First Task

Programming Languages
LVA 185.208
2020 S
TU Wien

What to do

Develop a programmable calculator according to the following specification, and write program text to test the calculator as described below. To implement the calculator you can use any language you like, but program text for testing (including the startup procedure, see "Testing") must be written in the language of the calculator.

How to use the Calculator

The calculator uses post-fix notation: first the arguments, then the operator. Arguments are floating-point numbers or lists, see below. For example, `5.1 12.3+` applies the operator `+` to the arguments `5.1` and `12.3`, giving `17.4` as result. White space between `5.1` and `12.3` separates the numbers from each other.

Post-fix expressions are evaluated using a stack. Arguments are simply pushed onto the stack, operators pop arguments from the stack and push results onto the stack. For example, the expression `15.3 2 3.1 4+*-` is evaluated to `1.1`: First, the four numbers (all floating-point numbers) are pushed onto the stack, then the evaluation of `+` pops `3.1` and `4` from the stack and pushes `7.1` onto the stack, the evaluation of `*` pops `2` and `7.1` from the stack and pushes `14.2` onto the stack, and finally `-` pops `15.3` and `14.2` from the stack and pushes `1.1` onto the stack.

A sequence of characters enclosed in parentheses is a list. A list is evaluated when applying the operator `@` to it. For example, `(2*)` is a list. When evaluating `4 3(2*)@+`, first `4`, `3` and `(2*)` are pushed onto the stack, then `@` pops the list from the stack and causes its contents to be evaluated by pushing `2` onto the stack and applying `*` to `3` and `2`, and finally `+` adds `4` and `6`. Hence `(2*)@` doubles the number on top of the stack.

Architecture of the Calculator

The calculator consists of the following parts:

Command stream: A stream of characters regarded as commands to be executed in sequential order. When switching on the calculator the command stream is initialized with the contents of the list in register a (see below).

Operation mode: An integer used in controlling the interpretation of commands. Its value is 0 while executing commands, -1 while reading digits belonging to numbers to the left of ".",

smaller than -1 while reading digits belonging to the right of "." and larger than 0 while reading lists. The operation mode is 0 when switching on the calculator.

Data stack: This is the stack holding floating-point numbers and lists when evaluating expressions in post-fix notation. The stack is empty when switching on the calculator.

Register set: A set of 26 registers named by lowercase letters a to z, each holding a single floating-point number or list. Registers contain predefined values when switching on the calculator, where the predefined value of register a must be a list containing the initial content of the command stream. There are commands for reading and writing registers.

Input stream: A stream of characters typed in using the keyboard. There is a command for reading a whole line of input (terminated by "enter") and converting it to a list or number, depending on the content. The input stream is empty when switching on the calculator.

Output stream: A stream of characters displayed on the screen. There is a command for converting numbers and lists to character sequences and writing them to the output stream.

For simplification there can be reasonable limits on the sizes of numbers, lists, the command stream and the data stack.

Operations

The first character in the command stream, we call it *input character*, gets executed. On execution this character is removed from the command stream and the next character becomes executable. The kind of character and the operation mode together determine the operation to be executed.

Whole Number Construction Mode

Operation mode -1 is used for constructing whole numbers from digits. In this mode, the top entry on the data stack must be a whole number, and the input character causes the following behavior:

Digit `'0'` to `'9'` multiplies the top entry on the data stack by 10 and then adds the value of the input character (0 to 9).

Dot `'.'` causes the operation mode to become -2.

Each other character causes the operation mode to become 0, then this input character is executed using execution mode 0.

Decimal Place Construction Mode

Operation modes m less than -1 are used for constructing decimal places of numbers from digits. In this mode, the top entry on the data stack must be a number, and the input character causes the following behavior:

Digit `'0' to '9'` adds the floating-point value of the input character (0.0 to 9.0) multiplied by 10^{m+1} to the top entry on the data stack, and the operation mode becomes $m - 1$.

Dot `'.'` pushes a number of value 0 onto the data stack, and the operation mode becomes -2 (this is, initiates the construction of a new number).

Each other character causes the operation mode to become 0, and this input character is executed using execution mode 0.

List Construction Mode

Operation modes m larger than zero are used for constructing lists from characters, where m is the number of open parentheses. In these operation modes, the top entry on the data stack must be a list, and the input character causes the following behavior:

`'('` adds `'('` to the list on top of the data stack, and the operation mode becomes $m + 1$.

`')'` adds `')'` to the list on top of the data stack if $m > 1$, and causes the operation mode to become $m - 1$ in every case (this is, nothing is added if the operation mode becomes 0).

Each other character adds the input character to the list on top of the data stack.

Execution Mode

In operation mode 0 the input character is regarded to be an executable command causing the following behavior:

Digit `'0' to '9'` pushes the value of the input character (0 to 9) as a whole floating-point number onto the data stack, and the operation mode becomes -1.

Dot `'.'` pushes a number of value 0 onto the data stack, and the operation mode becomes -2.

`'('` pushes an empty list onto the data stack, and the operation mode becomes 1.

Lowercase letter `'a' to 'z'` pushes the contents of the corresponding data register a to z onto the data stack.

Uppercase letter `'A'` to `'Z'` pops a value from the data stack and stores this value in the corresponding lowercase data register a to z, thereby destroying the old register value.

Comparison operation `'='`, `'<'` or `'>'` pops two entries from the data stack, compares them and pushes a number 0 or 1 onto the data stack. To deal with inaccurate calculations with floating-point numbers, we need a predefined value *epsilon* (assume an appropriate value of epsilon) representing the allowed inaccuracy. Each number between -epsilon and epsilon is regarded as Boolean value *false*, each other number as *true*. When comparing two numbers, these numbers are regarded as equal if they differ at most by epsilon multiplied by the larger value; this usually works fine when comparing values different from zero. When comparing two lists, the comparison is on the corresponding strings in lexicographical ordering. When comparing a list with a number, the number is always smaller than the list. Hence, we can easily decide if a value is a number or list by comparing this value with the empty list `()`.

Check `'?'` pops a value from the data stack and pushes 1 onto the data stack if the poped value is the empty list or a number representing the Boolean value false (see above), otherwise pushes 0 onto the data stack. This operator shall be used to effectively compare numbers with zero (range -epsilon to epsilon). It can also be used to negate Booleans.

Arithmetic or logic operation `'+'`, `'-'`, `'*'`, `'/'`, `'&'` or `'|'` pops two entries from the data stack, applies the operation on them and pushes the result to the data stack. These operators have the usual semantics when applied to numbers, where `'&'` is the logical AND and `'|'` the logical OR, resulting in 0 or 1 based on the definitions of true and false given above. The empty list `()` is pushed to the data stack if an argument is not a number or the result would be the special value NaN (not a number). The ordering of arguments has to be considered for non-associative operations: `4 2-` and `4 2/` have `2` as result.

Negation `'~'` changes the sign of the top entry on the data stack if it is a number, otherwise it replaces the top entry with the empty list `()`.

Rounding `'%'` pushes a value onto the data stack depending on the current value on top of the stack. If the top of stack value is a number, the pushed value is a number in the range $(-0.5, 0.5)$ such that adding this number to the current top of stack value results in the closest possible whole number. Otherwise the empty list `()` is pushed.

Square root `'_'` (underline) replaces the top entry v on the data stack with the square root of v if v is a positive number. There is no effect if v is not a positive number.

Copy `'!'` replaces the top entry v on the data stack with a copy of the nth entry on the data stack (counted from the top of stack) if v is an appropriate number, where n is constructed by rounding v to the closest whole number. There is no effect if v is not a number or n is not in the appropriate range.

Delete `'$'` pops the top entry v from the data stack and (if n is an appropriate number) removes the nth entry from the data stack (counted from the top of the stack), where n is constructed by rounding v to the closest whole number. Pops only v from the data stack if v is not a number or n is not in the appropriate range.

Apply immediately `'@'` pops a list from the data stack (if the top entry is a list) and inserts the list contents at the begin of the command stream to be executed next. There is no effect if the top entry is a number.

Apply later `'\'` pops a list from the data stack (if the top entry is a list) and inserts the list contents at the end of the command stream to be executed after everything else in the command stream. There is no effect if the top entry is a number.

Stack size `'#'` pushes the current number of stack entries onto the stack.

Read input `'\''` (single quote) waits until the input stream contains a line (terminated by "enter") and converts the line (except of "enter") to an input value: If the characters in the line can be interpreted as a number, the input value is the corresponding number. Otherwise if the line is well-formed (this is, if there are parentheses, opening and closing parentheses match), the input value is a list containing the sequence of characters in the line. Otherwise (not well-formed) the input value is the empty list `()`. The input value is pushed onto the data stack.

Write output `'"'` pops a value from the data stack and writes it to the output stream. If the value is a list, the characters in the list are directly written to the output stream (without additional parentheses). If the value is a number, it is written to the output stream in an appropriate format (beginning with `-` for negative numbers and avoiding unnecessary digits).

Everything else in the command stream does nothing (except of separating two adjacent numbers from each other).

In addition to above cases, an error is reported if the data stack does not have enough entries. If an error occurs, the calculator simply stops its execution and gives an error message.

Examples

The following examples clarify the use of some commands and introduce specific programming techniques. We specify the state of the calculator by the contents of its data stack (to the left of ◁, lists in boxes, the top of the stack adjacent to ◁) and its command stream (to the right of ◁, the first entry adjacent to ◁). Arrows between state specifications show how states change by executing commands (⟶ for one command, ⟹ for several commands).

The first example shows how to deal with conditional execution: On the data stack we expect a number used as Boolean value. Depending on this value we want to execute the one or the other list. First we push a list for the false-path `(9~)` onto the stack, then one for the true-path `(8)`, and finally we execute `(4!4$1+$@)`, the code for conditional execution. The steps (only those in execution mode) show what happens if previously 0.0 was on the data stack.

	0.0				◁	`(9~)(8)(4!4$1+$@)@`
⟹	0.0	`9~`			◁	`(8)(4!4$1+$@)@`
⟹	0.0	`9~`	`8`		◁	`(4!4$1+$@)@`
⟹	0.0	`9~`	`8`	`4!4$1+$@`	◁	`@`
⟶	0.0	`9~`	`8`		◁	`4!4$1+$@`
⟶	0.0	`9~`	`8`	4.0	◁	`!4$1+$@`
⟶	0.0	`9~`	`8`	0.0	◁	`4$1+$@`
⟶	0.0	`9~`	`8`	0.0 4.0	◁	`$1+$@`
⟶	`9~`	`8`	0.0		◁	`1+$@`
⟶	`9~`	`8`	0.0 1.0		◁	`+$@`
⟶	`9~`	`8`	1.0		◁	`$@`
⟶	`9~`				◁	`@`
⟶					◁	`9~`
⟶	9.0				◁	`~`
⟶	−9.0				◁	

The example on page 8 shows recursion in the computation of the factorial of 3. We use A as a shorthand for `3!3!1-2!1=4!()(`C`)@2$*` and C as a shorthand for `4!4$1+$@` – see above. Please note that the only purpose of A and C is a simplification to improve readability. In the calculator the full text occurs instead of its shorthand.

Testing

Please write code for testing (including the contents of registers) only in the language of the calculator. Do not extend the calculator with additional operations. Use registers to provide program code supposed to exist when switching on the calculator.

Please provide program code in register a that causes the calculator to start with a welcome message and to repeatedly ask for input to be executed (showing stack contents after executions). It shall be possible to execute arbitrary programs in this way.

Write program code (in one of the registers) that computes the surface area of a triangle specified by three corner points, where each corner point is specified by coordinates in a three-dimensional space.

Also write program code that computes the sum of surface areas for an arbitrary (but given) number of such triangles, where coordinates of corners are on the stack.

Also write program code that computes the surface area of an octahedron (Oktaeder, see e.g. Wikipedia) of a given size twice and shows both results. One surface area shall be computed using the usual formular for doing so, the other by finding appropriate coordinates of the octahedron's corners and applying the register contents described above.

Appendix (Factorial of 3 – see page 6)

	3.0	◁ (*A*)3!3$3!@2$
⟹	3.0 [*A*]	◁ 3!3$3!@2$
⟶	3.0 [*A*] 3.0	◁ !3$3!@2$
⟶	3.0 [*A*] 3.0	◁ 3$3!@2$
⟶	3.0 [*A*] 3.0 3.0	◁ $3!@2$
⟶	[*A*] 3.0	◁ 3!@2$
⟶	[*A*] 3.0 3.0	◁ !@2$
⟶	[*A*] 3.0 [*A*]	◁ @2$
⟶	[*A*] 3.0	◁ *A*2$
⟶	[*A*] 3.0 3.0	◁ !3!1-2!1=4!()(*C*)@2$*2$
⟶	[*A*] 3.0 [*A*]	◁ 3!1-2!1=4!()(*C*)@2$*2$
⟶	[*A*] 3.0 [*A*] 3.0	◁ !1-2!1=4!()(*C*)@2$*2$
⟶	[*A*] 3.0 [*A*] 3.0	◁ 1-2!1=4!()(*C*)@2$*2$
⟶	[*A*] 3.0 [*A*] 3.0 1.0	◁ -2!1=4!()(*C*)@2$*2$
⟶	[*A*] 3.0 [*A*] 2.0	◁ 2!1=4!()(*C*)@2$*2$
⟶	[*A*] 3.0 [*A*] 2.0 2.0	◁ !1=4!()(*C*)@2$*2$
⟶	[*A*] 3.0 [*A*] 2.0 2.0	◁ 1=4!()(*C*)@2$*2$
⟶	[*A*] 3.0 [*A*] 2.0 2.0 1.0	◁ =4!()(*C*)@2$*2$
⟶	[*A*] 3.0 [*A*] 2.0 0.0	◁ 4!()(*C*)@2$*2$
⟶	[*A*] 3.0 [*A*] 2.0 0.0 4.0	◁ !()(*C*)@2$*2$
⟶	[*A*] 3.0 [*A*] 2.0 0.0 [*A*]	◁ ()(*C*)@2$*2$
⟹	[*A*] 3.0 [*A*] 2.0 0.0 [*A*] []	◁ (*C*)@2$*2$
⟹	[*A*] 3.0 [*A*] 2.0 [*A*]	◁ @2$*2$
⟶	[*A*] 3.0 [*A*] 2.0	◁ *A*2$*2$
⟶	[*A*] 3.0 [*A*] 2.0 3.0	◁ !3!1-2!1=4!()(*C*)@2$*2$*2$
⟶	[*A*] 3.0 [*A*] 2.0 [*A*]	◁ 3!1-2!1=4!()(*C*)@2$*2$*2$
⟶	[*A*] 3.0 [*A*] 2.0 [*A*] 3.0	◁ !1-2!1=4!()(*C*)@2$*2$*2$
⟶	[*A*] 3.0 [*A*] 2.0 [*A*] 2.0	◁ 1-2!1=4!()(*C*)@2$*2$*2$
⟶	[*A*] 3.0 [*A*] 2.0 [*A*] 2.0 1.0	◁ -2!1=4!()(*C*)@2$*2$*2$
⟶	[*A*] 3.0 [*A*] 2.0 [*A*] 1.0	◁ 2!1=4!()(*C*)@2$*2$*2$
⟶	[*A*] 3.0 [*A*] 2.0 [*A*] 1.0 2.0	◁ !1=4!()(*C*)@2$*2$*2$
⟶	[*A*] 3.0 [*A*] 2.0 [*A*] 1.0 1.0	◁ 1=4!()(*C*)@2$*2$*2$
⟶	[*A*] 3.0 [*A*] 2.0 [*A*] 1.0 1.0 1.0	◁ =4!()(*C*)@2$*2$*2$
⟶	[*A*] 3.0 [*A*] 2.0 [*A*] 1.0 1.0	◁ 4!()(*C*)@2$*2$*2$
⟶	[*A*] 3.0 [*A*] 2.0 [*A*] 1.0 1.0 4.0	◁ !()(*C*)@2$*2$*2$
⟶	[*A*] 3.0 [*A*] 2.0 [*A*] 1.0 1.0 [*A*]	◁ ()(*C*)@2$*2$*2$
⟹	[*A*] 3.0 [*A*] 2.0 [*A*] 1.0 1.0 [*A*] []	◁ (*C*)@2$*2$*2$
⟹	[*A*] 3.0 [*A*] 2.0 [*A*] 1.0 []	◁ @2$*2$*2$
⟶	[*A*] 3.0 [*A*] 2.0 [*A*] 1.0	◁ 2$*2$*2$
⟶	[*A*] 3.0 [*A*] 2.0 [*A*] 1.0 2.0	◁ $*2$*2$
⟶	[*A*] 3.0 [*A*] 2.0 1.0	◁ *2$*2$
⟶	[*A*] 3.0 [*A*] 2.0	◁ 2$*2$
⟶	[*A*] 3.0 [*A*] 2.0 2.0	◁ $*2$
⟶	[*A*] 3.0 2.0	◁ *2$
⟶	[*A*] 6.0	◁ 2$
⟶	[*A*] 6.0 2.0	◁ $
⟶	6.0	◁