

EDUCACIÓN



INSTITUTO TECNOLÓGICO NACIONAL DE MÉXICO EN CELAYA

Materia: Lenguajes y Automatas II

Maestro: ISC. Ricardo González González

Alumnos:

Isacc Salvador Bravo Estrada 2003048

Guillermo Peasland Aguilar 20030737

Maria del Carmen Chávez Patiño

20030296

Luis Fernando Mendoza Jaudera

1930536

"Actividad 6"

Fecha de entrega: 15 de Octubre
de 2024

EQUIPO N°. 3



DEPARTAMENTO DE SISTEMAS COMPUTACIONALES E INFORMÁTICA

ASUNTO: **SOLICITUD DE ACTIVIDADES**

Celaya, Guanajuato, 07 /octubrebre / 2024

LENGUAJES Y AUTÓMATAS II

DOCENTE DESIGNADO: ISC. RICARDO GONZÁLEZ GONZÁLEZ

SEMESTRE AGOSTO-DICIEMBRE 2024

ACTIVIDAD 6 (VALOR 44 PUNTOS)

LEA CUIDADOSAMENTE, Y REALICE LAS SIGUIENTE ACTIVIDADES, CONSIDERANDO LOS CRITERIOS DE CALIDAD PROPUESTOS EN LOS DOCUMENTOS DE LA [GUÍA TUTORIAL](#), Y LA [RÚBRICA DE EVALUACIÓN](#),

EL LECTOR DEBE TOMAR MUY EN CUENTA QUE ESTA ACTIVIDAD ES UN EXAMEN, Y NO UNA SIMPLE TAREA, PUES DEMANDA DEDICACIÓN PARA INVESTIGAR, LEER, ANALIZAR, REDACTAR, ILUSTRAR Y PROPOSER DE MANERA PROFESIONAL LOS TEMAS PROPUESTOS EN LA ESTRUCTURA TEMÁTICA DE ESTA ASIGNATURA.

4. ANÁLISIS SEMÁNTICO.

INVESTIGUE, LEA, COMPREnda Y ELABORE UNA **MONOGRAFÍA TÉCNICA** COMPLETAMENTE APEGADA A LO SOLICITADO EN LA [GUÍA TUTORIAL](#) (PUNTO 3, INCISO a) ACERCA DE LOS SIGUIENTES TEMAS :

- TEMA 4.1 ÁRBOLES DE EXPRESIONES.
- TEMA 4.2 ACCIONES SEMÁNTICAS DE UN ANALIZADOR SINTÁCTICO.
- TEMA 4.3 COMPROBACIONES DE TIPOS EN EXPRESIONES.
- TEMA 4.4 PILA SEMÁNTICA EN UN ANALIZADOR SINTÁCTICO.
- TEMA 4.5 ESQUEMA DE TRADUCCIÓN.
- TEMA 4.6 GENERACIÓN DE LA TABLA DE SÍMBOLOS Y DE DIRECCIONES.
- TEMA 4.7 MANEJO DE ERRORES SEMÁNTICOS.

CONSIDERACIÓN :

DEBE USTED ENTENDER EL VALOR QUE TIENE ESTA ACTIVIDAD Y QUE LOS TEMAS ANTES REFERIDOS, PARA NADA DEBEN SER ABORDADOS COMO SIMPLES CONCEPTOS REDACTADOS CON LA LIGEREZA, PUES ESTA ACTIVIDAD ESTÁ CONSIDERADA COMO UN EXAMEN.

ANALICE CADA TEMA, SUS CARACTERÍSTICAS, SU IMPORTANCIA, SUS CONCEPTOS, SUS EJEMPLOS, SUS ILUSTRACIONES, Y LOS TIPOS DE EVIDENCIAS QUE USARÁ PARA DEMOSTRAR QUE USTED HA ADQUIRIDO UN VERDADERO CONOCIMIENTO ACERCA DE ÉSTOS.





A MODO DE PRÁCTICAS REALICE ESTE PUNTO Y ELABORE EJERCICIOS NECESARIOS CON LOS CUÁLES USTED DEMUESTRE

- **ELABORE DOS VIDEOS** (AL MENOS DE 30 MINUTOS) DISTRIBUIDOS DE LA SIGUIENTE FORMA Y EN LOS QUE EXPONGA SUS CONOCIMIENTOS ADQUIRIDOS. DESPUÉS COLOQUE SUS MATERIALES EN YOUTUBE E INCLUYA LAS LIGAS EN SU EXAMEN.

VIDEO 1 : TEMAS 4.1, 4.2, 4.3, 4.4

VIDEO 2 : TEMAS 4.5, 4.6, 4.7

MUY IMPORTANTE: SI ESTA ACTIVIDAD ES ENTREGADA EN EQUIPO, CADA UNO DE LOS INTEGRANTES DE ÉSTE DEBEN PARTICIPAR EN CADA VIDEO, EXPONIENDO JUNTO A SUS COMPAÑEROS CADA TEMA SOLICITADO.

POR FAVOR NO USE APUNTADORES O MATERIALES DE APOYO TAN SOLO LEER LOS CONCEPTOS. LA IMPORTANCIA Y EL VALOR DE LOS VIDEOS RADICA EN EXPRESAR Y EVALUAR CORRECTAMENTE SU CONOCIMIENTO EN ESTOS TEMAS.

IMPORTANTE: SI LO REQUIERE PUEDE CONSULTAR EL [SIGUIENTE DOCUMENTO](#) PARA ORIENTAR SU TRABAJO EN CONOCER QUÉ ES Y CÓMO HACER UNA MONOGRAFÍA CON EL RIGOR ACADÉMICO REQUERIDO.

POR ÚLTIMO, RECUERDE LEER LA [GUÍA TUTORIAL](#) PARA EL CORRECTO TRATAMIENTO DE ESTE INCISO.

¿ QUÉ SE CALIFICARÁ ?

LA RÚBRICA PARA EVALUAR ESTA ACTIVIDAD ESTARÁ INTEGRADA POR LOS SIGUIENTES CRITERIOS.

- a. **LA OPORTUNIDAD.** SI EL TRABAJO FUE ENTREGADO OPORTUNAMENTE.
- b. **LA COMPRENSIÓN.** SE VALORARÁ EL GRADO DE COMPRENSIÓN DEL TEMAS ANALIZADOS.
- c. **LA CALIDAD.** SI LAS EVIDENCIAS ENVIADAS CORRESPONDEN A LA CALIDAD ESPERADA PARA ESTE NIVEL PROFESIONAL QUE SE CURSA.
- d. **LA CAPACIDAD DE SÍNTESIS.** SI LAS EVIDENCIAS ENTREGADAS TIENEN EL NIVEL DE DETALLE Y PROFUNDIDAD REQUERIDA, O EN BIEN SI SE OMITIERON CONCEPTOS CON EL AFÁN DE SIMPLIFICAR Y ENTREGAR UN MATERIAL ACADÉMICA Y TÉCNICAMENTE POBRE.
- e. **LA CREATIVIDAD.** LA MANERA EN QUE SE EXPRESAN LOS CONCEPTOS Y EL TRATAMIENTO QUE SE DA A LA INFORMACIÓN ANALIZADA PARA QUE ÉSTA SEA COMPRESIBLE EN SU ESENCIA.

IMPORTANTE : CUENTA CON EL TIEMPO SUFFICIENTE PARA REALIZAR ESTA ACTIVIDAD Y SUMAR PUNTOS IMPORTANTES A SU CALIFICACIÓN DE ESTA EVALUACIÓN.

IMPORTANTE : TODO EL MATERIAL ESCRITO DEBERÁ SER HECHO A MANO.





CONSIDERACIONES.

CADA UNO DE LOS PUNTOS ANTERIORES DEBE SER DESARROLLADO CON LA PROFUNDIDAD ACORDE A UN NIVEL PROFESIONAL, Y APEGÁNDOSE COMPLETAMENTE A LAS DIRECTRICES DE LA GUÍA TUTORIAL.

NO CONCIBA ESTE TRABAJO, COMO UN SIMPLE RESUMEN O EJERCICIO DE TRANSCRIPCIÓN, PUES EL VALOR INDICADO AL INICIO DE ESTA ACTIVIDAD LE DARÁ A USTED UNA BUENA IDEA DE LO QUE SE ESPERA DE ELLA, EN CUANTO A CALIDAD Y EL APRENDIZAJE OBTENIDO, MISMO QUE SERÁ PUESTO A PRUEBA MEDIANTE UN EXAMEN ESCRITO O BIEN ORAL EN CLASE.

SI DECIDIÓ ELABORAR ESTA ACTIVIDAD EN EQUIPO, CADA INTEGRANTE DE ÉSTE DEBERÁ POSEER EL MISMO NIVEL DE CONOCIMIENTO, PUES TAN SOLO REPARTIR TEMAS ENTRE LOS INTEGRANTES DEL EQUIPO, SUPONDRÍA UN GRAVE ERROR DE INTERPRETACIÓN A LA INTENCIÓN DIDÁCTICA REAL DE ESTA ACTIVIDAD.

POR ÚLTIMO, ESTA ACTIVIDAD SOLO SE PODRÁ DESARROLLAR EN EQUIPO, SI SE REGISTRÓ EN UNO PREVIAMENTE, UTILIZANDO EL FORMATO ENTREGADO EN LA ACTIVIDAD INICIAL. DE LO CONTRARIO DEBERÁ ELABORAR Y ENTREGAR LA ACTIVIDAD DE FORMA INDIVIDUAL.

LA ENTREGA DE DICHO REGISTRO SE HARÁ VÍA CORREO ELECTRÓNICO ENVIANDO ÉSTE AL PROFESOR DESIGNADO, Y POSTERIORMENTE EN CLASE ENTREGANDO LA HOJA EN FÍSICO.

OBSERVACIONES:

- CADA HOJA QUE ENTREGUE DE SU ACTIVIDAD, DEBERÁ ESTAR FIRMADA AL MARGEN DERECHO, INCLUIDA LA PROPIA SOLICITUD DE LA ACTIVIDAD.
- INTEGRE TODO SU TRABAJO EN UN SOLO ARCHIVO DE TIPO .PDF, Y ASIGNE EL NOMBRE QUE A CONTINUACIÓN SE INDICA.

NO OLVIDE ANEXAR LAS HOJAS DE ESTA ACTIVIDAD Y DE SU TRABAJO DESPUÉS DE SU PORTADA.

- UNA VEZ ELABORADA SU ACTIVIDAD, RECUERDE DIGITALIZARLA Y NOMBRARLA EN BASE A LA NOMENCLATURA QUE SE INDICA MÁS ADELANTE EN ESTE DOCUMENTO.
- SI SUS EVIDENCIAS ENVIADAS POR CORREO, NO CUMPLEN CON LA NOMENCLATURA SOLICITADA, NO SERÁN CONSIDERADAS COMO EVIDENCIAS PARA SU EVALUACIÓN.
- POR ÚLTIMO, POR FAVOR GESTIONE APROPIADAMENTE SU TIEMPO, Y SEA PUNTUAL EN SU ENTREGA Y ASÍ EVITAR PROBLEMAS DE NULIDAD POR EXTEMPORANEIDAD.



LA NOMENCLATURA SOLICITADA PARA ENVIAR SU TRABAJO ES LA SIGUIENTE :

AAAA-MM-
DD_TNM_CELAYA_MATERIA_DOCUMENTO_[EQUIPO]_NOCTROL_APELLIDOS_NOMBRE_SEM.PDF

(NOTA : * TODO DEBE SER ESCRITO USANDO LETRAS MAYÚSCULAS ***)**

DONDE :

TNM_CELAYA	:	INSTITUCIÓN ACADÉMICA
AAAA	:	AÑO
MM	:	MES
DD	:	DÍA
MATERIA	:	LAI _{II} , LI MÁS EL GRUPO (-A , -B, -C)
DOCUMENTO	:	A1-ACTIVIDAD 1, P1-PRACTICA 1, R1-REPORTE 1, T1-TAREA 1, PG1-PROGRAMA, ETC. (CAMBIANDO EL NÚMERO CONSECUТИVO POR EL QUE CORRESPONDA)
[EQUIPO]	:	NÚMERO DEL EQUIPO QUE CORRESPONDA SEGÚN INDICACIÓN DEL PROFESOR. [OPCIONAL]
NOCTROL	:	SU NÚMERO DE CONTROL
APELLIDOS	:	SUS APELLIDOS
NOMBRE	:	SU NOMBRE
SEM	:	EL PERIODO SEMESTRAL EN CURSO: AGO-DIC

EJEMPLO :

SI EL TRABAJO SE SOLICITÓ EN EQUIPO.

2024-10-07_TNM_CELAYA_LAI_{II}-A_A6_EQUIPO_99_9999999_PEREZ_PEREZ_JUAN_AGO-DIC24.PDF

DONDE EL NOMBRE DEBERÁ CORRESPONDER AL JEFE DE EQUIPO QUE HACE LA ENTREGA DEL TRABAJO.

SI EL TRABAJO SE SOLICITÓ INDIVIDUALMENTE.

2024-10-07_TNM_CELAYA_LAI_{II}-A_A6_9999999_PEREZ_PEREZ_JUAN_AGO-DIC24.PDF





FECHA Y HORA DE ENTREGA:

LA INDICADA EN LA PLATAFORMA VIRTUAL.

EN CASO DE QUE EL TRABAJO SE HAYA SOLICITADO EN EQUIPO, EL JEFE DEL MISMO SERÁ EL ÚNICO RESPONSABLE DE ENVIAR LA ACTIVIDAD EN LA PLATAFORMA VIRTUAL.

MUY IMPORTANTE:

1. DESPUÉS DE LA HORA INDICADA EN LA PLATAFORMA VIRTUAL (AÚN CUANDO SOLO SEA UN MINUTO O VARIOS), LA ACTIVIDAD SERÁ CONSIDERADA COMO EXTEMPORÁNEA Y NO CONTARÁ COMO EVIDENCIA PARA SU EVALUACIÓN.

SE LE SUGIERE ENVIAR CON ANTICIPACIÓN SU ACTIVIDAD A FIN DE EVITAR CONFLICTOS POR NO ENTREGAR ÉSTA A TIEMPO.

BAJO NINGÚN PRETEXTO O JUSTIFICACIÓN SE ACEPTARÁN LOS TRABAJOS EXTEMPORÁNEOS, EVITE LA PENA DE RECORDAR A USTED QUE EL VALOR DE LA PUNTUALIDAD ES PARTE IMPORTANTE DE SUS EVIDENCIAS Y ES EL PRIMER PUNTO QUE SE HA DE EVALUAR.

2. NO OLVIDE ANEXAR A SU ARCHIVO .PDF DE EVIDENCIAS UNA PORTADA PROFESIONAL, Y ESTA SOLICITUD DE ACTIVIDADES CON TODAS LAS HOJAS FIRMADAS EN EL MARGEN DERECHO.
3. POR ÚLTIMO, TODA EVIDENCIA GENERADA QUE CONTENGA AL MENOS UNA TRANSCRIPCIÓN DE CUALQUIER FUENTE Y DE CUALQUIER TIPO, ES DECIR CON MATERIAL PLAGIADO SERÁ ANULADA DE FORMA INCONTROVERTIBLE.



EDUCACIÓN



INSTITUTO TECNOLÓGICO NACIONAL DE MÉXICO EN CELAYA

Materia: Lenguajes y Automatas II

Maestro: ISC. Ricardo González González

Alumnos:

Isacc Salvador Bravo Estrada 2003048

Guillermo Peasland Aguilar 20030737

Maria del Carmen Chávez Patino 20030296

Luis Fernando Mendoza Jaujera 1930556

“Monografía Técnica sobre el
Análisis Semántico”

Fecha de entrega: 15 de Octubre de
2024

EQUIPO NO.3

Índice de Temas

4.1 Árboles de Expresiones

- 4.1.1 Introducción
- 4.1.2 Definición
- 4.1.3 Definición Formal
- 4.1.4 Tipos de árboles de expresiones
- 4.1.5 Uso en Análisis de Lenguajes Formales
- 4.1.6 Métodos para construir un Árbol de Expresión
- 4.1.7 Ejemplos
- 4.1.8 Optimización
- 4.1.9 Beneficios
- 4.1.10 Conclusión

4.2 Acciones Semánticas de un Analizador Sintáctico

- 4.2.1 Introducción
- 4.2.2 Contexto teórico
- 4.2.3 Acciones Semánticas
- 4.2.4 Integración en el Proceso de Parsing
- 4.2.5 Tipos de Acciones Semánticas
- 4.2.6 Implementación Práctica
- 4.2.7 Aplicación y Beneficios
- 4.2.8 Desafíos en Diseño de Acciones Semánticas
- 4.2.9 Conclusión

4.3 Comprobaciones de Tipos en Expresiones

- 4.3.1 Introducción
- 4.3.2 Características
- 4.3.3 Importancia
- 4.3.4 Sistemas de Tipos
- 4.3.5 Expresiones de Tipos

4.3.6 Errores comunes de Tipos

4.3.7 Conclusión

4.3.8 Ejemplo Practico

4.4 Pila Semántica en un Analizador Sintáctico

4.4.1 Introducción

4.4.2 ¿Qué es la pila semántica?

4.4.3 ¿Para que se usa la pila?

4.4.4 Características

4.4.5 ¿Cómo funciona la pila semántica?

4.4.6 ¿Cuál es su objetivo Teórico?

4.4.7 Errores semánticos y su manejo

4.4.8 Conclusión

4.4.9 Ejemplo Practico

4.5 Esquema de Traducción

4.5.1 Introducción

4.5.2 Componentes fundamentales del esquema de traducción

4.5.3 Tipos de esquemas de traducción

4.5.4 Fases del esquema de traducción

4.5.5 Técnicas avanzadas en esquemas de traducción

4.5.6 Implementación de esquemas de traducción

4.5.7 Ejemplo detallado de esquema de traducción

4.5.8 Ventajas del esquema de traducción

4.5.9 Conclusión

4.6 Generación de la Tabla de Símbolos y de Direcciones

4.6.1 Introducción

4.6.2 Generación y estructura de la tabla de símbolos

4.6.3 Estructura de la tabla de símbolos

4.6.4 Implementación de la tabla de símbolos

4.6.5 Manejo de colisiones en la tabla de símbolos

4.6.6 Generación y calculo de direcciones de memoria

4.6.7 Importancia de la tabla de símbolos y el calculo de direcciones

4.6.8 Implementación de la tabla de símbolos en un compilador

4.6.9 Conclusión

4.7 Manejo de Errores Semánticos

4.7.1 Introducción

4.7.2 Definición

4.7.3 Importancia del análisis semántico

4.7.4 Tipos de errores semánticos

4.7.5 Técnicas de detección

4.7.6 Estrategias de manejo de errores semánticos

4.7.7 Técnicas de recuperación

4.7.8 Prevención de errores semánticos

4.7.9 Errores semánticos y autómatas

4.7.10 Conclusión

Árboles de Expresiones

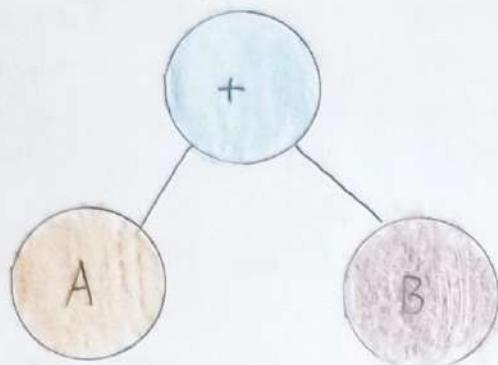
Introducción

En el campo de la informática y la teoría de la computación, los árboles de expresiones son estructuras de datos fundamentales que permiten representar expresiones matemáticas, lógicas y de programación de forma jerárquica. Cada nodo de un árbol de expresión representa un componente de una expresión, ya sea un operador o un operando, organizandolos según su prioridad y secuencia de evaluación. Esta estructura facilita el análisis, la manipulación y la evaluación de las expresiones, convirtiéndolas en herramientas clave para el desarrollo de compiladores, intérpretes y sistemas de procesamiento de lenguajes.

Derivados de los árboles binarios, los árboles de expresión organizan sus nodos de manera que la raíz y los nodos internos representan operadores, mientras que las hojas contienen operandos. Los compiladores emplean esta estructura para descomponer el código fuente en instrucciones simples, transformando expresiones complejas, como $(a+b)*(c-d)$ en código objeto optimizado que pueda ser ejecutado por un procesador. Gracias a su organización jerárquica, los árboles de expresiones permiten una interpretación clara y eficiente del código, ayudando a mejorar el rendimiento y la eficiencia de los programas.

Definición

Un árbol de expresiones es una estructura de datos en forma de árbol que representa de manera gráfica una expresión matemática, lógica o de programación. Cada nodo del árbol corresponde a un elemento de la expresión y la estructura del árbol refleja la jerarquía de las operaciones.



Definición Formal

Un árbol de expresiones es un grafo acíclico dirigido (DAG, tipo de grafo en el que las aristas tienen dirección y no existen ciclos)

- Cada nodo tiene como máximo 2 hijos
- Los nodos internos representan operadores
- Los hojas representan operandos
- La raíz representa la operación principal de la expresión

Notación

- **Nodo:** representado por un círculo
- **Arista:** representa la relación entre nodos (Páter e hijos)

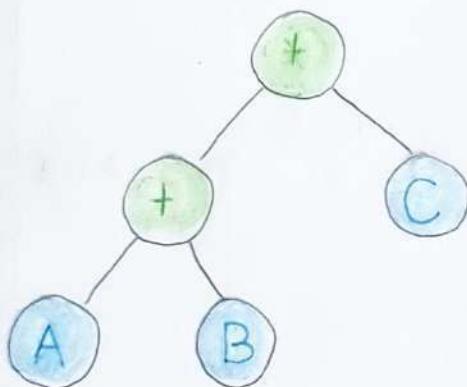


Componentes:

- **Nodos:** Son los elementos fundamentales de l árbol. Cada nodo tiene un operador o un operando.
- **Operadores:** Nodos internos que especifican la operación a realizar (sumas, restas, multiplicación, etc)
- **Operandos:** Nodos hoja que representan los valores en los cuales se realizan las operaciones.
- **Hojas:** Nodos que no tienen hijos y representan los operandos de las expresión.

Ejemplo Visual:

Consideremos la expresión aritmética $A + B * C$. Su representación como árbol de expresión sería:



Raíz: El operador de multiplicación (*) es la operación principal.

Nodos internos: El operador de suma (+) es un nodo interno.

Hojas: A, B y C son las hojas, representando los operandos.

Explicación

- La multiplicación (*) tiene mayor prioridad que la suma (+), por lo que la multiplicación se realiza primero
- El subárbol izquierdo representa la suma de A y B
- El subárbol derecho representa el operando C

Tipos de árboles de expresiones

Tipo de árbol de Expresión	Propósito Principal	Operadores Típicos	Ejemplo	Diferencia Clave
Aritmética	Evaluar expresiones matemáticas	+,-,*,/,%	(2+3)* 4	Se enfoca en operaciones numéricas.
Lógica	Evaluar expresiones booleanas	AND, OR, NOT ==, !=, >	(A == B) OR NOT C	Se enfoca en operaciones lógicas (True or False)
Lenguajes de programación	Representar expresiones en lenguajes de programación	Algoritmos, lógicos, de comparación, asignación, llamadas a funciones.	X = Y + f (a,b)	Más general, puede incluir cualquier tipo de expresiones

Uso en Análisis de Lenguajes Formales

Los árboles de expresión son herramientas fundamentales en el análisis de lenguajes formales. Sirven como una representación interna de las estructuras sintácticas de un lenguaje.

- **Análisis sintáctico:** Los compiladores utilizan árboles de expresión para representar la estructura jerárquica de las expresiones de un programa. Cada nodo del árbol corresponde a una regla gramatical y las hojas representan los tokens del lenguaje.
- **Análisis Semántico:** Una vez construido el árbol de expresión, se pueden realizar análisis semánticos para verificar la coherencia de la expresión, como la comprobación de tipos y la resolución de nombres.

Aplicación en Autómatos

- **Autómatas de pila:** Los autómatas de pila son máquinas de estado que utilizan una pila para reconocer lenguajes libres de contexto. Los árboles de expresión pueden ser utilizados para representar las configuraciones de la pila de un autómata de pila durante el reconocimiento de una cadena.
- **Validación de expresiones:** Los árboles de expresión pueden ser utilizados para validar si una cadena pertenece a un lenguaje definido por una gramática. Al construir el árbol correspondiente a la cadena y verificar si cumple con las reglas de la gramática, se puede determinar su validez.

Relación con Expresiones regulares

- **Evolución de expresiones regulares:** Las expresiones regulares pueden ser representadas con árboles de expresión. Cada operador de la expresión regular corresponde a un nodo del árbol.
- **Automatos finitos:** Los árboles de expresión pueden ser utilizados para construir automatos finitos equivalentes a una expresión regular dada.

Métodos para construir un Árbol de Expresión

La construcción de un árbol de expresión a partir de una expresión matemática o lógica implica transformar una expresión de una estructura jerárquica. Existen diversos métodos para lograrlo, pero todos comparten el objetivo de capturar la estructura sintáctica de la expresión.

Algoritmo básico de construcción

Este método suele seguir estos pasos:

- 1.- **Tokenización:** Dividir la expresión en tokens individuales (operadores, operandos)
- 2.- **Análisis Sintáctico:** Aplicar reglas gramaticales para reconocer la estructura de la expresión y construir un árbol de análisis sintáctico.
- 3.- **Creación del árbol de expresión:** Transformar el árbol de análisis sintáctico en un árbol de expresión, asignando a cada nodo un operador o un operando.

Ejemplos

1- Consideremos la expresión $(A+B)*C$

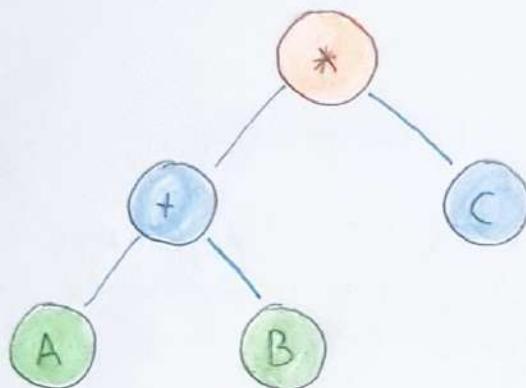
1- Tokenización:

- Tokens: (, A, +, B,), *, C

2- Análisis sintáctico

- Se reconoce que la expresión es una multiplicación donde el primer operando es una suma.

3- Creación del árbol



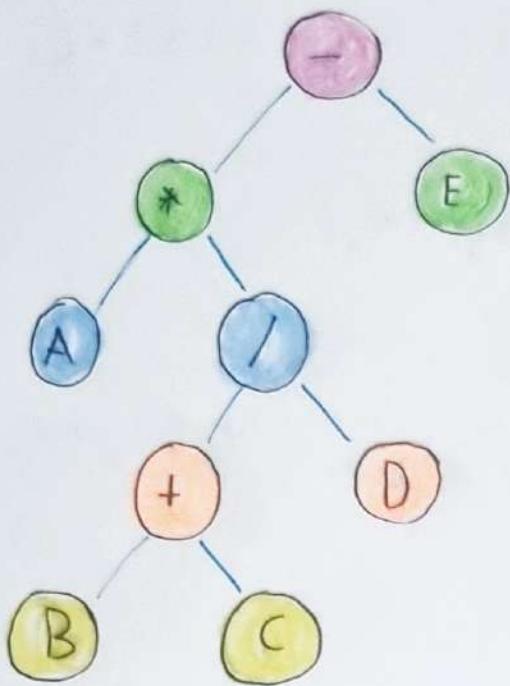
Ejemplo 2

Consideramos la expresión más compleja
 $A * (B + C / D) - E$

1- Tokenización: A, *, (, B, +, C, /, D,), -, E

2- Análisis sintáctico: se reconoce diversas operaciones

3.- Creación del árbol



Consideraciones adicionales

- **Prioridad de operadores:** El algoritmo de construcción debe tener en cuenta la precedencia de los operadores para construir el árbol correctamente.
- **Asociativa:** La asociatividad de los operadores (izquierdo a derecho) determina cómo se agrupan los operandos cuando hay múltiples operadores del mismo tipo.
- **Ambigüedades:** Algunas gramáticas pueden ser ambiguas, lo que significa que una misma cadena de entrada puede tener múltiples árboles de análisis sintáctico. En estos casos se pueden utilizar técnicas de resolución de ambigüedades.

Ejemplo 3

Consideramos la siguiente expresión

$$a * (a + c) + a * (b + c)$$

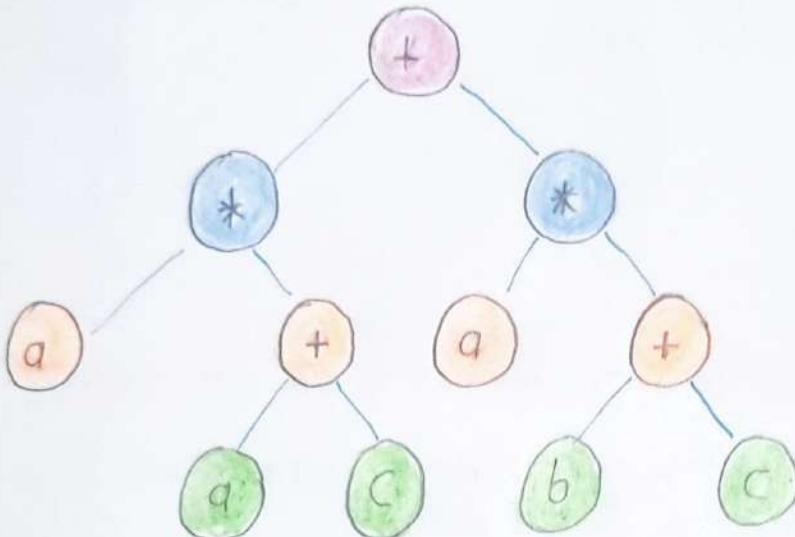
1.- Tokenización

$$a, *, (, a, +, c,), +, a, *, (, b, +, c,), .$$

2.- Análisis sintáctico:

La expresión sigue la gramática de las expresiones aritméticas, donde los operadores ($*$, $+$) tienen una precedencia y asociatividad definidas.

3.- Árbol de expresiones.



Optimización

Se examina la expresión para identificar

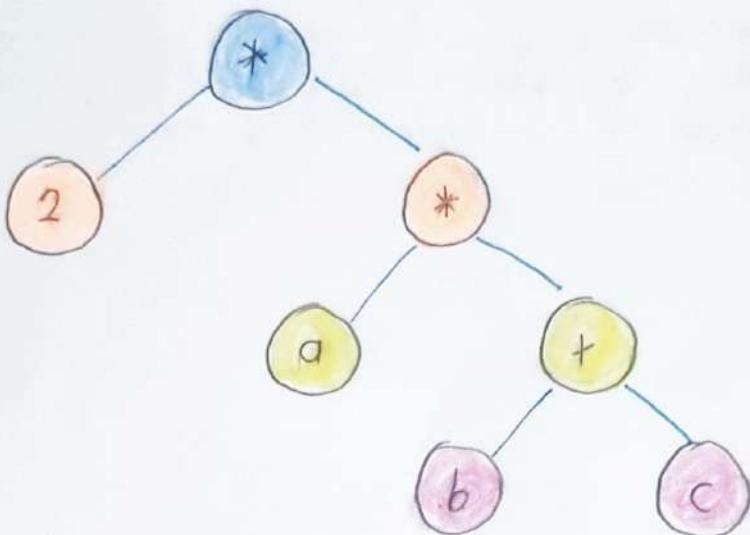
Subexpresiones comunes, en este caso $a * (b + c)$

aparece 2 veces.

Al identificar la subexpresión repetida se procede a factorizar para simplificar el cálculo.

- Factoriza $a * (b + c)$ da como resultado
 $2 * a * (b + c)$

el arbol que dara de la siguiente forma



Resultados

El uso de árboles de expresión en este caso ha permitido:

- Identificar subexpresiones comunes; facilitando la optimización del código.
- Visualizar la estructura de la expresión; Permitiendo un análisis más detallado y la aplicación de diversos técnicas de optimización.
- Generar código más eficiente; Reduciendo el número de operaciones y mejorando el rendimiento.

Beneficios

Los árboles de expresiones ofrecen una representación visual y estructurada lo cual genera una representación clara, así facilitando el análisis semánticos como la comprobación de tipos y resolución de nombres.

También permitiendo aplicar diversas técnicas de optimización, como la eliminación de subexpresiones comunes, la propagación de constantes y la simplificación de expresiones.

Conclusión

Los árboles de expresión son una herramienta fundamental en la construcción de compiladores. Al representar de manera jerárquica las expresiones matemáticas y lógicas, facilitando tareas como el análisis sintáctico, la optimización de código.

4.2 Acciones Semánticas de un analizador sintáctico.

4.2.1 Introducción

En el proceso de compilación, el análisis sintáctico se encarga de validar las estructuras de un programa en base a reglas gramaticales predefinidas. Sin embargo, la validación estructural no es suficiente para asegurar la corrección del programa. Aquí entran en juego las acciones semánticas.

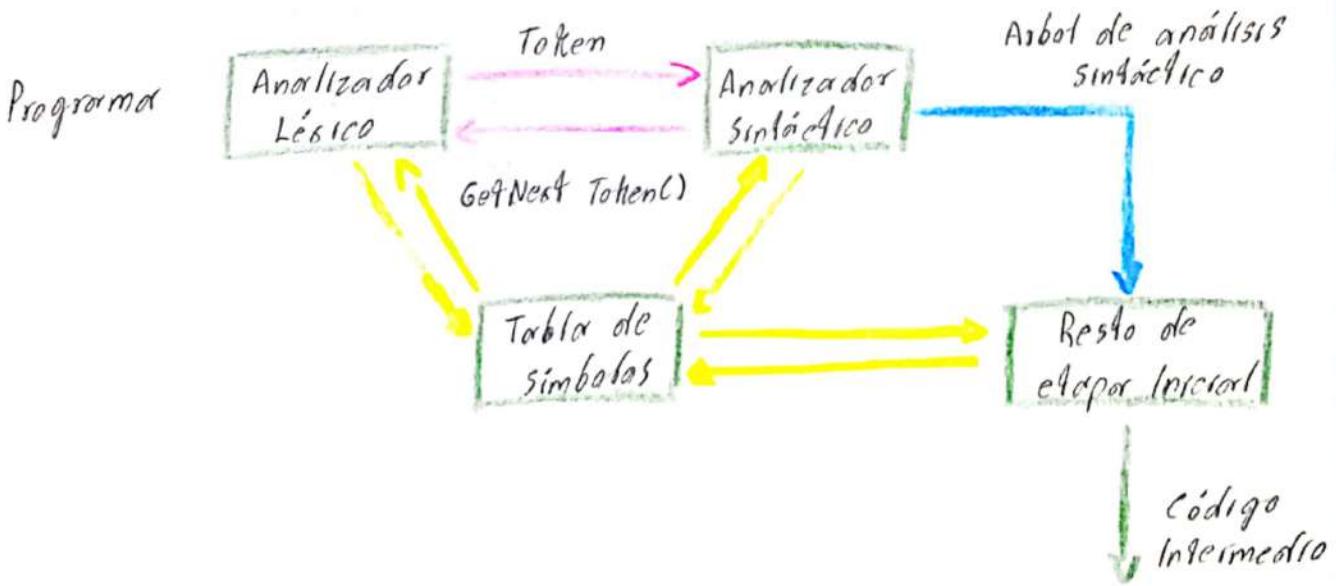
Las acciones semánticas son tareas que se llevan a cabo durante el análisis sintáctico para capturar información sobre el significado del código. Con estas acciones se permite al compilador verificar y traducir el significado de las estructuras sintácticas, facilitando otras tareas como la generación de código intermedio, comprobación de tipos y la creación de representaciones intermedias.

4.2.2 Contexto teórico

El análisis sintáctico, conocido como parsing también, es la fase del proceso de compilación donde se analizan las estructuras de un programador siguiendo las reglas de una gramática formal. Este recibe una secuencia de tokens producidas por el analizador léxico, y verificar que formen construcciones válidas según la gramática.

Existen dos tipos de analizadores sintácticos:

- Descendentes: Se construye el árbol de derivación desde la raíz hasta las hojas, usando reglas gramaticales predictivas.
- Ascendentes: Se construye el árbol de derivación a partir de las hojas, combinando subexpresiones para formar expresiones más grandes.



4.2.3 Acciones Semánticas

Las acciones semánticas son tareas o ejecutar en puntos específicos del proceso de análisis sintáctico para capturar y procesar información semántica. Es así como estas acciones permiten el realizar tareas como la construcción de representaciones intermedias, verificación de tipos, gestión de variables y la preparación para la generación de código.

Dicho de forma más simple, las acciones semánticas permiten traducir la estructura sintáctica en una representación semántica comprensible, esto siendo esencial para que las máquinas puedan comprender el texto.

Algunos ejemplos de acciones semánticas:

- Creación de un Árbol de Sintaxis Abstracta (AST): Estructuras que representan la jerarquía de expresiones en un programa.
- Manejo de Tabla de Símbolos: Estructuras de datos que rastrear variables y funciones declaradas en el programa.
- Comprobación de Tipos: Validar que operaciones y expresiones involucren tipos de datos compatibles.
- Generación de Código Intermedio: Produce una versión intermedia del código que será posteriormente optimizado y convertido en código objeto.

4.2.4 Integración en el Proceso de Parsing

Acciones Semánticas en Parsers LL(1)

Los parsers LL(1) son analizadores descendentes, estos procesan la entrada de izquierda a derecha, realizando predicciones sobre las reglas gramaticales o aplicar. Para este tipo de parser, las acciones semánticas se colocan en las producciones de la gramática y se ejecutan inmediatamente cuando se predice una producción.

Ejemplo

$$\text{exp} \rightarrow \text{term} \cdot '+' \text{exp} \{ \$\$ = \$1 + \$3; \}$$

$$\text{term} \rightarrow 'id' \{ \$\$ = \text{valor_de_identificador}(\$1); \}$$

Pasar el ejemplo anterior, cuando se reduce la producción exp, se realiza la suma de los valores semánticos de las subexpresiones. Las acciones semánticas ayudan en la construcción del valor de exp a partir de sus partes.

Acciones Semánticas en Parsers LR(1)

Los parsers LR(1) son analizadores ascendentes, estos construyen el árbol sintáctico conforme van reduciendo las producciones gramaticales. Las acciones semánticas se colocan en las reglas de reducción y se ejecutan al aplicar una reducción.

Ejemplo

$\text{exp} \rightarrow \text{exp} \cdot '1' \text{ term} \{ \$\$ = \$1 + \$3; \}$

$\text{term} \rightarrow \text{id} \{ \$\$ = \text{buscar_en_tabla_simbolos} (\$1); \}$

Pasar el ejemplo anterior, las acciones semánticas se ejecutan cuando el parser reduce las reglas. Cuando se reduce exp , se calcular el valor semántico sumando los valores de exp y term .

4.2.5 Tipos de Acciones Semánticas

Acciones Semánticas en Parsing Ascendente

Las acciones semánticas se ejecutan en el momento en que una producción gramatical es completamente reducida. Este enfoque tiene la ventaja de que el parser ya tiene disponible todo la información necesaria para calcular el valor semántico.

Acciones Semánticas en Parsing Descendente

Las acciones semánticas se ejecutan mientras se están aplicando las reglas gramaticales. Las acciones pueden tener lugar en cualquier punto durante la derivación, complicando su implementación debido a que no se dispone de todo la información en el momento de ejecutar la acción.

4.2.6 Implementación Práctica

Las acciones semánticas se integran en herramientas de parsing como YACC, en estas los desarrolladores definen reglas gramaticales y especifican acciones semánticas por cada regla. En YACC por ejemplo, se puede usar la notación \$\$, \$1, \$2, etc. Con esto se refiere a los valores semánticos de los componentes de la regla.

Ejemplo

```
exp: exp '+' term { $$ = $1 + $3; }  
| term { $$ = $1; };
```

En el ejemplo anterior, la acción semántica sumar los valores de exp y term cuando se reduce la regla $\text{exp} \rightarrow \text{exp} + \text{term}$

4.2.7 Aplicación y Beneficios

Las acciones semánticas son importantes para generar representaciones intermedias y facilitar el análisis estático, como la comprobación de tipos y optimización de código. También permiten el generar código eficiente y evitar errores durante la compilación. Como ejemplo, sin las acciones semánticas, un programa puede ser sintácticamente correcto, pero semánticamente erróneo, como cuando se intenta sumar un carácter de texto con un número.

4.2.8 Desafíos en Diseño de Acciones Semánticas

Cuando se diseñan acciones semánticas, surgen algunos desafíos como:

- Dependencias de información: hay acciones que requieren información que aún no está disponible en el punto donde se ejecutan.
- Compatibilidad de tipos: se refiere a asegurar que las acciones semánticas no generen errores de tipo o inconsistencias.

Estrategias de Depuración

Trazado (Tracing)

Consiste en registrar la ejecución de las acciones semánticas y las decisiones tomadas por el analizador sintáctico. Se puede agregar código de depuración dentro de las acciones semánticas para así imprimir mensajes de salida mostrando qué acciones se están ejecutando y los valores que están procesando.

Ejemplo

```
exp: exp `1' term & printf ("sumar de %d y %d\n", $1, $3); $b = $1 + $3;
    | term & printf ("Valor de term: %d\n", $1); $b = $1;
```

Con esto se permite:

- Ver cuándo y cómo se ejecutan las acciones semánticas.
- Observar valores intermedios que las acciones procesan, ayudando a detectar errores en la manipulación de datos o tipos.

Puntos de Interrupción (Breakpoints)

Permiten detener la ejecución del programa en puntos específicos y examinar el estado del sistema, incluidas variables locales y valores de las acciones semánticas.

Si el parser estuviera implementado en un lenguaje como Java, se usaría el depurador integrado en un entorno de desarrollo como IntelliJ. En este se configuran puntos de interrupción en las líneas de código donde se ejecutan las acciones semánticas.

Ejemplo

Se configura un breakpoint en la acción semántica donde se realizar una suma:

```
exp: exp '1' term { $$ = $1 + $3; }
```

Cuando se ejecuta y alcanzar el breakpoint, el depurador permite examinar los valores de \$1 y \$3 antes de ejecutar la acción.

Inspección de la Tabla de Símbolos

La tabla de símbolos es un componente crítico para rastrear las definiciones de variables, funciones y tipos. Errores en la gestión de la tabla de símbolos son comunes, como no declarar una variable antes de su uso o al haber un conflicto de tipos.

Ejemplo

Se pueden agregar instrucciones de depuración para inspeccionar el estado de la tabla de símbolos en momentos clave:

Termo 'id' {

printf("Consultando la tabla de símbolos para el identificador: %s\n", \$1);
\$\$ = buscar_en_Tabla_de_símbolos(\$1);

};

Con esto se puede verificar que el identificador está presente en la tabla y que tiene el tipo correcto antes de realizar operaciones sobre él.

4.2.9 Conclusion

Las acciones semánticas son parte integral de los analizadores sintácticos y juegan un papel crucial para el análisis y traducción de lenguajes. Con esto permiten verificar el significado de las estructuras de un programa, generar código intermedio y garantizar la validez de un programa no solo desde el punto de vista sintáctico, sino también semántico.



Monografía Técnica

Tema 4.3 Comprobaciones de tipos en expresiones.

INTRODUCCIÓN

En el desarrollo de software, es importante la correcta gestión de los tipos de datos porque es esencial asegurar que los programas funcionen correctamente. Las comprobaciones de tipos en expresiones hace referencia a los mecanismos utilizados por los compiladores e intérpretes para así verificar que las operaciones realizadas sobre las variables y constantes sean compatibles en cuanto a sus tipos. Esta verificación garantiza que no se realicen operaciones inválidas, ya sea como intentar sumar un número con una cadena de texto, ya que esto podría provocar errores en tiempo de compilación o ejecución.

Este proceso no solo previene errores comunes en la programación, sino que también garantiza que la optimización de la ejecución del código pueda permitir que los lenguajes de programación gestionen los recursos de manera mucho más eficiente. A lo largo de este trabajo, se podrá explorar las características, importancia, conceptos clave y ejemplos relacionados con la comprobación de tipos en expresiones, pudiendo brindar una visión mucho más clara de como influye este aspecto en la calidad y seguridad del software.

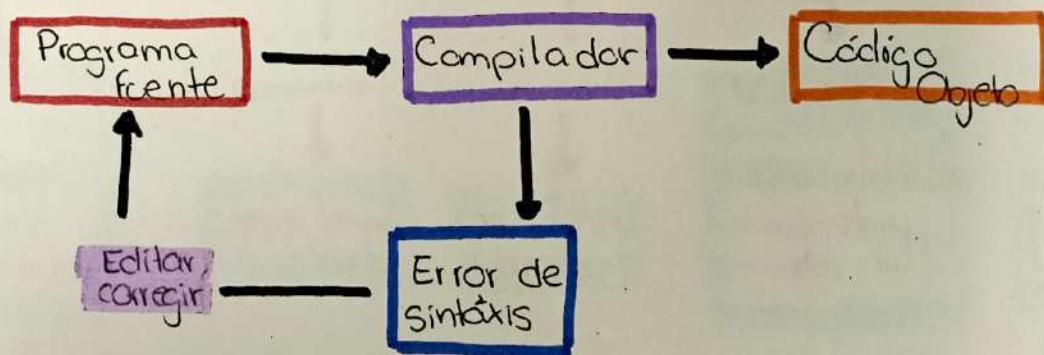


Figura 1. Análisis léxico.



CARACTERISTICAS:

Validación de tipos: Implica verificar que los operadores y operandos en una expresión sean compatibles en cuanto a sus tipos. Por ejemplo: sumar un número entero con un string sería inválido en la mayoría de los lenguajes.

Inferencia de Tipos: En algunos lenguajes, el compilador puede deducir el tipo de una expresión basándose en los operandos.

Coerción: Algunos lenguajes permiten la conversión automática de un tipo a otro (por ejemplo, convertir un entero a Flotante para realizar una operación matemática)

Chequeo en Tiempo de Compilación o Ejecución: Las comprobaciones de tipos pueden realizarse durante la compilación (tiempo de compilación) o durante la ejecución (tiempo de ejecución).

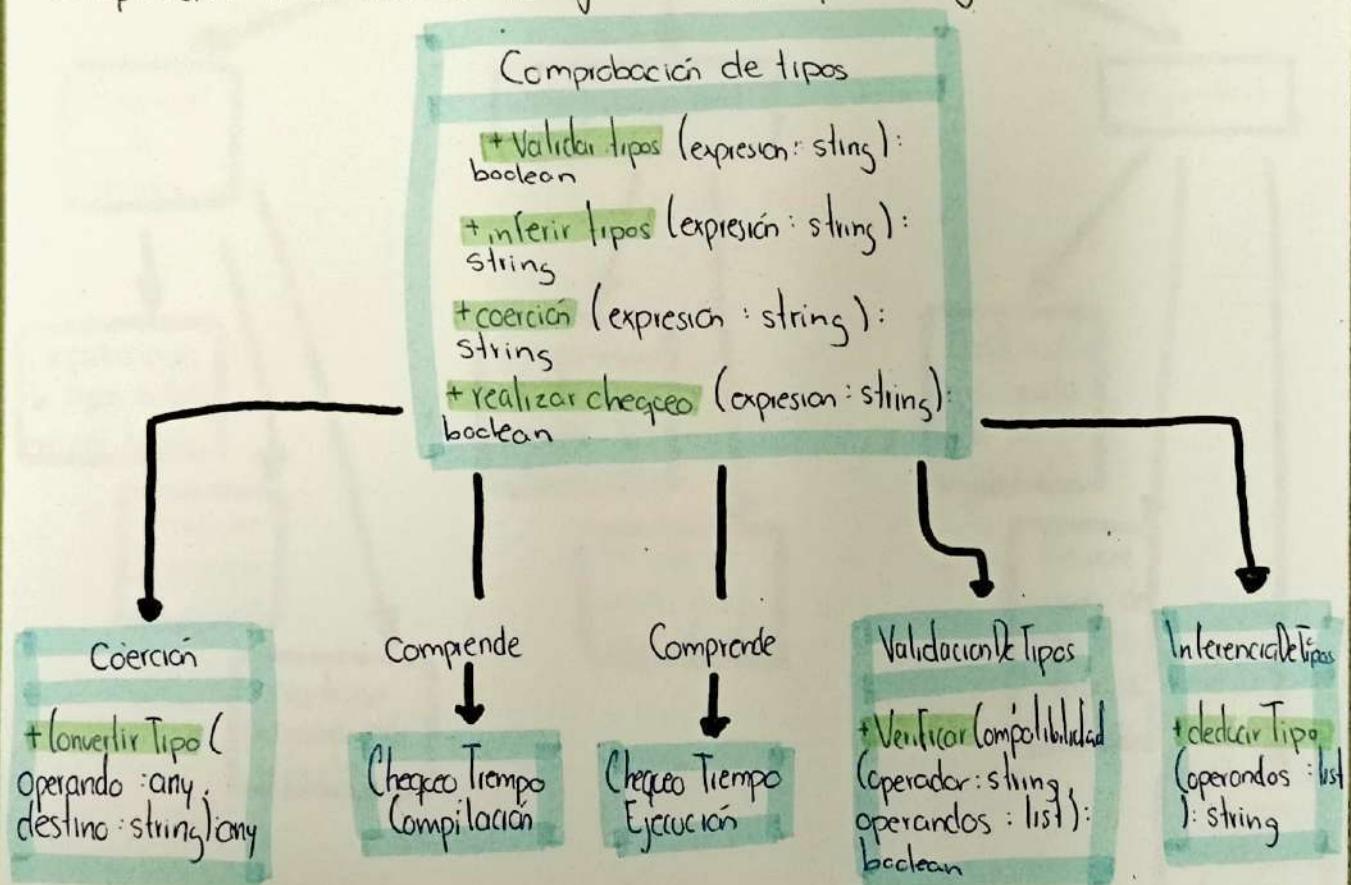


Figura 2. Comprobación de Tipos.



IMPOR TANCIA:

Prevención de errores: Las comprobaciones de tipos ayudan a evitar errores lógicos y de interpretación de datos. Estos garantizan que un programa sea más robusto, al evitar que operaciones no válidas (como sumar una cadena de texto a un número) lleguen a ejecutarse.

Optimización de código: A conocer los tipos de operandos, el compilador puede generar código más eficiente y optimizado para la ejecución.

Seguridad: Para garantizar que los tipos sean consistentes evita posibles fallos de seguridad, como el desbordamiento de memoria o errores de tipo en tiempo de ejecución.

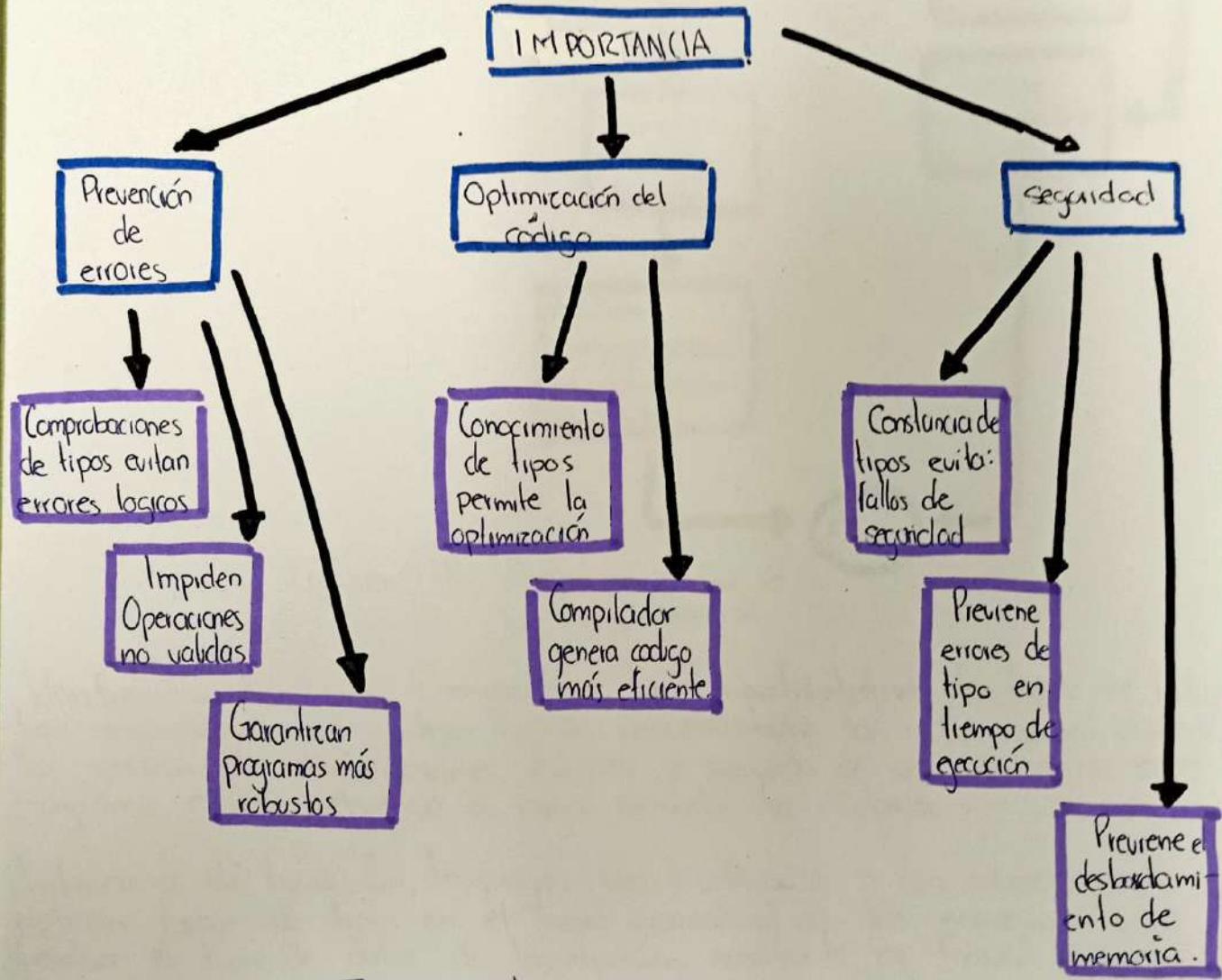


Figura 3. Importancia.



EDUCACIÓN

SECRETARÍA DE EDUCACIÓN PÚBLICA



TECNOLÓGICO
NACIONAL DE MÉXICO



La labor de la comprobación de tipos consiste en conferir a las construcciones sintácticas del lenguaje la semántica de tipificación y en realizar todo tipo de comprobaciones de dicha índole. Por su naturaleza, sin embargo, ésta se encuentra repartida entre la fase de análisis semántico y la generación de código intermedio.

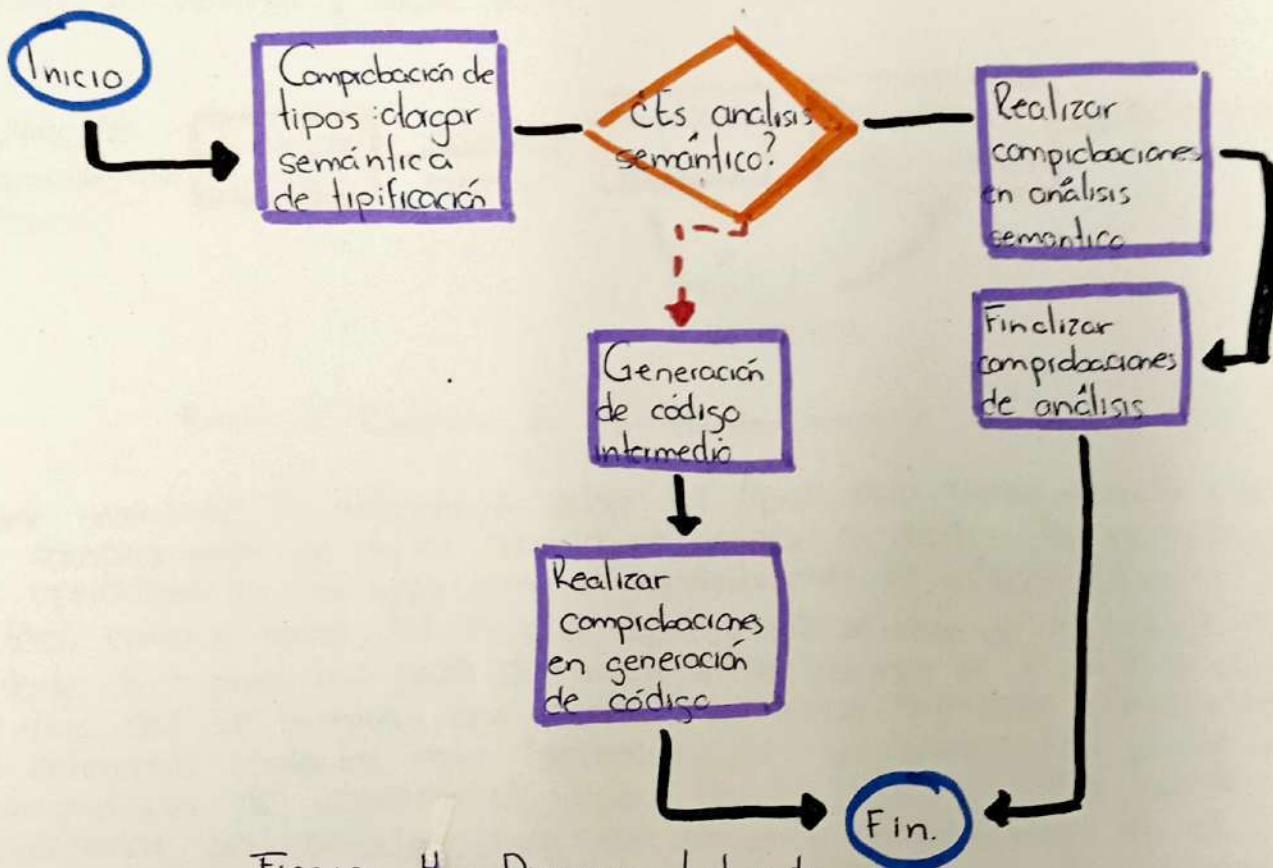


Figura 4. Diagrama Labor de la comprobación

Verificación de tipos: Comprueba la compatibilidad de tipos de todas las expresiones del código fuente recuperando la información durante la gestión de declaraciones. Además se asegura de que no existe en el programa ninguna referencia a ningún símbolo no declarado.

Inferencia de tipos: En lenguajes sin tipificación o con sobrecarga se aplican tareas de tipos en el nivel gramatical de las expresiones para resolver el tipo de datos de la expresión resultante en función del contexto de evaluación.



Un comprobador de tipos se asegura de que el tipo de una construcción coincida con el previsto en su contexto. Por ejemplo: el operador aritmético piedefinido mod en Pascal exige operandos de tipo entero, de modo que un comprobador de tipos debe asegurarse de que los operandos mod tengan tipo entero. De igual manera, el comprobador de tipos debe asegurarse de que la desreferenciación se aplique sólo a un apuntador, de que la indicación se haga sólo sobre una matriz, de que una función definida por el usuario se aplique al número y tipo correctos de argumentos, etcétera.

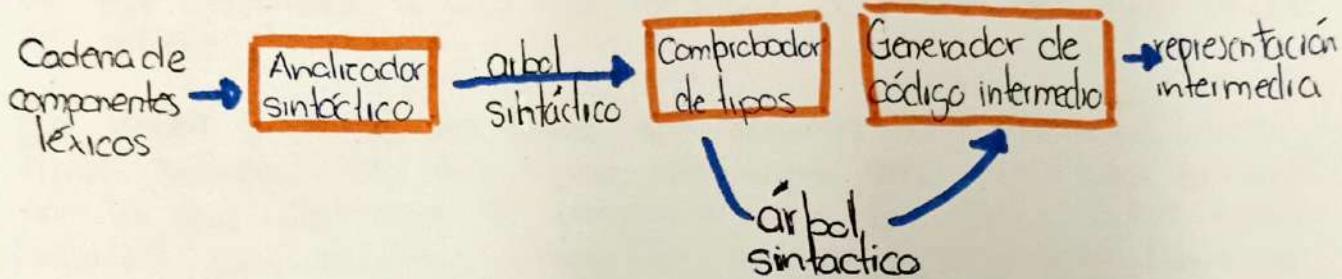


Figura 5. Cadena de componentes léxicos.

Puede necesitarse la información sobre los tipos esta siendo reunida por un comprobador de tipos cuando se genera el código. Por ejemplo, los operadores aritméticos como + normalmente se aplican tanto a enteros como a reales, tal vez otros tipos, y este se debe de examinar el contexto de + para así poder determinar el sentido que se le pretende dar. Se dice que un símbolo que puede representar diferentes operaciones en diferentes contextos está "sobrecargado". La sobrecarga puede ir acompañada de coerción de tipos, donde un compilador puede proporcionar un operador para así convertir un operando en el tipo esperado por el contexto. Una noción diferente de la sobrecarga es la del "polimorfismo". El cuerpo de una función polimórfica puede ejecutarse con argumentos de varios tipos.

Sistemas de tipos

El diseño de un comprobador de tipos para un lenguaje se basa en información acerca de las construcciones sintácticas del lenguaje, la noción de tipos y las reglas para asignar tipos a las construcciones de lenguaje. "Los siguientes extractos del informe de Pascal y del manual de Referencia de C", respectivamente son ejemplos de la información con la que el diseñador de un compilador podría verse obligado a comenzar.



- "Si ambos operandos de los operadores aritméticos de suma, sustracción y multiplicación son de tipo entero, entonces el resultado es de tipo entero"
- "El resultado del operador *asignación* & es un apuntador hacia el objeto al que se refiere el operando. Si el tipo del operando es '...', el tipo del resultado es 'apuntador a ...'."

En los anteriores extractos se encuentra implícita la idea de que cada expresión tiene asociado un tipo. Además, los tipos tienen estructura; el tipo "apuntador a ..." se construye a partir del tipo al que se refiere "...".

En Pascal y en C, los tipos son básicos o construidos. Los tipos básicos son los tipos atómicos sin estructura interna por lo que concierne al programador. En Pascal, los tipos básicos son boolean, character, integer y real. Los tipos de subrango como 1..10, y los tipos enumerados, como (violeta, índigo, azul, verde, amarillo, naranja, rojo) se pueden considerar como tipos básicos. Pascal admite que un programador construya tipos a partir de tipos básicos y otros tipos construidos, como matrices (array), registros (record) y conjuntos (set). Además los apuntadores y las funciones también pueden considerarse como tipos construidos.

Expresiones de tipos.

El tipo de una construcción de un lenguaje denotará esto mediante una "expresión de tipo". De manera informal, una expresión de tipo es, o bien un tipo básico o se forma aplicando un operador llamado constructor de tipos a otras expresiones de tipos. Los conjuntos de tipos y constructores básicos dependen del lenguaje que deba comprobarse. Cada lenguaje de programación requerirá unas expresiones de tipos adecuadas a sus características. A continuación, a modo ejemplo, se definen las expresiones de tipos más comunes:

Tipos Simples

Son expresiones de tipos las más simples del lenguaje y algunos tipos

Integer

Real

Char

Boolean

Void

Error

Figura 6. Tipos Simples.



Los cuatro primeros son los tipos de datos simples más comunes en los lenguajes de programación, los dos últimos son tipos simples especiales que usaremos para su atribución a diversas partes de un programa, a fin de homogenizar el tratamiento de todo el conjunto mediante el método de las expresiones de tipos.

Tomando el lenguaje C como ejemplo, el segmento de código al que está asociada la expresión de tipos integer es aquella en que aparece la **palabra reservada int**, etc.

• **Constructores de tipos**: Permiten formar tipos complejos a partir de otros más simples. La semántica de cada lenguaje tiene asociada unos constructores de tipos propios. En general, en los lenguajes de programación se definen los siguientes constructores:

- **Matrices**: Si T es una expresión de tipos, entonces array (R, T) también una expresión de tipos que representa a una matriz de rango R de elementos de tipo T
- **Productos**: Sea T₁ y T₂ expresiones de tipos, T₁ × T₂ es una expresión de tipos que representa al producto cartesiano de los tipos T₁ y T₂. A fin de simplificar se considera que el constructor u operador de tipos × es asociativo por la izquierda.
- **Registros**: Sea un registro formado por los campos u₁, u₂ ... u_N, siendo cada uno de ellos los tipos T₁, T₂ ... T_N, entonces la expresión de tipos asociada al conjunto es: record ((u₁: T₁) × (u₂: T₂) × ... × (u_N: T_N))
- **Punteros**: Si T es una expresión de tipos, entonces pointer (T) es una expresión de tipos que representa a un puntero a una posición de memoria ocupada por un dato del tipo T.
- **Funciones**: Sean T₁, T₂ ... T_N las expresiones de tipos asociadas a los segmentos de código correspondientes a los argumentos de una función, y sea T el tipo devuelto por la función. Entonces, la expresión de tipos asociada a la función es : ((T₁ × T₂ × ... × T_N) → T)

Las expresiones de tipo pueden contener variables cuyos valores son expresiones de tipos.



Ventajas	Desventajas
Detección temprana de errores: Permite identificar errores antes de ejecutar el programa, reduciendo el tiempo de depuración.	Menor flexibilidad: Los lenguajes con tipos estáticos estrictos pueden limitar la flexibilidad del código.
Optimización del rendimiento: Facilita al compilador generar código más eficiente, mejorando el rendimiento.	Mayor tiempo de desarrollo: La necesidad de especificar tipos puede hacer que el desarrollo sea más lento.
Seguridad: Garantiza que las operaciones entre tipos sean válidas, previniendo fallos en tiempo de ejecución.	Sobrecarga en tiempo de ejecución: Los lenguajes con comprobaciones dinámicas pueden reducir el rendimiento en el programa.
Claridad en el código: La definición explícita de tipos hace que el código sea más fácil de leer y mantener.	Conversión de tipos (coerción): Las conversiones automáticas pueden causar errores sutiles y resultados inesperados.

Tabla 1. Ventajas y Desventajas.

Errores Comunes de Tipos:

- Desbordamiento de tipo: Por ejemplo, almacenar un número demasiado grande en un entero.
- Conversión implícita de tipos: Cuando el lenguaje convierte un tipo a otro automáticamente, lo que puede generar resultados inesperados.
- Tipos incompatibles: Realizar operaciones entre tipos que no se pueden combinar, como sumar un número o una cadena.



Ejemplo:

Tenemos la siguiente expresión en un programa:

```
int a = 5;  
float b = 4.5;  
a = a + b;
```

Comprobación de tipos en esta expresión:

- Declaración de variables: El compilador ya sabe que `a` es de tipo `int` y `b` es de tipo `float`.
- Expresión: En `a + b`, el compilador ve que se están sumando dos variables (`int` y `float`)
 - Regla de conversión: El compilador tiene que asegurarse de que el tipo de la expresión resultante sea compatible con el tipo de `a`. En muchos lenguajes, cuando se suman un `int` y un `float`, el `int` se convierte implícitamente a `float` para realizar la operación, ya que los `float` son más "generales" y pueden almacenar números con decimales. El resultado de `a + b` sería un `float`.
 - Error Potencial: Sin embargo, como `a` es un `int`, después de la operación, el resultado se intentará almacenarlo en `a`, lo que puede causar la pérdida de la parte decimal del resultado. Si el lenguaje no permite este tipo de conversión implícita, podría lanzar un error de tipo, indicando que no se puede asignar un `float` a un `int` sin hacer una conversión explícita.

Solución:

Para evitar errores de tipo, se podría hacer una conversión explícita:

`a = (int)(a + b); //convertir el resultado a int`

O cambiar el tipo de `a` a `float`:

`float a = 5;`



CONCLUSIÓN :

La comprobación de tipos en expresiones es una etapa crucial en el análisis semántico de un compilador. Ya que su objetivo es garantizar que los operadores y operandos sean compatibles en términos de sus tipos de datos. Esto permite evitar errores durante la ejecución de programas. Un sistema de tipos bien diseñado ayuda a identificar posibles inconsistencias, como intentar sumar un número con una cadena de texto, mejorando así la confiabilidad y seguridad del código. Además, la comprobación de tipos puede realizarse de manera estática (en tiempo de compilación) o dinámica (en tiempo de ejecución), dependiendo del lenguaje de programación.

Referencias

- 4.4 Tipos de Datos y Verificación de Tipos. (S/F). Edu.mx. Recuperado el 14 de Octubre de 2024, de http://cidecame.uaeh.edu.mx/lcc/mopo/PROYECTO/libro_32/actacon/autocentido/autocan/44_tipos_de_datos_y_verificacion_de_tipos.html.
- Comprobaciones de tipos en expresiones . (S/F) Genially . Recuperado el 14 de Octubre de 2024, de <https://view.genially.com/6545cf1952e302900127db819/presentation-comprobaciones-de-tipos-en-expresiones>
- (S/f) . Site 123.me . Recuperado el 14 de Octubre de 2024, de <http://5e34473505b1.site.123.me/unidad-i-%C3%81nalysis-sem%C3%A1ntico/13-comprobaciones-de-tipos-en-expresiones>

CARRERA	NOMBRE DE LA ASIGNATURA
INGENIERIA EN SISTEMAS COMPUTACIONALES	LENGUAJES Y AUTOMATAS 2

DOCENTE DESIGNADO
ISC. RICARDO GONZÁLEZ GONZÁLEZ

PRACTICA No.	NOMBRE DE LA PRACTICA	DURACIÓN (HORAS)
14	TEMA 4.3 COMPROBACIONES DE TIPOS EN EXPRESIONES.	1 HORA

1	INTRODUCCIÓN
En la programación, uno de los conceptos fundamentales es el manejo de los tipos de datos. Los lenguajes de programación, como C++, Java o Python, permiten trabajar con diferentes tipos de datos, como enteros, flotantes o cadenas de texto. Cada tipo de dato tiene su propio conjunto de operaciones permitidas y es crucial que el compilador o intérprete del lenguaje se asegure de que los tipos sean compatibles cuando se combinan en expresiones. Esta verificación es lo que se llama comprobación de tipos.	

En esta práctica aprenderemos cómo funcionan las comprobaciones de tipos, qué tipos de conversiones se pueden hacer automáticamente (como convertir un entero a decimal) y qué tipos de combinaciones son inválidas, resultando en errores. Esto es importante para evitar fallos y asegurarnos de que nuestro código sea correcto y funcione como esperamos.

2	OBJETIVO (COMPETENCIAS)
<ul style="list-style-type: none"> ✚ Comprender cómo los compiladores manejan las comprobaciones de tipos al evaluar expresiones. ✚ Identificar errores comunes relacionados con tipos de datos incompatibles. ✚ Distinguir entre conversiones de tipo implícitas (automáticas) y errores de tipo. ✚ Aplicar correctamente los tipos de datos en las expresiones de un programa. ✚ Corregir errores de tipo utilizando métodos apropiados para asegurar que las operaciones sean válidas. 	



3

MARCO TEÓRICO REFERENCIAL

- En un programa, los tipos de datos son las características que definen qué tipo de valor puede tomar una variable y cómo puede interactuar con otras variables. Los principales tipos de datos incluyen:
 - Enteros (int): números sin decimales.
 - Flotantes (float): números con decimales.
 - Cadenas de texto (string): secuencias de caracteres.
- Las comprobaciones de tipos son las reglas que el compilador sigue para asegurarse de que los tipos de datos utilizados en las expresiones sean compatibles. Por ejemplo, en una expresión como $a + b$, si a es un entero y b es un flotante, el compilador puede realizar una conversión implícita de a a flotante para que la operación tenga sentido. Sin embargo, si intentamos sumar un entero con una cadena de texto, el compilador no podrá hacerlo y generará un error de tipo.
- En muchos lenguajes, como C++ y Java, los tipos de datos son fuertemente tipados, lo que significa que las comprobaciones de tipos se realizan en tiempo de compilación, antes de que el programa se ejecute. Otros lenguajes, como Python, permiten más flexibilidad, pero aún así verifican los tipos durante la ejecución.

4

MATERIALES UTILIZADO

- **Computadora** con un entorno de desarrollo integrado (IDE) o un compilador de un lenguaje de programación como C++, Java o Python.
- **Paiza.io:** un compilador en línea que permite ejecutar código en múltiples lenguajes de programación sin necesidad de instalar software en la computadora.
- **Acceso a internet** para consultar documentación en caso de ser necesario.
- **Ejemplos de código** que incluya expresiones con tipos de datos diferentes (esto será proporcionado en la práctica).

5

REQUISITOS BÁSICOS.

Antes de realizar esta práctica, debemos conocer:

- Qué son los **tipos de datos** y para qué se usan.
- Cómo declarar **variables** en un lenguaje de programación.
- Las **operaciones básicas** que se pueden realizar entre diferentes tipos de datos (como suma, resta, concatenación).
- Tener un **compilador o intérprete** instalado para ejecutar los ejemplos.

Paso 1: Abrí Paiza.io

Primero, accedí al compilador en línea [Paiza.io](#), que me permitió trabajar sin necesidad de instalar un compilador local. Seleccioné el lenguaje de programación C++ para esta práctica y comencé un nuevo proyecto.

Paso 2: Escribí el código de ejemplo

En el editor de Paiza.io, escribí el siguiente código en C++:

```
#include <iostream>
using namespace std;

int main() {
    int a = 5;                  // Tipo entero
    float b = 3.2;              // Tipo flotante
    string c = "Hola";          // Tipo cadena

    // Operación válida: suma entre int y float
    float resultado = a + b;
    cout << "Resultado: " << resultado << endl;

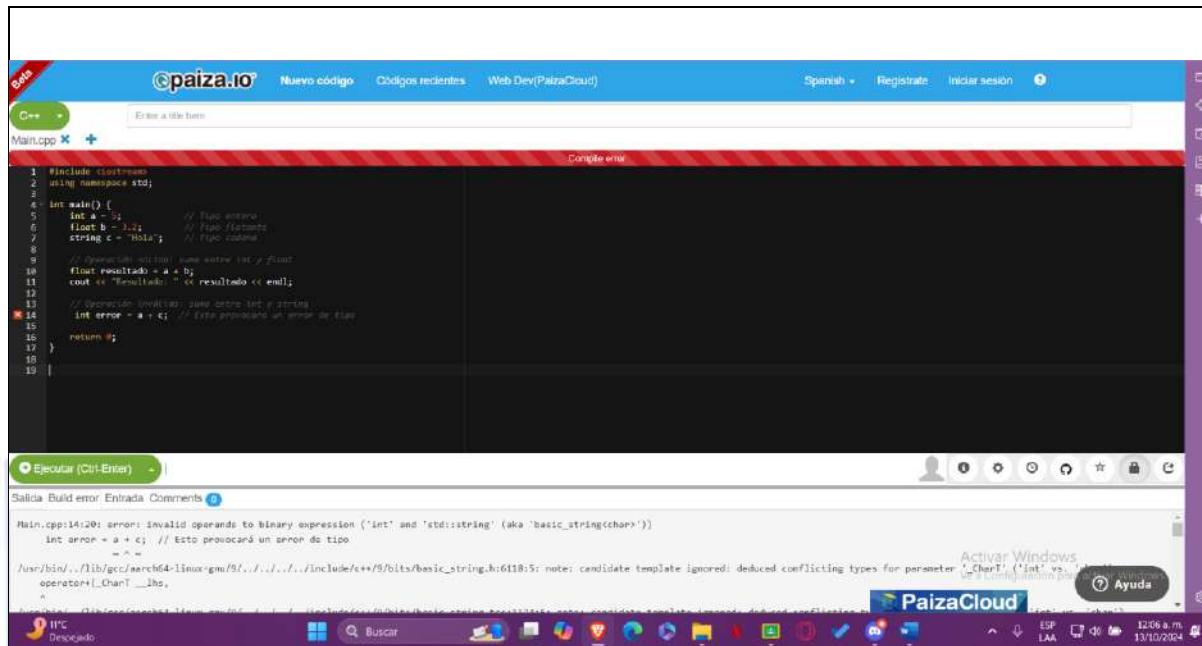
    // Operación inválida: suma entre int y string
    // int error = a + c; // Esto provocará un error de tipo

    return 0;
}
```

Al escribir el código, probé sumar un entero con un flotante, lo cual es válido, y luego intenté sumar un entero con una cadena, lo que sabía que produciría un error de tipos.

Paso 3: Ejecuté el programa en Paiza.io

Hice clic en el botón "**Run**" en Paiza.io para compilar y ejecutar el programa. En la primera ejecución, la consola mostró el resultado de la suma válida entre el entero y el flotante, mientras que al descomentar la línea que intentaba sumar el entero con la cadena, Paiza.io generó un error de compilación, indicando una incompatibilidad de tipos.



The screenshot shows the Paiza.io web-based IDE interface. The code editor contains the following C++ code:

```

1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     int a = 5;           // Tipo entero
6     float b = 3.2;      // Tipo flotante
7     string c = "Hola";  // Tipo cadena
8
9     // Operación válida: suma entre int y float
10    float resultado = a + b;
11    cout << "Resultado: " << resultado << endl;
12
13    // Operación inválida: suma entre int y string
14    int error = a + c; // Esto provocará un error de tipo
15
16    return 0;
17 }
18
  
```

The terminal window below shows a build error:

```

Main.cpp:14:20: error: invalid operands to binary expression ('int' and 'std::string' (aka 'basic_string<char>'))
      int error = a + c; // Esto provocará un error de tipo
                           ^
/usr/bin/../lib/gcc/x86_64-linux-gnu/7.1.0/../../../../include/c++/7.1.0/bits/basic_string.h:6118:5: note: candidate template ignored: deduced conflicting types for parameter '_CharT' ('int' vs. 'basic_string<char>')
operator+(_CharT __lhs,
          ^

  
```

The error message indicates that the compiler cannot perform a binary operation between an integer and a string.

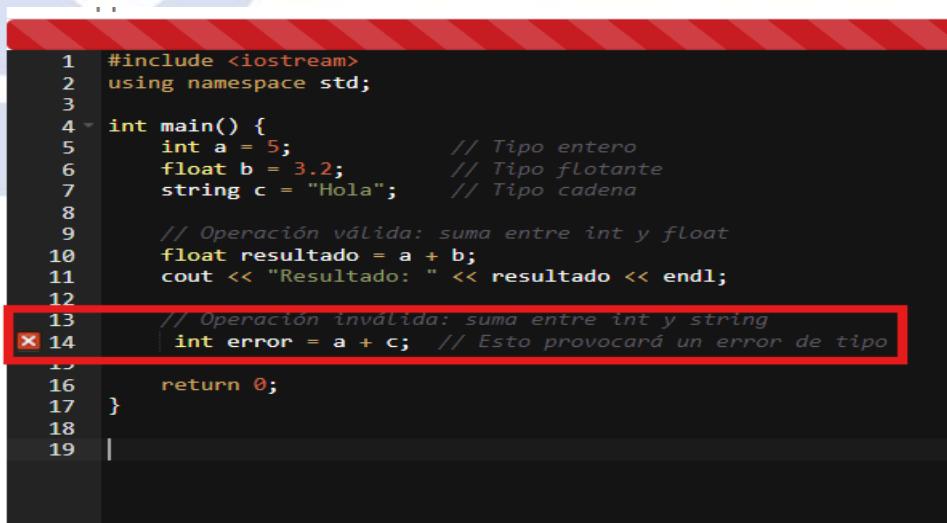
Ilustración 1. Presentación del problema

Paso 4: Analicé los errores de tipo

Paiza.io me mostró un mensaje de error que indicaba que no era posible sumar un número entero (int) con una cadena de texto (string). Esto confirmó que el compilador verifica los tipos de datos y detiene la ejecución si encuentra combinaciones inválidas.

Paso 5: Corregí los errores

Comenté de nuevo la línea problemática que intentaba sumar el entero con la cadena y volví a ejecutar el código. Esta vez, el programa se ejecutó correctamente, mostrando el resultado esperado solo de las operaciones válidas.



The terminal window shows the corrected C++ code:

```

1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     int a = 5;           // Tipo entero
6     float b = 3.2;      // Tipo flotante
7     string c = "Hola";  // Tipo cadena
8
9     // Operación válida: suma entre int y float
10    float resultado = a + b;
11    cout << "Resultado: " << resultado << endl;
12
13    // Operación inválida: suma entre int y string
14    int error = a + c; // Esto provocará un error de tipo
15
16    return 0;
17 }
18
  
```

The output of the program is:

```

Resultado: 8.2
  
```

The line containing the invalid operation (line 14) is highlighted with a red box.

Ilustración 2. Error de tipo

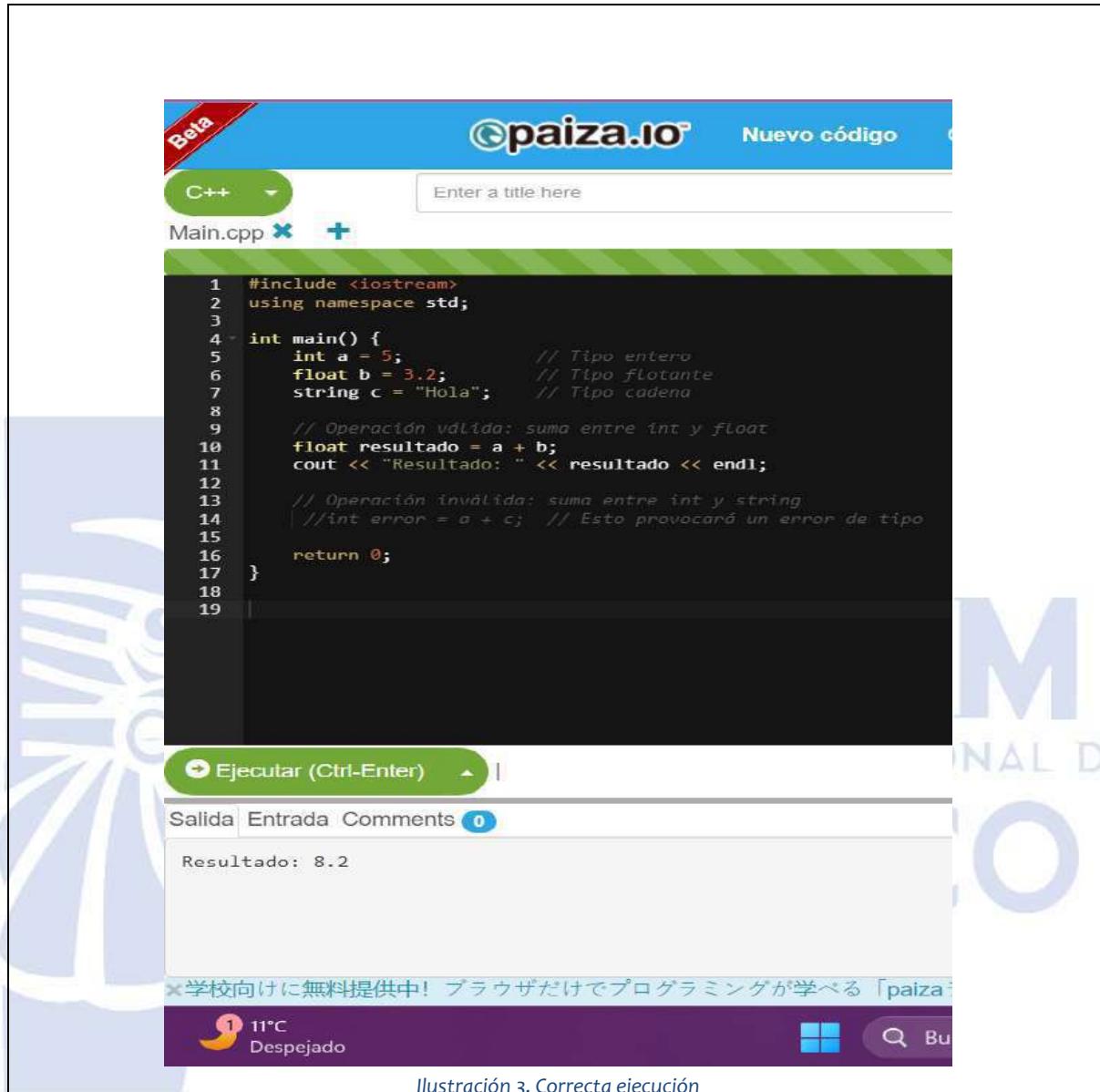


Ilustración 3. Correcta ejecución

Paso 6: Experimenté con otras combinaciones de tipos

Realicé más pruebas con diferentes combinaciones de tipos en Paiza.io. Probé sumar un número flotante con una cadena de texto, lo cual nuevamente generó un error. También experimenté con la suma de enteros y booleanos, y Paiza.io me permitió ejecutar la operación sin problemas, ya que los booleanos pueden ser tratados como números (true como 1 y false como 0).

Paso 7: Anoté las conclusiones

Después de realizar las pruebas, anoté que Paiza.io detecta correctamente los errores de tipo y permite conversiones implícitas entre algunos tipos de datos, como int a float, pero no entre tipos que no tienen sentido juntos, como int y string. Esto me permitió comprender mejor cómo los compiladores evitan errores en tiempo de ejecución mediante la comprobación de tipos.


5
BITÁCORA DE INCIDENCIAS

PROBLEMA	FECHA	HORA	SOLUCIÓN	FECHA	HORA
Error al sumar un entero con una cadena (incompatibilidad de tipos)	12/10/2024	10:15 AM	Comentar la línea de código que genera la operación inválida	12/10/2024	10:20 AM
Paiza.io no ejecutaba correctamente el código debido a un fallo en la conexión	12/10/2024	10:30 AM	Recargué la página y volví a ejecutar el programa	12/10/2024	10:35 AM
Mensaje de error: tipos incompatibles al sumar un flotante y una cadena	12/10/2024	10:45 AM	Comenté la línea problemática y compilé de nuevo	12/10/2024	10:50 AM

6
OBSERVACIONES

Durante el desarrollo de la práctica, observé que el compilador de Paiza.io es eficiente al detectar errores de tipo y evitar que el programa se ejecute con errores graves. Sin embargo, es importante estar atento a los mensajes de error para corregir rápidamente cualquier conflicto de tipos. También noté que algunas operaciones entre tipos de datos, como la conversión automática de int a float, se manejan sin problemas, pero al mezclar tipos no compatibles, como cadenas y números, siempre generará errores.

Además, trabajar en un compilador en línea tiene sus ventajas en cuanto a accesibilidad, pero en ocasiones puede haber problemas de conectividad que retrasen el trabajo. Es importante siempre verificar que las operaciones que realizamos entre variables de diferentes tipos sean válidas antes de intentar compilarlas.

7
ANEXOS
ILUSTRACIÓN 1. PRESENTACIÓN DEL PROBLEMA 4
ILUSTRACIÓN 2. ERROR DE TIPO 4
ILUSTRACIÓN 3. CORRECTA EJECUCIÓN 5
7
REFERENCIAS BIBLIOGRÁFICAS

- Tutorial en línea: "Guía para la Comprobación de Tipos en C++", disponible en: <https://cplusplus.com>.
- Paiza.io: Plataforma de compilación en línea, disponible en: <https://paiza.io>.

Monografía Técnica

Tema 4.4 Pila Semántica en un analisador sintáctico

INTRODUCCIÓN

En el contexto de la construcción de compiladores, los analizadores sintáticos juegan un papel crucial de descomponer y organizar el código fuente de un programa en una estructura jerárquica comprensible para una máquina. Sin embargo, para que un compilador no solo entienda la estructura, sino también el significado, del código, se necesita un mecanismo adicional: la pila semántica. Este componente almacena información relacionada con el contexto semántico de las construcciones del programa mientras se realiza el análisis sintáctico.

La pila semántica permite que el compilador maneje aspectos como el control de tipos, la generación del código intermedio y la detección de errores semánticos.

En este proceso, los valores y símbolos se apilan y desapilan de manera ordenada asegurando que las decisiones semánticas se realicen correctamente a medida que se recorre la gramática del lenguaje. En esta monografía exploraremos el funcionamiento y la importancia de la pila semántica dentro del análisis sintáctico, explicando su rol en el ciclo completo de la compilación.

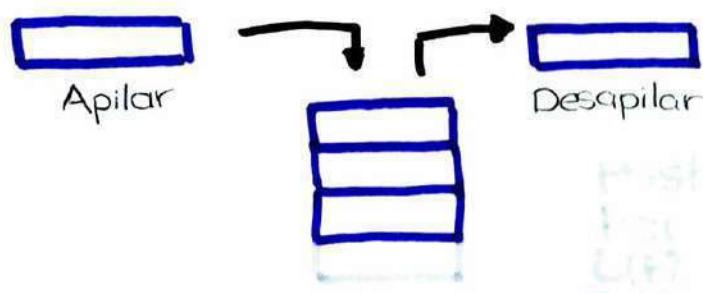


Figura 1. Representación de una pila

EDUCACIÓN



La pila semántica es una estructura de datos fundamental utilizada durante la fase de análisis sintáctico en el proceso de compilación, específicamente en los analizadores sintácticos con acciones semánticas. Esta pila almacena información que se va generando a medida que el código fuente es procesado y ayuda a resolver aspectos relacionados con el significado (semántica) del código, como la evaluación de expresiones, la verificación de tipos y la generación de código intermedio.

¿Qué es la pila semántica?

La pila semántica es una estructura de tipo LIFO (Last In, First Out) que guarda valores y atributos asociados con los símbolos de una gramática durante el análisis sintáctico. Cada vez que se reduce una producción de la gramática, se ejecutan acciones semánticas que manipulan la pila semántica, ya sea añadiendo o quitando valores.

Este proceso está estrechamente ligado al análisis LR o descendente recursivo de las gramáticas, donde las decisiones semánticas se toman al reducir reglas.

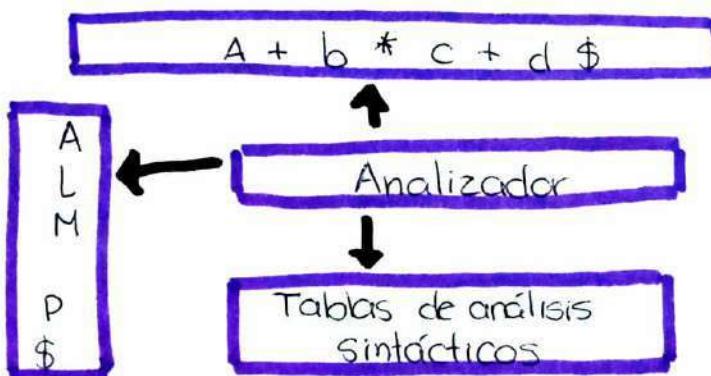


Figura 2. Ejemplo de pila semántica

El análisis semántico usa como entrada el árbol sintáctico para comprobar restricciones de tipo y otras limitaciones semánticas y preparar la generación de código.

¿Para qué se usa la pila?

Para contener la información semántica asociada a los operandos (y operadores) en forma de registros semánticos tomando en cuenta las reglas semánticas (conjunto de normas y especificaciones que definen al lenguaje). → Conversiones implícitas.



Características

Sus características fundamentales es que al extraer se obtiene siempre el último elemento que acabe insertarse. Por esta razón también se conoce como estructuras de datos LIFO, una posible implementación mediante listas enlazadas sera insertando y extrayendo siempre por el principio de la lista.

Además una de las principales funciones de la pila semántica es poder almacenar los atributos semánticos de los símbolos que componen el lenguaje. Estos atributos pueden ser valores numéricos, tipos de datos, operadores y otros elementos necesarios para realizar evaluaciones o verificar el código. A medida que el analizador sintáctico reduce una producción gramatical, se ejecutan acciones semánticas que manipulan los valores en la pila.

Por ejemplo: cuando se detecta una expresión aritmética, los operandos y el operador son extraídos de la pila, se realiza la operación, y luego el resultado es apilado nuevamente.

La pila semántica es compatible tanto con analizadores LR (de tipo ascendente) como con analizadores LL (de tipo descendente), lo que le convierte en una herramienta versátil dentro de los distintos enfoques de análisis sintáctico. Durante este proceso, también juega un papel clave en la evaluación de expresiones y la generación de código intermedio. Los valores apilados pueden ser utilizados para crear un árbol de sintaxis abstracta (AST) o para almacenar instrucciones temporales que luego serán transformadas en código ejecutable.

Otro característica importante de la pila semántica es su capacidad para facilitar el control de tipos. A medida que se apilan y desapilan elementos, el compilador puede verificar que los tipos de los operandos sean compatibles con las operaciones que se realizan, evitando errores de ejecución. Además, esta estructura es fundamental para manejar contextos anidados, como funciones dentro de funciones, ya que permite almacenar y recuperar la información necesaria para cada ámbito semántico.

Por ultimo, la pila semántica ayuda a detectar errores semánticos durante el análisis sintáctico. Si se detecta un error, como el uso incorrecto de tipos o variables no declaradas, la información almacenada en la pila puede proporcionar un contexto útil para generar mensajes de error detallados. Esta pila está profundamente integrada con las fases de análisis léxico y sintáctico, proporcionando una herramienta eficiente.

¿Cómo funciona la pila semántica?

Durante el proceso de análisis sintáctico, cuando el parser (análisis sintáctico) detecta una reducción en una producción, ejecuta una acción semántica asociada a esa producción. Esta acción puede ser la evaluación de una expresión, la construcción de un símbolo de sintaxis abstracto, o la generación de código intermedio, entre otras cosas. La pila semántica se utiliza para:

1: Guardar atributos semánticos: Mientras el parser avanza, la pila semántica almacena atributos de los símbolos gramaticales, como valores de variables, operadores, tipos de datos, entre otros.

2: Resolver expresiones: En cada paso de reducción, los operandos y operadores necesarios para resolver una expresión son extraídos de la pila, se evalúa el resultado y luego se vuelve a apilar.

3. Realizar comprobaciones semánticas: A medida que se reduce una producción, el compilador puede realizar verificaciones semánticas, como el control de tipos o la validación de operadores, utilizando los valores en la pila semántica.

4. Generar código Intermedio: En compiladores que utilizan varias fases, la pila semántica también se puede utilizar para almacenar instrucciones temporales o fragmentos de código que luego se convertirán en código máquina o código optimizado.

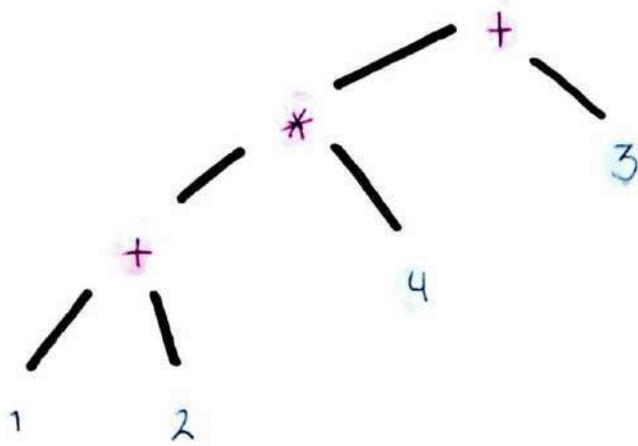


Figura 3. Ejemplo de una pila semántica



Ventajas	Desventajas
1- Estructura simple y eficiente (LIFO): El comportamiento LIFO de la pila semántica permite manipular los valores de manera ordenada y rápida.	1- Complejidad en gramáticas: En lenguajes con gramáticas muy complejas, la gestión de la pila semántica puede volverse difícil y propensa a errores.
2- Evaluación de expresiones en tiempo real: Permite procesar y evaluar expresiones semánticas mientras se realiza el análisis sintáctico, mejorando la eficiencia.	2- Gestión manual: La correcta implementación de las acciones semánticas y el manejo de la pila puede requerir mucho código adicional y ser tediosa.
3- Soporte para generación de código intermedio: Facilita la construcción de código intermedio o árboles de sintaxis abstracta (AST), agilizando el proceso de compilación.	3- Difícil Depuración: Si ocurre un error semántico, puede ser complicado identificar el problema dentro de la pila, lo que dificulta la depuración.
4- Verificación de tipos: Proporciona un mecanismo para verificar tipos de datos y asegurar que las operaciones sean válidas antes de la ejecución.	4- No maneja errores semánticos automáticamente: La pila no puede por sí sola prevenir o corregir errores; necesita un manejo explícito en el código del compilador.
5- Manipulación de contextos anidados: Es útil para manejar diferentes niveles de ámbito o contexto, como en funciones o bloques anidados.	5- Consumo de memoria: En programas muy grandes o muchas operaciones, la pila semántica puede crecer significativamente y consumir más memoria.
6- Integración con análisis léxico y sintáctico: Facilita el procesamiento conjunto de la estructura y el significado.	6- Requiere un diseño cuidadoso: El mal uso de la pila puede llevar a inconsistencias afectando la semántica.

¿Cuál es su objetivo teórico?

Es construir un árbol de análisis sintáctico, este no únicamente se construye como tal, sino que las rutinas semánticas, intercambios van generando el árbol de Sintaxis abstracta. Se especifica mediante una gramática libre de contexto.

El análisis semántico detecta la validez semántica de las sentencias aceptadas por el analizador sintáctico. El analizador semántico se le trabaja simultáneamente al analizador sintáctico y en una estrecha cooperación. Se entiende por semántica como el conjunto de reglas que especifican el significado de cualquier sentencia. Sintácticamente correcta y escrita en determinado lenguaje. Las rutinas semánticas están asociadas a cada producción de la gramática.

El análisis sintáctico es la fase en la que se trata de determinar el tipo de los resultados intermedios, poder comprobar que los argumentos que tiene un operador pertenecen al conjunto de los operadores posibles, y si son compatibles entre sí, etc.

En definitiva comprobara que el significado de lo que se va leyendo es válido. La salida teórica de la fase de análisis semántico sería: un árbol semántico. Consiste en un árbol sintáctico en el que cada una de sus ramas ha adquirido el significado que debe tener.

Se compone de un conjunto de rutinas independientes, llamadas por los analizadores morfológico y sintáctico. El análisis semántico utiliza como entrada el árbol sintáctico detectado por el análisis sintáctico para comprobar restricciones de tipo y otras limitaciones semánticas y preparar la generación de código.

Las rutinas semánticas suelen hacer uso de una pila que contiene la información semántica asociada a los operadores en forma de registros semánticos.

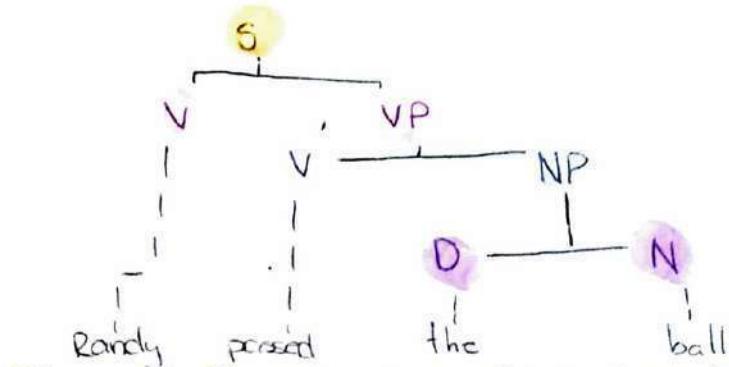


Figura 4. Ejemplo de un árbol de análisis



Ejemplo

Expresión:

$$3 + 5 * 2$$

Paso 1: Análisis Sintáctico

Un analizador sintáctico basado en una gramática de expresiones aritméticas puede reducir la expresión aplicando las reglas de precedencia, comenzando así con la multiplicación antes que la suma.

$$\begin{aligned} \text{Expr} &\rightarrow \text{Expr} + \text{Term} \\ \text{Term} &\rightarrow \text{Term} * \text{Factor} \\ \text{Factor} &\rightarrow \text{número} \end{aligned}$$

Paso 2: Acciones semánticas

A medida que el analizador sintáctico identifica las producciones de la gramática, ejecuta acciones semánticas asociadas con las reducciones. Las acciones semánticas utilizan la pila semántica para evaluar los resultados de las operaciones.

Ejecución de la pila semántica

1: Identificación del número 3:

- El número 3 es un factor. Se apila en la pila semántica
- Pila semántica [3]

2: Identificación del operador +:

El analizador identifica el operador +, pero no realiza ninguna acción aún, solo espera el siguiente operando.

3: Identificación del número 5:

- El número 5 también es un factor y se apila.
- Pila semántica [3, 5]

4: Identificar el operador *:

Se detecta el operador *. Dado que tiene mayor precedencia que +, el analizador procederá a evaluar la multiplicación.

5: Identificación del número 2:

- El número 2 es apilado.
- Pila semántica [3, 5, 2]



6. Evaluación de la multiplicación $5 * 2$

El analizador sintáctico reduce la operación $\text{Term} \rightarrow \text{Term} * \text{Term}$. La acción semántica correspondiente desapila los dos operandos 5 y 2 , realiza la multiplicación, y el resultado (10) es apilado.

- Pila semántica: $[3, 10]$

7. Evaluación de la suma $3 + 10$

Ahora se reduce la producción $\text{Expr} \rightarrow \text{Expr} + \text{Term}$. Se desapilan los operandos 3 y 10 , se realiza la suma, y el resultado (13) se apila como el valor final.

- Pila semántica: $[13]$

Resultado Final

El resultado de la expresión $13 + 5 * 2$ es 113 , y esto es lo que queda en la pila semántica al finalizar el análisis.

Este ejemplo nos muestra cómo un analizador sintáctico utiliza la pila semántica para evaluar expresiones aritméticas de acuerdo con las reglas gramaticales y las acciones semánticas correspondientes. Durante el proceso, los valores son apilados y desapilados para realizar operaciones respetando la precedencia de operaciones.

Errores semánticos y su manejo

Tipos de datos incompatibles:

Si se intenta sumar un entero a una cadena de texto, el compilador puede usar la pila para verificar si los operandos son compatibles antes de realizar la operación.

Variables no inicializadas:

Si se intenta usar una variable sin haberle asignado un valor previamente, la pila semántica puede almacenar la información sobre si una variable ha sido inicializada.



Conclusion

La pila semántica es una herramienta clave dentro de los analizadores sintácticos, ya que nos permite manejar y procesar el significado de las expresiones que aparecen en el código. Ya lo largo de la monografía pudimos ver cómo la pila semántica funciona en conjunto con el análisis sintáctico para evaluar operaciones como sumas y multiplicaciones, garantizando que se respeten las reglas del lenguaje de programación. Además aunque su implementación puede volverse un poco complicada en algunos casos, su utilidad es muy importante para detectar errores y generar código intermedio. En resumen, entender la pila semántica nos ayuda a comprender mejor cómo se traduce el código que escribimos en un lenguaje de alto nivel a algo que la computadora puede entender y ejecutar.

Referencias

- Automatas, L.Y., & Isccamc, U.1@. (S/F) 7.4. PILA SEMANTICA EN UN ANALIZADOR SINTÁCTICO <https://en linea.zacatecas.tecm.mx/pluginfile.php/10833/mod-resource/content/1/1.4%20%20Semantica%20en%20un%20analizador%20sintactico.pdf>
- De expresiones, A. (S/F). Unidad 1: Análisis semántico. Itpn.mx. Recuperado el 14 de Octubre de 2024, de <http://itpn.mx/recursos/isc/7semestre/lenguajesyautomatas2/Unidad%201.pdf>.
- (S/F). Site 123.me. Recuperado el 14 Octubre de 2024, de <https://5e344735705b1.site.123.me/unidad-1-%C3%88an%C3%A1lisis-sem%C3%A1ntica-%C3%A1ltantico/14-pila-sem%C3%A1ntica-en-un-analizador-sint%C3%A1ctico>.



CARRERA	NOMBRE DE LA ASIGNATURA
INGENIERIA EN SISTEMAS COMPUTACIONALES	LENGUAJES Y AUTOMATAS 2

DOCENTE DESIGNADO
ISC. RICARDO GONZÁLEZ GONZÁLEZ

PRACTICA No.	NOMBRE DE LA PRACTICA	DURACIÓN (HORAS)
15	TEMA 4.4 PILA SEMÁNTICA EN UN ANALIZADOR SINTÁCTICO.	1 hora

1

INTRODUCCIÓN

En esta práctica, se estudió el uso de la pila semántica en los analizadores sintácticos. Los analizadores sintácticos son herramientas que ayudan a leer y entender el código de un programa, verificando que las instrucciones sigan las reglas del lenguaje que estamos usando. Una parte importante de este proceso es saber cómo se evalúan las expresiones matemáticas.

La pila semántica es una estructura de datos que funciona como una caja donde podemos guardar cosas temporalmente. En este caso, guarda los números y los operadores matemáticos (como +, -, *, /) mientras evaluamos una expresión. Por ejemplo, cuando tenemos una expresión como $3 + 5 * 2$, la pila nos ayuda a calcular el resultado siguiendo el orden correcto de las operaciones.

Durante esta práctica, implementé un programa en C++ que utiliza pilas semánticas para evaluar expresiones aritméticas. A través de este ejercicio, aprendí cómo funcionan las pilas y por qué son importantes en el análisis sintáctico. También vi cómo se pueden usar para resolver problemas en la programación.

2

OBJETIVO (COMPETENCIAS)

El objetivo de esta práctica es:

- Comprender y aplicar el concepto de pila semántica en un analizador sintáctico.
- Desarrollar una implementación en C++ que permita evaluar expresiones aritméticas utilizando pilas.
- Fortalecer las habilidades de programación en C++ y la utilización de estructuras de datos como pilas.
- Aprender a utilizar plataformas en línea como [Paiza.io](#) para la ejecución y prueba de programas.

3

MARCO TEÓRICO REFERENCIAL

3.1. ¿Qué es una Pila Semántica?

Una pila semántica es como una herramienta que nos ayuda a manejar las cosas mientras estamos evaluando expresiones en un programa. La pila funciona de manera que lo último que se guarda es lo primero que se saca, esto se llama LIFO (Last In, First Out). Esto es útil porque nos permite mantener un orden en las operaciones que realizamos.

3.2. Cómo Funcionan las Pilas

Las pilas son estructuras de datos muy simples. Imagina una pila de platos: puedes poner un plato arriba y, cuando necesites uno, solo puedes sacar el que está en la parte superior. En programación, usamos pilas para guardar números y operaciones mientras evaluamos expresiones matemáticas.

3.3. Precedencia y Asociatividad

Cuando estamos trabajando con operaciones matemáticas, necesitamos saber qué operación hacer primero. La precedencia de los operadores nos dice el orden en el que se deben realizar. Por ejemplo, en la expresión $3 + 5 * 2$, primero debemos multiplicar 5 por 2 y luego sumar 3, porque la multiplicación tiene una mayor precedencia que la suma.

La asociatividad se refiere a cómo se manejan los operadores que tienen la misma prioridad. Por lo general, se evalúan de izquierda a derecha, como en $6 - 2 - 1$, donde primero restamos 2 de 6 y luego restamos 1 del resultado.

3.4. ¿Para Qué Sirven las Pilas Semánticas?

Las pilas semánticas son muy útiles en muchas áreas de la programación. No solo sirven para evaluar expresiones matemáticas, sino también para ayudar a los compiladores a entender el código. También se utilizan en juegos, aplicaciones y cualquier lugar donde necesitemos organizar información de manera temporal.

3.5. Importancia en la Programación

Entender cómo funcionan las pilas es esencial para cualquier persona que quiera programar. Nos ayudan a resolver problemas de manera más eficiente y son la base para construir programas más complejos. Además, aprender sobre pilas nos enseña a pensar de manera lógica, algo que es fundamental en la programación.

4

MATERIALES UTILIZADO

- **Lenguaje de programación:** C++
- **Entorno de desarrollo:** Paiza.io (plataforma en línea)
- **Documentación:** Recursos en línea sobre pilas y análisis sintáctico.
- **Referencias de código:** Ejemplos de código de pilas semánticas.



5

REQUISITOS BÁSICOS.

- Conocimientos básicos de programación en C++.
- Comprensión de estructuras de datos, especialmente pilas.
- Familiaridad con conceptos de análisis sintáctico y semántico.
- Acceso a la plataforma Paiza.io para la ejecución del código.

6

DESARROLLO DE LA PRÁCTICA (PASO A PASO)

1. Configuración del Entorno:
Accedí a Paiza.io y seleccioné el entorno de C++ para comenzar a escribir el código.
2. Implementación de la Pila Semántica:
Desarrollé un programa en C++ para evaluar expresiones aritméticas. Comencé definiendo funciones para aplicar operaciones y determinar la precedencia de los operadores.

```
#include <iostream>
#include <stack>
#include <string>
#include <cctype>

using namespace std;

// Función para aplicar una operación a dos operandos
int aplicarOperacion(int operando1, int operando2, char operador) {
    switch (operador) {
        case '+': return operando1 + operando2;
        case '*': return operando1 * operando2;
        default: return 0;
    }
}

// Función para determinar la precedencia de los operadores
int precedencia(char operador) {
    if (operador == '+') return 1;
    if (operador == '*') return 2;
    return 0;
}

// Función principal para evaluar una expresión usando pilas
semánticas
int evaluarExpresion(const string& expresion) {
    stack<int> pilaValores;           // Pila semántica (almacena los
valores)
    stack<char> pilaOperadores;     // Pila de operadores (almacena los
operadores)

    for (int i = 0; i < expresion.length(); ++i) {
        // Ignorar los espacios en blanco
        if (expresion[i] == ' ') continue;

        // Si el carácter es un número, lo empujamos a la pila de
valores
```

```
if (isdigit(expresion[i])) {
    int valor = 0;
    // Manejo de números con más de un dígito
    while (i < expresion.length() && isdigit(expresion[i])) {
        valor = valor * 10 + (expresion[i] - '0');
        i++;
    }
    pilaValores.push(valor);
    i--; // Retrocedemos uno porque el índice ha avanzado
más de la cuenta
}

// Si el carácter es un operador
else if (expresion[i] == '+' || expresion[i] == '*') {
    // Aplicamos las operaciones de la pila de operadores con
mayor o igual precedencia
    while (!pilaOperadores.empty() &&
           precedencia(pilaOperadores.top()) >=
precedencia(expresion[i])) {
        char operador = pilaOperadores.top();
        pilaOperadores.pop();

        int operando2 = pilaValores.top(); pilaValores.pop();
        int operando1 = pilaValores.top(); pilaValores.pop();

        int resultado = aplicarOperacion(operando1,
operando2, operador);
        pilaValores.push(resultado);
    }
    // Empujamos el operador actual a la pila de operadores
    pilaOperadores.push(expresion[i]);
}
}

// Aplicamos los operadores restantes
while (!pilaOperadores.empty()) {
    char operador = pilaOperadores.top();
    pilaOperadores.pop();

    int operando2 = pilaValores.top(); pilaValores.pop();
    int operando1 = pilaValores.top(); pilaValores.pop();

    int resultado = aplicarOperacion(operando1, operando2,
operador);
    pilaValores.push(resultado);
}

// El valor final en la pila de valores es el resultado de la
expresión
return pilaValores.top();
}

int main() {
```

```

string expresion = "3 + 5 * 2";
int resultado = evaluarExpresion(expresion);
cout << "El resultado de la expresión '" << expresion << "' es: "
<< resultado << endl;
return 0;
}
  
```

3. Estructura del Código:

Implementé las pilas necesarias para almacenar valores y operadores. Usé stack<int> para la pila de valores y stack<char> para la pila de operadores.

```

// Función principal para evaluar una expresión usando pilas semánticas
int evaluarExpresion(const string& expresion) {
    stack<int> pilaValores;      // Pila semántica (almacena los valores)
    stack<char> pilaOperadores; // Pila de operadores (almacena los operadores)

    for (int i = 0; i < expresion.length(); ++i) {
        // Ignorar los espacios en blanco
        if (expresion[i] == ' ') continue;

        // Si el carácter es un número, lo empujamos a la pila de valores
        if (isdigit(expresion[i])) {
            ...
        }
    }
}
  
```

Ilustración 1 implementación de pilas

4. Evaluación de la Expresión:

Programé la lógica para recorrer la expresión, identificar números y operadores, y aplicar las operaciones según la precedencia. Usé un bucle for para iterar sobre cada carácter de la expresión.

```

for (int i = 0; i < expresion.length(); ++i) {
    // Ignorar los espacios en blanco
    if (expresion[i] == ' ') continue;

    // Si el carácter es un número, lo empujamos a la pila de valores
    if (isdigit(expresion[i])) {
        int valor = 0;
        // Manejo de números con más de un dígito
        while (i < expresion.length() && isdigit(expresion[i])) {
            valor = valor * 10 + (expresion[i] - '0');
            i++;
        }
        pilaValores.push(valor);
        i--; // Retrocedemos uno porque el índice ha avanzado más de la cuenta
    }
}
  
```

Ilustración 2. Evaluación de expresión

5. Ejemplo de Evaluación:

Proporcioné un ejemplo de evaluación de la expresión "3 + 5 * 2" en el código, asegurándome de que el resultado final se mostrara correctamente en la consola.

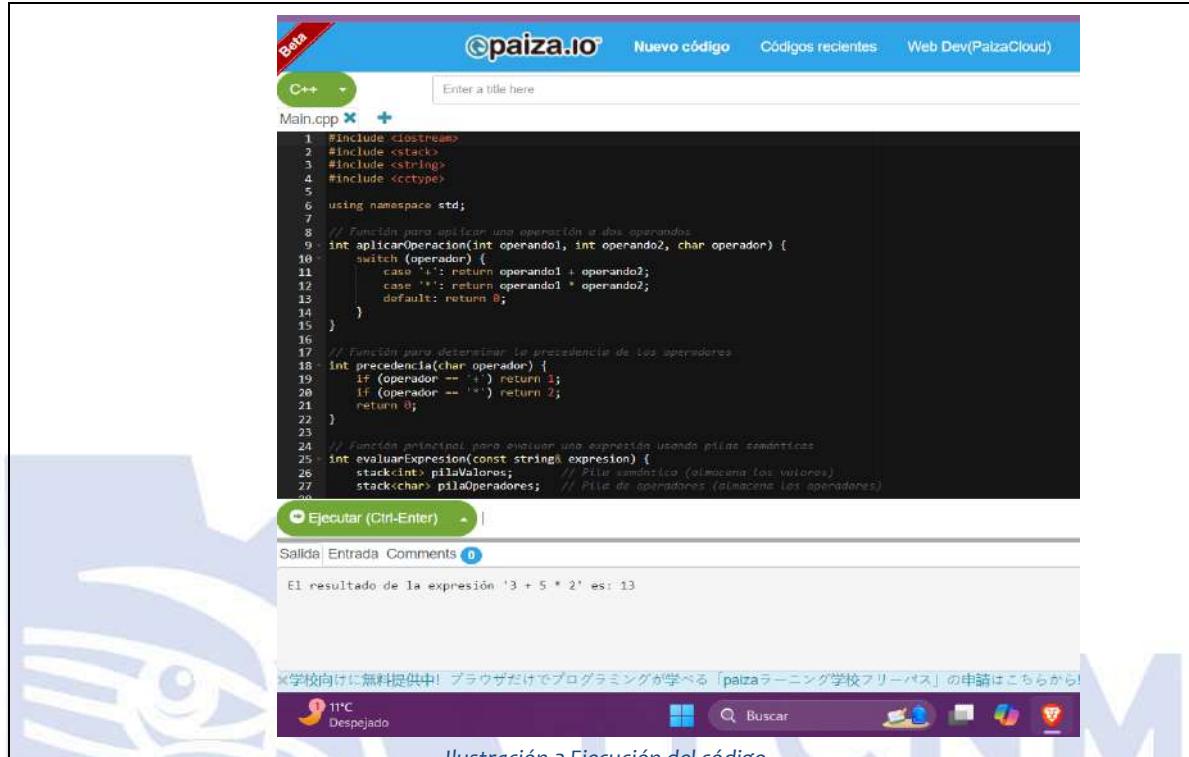


Ilustración 3 Ejecución del código

6. Pruebas y Resultados:

Ejecuté el programa en Paiza.io y verifqué que el resultado de la expresión fuera el esperado: 13. Realicé pruebas adicionales con diferentes expresiones para asegurarme de que la implementación funcionara correctamente.

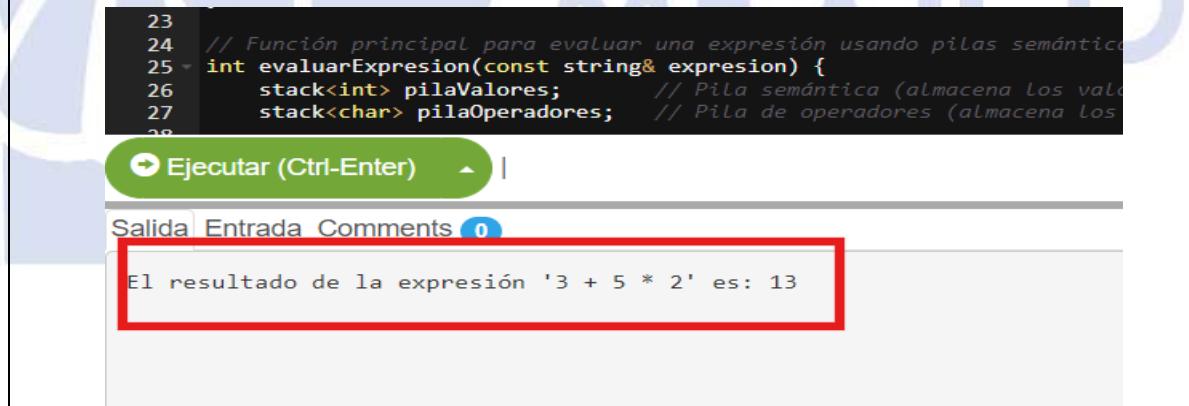


Ilustración 4 Ejecución correcta del programa

5

BITÁCORA DE INCIDENCIAS

PROBLEMA	FECHA	HORA	SOLUCIÓN	FECHA	HORA
Error de compilación debido a una variable no definida.	13/10/2024	1:00 AM	Revisé el código y corregí el nombre de la variable.	13/10/2024	1:10 AM
La pila semántica no estaba evaluando correctamente la expresión.	13/10/2024	1:15 AM	Implementé un chequeo de precedencia en el código	13/10/2024	1:20 AM
El programa no mostraba el resultado de la evaluación.	13/10/2024	1:20 AM	Añadí una función para imprimir el resultado en pantalla.	13/10/2024	1:30 AM
Problemas al ejecutar el programa en Paiza.io.	13/10/2024	1:30 AM	Cambié la configuración del entorno de ejecución	13/10/2024	1:35 AM

6

OBSERVACIONES

Durante la realización de esta práctica, noté que la comprensión de las pilas semánticas es fundamental para el análisis sintáctico. Al principio, me costó entender cómo se manejaban las operaciones en la pila, pero después de resolver algunos problemas, logré ver la lógica detrás de su funcionamiento. También me di cuenta de que realizar pruebas con diferentes expresiones es clave para asegurar que el programa funcione correctamente.

7

ANEXOS

ILUSTRACIÓN 1 IMPLEMENTACIÓN DE PILAS.....	5
ILUSTRACIÓN 2. EVALUACIÓN DE EXPRESIÓN	5
ILUSTRACIÓN 3 EJECUCIÓN DEL CÓDIGO	6
ILUSTRACIÓN 4 EJECUCIÓN CORRECTA DEL PROGRAMA	6

7

REFERENCIAS BIBLIOGRÁFICAS

- 4.4 Tipos de Datos y Verificación de Tipos. (s/f). Edu.mx. Recuperado el 14 de octubre de 2024, de http://cidecame.uaeh.edu.mx/lcc/mapa/PROYECTO/libro32/autocontenido/autocon/44_tipos_de_datos_y_verificacion_de_tipos.html
- Comprobaciones de tipos en expresiones. (s/f). Genially. Recuperado el 14 de octubre de 2024, de <https://view.genially.com/6545d1952e302900127db819/presentation-comprobaciones-de-tipos-en-expresiones>
- (S/f). Site123.me. Recuperado el 14 de octubre de 2024, de <https://5e344735705b1.site123.me/unidad-i-%C3%81nalisis-sem%C3%A1ntico/13-comprobaciones-de-tipos-en-expresiones>

GRUPO 10
FACULTAD DE INGENIERIA
ESTUDIANTE: [Signature]



4.5 ESQUEMA DE TRADUCCIÓN

ANÁLISIS DEL ESQUEMA

ANÁLISIS

COMBINACIÓN DE LAS FUNCIONES

DE LA TRADUCCIÓN

I NTRODUCCIÓN

El esquema de traducción es una metodología esencial en la construcción de compiladores que permite transformar el código fuente de un lenguaje de programación en una representación intermedia o directamente en código objeto. Este esquema combina la definición sintáctica del lenguaje mediante gramáticos libres de contexto (GLC) con acciones semánticas que manipulan datos durante el proceso de análisis sintático. La integración de la sintaxis y la semántica facilita la generación de código eficiente y la verificación de la corrección del programa.

COMPONENTES FUNDAMENTALES DEL ESQUEMA DE TRADUCCIÓN

1: GRAMÁTICA LIBRE DE CONTEXTO (GLC):

- Define la estructura sintáctica del lenguaje de programación.
- Consiste en un conjunto de producciones que describen cómo se pueden combinar símbolos terminales y no terminales para formar sentencias válidas.

→ Ejemplo de una producción:

Expresión → Expresión + Término

2: ACCIONES SEMÁNTICAS:

- Son fragmentos de código que se ejecutan en momentos específicos durante el análisis sintáctico.
- Asociadas a las producciones de la gramática, permiten realizar tareas como generación de código, cálculos de valores y manipulación de la tabla de símbolos.

- Pueden estar escritas en lenguajes como C, C++, Java o pseudocódigo, dependiendo del compilador.

3. ÁRBOL DE DERIVACIÓN ANOTADO:

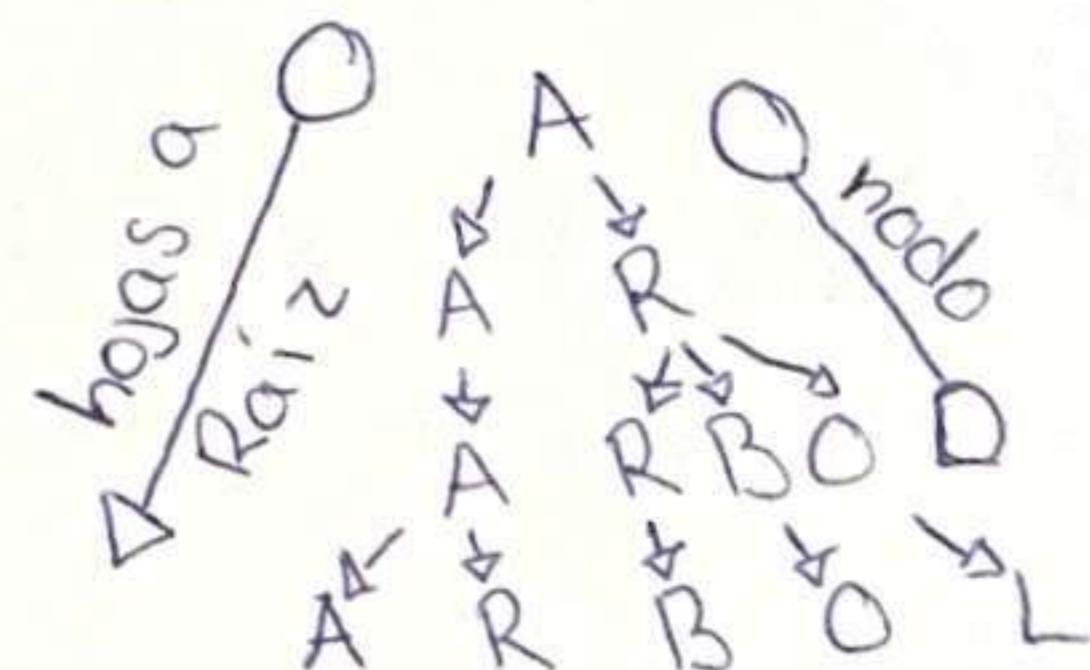
- Representa la estructura jerárquica del programa, incluyendo información semántica adicional.
- Facilita la generación de código al mantener datos relevantes en cada nodo del árbol.
- Permite realizar transversals para optimizar o transformar el código generado.

TIPOS DE ESQUEMAS DE TRADUCCIÓN

1. Esquemas Dirigidos Por Sintaxis (Syntax-Directed Translation):

Esquemas Dirigidos por Atributos:

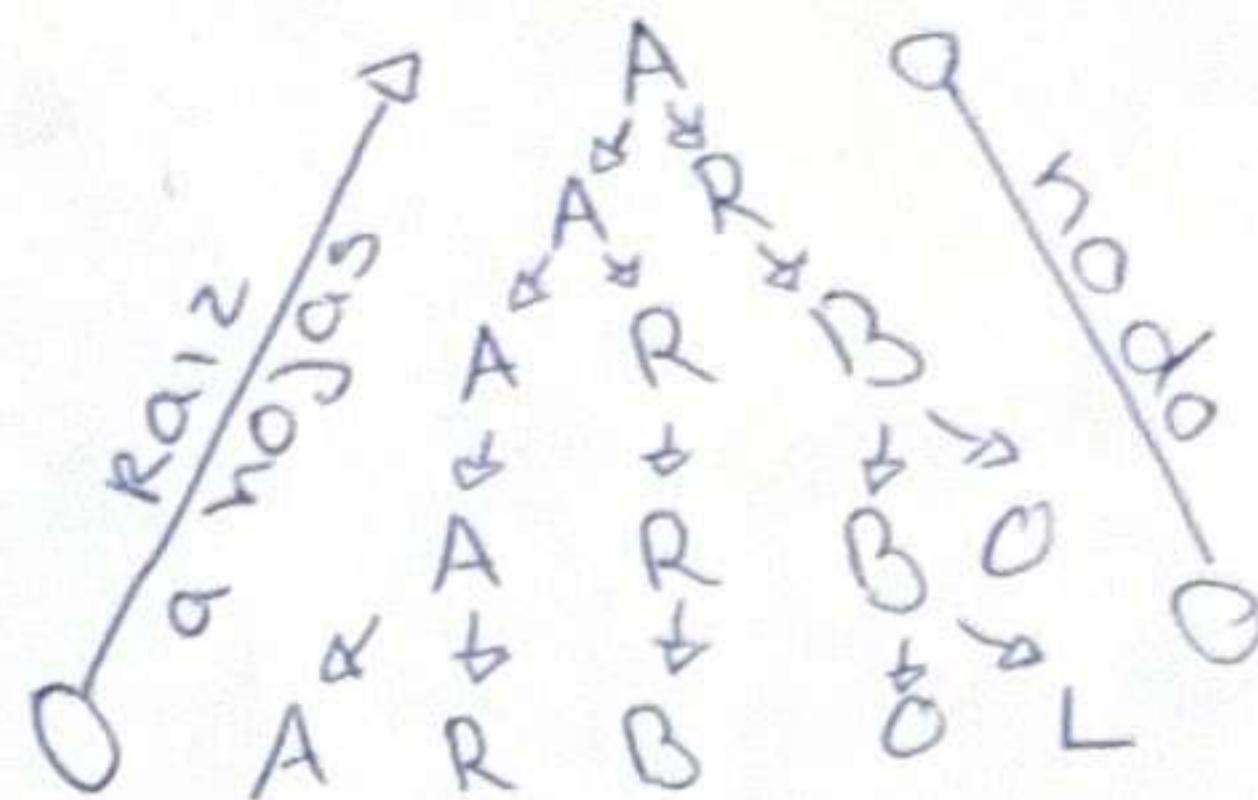
- Atributos Sintéticos: Los valores se calculan desde las hojas hacia la raíz del árbol de derivación.



(F)
Atributos Heredados: Los valores se propagan desde la raíz hacia las hojas

Atributos Compuestos:

Combinan atributos Sintéticos y heredados para manejar dependencias más complejas.



Evaluación De Atributos: ✓✓

Se realiza mediante reglas de evolución y evaluación que definen cómo se calculan los atributos basándose en los atributos de los símbolos circundantes.

2. TRADUCCIÓN DIRIGIDA POR EL ANÁLISIS

SINTÁCTICO

Analizadores LL:

- Utilizan una estrategia de análisis de arriba hacia abajo ↴
- Las acciones semánticas se ejecutan mientras se expande una regla de la gramática.

Analizadores LR:

- Utilizan una estrategia de análisis de abajo hacia arriba
- Las acciones Semánticas Se ejecutan durante las reducciones de la pila de análisis.

FASES DEL ESQUEMA DE TRADUCCIÓN

1: Análisis Léxico:

- Transforma el código fuente en una Secuencia de tokens.
- Cada token representa una unidad léxica, como palabras clave, identificadores, operadores, etc.
- La información relevante de los tokens se pasa a las acciones Semánticas.

2: Análisis Sintáctico:

- Verifica la conformidad de la secuencia de tokens con la gramática del lenguaje
- Construye el árbol de derivación que representa la estructura sintáctica del programa.

3.- Análisis Semántico:

Comprueba la corrección semántica del programa, como la consistencia de tipos y la declaración de variables.

Genera representaciones intermedias o código objeto a través de las acciones semánticas.

4.- Generación De Código:

- Transforma la representación intermedia en código de bajo nivel o código objeto ejecutable.
- Optimiza el código para mejorar la eficiencia y el rendimiento.

TECNICAS AVANZADAS EN ESQUEMAS DE TRADUCCIÓN

1. Traducción Basada En Esquemas de Árboles De Sintaxis Abstracta (AST):

- Los AST simplifican el árbol de derivación eliminando detalles sintácticos irrelevantes.
- Facilitan la manipulación y optimización del código durante la generación.

2. Atributos Dinámicos Y Estáticos:

Atributos Estáticos: Determinados en tiempo de compilación.

Atributos Dinámicos: Determinados en tiempo de ejecución, aunque generalmente se simulan en el esquema de traducción.

3. Optimización De Código En El Esquema De Traducción:

- Eliminación De Código Muerto: Remueve código que nunca se ejecuta.
- Reordenamiento De Instrucciones: Mejoran la eficiencia de la ejecución.
- Propagación De Constantes: Sustituye variables con valores constantes conocidos en tiempo de compilación.

4. Análisis De Flujo De Datos:

- Permite optimizar el código mediante el análisis de cómo los datos fluyen a través de las variables del programa.
- Facilita la detección de dependencias y la eliminación de redundancias.

5. Manejo De Excepciones Y Control De Errores:

- Integración de mecanismos para manejar excepciones y errores semánticos durante la traducción.
- Mejora la robustez y la fiabilidad del compilador.

IMPLEMENTACIÓN DE ESQUEMAS DE TRADUCCIÓN

1. Definición De La Gramática

- Selección de una GLC apropiada que defina la sintaxis del lenguaje de origen.
- Asegurar que la gramática sea libre de ambigüedades o resolver las ambigüedades mediante la prioridad de operadores.

2. Asociación De Acciones Semánticas:

- Integración de acciones semánticas específicas a cada producción de la gramática.
- Estas acciones pueden incluir la generación de código intermedio, la inserción en la tabla de símbolos y la verificación de tipos.

3. Construcción Del Árbol De Derivación:

- Durante el análisis sintáctico, se construye un árbol que representa la estructura del programa.
- Cada nodo del árbol está anotado con información semántica relevante gracias a las acciones semánticas.

4. Generación De Código Intermedio:

- Utilización de representaciones intermedias como instrucciones de tres direcciones, representaciones de árbol de sintaxis abstracta (AST) o código de bytecode.
- Este código intermedio es más cercano al lenguaje de máquina y facilita la optimización y la generación de código final.

EJEMPLO DETALLADO DE ESQUEMA DE TRADUCCIÓN

- 1= $\text{Expresión} \rightarrow \text{Expresión} + \text{Término} \quad \{\text{Expresión}.code = \text{newTemp}();$
 $\quad \quad \quad \text{print}(\text{Expresión}.code + "=" + \text{Expresión1}.code + "+" +$
 $\quad \quad \quad \text{Término}.code)\}$
- 2= $\text{Expresión} \rightarrow \text{Término} \quad \{\text{Expresión}.code = \text{Término}.code\}$
3. $\text{Término} \rightarrow \text{Término} * \text{Factor} \quad \{\text{Término}.code =$
 $\quad \quad \quad \text{newTemp}();$
 $\quad \quad \quad \text{print}(\text{Término}.code + "=" + \text{Término1}.code + "*"+$
 $\quad \quad \quad \text{Factor}.code)\}$
4. $\text{Término} \rightarrow \text{Factor} \quad \{\text{Término}.code = \text{Factor}.code\}$
5. $\text{Factor} \rightarrow (\text{Expresión}) \quad \{\text{Factor}.code = \text{Expresión}.code\}$
6. $\text{Factor} \rightarrow \text{id} \quad \{\text{Factor}.code = \text{id}\}$
7. $\text{Factor} \rightarrow \text{numero} \quad \{\text{Factor}.code = \text{numero}\}$

Acciones Semánticas:

- newTemp(): Función que genera un nuevo temporal.
- print(): Función que emite la instrucción de tres direcciones.

Proceso De Traducción:

Para la expresión $a + b * c$, el esquema de traducción generaría el siguiente código intermedio:

$$t1 = b * c$$

$$t2 = a + t1$$

Ventajas Del Esquema De Traducción

Modularidad: Separación clara entre la sintaxis y las acciones semánticas.

Flexibilidad: Permite adaptar el esquema para diferentes objetivos, como generación de código o verificación de tipos.

Reusabilidad: Las acciones semánticas pueden reutilizarse para diferentes gramáticas con estructuras similares.

Desafíos Y Consideraciones

Ambigüedad en la gramática: Puede complicar la definición de acciones semánticas claras.

Gestión de errores: Implementar un manejo robusto de errores semánticos puede ser complejo.

Optimización del rendimiento: Equilibrar la precisión de las acciones semánticas con la eficiencia del proceso de traducción.

Conclusión

El esquema de traducción es una piedra angular en la construcción de compiladores, permitiendo la transformación sistemática y eficiente del código fuente en una representación intermedia o código objeto ejecutable. Al combinar una gramática libre de contexto con acciones semánticas bien definidas, se facilita la generación de código, la verificación de la corrección semántica y la optimización profunda de los componentes, tipos, fases y técnicas avanzadas del programa. La comprensión robusta y para desarrollar compiladores eficientes que puedan manejar la complejidad de los lenguajes de programación modernos.

4.6 GENERACIÓN DE LA TABLA DE SIMBOLOS Y DE DIRECCIONES

INTRODUCCIÓN

La tabla de símbolos es una estructura de datos central en los compiladores que almacena información sobre los identificadores utilizados en el programa, como variables, funciones, tipos de datos, entre otros. La tabla de direcciones, por otro lado, gestiona la asignación y cálculo de direcciones de memoria para estos identificadores durante la ejecución del programa. Juntas, estas tablas permiten al compilador gestionar eficientemente el acceso y la manipulación de los recursos del programa.

Generación Y Estructura De La Tabla De Símbolos.

Fases de Construcción de la Tabla de Símbolos:

1 Análisis Léxico:

- Identificación de tokens que representan identificadores.
- Inserción de nuevos símbolos en la tabla con atributos básicos.

2. Análisis Sintáctico:

- Reconocimiento de estructuras como declaraciones de variables, funciones, clases, etc.
- Actualización de atributos en la tabla de Símbolos Según el contexto Sintáctico

3. Análisis Semántico:

- Verificación de la coherencia de los tipos y el alcance.
- Resolución de referencias y aseguramiento de que los símbolos sean utilizados correctamente.

Estructura de la Tabla de Símbolos:

Atributo	Descripción Identificador del Símbolo (e.g., variable, función).
Nombre	Tipo asociado al símbolo (e.g., int, float, void)
Tipo de dato	Contexto en el que el símbolo es válido (global, local a función)
Alcance (Scope)	Ubicación en memoria asignada al símbolo durante la ejecución.
Dirección de Memoria	Especificadores adicionales como const, static, extern.
Modificadores	Listas y tipos de parámetros que una función recibe.
Parámetros (para funciones)	Valor asignado al símbolo en el momento de su declaración (si aplica).
Valor inicial	Espacio de memoria requerido para el símbolo.
Tamaño	Información sobre donde se almacena la información del símbolo (e.g., pila, segmento de datos).
Tipo de almacenamiento	

Implementación De La Tabla De Símbolos

Estructuras de Datos Utilizadas:

Tablas Hash: Proporcionan búsquedas rápidas mediante funciones de hash para distribuir los símbolos uniformemente.

Arboles de Símbolos (Symbol Trees): Utilizan estructuras de árbol para organizar los símbolos jerárquicamente según su alcance.

Listas Encadenadas: Simplifican la inserción de nuevos símbolos pero pueden ser menos eficientes para búsquedas.

Manejo De Alcance:

Tablas Jerárquicas o Anidadas: Cada bloque de código (funciones, bucles, etc) tiene su propia tabla de símbolos que referencia a la tabla de símbolos padre.

Desplazamientos (Offsets): Utilizados para calcular direcciones relativas dentro de diferentes ámbitos.

MANEJO DE COLISIONES EN LA TABLA DE SÍMBOLOS

Las colisiones ocurren cuando diferentes símbolos comparten el mismo nombre pero residen en distintos ámbitos. Para resolverlas se utilizan las siguientes técnicas:

1. Tablas Jerárquicas o Anidadas:

- cada ámbito (por ejemplo, una función o un bloque) tiene su propia tabla de símbolos.
- Cuando se busca un símbolo, se comienza en la tabla de símbolos más interna y se avanza hacia las tablas externas si no se encuentra.

2. Árboles De Símbolos:

- Organizan los símbolos en una estructura de árbol donde cada nivel representa un ámbito diferente.
- Facilitan la búsqueda y la inserción de nuevos símbolos manteniéndolo íntegro de los distintos ámbitos.

3. Tablas Hash con Gestión de Alcance:

Utilizan funciones de hash para distribuir los símbolos en buckets.

Incorporación de la información sobre el alcance en la clave de hash para diferenciar símbolos con el mismo nombre pero en diferentes ámbitos.

GENERACIÓN Y CALCULO DE DIRECCIONES DE MEMORIA

Asignación De Direcciones

1. Variables Globales:

- Asignadas en el segmento de datos estáticos
- Su dirección se determina durante la fase de compilación y permanece constante durante la ejecución.

2 Variables Locales:

- Asignadas en la pila (stack).
- Sus direcciones son relativas al puntero de pila (SP) y se calculan en tiempo de compilación.
- Se liberan automáticamente al finalizar el bloque de código en el que se declararon.

3. Variables Dinámicas:

- Asignadas en el heap.
- Requieren manejo en tiempo de ejecución para la asignación y liberación de memoria.

Cálculo De Direcciones:

Desplazamientos (offsets):

- Determinan la posición relativa de una variable dentro de su ámbito.
- Se calculan sumando el tamaño de los tipos de datos previos en la pila o el segmento de datos.

Direcciones Absolutas Vs. Relativas:

Absolutas: Direcciones fijas en memoria, típicamente usados para variables globales.

Relativas: Direcciones calculadas en relación con un riesgo o registro base, como el puntero de pila para variables locales

Asignación Estática Y Dinámica

Estática: Direcciones asignadas en tiempo de compilación, usadas para variables globales y estáticas.

Dinámica: Direcciones asignadas en tiempo de ejecución, usadas para variables locales y dinámicas.

Espacio Para Estructuras Complejas:

Arreglos:

- Dirección base asignada durante la compilación
- Acceso a elementos mediante cálculos de desplazamiento basados en índices.

Registros (Structs en C):

- Dirección base para el registro completo.
- Desplazamientos individuales para campo dentro del registro.

IMPORTANCIA DE LA TABLA DE SÍMBOLOS Y EL CÁLCULO DE DIRECCIONES

1. Verificación De Tipos Y Compatibilidad

- Asegura que las operaciones realizadas sobre los símbolos son semanticamente correctas.
- Previene errores como asignaciones de tipos incompatibles o llamadas a funciones con parámetros incorrectos.

Validar el flujo de datos

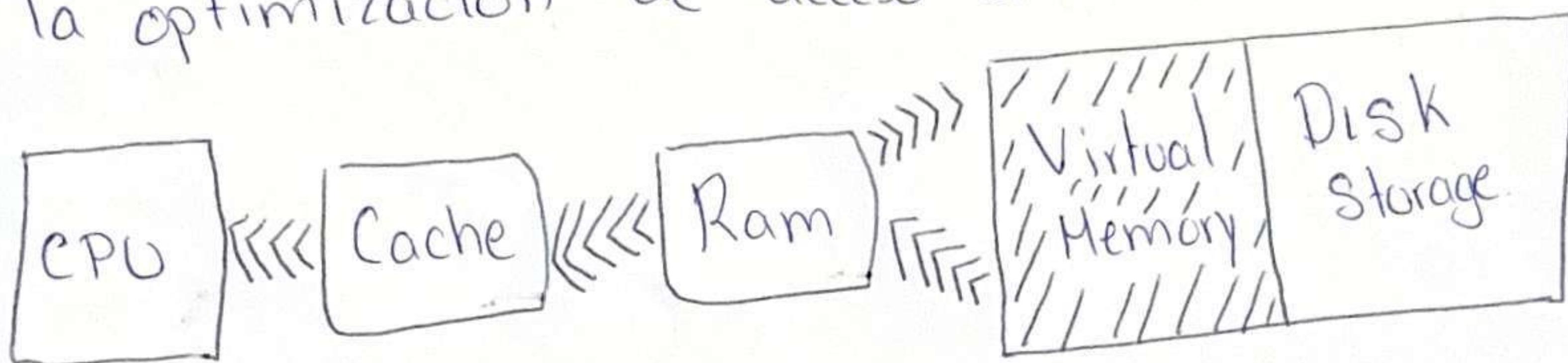
Identificar Dependencias externas

Escenarios de prueba Personalizados

Validación de escalabilidad.

2 Optimización Del Uso De Memoria:

- Permite una asignación eficiente de direcciones para minimizar el desperdicio de espacio.
- Facilita técnicas como el reuso de registro y la optimización de acceso a variables.



3. Generación De Código Eficiente:

- Conocimiento preciso de las direcciones permite generar instrucciones de acceso a memoria más rápidas.
- Falta la generación de código optimizado mediante la eliminación de accesos redundantes.

4. Detección De Errores Semánticos

- Identificaciones de problemas como variables no inicializadas, símbolos no declarados, y conflictos de nombres.
- Mejora la robustez y la confiabilidad del compilador.

5. Facilitación De La Depuración:

- Proporciona información detallada sobre los símbolos y sus ubicaciones, ayudando en la identificación y corrección de errores durante el desarrollo.

IMPLEMENTACION DE LA TABLA DE SIMBOLOS EN UN COMPILADOR.

Ejemplo de Programa en C:

```
int x = 10;  
float y;  
  
void sumar(){  
    int z = 5;  
    y = x + z;
```

Construcción de la Tabla de Símbolos:

Nombre	Tipo	Alcance	Dirección	Otros Atributos
x	int	Global	0x1000	Valor inicial: 10
y	float	Global	0x1004	Valor inicial: 0.0
sumar	void	Global	0x2000	Parametros: none
z	int	local a Sumar()	SP+4	Valor inicial: 5

Cálculo de direcciones:

1. Variables Globales:

x se asigna a la dirección 0x1000.
y se asigna a la dirección 0x1004.

2. Funciones:

Sumar se asigna a la dirección 0x2000

3. Variables Locales:

z en la función sumar se asigna al desplazamiento SP+4, donde SP es el puntero de pila al inicio de la función.

Proceso Detallado De Generación:

1. Declaración de int x = 10 ; :

- Inserta x en la tabla de símbolos con tipo int, alcance global, dirección 0x1000, y valor inicial 10.

2. Declaración de float y ; :

- Inserta y en la tabla de símbolos con tipo float, alcance global, dirección 0x1004, y valor inicial 0.0 (por defecto).

3. Declaración de la función Sumar :

- Inserta sumar en la tabla de símbolos con tipo void, alcance global, y dirección 0x2000
- Crea una nueva tabla de símbolos para el ámbito local de sumar.

4. Declaración de int z=5 ; dentro de sumar :

- inserta z en la tabla de símbolos local con tipo int, alcance local a sumar, desplazamiento SP+4, y valor inicial 5.

5. Asignación y=x+z ; :

- Verifica que x y z estan declaradas y son de tipos compatibles

- Genera código intermedio que suma x y z y almacena el resultado en y.

Conclusión

La tabla de símbolos y el cálculo de direcciones son componentes críticos en el diseño y funcionamiento de un compilador. Su correcta implementación garantiza que el programa fuente se traduzca de manera eficiente y precisa en código ejecutable, permitiendo la verificación de tipos, la optimización de memoria, y la generación de código robusto. La integración de estas estructuras con el esquema de traducción proporcionada y proporciona una base sólida para el desarrollo de compiladores eficientes, y fiables, capaces de manejar la complejidad de los lenguajes de programación modernos.

Manejo de errores semánticos

Introducción

En el ámbito de la programación, los errores semánticos representan un desafío constante para desarrolladores. A diferencia de los errores sintácticos, que violan las reglas gramaticales del lenguaje, los errores semánticos surgen cuando el código es sintácticamente correcto pero su significado no coincide con la intención del programador.

Estos errores pueden manifestarse en distintas formas, desde operaciones inválidas hasta cálculos erróneos, y pueden ser difíciles de detectar debido a su naturaleza sutil.

Definición

Los errores semánticos son aquellos que surgen en un programa de computadora cuando, a pesar de que el código se adhiere a las reglas gramaticales del lenguaje (sintaxis), el significado o interpretación de ese código no es el que el programador pretendía. A diferencia de los errores sintácticos, que son detectados por el compilador o intérprete, los errores semánticos suelen manifestarse durante la ejecución del programa y puede llevar a resultados inesperados, incorrectos o incluso a la detención abrupta del programa.

Importancia del análisis semántico

En el análisis semántico, una etapa crucial en la compilación, se busca determinar si un programa es semánticamente correcto. Los compiladores utilizan herramientas como tablas de símbolos y análisis de flujo de control para detectar posibles errores semánticos. Sin embargo, debido a la naturaleza más abstracta de la semántica, muchos errores de este tipo pueden pasar desapercibidos hasta que el programa se ejecuta en un entorno específico.

Ejemplo simple

suma = 5 + "3"

En este ejemplo, se intenta sumar un número a una cadena de texto, lo cual es una operación semánticamente inválida en muchos lenguajes de programación.

Características de los errores semánticos.

- **Subjetividad:** La corrección semántica puede depender del contexto y de la interpretación del programador.
- **Dependencia del contexto:** El significado de un fragmento de código puede variar según el contexto en el que se utilice.
- **Dificultad de detección:** Los errores semánticos pueden ser difíciles de identificar, especialmente en programas complejos, ya que pueden

manifestarse de formas sutiles y en puntos distantes del código donde se originó el problema.

- **Consecuencias variadas:** Los errores semánticos pueden causar resultados incorrectos, comportamientos inesperados, bucles infinitos, bloqueos del programa o incluso vulnerabilidades de seguridad.

Tipos de errores Semánticos

Los errores semánticos, como hemos visto, se originan en el significado de un código no es el esperado. Estos errores pueden clasificarse en distintos tipos, cada uno con sus particularidades. A continuación, exploraremos algunos:

Errores de tipo

Se producen cuando se intenta realizar una operación con datos de tipos incompatibles.

edad = "30"

$$\text{Promedio} = (\text{edad} + 5) / 2$$

Error: no se pueden sumar un strin con un numero y luego dividir.

Errores de ámbito

Estos errores están relacionados con la visibilidad y alcance de las variables dentro de un programa. Se producen cuando se intenta acceder a una variable que no está definida en el contexto actual o cuando se utiliza una variable con el mismo nombre en diferentes ámbitos.

contador = 10

s; (0 > 1) entonces

 contador = 5 // variable local enmascarada la
 // variable global

fin-s

 imprimir(contador) // imprima 10 valor de variable
 // global

Errores de operaciones

Estos errores se producen al utilizar operadores de forma incorrecta o al realizar operaciones que no están definidas para los tipos de datos involucrados

- División por cero

resultado = 10 / 0

- Desbordamiento

int x = INT_MAX;
x++

// el valor maximo del entero se le suma 1 por lo que
// x se vuelve negativo

Técnicas de detección

La detección de errores semánticos es un proceso complejo que implica la combinación de técnicas estáticas y dinámicas.

- **Análisis estático:** se realizan antes de la ejecución del programa y se basa en el análisis del código fuente para identificar posibles errores. Algunas técnicas incluyen:
 - ▲ **Análisis de tipo:** Verificar la compatibilidad de tipos en expresiones y asignaciones.
 - ▲ **Análisis de flujo de control:** Analizar el flujo de ejecución del programa para detectar posibles bucles infinitos o condiciones inalcanzables.
 - ▲ **Análisis de alcance:** Verificar que las variables se utilicen dentro de su ámbito de definición.
- **Análisis dinámico:** Se realizan durante la ejecución del programa y consiste en monitorear el comportamiento del programa en tiempo real para detectar errores. Algunas técnicas incluyen:
 - ▲ **Pruebas unitarias:** Escribir casos de prueba para verificar el comportamiento del código en diferentes escenarios.
 - ▲ **Depuración:** Utilizar herramientas de depuración para inspeccionar el estado del programa en tiempo de ejecución.

Análisis de flujo de datos

Este análisis es una técnica utilizada para rastrear el flujo de valores a través de un programa. Se utiliza para detectar errores como:

- Variables no inicializadas: Variables que se utilizan antes de ser asignadas un valor.
- Variables con valores indefinidos: variables que pueden tener valores inesperados debido a condiciones extrañas o errores de programación.
- Uso de variables después de ser liberadas: Acceso a memoria que ya no ha sido liberada

Ejemplo de detección

Código

```
función dividir (a,b)
    resultado = a/b
    retornar resultado
```

$x = 10$

$y = 0$

```
z = dividir (x,y)
imprimir (z)
```

Proceso de detección:

1.- Análisis estático: el compilador detectaría que la variable (y) se inicializa con el valor 0

2.- Análisis de flujo de datos: este identificaría que la variable (b) en la función dividir podría tener el valor 0, lo que podría provocar una división por cero.

3.- Ejecución: Al ejecutar el código se produciría una excepción de división por cero, lo que confirmaría el error semántico detectado en el análisis estático.

Estrategias de Manejo de errores semánticos

Los errores semánticos, al ser más sutiles y difíciles de detectar que los sintácticos, requieren de estrategias específicas para su manejo. A continuación, se exploran las principales técnicas y enfoques para abordar estos errores.

Manejo en Tiempo de compilación

- Análisis semántico exhaustivo: Los compiladores modernos realizan un análisis semántico profundo para detectar la mayor cantidad posible de errores en esta etapa. Esto incluye la verificación de tipos, el análisis de flujo de control y resolución de referencias.
- Mensajes de error claros y concisos: Al detectar un error semántico, el compilador debe generar un mensaje de error que indique claramente la naturaleza del problema.
- Sugerencias de corrección: En algunos casos, los compiladores pueden ofrecer sugerencias de cómo corregir el error, lo que facilita la tarea del programador.

Manejo en tiempo de ejecución

- Excepciones: La mayoría de los lenguajes de programación modernos proporcionan mecanismos para manejar excepciones, que son eventos que

interrumpen el flujo normal de ejecución de un programa. Las excepciones pueden ser utilizadas para capturar errores semánticos y tomar acciones correctivas.

- **Aserciones:** Los aserciones son afirmaciones que el programador hace sobre el estado del programa en un punto determinado. Si una falla, se genera una excepción
- **Validación de entrada:** Esto es fundamental validar toda la entrada que recibe un programa para evitar errores semánticos causados por datos inválidos
- **Manejo de errores personalizados:** Los programadores pueden definir sus propias clases de excepción para representar diferentes tipos de errores semánticos y proporcionar información detallada.

Técnicas de recuperación

Son distintas técnicas para intentar retomar el curso del programa o proteger este.

- **Valores predeterminados:** Cuando se produce un error semántico, se puede asignar un valor predeterminado a una variable o retornar un valor por defecto de una función.
- **Lógica de respaldo:** Se puede implementar una lógica de respaldo para manejar situaciones en las que se produce un error
- **Registro de errores:** Es importante registrar los errores semánticos que ocurren durante la ejecución para facilitar la depuración y análisis posterior.

Prevención de errores Semánticos

Para prevenir este tipo de errores es necesario aplicar las siguientes recomendaciones

- **Pruebas exhaustivas:** Las pruebas unitarias, de integración y de sistema son esenciales para evitar dejar pasar estos errores.
- **Revisión de código:** La revisión por pares es una práctica recomendada para identificar posibles errores semánticos.
- **Documentación clara:** esto ayuda a los programadores a comprender mejor el propósito y funcionamiento del código.
- **Estilos de codificación:** Adoptar estilos de codificación consistentes y bien definidos para ayudar a prevenir estos errores.

Ejemplo

edad entero
escribir "ingrese su edad"
leer edad

Si edad < 0 entonces

escribir "Error: La edad no puede ser negativa"
retornar 1

fin si

Este ejemplo, se valida la entrada del usuario para asegurarse que la edad ingresada sea válida.

Errores semánticos y autómatas

Los errores semánticos tienen una estrecha relación con los autómatas

- **Construcción de autómatas:** Un error semántico en la definición de un lenguaje formal puede llevar a la construcción de un autómata incorrecto.
- **Reconocimiento de cadenas:** Un autómata diseñado para reconocer un lenguaje formal puede fallar en su tarea si el lenguaje contiene ambiguidades o si el autómata ha sido construido incorrectamente.
- **Generación de código:** Los compiladores utilizan autómatas para analizar el código fuente y generar código máquina. Un error semántico puede provocar que el autómata genere código incorrecto o que no se genere ningún código.

CONCLUSIÓN

El manejo de errores semánticos es una tarea esencial para garantizar la calidad y fiabilidad del software. Al comprender al fondo las causas y consecuencias de estos errores, los desarrolladores pueden adoptar estrategias proactivas para prevenirlas y, en caso que ocurran, identificarlos y corregirlos de manera eficiente.

Referencias

- Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools* (2nd ed.). Pearson.
- Grune, D., & Jacobs, C. J. H. (2008). *Parsing Techniques: A Practical Guide* (2nd ed.). Springer.
- Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools* (2nd ed.). Addison-Wesley.
- Muchnick, S. S. (1997). *Advanced Compiler Design and Implementation*. Morgan Kaufmann.
- Cooper, K. D., & Torczon, L. (2011). *Engineering a Compiler* (2nd ed.). Morgan Kaufmann.
- Sethi, R., & Ullman, J. D. (1977). *Principles of Compiler Design*. Prentice Hall.
- El Programa Educativo Ingeniería En Sistemas Computacionales, P. (s/f). APUNTES PARA LA MATERIA DE LENGUAJES Y AUTÓMATAS II. Tecnm.mx. Recuperado el 11 de octubre de 2024, de
<https://rinacional.tecnm.mx/bitstream/TecNM/7204/2/REPORTE%20FINAL%20LENGUAJES%20Y%20AUTOMATAS%20II.pdf>
- Moreno Sandoval, A. (2001). *Linguistica computacional*. Sintesis Editorial.
- Lenguajes y Automatas 2. (s/f). Github.io. Recuperado el 10 de octubre de 2024, de <https://byte0.github.io/Lenguajes-Automatas-II/pags/unidadUno.html>

Links a videos

Parte 1

<https://youtu.be/ltVXYK-0cJw>

Parte 2

https://youtu.be/mJUs4wQ_w20