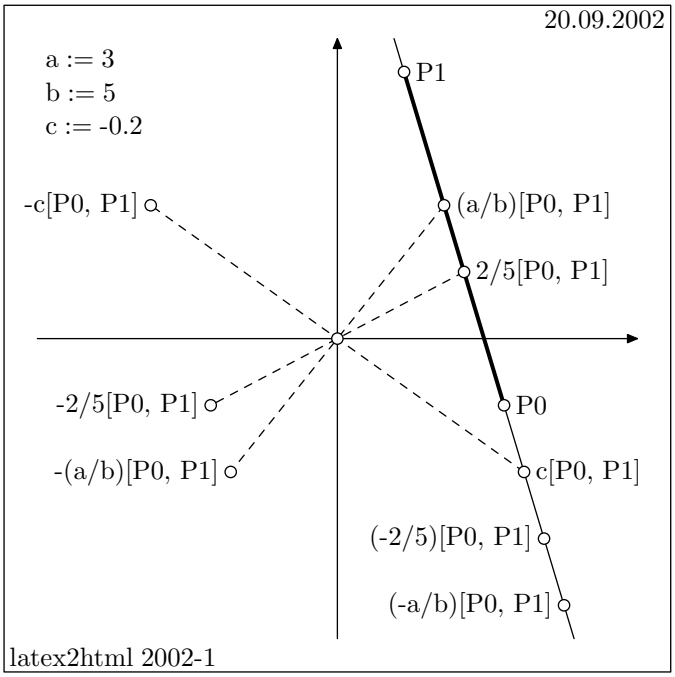
Precedence

osurs@bluewin.ch Urs Oswald http://www.ursoswald.ch

September 23, 2002

1 Why Are the Mediation Points Where They Are?



For MetaPost, '2/5' is a ⟨numeric expression⟩. As a consequence, '2/5[P0, P1]' is equivalent to '(2/5)[P0, P1]'. Furthermore, the figure above reveals that MetaPost reads '-2/5[P0, P1]' as '-((2/5)[P0, P1])'. Therefore, '-2/5[P0, P1]' is equivalent to '2/5[P0, P1] rotated 180'. And '-c[P0, P1]' is, for MetaPost, equivalent to '-(c[P0, P1])', which in turn is equivalent to 'c[P0, P1] rotated 180'.

In order to understand how MetaFont processes the input, one has to understand two things:

- how MetaPost breaks down the input into basic sequences called *tokens*,
- how MetaPost groups these tokens to *expressions*.

2 Breaking Down the Input Into Tokens

2.1 An Experimental Approach

In order to find out how MetaPost breaks down an input sequence like

- sqrt3.142/2.718[(4, 7), 5.2(cosd45, sind45)]∗(1+2.5∗1.732),

write a file `test.mp` consisting of just one line, namely

- `..sqrt3.142/2.718[(4, 7), 5.2(cosd45, sind45)]∗(1+2.5∗1.732).`

Then start MetaPost: `mpost test` (or, depending on the system, `mp test`), and answer each '?' of MetaPost with '1'. This causes MetaPost to read the input line token by token. (You exit by entering 'x'.)

In the case of the above input sequence, you will find that it is broken down into the tokens

sqrt 3.142 / 2.718 [(4 , 7) , 5.2 (cosd
45 , sind 45)] ∗ (1 + 2.5 ∗ 1.732).

2.2 Three Kinds of Tokens

MetaPost knows three kinds of tokens:

- numeric tokens,
- string tokens,
- symbolic tokens.

In the above example, the tokens

3.142 2.718 4 7 5.2 45 45 1 2.5 1.732

are *numeric*, all the others are *symbolic*. The exact rules of how input is broken down into tokens can be found in KNUTH[2] or HOBBY[1].What is or is not a numeric token is defined by the following set of syntactic rules (KNUTH[2]):

⟨decimal digit⟩ ⟶ **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** (A)
⟨digit string⟩ ⟶ ⟨decimal digit⟩ | ⟨digit string⟩⟨decimal digit⟩ (B)
⟨numeric token⟩ ⟶ ⟨digit string⟩ | **.**⟨digit string⟩ | ⟨digit string⟩**.**⟨digit string⟩ (C)

In order to prove that '1.4142' is a ⟨numeric token⟩, one first establishes, by rule (A), that each of '1', '4', '2' is a ⟨decimal digit⟩. So by the first part of rule (B), '1' and '4' are also of expression type ⟨digit string⟩. Using the second part of (B) thrice in a row, one proves that '4142' is a ⟨digit string⟩. Finally, by the third part of rule (C) it follows that '1.4142' is a ⟨numeric token⟩.

The notation used here and afterwards to formulate syntactic rules for expressions were introduced about 1960 by John Backus and Peter Naur. Rule (A) determines that a ⟨decimal digit⟩ is either of **0**, **1**, **2**, **3**, **4**, **5**, **6**, **7**, **8**, **9**, **and nothing else**. Similarly, rule (C) determines that an expression is a ⟨numeric token⟩ *iff* (if and only if) it is of one of the three following expression types: '⟨digit string⟩', '**.**⟨digit string⟩', '⟨digit string⟩**.**⟨digit string⟩'.

How can one prove that '.4.1' is *not* a ⟨numeric token⟩? One has to prove that it is neither a ⟨digit string⟩, nor of one of the two forms '**.**⟨digit string⟩' or '⟨digit string⟩**.**⟨digit string⟩'. As a ⟨digit string⟩ consists of decimal digits only, neither of '.4.1', '4.1', '.4' is a ⟨digit string⟩. So '.4.1' has none of the three possible forms of a ⟨numeric token⟩.

For MetaPost, the input sequences 'sqrt3a' and 'sqrt 3a' are equivalent, as both are broken down into the tokens

sqrt 3 a,

and so are the input sequences 'www.latex2html.org' and 'www latex 2html org', both of which result in the sequence

www latex 2 html org.

And both of 'sind30∗cosd60+sind60∗cosd30' and 'sind 30 ∗ cosd 60 + sind 60 ∗ cosd 30' yield the token sequence

sind 30 ∗ cosd 60 + sind 60 ∗ cosd 30.

3 Numeric Expressions

3.1 The Complete Set of Syntactic Rules

The following diagram is taken from KNUTH[2], p. 211. It contains the complete and final set of syntactic rules for numeric expressions.

⟨numeric atom⟩ ⟶ ⟨numeric variable⟩ (1)
| ⟨numeric argument⟩ (2)
| ⟨numeric token atom⟩ (3)
| ⟨internal quantity⟩ (4)
| **normaldeviate** (5)
| **(**⟨numeric expression⟩**)** (6)
| **begingroup** ⟨statement list⟩ ⟨numeric expression⟩ **endgroup** (7)
⟨numeric token atom⟩ ⟶ ⟨numeric token⟩**/**⟨numeric token⟩ (8)
| ⟨numeric token not followed by '**/**⟨numeric token⟩'⟩ (9)
⟨numeric primary⟩ ⟶ ⟨numeric atom⟩ (10)
| ⟨numeric atom⟩[⟨numeric expression⟩ , ⟨numeric expression⟩] (11)
| **length** ⟨numeric primary⟩ (12)
| **length** ⟨pair primary⟩ (13)
| **length** ⟨path primary⟩ (14)
| **length** ⟨string primary⟩ (15)
| **ASCII** ⟨string primary⟩ | **oct**⟨string primary⟩ | **hex**⟨string primary⟩ (16)
| ⟨pair part⟩⟨pair primary⟩ (17)
| ⟨transform part⟩⟨transform primary⟩ (18)
| **angle** ⟨pair primary⟩ (19)
| **turningnumber** ⟨path primary⟩ (20)
| **totalweight** ⟨picture primary⟩ (21)
| ⟨numeric operator⟩⟨numeric primary⟩ (22)
| **directiontime** ⟨pair expression⟩ **of** ⟨path primary⟩ (23)
⟨pair part⟩ ⟶ **xpart** | **ypart** (24)
⟨transform part⟩ ⟶ ⟨pair part⟩ (25)
| **xxpart** | **xypart** | **yxpart** | **yypart** (26)
⟨numeric operator⟩ ⟶ **sqrt** | **sind** | **cosd** | **mlog** | **mexp** (27)
| **floor** | **uniformdeviate** | ⟨scalar multiplication operator⟩ (28)
⟨scalar multiplication operator⟩ ⟶ ⟨plus or minus⟩ (29)
| ⟨numeric token atom not followed by + or - or a numeric token⟩ (30)
⟨numeric secondary⟩ ⟶ ⟨numeric primary⟩ (31)
| ⟨numeric secondary⟩⟨times or over⟩⟨numeric primary⟩ (32)
⟨times or over⟩ ⟶ * | / (33)
⟨numeric tertiary⟩ ⟶ ⟨numeric secondary⟩ (34)
| ⟨numeric tertiary⟩⟨plus or minus⟩⟨numeric secondary⟩ (35)
| ⟨numeric tertiary⟩⟨Pythagorean plus or minus⟩⟨numeric secondary⟩ (36)
⟨plus or minus⟩ ⟶ + | − (37)
⟨Pythagorean plus or minus⟩ ⟶ ++ | +−+ (38)
⟨numeric expression⟩ ⟶ ⟨numeric tertiary⟩ (39)

3.2 Some Consequences

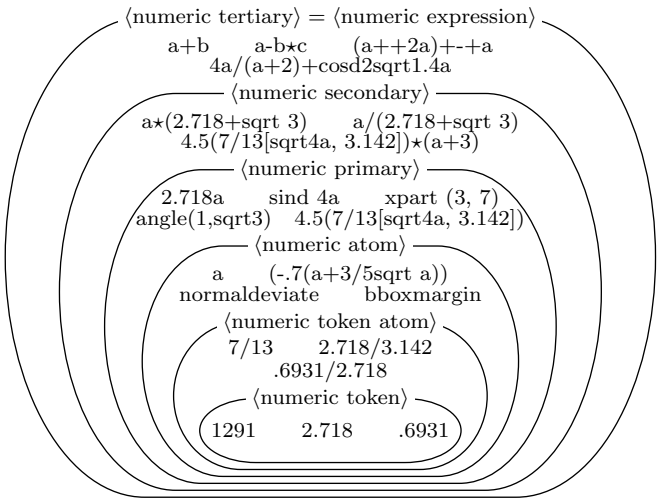
The above set of rules implies:

- Each ⟨numeric token atom⟩ is a ⟨numeric atom⟩. (1), (3)
- Each ⟨numeric atom⟩ is a ⟨numeric primary⟩. (10)
- Each ⟨numeric primary⟩ is a ⟨numeric secondary⟩. (31)
- Each ⟨numeric secondary⟩ is a ⟨numeric tertiary⟩. (34)
- Each ⟨numeric tertiary⟩ is a ⟨numeric expression⟩. (39)

By lines 8 and 9, a ⟨numeric token atom⟩ is either a ⟨numeric token⟩ or a ⟨numeric token not followed by '/⟨numeric token⟩'⟩. So a ⟨numeric token atom⟩ is a ⟨numeric token⟩ or has one of the 9 forms

4/7	.693/7	3.14/7
4/.618	.693/.618	3.14/.618
4/2.718	.693/2.718	3.14/2.718

The figure below gives an overview of the subtypes of ⟨numeric expression⟩. The classification of expressions is not absolute, but it depends on the context, as the following example shows. In the expression '.61803[2, 5]', the sequence '.61803' is a ⟨numeric token atom⟩; in the expression '.61803/3.14[2, 5]' it is not.



3.3 Two Examples

Each one of the input sequences 'sind4a', 'sind 4 a', and 'sind4.a' will result in the token sequence

$$\boxed{\text{sind}} \quad \boxed{4} \quad \boxed{\text{a}}.$$

(The 'd' in 'sind' refers to *degree*, as 'sind' expects the argument to be measured in degrees.) Let 'a' be numeric. By line 27 of the set of syntactic rules for numeric expressions, the first token, $\boxed{\text{sind}}$, is a ⟨numeric operator⟩. The second token is a ⟨numeric token atom not followed by + or - or a numeric token⟩. It is therefore, by line 30, a ⟨scalar multiplication operator⟩, and, by line 28, a ⟨numeric operator⟩. The third token is a ⟨numeric variable⟩ which is a ⟨numeric atom⟩ by line 1 and a ⟨numeric primary⟩ by line 10. Therefore, the token sequence has the form

⟨numeric operator⟩⟨numeric operator⟩⟨numeric primary⟩.

The only way of reducing this sequence is by line 22: ⟨numeric operator⟩⟨numeric primary⟩ yields ⟨numeric primary⟩, which means that 'sind4a' is equivalent with 'sind(4a)'. Again by use of line 22, we can finally reduce the sequence to ⟨numeric primary⟩.

The input sequence 'sind4∗a' (or, equivalently, 'sind 4 ∗ a' or 'sind 4∗a') has the form

sind⟨numeric token atom⟩∗⟨numeric variable⟩,

and can therefore be converted by the syntax rules into

- ⟨numeric operator⟩⟨numeric primary⟩⟨times or over⟩⟨numeric primary⟩,
- ⟨numeric primary⟩⟨times or over⟩⟨numeric primary⟩, (22)
- ⟨numeric secondary⟩⟨times or over⟩⟨numeric primary⟩, (31)
- ⟨numeric secondary⟩. (32)

The use of line 22 in the second step implies that 'sind4∗a' is equivalent to '(sind4)∗a'.

It may seem arbitrary that this time, we do not classify the token $\boxed{4}$ as a ⟨numeric token atom not followed by + or - or a numeric token⟩, as we did in the previous example. Had we done so, we would have ended up with

⟨numeric operator⟩⟨numeric operator⟩⟨times or over⟩⟨numeric primary⟩,

a sequence which cannot be proven to be a ⟨numeric expression⟩ by the syntax rules.

While each ⟨numeric primary⟩ is also a ⟨numeric secondary⟩, the reverse is not true. For the ⟨numeric secondary⟩ '2*3.1415', we can prove that it is not a ⟨numeric primary⟩ by proving that it is of neither of the possible expression types given in lines 10 through 23: It is neither an isolated ⟨numeric atom⟩ (10) nor a mediation expression (11), and interpreting the leading digit '2' as a ⟨numeric operator⟩ leads nowhere. It neither starts with a ⟨pair part⟩ nor a ⟨transform part⟩, and it does not begin with any of the remaining operators **length**, **ASCII**, **oct**, **hex**, **angle**, **turningnumber**, **totalweight**, **directiontime ... of**.

4 Syntax of Pair Expressions

4.1 The complete set of syntactic rules

The following diagram is taken from Donald E. Knuth, *The METAFONTbook,* p. 212:

⟨pair primary⟩ ⟶ ⟨pair variable⟩ (40)
| ⟨pair argument⟩ (41)
| **(**⟨numeric expression⟩ , ⟨numeric expression⟩**)** (42)
| **(**⟨pair expression⟩**)** (43)
| **begingroup** ⟨statement list⟩⟨pair expression⟩ **endgroup** (44)
| ⟨numeric atom⟩[⟨pair expression⟩ , ⟨pair expression⟩] (45)
| ⟨scalar multiplication operator⟩⟨pair primary⟩ (46)
| **point** ⟨numeric expression⟩ **of** ⟨path primary⟩ (47)
| **precontrol** ⟨numeric expression⟩ **of** ⟨path primary⟩ (48)
| **postcontrol** ⟨numeric expression⟩ **of** ⟨path primary⟩ (49)
| **penoffset** ⟨pair expression⟩ **of** ⟨pen primary⟩ (50)
| **penoffset** ⟨pair expression⟩ **of** ⟨future pen primary⟩ (51)
⟨pair secondary⟩ ⟶ ⟨pair primary⟩ (52)
| ⟨pair secondary⟩⟨times or over⟩⟨numeric primary⟩ (53)
| ⟨numeric secondary⟩**∗**⟨pair primary⟩ (54)
| ⟨pair secondary⟩⟨transformer⟩ (55)
⟨transformer⟩ ⟶ **rotated**⟨numeric primary⟩ (56)
| **scaled**⟨numeric primary⟩ (57)
| **shifted**⟨pair primary⟩ (58)
| **slanted**⟨numeric primary⟩ (59)
| **transformed**⟨transform primary⟩ (60)
| **xscaled**⟨numeric primary⟩ (61)
| **yscaled**⟨numeric primary⟩ (62)
| **zscaled**⟨pair primary⟩ (63)
⟨pair tertiary⟩ ⟶ ⟨pair secondary⟩ (64)
| ⟨pair tertiary⟩⟨plus or minus⟩⟨pair secondary⟩ (65)
| ⟨path tertiary⟩**intersectiontimes**⟨path secondary⟩ (66)
⟨pair expression⟩ ⟶ ⟨pair tertiary⟩ (67)

4.2 Mediation Points

Now let's go back to the mediation expressions '2/5[P0, P1]' and '-2/5[P0, P1]'. (We suppose that a preliminary declaration `pair P[]` has been made.) The first is of the form

⟨numeric token atom⟩[⟨pair variable⟩**,**⟨pair variable⟩].

So we can establish each of the forms

- ⟨numeric atom⟩[⟨pair primary⟩**,**⟨pair primary⟩], (3), (40)

- ⟨numeric atom⟩[⟨pair secondary⟩,⟨pair secondary⟩], (52)
- ⟨numeric atom⟩[⟨pair tertiary⟩,⟨pair tertiary⟩], (64)
- ⟨numeric atom⟩[⟨pair expression⟩,⟨pair expression⟩], (67)
- ⟨pair primary⟩. (45)

The only difference between the two sequences is the leading '-' in the second one. By lines 38 and 29, this '-' is a ⟨plus or minus⟩ and a ⟨scalar multiplication operator⟩. As the remaining sequence has been treated already in the first example, the second example reduces to

⟨scalar multiplication operator⟩⟨pair primary⟩

and finally, by line 46, to ⟨pair primary⟩.

What about 'a/b[P0, P1]'? This sequence is of the form

⟨numeric variable⟩/⟨numeric variable⟩[⟨pair expression⟩,⟨pair expression⟩]

and therefore

⟨numeric atom⟩/⟨numeric atom⟩[⟨pair expression⟩,⟨pair expression⟩]

by line 1 of the syntax rules. There are no rules for reducing ⟨numeric atom⟩/⟨numeric atom⟩. So MetaPost tries by applying line 45 to ⟨numeric atom⟩[⟨pair expression⟩,⟨pair expression⟩], reducing this expression to ⟨pair primary⟩. But then it gets stuck with

⟨numeric atom⟩/⟨pair primary⟩,

as it cannot divide a ⟨numeric atom⟩ by a ⟨pair primary⟩. The error message is:

```
! Not implemented: (known numeric)/(pair).
```

References

[1] John D. Hobby, *A User's Manual for MetaPost*.
http://cm.bell-labs.com/who/hobby/
ftp://ftp.dante.de/tex-archive/info/epslatex.ps (784 KB)
ftp://ftp.dante.de/tex-archive/info/epslatex.pdf (1665 KB)

[2] Donald E. Knuth. *The METAFONTbook*.
ADDISON-WESLEY, 1992, ISBN 0-201-13445-4 und ISBN 0-201-13444-6 (soft)