Gem #7: The Beauty of Numeric Literals in Ada

Author: Franco Gasperoni (AdaCore)

Abstract: Ada Gem #7 — Did you know that the marvels of Ada extend to integer and real numbers (numeric literals really). Read on to learn about this Ada Gem.

Let's get started…

On an unusually hot end-of-summer day some years ago, I stepped into the Courant Institute of Mathematical Sciences. It was my first day of class at New York University and my first day with Ada. The heat was pounding and the classroom air-conditioning unit was being repaired.

The breeze that day came from something simple and elegant: numeric literals in Ada. I was fortunate that my programming languages class started with something so cool. The first thing that struck me is the ability to use underscores to separate groups of digits. I always found

```
  3.14159_26535_89793_23846_26433_83279_50288_41971_69399_37510
```

more readable and less error prone to type than

```
     3.14159265358979323846264338327950288419716939937510
```

what do you think? (I wish underscores were allowed when typing one's credit card number on the internet.)

This is just a cool beginning. Let's talk about based literals. I never understood the rationale behind the strange and unintuitive method to specify based literals in the C programming language where you have only 3 possible bases: 8, 10, and 16 (why no base 2?). Furthermore, requiring that numbers in base 8 be preceded by a zero feels like a bad joke on us programmers (what values do 0210 and 210 represent in C?)

To my pleasant surprise that humid end-of-summer day, I learnt that Ada allows any base from 2 to 16 and that we can write the decimal number 136 in any one of the following notations

```
        2#1000_1000#     8#210#     10#136#     16#88#
```

Coming from a microcontroller background where my I/O devices were memory mapped I liked the ability to write:

```
        Lights_On  : constant := 2#1000_1000#;
        Lights_Off : constant := 2#0111_0111#;
```

and have the ability to turn on/off the lights as follows:

```
   Output_Devices := Output_Devices  or   Lights_On;
   Output_Devices := Output_Devices  and  Lights_Off;
```

Of course we can also use records with representation clauses to do the above, which is even more elegant, and I leave that to a future gem (any volunteers out there?).

Back to based literals. The notion of base in Ada allows for exponents. That is particularly pleasant. For instance we can write:

```
     Kilobinary  : constant := 2#1#e+10;
     Megabinary  : constant := 2#1#e+20;
     Gigabinary  : constant := 2#1#e+30;
     Terabinary  : constant := 2#1#e+40;
     Petabinary  : constant := 2#1#e+50;
     Exabinary   : constant := 2#1#e+60;
     Zettabinary : constant := 2#1#e+70;
     Yottabinary : constant := 2#1#e+80;
```

In based literals the exponent, like the base, uses the regular decimal notation and specifies the power of the base that the based literal should be multiplied with to obtain the final value. For instance `2#1#e+10 = 1 x` 2^{10} `= 1_024` (in base 10), whereas `16#F#e+2 = 15 x` 16^{2}`= 15 x 256 = 3_840` (in base 10).

Based numbers apply equally well to real literals. We can for instance write:

```
    One_Third : constant := 3#0.1#;  --  same as 1.0/3
```

Whether we write `3#0.1#` or `1.0/3`, or even `3#1.0#e-1`, Ada allows us to specify exactly rational numbers for which decimal literals cannot be written.

This brings us to the last nice feature of Ada for this gem. As Bob Duff would put it: Ada has an open-ended set of integer and real types.

As a result, numeric literals in Ada do not carry with them their type as in C. The actual type of the literal is determined from the context. This is particularly helpful in avoiding overflows, underflows, and loss of precision (think about 32l in C, which is very different from 321).

And this is not all: all constant computations done at compile time are done in infinite precision be they integer or real. This allows us to write constants with whatever size and precision without having to worry about overflow or underflow. We can for instance write:

```
           Zero : constant := 1.0 - 3.0 * One_Third;
```

and be guaranteed that constant Zero has indeed value zero. This is very different from writing:

```
One_Third_Approx : constant := 0.3333333333333333333333333333;
```

```
Zero_Approx      : constant := 1.0 - 3.0 * One_Third_Approx;
```

where `Zero_Approx` is really `1.0e-29` (and that will show up in your numerical computations.) The above is quite handy when we want to write fractions without any loss of precision. Along these same lines we can write:

```
   Big_Sum : constant := 1           +
                         Kilobinary  +
                         Megabinary  +
                         Gigabinary  +
                         Terabinary  +
                         Petabinary  +
                         Exabinary   +
                         Zettabinary;

   Result : constant := (Yottabinary - 1) / (Kilobinary - 1);
   Nil : constant := Result - Big_Sum;
```

and be guaranteed that Nil is equal to zero.

But I am getting carried away by the elegance of Ada numeric literals and almost forgot about the date of our next gem which will be on the `2#10_10#` of September (sorry I couldn't resist :) .

Have a happy summer fellow developers.

Related Source Code

Ada Gems example files are distributed by AdaCore and may be used or modified for any purpose without restrictions.