Evaluación continua 1 - Prueba 5

1. Objetivo

El objetivo es diseñar e implementar un *visitor* para un lenguaje imperativo con sentencias de control y expresiones booleanas.

2. Indicaciones

Se cuenta con un *scanner*, *parser* y *visitor* que trabajan sobre la siguiente gramática:

$$
\begin{aligned}
Program &::= StmtList \\
StmtList &::= Stmt\ (';'\ Stmt)^* \\
Stmt &::= id\ '='\ CExp \\
&\quad \mid \text{print}\ '('\ CExp\ ')' \\
&\quad \mid \text{if}\ CExp\ \text{then}\ StmtList\ [\ \text{else}\ StmtList\ ]\ \text{endif} \\
&\quad \mid \text{while}\ CExp\ \text{do}\ StmtList\ \text{endwhile} \\
CExp &::= Bexpr\ [<\ Bexpr] \\
BExp &::= expr\ \{(+ \mid -)\ expr\}^* \\
expr &::= term\ \{(* \mid /)\ term\}^* \\
term &::= factor\ [**\ factor] \\
factor &::= number \mid (CExp) \mid id
\end{aligned}
$$

Esta gramática define un mini-lenguaje imperativo que permite construir programas compuestos por una lista de sentencias separadas por punto y coma. Ejemplos de programas válidos:

- **Ejemplo 1 – Asignación y operación aritmética**

```
x = 5 + 3 * 2 ;
print(x)
```

- **Ejemplo 2 – Condicional simple**

```
x = 10 ;
if x < 20 then
    print(x)
else
    print(0)
endif
```

- **Ejemplo 3 – Bucle while**

```
n = 5 ;
while n < 10 do
    n = n + 1 ;
    print(n)
endwhile
```

- **Ejemplo 4 – Expresiones con potencia y paréntesis**

```
y = (2 + 3) ** 2 ;
print(y)
```

- **Ejemplo 5 – Condicional anidado con operaciones**

```
a = 4 * 2 ;
b = a - 3 ;
if b < 5 then
    print(b)
else
    while b < 15 do
        b = b + 2 ;
        print(b)
    endwhile
endif
```

La gramática debe ser extendida de modo que el *scanner*, el AST, el *parser* y el *visitor* soporten las siguientes construcciones adicionales:

$$
\begin{aligned}
\mathit{Program} &::= \mathit{StmtList} \\
\mathit{StmtList} &::= \mathit{Stmt}\ (;\ \mathit{Stmt})^{*} \\
\mathit{Stmt} &::= \mathit{id} = \mathit{AExp} \\
&\mid \textbf{print}\ (\ \mathit{AExp}\ ) \\
&\mid \textbf{if}\ \mathit{AExp}\ \textbf{then}\ \mathit{StmtList}\ [\ \textbf{else}\ \mathit{StmtList}\ ]\ \textbf{endif} \\
&\mid \textbf{while}\ \mathit{AExp}\ \textbf{do}\ \mathit{StmtList}\ \textbf{endwhile} \\
&\mid \textbf{switch}\ \mathit{AExp}\ \mathit{partcase}^{+}\ [\ \mathit{partdefault}\ ]\ \textbf{endswitch} \\
\mathit{partcase} &::= \textbf{case}\ \mathit{AExp} : \mathit{StmtList} \\
\mathit{partdefault} &::= \textbf{default} : \mathit{StmtList}
\end{aligned}
$$

$$
\begin{aligned}
AExp &::= BExp\ [(\textbf{and} \mid \textbf{or})\ BExp] \\
BExp &::= CExp\ [(< \mid >)\ CExp] \\
CExp &::= DExpr\ \{(+ \mid -)\ DExpr\}^* \\
DExpr &::= Term\ \{(* \mid /)\ Term\}^* \\
Term &::= Factor\ [** Factor] \\
Factor &::= number \mid \textbf{true} \mid \textbf{false} \mid \textbf{not}\ (AExp) \mid (AExp) \mid \textbf{id}
\end{aligned}
$$

Ejemplos de cadenas válidas con las nuevas reglas:

- **Programa 1**

```
x = 1;
switch x
case 1: print(100)
endswitch
```

- **Programa 2**

```
y = 2;
switch y
case 1: print(10)
case 2: print(20)
endswitch
```

- **Programa 3**

```
z = 3;
switch z
case 1: print(111)
case 2: print(222)
default: print(0)
endswitch
```

- **Programa 4**

```
x = 0;
while (x < 3) do
    switch x
    case 0: print(10)
    case 1: print(20)
    default: print(30)
    endswitch
    x = x + 1
endwhile
```

- **Programa 5**

```
n = 2;
m = 5;
switch n
case 1:
    print(100)
case 2:
    if not (m < 4) then
        switch m
        case 5: print(500)
        case 6: print(600)
        default: print(0)
        endswitch
    else
        print(200)
    endif
default:
    print(999)
endswitch
```

3. Sugerencias:

- Los valores `true` y `false` deben ser admitidos dentro de `NumberExp`, asignándoles los valores enteros 1 y 0 respectivamente.
- El procedimiento recomendado es: primero agregar los **tokens**, luego ajustar el **escanner**; posteriormente modificar el **AST**, después actualizar el **parser** y, finalmente, el **visitor**.

- Para extender el AST es necesario crear dos nuevas clases:
  - `SwitchStm` como subclase de `Stm`, que represente la construcción `switch`.
  - `NotExp` como subclase de `Exp`, que modele la operación unaria `not`.