2.6.2.3 Gleitkomma dezimal

Das größte PL/I-Ereignis des Jahres 2007 war zweifellos die Einführung der hardwareunterstützten dezimalen Gleitkomma-Arithmetik! Eingebaut in IBM-Großrechner wurde nämlich das ebenfalls von der Normungsorganisation IEEE definierte Decimal-Floating-Point-Format (DFP) – und dieses kann mit dem seit Jahrzehnten in PL/I schon vorhandenen Datentyp `float decimal` einfach angesprochen werden.

Wozu brauchen wir nun eigentlich dezimale Gleitkommazahlen? Nun – wir haben schon bei der Einführung dezimaler Festkommazahlen gesehen, dass Dezimalbrüche, also Zahlen mit Nachkommastellen, oft nicht exakt im Zweiersystem dargestellt werden können. Und warum kommen wir nicht mit dezimalen Festkommazahlen aus? Man könnte doch meinen, wenn man in Cents rechnete, reiche das völlig. Nehmen wir aber das Beispiel von Tageszinsen. Diese können z. B. 0,0171 % sein (entspricht ungefähr 6,4 % im Jahr). Da ist es dann einfach sinnvoll, mit 30 Stellen zu rechnen und die Hardware ein gleitendes Komma setzen zu lassen. Solche Zinssätze müssen exakt in die Rechnung eingehen, binäre Gleitkommazahlen kämen also auch nicht in Frage.

Damit wir wissen, wovon wir reden, hier erst einmal die interne Speicherung solcher Zahlen, wiederum als 4-Byte-, 8-Byte- und 16-Byte-Zahlen:

V	Kombifeld	Forts. Exponent	Fortsetzung Mantisse
1	5	6	20

V	Kombifeld	Forts. Exponent	Fortsetzung Mantisse
1	5	8	50

V	Kombifeld	Forts. Exponent	Fortsetzung Mantisse
1	5	12	110

Abbildung 31. Dezimales Gleitkommaformat (IEEE) auf IBM-Großrechnern

Fangen wir von hinten an. Die Ziffern der Mantisse sind natürlich Dezimalziffern (wie der Name des Formats schon andeutet). Hier hat man aber nicht das schon von `fixed decimal` bekannte Zahlenformat genommen, in dem zwei Ziffern in ein Byte gepackt werden, sondern ein neues Format („densely packed decimal“), wo drei Ziffern in zehn Bits passen. Eine solche Dezimalziffer heißt Deklet. Schauen wir uns das 4-Byte-Format an: 20 Bits sind also sechs Deklets. Nun lesen wir aber etwas von „Fortsetzung Mantisse“: Eine weitere Ziffer wurde nämlich im Kombinationsfeld versteckt, welches tatsächlich mehreren Zwecken dient, wie wir sehen werden.

Wir sehen auch, dass `float decimal (7)` hier nur vier Bytes benötigt, wir habe also eine Ziffer mehr als bei den anderen Gleitkommaformaten. Auch die Maximalanzahl ist 34, nicht 33.

Das nächste Feld von rechts ist die Fortsetzung des Exponenten. Sie raten sicher schon, der Anfang des Exponenten ist wieder im Kombinationsfeld versteckt! Der Exponent ist zwar zur Basis 10, er selbst ist aber eine Binärzahl. Wie beim IEEE-Binärformat ist der Bereich

Hierbei kann aber im IBM-Standard die letzte Nachkommastelle falsch sein, z. B. wird aus 0.1 nicht 1/10, sondern 1/16. Die binäre Zahl ist nach der Umwandlung also immer kleiner oder gleich der ursprünglichen dezimalen! In den Formeln wird immer noch 1 zur Gesamtstellenzahl hinzugezählt. Damit das Wissen darüber nicht verloren geht, hier die Begründung[36]:

Ist die dezimale Genauigkeit (p,q) dann verlangt die Formel im Handbuch für die binäre Genauigkeit (für positive Zahlen p und q):

(1 + ceil(p*3.32), ceil(q*3.32))

Auch hinter dem Komma wird also die nächst größere ganze Zahl berechnet. Damit können wir im Prinzip eine weitere Stelle hinter dem Komma bekommen, die wir dann natürlich auch der Gesamtstellenzahl zuschlagen müssen. Ein Beispiel:

dec(2,1) → bin(1+6.64 , 3.32) → bin(1+7,4) → bin(8,4)

Nehmen wir die Zahl 9.5, diese benötigt tatsächlich 4 Stellen vor dem Komma, da 9 dezimal gleich 1001 binär ist. Ohne Nachkommastellen hätte man die 1 in der Formel durchaus weglassen können, 95 passt in 7 Binärstellen. Aber dann hätte man noch eine Regel mehr gehabt.

2.6.3.2 FLOAT-Operationen

Bei `float`-Addition, -Multiplikation und -Division hat das Ergebnis immer die Mantissenlänge der Zahl mit längerer Mantisse. Gleiches gilt für die Exponentiation bei Angabe eines `float`-Exponenten. Ein Fallstrick ist hier, dass bei `X ** Y` die Variable `X` nicht kleiner als Null sein darf (wegen der Berechnung über Logarithmen). Allgemein gibt es so viele mathematisch unterschiedliche Möglichkeiten, dass ich sie hier nicht aufzählen möchte:

Am besten schaut man bei jeder Exponentiation in der Tabelle des PL/I-Handbuchs nach!

Ein Problem ist mit Einführung der dezimalen Gleitkommaoperationen in Hardware (DFP) hinzugekommen. Da man natürlich die Vorteile der exakten Dezimalarithmetik nutzen möchte, muss man darauf achten, nicht auf Grund der PL/I-Regeln wieder in Binärarithmetik abzurutschen. Man überdenke z. B. die folgenden Zeilen:

```
dcl K fixed bin;
dcl A float dec;
A = K + A;
```

Nun wissen wir ja, dass bei gemischten Operationen `float` über `fixed` „siegt“ und `binary` über `decimal`. Die Begründung war ein größerer Zahlenbereich von Gleitkomma und die größere Schnelligkeit von binären Operationen. Deshalb passiert bei dieser Addition der schlimmste aller Fälle: Beide Operanden werden umgewandelt, so dass die Operation letztlich in `float binary` ausgeführt wird, nur um sofort wieder nach `float decimal` umgewandelt zu werden. Wie kann man hier Abhilfe schaffen? Man kann einfach schreiben:

```
dcl K fixed bin;
dcl A float dec;
A = decimal(K) + A;
```

36 Diese Begründung verdanke ich Robin Vowels, der sie der PL/I-Mailingliste am 25. Juli 2007 zuschickte.

der Tafel in der gleichen Zeile und Spalte zu finden ist. Man sieht also, dass `'0'b` mit `'0'b` wieder `'0'b` ergibt, `'0'b` mit `'1'b` sowie `'1'b` mit `'0'b` ergeben dagegen `'1'b` und `'1'b` mit `'1'b` wiederum `'0'b`.

2.7.2 Mengenlehre – Arbeiten mit Bitfolgen

Bitfolgen eignen sich sehr schön dazu, Mengen-Operationen durchzuführen.[41] Man kann sehr elegante Programme schreiben, wie das folgende Beispiel zeigt (man achte auch auf die zusätzlichen Klammern um die `initial`-Ausdrücke!):

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

dcl Freitag bit (366) init (((52)'0000001'b || '00'b));
dcl Dreizehnter 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 ('Erster Freitag, der 13., im Jahre 2000 ist der '
          || search(Freitag & Dreizehnter, '1'b) || '. Tag!');

end B30;
```

Die Analogie zwischen geordneten Mengen und Bitfolgen liegt auf der Hand: Die maximale Mächtigkeit der Menge entspricht der Länge der Bitfolge. Jedes mögliche Element entspricht einem Bit in der Folge; ist das Element in der Menge enthalten, sei das entsprechende Bit auf `'1'b` gesetzt, anderenfalls auf `'0'b`.

Die Tage eines Jahres kann man also auf `bit (366)` abbilden, sofern es sich um ein Schaltjahr handelt. Im Jahre 2000 war der 1. Januar ein Samstag, in der Variablen `Freitag` ist jeder 7. Tag auf `'1'b` gesetzt worden. Die Variable `Dreizehnter` enthält analog ein `'1'b` für jeden 13. eines Monats. Wenn man also feststellen will, welche Tage im Jahre 2000 ein Freitag, der 13., waren, braucht man nur aus der Menge der Freitage und der Menge der 13. die Schnittmenge zu bilden. Die Schnittmenge von zwei Mengen ist die Menge der Elemente, die in beiden Mengen enthalten ist. In der Analogie handelt es sich also um eine Bitfolge, die genau an den Positionen `'1'b` enthält, wo auch beide Grundfolgen `'1'b` enthalten: Gesucht ist also das logische Und, in PL/I ausgedrückt: `Freitag & Dreizehnter`.

Sucht man nun den ersten Freitag, den 13., im Jahre, so kümmern wir uns nicht mehr um Mengen, sondern nehmen die Mächtigkeit der PL/I-Bit-Manipulation in Anspruch: mit `search` stellen wir entweder fest, es gibt keinen (d. h. Funktionswert 0), oder erhalten die Position des ersten Bits, das gleich `'1'b` ist. Wen es interessiert: Der erste Freitag, der 13. im Jahre 2000 war erstaunlicherweise erst im Oktober.

Wir können festhalten:

41 In Sprachen wie Pascal oder Modula-2 gibt es für Mengen-Operationen eigene Datentypen und Operatoren. Bei der Definition der Programmiersprache Ada hat man diese absichtlich weggelassen und empfiehlt stattdessen auch Bit-Operationen für diesen Zweck.

2.8.1 Typen mit Decknamen – DEFINE ALIAS

Die einfachste Art, eigene Datentypen zu definieren, geschieht mit der define-alias-Anweisung. Sind Sie z. B. aus Ihrer bisherigen Lieblingsprogrammiersprache an den Typ *int* gewöhnt, so brauchen Sie nur zu schreiben:

```
define alias int fixed bin (31);
```

und schon können Sie diesen Datentyp benutzen:

```
dcl X type int;
```

wobei das Schlüsselwort type jedem verdeutlicht, dass das Wort int nicht ein PL/I-Attribut ist (das gibt es zwar auch, und das mit anderer Bedeutung, aber das braucht Sie nicht zu kümmern), sondern von Ihnen ausgedacht wurde. Sie könnten übrigens sogar eine Variable Int als type int deklarieren, das würde nicht negativ auffallen. Man sagt, die sogenannten Namensräume für Variablen und Typen sind verschieden!

Mehr Vorteile als Schreibersparnis oder höhere Übersichtlichkeit dürfen Sie nicht erwarten. Variablen vom Typ int und Variablen vom Typ fixed bin (31) gelten als vom selben Typ, können beispielsweise miteinander verknüpft werden.[43] Als Basistyp sind alle Berechnungsdaten und alle Programmsteuerungsdaten zugelassen, auch die, die wir erst noch kennenlernen werden. Nicht aufführen darf man allerdings das Dimensionsattribut oder eine Strukturierung.

2.8.2 Farbe bekennen – Aufzählungstypen

Waren die Aliasnamen nur andere Wörter für schon bekannte Datentypen, so sollen jetzt eigene Datentypen vorgestellt werden, die nichts mit schon bekannten zu tun haben. Hierzu führen wir zunächst eine weitere Art der define-Anweisung ein:

```
define ordinal Farbe
   (rot, orange, gelb, grün, blau, indigo, violett);
```

Hinter define ordinal steht der zu definierende Datentyp, in Klammern wird die Liste der möglichen Werte aufgezählt. Die Farben rot, orange usw. sind also Konstanten. Bleibt noch die Frage, wie man Variablen deklariert:

```
dcl Oberflächenfarbe type Farbe;
```

Der Variablen Oberflächenfarbe dürfen jetzt nur Konstanten aus der Liste oder Ausdrücke des Typs Farbe zugewiesen werden. Alles andere verhindert der Compiler! ordinal-Typen lassen sich nicht mit anderen Datentypen verknüpfen.

Auf das erste Problem stoßen wir, wenn wir versuchen, mit Hilfe der do-Schleife alle Farben zu durchlaufen. Die to-Option ist für Zahlen gedacht und nicht für Farben. Wenn Sie sich erinnern, die Laufvariable der Zählschleife wird immer um ein Inkrement zu weit gezählt – es ist wirklich nicht ersichtlich, was in unserer Liste hinter violett kommen soll. Abhilfe schafft hier, statt der Schlüsselwörter to und by das Schlüsselwort upthru zu benutzen. Es ist so definiert, dass exakt nur die möglichen Werte durchlaufen werden, in der Reihenfolge der Definitionsliste. In umgekehrter Reihenfolge durchläuft man die Liste mit downthru, hier beide Möglichkeiten:

43 Für Sprachtheoretiker: Dies ist also kein *strong typing*!

auch einen beschränkten Ausdruck enthalten (einen, den der Compiler schon auswerten kann).

Früher war es üblich, das `defined`-Attribut auch bei nicht zueinander passenden Variablen zu verwenden. Wenn Sie also ein altes PL/I-Programm neu übersetzen, bekommen Sie dafür eine Fehlermeldung. Diese können Sie aber abschalten, wenn Sie dem Compiler die Option `rules(laxdef)` mitgeben.

6.2.4 Überwältigend – iSUB-Definition

Dass es noch allgemeiner geht, zeigt PL/I mit der dritten Art der Variablendefinition, der sogenannten `isub`-Definition. Diese Art kennen „normale" Programmierer höchstens aus Fehlermeldungen, in denen es heißt, dass ein Scheinargument erstellt worden sei, weil die Attribute von Argument und Parameter nicht passten oder das Argument `isub`-definiert sei. Meistens gilt der erste Fall, wer macht sich dann schon die Mühe, im Handbuch nachzuschlagen, was wohl der andere bedeuten könnte? Dadurch entgeht einem aber eine der reizvollsten Möglichkeiten von PL/I! Mit Begeisterung kann ich vermelden, dass der neue IBM-Compiler jetzt iSUB-Definition für Matrizen aus Skalaren unterstützt, alle folgenden Beispiele sind also funktionsfähig!

Was fehlt also noch für eine Variablendefinition? Bei der Korrespondenzdefinition müssen Grund- und definierte Variable die gleiche Anzahl an Dimensionen besitzen, z. B. beides zweidimensionale Matrizen sein. Diese Einschränkung entfällt bei der `isub`-Definition. Es ist möglich, eine Variable so auf einer Matrix zu definieren, dass sie z. B. die Hauptdiagonale oder eine Nebendiagonale darstellt:

```
dcl Matrix         dim (10,10) float;
dcl Hauptdiagonale dim (10) float
                   defined Matrix (1sub, 1sub);
dcl Nebendiagonale dim (9) float
                   defined Matrix (1sub+1, 1sub);
```

Mit `1sub` ist der erste Index (engl. *subscript*) einer Matrix gemeint. Um also festzustellen, wo die Variable `Nebendiagonale(K)` liegt, schaut PL/I bei `Matrix(K+1,K)` nach. Die `isub`-Formel wird also jedesmal neu ausgewertet, wenn die `isub`-definierte Variable angesprochen wird. Sehr einfach lässt sich auch eine transponierte Matrix definieren, wie das folgende Beispiel zeigt:

```
B65: /* Matrix-Transposition (iSUB-DEFINED) */
procedure options (main);

dcl Matrix dim (10,10) float;
dcl Matrix_Transponiert dim (10,10) float
                        defined Matrix(2sub,1sub);
get list (Matrix);
put list (Matrix_Transponiert);

end B65;
```

Die transponierte Matrix hat genau dasselbe Dimensionsattribut wie die Grundmatrix, nur die Indizes sind beim Zugriff zu vertauschen, wie die `isub`-Angabe vorschreibt: erst der zweite Index (`2sub`), dann der erste (`1sub`).

Übrigens ist es möglich, eine Matrix mit 1,6 Mrd. Byte zu deklarieren, dabei aber nur 1 Byte Speicherplatz zu verbrauchen:

8. weist schließlich den von `Arabisch` erhaltenen Wert auf das Element zu.

Mit Kennung ist hier etwas gemeint, das Java zum Auffinden der Sache befähigt, etwa ein Zeiger oder eine andere Zahl. Ein Fallstrick ist die Vorschrift, in der Signatur des String-Elements Schrägstriche zu verwenden, also für ein Element der Klasse `java.lang.String` unerwarteterweise `java/lang/String` schreiben zu müssen. Die Aufrufe zur Bereitstellung und Freigabe der für PL/I lesbaren Zeichenfolge braucht ein echtes Java-Programm natürlich nicht auszuführen – Java kennt ja, wie erwähnt, nur Unicode!

Das Unterprogramm zur Ermittlung des Werts einer römischen Zahl ist schon in Abschnitt 3.3.2 besprochen worden, die Modifikationen sind nur gering.

7.4.2 Ohne Java – Java-Klassen für PL/I

In den Anfangszeiten von PL/I wurde jede neue Errungenschaft, die das damalige IBM-Betriebssystem OS/360 brachte, auch als Anweisung in PL/I eingebaut, man denke etwa an Multitasking. Als das aber überhand nahm, beschränkte man sich auf die Möglichkeit, Neues per Assembler-Unterprogramm von PL/I aus zu nutzen. In der heutigen Zeit haben nun die Erbauer von Java den Ehrgeiz, alles, was die Programmierszene bietet, als Paket mit Klassen und Methoden dem Java-Programmierer zur Verfügung zu stellen, sei es Internet-Anschluss oder Verschlüsselung, um nur zwei zu nennen. Durch das sogenannte *Invocation Interface* des JNI kann man dies alles auch von PL/I aus nutzen.

Im folgenden Beispiel zeige ich, wie man von einem PL/I-Hauptprogramm aus die sogenannte *Java Virtual Machine* (JVM) startet, mit Hilfe von Java-Klassen Verbindung zum Internet aufnimmt, einen Zeit-Server nach dessen Datum und Uhrzeit fragt und dann die JVM wieder stoppt. Ehe ein PL/I-Programmierer `call`-Anweisungen mit obskuren Parameterlisten hinschreibt, erfindet er lieber ein paar PL/I-Makros und durchdenkt die Abhängigkeiten nur einmal. Deshalb sei hier also ein sehr übersichtliches Programm vorgestellt, das den Eindruck erweckt, der Umgang mit Objekten sei in PL/I eingebaut. Die Makros sind rein äußerlich so definiert, dass der Positionsparameter direkt hinter dem Makro-Namen einen Wert erhält, die anderen Parameter werden nur „zur Kenntnis genommen“:

```
B76: /* Timeserver nach Zeit fragen (PLI ruft Java) */
package;

%include java;

Hauptprogramm:
procedure options (main);

dcl Timeserver char (*) value ('time.uni-muenster.de');
put ('Timeserver ' || Timeserver || ' behauptet, es sei '
   || Datum_und_uhrzeit(TimeServer) || ' GMT.');

end Hauptprogramm;

Datum_und_uhrzeit:
procedure (Timeserver) returns (char (17));

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

Man kann jetzt nahezu beliebige Java-Klassen benutzen, Objekte erzeugen und deren Methoden aufrufen – und wenn es aus dem Java2D-Paket ist und die Dodekaeder aus Abbildung 1 auf dem Bildschirm dargestellt werden!

7.5 CGI und XML

Das *Common Gateway Interface* (CGI) ist ein Standard, der festlegt, wie der Informationsaustausch z. B. mit einem Web-Server zu geschehen hat. Der Klient ist dann ein Browser, etwa Firefox oder MS Internet Explorer. Hier soll beschrieben werden, wie ein PL/I-Programm zum einen direkt auf die Wünsche des Browsers reagieren kann oder zum anderen durchaus komplexe XML-Dokumente in eine Antwort an den Klienten umsetzen kann.

7.5.1 Klassisch – CGI in PL/I

Gehen wir einmal davon aus, dass ein Benutzer eine Web-Adresse (URL) in das entsprechende Feld seines Browsers eingetragen und die Eingabetaste gedrückt hat. Der angesprochene Web-Server ruft dann das angegebene Programm, unser PL/I-Programm auf. In diesem einfachen Fall reagieren wir in der Weise, dass wir ein HTTP-Dokument in die Standard-Ausgabedatei schreiben, also nach `Sysprint`:

```
Content-type: text/html

<html><head>
<title>Umwandlung r&ouml;mischer Zahlen</title>
</head><body>
<h1>Umwandlung r&ouml;mischer Zahlen</h1>
<form action="http://www.irgendwo.de/unser_pli_programm">
<p>Bitte geben Sie eine r&ouml;mische Zahl ein:
<input type='text' name='X'>
<input type='submit' name='Y' value='Los'></p>
</form>
</body></html>
```

Wichtig ist die Leerzeile hinter der `Content-Type`-Zeile. Außerdem sehen wir die Definition eines Formulars. Unser Browser interpretiert diese Daten und zeigt das folgende Bild:

Abbildung 78. Browser-Bild von Beispiel B77 beim ersten Aufruf.