Compiladores
Facultad de Ciencias UNAM
Representaciones Intermedias

Lourdes Del Carmen González Huesca [*]

22 de abril de 2024

Resumen

En este punto del proceso de compilación se ha realizado el análisis léxico, que devolvió un *stream* de *tokens*, un análisis sintáctico que definió si el código era correcto de acuerdo a la gramática y un análisis semántico que determinó si el código era semánticamente válido bajo ciertas características. Por tanto se han cubierto todas las fases del *front-end* de un compilador. Para pasar a las fases siguientes es necesario transformar el código fuente en uno que facilite su manejo. A lo largo de esta nota se presentan algunas representaciones intermedias que puede tener el código y los tipos de éstas para cubrir diferentes objetivos.

Términos clave:
Es: Generación de código intermedio, Código de tres direcciones, Gráficas de dependencias de atributos, Gráficas de flujo.
En: Intermediate-code generation, Three-address code, Directed Acyclic Graphs for Expressions, Flow graphs.

Después de las fases de análisis en el front-end, el compilador ha obtenido suficiente información del programa fuente para realizar una traducción que facilite el manejo del programa en las fases de back-end. Esta información será utilizada para tener una **representación intermedia** (IR *Intermediate Representation*) que sea fiel al código fuente y se traduzca a código de bajo nivel.

Una representación intermedia es una estructura de datos que sólo existe en tiempo de compilación. Dependiendo del diseño e implementación del compilador es posible que se usen diferentes representaciones intermedias para mejorar las traducciones entre fases así como utilizar optimizaciones a la representación obtenida por el front-end para que el programa traducido y equivalente al fuente se ejecute de forma eficiente.

Las representaciones intermedias tienen como objetivos principales:

- simplificar el tratamiento de código y separar las fases de front-end y back-end;
- mantener el compilador modularizado para mejor implementación y manutención;
- facilitar optimizaciones independientes de la máquina.

Y tienen las siguientes propiedades:

- hacer más fácil la generación y manipulación de código intermedio;
- proveer una mejor abstracción del programa fuente.

[*]Material revisado por el servicio social de Apoyo a la Docencia y Asesoría Académica en la Facultad de Ciencias de la UNAM con clave 2023-12/12-292, de Diana Laura Nicolás Pavia.

Existen tres categorías de representaciones intermedias:

Estructurales o gráficas
: son representaciones orientadas a gráficas utilizadas en su mayoría por traductores entre lenguajes fuente.

Lineales
: representaciones que están dirigidas a máquinas abstractas y que son simples y compactas. El código compilado es representado por una secuencia ordenada de operaciones.

Híbridas
: estas son combinación de las anteriores para obtener los beneficios de ambas.

Código de tres direcciones

Esta representación intermedia o lenguaje intermedio divide expresiones en instrucciones más sencillas y cuya sintaxis es más parecida a las instrucciones de bajo nivel. Cada instrucción incluye asignaciones y operaciones básicas, contiene a lo más 4 items: tres operandos y un operador. Las asignaciones se realizan en direcciones relativas que serán más fáciles de traducir a direcciones físicas en una fase del back-end llamada asignación de registros. Las instrucciones siguen un orden lineal (estructurado) pero puede haber saltos (`goto`), bifurcaciones, condicionales, llamadas a funciones o subrutinas, accesos a memoria y asignaciones, etc.

Por ejemplo, consideremos un condicional sencillo y su traducción a código de tres direcciones:

```
if (x + y*z > x*y +z)          t1 = y*z
    a = 0;                     t2 = x+t1
                               t3 = x*y
                               t4 = t3+z
                               if (t2 <= t4) goto L
                               a = 0
                           L:
```

Este código secciona las operaciones del condicional en instrucciones sencillas usando direcciones temporales (`ti`) para almacenar los resultados intermedios. La etiqueta `L` indica la siguiente instrucción al término del condicional y permite continuar con la ejecución del resto del programa. Esta representación puede ser obtenida a partir de la gramática del lenguaje, sólo es cuestión de extender las reglas semánticas mediante acciones de traducción.

Ejemplo 1. Definición dirigida por la sintaxis para construir código de tres direcciones para expresiones que representan asignaciones, los atributos a usar son `code` para el código y `addr` para guardar la dirección que almacenará el valor de una expresión:

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

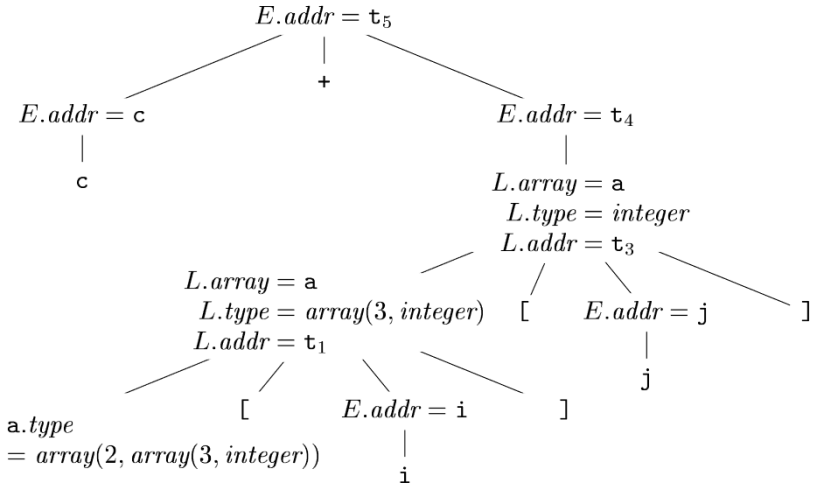
Ejemplo 2. La siguiente traducción obtiene una representación intermedia de tres direcciones en un arreglo de referencias. El arreglo se define mediante la gramática

$$L \to L\,[E] \;\mid\; \mathbf{id}\,[E]$$

que extiende a $S \to \mathbf{id} = E \mid L = E \qquad E \to E + E \mid \mathbf{id} \mid L.$
Los atributos usados son: `addr` para almacenar temporalmente el desplazo en el arreglo, `array` es el apuntador a la tabla de símbolos donde se almancena el arreglo y `type` es el tipo del subarreglo correspondiente:

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

Y el siguiente árbol es un parse tree decorado para la expresión `c+a[i][j]` donde `a` es un arreglo de enteros de dimensión 3×2 de tamaño 24 considerando que un entero es de tamaño 4:



El código de tres direcciones que representa esa misma expresión es:

$$
\begin{aligned}
\mathtt{t_1} &= \mathtt{i} * \mathtt{12} \\
\mathtt{t_2} &= \mathtt{j} * \mathtt{4} \\
\mathtt{t_3} &= \mathtt{t_1} + \mathtt{t_2} \\
\mathtt{t_4} &= \mathtt{a}[\mathtt{t_3}] \\
\mathtt{t_5} &= \mathtt{c} + \mathtt{t_4}
\end{aligned}
$$

Ejemplo 3. Three-Address Code [1]

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

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

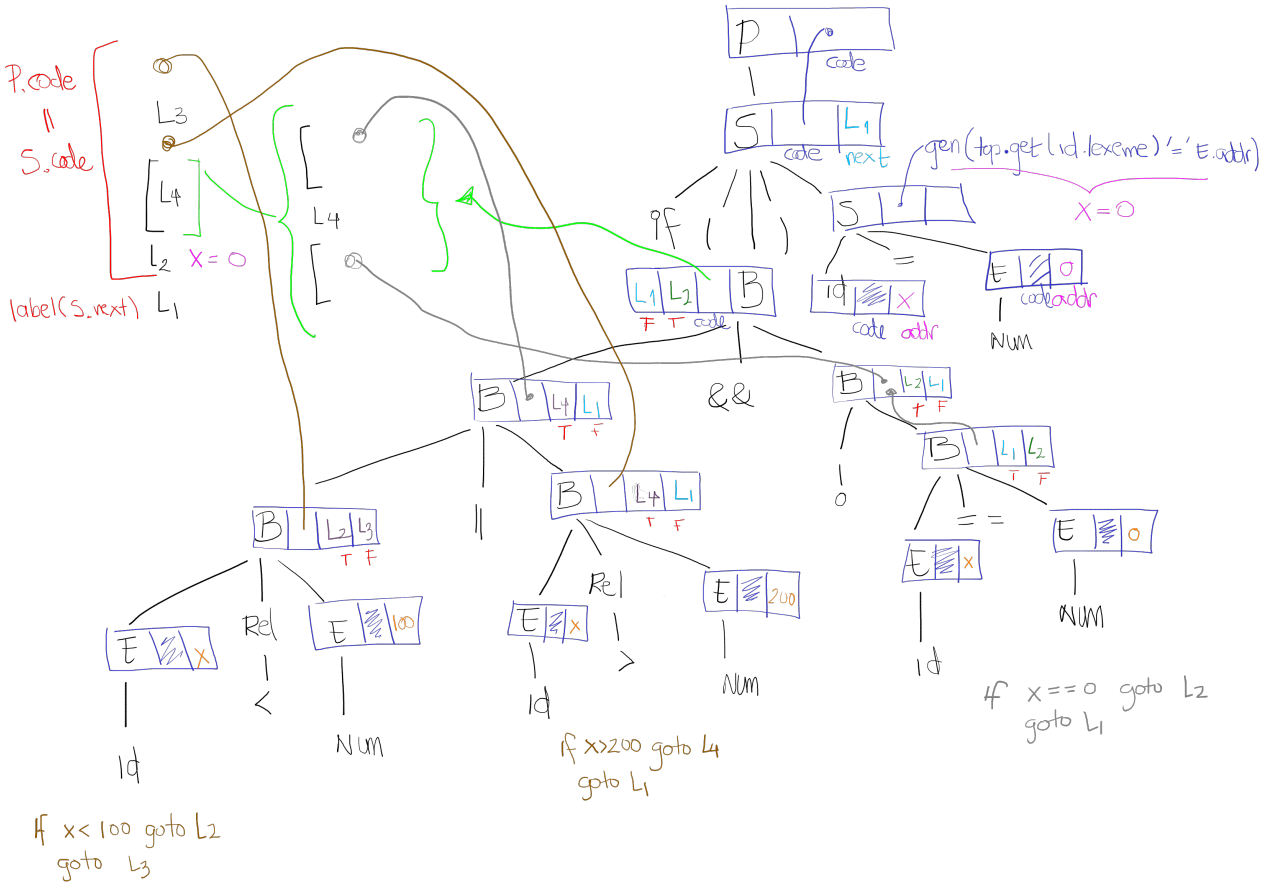
PRODUCTION	SEMANTIC RULES
$S \rightarrow$ **id** `=` E `;`	$S.code = E.code \,\|\|$ $gen(top.get($**id**$.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1$ `+` E_2	$E.addr =$ **new** $Temp\,()$ $E.code = E_1.code \,\|\|\, E_2.code \,\|\|$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
| `-` E_1	$E.addr =$ **new** $Temp\,()$ $E.code = E_1.code \,\|\|$ $gen(E.addr\ '='\ '$**minus**$'\ E_1.addr)$
| `(` E_1 `)`	$E.addr = E_1.addr$ $E.code = E_1.code$
| **id**	$E.addr = top.get($**id**$.lexeme)$ $E.code = ''$

[1] Figuras y ejemplo tomado del libro "Compilers, Principles, Techniques and Tools", Aho A. et al., Capítulo 6.

Código fuente y código de tres direcciones usando las funciones semánticas:

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

```
    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:
```



Gráficas de dependencias de atributos

Representación intermedia que modela el flujo de la información entre las instancias de los atributos en un parse-tree. Es una gráfica (alternativa) donde los nodos son los diferentes atributos asociados a cada símbolo y las aristas dirigidas representan las restricciones de las reglas semánticas en el cálculo de los atributos.

Ejemplo 4. Considera la siguiente gramática para representar números binarios con signo

$$num \rightarrow sign\ list \qquad sign \rightarrow + \mid - \qquad list \rightarrow bit \mid list\ bit \qquad bit \rightarrow 0 \mid 1$$

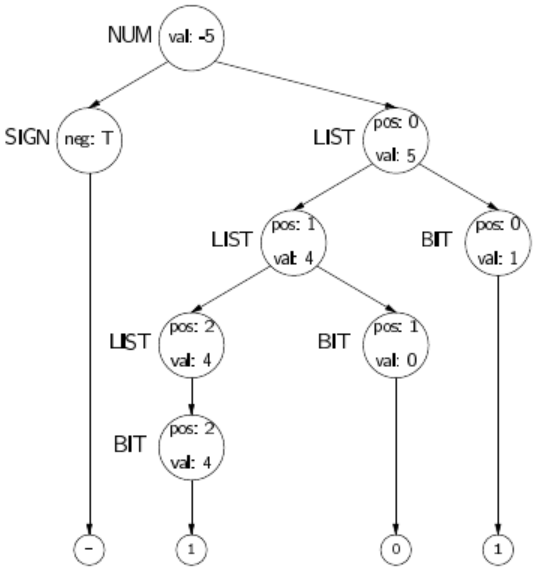
La gramática extendida con reglas semánticas que obtienen el valor decimal que representa el número binario es la siguiente:

PRODUCTION	SEMANTIC RULES

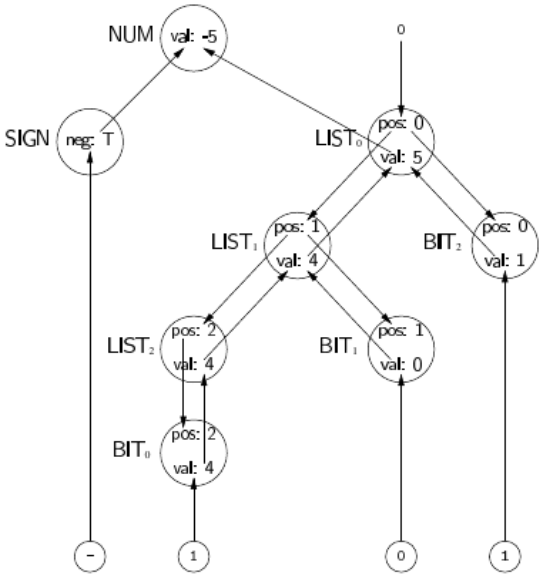
NUM → SIGN LIST	LIST.pos := 0
	if SIGN.neg
	  NUM.val := -LIST.val
	else
	  NUM.val := LIST.val

SIGN → +	SIGN.neg := false
SIGN → -	SIGN.neg := true
LIST → BIT	BIT.pos := LIST.pos LIST.val := BIT.val
LIST → $LIST_1$ BIT	$LIST_1$.pos := LIST.pos + 1 BIT.pos := LIST.pos LIST.val := $LIST_1$.val + BIT.val
BIT → 0	BIT.val := 0
BIT → 1	BIT.val := $2^{\mathrm{BIT}.pos}$

El siguiente árbol se obtiene al procesar la cadena -101 usando la gramática anterior:



Después de obtener el parse-tree, se pueden (re)visitar los nodos para calcular los atributos:



El primer atributo en calcularse es el signo seguido del atributo `pos` en los nodos LIST_0, LIST_1 y LIST_2 en ese orden. Con estos atributo se puede calcular el atributo `pos` para los nodos BIT_0, BIT_1 y BIT_2. Después se calcula el valor de los bits y de las listas en el siguiente orden: BIT_0, LIST_2, BIT_1, LIST_1, BIT_2, LIST_0. Este orden está definido por la regla semántica $\mathsf{LIST.val}{:=}\mathsf{LIST}_2\mathsf{.val}{+}\mathsf{BIT.val}$. Finalmente se calcula el valor decimal en el nodo raíz.

Gráficas de control de flujo

Representación que modela la transferencia de control en el programa. Es una gráfica donde los nodos son bloques básicos y las aristas dirigidas representan el flujo del control de las estructuras de control (ciclos, casos, saltos, etc.).

Esta es una representación que respeta el significado original del código fuente, pero que será fácil de traducir al lenguaje máquina.

Los nodos de la gráfica de flujo son nodos básicos. Hay un arista del bloque B al bloque C si y sólo si es posible que la primera instrucción del bloque C se siga a partir de la última instruccion del bloque B. Lo que significa que existen dos formas en que un arista puede ser justificado:

1. Existe un salto condicional o incondicional desde la última instrucción de B hacia la primera de C.
2. La primera instrucción de C se sigue inmediamente a la última de B en el orden original del código de tres direcciones y B no termina en un salto incondicional.

Referencias

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques and Tools*. Pearson Education Inc., Second edition, 2007.

[2] Torben Ægidius Mogensen. *Basics of Compiler Design*. Lulu Press, 2010.

[3] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: An Appetizer (Undergraduate Topics in Computer Science)*. Springer-Verlag, Berlin, Heidelberg, 2007.

[4] Frank Pfenning. Notas del curso (15-411) Compiler Design. `https://www.cs.cmu.edu/~fp/courses/15411-f14/`, 2014.

[5] Michael Lee Scott. *Programming Language Pragmatics*. Morgan-Kauffman Publishers, Third edition, 2009.

[6] Yunlin Su and Song Y. Yan. *Principles of Compilers, A New Approach to Compilers Including the Algebraic Method*. Springer-Verlag, Berlin Heidelberg, 2011.

[7] Linda Torczon and Keith Cooper. *Engineering A Compiler*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2011.