# Generación de Código Intermedio

**Francisco Rodríguez Zamora**
Universidad de Costa Rica, Escuela de Ciencias de Computación e Informática
San José, Costa Rica
francisco.rodriguez@ecci.ucr.ac.cr

**Jonathan Calderón Varela**
Universidad de Costa Rica, Escuela de Ciencias de Computación e Informática
San José, Costa Rica
jonathan.calderon@ecci.ucr.ac.cr

**Abstract**

An intermediate representation is a data structure that represents the program source during the process of translation into object code. At first, the compiler translates source code into a more suitable form for optimization, and then turns it into machine code. The compiler transforms the code into a more manageable language, usually three-address code, which represents exactly one instruction code machine, does type checking and translates expressions.

**Keywords: compiler, intermediate code, three-address code, type checking, backpatching.**

**Resumen**

Una representación intermedia es una estructura de datos que representa al programa fuente durante el proceso de la traducción a código objeto. El compilador primero traduce el código fuente en una forma mas situable para la optimización, para luego convertirla en el código de maquina. El compilador transforma el código en un lenguaje mas manejable, usualmente código de tres direcciones, el cual representa exactamente una instrucción de código de maquina, verifica tipos y traduce expresiones.

**Palabras clave: compilador, código intermedio, código de tres direcciones, verificación de tipos, backpatching.**

## Introducción

El proceso de la compilación se desglosa en dos partes: la parte que depende solo del lenguaje fuente (etapa inicial o front-end) y la parte que depende solo del lenguaje objeto (etapa final o back-end).

La etapa inicial traduce un programa fuente a una representación intermedia s partir de la cual la etapa final genera el código objeto. De esta forma, los detalles que tienen que ver con las características del lenguaje objeto (código ensamblador, código maquina absoluto o relocalizable,...), la arquitectura de la maquina (numero de registros, modos de direccionamiento, tamaño de los tipos de datos, memoria cache,...), el entorno de ejecución (estructura de registros y memoria de la máquina donde se va a ejecutar el programa...) y el sistema operativo se engloban en la etapa final y se aíslan del resto.

El código intermedio es un código abstracto independiente de la máquina para la que se generará el código objeto. El código intermedio ha de cumplir dos requisitos importantes: ser fácil de producir a partir del análisis sintáctico, y ser fácil de traducir al lenguaje objeto.

Con una representación intermedia bien definida, un compilador para el lenguaje i y la maquina j puede ser construido combinando el front-end para el lenguaje i con el back-end de la maquina j. De esta manera se pueden construir m*n compiladores escribiendo únicamente m front-ends y n back-ends.

## 1 Variantes a los árboles de Sintaxis

Tomando como base los árboles de análisis sintáctico, que representan las instrucciones del programa, se puede generalizar un nuevo tipo de grafo, llamado un grafo acíclico diseccionado para las subexpresiones de una expresión [1].

En este grafo, una subexpresion N tendría más de un padre, si esta apareciera varias veces en la misma expresión (figura 1). En un árbol de sintaxis regular, la expresión parecería tantas veces como aparezca en la expresión original. Esta característica de los grafos de subexpresiones hace que el compilador genere código más eficiente al evaluar una expresión.

Una forma de implementar un grafo aciclico de subexpresiones es mediante un vector. En cada entrada del vector se guarda un código de operación y, dependiendo de la operación, un valor, o referencias a las entradas del vector que pueda utilizar ese vector (figura 2).

El principal problema de esta forma de implementación, es que cada vez que se quiera encontrar un nodo de la expresión, hay que recorrer todo el vector. Una alternativa de implementación es con una tabla hash, en la cual en cada espacio se guarden los nodos, proveyendo acceso casi inmediato a los nodos.

## 2 Código de Tres Direcciones

Es una manera linear de representar un grafo de subexpresiones en donde los nombres explícitos corresponden a los nodos interiores del grafo (figura 3), muy útil principalmente a la hora de optimizar. Las instrucciones se representan utilizando cuando mucho un solo operador en la parte derecha de la expresión, es decir, una expresión del tipo "x+y*z" se representaría de la manera "t1=y*z y t2=x+t1".

El código de tres direcciones se basa en 2 conceptos principalmente en 2 conceptos: direcciones e instrucciones.

- Nombres: los nombres que se usan en el programa puede aparecer también en el código de 3 direcciones, aunque a la hora de implementar se usara un puntero o su entrada en la tabla de símbolos.
- Una constante: el compilador tiene que poder interactuar con muchos tipos de constantes.

Las instrucciones más básicas pueden definirse de la siguiente manera [2]:

- x=y'op'z, donde 'op' es un operador binario aritmético o una expresión lógica.
- x='op'y, donde 'op' es una operación unaria.
- x=y, donde a x se le asigna el valor de y
- goto L, la instrucción con la etiqueta L se ejecutara.
- if x goto L o if false goto L, se ejecutaría la instrucción con la etiqueta L dependiendo de si x es verdadero o falso.
- if x relop y goto L, se ejecutaría la instrucción L dependiendo del operador relacional que se le aplique a x y y
- param x y call p,n, donde param x indica que x es un parámetro. call llama un procedimiento p con un número de parámetros n.

En el diseño del código intermedio, es necesario escoger un buen conjunto de operadores. Se Usando el código de 3 direcciones, el compilador puede representar código intermedio y ayudarse en la implementación mas optima del código final.

Las direcciones se pueden definir como [2]:deben tener los suficientes operadores para satisfacer las instrucciones del lenguaje, y que a su vez sean cercanos al lenguaje de maquina [3]. Sin embargo, si se crean demasiadas instrucciones, el optimizados y el generador de código tendrán que trabajar mas para generar el código final.

Una proposición de código de 3-direcciones se puede implantar como una estructura tipo registro con campos para el operador, los operandos y el resultado. La representación final será entonces una lista enlazada o un vector de proposiciones. Hay dos formas principales de implementar el código de tres direcciones:

21. **Cuadruplas**

Una cuadrupla es una estructura tipo registro con cuatro campos que se llaman (op, result, arg1, arg2). El campo op contiene un código interno para el operador.

Por ejemplo, la proposición de tres direcciones x = y + z se podría representar mediante la cuadrupla (ADD, x,y, z). Las proposiciones con operadores unarios no usan el arg2. Los campos que no se usan se dejan vacíos o un valor NULL. Como se necesitan cuatro campos se le llama representación mediante cuadruplas[2].

**32. Tripletas**

Para evitar tener que introducir nombres temporales en la tabla de símbolos, se hace referencia a un valor temporal según la posición de la proposición que lo calcula. Las propias instrucciones representan el valor del nombre temporal. La implementación se hace mediante registros de solo tres campos (op, arg1, arg2) [1].

En la notación de tripletes se necesita menor espacio y el compilador no necesita generar los nombres temporales. Sin embargo, en esta notación, trasladar una proposición que defina un valor temporal exige que se modifiquen todas las referencias a esa proposición. Lo cual supone un inconveniente a la hora de optimizar el código, pues a menudo es necesario cambiar proposiciones de lugar [1].

Una forma de solucionar esto consiste en listar las posiciones a las tripletas en lugar de listar las tripletas mismas. De esta manera, un optimizador podría mover una instrucción reordenando la lista, sin tener que mover las tripletas en si.

# 3 Tipos y Declaraciones

Las expresiones de tipo son las estructuras de tipos que definen un programa, pueden ser de tipos básicos (booleano, char, integer, float, void) o formados por una estructura de tipos básicos creada por operadores de constructores de tipo [3].

Una forma conveniente de representar una expresión de tipo es usando un grafo. Los nodos interiores se usaran para constructores de tipo y las hojas se usaran para los tipos básico. Cuando las expresiones de tipos están representadas por grafos se pueden comparar para ver si son de tipos equivalentes [1]. Dos tipos son estructuralmente equivalentes si:

- son el mismo tipo básico
- son formados utilizando el mismo constructor con los mismos tipos básicos
- uno es un tipo que denota otro

Dado un tipo de una variable, se puede calcular la cantidad de almacenamiento que se necesitara para la variable en tiempo de ejecución. El tipo y la dirección relativa se guardaran en la tabla de símbolos para cada variable. Para los tipos de dimensión variable, tales como hileras de caracteres o vectores dinámicos, se les asignara una cantidad de almacenamiento para un puntero al tipo.

Parte importante de la generación de código de un compilador es la verificación de tipos. Se debe verificar que el programa siga las reglas de tipos especificados por el programa, una implementación que se base en estas reglas encontrará errores en las modificaciones al compilar.

# 4 Verificación de tipos

La verificación de tipos es una técnica utilizada para asegurar que un programa obedece las reglas de compatibilidad específicas del lenguaje de programación utilizado. La verificación se puede realizar de dos formas: en tiempo de ejecución o de manera estática.

Para que un compilador pueda realizar el chequeo se debe asignar una expresión de tipo que identifique cada componente del lenguaje fuente. Con estas definiciones de tipos se pueden formular reglas lógicas que son llamadas: sistema de tipos, el cuál define como un lenguaje de programación clasifica sus elementos en tipos y determina reglas de relación y comportamiento entre estos tipos. Una violación a alguna regla es llamada choque de tipos.

## 5.1 Tipos de chequeo

La verificación de datos se puede realizar de dos maneras: por medio de síntesis y con el método de la inferencia.

El método de la síntesis consiste en derivar el tipo de una expresión basándose en los tipos de sus subexpresiones. En la síntesis se requiere que los tipos estén declarados antes de ser utilizados y deben poseer un nombre que los identifique.

La inferencia de tipos se basa en determinar el tipo de una expresión o elemento según la manera en que sea utilizado, es decir si tenemos una función f(x) donde se supone que x sea de un tipo Y, tendremos que cada vez que se aplique F () sobre algún elemento este elemento será de tipo Y.

### 5.2 Conversión de tipos

Algunos tipos diferentes de datos son tratados de forma diferente por el computador, sin embargo es necesario poder operar entre ellos, por lo que los lenguajes de programación implementan la conversión de tipos, que se basa en cambiar el tipo de una expresión por otro.

Por ejemplo si deseamos multiplicar un numero entero y uno flotante, el compilador debe unificar los tipos ya que el almacenamiento y manejo de la maquina es diferente para estos 2 tipos. Lo que el compilador hacer es convertir el entero a flotante (figura 4)

### 5.3 Sobrecarga de funciones y operadores

La sobrecarga de operadores es utilizada cuando necesitamos realizar varias funciones con el mismo símbolo o función. El operador se trata dependiendo del contexto donde se encuentre.

Por ejemplo el operador "+" en Java puede ser utilizado para concatenar una hilera o para sumar numerales. "Las funciones definidas por el usuario se pueden sobrecargar como se muestra a continuación:

```
Void err ()
void err(String s)
```

{…}
*{…}*

La escogencia de una de estas dos funciones se realiza tomando en cuenta su número y tipo de argumentos." [1].

### 5.4 Funciones polimórficas

Las funciones normales permiten ser ejecutadas con argumentos de tipos fijos, una función polimórfica acepta parámetros de tipos variables. El término polimórfico aplica también a operadores.

Las funciones polimórficas son muy útiles ya que permite crear algoritmos independientes del tipo de operadores.

## 5 Control de flujo

En los lenguajes de programación hay estructuras y operadores que permiten controlar el flujo de la ejecución, estos pueden ser ciclos, saltos, condiciones entre otros.

### 51. Expresiones booleanas

Las expresiones booleanas son utilizadas para determinar si una o más condiciones son verdaderas o falsas, y el resultado de su evaluación es un valor de verdad. Las expresiones pueden ser relacionales, es decir que comparan dos valores y determinan si existe o no una cierta relación entre ellos son de la forma *E*1 oprer *E*2, donde *E*1 y *E*2 son expresiones aritméticas y oprer es cualquier operador relacional <, >, <=, >=, etc. o pueden ser funciones booleanas que retornan el valor de verdad, estas tienen la forma p (x). [2]

### 52. Saltos

En el código de los saltos los operadores lógicos &&, || y ! son traducidos a saltos aunque estos no aparecen realmente en el código. Por ejemplo la expresión: if (x < 100 || x > 200 && x!= y ) x=0; se puede traducir como las siguientes instrucciones:

```
If x < 100 goto L2
If False x > 200 goto L1
If False x != y goto L1
L2: x =0
```

L1:

Si la expresión es verdadera se alcanza la etiqueta L2 y se realiza la asignación en caso contrario no haría nada. [1]

## 6 *Backpatching*

Cuando se genera un código con saltos condicionales y expresiones booleanas se enfrenta un problema muy común, en la expresión if (B) S debemos establecer el salto hacia S cuando se da B, el problema es que no siempre se conoce B a la hora de hacer esta transformación.

*Backpatching* es un enfoque donde al momento de generar un salto el objetivo de este salto se deja temporalmente indefinido. Cada salto es almacenado en una lista de saltos donde las etiquetas de sus destinos son establecidas en el momento en que ya pueden ser determinadas.

La manipulación de la lista de etiquetas se realiza con los métodos:

- *makelist*(i) crea una nueva lista que contiene solo a **i**, un índice en el vector de instrucciones y además el método *makelist* retorna un puntero a la lista creada.
- *merge(pl , p2 )* concatena las listas apuntadas por *pl* y *p2* , y retorna el puntero a la nueva lista.
- *backpatch(p, i)* inseta ***i*** como la etiqueta objetivo para cada instruccion en la lista apuntada por **p.**

## 7 Instrucciones Switch

La declaración *switch-case* es utilizada en muchos lenguajes de programación. Esta instrucción es un bloque de selección donde se evalúa una condición y en base a ella se elije una acción determinada.

La estructura tiene la siguiente sintaxis:

```
Switch (E) {
    Case V1: acción 1
    Case V2: acción 2
    Case Vn-1: acción n-1
    Default: acción n
}
```

### 71. Traducción de instrucciones *Switch-case*

La traducción entendida de la estructura es la siguiente:

- Evaluar la expresión E.
- Buscar el valor V que corresponde a la condición E en la lista de casos. Si ningún valor de V es adecuado se ejecuta la etiqueta *default,*
- Se ejecuta la acción dentro del caso seleccionado.

El paso 2 se puede realizar de muchas maneras, la más intuitiva que sería una serie de saltos que evalúen cada una de las etiquetas, u otras mas elaboradas. [1]

### 72. Traducción de Switch-case dirigida por sintaxis

La figura 5 es una representación de una posible traducción de una estructura switch la cual genera un código eficiente para reconocer el valor de E en un conjunto de valores T.

Dependiendo de la traducción que se produzca es posible que el compilador deba hacer una comparación exhaustiva y varios saltos hasta encontrar el valor adecuado, que en el peor de los casos será el último, o simplemente se puede crear una etiqueta donde se haga el manejo lógico como en la figura 2 lo hace la etiqueta *Test.[1]*

## *8* Figuras y tablas

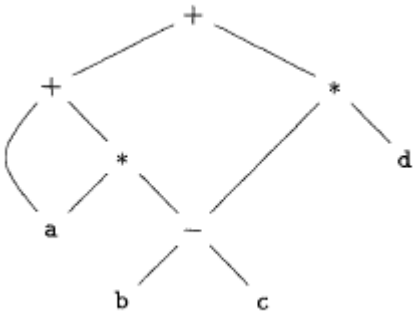


Figure 1. Grafo aciclico diseccionado de subexpresiones

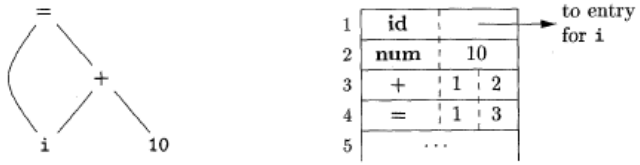


Figura 2. Implementación de un grafo aciclico mediante vectores

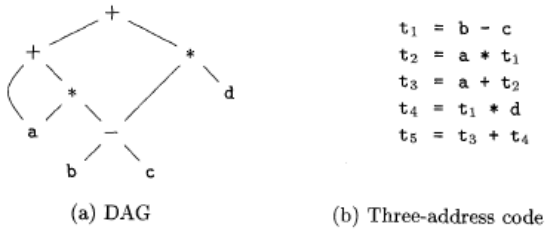


Figura 3. Correspondencia entre un grafo aciclicico de subexpresiones y codigo de tres direcciones

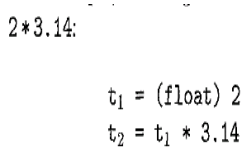


Figura 4. Conversión de un Int a Float

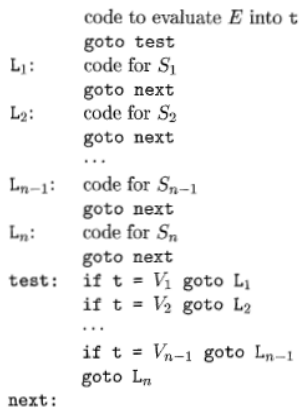


Figura 5 Traducción de una estructura switch-case

## References


[1] Aho A, Lam M, Sethi R, Ullman J. Compilers*: Pinciples, technicles and tools. Second Edition.* Pearson. 2007. Capitulo 6

2] Berendí M. *Procesadores de Lenguajes.* 2007 Capitulo 6

3] Moreno A. *The design and implementation of intermediate code for software visualization.* University of Joensuu. 2005.

4] Bornat R. *Understanding and writing compilers.* Middlesex University. 1979 capitulo 2