Código intermedio y tabla de símbolos

Organización de lenguajes y compiladores 2

Aux. Oscar Estuardo de la Mora

Código intermedio

El microprocesador de una computadora no puede interpretar instrucciones muy complejas, para poder ser ejecutadas es necesario transformar este código complejo en instrucciones simples. El código intermedio es un código equivalente de bajo nivel de un código de alto nivel es la solución a este problema y es la forma en que los compiladores funcionan. El código intermedio en nuestro caso, esta escrito en código C/C++, se usan sentencias simples ya existentes en este lenguaje para generar dicho código intermedio.

Por ejemplo, una computadora no puede ejecutar este código:

*P = a + b + c / d -5*Z;*

Si no que lo transforma a código intermedio

t1 = a + b;
t2 = c/d;
t3 = t1 + t2;
*t4 = 5*Z;*
P = t3 – t4;

Pero... que son esas variables nuevas generadas??, Estas variables son llamadas variables temporales, son variables auxiliares que nos sirven para poder ejecutar código simple y funcionan guardando temporalmente instrucciones equivalentes, podemos ver en este caso que se respeto la precedencia de los operadores, en otras palabras, el orden en que las instrucciones deben ejecutarse, por ejemplo, las multiplicaciones y divisiones tienen mayor precedencia que las sumas y restas, entonces ejecutamos esas antes.

Podemos ver que las instrucciones son más simples, pero equivalentes en funcionamiento que la original.

Código intermedio y tabla de símbolos

Organización de lenguajes y compiladores 2

Aux. Oscar Estuardo de la Mora

Este código intermedio es llamado código Tres Direcciones (3Dir de ahora en adelante) y se llama así pues maneja tres direcciones de memoria en cada instrucción. En dicho código tenemos ciertas restricciones:

- No hay ciclos while, for, do-while (o repeat)
- No hay operaciones aritméticas complejas
- No hay comparaciones lógicas complejas
- No hay paso de parámetros por parte de las funciones y/o procedimientos

Instrucción IF (sí condicional)

La instrucción **if** es la mas simple de las instrucciones en 3Dir:

Su código equivalente es:

```
if( a > b){
        //codigo si es verdadero
}
//codigo si es falso
```

Su código equivalente es:

```
if(a > b) goto L1;
goto L2;
L1:
        //codigo si es verdadero
L2:
        //código si es falso
```

Tenemos la instrucción con la condición, y luego tenemos las instrucciones **goto** que tienen asociadas "**etiquetas**", esto son saltos que usan dichas etiquetas para poder ir a cierto lugar del código. Si la instrucción es verdadera, ejecutará el primer **goto** e irá hacia la etiqueta L1, si es falsa, se ejecutará el segundo **goto** con la etiqueta L2 y saltará al código si es falso.

Podemos notar también que el código falso de todos modos se va a ejecutar después de las instrucciones si la condición es verdadera, así mismo también el código 3Dir equivalente, si el programa encuentra una etiqueta, simplemente la ignorará y seguirá ejecutando el código siguiente.

Ahora, que pasa si encontramos instrucciones complejas como este:

```
if(a > b) and (b == 5) {
        //codigo si es verdadero
}
//codigo si es falso
```

Para poder convertir este código usaremos las reglas de la lógica matemática

A	B	A y B
V	V	V
V	F	F
F	V	F
F	F	F

A	B	A o B
V	V	F
V	F	F
F	V	F
F	F	F

Como en la instrucción tenemos un **AND** usamos las reglas del **AND**:

```
if(a > b) goto L1;
goto L2;
```

Ahora analicemos, si es verdadero, tendremos que analizar la siguiente condición, si no fuera verdadero, no necesitamos ir a verificar la siguiente instrucción, entonces lo siguiente que escribimos es la etiqueta que es verdadera.

```
if(a > b) goto L1;
goto L2;
L1:
if(b == 5) goto L3;
goto L4;
```

Luego escribiremos la siguiente condición, y que pasa con una instrucción con un **AND**, si todas las condiciones resultan verdaderas, se ejecutaran las instrucciones dentro del if

```
if(a > b) goto L1;
goto L2;
L1:
 if(b == 5) goto L3;
goto L4;
L3:
        //código si es verdadero
L2: L4:
        //código si es falso
```

Ahora escribimos la etiqueta L3 que nos lleva al código si es verdadero, pero... que paso con las etiquetas falsas?? Se colocarán al final del código, consecutivamente y ejecutarán el siguiente código, respetando las reglas de la lógica del **AND**.

Ahora una instrucción con el OR lógico

```
if(a > b) or (b == 5) {
        //código si es verdadero
}
//código si es falso
```

```
if(a > b) goto L1;
goto L2;
```

Ahora analicemos, si es falso con un **OR** lógico, debemos ver si la siguiente condición es verdadera, si fuese verdadera entonces se ejecutará el código verdadero, entonces, escribiremos la etiqueta que es falsa para saltar hacia la siguiente condición y analizarla

```
if(a > b) goto L1;
goto L2;
L2:  if(b == 5) goto L3;
goto L4;
L1:L3:
        //código si es verdadero
L4:
        //código si es falso
```

Escribimos el siguiente código con la condición y luego las etiquetas de los saltos, con la lógica del OR si al menos una condición (en este caso de las dos) es verdadera, saltaremos directamente al código verdadero, si fuera falso, analizaremos las siguientes condiciones para ver si son verdaderas y saltar al código verdadero, si al final todas fueron falsas, no pasaremos por el código verdadero.

Si tuvieramos combinaciones de **ANDs** y **ORs** debemos usar la precedencia de operadores, tal y como el sumas y multiplicaciones el AND tiene mas precedencia que el **OR**.

Ahora con el operador lógico **NOT**, simplemente se intercambian de posicion las etiquetas

```
if(NOT(a < b)){
        //código verdadero
}
//código falso
```

```
if( a < b) goto L1;
goto L2;
L2:
        //código verdadero
L1:
        //código falso
```

Para poder transformar estas instrucciones complejas como ciclos, usamos unas instrucciones llamados “saltos” y usan el código en C/C++ goto, asi mismo se usan etiquetas que especifican hacia donde van los saltos. Por ejemplo:

```
L2:
        printf(“Hola Mundo\n”);
goto L2;
```

Este código ejecutará indefinidamente la impresion de la frase *Hola Mundo*. El **goto** saltará hacia donde está la etiqueta **L2** y se ejecutará el código que precede a la etiqueta, como despues del **printf** viene acontinuacion un **goto,** lo ejecutará de nuevo y volvera a saltar hacia **L2**.
Con esta herramienta ahora podemos construir nuestros ciclos:

Código intermedio y tabla de símbolos

Organización de lenguajes y compiladores 2

Aux. Oscar Estuardo de la Mora

Ciclo While

```
int i = 0;
while(i < 10){
        //codigo
        i++;
}
```

Lo primero que tenemos es una inicialización, luego la condición, ejecuta el código y luego incrementamos.

```
i = 0;
L1:
if(i < 10) goto L2;
goto L3;
        //codigo
        i = i + 1;
goto L1:
L3:
```

Inicializamos la variable, a continuación vemos la estructura **if**. Este contendrá la condicion del **while** y tendra dos **goto**, uno que nos lleva a la instrucción si es verdadera y otro que nos lleva a otra instrucción si es falsa.

Si es verdadera, nos llevara a ejecutar //código, luego el incremento de la variable y luego como un ciclo **while**, regresaremos a verificar la condición. Si es verdadera, entraremos de nuevo al cuerpo del ciclo, pero si es falsa, saltaremos hacia afuera del cuerpo del cidó y este conduirá.

Ciclo Do-While (Repeat)

```
int i = 0;
do{
        //código
        i = i + 1;
}while(i < 10);
//demás código
```

El ciclo funciona de tal manera de que se ejecute al menos una vez el código dentro del ciclo

```
i = 0;
L1:
        //codigo
        i = i + 1;
if(i < 10) goto L1;
goto L2;
L2:
        //demas código
```

Inicializamos la variable, luego encontramos un **L1** pero la ignoramos pues solo indica el lugar a donde llega un salto, luego ejecutamos el código dentro del ciclo y luego el incremento, por último encontramos el **if** equivalente en el **while** donde salta hacia las instrucciones falsas o verdaderas dependiendo del resultado de la condición.

Todo en esta instrucción esta bien, este código 3Dir es incorrecto, si lo generamos con un compilador este código no puede ser generado. **"Todo código que cumpla con las reglas de tres direcciones es código tres direcciones, pero no todo código tres direcciones es código correcto".**

El código correcto es el siguiente:

```
L1:
        //codigo
        i = i + 1;
if(i < 10) goto L2;
goto L3;
L2:
        goto L1;
L3:
        //demas código
```

Cuando implementemos nuestras gramáticas veremos que muchos pedazos pueden reutilizarse y veremos que las condiciones son las mismas donde seas que se usen, ya sea en ciclos, o condiciones, los saltos se generarán junto a las condiciones, por eso es que no podemos usar el código anterior para poder generar el ciclo do-**while**, pues estamos poniendo una etiqueta generada anteriormente al **if** de la condición.

Ciclo For

```
for(int i = 0; i<10; i++){
        //código
}
//demás código
```

Código equivalente

```
i = 0;
L1:
if(i < 10) goto L2;
goto L3;
L2:
        //código
        i = i + 1;
goto L1;
L3:
//demás código
```

Instrucción Switch-Case

La instruccion switch-case es basicamente un if resumido y mas amigable, tenemos el siguiente código:

```
int a = 1;
switch(a){
        case 1:{ println("Es numero 1"); }break;
        case 2:{ println("Es numero 2"); }break;
        case 3:{ println("Es numero 3"); }break;
        default:{ println("Es el default"); }
}
//demas código
```

Código intermedio:

```
a = 1;
if(a == 1) goto L2;
goto L3;
L2:
        print("Es numero 1");
goto L1;
L3:
if(a == 2) goto L4;
goto L5;
L4:
        print("Es numero 2");
goto L1;
L5:
if(a == 3) goto L6;
goto L7;
L6:
        print("Es numero 3");
goto L1;
L7:
        print("Es el default");
L1:
//demas codigo
```

Podemos notar que son muchas instrucciones if con la lógica que tiene un switch-case.

Los métodos (funciones y procedimientos) se verán mas adelante.

Código intermedio y tabla de símbolos

Organización de lenguajes y compiladores 2

Aux. Oscar Estuardo de la Mora

Tabla de símbolos

La tabla de símbolos es una estructura de metadatos para almacenar información de la compilación, para generar código intermedio a partir de un código de alto nivel que se esta analizando. Generalmente antes de generar código intermedio se construye primero esta estructura (junto al análisis léxico y sintáctico), pues contiene información que usaremos mas adelante.

Una tabla de símbolos básica tiene los atributos:

- **Nombre:** Identificador del atributo
- **Tipo:** Tipo del atributo
- **Ámbito:** Lugar donde se encuentra el atributo
- **Rol:** Función del atributo
- **Apuntador:** Dirección de memoria del atributo

Pero antes, que es un metadato?? Un metadato es un dato compuesto por datos, un ejemplo fácil son los metaobjetos, son objetos compuestos por objetos:

```
class Ruedas{
        int cantidad;
}
class Motor{
        int caballosFuerza;
}

class Vehiculo{
        Ruedas rueda;
        Motor motorcito;
        int modelo;
}
```

Vehículo es un metaobjeto, pues esta compuesto por otros objetos, así también de datos simples.

Un dato que contenga datos simples entonces es un metadato. Es cuestión de la implementación que se le de.

Código intermedio y tabla de símbolos

Organización de lenguajes y compiladores 2

Aux. Oscar Estuardo de la Mora

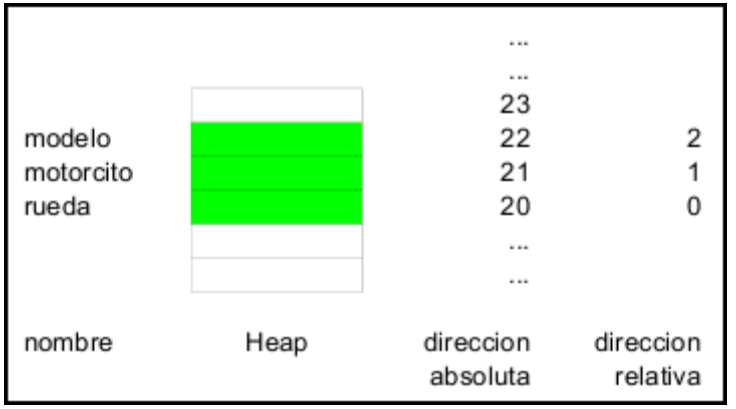
Ahora especificando un poco más los atributos de la tabla de símbolo si tomamos de ejemplo el código anterior:

Nombre	Tipo	Ámbito	Rol	Apuntador
Ruedas			Clase	
cantidad	int	Ruedas	Variable de clase	0
Motor			Clase	
caballosFuerza	int	Motor	Variable de clase	0
Vehiculo			Clase	
reuda	Ruedas	Vehiculo	Variable de clase	0
motorcito	Motor	Vehiculo	Variable de clase	1
modelo	int	Vehiculo	Variable de clase	2

Ahora con la tabla de símbolos llena podemos ver como es que funcionan los atributos. Por ejemplo **'cantidad'** esta dentro de la clase **Ruedas**, esta en su ámbito, **'caballosFuerza'** esta en la clase Motor, esta en otro ámbito, luego en apuntador, podemos ver que esta en valor cero ('0'), pero que significa ??

Cuando creamos los objetos, estos se almacenan en el *Heap* de memoria, y los atributos del objeto son colocados en cierta parte del Heap. Y tienen un orden en específico para poder localizar los atributos. Tomaremos un objeto **Vehiculo**

Si tomamos un segmento en el *Heap* de memoria tenemos que el objeto esta alojado del segmento 20 al 22 de la memoria. Pero como sabemos a que segmento de memoria ir para obtener los datos que necesitamos??
Pues en algún lugar del programa tendremos la dirección de memoria del objeto, para obtener el dato que necesitamos por ejemplo, el atributo modelo, solo debemos desplazarnos 2 celdas de memoria para alcanzar el atributo. Si queremos alcanzar el objeto motorcito, debemos desplazarnos una celda de memoria para poder obtener su valor.



El atributo puntero nos va a servir para poder hacer esos desplazamientos, primero obtener la dirección absoluta de memoria de donde esta el objeto y luego desplazamos para poder alcanzar los atributos que necesitamos obtener o modificar.

Código intermedio y tabla de símbolos

Organización de lenguajes y compiladores 2

Aux. Oscar Estuardo de la Mora

Veamos otro ejemplo:

```
class Ejemplo{
        public static int suma(int a, int b){
                int c = a + b;
                return c;
        }
}
```

Algo que debemos tomar en cuenta es que los parámetros de los métodos (funciones y procedimientos) son considerados variables locales. En el caso de métodos que regresan valores (funciones) el **return** es considerado una variable local del tipo del método. Nuestra tabla de símbolos nos quedaría de la siguiente forma:

Nombre	Tipo	Ámbito	Rol	Apuntador
Ejemplo			Clase	
suma	Int	Ejemplo	Método	
a	Int	suma	Variable local	0
b	Int	suma	Variable local	1
c	Int	suma	Variable local	2
return	Int	suma	Variable local	3

Otra cosa que debemos tomar en cuenta es que el **return** SIEMPRE va a ser el último atributo que va a tener un método que regresa valores (funciones), por eso lo ponemos al final de todas las variables locales.

Como podemos notar, la tabla de símbolos puede extenderse en atributos, este método tiene los modificadores ***static*** y ***public*** y pueden ser muy útil almacenar estos modificadores en nuestra tabla para posteriores usos semánticos al generar código intermedio.

Tal y como objetos en el *Heap*, también los atributos y referencias están ordenados en el *Stack* en este caso, podemos ver el ámbito del método **suma()** y que los atributos están dentro de este mismo. El método comienza en la dirección absoluta 11 del *Stack* pero cada atributo del método tiene su posición relativa correspondiente para poder saber a cual se están refiriendo.