COBOL 101: Writing your COBOL Program

COBOL, which stands for Common Business Oriented Language, is a programming language that is often associated with mainframes. It was created in the 1950s, and was meant to serve many business, financial, and even governmental systems. It was designed to be easily readable by humans, including those who work in business or governmental roles with no prior programming experience or previous computer science knowledge. And while COBOL may be quite different from other programming languages that exist, because of how it was designed to be easily readable, even without prior coding experience, it is not a super difficult language to pick up and learn, especially if you have prior coding experience. There are many important sectors of our modern world that not only use COBOL but are reliant upon it, such as banking systems, governmental agencies (social security, as a specific example), insurance systems, and healthcare systems. It is a very common misconception that COBOL is a dying language that is outdated and will be replaced with newer and more modern languages. One fact that could change one's mind about this misconception is simple: There are over 800 billion lines of COBOL code that are run daily, and have gotten the job done for the companies that use them for over 60 years in an effective and efficient way. This lesson will serve as an introduction to COBOL, to give an understanding of what COBOL code can do and what it may look like, and to provide an understanding of its usefulness to someone with no prior knowledge of what COBOL is. Similar to the JCL lesson, this will not be super in-depth or technical, and serves to be an introduction to COBOL. If you are interested in taking a deeper dive into COBOL, take CSCI 465! It

will provide an opportunity to take a deeper dive into the technical aspect of COBOL, as well as allow you to learn how to code your own COBOL.

What is the structure of a COBOL program?

Firstly, COBOL code needs to be organized in specific columns, depending on the type of information on each line.

- Lines 1 through 6 should always remain empty (at least for modern coding purposes)
- Line 7 should be used for indicators (such as comments or continuation lines)
- Lines 8 through 11 are what is known as “Section A” of a COBOL program, which belongs to names such as the division names, and 01 level items (these will be covered more later)
- Lines 12-72 are what is known as “Section B”, which contains COBOL statements (program logic), value clauses, and essentially anything else not in Section A.
- Any lines past 72 should remain blank, as the compiler often ignores them

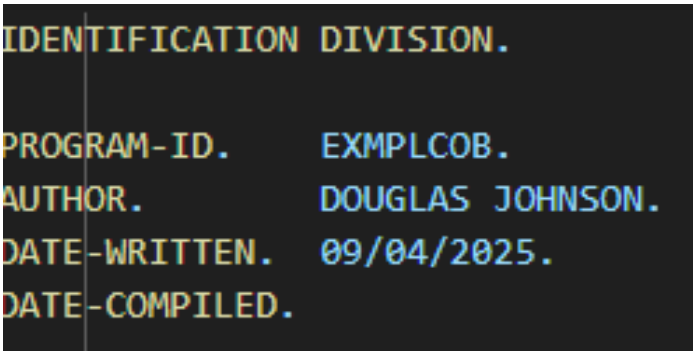
There are 4 divisions of a COBOL program that all play their own role within the program. These 4 divisions are: the **identification** division, the **environment** division, the **data** division, and the **procedure** division. Also, within each division, there are some sub-divisions, called sections, that also play a role. These will be further discussed within the discussion of each specific division. These four divisions should always be written in the order defined here, meaning that every COBOL program should start with an identification division and end with a procedure division.

Quick note on terminology: Paragraphs

Whenever talking about COBOL code, you may hear the term “paragraph” used when referring to the code. All this term refers to is a named block of code. In other languages, such as C++, you may be more familiar with, this may be thought of as something like a function.

Identification division

The identification division is the starting point for a COBOL program, providing the compiler with basic information about the program. The exact information contained here is largely determined by the programmer (or their company, depending on what they want documented), but the only required field in this section is a “PROGRAM-ID”. There are other optional fields, such as “AUTHOR”, “DATE-WRITTEN”, or “DATE-COMPILED” are also commonly seen within the identification division, but they are not strictly required by the compiler, and rather serve for documentation purposes. An example of a COBOL identification division can be seen below:



In the image above, you can see that the identification division is pretty simple. As mentioned, the PROGRAM-ID is the only part of this that is required, so that the compiler knows how to refer to this program. The AUTHOR, DATE-WRITTEN, and DATE-COMPILED fields are all optional and are there to serve as documentation purposes. This is so that if someone else were to review your code, they could see who

did what and when it was done, in case that information is relevant to the project being worked on.

Environment division

Within the environment division, there are 2 sections that exist. These are the configuration section and the input-output section. The **configuration section** exists as a means to define system-specific settings, such as the specific type of computer (ie, IBM 370) a program was built for. This section is optional, and for the purpose of learning COBOL in modern times, it is not critical to dive deep into it.

The other section here is the **input-output section**, which is much more important for a beginner to understand its purpose. Within the input-output section, there is 1 mandatory paragraph to include, as well as 1 optional paragraph to include. I will only touch on the mandatory paragraph here, which is the FILE-CONTROL paragraph. This paragraph is used to define the relationship between file names used within the program's logic, to external files, either being used as input or output (ie, the DD statements listed within the JCL). This is done using what are known as "SELECT … ASSIGN" statements. An example of what a "SELECT … ASSIGN" statement is shown below:

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.

    SELECT EXAMPLE-FILE ASSIGN TO EXAMFILE.
    SELECT OUTPUT-FILE  ASSIGN TO OUTFILE.
```

In the image above, you can see what an example Environment Division may look like. It starts with the keyword ENVIRONMENT DIVISION, followed by the label

INPUT-OUTPUT SECTION, which is then followed by the label for the paragraph for FILE-CONTROL, where you can see the use of the “SELECT … ASSIGN” statements. The way the “SELECT … ASSIGN” statements work is simple.

- The first file mentioned, which is EXAMPLE-FILE and OUTPUT-FILE, is are file name that will be used within the COBOL program’s logic to refer to that file.
- The second names listed, which are EXAMFILE and OUTFILE, are used to refer to the DD statements in the JCL that use those names. So, for the example above, here's what JCL for it may look like:

```
//JSTEP01  EXEC PGM=TEMPNAM0,COND=(0,LT)
//*
//STEPLIB  DD   DSN=Z63081.PERSON.LOADLIB,DISP=SHR
//*
//EXAMFILE DD   DSN=Z63081.PERSONAL.PROJECTS(EXAMDATA),DISP=SHR
//*
//OUTFILE  DD   SYSOUT=*
//*
//SYSUDUMP DD   SYSOUT=*
```

In the image above, the names EXAMFILE and OUTFILE are used as names of DD statements used in the JCL. These names must match the ones used after the ASSIGN clause in the FILE-CONTROL paragraph. This is how the operating system links your COBOL program to the correct dataset.

So, for example, when EXAMFILE is assigned to EXAMPLE-FILE in my COBOL program, when I refer to the name EXAMPLE-FILE in my program logic, It will use the data set listed at the DSN associated with the EXAMFILE DD statement, which here is Z63081.PERSONAL.PROJECTS(EXAMDATA).

And for the OUTFILE DD, instead of referencing a dataset, I use SYSOUT=*. This tells the operating system to output to standard output, rather than writing it to a file, since that file is strictly used for output.

Picture Clauses

Before getting into the data division, a very important concept in COBOL programming, known as picture clauses or PIC clauses, must be discussed. PIC clauses are used to define the type, size, and format of a data item, and tell this information to the compiler so it knows how to handle it. The two types of PIC clauses to know as a beginner are PIC X clauses and PIC 9 clauses. PIC X clauses are used to refer to alphabetical data, where PIC 9 clauses are used to refer to numerical data. For example, PIC X(25) will be seen in my following example, and all this means is that the variable associated with it is character data, and has 25 characters. Also, for something like PIC 999, this variable associated with it is a 3-digit number.

Note: With numerical data, you may notice an S in between the PIC and the 9's; this simply means it's a signed number. Also, you may notice a V in between 9's. This is used to signify an implied decimal place to the compiler.

Data division

The data division is used to define all data items a program will use. This includes, for purposes of internal processing, such as variables, and input/output operations, such as defining a detail line or a header. 3 sections make up data divisions, and these are the FILE SECTION, the WORKING-STORAGE SECTION, and the LINKAGE SECTION; however, here I will mainly focus on the file section and the working-storage section.

File section

The file section is used to describe the structure of input and output files in a COBOL program. Each file that follows the SELECT clause in the environment division should be defined here. This allows you to refer to specific parts of an input record using variable names. This is why knowing the data that you are processing is significant to becoming a COBOL programmer. Take, for example, the file section of my example COBOL program:

```
DATA DIVISION.
FILE SECTION.

FD  EXAMPLE-FILE
    RECORDING MODE F.

01  FINANCIAL-RECORD.
    05  IN-NAME          PIC X(25).
    05  IN-DEFAULT-AMT   PIC S9(5)V99.
    05  IN-DEPT-AMT      PIC S999V99.
    05  IN-WITH-AMT      PIC S999V99.
    05  FILLER           PIC X(38).
```

```
Jonathan A. Smith         00007500002500010
Elizabeth K. Johnson      00050000010000050
Michael B. Jordan III     00123450020000150
Sophia Martinez           00025000005000075
Christopher L. Anderson   00100000030000100
Isabella O'Neill          00001200001000005
Alexander von Humboldt    00200000050000250
Olivia Williams           00012500015000075
```

In the image above, you can see my EXAMPLE-FILE definition and the associated record layout FINANCIAL-RECORD, as well as some of the data associated with it. The purpose of the FINANCIAL-RECORD is to break down each input record byte-by-byte and assign its contents to variables that can be referred to in my program.

For example:

- The first variable is IN-NAME, which has a PIC X(25) clause associated with it, which means the first 25 bytes of the record are designated to be character data, which will be assigned to this variable, IN-NAME, whenever a record is read.
- The next variable, IN-DEFAULT-AMT, is defined with PIC S9(5)V99, meaning the next 7 bytes will be associated with the IN-DEFAULT-AMT.

While it may be hard to visually interpret the numeric data when it's all packed together as shown above, having a well-defined layout of the data (ie, knowing which data is at which bytes) makes it easy to manage. As long as you know the layout of the data, and how each record is formatted, it's easy to accurately reference your data without needing unnecessary blank spaces. This also helps save memory space, which is especially useful when working with larger datasets.

Working-storage section

This section is used to declare all program-level variables that will be used within a COBOL program's procedure division (where the program logic is). This includes variables such as:

- Variables for printing (detail lines, header lines)
- Variables used for computations
- Counters and flags

  Some of my working-storage section can be seen below:

```
01  DETAIL-LINE.
    05  OUT-NAME          PIC X(25).
    05                    PIC X(8) VALUE SPACES.
    05  OUT-DEFAULT-AMT   PIC $$$,$$9.99.
    05                    PIC X(12) VALUE SPACES.
    05  OUT-DEPT-AMT      PIC $$9.99.
    05                    PIC X(15) VALUE SPACES.
    05  OUT-WITH-AMT      PIC $$9.99.
    05                    PIC X(5) VALUE SPACES.
    05  OUT-TOTAL-AMT     PIC $$$$,$$9.99.
    05                    PIC X(34) VALUE SPACES.

01  PACKED-FIELDS.
    05  DEFAULT-AMT       PIC S9(5)V99 PACKED-DECIMAL VALUE 0.
    05  DEPT-AMT          PIC S999V99  PACKED-DECIMAL VALUE 0.
    05  WITH-AMT          PIC S999V99  PACKED-DECIMAL VALUE 0.
    05  TOTAL-AMT         PIC S9(6)V99 PACKED-DECIMAL VALUE 0.
```

In the image above, you can see how I defined my detail line (print line), as well as my packed fields, which are the fields that will be used for arithmetic in my program. For the detail line, I break my DETAIL-LINE variable down into smaller parts, with each part defining what should be printed, as well as how many characters that takes up from the print line. For example:

- The first variable on the detail line is OUT-NAME with a PIC X(25) clause. This means the first thing printed should be the variable OUT-NAME, and it will consist of 25 characters.
- I follow with an empty PIC X(8) clause. Why? This serves as spacing between the data, so it is formatted in an easy way to comprehend
- I then follow with OUT-DEFAULT-AMOUNT, which is defined with a PIC $$$,$$9.99 clause. This is how you format numeric data for display, with here representing that the number represents currency. This is just a fancy way of suppressing leading zeros, as if it were 999,999.99, it would show all digits, even if they were zero. This is done this way so a number like 708.98 is displayed as $708.98, rather than $00,708.98.

Once values are moved into each of the variables listed on the detail line, and it is printed, it will be formatted exactly as defined by the programmer, down to each byte. Each print line must add up to 132 bytes, even if you need to fill bytes with blank spaces. This precise amount of bytes ensures that the output will be aligned correctly when it is printed to a report.

For the PACKED-FLDS, I am initializing them to hold numeric data of various lengths, as defined by the number of 9's in the PIC 9 clauses, and then I am initializing their values to 0.

Procedure division

The procedure division is where the program logic and instructions are written, in the form of paragraphs. The simplest way to think about it is that the identification division, environment division, and data division are all set up for the procedure division. This is where you tell the compiler step-by-step instructions of what you want your program to do and how you want it to manipulate the data that is inputted. These are executable statements that tell the compiler how to process data defined in the data division. Some common things done within the procedure division include:

- Reading from files / writing to files
- Performing computations
- Looping through all the records of a dataset
- Moving data to its proper field
- Calling other programs or subroutines to perform other tasks

Below is a snippet of some of my procedure division for my program:

```
PERFORM UNTIL EOF-FLAG = 'Y'

    MOVE IN-NAME TO OUT-NAME
    MOVE IN-DEFAULT-AMT TO DEFAULT-AMT
    MOVE IN-DEPT-AMT TO DEPT-AMT
    MOVE IN-WITH-AMT TO WITH-AMT

    COMPUTE TOTAL-AMT = DEFAULT-AMT + DEPT-AMT - WITH-AMT
    MOVE DEFAULT-AMT TO OUT-DEFAULT-AMT
    MOVE DEPT-AMT TO OUT-DEPT-AMT
    MOVE WITH-AMT TO OUT-WITH-AMT
    MOVE TOTAL-AMT TO OUT-TOTAL-AMT

    WRITE REPORT-RECORD FROM DETAIL-LINE AFTER 2
    READ EXAMPLE-FILE AT END MOVE 'Y' TO EOF-FLAG
    END-READ

END-PERFORM.
```

As seen in the image above, this is what looping through a file may look like. Since the input file was defined in the file section earlier, the IN- variables already hold the values of a specific record once read in. These are then moved to their proper fields:

- IN-NAME is moved straight to OUT-NAME, because it is character data with no manipulation needing to be done
- All the other input variables are moved to packed fields defined in the working-storage section, so they can properly execute the computations necessary
- Once the computation is completed, all packed variables are moved to their proper output fields, and then they are written to REPORT-RECORD, in the format of the DETAIL-LINE
- **Note:** REPORT-RECORD was defined in the file section as well, but it is simply a 132-byte field that is not broken down at all, unlike the FINANCIAL-RECORD

shown earlier. This is because, as the programmer, you are breaking down how you want it outputted through the use of the DETAIL-LINE, whereas for the input FINANCIAL-RECORD, that structure is already defined for you.

- Once the detail line is printed, it checks if at the end of the file. If so, it will terminate the loop. If not, it will read the next record. This process will continue until the end of the file is reached.

As seen from the syntax, it is easy to see why some may call COBOL an easy-to-read language, even for those with no technical programming experience. Its use of mostly English makes it understandable, and one of the main reasons a language like COBOL rose to prominence in the first place, and remains a significantly relevant language in many legacy systems today.

The Results

The entirety of this example COBOL program, along with the JCL and data to properly execute it, is provided on our GitHub https://github.com/niu-acm/SIGmainframe

But to see briefly what happens when the JCL to execute this program is run, here is what is written to standard output following the job submission:

NAME	INITIAL AMOUNT	DEPOSIT AMOUNT	WITHDRAWAL AMOUNT	FINAL AMOUNT
0Jonathan A. Smith	$7.50	$0.25	$0.10	$7.65
0Elizabeth K. Johnson	$50.00	$1.00	$0.50	$50.50
0Michael B. Jordan III	$123.45	$2.00	$1.50	$123.95
0Sophia Martinez	$25.00	$0.50	$0.75	$24.75
0Christopher L. Anderson	$100.00	$3.00	$1.00	$102.00
0Isabella O'Neill	$1.20	$0.10	$0.05	$1.25
0Alexander von Humboldt	$200.00	$5.00	$2.50	$202.50
0Olivia Williams	$12.50	$1.50	$0.75	$13.25
0Benjamin T. Franklin	$1,000.00	$10.00	$5.00	$1,005.00
0Emma Watson	$30.00	$2.00	$1.00	$31.00
0Jacob de la Cruz	$9.00	$0.50	$0.20	$9.30
0Mia Rodriguez	$55.00	$1.20	$0.60	$55.60
0David Garcia	$20.00	$0.75	$0.25	$20.50
0Ava Thompson	$40.00	$2.00	$1.00	$41.00
0William B. Harrison	$150.00	$4.00	$2.00	$152.00
0Charlotte Brown	$65.00	$1.50	$0.75	$65.75
0Henry-Louis Eschbaum	$80.00	$3.00	$1.50	$81.50
0Amelia Earhart	$205.00	$5.00	$2.50	$207.50
0Theodore Roosevelt	$500.00	$20.00	$10.00	$510.00
0Evelyn Clark	$8.00	$0.25	$0.15	$8.10
0Christopher Columbus	$300.00	$6.00	$3.00	$303.00
0Penelope Cruz	$11.00	$0.75	$0.45	$11.30
0Sebastian Vettel	$99.00	$1.00	$0.50	$99.50
0Victoria Beckham	$77.00	$1.20	$0.60	$77.60
0Maximilian von Bayern	$250.00	$7.00	$3.50	$253.50

As seen in the image above, suddenly what was seemingly a bunch of jumbled together records of names and numbers has been processed and outputted efficiently, in a way that is easy to read and understand for anyone.

Note: The entire print line actually consists of 133 bytes. However, the first byte will always be taken up by a line separator character. That is why in the example output above, you will notice 0's before all the names. 0 is the line separator character used for double spacing. Meaning that on an actual job report, those lines will be double-spaced when outputted.

Resources

https://www.ibm.com/docs/en/i/7.4.0?topic=cobol-ile-language-reference