LaTeX math parsing and manipulation

This is a programming tool that can read LaTeX math equations and parse them. This will create an AST (abstract syntax tree), which stores all information needed in order to manipulate the expression.

Usage

You can either import the function called 'N' from main.py or just run main.py through the command line. Then you can provide a latex expression and n-level

Basic explanation

So as of now (begin august 2024) this 'library' is not fully complete or tested yet. It may contain a good amount of bugs, but it acts as a foundation to build something working on! Here I will provide a rough explanation on the general flow of the application.

So, this is an application to read in LaTeX math expression, which can then be transformed, which results in a new LaTeX expression. The basic steps are as follows: Clean string and make some basic changes (remove whitespace, convert multiplication symbols) Feed the cleaned string into the lexer to get a sequence of tokens. Feed the tokens into the parser to build a meaningful expression tree. Transform the expression tree to your liking using so-called traversals. Done!

Lexer

Most importantly: https://www.dabeaz.com/ply/ply.html This is the PLY documentation which I consulted mostly.

It starts by putting the string through a lexer and parser, this is what the PLY library is used for. The lexer.py file contains the lexer, this converts the raw string into a sequence of so-called tokens. A token is in most cases the same or very similar to the character in the string. A left parentheses gets converted to a 'LPAREN' token. Some more examples:

- 5 -> NUMBER
- -5 -> MINUS NUMBER
- a + b -> SYMBOL PLUS SYMBOL
- \frac{5}{x * p} -> FRAC LBRACKET NUMBER RBRACKET LBRACKET SYMBOL TIMES SYMBOL RBRACKET

These characters are converted into tokens using regular expressions (https://regexr.com/), which tells the lexer how to recognize the tokens.

Some tokens store data as well, like the NUMBER token, which stores what value it used to be. The lexer basically 'dumbs down' the raw string, into something more readable and abstract for the parser.

Parser

Now that we have a more abstract version of the expression, we can use the parser (yacc.py) to convert the tokens into an AST (abstract syntax tree). In our case, this AST is built from Expr objects, which is a custom class made in Python. This will be explained later.

The main part of the parser is the grammar (https://en.wikipedia.org/wiki/Context-free_grammar). Which is just a set of rules that dictates how sequences of tokens and non-terminals should be interpreted. Non-terminals are like an intermediary token, they are recognized in the parser by lowercase notation.

A rule looks like this: sum -> term PLUS term

This rule tells the parser, that a sum consists of a term, following a + sign and then another term. The parser file contains multiple functions, each coupled with one or more grammar rules. These functions allow you to build the AST, you can instantiate classes and combine them while the parser parses.

As of now the parser does not have ambiguous rules or any major issues with precedence. It is structered in such a way that precedence is built-in by the rules (numbers and stuff are first converted into an atom, which can be raised to some power. These atoms are then converted into factors, then terms, etc.)

The parser can be confusing, but I find the documentation helps really well.

AST

The abstract syntax tree consists of instances from the data_types.py file. This file contains all the classes that can be encountered in the expression. IMPORTANT: All classes are derived form the Expr class.

The general idea is that each expression can consist of multiple expressions. It is structured like a tree.

For example: 5x + \frac{a}{b} starts as a Sum instance (because the top-level expression is a sum of two terms.). This Sum keeps track of its terms, which in this case are 5x and \frac{a}{b}. 5x is a Mul object (because it is 5 times x). \frac{a}{b} is Fraction instance, with a as its numerator and b as its denominator. In the 5x (or 5 * x), 5 and x are its factors. 5 is a Number instance and x is a Symbol instance.

All these classes, Mul, Sum, Fraction, etc. Are derived from the Expr class. Because they can all act as an expression. They can all have an exponent, be negative, have parentheses, etc.

Currently these classes have various methods, some of them are pretty self-explanatory. They also partially implement so-called dunder methods. These are special Python methods used in special cases. For example: 5 + 2 uses the __add__ dunder method builtin with the 'int' datatype. 5 + 2 is equivalent to 5.__add__(2). This is also used in our AST classes, a Number plus a Number returns a Sum instance. It does not evaluate its value, because in this application we are mainly interested in notation.

Traversals

To make changes to our expression, we can alter the AST by using traversals. A traversal is just a function, I've prefixed them with trav_. It starts by calling traverse_children, which is a method implemented in each type that applies the traversal to its child expression. The function should return an instance of an Expr, this will become the new (or same if you just return the original expression) expression in that location of the AST. This makes it possible to modify existing instances in the AST, but also add or remove expressions.

Conclusion

This is the general explanation, you can always ask for help to whomever. Reading through the code of course also helps. Apologies for the simple documentation and incomplete untested application. But this acts as a nice start hopefully.

Requirements

PLY fractions