Literals

This is a rant about numeric and possibly other literals. We had a bad week last week for other reasons – TV recorder went wrong; we bought a new one and that has proved to be utter rubbish – even worse than Brexit. Please bear with me (should that be bare? growl).

The background to this is that I don't like AI-249 on user-defined literals and their use in the AI on big numbers as they currently stand. I am sure we can do better.

Symbols

I think part of the trouble with this topic is that the solidus character / (47, solidus) gets overused. Maybe this is another problem with that old typewriter which I believe triggered the double space situation discussed recently.

In the old days, we always used ÷ (247, division sign) for division. But the absence of ÷ from the typewriter made us use / instead.

Your teacher at infant school teaching you sums I am sure would have set problems like

Find 39 ÷ 3

They would not have set the problem as 39/3. That's an improper fraction not a division calculation at all. Indeed, fractions used to be typeset using a horizontal line thus

$$\frac{39}{3}$$

(Gosh that's difficult in Word).

I have looked through a lot of math(s) books and the use of a horizontal line for fractions such as 1/2 still occurs sometimes.

We typically use a solidus these days for fractions not having a handy means of doing a horizontal line. It also means that the numerator and denominator can be written in a larger font. It is interesting that Latin 1 has characters for ¼, ½, and ¾. If we need other fractions in that style we have to use the solidus and maybe superscript and subscript thus $^{5}/_{7}$.

Interestingly, the character set used on a flexowriter for Algol 60 which was the first dccent programming language I used included a proper divide character (octal 221) and a nice multiply (octal 261) as well as an up arrow (octal 201) which was used to indicate a power (rather than **).

The very fact that Latin-1 (and some typewriters) have special symbols for the commoner fractions such as ¼ and ½ shows that they should be thought of as literals and not divisions.

Complex numbers

These are a good example of numbers that have more than one part in their structure (we call them components in Ada).

In the case of complex numbers we chose to make the internal representation visible (may have been a mistake) thus

```
type Complex is
  record
    Re, Im: Real'Base;
  end record;
```

where Real is the generic parameter. We can then set literal values thus

```
X: Complex := (2.4, 5.0);
```

So we can consider the aggregate form (2.4, 5.0) to be the literal representation for Complex.

Ada 2012 also provides copious operations on complex numbers and for composing them. It also provides the constant i so that we can write things like

```
X := 2.4 + i*5.0;
```

I cannot make up my mind whether that is nice or not. It has a blatant multiplication. Note that it I write

```
N: Integer := 5;
```

then I do not consider that to be a multiplication of 1 by 5 – it's just 5 units along the real axis. But if I want to express 5i, I would prefer to think of that not as a multiplication of i by 5 but simply the value of the point 5 units along the imaginary axis. So, if I write

```
M := 5*i;
```

then I would prefer that not to mean multiplication but it does look rather like it.

Note also that Ada makes no provision for complex integers often called Gaussian integers. We cannot write

```
Z: Complex := (2, 5);
```

Well, we can but the compiler will complain. We need to declare a type called something like Integer_Complex.

So in the discussion above maybe I should have written

```
N := 5.0;   M := 5.0*i;
```

We could do literals for quaternions much the same way. Quaternions are important for expressing rotations in three dimensions (and four dimensions) and are widely used in the animation industry. A quaternion is somewhat like a complex number but as well as a real axis, there are three imaginary axes for i, j, and k. A general quaternion might be expressed as

```
6 + 4i + 7j – 2k
```

If we implemented them as a visible record then we might write

```
Q: Quaternion := (6, 4, 7, –2)
```

and there is no visibility of any multiplication.

Fractions and Rationals

At the moment Ada provides no facilities for operating on fractions. Of course we can operate on decimal fractions but we cannot declare and store 1/3 (that is one-third) or 1/7 as such. The Babylonians did all their calculations in unit fractions.

If I write 3/7, I do not think of that as 3 divided by 7 but more as one-seventh multiplied by 3. Indeed, we say "three sevenths".

If we think of fractions as just denoting some value on the real continuum, then indeed they have only one component.

However, if we think of them as numerator, denominator pair then this leads us into the field of rational numbers.

(That is a ghastly pun on the word field.)

A field in mathematics is an abstract algebraic structure with roughly the following properties.

> It has two operations (typically denoted by + and ×).
> Both operations have a unit (typically 0 for + and 1 for ×).
> Every element except 0 has an inverse.

The real numbers form an infinite field. The integers do not form a field because the inverse of an integer is not an integer.

However, integers mod 7 do form a finite field. The key point being that every integer has an inverse in that field. The inverse of 5 is 3 since $5 \times 3 = 15 = 1 \bmod 7$. It works because 7 is prime.

The mod 7 field is easy to deal with in Ada. Indeed we have modular types anyway and the obvious form for a literal is simply an integer such as 4.

The rational numbers also form a field. Every number of the form a/b where a and b are integers and b is not zero has an inverse, clearly it is b/a.

Another important example occurs in the analysis of algebraic equations. We can extend the rational field to give a Galois Field whose elements are the entities

$c + d\sqrt{2}$ where c and d are rationals.

This field is usually denoted by $R(\sqrt{2})$.

This is a genuine field since every element other than zero has an inverse. For example, the inverse of $3 + 2\sqrt{2}$ is perhaps surprisingly $3 - 2\sqrt{2}$.

If we were dealing with this field we could call the type R2 (this is the standard mathematical notation) and probably implement it as a record of four components

```
type R2 is record
   C_Num: Integer;
   C_Den: Integer;
   D_Num: Integer;
   D_Den: Integer;
end record;
```

and then we would like a notation for a literal value. We could simply write

X: R2 := (3, 1, 2, 1);

But I might prefer

X: R2 := (3/1, 2/1);

Note that the / here is not an operator but a constructor.

Or maybe we would like to write simply

X: R2 := 3 + 2√2;

But maybe that looks too much like a calculation than a literal. In any case the value of the square root of two cannot be represented as a real number such as 1.414...

Anyway, the point is that I might want to create literal values from somewhat arbitrary structures. Maybe involving strange symbols such as √.

Big numbers

The proposed big number package provides two main things: the ability to deal with giant integers (Big_Integer) and the ability to deal with rational numbers whose numerator and denominator are the giant integers (Big_Rational).

Maybe the rational package should be generic so that it can be instantiated with Integer, Long_Integer, or Big_Integer.

Now we need literals for all relevant types. We do it for the predefined types such as Integer, Float, Character, and String. So we should have literals for Big_Integer and Big_Rational. It is convenient to have literal forms for both.

Now AI-249 provides the ability to create a literal by association with a function that takes a string. This is done using aspects. The idea is to remove the clutter associated with the explicit call of the function every time. The AI gives the example

```
type Big_Integer is private
  with Integer_Literal => Big_Integer_Value;

function Big_Integer_Value(S: String)
  return Big_Integer;

...

Y: Big_Integer := –3:

-- equivalent to
--  Y:Big_Integer := – Big_Integer_Value("3");
```

Note that the minus sign is not part of the literal in this case.

AI-249 says nothing about the expected form of the string – that is perhaps sensible since we do not wish to have constraints on what we can do.

But for some reason the AI defines aspects Integer_Literal and Real_Literal. But the text of AI-249 shows no distinction between them. I believe that maybe there was a thought that some type might need two forms. But that can easily be factored into the processing of the string. And it is rather rigid to preempt the situation by tagging them with Real and Integer.

The predefined Ada types do indeed use numeric literals like that, but we should aim to be much more flexible. So perhaps just one aspect Literal is all we need.

I know that 2.4 defines the terms real_literal and integer_literal and gives their syntax. Its been like that since 1983. Time has moved on. That's 36 years ago.

Turning now to AI-208 on big numbers. I am quite happy with most of it although I see that my request for Square Root has been ignored! Groan.

But the literals are curious. The type Big_Integer has aspect Integer_Literal. Fine, that is the sort of thing one would expect. The definition of the structure expected piggybacks on Get for Integer_IO which is perhaps a bit naughty but does the job. So although the type Big_Integer is private we can conveniently set values into it such as

```
BI: Big_Integer := 123456;
```

and don't have to write

```
BI: Big_Integer := From_String("123456");
```

But when it comes to the type Big_Rational which is again private, we find it has aspect Real_Literal. And the outcome is that we can write

```
BR: Big_Rational := 3.8;
```

rather than the cumbersome

```
BR: Big_Rational := From_String("3.8");
```

The definition of the structure expected piggybacks on Get for Float_IO. That's very strange. Rationals have nothing to do with floating point.

Note that writing

```
BR: Big_Rational := 3.8;
```

results in giving BR the rational value 19/5. Note that Numerator and Denominator have to be adjusted to remove common prime factors so that the gcd is 1.

The serious trouble is that we cannot generally assign a rational value if the Denominator has prime factors other than 2 or 5. If the denominator is less than 16 then we could use a base other than 10 and for example set 1/13 in BR by

```
BR := 13#0.1#;              -- Yuk!
```

The situation is to some extent salvaged by a cunning trick. There is a divide function between two parameters of type Big_Integer to give a Big_Rational so we can write

```
BR := 1/13;
```

after all. But note that this is really

```
BR := From_String("1") / From_String("13");
```

The really awkward thing is that in order to check for zero we have to write things such as

if BR = 0.0 then

which looks as if BR is being compared with floating point zero and is as bad as BRexit.

The whole situation disgusts me.

Thoughts

So what to do. Well maybe change AI-249 to provide only one literal and please call it just Literal and change the !subject of the AI to "User-defined literals". I think that if AI-249 defines aspects for Real_Literal and Integer_Literal then it ought to say exactly what they are.

I feel perhaps that AI-249 is trying to be a bit too clever. Maybe it removes too much clutter. And bearing in mind that we might want to declare literals with several subcomponents we could use some brackets. It seems that square brackets are in fashion. They would indicate that it came via a Literal aspect.

Remember also that a key goal of Ada has always been that programs should be easy to maintain by someone other than the author. Perhaps we need some mark to indicate that the literal has come via an aspect. We might write

BI: Big_Integer := [–123456];

that would also have the merit of saying that the minus is part of the constructor of the literal and not an operator.

For the rationals we could have

BR : Big_Rational := [1, 13];

We could of course use solidus rather than a comma to separate numerator and denominator thus [1/13]

BR : Big_Rational := [1/13];

Hopefully that would clarify that the solidus is simply part of the literal and is not an operation.

The processing of the rational literal should be able to omit the denominator if it is 1. That is do not just piggyback on Get. So we could write

Zero: Big_Rational := [0];

and the test that a rational is zero becomes (ARG = [0]) rather than (ARG = 0.0).

This approach of using square brackets would permit other literal forms to be constructed for things like the Galois field and other algebraic structures discussed earlier. Thus we might have

X: R2 := [3 + 2√2];

and that would clarify that the + is not an operation but a constructor.

What to do

Here are some possible actions

1. Do nothing, leave it as it is. There is a risk that old John will resign from the ARG and not review anything else ever, ever! A bit unlikely since one supposes that he has to update his book. But he is very grumpy.

2. Remove the real_aspect for Big_Rational. Declare a constant Zero, so that instead of the obnoxious

(Arg =0.0)

we can write (Arg = Zero).

(PS Is Arg a good identifier. Confuses with ARG?)

All other rational values can then by created by the dubious trick of dividing big numbers thus

Seven: Big_Rational := 7/1;

which makes it quite clear what is going on.

Incidentally in my own use of my big number package of 20 years, I have rarely if ever found the need to use a big literal in the program text. I have sometimes declared a variable or constant called One or maybe Two but not much else. All big numbers have either been calculated or entered from the keyboard.

3. Do something more exciting as outlined above. But at least get rid of the Real_Literal and Integer_Literal aspects. Make it just Literal. Or if you must keep them, define exactly what the literal form is and provide a third aspect Literal for the ambitious user. Or maybe give the user the ability to add aspects so that Rational_Literal can be added.

4. Do not confuse rational numbers with a means of providing highly accurate real arithmetic. Define Big_Float or Big_Real. Preferably Big_Real.

To provide rationals is so easy that it hardly merits being in the language. But if it is, provide it as a separate item and do it properly.

Whatever is done, correct the phrase arbitrary-precision rationals from A.5.5. Rationals are always precise.

Acknowldgement

I am grateful to Jeff Cousins for his comments and suggestions on this topic.