Procesadores de lenguajes. Práctica 12.

GENERACIÓN DE CÓDIGO INTERMEDIO EN EL COMPILADOR DE TINTO.

Índice

1.1 Objetivos

- Describir las instrucciones del código intermedio del compilador de Tinto
- Describir el proceso de generación de código intermedio en el compilador de Tinto.

- Las clases asociadas a la generación de código intermedio en el compilador de Tinto están contenidas en el paquete tinto.code

1.2 El código intermedio del compilador de Tinto

- Compilador de Tinto utiliza un código intermedio propio formado por 23 instrucciones.
- Código de tres direcciones que se almacena en los campos *target*, *source1* y *source2*.
- Las instrucciones son:
  - LABEL: Etiqueta a la que se puede saltar. Campo *target* identificador de la etiqueta.
  - ASSIGN: Instrucción de asignación. Campo *target* almacena la referencia a la variable destino y *source1* almacena el origen de la asignación, que puede ser una referencia a una variable o un valor constante.
  - ADD: Instrucción se suma. Campo *target* almacena la referencia a la variable destino y los campos *source1* y *source2* almacena los operandos.
  - SUB: Describe una instrucción de resta. Campo *target* almacena la referencia a la variable destino y los campos *source1* y *source2* almacena los operandos.
  - MUL: Describe una instrucción de multiplicación. Campo *target* almacena la referencia a la variable destino y los campos *source1* y *source2* almacena los operandos.

1.2 El código intermedio del compilador de Tinto

- Las instrucciones son:
  - DIV: Describe una instrucción de división. Campo *target* almacena la referencia a la variable destino y los campos *source1* y *source2* almacena los operandos.
  - MOD: Describe una instrucción de módulo. Campo *target* almacena la referencia a la variable destino y los campos *source1* y *source2* almacena los operandos.
  - INV: Describe una instrucción de cambio de signo. Campo *target* almacena la referencia a la variable destino y el campo *source1* almacena el operando.
  - AND: Describe una instrucción de conjunción lógica (and). Campo *target* almacena la referencia a la variable destino y los campos *source1* y *source2* almacena los operandos.
  - OR: Describe una instrucción de disyunción lógica (or). Campo *target* almacena la referencia a la variable destino y los campos *source1* y *source2* almacena los operandos.
  - NOT: Describe una instrucción de negación lógica (not). Campo *target* almacena la referencia a la variable destino y el campo *source1* almacena el operando.

1.2 El código intermedio del compilador de Tinto

- Las instrucciones son:
  - JUMP: Describe una instrucción de salto incondicional. Campo *target* almacena el identificador de la etiqueta de salto.
  - JMPEQ: Describe una instrucción de salto condicionado a la igualdad entre los operandos. El campo *target* almacena la etiqueta de salto. Los campos *source1* y *source2* contienen las referencias a los datos que hay que comparar. Si *source1* es igual que *source2* se producirá el salto.
  - JMPNE: Describe una instrucción de salto condicionado a la desigualdad entre los operandos. El campo *target* almacena la etiqueta de salto. Los campos *source1* y *source2* contienen las referencias a los datos que hay que comparar. Si *source1* es distinto a *source2* se producirá el salto.
  - JMPGT: Describe una instrucción de salto condicionado a la comparación "mayor". El campo *target* almacena la etiqueta de salto. Los campos *source1* y *source2* contienen las referencias a los datos que hay que comparar. Si *source1* es mayor que *source2* se producirá el salto.

1.2 El código intermedio del compilador de Tinto

- Las instrucciones son:
  - JMPGE: Describe una instrucción de salto condicionado a la comparación "mayor o igual". El campo *target* almacena la etiqueta de salto. Los campos *source1* y *source2* contienen las referencias a los datos que hay que comparar. Si *source1* es mayor o igual que *source2* se producirá el salto.
  - JMPLT: Describe una instrucción de salto condicionado a la comparación "menor". El campo target almacena la etiqueta de salto. Los campos source1 y source2 contienen las referencias a los datos que hay que comparar. Si source1 es menor que source2 se producirá el salto.
  - JMPLE: Describe una instrucción de salto condicionado a la comparación "menor o igual". El campo target almacena la etiqueta de salto. Los campos source1 y source2 contienen las referencias a los datos que hay que comparar. Si source1 es menor o igual que source2 se producirá el salto.
  - JMP1: Describe una instrucción de salto condicionado a un valor booleano. El campo target almacena la etiqueta de salto. El campo source1 contiene la referencia al dato booleano. Si source1 es 1 (codificación de true) se producirá el salto.

1.2 El código intermedio del compilador de Tinto

- Las instrucciones son:
  - PARAM: Describe una instrucción de almacenamiento de un parámetro de llamada a una función. Antes de realizar la llamada a la función es necesario evaluar los parámetros y almacenar sus valores. El campo *target* contiene el valor a almacenar. El campo *source1* contiene el índice del parámetro, es decir, su posición en la llamada a la función.
  - CALL: Describe una instrucción de llamada a una función. El campo source1 contiene el identificador de la función (la etiqueta de comienzo de la función). El campo target indica la variable en la que se almacenará el valor de retorno de la función. Los parámetros de la llamada deben ser almacenados previamente con instrucciones PARAM.
  - RETURN: Describe una instrucción de retorno de una función. El campo target contiene la referencia al dato a devolver.
  - PRECALL: Describe una instrucción de preparación de una llamada a función. El campo target contiene el número de parámetros que tendrá la llamada. El proceso de llamada a una función comienza indicando el número de parámetros de la llamada (PRECALL), a continuación, se calculan los valores de los parámetros y se almacenan en su posición (PARAM) y por último se realiza la llamada (CALL).

1.3 Representación de instrucciones en código intermedio

tinto.code contiene las siguientes clases dedicadas a representar las instrucciones en código intermedio:

- tinto.code.CodeConstants: Se trata de una interfaz que define las constantes que identifican los tipos de instrucciones del código intermedio.
- tinto.code.CodeAddress: Clase abstracta que describe una dirección en el código de tres direcciones. Las direcciones utilizadas en las instrucciones pueden ser etiquetas, literales (valores constantes) y referencias a variables.
- tinto.code.CodeLabel: Subclase de CodeAddress. Define una etiqueta que puede ser utilizada como campo en las instrucciones del código intermedio.
- tinto.code.CodeLiteral:Subclase de CodeAddress. Define un valor constante (un literal) que puede ser utilizado como campo en las instrucciones del código intermedio. El valor de la constante se almacena en formato hexadecimal.

1.3 Representación de instrucciones en código intermedio

tinto.code contiene las siguientes clases dedicadas a representar las instrucciones en código intermedio:

- tinto.code.CodeVariable: Subclase de CodeAddress. Define una referencia a una variable que puede ser utilizada como campo en las instrucciones del código intermedio. La generación de código intermedio transforma los nombres de las variables para evitar las duplicidades. El nombre original de la variable (descrita inicialmente en un objeto tinto.ast.struct.Variable) se almacena en el campo des. El nuevo nombre se almacena en el campo name. Los nombres de las variables en código intermedio son "local_i" (para las variables locales), "arg_i" (para los argumentos) y "tmp_i" (para las variables temporales). La clase contiene también información necesaria para el generador de código ensamblador (posición que ocupará en el registro de activación o registro del procesador en el que se almacenará).

- tinto.code.CodeInstruction: Clase que representa una instrucción en código intermedio. El campo kind indica el tipo de instrucción (alguno de los códigos definidos en CodeConstants). Los campos target, source1 y source2 contienen las tres direcciones de la instrucción. El campo comment se utiliza para generar un comentario asociado a la instrucción que pretende describir la instrucción en un formato más interpretable.

- tinto.code.CodeInstructionList: Describe una lista de instrucciones. Esta clase facilita la generación del código intermedio al desarrollar la concatenación de listas de instrucciones o la inclusión de nuevas instrucciones a la lista.

1.4 Generación de código intermedio

El analizador semántico del compilador de Tinto genera como resultado una clase **tinto.ast.struct.LibraryDeclaration** que almacena toda la información reconocida en un archivo fuente escrito en lenguaje Tinto. El objetivo del generador de código intermedio es transformar esta representación en una nueva representación basada en instrucciones de código intermedio. El resultado de la generación será un nuevo objeto **tinto.code.LibraryCodification**.

La clase **tinto.code.LibraryCodification** contiene los campos *name*, que almacena el nombre de la biblioteca, *imported*, con la lista de bibliotecas importadas, y *function*, que almacena una representación de las funciones que ya han sido transformadas a código intermedio. Además de los métodos *get..()* y *set..()*, la clase contiene el método *print()* que genera una descripción del código intermedio sobre un archivo.

1.4 Generación de código intermedio

La clase **tinto.code.FunctionCodification** describe el código intermedio asociado a una función de una biblioteca escrita en Tinto. El campo *label* almacena la etiqueta que identifica a la función y que coloca como etiqueta inicial en el código. El campo *type* indica el tipo de dato que devuelve la función. Los campos *labelcount* y *tmpcount* son contadores que se utilizan para generar nombres de etiquetas y variables temporales al generar el código. Los campos *arg* y *var* contienen las referencias a los argumentos y variables (locales y temporales) de la función. El campo *astvar* mantiene la desipción de variables contenida en el árbol de sintaxis abstracta. Esta lista se mantiene para poder localizar el objeto **CodeVariable** asociado a cada objeto **Variable** del árbol de sintaxis abstracta. Por último, el campo *list* contiene la lista de instrucciones que desarrolla la función en código intermedio. Además de los métodos *get..()* y *set..()* la clase contiene el método *print()*, que genera la representación de la función sobre un archivo, y el método *getFrameSize()*, utilizado en la generación de código ensamblador para generar el registro de activación de la función.

1.4 Generación de código intermedio

La clase **tinto.code.CodeGenerator** es la encargada de generar el código intermedio en el compilador de Tinto. El método *generateLibraryCodification()* es el único método público de la clase y se encarga de crear un objeto **LibraryCodification** a partir de un objeto **Library**. Para ello realiza llamadas al método *generateFunctionCode()*, que crea objetos **FunctionCodification** a partir de objetos **Function**. La clase contiene numerosos métodos que se dedican a generar el código intermedio asociado a las diferentes estructuras del árbol de sintaxis abstracta. Entre estos métodos destacan lo métodos genéricos para generar el código asociado a una instrucción, a una expresión y a una condición. El método *generateCodeOfStatement()* obtiene el código asociado a una instrucción en forma de objeto **CodeInstructionList**. El método *generateCodeForExpression()* almacena el código asociado a una expresión en el parámetro *codelist* y devuelve la referencia a la variable que almacena el resultado de la expresión. El método *generateCodeForCondition()* obtiene el código asociado a una condición y requiere el paso de las etiquetas *LabelTrue* y *LabelFalse* como argumentos de la llamada.

1.4 Generación de código intermedio

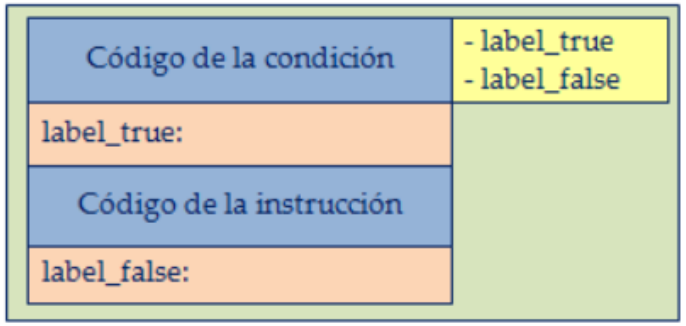
Para que el compilador genere los archivos de código intermedio se ha modificado la clase **tinto.TintoCompiler**. En esta clase se ha incluido un nuevo método *createCode()* que genera el código intermedio asociado a un objeto **LibraryDeclaration** y lo almacena como un archivo con la extension ".tmc". El método *main()* de la clase se ha ampliado para crear un objeto **CodeGenerator** y generar el código de todas las bibliotecas importadas en la tabla de símbolos.

1.5 Estructuras del código

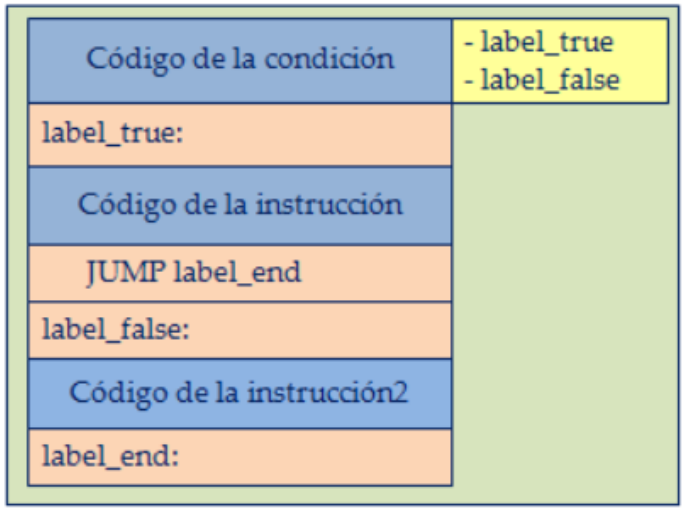
- La generación de código intermedio hace un recorrido por las instrucciones de cada función (objetos **Statement** definidos en el cuerpo de los objetos **Function**) generando una traducción de estas instrucciones a código intermedio (**CodeInstructionList**). Para realizar esta traducción en ocasiones es necesario crear nuevas etiquetas de salto (llamadas a la función *getNewLabel()*) o nuevas variables temporales (llamadas a *getNewTemp()*). A continuación se describe la estructura que tiene el código intermedio que se genera como traducción a las diferentes instrucciones, expresiones y condiciones utilizadas en Tinto.

1.5 Estructuras del código

- **Estructura asociada a la instrucción IF-THEN**

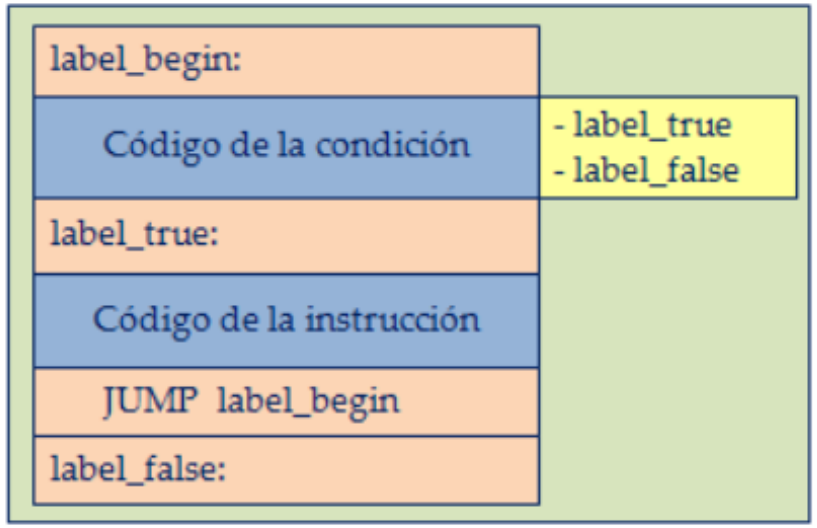


- **Estructura asociada a la instrucción IF-THEN-ELSE**

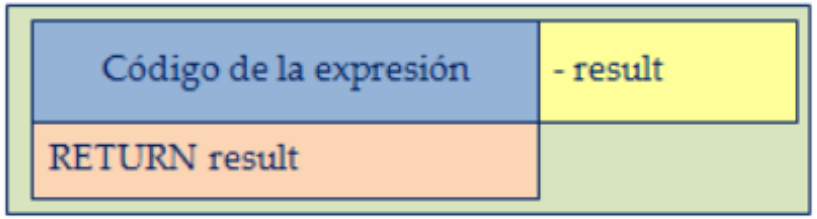


1.5 Estructuras del código

- **Estructura asociada a la instrucción WHILE**

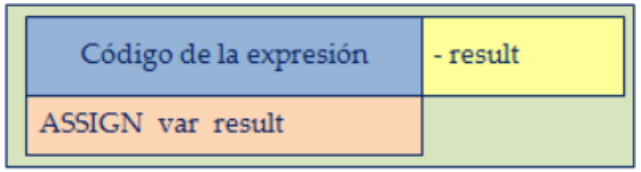


- **Estructura asociada a la instrucción RETURN**



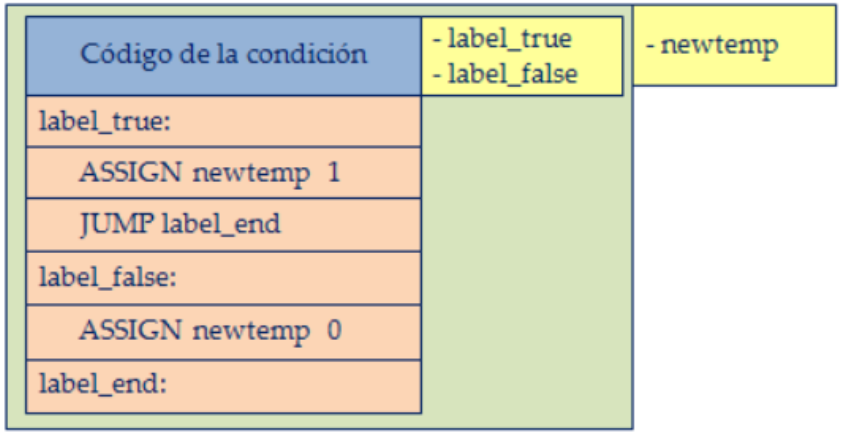
1.5 Estructuras del código

- **Estructura asociada a la instrucción ASIGNACIÓN**



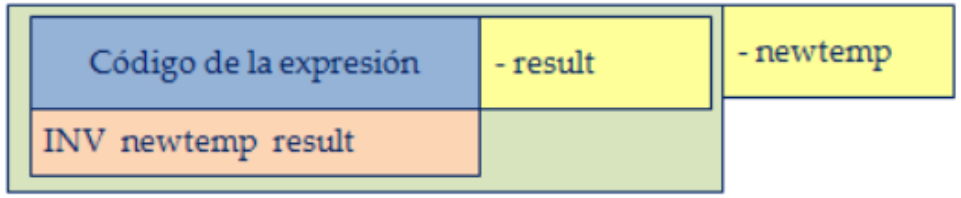
- **Estructura asociada a la expresión LITERAL (de tipo *int*, *char* o *boolean*)**

- **Estructura asociada a una expresión booleana**

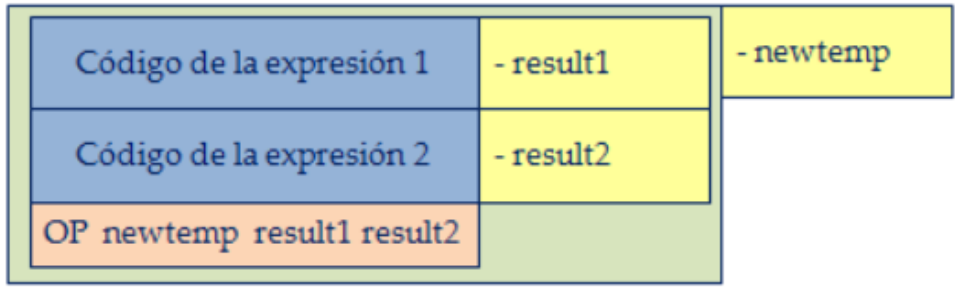


1.5 Estructuras del código

- **Estructura asociada a una expresión aritmética unaria (cambio de signo)**



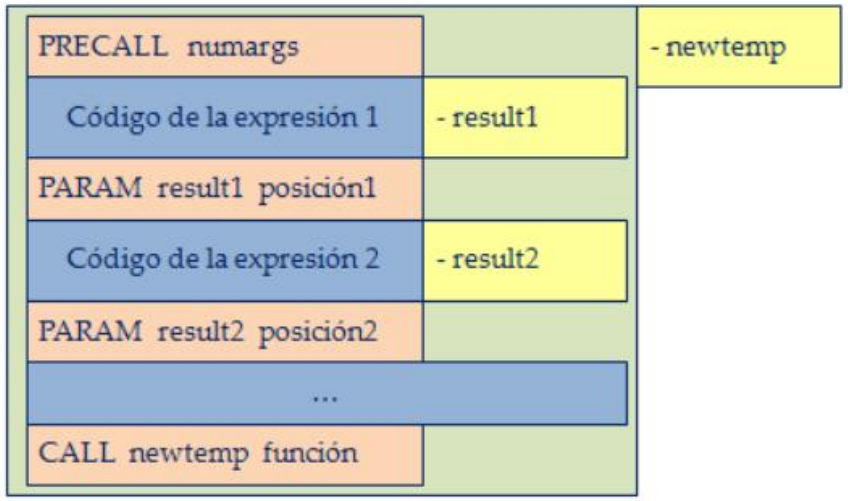
- **Estructura asociada a una expresión aritmética binaria**



La instrucción *op* puede ser ADD, SUB, MUL, DIV o MOD.

1.5 Estructuras del código

- **Estructura asociada a una expresión de llamada a una función**

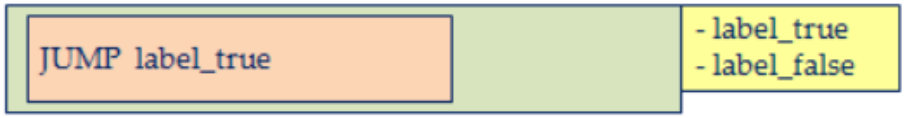


La instrucción PRECALL reserva la memoria para los argumentos de la llamada. Cada argumento debe generar el código asociado al cálculo de su valor y una instrucción PARAM. La instrucción CALL contiene la etiqueta de inicio de la función a la que se llama y la variable en la que se almacena el resultado de la función

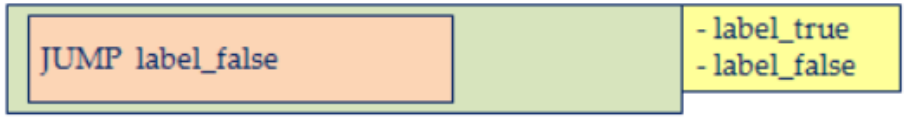
1.5 Estructuras del código

- **Estructura asociada a una expresión de referencia a una variable**

- **Estructura asociada a una condición formada por el literal TRUE**

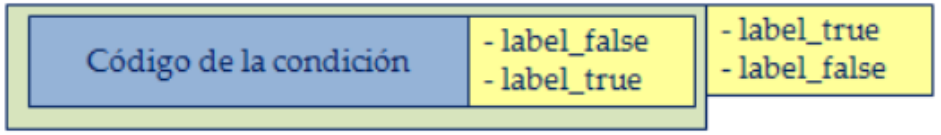


- **Estructura asociada a una condición formada por el literal FALSE**

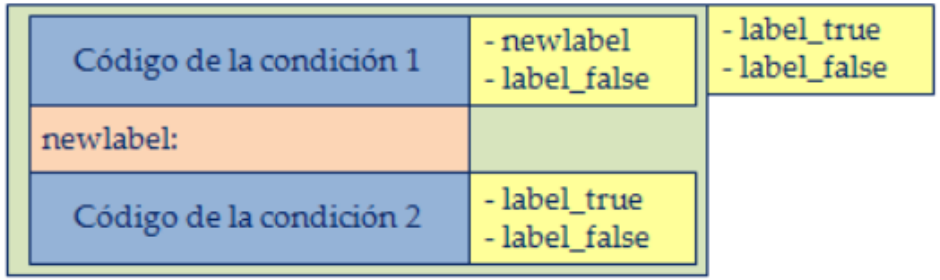


1.5 Estructuras del código

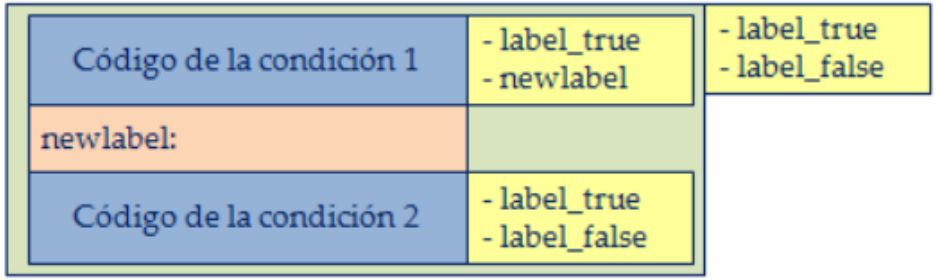
- **Estructura asociada a una condición formada la negación de otra condición**



- **Estructura asociada a una condición formada el AND entre dos condiciones**

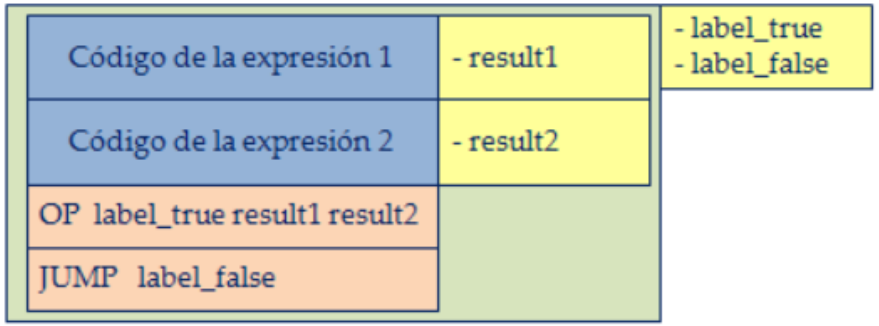


- **Estructura asociada a una condición formada el OR entre dos condiciones**



1.5 Estructuras del código

- **Estructura asociada a una condición formada por la comparación entre dos expresiones**



La instrucción *op* puede ser JMPEQ, JMPNE, JMPGT, JMPGE, JMPLT o JMPLE.

- **Estructura asociada a una condición formada por una expresión booleana (una variable o una llamada a una función)**