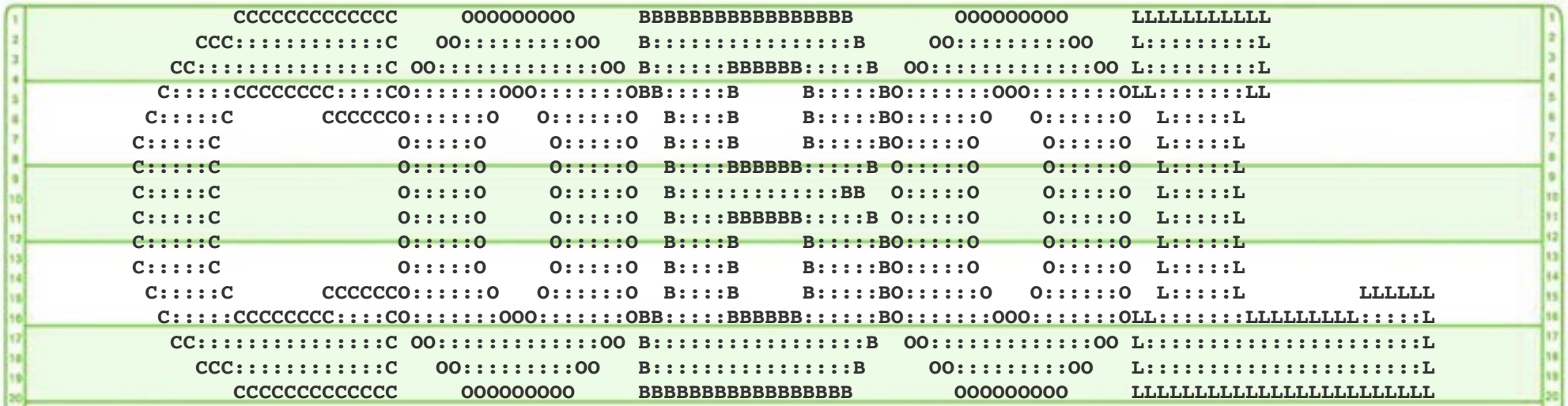


57 Years and Still Punching (Cards)

March 23, 2016

Presenter: Lincoln Roop

Talk Sponsored by Green Hills Software (who makes great compilers and other cool software, but does not make COBOL compilers).

Why COBOL?

* By the 1950s, computers were still quite expensive, but starting to see use in business.

* However, most programming was done in assembly language.

* FORTRAN slightly predates COBOL (1957) but wasn't well-suited for business tasks.

Where did COBOL come from?

* In April 1959 a group of academics, computer company employees, and computer users met at the University of Pennsylvania.

* Their Goal: To discuss the idea of a platform-independent programming language for business computing (and get the Department of Defense to pay for it).

CODASYL

* In May 1959, many of the same people met at the Pentagon to further this discussion.

* On June 4, CODASYL (The Committee on Data Systems Languages) was formed to develop a standard for COBOL.

* So yes, COBOL was in fact designed by committee.

Desired COBOL Features

* Use of English language instead of symbols (apparently mathematical notation confuses business types).

* Platform-independence even at the cost of performance. This was a big deal in the late 1950s.

* Features should not be limited by the computers of the era (for instance, size of numbers).

Development of COBOL

* COBOL was based on FLOW-MATIC, AIMACO, and COMTRAN.

* In the words of CODASYL technical advisor Grace Hopper, COBOL is “95% FLOW-MATIC.”
  * FLOW-MATIC was designed by Hopper at Remington-Rand, and since she was CODASYL's technical advisor COBOL inherited a lot of FLOW-MATIC's design decisions and syntax.
  * AIMACO was a slightly modified FLOW-MATIC derivative.
  * COMTRAN was developed by IBM, and the committee didn't want it to appear that IBM had too much influence, so the amount of material adopted from COMTRAN was kept to a minimum.

Structure of a COBOL Program

* COBOL programs are divided into **Divisions**, and **Divisions** are subdivided into **Sections**.

* Many languages have some notion of this, but often aren't so blunt.

* Other languages don't require such strict division, but this requires smarter compilers than what was feasible in the late 1950s.

Structure of a COBOL Program

* COBOL programs begin with the **Identification Division.** If you see the line "IDENTIFICATION DIVISION." at the beginning of what appears to be English text, you may have a COBOL program.

* The **Identification Division** has entries to describe what the program does, who wrote it, etc.

Structure of a COBOL Program

* Next comes the **Environment Division**
* It contains the **Configuration Section**, which gives the compiler information about the computer.
* It also contains the **Input-Output Section**, where file handles are declared.
* The **Environment Division** can be omitted if these sections aren't necessary.

Structure of a COBOL Prorgam

* After that is the **Data Division.**
* It contains the **Working-Storage Section,** where variables used by the program are declared.
* It also includes the **File Section.** Each file handle declared has to have an entry here to tell COBOL how the file is structured.
* There are other sections: **Linkage, Communication, and Screen,** but we won't cover those in this talk.

Structure of a COBOL Program

* Finally, the division we've all been waiting for: The **Procedure Division.**

* The previous three divisions handled setup and data formatting.

* This division is where your actual program code goes.

Hello, World

* The often-seen example of this online is more verbose than even the infamously verbose COBOL requires.
* The example below is far less verbose, and has been tested to work with GnuCOBOL 1.1.0 on Debian Linux.

```
*****************************************************************
Identification Division.
Program-Id. Hello-World.

Procedure Division.
    Display "Hello, World!".
    Stop Run.
*****************************************************************
```

COBOL Variables – the PICTURE Clause

* The **PICTURE** clause (part of the 5% of COBOL that came from COMTRAN) is used for declaring variables.

* It lets us define the type, size, and other properties of variables.

* For program variables, it belongs in the **Working-Storage Section.**

* For file records, it belongs in the **File Section.**

COBOL Variables – the PICTURE Clause

* Variables can be alphabetic, numeric, or alphanumeric.

* There are other variable types, but we won't cover those here.

* An important note: COBOL does math in **decimal**, not binary. Why? Accounting. Representing cents and fractional cents in binary (fixed or floating point) could result in rounding errors.

Numeric Data Types

* Numeric types contain only numbers, and can be used for doing math.

* The characters used in a PICTURE clause to declare a numeric type are 9 (digit), S (sign), and V (decimal point).

Examples:

```
01 Item-Price PICTURE 9999V99.  - 6 digits with decimal point
01 Bin-Number PICTURE 999.      - 3 digits, no decimal
01 Stock-Qty PICTURE 9(10).     - 10 digits
01 Account-Balance PICTURE S9(20)V99. – 22 digits w/sign and decimal
```

Formatting Numbers

* Let's see what the numeric variables we defined before look like with some data.

```
**************************SOME CODE OMITTED*****************************
Procedure Division.
    Set Bin-Number TO 143.
    Set Item-Price TO 0140.64.
    Set Stock-Qty TO 85806502.
    Set Account-Balance to -3.27
    Display "Bin: " Bin-Number.
    Display "Price: " Item-Price.
    Display "Qty. Avail: " Stock-Qty.
    Display "Account Balance: " Account-Balance.
    Stop Run.
**************************PROGRAM OUTPUT*********************************
Bin: 143
Price: 0140.64
Qty. Avail: 0085806502
Account Balance: -000000000000000003.29
************************************************************************
```

* **This isn't terribly human-readable, can we do better? Yes.**

Formatting Numbers

* Numeric **Edited** variables are a special type that allows us to format numbers.

* However, COBOL treats them as alphanumeric, so we can't do math on them.

Examples:

```
01 Nice-Price PICTURE ZZZ9.99.   - 6 digits with decimal, no leading
                                   zeros.
01 Nice-Qty PICTURE Z(9)9.        - 10 digits without leading zeroes.
01 Nice-Balance PICTURE -$Z(19)9.99. – 22 digits with dollar sign,
                                     Negative sign if negative, no
                                     leading zeros, and decimal point.
```

Formatting Numbers

* Let's see how the **edited** variables compare to our original ones for output:

```
**************************PROGRAM OUTPUT*****************************
Bin: 143
Price: 0140.64
Qty. Avail: 0085806502
Account Balance: -00000000000000000003.29
Formatted Price:  140.64
Formatted Quantity:   85806502
Formatted Balance:                    -$3.29
```

* This is substantially more legible, but there are still some oddities.

Actually Doing Some Calcuations

* COBOL has the standard math functions you'd expect, however English phrasing is used instead of math symbols.

* “Divide A by B giving C remainder D” is equivalent to “C = A / B” and “D = A % B” in most languages.

* Add and Subtract work similarly, exponentation is “Raise A to the power B giving C”, etc.

Is there a more compact way?

* Yes, the COMPUTE statement.
  COMPUTE makes math in COBOL look a lot like math in other languages.

* For instance, “Raise A to the power B giving C” can be replaced by
  COMPUTE C = (A ** B).

* But where's the fun in that?

Alphabetic and Alphanumeric Data

* Alphabetic variables (declared with the letter 'A' in a PICTURE clause) can only store letters.

* Alphanumeric variables (declared with the letter 'X') can store letters and numbers, as well as most printing ASCII characters.

Examples:

```
01 FIRST-NAME PIC A(20).
01 LAST-NAME PIC A(20).
01 ACCOUNT-ID PIC X(15).
01 LICENSE-PLATE PIC AAA-9999.
```

Loops – The PERFORM Statement

* The **PERFORM** statement is used for looping in COBOL, similar to DO...WHILE and FOR in other languages.

* **Inline** PERFORM statements contain the code to loop through inside them.

* There are also **Out-Of-Line** PERFORM statements, where PERFORM is used to call another paragraph of code.

Inline PERFORM

*Inline PERFORM statements look pretty much like a FOR or DO loop.

```
Identification Division.
Program-Id. Inline-Perform.

Data Division.
Working-Storage Section.
01 i PIC 99.
01 temp PIC Z(19)9.

Procedure Division.
Perform varying i from 0 by 1 until i is greater than 32
    Compute temp = (2 ** i)
    Display "2^" i " = " temp
End-Perform
Stop Run.
```

Out-Of-Line PERFORM

* Out-Of-Line PERFORM does the same thing, but makes code reuse easier.

* Think of it as a 'mini' subroutine.

```
Procedure Division.
    Set p to 2.
    Display "Displaying Powers of 2 from 0 to 8:".
    Perform Exponent varying i from 0 by 1 until i is greater than 8
    Set p to 4.
    Display "Displaying Powers of 4 from 0 to 8:".
    Perform Exponent varying i from 0 by 1 until i is greater than 8
    Stop Run.

    Exponent.
    Compute temp = (p ** i)
    Display p "^" i " = " temp.
```

Enough COBOL Syntax

* I could go on, but I think these examples are enough to give you an idea of what COBOL code actually looks like, and how it works as a language.

* If you want to learn more COBOL, there will be some resources at the end of the presentation.

The Y2k Problem

* Short-sighted COBOL program design was one of the major causes of actual Y2k-related computer issues.

* To save memory, many COBOL programs used only 2 digits to indicate the year, blindly assuming that the century would be '19'.

Example:

```
01 Birth-Date.
    05 Year Pic 99.
    05 Month Pic 99.
    05 Day Pic 99.
```

The Y2k Problem

* This was known to be a problem even by the 1970s, but was largely ignored.

* Finally, in the mid 1990s, companies started having to fix old COBOL programs still in use.

* But how to fix it? The right way would be to add 2 digits to the year, but sometimes lazy solutions were applied instead.

Fixing Y2k

* Adding 2 digits to the year requires changing record formats and updating old records.
* In most cases, the correct century for old records could be determined automatically, since those records happened in the past.

Record Change 1:

```
01 Birth-Date.
    05 Year Pic 9999.
    05 Month Pic 99.
    05 Day Pic 99.
```

Record Change 2:

```
01 Birth-Date.
    05 Century Pic 99.
    05 Year Pic 99.
    05 Month Pic 99.
    05 Day Pic 99.
```

The Lazy Y2k Fix

* Don't want to update all your records? You can kick the can a few decades by using a 'sliding window' method.

* Pick some arbitrary year (for instance “25”) and say: If the year is less than 25, the century must be 2000. If it's greater, the century must be 1900.

Did COBOL Achieve Its Goals?

* Platform Independence: Not quite. Individual computer and compiler makers made their own minor tweaks.

* User Friendly: Not today, but it was better than S/360 Assembler.

* Not restricted to computers of the era: Sure. Many COBOL programs can run on modern servers.

* Self-Documenting: It depends.

But Did It Work?

* COBOL, despite its imperfections, did serve a purpose as a language for automating business processes.

* Would you want to write an OS in COBOL? No (although it was done - IPI's BLIS/COBOL from 1977).

* Would you want to write an interactive program in COBOL? No. It really isn't intended for that.

But Did It Work?

* Would it make sense to write accounting or payroll software in COBOL? Yes, and many companies did.

* Even now, many finance transactions are processed by COBOL code.

* Is your paycheck handled by ADP? If so, it was processed by COBOL code running on an IBM System Z.

COBOL Today

* IBM still maintains COBOL for the System Z mainframe.

* HP still maintains HP COBOL for OpenVMS (Originally Digital COBOL).

* GnuCOBOL exists but isn't 100% feature complete. However, it does pass many of the ANSI COBOL 85 tests, and was used for the examples in this talk.

COBOL Today

* Object-Oriented COBOL exists. I don't know how popular it is, but a company called Micro Focus develops and maintains it.
  * Unfortunately, they didn't continue the trend set by C++ and call it “ADD 1 TO COBOL GIVING COBOL.”

* A COBOL bridge for Node.JS even exists.

In All Seriousness

* COBOL is a very old language, and it shows.

* Much of the COBOL hate revolves around design decisions that seemed logical in 1959 but are now arcane.

* It's still here because of legacy applications.

* That being said, it isn't a bad language for its intended purpose.

Practical COBOL?

* Would I recommend developing a new business application in COBOL?

* Probably not. Many of the compilers are expensive and controlled by the last vestiges of the “big iron” computer industry.

* On top of that, finding competent COBOL developers in the 21st century is no easy task.

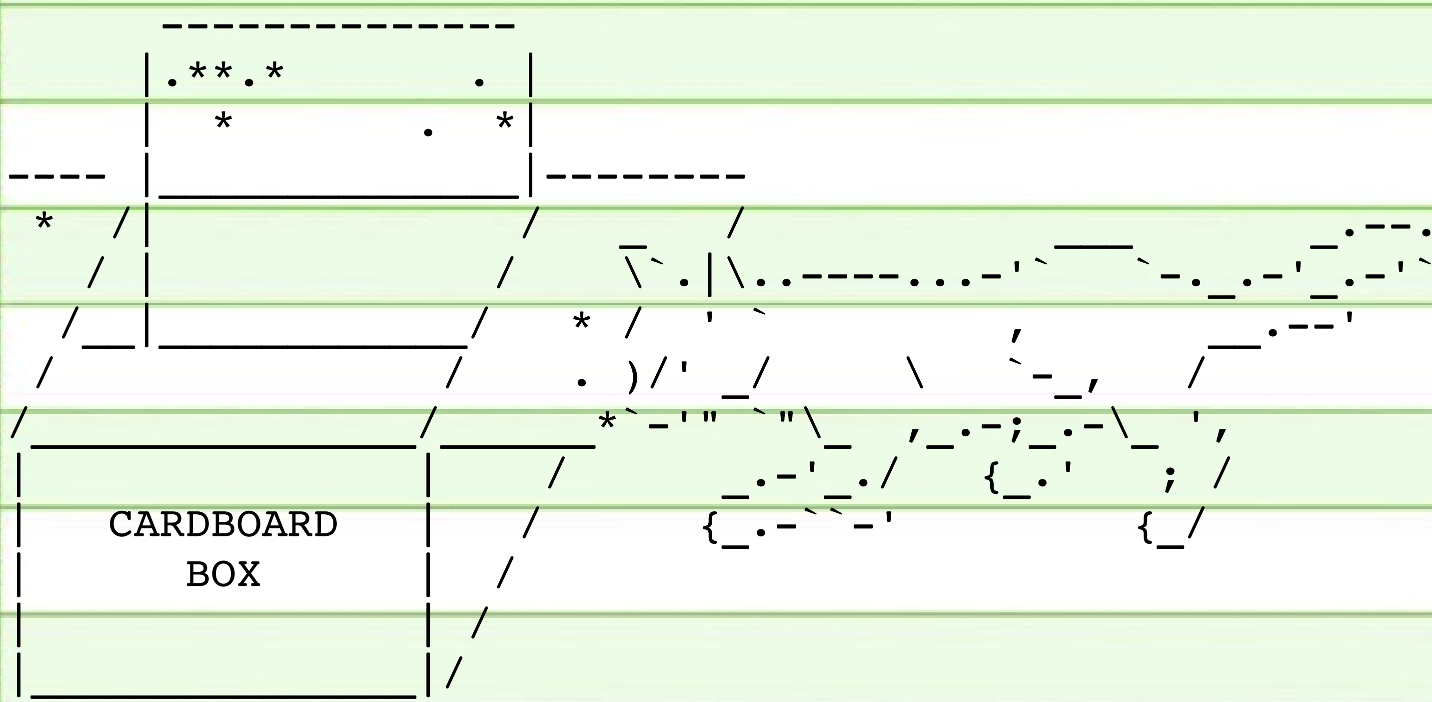
Want to learn more?

* Shockingly enough, there's still a COBOL class at Carnegie Mellon: **67-211 Introduction to Business Systems Programming**

* If you want to learn more about COBOL and/or are looking for a resume item that's guaranteed to get looks of bewilderment from recruiters, give it a try.

The End

Thank you for attending tonight's talk!



This talk brought to you in part by ASCII Art Paul.
]]]