UNIVERSIDAD AUTÓNOMA DE CHIAPAS

FACULTAD DE CONTADURÍA Y ADMINISTRACIÓN

LIDTS

6 M

COMPILADORES

SUBCOMPETENCIA 3 - ANÁLISIS SEMÁNTICO

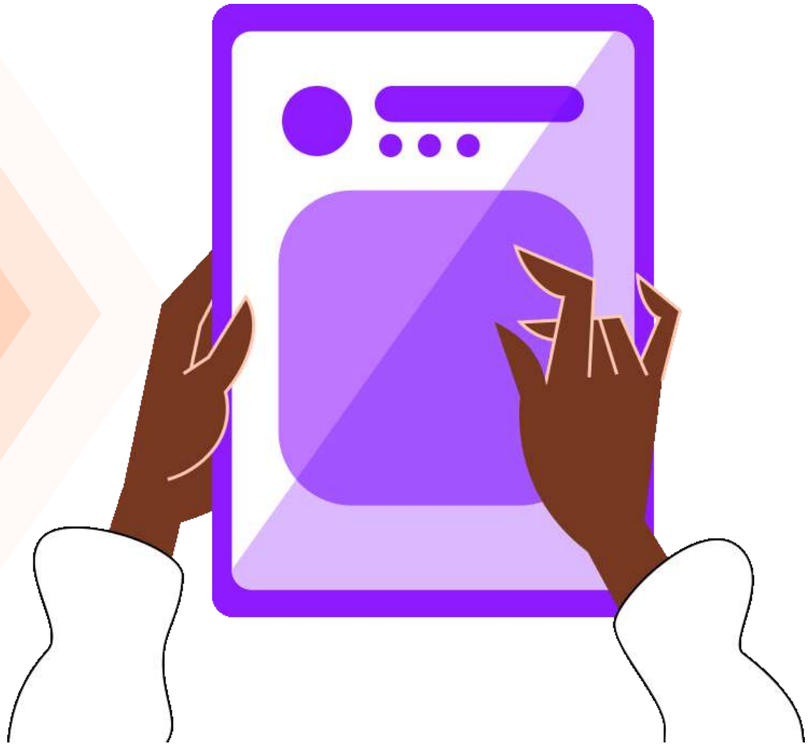
ACT. 3.3 INVESTIGAR PILAS SEMANTICA, ESQUEMA DE TRADUCCIÓN

ALUMNO: MARCO ANTONIO ZÚÑIGA MORALES

DOCENTE: DR. LUIS GUTIÉRREZ ALFARO

TUXTLA GTZ, CHIAPAS

NOVIEMBRE, 2023



PILAS SEMANTICA, ESQUEMA DE TRADUCCIÓN, GENERACIÓN DE LA TABLA DE SÍMBOLOS Y DE DIRECCIONES

INTRODUCCIÓN

En el complejo mundo de la construcción de compiladores e intérpretes, la eficiencia y precisión son imperativos. Tres componentes fundamentales juegan un papel crucial en este proceso: la pila semántica en un analizador sintáctico, el esquema de traducción y la generación de la tabla de símbolos y direcciones. Estos elementos, interconectados de manera intrincada, desempeñan roles esenciales para garantizar la correcta interpretación y ejecución del código fuente. Exploraremos cómo la pila semántica facilita la gestión de información clave, cómo el esquema de traducción establece las reglas para la transformación del código, y cómo la generación de la tabla de símbolos y direcciones asegura la coherencia y eficiencia en el proceso, delineando así los cimientos de un desarrollo de software robusto y eficaz.

PILA SEMÁNTICA EN UN ANALIZADOR SINTÁCTICO

¿Qué es?

La pila semántica es una estructura de datos utilizada en análisis sintáctico y semántico de lenguajes de programación. Se utiliza para rastrear información semántica a medida que se analiza un programa o una expresión, lo que ayuda a construir un árbol de análisis sintáctico y, eventualmente, a evaluar o generar código intermedio.

La pila semántica suele ser una pila (stack) que almacena información relacionada con las expresiones y operaciones que se están analizando. A medida que se procesa el código fuente, la información semántica se empuja (push) y pop (sacar) de la pila, y se realizan operaciones semánticas en función de lo que se encuentra en la parte superior de la pila.

PILA SEMÁNTICA EN UN ANALIZADOR SINTÁCTICO

Características

1. **Almacenamiento de información semántica:** La pila semántica almacena información relacionada con las expresiones y operaciones que se están analizando, como tipos de datos, valores temporales y estructuras de control semántico.
2. **Mantenimiento de contexto:** Ayuda a mantener el contexto y el ámbito de las variables, lo que es esencial para el análisis semántico y la resolución de conflictos de nombres.
3. **Coherencia semántica:** Garantiza que las operaciones se realicen en el orden correcto y que las restricciones semánticas del lenguaje se cumplan, lo que contribuye a la generación de un árbol de análisis sintáctico y a la detección de errores semánticos.

PILA SEMÁNTICA EN UN ANALIZADOR SINTÁCTICO

Ventajas

1. **Facilita la generación de código intermedio:** permite organizar la información semántica y las operaciones de manera eficiente.
2. **Facilita la comprobación de tipos:** Permite realizar comprobaciones de tipos y garantizar que las operaciones sean coherentes con los tipos de datos involucrados.
3. **Ayuda en la detección temprana de errores semánticos:**

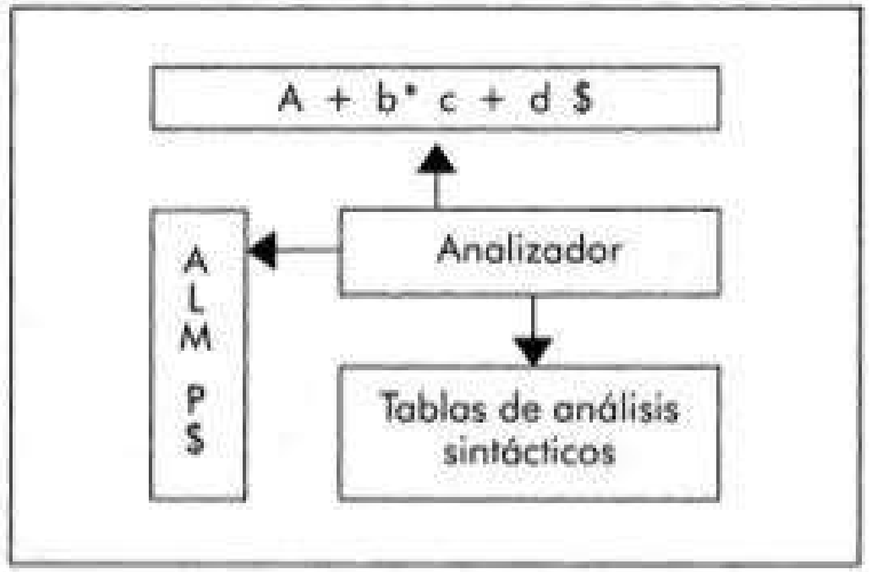
Desventajas

1. **Requiere consumo de memoria:** El uso de una pila semántica puede requerir memoria adicional para almacenar la información semántica, lo que puede ser un problema en sistemas con recursos limitados.
2. **Mayor complejidad:** La implementación de una pila semántica agrega complejidad al analizador sintáctico.
3. **Posible sobrecarga en el rendimiento**

PILA SEMÁNTICA EN UN ANALIZADOR SINTÁCTICO

Ejemplos de lo que podría almacenarse en la pila semántica

1. Valores temporales o resultados parciales de expresiones.
2. Tipos de datos de variables y expresiones.
3. Información sobre operaciones pendientes, como asignaciones, llamadas a funciones, etc.
4. Información de control de flujo, como la gestión de ámbitos (scope) de variables.



PILA SEMÁNTICA EN UN ANALIZADOR SINTÁCTICO

Ejemplo

```
resultado = (3 + 5) * 2
```

1. **Inicialización de la pila semántica:**
   La pila semántica comienza vacía.

2. Análisis sintáctico y proceso de la expresión:
Encuentra el número 3 y lo empuja a la pila.
Encuentra el operador de suma (+) y lo empuja a la pila.
Encuentra el número 5 y lo empuja a la pila.
Encuentra el operador de cierre de paréntesis (no lo coloca en la pila, pero indica que se debe realizar una operación cuando se encuentra un paréntesis de cierre).
Encuentra el operador de multiplicación (*) y lo empuja a la pila.
Encuentra el número 2 y lo empuja a la pila.

PILA SEMÁNTICA EN UN ANALIZADOR SINTÁCTICO

Ejemplo

```
resultado = (3 + 5) * 2
```

3. Operaciones en la pila semántica:
Cuando se encuentra el paréntesis de cierre, se realiza la operación de suma en la pila (3 + 5) y se coloca el resultado (8) en la pila.
Luego, se realiza la operación de multiplicación en la pila (8 * 2) y se coloca el resultado (16) en la pila.

Asignación del resultado:
Finalmente, el resultado (16) se asigna a la variable "resultado".

ESQUEMA DE TRADUCCIÓN

¿Qué es?

Un esquema de traducción, en el contexto de la construcción de compiladores o intérpretes, se refiere a un conjunto de reglas que definen cómo se traducen las estructuras sintácticas de un lenguaje fuente a un lenguaje destino. Estas reglas se utilizan para guiar el proceso de generación de código o de ejecución de un programa.

Un esquema de traducción define las reglas y las acciones que ocurren en cada etapa del proceso de traducción. Puede incluir reglas para traducir declaraciones, expresiones, operadores, estructuras de control, llamadas a funciones y otros elementos del lenguaje fuente.

ESQUEMA DE TRADUCCIÓN

1. Análisis Sintáctico: En esta etapa, el analizador sintáctico divide el código fuente en unidades léxicas (tokens) y construye una estructura de árbol de análisis sintáctico (AST) que representa la estructura del programa.

2. Análisis Semántico: En esta etapa, el analizador semántico verifica la coherencia semántica del programa. Esto puede incluir la comprobación de tipos, la resolución de nombres, la detección de errores semánticos y la construcción de una tabla de símbolos.

3. Generación de Código Intermedio: Si el objetivo es generar código intermedio, esta etapa implica la traducción del AST en una representación intermedia independiente de la plataforma. Esta representación puede ser en forma de código de tres direcciones, árbol de expresiones, código de máquina virtual, etc.

ESQUEMA DE TRADUCCIÓN

4. Optimización de Código Intermedio: Opcionalmente, se puede aplicar una serie de optimizaciones en el código intermedio para mejorar la eficiencia y reducir la redundancia.

5. Generación de Código Destino: Si el objetivo es generar código ejecutable, esta etapa implica la traducción del código intermedio en el lenguaje de la máquina objetivo. Aquí se pueden realizar optimizaciones específicas de la plataforma.

6. Ensamblaje o ejecución: En el caso de la generación de código ejecutable, el código destino se ensambla en un programa ejecutable. En el caso de un intérprete, se ejecuta directamente.

ESQUEMA DE TRADUCCIÓN

Características

- **Estructura de reglas:** Los esquemas de traducción consisten en un conjunto de reglas que definen cómo se traducen las estructuras sintácticas del lenguaje fuente a un lenguaje destino.

- **Fases del proceso:** Los esquemas de traducción son típicamente divididos en fases, como el análisis léxico, análisis sintáctico, análisis semántico, generación de código intermedio, optimización y generación de código destino.

- **Independencia de plataforma:** Los esquemas de traducción permiten que el mismo código fuente sea traducido a diferentes plataformas de destino (por ejemplo, sistemas operativos o arquitecturas de CPU) mediante la generación de código destino específico para cada plataforma.

ESQUEMA DE TRADUCCIÓN

Ventajas

1. **Portabilidad:** Puedes ejecutar el mismo código en diferentes plataformas sin necesidad de modificarlo.
2. **Optimización:** Las fases de optimización mejoran la eficiencia del programa resultant e.
3. **Abstracción de detalles:** Los programadores se centran en la lógica del programa, no en detalles de la plataforma.

Desventajas

1. **Complejidad:** La construcción de un compilador o intérprete es un proceso complejo y que consume tiempo.
2. **Rendimiento:** La generación de código intermedio puede introducir sobrecarga de rendimiento.
3. **Limitaciones del lenguaje fuente:** El esquema de traducción puede estar limitado por las características del lenguaje fuente y no ser eficiente para ciertos algoritmos o estructuras de datos.

ESQUEMA DE TRADUCCIÓN

Ejemplo

Regla de Traducción: EXPRESION -> ID = EXPRESION

Esta regla indica que cuando encontramos una expresión de la forma ID = EXPRESION, debemos generar código para asignar el valor de la expresión a la variable representada por ID.

Ejemplo de código fuente

```
resultado = (3 + 5) * 2
```

ESQUEMA DE TRADUCCIÓN

```
resultado = (3 + 5) * 2
```

Aplicación de la Regla de Traducción:

1. Identificamos la expresión resultado = (3 + 5) * 2.
2. Aplicamos la regla de traducción: EXPRESION -> ID = EXPRESION.
3. Generamos código para realizar la operación y asignar el resultado:

Código Intermedio Generado

```
LOAD 3          ; Cargar el valor 3 en un registro
LOAD 5          ; Cargar el valor 5 en un registro
ADD             ; Sumar los valores en los registros
LOAD 2          ; Cargar el valor 2 en un registro
MUL             ; Multiplicar el resultado de la suma por 2
STORE resultado ; Almacenar el resultado en la variable "resultado"
```

GENERACIÓN DE LA TABLA DE SÍMBOLO

¿Qué es?

La tabla de símbolos es una estructura de datos utilizada para almacenar información sobre los identificadores (nombres de variables, funciones, constantes, etc.) presentes en el código fuente de un programa. Cada entrada en la tabla de símbolos suele contener información como el nombre del símbolo, su tipo, su ubicación en la memoria, su ámbito o alcance (scope), entre otros detalles relacionados.

Proceso

Durante el análisis sintáctico y semántico, el compilador o intérprete recopila información sobre los identificadores y sus atributos y la almacena en la tabla de símbolos. Se actualiza y consulta a medida que se procesa el código fuente para garantizar que los identificadores se declaren y utilicen correctamente.

Uso

La tabla de símbolos es esencial para la comprobación de tipos, la resolución de nombres, la detección de errores semánticos y la generación de código. Garantiza que se respeten las reglas del lenguaje y que los identificadores se utilicen de manera coherente.

GENERACIÓN DE LA TABLA DE SÍMBOLO

Características

1. **Registro de Identificadores**: Guarda nombres como variables o funciones.
2. **Almacenamiento de Información:** Puede almacenar tipos, ubicaciones de memoria, etc.
3. **Uso en Análisis Semántico:** Es crucial para verificar la coherencia semántica en el código.

Ventajas

- **Detección de Errores:** Facilita encontrar errores semánticos.
- **Manejo de Ámbitos:** Ayuda a gestionar ámbitos y resolver conflictos de nombres.
- **Optimización y Análisis:** Permite realizar optimizaciones avanzadas.

Desventajas

- **Consumo de Recursos:** Puede usar muchos recursos, especialmente en programas grandes.
- **Complejidad de Implementación:** Su implementación puede ser compleja.
- **Impacto en el Rendimiento:** Un acceso ineficiente puede afectar el rendimiento.

GENERACIÓN DE LA TABLA DE SÍMBOLO

Ejemplo

```
int a = 5;
float b = 3.14;
```

Durante el análisis sintáctico, se generarían las entradas correspondientes en la tabla de símbolos. Cada entrada podría contener información como el nombre del símbolo, su tipo de dato y posiblemente su dirección de memoria. Aquí está cómo podría ser la tabla de símbolos después de analizar este fragmento:

Tabla de Símbolos Generada

Nombre	Tipo	Dirección
a	int	Dirección_1
b	float	Dirección_2

En este ejemplo:

- a es una variable de tipo int con una dirección de memoria (hipotética) llamada Dirección_1.
- b es una variable de tipo float con una dirección de memoria (hipotética) llamada Dirección_2.

GENERACIÓN DE DIRECCIONES

¿Qué es?

La generación de direcciones se refiere al proceso de asignar direcciones de memoria o ubicaciones a las variables y estructuras de datos utilizadas en un programa. Es una parte fundamental del proceso de generación de código.

Proceso

Durante la generación de código, el compilador o intérprete asigna ubicaciones de memoria a las variables y estructuras de datos. Esto implica la asignación de direcciones de memoria o registros a cada variable o dato utilizado en el programa.

Uso

La generación de direcciones es crucial para la traducción del código fuente a un código ejecutable o código intermedio. Garantiza que las instrucciones del programa accedan a las ubicaciones de memoria adecuadas para leer y escribir datos.

GENERACIÓN DE DIRECCIONES

Características

1. **Asignación de Memoria:** Da lugar a la ubicación de datos en la memoria.
2. **Manejo Eficiente:** Gestiona cómo se utiliza y reserva la memoria.
3. **Considera Tipos de Datos:** Asegura que la asignación sea adecuada según el tipo de dato.

Ventajas

- **Eficiencia de Memoria:** Asigna espacio de manera eficiente.
- **Acceso Rápido:** Permite un acceso rápido y ordenado a los datos.
- **Optimización de Almacenamiento:** Mejora el rendimiento mediante una mejor gestión de la memoria.

Desventajas

- **Complejidad:** Puede volverse complejo en programas grandes.
- **Fragmentación Posible:** Podría causar fragmentación de la memoria si no se gestiona adecuadamente.
- **Dependencia de Arquitectura:** Debe adaptarse a la arquitectura específica de la máquina.

GENERACIÓN DE DIRECCIONES

Ejemplo

```
int a;
float b;
char c;
```

Durante la generación de direcciones, asignaríamos direcciones de memoria a cada variable declarada. Aquí está un ejemplo sencillo de cómo podríamos asignar direcciones a las variables:

Variable	Tipo	Dirección
a	int	1000
b	float	1004
c	char	1008

En este ejemplo:

- La variable **a** de tipo **int** se asigna a la dirección de memoria **1000**.
- La variable **b** de tipo **float** se asigna a la siguiente dirección de memoria disponible, **1004**.
- La variable **c** de tipo **char** se asigna a la siguiente dirección de memoria disponible, **1008**.

CONCLUSIÓN

En conclusión, la pila semántica en un analizador sintáctico, el esquema de traducción y la generación de la tabla de símbolos y direcciones son elementos esenciales en el desarrollo de compiladores e intérpretes. La pila semántica facilita la gestión eficiente de información semántica durante el análisis sintáctico, el esquema de traducción establece reglas clave para la transformación del código fuente, y la generación de la tabla de símbolos y direcciones garantiza la coherencia y la eficiencia en la ejecución del programa. Juntos, estos componentes forman un marco sólido que aborda aspectos cruciales del procesamiento del código, desde la interpretación de su significado hasta la asignación eficiente de recursos durante la ejecución.

REFERENCIA BIBLIOGRÁFICA

Arroyo, M., & Aguirre, J. (1998). IBURG: su aplicación para la generación de generadores de código en un proyecto de compiladores. In IV Congreso Argentina de Ciencias de la Computación

Zamora, F. R., & Varela, J. C. Generación de Código Intermedio.

Díaz, M. L. G. Introducción a la construcción de compiladores. Departamento de Informática de la Universidad de Valladolid.

Michael Sipser. Introduction to the Theory of Computation".

Andrew W. Appel. "Modern Compiler Implementation in Java".

Aho, A. V., Sethi, R., & Ullman, J. D. (2006). Compiladores: Principios,técnicas y herramientas. Editorial: Addison-Wesley

REFERENCIA BIBLIOGRÁFICA

Rodríguez, J. A. (2015). Pilas semaánticas en compiladores. Universidad de Almeria.

García, J. M., & González, J. M. (2019). Fundamentos de la compilación. Editorial: Pearson.

López, J. M., & González, J. M. (2022). Análisis y generación de código. Editorial: McGraw-Hill

MUCHAS GRACIAS

POR VER ESTA PRESENTACIÓN