Evaluación continua 2 - Prueba 1

1. Objetivo

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

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:

$$
\begin{aligned}
Program &::= Body \\
Body &::= VarDecList\ StmtList \\
VarDecList &::= (VarDec)^* \\
VarDec &::= var\ Type\ VarList\ ; \\
Type &::= id \\
VarList &::= id\ (,\ id)^* \\
StmtList &::= Stmt\ (;\ Stmt)^*
\end{aligned}
$$

$$
\begin{aligned}
Stmt &::= id \ = \ CExp \\
&\ \ | \quad print \ (\ CExp\ ) \\
&\ \ | \quad if \ CExp \ then \ Body \ [\ else \ Body\ ] \ endif \\
&\ \ | \quad while \ CExp \ do \ Body \ endwhile \\
CExp &::= BExp \ [< \ BExp\ ] \\
BExp &::= Expr \ \{ \ (+ \mid -) \ Expr \ \}^* \\
Expr &::= Term \ \{ \ (* \mid /) \ Term \ \}^* \\
Term &::= Factor \ [* * \ Factor\ ] \\
Factor &::= number \ \mid \ (CExp) \ \mid \ id \ \mid \ true \ \mid \ false
\end{aligned}
$$

Ejemplos de cadenas válidas con las nuevas reglas:

- Programa 1

```
var int a, b, c;
a = 2;
b = 3;
c = a * b + 4;
print(c)
```

- Programa 2

```
var int x, y;
x = 10;
y = 20;
if x < y then
    var int z;
    z = y - x;
    print(z)
else
    var int w;
    w = x - y;
    print(w)
endif
```

- Programa 3

```
var int x;
var bool flag;
x = 3;
flag = true;
if flag then
    var int y;
    y = x * 2;
    print(y)
else
    var int y;
    y = 0;
    print(y)
endif;
print(flag)
```

- Programa 4

```
var int x,y;
x = 3;
y = 2;
if x < 5 then
    var int y;
    y = 10;
    print(y)
endif;
print(y)
```

- Programa 5

```
var int limit, temp;
limit = 3;
temp = 0;
while 0  < limit do
    var int temp;
    temp = limit * 2;
    print(temp);
    temp = temp + 1;
    print(temp);
    limit = limit - 1
endwhile;
print(limit);
print(temp)
```

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.
- Utilice la clase clase `Environment` que administra los entornos (niveles o ámbitos) de variables en un lenguaje imperativo con soporte para tipos y valores.
  - **Estructura principal:**
    - `levels`: vector de mapas que asocia nombres de variables con sus valores enteros.
    - `type_levels`: vector paralelo que almacena los tipos de cada variable.
  - **Gestión de niveles:**
    - `add_level()`: crea un nuevo nivel o bloque (por ejemplo, al entrar en un `if` o `while`).
    - `remove_level()`: elimina el nivel más reciente (al salir del bloque).
    - `clear()`: limpia todos los niveles.
  - **Manejo de variables:**
    - `add_var(var, value, type)`: agrega una variable con valor inicial y tipo.
    - `add_var(var, type)`: agrega una variable sin valor inicial (asigna 0 por defecto).
    - `update(x, v)`: actualiza el valor de una variable existente.
    - `check(x)`: verifica si una variable está declarada en algún nivel.
  - **Búsqueda y validación:**
    - `lookup(x)`: devuelve el valor actual de la variable.
    - `lookup_type(x)`: devuelve el tipo asociado a la variable.
    - `typecheck(var, expected_type)`: valida que el tipo real coincida con el tipo esperado.
  - **Soporte para anidamiento:** Cada vez que se entra a un nuevo bloque (como en estructuras `if`, `while`, o funciones), se crea un nuevo nivel donde las variables pueden volver a declararse sin afectar las de niveles superiores.