Two Specifications of Tabulated Equations

Prabhaker Mateti

This note contains both a "good" answer, and a "really"[1] precise answer to an exam question in a Software Engineering class. The occasional commentary by this instructor is set in the type size of this paragraph.

1 Informal Statement of the Problem

Consider the informal description, given in the next paragraph, of a program. Give a carefully written specification of it, using well-chosen notations, and words. Pretend that the requirements analysis is already done: That is, wherever relevant in the specs, explain in English the appropriate parts of requirements.

Our mathematician friend wants a program that can "tabulate" a sequence of symbolic equations found in typical linear algebra into a matrix form.

For example, given the input shown in Figure **??**,

```
3x + 5y + 714z = 12g + 53b,
76489x +3z = 2g+8b+99a + 1024,
           x+ 49 = 7g,
1234567y+ 2z = a + 999b + 6x + 911,
 + 3x + 5y =  53b + 12g.
```

Figure 1: The Input

your program should produce the result shown in Figure **??**.

Our friend tells us that her equations do not contain minus signs, nor floating point numbers, but that sometimes the coefficients are larger than can fit in 64-bits, and the equations become so long that she has to write them on a long scroll of paper sideways. We impressed her by remarking that we will imagine the paper width to be infinity.

[1] Do you really know the meaning of "really"? Look it up in a decent dictionary, such as Longman Dictionary of Contemporary English.

```
    3x +       5y + 714z      = 12g +  53b,
76489x +              3z      =  2g +   8b + 99a +      1024,
     x +                   49 =  7g,
         1234567y +   2z      =       999b +   a + 6x +  911,
    3x +       5y             = 12g +  53b.
```

Figure 2: Output Corresponding to Input of Figure **??**

2 A Good Solution

There are essentially three things we must consider. (1) What are the preconditions? (2) Output looks good. (3) Output is equivalent to the given input. This solution is deliberately written in an informal way to emphasize what needs to be specified versus how to specify it. So, find all the ambiguities – but, do not just nit pick!

Since nothing particular is said about how i/o ought to be, we assume that the required `tabulateeqn` program takes its textual input from `stdin` and outputs to `stdout`.

2.1 Syntax of Equations

The input must be a sentence derived from the non-terminal ***eqns*** of grammar G given below. Note that (a) an equation can span multiple lines, and (b) terms of the same variable may occur multiply either in the ***rhs*** or in the ***lhs*** or both.

```
eqns ::= eqn {',' eqn } '.'
eqn ::= lhs '=' rhs
lhs ::= exp
rhs ::= exp
exp ::= term | term '+' exp
term ::= number | coeff varid
coeff ::= empty | ['+'] number
varid ::= letter
```

(Requirements Analysis: It is not clear if long equations are presented on multiple lines in the input. We will de facto allow it as our grammar/parser of input takes care of it. If our client insists that that should not be permitted, our task becomes more complex. The required grammar is no longer context-free.)

Other than using some kind of grammar, I know of no way to describe this precondition. Syntax diagrams etc. are, of course, other ways of specifying context-free grammars. Why should we let `coeff` begin with an optional plus? Is the grammar "unambiguous" ? What does unambiguous mean in this context? The same as in an "unambiguous specification" ? If duplicate-var-id terms are not to be permitted in an equation, why does the required grammar become more complex? For that matter, are Pascal, C, Ada, ... context-free?

2.2 Output is Equivalent to the Input

Item 3 was easier to do than item 2. So we do that next.

The output is also a sentence derivable from `eqns`. The i-th equation of the input is *semantically equivalent* to the i-th equation of output. Two equations $e_1 = e_2$ and $e_3 = e_4$ are equivalent if e_1 is equivalent to e_3, and e_2 is equivalent to e_4. An expression e is equivalent to f if both e and f have the same terms, same number times in each. Note that permutations of terms is allowed. Simplification of terms is not even attempted.

2.3 Output is Well-Formatted

(Requirements Analysis: Since we impressed our customer by remarking that we will imagine the paper width to be infinity, we choose to output each equation as one line, regardless of its length.)

Let n be the number of lines in the output.

The output consists of equations printed one per line. The order of the equations is the same as that of input.

To describe the formatting of the output, imagine dividing the 2-d output

into columns of n rows as follows. Draw vertical straight-lines, from the topmost line to the bottom-most spanning the n lines of output, immediately surrounding each variable id letter, each `" + "` and the `" = "`. Similarly draw a line to the immediate right of each number, and one straight line at the leftmost edge of the output.

The first (i.e., the leftmost) column must be a number-column. Each row in a column of numbers must be either a string of blanks, or a right-justified number.

Each row of the equals-column must contain `" = "`. Each row of each plus-column must contain either a `" + "` or three blanks. All the rows of a variable id column must be either one blank or contain some letter, and all these letters must be the same.

Should we also say "no column is all blanks". I think not; what say you? What about "The first (i.e., the leftmost) column is a number-column" ?

3 A Precise Solution

The following is a precise and rigorous spec. Is it error-free? Is it complete? Obviously, some of the sentences from the preceding section need to be reproduced below if this section were the only thing you wrote.

3.1 Syntax of Equations

Imagine (better yet: You write it!) a predicate *parses-ok*(G, N, s) that yields true iff the string s is a sentence derivable from the non-terminal N in the grammar G.

Input must be a well-formed sentence of G: parses-ok(G, ***`eqns`***, `input`).

Let us call the required program as `TE`. Its signature is:

```
function TE(fi:  seq of eqn) returns fo:  seq of line
```

where `eqn` is a set of sentences generated by the ***`eqn`*** of the grammar above.

The auxiliary function that obtains the `fi` and `fo` sequences from their derivation trees is left as an exercise to you. Note that while the commas and the period in the input are helpful in parsing, the `fi` sequence does not contain them; but the `fo` does.

3.2 Output is Equivalent to the Input

Two equations $e : e_1 = e_2$ and $e' : e_3 = e_4$ are *semantically equivalent*,

semantically-eq$(e, e') \widehat{=}$
bag-of-terms(e_1) = bag-of-terms(e_3),
bag-of-terms(e_2) = bag-of-terms(e_4).

semantically-equal-eqns(input, output) $\widehat{=}$
forall $i : 1..n$ (parses-ok(G, `eqn`, `fo`$[i]$) and semantically-eq(`fi`$[i]$, `fo`$[i]$)).

Note that in parses-ok(G, `eqn`, `fo`$[i]$), it is `fo` and not `fi`. Why? What is the signature of semantically-eq? Is it ok to write semantically-eq(`fi`$[i]$, `fo`$[i]$) ?

3.3 Output is Well-Formatted

3.3.1 There Exists a Matrix

• We now imagine a matrix M of size $n \times c$, with the following properties.

well-formatted$(M, c) \widehat{=}$
forall $y : 1..c$ (is-seq-chars(y) and equal-width(y)) and
forall $y : 1..c$ (is-a-plus(y) or is-an-equals(y) or is-a-varid(y) or is-a-num(y))

Recall that n is the number of equations. For now, we do not know c; we simply postulate its existence. We refer to $M[*, y]$ as the y-th column.

We need the following aux functions.

• isblank$(s) \widehat{=}$ **forall** $i : 1..\#s(s[i] =$ blank).

• isnumber$(s) \widehat{=}$ parses-ok$(G, number, s)$.

• rjustified(s) $\widehat{=}$ **forsome** $k : 1..\#s$
(isblank($s[1..k-1]$) and isnumber($s[k..k]$) and isnumber($s[k..\#s]$))

Could we drop isnumber($s[k..k]$)? Should we change isnumber($s[k..\#s]$) to isnumber($s[k+1..\#s]$)?

• Each cell, $M[i, j]$, contains a seq of characters:
is-seq-chars(y) $\widehat{=}$ **forall** $i : 1..n$ ($M[i, y]$ in seq of char).

• Each row in a given column y is equal in width to the others in that column:
equal-width(y) $\widehat{=}$ **forall** $i, j : 1..n$ ($\#M[i, y] = \#M[j, y]$)

• Column y is a plus-column:
is-a-plus(y) $\widehat{=}$
forall $i : 1..n$ ($M[i, y] =$ `" + "` or isblank($M[i, y]$)) and
forsome $i : 1..n$ ($M[i, y] =$ `" + "`)

• Column y is an equals-column:
is-an-equals(y) $\widehat{=}$ **forall** $i : 1..n$ ($M[i, y]) =$ `" = "`)

• Column y is a varid column. At least one row contains a letter. Each row is either one blank or this letter.

is-a-varid(y) $\widehat{=}$
forsome z : letter
(**forsome** $x : 1..n$ ($M[x, y] = z$) and
. **forall** $i : 1..n$ ($M[i, y] = z$ or $M[i, y] =$ blank))

• Column y is a number-column. At least one row contains a number. Each row is either all-blanks or contains a right-justified number:
is-a-num(y) $\widehat{=}$
forall $i : 1..n$ (isblank($M[i, y]$) or rjustified($M[i, y]$)) and
forsome $i : 1..n$ (isnumber($M[i, y]$))

3.3.2 Print the Rows of M

The i-th line of output is a printing of the i-th row of M followed by either a comma if $i < n$, or a period if $i = n$.

• print-all(M) $\widehat{=}$
forall $i : 1..n-1$ (`fo`$[i]$ = print(i, M)| `","`) and `fo`$[n]$ = print(n, M)| `"."`

Print(i, M) is simply a catenation of all the columns of the i-th row and then trimming any blanks at the tail end.

Define print(i, M). Shouldn't the previous section also say this? Whatever happened to the value c?

3.3.3 Output is Looking Gooood!

• looks-good(s, n, c) $\widehat{=}$
forsome m : array $[1..n, 1..c]$ of seq of char
(well-formatted(m, c) and s = print-all(m))

3.4 Putting it all Together

output $\widehat{=}$ `tabulateeqns`(input), where
tabulateeqns(input) $\widehat{=}$ `TE`(q), where $q \widehat{=}$ convert-to-eqn-seq(input).

convert-to-eqn-seq(input) is a companion to parses-ok.

This output is such that
semantically-equal-eqns(input, output) and
forsome c : nat (looks-good(output, $\#q, c$)).

4 Discussion

The following are some imagined questions from students.

If we have some 40 minutes to spend on this problem, do you expect us to finish it this well?

Well, if you want an A, yes! More seriously, an answer along the lines of Section 2 is certainly expected. Section 3 is home work that you should be able to do a

good draft in a few, say 4, hours. Such a draft probably contains errors. It is best to let others read it. If that is not possible, read it yourself, but after a day or two.

Is it worth spending this much effort in specifying instead of producing several hundreds of lines of code?

I think so. There are no statistics that can validate or invalidate this belief. Assuming that the program being developed is not a throw-away program, but has a long lifetime, the careful translation of requirements to specs and then paraphrasing them back to the user/client is an effective technique to prevent expensive design and coding errors.