New RPG Free-Format operations

The free-format statements introduced in TR7 constitute a major step forward in terms of RPG code; at last RPG is nearly free! Program source code using the new instructions, such as CTL-OPT, DCL-PROC, DCL-PR, and DCL-PI, becomes instantly more legible. Fixed column specs have been replaced by declarative statements. The new format removes the need to break in, and out of FREE, (skip using the /FREE, /END-FREE) directives. (The compiler does not complain if they are present, but does not require them.) Most programmers should find the new format relatively easy to assimilate.

Control statements are probably the easiest to interpret. The 'H' control specifications are replaced with the CTL-OPT statement. The keywords that most programmers are already familiar with are still recognizable inside the new statement. Options may be keyed on multiple lines, with a semicolon delimiter at the end of each line (a), or with a single semicolon at the end of the options (b).

```
a) ctl-Opt debug(*yes) DFTACTGRP(*NO) ACTGRP('QILE');
   ctl-Opt OPTION(*SRCSTMT : *NODEBUGIO);
   ctl-Opt bnddir('QC2LE');

b) ctl-Opt DEBUG(*YES) MAIN(TranslateIFS)
    OPTION(*SRCSTMT : *NODEBUGIO) DFTACTGRP(*NO) ACTGRP('QILE')
    BNDDIR('QC2LE');
```

File specifications look very similar to fixed file entries, though there are some usage variations to be wary of when using the new file declarative statement. Using F specs, update (UF) implies that rows may also be deleted. This is no longer true using the DCL-F statement. Usage must be explicitly defined for a full procedural file; USAGE (*UPDATE : *DELETE : *OUTPUT). This is actually adds a degree of flexibility over using an F-spec to define the file as UF, in that if rows are not to be deleted, simply omit the *DELETE from the usage, and rows many not be deleted from the table. (BTW, USAGE (*INPUT:*OUTPUT) is the default for WORKSTN files.)

```
DCL-F SCGATEPF DISK(*EXT) USAGE(*INPUT:*OUTPUT:*DELETE);

DCL-F SC0215DF WORKSTN INFDS(WSDS) USROPN;
```

Device keywords DISK, KEYED, PRINTER, WORKSTN, etc. can be used to further define the file, such as providing a record length. The default for the device keyword is *EXT, so if omitted, an external definition for the file is assumed. However, certain file definitions are not supported with the new file declaration. Input primary files, secondary files, limits processing, and input or combined table files are not supported by the new file definition statement.

Standalone field definitions have not changed radically. And, in many cases, the change is intuitive. A ten-character field with the initial value of 'PROMPT' is a standalone field declared as type CHAR, with the initialization keyword providing the same value. Decimal values are not that much different; zoned decimal and packed decimal definitions are easily interpreted.

```
D PNLNAM          S              10A   INZ('PROMPT')
D COUNTER         S               7P 0
```

```
D INDEX_KEY       S              4S 0
D LEVEL_TOTAL     S              9P 2
D ERROR_CODE      S             10I
D NULL_IND        S              5I
```

```
DCL-S PNLNAM      CHAR(10) INZ('PROMPT'); // Character
DCL-S INDEX_KEY   ZONED(4,0);             // Zoned decimal
DCL-S LEVEL_TOTAL PACKED(9,2);            // Packed decimal
DCL-S ERROR_CODE  INT(10);                // Integer
DCL-S NULL_IND    INT(5);                 // Small integer
DCL-S ELEMENTS    UNS(10);                // Unsigned integer
```

There are exceptions however. Pointers are explicitly identified by a data type keyword rather than implicitly defined by data type. The varying keyword becomes a new data type of VARCHAR. For example:

```
d indPtr          s               *   inz( %addr(*in) )
D Description     s            72a    Varying

  DCL-S INDPTR POINTER INZ(%ADDR(*IN));
  DCL-S Description varChar(72);
```

The LIKE keyword takes on a slightly different aspect in free format—the length adjustment becomes the second parameter of the function. In my opinion, it makes a more logical presentation than does the fixed format definition.

```
D tPath           S        +     1    LIKE( p$path )

  Dcl-S w_path    LIKE( p$path : +1 );
```

Data structure changes are perhaps a little less intuitive than declaring standalone fields. The overlay keyword is still valid when referencing subfield definitions, but positioning using the overlay is not permitted with a named data structure.

```
Dcl-DS dsec                     Occurs(99) ;
  dsel1         char(12)                   ;
  dsel2         char(12)                   ;
  dsel3         char(12)                   ;
  dsel4         char(12)                   ;
  dspct        zoned(5:2)                  ;
  dsnam         char(25)                   ;
  dsamt        zoned(9:2)                  ;
  dsqtyx       zoned(8:0)                  ;
END-DS;
```

```
D DSEC            DS                  OCCURS(99)
D  DSEL1                   1     12
D  DSEL2                  13     24
D  DSEL3                  25     36
D  DSEL4                  37     48
D  DSPCT                  49     53  2
D  DSNAM                  54     78
D  DSAMT                  79     87  2
D  DSQTYX                 88     95  0


d indPtr          s                 *   inz( %addr(*in) )

d indicators      ds              99    based( indPtr )
d  ScreenChange                     n   overlay( indicators : 22 )
d  SflControl                       n   overlay( indicators : 50 )
d  SflDisplay                       n   overlay( indicators : 51 )
d  SflInitialize                    n   overlay( indicators : 53 )
d  SflClear                         n   overlay( indicators : 52 )
d  SflEnd                           n   overlay( indicators : 54 )
d  SflDelete                        n   overlay( indicators : 55 )
d  SflNxtChange                     n   overlay( indicators : 58 )
d  SflMSGQdisplay...
d                                   n   overlay( indicators : 59 )
```

The data structure based on the indicator array (*IN) makes use of the overlay keyword to assign a meaningful name to the indicators used by the program to communicate with the display file. Re-writing the fixed format data structure definition, results in the following code. The OVERLAY keyword is not permitted with the data structure name. Instead POS (position) is used.

```
dcl-S indPtr pointer inz(%addr(*in));

 dcl-DS indicators len(99) based( indPtr );
   ScreenChanged  ind pos( 22 );
   SflControl     ind pos( 50 );
   SflDisplay     ind pos( 51 );
   SflClear       ind pos( 52 );
   SflInitialize  ind pos( 53 );
   SflEnd         ind pos( 54 );
   SflDelete      ind pos( 55 );
   SflNxtChange   ind pos( 58 );
   SflMSGQdisplay ind pos( 59 );
 end-DS;
```

However, the OVERLAY keyword is still valid—as described in the code below. An unnamed data structure (*N), has two subfields defined. One is set to overlay the other. In this instance, overlay is still valid. It is fairly simple to remember the rules, use position when making subfields relate to the data structure name, and use overlay to demonstrate the parsing of subfields.

```
dcl-DS *n ;
    DEC   binDec(4);
    BIN   char(1) OVERLAY(DEC:2);
 end-DS;
```

Externally defined data structures changes are very subtle with the new declarative statements. With fixed format definitions the file name used to describe the data structure is treated as an externally defined name. In the new DCL-DS declarative, an unquoted name would be treated as if it was a reference to a named constant, so the file name must be enclosed in quotes. Note: since no additional fields have been defined, the END-DS may appear on the same line as the DCL-DS. (See examples above, where the END-DS appears after the subfield definitions.) What I find to be rather odd, END-DS is not always required. When using the keyword LIKEDS or the keyword LIKEREC, the END-DS is not actually required, since no additional subfields may be defined when using these keywords.

```
D AFTER         E DS                  EXTNAME(SCGATEPF) prefix(a_)

  DCL-DS DATAREC EXTNAME('SCGATEPF') END-DS;
```

Provisions have been made with the data structure declarative statements to accommodate the program status data structure and the file I/O feedback area. The program status data structure may be defined with the keyword PSDS when declaring a data structure.

```
 DCL-DS pgm_stat PSDS;
     status *STATUS;
     routine *ROUTINE;
     library CHAR(10) POS(81);
 END-DS;
```

Specify the INFDS keyword on the file specification, giving the file information data structure a name, and then define the data structure. Subfields may be defined using the position keyword, or the predefined subfield definitions.

```
 DCL-F anyfile DISK(*EXT) INFDS(anyfileDS);

  DCL-DS anyfileDS;
     status *STATUS;
     opcode *OPCODE;
     msgid CHAR(7) POS(46);
 END-DS;
```

A final note on data structure definitions—subfields may be explicitly defined. In the example below, CUSTOMERNAME is implicitly defined as a subfield of an unnamed data structure. ‘Select’ happens to be a reserved word, a valid op code in free format calculations. It may still be used, but must be qualified with the subfield declarative statement. Customer name is not an op code, so the subfield declaration is not required. Address is not an op code, so the declarative is not required. But, the explicit declarative is valid.

```
  DCL-DS *N;
     DCL-SUBF Select char(1);
     CustomerName CHAR(10);
     DCL-SUBF address CHAR(40);
  END-DS;
```

Defining procedures within RPG have been streamlined considerably with the application of TR7. Without having to break out of free-format, procedure entries are now defined much in the same manner as logic loops, DCL-PROC to start the procedure definition and END-PROC to mark the end of the

procedure. Procedure prototypes and interfaces are much the same, DCL-PR and DCL-PI to mark the start of the definition and END-PR, END-PI to mark the end of the definition. However, if the procedure is internal, there is no need to include a procedure prototype.

```
dcl-proc ClearRecordFormatDS Export;
  dcl-PI *n end-PI                ;
  clear tableRec                  ;
  return                          ;
end-proc                          ;
```

The declarative statement for the procedure requires a name and whether it is exported or imported. However, the procedure interface may or may not reflect a name entry. *N may be used in place of a name on the procedure interface statement. If no input parameters are defined for the interface, the END-PI statement may appear on the same line as the DCL-PI.

The end of procedure statement may or may not contain a name; it is not required. However, for procedures that contain more instructions than may be viewed on a single page, it is probably a good practice to include the procedure name on the end statement, so there is no need to scroll up to determine what procedure is ending. Where the procedure is just a few lines, the name is probably not material.

```
dcl-PR gateFileMaintenance   extPgm('SCO215RP');
   p_Area char(15);
   p_Gate char(10);
   p_code int(5);
   p_mode char(1) const;
END-PR;
```

One major departure from the expected is the inclusion of a parameter declarative. The parameter SELECT happens to be the name of an RPG op code. You may still use the name in the parameter list as long as it is qualified with the DCL-PARM declaration. (Why you would choose to name a field the same as an op code is another departure from the expected!)

```
dcl-PR gateFileSelection  ind  extPgm('SCO205RP');
   p_Area char(15);
   p_Gate char(10);
   dcl-Parm Select char(10);
END-PR;
```

The new format should be easy to incorporate into new code—most developers will find it fairly easy to migrate away from fixed-format specifications to the new free format statements. It is a welcome enhancement to the language—one I was not sure I would see before I retired! The incremental steps to make RPG a free-form language have been slow to materialize. I thought that IBM's first iteration was somewhat clumsy. I still recall the frustration of having to break out of free-format (/end-free) in order to execute an embedded SQL operation, because the SQL pre-compiler was still looking for C+ in order to validate SQL statements.

```
C/Exec SQL
```

```
C+          Set Option Commit = *None
C/End-Exec
C/Exec sql
C+ SELECT *
C+    into :GATEPR
C+    from SWSECGP
C+ WHERE MGGATE = :p_GATE   AND
C+       MGCATG = :p_CAT
C/End-exec
```

Even after embedded SQL statements became part of free-format code, there was still the requirement to end free-format instructions to code P-specs. With the new free-format definitions, RPG code actually does look like a free-format language—almost.

When I work with JavaScript, or define a cascading style sheet, I can start typing in column one, and key a statement that ends in column 102 if I choose. The editor, (and the browser), have no trouble interpreting that statement. Though it is vestigial, RPG still clings to its fixed column roots. Try beginning a DCL-S statement in column seven and you will see what I mean. Start a statement in column eight and try to extend it past column 80 and see what happens. RPG is free—free to roam anywhere between column seven and column 81. Of course, this concession to the past is to preserve legacy code and allow the compiler to make an easy decision as to whether to interpret a statement as fixed-column or free-format.

To the best of my recollection, Paul Conte introduced RPG/Free; a free format version of RPG that demonstrated that it was possible to write free format instructions nearly 24 years ago, (News 3X/400). Speaking for myself, I am grateful that the Toronto Language Lab has finally followed suit. This time IBM has managed to produce a well-executed free format version of RPG, without mixing in compiler directives, while retaining the frame work of its columnar forerunner—not an easy task. Good job.

- Steve Croy