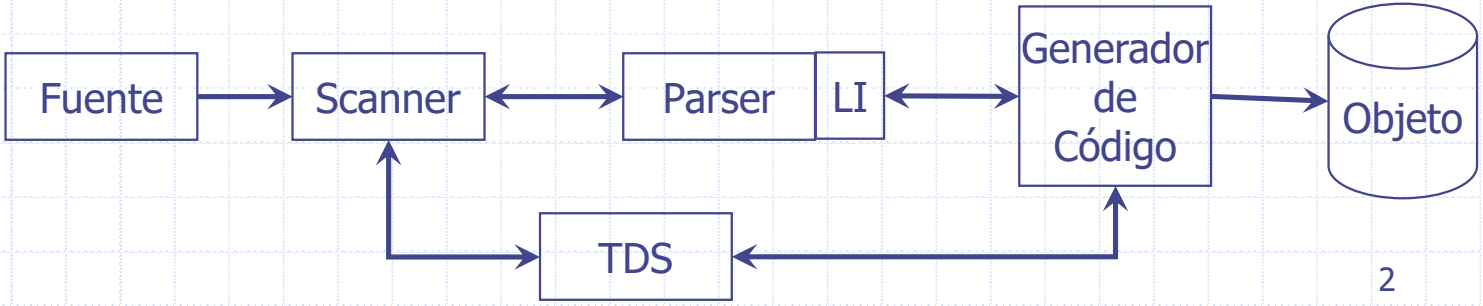
Tema 7. Generación de código intermedio

Procesadores del Lenguaje
Curso 2020-2021

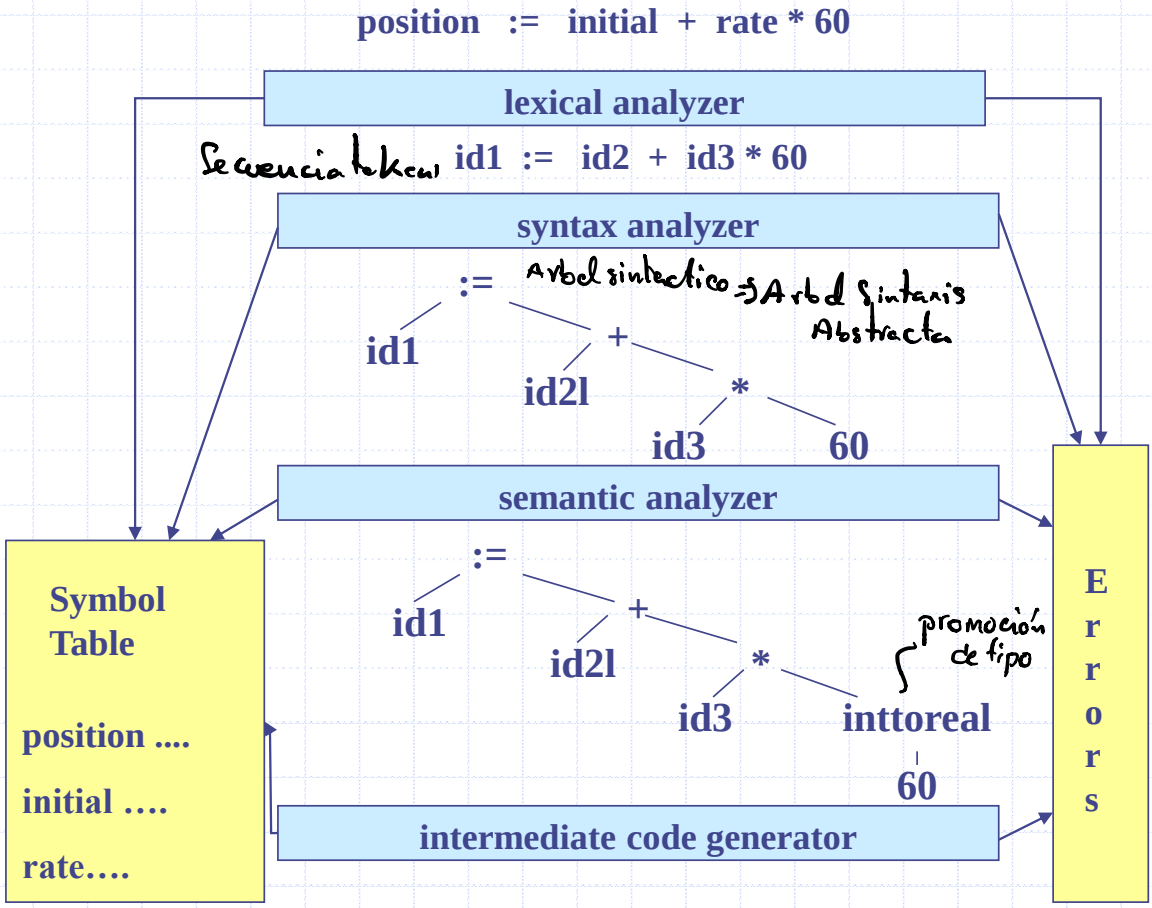
Tras el análisis, en la parte de Generación

Generación de Código Intermedio

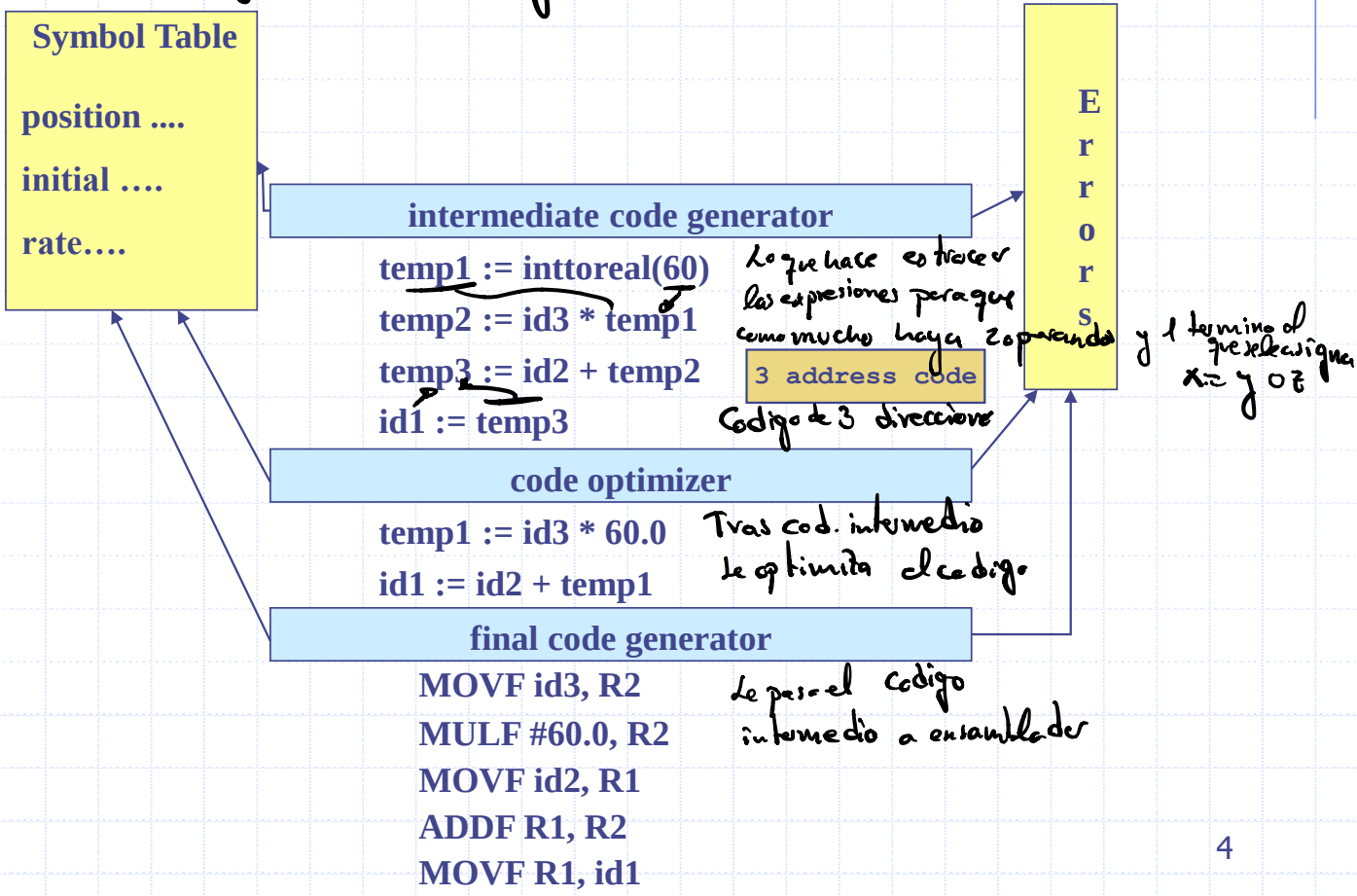
- Proceso de Síntesis
  - Lenguaje Intermedio
  - Generación de Código
- Ventajas del código intermedio
  - Facilitar la fase de optimización
  - Aumentar la portabilidad del compilador de una máquina a otra
    - Se puede utilizar el mismo analizador para diferentes generadores
    - Se pueden utilizar optimizadores independientes de la máquina
  - Facilitar la división en fases del proyecto



Fase de Análisis



Fase de Síntesis



Tipos de representaciones intermedias

- La representación del código intermedio depende de la máquina objeto:
  - 0-direcciones: código para máquinas de pila, tienen que ser extremadamente sencilla. solo lo que ve.
  - 2-direcciones: códigos para máquinas con operaciones sobre registros de memoria Puede almacenar valores en registros
  - 3-direcciones: código para máquinas de arquitectura RISC Puede indexar memoria y acceder a distintas partes de memoria.
- En todo caso, añade un recorrido descendente adicional para generar el código final

Tipos de representaciones intermedias

- Árboles de Sintaxis Abstracta, es una posible representación aunque no la + eficiente muestra la info pero no que ve el orden.
- Notación Polaca Inversa (RPN)
  - Los operadores van después de los operandos
    - S = A + B * C ⟹ S A B C * + =
  - Ventajas
    - Facilidad para generar código a partir de ella
    - Es la notación más sencilla para el lenguaje intermedio
  - Inconvenientes
    - El código es difícil de entender
    - No es útil para optimización de código

Altera el orden para que se puedan hacer en pila. El operando se aplica sobre los anteriores

- Códigos de tres direcciones
  - Cuartetos, es la representación más directa.
  - Tercetos
  - Tercetos Indirectos

Códigos de Tres Direcciones

Instrucciones que como mucho involucran tres direcciones

- Sentencias del estilo x = y op z
  - Cada línea de código tiene un operador y hasta tres direcciones
  - Uso abundante de variable temporales
- Directamente relacionado con evaluación de expresiones
  - Ejemplo: a = b*(c+d) se traduce a:

    tmp1 = c+d

    a = b*tmp1

Códigos de Tres Direcciones

- Es una representación lineal del ASA

Ejemplo 1:

a=b*-c+4

Codigo 3 direcciones

t1=-c
t2=b*t1
t3=4
t4=t3+t2
a=t4

En expresiones se generan variables temporales

Ejemplo 2:

```
if cond then then_statements
else
  else_statements;
end if;
```

Codigo 3 direcciones

```
t1 = cond
if not t1 goto else_label
codigo para "then_statements"
goto endif_label
else_label:
codigo para "else_statements"
endif_label:
mas instrucciones
```

En estructuras de control de flujo se ponen ¿sentencias? y saltos condicionales

Instrucciones en Código 3 direcciones

- Asignaciones: **x:=y op z** (op aritmético o lógico)
- Asignaciones 1 argumento **x:= op y**
- Copia **x:=y**
- Salto incondicional: **goto L** (L etiqueta)
- Salto condicional **gotoc x L**
  - Condición falsa **gotocf x L**
  - Relacional **gotoge x y L** (*si x>=y goto L*)
- Llamada a subrutina:

  param x1

  ...

  param xn

  call p,n (donde *p* es le subrutina a llamar y *n* el nº de agumentos especificados previamente)
- Asignaciones indexadas

  x:=y[i]

  x[i]:=y

Códigos de 3 direcciones

- Cada línea de código tiene un operador y hasta tres direcciones
- Tipos: Cuartetos, Tercetos, Tercetos Indirectos
- Cuartetos
  - Se representan por cuatro valores:
    (<OPERADOR>,<Operando$_1$>,<Operando$_2$>,<Resultado>)
  - Ejemplos

Expresión	**Cuartetos**			**Representación**
S:=A+B*C	*	B	C	T_1	(*, B, C, T_1)
	+	A	T_1	T_2	(+, A, T_1, T_2)
	:=	T_2		S	(:=, T_2, , S)
IF A<B THEN X:=B	<	A	B	E_1	(<, A, B, E_1)
	GOTOCF	E_1		E_2	(GOTOCF, E_1, , E_2)
	:=	B		X	(:=, B, , X)
	LABEL			E_2	(LABEL, , , E_2)

Implementación de código 3 direcciones

- Implementación con cuartetos
  - 4 campos: (op,y,z,x) para x=y op z
  - Campos se anulan si no se necesitan, ej.: (label,x,,)
- Implementación con tercetos
  - El cuarto elemento siempre es un temporal
  - En lugar de nombrarlo, utilizar el propio índice del terceto

Ejemplo: a = b+(c*d)

[cuartetos]	[tercetos]
1. (mul,c,d,t1)	1: (mul,c,d)
2. (add,b,t1,t2)	2: (add,b,(1))
3. (asn,a,t2,_)	3: (asn,a,(2))

Tercetos

- Los cuartetos son la herramienta más general
- Inconvenientes
  - Ocupan más espacio
  - Más variables auxiliares para resultados intermedios
- Los tercetos suprimen el operando del resultado, queda implícito en el terceto

  (<OPERADOR>, <Operando$_1$>, <Operando$_2$>)
  - Hacen referencia a otro terceto
  - Son equivalentes a Árboles de Sintaxis Abstracta

GCI para declaraciones

- Construcción explícita de la tabla de símbolos
  - Se crean entradas para cada variable según su tipo
  - Reserva de espacio en función del tipo empleado
    - **offset** (*desplazamiento*) es una variable global que contiene la siguiente dirección relativa disponible, valor que obtiene de la tabla de símbolos actual
    - **T.tipo, T.ancho** caracterizan cada entrada en la tabla de símbolos (*T.tipo* es el tipo de dato y *T.ancho* el nº de unidades de memoria empleadas por los objetos de tipo **T.tipo** – 4 para enteros y 8 para reales por ejemplo –)
- Simplificación: La tabla de símbolos puede simplificarse si sólo se tiene un procedimiento (sin necesidad de resolución de ámbitos), si no ➔ Extensiones para declaraciones en procedimientos y procedimientos anidados (resolución de ámbitos)

GCI para expresiones

- Se precisan nombres de variables temporales para los nodos interiores del árbol sintáctico
- Se calcula el valor del no terminal E en el lado izquierdo de E→E+E dentro de un nuevo temporal t: **E_1 a t_1; E_2 a t_2; t= t_1+t_2**
  - **E.pos**, es el nombre que contendrá el valor de E ("posición" en TS)
  - **E.cod**, son las instrucciones de tres direcciones que evalúan E
  - La función *tempnuevo()* devuelve una secuencia de nombres distintos *temp1*, *temp2*,... En sucesivas llamadas
  - Las variables tienen el nombre id.lex, y se accede a su posición en la tabla de símbolos como lookup(id.lex)
  - gen(.) genera código de tres direcciones
  - || concatenación de código
- Podría optimizarse el número de temporales

Asignaciones y Aritmética

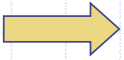
La va concatenando el código con los cuartetos de calculo.

Producciones	Regla Semántica
S→ **id** := E	{p=lookup(id.lex); posición del id porema en pelvis de E.pos S.cod := E.cod || gen(p ':=' E.pos)}
E→ E+E	{E_0.pos := tempnuevo(); E_0.cod := E_1.cod || E_2.cod || 3 gen(E_0.pos ´:=´ E_1.pos ´+´ E_2.pos)}
E→ E*E	{E_0.pos := tempnuevo(); resultado a var temp. E_0.cod:= E_1.cod || E_2.cod || 2 gen(E.pos ´:=´ E_1.pos ´*´ E_2.pos)}
E→ -E	{E_0.pos := tempnuevo(); E_0.cod := E_1.cod || gen(E_0.pos´:=menosu´ E_1.pos)}
E→ (E)	{E_0.pos := E_1.pos; E_0.cod := E_1.cod}
E→ **id**	{p=lookup(id.lex), E.pos := p;}

Con conversiones de tipos

Producciones	Regla Semántica
$S \rightarrow$ **id** := E	{p=lookup(id.lex); S.cod := E.cod || gen(p ':=' E.pos)}
$E \rightarrow E+E$	{E_0.pos := tempnuevo(); E_0.cod := E_1.cod || E_2.cod || if (E_1.tipo=integer and E_2.tipo=integer) then gen(E_0.pos ´:=´ E_1.pos ´+´ E_2.pos) E_0.tipo=int elseif (E_1.tipo=real and E_2.tipo=real) then gen(E_0.pos ´:=´ E_1.pos ´+´ E_2.pos) E_0.tipo=real elseif (E_1.tipo=integerl and E_2.tipo=real) then gen(E_0.pos ´:=inttoreal´ E_1.pos ´+´ E_2.pos) E_0.tipo=real elseif (E_1.tipo=real and E_2.tipo=integer) then gen(E_0.pos ´:=I´ E_1.pos ´+ inttorea´ E_2.pos) E_0.tipo=real}
....	

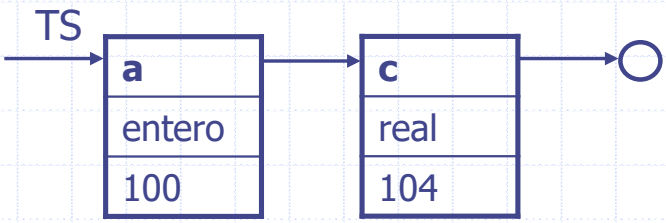
Ej.: x:=y+i*j

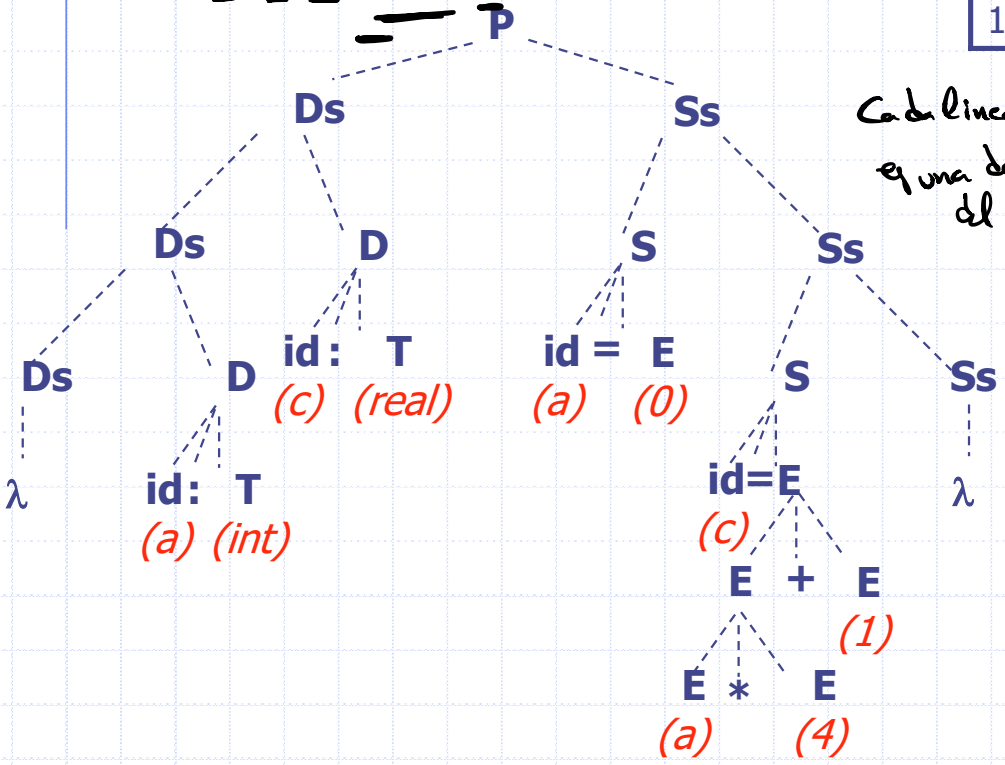


t_1:=i*j
t_3:=inttoreal t_1
t_2:=y + t_3
x:=t_2

Evaluación de Declaraciones y Expresiones

Ej.: a: int; c: real;
a=0;c=a*4+1;





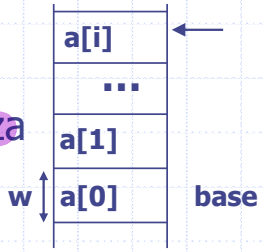
Cada línea de código es una de las reducciones del árbol.

Código Intermedio:

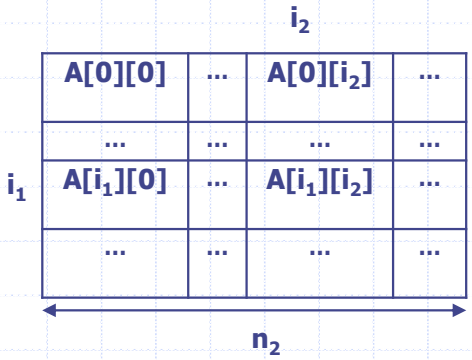
t_1:=0
a:=t_1
t_2:=inttoreal a
t_3:= t_2 *4
t_4:= t_3+1
c:=t_4

Evaluación de Arrays

- Un array se accede como un bloque de posiciones consecutivas. Si empieza en 0:
  - posicion a[i]: base+i*w



- Matriz 2 dimensiones (por filas): $A[i_1][i_2]$
  - n_1 filas, n_2 columnas
  - posicion: base+$(i_1*n_2+i_2)$*w

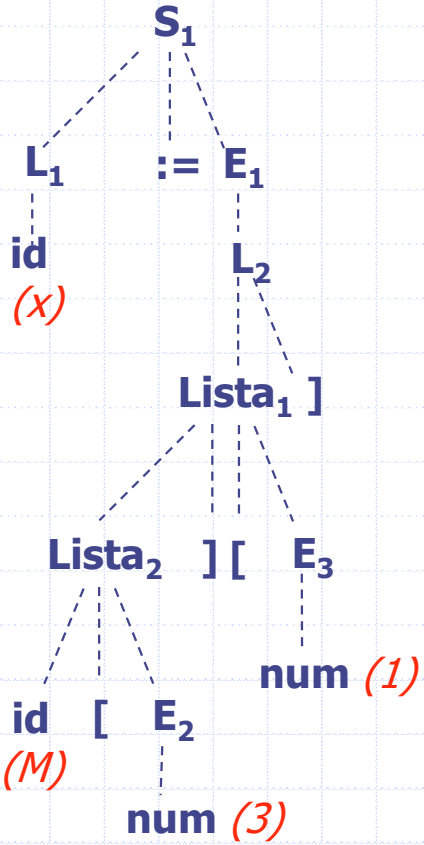


- Generalización a k dimensiones: $A[i_1][i_2]...[i_k]$
  - posicion: base+((...$((i_1*n_2+i_2)*n_3+i_3)$...)$*n_k+i_k$)*w
  - Ecuación recursiva:
    - $e_1=i_1$
    - $e_m=e_{m-1}*n_m+i_m$

Evaluación de Arrays

Ej.: x:=M[3][1] (M de tipo array{[0…4]x[0…4],int})

- base
- limite(i)
- ancho celda



Código Intermedio:

t_1:=3

t_2:=1

t_3:=t_1*5 salto fila

t_3:=t_3+ t_2 nueva columna

t_4:=base(M)

t_5:=t_3*4 bytes de desplazamiento

t_6:=t_4[t_5] Entra en desplaz ts bytes

x:=t_6 valor de esa pos

GCI control de flujo y exp booleanas

- Dos posibilidades:
  - Valores numéricos (0,1) …
  - Etiquetas para control de flujo (CF)
- Una expresión booleana tiene dos atributos heredados: E.true, E.false, posiciones destino de CF

if E then S_1 else S_2

Codigo 3 direcciones

código de evaluación de E (hacia qué etiqueta salto)
label E.true:
código para S_1
goto next
label E.false:
código para S_2
label next:
…siguientes instrucciones

GCI- Sentencias Condicionales

- Expresiones booleanas E:
  - E.true, E.false: etiquetas de salto (atributos heredados)
  - E.cod: código de evaluación de salto (atributo sintetizado)

- Sentencias condicionales S:
  - S.cod: código, atributo sintetizado
  - S.next: atributo heredado: instrucción continuación del bloque de código de S

    - Problema: etiquetas desconocidas a priori

GCI- Condiciones Booleanas

Producción	Regla Semántica
E:=true	{E.cod = gen (‘goto’ E.true)}
E:=false	{E.cod = gen (‘goto’ E.false)}
E:=id relop id	{E.cod= gen(‘gotorel’, id_1.pos, id_2.pos, E.true)|| gen(‘goto’ E.false)}

Salto directamente al valor de la etiqueta.

Instrucciones de salto hacia las etiquetas

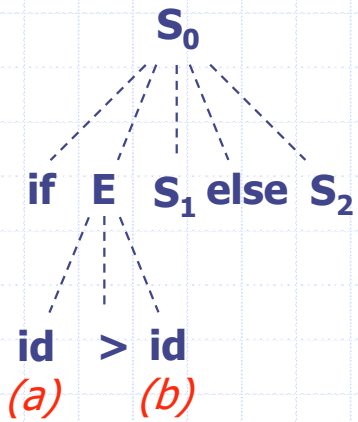
GCI- Sentencias Condicionales

Producción	Regla Semántica
S:=if E then S	{E.true= newlabel() E.false= S_0.next S_1.next= S_0.next S_0.cod= E.cod||gen(`label: ' E.true)|| S_1.cod}
S:=if E then S else S	{E.true= newlabel() E.false= newlabel() S_1.next= S_0.next S_2.next= S_0.next S_0.cod= E.cod || gen(`label: ' E.true) || $S_1$.cod || gen(`goto' S_0.next) || gen(`label: ' E.false) || S_2.cod}

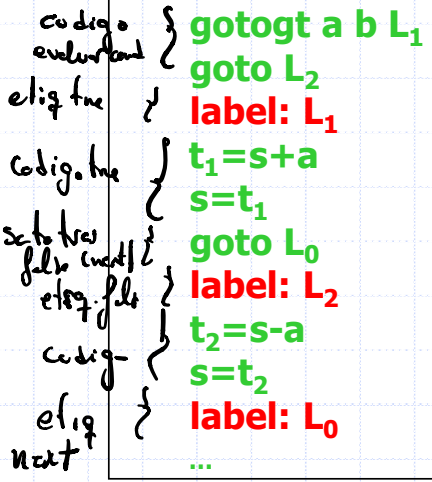
Instrucciones de salto e intercala etiquetas

Ejemplo control de flujo I

```
if a>b
    s=s+a
else
    s=s-a
```



Código Intermedio



Ejemplo control de flujo I

S_0.next=L_0 ***/*S_0->if E then S_1 else S_2*/***

E.true=L_1

E.false=L_2

E: ***/*E->id op id */***

E.cod=gotogt a b L_1

goto L_2

S_1.next=L_0

S_1: ***/*S_1->s=s+a */***

S_1.cod=t_1=s+a

s=t_1

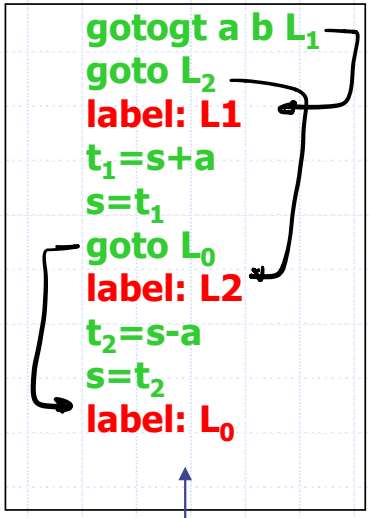
S_2.next=L_0

S_2: ***/*S_2->s=s-a */***

S_2.cod=t_2=s-a

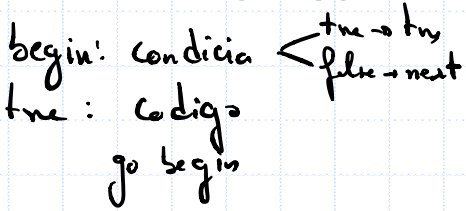
s=t_2

S_1.codigo=



GCI- Sentencias Condicionales

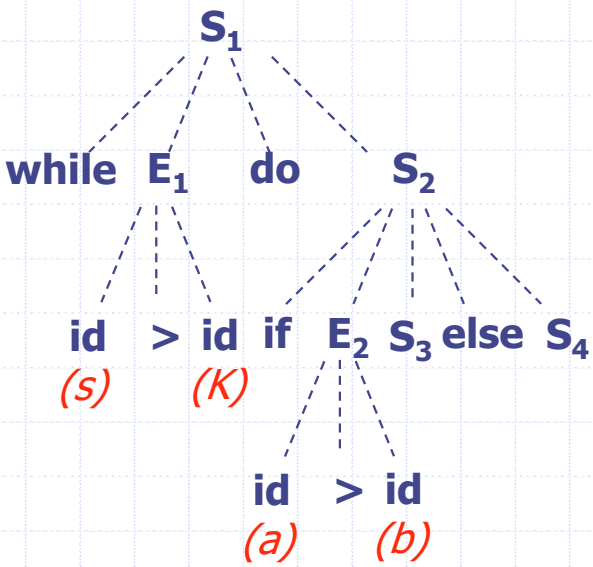
Producción	Regla Semántica
S:= while E do S	{begin = newlabel() E.true= newlabel() E.false= S_0.next ~fin de bloque S_1.next= begin S_0.cod= gen('label: ' begin) || E.cod || gen(E.true ':')|| S_1.cod || gen('goto' begin)}



Instrucciones de salto e intercala etiquetas

Ejemplo control de flujo II

```
while s<K do
    if a>b
        s=s+a
    else
        s=s-a
```



Código Intermedio

```
label: L1
gotolt  s K L2
goto L0
label: L2
gotogt a b L3
goto L4
label: L3
t1=s+a
s=t1
goto L1
label: L4
t2=s-a
s=t2
goto L1
label: L0
...
```

S_1.next=L_0 (**)
S_1: ***/S_1->while E_1 do S_2/***
S_1.next=L_0
begin=L_1
E_1.true=L_2
E_1.false=L_0
E_1: ***/E_1->id op id/***
E_1.cod=gotolt s K L_2
goto L_0
S_2.next=L_1
S_2: ***/S_2->if E_2 then S_3 else S_4/***
E_2.true=L_3
E_2.false=L_4
E_2: ***/E_2->id op id/***
E_2.cod=gotogt a b L_3
goto L_4
S_3.next=L_1
S_3: ***/S_3->s=s+a/***
S_3.cod=t_1=s+a
s=t_1
S_4.next=L_1
S_4: ***/S_4->s=s-a/***
S_4.cod=t_2=s-a
s=t_2
S_2.cod=
S_1.codigo=

```
gotogt a b L3
goto L4
label: L3
t1=s+a
s=t1
goto L1
label: L4
t2=s-a
s=t2
```

```
label: L1
gotolt  s K L2
goto L0
label: L2
gotogt a b L3
goto L4
label: L3
t1=s+a
s=t1
goto L1
label: L4
t2=s-a
s=t2
goto L1
label: L0
...
```

Operaciones booleanas en cortocircuito

- Evaluación más eficiente
- Permite evitar evaluar instrucciones incorrectas
  - if i < MAX and a[i]!=num
- Equivalencia a condicionales internos
  - B1 and B2

```
if B1 then
    B2
else
    false
```

  - B1 or B2

```
if B1 then
    true
else
    B2
```

GCI- Expresiones Booleanas en cortocircuito

Producción	Regla Semántica
E:=true	{E.cod = gen ('goto' E.true)}
E:=false	{E.cod = gen ('goto' E.false)}
E:=E or E	{E_1.false= newlabel() E_1.true= E_0.true E_2.true= E_0.true E_2.false= E_0.false E_0.cod= E_1.cod || gen('label: ' E_1.false) || E_2.cod}
E:=E and E	{E_1.true= newlabel() E_1.false= E_0.false E_2.true= E_0.true E_2.false= E_0.false E_0.cod= E_1.cod || gen('label:' E_1.true) || E_2.cod}

GCI- Condiciones Booleanas

Evaluación "en cortocircuito". Ejemplo:

```
if (E) S:
  codigo de E
Lt0:
  código de S
Lf0:
  …
```

Ejemplo:

if (a<(b+32) || (b>0 && a>0))
x=1

Código intermedio

```
t1=b+32
if a < t1 goto Lt0
if b > 0 goto L1
goto Lf0
label: L1
if a > 0 goto Lt0
goto Lf0
Label: Lt0
x=1
Label: Lf0
…
```