K. (Kengdie) Mua(II) - Expression Evaluation

In this problem-series, you're to implement a subset of the Lua language (version 5.1), called mini-lua (mua). This is one of Rujia Liu's experimental languages, mainly for implementing algorithms, not real-world programs.

This is the second problem in the series, which requires you to write an expression evaluator. The grammar below is described in extended BNF, in which {x} means "x appears one or more times", [x] means "x appears 0 or 1 time", and x | y means "either x or y (but not both) appear exactly 1 time".

Types

mini-lua is a dynamically typed language. This means that variables do not have types; only values do. There are no type definitions in the language. All values carry their own type.

There are six basic types in Lua: nil, boolean, number, string, function, table.

- Nil is the type of the value nil, whose main property is to be different from any other value; it usually represents the absence of a useful value.
- Boolean is the type of the values false and true.
- Number represents real (double-precision floating-point) numbers. Internally, numbers are represented by IEEE-754 numbers (conventional double variable in C/C++ or Java). Note that integers are also stored in this way. It's no big deal, because IEEE-754 can represent integers precisely, as long as you don't perform inaccurate operations (i.e. divisions).
- String represents arrays of ASCII (8-bit) characters.
- Function is function. It doesn't have a name (though the variable holding the function has a name).
- Table implements associative arrays, that is, arrays that can be indexed not only with numbers, but with any value (except nil). Tables can be heterogeneous; that is, they can contain values of all types (except nil).

Tables and functions values are objects: variables do not actually contain these values, only references to them. Assignment, parameter passing, and function returns always manipulate references to such values; these operations do not imply any kind of copy.

To lua programmers: Lua has two more basic types: userdata and thread, both are not supported in Mini-Lua. Mini-lua has very limited support for FP (functional programming). For example, lambda expression is not supported, and you cannot return a function or closure in a function.

Variables

A variable defined as follows:

```
var -> NAME { `.' NAME | `[' expr `]' }
```

Here "`expr`" means any valid expression, because tables can be indexed with any value. Note that the "dot" syntax is a syntactic sugar that makes the expression "look like accessing object member". For example, `a.name` is equivalent to `a["name"]`.

Variables have values of nil by default. If you assignment nil to a variable, you're removing that variable. Similarly, you can assign a nil to an element in a table to remove it from the table.

To lua programmers: Lua provides more syntactic sugars, `NAME `:' NAME args`. Let's ignore these.

Simple Expressions

An expression consists of so-called "simple expressions":

```
simpleexp -> NUMBER | STRING | nil | true | false | `{' '}' | var |
functioncall
```

Here `{' '}' means an empty table.

To lua programmers: Lua provides more convenient ways to construct tables. However, they add complexity to the language so we don't use them. Moreover, mini-lua does not support lambda expressions.

Function calls

The functioncall expression in the last section is defined this way:

```
functioncall -> var `(' [ expr {`,' expr } ] `)'
```

To lua programmers: In mini-lua, the only way to call a function is directly specifying the variable holding that function. For example `(print)(1)` won't work, but `a = (f)` is allowed, because `(f)` is a valid expression (see below). In Lua, if you call a function with a string constant or table constructor, the parenthesis' could be omitted. This syntax is not supported in mini-lua.

Expressions

Now we have all the building blocks. We use operators to combine simple expressions into complex expressions (i.e. "expr"):

```
expr -> simpleexp | expr binop expr | unop expr | `(' expr `)'
```

Note that this grammar does not consider operator precedence's, which is summarized in the following table (from low to high):

```
or
and
<     >     <=    >=    ~=    ==
..
+     -
*     /     %
not   #     - (unary)
^
```

Most of them are common-sense, but there are several things to mention:

- "not equal" is not "<>" or "!=". it is "~="
- "%" (mod) is defined on real numbers. it's equivalent to `a - math.floor(a/b)*b`
- "^" (power) is the only right-associative operator.
- "`not`", "`and`" and "`or`" always returns false or true. Only "`and`" and "`or`" are short-cut operators.
- Concatenation is "`..`", not "+".

- operator # is getting the length of a table or a string. The length of a table `T` is the maximal non-negative integer `n` such that `T[1], T[2], ..., T[n]` all exists (i.e. not nil). In this way, each table `T` can be regarded as an array, having `#T` elements.

To lua programmers: In lua, concatenation is right associative (See: http://lua-users.org/wiki/AssociativityOfConcatenation), but for this problem, whether it is left or right associative doesn't affect the result. Morever, "and" and "or" do not always return boolean values in Lua, but we restrict the result to boolean values in mini-lua in order to simplify the language. Maybe the most import change is: No automatic conversions between strings and numbers (i.e. no coercion).

Library functions

For testing purpose, here are some functions you should implement (extracted from http://www.lua.org/manual/5.1/manual.html , with simplifications):

- `tonumber(e)`

Tries to convert its argument to a number. If the argument is already a number or a string convertible to a number, then tonumber returns this number; otherwise, it returns nil. Note that there is no optional parameter as in Lua.

- `tostring(e)`

Receives an argument of any type and converts it to a string in a reasonable format. In mini-lua, if `e` is a function or a table, return `"function"` or `"table"` instead. If e is a number, convert it to a string using the traditional C format string `"%.14g"` (it's the default format string in lua 5.1, see luaconf.h). Don't worry you're using other programming languages and do not know the exact semantics of the format string, you can choose your own way. We're not testing the strict format in the judge data.

- `print(e)`

Prints `tostring(e)` to standard output, followed by a newline. Note that in mini-lua, this function can only print one value.

- `string.rep(s,n)`

Returns a string that is the concatenation of `n` copies of the string `s`.

- `string.sub(s, i, [,j])`

Returns the substring of `s` that starts at `i` and continues until `j`; `i` and `j` can be negative. If j is absent, then it is assumed to be equal to -1 (which is the same as the string length). In particular, the call string.sub(s,1,j) returns a prefix of s with length j, and string.sub(s, -i) returns a suffix of s with length i.

- `table.concat(table, [,sep])`

Given an array where all elements are strings, returns `table[1]..table[2]..` ... `table[#table]`. The default value for `sep` is the empty string.

- `table.sort (table [, comp])`

Sorts table elements in a given order, in-place, from `table[1]` to `table[n]`, where `n` is the length of the table. If `comp` is given, then it must be a function that receives two table elements, and returns `true` when the first is less than the second (so that no `comp(a[i+1],a[i])` will be true after the sort). If `comp` is not given, then the standard operator `<` is used instead. Note that the sort algorithm is not stable, but we're carefully designing the test data so that how the sort is implemented does not affect the final.

- `math.abs(x), math.floor(x), math.ceil(x)`

Returns the absolute value of `x`, the largest integer smaller than or equal to `x`, and the smallest integer larger than or equal to `x`, respectively.

- `math.sqrt(x), math.exp(x), math.log(x), math.log10(x)`

Returns the square root of `x`, the value e^x, $\ln(x)$ and $\log_{10}(x)$.

- `math.pi, math.rad(x), math.deg(x)`

`math.pi` is a variable holding the value of pi, while the other two functions converts between degrees and radians.

- `math.acos(x), math.asin(x), math.atan(x), math.atan2(y,x)`

Returns the value of math functions, in radians.

- `math.cos(x), math.sin(x), math.tan(x)`

Returns the value of math functions, assuming `x` is in radians.

- `math.min(x,y), math.max(x,y)`

Returns the smaller/larger value of `x` and `y`.

Input

There will be multiple mini-lua programs. Each program consists of lines. Each line is either in the form `print(expr)`, or in the form `var = expr` (indicates an assignment). All the expressions will obey the rules above. An empty line terminates a program (all the variables should be reset to nil). The expressions will be correct and evaluates to a reasonable value (for example, you don't have to handle NaN or arithmetic overflows, and you will not be asked to compare a number with a string). We will not re-assign these functions/variables, though we may assign them to new variables. There will be no comments in the program.

Output

For each line in the form `print(expr)`, print the expression. When printing numbers, print as many digits as you like, as long as the relative error OR the absolute error is no more than 1e-9.

Sample Input

```
print(1+2*3/4)
f = math.sin
print(f(1.57))
print(1<2 and 4+5==9)
print(math.max(3,-math.min(5*7,-4)))
a={}
a[1]={}
a[1][2]={}
a[1][3]="hehe\'\\"
a[1][1]=a
print(#a)
print(#a[1])
print(#a[1][3])
print(a)
print(f)
print(a[1][1][1][3] .. "\n" .. "..")
```

Output for Sample Input

```
2.5
0.99999968293183
true
4
1
3
6
table
function
hehe'\
..
```

Hint

If you want to learn from the source code of official lua compiler, download the 5.1.4 version and go straight to the "subexpr" function in lparser.c (you can see the precedence table right before it).