L. (Last) Mua (III) - Full Interpreter

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 third (and last!) problem in the series, which requires you to write a full interpreter. Make sure you've solved the first two problems before attempting this problem. You may have trouble understanding this problem if you haven't done so.

Chunk

A chunk (a program or a piece of code that are run as a whole) is a single block:

```
block -> {stat EOL} [laststat EOL]
laststat -> return [expr] | break
```

Note that you can use break only as the last statement in a block (to keep things simple, lua even doesn't support continue!)

To lua programmers: In lua, you can write two statements in one line, even *without* a semicolon. In mini-lua, you can't write multiple statements in a line. Actually semicolon cannot be used as a statement delimiter. Moreover, a block can only return one value.

Simple Statements

The following types of statements are easy to understand:

- Empty statement: `stat ->`
- Function call, then discard the result: `stat -> functioncall`
- Assignment: `stat -> var `=' expr`
- Block statement: `stat -> do block end`

To lua programmers: Multiple assignment (like `a,b=b,a`) is not supported in mini-lua. No tail-recursion optimization is necessary in this problem.

Control Flows

While, repeat and if statements all use conditionals. Both nil and false make a condition false; any other value makes it true. These statements are defined as follows:

- While statement: `stat -> while expr do block end`
- Repeat statement: `stat -> repeat block until expr`
- If statement: `stat -> if expr then block { elseif expr then block } [else block] end`

And there are two kinds of for loops. The first one is:

```
stat -> for NAME `=' expr `,' expr [`,' expr] do block end
```

The three expressions in this loop are the initial value, the upper limit (when step > 0) / lower limit (when step <= 0), and the step (default: 1). All three expressions are evaluated exactly once, converted to number (using `tonumber(e)` function), before the loop starts. All three expressions must all result in numbers (`tonumber(e)` should not return nil). Note that `NAME` is local (you cannot access it after the loop ends), and you can use `break` to exit the loop, but there is no `continue` statement.

The second one is a more general iteration, looping for all the keys in a table:

```
stat -> for NAME in iterator do block end
```

Here `iterator` is either `ipairs(table)` or `pairs(table)`(of course the actual variable being iterated can have other names other than `table`). The difference is: `ipairs` loops from `1` to `#table`, but `pairs` loops for all the keys in table, in no particular order.

Function definition

Given other building blocks discussed above, function definitions are quite simple:

```
stat -> function NAME `(' [NAME {`,' NAME} ] `)' block end
```

Note that all the parameters are local variables. As discussed before, you can use `return` statement to exit a function, carrying a single return value if you like. You can only use `return` in the last statement of a block, so if you want to exit a function in the middle, you can wrap it in a block, like `do return end`. It means to exit from the inner-most function that includes the block.

Scoping and Visibility

You can declare a local variable this way:

```
stat -> local NAME [ `=' expr ]
```

Like Lua, mini-Lua is a lexically scoped language. The scope of variables begins at the first statement *after* their declaration and last until the end of the innermost block that includes the declaration. If there is another variable having the same name in an outer block, that variable is shadowed (it still exists, but you cannot access it). Only after the inner block ends, we regain the access to it, because the inner variable no longer exists. Recall that a chunk is also a block, so you can also declare local variables in a trunk.

Note that the local variable is accessible only after the declaration, so you can use `local x = x` to declare a local variable called `x`, initialized with the value of outside variable whose name is also `x`.

Note that in the **repeat-until** loop, the inner block does not end at the **until** keyword, but only after the condition. So, the condition can refer to local variables declared inside the loop block.

Because of the lexical scoping rules, local variables can be freely accessed by functions defined inside their scope. However, to keep things simple, all the functions will be declared globally (not nested in another function or a block), and local variables in the chunk (if any) are always declared after the functions.

Input

There will be multiple mini-lua programs. Each program starts with a line starting with `--PROGRAM` (it will not appear inside a program).

Output

For each program, print the test case number and the output from the program. Print an empty after each test case.

Sample Input

```
-- PROGRAM: Eight-Queen problem solver in Mini-Lua
```

```
function dfs(d)
  if d == n then
    cnt = cnt + 1
  else
    for i = 1, n do
      if (not vis[i]) and (not vis2[d-i]) and (not vis3[d+i]) then
        vis[i] = true
        vis2[d-i] = true
        vis3[d+i] = true
        dfs(d+1)
        vis[i] = nil
        vis2[d-i] = nil
        vis3[d+i] = nil
      end
    end
  end
end

vis = {}
vis2 = {}
vis3 = {}
cnt = 0
n = 8
dfs(0)
print(cnt)

-- PROGRAM: Scoping and Visibility rules
x = 10
do
  local x = x
  print(x)
  x = x+1
  do
    local x = x+1
    print(x)
  end
  print(x)
end
print(x)
```

Output for Sample Input

```
Program 1:
92

Program 2:
10
12
11
10
```

Hint

Haven’t you noticed that mua is in LL(1)? A recursive decent parser is enough for this problem.