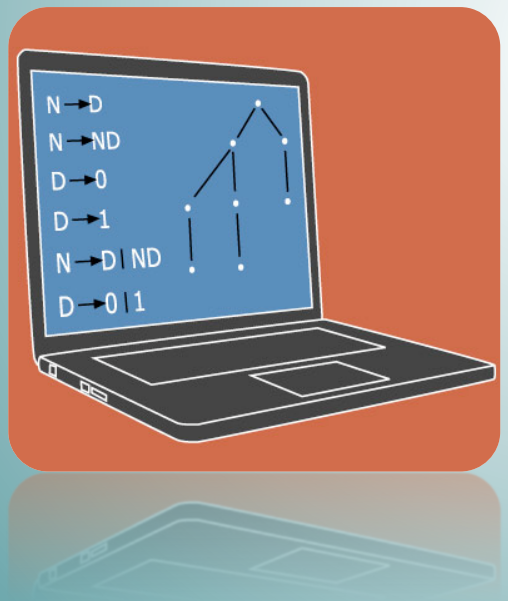
Procesadores de Lenguaje

Generación de código intermedio



Cristina Tirnauca

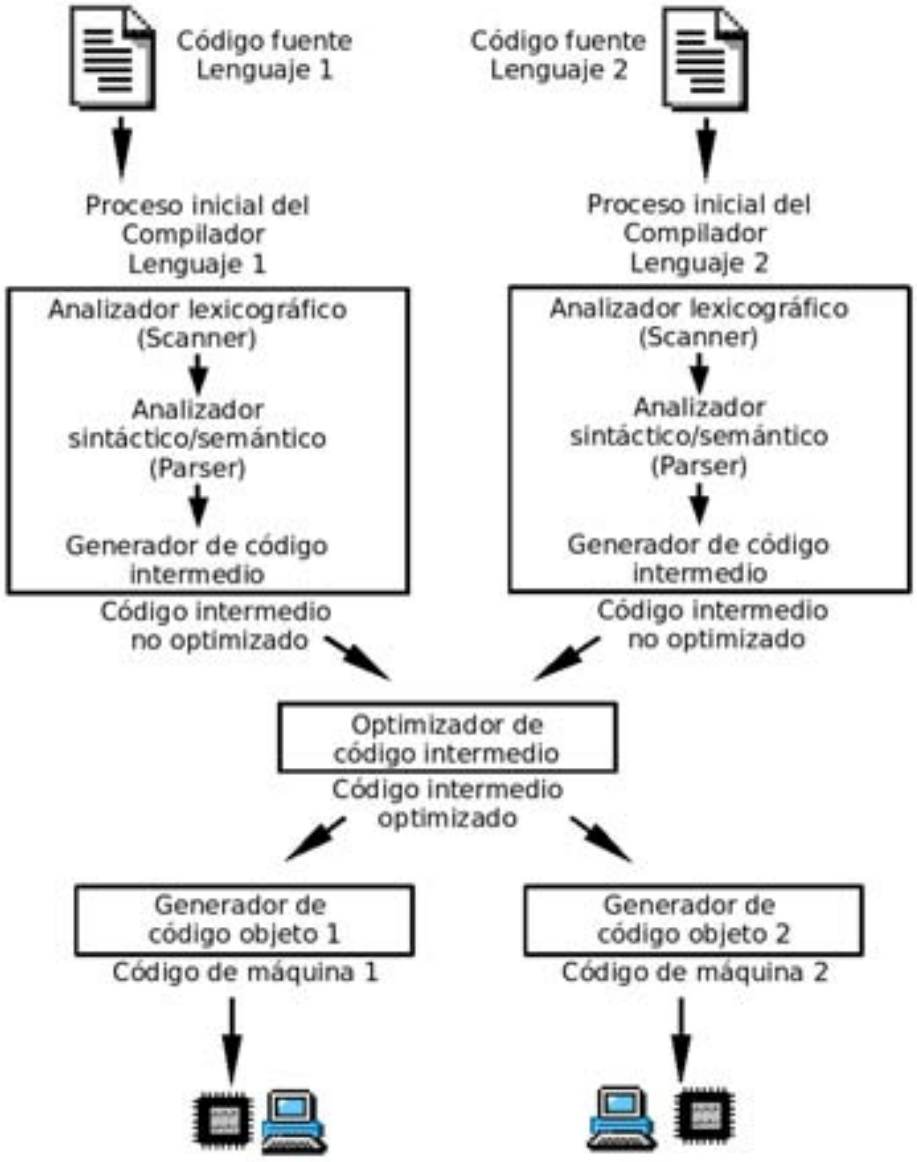
Domingo Gómez Pérez

Dpto. de Matemáticas,

Estadística y Computación

Recordemos

Fuente: Wikipedia



Lenguajes de Programación Involucrados

Incluso antes de analizar cómo es el compilador por dentro

Hay que tener presentes no menos de tres lenguajes:

- El lenguaje fuente en que estarán escritos los programas a compilar;
- el lenguaje objeto (ejecutable o interpretable) en que estarán "escritos" los programas compilados;

Lenguajes de Programación Involucrados

Incluso antes de analizar cómo es el compilador por dentro

Hay que tener presentes no menos de tres lenguajes:

- El lenguaje fuente en que estarán escritos los programas a compilar;
- el lenguaje objeto (ejecutable o interpretable) en que estarán "escritos" los programas compilados;
- el lenguaje en que está "escrito" el compilador.

Lenguajes de Programación Involucrados

Incluso antes de analizar cómo es el compilador por dentro

Hay que tener presentes no menos de tres lenguajes:

- El lenguaje fuente en que estarán escritos los programas a compilar;
- el lenguaje objeto (ejecutable o interpretable) en que estarán "escritos" los programas compilados;
- el lenguaje en que está "escrito" el compilador.

Cuando miramos dentro del compilador, las cosas se complican:

- Lenguajes de especificación de analizadores y gramáticas,
- códigos intermedios...

Una técnica útil: diagramas en T.

Código Intermedio

En acción

Códigos de tres direcciones

- Cada órden son funciones de dos registros;
- solo se puede utilizar tipos básicos, no hay funciones y todo el código se realiza por bloques;
- la gestión de memoria se realiza en el código;

Tendremos que afrontar algunas cuestiones:

- Gestión de etiquetas para implementar mediante saltos condicionales o incondicionales las estructuras de control (“mientras”).
- Tabla de símbolos:
  - comprobar las reglas sobre declaraciones de identificadores;
  - asociar, a cada identificador que encontramos en “tiempo de compilación”, la memoria que le corresponderá en “tiempo de ejecución”.

Lenguaje Fuente

Ingredientes que distinguen unos de otros

Aspectos a considerar:

1. Estructuras de control;
2. Expresiones;
3. Tipos básicos;
4. Tipos estructurados:

Lenguaje Fuente

Ingredientes que distinguen unos de otros

Aspectos a considerar:

1. Estructuras de control;
2. Expresiones;
3. Tipos básicos;
4. Tipos estructurados:
   - tamaño fijo,
   - tamaño variable;

Lenguaje Fuente

Ingredientes que distinguen unos de otros

Aspectos a considerar:

1. Estructuras de control;
2. Expresiones;
3. Tipos básicos;
4. Tipos estructurados:
   - tamaño fijo,
   - tamaño variable;
5. Objetos o punteros, gestión de memoria;

Lenguaje Fuente

Ingredientes que distinguen unos de otros

Aspectos a considerar:

1. Estructuras de control;
2. Expresiones;
3. Tipos básicos;
4. Tipos estructurados:
   - tamaño fijo,
   - tamaño variable;
5. Objetos o punteros, gestión de memoria;
6. Funciones y parámetros; visibilidad (*scoping*);

Lenguaje Fuente

Ingredientes que distinguen unos de otros

Aspectos a considerar:

1. Estructuras de control;
2. Expresiones;
3. Tipos básicos;
4. Tipos estructurados:
   - tamaño fijo,
   - tamaño variable;
5. Objetos o punteros, gestión de memoria;
6. Funciones y parámetros; visibilidad (*scoping*);
7. Recursividad.

Lenguaje Objeto

¿Por qué no directamente código nativo?

Hay varias fuentes de ineficiencia en el código generado:

1. Copiamos a un registro un valor que ya está en él;
2. copiamos a memoria un valor que ya está en esa posición;

Lenguaje Objeto

¿Por qué no directamente código nativo?

Hay varias fuentes de ineficiencia en el código generado:

1. Copiamos a un registro un valor que ya está en él;
2. copiamos a memoria un valor que ya está en esa posición;
3. no aprovechamos bien la disponibilidad de varios registros;
4. no aprovechamos bien todo el repertorio de instrucciones máquina;

Lenguaje Objeto

¿Por qué no directamente código nativo?

Hay varias fuentes de ineficiencia en el código generado:

1. Copiamos a un registro un valor que ya está en él;
2. copiamos a memoria un valor que ya está en esa posición;
3. no aprovechamos bien la disponibilidad de varios registros;
4. no aprovechamos bien todo el repertorio de instrucciones máquina;
5. es muy fácil que quede código "muerto" que en realidad nunca se puede ejecutar.

Lenguaje Objeto

¿Por qué no directamente código nativo?

Hay varias fuentes de ineficiencia en el código generado:

1. Copiamos a un registro un valor que ya está en él;
2. copiamos a memoria un valor que ya está en esa posición;
3. no aprovechamos bien la disponibilidad de varios registros;
4. no aprovechamos bien todo el repertorio de instrucciones máquina;
5. es muy fácil que quede código "muerto" que en realidad nunca se puede ejecutar.

Solución habitual: dos, o a veces tres, lenguajes objeto.

1. generar un código intermedio y analizarlo de varias maneras para "optimizarlo"
2. y luego obtenemos código ejecutable nativo

Lenguaje Objeto

¿Por qué no directamente código nativo?

Hay varias fuentes de ineficiencia en el código generado:

1. Copiamos a un registro un valor que ya está en él;
2. copiamos a memoria un valor que ya está en esa posición;
3. no aprovechamos bien la disponibilidad de varios registros;
4. no aprovechamos bien todo el repertorio de instrucciones máquina;
5. es muy fácil que quede código "muerto" que en realidad nunca se puede ejecutar.

Solución habitual: dos, o a veces tres, lenguajes objeto.

1. generar un código intermedio y analizarlo de varias maneras para "optimizarlo"
2. y luego obtenemos código ejecutable nativo
3. o bien un código interpretable por una máquina virtual.

Desiderata

Propiedades deseables:

- Próximo al lenguaje fuente (facilita el "front-end").
- Próximo al lenguaje objeto (facilita el "back-end").
- Permite implementar algoritmos sofisticados sobre fragmentos de código (facilita la "optimización" de código).

Desiderata

Propiedades deseables:

- Próximo al lenguaje fuente (facilita el "front-end").
- Próximo al lenguaje objeto (facilita el "back-end").
- Permite implementar algoritmos sofisticados sobre fragmentos de código (facilita la "optimización" de código).

Evidentemente, esos requisitos entran en conflicto mutuamente.

Desiderata

Propiedades deseables:

- Próximo al lenguaje fuente (facilita el "front-end").
- Próximo al lenguaje objeto (facilita el "back-end").
- Permite implementar algoritmos sofisticados sobre fragmentos de código (facilita la "optimización" de código).

Evidentemente, esos requisitos entran en conflicto mutuamente.

Optimización (nombre inadecuado): es especialmente importante porque el código generado suele tener ineficiencias obvias.

Desiderata

Propiedades deseables:

- Próximo al lenguaje fuente (facilita el "front-end").
- Próximo al lenguaje objeto (facilita el "back-end").
- Permite implementar algoritmos sofisticados sobre fragmentos de código (facilita la "optimización" de código).

Evidentemente, esos requisitos entran en conflicto mutuamente.

Optimización (nombre inadecuado): es especialmente importante porque el código generado suele tener ineficiencias obvias.

Hay que afrontar esa tarea con mucha precaución.

Desiderata

Propiedades deseables:

- Próximo al lenguaje fuente (facilita el "front-end").
- Próximo al lenguaje objeto (facilita el "back-end").
- Permite implementar algoritmos sofisticados sobre fragmentos de código (facilita la "optimización" de código).

Evidentemente, esos requisitos entran en conflicto mutuamente.

Optimización (nombre inadecuado): es especialmente importante porque el código generado suele tener ineficiencias obvias.

Hay que afrontar esa tarea con mucha precaución.

"Premature optimization is the root of all evil."
(D.E. Knuth, C.A.R. Hoare)

Códigos Intermedios

Motivo del plural

Nada impide usar más de un código intermedio.

Cuando la distancia entre el código ejecutable y el fuente es más grande, puede ser conveniente usar dos.

- Facilitan el diseño del compilador.
- Cada tarea a realizar se puede elegir en qué fase se realiza.

Códigos Intermedios

Motivo del plural

Nada impide usar más de un código intermedio.

Cuando la distancia entre el código ejecutable y el fuente es más grande, puede ser conveniente usar dos.

- Facilitan el diseño del compilador.
- Cada tarea a realizar se puede elegir en qué fase se realiza.

  (En particular, las tareas de optimización de código.)

Códigos Intermedios

Motivo del plural

Nada impide usar más de un código intermedio.

Cuando la distancia entre el código ejecutable y el fuente es más grande, puede ser conveniente usar dos.

- Facilitan el diseño del compilador.
- Cada tarea a realizar se puede elegir en qué fase se realiza.

  (En particular, las tareas de optimización de código.)
- Se puede decidir interpretar pero no compilar el último código intermedio

Códigos Intermedios

Motivo del plural

Nada impide usar más de un código intermedio.

Cuando la distancia entre el código ejecutable y el fuente es más grande, puede ser conveniente usar dos.

- Facilitan el diseño del compilador.
- Cada tarea a realizar se puede elegir en qué fase se realiza.

  (En particular, las tareas de optimización de código.)
- Se puede decidir interpretar pero no compilar el último código intermedio

  (o hacerlo sólo "just-in-time").

¿Y cómo son?

1. Código de tres direcciones, con o sin llamadas a función.
2. Representación en árboles similares a los árboles sintácticos.

Cuestiones Prácticas

¿Cómo lo combinamos con el análisis sintáctico y semántico?

La última fase de análisis dispone de una estructura de datos en la que se construye el código intermedio generado.

Puede ser:

- Una estructura arborescente (*abstract syntax tree*), similar al árbol de análisis sintáctico pero reducido a la información imprescindible.
- Una estructura arborescente pero más alejada de la sintaxis, orientada al código a generar.
- Una estructura secuencial en memoria, lo cual nos permite varios recorridos (por ejemplo para colocar etiquetas de salto coherentes);
- Directamente el fichero de salida, en el que vamos escribiendo el resultado del paso correspondiente;

Depende de cuánto de lejos estén el lenguaje fuente y el objeto.

Código de Tres Direcciones

Uno concreto y sencillísimo para nuestros primeros ejemplos y ejercicios

Vamos a simplificar un poco el código de tres direcciones.

```
# comentario
$p2 :read              # lectura
$p2 :write             # escritura
$p4 : $p3              # copia
$p21 : $p32 + $p13     # operacion, admite +, -, *, /
ident:                 # etiqueta
$p8 : jump ident       # salto condicional: salta si <= 0
```

Simplificaciones:

Todas las variables son "temporales" (se ordena leer y escribir en memoria sin decir donde) y equivalentes entre sí(todos son números enteros).

(No distinguimos acumulador, ni registros entre sí, ni registros de posiciones de memoria; no hay *offsets*. . .)

Un Ejemplo Sencillo

El de siempre, claro

```
### 3-address code
                    # "el factorial"
$p2 :read           # dato
$p3 : 1             # inicializa a uno
loop0:              # bucle
$p2 : jump end0     # salir si dato cero o negativo
$p3 : $p3 * $p2     # multiplica
$p2 : $p2 - $p1     # reduce
$p3 :write          # escribe resultado parcial
$p0 : jump loop0    # comprueba condicion del bucle
end0:               # fin de bucle y programa

### End of 3-address code
```

Modus Operandi

¿Cómo se logra construir cosas así?

Ejemplo:

La rutina semántica correspondiente a multiplicar expresiones.

`emit()`: función que "emite" una nueva instrucción de tres direcciones y la añade al código que estamos generando.

Valor semántico de `expr`: posición de la variable que contiene el resultado.

```
expr : expr POR expr
   {
   tempvarnum = newtempvar();
   emit("$p",tempvarnum," : $p",$1," * p",$3," # mult.\n")
   $$ = tempvarnum;
   }
```

Traducción de una Expresión

La traducción es una función que toma una Expresión, la tabla de símbolos, la tabla de funciónes y un registro. Se denota por *TransExp(Exp,vtable,ftable,place) devuelve Code*

num	v= value(num) place=v
id	x=lookup(vtable,name(id)) place=x
unop Exp1	code=TransExp(Exp1,vtable,ftable,$p0) code+$pe= unop $p0+place=$pe
Exp1 binop Exp2	code1=TransExp(Exp1,vtable,ftable,$p0) code2=TransExp(Exp2,vtable,ftable,$p1) code1+code2+$pe=$p0 binop $p1+place=$pe

Cuadro: Tabla de Traducción

Tabla de Traducción

Ahora traducimos conjuntos de Expresiones,
TransExps(Exp,vtable,ftable) devuelve Code

Exp	code=TransExp(Exp,vtable,ftable,p0)
Exps Exp	code=TransExp(Exp,vtable,ftable,p0) code1=TransExps(Exps,vtable,ftable) code+code1

Cuadro: Tabla de Traducción

Generación de Código Intermedio

A partir de aquí, ir siguiendo

Planteamos sucesivamente:

- generación de código para cada forma de las expresiones (gestión de variables temporales),
- generación de código para el repertorio de instrucciones de alto nivel (gestión de las etiquetas),
- generación de código para tipos estructurados (gestión de zonas de memoria),
- generación de código que use el *heap* con punteros,
- generación de código para llamadas a función (gestión de parámetros y resultados, gestión de ámbitos de visibilidad, recursividad. . .)
- . . .

En cuanto tengamos los diagramas en T, haremos algunas de estas cosas para nuestro código simplificado y para MIPS.