Graded Homework Assignment n. 3

Due date: *Friday, December 23, 2022 at 22:00*

In a source file called `gorp.c` or `gorp.cc` write a C or C++ program that executes a *gorp* program.

The syntax of the *gorp* programming language is very simple: a program is a sequence of strings of non-space characters separated by space characters. The following strings are reserved keywords: `{`, `}`, `.`, `<`, `>`, `<=`, `>=`, `==`, `!=`, `+`, `-`, `*`, `/`, `load`, `store`, `input`, `output`, `if`, `ifelse`, `while`, `dup`. The execution of the program is based on a stack, very much like those calculators that use the "reverse polish notation." For example, here is a simple program that prints the sum of two numbers:

```
7 13 + output
```

The string `7` is not a keyword and therefore is pushed onto the stack. Same for `13`. The string `+` is instead a keyword that invokes a sum operation. The sum pops two strings from the stack, interprets them as integers a and b, and then pushes the string representing the sum $a + b$ onto the stack. The following string `output` is also a keyword that invokes the *output* operation, which pops a string s from the stack and prints that string onto the standard output.

In the same way, you can compute more complex arithmetic calculations. For example, the following program converts Fahrenheit in Celsius, reading the Fahrenheit value from the standard input.

```
input 32 - 5 * 9 / output
```

Notice however, that all arithmetic operations must interpret the strings arguments popped from the stack as integers, and their results are also integer. In particular, the `/` operation performs an integer division. So, the conversion is not exact. For example, with an input 60, the program must output 15, whereas the exact conversion would be $15.55\ldots$.

In addition to strings, the stack can also contain (or simply refer to) blocks of code, meaning subsequences of strings. This is how you write conditional statements and loops in *gorp*. Here is an example:

```
input
100 > { Good! } { Bad! } ifelse
output
```

This program reads a string representing a number n from the input, and then outputs the string "Good!" if n is greater than 100, or otherwise prints the string "Bad!" Here is a step-by-step explanation of the execution of this program: the first string in the program is the keyword `input` that causes the program to push onto the stack a string read from the standard input. Then the string `100` is also pushed onto the stack, and then the string `>` invokes the *greater-than* operation that pops two strings from the stack, first $b = 100$ and then a corresponding to the input, interprets them as integers, and pushes onto the stack the result of comparison $a > b$.

The program then proceeds to the `{` string, which is a keyword that represents the beginning of a block. In this case, the program pushes the block of string instructions enclosed within `{` and `}` onto the stack. This block may contain other blocks of code. So, the block does not stop at the first `}` instruction, but rather after a set of *balanced* `{` and `}` instructions. In the example above, this is done for the block that contains the string `Good!` and also for the block that contains the

string `Bad!`. Then the program proceeds to the string `ifelse`, which is a keyword that executes an if-then-else operation. In particular, the program pops three things from the stack: first the second block (*else-block*), then the first block (*if-block*), and then a Boolean condition. The program then executes the *if-block* if the condition is *true* or the *else-block* if the condition is *false*. A condition is simply a string, such that the empty string is interpreted as *false*, while any other string is interpreted as *true.*

Below is another, slightly more complex example that also illustrates loops and nested blocks.

```
0
do {
    input
    dup { + continue } if
} while
output
```

This reads a sequence of strings representing numbers from the standard input, adds them all up (as integers) and then outputs the result. Let's see exactly how the program does that. The first string `0` is pushed onto the stack. The string `do` is also pushed onto the stack, since it is not a keyword. Then the block of code `{ input dup { + continue } if }` is pushed onto the stack, and then the program proceeds to the `while` keyword that executes a while-loop. The loop works by popping the code block and then the condition from the stack. The program executes the block as the body of the loop if the condition is *true*. In the example, the condition is the string `do`, which is interpreted as *true*. Thus the program executes the body of the loop. At the end of the loop, the program again pops a condition *C* and reruns through the body of the loop if *C* is *true*.

The body of the loop contains the keyword `input` that reads and pushes onto the stack a string from the input. Then the keyword "dup" duplicates the string on top of the stack. Then the program runs `{ + continue } if`, which pops the *if-block* and then a condition C_{if}, and executes the *if-block* only if C_{if} is *true*. The *if-block* then adds the two elements that are on the stack, leaving their sum on the stack, and then also pushes a `continue` onto the stack, which is interpreted as a *true* condition for the continuation of the while loop.

Notice that condition is the string read from the input. Now, notice that, on success, the `input` operation reads and pushes a non-empty string onto the stack. Instead, when the input ends, the `input` operation pushes the empty string onto the stack, which is then evaluated as *false* by the `if` operation. Therefore, the end of input causes the `if` instruction to skip the *if-block*, and also causes the `while` operation to exit from the loop.

In addition to the stack, the *gorp* language has named variables. Here is an example.

```
0 n store
do {
    input
    dup {
        dup A n load . store
        n load 1 + n store
        continue
    } if
} while
do {
    n load
    0 !=
    dup {
        A n load 1 - dup n store . load
        output
        continue
    } if
} while
```

This program reads a sequence of strings from the standard input, stores each one of then in variables named `A0`, `A1`, etc., and then prints them out in reverse order. The new and important operations we use in this example are the `load` and `store` operations, as well as the string concatenation operation represented by the "`.`" keyword. The first line of code `0 n store` stores the string `0` in a variable named `n`. That value is then later pushed back onto the stack with `n load`.

Below is a reference of all the operations of all the keywords.

keyword	stack before	stack after	semantics
`{`		*code-block*	read and push a code block
`}`		*code-block*	close a code block
`.`	a b	ab	concatenate two strings
`<`	a b	$a < b$	less-than relation
`>`	a b	$a > b$	greater-than relation
`<=`	a b	$a \leq b$	less-than or equal relation
`>=`	a b	$a \geq b$	greater-than or equal relation
`==`	a b	$a = b$	equals relation
`!=`	a b	$a \neq b$	not-equals relation
`+`	a b	$a + b$	add two integers
`-`	a b	$a - b$	subtract two integers
`*`	a b	$a \cdot b$	multiplies two integers
`/`	a b	a/b	integer division
`load`	*name*	value of *name*	push the value of variable *name*
`store`	*value name*		store *value* into variable *name*
`input`		x	read and push x from the standard input
`output`	x		pops and prints x onto the standard output
`if`	*cond block*		execute *block* if *cond* is *true* (not empty)
`ifelse`	*cond block1 block2*		execute *block1* if *cond*, otherwise *block2*
`while`	*cond block*		execute *block* repeatedly if *cond*
`dup`	x	x x	duplicates the string value on top of the stack

The gorp interpreter—your solution—must read the program source from a file whose name is given as command-line argument.

Hints

Conceptually, the stack contains strings, code blocks, and Boolean values. However, as explained in the examples above, you don need to represent Boolean values, and instead you can simply interpret an empty string as *false*, and any other string as *true*. You also don't need to store code onto the stack. Instead, you can represent a code block as a sub-sequence of instructions (strings) of the program. In fact, it makes sense to read and store the entire program before starting its execution.

Submission Instructions

Submit one source file as required through the iCorsi system. Add comments to your code to explain sections of the code that might not be clear. You must also add comments at the beginning of the source file to properly acknowledge any and all external sources of information you may have used, including code, suggestions, and comments from other students. If your implementation has limitations and errors you are aware of (and were unable to fix), then list those as well in the initial comments.

You may use an integrated development environment (IDE) of your choice. However, *do not submit any IDE-specific file,* such as project description files. Also, *make absolutely sure that the file you submit can be compiled and tested with a simple invocation of the standard C/C++ compiler.*