Generación de código intermedio

Pedro O. Pérez M., PhD

Desarrollo de aplicaciones avanzadas de ciencias computacionales
Módulo 3: Compiladores
Tecnológico de Monterrey

pperezm@tec.mx

02-2024

1. 6.1 - 6.3
   - Introducción
   - 6.1 Variantes de árboles de sintaxis
   - 6.2 Código de tres direcciones
   - 6.3 Tipos y declaraciones

2. 6.4 - 6.6
   - 6.4 Traducción de expresiones
   - 6.5 Verificación de tipo
   - 6.6 Control de flujo

En el modelo análisis-síntesis de un compilador, el front end analiza el código fuente y crea una representación intermedia, de la cual el back end generar el código destino.
Esta aproximación permite crear una gran cantidad de compiladores con muy poco esfuerzo, ya que sólo se necesita escribir m front ends y n back ends para generar $m * n$ compiladores.

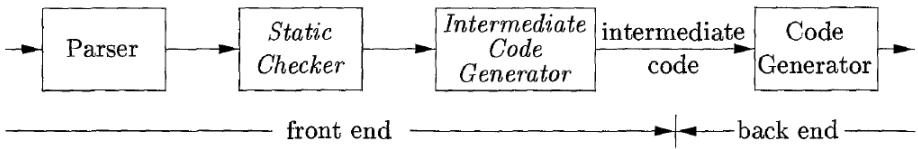


Figure 6.1: Logical structure of a compiler front end

Donde...

- La revisión estática incluye la revisión de tipo, la cual asegura que los operadores son aplicados a operadores compatibles. Esta revisión también incluye temas como en *break* de C (debe sr invocado dentro de un ciclo, sino habría que marcar error).

- Este esquema permite usar un gran gama de representaciones intermedias como pueden ser los árboles de análisis y el código de tres direcciones.
- El código de tres direcciones es el conjunto de instrucciones cuya forma general es $x = y\ op\ z$ (tres direcciones: dos para los operandos, y y z, uno para el resultado, x).

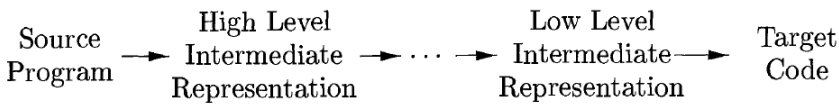


Figure 6.2: A compiler might use a sequence of intermediate representations

Grafos direcciones acíclicos (Direct Acyclic Graphs - DAG) para expresiones

Al igual que los árboles de expresión, los DAG tiene hojas que corresponden a operadores atómicos y los nodos interiores corresponden a operadores. La diferencia es un nodo N en un DAG puede tener más de un padre, sí N representa una subexpresión común.

Example 6.1 : Figure 6.3 shows the DAG for the expression

```
a + a * (b - c) + (b - c) * d
```

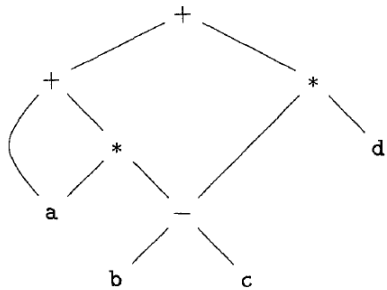


Figure 6.3: Dag for the expression `a + a * (b - c) + (b - c) * d`

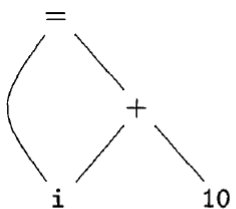
	PRODUCTION	SEMANTIC RULES
1)	$E \rightarrow E_1 + T$	$E.node =$ **new** $Node('+', E_1.node$
2)	$E \rightarrow E_1 - T$	$E.node =$ **new** $Node('-', E_1.node$
3)	$E \rightarrow T$	$E.node = T.node$
4)	$T \rightarrow (E)$	$T.node = E.node$
5)	$T \rightarrow$ **id**	$T.node =$ **new** $Leaf($**id**, **id**.*entry*$)$
6)	$T \rightarrow$ **num**	$T.node =$ **new** $Leaf($**num**, **num**.v

Figure 6.4: Syntax-directed definition to produce syntax tre

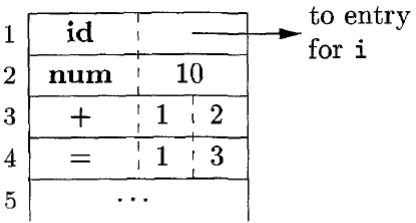
1) $p_1 = Leaf($**id**, *entry-a*$)$
2) $p_2 = Leaf($**id**, *entry-a*$) = p_1$
3) $p_3 = Leaf($**id**, *entry-b*$)$
4) $p_4 = Leaf($**id**, *entry-c*$)$
5) $p_5 = Node('-', p_3, p_4)$
6) $p_6 = Node('*', p_1, p_5)$
7) $p_7 = Node('+', p_1, p_6)$
8) $p_8 = Leaf($**id**, *entry-b*$) = p_3$
9) $p_9 = Leaf($**id**, *entry-c*$) = p_4$
10) $p_{10} = Node('-', p_3, p_4) = p_5$
11) $p_{11} = Leaf($**id**, *entry-d*$)$
12) $p_{12} = Node('*', p_5, p_{11})$
13) $p_{13} = Node('+', p_7, p_{12})$

Figure 6.5: Steps for constructing the DAG of Fig. 6.3

Antes de crear un nuevo nodo, se revisa que no existe un nodo con la misma etiqueta *op* e hijos *left* y *right*. Si existe, se regresa ese nodos; en caso contrario, se crea un nuevo nodo.



(a) DAG



(b) Array.

Figure 6.6: Nodes of a DAG for $i = i + 10$ allocated in an array

Suppose that nodes are stored in an array, as in Fig. 6.6, and each node is referred to by its value number. Let the *signature* of an interior node be the triple $\langle op, l, r \rangle$, where op is the label, l its left child's value number, and r its right child's value number. A unary operator may be assumed to have $r = 0$.

Algorithm 6.3: The value-number method for constructing the nodes of a DAG.

INPUT: Label op, node l, and node r.

OUTPUT: The value number of a node in the array with signature $\langle op, l, r \rangle$.

METHOD: Search the array for a node M with label op, left child l, and right child r. If there is such a node, return the value number of M. If not, create in the array a new node N with label op, left child l, and right child r, and return its value number. $\square$

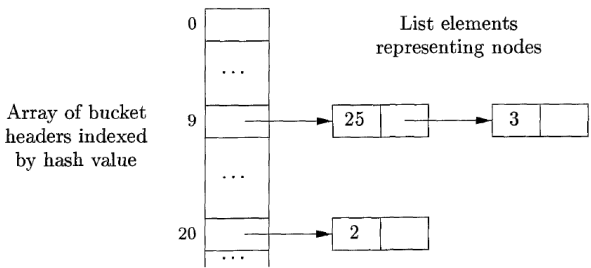


Figure 6.7: Data structure for searching buckets

- Buscar a lo largo de todo el arreglo es ineficiente, es mejor usar una tabla de dispersión.
- Para ello, sería necesario construir un función de dispersión basada en la tripleta $< op, I, r >$, que llamaremos $h(op, I, r)$.

Ejercicio 6.1

Construye el DAG para la siguiente expresión:

$$((x + y) - ((x + y) * (x - y))) + ((x + y) * (x - y))$$

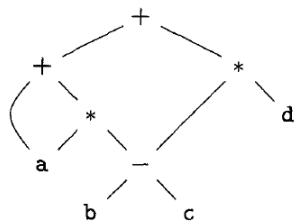
En el código de tres direcciones, hay a los más un operador en el lado derecho de la instrucción; es decir, no se permiten expresiones aritméticas acumuladas. Por ejemplo, la expresión $x + y * z$ debe ser traducido en la secuencia de tres direcciones:

$$t_1 = y * z$$
$$t_2 = x + t_1$$

donde t_1 y t_2 son nombre temporales generados por el compilador.

Esta separación de las expresiones aritméticas de múltiples operadores y de las declaraciones de flujo de control anidadas hace que el código de tres direcciones sea deseable para la generación y optimización del código de destino. El uso de nombres para los valores intermedios calculados por un programa permite reorganizar fácilmente el código de tres direcciones.

Example 6.4: Three-address code is a linearized representation of a syntax tree or a DAG in which explicit names correspond to the interior nodes of the graph. The DAG in Fig. 6.3 is repeated in Fig. 6.8, together with a corresponding three-address code sequence. □



(a) DAG

$t_1 = b - c$
$t_2 = a * t_1$
$t_3 = a + t_2$
$t_4 = t_1 * d$
$t_5 = t_3 + t_4$

(b) Three-address code

Figure 6.8: A DAG and its corresponding three-address code

El código de tres direcciones está construido en base a dos conceptos: direcciones e instrucciones.
Una dirección puede ser uno de los siguientes:

- **Nombre.** Por conveniencia, empleamos los nombres del código fuente aparecen como direcciones. En implementación, los nombres del código son reemplazados por un apuntador a la tabla de símbolos.
- **Constante.** En la práctica, un compilador debe trabajar con diferentes tipos de constantes y variables.
- **Temporal generada por el compilador.** Es muy útil, sobre todo para la optimización.

Antes de presentar las instrucciones de código de tres direcciones más comunes, debemos hablar sobre `etiquetas simbólicas`. Las etiquetas simbólicas son usadas por las instrucciones para alterar el control del flujo del programa.

Estas son las instrucciones de código de tres direcciones más comunes:

- Asignación de la forma $x = y\ op\ z$ donde *op* es un operador aritmético o lógico, y x, y y z son direcciones.
- Asignación de la forma $x = opy$, donde *op* es una operación unitaria. Esencialmente, las operaciones unitarias incluyen el signo $-$ unitario, negación lógica, operadores de cambio (`shift`) y operadores de conversión.
- Instrucción de copia de la forma $x = y$, donde a x se le asigna el valor de y.
- Un salto incondicional *goto* L. La instrucción de código de tres direcciones con la etiqueta L es la siguiente a ser ejecutada.

- Saltos condicionales de la forma *if x goto L* y *ifFalse x goto L*. Estas instrucciones ejecutan la instrucción con la etiqueta L si x es verdadero y falso, respectivamente. De otra forma, la siguiente instrucción de la secuencia es ejecutada.
- Saltos condicionales de la forma *if x relop y goto L*, la cual aplica un operador relacional ($<$, $<=$, $>$, etc.) a x y y, y ejecuta la instrucción con la etiqueta L si la condición es verdadera. Si no, se ejecuta la siguiente instrucción de la secuencia.

- Las llamadas a procedimiento y regreso son implementados usando las siguiente instrucciones: *param* x para parámetros; *call* p, n para llamadas a procedimientos o funciones, respectivamente; y regresa y, donde y, representando un valor de regreso, es opcional. Su uso típico es como secuencia de código de tres direcciones:
  param x_1
  param x_2
  ...
  param x_n
  call p, n

- Instrucciones de copia indexadas de la forma $x = y[i]$ y $x[i] = y$. La instrucción $x = y[i]$ asignas a x el valor en la `i-ésima` locación después de la localidad y. La instrucción $x[i] = y$ asigna el contenido de la `i-ésima` localidad más allá de x el valor de y.
- Asignaciones a punteros y direcciones de la forma $x = \&y$, $x = *y$, y $*x = y$. La instrucción $x = \&y$ asigna la localidad de y de `l-value` al `r-value` de x. Presumiblemente y es un nombre, quizás temporal, que indica una expresión con un `l-value` tal como $A[i][j]$. y x es el nombre de un apuntador o temporal. En la instrucción $x = *y$, presumiblemente y es un apuntador o temporal cuyo `r-value` es un localidad. El `r-value` de x copia el valor del contenido de esa localidad. Finalmente, $*x = y$ asigna el `r-value` de el objeto apuntado por x al valor `r-value` de y.

```
do i = i+1; while (a[i] < v);
```

```
L:  t1 = i + 1
    i = t1
    t2 = i * 8
    t3 = a [ t2 ]
    if t3 < v goto L
```

(a) Symbolic labels.

```
100:  t1 = i + 1
101:  i = t1
102:  t2 = i * 8
103:  t3 = a [ t2 ]
104:  if t3 < v goto 100
```

(b) Position numbers.

Figure 6.9: Two ways of assigning labels to three-address statements

La descripción del código de tres direcciones especifica los componentes de cada tipo de instrucción, pero no especifica la estructura de datos con el que se representan estas instrucciones. Existen tres formas de representarlas: cuádruples, triples y triples indirectas.

Un cuádruple tiene cuatro campos, las cuales son *op*, arg_1, arg_2, y *result*. El campo *op* contiene el código interno del operador. Por ejemplo, $x = y + z$ es representado colocando $+$ como *op*, y en arg_1, z in arg_2, y x en *result*. Aunque existen algunas excepciones son:

- Instrucciones con operadores unitarios como $x = minus\, y$ o $x = y$ no usan arg_2.
- Operadores como *param* no usan ni arg_2, ni *result*.
- Saltos condicionales o incondicionales ponen la etiqueta del salto en *result*.

```
a = b*-c+b*-c;
```

```
t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
 a = t5
```

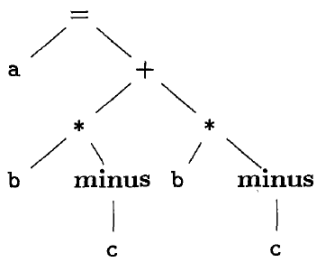
(a) Three-address code

	op	arg_1	arg_2	*result*
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
		...		

(b) Quadruples

Figure 6.10: Three-address code and its quadruple representation

Un triple tiene solo tres campos, los cuales llamaremos *op*, arg_1, y arg_2. Nota que el campo *result* en la figura previa es usada principalmente para nombres temporales. Usando triples, referimos a los resultados de una operación $x\ op\ y$ por su posición, más que por un nombre temporal explícito. Es decir, en vez de t_1, en representación triple se referencia por la posición (0).



(a) Syntax tree

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
	...		

(b) Triples

Figure 6.11: Representations of $a + a * (b - c) + (b - c) * d$

Una ventaja del cuadrúple sobre los triples se observa en la etapa de optimización, donde reorganizan las instrucciones generadas. Con un cuadrúple, si movemos la instrucción que calcula una variable temporal t, entonces las instrucciones que usan a t no requieren cambio. En un triple, dado que el resultado de una operación es referenciado por su posición, mover las instrucciones implica cambiar todas las referencias.

Triples indirectos consiste de una lista de apuntadores a triples, más que una lista de triples mismas. Con triples indirectos, una optimización puede mover una instrucción reordenando las listas de instrucciones, sin afectar las triples mismas.

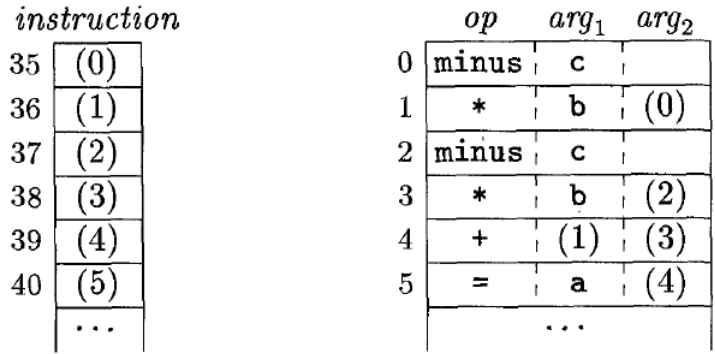


Figure 6.12: Indirect triples representation of three-address code

Forma estática de asignación única

La forma estática de asignación única (**Static Single-Assigment Form**, **SSA**)es una representación intermedia que facilita la optimización de código. Dos aspectos importantes distinguen a SSA del código de tres direcciones:

- La primera es que todas las asignaciones en SSA son variables con nombre distinto.

```
p = a + b
q = p - c
p = q * d
p = e - p
q = p + q
```

(a) Three-address code.

$$
\begin{aligned}
p_1 &= a + b \\
q_1 &= p_1 - c \\
p_2 &= q_1 * d \\
p_3 &= e - p_2 \\
q_2 &= p_3 + q_1
\end{aligned}
$$

(b) Static single-assignment form.

Figure 6.13: Intermediate program in three-address code and SSA

Pero, ¿qué pasa cuando tenemos una instrucción como ésta?

```
if ( flag ) x = -1; else x = 1;
y = x * a;
```

Aquí entra la segunda distinción de SSA: se usa una notación convencional llamada ϕ-function para combinar esta dos distinciones:

```
if ( flag ) x_1 = -1; else x_2 = 1;
x_3 = φ(x_1, x_2);
```

Exercise 6.2.1 : Translate the arithmetic expression $a + -(b + c)$ into:

a) A syntax tree.

b) Quadruples.

c) Triples.

d) Indirect triples.

La aplicación de los tipos pueden ser agrupados bajo la verificación y traducción:

- Verificación de tipo use reglas lógicas para determinar el comportamiento de un programa a tiempo de ejecución. Específicamente, asegurar que los tipos de los operadores empaten los tipos esperados por un operador.
- Aplicaciones para la traducción. Desde el tipo de una variable, un compilador puede determinar el espacio de almacenamiento que se necesita para esa variable al tiempo de ejecución. La información del tipo se necesita también para calcular la dirección señalada por la referencia de un arreglo, insertar conversiones de tipo explícita, y para elegir la versión correcta de un operador aritmético, entre otras cosas.

Los tipos tienen estructura, los cuales podemos representar usando `expresión tipo`: puede ser un tipo de dato básico o estar formado por la aplicación de un operador llamado `constructor tipo` a una expresión tipo.

Podemos definir las siguientes definiciones de expresiones tipo:

- Un tipo básico es una expresión tipo.
- Un nombre tipo es una expresión tipo.
- Una expresión tipo puede ser formado al aplicar el constructor tipo `array` de un número y una expresión tipo.
- Un registro es una estructura de datos con campos. Un expresión tipo pueden ser formado aplicando el constructor tipo `record` a los campos y sus tipos.
- UNa expresión tipo puede ser formado usando el constructor $\rightarrow$ para función tipo. Escribimos $s \rightarrow t$ para "función del tipo s al tipo t".

- Si s y t son expresión tipo, entonces su producto cartesiano sxt es una expresión tipo. Este producto puede ser representada una lista o tupla de tipos (por ejemplo, parámetros de una función).
- La expresión tipo puede contener variables cuyos valores son expresiones tipo.

Equivalencia de tipo

¿Cuán podemos decir que dos expresiones tipo son equivalentes? Muchas reglas de verificación de tipo tiene de la forma: "si dos expresiones tipo son iguales entonces regresa un cierto tipo si no, error".
Una forma conveniente de representar una expresión tipo es través de un grafo.
Si empleamos esta representación, dos tipos son estructuralmente equivalente si y sólo si una de las siguientes condiciones es cierta:

- Si son del mismo tipo básico.
- Están formados a través de la aplicación del mismo constructor a tipos estructuralmente equivalentes.
- Uno es un nombre tipo que denota al otro.

Esquema de almacenamiento para nombres locales

Por el tipo de un nombre, es posible determinar la cantidad de almacenamiento que será necesario para ese nombre al tiempo de ejecución. Al momento de compilación, podemos usar estas cantidades para asignar a cada nombre una dirección relativa. El tipo y la dirección relativos son almacenados en la tabla de símbolos para el nombre.

$$
\begin{array}{lcll}
T & \rightarrow & B & \{\ t = B.type;\ w = B.width;\ \} \\
 & & C & \\
B & \rightarrow & \textbf{int} & \{\ B.type = integer;\ B.width = 4;\ \} \\
B & \rightarrow & \textbf{float} & \{\ B.type = float;\ B.width = 8;\ \} \\
C & \rightarrow & \epsilon & \{\ C.type = t;\ C.width = w;\ \} \\
C & \rightarrow & [\ \textbf{num}\ ]\ C_1 & \{\ array(\textbf{num}.value,\ C_1.type); \\
 & & & \quad C.width = \textbf{num}.value \times C_1.width;\ \}
\end{array}
$$

Figure 6.15: Computing types and their widths

$$
\begin{array}{lll}
P \rightarrow & & \{\ offset = 0;\ \} \\
 & D & \\
D \rightarrow & T\ \mathbf{id}\ ; & \{\ top.put(\mathbf{id}.lexeme,\ T.type,\ offset); \\
 & & \quad offset\ =\ offset + T.width;\ \} \\
 & D_1 & \\
D \rightarrow & \epsilon &
\end{array}
$$

Figure 6.17: Computing the relative addresses of declared names

$$
\begin{array}{lll}
T \rightarrow \textbf{record}\ '\{' & \{\ \mathit{Env.push}(\mathit{top});\ \mathit{top} = \textbf{new}\ \mathit{Env}(); & \\
& \quad \mathit{Stack.push}(\mathit{offset});\ \mathit{offset} = 0;\ \} & \\
\quad D\ '\}' & \{\ T.\mathit{type} = \mathit{record}(\mathit{top});\ T.\mathit{width} = \mathit{offset}; & \\
& \quad \mathit{top} = \mathit{Env.pop}();\ \mathit{offset} = \mathit{Stack.pop}();\ \} &
\end{array}
$$

Figure 6.18: Handling of field names in records

Exercise 6.3.1: Determine the types and relative addresses for the identifiers in the following sequence of declarations:

```
float x;
record { float x; float y; } p;
record { int tag; float x; float y; } q;
```

PRODUCTION	SEMANTIC RULES
$S \rightarrow \mathbf{id} = E\ ;$	$S.code = E.code \mid\mid$ $gen(top.get(\mathbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \mathbf{new}\ Temp()$ $E.code = E_1.code \mid\mid E_2.code \mid\mid$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
$\mid\ - E_1$	$E.addr = \mathbf{new}\ Temp()$ $E.code = E_1.code \mid\mid$ $gen(E.addr\ '='\ '\mathbf{minus}'\ E_1.addr)$
$\mid\ (\ E_1\ )$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\mid\ \mathbf{id}$	$E.addr = top.get(\mathbf{id}.lexeme)$ $E.code = ''$

Figure 6.19: Three-address code for expressions

Example 6.11: The syntax-directed definition in Fig. 6.19 translates the assignment statement `a = b + - c ;` into the three-address code sequence

El atributo de código suelen ser cadenas de caracteres extremadamente largas, así que usualmente son generadas de manera incremental.

$$
\begin{array}{lll}
S \rightarrow \textbf{id} = E\ ; & \{\ gen(\ top.get(\textbf{id}.lexeme)\ '='\ E.addr);\ \} \\
E \rightarrow E_1 + E_2 & \{\ E.addr = \textbf{new}\ Temp\,();\ gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr);\ \} \\
\mid\ - E_1 & \{\ E.addr = \textbf{new}\ Temp\,();\ gen(E.addr\ '='\ '\textbf{minus}'\ E_1.addr);\ \} \\
\mid\ (\ E_1\ ) & \{\ E.addr = E_1.addr;\ \} \\
\mid\ \textbf{id} & \{\ E.addr = top.get(\textbf{id}.lexeme);\ \}
\end{array}
$$

Figure 6.20: Generating three-address code for expressions incrementally

Direccionando elementos de un arreglo

En C y Java, los arreglos están numerados $0, 1, 2, ..., n-1$, para un arreglo de n elementos. Si el ancho (tamaño) de cada elemento del arreglo es w, entonces el i-ésimo elemento del arreglo A se encuentra la localidad:

$$base + i * w$$

donde *base* es la dirección relativa de almacenamiento del arreglo. Esta fórmula se puede generalizar para dos o más dimensiones. Por ejemplo, para acceder la localidad $A[i_1][1_2]$, tendremos que calcular:

$$base + i_1 * w_1 + i_2 * w_2$$

En general, para k dimensiones:

$$base + i_1 * w_1 + i_2 * w_2 + ... + i_k * w_k$$

Alternativamente, se puede emplear el número de elementos, n_j a lo largo de la dimensión j para calcular la dirección. Es decir, si queremos acceder la localidad $A[i_1][i_2]$:

$$base + (i_1 * n_2 + i_2) * w$$

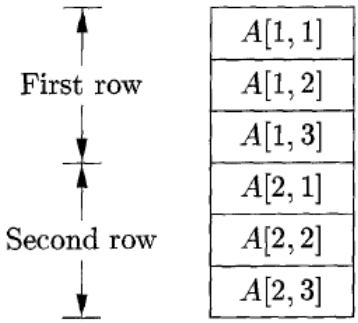
Si tenemos k dimensiones: En general, para k dimensiones:

$$base + ((...(i_1 * n_2 + i_2) * n_3 + i_3)... * n_k + i_k) * w$$

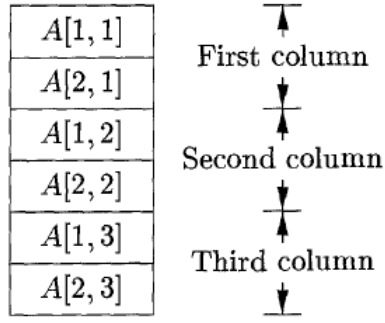
En mucho lenguajes, la posición de los elementos de un arreglo empiezan con 0. Por ejemplo, en Pascal, un arreglo unidimensional puede ser numerado $low, low + 1, ..., high$. En este caso $base$ es relativa a la posición de $A[low]$:

$$base + (i - low) * w$$

Aunque también se puede reescribir como $i * w + c$ donde $c = base - low * w$ puede ser precalculado en tiempo de ejecución.
Sin embargo, el uso del precálculo no siempre se puede usar. Si tenemos un arreglo de tamaño dinámico al tiempo de compilación, no podemos calcular constantes como c.



(a) Row Major



(b) Column Major

Figure 6.21: Layouts for a two-dimensional array.

Traducción de referencias de arreglos

El no terminal L tiene tres atributos sintetizados:

1. *L.addr* indica un temporal que es usado para calcular el desplazamiento de un arreglo usando los términos $i_j x w_j$.
2. L.array es un apuntador a la tabla de símbolos. La dirección base del arreglo, *L.array.base* es usado para terminar el actual $I - value$ de una referencia de arreglo después de todos los índices son analizados.
3. *L.type* es el tipo de un subarreglo generado por *L*.

$$
\begin{array}{lll}
S \rightarrow \textbf{id} = E\ ; & \{\ gen(\ top.get(\textbf{id}.lexeme)\ '='\ E.addr);\ \} \\
\mid\ L = E\ ; & \{\ gen(L.addr.base\ '['\ L.addr\ ']'\ '='\ E.addr);\ \} \\
E \rightarrow E_1 + E_2 & \{\ E.addr = \textbf{new}\ Temp\,();\ gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr);\ \} \\
\mid\ \textbf{id} & \{\ E.addr = top.get(\textbf{id}.lexeme);\ \} \\
\mid\ L & \{\ E.addr = \textbf{new}\ Temp\,();\ gen(E.addr\ '='\ L.array.base\ '['\ L.addr\ ']');\ \} \\
L \rightarrow \textbf{id}\ [\ E\ ] & \{\ L.array = top.get(\textbf{id}.lexeme);\ L.type = L.array.type.elem;\ L.addr = \textbf{new}\ Temp\,();\ gen(L.addr\ '='\ E.addr\ '*'\ L.type.width);\ \} \\
\mid\ L_1\ [\ E\ ] & \{\ L.array = L_1.array;\ L.type = L_1.type.elem;\ t = \textbf{new}\ Temp\,();\ L.addr = \textbf{new}\ Temp\,();\ gen(t\ '='\ E.addr\ '*'\ L.type.width);\ gen(L.addr\ '='\ L_1.addr\ '+'\ t);\ \}
\end{array}
$$

Figure 6.22: Semantic actions for array references

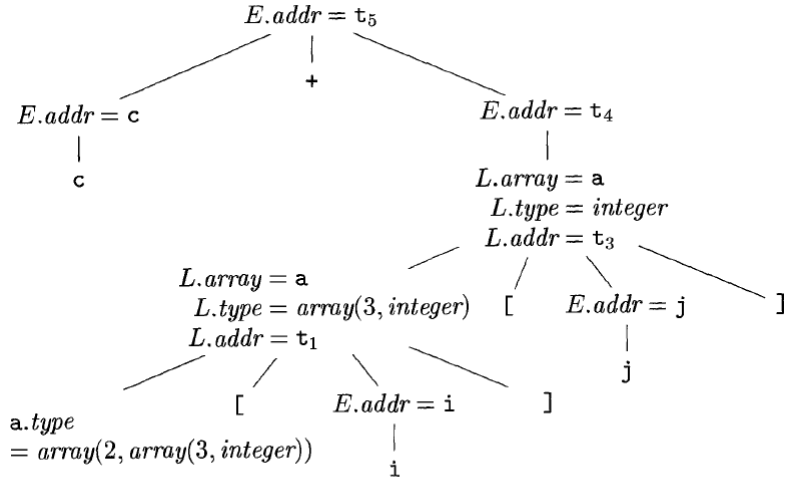


Figure 6.23: Annotated parse tree for `c + a[i][j]`

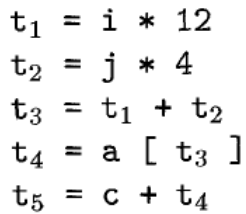


Figure 6.24: Three-address code for expression `c + a[i][j]`

Exercise 6.4.1: Add to the translation of Fig. 6.19 rules for the following productions:

a) $E \rightarrow E_1 * E_2$.

b) $E \rightarrow + E_1$ (unary plus).

PRODUCTION	SEMANTIC RULES
$S \rightarrow \mathbf{id} = E\ ;$	$S.code = E.code \mid\mid$ $gen(top.get(\mathbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \mathbf{new}\ Temp()$ $E.code = E_1.code \mid\mid E_2.code \mid\mid$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
$\mid\ - E_1$	$E.addr = \mathbf{new}\ Temp()$ $E.code = E_1.code \mid\mid$ $gen(E.addr\ '='\ '\mathbf{minus}'\ E_1.addr)$
$\mid\ (\ E_1\ )$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\mid\ \mathbf{id}$	$E.addr = top.get(\mathbf{id}.lexeme)$ $E.code = ''$

Figure 6.19: Three-address code for expressions

Exercise 6.4.3: Use the translation of Fig. 6.22 to translate the following assignments:

a) `x = a[i] + b[j]`.

Para hacer la verificación de tipo, un compilador necesita asignar una expresión tipo a cada componente del programa fuente.
La verificación de tipo puede tomar dos formas: sintetizado y heredado. El `tipo sintetizado` construye el tipo de una expresión a partir de los tipos de las subexpresiones.

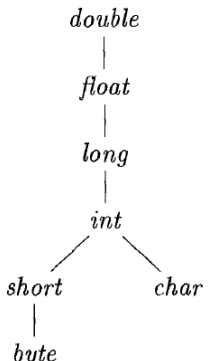
if f has type $s \to t$ **and** x has type s,
then expression $f(x)$ has type t

El `tipo inferido` determina el tipo de una construcción de lenguaje con base a la forma en que se usa.

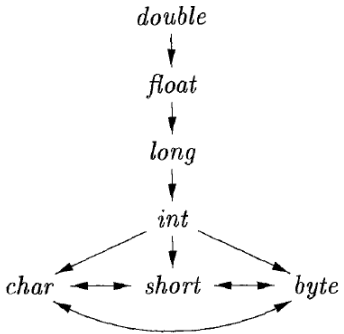
> **if** $f(x)$ is an expression,
> **then** for some α and β, f has type $\alpha \to \beta$ **and** x has type α

```
t1 = (float) 2
t2 = t1 * 3.14
```

if ($E_1.type = integer$ **and** $E_2.type = integer$) $E.type = integer$;
else if ($E_1.type = float$ **and** $E_2.type = integer$) $\cdots$
$\cdots$



(a) Widening conversions



(b) Narrowing conversions

Figure 6.25: Conversions between primitive types in Java

La acción semántica para verificación de $E \rightarrow E_1 + E - 2$ usa dos funciones:

1. $max(t_1, t_2)$ toma dos tipos, t_1 y t_2, y regresa el máximo de los dos tipos, de acuerdo a la jerarquía de ampliación.
2. $widen(a, t, w)$ genera conversiones de tipo is necesita .ªmpliarüna dirección a de tipo t e n un valor de tipo w. Regresa a is t y w son del mismo tipo. De otra forma, genera una instrucción para hacer la conversión y colocar el resultado en una t temporal, la cuál se regresa como resultado.

```
Addr widen(Addr a, Type t, Type w)
        if ( t = w ) return a;
        else if ( t = integer and w = float ) {
                temp = new Temp();
                gen(temp '=' '(float)' a);
                return temp;
        }
        else error;
}
```

Figure 6.26: Pseudocode for function *widen*

$$
\begin{aligned}
E \rightarrow E_1 + E_2 \quad \{\ & E.type = max(E_1.type, E_2.type); \\
& a_1 = widen(E_1.addr, E_1.type, E.type); \\
& a_2 = widen(E_2.addr, E_2.type, E.type); \\
& E.addr = \textbf{new}\ Temp\,(); \\
& gen(E.addr\ '='\ a_1\ '+'\ a_2);\ \}
\end{aligned}
$$

Figure 6.27: Introducing type conversions into expression evaluation

if f can have type $s_i \to t_i$, for $1 \leq i \leq n$, where $s_i \neq s_j$ for $i \neq j$
and x has type s_k, for some $1 \leq k \leq n$
then expression $f(x)$ has type t_k

INPUT: A program consisting of a sequence of function definitions followed by an expression to be evaluated. An expression is made up of function applications and names, where names can have predefined polymorphic types.

OUTPUT: Inferred types for the names in the program.

METHOD: For simplicity, we shall deal with unary functions only. The type of a function $f(x_1, x_2)$ with two parameters can be represented by a type expression $s_1 \times s_2 \rightarrow t$, where s_1 and s_2 are the types of x_1 and x_2, respectively, and t is the type of the result $f(x_1, x_2)$. An expression $f(a, b)$ can be checked by matching the type of a with s_1 and the type of b with s_2.

Informalmente, unificación es el problema de determinar cuando dos expresiones s y t pueden ser hechas idénticas sustituyendo expresiones por variables en s y t. Probar igual de expresiones es un caso especial de unificación: si s y t tienen constantes pero no variables, entonces s y t se unifican si y sólo si ellas son idénticas.

Algorithm 6.19: Unification of a pair of nodes in a type graph.

INPUT: A graph representing a type and a pair of nodes m and n to be unified.

OUTPUT: Boolean value true if the expressions represented by the nodes m and n unify; false, otherwise.

METHOD: A node is implemented by a record with fields for a binary operator and pointers to the left and right children. The sets of equivalent nodes are maintained using the *set* field. One node in each equivalence class is chosen to be the unique representative of the equivalence class by making its *set* field contain a null pointer. The *set* fields of the remaining nodes in the equivalence class will point (possibly indirectly through other nodes in the set) to the representative. Initially, each node n is in an equivalence class by itself, with n as its own representative node.

El algoritmo de unificación usa las siguientes dos operaciones sobre nodos:

- *find*(n) regresa el nodo representativo de la clase equivalente que actualmente contiene el nodo n.
- *union*(m, n) mezcla las clases equivalente conteniendo los nodos m y n. Si uno de los representativos para las clases equivalente de m y n es un nodo no variable, *union* hace que el nodo no variable sea el representativo de mezclas las clases equivalentes.

```
boolean unify(Node m, Node n) {
    s = find(m); t = find(n);
    if ( s = t ) return true;
    else if ( nodes s and t represent the same basic type ) return true;
    else if (s is an op-node with children s1 and s2 and
                t is an op-node with children t1 and t2) {
        union(s, t);
        return unify(s1, t1) and unify(s2, t2);
    }
    else if s or t represents a variable {
        union(s, t);
        return true;
    }
    else return false;
}
```

La traducción de instrucciones tales como `if-else` y `while` está ligada a la traducción de expresiones booleanas. En los lenguajes de programación, las expresiones booleanas son comúnmente usados para:

- `Alterar el control del flujo.`
- `Calcular valores lógicos.`

Las expresiones booleanas están compuesta por operadores booleanos (AND, OR y NOT) aplicado a elementos que son variables booleanas o expresiones relacionales.

$$B \rightarrow B \,||\, B \mid B \,\&\&\, B \mid !\, B \mid (\, B\,) \mid E \,\mathbf{rel}\, E \mid \mathbf{true} \mid \mathbf{false}$$

```
if ( x < 100 || x > 200 && x != y ) x = 0;
```

```
        if x < 100 goto L2
        ifFalse x > 200 goto L1
        ifFalse x != y goto L1
L2:     x = 0
L1:
```

Figure 6.34: Jumping code

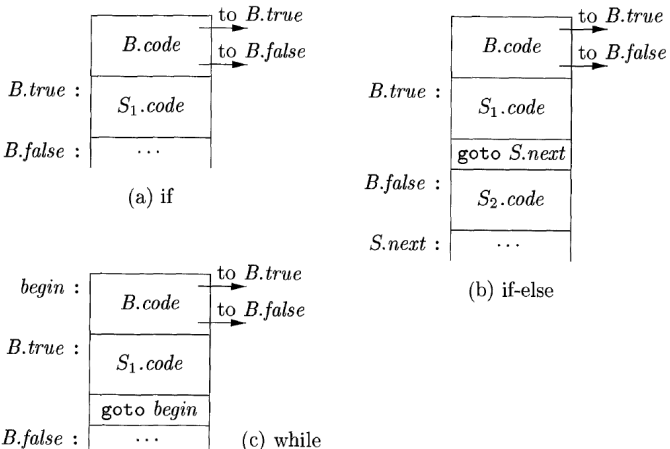


Figure 6.35: Code for if-, if-else-, and while-statements

PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \mid\mid label(S.next)$
$S \rightarrow$ **assign**	$S.code =$ **assign**$.code$
$S \rightarrow$ **if** (B) S_1	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \mid\mid label(B.true) \mid\mid S_1.code$
$S \rightarrow$ **if** (B) S_1 **else** S_2	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code$ $\mid\mid label(B.true) \mid\mid S_1.code$ $\mid\mid gen(\texttt{'goto'}\ S.next)$ $\mid\mid label(B.false) \mid\mid S_2.code$
$S \rightarrow$ **while** (B) S_1	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \mid\mid B.code$ $\mid\mid label(B.true) \mid\mid S_1.code$ $\mid\mid gen(\texttt{'goto'}\ begin)$
$S \rightarrow S_1\ S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \mid\mid label(S_1.next) \mid\mid S_2.code$

Figure 6.36: Syntax-directed definition for flow-of-control statements.

Traducción de control de flujo de expresiones booleanas

PRODUCTION	SEMANTIC RULES
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow B_1 \;\&\&\; B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.true) \mid\mid B_2.code$
$B \rightarrow ! B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1$ **rel** E_2	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen('\texttt{if}'\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ '\texttt{goto}'\ B.true)$ $\mid\mid gen('\texttt{goto}'\ B.false)$
$B \rightarrow$ **true**	$B.code = gen('\texttt{goto}'\ B.true)$
$B \rightarrow$ **false**	$B.code = gen('\texttt{goto}'\ B.false)$

Figure 6.37: Generating three-address code for booleans

Example 6.22: Consider again the following statement from Example 6.21:

```
if( x < 100 || x > 200 && x != y ) x = 0;        (6.13)
```

Using the syntax-directed definitions in Figs. 6.36 and 6.37 we would obtain the code in Fig. 6.38.

```
        if x < 100 goto L2
        goto L3
L3:     if x > 200 goto L4
        goto L1
L4:     if x != y goto L2
        goto L1
L2:     x = 0
L1:
```

Figure 6.38: Control-flow translation of a simple if-statement

Tomemos en cuenta el código `if` que revisamos antes:

Example 6.22: Consider again the following statement from Example 6.21:

```
if( x < 100 || x > 200 && x != y ) x = 0;          (6.13)
```

Using the syntax-directed definitions in Figs. 6.36 and 6.37 we would obtain the code in Fig. 6.38.

```
        if x < 100 goto L2
        goto L3
L3:     if x > 200 goto L4
        goto L1
L4:     if x != y goto L2
        goto L1
L2:     x = 0
L1:
```

Figure 6.38: Control-flow translation of a simple if-statement

$$
\begin{aligned}
B.true &= fall \\
B.false &= S_1.next \;=\; S.next \\
S.code &= B.code \;||\; S_1.code
\end{aligned}
$$

$$test = E_1.addr \ \mathbf{rel}.op \ E_2.addr$$

$$\begin{aligned} s = \ & \mathbf{if}\ B.true \neq fall\ \mathbf{and}\ B.false \neq fall\ \mathbf{then} \\ & \quad gen(\texttt{'if'}\ test\ \texttt{'goto'}\ B.true) \ ||\ gen(\texttt{'goto'}\ B.false) \\ & \mathbf{else\ if}\ B.true \neq fall\ \mathbf{then}\ gen(\texttt{'if'}\ test\ \texttt{'goto'}\ B.true) \\ & \mathbf{else\ if}\ B.false \neq fall\ \mathbf{then}\ gen(\texttt{'ifFalse'}\ test\ \texttt{'goto'}\ B.false) \\ & \mathbf{else}\ '' \end{aligned}$$

$$B.code = E_1.code \ ||\ E_2.code \ ||\ s$$

Figure 6.39: Semantic rules for $B \rightarrow E_1$ **rel** E_2

$$
\begin{aligned}
B_1.true &= \textbf{if } B.true \neq fall \textbf{ then } B.true \textbf{ else } newlabel() \\
B_1.false &= fall \\
B_2.true &= B.true \\
B_2.false &= B.false \\
B.code &= \textbf{if } B.true \neq fall \textbf{ then } B_1.code \mid\mid B_2.code \\
&\quad\ \textbf{else } B_1.code \mid\mid B_2.code \mid\mid label(B_1.true)
\end{aligned}
$$

Figure 6.40: Semantic rules for $B \rightarrow B_1 \mid\mid B_2$

```
if( x < 100 || x > 200 && x != y ) x = 0;
```

translates into the code of Fig. 6.41.

```
        if x < 100 goto L2
        ifFalse x > 200 goto L1
        ifFalse x != y goto L1
L2:     x = 0
L1:
```

Figure 6.41: If-statement translated using the fall-through technique