# Tecnológico de Monterrey

Instituto Tecnológico de Estudios Superiores Monterrey

CAMPUS MONTERREY

TC3002B | Desarrollo de aplicaciones avanzadas de ciencias computacionales

**Grupo** 502

**Patito**

**Alumno:**

Erick Alfredo García Huerta - A01708119

# Cubo Semántico

**Estructura de Datos Seleccionada:** Diccionarios anidados

**Razones para su Selección:**

El cubo semántico es una estructura que permite determinar el tipo de resultado de una operación dado el tipo de los operandos y el operador. Los diccionarios anidados son ideales para este propósito debido a:

- **Acceso Rápido:** Proporcionan complejidad temporal constante para búsquedas, lo cual es esencial en compiladores para mantener la eficiencia.
- **Representación Clara:** Permiten una visualización explícita de las combinaciones de tipos y operaciones, facilitando el mantenimiento y la extensibilidad.
- **Flexibilidad:** Es sencillo añadir nuevos tipos u operaciones según evolucione el lenguaje.

**Estructura del Cubo:**

El cubo semántico se implementa como un diccionario donde la clave principal es el operador, y los valores son diccionarios anidados que representan los tipos de los operandos y el resultado:

```
# Operación
'+': { # Suma
    # Tipo de operando : {Tipo de operando: Resultado | Tipo de operando: Error}
    'int': {'int': 'int', 'float': 'float', 'bool': 'error'},
    'float': {'int': 'float', 'float': 'float', 'bool': 'error'},
    'bool': {'int': 'error', 'float': 'error', 'bool': 'error'}
},
```

## Operaciones principales

```
def get_result_type(self, operation, type1, type2):
  # Obtiene el resultado de la operación entre dos tipos o devuelve error
  return self.cube.get(operation, {}).get(type1, {}).get(type2, 'error')
```

# Directorio de Funciones

**Estructura de Datos Seleccionada:** Diccionario

**Razones para su Selección:**

El directorio de funciones actúa como una tabla de símbolos para las funciones, almacenando información como el tipo de retorno, parámetros y variables locales. Un diccionario es apropiado porque:

- **Mapeo Directo:** Las funciones se identifican de manera única por su nombre, que sirve como clave.
- **Eficiencia en Búsquedas:** Permite acceder rápidamente a la información de una función por su nombre.
- **Facilidad de Actualización:** Es sencillo añadir o modificar información de funciones.
- **Extensibilidad:** Se puede ampliar para incluir más atributos si es necesario.

```
# Cada función tiene su propia tabla de variables locales
self.functions[name] = {
  'return_type': return_type,
  'parameters': parameters,  # Lista de parámetros [(nombre, tipo)]
  'variables': {},  # Tabla de variables locales
}
```

## Operaciones principales

Esta operación añade una función nueva a la tabla o regresa un error en caso de que se intente agregar varias veces la misma.

```
def add_function(self, name, return_type, parameters):
  if name in self.functions:
    raise Exception(f"Error: La función '{name}' ya ha sido declarada.")
  else:
    # Cada función tiene su propia tabla de variables locales
    self.functions[name] = {
      'return_type': return_type,
      'parameters': parameters,  # Lista de parámetros [(nombre, tipo)]
      'variables': {},  # Tabla de variables locales
    }
```

Esta función auxiliar apoya para tomar una función de la tabla para trabajos específicos

```
def get_function(self, name):
  return self.functions.get(name, None)
```

En caso de que una variable  esté siendo declarada en la función, se llama para añadirla a la lista de variables en la función

```
def add_variable_to_function(self, func_name, var_name, var_type):
  function = self.get_function(func_name)
  if function:
    if var_name in function['variables']:
      raise Exception(f"Error: La variable '{var_name}' ya ha sido declarada en la función '{func_name}'.")
    else:
      function['variables'][var_name] = {
```

```
            'type': var_type,
        }
    else:
        raise Exception(f"Error: La función '{func_name}' no está definida.")
```

# Tablas de Variables

**Estructura de Datos Seleccionada:** Diccionario

**Razones para su Selección:**

Las tablas de variables almacenan información sobre las variables, incluyendo su tipo y ámbito (global o local). Un diccionario es eficaz para este propósito debido a:

- **Acceso Rápido:** Permite búsquedas y actualizaciones eficientes.
- **Unicidad de Claves:** Garantiza que cada variable en un ámbito tenga un nombre único.
- **Detección Sencilla de Errores:** Facilita la identificación de variables re declaradas.
- **Extensibilidad:** Se puede ampliar para incluir direcciones de memoria u otros atributos.

**Estructura de la Tabla:**

```
self.variables[name] = {
    'type': var_type,
    'scope': scope,
}
```

## Operaciones principales

La función añade una nueva variable en la tabla usando su nombre como llave, y almacenando su tipo y ámbito

```
def add_variable(self, name, var_type, scope):
    if name in self.variables:
        raise Exception(f"Error: La variable '{name}' ya ha sido declarada en el ámbito '{scope}'.")
    else:
        self.variables[name] = {
            'type': var_type,
            'scope': scope,
        }
```

Esta función nos permite acceder a una variable en específico en la tabla

```
def get_variable(self, name):
    return self.variables.get(name, None)
```

# Puntos Neurálgicos

Para los puntos neurálgicos había dos opciones usando antlr, listeners y visitors. Debido a la complejidad del visitor, se escogió el listener haciendo uso de una modificación en el script de antlr para crear un archivo listener. Fué el siguiente:

```
java -jar antlr-4.13.2-complete.jar -Dlanguage=Python3 patito.g4 -listener -o antlr_files
```

Para hacer esta actividad se creó el cubo semántico, que nos permite validar una operación, un operando y sus resultados. Además de que se creó un directorio de funciones que es una estructura que nos permite conocer el scope de nuestra tabla de variables, así mismo, la tabla nos permite almacenar las variables y sus tipos con ayuda del cubo semántico.