UTICPC Take 7 and QIPC 2003: Structured Programming

Albert Y. C. Lai, University of Toronto
Thomas S. Y. Tang, Queen's University

January 25, 2003

Structured Programming

On April 27, 2002, Ole-Johan Dahl formally received his share (along with Kristen Nygaard) of the 2002 ACM Turing Award. Thirty years after the publication of the book *Structured Programming*, we can finally say that all three of its authors are Turing Award laureates: Edsger W. Dijkstra in 1972, C. A. R. Hoare in 1980, and Ole-Johan Dahl in 2002.

It is no exaggeration to say that *Structured Programming* was precisely what they were awarded for (even though the book came out shortly after Dijkstra got his award): Dijkstra, for being "one of the principal exponents of the science and art of programming languages in general"; Hoare, for "his fundamental contributions to the definition and design of programming languages"; and Dahl, for "ideas fundamental to the emergence of object oriented programming". In Chapter I, Dijkstra argues the case for program clarity through the use of good flow-control constructs and step-wise refinement. In Chapter II, Hoare promotes advanced data types such as records, unions, finite sets, and lists. In Chapter III, Hoare and Dahl together outline a new, hierarchical way of structuring programs—which we now call object-oriented programming, where "hierarchical" refers to class hierarchies.

Today all these are considered common sense by even first-year students, but you have to understand that in 1972, when the like of FORTRAN and line-numbered BASIC ruled the minds of programmers then, it took laborious exposition and eloquent argumentation to put forward that there were better ways of structuring programs than GOTO, and richer ways of structuring data than mere arrays. As always, it takes several giants to discover and point out something obvious!

Faced with the immense difficulty of programming, Dijkstra was not completely pessimistic, but instead found novel value in computers:

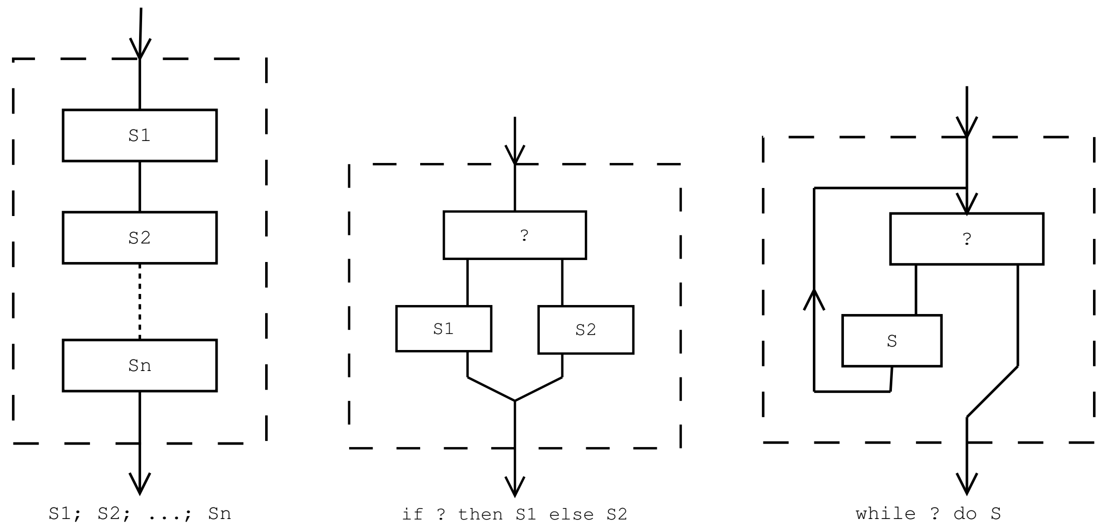
> I would venture the opinion that as long as we regard them primarily as tools, we might grossly underestimate their significance. Their influence as tools might turn out to be but a ripple on the surface of our culture, whereas I expect them to have a much more profound influence in their capacity of intellectual challenge!

And that explains why we have programming contests.

Enjoy.

A Understanding Programs

If we want to understand a program, we can decompose it into a set of components, in which each component performs a certain task. Dijkstra identified three types of decomposition: concatenation, selection, and repetition (see Figure A).



(a) Concatenation. (b) Selection. (c) Repetition.

Figure 1: Three types of decomposition. Each component is enclosed with a dotted box.

Dijkstra writes, “These flowcharts also share the property of a single entry at the top and a single exit at the bottom. They enable us to express that the action represented by the dotted block is on closer inspection a time-succession of ‘a sufficient number’ of subactions of a certain type.”

If a program is well-structured, such decomposition should be easy to do. Assuming you are analyzing a big flowchart, and you have identified a number of components. You want to enclose each component with a dotted box padded evenly with space. You can ignore the arrows between flowchart elements by assuming that the dotted box will be large enough to enclose all the arrows. Note that the boxes for the flowcharts above are not padded evenly with space, they are for illustration only.

Input

The input starts with a line containing two numbers, n and g, which are the number of components you want to enclose, and the padding length. Then n components follows. Each component starts with a line containing the number r, which is the number of rectangles this component has, followed by r lines. Each line has four numbers, x_{min}, y_{min}, x_{max}, y_{max}, indicating the coordinates of the lower-left and upper-right corner of the rectangle. The sides of all rectangles are parallel with either the x or y axis. All numbers are integers.

Output

For each component, output a line containing four numbers, indicating the coordinates of the dotted box in the same way that rectangles are represented in the input.

Sample Input

```
2 4
3
0 0 5 5
0 10 5 15
0 20 5 25
3
1 1 4 3
5 5 9 7
-2 10 4 12
```

Sample Output

```
-4 -4 9 29
-6 -3 13 16
```

B Axioms for an Abstract Data Structure

An abstract data structure is called "abstract" because there is a line drawn, for the sake of encapsulation, between the usage and the implementation; we call the line "interface" and "contract". The contract documents the operations provided, the parameter types and result types of the operations, and above all, the external behaviour of the operations. If you use the data structure, you do not need to know how it is implemented; you just assume it conforms to the contract. If you implement the data structure, you do not need to know how it is used; you just make sure it conforms to the contract. (If you work in solitude and do both sides, well, you go multiple personalities.) In this way encapsulation is achieved.

The part about documenting external behaviour trips up many a programmer. This is what makes it abstract, and at times unfamiliar. For example, in the contract for queues, you do not say, "enqueuing x means creating a node to store x and putting it at the end of the linked list." The implementer may want to use a ring buffer instead; and even if it were a linked list, it would not be externally accessible anyway. So what do you say? Well, a sensible data structure must have some accessors, so you can say what is returned by an accessor in certain contexts, and *that* is externally observable. For example, you can say,

- Right after `enqueue` x to an empty queue, `head` returns x.
- If a queue is empty, `enqueue` followed by `dequeue` will leave it empty again;
- otherwise, `enqueue` followed by `dequeue` is the same as `dequeue` followed by `enqueue`.

The latter two are relevant to external behaviour in this way: if you start with an empty queue, then `enqueue` 1, then `enqueue` 2, then `dequeue`, then finally ask for `head`, the contract should logically imply that 2 will be returned. Here is how: use the third rule to swap `enqueue` 2 and `dequeue`, then use the second rule to cancel out `enqueue` 1 and `dequeue`, and now the first rule says that 2 will be returned.

(Optional.) We can go one step further and replace natural language by mathematical notation. The above rules may read:

$$\begin{aligned} q.\mathit{isempty}() &\Rightarrow q.\mathit{enqueue}(x).\mathit{head}() = x \\ q.\mathit{isempty}() &\Rightarrow q.\mathit{enqueue}(x).\mathit{dequeue}() = q \\ \neg q.\mathit{isempty}() &\Rightarrow q.\mathit{enqueue}(x).\mathit{dequeue}() = q.\mathit{dequeue}().\mathit{enqueue}(x) \end{aligned}$$

And the above logical deduction goes:

$$\begin{aligned} & e.\mathit{enqueue}(1).\mathit{enqueue}(2).\mathit{dequeue}().\mathit{head}() \\ = \; & e.\mathit{enqueue}(1).\mathit{dequeue}().\mathit{enqueue}(2).\mathit{head}() \\ = \; & e.\mathit{enqueue}(2).\mathit{head}() \\ = \; & 2 \end{aligned}$$

This demonstrates that contracts can be written and used algebraically.

The above rules do not specify erroneous situations such as when the queue is empty and you read the head. Depending on the author of the contract, this may be handled either robustly or in the garbage-in garbage-out manner. The robust approach adds another rule that says an exception will be thrown. The garbage-in garbage-out way adds no rule, giving the implementer complete freedom to do whatever he/she sees fit—garbage will be returned, programs will crash, nations will fall, hit songs will be written, whole classes of students will be hypnotized, ... You are forewarned![1]

This problem specifies the contract of a mysterious data structure T. You will implement it, and we will test it. Here is the contract:

[1] UofT CS insider joke. Someone posted to the local newsgroup whining about difficult assignments, inadequate teaching, etc., the usual whine. But it turned bizarre when he began to claim he was a prophet, he had caused nations to fall, ...

- Operation `P` takes an integer parameter.
- Operation `Q` takes no parameter.
- Accessor `G` returns an integer.

  (Being an accessor, it does not affect the data structure as far as the contract is concerned. So for example "`P 42` then `G` then `Q`" is the same as "`P 42` then `Q`" apart from the act of returning an integer in the middle.)
- `P` followed by `Q` is the same as doing nothing (apart from wasting time and draining my laptop battery).
- Right after `P` n, `G` returns n.

  (Optional.) Again the last two rules can be written algebraically:

$$\begin{aligned} t.P(n).Q() &= t \\ t.P(n).G() &= n \end{aligned}$$

Input

The first line of the input is the number $N \leq 300$ of operations and accessors you will read. The next N lines are the operations and accessors, one per line.

Output

For each `G` in the input, output the return value. One per line.

Sample Input

```
10
P 42
P 24
G
Q
P 0
G
Q
G
G
Q
```

Sample Output

```
24
0
42
42
```

C Grouping

You have just started a new job as a project manager. The previous project manager was very strict, making all his (or her, you do not know) programmers follow a strict coding scheme with lots of inline documentation. In particular, for each function, it was required to list at the beginning the variables, types, and other functions it will use.

Now all your programmers have submitted their codes, and you want to merge all of them. However, you do not know how to group them together, as one programmer's code may depend on the another programmer's code. In other words, if code *alpha* needs code *beta*, you must put *beta* ahead of *alpha*. Now the coding scheme comes in handy: you can use that information to determine an ordering to merge all the codes.

Input

The input consists of pieces of code. Each piece takes one line of the input, started by a word indicating the name of this piece. After the name there will be an integer p, followed by p words, indicating the number of code pieces, and the names of the code pieces it needs. (Therefore a code piece that does not depend on anything will have only its name followed by the number "0".) All number/words are separated by one space. The input ends when the first word is "END". No name will have more than 10 characters, and no code piece will have the name "END".

Output

A correct output will be the names of the code pieces, one for each line, in the order of their placement such that all dependencies can be satisfied. There will be no more than 10000 code pieces. If there is more than one solution, output any solution. You may assume that the input will have at least one possible solution.

Sample Input

```
tan 2 sin cos
cos 1 factorial
factorial 0
sin 1 factorial
log 0
END
```

Sample Output

```
log
factorial
sin
cos
tan
```

D Type Check

A major advantage of high-level programming languages is the prevention of a large class of mistakes: type mismatches. For example, it is impossible to erroneously perform boolean operations on strings, as the compiler would catch it. It is also impossible to absent-mindedly use integer arithmetic instructions on floating-point numbers, as the compiler knows enough to choose the right instructions when it produces machine code. In this problem, you will write a program to catch the former kind of errors in a simple setting. You will be given simple expressions, and you will determine if they are correct in terms of types.

The syntax of expressions is a prefix kind of notation—an operator and then its operands. For example, instead of “`1 + 2`”, we write “`+ 1 2`”; and instead of “`0 * (1 + 2)`”, we write “`* 0 + 1 2`”. Here is how you read the last one:

$$\underbrace{*}_{\text{operator}} \qquad \underbrace{\texttt{0}}_{\substack{\text{first}\\\text{operand}}} \qquad \underbrace{\texttt{+ 1 2}}_{\substack{\text{second}\\\text{operand}}}$$

As you can see, there is no complication resulting from precedence rules, parentheses, etc. Although it is a bit harder for the human, it is a bit easier for the programmer, and it certainly poses no problem to Master Yoda, so endure it you will!

Here is the syntax of expressions:

- Constants: `true`, `false`, non-negative integers such as `42`, double-quoted things such as `""` and `"hi"` containing 0 or more letters and digits.
- Unary expressions: *op e*, where *op* is one of `not`, `len`, `null`; *e* is an expression.
- Binary expressions: *op e f*, where *op* is `+`, `*`, `&&`, `||`, `=`, `<=`, `++`; *e* and *f* are expressions.
- Ternary expression: *op e f g*, where *op* is one of `cond`, `sub`; *e*, *f*, and *g* are expressions.

Now you may wonder: doesn’t this syntax allow the nonsensical `+ 0 true`, say? Yes, the syntax specifies the kind of expressions you will read, and some of them will be nonsensical; it is *your* job to determine which of them are nonsense. So here are the type rules:

- There are three types: booleans, numbers, strings.
- `true` and `false` are booleans.
- Non-negative integers are numbers.
- Double-quoted things are strings.
- `not` *e* is a boolean, but *e* must be a boolean.
- `len` *e* is a number, but *e* must be a string.
- `null` *e* is a boolean, but *e* must be a string.
- `+` *e f* is a number, but *e* and *f* must be numbers.
- `*` *e f* is a number, but *e* and *f* must be numbers.
- `&&` *e f* is a boolean, but *e* and *f* must be booleans.
- `||` *e f* is a boolean, but *e* and *f* must be booleans.
- `=` *e f* is a boolean, and *e* and *f* can have any type, but they must have the same type.

- `<=` e f is a boolean, but e and f must be numbers.
- `++` e f is a string, but e and f must be strings.
- `cond` e f g has the same type as f and g, but e must be a boolean, and f and g must have the same type.
- `sub` e f g is a string, but e must be a string and both f and g must be numbers.

Input

The first line of the input is the number $N \leq 100$ of expressions you will read. The next N lines are the N expressions, one per line. Each expression takes at most 100 characters.

Output

For each expression, output a line `yes` if it satisfies the type rules; otherwise output a line `no`.

Sample Input

```
6
+ 0 * 1 2
+ 0 true
= + 0 1 * 1 2
= ++ "hello" "hi" sub "hellohi" 0 5
= ++ "hello" "hi" + 0 1
sub "hellohi" len "heya" cond && <= 0 1 not null "hi" + 0 1 * 0 1
```

Sample Output

```
yes
no
yes
yes
no
yes
```

E Correctness II

In the practice problem, you contemplated the chance of getting a program to work in a simple model, where correctness of components are mutually independent, and so you could just multiply up all the probabilities. The reality is usually more complicated than that. If component Y uses components X_1 and X_2 directly, perhaps Y does not work if X_1 or X_2 breaks, but Y has some probability to work if both X_1 and X_2 work. It is fair to assume that we are given this probability, denoted $P(Y|X_1 \wedge X_2)$. Then the probability that Y works is given by:

$$P(Y) = P(Y|X_1 \wedge X_2) \times P(X_1 \wedge X_2)$$

Now $P(X_1 \wedge X_2)$ is not always $P(X_1) \times P(X_2)$. Assume that X_1 and X_2 do not use each other directly or indirectly. Let's say X_1 uses W_1 and W_2 directly, and X_2 uses W_1' and W_2' directly. When W_1, W_2, W_1', W_2' all work, it is fair to say that X_1 and X_2 are independent in this context. Using probability theory, this allows us to conclude:

$$P(X_1 \wedge X_2) = s \times s' \times P(W_1 \wedge W_2 \wedge W_1' \wedge W_2')$$

where the first and second terms are given:

$$\begin{aligned} s &= P(X|W_1 \wedge W_2) \\ s' &= P(X'|W_1' \wedge W_2') \end{aligned}$$

The third term can be computed by the same idea.

The foregoing can be generalized to $X_1 \wedge \cdots \wedge X_n$, $W_1 \wedge \cdots \wedge W_m$, etc.

You are to compute the probability that the main program works. The system is layered: the main program alone is layer 0; components used by layer n constitute layer $n+1$. Each layer has at most 10 components. Component in the same layer do not use each other directly or indirectly, so that a certain assumption above is satisfied.

Input

The input consists of up to 500 pieces of code. Each piece takes one line of the input, started by a word indicating the name Y of this piece. After the name there will be an integer n, followed by n words, indicating the number of code pieces, and the names X_i of the code pieces it uses directly, followed by a floating-point number $r = P(Y|X_1 \wedge \cdots \wedge X_n)$. (Therefore a code piece that does not depend on anything will have only its name followed by the number "0" and then the number r.) All number/words are separated by one space. The input ends when the first word is "END". No code piece will have the name "END". There is a main program named MAIN; it is not used by any other pieces.

Output

Output the probability that MAIN works.

Sample Input

```
tan 2 sin cos 0.9
cos 1 factorial 0.8
factorial 0 0.7
sin 1 factorial 0.85
log 0 0.95
MAIN 2 tan log 0.75
END
```

Sample Output

```
0.305235
```