Down With Abstractions

Lambda calculus is a model of computation based on nothing but anonymous functions. Despite being one of the simplest and most elegant such models, it is perceived as a little too complicated by some. After all, it has function abstraction and variables! Combinatory logic, developed by Schonfinkel and Curry, is an equivalent model of computation that uses nothing but function application and a very small set of pre-defined combinators. Moreover, expressions of lambda calculus can be mechanically transformed to equivalent expressions in combinatory logic using a simple algorithm. Your task is to implement this algorithm.

Combinators

We will use the following combinators for the language of resulting combinatory logic expressions:

```
K = (\x y. x)
```

K is a constant function generator. Given some *x*, it evaluates to a function that evaluates to *x* for all inputs.

```
S = (\x y z. ((x z) (y z)))
```

S is a substitution function.

Note that *S* and *K* alone form a Turing-complete model of computation. But for convenience we will use three other combinators as well.

```
I = (\x. x)
```

Identity function. Extensionally equivalent to *SKK*.

```
B = (\x y z. (x (y z)))
```

B is function composition, aka "*dot*". Extensionally equivalent to *S(KS)K*.

```
C = (\x y z. ((x z) y))
```

C returns a given binary function with its two parameters flipped. Extensionally equivalent to *S(S(K(S(KS)K))S)(KK)*.

B and *C* are specializations of *S*.

Grammar for lambda calculus expressions

We will use a variant of lambda expression syntax for our inputs. The differences from the standard notation are as follows:

1. Lowercase lambda in lambda expressions is represented by a backslash ('`\`').
2. Function applications must be parenthesized.
3. Lambda expressions must be parenthesized.
4. Lambda expressions are extended to easily formulate functions of arbitrary arity (in fully curried form). `(\x y z. ((z y) x))` is equivalent to `(\x. (\y. (\z. ((z y) x))))`

Variable names are represented by arbitrary non-empty sequences of Latin upper- and lowercase letters, numbers and underscores. Other tokens are '`(`', '`)`', '`\`' and '`.`'. Tokens may be separated by arbitrary amount of whitespace without changing the meaning. Whitespace is required in some contexts, as described below.

```
EXP = VAR
EXP = '(' EXP ' ' EXP ')'
EXP = '(' '\' VARLIST '.' EXP ')'
VARLIST = VAR VARLIST0
VARLIST0 = eps
VARLIST0 = ' ' VAR VARLIST0
```

Abstraction elimination algorithm

Use the version with rules for *S*, *K*, *I*, *B*, *C* at:

http://en.wikipedia.org/wiki/Abstraction_elimination#Combinators_B.2C_C

Apply eta-reduction wherever possible:

http://en.wikipedia.org/wiki/Abstraction_elimination#.CE.B7-reduction

Eta-reduction removes unnecessary abstraction, making use of the fact that `(\x. (E x))` is equivalent to `E`.

Free variables in lambda calculus

Performing abstraction elimination requires determining the free variables of a given expression. In lambda calculus, a variable is free in an expression if it's not a parameter in any of the enclosing lambda expressions. The opposite of a free variable is a bound variable. For example:

```
(\x. (z (\y. (y x)))
```

In this expression, `z` is a free variable, while `x` and `y` are bound variables. If we consider just the following sub-expression:

```
(z (\y. (y x))
```

...then only `y` is a bound variable in it. Both `x` and `z` are free variables in this expression.

Input

First line contains number of test cases, 1 <= *T* <= 100. *T* lines follow. Each line contains a lambda expression. Lambda expressions in the test cases contain no free variables at the top level. This ensures that the equivalent combinatory logic expression can be written using nothing but the five combinators listed above.

Output

Output *T* lines with equivalent representations of given lambda expressions in SKIBC-basis combinatory logic.

Sample Input

```
3
(\x. x)
(\test. (\ignored_1. test))
(\x. (\y. (y (\z. (\t. ((z (\x. x)) x))))))
```

Sample Output

```
I
K
B(CI)(B(BK)(C(CII)))
```

Notes

When outputting the results, consider function application to be left-associative and do not output superfluous parentheses.

There are infinitely many combinatory logic expressions that are extensionally equivalent to a given lambda expression. Moreover, the problem of determining extensional equivalence is undecidable. To pass the test cases, use the algorithm as outlined on the linked page, using all five combinators, and performing eta-conversion whenever possible.