COBOL 13

Language Summary

Version 0.01

Bryan Silvia
Theory of Programming Languages
Alan Labouseur
Due: 16 May 2013

1. Introduction

COBOL traditionally stands for **C**ompletely **O**bsolete **B**urdensome **O**ld **L**anguage. COBOL 13 then is an attempt at modernizing COBOL for the year 2013, making programming in it a less painful and more enjoyable experience while still maintaining a language that is easy to understand like natural English. The language is imperative, object-oriented, and strongly typed based on COBOL 85 and Visual Basic but differing in the following ways:

1. Unlike COBOL 85 which only has three data types and Visual Basic which has seventeen, COBOL 13 has five: integer, decimal (real), character, string, and Boolean.

2. Users can define their own subprograms like in Visual Basic however, COBOL 13 provides pattern matching.

3. The identification, environment, data, and procedure divisions traditionally found in COBOL are gone in favor of write-ability along with replacing `MOVE TO` with the familiar assignment operator '`=`'.

4. While COBOL 13 maintains a level of readability like that of natural English, some keywords have been replaced by either different keyword(s) or mathematical symbol(s) to more closely reflect the structure of modern languages.

5. Unlike COBOL 85 or Visual Basic, COBOL 13 required that every expression be terminated and this is done with a semicolon ';'.

1.1 Hello World

```
Program HelloWorld
     Display "Hello World";
End Program
```

1.2 Program Structure

The key organizational concepts in COBOL 13 are as follows:

1. Every file must have at least one `Program` block where execution begins. This is analogous to a main method in Java.

2. Every file can have multiple subprograms whose visibility is defined by the keywords `Public, Protected,` or `Private` much like in Visual Basic.

3. Instead of curly braces {} to define scope, all programs, subprograms, and conditional statements begin with their respective starting keyword and are closed with the `End` keyword followed by the respective starting keyword.

4. Loops are defined by their starting keyword and are closed with the `Loop` keyword.

5. Parentheses are not required around conditional statements unless multiple conditions are being evaluated with a combination of `and` and `or`.

Example Code:

```
File Source.Example
      Program Fibonacci
            Declare x,y As Integer;
            x = 0, y = 1;
            Declare userInput As Integer;
            userInput = getInput();

            For count From 1 To userInput Do
                  Declare temp As Integer;
                  temp = y;
                  y = x + y;
                  x = y;
                  count = count + 1;
            Loop

            Display y;
      End Program

      Private SubProgram getInput() As Integer
            Display “Please enter the desired nth Fibonacci number: ”;
            Return ReadInput;
      End SubProgram
End File
```

The above example illustrates the classic Fibonacci number program. The program titled `Fibonacci` is in the file `Example` contained in the folder `Source`, similar to a package system in Java. The file has one subprogram, `getInput`, which reads the user input with the system method `ReadInput` and returns an integer value. Note the for loop structure and how `count` does not need to be declared beforehand. Also note the need to increment `count` ourselves; this is different from both COBOL 85 and Visual Basic and is meant to force the programmer to be more conscientious of how the loop is progressing.

1.3 Types and Variables

There are two kinds of types in COBOL 13: ***value types*** and ***reference types***. Variables of value types directly contain their data whereas variables of reference types store references to their data, the latter being known as objects. With reference types, COBOL 13 does not allow for two variables to reference the same object by forcing a deep copy of the reference value in the heap and assigning a new pointer. Therefore, two variables can never reference the same object, increasing data integrity. There is one exception. By default, parameter passing follows this rule. However, in order to avoid copying the reference value every time, you can specify it as passing by reference using the *ByReference* keyword like so:

```
Public SubProgram ChangeMe(me As PersonObj ByReference)
```

1.4 Statements Differing from COBOL 85 and Visual Basic

Statement	**Example**

Expression statement	`Program Expr`
	`  Declare x As Integer;`
	`  Declare s As String;`
	`  Declare r As Decimal;`
	`End Program`

`If` statement	`Program IfStat`
	`  If x > y Then`
	`  Display “Greater”;`
	`  Else If y > x Then`
	`  Display “Less”;`
	`  Else`
	`  Display “Equal”;`
	`  End If`
	`End Program`

`For` statement	`Program ForStat`
	`  For c From 0 To “Hello World”.length() Do`
	`  Display “Hello World”.characterAtIndex(c);`
	`  c = c + 1;`
	`  Loop`
	`End Program`

`While` statement	`Program WhileStat`
	`  While y < 100 Do`
	`  y = y * 2;`
	`  Display y;`
	`  Loop`
	`End Program`

2. Lexical Structure

2.1 Programs

A COBOL 13 program consists of one or more source files where at least one has a `Program` block. A source file is an ordered sequence of Unicode characters. Conceptually speaking, a program is compiled using three steps:

1. Transformation, which converts a file from a particular character repertoire and encoding scheme into a sequence of Unicode characters.

2. Lexical analysis, which translates a stream of Unicode input characters into a stream of tokens.

3. Syntactic analysis, which translates the stream of tokens into executable code.

2.2 Grammars

This specification presents the syntax of the COBOL 13 programming language where it differs from COBOL 85 and Visual Basic.

2.2.1 Lexical grammar where different from COBOL 85 and Visual Basic

The lexical grammar of COBOL 13 is similar to Visual Basic except COBOL 13 only has three field types (access modifiers) whereas Visual Basic has five. Also, should identifiers contain one or more digits, those digits must appear at the end of the identifier.

```
<field_type>        →   Public
                    →   Protected
                    →   Private

<identifier>        →   <character> <character_list>
                    →   <character> <digit_list>
                    →   <character>

<character_list>    →   <character> <character_list>
                    →   <character>

<digit_list>        →   <digit> <digit_list>
                    →   <digit>

<digit>             →   0,1,2,3,4,5,6,7,8,9
```

<character> → a,b,c, … ,z,A,B,C, … ,Z

2.2.2 Syntactic ("parse") grammar where different from COBOL 85 and Visual Basic

<program_stmt> →
```
Program <identifier>
        <stmt>
End Program
```

<subprogram_stmt> →
```
<field_type> SubProgram  <identifier> As <return_type>
        <stmt>
End SubProgram
```

<comment_stmt> →
```
Comment
        <character_list>
End Comment
```

<if_stmt> →
```
If <logic_expr> Then
        <stmt>
End If
```

<if_else_stmt> →
```
If <logic_expr> Then
        <stmt>
Else
        <stmt>
End If
```

<if_else_if_stmt> →
```
If <logic_expr> Then
        <stmt>
Else If  <logic_expr> Then
        <stmt>
End If
```

<for_stmt> →
```
For <identifier> From  <value> To <comp_expr> Do
        <stmt>
Loop
```

<while_stmt> →
```
While <logic_expr> Do
        <stmt>
Loop
```

2.3 Lexical Analysis

2.3.1 Comments

Two forms of comments are supported: ***single-line comments*** and ***delimited comments***. Single-line comments start with the characters // and extend to the end of the source line. Delimited comments start with the keyword `Comment` and end with the keywords `End Comment`. Delimited comments may span multiple lines. Comments do not nest. Typically comments should not be required since the language strives to be readable like natural English.

2.4 Tokens

There are several kinds of tokens: identifiers, keywords, literals, operators, and punctuators. White space and comments are not tokens, though they act as separators for tokens where needed.

Tokens:

identifier
keyword
integer-literal
real-literal
character-literal
string-literal
operator-or-punctuator

2.4.1 Keywords different from COBOL 85 and Visual Basic

A ***keyword*** is an identifier-like sequence of characters that is reserved, and cannot be used as an identifier.

New Keywords:

Program
File (COBOL 85 contains this keyword but has a different usage)
And (Overloaded to perform both a logical conjunction on two Boolean expressions and string concatenation)
ReadInput
Comment
Matching

Removed Keywords:

IDENTIFICATION DIVISION
ENVIRONMENT DIVISION
DATA DIVISION

PROCEDURE DIVISION
WORKING-STORAGE SECTION
Accept
Move <value | identifier> *To* <identifier>
Goto
GoSub
Add

Modified Keywords:

Original Keyword	New Keyword
Sub	*SubProgram*
Dim *Pic* | *Picture*	*Declare*
Next	*Loop*
Char	*Character*
ByRef	*ByReference*
ByVal	*ByValue*

3. Types

COBOL 13 types are divided into two main categories: ***Value types*** and ***Reference types***.

3.1 Value Types

COBOL 13 has four value types that directly contain their data on the stack: integer, decimal, character, and Boolean.

Examples:

```
File ValueTypeExample
     Program ValTypes
          Declare x As Integer;
          Declare d As Decimal;
          Declare c As Character;
          Declare b As Boolean;
          x = 15;
          d = 15.0;
          c = 'A';
          b = True;
     End Program
End File
```

3.2 Reference Types

COBOL 13 has one data type that is a reference type: string (essentially an array of characters). Additionally, arrays and any user defined Object is a reference type whose values are stored on the heap with references or pointers to their values stored on the stack. COBOL 13 does not allow de-referencing nor assignment of a pointer to another variable with the exception of parameter passing as explained in section 1.3.

Examples:

```
File Source.Student
     Object Student
          Declare name As String;
          Declare ID As Integer;

          Program Student()
               name = "";
               ID = -1;
          End Program
```

```
            Program Student(n As String, num As Integer)
                  name = n;
                  ID = num;
            End Program

            Public SubProgram getName() As String
                  Return name;
            End SubProgram

            Public SubProgram getID () As Integer
                  Return ID;
            End SubProgram

            Public SubProgram setName(n As String) As Void
                  name = n;
            End SubProgram

            Public SubProgram setID(num As Integer) As Void
                  ID = num;
            End SubProgram
      End Object
End File

File Source.ReferenceTypeExample
      Open Student;

      Program Class
            Declare s1 As New Student("Jack", 10112123);
            Declare s2 As New Student();
            s2 = s1;
            s2.setName("John");
            Display s2.getName() And ", " And s1.getName();
      End Program
End File
```

Output: "John, Jack"

4. Example Programs

4.1 Caesar Cipher Encrypt

```
File Examples.Encrypt
	Program Encrypt
		Declare s,encrypted As String;
		Declare shiftAmt As Integer;
		Display "Enter a string to encrypt: ";
		s = ReadInput.toUpperCase();
		Display "Enter a shift amount: ";
		shiftAmt = ReadInput;

		For i From 0 To s.length() Do
			encrypted = encrypted And
					shift(shiftAmt, s.characterAtIndex(i));
			i = i + 1;
		Loop

		Display encrypted;
	End Program

	Private Subprogram shift(x As Integer, c As Character) As String
		Declare y As Integer;
		y = c.toASCII();
		If y = 32 Then
			Return y.toString();
		Else
			y = y + x;
			If y > 90 Then
				y = y - 26;
			End If
			Return y.toString();
		End If
	End Subprogram
End File
```

4.2 Caesar Cipher Decrypt

```
File Examples.Decrypt
     Program Decrypt
          Declare s,decrypted As String;
          Declare shiftAmt As Integer;
          Display "Enter a string to encrypt: ";
          s = ReadInput.toUpperCase();
          Display "Enter a shift amount: ";
          shiftAmt = ReadInput;

          For i From 0 To s.length() Do
               decrypted = decrypted And
                           shift(shiftAmt, s.characterAtIndex(i));
               i = i + 1;
          Loop

          Display decrypted;
     End Program

     Private Subprogram shift(x As Integer, c As Character) As String
          Declare y As Integer;
          y = c.toASCII();
          If y = 32 Then
               Return y.toString();
          Else
               y = y - x;
               If y < 65 Then
                    y = y + 26;
               End If
               Return y.toString();
          End If
     End Subprogram
End File
```

4.3 Caesar Cipher Solve

```
File Examples.Solve
      Program Solve
            Declare s,cipher As String;
            Declare shiftAmt,maxShift As Integer;
            Display "Enter a string to encrypt: ";
            str = ReadInput.toUpperCase();
            Display "Enter the max shift amount: ";
            maxShift = ReadInput;
            shift = maxShift;

            For k From 0 To maxShift Do
                  For i From 0 To s.length() Do
                        cipher = cipher And
                                 shift(shiftAmt, str.characterAtIndex(i));
                        i = i + 1;
                  Loop
                  Display "Caesar " And shiftAmt And ": " And cipher;
                  cipher = "";
                  shiftAmt = shiftAmt - 1;
                  k = k + 1;
            Loop
      End Program

      Private Subprogram shift(x As Integer, c As Character) As String
            Declare y As Integer;
            y = c.toASCII();
            If y = 32 Then
                  Return y.toString();
            Else
                  y = y + x;
                  If y > 90 Then
                        y = y - 26;
                  End If
                  Return y.toString();
            End If
      End Subprogram
End File
```

4.4 Insertion Sort

```
File Examples.InsertionSort
      Program InsertionSort(array As Integer[])
            Declare x,y,temp As Integer;
            y = 1;

            While y < array.length() Do
                  x = y - 1;
                  temp = array[j];
                  While x >= 0 And array[x] > temp Do
                        array[x+1] = array[x];
                        x = x - 1;
                  Loop
                  array[x+1];
                  y = y + 1;
            Loop
      End Program
End File
```

4.5 Recursive Fibonacci with Pattern Matching

```
File Examples.Fib
      Program Fib
            Display "Enter the nth Fibonacci number to calculate: ";
            Display helper(ReadInput);
      End Program

      Private SubProgram helper(x As Integer Matching 0) As Integer
            Return 0;
      End
      Private SubProgram helper(x As Integer Matching 1) As Integer
            Return 1;
      End
      Private SubProgram helper(x As Integer) As Integer
            Return helper(x-1) + helper(x-2);
      End
End File
```