Representaciones Intermedias
Compiladores 2020-1, Nota de clase 10

Lourdes Del Carmen González Huesca

13 de noviembre de 2019
Facultad de Ciencias UNAM

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 al código objeto.

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.

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 operaicones.

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

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

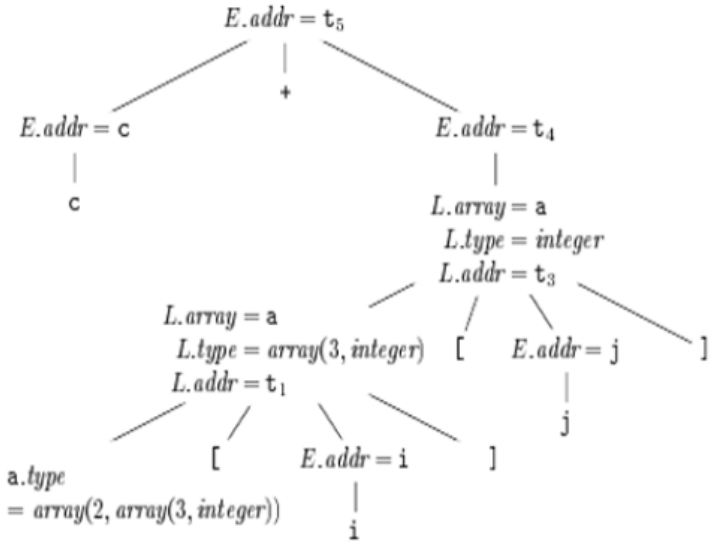
Y la siguiente traducción obtiene una representación intermedia de tres direcciones en un arreglo de referencias. El arreglo se define mediante la gramática

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

y 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:

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

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:

$$
\begin{aligned}
\mathtt{t}_1 &= \mathtt{i} * 12 \\
\mathtt{t}_2 &= \mathtt{j} * 4 \\
\mathtt{t}_3 &= \mathtt{t}_1 + \mathtt{t}_2 \\
\mathtt{t}_4 &= \mathtt{a}[\mathtt{t}_3] \\
\mathtt{t}_5 &= \mathtt{c} + \mathtt{t}_4
\end{aligned}
$$

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, 2014. `https://www.cs.cmu.edu/~fp/courses/15411-f14/`.

[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.