Deep dive into RPG free-form

Barbara Morris
IBM

A fully-free RPG program, starting in column 1

```
**free
ctl-opt bnddir('ACCRCV');

dcl-f custfile usage(*update);
dcl-ds custDs likerec(custRec);
dcl-f report printer;

read custfile custDs;
dow not %eof;
   if dueDate > %date(); // overdue?
      sendOverdueNotice ();
      write reportFmt;
      exec sql insert :name, :duedate into
             mylib/myfile;
   endif;
   read custfile custDs;
enddo;
inlr = '1';

dcl-proc sendOverdueNotice;
   /copy invoices
   sendInvoice (custDs : IS_OVERDUE);
end-proc;
```

First line has **FREE

All free-form statements

No fixed-form allowed

Fully free-form RPG

7.1 and 7.2 PTFs

PTFs for 7.1 and 7.2 provide the ability to code free-form RPG starting in column 1 and going to the end of the line.

There is no practical limit on the length of a source line.

- CRTSRCPF has a limit of 32766
- IFS files have no limit

Fully free-form RPG – how long should your lines be?

Various style-guides for other languages recommend a maximum line length of 80, 132, 120 etc.

The “80” comes from IBM punch cards.

Google [maximum length of a code line] to see some of discussions about line length.

If you create your RPG source files with RCDLEN(112), then that gives you 100 characters, which is probably ideal.

Fully free-form RPG – source must start with **FREE

Any source member that contains fully-free code must have **FREE in column 1 of the first line of the source.

```
**FREE
ctl-opt main(greeting);

dcl-proc greeting;
   dsply ‘Hello’;
end-proc;
```

Fully free-form RPG

- All code in a **FREE source member must be free-form. If you need any fixed-form code, you can put it in a /COPY file

- Source lines must not begin with ** unless they are the special directives for compile-time data, file-translation, or alternate collating sequence.

- /FREE and /END-FREE are not allowed in a **FREE source member

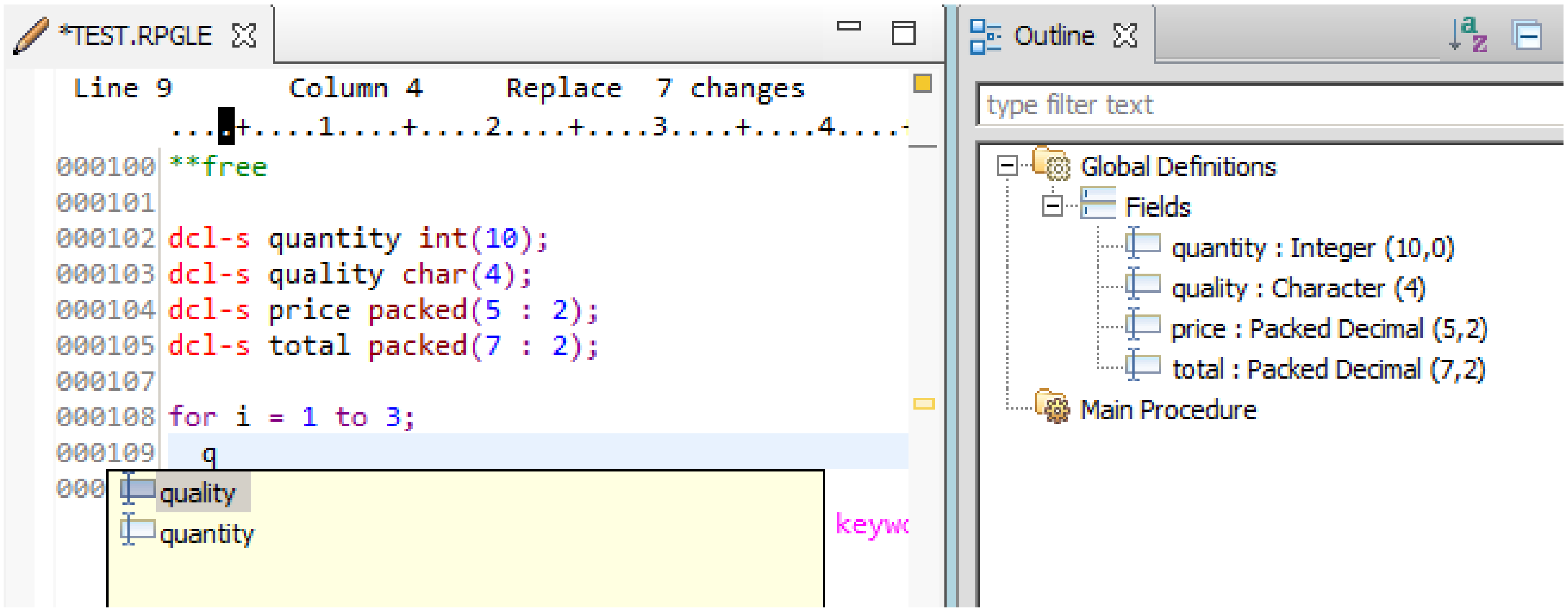
Fully free-form RPG – copy files

- Each copy file has its own source mode

- A copy file is always assumed to have column-limited source mode unless it has **FREE in line 1

Fully free-form RPG – RDI

RDI supports fully-free RPG code (SEU does not)



```
....+....1....+....2....+....3....+....4....
**free

dcl-s quantity int(10);
dcl-s quality char(4);
dcl-s price packed(5 : 2);
dcl-s total packed(7 : 2);

for i = 1 to 3;
  q
```

```
quality
quantity
```

```
Global Definitions
  Fields
    quantity : Integer (10,0)
    quality : Character (4)
    price : Packed Decimal (5,2)
    total : Packed Decimal (7,2)
Main Procedure
```

Fully free-form RPG – Embedded SQL

The SQL precompiler supports fully-free RPG code

```
**FREE
dcl-s greeting char(10);

exec sql set :greeting = 'Hello';
dsply greeting;
return;
```

What is wrong with fixed-form code?

- Most programmers today have never seen fixed form code
- When they see RPG code like this, it looks like gibberish

```
H bnddir('ACCRCV') dftactgrp(*no)
Fcustfile  if   e             disk
Freports   o    e             printer
```

- Here's what happens when a non-RPG programmer tries to make a change

```
H bnddir('ACCRCV')
Fcustfile  if   e             disk
Freport   o    e             printer
RNF0289E Entry contains data that is not valid; only valid data is used.
RNF2013E The Device entry is not PRINTER, DISK, SEQ, WORKSTN or SPECIAL;
     defaults to DISK.
RNF2003E The File Type is not I, O, U, or C; defaults to O if File
     Designation is blank, otherwise to I.
RNF2005E The Sequence entry is not blank, A, or D; defaults to blank.
... more error messages
```

RPG is still not 100% free

There are still some areas where RPG is not free (and may never be)

- I specs and O specs must be coded in fixed-form
  - I and O specs are considered deprecated by many RPG programmers in favor of externally-described files
- Code related to the RPG cycle must be coded in fixed-form
  - The cycle is considered deprecated by many RPG programmers in favor of using SQL for scenarios where the cycle formerly shone

What does an all-free RPG mean?

- Fewer "secret codes" to remember ("E in column 19 means externally-described")
- Indented code is more maintainable
- Better token-colorization in the RDI editor, allowing programmers to have the same look-and-feel for RPG code as for other languages like Java or PHP
- New programmers will only have to learn how to use RPG, without having to struggle with how it is coded

Removal of many frustrations

- /FREE and /END-FREE in every procedure
- Two lines for many definitions in fixed-form

```
D getNextCustomer...
D                 pr


 vs
 dcl-pr getNextCustomer;
```

- Insufficient room in D-spec keywords for long strings

```
D HSSFCellStyle   c
D                                'org.apache.poi.hssf.-
D                                 usermodel.HSSFCellStyle'
```

vs

```
 dcl-c HSSFCellStyle 'org.apache.poi.hssf.usermodel.HSSFCellStyle';
```

More information

Documentation

- The ILE RPG Reference in the 7.2 and 7.3 Knowledge Center has all the information about free-form. The free-form information also applies to 7.1.

RPG Café wiki page with PTF information:

https://ibm.biz/rpgcafe_fullyfree_rpg

Conversion

- RDI free-form conversion does not do any conversion from H F D P to free-form.
- ARCAD has a product that converts H F D C and P specs to fully-free-form.
- Linoma's conversion tool converts H F D C and P specs to fully-free-form.

The details

Let's look at the details

- General features
- Control (H)
- File declaration (F)
- Data declaration (D)
- Procedure (P)

Some general features

The new statements all

- Start with an "opcode"
- End with a semicolon

Just like calculation statements in RPG:

```
if duedate > today;
   sendAngryLetter (customer);
endif;
```

Some general features

Unlike free-form calculations, can have /IF, /ELSEIF, /ELSE, /ENDIF within a statement

```
dcl-s salary
  /if defined(large_vals)
     packed(13 : 3)
  /else
     packed(7 : 3)
  /endif
     ;
```

Some general features

Can mix fixed-form and free-form without /FREE and /END-FREE

Example: Defining the TAG for SQL "whenever"

```
    exec sql whenever sqlerror goto err;
    ...
    return;
C       err           tag
    ok = *off;
    reportSqlError ();
```

Control statements

CTL-OPT (Control Option) statement

- Start with CTL-OPT
- Zero or more keywords
- End with semicolon

```
ctl-opt option(*srcstmt : *nodebugio)
        dftactgrp(*no);
```

Control statements

- Can have multiple CTL-OPT statements
- The rules about not repeating keywords apply across all statements

```
   ctl-opt; // no keywords
   ctl-opt option(*srcstmt : *nodebugio)
           dftactgrp(*no); // two keywords
 H datfmt(*iso) text('My Program')
   ctl-opt alwnull(*usrctl); // free again
```

Control statements

One little enhancement for free-form H:

- If there is at least one free-form control statement, you don't need DFTACTGRP(*NO) if you have one of the ACTGRP, BNDDIR, or STGMDL keywords

File statements

DCL-F (Declare file) statement

- Start with DCL-F
- File name
- Keywords
- End with semicolon

File statements

- Only full-procedural and output – no cycle, RAF or table files

- The name can be longer than 10 as long as there's an EXTFILE keyword (and an EXTDESC keyword if externally-described)

```
dcl-f year_end_report printer
          oflind(overflow)
          extdesc('YERPT')
          extfile(*extdesc);
```

File statements – the device

Device keyword or LIKEFILE must be the first keyword

DISK, PRINTER, SEQ, SPECIAL, WORKSTN

- Defaults to DISK

Externally-described: *EXT (default)

Program-described: record-length

```
dcl-f orders; // defaults to DISK(*EXT)
dcl-f qprint printer(132);
dcl-f screen workstn; // defaults to *EXT
```

File statements – the usage

USAGE keyword

*INPUT, *OUTPUT, *UPDATE, *DELETE

Equivalent of fixed-form File Type (I, O, U, C) and File-Addition

Default for USAGE depends on the device

```
dcl-f orders disk;      // *INPUT
dcl-f report printer;   // *OUTPUT
dcl-f screens workstn;  // *INPUT : *OUTPUT
```

- SEQ and SPECIAL default to USAGE(*INPUT)

File statements – the usage

Some usage values imply other values

- *UPDATE implies *INPUT
- *DELETE implies *UPDATE and *INPUT

```
// USAGE(*INPUT : *UPDATE)
dcl-f orders disk usage(*update);

// USAGE(*INPUT : *UPDATE : *DELETE)
dcl-f arrears disk usage(*delete);
```

Can specify implied values explicitly too

```
dcl-f orders disk usage(*update : *input);
```

File statements – the usage

If you specify the USAGE keyword, the defaults are not considered

```
// output only
dcl-f f1 disk usage(*output);

// input and output
dcl-f f2 disk usage(*input : *output);
```

File statements – difference for *DELETE

In fixed form, U enables update and delete

In free form, *UPDATE does not enable delete

- *DELETE must be coded explicitly

File statements – Keyed files

For externally-described files, KEYED keyword

```
dcl-f orders disk keyed;
```

For program-described files, KEYED(*CHAR:len)

```
dcl-f generic disk(2000) keyed(*CHAR:100);
```

File statements – Program-described keyed files

Only character keys supported for program-described

For other types, use a data structure

```
dcl-f generic disk(2000) keyed(*CHAR:7);

dcl-ds key len(7) qualified;
   item_num packed(12);
end-ds;

key.item_num = 14;
chain key generic;
```

File statements

F specs can be mixed with D specs (even in fixed form).

Group related items together

```
dcl-f orders
      usage (*update : *output) keyed;
dcl-ds orders_dsi
      likerec (ordersR:*input);
dcl-ds orders_dso
      likerec (ordersR:*output);
dcl-s num_orders int(10);
```

```
dcl-f report printer;
dcl-ds report_ds
      likerec (reportR:*output);
```

File-related data structures must be defined after the file

If your file has a related data structure such as an INFDS or INDDS, the data structure must be coded after the file.

Bad: The DS is defined before the file

```
~~dcl-ds orders_infds;~~
~~   status *status;~~
~~end-ds;~~
~~dcl-f orders infds(orders_infds);~~
```

Good: The DS is defined after the file

```
dcl-f orders infds(orders_infds);
dcl-ds orders_infds;
   status *status;
end-ds;
```

File statements

Named constants can be used for file keywords

```
dcl-c YEAR_END_RPT_FILE 'YERPT';
dcl-f year_end_report printer
      oflind(overflow)
      extdesc(YEAR_END_RPT_FILE)
      extfile(*extdesc);
dcl-ds report_ds
      extname(YEAR_END_RPT_FILE:*output);
```

Data definition statements

- Start with DCL-x
- Item name – can be *N if not named
- Keywords
- End with semicolon

```
dcl-s name like(other_name);
```

Standalone fields

The first keyword must be a data-type keyword.

```
dcl-s salary packed(9:2) inz(0);
```

If you are using the LIKE keyword, it doesn't have to be first.

```
dcl-s annual_salary inz(0)
          like(salary : +2);
```

Data-type keywords

Some data-type keywords match the Data-Type entry exactly

```
CHAR, INT, POINTER ...
```

Some merge the Data-Type entry with another keyword

```
VARCHAR = A + VARYING

DATE = D + DATFMT

OBJECT = O + CLASS
```

Data-type keywords – String data types

Fixed length:

- CHAR(characters)
- GRAPH(characters)
- UCS2(characters)

Varying length

- VARCHAR(characters)
- VARGRAPH(characters)
- VARUCS2(characters)

Varying length with specific prefix-size

- VARCHAR(characters : 4)
- VARGRAPH(characters : 4)
- VARUCS2(characters : 4)

Indicator

- IND

Data-type keywords – Numeric data types

Decimal types with default zero decimal postions:

- PACKED(digits)
- ZONED(digits)
- BINDEC(digits) ("BINDEC" is explained on the next slide)

Decimal types with specific decimal positions

- PACKED(digits : decimals)
- ZONED(digits : decimals)
- BINDEC(digits : decimals)

Integer, unsigned, float

- INT(digits)
- UNS(digits)
- FLOAT(bytes)

BINDEC keyword – reduce confusion over RPG's "binary" type

RPG's "binary" type is a decimal type stored in binary form, not a "true binary".

```
D binfld          S                   9B 3
```

- Values between -999999.999 and 999999.999

RPG programmers see "binary" in API documention and think they should code B in their RPG programs

Non-RPG programmers see "binary" as the RPG data type, and think it means true binary

- When they want an 4 byte binary, they code 4B which is a 2-byte binary with 4 digits

Other data types

Date, time, timestamp with default format

- DATE
- TIME
- TIMESTAMP

Date, time

- DATE(*YMD-)
- TIME(*HMS:)

Pointer and procedure pointer

- POINTER
- POINTER(*PROC)

Object

- OBJECT(*JAVA : CLASS) (parameters not needed for the prototype of a constructor)

Tip for remembering the data-type keywords

If there is a related built-in function, the data-type keyword has the same name:

%CHAR	- CHAR and VARCHAR
%GRAPH	- GRAPH and VARGRAPH
%UCS2	- UCS2 and VARUCS2
%DATE	- DATE
%TIME	- TIME
%TIMESTAMP	- TIMESTAMP
%INT	- INT
%UNS	- UNS
%FLOAT	- FLOAT

Exception: %DEC. The decimal data types are PACKED, ZONED, BINDEC.

Data structures

Data-structures end the subfield list with END-DS

- not used for LIKEDS or LIKEREC data structures

END-DS is optionally followed by the DS name

```
dcl-ds info;
   name varchar(25);
   price packed(4 : 2);
end-ds info;
```

If no subfields, code END-DS on the DCL-DS line

```
dcl-ds prt_ds len(132) end-ds;
```

Data structures

DCL-DS is used to begin a data structure.

END-DS is not used if LIKEREC or LIKEDS is used (because you can't code additional subfields)

```
dcl-ds info likeds(info_t);
dcl-ds custInDs likerec(custrec : *input);
```

END-DS is needed for an externally-described DS

```
dcl-ds custDs extname('CUSTFILE') end-ds;
```

Subfields

Subfields officially start with the DCL-SUBF opcode

The opcode is optional unless the name is the same as a free-form opcode

```
dcl-ds info;
   name char(25);
   dcl-subf select int(10);
end-ds info;
```

DCL-SUBF must be used because "select" is an opcode supported in free-form

Same as the rule for EVAL and CALLP

```
name = 'Sally';
eval select = 5;
```

Subfields

The POS keyword replaces

- From-and-to positions
- OVERLAY(dsname)

```
D info     DS
D  sub1       25   34A
D  sub2              D     OVERLAY(info:100)
D  sub3             5P 2   OVERLAY(info)
```

```
dcl-ds info;
   sub1 char(10) pos(25);
   sub2 date pos(100);
   sub3 packed(5 : 2) pos(1);
end-ds info;
```

Subfields

Free-form OVERLAY only overlays subfields

- No free-form equivalent for OVERLAY(ds:*NEXT)
- OVERLAY(ds:*NEXT) means "after all previous subfields" which is the same as not having the OVERLAY keyword at all
- SUB3 starts at position 101, after all previous subfields.

```
D info      DS
D  sub1          1  100A
D  sub2         11   20A
D  sub3               5A   OVERLAY(info:*next)
```

Equivalent:

```
dcl-ds info;
   sub1 char(100) pos(1);  // 1-100
   sub2 char(10) pos(11);  // 11-20
   sub3 char(5);           // 101-105
```

Nested data structures

7.2 and 7.3 PTFs

If you define your data structure in free-form, you can code nested data structures directly

- Define your subfield using DCL-DS and END-DS

```
dcl-ds info qualified;
   num_employees int(10);
   dcl-ds employees dim(100);
      name varchar(25);
      salary packed(6:2);
   end-ds;
end-ds;
```

PTF information is at **http://ibm.biz/spring_2017_rpg_enhancements**

PSDS and INFDS

Use the PSDS keyword to define a program-status data structure.

Use values like *STATUS to define the special PSDS or INFDS subfields.

```
dcl-ds statusDs PSDS;
   moduleStatus *STATUS;
end-ds;

dcl-f myfile INFDS(myfileInfds);

dcl-ds myfileInfds;
   myfileStatus *STATUS;
end-ds;
```

Prototypes and procedure interfaces

Prototypes and procedure interfaces are similar

```
dcl-pr qcmdexc extpgm;
   cmd char(3000);
   cmd_len packed(15 : 5);
end-pr;

dcl-pr init end-pr; // no parameters

dcl-pr init;
end-pr;      // can be a separate statement

dcl-pi *n varchar(25); // name not needed
   id int(10);
end-pi;
```

Bonus feature:
EXTPGM parameter
is optional

When do you need a name for the procedure interface?

A ”cycle-main” procedure is the procedure coded before any subprocedures are coded.

If you need a prototype for your main procedure (probably in a /COPY file)

- Then you need to give a name for the PI so the RPG compiler knows which prototype to use

```
ctl-opt dftactgrp(*yes);
dcl-pr subproc end-pr;
dcl-pr mypgm end-pr; // “Main” prototype
dcl-pi mypgm end-pi; // “Main” PI

subproc(); // These are the calculations
return;    // for the main procedure
```

… Subprocedures follow the main procedure …

When do you need a name for the procedure interface?

If your program is **never** going to be called from another RPG program or module

- If it is always called from a CL program
- If it is the command-processing program for a command

Then you don't need a prototype for the program

(Otherwise, you do need a prototype, and it should be in a /copy file)

If you don't need a prototype, just code *N as the name for the procedure interface

```
ctl-opt dftactgrp(*yes);
dcl-pr subproc end-pr;
dcl-pi *N end-pi; // "Main" PI (no PR)

...
```

*DCLCASE for external procedure names

A common bug:

- EXTPROC is needed for the mixed-case name
- The programmer uses copy-paste and forgets one change

Bug!

```
D Qc3EncryptData...
D                 pr      extproc('Qc3EncryptData')
D Qc3DecryptData...
D                 pr      extproc('Qc3EncryptData')
```

Use *DCLCASE to avoid retyping the name:

```
dcl-pr Qc3EncryptData extproc(*dclcase);
dcl-pr Qc3DecryptData extproc(*dclcase);
```

- Less error prone when coding
- Easier for code reviewers to see that it's correct

Parameters

Parameters officially start with DCL-PARM

DCL-PARM is optional. Same rule as for subfields

```
dcl-pr proc;
   name char(25) const;
   dcl-parm clear ind value;
end-pr;
```

Procedure statements

Begin a procedure

- DCL-PROC
- Procedure name
- Keywords
- End with semicolon

```
dcl-proc myProc export;
```

End a procedure

- END-PROC
- Optional procedure name
- End with semicolon

```
end-proc myProc;
```

or

```
end-proc;
```

Procedure example

```
dcl-proc getCurUser export;
   dcl-pi *n char(10) end-pi;

   dcl-s curUser char(10) inz(*user);

   return curUser;
end-proc;
```

- The PI uses the place-holder *N for the name
- END-PI is specified as a keyword at the end of the DCL-PI statement

Can use named constants for keywords

```
dcl-c SYS_NAME_LEN 10;

dcl-ds sys_obj qualified;
   obj char(SYS_NAME_LEN);
   lib char(SYS_NAME_LEN);
end-ds;
```

Can use named constants for keywords

In fixed form, some keywords allow literals to be specified without quotes: DTAARA, EXTNAME, EXTFLD

What data area is used for fld1?

```
D fld1          S         10A   DTAARA(dta1)
```

What about fld2?

```
D dta2          C               'MYLIB/DTAARA2'
D fld2          S         10A   DTAARA(dta2)
```

DTAARA keyword difference

In free-form, an unquoted name is always a variable or named constant

```
D dta1          C                 'MYLIB/DTAARA1'
```

```
D fld1a         S        10A    DTAARA(dta1)
dcl-s fld1b char(10) dtaara('DTA1');
```

*LIBL/DTA1

```
dcl-s fld1c char(10) dtaara(dta1);
```

MYLIB/DTAARA1

```
D fld2a       S        10A    DTAARA(*VAR:nameFld)
   dcl-s fld2b char(10) dtaara(nameFld);
```

Value of nameFld

Gotchas

- Update does not imply delete
- END-DS, END-PR, END-PI are needed at the end of a subfield or parameter list (even when there are no subfields or parameters)
- Keywords like DTAARA and EXTNAME that assume unquoted names are named constants or variables

(These have already been discussed)

Another gotcha

If you are in the habit of using ellipsis at the end of D and P spec names

```
D customerName...
D          S              50A
```

That will not work for free-form declarations

```
dcl-s customerName...
       char(50);
```

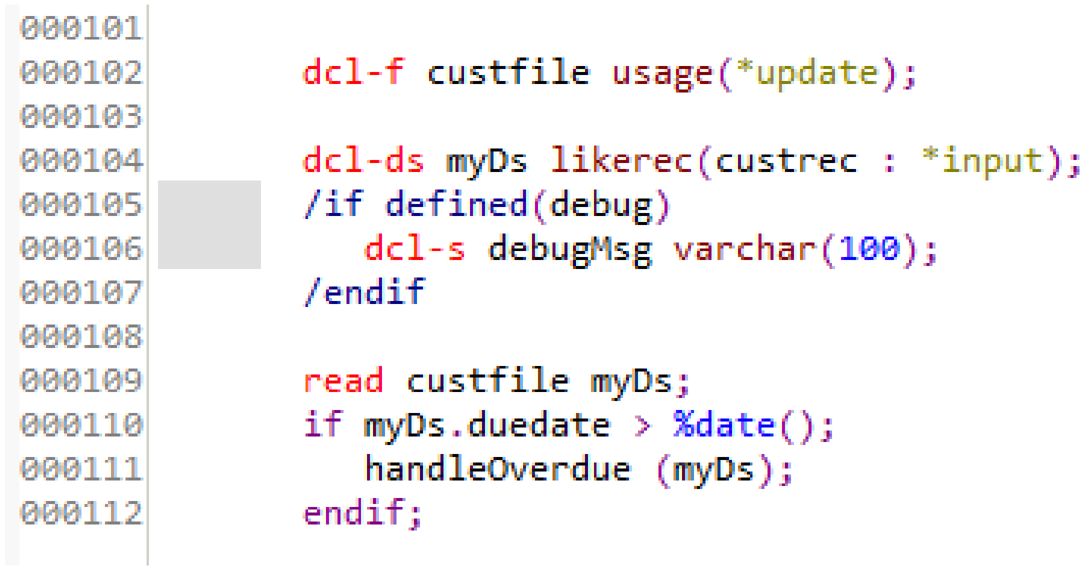
The name is customerNamechar, and "(50)" is found where the compiler expects to find the data type.

```
dcl-s customerName
       char(50);
```

Colorization in RDI

Much more control for colorizing your code

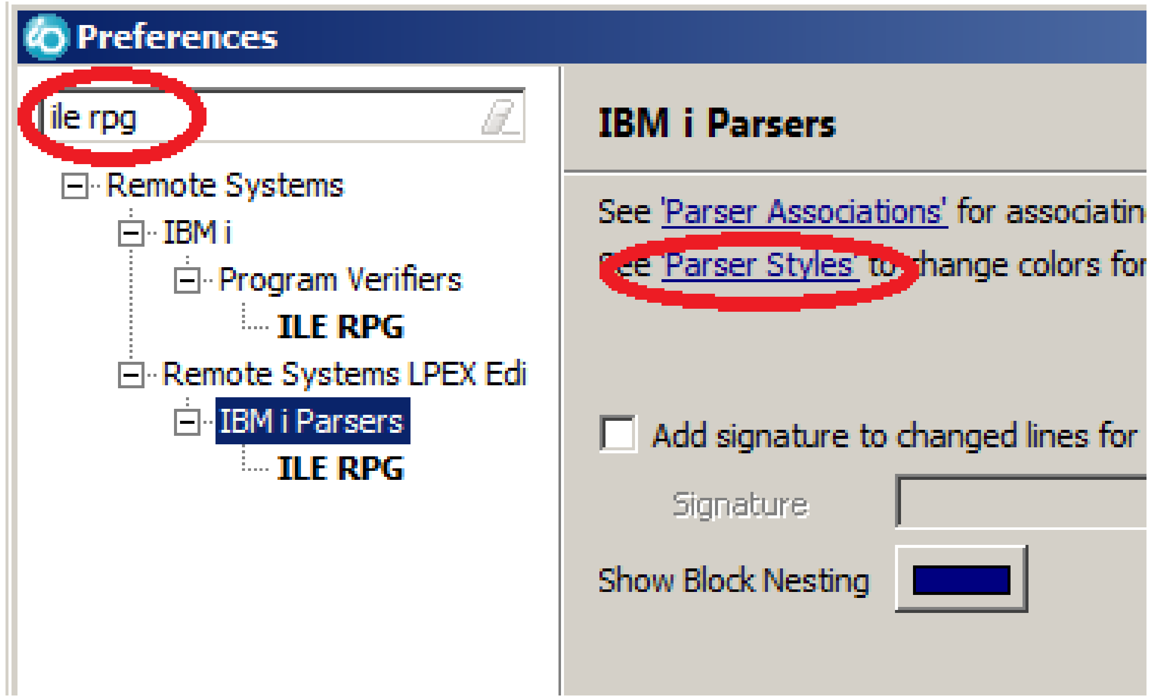
Here is some code with the default colors



```

       dcl-f custfile usage(*update);

       dcl-ds myDs likerec(custrec : *input);
       /if defined(debug)
          dcl-s debugMsg varchar(100);
       /endif

       read custfile myDs;
       if myDs.duedate > %date();
          handleOverdue (myDs);
       endif;
```

Navigate to the color preferences

- Window > Preferences
- Search for ILE RPG
- Click on Parser Styles


Preferences

ile rpg

- Remote Systems
  - IBM i
    - Program Verifiers
      - ILE RPG
  - Remote Systems LPEX Edi
    - IBM i Parsers
      - ILE RPG

IBM i Parsers

See 'Parser Associations' for associatin

See 'Parser Styles' to change colors for

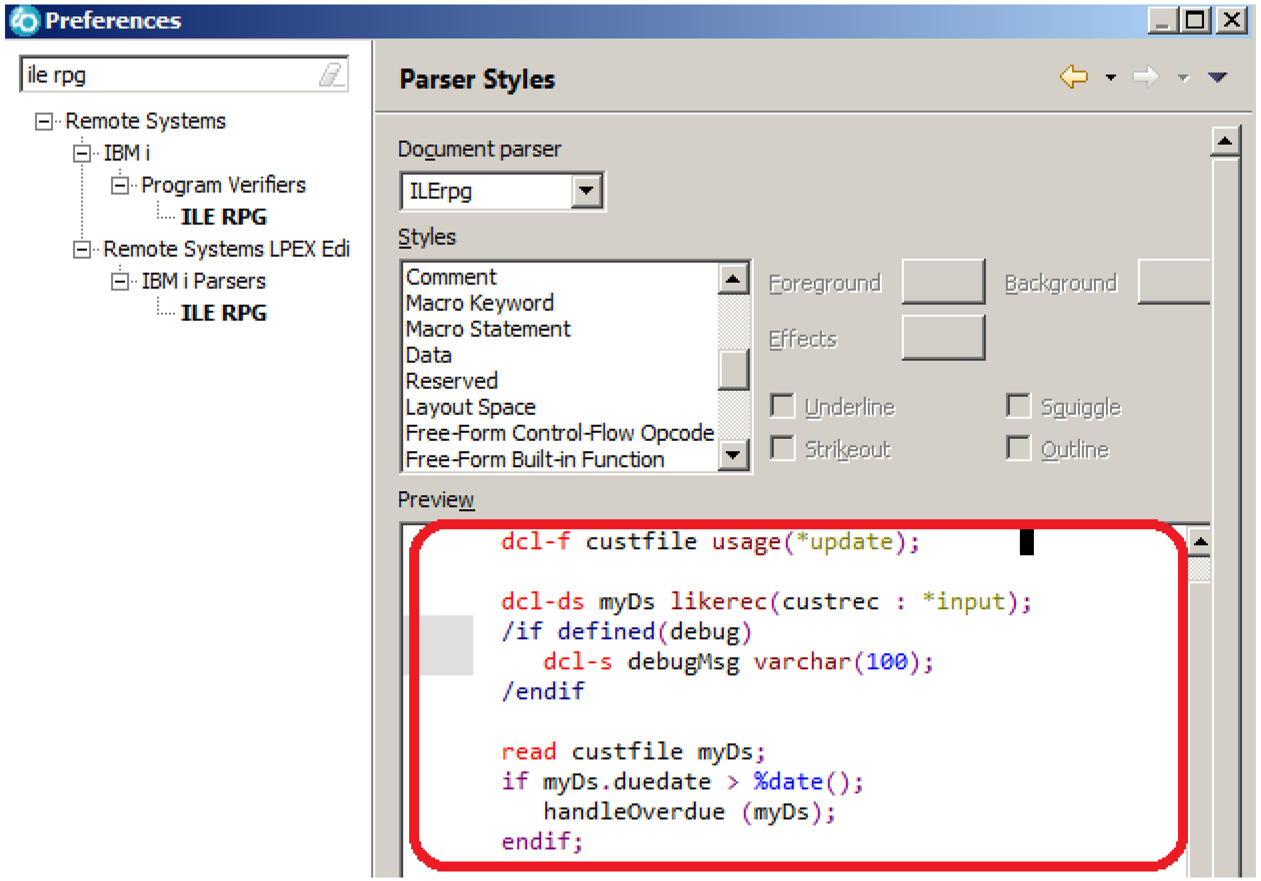
☐ Add signature to changed lines for

Signature

Show Block Nesting

You can change the code to work with

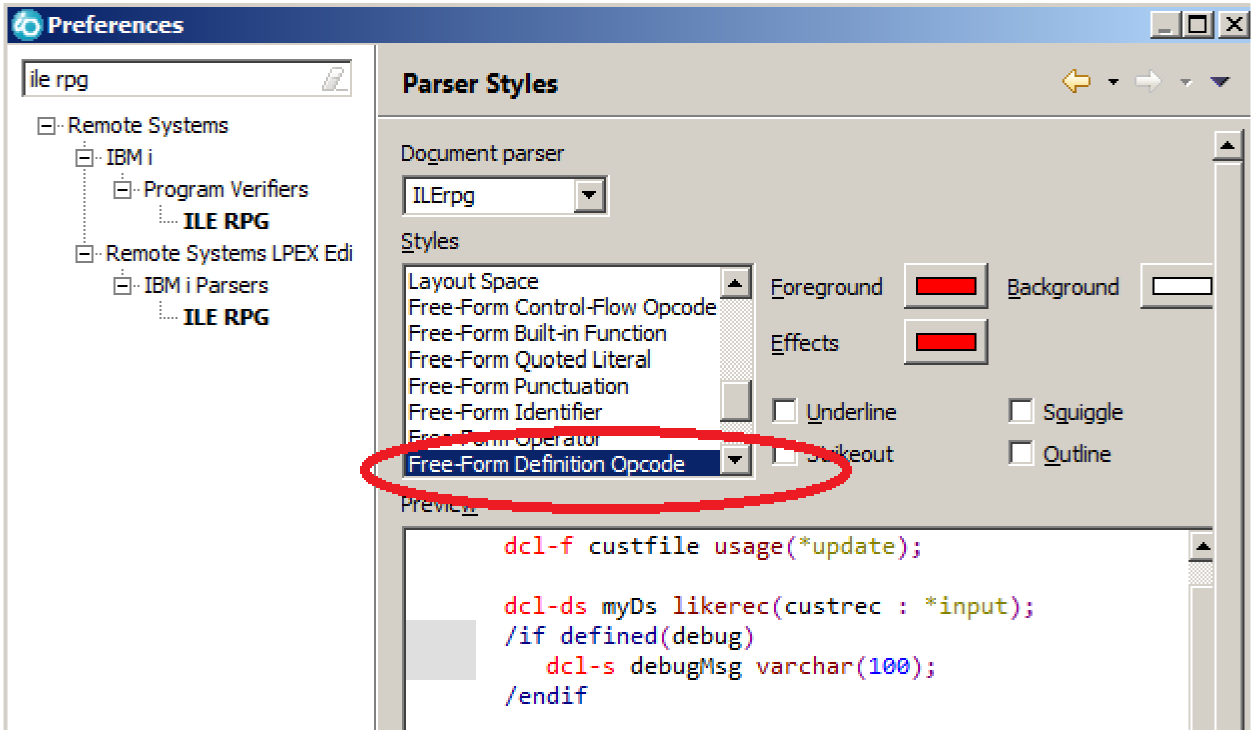
- In the code section, I like to paste in a bit of my own code at the top



```
dcl-f custfile usage(*update);

dcl-ds myDs likerec(custrec : *input);
/if defined(debug)
   dcl-s debugMsg varchar(100);
/endif

read custfile myDs;
if myDs.duedate > %date();
   handleOverdue (myDs);
endif;
```

Choose which style you want to change

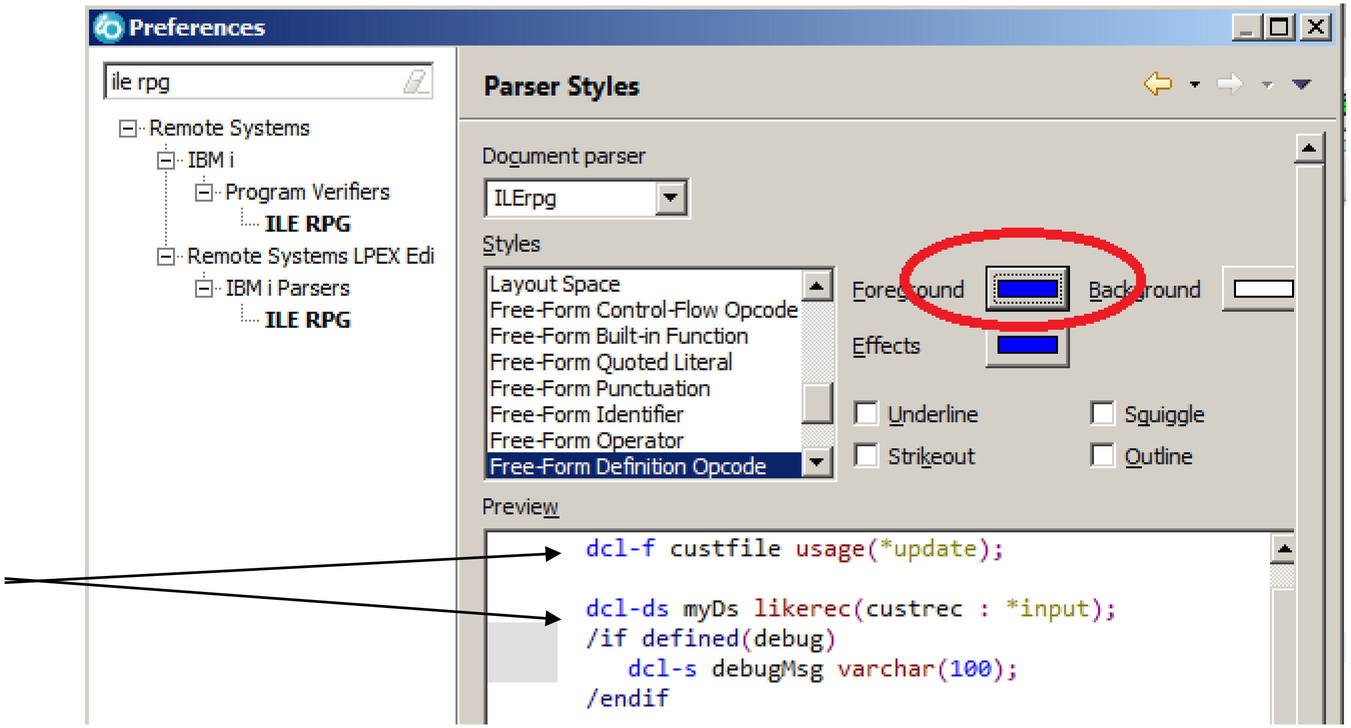
- Then click on the code you want to change the color for
- The top section will automatically position to the relevant style



```
       dcl-f custfile usage(*update);

       dcl-ds myDs likerec(custrec : *input);
       /if defined(debug)
          dcl-s debugMsg varchar(100);
       /endif
```

Customize your colors

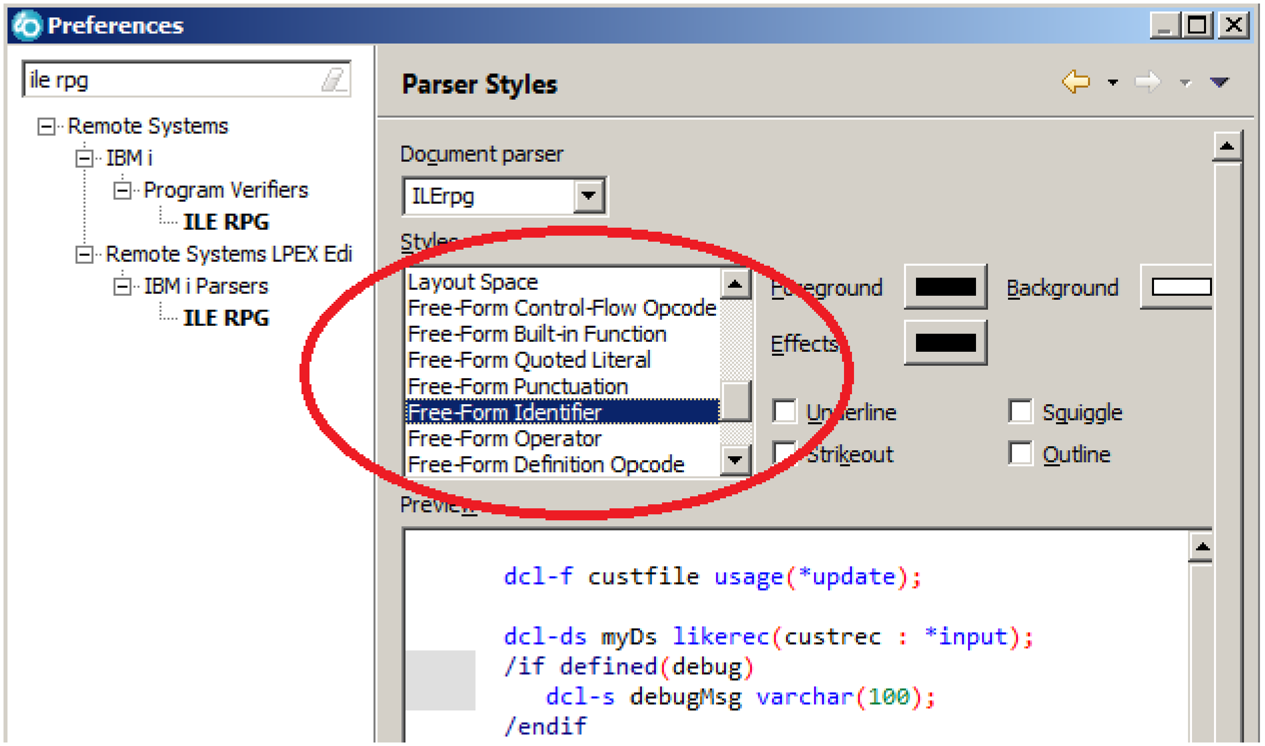
- Choose the color you want
- It will automatically be colored in the code section so you can see the effect it has



```
dcl-f custfile usage(*update);

dcl-ds myDs likerec(custrec : *input);
/if defined(debug)
   dcl-s debugMsg varchar(100);
/endif
```

Another way to choose the style

- For most free-form code, the styles are listed together
- You can select them one-by-one, adjusting the colors

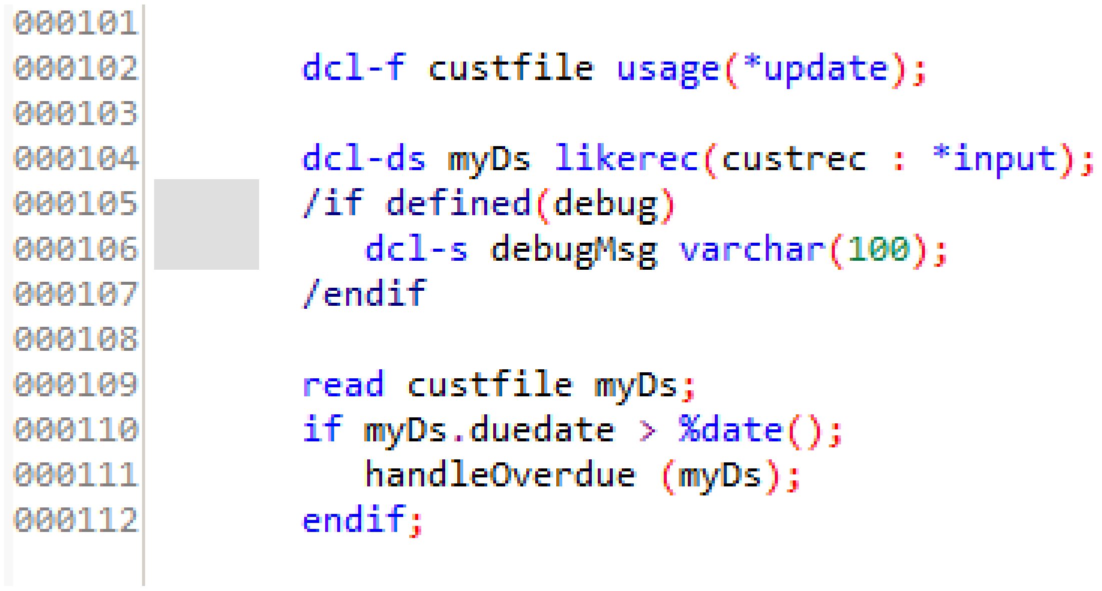


Layout Space
Free-Form Control-Flow Opcode
Free-Form Built-in Function
Free-Form Quoted Literal
Free-Form Punctuation
Free-Form Identifier
Free-Form Operator
Free-Form Definition Opcode

```
       dcl-f custfile usage(*update);

       dcl-ds myDs likerec(custrec : *input);
       /if defined(debug)
          dcl-s debugMsg varchar(100);
       /endif
```

IBM

Here's how I like it



```
dcl-f custfile usage(*update);

dcl-ds myDs likerec(custrec : *input);
/if defined(debug)
   dcl-s debugMsg varchar(100);
/endif

read custfile myDs;
if myDs.duedate > %date();
   handleOverdue (myDs);
endif;
```

Summary

We had two goals when designing the new free-form syntax:

- Easy for non-RPG programmers to learn
- Easy for existing RPG programmers to learn

We have a few years of evidence that we have indeed accomplished those goals!

www.ibm.com/software/rational

Special notices

Special notices (cont.)