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 PROPONER 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 EM 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 SUFICIENTE 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	:	LAII, 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 CONSECUTIVO 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_LAII-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_LAII-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.

Instituto Tecnológico de México en Celaya

Ingenieria en Sistemas Computacionales

Lenguajes y automatas 2

ACTIVIDAD 6

EQUIPO 4

PROF. RICARDO GONZÁLEZ GONZÁLEZ

MIEMBROS

- Acevedo Colunga Erick Eduardo
- Plascencia Fonseca Adrian
- Quintana Villicaña Cristian
- Rivera López Rubén Eliezer

15-10-2024

ÁRBOLES DE EXPRESIONES

Y

ACCIONES SEMANTICAS DE UN ANALIZADOR LÉXICO

Introducción

En el campo de la compilación y el procesamiento de lenguajes de programación, se requiere una profunda comprensión de la manera en que se transforman las instrucciones del código fuente en estructuras más simples que puedan ser manipuladas por la máquina. Dentro de este proceso, dos conceptos fundamentales surgen como elementos esenciales para el análisis sintáctico y semántico de los lenguajes: los árboles de expresiones y las acciones semánticas.

Los arboles constituyen una representación jerárquica y estructurada de las expresiones dentro de un programa, lo que facilita no solo su análisis sino también su posterior evaluación. Estas estructuras permiten a los compiladores descomponer las operaciones en unidades más manejables, que pueden ser optimizadas o traducidas en código ejecutable.

El propósito de esta monografía es explorar en detalle estos conceptos, centrándonos en su relevancia dentro del diseño y desarrollo de compiladores y otras herramientas de procesamiento de lenguajes de programación. Analizaremos cómo los árboles de expresiones y las acciones semánticas interactúan dentro del proceso de análisis, así como las ventajas y desafíos que presentan en diferentes aplicaciones.

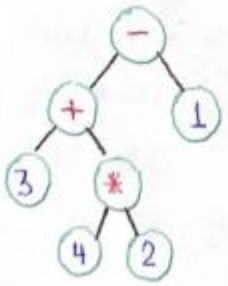
Desarrollo

Árboles de Expresiones

Un árbol de expresiones es una estructura de datos jerarquica que permite representar de manera estructurada expresiones matemáticas o lógicas. Cada nodo de este árbol puede representar un operador o un operando. Esto significa que los operadores con mayor precedencia se encuentran más abajo en el árbol, mientras que aquellos con menor precedencia se ubican más cerca de la raiz. Este tipo de representación facilita la evaluación de expresiones complejas y respeta las reglas fundamentales de precedencia y asociatividad.

Un ejemplo es el siguiente: 3 + 4 * 2 -1

Su arbol de precedencia seria el siguiente:



Tipos de nodos en los arboles de expresiones

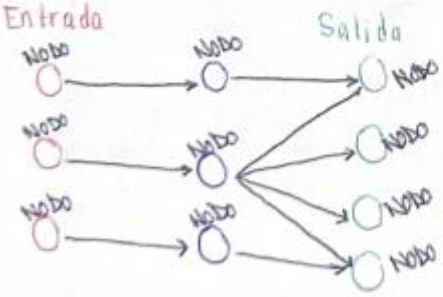
En los arboles de expresiones, los nodos se pueden clasificar en:

- Nodos de operadores; que representan operaciones como suma, resta, multiplicación o división. También incluyen operadores lógicos y de comparación.

- Nodos de operandos, que son los valores constantes o variables en una expresión. Estos pueden ser números, literales de cadena o variables que almacenan datos

- Nodos de funciones, los cuales representan llamadas a funciones dentro de una expresión.

- Nodos de control de flujo; que aparecen en estructuras más complejas, como en las expresiones condicionales o bucles.



Ejemplo de un grafo, este ejemplo NO muestra un arbol sin embargo muestra nodos.

Un nodo es un punto de conexión física o lógica en una red informática o en una estructura de datos.

Aplicaciones de los árboles de expresion

Los arboles de expresiones pueden ser aplicados en diversas áreas del desarrollo de compiladores:

- Evaluación de expresiones matemáticas: Permiten la evaluación jerarquica de operaciones matemáticas, respetando la precedencia y asociatividad de los operadores.
- Optimización de código: Los compiladores utilizan árboles de expresiones para optimizar el código. Pueden plegar constantes o eliminar subexpresiones comunes, reducrendo el número de cálculos innecesarios.
- Bases de datos: En sistemas de gestión de bases de datos, los árboles se usan para representar y optimizar consultas SQL.
- Álgebra computacional: En software como Mathematica o Maple, se usan árboles para simplificar y manipular expresiones algebraicas.
- Generación de código: Los árboles de expresiones son crucrales para transformar las instrucciones en código de máquina o bytecode.

Ventajas y Desafíos de los Árboles

Los arboles de expresiones ofrecen una serie de beneficios clave. Representan claramente la estructura jerarquica de una expresión, lo que permite la manipulación de expresiones complejas de una manera comprensible y organizada

Además facilitan la optimización del código, ya que proporcionan una estructura que puede ser recorrida y modificada para reducir redundancias o mejorar la eficiencia del programa.

Sin embargo, presentan también ciertos desafíos. La construcción de árboles para expresiones largas puede volverse compleja y, en casos extremos, puede consumir una gran cantidad de memoria. Además, para realizar ciertas operaciones, es necesario utilizar recorridos específicos como in-order, pre-orden o post-orden, lo que puede añadir un nivel adicional de complejidad. También en lenguajes con tipos complejos, los árboles de expresiones deben manejar adecuadamente la información sobre los tipos de datos

Acciones Semánticas de un Analizador Sintáctico

Las acciones semánticas de un analizador sintáctico desempeñan un papel fundamental en la construcción de un compilador, ya que proporcionan coherencia y significado a las estructuras que han sido validadas sintácticamente. Mientras que el análisis sintáctico se centra en asegurar que la estructura del código cumple con las reglas formales de la gramática y el análisis semántico verifica que las operaciones y expresiones en el programa tengan

sentido lógico y respeten los reglas del lenguaje en cuanto a tipos, variable, funciones y flujo de control.

Dependiendo del tipo de sentencias, las acciones semánticas, las acciones pueden agruparse:

* Sentencias: de declaración: completar la sección de tipos de la tabla de símbolos

* Sentencias ejecutables: realizar comprobaciones de tipos entre los operandos implicados

* Funciones y procedimientos: comprobar el número, orden y tipo de los parámetros actuales en cada llamada a una función o procedimiento

* Identificación de variables: comprobar si el identificador ha sido declarado antes de utilizarlo.

Etiquetas: comprobar si hay etiquetas repetidas y validación

Constantes: comprobar que no se utilizen en la parte izquierda de una asignación

Conversiones: y equivalencias de tipo: verificación.

Sobrecarga de operadores y funciones: detectar y solventar

Análisis Semántico y Comprobaciones en el proceso de Compilación

El análisis semántico es una fase crítica en el proceso de compilación, ya que se encarga de revisar el programa fuente para identificar errores semánticos y asegurar que el código tenga un significado correcto de acuerdo con las reglas del lenguaje de programación. A diferencia del análisis sintáctico, qué solo se preocupa por la estructura del código. En esta etapa se describen las principales comprobaciones que se llevan a cabo durante el analisis semántico.

1- Verificación de Tipos de Expresiones

Una de las comprobaciones fundamentales en el análisis semantico es la verificación de tipos. Esta se realiza asignando un valor de tipo a cada uno de los componentes léxicos, como variables, literales y operadores. Estos valores de tipo son entonces comparados para verificar que las operaciones y expresiones sean coherentes y que los tipos involucrados sean compatibles. Si los tipos no coinciden, no se pueden realizar los cálculos o las operaciones indicadas.

Coerción y conversión de tipos

A menudo, puede haber situaciones en las que se desee almacenar un valor de un tipo de dato en una variable de un tipo diferente. Dependiendo del lenguaje de programación, algunas conversiones de tipos se hacen de

forma automática, lo que se conoce como "coercion". En otros casos se requiere una conversión explícita

2- Conversiones del flujo de control

El flujo de control en un programa define el orden en el que se ejecutan las instrucciones. Es fundamental que las estructuras con las que se controla este flujo, como bucles y las sentencias condicionales, sean semánticamente validas. Por ejemplo una sentencia "break" en un lenguaje como C debe hacer que el control abandone la estructura de control más cercana que la engloba, ya sea un bucle "while", "for", o una sentencia "switch"

3- Comprobación de unicidad

La unicidad es una comprobación que asegura que cada identificador, como una variable o función, sea único dentro de su ámbito. Un error común en este sentido es declara una variable o función, sea único dentro de su ambito. Un error común en este sentido es declarar una variable con el mismo nombre más de una vez dentro del mismo bloque de código.

4- Comprobaciones relacionadas con nombres

En ciertos contextos, como en bloques de transacciones, el mismo nombre puede aparacer más de una vez. El compilador debe asegurar que estos nombres sean coherentes y no generen conflictos, especialmente en lenguajes que

permiten nombres repetidos en diferentes niveles de ámbito.

5- Comprobación Estática y Dinámica

Las comprobaciones pueden dividirse en dos tipos principales:

- Comprobación estática: Se realiza en tiempo de compilación, verificando errores como la incompatibilidad de tipos sin necesidad de ejecutar el programa.

- Comprobación dinámica: Ocurre en tiempo de ejecución, detectando errores que solo se manifiestan cuando el programa está corriendo, como desbordamiento de arreglos.

6. Sistema de Tipos

Un sistema de tipos establece reglas para asignar tipos a las expresiones del programa. Puede incluir tipos simples (int, float). o más complejos, como clases. Las operaciones deben ser coherentes con los tipos involucrados, por ejemplos, la suma de 2 reales da un real y la concatenación de caracteres produce una cadena.

Importancia de las Acciones Semánticas en el proceso de compilación

Las acciones semánticas son cruciales porque permiten al compilador verificar no solo la estructura, sino también el significado del código. Esto garantiza que un programa no solo sea sintácticamente correcto sino también lógicamente coherente. Por ejemplo, pueden detectar errores en expresiones de bucles mal formados o inconsistencias en tipos de datos. Además, estas acciones contribuyen a la optimización del código, eliminando variables innecesarias y evaluando expresiones constantes, mejorando así el rendimiento final del programa.

Conclusión

Los arboles de expresiones y las acciones semánticas representan dos pilares fundamentales en el diseño de compiladores y herramientas de procesamiento de lenguajes de programación. Los árboles de expresiones proporcionan una estructura clara y jerárquica que facilita la evaluación y manipulación de las expresiones de un programa. Su papel en la optimización y generación de código es esencial, y su versatilidad, les permite ser usados en una variedad de aplicaciones más allá de los compiladores tradicionales.

Por otro lado, las acciones semánticas son una herramienta poderosa que permiten extender el análisis sintáctico a un nivel semántico, asegurando que el código fuente no solo tenga una estructura correcta, sino también un significado coherente.

A medida que los lenguajes de programación continúan evolucionando, es probable que los principios detrás de los árboles de expresiones y las acciones semánticas se mantengan relevantes.

Referencias

1:- Computerphile. (2016, Octubre 5). Compilers: Crash Course Computer Science #21 [Video]. Youtube. https://www.youtube.com/watch?v=_u-rOQ_Bmro

2:- Geekific. (2020, Marzo 19). Semantic Analysis in Compilers [Blog Post]. Geekific https://www.geekific.com/semantic_analysis-in-compilers/

3:- GeeksforGeeks. (2023, Enero 20). Semantic Design Analysis in Compilers. GeekGeekforGeeks. https://www.geekforgeeks.org/semantic-analysis-in-compilers-design

4:- Williams, A. (2021, Junio 3). Statyc vs Dynamic. Typing in Programming Languges [Video]. Youtube. https://www.youtube.com/watch?v=1tTgZpCnmxI

5:- Free Code Camp. (2021, Abril 17). How Does a Compiler Work? - FrontEnd Masters [Video]. Youtube. https://www.youtube.com/watch?v=TSVv_VcnRI

6:- Tutorials Point. (2023). Phases of a Compiler. TutorialsPoint. https://www.tutorialspoint.com/compiler_design/compiler_design-phases_of_compiler.htm

4.3 Comprobaciones de Tipos de Expresiones.

Tabla de Contenidos:

1. Introducción
2. Fundamentos de la Comprobación de Tipos
   - 2.1. Definición y propósito
   - 2.2. Importancia en los Lenguajes de p.
3. Tipos de Datos y Operadores
   - 3.1 Tipos de datos en los Lenguajes
   - 3.2 Operadores y Consistencia de Tipos
4. Sistemas de comprobación de tipos
   - 4.1 Comprobación Estática
   - 4.2 Comprobación Dinámica
5. Verificación e Inferencia de Tipos
   - 5.1 Verificación de Tipos
   - 5.2 Inferencia de Tipos
6. Comprobación de Tipos en Autómatas y...
   - 6.1 Rol de los Autómatas en la Comprob.
   - 6.2 Gramáticas y Reglas de producción para..
7. Errores Comunes y Soluciones
   - 7.1 Errores sintácticos y semánticos
   - 7.2 Ejemplos de errores de tipos y Correcc.

8. Conclusión
9. Referencias.

1. Introducción

La comprobación de tipos de expresiones, en el contexto de los lenguajes de programación, tiene como objetivo garantizar que los tipos de datos utilizados en operaciones y funciones sean consistentes y compatibles. Este proceso asegura que las operaciones aritméticas, lógicas o de otro tipo respeten las reglas de tipificación definidas en el lenguaje, evitando errores que podrían surgir durante la ejecución del programa.

En este trabajo, exploraremos cómo se lleva a cabo la comprobación de tipos en diferentes contextos, los distintos enfoques (estático y dinámico) que pueden adoptarse, y la importancia de esta verificación para la creación de código eficiente y robusto. Además, analizaremos cómo los autómatas y los lenguajes formales juegan un papel en este proceso, y presentaremos ejemplos concretos y como solucionarlos.

2. Fundamentos de Comprobación de Tipos.

2.1 Definición y propósito

La comprobación de tipos es el proceso que asegura que los datos utilizados en una expresión son compatibles entre sí, siguiendo las reglas de tipificación definidas en el lenguaje de programación. Cada lenguaje cuenta con un sistema de tipos que determina las operaciones válidas entre datos de diferentes tipos. Por ejemplo, no es válido sumar un entero (int) con una cadena (string) sin haber realizado la conversión adecuada.

2.2 Importancia en los Lenguajes de programación

La comprobación de tipos garantiza la consistencia y coherencia en las operaciones de un programa. Al prevenir que tipos incompatibles se combinen, se reduce la posibilidad de errores en tiempo de ejecución y se mejora la eficiencia del código generado por el compilador. Esto es particularmente importante en lenguajes que apuntan a un alto rendimiento o que buscan minimizar errores críticos en sistemas en tiempo real o industriales.

3. Tipos de Datos y Operadores

3.1 Tipos de Datos en los Lenguajes:

Los tipos de datos más comunes en los lenguajes modernos son:

- Enteros (int): Números sin decimales
- Flotantes (float): Números con punto decimal
- Doubles (double): Números de mayor precisión que los float
- Cadenas (string): Secuencias de caracteres
- Booleanos (bool): Valores verdadero o falso

Cada tipo está asociado a un conjunto de operaciones válidas que se pueden realizar.

3.2 Operadores y Consistencia de Tipos:

Los operadores (suma, resta, multiplicación, comparación, etc.) deben aplicarse a operandos que sean compatibles en cuanto a tipos. Por ejemplo:

```
int a = 5;
float b = 4.2;
```

En la operación "a + b", el lenguaje debe definir cómo manejar la suma entre un entero y un flotante. Algunos lenguajes permiten la promoción de tipo (convierten el int a float), mientras que otros podrían requerir una conversión explícita por parte del programador.

4. Sistemas de Comprobación de Tipos

4.1 Comprobación Estática:

La comprobación estática se realiza durante el proceso de compilación del programa, antes de que se ejecute. Si se detecta un error de tipo, el código no se compila y, por lo tanto, no puede ejecutarse. Esto ofrece un alto grado de seguridad y fiabilidad. Un ejemplo podría ser:

```
float a = 3.14;
int b = 2;
b = a;
```

El error en este código es que los tipos de operandos deben ser compatibles en una expresión aritmética (dependiendo de la definición del lenguaje). Lo que significa que no puedo asignar un entero a un float sin hacer antes la conversión (casting).

4.2 Comprobación Dinámica

La comprobación dinámica, por otro lado, ocurre en tiempo de ejecución. Es utilizada por lenguajes atipificados como python, JS y php, donde el tipo de las variables puede cambiar en tiempo de ejecución.

Aunque ofrece mayor flexibilidad, puede introducir errores durante la ejecución si no se gestionan correctamente los tipos, además de afectar el rendimiento.

```
x = "Hello"
x = 10
```

Es correcto ya que el tipo cambia durante la ejecución.

5. Verificación e Inferencia de Tipos

5.1 Verificación de tipos

La verificación de tipos garantiza que los tipos utilizados en las expresiones sean coherentes con la declaración original de los datos. Por ejemplo, si se declara una variable como un número entero, cualquier operación que involucre esa variable debe respetar ese tipo. En los lenguajes de tipado fuerte, cualquier discrepancia (lenguajes de) genera un error.

5.2. Inferencia de tipos

Este es un proceso mediante el cual el compilador deduce el tipo de una variable o expresión sin que el programador tenga que especificarlo explícitamente. Este enfoque se utiliza en lenguajes como Haskell y Scala. La inferencia se basa en las reglas gramaticales del lenguaje y el contexto en que se usen las variables.

```
x = 5
```

El compilador infiere que es un entero (int).

6. Comprobación de Tipos en Autómatas y Lenguajes Formales

6.1. Rol de los Autómatas en la comprobación de tipos

Los autómatas y las gramáticas son de gran importancia en la comprobación de tipos, especialmente durante la fase de análisis semántico en un compilador. Los autómatas de pila se utilizan para procesar expresiones aritméticas y verificar su consistencia tipológica, asegurando que los operandos y operadores se combinen correctamente.

6.2 Gramáticas y Reglas de Producción para la tipificación

Las gramáticas Libres de contexto (GLC) son utilizadas para definir reglas sintácticas de los lenguajes de programación. Estas reglas no solo cubren la estructura de las expresiones, sino que también las relaciones de tipos, deben cumplirse. Un árbol sintáctico generado a partir de una expresión puede utilizarse para verificar los tipos de sus nodos y garantizar que las operaciones sean válidas.

7. Errores Comunes y Soluciones

7.1 Errores sintácticos y semánticos relacionados con tipos

Un error común es intentar asignar un valor de un tipo más grande a una variable de un tipo más pequeño sin una conversión explícita. Por ejemplo, asignar un valor double a una variable float puede provocar un error en un tipo y si no se castea correctamente puede llevar a la pérdida de información.

Por ejemplo

```
float a = 3.14;
double b = 6.28;
a = b;
```

Error, ya que estoy intentando ingresar 8 bytes en el espacio de 4 bytes, habrá perdida de información.

7.2 Ejemplos de errores de tipos y su corrección

```
float a;
double b = 6.28;
a = (float) b;
```

Conversión Explícita para evitar errores de tipo.

8. Conclusión:

La comprobación de tipos de expresiones es un proceso esencial para garantizar la robustez, seguridad y eficiencia del código en los lenguajes de programación. Al asegurar que las operaciones entre tipos de datos sean consistentes y este conocimiento será de vital importancia para su implementación en nuestro propio compilador.

9. REFERENCIAS:

- Fernando Acosta. (2020, 25 noviembre). Comprobación de tipos [video]. youtube. https://www.youtube.com/watch?v=QXdp_gSCk9Y
- Desconocido. Comprobación de tipos. https://3e3447535705b1.site123.me/unidad-i-%C3%81analisis-sem%C3%AAntico/13-comprobaciones-tipos-expresiones

• gilbert adoni tovar salazar. (2015, 3 octubre) COMPROBACIÓN DE TIPOS [video]. Youtube. https://www.youtube.com/watch?v=-SkGQ9i-pag

4.4 PILA SEMÁNTICA EN UN ANALIZADOR SINTÁCTICO

TABLA DE CONTENIDO

1. Introducción
2. ¿Qué es una Pila en Estructuras de Datos?
   2.1 Definición y Características
   2.2 Ejemplos de uso de pilas.
3. El Analizador sintáctico
   3.1 Función del Analizador sintáctico en la Com
   3.2 Relación entre el Analizador sintáctico y la pila
4. La Pila Semántica
   4.1 Propósito de la Pila Semántica
   4.2 Funcionamiento de la Pila Semántica.
5. Ejemplo de Validación de Paréntesis
6. Conclusión
7. Referencias.

1. INTRODUCCIÓN

En el análisis sintáctico y semántico de un lenguaje de programación, las pilas son las encargadas de mantener el control sobre la estructura de las expresiones y las reglas del lenguaje. En particular, la pila semántica es una herramienta crítica dentro del analizador sintáctico para evaluar y verificar que las expresiones siguen las reglas gramaticales y semánticas adecuadas.

Esta monografía aborda el concepto de pila semántica en el análisis sintáctico, explicando su función, cómo se relaciona con las pilas en estructura de datos, y su importancia en la verificación de la estructura de un programa.

2. ¿Qué es una pila en Estructuras de Datos?

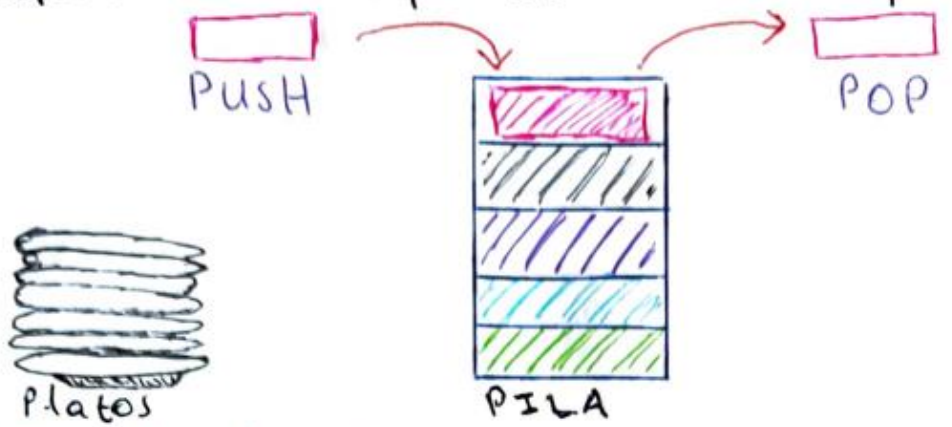
2.1 Definición y Características

Una pila es una estructura de datos que sigue el principio LIFO (Last In, First, Out), lo que significa que el último elemento en ser agregado a la pila será el primero en salir. Solo se permite el acceso a un lado de la estructura, denominado cima o tope de la pila.

Las operaciones básicas que se pueden realizar en una pila son:

- Push : Insertar un elemento en la cima de la Pila
- Pop : Eliminar el elemento en la cima de la Pila.

Representación simplificada de una pila:



2.2 Ejemplo de Uso de Pilas

Una de las analogías más comunes que nos permitirá entender de mejor manera el funcionamiento de las pilas es imaginarla como una pila de platos, donde, el último plato que colocas será el primero en ser retirado, ya que, si extraes uno de enmedio podrías tirar y romper los de arriba.

En programación las pilas cumplen con diversas funciones:

- Evaluación de expresiones: Permiten procesar y evaluar expresiones matemáticas con operadores y operandos.
- Recursión: Las llamadas recursivas a funciones suelen manejarse con pilas, donde cada nueva llamada se apila y se desapila cuando se termina de ejecutar.

En los compiladores, las pilas también son escenciales para tareas más avanzadas como la verificación del balance de paréntesis o el manejo de estructuras anidadas.

3. ANALIZADOR SINTÁCTICO

3.1 Función del Analizador Sintáctico en la compilación

El analizador sintáctico es una de las fases del compilador, responsable de verificar la estructura de un programa en base a las reglas gramaticales del lenguaje. Recibe como entrada los tokens generados por el analizador léxico y organiza estos tokens en un árbol de derivación que refleja la jerarquía y estructura del programa.

La función principal del analizador sintáctico es asegurarse de que la secuencia de tokens es válida, lo que implica comprobar que las expresiones y sentencias del programa respetan las reglas sintácticas definidas en el lenguaje.

3.2 Relación entre el Analizador sintáctico y la pila

El analizador sintáctico utiliza una pila para manejar de manera eficiente las estructuras anidadas que pueden aparecer en el código fuente, como los paréntesis, los corchetes, o los bloques de código. La pila ayuda a realizar un seguimiento de los elementos que se están procesando, permitiendo la correcta formación del árbol de derivación.

Una pila es particularmente útil en un autómata de pila, ya que permite manejar de manera eficaz gramáticas libres de contexto.

que son utilizadas en la mayoría de los lenguajes de programación. Un autómata de pila puede almacenar símbolos temporalmente mientras analiza una cadena de tokens, permitiendo verificar que las expresiones anidadas son válidas.

4. La Pila Semántica:

4.1 Propósito de la Pila Semántica

La pila semántica es una extensión de la pila que se usa en el análisis sintáctico, pero con un propósito más específico: evaluar el significado de las expresiones y sentencias, no solo su estructura. Mientras que la pila del analizador sintáctico se encarga de validar la correcta estructura gramatical, la pila semántica asegura que las operaciones dentro de esas estructuras sean coherentes a nivel de significado.

Por ejemplo, en una expresión como "a + b", la pila semántica no solo verifica que el operador + esté correctamente colocado entre dos operandos, sino que también dichos operandos sean compatibles en tipo (e.g., que ambos sean numéricos y no una mezcla entre un número y una cadena de texto).

```
int a = 5
int b = 3
int c = a + b
```

```
int a = 5
String b = "3"
int c = a + b
```

4.2 Funcionamiento de la Pila Semántica:

1. Almacenamiento temporal de tokens: A medida que el analizador sintáctico recorre una expresión, los tokens que representan valores, operadores y delimitadores (como paréntesis) son empujados a la pila semántica.

2. Evaluación de reglas semánticas: En lugar de desapilar simplemente al finalizar el recorrido, la pila semántica evalúa reglas semánticas conforme avanza. Por ejemplo, cuando se encuentra con un operador, verifica si los operandos adyacentes son compatibles en tipo antes de permitir la compilación o ejecución.

Por lo tanto, la pila semántica contribuye a la construcción de un programa robusto y coherente, asegurando que las operaciones entre diferentes tipos de datos sean válidas y que la lógica interna sea consistente.

5. Ejemplo de validación de paréntesis

Para ilustrar como se utiliza la pila en un analizador sintáctico para validar el orden de los paréntesis podemos considerar la siguiente expresión matemática:

(9 + 4 * (4/2) * (4+1))

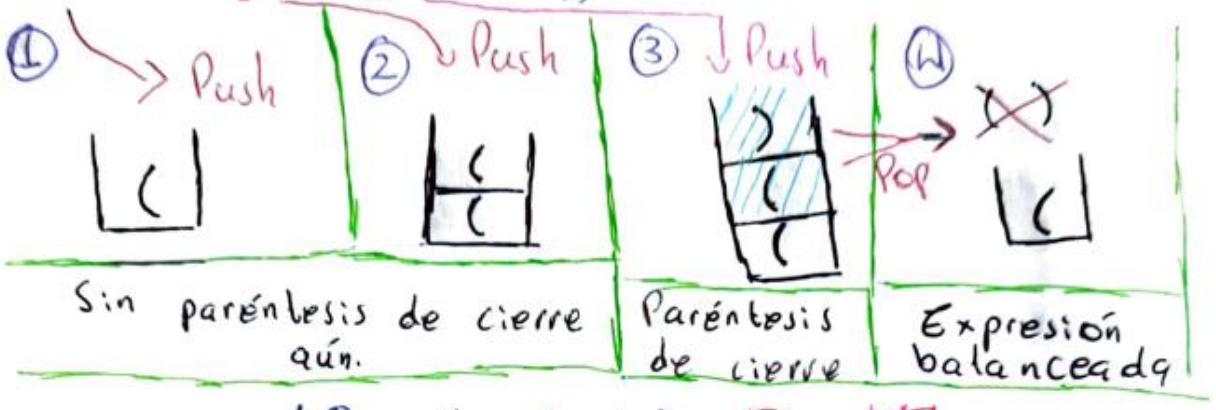
El objetivo es asegurarnos de que los paréntesis están correctamente balanceados, es decir que cada paréntesis de apertura tiene un correspondiente paréntesis de cierre y su orden es correcto.

Algoritmo de Pila:

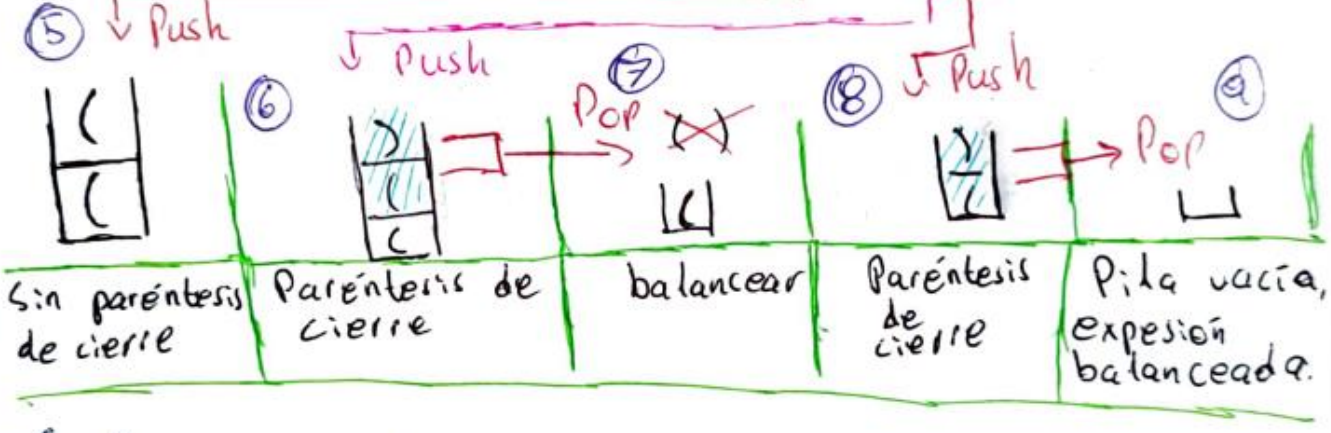
1. Recorrer la expresión de izquierda a derecha.
2. Cada vez que encontremos un paréntesis de apertura "(" lo insertamos en la pila (push)
3. Si encontramos un paréntesis de cierre ")" verificamos:
   - Si la pila está vacía, significa que este paréntesis de cierre no tiene su correspondiente apertura, lo cual es un error.
   - Si la pila no está vacía, retiramos el elemento en la cima junto con el siguiente elemento (su correspondiente apertura), lo que significa que ha sido cerrado correctamente.
4. Al final la pila debería estar vacía, de lo contrario significaría que un paréntesis no tiene su pareja opuesta.

Demostración grafica:

[(a + 4 * [(4/2)] * (4+1))



(a + 4 * (4/2) * [(4+1)])



6. CONCLUSIÓN:

La pila semántica es una herramienta crucial en el analizador sintáctico que permite no solo verificar la correcta estructura de una expresión, sino también evaluar su validez semántica. Al manejar eficientemente las estructuras anidadas y los tipos de datos, la pila semántica garantiza que las operaciones dentro de un programa sean consistentes y coherentes evitando errores a nivel de significado antes de la ejecución del código. Estos conocimientos servirán como base para la creación de nuestro propio compilador seguro, eficiente y robusto.

7. REFERENCIAS

- Desconocido. Pila semantica en Analizador Sintactico. Sitio web: https://en línea.zacatecas.tecnm.mx/pluginfile.php/10833/mod_resource/content/1/1.4%20Pila%20Semantica%20un%20analizador%20sintactico.pdf.
- Fernando Acosta. (2020 b, noviembre 25). Pila Semántica en un Analizador Sintáctico [video] Youtube. https://www.youtube.com/watch?v=NPZwq1mscM
- Diego Martínez Zavala (2020, 14 julio). Pila semántica en unanalizador Sintáctico [video]. Youtube. https://www.youtube.watch?v=36Cf22-1DNY

4.5

Índice

4.5 Esquema de traducción 1
4.5.1 Creación de esquemas de traducción 3
4.5.2 Conclusiones 6

Índice de figuras

Figura 1 1
Figura 2 1
Figura 3 2
Figura 4 2
Figura 5 3
Figura 6 4
Figura 7 5

4.5 Esquema de traducción

Un esquema de traducción es una gramática independiente de contexto, también conocidas como gramaticas de tipo 2, en la cual se ha agregado un fragmento de código en la parte derecha de las reglas de producción, las cuales se conocen como acciones semánticas y van encerradas entre llaves.

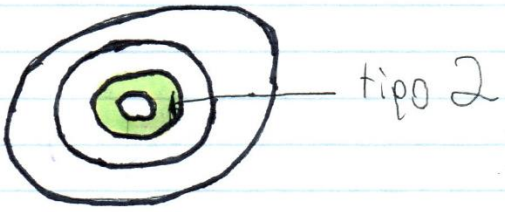


Figura 1.. Gramática de tipo 2, de acuerdo a la jerarquía de Chomsky

Con esto, se tiene que se componen de dos partes, las reglas de producción, que es la parte sintáctica del lenguaje, y las acciones, que representan la parte semántica

$$A \rightarrow A\{accion\}B$$

Figura 2.. Uso de los esquemas

Dichos fragmentos son los que modifican y calculan el valor de los atributos que están relacionados con los nodos que tiene el árbol sintáctico, por lo que si bien, una acción pudiera llegar a tener una

sintaxis adecuada, podría llegar a ser incorrecta si semánticamente contara con algun error. Para obtener un ejemplo simple pero claro de esto hice un pequeño programa en python.

léxico sintáctico semántico

```
def funtion-prueba():
    val1 = 123
    val2 = "prueba"
    val3 = val1 + val3
    print(val3)

funtion-prueba()
```

Figura 3. Código en python con un error de tipo de dato

Este codigo cumple con los tipos de análisis propuestos hasta ahora, sin embargo, no pasa el tercero, dado que no hay tal cosa como una suma entre un entero y cadena, lo cual es el tipo de propuestas que tienen estos esquemas mediante sus acciones.

Regresando a los esquemas, estos se evaluan revisando primero las partes profundas del árbol sintáctico.

Reglas

E -> E+E
E -> id
E -> num

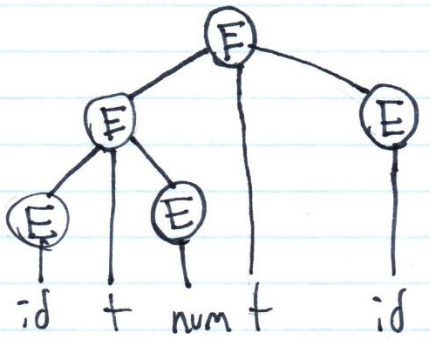


Figura 4. Generación de un árbol sintáctico

4.5.1 Creación de esquemas de traducción

Ahora bien, la gracia de esta herramienta no es únicamente para su analisis, sino para poder implementarla, por lo que hay que ver el proceso que se debe llevar a cabo.

Los requisitos son:
1- Conocer el lenguaje a utilizar
2- Conocer los simbolos gramaticales
3- Saber usar las acciones semánticas
4- Formar las reglas

Hay que destacar que se deben hacer en ese orden, como si de un tren se tratara.

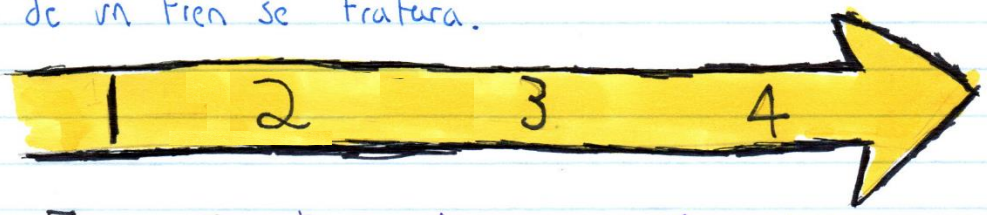
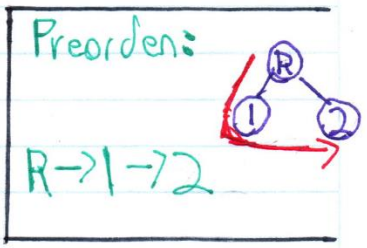


Figura 5. Representación gráfica de los requisitos

Una vez se tenga esto, se van a usar las reglas de produccion, las cuales se derivarán en un árbol sintactico y se le dará un recorrido en preorden.



1.- Construir el árbol sintáctico sin tomar en cuenta las acciones semánticas
2.- A los simbolos de derivación agregarles las acciones que deben realizar
3.- Recorrer en preorden el árbol, ejecutando las acciones.

Para reforzar lo antes visto, lo mejor será poner un ejemplo en el que se implemente esto, el cual será la expresión 7 + 4 * 5

Las reglas a usar serán las siguientes:

P -> E
E -> E + E
E -> E * E
E -> (E)
E -> digito

Son un conjunto pequeño de reglas, pero servirán bien para la explicación

Una vez se tiene esto, lo primero será hacer una derivación simple, lo cual nos dará un árbol con cada uno de los elementos necesarios para desarrollar la expresión.

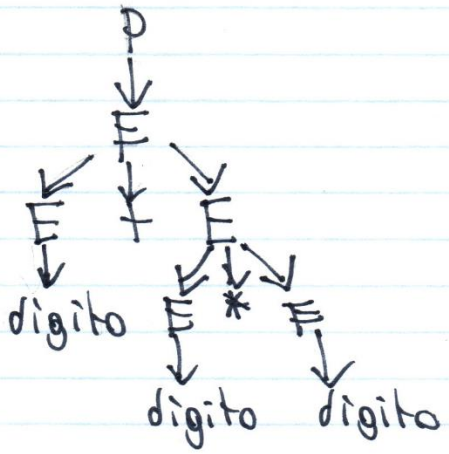


Figura 6 Árbol de derivación para llegar a la expresión

Lo siguiente es hacer uso de este árbol para la generación correcta de las acciones.

Tras hacer el recorrido, siempre tomando la raiz como primer objetivo, el árbol evolucionó de la siguiente forma.

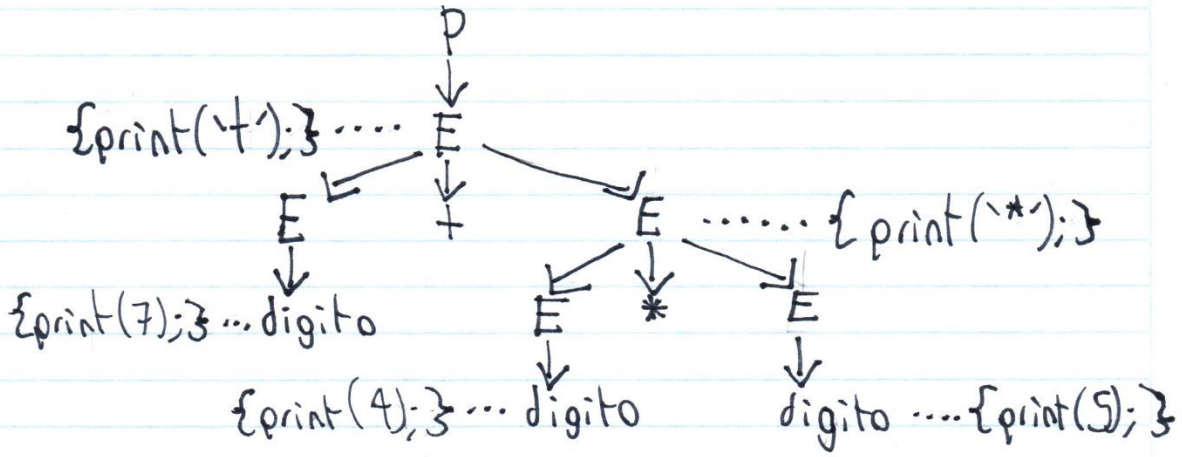


Figura 7. Evolución del árbol de derivación

Y respecto a las oreglas, quedan de la siguiente forma.

P -> E
E -> {print('+');} E + E
E -> {print('*');} E * E
E -> (E)
E -> digito { print(digito . lexval); }

Una vez se tiene esto, se puede dar por concluida la creación de los esquemas de traducción, los cuales trabajan en conjunto con los demás tipos de herramientas para la creación de compiladores e intérpretes.

4.5.2 Conclusiones

Las acciones semanticas vistas previamente son una parte indispensable en el diseño y desarrollo de compiladores e interpretes debido a que son los encargados de que el análisis léxico y sintactico tengan tanto un significado como un funcionamiento, con ello siendo capaz de transformar código unicamente representativo en uno funcional, asegurando los tres análisis indispensables para estas tecnologías.

Fuentes

Marc M. (2004, 02 de diciembre) Translation Schemes. Western University.
https://www.csd.uwo.ca/~mmorenom/CS447/Lectures/Translation.html/node3.html

Anónimo (recuperado el 13/10/2024). Esquemas de traducción Tecnológico Nacional de México en Zacatecas.
https://enlinea.zacatecas.tecnm.mx/pluginfile.php/10834/mod_resource/content/1/1.5 Esquemas de traducción.pdf

Casiano R. (2006, 21 de febrero) Esquemas de traducción. Universad de La Laguna
https://cc.etsii.ull.es/ftp/antiguo/asignas/casiano/node391.html

4.6

Índice

4.6 Generación de tabla de símbolos y direcciones 1
4.6.1 Creación 2
4.6.2 Contenido y operaciones 3
4.6.3 Ventajas y desventajas 5
4.6.4 Ejemplo 6
4.6.5 Conclusión 8

Índice de Figuras

Figura 1 1
Figura 2 4
Figura 3 4
Figura 4 5
Figura 5 6
Figura 6 7

4.6 Generación de tabla de símbolos y direcciones

Antes de avanzar al tema que nos compete, que es la creación de la tabla en sí, hay que revisar brevemente que es y cuales son sus características.

La tabla de símbolos es uno de los elementos de mayor importancia en la creación del compilador, ya que ésta estructura de datos se usa para definir donde están los identificadores que son usados en el programa. Esto además de permitir usar los identificadores, ayuda al compilador al manejo de errores y a la optimización de código.

Contiene todos los identificadores, tanto de variables como funciones, que son usados en el programa, junto con varias características. La tabla se construye en las etapas más tempranas de la compilación y ayuda a asegurar el uso correcto de todos los identificadores de acuerdo con las reglas del lenguaje.

ID	Type	Address

Figura 1.
Tabla representativa del tema

4.6.1 Creación

Se realiza en las fases de análisis, pero es necesario que sea con los tres, para poder identificar diferentes cosas:

- Léxico: Lo identifica como identificador.
- Sintáctico: Añade información como el tipo de dato, el tamaño, el alcance, etc.
- Semántico: Usa la información disponible y verifica que las expresiones son adecuadas y las actualiza.

Ahora bien, la información que es recolectada en las fases antes mencionadas es usada en las fases de síntesis con el fin de generar código intermedio, ser optimizado y al final obtener el código objeto.

Así que una vez sabemos como se obtiene, veamos para que sirve.

- Generación de código intermedio: Al hacer referencia es capaz de saber el tipo de dato y cuánto tiempo se usa, así como el para que se usa.
- Optimización de código: Usa la información obtenida para optimizar el código, limpiando los símbolos y funciones sin trascendencia dentro de la operación.
- Código objeto: Usando la información de la tabla genera el código final.

4.6.2 Contenido y operaciones

Una vez que se ha hecho esto se puede realizar un seguimiento a lo que ha sido almacenado en la tabla, pero ¿qué cosas se pueden almacenar en ella?

La respuesta a eso es:

- Variables y constantes
- Procedimientos y funciones
- Etiquetas del lenguaje
- Valores temporales generados

Todo esto es lo que se va a almacenar en la tabla de lo cual interesa conocer sus valores, para diferentes motivos, como pueden ser: referencia a una variable, reconocer el tipo de valor del identificador, y saber la dirección en memoria y su tamaño. Con lo cual la información que se almacena es la siguiente:

- Nombre
- Tipo (variable, constante, función, procedimiento).
- Dirección (se indica donde se almacena y de forma implícita el tamaño disponible).
- Alcance (si es global o local, aunque esto cambia un poco dependiendo del lenguaje).
- Un poco de información adicional, que ayuda más que nada al desarrollador y poder encontrar errores.

Así mismo, las operaciones que se usan en la tabla son:

Operaciones
- Insertar
- Borrar
- Buscar
- Actualizar

Figura 2. Cuadro sinóptico de las operaciones

Todas ellas siempre haciendo referencia a los objetos que posee.

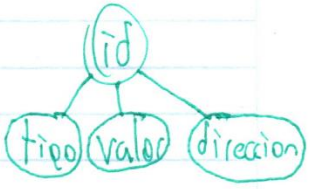
La tabla en cuestión se puede implementar con diferentes estructuras de datos, como puede ser:

Almacenamiento

- **lista**: Se usa un arreglo para almacenar la información necesaria de los objetos que se desea almacenar

id1	inf1	id2	inf2	id3

- **árbol**: Mediante nodos se es capaz de almacenar el contenido



- **matriz**: Mediante un arreglo bidimensional se pueden almacenar los valores deseados

id1	inf1
id2	inf2
id3	inf3

Figura 3. Cuadro sinóptico de los tipos de implementación.

4.6.3 Ventajas y desventajas

Ya para terminar la parte teorica, repasemos las ventajas y desventajas que tiene la herramienta antes revisada.

Ventajas	Desventajas
La eficiencia del programa puede ser aumentada gracias a mantener en orden los valores y su ubicación	Aumenta la cantidad de memoria necesaria para ejecutarse el programa
Se puede optimizar el código, retirando los segmentos innecesarios	Incrementa el tiempo de procesamiento para obtener la tabla
Al estar optimizado y ordenado, el código se ejecuta con mayor velocidad	Aumenta la complejidad del compilador, dado que si bien las tablas las genera el compilador, el que lo hace posible es el desarrollador
Permite migrar el código, ya sea entre sistemas, o (con ayuda del desarrollador) entre lenguajes	Si requiere una cantidad de datos bastante alta no es viable
Permite encontrar errores en las ejecuciones de forma más sencilla	Darle mantenimiento puede ser algo complicado

Figura 4. Cuadro comparativo de ventajas y desventajas

Esta herramienta, si bien presenta varias desventajas. lo cierto es que al final del día (y así como es la razón de estarla revisando) es una herramienta útil si se le sabe usar, por lo que centrarnos en el aspecto negativo es importante, para conocer sus limitantes, es de mayor relevancia conocer lo que ofrece.

4.6.4 Ejemplo

Ya casi para finalizar veamos un pequeño ejemplo que obtuve de una página, que es una operación para calcular el area de un circulo.

```
def calcular_area(radio):
    return math.pi * radio * radio

distancia = 25
print(calcular_area(distancia))
```

Figura 5. Código en python para el ejercicio

Para ello se necesitan 4 identificadores:

- La distancia recorrida
- El valor de pi
- El radio
- El área del circulo

Todos estos cumplen su labor para el procedimiento: la distancia nos indica el valor del radio, pi y el radio se usan para calcular el área, y con calcular área mandamos llamar la función. Esto no almacena el área en ningun lado

Veamos la tabla resultante de este ejercicio.

nombre	tipo	alcance	dirección en memoria	valor	información adicional
distancia	variable	global	0x1000	25	Data type: float
pi	constante	global	0x1004	3.1416	Data type: float solo lectura
calcular area	Función	global	0x1008	N/A	Return: float
radio	parametro	local	0x2000	0x1000	Data type: float

Figura 6. Tabla de símbolos y direcciones

Aquí ocurren los siguientes eventos:
- La tabla indica que el valor de la distancia es 25, coloca el tamaño de 4 bytes, su dirección y su alcance.
- El valor de pi viene de una libreria y es un float fijo, nunca cambia.
- El cálculo del área es una función que regresa el valor que da la operación resultante, que regresa un float, así como que es global.
- El parámetro de radio no es global, dado que solo se usa en la función, es float y su valor es la dirección en memoria de la distancia.

Como se puede apreciar, todos son de tipos diferentes, uno de ellos tiene un alcance diferente, se indican las direcciones de memoria, el valor que poseen y algo de información adicional.

4.6.5 Conclusión

La tabla registra todos los valores relevantes asignados con el identificador, lo cual permite conocer tanto las características del identificador como la ubicación en memoria del mismo.

El uso de estos dos tablas en conjunto es lo que permite el uso adecuado de la información, ya que por separado tendrían numerosas limitaciones.

Fuentes

Victoria G. Emanuel M. Diego M. Maria U. (2024, 16 de agosto) Esquema de traducción & Generación de tabla de símbolos y tabla de direcciones. Instituto Tecnológico Superior de Huichapan.
https://es.slideshare.net/slideshow/1-5-y-1-6-esquema-de-traducción-y-generacion-de-tabla-pptx/271058160#16

Ankit87 (2024, 27 de septiembre). Symbol Table in Compiler. Geeks for Geeks.
https://www.geeksforgeeks.org/symbol-table-compiler/

Tema 4.7: Manejo de errores semánticos.

Índice:

Introducción

1. ¿Qué son los errores semánticos?
   1.1 Ejemplos de errores semánticos
2. Causas comunes de los errores semánticos
   2.1 Confusión en la lógica del programa
   2.2 Asignación incorrecta de tipos de datos
   2.3 Error en la lógica de bucles
3. Detección de errores semánticos
   3.1 Pruebas unitarias
   3.2 Depuración (Debugging)
   3.3 Revisiones de código (code Reviews)
4. Manejo de errores semánticos en un precompilador
   4.1 Recopilar información contextual
   4.2 Tipos de errores semánticos
   4.3 Estructura de manejo de errores
   4.4 Formato claro y detallado del mensaje de error
   4.5 Integración con otras fases del compilador
   4.6 Validación cruzada con la tabla de símbolos

Conclusión

Bibliografía

Tema 4.7: Manejo de errores semánticos.

Introducción:

El manejo de errores semánticos es una parte crucial en el desarrollo de software. A diferencia de los errores sintácticos, que son detectados fácilmente por compiladores o intérpretes, los errores semánticos ocurren cuando el código es gramaticalmente correcto, pero no ejecuta lo que el programador tenía la intención de hacer. Estos errores suelen generar resultados incorrectos o comportamientos inesperados, lo que puede afectar negativamente la funcionalidad del software.

Esta monografía explora en profundidad el concepto de los errores semánticos, sus causas, cómo se pueden detectar, y qué estrategias se pueden aplicar para prevenir y corregir estos errores.

1. ¿Qué son los errores semánticos?

Los errores semánticos son aquellos errores relacionados con el significado del código. Son fallos en la lógica de un programa que no necesariamente hacen que el compilador o intérprete se detenga, pero que provocan que el programa no se comporte como se espera. Un error semántico puede pasar desapercibido durante la compilación o interpretación porque el código es válido en términos de la gramática del lenguaje, pero tiene un comportamiento lógico incorrecto.

1.1 Ejemplos de errores semánticos

a) Asignación incorrecta de variables

Un error semántico común ocurre cuando una variable se asigna incorrectamente. Por ejemplo, si un programador intenta intercambiar los valores de dos variables pero comete un error en la lógica:

```
a = 5
b = 10
a = b
b = a
```

(Lenguaje Python)

Asignación correcta $\Longrightarrow$

```
a = 5
b = 10
temp = a
a = b
b = temp
```

En este caso, después de la ejecución, ambas variables a y b tendrán el valor 10, lo cual no era la intención original. La intención era que a tuviera el valor 10 y b el valor 5.

b) Condiciones lógicas mal formuladas

Las condiciones en estructuras de control, como los bucles o las sentencias condicionales, pueden ser semánticamente incorrectas. Por ejemplo:

```
edad = 19
if (edad > 18):
    print("Es menor de edad") # Error semántico
```

El código es sintácticamente correcto, pero la condición es semánticamente errónea porque, si la edad es mayor a 18, se deberá imprimir "Es mayor de edad", no "Es menor de edad".

c) Acceso incorrecto a datos en estructuras de datos, como listas o arrays, de manera incorrecta:

```
lista = [1, 2, 3]
print (lista[3])  # Error semántico
```

Este código no tiene problemas de sintaxis, pero en tiempo de ejecución fallará porque el índice 3 está fuera de los límites de la lista.

2. Causas comunes de los errores semánticos

Los errores semánticos suelen ser difíciles de identificar porque no generan un mensaje de error directo al programador. Estas son algunas de las causas más comunes:

2.1 Confusión en la lógica del programa.

Una de las causas más comunes de errores semánticos es cuando el programador no ha comprendido completamente la lógica que quiere implementar. Esto puede llevar a que el código haga algo diferente a lo que se pretendía.

Por ejemplo, en un algoritmo que ordena una lista de números, si se confunde el criterio de comparación, el programa podría ordenar incorrectamente los elementos. Este tipo de error es difícil de detectar porque, a nivel sintáctico, el programa se ejecuta sin problemas.

2.2 Asignación incorrecta de tipos de datos

Otra fuente común de errores semánticos es el mal uso de los tipos de datos. Los lenguajes de programación suelen tener diferentes tipos de datos, como enteros, flotantes, cadenas de texto y booleanos. Si se utiliza un tipo de dato incorrecto en un contexto donde se espera otro, los resultados pueden ser inesperados.

```
int resultado = true;  // Error semántico: se espera un número
(Lenguaje Java)
```

Aunque el código es sintácticamente válido, el programador cometió un error lógico al intentar asignar un valor booleano a una variable que debería ser un entero.

2.3 Error en la lógica de bucles.

Los bucles que no terminan correctamente o que no ejecutan la cantidad de iteraciones esperada también pueden generar errores semánticos. Un error común es usar mal una condición de finalización en un bucle, como:

```
for (int i = 0; i <= 10; i --) {  // Error semántico: infinito
        System.out.println(i);
}
```

Este bucle es sintácticamente correcto, pero la lógica es defectuosa, ya que la condición i-- hace que el valor de i disminuya indefenidamente, provocando un bucle infinito.

3. Detección de errores semánticos

Detectar errores semánticos es más difícil que detectar errores sintácticos, ya que los compiladores e intérpretes no suelen marcados como fallos. Aquí se presentan algunas técnicas efectivas para la detección de estos errores.

3.1 Pruebas unitarias

Las pruebas unitarias son una técnica fundamental para detectar errores semánticos. Consisten en crear pequeños test que verifican el correcto funcionamiento de cada unidad o componente del software. Estas pruebas permiten validar que las funciones o métodos individuales hagan lo que se espera de ellos.

Ejemplo en Python con la librería unittest:

```
import unittest
def suma(a,b):          # Definición función suma
    return a+b

class TestSuma(unittest.TestCase):      #Definición del la
                                        #clase prueba
    def test_suma(self):
        self.assertEqual(suma(2,3),5)   #Test pasará si
                                        #suma funciona correc-
                                        #tamente
if __name__ == '__main__':
    unittest.main()
                                        #Punto de entrada
```

Si alguna parte del código no pasa la prueba, es una señal de que hay un error semántico en la lógica implementada.

3.2 Depuración (Debugging):

El uso de herramientas de depuración (debugging) es esencial para la detección de errores semánticos. Un depurador permite ejecutar el programa paso a paso, inspeccionar el valor de las variables en tiempo real, y verificar el flujo de ejecución.

Por ejemplo, en lenguajes como C++, Python o Java, los depuradores permiten "pausar" el programa en puntos específicos y revisar si los valores son los esperados. Esto es útil para identificar errores lógicos ocultos.

3.3 Revisiones de código (Code Reviews)

Las revisiones de código son otro método eficaz para identificar errores semánticos. Involucran que otros programadores revisen el código para verificar que esté implementado correctamente. A menudo, un programador que escribió el código puede detectar errores o malas prácticas que el autor original no vio.

El proceso de revisión también ayuda a mejorar la calidad del código y a garantizar que cumpla con los estándares de la organización.

4.- Manejo de errores semánticos en un precompilador.

El sistema debe informar de manera clara y precisa los problemas encontrados, especificando qué error ocurrió, en qué parte del código, y por qué ese error es un problema.

4.1 Recopilar información contextual

Para proporcionar mensajes de error útiles, es necesario que el precompilador recopile información clave, como:

- Número de línea donde ocurrió el error.
- Contexto del error, es decir, la operación o expresión que causó el problema.
- Nombre del archivo (si se trabaja con múltiples archivos).

Esta información se puede obtener desde el análisis léxico o análisis sintáctico, ya que estas fases normalmente asignan números de línea y posición a cada token o nodo del árbol de sintaxis abstracta (AST).

4.2 Tipos de errores semánticos

Los errores semánticos pueden abarcar diferentes categorías, y para cada tipo de error necesitas un mensaje de error específico. Algunos ejemplos son:

- Uso de variables no declaradas.
- Incompatibilidad de tipos en operaciones.
- Uso de variables fuera de su ámbito (scope).
- Return de tipo incorrecto en funciones.

Para cada uno de estos errores, define mensajes que expliquen de manera sencilla el problema. Aquí algunos ejemplos de cómo deberían verse los mensajes:

- Error semántico en línea 25, archivo main.c: La variable 'x' no ha sido declarada antes de su uso.
- Error semántico en línea 45, archivo funciones.c: Tipos incompatibles en la suma: int + string.

4.3 Estructura de manejo de errores

La detección de errores semánticos puede implementarse como una estructura centralizada que gestione los errores y los recoja a medida que se recorren las expresiones o declaraciones en el AST.

Ejemplo de un diseño básico de gestión de errores en el lenguaje Python:

Clase o Módulo de Errores

Definir una clase o módulo encargado de almacenar y reportar los errores

```
class ErrorManager:
    def __init__(self):
    # Inicializa una lista vacía para almacenar mensajes de error
    self.errors = []

    def report_error(self, line, filename, message):
    # Formatea el mensaje de error con el número de línea, nombre
# del archivo y mensaje proporcionados
    error message = f"Error en línea {line}, archivo {filename}:
                    {message}"
```

```
#añade el mensaje de error formateado a la lista de errores
self.errors.append(error_message)
#imprime el mensaje de error en la consola
print(error_message)

def has_errors(self):
    #Devuelve True si hay algún error en la lista, de lo contrario
#devuelve False
    return len(self.errors) > 0
```

4.4 Formato claro y detallado del mensaje de error.

El mensaje de error debe contener al menos estos elementos:

- Tipo de error: Indicar que es un error semántico para diferenciarlo de los errores sintácticos o léxicos.
- Ubicación del error: Proporcionar la línea, la columna (opcional) y el nombre del archivo donde ocurrió el error.
- Descripción del error: Explicar qué causó el error, y en casos complejos, proporcionar una pequeña sugerencia de corrección.

Ejemplo completo:

Error semántico en línea 15, archivo operaciones.c:
La variable 'total' no ha sido declarada antes de su uso.
Asegúrate de declarar la variable en el ámbito adecuado.

4.5 Integración con otras fases del compilador.

Aunque el precompilador suele detenerse en la fase de análisis semántico cuando se encuentran errores graves, se puede decidir permitir que el compilador siga funcionando para mostrar todos los errores posibles antes de detenerse. Esto permite que los desarrolladores corrijan varios errores en una sola pasada.

4.6 Validación cruzada con la tabla de símbolos.

El precompilador debería realizar una verificación final cruzada con la tabla de símbolos para asegurarse de que todos los identificadores usados en el código tienen un tipo y una declaración válida. Esto reduce la probabilidad de que errores pasen desapercibidos.

Conclusión:

El manejo de errores semánticos en un precompilador consiste en crear un sistema que detecte y notifique claramente los errores encontrados durante el análisis del código fuente.
A diferencia de los errores sintácticos, que suelen ser identificados por compiladores o intérpretes, los errores semánticos no son fáciles de detectar y pueden generar fallos lógicos difíciles de identificar. Estos errores ocurren cuando el código, aunque sintácticamente correcto, no realiza la acción que el programador tenía en mente, afectando el comportamiento de las aplicaciones.

Bibliografía:

- 1.5 Errores semánticos (s.f.). https://cidecame.uaeh.edu.mx/lcc/mapa/PROYECTO/libro32/45--errores--semanticos.html.

- Jammy. (s.f.). ¿Qué es el error de sintaxis y el error semántico? https://adcad.com/es/que-es-el-error-de-sintaxis-y-el-error-semantico/.

- Galvão, P. (2023, 6 noviembre). Sintaxis en el lenguaje Java y Semántica: aprende qué es: PG Tech; https://tech.pedrogalvao.com/es/dev/aprender-java/sintaxis-y-semantica-en-java/

- Sheth, D. (2023, 15 octubre). Handling Errors like a Pro: Practical Strategies for Python Error Management. Medium. https://medium.com/@darshilsheth10/handling-errors-like-a-pro-practical-strategies-for-python-error-management-8adffc17e178.

Videos

Video 1

https://www.youtube.com/watch?v=OnOOyHAro6g

Video 2

https://www.youtube.com/watch?v=_7LPcRcisio