1151044 - Programación Orientada a Objetos
Tarea 5 - 2026-I

Se desea implementar un intérprete capaz de ejecutar una secuencia de instrucciones de los siguientes tipos:

- `ASIGNA` *variable-destino entero*
  Esta instrucción le asigna el valor entero dado a la variable con el nombre indicado. Si la variable no existe, la crea. Si la variable ya existe, sobreescribe su valor.
- `ASIGNA_CONDICIONAL` *variable-condición variable-destino entero*
  Esta instrucción es similar a la anterior, pero sólo tiene efecto si la variable condición tiene un valor distinto de cero. Si la variable condición no existe, se genera un error.
- `IMPRIME` *variable*
  Esta instrucción imprime el valor de la variable con el nombre indicado. Si la variable no existe, se genera un error.

Para implementar el intérprete anterior, se desea definir la clase abstracta `instruccion`[1] además de tres clases concretas `instruccion_asignacion`, `instruccion_asignacion_condicional` e `instruccion_impresion` que modelan respectivamente cada uno de los tipos de instrucciones del intérprete.

```
class instruccion {
protected:
   int numero_instruccion;
public:
   explicit instruccion(int numero) {
      numero_instruccion = numero;
   }
   virtual ~instruccion( ) = default;
   virtual void ejecuta(std::map<std::string, int>& tabla_variables, std::ostream& salida) const = 0;
};

class instruccion_asignacion : public instruccion {
private:
    //...
public:
    //...
    void ejecuta(std::map<std::string, int>& tabla_variables, std::ostream& salida) const {
       //...
    }
};

class instruccion_asignacion_condicional : public instruccion {
private:
    //...
public:
    //...
    void ejecuta(std::map<std::string, int>& tabla_variables, std::ostream& salida) const {
       //...
    }
};

class instruccion_impresion : public instruccion {
private:
    //...
public:
    //...
    void ejecuta(std::map<std::string, int>& tabla_variables, std::ostream& salida) const {
       //...
    }
};
```

El comportamiento esperado de la función `ejecuta(...)` es el siguiente:

- El parámetro `std::map<std::string, int>&` es una referencia a un mapa que asocia el nombre de cada variable existente con su valor entero.

[1] Recordar que una clase abstracta tiene funciones virtual puras, pero a diferencia de las interfaces, sí puede tener variables miembro.

- El parámetro `std::ostream& salida` es el flujo de salida sobre el que se deben ejecutar las instrucciones `IMPRIME`[2].
- La función debe ejecutar la semántica esperada de la instrucción modelada por el tipo concreto. Si una instrucción falla, se debe elevar un error de tipo `std::runtime_error` donde el mensaje de error comience con el número de instrucción que produjo el error seguido de dos puntos (por ejemplo, `"3: Variable no existe"`).

Con base en lo anterior, implementa los tipos concretos descritos y también las siguientes dos funciones:

```
std::vector<std::shared_ptr<instruccion>> preprocesa(std::istream& entrada) {
   //...
}
```

Esta función toma un flujo de entrada de donde se debe leer una secuencia de líneas, cada una con una instrucción a analizar, y debe devolver un vector de apuntadores a objetos que implementan la clase abstracta `instruccion` y que correspondan con las instrucciones que se leyeron, en el orden en el que aparecían en la entrada. Las instrucciones se numeran implícitamente a partir de 1 según el número de línea en la que aparecen. Puedes suponer que no hay líneas vacías, que todas las instrucciones son léxica y sintácticamente correctas, que todas los nombres de variables están conformados exclusivamente por entre una y cinco letras minúsculas, y que los valores enteros están en el rango de -100 a $+100$.

```
void ejecuta(const std::vector<std::shared_ptr<instruccion>>& instrucciones, std::ostream& salida) {
   //...
}
```

Esta función toma un arreglo de apuntadores a `instruccion`, construye un mapa inicialmente vacío y ejecuta las instrucciones del arreglo, una tras usando y usando el mismo mapa. La salida de las instrucciones debe enviarse al flujo de salida dado.

Tu programa no debe presentar fugas de memoria. Tu código no debe declarar variables estáticas ni globales. Un programa que procese a lo mucho cien instrucciones debe terminar en menos de un segundo. Los mensajes de error son libres excepto por la indicación del número de instrucción en el que se producen. Cada tipo concreto de instrucción debe implementar únicamente la lógica correspondiente a la instrucción que modela. Puedes consultar una página de prueba en https://racc.mx/uam/trimestre-actual/2026-i/poo/tarea5.html. El código que envíes no debe contener `main`, no debe contener la definición de `instruccion` y tampoco debe leer ni imprimir nada directamente.

Envía tu código fuente desde tu cuenta institucional al formulario en https://forms.gle/hKTWWHzwj873dGN38. Tu código será evaluado con varios casos de prueba y se espera que cumpla la semántica descrita.

Código	Entrada	Salida

`int main( ) try {`
`  auto instrucciones = preprocesa(std::cin);`
`  ejecuta(instrucciones, std::cout);`
`} catch (const std::exception& ex) {`
`  std::cout << ex.what( );`
`}`
	`ASIGNA a 5`
	`IMPRIME a`
	`ASIGNA b 0`
	`ASIGNA_CONDICIONAL b a 10`
	`IMPRIME a`
	`ASIGNA b 1`
	`ASIGNA_CONDICIONAL b a 12`
	`IMPRIME a`
		`5`
		`5`
		`12`

`// el mismo código`	`ASIGNA a 11`
	`ASIGNA a 12`
	`ASIGNA a 13`
	`ASIGNA a 14`
	`ASIGNA a 15`
	`IMPRIME b`
		`6: Variable no existe`

[2]Nótese que sólo a uno de las tres tipos de instrucciones le interesa el flujo de salida. A veces los requerimientos de un tipo derivado pueden terminando "contaminando" la jerarquía completa, y esto es señal de que tal vez convenga un diseño alternativo con `dynamic_cast` o `typeid`.