Reglas semánticas y de tipos

Gracias a que tenemos una gramática definida para el lenguaje YAPL, se puede usar esta gramática como base para establecer las reglas semánticas y de tipos. Gracias a las reglas se especifica cómo se tienen que evaluar las expresiones, cómo se hacen las asignaciones, como se manejan los diferentes tipos de datos, como se resuelven operaciones, etc.

Definir las reglas de tipos que YAPL debe cumplir, tomando como base lo descrito en el documento Aspectos semánticos YAPL (puede extenderse y adaptarse según los requisitos y características específicas del lenguaje YAPL)

- Reglas de tipos para expresiones aritméticas:
    - Las operaciones de suma, resta, multiplicación y división solo están definidas para operandos de tipo Int.
    - La operación unaria TILDE, o de negación '~', solo está definida para operandos de tipo Int.
- Reglas de tipos para expresiones de comparación:
    - Las operaciones de comparación (<, >, <=, >=, ==) están definidas para operandos de tipo Int y Bool.
- Reglas de tipos para expresiones booleanas:
    - Las operaciones AND y OR solo están definidas para operandos de tipo Bool.
- Reglas de tipos para asignaciones:
    - El tipo de dato de la expresión asignada debe ser compatible con el tipo de la variable.
    - La asignación solo puede realizarse a variables previamente declaradas.
- Reglas de tipos para la declaración de variables:
    - Las variables deben declararse antes de ser utilizadas.
    - No se permite la redeclaración de variables dentro del mismo alcance.
    - El tipo de dato de la variable debe ser válido (Int, Bool, String o un tipo definido por el usuario).
- Reglas de ámbito y alcance:
    - Verificar que las variables utilizadas estén definidas dentro del alcance adecuado.
- Reglas de uso del tipo SELF_TYPE:
    - SELF_TYPE solo puede usarse en el contexto de definición de clases.
    - SELF_TYPE se refiere al tipo de la clase actual en la que se está definiendo.
- Reglas de llamadas a funciones:
    - El número y tipo de argumentos en una llamada a función deben coincidir con la definición de la función.
- Reglas de herencia:
    - Una clase derivada solo puede heredar de una clase base válida.
    - Los miembros de la clase base están accesibles en la clase derivada.
- Reglas para la creación y uso de clases e instancias:
    - Manejar la definición y el uso de clases, incluyendo herencia y creación de instancias.

En la tabla de símbolos, también se pueden implementar reglas de alcance y visibilidad para asegurar que las variables y funciones se utilicen de manera coherente dentro del programa.

Definir el sistema de tipos a implementar en la fase de análisis semántico del compilador (puede extenderse y adaptarse según los requisitos y características específicas del lenguaje YAPL)

Es necesario definir el sistema de tipos para verificar compatibilidad y corrección de los tipos en el lenguaje YAPL. Podemos ver cómo los tipos de datos interactúan entre sí en expresiones, asignaciones y llamadas a funciones.

- Tipos básicos:
  - Int: Representa valores enteros.
  - Bool: Representa valores booleanos (verdadero o falso).
  - String: Representa cadenas de texto.
- Tipos de usuario (clases):
  - YAPL permite la definición de clases por el usuario. Cada clase define un nuevo tipo de dato.
  - Los objetos se crean a partir de clases y tienen el tipo de esa clase.
  - Las clases pueden heredar de otras clases, lo que permite la creación de una jerarquía de clases.
- Reglas de herencia y tipos derivados
  - Una clase derivada puede heredar de una clase base y se considera un tipo derivado de la clase base.
  - Un objeto de una clase derivada es también un objeto de la clase base, lo que permite el uso polimórfico.
- Reglas de subtipado:
  - En YAPL, se permite el subtipado en el caso de las clases. Esto significa que una clase derivada se puede usar en lugar de una clase base en la mayoría de los contextos, ya que un objeto de la clase derivada también es un objeto de la clase base.
  - Por ejemplo, si ClassA es una clase base y ClassB es una clase derivada de ClassA, entonces un objeto de tipo ClassB puede ser utilizado donde se espera un objeto de tipo ClassA.
- Uso de SELF_TYPE:
  - El tipo SELF_TYPE se utiliza dentro de la definición de una clase para referirse al tipo de la clase actual que se está definiendo.
  - En la fase de análisis semántico, se realizará una sustitución de SELF_TYPE por el nombre de la clase actual.

Diseñar una estructura de datos inicial para definición e implementación de Tabla de Símbolos.

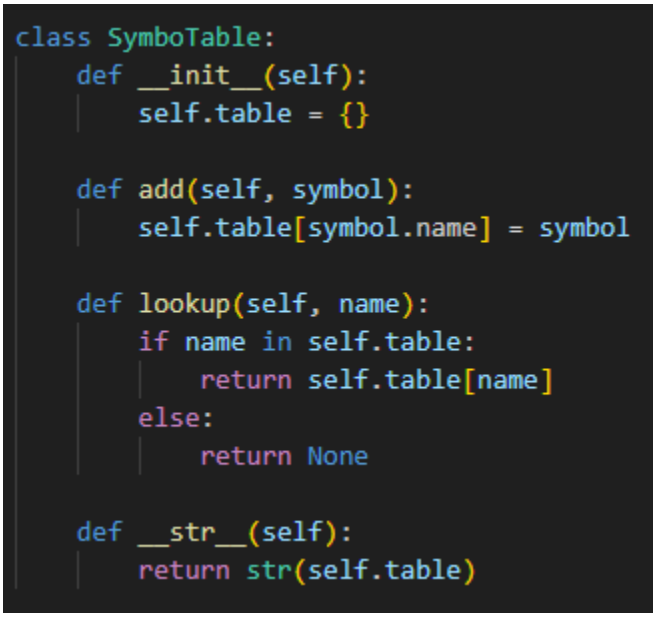
a. **Las operaciones mínimas sobre la tabla de símbolos deben ser las de agregar y consultar un símbolo dentro del ámbito actual.**

La funcionalidad básica requerida para la tabla de símbolos debe permitir agregar nuevos símbolos a la tabla y consultar los símbolos que ya están presentes en el ámbito actual.

Cada símbolo tiene un nombre y un tipo asociado. Los símbolos son representados mediante la clase Symbol, donde cada objeto Symbol tiene un atributo name y un atributo type.

Para agregar y consultar un símbolo en el ámbito actual:

- Crear una tabla de símbolos vacía al inicio del análisis semántico.
- Durante el análisis semántico, al encontrar un nuevo símbolo (por ejemplo, al analizar una declaración de variable o una función), crear un objeto Symbol con el nombre y el tipo del símbolo y agrégarlo a la tabla de símbolos usando el método add() de la clase SymboTable.
- Cuando se necesite consultar un símbolo, es decir, cuando se encuentre una referencia a un símbolo en el código (por ejemplo, cuando se usa una variable o se llama a una función), buscar el símbolo en la tabla de símbolos usando el nombre del símbolo y el método lookup() de la clase SymboTable. Si el símbolo está presente en la tabla, se puede obtener su información (como el tipo) y usarla en el análisis semántico.



```
class SymboTable:
    def __init__(self):
        self.table = {}

    def add(self, symbol):
        self.table[symbol.name] = symbol

    def lookup(self, name):
        if name in self.table:
            return self.table[name]
        else:
            return None

    def __str__(self):
        return str(self.table)
```