Universidad Nacional Autónoma de México

Facultad de Ciencias

Compiladores

Tarea 5:

Autores:

Escamilla Soto Cristopher Alejandro
Montiel Manriquez Ricardo

20 de Mayo del 2022

1. La siguiente gramática genera expresiones en notación polaca inversa, es decir, los argumentos preceden al operador.

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

Supón 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.

a) Diseña una gramática con atributos para organizar el atributo val de la raíz del parse tree de tal forma que guarde la traducción de la expresión en notación inflija (utiliza los paréntesis necesarios).

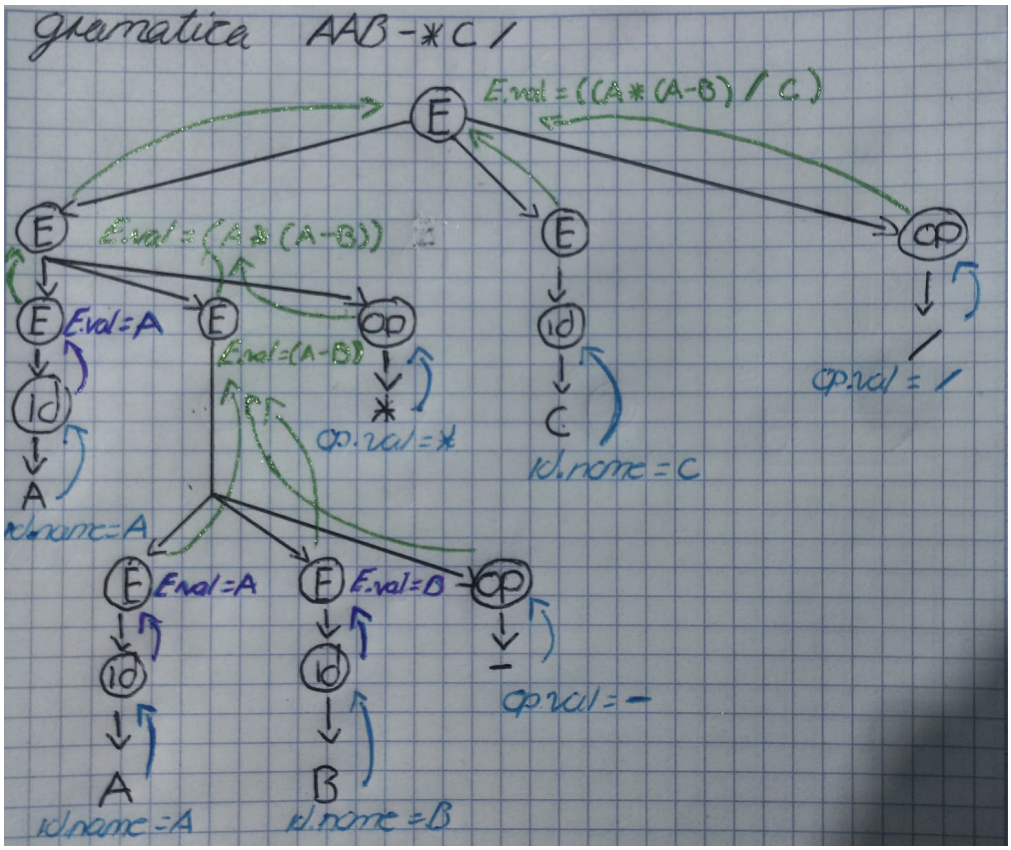
Es fácil crear las reglas de producción semántica para la gramática dada porque, un que se usa la notación Polaca para generar el lenguaje a través de las reglas semánticas podemos generar en valor de la raíz, en notación infija pues su valor se calcula únicamente a partir de los valores de los atributos sintetizados pertenecientes a sus hijos en el árbol de análisis; Además podemos decir que es una gramática S-atribuida

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

Reglas de producción semántica:

$E.val ::= (E.val\ op.val\ E.val)$
$E.val ::= id.name$
$op.val ::= + \mid - \mid * \mid /$

b) Verifica la gramática propuesta usando $A\ A\ B - * C\ /$.



2. La siguiente gramática genera los números reales positivos en sistema binario:

$$\texttt{Start} \rightarrow \texttt{Dgts . Dgts}$$
$$\texttt{Start} \rightarrow \texttt{Dgts}$$
$$\texttt{Dgts} \rightarrow \texttt{Dgts Bit}$$
$$\texttt{Dgts} \rightarrow \texttt{Bit}$$
$$\texttt{Bit} \rightarrow \texttt{1}$$
$$\texttt{Bit} \rightarrow \texttt{0}$$

a) Aumenta la gramática para convertirla en una gramática con atributos en la que las funciones semánticas acumulen el valor de la constante en un atributo val en la raíz del árbol de sintáxis concreta en decimal.

$\texttt{Start} \rightarrow \texttt{Dgts}_1 \texttt{ . Dgts}_2 \Rightarrow \texttt{Start.val := Dgts}_1\texttt{.val + (Dgts}_2\texttt{.val / 10}^{Dgts_2.val}$

$\texttt{Start} \rightarrow \texttt{Dgts} \Rightarrow \texttt{Start.val := Dgts.val}$

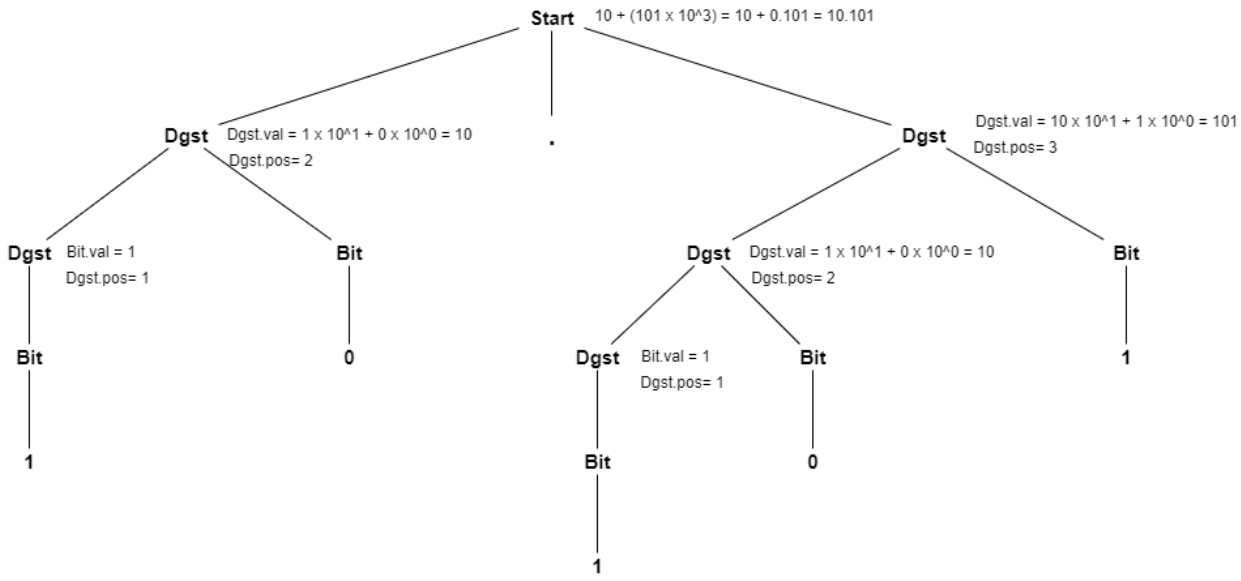
Dgts → Dgts Bit ⇒ Dgts.val := (Dgts.val x 10^1) + (Bit.val x 10^0) | Dgts.pos := Dgts.pos + 1

Dgts → Bit ⇒ Dgts.val := Bit.val | Dgts.pos := 1

Bit → 1 ⇒ Bit.val := 1

Bit → 0 ⇒ Bit.val := 0

b) Calcula el árbol de 10.101 usando tu respuesta. Agrega los atributos a cada nodo indicando el flujo del cálculo.



3. La siguiente gramática genera programas que constan de declaraciones y expresiones:

```
P → D ; E
P → D ; D | id : T
T → char | integer | array [num] of T
E → literal | num | id | E mod E | E | E [E]
```

Por ejemplos:

```
a : integer; b : Array [5] of char; a mod b[4]
```

Este programa está asignando tipos incorrectos, demuéstralo extendiendo la gramática con las funciones semánticas necesarias para la verificación de tipos de las expresiones asumiendo que existe una función **addtype(id.entry, T.type)** que agrega el tipo declarado a la tabla de símbolos en la posición correspondiente a **id**.

Primero extenderemos la gramatica:

```
P → D ; E
D → D ; D
D → id : T {addtype (id.entry T.type)}
T → char {T.type := char}
T → integer {T.type := integer}
T → array[num] of Ti {T.type := array of Ti.type}
E → literal {E.type := char}
E → num {E.type := integer}
E → id {E.type := if type (id.entry)
                  then type(id.entry)
                  else error de tipado}
E → E1 mod E2 {E.type := if E1.type = integer && E2.type = integer
                         then integer
                         else error de tipado}
E → E1 [E2] {E.type := if E2.type = integer && E1.type = array of T
                       then T
                       else error de tipado}
```

Entonces, sea `a : integer; b : Array [5] of char; a mod b[4]`