Escuela Profesional de Ciencia de la Computación
Lenguajes de Programación - 2022-2
CCOMP7-1
Prof. Marcela Quispe Cruz
25/08/2022

Actividad 02

Nombres:

Mauricio Carazas Segovia

Lucia Angie Alejandra Dueñas Flores

Pregunta 01
Utilizando la gramática del ejemplo 3.4 del libro de texto:

```
<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <id> + <expr>
     | <id> * <expr>
     | (<expr>)
     | <id>
```

Muestre un árbol de análisis sintáctico y una derivación más a la izquierda para cada una de las siguientes sentencias:

1. `A = A * (B + (C * A))`

1.1 $A = A^* (B + (C * A))$

<assign> → <id> = <expr>

<id> = <expr>

A = <expr>

A = <id> * <expr>

A = A * <expr>

A = A * (<expr>)

A = A * (<id> + <expr>)

A = A * (B + <expr>)

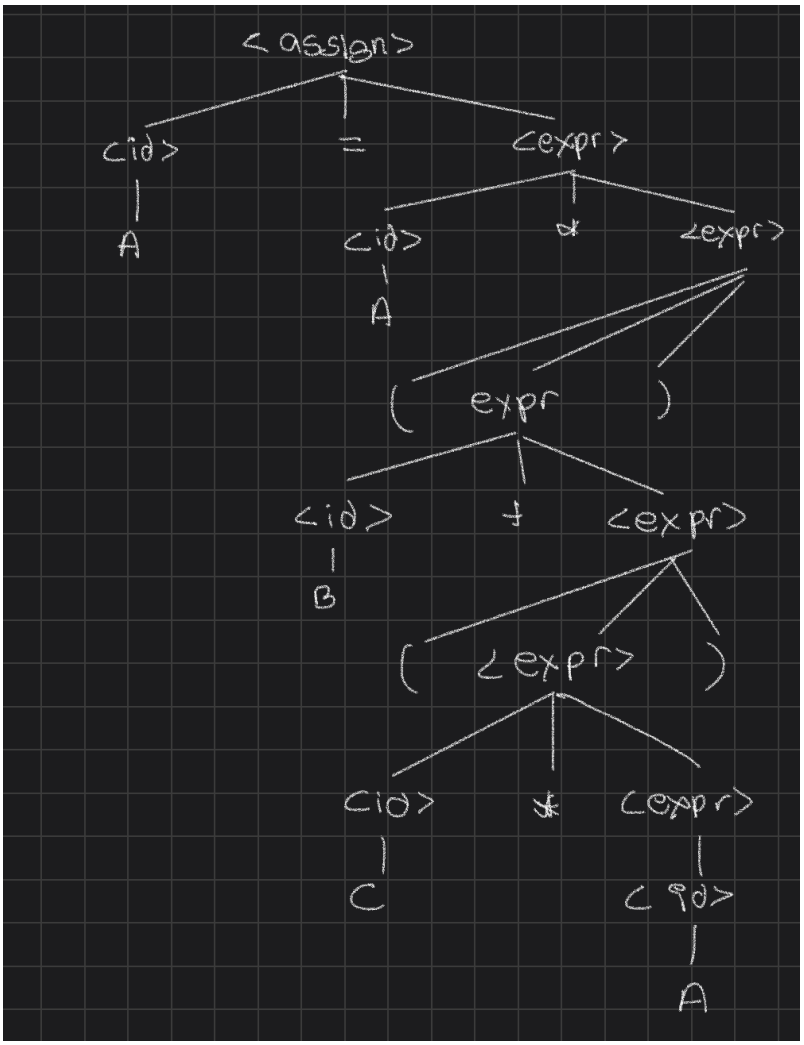
A = A * (B + (<expr>))

A = A * (B + (<id> * <expr>))

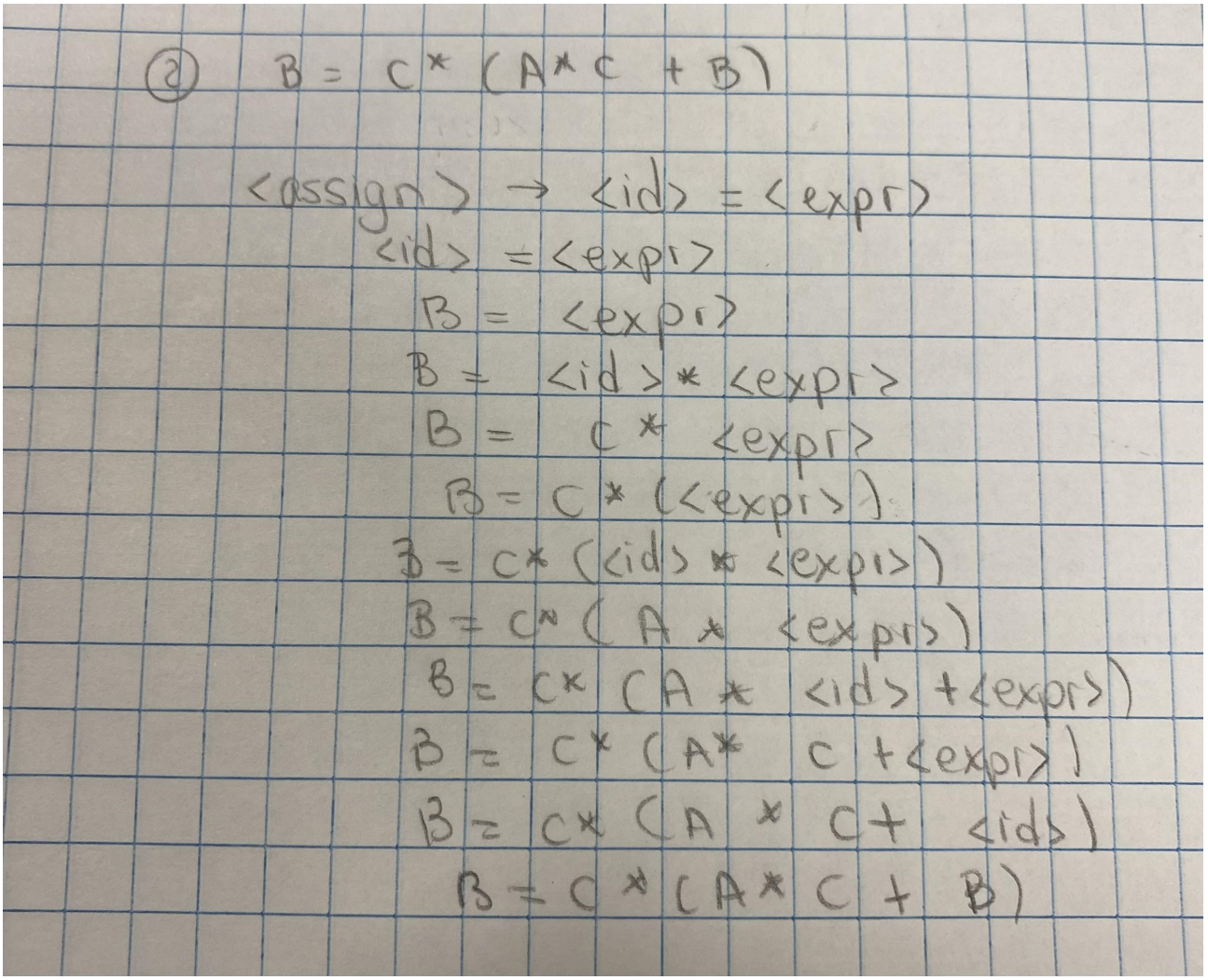
A = A * (B + (C * <expr>))

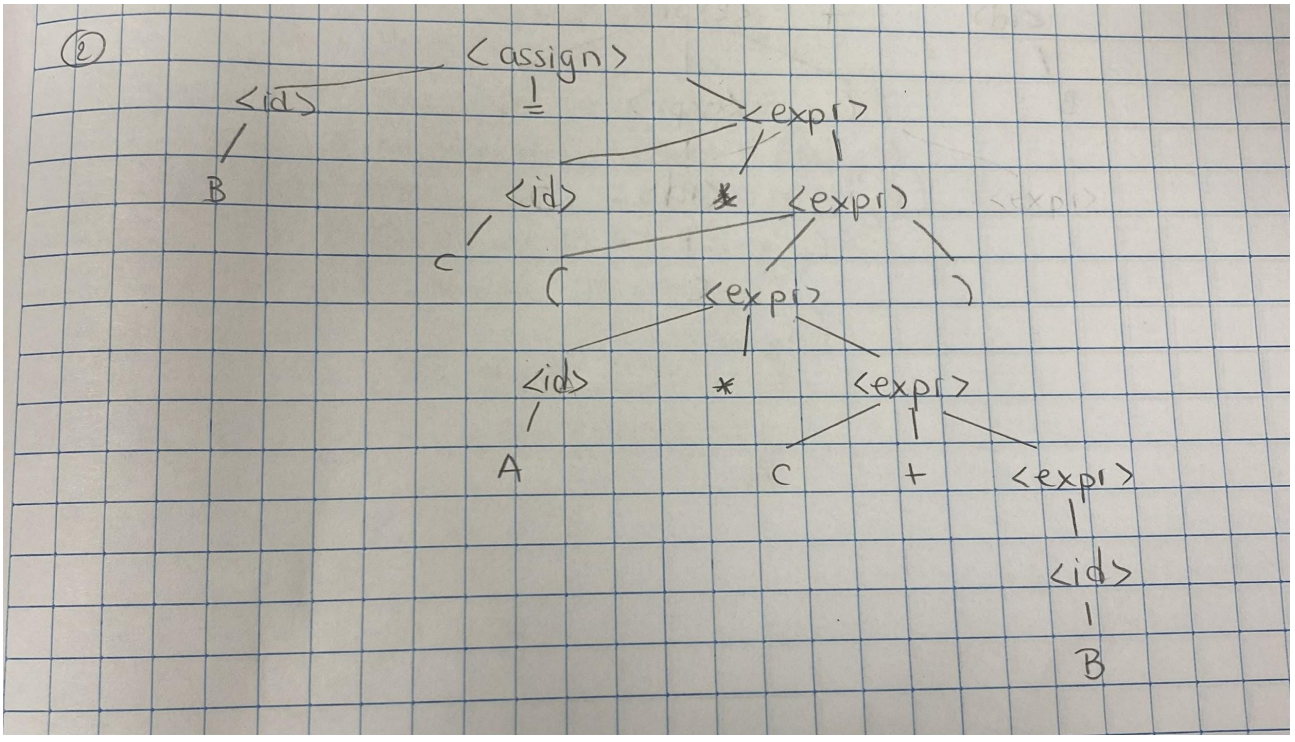
A = A * (B + (C * <id>))

A = A * (B + (C * A))

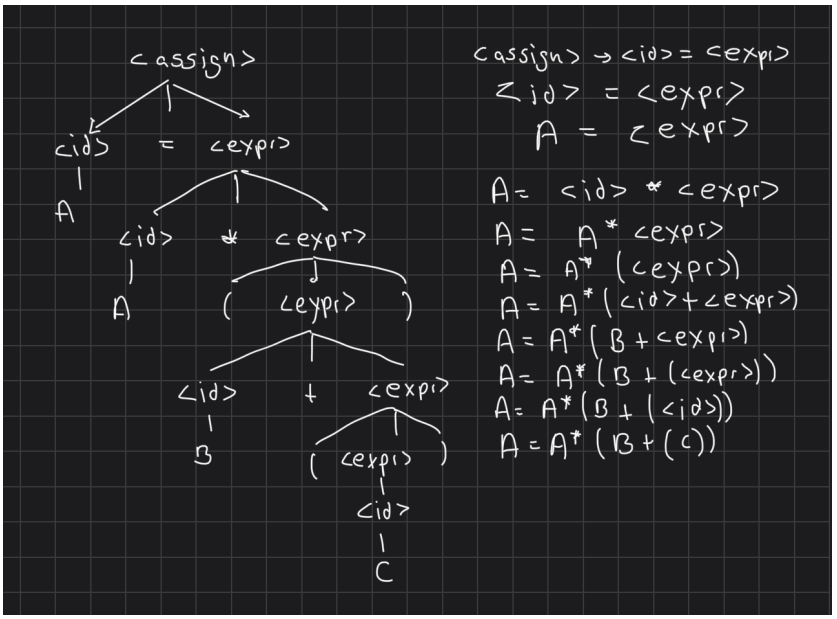


2. B = C * (A * C + B)


② B = C* (A* C + B)
<assign> → <id> = <expr>
<id> = <expr>
B = <expr>
B = <id> * <expr>
B = C * <expr>
B = C * (<expr>)
B = C* (<id> * <expr>)
B = C* (A * <expr>)
B = C* (A * <id> + <expr>)
B = C* (A* C + <expr>)
B = C* (A * C+ <id>)
B = C * (A* C + B)


②
<assign>
<id>
B
=
<expr>
<id>
C
*
<expr>
(
<expr>
)
<id>
A
*
<expr>
C
+
<expr>
<id>
B

3. `A = A * (B + (C))`



Pregunta 02 .
Considere la siguiente gramática:

```
<lexp> → <atomo> | <lista>
<atomo> → número | identificador
<lista> → (<lexp-sec>)
<lexp-sec> → <lexp-sec> <lexp> | <lexp>
```

Escriba una derivación más a la derecha para la cadena *(a 23 (m x y))* y construya un árbol de análisis sintáctico.

② (a 23 (m x y))

<lexp> → <list>

<list> → (<lexp-sec>)

(<lexp-sec>) → (<lexp-sec><lexp>)

(<lexp-sec><lista>)

(<lexp-sec>(<lexp-sec>))

(<lexp-sec>(<lexp-scc><lexp>))

(<lexp-sec>(<lexp_sec><atomo>))

(<lexp-sec>(<lexp-sec> y))

(<lexp-sec>(<lexp-sec><lexp>y))

(<lexp-sec>(<lexp-sec><atomo>y))

(<lexp-sec>(<lexp-sec> x y))

(<lexp-sec> (<lexp) x y))

(<lexp-sec> (<atom> x y))

(<lexp-sec> (m x y))

(<lexp-sec> <lexp> (m x y))

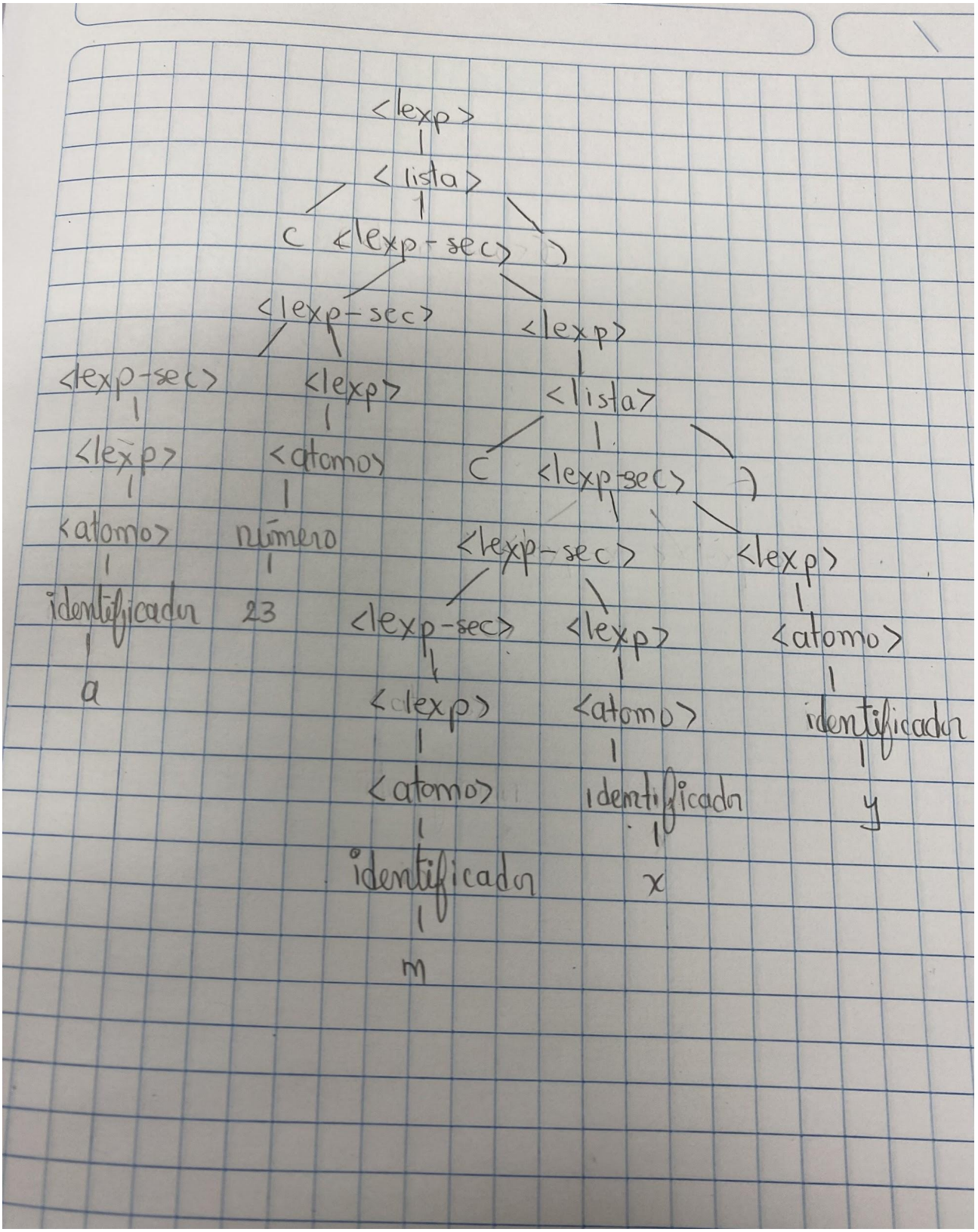
(<lexp-sec> <atom> (m x y))

(<lexp-sec> 23 (m x y))

(<lexp> 23 (m x y))

(<atom> 23 (m x y))

(a 23 (m x y))



Pregunta 03. De una gramática para el comando si de Python

Rpta:

<if - statement> ::= if(<boolean>) <statement> |
if (<boolean>) <statement><elif> else <statement > |
if (<boolean>) <statement> else <statement >

<elif> ::= elif (<boolean>) <statement> | <elif>

Pregunta 04. Considere el trecho extraído de la documentación del lenguaje LUA:

The control structures **if**, **while**, and **repeat** have the usual meaning and familiar syntax:

```
        stat ::= while exp do block end
        stat ::= repeat block until exp
        stat ::= if exp then block {elseif exp then block} [else
block] end
```

Ese trecho describe en BNF los comandos de control **if**, **while** y **repeat**. De la descripción arriba, un recurso que flexibiliza bastante estas descripciones son los operados { } y [ ], presentes en la descripción del **if**. El operador { } indica que un trecho puede repetirse innúmeras veces, o sea, podemos tener varios **elseif**'s para un mismo **if**. El operador [ ] indica que un trecho es opcional. En este caso, el operador está indicando que un **else** puede o no ocurrir en un **if**.

Considere la función abajo que calcula el n-ésimo término de la serie de fibonacci. En seguida tenemos comandos que testean la función definida (formato común de lenguajes script, donde no hay una función main explícita que indica el inicio de la ejecución; el interpretador ejecuta línea a línea, declarando la función y, en seguida, la asignación y la impresión ("..” es concatenación en Lua).

```
function nfib (n)
   if ((n == 1) or (n == 2)) then
      return 1
   end
   return nfib (n-1) + nfib(n-2)
end

x = 7
print('N-esimo fib de ' .. x .. ' es ' .. nfib(x))
```

Implemente una versión iterativa de esa función en Lua, teniendo como base la gramática presentada y el ejemplo de programa (Fibonacci).

```
Function m = FibIt(n)
   v=[0,1]
   i=2
    while (i <= n) do
        aux = v[0]
        v[0] = v[0] + v[1]
       v[1] = aux

    end
m = v[0]
```

end

Pregunta 05. Dé un ejemplo de una gramática que capture la asociatividad por la derecha para un operador de exponenciación (por ejemplo, ** en Fortran).

Rpta:

<exponenciación> => <expresión>

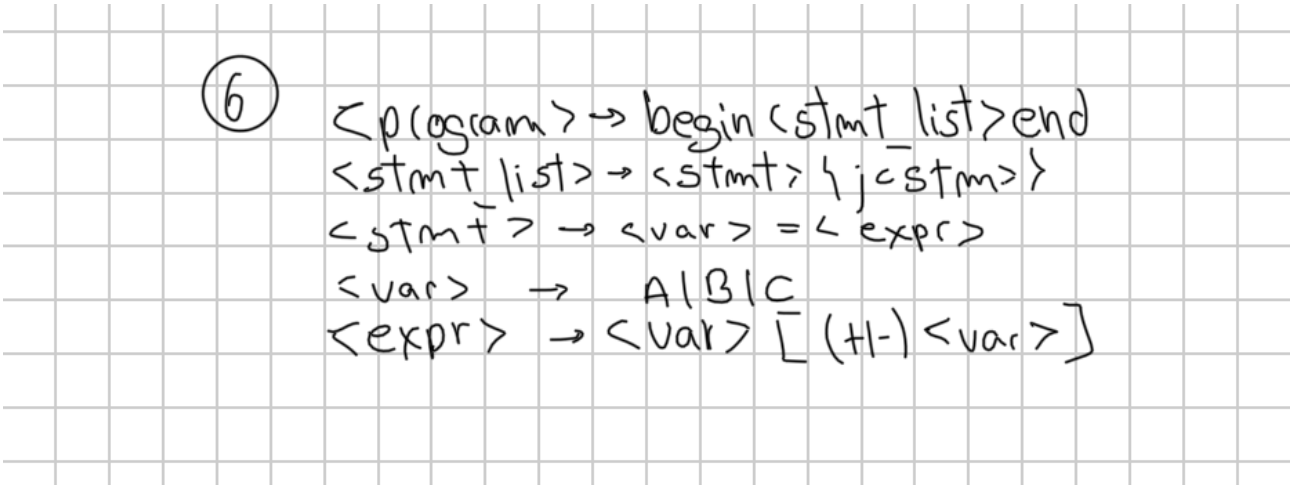
<expresión> => <base> ** <valor>

<base> => (<base>) | <base> ** <valor> | <valor>

<valor> => número entero positivo

Pregunta 06. Problema 15 del capítulo 03 del libro de texto. Transforme la siguiente gramática en EBNF:

```
<program>    → begin <stmt_list> end
<stmt_list>  → <stmt>
             | <stmt> ; <stmt_list>
<stmt>       → <var> = <expression>
<var>        → A | B | C
<expression> → <var> + <var> | <var> – <var> | <var>
```



Pregunta 07. Problema 16 del capítulo 03 del libro de texto. Transforme la siguiente gramática en EBNF:

```
<assign> → <id> = <expr>
<id>     → A|B|C
<expr>   → <expr> + <expr>
```

```
        | <expr> * <expr> | (<expr>)
        | <id>
```

Rpta:

<assign> -> <id> = <expr>
<id> -> A | B | C
<expr> -> <expr> { (+ | *) <expr>} | <id>