Input Syntax for LM Suite (version 7)

Mark van Schilfgaarde

1 Nov, 2007

1 Introduction

The input system for the LM program suite is unique in the following respects:

1. Input files are nearly free-format [1] and input does not need to be arranged in a particular order. Data parsed by identifying *tokens* (labels) in the input file, and reading the information following the token. In the string:
   `NSPIN=2`
   token `NSPIN` tells the input parser where to find the contents (`2`) associated with it. Note that a token such as `NSPIN` only acts as a marker to locate data: they are not themselves part of the data.

2. A tree of tokens completely specifies a particular marker. The full identifer we call a *tag*; it is written as a string of tokens separated by underscores, e.g. `SPEC_SCLWSR`, `SPEC_ATOM_Z`, `ITER_CONV`. Thus a tag is analogous to a path in a tree directory structure, and a token is analogous to either a directory or a file. Tokens analogous to 'files' (e.g. `NSPIN` above) are markers for data; tokens analogous to directories contain as their contents tokens nested more deeply into the tree.

   The same token may appear in more than one tag; their meaning is distinct, as we will see below. Thus contents of token `NIT` in the tag `STR_IINV_NIT` are different from the contents of `NIT` in the tag `ITER_NIT`. Sec. **??** shows how the structure is implemented for input files, which enables these cases to be distinguished.

3. The parser can read algebraic expressions. Variables can be assigned and used in the expressions.

4. The input parser has a limited programming language. Input files can contain directives such as
   `%if (expression)`
   that are not part of the input proper, but control what is read into the input stream, and what is left out. Thus input files can serve multiple uses — containing information for many kinds of calculations, or as a kind of data base.

2 Input structure: syntax for parsing tokens

This section explain how the tree structured tokens are supplied in the input file. A typical input fragment looks something like:

```
ITER NIT=2  CONV=0.001
     MIX=A,b=3
DYN  NIT=3
... (fragment 1)
```

The full path identifier we refer to as a *tag*. Tags in this fragment are: `ITER, ITER_NIT, ITER_CONV, ITER_MIX, DYN, DYN_NIT`. (Tags do not explicitly appear in the input, only tokens do.)

A token is one link in the path. A token's contents consist of a string, which may include data (when it is the last link in the path, e.g. `NIT`), or other tokens which name links further down the tree. It is analogous

[1] there are some mild exceptions to this; see discussion of categories in Sec. **??**

to a file directory structure, where names refer to files, or to directories (folders) which contain files or other directories.

The first or top-level tokens we designate as *categories*, (`ITER, DYN` in the fragment above). What designates the range of a category? Any string that begins in the first column of a line is a category. A category's contents begin right after its name, and end just before the start of the next category. In the fragment shown, `ITER` contains this string:

'`NIT=2 CONV=0.001 MIX=A,b=3`'

while `DYN` contains

'`NIT=3`'

Thus categories are treated a little differently from other tokens. The input data structure usually does not rely on line or column information; however this use of columns to mark categories and delimt their range is an important exception.

When a token's contents contain data, the kind of data it contains depends on the token. Data may consist of logical, integers or real scalars or vectors, or a string. The contents of `NIT`, `CONV`, and `MIX` are respectively an integer, a real number, and a string. This fragment illustrate tokens `PLAT` and `NKABC` that contain vectors:

```
STRUC  PLAT= 1 1/2 -1/2    1/2 -1/2 0   1 1 2
BZ     NKABC=3,3,4
```

Numbers (more properly, expressions) may be separated either by spaces or commas.

How are the start and end points of a token delineated in a general tree structure? The style shown in the input `fragment 1` does not have the ability to handle tree-structured input in general. Some other information must be added when the path has more than two levels, e.g. `STR_IINV_NIT`. A logical and unambiguous way to delimit the range of a token would be to embed its contents in brackets `[]`, e.g.

```
ITER[ NIT[2]  CONV[0.001]  MIX=[A,b=3]]
DYN[NIT[3]]
STR[RMAX[3] IINV[NIT[5] NCUT[20] TOL[1E-4]]]
... (fragment 2)
```

Tags `ITER` and `STR_IINV` contain these strings:

'`NIT[2] CONV[0.001] MIX=[A,b=3]`' and '`NIT[5] NCUT[20] TOL[1E-4]`'

while `ITER_NIT`, `DYN_NIT` and `STR_IINV_NIT` are all readily distinguished (contents `2,` `3`, and `5`).

The LM parser reads input structured by the bracket delimiters, as in `fragment 2`. However such a format is aesthetically unpleasant and hard for a person to read. For aesthetic reasons, some small compromises are made, and ambiguities tolerated, so that the format similar to that of `fragment 1` at the beginning of this section can be used most of the time. These are:

1. Categories must start in the first column.

2. When brackets are not used, a token's contents are delimited by the end of the category. Thus the content of `ITER_CONV` from `fragment 1` is '`0.001 MIX=A,b=3`', while in `fragment 2` it is the more sensible '`0.001`'.

   In practice this difference matters only occasionally. Usually contents refer to numerical data. The parser will read only as many numbers as it needs. If `CONV` contains only one number, the difference is moot. On the other hand a suppose the contents of `CONV` can contain more than one number. Then the two styles might generate a difference. In practice, the parser can only find one number to convert in the contents of `fragment 2`, and that is all it would generate.[2] For `fragment 1`, the parser would see a second string '`MIX=...`' but it fail to convert it to a number (it not a valid representation of a number). Thus, the net effect would be the same: only one number would be parsed.

[2]Whether or not reading only one number later produces an error, will depends on whether `CONV` *must* contain more than one number or it only *may* do so.

3. When a token's contents consist of a string (as distinct from a string representation of a number) and brackets are *not* used, there is an ambiguity in where the string ends. In this case, the parser will delimit strings in one of two ways. Usually a space delimits the end-of-string, as in `MIX=A,b=3`. However, in a few cases the end-of-category delimits the end-of-string — usually when the entire category contains just a string, as in `SYMGRP R4Z M(1,1,0) R3D`. If you want to be sure, use brackets.

4. Tags containing three or more levels of nesting, e.g `STR_IINV_NIT`, must be bracketed after the second level. Any of the following are acceptable:
   `STR[RMAX[3] IINV[NIT[5] NCUT[20] TOL[1E-4]]]`
   `STR[RMAX=3 IINV[NIT=5 NCUT=20 TOL=1E-4]]`
   `STR RMAX=3 IINV[NIT=5 NCUT=20 TOL=1E-4]`

Finally, multiple occurences of a token are sometimes required, for example multiple site positions or species data. The following fragment illustrates such a case:

```
SITE   ATOM[C1  POS= 0          0   0    RELAX=1]
       ATOM[A1  POS= 0          0   5/8  RELAX=0]
       ATOM[C1  POS= 1/sqrt(3)  0   1/2]
```

The parser will try to read multiple instances of tag `SITE_ATOM`, as well as its contents[3] The contents of the first and second occurences of token `ATOM` are thus: '`C1 POS= 0 0 0 RELAX=1`' and '`A1 POS= 0 0 5/8 RELAX=0`'.

The format shown is consistent with rule 4 above. For historical reasons, LM accepts another kind of format for this special case of repeated inputs:

```
SITE    ATOM=C1  POS= 0          0   0    RELAX=1
        ATOM=A1  POS= 0          0   5/8  RELAX=0
        ATOM=C1  POS= 1/sqrt(3)  0   1/2
```

In the latter format, the contents of tag `SITE_ATOM` are delimited by either the *next* occurence of this tag, or by the end-of-category, whichever occurs first.

[3] Note that token `ATOM` plays a dual role: it is simultaneously a marker for input data—the string for `ATOM`'s label (e.g. `C1`)—and a marker for tokens nested one level deeper, (e.g. contents of tags `SITE_ATOM_POS` and `SITE_ATOM_RELAX`).