APUNTES PARA LA MATERIA DE LENGUAJES Y AUTÓMATAS II

PARA EL PROGRAMA EDUCATIVO INGENIERÍA EN SISTEMAS COMPUTACIONALES

DEPARTAMENTO DE SISTEMAS Y COMPUTACIÓN

AUTOR: M.A.T.I. CLARIBEL BENÍTEZ QUECHA

CENTRO DE TRABAJO: TecNM CAMPUS OAXACA

28 DE FEBRERO DEL 2023

CONTRIBUCIÓN ACADÉMICA

La cada vez mayor multidisciplinariedad entre áreas profesionales y la tecnología, ha hecho de la carrera de Ingeniería en Sistemas Computacionales, una de las más demandadas a nivel global.

Las industrias implementan un mayor grado de sofisticación en los dispositivos que construyen y que tienen como base la electrónica.

El auge de los robots no es algo temporal, si no un área en crecimiento, ya que permite obtener una mayor exactitud en los procesos y el poder trabajar en contextos peligrosos para el ser humano.

Esta vinculación entre la electrónica y otros dispositivos mecánicos requiere un alto grado de conocimientos que permita manipularlos a deseo del usuario.

La materia de Lenguajes y Autómatas II dota al estudiante de la carrera de Ingeniería en Sistemas Computacionales, de los conocimientos que les permitan crear software que haga posible la interacción entre diferentes tecnologías, dispositivos o inclusive plataformas, integrando los desarrollos obtenidos para solucionar problemas de diferentes contextos o profesiones.

Un compilador es un software de sistemas que permite la interacción entre dispositivos tecnológicos de comportamientos y bases diferentes. Logrando que se comuniquen entre sí, aun teniendo diferentes lenguajes.

El ingeniero en Sistemas Computacionales se avoca principalmente al desarrollo de Software de Sistemas, esto es, el software que permite manipular dispositivos tecnológicos diferentes, para lograr que trabajen de forma colaborativa.

El compilador es el ¨core¨ de este trabajo colaborativo, ya que implementa las acciones a nivel de software que hacen posible la manipulación de dispositivos basados en tecnología, a través de un procesador computacional.

La segunda parte del proceso del compilador se cubre en la materia de Lenguajes y Autómatas II, donde el alumno adquiere las herramientas que le permiten construir software de sistemas para manipular a través de un procesador a otros dispositivos enlazados entre sí.

Desde la creación de software para juegos hasta la manipulación de robots altamente especializados, tienen como base para su manipulación y control, a un compilador.

Áreas como la domótica, que se refiere a la automatización de dispositivos de uso casero, y que son muy demandados hoy en día, están basadas para su desarrollo en la creación de compiladores. Desde apertura de puertas vía remota, encendido de luces, control de cámaras, manejo de temperatura ambiental, etc. Este desarrollo cuenta con un cada vez mayor número de adeptos, que se ha visto incrementado con el desarrollo de lo que conocemos como el internet de las cosas.

Un compilador también es el software de sistemas que permite desarrollar robots tan especializados como se requiera.

Así también la teoría de autómatas que se cubre en esta materia le da un sentido al desarrollo que se ha logrado en el ámbito de la Inteligencia Artificial, permitiendo integrar ésta a un mayor número de usos tanto en la industria, la economía, la salud, la

educación, proporcionando un abanico enorme de posibilidades en un gran número de contextos.

Otra área de aplicación para la Teoría de Autómatas es la de las interfaces hombre-máquina o HMI, que son altamente demandadas en la actualidad, para la robotización y computarización de acciones y eventos que requieran detonar acciones en base a una señal. Acciones de manipulación de dispositivos mecánicos y/o electrónicos.

Para todo el desarrollo anteriormente mencionado se requiere un compilador, por lo que esta materia tiene un aporte significativo dentro de la carrera de Ingeniería en Sistemas Computacionales.

Podemos resumir, acotando, que esta materia complementa la información adquirida en la materia de Lenguajes y Autómatas I, ya que en ella se conoce una de las dos fases del proceso de un compilador, a saber la Fase de Análisis, y en Lenguajes y Autómatas II se termina de conocer la fase de Análisis con el análisis semántico y la totalidad de fase de Síntesis, donde el estudiante ya se encuentra capacitado para crear sus propios compiladores, al entender la teoría de autómatas y las diferentes técnicas del proceso del compilador en ésta segunda fase, donde se requiere un amplio conocimiento de Lenguaje Ensamblador, conocimiento que es a su vez adquirido en la materia de Lenguajes de Interfaz.

INTRODUCCIÓN.

Un compilador es un programa que traduce el código fuente de un lenguaje de programación a un lenguaje de máquina (código objeto).

El esquema de traducción de un compilador generalmente se compone de los siguientes pasos:

1. Análisis léxico: el compilador escanea el código fuente para encontrar las palabras clave y los tokens.

2. Análisis sintáctico: el compilador verifica la sintaxis del código fuente para garantizar que se estén usando los patrones correctos.

3. Análisis semántico: el compilador verifica el significado de los tokens para determinar si el código tiene sentido.

4. Generación de código intermedio: el compilador genera código intermedio para representar el programa.

5. Optimización: el compilador realiza cambios en el código intermedio para mejorar el rendimiento y la eficiencia.

6. Generación de código objeto: el compilador genera el código objeto, que es una versión binaria del programa.

7. Ensamblado: el compilador ensambla el código objeto en el lenguaje de máquina correspondiente. Hasta aquí termina el proceso del compilador.

8. Generación de ejecutable: El enlazador o linker genera un archivo ejecutable que contiene el código traducido. El ejecutable se puede ejecutar en la máquina.

Las primeras acciones del compilador como son: Análisis léxico y Análisis Sintáctico son cubiertas en la materia de Lenguajes y Autómatas I, y las subsecuentes (Análisis semántico, Generación de código intermedio, Optimización y Generación de código objeto), se ven en esta materia, Lenguajes y Autómatas II.

Estos apuntes están esquematizados de la siguiente manera:

Cada capítulo del presente trabajo engloba una unidad temática de acuerdo a la retícula de la materia de Lenguajes y Autómatas II con clave SCD-1016, definida para la carrera de Ingeniería en Sistemas Computacionales con clave ISIC-2010-224 del tecnológico Nacional de México.

Por cada unidad temática se desarrollan diferentes complementos educacionales como antologías con información recabada de diferentes fuentes bibliográficas, ejercicios para resolver con sus respectivas soluciones, vídeos explicativos, exámenes indicadores, y como corolario final se define un proyecto integrador.

ÍNDICE

TABLA DE CONTENIDO

I. DESARROLLO

1. Tema 1 Análisis semántico.

Un análisis semántico en compiladores es una etapa de la compilación de un programa de computadora en la que se realiza un análisis de los significados de las instrucciones del programa. Esta etapa se encarga de verificar que todas las instrucciones estén bien formadas y tengan sentido. Esto significa que el compilador comprueba que los tipos de datos sean correctos, que los operadores estén al lado correcto de la expresión que se está evaluando y que todas las variables estén correctamente definidas. El análisis semántico también se encarga de detectar errores lógicos en el programa, como la falta de cierre de un bucle o el uso de una variable antes de que se haya definido. Esto ayuda a los programadores a detectar errores en sus programas antes de su uso. En general, el análisis semántico se lleva a cabo después del análisis sintáctico, ya que es importante que los programas sean correctos desde un punto de vista lógico antes de que se ejecuten. Algunos compiladores también realizan un análisis semántico en el mismo momento en que detectan errores sintácticos, para que el programador pueda solucionarlos de inmediato.

Refiriéndonos a la estructura operacional de un compilador, podemos visualizar en qué parte del proceso de compilación, se lleva a cabo el análisis semántico, lo cual podemos ver en la siguiente figura (figura 1).

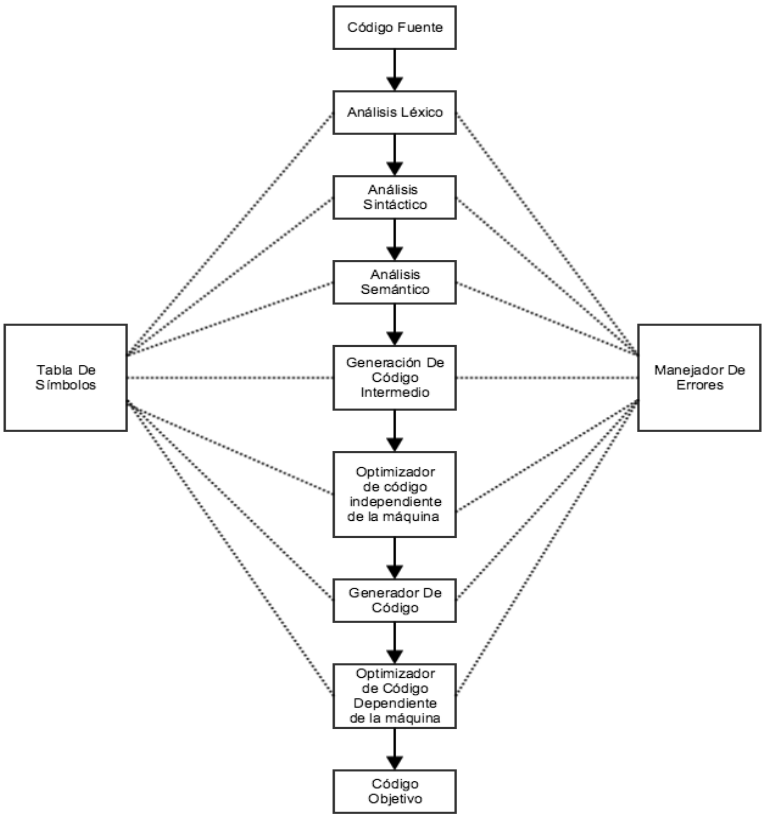


Figura. 1 Geek, Ricardo (2016). Fases de un compilador (Diseño). https://ricardogeek.com/fases-del-compilador/

El análisis semántico se compone de un conjunto de rutinas independientes, llamadas por los analizadores lexicográfico y sintáctico. Utiliza como entrada el árbol sintáctico generado por el análisis sintáctico para llevar a cabo el proceso de validación de los lineamientos con respecto a los tipos y otras especificaciones semánticas, y pasar a la fase de síntesis, donde se lleva a cabo la generación de código. El instrumento más utilizado para conseguirlo es la gramática de atributos.

Lleva a cabo las comprobaciones necesarias sobre el árbol sintáctico para determinar el correcto significado del programa.

Las tareas básicas a realizar son:

* Verificar que las variables usadas estén declaradas.

* La verificación e inferencia de tipos en asignaciones y expresiones, la declaración del tipo de variables y funciones antes de su uso, el correcto uso de operadores, el ámbito de las variables y la correcta llamada a funciones, así como comprobar corrección semántica.

Estos apuntes se limitan al análisis semántico estático (en tiempo de compilación), donde es necesario hacer uso de la Tabla de símbolos, como estructura de datos para almacenar información sobre los identificadores que van surgiendo a lo largo del programa. El análisis semántico suele agregar atributos (como tipos de datos) a la estructura del árbol semántico.

1.1. Árboles de expresiones.

Los árboles son estructuras de datos que tienen una forma jerárquica, compuestos por elementos organizados y que son dinámicos. Los elementos son conocidos como nodos.

- Jerárquica, porque sus elementos se ubican por niveles.
- Organizada, porque hay una lógica que guía la formación de un nodo.
- Dinámica porque sus características, como forma, tamaño y contenido pueden ir variando durante la ejecución de estos.

Un tipo especial de árbol, son los árboles binarios, los cuales tienen la característica de que son árboles de grado 2, esto es, que cualquiera de sus nodos puede tener como máximo dos hijos o nodos de derivación.

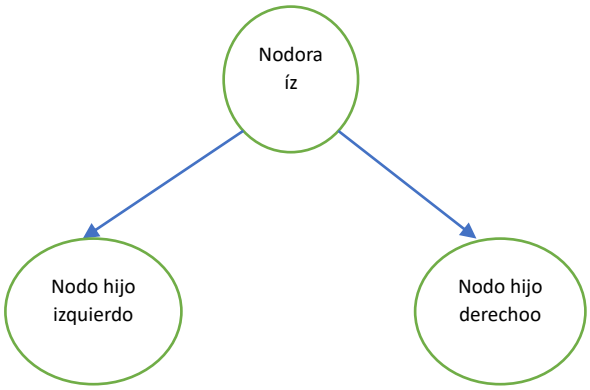


Figura 2. Elementos de un árbol de expresión.

Un árbol de expresión es una estructura de datos utilizada para representar una expresión matemática, lógica o de programación en un lenguaje de programación. Esta estructura se compone de operadores, constantes y variables como nodos en una jerarquía de ramas y hojas. El árbol de expresión es utilizado por los compiladores para transformar una expresión en una secuencia de instrucciones que el procesador pueda entender.

Un árbol de expresión deriva de los árboles binarios, y hereda todas las características de estos, por ejemplo: que un nodo sólo puede tener como máximo dos hijos.

Una expresión es una secuencia de tokens (componentes léxicos), los cuales siguen reglas establecidas por una gramática. El token puede ser un operador o un operando.

En el árbol de expresión los tokens están representados por los nodos, los cuales pueden ser:

- Operadores: son nodos que representan acciones matemáticas, lógicas o de programación. Y están compuestos por los símbolos +, -, *, /, %, ==, !=, &&, ||, <, >, etc.
- Constantes: estos nodos contienen valores numéricos. Representan números enteros, reales, cadenas de caracteres, etc.
- Variables: las variables son nodos representan identificadores, cuya información puede cambiar durante el proceso de ejecución del programa.

Estos nodos se utilizan para almacenar información que el usuario ingresa o que el programa genera durante su ejecución.
Se utiliza para convertir una expresión en una secuencia de instrucciones que el procesador pueda entender.

Esta secuencia de instrucciones se conoce como código objeto y se utiliza para ejecutar el programa.

Los árboles de expresión son una herramienta muy útil para los compiladores, ya que les permiten transformar expresiones complejas en instrucciones simples entendibles por los procesadores. Esto permite que los programadores escriban código de manera más sencilla y eficiente.

En conclusión, un árbol de expresión para un compilador es una estructura de datos que se utiliza para representar expresiones matemáticas, lógicas o de programación en un lenguaje de programación. Esta estructura está compuesta por tokens, que pueden ser operadores, constantes y variables, y se utiliza para transformar las expresiones en código objeto entendible por los procesadores.

Una de las aplicaciones de árboles binarios son los llamados árboles de expresión.

Las propiedades de un árbol de expresión son las siguientes:

- Cada hoja es un operando
- El nodo raíz y los nodos internos son operadores
- Los subárboles son sub-expresiones en las que el nodo raíz es un operador

La siguiente figura muestra un ejemplo de un árbol de expresión de la expresión

(a+b) * (c-d).

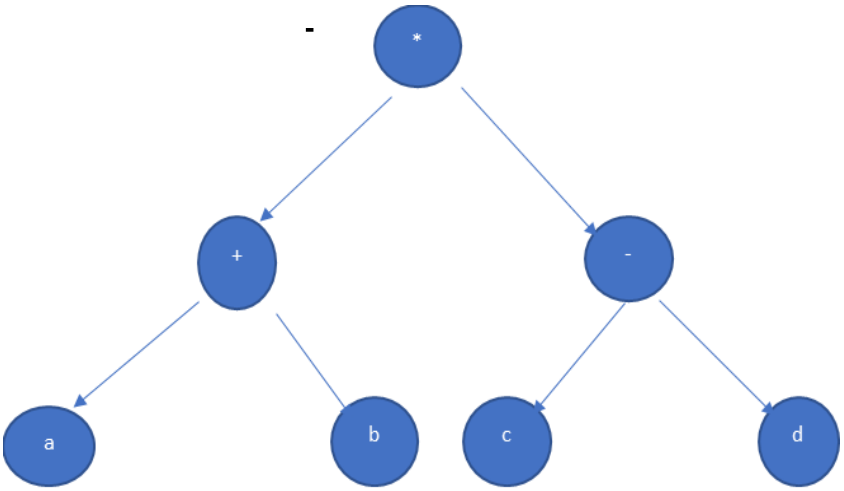


*Figura 3. Árbol de expresión para la expresión (a+b) * (c-d).*

Algoritmo para la construcción de un árbol de expresión.

Mientras carácter diferente de nulo
Leer carácter de la lista
Si es paréntesis pasar al siguiente carácter
Crear un nodo nuevo que contenga ese carácter
Si el carácter que tiene nuevo es un:
Operando
    si el árbol está vacío hacer raíz a nuevo
    si no recorrer el árbol por la derecha hasta llegar a un nodo con hojas
        si la hoja izquierda, no está etiquetada colocar operando
        si no colocarlo en la hoja derecha.
Operador
    si la raíz es un operando
        insertar nuevo en ese nodo, y convertir el operando
        en el hijo izq.
    si no
    si hay un paréntesis abierto
        insertar nuevo en la última hoja derecha y colocar operando
        como hijo izquierdo

Para llevar a cabo la evaluación de un árbol de expresión, cada nodo se debe declarar de la siguiente manera:

```
# define OPERADOR 0
# define OPERANDO 1
struct nodarbol{
   short int tipo;    //OPERADOR U OPERANDO//
   union{
      char operador[10];
      float val;
}info;
struct nodarbol *izq;
struct nodarbol *der;
```

```
};
typedef nodarbol *NODEPTR;


void evalua(NODEPTR raiz){
    float value;
    NODEPTR aux;
    evalua(aux->izq);
        evalua(aux->der);
        apply(aux);
}
```

La función **evalúa** árbol queda de la siguiente forma;

```
float evalarbol(NODEPTR raiz){

evalua(raiz);

     return (raiz->val);

     free(raiz);

}
```

Si consideramos que la parte de información de un nodo puede contener tanto operandos como operadores, la parte del nodo que es la información se considera un componente unión de la estructura.

Para poder realizar la función evalúa se requiere la función apply(p), la cual se deja para realizar por cuenta propia, esta debe aceptar un apuntador a un árbol de expresión que contiene un operador único y sus operandos numéricos, y retornar lo obtenido al evaluar la expresión aplicando el operador a los operandos.

La función **evalúa** recibe un apuntador a un árbol de expresión y lo sustituye creando un nodo que ya contiene la evaluación de la expresión.

1.2. Acciones semánticas de un analizador sintáctico.

Las acciones semánticas de un analizador sintáctico se refieren a la generación de código o acciones adicionales realizadas por el analizador sintáctico en respuesta a los patrones de entrada, después de que haya analizado correctamente una frase.

Las acciones semánticas se usan para realizar tareas como la asignación de tipos a los operandos, la generación de código intermedio para la ejecución de expresiones, la evaluación de condiciones, etc.

Estas acciones suelen estar codificadas en el código generado por el analizador sintáctico, lo que permite al compilador realizar tareas específicas al procesar el código fuente.

Es importante destacar que las acciones semánticas no son parte del análisis sintáctico en sí mismo. En su lugar, se consideran acciones separadas que se realizan después de que el analizador sintáctico haya procesado correctamente la entrada.

Las acciones semánticas son tareas realizadas por un analizador sintáctico durante el proceso de análisis.

Estas tareas incluyen la generación de tokens, la verificación de la sintaxis de un lenguaje y la evaluación de expresiones. Normalmente se realizan a medida que se recorren los elementos de un programa.

Algunos de los ejemplos más comunes de acciones semánticas incluyen:

- Verificación de tipos: verificar que los tipos de los operandos coinciden con las reglas de un lenguaje específico.

- Verificación de reglas: verificar que una operación es válida según las reglas de un lenguaje específico.

- Evaluación de expresiones: calcular el valor de una expresión dada. - Generación de código intermedio: generar código intermedio para una determinada expresión.

- Generación de código objeto: generar código objeto a partir del código intermedio.

- Verificación de alcance: verificar que los nombres de variables, procedimientos y clases se han definido correctamente.

- Verificación de errores: detectar errores en el código fuente.

- Generación de informes: generar informes de errores, avisos o advertencias. - Generación de tablas de símbolos: generar tablas de símbolos que contengan información acerca de los objetos utilizados en el programa.

- Generación de optimizaciones: detectar oportunidades de optimización en el código fuente.

- Generación de análisis estático: realizar análisis estáticos como el análisis de flujo de datos para identificar errores de programación.

- Generación de análisis dinámico: realizar análisis dinámicos como el análisis de tiempo de ejecución para identificar errores de programación.

- Generación de informes de seguridad: generar informes de seguridad para ayudar a los desarrolladores a detectar vulnerabilidades de seguridad en el código. - Generación de documentación: generar documentación para ayudar a los desarrolladores a comprender un programa.

- Verificación de estilo: verificar que el código está escrito según las convenciones de un lenguaje específico.

- Verificación de correctitud: verificar que el código cumple con los requisitos de un lenguaje específico.

- Verificación de consistencia: verificar que el código se adhiere a los estándares de un lenguaje específico.

- Verificación de portabilidad: verificar que el código se puede compilar en diferentes plataformas.

- Verificación de rendimiento: verificar que el código está optimizado para obtener el mejor rendimiento.

- Generación de pruebas: generar pruebas para validar la lógica de un programa.

- Verificación de cobertura: verificar que el código se ha probado completamente. - Generación de informes de prueba: generar informes de prueba para evaluar el comportamiento de un programa.

- Verificación de calidad: verificar que el código cumple con los estándares de calidad.

1.3. Comprobaciones de tipos en expresiones.

Las comprobaciones de tipos en expresiones se refieren a la verificación de que los tipos de datos de los elementos de una expresión sean los correctos para que la expresión se pueda evaluar correctamente.

Esto es especialmente importante en lenguajes de programación como Java, donde los tipos de datos tienen que coincidir para que la expresión se pueda evaluar con éxito.

Por ejemplo, si una expresión de suma contiene un entero y una cadena, se producirá un error de tipo, ya que un entero no se puede sumar con una cadena. Si la expresión contiene dos enteros, se evaluará con éxito. Esto es una comprobación de tipos que se realiza antes de que la expresión se evalúe.

En otros lenguajes de programación como Python, las comprobaciones de tipos no son necesarias porque el lenguaje soporta la conversión de datos implícita.

Esto significa que, aunque una expresión contenga un entero y una cadena, Python intentará convertir automáticamente la cadena en un entero para que la expresión se pueda evaluar con éxito.

Sin embargo, en algunos casos, esta conversión implícita no es posible, de modo que se producirá un error. En este caso, se debería realizar una comprobación de tipos para asegurarse de que los tipos de datos sean los correctos para que la expresión se evalúe con éxito.

La tarea de la verificación de tipos es dar semántica de escritura a las construcciones si ntácticas
del lenguaje y realizar todo tipo de verificación de tipos. Sin embargo, esto se divide na turalmente en una fase de análisis semántico y una fase de generación de código inter medio.

La verificación de tipos puede ser una tarea que se lleve a cabo de forma estática o dinámica.

• Comprobaciones estáticas

Las comprobaciones estáticas contienen un resumen de todas las tareas de naturaleza semántica que se pueden realizar directamente durante la fase de compilación utiliza ndo los artefactos y mecanismos típicos de esta fase. Estos tipos de controles son bene ficiosos porque agregan seguridad a la ejecución del programa.

.Propiedades

- Difiere del tiempo de ejecución dinámico.
- Ejemplos: verificación de tipos, flujo de control, unicidad.

• Comprobaciones dinámicas

Las comprobaciones dinámicas no se realizan durante la etapa de compilación, sino q ue se delegan en el tiempo de ejecución del programa. Esto requiere generar un códig o ejecutable diseñado específicamente para realizar dichas comprobaciones. Los leng uajes que realizan muchas pruebas dinámicas hacen que los programas se ejecuten d urante más tiempo, son más lentos y menos seguros.

. Verificación de tipos
Obtiene información mientras procesa declaraciones para verificar la compatibilidad de tipos para todas las expresiones en el código fuente. También garantiza que no haya referencias a símbolos no declarados en su programa.

.Inferencia de tipo

En lenguajes sin variables tipificadas ni sobrecarga, la tarea de inferencia de tipo se aplica a nivel de la gramática de la expresión para resolver el tipo de datos de la expresión resultante en función del contexto de evaluación.

En definitiva, las comprobaciones de tipos en expresiones son necesarias en lenguajes de programación como Java para asegurarse de que los tipos de datos sean los correctos. En lenguajes como Python, las comprobaciones de tipos no son necesarias, pero en algunos casos es aconsejable realizarlas para evitar errores.

Implementación de la rutina para un comprobador de tipos sencillo, en lenguaje pascal, para la declaración de tipos.

```
lCampos ::= ID:id DP rTipo:t {:
String name = id.getLexema ();
TypeIF type = t.getType ();
TypeRecord tR = new TypeRecord ();
tR.addField (name, type);
RESULT = new LCampos (tR, type);
:}
| ID:id COMA lCampos:t {:
String name = id.getLexema ();
TypeRecord tR = t.getTypeRecord ();
if (!(tR.containsField (name))) {
TypeIF type = t.getType ();
tR.addField (name, type);
} else
semanticErrorManager.semanticFatalError (...);
:}
```

```
dVariables ::= dVariables dVariable
| dVariable;
dVariable ::= ID:id DP rTipo:t {:
String name = id.getLexema ();
TypeIF type = t.getType ();
ScopeIF scope = scopeManager.getCurrentScope ();
SymbolTable sTable = scope.getSymbolTable ();
if (!(sTable.containsSymbol (name))) {
SymbolVariable sV = new SymbolVariable (name, type);
sTable.addSymbol (name, sV);
RESULT = new DVariables (type);
} else semanticErrorManager.semanticFatalError (...);
:}
| ID:id COMA dVariables:t {:
<<igual que en la regla anterior>>
:};
```

Implementación de la rutina para un comprobador de tipos sencillo, en lenguaje pascal, para la declaración del tipo ´expresión´.

```
exp ::= exp:e1 MAS exp:e2 {:
TypeIF t1 = e1.getType ();
TypeIF t2 = e2.getType ();
if (t1.isCompatible (t2, TypeSimple.MAS)) {
TypeIF t = t1.cast (t2, TypeSimple.MAS);
RESULT = new Exp (t);
} else semanticErrorManager.semanticFatalError (...);
:}
| exp MENOS exp {:...:}
| exp POR exp {:...:}
| exp DIV exp {:...:}
| rNumero:rn {:
Object value = rn.getValue ();
TypeIF type;
if (value instanceof Integer) type = TypeSimple.ENTERO;
if (value instanceof Float) type = TypeSimple.REAL;
RESULT = new Exp (type); :}
```

1.4. Pila semántica en un analizador sintáctico.

La pila semántica es una parte fundamental de un analizador sintáctico. Es una estructura de datos que se utiliza para almacenar información semántica durante el análisis sintáctico. Esta información se utiliza para verificar la correcta interpretación de la entrada y para construir el árbol de sintaxis abstracta (AST) que los intérpretes y compiladores usan para generar código.

La pila semántica se inicializa con una referencia al nodo raíz del árbol de sintaxis abstracta. A medida que el analizador sintáctico reconoce una entrada, se añaden elementos a la pila semántica.

Estos elementos incluyen información sobre la regla sintáctica que se está procesando, el valor de los parámetros, los nodos del árbol de sintaxis abstracta creados como resultado del reconocimiento de la entrada, y otros.

Cuando el analizador sintáctico completa el reconocimiento de la entrada, la pila semántica contiene toda la información necesaria para construir el árbol de sintaxis abstracta.

El analizador sintáctico recorre la pila semántica, procesando los elementos de la misma para generar el árbol de sintaxis. Una vez que el árbol está completo, se pasa al intérprete o compilador para generar código.

En resumen, la pila semántica es una parte fundamental de un analizador sintáctico. Se utiliza para almacenar información durante el análisis sintáctico y para construir el árbol de sintaxis abstracta. Esta información se pasa al intérprete o compilador para generar código.

Aquí tenemos un ejemplo simplificado de cómo se podría implementar una pila semántica en Python para evaluar una expresión aritmética que contiene operaciones de suma y resta. Este código se centra en el proceso de evaluación y utiliza una pila para rastrear los operandos y operadores:

```
class PilaSemantica:
    def __init__(self):
        self.pila = []

    def apilar(self, valor):
        self.pila.append(valor)

    def desapilar(self):
        if not self.esta_vacia():
            return self.pila.pop()
        else:
            raise Exception("La pila está vacía")

    def esta_vacia(self):
        return len(self.pila) == 0

def evaluar_expresion(expresion):
    pila = PilaSemantica()
    tokens = expresion.split()  # Supongamos que la expresión está en notación posfija (postfix)

    for token in tokens:
        if token.isdigit():
            pila.apilar(int(token))
        elif token in ('+', '-'):
            if len(pila.pila) < 2:
                raise Exception("Error: No hay suficientes operandos para la operación.")
            operando2 = pila.desapilar()
            operando1 = pila.desapilar()
            if token == '+':
                resultado = operando1 + operando2
            else:  # token == '-'
                resultado = operando1 - operando2
            pila.apilar(resultado)
        else:
            raise Exception(f"Error: Token no reconocido - {token}")
```

```
    if len(pila.pila) == 1:
        return pila.desapilar()
    else:
        raise Exception("Error: La pila contiene más de un valor al final de la evaluación.")

# Ejemplo de uso
expresion = "3 4 + 5 -"
try:
    resultado = evaluar_expresion(expresion)
    print(f"Resultado de la expresión: {resultado}")
except Exception as e:
    print(f"Error: {e}")
```

En este ejemplo, se ha creado una clase PilaSemantica que implementa una pila básica. La función evaluar_expresion toma una expresión en notación posfija (postfix), la divide en tokens y luego recorre esos tokens para evaluar la expresión. La pila se utiliza para realizar el seguimiento de los operandos y operadores.

El ejemplo de uso muestra cómo se puede utilizar esta implementación para evaluar la expresión "3 4 + 5 -", que representa la operación (3 + 4) - 5.

Hay que tener en cuenta que este ejemplo se enfoca en expresiones posfijas simples con operaciones de suma y resta. La implementación de una pila semántica para expresiones más complejas requeriría manejar operadores adicionales y paréntesis, así como reglas de precedencia y asociatividad.

1.5. Esquema de traducción.

¨Un esquema de traducción es una notación para unir los fragmentos de un programa a las producciones de una gramática. Los fragmentos del programa se ejecutan cuando se utiliza la producción durante el análisis sintáctico. El resultado combinado de todas estas ejecuciones de los fragmentos, en el orden inducido por el análisis sintáctico, produce la traducción del programa al cual se aplica este proceso de análisis/síntesis¨. AHO

Los fragmentos de un programa incrustados dentro de los cuerpos de las producciones se llaman acciones semánticas. La posición en la que debe ejecutarse una acción se muestra encerrada entre llaves y se escribe dentro del cuerpo de producción, como en el siguiente ejemplo:

resto → + term {print(´+´)} resto1

Las traducciones orientadas a la sintaxis son utilizadas para traducir una expresión infija a una notación postfija, así también para llevar a cabo la evaluación de expresiones y la construcción de árboles sintácticos.

Un esquema de traducción orientado a la sintaxis (TOS) es un método para traducir un lenguaje de programación en otro.

La idea es que la entrada de un programa se procesa por un compilador para producir una salida en otro lenguaje. El compilador usa un TOS para analizar la sintaxis de la entrada, descomponerla en partes y luego generar la salida en el lenguaje de destino.

Un compilador que usa un TOS normalmente comienza por analizar la sintaxis de la entrada. Esto significa que el compilador debe tener una comprensión de la gramática del lenguaje de entrada, para que pueda identificar la estructura de los programas. Una vez que el analizador sintáctico ha identificado la estructura de la entrada, el compilador puede descomponer el programa en un árbol de análisis sintáctico.

Esta estructura de árbol representa la estructura de la entrada, con la sintaxis como los nodos y los elementos del programa como los hijos. Una vez que el compilador ha descompuesto el programa en un árbol de análisis sintáctico, puede comenzar a generar la salida en el lenguaje de destino. El compilador recorrerá el árbol de análisis sintáctico, traduciendo los nodos y los elementos del programa en el lenguaje de destino.

Esta traducción se realiza usando un conjunto de reglas predefinidas que dictan cómo los elementos de un lenguaje se traducen en los del otro.

Estas reglas se conocen como reglas del compilador, y se basan en la sintaxis y la semántica de los lenguajes involucrados.

Una vez que el compilador ha traducido todos los elementos de la entrada en el lenguaje de destino, genera la salida. Esta salida suele ser un archivo de código fuente en el lenguaje de destino, que se puede compilar y ejecutar.

Por lo tanto, el compilador usando un TOS es un útil medio para traducir un lenguaje de programación en otro.

Ejemplo:
Dada la siguiente gramática, derivar las expresiones regulares para reconocer la expresión ´a+b´.
Gramática

S -→ E

E -→ E+T|E-T|T

T --→ T*F|T/F|F

F -→ ID|NUM

ID -→ LETRA(LETRA|DIGITO)*

NUM -→ DIGITO(DIGITO)*

LETRA --→ a..z

DIGITO --→ 0..9

Reconocimiento:

S -→ E

E -→ E+T

E -→ T

T -→ F

F -→ ID

E → T + T

E → F +F

E → ID + ID

ID -→ LETRA(LETRA|DIGITO)* E → LETRA + LETRA

LETRA --→ a..z E → a + b

Para determinar si una expresión pertenece o no a una gramática, podemos utilizar un analizador sintáctico que siga las reglas de la gramática para verificar la estructura de la expresión. A continuación, se proporciona un seudocódigo básico que muestra cómo podríamos hacer esto utilizando un analizador sintáctico descendente recursivo (top-down) en un lenguaje de alto nivel:

Supongamos que se tiene una gramática simple para expresiones aritméticas con números, operadores de suma y resta, y paréntesis:

```
r

E -> E + T | E - T | T
T -> T * F | T / F | F
F -> (E) | número
```

Aquí tenemos el seudocódigo que verifica si una expresión pertenece a esta gramática:

```plaintext
funcion esExpresionValida(expresion):
    indice <- 0  # Inicializa el índice en el primer carácter de la expresión
    resultado <- esE(indice)  # Comienza con la regla E

    si resultado es True y indice es igual a la longitud de la expresión:
        retornar True
    sino:
        retornar False

funcion esE(indice):
    resultado <- esT(indice)
    mientras expresion[indice] es '+' o expresion[indice] es '-':
        indice <- indice + 1
        resultado <- resultado y esT(indice)
    retornar resultado

funcion esT(indice):
    resultado <- esF(indice)
    mientras expresion[indice] es '*' o expresion[indice] es '/':
        indice <- indice + 1
        resultado <- resultado y esF(indice)
```

```
funcion esF(indice):
    si expresion[indice] es '(':
        indice <- indice + 1
        resultado <- esE(indice)
        si expresion[indice] es ')':
            indice <- indice + 1
            retornar resultado
        sino:
            retornar False
    sino si expresion[indice] es un número:
        indice <- indice + 1
        retornar True
    sino:
        retornar False
```

Este seudocódigo utiliza un enfoque de análisis sintáctico descendente para verificar si la expresión cumple con las reglas de la gramática. Comienza con la regla E y luego se desplaza hacia abajo en las reglas de la gramática según los operadores y paréntesis en la expresión. Si al final se llega al final de la expresión y todas las reglas se cumplen, entonces la expresión es válida según la gramática, de lo contrario, se considera inválida.

Hay que tener en cuenta que este es un ejemplo simplificado para una gramática simple. Las gramáticas más complejas requerirán un analizador sintáctico más sofisticado, posiblemente implementado utilizando técnicas como el análisis sintáctico descendente recursivo, análisis sintáctico LR o análisis sintáctico LL. Además, este seudocódigo es de alto nivel y debe adaptarse e implementarse en un lenguaje de programación específico para su uso real.

1.6. Generación de la tabla de símbolo y tabla de direcciones.

La tabla de símbolos es una estructura de datos utilizada por un compilador para guardar o almacenar información sobre los símbolos o identificadores del programa fuente. Por ejemplo, la tabla de símbolos puede almacenar el nombre de una variable o símbolo, el tipo de datos que contiene y su ubicación en memoria.

Esta tabla se construye durante el proceso de compilación. En la tabla de símbolos se almacenará el nombre del símbolo, su tipo (entero, real, etc), su valor si es una constante, su escope (ámbito en el cual está definido), si se trata de una función, el número de argumentos, etc. Esta información es necesaria para realizar la compilación y el enlazado del programa.

La Tabla de Símbolos es usada durante la compilación para verificar que todos los símbolos sean definidos correctamente. También es usada por la generación del código para construir la tabla de símbolos del programa objeto.

La Tabla de Símbolos es una tabla con una estructura de hash, es decir, los símbolos son almacenados en la tabla en una forma que permite una búsqueda rápida. Esto significa que el compilador puede encontrar un símbolo a partir de su nombre en un tiempo constante. Esta tabla contiene información acerca de los símbolos utilizados en el programa fuente y es usada por el enlazador para relacionar el código objeto con los símbolos del programa fuente.

Tabla de direcciones La tabla de direcciones es una estructura de datos utilizada por un compilador para guardar información sobre los elementos del programa. Por ejemplo, el tamaño de los datos, la dirección de memoria donde se encuentran los datos, las etiquetas para el salto, etc. Esta información es necesaria para realizar el enlazado del programa.

La tabla de direcciones se usa para almacenar información sobre la ubicación de los datos en la memoria. Contiene dos campos, el primero es el nombre del símbolo y el segundo es la dirección de memoria donde se encuentran los datos asociados con el símbolo.

Esta tabla es útil para localizar los datos en la memoria una vez que se haya compilado el programa.

Además de estas dos tablas, los compiladores también utilizan otras estructuras de datos para realizar el proceso de compilación y enlazado. Entre ellas podemos encontrar la tabla de símbolos externos, la tabla de relocación, la tabla de errores, etc.

Estas tablas son necesarias para realizar el proceso de compilación y enlazado de forma eficiente y segura.

El compilador utiliza estas estructuras de datos para optimizar el proceso de compilación y enlazado, reduciendo los tiempos de ejecución.

Una tabla de símbolos es realmente una tabla lineal o una tabla de tipo hash. En ella se registra una entrada para cada uno de los símbolos definidos por el usuario siguiendo el formato:

```
2.      <nombre del símbolo, tipo, atributo>
```

Por ejemplo, dada la siguiente declaración de variables:

```
3.      static int interest;
```

La entrada a la tabla se registra de la siguiente manera:

```
4.      <interest, int, static>
```

Para la creación de las tablas de símbolos, se utilizan algunos de los siguientes formatos:

- Lista lineal (ordenadas o desordenadas)
- Árbol de búsqueda binaria

- Tabla Hash

1.7. Manejo de errores semánticos.

Un error semántico ocurre cuando el código se interpreta de manera diferente a lo que se pretendía. Por lo tanto, el manejo de errores semánticos implica identificar la causa del error y corregirla.

Algunos ejemplos de errores semánticos son escribir el código de manera incorrecta, escribir el código de manera confusa o mal interpretar el propósito de una función.

Los errores semánticos son errores que ocurren cuando el programa cumple con el lenguaje de programación, pero la lógica del programa no es correcta. Ocurren cuando el compilador interpreta incorrectamente el significado de una expresión o declaración Estos errores suelen ser difíciles de detectar y solucionar para un compilador, ya que no se identifican durante la compilación, porque para detectarlos se requiere un entendimiento del contexto y el significado de la expresión.

La mejor forma de manejar errores semánticos durante la compilación es usar un compilador con un analizador léxico y sintáctico, lo que ayudará a detectar errores de sintaxis. Esto hará que el proceso de compilación sea más eficiente, ya que el compilador detectará errores antes de que el programa se ejecute.

Además, se debe tener en cuenta que los errores semánticos son más difíciles de detectar que los errores sintácticos, por lo que es importante que los desarrolladores sigan buenas prácticas de programación. Esto incluye documentar el código de manera adecuada para facilitar el seguimiento de errores, probar el código para detectar cualquier fallo lógico y realizar pruebas de regresión para confirmar que los cambios no provocan errores. Finalmente, los errores semánticos se pueden solucionar mediante la depuración. Esto significa que el programador debe investigar el código para encontrar el origen del problema y solucionarlo.

Para ayudar a un compilador a detectar errores semánticos, se pueden emplear herramientas como las reglas de análisis semántico, que le permiten al compilador comprobar si una sentencia cumple con ciertas reglas. Estas reglas pueden incluir, por ejemplo, la comprobación de tipos de datos, la especificación de los parámetros de una función, la comprobación de límites de variables, etc.

Además, los compiladores también pueden usar herramientas de análisis léxico para detectar errores semánticos. Estas herramientas permiten al compilador identificar palabras clave y símbolos para comprobar si están en el contexto correcto.

Finalmente, el compilador también puede usar herramientas de prueba de unidad para detectar errores semánticos. Estas herramientas permiten al compilador ejecutar las instrucciones individualmente para comprobar si hacen lo que se espera. Esto ayuda a asegurar que todas las instrucciones están escritas correctamente y que no hay errores semánticos.

Para manejar los errores semánticos, es recomendable una revisión exhaustiva del código para verificar si hay errores lógicos, malas interpretaciones, etc. También es útil realizar pruebas de regresión para verificar si el código se comporta de la manera esperada.

El manejo de errores semánticos implica identificar y corregir la causa del error.

ACTIVIDADES DE APRENDIZAJE DE LOS TEMAS DE LA ASIGNATURA

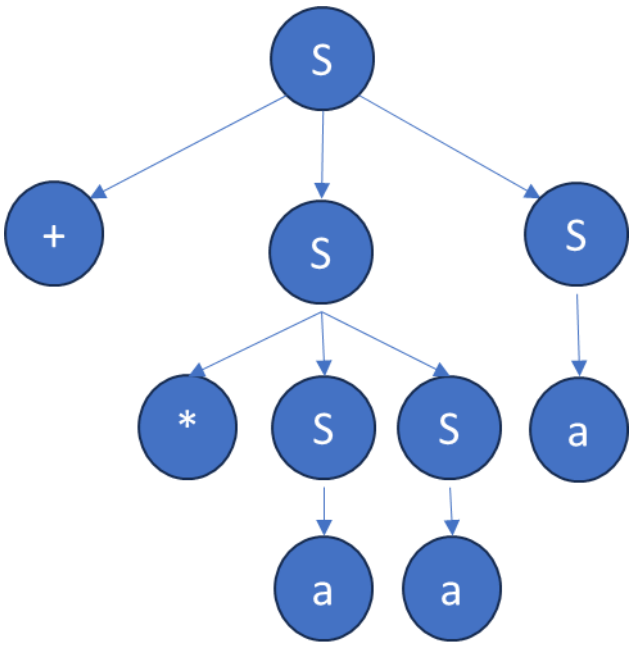
A Recursos de Evaluación

Evaluación diagnóstica

1.- Defina el concepto de GRAMÁTICA.
Es el conjunto de reglas sintácticas y semánticas que definen un lenguaje, y que permiten la creación correcta de una instrucción.

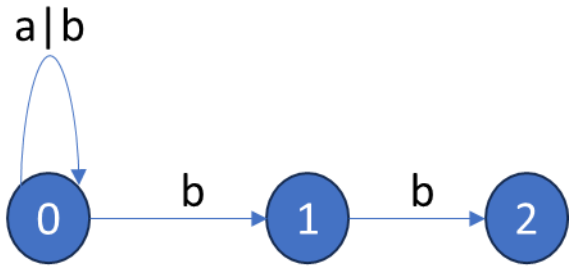
2.- Cuál es la principal función del analizador léxico?
La función principal del analizador léxico es convertir el flujo de caracteres de una fuente de programa en un flujo de tokens o lexemas, que son unidades significativas del lenguaje de programación. Estos tokens son la base para el análisis sintáctico y semántico posterior de un programa en un compilador o intérprete.

3.- Para: S -> +SS| *SS| a, reconozca y elabore el árbol sintáctico para la cadena: + * aaa



4.- Establezca las reglas de producción que permitan reconocer la expresión: x+6
E → E | E+T| E-T| T
T → T*F|T/F|F
F → ID|DIGITO
ID → LETRA[LETRA|DIGITO]*
LETRA → a..z
DIGITO → 0..9

5.- Represente el AFN para la expresión regular: (a|b)*bb.



Evaluaciones parciales

1. ¿Cuáles son las tareas del analizador semántico?

Registrar declaraciones, Inferir tipos, Comprobar tipos, Comprobar corrección semántica.

2. ¿A qué le llamamos símbolo, dentro de un programa?

Es cualquier elemento con nombre que el programador haya establecido como válido dentro de él a través de una declaración

3. ¿Qué significa tipificar un símbolo, definirle un tipo?

El tipo permite asignarle características a un símbolo. Por ejemplo, que tipo de operaciones se pueden realizar con él, cómo se representa, cuánto espacio hay que asignarle en memoria RAM

4. ¿Qué caracteriza a un lenguaje de bajo nivel?

Su principal característica es que sus instrucciones permiten interactuar directamente con el hardware y esto hace que se puedan enfocar en la optimización del hardware, más que en la facilidad de programación.

5. ¿Qué es un ámbito de declaración?

Es un bloque acotado sintácticamente dentro del cual los símbolos allí declarados tienen vigencia

6. ¿Por qué se dice que la generación de código es libre del contexto?

Porque se realiza considerando las características del hardware, por lo que se utiliza lenguaje ensamblador. y no depende de la sintaxis de un lenguaje de alto nivel.

7. ¿En qué parte del proceso de compilación se lleva a cabo la optimización?

En la fase de síntesis, después de la generación de código intermedio.

8. ¿Por qué la optimización es dependiente del contexto?

Porque puede ser a nivel de software y es dependiente de las características del lenguaje de alto nivel.

9. Dar un ejemplo donde se use folding.

X: = 2+3+A+B
USANDO FOLDING
X:= 5+A+B

10. Definir un bloque básico.

Es aquel conjunto de sentencias que se encuentran acotadas sintácticamente por instrucciones de inicio y de fin.

11. Dadas las reglas de producción: S -> +SS| *SS| a , reconocer la cadena: + * aaa

S → +SS
S → +*SSS
S → +*aaa

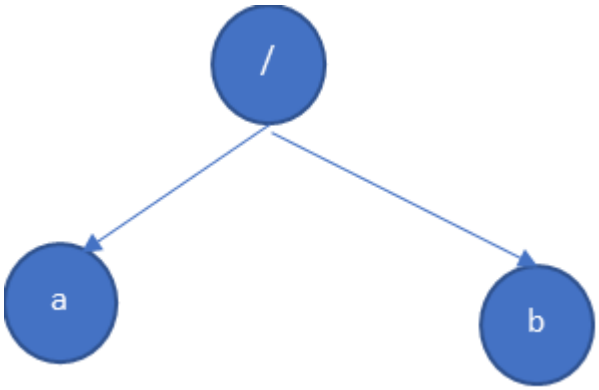
B Ejercicios Propuestos y Resueltos

1. Crear el árbol de expresión, para las siguientes expresiones:

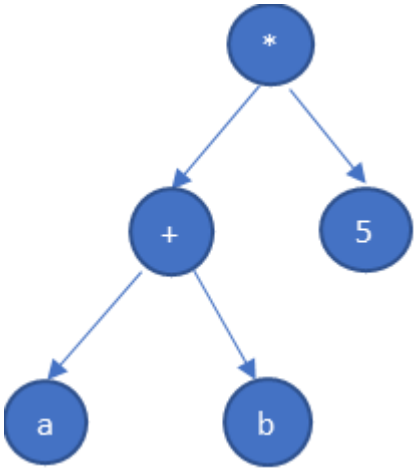
a) Expresión: a/b
b) Expresión: (a + b) * 5
c) Expresión: (3 + 4) * (2 + 5)
d) Expresión: [(a+b)*(c-d)]

SOLUCIÓN

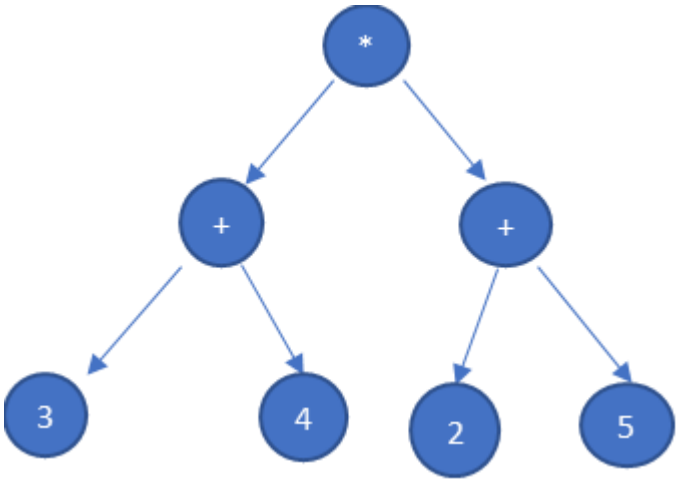
a) Expresión: a/b



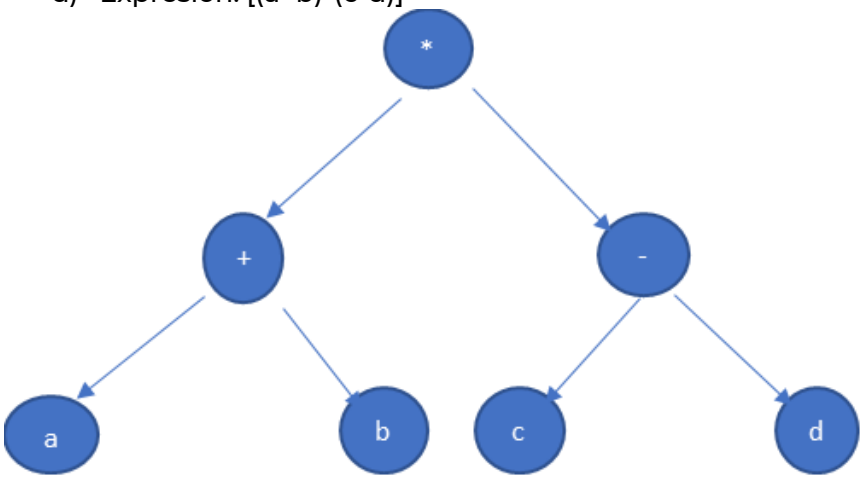
b) Expresión: (a + b) * 5



c) Expresión: (3 + 4) * (2 + 5)



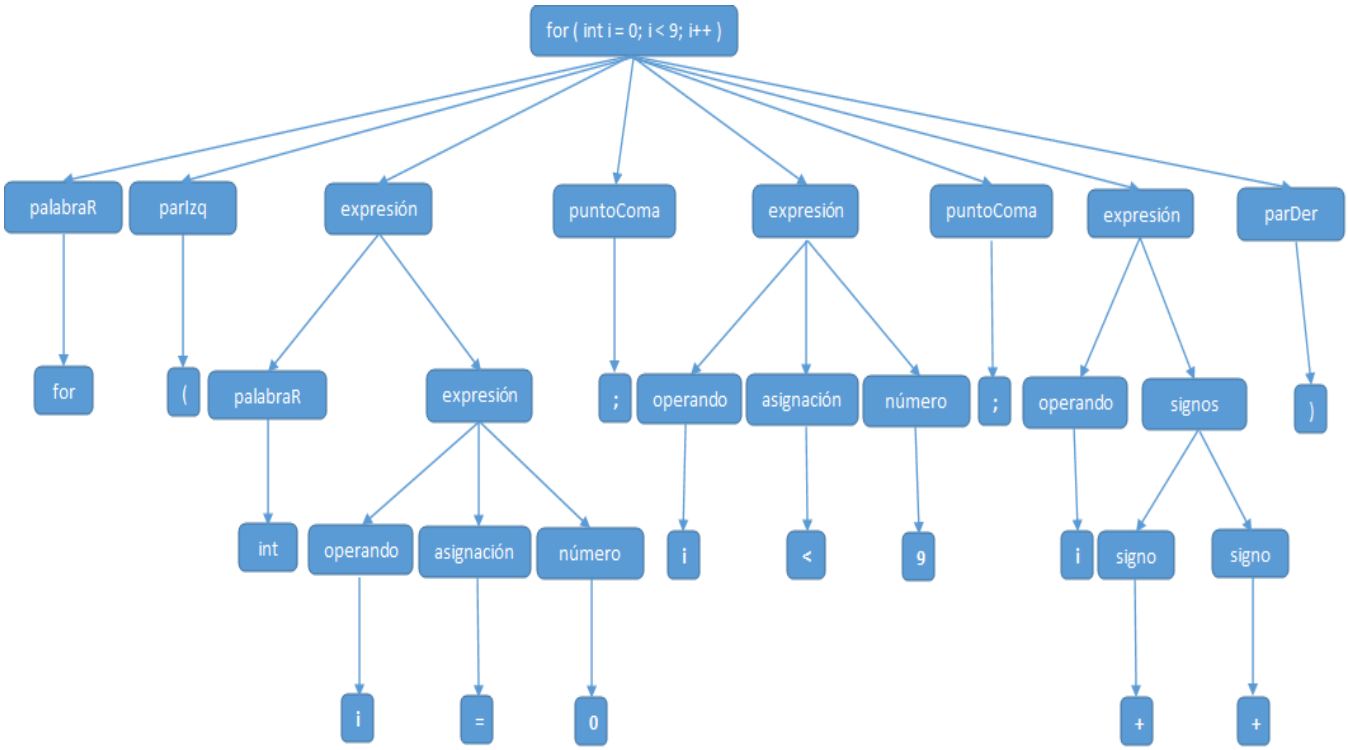
d) Expresión: [(a+b)*(c-d)]



2. Crear el árbol de expresión para el reconocimiento sintáctico de la expresión:

FOR (int i = 0; i < 9; i++)
PalabraR ---> palabra reservada
ParIz ----> paréntesis izquierdo
ParDer -----> paréntesis derecho

SOLUCIÓN

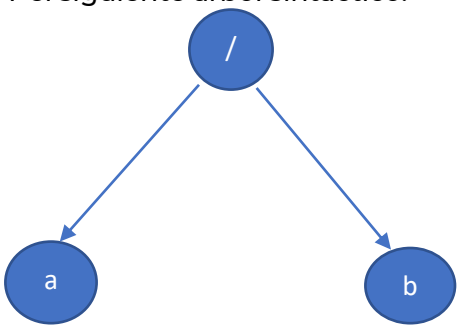


Comprobaciones de tipos en expresiones.

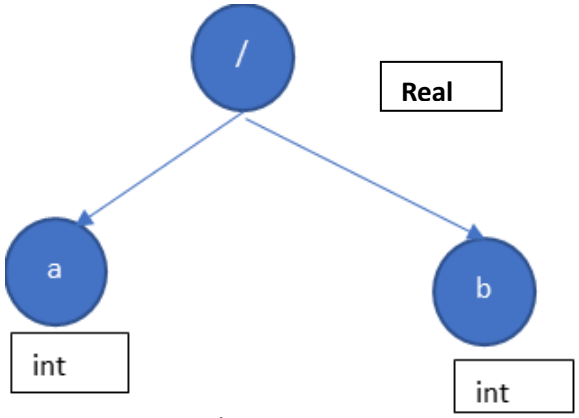
a) Verificar la coherencia semántica, mediante la comprobación de tipos con un árbol de expresiones, dado:

Var a,b: int

Y el siguiente árbol sintáctico:



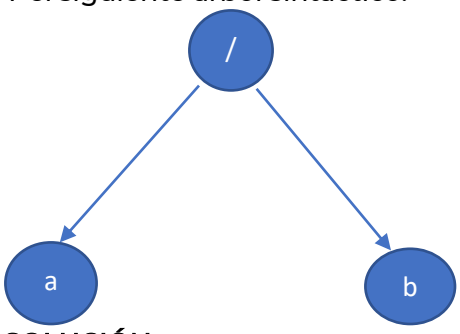
SOLUCIÓN



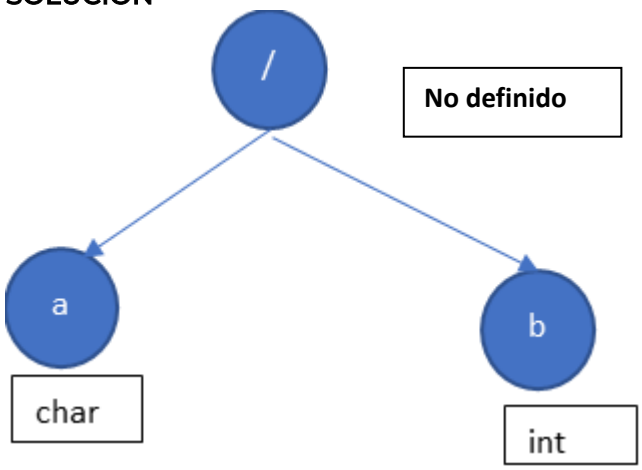
Tendríamos int/int lo cual es válido de acuerdo con el sistema de tipos que indica que dos enteros son compatibles con respecto al operador de división.

b) Verificar la coherencia semántica, mediante la comprobación de tipos con un árbol de expresiones, dado:

Var a:char
b: int
Y el siguiente árbol sintáctico:



SOLUCIÓN



Tendríamos char / int lo cual NO es válido de acuerdo con el sistema de tipos que indica que un char no es compatibles con un entero con respecto al operador de división.

4.- Esquema de traducción

GRAMÁTICA A USAR

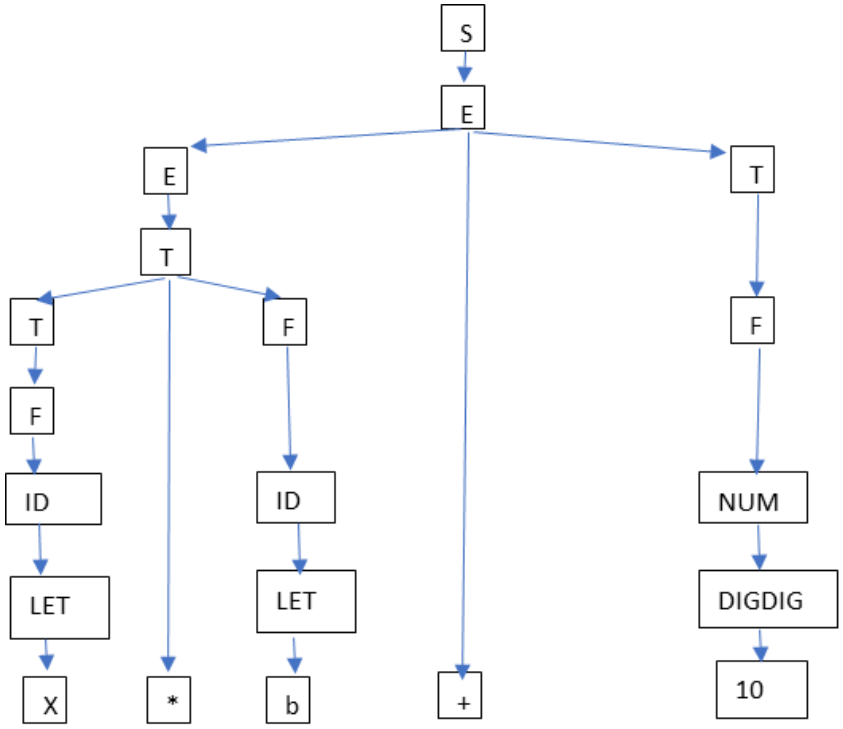
S ---> E
E ---> E+T|E-T|T
T ----> T*F|T/F|F
F ---> ID|NUM

ID ---> LETRA(LETRA|DIGITO)*
NUM ---> DIGITO(DIGITO)*
LETRA ----> a..z
DIGITO ----> 0..9

Dada la gramática anterior, generar el esquema de traducción para reconocer las expresiones:

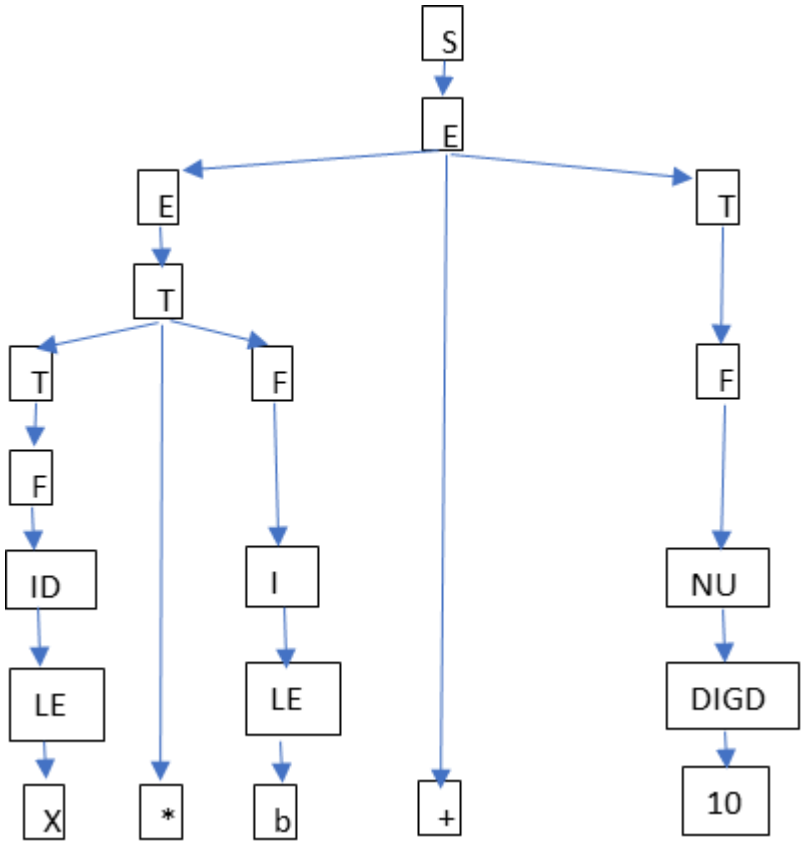
a) Expresión: x*b+10;

SOLUCIÓN



b) Expresión: 4+6/2;

SOLUCIÓN



C Recursos Electrónicos de Apoyo

c1 Vídeo explicativo de la información a adquirir en la unidad temática

c2 Presentación electrónica para ejemplificar la creación de árboles de expresiones a partir de un árbol sintáctico, así como la comprobación de tipos.

c3 Video con proceso de solución de ejercicios que contenga información de apoyo a los subtemas del 1.4 al 1.7

c4 Apoyo bibliográfico para la unidad.

2. Tema 2 Generación de código intermedio.

En esta etapa, el compilador toma el código fuente escrito por el usuario y lo transforma en una secuencia de instrucciones que se pueden ejecutar por una computadora.

La generación de código intermedio en un compilador consiste en una parte crucial del proceso de compilación. Esto implica la traducción de las instrucciones escritas en un lenguaje de programación fuente a un código intermedio, que luego se utiliza para generar el código objeto.

Durante la generación de código intermedio, los errores de sintaxis y semántica no corregidos previamente pueden ser detectados, permitiendo al compilador generar un código óptimo para la plataforma de destino.

El código intermedio se compone de instrucciones abstractas simples, en lugar de instrucciones de una CPU específica. Estas instrucciones abstractas se pueden representar con una representación de árbol abstracto (AST) o una representación sintáctica abstracta (SSA).

Estas representaciones permiten que el compilador inspeccione e interprete el código fuente, detecte errores semánticos, optimice el código para la plataforma de destino y genere código objeto final.

La verificación y corrección de errores semánticos, incluye cosas como la verificación de tipos, la verificación de compatibilidad de argumentos y otras tareas de validación de lógica. Si el compilador detecta un error semántico, generará un mensaje de error útil para el usuario que incluirá información detallada sobre el error y cómo solucionarlo. Una vez que el compilador detecta y corrige los errores semánticos, generará el código intermedio apropiado para ser ejecutado por una computadora. Esto generalmente se realiza mediante la generación de código de máquina, un lenguaje de bajo nivel que es entendido directamente por la computadora.

El código de máquina se puede generar directamente a partir del código fuente escrito por el usuario, o el compilador puede generar un código intermediario más abstracto que luego se traduzca al código de máquina.

2.1. Notaciones.

Durante la traducción a notaciones simples, el compilador puede aplicar reglas semánticas para verificar la semántica del código.

Esto incluiría verificar que los ciclos se utilicen correctamente, que se produzcan los resultados esperados para las funciones y que se cumplan los requisitos para el uso de variables. En ambos casos, el compilador puede generar un informe de errores para informar al usuario sobre los errores semánticos detectados.

Este informe de errores puede incluir información como el lugar exacto del código donde se detectó el error, el tipo de error y la descripción de este. Esta información ayuda al usuario a corregir los errores de manera rápida y eficiente.

En resumen, los compiladores pueden detectar errores semánticos a través de la aplicación de reglas semánticas durante la generación de código intermedio o la

traducción a notaciones simples. Esto se logra mediante la verificación de tipos de datos, nombres de símbolos, uso de ciclos, resultados de funciones y requisitos de variables. Si se detectan errores, el compilador genera un informe de errores para informar al usuario y ayudarlo a corregir los errores de manera rápida y eficiente.

2.1.1. Prefija.

Los errores semánticos en un compilador se pueden manejar usando una variedad de herramientas. Una de ellas es la notación prefija. Esta herramienta analiza los símbolos y palabras de un programa y los reordena en una estructura jerárquica usando una notación prefija.

Esto permite al compilador entender el código como si fuera un árbol de sintaxis y significa que el compilador puede detectar errores semánticos en el programa, como variables no declaradas o errores de lógica en los cálculos. Esta herramienta es muy útil para detectar cualquier error en el flujo lógico del programa. Otra herramienta que se puede usar para detectar errores semánticos es la lógica formal. Esta herramienta analiza el programa usando un lenguaje lógico para detectar errores semánticos. Esto permite al compilador entender el programa a un nivel más profundo, lo que ayuda a identificar errores sutiles en la lógica, como variables no declaradas o errores de lógica en los cálculos. Esta herramienta es muy útil para detectar cualquier error lógico en el programa.

Finalmente, se puede usar la depuración para detectar errores semánticos. Esta herramienta se utiliza para detectar errores en tiempo de ejecución del programa. Esto significa que el compilador ejecuta el programa paso a paso y detecta errores semánticos, como variables no declaradas o errores de lógica en los cálculos.

La notación prefija es una notación para expresiones aritméticas en la que los operadores preceden a los operandos. Esta notación es similar a la notación posfija, excepto que los operadores aparecen antes de los operandos en lugar de después.

Por ejemplo

la expresión infija 2 + 3 se escribiría como

+ 2 3 en notación prefija.

Esta notación es útil para expresiones más complejas que contienen varios operadores, ya que los operadores se pueden aplicar en el orden correcto sin necesidad de usar paréntesis.

Por ejemplo

la expresión infija 2 + 3 × 4 se escribiría como

+ 2 × 3 4 en notación prefija.

Esta notación también se conoce como notación polaca anterior y se utiliza comúnmente en lenguajes de programación como APL, y también para la evaluación de expresiones aritméticas en árboles de expresión.

La notación prefija es una de las tres notaciones principales para representar expresiones aritméticas. Las otras dos son la notación infija (en la que los operadores se escriben entre sus operandos) y la notación posfija (en la que los operadores se escriben después de sus operandos). En general, la notación prefija es menos intuitiva

que la notación infija, ya que los operadores aparecen antes de sus operandos correspondientes. Esta notación también es más fácil de evaluar para el ordenador, ya que los operadores se aplican en el orden correcto sin necesidad de usar paréntesis.

Sus características principales son:

- Los operadores conservan el mismo orden que la notación infija equivalente.
- No requiere de paréntesis para indicar el orden de precedencia de operadores ya que él es una operación.
- Se evalúa de izquierda a derecha hasta que encuentra el primer operador seguido inmediatamente de un par de operando.
- Se evalúa la expresión binaria y el resultado se cambia como un nuevo operando. Se repite hasta que nos quede un solo resultado.
- El orden es operador, primer operando, segundo operando.

2.1.2. Infija.

Es una expresión aritmética escrita de manera tal que cada operación se escribe con los dos operandos al lado del operador.

Ejemplo: 5 + 3

En esta notación, los dos números se encuentran al lado del operador "+", lo cual significa que se quiere sumar 5 más 3.

Otro ejemplo: 8 × 5

En este caso, los dos números se encuentran al lado del operador "×", lo cual significa que se quiere multiplicar 8 por 5.

Esta notación se conoce como notación infija, ya que los operadores se encuentran entre los operandos.

La notación infija se contrasta con la notación prefija o notación polaca inversa, en la cual los operadores se encuentran antes de los operandos.

Por ejemplo, en la notación prefija, la operación anterior se escribiría como "× 8 5". La notación infija es la más comúnmente usada en matemáticas, ya que es la más intuitiva para la mayoría de los matemáticos.

2.1.3. Postfija.

También hay una notación posfija, en la cual los operadores se encuentran después de los operandos. Por ejemplo, en la notación posfija, la operación anterior se escribiría como "8 5 ×".

Estas notaciones se utilizan en computación para facilitar el procesamiento de los operadores y operandos.

Es una notación para representar expresiones aritméticas en la que los operandos preceden a los operadores.

Los operandos se escriben antes de los operadores, en lugar de entre paréntesis como se suele hacer en la notación infija.

Esto significa que los operadores aparecen después de sus operandos correspondientes.

Por ejemplo

la expresión infija 2 + 3 se escribiría como

2 3 + en notación posfija.

Esta notación es útil para expresiones más complejas que contienen varios operadores, ya que los operadores se pueden aplicar en el orden correcto sin necesidad de usar paréntesis.

Por ejemplo

la expresión infija 2 + 3 × 4 se escribiría como

2 3 4 × + en notación posfija.

Esta notación también se conoce como notación polaca inversa o notación polaca posterior. Se utiliza comúnmente en lenguajes de programación como Lisp, y también se utiliza para la evaluación de expresiones aritméticas en árboles de expresión.

En general, la notación posfija se considera más fácil de usar que la notación infija, ya que los operadores no tienen que estar entre los paréntesis. Esta notación también es más fácil de evaluar para la computadora, ya que los operadores se aplican en el orden correcto sin necesidad de usar paréntesis.

2.2. Representaciones de código Intermedio.

Representación de código intermedio (CIR) es una representación intermedia del código objeto generado por un compilador.

El código intermedio está compuesto por instrucciones simples que representan las instrucciones de un lenguaje de programación de alto nivel.

Estas instrucciones se utilizan para representar un programa de alto nivel en forma de una secuencia de operaciones elementales. Puede ser utilizada como una forma de optimizar el código antes de generar el código objeto final.

El CIR también puede ser utilizado para la depuración de errores de lenguaje de programación de alto nivel. El compilador puede generar una representación del código de alto nivel, que incluye el número de línea y el código de alto nivel original. Esto permite que los errores de lenguaje de alto nivel sean detectados y corregidos con mayor facilidad.

La representación de código intermedio se utiliza comúnmente en compiladores de lenguajes de alto nivel como C, C++ y Java.

El compilador primero traduce el código de alto nivel a una representación en lenguaje intermedio, luego genera el código objeto a partir de esta representación. Esto permite que el compilador optimice el código antes de generar el código objeto.

Por ejemplo, el compilador puede reordenar instrucciones para aumentar el rendimiento, o eliminar instrucciones redundantes.

¨La fase de generación de código intermedio se encarga de traducir el programa a una representación de código intermedio para que luego ésta pueda ser transformada en las fases subsiguientes a un código ejecutable en una determinada arquitectura física¨ (Vélez Reyes, Javier).

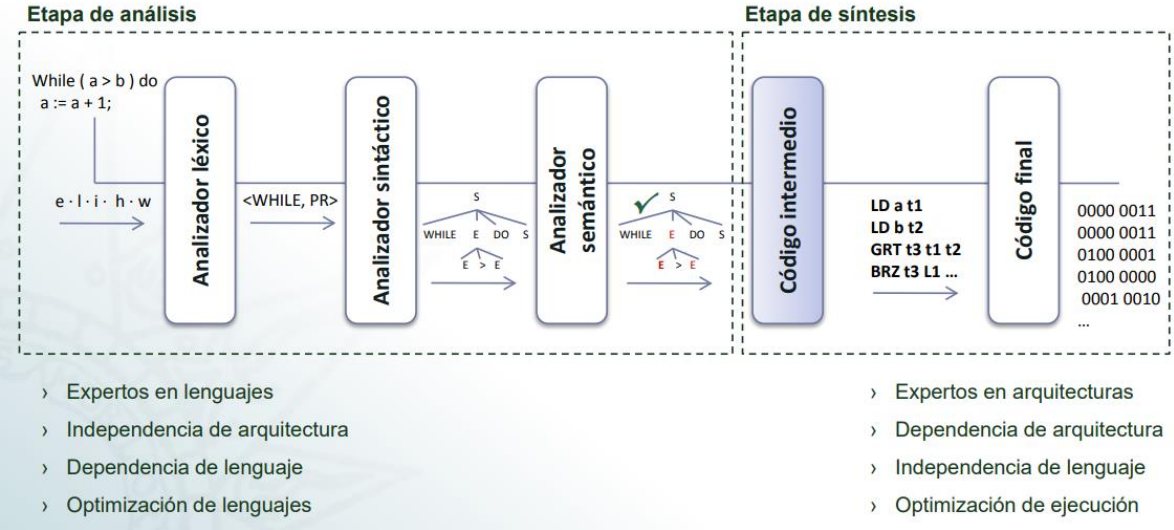


Figura 1. Vélez Reyes, Javier (2010). Esquema de Generación de Código Intermedio (Diseño). Procesadores de Lenguajes. Ingeniería Técnica Superior de Ingeniería Informática. Departamento de Lenguajes y Programas Informáticos. Editorial UNED.

2.2.1. Notación Polaca.

La notación polaca es una forma de notación matemática que permite expresar operaciones matemáticas sin usar paréntesis.

En lugar de usar paréntesis para indicar la prioridad de las operaciones, los operadores se escriben antes de sus dos operandos. Esto permite una sintaxis más simple que la notación tradicional, y también hace que sea más fácil de interpretar por una computadora.

La notación polaca se basa en la expresión de operaciones binarias, como la suma, la resta, la multiplicación y la división.

Cada operación se escribe como una palabra clave, seguida de los dos operandos. Por ejemplo, la suma se escribe como "suma" seguida de los dos números a sumar.

Esta notación se puede extender a operaciones más complejas usando paréntesis, como en la notación tradicional.

La notación polaca se inventó en 1924 por el matemático polaco Jan Łukasiewicz. Se ha convertido en un estándar en programación de computadoras, donde se utiliza para escribir algoritmos y expresiones matemáticas.

Esta notación es especialmente útil para la ejecución de operaciones aritméticas por parte de una computadora, ya que elimina la necesidad de usar paréntesis para indicar la prioridad de los operadores. Esto permite un código más compacto y legible.

La notación polaca es un tipo de notación matemática en la que se escriben los operadores entre los operandos, en lugar de antes y después de ellos. Para compilarla, una forma de hacerlo es recorrer la cadena de entrada de izquierda a derecha.

Algoritmo:

- Para cada token encontrado:

  - si es un operador, tomar los últimos dos elementos de la pila de operandos y realizar la operación correspondiente. Después de la operación, el resultado se coloca en la pila de operandos.

  - si es un operando, se coloca en la pila.

Al final, la pila debería contener un solo elemento, el resultado de la operación.

Si la pila contiene más de un elemento, esto significa que el número de operadores no fue el correcto para el número de operandos.

Por ejemplo, para compilar la expresión "2 + 3 * 5", el compilador debería realizar los siguientes pasos:

1. Leer el token "2": se coloca en la pila de operandos.

2. Leer el token "+": tomar los últimos dos elementos de la pila (2 y 3) y realizar la operación de suma. El resultado se coloca en la pila.

3. Leer el token "*": tomar los últimos dos elementos de la pila (5 y 7) y realizar la operación de multiplicación. El resultado (35) se coloca en la pila.

4. La pila contiene un solo elemento, el resultado de la operación (35). El compilador ha terminado de compilar la expresión.

En general, la notación polaca es una forma útil de escribir expresiones matemáticas para que sean fácilmente compiladas por un compilador. Esto se debe a que no hay ambigüedad en la forma en que se escriben las operaciones. Además, al leer el código fuente de izquierda a derecha, el compilador puede realizar la operación correspondiente sin tener que preocuparse por el orden de las operaciones. Esto hace que sea más fácil para el compilador procesar la expresión y generar el código de salida correcto.

Algoritmo para la notación polaca generada a través de un autómata con pila.

- Representar la expresión en forma de árbol sintáctico.
- Recorrer el árbol en postorden

Ejemplo: a + b * c-d

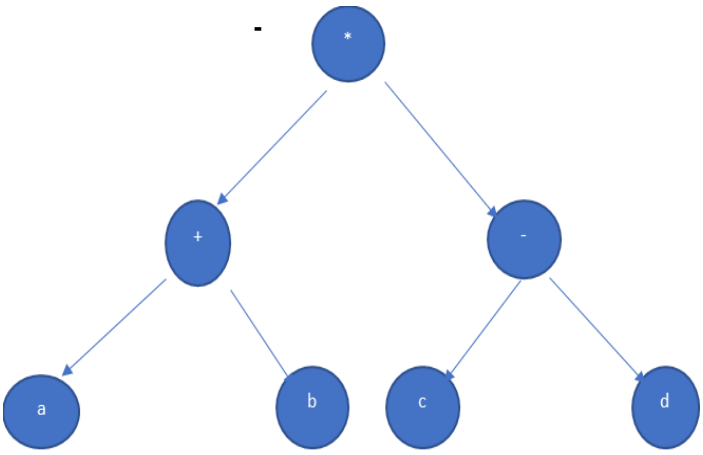


*Figura 2. Árbol de expresión para la expresión a + b * c-d. Código a b c * + d-*

Ventajas y desventajas de la notación polaca

- Generación de código: simple, no utiliza registros.
- Optimización: es difícil de reordenar ya que hay que considerar el contenido de la pila.
- Interpretación rápida: es muy fácil de interpretar ya que solo necesita una pila.
- Transportable: si, ya que todos los procesadores implementan una pila.

2.2.2. Código P.

El código P es un lenguaje de programación creado por el equipo de desarrolladores de compiladores de la Universidad de Stanford.

Está diseñado para simplificar el diseño de compiladores y comprender cómo funcionan los lenguajes de programación modernos. Está escrito en lenguaje C y tiene una serie de características que lo hacen adecuado como lenguaje base para compiladores.

Estas características incluyen una sintaxis sencilla, una sintaxis estándar, una estructura de datos avanzada, una biblioteca de funciones y una arquitectura de compilador de varias etapas.

El compilador de código P es un compilador de código fuente que se usa para convertir un programa escrito en código P a lenguaje de máquina. Esta herramienta se usa para crear programas eficientes que se ejecutan en diferentes plataformas y para optimizar los programas para que se ejecuten de manera más rápida y eficiente. Así también se usa para detectar los errores en el programa, al proporciona información sobre cada línea de código para ayudar al programador a solucionar errores de forma más rápida.

El código intermedio P (también conocido como P-code) es un lenguaje de programación de bajo nivel diseñado para ser fácilmente convertible a código de máquina. Fue desarrollado originalmente para su uso en el sistema operativo UCSD Pascal en la década de 1970, pero también se ha utilizado en otros entornos de programación.

El código intermedio P es un conjunto de instrucciones similares a las de una CPU, pero que están diseñadas para ser independientes de la arquitectura de la CPU

subyacente. Esto significa que el código intermedio P puede ser traducido a código de máquina para cualquier plataforma sin necesidad de reescribir todo el código.

El código intermedio P también es una forma útil de optimizar el código antes de su compilación final. Los compiladores de lenguajes de alto nivel a menudo generan código intermedio P antes de traducirlo a código de máquina. Al hacer esto, pueden realizar optimizaciones adicionales y transformaciones de código para mejorar el rendimiento del programa.

En resumen, el código intermedio P es un lenguaje de bajo nivel que se utiliza como una forma intermedia de representar el código fuente antes de la generación del código de máquina final. Es independiente de la plataforma y permite una mayor optimización del código antes de su compilación final.

2.2.3. Triplos.

Una representación intermedia de tripletas es una estructura de datos que contiene un conjunto de tripletas (sujeto, predicado, objeto) que describen relaciones entre conceptos. Estas tripletas están diseñadas para ser usadas como una representación intermedia para almacenar información en una base de datos y pueden ser usadas para representar relaciones complejas entre objetos. Por ejemplo, una tripleta puede describir una relación entre una persona y un lugar, como: (persona, vive en, lugar). Estas tripletas son muy útiles para modelar relaciones complejas entre conceptos y pueden ser usadas para almacenar información en una base de datos.

Un compilador puede representar tripletas en una forma intermedia de lenguaje de máquina. Esto significa que el compilador analizará el código fuente y lo convertirá en una secuencia de tripletas. Estas tripletas contienen instrucciones simples para la computadora para ejecutar.

Las tripletas se componen de un operador, un operando izquierdo y un operando derecho. El operador es la instrucción, el operando izquierdo es el argumento de la instrucción y el operando derecho es el destino.

1. Operador: el tipo de operación que se realiza, como suma, resta, multiplicación, división, asignación, entre otros.
2. Operando 1: el primer valor utilizado en la operación.
3. Operando 2: el segundo valor utilizado en la operación.

Por ejemplo, una tripleta podría ser: **sumar, x, y;** lo que significa que el valor de la variable ¨x¨ se sumará con el valor de la variable ¨y¨ y el resultado se almacenará en la variable ¨y¨.

El compilador también puede simplificar este proceso para hacer que sea más eficiente, como reemplazar los nombres de variables por números o agrupar instrucciones para reducir la cantidad de tripletas necesarias.

Las tripletas pueden utilizarse para crear una tabla de símbolos para mantener un seguimiento de las variables y los valores asignados a ellas. Esta tabla es útil para el compilador para realizar tareas como la verificación de tipos y generar código para la máquina.

Esta tabla contendrá tripletas con los nombres de las variables, sus tipos y sus valores. Esto le permitirá al compilador seguir el flujo de datos a través del programa y verificar que los tipos de datos sean correctos.

Las tripletas también son utilizadas por el compilador para generar código en lenguaje de máquina. Esto significa que el compilador convertirá las tripletas en instrucciones máquina que puedan ejecutarse por la computadora.

Estas instrucciones son diferentes para cada tipo de computadora y se pueden crear un conjunto específico para cada computadora. Esto significa que el compilador tendrá que generar el código para cada computadora específica en la que se desea ejecutar el programa.

En la compilación, una tripleta se representa como un conjunto de instrucciones de tres direcciones, una para cada uno de los tres componentes de la tripleta. Estas instrucciones están diseñadas para realizar operaciones con los tres componentes de la tripleta para producir los resultados deseados. Por ejemplo, una tripleta que represente una suma se representará como un conjunto de instrucciones que tomen los dos operandos, los sumen y guarden el resultado en una ubicación de memoria. Estas instrucciones se llaman instrucciones de tripleta y se usan para representar cualquier cosa que un programador desee hacer con los operandos. En algunos lenguajes de programación, como C, el compilador generará código de tripleta para representar cada tripleta. Esto significa que cada instrucción se traduce a un conjunto de instrucciones de tripleta para realizar la misma operación. Esto ayuda a simplificar la compilación y hace que el código generado sea más eficiente.

Ejemplo Triplos

<operador> <operando1> <operando2>

Ejemplo: W * X + (Y + Z)

1. (*, W, X)

2. (+, Y, Z)

3. (+, (1), (2))

Control de flujo:

IF X>Y

THEN Z=X

ELSE Z=Y+1

1. (>, X, Y)

2. (Saltar si falso, (1), (5))

3.(=, Z, X)

4. (Saltar,, (7))

5. (+, Y, 1)

6. (=, Z, (5))

7.- RET

El resultado se asocia al número de tripleta, cada tripleta se enumera con un número consecutivo. Al hacer referencia a los números de cada tripleta se encierran en paréntesis, para diferenciarlos de un número que está siendo usado como operando.

Como podemos ver, una tripleta es un conjunto de tres elementos que describen una instrucción de un programa. Estas tripletas se usan para generar código intermedio, que es un lenguaje que usan los lenguajes de programación para comunicarse entre sí. Esto significa que el código intermedio se usa para traducir el código fuente escrito por el programador en un lenguaje de programación particular a un lenguaje de programación diferente. El código intermedio se compone de instrucciones de lenguaje de máquina simples, como instrucciones de suma, resta, multiplicación y división. Estas instrucciones se expresan como tripletas, que contienen los operandos, las instrucciones y el resultado.

Las tripletas se usan para generar el código intermedio, que luego se compila para generar el código de lenguaje de máquina.

El uso de tripletas para generar código intermedio permite a los programadores escribir código fuente en cualquier lenguaje de programación y que el compilador lo traduzca directamente a lenguaje de máquina. Esto hace que la codificación de programas sea mucho más fácil y rápida. Estas tripletas se usan para generar código intermedio, que es un lenguaje que usan los lenguajes de programación para comunicarse entre sí.

Una tripleta en la compilación es una estructura de datos que generalmente consta de tres elementos: el operador, el primer operando y el segundo operando. Para generar una tripleta a partir de una expresión simple como "A + B", podrías utilizar una representación de árbol sintáctico para organizar la información de manera más clara.

Aquí tenemos un ejemplo de cómo podríamos generar una tripleta desde una expresión "A + B" en Python:

```
class Tripleta:
    def __init__(self, operador, operando1, operando2):
        self.operador = operador
        self.operando1 = operando1
        self.operando2 = operando2

def generar_tripleta(expresion):
    operadores = ['+', '-', '*', '/']

    for operador in operadores:
        if operador in expresion:
            operando1, operando2 = expresion.split(operador)
            return Tripleta(operador, operando1.strip(), operando2.strip())

# Ejemplo de uso
expresion = "A + B"
tripleta = generar_tripleta(expresion)
print("Operador:", tripleta.operador)
print("Operando1:", tripleta.operando1)
print("Operando2:", tripleta.operando2)
```

En este ejemplo, se ha definido una clase Tripleta para representar la tripleta resultante y una función generar_tripleta que toma una expresión y busca uno de los operadores permitidos ('+', '-', '*', '/') en ella. Luego, se divide la expresión en dos operandos y se crea una instancia de la clase Tripleta con el operador y los operandos encontrados.

Hay que tener en cuenta que este ejemplo es muy simple y asume que la expresión contiene un solo operador binario. Para expresiones más complejas con varios operadores, paréntesis y precedencia de operadores, se necesita una implementación más sofisticada que analice la expresión de manera más completa.

2.2.4. Cuádruplos.

Un cuádruplo es una forma de representar información en forma de estructuras de datos. Está formado por cuatro elementos, conocidos como "componentes", que se utilizan para describir una relación entre dos objetos.

Esta representación intermedia de cuádruplos es una forma sencilla de almacenar y transmitir información relacional, lo que la hace ideal para aplicaciones como bases de datos, sistemas de recomendación y análisis de datos. A diferencia de otras formas de representación, como los diagramas de flujo o los árboles de decisión, los cuádruplos permiten representar relaciones entre objetos de manera clara y concisa.

En la compilación, un cuádruplo es una representación intermedia de una instrucción de programa que contiene cuatro elementos. Estos cuatro elementos son el operador, los dos operandos y el resultado.

Esta representación intermedia se usa para ayudar a un compilador a generar código objeto o código de máquina a partir de un programa fuente. Los cuádruplos se usan principalmente para representar instrucciones de asignación, ciclos, llamadas a funciones, condicionales y saltos a direcciones específicas.

Esta representación intermedia también se utiliza para proporcionar un análisis estático del programa fuente.

Los cuádruplos están construidos a partir de una instrucción de programa y contienen los siguientes elementos:

1. Operador: Es el símbolo utilizado para representar la operación que se realizará.

2. Operandos: Estos son los valores de entrada o variables implicadas en la operación.

3. Resultado: Esto es el valor que producirá la operación.

4. Dirección: Esto es la ubicación en la memoria donde se encuentra el resultado de la operación.

Los cuádruplos se usan para ayudar al compilador a construir el árbol de sintaxis abstracto (AST) de un programa. El AST es una representación abstracta de un programa fuente. Los cuádruplos se usan para proporcionar al compilador información sobre la estructura del programa y la relación entre los diferentes bloques de código. Esta información ayuda al compilador a generar código objeto o código de máquina de manera eficiente y con una menor cantidad de errores. Los cuádruplos también se usan para ayudar al compilador a realizar análisis estáticos del programa fuente.

Esto le permite al compilador detectar errores en tiempo de compilación antes de que el código se ejecute, ayuda a mejorar la calidad del código y reduce el tiempo de depuración.

Un cuádruplo es una estructura de datos compuesta por cuatro elementos: operador, dos operandos y un resultado.

Los cuádruplos se usan para representar instrucciones en lenguajes de programación.

Por ejemplo, la instrucción "x = y + z" se puede representar en un cuádruplo como (+, y, z, x).

El primer elemento es el operador (+) y los siguientes elementos son los operandos (y, z). El último elemento es el resultado (x). El cuádruplo indica que se debe sumar los valores de y y z, y luego asignar el resultado a la variable x.

Esta representación intermedia es una forma de codificar instrucciones de manera compacta y clara.

Ejemplo:

<operador> <operando1> <operando 2> <destino>

(A+B)*(C+D)-E

1. (+, A, B, T1)

2. (+, C, D, T2)

3. (*, T1, T2, T3)

4. (-, T3, E, T4)

2.3. Esquema de generación.

Un esquema de generación a partir de una gramática consiste en un conjunto de reglas que describen cómo se combinan los elementos léxicos para formar palabras, frases y oraciones.

Estas reglas se basan en la sintaxis de un idioma dado, lo que significa que definen la forma en que se deben combinar los elementos para formar una estructura gramaticalmente correcta.

Estas reglas se utilizan para generar oraciones de manera aleatoria, para ayudar a los programadores a escribir código que produzca oraciones gramaticalmente correctas.

Un esquema de generación a partir de una gramática consta de los siguientes elementos:

1. Un conjunto de símbolos léxicos: Estos símbolos son los elementos básicos de una oración, como palabras reservadas, identificadores, constantes, operadores, etc. Que conforman los tokens o unidades léxicas.

2. Un conjunto de reglas gramaticales: Estas reglas definen cómo se deben combinar los elementos léxicos para formar frases y oraciones gramaticalmente correctas.

3. Un conjunto de reglas semánticas: Estas reglas definen el significado de una frase u oración.

4. Un conjunto de herramientas de generación: Estas herramientas permiten a los programadores generar oraciones de manera aleatoria a partir de un esquema de generación.

Para generar cadenas de texto a partir de una gramática, se puede seguir el siguiente esquema:

a) Definir una gramática formal: La gramática formal debe especificar las reglas para construir cadenas de texto. Para ello, se pueden definir símbolos terminales y no terminales, producciones, reglas de derivación, entre otros elementos.
b) Identificar el símbolo inicial: El símbolo inicial es el punto de partida para la generación de cadenas de texto. En general, se utiliza un símbolo no terminal como símbolo inicial.
c) Aplicar las reglas de derivación: Las reglas de derivación indican cómo se pueden transformar los símbolos no terminales en otros símbolos terminales y no terminales. Para generar una cadena de texto, se puede aplicar una secuencia de reglas de derivación partiendo del símbolo inicial hasta que se obtengan únicamente símbolos terminales.
d) Repetir el paso anterior para generar más cadenas: Una vez que se ha generado una cadena de texto, se puede repetir el proceso para obtener más cadenas siguiendo los mismos pasos.

Es importante destacar que la generación de cadenas de texto a partir de una gramática puede no ser única. Es decir, una misma gramática puede generar múltiples cadenas de texto diferentes.

2.3.1. Variables y constantes.

Durante el proceso de compilación, el compilador traduce las variables y constantes de un lenguaje de programación a un formato específico para su uso en la máquina. Las variables son representaciones de información que pueden cambiar. Estos incluyen cadenas, enteros, booleanos y objetos. Esto significa que el compilador convierte los datos en lenguaje de máquina para que el procesador pueda leer la información. Las constantes son valores que no cambian. Estos se utilizan para representar información que no cambia, como números, caracteres, cadenas y objetos. Además, se asignan códigos de bytes específicos a cada constante para permitir una referencia rápida a la información.

El esquema de traducción de variables y constantes en un compilador consiste en un proceso de conversión de los nombres y valores de variables y constantes en código de máquina. Esta conversión se realiza a través de una tabla de símbolos que contiene información acerca de los nombres y valores de las variables y constantes. Esta tabla se actualiza durante el proceso de compilación para reflejar los nombres y valores de variables y constantes. El compilador también realiza optimizaciones como la sustitución de llamadas de función por llamadas directas al código de máquina para mejorar el rendimiento de la aplicación. El proceso de traducción también incluye el análisis de la sintaxis para identificar errores de programación, como la asignación de un valor incorrecto a una variable o la inclusión de una constante que no está definida. Si el compilador detecta un error, emitirá un mensaje de error y detendrá la compilación.

El proceso de código intermedio para variables y constantes es el proceso de traducción de instrucciones de lenguaje de alto nivel en instrucciones de lenguaje de bajo nivel. Se realiza mediante una serie de pasos, comenzando con la identificación de variables y constantes. Una vez que se hayan identificado estas variables y constantes, el siguiente paso es asignarles una dirección de memoria. Esto permite que el compilador sepa dónde buscar los datos cuando se ejecute el código. A continuación, el compilador comenzará a generar instrucciones de código de bajo nivel para cada línea de código de alto nivel. Estas instrucciones traducirán la lógica del programa en instrucciones que el procesador pueda entender.

Finalmente, se compilará el código intermedio, lo que generará un archivo ejecutable, listo para ejecutarse en el procesador.

2.3.2. Expresiones.

El proceso de traducción de expresiones requiere varios pasos.

El primer paso es analizar la expresión, lenguaje o autómata. Esto implica identificar qué elementos se incluyen y cómo están relacionados.

Por ejemplo, en un lenguaje, se podrían identificar las palabras clave, las reglas sintácticas y las estructuras de control.

Después de analizar la expresión, el siguiente paso es traducir los elementos de un lenguaje a otro. Esto puede implicar la creación de un intérprete, un compilador o una máquina virtual. Estos programas traducirán la expresión, lenguaje o autómata a un lenguaje de programación específico.

Finalmente, se pueden realizar pruebas para verificar la exactitud de la traducción. Se pueden usar herramientas de prueba para probar el código generado y asegurarse de que produce los resultados esperados. Esto ayudará a garantizar que la traducción es correcta.

Proceso:

1. El compilador comenzará leyendo la expresión de entrada y analizará su sintaxis.
2. El compilador interpretará la expresión para determinar su significado.
3. El compilador traducirá la expresión a un lenguaje de máquina.
4. El compilador generará código de máquina para ejecutar la expresión.
5. El compilador realizará pruebas para asegurarse de que el código generado es correcto.
6. El compilador creará un archivo ejecutable que contenga el código de máquina generado.
7. El compilador proporcionará información de depuración para ayudar a identificar y solucionar errores.
8. El compilador mostrará la salida generada como resultado de la ejecución de la expresión.

Rutina para la generación de código intermedio de una Expresión:

```
exp ::= exp:e1 MAS exp:e2 {:
Exp e = new Exp();
<<comprobación de tipos de e1 y e2>>
ScopeIF scope = scopeManager.getCurrentScope();
```

```
TemporalFactory tF = new TemporalFactory (scope);
IntermediateCodeBuilder cb = new ...Builder (scope)
TemporalIF temp1 = e1.getTemporal ();
TemporalIF temp2 = e2.getTemporal ();
TemporalIF temp = tF.create ();
cb.addQuadruples (e1.getCode ());
cb.addQuadruples (e2.getCode ());
cb.addQuadruple (“ADD”, temp, temp1, temp2);
e.setTemporal (temp);
e.setCode (cb.create());
RESULT = e
:}
```

2.3.3. Instrucción de asignación.

El esquema de traducción de una instrucción de asignación generalmente sigue los siguientes pasos:

1. Analizar la sintaxis de la instrucción de asignación para determinar la operación que se está realizando, así como los operandos que se están utilizando.
2. Identificar el tipo de datos de los operandos para garantizar que sean compatibles con la operación que se está realizando. Por ejemplo, si se está asignando un valor numérico a una variable, se debe asegurar que la variable sea de tipo numérico.
3. Verificar que la variable existe y tiene un tipo de dato compatible con el valor que se le asignará para generar el código que realiza la operación de asignación. Esto podría implicar la carga de valores de una ubicación de memoria y la asignación de esos valores a otra ubicación de memoria. Por ejemplo, para asignar el valor 5 a una variable "x", se podría generar el siguiente código en lenguaje de ensamblador: `MOV x, 5`
4. En algunos casos, se pueden requerir conversiones de tipo o verificaciones de error. Por ejemplo, si se está asignando un valor flotante a una variable entera, se podría necesitar redondear el valor antes de asignarlo y verificar que el valor flotante se encuentra dentro del rango de la variable entera.
5. Finalmente, se genera el código de salida que indica que la operación de asignación se ha completado con éxito. Por ejemplo, en lenguaje de ensamblador, se podría generar el siguiente código: `MOV eax, 0 ; Indica éxito`

El proceso puede variar dependiendo del lenguaje de programación o plataforma específicos que se estén utilizando.

Un ejemplo de aplicación de este esquema podría ser la siguiente instrucción de asignación en lenguaje C:

```
x = y + z;
```

Siguiendo el esquema, se realizarían las siguientes acciones:

1. La sintaxis de la instrucción de asignación es correcta.
2. La variable a la que se le asignará un valor es "x" y el valor es la suma de las variables "y" y "z".

3. Se verifica que la variable "x" existe y tiene un tipo de dato compatible con el resultado de la suma de "y" y "z". Suponiendo que todas estas variables son de tipo entero, la asignación es válida.
4. Se genera el código correspondiente para realizar la suma de "y" y "z" y asignar el resultado a "x":

```
Mov r1, y
add r1, z
mov x, r1
```

El código exacto puede variar dependiendo del lenguaje de programación y la arquitectura del procesador utilizado.

En términos generales, la traducción de una instrucción de asignación en la compilación involucra los siguientes pasos:

1. Análisis léxico: El compilador divide la instrucción de asignación en tokens, que son unidades básicas de sintaxis como palabras clave, identificadores, operadores y constantes. En el caso de una instrucción de asignación, los tokens pueden incluir el identificador de la variable, el operador de asignación y la expresión que se asigna a la variable.
2. Análisis sintáctico: El compilador verifica que la estructura de la instrucción de asignación sea gramaticalmente correcta y cumpla con la sintaxis del lenguaje de programación, para verificar si los tokens forman una expresión válida de acuerdo con la gramática del lenguaje de programación, por ejemplo, que tenga la forma "variable = expresión".
3. Análisis semántico: El compilador verifica que la instrucción de asignación tenga sentido en el contexto del programa y que los tipos de datos de la variable y la expresión sean compatibles. Esto incluye verificar si el tipo de la variable y la expresión que se le asigna son compatibles y si el identificador de la variable se ha declarado previamente. Por ejemplo, que una variable de tipo entero no reciba una expresión de tipo cadena.
4. Generación de código intermedio: El compilador genera una representación intermedia de la instrucción de asignación en un lenguaje de bajo nivel que es más cercano al lenguaje de la computadora. Esta representación intermedia puede ser en forma de código de tres direcciones, código de máquina virtual o código ensamblador.
5. Optimización de código: El compilador aplica diversas técnicas de optimización al código intermedio para mejorar su eficiencia y reducir su tamaño. Por ejemplo, puede eliminar código redundante o reorganizar las instrucciones para aprovechar mejor la memoria caché.
6. Generación de código objeto: El compilador traduce el código intermedio optimizado a código objeto que es código de máquina específico para la plataforma de destino, como el código de ensamblador o el código binario, que puede ejecutarse en el hardware de la computadora. Este código es la representación final de la instrucción de asignación y es lo que se ejecutará en el sistema objetivo.
7. Vinculación: Si la instrucción de asignación hace referencia a variables o funciones definidas en otros archivos de código fuente, el compilador debe vincular esos archivos para crear un archivo ejecutable completo.

Con el proceso anterior se genera una representación en código intermedio de la instrucción de asignación. Este código intermedio es independiente del lenguaje de programación y de la arquitectura de la computadora y es una representación del código fuente. Deben representar la secuencia de operaciones para que se lleve a cabo la asignación. Esto es las operaciones para que se copie el valor del operando derecho al operando izquierdo, en la ubicación de memoria que el compilador le asignó.

Por ejemplo, si consideramos la siguiente instrucción de asignación en el lenguaje de programación C:

```
x = y + z;
```

El proceso de generación de código intermedio para esta instrucción podría implicar los siguientes pasos:

1. Análisis léxico y sintáctico: el compilador verifica que la sintaxis de la instrucción sea válida.
2. Análisis semántico: el compilador verifica que las variables **y** y **z** tengan un tipo de dato compatible con la variable **x**. Y que se encuentren declaradas.
3. Generación de código intermedio: el compilador genera código intermedio que representa la asignación. Por ejemplo, podría generar código que almacene el valor de **y** en un registro, luego sume el valor de **z** en ese registro y finalmente almacene el resultado en la variable **x**.

Rutina para la generación de código intermedio para una Sentencia de asignación

```
sentenciaAsignacion ::= referencia:r IGUAL exp:e {:

SentenciaAsignacion sa = new SAsignacion ();

<<comprobación de tipos>>

ScopeIF scope = scopeManager.getCurrentScope();

TemporalFactoryIF tF = new TemporalFactory (scope);

TemporalIF eTemp = e.getTemporal ();

TemporalIF rTemp = r.getTemporal ();

TemporalIF rTempI = r.getTemporalIndex ();

TemporalIF rTempO = r.getTemporalOffset ();

TemporalIF temp = tF.create ();

IntermediateCodeBuilder cb = new ...Builder (scope);

cb.addQuadruples (e.getCode ());

cb.addQuadruples (r.getCode ());

cb.addQuadruple (“MUL”, temp, rTempI, rSize);

cb.addQuadruple (“ADD”, temp, temp, rTemp);
```

cb.addQuadruple (“ADD”, temp, temp, rTempO);

cb.addQuadruple (“STP”, temp, eTemp);

sa.setCode (cb.create()); RESULT = sa; :}

2.3.4.Instrucciones de control.

La generación de código intermedio para una instrucción de control depende del tipo de instrucción de control que se esté considerando. A continuación, se muestran algunos ejemplos de generación de código intermedio para diferentes tipos de instrucciones de control:

1. Instrucción de salto incondicional: Una instrucción de salto incondicional simplemente salta a una etiqueta específica en el código. El código intermedio generado para una instrucción de salto incondicional sería simplemente la etiqueta a la que se debe saltar. Por ejemplo, si la instrucción de salto incondicional es "goto label1;", el código intermedio generado sería "label1:".
2. Instrucción de salto condicional: Una instrucción de salto condicional salta a una etiqueta específica si se cumple una determinada condición. El código intermedio generado para una instrucción de salto condicional depende de la condición. Supongamos que tenemos la siguiente instrucción de salto condicional:

   "if (a > b) goto label1;".

   El código intermedio generado para esta instrucción podría ser:

   t1 = a > b;

   if t1 goto label1;

   En este caso, el valor booleano de la expresión "a > b" se almacena en la variable temporal "t1". Si el valor de "t1" es verdadero, se salta a la etiqueta "label1".

3. Instrucción de retorno: Una instrucción de retorno devuelve un valor de una función y sale de ella. El código intermedio generado para una instrucción de retorno depende del valor de retorno y de si la función tiene un tipo de retorno. Supongamos que tenemos la siguiente instrucción de retorno: "return a + b;". El código intermedio generado para esta instrucción podría ser:

   t1 = a + b;

   return t1;

   En este caso, el valor de la expresión "a + b" se almacena en la variable temporal "t1" y se devuelve. Si la función no tiene un tipo de retorno, la instrucción de retorno simplemente saldría de la función sin generar código intermedio adicional.

En la compilación, la generación de código intermedio para una instrucción de control, como una instrucción condicional "if" o una instrucción de bucle "while", generalmente involucra los siguientes pasos:

1. Evaluación de la expresión condicional: En primer lugar, se evalúa la expresión condicional de la instrucción de control, para determinar si se debe o no ejecutar el bloque de código asociado.
2. Generación de código para el bloque de código: Si se debe ejecutar el bloque de código, se genera el código intermedio correspondiente para ese bloque. Esto puede implicar la generación de código para una serie de instrucciones individuales, cada una de las cuales realiza una tarea específica.
3. Generación de código de salto: Después de generar el código para el bloque de código, se genera un código de salto para saltar a la siguiente instrucción después del bloque de código, o para volver al inicio del bucle en el caso de una instrucción de bucle.
4. Actualización de las direcciones de salto: Si se usaron saltos para implementar la instrucción de control, se debe actualizar la dirección de salto en la instrucción de salto para apuntar a la ubicación correcta en el código intermedio.

2.3.5. Funciones.

La generación de código intermedio es un proceso importante en la compilación de programas. A continuación, se describen los pasos generales para la generación de código intermedio para una función:

1. Análisis léxico y sintáctico: El primer paso es analizar el código fuente de la función para asegurarse de que esté bien formado. Esto se hace mediante el análisis léxico y sintáctico.
2. Análisis semántico: Una vez que se ha analizado la sintaxis, se realiza un análisis semántico para verificar que el código tenga sentido. Esto incluye la verificación de tipos, la resolución de identificadores y la detección de errores semánticos.
3. Creación del árbol de sintaxis abstracta (AST): A continuación, se crea un árbol de sintaxis abstracta (AST) que representa la estructura de la función. El AST es una estructura de datos que se utiliza para representar el código fuente en una forma más manejable y fácil de procesar.
4. Generación de código intermedio: A partir del AST, se genera el código intermedio. El código intermedio es una representación del código fuente en un nivel más bajo que es más fácil de traducir en código de máquina. Este paso incluye la generación de instrucciones de código intermedio, la asignación de registros y la gestión de la pila de memoria.
5. Optimización de código intermedio: Después de generar el código intermedio, se puede aplicar una serie de optimizaciones para mejorar su eficiencia y rendimiento. Esto incluye técnicas como la eliminación de código muerto, la propagación de constantes y la reorganización de instrucciones para minimizar la cantidad de saltos de código.
6. Generación de código de máquina: Finalmente, se traduce el código intermedio en código de máquina que puede ser ejecutado por la CPU. Este proceso incluye la asignación de direcciones de memoria y la generación de código de máquina para cada instrucción en el programa.

En resumen, la generación de código intermedio para una función es un proceso complejo que involucra varios pasos, incluyendo análisis léxico y sintáctico, análisis semántico, creación de AST, generación de código intermedio, optimización de código

intermedio y generación de código de máquina. Cada paso es esencial para crear un código de máquina eficiente y funcional a partir del código fuente original.

La generación de código intermedio es una etapa importante en la compilación de programas, que tiene lugar después del análisis sintáctico y semántico y antes de la optimización y generación de código final. Durante esta etapa, se crea un código intermedio que representa la función en un nivel más abstracto que el código fuente original, pero aún no se ha traducido completamente a código de máquina.

El proceso de generación de código intermedio implica la transformación del árbol de sintaxis abstracta (AST) generado durante el análisis sintáctico y semántico en un conjunto de instrucciones intermedias que se ajusten a una representación uniforme y fácilmente manejable. Estas instrucciones intermedias pueden ser más fáciles de analizar y optimizar que el código fuente original.

Las instrucciones intermedias se organizan típicamente en un formato de tres direcciones, en el cual cada instrucción tiene una forma "op a, b, c", donde "op" es una operación como suma, resta, multiplicación, división, etc., y "a", "b" y "c" son registros o variables temporales que se utilizan para almacenar los resultados y valores intermedios.

Un ejemplo simple de una función en C y su correspondiente código intermedio en formato de tres direcciones podría ser el siguiente:

Función en C:

int sum(int a, int b) {

return a + b;

}

python Copy code

```python
int sum(int a, int b) {
  return a + b;
}
```

Código intermedio en formato de tres direcciones:

css Copy code

```
t1 = a + b
return t1
```

Este código intermedio puede ser posteriormente optimizado y traducido a código de máquina por el compilador, para producir un ejecutable final que implementa la función de manera eficiente y correcta.

En este seudocódigo, hemos definido una estructura Cuadrupla para representar las cuádruplas y una lista cuadruplas para almacenarlas. La función agregarCuadrupla se utiliza para agregar cuádruplas a la lista.

Luego, hemos definido una función suma que realiza una suma y genera una cuádrupla correspondiente. En el ejemplo de llamada a función, asignamos valores a x y y, y luego llamamos a la función suma(x, y) para calcular la suma y almacenar el resultado en z.

Finalmente, después de ejecutar la llamada a función, imprimimos las cuádruplas generadas.

Ten en cuenta que este es un ejemplo simplificado para ilustrar el concepto de funciones y llamadas a funciones utilizando cuádruplas en un contexto de compilación. En un compilador real, las cuádruplas se utilizan para representar instrucciones intermedias más complejas, y el manejo de funciones suele ser más elaborado, incluyendo manejo de ámbito, paso de argumentos, gestión de la pila de llamadas, entre otros aspectos.

Aquí tenemos un seudocódigo que muestra cómo podríamos representar funciones y llamadas a funciones utilizando cuádruplas en un contexto de compilación:

```plaintext
# Definición de las estructuras de cuádruplas
estructura Cuadrupla:
    operador: cadena
    operando1: cadena
    operando2: cadena
    resultado: cadena

# Lista para almacenar cuádruplas
cuadruplas = []

# Función para agregar una cuádrupla a la lista
funcion agregarCuadrupla(operador, operando1, operando2, resultado):
    cuadrupla = Cuadrupla(operador, operando1, operando2, resultado)
    cuadruplas.agregar(cuadrupla)

# Definición de una función
funcion suma(a, b):
    resultado = a + b
    agregarCuadrupla('+', a, b, resultado)
    retornar resultado

# Ejemplo de llamada a función
x = 3
y = 7
z = suma(x, y)

# Imprimir cuádruplas generadas
para cada cuadrupla en cuadruplas:
    imprimir cuadrupla
```

En este seudocódigo, se ha definido una estructura Cuadrupla para representar las cuádruplas y una lista cuadruplas para almacenarlas. La función agregarCuadrupla se utiliza para agregar cuádruplas a la lista.

Luego, se ha definido una función suma que realiza una suma y genera una cuádrupla correspondiente. En el ejemplo de llamada a función, se asignan valores a ¨x ¨y ¨y¨, y luego se llama a la función suma(x, y) para calcular la suma y almacenar el resultado en z.

Finalmente, después de ejecutar la llamada a función, se imprimen las cuádruplas generadas.

Tener en cuenta que este es un ejemplo simplificado para ilustrar el concepto de funciones y llamadas a funciones utilizando cuádruplas en un contexto de compilación. En un compilador real, las cuádruplas se utilizan para representar instrucciones intermedias más complejas, y el manejo de funciones suele ser más elaborado, incluyendo manejo de ámbito, paso de argumentos, gestión de la pila de llamadas, entre otros aspectos.

2.3.6. Estructuras.

En el caso específico de una estructura en el código fuente, el compilador deberá generar código intermedio que permita representar y manipular la estructura durante la compilación. Esto implica identificar los miembros de la estructura, asignarles direcciones de memoria y generar código para acceder a ellos.

Por ejemplo, si se tiene la siguiente definición de estructura en C:

```c
struct persona {
    char nombre[20];
    int edad;
    float altura;
};
```

El compilador deberá generar código intermedio que permita representar esta estructura en la memoria y acceder a sus miembros. Esto puede involucrar la asignación de direcciones de memoria para cada miembro, como se muestra a continuación:

```python
persona:
  .nombre: resb 20   ; reserva 20 bytes para el miembro 'nombre'
  .edad: resd 1      ; reserva 4 bytes para el miembro 'edad'
  .altura: resd 1    ; reserva 4 bytes para el miembro 'altura'
```

Luego, el compilador deberá generar código para acceder a los miembros de la estructura, lo que puede implicar el uso de desplazamientos (offsets) desde la dirección base de la estructura. Por ejemplo:

```csharp
mov eax, persona    ; carga la dirección base de la estructura 'persona' en el regi
mov ebx, [eax+.nombre]  ; accede al miembro 'nombre' de la estructura usando un des
mov ecx, [eax+.edad]    ; accede al miembro 'edad' de la estructura usando un despl
fld [eax+.altura]   ; accede al miembro 'altura' de la estructura usando un desplaz
```

Estos son solo ejemplos de cómo se podría generar código intermedio para una estructura durante la compilación. El código real generado dependerá del lenguaje fuente, la arquitectura de la máquina objetivo y otros factores.

Generar cuádruplas para estructuras de control, como ciclos o bucles y condicionales, en un compilador implica mantener un seguimiento de las etiquetas de inicio y finalización de las estructuras, así como realizar saltos condicionales o incondicionales según sea necesario. A continuación, se presenta un seudocódigo básico que muestra cómo se podrían generar cuádruplas para un bucle while:

plaintext

```
# Definición de las estructuras de cuádruplas
estructura Cuadrupla:
    operador: cadena
    operando1: cadena
    operando2: cadena
    resultado: cadena

# Lista para almacenar cuádruplas
cuadruplas = []

# Función para agregar una cuádrupla a la lista
funcion agregarCuadrupla(operador, operando1, operando2, resultado):
    cuadrupla = Cuadrupla(operador, operando1, operando2, resultado)
    cuadruplas.agregar(cuadrupla)

# Inicio del bucle while
etiqueta_inicio = obtenerNuevaEtiqueta()
etiqueta_fin = obtenerNuevaEtiqueta()
```

```
agregarCuadrupla('ETIQUETA', '', '', etiqueta_inicio)

condicion = True  # Supongamos que la condición es verdadera inicialmente

mientras condicion:
    # Cuerpo del bucle
    # ...

    # Actualizar la condición y generar un salto condicional
    condicion = evaluarCondicion()  # Evaluar la condición
    si condicion es True:
        agregarCuadrupla('SALTO_CONDICIONAL', etiqueta_inicio, '', '')

# Etiqueta de finalización del bucle
agregarCuadrupla('ETIQUETA', '', '', etiqueta_fin)

# Resto del programa
# ...

# Imprimir cuádruplas generadas
para cada cuadrupla en cuadruplas:
    imprimir cuadrupla
```

En este seudocódigo, se ha definido una estructura Cuadrupla para representar las cuádruplas y una lista cuadruplas para almacenarlas. La función agregarCuadrupla se utiliza para agregar cuádruplas a la lista.

Para representar un bucle while, se han utilizado etiquetas (ETIQUETA) para marcar el inicio y el final del bucle. El bucle se ejecuta mientras la condición sea verdadera y se evalúa en cada iteración. Cuando la condición se vuelve falsa, se genera un salto condicional de regreso al inicio del bucle.

Este es un ejemplo simplificado de cómo podríamos generar cuádruplas para un bucle while en un compilador. En un compilador real, habría que considerar más detalles, como el manejo de ámbito, el control de flujo, la optimización, etc.

ACTIVIDADES DE APRENDIZAJE DE LOS TEMAS DE LA ASIGNATURA

A Recursos de Evaluación

1.- ¿Cuál es el objetivo de la fase de Generación de Código Intermedio?

Reducir al máximo el número de traducciones de código fuente a código objeto.

2.- ¿Cuál es la entrada a la Fase de Síntesis?

La representación intermedia

3.- Representar en notación postfija la expresión: a*(-b+c/d).

acd/b+*

4.- Usar tripletas para generar el código intermedio de: While a>b b:=b+1

1.- (>, a,b)

2.-(si_no,(1), (5))

3.- (+,1,b)

4.- (=,(3),b)

5.- RET

5.- ¿Cómo representamos una operación diádica mediante cuádruplas?

(operador, operando 1, operando 2, localidad de memoria para resultado)

6.- Generar las cuádruplas para la instrucción: if (a>b) then b:=b+1.

1.- (>, a,b,T1)

2.-(si_no,T1, , (5))

3.- (+,1,b,T1)

4.- (=,T1, ,b)

5.- RET

B Ejercicios Propuestos y Resueltos

1. Dada la siguiente expresión: **a+b*c**, ponerla en notación:

   a) Prefija

SOLUCIÓN

*bc+a

   b) Postfija

SOLUCIÓN

bc*a+

2. Dada la siguiente expresión:

(a+b)/(c-d) , ponerla en notación:

a) Prefija

SOLUCIÓN

/+ab-cd

b) Posfija

SOLUCIÓN

ab+cd-/

3. Dada la siguiente expresión:

[(a*b)+(c/d)], ponerla en notación:

a) Prefija

SOLUCIÓN

+*ab/cd

b) Posfija

SOLUCIÓN

ab*cd/+

4. Dada la siguiente expresión:

+ab-cd/ , ponerla en notación infija:

SOLUCIÓN

(a+b)/(c-d)

5. Pasar a CÓDIGO P, el siguiente fragmento.

While I< (a+b)

Write (Arreg[I])

I:=I+1

Endwhile

SOLUCIÓN

LAB CICLO

LOD A

LOD B

ADI

STN

LOD I

GRT

FJP SALIR

LOD ARREG[I]

WRI

LDA I

LOD I

LDC 1

ADI

STO

UJP CICLO

LAB SALIR

STP

6.- Pasar a código P, el siguiente fragmento.

z:= (a+b)/(c-d);

SOLUCIÓN

LDA Z

LOD A

LOD B

ADI

STN

LOD C

LOD D

SBI

STN

DVI

STO

STP

7.- Generar una tripleta y una cuádrupla para el siguiente segmento de código.

If a› b then

b:= b+1;

else

a:= a+1;

a) Tripleta.

SOLUCIÓN

1.- (›, a, b)

2.- (siverdadero, (1), (6))

3.- (+, 1, a)

4.- (=, (3), a)

5.- (ir_a , ,(8))

6.- (+, 1, b)

7.- (=, (6), b)

8.- RET

b) Cuádrupla.

SOLUCIÓN

1.- (›, a, b, T1)

2.- (si no, T1, ir_a_(6))

3.- (+,b,1,T1)

4.- (=, T1, ,b)

5.- (ir_a, , ,(8))

6.- (+, a, 1,T1)

7.- (=, T1, , a)

8.- RET

8.- Generar una tripleta y una cuádrupla para el siguiente segmento de código.

While a > b

b:=b+1

a) Tripleta

SOLUCIÓN

1 (>, a, b)

2 (sino, (1), ,(6))

3 (+, b, 1)

4 (=, (3), b)

5 (ir a, ,(1))

6 RET

b) Cuádrupla

SOLUCIÓN

1 (>, a, b, T1)

2 (sino, T1, ,(6))

3 (+, b, 1, T1)

4 (=, T1, , b)

5 (ir a, , ,(1))

6 RET

9.- Generar una tripleta y una cuádrupla para el siguiente segmento de código.

Int arreglo[ ] = new int[5];

For (int i=0; i<5; i++)

{arreglo[i]=i+1;]

a) Tripleta

SOLUCIÓN

1 (=, 0, i)

2 (new , , arreglo[ ])

3 (+, 1, i)

4 (=, (3), i)

5 (=, i, arreglo[i])

6 (<, i, 5)

7 (si true, (6), (2))

8 RET

b) Cuádrupla

SOLUCIÓN

1 (=, o, ,i)

2 (new , , ,arreglo[ ])

3 (+, 1, i, T1)

4 (=, T1, ,i)

5 (=, i, , arreglo [i])

6 (<, i, 5, T1)

7 (si true, T1, , (2))

8 RET

10.- Generar una tripleta y una cuádrupla para el siguiente segmento de código.

Repeat

a:=a+1

Until a > b

a) Tripleta

SOLUCIÓN

1 (+, 1, a)

2 (=, (3), a)

3 (>, a, b)

4 (si true, (1), (6))

5 (ir a, ,(1))

6 RET

b) Cuádrupla

SOLUCIÓN

1 (+, 1, a, T1)

2 (=, T1, , a)

3 (>, a, b, T1)

4 (si true, T1, , (6))

5 (ir a, , , (1))

6 RET

C Recursos Electrónicos de Apoyo

c1 Vídeo explicativo sobre el objetivo de la generación de código intermedio

c2 Presentación electrónica para explicar con ejercicios los subtemas 2.1 a 2.3

c3 Video explicativo sobre el proceso de solución aplicando las diferentes técnicas de generación de código intermedio, mencionadas en los subtemas de la unidad temática.

c4 Archivo de power point con información de apoyo y bibliografía de consulta para los temas de la unidad.

3. Tema 3 Optimización

La optimización en la generación de código durante la compilación es un proceso mediante el cual el compilador utiliza técnicas para mejorar el rendimiento y eficiencia del código generado a partir del código fuente del programa.

El proceso de optimización se realiza durante la compilación del código fuente, en el cual el compilador analiza el código y aplica transformaciones que mejoran el rendimiento del programa. Algunas técnicas de optimización comunes son:

1. Eliminación de código redundante: El compilador puede detectar y eliminar partes del código que no son necesarias para el funcionamiento del programa.
2. Optimización de bucles: El compilador puede reorganizar y simplificar los bucles del programa para mejorar el rendimiento.
3. Inlining: El compilador puede reemplazar llamadas a funciones con el código de la función en sí misma, lo que puede mejorar el rendimiento al reducir la sobrecarga de las llamadas a funciones.
4. Vectorización: El compilador puede transformar código que opera sobre datos en bucles en código que opera en vectores, lo que puede mejorar significativamente el rendimiento en procesadores que admiten instrucciones SIMD (Single Instruction, Multiple Data).
5. Eliminación de código muerto: El compilador puede detectar y eliminar partes del código que nunca se ejecutan, lo que puede reducir el tamaño del programa y mejorar la eficiencia.
6. Reordenamiento de instrucciones: El compilador puede reorganizar las instrucciones del programa para aprovechar mejor la jerarquía de memoria y reducir los tiempos de espera en la ejecución de instrucciones.

En general, la optimización del código durante la compilación puede mejorar significativamente el rendimiento y la eficiencia de los programas, lo que puede ser especialmente importante en aplicaciones críticas en cuanto a rendimiento. Sin embargo, también puede aumentar el tiempo de compilación y el tamaño del código generado, por lo que es importante equilibrar el nivel de optimización con las necesidades del programa y las limitaciones de hardware.

1. Usar algoritmos de optimización: Los algoritmos de optimización de compiladores pueden mejorar el rendimiento del código generado al identificar y reorganizar el flujo de control y los cálculos para que se ejecuten de forma más rápida. Estos algoritmos se pueden aplicar para reducir el uso de la memoria, mejorar la precisión y aumentar la velocidad de ejecución.

2. Utilizar herramientas de optimización: Las herramientas de optimización de compiladores permiten al usuario realizar cambios en el código fuente antes de la compilación. Estas herramientas pueden mejorar el rendimiento del código al permitir que el usuario realice cambios directos en el código fuente para reducir el tamaño de los archivos generados, eliminar líneas innecesarias, fusionar variables, reducir la cantidad de llamadas a la memoria y realizar otras optimizaciones.

3. Utilizar lenguajes de alto nivel: Los lenguajes de alto nivel permiten al usuario escribir código de forma más clara y estructurada, lo que facilita la optimización durante la compilación. Estos lenguajes también proporcionan funciones predefinidas

para reducir la cantidad de código necesario, lo que mejora el rendimiento de la compilación.
4. Utilizar compiladores de propósito especializado: Los compiladores de propósito especializado permiten optimizar el código durante la compilación porque están diseñados para una aplicación en particular. Estos compiladores se pueden configurar para producir código más eficiente y optimizado para una aplicación específica.
5. Utilizar herramientas de depuración: Las herramientas de depuración de compiladores permiten al usuario encontrar y solucionar errores del código antes de la compilación. Esto mejora el rendimiento de la compilación al evitar la generación de código defectuoso que puede provocar problemas de ejecución.
6. Utilizar compiladores de última generación: Los compiladores más recientes contienen mejoras en los algoritmos de optimización que permiten generar código más eficiente y optimizado. Estos compiladores también suelen incluir características tales como paralelización de código, optimización de memoria y mejoras en la generación de código.
7. Utilizar herramientas de análisis de rendimiento: Las herramientas de análisis de rendimiento permiten al usuario ver qué partes del código son más críticas para el rendimiento. Estas herramientas le permiten identificar los cuellos de botella en el rendimiento y optimizar el código para mejorar el rendimiento.
8. Utilizar procesamiento paralelo: El procesamiento paralelo permite a los compiladores paralelizar el código para que se ejecute en múltiples núcleos a la vez. Esto reduce el tiempo de ejecución del código al permitir que se ejecuten varias instrucciones al mismo tiempo.

En conclusión, la optimización de la generación de código durante la compilación se puede lograr utilizando algoritmos de optimización, herramientas de optimización, lenguajes de alto nivel, compiladores de propósito especializado, herramientas de depuración, compiladores de última generación, herramientas de análisis de rendimiento y procesamiento paralelo.

3.1. Tipos de optimización.

3.1.1. Locales.

La optimización local de un compilador implica realizar cambios en un fragmento de código para mejorar su eficiencia.

Estas optimizaciones pueden incluir la reorganización del código para reducir los ciclos de reloj, la eliminación de código redundante, la reutilización de variables, etc.

La optimización local también puede involucrar la reescritura de ciertos fragmentos de código para hacerlos más eficientes. Esto se hace a menudo mediante la utilización de técnicas como la reasignación de variables, la eliminación de bucles, etc.

Algunos compiladores también ofrecen herramientas específicas para la optimización local, como el análisis de línea a línea o los algoritmos de optimización como el algoritmo de optimización de flujo de control. La optimización local de un compilador es una parte importante del proceso de compilación y puede mejorar significativamente los resultados de rendimiento.

La optimización local también se puede usar para reducir el tamaño del código fuente, lo que hace que el compilador sea más rápido al procesar el código. Esta optimización puede también mejorar la seguridad del código. Si bien la optimización local de un compilador no siempre mejorará significativamente el rendimiento, es una parte

importante del proceso de compilación y debe considerarse cuando se diseñan aplicaciones.

Algunas de las herramientas de optimización local de un compilador incluyen:

- Analizador de línea a línea: Esta herramienta examina cada línea de código y busca oportunidades para mejorar el rendimiento.

- Reasignación de variables: Esta herramienta reasigna variables para mejorar el rendimiento. - Eliminación de bucles: Esta herramienta elimina los bucles innecesarios para mejorar el rendimiento.

- Algoritmos de optimización: Estos algoritmos examinan el código y buscan formas de mejorar el rendimiento.

- Analizador de memoria: Esta herramienta analiza la memoria usada por el programa y busca oportunidades para mejorar el rendimiento.

- Analizador de código: Esta herramienta busca formas de mejorar el rendimiento al reescribir ciertos fragmentos de código.

- Herramientas de caché: Estas herramientas ayudan a optimizar el uso de la memoria caché para mejorar el rendimiento.

- Herramientas de optimización de flujo de control: Estas herramientas examinan los programas para encontrar oportunidades de optimización de flujo de control.

- Herramientas de optimización de memoria: Estas herramientas ayudan a optimizar el uso de la memoria para mejorar el rendimiento.

- Herramientas de optimización de instrucciones: Estas herramientas ayudan a optimizar el uso de instrucciones para mejorar el rendimiento.

- Herramientas de optimización de registros: Estas herramientas ayudan a optimizar el uso de registros para mejorar el rendimiento.

La optimización local de un compilador es una técnica esencial para mejorar el rendimiento de un programa. La optimización local ayuda a acelerar el proceso de compilación y mejorar el rendimiento del programa al mismo tiempo. Esta técnica es especialmente útil para los programas que se ejecutan en entornos de tiempo real. La optimización local de un compilador puede ser un proceso complicado. Muchas de las herramientas de optimización local pueden ser difíciles de entender y configurar. Si bien hay una variedad de herramientas y técnicas de optimización local, es importante entender la naturaleza del proceso de compilación antes de intentar realizar optimizaciones locales. Esto ayudará a garantizar que todas las optimizaciones sean eficaces y no afecten negativamente la estabilidad del programa.

Esta técnica implica cambios en el código fuente para mejorar su eficiencia. Se puede lograr esto mediante la reasignación de variables, la eliminación de bucles, la reutilización de variables, y la reescritura de ciertos fragmentos de código para su reorganización, eliminación de código muerto, la eliminación de redundancias y la aplicación de técnicas de optimización específicas del hardware.

Estas optimizaciones pueden mejorar significativamente el rendimiento del programa. Sin embargo, es importante comprender la naturaleza del proceso de compilación antes de intentar optimizaciones locales.

La optimización local en la compilación se refiere a la aplicación de técnicas de optimización de código a nivel de compilación para mejorar el rendimiento del programa resultante. Estas técnicas se aplican en el proceso de compilación. La optimización local es importante porque puede mejorar significativamente el rendimiento de un programa sin necesidad de cambiar su lógica o estructura. Además, la optimización local es a menudo más eficiente que la optimización global, ya que puede aplicarse de manera selectiva a secciones específicas de código que se ejecutan con más frecuencia.

Sin embargo, la optimización local también tiene sus limitaciones. En algunos casos, las técnicas de optimización pueden generar código más complejo o menos legible, lo que dificulta el mantenimiento del programa. Además, algunas técnicas de optimización pueden ser incompatibles con ciertas características del hardware o del sistema operativo.

En general, la optimización local es una herramienta valiosa para mejorar el rendimiento de los programas, pero debe usarse con precaución y evaluarse cuidadosamente en función de las necesidades del programa y las limitaciones del hardware y del sistema operativo.

3.1.2. Ciclos.

La optimización de ciclos en un compilador es un proceso mediante el cual el compilador trata de reducir el número de instrucciones necesarias para ejecutar un programa. Esto se logra analizando el código fuente en busca de ciclos que se puedan reutilizar para reducir el número de instrucciones.

Las optimizaciones de los ciclos pueden incluir la eliminación de instrucciones redundantes, la combinación de instrucciones para reducir el tiempo de ejecución, el reordenamiento de instrucciones para aprovechar mejor los recursos de la memoria y la unificación de instrucciones para reducir la cantidad de memoria requerida y aumentar la velocidad de ejecución.

La optimización de los ciclos en un compilador es una tarea difícil, ya que se trata de analizar una gran cantidad de datos de entrada para determinar qué ciclos se pueden reutilizar y cómo. Esto requiere una gran cantidad de conocimientos acerca de la arquitectura de la computadora y el lenguaje de programación en el que se está escribiendo el código. Además, se necesitan herramientas especializadas para llevar a cabo este análisis.

Las técnicas de optimización de ciclos incluyen:

1. Eliminación de código muerto: el compilador elimina cualquier instrucción dentro del ciclo que no tenga ningún efecto sobre el resultado final del ciclo.
2. Reducción de fuerza de cálculo: el compilador reduce el número de cálculos dentro del ciclo, por ejemplo, mediante el reemplazo de una operación de multiplicación por una operación de suma.
3. Fusión de bucles: el compilador combina bucles que realizan tareas similares para reducir el número de iteraciones y minimizar el número de veces que se accede a los datos.
4. Reordenamiento de bucles: el compilador reorganiza el orden de las instrucciones dentro del ciclo para maximizar la eficiencia del procesador.

5. Vectorización: el compilador transforma el bucle en un vector para aprovechar las capacidades de procesamiento de vectores de la CPU.
6. Paralelización: el compilador divide el bucle en varias partes y realiza el procesamiento en paralelo en diferentes núcleos de la CPU o en diferentes hilos.

Al aplicar estas técnicas de optimización de ciclos, el compilador puede mejorar significativamente el rendimiento del código generado y, por lo tanto, reducir el tiempo de ejecución de un programa. Sin embargo, es importante tener en cuenta que la optimización del compilador puede generar código más complejo y difícil de entender, lo que puede hacer que el depurado sea más difícil.

3.1.3. Globales.

La optimización global en un compilador es una técnica de optimización que se centra en la mejora de la eficiencia de la ejecución de código de aplicaciones.

Esta técnica se aplica al código fuente de la aplicación antes de su compilación y se enfoca en la reorganización del código y la eliminación de instrucciones innecesarias. Esta optimización se dirige a la mejora de la eficiencia general de la aplicación al reducir el tiempo de ejecución y el uso de la memoria.

La optimización global en un compilador también puede incluir la reescritura de código para aprovechar características específicas del procesador o del sistema operativo.

Algunas técnicas de optimización global incluyen el análisis de flujo de datos, la optimización de accesos a la memoria, la eliminación de instrucciones redundantes y la reordenación de instrucciones para aprovechar mejor la arquitectura del procesador.

Esta técnica se centra en la reorganización del código y la eliminación de instrucciones innecesarias para mejorar significativamente el rendimiento de la aplicación. Puede incluir la reescritura de código para aprovechar características específicas del procesador o del sistema operativo.

La optimización global en la compilación es un proceso mediante el cual un compilador realiza un análisis de todo el código fuente y busca formas de mejorar el rendimiento de la aplicación. Esta optimización puede incluir tareas tales como la reorganización del código para minimizar los saltos entre instrucciones, la eliminación de código redundante, la reutilización de expresiones comunes y la optimización de los bucles a través del reemplazo de bucles de control con instrucciones más eficientes. Estas técnicas permiten que el compilador produzca código más eficiente y con mejor rendimiento. La optimización global también puede incluir la optimización de los recursos del sistema. Esto significa que el compilador puede optar por usar recursos del sistema de manera más eficiente, como evitar la creación de instancias innecesarias de una clase o evitar la asignación de memoria innecesaria.

Estas técnicas mejoran el rendimiento general de la aplicación y minimizan el uso de recursos del sistema.

El compilador realiza un análisis detallado del código fuente y busca formas de mejorar el rendimiento de la aplicación.

La optimización global en la generación de código final se refiere a aplicar mejoras a la estructura y el flujo del código, así como a la forma en que se traduce el código fuente a código de máquina.

La optimización global también puede implicar la reordenación de instrucciones de manera que se aproveche mejor la memoria caché y los buses de datos. Esto mejora la eficiencia general del programa, aumenta el rendimiento y reduce el tamaño del código final.

Es importante tener en cuenta que la optimización global debe realizarse con cuidado, ya que algunas optimizaciones pueden tener un efecto negativo en el rendimiento general del programa. Por lo tanto, se recomienda probar y medir la eficiencia antes de implementar cualquier optimización.

Este proceso se realiza en la fase de generación de código final, después de que se ha completado la fase de optimización local.

La optimización global puede mejorar significativamente el rendimiento del programa al permitir la identificación y eliminación de redundancias, la simplificación de expresiones complejas y la reorganización del código para mejorar el acceso a la memoria y reducir las operaciones de salto y ramificación.

Entre las técnicas comunes utilizadas en la optimización global se incluyen:

1. Eliminación de código muerto: La eliminación de código que no se utiliza en el programa puede reducir el tamaño del archivo ejecutable y mejorar el rendimiento del programa.
2. Propagación de constantes: Esta técnica reemplaza las variables que tienen un valor constante en todo el programa con su valor real. Esto puede reducir la cantidad de operaciones necesarias y mejorar el rendimiento.
3. Análisis de flujo de datos: Esta técnica identifica los datos que se utilizan y cómo se transforman a través del programa. Esto puede ayudar a identificar oportunidades de optimización, como la eliminación de cálculos innecesarios.
4. Reorganización del código: Esta técnica implica la reorganización del código para mejorar la localidad de referencia y reducir la cantidad de saltos y ramificaciones. Esto puede mejorar el rendimiento al reducir el número de ciclos de reloj necesarios para ejecutar el programa.
5. Inlining: Esta técnica implica la sustitución de llamadas a funciones con el código real de la función. Esto puede mejorar el rendimiento al reducir la sobrecarga de llamadas a funciones.

3.1.4. De mirilla.

La optimización de mirilla (en inglés, "loop optimization" o ¨profiling¨) es una técnica utilizada en la compilación de código para mejorar el rendimiento de los bucles en el programa. El objetivo de la optimización de mirilla es reducir el número de iteraciones que el bucle debe realizar o mejorar el acceso a los datos en el bucle.

La optimización de mirilla en la generación de código final se refiere a la técnica de analizar el código generado por un compilador y realizar ajustes para mejorar su eficiencia y rendimiento. La mirilla es una herramienta que permite a los desarrolladores de compiladores ver el código generado para una sección específica de código fuente.

Algunas de las técnicas de optimización de mirilla más comunes incluyen:

1. Eliminación de código redundante: Los compiladores a menudo generan código redundante que se puede eliminar para mejorar la eficiencia del programa.
2. Reorganización del código: La reorganización del código puede ayudar a mejorar la eficiencia al reducir el número de saltos y reducir la cantidad de ciclos de reloj necesarios para ejecutar el programa.
3. Optimización de registro: La optimización de registro implica asignar variables a registros de la CPU en lugar de la memoria, lo que puede reducir el tiempo de acceso y mejorar el rendimiento.
4. Optimización de bucles: La optimización de bucles implica reorganizar el código de un bucle para mejorar la eficiencia, como cambiar la orden de las instrucciones para reducir el número de saltos.
    a. Desenrollado de bucles: Esta técnica consiste en dividir el bucle en varias partes más pequeñas y ejecutar cada parte por separado. Esto puede mejorar el rendimiento al reducir el número de veces que el procesador debe verificar la condición del bucle.
    b. Reordenamiento de bucles: Esta técnica implica reordenar el código dentro del bucle para optimizar el acceso a los datos. Por ejemplo, se puede reorganizar el código para que los datos que se utilizan con mayor frecuencia estén más cerca del principio del bucle, lo que puede mejorar la velocidad de acceso.
    c. Vectorización: Esta técnica implica utilizar registros vectoriales en lugar de registros escalares para procesar múltiples datos a la vez. Esto puede ser especialmente útil para bucles que realizan operaciones matemáticas o estadísticas en grandes conjuntos de datos.
    d. Paralelización: Esta técnica implica dividir el bucle en partes más pequeñas y ejecutar cada parte en un núcleo de procesador diferente. Esto puede mejorar significativamente el rendimiento en sistemas de múltiples núcleos.
5. Eliminación de saltos innecesarios: Los saltos innecesarios pueden eliminarse para mejorar el rendimiento.

En general, la optimización de mirilla es una técnica muy efectiva para mejorar el rendimiento de los bucles en programas de computadora. Sin embargo, es importante tener en cuenta que no todas las optimizaciones son adecuadas para todos los programas y que puede haber algunas situaciones en las que la optimización de mirilla no tenga un impacto significativo en el rendimiento.

La optimización de mirilla se realiza en el código generado, no en el código fuente original. Esto se debe a que la mayoría de las optimizaciones son específicas de la arquitectura de la CPU y, por lo tanto, no se pueden realizar sin conocer la plataforma de destino. Es muy útil para identificar los cuellos de botella en la ejecución del programa.

Consiste en medir el tiempo de ejecución de cada parte del programa y así identificar cuáles son las secciones que consumen más tiempo de procesamiento. Para realizar esto, se puede utilizar herramientas específicas como "profilers" que se encargan de medir el tiempo de ejecución de cada línea de código.

Una vez identificadas las secciones del código que consumen más tiempo, se pueden aplicar diversas técnicas de optimización, como el uso de algoritmos más eficientes, la reducción de operaciones innecesarias, la eliminación de bucles redundantes, entre otras.

Es importante destacar que la técnica de mirilla no siempre resulta en mejoras significativas en la velocidad de ejecución del programa. En algunos casos, puede ser que las secciones que consumen más tiempo de procesamiento sean necesarias para la funcionalidad del programa, o que ya se estén utilizando las técnicas más eficientes posibles en el código. Por eso, es importante analizar cuidadosamente los resultados obtenidos con la técnica de mirilla antes de aplicar cualquier tipo de cambio en el código.

3.2. Costos.

La optimización en la compilación puede tener algunos costos asociados, como:

1. Mayor tiempo de compilación: Cuanto más agresiva sea la optimización, más tiempo puede llevar el proceso de compilación. Esto se debe a que el compilador necesita realizar un análisis más exhaustivo del código para determinar qué cambios realizar.
2. Mayor consumo de recursos: La optimización en la compilación puede requerir más recursos del sistema, como memoria y CPU. Esto puede hacer que el proceso de compilación sea más lento y aumentar el uso de recursos del sistema.
3. Posibles errores de optimización: Si el compilador realiza una optimización incorrecta, puede provocar errores en el código compilado. Por lo tanto, es importante que las optimizaciones se realicen de manera cuidadosa y se prueben exhaustivamente.
4. Código más difícil de depurar: La optimización en la compilación puede hacer que el código resultante sea más difícil de entender y depurar. Esto se debe a que las optimizaciones pueden cambiar la forma en que se ejecuta el código, lo que puede dificultar la identificación de errores.

A pesar de estos costos, la optimización en la compilación puede mejorar significativamente el rendimiento del código resultante, lo que puede ser especialmente importante en aplicaciones de alta demanda de recursos.

Por lo tanto, es importante sopesar cuidadosamente los costos y beneficios de la optimización en la compilación antes de decidir si usarla o no.

3.2.1. Costo de ejecución. (Memoria, registros, pilas).

Durante la compilación, se utilizan varios recursos, como la memoria, los registros y las pilas, para ejecutar el código fuente y generar el código objeto o ejecutable.

La memoria es necesaria para almacenar el código fuente y los datos utilizados por el compilador. El tamaño de la memoria requerida depende del tamaño del código fuente y la complejidad del programa.

Los registros son bloques de memoria que se utilizan para almacenar temporalmente datos en el procesador. Durante la compilación, el compilador utiliza los registros para almacenar datos intermedios y realizar operaciones aritméticas. La cantidad de

registros utilizados depende del tamaño del código fuente y la arquitectura del procesador.

Las pilas se utilizan para almacenar temporalmente datos y direcciones de retorno durante la ejecución de una función. Durante la compilación, el compilador utiliza la pila para almacenar los valores de las variables locales y los parámetros de la función. La cantidad de espacio de pila utilizado depende del tamaño y la complejidad de las funciones.

En general, el costo de ejecución en la compilación depende de varios factores, como la complejidad del programa, la arquitectura del procesador y la optimización del compilador. Un compilador optimizado puede reducir el uso de memoria, registros y pilas, lo que puede mejorar el rendimiento del programa.

Los costos de ejecución en la compilación dependen del lenguaje de programación que se esté utilizando. Los lenguajes de programación modernos usan una variedad de mecanismos para optimizar el uso de la memoria, los registros y la pila.

Estos mecanismos incluyen el código de optimización, el uso de bibliotecas eficientes y la optimización del código de ensamblaje.

Los costos de ejecución en la compilación también dependen de la arquitectura de la computadora en la que se ejecuta el código. Por ejemplo, una arquitectura de 32 bits usará menos memoria, registros y pilas que una arquitectura de 64 bits.

Los costos de ejecución en la compilación se pueden reducir mediante el uso de herramientas de optimización.

Estas herramientas permiten al usuario identificar y eliminar código redundante y optimizar el uso de recursos. Esto puede mejorar significativamente el tiempo de ejecución y la eficiencia del código. Además, los lenguajes de programación modernos también ofrecen características que permiten al usuario crear código eficiente sin tener que usar herramientas de optimización.

En general, los costos de ejecución en la compilación son una parte importante de la optimización de un programa.

3.2.2. Criterios para mejorar el código.

Existen varios criterios que se pueden utilizar para mejorar el código durante la compilación. Algunos de ellos son:

1. Eficiencia: El código debe ser eficiente en términos de tiempo de ejecución y uso de recursos. Para ello, se pueden aplicar técnicas de optimización como la eliminación de código redundante, la reutilización de variables, el uso de algoritmos más eficientes, entre otros.
2. Funcionalidad: Que los cambios efectuados no modifiquen la función original de la aplicación.
3. Legibilidad: El código debe ser fácil de entender y mantener. Para ello, se pueden aplicar convenciones de codificación claras y consistentes, utilizar nombres de variables y funciones descriptivos, agregar comentarios donde sea necesario, entre otros.
4. Modularidad: El código debe estar organizado en módulos que realicen tareas específicas y se puedan reutilizar en otros proyectos. Para ello, se pueden

aplicar patrones de diseño de software y técnicas de programación orientada a objetos.

5. Portabilidad: El código debe ser compatible con diferentes plataformas y sistemas operativos. Para ello, se pueden utilizar herramientas y bibliotecas multiplataforma, y evitar dependencias específicas de una plataforma.
6. Seguridad: El código debe ser seguro y proteger los datos de los usuarios. Para ello, se pueden aplicar técnicas de codificación segura, como la validación de entradas de usuario, el cifrado de datos sensibles, entre otros.
7. Escalabilidad: El código debe ser escalable para adaptarse a futuros cambios y requerimientos del proyecto. Para ello, se pueden aplicar patrones de diseño escalables, utilizar estructuras de datos eficientes, entre otros.
8. Mantenibilidad: El código debe ser fácil de mantener y actualizar en el futuro. Para ello, se pueden aplicar técnicas de documentación, como comentarios en el código y documentación externa, y seguir convenciones de codificación claras y consistentes.

3.2.3. Herramientas para el análisis del flujo de datos

El análisis de flujo de datos es una técnica útil para optimizar el código. Esta técnica se usa para identificar los cuellos de botella en el rendimiento del código, al analizar el flujo de datos a través de los componentes del programa. Esto permite al programador encontrar los puntos críticos en el código y optimizar los métodos de procesamiento para mejorar el rendimiento.

El análisis de flujo de datos también se puede usar para identificar los componentes innecesarios del código, lo que puede ayudar a reducir el tamaño del código y mejorar la eficiencia del mismo.

Para realizar un análisis de flujo de datos, primero se deben identificar los diferentes componentes del código. Luego, se debe trazar el flujo de datos entre los componentes del código. Esto implica identificar los datos que entran y salen de cada componente, así como los caminos de procesamiento de datos entre los componentes. Al hacer esto, se pueden encontrar los cuellos de botella y otros puntos de optimización potencial.

Una vez que se ha identificado un cuello de botella, como programador se puede trabajar para mejorar el rendimiento en este punto. Esto generalmente implica reescribir el código para eliminar los pasos innecesarios, reordenar el código para aprovechar la memoria de forma más eficiente, o utilizar algoritmos más rápidos.

Consiste en identificar los datos que se utilizan en un programa, cómo se procesan y cómo se almacenan, y luego utilizar esta información para mejorar el rendimiento del programa.

El análisis de flujo de datos se puede realizar de varias maneras, incluyendo:

1. Identificar los datos de entrada y salida del programa, y cómo se transforman los datos dentro del programa.
2. Identificar los cuellos de botella en el procesamiento de datos y buscar formas de optimizar el código para evitarlos.
3. Identificar las variables y los objetos que se utilizan en el programa y cómo se almacenan en memoria.

4. Identificar los flujos de control y las estructuras de datos utilizadas en el programa y buscar formas de optimizar su uso.

Una vez que se ha realizado el análisis de flujo de datos, se pueden tomar varias medidas para optimizar el código. Algunas de estas medidas incluyen:

1. Reducir la cantidad de datos que se procesan en el programa para mejorar la eficiencia.
2. Utilizar algoritmos más eficientes para procesar los datos.
3. Utilizar estructuras de datos más eficientes para almacenar los datos.
4. Utilizar técnicas de programación más eficientes para reducir el tiempo de ejecución del programa.

ACTIVIDADES DE APRENDIZAJE DE LOS TEMAS DE LA ASIGNATURA

A Recursos de Evaluación

1. ¿En qué parte del proceso de compilación se lleva a cabo la optimización?

Después de la generación de código intermedio.

2. ¿Por qué la optimización es dependiente del contexto?

Porque es necesario tomar en cuenta el contexto físico de la arquitectura donde se procesará el programa.

3. Explicar por lo menos 3 criterios para mejorar código.

-Reducir el tamaño del código, mediante la eliminación de expresiones redundantes, o código inalcanzable, así como depuración de instrucciones en bloques de loops.

-Sustituir instrucciones menos perfomantes por otras más performantes, para obtener una mayor velocidad de procesamiento.

-Utilizar en mayor medida los registros, para reducir los accesos a la memoria RAM.

-Preservar la función original, que las transformaciones que se realicen permitan que el código siga realizando su tarea original.

4. ¿Por qué es útil dividir el código en bloques básicos, para la labor de optimización?

La idea del bloque básico es encontrar partes del programa cuyo análisis necesario para la optimización sea lo más simple posible.

5. ¿Cuáles son las optimizaciones que son independientes de la máquina?

- Simplificaciones algebraicas
- Eliminar código inalcanzable
- Folding
- Reducción de subexpresiones comunes
- Propagación de copias y constantes

6. ¿Una optimización puede ser dependiente de la máquina objeto?

SI

7. Realizar la optimización de tipo mirilla con la expresión:

ADD var,02

MOV X, AH

MOV AL, X

OPTIMIZACIÓN:

MOV AL, AH

8. ¿Folding es una optimización independiente o dependiente de la máquina?

INDEPENDIENTE

9. ¿Para qué nos sirve identificar las variables vivas en un código?

Para eliminar aquellas variables que ya no son utilizadas después de un bloque básico.

10. Explicar la siguiente fórmula: ∑ (use(x,B) + 2 * live (x,B))
Bloques B en L

Esta fórmula nos permite identificar que tan deseable es guardar una variable x, en un registro, mediante la evaluación del comportamiento de ésa variable en todos los bloques B en un ciclo L, sumando las veces que aparece la variable x en cada bloque, y multiplicando por 1 si la variable sigue viva al salir del ciclo y por 0 si la variable ya no es usada.

B Ejercicios Propuesto y Resueltos

1.- Llevar a cabo la optimización del siguiente código usando la técnica de FOLDING.

1a.-

```
Int a;

Int b;

Int c;


public int Ganancias ()

{

a = 5+7+4-2 + a + b + c;

return a;

}
```

SOLUCIÓN

public int Ganancias () {

a = 14 + a + b + c;

return a;

}

1b.

Int total=0,b=2;

Total=1*4*2/b;

SOLUCIÓN

total=8/b;

En esta optimización se pretende evaluar las expresiones con operandos constantes, para evitar la evaluación en tiempo de ejecución.

2.- Optimizar el siguiente código con la técnica de reutilización de expresiones comunes:

2a

Int a,b,c,d,e,f,g,h;

Public int índice_masa_corporal()

{

a=(b+c)/2;

d=e*f;

g=(b+c)/2;

h=e*f;

SOLUCIÓN

Public int índice_masa_corporal()

{

a=(b+c)/2;

d=e*f;

g=a;

h=d;

}

2b.

Float total=sueldo+0.30, total2;

```
Public void calcular()
{
    Total2=sueldo+0.30;
    Total2=total2+comisiones;
}
```

SOLUCIÓN

```
Float total=sueldo+0.30, total2;
Public void calcular()
{
    Total2=total+comisiones;
}
```

En este ejemplo podemos notar que la expresión: sueldo+0.30, es redundante puesto que se está haciendo dos veces el mismo proceso, entonces para la optimización se pretende reutilizar el resultado anterior calculado, en este caso sobre el identificador "total"

3.- Optimizar el siguiente código con la técnica de propagación de constantes:

3a.

```
Int a=0;
Int b=0;
Int c=0;
Public int Ganancias()
{
a = 10;
b = 10 + a;
c = a + b;
return c;
}
```

SOLUCIÓN

```
Int a=10;
Int b=10;
Int c=0;
Public int Ganancias()
```

{

c= a+b;

return c;

}

3b.

Int total=0,b=2;

total=8/b;

SOLUCIÓN

total=8/2;

total=4;

En este caso la constante se sustituye y es evaluada.

4.- Optimizar utilizando reducción de potencia:

4a.

Float b;

Float c;

Public float Area()

{

C= b*5;

Return c;

}

SOLUCIÓN

Public float Area()

{

C= b+b+b+b+b;

Return c;

}

4b.

Int exp=3;

P=Pow(2,exp);

SOLUCIÓN

Int exp=3;

```
for(a=1;a<=exp;a++){
   p=p*2; }
```

Se pretende reemplazar expresiones costosas, en este caso la función “pow” por otras más simples.

5.- Optimizar utilizando Propagación de copias

5a

```
Int  X,Z,Y,Q,P,F,M;


Public int cálculos()
{
X=3+Z;
F=X;
Y=F+Q;
M=F+P;
}
```

SOLUCIÓN

```
Public int cálculos()
{
X=3+Z;
Y=X+Q;
M=X+P;
}
```

5b

```
void funcion(int a, int b){  for(int j = 0; j <= 10; j++){
                              a          = j;
for (int i = 0; i <= 10; i++){
                              b          = i;
cout<<”tabla del ” << a << a*b << endl; }
                        }
}
```

SOLUCIÓN

```
void funcion(){
        for(int j = 0; j <= 10; j++){
for (int i = 0; i <= 10; i++){
cout<<"tabla del " << j << j*i << endl;
}
                }
}
```

6.- Optimizar utilizando transformaciones algebraicas.

6a

```
int a;
int b = 29;
int c = 34;
int d;
int resul;

public int Descuento ()
{   a = ((23 + c) * b) + 89;
d = ((23 +c) * b) + 10;
resul = (((23 +c) * b) + 89) + (((23 +c) * b) + 10);
return resul;
}
```

SOLUCIÓN

```
int a;
int b = 29;
int c = 34;
int resul;

public int Descuento()
{ a = (25 + c) * b;
resul = (a+ 89) + (a + 10);
```

```
return resul;
}
```

6b

```
float DistanciaEntreDosPuntos(int x1, int y1, int x2, int y2){
float distancia = Math.sqrt ((Math.pow((x2-x1),2) +
      Math.pow((y2-y1),2)), 2);
    return distancia;
}
```

SOLUCIÓN

```
float void DistanciaEntreDosPuntos(int x1,int y1,int x2,int y2){
 return Math.sqrt (((x2-x1)*(x2-x1) + (y2-y1)*(y2-y1)), 2)  }
```

7.- Utilizar Optimización local, para el siguiente código.

7a

```
public double division( int num1, int num2){
    double result;
    result= num1 / num2;

 return result;

}
```

SOLUCIÓN

```
public double division( int num1, int num2)
{
    double result= num1 / num2;
    return result;
}
```

7b

```
for (i=3;i<5;i++)
{
j=j+c[i];
```

b=j*i;

}

c=3;

j=2;

salto l;

b=8;

l: z=2;

h=6;

SOLUCIÓN

BB1

i=3;

BB2

i<5:

BB3

j=j+c[i];

b=j*i;

i++;

BB4

c=3;

j=2;

salto l;

BB5

b=8;

BB6

l: z=2;

h=6;

8.- Optimizar sobre un ciclo o bucle.

8a

do

{

```
ítem = 10;
value = value + ítem;
} while(value<100);
```

SOLUCIÓN

```
ítem = 10;
do {
value = value + ítem;
}while(value<100);
```

8b Con reducción por invariante de ciclo o bucle.

```
Int a=0,b=10;
For(int i=1 ;i<b;i++)
{
  System.out.print(i +"'\t"+a)
    a=1;
}
```

SOLUCIÓN

```
Int a=0,b=10;
a=1;
For(int i=1 ;i<b;i++)
{
  System.out.print(i +"'\t"+a)
}
```

En este ejemplo podemos observar que dentro del ciclo, siempre se hace la asignación a=1; y es algo que nunca va a cambiar dentro del bucle o ciclo y no tiene efectos sobre las operaciones dentro del mismo. Por lo consiguiente en la optimización esta asignación se puede mover fuera del ciclo.

8c. Con fusión de ciclo o bucle.

```
For(int i=0;i<palabra1.length();i++)
{
System.out.println(palabra1.charAt(i));
}


For(int i=0;i<palabra2.length();i++)
```

{

System.out.println(palabra1.charAt(i));

}

SOLUCIÓN

Public void imprimir(String palabra){

For(int i=0;i<palabra.length();i++){

System.out.println(palabra.charAt(i));}

}

Imprimir(palabra1);

Imprimir(palabra2);

En este ejemplo se pretende mostrar que el fusionar bucles, que tienen una misma estructura, puede reducir el número de pruebas que realiza cada uno.

En este caso vemos que en la parte izquierda de nuestro código se utilizan dos ciclos con la misma estructura, por lo tanto, podemos hacer un método que contenga la función de este ciclo, el cual recibe como parámetro una cadena que son las que estamos utilizando en nuestro ejemplo. Finalmente, mandamos a llamar al método que contiene a nuestro ciclo con sus respectivas cadenas.

9.- Utilizar optimización global, para el siguiente código.

9a

i:=1;

while i<= 1000 do

{

a:=4*i+3;

b:=2*i;

i:=i+1;

}

SOLUCIÓN

i:=1;

t1 :=7;

t2:=2;

while i<= 1000

do{

a:=t1;

```
   b:=t2;
   I:=i+1;
  t1:=t1+4;
   t2:=t2+2;
}
```

9b.

```
Int a,b=1,c=2,d=3;
String pal=”hola”, pal2=”mundo”, pal3=””;
Public void metodo1()
{
  For(a=0;a<d;a++)
{
   B=c+a;
}
}
Public void metodo2()
{
  Pal3=pal+pal2;
}
```

SOLUCIÓN

```
String pal=”hola”, pal2=”mundo”,
Int d=3, c=2;
Public void metodo1()
{
  For(int a=0;a<d;a++)
{
  Int b=c+a;
}
}
Public void metodo2()
{
```

```
 String  Pal3=pal+pal2;
}
```

En este ejemplo vemos que en el primer código todas las variables declaradas son globales, lo cual no es muy recomendable puesto que produce:

- Legibilidad menor.
- Nos condiciona en muchos casos que el programa sólo sirva para un conjunto de casos determinados.

En cambio, el utilizar variables locales (como en el código derecho) hace que las variables declaradas dentro de una función son automáticas por defecto, es decir, sólo existen mientras se ejecuta la función. Por esto mismo optimiza nuestro código.

También puede suceder que en un mismo ámbito aparezcan variables locales y globales con el mismo nombre. Cuando sucede esta situación, siempre son las variables locales y argumentos formales los que tienen prioridad sobre las globales.

10.- Utilizar optimización de mirilla, para eliminar cargas innecesarias.
10a

```
mov ah,01h
int 21h
cmp al,35h
JNE noson
mov ah,00h
mov al,03h
int 10h
mov ah,09h
mov dx,OFFSET mensajes
int 21h
jmp fin
noson:
mov ah,00h
mov al,03h
int 10h
mov ah,09h
mov dx,OFFSET mensaje
int 21h
SOLUCIÓN
mov ah,01h
int 21h
cmp al,35h
JNE noes
  Call limpia
mov ah,09h
mov dx,OFFSET mensaje1
int 21h
jmp fin
```

```
noes:
  call limpia
mov ah,09h
mov dx,OFFSET mensaje2
int 21h
10b
Mov bl,03h
Mov al,00h
Ciclo:
Cmp al,bl
Jle proceso
Jmp ret
Proceso:
Inc al
Jmp ciclo
SOLUCIÓN
          xor cx,cx   ;se inicializa cx en 0
loop1    proceso
          inc cx
          cmp cx,
          jle loop1
```

En este ejemplo podemos observar cómo, para hacer un ciclo for en ensamblador, existen diferentes formas. Sin embargo, en la parte izquierda se puede ver como al realizar este ciclo con puros saltos a etiquetas el código se hace un poco más extenso a comparación del código de lado derecho, puesto que de este lado solo utilizamos directamente la instrucción LOOP, cuyo propósito es generar un ciclo en el programa con ayuda del registro cx. Así optimizamos este código haciéndolo más simple.

C Recursos Electrónicos de Apoyo

c1 Archivo de power point que contenga fuentes bibliográficas que permitan obtener información de apoyo sobre las situaciones en las que se aplican los diferentes tipos de optimización, así como apuntes que expliquen de forma puntual y breve lo que caracteriza a cada una de estas técnicas de optimización.

c2 Video con la explicación paso a paso para implementar los diferentes tipos de optimización mencionados en el subtema 3.1, sobre bloques de código.

c3 Presentación electrónica que sirva como refuerzo para repaso individual y posterior a una exposición en clase del subtema 3.2

4. Tema 4 Generación de código objeto.

La generación de código objeto se refiere al proceso de traducir el código fuente de un programa en un lenguaje de programación de alto nivel a un código de máquina de bajo nivel que puede ser entendido por el procesador de la computadora.

El código objeto es el resultado de este proceso de traducción y es una representación binaria del programa en el que se han resuelto todas las referencias a símbolos y se han aplicado las optimizaciones necesarias.

El código objeto generado a partir del código fuente generalmente no es ejecutable directamente, ya que aún necesita ser vinculado con otras bibliotecas y componentes del sistema para crear un programa ejecutable completo. Esto se realiza mediante un proceso de vinculación que combina los diferentes archivos objeto generados a partir del código fuente y las bibliotecas utilizadas por el programa. Tarea realizada por el enlazador o linker.

La generación de código objeto es una etapa en el proceso de compilación de un programa de computadora. Esto se realiza después de la etapa de análisis sintáctico, y antes de la etapa de enlace. El propósito de esta etapa es convertir el código fuente escrito por el programador en un lenguaje de máquina que el procesador pueda entender. Esto se logra a través de una traducción a un lenguaje intermedio, como el código objeto o ensamblador.

El código objeto se puede compilar en un lenguaje de máquina diferente para ejecutarse en diferentes arquitecturas. Esto significa que un programa escrito en un lenguaje de alto nivel puede ser compilado para varios sistemas operativos.

La generación de código objeto se realiza en varias etapas. En primer lugar, el compilador analiza el código fuente para identificar los tokens. Estos tokens se usan para construir un árbol de sintaxis abstracta (AST). Esto es una representación estructurada del código. El AST se utiliza para generar el código objeto.

El código objeto es el código que ejecuta el procesador. La generación de código objeto se realiza en varios pasos. Primero, se realiza una optimización para asegurar que el código objeto sea eficiente. Luego, el código objeto se genera en un lenguaje de máquina. Finalmente, el código objeto se almacena en un formato de archivo específico.

Generar código objeto es una parte importante del proceso de compilación. Esto permite que los programas sean portables entre diferentes plataformas. Es importante que el código objeto sea optimizado para que pueda ejecutarse de manera eficiente. Esto asegura que el programa tenga un rendimiento óptimo.

Generar código objeto es una tarea compleja y requiere una gran cantidad de recursos. Es importante que los compiladores sean eficientes para que los programadores puedan crear programas eficientes.

Los compiladores modernos se diseñan para minimizar el tiempo de compilación, así como para generar código óptimo para los programas.

En conclusión, la generación de código objeto es una etapa fundamental en el proceso de compilación. Esta etapa se encarga de convertir el código fuente escrito por el programador en un lenguaje de máquina que el procesador pueda entender. Esto se logra a través de una traducción a un lenguaje intermedio, como el código objeto o

ensamblador. Esto permite que los programas sean portables entre diferentes plataformas.

4.1. Registros.

Uno de los principales aspectos de la generación de código objeto involucra el manejo de registros.

Los registros son áreas de memoria especiales en el procesador que se usan para almacenar datos temporales durante la ejecución de un programa.

Estos registros se usan para almacenar los datos que se usan con más frecuencia, para que estén disponibles cuando se necesiten. El compilador puede usar los registros de forma eficiente para mejorar el rendimiento del programa.

Por ejemplo, el compilador puede usar los registros para almacenar variables, contadores y otros datos temporales. Esto evita que el compilador tenga que acceder a la memoria principal para recuperar estos datos, lo que reduce el tiempo de ejecución del programa.

Otra forma en que el compilador puede usar los registros es para realizar operaciones. El compilador puede usar los registros para realizar operaciones aritméticas y lógicas, como la suma, la multiplicación, la división y el desplazamiento de bits. Esto también mejora el rendimiento del programa, ya que permite que el procesador realice estas operaciones mucho más rápido que si tuviera que utilizar la memoria principal.

El manejo de registros en la generación de código objeto se refiere al proceso de asignar y utilizar registros de la CPU de manera eficiente durante la generación de código objeto a partir de un programa fuente. Los registros son una cantidad limitada de memoria de alta velocidad dentro de la CPU que se utilizan para almacenar datos temporalmente durante la ejecución de un programa.

Durante la generación de código objeto, es importante utilizar los registros de manera efectiva para minimizar la cantidad de accesos a la memoria principal, lo que puede mejorar significativamente el rendimiento del programa resultante. Para lograr esto, el compilador debe realizar un seguimiento de qué registros están disponibles en cada punto del programa y asignarlos a las variables de manera óptima.

El proceso de asignación de registros se puede dividir en dos pasos: asignación y liberación. En el paso de asignación, el compilador determina qué variables se pueden asignar a un registro disponible en un momento dado. En el paso de liberación, el compilador debe asegurarse de que los registros asignados se liberen cuando ya no se necesiten, de modo que se puedan reutilizar para otras variables.

Existen diversas técnicas para el manejo eficiente de registros, incluyendo la asignación basada en grafos de interferencia y la asignación basada en heurísticas.

En general, el objetivo es minimizar el número de accesos a memoria y maximizar el uso de los registros disponibles para mejorar el rendimiento del programa generado.

La reserva y asignación de registros de memoria durante la generación de código objeto es un proceso en el que se asignan ciertos segmentos de memoria a los diversos componentes de un programa.

Estos segmentos de memoria son pequeñas porciones de memoria que se asignan para almacenar variables, constantes, instrucciones y cualquier otro elemento del programa. Al asignar estos segmentos de memoria, el compilador puede optimizar mejor el programa para asegurar un funcionamiento más rápido y un uso más eficiente de los recursos.

La asignación de registros de memoria durante la generación de código objeto se realiza de dos formas. En primer lugar, el compilador puede asignar automáticamente segmentos de memoria a los componentes del programa. Esto significa que el compilador asignará automáticamente una parte de la memoria al programa. Esta asignación se realiza en función del tamaño del programa y del tipo de datos. Esto significa que el compilador asignará un segmento de memoria mayor para aquellos componentes que contengan una gran cantidad de información. En segundo lugar, el compilador también puede permitir que el programador asigne manualmente los segmentos de memoria a los componentes del programa. Esto significa que el programador tendrá que asignar manualmente los segmentos de memoria a los componentes. Esto le permitirá optimizar el uso de los recursos y asegurar un mejor rendimiento del programa.

Por lo tanto, la asignación de registros de memoria durante la generación de código objeto es una herramienta importante para optimizar el programa.

La reserva y asignación de registros en la generación de código objeto es una técnica que se utiliza en la compilación de lenguajes de programación para asignar los registros de la CPU a las variables del programa. Esta técnica es especialmente importante en arquitecturas de procesadores que tienen un número limitado de registros.

La reserva de registros implica reservar un conjunto de registros para ser utilizados por el compilador para almacenar variables temporales o valores intermedios. La asignación de registros implica asignar un registro específico a una variable particular. Esto se hace de manera que los registros se utilicen de manera eficiente y se minimice el costo de acceso a la memoria.

Existen diferentes estrategias para la reserva y asignación de registros. Algunas de ellas son:

- Asignación por demanda: los registros se asignan según sea necesario durante la ejecución del programa.
- Asignación estática: los registros se asignan en tiempo de compilación y se mantienen asignados durante toda la ejecución del programa.
- Asignación basada en heurísticas: se utilizan técnicas heurísticas para determinar la mejor asignación de registros. Estas técnicas pueden incluir la evaluación del tiempo de vida de la variable, la frecuencia de acceso a la variable, el tamaño de la variable, entre otros factores.

La reserva y asignación de registros en la generación de código objeto es una técnica importante para mejorar el rendimiento del código generado y reducir el costo de acceso a la memoria. Sin embargo, su implementación puede ser compleja y requiere un conocimiento detallado de la arquitectura del procesador y del lenguaje de programación utilizado.

Algoritmo para la asignación de registros:

Obtenreg(x:=y op z)

1.usar el registro de y si está en un registro que no tiene otra variable, y además y no está viva ni tiene uso posterior. Si no:
2. usar un registro vacío si hay. Si no:
3. usar un registro ocupado (spill) si op requiere que x esté en un registro o si x tiene uso posterior.
Actualizar el descriptor de registro. Si no:
5. usar la posición de memoria de x

Ejemplo de asignación local de registros:

f=(a-b)*(c+d)+e

Distinguiendo variables temporales en bloque básico	Instrucciones
	t1=a-b
	t2=c+d
	t3=t1*t2
	t4=t3+e
	f=t4
		Código
		MOV a, R0
		SUB b, R0
		MOV c, R1
		ADD d, R1
		MULT R0, R1
		ADD e, R1
		MOV R1, f
			Descriptores Reg
			R0 con t1
			R1 con t2
			R1 con t3
			R1 con t4

Las variables temporales no se distinguen	Instrucciones
	t1=a-b
	t2=c+d
	t3=t1*t2
	t4=t3+e
	f=t4
		Código
		MOV a, R0
		SUB b, R0
		MOV c, R1
		ADD d, R1
		MOV R1, R2
		MULT R0, R2
		MOV R2, R3
		ADD e, R3
		MOV R3, f
		//SALVAR R1-R4
			Descriptores Reg
			R0 con t1
			R1 con t2
			R2 con t3
			R3 con t4

4.2 Lenguaje ensamblador.

El lenguaje ensamblador es una representación simbólica del lenguaje de máquina de un procesador específico.

Es utilizado por los programadores para escribir programas de bajo nivel que interactúan directamente con el hardware de la computadora.

En la generación de código objeto, el lenguaje ensamblador se utiliza para traducir el código fuente escrito en lenguaje de alto nivel en código objeto, que es un archivo binario que contiene el código ejecutable para la máquina en la que se está desarrollando el programa.

El proceso de generación de código objeto implica varias etapas. Primero, el programador escribe el código fuente en lenguaje ensamblador. Luego, un ensamblador procesa el código fuente y lo convierte en código objeto. El código objeto es un archivo binario que contiene el código ejecutable, así como información sobre las direcciones de memoria donde se cargan los datos y las instrucciones del programa.

El lenguaje ensamblador se utiliza en la generación de código objeto debido a su capacidad para proporcionar un control granular sobre el hardware de la computadora. Los programadores pueden escribir instrucciones precisas que interactúan directamente con los registros de la CPU, los buses de datos y otros componentes de hardware. Además, el lenguaje ensamblador permite una optimización precisa del código, lo que puede resultar en programas más eficientes y rápidos.

Sin embargo, el lenguaje ensamblador también tiene sus limitaciones. Es un lenguaje de programación de bajo nivel que puede ser difícil de aprender y utilizar, especialmente para programadores novatos. Además, los programas escritos en lenguaje ensamblador pueden ser más difíciles de mantener y depurar debido a la complejidad y la falta de abstracción del código.

El código ensamblador es una abstracción de la máquina real, y el código objeto es una representación exacta del código que se ejecutará en la máquina. El código objeto se crea a partir del código ensamblador mediante un proceso llamado enlazado. El enlazador toma el código ensamblador y lo traduce a código de máquina, lo cual permite que el programa se ejecute en la máquina.

Es importante notar que el código objeto no se puede ejecutar directamente en la máquina. Primero debe pasar por un proceso de enlazado para convertirse en código de máquina.

El lenguaje ensamblador es un lenguaje de programación muy eficiente y útil para la generación de código objeto. Dado que el código ensamblador es un lenguaje de alto nivel, es más fácil de leer y entender que el código de máquina. Esto hace que el lenguaje ensamblador sea una excelente herramienta para los desarrolladores que necesitan escribir programas eficientes para una variedad de plataformas. Además, el código objeto generado a partir del código ensamblador es a menudo más eficiente que el código generado por otros lenguajes de programación. Esto significa que los programas escritos en lenguaje ensamblador suelen ser más rápidos y tienen un mayor rendimiento que los programas escritos en otros lenguajes.

Por lo tanto, el lenguaje ensamblador es una herramienta útil para los desarrolladores que buscan obtener el máximo rendimiento de sus programas.

El lenguaje ensamblador al ser un lenguaje de bajo nivel, permite interactuar directamente con el hardware, a través de sus instrucciones.

Por lo que el código objeto es mayormente implementado en éste lenguaje, ya que aquel requiere considerar las características físicas de la máquina destino.

El código intermedio usando tripletas y cuádruplas puede fácilmente traducirse a lenguaje ensamblador, ya que su formato es similar al formato de las instrucciones en lenguaje ensamblador.

Formato de una instrucción en lenguaje ensamblador:

Label:	operation code	argument1	argument2

Formato de una tripleta:

Label (operation code, argument1, argument2)

Ejemplo:

x:= a+5

Tripleta	Lenguaje ensamblador
	Mov r1,a
1.- (+, 5, a)	add r1,5
2.- (=, (1), x)	mov x,r1

4.3 Lenguaje máquina.

El lenguaje máquina es un lenguaje de bajo nivel que es utilizado por las computadoras para ejecutar instrucciones. En la generación de código objeto, el lenguaje máquina se utiliza para traducir el código fuente de un programa en un archivo ejecutable que la computadora puede entender.

El proceso de generación de código objeto comienza con el compilador, que toma el código fuente y lo convierte en código objeto. El compilador traduce el código fuente en lenguaje de programación de alto nivel en un conjunto de instrucciones en lenguaje máquina que puede ser ejecutado directamente por la computadora. Estas instrucciones se escriben en un archivo objeto que contiene código máquina y datos relacionados con el programa.

El código objeto todavía no es ejecutable, ya que se encuentra en un formato binario que la computadora no puede interpretar directamente. El siguiente paso es el enlazador (linker), que combina el código objeto con las bibliotecas de tiempo de ejecución necesarias para crear un archivo ejecutable que puede ser cargado y ejecutado por la computadora.

El lenguaje de máquina es un lenguaje de programación de bajo nivel que se ejecuta directamente en un procesador. Es el lenguaje de programación más bajo y más cercano al hardware, ya que los procesadores solo pueden entender códigos binarios.

El código objeto es un código intermedio entre el lenguaje de máquina y el lenguaje de alto nivel.

El compilador traduce el código fuente escrito en un lenguaje de alto nivel (como C, C ++, Java, etc.) en código objeto.

El código objeto, como su nombre lo indica, es un código que se encuentra entre el lenguaje de alto nivel y el lenguaje de máquina. Se compone de instrucciones binarias que pueden ser entendidas por el procesador, pero aún no se han optimizado para realizar tareas específicas.

El código objeto se optimiza aún más para convertirlo en el código de máquina que el procesador puede entender y ejecutar.

Por lo tanto, el código objeto es un paso intermedio entre el lenguaje de alto nivel y el lenguaje de máquina. El compilador traduce el código fuente escrito en un lenguaje de alto nivel (como C, C++, Java, etc.) en código objeto. Por lo tanto, el lenguaje de máquina es un paso necesario para la generación de código objeto.

El lenguaje de máquina proporciona una forma de escribir instrucciones binarias que el procesador puede ejecutar directamente.

Estas instrucciones se optimizan para aprovechar al máximo el rendimiento del procesador. Esto permite que el código objeto sea más eficiente y se ejecute más rápido que el código fuente escrito en un lenguaje de alto nivel.

4.4Administración de memoria.

La administración de memoria en la generación de código objeto es un proceso crítico que se lleva a cabo durante la compilación de un programa.

En términos generales, la administración de memoria se refiere a cómo se asigna, utiliza y libera la memoria durante la ejecución del programa.

Durante la generación de código objeto, el compilador debe asegurarse de que la memoria se asigna de manera eficiente y que se utiliza de forma adecuada para evitar errores y garantizar el rendimiento del programa.

En el proceso de generación de código objeto, el compilador se encarga de convertir el código fuente del programa en un archivo ejecutable que puede ser cargado en memoria y ejecutado por la computadora. Durante este proceso, el compilador debe determinar cómo se asignará la memoria para las variables y estructuras de datos del programa, así como para las instrucciones y funciones que componen el programa.

Existen diferentes técnicas para la administración de memoria en la generación de código objeto. Una técnica comúnmente utilizada es el uso de tablas de símbolos para almacenar información sobre las variables y funciones del programa. Las tablas de símbolos permiten al compilador determinar la ubicación de cada variable y función en memoria y generar el código objeto adecuado para acceder a ellas.

Otra técnica importante en la administración de memoria en la generación de código objeto es la optimización del código. El compilador debe intentar reducir la cantidad

de memoria utilizada por el programa al eliminar código redundante y reutilizar la memoria siempre que sea posible. La optimización del código también puede mejorar el rendimiento del programa al reducir el tiempo de ejecución y el uso de la CPU y la memoria.

La administración de memoria es una técnica de gestión de memoria usada para optimizar el uso de la memoria en un sistema informático. Se refiere al proceso de asignar, administrar y liberar los recursos de memoria de manera que se optimice el rendimiento de un sistema y se minimicen los errores de memoria.

La administración de memoria es esencial para la generación de código objeto. Durante el proceso de compilación, los compiladores deben asignar memoria para almacenar los códigos objeto. Esto significa que el compilador debe asegurarse de que el código objeto se almacene en la memoria de manera segura y eficiente. Además, la administración de memoria es importante para la optimización del código.

Los compiladores optimizan el código para maximizar el uso de la memoria y para aumentar la velocidad de ejecución del código. Esto significa que los compiladores deben usar la administración de memoria para asegurarse de que el código se almacene de la manera más eficiente posible.

Por último, la administración de memoria también es importante para garantizar la seguridad del código. Los compiladores deben asegurarse de que el código objeto se almacene de manera segura, para evitar que los usuarios no autorizados puedan acceder a él. Esto significa que los compiladores deben usar técnicas de administración de memoria para asegurarse de que el código objeto se almacene de manera segura.

ACTIVIDADES DE APRENDIZAJE DE LOS TEMAS DE LA ASIGNATURA

A Recursos de Evaluación

1. ¿Qué es lo que recibe como entrada el generador de código objeto?

La representación intermedia de las instrucciones (RI).

2. ¿Para qué requiere el uso de la tabla de símbolos el generador de código objeto?

para determinar las direcciones en tiempo de ejecución de los objetos de datos denotados por los nombres en la Representación intermedia

3. ¿Por qué la generación de código óptimo es un problema NP completo?

Porque no existe una fórmula de tipo polinomial que nos indique que el código generado es el óptimo, siempre podrá haber una mejor versión.

4. ¿Cuáles son los parámetros que miden la calidad del código generado?

El tamaño y la velocidad de procesamiento

5. ¿Qué diferencia hay entre código máquina absoluto y reubicable?

El código máquina absoluto hace referencia a una dirección física en la ram y el código reubicable se refiere a una dirección lógica, esto es referencia una localidad de memoria a través del nombre y no de la dirección física

6. ¿Cuál es la parte considerada frontend, en un compilador y por qué?

La fase de síntesis se considera backend, porque en esta fase ya no se interactúa con el usuario a través de mensajes de error.

7. Dar un ejemplo de código máquina reubicable (en lenguaje ensamblador).

Mov dato,10

8. Generar el código objeto en assembler para la instrucción: IF a>b Then a:= a + 1;

MOV AH, A

CMP AH, B

JBE SALIR

INC AH

SALIR: RET

9. ¿Cómo se determinan las direcciones de los datos en tiempo de ejecución?

A través de la tabla de símbolos

10. ¿A qué se refiere el problema de assignation?

Definir en qué registros vivirán las variables elegidas en la fase allocation.

C Recursos Electrónicos de Apoyo

c1 Presentación electrónica con información que sirva de apoyo para reforzar lo visto en clase de los subtemas de la unidad temática.

c2 Archivo de power point con información sobre textos de apoyo bibliográfico, así como información condensada de los subtemas de la unidad temática.

c3 Video explicativo sobre la estructura de los diferentes tipos de salida de un compilador.

c4 Material computacional que implemente un quiz (examen), que permita a través de preguntas repasar la información de los subtemas, las preguntas se elegirán aleatoriamente de un banco de preguntas de tal forma que varíen en una misma sesión.

II. REFERENCIAS

1 Aho, A.V., Sethi, R., Ullman, J.D. (2008), Compiladores: principios, técnicas y herramientas, 2ª Edición, Editorial Pearson Education, México.

2 Barcelona Geeks, Recuperado de Árbol de expresión – Barcelona Geeks, el día 03 de febrero del 2023.

3 Fabra, Javier. Montesano, Luis. Neira, José. *Compiladores II*. Recuperado de https://webdiis.unizar.es/~neira/12048/presentacion.pdf, el día 2 de enero del 2023.

4 Lemone, K. A. (1996). Fundamentos de compiladores: cómo traducir al lenguaje de computadora. México D.F.: Compañía Editorial Continental

5 Martin, J. (2004). Lenguajes formales y teoría de la computación. México: McGraw-Hill / Interamericana de México.

6 Martínez López, Francisco Javier. Ramallo Martínez, Alejandro. (2015) Teoría, diseño e implementación de Compiladores de Lenguajes, Editorial ra-ma, España

7 Ruíz Catalán, Jacinto. (2010) Compiladores teoría e implementación. Editorial RC Libros, España.

8 Ruíz, J. (2009). Compiladores-Teoría e implementación. México, Ed. Alfaomega.

9 Vélez Reyes, Javier. (2010) Procesadores de Lenguajes. Ingeniería Técnica Superior de Ingeniería Informática. Departamento de Lenguajes y Programas Informáticos. Editorial UNED.

.

III. INSTRUMENTACIÓN DIDÁCTICA

INSTRUMENTACION
LENGUAJES Y AUTOM