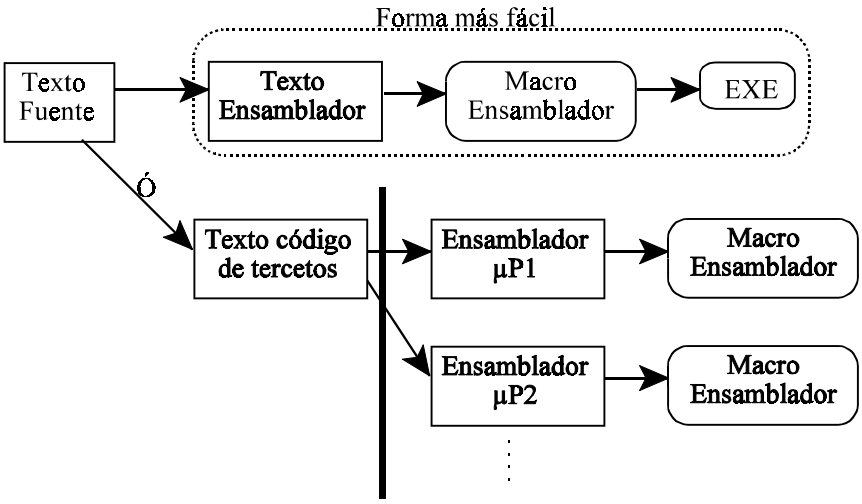
Tema 7
Generación de Código

En el modelo de análisis y síntesis de un compilador, la etapa inicial traduce un programa fuente a una representación intermedia a partir de la cual la etapa final genera el código objeto. Los detalles del lenguaje objeto se confinan en la etapa final, si esto es posible. Aunque un programa fuente se puede traducir directamente al lenguaje objeto, algunas ventajas de utilizar una forma intermedia independiente de la máquina son:

1. Se facilita la redestinación; se puede crear un compilador para una máquina distinta uniendo una etapa final para la nueva máquina a una etapa inicial ya existente.

2. Se puede aplicar a la representación intermedia un optimizador de código independiente de la máquina.



Hay lenguajes que son pseudointerpretados que utilizan un código intermedio llamado código-P que utiliza lo que se denomina bytecodes (sentencias de un µP hipotético). Por ejemplo Java utiliza los ficheros .class, éstos tienen unos bytecodes que se someten a una JavaVirtualMachine, para que interprete esas sentencias.

En este capítulo se muestra cómo se pueden utilizar los métodos de analizadores dirigidos por la sintaxis para traducir a un código intermedio, construcciones de lenguajes de programación como declaraciones, asignaciones y proposiciones de flujo de control. La generación de código intermedio se puede intercalar en el análisis sintáctico.

Código de Tercetos

Para facilitar la comprensión de esta fase, no generaremos código máquina puro, sino un código intermedio cercano a la máquina, que además facilitará la optimización de código.

El código intermedio que vamos a usar , posee cuatro apartados:

- Operando 1º
- Operando 2º
- Operador
- Resultado

y se denomina código de 3 direcciones, de tercetos, o máximo 3 operandos.

Las instrucciones de tres direcciones son análogas al código ensamblador, pueden tener etiquetas simbólicas y existen instrucciones para el flujo de control.

Hay algunas instrucciones que carecen de algunos de estos apartados; los tercetos que podemos usar son:

- Asignación binaria: *x := y op z*, donde *op* es una operación binaria aritmética o lógica.

- Asignación unaria: *x := op*, donde *op* es una operación unaria. Las operaciones unarias principales incluyen el menos unario, la negación lógica, los operadores de desplazamiento y operadores de conversión de tipos.

- Asignación simple o copia: *x := y*, donde el valor de *y* se asigna a *x*.

- Salto incondicional: *goto etiqueta*.

- Saltos condicionales: *if x oprelacional y goto etiqueta*

- Para llamar a un procedimiento se tienen códigos para meter los parámetros en la pila de llamadas, para llamar al procedimiento indicando el número de parámetros que debe recoger de la pila, para tomar un valor de la pila, y para retornar.

param x	Mete al parámetro real *x* en la pila.
call p, n	Llama al procedimiento *p*, y le dice que tome *n* parámetros de la cima de la pila
pop x	Toma un parámetro de la pila
return y	Retorna el valor *y*

- Asignación indexada: x := y[i]; x[i] := y, donde *x* o *y* es la dirección base, y donde *i* es el desplazamiento

- Asignación indirecta: x:=&y; x:=*y; Son asignaciones de direcciones y asignaciones de punteros

Código de Tercetos

La elección de operadores permisibles es un aspecto importante en el diseño de código intermedio. El conjunto de operadores debe ser lo bastante rico como para implantar las operaciones del lenguaje fuente. Un conjunto de operadores pequeño es más fácil de implantar en una nueva máquina objeto. Sin embargo un conjunto de instrucciones limitado puede obligar a la etapa inicial a generar largas secuencias de proposiciones para algunas operaciones del lenguaje fuente. En tal caso, el optimizador y el generador de código tendrán que trabajar más si se desea producir un buen código.

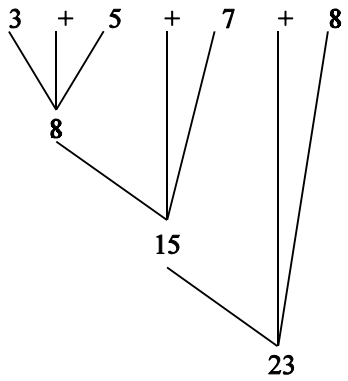
Ejemplo de código de tercetos

```
        c = a
label etqBucle
        if b = 0 goto etqFin
        b = b-1
        c = c+1
        goto etqBucle
label etqFin
```

Para ilustrar como se utiliza el código de tercetos en una gramática vamos a suponer que nuestra calculadora en vez de ser una calculadora interpretada es una calculadora compilada, es decir, en vez de interpretar las expresiones vamos a generar código intermedio equivalente.

Tendremos en cuenta la posibilidad de utilizar variables.

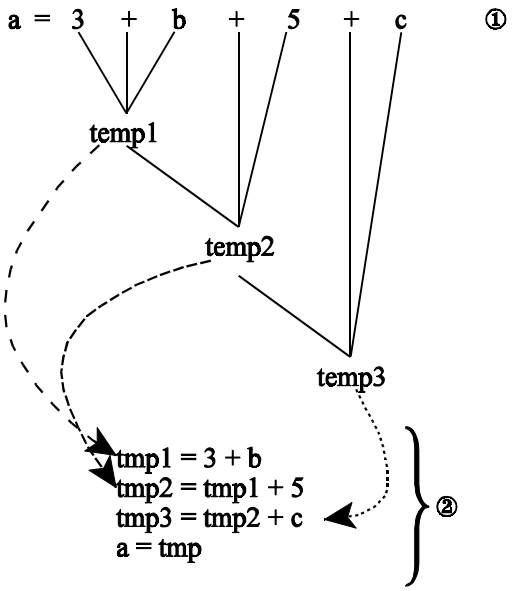
Las variables temporales sirven para almacenar resultados intermedios a medida que vamos calculando el resultado final.



Código de Tercetos

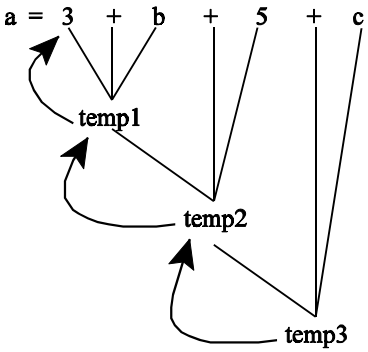
Cuando se genera código de tercetos, se construyen variables temporales para los nodos interiores del árbol sintáctico. Así, si por ejemplo tuviéramos la entrada

a = 3 + b + 5 + c



La pregunta es ¿① y ② son equivalentes? - Sí, y además ② es muy parecido a un código máquina.

Lo importante es saber hacer las reducciones y tener una gramática adecuada. El no terminal tiene que tener unos atributos asociados que los representen



temp1 representa a 3 + b
temp2 representa a temp1 + 5
temp 3 representa a temp2 +c

Lo importante es saber qué representa cada atributo y como puedo generar código.

Código de Tercetos

Hasta ahora, para generar el código de tercetos no hemos tenido que tener en cuenta los valores de *a*, *b*, y *c*. Por lo tanto no nos ha hecho falta ninguna tabla de símbolos para generar el código de terceto. Según el problema que tengamos planteado hay que ver si hace falta o no una tabla de símbolos. En el ejemplo anterior no hace falta una tabla de símbolos, porque no hay chequeo de tipos. Pero si tenemos un problema en el que exista una zona de declaración y una zona de definición, por ejemplo hacer un chequeo de tipos y un código de terceto, entonces sí nos haría falta tener una tabla de símbolos, o por ejemplo si decimos que el contenido de la dirección de memoria 12h = a, entonces también necesitamos una tabla de símbolos.

Más adelante veremos un ejemplo implementado, en el que haremos la generación de código de tercetos para nuestra calculadora. Comentaremos algunos aspectos de dicho programa

Sólo vamos a necesitar hacer una traducción, es decir, entra un texto y sale otro resultado de hacer la traducción del texto que entra.

Ejemplo: Entrada: a := 0

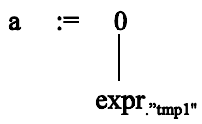
Salida: tmp1 = 0
a = tmp1

Lo único que vamos a hacer es sacarlo por pantalla, mediante *printf* .

- Código LEX. El *0* de *a:=0* no necesitamos pasarlo a entero con atoi, ya que solo lo vamos a imprimir en pantalla. Al no terminal *ID* y al no terminal *NUMERO*, le vamos a asignar un atributo del tipo cadena llamado nombre.

- Código YACC:

  Los atributos asociados van a ser

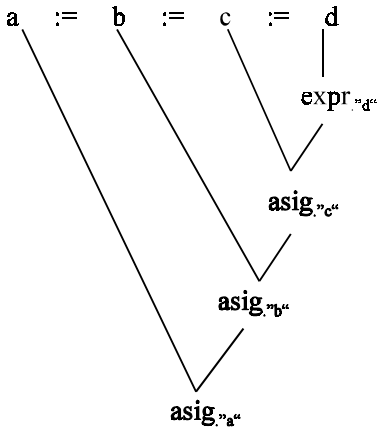


  por este motivo tengo que asignarle el tipo nombre a *expr*. Mediante la instrucción

  %type <nombre> asig expr

  asig necesita tener asociado un tipo, para poder hacer la propagación en las asignaciones múltiples. Ejemplo:

Código de Tercetos



Vemos la regla *prog*

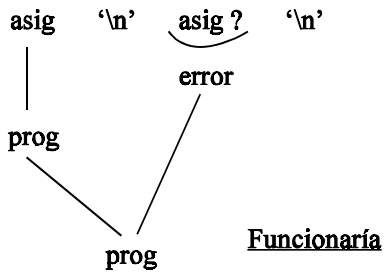
```
prog    :       asig '\n'
        |       prog asig '\n'
        |       prog error '\n'
        |       ε
        ;
```

Si no me interesa que un prog sea vacío (ε), no me basta con quitar la regla ε. Veamos que ocurre si lo hiciésemos así

```
prog    :       asig '\n'
        |       prog asig '\n'
        |       prog error '\n'
        ;
```

Esto tiene problemas.

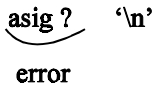
Ejemplo: Si el error da en la segunda asignación.



Funcionaría

Código de Tercetos

Pero, si el error da en la primera asignación. No funcionaría el compilador ya que nunca entraría por *prog*



¿Como se soluciona esto? - Añadiendo una regla

```
prog    :       asig '\n'
        |       prog asig '\n'
        |       prog error '\n'
        |       error '\n'
        ;
```

otra_etq( ): Genera variables temporales nuevas. En nuestro ejemplos se llaman var1, var2, ...

Resumiendo, los objetivos fundamentales en el código de tercetos son:

- Realizar una secuencia de printf (para la traducción)
- Gestionar adecuadamente los atributos

CÓDIGO LEX: Ejem5l.lex

```
%%

[a-zA-Z_][a-zA-Z_0-9]*      {
                              strcpy(yylval.nombre,yytext);
                              return ID;
                            }
":="            {       return ASIG;}
[0-9]+          {
                        strcpy(yylval.nombre,yytext);
                        return NUMERO;

                }
[ \t]           {;}
.|\n            {return yytext[0];}
```

CÓDIGO YACC: Ejem5y.yac

```
%{
#include <stdlib.h>
```

```
typedef char cadena[20];
%}

%union
        {
          cadena nombre;
        };

%token <nombre> ID NUMERO
%token ASIG
%type <nombre> asig expr

%left '+' '-'
%left '*' '/'
%right MENOS_UNARIO

%%

prog    : asig '\n'
                | prog asig '\n'
                | prog error '\n'           {yyerrok;}
                | error '\n'                {yyerrok;}
                ;

asig: ID ASIG expr    {
                  strcpy($$,$1);
                  printf("%s = %s\n",$$,$3);
                 }
        | ID ASIG asig    {
                  strcpy($$,$1);
                  printf("%s = %s\n",$$,$3);
                 }
        ;

expr: expr '+' expr {
                  strcpy($$,otra_etiq());
                  printf("%s = %s + %s\n",$$,$1,$3);
                 }
        | expr '-' expr {
                  strcpy($$,otra_etiq());
                  printf("%s = %s - %s\n",$$,$1,$3);
                 }
```

CÓDIGO YACC: Ejem5y.yac

```
        | expr '*' expr {
                  strcpy($$,otra_etiq());
```

```
                printf("%s = %s * %s\n",$$,$1,$3);
               }
        | expr '/' expr {
                strcpy($$,otra_etiq());
                printf("%s = %s / %s\n",$$,$1,$3);
               }
        | '-' expr %prec MENOS_UNARIO    {
                                strcpy($$,otra_etiq());
                                printf("%s = -%s\n",$$,$2);
                               }
        | '(' expr ')'      {
                              strcpy($$,$2);
                             }
        | NUMERO    {
                              strcpy($$,$1);
                             }
        | ID                {
                              strcpy($$,$1);
                             }
        ;
%%
#include "ejem5l.c"

void main()
{
        yyparse();
}
void yyerror(char * s)
{
        printf("%s\n",s);
}
char * otra_etiq()
{
        char * retorno;
        static int actual=0;
                actual++;
                retorno = (char *)malloc(sizeof(cadena));
                strcpy(retorno,&"var");
                itoa(actual,&(retorno[3]),10);
                return retorno;
}
```

Generación de código de tercetos en Sentencias de Control

Utilizando código de tercetos , vamos a generar ahora el código correspondiente no sólo a las expresiones, sino también el correspondiente a las sentencias de control.

En el caso de las expresiones, nos basábamos en las variables temporales para generar código. Ahora el problema son los cambios de flujos

IF - THEN- ELSE
CASE
WHILE
REPEAT

Vamos a trabajar con cambios de flujos mediante condiciones:

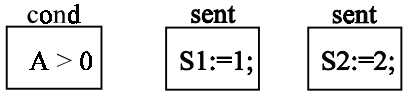
WHILE	⇨	Mientras se cumple la condición
REPEAT	⇨	Hasta que se cumpla la condición

Por ejemplo, si tenemos una instrucción como:

Entrada	Salida
IF A > 0 THEN	if A > 0 goto etq1
S1 := 1;	goto etq2
ELSE	label etq1
S2 := 2;	S1 = 1
FIN SI;	goto etq3
	label etq2
	S2 = 2
	label etq3

Ahora vamos a ver cómo vamos a generar el código anterior.

Primero identificamos los bloques principales de la entrada



la clave de todo está en la condición

En el momento en el que ponemos el no terminal *sent*, justo detrás está generado su código, según la regla de producción:

Generación de código de tercetos en Sentencias de Control

```
sent    :    IF cond THEN sent ELSE sent { Generar código de tercetos}
        |    ID ASIG expr {Generar código de tercetos}
```

Antes de reducir el IF, tiene que tener el código generado de las dos *sent*. De forma que en pantalla vamos a tener

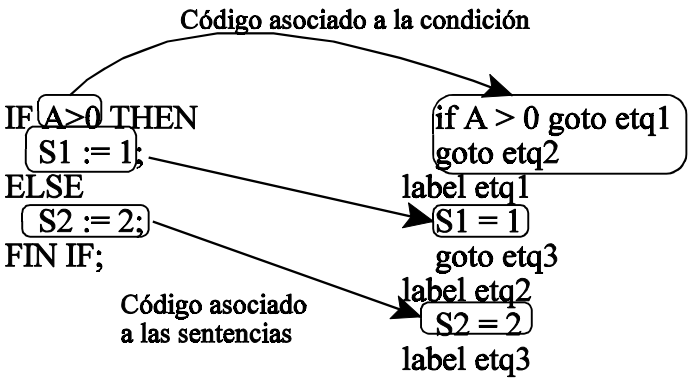
código generado sent 1 ⇨ S1 = 1
código generado sent 2 ⇨ S2 = 2

Esto es un problema porque entre las dos sentencias existe código

```
...
        S1 = 1              ⇦ Código generado por sent1
        goto etq3
label etq2
        S2 = 2              ⇦ Código generado por sent2
...
```

Hay que poner el código que existe entre las dos sentencias, esto podríamos hacerlo mediante la inclusión de reglas intermedias.

Otro problema que nos encontramos es que tenemos que generar el código asociado a la condición



cond : expr '>' expr {genero código de tercetos}
↑
printf ("if %s > %s goto etq_verdad", $1,$3);
printf("goto etq_falso");

Posteriormente lo veremos con más detenimiento.

Generación de código de tercetos en Sentencias de Control

Una pregunta que cabe plantearse es ¿cómo se generan las otras líneas que no están asociadas ni a la condición ni a las expresiones?. Estas líneas se generan en reglas intermedias, como ya dijimos anteriormente, pero ¿donde?

sent : IF cond THEN sent ELSE sent {generar código de tercetos}

↑ ↑ ↑ ↑ ↑ ↑
① ② ③ ④ ⑤ ⑥

El código *label etq1* se puede generar en ② ó en ③; *goto etq3* y *label etq2* se puede generar en ④ ó en ⑤; label etq3 sólo se puede generar en ⑥.

Por ejemplo, el trozo de código

```
        if A > 0 goto etq1
        goto etq2
label etq1      ←
        S1 = 1
        goto etq3
label etq2
        S2 = 2
label etq3
```

se puede poner entre ② ó ③, es decir, antes de *sent* y después de *cond*, ya que ese trozo se encuentra entre la condición y la sentencia primera. Supongamos que lo ponemos en ③

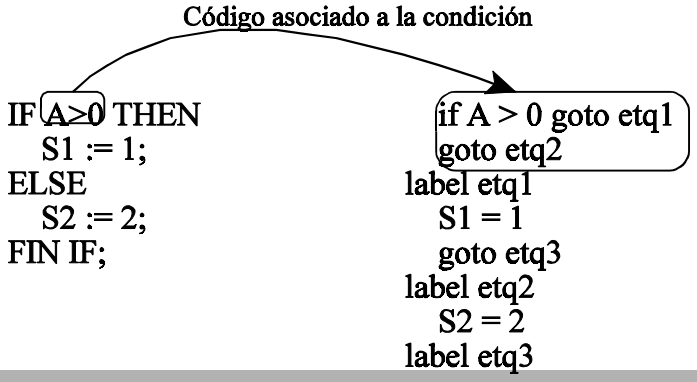
sent : IF cond THEN sent ELSE sent {generar código de tercetos}

↑

③ { *printf("label etq_verdad");* }

¿qué *etq_verdad* utilizo? - La que generó la condición

- **Condición Simple:** Veamos que tenemos que hacer cuando nos encontremos una condición



Generación de código de tercetos en Sentencias de Control

cond : expr '>' expr {generar código de tercetos}

↑

{strcpy($$.etq_verdad, nueva-etq( ));

strcpy($$.etq_falso, nueva-etq( ));
printf("if %s > %s goto %s, $1,$3,$$.etq_verdad");
printf("goto %s", $$.etq_falso);}

De manera que una condición tendrá asociada en el esquema de traducción a un par de atributos que serán dos etiquetas: una a la que se saltará en caso de que la condición sea cierta (etq_verdad), y otra cuando la condición sea falsa (etq_falso).

Una condición genera un bloque de código que es

```
if A>0 goto etq1
goto etq2
```

el cual tiene una cualidad muy importante, es un bloque que tiene dos goto (etq_verdad, etq_falso), pero no tiene los dos label's equivalentes

La regla que usa la condición puede hacer con los atributos lo que quiera, esto es, poner los label donde más convenga.

Hasta ahora la condición quedaría de la siguiente forma:

cond : expr '>' expr *{strcpy($$.etq_verdad, nueva-etq( ));*
strcpy($$.etq_falso, nueva-etq( ));
printf("if %s > %s goto %s, $1,$3,$$.etq_verdad");
printf("goto %s", $$.etq_falso);}

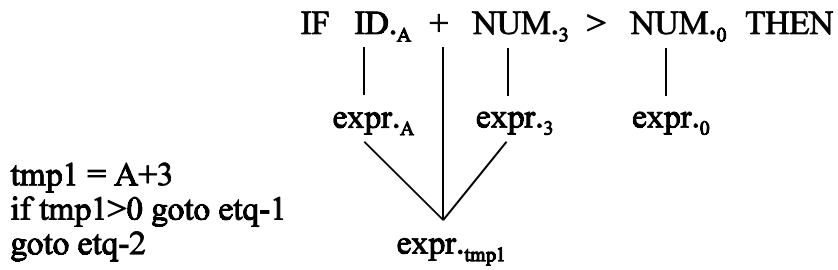
En el caso de una sentencia

sent : IF cond THEN sent ELSE sent FIN IF

cuando se llega a *sent*, ya se ha puesto la acción de condición, al igual que en *cond:expr'>'expr* cuando se llega a la acción ya tenemos que tener el valor de las dos expresiones (*expr*) en dos variables temporales.

Generación de código de tercetos en Sentencias de Control

Por ejemplo: IF A+3 > 0 THEN



Es decir, cuando llegamos a la acción que hay al final de *cond: expr '>' expr* todo el código asociado a expr ya se ha generado.

Con esto generamos código para una condición simple. Una condición compuesta es aquélla que tiene operaciones lógicas.

✯ **Condición compuesta :** Si se enlazan condiciones mediante operadores lógicos AND u OR emplearemos la técnica del cortocircuito, de manera que si enlazamos *cond1* y *cond2* con un AND, (*cond1 AND cond2*), pues si *cond1* es falso, no evaluaremos *cond2*, dado que su función será falsa sea cual sea el valor de *cond2*.
Si el conector es OR, (*cond1 OR cond2*), la *cond2* sólo se evaluará si la *cond1* es falsa, pues en caso contrario, su disyunción será verdad para cualquier valor de *cond2*.

Por ejemplo:

Sea :	Se traduce a :
IF (A >B) AND (B>C) THEN S1 := 1; FIN IF;	if A > B goto etq4 goto etq5 label etq4 if B>C goto etq6 goto etq7 label etq5 goto etq7 label etq6 S = 1 goto etq8 label etq7 label etq8

Generación de código de tercetos en Sentencias de Control

Cada vez que se reduce a una condición en base a expresiones, se genera el código

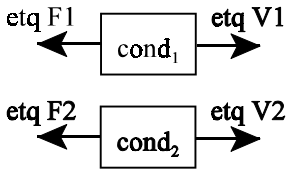
if arg1 op arg2 goto etq_verdad
goto etq_falso

En el momento en que nos encontramos con un operador lógico, sabemos que a continuación nos vamos a encontrar otra condición, que también generará un código con la misma estructura, así como otras dos etiquetas, una de certeza, y otra de falsedad. Ambas condiciones y el operador lógico, se reducirán a una condición. Ahora bien, dicha condición reducida deberá tener solo dos etiquetas, ¿qué etiquetas le asignamos?. La solución depende de la conectiva que se emplee.

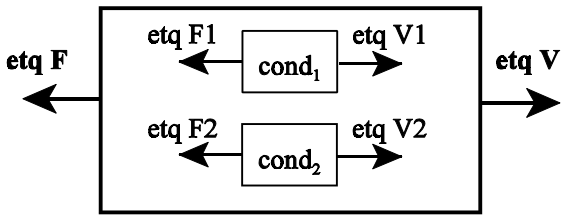
Veamos cada uno de los casos:

- *cond : cond AND cond* { }

Antes de la acción tengo que tener el código asociado a las dos condiciones. Esto me genera cuatro *goto* sin ningún *label*. Ahora tenemos que reducir los cuatro goto a dos goto, ya que una condición tiene que tener una etiqueta de verdad y otra de falso.
Si tengo: $cond_1$ *AND* $cond_2$ el código que tenemos es

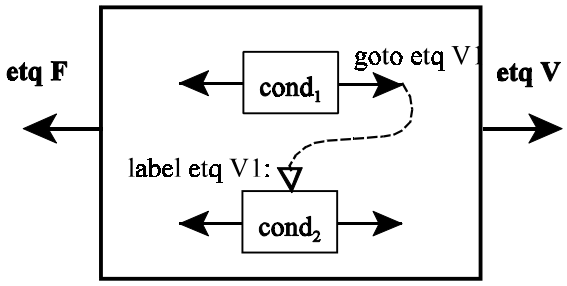


Mi objetivo es obtener un bloque de código que represente entera a *cond*. Así que ese código tiene que tener una sola *etq_v* y una sola *etq_f*

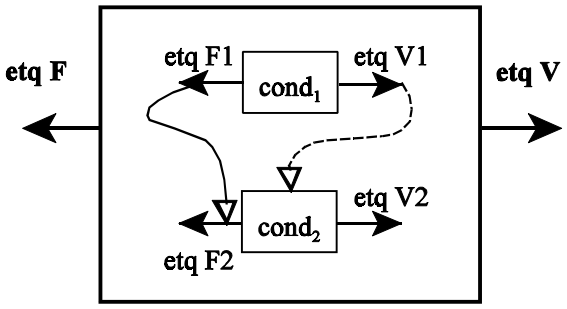


Como estamos en el caso de una conectiva AND, si la $cond_1$ es verdad, tengo que evaluar la $cond_2$.

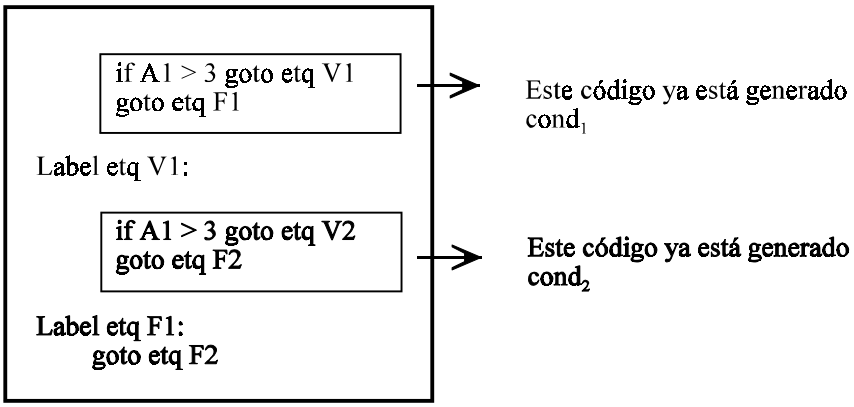
Generación de código de tercetos en Sentencias de Control



Si la cond1 es falso tengo que ir a la condición de falso de la cond2. ¿A donde irán las dos etiquetas de falso? - A la condición de falso de todo el bloque



Vamos a ver el código que se genera, por ejemplo



En éste código hay dos goto sin sus label, esto es una condición.¿Cuales son las etiquetas de verdad y de falso de esta condición grande?

etq_V = etq_V2
etq_F = etq_F2

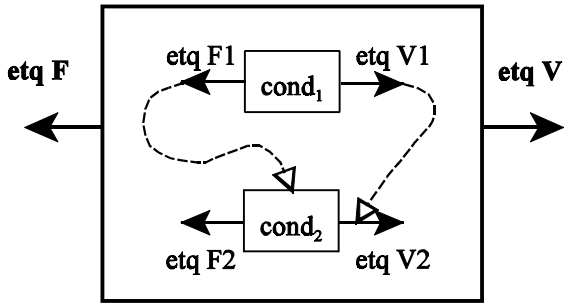
Generación de código de tercetos en Sentencias de Control

Una vez que sabemos todo lo que hay que hacer , vamos a ver qué código tendríamos que poner y en qué lugar.

```
          $1    $2        $3
cond : cond AND      { printf("label %s", $1.etq_V);}
          $4
          cond       { printf ("label %s", $1.etq_F);
                       printf ("goto %s",$4.etq_F);
                       strcpy($$.etq_V,$4.etq_V);
                       strcpy($$.etq_F,$4.etq_F);}
```

- *cond : cond OR cond*

Es igual, pero invirtiendo los papeles de las etiquetas.



¿Cuales son las etiquetas de verdad y de falso de esta condición?

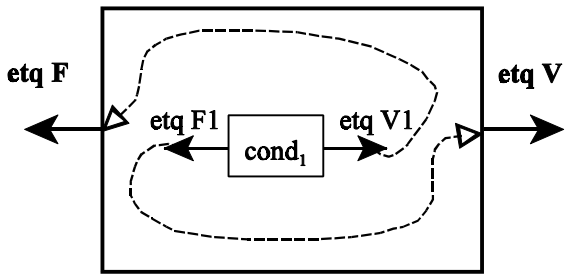
etq_V = etq_V2
etq_F = etq_F2

El código que tendríamos que añadir sería:

```
          $1    $2        $3
cond : cond AND      { printf("label %s", $1.etq_F);}
          $4
          cond       { printf ("label %s", $1.etq_V);
                       printf ("goto %s",$4.etq_V);
                       strcpy($$.etq_V,$4.etq_V);
                       strcpy($$.etq_F,$4.etq_F);}
```

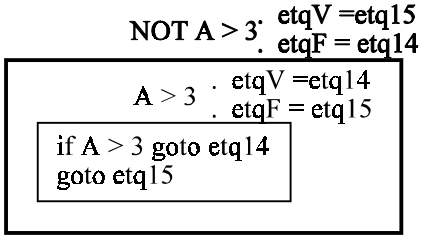
Generación de código de tercetos en Sentencias de Control

- cond : NOT cond



Ejemplo: NOT A>3

Supongamos que la etiqueta de verdad de A>3 es etq14, y que la etiqueta de falso de A>3 es etq15,



El código asociado sería:

cond : NOT cond { *strcpy($$.etqV,$2.etqF);*
strcpy ($$.etqF,$2.etqV); }

Siguiendo este criterio podemos hacer cualquier operador lógico. Por ejemplo : NAND

0 0 ⇨ V
0 1 ⇨ V
1 0 ⇨ V
1 1 ⇨ F

En este caso no se puede aplicar el método del cortocircuito, habría que evaluar las dos condiciones

Podemos, mediante este criterio, englobar cualquier tipo de condición porque hemos hecho un tratamiento uniforme.

✯ **Sentencias que nos permite nuestro lenguaje**

Ahora se trata de utilizar estas condiciones y sus etiquetas asociadas, para generar el código de sentencias que implican condiciones, como son IF, WHILE y REPEAT.

- *Sentencia IF-THEN-ELSE*

El caso del IF es el caso más simple. Aquí basta, con indicar que la etiqueta de verdad de la condición está asociada al código a continuación del THEN, y la etiqueta de falso se asocia al código que puede haber tras el ELSE. En cualquier caso, una vez acabadas las sentencias del THEN se debe producir un salto al final del IF, porque no queremos que se ejecuten también las sentencias del ELSE. Por tanto, tras la sentencias del THEN, creamos una nueva etiqueta a la cual produciremos un salto, y colocamos el destino de tal etiqueta al final del código del IF. Lo vemos con un ejemplo:

IF A > 0 THEN
  S1 := 1;
ELSE
  S2 := 2;
FIN IF
➜
	if A>0 goto etq1
	goto etq2
	label etq1
	S1 = 1
	goto etq3
	label etq2:
	S2 = 2
	label etq3

(Nota: No seguiremos fielmente la notación de C.)

{ *printf(label $2.etqV);*}
↓
sent : IF cond THEN sent ELSE sent FIN IF {*printf(label etqFINIF);*}
↑
{*printf (goto etqFINIF);*
printf (label $2.etqF);}

Problemas que nos encontramos. Observar que, la etiqueta FIN IF tiene que ser distinta para cada IF, por lo tanto tengo que tener una variable que sea un array de caracteres que guarde el nombre de la etiqueta FIN IF. Ahora bien, ¿Donde se declara esa variable?

sent : IF cond THEN sent ELSE sent FIN IF {*printf("label %s", etqFINIF);*}
↑
{ *char etqFINIF[20];*
strcpy(etqFINIF, nueva_etq( ));
printf ("goto %s",etqFIN IF);}

Esto **no** se podría hacer, porque *etqFINIF* es local a la acción en la que se declara, y el ámbito es esa acción, no puede utilizarse fuera.

Generación de código de tercetos en Sentencias de Control

¿Y si lo declaramos en YACC?

```
%{
char etqFINIF [20];
%}
...
%%
```

Esto funciona si sólo tenemos IF simples, si hay IF anidados da problemas:

Si sent es un IF entonces utiliza la etqFINIF mal.

⇓

sent : IF cond THEN sent ELSE sent FIN IF { *utilizo etqFINIF*}

↑

{ *utilizo etqFINIF*}

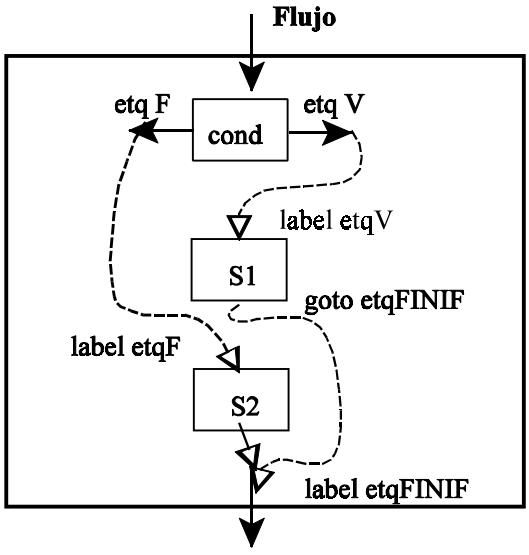
La solución estriba en hacer un truco, que consiste en asignarle al IF el atributo etqFINIF.

sent : IF.$_{etq}$ cond THEN sent ELSE sent FIN IF {*printf("label %s", $1.etq);*}

↑

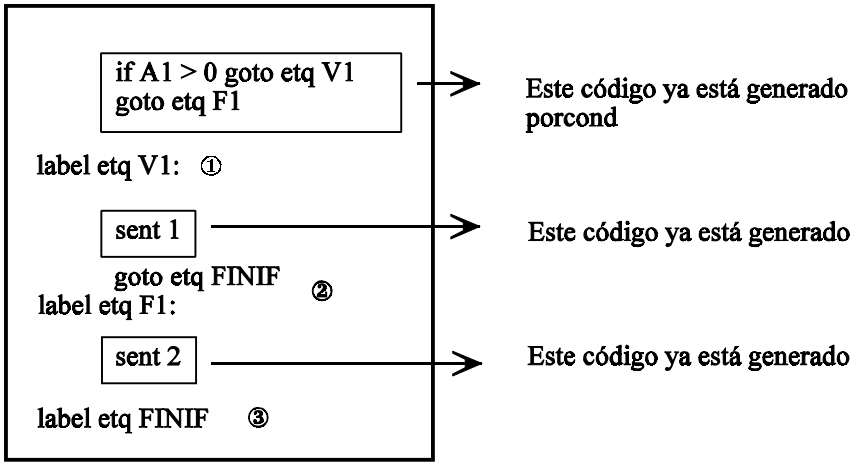
{ *strcpy($1.etq, nueva_etq( ));*
printf ("goto %s",$1.etq);}

Si hay un IF anidado, tendrá su etq asociada a su IF y no machacará el atributo de nuestro IF.
Vemos la sentencia IF en diagrama de bloques:



Generación de código de tercetos en Sentencias de Control

Veamos un ejemplo del código que generaría:



Vamos a ver donde hay que poner cada una de las acciones:

sent : IF cond THEN sent ELSE sent FIN IF
↑ ↑ ↑ ↑ ↑ ↑ ↑
a **b** c **d** e f **g** ③{*printf(“label %s”,$1.etq);*}
↑ ②{*strcpy($1.etq,etq_nueva( ));*
↑ *printf (“goto %s”,$1.etq);*
↑ *printf(“label %s”,$2.etqF);*}
①{*printf(“label %s”,$2.etqV);*}

- *Sentencia WHILE*

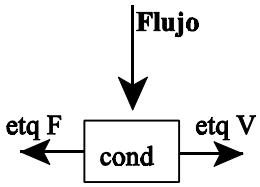
El caso de WHILE y REPEAT es muy similar. En ambos creamos una etiqueta al comienzo del bucle, a la que se saltará para repetir cada iteración.

En el caso del WHILE, a continuación se genera el código de la condición. La etiqueta de verdad se pone justo antes de las sentencias del WHILE, que es lo que se debe ejecutar si la condición es cierta. Al final de las sentencias se pondrá un salto al inicio del bucle, donde de nuevo se comprobará la condición. La etiqueta de falso de la condición, se pondrá al final de todo lo relacionado con el WHILE, o lo que es lo mismo, al principio del código generado para las sentencias que siguen al WHILE.

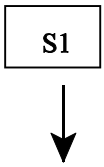
Generación de código de tercetos en Sentencias de Control

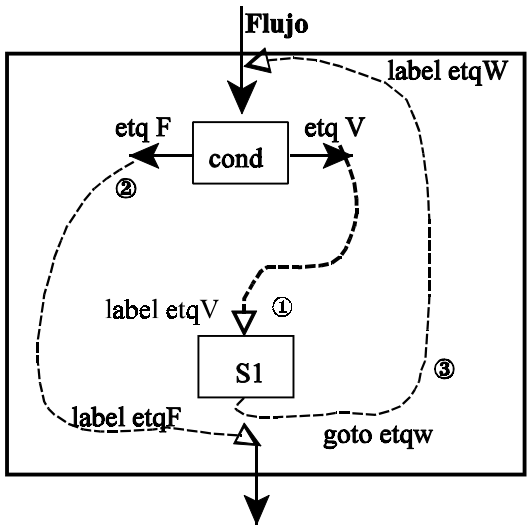
```
sent    : IF cond THEN sent ELSE sent FIN IF
        | WHILE cond DO sent FIN WHILE
                                  ↑
```

¿En este punto que tenemos generado?
- Código para la condición
- Código para la sentencia



Esto lo tenemos ya, sin hacer nada, ahora tenemos que empezar a poner flechas para que funcione como un WHILE





1º Tenemos que comprobar la condición.
2º Si etq es verdad ⇨ ejecuto la sentencia. ①
3º Si etq es falso ⇨ nos vamos del bucle. ②
4º Cuando acabe la ejecución de la sentencia, que vuelva otra vez a la condición ③

Vamos a ver el código que tendríamos que incluir. Para ello vamos a señalar con letras las posiciones donde podemos intercalar código

```
| WHILE   cond   DO   sent   FIN  WHILE
          ↑      ↑     ↑      ↑     ↑        ↑③{ printf("goto %s", $1.etqW);
          a      b     c      d     e        f  ② printf("label %s",$3.etqF);}
          ↑              ① {printf ("label %s", $2.etqV);}
          ③ {strcpy($1.etqW,nueva_etq());
             printf("label %s",$1.etqW);}
```

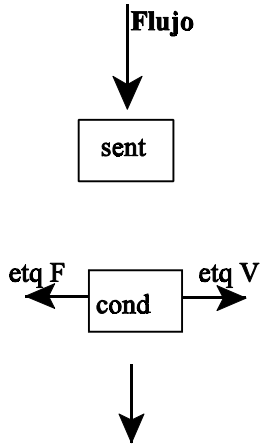
Generación de código de tercetos en Sentencias de Control

- *Sentencia REPEAT*

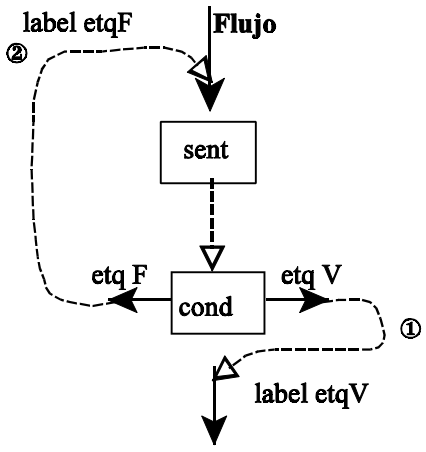
Como ya se dijo, creamos una etiqueta al comienzo del bucle, a la que se saltará para repetir cada iteración

En el caso del REPEAT, a continuación de la etiqueta de comienzo del bucle colocamos el código de las sentencias, ya que la condición se evalúa al final. Tras las sentencias colocamos el código de la condición. Ahora, debemos hacer coincidir la etiqueta de comienzo del bucle con la etiqueta de falso de la condición,. Como ambos nombres ya están asignados, la solución es colocar la etiqueta de falso, y en ella un goto al comienzo del bucle.
Al final de todo el código asociado al REPEAT pondremos la etiqueta de verdad.

```
sent    : IF cond THEN sent ELSE sent FIN IF
        | WHILE cond DO sent FIN WHILE
        | REPEAT  sent  UNTIL  cond
```



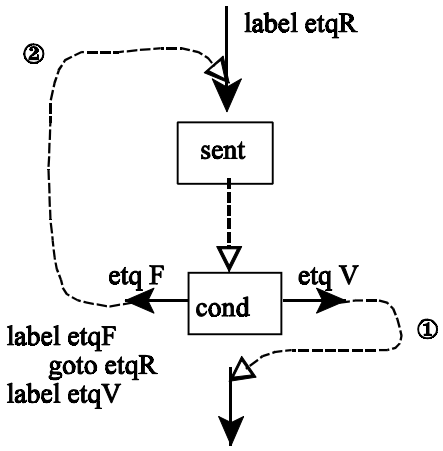
En este caso la sentencia se ejecuta siempre una vez, y después entra en la condición. Si la condición se cumple, salimos del bucle, y si no hacemos una nueva iteración.



Generación de código de tercetos en Sentencias de Control

| REPEAT sent UNTIL cond { printf (“label %s”, $4.etqV);}
↑ ↑ ↑ ↑
a b c d

En la posición **a** tendría que poner { printf(“label %s”, $4.etqF);}, pero en este punto no puedo utilizar $4, porque está por detrás de esta regla intermedia. Por lo tanto esto no puedo hacerlo así. Para solucionar este problema hacemos lo que se llama una indirección, esta etiqueta se podría optimizar posteriormente.



| REPEAT sent UNTIL cond { *printf (“label %s, $5.etqF”);* ②
↑ *printf(“goto %s”, $1.etqR);* ②
printf (“label %s”, $4.etqV);} ①
{*strcpy($1.etqR, nueva_etq( ));*
Printf (“label %s”, $1.etqR); }
(Esto es parte del ②)

- *Sentencia CASE*

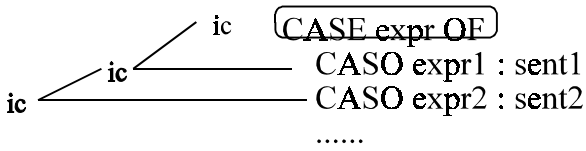
La sentencia más compleja de todas es la sentencia CASE, ya que ésta permite un número indeterminado, y potencialmente infinito de condiciones, lo cual obliga a arrastrar una serie de parámetros a medida que se van efectuando reducciones.

El problema es que la sentencia CASE necesita una recursión (no podemos utilizar una única regla de producción). Es necesario una regla de producción para la sentencia CASE diferente.

Generación de código de tercetos en Sentencias de Control

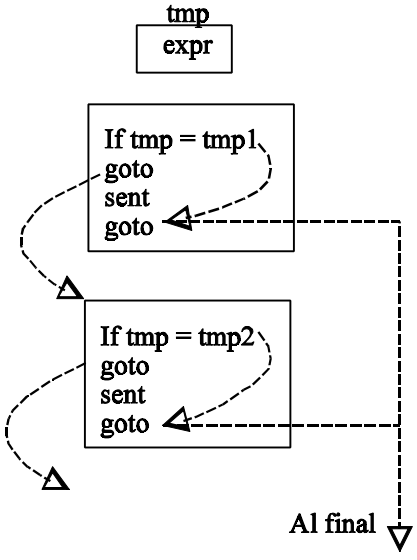
```
sent      : IF cond THEN sent ELSE sent FIN IF
          | WHILE cond DO sent FIN WHILE
          | REPEAT  sent  UNTIL  cond
          | sent_case
          ;

sent_case       : inicio_case FIN CASE
                | inicio_case OTHERWISE sent FIN CASE
                ;
inicio_case     : CASE expr OF
                | inicio_case CASO expr ':' sent
```

Esto me permite hacer cosas como: (ic ≡ inicio_case)



Hay que considerar que aunque en cada caso aparezca sólo una expresión, debemos convertirla en una comparación, tal que si es falsa se pase a comprobar la siguiente expresión, y si es cierta, se ejecutará el código asociado, al final del cual se producirá un salto al final del CASE.

Cada uno de los casos me va a generar un bloque de código diferente. Dentro de cada bloque todos los casos tiene la misma estructura.



Generación de código de tercetos en Sentencias de Control

La gramática que tenemos, permite cosas como CASE A OF FIN CASE, pero nosotros lo permitiremos. Si quisiéramos evitarlo tendríamos que poner una regla como:

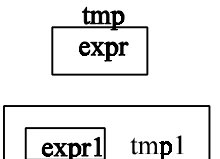
: CASE expr OF CASO expr ':' sent en lugar de CASE expr OF

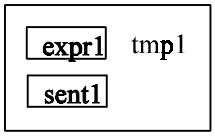
Nosotros no lo pondremos.

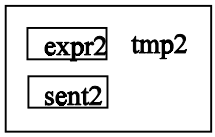
Si tenemos:

```
CASE A OF
        CASO 1 : S1;
        CASO 7: S2;
        .....
FIN CASE
```

el código que se genera sería :

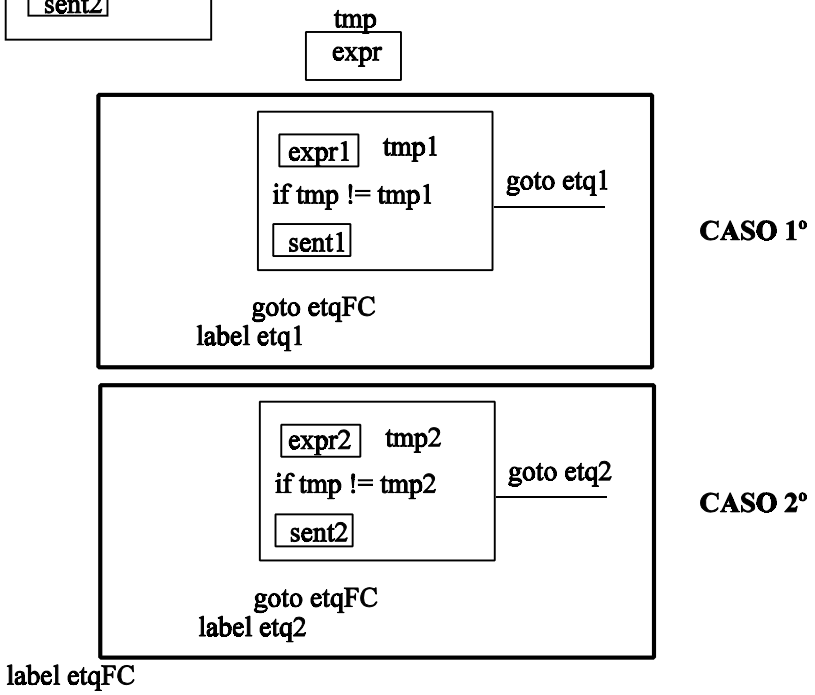






Todos los bloques del CASE tienen que ser iguales. ¿Qué me interesaría colocar para que el conjunto se comporte como una sentencia CASE?.

Se produce una comparación entre la expresión principal y la expresión subordinada, esto es siempre igual para todas.



Generación de código de tercetos en Sentencias de Control

¿Qué etiquetas he usado en el bloque?,¿ que es igual en todos, y que es diferente ?

Igual	Diferente
etqFC	tmp_x
tmp	etq_x

Cuando indicamos en los bloques algo que no es lo mismo, esto quiere decir que son variables locales.

tmp *Aquí haremos todo lo de las variables globales (etqFC)*

↓ ↓

inicio_ case : CASE expr OF { ---------- }

| inicio_case CASO expr ':' sent { ---------- }

↓ ↓

Común a todo *Aquí irán las variable locales. (Tmp_x, etq_x)*

Además propagaré lo que es global al otro inicio_case.

Cuando indicamos en los bloques algo que siempre es lo mismo, esto quiere decir que son globales.

$ic._{var=tmp,\ .etq=etqFC}$ { $ic._{var=tmp,\ .etq=etqFC}$ { $ic._{var=tmp,\ .etq=etqFC}$ { CASE A OF { $$.var = $2; $$.etq = nueva_etq( );}

CASO 1 : S1;

CASO 2 : S2;

Por tanto, el CASE debe tener asociado los siguiente atributos:

- Una variable temporal que representa a la expresión que se va a ir comparando.
- Una etiqueta que representa el final del CASE, etiqueta a la que se saltará tras poner el código asociado a la sentencia de cada CASO.

Generación de código de tercetos en Sentencias de Control

Nos queda colocar los correspondientes printf.

El *if tmp != tmp_1 goto etq_1* , donde lo colocamos: está antes de la sentencia, pero después de el código generado para expr.

```
inicio_ case    : CASE expr OF { strcpy ($$.var,$2);
                                 strcpy ($$.etq, nueva_etq ( )); }
                | inicio_case CASO expr ':' sent {strcpy ($$.var, $1.var);
                      ↑                  ↑       strcpy( $$.etq, $1.etq);
                      ❶                  a       printf("goto %s", $2.etq);
                                                 printf("label %s", $2.etq); }
```

En la posición **a** irá el siguiente código: (atributos de CASO)

```
{
  strcpy($2.etq, nueva_etq( ));
  printf ("if %s != %s goto %s", $1.var, $3, $2.etq);
}
```

que luego se utilizarán en la acción del final

El inicio_case ❶, es el que mantiene la variable tmp, y la etiqueta etq_FC.

¿Donde se pone el *label etq_FC*?

```
                  $1
sent_ case      : inicio_case OTHERWISE sent FIN CASE { printf("label %s",$1.etq);}
                  $1
                | inicio_case FIN CASE { printf("label %s", $1.etq); }
```

No es necesario colocar nada detrás del OTHERWISE, porque tal y como está hecho, después del último CASE, si no se cumple, se va al OTHERWISE.

Generación de código de tercetos. Gramática a reconocer

```
prog      :     prog sent ';'
          |     prog error ';'
          |
          ;
sent      :     ID ASIG expr
          |     IF cond THEN sent ';'
                      opcional
                FIN IF
          |     '{' lista_sent '}'     {;}
          |     WHILE cond DO
                      sent ';'
                FIN WHILE
          |     REPEAT
                      sent ';'
                UNTIL cond
          |     sent_case
          ;
opcional        :     ELSE sent';'
                |
                ;
lista_sent      :     /*Epsilon*/
                |     lista_sent sent ';'
                |     lista_sent error ';'  {yyerrok;}
                ;
sent_case       :     inicio_case
                      OTHERWISE sent ';'
                      FIN CASE
                |     inicio_case
                      FIN CASE
                ;
inicio_case     :     CASE expr OF
                |     inicio_case
                      CASO expr ':'      sent ';'
                ;
expr      :     NUMERO
          |     ID
          |     expr'+'expr
          |     expr'-'expr
          |     expr'*'expr
```

```
        |       expr'/'expr
        |       '-'expr %prec MENOS_UNARIO
        |       '('expr')'
        ;
0cond   :       expr'>'expr
        |       expr'<'expr
        |       expr'MAI'expr
        |       expr'MEI'expr
        |       expr'='expr
        |       expr'DIF'expr
        |       NOT cond
        |       cond AND cond
        |       cond OR cond
        |       '(' cond ')'
        ;
```

Código LEX: Ejem6l.lex

```
%{
int linea_actual = 1;
%}

%START COMENT

%%

^[ \t]*"*"      {BEGIN COMENT;}
<COMENT>.+      {;}
<COMENT>\n      {BEGIN 0; linea_actual++;}

":="    {return ASIG;}
">="    {return MAI;}
"<="    {return MEI;}
"!="    {return DIF;}
CASE    {return CASE;}
OF      {return OF;}
CASO    {return CASO;}
OTHERWISE       {return OTHERWISE;}
REPEAT  {return REPEAT;}
UNTIL   {return UNTIL;}
IF      {return IF;}
THEN    {return THEN;}
ELSE    {return ELSE;}
WHILE   {return WHILE;}
DO      {return DO;}
AND     {return AND;}
OR      {return OR;}
NOT     {return NOT;}
FIN     {return FIN;}
```

```
[0-9]+ {
        yylval.numero = atoi(yytext);
        return NUMERO;
        }

[A-Za-z_][A-Za-z0-9_]*      {
                strcpy(yylval.variable_aux, yytext);
                return ID;
                }
[ \t]+    {;}
\n        {linea_actual++;}
.         {return yytext[0];}
```

Código YACC: Ejem6y-2.yac

```
%{
typedef struct _doble_cond
        {
                char etq_verdad[21],
                     etq_falso[21];
        }doble_cond;

typedef struct _datos_case
        {
                char etq_final[21];
                char variable_expr[21];
        }datos_case;
%}

%union{
        int numero;
        char variable_aux[21];
        char etiqueta_aux[21];
        char etiqueta_siguiente[21];
        doble_cond bloque_cond;
        datos_case bloque_case;
}

%token <numero> NUMERO
%token <variable_aux> ID
%token <etiqueta_aux> IF WHILE REPEAT
%token <etiqueta_siguiente> CASO
%token ASIG THEN ELSE FIN DO UNTIL CASE OF OTHERWISE
%token MAI MEI DIF

%type <variable_aux> expr
%type <bloque_cond> cond
%type <bloque_case> inicio_case
```

```
%left OR
%left AND
%left NOT
%left '+"-'
%left '*"/'
%left MENOS_UNARIO
```

Código YACC: Ejem6y-2.yac

```
%%
prog    :       prog sent ';'
        |       prog error ';'    {yyerrok;}
        |
        ;
sent    :       ID ASIG expr {
                        printf("\t%s = %s\n",$1,$3);
                        }
        |       IF cond
                        {
                        printf("label %s\n",$2.etq_verdad);
                        }
                THEN sent ';'
                        {
                        nueva_etq($1);
                        printf("\tgoto%s\n",$1);
                        printf("label %s\n",$2.etq_falso);
                        }
                opcional
                FIN IF
                        {
                        printf("label %s\n",$1);
                        }
        |       '{' lista_sent '}'          {;}
        |       WHILE
                        {
                        nueva_etq($1);
                        printf("label %s\n",$1);
                        }
                cond
                        {
                        printf("label %s\n",$3.etq_verdad);
                        }
                DO sent ';'
                        {
```

```
                printf("\tgoto %s\n",$1);
                }
        FIN WHILE
                {
                printf("label %s\n",$3.etq_falso);
                }
```

Código YACC: Ejem6y-2.yac

```
    |   REPEAT
                {
                nueva_etq($1);
                printf("label %s\n",$1);
                }
        sent ';'
        UNTIL cond
                {
                printf("label %s\n",$6.etq_falso);
                printf("\tgoto %s\n",$1);
                printf("label %s\n",$6.etq_verdad);
                }
    |   sent_case
    ;

opcional    :   ELSE sent';'
            |
            ;

lista_sent  :   /*Epsilon*/
            |   lista_sent sent ';'
            |   lista_sent error ';'    {yyerrok;}
            ;

sent_case   :   inicio_case
                OTHERWISE sent ';'
                FIN CASE
                        {
                        printf("label %s\n",$1.etq_final);
                        }
            |   inicio_case
                FIN CASE
                        {
                        printf("label %s\n",$1.etq_final);
                        }
```

```
                ;
```

Código YACC: Ejem6y-2.yac

```
inicio_case     :       CASE expr OF
                                {
                                 strcpy($$.variable_expr,$2);
                                 nueva_etq($$.etq_final);
                                }
                |       inicio_case
                        CASO expr':'
                                {
                                 nueva_etq($2);
                                 printf("\tif %s != %s goto %s\n", $1.variable_expr,$3,$2);
                                }
                        sent ';'
                                {
                                 printf("\tgoto %s\n",$1.etq_final);
                                 printf("label %s\n",$2);
                                 strcpy($$.variable_expr,$1.variable_expr);
                                 strcpy($$.etq_final,$1.etq_final);
                                }
                ;
expr    :       NUMERO    {
                        nueva_var($$);
                        printf("\t%s = %d;\n", $$,$1);
                         }
        |       ID      {
                        strcpy($$,$1);
                        }
        |       expr'+'expr     {
                         nueva_var($$);
                         printf("\t%s = %s + %s;\n",$$,$1,$3);
                        }
        |       expr'-'expr     {
                         nueva_var($$);
                         printf("\t%s = %s - %s;\n",$$,$1,$3);
                        }
        |       expr'*'expr     {
```

```
                        nueva_var($$);
                        printf("\t%s = %s * %s;\n",$$,$1,$3);
                        }
        |       expr'/'expr     {
                        nueva_var($$);
                        printf("\t%s = %s / %s;\n",$$,$1,$3);
                        }
```

Código YACC: Ejem6y-2.yac

```
        |       '-'expr %prec MENOS_UNARIO     {
                        nueva_var($$);
                        printf("\t%s = - %s;\n",$$,$2);
                        }
        |       '('expr')'          {
                        strcpy($$,$2);
                        }
        ;
cond    :       expr'>'expr     {
                        nueva_etq($$.etq_verdad);
                        nueva_etq($$.etq_falso);
                        printf("\tif %s > %s goto %s\n",$1,$3,$$.etq_verdad);
                        printf("\tgoto %s\n",$$.etq_falso);
                        }
        |       expr'<'expr     {
                        nueva_etq($$.etq_verdad);
                        printf("\tif %s < %s goto %s\n",$1,$3,$$.etq_verdad);
                        nueva_etq($$.etq_falso);
                        printf("\tgoto %s\n",$$.etq_falso);
                        }
        |       expr MAI expr{
                        nueva_etq($$.etq_verdad);
                        nueva_etq($$.etq_falso);
                        printf("\tif %s >= %s goto %s\n",$1,$3,$$.etq_verdad);
                        printf("\tgoto %s\n",$$.etq_falso);
                        }
        |       expr MEI expr           {
                        nueva_etq($$.etq_verdad);
                        nueva_etq($$.etq_falso);
                        printf("\tif %s <= %s goto %s\n",$1,$3,$$.etq_verdad);
                        printf("\tgoto %s\n",$$.etq_falso);
                        }
        |       expr'='expr     {
                        nueva_etq($$.etq_verdad);
                        nueva_etq($$.etq_falso);
```

```
                printf("\tif %s = %s goto %s\n",$1,$3,$$.etq_verdad);
                printf("\tgoto %s\n",$$.etq_falso);
                }
        |       expr DIF expr {
                nueva_etq($$.etq_verdad);
                nueva_etq($$.etq_falso);
                printf("\tif %s != %s goto %s\n",$1,$3,$$.etq_verdad);
                printf("\tgoto %s\n",$$.etq_falso);
                }
```

Código YACC: Ejem6y-2.yac

```
        |       NOT cond      {
                strcpy($$.etq_verdad,$2.etq_falso);
                strcpy($$.etq_falso,$2.etq_verdad);
                }
        |       cond AND
                {
                printf("label%s\n",$1.etq_verdad);
                }
                cond    {
                printf("label %s\n",$1.etq_falso);
                printf("\tgoto %s\n",$4.etq_falso);
                strcpy($$.etq_verdad,$4.etq_verdad);
                strcpy($$.etq_falso,$4.etq_falso);
                }
        |       cond OR
                {
                printf("label %s\n",$1.etq_falso);
                }
                cond    {
                printf("label %s\n",$1.etq_verdad);
                printf("\tgoto %s\n",$4.etq_verdad);
                strcpy($$.etq_verdad,$4.etq_verdad);
                strcpy($$.etq_falso,$4.etq_falso);
                }
        |       '(' cond ')'        {
                strcpy($$.etq_verdad,$2.etq_verdad);
                strcpy($$.etq_falso,$2.etq_falso);
                }
        ;

%%

#include "ejem6l.c"

void main()
```

```
{
	yyparse();
}

void yyerror(char* s)
{
	fprintf(stderr, "Error de sintaxis en la linea %d\n", linea_actual);
}
```

Código YACC: Ejem6y-2.yac

```
void nueva_var(char * s)
{
  static actual=0;
	strcpy(s,&"tmp");
	itoa(++actual, &(s[3]),10);
}


void nueva_etq(char * s)
{
  static actual=0;
	strcpy(s,&"etq");
	itoa(++actual, &(s[3]),10);
}
```

Ejemplos de ejecución.

Sentencias a reconocer.

```
ALFA := n;
FACTORIAL := 1;
WHILE ALFA > 1 DO
 {
  FACTORIAL := FACTORIAL * ALFA;
  ALFA := ALFA - 1;
 };
FIN WHILE;
```

Código generado.

```
        ALFA = n
        tmp1 = 1;
        FACTORIAL = tmp1
label etq1
        tmp2 = 1;
        if ALFA > tmp2 goto etq2
        goto etq3
label etq2
        tmp3 = FACTORIAL * ALFA;
        FACTORIAL = tmp3
        tmp4 = 1;
        tmp5 = ALFA - tmp4;
        ALFA = tmp5
        goto etq1
label etq3
```

Ejemplos de ejecución.

Sentencias a reconocer.

```
CASE NOTA OF
 * Podemos emplear comentarios, dedicando una linea a cada uno de ellos.
 CASO 5 : CALIFICACION := SOBRESALIENTE;
 CASO 4 : CALIFICACION := NOTABLE;
 CASO 3 : CALIFICACION := APROBADO;
 CASO 2 : CALIFICACION := INSUFICIENTE;
OTHERWISE
 CALIFICACION := MUY_DEFICIENTE;
FIN CASE;
```

Código generado.

```
        tmp1 = 5;
        if NOTA != tmp1 goto etq2
        CALIFICACION = SOBRESALIENTE
        goto etq1
label etq2
        tmp2 = 4;
        if NOTA != tmp2 goto etq3
        CALIFICACION = NOTABLE
        goto etq1
label etq3
        tmp3 = 3;
        if NOTA != tmp3 goto etq4
        CALIFICACION = APROBADO
        goto etq1
label etq4
        tmp4 = 2;
        if NOTA != tmp4 goto etq5
        CALIFICACION = INSUFICIENTE
        goto etq1
label etq5
        CALIFICACION = MUY_DEFICIENTE
label etq1
```

Ejemplos de ejecución.

Sentencias a reconocer.

```
JUGAR := DESEO_DEL_USUARIO;
WHILE JUGAR = VERDAD DO
 {
  TOTAL := 64;
  SUMA_PUNTOS := 0;
  NUMERO_TIRADAS := 0;
  TIRADA_ANTERIOR := 0;
  REPEAT
   {
    DADO := RANDOMIZE * 5 + 1;
    IF TIRADA_ANTERIOR != 6 THEN
      NUMERO_TIRADAS := NUMERO_TIRADAS + 1;
    FIN IF;
    SUMA_PUNTOS := SUMA_PUNTOS + DADOS;
    IF SUMA_PUNTOS > TOTAL THEN
      SUMA_PUNTOS := TOTAL -(SUMA_PUNTOS - TOTAL);
    ELSE
      IF SUMA_PUNTOS != TOTAL THEN
        CASE DADO OF
          CASO 1: UNOS := UNOS + 1;
          CASO 2: DOSES := DOSES + 1;
          CASO 3: TRESES := TRESES + 1;
          CASO 4: CUATROS := CUATROS + 1;
          CASO 5: CINCOS := CINCOS + 1;
        OTHERWISE
              SEISES := SEISES + 1;
        FIN CASE;
      FIN IF;
    FIN IF;
    TIRADA_ANTERIOR := DADO;
   };
   UNTIL SUMA_PUNTOS = TOTAL;
   JUGAR := DESEO_DEL_USUARIO;
 };
FIN WHILE;
```

Ejemplos de ejecución.

Código Generado.

```
        JUGAR = DESEO_DEL_USUARIO
label etq1
        if JUGAR = VERDAD goto etq2
        goto etq3
label etq2
        tmp1 = 64;
        TOTAL = tmp1
        tmp2 = 0;
        SUMA_PUNTOS = tmp2
        tmp3 = 0;
        NUMERO_TIRADAS = tmp3
        tmp4 = 0;
        TIRADA_ANTERIOR = tmp4
label etq4
        tmp5 = 5;
        tmp6 = RANDOMIZE * tmp5;
        tmp7 = 1;
        tmp8 = tmp6 + tmp7;
        DADO = tmp8
        tmp9 = 6;
        if TIRADA_ANTERIOR != tmp9 goto
etq5
        goto etq6
label etq5
        tmp10 = 1;
        tmp11 = NUMERO_TIRADAS + tmp10;
        NUMERO_TIRADAS = tmp11
        gotoetq7
label etq6
label etq7
        tmp12 = SUMA_PUNTOS + DADOS;
        SUMA_PUNTOS = tmp12
        if SUMA_PUNTOS > TOTAL goto etq8
        goto etq9
label etq8
        tmp13 = SUMA_PUNTOS - TOTAL;
        tmp14 = TOTAL - tmp13;
        SUMA_PUNTOS = tmp14
        gotoetq10
label etq9
        if SUMA_PUNTOS != TOTAL goto
etq11
        goto etq12
label etq11
        tmp15 = 1;
        if DADO != tmp15 goto etq14
        tmp16 = 1;
        tmp17 = UNOS + tmp16;
        UNOS = tmp17
        goto etq13
label etq14
        tmp18 = 2;
        if DADO != tmp18 goto etq15
        tmp19 = 1;
        tmp20 = DOSES + tmp19;
        DOSES = tmp20
        goto etq13
label etq15
        tmp21 = 3;
        if DADO != tmp21 goto etq16
        tmp22 = 1;
        tmp23 = TRESES + tmp22;
        TRESES = tmp23
        goto etq13
label etq16
        tmp24 = 4;
        if DADO != tmp24 goto etq17
        tmp25 = 1;
        tmp26 = CUATROS + tmp25;
        CUATROS = tmp26
        goto etq13
label etq17
        tmp27 = 5;
        if DADO != tmp27 goto etq18
        tmp28 = 1;
        tmp29 = CINCOS + tmp28;
        CINCOS = tmp29
        goto etq13
label etq18
        tmp30 = 1;
        tmp31 = SEISES + tmp30;
        SEISES = tmp31
label etq13
        gotoetq19
label etq12
label etq19
label etq10
        TIRADA_ANTERIOR = DADO
        if SUMA_PUNTOS = TOTAL goto etq20
        goto etq21
label etq21
        goto etq4
label etq20
        JUGAR = DESEO_DEL_USUARIO
        goto etq1
label etq3
```