GENERACIÓN DE CÓDIGO INTERMEDIO

INTRODUCCION

Esta fase del compilador no es en realidad una parte separada del compilador, la mayoría de los compiladores generan código como parte del proceso de análisis sintáctico, esto es debido a que requieren del árbol de sintaxis y si este no va a ser construido físicamente, entonces deberá acompañar al analizador sintáctico al barrer el árbol implícito. En lugar de generar código ensamblador directamente, los compiladores generan un código intermedio que es más parecido al código ensamblador, las operaciones por ejemplo nunca se hacen con más de dos operandos. Al no generarse código ensamblador el cual es dependiente de la computadora especifica, sino código intermedio, se puede reutilizar la parte del compilador que genera código intermedio en otro compilador para una computadora con diferente procesador cambiando solamente el generador de código ensamblador al cual llamamos back-end, la desventaja obviamente es la lentitud que esto conlleva.

La tarea de síntesis suele comenzar generando un código intermedio. El código intermedio no es el lenguaje de programación de ninguna máquina real, sino que corresponde a una máquina abstracta, que se debe de definir lo más general posible, de forma que sea posible traducir este código intermedio a cualquier máquina real.

El objetivo del código intermedio es reducir el número de programas necesarios para construir traductores, y permitir más fácilmente la transportabilidad de unas máquinas a otras. Supóngase que se tienen n lenguajes, y se desea construir traductores entre ellos. Sería necesario construir n*(n-1) traductores.

Sin embargo si se construye un lenguaje intermedio, tan sólo son necesarios 2*n traductores.

Así por ejemplo un fabricante de compiladores puede construir un compilador para diferentes máquinas objeto con tan sólo cambiar las dos últimas fases de la tarea de síntesis.

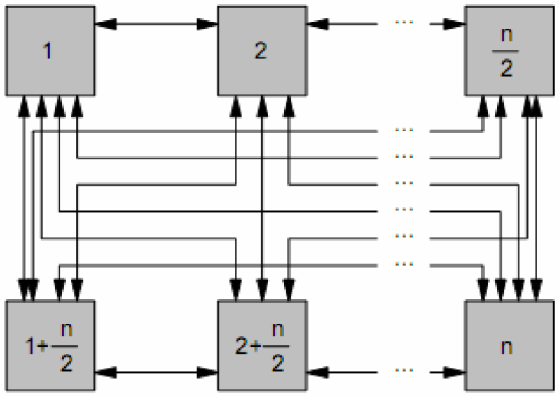


Figura 1. n*(n-1) traductores entre n lenguajes

Las máquinas abstractas deben definirse completamente: por una parte se definirá su arquitectura y por otra su repertorio de instrucciones. La arquitectura de la máquina abstracta se elegirá de forma que contribuya a facilitar la portabilidad dentro del grupo de arquitecturas hacia las que previsiblemente se dirigirá el código objeto. Habitualmente las arquitecturas típicas de máquinas abstractas son: Máquinas basadas en pila, basadas en registros, combinación de pilas y registros, orientadas a objetos. También se pueden clasificar desde el punto de vista de la cantidad y complejidad de sus instrucciones en máquinas CISC (Complex Instruction Set Computer)y RISC (Reduced Instruction Set Computer).

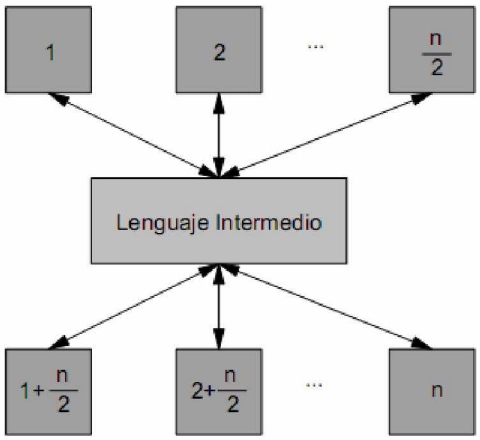


Figura 2. 2*n traductores entre n lenguajes

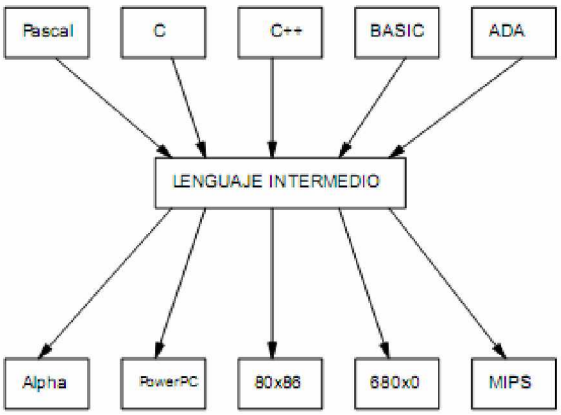


Figura 3. Ejemplos de compiladores de distintos lenguajes para distintas máquinas

La forma de las instrucciones del código intermedio puede variar según sea el diseño de la máquina abstracta. Por ejemplo la expresión:

$$(A+B)*(C+D)$$

Puede traducirse al siguiente conjunto de cuartetos:

(+, A, B, T1)
(+, C, D, T2)
(*, T1, T2, T3)

Donde (+ , A , B , T1) se interpreta como suma A y B y coloca el resultado temporalmente en T1. El resto de los cuartetos se interpretan de forma similar. T1, T2, y T3 pueden ser registros de la máquina o posiciones de memoria temporales. Otra forma de notación es la llamada notación polaca inversa (Reverse Polish Notation o RPN). Así la expresión anterior(+, A, B, T1) en notación polaca inversa es:

AB + CD + *

De esta forma los operadores aritméticos se colocan en el orden en que serán ejecutados, y además no es necesario el uso de paréntesis. Habitualmente la notación polaca va ligada a máquinas que utilizan una pila, y su representación interna es de la forma:

```
1  PUSH A
2  PUSH B
3  ADD
4  PUSH C
5  PUSH D
6  ADD
7  MUL
```

El código-P (abreviatura de código empaquetado, packed) utiliza operadores postfijos para el manejo de máquinas abstractas de pila. Los compiladores comerciales de Microsoft utilizan este tipo de código intermedio, su descripción puede consultarse en el manual Programming Techniques, capítulo 3 del compilador de C/C++ Versión 7.0

Otros traductores utilizan como código intermedio un lenguaje de medio nivel como puede ser el lenguaje C, que se caracteriza por su transportabilidad, por ejemplo el lenguaje Eiffel, También se está utilizando por su eficiencia el lenguaje C++ para el desarrollo de lenguajes y herramientas orientadas a objetos

La idea de un lenguaje intermedio universal no es nueva, y ya fue propuesta por T.B. Steel en 1961 con su célebre UNCOL (Universal Comunicación Oriented Language). El proyecto siempre fue bien recibido, pero la gran variedad de arquitecturas de ordenadores y otros intereses comerciales hicieron que el proyecto fracasase. Sin embargo la idea todavía permanece en la mente de muchas personas, y se puede decir que en muchas áreas de la Informática los lenguajes C y C++ se han convertido en una especie de lenguaje intermedio universal. Actualmente la idea del lenguaje intermedio universal parece renacer debido a la necesidad de ejecutar aplicaciones independientes de plataforma en la red Internet. Así el código binario bytecode (ficheros con la extensión .class) de la máquina abstracta JVM (Java Virtual Machine) se está convirtiendo en el código intermedio universal, ya existen intérpretes de JVM para todos los sistemas operativos.

TECNICAS BASICAS DE GENERACION DE CODIGO INTERMEDIO

En esta sección comentaremos los enfoques básicos para la generación de código en general, mientras que en secciones posteriores abordaremos la generación de código para construcciones de lenguaje individuales por separado.

Código intermedio o código objetivo como un atributo sintetizado

La generación de código intermedio (o generación de código objetivo directa sin código intermedio) se puede ver como un cálculo de atributo similar a muchos de los problemas de atributo estudiados en el capítulo 6. En realidad, si el código generado se ve como un atributo de cadena (con instrucciones separadas por caracteres de retorno de línea), entonces este código se convierte en un atributo sintetizado que se puede definir utilizando una gramática con atributos, y generado directamente durante el análisis sintáctico o mediante un recorrido postorden del árbol sintáctico.

Para ver cómo el código de tres direcciones, o bien, el código P se pueden definir como un atributo sintetizado, considere la siguiente gramática que representa un pequeño subconjunto de expresiones en C:

Exp à id = exp | aexp
aexp à aexp +factor | factor
factor à (exp) | num | id

Esta gramática sólo contiene dos operaciones, la asignación (con el símbolo =) y la adición (con el símbolo +). El token id representa un identificador simple, y el token num simboliza una secuencia simple de dígitos que representa un entero. Se supone que ambos tokens tienen un atributo strval previamente calculado, que es el valor de la cadena, o lexema, del token (por ejemplo, "42" para un num o "xtemp" para un id).

Código P

Consideraremos el caso de la generación del código P en primer lugar, ya que la gramática con atributos es más simple debido a que no es necesario generar nombres para elementos temporales. Sin embargo, la existencia de asignaciones incrustadas es un factor que implica complicaciones. En esta situación deseamos mantener el valor almacenado como el valor resultante de una expresión de asignación, ya que la instrucción de código P estándar esto es destructiva, pues el valor asignado se pierde. (Con esto el código P muestra sus orígenes de Pascal, en el cual no existen las asignaciones incrustadas.) Resolvemos es- te problema introduciendo una instrucción de almacenamiento no destructiva están en nuestro código P, la que como la instrucción ato, supone que se encuentra un valor en la parte superior o tope de la pila y una dirección debajo de la misma; está almacena el valor en la dirección pero deja el valor en el tope de la pila, mientras que descarta la dirección. Con esta nueva instrucción, una gramática con atributos para un atributo de cadena de código P se proporciona en la tabla 1. En esa figura utilizamos el nombre de atributo pcode la cadena de código P. También empleamos dos notaciones, diferentes para la concatenación de cadena: + + cuando las instrucciones van a ser concatenadas con retornos de línea insertados entre ellas y // cuando se está construyendo una instrucción simple y se va a insertar un espacio.

Dejamos al lector describir el cálculo del atributo pcode en ejemplos individua mostrar que, por ejemplo, la expresión (x=x+3) +4 tiene ese atributo

lda x
lod x
ldc 3
adi
stn
ldc 4
adi

Tabla1. Gramática con atributos de código P tomo un atributo de cadena sintetizado

Regla gramatical	Reglas semánticas
$exp_1 \rightarrow$ `id` = exp_2	exp_1.pcode = "`lda`" ‖ `id`.strval ++ exp_2.pcode ++ "`stn`"
$exp \rightarrow aexp$	exp.pcode = aexp.pcode
$aexp_1 \rightarrow aexp_2$ + factor	$aexp_1$.pcode = $aexp_2$.pcode ++ factor.pcode ++ "`adi`"
$aexp \rightarrow factor$	aexp.pcode = factor.pcode
$factor \rightarrow$ (exp)	factor.pcode = exp.pcode
$factor \rightarrow$ `num`	factor.pcode = "`ldc`" ‖ `num`.strval
$factor \rightarrow$ `id`	factor.pcode = "`lod`" ‖ `id`.strval

Código de tres direcciones

Una gramática con atributos para código de tres direcciones de 1 gramática de expresión simple anterior se proporciona en la tabla 2. En esa tabla utilizamos el atributo de código que se llama tacode (para código de tres direcciones), y como en la tabla 1, + + para concatenación de cadena con un retorno de Línea y // para concatenación de cadena con un espacio. A diferencia del código P, el código de tres direcciones requiere que se generen nombres temporales para resultados intermedios en las expresiones, y esto requiere que la gramática con atributos incluya un nueva atributo name para cada nodo.

Este atributo también es sintetizado, pero para asignar nombres temporales recién genera a nodos interiores utilizamos una función newremp( ) que se supone genera una secuencia de nombres temporales $t_1, t_2, t_3, \ldots$. (Se devuelve uno nuevo cada vez que se llama a newtemp( )). En este ejemplo simple sólo los nodos correspondientes al operador + necesitan nuevos nombres temporales; la operación de asignación simplemente utiliza el nombre de la expresión en el lado derecho.

Advierta en la tabla 2, que en el caso de las producciones unitarias exp à aexp y aexp -à factor, el atributo name, además del atributo tacode, se transportan de hijos a padres y que, en el caso de los nodos interiores del operador, se generan nuevos atributos name antes del tacode asociado. Observe también que, en las producciones de hoja factor à num y factor à id, el valor de cadena del token se utiliza como factor.name, y que (a diferencia del código P) no se genera ningún código de tres direcciones en tales nodos (utilizamos para representar la cadena vacía).

De nueva cuenta, dejamos al lector mostrar que, dadas las ecuaciones de atributo de la tabla 2, la expresión (x=x+3) +4 tiene el atributo tacode.

t1 = x +3
x = t1
t2 = t1 + 4

Regla gramatical	Reglas semánticas
$exp_1 \rightarrow$ **id** = exp_2	exp_1 .name = exp_2 .name exp_1 .tacode = exp_2 .tacode ++ **id** .strval || "=" || exp_2 .name
$exp \rightarrow aexp$	exp .name = aexp .name exp .tacode = aexp .tacode
$aexp_1 \rightarrow aexp_2$ + factor	$aexp_1$ .name = newtemp( ) $aexp_1$ .tacode = $aexp_2$ .tacode ++ factor .tacode ++ $aexp_1$ .name || "=" || $aexp_2$ .name || "+" || factor .name
$aexp \rightarrow factor$	aexp .name = factor .name aexp .tacode = factor .tacode
$factor \rightarrow$ (exp)	factor .name = exp .name factor .tacode = exp .tacode
$factor \rightarrow$ **num**	factor .name = **num** .strval factor .tacode = ""
$factor \rightarrow$ **id**	factor .name = **id** .strval factor .tacode = ""

(Esto supone que newtemp( ) es llamado en postorden y genera nombres temporales comenzando con t1.) Advierta coma la asignación x=x+3 genera dos instrucciones de tres direcciones utilizando un elemento temporal. Esto es una consecuencia del hecho de que la evaluación de atributos siempre crea un elemento temporal para calla subexpresión, incluyendo los lados derechos de las asignaciones.

Generación de Código Práctica

Las técnicas estándar de generación de código involucran modificaciones de los recorridos postorden del árbol sintáctico implicado por las gramáticas con atributos de Los ejemplos precedentes o, si no se genera un árbol sintáctico de manera explícita, acciones equivalentes. Durante un análisis sintáctico. El algoritmo básico puede ser descrito como el siguiente procedimiento recursivo (para nodos de árbol con dos hijos como máximo, pero fácilmente extensibles a más):

```
procedure genCode ( T: treenode );
begin
if T no es nil then
genere código pares preparar en el caso del código del hijo izquierdo de T ;
genCode (hijo izquierdo de T);
genere código para preparar en el caso del código del hijo derecho de T;
genCode (hijo derecho de T) ;
genere código para implementar la acción de T ;
end;
```

Observe que este procedimiento recursivo no solo tiene un componente postorden (que: genera código para implementar la acción de 7) sino también un componente de preorden y un componente enorden (que genera el código de preparación para los hijos izquierdo

y: derecho de T). En general, cada acción que representa T requerirá una versión un poco diferente del código de preparaci6n preorden y enorden.

Generación de código objetivo a partir del código intermedio

Si un compilador genera código intermedio, ya sea directamente durante en análisis sintáctico o desde un árbol sintáctico, entonces debe efectuarse otro paso en el código intermedio para generar el código objetivo final (por lo regular después de algún procesamiento adicional del código intermedio). Este paso puede ser bastante complejo por sí mismo, en particular si el código intermedio es muy simb6lico y contiene poca o ninguna información acerca de la máquina objetivo o el ambiente de ejecuci6n. En este caso, el paso de la generación de código final debe suministrar todas las ubicaciones reales de variables y temporales, mas el código necesario para mantener el ambiente de ejecución. Una cuesti6n particularmente importante es la asignaci6n apropiada de los registros y el mantenimiento de la información sobre el use de los mismos (es decir, cuales registros se encuentran disponibles y cuales contienen valores conocidos). Aplazaremos un análisis detallado de tales cuestiones de asignación hasta más adelante en este capítulo. Por ahora comentaremos solo técnicas generales para este proceso.

Por lo regular la generación de. Código a partir del código intermedio involucra alguna o ambas de las dos técnicas estándar: Expansión de macro y simulación estática. La expansión de: Macro involucra el reemplazo de cada clase de instrucción del código intermedio con una secuencia equivalente de instrucciones de código objetivo. Esto requiere que el compilador se mantenga al tanto de las decisiones acerca de ubicaciones e idiomas de código en estructuras de datos separadas y que los procedimientos del macro varíen la secuencia de código como se requiera mediante las clases particulares de datos involucradas en la instrucción de código intermedio. De este modo, cada paso puede ser mucho más complejo que las formas simples de expansión de macro disponibles del preprocesador de C o ensambladores de macro. La simulación estática involucra una simulación en línea recta de los efectos del código intermedio y generación de código objetivo para igualar estos efectos. Esto también requiere de más estructuras de datos, y puede variar de formas muy simples de seguimiento empleadas en conjunto con la expansión de macro, pasta la altamente sofisticada interpretación abstracta (que mantiene los valores algebraicamente a medida: que son calculados).

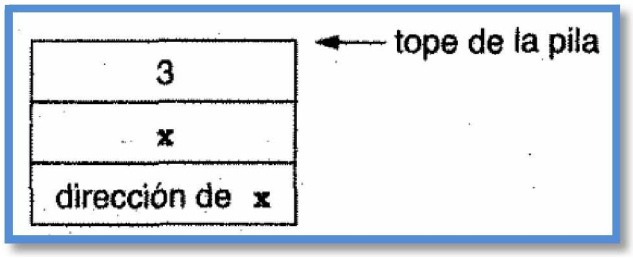
Podemos obtener alguna idea de los detalles de estas técnicas considerando el problema de traducir desde el código P at código de tres direcciones y viceversa. Consideremos la pequeña gramática de expresión que hemos estado utilizando como ejemplo de ejecución en esta sección, y consideremos la expresión (x=x+3) +4. Consideremos primero la traducción del código P para esta expresión:

```
lda x
lod x
ldc 3
adi
stn
ldc 4
adi
```

En su correspondiente código de tres direcciones

```
t1 = x +3
x  = t1
t2 = t1 + 4
```

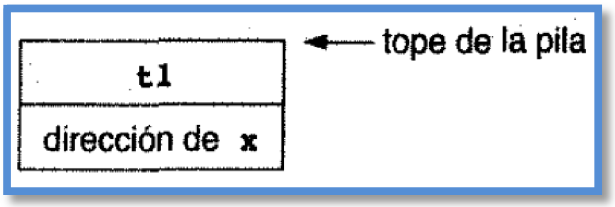
Esto requiere que realicemos una simulación estática de la pila de maquina P para encontrar equivalentes de tres direcciones del código dado. Hacemos esto con una estructura de datos de pila real durante la traducción. Después de las primeras tres instrucciones de código P, no se han generado todavía instrucciones de tres direcciones, pero la pila de maquina P se ha modificado para reflejar Las cargas, y la pila tiene el aspecto siguiente:



Ahora, cuando se procesa la operación adi, se genera la instrucción de tres direcciones

t1 = x + 3

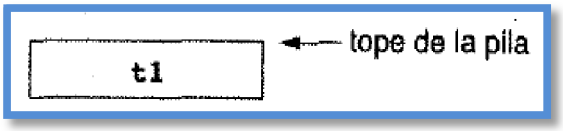
Y la pila se cambia a



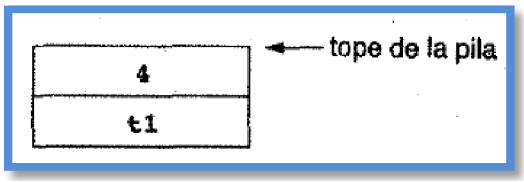
La instrucción stn provoca entonces que se genere la instrucción de tres direcciones

x = t1

Y la pila se cambie a



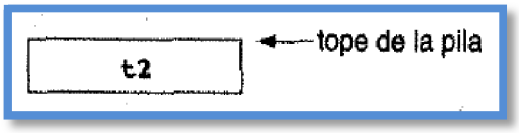
La instrucción siguiente inserta la constante 4 en la pila:



Finalmente, la instrucción adi provoca que se genere la instrucci6n de tres direcciones

t2 = t1 + 4

Y la pila se cambie a



Esto completa la simulación estática y la traducción.

Ahora consideraremos el caso de traducir del código de tres direcciones al código P. Si ignoramos la complicación agregada de los nombres temporales, esto puede hacerse mediante expansión simple de macro. Por consiguiente, una instrucci6n de tres direcciones

```
a = b + c
```

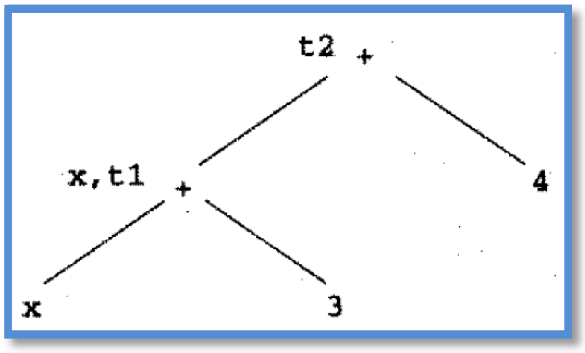
Siempre se puede traducir en la secuencia de código P

```
lda a
lod b    ; o ldc b si b es urea constante
lod c    ; o ldc c si c es una constante
adi
sto
```

Esto resulta en la siguiente traducción (algo insatisfactoria) del anterior código de tres direcciones en código P:

```
lda t1
lod x
ldc 3
adi
sto
lda x
lodt 1
sto
lda t2
lod t1
ldc 4
adi
sto
```

Si queremos eliminar los elementos temporales extra, entonces debemos utilizar un esquema más sofisticado que la expansión de macro pura. Una posibilidad es generar un nuevo árbol a partir del código de tres direcciones, indicando el efecto que el código tiene al etiquetar los nodos del árbol tanto con el operador de cada instrucción coma con el nombre que se le asigna. Esto puede visualizarse como una forma de simulación estática, y el árbol resultante para el anterior código de tres direcciones es



Advierta c6mo la instrueci6n de tres .direcciones(x = t1) no provoca que se creen nodos extra en este árbol, Pero sí que el nodo con nombre t1: adquiera el nombre adicional x. Este árbol es similar, pero no idéntico, al árbol sintáctico de la expresión original. El código P se puede generar a partir de este árbol de manera muy semejante a como se genera el código P a partir de un árbol sintáctico, de la manera antes descrita, pero con elementos temporales eliminados al hacer asignaciones s6lo a nombres permanentes de nodos interiores. De este modo, en el árbol de muestra, sólo se asigna x, los nombres t1 y t2 nunca se emplean en el código P generado, y el valor correspondiente al nodo raíz (con nombre t2) se deja en la pila de la maquina P. Esto produce exactamente la misma generación de código P que antes, aunque se utiliza stn en lugar de sto siempre que se realiza un almacenamiento.