E \rightarrow$	E_1 + E_2	{ $E.addr =$ **new** $Temp()$; $gen(E.addr$ $'='$ $E_1.addr$ $'+'$ $E_2.addr)$; }
|	**id**	{ $E.addr = top.get($**id**$.lexeme)$; }
|	L	{ $E.addr =$ **new** $Temp()$; $gen(E.addr$ $'='$ $L.array.base$ $'['$ $L.addr$ $']')$; }
$L \rightarrow$	**id** [E]	{ $L.array = top.get($**id**$.lexeme)$; $L.type = L.array.type.elem$; $L.addr =$ **new** $Temp()$; $gen(L.addr$ $'='$ $E.addr$ $'*'$ $L.type.width)$; }

$L \rightarrow$	L_1 [E]	{ $L.array = L_1.array$; $L.type = L_1.type.elem$; $t =$ **new** $Temp()$; $L.addr =$ **new** $Temp()$; $gen(t$ $'='$ $E.addr$ $'*'$ $L.type.width)$; $gen(L.addr$ $'='$ $L_1.addr$ $'+'$ $t)$; }

Table 1: Gramatica Extendida

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

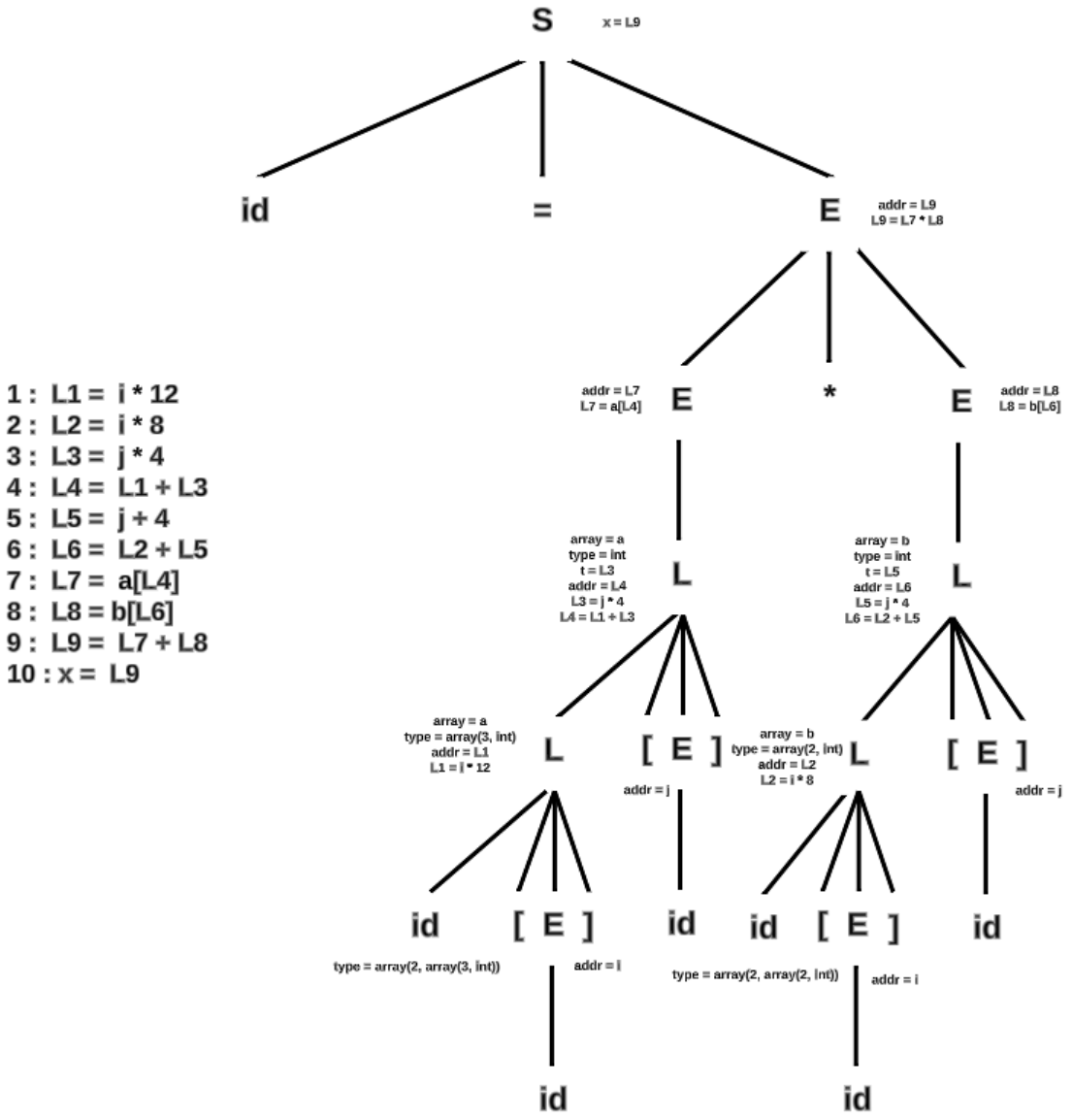
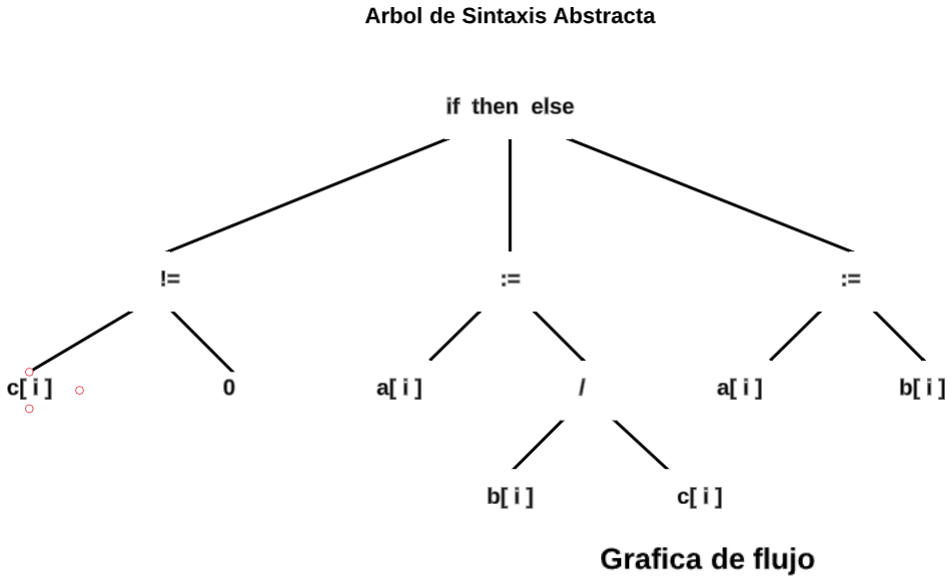


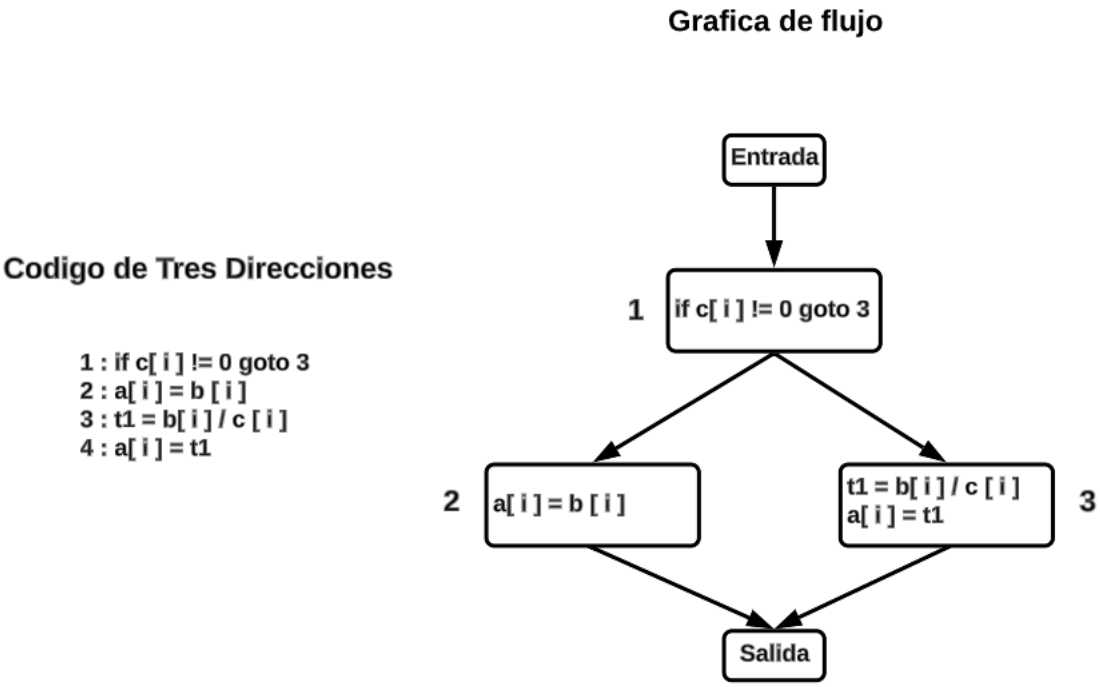
Figure 1: Codigo tres direcciones

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.





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?