INL/CON-24-81521-Revision-0

A Practical guide to Parsing MCNP Inputs: Lessons Learned from Implementing Context-Free Parsing in MontePy

April 2025

Micah D Gale, Travis J Labossiere-Hickman, Brenna Anne Carbno

INL/CON-24-81521-Revision-0

A Practical guide to Parsing MCNP Inputs: Lessons Learned from Implementing Context-Free Parsing in MontePy

Micah D Gale, Travis J Labossiere-Hickman, Brenna Anne Carbno

April 2025

Idaho National Laboratory
Idaho Falls, Idaho 83415

http://www.inl.gov

Prepared for the
U.S. Department of Energy
Under DOE Idaho Operations Office
Contract DE-AC07-05ID14517

A Practical Guide to Parsing MCNP Inputs: Lessons Learned from Implementing Context-Free Parsing in MontePy

Micah D. Gale [1,*], Travis J. Labossiere-Hickman[1], Brenna A. Carbno[1]

[1]Idaho National Laboratory, Idaho Falls, Idaho

[leave space for DOI, which will be inserted by ANS]

ABSTRACT

Monte Carlo N-Particle (MCNP) is a widely used Monte Carlo transport solver. Its development began in the 1960s, and due to this MCNP input files use a custom input syntax for which no off-the-shelf parsing libraries are available. For MontePy to create an effective object-oriented interface for MCNP input files, a context-free parser was implemented to fully parse the files. MontePy uses a number of shortcuts and optimizations to avoid creating a single universal input file parser. These lessons can be applied to working with the many other custom input syntax languages persistent throughout the nuclear industry.

Keywords: MCNP input, parsing, context-free grammar, domain-specific language

1. INTRODUCTION

Monte Carlo N-Particle (MCNP) is a popular Monte Carlo radiation transport solver. Its development began in the 1960s [1] and predates almost all modern markup and data serialization languages. Thus, it uses a custom input syntax. Many Monte Carlo models are large and complex with thousands of cells. In addition, the input models are heavily dependent on pointers (i.e., referencing other objects by number). This can make editing MCNP input files tedious and error-prone. For instance, changing a material's number could require changing its number in hundreds of locations. This makes editing MCNP input files an ideal candidate for automation. Though users have automated this editing for decades, the existing scripts are very domain-specific and cannot work with all possible MCNP input files.

MontePy is a Python library for reading, editing, and writing MCNP input files in a generalized way. MontePy provides an object-oriented interface for these files and aims to handle all valid input files. This allows users to focus on the task at hand rather than the specifics of MCNP syntax. Readers can learn more about MontePy at *www.montepy.org*. This work discusses implementing a context-free parser in MontePy to fully parse an input file and preserve user formatting. A context-free parser is capable of parsing context-free grammar, which categorically includes most programming and data languages [2].

2. BACKGROUND

MCNP quite famously has a completely unique input file syntax, or domain-specific language [1]. This means there are no off-the-shelf parsers available for its syntax. Any robust application programming interface (API) for MCNP input files needs to implement its own parser of this syntax. Previous tools have attempted to implement some of the functionality of MontePy [3]. MontePy is unique in that it is the only tool so far that combines: the ability to read, edit, and write MCNP input files in an object-oriented way; ease of installation and use; and extensive testing.

*micah.gale@inl.gov

2.1. Formal Language Theory and Grammar

To be able to parse any language robustly and extensively, be it a programming language or input file language, the language needs to be classified and rigorously defined. Generally, only regular and context-free languages are parsed. The following summary of regular and context-free languages is not meant to be exhaustive. For deeper insight, refer to a compiler textbook. The authors recommend *Engineering a Compiler* by Cooper and Torczon [2].

2.1.1. Regular Languages

Regular languages are among the simplest languages and are usually the ones most programmers are familiar with. Most programmers are familiar with these languages because they can be parsed by regular expressions.

It may be helpful to understand how regular expressions can be used. Regular expressions can be recognized by deterministic finite automata (DFA). DFAs can be represented as a finite state machine that decides what state to transition to next based on the next character in the stream being parsed [2]. A DFA might be implemented in Python as shown in Figure 1.

```
    def recognize(stream):
        start_state = 0
        state = start_state
        transition = {(0, "a"): 1....}
        result = ""
        for next_char in stream:
            next_state = transition[(state, next_char)]
            result += next_char
            state = next_state
```

Figure 1. Example implementation of a DFA for reading a regular language.

Regular languages are therefore rather limited by what a DFA can parse. Generally only the simplest languages are regular, and as such regular expressions cannot be used on their own as a parser. Two common language features that regular expressions cannot handle are properly matching parentheses and implementing operator precedence for mathematical equations [2].

MCNP is not a regular language and *should not* be read solely with regular expressions. It requires a full parser, or rather, ideally, a parser such as MontePy should be imported and used. Full context-free parsers are the only way to properly interpret the constructive solid geometry (CSG) definition of cells.

2.1.2. Context-Free Languages

Context-free languages comprise a much broader group of languages that includes most modern programming languages and regular languages. Context-free grammars are generally written in a top-down way, by starting with the ultimate goal of the rule, such as a sentence in a natural language, or a statement in a programming language, and then defining what can compose that goal [2].

An excellent example from *Engineering a Compiler* is defining the language *"SheepNoise"*:

$$\begin{aligned} \textit{SheepNoise} &\rightarrow \texttt{baa}\ \textit{SheepNoise} \\ &\mid \texttt{baa} \end{aligned}$$

SheepNoise can be either made of the **terminal** string "baa," or "baa" and the **nonterminal** *SheepNoise*. This recursive definition allows *SheepNoise* to be arbitrarily long. This example of a rule, or production, is very simple, and it is in fact also an example of a regular language. It also shows how a context-free grammar (for a context-free language) can be formally defined. For a language to be context free, the productions defining its grammar must have only one symbol (specifically a nonterminal) on their left side [2].

2.1.3. Syntax Trees

Tree structures are generally the most desirable way to represent complex statements, or sentences. As a reminder, an MCNP cell must define a unique number for the cell, its material composition (and its density if appropriate), a constructive solid geometry (CSG) definition, and other optional data as key-value pairs [1].

The cell input "`100 20 -3.0 1:2 -3 imp:n=1`" can be interpreted as the tree given in Figure 2.

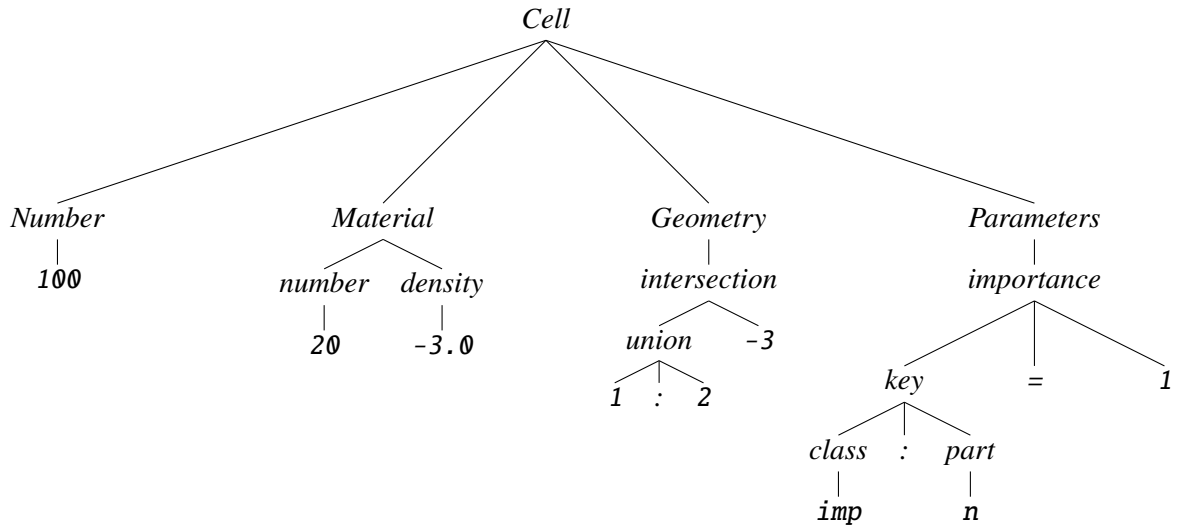

Figure 2. An example syntax tree for an MCNP cell.

The tree's representation of the CSG is not in the order that would be intuitive if read left to right because the implicit intersection operator takes precedence over the union operator. This tree is an *abstract syntax tree*. Extra information (white space, line continuation symbols, and comments) has been removed. When only the information from an input file is needed, this is sufficient. However, if the original input file is to be written out again, this information cannot be thrown away. In other words, if user formatting and comments are to be preserved, a concrete syntax tree is needed.

2.2. Scanners and Parsers

Parsing is generally completed in two phases: *scanning* and context-free *parsing* proper. Scanning, also known as "lexical analysis" or "lexxing," is a preprocessing step. Scanning uses regular expressions to split the input stream into tokens. Tokens identify the terminal type and the string that matched and may discard unneeded information. Offloading some grammar rules to the scanner simplifies the grammar that must be defined for the parser and speeds up the overall process, as scanners are generally more efficient than context-free parsers.

The context-free parser takes the stream of tokens processed by the scanner. If the tokens do not match the given grammar, the parser rejects them. Else, the parser processes the token and continues to the next token.

2.3. MCNP Grammar

MCNP syntax predates all modern markup and data serialization text languages, such as the extensible markup language (XML), java-script object notation (JSON), and YAML. As a result, MCNP has a custom and terse syntax designed for punch cards.

MCNP syntax rules are highly context dependent, though not necessarily in the formal context-free meaning. Broadly speaking, MCNP syntax can be described as a text-based, space-delineated list of either numerical values or key-value pairs, with numerous exceptions. MCNP input files consist of three blocks of inputs, each with its own syntax: one for cells, one for surfaces, and one for "data" (everything else). For example, consider a data block for which a prefix mnemonic determines the type of input: `"kcode"` inputs define eigenvalue mode settings, `"m"` inputs define a material as a list of nuclides, etc.

3. IMPLEMENTING MCNP FORMAL PARSING

3.1. Need

Useful MCNP models tend to be large and complex. Due to the use of pointers, a surface's or material's number could be used hundreds of times in a single file. If this number needs to change—for instance, to avoid a numbering collision when merging two models—simple tools such as find and replace will not work because a number can be reused to represent a cell, surface, material, etc.

Many programmers use an abundance of regular expressions ("regexes") to solve this problem. As discussed previously, MCNP input files do not use a regular language, and regular expressions can never robustly parse this language. On the other hand, most parsers produce an abstract syntax tree in which nonessential information is removed, such as white-space and comments. If an abstract syntax tree is edited and then rewritten to file, all the formatting of the original file is lost. This is highly undesirable as most MCNP users have developed their own unique style conventions to make MCNP inputs more legible. Therefore, it is necessary to create a concrete syntax tree for editing MCNP input files.

3.2. Objectives

MontePy was written to satisfy the needs described in the previous section. Its design priorities are:

1. Make it right (preserve the meaning of the user's input file)
2. Make it valid (ensure it uses proper MCNP grammar)
3. Make it easy (make MontePy easy for Python programmers to install and use)
4. Make it pretty (preserve user formatting)
5. Make it fast (optimize performance)

The authors are aware of only one other Python API for MCNP input files that use context-free parsers: MCNPy, written by Kowal et al. [4]. In contrast with MCNPy, MontePy does not use metamodel-driven design and is written purely in Python, rather than being a Python wrapper for a Java library. That is to say, MontePy does not define a high-level model description of the MCNP grammar and then rely on a framework to define the parser grammars and create classes, templates (default syntax trees), etc.

The authors did not implement a metamodel-design pattern with a Java framework because it conflicts with priority 3, by making it more difficult for Python users to install and apply the tool. Implementing a metamodel design pattern with a Java framework is not planned. The trade-off is that MontePy's parser grammars, object classes, and templates had to be manually written, though there are many opportunities for abstraction and meta-programming with the MCNP object design.

3.3. Decontextualizing and Creating Context-Free Language Atoms

It is impractical, if not impossible, to define one complete context-free grammar capable of parsing an entire MCNP input file. MontePy instead implements a number of practical shortcuts to reduce parser complexity.

First, the detection of multiline inputs, and different blocks is performed outside of the parser. These operations can be efficiently implemented with simple string operations, and doing so greatly simplifies implementing the parser. Next, block-specific scanning is performed. A separate scanner is used for each input block because the meaning of certain keywords and mnemonics are context specific. For instance, `"P"` in a surface input signifies a generic plane surface type, but elsewhere it signifies photon transport. Finally, dedicated parsers are used for certain input types. The cell and surface blocks each use a dedicated parser. The data block, however, is far more complicated to parse.

For data inputs, it is necessary to read the first value of each input, the "classifier," to determine the input type. For every input in the data block, the input is first fed to a classifier parser that determines what the input mnemonic is. A material parser handles material inputs consisting of a material number, nuclide-fraction pairs, and modifier parameters. A generic parser handles data inputs that are just lists of space-separated values and key-value pairs. Some input types have a grammar that breaks the general conventions of MCNP syntax. For example, in the context of the dose-energy input, `"DE"` uses the keyword `"log"` in a way that breaks the logarithmic interpolation shortcut in MCNP. Such input types are not parsed at this time. Instead, the original lines of text from the input file are scanned and stored as is.

3.4. Why SLY

The Python library Sly Lex Yacc (SLY) [5] was chosen as the parser generator for MontePy for a few reasons:

1. It is written in pure Python.
2. It has no additional dependencies, keeping it easy to install.
3. There is no need to compile the parsers prior to use.
4. The grammar is written using Python function decorators, making it very legible.

Parser generators tend to be written in compiled languages to improve performance, so there is a performance penalty to using SLY over some alternatives [5]. Given the design priorities of MontePy, this cost is well worth it, but this may be revisited in the future when the resulting overhead becomes a priority.

3.5. Handling MCNP Shortcuts

As noted by Kowal et al. [6], MCNP's support for shortcuts, such as `"1 10r"` to repeat `"1"` eleven times, adds complexity to parsing MCNP inputs. MontePy tackles this by treating shortcuts as part of the grammar of MCNP and making them transparent to the higher-level functions of MontePy. Most shortcuts are handled as special cases of a grammar for producing a list of numbers. A specific syntax tree node was subclassed from the list node for them. Once created, the shortcut is internally expanded into a list of values, and that whole node is inserted into a parent list-node. The parent list is then able to iterate over all values hiding the existence of the shortcut. This is done to track the shortcut so it can be recompressed at export.

3.6. Making a Concrete Syntax Tree

As mentioned, MontePy must work with concrete syntax trees, not abstract syntax trees. This complicates parsing. To address this, MontePy introduced the concept of "padding" to encompass all the information that is usually discarded in an abstract syntax tree, such as whitespace, newlines, and comments. The padding is then "glued" to the terminals, such as a number, to create a "phrase." This helped simplify the syntax tree, since for most use cases, the leaves of the tree were just the actual terminals, even though each leaf nodes

technically has two leaves: the actual terminal and any padding following it.

3.7. Post-Parsing Processing

Post-parsing processing is needed after parsing because the MCNP problem object is not ready for user interaction. The most important task at this stage is linking objects, or creating references. This is possible because assignments in Python create references [7]. This feature of Python is used extensively in MontePy. It allows a cell to refer to all the surfaces that it comprises, thus allowing the cell to be able to update the numbers of the surfaces in its own geometry definition without risking object duplication or creating multiple sources of truth. In addition to creating these references, data are also shifted between inputs to ensure a single source of truth and make a more intuitive interface.

3.8. Preserving User Intent with Changes

As mentioned before, MCNP users tend to have strong preferences for specific formatting. Preserving user formatting goes beyond just keeping comments and new lines. For example, it is desirable to preserve number formatting.

For floating-point numbers, this involves determining if scientific notation is used, how many digits of precision are kept, and so on. When a numerical value has changed and is reexported, MontePy tries to reverse engineer and replicate the formatting used. The other aspect of preserving number formatting is attempting to preserve user input alignment, which helps minimize the impact on formatting and keep inputs aligned. MontePy does this by counting the trailing white space and adjusting it to keep the overall number of characters unchanged. Sometimes this is not possible; for instance, if the cell number in the input `"1 0 -2 imp:n=1"` is changed to 1000, there is no whitespace to expand into. In this case, the expansion still occurs, and a warning is provided to the user.

3.9. Representing Cell Modifier Data

MCNP has a number of inputs that impact cells and can appear either in the cell block as key-value pairs, or as a long list in the data block. MontePy refers to these as "cell modifiers" because they modify how cells behave and provide more information. Examples include the cell `"universe"`, `"fill"`, `"importance"`, and `"volume"` inputs. These data can only appear in either the cell block or the data block.

This presents a design challenge for any object-oriented MCNP tool. How should this information be stored so that there is a single source of truth, cells can be added or removed, and the information can be moved from one block to the other? MontePy stores this information in the cell object as child objects that represent each input type. This is the most intuitive way to store this information from an object-oriented design perspective. When cell modifiers are present in the data block their information is transferred to the cells during post-parsing processing, and then internal copies of their data are deleted. If an input is not given in the data block a default object is created as a placeholder. The problem object then has a user-modifiable flag that determines whether the data should be printed in the data block or in the cells.

3.10. Rebuilding (Recompressing) a Shortcut

Preserving user formatting is also important for shortcuts. Imagine a large model without variance reduction, in which all cells have the same importance except for a single graveyard (vacuum boundary of zero-importance) cell. If the graveyard is the final cell, the importances may be provided in the data block as `"imp:n 1 1000r 0"`. If a user adds a new cell with unity importance to the problem right before the graveyard, it is unlikely that the user would want this action to push all importances to the cells, or to receive

“`imp:n 1 1000r 1 0`”. Rather, the user would be more likely to wish to receive “`imp:n 1 1001r 0`”. MontePy follows three principles with shortcut recompression:

1. Be valid (If a cell in the middle of the cell block has an importance of 0, preserve that information).
2. Be where the user intended (Do not go looking for shortcuts where the user had none).
3. Be greedy (If a shortcut can grow to include its neighbors, it should do so).

To handle shortcut recompression, first the new values are collected; usually this is completed for cell modifier. Next, the previous shortcuts in the list attempt to find the first values they produced and latch onto those points. The shortcut then iterates over the values to the left and right, checking to see if it can consume those values. The shortcut continues to “zipper” in values until it finds one that can’t be zippered in.

4. RESULTS

To demonstrate how the parser works for a simple problem, consider the cell defined as:

```
1 0 (-1 : 2) 3 imp:n=1
```

The first step in parsing is to scan this and to create a stream of tokens given by token type and value:

```
('NUMBER', '1')
('SPACE', ' ')
('NULL', '0')
...
```

Most of the tokens are intuitive, such as `NUMBER` and `SPACE`. The tokens are then fed into the cell parser, which creates a syntax tree, as shown in Figure 3.

Notice that the geometry of surface 3 is at the top level of the geometry definition in the tree, giving it operator precedence. This shows that the parentheses were interpreted properly. The syntax tree is complex.

4.1. Quantifying Performance

Using a context-free parser comes with a price. Prior to version 0.2.0, MontePy did not use context-free parsers: parsing was done primarily by splitting inputs by spaces, using the first value as a cell number, the second as a material number, etc. The parsing operation was very fast. With version 0.2.0, there was a noticeable increase in the time it takes to parse a large input file. In order to effectively quantify the change in performance, a large number of practical MCNP input files was needed.

A set of 1,472 models from the 2020 International Criticality Safety Benchmark Evaluation Project (ICSBEP) and the 2023 International Handbook of Evaluated Reactor Physics Benchmark Experiments (IRPhE) was used in this analysis. The time it took both MontePy 0.5.3 (with context-free parsers) and MontePy 0.1.7 (without context-free parsers) to read, parse, and post-parse process the input file was recorded. To eliminate effects of variance in computer demand, five trials were run for each version and the minimum time for each trial was tabulated. These times were then compared against the number of characters in the input files. These results are given in Figure 4.

The results in Figure 4 show that the older versions tested, 0.1.7 and 0.2.0, scale with $O(n^2)$. Both MontePy 0.1.7 and 0.2.0 scale in that way, so this effect is not caused by the type of parsers used but may be exacerbated by them. The source of this inefficiency was investigated. Profiling data were collected for reading in all

```
(Node: cell: {
'start_pad': (Padding, []),
    'cell_num': (Value, 1, padding: (Padding, [' '])),
    'material': (Node: material: {
        'mat_number': (Value, 0, padding: (Padding, [' '])),
        'density': (Value, None, padding: None)
    }),
    'geometry': Geometry: (
        Geometry: (
            Geometry: (
                 Geometry: ( (Value, -1.0, padding: (Padding,
                    [' '])) Operator._SHIFT None )
                 Operator.UNION (Value, 2.0, padding: None)
            )
            Operator._SHIFT None
        )
        Operator.INTERSECTION (Value, 3.0, padding: None)
    ),
    'parameters': (Parameters,
        {'imp:n': (Node: imp: {
            'classifier': (Classifier:
                mod: None,
                prefix: (Value, imp, padding: None),
                number: None,
                particles: :n),
            'seperator': (Padding, ['=']),
            'data': (list: number sequence, [(Value, 1.0,
               padding: None)])
        })
   })
})
```

Figure 3. Example syntax tree parsed by MontePy of the cell definition `1 0 (-1 : 2) 3 imp:n=1`.

the previous models in MontePy 0.4.1 and 0.2.0. In these older versions, the actual bottleneck was the `numbered_object_collection.py(append)` method, and not the parser. This collection holds the cells, surfaces, materials, etc. When a new object is added it checks for a numbering collision by checking the existing numbers of the other objects. For every object added, there is a check of every other object: hence $O(n^2)$. This was resolved in MontePy 0.5.3 by using hash tables mapping object numbers to the given objects. However, changing an object's number would invalidate the number in the hash table. Since MontePy 0.5.0 in almost all cases the objects in these collections have a link back to the collection. When their number is updated they trigger an update of the cache. This allows accessing and verifying if a number is in the collection in $O(1)$ time in almost all cases. This change can be seen in that MontePy 0.5.3 reads models in $O(N)$ time, which is the best scaling possible in this instance.

The profiling data from reading in all of these models are presented in Table I. From these data, it can be seen that the parse method of `yacc.py` (the parsing engine of SLY) is the second most expensive method called. The first is a method using for creating the syntax trees during the parsing process. With the $O(N^2)$ performance bugs now solved, switching to a more optimized parsing engine would improve performance. For most models, this cost of parsing is manageable. However, for very large models such as ITER, this performance cost is not as acceptable. To solve this there, are many possible avenues for further research:

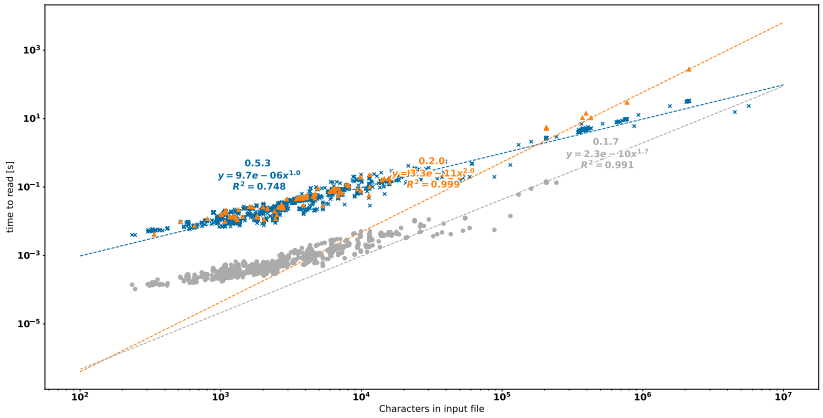

Figure 4. Time to parse an MCNP model vs. total number of characters in the model for MontePy 0.1.7, 0.2.0, and 0.5.3

1. The parser takes much longer than reading the file from the disk. This means that the parsing process could be safely parallelized to speed up the process.
2. Many users do not need to fully parse the whole file for their task so implementing a "just-in-time" parser could speed up working with an input model for many use cases.
3. Switching to a more optimized and compiled parser would ultimately improve performance in all use cases, but would come with significant development costs.

5. CONCLUSIONS

MCNP input files have a sophisticated input format. MCNP inputs are not regular languages and should not be parsed with regular expressions. We have shown that context-free parsers are capable of parsing MCNP input files. Due to these parsers, MontePy is able to be more robust and provide sophisticated editing features. These parsers come with a performance penalty. There are other optimizations of the code that should be considered before an alternate parsing engine is considered.

When creating new software, custom input syntaxes should not be created. There are multiple fully featured markup and data serialization languages with extensive software library support.

Writing a parser for legacy software is better than not being able to automate input generation, but it should be viewed as a last-resort effort. Creating these parsers is labor-intensive. Nuclear engineers should not spend their limited resources reinventing the wheel.

ACKNOWLEDGMENTS

Work supported through the Advanced Fuels Campaign (AFC) under DOE Idaho Operations Office Contract DE-AC07-05ID14517. This research made use of Idaho National Laboratory's High Performance Computing systems located at the Collaborative Computing Center and supported by the Office of Nuclear Energy of the U.S. Department of Energy and the Nuclear Science User Facilities under Contract No. DE-AC07-05ID14517.

Table I. The profiling results for reading in 1,472 models in MontePy 0.5.3. "cumulative time" is the time spent in the function in others it calls. "total time" is the time spent only in the function itself.

number of calls	total time [s]	cumulative time [s]	filename:lineno (function)
3773243	24.715	36.799	`syntax_node.py:566 (PaddingNode.__init__)`
166034	20.394	83.869	`yacc.py:2064 (parse)`
5002490	16.608	16.608	`{method 'copy' of 'dict' objects}`
925705	12.518	12.741	`syntax_node.py:1511 (ListNode.__init__)`
27397680/441026	11.701	255.44	`mcnp_object.py:38 (wrapped)`
4362544	11.333	30.359	`syntax_node.py:910 (ValueNode.__init__)`
1034158	10.148	10.445	`syntax_node.py:2460 (ParametersNode.__init__)`
12692938	7.751	7.751	`syntax_node.py:34 (SyntaxNodeBase.__init__)`
2164908	6.484	14.748	`lex.py:360 (tokenize)`
9501895	5.189	7.761	`utilities.py:76 (getter)`

REFERENCES

[1] J. Kulesza et al. "MCNP Code Version 6.3.0 Theory & User Manual." Report LA-UR-22-30006, Rev. 1, Los Alamos National Laboratory (2022). URL https://www.osti.gov/biblio/1889957.

[2] K. D. Cooper and L. Torczon. *Engineering a Compiler*. Morgan Kaufmann Publishers, San Francisco, CA (2004).

[3] M. D. Gale, T. J. Labossiere-Hickman, B. A. Carbno, and A. J. Bascom. "MontePy: a Python library for reading, editing, and writing MCNP input files." *Journal of Open-Source Software* (Submitted 2024).

[4] P. J. Kowal et al. "A (Meta-)Model Driven Approach to MCNP Modeling and Editor Services." In *2021 ANS Winter Meeting*, volume 125, pp. 1114–1117. American Nuclear Society (2021).

[5] D. Beazly and SLY contributors. "SLY (Sly Lex Yacc)." (2016). URL https://sly.readthedocs.io/en/latest/.

[6] P. J. Kowal et al. "Development of a Syntactic Validation Capability for the use of MCNP." (2021).

[7] Python Software Foundation. "Programming FAQ: How do I write a function with output parameters (call by reference)?" (2024). URL https://docs.python.org/3/faq/programming.html.