



EDUCACIÓN
SECRETARÍA DE EDUCACIÓN PÚBLICA



TECNOLOGICO
NACIONAL DE MEXICO.



INSTITUTO TECNOLÓGICO NACIONAL DE MÉXICO EN CELAYA.

Materia: Lenguajes y Autómatas II

Maestro: ISC. Ricardo González González

Alumnos:

Isacc Salvador Bravo Estrada 2003048

Guillermo Peasland Aguilar 20030737

Maria del Carmen Chávez Patiño 20030296

Luis Fernando Mendoza Javalera 1930536.

► ACTIVIDAD 8 ◄

Fecha de entrega: 19 de Noviembre
de 2024.

- EQUIPO N.º 3 -



DEPARTAMENTO DE SISTEMAS COMPUTACIONALES E INFORMÁTICA

ASUNTO: **SOLICITUD DE ACTIVIDADES**

Celaya, Guanajuato, 11 / noviembre / 2024

LENGUAJES Y AUTÓMATAS II

DOCENTE DESIGNADO: ISC. RICARDO GONZÁLEZ GONZÁLEZ
SEMESTRE AGOSTO-DICIEMBRE 2024

ACTIVIDAD 8 (VALOR 35 PUNTOS)

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

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

6. OPTIMIZACIÓN.

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

- TEMA 6.1 TIPOS DE OPTIMIZACIÓN :

LOCALES, CICLOS, GLOBALES, Y DE MIRILLA.

- TEMA 6.2 COSTOS :

COSTOS DE EJECUCIÓN, (MEMORIA, REGISTROS, PILAS, ETC).

CRITERIOS PARA MEJORAR EL CÓDIGO.

HERRAMIENTAS PARA EL ANÁLISIS DEL FLUJO DE DATOS

LOCALES, CICLOS, GLOBALES, Y DE MIRILLA.





CONSIDERACIÓN :

DEBE USTED ENTENDER EL VALOR QUE TIENE ESTA ACTIVIDAD Y QUE LOS TEMAS ANTES REFERIDOS, PARA NADA DEBEN SER ABORDADOS COMO SIMPLES CONCEPTOS REDACTADOS CON LA LIGEREZA QUE YA SE HA OBSERVADO EN ACTIVIDADES PREVIAS.

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

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

- ELABORE DOS VIDEOS CORTOS (NO MÁS DE 30 MINUTOS).
- EN CADA UNO DE LOS VIDEOS DEBERÁ USAR MATERIAL DIDÁCTICO PARA EXPLICAR A FONDO Y CON DETALLE LOS TEMAS 6.1 Y 6.2.
- PONGA UN ESPECIAL ÉNFASIS EN EL PUNTO 6.1, CONCRETAMENTE EN CADA TIPO DE OPTIMIZACIÓN, MENCIONANDO SU EVOLUCIÓN E INCLUYENDO EJEMPLOS DE CÓMO MEJORAR EL CÓDIGO Y LOS COSTOS EN EL USO DE LOS RECURSOS.
- REGISTRE SUS MATERIALES EN YOUTUBE E INCLUYA LAS LIGAS EN SU EXAMEN DE MODO QUE AL HACER CLIC EN ÉSTOS SE PUEDA ACCEDER FÁCILMENTE PARA SU EVALUACIÓN.

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

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

POR ÚLTIMO, RECUERDE LEER LA GUÍA TUTORIAL PARA EL CORRECTO TRATAMIENTO DE ESTE INCISO.

7. GENERACIÓN DE CÓDIGO OBJETO.

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

- **TEMA 7.1 REGISTROS**
- **TEMA 7.2 LENGUAJE ENSAMBLADOR**
- **TEMA 7.3 LENGUAJE MÁQUINA**





CONSIDERACIÓN :

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

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

- ELABORE UN VIDEO CORTO (DE 30 MINUTOS).
- EN EL VIDEO DEBERÁ USAR MATERIAL DIDÁCTICO PARA EXPLICAR A FONDO Y CON DETALLE LOS TEMAS 7.1 AL 7.3.

REGISTRE SUS MATERIALES EN YOUTUBE E INCLUYA LAS LIGAS EN SU EXAMEN DE MODO QUE AL HACER CLIC EN ÉSTOS SE PUEDA ACCEDER FÁCILMENTE PARA SU EVALUACIÓN

¿ QUÉ SE CALIFICARÁ ?

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

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

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

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

P-E-D

ELABORACIÓN DEL **PORTAFOLIOS DE EVIDENCIA DIGITAL**. EN LA PLATAFORMA ENCONTRARÁ UNA ACTIVIDAD COMPLEMENTARIA PARA GENERAR EL P-E-D DE CADA EQUIPO DE TRABAJO, O BIEN SI TRABAJÓ DE MANERA INDIVIDUAL TAMBIÉN SERÁ NECESARIO DESARROLLAR ESTE PUNTO.





CONSIDERACIONES.

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

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

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

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

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

OBSERVACIONES:

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

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

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





LA NOMENCLATURA SOLICITADA PARA ENVIAR SU TRABAJO ES LA SIGUIENTE :

AAAA-MM-DD_TNM_CELAYA_MATERIA_DOCUMENTO_[EQUIPO]_NOCTROL_APELLIDOS_NOMBRE_SEM.PDF

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

DONDE :

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

EJEMPLO :

SI EL TRABAJO SE SOLICITÓ EN EQUIPO.

2024-11-11_TNM_CELAYA_LAI_{II}-A_A8_EQUIPO_99_9999999_PEREZ_PEREZ_JUAN_AGO-DIC24.PDF

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

SI EL TRABAJO SE SOLICITÓ INDIVIDUALMENTE.

2024-11-11_TNM_CELAYA_LAI_{II}-A_A8_9999999_PEREZ_PEREZ_JUAN_AGO-DIC24.PDF





FECHA Y HORA DE ENTREGA:

LA INDICADA EN LA PLATAFORMA VIRTUAL.

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

MUY IMPORTANTE:

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

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

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

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





EDUCACIÓN
SECRETARÍA DE EDUCACIÓN PÚBLICA



TECNOLÓGICO
NACIONAL DE MÉXICO



INSTITUTO TECNOLÓGICO NACIONAL DE MÉXICO EN CELAYA.

Materia: Lenguajes y Autómatas II

Maestro: ISC. Ricardo González González.

Alumnos:

Isacc Salvador Bravo Estrada 2003048.

Guillermo Peasland Aguilar 20030737

Maria del Carmen Chávez Patiño 20030296

Luis Fernando Mendoza Javalera 1930536

MONOGRAFÍA TÉCNICA (OPTIMIZACIÓN)

Fecha de entrega: 19 de Noviembre
de 2024

EQUIPO NO. 3



TEMA 6.1 TIPOS DE OPTIMIZACIÓN: (LOCALES, CICLOS, GLOBALES Y DE MIRILLA).

INTRODUCCIÓN

En el campo de la compilación de programas, existen diversas técnicas que permiten mejorar el rendimiento y eficiencia del código generado. Estas técnicas de optimización buscan reducir el tiempo de ejecución, disminuir el uso de memoria y maximizar el aprovechamiento de los recursos del sistema, manteniendo la corrección del programa. Entre los enfoques más comunes que se encuentran las optimizaciones locales, ciclos, globales y de mirilla.

Cada tipo de optimización tiene un enfoque y un alcance diferente. Las optimizaciones locales se aplican a pequeñas secciones del código, como un bloque básico, mientras que las optimizaciones globales afectan a grandes porciones del programa, teniendo en cuenta su estructura y flujo de control completo. Las optimizaciones de ciclos están orientadas a mejorar el rendimiento de los bucles, que suelen ser puntos críticos en la ejecución. Por último, las optimizaciones de mirilla se enfocan en realizar mejoras muy específicas y precisas en regiones pequeñas del código, identificando patrones que pueden ser reemplazados por instrucciones más eficientes.

En esta monografía, exploraremos a detalle cada uno de estos tipos de optimización, analizando sus principios, objetivos y las técnicas empleadas para su implementación.

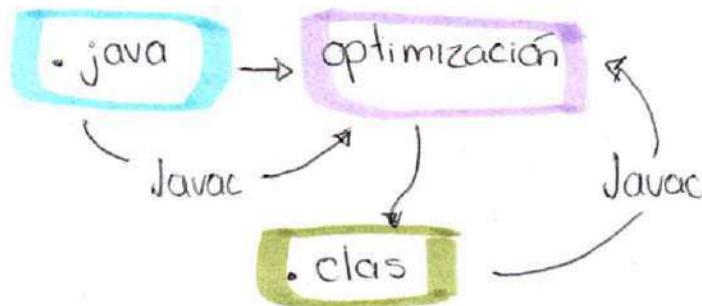


Figura 1. Optimización Diagrama



OPTIMIZACIÓN:

Las optimizaciones pueden realizarse de diferentes formas. Las optimizaciones se realizan con base al alcance ofrecido por el compilador. La optimización va a depender del lenguaje de programación y este es directamente proporcional al tiempo de compilación; es decir, entre más optimización mayor tiempo de compilación.

Como el tiempo de optimización es gran consumidor de tiempo (dado que tiene que recorrer todo el árbol de posibles soluciones para el proceso de optimización) la optimización se deja hasta la fase de prueba final. Algunos editores ofrecen una versión de depuración y otra de entrega final.

La optimización es un proceso que tiene que maximizar o minimizar alguna variable de rendimiento, generalmente tiempo, espacio, procesador, etc. Desafortunadamente no existen optimizadores que hagan un programa más rápido y que ocupe menor espacio.

La optimización se realiza reestructurando el código de tal forma que el nuevo código generado tenga mayores beneficios. La mayoría de los compiladores tienen una optimización baja, se necesita de compiladores especiales para poder realmente optimizar el código.

TIPOS DE OPTIMIZACIÓN

Optimización de Código: La optimización de código puede realizarse durante la propia generación o como paso adicional, ya sea intercalando entre el análisis semántico y la generación de código (se optimizan las cuádruplas) o situado después de ésta (se optimiza a posteriori del código generado).

Hay teoremas (Aho, 1970) que demuestran que la optimización perfecta es indecidible. Por tanto, las optimizaciones de código en realidad proporcionan mejoras, pero no aseguran el éxito total.

Clasificación de optimizaciones:

1: Dependientes de la máquina

- Asignación de registros
- Instrucciones especiales
- Reordenación del código

2: Independientes de la máquina

- Ejecución en tiempo de compilación
- Eliminación de redundancias
- Cambio de orden.
- Reducción de frecuencia de ejecución (invariancias).
- Reducción de fuerza.



Locales =

La optimización local se realiza sobre módulos del programa. En la mayoría de las ocasiones a través de funciones, métodos, procedimientos, clases, etc.

La característica de las optimizaciones locales es que solo se ven reflejados en dichas secciones.

La optimización local sirve cuando un bloque de un programa o sección es crítico por ejemplo: E/S, la concurrencia, la rapidez y confiabilidad de un conjunto de instrucciones. Como el espacio de soluciones es más pequeño la optimización local es más rápida. Como el espacio de soluciones es más pequeño la optimización local es más rápida.

Características:

La optimización local se aplica dentro de un bloque básico, es decir, una secuencia de instrucciones sin saltos. Esta se enfoca en eliminar redundancias, simplificar los cálculos y mejorar la eficiencia de fragmentos pequeños del código. Es rápida de implementar y aplicar.

Importancia:

Es útil para mejorar secciones pequeñas del código sin necesidad de un análisis exhaustivo. Aunque su impacto global es limitado, es fundamental en las primeras fases de la optimización, ya que mejora de forma inmediata partes del código.

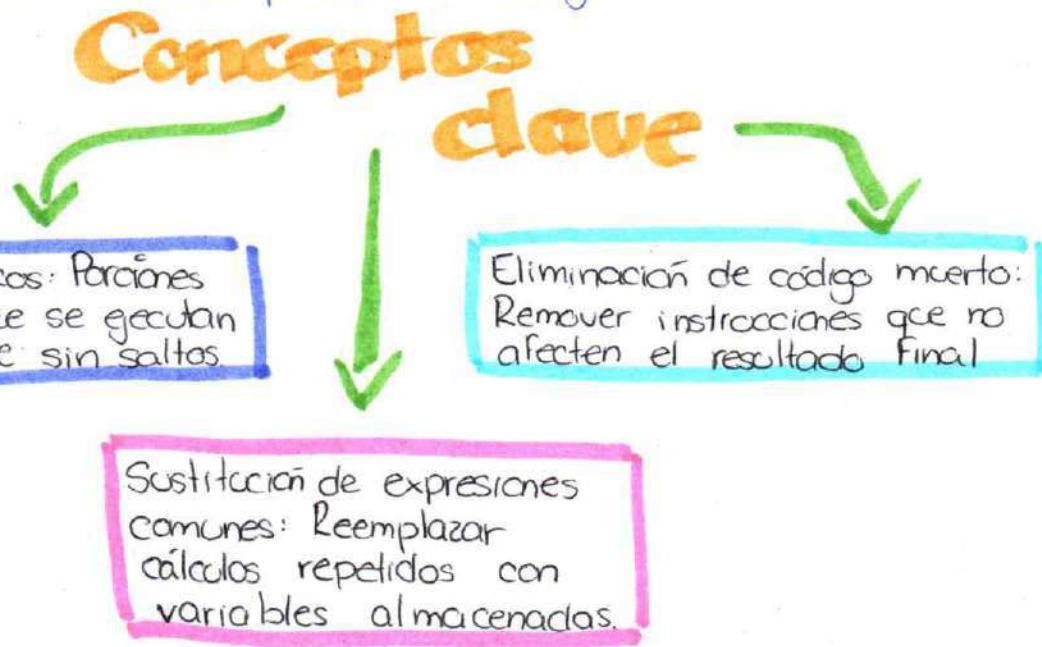


Figura 2. Locales Conceptos Clave.



Ciclos:

Los ciclos son una de las partes más esenciales en el rendimiento del programa dado que realizan acciones repetitivas, y si dichas acciones están mal realizadas, el problema se hace N veces más grande. La mayoría de las optimizaciones sobre ciclos tratan de encontrar elementos que no deben de repetirse en un ciclo.

Ejemplo:

```
while (a == b) {  
    int c = a;  
    c = 5;  
    ...}
```

En este caso es mejor pasar el `int C=a;` fuera del ciclo de ser posible. El problema de la optimización en ciclos y en general radica en que es muy difícil saber el uso exacto de algunas instrucciones. Así que no todo código de proceso puede ser optimizado. Otras de sus usos de la optimización pueden ser el seguimiento de consultas en SQL o en aplicaciones remotas (Sockets, E/S.)

Características:

Se centra en mejorar la eficiencia de los bucles, que suelen ser puntos críticos del código. Técnicas comunes incluyen el desenrollado de bucles, paralelización y vectorización, las cuales buscan reducir la cantidad de iteraciones o mejorar la forma en que el hardware ejecuta las instrucciones dentro de un bucle.

Importancia:

Los ciclos representan las partes más costosas en muchos programas, por lo que optimizarlos puede reducir drásticamente el tiempo de ejecución. Es clave en aplicaciones donde los bucles dominan el uso de recursos, como simulaciones o procesamiento de datos.

Conceptos Clave

Desenrollado de bucles: Aumentar el tamaño del cuerpo del bucle para reducir el número de iteraciones.

Paralelización: Ejecutar iteraciones del bucle en paralelo utilizando múltiples núcleos de la CPU.

Vectorización: Procesar múltiples datos en una sola instrucción utilizando las capacidades SIMD.

Figura 3. Ciclos
Conceptos Clave



Globales:

La optimización global se da con respecto a todo el código. Este tipo de optimización es más lenta pero mejora el desempeño general de todo programa. Las optimizaciones globales pueden depender de la arquitectura de la máquina.

En algunos casos es mejor mantener variables globales para agilizar los procesos (el proceso de declarar variables y eliminarlas toma su tiempo) pero consume más memoria. Algunas optimizaciones incluyen utilizar como variables registros del CPU, utilizar instrucciones en ensamblador.

Características:

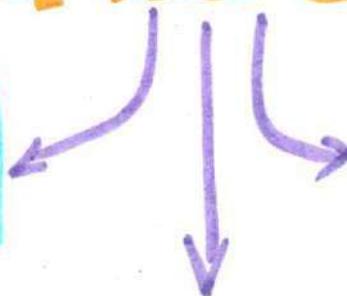
Abarca todo el programa y considera la interacción entre bloques básicos y funciones. Analiza el flujo de control y de datos a lo largo del programa completo, lo que le permite optimizar de manera más integral, como eliminar redundancias que no son evidentes localmente.

Importancia:

La optimización global es crucial para mejorar el rendimiento general del programa, ya que considera dependencias y oportunidades que las optimizaciones locales no pueden detectar. Sin embargo, es más costosa en términos de tiempo de compilación y complejidad.

Conceptos clave

Propagación de constantes:
Reemplazar el valor de una variable cuando es constante a lo largo del programa.



Análisis del flujo de datos:
Examinar cómo las variables y los valores influyen entre diferentes partes del código.

Eliminación de código no alcanzable:
Remover bloques de código que nunca se ejecutan debido a la lógica del programa.

Figura 4. Globales Conceptos Clave



Mirilla =

La optimización de mirilla trata de estructurar de manera eficiente el flujo del programa, sobre todo en instrucciones de bifurcación como son las decisiones, ciclos y saltos de rutinas.

La idea es tener los saltos lo más cerca de las llamadas, siendo el salto lo más pequeño posible.

Instrucciones de bifurcación

Interrompen el flujo normal de un programa, es decir que evitan que se ejecute alguna instrucción del programa y salta a otra parte del programa.

Por ejemplo el Break

Switch (expresión que estamos evaluando)

{

Case 1 : cout << "Hola";
Break;

Case 2 : cout << "amigos";
Break;

}

Características:

La optimización de mirilla, también llamada "peephole optimization" se aplica a pequeñas ventanas del código (generalmente un par de instrucciones) y busca reemplazar patrones inefficientes por versiones más óptimas. Es una técnica simple y precisa que se realiza a nivel de instrucción.

Importancia:

Aunque su impacto global es menor, es útil para mejorar detalles que otros tipos de optimización pueden pasar por alto. Se suele aplicar al final del proceso de compilación, actuando como un pulido final para el código.



Conceptos Clave =

Bloques básicos : Porciones del código que se ejecutan secuencialmente sin saltos.

Eliminación de código muerto: Remover instrucciones que no afectan el resultado final.

Sustitución de expresiones comunes: Reemplazar cálculos repetidos con variables almacenadas.

Figura 5. Mirilla Conceptos Clave.

Ventajas y desventajas.

Tipo de optimización	Ventajas	Desventajas
Optimización Local	<ul style="list-style-type: none"> • Simplicidad en su implementación • Rápida aplicación • Impacto inmediato en pequeñas secciones del código. 	<ul style="list-style-type: none"> • Alcance limitado, solo a bloques básicos. • No considera interacciones globales. • Eficiencia subóptima a nivel de programa completo.
Optimización de Ciclos	<ul style="list-style-type: none"> • Mejora significativa en ciclos, que son críticos en rendimiento. • Reduce el tiempo de ejecución de bucles. • Aumenta la eficiencia de la CPU. 	<ul style="list-style-type: none"> • Técnicas complejas como paralelización. • Solo aplicable a bucles. • Puede aumentar el tamaño del código (ej. desenrollando bucles.)
Optimización Global	<ul style="list-style-type: none"> • Alcance amplio en el programa completo. • Eficiencia mejorada a nivel global 	<ul style="list-style-type: none"> • Alto costo computacional durante la compilación. • Implementación compleja.



	<ul style="list-style-type: none"> Optimización de variables vivas y uso de registros entre bloques. 	<ul style="list-style-type: none"> Cambios globales pueden tener efectos colaterales no deseados.
Optimización de Mirilla.	<ul style="list-style-type: none"> Alta precisión en ventanas pequeñas de código. Facil de implementar Elimina redundancias y combinaciones inefficientes de instrucciones. 	<ul style="list-style-type: none"> Impacto limitado en el rendimiento general. No considera el contexto global del programa. Solo optimiza patrones simples y evidentes.

Tabla 1 Ventajas y Desventajas.

Ejemplos prácticos:

Optimización de Mirilla:

Reemplazar un conjunto de instrucciones inefficientes por una sola instrucción ensamblador optimizada.

Contexto: Una pequeña ventana de código puede contener operaciones redundantes o inefficientes que deben simplificarse.

Código original:

```
MOV AX, a ; cargar valor de a
ADD AX, 0 ; sumar 0 (innecesario)
MOV b, AX ; Guardar en b
```

Código Optimizado:

```
MOV AX, a ; Cargar valor de a directamente.
MOV b, AX ; Guardar en b sin suma innecesaria.
```

Beneficio:

Se elimina la instrucción ADD AX, 0, que no tiene impacto en el resultado final pero ocupa ciclos de CPU.

Optimización local:

Sustitución de una operación como $a * 2$ por $a \ll 1$ en ensamblador.

Contexto: En lenguajes de bajo nivel como ensamblador, multiplicar por potencias de 2 puede ser sustituido por un desplazamiento de bits hacia la izquierda, ya que cada desplazamiento equivale a multiplicar por 2.



Código Original:

```
MOV AX, a ; Cargar valor de a  
MUL 2 ; Multiplicar por 2  
MOV b, AX ; Guardar resultado en b.
```

Beneficio:

La instrucción SHL es más rápida que MUL, lo que reduce el tiempo de ejecución.

Optimización por ciclos:

Aplicar desenrollado de un bucle for para procesar múltiples elementos de cada iteración.

Contexto:

En lugar de iterar una vez por cada elemento, se puede procesar varios elementos en una sola iteración para reducir la sobrecarga del bucle.

Código original (sin desenrollar):

```
for (int i = 0; i < 8; i++) {  
    arr[i] = arr[i] * 2;  
}
```

Código optimizado (desenrollado):

```
for (int i = 0; i < 8; i += 2) {  
    arr[i] = arr[i] * 2;  
    arr[i + 1] = arr[i + 1] * 2;  
}
```

Beneficio:

Se reduce el número de comprobaciones y actualizaciones del índice del bucle, lo que mejora el rendimiento, especialmente en bucles largos.

Optimización global:

Propagación de constantes en funciones que interactúan entre sí.

Contexto:

Si se sabe que una variable tiene un valor constante en todo el programa, este valor puede ser propagado directamente para eliminar cálculos innecesarios.

Código original:

```
int add(int x, int y) {  
    return x + y;  
}  
int main() {  
    int a = add(5, 3); // Llamada con valores const.  
    printf("%d", a);  
}
```

Código optimizado:

```
int main() {  
    int a = 8; // Reemplaza la llamada a la fun.  
    printf("%d", a);  
}
```



Beneficio:

Se elimina completamente la necesidad de llamar a la función add, reduciendo el tiempo de ejecución y el uso de recursos.

Conclusión =

Los diferentes tipos de optimización -locales, de ciclos, globales y de mirilla- son herramientas esenciales en el proceso de mejora del rendimiento del código, cada una con su propio enfoque, alcance y beneficios. La optimización local y de mirilla se enfocan en áreas específicas de ciclos y la global permiten abordar aspectos más amplios y críticos del programa, como el rendimiento de bucles intensivos y la interacción entre distintas partes del código.

Cada tipo de organización desempeña un rol importante según el contexto: mientras que optimizaciones locales y de mirilla se implementan con facilidad y en menor tiempo, las globales y de ciclos requieren análisis más complejos, pero producen beneficios significativos en el rendimiento global. Esto resalta la importancia de combinar estas técnicas de manera estratégica durante el desarrollo y compilación de programas.

En pocas palabras, la optimización del código no solo busca un mejor rendimiento, sino también un equilibrio entre el costo de implementación y los beneficios obtenidos. Entender estas técnicas y aplicarlas correctamente es crucial para diseñar programas eficientes y adaptables a la demandas modernas de rendimiento.

Referencias:

- Cs/f). Itpn.mx . Recuperado el 17 de noviembre de 2024, de <http://itpn.mx/recursosisc/7semestre/lenguajesyautomatas2/Unidad%20III.pdf>.
- Guest (2020, noviembre 24). Unidad 3 y 4. Docx. Pdfcoffee.com /unidad-3y4.docx-3-pdf-free.html
- De compilación LOPR de FLC se R en B al A OP el CLO va (S/f) Un VII . Optimización . Wordpress.com . Recuperado el 17 de Noviembre de 2024 , de <https://ingarely.wordpress.com/wp-content/uploads/2012/11/Unidad-vii.pdf>

CRITERIOS PARA MEJORAR EL CÓDIGO DE INTERFAZ.

La mejora del código en lenguajes de interfaz no solo indica e implica garantizar funcionalidad, sino también optimizar la eficiencia, reducir los costos de ejecución y hacer que el mantenimiento del código sea más manejable. A continuación, se desarrollan subtemas con mayor detalle, y ejemplos que ilustran su implementación.

1. OPTIMIZACIÓN ALGORÍTMICA Y ESTRUCTURAL

1.1 Selección de algoritmos eficientes.

La complejidad computacional de un algoritmo impacta directamente el tiempo y los recursos necesarios para ejecutar un programa. Seleccionar algoritmos óptimos para tareas específicas es esencial en aplicaciones críticas.

• Ejemplo de Ordenamiento:

Cambiar de un algoritmo de burbuja ($O(n^2)$) a uno como Quicksort ($O(n \log n)$) en promedio reduce el tiempo para grandes volúmenes de datos.

Ineficiente

```
for i in range(len(arr)):  
    for j in range(len(arr)-i-1):  
        if arr[j] > arr[j+1]:  
            arr[j], arr[j+1] = arr[j+1], arr[j]
```

Eficiente

```
arr.sort()
```

1.2 Rediseño de estructuras.

El diseño ineficiente de estructuras de datos puede causar redundancia y tiempos de acceso elevados. Por ejemplo, reemplazar una lista por un diccionario mejora busquedas clave-valor ($O(1)$) frente a $O(n)$.

1.3 Técnicas de eliminación de Recursividad

La recursividad profunda puede ocasionar un uso excesivo de pila. Transformar algoritmos recursivos e iterativos evita estos problemas.

Ejemplo: Factorial recursivo Vs Iterativo:

Recursivo

```
def factorial(n):  
    if n == 0:  
        return 1  
    return n * factorial(n-1)
```

Iterativo

```
def factorial(n):  
    result = 1  
    for i in range(1, n+1):  
        result *= i  
    return result.
```

2. MINIMIZACIÓN DE ACCESOS A MEMORIA

2.1 LOCALIDAD DE DATOS

El acceso eficiente a memoria caché aprovecha la localidad espacial y temporal de datos. Acceder secuencialmente a arreglos o estructuras contiguas mejora significativamente el rendimiento.

• Ejemplo de localidad espacial:

// Acceso ineficiente

```
for (int i = 0; i < cols; i++) {
    for (int j = 0; j < rows; j++) {
        matrix[j][i] = i + j;
    }
}
```

// Acceso eficiente

```
for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
        matrix[i][j] = i + j;
    }
}
```

2.2 Uso de registros en lugar de memoria

Principal

Al asignar variables a registros de procesador, se minimiza el tiempo de acceso. Esto se logra mediante optimizaciones automáticas de compiladores como gcc o clang.

2.3 Eliminación de operadores redundantes en accesos de Memoria.

Evitar acceder repentinamente al mismo espacio en memoria almacenando temporalmente los resultados en variables locales.

3. DESARROLLO DE CICLOS

3.1 Desenrollado Manual.

Reestructurar bucles para ejecutar varias iteraciones en un solo ciclo puede reducir el número de instrucciones de control.

Ejemplo:

// Sin desarrollar

```
for (int i=0; i<n; i++){
    array[i] += 1;
```

// Desenrollado Manual

```
for (int i=0; i < n; i+=4){  
    array[i] += 1;  
    array[i+1] += 1;  
    array[i+2] += 1;  
    array[i+3] += 1;
```

3.2 Paralelización De Bucles

Descomponer ciclos para ejecutarlos en múltiples hilos o procesadores, eliminando dependencias cruzadas. Esto es particularmente efectivo en lenguajes con soporte multithreading, como Java o python con multiprocessing.

3.3 Fusionado y tiling De Bucles

Fusionar bucles reduce sobrecargas de control al combinar operaciones similares en un único ciclo. Por otro lado, el tiling mejora el acceso a caché dividiendo bucles en bloques más pequeños.

4 ELIMINACIÓN DE REDUNDANCIA

4.1 Código Muerto

El código que no se ejecuta o cuya salida no afecta al programa debe eliminarse. Herramientas de análisis estático detectan estas secciones automáticamente.

4.2 Factorización de expresiones comunes.

Reutilizar resultados de cálculos repetitivos evita operaciones innecesarias.

Ejemplo:

// Redundante

int a = b + c;

int a = b * c + e;

// Optimizado

int temp = b * c;

int a = temp;

int d = temp + e;

HERRAMIENTAS PARA EL ANÁLISIS DEL FLUJO DE DATOS

El análisis del flujo de datos en lenguajes de interfaz implica identificar cómo las variables y datos se propagan a través de un programa, desde su declaración hasta su uso final. Este análisis es crucial para mejorar la eficiencia, detectar errores y aplicar optimizaciones avanzadas. A continuación, se presentan diferentes tipos y niveles de análisis y técnicas relacionadas, cada uno con ejemplos y un desarrollo exhaustivo.

1. ANÁLISIS LOCAL

El análisis local se centra en bloques básicos de código, donde no hay bifurcaciones o saltos. Este nivel identifica cómo las variables definen y usan dentro de un segmento cerrado, ayudando a minimizar el consumo innecesario de recursos.

VARIABLES VIVAS

El análisis de variables vivas determina cuáles de ellas son necesarias en un momento dado de la ejecución. Las variables que ya son o no son utilizadas pueden ser eliminadas o reutilizadas.

Ejemplo Práctico:

// Sin optimizar

```
int x=10;
```

```
int y=x+5;
```

// 'x' ya no se utiliza aquí.

// Optimizado

```
int y=15; // 'x' es eliminado
```

```
z=y*2;
```

Cadenas Def-Use

Estas cadenas rastrean cada definición de una variable y sus usos posteriores, facilitando la identificación de dependencias y optimizaciones potenciales.

Ejemplo:

En un bloque de código donde se define una variable X y se usa posteriormente en varias expresiones, las cadenas Def-Use ayudan a determinar si esas expresiones pueden ser simplificadas o fusionadas.

2. Análisis Cíclico

El análisis cíclico se enfoca en estructuras iterativas, como bucles, que suelen ser responsables de una gran parte del tiempo de ejecución de un programa. La optimización de bucles puede tener un impacto significativo en el rendimiento.

Análisis De Dependencias.

Este análisis evalúa si las iteraciones de un bucle dependen unas de otras. Si no hay dependencias, es posible ejecutar las iteraciones en paralelo, aumentando la eficiencia en sistemas multicore.

Ejemplo:

// Dependencia Presente

```
for (int i=1; i<n; i++) {
```

```
    array[i] = array[i+1] + 1; // Cada iteración  
    // depende de la anterior.
```

```
}
```

// Sin dependencias

```
for (int i=0; i<n; i++) {
```

```
    array[i] = i*2; // Las iteraciones son independientes
```

```
}
```

Paralelización de Búdes

La paralelización de búdes es una técnica avanzada que busca dividir la ejecución de iteraciones en múltiples hilos o procesadores, aprovechando los recursos paralelos de hardware.

Esta técnica es especialmente útil para tareas intensivas en datos, como el procesamiento de imágenes, simulaciones científicas o cálculos matemáticos complejos.

Características de los bucles paralelizables

1. Independencia de Iteraciones

Para que un bucle sea paralelizable, cada iteración debe ser independiente de las demás, es decir, no debe haber dependencias cruzadas entre iteraciones.

Bucle Paralelizable

```
for (int i=0; i < n; i++)  
{ array[i] = array[i]*2; // Es independiente }
```

Bucle No Paralelizable

```
for (int i=1; i < n; i++)  
{ array[i] = array[i] + array[i-1]; // Es dependiente }
```

2. Granularidad Del Trabajo

La parallelización es más efectiva cuando las iteraciones realizan suficiente trabajo para compensar la sobrecarga de gestionar múltiples hilos.

Implementación Con Herramientas.

1 Open MP:

Open MP es una herramienta o más bien una API ampliamente utilizada para parallelización en CPUs. Proporciona directivas simples que permiten transformar un bucle Secuencial en paralelo.

2 CUDA:

CUDA es una plataforma de parallelización diseñada para ejecutar bucles en GPUs aprovechando su capacidad de manejar miles de hilos en paralelo.

Transformación De Bucles

La transformación de bucles engloba diversas técnicas destinadas a mejorar el rendimiento y la eficiencia de ejecución de un programa. Estas transformaciones suelen aplicarse en lenguajes de bajonivel o a través de compiladores optimizados.

Fusión de Bucles.

La fusión combina multiples bucles independientes en uno solo, reduciendo la Sobrecarga de control y mejorando la localidad de datos:

Ejemplo sin fusión:

```
for (int i=0; i<n; i++) {  
    array[i] = array[i] + 1;  
}  
  
for (int i=0; i<n; i++) {  
    array2[i] = array2[i] * 2;  
}
```

Ejemplo con Fusión

```
for (int i=0; i<n; i++) {  
    array1[i] = array1[i] + 1;  
    array2[i] = array2[i] * 2;  
}
```

Ventajas

- Reduce la sobrecarga de control al procesar ambas operaciones en un mismo bucle.
- Mejora la localidad espacial en la memoria al acceder a elementos gráficos o contiguos en Secuencia.

3. ANÁLISIS GLOBAL

El análisis global considera el programa completo, incluyendo múltiples funciones y módulos. Este enfoque es esencial para identificar optimizaciones que trascienden en un solo bloque o ciclo de código.

ANALISIS INTERPROCEDIMENTAL

Evaluá como los datos fluyen entre funciones o procedimientos, ayudando a identificar efectos secundarios y optimizar el uso de variables globales.

Ejemplo:

```
int global_x; // variable global  
void funcionA(){  
    global_x = 5; // Modifica variable global  
}  
void funcionB(){  
    int local_y = global_x + 2; // Usa el valor global.
```

El análisis interprocedimental detectaría si global-x puede convertirse en una variable local o si se pueden reducir las interdependencias entre las funciones.

Análisis de Alias.

Determina si diferentes referencias apuntan al mismo espacio de memoria, lo que puede llevar a inconsistencias o a la Sobrecarga de recursos.

Ejemplo Práctico:

```
int *p = &x;  
int *p = &x; //alias de P.
```

```
*P = 10; //Cambia el valor al que apunta q.
```

4. Análisis De Mirilla

Este análisis se realiza en ventanas pequeñas de código (generalmente unas pocas instrucciones) para identificar oportunidades inmediatas de optimización.

Simplificación de Ecuaciones

Identifica operaciones que pueden ser reemplazadas por alternativas más eficientes. Por ejemplo, en lugar de realizar una división, es posible usar multiplicaciones por el recíproco en algunas arquitecturas.

Ejemplo:

// Sin optimizar

$x = y / 8;$

// optimizado

$x = y * 0.125;$ // Más rápido en ciertos procesadores.

Eliminación de Código Redundante (Instrucciones Redundantes)

Este análisis elimina instrucciones duplicadas o que no afectan el resultado del programa.

Ejemplo práctico:

```
int a = 5;
```

```
int b = a * 2; // b = 10
```

int c = a * 2; // Redundante, B ya vale eso.

Reemplazo de Saltos Costosos

Se optimizan saltos condicionales costosos, como aquellos que interrumpen la ejecución secuencial.

Ejemplo:

```
// Sin optimizar
```

```
if (x > 0) {
```

```
    y = x;
```

```
} else {
```

```
    y = 0;
```

// Optimizado

```
y = (x > 0) ? x : 0; // Menor sobrecarga de control.
```

Referencias

- Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). Compilers: Principles, techniques, and tools (2nd ed.). Pearson.
- Muchnick, S. S. (1997). Advanced compiler design and implementation. Morgan Kaufmann.
- Zaki, M. J., & Meira, W. (2014). Data mining and analysis: Fundamental concepts and algorithms. Cambridge University Press.
- Smith, A. J. (1982). "Cache memories." ACM Computing Surveys (CSUR), 14(3), 473-530.
<https://doi.org/10.1145/356887.356892>
- Wolfe, M. (1996). High-performance compilers for parallel computing. Addison-Wesley.

INSTITUTO TECNOLÓGICO NACIONAL DE MÉXICO EN CELAYA

Materia: Lenguajes y Autómatas II

Maestro: ISC. Ricardo González González

Alumnos:

Isacc Salvador Bravo Estrada 2003048

Guillermo Peasland Aguilar 20030737

Maria del Carmen Chávez Patiño 20030296

Luis Fernando Mendoza Javalera 1930536

Monografía Técnica

Generación de código objeto

Fecha de entrega: 19 de Noviembre de
2024

EQUIPO N° 3

Generación de código objeto

Introducción

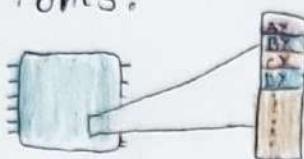
La generación de código objeto constituye una de las etapas finales en el proceso de compilación, en la cual el código fuente, tras ser analizado y optimizado, se traduce a un formato que puede ser ejecutado directamente por el procesador. Este proceso es crucial, ya que define la relación entre el software y el hardware, asegurando la correcta interpretación y ejecución de las instrucciones programadas.

En este contexto, los registros, el lenguaje ensamblador y el lenguaje Máquina desempeñan un papel fundamental. Los registros actúan como elementos de almacenamiento rápido en el procesador, facilitando la ejecución eficiente de operaciones aritméticas y de control. Por su parte, el lenguaje ensamblador se sitúa como un puente entre el código de alto nivel y el lenguaje Máquina, permitiendo una representación más comprensible para los desarrolladores. Finalmente, el lenguaje Máquina, compuesto por instrucciones binarias, representa el nivel más bajo de interacción con el hardware.

Registros y Arquitectura del procesador.

Los registros son pequeños espacios de almacenamiento ubicados dentro del procesador, diseñados para guardar datos de manera temporal durante la ejecución de un programa. A diferencia de la memoria principal, los registros ofrecen una velocidad de acceso significativamente más alta, lo que los convierte

en recursos clave para la eficiencia en la ejecución de instrucciones.



Tipos de Registros

Existen diferentes tipos de registros, cada uno con funciones específicas:

- **Registros de propósito general:** Utilizados para almacenar datos temporales y resultados intermedios de operaciones aritméticas o lógicas.
- **Registros de propósito específico:** incluyendo registros como el contador de programa (PC), que rastrea la próxima instrucción a ejecutar, y el registro de estado (SR), que almacena información sobre el estado actual del procesador.
- **Registros de segmento y punteros:** usados para operaciones relacionadas con la administración de memoria, como acceder a datos en diferentes segmentos de memoria.

Impacto de la arquitectura en la generación de código

La arquitectura del procesador determina la cantidad, tipo y usos de los registros disponibles. En arquitecturas RISC (Reduced Instruction Set Computing) por ejemplo, se enfatiza el uso de múltiples registros para optimizar el rendimiento mediante instrucciones simples y rápidas. Por otro lado, en arquitecturas CISC (Complex Instruction

Set Computing), las instrucciones pueden ser más complejas con un menor número de registros necesarios.

La generación de código objeto debe considerar estas características arquitectónicas para producir instrucciones eficientes. Por ejemplo en la arquitectura RISC, es crítico realizar una asignación óptima de registros, mientras que en un procesador CISC es más relevante aprovechar al máximo las instrucciones complejas.

Lenguaje Ensamblador.

El lenguaje ensamblador es una representación simbólica de las instrucciones del lenguaje máquina, diseñado para ser más comprensible para los programadores. Aunque sigue siendo un lenguaje de bajo nivel, su uso facilita el desarrollo de programas al permitir el uso de nombres y etiquetas en lugar de instrucciones binarias puras.

Definición y características principales.

El ensamblador traduce las instrucciones del código objeto en un formato que es más legible para los humanos utilizando símbolos y abreviaturas para representar operaciones, registros y direcciones de memoria. Entre sus características destacan:

- Su dependencia directa de la arquitectura del procesador.
- La necesidad de un ensamblador, un programa que convierte el código ensamblador en la lenguaje máquina.
- La posibilidad de acceder y controlar directamente los registros del procesador y la memoria.

Traducción del código intermedio al ensamblador.

En el proceso de compilación, el código intermedio es transformado en ensamblador como una etapa previa a su conversión al lenguaje máquina. Este paso incluye:

- 1.- Asignación de registros: Seleccionar qué registros almacenarán variables o resultados intermedios.
- 2.- Generación de instrucciones: Traducir operaciones en instrucciones específicas de ensamblador.
- 3.- Optimización: Reducir la cantidad de instrucciones generadas para mejorar el rendimiento.

Ejemplo básico

En el siguiente ejemplo se muestra como se traduce una operación en código intermedio al lenguaje ensamblador:

$$\begin{array}{l} t_1 = a + b \\ c = t_1 * b \end{array} \quad \text{código intermedio}$$

;

$$t_1 = a + b$$

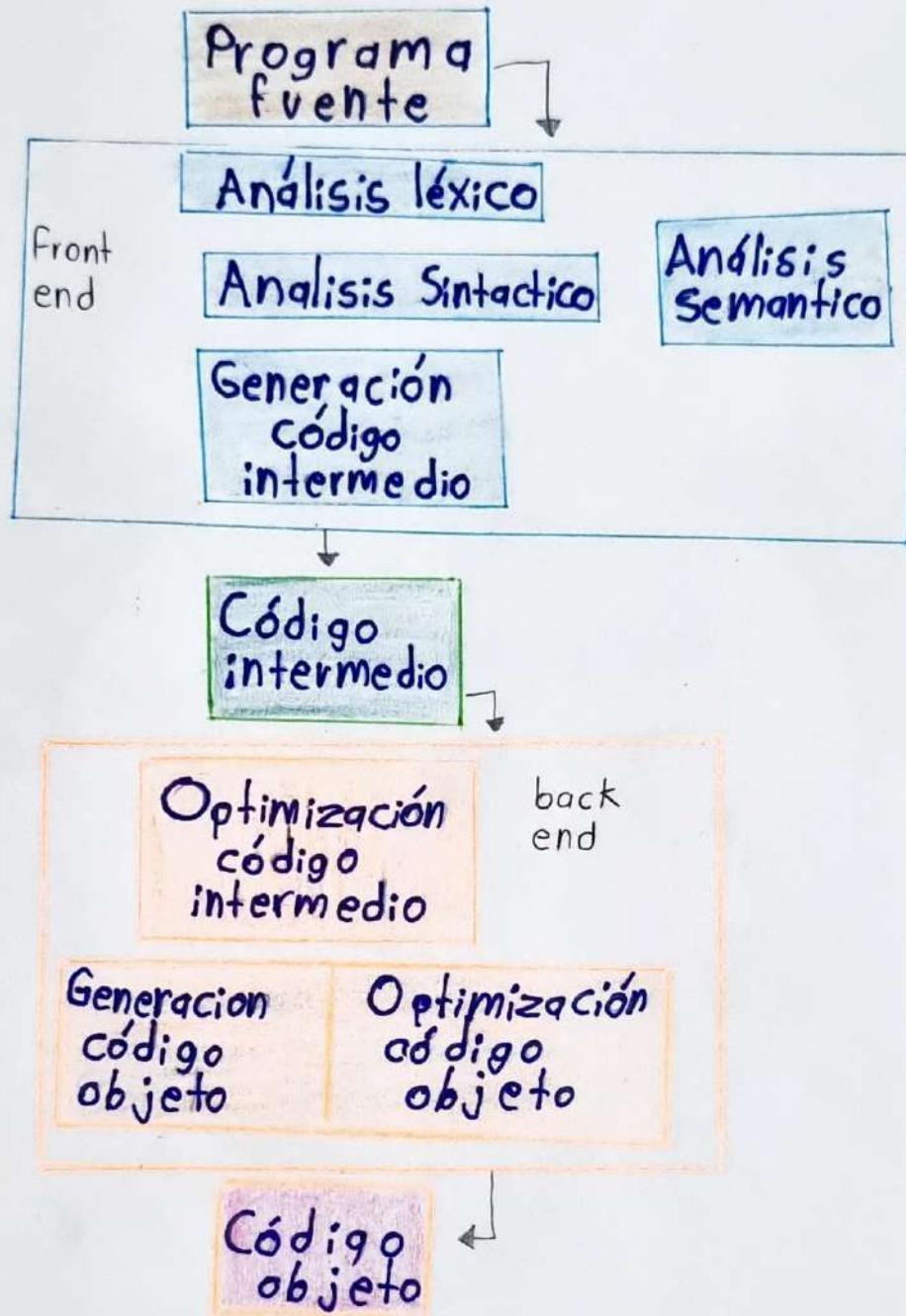
```
MOV AL, A
ADD AL, B
Mov t1, AL
```

Código en lenguaje ensamblador

;

$$c = t_1 * d$$

```
MOV AL, t1
MUL D
MOV C, AL
```



organización de las fases en
front end y back end

Lenguaje Máquina

El lenguaje máquina representa el nivel más bajo en la jerarquía de lenguajes de programación, compuesto exclusivamente por instrucciones binarias que el procesador puede ejecutar directamente. Estas instrucciones están diseñadas específicamente para una arquitectura de hardware, lo que los hace poco portables y difíciles de interpretar para los humanos.

¿Qué es el lenguaje máquina?

El lenguaje máquina consiste en secuencias de bits organizados en instrucciones, las cuales especifican operaciones a realizar, datos a procesar y registros o direcciones de memoria a utilizar.

Cada procesador tiene su propio conjunto de instrucciones (ISA, Instruction Set Architecture), que define las operaciones soportadas y su representación en binario.

Representación binaria de instrucciones

Las instrucciones en el lenguaje máquina están formadas por diferentes campos:

- **Opcode (Código de operación):** Especifica la operación a realizar (por ejemplo, suma, resta, etc)
- **Operando(s):** indican los datos o registros involucrados en la operación.
- **Modificadores:** Especifican modos de direccionamiento, tamaños de datos o condiciones adicionales.

Ejemplo 00000011 11001001
 ↓ ↓
 instrucción señaliza
 ADD en registro CL y CX

Relación entre ensamblador y lenguaje máquina

El lenguaje ensamblador simplifica la escritura de instrucciones al utilizar mnemonics (abreviatura para representar operaciones, que posteriormente son traducidas al formato binario del lenguaje máquina)

Ejemplo

Ensamblador ADD AX, BX

Máquina	0001_1101	1000
	01	D
		8

La generación del código máquina implica traducir cada instrucción en ensamblador a su formato binario correspondiente, lo que requiere conocer la arquitectura específica del procesador y el esquema de codificación de sus instrucciones.

Optimización del Código Objeto

La optimización del código objeto busca generar un código más eficiente en términos de tiempo de ejecución y uso de recursos. Al transformar el código intermedio en código objetos, se pueden aplicar diversas técnicas para mejorar su rendimiento.

Uso eficiente de registros

Los registros son recursos valiosos en un procesador, ya que permiten un acceso a la información mucho más rápido que la memoria principal. Un uso eficiente de los registros puede reducir significativamente el número de accesos a la memoria.

- Asignación de registros: Un buen algoritmo de asignación de registros puede minimizar el número de veces que un valor debe ser cargado o almacenado en memoria.

• Spill y fill: Cuando hay más variables en uso que registros disponibles, se utiliza la técnica de spill para almacenar temporalmente un valor en la pila y fill para cargarlo en un registro cuando sea necesario.

Ejemplo con 8086

Supongamos que tenemos un fragmento de código en ensamblador 8086 que realiza una suma:

```
MOV AX, [Num1]  
MOV BX, [Num2]  
ADD AX, BX  
MOV [Resultado], AX
```

Podemos optimizar este código asignando los valores de Num1 y Num2 directamente a registros:

```
MOV AX, [Num1]  
ADD AX, [Num2]  
MOV [Resultado], AX
```

Al eliminar una instrucción MOV, se reduce el número de ciclos de reloj necesarios para ejecutar el código.

Ejemplo

Calcular el factorial de un número.

Supongamos que el número a calcular el factorial
está en BX

MOV AX, 1 ; iniciamos el resultado en 1

factorial:

CMP BX, 1

JE Fin

MUL BX

DEC BX

JMP factorial

Fin:

el resultado está en AX

Solución optimizada

Supongamos que el número a calcular el factorial
está en BX

MOV AX, 1 ; iniciamos el resultado en 1

factorial:

CMP BX, 2

JE Fin

MUL BX

DEC BX

JMP factorial

Fin:

el resultado está en AX

Esta optimización se aprovecha el hecho de que
el factorial de 1 es 1, por lo que no es necesario
realizar la multiplicación en ese caso.

Otras técnicas de optimización

- Planificación de instrucciones: Ordenar las instrucciones para aprovechar al máximo las unidades funcionales del procesador.
- Eliminación de código muerto: Eliminar código que nunca se ejecuta.
- Plagado de constantes: Se trata de evaluar en tiempo de compilación expresiones que solo involucran constantes, obteniendo así un resultado constante que puede ser utilizado directamente en el código generado.
- Eliminación de sub-expresiones comunes: Identifica subexpresiones que aparecen múltiples veces en el código y las calculan una sola vez, almacenando el resultado en un registro.
- Introducción de código en línea: Sustituye llamadas a funciones pequeñas por el código de la función directamente en el punto de llamada evitando la sobre carga de llamadas de función.

Limitaciones de la optimización

Es importante tener en cuenta que la optimización del código tienen ciertas limitaciones:

- Complejidad computacional: Algunas técnicas de optimización pueden ser computacionalmente costosas, especialmente para programas grandes.
- Compromiso entre rendimiento y legibilidad. La optimización ejecutiva puede dificultar la lectura y mantenimiento del código.

- Dependencia de la arquitectura: Las técnicas de optimización pueden variar significativamente entre diferentes arquitecturas de procesador.

Conclusiones

La generación de código objeto es un proceso fundamental en la compilación de programas, que involucran la transformación de código intermedio en un formato ejecutable por la máquina. Comprender los detalles de este proceso, desde la asignación de registros hasta la optimización del código, es esencial para cualquier desarrollador que desee escribir software eficiente.

El conocimiento de la generación de código objeto proporciona al programador una visión más profunda de cómo funciona el compilador y cómo se ejecuta su código. Esto permite:

- Escribir código más eficiente: Al entender cómo se optimiza el código, se puede tomar decisiones de diseño que favorecen la generación de código más rápido y compacto.
- Depurar programas de manera más efectiva: Identificar errores en el código generado puede ser más fácil si se comprende cómo se traduce el código fuente al lenguaje máquina.
- Aprovechar las características específicas de la arquitectura: Conocer la arquitectura del procesador objetivo permite escribir código que se adapte mejor a sus capacidades.

Referencias

Aho, A. V. (2000). Compiladores: Principios, tecnicas y herramientas. Addison Wesley Publishing Company.

Garrido Alenda, A. I. (2002). Diseno de Compiladores. Digitalia- Universidad de Alicante.

Hennessy, J. L., & Patterson, D. A. (2012). Computer architecture : a quantitative approach. Morgan Kaufmann.

Links a videos

<https://youtu.be/bXcnC6tLyFA>

<https://youtu.be/-gWMZZrQsFU>

<https://youtu.be/CwxlWTjekSs>

Link a portafolio de evidencias

<https://drive.google.com/file/d/1izY8s-pkOHWLZLpW5gblai8j3A0zxvhX/view?usp=sharing>

2023
MAYO
JUNIO
JULIO
AGOSTO
SEPTIEMBRE
OCTUBRE
NOVIEMBRE
DICIEMBRE