Introductory COBOL

This unit will cover the following topics

1. The structure of a Cobol program
2. Compiling a Cobol Program
3. IDENTIFICATION Division
4. ENVIRONMENT Division
5. DATA Division
6. PROCEDURE Division

This course uses OpenCobol, for which an extremely good guide is available online.

http://opencobol.add1tocobol.com/OpenCOBOL%20Programmers%20Guide.pdf

Unit 8: Introduction to COBOL

COBOL - One of the most widely used languages on an absolute basis,and the most widely used for business applications. Technical attributes include real attempts at an English-like syntax and at machine independence.[1]

What is it?
COBOL is short for COmmon Business Oriented Language, and it was designed primarily for business, finance, and administrative systems.

Where did it originate?
One of the oldest programming languages, COBOL first appeared in 1959. COBOL was developed in 1959 over a six month period by the Conference on Data Systems Languages (CODASYL). This committee was a formed by a joint effort of industry, major universities, and the United States Government. It's ancestry included FLOWMATIC developed by Grace Murray Hopper, and the first English-like data processing language.

History?

Over the years there have been many Cobol standards:

- 1959: CODASYL (DoD)
- 1960: First Cobol compiler
- 1961: Cobol-61 (first revision)
- 1968: ANSI Cobol-68 (first standard)
- 1974: ANSI Cobol-74
- 1985: ANSI Cobol-85

How widespread is it?
- 310 billion lines of legacy code
- 5 billion lines of new COBOL written every year.
- 15% of new app. development written in COBOL
- 34% of coding activities in COBOL
- 75% of business data resides on mainframes

What makes it special?
COBOL uses a more English-based syntax. C would use a syntax similar to:

```
    balance = balance + interest
```

the equivalent in COBOL would be:

[1] Sammet, J.E., "Programming languages: History and future", *Communications of the ACM*, 15(7), pp. 601-610 (1972)

```
    ADD INTEREST TO BALANCE
```

Cobol has a number of distinct features:

- The language is simple
- No pointers
- No user defined types
- No user defined functions
- 'Structure like' data types
- File records are also described with great detail
- COBOL is self documenting

Why is it used?
Code stability - COBOL carries its legacy fairly well, major upgrades are rare. This may not be a selling point if you are in the business of developing code. However, if you are the one paying for it COBOL gets high marks on this one.
Performance - COBOL applications are generally developed where volume and/or throughput are critical (eg. processing monthly bank statements, tax returns, etc.)
Track Record - Organizations that use COBOL generally know their track record. They have a certain comfort level with cost/time estimates for major development projects using COBOL and related technologies. Taking on a new language and supporting technology to implement mission critical applications involves additional and unknown risks (and unknown benefits). Business applications between 10-30 years old are not uncommon.

Benefits:
- Wide use - broadly supported by industry and government.
- Portability - largely machine independent
- Standardized - ANSI supported
- Business - oriented - suited for business applications
- English-like - self documenting
- Data-handling - extensive features for data editing and file management
- Modularity - enforced structure

Limitations:
- Verbosity - Normally longer than other programs
- Limited scope - not suitable for scientific computations
- Rigid - verbose with strict programming rules
- Non-interactive - batch oriented
- Not-strongly typed - auto conversions and data types may be problematic
- Non-recursive

Where is it used?
COBOL is designed for developing business, typically file-oriented, applications, and is not designed for writing systems programs. COBOL applications often run in critical areas of business. For instance, over 95% of finance–insurance data is processed with COBOL.

(Arranga et al - In COBOL's Defense: Roundtable Discussion (March/April 2000) - IEEE Software).

Legacy issues
Most of the applications affected by the Y2K problem were in COBOL (12,000,000 COBOL applications vs. 375,000 C and C++ applications in the US alone - Jones, Capers - The global economic impact of the year 2000 software problem (Jan, 1997)

Beware, that programming in Cobol is not like any other language you have ever experienced. Although it appears very natural-language based with its English-like syntax, it is inherently complex, especially when dealing with arithmetic statements and file I/O, for which it is exceptionally good at... but there are A LOT of options. It also has numerous string-handling facilities to parse and analyze character-based data.

1. The structure of a Cobol program

Cobol programs are heavily modularized.

Every Cobol program has four parts:

1. **IDENTIFICATION** division - used to identify the program.
2. **ENVIRONMENT** division - associate files to program.
3. **DATA** division - specify the contents of the files, and describe the hierarchy of the data items, assign names to data items.
4. **PROCEDURE** division - process of treating the data.

For example the classic 'Hello World' written in an older Cobol (fixed-uppercase) format:

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID. HELLO.
000300 PROCEDURE DIVISION.
000400     DISPLAY "Hello, World!".
000500     STOP RUN.
```

And written in modern Cobol (free-anycase) format:

```
identification division.
program-id. hello.
procedure division.
    display "Hello World! ".
```

⚠ In Cobol, a single statement is often called a *sentence*. The sentences are always terminated with a period. A series of sentences, possibly in the form of a subprogram is called a *paragraph*.

⚠ Cobol 85 introduced block structuring.

⚠ Comments in Cobol start with an * in the first column. For example:

```
* Program to print the Rosetta Stone of programming.
identification division.
program-id. hello.
procedure division.
    display "Hello World! ".
```

2. Compiling

Cobol programs can be compiled using the OpenCobol compiler **cobc**. OpenCobol translates Cobol source code to intermediate , which is then compiled to native binary for execution, or as object code or into a dynamic library for linkage.

Consider the modern "Hello World" program from the preceding section.

```
identification division.
program-id. hello.
procedure division.
    display "Hello World! ".
```

To compile this, create a file called **hello.cob** and use the following sequence to compile it:

```
> cobc -x -free -Wall hello.cob
```

The **-x** flag tells cobc to produce an executable. The **-free** flag tells the cobc compiler to use the free source code format. Without it, the compiler will require you to enter 7 spaces at the beginning of each line.

This creates an executable program called '**hello**'.

Note that the compiler may produce few warnings of the form:

```
warning: dereferencing type-punned pointer will break
strict-aliasing rules
```

but these can just be ignored (it has to do with the gcc compiler).

3. IDENTIFICATION Division

The *identification* division only exists for the purposes of identifying a program. The two statements which normally appear in the identification division are:

```
identification division.
program-id. program-name.
```

where **program-name** is an identifier naming the program. Most versions of dialect also allow various other descriptive statements relating to information such as:

```
date-written.
date-compiled.
author.
installation.
security.
remarks.
```

However many of these are now obsolete and are normally ignored.

4. ENVIRONMENT Division

The *environment* division provides information about the computer systems components used to compile and execute the program.

```
environment division.
```

The environment section generally consists of two sections: **configuration** and **input-output**. The **configuration** section defines the computer system, and includes statements of the form:

```
configuration section.
source-computer.
object-computer.
special-names.
```

source-computer - The program upon which the program is compiled is specified with.
object-computer - The program upon which the program is executed is specified with.
special-names - A means of configuring a Cobol program created on another system.

The **input-output** section defines files the program will access. Statements are of the form:

```
file-control.
i-o-control.
```

The **i-o-control** is used to optimize certain aspects of file processing. **file-control** is used to control all aspects of file processing. These are complex statements which allow for specifications such as record locking and rollback. For example:

```
file-control.
select ifile assign to filename_in
file status is ws-status-in.
```

Assigns the file name **ifile** to the file **filename_in**, with the associated variable **ws-status-in** used to store the status of the file while being used.

5. DATA Division

The *data* division is used to give a detailed specification of the files and data. The data division may have a number of sections:

```
file section.
working-storage section.
linkage section.
local-storage section.
```

Here we describe the first two sections. The **file** section is used to define the characteristics of the input and output files named in the environment division. The **working-storage** section is used to define the variables used in the **procedure** division.

⚠ The level numbers in a Cobol program are unsigned 2-digit integers used in the data division of a program. For independent data items, level numbers **01** or **77** can be used interchangeably. For group items, the hierarchical structure can be defined by using **01** for the highest level item, and levels **02** through **49** for subordinate entries. The number **88** is reserved for condition name specification, and **66** for the **renames** clause.

5.1 Data types and variables

Variables and records are created in the **working-storage** section. A variable name in Cobol is an identifier which contains at least one alphabetic character. They may not contain any special characters other than the hyphen. For example:

```
a    alpha    2T7      federal-tax
```

The **picture** (**pic**) clause is used to describe the class and other attributes of a variable. The general form is:

```
level variable-name pic picture-string.
```

where **level** is a level number and **picture-string** is a string consisting of picture characters used to describe an item. Four of the main picture characters are: **X**, **9**, **V**, and **S**.

When the value of a variable is *numeric*, the picture character **9** is used. For example to create a variable **count**, used to store values that contain no more than five decimal digits:

```
77 count pic 99999.
```

This assumes all the values of count are positive. To allow count to contain both positive and negative values, the declaration is modified using the picture character **S**:

```
77 count pic S99999.
```

This assumes only integer values. The picture character **V** is used to indicate an assumed decimal point in a numeric data item. For example, a numeric variable **price** that contains as many as five digits with two decimal places is declared as:

```
77 price pic 999V99.
```

Or a variable interest-rate with four decimal places:

```
77 interest-rate pic V9999.
```

The picture character **X** is used to declare a string variable name. The number of Xs indicates the length of the string. For example:

```
02 city pic XXXXXXXXXXXXXXXXXXXX.
```

creates a variable **city** which can store up to 20 characters.

Because the number of **X**s and **9**s used in a picture string may be large, Cobol provides a repeat factor. For example the following are equivalent:

```
77 count pic 99999.
77 count pic 9(5).
```

The picture character **A** is used to specify a single alphabetic character. To assign an initial value to a variable, use the **value** clause. For example:

```
77 interest-rate pic V9(4) value .0397.
```

For example the declaration:

```
01 total-sum pic 999.
```

defines an integer with a field length of 3, and the procedure division statement:

```
move 73 to total-sum.
```

will store the value **073** in **total-sum**. For another example using floating-point numbers:

```
01 salary pic 9(5)V99.
```

defines a real with a field length of 7, and the procedure division statement:

```
move 9782.37 to salary.
```

will store the value **0978237** in **salary**, with an assumed decimal point between the 2 and 3.

5.2 Editing

Editing implies that data to be stored is *modified* according to the picture before the storing is performed. A number of picture characters are used for editing:

-	(minus) insertion of a blank or a -
$	currency sign
*	used in editing numeric values
,	(comma) insertion of a ,
.	(period) insertion of a decimal point
/	(slash) inserts a /
+	insertion of a + or -
0	(zero) inserts a zero character
B	inserts a space
Z	floating zero suppression

Here are some examples (^ represents a blank space):

Picture character	Input	Picture string	Output
-	-247	-----99	^^^-247
-	247	-----99	^^^^247
0	176	999000	176000
, .	112761	999,999.99	1,127.61
B	374	9B(6)99	3^^^^^^74
.	12345	99.999	12.345
.	345	9.9999	3.45
.	127	9999.99	1.27
$	19	$999	$19
*	102	***99	**102
Z	00102	ZZZ99	102
Z	00004	ZZZ99	04

Z .	001034	Z(4).99	10.34
Z .	9	Z(3).99	9.00
+ $.	-999	+$99.99	-$09.99
$	7	$$$99	$07
$	004	$$9.99	$0.04

NOTE that there are literally millions of combinations!

5.3 Data structures

Data structures in Cobol resemble records. This structure is obtained by combining several data items into a more complex structure. For example:

```
01  tool-data.
    05  plane-data.
        10  planetype    pic x(20).
        10  p-maker      pic x(20).
        10  p-year       pic 9999.
        10  p-model      pic x(20).
    05  saw-data.
        10  sawtype      pic x(20).
        10  s-maker      pic x(20).
        10  s-year       pic 9999.
        10  s-model      pic x(20).
```

Notice the hierarchical structure to the data, which is achieved using level numbers. An entry with a larger level number is subordinate to the nearest preceding entry with a smaller level number. Entries with the same level number within a group are independent of each other. Entries that have a **pic** clause are elementary data items. Conceptually, this is what this would look like:

tool-data							
plane-data				saw-data			
planetype	p-maker	p-year	p-model	sawtype	s-maker	s-year	s-model

Elementary data item names can be used to reference individual cells, for example:

```
move p-year to year.
```

Or the four field **plane-data** can be moved. Here's another example:

```
01  contact-data.
    05  last-name      pic x(15).
    05  first-name     pic x(15).
    05  phone-no       pic 9(10).
    05  email          pic x(25).
```

The **move** statement can be used to transfer contents of data structures to other fields or data structures. If the data structures are identical, the whole structure can be moved. If they are not identical, then equivalent components should be moved individually.

6. PROCEDURE Division

The *procedure* division contains the statements which are derived from the algorithm.

The procedure division is written as a sequence of statements. A statement begins with a keyword, the verb, which tells what is going to happen and after this comes the data involved. For example:

```
add 1 to total
```

ADD is the verb, 1 is a numeric literal, TOTAL is a name, and TO is a word modifying the action of ADD.

One or more statements together, terminated by a period followed by a space form a *sentence*. Sometimes sentences contain subsets that complement the statements or specify certain attributes - these are called *clauses*.

```
summation.
add X to sum-1.
add 1 to total.
go to ready.
```

Every sentence must belong to a *paragraph*. A paragraph must be followed by a period and a space. The first paragraph has the name **summation**, and contains three sentences.

It is possible to combine all the paragraphs into *sections*. A section contains one or more paragraphs preceded by a section header consisting of a name followed by the word **section** and a period.

```
final-computation section.
summation.
    add X to sum-1.
    add 1 to total.
    go to ready.
write-out.
    display sum-1.
    display total.
```

6.1 Data Movement

To assign a value to a variable, or record, use the **move** statement. The general form is:

```
move value to variable-list.
```

where **value** can be a constant, variable name, subgroup name, or record name, and **variable-list** is a sequence consisting of one or more variable, subgroup or record names. For example, assigning the numeric constant 41.5 to the variable **age**:

```
    move 41.5 to age.
```

Or assigning zero to a list of three variables:

```
    move 0 to i, j, k.
```

Note the commas are only for readability, they are not required. The next example assigns the value of the variable **prov-tax** to the variable **final-tax**.

```
    move prov-tax to final-tax.
```

⚠ What happens if a character string or number being moved is too long or too short for the receiving field? Consider what happens with the following declaration:

```
    77 jedi pic X(7).

    move “yoda” to jedi.          jedi = “yoda   “
    move “skywalker” to jedi.     jedi = “skywalk”

    77 taxes pic 999V99.

    move 2.3 to taxes.            taxes = 00230
    move 2345.678 to taxes.       taxes = 34567
```

Here are some more examples of using **move** (The □ represents a space, or empty memory cell):

	a	stored in **a**
move 0142 to a.	pic 9(4)	142
	pic X(6)	0142□□
	pic X(2)	10
	pic 9(6)	000142
	pic 9(5)V99	00142^00

	b	stored in **b**

move a to b.	pic 9(4)	0012
a is 9(2)V99	pic 9(2)V999	12^340
12v34 is stored in a	9(2)V9	12^3
	9(5).999	00012.340

	b	stored in **b**
move a to b.	pic a(6)	LUKE□□
a is a(4)	pic x(2)	LU
LUKE is stored in a		

The **move** command can also be used to move data structures, as long as the structures are completely compatible. If they are not compatible, move the data items separately.

The following are valid combinations for the **move** statement.

Alphabetic ➝ Alphabetic, Alphanumeric
Alphanumeric ➝ Alphanumeric, Alphabetic
Numeric integer ➝ Alphanumeric, numeric (any)
Numeric noninteger ➝ Numeric (any)

6.2 Arithmetic Manipulation

In addition there are eight arithmetic statements used for assignment: two versions each of **add**, **subtract**, **multiply**, and **divide**. The first form of **add** and **subtract** are:

```
add variable-constant-list to variable.
subtract variable-constant-list from variable.
```

For example:

```
add 1 to n.                   ≈    n = n + 1
subtract a, b from c.         ≈    c = c - (a + b)
```

The second form of the **add** and **subtract** statements are:

```
add variable-constant-list giving variable.
```

```
subtract variable-constant-list from variable-constant
         giving variable.
```

For example:

```
add a, b giving c.                    ≈    c = a + b
subtract 4.2, b from c giving d.      ≈    d = c - (4.2 + b)
```

The first form of **multiply** and **divide** are:

```
multiply variable-constant by variable.
divide variable-constant into variable.
```

For example:

```
multiply rt by gr.                    ≈    gr = gr * rt
divide 13 into doz.                   ≈    doz = doz / 13
```

The second form of the **multiply** and **divide** statements are:

```
multiply var-con1 by var-con2 giving variable.
divide var-con1 into var-con2 giving variable.
```

For example:

```
multiply pay by rate giving tax.      ≈    tax = pay * rate
divide 12 into rate giving monrt.     ≈    monrt = rate / 12
```

The final Cobol statement is the compute statement, and is of the form:

```
compute variable = arithmetic-expression.
```

The **compute** statement uses arithmetic operators: +, -, *, /, ** (exponentiation) There can also be replaced by the words **plus**, **minus**, **times**, or **multiplied by**, **divided by**, and **exponentiated by**.

```
compute A = B + C.
compute D = X ** 2 - Y ** 2.
compute velocity = distance / time.
compute overtime = hourly-wage * 1.5 * (time - 40).
```

Consider the following statements translated from arithmetic statement to their **compute** equivalent.

Arithmetic statement	compute equivalent
`add 1 to x.`	`compute x = x + 1.`
`add 1, y to x w.`	`compute x, w = x + y + 1.`
`add 1, y, z giving x, w.`	`compute x, w = 1 + y + z.`
`multiply 3 by y.`	`compute y = 3 * y`

`multiply x by y, z rounded.`	`compute y = x * y.`
	`compute z rounded = x * z.`

`divide 3 into x giving y, z.`	`compute y, z = x / 3.`

`divide x by 3 giving y remainder z.`	`compute y = x / 3.`
	`compute z = x - (x/3) * 3.`

⚠ Cobol normally truncates excess digits if the result has more decimal places than the variable to which the value is to be assigned. For example:

```
77 result pic 99V9.
```

Executing the following statement:

```
divide 5 into 7.9 giving result.
```

The value 1.58 would be truncated to 1.5. This can be avoided using the **rounded** clause.

```
divide 5 into 7.9 giving result rounded.
```

The value of **result** is now 1.6.

Arithmetic expressions use the traditional operators: `+, -, *, /, **`
For example:

$(x+e)^{-\sqrt{c}}$ `(x + e) ** (-(c ** 0.5))`

The table below illustrates the use of arithmetic statements:

	A	**B**	**C**	**D**	**E**
Value before execution:	5	15	8	-3	12
	Value after execution				

add C **to** E			8		20
add A B C **to** D	5	15	8	25	
add 3 A **to** A	13				
add A B **to** C D	5	15	28	17	
add A B **giving** C	5	15	20		
add C D **giving** B D E		5	8	5	5
subtract A **from** B	5	10			
subtract E 2 **from** B **giving** D		15		1	12
multiply A **by** B	5	75			
multiply 3 **by** A D	15			-9	
multiply A **by** B **giving** D	5	15		75	
divide A **into** B	5	3			
divide 4 **into** C E			2		3
divide 3 **into** 90 **giving** D				30	
divide 150 **by** B **giving** A B	10	10			