Stack Controller

The stack controller is a device which when given an input either accepts the input or rejects it. Internally the stack controller maintains a

- set of rules,
- stack
- lookup table generated from the set of rules.

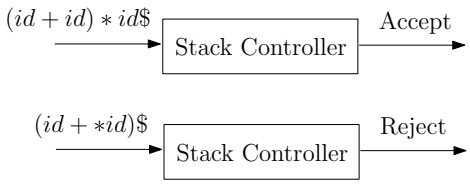
There are only two kinds of symbols:

- non-final Symbols: single capital Letters e.g. F, E, B etc
- final Symbols: everything else. e.g. id, (, +, *, $,) etc

Rules are represented as: $A\,!\,x$, where A is called *Head* (non final symbol) and x is called the *Tail* (sequence of final and non-final symbols) of the rule.

An input to the stack controller is a sequence of final symbols as shown in the following figures. As an example, the following set of 6 rules are used.

(1) $X\,!\,X + Y$
(2) $X\,!\,Y$
(3) $Y\,!\,Y * Z$
(4) $Y\,!\,Z$
(5) $Z\,!\,(X)$
(6) $Z\,!\,id$



Here the non-final symbols are X, Y, and Z while the final symbols are id, $+$, $*$, (and). An additional symbol \$ marks the end of the input.

Some examples of input sequences accepted by the stack controller are $id\$$, $(id + id)\$$, $(id + id) * id\$$ etc and some input sequences rejected are $(id + *id)\$$, $(id * id)id\$$ etc. Table 1 shows the lookup table corresponding to these example set of rules:

While reading the input, the stack controller moves from one state to another. The initial state is 0. The top of the stack contains the current state. With the exception of the first symbol in the input, a symbol in the input is read only after the previous symbol is pushed onto the stack. The stack controller accepts the input sequence if at end of input (at \$), it reaches the state with entry as '*acc*'.

Interpretation of the entries in the table is as follows.

1. pi for p : push, i : state. Action consists of :
   - push the current input symbol on the stack and then
   - push state i onto the stack.
2. rj for r : replace, j : rule number. Remember that rules are of the form $H!T$ where H is head and T is tail. Action consists of :
   - All the symbols from the stack are popped until the sequence forms the Tail of rule j. Ignore the states that are popped.

STATE	push / replace						goto		
	id	+	*	(	)	$	X	Y	Z
0	p5			p4			1	2	3
1		p6				acc			
2		r2	p7		r2	r2			
3		r4	r4		r4	r4			
4	p5			p4			8	2	3
5		r6	r6		r6	r6			
6	p5			p4				9	3
7	p5			p4					10
8		p6			p11				
9		r1	p7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Table 1. Lookup Table

- Once all the symbols that make up the tail of Rule j are popped (in sequence) – the symbol on the top of the stack will be a number representing a state, say S.
- Look up the entry for S(row) and H(column) in the 'Goto' section of the table. The entry represents a state Q.
- Push H and then Q on the stack.

3. *acc* means accept the input
4. blank means reject the input. Blank is represented by $-$.

Below is the execution on input **id * id + id**$ using stack.

Write a complete program in C/C++ for above described controller. Note that the set of rules, the lookup table and the input to the stack controller are to be read from a file.

- Input: The program expects the input in the following format:
  An example of an input for the rules and the lookup table given above is as follows:

	STACK	INPUT	ACTION
1	0	id * id + id $	push
2	0 id 5	* id + id $	replace by $Z\,!\,id$
3	0 Z 3	* id + id $	replace by $Y\,!\,Z$
4	0 Y 2	* id + id $	push
5	0 Y 2 * 7	id + id $	push
6	0 Y 2 * 7 id 5	+ id $	replace by $Z\,!\,id$
7	0 Y 2 * 7 Z 10	+ id $	replace by $Y\,!\,Y * Z$
8	0 Y 2	+ id $	replace by $X\,!\,Y$
9	0 X 1	+ id $	push
10	0 X 1 + 6	id $	push
11	0 X 1 + 6 id 5	$	replace by $Z\,!\,id$
12	0 X 1 + 6 Z 3	$	replace by $Y\,!\,Z$
13	0 X 1 + 6 Y 9	$	replace by $X\,!\,X + Y$
14	0 X 1	$	accept

Input file content	Explanation
`6`	: Number of rules
`X! X + Y`	: Rule 1 (symbols separated by a space)
`X!Y`	: Rule 2
`Y!Y * Z`	: Rule 3
`Y!Z`	: Rule 4
`Z!( X )`	: Rule 5
`Z!id`	: Rule 6
`6 3`	: Number of final and non-final symbols
`id + * ( ) $ X Y Z`	: The final symbols followed by non-final symbols.
`12 9`	: Number of rows and columns of the lookup table
`p5 - - p4 - - 1 2 3`	: a[0][0] a[0][1] ... a[0][8]
`- p6 - - - acc - - -`	: a[1][0] a[1][1] ... a[1][8]
⋮	⋮
`- r5 r5 - r5 r5 - - -`	: a[11][0] a[11][1] ... a[11][8]
`3`	: Number of input sequences
`id + id * id$`	: Input sequence 1 (symbols separated by a space)
`( id + id ) * id$`	: Input sequence 2
`id + * id$`	: Input sequence 3

– Design suitable data structures for storing the lookup table and the rules.

– Write a function to accept or reject an input.

– Write a function to display the rules in *reverse order* of their applications.

– For the example input provided above, the format of the output is the following:

Output File Content	Explanation
`accepted`	: accepted/rejected
`X!X + Y`	: rules applied in reverse order if accepted
`Y!Y * Z`	
`Z!id`	
`Y!Z`	
`Z!id`	
`X!Y`	
`Y!Z`	
`Z!id`	
`accepted`	: accepted/rejected
`X!Y`	: rules applied in reverse order if accepted
`Y!Y * Z`	
`Z!id`	
`Y!Z`	
`Z!(X)`	
`X!X + Y`	
`Y!Z`	
`Z!id`	
`X!Y`	
`Y!Z`	
`Z!id`	
`rejected`	: accepted/rejected

Marks Distribution (Total: 25)

Designing correct data structures	:	**5**
Function for accept/reject and correct output for test cases	:	**12**
Display of rules in reverse order correctly	:	**5**
Documentation	:	**2**
Overall correctness	:	**1**

Please Note that

1. No Windows programs are allowed and your program has to work correctly on a Linux machine.
2. Marks will be deducted for use of global variables and memory leaks.
3. Marks will be deducted if your program does not conform to the specified input and output format.