**Brush up your COBOL: Why is a 60 year old language suddenly in demand?**

The suddenly strained unemployment systems often run on a 60-year-old programming language, COBOL. So, how can you learn it, make big bucks, and save lots of state agencies that need new code to deal with all the new government stimulus programs?

Avatar for 

[Charles R. Martin](https://stackoverflow.blog/author/charles-r-martin/)

Once upon a time, when the world and computers were new, I was in an Associate’s Degree program for Data Processing—there were no “computer science” programs then—in which I had to study accounting, math, statistics, and three computer languages: IBM/360 Basic Assembly Language, FORTRAN, and COBOL. By the 80’s, students were being told that COBOL was a dead language, and no one was studying it any more.

Now, in 2020, [governments and banks](https://www.cnn.com/2020/04/08/business/coronavirus-cobol-programmers-new-jersey-trnd/index.html) are [pleading for COBOL](http://www.kake.com/story/41983960/unemployment-woes-continue-as-system-remains-overloaded) programmers, the [language that wouldn’t die](https://thenewstack.io/cobol-everywhere-will-maintain/).

Governor Laura Kelly of Kansas [said](http://www.kake.com/story/41983960/unemployment-woes-continue-as-system-remains-overloaded):

*“So many of our Departments of Labor across the country are still on the COBOL system. You know very, very old technology,” Kelly said Tuesday. “Our Department of Labor had recognized that that was an issue and had initiated modernization, and, unfortunately, that’s something that takes time. This (virus) interfered and they had to cease the transition to a much more robust system. So they’re operating on really old stuff.”*

New Jersey Governor Phil Murphy [made a television appearance](https://www.youtube.com/watch?v=HSVgHlSTPYQ) to plead for COBOL programmers to help.

So, how can you learn COBOL, make big bucks, and save lots of state agencies that need new code to deal with all the new government stimulus programs?

Let’s find out.

**COBOL? What’s this COBOL?**

COBOL stands for COmmon Business Oriented Language. One of the first of the high-level languages, it was put together by a group sponsored by the Department of Defense to develop a common business language. That group came to be called CODASYL—the Committee on Data Systems Languages—and defined a “common business oriented language,” drawing from Grace Hopper’s FLOW-MATIC, and other languages including Univac’s AIMACO and IBM’s COMTRAN. The resulting language went through more revisions, but rapidly became the dominant language for building business systems, and it has remained dominant since.

Plenty of companies still use COBOL, including IBM, UPS, and Cigna. Mario Ceballos, a software engineer at Cigna, told me, “The syntax is kept simple to allow **non-programmers** (“The Business”) to read it and understand it. COBOL is meant to be explicit, because there shouldn’t be room for assumptions.”

Of course, it has had its critics. In 1975, Edsgar Dijkstra [famously proclaimed](https://www.goodreads.com/quotes/496316-the-use-of-cobol-cripples-the-mind-its-teaching-should) that “The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offence[sic].” This undoubtedly led to the decline of teaching COBOL in universities, but it remained the dominant business language.

But finding people with COBOL skills can be tough. “The mainframe is a very difficult platform to learn, and that’s due to the cost,” said Ceballos. “Individuals do not have the money to pay to lease a mainframe. A very small amount of schools teach courses on mainframes and COBOL. When IBM started remote work and outsourcing, they stopped incentivizing American schools to teach courses in Mainframes and COBOL. The talent pool shifted from on-shore to off-shore.  Any local talent will be expensive with their consulting fees.”

**Why is COBOL still dominant?**

Compared to common programming languages today, COBOL is different, and in some ways very limited: you can’t do dynamic memory allocation, you can’t easily access low-level features of the operating system or particular computer architecture. The most common forms of the language can’t use recursion. You’d never want to write a compiler in COBOL. A computer science student presented with COBOL would be appalled.

This is a category error. In modern terminology, COBOL is actually a [domain-specific language](https://stackoverflow.com/questions/809574/what-is-a-domain-specific-language-anybody-using-it-and-in-what-way), specific to the particular domain of business programming. Robert Glass identified specific ways in which COBOL is better suited to business programming than general-purposes languages, among them:

* A business-oriented language needs to declare, manage, and manipulate heterogenous data. Business programs mix fixed and variable length strings, floating-point, integer, and decimal data with wild abandon in complicated record structures, often with variable parts. Database programmers are familiar with some of these issues, and object-relational mapping tools trip over these complexities regularly.
* Business and financial data needs to be managed using true decimal data types. Accounting systems must be correct to the last decimal digit and need to reproduce exactly the results of hand-calculation; conventional floating-point numbers lead to complexities and errors.
* A business-oriented language needs to access and manipulate large amounts of record-structured data maintained externally.

Now, none of this is beyond the capabilities of general-purpose programming languages, of course. But in COBOL, it’s native to the language.

We can debate the need for COBOL, but the fact is that hundreds of billions of lines of COBOL exist, and attempts to migrate away from COBOL have not generally been successful.

**Your first COBOL program**

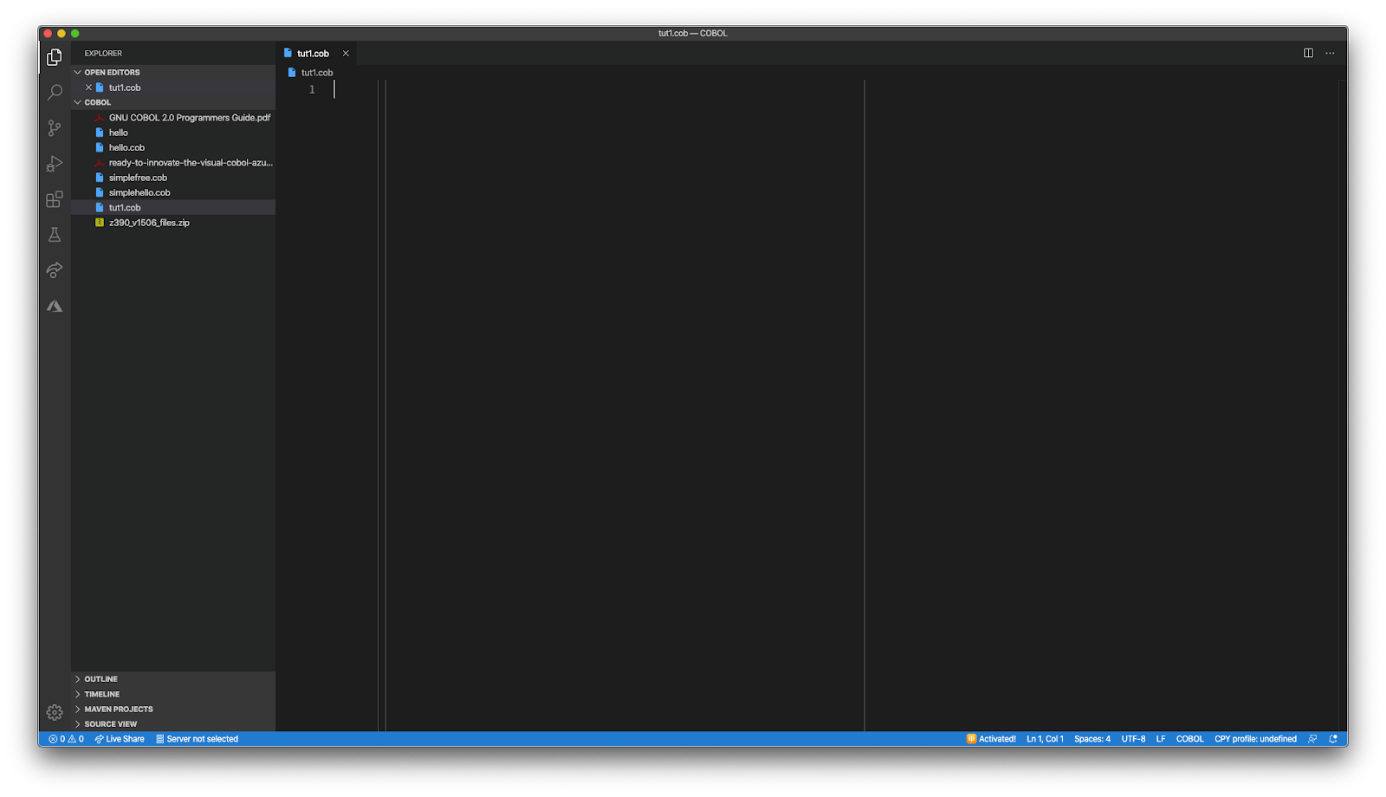
The source files are simple text files. Having a useful programming editor with language support is as convenient for COBOL as any other language, if not more so. The easiest thing for a beginner is to use [Visual Studio Code](https://code.visualstudio.com/), the only competitor for my affections since EMACS.

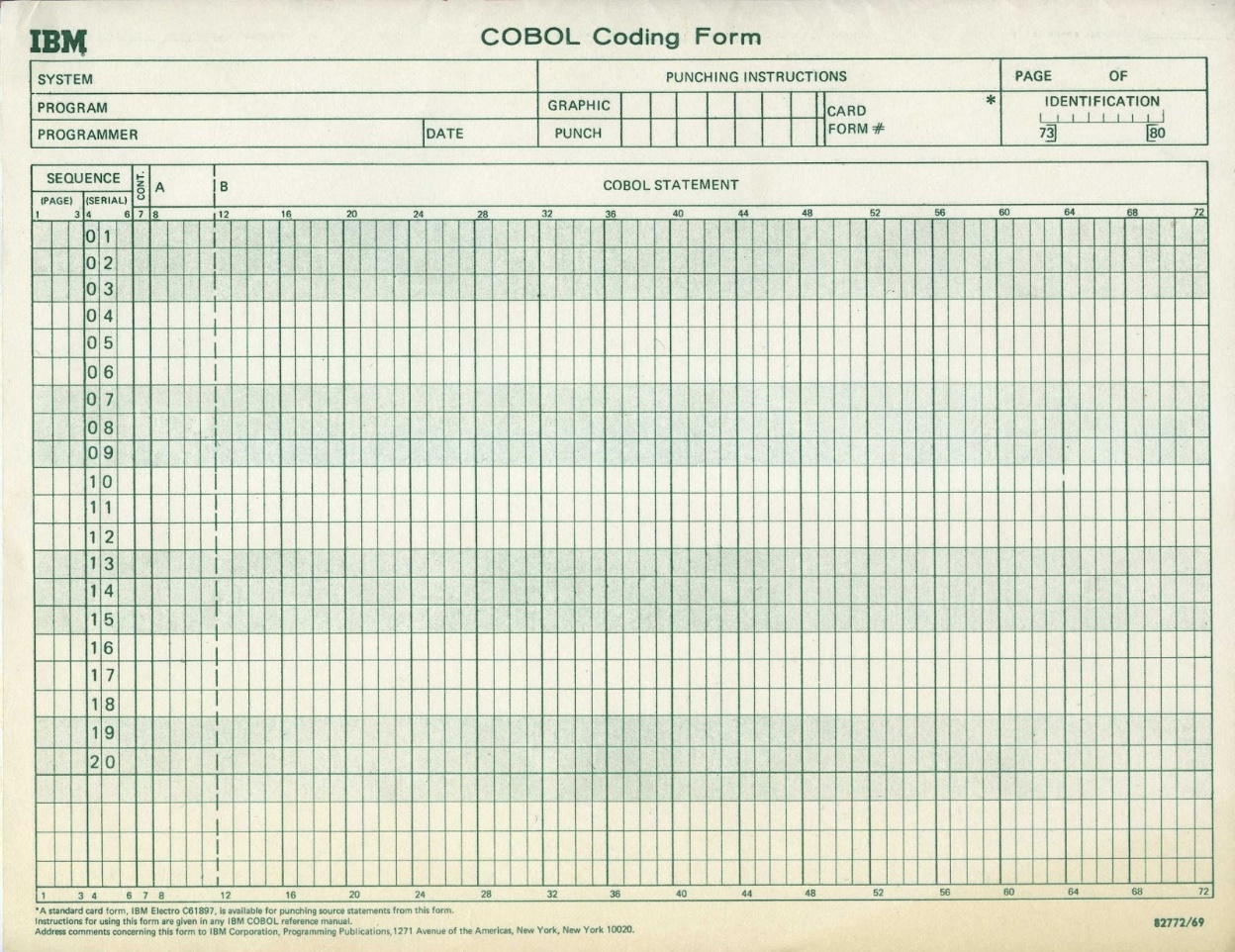
There are surprisingly many VSCode extensions for COBOL. Right now, I’m using the [bitlang](https://marketplace.visualstudio.com/items?itemName=bitlang.cobol) code colorizer and [Broadcom COBOL language support](https://marketplace.visualstudio.com/items?itemName=broadcomMFD.cobol-language-support). A lot of the others are intended for people programming in a mainframe environment, but that adds complexity we don’t need for an introduction.

So, to summarize, to begin to experiment with COBOL:

1. Download and install Visual Studio Code if you haven’t already.
2. Install the [bitlang.cobol](https://marketplace.visualstudio.com/items?itemName=bitlang.cobol) and [Broadcom COBOL Language Support](https://marketplace.visualstudio.com/items?itemName=broadcomMFD.cobol-language-support) extensions.
3. Install [GnuCOBOL](https://open-cobol.sourceforge.io/faq/index.html" \l "using-gnucobol). (Honestly, if anything is going to cause trouble, it will be this. The Homebrew installation on MacOS worked fine, and I don’t have other systems with which to test. On Windows, MicroFocus has a free trial for Visual Studio COBOL and Azure support for experimentation.)

There you are, you’ve installed everything and you’re ready to write your first COBOL program. As is traditional, we’ll start with the Ur-program, “Hello, world”.

So here’s your first surprise as a new COBOL programmer: COBOL cares about what column your code is in. In a traditional COBOL program, the source has several components:



Columns 1-6 are there for a sequence number. Column 7 is called the indicator area; in general, it’s mostly used to indicate comments by putting an asterisk ‘\*’ in that column. Code then goes in columns 8 through 72, and columns 73-80 are basically free for the programmers use.

This is all based around the days when we put our source into 80-column Hollerith cards.

Modern COBOL compilers also accept a free format which doesn’t force your code into the 80-column straitjacket, but a very large proportion of existing code is still in the card-image format. For right now, we’ll stick with card images.

Brace yourselves: COBOL is not a block-structured language like nearly any other language you’ve ever used. A major design goal for COBOL from the first was that it should be “self-documenting” with an English-like syntax. Instead of having functions or subroutines and blocks, we have **divisions**, **sections**, **paragraphs**, and **statements**. (We’ll see something almost like a subroutine with the PERFORM verb below.)

Oh, right, we also have verbs for COBOL operators.

Here’s “Hello, World” in COBOL:

IDENTIFICATION DIVISION.

PROGRAM-ID. HELLO.

PROCEDURE DIVISION.

DISPLAY "Hello, world".

END PROGRAM HELLO.

Compared to some languages it’s a little wordy, but honestly not so bad. Compare it to a simple Java version:

public class Hello {

public static void main(String[] args){

System.out.println("Hello, world!");

}

}

Like all “Hello, world” programs it doesn’t do much—but if you’ve been told that it takes 90 lines to write a basic program in COBOL, well, you’ve been misled.

Now let’s take the “Hello world” program apart for our first example.

The first line is:

IDENTIFICATION DIVISION.

COBOL programs always have at least an identification division and a procedure division. The identification division has one important paragraph, the PROGRAM-ID. You need to give the program a name here. The name doesn’t need to correspond to the file name or pretty much anything, except when your COBOL program is being called from another COBOL program. This is through the CALL verb, which we’re not going to cover.

We do need to have a program ID, so we add

IDENTIFICATION DIVISION.

PROGRAM-ID. HELLO.

There are a lot of other things that commonly go into the identification division. I’ll add a couple of common examples.

IDENTIFICATION DIVISION.

PROGRAM-ID. HELLO.

AUTHOR. CHARLES R MARTIN.

DATE-WRITTEN. 2020-APR-11.

In modern environments, however, these are comments.

Speaking of modern environments, by the way, COBOL doesn’t require all-caps like I’ve been using. GnuCOBOL would be perfectly happy with

identification division.

program-id. tut2.

author. charlie martin.

procedure division.

display "hello, world".

end program tut2.

I’m just having a little misty-eyed nostalgia here.

Don’t judge me.So let’s finish up our “Hello, world.” The execution part of a COBOL program is in the **procedure division**.

IDENTIFICATION DIVISION.

PROGRAM-ID. HELLO.

PROCEDURE DIVISION.

DISPLAY "Hello, world".

END PROGRAM HELLO.

There’s one more bit of card-image format here. Notice that `DISPLAY “Hello, world”` is indented four columns. That’s because the part from column 8-72 actually has two parts: the A section, from column 8-11, and the B section from column 12 on. Divisions, sections, and paragraphs need to start in the A section; code statements should start in the B section.

**Extended COBOL Example**

Of course, “Hello, World” doesn’t really give you a good picture for any language, so let’s look a little deeper into COBOL with something that at least resembles a real business program. We’re going to use a pretty common example: computing a paycheck for hourly employees, including computing Federal, State, and FICA tax.

Having done it, I can tell you this is not an easy thing to do in reality—the tax tables are complex and arcane—so we’re going to simplify and make the Federal tax rate 16.4 percent, state 7 percent, and fix the FICA rate at 6.2 percent while carefully choosing our pay rate and hours worked to not hit the FICA cap. We’re only doing hourly workers, and we compute hours over 40 as overtime at 1.5 times the base rate.

No point in repeating the identification division. We start with a new division, the **environment division**, which exists to collect the interface between the COBOL program and the outside world.

ENVIRONMENT DIVISION.

INPUT-OUTPUT SECTION.

FILE-CONTROL.

SELECT TIMECARDS

ASSIGN TO "TIMECARDS.DAT"

ORGANIZATION IS LINE SEQUENTIAL.

Once again, we’re going to exercise some aspects of COBOL that will be surprising to people who haven’t worked in the record-oriented world of data-processing. In UNIX, Linux, MacOS, or Windows, a **record**is a line of text followed by some end of line character or characters. This causes a problem for traditional COBOL, but COBOL compilers implement a non-standard extension to handle this: ORGANIZATION IS LINE SEQUENTIAL.

The input-output section simply assigns a symbolic name (TIMECARDS) to the file and connects it to the file in the outside environment.

The next part of the program describes the data we’re working with. In COBOL, all data is generally presumed to be contained in fixed-format records. Those records have a hierarchical structure that’s indicated by the level numbers: 01 is the top level, and subdivisions get higher numbers. I used 02, 03, and so forth, but that’s arbitrary; we used to use 01, 05, and so on because it was easier to insert cards without repunching them all.

But now we introduce another division, the **data division**. As you probably guessed, this is for data. We’re using two **sections**. First is the**file section**.

DATA DIVISION.

FILE SECTION.

FD TIMECARDS.

01 TIMECARD.

02 EMPLOYEE-NAME.

03 EMP-FIRSTNAME PIC X(10).

03 EMP-SURNAME PIC X(15).

02 HOURS-WORKED PIC 99V9.

02 PAY-RATE PIC 99.

This is our input, which is fixed format; we’re connecting it to the TIMECARDS file with the FD line. Following that is the **working storage**section. It looks a little unfamiliar if you’re not used to COBOL, but really, I’m just declaring variables I’ll use in the program later.

WORKING-STORAGE SECTION.

\* temporary variables in computational usage.

\* intermediate values for computing paycheck with overtime

01 REGULAR-HOURS PIC 9(4)V99 USAGE COMP.

01 OVERTIME-HOURS PIC 9(4)V99 USAGE COMP.

01 OVERTIME-RATE PIC 9(4)V99 USAGE COMP.

01 REGULAR-PAY PIC 9(4)V99 USAGE COMP.

01 OVERTIME-PAY PIC 9(4)V99 USAGE COMP.

\* computed parts of the paycheck

01 GROSS-PAY PIC 9(4)V99 USAGE COMP.

01 FED-TAX PIC 9(4)V99 USAGE COMP.

01 STATE-TAX PIC 9(4)V99 USAGE COMP.

01 FICA-TAX PIC 9(4)V99 USAGE COMP.

01 NET-PAY PIC 9(4)V99 USAGE COMP.

The *unfamiliar*part of this is the PIC (or PICTURE) clause. COBOL is not strongly typed at all. Instead, more like C, every declaration is identifying a piece of memory; the PIC tells COBOL how to interpret that memory with a “picture”. In this case, 9(4)v99 tells COBOL that a chunk of memory named, for example REGULAR-HOURS is to be interpreted as a six-digit number that is assumed to have a decimal point (the V) in front of the last two digits. USAGE COMP tells COBOL to use an internal format that’s suited to fast arithmetic. What that format actually is is somewhat flexible and depends on the architecture, which means you’d best not depend on it being the same everywhere.

If you want to be confident of this, don’t use USAGE COMP, which leads to another part of the data, the format for a check to be output. These fields are the default usage, which is printable where USAGE COMP is not.

01 PAYCHECK.

02 PRT-EMPLOYEE-NAME PIC X(25).

02 FILLER PIC X.

02 PRT-HOURS-WORKED PIC 99.9.

02 FILLER PIC X.

02 PRT-PAY-RATE PIC 99.9.

02 PRT-GROSS-PAY PIC $,$$9.99.

02 PRT-FED-TAX PIC $,$$9.99.

02 PRT-STATE-TAX PIC $,$$9.99.

02 PRT-FICA-TAX PIC $,$$9.99.

02 FILLER PIC X(5).

02 PRT-NET-PAY PIC $\*,\*\*9.99.

The only really fun stuff here is that we have some new PIC formats: $,$$9.99 has a leading dollar sign that is always against the leftmost digit, and $\*,\*\*9.99 fills the space between the dollar sign and the first digits with \*.

I’ll show the entire program shortly, but I do want to point out the way COBOL does math, as in COMPUTE-GROSS-PAY:

COMPUTE-GROSS-PAY.

IF HOURS-WORKED > 40 THEN

MULTIPLY PAY-RATE BY 1.5 GIVING OVERTIME-RATE

MOVE 40 TO REGULAR-HOURS

SUBTRACT 40 FROM HOURS-WORKED GIVING OVERTIME-HOURS

MULTIPLY REGULAR-HOURS BY PAY-RATE GIVING REGULAR-PAY

MULTIPLY OVERTIME-HOURS BY OVERTIME-RATE

GIVING OVERTIME-PAY

ADD REGULAR-PAY TO OVERTIME-PAY GIVING GROSS-PAY

ELSE

MULTIPLY HOURS-WORKED BY PAY-RATE GIVING GROSS-PAY

END-IF

.

Yes, standard COBOL makes you spell it out.

So here’s the full program.

IDENTIFICATION DIVISION.

PROGRAM-ID. PAYCHECKS.

AUTHOR. CHARLES R. MARTIN.

DATE-WRITTEN. 2020-APR-15.

ENVIRONMENT DIVISION.

INPUT-OUTPUT SECTION.

FILE-CONTROL.

SELECT TIMECARDS

ASSIGN TO "TIMECARDS.DAT"

ORGANIZATION IS LINE SEQUENTIAL.

DATA DIVISION.

FILE SECTION.

FD TIMECARDS.

01 TIMECARD.

02 EMPLOYEE-NAME.

03 EMP-FIRSTNAME PIC X(10).

03 EMP-SURNAME PIC X(15).

02 HOURS-WORKED PIC 99V9.

02 PAY-RATE PIC 99.

WORKING-STORAGE SECTION.

\* temporary variables in computational usage.

\* intermediate values for computing paycheck with overtime

01 REGULAR-HOURS PIC 9(4)V99 USAGE COMP.

01 OVERTIME-HOURS PIC 9(4)V99 USAGE COMP.

01 OVERTIME-RATE PIC 9(4)V99 USAGE COMP.

01 REGULAR-PAY PIC 9(4)V99 USAGE COMP.

01 OVERTIME-PAY PIC 9(4)V99 USAGE COMP.

\* computed parts of the paycheck

01 GROSS-PAY PIC 9(4)V99 USAGE COMP.

01 FED-TAX PIC 9(4)V99 USAGE COMP.

01 STATE-TAX PIC 9(4)V99 USAGE COMP.

01 FICA-TAX PIC 9(4)V99 USAGE COMP.

01 NET-PAY PIC 9(4)V99 USAGE COMP.

\* print format of the check

01 PAYCHECK.

02 PRT-EMPLOYEE-NAME PIC X(25).

02 FILLER PIC X.

02 PRT-HOURS-WORKED PIC 99.9.

02 FILLER PIC X.

02 PRT-PAY-RATE PIC 99.9.

02 PRT-GROSS-PAY PIC $,$$9.99.

02 PRT-FED-TAX PIC $,$$9.99.

02 PRT-STATE-TAX PIC $,$$9.99.

02 PRT-FICA-TAX PIC $,$$9.99.

02 FILLER PIC X(5).

02 PRT-NET-PAY PIC $\*,\*\*9.99.

\* Tax rates -- 77 level aha！

77 Fed-tax-rate Pic V999 Value Is .164 .

77 State-tax-rate Pic V999 Value Is .070 .

77 Fica-tax-rate Pic V999 Value Is .062 .

\* 88 Level is for conditions.

01 END-FILE PIC X.

88 EOF VALUE "T".

PROCEDURE DIVISION.

BEGIN.

PERFORM INITIALIZE-PROGRAM.

PERFORM PROCESS-LINE WITH TEST BEFORE UNTIL EOF

PERFORM CLEAN-UP.

STOP RUN.

INITIALIZE-PROGRAM.

OPEN INPUT TIMECARDS.

PROCESS-LINE.

READ TIMECARDS INTO TIMECARD

AT END MOVE "T" TO END-FILE.

IF NOT EOF THEN

PERFORM COMPUTE-GROSS-PAY

PERFORM COMPUTE-FED-TAX

PERFORM COMPUTE-STATE-TAX

PERFORM COMPUTE-FICA

PERFORM COMPUTE-NET-PAY

PERFORM PRINT-CHECK

END-IF.

COMPUTE-GROSS-PAY.

IF HOURS-WORKED > 40 THEN

MULTIPLY PAY-RATE BY 1.5 GIVING OVERTIME-RATE

MOVE 40 TO REGULAR-HOURS

SUBTRACT 40 FROM HOURS-WORKED GIVING OVERTIME-HOURS

MULTIPLY REGULAR-HOURS BY PAY-RATE GIVING REGULAR-PAY

MULTIPLY OVERTIME-HOURS BY OVERTIME-RATE

GIVING OVERTIME-PAY

ADD REGULAR-PAY TO OVERTIME-PAY GIVING GROSS-PAY

ELSE

MULTIPLY HOURS-WORKED BY PAY-RATE GIVING GROSS-PAY

END-IF

.

COMPUTE-FED-TAX.

MULTIPLY GROSS-PAY BY FED-TAX-RATE GIVING FED-TAX

.

COMPUTE-STATE-TAX.

\* Compute lets us use a more familiar syntax

COMPUTE STATE-TAX = GROSS-PAY \* STATE-TAX-RATE

.

COMPUTE-FICA.

MULTIPLY GROSS-PAY BY FICA-TAX-RATE GIVING FICA-TAX

.

COMPUTE-NET-PAY.

SUBTRACT FED-TAX STATE-TAX FICA-TAX FROM GROSS-PAY

GIVING NET-PAY

.

PRINT-CHECK.

MOVE EMPLOYEE-NAME TO PRT-EMPLOYEE-NAME

MOVE HOURS-WORKED TO PRT-HOURS-WORKED

MOVE PAY-RATE TO PRT-PAY-RATE

MOVE GROSS-PAY TO PRT-GROSS-PAY

MOVE FED-TAX TO PRT-FED-TAX

MOVE STATE-TAX TO PRT-STATE-TAX

MOVE FICA-TAX TO PRT-FICA-TAX

MOVE NET-PAY TO PRT-NET-PAY

DISPLAY PAYCHECK

.

CLEAN-UP.

CLOSE TIMECARDS.

END PROGRAM PAYCHECKS.

Here’s the data file:

Charlie Martin 41015

Terry Lacy 32007

and here’s the output:

$ cobc -x paycheck.cob

$ ./paycheck

Charlie Martin 41.0 15.0 $622.50 $102.09 $43.57 $38.59 $\*\*438.25

Terry Lacy 32.0 07.0 $224.00 $36.73 $15.68 $13.88 $\*\*157.71

$

**Resources to learn COBOL**

There are actually quite a number of courses and books to learn COBOL. Many of the courses are from overseas, because offshoring firms have been meeting the demand for COBOL for years.

I bought and ran through [this Udemy course](https://www.udemy.com/course/mainframe-the-complete-cobol-course-from-beginner-to-expert/), which is pretty good, and among several COBOL books on Kindle, I like [Beginning COBOL for Programmers](https://www.amazon.com/Beginning-COBOL-Programmers-Michael-Coughlan/dp/1430262532) by Michael Coughlan. There are a mountain of YouTube videos, of which I only looked at a few. [This one](https://www.youtube.com/watch?v=f3WVa2zyyGE) seems good, but [search for COBOL](https://www.youtube.com/results?search_query=COBOL) and you’ll find lots more.

There will be more to come as well. On April 9th, IBM and the Open Mainframe Project [announced a joint project](https://newsroom.ibm.com/2020-04-09-IBM-and-Open-Mainframe-Project-Mobilize-to-Connect-States-with-COBOL-Skills) to connect states with COBOL skills and to teach COBOL Programming. It has several resources, including a [bulletin board for COBOL programmers](https://community.openmainframeproject.org/c/calling-all-cobol-programmers/) who want to get back in the business, and the beginnings of [an open source COBOL course](https://github.com/openmainframeproject).

**Why does COBOL have a bad reputation?**

As you can see from this little example, COBOL is not like your normal programming language. You can’t write a compiler or a kernel module in COBOL, and the syntax is not what we’ve grown to expect. But then consider another common domain-specific language: SQL. The syntax is kind of weird, and the semantics depend on relational algebra.

 “Programming on the mainframe gives you a glimpse on how software used to be built,” said Ceballlos. “It’s like a time capsule for any modern programmer. Most of it is still very manual compared to modern DEVOPS and automation techniques.”

COBOL is, in a lot of ways, an antiquated, bad programming language. But for its particular domain, it’s better than anything else.

Tags: [bulletin](https://stackoverflow.blog/tag/bulletin/), [cobol](https://stackoverflow.blog/tag/cobol/), [stackoverflow](https://stackoverflow.blog/tag/stackoverflow/)

(<<<https://stackoverflow.blog/2020/04/20/brush-up-your-cobol-why-is-a-60-year-old-language-suddenly-in-demand/>>>)