First Task

Programming Languages
LVA 185.208
2025 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 operands, then the operator. Operands are integers, floating-point numbers or strings, see below. For example, `5.1 12.3+` applies the operator `+` to the operands `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. Operands are simply pushed onto the stack, operators pop operands from the stack and push results onto the stack. For example, the expression `15 2 3 4+*-` is evaluated to 1: First, four integers are pushed onto the stack, then the evaluation of `+` pops `3` and `4` from the stack and pushes `7` onto the stack, the evaluation of `*` pops `2` and `7` from the stack and pushes `14` onto the stack, and finally `-` pops `15` and `14` from the stack and pushes `1` onto the stack.

A sequence of characters enclosed in parentheses is a string. A string is evaluated when applying the operator `@` to it. For example, `(2*)` is a string. When evaluating `4 3(2*)@+`, first `4`, `3` and `(2*)` are pushed onto the stack, then `@` pops the string 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 integer 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 string 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 an integer (or floating-point number to the left of "."), smaller than -1 while reading digits

belonging to a floating-point number to the right of ".", and larger than 0 while reading a string. The operation mode is 0 when switching on the calculator.

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

Register set: A set of 52 read-only registers named by letters A to Z and a to z, each holding a single integer, floating-point number or string. Registers contain predefined values when switching on the calculator, where the predefined value of register a must be a string containing the initial content of the command stream. There are commands for reading 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 an integer, floating-point number, or string, depending on the contents. 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 integers, floating-point numbers, and strings to character sequences and writing them to the output stream.

There can be reasonable limits on the sizes of integers, floating-point numbers, strings, the command stream and the data stack.

Operations

The first character in the command stream, we call it *command 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.

Integer Construction Mode

Operation mode -1 is used for constructing integers from digits. In this mode, the top entry on the data stack must be an integer, and the command 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 command character (0 to 9).

Dot `'.'` converts the integer on the stack to a floating-point number and causes the operation mode to become -2.

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

Decimal Place Construction Mode

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

Digit `'0'` to `'9'` adds the floating-point value of the command 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 floating-point number of value 0.0 onto the data stack, and the operation mode becomes -2 (this is, initiates the construction of a new floating-point number).

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

String Construction Mode

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

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

`')'` adds `')'` to the string 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 command character to the string on top of the data stack.

Execution Mode

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

Digit `'0'` to `'9'` pushes the value of the command character (0 to 9) as an integer onto the data stack, and the operation mode becomes -1.

Dot `'.'` pushes a floating-point number of value 0.0 onto the data stack, and the operation mode becomes -2.

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

Letter `'A'` to `'Z'` or *a* to *z* pushes the contents of the corresponding register A to Z or a to z onto the data stack.

Comparison operation `'='`, `'<'` or `'>'` pops two entries from the data stack, compares them and pushes an integer 0 or 1 onto the data stack. The integer value 0 is regarded as Boolean value *false*, each other integer value as *true*. 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. When comparing two floating-point numbers, these numbers are regarded as equal if they differ at most by epsilon (if both numbers are in the range between -1.0 and 1.0) or epsilon multiplied by the larger absolute value of the two numbers (if the absolute value of at least one of the numbers is larger than 1.0). For comparing an integer with a floating-point number, the integer is converted to a floating-point number and then the two floating-point numbers are compared. Two integers are compared accurately. Two strings are compared according to the lexicographical ordering. When comparing a string with a number (integer or floating-point number), the number is smaller than the string. We can decide if a value is a number or string by comparing the value with the empty string `()`.

Arithmetic or string 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 two integers (resulting in an integer) or two floating-point numbers (resulting in a floating-point number). If one operand is an integer and the other a floating-point number, the integer is converted to a floating-point number before executing the operation. The ordering of operands has to be considered for non-associative operations: `4 2-` and `4 2/` have `2` as result. `'%'` stands for the rest of a division; an application of `'%'` to floating-point numbers results in `()`. When applied to strings, `'+'` performs string concatenation. If only one argument of `'+'` is a string, the other argument is converted to a string representation of the number and then the two strings are concatenated. If `'*'` is applied to a string and an integer n in the range between 0 and 128, the result is the string argument extended with the character corresponding to the ASCII code n at the begin (the first argument is an integer) or end (otherwise). If `'-'` is applied to a string and a positive integer n, the result is a string equal to the string argument after removing n characters from the begin (if the first argument is an integer) or end (otherwise). If `'/'` is applied to two strings, the result is an integer giving the first position where the second string occurs in the first string, or -1 if it does not occur. If `'%'` is applied to a string and a positive integer, the result is the ASCII code of

the character at index n in the string. The empty string `()` is pushed to the data stack if there is no useful semantics of the operation, or a number should be divided by 0 or a small floating-point number between −epsilon and epsilon.

Logic operation `'&'` or `'|'` pops two entries from the data stack, applies the logic operation on them and pushes the result to the data stack, 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 string `()` is pushed to the data stack if an operand is not an integer.

Null-Check `'_'` (underline) pops a value from the data stack and pushes 1 onto the data stack if the popped value is the empty string, the integer 0, or a floating-point number between −epsilon and epsilon, otherwise pushes 0 onto the data stack. This operator can be used to negate Booleans.

Negation `'~'` changes the sign of the top entry on the data stack if it is an integer or floating-point number, otherwise it replaces the top entry with the empty string `()`.

Integer conversion `'?'` replaces the top entry of the stack, which is a floating-point number, with a corresponding truncated number as an integer. Replaces the top entry, which is a string or an integer, with the empty string `()`.

Copy `'!'` replaces the top entry n (an integer) on the data stack with a copy of the nth entry on the data stack (counted from the top of stack). There is no effect if n is not an integer or n is not in the appropriate range.

Delete `'$'` pops the top entry n (an integer) from the data stack and then removes the nth entry from the data stack (counted from the top of the stack). Pops only from the data stack if the top entry is not an integer in the appropriate range.

Apply immediately `'@'` pops a string from the data stack (if the top entry is a string) and inserts the string contents at the begin of the command stream to be executed next. There is no effect if the top entry is not a string.

Apply later `'\'` pops a string from the data stack (if the top entry is a string) and inserts its contents at the end of the command stream to be executed after everything else currently in this stream. There is no effect if the top entry is not a string.

Stack size `'#'` pushes the current number of stack entries onto the stack (as an integer).

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 an integer, the input value is the corresponding integer. Otherwise if the characters in the line can be interpreted as a floating-point number, the input value is the corresponding floating-point number. Otherwise the input value is a string containing the sequence of ASCII characters in the line. None-ASCII characters shall be ignored. Please note that strings can contain unbalanced parentheses. 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 string, the characters in the string are directly written to the output stream (without additional parentheses). If the value is a number (integer or floating-point number), it is written to the output stream in an appropriate format (beginning with `-` for negative numbers and avoiding unnecessary digits).

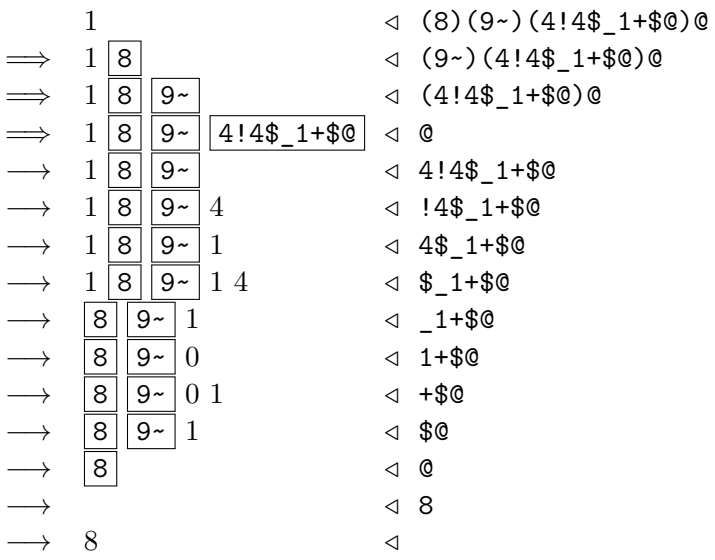
Everything else in the command stream does nothing (except of separating two adjacent numbers from each other). A string can be used as plain text if neither `'@'` nor `\` is applied to it.

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 $\triangleleft$, strings in boxes, the top of the stack adjacent to $\triangleleft$) and its command stream (to the right of $\triangleleft$, the first entry adjacent to $\triangleleft$). Arrows between state specifications show how states change by executing commands ($\longrightarrow$ for one command, $\Longrightarrow$ for several commands).

The first example shows how to deal with conditional execution: On the data stack we expect an integer used as Boolean value. Depending on this value we want to execute the one or the other string. First we push a string for the true-path `(8)` onto the stack, then one for the false-path `(9~)`, finally we execute `(4!4$_1+$@)`, the code for conditional execution. The steps (only those in execution mode) show what happens if previously 1 was on the data stack.



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=()5!(`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. Registers are inappropriate for storing intermediate results.

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 (accessible through registers) to modify and analyze strings typed in at run time. Each character in a string is classified as either letter, digit, white-space character or special character. Each longest possible character sequence in a string consisting only of letters and digits is a *word*. The program output consists of

- the input string where each occurring word is reversed,
- and the numbers of words, letters, digits, white-space characters and special characters in the string (for each categorie counted separately).

For example, if the input is `abc+25 a3/X)$` (without enclosing parentheses), the output string shall be `cba+52 3a/X)$`. This string contains 4 words, 5 letters, 3 digits, 1 white-space character and 3 special characters. Please use the data stack to hold all data needed in the computation.

Appendix (Compute Factorial of 3)

	3	◁ (A)3!3$3!@2$
$\Longrightarrow$	3 [A]	◁ 3!3$3!@2$
$\longrightarrow$	3 [A] 3	◁ !3$3!@2$
$\longrightarrow$	3 [A] 3	◁ 3$3!@2$
$\longrightarrow$	3 [A] 3 3	◁ $3!@2$
$\longrightarrow$	[A] 3	◁ 3!@2$
$\longrightarrow$	[A] 3 3	◁ !@2$
$\longrightarrow$	[A] 3 [A]	◁ @2$
$\longrightarrow$	[A] 3	◁ 3!3!1-2!1=()5!(C)@2$*2$
$\longrightarrow$	[A] 3 3	◁ !3!1-2!1=()5!(C)@2$*2$
$\longrightarrow$	[A] 3 [A]	◁ 3!1-2!1=()5!(C)@2$*2$
$\longrightarrow$	[A] 3 [A] 3	◁ !1-2!1=()5!(C)@2$*2$
$\longrightarrow$	[A] 3 [A] 3	◁ 1-2!1=()5!(C)@2$*2$
$\longrightarrow$	[A] 3 [A] 3 1	◁ -2!1=()5!(C)@2$*2$
$\longrightarrow$	[A] 3 [A] 2	◁ 2!1=()5!(C)@2$*2$
$\longrightarrow$	[A] 3 [A] 2 2	◁ !1=()5!(C)@2$*2$
$\longrightarrow$	[A] 3 [A] 2 2	◁ 1=()5!(C)@2$*2$
$\longrightarrow$	[A] 3 [A] 2 2 1	◁ =()5!(C)@2$*2$
$\longrightarrow$	[A] 3 [A] 2 0	◁ ()5!(C)@2$*2$
$\Longrightarrow$	[A] 3 [A] 2 0 [ ]	◁ 5!(C)@2$*2$
$\longrightarrow$	[A] 3 [A] 2 0 [ ] 5	◁ !(C)@2$*2$
$\longrightarrow$	[A] 3 [A] 2 0 [ ] [A]	◁ (C)@2$*2$
$\Longrightarrow$	[A] 3 [A] 2 [A]	◁ @2$*2$
$\longrightarrow$	[A] 3 [A] 2	◁ 3!3!1-2!1=()5!(C)@2$*2$*2$
$\longrightarrow$	[A] 3 [A] 2 3	◁ !3!1-2!1=()5!(C)@2$*2$*2$
$\longrightarrow$	[A] 3 [A] 2 [A]	◁ 3!1-2!1=()5!(C)@2$*2$*2$
$\longrightarrow$	[A] 3 [A] 2 [A] 3	◁ !1-2!1=()5!(C)@2$*2$*2$
$\longrightarrow$	[A] 3 [A] 2 [A] 2	◁ 1-2!1=()5!(C)@2$*2$*2$
$\longrightarrow$	[A] 3 [A] 2 [A] 2 1	◁ -2!1=()5!(C)@2$*2$*2$
$\longrightarrow$	[A] 3 [A] 2 [A] 1	◁ 2!1=()5!(C)@2$*2$*2$
$\longrightarrow$	[A] 3 [A] 2 [A] 1 2	◁ !1=()5!(C)@2$*2$*2$
$\longrightarrow$	[A] 3 [A] 2 [A] 1 1	◁ 1=()5!(C)@2$*2$*2$
$\longrightarrow$	[A] 3 [A] 2 [A] 1 1 1	◁ =()5!(C)@2$*2$*2$
$\longrightarrow$	[A] 3 [A] 2 [A] 1 1	◁ ()5!(C)@2$*2$*2$
$\Longrightarrow$	[A] 3 [A] 2 [A] 1 1 [ ]	◁ 5!(C)@2$*2$*2$
$\longrightarrow$	[A] 3 [A] 2 [A] 1 1 [ ] 5	◁ !(C)@2$*2$*2$
$\longrightarrow$	[A] 3 [A] 2 [A] 1 1 [ ] [A]	◁ (C)@2$*2$*2$
$\Longrightarrow$	[A] 3 [A] 2 [A] 1 [ ]	◁ @2$*2$*2$
$\longrightarrow$	[A] 3 [A] 2 [A] 1	◁ 2$*2$*2$
$\longrightarrow$	[A] 3 [A] 2 [A] 1 2	◁ $*2$*2$
$\longrightarrow$	[A] 3 [A] 2 1	◁ *2$*2$
$\longrightarrow$	[A] 3 [A] 2	◁ 2$*2$
$\longrightarrow$	[A] 3 [A] 2 2	◁ $*2$
$\longrightarrow$	[A] 3 2	◁ *2$
$\longrightarrow$	[A] 6	◁ 2$
$\longrightarrow$	[A] 6 2	◁ $
$\longrightarrow$	6	◁