TEX Macros for COBOL Syntax Diagrams

MARY MCCLURE

Unisys Corporation
19 Morgan Avenue
Irvine, CA 92718-2093

ABSTRACT

COBOL syntax diagrams have a unique format that has evolved into an industry-wide standard. This format is particularly difficult to accommodate without treating the diagram as artwork. When a manual contains over a hundred syntax diagrams, as several of our manuals at Unisys do, the production process becomes quite unwieldy.

However, TEX's math mode can be exploited to allow inclusion of COBOL syntax diagrams within the document itself. This paper presents macros that typeset COBOL syntax diagrams. The paper is divided into two parts: the first demonstrates how to use a set of macros to create the diagrams, and the second part lists and explains the actual macro definitions.

1. The Diagrams

COBOL diagrams are composed of four basic kinds of elements: items that are required, items that are optional, items that offer a choice, and items that can be repeated. In addition, COBOL diagrams can contain reserved words, which are displayed in uppercase, and programmer-supplied information, which is displayed in lowercase.

These elements can be combined to form diagrams that are quite complicated. But first, let's look at each element by itself. The composition of the macros used to create these elements is discussed later in this paper (see Section 2).

1.1 Required Elements

Required elements are underlined. For example, the **VALUE** clause looks like this:

VALUE IS literal

and can be coded using simply the \req macro:[1]

```
\syntax{%
  \req{VALUE} IS literal
}
```

If several required items occur in a row, they must be identified individually. For example,

```
\syntax{%
  \req{MOVE} \req{CORRESPONDING} identifier-1 \req{TO} identifier-2
}
```

produces

MOVE CORRESPONDING identifier-1 TO identifier-2

[1] COBOL syntax diagrams always begin with the \syntax macro.

1.2 Optional Elements

Optional elements are enclosed in square brackets. For example, the **IF** statement looks like this:

IF condition [THEN] statement-1 [ELSE statement-2]

and is coded using both the \req and \option macros, like this:

```
\req{IF} condition \option{!THEN!} statement-1 \option{!\req{ELSE}
        statement-2!}
```

The function of the exclamation points (!)[2] in this example is not intuitively obvious. They are necessary to delimit optional elements that are composed of more than one item. When a series of items appears in an \option macro, it is a good idea to stack them one atop another in the macro call to keep track of where one item ends and another begins. For example, the **RECORD** clause contains a stack of optional items, of which one or none can be chosen:

RECORD CONTAINS [integer-1 TO] integer-2 [ASCII / COMPUTATIONAL / COMPUTATIONAL-2 / DISPLAY] [CHARACTERS / WORDS]

The coding for the **RECORD** clause gets a little more complicated and looks like this:

```
\syntax{%
  \req{RECORD} CONTAINS \option{!integer-1 \req{TO}!} integer-2
  \option{!ASCII!
          !COMPUTATIONAL!
          !COMPUTATIONAL-2!
          !DISPLAY!
         } \option{!CHARACTERS!
                   !\req{WORDS}!
                  }
}
```

1.3 Choice Elements

Elements that offer a choice are very similiar to optional elements. However, instead of being enclosed in square brackets, they are enclosed in curly braces to indicate that one of the items within must be chosen. For example, one form of the **OPEN** statement looks like this:

OPEN {INPUT / OUTPUT} **file-name REEL – NUMBER** {literal / data-name}

and is coded thusly:

```
\syntax{%
  \req{OPEN} \choice{!\req{INPUT}!
                     !\req{OUTPUT}!
                    } file-name \req{REEL-NUMBER} \choice{!literal!
                                                          !data-name!
                                                         }
}
```

As in optional elements, each of the items in a choice element must be delimited by exclamation points.

1.4 Elements That Can Be Repeated

When an element can occur more than once in the syntax of a command, it is followed by a series of dots called an ellipsis. For example, the **ADD** command can operate on any number of variables:

ADD {identifier / literal} **...TO identifier-n**

[2] The exclamation point was chosen as the delimiter because exclamation points are not members of the ANSI-standard COBOL character set.

The ellipsis is produced with the \repeatable macro. The **ADD** command is coded as:

```
\syntax{%
  \req{ADD} \choice{!identifier!
                    !literal!
                   } \repeatable \req{TO} identifier-n
}
```

1.5 Formatting Commands

Because COBOL syntax diagrams are quite complicated, their length often exceeds a single line. Left to its own devices, TEX will break the diagram at some point between two elements. For example, the **MULTIPLY** command is automatically broken between the seventh and eighth elements:

MULTIPLY { identifier-1 / literal-1 } **BY** { identifier-2 / literal-2 } **GIVING identifier-3** [**ROUNDED**] [, **identifier-4** [**ROUNDED**]]...

You can insert \par commands to override TEX's line-breaking algorithm and instead force the line to break earlier in the diagram. For example, if you wanted to break the above diagram into two approximately equal parts, you would code:

```
\syntax{%
\req{MULTIPLY}\choice{%
                      !identifier-1!
                      !literal-1!
                     }
    \req{BY} \choice{%
                      !identifier-2!
                      !literal-2!
                     }

    \req{GIVING}
\par
    identifier-3
          \option{%
                   !\req{ROUNDED}!}
          \option{%
                 !, identifier-4
          \option{%
                   !\req{ROUNDED}!}!}\repeatable
}
```

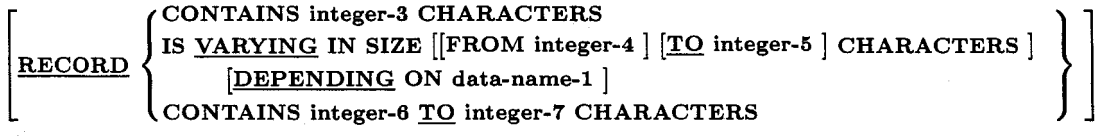
producing:

MULTIPLY { identifier-1 / literal-1 } **BY** { identifier-2 / literal-2 }
GIVING
identifier-3 [**ROUNDED**] [, **identifier-4** [**ROUNDED**]]...

A \par command can be used only to produce a line break between two elements — it cannot be used in the middle of the \option or \choice macros. If you want a line break inside an \option or \choice macro, you should use the \midbreak macro. For example, the **RECORD** portion of an **FD** statement is quite lengthy:

[**RECORD** { **CONTAINS integer-3 CHARACTERS**
IS VARYING IN SIZE [[**FROM integer-4**] [**TO integer-5**] **CHARACTERS**] [**DEPENDING ON data-nam**
CONTAINS integer-6 TO integer-7 CHARACTERS

A line break is needed in the second item of the choice element. If you insert a \midbreak command after the **CHARACTERS** item, you obtain a more satisfactory diagram:



2. The Macros

Before we can define the macros, we must declare a few variables, define the font, and make the exclamation point an active character so that it can be used as the delimiter in the `\option` and `\choice` macros:

```
\newif\ifmchoice
\newif\ifmoption
\newif\ifmreq
\newif\ifmrepeat
\newif\ifstarted

\font\cobfont=CMB10
\catcode`\!=\active
```

2.1 The \syntax Macro

All COBOL syntax diagrams must begin with the `\syntax` macro. `\syntax` sets up the environment and adds some white space before and after the diagram:

```
\long\def\syntax#1{%
  \begingroup
    \cobfont
    \textfont1=\cobfont
    \mathcode`-="012D
    \let!=\startorstop
    \baselineskip=12pt
    \lineskip=2pt
    \parindent=0pt
    \pretolerance=10000
    \medskip
    #1
    \medskip
  \endgroup
}%
```

The macro definition must be preceeded by `\long` so that `\par` commands can occur within the diagram to force line breaks.

The macro loads the font defined as `\cobfont`, defines `\cobfont` to be the font accessed when TEX is in math mode, and changes the `\mathcode` of the hyphen. By default, a hyphen maps to a minus sign in the Computer Modern Math Italic font when it is encountered in math mode (Knuth 1984:153–154, 344, 351). By changing the `\mathcode`, we map the character to the hyphen in the normal text font. This is necessary at Unisys because some of our reserved words in COBOL contain hyphens and they look strange when the hyphen is displayed as a minus sign. Your site may encounter similar problems with other characters — if anything ends up something other than you expected, you should check the character's `\mathcode` and modify it to something more appropriate.

All of this font wizardry is local to the group, so if TEX enters math mode outside of a syntax diagram, it uses the default `cmti` 10 point font and maps characters using the definitions of `plain.tex`.

The `\syntax` macro also defines the active character `!` to be a call to the `\startorstop` macro, described below in Section 2.5.

The settings of `\baselineskip`, `\lineskip` and `\parindent` control amounts of white space. `\baselineskip` determines white space between vertically stacked items in a choice of optional elements. `\lineskip` determines white space between lines of a diagram when the diagram is too long

to fit on a single line. And \parindent determines how far the diagram is indented from the left margin.

The \pretolerance command is necessary to tell TEX it is OK to break lines and create underfull \hboxes. When TEX's own line-breaking algorithm analyzes COBOL syntax diagrams, it finds breakpoints only when math mode is turned off. This conveniently occurs between each element of the diagram, but TEX calculates the badness of each of these breakpoints to be so extreme that it ignores them all unless \pretolerance is set very high.

Finally, the \syntax macro uses the \medskip macro of plain.tex to surround a COBOL syntax diagram with a certain amount of white space.

2.2 The \req Macro

All the \req macro does is underline an item. All that is required is to enter math mode and use TEX's \underline command. However, sometimes TEX encounters the \req macro when it is already in math mode, so some logic is required to determine if math mode should be turned on and off.

```
\def\req#1{%
    \ifmmode\relax\else\mreqtrue$\fi
    \underline{#1}%
    \ifmreq$\mreqfalse\fi
}%
```

2.3 The \option Macro

The main function of the \option macro is to enclose the parameter text in square brackets ([ ]). The parameter text can be quite complicated and can contain calls to the \req macro or the \choice macro. The parameter text *always* contains at least two exclamation points (!) to delimit items in the optional element.

```
\long\def\option#1{%
  \begingroup
  \startedfalse                          % Local to the group.
%
  \ifmmode\relax\else\moptiontrue$\fi    % As with \req, math mode
                                         % may or may not need to be
                                         % entered.
%
  \left\lbrack                           % Left square bracket.
    \vcenter{%
      \vbox{%
        \cobfont                         % Load the desired font.
        #1
      }%
    }%
  \right\rbrack                          % Right square bracket.
  \ifmoption$\moptionfalse\fi            % End math mode if need be.
  \endgroup
}%
```

The commands \left and \right allow TEX to determine how tall the square brackets need to be. These two commands are what make TEX so ideal for COBOL syntax diagrams (Knuth 1984:148). By using them, you make TEX stretch and shrink the brackets to correctly enclose the items, so you don't have to worry about the effect of adding or deleting items as the syntax of a COBOL command changes.

The entire parameter text is enclosed in a \vbox so that the \vcenter command can be used to center the text within the square brackets.

2.4 The \choice Macro

The \choice macro is exactly like the \option macro except that it encloses the parameter text in curly braces ({ }) rather than square brackets.

```
\def\choice#1{%
  \begingroup
  \startedfalse
  \ifmmode\relax\else\mchoicetrue$\fi
  \left\lbrace
    \vcenter{%
      \vbox{%
        \cobfont
        #1
      }%
    }%
  \right\rbrace
  \ifmchoice$\mchoicefalse\fi
  \endgroup
}%
```

2.5 Exclamation Points and the \startorstop Macro

The format of COBOL syntax diagrams requires that if more than one item occurs in an optional or choice element, the items must be stacked one on top of the other. Since TEX is designed to stack a series of \hboxes one atop another, this requirement is easily met by enclosing each item in an \hbox. But typing \hbox{ and } around each item gets a little tedious and takes up extra space on the line; you can make the exclamation point (or any character you choose) an active character and let TEX do some of the work.

If you make the exclamation point an active character and then assign it to be a control sequence that calls a macro, you can use that macro to determine if the exclamation point denotes the beginning or the end of the item. For example:

```
\catcode`\!=\active
\let!=\startorstop

\def\startorstop{%
  \ifstarted
    \egroup
    \startedfalse       % Order of commands is important here. Flag
  \else                 % should be turned on INSIDE the hbox and turned
    \hbox\bgroup        % off OUTSIDE the hbox.
    \startedtrue
  \fi
}%
```

Thus when the construction !RECORD IS! is encountered, it is transformed into \hbox{RECORD IS}.

Using \begingroup and \endgroup in the \option and \choice macro makes the value of the \started flag always local to the group. This enables nesting of elements, for example:

```
\option{!LABEL \choice{!RECORD IS!
                       !RECORDS ARE!} STANDARD!}
```

If the value of \started is not local to the group, the exclamation point before **LABEL** is correctly identified as the starting point, and \started set to true. But when TEX encounters the exclamation point before **RECORD**, the \ifstarted command is evaluated as true and TEX attempts to end an \hbox when actually it is supposed to start a second \hbox!

In fact, the whole concept of using a single character for a macro call can be carried to extremes. What if, instead of requiring the user to remember that he needs to use the \option macro to get

the square brackets in his diagram, we allow him to just type an opening square bracket where the optional element begins and a closing square bracket where the optional element ends? The following commands would allow this:

```
\catcode`\[=\active
\catcode`\]=\active

\let[=\option\bgroup
\let]=\egroup
```

This works great, but the analogous situation of allowing curly braces to be used for choice elements creates complications. Once the curly braces are made active, they can no longer be used as they were originally intended to define macros, delimit parameters in macro calls, and generally serve as beginning and end or group markers. So then some other characters, perhaps parentheses, must be redefined to take on the traditional function of the curly braces, and then what will you do when you want to use parentheses in *their* normal context?

Pretty soon things become quite confusing to a person familiar with the traditional workings of TEX. But if you can keep your character codes straight, all this re-defining of character functions might be helpful to a person sitting down to code COBOL syntax diagrams who is totally unfamiliar with TEX. Users might find it helpful to be able to type:

```
\beginsyntax

_RECORD_ CONTAINS

[!integer-1 _TO_!]

integer-2

[!ASCII
 !COMPUTATIONAL!
 !COMPUTATIONAL-2!
 !DISPLAY!]

[!CHARACTERS!
 !_WORDS_!]

\endsyntax
```

instead of what was previously described in this paper to obtain a syntax diagram for the **RECORD** clause.

2.6 The \repeatable Macro

Like the \req macro, the \repeatable macro is quite simple. It boils down to entering math mode, if need be, and calling the \ldots macro of plain.tex to create the ellipsis indicating repeatability:

```
\def\repeatable{%
  \ifmmode\relax\else\mrepeattrue$\fi
  \ldots
  \ifmrepeat$\mrepeatfalse\fi
}%
```