Parseo y Generación de Código

2 de noviembre de 2017

Traducción dirigida por la sintaxis
Generación de código intermedio

Licenciatura en Informática con Orientación en Desarrollo de Software
Universidad Nacional de Quilmes

Traducción dirigida por la sintaxis

Traducción dirigida por la sintaxis

En muchos casos es posible aplicar una transformación sobre el lenguaje fuente a medida que se hace el análisis sintáctico, sin construir una representación intermedia como un AST.

Para esto se asocia un conjunto de atributos a cada nodo del árbol de derivación, y se calcular esos atributos a medida que el analizador sintáctico emite producciones, sin construir explícitamente un árbol.

Se distingue entre dos tipos de atributos: **sintetizados** y **heredados**:

- Los atributos **sintetizados** para un nodo del árbol se calculan a partir de los atributos de sus hijos.
- Los atributos **heredados** para un nodo del árbol se calculan a partir de los atributos de su padre y hermanos.

Traducción dirigida por la sintaxis

Una **gramática de atributos** es una gramática libre de contexto (N, Σ, P, S) en la que cada producción está anotada con una **acción semántica**.

Para una producción:

$$A \to X_1 \ldots X_n$$

la acción semántica es un conjunto de ecuaciones entre los atributos de los nodos, que indican:

- De qué manera los atributos sintetizados del nodo A se calculan a partir de los atributos de los nodos X_1, ..., X_n.
- De qué manera los atributos heredados del nodo X_i se calculan a partir de los atributos del nodo A y de los hermanos de X_i.

Además de indicar cómo se propagan los valores de los atributos, las acciones semánticas eventualmente pueden incluir código arbitrario en el lenguaje de programación en el que se implemente el *parser*, e incorporar efectos (p.ej. entrada/salida, reporte de errores).

Traducción dirigida por la sintaxis

- Los **atributos sintetizados** se pueden implementar fácilmente si se utiliza una técnica de **análisis sintáctico ascendente**, en la que el árbol de derivación se construye desde las hojas hacia arriba.
- Los **atributos heredados** se pueden implementar fácilmente si se utiliza una técnica de **análisis sintáctico descendente**, en la que el árbol de derivación se construye desde la raíz hacia abajo. Generalmente, para que esto sea posible, se introduce alguna restricción sobre las dependencias de los atributos de los hermanos (típicamente los atributos heredados de X_i pueden depender de los atributos de los hermanos anteriores, $X_1, \ldots, X_{i-1}$ pero no de los siguientes).

Traducción dirigida por la sintaxis

Ejercicio. Agregar dos atributos sintetizados, *valor* y *costo* a la siguiente gramática $(\{E, T, F\}, \{\mathbf{n}, +, *, (,)\}, P, E)$ para calcular respectivamente el valor de la expresión aritmética, y la cantidad de operaciones que se necesitan para computarla:

$$
\begin{array}{lcl}
E & \to & T \\
 & | & T + E \\
T & \to & F \\
 & | & F * T \\
F & \to & \mathbf{n} \\
 & | & (E)
\end{array}
$$

Traducción dirigida por la sintaxis

Ejercicio. Dada la siguiente gramática $(\{S, S', S\}, \{0, 1\}, P, S')$:

$$\begin{array}{lcl} S' & \to & S \\ S & \to & SB \\ & | & \epsilon \\ B & \to & 0 \\ & | & 1 \end{array}$$

Agregar:

- un atributo heredado *peso* al símbolo S,
- un atributo sintetizado *valor* a los símbolos S', S, B,

para calcular el valor que representa la cadena interpretada en binario.

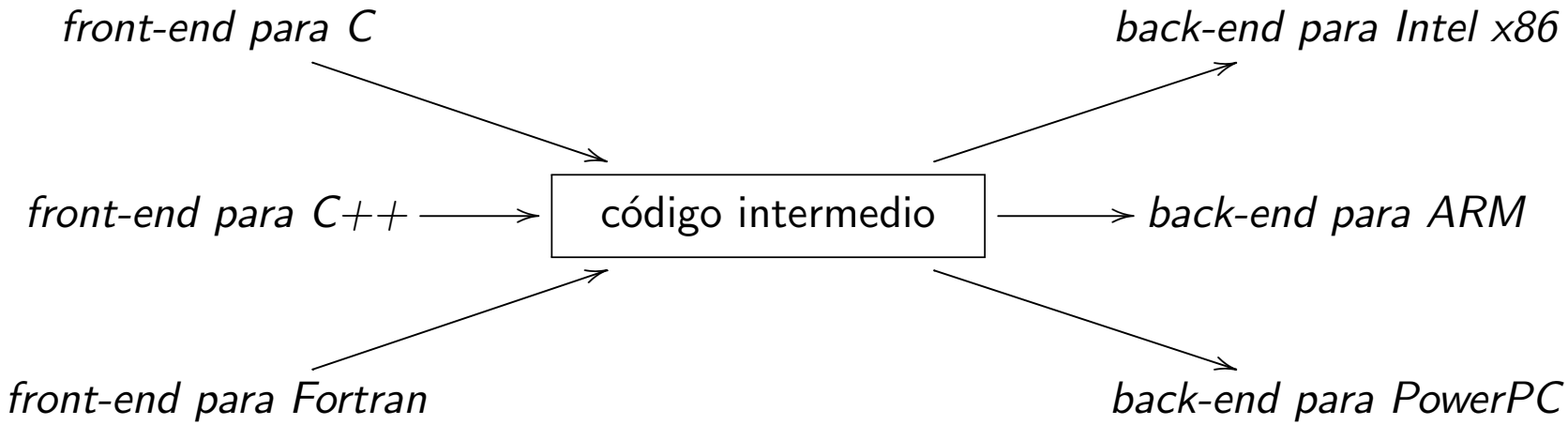
Código intermedio

Objetivo del código intermedio

Idealmente un compilador se estructura en dos etapas:

- **Front-end:** código fuente $\rightarrow$ código intermedio.
- **Back-end:** código intermedio $\rightarrow$ código objeto.

Por ejemplo gcc cuenta con varios *front-ends* y varios *back-ends*:



Objetivo del código intermedio

Una **representación intermedia** es una representación del programa en un modelo de ejecución *abstracto*[1].

Propiedades deseables:

- Debe abstraer detalles de la arquitectura real:
  - Modos de direccionamiento.
  - Cantidad de registros disponibles.
  - Instrucciones disponibles.
- Debe ser apta para etapas posteriores del compilador:
  - Análisis semántico, tipado.
  - Análisis estático.
  - Optimización.
  - Generación de código nativo para una arquitectura real.

[1] Es decir, no necesariamente corresponde a la arquitectura de una máquina física.

Objetivo del código intermedio

Algunas representaciones intermedias:

- Árbol de sintaxis abstracta (AST).
- *Grafos* de sintaxis.
- Máquinas de pila.
- Código de tres direcciones.
- *Continuation-passing style* (usado para lenguajes funcionales).
- ...

Máquinas de pila

Máquinas de pila

Las **máquinas de pila** son una representación intermedia muy común. Se caracterizan por:

- Contar con una pila de valores.
- Contar con instrucciones que en su mayoría operan:
  - sacando los operandos del tope de la pila,
  - metiendo los resultados en el tope de la pila.

Por ejemplo, la Java Virtual Machine y el *bytecode* de Python son máquinas de pila.

Máquinas de pila

Por ejemplo si la arquitectura cuenta con estas operaciones:

```
data Op =
    OpPushInt Int -- Mete una constante.
  | OpAdd         -- Suma.
  | OpMul         -- Multiplicación.
```

Con la siguiente semántica:

```
run :: [Op] -> [Int] -> [Int]
run []                    s           = s
run (OpPushInt n : ops) s             = run ops (n : s)
run (OpAdd : ops)         (y : x : s) = run ops (x + y : s)
run (OpMul : ops)         (y : x : s) = run ops (x * y : s)
```

Máquinas de pila

El siguiente lenguaje de expresiones:

```
data Expr =
    ExprInt Int
  | ExprAdd Expr Expr
  | ExprMul Expr Expr
```

Con la siguiente semántica:

```
eval :: Expr -> Int
eval (ExprInt n)      = n
eval (ExprAdd e1 e2) = eval e1 + eval e2
eval (ExprMul e1 e2) = eval e1 * eval e2
```

Se puede compilar de manera inmediata, haciendo un recorrido *post-order* sobre el AST:

```
compile :: Expr -> [Op]
```

Ejercicio. Definir la función `compile`.
Ejercicio. Compilar la expresión $2 + 8 * 5$.

Máquinas de pila

Si el compilador está bien definido, se puede demostrar que en efecto es correcto:

Observación. Vale la siguiente propiedad:

```
run ops2 (run ops1 stack) = run (ops1 ++ ops2) stack
```

Se puede probar por inducción en la estructura de `ops1`.

Ejercicio. Demostrar:

```
eval expr : stack = run (compile expr) stack
```

por inducción en la estructura de `expr`.

Máquinas de pila

Compilación del `if`.

```
if (condición) {
    bloqueThen
} else {
    bloqueElse
}
```

```
<código para evaluar condición>
OpJumpIfFalse .label_else
<código para evaluar bloqueThen>
OpJump .label_end
.label_else:
<código para evaluar bloqueElse>
.label_end:
```

Máquinas de pila

Compilación del `while`.

`while (condición) {`
`  bloque`
`}`
	`.label_start:`
	`<código para evaluar condición>`
	`OpJumpIfFalse .label_end`
	`<código para evaluar bloque>`
	`OpJump .label_start`
	`.label_end:`

`break;`	`OpJump .label_end`
`continue;`	`OpJump .label_start`

Máquinas de pila

Compilación de funciones.

Hay muchos mecanismos para compilar el pasaje de parámetros y la devolución de resultados. Estos mecanismos se conocen como **convenciones de llamada**.

- Los parámetros se pueden pasar en distintas ubicaciones:
  - pila
  - registros
  - *heap*
- Los parámetros se pueden pasar de distintas maneras:
  - por valor
  - por referencia
  - otras (p.ej. por copia y restauración)
- Casi siempre se usan mecanismos híbridos.

Máquinas de pila

Típicamente cada invocación a una función cuenta con un **registro de activación** (*activation record* o *stack frame*).

El registro de activación incluye:

- Valores de los parámetros.
- Dirección de retorno.
- Valores de las variables locales.
- Puntero al registro de activación de la función invocadora.

Máquinas de pila

Compilación de un llamado a una función.

Usando pasaje de parámetros **en el heap**, **por valor**, una invocación a una función se podría compilar así:

$f(expr_1, \ldots, expr_n)$	`<código para evaluar` $expr_1$`>`
	`...`
	`<código para evaluar` $expr_n$`>`
	`OpMakeActivationRecord` n
	`OpJump .label_start_f`

La instrucción `OpMakeActivationRecord` n tiene el siguiente efecto:

- reserva espacio en el *heap* para un registro de activación con n parámetros,
- guarda el puntero al viejo registro de activación,
- saca los n valores del tope de la pila y los guarda en el nuevo registro de activación,
- guarda el *instruction pointer* actual en el nuevo registro de activación.

Máquinas de pila

Compilación de una definición de función.

Con la convención anterior, la definición de una función se podría compilar así:

`fun f(`x_1`, ..., `x_n`) {`
`  cuerpo`
`  return expr;`
`}`
	`.label_start_f:`
	`<código para evaluar cuerpo>`
	`<código para evaluar expr>`
	`OpReturn`

La instrucción `OpReturn` tiene el siguiente efecto:

- libera el espacio del registro de activación local,
- recupera el puntero al registro de activación del llamador,
- salta al *instruction pointer* guardado en el registro de activación local.

Máquinas de pila

Compilación de una definición de función.

Cada vez que se utiliza el parámetro x_i, se compila así:

`OpPushParameter` *i*

El efecto de la instrucción `OpPushParameter` *i* es tomar el valor del *i*-ésimo parámetro almacenado en el registro de activación local y meterlo en la pila.

Máquinas de pila

Ejercicio. Compilar el siguiente código con la convención de llamadas descripta anteriormente:

```
fun g(n, m) {
  return f(2 * n + m)
}

fun f(x) {
  return x + x
}

fun main() {
  g(10, 1)
}
```

Y hacer el seguimiento de la ejecución de la función `main`.

Máquinas de pila

Compilación de vectores.

`[expr`$_1$`, ..., expr`$_n$`]`	`<código para evaluar expr`$_1$`>`
	`...`
	`<código para evaluar expr`$_n$`>`
	`OpVecMake` n

`expr`$_1$`[expr`$_2$`]`	`<código para evaluar expr`$_1$`>`
	`<código para evaluar expr`$_2$`>`
	`OpVecDeref`

`expr`$_1$`[expr`$_2$`] := expr`$_3$	`<código para evaluar expr`$_1$`>`
	`<código para evaluar expr`$_2$`>`
	`<código para evaluar expr`$_3$`>`
	`OpVecAssign`

Máquinas de pila

Compilación de clases y objetos.

Para compilar un lenguaje orientado a objetos basado en clases y con sistema de tipos estático[2] se suelen usar **vtables** o tablas de métodos virtuales.

Veámoslo con un ejemplo:

```
class A {
    int x;
    void f(int x) { ... };
}
```

```
class B extends A {
    int y;
    void g() { ... };
}
```

`new A(x := 10)`	`OpPushInt 10`
	`OpPushAddress .label_start_f`
	`OpVecMake 2`

`new B(x := 10, y := 20)`	`OpPushInt 10`
	`OpPushAddress .label_start_f`
	`OpPushInt 20`
	`OpPushAddress .label_start_g`
	`OpVecMake 4`

[2] Más precisamente, lenguajes con *early binding* como C++ y Java.

Máquinas de pila

Compilación de clases y objetos.

`obj.f(42);`	`<código para evaluar obj>`
	`OpDup  -- duplica el tope de la pila`
	`OpPushInt 42`
	`OpMakeActivationRecord 2`
	`OpPushInt 2  -- offset de f`
	`OpVecDeref`
	`OpJumpDynamic  -- salta a la dirección`
	`  -- en el tope de la pila`

`obj.g();`	`<código para evaluar obj>`
	`OpDup  -- duplica el tope de la pila`
	`OpMakeActivationRecord 1`
	`OpPushInt 4  -- offset de g`
	`OpVecDeref`
	`OpJumpDynamic  -- salta a la dirección`
	`  -- en el tope de la pila`

Máquinas de pila

Observar que:

- `obj.f(...)` funciona para instancias de ambas clases.
- "`self`" se pasa siempre como primer parámetro a todos los métodos.
- Los métodos se compilan como funciones comunes y corrientes, con `self` como parámetro extra.

Adicionalmente:

- Cada objeto podría contar con un campo más que represente un *tag* indicando cuál es su clase.
- P.ej.: 100 para las instancias de la clase A, 101 para las instancias de la clase B, etc.

Máquinas de pila

Compilación de excepciones.

Para compilar excepciones se dispone de una **pila global** en la que se guardan punteros a las posiciones del código donde hay rutinas de manejo de errores esperando que potencialmente se levante una excepción.

`try {`
`  cuerpo`
`} catch (E e) {`
`  manejador`
`}`
	`OpPushHandler .label_error`
	`<código para cuerpo>`
	`OpJump .label_end`
	`.label_error:`
	`OpRethrowIfNotInstanceOf E`
	`<código para manejador>`
	`.label_end:`

`throw exception;`	`<código para exception>` `OpThrow`

La operación `OpThrow` saca el primer puntero de la pila de *handlers* y salta a él. Requiere cierto cuidado con el manejo de registros de activación.

Código de tres direcciones

Código de tres direcciones

El **código de tres direcciones** es una representación intermedia que se basa en instrucciones que en su mayoría operan con variables temporales que representan registros o direcciones de memoria.

Código de tres direcciones

Por ejemplo si la arquitectura permite referenciar una cantidad en principio ilimitada de registros t_1, t_2, ..., t_n, ..., y las siguientes operaciones:

```
type Reg = Int

data Op =
    OpMovInt Reg Int    -- t_i := n
  | OpAdd Reg Reg Reg   -- t_i := t_j + t_k
  | OpMul Reg Reg Reg   -- t_i := t_j * t_k
```

Con la siguiente semántica:

```
type Memory = Map Reg Int

run :: [Op] -> Memory -> Memory
run [] m = m
run (OpMovInt i n : ops) m = run ops (insert i n m)
run (OpAdd i j k : ops) m =
  run ops (insert i (lookup j m + lookup k m) m)
run (OpMul i j k : ops) m =
  run ops (insert i (lookup j m * lookup k m) m)
```

Código de tres direcciones

El lenguaje de expresiones:

```
data Expr =
    ExprInt Int
  | ExprAdd Expr Expr
  | ExprMul Expr Expr
```

Se puede compilar con una función:

```
compile :: Reg -> Expr -> [Op]
```

Respetando la siguiente convención:

- El resultado de (`compile` i `expr`) se ubica en el registro t_i.
- El código que se obtiene a partir de (`compile` i `expr`) asume que todos los registros t_j con $j \geq i$ se encuentran disponibles para usar.
- El código que se obtiene a partir de (`compile` i `expr`) nunca modifica los registros t_j con $j < i$.

Ejercicio. Definir la función `compile`.

Ejercicio. Compilar la expresión $2 + 8 * 5$.

Código de tres direcciones

- La mayor parte de las construcciones (`ifs`, `whiles`, vectores, etc.) se compilan de manera análoga al caso de las máquinas de pila.
- Diferencia importante: definiciones y llamados a **funciones**.

Ejemplo. ¿Qué problema surge al compilar este programa?

```
fun factorial(n) {
  if (n == 0) {
    return 1
  } else {
    return n * factorial(n - 1)
  }
}
```

Código de tres direcciones

Solución típica para este problema:

- Cada función puede usar todos los registros $t_1, \ldots, t_n, \ldots$.
- Se dispone de una pila en la que se guardan los valores de los registros para poder sobreescribirlos y restaurar sus antiguos valores después de haberlos usado.
- Dos variantes:
  - *Caller saves:* antes de invocar una función, se guardan todos los registros que estaban siendo utilizados por la función invocadora.
  - *Callee saves:* al inicio de la definición de cada función, se guardan todos los registros que van a ser utilizados en el cuerpo de la función.

Código de tres direcciones

Ejemplo de esquema caller saves. Se modifica el código para invocar una función suponiendo que los registros $t_1, \ldots, t_{i-1}$ están usados y que los valores se retornan en t_1. El código para:

```
compile i (ExprCall func [arg_1, ..., arg_n])
```

podría tener la siguiente estructura:

```
<código para evaluar arg_1 en el registro t_{i+1}>
...
<código para evaluar arg_n en el registro t_{i+n}>
OpPush t_1
...     -- Guardar los registros usados.
OpPush t_{i-1}

OpMakeActivationRecord t_{i+1} t_{i+n}
OpJump .label_<func>_start
OpMov t_i t_1 -- Resultado del return (t_i := t_1).

OpPop t_{i-1}
...     -- Restaurar los registros.
OpPop t_1
```

Código de tres direcciones

Ejemplo de esquema callee saves. Se modifica el código para la definición de una función suponiendo que la función necesita usar los primeros i registros, es decir $t_1, \ldots, t_i$, y que los valores se retornan en un registro t_0. El código para:

```
fun f(x_1, ..., x_n) { cuerpo }
```

podría tener la siguiente estructura:

```
.label_f_start:
OpPush t_1
...     -- Guardar los registros para sobreescribirlos.
OpPush t_i

<código para evaluar cuerpo,
 guardando el valor de retorno en el registro t_0>

OpPop t_i
...     -- Restaurar los registros.
OpPop t_1
OpReturn
```

Números de Ershov

El **número de Ershov** de una expresión e formada a partir de constantes, variables y operadores binarios, se define como la **mínima cantidad de registros** necesaria para calcular el valor de la expresión e usando código de tres direcciones.

Ejercicio. Hay esencialmente dos maneras posibles de computar la expresión $x + y * z$ usando código de tres direcciones. Describirlas y determinar su número de Ershov.

Números de Ershov

Algoritmo para computar una expresión usando la mínima cantidad de registros.

```
compile :: Reg -> Expr -> [Op]
compile i (ExprInt n) = OpMovInt i n    -- t_i := n
compile i (ExprBinop operator e1 e2)
  | ershov1 > ershov2 =
      compile i e1 ++
      compile (i + 1) e2 ++
      [OpBinop operator i i (i + 1)]   -- t_i := t_i ◊ t_{i+1}
  | otherwise =
      compile i e2 ++
      compile (i + 1) e1 ++
      [OpBinop operator i (i + 1) i]   -- t_i := t_{i+1} ◊ t_i
  where
    ershov1 = maxReg (compile 1 e1)
    ershov2 = maxReg (compile 1 e2)

maxReg :: [Op] -> Reg
maxReg ops = ... -- máximo registro referenciado en ops
```

Ojo: el algoritmo dado arriba es exponencial.

Compilación del case

Compilación del case

Consideremos una estructura de control tipo "case" sobre números enteros como la siguiente:

```
switch (n) {
  case x1: bloque1; break;
  case x2: bloque2; break;
  ...
  case xn: bloquen; break;
  default: bloquedefault
}
```

Hay varias maneras de compilar esta estructura.

Compilación del case

Como una cadena de ifs.

```
switch (n) {
  case x1: bloque1; break;
  case x2: bloque1; break;
  ...
  case xn: bloquen; break;
  default: bloquedefault
}
```

```
if (n == x1)
  bloque1
else if (n == x2)
  bloque2
...
else if (n == xn)
  bloquen
else
  bloquedefault
end
```

Compilación del case

Como una tabla de saltos precomputada.

Sean $A = \text{mín}\{x_1, \ldots, x_n\}$ y $B = \text{máx}\{x_1, \ldots, x_n\}$. La estructura se puede compilar así:

```
switch (n) {
  case x1: bloque1; break;
  case x2: bloque1; break;
  ...
  case xn: bloquen; break;
  default: bloquedefault
}
```

```
if (A ≤ n && n ≤ B)
  jump table[n - A]
else
  jump .label_default
end
.label_1: bloque1
          jump .label_end
.label_2: bloque2
          jump .label_end
...
.label_n: bloquen
          jump .label_end
.label_default:
          bloquedefault
.label_end:
```

donde `table` es una tabla que se inicializa de tal modo que:

```
table[xi] = .label_i          para cada i ∈ {x1, ..., xn}
table[y] = .label_default     para y ∈ [A..B] \ {x1, ..., xn}
```

Compilación del case

Como un árbol de decisión.

Se puede armar un árbol binario de búsqueda balanceado de los valores $\{x_1, \ldots, x_n\}$, y generar código con `ifs` anidados.

Por ejemplo, para el conjunto $\{1, 2, 3, 4, 5, 6\}$:

```
if n <= 3
  if n <= 1
      if n == 1 then bloque_1 else bloque_default
  else
      if n == 2 then bloque_2 else bloque_3
  end
else
  if n <= 5
      if n == 4 then bloque_4 else bloque_5
  else
      if n == 6 then bloque_6 else bloque_default
  end
end
```

Compilación del case

Generalmente los compiladores combinan estas técnicas:

- Si el conjunto $\{x_1, \ldots, x_n\}$ es pequeño, se usa una cadena de `ifs` lineal.
- Si el conjunto $\{x_1, \ldots, x_n\}$ es denso, es decir, si el cociente $\frac{n}{B-A}$ es superior a un umbral, se usa una tabla de saltos.
- Si el conjunto $\{x_1, \ldots, x_n\}$ es grande y esparso (es decir, no denso), se usa un árbol de decisión.

Estas técnicas se pueden combinar incluso dentro de una misma estructura `case`.