МАТЕМАТИКА И МАТЕМАТИЧЕСКО ОБРАЗОВАНИЕ, 2006
MATHEMATICS AND EDUCATION IN MATHEMATICS, 2006
Proceedings of the Thirty Fifth Spring Conference of
the Union of Bulgarian Mathematicians
Borovets, April 5–8, 2006

ON THE DEFINITION OF INTEGER DIVISION AND MODULUS IN PROGRAMMING LANGUAGES

Boyko B. Bantchev

Problems with the definitions of integer division, modulus, and related operations in programming languages are discussed. It is demonstrated that unifying, generic definitions can be employed to provide more clarity, consistency, and coherence in this area.

Introduction. In the world of computer programming, there is no kind of data more fundamental than integer numbers. Indeed, in any computer integers are the basis invariably used to represent floating-point, text, Boolean, and other data, even the programs themselves. There is hardly a programming language ever invented that would not have integers built in itself. Programmers make heavy use of integers whatever their application domain is. In view of this, it is striking to observe how weakly the arithmetic of integers is standardised across programming languages. Not only that, but too often within a single language there are omissions, inconsistencies, and incoherence with respect to how integer arithmetic is defined.

Of course, addition and subtraction present little confusion, if at all, but integer division and modulus are notorious. The problem of consistent and convenient definition of these and related operations is an old one, and keeps emerging in the computing literature, although not frequently [1, 2, 3, 6, 8, 10, 11].

In the rest of this article, we are going to:

- review the operations in the scope of our interest as present in programming and the shortcomings we observe with respect to their definitions
- introduce a general, systematic and coherent way of presenting such definitions.

Each of the proposed definitions is general enough to capture all acceptable meanings of the corresponding operation and embodies the essential properties of that operation without favouring particular instances of it.

Div, mod and their relatives. Most programming languages have some sort of integer division (**div**) and modulus (**mod**) operations. These two operations are usually defined on integers or subsets of integers, although the wider domain of "real" (i.e., floating-point) numbers is also suitable, and indeed used in some languages.

Defining **div** requires that, whenever a is not a multiple of b, some integer approximation of the actual quotient a/b is chosen for a **div** b. For example, the quotient can be truncated to an integer or rounded to the nearest one, to mention but two possibilities.

Accordingly, a **mod** b is defined so that a certain relation to division be preserved. One popular option is $a = (a \textbf{ div } b) \cdot b + a \textbf{ mod } b$. Thus, the choice of a **div** operation

determines that of **mod**, or both determine each other. Other possible relations between **div** and **mod** will be mentioned later on.

Integer division has to make use of integer approximation, but the latter is an useful operation in itself and as such is also present in most languages.

In mathematics and in some programming languages there is an operation called *integer part*, which is similar or the same as finding the integer approximation of a real number. A closely related operation is the *fractional part*.

How the mentioned operations are defined in programming is important for several reasons, such as:

- We need unambiguous, without flaws and omissions, and easily understandable definitions in order to be able to write correct and reliable programs. This includes consistency and coherence of the definitions within a language.
- We need practical definitions with sufficient generality and well studied properties in order to make a good use of the corresponding operations in our programs.
- We need to communicate algorithms and to maintain reproductivity of the results of our computations among different programming languages, therefore our definitions must be consistent across languages.

We put a special stress on the importance of *consistency*, *coherence* and *generality*. Consistency means that the definitions are non-contradicting as well as non-omissive. It also means that the corresponding operations are uniformly expressed in the language. Coherence is ensuring that the correlation between concepts is made explicit through their definitions. Generality means avoiding arbitrary restrictions in each definition alone, as well as ensuring that as many useful definitions/operations are provided as necessity and practicality dictate.

There are, however, serious flaws with respect to all desired properties. Let us summarize some of them.

• In some programming languages, integer division and modulus are introduced without actually defining them, and thus are left open for arbitrary interpretation by the implementer. It seems that the language authors' idea of these operations is so unquestionable to them that they "specify" nothing beyond "a **div** b produces an integer quotient" and "a **mod** b gives the remainder of dividing a and b".

• In most languages, there are restrictions on the domains of **div** and **mod**. Much more often than not, these operators are defined only on integers, and in some cases – only on positive divisors or both arguments positive. Such is the case e.g. with Pascal and Modula-2. In fact, not only there is no need for arithmetic sign restrictions, but it is very convenient to have **div** and **mod** on reals as well. This is long being advocated by D. Knuth [7] and present in some languages, such as Fortran, PL-1, Apl, Lisp, and Javascript, but is not widely popular.

Perhaps real arguments to **div** and **mod** tend to be avoided by language creators on the ground that "integer division is a division of integers", while it is really a division with integral result.

• Finding an integer approximation of a real number is an operation provided in most languages, but sometimes it is disguised as implicit type conversion, explicit type "cast" or other forms. It might be difficult to tell whether the different forms correspond to the same operation, and what exactly it is: truncation, rounding to nearest or something else.

The complementary operation "fractional part" is not so commonly provided.

Furthermore, in the presence of integer division on reals, x **div** 1 is a form of integer approximation of x, which in turn depends on how **div** is defined. Also, x **mod** 1 is a fractional part of x. In this case, the problem arises of x **div** 1 and x **mod** 1 being in conformance or not with other operations of the kind integer/fractional part existing in the language.

• In some languages, there are more than one operations of a kind, say **mod**, with different results. Not always, however, every such operation is matched by a corresponding **div** operation, so that the two be related as mentioned above. ADA is an example of such inconsistence: it has **mod** and **rem** operations and its **div** matches **rem**, but there is no division operation to match **mod**. LISP, on the contrary, is an example of consistent design in this respect by having several kinds of paired **div**s and **mod**s.

• A constant source of amazement to us is what a great diversity of definitions one can find for apparently similar concepts, even for the same concept. What is taken as a defining property in one language is a consequence of the definition of the same thing in another, and contrariwise. One is often compelled to prove theorems of equivalence just to find out whether two definitions have the same meaning.

For example, **mod** in PL/1 is defined by $a = (a \textbf{ div } |b|) \cdot |b| + a \textbf{ mod } b$, whereas the value of **mod** in ALGOL 68 features a definition involving integer division (called `over`) and a conditional expression: `(int r = a - a over b * b; r < 0 | r + abs b | r)`. The two **mod**s are nevertheless the same operation, elsewhere defined in yet another way, namely by a phrase such as "**mod** is always positive".

• Sometimes a language is inconsistent in the forms it uses to provide certain operations. C (and almost any of its derivatives), for example, has some of the discussed operations built-in, while others (**div**, **fmod**, **modf**) are library functions. This might seem a minor issue, but in fact such a separation implies different levels of importance, which is hardly well-grounded.

• Some programming languages simply lack useful arithmetic operations of the discussed kind. In many more, actually in a highly prevailing majority, unfortunate choices were made in respect to what particular variants of operations to introduce. (See the "Related work" section about the use of "floor".)

• Historically, a series of languages – e.g. C, FORTH, ADA, APL – changed their integer arithmetic support in important ways. Even when the change is an improvement (and sometimes it is not, as is the case with the FORTH's current standard), it is an evidence of problems with the established definitions.

• The apparent confusion with the meanings of integer division, modulus, etc. is partly due to the fact that there is no well-standardised terminology in the field. For example, each of the words "modulo", "modulus", "residue", and "remainder", as well as the abbreviations "mod", "res", and "rem" are often used to denote the same thing, and if two of them are used with different meanings, these meanings always need clarification: one cannot guess the definition by the name of the operation.

It is worth noting that the lack of a steadily used, consistent notation in the discussed area is a problem not only to programming and programming languages, but to mathematics in general [7, 9].

Some authors, e.g. [6, 11] draw attention to the fact that the modulus operation in programming is especially confusing not only to programmers but often to mathemati-

cians and computer scientists. In [11] some statistics were gathered showing that a significant percentage of professionals, asked of the properties of integer division and modulus, would tend to think that $\mathrm{sign}(a \textbf{ mod } b) = \mathrm{sign}(a) \cdot \mathrm{sign}(b)$ – a false assumption regardless of precisely how **mod** is defined. Seemingly, the delusion is due to inappropriately extrapolating to **mod** the fact that arithmetical sign distributes over multiplication and division. (As another evidence of the same, let us mention the following: a very respected scientist once wrote in a book on Ada that "[...] the remainder A **mod** B is in general a negative number when A/B is negative".)

In [6] the confusion is attributed to not paying attention to the difference between "modulo" as a relation (congruence) and as an operation: in the latter case the range of values can be defined in several different ways, and is a matter of choice in general.

Redoing the definitions. Let us now see how the set of operations we discuss can be redefined in a more general, consistent and coherent way, so that it be both more useful and less confusing.

It seems appropriate to start with the definition of integer approximation. We postulate the integer approximation of a real number x to be an integer $(\!(x)\!)$ satisfying

$$|(\!(x)\!) - x| < 1. \tag{1}$$

Our deliberate choice was to give as general definition for $(\!(x)\!)$ as possible, and we believe that the above one is what we need: it is simple enough and yet it ensures the basic properties of what might be called an integer approximation without favouring any particular kind of approximation. More precisely, from (1) it follows:

- $(\!(x)\!) = x \Leftrightarrow x$ is an integer;
- If x is not an integer, $(\!(x)\!)$ is one of the two integers that are closest to x from below and above; i.e., $(\!(x)\!)$ is either n or $n+1$ where $n < x < n+1$;
- Any one of the widely known specific kinds of integer approximation, and also many other useful definitions can be derived from (1) by specializing it.

Two popular kinds of integer approximation are *floor* and *ceiling*, which round their argument toward $-\infty$ and $+\infty$, respectively. As proposed by K. Iverson and popularized by D. Knuth, we denote them by $\lfloor x \rfloor$ and $\lceil x \rceil$. Also we use $[x]$ to mean "rounding to nearest integer".

Another known kind of $(\!(x)\!)$ is the *truncation* function, which rounds x towards 0. There is also, although less frequently used, a function that we will call *completion* – it rounds $x > 0$ towards ∞ and $x < 0$ towards $-\infty$. Inspired by the notation for floor and ceiling, we denote truncation and completion of x by $\lceil x \rfloor$ and $\lfloor x \rceil$.

Because we can call $(\!(x)\!)$ the "integer part of x", we can also define x's "fractional part" **frac** x coherently with $(\!(x)\!)$ by: $x = (\!(x)\!) + \textbf{frac } x$, and then introduce $\textbf{frac}_{\lfloor\rfloor}$, $\textbf{frac}_{\lceil\rceil}$ etc. for the specific kinds of **frac**.

It obviously follows from the definition that $|\textbf{frac } x| < 1$ always holds, and that $\textbf{frac } x = 0$ whenever x is integer.

The following table shows the values of all these functions for some example values of x in the leftmost column. For each kind of rounding, the corresponding pair consists of the integer and the fractional parts of x. The contents of the rightmost column will be explained in the "Related work" section below.

	⌊ ⌋		⌈ ⌋		⌊ ⌉		⌈ ⌉		[ ]		E	
1.7	1	.7	1	.7	2	-.3	2	-.3	2	-.3	1	.7
1.2	1	.2	1	.2	2	-.8	2	-.8	1	.2	1	.2
-1.2	-2	.8	-1	-.2	-2	.8	-1	-.2	-1	-.2	-2	.8
-1.7	-2	.3	-1	-.7	-2	.3	-1	-.7	-2	.3	-2	.3

Other possible kinds of $(\!(\,)\!)$ and **frac** are e.g. probabilistic and otherwise non-deterministic. A probabilistic definition would imply making a uniform choice between the two closest integers n and $n+1$, or a choice with probabilities for n and $n+1$ depending on how close x is to them etc.

Note that all of the above variants of $(\!(\,)\!)$ exhibit monotonicity ($x < y \Rightarrow (\!(x)\!) \leq (\!(y)\!)$) but in general $(\!(\,)\!)$ is not monotonic. A probabilistic variant, for example, is necessarily non-monotonic.

Using $(\!(\,)\!)$, it is natural to also define **div** for any two reals a and $b \neq 0$ by: $a \;\mathbf{div}\; b = (\!(a/b)\!)$, thus obtaining a generic definition of division. Some essential properties, such as $a \;\mathbf{div}\; 1 = (\!(a)\!)$ and $|a \;\mathbf{div}\; b - a/b| < 1$ follow immediately from the definition of **div**. (Others can also be proven, but we skip them for lack of space.)

Finally, we have to define **mod**. To this end, we can choose any of the following two equations as a defining condition for **mod**:

$$a = (a \;\mathbf{div}\; b) \cdot b + a \;\mathbf{mod}\; b \qquad \text{or} \tag{2}$$

$$a = (a \;\mathbf{div}\; |b|) \cdot |b| + a \;\mathbf{mod}\; b \tag{3}$$

or, equivalently:

$$\begin{aligned} a \;\mathbf{mod}\; b &= (\mathbf{frac}\; a/b) \cdot b \qquad \text{or} \\ a \;\mathbf{mod}\; b &= (\mathbf{frac}\; a/|b|) \cdot |b| \end{aligned}$$

In both cases the following basic properties of **mod** are fulfilled:

$$\begin{aligned} &|a \;\mathbf{mod}\; b| < |b|, \\ &a \;\mathbf{mod}\; b = 0 \;\Leftrightarrow\; a/b \text{ is an integer.} \end{aligned} \tag{4}$$

The two definitions (2) and (3) seem to be the only ones really used in programming languages, so their generalizing, through the generic use of **div**, is seemingly all we need in respect of **mod**. Unfortunately, choosing between (2) and (3) means that we can no longer keep a single generic definition. Instead, we now have two branches of the generic **mod**.

The reason for having to consider both equations (2) and (3) is that some of the options that (3) caters for cannot be covered by (2). For example, specializing (3) by use of $\mathbf{div}_{\lfloor\rfloor}$ leads to the range $[0, |b|)$ for $\mathbf{mod}_{\lfloor\rfloor}$ (i.e. $\mathbf{mod}_{\lfloor\rfloor} \geq 0$ holds). Specializing (3) through $\mathbf{div}_{\lceil\rceil}$ leads to the range $(-|b|, 0]$ for $\mathbf{mod}_{\lceil\rceil}$ ($\mathbf{mod}_{\lceil\rceil} \leq 0$ holds). A specialization of (2), however, always gives the range $(-|b|, |b|)$ (as does the specialization of (3) through either of $\mathbf{div}_{\lceil\rfloor}$, $\mathbf{div}_{\lfloor\rceil}$ or $\mathbf{div}_{[]}$; in fact, (2) and (3) are identical in these cases).

The following table shows the values for the different variants of $a \;\mathbf{div}\; b$ and $a \;\mathbf{mod}\; b$, the latter having been defined through (2), with some example values of a and b in the

leftmost column. For each kind of rounding, the value pair for **div** and **mod** is displayed in the corresponding cell. The contents of the rightmost column will be explained in the "Related work" section below.

		⌊ ⌋		⌈ ⌋		⌊ ⌉		⌈ ⌉		[ ]		E	
34	5	6	4	6	4	7	-1	7	-1	7	-1	6	4
34	-5	-7	-1	-6	4	-7	-1	-6	4	-7	-1	-6	4
-34	5	-7	1	-6	-4	-7	1	-6	-4	-7	1	-7	1
-34	-5	6	-4	6	-4	7	1	7	1	7	1	7	1

Is it possible to replace (2) and (3) by a single, more general definition? Yes, but in such a definition a **mod** b will become unrelated to a **div** b, instead referring directly to a/b:

$$a \textbf{ mod } b = a - k \cdot b,$$

where k is an integer such that

$$|a/b - k| < 1.$$

As with 2 and 3, the above definition can be shown to satisfy (4).

We only demonstrated in this article how the definitions of several important operations related to integer division can be generalized and thus made more useful. Although we leave this out here, other operations can also benefit from the same approach. Two obvious examples are **gcd** and **lcm** which, through the Euclid algorithm, can be generalized and extended to the domain of reals.

Related work. Studies on the discussed topic are carried out in several directions. Some authors [2] present comparative data about the definitions of **div** and **mod** used in the programming languages of past and today, and describe the related arithmetic properties. Others [3, 6, 11] bring attention to the fact that **div** and **mod** are often not properly understood, and seek to explain the reasons for that.

The studies on the arithmetic properties of **div** and **mod** in [1, 8, 10] have lead their authors to the conclusion that the definitions which stem from the floor ($\lfloor\ \rfloor$)) rounding have more useful properties than e.g. those based on truncation, the latter being more popular only because most computer architectures implement them directly. Of particular interest is the so called "Euclidean division", defined in [1] and also discussed in [8]. Instead of first defining **div** and then using (2) or (3) to define **mod** it mandates

$$a = (a \textbf{ div } b) \cdot b + a \textbf{ mod } b$$
$$0 \leq a \textbf{ mod } b < |b|$$

as defining conditions for both **div** and **mod** simultaneously. It is argued that the Euclidean **div**/**mod** are as feature-rich as $\textbf{div}_{\lfloor\rfloor}$/$\textbf{mod}_{\lfloor\rfloor}$, and that they are superior to all other variants. In view of the approach followed in this article, the Euclidean definition is interesting in the following way: as a **div** b and a **mod** b are uniquely defined for each pair a, b, we could have also used the above equations to define integer and fractional parts of a number x as $(\!(x)\!)_E = x \textbf{ div } 1$ and $\textbf{frac}_E\, x = x \textbf{ mod } 1$, i.e. reverse the direction of dependence of the definitions. The rightmost columns of the two tables in the previous section show how the Euclidean definitions work on the sample numbers. One may notice that $(\!(x)\!)_E = \lfloor x \rfloor$ and $\textbf{frac}_E\, x = \textbf{frac}_{\lfloor\rfloor}\, x$.

Finally, serious attempts at systematizing the computer arithmetic have been done in two different (although related) standards: [4] and [5]. There is a lot of practical wisdom and mathematical rigour in the highly elaborated definitions of these documents, but from the point of view of the research presented in this article those definitions are somewhat unsuitable.

REFERENCES

[1] R. T. Boute. The Euclidean definition of the functions div and mod. *In ACM Trans. on Prog. Lang. and Systems (TOPLAS)*, **14** (1992), No. 2, 127–144.
[2] A. P. Chang. A note on the modulo operation. *SIGPLAN Notices*, **20** (1985), No. 4, 19–23.
[3] G. A. Hill. A note on the modulo operation in Edison. *SIGPLAN Notices*, **22** (1987), No. 4, 28–29.
[4] IEEE Standard 754-1985: Binary floating-point arithmetic, 1985.
[5] International Standard ISO/IEC 10967. Information Technology—Language Independent Arithmetic, 1994–2004.
[6] M. J. Jamieson. Integer division. Letter to the Editor. *Software—Practice and Experience*, **10** (1980), No. 4, 333.
[7] D. E. Knuth. The Art of computer programming, Vol. 1: Fundamental algorithms, 3rd ed. Addison-Wesley, 1998.
[8] D. Leijen. Division and Modulus for Computer Scientists. `citeseer.ist.psu.edu/463879.html`, 2001.
[9] MathWorld—A Wolfram Web Resource. `http://mathworld.wolfram.com`
[10] R. L. Smith. Signed Integer Division. *Dr. Dobb's Journal*, **8** (1983), No. 9.
[11] B. A. Wichmann. Integer division. *Software—Practice and Experience*, **9** (1979), No. 6, 507–508.

Boyko B. Bantchev
Institute of Mathematics and Informatics
Acad. G. Bontchev Str., bl. 8
1113 Sofia, Bulgaria
e-mail: `bantchev@math.bas.bg`

ВЪРХУ ОПРЕДЕЛЕНИЕТО НА ЦЕЛОЧИСЛЕНО ДЕЛЕНЕ И ОСТАТЪК ПО МОДУЛ В ЕЗИЦИТЕ ЗА ПРОГРАМИРАНЕ

Бойко Бл. Банчев

Обсъждат се недостатъци на определенията за целочислено делене, намиране на остатък по модул и близки до тях действия в езиците за програмиране. Показва се, че използването на обобщени определения води до по-голяма яснота, последователност и съгласуваност.