Интерпретатор за функционалния език **ListFunc**

Интерпретаторът трябва да може да работи в интерактивен режим, в който позволява на потребителя да пише ред код, който се оценява и се извежда резултат от оценката. Също така вашият интерпретатор трябва да може да се стартира върху файлове, които да се изпълняват и резултатът от тях да се отпечатва на изхода на вашата програма.

Нека разгледаме как се дефинира езикът **ListFunc**. В този език има два типа литерали – реални числа <real-number> и списъци <list-literal>. За да е функционален този език, ще трябва да може да дефинираме и изпълняваме функции. Изпълнението на една функция ще става чрез нейното име и списък от аргументи на функцията, заграден в кръгли скоби. Може да има функции без аргументи (с празен списък с аргументи).

```
<real-number> ::= <всички валидни записи на double>
<list-literal> ::= [ [<expression0>, <expression1>, ...] ] *
<expression> ::= <list-literal> | <real-number> | <function-call>
<function-name> ::= <валиден идентификатор в C++>
<function-call> ::= <function-name>([<expression>, ...])
* външните скоби са литерали, вътрешните показват опционални елементи
```

Вашият интерпретатор трябва да предостави следните вградени функции

- **eq**(#0, #1) връща булевата оценка на #0 == #1
  - две числа са равни когато стойностите им са равни
  - два списъка са равни когато съответните им елементи са равни
  - число и списък са равни ако списъка има 1 елемент и той е равен на числото
- **le**(#0, #1) връща булевата оценка на #0 < #1.
- **nand**(#0, #1) връща булевата оценка на !#0 || !#1.
- **length**(#0) връща броя елементи в подадения списък или -1 ако аргумента е число.
- **head**(#0) връща първия елемент на списък.
- **tail**(#0) връща списък от всички без първия елемент на входния списък.
- **list**(#0) връща безкраен списък с начален елемент #0 и стъпка 1.
- **list**(#0, #1) връща безкраен списък с начален елемент #0 и стъпка #1.
- **list**(#0, #1, #2) връща списък с начален елемент #0, стъпка #1, и брой елементи #2.
- **concat**(#0, #1) връща списък, конкатенация на двата аргумента, които са списъци.
- **if**(#0, #1, #2) оценява #0 като булева стойност и връща #1 ако оценката е true или #2 ако оценката е false.
- **read**() връща число, прочетено от стандартния вход.

- **write**(#0) записва #0 на стандартния изход и връща 0 при успех и различно от 0 число при провал.
- **int**(#0) връща #0 без дробната част.

Някои функции като **if** и **nand** не е нужно да оценява всичките си аргументи за да върнат верен резултат.

Добавете и следните аритметични функции върху числа: **add**, **sub**, **mul**, **div**, **mod, sqrt.** Функцията **mod** е валидна само върху цели числа.

Освен това езикът трябва да позволява да се декларират функции - с име и израз съдържащ техните аргументи. Аргументите на функцията ще се дефинират с цяло число, индексът на аргумента, предхождащ се от символа '#'

```
<param-expression> ::= <expression> | #integer | <function-name>([<param-expression>,...])
<function-declaration>::= <function-name> -> <param-expression>
```

Декларацията на функция декларира нова или подменя стара декларация, връща 0 ако е нова декларация, връща 1 ако декларацията подменя стара декларация.

При четене и изпълнение на кода могат да възникнат няколко типа грешки - грешки при използването на недекларирани функции, използването на невалидни символи в имената на функциите, грешки при изпълнение на кода (например деление на нула) или други. За всяка грешка трябва да се погрижите да уведомите потребителя с подходящо съобщение.

Обърнете внимание, че при деклариране и оценяване на функциите е възможно да се получи рекурсия, която трябва да поддържате.

По желание може да поддържате коментари, и възможност за разделяне на сложни изрази с разстояние, табулация и нов ред (както е в последния пример).

Нека разгледаме няколко примера, редовете започващи с > са изход на интерпретатора:

```
32
> 32
[1 2 3]
> [1 2 3]
myList -> [3 4 5 7 9 10]
> 0
myList()
> [3 4 5 7 9 10]
head(myList())
> 3
tail(myList())
```

```
> [4 5 7 9 10]
isOdd -> eq(mod(int(#0), 2), 1)
isEven -> nand(isOdd(#0), 1)
> 0

list(1, 1, 10)
> [1 2 3 4 5 6 7 8 9 10]
list(5, -0.5, 3)
> [5 4.5 4]
list(read(), read(), read())
> read(): 12
> read(): -3
> read(): 4
[12 9 6 3]

list(1)
[1 2 3 4 5 6 7 8 ....

Програма която намира/принтира прости числа

not -> nand(#0, 1)
> 0
and -> not(nand(#0, #1))
> 0

// генерира всички делители които да се проверят за числото #0
divisors -> concat(
    [2],
    list(3, 2,
       add(1, int(sqrt(#0)))
    )
)
> 0

// проверява дали числото #1 има делител в списъка #0
containsDevisors -> if(
    length(#0),
    if(mod(#1,head(#0)), 0, containsDevisors(tail(#0), #1)),
    0
)
> 0

// проверява дали числото #0 е просто
```

```
isPrime -> not(containsDevisors(divisors(#0), #0))
> 0

// #0 list of numbers, leaves only primes
filterPrimes -> if(
    isPrime(#0),
    concat([head(#0)], filterPrimes(tail(#0))),
    filterPrimes(tail(#0))
)
> 0

// всички прости числа до 10
primes10 -> filterPrimes(concat([2], list(3, 1, 7)))
> 0
primes10()
> [2 3 5 7]

// prints primes for
allPrimes -> filterPrimes(concat([2], list(3)))
> 0
allPrimes()
> [2 3 5 7 11 13 17 19 … // принтира прости числа до "безкрай"
```