Preface

The new Workstation Compiler from the IBM Corporation, originally available only for the OS/2 Operating System, has since made its way from Windows and AIX (IBM's version of Unix) to the mainframe z/OS Operating System, including the version Unix System Services.

This book claims to be a basis for certification as an “IBM PL/I Certified Programmer/Developer”. On IBM computers, there is now decimal floating-point hardware available, which PL/I supports via the well-known DECIMAL FLOAT attribute and a new compiler option. So in the 7th edition, the section on floating-point arithmetic has been substantially rewritten.

IBM has worked on a set of questions which enables one to become certified as a PL/I Certified Programmer as well as a PL/I Certified Developer. Since the author was honored to help in developing the questionnaire, I have modified the text to reflect this test.

A Certified Programmer is expected to have only two to three years of experience with IBM PL/I. The expectation of the PL/I Certified Developer is that he or she will have had five to six years experience with the programming language. I would go so far as to imply that anyone who has read and understood this book should not have any problems answering the PL/I related questions of the certification test.

This book offers a modern introduction to “Programming Language Number One”. Its aim is to provide beginners with material for self-study as well as to provide professionals with new ideas, particularly on the basis of its comprehensive presentation of the language. If an example should at first appear unintelligible to an experienced PL/I programmer, I hope that from its explanation he or she can pick up something new in the language.

You should expect no theoretical discussions of algorithms or structure diagrams, but instead a practical introduction which should allow you to solve real problems with the help of PL/I in a straightforward way. In contrast to a reference manual, I will always advise you which usage of the language is “good” and which is “bad”.

This book would not exist without the Internet. It's obvious that I could not inflict an English reader to read a book written by a non-native speaker. So I wrote a posting to the PL/I mailing list asking for proofreaders. It's a great honor for me to thank Richard Barrow, Francis Byrne, Peter Elderon, Peter Flass, John Gilmore, Tom Linden, Ray Mullins and Robin Vowels for undertaking the task to correct and improve the English created by a German who never lived more than three weeks in succession in an English speaking country.

An unexpected but inevitable side effect happened: many suggestions helped even improve the German version of this book. If you nevertheless find errors, then this is probably my fault when incorporating the proofreaders' comments into the final text.

All examples and the PARSE macro mentioned in the next to last section (similar to the REXX statement) can be found on the web under:

```
http://www.uni-muenster.de/ZIV/Mitarbeiter/EberhardSturm.html
```

Münster, August 2008 E. S.

Contents

digits and let the hardware set a floating decimal point. Such interest rates have exactly to be taken into account. Binary floating-point numbers would be out of the question.

So that we know what we are talking about, first the internal coding of such numbers, again as a 4-byte number, 8-byte number and a 16-byte number:

S	Combo field	Cont. Exponent	Continuation Mantissa
1	5	6	20

S	Combo field	Cont. Exponent	Continuation Mantissa
1	5	8	50

S	Combo field	Cont. Exponent	Continuation Mantissa
1	5	12	110

Figure 31. Decimal floating-point format (IEEE) on IBM mainframe

Let's start from the right. The digits of the mantissa are, of course, decimal digits (as the name suggests). But not the already well-known `fixed decimal` numeric format has been used, in which two digits are packed in one byte, but a new format ("densely packed decimal") where three digits fit in ten bits. Such a decimal digit is called a declet. Let us look at the 4-byte format: so 20 bits are six declets. But now we read something about "continuation mantissa": Another digit is hidden in the combination field, the latter serving several purposes as we will see.

We also see that `float decimal (7)` needs only four bytes, we have one digit more than in the other floating-point formats. Also, the maximum precision is 34, not 33.

The next field from the right is the continuation of the exponent. You have certainly guessed it, the beginning of the exponent is also hidden in the combo field! The exponent is indeed to base 10, but it itself is a binary number. As with the IEEE binary format, the range of the exponent is also larger in the longer formats (in contrast to hexadecimal). It is a happy coincidence that a growing field for the exponent arises when the continuation field of the mantissa is a multiple of 10.

The range of exponent of 4-byte numbers is from -94 to +97, 8-byte numbers go from -382 to +385, and 16-byte numbers from -6142 to +6145. This is, in any case, much more than for binary floating-point numbers, since here an exponent to base 10 is encoded.

Let us now turn to the combination field (in the figure called combo field). Understanding all bits is a little intellectual challenge and is left to the interested reader. Here is just so much: in the combination field not only the first decimal digit is coded and the beginning of the exponent, but also the "special numbers" plus and minus infinity and NaN (abbrev. of Not a Number). The latter is suitable, for instance, to identify a variable as being without value:

```
dcl F float decimal init ('7E000000'x); /* NaN */
F += 1;
```

are simply concatenated row by row. You have to read the table in such a way that the result of the combination of a value of the first row with one of the first column is to be found inside the table in the same row and column. Thus you see that `'0'b` combined with `'0'b` again results in `'0'b`, both `'0'b` with `'1'b` as well as `'1'b` with `'0'b` yields `'1'b`, and `'1'b` with `'1'b` again `'0'b`.

2.7.2 Set theory – working with bit strings

Bit strings are particulary well suitable for accomplishing set operations.[36] You can write very elegant programs, like the following example shows (pay attention also to the additional parentheses around the `initial` expressions!):

```
B30: /* Set theory (BIT) */
procedure options (main);

dcl Friday bit (366) init (((52)'0000001'b || '00'b));
dcl Thirteenth bit (366)
                init (((12)'0'b || '1'b || (30)'0'b
               || '1'b ||  (28)'0'b || '1'b || (30)'0'b
               || '1'b ||  (29)'0'b || '1'b || (30)'0'b
               || '1'b ||  (29)'0'b || '1'b || (30)'0'b
               || '1'b ||  (30)'0'b || '1'b || (29)'0'b
               || '1'b ||  (30)'0'b || '1'b || (29)'0'b
               || '1'b ||  (18)'0'b));

put list ('First Friday, the 13th in 2000 is day number '
          || search(Friday & Thirteenth, '1'b) || '.');
end B30;
```

The analogy of ordered sets to bit strings is obvious: The maximum power of the set corresponds to the length of the bit string. Each possible element corresponds to a bit in the string; if the element is contained in the set, the appropriate bit is set to `'1'b`, otherwise to `'0'b`.

In this way you can map the days of a year to `bit (366)`, if it concerns a leap year. In the year 2000 the 1st of January was a Saturday, in the variable `Friday` each 7th day is set to `'1'b`. The variable `Thirteenth` similarly contains a `'1'b` for each 13th of a month. If you want to determine now, which day in the year 2000 is a Friday, the 13th, you need only form the intersection from the set of Fridays and the set of the 13th. The intersection of two sets is the set of those elements, which are contained in both sets. Thus in the analogy it concerns a bit string, which exactly contains `'1'b` at those positions, where both basic strings are `'1'b`: thus the logical “and” is searched, expressed in PL/I: `Friday & Thirteenth`.

If we now look for the first Friday, the 13th, in the year, then we don't care for sets any longer, but call upon the power of PL/I bit manipulation: with `search` we determine either, whether there is none (i.e. function value 0), or get the position of the first bit, which is equal to `'1'b`. For those who are interested: the first Friday, the 13th in the year 2000 was surprisingly not until October.

We can keep in mind:

36 In languages such as Pascal or Modula-2 there are special data types and operators for set operations. With the definition of the programming language Ada one has intentionally omitted these and instead also recommends bit operations for this purpose.

Here the keyword type clarifies to everyone the fact that the word int is not a PL/I attribute, but was invented by you. (The PL/I attribute indeed exists, having a different meaning, however, but you don't need to worry about that.) You might even define, by the way, a variable Int as type int, which would not negatively be noticeable. It is said, that the so-called name spaces for variables and types are different!

You must not expect more advantages than saving writing effort or higher clarity. Variables of type int and variables of type fixed bin (31) are considered to be of the same type, can be combined, for example, with one another.[38] As a base type, all kinds of computational data and all kinds of program control data are permitted, also those with which we will become acquainted later in this book. You must not specify, however, the dimension attribute or a structuring.

2.8.2 Showing colors – enumerating types

Whereas the alias names were only other words for already well-known data types, now real data types of your own are to be presented, which do not have to do anything with already well-known ones. For this, we first introduce another kind of define statement:

```
define ordinal Color
   (red, orange, yellow, green, blue, indigo, violet);
```

Behind define ordinal you see the data type which is to be defined. In parentheses, a list of possible values is enumerated. Thus the colors red, orange etc. are constants. Still the question remains, how to declare variables:

```
dcl Surface_color type Color;
```

Only constants from the list or expressions of type color may be assigned to the variable Surface_color. The compiler prevents everything else! ordinal types cannot be combined with other data types.

We encounter the first problem, if we try to go through all colors with the help of the do loop. The to option is meant for numbers and not for colors. If you remember, the control variable of the counter loop always counted one increment too far – it is not really evident, which will come after violet in our list. But we can take remedial measures by using the keyword upthru instead of the keywords to and by. It is defined in such a way that exactly only the possible values will be run through, in the order of the definition list. In reverse order you can go through the list by using downthru; here are examples of both possibilities:

```
do Color = red upthru violet;
   ...
   end;

do Color = indigo downthru orange;
   ...
   end;
```

The next problem arises, if you want to read in or write out ordinal data. How can you write out values, which are intentionally not represented by numbers or character strings? Well – nevertheless there is a built-in function, the purpose of which is to convert an ordinal value

38 For language theoreticians: This is not strong typing!

An ID is something which Java uses for finding a thing such as a pointer or any other number. A catch in Java is that Java prescribes using a slash in the signature of the string element, i.e. you have to write `java/lang/String` for an element of class `java.lang.string`. A real Java program obviously does not need to execute the calls for the purpose of provision and releasing of PL/I readable strings – Java only knows Unicode, as mentioned before!

The sub-program for determining the value of a Roman numeral has been discussed in section 3.3.2, with only slight modifications.

7.4.2 Without Java – Java classes for PL/I

In the early years of PL/I, every new feature which the then IBM operating system OS/360 had brought, was built-in as a statement in PL/I; as an example think of multitasking. After this got out of hand, however, they confined themselves to the possibility of utilizing something new by calling assembler sub-programs from PL/I. At the present time, the creators of Java have the ambition to provide the Java programmer with everything the programmer's scene offers as a package with classes and methods – whether it be either connecting to the Internet or using encryption, to name but two. Using the so-called *Invocation Interface* of JNI you can also use all of this from PL/I.

In the following example, I will show how to start the so-called Java Virtual Machine (JVM) from a PL/I main program, then to connect to the Internet with the help of Java classes, then to ask a time-server for date and time and then to stop the JVM again. Before a PL/I-programmer would code `call` statements using obscure parameter lists, the programmer would most likely prefer to invent some PL/I macros and thus think about the dependencies only once. Therefore, I have presented a very clear program here which gives the impression that dealing with objects is something built-in in PL/I. The macros are outwardly defined purely in a way that the positional parameter, which comes directly after the macro name in the invocation, receives a value, with the other parameters are only to be "taken note of":

```
B76: /* Asking a time server for the time (PL/I calls Java) */
package;

%include java;

Mainprogram: /************************************************/
procedure options (main);

dcl Timeserver char (*) value ('time.uni-muenster.de');

put ('Timeserver ' || Timeserver || ' claims, that it is '
   || Date_and_time(TimeServer) || ' GMT.');

end Mainprogram;

Date_and_time: /*********************************************/
procedure (Timeserver) returns (char (17));

dcl Timeserver           widechar (*) varz parm nonasgn;
dcl DataInputStreamClass type jclass;
dcl DataInputStream      type jobject init (null());
dcl Inputstream          type jobject;
```

7.5 CGI and XML

The *Common Gateway Interface* (CGI) is a standard that defines the exchange of information with, for example, a Web server. The client is a browser, such as Firefox or Microsoft Internet Explorer. In this section we will describe how a PL/I program can work with the wishes of the browser and how to implement quite complex XML documents in response to the client.

7.5.1 Classical – CGI in PL/I

Let us assume that a user has entered a web address (URL) in the appropriate field of their browser and has pressed the Enter key. This web server then calls the specified program which is our PL/I program. In this simple case, we write a HTTP document in our reply to the standard output file, that is to `Sysprint`:

```
Content-type: text/html

<html><head>
<title>Conversion of Roman numerals</title>
</head><body>
<h1>Conversion of Roman numerals</h1>
<form action="http://www.anywhere.com/out_pli_program">
<p>Enter a Roman numeral, please:
<input type='text' name='X'>
<input type='submit' name='Y' value='Go'></p>
</form>
</body></html>
```

What is important is the blank line after the `Content-Type` line. Furthermore, we see the definition of a form. Our browser interprets this data and shows the following picture:

Figure 78. Browser picture of Example B77 of the first call.

When a user now enters a Roman numeral in the entry field and clicks on the Go-button, input data is sent to our program, that is the name and contents of the entry field as well as name and contents of the button pressed. How will a program now get at this data? The following example program shows us how it is done: