



Universidad Nacional Autónoma de México

FACULTAD DE CIENCIAS

TAREA 05

Compiladores

Bonilla Reyes Dafne
Castañón Maldonado Carlos Emilio
García Ponce José Camilo
Velasco García Jorge Daniel

Profesora: Lourdes del Carmen González Huesca

Ayudante: Fausto David Hernández Jasso
Ayudante: Juan Alfonso Garduño Solís
Ayudante: Juan Pablo Yamamoto Zazueta

1. (2 pts) Considera la siguiente gramática para expresiones aritméticas y que construye un árbol sintáctico usando los atributos st y ptr que son apuntadores a nodos de un árbol sintáctico.

1.	$E \rightarrow T \text{ } TT$	$\text{TT}.st := T.ptr$ $E.ptr := TT.ptr$
2.	$TT_1 \rightarrow + \text{ } T \text{ } TT_2$	$TT_2.st := \text{make_bin_op}("+", TT_1.st, T.ptr)$ $TT_1.ptr := TT_2.ptr$
3.	$TT_1 \rightarrow - \text{ } T \text{ } TT_2$	$TT_2.st := \text{make_bin_op}("-", TT_1.st, T.ptr)$ $TT_1.ptr := TT_2.ptr$
4.	$TT \rightarrow \epsilon$	$TT.ptr := TT.st$
5.	$T \rightarrow F \text{ } FT$	$FT.st := F.ptr$ $T.ptr := FT.ptr$
6.	$FT_1 \rightarrow * \text{ } F \text{ } FT_2$	$FT_2.st := \text{make_bin_op}("*", FT_1.st, F.ptr)$ $FT_1.ptr := FT_2.ptr$
7.	$FT_1 \rightarrow / \text{ } F \text{ } FT_2$	$FT_2.st := \text{make_bin_op}("/", FT_1.st, F.ptr)$ $FT_1.ptr := FT_2.ptr$
8.	$FT \rightarrow \epsilon$	$FT.ptr := FT.st$
9.	$F_1 \rightarrow - \text{ } F_2$	$FT.ptr := \text{make_un_op}("-", FT_2.ptr)$
10.	$F \rightarrow (E)$	$F.ptr := E.ptr$
11.	$F \rightarrow \text{const}$	$F.ptr := \text{make_leaf}(\text{const.val})$

Muestra los pasos para obtener un árbol de sintaxis para la expresión $(4/2 * 5)$ indicando los subárboles de parsing y los atributos de cada nodo, usando flechas para indicar los apuntadores que construyen el árbol.

Para realizar el árbol de sintaxis, primero consideremos la siguiente nomenclatura:

- \square = ptr
- \circlearrowleft = st

De esta forma, construyamos el árbol general:

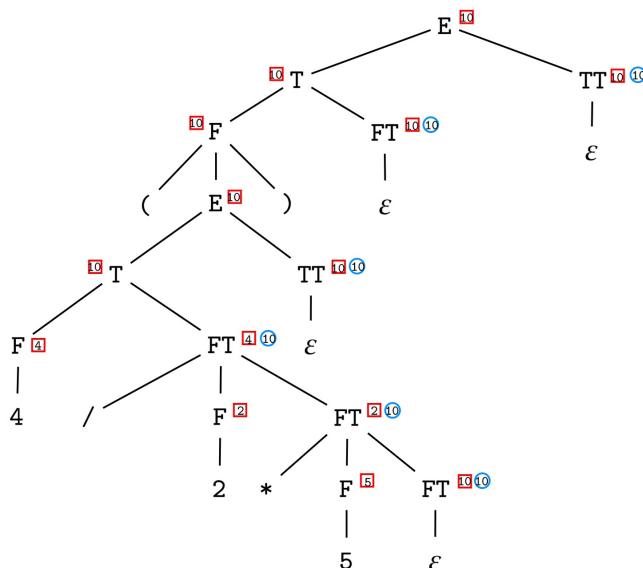


Figura 1. Árbol de Sintaxis General

Notemos que este árbol contiene el cálculo de la expresión, cuyo resultado final es **10**, que también es el resultado de la operación $(4/2 * 5)$, por lo que podemos decir que el árbol se creó correctamente. Ahora bien, mostremos los subárboles de parsing y los atributos de cada nodo, incluyendo también las flechas que indican a los apuntadores. Para el primer subárbol tenemos entonces:

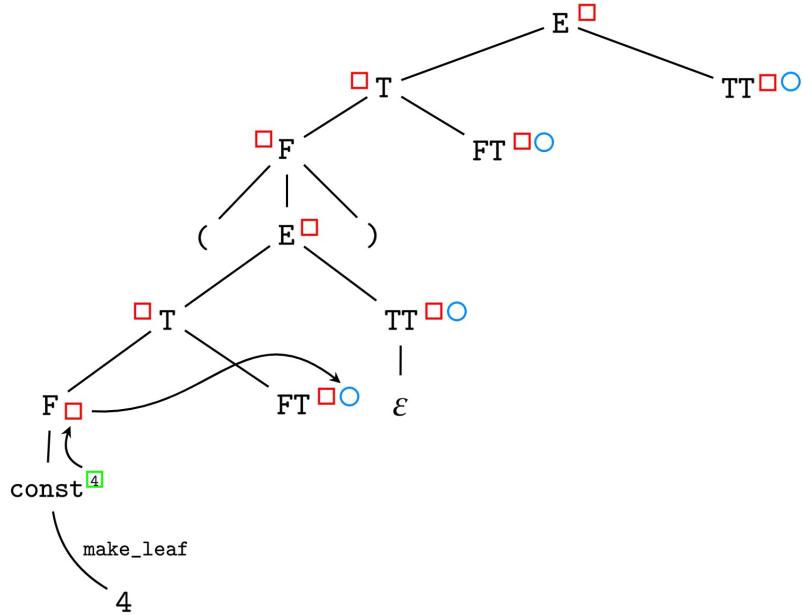


Figura 2. Subárbol de Sintaxis 1

Ahora, mostremos el subárbol de parsing 2:

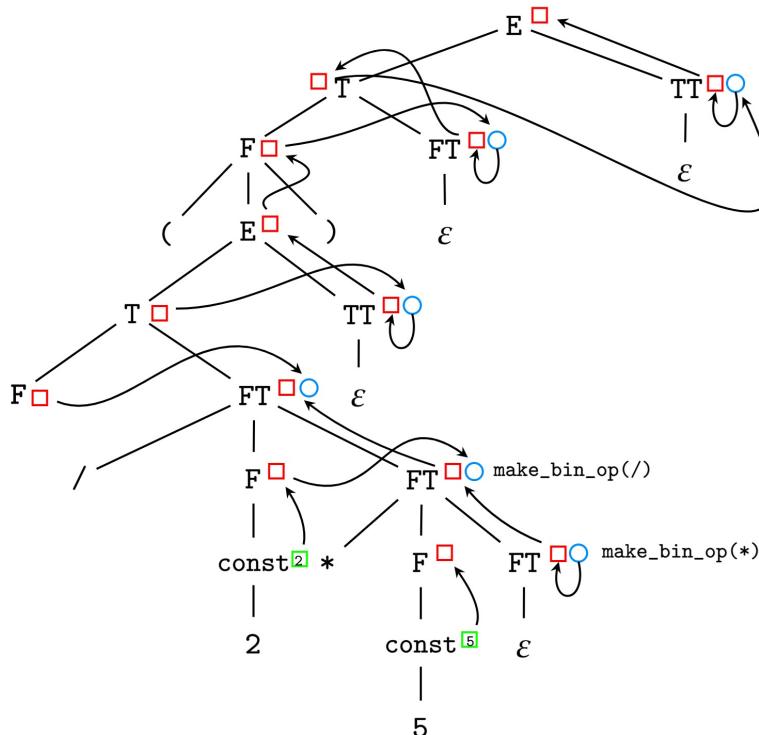


Figura 3. Subárbol de Sintaxis 2

Teniendo los 2 anteriores en cuenta, veamos como queda el árbol de sintaxis completo:

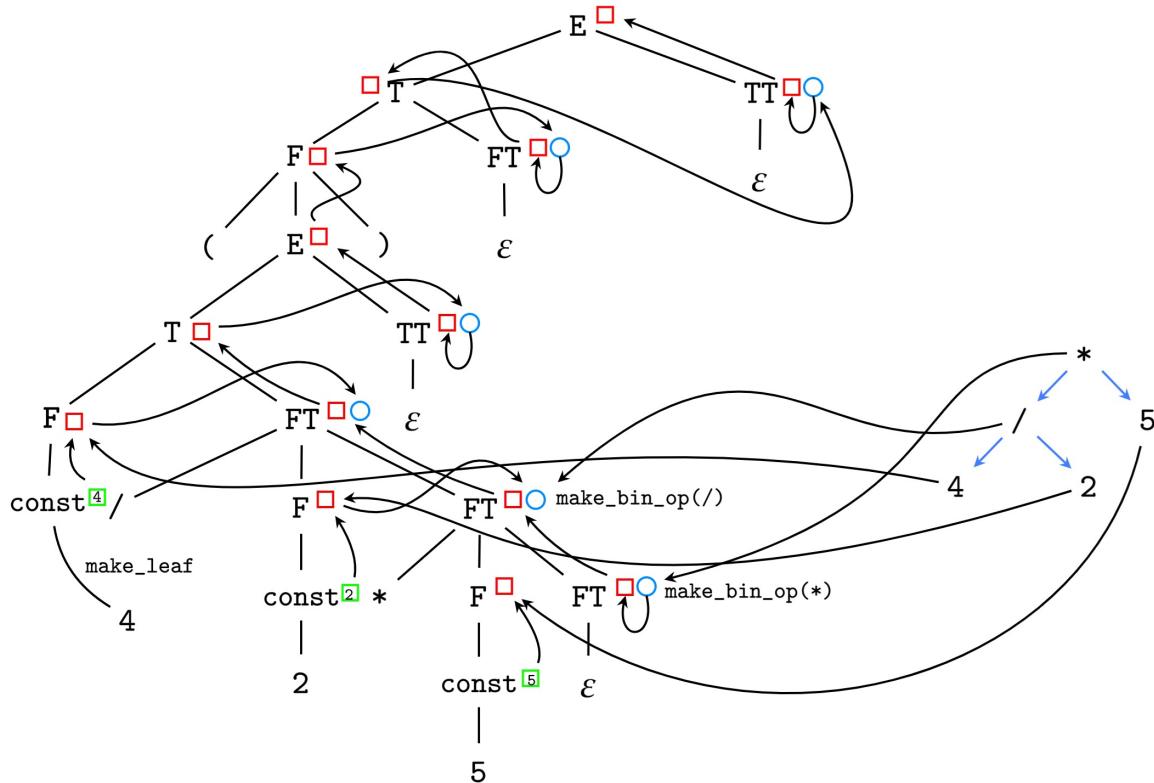


Figura 4. Árbol de Sintaxis Final

2. Considera la siguiente gramática cuyas funciones semánticas permiten obtener tamaños de tipos básicos y arreglos:

$T \rightarrow BC$	$t = B.\text{type}$ $w = B.\text{width}$
	$T.\text{type} = C.\text{type}$ $T.\text{width} = C.\text{width}$
$B \rightarrow \text{int}$	$B.\text{type} = \text{integer}$
	$B.\text{width} = 4$
$B \rightarrow \text{float}$	$B.\text{type} = \text{float}$
	$B.\text{width} = 8$
$C \rightarrow \epsilon$	$C.\text{type} = t$
	$C.\text{width} = w$
$C \rightarrow [\text{num}]C_1$	$C.\text{type} = \text{array}(\text{num.value}, C_1.\text{type})$
	$C.\text{width} = \text{num.value} \times C_1.\text{width}$

Los atributos `type` y `width` de los símbolos no-terminales son sintetizados mientras que las variables `t` y `w` se usan para pasar la información del tipo y el tamaño (en bytes) de un nodo `B` al nodo de la producción `C → ε` en el árbol de sintaxis concreta. Estos últimos atributos son heredados para la variable `C`.

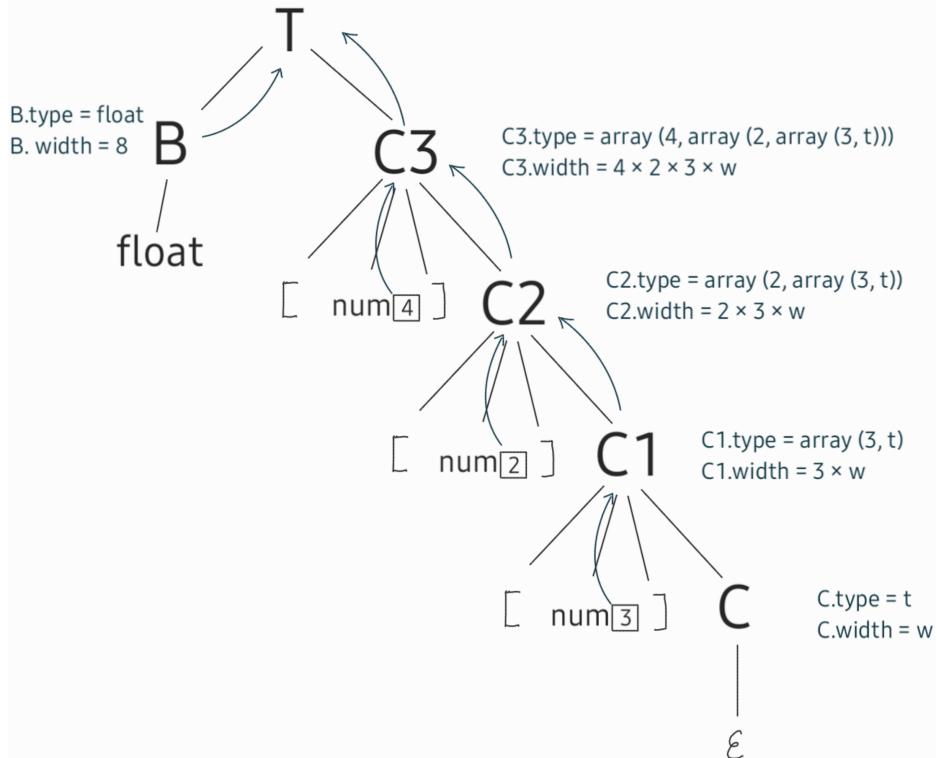


(a) Describe con palabras qué realizan las funciones semánticas para el resto de las reglas.

- $t = B.type$ obtiene el tipo del símbolo no terminal B
- $w = B.width$ obtiene el tamaño de tipo del símbolo no terminal B
- $T.type = C.type$ asigna el tipo del no-terminal T al tipo del no-terminal C
- $T.width = C.width$ asigna el tamaño de tipo del no-terminal T al tamaño de tipo del no-terminal C
- $B.type = \text{integer}$ asigna el tipo del no terminal B a entero
- $B.width = 4$ asigna el tamaño de tipo del no terminal B a 4 (bytes)
- $B.type = \text{float}$ asigna el tipo del no terminal B a flotante
- $B.width = 8$ asigna el tamaño de tipo del no terminal B a 8 (bytes)
- $C.type = t$ asigna el tipo del no terminal C a t definido anteriormente
- $C.width = w$ asigna el tamaño de tipo del no terminal C a w definido anteriormente
- $C.type = \text{array}(\text{num.value}, C_1.type)$ asigna el tipo del no terminal T como un arreglo que define un tamaño de tipo num y otro arreglo (no terminal C_1) del mismo tipo.
- $C.width = \text{num.value} \times C_1.width$ asigna el tamaño de tipo del no terminal C como el tamaño de un arreglo multiplicado por el tamaño del no terminal C_1

(b) Muestra el árbol de sintaxis decorado para la expresión $\text{float}[4][2][3]$ indicando el flujo del cálculo de los atributos usando flechas en el árbol.

```
t = float
w = 8
T.type = array (4, array (2, array (3, float)))
T.width = 4 × 2 × 3 × 8
```

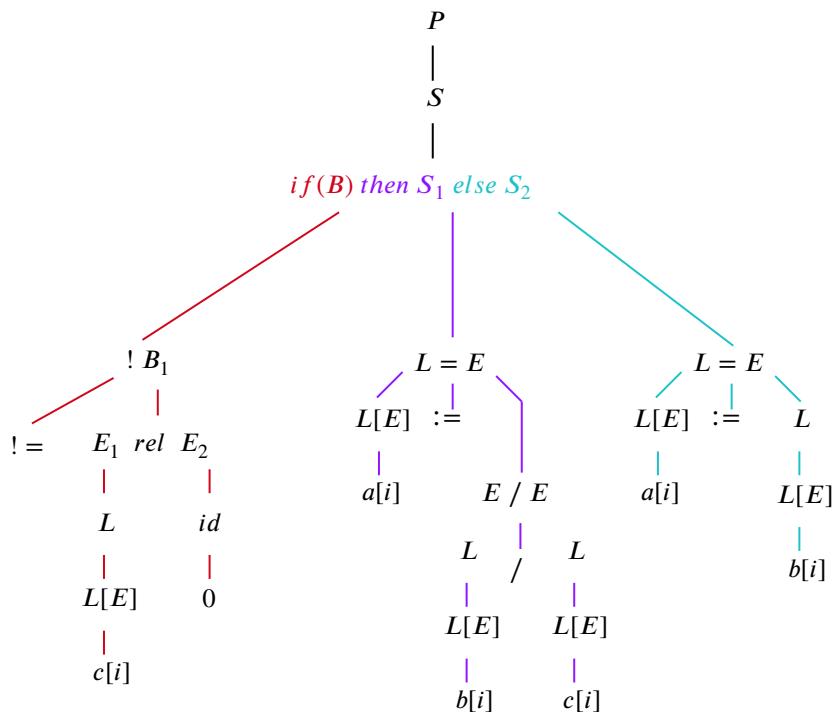


3. (2.5pts.) Considera el siguiente fragmento de código:

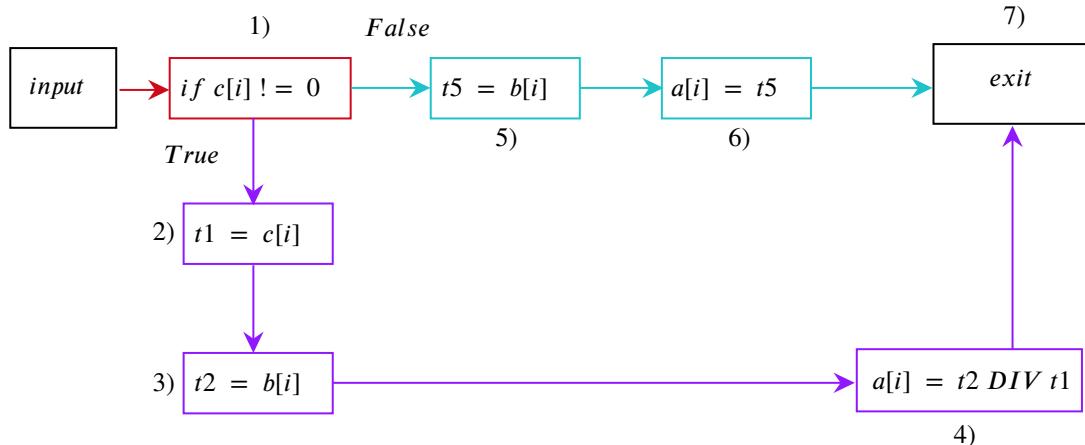
```
if ( c[i] != 0 )
then
    a[i] := b[i] / c[i];
else
    a[i] := b[i];
```

Obtener las representaciones intermedias correspondientes a 1) árbol de sintaxis abstracta; 2) gráfica de control de flujo y 3) código de tres direcciones. Discutir (ampliamente) las ventajas de cada representación considerando las observaciones comentadas en clase.

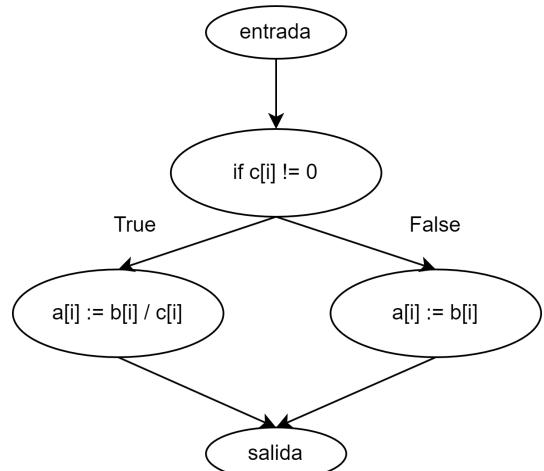
(a) Árbol de sintaxis abstracta



Notemos como es que el arbol nos ofrece una representación abstracta y estructurada del código fuente, lo que permite al compilador centrarse en aspectos semánticos y estructurales del programa, en lugar de preocuparse por detalles de implementación específicos de un lenguaje, además de que los arboles incluyen información semántica sobre el programa, como el tipo de datos de las variables y las relaciones entre las diferentes partes del código, esto facilita la realización de análisis semántico, como la verificación de tipos y la detección de errores semánticos.

(b) Gráfica de control de flujo
Vers - 1:


Notemos como es que lo anterior proporciona una representación visual clara y comprensible del flujo de control dentro del programa, además de que las gráficas de control de flujo pueden ser utilizadas para realizar análisis estático del código, como la detección de puntos inalcanzables, la identificación de bucles y la verificación de la estructura del programa, esto es útil en la detección de errores.

Vers - 2:

(c) Código de tres direcciones

```

(1) t1 = c[i]
(2) if t1 == 0 goto (8)
(3) t2 = b[i]
(4) t3 = c[i]
(5) t4 = t2 DIV t3
(6) a[i] = t4
(7) goto (10)
(8) t5 = b[i]
(9) a[i] = t5
(10) exit
    
```

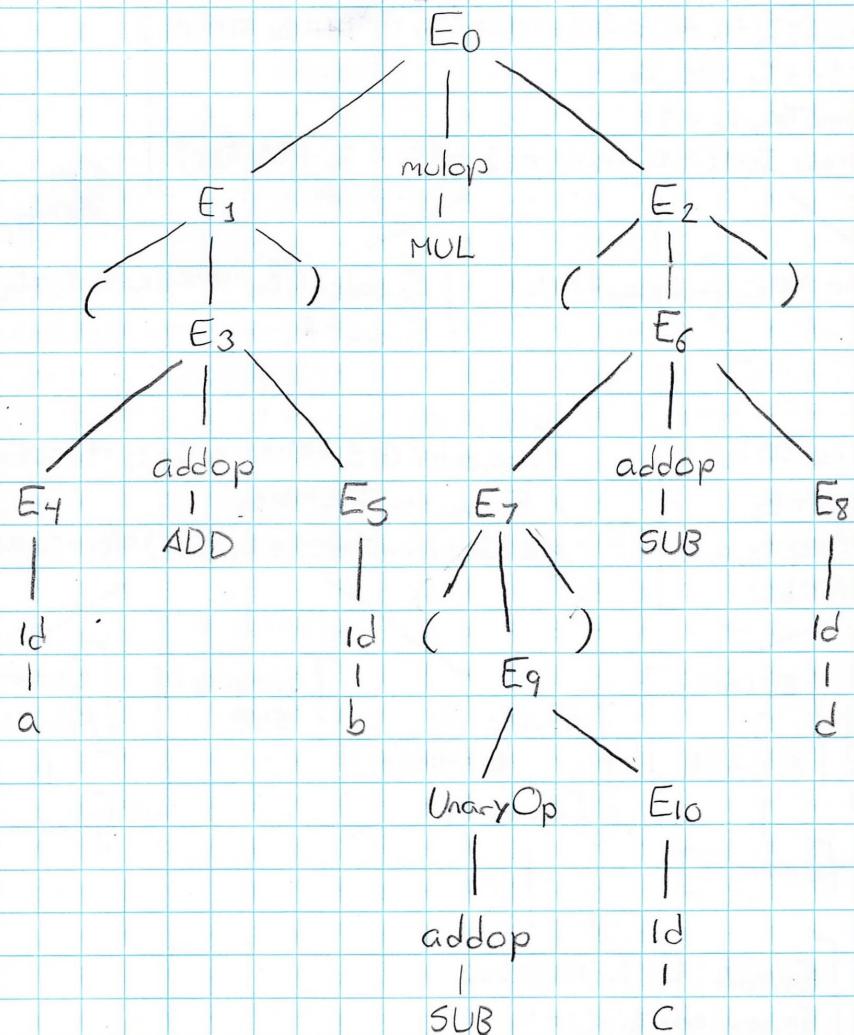
Entre sus múltiples ventajas podemos encontrar que es una representación intermedia del código fuente que abstracta los detalles específicos de la arquitectura de la máquina objetivo, por ende, permite que el compilador se enfoque en la lógica del programa y las optimizaciones sin verse afectado por las peculiaridades de la arquitectura subyacente, además de que esta al ser una representación flexible puede ser extendida según las necesidades del compilador, esto nos permite implementar características avanzadas, como la generación de código intermedio para optimizaciones avanzadas o la integración de características de lenguaje específicas.



4. (2.5pts.) Determina el código de tres direcciones de la siguiente expresión usando las reglas semánticas en las figuras.

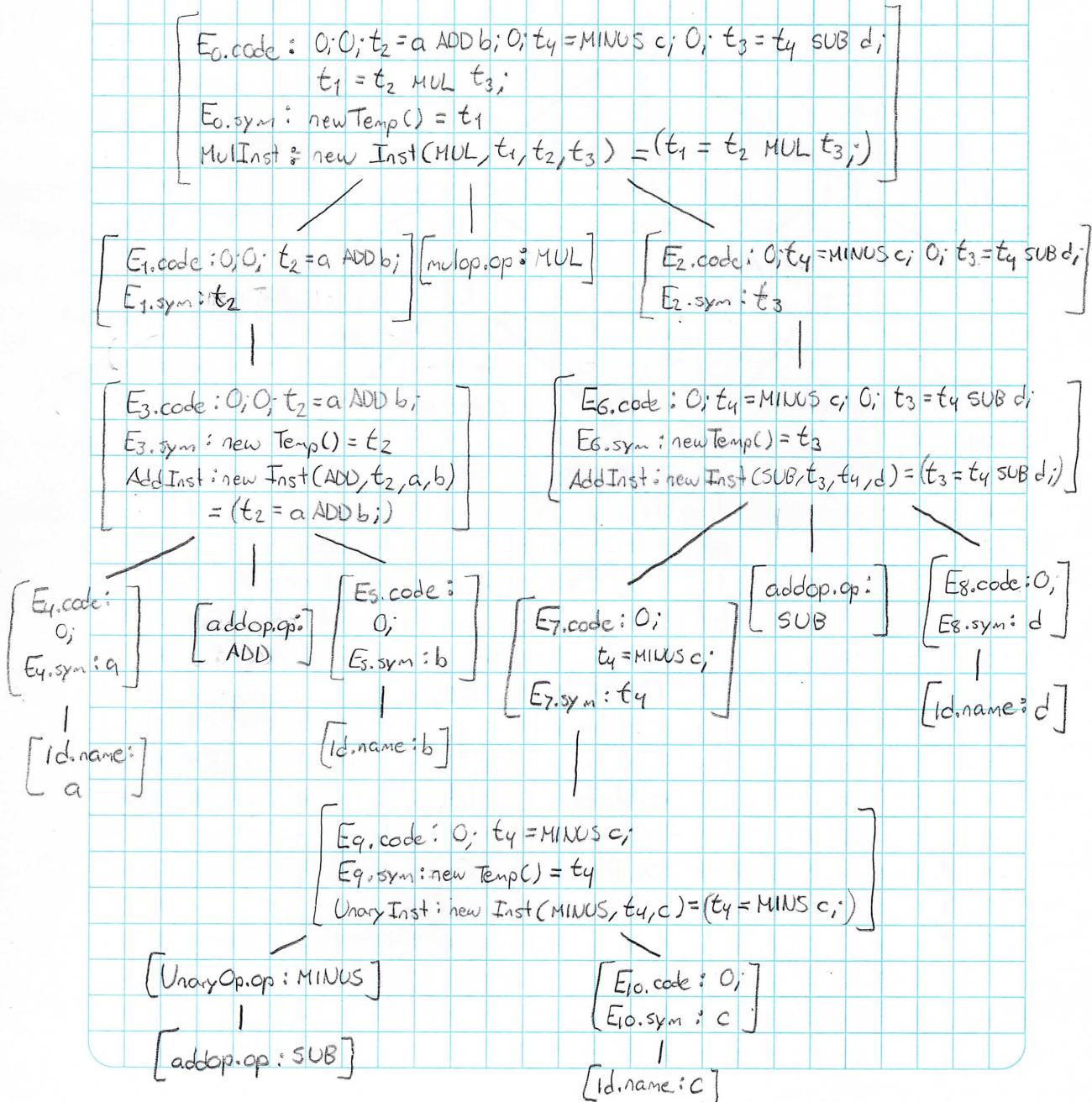
(*a ADD b) MUL ((MINUS c) SUB d)*)

primero hacemos el arbol de la expresión





ahora ponemos los atributos sintetizados, obteniendo este arbol





entonces el código de 3 direcciones es Eo.code

el cual es

- (1) $0;$
- (2) $0;$
- (3) $t_2 = a \text{ ADD } b;$
- (4) $t_4 = \text{MINUS } c;$
- (5) $0;$
- (6) $t_3 = t_4 \text{ SUB } d;$
- (7) $t_1 = t_2 \text{ MUL } t_3;$

y quitando las líneas con $0;$ (no hacen nada), nos queda

- (1) $t_2 = a \text{ ADD } b;$
- (2) $t_4 = \text{MINUS } c;$
- (3) $t_3 = t_4 \text{ SUB } d;$
- (4) $t_1 = t_2 \text{ MUL } t_3$