Proceso de compilación

- **Lenguaje Fuente:** Es el lenguaje de programación en el que originalmente está escrito el código. Este es el lenguaje que los desarrolladores utilizan para crear programas. Ejemplos comunes de lenguajes fuente incluyen C, Java, Python, entre otros.
- **Lenguaje Objeto:** Es el lenguaje al que se traduce el código fuente después de pasar por el proceso de compilación. Generalmente, el lenguaje objeto es un código de bajo nivel, como el código máquina o el lenguaje ensamblador, que puede ser directamente ejecutado por el hardware del ordenador.

Un compilador lee un programa escrito en un ***lenguaje fuente*** y lo traduce a un programa equivalente en un ***lenguaje objeto.***

El proceso de compilación está formado por dos partes:

1. el **ANÁLISIS**, que, a partir del programa fuente, crea una representación intermedia del mismo; y
2. la **SÍNTESIS**, que, a partir de esta representación intermedia, construye el programa objeto.

El **ANÁLISIS** del programa fuente

En la compilación, el análisis está formado por tres fases:

a) el Análisis Léxico.
b) el Análisis Sintáctico.
c) el Análisis Semántico.

Análisis Léxico (Scanner)

El análisis léxico es la primera etapa del proceso de compilación. Se encarga de leer el código fuente, que es solo texto, y lo divide en pequeñas partes llamadas **tokens**.

¿Qué son los Tokens?

Los tokens son piezas básicas del lenguaje que el compilador entiende. Podemos pensarlos como palabras de un idioma: cada palabra tiene un significado específico. En programación, ejemplos de tokens son palabras clave como if, while, nombres de variables, números, operadores (+, -, =), etc.

Los espacios y saltos de línea se obvian porque no son necesarios para entender el código.

Proceso del Análisis Léxico

El análisis léxico detecta lexemas que componen un programa y traduce esta secuencia en una secuencia de tokens. El flujo de caracteres se convierte en una secuencia de lexemas, que a su vez se traduce en una secuencia de tokens.

Flujo de Caracteres → Secuencia de Lexemas → Secuencia de Tokens

Ejemplo

```
int WHILE;
while (WHILE > 32)
  WHILE = WHILE - 2.46;
```

El análisis léxico identificará int, WHILE, ;, while, (, WHILE, >, 32,), etc., como tokens.

Aunque WHILE se repite, el análisis léxico lo considera el mismo token (un nombre de variable), pero ubicado en diferentes partes del código.

Representación de Tokens

Cada token se representa mediante un número entero o un valor de un tipo enumerado, y así será reconocido por el resto del compilador. Si bien la distinción entre operadores de comparación no afecta el análisis sintáctico, el scanner deberá retornar el par ordenado (operadorComparación, cuálEs) para que en el análisis semántico y en la etapa de síntesis se conozca la información exacta.

Representación de Lexemas y Tokens

- **Palabras Reservadas:** Se representan mediante un par ordenado (palabraReservada, índice) donde índice indica su posición en la Tabla de Símbolos (TS).
- **Identificadores:** Se representan mediante un par ordenado (identificador, índice) donde índice es su posición en la TS.
- **Constantes Literales:** En el caso de literales-cadena y constantes numéricas, se presentan dos soluciones:
    - ***Guardado en la TS:*** El scanner guarda la secuencia de caracteres que forma cada constante numérica en la TS, y retorna (tipoConstanteNumérica, índice).
    - ***Conversión Directa:*** El scanner convierte la secuencia de caracteres que forma la constante al valor numérico correspondiente y retorna (tipoConstanteNumérica, valor).

Resumen

- **Entrada:** Tu código fuente, como un conjunto de caracteres.
- **Salida:** Una lista de tokens, que son las piezas básicas que el compilador usará en los siguientes pasos.

Análisis Sintáctico (Parser)

El análisis sintáctico es la segunda etapa del proceso de compilación. Su trabajo es asegurarse de que los tokens, que fueron identificados en el análisis léxico, estén organizados de manera correcta de acuerdo con las reglas de la gramatica sintactica del lenguaje de programación, ósea, se asegura de que tu código esté correctamente estructurado antes de seguir a las fases más profundas del análisis, como la semántica.

¿Qué hace?

Imagina que tienes un conjunto de piezas de Lego (los tokens). El análisis sintáctico verifica que las piezas estén conectadas en el orden correcto para construir algo coherente, como una estructura de Lego. Por ejemplo, si escribes if (a > b), el análisis sintáctico se asegura de que esta estructura sea válida, es decir, que if esté seguido de paréntesis, y dentro de ellos, una condición.

¿Qué no puede hacer?

El análisis sintáctico no puede decirte si las variables que usas están declaradas o si haces operaciones lógicas con los tipos correctos. Esto se revisa en la siguiente etapa, que es el análisis semántico.

Por ejemplo, si tienes dos declaraciones de la misma variable en diferentes lugares del código, el análisis sintáctico no lo detectará como un error. Solo verificará si la estructura del código sigue las reglas de la gramática del lenguaje.

Formas de Análisis Sintáctico

Por el Orden en que se Derivan los No Terminales:

- ***Derivación por Izquierda:*** Se reemplaza el primer no terminal que se encuentra en una cadena de derivación leída de izquierda a derecha. (BISON Y PROFE)
- ***Derivación por Derecha:*** Se reemplaza el último no terminal de la cadena de derivación leída de izquierda a derecha.

Por el Enfoque de Análisis

Existen dos enfoques principales para realizar el análisis sintáctico:

- ***Análisis Sintáctico Descendente:*** Este método permite que un programador construya manualmente el análisis. Utiliza un enfoque de arriba hacia abajo, donde se comienza por el axioma de la gramática y se desciende a las producciones.
    - ***Análisis Sintáctico Descendente Recursivo (ASDR):*** Es una forma específica de análisis sintáctico descendente que utiliza rutinas recursivas para construir un árbol de análisis sintáctico (AAS). Este árbol parte del axioma de la gramática independiente del contexto (GIC) y representa la derivación de una construcción, que puede ser una declaración, sentencia, expresión, bloque, o incluso el programa completo.
- ***Análisis Sintáctico Ascendente:*** También conocido como "bottom-up", este enfoque requiere el uso de programas especializados, como yacc, para analizar el código de abajo hacia arriba, comenzando con los tokens y ascendiendo hacia el axioma.

En el ASDR cada noterminal tiene asociado un PAS

Cada PAS sigue el desarrollo del lado derecho de la producción que implementa

1) Si se debe procesar un noterminal <A>, se invoca al PAS correspondiente.
2) Para procesar un terminal t, se invoca al procedimiento Match con argumento t.

Ejemplo formas de análisis sintactico

Consideremos la gramática con producciones:

S -> aST
S -> b
T -> cT
T -> d

Y analicemos si la construcción aabcdd es sintácticamente correcta.

1. Derivación por Izquierda – Análisis Sintáctico Descendente

Se reemplaza el primer no terminal que se encuentra en una cadena de derivación leída de izquierda a derecha.

Cadena de Derivación Obtenida | Próxima Producción a Aplicar

S (axioma) | S -> aST (1)
aST | S -> aST (1)
aaSTT | S -> b (2)
aabTT | T -> cT (3)
aabcTT | T -> d (4)
aabcdT | T -> d (4)
aabcdd (correcta)

2. Derivación por Derecha

Se reemplaza el último no terminal de la cadena de derivación leída de izquierda a derecha.

Cadena de Derivación Obtenida | Próxima Producción a Aplicar

S (axioma) | S -> aST (1)
aST | T -> d (4)
aSd | S -> aST (1)
aaSTd | T -> cT (3)
aaScTd | T -> d (4)
aaScdd | S -> b (2)
aabcdd

3. Análisis Sintáctico Ascendente

Derivación a derecha, pero en orden inverso.

Cadena de Derivación | Próxima Producción a Aplicar

aabcdd | S -> b (2)
aaScdd | T -> d (4)
aaScTd | T -> cT (3)
aaSTd | S -> aST (1)
aSd | T -> d (4)
aST | S -> aST (1)
S (correcta)

Árbol de Análisis Sintáctico (AAS)

El árbol de análisis sintáctico es una estructura que se genera durante el análisis sintáctico y tiene las siguientes características:

- **Raíz:** Está etiquetada con el axioma de la GIC.
- **Hojas:** Cada hoja está etiquetada con un token. Si lees las hojas de izquierda a derecha, representan la construcción derivada.
- **Nodos interiores:** Están etiquetados con no terminales, que son las producciones de la gramática.

Ejemplo

```
if (x > y) {
   z = x + y;
}
```

Tokens: if, (, x, >, y,), {, z, =, x, +, y, ;, }

Análisis Sintáctico: Aquí, se verifica que este código siga la estructura correcta de un condicional if. El análisis sintáctico se asegura de que los paréntesis estén bien colocados, que haya una condición válida dentro de los paréntesis, y que las llaves {} encierren un bloque de código.

Resumen

- **Entrada:** Los tokens generados por el análisis léxico.
- **Función principal:** Verificar que los tokens estén organizados de manera correcta según las reglas del lenguaje.
- **Resultado:** Si la estructura es válida, el proceso sigue; si hay un error, el compilador te lo indicará.

Análisis Semántico

El análisis semántico es la etapa en la que se verifica que el código fuente no solo esté bien estructurado (como se revisó en el análisis sintáctico), sino que también tenga sentido lógico y cumpla con las reglas semánticas del lenguaje de programación.

¿Qué hace el Análisis Semántico?

- **Verificación de tipos:** Comprueba que las operaciones en el código se realicen entre tipos de datos compatibles. Por ejemplo, no puedes sumar un número con una cadena de texto.
- **Declaración de variables:** Verifica que todas las variables utilizadas estén correctamente declaradas antes de ser usadas.
- **Chequeo de consistencia:** Asegura que las funciones se llamen con el número correcto de argumentos y que las variables no se redefinan en un mismo ámbito.

Conceptos Clave

- **Semántica Estática:** Reglas que deben cumplirse antes de la ejecución, como la declaración de variables y la compatibilidad de tipos. Estas reglas son sensibles al contexto, es decir, dependen del entorno en el que se encuentran las variables y funciones.
- **Semántica en Tiempo de Ejecución:** Reglas que se verifican mientras el programa se ejecuta, aunque generalmente es el entorno de ejecución quien maneja esto.

Rol de las GICs y las Limitaciones

- **GICs (Gramáticas Independientes del Contexto):** Son herramientas que se utilizan principalmente en el análisis sintáctico para definir la estructura básica de un lenguaje de programación. Las GICs pueden describir reglas como la correcta formación de expresiones, sentencias, y otras estructuras sintácticas del lenguaje.
- **Limitaciones de las GICs:** Aunque las GICs son útiles para describir la estructura general del lenguaje (como la correcta colocación de operadores y operandos), no pueden capturar todas las reglas necesarias para asegurar que un programa sea semánticamente correcto. Por ejemplo, las GICs no pueden verificar si una variable ha sido declarada antes de ser utilizada o si los tipos de datos son compatibles en una operación. Estas verificaciones dependen del contexto, y es aquí donde entra en juego el análisis semántico.

Ejemplo de Errores Detectados

```
void XX (void) {
  int a;
  double a;
  if (a > a) return 12;
}
```

- ***Error 1:*** La variable a se declara dos veces, una como int y otra como double. El análisis semántico detecta este error porque no se permite la redefinición de una variable en el mismo ámbito. Este tipo de error no puede ser detectado solo por las GICs, porque depende del contexto en el que las declaraciones ocurren.
- ***Error 2:*** Comparar a > a puede ser sintácticamente correcto (verificado por la GIC), pero el análisis semántico se asegurará de que a sea de un tipo compatible para la comparación.

Funciones del Análisis Semántico

- ***Chequeo de Semántica Estática: Verifica*** que todas las reglas del lenguaje que no dependen del contexto (por ejemplo, la compatibilidad de tipos) se cumplan antes de que el programa se ejecute. Estas reglas no pueden ser expresadas completamente por GICs.
- ***Traducción a Código: Si*** el código es semánticamente correcto, el análisis semántico también genera el código que será ejecutado por la máquina virtual o procesador.

Resumen

- **Entrada:** La estructura del código verificada por el análisis sintáctico.
- **Función principal:** Asegurarse de que el código sea lógico y coherente, que las operaciones sean válidas y que las variables estén correctamente declaradas y utilizadas.
- **Limitaciones de las GICs:** Aunque las GICs ayudan a definir la estructura sintáctica del lenguaje, no pueden verificar todas las reglas semánticas, especialmente aquellas que dependen del contexto, como la declaración de variables y la compatibilidad de tipos.
- **Resultado:** Si el código pasa la verificación semántica, se traduce a instrucciones que una máquina virtual o un procesador pueden ejecutar.

Tabla de Símbolos

La Tabla de Símbolos (TS) es una estructura de datos crucial en la compilación de programas. Sirve para almacenar información sobre los identificadores y otros elementos del código fuente, y facilita el proceso de análisis y generación de código. A continuación, se explica cómo funciona y su propósito.

¿Qué es la Tabla de Símbolos?

La Tabla de Símbolos es una base de datos utilizada por el compilador para almacenar información sobre todos los identificadores que aparecen en el programa fuente. Los identificadores pueden ser variables, funciones, tipos de datos, etc. La tabla ayuda a gestionar estos elementos durante el proceso de compilación, proporcionando información esencial para la generación de código.

Información Almacenada en la Tabla de Símbolos

Cada entrada en la Tabla de Símbolos representa un identificador y contiene atributos que describen sus características. Los atributos comunes incluyen:

- **Tipo:** Indica el tipo de dato del identificador (por ejemplo, entero, flotante, carácter).
- **Ámbito:** Define el alcance o la visibilidad del identificador (local, global, etc.).
- **Parámetros:** En el caso de funciones, la cantidad y tipo de parámetros que acepta.
- **Tipo de Retorno:** Para funciones, el tipo de valor que retorna.
- **Tamaño:** Para variables y arrays, el tamaño o la cantidad de memoria que ocupa.

Ejemplo de Tabla de Símbolos

Consideremos la siguiente función en ANSI C:

```
int XX(double a) {
   char s[12];
   double b;
   b = a + 1.2;
   if (b > 4.61) return 1;
   else return 0;
}
```

Para esta función, la Tabla de Símbolos podría tener las siguientes entradas:

XX:

- ***Tipo:*** Función
- ***Tipo de Retorno:*** int
- ***Número de Parámetros:*** 1
- ***Tipo del Parámetro:*** double

a:

- ***Tipo:*** double
- ***Ámbito:*** Local a la función XX

s:

- ***Tipo:*** char[12]
- ***Ámbito:*** Local a la función XX

b:

- ***Tipo:*** double
- ***Ámbito:*** Local a la función XX

Cómo se Utiliza la Tabla de Símbolos

- **Análisis Léxico:** Durante esta etapa, el compilador identifica los tokens y los almacena en la tabla de símbolos con los atributos adecuados.
- **Análisis Sintáctico:** El compilador utiliza la tabla de símbolos para verificar que los identificadores estén correctamente utilizados según las reglas del lenguaje.
- **Análisis Semántico:** Aquí se realizan comprobaciones adicionales, como verificar el tipo de datos y el alcance de los identificadores, utilizando la información de la tabla.
- **Generación de Código:** La tabla de símbolos ayuda a generar el código de máquina o el código intermedio, proporcionando detalles sobre los identificadores y sus atributos.

La Tabla de Símbolos es fundamental para garantizar que el código fuente sea válido y para facilitar la traducción del código en una forma que la máquina pueda entender y ejecutar

Lenguaje de programación C

ANSI C Es un estándar creado por el Instituto Nacional Estadounidense de Estándares (ANSI) para el lenguaje de programación C. Su objetivo es facilitar que el código sea portable, es decir, que funcione en diferentes plataformas.

ANSI C e ISO C El primer estándar para C fue publicado por ANSI en 1989. Luego, la ISO (Organización Internacional de Normalización) lo adoptó y se le conoce como ISO C o "estándar C".

Diferencias entre C y C++ C es el lenguaje base, mientras que C++ es una versión mejorada con más funcionalidades. Por ejemplo, en C++ se usa cin y cout para entrada y salida, mientras que en C se usan funciones como scanf() y printf().

Estándares de C

- C90: Versión de C adoptada por ISO en 1990.
- C99: Nueva versión de C adoptada en 1999.
- C11: Versión más reciente del estándar, adoptada en 2011.

Estructura de un programa en C Un programa típico en C tiene:

1. Comentarios sobre el módulo, autor, y fecha.
2. Directivas del preprocesador, como #include y #define.
3. Declaraciones de variables y funciones.
4. Definiciones de funciones.
5. Uso de macros para mejorar la legibilidad del código.

Macros: Ejemplo de macro: #define PI 3.14159. Cada vez que aparece PI, se reemplaza por ese valor.

Tipos enumerados: Los tipos enumerados son una lista de valores constantes, como enum diasSemana {Lunes, Martes, Miércoles}.

Operadores: C tiene operadores aritméticos (+, -, *, /) y operadores de incremento (++) y decremento (--).

Funciones de entrada y salida: C usa printf() y scanf() para manejar la entrada y salida estándar.

Expresión condicional: El operador ?: simplifica una sentencia if-else. Ejemplo: a > b ? x : y es equivalente a if (a > b) x; else y;.

Directivas del preprocesador

1. #if, #elif, #endif: Seleccionan qué partes del código compilar según ciertas condiciones.
2. #include: Inserta el contenido de un archivo en el código.
3. #define: Define macros.
4. #undef: Elimina macros definidas previamente.
5. #pragma: Usada por el compilador para procesar directivas especiales.
6. #line: Cambia número de línea y nombre de archivo que muestra compilador en los errores.
7. # (directiva nula): Es una línea vacía que el preprocesador ignora.
8. Identificadores predefinidos:
    - __LINE__: Número de línea actual.
    - __FILE__: Nombre del archivo actual.
    - __DATE__: Fecha de la compilación.
    - __TIME__: Hora de la compilación.
    - __STDC__: Tiene valor 1 si el compilador sigue el estándar ANSI C.

- Codigo bruno: Descendente, enfoque de arriba hacia abajo, donde se comienza por el axioma de la gramática y se desciende a las producciones.
- Codigo nuestro: Ascendente, analizr el código de abajo hacia arriba, comenzando con los tokens y ascendiendo hacia el axioma.

Gramáticas y Análisis Sintáctico

Gramáticas Independientes del Contexto (GIC)

Las GIC son herramientas usadas en el análisis sintáctico para definir la estructura de un lenguaje de programación, es decir, cómo deben construirse expresiones y sentencias. Estas gramáticas verifican la estructura sintáctica, pero no pueden comprobar reglas más complejas como la declaración de variables o la compatibilidad de tipos de datos, que son parte del análisis semántico. Estan compuesta por reglas de producción.

Componentes de una Gramática

- **Símbolos no terminales:** Categorías abstractas que se descomponen en otras más simples. Se representan entre ángulos, como <expresión>.
- **Símbolos terminales:** Elementos básicos del lenguaje, como palabras clave u operadores.
- **Reglas de producción, producccion o regla gramatical:** Definen cómo un símbolo no terminal puede reemplazarse por una secuencia de terminales y no terminales, usando la forma no terminal ::= secuencia de símbolos.
    - Resultado: No terminal que genera
    - Componentes: Secuencia de no terminales y terminales que lo generan
- **Axioma:** Es el símbolo no terminal inicial desde el cual comienza la derivación de una cadena de terminales, también conocido como símbolo inicial.

Gramática Sintáctica

Una gramática sintáctica es el conjunto de reglas que define cómo deben organizarse los símbolos de un lenguaje para ser considerados válidos desde el punto de vista sintáctico. Es esencial en el análisis sintáctico, ya que describe cómo se combinan elementos como palabras clave, operadores y variables para formar expresiones e instrucciones completas. Las GIC definen estas reglas en lenguajes de programación.

Notación BNF (Backus-Naur Form)

La BNF es el formato estándar que se emplea para escribir las reglas de una GIC, permitiendo que la gramática sea formalmente entendida y procesada por herramientas de análisis sintáctico.

- **Análisis Léxico:** Convierte el código en tokens.
- **Análisis Sintáctico: Verifica la estructura gramatical.**
- **Análisis Semántico:** Valida la lógica y el significado del código.

Constantes enumeradas enum: Son los valores declarados que forman parte del enum. Cada constante se asocia a un valor entero, comenzando desde 0 por defecto, a menos que se especifique un valor diferente.

- **Lexema:** Es el texto literal encontrado en el código.
- **Token:** Es la representación estructurada del lexema, que incluye información sobre su categoría y, a veces, su valor.

Flex (analizador lexico o scanner)
La especificación de flex tiene el siguiente formato:
%{ Declaraciones }%
definiciones (expresiones regulares)
%%
reglas
%%
código del usuario(optativa, si no se quiere no es necesario el seguno %%)

yylex() es el escáner. Devuelve el token encontrado como un int.

yytext variable con el lexema encontrado, en modo flex es un char * (solo accesible desde flex, se debe copiar en un registro semántico para ser usado en bison)

El . se utiliza en la última regla para coincidir con cualquier carácter que no haya sido capturado por las reglas anteriores.

En el contexto de un analizador léxico (scanner), la forma apropiada de decir que una regla es capturada o seleccionada podría variar ligeramente según el estilo de redacción, pero aquí hay algunas expresiones comunes y precisas:

1. "La regla fue activada": Esto sugiere que la regla se ha puesto en funcionamiento debido a que el texto de entrada coincide con la expresión regular correspondiente.
2. "La regla fue coincidente": Indica que la regla ha encontrado una coincidencia con el texto de entrada.
3. "Se ha seleccionado la regla": Esta frase implica que, entre las reglas definidas, una ha sido elegida para manejar la entrada actual.
4. "La regla se aplicó": Esto implica que, después de que se encontró una coincidencia, se ejecutó la acción correspondiente de esa regla.
5. "La regla fue reconocida": Sugiere que el analizador léxico ha identificado una coincidencia con la regla.

yywrap en Flex

- **Propósito:** Indica el final del archivo de entrada en el análisis léxico.
- **Valor de Retorno:**
  - 1: No hay más archivos para procesar; finaliza el análisis.
  - 0: Hay más archivos; continúa el análisis.
- **Implementación Predeterminada:** Si no se redefine, Flex devuelve 1, deteniendo el análisis al final del archivo.

Acción Léxica: Una acción léxica es un fragmento de código que se ejecuta cuando el analizador léxico (scanner) reconoce un patrón token en la entrada.

Bison (analizador sintactico o parser)

ESTRUCTURA
%{
declaraciones en C
%}
Declaraciones de Bison
%%
Reglas gramaticales
%%
Código C adicional

Para usar múltiples tipos de datos en un analizador con Bison, se requieren dos pasos:

- **Especificar tipos:** Declarar todos los tipos de datos posibles usando %union.
- **Asignar tipos:** Elegir un tipo para cada símbolo (terminal o no terminal) mediante la declaración %token.

acción semántica: todo lo que esta entre llaves {} en una regla de producción. La denominación de "acciones semánticas" en el contexto de un analizador sintáctico como Bison se debe a la función que estas acciones cumplen en el proceso de análisis.

yyleng largo de yytext

yyin es el FILE * de donde el escáner lee la entrada, por defecto stdin

yyout es el FILE * a donde el escáner dirige su salida, por defecto stdout

- **Patrón:** Es una expresión regular (ER) que no debe estar indentada y se separa de la acción por el primer espacio (blanco) o tabulador sin comillas ni barra invertida.
- **Acción:** Es código en C que debe comenzar en la misma línea que el patrón. Si es una sentencia compuesta, puede continuar en las siguientes líneas.

Producción error en Bison: La producción error se usa para capturar y manejar errores en la entrada. Se puede incluir en la definición de una regla de producción como alternativa, permitiendo que el analizador reconozca cuando algo no se ajusta a la gramática esperada.

yyclearin: Esta función se utiliza para limpiar el estado de error del analizador, eliminando cualquier token de error que haya sido reconocido.

yyerrok: Resetea el estado de error del analizador sintáctico, permitiendo que el análisis continúe después de que se ha manejado el error.