SI100B Homework 3: Symbolic Expression Evaluator

- **Authors**: Jianwen Luo luojw@shanghaitech.edu.cn, Jiadi Cui cuijd@shanghaitech.edu.cn, Zhijie Yang yangzhj@shanghaitech.edu.cn.
- **Supervised, proofread, edited and approved** by Prof. Yue Qiu qiuyue@shanghaitech.edu.cn.
- **Proofread and calibrated** by Ziqi Gao gaozq@shanghaitech.edu.cn and Qifan Zhang zhangqf@shanghaitech.edu.cn.

- **Release Date**: Apr. 20, 2020
- **Deadline**: 23:59:00 May 7, 2020, China Standard Time (UTC+8:00)
- **Last Modified**: Apr. 13, 2020

Introduction

- *No meaningless stories here. - cuijd*
- *But a story with no word is the longest one. - luojw*
- *Then it can be an idea to have it without a story. - yangzhj*

The ability to evaluate symbolic expressions is of central importance in mathematical softwares such as Mathematica and Matlab. In this homework, you are required to implement an extremely simplified symbolic expression evaluator. It includes some functions like symbolic arithmetic, which can be seen as a powerful scientific calculator. As the inputs are written in a home-made toy language, Python infrastructures cannot automatically deal with them, which means you need to parse the input first and then make further evaluations. Here are some preliminary concepts.

Getting Started

Please simply fork the repository on GitLab and follow the structure and submission guidelines below and on Piazza.

Remember to **make your repository private** before any commits.

Note: Markdown text with file extension ***.md*** could be displayed properly using plug-ins in your browsers, IDEs or specialized markdown editors (like typora).

Repository Structure

README.md

Homework description and requirements.

sample_inputs/ and sample_outputs/

Some sample test cases for you to understand the whole homework. They are in the format of `input{num}.txt` and `output{num}.txt`.

For one `input{num}.txt`, the output of your program (i.e. `output.txt`) should have **no** difference with `output{num}.txt`. In the other word, when you use `diff` command to check the difference between these two files, there should be no output in your terminal.

main.py

Template for Homework 3. You need to implement the program as instructed in readme.

You should **NOT** submit files that are not mentioned in *Repository Structure*. You are encouraged to make use of `.gitignore` to avoid unexpected submissions.

Submission

You should check in main.py to GitLab.

First, make a commit and push your files. From the root folder of this repo, run

```
git add main.py
git commit -m '{your commit message}'
git push
```

Then add a tag to create a submission.

```
git tag {tagname} && git push origin {tagname}
```

You need to define your own submission tag by changing `{tagname}`, e.g.

```
git tag first_trial && git push origin first_trial
```

Please use a new tag for every new submission.

Every submission will create a new Gitlab issue, where you can track the progress.

Regulation

- You may **not** use third-party libraries.
- No late submissions will be accepted.
- You have 30 chances of grading (i.e. `git tag`) in this homework. If you hand in more than 30 times, each extra submission will lead to 10% deduction. In addition, you are able to require grading at most 10 times every 24 hours.
- We enforce academic integrity strictly. If you participate in any form of cheating, you will fail this course immediately. **DO NOT** try to hack Gitlab in any way. You can view the full version of the code of conduct on the course homepage: https://si100b.org/resource-policy/#policies.
- If you have any questions about this homework, please ask on Piazza first so that everyone else could benefit from the questions and the answers.

Specification

The specification is given in **Backus normal form (BNF)**. `BNF` is a widely accepted notation for stating grammars formally. In this homework, specification of the inputs is described in a dialect of BNF. If you have difficulty reading BNF notations, you may first go straight ahead to the input examples for an intensive understanding of the inputs and then head back for more precise

details.

Input Format Specification

Though this homework is divided into several tasks with simpler or harder input restriction, the inputs of the tasks all share the following specification:

A `program` which is expressed as the following will be offered in the input (You should omit '\r', '\n', '\t' in the input. Multiple whitespaces should be treated as if there is only one.):

```
alphabetic = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" |
             "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" |
             "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z";
nonzerodigit = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;

identifier = alphabetic, {alphabetic | digit};

dot = "."
digit = "0" | nonzerodigit;
digits = "0" | nonzerodigit, {digit};

uint = digits;
int = [-], uint;
ufloat = digits, [dot, {digit}];
float = [-], ufloat;

lpar = "(";
rpar = ")";

apply = "apply";
numi = "numi"
numf = "numf"
lambda = "lambda";

unary = "neg" | "cos" | "sin" | "exp";
binary = "+" | "-" | "*" | "pow";

numconstexpr = lpar numi int rpar | lpar numf float rpar;
unaryconstexpr = lpar unary constexpr rpar;
binaryconstexpr = lpar binary constexpr constexpr rpar;
constexpr = numconstexpr | unaryconstexpr | binaryconstexpr;

unaryvarexpr = lpar unary identifier rpar | lpar unary varexpr rpar;
binaryvarexpr = lpar binary identifier identifier rpar |
                lpar binary constexpr varexpr rpar |
                lpar binary varexpr constexpr rpar |
                lpar binary varexpr varexpr rpar;
varexpr = unaryvarexpr | binaryvarexpr;

plainexpr = constexpr | varexpr;

lambdaexpr = lpar lambda identifier identifier rpar | lpar lambda
identifier plainexpr rpar | lpar lambda identifier lambdaexpr rpar;
applyexpr = lpar apply identifier constexpr lambdaexpr rpar | lpar apply
identifier constexpr applyexpr rpar | lpar apply identifier constexpr
constexpr rpar;

```

```
program = constexpr | lambdaexpr | applyexpr;
```

with the following constraints:

1. All the `int` and `float` literals presented in the input are restricted to be free of `-` (negative sign).
2. All the `identifier`s presented in the operands of an `lambdaexpr` are bounded.
3. All the `identifier`s presented in the operands of `lambdaexpr`s are distinct; in other words, different `lambdaexpr`s have different `identifier`s.

For the output, here are the rules you should and should only consider:

1. Any `float` presented in the output should be in the `%.5f` format (i.e. keep 5 decimals).
2. You should fold all `constexpr`s into `numconstexpr`s.
3. `neg`,`+`, `-`, `*` in `constexpr`s should give `numi` result unless there present any `numf` values as their operand(s).
4. The identifier of an `applyexpr` may or may not appear in its operands. Once there is a `lambdaexpr` sharing the same `identifier` with an `applyexpr`, the `lambdaexpr` should be evaluated as substituting all that identifier with the constant indicated by the `applyexpr` from its operand.
5. You should **not** fold the `identities` such as folding `(lambda x (neg (neg x)))` to be `(lambda x x)`.
6. Though can be inferred from the semantics, we'd like to emphasize it here: ***a correct output can only be either a `numi` expression, a `numf` expression or a lambda expression***.

Task 1: Chained Variable Scope (10%)

In this task, you should implement a Python `class` called `chained variable scope`. **This task does not involve file IO**, you only need to implement the APIs correctly. A template named `class EvaluationContext` has been provided in the template.

Supposing in a single scope, you may store and load values with certain names. The problem is, if you want to modularize your code, it would be quite upsetting if you found the function you call ruins your whole program only because a variable in the function has the same name as that of another variable in the caller function.

Now consider a stack of scopes. You push an empty scope into the stack when you enter a new environment and pop it when you leave there. You store the values only into the scope at the top but load the one with the correct name from the scope closest to the top. Now you do not need to worry about problems caused by coincided names of variables!

Here are the APIs you should implement:

- `__init__(self, prev)`: This is where you initialize your scope stack. `prev` is the previously tracked scopes. Add any other member fields here if you like. You should **NOT** modify the argument list.
- `store(self, name, value)`: Associate `value` to `name` in the scope at the top.
- `load(self, name)`: If any value is associated to `name` so far in any of the scopes in the stack, return the value in the scope closest to the top; otherwise, return `None`.
- `push(self)`: Return another `EvaluationContext` which tracks all the scope `self` is tracking in a consistent order but with a new empty scope at the top.
- `pop(self)`: Return another `EvaluationContext` which tracks almost all the scope `self` is tracking in a consistent order but abandon the scope at the top of the scopes of `self`. Cases where calling `pop` with an empty scope will not be covered in the grading testcases.

Task 2: Constant Expressions (20%)

Feel free to take your first step to implement a symbolic evaluator: implement a numerical calculator. In this task, you are required to correctly evaluate:

- The numerical literals `numi` and `numf`. Input example: `(numf 2.00)`. Notice that in the input the numbers involved in these two operators are free of negative sign so you can spend less effort on parsing the numerical literals.
- Unary operators `neg` (negate), `sin` (sine), `cos` (cosine), `exp` (exponential). Input example: `(neg (numi 1))`. Notice that for `neg` unless a `numf` appears as an operand, it should return a `numi`. `sin`, `cos`, `exp` should always return a `numf`.
- Binary operators `+` (addition), `-` (subtraction), `*` (multiplication), `pow` (power). Input example: `(+ (numi 1) (numf 2))`. Notice that unless a `numf` appears as an operand of these operators, they should return a `numi`. Division is not involved so do not worry about division by zero. Ill inputs like 0^0 or $(-1)^{0.5}$ will not appear in the testcases.
- Any of their legal combinations not mentioned above. For example, `(+ (sin (numf 1)) (neg (numi 2)))`.

Input / Output Requirement

You should load the input from `input.txt`. This file will be filled with an input with its format given in the EBNF above. The character set is ASCII.

You should store your output to `output.txt`. The output should also obey the same format as that of the input. The character set is ASCII.

Task 3: Lambda and `apply` Expressions with One Variable (30%)

A basic symbolic evaluator should be capable of handling expressions with one single variable. In this task, you are required to correctly evaluate:

- Lambda expression with one variable. These are lambda expressions with their `contents` free of lambda expressions. Input example: `(lambda x x)`. They should be left invariant if they are not included by an `apply` expression with its identifier even if its 'content' does not contain any variable. **Be careful**: the evaluation result can be a lambda expression. For example, `(lambda x x)` should be evaluated to be `(lambda x x)`. You **do not need to** and you **should not** consider simplifications on some identities like evaluating `(lambda x (neg (neg x)))` to be `(lambda x x)`, just leave it as `(lambda x (neg (neg x)))`.
- Apply expression with one variable. Input example: `(apply x (numi 1) (lambda x x))`. Notice that the identifier of the `apply` operator can be different from the identifier of any of the contained lambda expressions: when this happens, eliminate this `apply` operator and continue evaluating on its `contents`. In this task, values bound to the variable are given only by `numi` or `numf` expressions. The evaluation result of the given example should be `(numi 1)`.

Input / Output Requirement

You should load the input from `input.txt`. This file will be filled with an input with its format given in the EBNF above. The character set is ASCII.

You should store your output to `output.txt`. The output should also obey the same format as that of the input. The character set is ASCII.

Task 4: Multi-Variable Expressions(40%)

Now here comes a more practical challenge: dealing with multi-variable lambda expressions and `apply` expressions. Multi-variable is achieved by layering these two operators in a curried manner:

- The `content` of an `apply` expression is now allowed to be an `apply` expression. The identifiers of the `apply` operators are promised to be distinct. Input example: `(lambda x (lambda y (lambda z (+ x y))))`
- The `content` of a lambda expression is now allowed to be a lambda expression. All the identifiers of the layered lambda expressions are promised to be distinct. Input example: `(apply x (numi 1) (apply u (numf 2) (lambda xy xy)))`. In this task, values bound to the variables are given only by `numi` or `numf` expressions.

Input / Output Requirement

You should load the input from `input.txt`. This file will be filled with an input with its format given in the EBNF above. The character set is ASCII.

You should store your output to `output.txt`. The output should also obey the same format as that of the input. The character set is ASCII.

Bonus: Apply Values to Variables using Expressions (20%)

Tough problems are often described using only short words. In this task, values bound to the variables in `apply` expressions can be any valid constant expressions free of `apply` and `lambda`. Input example: `(apply x (cos (numf 3.14159)) (lambda x (+ x (numi 1))))`. Be brave!

Input / Output Requirement

You should load the input from `input.txt`. This file will be filled with an input with its format given in the EBNF above. The character set is ASCII.

You should store your output to `output.txt`. The output should also obey the same format as that of the input. The character set is ASCII.

Overall Examples

Input:

```
(numi 1)
```

Output:

```
(numi 1)
```

Input:

```
(+ (numi 1) (numf 1))
```

Output:

```
(numf 2.00000)
```

Input:

```
(+ (numi 1) (numi 1))
```

Output:

```
(numi 2)
```

Input:

```
(lambda x (neg (neg x)))
```

Output:

```
(lambda x (neg (neg x)))
```

Input:

```
(apply y (numi 1) (lambda x x))
```

Output:

```
(lambda x x)
```

Input:

```
(apply x (numi 1) (lambda x x))
```

Output:

```
(numi 1)
```

Input:

```
(apply x (numi 1) (apply y (numi 1) (lambda x (lambda y (+ x y)))))
```

Output:

```
(numi 2)
```

Appendix and Hints

Starting from this homework, you will find yourself into the field of theoretical computer science. To begin with coding, you need to have a clear abstract of the data structures and the algorithms to be used in this homework.

Handling input

When you have successfully coped with reading from files, the very first thing you need to do (which is pretty the same as previous homework) is parse the input.

Parsing

In this homework, you need to parse the statements or expressions in the input according to the input specification by yourself. As Python can not understand these non-Python expressions or commands (such as `apply`), it is necessary to do a formal analysis about these expressions, obtain the vital information or relationship about them and do further evaluation.

In the parsing, you need to mainly implement three classes, which are `Syntax`, `ASTNode` and `AST` class. (You may also derive subclasses from them instead.) Here `AST` means Abstract Syntax Tree.

It is recommended to build the parsing pipeline like the following:

```
characters -> syntaxes -> AST
```

A possible classification for syntaxes can be the following:

```
LPAR, RPAR, INTEGER(ivalue), FLOAT(fvalue), TERM(str)
```

For example, for the input `(lambda x x)`, you have 12 characters (including spaces), you can extract symbols `(`, `lambda`, `x`, `x` and `)`, a correct interpretation will leads to syntaxes like `LPAR`, `TERM("lambda")`, `TERM("x")`, `TERM("x")` and `RPAR`. Observing the syntaxes, each time you see a `LPAR`, then you know the next syntax will indicate the way you should follow to interpret the following syntaxes **until you meet the corresponding** `RPAR`. Still using this example, *in mind* you can extract an AST like `LAMBDAEXPR(identifier=IDENTIFIER("x"), body=IDENTIFIER("X"))` or any other representation you would prefer. Another example can be a syntax sequence `LPAR TERM("+") LPAR TERM("numi") INTEGER(1) RPAR LPAR TERM("numf") INTEGER(1) RPAR RPAR` will lead you to an AST like `AddExpr(lhs=NUMI(value=1), rhs=NUMF(value=1.0))`. Notice that these representations just give you intuition in mind, **your output should follow the format given in the Format Specification**.

You may have already realized that `AST` contains many layers of and various kinds of `ASTNode`s, a recursive parsing scheme is strongly recommended.

Expressions

The expressions can be categorized into three types --- `constant expressions`, `lambda expressions` and `apply expressions`. All the expressions given follow the fashion of Polish Notation, which is to say, the operator is in preceding of its corresponding operand(s). To evaluate a PN expression, the most elegant method is to iterate it through its tail to its head,

treating it as an Reverse Polish Notation. You may opt this solution but this can be a tough job dealing with apply expressions. Another solution which may perfectly fit this homework as all the parentheses is given, is recursively calling the evaluate function until the innermost parentheses pair is met and the value within the pair is evaluated, then returning it to the outer parentheses pair.

Polish Notation (PN)

`PN` is also known as prefix notation. For example, the expression for adding two numbers `a` and `b` is written in Polish notation as `+ a b`, instead of `a + b`. Obviously, operators precede their operands in the Polish notation. Polish notation, as well as reverse Polish notation (RPN), is in common use in computer science, since the mathematical expressions with them can be more easily parsed into abstract syntax trees (ASTs). Here, we do **not** pay too much attention to the general theory, because each expression involved in this homework is separated by parentheses.

Constant expressions

Constant expressions can be further divided into types with unary operators, binary operators, and numerical literal expressions. Constant expressions usually have the highest precedence to be evaluated, and serves as constant numbers in formulas in real life.

Lambda expressions

Lambda expressions represent functions with exactly one input and one return value. A lambda expression without an apply expression assigning its input value should return the same value as its literal.

Lambda calculus

Lambda calculus is one of the elementary concepts of symbolic computations. It expresses an abstraction of functions in mathematical logics, which abstracts functions with 'anonymous' and 'single input' characteristics. 'Anonymous' makes the effect of the function irrelevant to its name, but just the input and output expressions. 'Single input' ensures that each lambda function only has one input. If a second input is required, then another lambda expression should be nested into the first one. See Motivation section for examples.

Apply expressions

As mentioned above, each lambda expression accepts only one input. To express a function with multiple inputs, the currying technique is often used. The transformation that gives opposite effect of `currying` is called `apply`, which applies values to the arguments of lambda functions. When implementing the evaluation for `apply` expressions, you need to be careful about the semantic of its three parameters --- the first one determines which argument in the follow lambda expression is to be assigned with the value. The value to be assigned is placed as the second parameter and the lambda expression is on the third place.

Currying

Currying provides a way to represent multi-input functions with single-input functions, which is used to convert a multi-input function into a sequence of functions that each takes exactly one input and evaluates that sequence of functions. In this homework, this is achieved by involving a lambda expression in another one but with distinct identifiers for their variables. The input containing lambda expressions will be given in curried formats. Example: When taking `(apply x (numi 1) (lambda x x))` as input, your evaluator should return `(numi 1)` as the result.

Data structure

As you have had the basic idea of what are the expressions to be handled in your homework, let's talk about the data structure you need to implement in this homework. It is something like an Abstract Syntax Tree (you can modify it as your needs), which gives a clear abstraction of the relationships among the operators, operands, and other expressions and their parameters in the given symbolic expression. To have a clearer understanding (and the illustration) of AST, you may refer to the wikipedia page linked below.

Abstract syntax tree (AST)

AST represents the syntactic structure of source codes regardless of the detailed syntax. It can contain variable types, execution orders, identifier and value pairs, and the sides of operands with respect to the binary operators. Each node in AST contains one of the aforementioned four elements. You will need to rephrase each given expression into an AST instance. When evaluating an expression with its AST, the nodes should be visited in a specific order to ensure consistence with the given expression's semantics. Note that even though you can give some of the answers correctly without actually building a fully functioning AST ahead, you are **required** to evaluate the expression with its AST representation.

Programming paradigm

Object-oriented programming (OOP)

You are **required** to use OOP in this homework. OOP abstraction provides a way to pack and isolate data and algorithms. Using OOP correctly can greatly improve the coding efficiency when, for example, dealing with recursive data structures like ASTs.

For this homework, for example, you may implement the abstract syntax tree (discussed in the next section) using layered abstractions based on OOP. Crazy stuff like combinatoric parsers using F-(co)algebra or (co-)monads is also welcomed.