Centro Universitario de Ciencias Exactas e Ingeniería

SEM. de Solución de Problemas de Traductores 2

Practica 3

LUIS FELIPE MUNOZ MENDOZA

Juan Pablo Hernández Orozco

219294285

Contenido

Objetivo:

Objetivo: Desarrollar un analizador semántico que valide la corrección de las estructuras sintácticas procesadas en la fase anterior, asegurando la correcta gestión de tipos de datos, ámbito de variables y evaluación de expresiones aritméticas.

Requerimientos:
1. Entrada:
- Se debe proporcionar una secuencia de tokens generada por el analizador léxico y validada por el analizador sintáctico.
- El programa deberá recorrer el árbol sintáctico y verificar posibles errores semánticos.

2. Validaciones a realizar:
- Declaración y uso de variables: Verificar que todas las variables utilizadas hayan sido declaradas previamente.
- Compatibilidad de tipos: Comprobar que las operaciones aritméticas, lógicas y de asignación sean compatibles con los tipos de datos definidos.
- Ámbito de variables: Identificar si se intenta acceder a una variable fuera de su alcance.
- Control de flujo: Validar las estructuras condicionales y de iteración (if, while, for) para evitar expresiones inválidas.
- Retorno de funciones: Asegurar que las funciones devuelvan un valor acorde a su tipo de retorno.

3. Evaluación de expresiones aritméticas:
-Generar un árbol sintáctico a partir de una expresión matemática proporcionada por el usuario.
-Expresiones permitidas:

- -Suma (+), resta (-), multiplicación (*), división (/).
- -Uso de paréntesis para definir prioridad.
- -Operaciones con números enteros y flotantes.

- Ejemplo de expresión válida:
(5 + 3) * (2 - 4)

-Implementar un recorrido del árbol (postorden o inorden) para evaluar la expresión.

- Cada nodo del árbol representará un operador o un operando.
- Prioridades de operadores:

  - Multiplicación y división tienen mayor prioridad que la suma y la resta.
  - Se respetará el uso de paréntesis para modificar la prioridad.

- Manejo de tipos:
Verificar que los operandos sean numéricos (enteros o flotantes).
Realizar conversión de tipos cuando sea necesario en operaciones mixtas.

4. Manejo de errores:
-División por cero: Generar una excepción o mensaje de error si ocurre.
-Operandos no numéricos: Si una expresión intenta operar con un tipo inválido, se debe notificar el error.
Errores de alcance o uso incorrecto de variables.

5. Salida:
- Si el código es semánticamente correcto, mostrar un mensaje de validación exitosa.
- Si se detectan errores, desplegar mensajes indicando el tipo de error y su ubicación.
- Mostrar el resultado de la evaluación de expresiones.
- Generar y mostrar una tabla de símbolos con información sobre las variables declaradas y su tipo.

Implementación:
Puede implementarse en cualquier lenguaje, siempre que cumpla los requerimientos.
Se recomienda utilizar estructuras como tablas de símbolos y árboles de sintaxis abstracta (AST).
Se pueden emplear herramientas como PLY (Python Lex-Yacc), ANTLR, Bison o Yacc para facilitar la implementación.

Entrega:
- Código fuente con comentarios explicativos.
- Reporte detallando el proceso de desarrollo, incluyendo:

  - Algoritmos utilizados.
  - Explicación de la tabla de símbolos.
  - Estructura del árbol sintáctico.
  - Casos de prueba con análisis de errores.

- Video corto mostrando el funcionamiento del analizador.

Recomendaciones y Recursos de Apoyo

Recomendaciones:
Construcción de la tabla de símbolos:
Implementar un diccionario o estructura hash para almacenar identificadores y sus atributos.

Incluir información como tipo de dato, valor asignado, ámbito y línea de declaración.

Implementación del Árbol Sintáctico:
Definir una estructura de nodos con operadores y operandos.
Implementar métodos de recorrido (postorden o inorden) para evaluar la expresión.

Manejo de errores semánticos:
Implementar un sistema de detección temprana para minimizar propagación de errores.
Mostrar mensajes claros con línea de error y posible causa.

Uso de herramientas:
PLY (Python Lex-Yacc): Biblioteca en Python basada en Yacc/Lex.
ANTLR: Framework para generar analizadores de lenguaje.
Bison/Yacc: Herramientas clásicas para generar analizadores en C/C++.

Introducción:

La práctica 3 consiste en implementar un analizador semántico en Python usando la librería PLY. Partiendo del AST generado por el parser, se recorre cada nodo para verificar que las variables se declaren antes de usarse, respetar tipos de datos y controlar ámbitos mediante un stack de tablas de símbolos. Además, se evalúan expresiones aritméticas con prioridad de operadores y se detectan errores como división por cero u operandos no numéricos. Finalmente, el programa informa éxito o detalle de fallos y muestra la tabla de símbolos.

Desarrollo:

Algoritmos utilizados:

- **Análisis Léxico (Tokenización)**
    - Se basa en **expresiones regulares** y el principio de *maximal munch*: el lexer recorre la entrada carácter a carácter, construye el lexema más largo que coincide con alguna regla y genera un token.
    - Internamente se modela como un **autómata finito determinista (AFD)** que reconoce patrones (números, identificadores, operadores, palabras reservadas) en tiempo lineal respecto al tamaño de la entrada.
- **Análisis Sintáctico (Parsing LALR(1))**
    - PLY genera tablas de parsing LALR(1) a partir de las producciones.
    - El parser implementa un **algoritmo shift-reduce**:
        - **Shift:** lee el siguiente token y lo apila.

        - **Reduce:** cuando el tope de pila coincide con el lado derecho de una producción, lo reemplaza por el no terminal del lado izquierdo.
    - Utiliza un **stack** de estados y símbolos para navegar la gramática con complejidad lineal en el número de tokens.
- **Gestión de la Tabla de Símbolos (Scopes Stack)**
    - Emplea un **stack** de diccionarios (symbol_stack) donde cada diccionario representa un ámbito.
    - **Declarar:** inserta { id → atributos } en el diccionario superior.
    - **Lookup:** recorre el stack de arriba hacia abajo hasta encontrar la definición.
    - **Push/Pop de ámbitos:** al entrar/salir de un bloque { … } se apila o desapila un diccionario, logrando aislamiento local de variables.
- **Representación y Recorrido del AST**
    - El AST se modela con **tuplas** heterogéneas, donde el primer elemento indica el tipo de nodo (e.g. '+', 'assign', 'if').
    - Para la evaluación semántica se usa un **recorrido postorden** (Depth-First Search):
        - Visitar recursivamente los hijos.
        - Evaluar el nodo actual combinando los resultados de subárboles.
- **Evaluación y Comprobación de Tipos**
    - Al evaluar cada nodo binario (+, -, *, /) se recuperan valores de sus hijos y se comprueba que ambos sean int o float.
    - Para asignaciones se asegura que el valor devuelto por la expresión coincida con el tipo declarado de la variable, lanzando excepción si no.
- **Detección Temprana de Errores**
    - En cada paso de evaluación se realizan chequeos semánticos (uso de variable no declarada, división por cero, condición no booleana).
    - Cuando se detecta un fallo, se **lanza una excepción** con un mensaje que incluye el tipo de error y la línea aproximada, interrumpiendo el análisis para evitar falsos positivos posteriores.

Explicación de la tabla de símbolos.

Campo	Descripción	Ejemplo
Identificador	Nombre de la variable o símbolo.	x, contador, total

Tipo	Tipo de dato declarado (controlado en la declaración).	int, float, bool, etc.
Valor	Valor actual de la variable después de su asignación (o None).	5, 3.14, True, None
Ámbito	Nivel de bloque donde fue declarada (global o bloque interno).	global, local (implícito en el stack)
Línea de declaración	Número de línea donde se declaró la variable (usado para errores).	3, 5, etc.

Estructura de almacenamiento

- Usas una lista llamada symbol_stack, que actúa como **pila** (stack).
- Cada elemento del stack es un **diccionario** { nombre → { tipo, valor, línea } }.
- El primer diccionario (symbol_stack[0]) siempre es el **ámbito global**.
- Cada vez que entras a un bloque {} (cuando reconoces un block en el AST), haces symbol_stack.append({}) para crear un nuevo **ámbito local**.

Operaciones sobre la tabla

- **Declarar una variable:**
  Al procesar una declaración (int x;), la variable x se agrega al diccionario actual con:
  - Tipo: int
  - Valor: None (no tiene valor hasta que se asigne)
  - Línea: número de línea donde fue declarada (aquí fijo en 0, pero podrías mejorarlo).

Si la variable ya existía en el mismo ámbito, lanza un error semántico.

- **Asignar un valor:**
  Cuando encuentras una asignación (x = 5;):
  - Buscas la variable en todos los ámbitos, del más interno al global.
  - Verificas que exista (si no, error de "variable no declarada").
  - Verificas compatibilidad de tipo (int, float, etc.).
  - Guardas el nuevo valor en su registro.
- **Buscar variables:**
  La función lookup(id) recorre el stack de ámbitos desde el más reciente al más antiguo, encontrando la primera definición.
  Esto **permite shadowing**: una variable local puede ocultar una variable global con el mismo nombre.

- **Manejo de bloques:**
  Cada bloque {} crea un nuevo ámbito:
    - Al entrar al bloque: symbol_stack.append({})
    - Al salir: symbol_stack.pop()

Supongamos que ejecutamos este código de prueba:

```
int x;
float y;
{
   int x;
   x = 10;
}
x = 5;
```

Durante el análisis semántico, la tabla de símbolos se verá así:

Identificador	Tipo	Valor	Ámbito	Línea
x	int	5	global	1
y	float	None	global	2

- El x interno (x = 10) fue **local** y **desapareció** al cerrar el bloque.
- El x global se mantiene y luego recibe el valor 5.

Mas adelante en las corridas de escritorio veremos mas ejemplos en donde vamos a generar sus tablas de símbolos.

Estructura del árbol sintáctico.

En la Práctica 3 se definió un Árbol Sintáctico Abstracto (AST) como una representación jerárquica de las construcciones del lenguaje, eliminando detalles inútiles (p. ej. paréntesis y punto y coma) y quedando solo la semántica esencial.

Nodo raíz

Se construyó como una tupla:

('program', lista_de_declaraciones, lista_de_sentencias)

donde lista_de_declaraciones agrupaba todos los nodos de declaración y lista_de_sentencias contenía las sentencias a ejecutar.

Declaraciones

Cada variable se representó con:

('declare', tipo, identificador)

De este modo, al recorrer el AST se podía invocar directamente la rutina de inserción en la tabla de símbolos.

Asignaciones

Las sentencias de asignación quedaron como:

('assign', identificador, subárbol_expresión)

Esto permitió evaluar primero la subexpresión y luego actualizar el valor de la variable.

Control de flujo

If sin else:

('if', condición, sentencia_true)

If con else:

('if', condición, sentencia_true, sentencia_false)

While:

('while', condición, cuerpo)

Cada uno se procesó evaluando la condición y luego el subárbol correspondiente al bloque verdadero o falso.

Bloques

Para agrupar declaraciones y sentencias en un ámbito local, se utilizó:

('block', [sentencia1, sentencia2, …])

Así se empujaba un nuevo diccionario en el stack de ámbitos al entrar y se desapilaba al salir.

Expresiones aritméticas

Las operaciones binarias se codificaron como tuplas de tres elementos:

('+', subárbol_izq, subárbol_der) # análogo para '-', '*', '/'

Y los números o identificadores como nodos hoja (5, 3.14, x). La precedencia y los paréntesis se resolvieron en la gramática, de modo que el AST respetó siempre el orden correcto de evaluación.

Con esta estructura, el recorrido recursivo postorden facilitó la implementación del análisis semántico, garantizando una evaluación ordenada de cada nodo y una detección temprana de errores.

Casos de prueba con análisis de errores.

#	Código	**Resultado esperado**	**Comentario**	Resultado en consola
1	x = 5;	[SemError] Línea 0: 'x' no declarada.	Asignación sin declaración previa	Ingrese una cadena a validar: x = 5; Árbol sintáctico: ('program', [], [('assign', 'x', 5)]) Error semántico: Variable 'x' no declarada.
2	int x; x = 5;	✓ Validación semántica exitosa Resultados: [5]	Resultados: [5] Declaración y asignación correctas	¿Desea validar otra cadena? (1.- Si / 2.- No): 1 Ingrese una cadena a validar: int x; x = 5; Árbol sintáctico: ('program', [('declare', 'int', 'x')], [('assign', 'x', 5)]) Validación semántica exitosa. Resultados: [5] Tabla de símbolos (ámbito global): {'x': {'type': 'int', 'value': 5}}
3	int x; x = 3.2;	[SemError] Línea 0: asignar 3.2 a int 'x'.	Tipo float incompatible con int	Ingrese una cadena a validar: int x; x = 3.2; Árbol sintáctico: ('program', [('declare', 'int', 'x')], [('assign', 'x', 3.2)]) Incompatible asignación: 'x' es de tipo int, pero se asigna 3.2.
4	float y; y = 2 + 3.5 *2;	✓ Validación semántica exitosa Resultados: [9.0]	Mezcla int–float, se produce float	Ingrese una cadena a validar: float y; y = 2 + 3.5 * 2; Árbol sintáctico: ('program',

				[('declare', 'float', 'y')], [('assign', 'y', ('+', 2, ('*', 3.5, 2)))]) Validación semántica exitosa. Resultados: [9.0] Tabla de símbolos (ámbito global): {'y': {'type': 'float', 'value': 9.0}}
5	int a; a = 1 / 0;	[SemError] Línea 0: división por cero.	División por cero	Ingrese una cadena a validar: int a; a = 1 / 0; Árbol sintáctico: ('program', [('declare', 'int', 'a')], [('assign', 'a', ('/', 1, 0))]) Error semántico: División por cero.
6	bool b; if (5) b = true;	[SemError] Línea 0: condición no booleana.	Condición de if no es bool	Ingrese una cadena a validar: bool b; if (5) b = true; Error de sintaxis en token: TRUE valor: true No se pudo construir el árbol sintáctico debido a errores.
7	{ int z; z = 2; } z = 3;	[SemError] Línea 0: 'z' no declarada.	z está fuera de su ámbito	Ingrese una cadena a validar: { int z; z = 2; } z = 3; Error de sintaxis en token: INT valor: int Error de sintaxis en token: RBRACE valor: } Árbol sintáctico: ('program', [], [('assign', 'z', 3)]) Error semántico: Variable 'z' no declarada.
8	int x; int x;	[SemError] Línea 0: 'x' ya declarada aquí.	Declaración duplicada en mismo ámbito	Ingrese una cadena a validar: int x; int x; Error de sintaxis: Entrada incompleta

				No se pudo construir el árbol sintáctico debido a errores.
9	int x; x = y;	[SemError] Línea 0: 'y' no declarada.	Uso de y sin declarar	Ingrese una cadena a validar: int x; x = y; Árbol sintáctico: ('program', [('declare', 'int', 'x')], [('assign', 'x', 'y')]) Error semántico: Variable 'y' usada antes de asignarse.
10	int x; x = (5 +) 3;	[SyntaxError] Línea 0: token inesperado ')'	Error de sintaxis en la expresión	Ingrese una cadena a validar: int x; x = (5 +) 3; Error de sintaxis en token: RPAREN valor:) Árbol sintáctico: ('program', [], [3]) Validación semántica exitosa. Resultados: [3] Tabla de símbolos (ámbito global): {}

Conclusión:

En esta Práctica 3 se trabajó de manera integral en el diseño e implementación de un analizador semántico usando PLY en Python. Se separaron claramente las fases de análisis léxico y sintáctico para generar un AST limpio, representado mediante tuplas que capturan declaraciones, asignaciones, expresiones y estructuras de control. Sobre ese AST se implementó un recorrido postorden que, junto con un stack de tablas de símbolos, permitió:

- **Verificar declaración y uso de variables**, impidiendo referencias a identificadores no declarados.
- **Comprobar compatibilidad de tipos** en asignaciones y operaciones aritméticas, detectando errores como asignar un float a un int.
- **Gestionar ámbitos** correctamente, creando y destruyendo tablas de símbolos en bloques, de modo que las variables locales no “contaminen” el ámbito global.
- **Evaluar expresiones** respetando precedencia y paréntesis, con manejo de errores semánticos (división por cero, operandos no numéricos) y sintácticos.
- **Soportar control de flujo** (if, while) validando condiciones booleanas y ramificaciones.

Los casos de prueba ejecutados confirmaron que el analizador detecta y reporta tanto errores de sintaxis como de semántica con mensajes claros y precisos. En conjunto, esta práctica cimenta las bases para fases posteriores de compilación—como generación de código u optimización—y refuerza las buenas prácticas de separación de responsabilidades y diseño modular en un compilador.

Repositorio:

https://github.com/ELJuanP/Seminario_de_Traductores_de_lenguajes_II/tree/8de131e1801ef3ca8cc06a8dabb0cd72dfdcaf00/Practica%203