## Task 3. LOLCODE: I CAN HAS TRANLASHUN?

### Available marks: 20

In 2007 Adam Lindsay from Lancaster University invented ***Lolcode***, a procedural programming language that uses phrases from the lolcat meme as keywords. It has many variants, and has been further adapted for Progcomp to keep the task manageable.

source: Jerry7171/Toshio Yamaguchi CC-BY-SA-2.0, via Wikimedia Commons

Your task is simple, at least in principle: write a Lolcode interpreter. Your program takes a .lol file that conforms to the language definition below, and maps the code into your favourite conventional programming language (python, perl, Lua and C are all suitable). This target file is then executed directly.

### Lolcode definition

The brief description below uses the following notation:

| | |
|---|---|
| `<text>` | any text up to end-of-line |
| `<var>` | variable (or parameter) name |
| `<name>` | function or loop name |
| `<value>` | literal number or variable |
| `<stringliteral>` | quoted string in the target language |
| `<arithexpr>` | arithmetic expression |
| `<boolexpr>` | Boolean expression |
| `<statements>` | zero or more well-formed statements |

#### File structure

- The file is line-structured, that is, each designated part of a statement occurs on a single line.
- Empty lines are ignored.
- The program comprises, in order, the keyword HAI, declarations, statements and function definitions, and finally the keyword KTHXBYE.
- Lines are indented using any number of spaces, all other separators use a single space character.
- Keyword phrases are always in UPPER CASE
- Comment statements are either single line: `BTW <text>` or multiline as shown at right. Comments should appear in the target file.
- Comments can be added to a statement using a space, a comma, space then `BTW` and any text.
- Variable and function names begin with a lower case letter, followed by any number of lower case letters, digits or underscores.
- Function definitions, if any, are placed after the main statements.

```
HAI

OBTW
Trivial program,
even simpler than
  "HAI WORLD!"
TLDR

KTHXBYE
```

In the syntax below [ ... ] indicates an optional construct and { ... } zero or more instances of a construct.

**Output, Input**

```
VISIBLE <var>
VISIBLE <stringliteral>

GIMMEH <var>
```

The last one reads a line of input, converts to a number and assigns to the variable.

**Variables**

Variables can store floating point values only, no other types and no arrays. They are declared and optionally initialised using one of:

```
I HAS A <var>
I HAS A <var> ITZ <value>
```

Normal assignment uses the R operator:

```
<var> R <arithexpr>
```

Variable names are *assumed not to clash* with keywords or standard identifiers in the target language. If they do you are permitted to change the lolcode source to avoid the clash.

```
HAI

I HAS A animal ITZ 42
I HAS A cat
cat R animal
VISIBLE cat
VISIBLE "\n"

KTHXBYE
```

**Arithmetic expressions**

There is one unary operator and four binary ones. All operators are prefix, which makes parsing easier. Simple operands are variables or literal numbers.

```
cat R FLIP PRODUKT OF -2.5 AN (: SUM OF animal AN 1 :)
```

| | |
|---|---|
| `FLIP <arithexpr>` | Change sign |
| `SUM OF <arithexpr> AN <arithexpr>` | Addition |
| `DIFF OF <arithexpr> AN <arithexpr>` | Subtraction |
| `PRODUKT OF <arithexpr> AN <arithexpr>` | Multiplication |
| `QUOSHUNT OF <arithexpr> AN <arithexpr>` | Division |

If more than one binary operator is used, all nested expressions *must be parenthesised* with `(: ... :)`.

**Conditional statement**

```
IZ <boolexpr> ?
  YA RLY
    <statements>
{ MEBBE <boolexpr> ?
    <statements> }
[  NO WAI
    <statements> ]
OIC
```

```
BTW signum calculashun
IZ SMALLR val AN 0 ?
  YA RLY
    sign R -1
  MEBBE SMALLR 0 AN val ?
    sign R 1
  NO WAI
    sign R 0
OIC
```

**Boolean expressions**

The relational operators are

| | |
|---|---|
| `BOTH SAEM <arithexpr> AN <arithexpr>` | Equal to |
| `SMALLR <arithexpr> AN <arithexpr>` | Less than |
| `SAEM OR SMALLR <arithexpr> AN <arithexpr>` | Less than or equal to |

The Boolean operators are

| | |
|---|---|
| `NOT <boolexpr>` | Complement |
| `BOTH OF <boolexpr> AN <boolexpr>` | And |
| `EITHER OF <boolexpr> AN <boolexpr>` | Or |

As with arithmetic operators, *nested expressions are parenthesised* with `(:` ... `:)`.

**Loops**. Lolcode has a single for/while loop:

```
IM IN YR <name> [ UPPIN YR <var> [ ITZ <value> [ TIL ITZ BIGGR DEN <value> ]]]
  <statements>
    GTFO  , BTW break statement
IM OUTTA YR <name>
```

The loop name is ignored. An incrementing variable (local to the loop) is optional, as are the initialiser and limit.

**Functions** are defined with

```
HOW U MAEK <name> [ OF YR <var> { AN YR <var> }]
  <statements>
  FOUND YR <arithexpr>
IF U SAY SO
```

```
HOW U MAEK max OF YR a AN YR b
  IZ SMALLR a AN b ?
    YA RLY
      FOUND YR b
    NO WAI
      FOUND YR a
  OIC
IF U SAY SO
```

Variables are local to the function.

In this dialect functions are called by assigning the result to a variable:

```
<var> IZ TEH <name> [ OF <arithexpr> { AN OF <arithexpr> }]
```

## Hints

- The translation can be done line-by-line with rewriting rules, there is no need to store a representation of the program as a data structure as a compiler might do.
- Parse nested expressions by working from the inner parentheses outward.
- You can assume the lolcode program is legal, no error checking is needed.

## Test Programs

There are three lolcode program files you can test your program with. They are (marks in parentheses):

| | |
|---|---|
| `BMI.lol` (5) | Body Mass Index calculator. Uses input, output, non-nested expressions and conditionals only. |
| `fib.lol` (6) | Fibonacci numbers. Uses a loop, simple expressions, no functions. |
| `sqrt.lol` (9) | Square root calculator using the Newton-Raphson iterative formula. Nested expressions, loops, conditionals and functions. |