Tema 2.2. Ampliación del lenguaje y su procesador: *Optimización de la traducción*

Profesor
Federico Peinado

Elaboración del material
José Luis Sierra
Federico Peinado

Ingeniería en Informática
Facultad de Informática – Universidad Complutense de Madrid
Curso 2009-2010

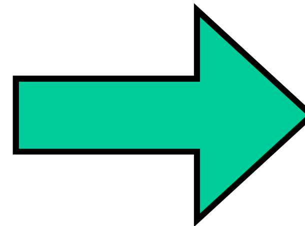
Optimización de la traducción

- En general, **optimizar un traductor** es hacer que genere programas *equivalentes pero más eficientes* en tiempo de ejecución o consumo de memoria
  - Disciplina fundamental en el diseño de compiladores reales
- Hay distintos aspectos que pueden ser optimizados:
  - Las estrategias de traducción a seguir
  - El código objeto una vez generado
  - El propio código fuente
- Nosotros sólo veremos algunos casos básicos, aunque Aho et al. enseñan métodos más complejos y sistemáticos

Traducción de instrucciones if-then-else sin bloque else

- Cuando una instrucción **if-then-else** no tiene bloque **else** se genera siempre un código en el bloque **then** con un salto **totalmente innecesario** a la instrucción siguiente
- Ejemplo:

```
if x = 5 then
  x := x + 1;
```



x ≡ Mem[0]

```
...
apila-dir(0)      0
apila(5)          1
igual             2
ir-f(9)           3
apila-dir(0)      4
apila(1)          5
suma              6
desapila-dir(0)   7
ir-a(9)           8
                  9
```

- ***La traducción se puede optimizar*** generando o no el código de ese salto según el bloque **else** esté vacío o no

Ampliación de la propuesta: if-then-else sin bloque else

- En nuestras ecuaciones semánticas de la gramática de atributos esta optimización queda así:

```
...
IIf ::= if Exp then I PElse
  IIf.cod = Exp.cod || ir-f(PElse.etqi) || I.cod ||
            PElse.cod
  Exp.etqh = IIf.etqh
  I.etqh = Exp.etq + 1
  PElse.etqh = I.etq
  IIf.etq = PElse.etq
PElse ::= else I
  PElse.cod = ir-a(I.etq) || I.cod
  I.etqh = PElse.etqi = PElse.etqh + 1
  PElse.etq = I.etq
PElse ::= λ
  PElse.cod = λ
  PElse.etq = PElse.etqi = PElse.etqh
```

Ampliación de la propuesta: if-then-else sin bloque else

- Y en los esquemas de traducción optimizados queda así:

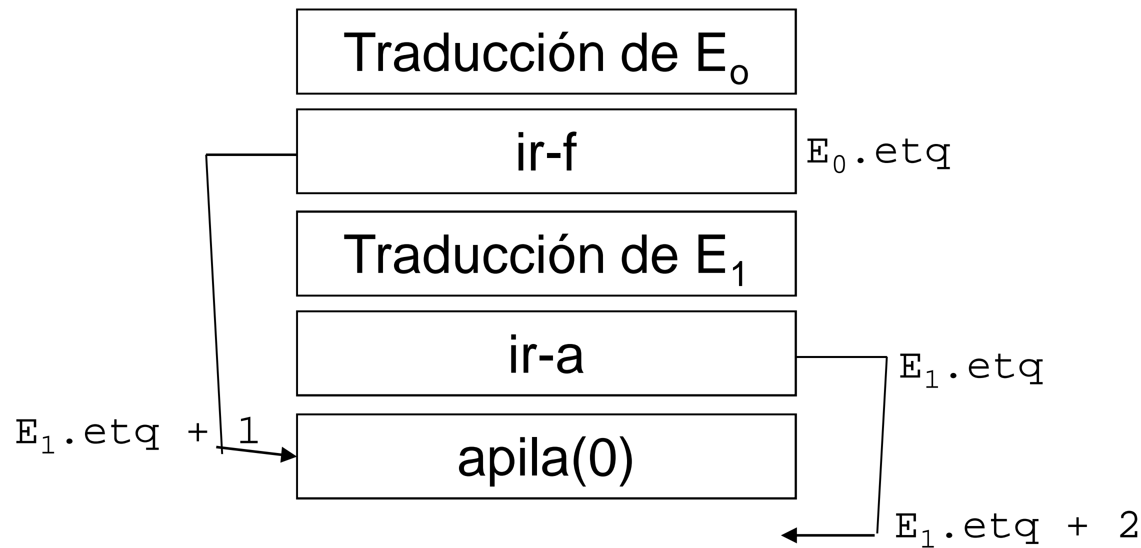
```
global cod, etq;
...
IIf::= {var etqaux}
       if Exp then
       {emite(ir-f(?));
        etqaux ← etq;
        etq ← etq + 1}
       I
       PElse(out etqi)
       {parchea(etqaux, etqi)}
```

Ampliación de la propuesta: if-then-else sin bloque else

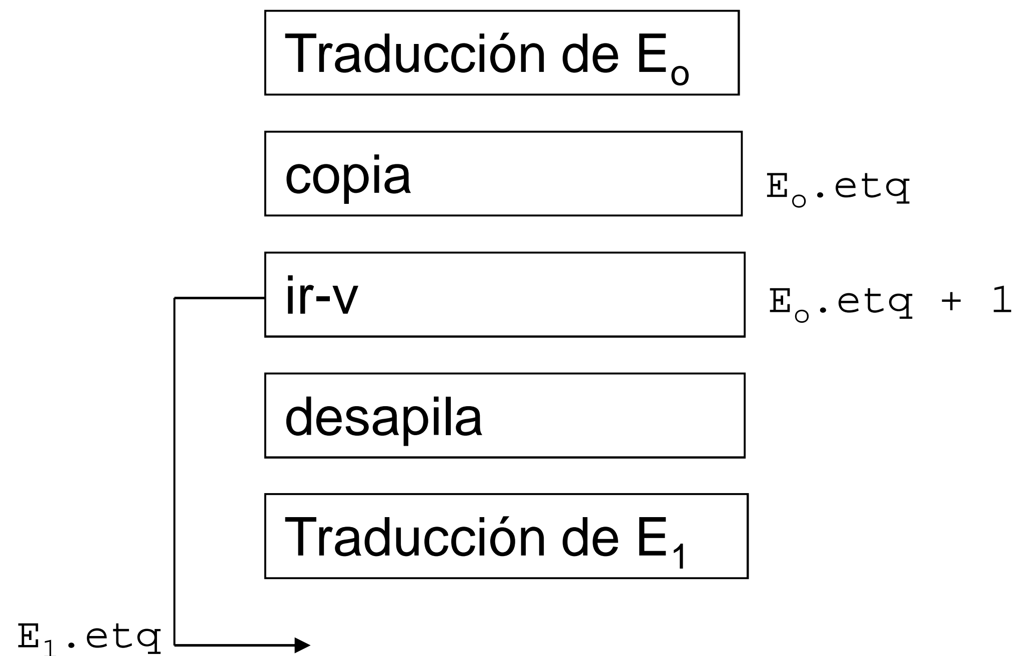
Continuación…

```
global cod, etq;
...
PElse(out etqi)::= {var etqaux}
                   else
                   {emite(ir-a(?));
                    etqaux ← etq;
                    etq ← etq + 1;
                    etqi ← etq}
                   I
                   {parchea(etqaux, etq)}
PElse(out etqi)::= λ
                   {etqi ← etq}
```

- Es habitual traducir las expresiones booleanas para que se calcule *solamente* el valor de aquellos *operandos imprescindibles* para conocer el resultado de la expresión
  - Usamos las mismas ideas de las sentencias condicionales
- Esquema de $\mathbf{E_0}$ **and** $\mathbf{E_1}$
  - Si E_0 es falso, todo vale falso; y si no, vale lo que valga E_1



- Esquema de **E_0 or E_1**
  - Si E_0 no es falso, todo vale lo que valga E_0 (verdadero); y si no, vale lo que valga E_1



* **copia** duplica la cima de la pila (lo copia otra vez apilándolo encima)
* **ir-v** realiza el salto si hay verdad en la cima de la pila (lo contrario de ir-f)
* **desapila** descarta la cima de la pila

- `copia`

```
Pila [CPila + 1] ← Pila[CPila]
CPila ← CPila + 1
CProg ← CProg + 1
```

- `ir-v(`*i*`)`

```
si Pila[CPila] ≠ 0
 CProg ← i
si no
  CProg ← CProg + 1
fsi
CPila ← CPila - 1
```

- `desapila`

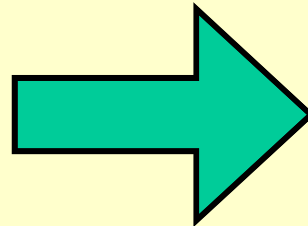
```
CPila ← CPila - 1
CProg ← CProg + 1
```

Traducción de expresiones booleanas en circuito corto

- Ejemplo:

```
(x=y) and (y=z) or
(u=v) and (w = r)
```

x ≡ Mem[0]
y ≡ Mem[1]
z ≡ Mem[2]
u ≡ Mem[3]
v ≡ Mem[4]
w ≡ Mem[5]
r ≡ Mem[6]
...



```
apila-dir(0)     0
apila-dir(1)     1
igual            2
ir-f(8)          3
apila-dir(1)     4
apila-dir(2)     5
igual            6
ir-a(9)          7
apila(0)         8
copia            9
ir-v(21)         10
desapila         11
apila-dir(3)     12
apila-dir(4)     13
igual            14
ir-f(20)         15
apila-dir(5)     16
apila-dir(6)     17
igual            18
ir-a(21)         19
apila(0)         20
                 21
```

Ampliación de la propuesta: Expresiones booleanas en circuito corto

- Para facilitar esta generación de código optimizado es conveniente **modificar la gramática incontextual** para discriminar explícitamente los operadores lógicos del resto
  - Añadiendo un par de producciones para los operadores:

```
...
ExpS ::= ExpS or Term
...
Term ::= Term and Fact
```

  - Eliminando los operadores de las otras producciones:

```
...
OpAd ::= + | - | or
...
OpMul ::= * | / | div | mod | and
```

Ampliación de la propuesta: Expresiones booleanas en circuito corto

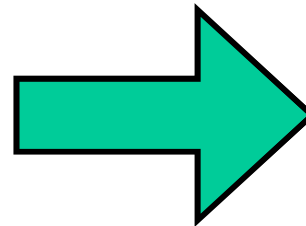
- De esa forma podemos definir las partes correspondientes a la traducción optimizada en la gramática de atributos:

```
...
ExpS ::= ExpS or Term
  ExpSo.cod = ExpS1.cod || copia || ir-v(Term.etq) ||
              desapila || Term.cod
  ExpS1.etqh = ExpSo.etqh
  Term.etqh = Exps1.etq + 3
  ExpSo.etq = Term.etq
...
Term ::= Term and Fact
  Termo.cod = Term1.cod || ir-f(Fact.etq + 1) ||
              Fact.cod || ir-a(Fact.etq + 2) || apila(0)
  Term1.etqh = Termo.etqh
  Fact.etqh = Term1.etq + 1
  Termo.etq = Fact.etq + 2
```

Optimización de la evaluación en circuito corto

- La optimización propuesta para la traducción de las expresiones booleanas produce saltos consecutivos **ineficientes** al anidar operadores lógicos
- Ejemplo:

```
(x=5) or (x > 6)
   or (x < -1)
```



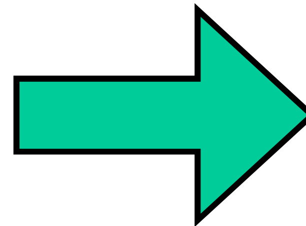
x ≡ Mem[0]

```
...
apila-dir(0)    0
apila(5)        1
igual           2
copia           3
ir-v(9)         4
desapila        5
apila-dir(0)    6
apila(6)        7
mayor           8
copia           9
ir-v(15)       10
desapila       11
apila-dir(0)   12
apila(-1)      13
menor          14
               15
```

Optimización de la evaluación en circuito corto

- ***La traducción se puede optimizar***, por segunda vez, haciendo que los saltos lleguen al final del anidamiento (que sean "saltos completos")
- Ejemplo:

```
(x=5) or (x > 6)
   or (x < -1)
```

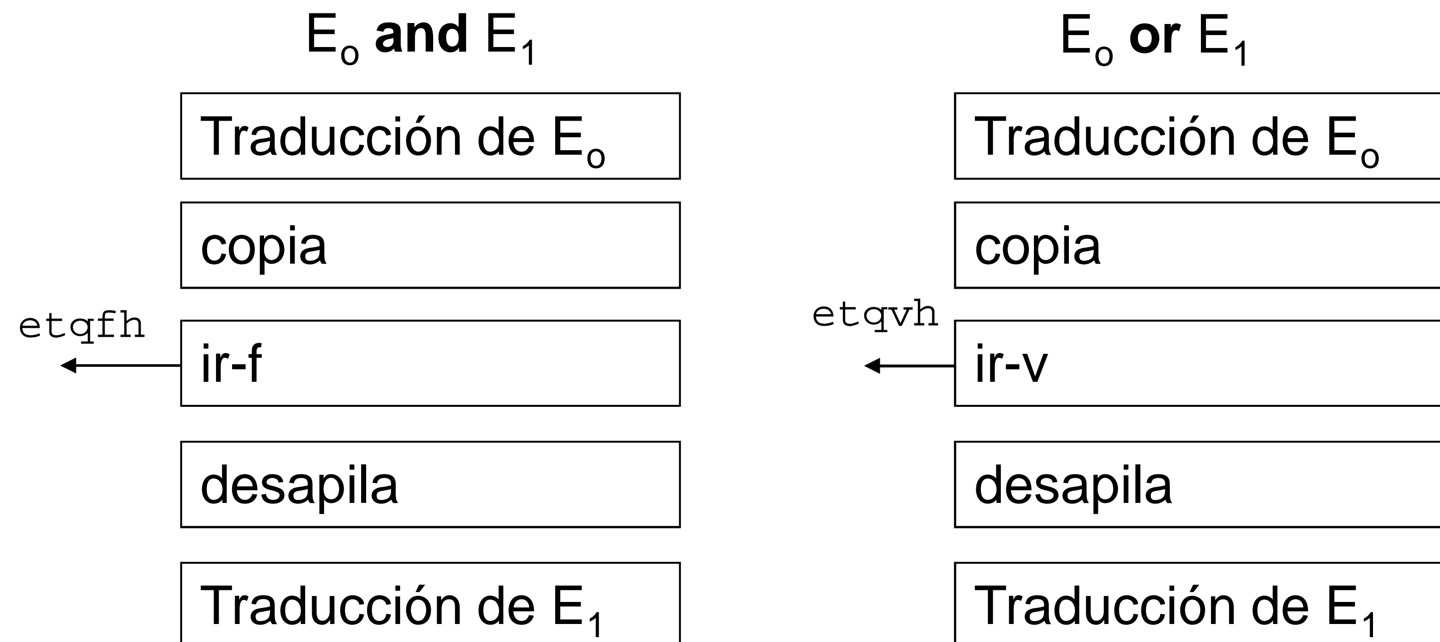


x ≡ Mem[0]

```
...
apila-dir(0)    0
apila(5)        1
igual           2
copia           3
ir-v(15)        4
desapila        5
apila-dir(0)    6
apila(6)        7
mayor           8
copia           9
ir-v(15)       10
desapila       11
apila-dir(0)   12
apila(-1)      13
menor          14
               15
```

Optimización de la evaluación en circuito corto

- La idea ahora es fijar *siempre* las etiquetas de los saltos condicionales **desde las estructuras de nivel superior**
  - Los saltos **ir-f** / **ir-v** serán *completos*: tendrán como dirección de destino el valor de un atributo heredado **`etqfh`** / **`etqvh`** que indica el final del anidamiento de operadores **and** / **or**



Ampliación de la propuesta: Optimización del circuito corto

- Así se expresan los valores de `etqfh` y `etqvh` en las ecuaciones semánticas de las **expresiones** y de **cualquier otra estructura que utilice expresiones**:

```
...
IWhile::= while Exp do I
  Exp.etqfh = Exp.etqvh = Exp.etq
...
Iif:: if Exp then I PElse
  Exp.etqfh = Exp.etqvh = Exp.etq


...

Exp ::= ExpS OpComp ExpS
  ExpS0.etqfh = ExpS0.etqvh = ExpS0.etq
  ExpS1.etqfh = ExpS1.etqvh = ExpS1.etq
...
```

Ampliación de la propuesta: Optimización del circuito corto

Continuación…

```
...
Exp ::= ExpS
  ExpS.etqfh = Exp.etqfh
  ExpS.etqvh = Exp.etqvh
ExpS ::= ExpS OpAd Term
  ExpS1.etqfh = ExpS1.etqvh = ExpS1.etq
  Term.etqfh = Term.etqvh = Term.etq
ExpS ::= Term
  Term.etqfh = ExpS.etqfh
  Term.etqvh = ExpS.etqvh
Term ::= Term OpMul Fact
  Term1.etqfh = Term1.etqvh = Term1.etq
  Fact.etqfh  = Fact.etqvh  = Fact.etq
Term ::= Fact
  Fact.etqfh = Term.etqfh
  Fact.etqvh = Term.etqvh
Fact ::= not Fact
  Fact1.etqfh = Fact1.etqvh = Fact1.etq
Fact ::= ( Exp )
  Exp.etqfh = Fact.etqfh
  Exp.etqvh = Fact.etqvh
...
```

Ampliación de la propuesta: Optimización del circuito corto

Continuación…

```
...
ExpS ::= ExpS or Term
  ExpS0.cod = ExpS1.cod || copia ||
              ir-v(ExpS0.etqvh) || desapila ||
              Term.cod
  ExpS1.etqfh = ExpS1.etq + 2
  Term.etqfh = Term.etq
  ExpS1.etqvh = ExpS0.etqvh
  Term.etqvh = ExpS0.etqvh
Term ::= Term and Fact
  Term0.cod = Term1.cod || copia ||
              ir-f(Term0.etqfh) || desapila ||
              Fact.cod
  Term1.etqfh = Term0.etqfh
  Fact.etqfh = Term0.etqfh
  Term1.etqvh = Term1.etq + 2
  Fact.etqvh = Fact.etq
```

Optimización de la evaluación en circuito corto

- La anterior gramática de atributos **no es l-atribuida**: ¡hay muchas categorías sintácticas que *heredan de sí mismas*!
  - Por ejemplo: `Exp.etqfh = Exp.etqvh = Exp.etq`
- Esto es un problema que requiere volver a utilizar la idea del parcheo (ahora *en la propia gramática de atributos*), **parcheando muchos saltos,** todos con el mismo destino
- Para ello añadimos **dos atributos sintetizados** que almacenan *listas de instrucciones pendientes de parchear*
  - Los saltos **ir-f** con valor indefinido se guardarán en **`listaf`**
  - Los saltos **ir-v** con valor indefinido se guardarán en **`listav`**
- Y también añadimos una *función semántica* **parchea** que realiza el parcheo "masivo" de las dos listas de instrucciones, dada la dirección de destino que hay que usar en las instrucciones de salto de cada una de ellas
  - **parchea** (`cod`, **`listaf`**, **`listav`**, **`etqf`**, **`etqv`**)

Ampliación de la propuesta: Optimización del circuito corto

- Así quedan las ecuaciones semánticas de la traducción:

```
...
IWhile::= while Exp do I
  IWhile.cod = parchea(Exp.cod, Exp.listaf,
                       Exp.listav, Exp.etq, Exp.etq) ||
               ir-f(I.etq + 1) || I.cod ||
               ir-a(IWhile.etqh)
...
IIf:: if Exp then I PElse
  IIf.cod = parchea(Exp.cod, Exp.listaf, Exp.listav,
                    Exp.etq, Exp.etq) ||
            ir-f(PElse.etqi) || I.cod ||
            PElse.cod
...
```

Ampliación de la propuesta: Optimización del circuito corto

Continuación…

```
...
Exp ::= ExpS OpComp ExpS
  Exp.cod = parchea(ExpS0.cod, ExpS0.listaf, ExpS0.listav,
                     ExpS0.etq, ExpS0.etq) ||
            parchea(ExpS1.cod, ExpS1.listaf, ExpS1.listav,
                     ExpS1.etq, ExpS1.etq) || OpComp.op
  Exp.listaf = Exp.listav = []
Exp ::= ExpS
  Exp.listaf = ExpS.listaf
  Exp.listav = ExpS.listav
ExpS ::= ExpS OpAd Term
  ExpS0.cod = parchea(ExpS1.cod,ExpS1.listaf,ExpS1.listav,
                     ExpS1.etq,ExpS1.etq) ||
              parchea(Term.cod,Term.listaf,Term.listav,
                     Term.etq,Term.etq) ||
             OpAd.op
  ExpS0.listaf = ExpS0.listav = []
ExpS ::= Term
  ExpS.listaf = Term.listaf
  ExpS.listav = Term.listav
...
```

Ampliación de la propuesta: Optimización del circuito corto

Continuación…

```
...
Term ::= Term OpMul Fact
  Termo.cod = parchea(Term1.cod, Term1.listaf, Term1.listav,
                      Term1.etq, Term1.etq) ||
              parchea(Fact.cod, Fact.listaf, Fact.listav,
                      Fact.etq, Fact.etq) || OpMul.op
  Termo.listaf = Termo.listav = []
Term ::= Fact
  Term.listaf = Fact.listaf
  Term.listav = Fact.listav
Fact ::= not Fact
  Fact0.cod = parchea(Fact1.cod, Fact1.listaf, Fact1.listav,
                      Fact1.etq, Fact1.etq)
  Fact0.listaf = Facto.listav = []
Fact ::= iden
  Fact.listaf = Fact.listav = []
Fact ::= número
  Fact.listaf = Fact.listav = []
Fact ::= ( Exp )
  Fact.listaf = Exp.listaf
  Fact.listav = Exp.listav
...
```

Ampliación de la propuesta: Optimización del circuito corto

Continuación…

```
...
ExpS ::= ExpS or Term
  ExpS0.cod = parchea(ExpS1.cod, ExpS1.listaf, [],
                      ExpS1.etq + 2, ?) ||
              copia || ir-v(?) || desapila ||
              parchea(Term.cod, Term.listaf, [], Term.etq, ?)
  ExpS0.listav = ExpS1.listav ++ Term.listav ++ [ExpS1.etq + 1]
  ExpS0.listaf = []
Term ::= Term and Fact
   Term0.cod = parchea(Term1.cod, [], Term1.listav,
                       ?, Term1.etq + 2) ||
               copia || ir-f(?) || desapila ||
               parchea(Fact.cod, [], Fact.listav, ?, Fact.etq)
   Term0.listf = Term1.listaf ++ Fact.listaf ++ [Term1.etq + 1]
   Term0.listav = []
```

* ***[ ]*** y ***?*** se usan en parchea cuando realmente sólo interesa parchear una lista

* **++** es la concatenación de listas

Ampliación de la propuesta: Optimización del circuito corto

- En los esquemas de traducción optimizados queda así:

```
global cod, etq;
...
IWhile ::= {var etqaux1, etqaux2}
            while
            {etqaux1 ← etq}
            Exp
            do
            {parchea(Exp.listav,Exp.listaf,etq,etq);
             emite(ir-f(?));
             etqaux2 ← etq;
             etq ← etq + 1}
            I
           {emite(ir-a(etqaux1));
            etq ← etq + 1;
            parchea(etqaux2,etq)}
```

* **parchea(lv, lf, ev, ef)** realiza parcheo masivo de *lv* con *ev* y de *lf* con *ef*

Ampliación de la propuesta: Optimización del circuito corto

Continuación…

```
global cod, etq;
...
ExpS(in listavh0, listafh0; out listav0, listaf0) ::=
        {var etqaux}
        ExpS(in listavh1, listafh1; out listav1, listaf1)
        {parchea([], listaf1, ?, etq + 2)}
        or
        {emite(copia);
         emite(ir-v(?));
         emite(desapila);
         etqaux ← etq + 1;
         etqaux ← etq + 3;}
        Term(in listavh2, listafh2; out listav2, listaf2)
        {parchea([], listaf2, ?, etq);
         listav0 = listav1 ++ listav2 ++ [etqaux];
         listaf0 = [];}
...
```

Traducción con precálculo de expresiones constantes

- Aunque la tarea del traductor no es interpretar el código fuente, **precalcular el valor de las expresiones constantes** será una excepción que realizaremos para *optimizar el código fuente*
- Esta optimización puede hacerse de varias formas
  - **Realizando dos pasadas**
    1. *Transformando* el código fuente
    2. Traduciendo el código fuente transformado
  - **Realizando una pasada de transformación y traducción**
    - El código fuente se transforma según se traduce (se complica un poco el proceso, pero será **nuestro enfoque**)
  - **Realizando una pasada y trabajando sobre una representación intermedia del programa**
    - A partir del código fuente se genera una *representación intermedia del programa* (= árbol de sintaxis abstracta)
    - Se *transforma* la representación intermedia del programa
    - Se genera código a partir de la representación intermedia del programa transformada

Traducción con precálculo de expresiones constantes

- Reglas de optimización
  - Una **expresión constante** se ve reemplazada por su `valor`
  - **true** y **false** son expresiones constantes de valor `cierto` y `falso`
  - **Número** es expresión constante de valor `valor(Número.lex)`
  - Cualquier operación **Op** sobre expresiones constantes **X** e **Y** es expresión constante de valor `opera(valor(X), valor(Y))`
  - La asignación de una expresión constante **X** a una variable **Id** añade `Id.lex` a una *tabla de variables constantes* **varscte** (formada por pares lexema-valor) con el valor asociado `valor(X)`
  - La asignación de una expresión no constante a una variable **Id** elimina **Id** de la tabla de variables constantes **varscte**
  - Una variable **Id** de la *tabla de variables constantes* **varscte** es expresión constante de valor `varscte.valor(Id.lex)`
  - Una instrucción **if-then-else** con expresión constante como condición es equivalente a su bloque **then** si la condición vale `cierto`, y a su bloque **else** si vale `falso`
  - Una instrucción **while-do** con expresión constante como condición es equivalente a λ
  - Etc.

Ampliación de la propuesta: Precálculo de expresiones constantes

```
Prog ::= Decs & Is
  Prog.cod = Decs.cod ++ "&" ++ Is.cod
  Is.varscteh = []
  Prog.varscte = Is.varscte
Decs ::= Decs ; Dec
  Decso.cod = Decs1.cod ++ ";" ++ Dec.cod
Decs ::= Dec
  Decs.cod = Dec.cod
Dec ::= Tipo iden
  Dec.cod = Tipo.cod ++ " " ++ iden.lex
Tipo ::= bool
  Tipo.cod = "bool"
Tipo ::= int
  Tipo.cod = "int"
```

Ampliación de la propuesta: Precálculo de expresiones constantes

Continuación…

```
Is ::= Is ; I
  Iso.cod = si I.cod ≠ λ
                 Is1.cod ++ ";" ++ I.cod
            si no Is1.cod
  Is1.varscteh = Iso.varscteh
  I.varscteh   = Is1.varscte
  Iso.varscte  = I.varscte
Is ::= I
  Is.cod = I.cod
  I.varscteh = Is.varscteh
  Is.varscte = I.varscte
I ::= IAsig
  I.cod = IAsig.cod
  IAsig.varscteh = I.varscteh
  I.varscte = IAsig.varscte
I ::= IIf
  I.cod = IIf.cod
  IIf.varscteh = IIf.varscteh
  IIf.varscte = IIf.varscte
```

Ampliación de la propuesta: Precálculo de expresiones constantes

Continuación…

```
I ::= IWhile
  I.cod = IIIf.cod
  IWhile.varscteh = IWhile.varscteh
  IWhile.varscte = IWhile.varscte
I ::= IComp
  I.cod = IComp.cod
  IComp.varscteh = I.varscteh
  I.varscte = IComp.varscte
IComp ::= begin IsOpc end
  IComp.cod = "begin "++IsOpc.cod++" end"
  IsOpc.varscteh = IComp.varscteh
  IComp.varscte = IsOps.varscte
IsOpc ::= λ
  IsOpc.cod = λ
  IsOpc.varscte = IsOpc.varscteh
IsOpc ::= Is
  IsOpc.cod = Is.cod
  Is.varscteh = IsOpc.varscteh
  IsOpc.varscte = Is.varscte
```

Ampliación de la propuesta: Precálculo de expresiones constantes

Continuación…

```
IAsig ::= iden := Exp
  IAsig.cod = iden.lex ++ ":=" ++ si Exp.cte entonces Exp.val
                                  si no Exp.cod
  IAsig.varscte = si Exp.cte entonces
                     añade(IAsig.varscteh,<iden.lex,Exp.val>)
                  si no elimina(IAsig.varscteh,iden.lex)
  Exp.varscteh = IAsig.varscte
IIf ::= if Exp then I PElse
  IIF.cod = si Exp.cte
             si Exp.val entonces I.cod
             si no PElse.cod
            si no "if "++Exp.cod++ " then "++I.cod++PElse.cod
  IIF.varscte = si Exp.cte
                 si Exp.val entonces I.varscte
                 si no PElse.varscte
                si no I.varscte ∩ PElse.varscte
  I.varscteh = IIf.varscteh
   PElse.varscteh = IIf.varscteh
```

Ampliación de la propuesta: Precálculo de expresiones constantes

Continuación…

```
PElse ::= else I
   PElse.cod = " else" ++ I.cod
   I.varscteh = PElse.varscteh
     PElse.varscte = I.varscte
PElse ::= λ
   PElse.cod = λ
   PElse.varscte = PElse.varscteh
IWhile ::= while Exp do I
   IWHile.cod = si Exp.cte ∧ ¬Exp.val entonces λ
                si no "while "++Exp.cod++" do "++I.cod
   I.varscteh = Exp.varscteh = []
   IWhile.varscte = si Exp.cte ∧ Exp.val entonces
                          IWhile.varscteh
                                 si no I.varscte
```

Ampliación de la propuesta: Precálculo de expresiones constantes

Continuación…

```
Exp ::= ExpS OpComp ExpS
  Exp.cod = (si ExpSo.cte entonces ExpSo.val
             si no ExpSo.cod) ++ OpComp.cod ++
            (si ExpS1.cte entonces ExpS1.val
             si no ExpS1.cod)
  Exp.cte = ExpSo.cte ∧ ExpS1.cte
  Exp.val = eval(OpComp.cod,ExpSo.val, ExpS1.val)
  ExpSo.varscteh = ExpS1.varscteh = Exp.varscteh
Exp ::= ExpS
  Exp.cod = ExpS.cod
  Exp.cte = ExpS.cte
  Exp.val = ExpS.val
  ExpS.varscteh = Exp.varscteh
```

Ampliación de la propuesta: Precálculo de expresiones constantes

Continuación…

```
ExpS ::= ExpS OpAd Term
  ExpSo.cod = (si ExpS1.cte entonces ExpS1.val
                si no ExpS1.cod) ++ OpAd.cod ++
             (si Term.cte entonces Term.val
              si no Term.cod)
  ExpSo.cte = ExpS1.cte ∧ Term.cte
  ExpSo.val = eval(OpAd.cod,ExpS1.val,Term.val)
  ExpS1.varscteh = Term.varscteh = ExpSo.varscteh
Exp ::= ExpS
  Exp.cod = ExpS.cod
  Exp.cte = ExpS.cte
  Exp.val = ExpS.val
  ExpS.varscteh = Exp.varscteh
```

Ampliación de la propuesta: Precálculo de expresiones constantes

Continuación…

```
Term ::= Term OpMul Fact
  Termo.cod = (si Term1.cte entonces Term1.val
                 si no Term1.cod) ++ OpMul.cod ++
              (si Fact.cte entonces Fact.val
               si no Fact.cod)
  Termo.cte = Term1.cte ∧ Fact.cte
  Termo.val = eval(OpMul.cod,Term1.val,Fact.val)
  Term1.varscteh = Fact.varscteh = Termo.varscteh
Term ::= Fact
  Term.cod = Fact.cod
  Term.cte = Fact.cte
  Term.val = Fact.val
  Fact.varscteh = Term.varscteh
```

Ampliación de la propuesta: Precálculo de expresiones constantes

Continuación…

```
Fact ::= not Fact
    Facto.cod = "not " ++ (si Fact1.cte entonces “pendiente"
                           si no Fact1.cod)
    Facto.cte = Fact1.cte
    Facto.val = eval("not",Fact1.val)
    Fact1.varscteh = Facto.varscteh
 Fact ::= ( Exp )
    Fact.cod = "("++ (si Exp.cte entonces Exp.val
                      si no Exp.cod) ++ ")"
    Fact.cte = Exp.cte
    Fact.val = Exp.val
    Exp.varscteh = Fact.varscteh
 Fact ::= num
    Fact.cte = true
    Fact.val = toNum(num.lex)
    Fact.cod = “pendiente 2"
```

Ampliación de la propuesta: Precálculo de expresiones constantes

Continuación…

```
ExpS ::= ExpS or Term
   ExpSo.cod =
     si ExpS1.cte ∧ ExpS1.val entonces "true"
     si no si ExpS1.cte ∧ ¬Term.cte entonces Term.cod
     si no si ExpS1.cte ∧ Term.val entonces "true"
     si no si ExpS1.cte ∧ ¬Term.val entonces "false"
     si no si Term.cte ∧ Term.val entonces "true"
     si no si Term.cte entonces ExpS1.cod
     si no ExpS1.cod ++ " and "++Term.cod
   ExpSo.cte = (ExpS1.cte ∧ ExpS1.val) ∨
               (Term.cte ∧ Term.val) ∨
               ExpS1.cte ∧ Term.cte
   ExpS1.varscteh = Term.varscteh = ExpSo.varscteh
```

Ampliación de la propuesta: Precálculo de expresiones constantes

Continuación…

```
Term ::= Term and Fact
  Termo.cod =
    si Term1.cte ∧ ¬Term1.val entonces "false"
    si no si Term1.cte ∧ ¬Fact.cte entonces Fact.cod
    si no si Term1.cte ∧ ¬Fact.val entonces "false"
    si no si Term1.cte ∧ Fact.val  entonces "true"
    si no si Fact.cte ∧ ¬Fact.val entonces "false"
    si no si Fact.cte entonces ExpS1.cod
    si no Term1.cod ++ " and "++Fact.cod
  Termo.cte = (Term1.cte ∧ ¬ Term1.val) ∨
              (Fact.cte ∧ ¬ Fact.val) ∨
              Term1.cte ∧ Fact.cte
  Term1.varscteh = Fact.varscteh = Termo.varscteh
```

Ampliación de la propuesta: Precálculo de expresiones constantes

Continuación…

```
Fact ::= iden
    Fact.cte = <iden,_> ∈ Fact.varscteh
    Fact.val = valorDe(iden.lex, Fact.varscteh)
    Fact.cod = iden.lex
Fact ::= true
    Fact.cte = true
    Fact.val = true
    Fact.cod = true
Fact ::= false
    Fact.cte = true
    Fact.val = false
    Fact.cod = "pendiente 3"
```

Ampliación de la propuesta: Precálculo de expresiones constantes

Continuación…

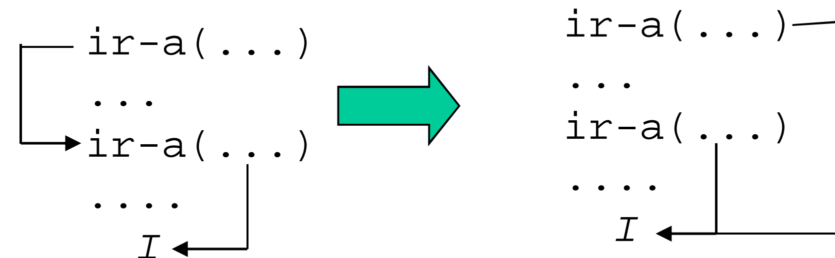
```
fun eval(op,v1,v2)
  si v1 = '?' ∨ v2 = '?' entonces '?'
  si no si op = '+' entonces v1+v2
        si no si op = '*' entonces v1*v2
    ....
ffun

fun eval(op, v)
  si v = '?' entonces '?'
  si no si op = 'not'
           si v=0 entonces 1
           si no entonces 0
     ...
ffun
```

Traducción con sustitución de grupos de instrucciones

- A veces el código objeto generado contiene grupos de instrucciones *inútiles* o *ineficientes* que pueden *eliminarse* o *sustituirse* por otros (= hacer **optimización *peephole***)
  - Existen situaciones triviales de tratar y otras más complejas
- Ejemplos:

```
apila-dir(1)
desapila-dir(1)
```
→ λ




```
ir-a(k)
I
```
→ `ir-a(k)`

(si *k está* fuera del grupo *I* y no hay ningún salto al grupo *I* en todo el programa)

```
apila(1)
multiplica
```
→ λ

Críticas, dudas, sugerencias…

Federico Peinado
www.federicopeinado.es