Introduction to Mainframe Programming with COBOL

Elaine Duffin

Contents

1 Getting Started

1.1 Signing on to the Mainframe environment

For this workshop, we are using a mainframe that IBM makes available to universities in Europe to use for teaching purposes as part of IBM Z Academic Initiative (The actual mainframe is located in Montpellier in France). To send and receive information from the mainframe, we use a terminal emulator. From the faculty Windows computers go to Campus Applications and find WC3270. For this part of the workbook, you are expected to read and key everything as described.

The terminal emulator connects to the mainframe which is known as ZEUS. Key `dial zos22a` as shown in Figure 1 and press enter.

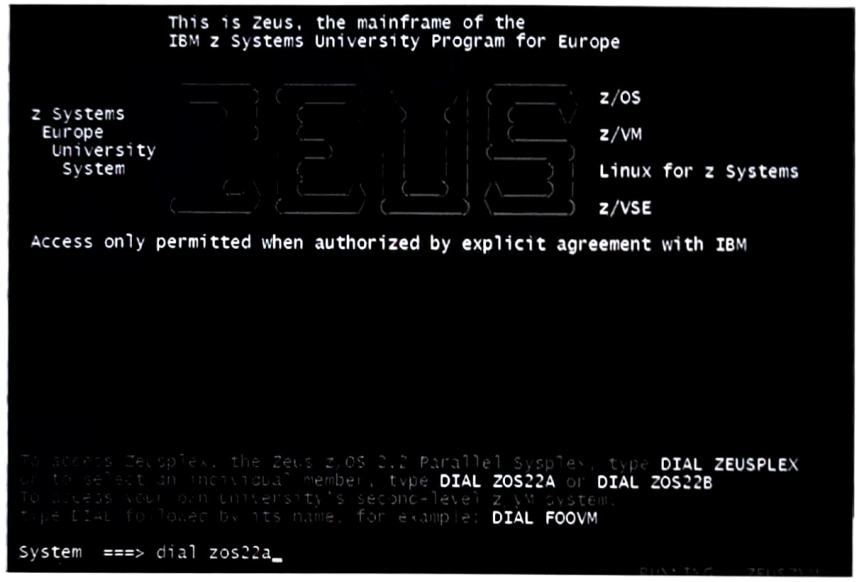

Figure 1: Initial ZEUS screen

You will be given a userid and password to use for this workshop and enter as indicated in Figure 2. (Note you do not see any characters when you key a password).

Figure 2: Signing in

If you press enter, you will get a message telling you that your password has expired as shown in Figure 2. This is deliberate. Choose a new password and type next to **New Password**, it can be up to 8 characters in length and is not case sensitive. Make sure that you remember your userid and password for this workshop as you will need to log off and back on again. In the Figures in this workbook, you will see a userid of MMU0006, please use the one you have been allocated instead.

When you have typed a new password and pressed enter, you will need to confirm that new password by typing it again in the gap as shown in Figure 3.

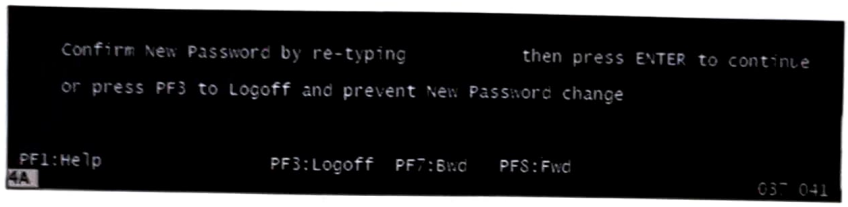

Figure 3: Resetting password

When a new password has been set, you will have the ZEUS menu as shown in Figure 4. To interact with the ZEUS, we use F keys (sometimes also called PF keys) as well as typing. Press F2.

Figure 4: Zeus main menu after password set and signon complete

You will see some messages come up as in Figures 5 and 6. Whenever a message finishes with ***, the system is waiting for you to press enter to continue. Press enter twice.

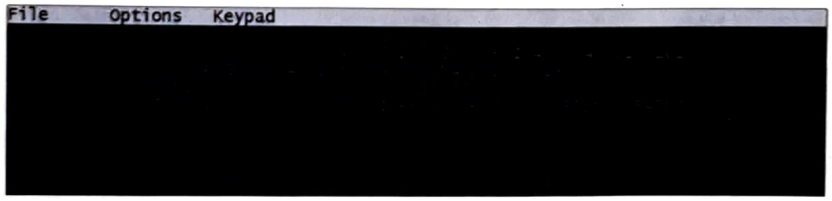

Figure 5: Logon messages

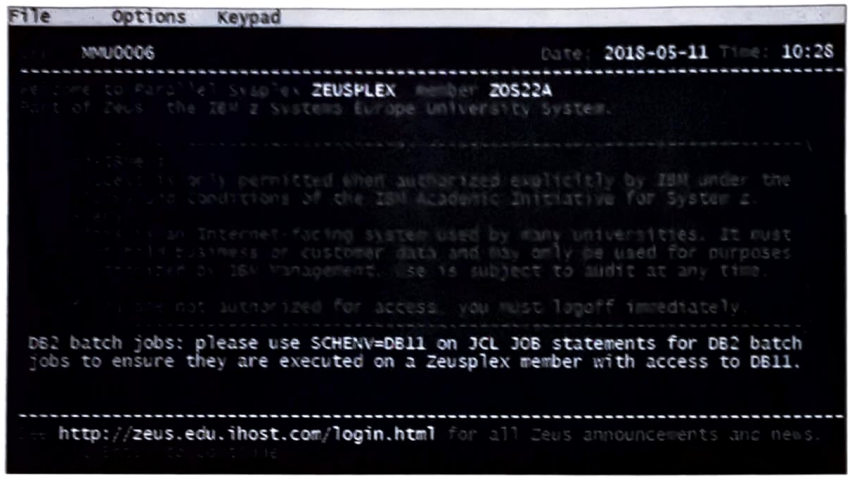

Figure 6: Initial messages about use of Zeus

1.2 ISPF

The home screen for all our interactions with the mainframe is ISPF shown in Figure 7. This gives us a list of options and a command line (after the ===>) where we can type options and other commands.

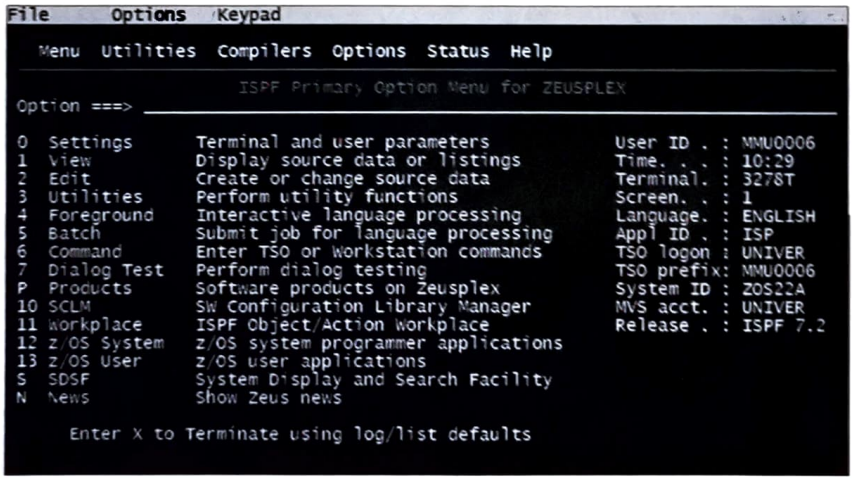

Figure 7: ISPF home page

To choose an option we can type the option and press enter. To exit from any screen, press the F3 key until you get back to this home screen. There are many more options than you will need. Even working full time for several years I only used a small number of the options.

Select option 3 and from the new screen that comes up, select option 4. You should be at the screen in Figure 8. Go back to the home screen by pressing F3 twice.

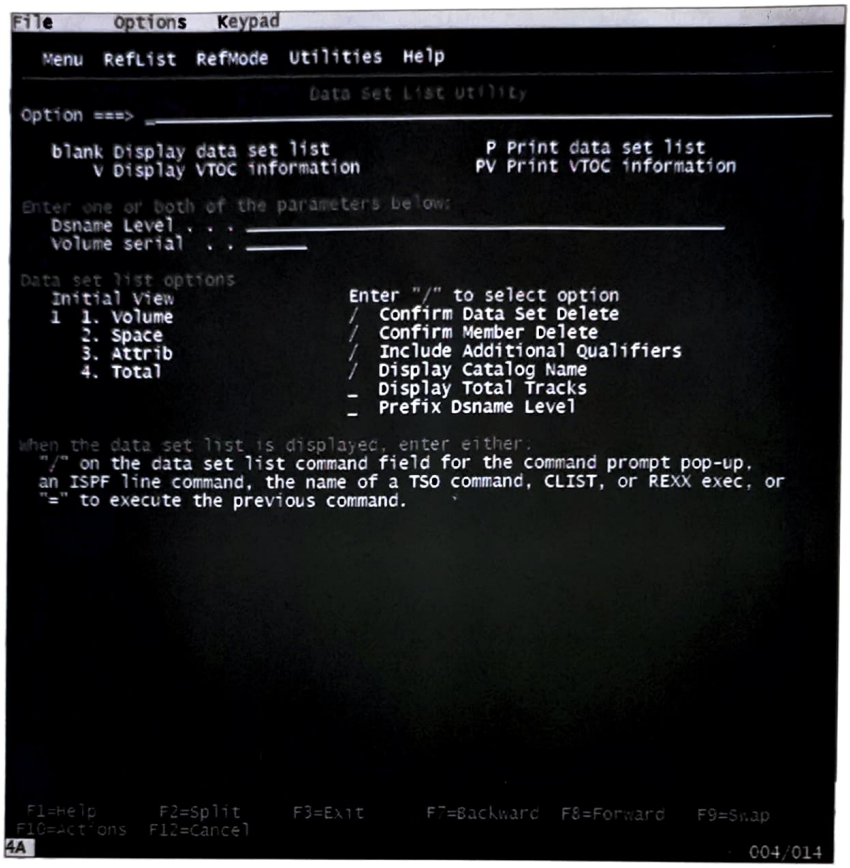

Figure 8: Data Set List Utility, known as 3.4

The screen that you navigated to is used all the time to access data on the mainframe and will be used a lot in this workshop. If you know that you need option 3 followed by option 4, you can just key `3.4` as shown in Figure 9 to get to Figure 8. If you are not on the home screen, you can key `=3.4` to do the same thing.

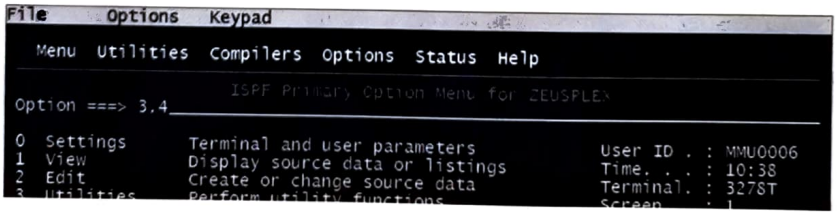

Figure 9: Going to 3.4

1.3 Log off

Before using data, we need to be sure that everyone can log out and back in again.

Use F3 multiple times to get back to the ISPF home screen. Press F3 once more to get to a screen with the word READY and type **logoff** as shown in Figure 10. You will need to press enter twice.

Figure 10: Logging off

You should be back at the screen shown in Figure 4. This time, press F12 to complete the log off process.

1.4 Viewing data

Following the steps from above, log back in (it will need the password that you set earlier and you can ignore the New Password entry) and navigate to 3.4.

In mainframe terminology, data is stored in datasets (also written as data sets) rather than files and folders. Any datasets that start with UNIMMU can be edited by tutors and viewed by students whose userids start with MMU. To see the UNIMMU datasets, key `UNIMMU` on the line next to `Dsname Level` (and press enter) to give a screen looking like Figure 11.

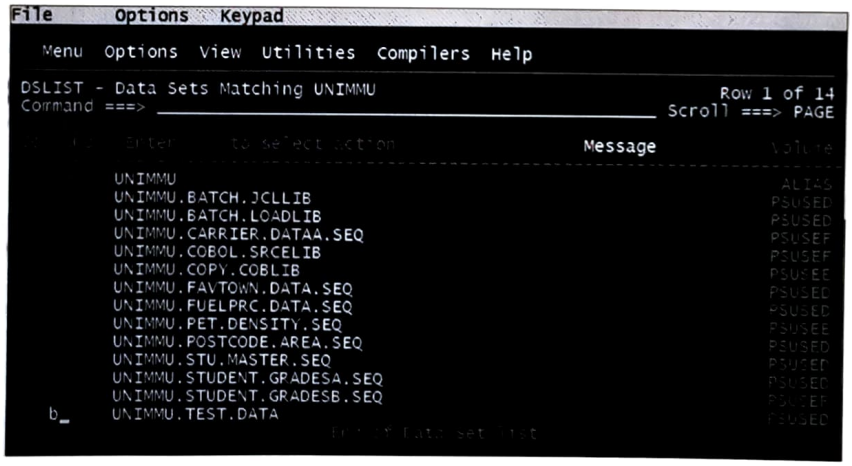

Figure 11: List of UNIMMU datasets

Each of the items listed starting with UNIMMU is a dataset, some of these are known as *partitioned datasets* and have members inside them (like files inside folders). As well as issuing commands on the command.line as we have done so far, you can often issue commands on selected lines. To browse the `UNIMMU.TEST.DATA` dataset (which has members inside it), type **b** to the left of the dataset name (as shown in Figure 11) and press enter.

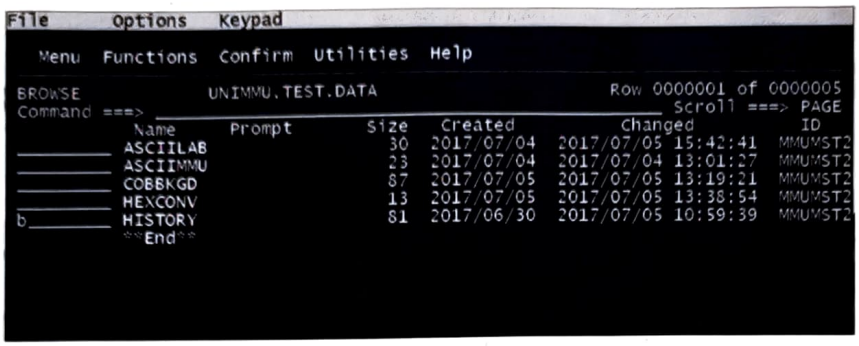

Figure 12: Members of `UNIMMU.TEST.DATA`

We now see a list of the members within `UNIMMU.TEST.DATA` and can use **b** again to browse individual ones and see the actual contents. Browse the member named `HISTORY`. When we are in a dataset we see the dataset name at the top with the member name in brackets.

To scroll up and down in the dataset, use F7 and F8 keys. To exit, use F3. These are keys that you will be using many times, the reminder of the use is at the bottom of the screen as shown in Figure 13.

Figure 13: Reminder for use of F keys

Exercise 1

Go back to the ISPF home screen, navigate back to the same dataset member and this time use **v** for view instead of **b** for browse. What difference does it make to the display?

2 Creating and editing data

2.1 Your personal datasets

As you don't have access to change data stored under UNIMMU, you need your own datasets where you can create and edit members (files). On a mainframe, you can always edit datasets starting with your userid. To save time and potential issues, you will run a job to allocate some datasets to you. We will come back to jobs later, but at the moment just follow these instructions.

Go back to the list of UNIMMU datasets. This time view the members in `UNIMMU.BATCH.JCLLIB`. Type **sub** next to the member `PDSSETUP` as in Figure 14 and press enter.

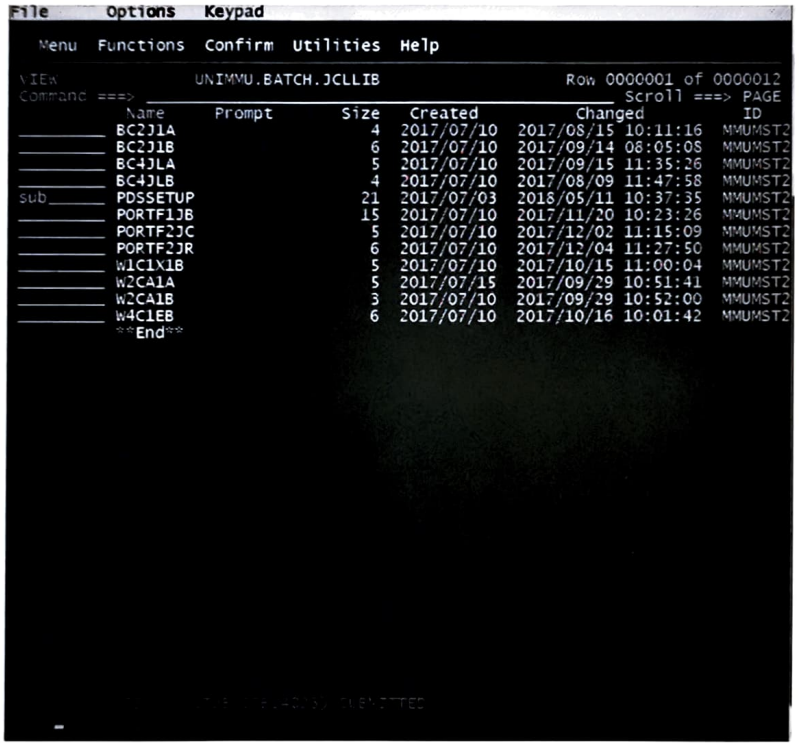

Figure 14: Submit a job

You will get a message stating that the job has been submitted, as in Figure 14. When the job has finished, the message in Figure 15 indicates (by `MAXCC=0000`) that the job has completed successfully.

Go back to the 3.4 screen and instead of typing UNIMMU to show the UNIMMU datasets, use the **MMU*nnnn*** userid that you have been allocated to see a list of your datasets. Note

Figure 15: Job completed successfullly

that from now on, I will use **MMU*nnnn*** to refer to your userid with ***nnnn*** representing the four numerals.

List the members of your **MMU*nnnn*.TEST.DATA** dataset (use **b** or **v** as we have done previously for UNIMMU). Type **e** next to **COBBKGD** to open this member in edit mode so that you can make and save changes. This contains a copy of the **HISTORY** data that you viewed earlier. This verison contains some errors for you to correct. Remember to use F7 and F8 keys to scroll up and down respectively.

2.2 Saving and cancelling changes

To save any changes, key **save** on the command line and press enter or press F3 to save and exit the document.

To cancel changes made since the last save, type **can** on the command line or press F12 to exit without saving.

2.3 Insert and overtype mode

The default mode is to overtype text.

Use the insert key to change to insert mode, and the text will move along if there is room. To change back to overtyping, press the insert key again. When you are in insert mode, you will see an **I** at the bottom of your terminal window.

In insert mode, if you type somewhere with no room to move the text along, you will not be able to type and will get the error **X Overflow** at the bottom of your screen. This happens if you try to type on the command line in insert mode.

2.4 Line commands

Additional functions are available by keying commands on the line numbers on the left.

- Insert line(s)

  To insert a line immediately after a given line, key **i** over the line number (and press enter). When you have inserted a line and type on it (even if you just type a space)

and press enter, you will be given another blank line. When you have finished typing, press enter again to get rid of any unused blank lines.

- Delete line(s)

  To delete a single line, type a **d** over its line number. To delete a block of a set number of **n** lines, type **dn** over the first line required. For example to delete three lines, type **d3**. Rather than counting lines, you can indicate the start and end of a block that you want to delete by keying **dd** on the line numbers at the top and bottom of the block.

- Move line(s)

  To move a line, we use an **m** on the line number. Similar to delete, we can use **m** with a number to indicate a number of lines to move and **mm** at the start and end of a block to move a complete block.

  If you have selected some lines to move and press enter, you will get a message near the top right stating "MOVE/COPY is pending" as shown in Figure 16.

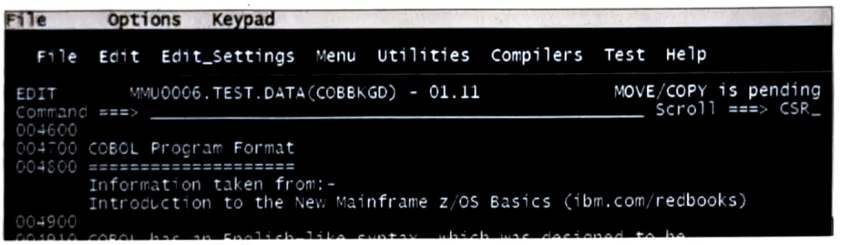

Figure 16: Moving a block of text

We need to indicate where to move the selected lines to by using the line command **a** if we want to place the content after a given line, or **b** for before a given line.

Exercise 2

Use the commands described above to make and save the following changes to the document.

- Line 000700 states "Grace Hopper was are pioneer", change the "are" to "a".
- Line 001400 includes "it was possible", change it to read "it was not possible".
- Insert a line immediately after the one that says IDENTIFICATION DIVISION and add the following text: "This must contain a program-id."
- Delete one of the two lines that reference the COBOL page on wikipedia.
- The two lines starting "COBOL has an English-like syntax" are in three places in the text. Use a method of deleting a block of text to reomove the one before the wikipedia link and the one at the bottom of the text.
- Move the two lines that start "Information taken from:-" to the end of the document.

3 Starting programming in COBOL

3.1 COBOL basic layout rules

COBOL has specific headings that must be used in all programs and also rules as to where the code can be placed.

The four COBOL divisions

- IDENTIFICATION DIVISION identifies the program
- ENVIRONMENT DIVISION describes aspects of the program that depend on the computing environment (e.g. files). We will not use this division in this workshop.
- DATA DIVISION defines the format of your data
- PROCEDURE DIVISION contains the instructions for the program to carry out. The program will run the instructions sequentially (in order one after the other) unless there is some code that changes the order.

Column rules

- Columns 1 to 6 are left blank
- Column 7 can contain a * to indicate a comment line, otherwise is blank.
- Columns 8 to 11 are used to define divisions and sections and some data storage. This is known as **area A**.
- Columns 12 to 72 are used for all other statements. This is known as **area B**.
- Columns 73 to 80 are not used

3.2 Hello World program

COBOL commands used:

- `DISPLAY`: allows us to output information
- `STOP RUN`: tells the system that our program has finished.

Navigate to the list of members in your `MMU`***`nnnn`***`.COBOL.SRCELIB` dataset. This is the dataset where you are going to create your COBOL programs.

Edit the member `HELLO` (use **e** next to it to open in edit mode). In the editor, type **cols** on the command line to get column numbering to help with alignment of the code. Notice that the provided code is already indented correctly.

Add a program-id and a **DISPLAY** statement as shown the code below. Make sure that the word **DISPLAY** is aligned with **STOP** (you will probably need to key some spaces). You can key in lower case and it will automatically change to upper case.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
PROCEDURE DIVISION.
     DISPLAY 'HELLO WORLD'
     STOP RUN.
```

3.3 Compile your program

You have created a COBOL program, but you cannot run a COBOL program directly, the readable code that you work with is known as source code and you need to *compile* it to create a version that you can run (or execute).

During this session, we are running programs in batch mode, they run in the background and we can view them when they have finished. In mainframe terminology, a unit of work running in the background is called a *JOB*. We need to use instructions in a specific format to control the running of a JOB. This format is Job Control Language (JCL). This workshop does not cover JCL but we need to be able to use enough to compile and run programs.

Navigate to your **MMU*nnnn*.BATCH.JCLLIB** dataset and list the members. The member **COBCOMP** will be used to compile your programs.

Edit the **COBCOMP** member and change the **MMU0000C** on the first line to your userid followed by a **C**. This is the jobname.

```
//MMU0006C JOB  UNIVER,CLASS=A,MSGCLASS=H,NOTIFY=&SYSUID
//STEP01   EXEC IGYWCL,LNGPRFX=IGY
//SYSIN    DD   DSN=&SYSUID..COBOL.SRCELIB(HELLO),DISP=SHR
//LKED.SYSLMOD DD DSN=&SYSUID..BATCH.LOADLIB(HELLO),DISP=OLD
```

The JCL for the compile job is shown above. The line that contains EXEC tells the mainframe what program/procedure to run. The compile job runs IGYWCL which is supplied by IBM for compiling COBOL programs.

The line that starts with SYSIN tells it where to look for the source code to compile. Your userid (**MMU*nnnn***) will be used in place of **&SYSUID**. As we saw earlier a dataset with member inside has the member name in brackets, so it will look for the **HELLO** member in **MMU*nnnn*.COBOL.SRCELIB**. The following line tells it to put the compiled and linked program that will be ready to run in a **HELLO** member in **MMU*nnnn*.BATCH.LOADLIB**. You will need to edit this JCL to change the member name to compile other programs.

Run your job by typing **sub** (or submit) from the command line when editing the JCL or on the line next to the member name when listing your dataset members.

If your program has compiled correctly, you will get a message including `MAXCC=0000`. If something has gone wrong, you will get a value above zero.

Whether you got `MAXCC=0000` or not, we now need to look at how to view the messages and output from jobs. To do this, we will use the facility SDSF.

Firstly, it is a good idea now to have more than one screen at a time. Move your cursor to the top of the display (where it has the words Menu etc) and press F2. You now have two displays and can use F9 to swap between them.

From a command line, enter **=S** to go to SDSF, now enter **ST** (Status of jobs). You will see a list of jobs on this mainframe. To see only jobs that you have run, type `owner MMU`***nnnn*** after the `===>` command prompt (and press enter). The job that is white (rather than blue) is you being currently signed on and so has no completion code.

You can list jobs in descending order so that the most recent one is at the top of the list. To do this, put your cursor on the word **Pos** (for position) and press enter to put them in ascending order. Repeat the process to reverse the order and make it descending.

In SDSF, you use the F7 and F8 keys to scroll up and down (as in ISPF). In both environments, we use F10 and F11 to scroll left and right. If you scroll right, you will find a column heading that says **Max-RC** and has the completion code of the job you just ran.

To look at what is in the job, type a **?** next to the job name as shown in Figure 17.

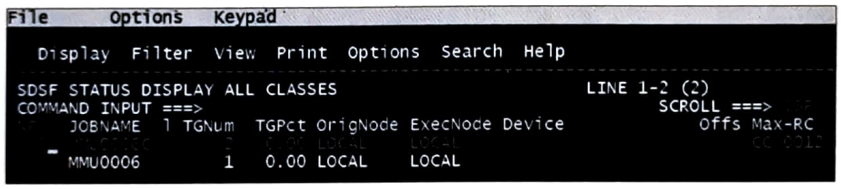

Figure 17: Split a job to see its contents

If your job has completed successfully, you will have a step that says `LKED`. In Figure 18, I am looking at a compile that failed and I use **s** to select the COBOL line to see the error messages.

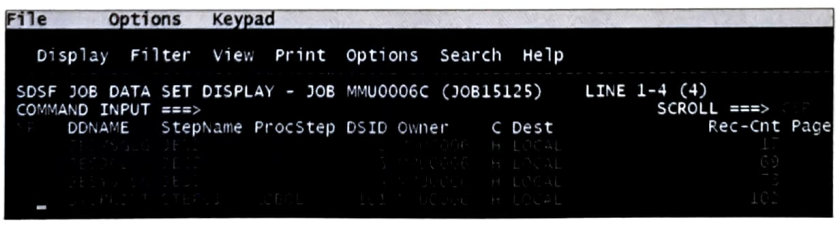

Figure 18: Selecting the COBOL step

When I have selected the COBOL step, I can scroll to the bottom to see a summary of the error messages. In Figure 19, I have one severe and one warning message. Just above the summary, I see each of the messages described in words. Above that, I find the program source code with the messages under the problem lines of code. In this case, I haven't put a space after PROGRAM-ID. and I have forgotten to put quotes round the word HELLO for the message that I wanted to display. These messsages are very wordy, so please ask if you need help.

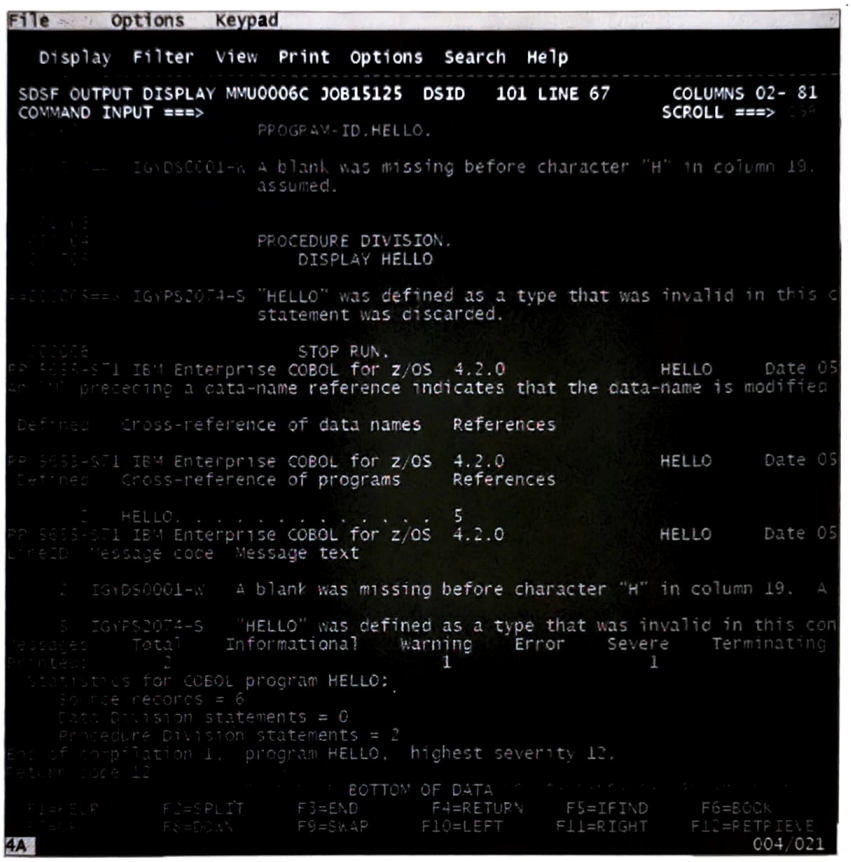

Figure 19: Reading the errors from a failed compile

Having the information from the failed compile, I can use F9 to swap screen to go back to the source code to correct the errors. When I think that I have corrected the errors, I can submit the compile job in the same way as before or alternatively I can do so from SDSF. In SDSF, you can select the jcl by typing **sj** next to a job as shown in Figure 20.

When you have selected the JCL using **sj** you can make any changes if required (the changes will not be saved back to your JCL library) and submit the job by typing **sub** as in Figure 21.

Figure 20: Select the JCL from a job in SDSF

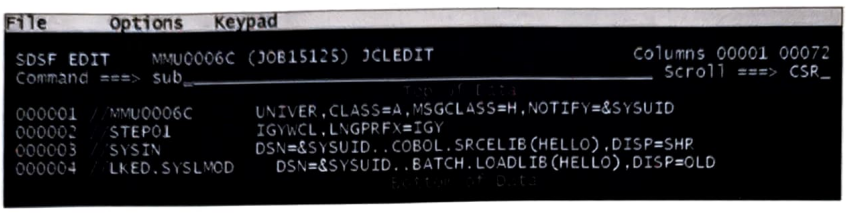

Figure 21: Submit the selected JCL

3.4 Run your program

When you have a successful compile, you can run your program. This time you need to edit the the **COBRUN** member member in your **MMU*nnnn*.BATCH.JCLLIB**.

Again change the jobname on the first line to use your userid. In the JCL (below) we see that this time the EXEC statement tells it to run a program called **HELLO** and to look in your **BATCH.LOADLIB** for the program.

```
//MMU0006R JOB  UNIVER,CLASS=A,MSGCLASS=H,NOTIFY=&SYSUID
//STEP01   EXEC PGM=HELLO
//STEPLIB  DD   DSN=&SYSUID..BATCH.LOADLIB,DISP=SHR
//SYSOUT   DD   SYSOUT=*
//SYSIN    DD   *
```

Submit the job, hopefully you get **MAXCC=0000**, if not ask one of us to look at it with you.

Go back to SDSF (it's easier if you keep SDSF so that you can swap screens with F9). Split the job with a **?**. You should have **SYSOUT** as one of the outputs, select this (use **s**) to see your program output.

Exercise 3

Amend your program to display multiple messages of your choice. Make sure that you compile and run your amended program and check the output in SDSF.

4 Using data in COBOL

4.1 Working-storage

In COBOL we use the WORKING-STORAGE section to store data that we want to manipulate while the program is running (called variables in most other programming languages).

We have to choose an appropriate name for some storage (called a field), state what type and how many characters of data it will need to hold and we can give it an initial value.

When naming the storage, we have to stick to the following rules.

- 1 to 30 characters
- Letters, digits, hyphens (-) only
- No embedded blanks
- At least one alphabetic character
- May not begin or end with hyphen
- May not be COBOL reserved word

To indicate the type and length of data we use a PICTURE clause (shortened to `PIC`). In a `PIC` clause storage for numeric data is indicated by a **9** and alphanumeric data (any characters) by an **X**. In both cases a number in brackets shows the number of characters.

To assign an initial value to a field, we use the `VALUE` clause.

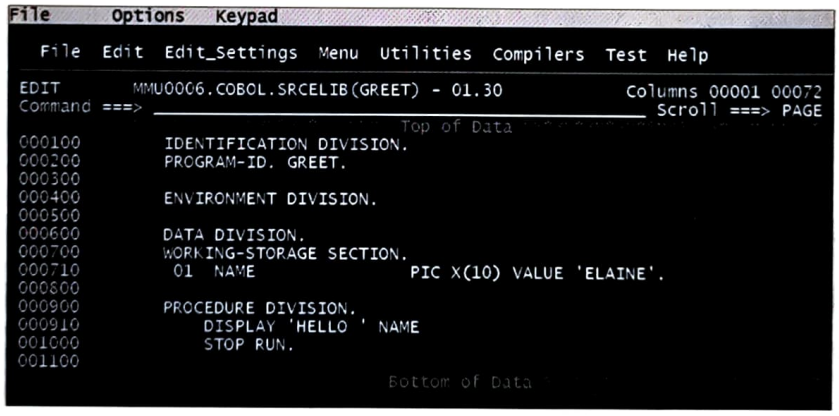

Figure 22: Using working-storage

Figure 22 shows a working-storage field called `NAME` that can hold 10 alphanumeric characters (`PIC X(10)`) and has the initial value `ELAINE`.

Note that all working-storage fields need a level number, in Figure 22 the level number is `01` and is in area A (columns 8 to 11). We will come back to level numbers shortly.

Note that when giving an initial value to an alphanumeric field, we use quotes. The **DISPLAY** statement in the procedure division uses text in quotes and the working-storage field. Note that we don't use quotes when specifying that the working-storage field should be used. Although COBOL programs are written in upper case, if you want to use lower case values, use `caps off` on the command line. (`caps on` will make any new typing upper case).

Exercise 4

In your **MMU*nnnn*.COBOL.SRCELIB** dataset, you will find the beginnings of a program in the member **GREET**. Add some working-storage for the title of a film and give it an initial value. Note that you will need to choose a suitable name for the storage and specify an appropriate length (give it a level number of `01`). (If you don't know any films, pick one from the IMDB website) Display a message that uses the film title from the working-storage. Compile and then run your program (you will need to change the JCL to use the new member name).

4.2 Receiving data to the program

COBOL programs can use data that comes from files and databases, but for this workshop we will pass data to the program through the JCL of the run job. The JCL line `//SYSIN DD *` tells the job that the input will follow immediately, so we can add additional line(s) to pass input to the program. In Figure 23, I have added a name as input.

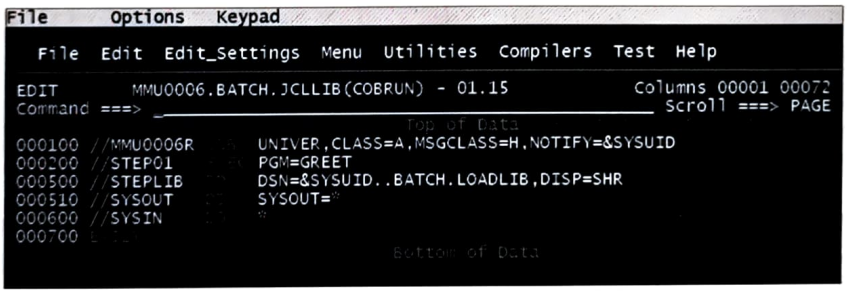

Figure 23: Passing data to a program from the JCL

In a COBOL program, the keyword **ACCEPT** will receive data from the JCL and store it in a specified field. In Figure 24 the first line of the procedure division causes the storage **NAME** to be filled from the JCL (shown in Figure 23). If the field **NAME** already contained any value, this would be overwritten at this time. In the example in Figure 24, I have removed the **VALUE** clause from the working-storage definition. The **DISPLAY** statement is as in the previous version of the program.

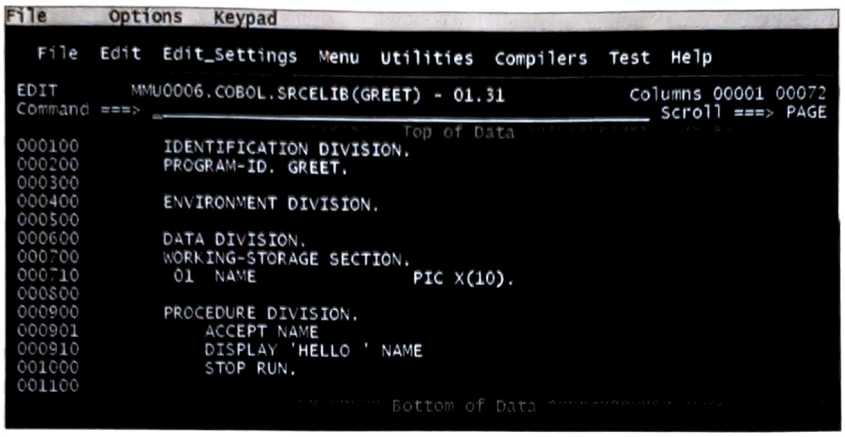

Figure 24: Receiving data using the ACCEPT statement

Exercise 5

Amend your program so that the film name is passed to the program from the JCL rather than being set as an initial value in the program. Compile and run your program. Remember to change your run job to include the line of information (the line command **i** will insert a line).

Experiment Add more than one line of information in the JCL and amend your program to read in multiple lines each time overwriting the same working storage. What happens if you try to read in more lines than are available?

4.3 Structured data

In our working storage so far we have a level number of `01` for all our fields. Now we will use an example to look at how level numbers can be used to show structure in data.

COBOL is designed to deal with business information that is stored in fixed width fields. That is, each data item takes up a fixed amount of space.

Suppose I have an employee record that contains an employee id, name (that consists of a title, surname and initials), department, start year and grade. The record might look as in Figure 25 where I have started to give field-names to the different parts. In this example, the employee id (`EMP-ID`) is the first 8 characters in the record. The employee title (`EMP-TITLE`) always takes up 4 characters and has spaces if the specific title is shorter than that.

When I look at the record in Figure 25, I can consider it at different levels, the whole record as one block (`EMPLOYEE-DETAILS`); the block that corresponds to the name (`EMP-NAME`) which consists of title, surname and initials; or each individual field separately. In COBOL, we use level numbers to indicate this type of structure. A level number of `01` is used for the

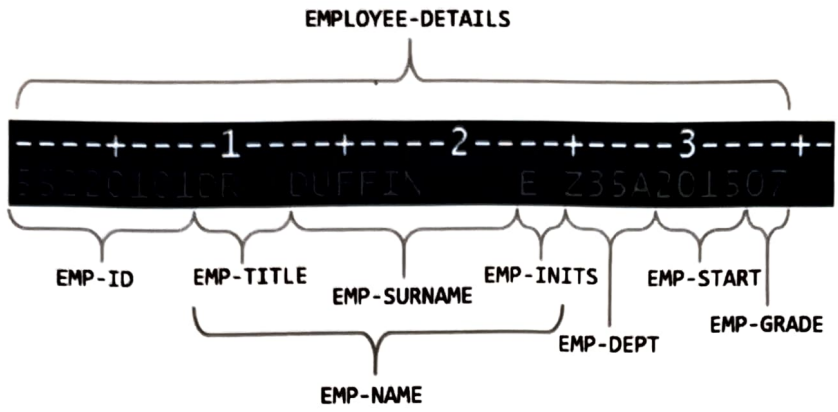

Figure 25: Employee details record

entire record. Parts within a record have higher level numbers as shown in the COBOL working-storage below.

```
01 EMPLOYEE-DETAILS.
  03 EMP-ID                PIC X(8).
  03 EMP-NAME.
    05 EMP-TITLE           PIC X(4).
    05 EMP-SURNAME         PIC X(10).
    05 EMP-INITS           PIC X(2).
  03 EMP-DEPT              PIC X(4).
  03 EMP-START             PIC 9(4).
  03 EMP-GRADE             PIC 9(2).
```

In this working-storage, **EMP-TITLE**, **EMP-SURNAME** and **EMP-INITS** all have the same level number as each other and are placed immediately after **EMP-NAME** which has a lower level number. This shows that **EMP-NAME** can be considered on its own or as composed of separate fields. Fields that are composed of separate fields are known as group fields and do not have a **PIC** clause. Fields that are not sub-divided are known as elementary fields and have a **PIC** clause. It is common (but not required) not to use consecutive numbers for level numbers and use 01, 03, 05, 07 etc. or 01, 05, 10, 15 etc. It is good practice to indent the storage to show the relationships as shown.

Remember that a **PIC** clause with a 9 is for numeric data. Numeric data will have leading zeros (as in **EMP-GRADE** in Figure 25) so that all characters are numeric.

4.4 Changing the value stored

We have seen that we can use the word **VALUE** to store an initial value in a field and **ACCEPT** to get data from the JCL into a field.

To change the value in the procedure division, we can use MOVE. If we assume that NAME1, NAME2 and NAME3 are all defined in working-storage for alphanumeric values, the code below will overwrite the contents of NAME2 with the contents of NAME1 (leaving NAME1 unchanged) and will store EDGAR in NAME3 (overwriting whatever was previously in NAME3).

```
MOVE NAME1 TO NAME2

MOVE 'EDGAR' TO NAME3
```

Exercise 6

Create structured storage to hold information about a film including film title, director, year of release and running length in minutes. Edit your JCL to hold the details under SYSIN. Receive the data from the JCL.

Display the film details. Compile and run your program and check the output in SDSF.

Use MOVE statements to swap round the film title and director name and then display the details again. Hint - you will need at least one additional working-storage field to do so. Compile and run your program again.

Experiment What happens if you move data to a field that is too big or too small for the field that you are moving to? Does the same happen with numeric fields? Make some changes to your program using MOVE and DISPLAY to try to discover the rules.

4.5 Calculations in COBOL

4.5.1 ADD, MULTIPLY etc

Suppose we have working storage fields NUM1 and NUM2 that contain numeric values then we can write the following:

```
ADD NUM1 TO NUM2
```

This will add whatever is stored in NUM1 to the value stored in NUM2 (and leave NUM1 unchanged).

We can also add specified values to that stored in a field as follows:

```
ADD 10 TO NUM2
```

If we want to add NUM1 and NUM2 but leave their contents unchanged and store the result in another field NUM3, we can use

```
ADD NUM1 TO NUM2 GIVING NUM3
```

There are similar versions for SUBTRACT and MULTIPLY, using FROM for SUBTRACT and BY for MULTIPLY.

When using DIVIDE, you can optionally store the REMAINDER as follows.

```
DIVIDE NUM1 BY NUM2 GIVING NUM3 REMAINDER NUM4
```

4.5.2 COMPUTE

As well as using the words for individual operations, we can also do calculations using mathematical symbols.

To do this we can use

```
COMPUTE NUM3 = NUM1 + NUM2
```

If you are used to other programming languages, this will be very familiar (apart from the word COMPUTE at the beginning). The values stored in NUM1 and NUM2 will be added together and the result stored in NUM3 leaving the values stored in NUM1 and NUM2 as they were. If you have multiple calculations to do, it is better to combine them into a single COMPUTE statement than to use a sequence of separate calculations using individual operators.

Exercise 7

Amend your program that displays details of a film. You should calculate how many years it is since the film was released and the number of hours that the film runs for and display the information.

4.6 Creating a new dataset member from an existing one

In order to do the next exercise, you will need a new member in your MMU*nnnn*.COBOL.SRCELIB dataset so we will have to cover how to do that in ISPF. Two methods to do this are described below. In both cases, you can use **ref** on the command line to refresh your list of datasets to see the change.

Copying a dataset member - Method 1

In Figure 26, I am in a member that I want to copy and use `create calcs` on the command line to create a member called CALCS in the same dataset as I am already in (in the Figure, it's MMU*nnnn*.COBOL.SRCELIB). I have to tell it which lines of the file to copy, in this case it was all of it so I typed **c99** on the first line number (if I specify more lines than there are, it will copy from there to the end of the data).

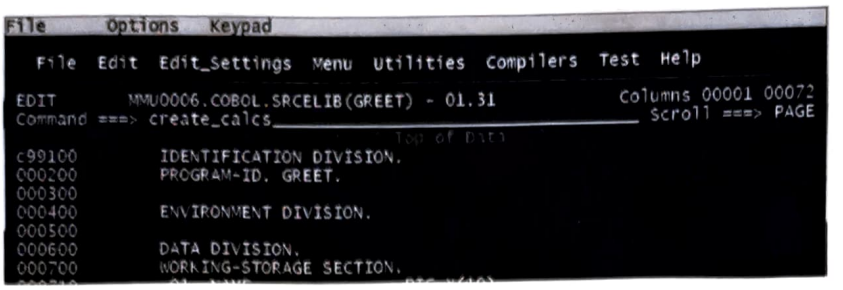

Figure 26: Creating a new member from an existing one

Copying a dataset member - Method 2

In this method, I can copy a complete member of a dataset by typing a **c** next to the member name. This will give me the screen shown in Figure 28. Here I can specify where I want to copy the member to and a new member name. In this case, I have put the userid for Project, Cobol for Group and srcelib for Type (that gives the structure of the dataset name). I then have to supply a new member name as it's the same dataset as I am copying from.

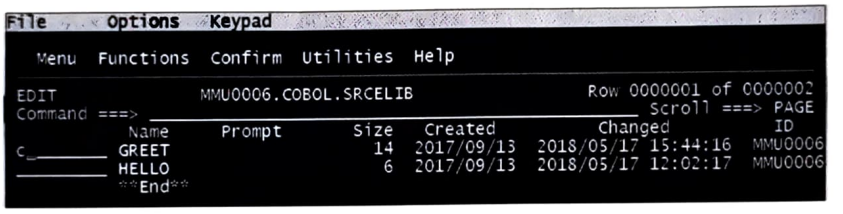

Figure 27: Indicating a member to copy

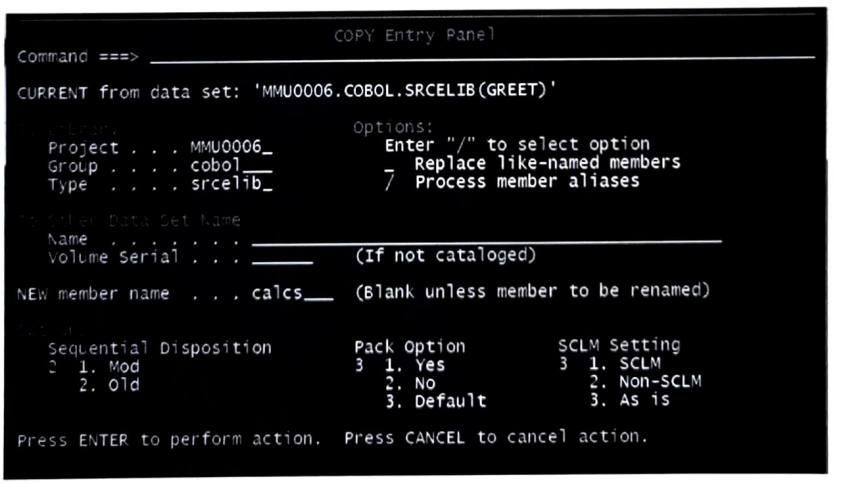

Figure 28: Specifying where to copy to

Exercise 8

Create a new program that takes in two 3-digit numbers from the JCL and then displays the numbers followed by the result of adding, subtracting and multiplying.

Note that you will need to use leading zeros to specify numbers under 100, so the JCL might look like that in Figure 29.

The output might look like that in Figure 30. Run your program several times with different values passed from the JCL.

Note - you have not been shown how to handle negative numbers or decimals.

Experiment What happens if the result is too large for the working storage field that you have defined to store the result of the calculation (for example when multiplying)? Run your program with values in the JCL that would cause this problem and see if you can explain the result.

Add division to your program and display both the result and the remainder.

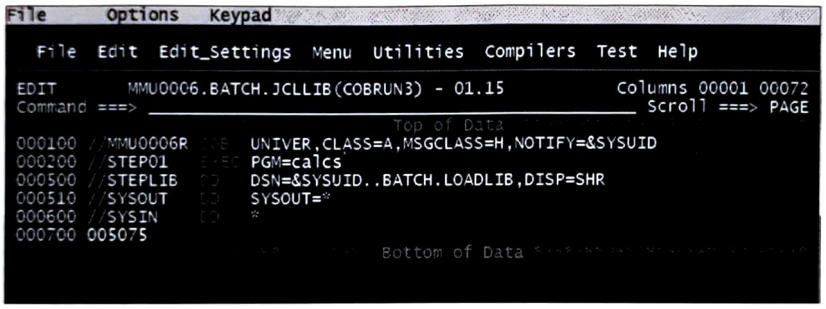

Figure 29: JCL for Calculations program

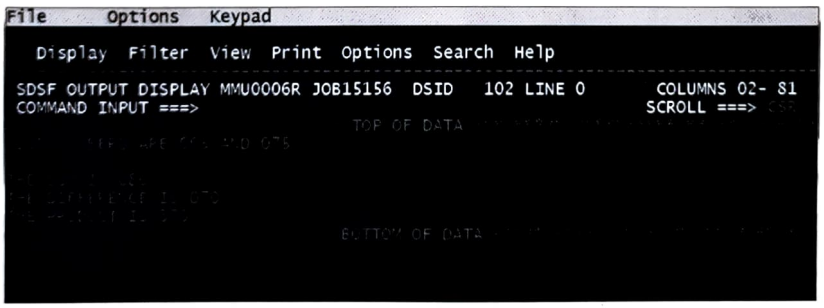

Figure 30: Calculations program example output

5 Conditions and Loops

For those of you who are familiar with programming in other languages, you might want to take things further and use programming constructs of conditions and loops that you already use elsewhere.

5.1 IF

The use of `IF` and `ELSE` in COBOL will be very familiar to those who program in other languages. In COBOL, conditions can be written in words or using symbols. To illustrate this I'll use an example in which I assume that a working-storage field `EMP-START` contains numeric data. `IF` and `ELSE` can be used as follows, showing that COBOL was developed to be readable.

```
IF EMP-START IS GREATER THAN OR EQUAL TO 2016 THEN
  DISPLAY 'Recent starter'
ELSE
  DISPLAY 'Experienced in this workplace'
END-IF
```

The same code could be written in the following way which may be more familiar to experienced programmers.

```
IF EMP-START >= 2016
  DISPLAY 'Recent starter'
ELSE
  DISPLAY 'Experienced in this workplace'
END-IF
```

Note that `END-IF` is used as a scope terminator, indentation is recommended but not necessary.

The relational operators that are available in COBOL are shown in Table 5.1 in both long and short format. For programmers who are used to a double equals sign to test for equality, note that a single equals sign is used in COBOL. Remember that COBOL does not use the equals sign for assignment, except in combination with COMPUTE.

Exercise 9

Amend your calculations program to also display the higher of the two numbers. Make sure that you test your program with both situations of the first and second number being higher, by running it multiple times with different JCL.

Using words	Using symbols
IS GREATER THAN	>
IS NOT GREATER THAN	NOT >
IS LESS THAN	<
IS NOT LESS THAN	NOT <
IS EQUAL TO	=
IS NOT EQUAL TO	NOT =
IS GREATER THAN OR EQUAL TO	>=
IS LESS THAN OR EQUAL TO	<=

Table 1: Relational operators in COBOL (Table adapted from Enterprise COBOL for z/OS Language Reference)

5.2 Loops

For this workshop, I will illustrate just one means of creating a loop in COBOL, using `PERFORM UNTIL`. I'll use the example of displaying all the numbers from 1 to 10 (inclusive) to illustrate a loop. Assume that `NUM1` is a numeric working-storage field that can hold number of at least 2 digits and has been given an initial value of 1.

In the example, the code between `PERFORM` and `END-PERFORM` is repeated and stops when it is true that `NUM1 > 10`. Note that there has to be something that will change the condition within the loop otherwise there would be an infinite loop. In the example, the statement `ADD 1 TO NUM1` means that the condition `NUM1 > 10` will become true (as long as `NUM1` can hold values greater than 10) and the loop will stop. The program would then continue with any code after the `END-PERFORM`.

```
PERFORM UNTIL NUM1 > 10
  DISPLAY NUM1
  ADD 1 TO NUM1
END-PERFORM
```

Note that in COBOL, the code in the loop runs every time the condition is **false** and does not run when the condition is **true**. This is opposite to the use of **while** in other languages, where the code runs while the condition is **true** and not if the condition is **false**.

Exercise 10

Create a program (or add to your calculations program) to show a times table for a number supplied in the JCL). Use a loop to achieve this. The output might start as follows:

```
01 x 05 = 005
02 x 05 = 010
03 x 05 = 015
```

This document was typeset by using LaTeX.