RPG V5R4

Additional Material For V5R4 – RPGIV

George Farr

IBM Toronto Laboratory

COMMON Mar 26-30,2006

Summary of RPG enhancements

- EVAL-CORR: Assign matching subfields from one data structure to another PREFIX keyword can completely remove characters
- Pass null-indicators with parameters
- Better debugging for null indicators
- XML support
  - XML-SAX: Access to a SAX parser
  - XML-INTO: Read directly from XML into an RPG variable
- Debugging aid for XML-SAX
- Free-form syntax checking (SEU and WDSc)

EVAL-CORR operation code

If you have two data structures whose subfields have the same names, you can use EVAL-CORR to assign all those subfields at once

EVAL-CORR operation code

```
DS1
 name   10A
 id      5P 0
 addr  100A
 status  1A
```

```
DS2
 name   20A
 status  1A
 id     10A
```

```
EVAL-CORR DS1 = DS2;
```

```
Equivalent to
 EVAL DS1.name = DS2.name;
 EVAL DS1.status = DS2.status;
 // Note: it has no effect on “ID” or “ADDR”
```

More on EVAL-CORR

- Subfields are assigned one by one, in the order they appear in the target data structure
- If any subfields overlap in the target data structure, the last assigned value “wins”
- Subfields not affected by EVAL-CORR are unchanged by the operation (unless some other subfield overlaps them)
- If an exception (decimal data error, or invalid varying length) occurs while assigning a subfield, the EVAL-CORR operation will halt immediately with the exception.
- However, if the data structures are defined with LIKEDS or LIKEREC so that they have the same parent data structure, then the EVAL-CORR operation is optimized to simply copy the storage from data structure to another, rather than assign the subfields one by one. If any subfield has an invalid value, this would not be detected by the EVAL-CORR operation; assigning subfield-by-subfield would detect the errors.

ALWNULL considerations

EVAL-CORR is **similar** to a series of EVAL operations, but not identical.

If ALWNULL(*USRCTL) was specified, and there are some null-capable subfields involved, the null-indicators for the subfields are also assigned as part of the EVAL-CORR operation.

EVAL-CORR with NULLs

```
DS1: fld1 10A ALWNULL
     fld2 10A ALWNULL
     fld3 10A
     fld4 10A
```

```
DS2: fld1 10A ALWNULL
     fld2 10A
     fld3 10A ALWNULL
     fld4 10a
```

```
     EVAL-CORR DS1 = DS2;
```

Equivalent to

```
  DS1.fld1 = DS2.fld1;
  DS1.fld2 = DS2.fld2;
  DS1.fld3 = DS2.fld3;
  DS1.fld4 = DS2.fld4;
  %NULLIND(DS1.fld1) = %NULLIND(DS1.fld1);
  %NULLIND(DS1.fld2) = *OFF;
```

Note: DS2.fld2 is not null-capable, so DS1.FLD2 null-indicator is just set off. DS1.FLD3 is not null-capable, so %NULLIND(DS2.FLD3) is ignored.

OPTIONS(*NULLIND)

A null-capable field has a “null-indicator” or “null-byte map” associated with it. This indicator is kept in separate storage. A data structure having null-capable subfields has an associated null-byte-map data structure kept in separate storage.

- Without OPTIONS(*NULLIND), if a null-capable parameter is passed, the called procedure has no access to the parameter’s null-byte map. If the parameter is an externally-described data structure or is defined with LIKEREC, and the external file has null-capable fields, then the called procedure would be able to use %NULLIND on the parameter’s subfields, but the called procedure would not be referencing the passed parameters null-indicators.
- When OPTIONS(*NULLIND) is specified for a parameter, the null-byte map is passed with the parameter, giving the called procedure direct access to the null-byte map of the caller's parameter.

PREFIX used to remove characters from names

F & D specs: **PREFIX(" : number_of_characters)**

- When an empty character literal (two single quotes specified with no intervening characters) is specified as the first parameter of the PREFIX keyword for File and Definition specifications, the specified number of characters is removed from the field names. For example if a file has fields XRNAME, XRIDNUM, and XRAMOUNT, specifying PREFIX(":2)on the File specification will cause the internal field names to be NAME, IDNUM, and AMOUNT.
- If you have two files whose subfields have the same names other than a file-specific prefix, you can use this feature to remove the prefix from the names of the subfields of externally-described data structures defined from those files.

Example showing how this aids the EVAL-CORR operation:

```
FILE1: fields F1NAME F1IDNO
FILE2: fields F2NAME F2IDNO
 * Use PREFIX to remove 2 characters from the names
D ds1          E DS          EXTNAME(file1) QUALIFIED
D                            PREFIX(’’:2)
D ds2          E DS          EXTNAME(file2) QUALIFIED
D                            PREFIX(’’:2)
   EVAL-CORR ds1 = ds2; // assigns the NAME and IDNO subfields
```

XML support

RPG's XML support currently only supports reading the XML documents. There is no support for creating or modifying XML documents. RPG has two ways to get data from XML documents.

- XML-INTO allows the programmer to read all or part of the XML document directly into an RPG variable.
- XML-SAX allows the programmer to get the data from XML document, one piece at a time. The XML data is passed to a user-provided procedure which may be called many times during the course of a single XML-SAX operation.
- RPG supports XML documents in memory (character or UCS-2 data) or in Integrated File System files.

XML support

In case you are unfamiliar with XML, here is a small XML document:

```
<pet species="dog">
  <name>Ruff</name>
  <age>3</age>
  <agetype>dog years</agetype>
</pet>
```

Elements: pet, name, age, agetype

Attribute: species

Brief introduction to XML

An XML document represents data in a tree structure. The data is all in text form.

Here is an introduction to the syntax:
http://en.wikipedia.org/wiki/XML#Quick_syntax_tour

There are two basic types of XML parsers: SAX and DOM.

- "SAX" stands for "Simple API for XML". A SAX parser operates by reading the XML document and calling a user-provided procedure for every "event" that the parser finds.
- "DOM" stands for "Document Object Model". The parser creates a tree representing the XML document, and provides several functions for randomly accessing the tree.

Both facets of the RPG support use a SAX parser, but only XML-SAX exposes the actual XML events to the RPG programmer. XML-INTO hides the complexity of the SAX event handling.

%XML builtin function

%XML(document { : options })

The %XML builtin function is used to identify an XML document and specify options to control how the XML document is parsed.

- The first operand can be the actual contents of the XML document, or it can indicate the location of the XML document. It can be a character or UCS-2 expression, either a constant or a variable.
- The second operand is optional; it allows the RPG programmer to customize the XML operation. It can be a character expression, either a constant or a variable.

The valid options depend on the operation code, XML-SAX or XML-INTO. One option is common to both operations: the “doc” option, which specifies whether the first operand is the actual document, or the location of the document.

The options are specified as a character expression in the form ‘opt1=value1 opt2=value2’, for example ‘doc=file ccsid=job’.

The option=value pairs are separated by blanks.

Examples:

```
%XML('<name>Bob</name>');        // 1st operand is an XML document
%XML('custinfo.xml' : 'doc=file'); // 1st operand is the name of an IFS file
```

Default values for %XML options

The valid options and valid values for each option depend on the context (opcode) of %XML.

If you would like different default options, you can define a constant value with the defaults you want, and then append updates or additions on each operation. (If the RPG runtime finds an option specified more than once, it takes the last value specified, so be sure to specify your defaults first.)

For example, if you would prefer the default for the “allowmissing” option for XML-INTO to be “yes”, and the “doc” option to be “file”, you could code like this:

```
D intoDfts     c     ‘allowmissing=yes doc=file ’
     xml-into ds %xml(xmldata
                 : intoDfts + ‘doc=string’);
     // overrides “doc=file” with “doc=string”
```

XML-INTO operation code

Using the XML-INTO operation requires knowledge of the XML document's structure. For example, an RPG programmer may know that an XML document will be laid out as follows, with the "document element" named "cust" and two "child elements" named "name" and "id":

```
<cust>
  <name>ABC Electronics</name>
  <id>1233273</id>
</cust>
```

The structure of the document can be represented by an RPG data structure named "cust" with two subfields "name" and "id". For example

```
D cust            ds
D    name                      100A    varying
D    id                         15P 0
```

The XML document can be "assigned" directly into the data structure:

```
XML-INTO cust %XML(custXml : 'doc=file');
```

- XML-INTO can be used to read data into a scalar variable, scalar array, or array of data structures.
- RPG subfields can be filled with data from either an XML element or XML attribute.

XML-INTO assumptions

By default, the RPG runtime makes several assumptions about the XML document:

- It assumes that XML element and attribute names in the document are the same as the names of the RPG variable and subfields, and that the XML names will be in lower case.
- For data structures:
  - It assumes that there is an XML element or attribute for every RPG subfield, and that every XML element or attribute has a matching RPG subfield to receive the data.
  - It assumes that if one of the RPG subfields is an array, that there will be exactly as many repeated XML elements as there are elements in the RPG array.
- If the target variable is not an array, it assumes that the outermost XML document has the same name as the RPG variable. If the target variable is an array, it assumes that the outermost element contains several child elements with the same name as the RPG variable.

However, XML-INTO has several options that allow the RPG programmer to override some of these assumptions.

XML-INTO options

- **case**: If the element and attribute names are all in upper case, specify ‘case=upper’; if they are in unknown or mixed case, specify ‘case=any’.
- **allowmissing:** Use ‘allowmissing=yes’ if the document may not contain XML data to fill every RPG subfield.
- **allowextra:** Use ‘allowextra=yes’ if the document may have XML data that has no matching RPG subfield to receive it.
- **trim:** Used to control whether “whitespace” (blanks, tabs, new-line characters) should be trimmed from the XML data before being assigned to the RPG subfields. The default is “all”, meaning that leading and trailing whitespace is trimmed, and interior whitespace is reduced to a single blank; To have the whitespace included in the RPG subfields, use ‘trim=none’. (Whitespace is always trimmed before assigning to numeric, date etc.)
- ***… more on next page***

XML-INTO options, continued

- **ccsid:** Used to control the CCSID that the document is parsed in. By default, the RPG runtime will choose the CCSID that will best preserve the data in the document ('ccsid=best'); if the document is in UCS-2, or if it is a CCSID other than the job CCSID, the RPG runtime will convert the document to UCS-2 before parsing. Otherwise, the RPG runtime will parse the document in the job CCSID. If the RPG programmer wants the document to be parsed in the job CCSID (which may involve fewer CCSID conversions during parsing), use 'ccsid=job'. If the RPG programmer wants the document to be parsed in UCS-2, use 'ccsid=ucs2'.
  - **Restriction:** The parser does not support all job CCSIDs. When the document may be in an unsupported CCSID, 'ccsid=ucs2' should be used.
- ***… more on next page***

XML-INTO options, continued

- **path:** Used to locate the XML data within the XML document. For example, option 'path=info/num' indicates that the outer XML element will be called "info" and it will have a child element called "num" which should be used to obtain the data for the RPG variable. 'path=employees/manager/address' indicates that the outer XML element called "employees" will have a child called "manager" which itself will have a child called "address". The "address" XML element should be used to obtain the XML data for the RPG variable.

  The path option can also be used to "rename" the outer XML element. If the data structure's name is 'custinfo', but the outer XML element is called "customer", then 'path=customer' should be used to inform the RPG runtime of the actual name of the outer element. **Note**: there is no way to indicate a different name for the XML element or attribute to match a particular subfield.

XML-INTO with repeating elements

An XML document may contain repeating XML elements, such as in the following example:

```
<customers>
 <cust name="J Smith" id="031"/>
 <cust name="M Jones" id="402"/>
</customers>
```

For this type of document, an RPG array can be used as the receiver variable.

```
D cust   DS        DIM(10)
D  name       50a  VARYING
D  id         10a
 XML-INTO customers
    %xml(xmldoc : 'doc=file');
```

Note the following about this example:

- The "path" option was not specified, even though the XML elements matching the RPG array are inner elements. The default for an array is to assume that the XML elements are child elements of the outer element. (This is because an XML document can only have one outer element.)
- The dimension of the RPG array is 10, but there are only 2 repeated elements in the XML document. This situation does not require "allowmissing=yes", since the target array is not a subfield.
- The actual number of array elements set by the operation can be obtained from a new "number of XML elements" subfield in the PSDS.

XML-INTO with an unknown number of repeating elements

If the number of repeating elements may be larger than the maximum number of elements in an RPG array (32767), then a more complex form of XML-INTO must be used.

Rather than read the repeating XML data into an RPG array variable, instead the parser will read the XML data into a temporary array, and call an RPG "XML-INTO handling procedure" whenever it has read enough XML data to fill the temporary array.

```
<band>
  <member name="John"/>
  <member name="Paul"/>
  <member name="George"/>
  <member name="Ringo"/>
</band>
```

If the RPG handler is capable of receiving 3 elements at once, it would be called twice, with the following data:

1. John, Paul, George (3 elems)
2. Ringo (1 elem)

The following slides describe the required RPG coding.

%HANDLER

%HANDLER(prototype : firstParameter)

The %HANDLER builtin function is used to describe a user-defined procedure that is to be called during the operation.

- The first operand is the name of the prototype for the handler. The required parameters and return type of the handler depend on the context in which %HANDLER is specified.
- The second operand is the parameter which is received by the **first** parameter of the handler procedure.

XML-INTO with %HANDLER

An XML-INTO handler has the following rules:

- The return type is a 4-byte integer (10i 0). Returning a value of zero indicates that parsing should continue to find XML data and call the handler again if necessary. Returning any other value indicates that parsing should stop immediately. This will cause an RPG exception to be issued for the XML-INTO operation.
- The first parameter can be any type, passed by reference. The parameter is passed as the second operand of the %HANDLER builtin function.
- The second parameter specifies the type of the array elements to be filled by the XML data. It can have any type, but the DIM and CONST keywords must be coded.
- The third parameter is for the actual number of elements passed in for that particular call to the handler. (Recall the previous example where the first call had 3 elements (John, Paul, George) and the second call had only 1 element (Ringo).)

XML-INTO %HANDLER example

Sample definitions required to use %HANDLER with XML-INTO:

```
 * The definition for the data structure to receive the XML data
D cust          DS         DIM(10)
D  name             50a    VARYING
D  id               10a
 * The prototype for the handler
D custHandler   PR  10i 0
D  foundBlankId       n
D  custs                   LIKEDS(cust)
D                          DIM(32767)
D  numCusts         10i 0  VALUE
```

The XML-INTO operation looks like this. Note that the "path" option is required for XML-INTO with %HANDLER.

```
 * Parameter to pass to the handler
D hadBlank      S      n
 /free
    hadBlank = *OFF;  // initialize the "communication" parameter
    xml-into %HANDLER(custHandler : hadBlank)
             %XML(xmldoc : 'doc=file path=customers/custs');
```

XML-INTO %HANDLER example

```
P custHandler  B                 EXPORT
D custHandler  PI  10i 0
D  foundBlankId…
D                    n
D  custs                         LIKEDS(cust)
D                                DIM(32767)
D  numCusts          10i 0 VALUE
 /free
   for i = 1 to numCusts;
       --- do something with the data -
   --
       if custs(i).id = ‘ ‘;
          foundBlankId = *on;
       endif;
   return 0;
 /end-free
P custHandler  E
```

- The handling procedure, custHandler is called repeatedly whenever the parser has read enough XML data to fill 32767 RPG elements.
- The variable “blankId” specified as the second parameter of %HANDLER is passed directly by reference to the handling procedure by the parser. If the handling procedure changes the parameter, the changes will be visible to the “XML-INTO” procedure.
- The procedure itself would normally do something useful with the data. In this case, it illustrates how the first parameter is used for communication between the RPG procedure doing the XML-INTO operation and the custHandler procedure.

SAX parsing

Consider this XML document

```
<cust
    type="business">
      <name>ABC
    Tools</name>
</cust>
```

The parsing "events" are

- Start document
- Start element ("cust")
- Attribute name ("type")
- Attribute value ("business")
- Start element ("name")
- Characters ("ABC Tools")
- End element ("name")
- End element ("cust")
- End document

XML-SAX operation

The XML-SAX operation identifies a handling procedure to handle the various SAX events generated by the parser. The procedure will be called several times during the XML-SAX operation, once for each event.

If the procedure needs to retain information about the events, it could either use static variables in the procedure, or it could use the “communication area” parameter passed by the procedure doing the XML-SAX operation.

%XML options for XML-SAX

- doc – same as for XML-INTO, identifies whether the first operand is an XML document, or the location of an XML document
- ccsid – indicates the CCSID for the data to be passed to the XML-SAX handling procedure. The default is the job CCSID; to have the data passed as UCS-2, use ‘ccsid=ucs2’; to have another CCSID for example 1208, use ‘ccsid=1208’. **Warning**: if a CCSID other than the job CCSID is used, and a character variable is used to access the data, the RPG compiler will assume that the character data is in the job CCSID. This may cause incorrect results. For example, if comparisons are done with other character data in the program, the comparison operands would not be in the same CCSID, and the results of the comparison would be incorrect.

XML-SAX events

For most events, the parameters passed to the procedure will include a string parameter containing the data associated with the event, and a string-length parameter. This parameter is defined as a pointer passed by value; the pointer points at either character or UCS-2 data, depending on the “ccsid” option. The pointer can be used as a basing pointer for a character or UCS-2 variable.

The data may be longer than can be handled by an RPG variable. In that case, the RPG programmer must use some means other than a simple based variable to access the data; for example, it could call a procedure in some other language.

For a few events, such as the start-document, end-document or exception events, there is no data associated with the event, and accessing the string parameter will result in an exception.

XML-SAX handler

An XML-SAX handler has the following rules:

- The return type is a 4-byte integer (10i 0). Returning a value of zero indicates that parsing should continue to find XML data and call the handler again if necessary. Returning any other value indicates that parsing should stop immediately. This will cause an RPG exception to be issued for the XML-SAX operation.
- The first parameter can be any type, passed by reference. The parameter is passed as the second operand of the %HANDLER builtin function.
- The second parameter identifies the SAX event. There are several new RPG special names in the form *XML_START_ELEM, *XML_ATTR_NAME etc, that can be used to determine which event is being handled.
- The third parameter is the pointer to the data for the event, passed by value.
- The fourth parameter is the length of the data, an 8-byte integer (20i 0) passed by value. In cases where no data is passed, the value is -1.
- The fifth parameter is only meaningful for exception events. It gives the return code associated with the exception. The meanings of the return codes are listed in the ILE RPG Programmer's Guide.

XML-SAX handling procedure

This handler looks for an “start-element” event with the event data “name”; on the next “characters” event, it updates the first parameter with the character data. If the XML document is <info><name>Bob</name></info>, then the “nameVal” parameter, passed by the XML-SAX operation, would be set to the value “Bob”.

```
P getName         B
D getName         PI        10I 0
D  nameVal                  20A
D  event                    10I 0 VALUE
D  string                     *   VALUE
D  stringlen                20I 0 VALUE
D  excpId                   10I 0 VALUE
D stringVal       S      65535A   BASED(string)
D hadNameElem     S           N   STATIC INZ(*OFF)
 /free
      if event = *XML_START_ELEM and %subst(stringVal:1:stringlen) = ‘name’;
        hadNameElem = *ON;
      elseif hadNameElem and event = *XML_CHARS;
        nameVal = %SUBST(stringVal : 1 : stringlen); // update caller’s parm
      endif;
      return 0;
 /end-free
P                 E
```

XML restrictions

For details, see the ILE RPG Programmer's Guide.

- Not all CCSIDs are supported. See the list of supported CCSIDs in the ILE RPG Programmer's Guide. If the job CCSID is one of the unsupported CCSIDs, the option 'ccsid=ucs2' should be used.
- If the RPG programmer has a pointer to XML data, and the data is longer than 64K, the data must be copied to a temporary IFS file before it can be parsed. There is a example (figure 76) in the ILE RPG Programmer's Guide of an RPG procedure that will do this. (Use the HTML version of the manual to enable cut and paste.)
- The parser can handle XML data up to 2 147 483 408 bytes in length. If the document is being parsed in UCS-2, it can only handle up to 1 073 741 704 UCS-2 characters. If the XML document is too long, the XML operation will be attempted; if the parser eventually requires the additional data in the document for the operation, an exception will be issued.
- The parser does not replace entity references for entities defined in the DOCTYPE declaration. When it encounters entity references, it simply reports the name of the reference. For XML-SAX, see the *UNKNOWN_REF and *XML_UNKNOWN_ATTR_REF events. For XML-INTO, the RPG variable will contain the reference in the form "&refname;".
- The parser does not support name spaces. It considers the colons in element and attribute names to be simply part of the name. XML-INTO cannot match an XML name in the form ns:name to an RPG subfield name.

Debugging aid for XML-SAX handlers

If DEBUG(*XMLSAX) is specified on the H spec, an array called _QRNU_XMLSAX will be generated into the module. This array contains the names of the events.

In the debugger:

```
==> eval event
 EVENT = 2
==> eval _QRNU_XMLSAX(event)
 _QRNU_XMLSAX(event) = 'ATTR_NAME'
```

Other DEBUG keyword changes

The DEBUG keyword controls the code that is generated by the compiler. It does not control debug view information (that is controlled by the DBGVIEW command parameter).

The DEBUG keyword formerly had only *YES and *NO values. If not specified, DEBUG defaulted to *NO; if specified without a parameter, it defaulted to *YES.

DEBUG(*YES) means two things:

- If there is a DUMP operation, perform the formatted dump.
- If there are unused fields on input specifications, load the fields anyway.

New DEBUG keyword values

The defaults for the DEBUG keyword remain the same.

Several new values are now allowed for the DEBUG keyword.

- *INPUT – load unused fields on I specs
- *DUMP – perform DUMP operations
- *XMLSAX – generate the XML-SAX debug aid
- *YES – same as *INPUT : *DUMP. This value is deprecated, since *INPUT and *DUMP are both more descriptive and allow more granular control.
- * NO – no debugging aids are generated into the module

Other changes

- Syntax-checking of free-form calculations is performed in both V5R4 SEU and WDSc 6.0.1.
- Free-form embedded SQL is supported. (This is actually an enhancement for the SQL precompiler for SQLRPGLE.)