Compiladores:
Generación de Código

Pontificia Universidad Javeriana Cali
Ingeniería de Sistemas y Computación
Prof. María Constanza Pabón

Generación de Código

- Mantener la semántica del programa
- El programa traducido debe tener alta calidad:
  - Uso efectivo de los recursos
  - Correr eficientemente = “buen código” (generar la traducción óptima es un problema indecidible)
- Producir código correcto

Generador de Código

- Tres tareas:
  - Selección de las instrucciones
  - Localización de Registros y asignación
  - Ordenar las instrucciones

- Depende de:
  - Representación Intermedia
  - Lenguaje Destino
  - Run-time system

Generador de Código

- Conjuntos de instrucciones y arquitecturas de máquina usuales:
  - RISC (Reduced Instruction Set)
  - CISC (Complex Instruction Set)
  - Basada en Pila (Stack based)

Generador de Código

- RISC (Reduced Instruction Set):
  - Muchos registros
  - Instrucciones de 3 direcciones
  - Modos de direccionamiento simples
  - Conjunto de instrucciones simple

- CISC (Complex Instruction Set):
  - Pocos registros
  - Instrucciones de 2 direcciones
  - Muchos modos de direccionamiento
  - Instrucciones de longitud variable

Generador de Código

- Basado en Pila (Stack based):
  - Las operaciones se realizan sobre operandos que están en el tope de la pila
  - Requiere muchas operaciones de intercambio y copia (swap, copy)
  - Ejemplo: JVM (Java Virtual Machine)

Selección de Instrucciones

- Si una instrucción de asignación de la forma x=y+z;

Se traduce a:

```
LD      R0, y       // R0 = y
ADD     R0, R0, z   // R0 = R0+z
ST      x, R0       // x = R0
```

- Como se traducen:
  - a = b + c; d = a + c;
  - a = a + 1;

  Que se puede optimizar?

- Cada instrucción de la máquina tiene un costo asociado

Localización y asignación de registros

- Localización: que variables se asignan a registros
- Asignación: establecer la relación variable-registro

La asignación óptima es un problema NP-Completo.

Orden de Evaluación

- El orden puede afectar la eficiencia del código
- En algún orden se puede requerir menos registros que en otro
- Encontrar un orden óptimo es un problema NP-Completo

Ejemplo:Lenguaje Destino

- Conjunto de Instrucciones:

Instrucción	Sentido	Ejemplo
LD dst, addr	dst = addr (dst es un registro)	LD r1, x LD r2,r1
ST dst, reg	dst = reg	ST x, r1

ADD dst, scr1, scr2
SUB dst, scr1, scr2
….
	dst = src_1 + src_2
	dst = src_1 - src_2
	(dst, src son registros o constantes)
		ADD r0, 100, r1

BR l	Branch (l es una etiqueta)	BR WH0
BLTZ reg, l	If reg < 0 Br l	BLTZ r0, WH1

Ejemplo:Lenguaje Destino

- Modos de direccionamiento:

Modo	Descripción	Sentido	Ejemplo
x	Variable de nombre x	Rvalue de x	valor
a(r)	Indexado, a es una variable, r un registro	contenido(Lvalue de a + contenido de r)	arreglo(r0)
entero(r)	Indexado	Contenido(entero + contenido de r)	100(r1)
*r	Indirecto	Contenido(contenido de r)	*r1
*entero(r)	Indirecto	Contenido(Contenido(entero + contenido de r))	*100(r0)
#entero	Constante entera	Entero	100

Ejemplo:Lenguaje Destino

- Costo de las Instrucciones:
  - El costo de cada instrucción es uno mas el costo de los modos de direccionamiento que usa
  - El uso de registros no genera costo (costo 0)
  - Cada acceso a memoria, o el uso de constantes, agrega 1 al costo.

Grafos de flujo

- Representación en grafos del código intermedio
- Permite
  - Mejor localización de registros
  - Mejor selección de instrucciones
- Se construye
  - Partir el código intermedio en **bloques básicos**
  - Los bloques básicos se convierten en nodos de un grafo de flujos, cuyos arcos indican que bloques pueden seguir a otros bloques

Bloques Básicos

- Secuencias de instrucciones que tienen las siguientes características:
  - El control de flujo ingresa al bloque solamente por su primera instrucción (no hay saltos a instrucciones en la mitad del bloque)
  - El control deja el bloque sin saltos, ni finalización del programa (halt), excepto por la ultima instrucción del bloque

Bloques Básicos

- Método de construcción:
  - Encontrar instrucciones líderes:
    - La primera instrucción del código intermedio
    - Cualquier instrucción que es destino de un salto condicional o incondicional
    - Cualquier instrucción que sigue inmediatamente a un salto condicional o incondicional
  - Para cada líder, su bloque básico se forma consigo misma y las instrucciones que le siguen, sin incluir otro líder ni el final del programa

Bloques Básicos

- Asignación de variables a Registros:
  - Siguiente uso de la variable: tener un indicador de en que instrucción vuelve a ser usada una variable.

```
(i) x = a * 10;
 ...
(j) p = x + z;
```

Flujo sin asignación a **x**

  - **(j)** usa un valor de **x** calculado en **(i)**, **x** esta viva en la instrucción **(i)**
  - Si la variable no es usada de nuevo dentro del bloque, se puede liberar el registro al cual se asignó.
  - Para ello se debe:
    - Determinar el tiempo de vida de la variable
    - Guardar en la tabla de símbolos el registro al que se asignó.

Bloques Básicos

- Asignación de variables a Registros: Método
  - La Tabla de Símbolos marca todas las variables como si estuvieran vivas al salir del bloque
  - Recorriendo el bloque desde la última instrucción hasta la primera, para toda instrucción de asignación, ej.

```
(i) x = y op z;
```

    - Relacione **(i)** con la información encontrada en la tabla de símbolos
    - Actualice en la tabla de símbolos **x** como “no viva” (no hay siguiente uso)
    - Actualice en la tabla de símbolos **y**, **z** como “vivas”, y su siguiente uso como **(i)**

Grafos de Flujo

- Representa como el control del flujo puede fluir entre los bloques básicos

- Hay un arco del bloque B al C si y solo si es posible que la primera instrucción de C se ejecute inmediatamente después de la última instrucción de B.
  - Salto de la instrucción final de B a al instrucción inicial de C
  - C sigue a B en el código

  Se dice que B es predecesor de C, y C sucesor de B.

- Se agregan dos nodos “Entrada” y “Salida” (no tienen instrucciones de código asociadas)
  - Se agrega un arco desde la “Entrada” hasta la primera instrucción del código.
  - Se agrega un arco desde cada bloque que pueda tener la última instrucción que se ejecute hasta la “Salida”
  - Se agrega un ancho desde cada bloque que tenga un salto a un código que no es parte del programa hasta la “Salida”

Grafos de Flujo

- Representación.
  - Reemplazar los saltos a instrucciones por saltos a bloques (para facilitar la manipulación de instrucciones dentro de cada bloque)
  - Cualquier estructura típica usada para representación de grafos, donde el contenido del nodo (puede ser una lista de instrucciones) contiene las instrucciones del bloque.
- Ciclos: la mayor parte del tiempo de ejecución de un programa esta en los ciclos
  - Un conjunto de nodos forma un ciclo L si y solo si:
    - Hay un nodo E, “Entrada al ciclo”, que es el único que tiene como predecesor un nodo externo a L.
    - Hay un camino no vacío desde cualquier nodo de L a E.

Optimización Local

- Representación de los bloques básicos con DAG (Directed acyclic graph):
  - Hay un nodo en DAG para cada uno de los valores iniciales de las variables que son usadas como operandos en alguna instrucción del bloque
  - Hay un nodo N asociado con cada instrucción S dentro del bloque, los hijos de N son las instrucciones anteriores a S que tienen las ultimas definiciones de los operandos usados en S.
  - El nodo N se marca con el operador que se aplica en S, y tiene asociada la lista de variables para las cuales esta es la última definición dentro del bloque.
  - Los nodos de salida son aquellos cuyas variables están vivas al salir del bloque (sus valores pueden ser usados después en otro bloque – para el análisis global).

Optimización

- Obtener una mejora sustancial en el tiempo de ejecución del código

- Optimización Local
  - Optimizar las instrucciones dentro de cada bloque básico
    - Subexpresiones comunes, Código muerto, Identidades algebraicas, referenciación de arreglos.

- Optimización Global
  - Optimizar las instrucciones de acuerdo con el flujo entre los bloques básicos
    - Vida de las variables, expresiones disponibles, eliminación de redundancia (subexpresiones comunes, invariantes de ciclos, expresiones parcialmente redundantes).

Optimización Local

- Subexpresiones comunes:
  - Ejemplo:

    a = b + c;

    b = a – d;

    c = b + c;

    d = a – d;

    Y se puede optimizar a:

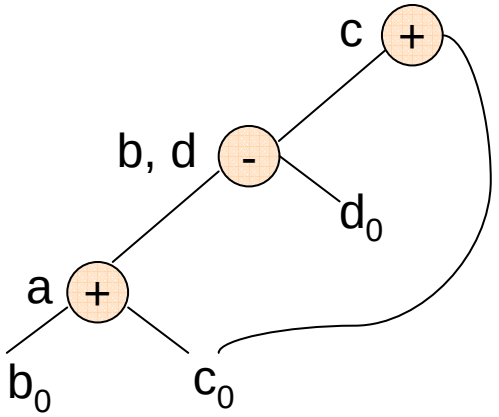
    a = b + c;

    b = a – d;

    c = b + c;

    d = b;

El DAG:



Optimización Local

- Reducción usando Identidades Algebraicas:
  - Ejemplo:

    a = b + c;

    b = b – d;

    c = c + d;

    d = b + c;

    Y se puede optimizar a:

    a = b + c;

    b = b – d;

    c = c + d;

    d = a;

El DAG: