Procesadores de Lenguaje

Tema 2.1. Ampliación del lenguaje y su procesador: *Instrucciones de control*

Profesor
Federico Peinado

Elaboración del material
José Luis Sierra
Federico Peinado

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

Ampliación del lenguaje y su procesador

- El lenguaje visto hasta ahora es *mínimo*, por lo que será **ampliado** para incluir:
  - Instrucciones de control
  - Tipos construidos
  - Subprogramas
- Los procesadores relacionados con este lenguaje (*compilador* y *máquina virtual*) también **deben ampliarse**, así que aprovechamos para realizar optimizaciones
  - **Optimizaciones a nivel gramatical** , como por ejemplo traducir las expresiones booleanas de manera que se evalúen sus operadores en *circuito corto* (calculando sólo lo imprescindible para conocer el valor de la expresión)

Sentencias compuestas

- Antes de hablar de instrucciones de control hay que entender la estructura de las **sentencias compuestas**

```
SCompuesta ::= begin Sentencias end
```

  - Secuencia de instrucciones que se tratan en bloque como si fueran una sola instrucción
  - Normalmente el bloque debe incluir al menos una sentencia, aunque es práctico permitir bloques *vacíos* de manera provisional mientras un programa está inacabado
  - Mecanismo púramente sintáctico (no cambia la semántica) que permite hacer instrucciones con instrucciones dentro

Ejemplo: Sentencia compuesta en Pascal

- La categoría léxica **;** se comporta como **separador de instrucciones**, siendo el "vacío" una instrucción válida

```
Is ::= Is ; I |
       I
I ::= λ |
      begin Is end |
      IAsig |
      if exp then I |
      ...
```

  - Esta sintaxis permite programas algo "absurdos"

```
begin out(5);;;;;;;;;; end
```

    y que pueden llevar a malas interpretaciones

```
if x > 2 then; out(32)
```

Ejemplo: Sentencia compuesta en C

- La categoría léxica **;** se comporta como **terminador de instrucción**, siendo el bloque vacío **{ }** una posible sentencia compuesta

```
Is ::= Is I |
       I
I ::= IAsig ; |
      IComp |
      ...
IComp ::= {Decs Is} |
          {Decs} |
          {Is} |
          {}
```

Ampliación de la propuesta: Sentencias compuestas

- Usaremos la sintaxis sugerida por Alfred Aho para ampliar el lenguaje propuesto con sentencias compuestas:

```
Is ::= Is ; I |
       I
I ::= begin IsOpc end |
      ...
IsOpc ::= λ |
          Is
```

  - Similar a Pascal pero sin sus problemas, dado que el bloque vacío se trata como caso aparte

Ampliación de la propuesta: Sentencias compuestas

- Añadimos estas nuevas producciones a la gramática incontextual:

```
...
I ::= IComp

IComp ::= begin IsOpc end

IsOpc ::= λ
IsOpc ::= Is
```

Ampliación de la propuesta: Sentencias compuestas

- Y añadimos las ecuaciones semánticas correspondientes a la gramática de atributos:

```
...
I ::= IComp
  IComp.tsh = I.tsh
  I.err = IComp.err
  I.cod = IComp.cod
IComp ::= begin IsOpc end
  IsOpc.tsh = IComp.tsh
  IComp.err = IsOpc.err
  IComp.cod = IsOpc.cod
IsOpc ::= λ
  IsOpc.err = false
  IsOpc.cod = λ
IsOpc ::= Is
  Is.tsh = IsOpc.tsh
  IsOpc.err = Is.err
  IsOpc.cod = Is.cod
```

Sentencias condicionales: if-then-else

- Instrucciones como **if-then-else** en principio tendrían una sintaxis sencilla…

```
...
I ::= IIf
IIf ::= if Exp then I PElse
PElse ::= else I | λ
```

… que en realidad encierra una peligrosa **ambigüedad** que aparece al anidar varias instrucciones:

```
if x > 5 then
  if x > 6 then x := x+1
  else x := x-1
```

- ¡Este **else** se podría interpretar como asociado al *primer* **if-then** o al *segundo*, con resultados bien distintos!

Soluciones al anidamiento de instrucciones if-then-else

1. Marcar el final de cada sentencia **if-then-else:**

```
...
I ::= IIf
IIf ::= if Exp then I PElse fi
PElse ::= else I | λ
```

- La ambigüedad queda eliminada a cambio de escribir más

```
if x > 5 then
  if x > 6 then
    x := x+1
  fi
else
    x := x-1
fi
```

```
if x > 5 then
  if x > 6 then
    x := x+1
  else
    x := x-1
  fi
fi
```

Soluciones al anidamiento de instrucciones if-then-else

2. Estructurar explícitamente los anidamientos **if-then-else:** (`Iife` anida instrucciones donde siempre hay **else** y `Iifne` anida instrucciones donde falta algún **else**)

```
...
I ::= Iife | Iifne
Iife ::= if Exp then Iife else Iife |
         ...
Iifne ::= if Exp then Iife else Iifne |
          if Exp then I
```

- La ambigüedad queda eliminada (*cada **else** asocia siempre con el **if-then** más próximo que esté "libre"*) a cambio de complicar la parte correspondiente de la gramática

Soluciones al anidamiento de instrucciones if-then-else

3. Mantener la ambigüedad en la gramática (!) y resolver el problema a la hora de implementar el procesador del lenguaje (= **mecanismo extragramatical**)

- ¡**Cualquier implementación recursiva** forzará a que cada **else** asocie con el **if-then** "libre" más próximo!
- Podemos dejar *intacta* la implementación que propusimos:

```
fun PElse()
  si token ∈ TkElse
    rec(else);
    I()
  fsi
ffun
```

Ampliación de la propuesta: Instrucciones if-then-else

- Usaremos la tercera solución, asumiendo *por convenio* que la implementación hará que cada **else** asocie con el **if-then** “libre” más próximo
- Añadimos estas nuevas producciones a la gramática incontextual:

```
...
I ::= IIf
IIf ::= if Exp then I PElse
PElse ::= else I | λ
```

Ampliación de la propuesta: Instrucciones if-then-else

- Y añadimos las ecuaciones semánticas correspondientes a la gramática de atributos:

```
...
I ::= IIf
  IIf.tsh = I.tsh
  I.err = IIf.err
IIf ::= if Exp then I PElse
  PELse.tsh = I.tsh = Exp.tsh = IIf.tsh
  IIf.err = (Exp.tipo ≠ bool) ∨ I.err ∨ PElse.err
PElse ::= else I
  I.tsh = PElse.tsh
  PElse.err = I.err
PElse ::= λ
  PElse.err = false
```

Traducción de las instrucciones de control

- Ampliar el lenguaje fuente con instrucciones de control requiere **añadirlas también al lenguaje objeto**
- La máquina virtual admitirá código P con instrucciones de control de bajo nivel que, según el estado de la máquina, cambian de una forma u otra el valor de **CProg** (registro con la dirección de la siguiente instrucción a ejecutar)
  - Típicamente se usan saltos (*gotos*)

Técnicas para representar los saltos en el código P

1. Incluir **etiquetas simbólicas** en el código objeto

- `etiqueta(`*s*`)`
  Inserta la etiqueta simbólica llamada *s*

- `ir-a(`*s*`)`
  Realiza un salto incondicional a la etiqueta *s*

- `ir-f(`*s*`)`
  Desapila el valor de la cima de la pila y si es *falso* (o cero) realiza un salto a la etiqueta *s*
  (en caso contrario la ejecución prosigue con normalidad)

Etiquetas simbólicas en el código objeto

- `etiqueta(s)`

```
CProg ← CProg + 1
```

- `ir-a(s)`

```
temp ← 0
mientras Prog[temp] ≠ "etiqueta(s)"
  temp ← temp + 1
fmientras
CProg ← temp
```

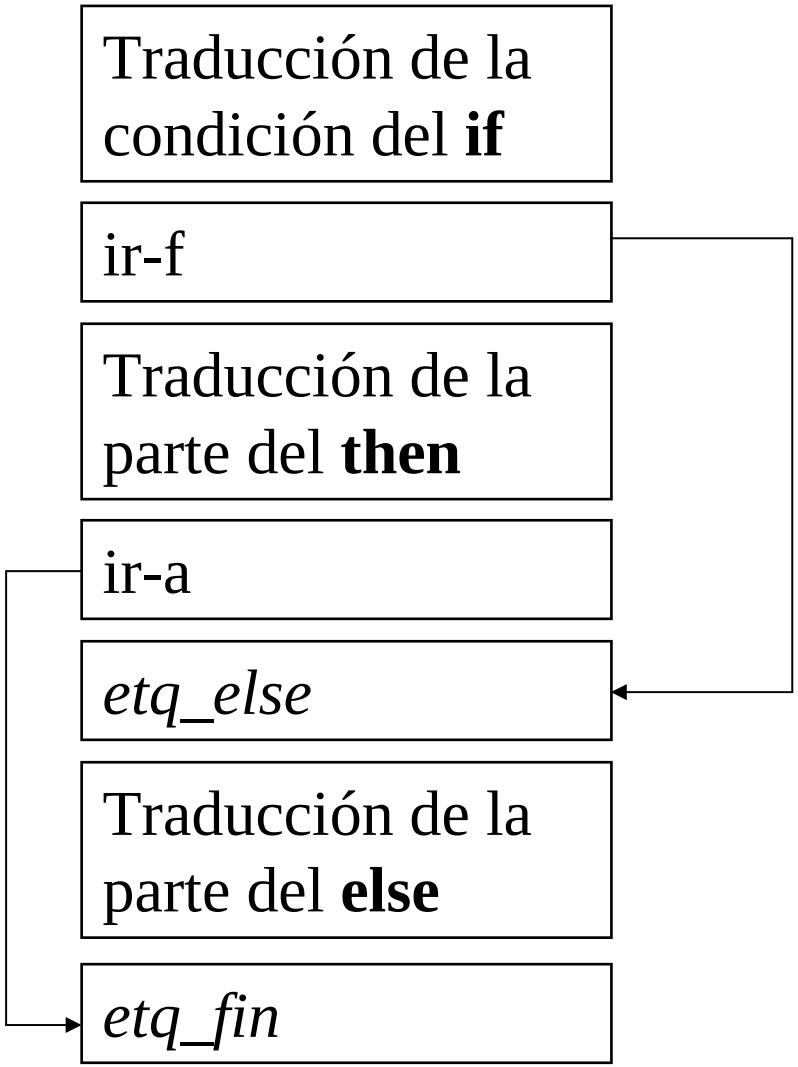
* Búsqueda lineal por el código hasta dar con la instrucción "etiqueta(s)" adecuada

- `ir-f(s)`

```
si Pila[CPila] = 0
  ir-a(s)
si no
  CProg ← CProg + 1
fsi
CPila ← CPila-1
```

Traducción de las instrucciones de control usando etiquetas

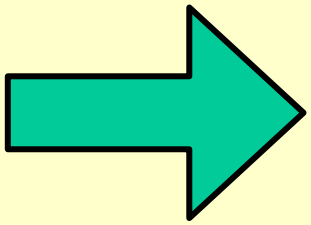
- Esquema para traducir la instrucción **if-then-else**:



Traducción de las instrucciones de control usando etiquetas

- Ejemplo:

```
if x < 5 then
  x := x + 1
else
  x := x - 1;
```



x ≡ Mem[0]

```
...
apila-dir(0)
apila(5)
menor
ir-f(etq_else)
apila-dir(0)
apila(1)
suma
desapila-dir(0)
ir-a(etq_fin)
etiqueta(etq_else)
apila-dir(0)
apila(1)
resta
desapila-dir(0)
etiqueta(etq_fin)
```

Traducción de las instrucciones de control usando etiquetas

- Ecuaciones semánticas de la gramática de atributos:

```
...
I ::= IIf
  IIf.else = nueva_etiqueta;
  IIf.fin = nueva_etiqueta;
  I.cod = IIf.cod
IIf ::= if Exp then I PElse
  IIf.cod = Exp.cod || ir-f(IIF.else) || I.cod ||
            ir-a(IIf.fin)|| etiqueta(IIf.else) ||
            PElse.cod || etiqueta(IIF.fin)
PElse ::= else I
  PElse.cod = I.cod
PElse ::= λ
  PElse.cod = λ
```

*** nueva_etiqueta** hace referencia a la generación de una etiqueta *completamente nueva* (única, que no se repite en ninguna otra parte del código generado) para cada categoría sintáctica que la necesite

Traducción de las instrucciones de control usando etiquetas

- Problema
  - El código objeto obtenido con etiquetas simbólicas **no es eficiente** porque obliga a buscar (de manera lineal sobre las instrucciones) *el punto exacto de destino* en cada salto
- Solución
  - Generar el código objeto **en dos pasadas**:
    1. Generar el código insertando etiquetas simbólicas
    2. Sustituir etiquetas simbólicas por números, haciendo que las instrucciones de salto coloquen directamente en CProg la dirección de la instrucción de destino

… pero en la medida de lo posible **queremos evitar el tener que hacer más de una pasada** para generar el código

Técnicas para representar los saltos en el código P

2. Mantener un **contador de instrucciones** e irlo incrementando con la generación de cada instrucción
   - Dicho contador se usará como “etiqueta” ya que contendrá la dirección de la instrucción de destino para los saltos

- `ir-a(`*`i`*`)`
  Realiza un salto incondicional a la dirección *i* de Prog

- `ir-f(`*`i`*`)`
  Desapila el valor de la cima de la pila y si es *falso* (o cero) realiza un salto a la dirección *i* de Prog
  (en caso contrario la ejecución prosigue con normalidad)

Contador de instrucciones en el traductor

- Las instrucciones de la máquina virtual son más sencillas:

- `ir-a(`*i*`)`

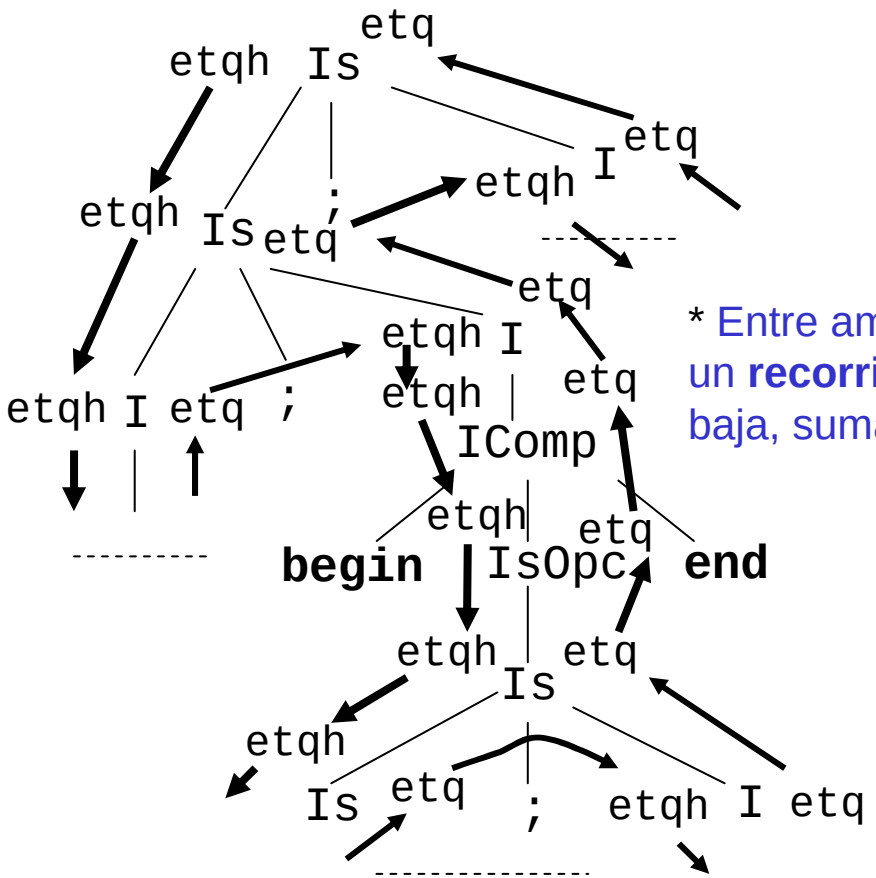
```
CProg ← i
```

- `ir-f(`*i*`)`

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

Contador de instrucciones en el traductor

- Pero en la gramática de atributos habrá que introducir el contador con la ayuda de un par de atributos `etq` y `etqh`:



* Entre ambos atributos es como si hicieran un **recorrido en postorden** del árbol (etqh baja, suma +1 cada instrucción y luego etq sube)

Contador de instrucciones en el traductor

- Ecuaciones semánticas relativas al contador de instrucciones en la gramática de atributos:

```
Prog ::= Decs ; Is
  Is.etqh = 0
Is ::= Is ; I
  Is1.etqh = Iso.etqh
  I.etqh = Is1.etq
  Iso.etq = I.etq
I ::= IComp
  IComp.etqh =  I.etqh
  I.etq = IComp.etq
IComp ::= begin IsOpc end
  IsOpc.etqh =  IComp.etqh
  IComp.etq = IsOpc.etq
IsOpc ::= λ
  IsOpc.etq = IsOpc.etqh
IsOpc ::= Is
  Is.etqh =  IsOpc.etqh
  IsOpc.etq = Is.etq
...
```

Contador de instrucciones en el traductor

Continuación…

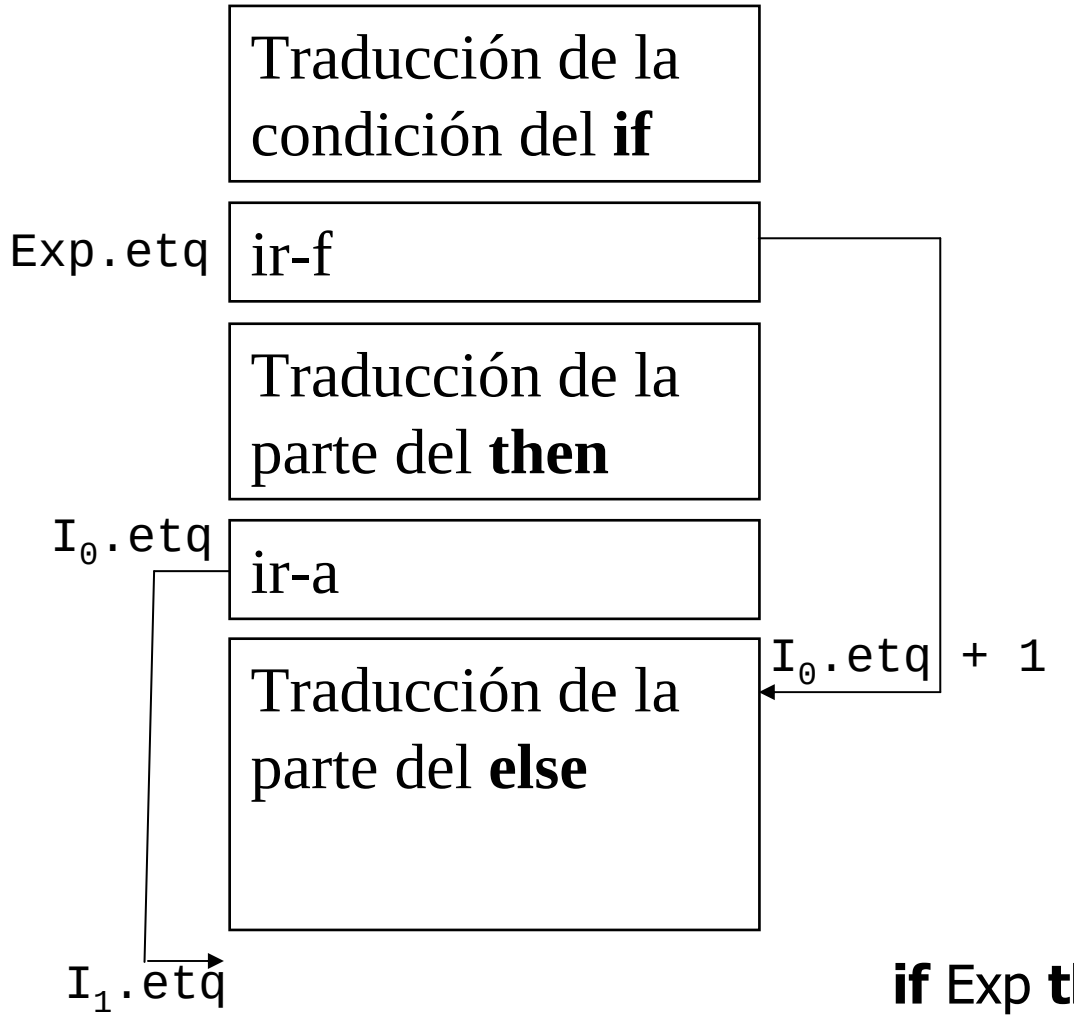
```
...
I ::= IIf
  IIf.etqh = I.etqh
  I.etq = IIf.etq
IAsig ::= iden := Exp
  Exp.etqh = IAsig.etqh
  IAsig.etq = Exp.etq + 1
Exp ::= ExpS OpComp ExpS
  ExpS0.etqh = Exp.etqh
  ExpS1.etqh = ExpS0.etq
  Exp.etq = ExpS1.etq + 1

...
```

- Y así seguiríamos por el resto de la gramática…

Traducción de las instrucciones de control usando un contador

- Esquema para traducir la instrucción **if-then-else**:

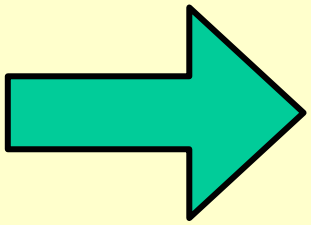


if Exp **then** I_o **else** I_1

Traducción de las instrucciones de control usando un contador

- Ejemplo:

```
if x < 5 then
   x := x + 1
else
   x := x - 1;
```



x ≡ Mem[0]

...

```
apila-dir(0)          0
apila(5)              1
menor                 2
ir-f(9)               3
apila-dir(0)          4
apila(1)              5
suma                  6
desapila-dir(0)       7
ir-a(13)              8
apila-dir(0)          9
apila(1)              10
resta                 11
desapila-dir(0)       12
                      13
```

Traducción de las instrucciones de control usando un contador

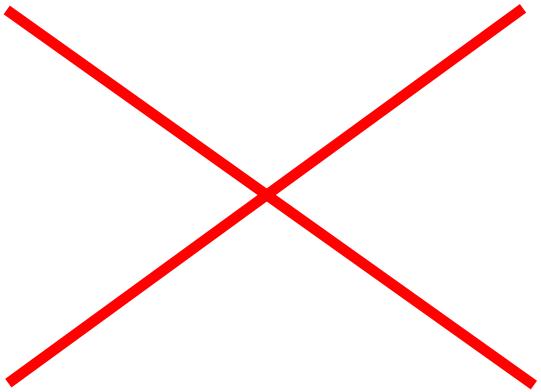
- Ecuaciones semánticas de la gramática de atributos:

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

Traducción de las instrucciones de control con saltos hacia delante

- **Problema**: Al hacer los *esquemas de traducción optimizados* de la ecuación semántica de `IIf.cod`, donde el código se "emite" y almacena globalmente…

```
global cod;
...
IIf(in etqh0; out etq0) ::= if
                            {etqh1 ← etqh0}
                            Exp(in etqh1; out etq1)
                            then
                            {emite(ir-f(etq2 + 1));
                             etqh2 ← etq1 + 1}
                            I(in etqh2; out etq2)
                            {emite(ir-a(etq3));
                             etqh3 ← etq2 + 1}
                            PElse(int etqh3, out etq3)
                            {etq0 ← etq3}
```



… ¡**no podemos generar las instrucciones de salto** porque se desconocen sus **valores de destino**! (sólo se conocen cuando ya se haya generado el código posterior)

Traducción de las instrucciones de control con saltos hacia delante

- **Solución**: Dejar las direcciones de todos los *saltos hacia delante* con **valores indefinidos (?)**, que se sustituirán por los valores correctos una vez sean conocidos

```
global cod;
...
IIf(in etqh0; out etq0) ::= if
                            {etqh1 ← etqh0}
                            Exp(in etqh1; out etq1)
                            then
                            {emite(ir-f(?));
                             etqh2 ← etq1 + 1}
                            I(in etqh2; out etq2)
                            {emite(ir-a(?));
                             parchea(etq1,etq2 + 1);
                             etqh3 ← etq2 + 1}
                           PElse(int etqh3, out etq3)
                           {etq0 ← etq3;
                            parchea(etq2,etq3)}
```

* **parchea(*instSalto*, *destinoSalto*)** modifica la instrucción de salto con dirección *instSalto*, sustituyendo el símbolo **?** de su dirección de destino por *destinoSalto*

… esta técnica se conoce como ***backpatching*** (**parchear** el código previamente generado)

Ampliación de la propuesta: Contador de instrucciones con parcheo

- Nosotros usaremos la técnica del contador de instrucciones parcheando los saltos hacia delante
- Y en los *esquemas de traducción optimizados* el contador de instrucciones `etq` también será una variable global:

```
global cod, etq;
...
IIf::= {var etqaux1, etqaux2}
       if Exp then
       {emite(ir-f(?));
        etqaux1 ← etq;
        etq ← etq + 1}
       I
       {emite(ir-a(?));
        etqaux2 ← etq;
        etq ← etq + 1;
        parchea(etqaux1,etq)}
       PElse
       {parchea(etqaux2,etq)}
```

Sentencias iterativas: while-do

- Instrucciones como **while-do** tienen una sintaxis sencilla y sin problemas de ambigüedad:

```
...
I ::= IWhile
IWhile ::= while Exp do I
```

- La traducción también requiere instrucciones de salto en el código objeto que pueden resolverse usando etiquetas simbólicas o con un contador de instrucciones
- Nosotros también usaremos la técnica del contador de instrucciones parcheando los saltos hacia delante

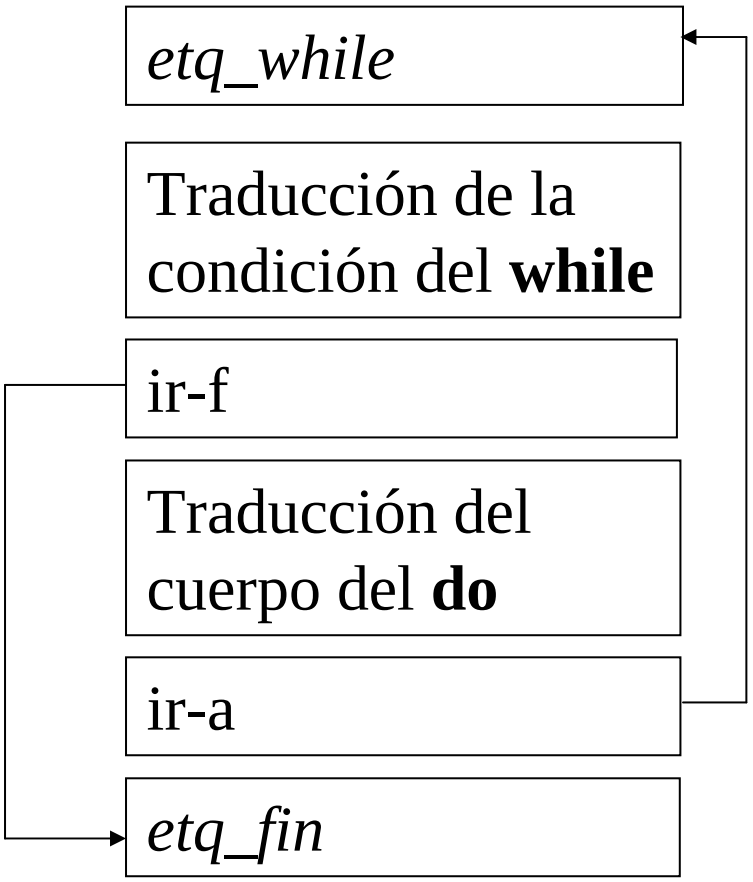
Ampliación de la propuesta: Instrucciones while-do

- Añadimos las producciones anteriores a la gramática incontextual y las ecuaciones semánticas correspondientes a la de atributos:

```
...
I ::= IIf
  IWhile.tsh = I.tsh
  I.err = IWhile.err
IWhile ::= while Exp do I
  I.tsh = Exp.tsh = IWhile.tsh
  IWhile.err = (Exp.tipo ≠ bool) ∨ I.err
```

Traducción de las instrucciones de control usando etiquetas

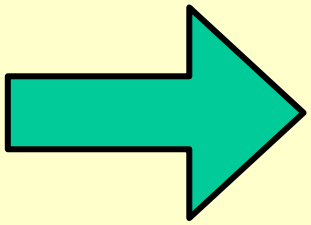
- Esquema para traducir la instrucción **while-do**:



Traducción de las instrucciones de control usando etiquetas

- Ejemplo:

```
while x < 0
begin
   s := x * s;
   x := x - 1
end;
```



x ≡ Mem[0]
s ≡ Mem[1]
...

```
etiqueta(etq_while)
apila-dir(0)
apila(0)
mayor
ir-f(etq_fin)
apila-dir(0)
apila-dir(1)
multiplica
desapila-dir(1)
apila-dir(0)
apila(1)
resta
despila-dir(0)
ir-a(etq_while)
etiqueta(etq_fin)
```

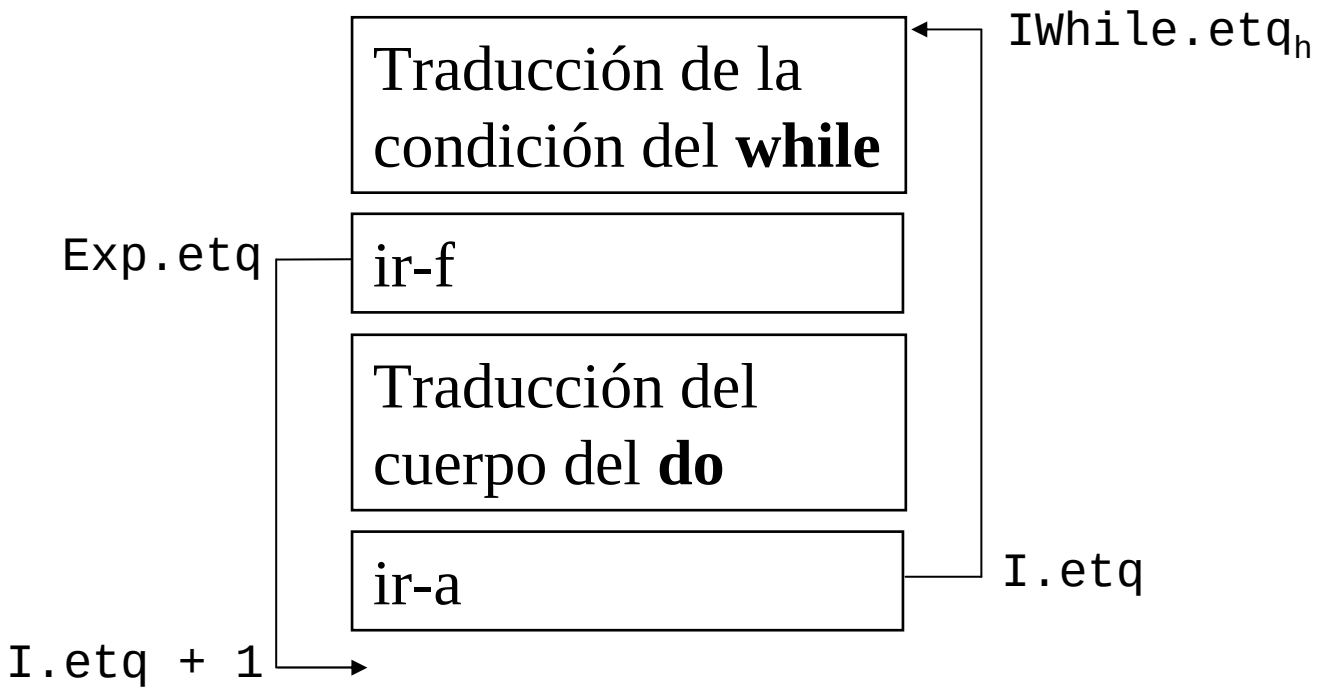
Traducción de las instrucciones de control usando etiquetas

- Ecuaciones semánticas de la gramática de atributos:

```
...
I ::= IWhile
  IWhile.while = nueva_etiqueta;
  IIf.fin = nueva_etiqueta;
  I.cod = IWhile.cod
IWhile ::= while Exp do I
  IWhile.cod = etiqueta(IWhile.while) ||
               Exp.cod ||
               ir-f(IWhile.while) ||
               I.cod ||
               ir-a(IWhile.while)||
               etiqueta(IWhile.fin)
```

Traducción de las instrucciones de control usando un contador

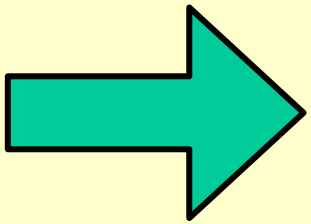
- Esquema para traducir la instrucción **while-do**:



Traducción de las instrucciones de control usando un contador

- Ejemplo:

```
while x < 0
begin
   s := x * s;
   x := x - 1
end;
```



x ≡ Mem[0]
s ≡ Mem[1]
...

```
apila-dir(0)          0
apila(0)              1
mayor                 2
ir-f(13)              3
apila-dir(0)          4
apila-dir(1)          5
multiplica            6
desapila-dir(1)       7
apila-dir(0)          8
apila(1)              9
resta                 10
despila-dir(0)        11
ir-a(0)               12
```

Traducción de las instrucciones de control usando un contador

- Ecuaciones semánticas de la gramática de atributos:

```
...
IWhile ::= while Exp do I
  IWhile.cod = Exp.cod ||
               ir-f(I.etq + 1) ||
               I.cod ||
               ir-a(IWhile.etqh)
  Exp.etqh = IWhile.etqh
  I.etqh = Exp.etq + 1
  IWhile.etq = I.etq + 1
```

Traducción de las instrucciones de control usando un contador

- Esquemas de traducción optimizados de la ecuación semántica de `IWhile.cod` usando parcheo de código:

```
global cod;
...
IWhile(in etqh0; out etq0) ::= while
                               {etqh1 ← etqh0}
                               Exp(in etqh1; out etq1)
                               do
                               {emite(ir-f(?));
                                etqh2 ← etq1 + 1}
                               I(in etqh2; out etq2)
                               {emite(ir-a(etqh0));
                                parchea(etq1,etq2 + 1);
                                etq0 ← etq2 + 1}
```

Ampliación de la propuesta: Contador de instrucciones con parcheo

- También usaremos la técnica del contador de instrucciones parcheando los saltos hacia delante
- Esquemas de traducción optimizados con el contador de instrucciones `etq` como variable global:

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

Críticas, dudas, sugerencias...

Federico Peinado
www.federicopeinado.es