Common Business Oriented Language

Presented by Cristopher Bohol

EXPECTED LEARNING OUTCOMES

- In this lesson, you will be able to know:
- The history of Cobol
- The Features of Cobol
- Its Advantages and Disadvantages of using COBOL
- Why you should use Cobol
- Its Program Organization, Grammatical Hierarchy and Structure.
- It’s coding rules in Cobol Language.
- How to accept and display values on screen.
- How to declare variables and store values to the variables.
- How to use common verbs in computing values.
- How to use conditional statements.

HISTORY OF COBOL

- COBOL was first designed in 1959 by CODASYL.
- In late 1962, IBM announced that COBOL is going to be their primary development language.
- COBOL edition 1965 introduces the facilities for handling mass storage files and tables
- In 1968, COBOL was recognized and approved by ANSI standard language for standard commercial use.
- By 1970, COBOL had become the widely used programming language in the world.
- In 1982, ISO installed then-SC5's first Working Group: WG4 COBOL
- In 1985, the ISO working group 4 was accepted this version of the ANSI proposed standard.
- In 2002, the first Object-Oriented COBOL was released, which could be encapsulated as part of COBOL.
- In 2012, Computerworld surveys found out that over 60% of organizations still using COBOL.
- COBOL 2014 includes features like Method overloading, Dynamic capacity tables, etc.

Features of COBOL

- Here, are some most important features of the COBOL programming language:
- Allows you to handle a considerable volume of data due to its advanced file managing capability.
- Logical structure in COBOL is easier to read and modify.
- It can be executed and compiled on machines like IBM, personal computers, etc.
- Testing and debugging tools are always accessible on all platforms of the computer. Therefore, it is a robust programming language.
- You can easily debug in COBOL as it has different divisions.
- COBOL was designed for business-oriented applications. It can handle large volumes of data due to its advanced file handling capabilities.

Features of COBOL

- Here, are some most important features of the COBOL programming language:
- Allows you to handle a considerable volume of data due to its advanced file managing capability.
- Logical structure in COBOL is easier to read and modify.
- It can be executed and compiled on machines like IBM, personal computers, etc.
- Testing and debugging tools are always accessible on all platforms of the computer. Therefore, it is a robust programming language.
- You can easily debug in COBOL as it has different divisions.
- COBOL was designed for business-oriented applications. It can handle large volumes of data due to its advanced file handling capabilities.

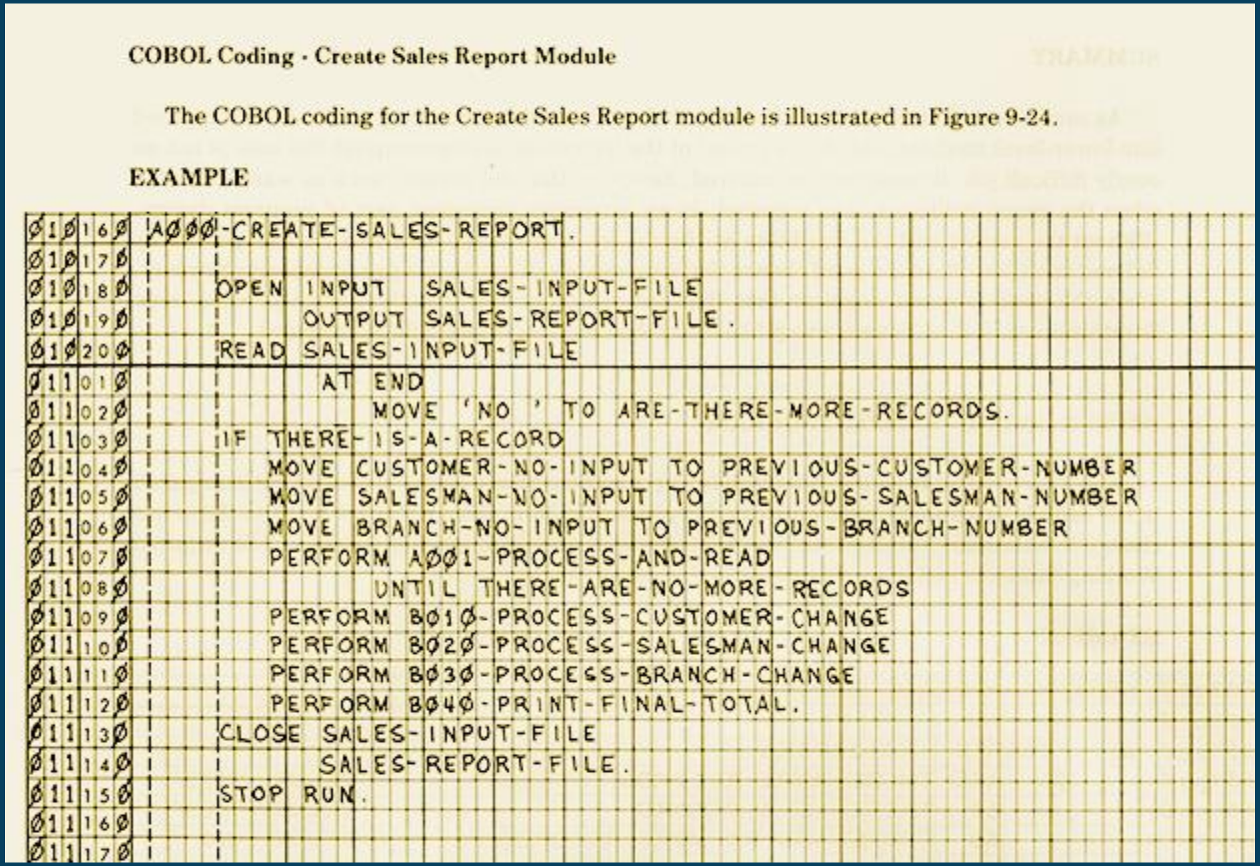
Advantages of COBOL

- Here, are important cons/benefits of using COBOL language:
- You can use COBOL as a self-documenting language.
- COBOL language can handle massive data processing.
- It is one of the primarily used high-level programming languages.
- Fully compatible with its past versions.
- COBOL language can handle massive data processing.
- Resolution of bugs is easier as it has an effective error message system.
- COBOL is also widely used as a self-documenting language.
- In COBOL, all the instructions can be coded in simple English words.

Features of COBOL

- Here, are some cons/disadvantages of using COBOL:
- It has very wordy syntax
- COBOL has the most rigid format
- It is not designed to handle scientific applications
- The time needed to compile a COBOL program is quite greater than machine-oriented programming languages.

COBOL HARDWARE

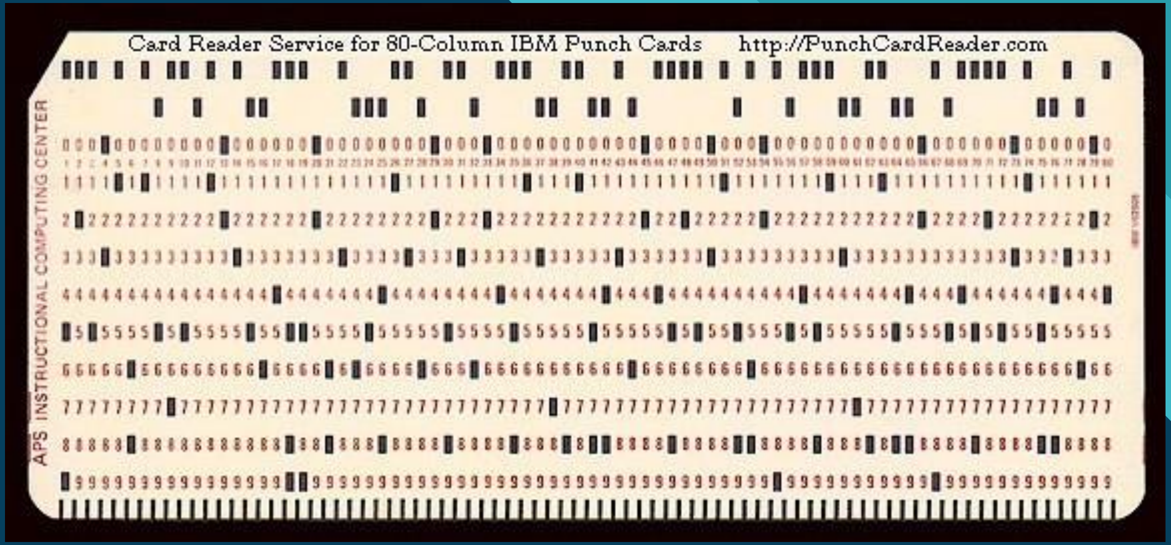


COBOL Coding - Create Sales Report Module

The COBOL coding for the Create Sales Report module is illustrated in Figure 9-24.

EXAMPLE

```
Ø1Ø16Ø A ØØØ-CREATE-SALES-REPORT.
Ø1Ø17Ø
Ø1Ø18Ø     OPEN INPUT  SALES-INPUT-FILE
Ø1Ø19Ø          OUTPUT SALES-REPORT-FILE.
Ø1Ø2ØØ     READ SALES-INPUT-FILE
Ø11Ø1Ø              AT END
Ø11Ø2Ø                  MOVE 'NO ' TO ARE-THERE-MORE-RECORDS.
Ø11Ø3Ø     IF THERE-IS-A-RECORD
Ø11Ø4Ø         MOVE CUSTOMER-NO-INPUT TO PREVIOUS-CUSTOMER-NUMBER
Ø11Ø5Ø         MOVE SALESMAN-NO-INPUT TO PREVIOUS-SALESMAN-NUMBER
Ø11Ø6Ø         MOVE BRANCH-NO-INPUT TO PREVIOUS-BRANCH-NUMBER
Ø11Ø7Ø         PERFORM AØØ1-PROCESS-AND-READ
Ø11Ø8Ø                 UNTIL THERE-ARE-NO-MORE-RECORDS
Ø11Ø9Ø         PERFORM BØ1Ø-PROCESS-CUSTOMER-CHANGE
Ø111ØØ         PERFORM BØ2Ø-PROCESS-SALESMAN-CHANGE
Ø1111Ø         PERFORM BØ3Ø-PROCESS-BRANCH-CHANGE
Ø1112Ø         PERFORM BØ4Ø-PRINT-FINAL-TOTAL.
Ø1113Ø     CLOSE SALES-INPUT-FILE
Ø1114Ø           SALES-REPORT-FILE.
Ø1115Ø     STOP RUN.
Ø1116Ø
Ø1117Ø
```


Card Reader Service for 80-Column IBM Punch Cards
http://PunchCardReader.com
APS INSTRUCTIONAL COMPUTING CENTER

Why Cobol?

- Billions of lines of existing code with more added each year
- Designed for business
- Great Compilers
- Runs fast
- Relatively simple to learn
- The Language keeps evolving

Program Organization

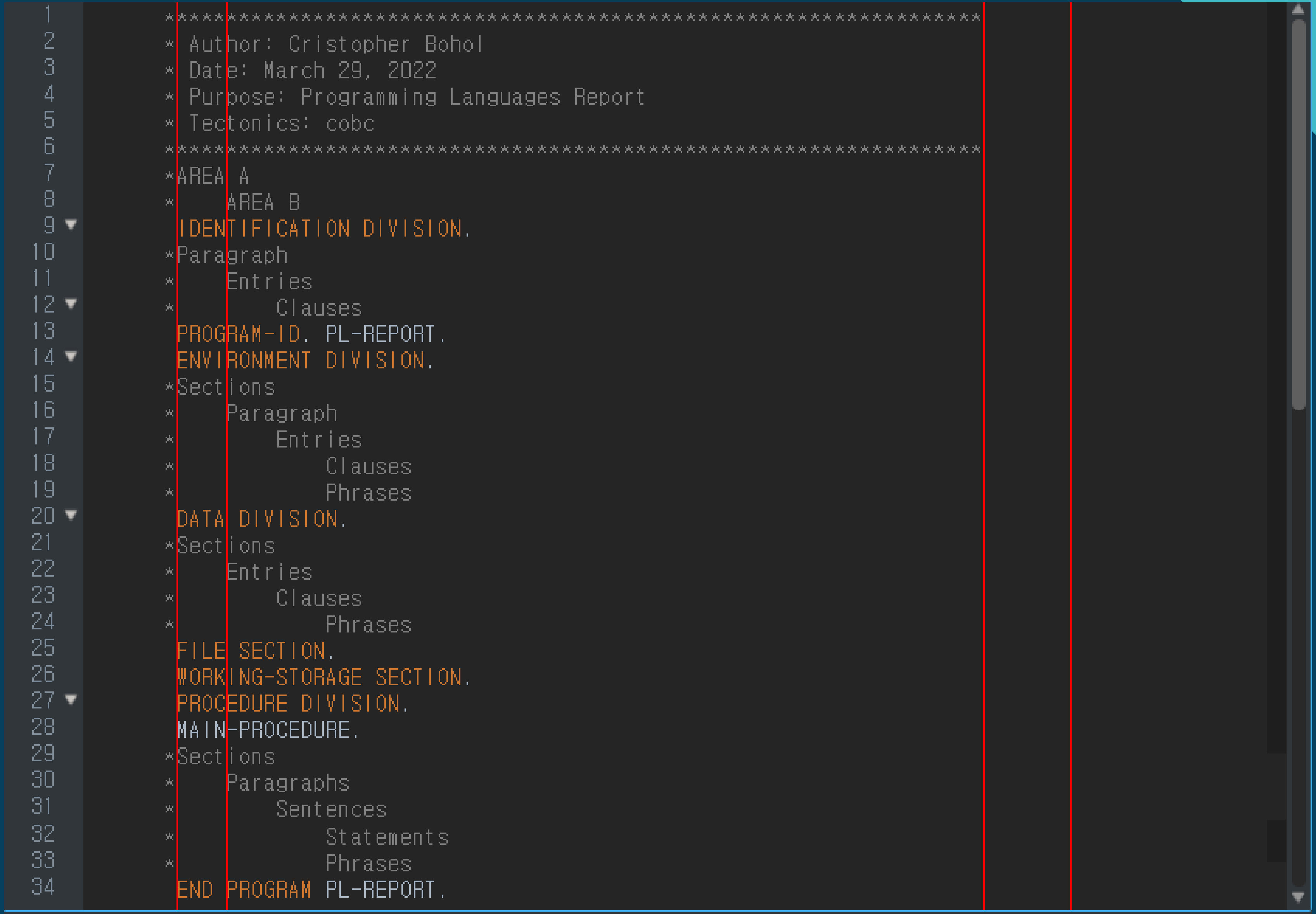
- Program – Organized like a book
- Division – Identification, Environment, Data, Procedure
- Section - Logical Subdivision of Program Logic
- Paragraph – Subdivision of section or division. It is either a user-defined or a predefined name followed by a period and consist of zero or more sentences/entries.
- Sentence – Combination of one or more statements. Sentences appear only in Procedure Division. A sentence must end with a period.
- Clause - Used to Specify how a data item is to be stored in the computer’s memory
- Phrase – Specifies the parameters that a program is called or the method is invoked.
- Word – a character-string that forms a user-defined word, a system-name or a reserved word.

Grammatical Hierarchy

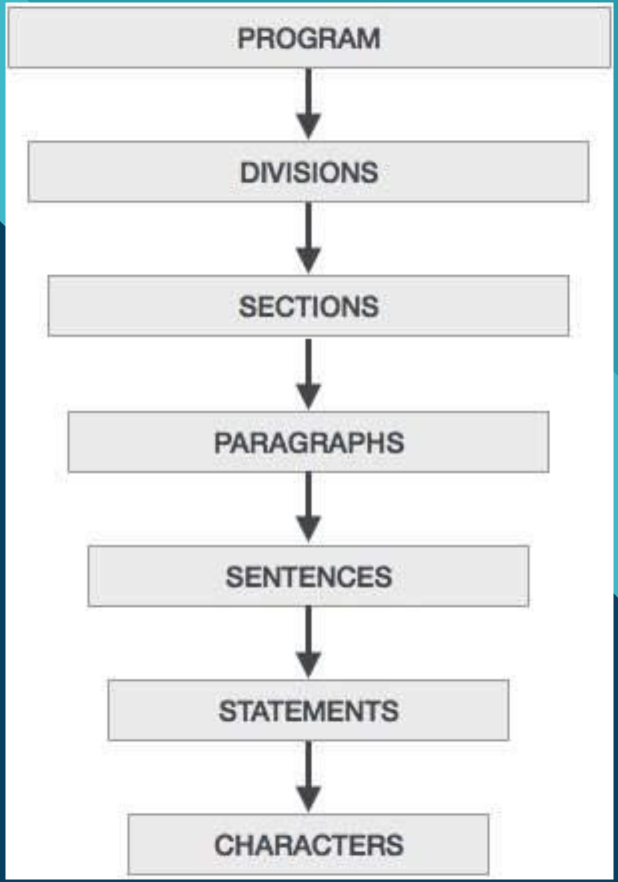
- The grammatical hierarchy follows this form:
- Identification division
  - Paragraphs
    - Entries
      - Clauses
- Environment division
  - Sections
    - Paragraphs
      - Entries
        - Clauses
        - Phrases
- Data division
  - Sections
    - Entries
      - Clauses
        - Phrases
- Procedure division
  - Sections
    - Paragraphs
      - Sentences
        - Statements
        - Phrases

0-format.cbl

Structure of a Program



```
      ******************************************************************
      * Author: Cristopher Bohol
      * Date: March 29, 2022
      * Purpose: Programming Languages Report
      * Tectonics: cobc
      ******************************************************************
      *AREA A
      *    AREA B
       IDENTIFICATION DIVISION.
      *Paragraph
      *    Entries
      *        Clauses
       PROGRAM-ID. PL-REPORT.
       ENVIRONMENT DIVISION.
      *Sections
      *    Paragraph
      *        Entries
      *            Clauses
      *            Phrases
       DATA DIVISION.
      *Sections
      *    Entries
      *        Clauses
      *            Phrases
       FILE SECTION.
       WORKING-STORAGE SECTION.
       PROCEDURE DIVISION.
       MAIN-PROCEDURE.
      *Sections
      *    Paragraphs
      *        Sentences
      *            Statements
      *            Phrases
       END PROGRAM PL-REPORT.
```



PROGRAM

DIVISIONS

SECTIONS

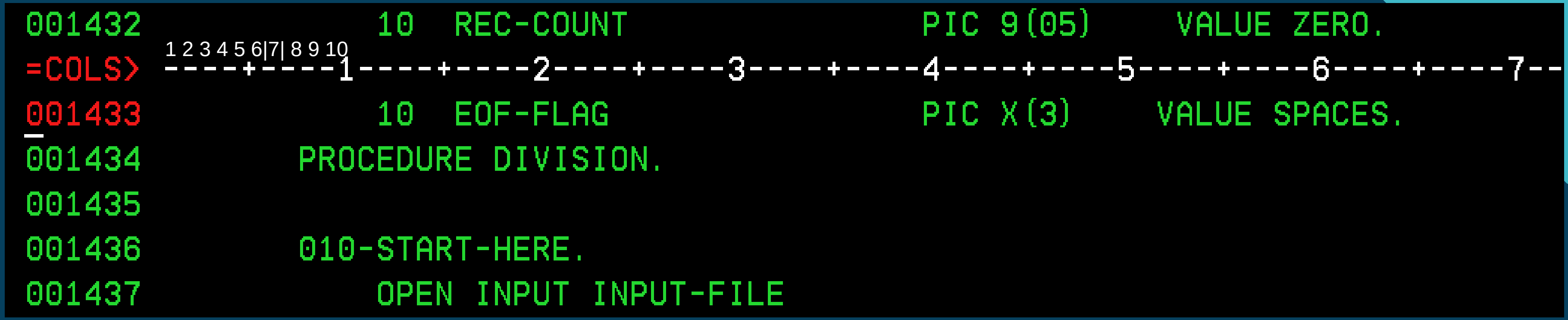
PARAGRAPHS

SENTENCES

STATEMENTS

CHARACTERS

Coding Rules



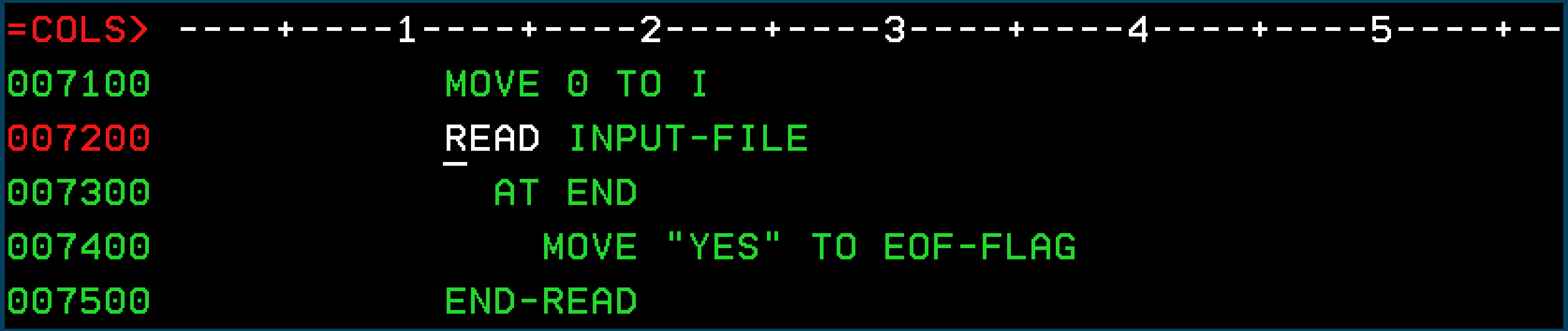
```
001432           10  REC-COUNT               PIC 9(05)    VALUE ZERO.
       1 2 3 4 5 6|7| 8 9 10
=COLS> ----+----1----+----2----+----3----+----4----+----5----+----6----+----7--
001433           10  EOF-FLAG                PIC X(3)    VALUE SPACES.
001434       PROCEDURE DIVISION.
001435
001436       010-START-HERE.
001437           OPEN INPUT INPUT-FILE
```

- Cols 1-6 – left blank. Compiler fills in with sequence numbers
- Col 7 – Usually blank,* means comment line, - is continuation, D for debugging lines
- Cols 8-11 – “A” margin or Area A
- Cols 12-72 – “B” margin or Area B
- Cols 73-80 – unused
- 1 2 3 4 5 6|7| 8 9 10 11|12 13 …71 71 |

  Seq Nos | | Area A | Area B |

Continuation of Statements

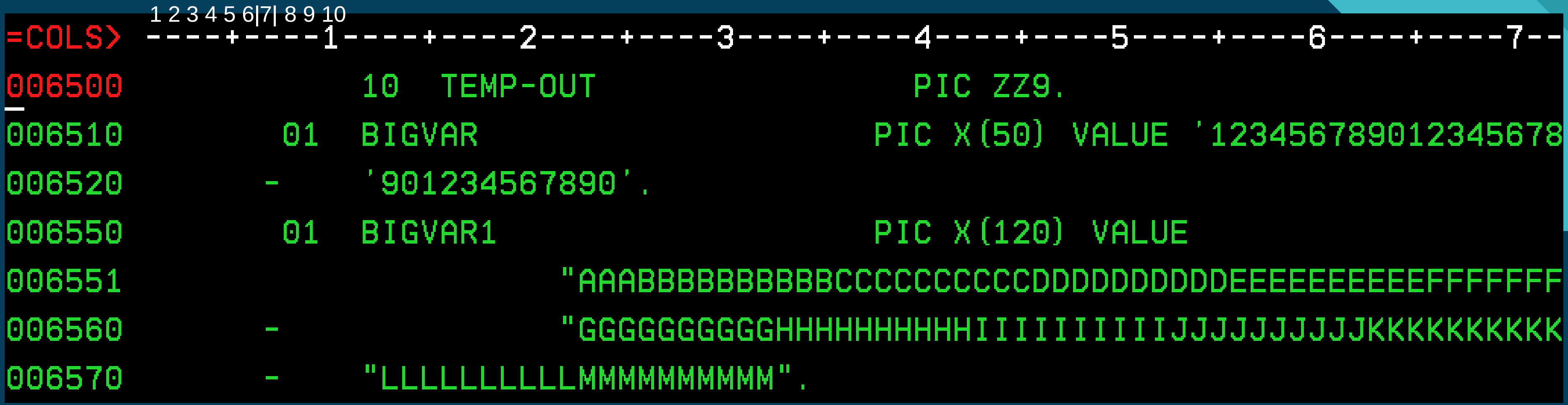
1 2 3 4 5 6|7| 89 10



```
=COLS> ----+----1----+----2----+----3----+----4----+----5----+--
007100            MOVE 0 TO I
007200            READ INPUT-FILE
007300              AT END
007400                MOVE "YES" TO EOF-FLAG
007500            END-READ
```

- Statements can be continued on the next line in Area B

Continuation of Literals



```
        1 2 3 4 5 6|7| 8 9 10
=COLS> ----+----1----+----2----+----3----+----4----+----5----+----6----+----7--
006500           10  TEMP-OUT                PIC ZZ9.
006510       01  BIGVAR                    PIC X(50) VALUE '12345678901234567 8
006520      -    '901234567890'.
006550       01  BIGVAR1                   PIC X(120) VALUE
006551                     "AAABBBBBBBBBBCCCCCCCCCCDDDDDDDDDDEEEEEEEEEEFFFFFFF
006560      -              "GGGGGGGGGGHHHHHHHHHHIIIIIIIIIIJJJJJJJJJJKKKKKKKKKK
006570      -    "LLLLLLLLLLMMMMMMMMMM".
```

- Continue the constant through column 71
- Put a “-” in column 7
- Continue constant with a ‘ OR “
- Continue constant in area B

Things That Go in Area A

Area A items:

- Division headers
- Section headers
- Paragraph headers or paragraph names
- Level indicators or level-numbers (01 and 77)
- `DECLARATIVES` and `END DECLARATIVES`
- End program, end class, and end method markers

Things That Go in Area B

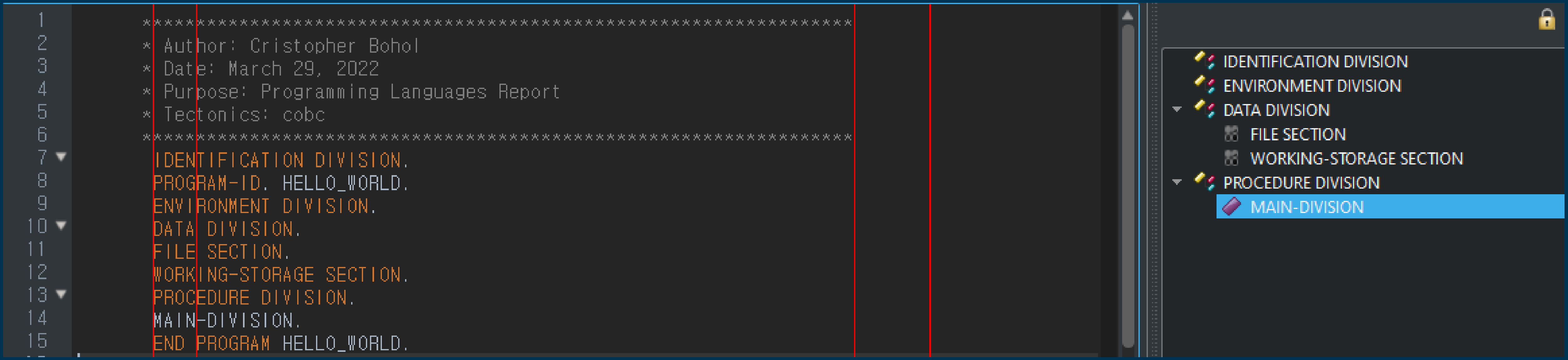
Area B items:

- Entries, sentences, statements, and clauses
- Continuation lines

Things That Go in Area A or B

- Area A or B
- Level-numbers
- Comment lines
- Compiler-directing statements
- Debugging lines
- Pseudo-text

PROGRAM STRUCTURE



```
      ******************************************************************
      * Author: Cristopher Bohol
      * Date: March 29, 2022
      * Purpose: Programming Languages Report
      * Tectonics: cobc
      ******************************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. HELLO_WORLD.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       FILE SECTION.
       WORKING-STORAGE SECTION.
       PROCEDURE DIVISION.
       MAIN-DIVISION.
       END PROGRAM HELLO_WORLD.
```

MAIN STRUCTURE OF COBOL

IDENTIFICATION DIVISION

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
AUTHOR. JOE SMITH.
INSTALLATION. TSYS.
DATE-WRITTEN. 12/03/2011.
DATE-COMPILED. 12/03/2011.
```

- Only **`PROGRAM-ID`** is required
- Some interesting parms can be coded on the **`PROGRAM-ID`**

PROCEDURE DIVISION

- The **`PROCEDURE DIVISION`** is where you code the executable statements in your COBOL program
- Divided into Paragraphs (terminated with periods):

```
100-MAIN.
      DISPLAY “HELLO…”
      PERFORM 200-SUB
      GOBACK
      .
 200-SUB.
      DISPLAY “…WORLD!”
      .
```

GOBACK

- Functions like an **`EXIT PROGRAM`** when coded at the end of a called program
- Functions like **`STOP RUN`** when coded in a main program
- I prefer coding this in place of **`STOP RUN`**

GO TO

- Causes an unconditional jump in program execution to the procedure that is named.
- This statement should be used only in very special situations, for instance, to branch to an error routine that terminates the program from a deeply nested area of your program.
- Overuse (any?) of this statement is unnecessary and leads to spaghetti code
- Don’t even think of using the alternate forms of `GO TO !`

PROCEDURE DIVISION

- To resolve ambiguity caused by not using periods, we will use statement delimiters:

```
END-IF
END-PERFORM
END-COMPUTE
. . .
```

PERFORM Paragraph

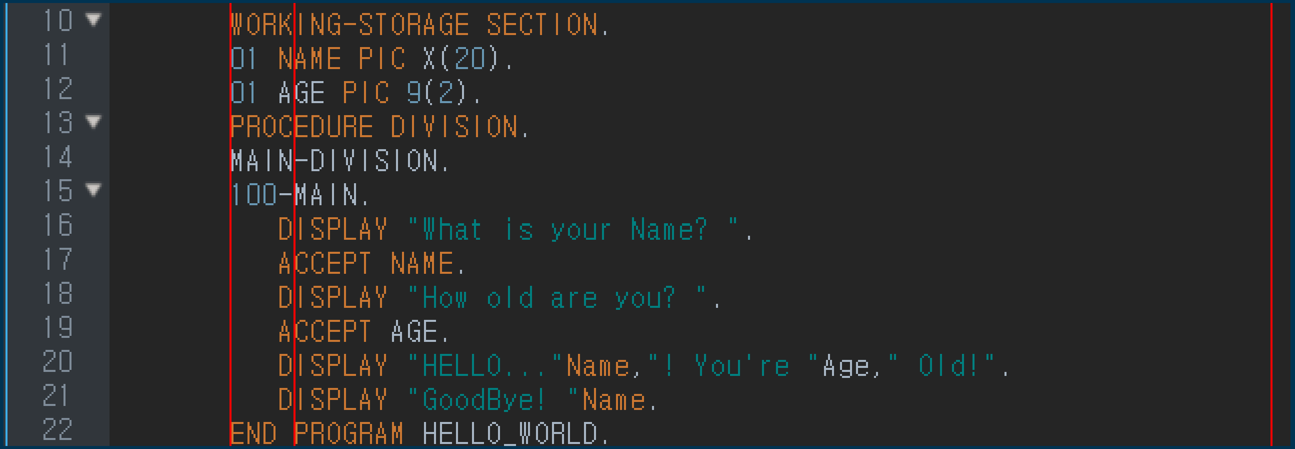
- PERFORM paragraph name
  - Execute all instructions in the paragraph
  - Return control to the next instruction after the **PERFORM**

```
PERFORM 100-ROUTINE
PERFORM 200-ROUTINE
PERFORM 100-ROUTINE
…
100-ROUTINE.
               …
200-ROUTINE.
               …
300-ROUTINE.
```

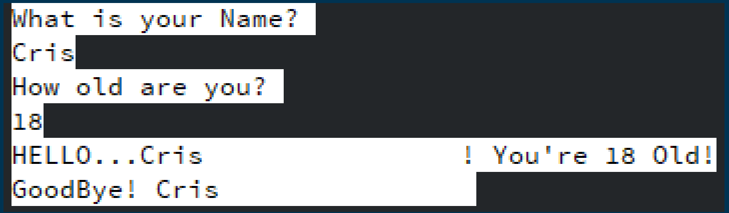
PERFORM PARAGRAPH

```
PERFORM 100-RTN
        WITH TEST AFTER
        VARYING X FROM 1 BY 1
        UNTIL  X = 100
…
100-RTN.
        ….
```

ACCEPT STATEMENT



```
WORKING-STORAGE SECTION.
01 NAME PIC X(20).
01 AGE PIC 9(2).
PROCEDURE DIVISION.
MAIN-DIVISION.
100-MAIN.
   DISPLAY "What is your Name? ".
   ACCEPT NAME.
   DISPLAY "How old are you? ".
   ACCEPT AGE.
   DISPLAY "HELLO..."Name,"! You're "Age," Old!".
   DISPLAY "GoodBye! "Name.
END PROGRAM HELLO_WORLD.
```



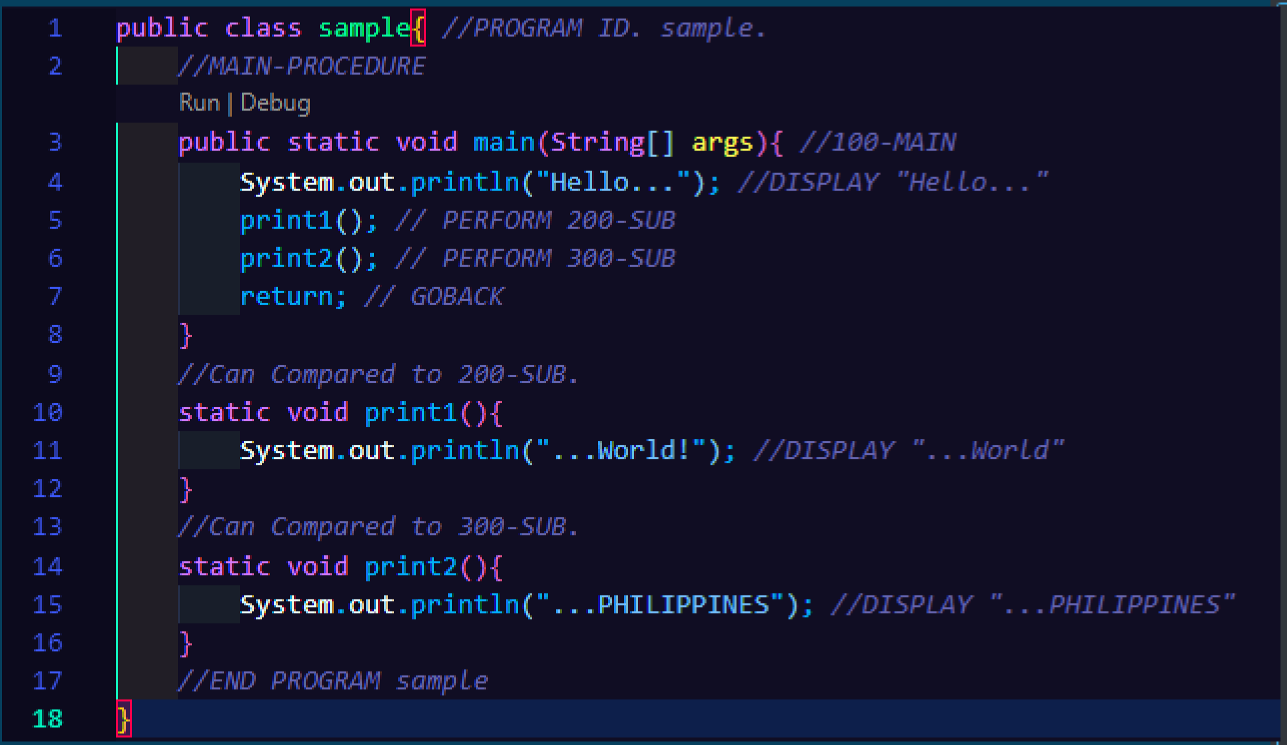
```
What is your Name?
Cris
How old are you?
18
HELLO...Cris                ! You're 18 Old!
GoodBye! Cris
```

1-Hello_World.cbl

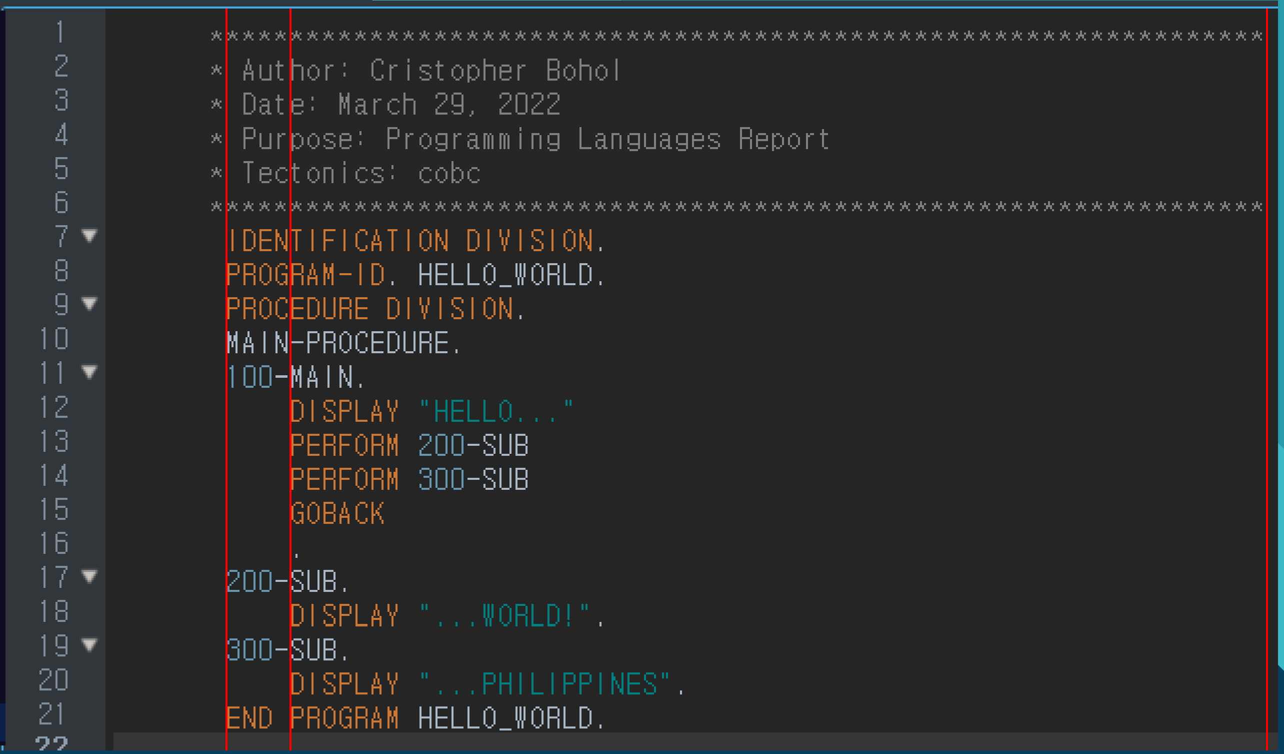
DIFFERENCES TO OTHER LANGUAGE

JAVA PROGRAM and Output

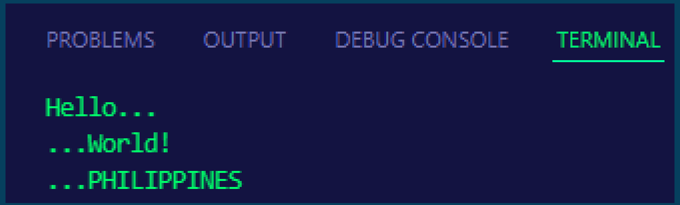
COBOL PROGRAM and Output



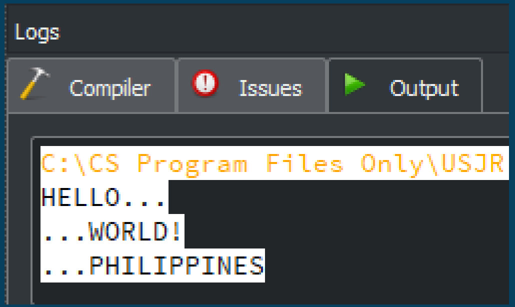
```
public class sample{ //PROGRAM ID. sample.
    //MAIN-PROCEDURE
    Run | Debug
    public static void main(String[] args){ //100-MAIN
        System.out.println("Hello..."); //DISPLAY "Hello..."
        print1(); // PERFORM 200-SUB
        print2(); // PERFORM 300-SUB
        return; // GOBACK
    }
    //Can Compared to 200-SUB.
    static void print1(){
        System.out.println("...World!"); //DISPLAY "...World"
    }
    //Can Compared to 300-SUB.
    static void print2(){
        System.out.println("...PHILIPPINES"); //DISPLAY "...PHILIPPINES"
    }
    //END PROGRAM sample
}
```



```
      ******************************************************************
      * Author: Cristopher Bohol
      * Date: March 29, 2022
      * Purpose: Programming Languages Report
      * Tectonics: cobc
      ******************************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. HELLO_WORLD.
       PROCEDURE DIVISION.
       MAIN-PROCEDURE.
       100-MAIN.
           DISPLAY "HELLO..."
           PERFORM 200-SUB
           PERFORM 300-SUB
           GOBACK
           .
       200-SUB.
           DISPLAY "...WORLD!".
       300-SUB.
           DISPLAY "...PHILIPPINES".
       END PROGRAM HELLO_WORLD.
```



```
Hello...
...World!
...PHILIPPINES
```



```
C:\CS Program Files Only\USJR
HELLO...
...WORLD!
...PHILIPPINES
```

DATA DIVISION

- Used to create variables and constant fields
- Only three data types
  - numeric `PIC 99999.`
  - alphanumeric (text/string) `PIC  XXX.`
  - alphabetic `PIC  AAA.`
- Level numbers indicate subordination of fields. Use levels 01-49
- Alphabetic is seldom used

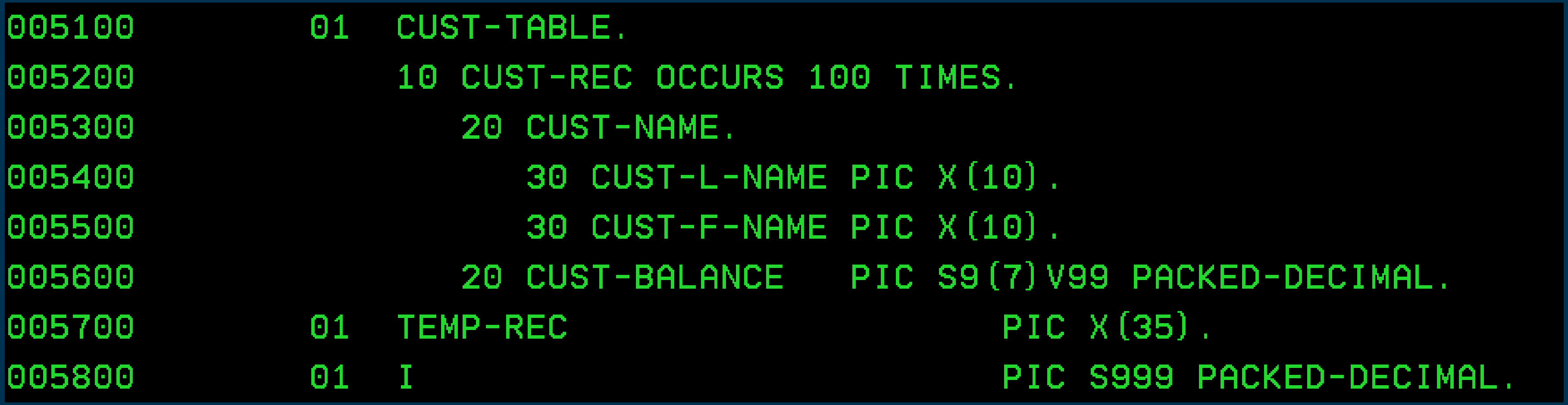
DATA DIVISION

We define data used in input-output operations.

```
FILE SECTION.
FD  CUSTOMER-FILE.
01 CUSTOMER-MASTER.
        05  CUST-NUM    PIC 9(2).
        05  CUST-FNAME  PIC X(20).
        05  CUST-LNAME  PIC X(20).
FD  SALES-REPORT.
01   REPORT-AREA    PIC X(132).
```

Level Numbers

- Group item – a subdivided field
- Elementary item – a non-subdivided field
- 01 – Group or independent item
- Higher numbers indicate subordinate fields



```
005100        01  CUST-TABLE.
005200            10 CUST-REC OCCURS 100 TIMES.
005300               20 CUST-NAME.
005400                  30 CUST-L-NAME PIC X(10).
005500                  30 CUST-F-NAME PIC X(10).
005600               20 CUST-BALANCE   PIC S9(7)V99 PACKED-DECIMAL.
005700        01  TEMP-REC                    PIC X(35).
005800        01  I                           PIC S999 PACKED-DECIMAL.
```

Level Numbers

- 66, 77, 88 have special significance
- 66 – Used to rename (no longer used)
- 77 – An independent item (choose 01)
- 88 – Condition name

Level Numbers

```
01  XXX.
    05  YYY.
        10    AAA  PIC X.
        10    BBB  PIC X.
    05  ZZZ   PIC X(20).

77   AAA     PIC 999V99.
```

Picture Clauses

- Picture clause values usually use 9, X, V, S, A
- 9 – a decimal digit
- X – any alphanumeric character
- V – an implied decimal point
- S – a sign
- A – A-Z, and blank

Picture Clauses

- PIC 9(6) // 000000
- PIC 9(6) value 4 // 000004
- PIC 9(6)V99 // 000000.00
- PIC 999999V99 // 000000.00
- PICTURE X(10)   // XXXXXXXXXX
- PIC XXXXXXXXXX // XXXXXXXXXX
- PIC S9(4)V9(4) // +0000.0000
- PIC S9999V9999 // +0000.0000
- PIC 9(18) //000000000000000000
- PIC X(4) value “4abc” // 4abc

Numeric Edited Fields

- `XXXBXXBXXXX //just a whitespace`
- `99/99/99 // 00/00/00`
- `ZZ,ZZZ.99DB // (5spaces).99(spaces)`
- `***,***.99 //******.00`
- `----.99 // (4spaces).00`
- `$$$9.99 //(2spaces)$0.00`
- `99999.99 //00000.00`

USAGE Clause

- Specifies the format in which data is stored in memory
- Normally, the phrase "**USAGE IS**" is omitted

```
01 FIRST-NAME  USAGE IS DISPLAY PIC X(20).
01 FIRST-NAME  PIC X(20).
```

DATA DIVISION

Define the data needed for internal processing in the **`WORKING-STORAGE SECTION`**.

Storage is statically allocated and exists for the life of the *run unit*.

```
WORKING-STORAGE SECTION.
01  TOTAL-FIELDS.
         05  CUST-TOTAL    PIC S9(7)V99 VALUE 0. // +0000000.00
         05  COST-TOTAL    PIC  S9(7)V99 VALUE 0. //+0000000.00
01  DATE-AND-TIME.
         05  CD-YEAR       PIC 9999. // 0000
         05  CD-MONTH      PIC 99. // 00
```

DATA DIVISION

- Describe data that exists in another program, or storage you want to associate with a symbolic name in the **`LINKAGE SECTION`**.

```
LINKAGE SECTION.
01  LK-DATA-AREA
       05   NAME    PIC X(40).
       05   AGE     PIC 999.
```

DATA DIVISION

The **LOCAL-STORAGE SECTION** is used to have storage allocated each time a program is entered, and deallocated on return from the program. Used for compatibility with C or Java.

```
LOCAL-STORAGE SECTION.
01  CUST-NO      PIC X(3).
01  COST         PIC 9(5)V99.
```

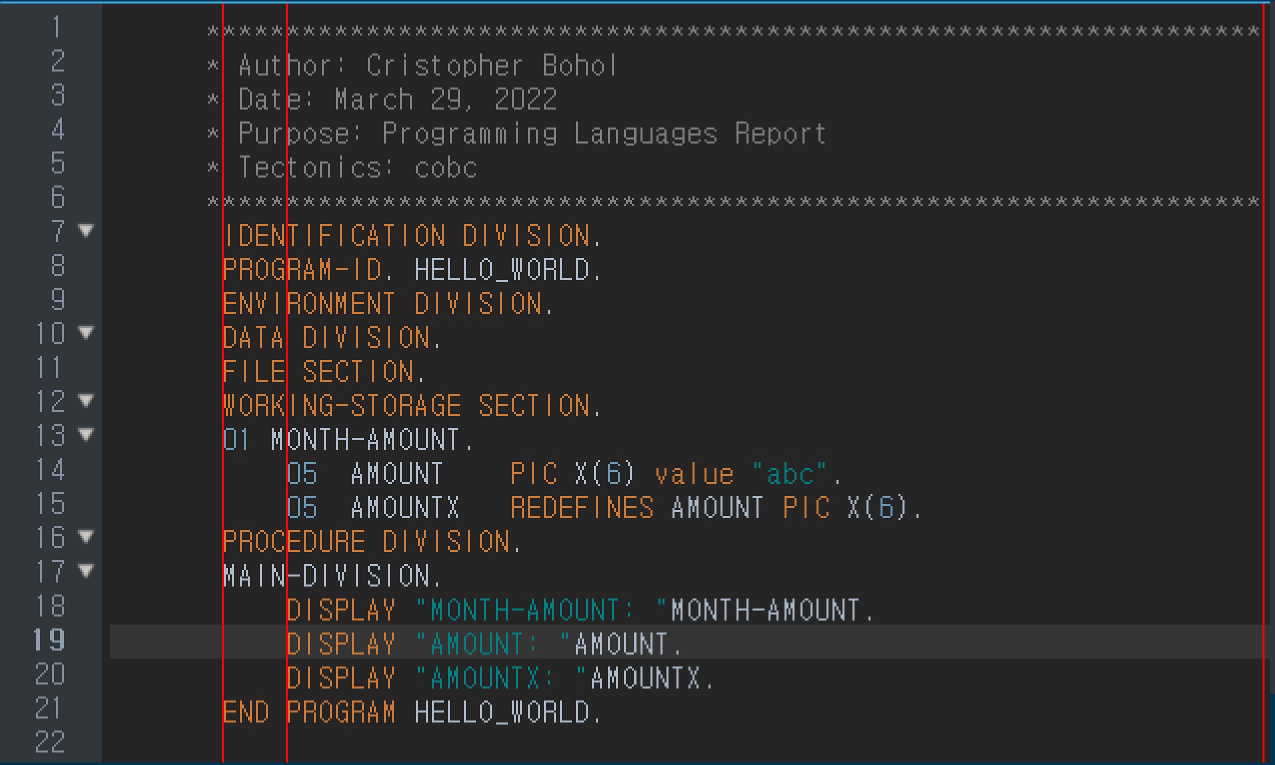
Initialization of Storage

- `WORKING-STORAGE` for programs is allocated at the start of the run unit.
- Any data items with `VALUE` clauses are initialized to the appropriate value at that time.

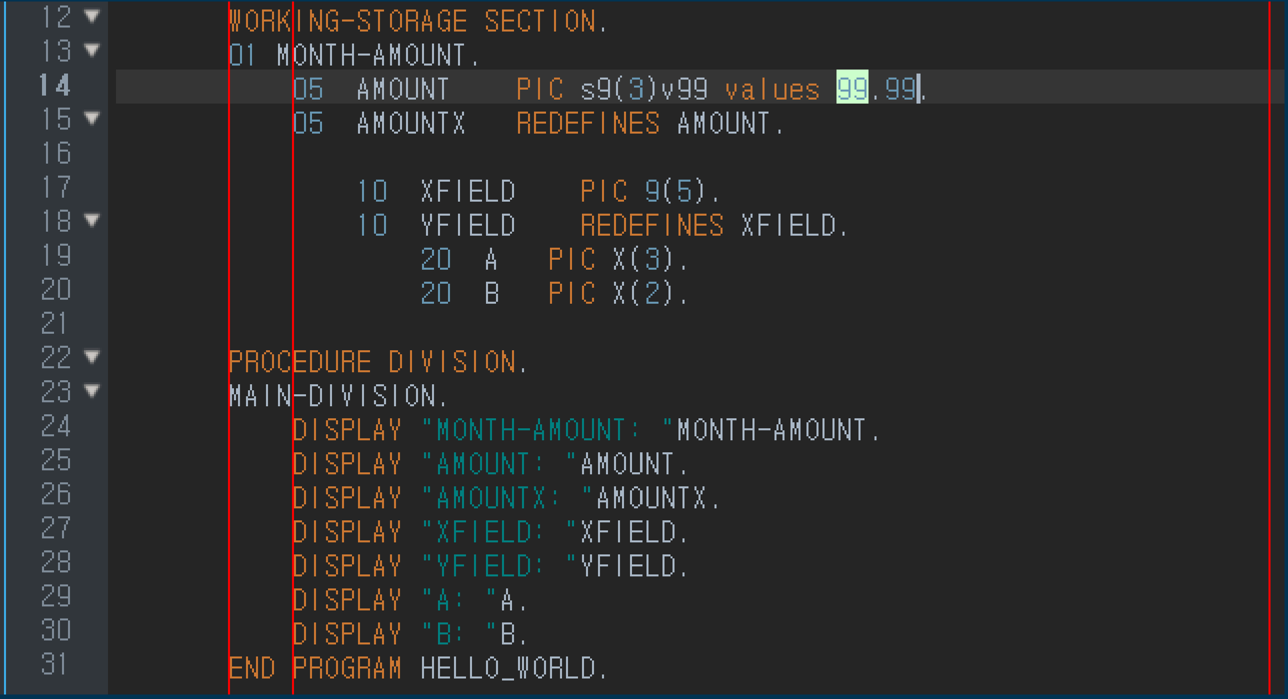
Group and Data Items

```
01 Customer-Record.
     05 Customer-Name.
         10 Last-Name Pic x(17).
         10 Filler Pic x.
         10 Initials Pic xx.
     05 Part-Order.
         10 Part-Name Pic x(15).
         10 Part-Color Pic x(10).
```

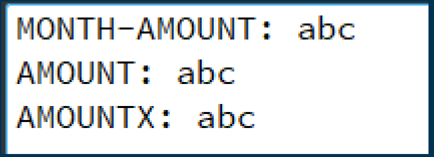
Redefines



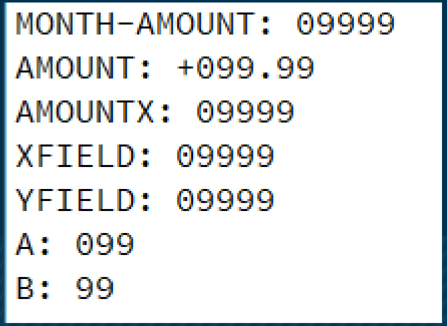
```
      ******************************************************************
      * Author: Cristopher Bohol
      * Date: March 29, 2022
      * Purpose: Programming Languages Report
      * Tectonics: cobc
      ******************************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. HELLO_WORLD.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       FILE SECTION.
       WORKING-STORAGE SECTION.
       01 MONTH-AMOUNT.
           05  AMOUNT     PIC X(6) value "abc".
           05  AMOUNTX    REDEFINES AMOUNT PIC X(6).
       PROCEDURE DIVISION.
       MAIN-DIVISION.
           DISPLAY "MONTH-AMOUNT: "MONTH-AMOUNT.
           DISPLAY "AMOUNT: "AMOUNT.
           DISPLAY "AMOUNTX: "AMOUNTX.
       END PROGRAM HELLO_WORLD.
```



```
       WORKING-STORAGE SECTION.
       01 MONTH-AMOUNT.
           05  AMOUNT     PIC s9(3)v99 values 99.99.
           05  AMOUNTX    REDEFINES AMOUNT.

               10  XFIELD    PIC 9(5).
               10  YFIELD    REDEFINES XFIELD.
                   20  A   PIC X(3).
                   20  B   PIC X(2).

       PROCEDURE DIVISION.
       MAIN-DIVISION.
           DISPLAY "MONTH-AMOUNT: "MONTH-AMOUNT.
           DISPLAY "AMOUNT: "AMOUNT.
           DISPLAY "AMOUNTX: "AMOUNTX.
           DISPLAY "XFIELD: "XFIELD.
           DISPLAY "YFIELD: "YFIELD.
           DISPLAY "A: "A.
           DISPLAY "B: "B.
       END PROGRAM HELLO_WORLD.
```

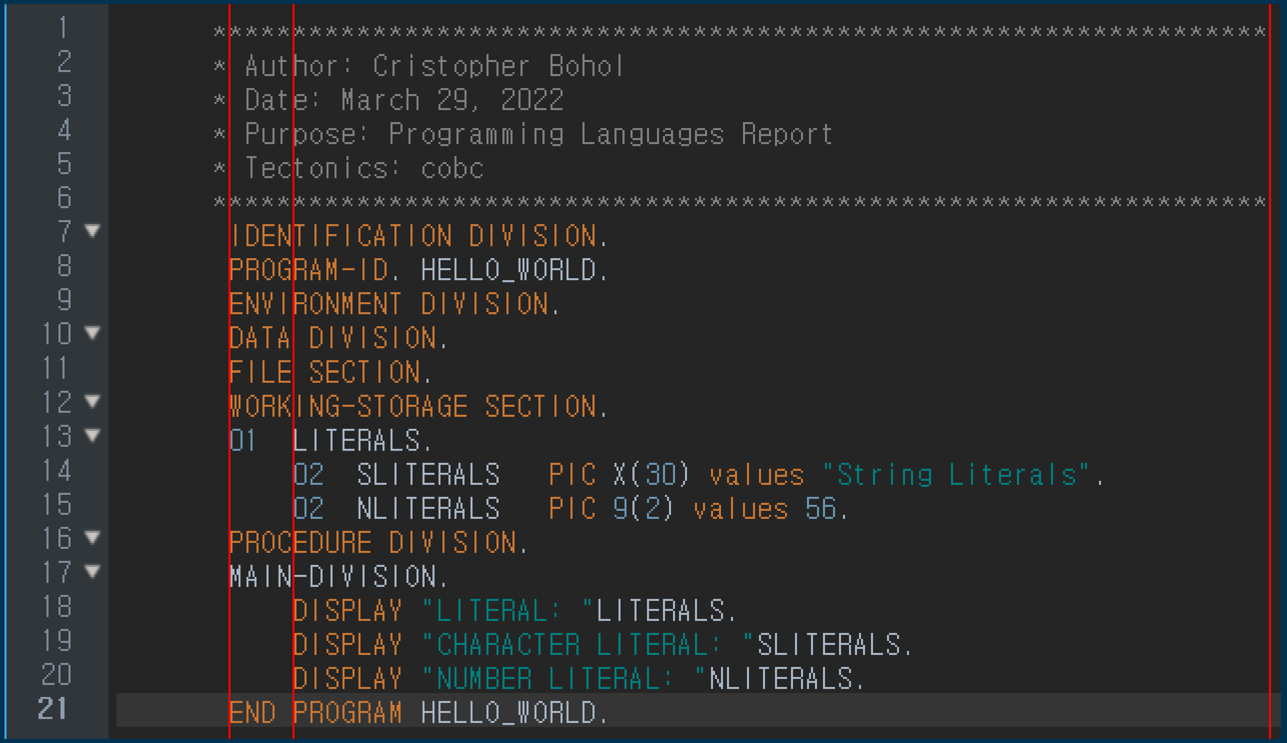


```
MONTH-AMOUNT: abc
AMOUNT: abc
AMOUNTX: abc
```

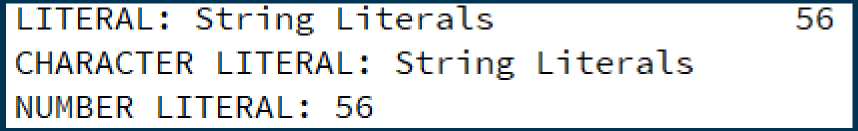


```
MONTH-AMOUNT: 09999
AMOUNT: +099.99
AMOUNTX: 09999
XFIELD: 09999
YFIELD: 09999
A: 099
B: 99
```

LITERALS



```
      ******************************************************************
      * Author: Cristopher Bohol
      * Date: March 29, 2022
      * Purpose: Programming Languages Report
      * Tectonics: cobc
      ******************************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. HELLO_WORLD.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       FILE SECTION.
       WORKING-STORAGE SECTION.
       01  LITERALS.
           02  SLITERALS   PIC X(30) values "String Literals".
           02  NLITERALS   PIC 9(2) values 56.
       PROCEDURE DIVISION.
       MAIN-DIVISION.
           DISPLAY "LITERAL: "LITERALS.
           DISPLAY "CHARACTER LITERAL: "SLITERALS.
           DISPLAY "NUMBER LITERAL: "NLITERALS.
       END PROGRAM HELLO_WORLD.
```



```
LITERAL: String Literals               56
CHARACTER LITERAL: String Literals
NUMBER LITERAL: 56
```

Constants

- A *constant* is a data item that has only one value and it can never change
- Unfortunately, COBOL does **not** define a construct specifically for constants
- Moral: All values are subject to change

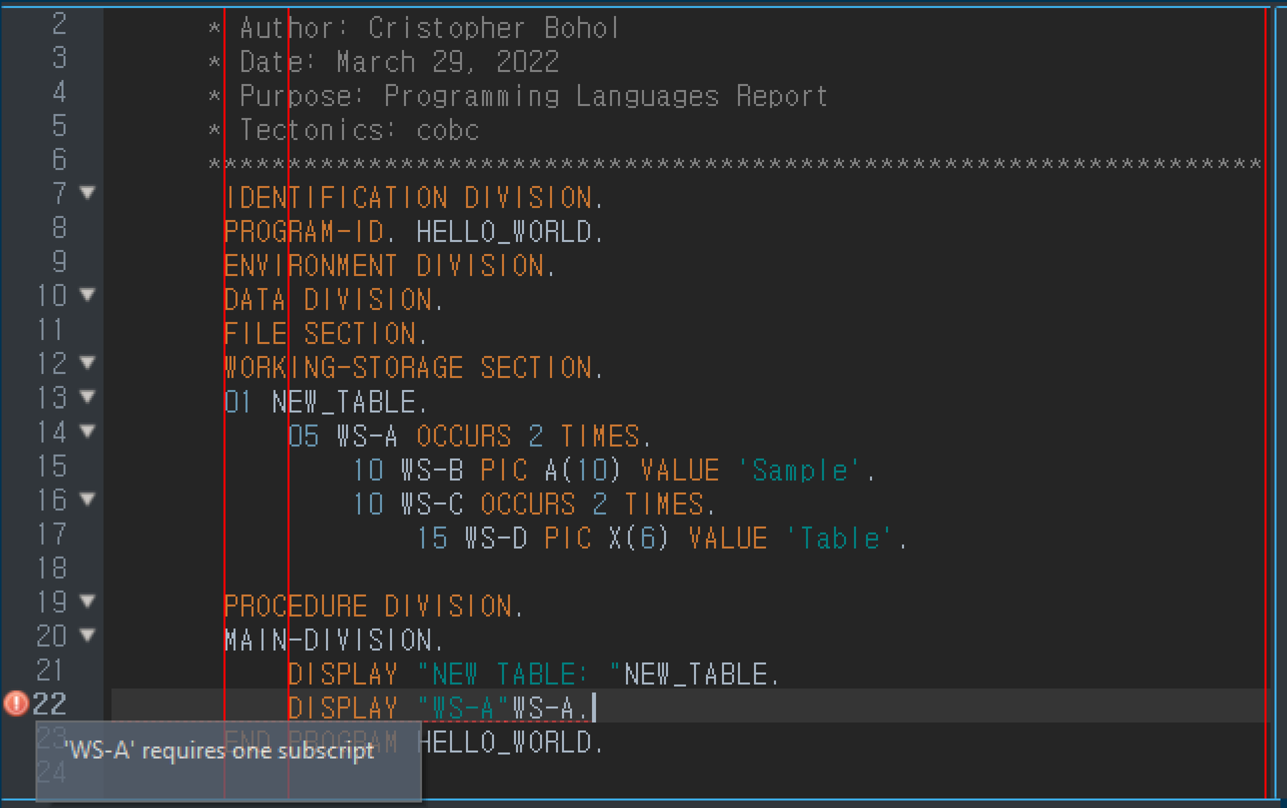
```
Data Division.
   01 Report-Header pic x(50)
                    value "Company Report".
   01 Interest      pic 9v9999
                    value 1.0265.
```

Figurative Constants

There are some figurative constants supplied by the language:

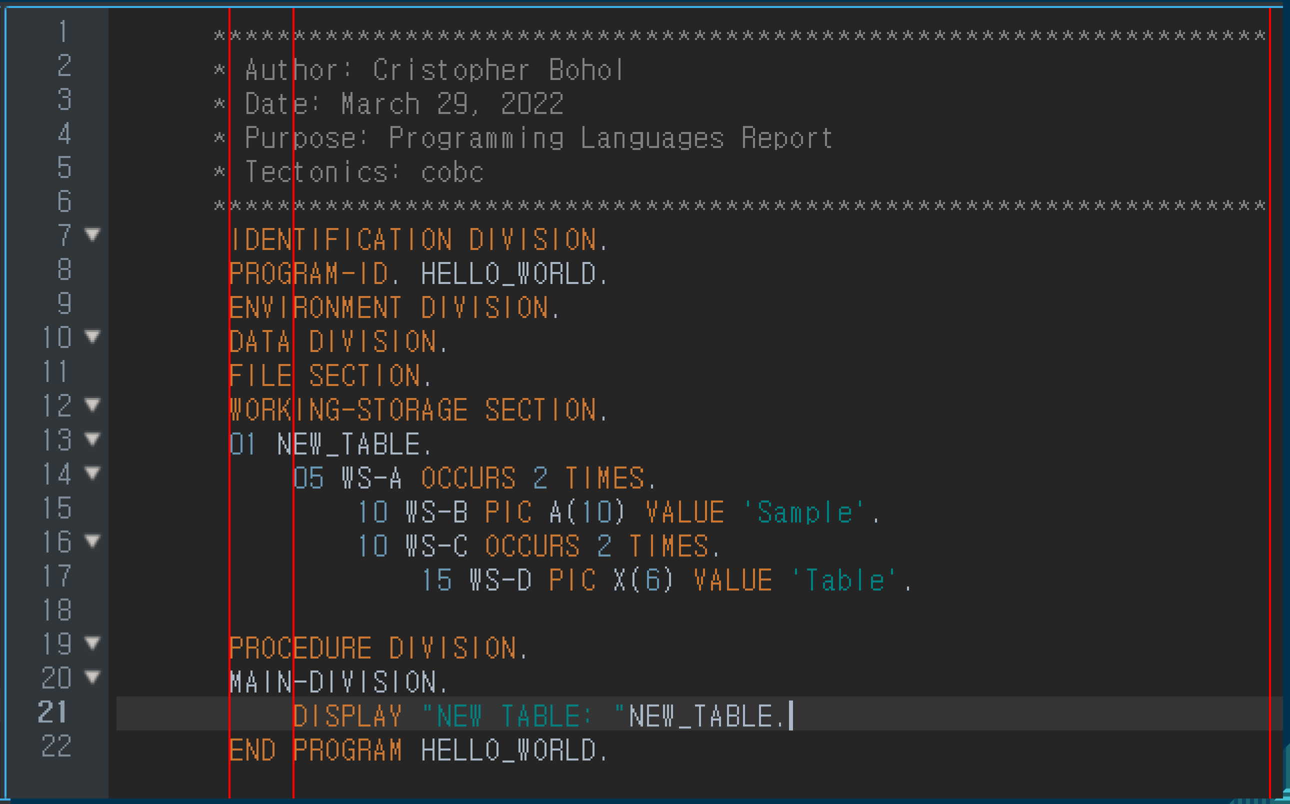
- ZERO - an appropriate form of 0
- SPACE - x’40’
- HIGH-VALUES - binary 1’s
- LOW-VALUES - binary 0’s
- QUOTE - a single quote
- NULL - binary 0’s used for pointers

TABLES (ARRAYS)

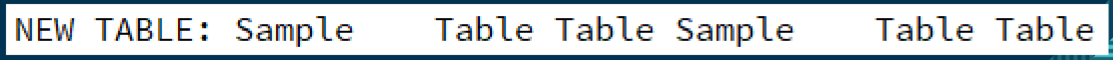


```
      * Author: Cristopher Bohol
      * Date: March 29, 2022
      * Purpose: Programming Languages Report
      * Tectonics: cobc
      ******************************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. HELLO_WORLD.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       FILE SECTION.
       WORKING-STORAGE SECTION.
       01 NEW_TABLE.
           05 WS-A OCCURS 2 TIMES.
               10 WS-B PIC A(10) VALUE 'Sample'.
               10 WS-C OCCURS 2 TIMES.
                   15 WS-D PIC X(6) VALUE 'Table'.

       PROCEDURE DIVISION.
       MAIN-DIVISION.
           DISPLAY "NEW TABLE: "NEW_TABLE.
           DISPLAY "WS-A"WS-A.
       END PROGRAM HELLO_WORLD.
```

'WS-A' requires one subscript

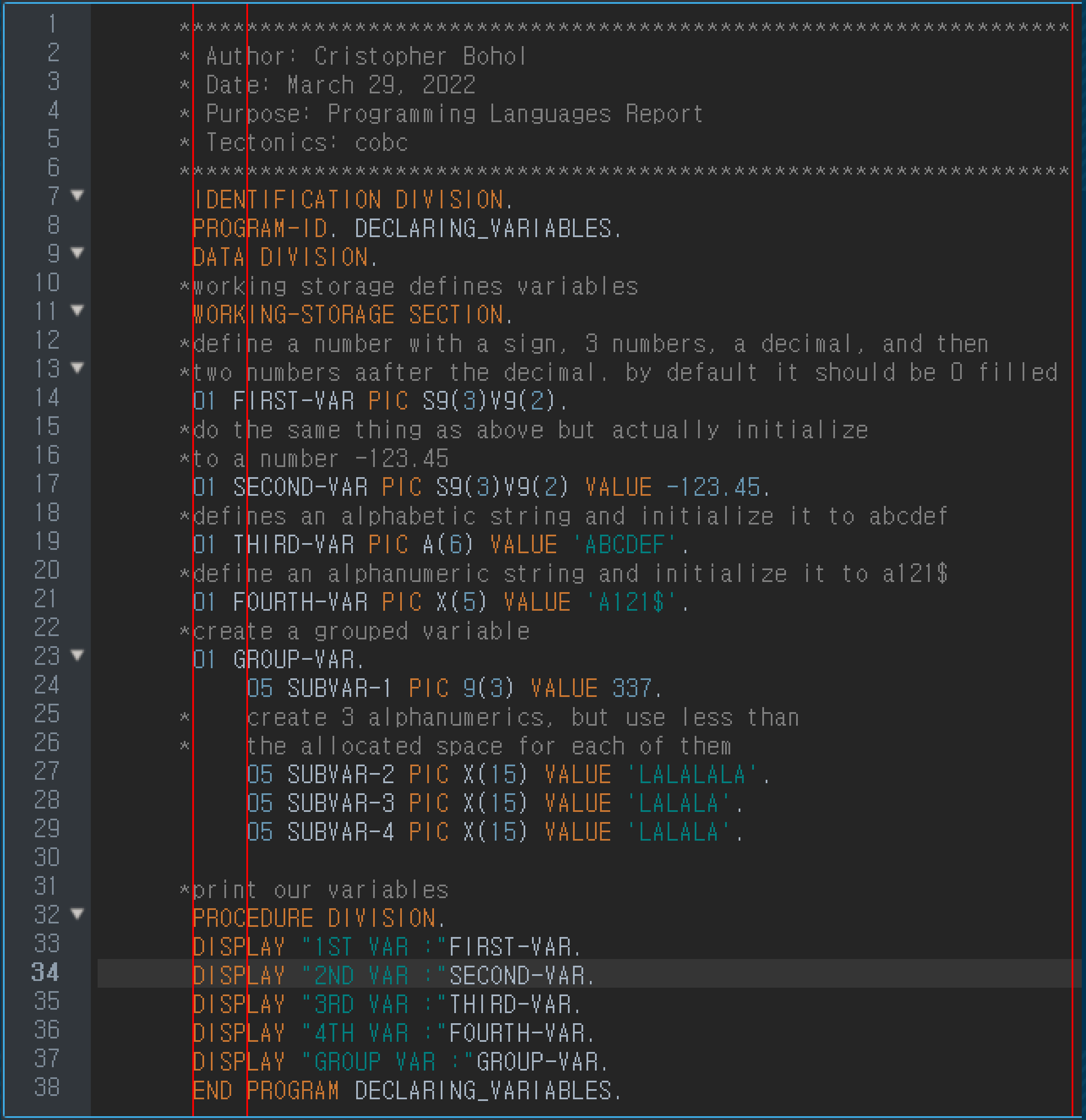


```
      ******************************************************************
      * Author: Cristopher Bohol
      * Date: March 29, 2022
      * Purpose: Programming Languages Report
      * Tectonics: cobc
      ******************************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. HELLO_WORLD.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       FILE SECTION.
       WORKING-STORAGE SECTION.
       01 NEW_TABLE.
           05 WS-A OCCURS 2 TIMES.
               10 WS-B PIC A(10) VALUE 'Sample'.
               10 WS-C OCCURS 2 TIMES.
                   15 WS-D PIC X(6) VALUE 'Table'.

       PROCEDURE DIVISION.
       MAIN-DIVISION.
           DISPLAY "NEW TABLE: "NEW_TABLE.
       END PROGRAM HELLO_WORLD.
```

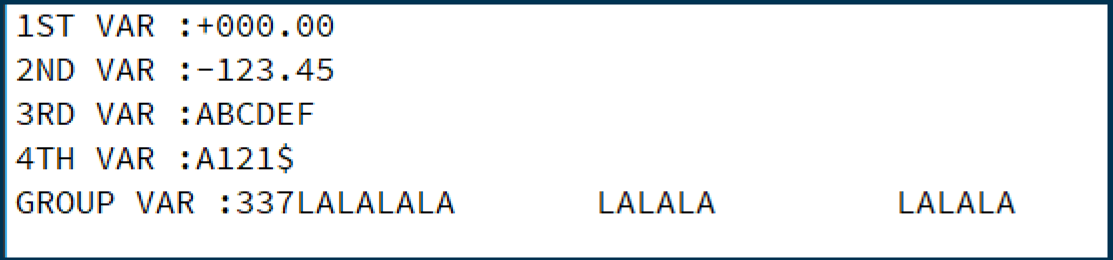


```
NEW TABLE: Sample    Table Table Sample    Table Table
```

2-Variables.cbl



```
      ******************************************************************
      * Author: Cristopher Bohol
      * Date: March 29, 2022
      * Purpose: Programming Languages Report
      * Tectonics: cobc
      ******************************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. DECLARING_VARIABLES.
       DATA DIVISION.
      *working storage defines variables
       WORKING-STORAGE SECTION.
      *define a number with a sign, 3 numbers, a decimal, and then
      *two numbers aafter the decimal. by default it should be 0 filled
       01 FIRST-VAR PIC S9(3)V9(2).
      *do the same thing as above but actually initialize
      *to a number -123.45
       01 SECOND-VAR PIC S9(3)V9(2) VALUE -123.45.
      *defines an alphabetic string and initialize it to abcdef
       01 THIRD-VAR PIC A(6) VALUE 'ABCDEF'.
      *define an alphanumeric string and initialize it to a121$
       01 FOURTH-VAR PIC X(5) VALUE 'A121$'.
      *create a grouped variable
       01 GROUP-VAR.
           05 SUBVAR-1 PIC 9(3) VALUE 337.
      *    create 3 alphanumerics, but use less than
      *    the allocated space for each of them
           05 SUBVAR-2 PIC X(15) VALUE 'LALALALA'.
           05 SUBVAR-3 PIC X(15) VALUE 'LALALA'.
           05 SUBVAR-4 PIC X(15) VALUE 'LALALA'.

      *print our variables
       PROCEDURE DIVISION.
       DISPLAY "1ST VAR :"FIRST-VAR.
       DISPLAY "2ND VAR :"SECOND-VAR.
       DISPLAY "3RD VAR :"THIRD-VAR.
       DISPLAY "4TH VAR :"FOURTH-VAR.
       DISPLAY "GROUP VAR :"GROUP-VAR.
       END PROGRAM DECLARING_VARIABLES.
```



```
1ST VAR :+000.00
2ND VAR :-123.45
3RD VAR :ABCDEF
4TH VAR :A121$
GROUP VAR :337LALALALA       LALALA         LALALA
```

MOVE STATEMENT

- Used to copy data from one field to another
- Example -

```
    MOVE X-FIELD TO Y-FIELD Z-FIELD
```

- Data is copied from the sending field to the receiving field

MOVE STATEMENT

- To move data from one field to another field, the two fields should be “compatible” but don’t have to be identically pictured

- Alphanumeric - `PIC X(10)`
- Numeric - `PIC 999v99`
- Numeric-Edited - `PIC 999.99-`

Compatible moves:

-Alphanumeric to Alphanumeric

-Numeric to Numeric

-Numeric to Numeric edited

MOVE STATEMENT

- **Compatible moves:**

  -Alphanumeric to Numeric if the sending field is an unsigned integer

  -Alphanumeric to Numeric edited if the sending field is an unsigned integer

  -Numeric to Alphanumeric if the sending field is an unsigned integer

MOVE STATEMENT

- If the receiving field is larger than the sending field, the receiving field is filled with leading 0’s in a numeric move:

```
01   X  PIC S9(3) VALUE 123.
01   Y  PIC S9(5) VALUE 0.
        MOVE X TO Y
 RESULT:    Y = +00123
```

MOVE STATEMENT

- If the receiving field is larger than the sending field, the receiving field is filled with trailing spaces in a alphanumeric move.

```
01   X  PIC X(3) VALUE “ABC”.
01   Y  PIC X(5) VALUE SPACES.
        MOVE X TO Y
 RESULT:    Y = ABC
```

MOVE STATEMENT

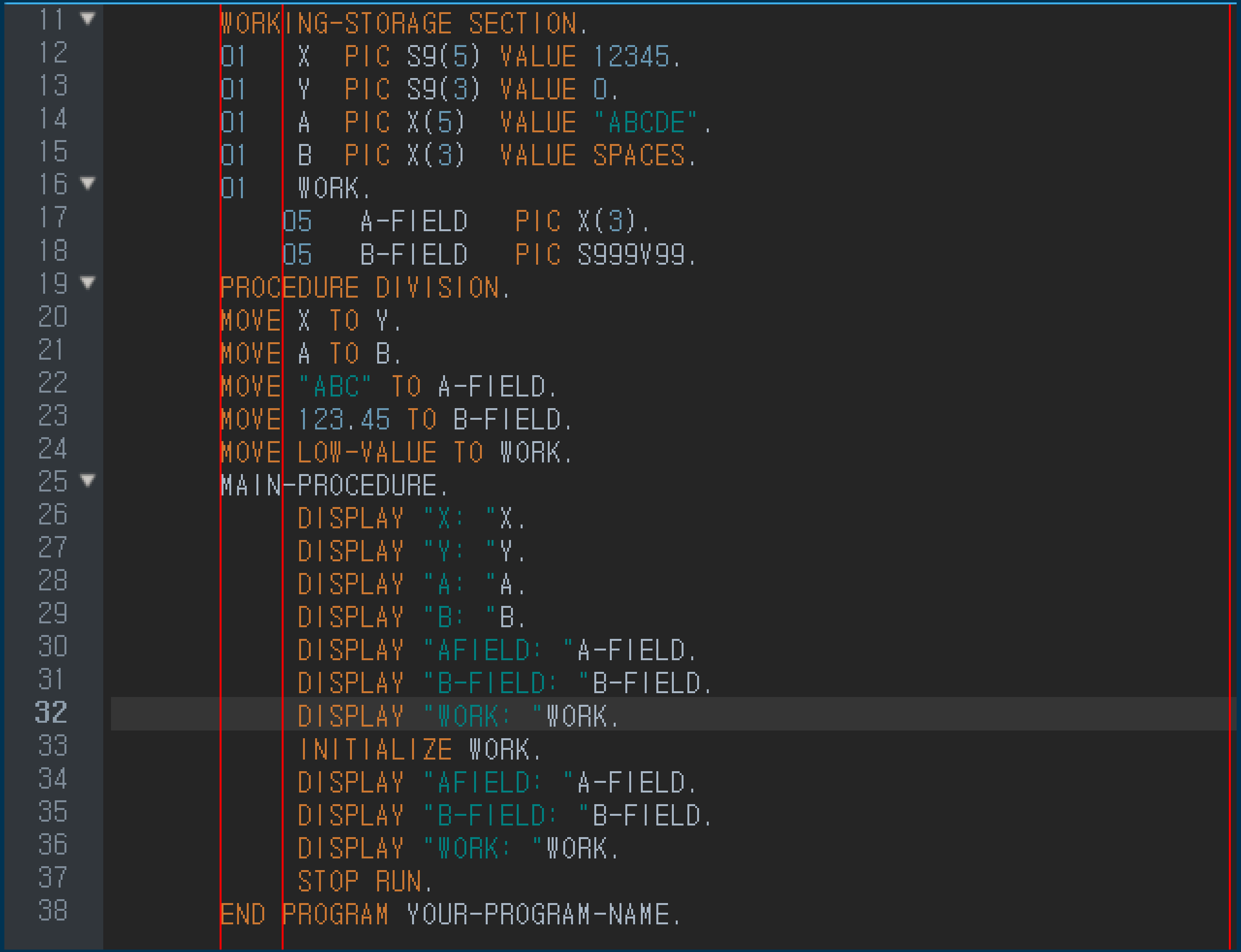
- If the receiving field is smaller than the sending field, data will be truncated on the left for numeric moves and on the right for alphanumeric moves

```
01    X   PIC S9(5) VALUE 12345.
01    Y   PIC S9(3) VALUE 0.
01    A   PIC X(5)  VALUE ‘ABCDE’
01    B   PIC X(3)  VALUE SPACES.
          MOVE X TO Y
          MOVE A TO B
 RESULT:      Y = +345
              B = ABC
```

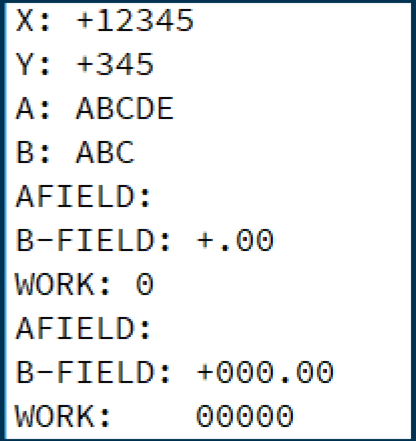
INITIALIZE

- **`SPACE`** is the implied sending item for receiving items of category alphabetic, alphanumeric, alphanumeric-edited, DBCS, national, or national-edited.
- **`ZERO`** is the implied sending item for receiving items of category numeric or numeric-edited.

INITIALIZE



```
WORKING-STORAGE SECTION.
01   X  PIC S9(5) VALUE 12345.
01   Y  PIC S9(3) VALUE 0.
01   A  PIC X(5)  VALUE "ABCDE".
01   B  PIC X(3)  VALUE SPACES.
01   WORK.
    05   A-FIELD   PIC X(3).
    05   B-FIELD   PIC S999V99.
PROCEDURE DIVISION.
MOVE X TO Y.
MOVE A TO B.
MOVE "ABC" TO A-FIELD.
MOVE 123.45 TO B-FIELD.
MOVE LOW-VALUE TO WORK.
MAIN-PROCEDURE.
     DISPLAY "X: "X.
     DISPLAY "Y: "Y.
     DISPLAY "A: "A.
     DISPLAY "B: "B.
     DISPLAY "AFIELD: "A-FIELD.
     DISPLAY "B-FIELD: "B-FIELD.
     DISPLAY "WORK: "WORK.
     INITIALIZE WORK.
     DISPLAY "AFIELD: "A-FIELD.
     DISPLAY "B-FIELD: "B-FIELD.
     DISPLAY "WORK: "WORK.
     STOP RUN.
END PROGRAM YOUR-PROGRAM-NAME.
```



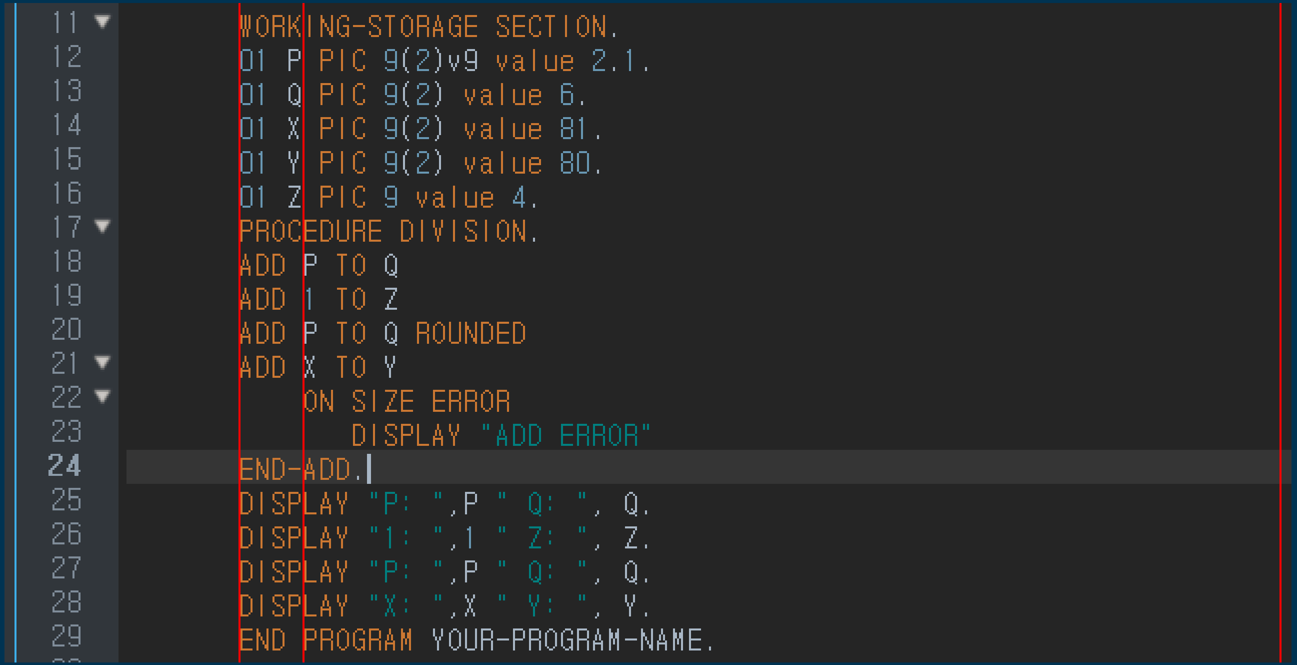
```
X: +12345
Y: +345
A: ABCDE
B: ABC
AFIELD:
B-FIELD: +.00
WORK: 0
AFIELD:
B-FIELD: +000.00
WORK:    00000
```

ADD Semantics

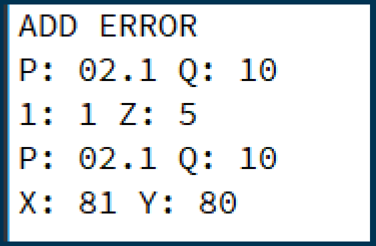
- All identifiers or literals that precede the keyword **`TO`** are added together, and this sum is added to and stored in *identifier-2*. This process is repeated for each successive occurrence of *identifier-2* in the left-to-right order in which *identifier-2* is specified.

```
ADD X Y Z TO  P Q
Before X=1, Y=2, Z=3, P=4,  Q=6
After  X=1, Y=2, Z=3, P=10, Q=12
```

ADD EXAMPLES



```
WORKING-STORAGE SECTION.
01 P PIC 9(2)v9 value 2.1.
01 Q PIC 9(2) value 6.
01 X PIC 9(2) value 81.
01 Y PIC 9(2) value 80.
01 Z PIC 9 value 4.
PROCEDURE DIVISION.
ADD P TO Q
ADD 1 TO Z
ADD P TO Q ROUNDED
ADD X TO Y
    ON SIZE ERROR
       DISPLAY "ADD ERROR"
END-ADD.
DISPLAY "P: ",P " Q: ", Q.
DISPLAY "1: ",1 " Z: ", Z.
DISPLAY "P: ",P " Q: ", Q.
DISPLAY "X: ",X " Y: ", Y.
END PROGRAM YOUR-PROGRAM-NAME.
```



```
ADD ERROR
P: 02.1 Q: 10
1: 1 Z: 5
P: 02.1 Q: 10
X: 81 Y: 80
```

ADD…GIVING Semantics

- All identifiers or literals that precede the keyword **TO** are added together, and this sum is added to *identifier-2* to obtain a temporary sum. (Identifier-2 is unchanged)
- The the temporary sum is moved to identifier-3.

```
ADD X Y Z TO V GIVING P
```

Before **X=1, Y=2, Z=3, V=4, P=6**

After **X=1, Y=2, Z=3, V=4, P=10**

SUBTRACT

- All identifiers or literals preceding the keyword **FROM** are added together and their sum is subtracted from and stored immediately in *identifier-2*. This process is repeated for each successive occurrence of *identifier-2*, in the left-to-right order in which *identifier-2* is specified.

```
SUBTRACT X Y FROM P Q
```

Before: **X=1,Y=2, P=3,Q=4**

After: **X=1,Y=2, P=0,Q=1**

SUBTRACT Semantics

- All identifiers or literals preceding the keyword **`FROM`** are added together and their sum is subtracted from *identifier-2* to obtain a temporary value which is moved to *identifier-3*.

```
SUBTRACT X Y FROM P GIVING Q
```

Before: **`X=1,Y=2,P=5,Q=6`**

After: **`X=1,Y=2,P=5,Q=2`**

MULTIPLY Semantics

- In format 1, the value of *identifier-1* or *literal-1* is multiplied by the value of *identifier-2*; the product is then placed in *identifier-2*. For each successive occurrence of *identifier-2*, the multiplication takes place in the left-to-right order in which *identifier-2* is specified.

```
MULTIPLY X BY P Q
```

Before: `X=2,P=4,Q=5`

After: `X=2,P=8,Q=10`

MULTIPLY

- In format 2, the value of *identifier-1* or *literal-1* is multiplied by the value of *identifier-2* or *literal-2*. The product is then stored in the data items referenced by *identifier-3*. Identifier-2 is unchanged.

```
MULTIPLY X BY Y GIVING Z
```

Before: `X=2, Y=3, Z=4`

After: `X=2, Y=3, Z=6`

DIVIDE

- In format 1, the value of *identifier-1* or *literal-1* is divided into the value of *identifier-2*, and the quotient is then stored in *identifier-2*. For each successive occurrence of *identifier-2*, the division takes place in the left-to-right order in which *identifier-2* is specified.

```
DIVIDE X INTO Y Z
```

Before: `X=3, Y=7, Z=12`

After: `X=3, Y=2, Z=4`

DIVIDE

- In format 2, the value of *identifier-1* or *literal-1* is divided into the value of *identifier-2* or *literal-2*. The value of the quotient is stored in each data item referenced by *identifier-3*.

```
DIVIDE X INTO Y GIVING Z
```

Before: `X = 2, Y = 13, Z = 1`

After: `X = 2, Y = 13, Z = 6`

DIVIDE

- In format 3, the value of *identifier-1* or *literal-1* is divided by the value of *identifier-2* or *literal-2*. The value of the quotient is stored in each data item referenced by *identifier-3*.

```
DIVIDE X BY Y GIVING Z
```

Before: `X = 10, Y = 3, Z = 1`

After: `X = 10, Y = 3, Z = 3`

DIVIDE

- In format 4, the value of *identifier-1* or *literal-1* is divided into *identifier-2* or *literal-2*. The value of the quotient is stored in *identifier-3*, and the value of the remainder is stored in *identifier-4*.

```
DIVIDE X INTO Y
       GIVING Z
       REMAINDER R
```

Before: `X = 2, Y = 9, Z = 8, R = 7`

After: `X = 2, Y = 9, Z = 4, R = 1`

COMPUTE

- **`COMPUTE`** can be used to initialize a numeric field
- Usually reserved for nontrivial computations. For simple computations choose **`ADD, SUBTRACT, MULTIPLY or DIVIDE`**

```
05  X   PIC    S9(4)V9.
COMPUTE X ROUNDED = (A + B) / 2.3
          ON SIZE ERROR
          DISPLAY “X WAS TRUNCATED”
END-COMPUTE
```

Arithmetic Operators

Operation	Operator
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponentiation

Parentheses provide precedence.

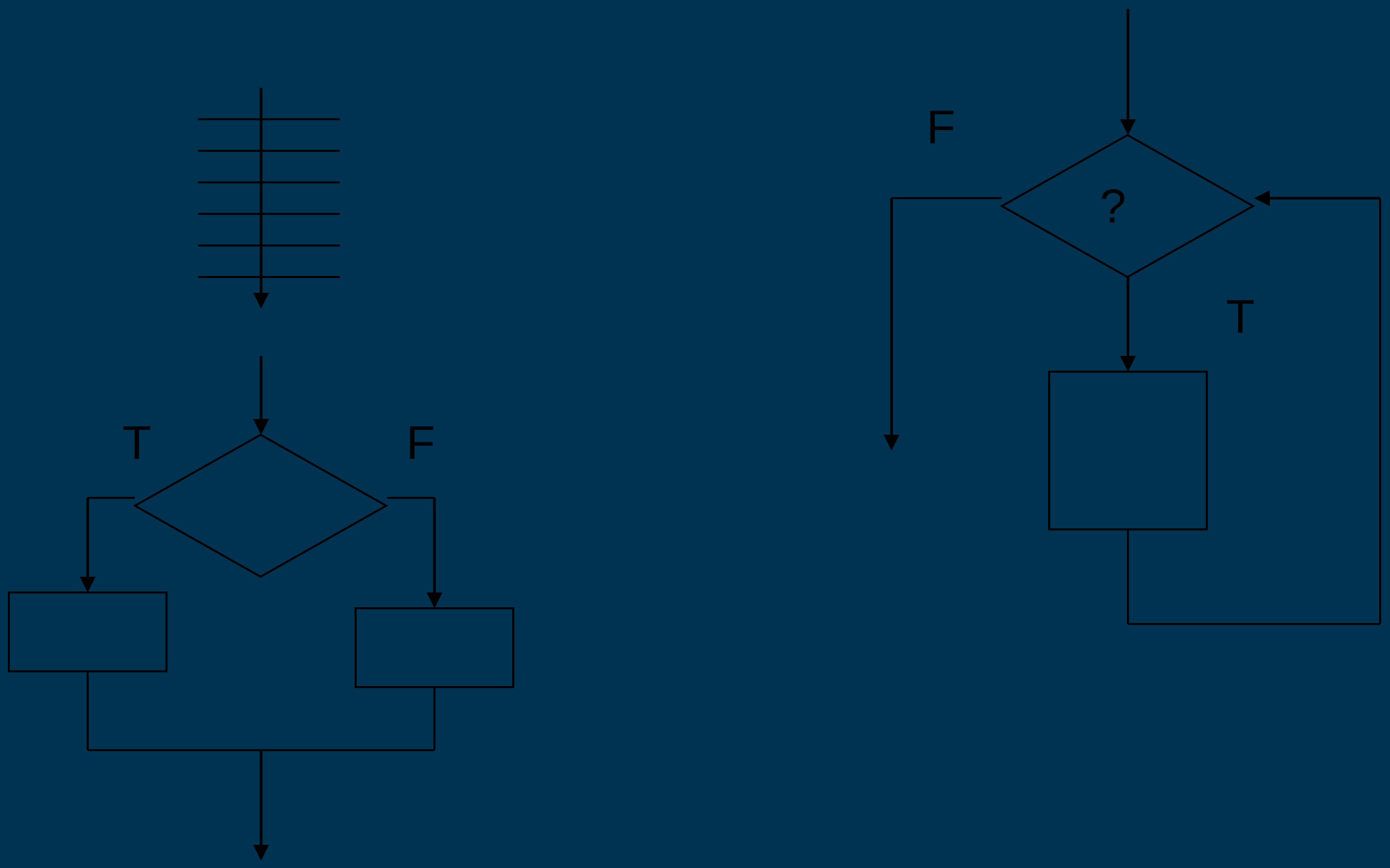
Always parenthesize!

```
((X + Y) * ( Z ** 3))
```

FLOW OF CONTROL

- There is a theoretical result in Computer Science by two Italian mathematicians, Boehm and Jacopini, that states that only 3 control structures are required to write any program:
- Sequence - Do this, now do this, now do this, …
- Selection - If something is true do this, else do that
- Repetition – While something is true, do this
- Practice has shown that being able to create procedures is helpful in overcoming complexity, but they aren’t strictly necessary
- One implication of this result is that GO TO statements aren’t needed ☺

FLOW OF CONTROL



IF

- The condition is tested and either the true or false blocks are selected for execution
- Don’t use **`NEXT SENTENCE`** if you are using **`END-IF`** as the delimiter (and you should). Use of **`NEXT SENTENCE`** causes execution to continue with the next closest period, which is probably the end of the paragraph.

IF Examples

```
IF  X < Y
    ADD 1 TO X
    DISPLAY “AAA”
ELSE
    DISPLAY “BBB”
END-IF

IF  X > Y
    DISPLAY “X WAS BIGGER”
END-IF
```

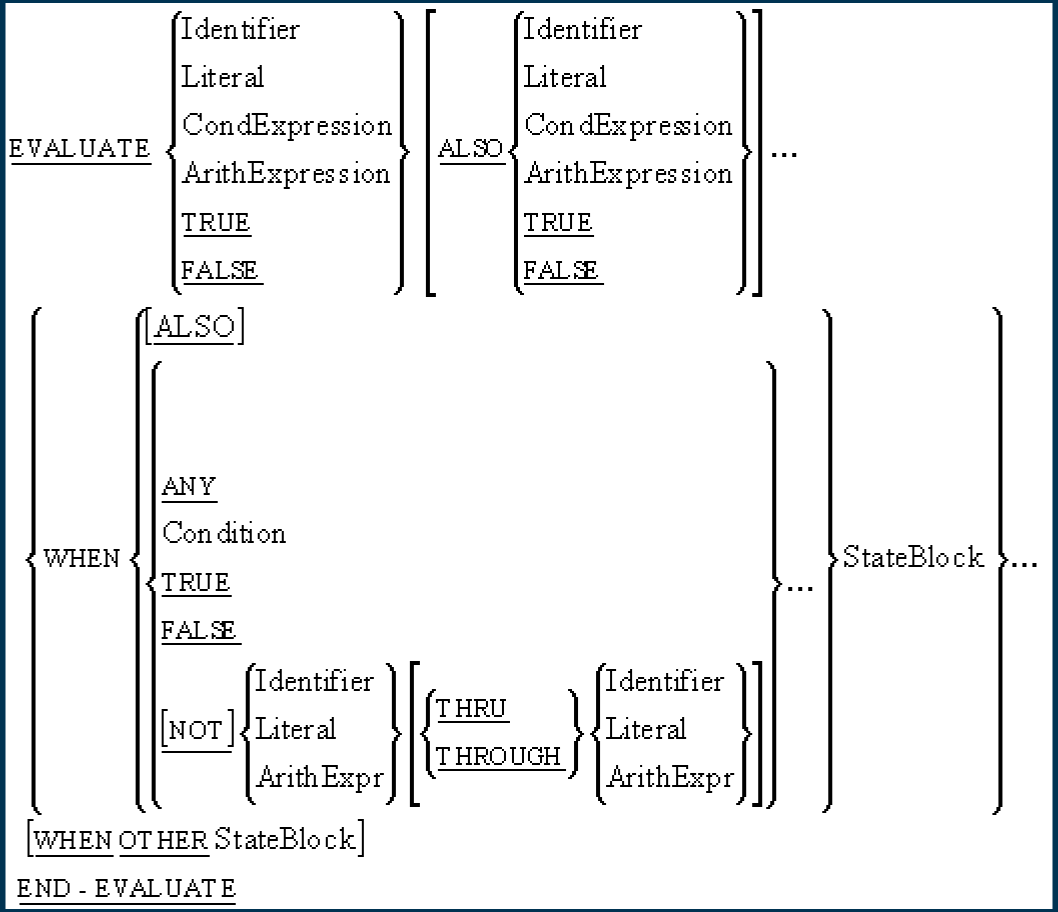
NESTED IFs

- Each ELSE is matched with the nearest preceding IF

```
IF X < Y
     DISPLAY “XXX”
     IF  Y < Z
         DISPLAY “ZZZ”
ELSE
     DISPLAY “AAA”
END-IF
```

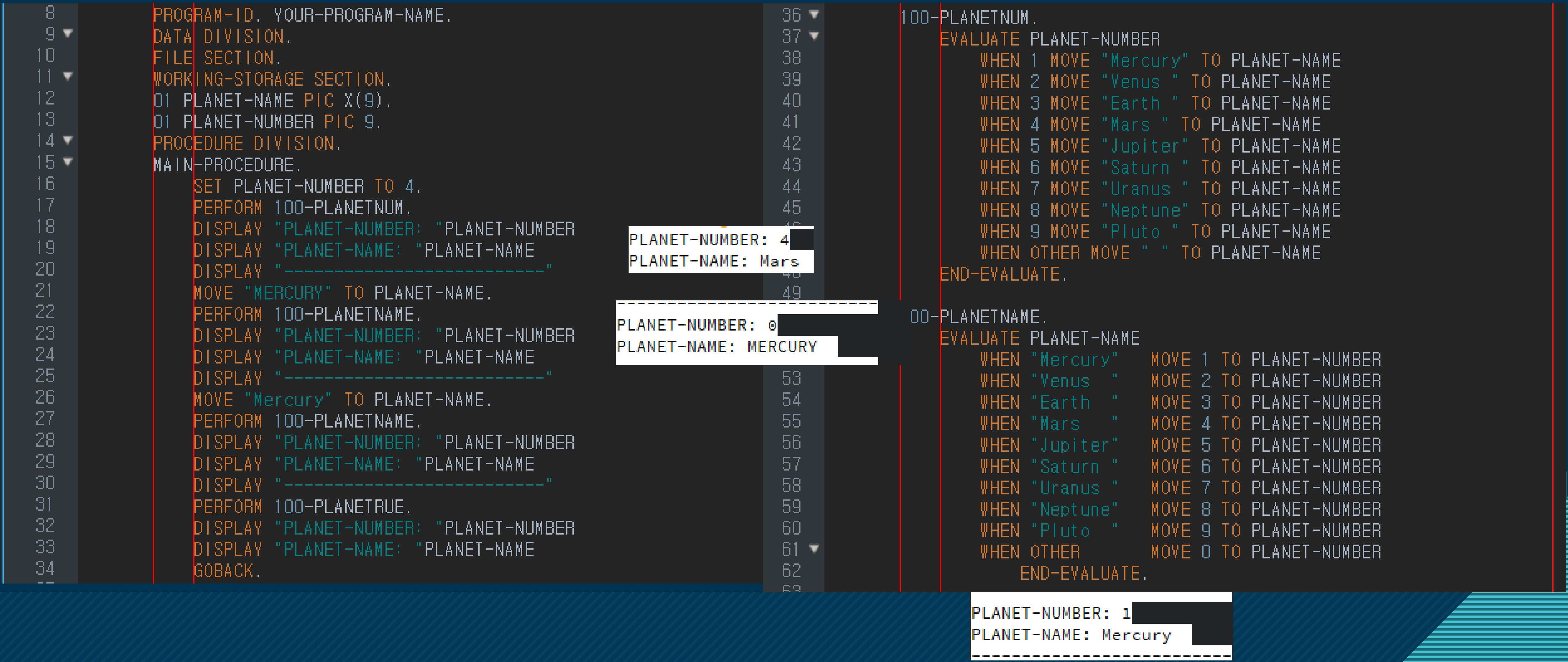
- MORAL: Indent properly and terminate all if statements with END-IF

EVALUATE



```
          Identifier              Identifier
          Literal                 Literal
EVALUATE  CondExpression    ALSO  CondExpression   ...
          ArithExpression         ArithExpression
          TRUE                    TRUE
          FALSE                   FALSE

          [ALSO]

          ANY
          Condition
  WHEN    TRUE                                                ... StateBlock ...
          FALSE
                   Identifier    THRU       Identifier
          [NOT]    Literal                  Literal
                   ArithExpr     THROUGH    ArithExpr
 [WHEN OTHER StateBlock]
END - EVALUATE
```

EVALUATE



```
       PROGRAM-ID. YOUR-PROGRAM-NAME.
       DATA DIVISION.
       FILE SECTION.
       WORKING-STORAGE SECTION.
       01 PLANET-NAME PIC X(9).
       01 PLANET-NUMBER PIC 9.
       PROCEDURE DIVISION.
       MAIN-PROCEDURE.
           SET PLANET-NUMBER TO 4.
           PERFORM 100-PLANETNUM.
           DISPLAY "PLANET-NUMBER: "PLANET-NUMBER
           DISPLAY "PLANET-NAME: "PLANET-NAME
           DISPLAY "--------------------------"
           MOVE "MERCURY" TO PLANET-NAME.
           PERFORM 100-PLANETNAME.
           DISPLAY "PLANET-NUMBER: "PLANET-NUMBER
           DISPLAY "PLANET-NAME: "PLANET-NAME
           DISPLAY "--------------------------"
           MOVE "Mercury" TO PLANET-NAME.
           PERFORM 100-PLANETNAME.
           DISPLAY "PLANET-NUMBER: "PLANET-NUMBER
           DISPLAY "PLANET-NAME: "PLANET-NAME
           DISPLAY "--------------------------"
           PERFORM 100-PLANETRUE.
           DISPLAY "PLANET-NUMBER: "PLANET-NUMBER
           DISPLAY "PLANET-NAME: "PLANET-NAME
           GOBACK.
```

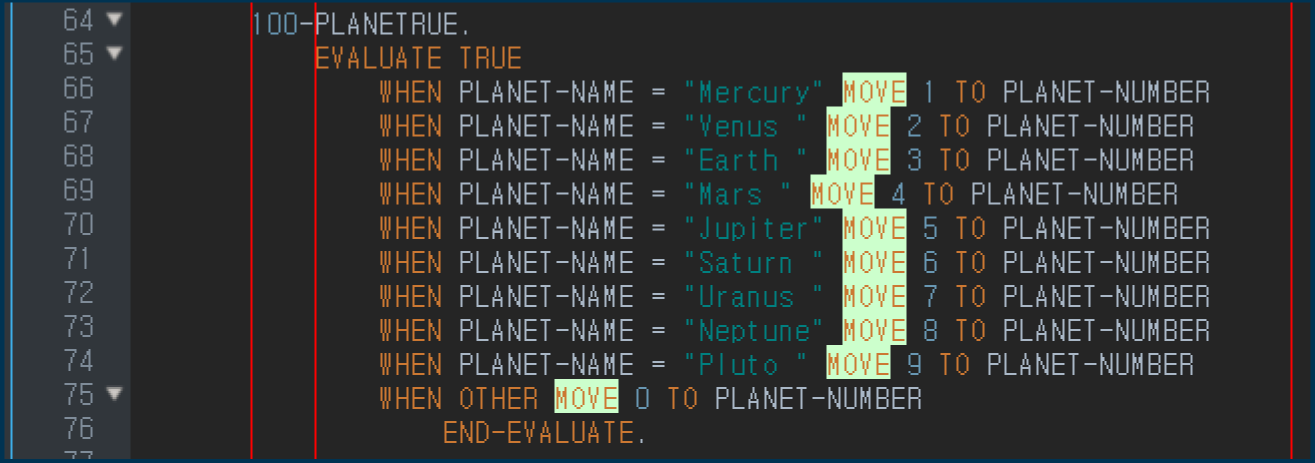
```
PLANET-NUMBER: 4
PLANET-NAME: Mars
```

```
--------------------------
PLANET-NUMBER: 0
PLANET-NAME: MERCURY
```

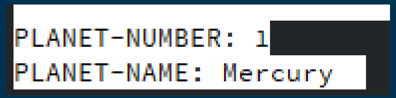
```
       100-PLANETNUM.
           EVALUATE PLANET-NUMBER
               WHEN 1 MOVE "Mercury" TO PLANET-NAME
               WHEN 2 MOVE "Venus  " TO PLANET-NAME
               WHEN 3 MOVE "Earth  " TO PLANET-NAME
               WHEN 4 MOVE "Mars  " TO PLANET-NAME
               WHEN 5 MOVE "Jupiter" TO PLANET-NAME
               WHEN 6 MOVE "Saturn " TO PLANET-NAME
               WHEN 7 MOVE "Uranus " TO PLANET-NAME
               WHEN 8 MOVE "Neptune" TO PLANET-NAME
               WHEN 9 MOVE "Pluto  " TO PLANET-NAME
               WHEN OTHER MOVE "  " TO PLANET-NAME
           END-EVALUATE.

        00-PLANETNAME.
           EVALUATE PLANET-NAME
               WHEN "Mercury"    MOVE 1 TO PLANET-NUMBER
               WHEN "Venus  "    MOVE 2 TO PLANET-NUMBER
               WHEN "Earth  "    MOVE 3 TO PLANET-NUMBER
               WHEN "Mars   "    MOVE 4 TO PLANET-NUMBER
               WHEN "Jupiter"    MOVE 5 TO PLANET-NUMBER
               WHEN "Saturn "    MOVE 6 TO PLANET-NUMBER
               WHEN "Uranus "    MOVE 7 TO PLANET-NUMBER
               WHEN "Neptune"    MOVE 8 TO PLANET-NUMBER
               WHEN "Pluto  "    MOVE 9 TO PLANET-NUMBER
               WHEN OTHER        MOVE 0 TO PLANET-NUMBER
                   END-EVALUATE.
```

```
PLANET-NUMBER: 1
PLANET-NAME: Mercury
--------------------------
```

EVALUATE



```
100-PLANETRUE.
    EVALUATE TRUE
        WHEN PLANET-NAME = "Mercury" MOVE 1 TO PLANET-NUMBER
        WHEN PLANET-NAME = "Venus " MOVE 2 TO PLANET-NUMBER
        WHEN PLANET-NAME = "Earth " MOVE 3 TO PLANET-NUMBER
        WHEN PLANET-NAME = "Mars " MOVE 4 TO PLANET-NUMBER
        WHEN PLANET-NAME = "Jupiter" MOVE 5 TO PLANET-NUMBER
        WHEN PLANET-NAME = "Saturn " MOVE 6 TO PLANET-NUMBER
        WHEN PLANET-NAME = "Uranus " MOVE 7 TO PLANET-NUMBER
        WHEN PLANET-NAME = "Neptune" MOVE 8 TO PLANET-NUMBER
        WHEN PLANET-NAME = "Pluto " MOVE 9 TO PLANET-NUMBER
        WHEN OTHER MOVE 0 TO PLANET-NUMBER
            END-EVALUATE.
```



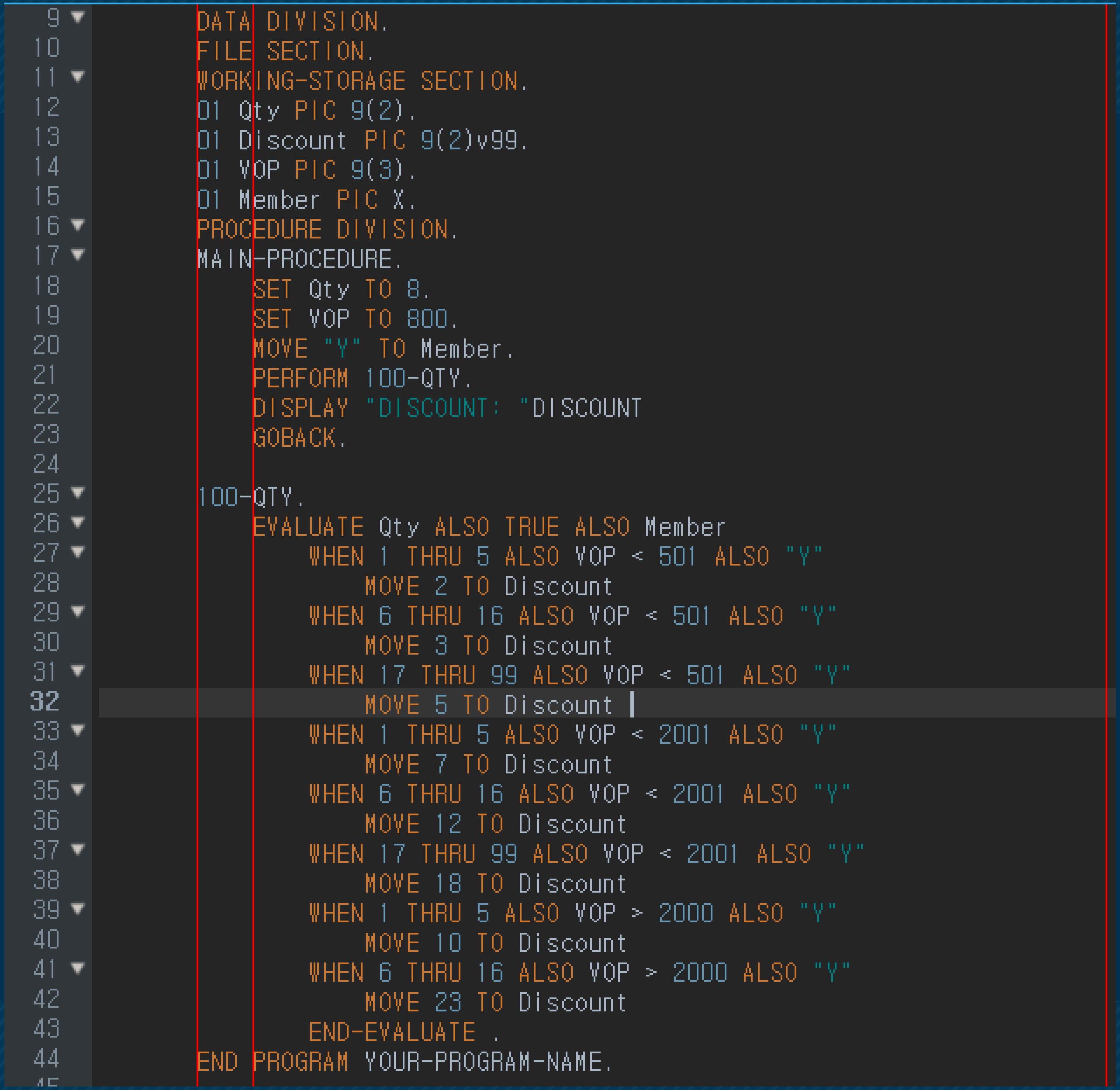
```
PLANET-NUMBER: 1
PLANET-NAME: Mercury
```

EVALUATE

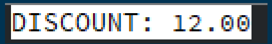
```
EVALUATE PLANET-NUMBER // 7
     WHEN 1 MOVE "Mercury" TO PLANET-NAME
     WHEN 2 MOVE "Venus " TO PLANET-NAME
     WHEN 3 MOVE "Earth " TO PLANET-NAME
     WHEN 4 MOVE "Mars " TO PLANET-NAME
     WHEN 5 MOVE "Jupiter" TO PLANET-NAME
     WHEN 6 MOVE "Saturn " TO PLANET-NAME
     WHEN 7 MOVE "Uranus " TO PLANET-NAME
     WHEN 8 MOVE "Neptune" TO PLANET-NAME
     WHEN 9 MOVE "Pluto " TO PLANET-NAME
     WHEN OTHER MOVE " " TO PLANET-NAME
END-EVALUATE.   //PLANET-NAME = Uranus
```

```
SWITCH(PLANET-NUMBER):
     case 1: PLANET-NAME =
           “MERCURY”;
     …
     …
     …
     …
     …
     …
     …
     default: PLANET-NAME
                 = “ ”.
```

EVALUATE



```
       DATA DIVISION.
       FILE SECTION.
       WORKING-STORAGE SECTION.
       01 Qty PIC 9(2).
       01 Discount PIC 9(2)v99.
       01 VOP PIC 9(3).
       01 Member PIC X.
       PROCEDURE DIVISION.
       MAIN-PROCEDURE.
           SET Qty TO 8.
           SET VOP TO 800.
           MOVE "Y" TO Member.
           PERFORM 100-QTY.
           DISPLAY "DISCOUNT: "DISCOUNT
           GOBACK.

       100-QTY.
           EVALUATE Qty ALSO TRUE ALSO Member
               WHEN 1 THRU 5 ALSO VOP < 501 ALSO "Y"
                   MOVE 2 TO Discount
               WHEN 6 THRU 16 ALSO VOP < 501 ALSO "Y"
                   MOVE 3 TO Discount
               WHEN 17 THRU 99 ALSO VOP < 501 ALSO "Y"
                   MOVE 5 TO Discount
               WHEN 1 THRU 5 ALSO VOP < 2001 ALSO "Y"
                   MOVE 7 TO Discount
               WHEN 6 THRU 16 ALSO VOP < 2001 ALSO "Y"
                   MOVE 12 TO Discount
               WHEN 17 THRU 99 ALSO VOP < 2001 ALSO "Y"
                   MOVE 18 TO Discount
               WHEN 1 THRU 5 ALSO VOP > 2000 ALSO "Y"
                   MOVE 10 TO Discount
               WHEN 6 THRU 16 ALSO VOP > 2000 ALSO "Y"
                   MOVE 23 TO Discount
               END-EVALUATE .
       END PROGRAM YOUR-PROGRAM-NAME.
```



```
DISCOUNT: 12.00
```

PERFORM THRU

- PERFORM paragraph name THRU paragraph name

```
PERFORM 100-XXX THUR 100-XXX-EXIT
100-XXX.
       DISPLAY ‘IN 100-XXX’.
100-XXX-EXIT.
       EXIT.
```

- There is an implicit **`EXIT`** in every paragraph so why do I need to code it explicitly?

PERFORM x TIMES

```
MOVE 5 TO COUNT
PERFORM COUNT TIMES
       DISPLAY “XXX”
END-PERFORM

PERFORM 100-DISPLAY COUNT TIMES
```

PERFORM UNTIL

-
  ```
  MOVE 0 TO X
   PERFORM UNTIL X > 10
       MOVE X TO X-EDITED
       DISPLAY X-EDITED
       ADD 1 TO X
   END-PERFORM
  ```

-
  ```
  PERFORM X-PARA UNTIL X > 10
  ```

-
  ```
  PERFORM X-PARA WITH TEST AFTER
                  UNTIL X > 10
  ```

Inline Perform

```
PERFORM VARYING X FROM 1 BY 1
                   UNTIL X > 100
      DISPLAY X
END-PERFORM
```

PRINTS:

```
1
2
3
…
100
```

Inline PERFORM

```
PERFORM VARYING X FROM 5 BY -1
                  UNTIL X =0
        DISPLAY X
END-PERFORM
```

PRINTS:

```
5
4
3
2
1
0
```

Inline PERFORM

```
MOVE 10 TO X
PERFORM WITH TEST AFTER
                   UNTIL  X = 0
       DISPLAY X
       SUBTRACT 1 FROM X
END-PERFORM
```

Alternate PERFORM

```
    PERFORM 100-PARA VARYING I FROM 1 BY 1 UNTIL I > 5
                 AFTER J FROM 1 BY 1 UNTIL J > 3
    END-PERFORM
100-PARA.
    DISPLAY I J
    .
```

```
1 1
1 2
1 3
2 1
2 2
2 3
3 1
3 2
3 3
4 1 …
```

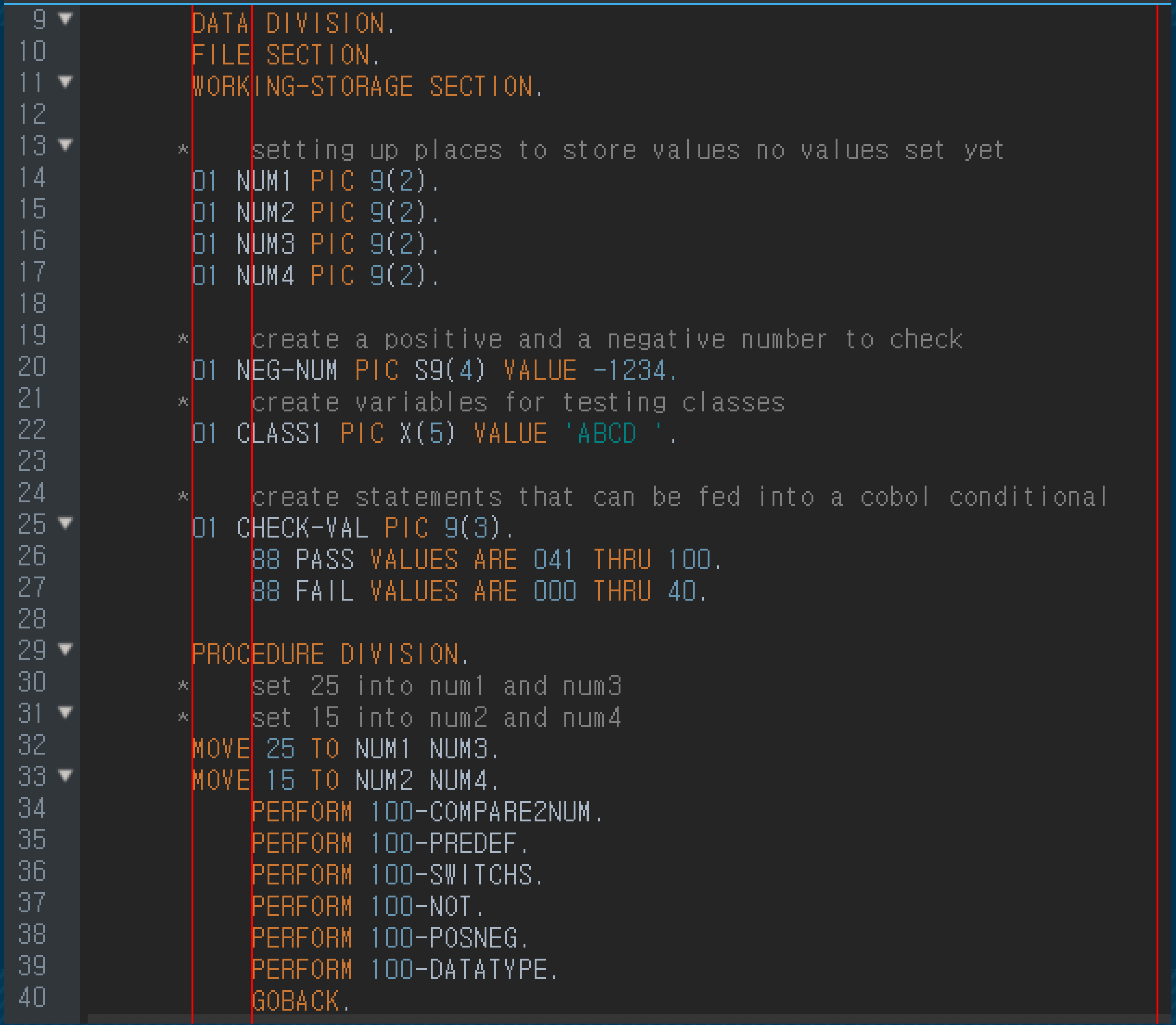
```
for(int I = 0; I < 5; I++){
     for(int j = 0; j < 3; j++){
             print(I + “ ” + J);
     }
}
```

EVALUATE

```
EVALUATE TRUE ALSO Position
  WHEN L-Arrow ALSO 2 THRU 10
    SUBTRACT 1 FROM Position
  WHEN R-Arrow ALSO 1 THRU 9
    ADD 1 TO Position
  WHEN L-Arrow ALSO 1
    MOVE 10 TO Position
  WHEN R-Arrow ALSO 10
    MOVE 1 TO Position
  WHEN DelKey ALSO ANY
    PERFORM DeleteChar
  WHEN Char ALSO 1 THRU 9
    PERFORM InsertChar
    ADD 1 TO Position
  WHEN Char ALSO 10
    PERFORM InsertChar
  WHEN OTHER PERFORM
    DisplayErrorMessage
END-EVALUATE
```

4-Conditional.cbl

DATA DIVISION AND PROCEDURE DIVISION



```
       DATA DIVISION.
       FILE SECTION.
       WORKING-STORAGE SECTION.

      *    setting up places to store values no values set yet
       01 NUM1 PIC 9(2).
       01 NUM2 PIC 9(2).
       01 NUM3 PIC 9(2).
       01 NUM4 PIC 9(2).

      *    create a positive and a negative number to check
       01 NEG-NUM PIC S9(4) VALUE -1234.
      *    create variables for testing classes
       01 CLASS1 PIC X(5) VALUE 'ABCD '.

      *    create statements that can be fed into a cobol conditional
       01 CHECK-VAL PIC 9(3).
           88 PASS VALUES ARE 041 THRU 100.
           88 FAIL VALUES ARE 000 THRU 40.

       PROCEDURE DIVISION.
      *    set 25 into num1 and num3
      *    set 15 into num2 and num4
       MOVE 25 TO NUM1 NUM3.
       MOVE 15 TO NUM2 NUM4.
           PERFORM 100-COMPARE2NUM.
           PERFORM 100-PREDEF.
           PERFORM 100-SWITCHS.
           PERFORM 100-NOT.
           PERFORM 100-POSNEG.
           PERFORM 100-DATATYPE.
           GOBACK.
```

Relation Condition

- Relation condition compares two operands, either of which can be an identifier, literal, or arithmetic expression. Algebraic comparison of numeric fields is done regardless of size and usage clause.
- **For non-numeric operands**
- If two non-numeric operands of equal size are compared, then the characters are compared from left with the corresponding positions till the end is reached. The operand containing greater number of characters is declared greater.
- If two non-numeric operands of unequal size are compared, then the shorter data item is appended with spaces at the end till the size of the operands becomes equal and then compared according to the rules mentioned in the previous point.

```
 [Data Name/Arithmetic Operation] [IS] [NOT] [Equal to (=),Greater than (>), Less than
 (<), Greater than or Equal (>=),
Less than or equal (<=) ] [Data Name/Arithmetic Operation]
```

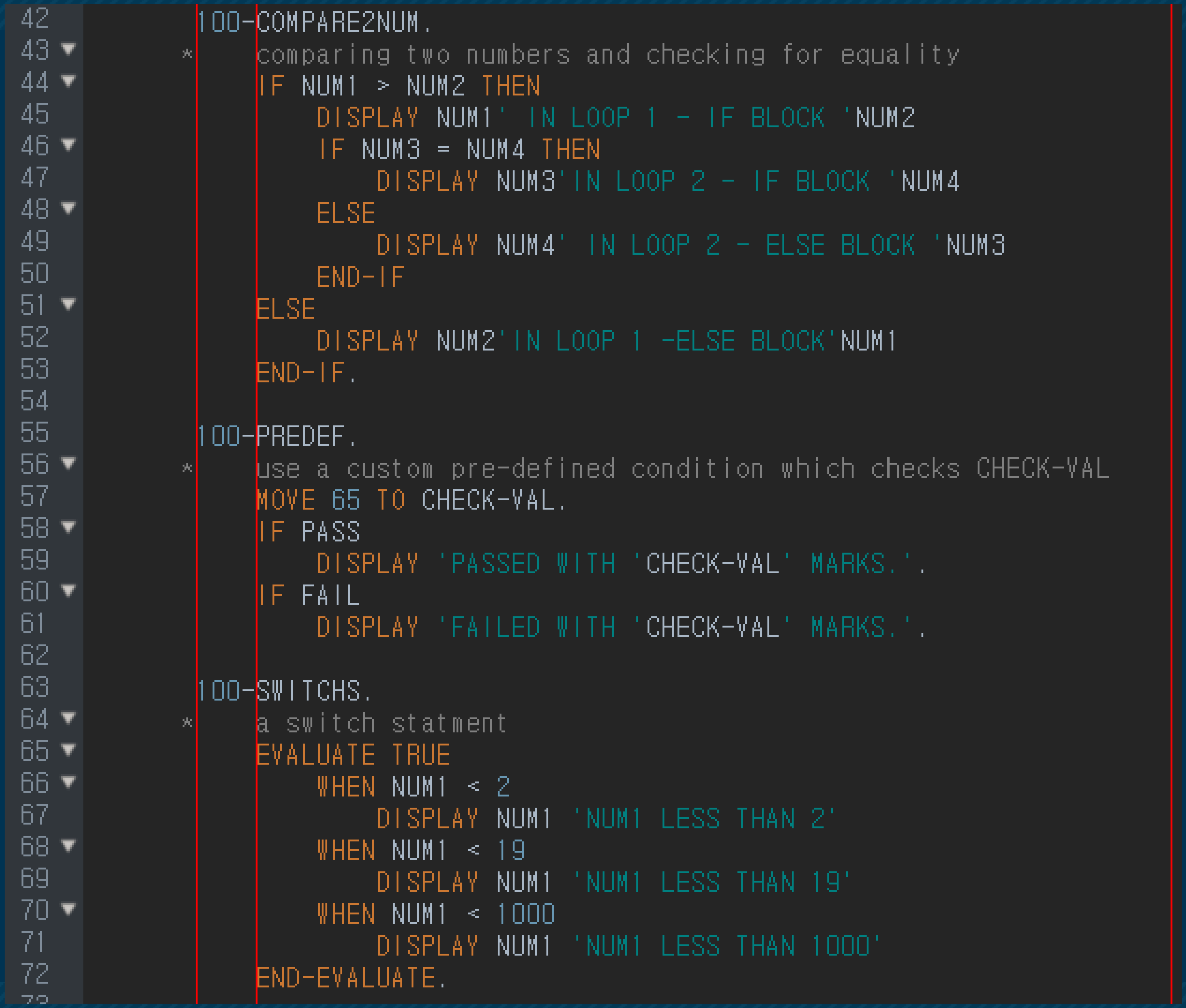
Condition-Name and Evaluate Verb Condition

- A condition-name is a user-defined name. It contains a set of values specified by the user. It behaves like Boolean variables. They are defined with level number 88. It will not have a PIC clause.

```
88 [Condition-Name] VALUE [IS, ARE] [LITERAL] [THRU
LITERAL].
```

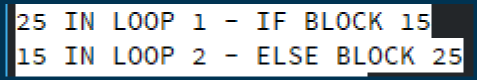
- Evaluate verb is a replacement of series of IF-ELSE statement. It can be used to evaluate more than one condition. It is similar to SWITCH statement in C programs.

DATA DIVISION



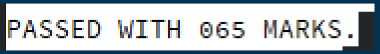
```
       100-COMPARE2NUM.
      *    comparing two numbers and checking for equality
           IF NUM1 > NUM2 THEN
               DISPLAY NUM1' IN LOOP 1 - IF BLOCK 'NUM2
               IF NUM3 = NUM4 THEN
                   DISPLAY NUM3'IN LOOP 2 - IF BLOCK 'NUM4
               ELSE
                   DISPLAY NUM4' IN LOOP 2 - ELSE BLOCK 'NUM3
               END-IF
           ELSE
               DISPLAY NUM2'IN LOOP 1 -ELSE BLOCK'NUM1
           END-IF.

       100-PREDEF.
      *    use a custom pre-defined condition which checks CHECK-VAL
           MOVE 65 TO CHECK-VAL.
           IF PASS
               DISPLAY 'PASSED WITH 'CHECK-VAL' MARKS.'.
           IF FAIL
               DISPLAY 'FAILED WITH 'CHECK-VAL' MARKS.'.

       100-SWITCHS.
      *    a switch statment
           EVALUATE TRUE
               WHEN NUM1 < 2
                   DISPLAY NUM1 'NUM1 LESS THAN 2'
               WHEN NUM1 < 19
                   DISPLAY NUM1 'NUM1 LESS THAN 19'
               WHEN NUM1 < 1000
                   DISPLAY NUM1 'NUM1 LESS THAN 1000'
           END-EVALUATE.
```

Relation Condition



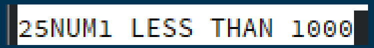
```
25 IN LOOP 1 - IF BLOCK 15
15 IN LOOP 2 - ELSE BLOCK 25
```

Condition-Name Condition



```
PASSED WITH 065 MARKS.
```

Evaluate Verb Condition



```
25NUM1 LESS THAN 1000
```

Negated and Combined Condition

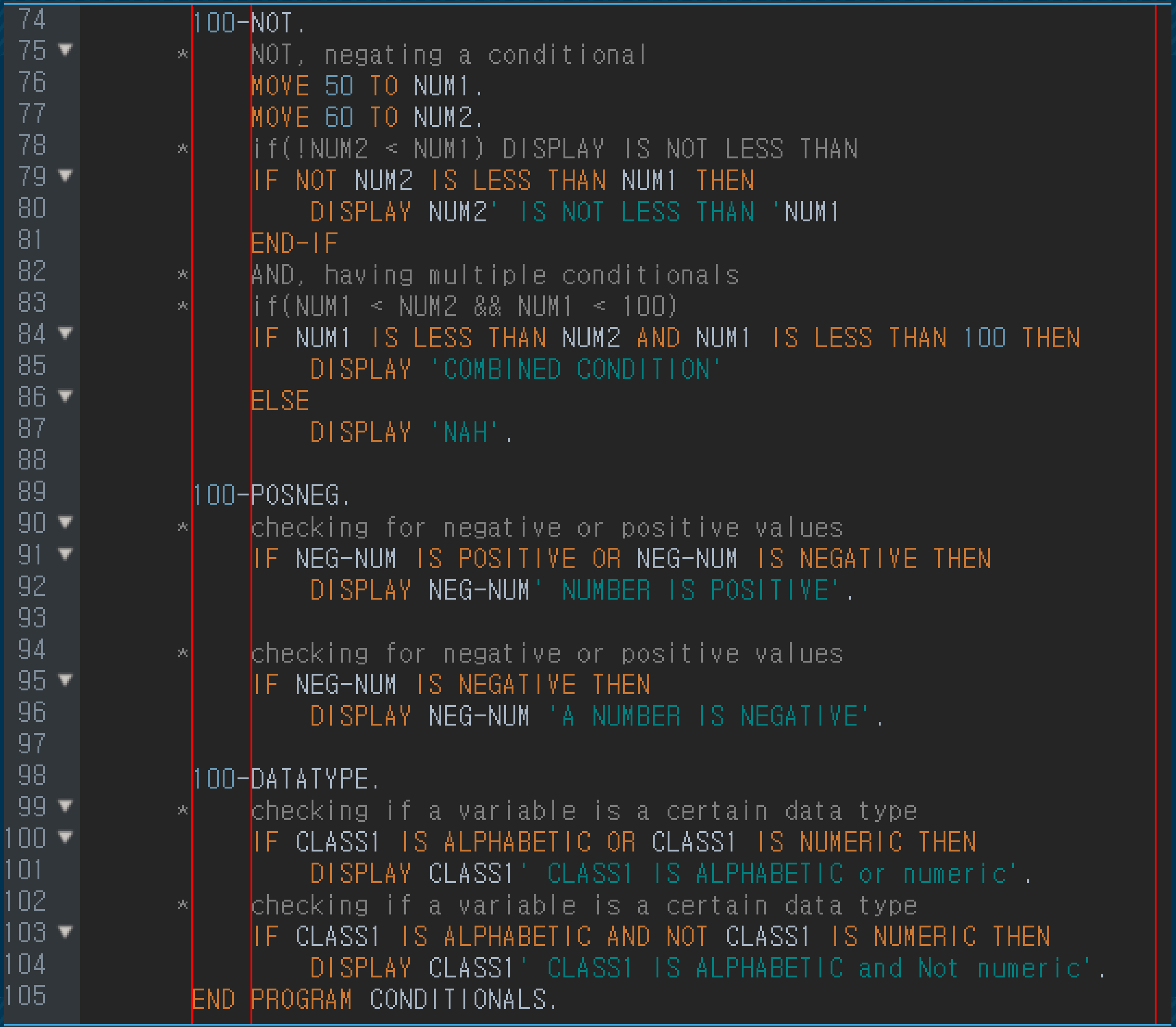
- Negated condition is given by using the NOT keyword. If a condition is true and we have given NOT in front of it, then its final value will be false.

```
IF NOT [CONDITION] COBOL Statements END-IF.
```

- A combined condition contains two or more conditions connected using logical operators AND or OR

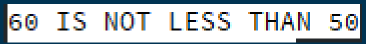
```
IF [CONDITION] AND [CONDITION] COBOL Statements END-IF.
```

DATA DIVISION

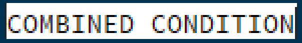


```
       100-NOT.
      *    NOT, negating a conditional
           MOVE 50 TO NUM1.
           MOVE 60 TO NUM2.
      *    if(!NUM2 < NUM1) DISPLAY IS NOT LESS THAN
           IF NOT NUM2 IS LESS THAN NUM1 THEN
               DISPLAY NUM2' IS NOT LESS THAN 'NUM1
           END-IF
      *    AND, having multiple conditionals
      *    if(NUM1 < NUM2 && NUM1 < 100)
           IF NUM1 IS LESS THAN NUM2 AND NUM1 IS LESS THAN 100 THEN
               DISPLAY 'COMBINED CONDITION'
           ELSE
               DISPLAY 'NAH'.

       100-POSNEG.
      *    checking for negative or positive values
           IF NEG-NUM IS POSITIVE OR NEG-NUM IS NEGATIVE THEN
               DISPLAY NEG-NUM' NUMBER IS POSITIVE'.

      *    checking for negative or positive values
           IF NEG-NUM IS NEGATIVE THEN
               DISPLAY NEG-NUM 'A NUMBER IS NEGATIVE'.

       100-DATATYPE.
      *    checking if a variable is a certain data type
           IF CLASS1 IS ALPHABETIC OR CLASS1 IS NUMERIC THEN
               DISPLAY CLASS1' CLASS1 IS ALPHABETIC or numeric'.
      *    checking if a variable is a certain data type
           IF CLASS1 IS ALPHABETIC AND NOT CLASS1 IS NUMERIC THEN
               DISPLAY CLASS1' CLASS1 IS ALPHABETIC and Not numeric'.
       END PROGRAM CONDITIONALS.
```

Negated Condition

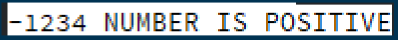


```
60 IS NOT LESS THAN 50
```

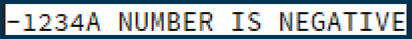


```
COMBINED CONDITION
```

SIGN Condition

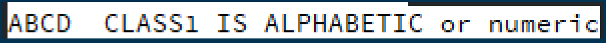


```
-1234 NUMBER IS POSITIVE
```

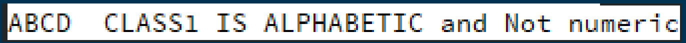


```
-1234A NUMBER IS NEGATIVE
```

CLASS Condition



```
ABCD  CLASS1 IS ALPHABETIC or numeric
```



```
ABCD  CLASS1 IS ALPHABETIC and Not numeric
```

SIGN And CLASS Condition

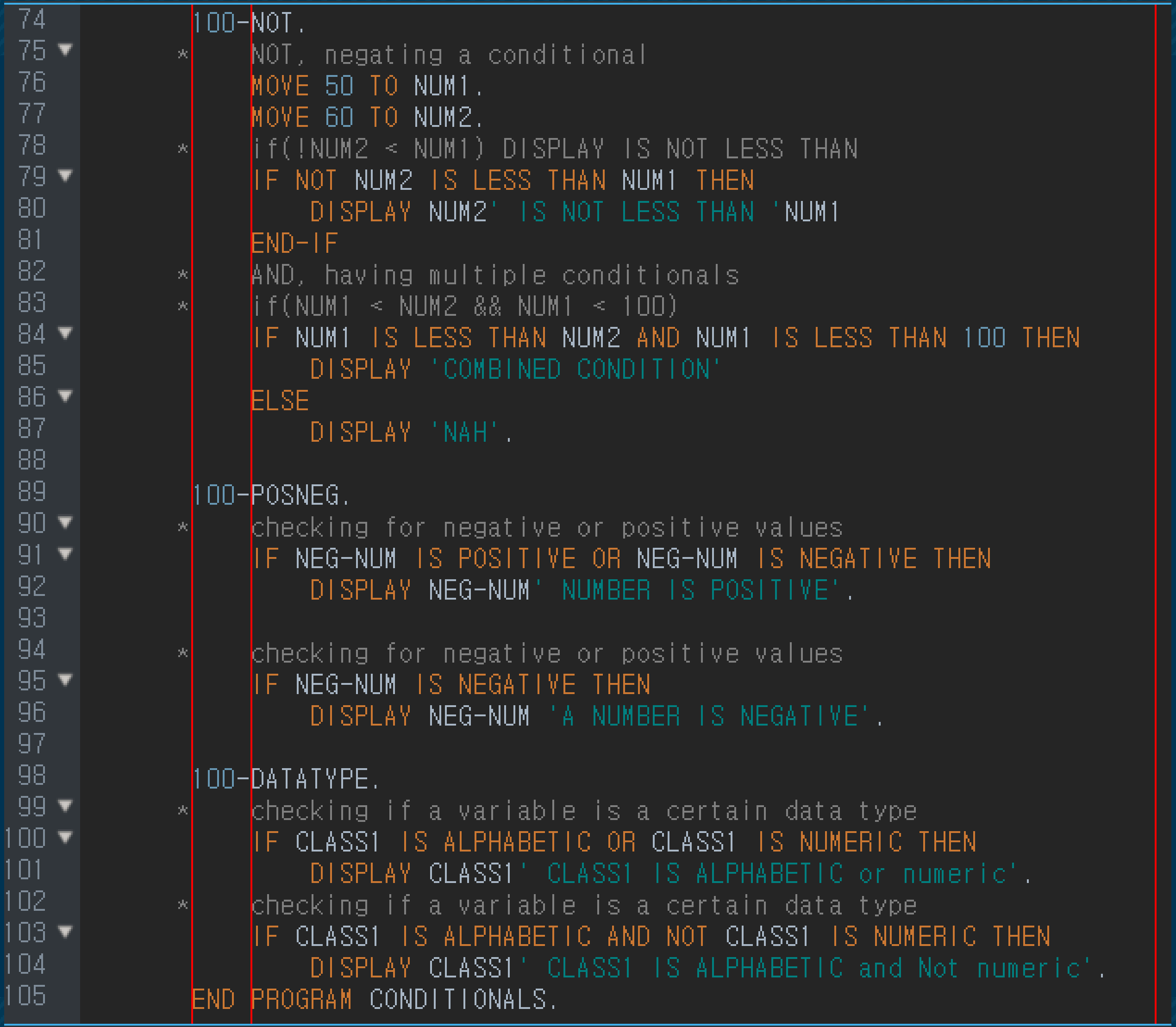
- Sign condition is used to check the sign of a numeric operand. It determines whether a given numeric value is greater than, less than, or equal to ZERO.

```
[Data Name/Arithmetic Operation] [IS] [NOT] [Positive, Negative or Zero] [Data
Name/Arithmetic Operation]
```

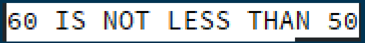
- Class condition is used to check if an operand contains only alphabets or numeric data. Spaces are considered in ALPHABETIC, ALPHABETIC-LOWER, and ALPHABETIC-UPPER.

```
[Data Name/Arithmetic Operation>] [IS] [NOT] [NUMERIC, ALPHABETIC,
ALPHABETIC-LOWER, ALPHABETIC-UPPER] [Data Name/Arithmetic Operation]
```

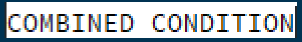
DATA DIVISION



```
       100-NOT.
      *    NOT, negating a conditional
           MOVE 50 TO NUM1.
           MOVE 60 TO NUM2.
      *    if(!NUM2 < NUM1) DISPLAY IS NOT LESS THAN
           IF NOT NUM2 IS LESS THAN NUM1 THEN
               DISPLAY NUM2' IS NOT LESS THAN 'NUM1
           END-IF
      *    AND, having multiple conditionals
      *    if(NUM1 < NUM2 && NUM1 < 100)
           IF NUM1 IS LESS THAN NUM2 AND NUM1 IS LESS THAN 100 THEN
               DISPLAY 'COMBINED CONDITION'
           ELSE
               DISPLAY 'NAH'.

       100-POSNEG.
      *    checking for negative or positive values
           IF NEG-NUM IS POSITIVE OR NEG-NUM IS NEGATIVE THEN
               DISPLAY NEG-NUM' NUMBER IS POSITIVE'.

      *    checking for negative or positive values
           IF NEG-NUM IS NEGATIVE THEN
               DISPLAY NEG-NUM 'A NUMBER IS NEGATIVE'.

       100-DATATYPE.
      *    checking if a variable is a certain data type
           IF CLASS1 IS ALPHABETIC OR CLASS1 IS NUMERIC THEN
               DISPLAY CLASS1' CLASS1 IS ALPHABETIC or numeric'.
      *    checking if a variable is a certain data type
           IF CLASS1 IS ALPHABETIC AND NOT CLASS1 IS NUMERIC THEN
               DISPLAY CLASS1' CLASS1 IS ALPHABETIC and Not numeric'.
       END PROGRAM CONDITIONALS.
```

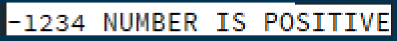


```
60 IS NOT LESS THAN 50
```

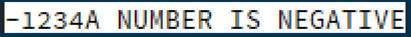


```
COMBINED CONDITION
```

SIGN Condition

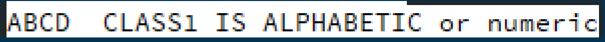


```
-1234 NUMBER IS POSITIVE
```

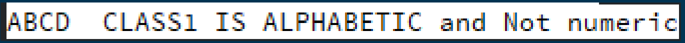


```
-1234A NUMBER IS NEGATIVE
```

CLASS Condition



```
ABCD  CLASS1 IS ALPHABETIC or numeric
```



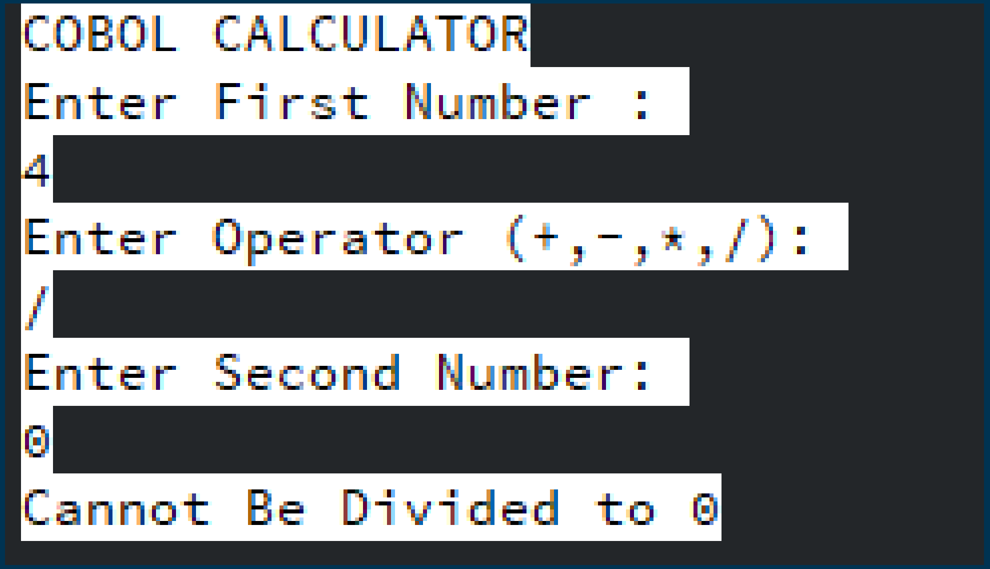
```
ABCD  CLASS1 IS ALPHABETIC and Not numeric
```

ACTIVITY: CALCULATOR (3-Common_Verbs.cbl)

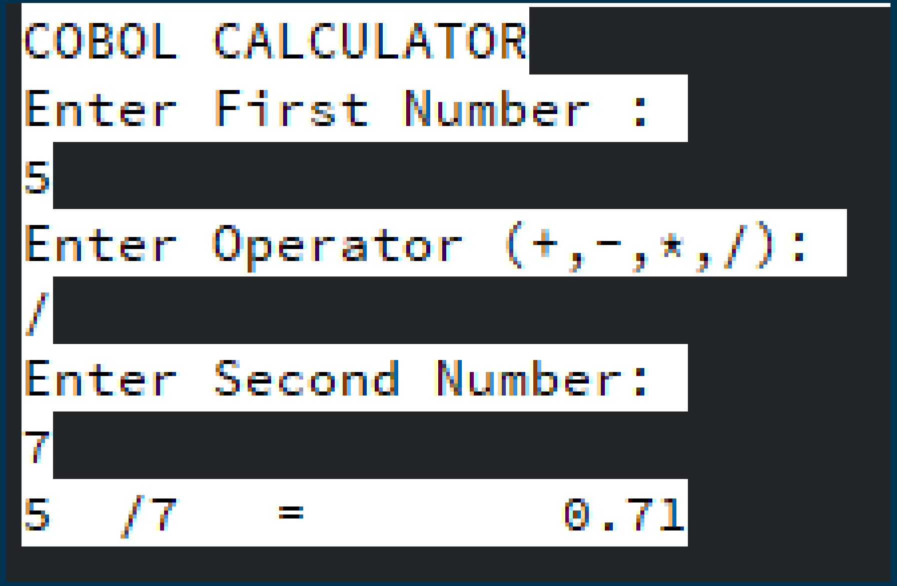
Make a Calculator Program using COBOL.

- Program that Accepts 2 INPUT and Operator
- Just Use the 4 Common Verbs (Multiply, Add, Divide and Add).
- Apply the Conditionals
- You can use the online compiler to run the program.
- Send the PDF file after clicking the Pretty Print

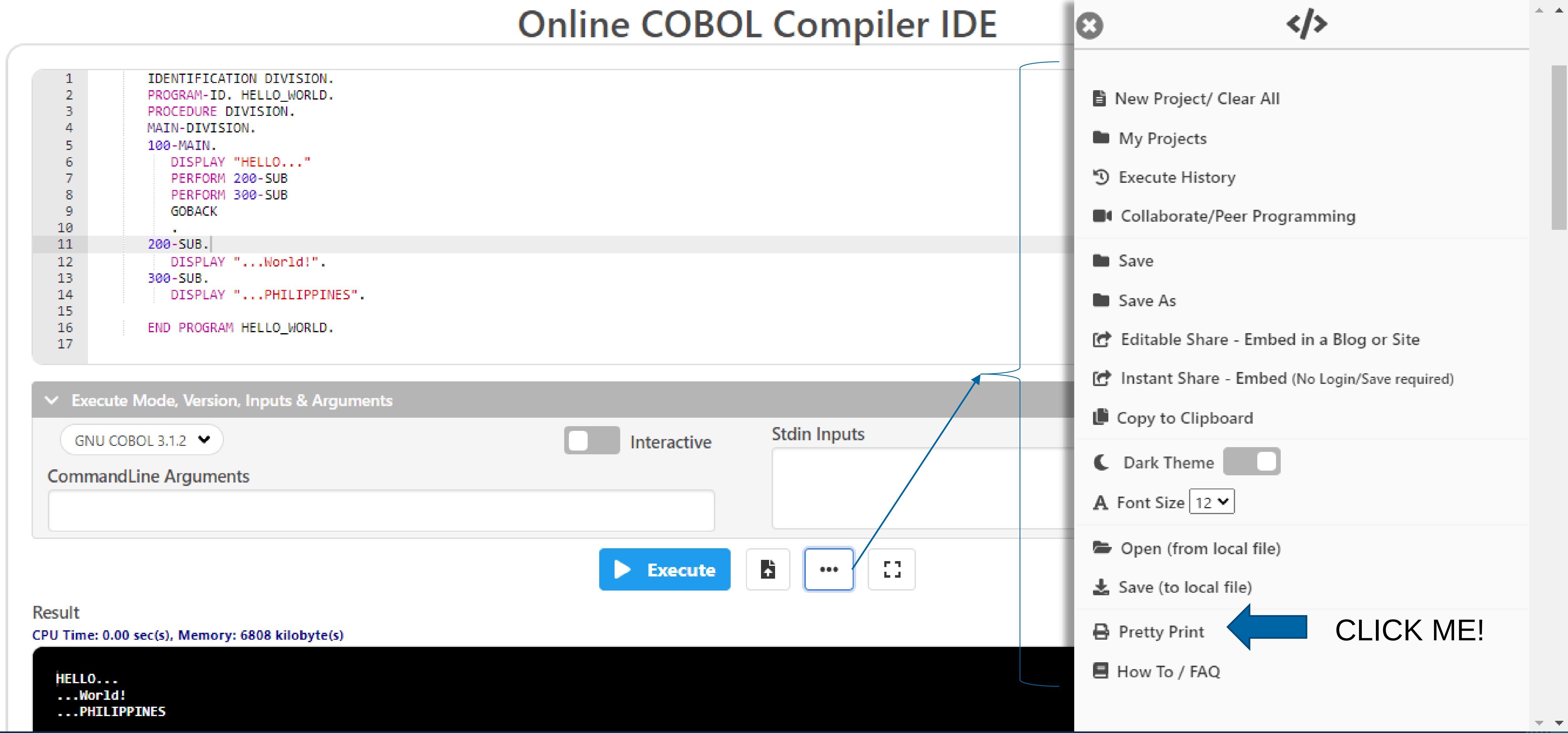
ACTIVITY: CALCULATOR (3-Common_Verbs.cbl)



```
COBOL CALCULATOR
Enter First Number :
4
Enter Operator (+,-,*,/):
/
Enter Second Number:
0
Cannot Be Divided to 0
```



```
COBOL CALCULATOR
Enter First Number :
5
Enter Operator (+,-,*,/):
/
Enter Second Number:
7
5  /7   =          0.71
```



```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO_WORLD.
PROCEDURE DIVISION.
MAIN-DIVISION.
100-MAIN.
   DISPLAY "HELLO..."
   PERFORM 200-SUB
   PERFORM 300-SUB
   GOBACK
   .
200-SUB.
   DISPLAY "...World!".
300-SUB.
   DISPLAY "...PHILIPPINES".

END PROGRAM HELLO_WORLD.
```

GNU COBOL 3.1.2

Interactive

Font Size 12

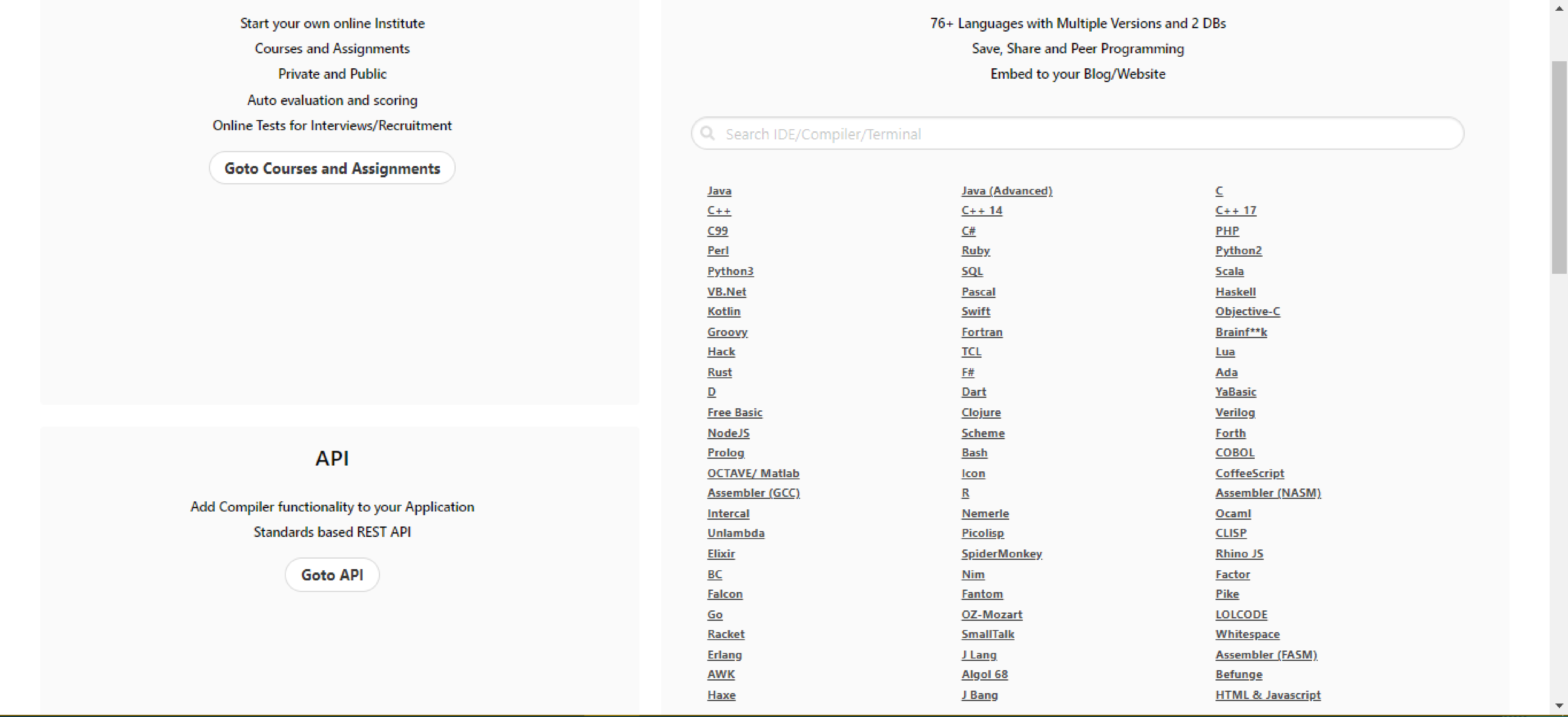
Result

CPU Time: 0.00 sec(s), Memory: 6808 kilobyte(s)

```
HELLO...
...World!
...PHILIPPINES
```

CLICK ME!

LINK: Online COBOL Compiler - Online COBOL Editor - Run COBOL Online - Online COBOL Runner (jdoodle.com)



Start your own online Institute

Courses and Assignments

Private and Public

Auto evaluation and scoring

Online Tests for Interviews/Recruitment

Goto Courses and Assignments

API

Add Compiler functionality to your Application

Standards based REST API

Goto API

76+ Languages with Multiple Versions and 2 DBs

Save, Share and Peer Programming

Embed to your Blog/Website

Search IDE/Compiler/Terminal

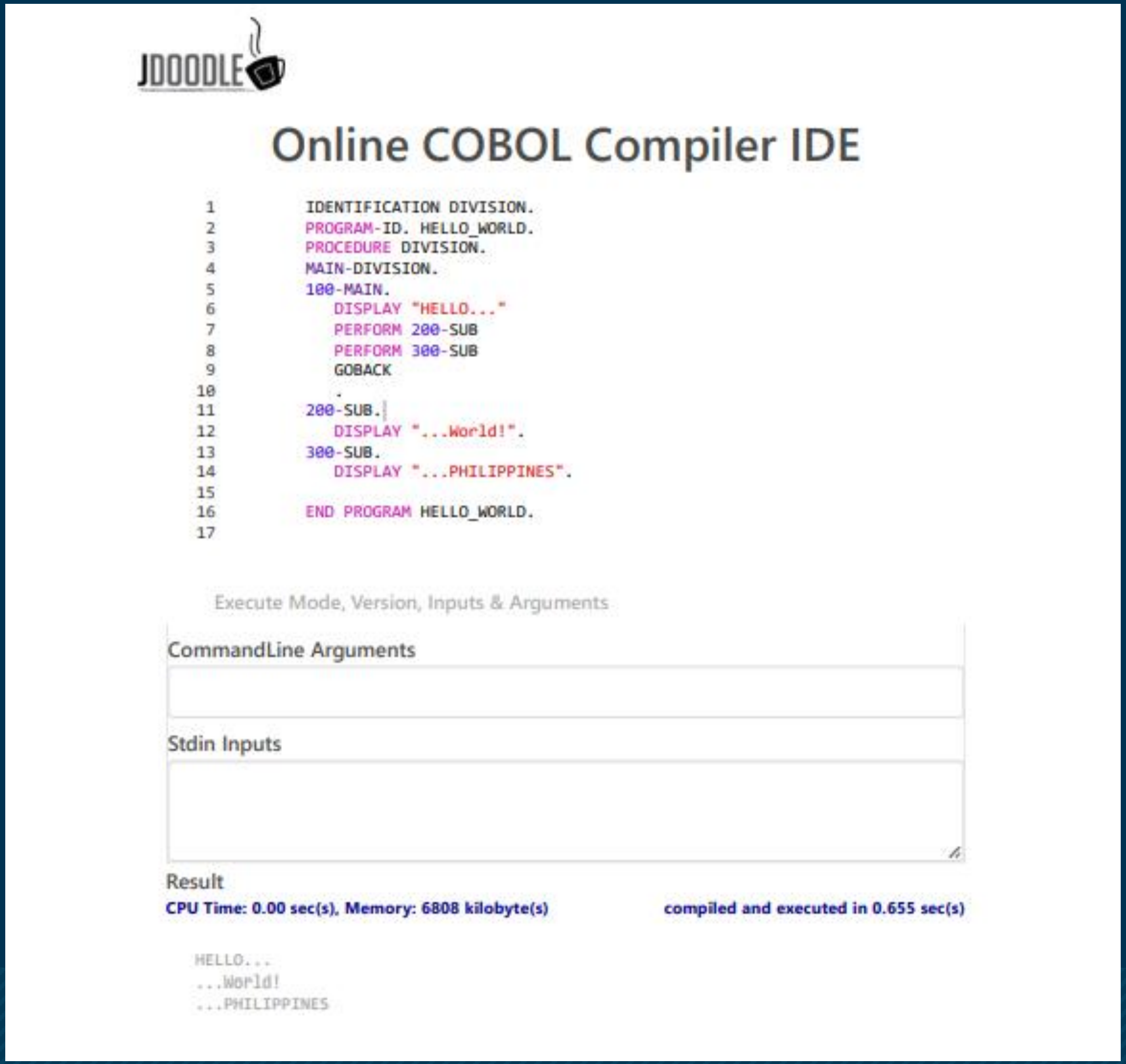
- Java
- C++
- C99
- Perl
- Python3
- VB.Net
- Kotlin
- Groovy
- Hack
- Rust
- D
- Free Basic
- NodeJS
- Prolog
- OCTAVE/ Matlab
- Assembler (GCC)
- Intercal
- Unlambda
- Elixir
- BC
- Falcon
- Go
- Racket
- Erlang
- AWK
- Haxe

- Java (Advanced)
- C++ 14
- C#
- Ruby
- SQL
- Pascal
- Swift
- Fortran
- TCL
- F#
- Dart
- Clojure
- Scheme
- Bash
- Icon
- R
- Nemerle
- Picolisp
- SpiderMonkey
- Nim
- Fantom
- OZ-Mozart
- SmallTalk
- J Lang
- Algol 68
- J Bang

- C
- C++ 17
- PHP
- Python2
- Scala
- Haskell
- Objective-C
- Brainf**k
- Lua
- Ada
- YaBasic
- Verilog
- Forth
- COBOL
- CoffeeScript
- Assembler (NASM)
- Ocaml
- CLISP
- Rhino JS
- Factor
- Pike
- LOLCODE
- Whitespace
- Assembler (FASM)
- Befunge
- HTML & Javascript

LINK: Online Compiler and Editor/IDE for Java, C/C++, PHP, Python, Perl, etc (jdoodle.com)

SAMPLE FORMAT OF FILE SUBMISSION



Online COBOL Compiler IDE

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO_WORLD.
PROCEDURE DIVISION.
MAIN-DIVISION.
100-MAIN.
   DISPLAY "HELLO..."
   PERFORM 200-SUB
   PERFORM 300-SUB
   GOBACK
   .
200-SUB.
   DISPLAY "...World!".
300-SUB.
   DISPLAY "...PHILIPPINES".

END PROGRAM HELLO_WORLD.

```

CommandLine Arguments

Stdin Inputs

Result

CPU Time: 0.00 sec(s), Memory: 6808 kilobyte(s)

compiled and executed in 0.655 sec(s)

```
HELLO...
...World!
...PHILIPPINES
```

FINAL NOTE:

For Questions and Clarifications:
Welcome to COBOL-Programming-Languages-Code-Summary Discussions! · Discussion #1 · cristoph143/COBOL-Programming-Languages-Code-Summary (github.com)
For IDE:
OpenCobolIDE project files : OpenCobolIDE (launchpad.net)

For Online Compiler:

Online COBOL Compiler - Online COBOL Editor - Run COBOL Online - Online COBOL Runner (jdoodle.com)

For Format:

https://github.com/cristoph143/COBOL-Programming-Languages-Code-Summary.git

REFERENCES

- COBOL - Program Structure (tutorialspoint.com)
- History of COBOL – Joysis Tech Voc Inc (joysistvi.edu.ph)
- https://qph.fs.quoracdn.net/main-qimg-06fb1e469419f1501f8fd08ea8a2b18b-c
- https://deidreadams.com/wp-content/uploads/2014/01/Code002.jpg
- The USING phrase - IBM Documentation
- COBOL - Conditional Statements (tutorialspoint.com)

Thank You