mathics-scanner

Release 1.0.1

The Mathics Team

Jun 26, 2021

CONTENTS:

This is the tokeniser or scanner portion for the Wolfram Language.

As such, it also contains a full set of translation between Wolfram Language named characters, their Unicode/ASCII equivalents and code-points.

CHAPTER

ONE

USING MATHICS-SCANNER

This is used as the scanner inside Mathics but it can also be used for tokenizing and formatting Wolfram Language code. In fact we intend to write one. This library is also quite usefull if you need to work with Wolfram Language named character and convert them to various formats.

- For tokenizing and scanning Wolfram Language code, use the `mathics_scanner.Tokenizer` class.
- To convert between Wolfram Language named characters and Unicode/ASCII, use the `mathics_scanner.characters.replace_wl_with_plain_text` and `mathics_scanner.characters.replace_unicode_with_wl` functions.
- To convert between qualified names of named characters (such `FormalA` for `\[FormalA]`) and Wolfram's internal representation use the `mathics_scanner.characters.named_characters` dictionary.

CHAPTER

TWO

API

2.1 Tokenization

Tokenization is performed by the `Tokeniser` class. The `next` method consumes characters from a feeder and returns a token if the tokenization succeeds. If the tokenization fails an instance of `TranslateError` is raised.

The tokens returned by `next` are instances of the `Token` class:

2.2 Feeders

A feeder is an intermediate between the tokeniser and the actual file being scanned. Feeders used by the tokeniser are instances of the `LineFeeder` class:

2.2.1 Specialized Feeders

To read multiple lines of code at a time use the `MultiLineFeeder` class:

To read a single line of code at a time use the `SingleLineFeeder` class:

To read lines of code from a file use the `FileLineFeeder` class:

2.3 Character Conversions

The `mathics_scanner.characters` module also exposes special dictionaries:

`named_characters` Maps fully qualified names of named characters to their corresponding code-points in Wolfram's internal representation:

```
for named_char, code in named_characters.items():
  print(f"The named character {named_char} maps to U+{ord(code):X}")
```

`aliased_characters` Maps the ESC sequence alias of all aliased characters to their corresponding code points in Wolfram's internal representation.

2.3.1 mathics_scanner.generate.rl_inputrc

CHAPTER

THREE

IMPLEMENTATION

3.1 The Tokeniser

Tokenization is performed by the `Tokeniser` class. The most important method in this class is by far the `next` method. This method consumes characters from the feeder and returns a token (if the tokenization succeeds).

3.1.1 Tokenization Rules

Tokenization rules can are defined by declaring methods (in the `Tokeniser` class) whose names are preceded by `t_`, such as in the following example:

```python
def t_SomeRule(self, match):
    # Some logic goes here...
    pass
```

A tokenization rule is supposed to take a regular expression match (the `match` parameter of type `re.Match`) and convert it to an appropriate token, which is then returned by the method. The rule is also responsible for updating the internal state of the tokeniser, such as incrementing the `pos` counter.

A rule is always expected to receive sane input. In other words, deciding which rule to call is a responsibility of the caller. Rules are are also automatically called from inside of `next`.

3.1.2 Messaging Functionality

Warnings and errors encountered during scanning and tokenization are collected in a message queue and stored in the feeders using the `message` and `syntax_message` methods of `LineFeeder`. The message queue is therefore a property of the feeder. The `Tokeniser` class also has a method to append messages to the message queue of it's feeder, the `syntax_message` method.

The messages are stored using Mathics' internal format, but this is going to be revised in the next release (in fact, we plan to replace messages by errors entirely).

3.2 Character Conversions

The `mathics_scanner.characters` module consists mostly of translation tables between Wolfram's internal representation and Unicode/ASCII. For maintainability, it was decided to store this data in a human-readable YAML table (in `data/named-characters.yml`).

The YAML table mainly contains information about how to convert a named character to Unicode and back. If a given character has a direct Unicode equivalent (a Unicode character whose description is similar as the named character's), this is specified by the `unicode-equivalent` field in the YAML table. Note that multiple named characters may share a common `unicode-equivalent` field. Also, if a named character has a Unicode equivalent, it's `unicode-equivalent` field need not to consist of a single Unicode code-point. For example, the Unicode equivalent of `\[FormalAlpha]` is `U+03B1 U+0323` (or `GREEK SMALL LETTER ALPHA + COMBINING DOT BELOW`).

If a named character has a `unicode-equivalent` field whose description fits the precise description of the character then it's `has-unicode-inverse` field in the YAML table is set to `true`.

The conversion routines `replace_wl_with_plain_text` and `replace_unicode_with_wl` use this information to convert between Wolfram's internal format and standard Unicode, but it should be noted that the conversion scheme is more complex than a simple lookup in the YAML table.

3.2.1 The Conversion Scheme

The `replace_wl_with_plain_text` functions converts text from Wolfram's internal representation to standard Unicode *or* ASCII. If set to `True`, the `use_unicode` argument indicates to `replace_wl_with_plain_text` that the input should be converted to standard Unicode. If set to `False`, `use_unicode` indicates to `replace_wl_with_plain_text` that it should only output standard ASCII.

The algorithm for converting from Wolfram's internal representation to standard Unicode is the following:

- If a character has a direct Unicode equivalent then the character is replaced by it's Unicode equivalent.
- If a character doesn't have a Unicode equivalent then the character is replaced by it's fully qualified name. For example, the `\[AliasIndicator]` character (or `U+F768` in Wolfram's internal representation) is replaced by the Python string `"\\[AliasIndicator]"`.

The algorithm for converting from Wolfram's internal representation to standard ASCII is the following:

- If a character has a direct Unicode equivalent and all of the characters of it's Unicode equivalent are valid ASCII characters then the character is replaced by it's Unicode equivalent.
- If a character doesn't have a Unicode equivalent or any of the characters of it's Unicode equivalent isn't a valid character then the character is replaced by it's fully qualified name.

The `replace_unicode_with_wl` function converts text from standard Unicode to Wolfram's internal representation. The algorithm for converting from standard Unicode to Wolfram's internal representation is the following:

- If a Unicode character sequence happens to match the `unicode-equivalent` of a Wolfram Language named character whose `has-unicode-inverse` field is set to `true`, then the Unicode character is replaced by the Wolfram's internal representation of such named character. Note that the YAML table is maintained in such a way that there is always *at most* one character that fits such description.
- Otherwise the character is left unchanged. Note that fully qualified names (such as the Python string `"\\[Alpha]"` or the Python string `"Alpha"`) are *not* replaced at all.

3.2.2 Optimizations

Because of the large size of the YAML table and the relative complexity of the conversion scheme, it was decided to store precompiled conversion tables in a file and read them from disk at runtime (when the module is imported). Our tests showed that storing the tables as JSON and using ujson to read them is the most efficient way to access them. However, this is merely an implementation detail and consumers of this library should not rely on this assumption.

The conversion tables are stored in the `data/characters.json` file, along side other complementary information used internally by the library. `data/characters.json` holds three conversion tables:

- The `wl-to-unicode` table, which stores the precompiled results of the Wolfram-to-Unicode conversion algorithm. `wl-to-unicode` is used for lookup when `replace_wl_with_plain_text` is called with the `use_unicode` argument set to `True`.
- The `wl-to-ascii` table, which stores the precompiled results of the Wolfram-to-ASCII conversion algorithm. `wl-to-ascii` is used for lookup when `replace_wl_with_plain_text` is called with the `use_unicode` argument set to `False`.
- The `unicode-to-wl` table, which stores the precompiled results of the Unicode-to-Wolfram conversion algorithm. `unicode-to-wl` is used for lookup when `replace_unicode_with_wl` is called.

The precompiled translation tables, as well as the rest of data stored in `data/characters.json`, is generated from the YAML table with the `mathics_scanner.generate.build_tables.compile_tables` function.

Note that multiple entries in the YAML table are redundant in the following sense: when a character has a Unicode equivalent equivalent but the Unicode equivalent is the same as it's Wolfram's internal representation (i.e. the `wl-unicode` field is the same as the `unicode-equivalent` field in the YAML table) then it is considered redundant for us, since no conversion is needed.

As an optimization, we explicitly remove any redundant characters from *all* precompiled conversion tables. Such optimization makes the tables smaller and easier to load. This implies that not all named characters that have a Unicode equivalent are included in the precompiled translation tables (the ones that are not included are the ones where no conversion is needed).

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search