Compiladores 24-2
Representaciones Intermedias

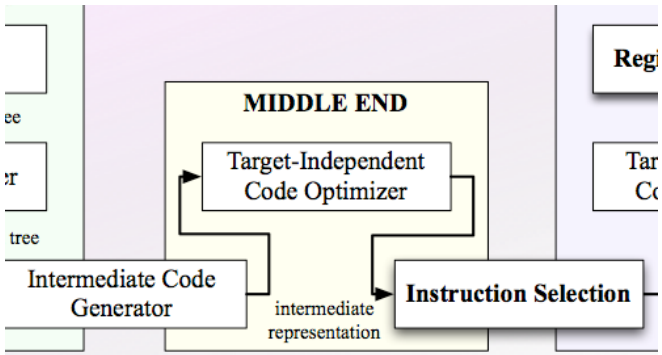
Lourdes del Carmen González Huesca
luglzhuesca@ciencias.unam.mx

Facultad de Ciencias, UNAM

22 abril 2024

Middle-end

La transición entre el front-end y el back-end es posible gracias a una representación intermedia.



- **Forma intermedia**
  representación o representaciones obtenidas del front-end:
  - árbol de sintaxis abstracta
  - árbol de sintaxis concreta
  - gráficas de control de flujo
  - gráficas de dependencias
  - código de 3 direcciones

Middle-end y representaciones intermedias

- ✓ Después de las fases de análisis en el front-end, el compilador tiene suficiente información del programa fuente para realizar una traducción que facilite el manejo del programa en las fases de back-end.

Middle-end y representaciones intermedias

- ✓ Después de las fases de análisis en el front-end, el compilador tiene 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*):

Middle-end y representaciones intermedias

- ✓ Después de las fases de análisis en el front-end, el compilador tiene 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*):
  - fiel al código fuente y que se traduzca al código objeto

Middle-end y representaciones intermedias

- ✓ Después de las fases de análisis en el front-end, el compilador tiene 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*):
  - fiel al código fuente y que se traduzca al código objeto
  - es una estructura de datos que sólo existe en tiempo de compilación

Middle-end y representaciones intermedias

- ✓ Después de las fases de análisis en el front-end, el compilador tiene 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*):
  - fiel al código fuente y que se traduzca al código objeto
  - 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 y realizar optimizaciones

Representaciones intermedias

objetivos y propiedades

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.

Representaciones intermedias

objetivos y propiedades

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

cód. fuente –> secuencia tokens –> parse tree –> rep. intermedia –> ... –> cód. bajo nivel

Representaciones intermedias

tipos

Representaciones de alto nivel que son legibles al ser humano, programador o programadora.

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 (estilo estructurado o secuencial).

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

Código de tres direcciones

- Representación intermedia lineal o lenguaje intermedio que divide expresiones en instrucciones más sencillas las cuales son semejantes en forma a las instrucciones de un procesador.

Código de tres direcciones

- Representación intermedia lineal o lenguaje intermedio que divide expresiones en instrucciones más sencillas las cuales son semejantes en forma a las instrucciones de un procesador.
- Cada instrucción tiene a lo más 4 items: tres operandos y un operador.
- Las instrucciones incluyen asignaciones y operaciones básicas.

Código de tres direcciones

- Representación intermedia lineal o lenguaje intermedio que divide expresiones en instrucciones más sencillas las cuales son semejantes en forma a las instrucciones de un procesador.
- Cada instrucción tiene a lo más 4 items: tres operandos y un operador.
- Las instrucciones incluyen asignaciones y operaciones básicas.
- Utiliza direcciones relativas que serán más fáciles de traducir a direcciones físicas en la asignación de registros: siguen el mismo orden lineal pero puede haber saltos (`goto`), bifurcaciones, condicionales, llamadas a funciones o subrutinas, accesos a memoria y asignaciones, etc.

Código de tres direcciones

- Representación intermedia lineal o lenguaje intermedio que divide expresiones en instrucciones más sencillas las cuales son semejantes en forma a las instrucciones de un procesador.
- Cada instrucción tiene a lo más 4 items: tres operandos y un operador.
- Las instrucciones incluyen asignaciones y operaciones básicas.
- Utiliza direcciones relativas que serán más fáciles de traducir a direcciones físicas en la asignación de registros: siguen el mismo orden lineal pero puede haber saltos (`goto`), bifurcaciones, condicionales, llamadas a funciones o subrutinas, accesos a memoria y asignaciones, etc.

```
if (x + y*z > x*y +z)
    a = 0;
```

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

Código de tres direcciones

ejemplo 1

Gramática extendida (ie. definición dirigida por la sintaxis) para construir código de tres direcciones de expresiones que representan asignaciones usando expresiones aritméticas.

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

Los atributos a usar son `code` para el código y `addr` para guardar la dirección que almacenará el valor de una expresión.

Código de tres direcciones

ejemplo 2

Gramática para representar arreglos de referencias incluyendo $L \rightarrow L\,[E] \mid \mathbf{id}\,[E]$ y con funciones semánticas para obtener una representación intermedia de tres direcciones:

- `addr` atributo para almacenar temporalmente el desplazo en el arreglo
- `array` es el apuntador a la tabla de símbolos donde se almancena el arreglo
- `type` es el tipo del subarreglo correspondiente.

Código de tres direcciones

ejemplo 2

Gramática para representar arreglos de referencias incluyendo $L \rightarrow L\,[\,E\,] \mid \mathbf{id}\,[\,E\,]$ y con funciones semánticas para obtener una representación intermedia de tres direcciones:

- `addr` atributo para almacenar temporalmente el desplazo en el arreglo
- `array` es el apuntador a la tabla de símbolos donde se almancena el arreglo
- `type` es el tipo del subarreglo correspondiente.

$S \rightarrow \mathbf{id} = E\ ;$ { $gen(top.get(\mathbf{id}.lexeme)\ '='\ E.addr);$ }

$\mid\ L = E\ ;$ { $gen(L.array.base\ '['\ L.addr\ ']'\ '='\ E.addr);$ }

$E \rightarrow E_1 + E_2$ { $E.addr = \mathbf{new}\ Temp();$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr);$ }

$\mid\ \mathbf{id}$ { $E.addr = top.get(\mathbf{id}.lexeme);$ }

$\mid\ L$ { $E.addr = \mathbf{new}\ Temp();$ $gen(E.addr\ '='\ L.array.base\ '['\ L.addr\ ']');$ }

$L \rightarrow \mathbf{id}\ [\ E\ ]$ { $L.array = top.get(\mathbf{id}.lexeme);$ $L.type = L.array.type.elem;$ $L.addr = \mathbf{new}\ Temp();$ $gen(L.addr\ '='\ E.addr\ '*'\ L.type.width);$ }

$L \rightarrow L_1\ [\ E\ ]$ { $L.array = L_1.array;$ $L.type = L_1.type.elem;$ $t = \mathbf{new}\ Temp();$ $L.addr = \mathbf{new}\ Temp();$ $gen(t\ '='\ E.addr\ '*'\ L.type.width);$ $gen(L.addr\ '='\ L_1.addr\ '+'\ t);$ }

Código de tres direcciones

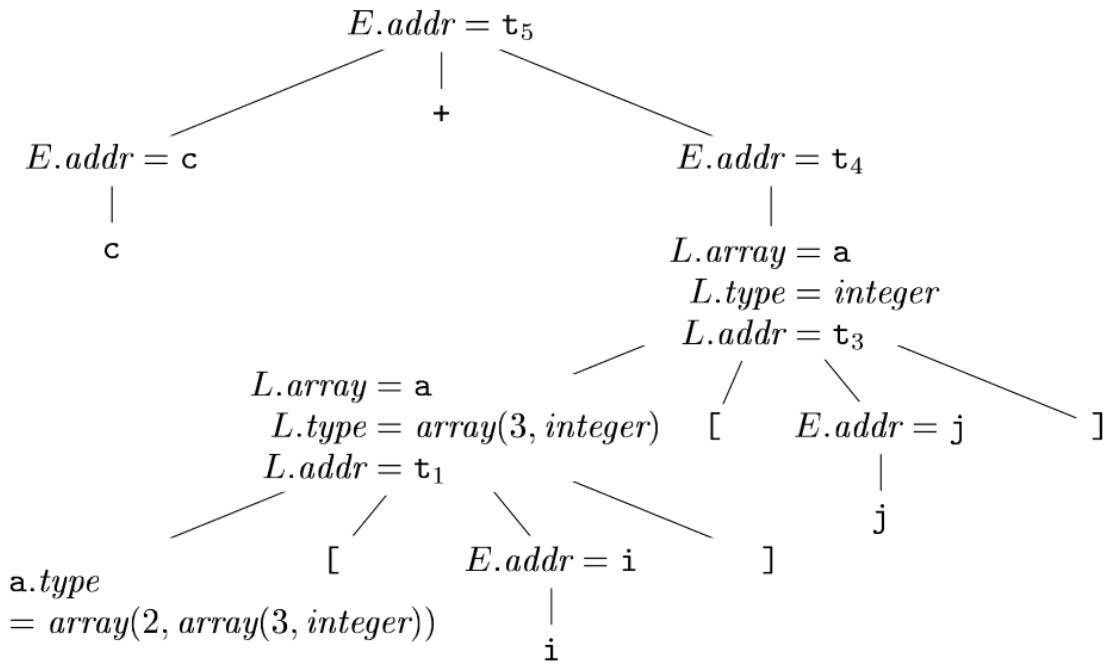
ejemplo 2

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:

Código de tres direcciones

ejemplo 2

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:



Gráficas de dependencias de atributos

- Representación intermedia que modela el flujo de la información entre las instancias de los atributos en un parse-tree.
- Una gráfica 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: números binarios con signo

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

Gráficas de dependencias de atributos

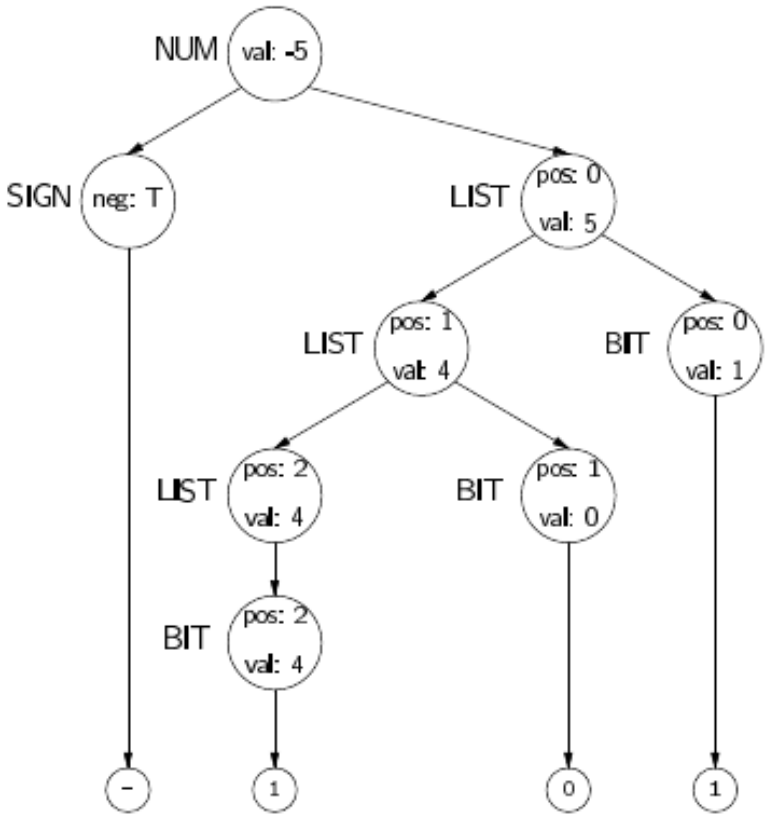
ejemplo números binarios con signo

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

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 := $2^{BIT.pos}$

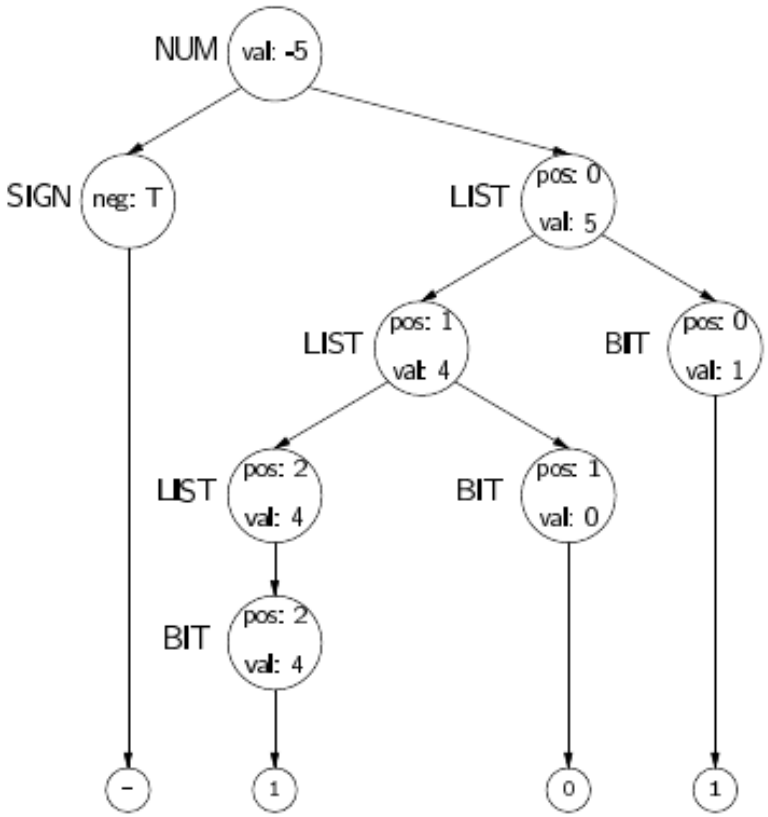
Gráficas de dependencias de atributos

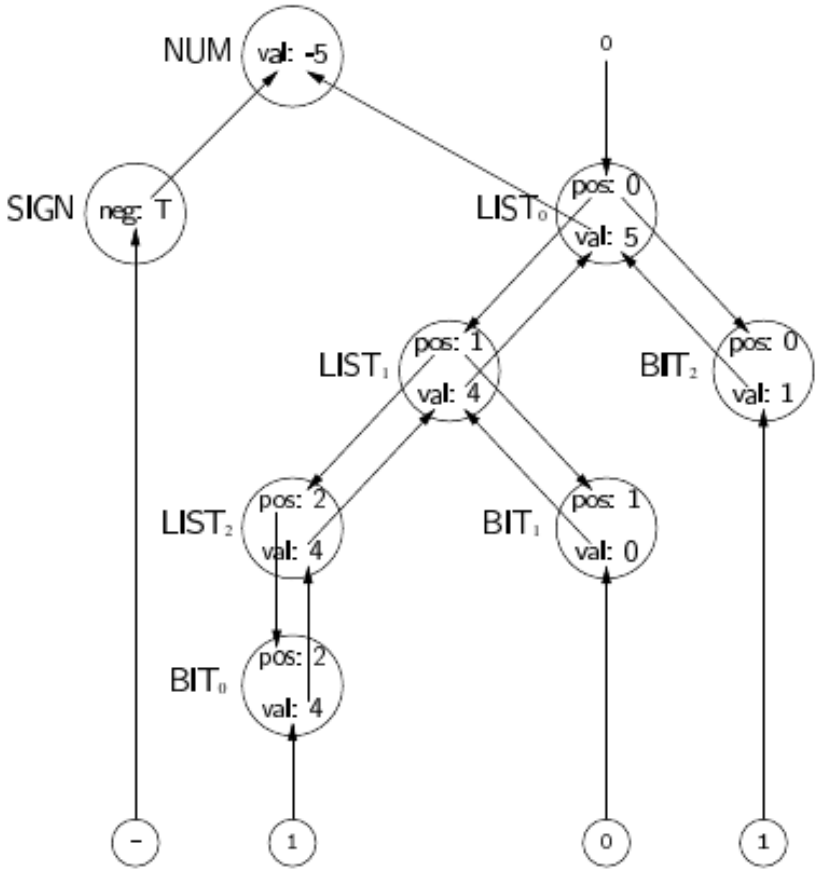
ejemplo números binarios con signo



Gráficas de dependencias de atributos

ejemplo números binarios con signo





Gráficas de control de flujo

- Representación intermedia que modela la transferencia de control en el programa.
- 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.).

Gráficas de control de flujo

- Representación intermedia que modela la transferencia de control en el programa.
- 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.).
- Representación que respeta el significado original del código fuente pero que será fácil de traducir a lenguaje máquina.

```
(1)    i = m-1
(2)    j = n
(3)    t1 = 4*n
(4)    v = a[t1]
(5)    i = i+1
(6)    t2 = 4*i
(7)    t3 = a[t2]
(8)    if t3<v goto (5)
(9)    j = j-1
(10)   t4 = 4*j
(11)   t5 = a[t4]
(12)   if t5>v goto (9)
(13)   if i>=j goto (23)
(14)   t6 = 4*i
(15)   x = a[t6]
(16)   t7 = 4*i
(17)   t8 = 4*j
(18)   t9 = a[t8]
(19)   a[t7] = t9
(20)   t10 = 4*j
(21)   a[t10] = x
(22)   goto (5)
(23)   t11 = 4*i
(24)   x  = a[t11]
(25)   t12 = 4*i
(26)   t13 = 4*n
(27)   t14 = a[t13]
(28)   a[t12] = t14
(29)   t15 = 4*n
(30)   a[t15] = x
```

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.

[8] Steve Zdancewic.
Notas del curso (CIS 341) - Compilers, Universidad de Pennsylvania, Estados Unidos.
https://www.cis.upenn.edu/~cis341/current/, 2018.