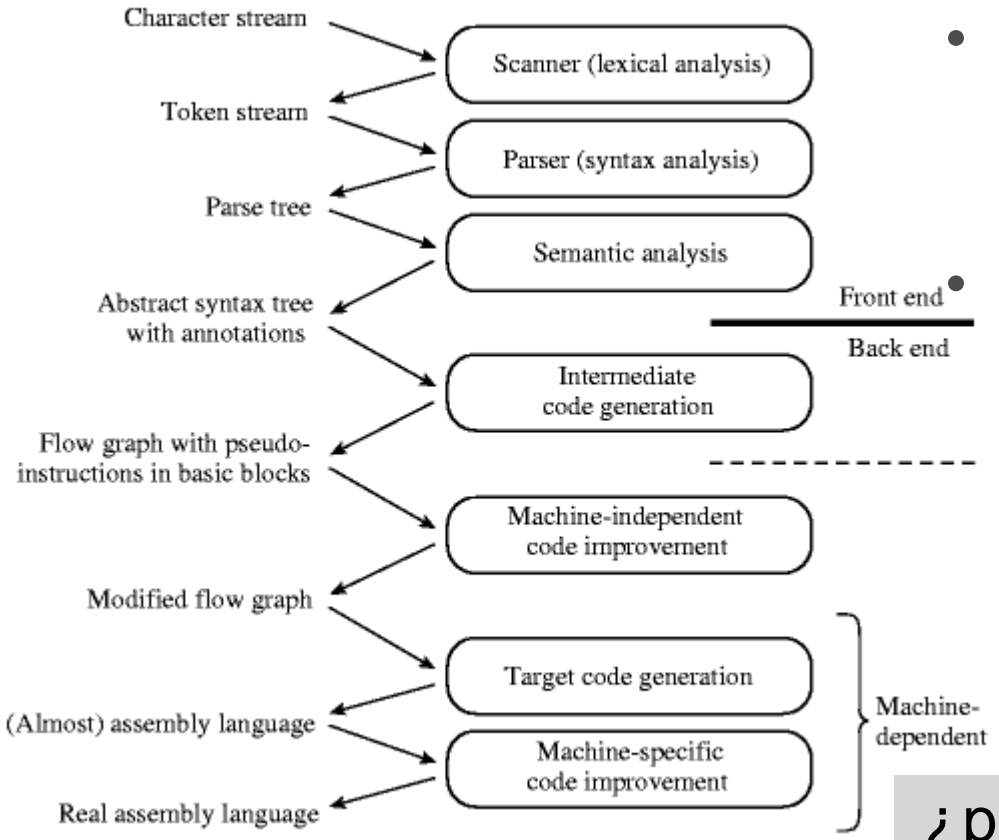
Lección 5:
Generación De Código

1. Introducción
2. Tipos de Código Intermedio
3. Declaraciones
4. Expresiones y Asignación
5. Estructuras de Control
6. Procedimientos y Funciones

Lecturas: Scott, capítulo 9
Muchnick, capítulos 4, 6
Aho, capítulo 8
Fischer, capítulos 10, 11 , 12 , 13, 14
Holub, capítulo 6
Bennett, capítulo 4, 10

1/6. Introducción

- **Código intermedio:** interfaz entre 'front-ends' y 'back-ends'



- Se busca:
  - transportabilidad
  - posibilidades de optimización
- Debe ser:
  - abstracto
  - sencillo
- No se tiene en cuenta:
  - modos de direccionamiento
  - tamaños de datos
  - existencia de registros
  - eficiencia de cada operación

¿por qué no generar el código final directamente?

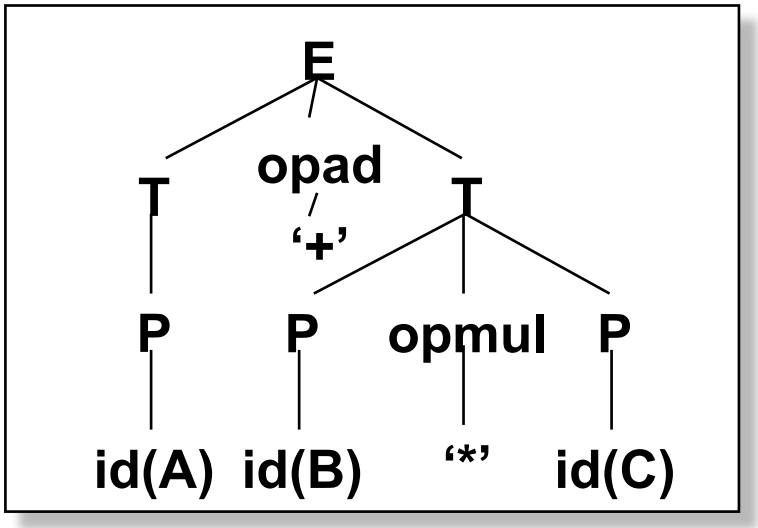
Código Intermedio

- **Ventajas:**
  - Permite ***abstraer la máquina***, separar operaciones de alto nivel de su implementación a bajo nivel.
  - Permite la ***reutilización*** de los front-ends y back-ends.
  - Permite ***optimizaciones generales***.
- **Desventajas:**
  - Implica ***una pasada más*** para el compilador (no se puede utilizar el modelo de una pasada, conceptualmente simple).
  - Dificulta llevar a cabo ***optimizaciones específicas*** de la arquitectura destino.
  - Suele ser ortogonal a la máquina destino, la traducción a una arquitectura específica será más ***larga*** e ***ineficiente***.

2/6. Tipos de Código Intermedio

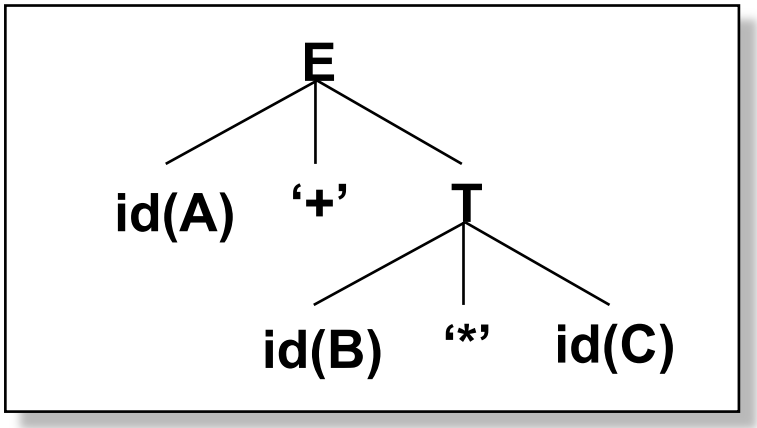
- ***AST (Abstract Syntax Trees):*** forma ***condensada*** de árboles de análisis, con ***sólo nodos semánticos*** y sin nodos para símbolos terminales (se supone que el programa es sintáctica-mente correcto).

Arbol de análisis:



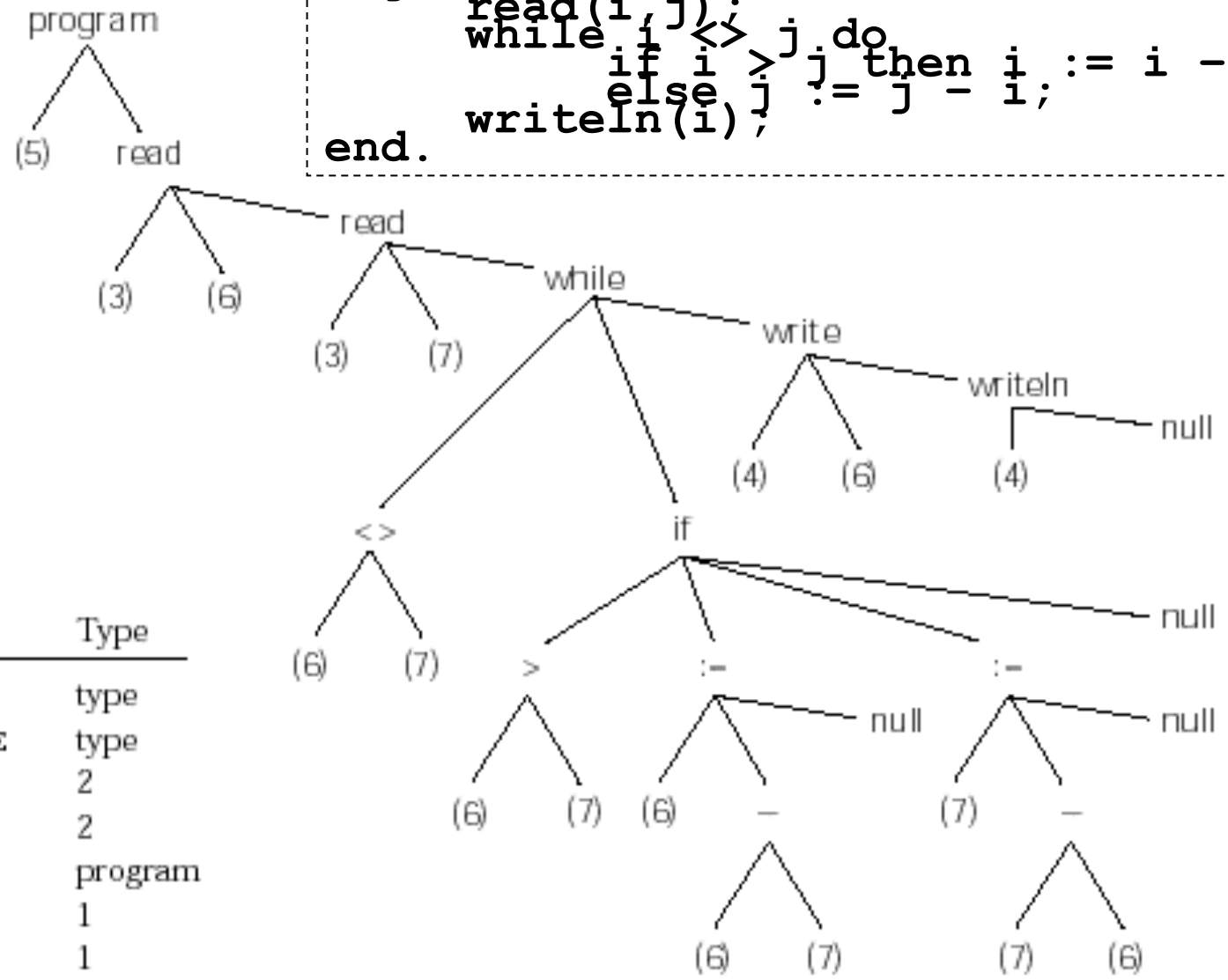
- **Ventajas:** unificación de pasos de compilación
  - Creación del árbol y la tabla de símbolos
  - Análisis semántico
  - Optimización
  - Generación de código objeto
- **Desventaja:**
  - espacio para almacenamiento

AST:



ASTs

```
program gcd(input, output);
var i,j : integer;
begin
      read(i,j);
      while i <> j do
            if i > j then i := i - j
            else j := j - i;
      writeln(i);
end.
```

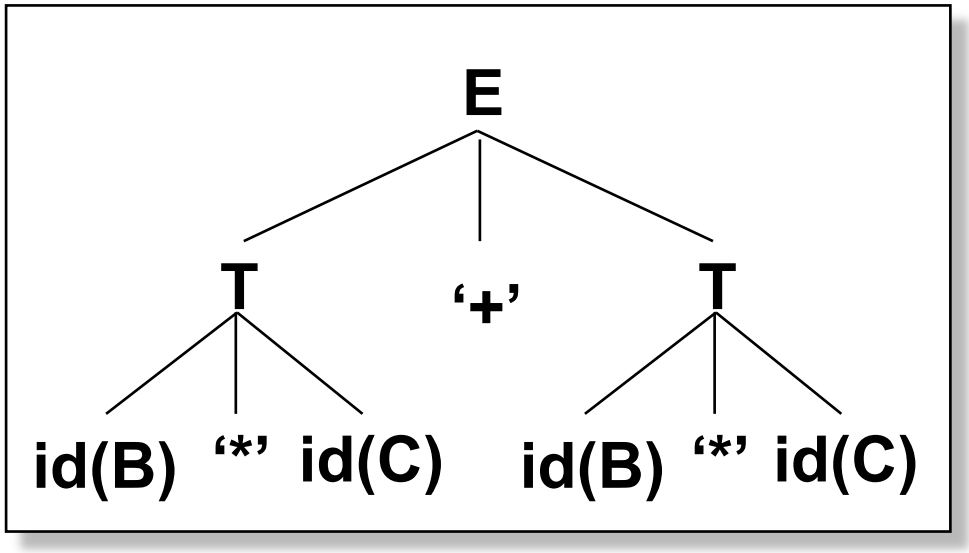


Index	Symbol	Type
1	INTEGER	type
2	TEXTFILE	type
3	INPUT	2
4	OUTPUT	2
5	GCD	program
6	I	1
7	J	1

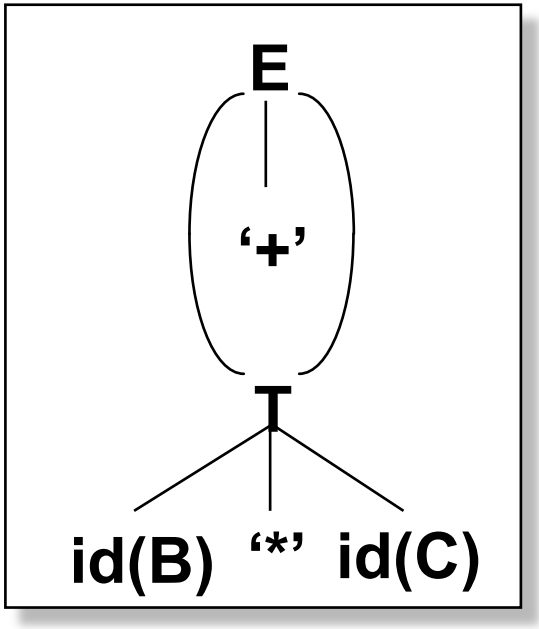
Tipos de Código Intermedio

- **DAG (Directed Acyclic Graphs):** árboles sintácticos concisos

árbol sintáctico:



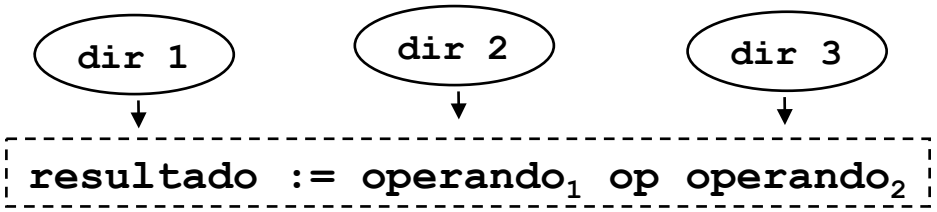
DAG:



- Ahorran algo de espacio
- Resaltan operaciones duplicadas en el código
- Difíciles de construir

Tipos de Código Intermedio

- ***TAC (Three-Address Code):*** secuencia de instrucciones de la forma:



  - operador: aritmético / lógico
  - operandos/resultado: constantes, nombres, temporales.

- Se corresponde con instrucciones del tipo:

```
a := b + c
```

- Las operaciones más complejas requieren varias instrucciones:
- Este 'desenrollado' facilita la optimización y generación de código final.

```
(a + b) x (c – (d / e))
```

```
r1 := a          -- push a
r2 := b          -- push b
r1 := r1 + r2    -- add
r2 := c          -- push c
r3 := d          -- push d
r4 := e          -- push e
r3 := r3 / r4    -- divide
r2 := r2 – r3    -- subtract
r1 := r1 × r2    -- multiply
```

TAC

- Ensamblador general y simplificado para una máquina virtual: incluye etiquetas, instrucciones de flujo de control…
- Incluye referencias explícitas a las direcciones de los resultados intermedios (se les da nombre).
- La utilización de nombres permite la reorganización (hasta cierto punto).
- Algunos compiladores generan este código como código final; se puede interpretar fácilmente (UCSD PCODE, Java).

- Operadores:

	CI	generación	optimización
muchos	**corto**	**compleja**	**compleja**
pocos	**largo**	**sencilla**	**sencilla**

```
     -- first few lines generated during symbol table traversal
     .data          -- begin static data
     .word i        -- reserve one word to hold i
     .word j        -- reserve one word to hold j
     .text          -- begin text (code)
     -- remaining lines accumulated into program.code
main:
     a1 := &input   -- "input" and "output" are file control blocks
                    -- located in a library, to be found by the linker
     call readint   -- "readint", "writeint", and "writeln" are library subroutines
     i := rv
     a1 := &input
     call readint
     j := rv
     goto L1
L2:  r1 := i        -- body of while loop
     r2 := j
     r1 := r1 > r2
     if r1 goto L3
     r1 := j        -- "else" part
     r2 := i
     r1 := r1 - r2
     j := r1
     goto L4
L3:  r1 := i        -- "then" part
     r2 := j
     r1 := r1 - r2
     i := r1
L4:
L1:  r1 := i        -- test terminating condition
     r2 := j
     r1 := r1 <> r2
     if r1 goto L2
     a1 := &output
     r1 := i
     a2 := r1
     call writeint
     a1 := &output
     call writeln
     goto exit      -- return to operating system
```

TAC

- Variaciones sintácticas:

(Fischer)

```
(READI, A)
(READI, B)
(GT, A, B, t1)
(JUMP0, t1, L1)
(ADDI, A, 5, C)
(JUMP, L2)
(LABEL, L1)
(ADDI, B, 5, C)
(LABEL, L2)
(SUBI,C, 1, t2)
(MULTI, 2, t2, t3)
(WRITEI, t3)
```

(Aho)

1. **Asignación:**

```
x := y op z
x := op y
x := y
x[i] := y
x := y[i]
x := &y
x := *y
*x := y
```

2. **Saltos:**

```
goto L
if x oprel y goto L
```

3. **Procedimientos:**

```
param x1
...
param xn
call p, n
```

Representaciones de TAC

Fuente:

```
a := b * c + b * d;
```

- **Cuádruplas**: el destino suele ser una temporal.

```
(*, b, c, t1)
(*, b, d, t2)
(+, t1, t2, a)
```

- **Tripletas:** sólo se representan los operandos

```
(1)  (*, b, c)
(2)  (*, b, d)
(3)  (+, (1), (2))
(4)  (:=, (3), a)
```

- **Supuesta ventaja:** es más concisa.

Falso: estadísticamente se requieren más instrucciones.

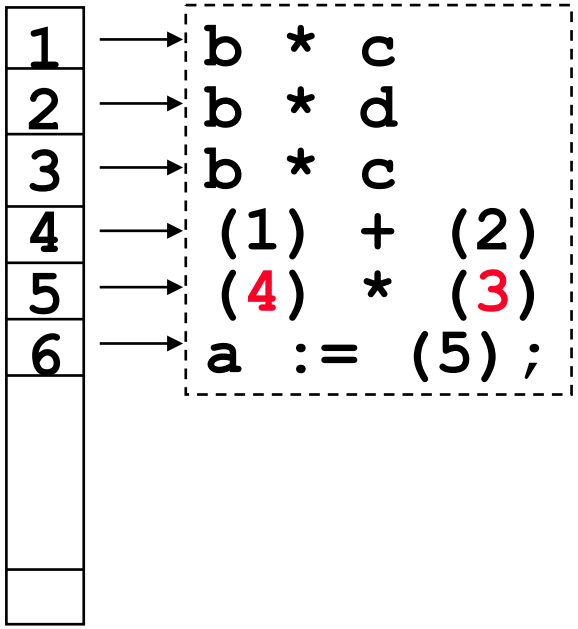
- **Desventaja:** el código generado es dependiente de la posición.

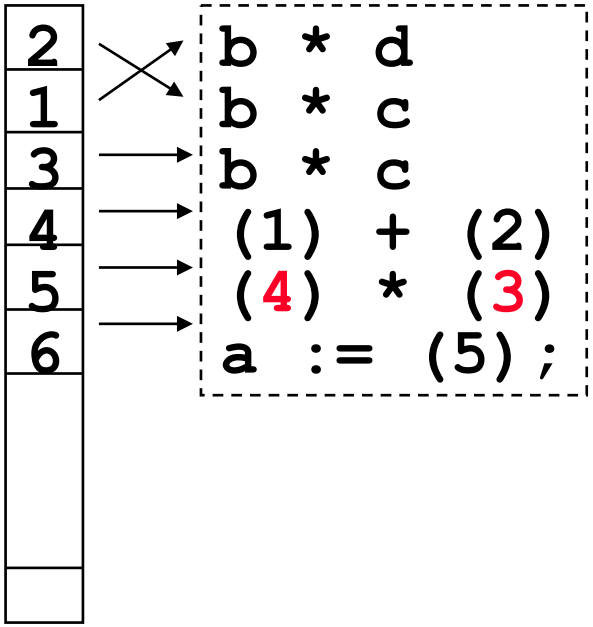
Cierto: la dependencia posicional dificulta la optimización.

Representaciones de TAC

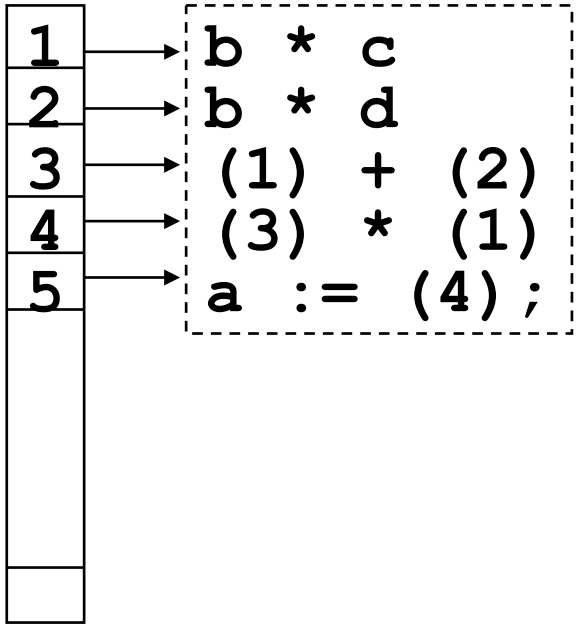
- Tripletas Indirectas: + vector que indica el orden de ejecución de las instrucciones.

```
a := (b * c + b * d) * b * c;
```





Reorganizar es eficiente



Se puede compartir espacio

RTL: register transfer language

```
d = (a + b) * c;
```

```
(insn 8 6 10 (set (reg:SI 2)
        (mem:SI (symbol_ref:SI ("a")))))

(insn 10 8 12 (set (reg:SI 3)
        (mem:SI (symbol_ref:SI ("b")))))

(insn 12 10 14 (set (reg:SI 2)
        (plus:SI (reg:SI 2)
            (reg:SI 3))))

(insn 14 12 15 (set (reg:SI 3)
        (mem:SI (symbol_ref:SI ("c")))))

(insn 15 14 17 (set (reg:SI 2)
        (mult:SI (reg:SI 2)
            (reg:SI 3))))

(insn 17 15 19 (set (mem:SI (symbol_ref:SI ("d")))
        (reg:SI 2)))
```

Notación posfija

- Polaca inversa, código de cero direcciones:
  - notación matemática libre de paréntesis
  - los operadores aparecen después de los operandos

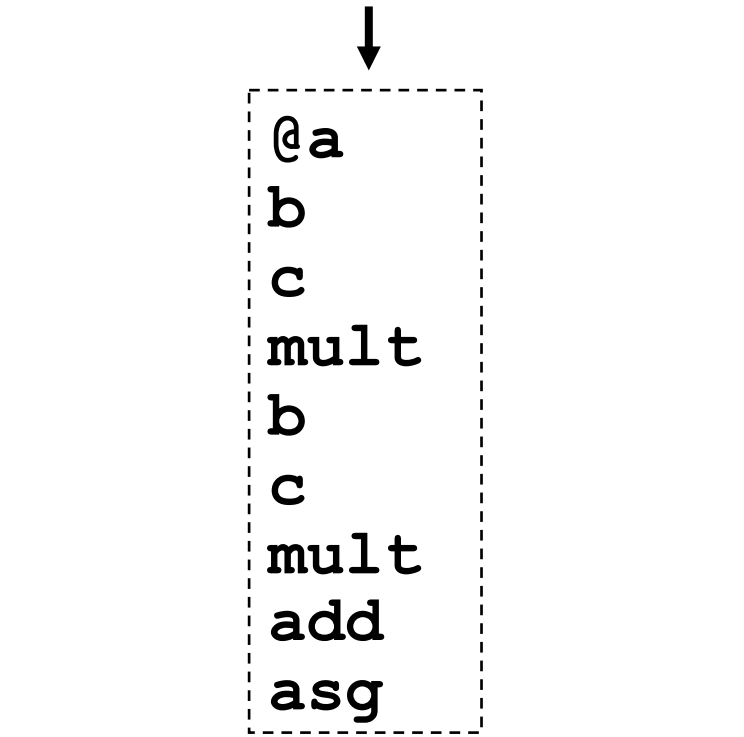
Expresiones:

1. E átomo $\Rightarrow$ E ’
2. E_1 op E_2 $\Rightarrow$ E_1’ E_2’ op
3. (E) $\Rightarrow$ E’

Asignación:

id := E $\Rightarrow$ @id E’ asg

```
a := b * c + b * c
```



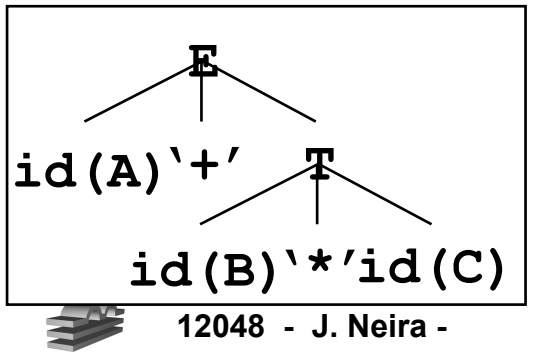
Notación Postfija

- **Ventajas:**
  - Código generado conciso.
  - No hacen falta temporales.
  - Algunas optimizaciones sencillas.
  - Mantiene la estructura sintáctica.
- **Desventajas:**
  - Código dependiente de la posición.
  - solo efectiva si el 'target' es de pila.

```
a := b * c + b * c
```

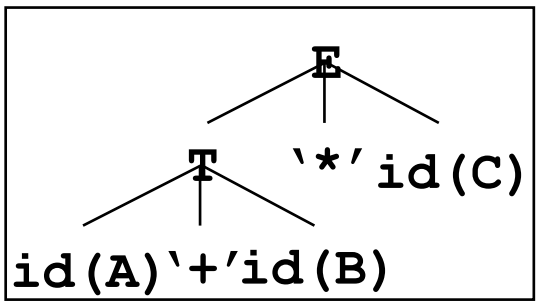
```
@a b c mult dup add asg
```

```
a + b * c
```




```
a
b
c
mult
add
```

```
(a + b) * c
```




```
a
b
add
c
mult
```

Generación de código intermedio

- **Consideración fundamental:** generación de código correcto.

```
a := b * c
```

```
; Dirección de A.
   SRF  0  5
; Acceso a B.
   SRF  0  4
   DRF
; Acceso a C.
   SRF  0  3
   DRF
   TMS
; Asignación.
   ASG
```

INEFICIENCIA EN TIEMPO:

¿hay alguna forma menos costosa de hacer la multiplicación?

INEFICIENCIA EN ESPACIO:

¿está b * c calculado en algún sitio?

- Sin hacerse estas preguntas, es razonablemente sencillo.

3/6. Procesamiento de declaraciones

Esencialmente se trata de completar la tabla de símbolos

- no se genera código (hay excepciones).
- se limita a resolver problemas relacionados con almacenamiento de los objetos:
  - Espacio requerido
  - Lugar en la memoria
  - Valor
- hay información explícita e implícita en el fuente.

- Declaración de variables:

```
int sig, nivel = 0;

void abrir_bloque()
{
   ... sig = INICIAL;
   ++nivel;
}

void cerrar_bloque()
{
   ... nivel--;
}

void crear_var (char *nom,
                int tipo)
{
   .....
   simbolo->dir = sig;

   switch (tipo) {
     case ENTERO :
          sig += 2; break;
     case REAL   :
          sig += 4; break;
     ....
   }
}
```

Al final, **sig** indica el tamaño del bloque de activación.

Ejemplo

```
Programa p;
   entero i, j, m;
            3   4   5
   accion dato (ref entero d);
   Principio                    3
   Fin
                                3   4                   5
   accion mcd(Val entero a, b; ref entero m);
   entero r; 6

   Principio
   Fin

Principio
Fin
```

Declaración de Variables

- ¿Y si se permite mezclar declaraciones de variables y procedimientos?

```
procedure P;
 var  i,j,k : intege:
 procedure Q;
  var l,m : integer;
  begin
     ....
  end;
 var n : integer;
```

```
i.dir = 0
j.dir = 2
k.dir = 4
```

```
l.dir = 0
m.dir = 2
```

```
n.dir = 4
 (¡igual
 que k!)
```

- ¿Y si necesitas mantener información sobre el tamaño de cada bloque?

- C: este problema no existe.
- Pascal: muchos compiladores lo prohíben.

Posible solución:

```
int sig[MAXANID], nivel = 0;

void abrir_bloque ()
{... sig[++nivel] = INICIAL;}

void cerrar_bloque ()
{    ... nivel--;}

void crear_var (char *nom,
                int tipo)
{   ....
    simbolo->dir = sig[nivel];
    ...
}
```

Inicializadores, C:

- **extern y static:**
  - Valor inicial por defecto: 0.
  - Admiten sólo valores constantes.
  - El inicializador se ejecuta una sola vez.

```
#include <stdio.h>
     /*  ¿legal?   */
int i = 1,
     /*  ¿legal?   */
     j = i + 1,
     /*  ¿legal?   */
     m = 34 + 1,
     f(),
     /*  ¿legal?   */
     k = f(),
     /*  ¿legal?   */
     l;

     /*  ¿legal?   */
extern int n = 0;
```

- **auto y struct:**
  - Valor inicial por defecto: ninguno.
  - Admiten valores no constantes.
  - Se ejecutan c/vez que se activa el bloque.

```
int f()
{
     /*  ¿legal?   */
int i = 1,
     /*  ¿legal?   */
     j = i + 1,
     g(),
     /*  ¿legal?   */
     k = g(i),
     /*  ¿legal?   */
     n = f(),
     l;

     /*  ¿legal?   */
  static int m = 1;
  .....
}
```

Esquema general, secuencial

```
%union {...
   char *cadena;...
}
...
```

```
programa p;
...
principio
...
fin
```

```
   ENP L0
   ...
; Comienzo de p
L0:
   ...
; Fin de p
   LVP
```

```
programa: tPROGRAMA tIDENTIFICADOR ';'
{
    inicio_generacion_codigo ();
    $<cadena>$ = nueva_etiqueta ();
    generar (ENP, $<cadena>$);
}
    declaracion_variables
    declaracion_acciones
{
    comentario (sprintf(msg,
        "Comienzo de %s", $2.nombre));
    etiqueta($<cadena>4);
}
    bloque_instrucciones
{
    comentario (sprintf(msg,
        "Fin de %s", $2.nombre));
    generar (LVP);
    fin_generacion_codigo();
}
```

Esquema general, AST

```
%union {...
  LISTA cod;
}
...
%type <cod> bloque_instrucciones
%type <cod> declaracion_acciones
```

```
programa p;
...
principio
...
fin
```

```
  ENP L0
  ...
; Comienzo de p
L0:
  ...
; Fin de p
  LVP
```

```
programa: tPROGRAMA tIDENTIFICADOR ';'
          declaracion_variables
          declaracion_acciones
          bloque_instrucciones
{
    char *lenp = newlabel(), msg[100];

    $$ = code (ENP, lenp);
    concatenar (&($$), $5);
    sprintf(msg, "Comienzo de %s",...);
    comment (&($$), msg);
    label(&($$), lenp);
    concatenar (&($$), $6);
    sprintf(msg, "Fin de %s",...);
    comment (&($$), msg);
    emit (&($$), LVP);
    dumpcode ($$, fich_sal);
}
```

4/6. Expresiones y Asignación

```
x[i, j] := a*b + c*d - e*f + 10;
```

- **De los operandos:**
  - Determinar su localización
  - Efectuar conversiones implícitas
- **De los operadores:**
  - Respetar su precedencia
  - Respetar su asociatividad
  - Respetar orden de evaluación (*si definido)*
  - Determinar interpretación correcta (*si sobrecargado)*
- **Instrucciones generadas**
  - Acceso a datos
    - » Variables simples
    - » Registros
    - » Vectores
  - Manipulación de datos
    - » Conversiones
    - » Operaciones
  - Flujo de control
    - » Validación de subrangos
    - » Operadores complejos

Algunas dificultades

- Orden de evaluación de los operandos:

```
push(pop() - pop());
```

¿Implementa correctamente SBT?
¿Es relevante en el caso de PLUS?

```
a[i] = i++;
```

¿el valor del subíndice es el anterior o actual valor de i?

```
FRAMES[pop()] = pop();
```

¿Corresponde a `ASG` o `ASGI`?

- Método de evaluación de los operadores:

```
push (pop() and pop());
```

¿Implementa `AND` correctamente en PASCAL?

```
push (pop() && pop());
```

¿Implementa `AND` correctamente en C?

En C, ni la suma ni la asignación tienen predefinido un orden de evaluación.

En C, el operador && se evalúa por corto circuito y por lo tanto de izquierda a derecha.

Operandos: acceso a nombres (sec)

- La información esencial debe obtenerse de la tabla de símbolos.

```
constante :
    TTRUE
    {
     generar (STC, 1);
    }
  | TFALSE
    {
     generar (STC, 0);
    }
  | TENTERO
    {
     generar (STC, $1);
    }
  | TCARACTER
    {
     generar (STC, $1);
    }
;
```

```
factor :
    TIDENT
    {
     generar (SRF, ?, ?);
     generar (DRF);
     /* ¿parametro? */
    }
  | TIDENT
    {
    generar (SRF, ?, ?);
    /* ¿parametro? */
    }
    '[' expresion ']'
    {
    /* ?tamano? */
    generar (DRF);
    }
  | TIDENT '(' args ')'
    {
    generar (OSF, ?, ?, ?);
    }
     .....
;
```

Operandos: acceso a nombres (AST)

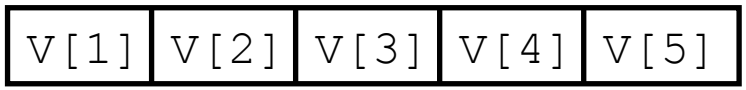
- La información esencial debe obtenerse de la tabla de símbolos.

```
constante :
   TTRUE
   {
   $$.cod = code (STC, 1);
   }
 | TFALSE
   {
   $$.cod = code (STC, 0);
   }
 | TENTERO
   {
   $$.cod = code (STC, $1);
   }
 | TCARACTER
   {
   $$.cod = code (STC, $1);
   }
;
```

```
factor :
   TIDENT
   {
   $$.cod = code (SRF,?,?);
   emit (&($$.cod), DRF);
   /* ¿parametro? */
   }
 | TIDENT '[' expresion ']'
   {
   $$.cod = code (SRF, ?, ?);
   /* ¿parametro? */
   concatenar (&($$.cod),
                $3.cod);
   /* ?tamano? */
   emit (&($$), (DRF);
   }
 | TIDENT '(' args ')'
   {
   $$.cod = $3;
   emit (&($$.cod),OSF,?,?,?);
   }
   .....
;
```

Vectores

- Componentes almacenadas de forma contigua



- Vectores de dimensión definida en compilación, desplazamiento de `V[i]`:

```
(i – lim_inf) x tamaño
```

- En C:

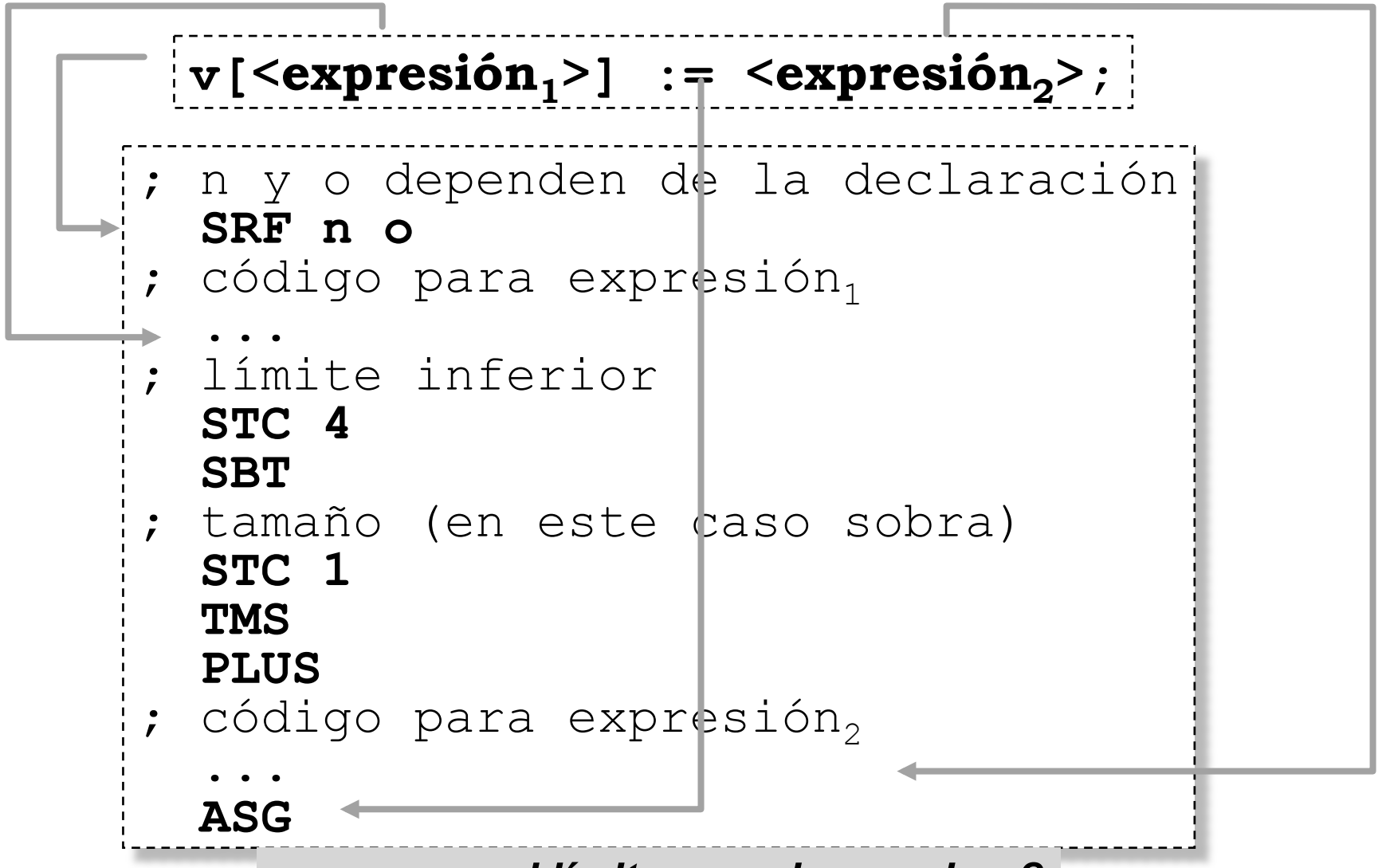
```
i x tamaño
```

- se reduce a aritmética de punteros:

`v[i]`
es equivalente a
`*(v + i)`

Procesamiento de Vectores

- Dada `var v : array[4..8] of integer;`



¿o sea que el límite superior no sirve?

Procesamiento de Vectores

- Dada `var v : array[-3..5] of boolean;`

```
if v[<expresión>] then ...;

; n y o dependen de la declaración
   SRF n o
; código para expresión
   ...
   ??
   ??
   STC -3
   GTE
   JMF ... ; procedimiento de error
   STC 5
   LTE
   JMF ... ; procedimiento de error
; límite inferior
   STC -3
   SBT
; tamaño
   STC 1
   TMS
   PLUS
   DRF
   JMF ...
```

¿qué hacer aquí?

Matrices contiguas por filas, v1

tipo v[$l_1..u_1, \ldots, l_n..u_n$];

dir de v[$e_1, \ldots, e_n$]?

```
entero v[2..5,3..7,1..8];

....
v[2,3,1] := ...;
v[5,7,8] := ...;
v[3,4,5] := ...;
```

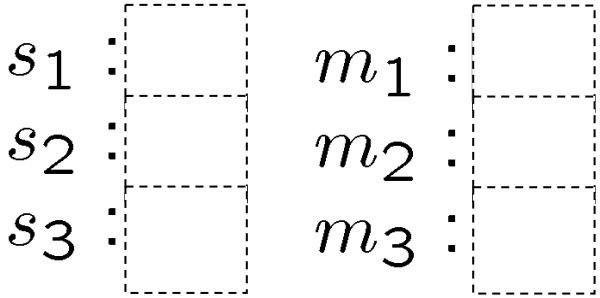
¿dirección?

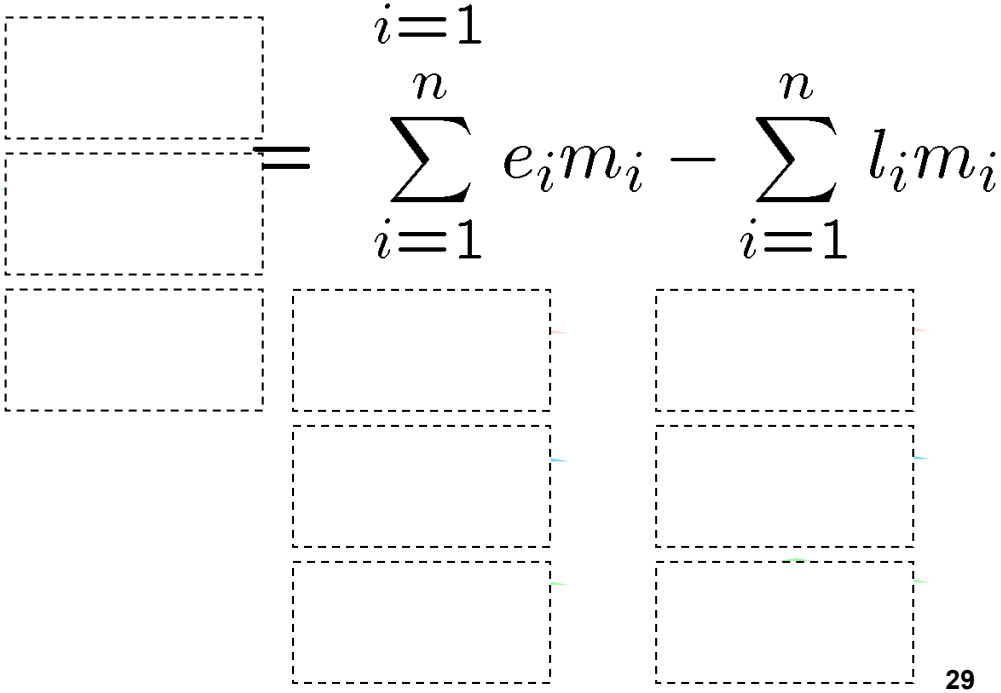
$$s_j = (u_j - l_j + 1)$$

$$m_i = \prod_{j=i+1}^{n} s_j$$

$$d_n = \sum_{i=1}^{n} (e_i - l_i) m_i$$

$$= \sum_{i=1}^{n} e_i m_i - \sum_{i=1}^{n} l_i m_i$$





Formulación alternativa, v2

$$d_n = (\cdots((e_1 s_2 + e_2)s_3 + e_3)\cdots)s_n + e_n$$
$$- (\cdots((l_1 s_2 + l_2)s_3 + l_3)\cdots)s_n + l_n$$

$$c_1 = e_1$$
$$c_i = c_{i-1}s_i + e_i$$
$$d_n = c_n - c$$

```
entero v[2..5,3..7,1..8];

....
v[2,3,1] := ...;
v[5,7,8] := ...;
v[3,4,5] := ...;
```

¿dirección?

s_1 :
s_2 :
s_3 :

c_1 :
c_2 :
c_3 :
d_3 :

Matrices: generación de código, v2

```
factor : TIDENT
         {
          int nivel = ..., /* nivel sint. */
          int offset = ....; /* dir en BA */

          generar (SRF, nivel, offset);
          /* ¿par x ref? */
         }
  c_n    '[' lista_indices ']'
         {
          int t = ...; /* Tamaño elems */
          int c = ...; /* Parte constante */

  d_n     generar (STC, c);
          generar (SBT);
          generar (STC, t);
addr      generar (TMS);
          generar (PLUS);
          generar (DRF);
         }
;
```

Matrices: generación de código, v2

```
%union { ..... int i; ..... }
%type <i> lista_indices
%%
lista_indices: expresion
               {
                 $$ = 1;
               }
             | lista_indices ',' expresion
               {
                 int s_i, i;

                 i = $1 + 1;
                 s_i = ...;
                 generar (STC, s_i);
                 generar (TMS);
               }
               expresion
               {
                 generar (PLUS);
                 $$ = $1 + 1;
               }
;
```

e_1

$c_1 = e_1$

c_{i-1}

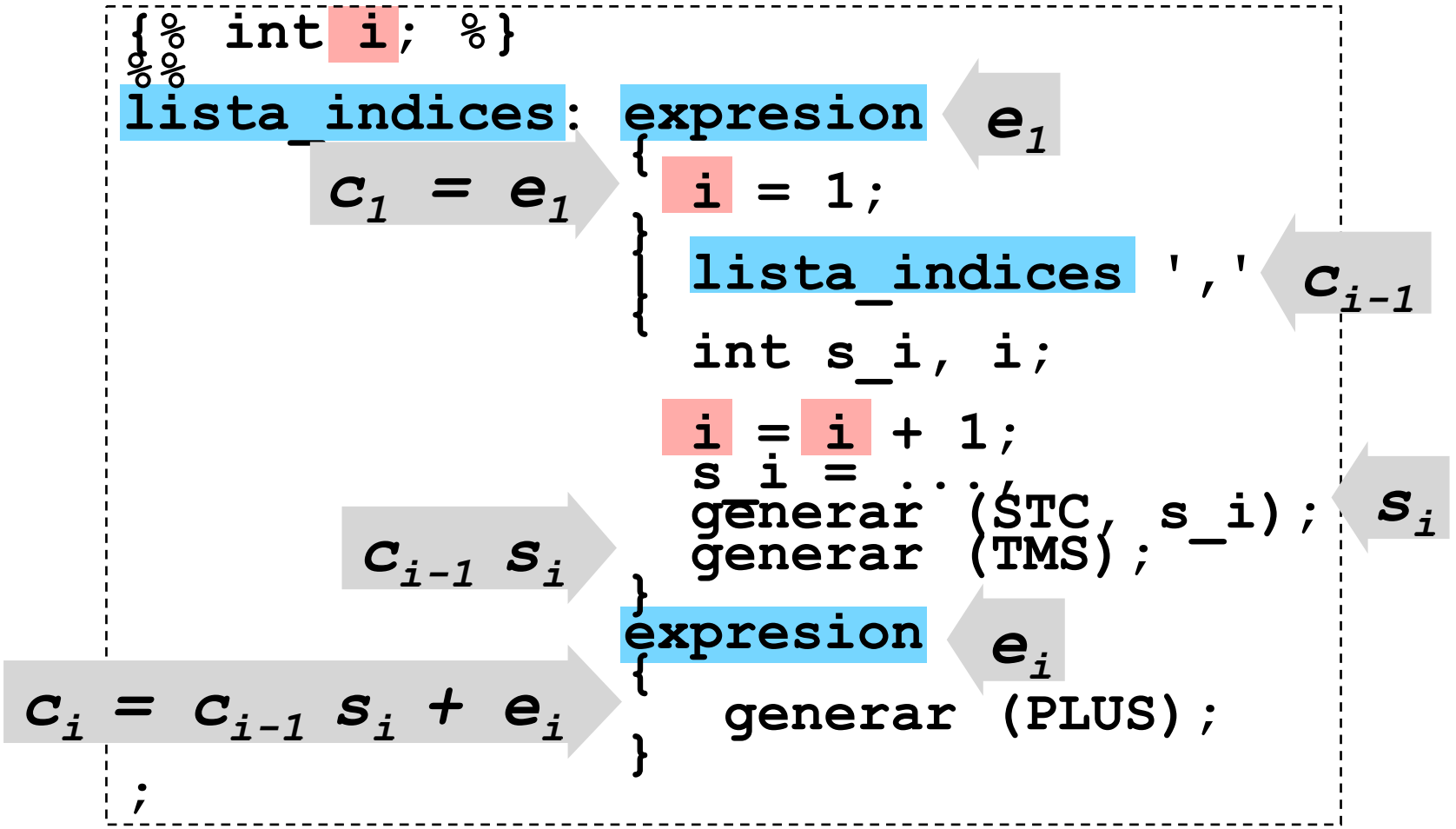
s_i

$c_{i-1}\ s_i$

e_i

$c_i = c_{i-1}\ s_i + e_i$

¿y porqué no...?




```
entero v[1..10], w[1..3, 1..4, 1..5, 1..6];

ẇ[v̇[1], v[2], v[3], v[4]] := ...;
```

Ejemplo: para `v[3,4,5]`

v1

```
; v
          SRF n o
; e_1
          STC 3
; m_1
          STC 40
          TMS
          PLUS
; e_2
          STC 4
; m_2
          STC 8
          TMS
          PLUS
; e_3
          STC 5
; m_3
          STC 1
          TMS
          PLUS
; fin de los indices
; c
          STC 105
          SBT
```

v2

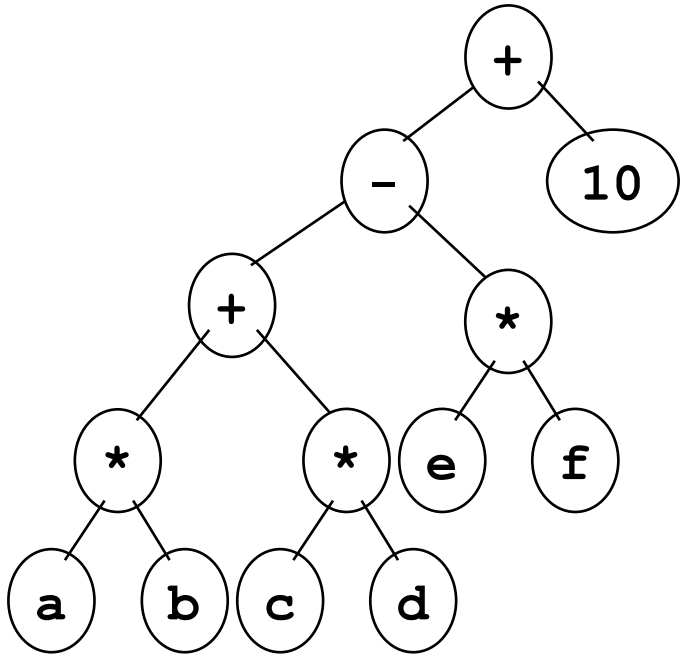
```
; v
          SRF n o
; e_1
          STC 3
; s_ 2
          STC 5
          TMS
; e_2
          STC 4
          PLUS
; s_3
          STC 8
          TMS
; e_3
          STC 5
          PLUS
; fin de los indices
; c
          STC 105
          SBT
          PLUS
```

¿cuál es más eficiente?

Operadores

- Algoritmo recursivo: comenzando en la raíz del árbol sintáctico:

```
a*b + c*d - e*f + 10
```




```
a b * c d * + e f * - 10 +
```

Para un operador n-ario:

1. Generar código para evaluar los operandos 1..n, almacenando los resultados en localizaciones temporales-
2. Generar código para evaluar el operador, utilizando los operandos almacenados en las n localizaciones temporales y almacenando el resultado en otra localización temporal.

Operadores Aritméticos (sec)

```
%union { ... int instr;  ... }
%type <instr> op_ad op_mul
%%
expr_simple : termino
            | '+' termino
            | '-' termino
              { generar (NGI); }
            | expr_simple op_ad termino
              { generar ($2); }
;
op_ad : '+' { $$ = PLUS; }
      | '-' { $$ = SBT; }
;
termino : factor
        | termino op_mul factor
          { generar ($2); }
;
op_mul : '*'  { $$ = TMS; }
       | TDIV { $$ = DIV; }
       | TMOD { $$ = MOD; }
;
```

Los operadores lógicos se tratan de forma similar, excepto....

Corto circuito: or else

- Implica operaciones de control de flujo:

A **or else** *B* → **if** *A* **then** *true* **else** *B*

```
     A
     JMT T
     B
     JMP Fin
T:   STC 1
Fin:
```

```
     A
     JMF F
     STC 1
     JMP Fin
F:   B
Fin:
```

A	v	v	f	f
B	v	f	v	f
Instr.	a+2	a+2	a+b +2	a+b +2

A	v	v	f	f
B	v	f	v	f
Instr.	a+3	a+3	a+b +1	a+b +1

Corto circuito: and then

A **and then** *B* → **if** *A* **then** *B* **else** *false*

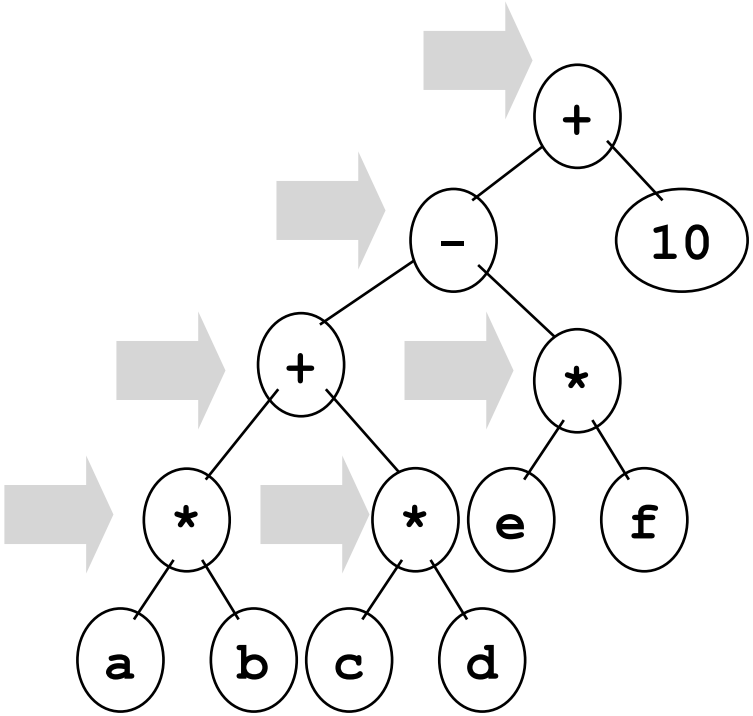
A	v	v	f	f
B	v	f	v	f
Instr.				

A	v	v	f	f
B	v	f	v	f
Instr.				

Siguiente problema....

- ¿qué hacer con los resultados intermedios?

```
a*1b +1 c*2d - e*3f +2 10
```



Depende del tipo de código intermedio

- Máquinas de pila:

```
;variable A
    SRF   0   3   DRF
;variable B
    SRF   0   4   DRF
;*1: A * B
    TMS
;variable C
    SRF   0   5   DRF
;variable D
    SRF   0   6   DRF
;*2:C * D
    TMS
;+1:(A*B)+(C*D)
    PLUS
;variable E
    SRF   0   7   DRF
;variable F
    SRF   0   8   DRF
;*3:E * F
    TMS
;-:((A*B)+(C*D))-(E*F)
    SBT
    STC 10
;+2:(((A*B)+(C*D))-(E*F))+10
    PLUS
```

TAC: Variables temporales

- Se supone disponible una cantidad ilimitada de variables

```
a*1b +1 c*2d - e*3f +2 10
```



```
t0 := A * B;
t1 := C * D;
t2 := t0 + t1;
t3 := E * F;
t4 := t2 - t3;
t5 := t4 + 10;
```

‘register allocation problem’

- Al traducir este código intermedio para una arquitectura destino, se utilizarán registros en lo posible, de lo contrario posiciones de memoria.

Código de tres direcciones

- Se utiliza un generador de nombres temporales:

```
char *newtemp ()
{
    static int c = 0;
    char *m = malloc (5);

    sprintf (m, "t%d", c++);
    return m;
}
```

- Las expresiones tienen como atributo el nombre de la variable en la que quedan almacenadas.

```
%{
extern char *newtemp();
%}

%union {...char *place;...}

%type <place> TIDENT expresion
%type <place> expresion_simple termino factor
```

Código de tres direcciones

```
expresion : simple { strcpy ($$, $1); }
          ;
simple : termino { strcpy ($$, $1); }
       | '+' termino { strcpy ($$, $2);}
       | '-' termino
       {
         strcpy ($$, newtemp());
         tac ("%s := -%s;\n",$$, $2);
       }
       | simple '+' termino
       {
         strcpy ($$, newtemp());
         tac("%s := %s + %s;\n",$$, $1, $3);
       }
       | simple '-' termino
       {
         strcpy ($$, newtemp());
         tac("%s := %s - %s;\n",$$, $1, $3);
       }
;
factor : TIDENT { strcpy ($$, $1); }
       | '(' expresion ')' { strcpy ($$, $2); }
       | TENTERO { sprintf ($$, "%d", $1);}
;
```

Reutilización de temporales

- Una vez que una variable temporal aparece como operando, deja de utilizarse.

```
t0 := A * B;
t1 := C * D;
t2 := t0 + t1;
t3 := E * F;
t4 := t2 - t3;
t5 := t4 + 10;
```

```
t0 := A * B;
t1 := C * D;
t0 := t0 + t1;
t1 := E * F;
t0 := t0 - t1;
t0 := t0 + 10;
```

```
0
1
2
1
2
1
1
```

- Al traducir el código intermedio, en lo posible se utilizarán registros (habrá una cantidad limitada); de lo contrario, posiciones de memoria.
- Reducción de temporales requeridas:

1. `c = 0`
2. Para generar un nuevo nombre temporal, utilizar `tc`, e incrementar `c` en `1`.
3. Cuando aparezca un nombre temporal como operando, decrementar `c` en `1`.

5/6. Estructuras de Control

- Sin consideraciones de eficiencia, la generación de código es relativamente sencilla:
- **Ejemplo 1:**

```
repetir
    <instr>
hasta <exp>
```



```
INSTR:
; <instr>
; <exp>
  JMF INSTR
```

```
repetir: tREPETIR
{
  $<instr>$ = nueva_etiqueta();
  etiqueta ($<instr>$);
}
lista_instrucciones
tHASTA_QUE expresion
{
  generar (JMF, $<instr>2);
}
;
```

Selección (sec)

```
si <exp>
   ent
    <instr1>
   si_no
    <instr2>
fsi
```



```
; <exp>
  JMF SINO
; <instr1>
  JMP FIN
SINO:
; <instr2>
FIN:
```

```
seleccion: tSI expresion
{
  $<sino>$ = nueva_etiqueta();
  generar (JMF, $<sino>$);
}
tENT lista_instrucciones
{
  $<fin>$ = nueva_etiqueta();
  generar (JMP,$<fin>$);
  etiqueta ($<sino>3);
}
resto_seleccion tFSI
{
  etiqueta($<fin>6);
}
;
resto_seleccion:
| tSI_NO lista_instrucciones
;
```

Optimalidad

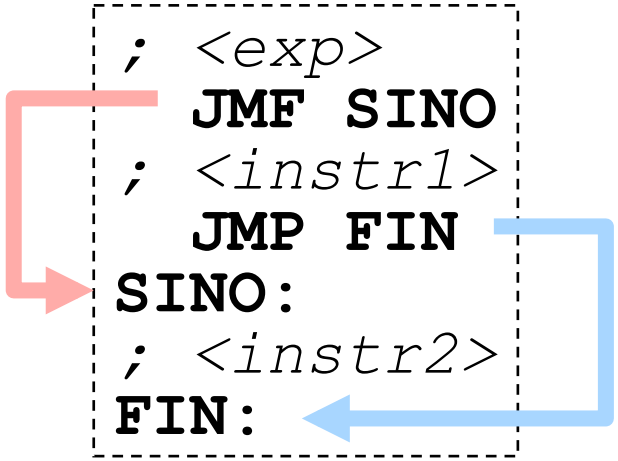
- La generación de código puede no ser óptima:

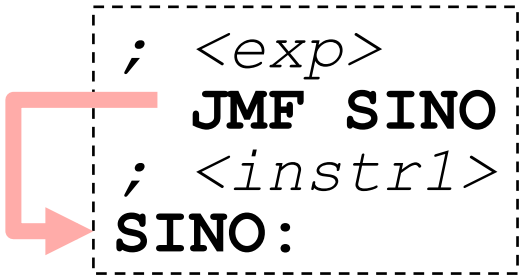
```
si i = 1
   ent i := i + 1;
fsi
```



```
; exp
    SRF n o
    DRF
    STC 1
    EQ
    JMF L0
; 11
    SRF n o
    SRF n o
    DRF
    STC 1
    PLUS
    ASG
    JMP L1
L0:
L1:
```

Selección (AST)

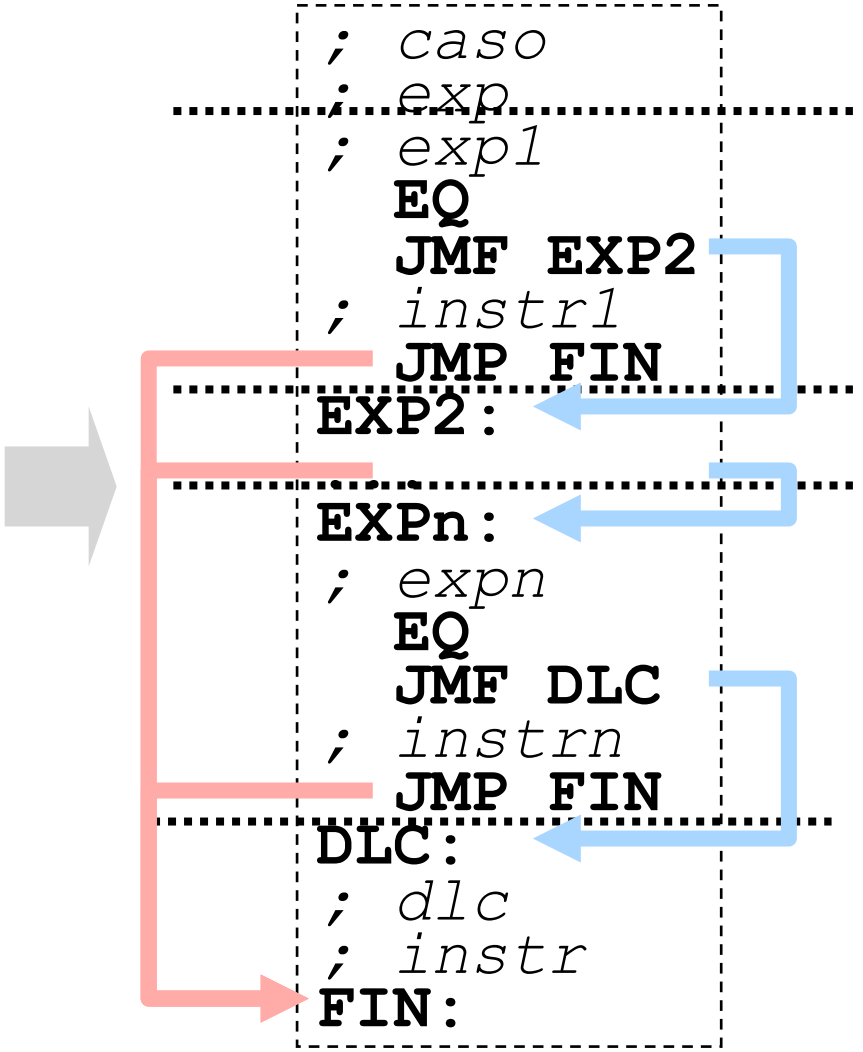





```
seleccion:
        tSI expresion
        tENT lista_instrucciones
        resto_seleccion tFSI
{
  char *lsino = newlabel(),
       *lfin  = newlabel();

  $$ = $2.cod;
  emit (&($$), JMF, lsino);
  concatenar (&($$), $4);
  if (longitud_lista($5)) {
     emit (&($$), JMP, lfin);
     label (&($$), lsino);
     concatenar (&($$), $5);
     label (&($$), lfin);
  }
  else label (&($$), lsino);
}
;
resto_seleccion:
{ $$ = newcode();}
| tSI_NO lista_instrucciones
{ $$ = $2};
```

Selección múltiple

```
caso <exp>
    <exp1> : <instr1> ;
    ...
    <expn> : <instrn> ;
    dlc <instr>
fcaso
```

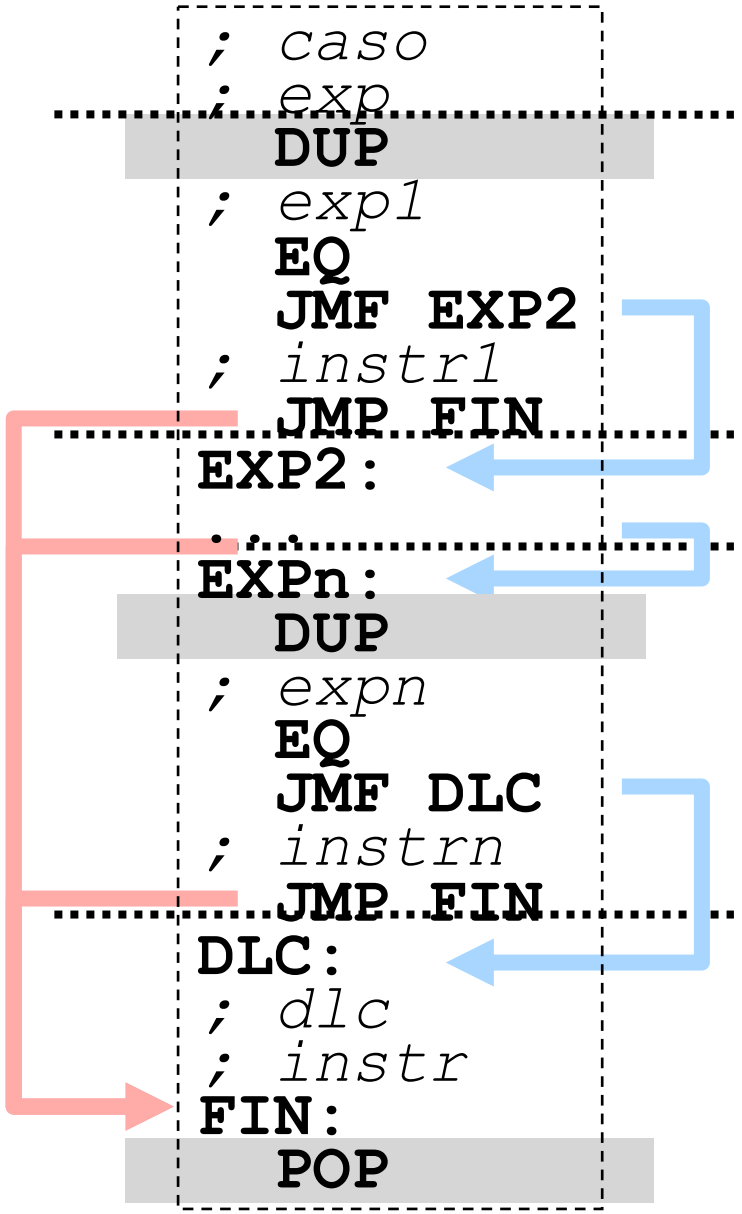


¿problemas?

Selección múltiple

equivale a:

```
si <exp> = <exp1>
   ent <instr1>
si_no si <exp> = <exp2>
   ent <instr2>
....
si_no si <exp> = <expn>
   ent <instrn>
si_no <instr>
fsi
...
fsi
```



Mientras que (sec)

```
mq <exp> hacer
    <instr>
fmq
```



```
EXP:
; <exp>
  JMF FIN
; <instr>
  JMP EXP
FIN:
```

```
mientras_que: tMQ
{
  $<exp>$ = nueva_etiqueta();
  etiqueta ($<exp>$);
}
expresion
{
  $<fin>$ = nueva_etiqueta();
  generar (JMF, $<fin>$);
}
instr tFMQ
{
  generar (JMP, $<exp>2);
  etiqueta($<fin>4);
}
;
```

Mientras que (AST)

```
mq <exp> hacer
    <bloque>
fmq
```



```
JMP COND
BLOQUE:
  ; <bloque>
COND:
  ; <exp>
  JMT BLOQUE
```

```
mientras_que: tMQ expresion bloque tFMQ
{
   char *lcond = newlabel(),
        *lbloque = newlabel();

   $$ = code (JMP, lcond);
   label (&($$), lbloque);
   concatenar( &($$), $3);
   label (&($$), lcond);
   concatenar (&($$), $2);
   emit (&($$), JMT, lbloque);
}
```

6/6. Procedimientos y Funciones

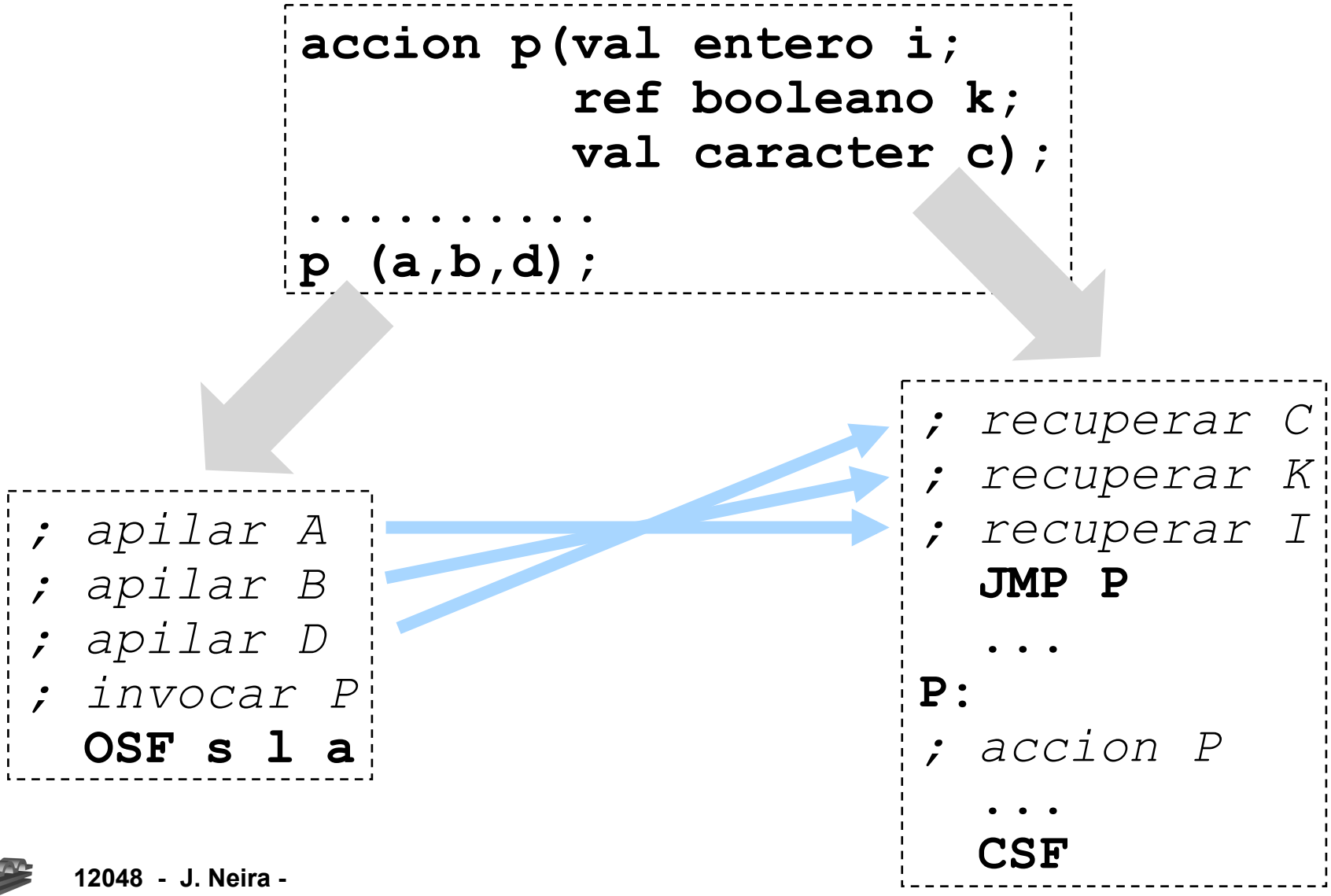
Declaraciones:
- Recuperar los argumentos
- Generar el código del procedimiento/función
- Devolver el valor resultado (funciones)

```
accion q (val entero i; ref booleano t);
entero j;
booleano f;

    accion r (ref entero j);
    entero i;
    principio
       q (i, t);
       r (j)
    fin

principio
  q (j, f);
  r (i)
fin
```

Invocaciones:
- Apilar los argumentos
- Ejecutar la invocación

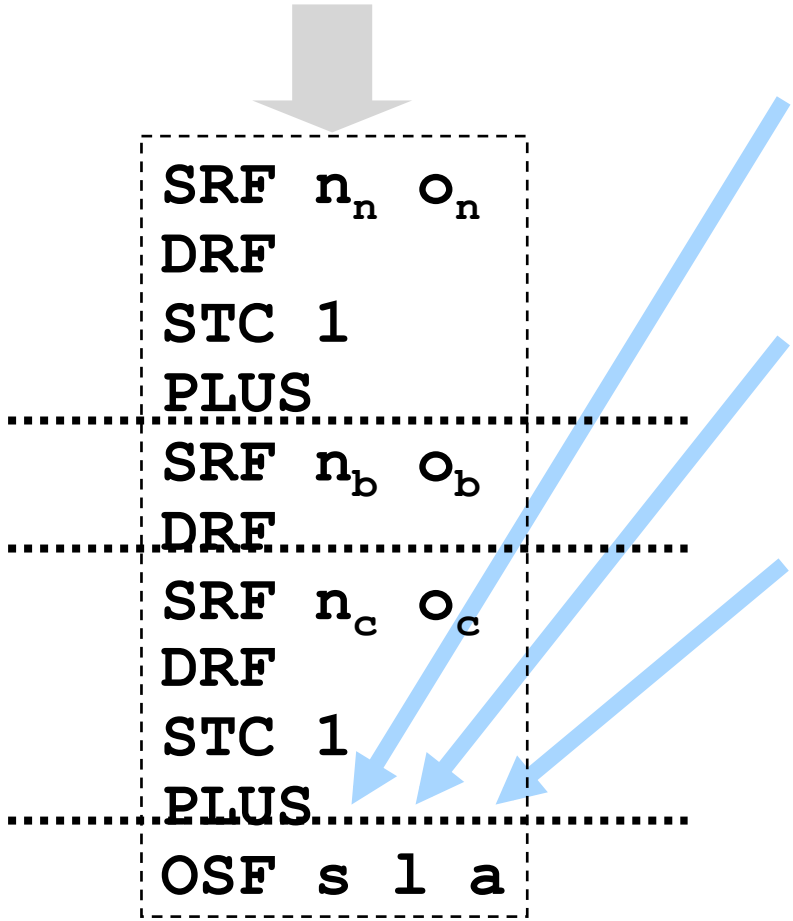
Argumentos

- Los argumentos se transmiten a través del stack



Invocación de Procedimientos

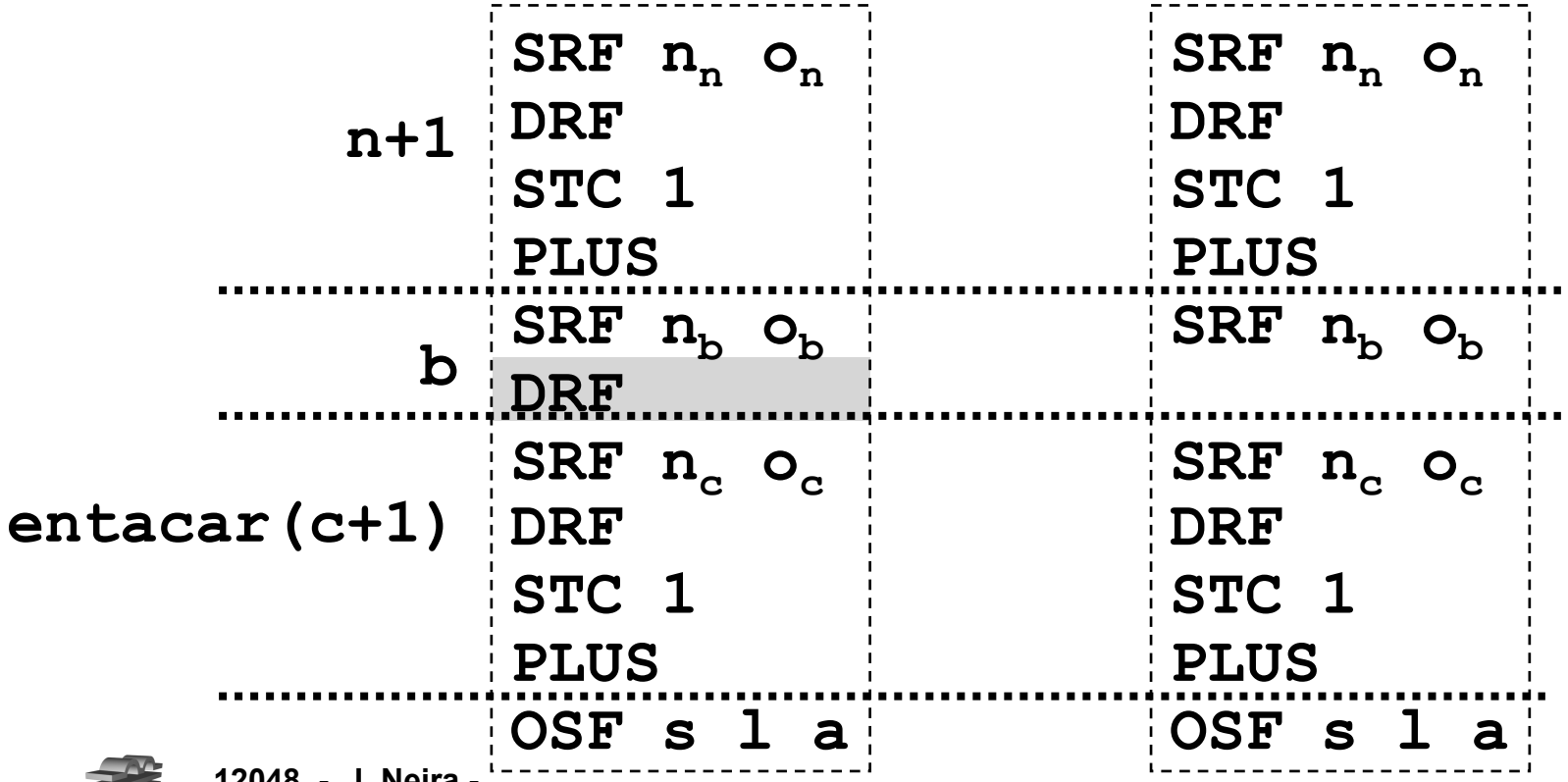
```
p (n+1,
   b,
   entacar(c+1));
```



- Cuando se evalúan las expresiones correspondientes a los argumentos, éstos van almacenándose en la pila.
- Al crear el BA del procedimiento, debe respetarse el tamaño del bloque actual
- El cambio de nivel es la diferencia entre el nivel actual y el invocado.
- La dirección del procedimiento o función se determina al introducir el símbolo en la tabla.

Invocación de Procedimientos

- Los parámetros **`ref`** requie-ren la **dirección** de la varia-ble referenciada; **`expresion`** genera código para obtener el **valor** de la variable.
- Para los parámetros **`ref`** **elimino** la última instruc-ción de código generado por **`expresion`**



Declaración de procedimientos

- Evitar el posible código de procedimientos y funciones locales

```
accion q (...);

    accion r (...);
    principio
    ...
    fin

principio
...
fin
```



```
; accion Q
; recuperar args Q
  JMP Q
; accion R
; recuperar args R
  JMP R
R:
; codigo de R
  CSF
Q:
; código de Q
  CSF
```

```
; accion R
; recuperar args R
R:
; codigo de R
  CSF
; accion Q
; recuperar args Q
Q:
; código de Q
  CSF
```

Recuperar los argumentos

- Los parámetros por **valor/referencia** se tratan de forma separada:
  - Todos se recuperan al inicio, unos son valores y otros son direcciones
  - Ninguno se devuelve al final.

```
; recuperar C
  SRF 0 5
  ASGI
; recuperar K
  SRF 0 4
  ASGI
; recuperar I
  SRF 0 3
  ASGI
```

- Los parámetros por **copia** se tratan como variables locales:
  - Todos se recuperan al inicio
  - Los que son S o E/S se devuelven al final

- Los argumentos de tipo vector pueden requerir que todas las componentes sean almacenadas en la pila y luego recuperadas.

Utilización de parámetros

```
programa p;
 entero i, j;

 accion q(ref entero m);
 principio
   escribir (m);
   m := 0
 fin

 accion r(val entero k;
          ref entero l);
 principio
   escribir (k, l);
   l := 0;
   q (k);
   q (l);
 fin

principio
  r (i, j);
fin
```

1. valor de un parámetro por referencia
2. dirección de un parámetro por referencia
3. valor de un parámetro por valor y otro por referencia
4. parámetro por valor utilizado como argumento a parámetro por referencia
5. parámetro por referencia utilizado como argumento a parámetro por referencia
6. variables utilizadas como parámetros por valor y referencia respectivamente

Código que debe ser generado

```
        ENP L0
; accion Q
        SRF   0   3 ;rec. M
        ASGI
        JMP   L1
L1:     SRF   0   3 ; M
        DRF
        DRF
        WRT   1
        SRF   0   3 ; M
        DRF
        STC   0
        ASG
        CSF
; accion R
        SRF   0   4 ;rec. L
        ASGI
        SRF   0   3 ;rec. K
        ASGI
        JMP L2
L2:     SRF   0   3 ; K
        DRF
        WRT   1
```

```
        SRF   0   4 ; L
        DRF
        DRF
        WRT   1
        SRF   0   4 ; L
        DRF
        STC   0
        ASG
        SRF   0   3 ; K
        OSF   5   1   1 ; Q
        SRF   0   4 ; L
        DRF
        OSF   5   1   1 ; Q
        CSF
; ppal P
L0:     SRF   0   3 ; I
        DRF
        SRF   0   4 ; J
        OSF   5   0 13 ; R
        LVP
```

Y para funciones.....

- Versión C:

```
funcion mcd(
     val entero a,b)
          dev entero;
entero r;
principio
   r := a mod b;
   mq r <> 0
      a := b;
      b := r;
      r := a mod b
   fmq;
   dev (b);
fin
```

```
; recuperar parametros
SRF    0   4 ; B
ASGI
SRF    0   3 ; A
ASGI
JMP   MCD
; codigo de mcd
MCD:
...
...
; dejar resultado en
; la pila
SRF    0   4
DRF
CSF
CSF
```

Y para funciones.....

- Versión Pascal:

```
funcion mcd(
    val entero a, b)
        dev entero;
entero r;
principio
  r := a mod b;
  mq r <> 0
    a := b;
    b := r;
    r := a mod b
  fmq;
  mcd := b;
fin
```

```
; recuperar parámetros
SRF    0   5 ; B
ASGI
SRF    0   4 ; A
ASGI
JMP   MCD
; código de mcd
MCD:
...
...
SRF    0   3 ; MCD
SRF    0   5 ; B
DRF
ASG
; dejar resultado en
; la pila
SRF    0   3 ; MCD
DRF
CSF
```