[75.27] Algoritmos y Programación IV

María Inés Parnisari

18 de julio de 2014

Índice

1 Introducción a la programación estructurada

Programación estructurada paradigma de programación orientado a mejorar la claridad, calidad y tiempo de desarrollo de un programa de computadora, utilizando únicamente subrutinas y tres estructuras: secuencia, selección (*if* y *switch*) e iteración (bucles *for* y *while*).

El **teorema del programa estructurado** demuestra que todo programa puede escribirse utilizando únicamente las tres instrucciones de control siguientes:

1. Secuencia: ejecución ordenada de sentencias.
2. Selección (*if / else*, *switch*)

```
program-id. Seleccion.

data division.
working-storage section.
01  nro     pic     S999.

procedure division.

   ACCEPT nro.
   IF (nro < 0) THEN
       DISPLAY "Numero negativo"
   ELSE IF (nro > 0) THEN
       DISPLAY "Numero positivo"
   ELSE
       DISPLAY "Cero"
   END-IF.

  EVALUATE nro
    WHEN 0
      DISPLAY "Cero"
    WHEN 1
      DISPLAY "Uno"

   DISPLAY "Presione ENTER para salir".
   ACCEPT nro.
```

3. Iteración (*do...while*)

```
program-id. Iteracion.

data division.
working-storage section.
01 I        PIC     99.

procedure division.

   MOVE 1 TO I.
   PERFORM ciclo UNTIL I > 10.

ciclo.
   DISPLAY I.
   ADD 1 TO I.
```

1.1 Diagramas de Jackson

- Cada cuadrado es un párrafo.
- Cada cuadrado con un asterisco es una instrucción de tipo `perform`.
- Cada cuadrado con dos barras en la esquina superior derecha es una llamada a otro módulo.

- Cada cuadrado con un círculo en la esquina superior derecha es una sentencia `if`.
- Cada cuadrado con una línea diagonal es una sentencia `else`.

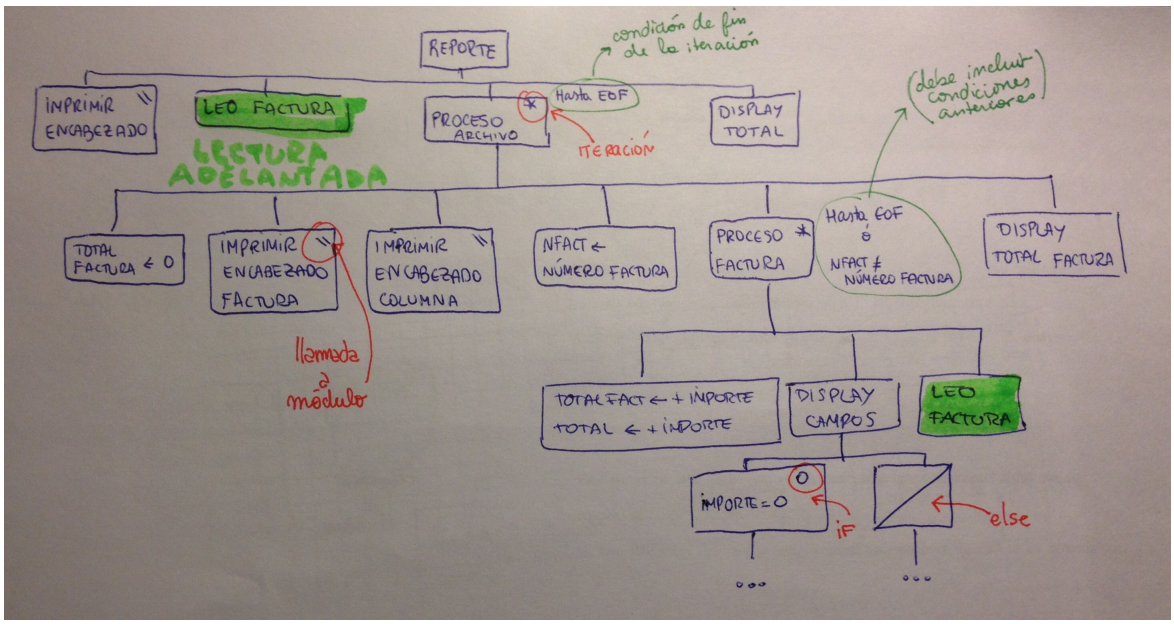

Figura 1: Ejemplo de diagrama de Jackson

2 Archivos secuenciales

Se leen con la sentencia `read`.

3 COBOL (COmmon Business Oriented Language)

- Año 1959
- Es un lenguaje compilado
- Sentencias deben finalizar con un punto.

El programa se divide en **divisions**, **sections** y **párrafos**. Una division está formada por una o más sections. Una section puede tener más de un párrafo.

```
IDENTIFICATION DIVISION.
program-id. Programa.

ENVIRONMENT DIVISION.
configuration section.
[special names
  decimal point is comma.]

input-output section.
file-control.
  select nom-arch
  assign to disk "path\al\archivo"

DATA DIVISION.
file section.

working-storage section.

linkage section.

PROCEDURE DIVISION.
```

```
  perform nom-parrafo until ...
  stop run.

nom-parrafo.
  ...
```

3.1 Declaración de variables

- X: caracter
- 9: número

3.2 Instrucciones

- `move corresponding id1 to id2`
  Mueve los campos del mismo nombre del grupo `id1` al grupo `id2`

4 Merge y apareo de archivos

4.1 Merge simple

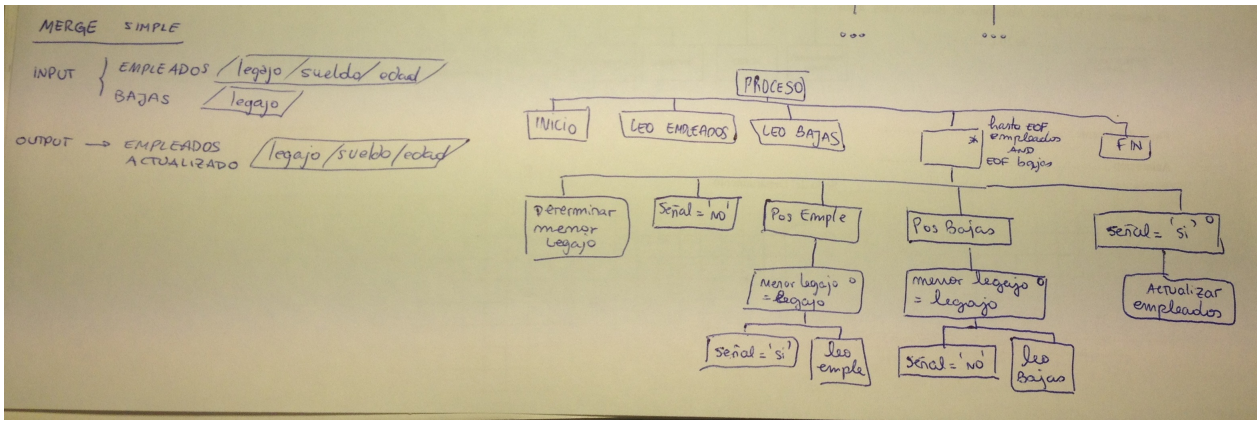

Figura 2: Merge simple

4.2 Merge compuesto

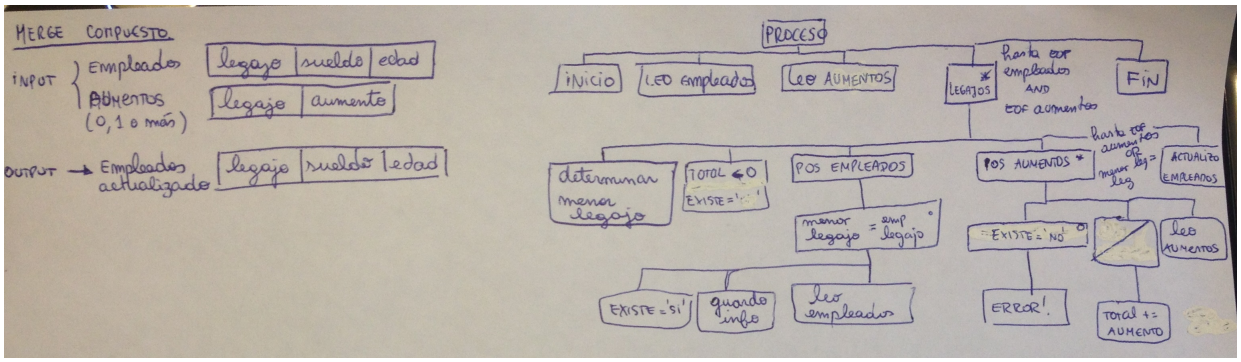

Figura 3: Merge compuesto

5 Tablas

- Definición de una tabla con 10 celdas.

```
01 tabladnis occurs 10 times.
    03  tipo    pic x.
    03  numero  pic 9(8).
```

- Carga de una tabla.

```
       identification division.
       program-id. Tablas.

       environment division.
       configuration section.
       file-control.
           select arch
           assign to disk "..\Archivos\dnis.txt"
           organization is line sequential
           file status is fs-arch.

       data division.
       file section.
       fd arch.
       01  registro.
           03  tipo    pic x.
           03  numero  pic 9(8).

       working-storage section.
       01 tabladnis occurs 10 times.
            03  tipo    pic x.
            03  numero  pic 9(8).

       01  indice      pic 99.
       01  fs-arch     pic xx.
           88  ok-arch    value "00".
           88  eof-arch   value "10".
       01  exitval        pic x.

       procedure division.
           open input arch.
           perform leerarch.
           move 1 to indice.
           perform cargar until eof-arch or indice > 9.
           close arch.
           accept exitval.

       cargar.
           move corresponding registro to tabladnis(indice).
           display tabladnis(indice).
           add 1 to indice.
           perform leerarch.

       leerarch.
           read arch.
           if (not ok-arch)
               display "ERROR␣AL␣LEER:" fs-arch
           end-if.
```

- Búsqueda en una tabla. Se requiere el uso de la sentencia INDEXED BY.
  - Si la tabla no está ordenada: verbo SEARCH (búsqueda lineal).
  - Si la tabla está ordenada: verbo SEARCH ALL (búsqueda binaria). Se requiere el uso de la sentencia KEY IS. Los elementos sin valores deben rellenarse con HIGH-VALUES para que esto funcione bien (búsqueda ascendente).

```
working-storage section.
01 tabladnis occurs 10 times
       ascending key is numero
       indexed by indice.
       03  tipo    pic x.
       03  numero  pic 9(8) value 99999999.
       03  nombre  pic x(30).

01  indicecarga      pic 99.
01  exitval        pic x.
01  in-dni         pic 9(8).
```

```
procedure division.
    open input arch.
    perform leerarch.
    move 1 to indicecarga.
    perform cargar until eof-arch or indicecarga > 9.
    close arch.
    display "Ingrese un dni para obtener el nombre: ".
    accept in-dni.

    initialize indice.
    search all tabladnis
        at end perform noencontrado
        when (numero(indice)) equals in-dni perform
        mostrarnombre
    end-search.

    accept exitval.
    stop run.
```

6 Archivos indexados

- Declaración de un archivo indexado

```
environment division.
file-control.

  select arch
  assign to disk "path\al\archivo"
  organization is indexed
  access mode is [sequential, dynamic, random]
  record key is nombre-clave-primaria
  alternate record key is nombre-clave-secundaria [with duplicates]
  file status is fs-arch
```

- Acceso secuencial (solo por clave primaria)

```
read nombrearch record.
if (fs-nombrearch <> oknombrearch)
  *hacer algo
end-if.
```

- Acceso aleatorio (por clave primaria o secundaria)

```
read nombrearch record
  [key is nombre-clave].
if (fs-nombrearch <> oknombrearch)
  *hacer algo
end-if.
```

- Acceso dinámico (por clave primaria o secundaria)

```
start nombrearch [key is operacion nombre-dato]
if (fs-nombrearch <> oknombrearch)
  *hacer algo

...

read nombrearch next record
  [key is nombre-clave].
if (fs-nombrearch <> oknombrearch)
  *hacer algo
end-if.
```

- Operaciones CRUD:

```
REWRITE   *para actualizar un registro
WRITE     *para insertar un registro
DELETE  *para borrar, LECTURA ANTERIOR ES OBLIGATORIA
```

7 Sort de archivos

```
sort arch-ordenar
  ascending key clave
  input procedure is entrada
  output procedure is salida
```

Para mandar un registro al archivo ordenado: `release`

Para obtener un registro del archivo ordenado: `return`

Con el "input procedure" seleccionamos los campos y armamos el registro que va a ir al archivo de sort.

Con el "output procedure", vamos leyendo registros con `return` y luego trabajamos sobre ellos (por ejemplo, imprimiendo un reporte).

El archivo de sort no es necesario abrilo ni cerrarlo.

8 Subprogramas

```
program-id. avg.

data division.
working-storage section.
01 a pic 99.
01 b pic 99.

linkage section.
01 res pic 99.

procedure division using a b res.
    compute res = ( a + b ) / 2.

end program avg.
```

9 SQL embebido en COBOL

9.1 Introducción

Let's say we've written a COBOL code which has some SQLs in it.

1. The COBOL compiler cannot understand the SQL statements. And if it were to encounter SQLs then it would flag compile time errors. So there needs to be someone who will be able to comment out these SQLs before the compiler runs. This someone is the **DB2 precompiler**.
2. The **DB2 precompiler** takes the source code (COBOL+DB2 commands), comments out the SQLs and inserts some CALL statements; the CALL statements are allowed in COBOL. Thus the code would be something like this:

   ```
   CALL 'DSNHLI' USING SQL-PLIST7
   ```

   DSNHLI is another program. While compiling the original source code, the compiler won't know what DSNHLI is. It leaves this task to the linker.
3. The DB2 precompiler filters out the SQLs from the source code and creates a **DBRM** (***Database Request Module***).

   Thus the output of the DB2 precompiler will be:

   a) a modified source code (with SQLs commented out and some extra CALL statements). This will be compiled as usual by the COBOL compiler and the object code will be created.

   b) a DBRM (containing the SQLs alone)
4. The DBRM doesn't have any information about how to access the data requested from the tables (like what is the best method to access the database, optimizing the query, etc). **Binding** achieves all of the above. We bind the DBRM into an **execution plan** and this plan contains the information about the DB2 resources required and access paths.

9.2 Definición de datos

When you access data with SQL statements in COBOL programs, you must provide corresponding data definitions in your COBOL source code. This requirement is usually accomplished by stating the data definitions in a copybook and including that copybook in your COBOL source code.

- **EXEC SQL ... END-EXEC Blocks.** Every SQL statement in your program must be included within an EXEC SQL ... END-EXEC. Each EXEC SQL ... END-EXEC block must contain only one SQL statement. If the SQL command is not terminated by an END-EXEC operand, a COBOL compiler error will be generated.
- **Host Variables.** Use host variables to store data that is accessed by both SQL and COBOL statements.

  There are two ways to use host variables:

  1. To store information returned by the database engine as the result of an SQL statement.
  2. To store information sent to the database engine and/or dynamic information that controls what is returned by a SELECT statement.

Algoritmo 1 All host variables (DEPT-NUM) appearing inside an EXEC SQL ... END-EXEC block should be preceded by a colon

```
EXEC SQL
     DECLARE EMPCURS CURSOR FOR
     SELECT LNAME, FNAME, PAYRATE, HOURS
         FROM EMPLOYEE
         WHERE DEPT = :DEPT-NUM
END-EXEC
```

- **Copybooks**. Use the EXEC SQL INCLUDE statement to include copybooks containing definitions for tables.

  Copybooks allow host variable definitions for a table's columns to be inserted within your COBOL code, allowing data from those columns to be used in your program. If a modification is made to a table (such as a change of data type or column length), then it is only necessary to modify the copybook rather than the source code definition for that table in all the programs that use it.

Algoritmo 2 Definición de un *copybook*

```
 01 EMP-TABLE.
     03 ENO    PIC S9(4) COMP.
     03 LNAME     PIC X(10).
     03 FNAME     PIC X(10).
     03 STREET   PIC X(20).
     03 CITY     PIC X(15).
     03 ST     PIC XX.
     03 ZIP       PIC X(5).
     03 DEPT     PIC X(4).
     03 PAYRATE     PIC S9(13)V99 COMP-3.
     03 COMP-3    PIC S9V99 COMP-3.
```

Algoritmo 3 Inclusión de un *copybook* en un programa COBOL

```
EXEC-SQL
    INCLUDE EMPREC
END-EXEC
```

- **SQL Communications Area**. Every COBOL program containing embedded SQL must have an SQL Communications Area (SQLCA) or the field SQLCODE defined in its Working-Storage Section. This definition is normally accomplished by including the SQLCA copybook provided with your COBOL system.
  - The SQLCA holds information on the status of the SQL statement last executed. It is updated after the execution of each EXEC SQL ... END-EXEC block of code.
  - Create error-checking routines that use the SQLCA to control the flow of the program.

9.3 *Cursor processing*

Cursor is a programming device that allows the SELECT statement to find a set of rows but return them one at a time.

Cursor processing is done in several steps:

1. Define the rows you want to retrieve, either in the working storage or in the procedure division. This is called declaring the cursor.

```
PROCEDURE DIVISION.
* declare cursor for select
     EXEC SQL
          DECLARE CursorJamesCameron CURSOR FOR
     Select Film_Title
       From Director_Film_Table
      Where Director_Last_Name equals "Cameron"
        and Director_First_Name equals "James"
      Order By Film_Title
     END-EXEC
```

2. Open the cursor. This activates the cursor and loads the data.

```
* open cursor
     EXEC SQL
          Open CursorJamesCameron
     END-EXEC
     MOVE SQLCODE TO DISP-CODE
```

3. Fetch the data into host variables. The fetch will take the current sequential row and put it into the host variable. It will then set the next sequential row to the current sequential row.

```
* fetch a data item
     EXEC SQL
          Fetch CursorJamesCameron
     Into :CameronMovieName
     END-EXEC
```

   You can repeat doing the fetch until you finish reading each row. When all rows are read the host variable will be set to null or spaces. Usually a special host variable is set by the dbms to indicate that the cursor is empty. In DB2 the SQLCODE is set to +100 when this happens.

4. Close the cursor.

```
* close the cursor
     EXEC SQL
          Close CursorJamesCameron
     END-EXEC
```

```
      identification division.
      program-id. DB2SQL.

      environment division.
      configuration section.

      data division.
      working-storage section.
      01 reg-cliente.
           03   reg-apeynom      pic      x(30) value spaces.
           03   reg-dni          pic      9(10) comp-3 value zeroes.
           03   reg-domicilio    pic      x(20) value spaces.
           03   reg-codpost      pic      9(4).
           03   reg-localidad    pic      x(10).
           03   reg-tel          pic      x(20).
           03   reg-fechanam     pic      9(8).
           03   reg-prof         pic      x(30).
           03   reg-ocup         pic      x(15).

      01 ws-localidad      pic x(10) value "CABA".
      01 ws-fechamin       pic x(10)  value "01011970".
      01 ws-fechamax       pic x(10)  value "31121979".
      01 ws-prof           pic x(30) value "estudiante".
```

```
01 ws-status         pic 999.
    88  no-enc       value 100.
    88  ok           value 000.

procedure division.
      exec sql
 declare micursor cursor for
     select apeynom
     from clientes
*    where localidad =:ws-localidad and
*         fechanaim >:ws-fechamin and
*         fechanaim >:ws-fechamax and
*         profesion =:ws-prof
     where profesion =:ws-prof
     end-exec.

    perform abrir-cursor.
    perform fetch-cursor.
    perform proceso until no-enc.
    perform cerrar-cursor.

abrir-cursor.
    exec sql
        open micursor
    end-exec.

fetch-cursor.
    exec sql
        fetch micursor
        into :reg-cliente.reg-apeynom
    end-exec.

proceso.
    display "Nombre:␣" reg-apeynom.
    display "DNI:␣" reg-dni.
    perform fetch-cursor.

cerrar-cursor.
    exec sql
        close micursor
    end-exec.

end program DB2SQL.
```

10 CICS (Customer Information Control System)

CICS es un monitor de comunicaciones (OLTP - *Online Transaction Processing*) de IBM, bajo el cual se pueden desarrollar transacciones de modo online.

Es una pieza clave en los servicios de muchos bancos, administraciones y grandes empresas.

CICS provee servicios que extienden o reemplazan las funciones del sistema operativo.

Sistema online produce resultados instantáneos, y permite que múltiples programas se ejecuten al mismo tiempo.

10.1 Componentes de una aplicación CICS

1. Programas (obligatorio)

```
EXEC CICS
  ...
END-EXEC
```

2. Transacciones (optativo)

   a) Ver 10.2

3. Mapas (optativo)

   Se utilizan para interactuar con el usuario. Contienen los distintos datos a presentar y recibir de un usuario. Son manejados por BMS (Basic Mapping Support) y son utilizados en la working storage de un programa COBOL.

```
SEND nombre-mapa *envia un mapa a la terminal
RECEIVE nombre-mapa *recibe los datos que un usuario ingresa
```

4. Archivos (optativo)
5. Bases de datos (optativo)
6. Terminales: para conectarse con el usuario. Ejemplos: teclado, impresora.

Cada programa CICS se inicia usando un identificador de transacción.

Una instalación CICS comprende una o más **regiones**, distribuídas a lo largo de una o más imágenes z/OS. Cada región CICS puede inicializarse como

- *batch job*: un proceso *batch* con sentencias JCL: es un *job* que corre indefinidamente.
- *started task*: una tarea

10.2 Transacciones CICS

Transacción CICS unidad de procesamiento iniciada por un único pedido que puede afectar uno o más objetos. Este procesamiento suele ser interactivo, pero también se permiten transacciones *background*.

Una transacción debe ser atómica. CICS puede asegurarse que una transacción es ejecutada completamente o no ejecutada.

> *Una transacción le da vida al programa. El programa es el conjunto de datos administrado por CICS. Una tarea es una instancia de una transacción.*

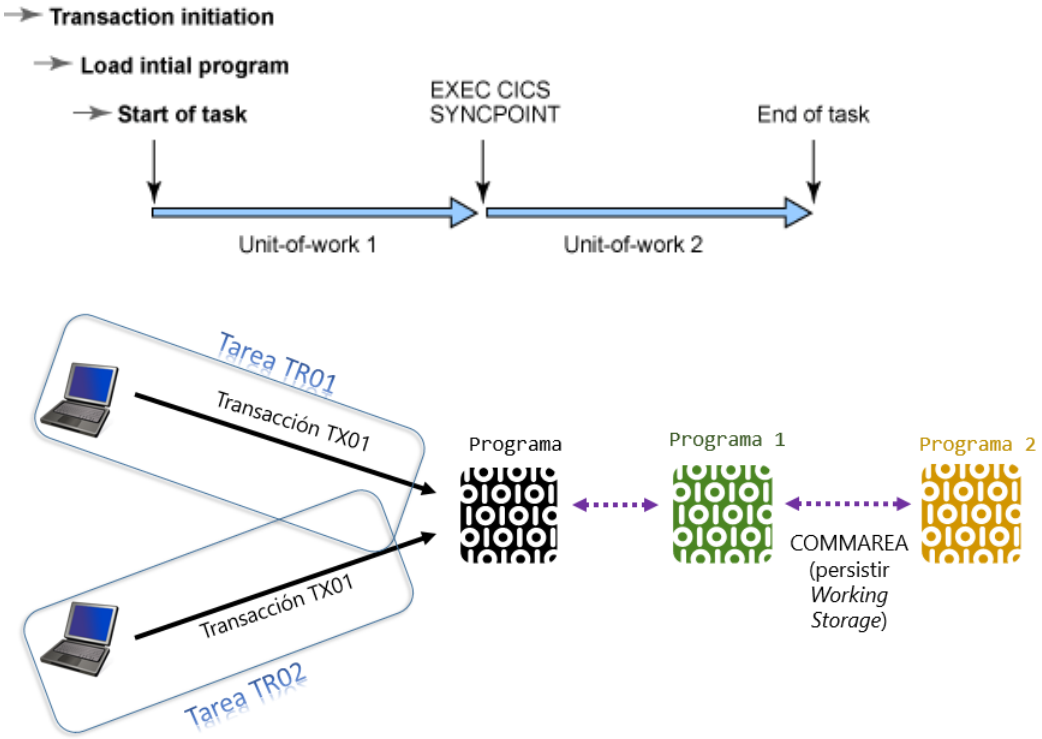

Figura 4: Transacciones, programas y tareas

Una transacción puede tener una relación 1-a-1 o 1-a-muchos con los programas a ejecutar.

Para administrar los componentes de una aplicación CICS se utilizan las **tablas de control** (cada tabla de control tiene un programa de control que corre en modo *background*).

1. FCT (*File Control Table*)

   Todos los archivos VSAM se registran aquí (modo de acceso, nombre del archivo, operaciones soportadas, *file status*, cantidad de buffers, etc.)

2. TCT (*Terminal Control Table*)

   Registra todas las terminales que pueden iniciar transacciones.

3. PCT (*Program Control Table*)

   Registra las asociaciones entre transacciones y los programas. Guarda el identificador de la transacción (1 a 4 caracteres), el nombre del programa, etc.

4. PPT (*Processing Program Table*)

   Registra todos los programas y los mapas.[1]

10.3 Transferring control in CICS environments

Transfers in the CICS environment are supported in the following ways:

- A transfer to program statement is implemented as a CICS XCTL command.
- A transfer to transaction statement is implemented as one of the following commands:
  - a CICS START command if the genReturnImmediate build descriptor option is set to NO
  - a CICS RETURN IMMEDIATE command if the genReturnImmediate build descriptor option is set to YES

  Setting genReturnImmediate to YES is supported only for CICS for z/OS® systems

  A show statement with an associated form and a returning clause is implemented as a CICS RETURN TRANSID command.
- Pase de datos utilizando la `COMMAREA`. Se utiliza para persistir la `working storage`.
- Instrucción LINK. Es como una CALL, pero la tarea sigue siendo la misma.

```
EXEC CICS
  LINK nombre-programa
  DISPLAY ... * esto si se ejecuta
END-EXEC
```

- Instrucción XCTL: Es como una CALL, pero el programa termina su ejecución.

```
EXEC CICS
  XCTL nombre-programa
  DISPLAY ... * esto NO se ejecuta
END-EXEC
```

- Llamadas asincrónicas:

```
EXEC CICS
  START TRANSID(xxxx) *comienza una nueva tarea
END-EXEC

EXEC CICS
  RETURN *pasar datos a una nueva transaccion
END-EXEC
```

[1] Puede haber programas que aparezcan en PPT y no en PCT, pero no al revés.

11 JCL (Job Control Language)

JCL es un lenguaje que se utiliza para describir los pasos de un *batch job*. Es el medio empleado para comunicarse con el Sistema Operativo y *Job Entry Subsystem* (JES2).

Job uno o varios grupos de sentencias de control que informan al sistema los programas a ejecutar, los archivos que éstos usarán, la cantidad de memoria necesaria, las característias de la tarea, el tipo de salida, etc.

Un job puede estar formado por uno o varios *job steps* (máximo 255).

Todo job es tratado por el JES2 como un archivo ordinario

Job step unidad de trabajo asociada a un programa.

11.1 Sentencias JCL

Sentencia	Sintaxis	Uso
`JOB`	`jobname JOB parametros`	Marcar el comienzo de un *job* y asignarle un nombre. Debe ser la primera en cada conjunto de sentencias.
`//`		Marcar el fin de un *job*
`EXEC`	`stepname EXEC parametros`	Marcar el comienzo de un *job step* y asignarle un nombre, identificar al programa a ejecutar
`DD`	`ddname DD parametros`	Identificar y describir un archivo a ser usado en un *job* o procedimiento. Debe colocarse a continuación de la sentencia EXEC.
`PROC`	`nombre PROC [parametros opcionales]`	Marcar el comienzo de un procedimiento
`PEND`	`PEND`	Marcar el fin de un procedimiento

Cuadro 1: Sentencias JCL