Clac

1 - Introducing the Claculator

Clac is a new stack-based programming language developed by a Pittsburgh-area startup called Reverse Polish Systems (RPS). Any similarities of Clac with Forth or PostScript are purely coincidental. In the first part of this assignment, we will be implementing the core features of the Claculator, and later we will be adding a few more interesting features.

Clac works like an interactive calculator. When it runs, it maintains an operand stack. Entering numbers will simply push them onto the operand stack. When an operation such as addition + or multiplication * is encountered, it will be applied to the top elements of the stack (consuming them in the process) and the result is pushed back onto the stack. When a newline is read, the number on top of the stack will be printed.

This is an example where we start Clac and type 3 4 +, followed by a newline:

```
clac>> 3 4 +
stack] 7
```

Here is another example:

```
clac>> 1 2 + 3 *
Stack] 9
```

Before moving on please make sure you understand why the following will output 11:

```
clac>> 2 4 3 * - 1 +
Stack] 11
```

In addition to the arithmetic operations, there are a few special operations you will have to implement. The table on the next page is the complete set of operations that you will be implementing. To specify the operations, we use the notation

S||Q → S'||Q'

to mean that the stack S and the queue Q transitions to become stack S' and the queue Q'. We use a queue to hold the numbers and operations Clac evaluates. Stacks are written with the top element at the right end! For example, the action of multiplication is stated as

S, x, y || *, Q → S, x*y || Q

which means: “When you dequeue the token * from the queue, pop the top element (y) and the next element (x) from the stack, multiply y by x, and push the result x * y back onto the stack.” The fact that we write S in the rule above means that there can be many other integers on the stack that will not be affected by the operation.

Every operation in Clac is determined by the token that has just been dequeued from the queue. The < and if tokens cause different things to happen depending on the specific values on the stack.

S, x, y || <, Q → S, 1 || Q if x < y
S, x, y || <, Q → S, 0 || Q if x ≥ y

In Clac, we use the integer 0 to mean false and we treat non-zero values like 1 as true. The if token runs some code (the next three tokens) only if the integer on the top of the stack is true (that is, nonzero). If the if token is reached while 0 is at the top of the stack, the following three tokens are skipped.

As you implement the Clac operations on the next page, if the instructions indicate that Clac should raise an error, you should throw the appropriate java exception.

2 - Implementing Clac

Before			After		Condition/Effect
`Stack`	`Queue`	→	`Stack`	`Queue`	
S	*n, Q*	→	*S, n*	*Q*	
S, n	*print, Q*	→	*S*	*Q*	prints number to screen
S	*quit, Q*	→		*Q*	*quits the clac interpreter*
S, x, y	*+ , Q*	→	*S, x+y*	*Q*	
S, x, y	*- , Q*	→	*S, x-y*	*Q*	
S, x, y	** , Q*	→	*S, x*y*	*Q*	
S, x, y	*/ , Q*	→	*S, x/y*	*Q*	
S, x, y	*% , Q*	→	*S, x%y*	*Q*	
S, x, y	*** , Q*	→	*S, x^y*	*Q*	
S, x, y	*< , Q*	→	*S, 1*	*Q*	if x < y
S, x, y	*< , Q*	→	*S, 0*	*Q*	if x >= y
S, x, y	*= , Q*	→	*S, 1*	*Q*	if x = y
S, x, y	*= , Q*	→	*S, 0*	*Q*	*if x != y*
S, x	*drop , Q*	→	*S*	*Q*	
S, x, y	*swap , Q*	→	*S, y, x*	*Q*	
S, x, y, z	*rot , Q*	→	*S, y, z, x*	*Q*	
S, x	*if , Q*	→	*S*	*Q*	x != 0
S, x	*if , tok1, tok2, tok3, Q*	→	*S*	*Q*	x = 0
S, xn, . . . , x1, n	*pick , Q*	→	*S, xn, . . . , x1, xn*	*Q*	See note #1
S, n	*skip , tok1, . . . , tokn, Q*	→	*S*	*Q*	See note #1

Note #1: The pick operation should throw an IndexOutOfBoundsException if n is less than or equal to 0. The skip operation should throw an IndexOutOfBoundsException if n is less than 0.

Task 1
Add implementations of +, -, *, **, /, %, <, and = according to the specification.

Task 2
Add definitions of drop, swap, rot, if, skip, and pick according to the specification.

3 - Function Definitions

Finally, we add definitions to Clac. A definition has the form:

`clac>> : name token`$_1$ `...` `token`$_n$ `;`

When we encounter the token : (colon) in the input queue, we interpret the following token as a name. Then we create a new (separate) queue, intended to hold the definition of name. Then we continue to scan the input queue, copying each token to the new queue until we encounter a token ; (semicolon) which signals the end of the definition. Then we add name, with the new queue as its definition, to the dictionary.

If the queue ends after the colon (:), or if there is no semicolon (;) in the remainder of the queue after name, an error should be signaled. In a definition, name can be any token, but if it is a built-in operator or a number, then the definition can never be invoked since it is always superseded by the predefined meaning.

Let's consider a simple example.

```
clac>> : dup 1 pick;
stack]
clac>> 2 dup
stack] 2 2
clac>> : square dup * ;
stack] 2 2
clac>> square
stack] 2 4
```

This defines dup and square. Whenever we see square in the input subsequently, it has the effect of replacing n on the top of the stack with n^2. For example,

How do we process a defined name when we encounter it in the queue of tokens? This is not entirely straightforward, as the following example illustrates:

```
clac>> 3 square 4 square +
stack] 25
```

When we see the first occurrence of square we cannot simply replace the rest of the queue with the definition of square, since after squaring 3 we have to continue to process the rest of the queue, namely 4 square +.

In order to implement this, we maintain a stack of queues of tokens. This is called the return stack or sometimes the call stack. When we encounter a defined token, we push the remainder of the queue onto the return stack, and begin using the definition as our queue of tokens. When we finish executing a definition we pop the prior queue of tokens from the return stack and continue with processing it. When there are no longer any queues on the return stack we return from the eval function.

You may remove the same queue from a dictionary more than one time, like the queue one associated with square in the example above. Therefore, it's important that each time you look up a queue in the dictionary, you make a copy of it before you dequeue from that queue. There is a method in the clac.java file that returns a copy of the queue passed as a parameter.

Here is an example of complex clac program that converts a decimal number to a binary number:

```
clac>> : convert 1 pick 2 / dec2bin 10 * swap 2 % + ;
stack]
clac>> : dec2bin 1 pick if convert ;
stack]
clac>> 37 dec2bin
Stack] 100101
```