Programmiersprachen – Programming Languages
LVA 185.208, VU, 3 ECTS, 2014 S

First Exercise

Exercise

Develop a simulator of a programmable calculator according to the following specification, and write the programs for the calculator described below. To implement the calculator you can use any language you like, but the programs for the calculator must be written in the special language of the calculator.

How to use the calculator

The calculator uses post-fix notation (first the arguments, then the operator) where the operator determines the number of arguments. Arguments can be integers as well as blocks in brackets (see below). For example, “5 12+” applies the operator “+” to the arguments 5 and 12, giving 17 as result. White space between 5 and 12 is necessary to separate the integers from each other.

Post-fix expressions are evaluated using a stack: Arguments are simply pushed onto the stack, operators take arguments from the stack and push results onto the stack. For example, “1 2 3 4+*-” is evaluated to 13: First, the four numbers are pushed onto the stack, then the evaluation of “+” takes 4 and 3 from the top of the stack and pushes the result 7 onto the stack, the evaluation of “*” takes 7 and 2 from the stack and pushes 14 onto the stack, and finally “-” takes 14 and 1 from the stack and pushes 13 onto the stack.

Data and operators enclosed in square brackets build blocks. Operators in blocks are not evaluated immediately. However, a block is evaluated when applying the operator “a” to it. For example, “[2*]” is a block. When evaluating “4 3[2*]a+”, first 4, 3 and “[2*]” are pushed onto the stack, then “a” takes the block from the stack and causes its contents to be evaluated by pushing 2 onto the stack and applying “*” to 2 and 3, and finally “+” adds 4 to 6. We can regard “[2*]a” as an operator that doubles the element on top of the stack. Hence, using blocks we can use operators as data as well as create operators within the language of the calculator.

Architecture of the calculator

The calculator consists of the following parts:

Input stream:
A stream of integers represents the characters typed into the keyboard

(ASCII codes).

Output stream:
: A display of limited size (e.g., 25 lines and 80 columns) shows the characters (represented by integers as ASCII codes) written to the output stream. If the display is not large enough to show all data, only the most recent data are displayed.

Data stack:
: This is the stack holding data when evaluating expressions in post-fix notation.

Code stack:
: This is the area containing the program code to be executed. The operation to be executed next is on top of the stack.

For simplification there can be (reasonable) limits for the sizes of integers, blocks, the data stack and the code stack.

Operations

The following operations shall be implemented.

Integers and blocks:
: When used as operations – this is, when occurring on top of the code stack – they are simply taken from the code stack and pushed onto the data stack.

Arithmetic, comparison and logical operators (“+”, “-”, “*”, “/”, “%”, “&”, “|”, “=”, “<”, “>”):
: These binary operators have the usual semantics, where “%” computes the rest of a division, “&” is the logical AND and “|” is the logical OR. Each of these operators takes two integers from the data stack and pushes an integer as result onto the data stack. The integer 0 corresponds to “false” and 1 to “true”. An error is reported if any of the two topmost elements on the data stack is no integer or an argument of “&” or “|” differs from 0 and 1. The comparison operator “=” is an exception: If “=” is applied to two equal blocks or two equal integers, then the result equals 1, otherwise the result is 0. The ordering of arguments has to be considered for non-associative operations: “2 4-” and “2 4/” and “4 2%” have 2 as result, and “2 4>” and “4 2<” give 1 (“true”). An error shall be reported when executing “/” or “%” if the second element from the top of the data stack equals 0.

Negation “~”:
: This unary operator changes the sign of an integer. An error is reported if its argument is not an integer.

Copy “c”
: replaces the top element *n* of the data stack with a copy of the *n*th element on the data stack (counted from the top of the stack). An error is reported if *n* is not a positive number or the stack does not contain a sufficient number of elements.

Delete “d”
: takes the top element *n* from the data stack and additionally removes the *n*th element from the data stack (counted from the top of the stack). An error is reported if *n* is not a positive number or the stack does not contain a sufficient number of elements.

Apply “a”:
: If the single argument is a block, then the block is taken from the data stack and its content is pushed onto the code stack so that the operations in the block are executed next. Otherwise the operation has no effect.

Read “r”
: takes the next character code from the input stream and pushes it onto the data stack.

Write “w”
: takes an integer from the top of the data stack and writes it as character code into the output stream. An error is reported if the argument is not a number in the range of character codes.

Group “g”
: takes two arguments from the data stack and composes a new block from them. If both arguments are blocks, then their contents are simply appended. If one argument is a block and the other an integer, then the integer is inserted into the block. If both arguments are integers, then a new block consisting of these integers is constructed.

Build block “b”
: takes an argument from the data stack and creates a new block. If the argument is a block, then a new block containing just this block is created. If the argument is an integer corresponding to the ASCII code of a character representing an operation, then the result is a block containing just this operation. Otherwise an error is reported.

Exit “x”
: stops the execution of the calculator.

If an error occurs, the calculator simulator shall simply stop its execution and report an error message.

Examples

The following examples clarify the use of some operators and introduce specific programming techniques. We specify the state of the calculator by the contents of its data stack (to the left of “^”, the top of the stack adjacent to “^”) and its code stack (to the right of “^”, the top of the stack adjacent to “^”). Arrows between state specifications show how the state changes by executing operations.

The first example shows how to deal with conditional execution: On the data stack we expect 0 (false) or 1 (true). Depending on this value we want to execute the one or the other block. First we push the block for the false-path “[9~]” onto the stack, then the one for the true-path “[9]”, and finally we

execute “[3c4d1+da]”, the code for conditional execution. The steps show what happens if previously 0 was on the data stack.

```
    0 ^ [9~][9][3c4d1+da]a
--> 0[9~] ^ [9][3c4d1+da]a
--> 0[9~][9] ^ [3c4d1+da]a
--> 0[9~][9][3c4d1+da] ^ a
--> 0[9~][9] ^ 3c4d1+da
--> 0[9~][9]3 ^ c4d1+da
--> 0[9~][9]0 ^ 4d1+da
--> 0[9~][9]0 4 ^ d1+da
--> [9~][9]0 ^ 1+da
--> [9~][9]0 1 ^ +da
--> [9~][9]1 ^ da
--> [9~] ^ a
--> ^ 9~
--> 9 ^ ~
--> -9 ^
```

The next example shows recursion in the computation of 3 factorial. For simplification we use A as shorthand for [2c1 3c-1c1=3c[]Ca2d*] and C as shorthand for [3c4d1+da] (see above). Please note that the only purpose of A and C is a simplification to improve readability. In the calculator the blocks occur instead of their shorthands.

```
    3 ^ A2c3d2ca2d
--> 3A ^ 2c3d2ca2d
--> 3A2 ^ c3d2ca2d
--> 3A3 ^ 3d2ca2d
--> 3A3 3 ^ d2ca2d
--> A3 ^ 2ca2d
--> A3 2 ^ ca2d
--> A3A ^ a2d
--> A3 ^ 2c1 3c-1c1=3c[]Ca2d*2d
--> A3 2 ^ c1 3c-1c1=3c[]Ca2d*2d
--> A3A ^ 1 3c-1c1=3c[]Ca2d*2d
--> A3A1 ^ 3c-1c1=3c[]Ca2d*2d
--> A3A1 3 ^ c-1c1=3c[]Ca2d*2d
--> A3A1 3 ^ -1c1=3c[]Ca2d*2d
--> A3A2 ^ 1c1=3c[]Ca2d*2d
--> A3A2 1 ^ c1=3c[]Ca2d*2d
--> A3A2 2 ^ 1=3c[]Ca2d*2d
--> A3A2 2 1 ^ =3c[]Ca2d*2d
--> A3A2 0 ^ 3c[]Ca2d*2d
--> A3A2 0 3 ^ c[]Ca2d*2d
--> A3A2 0A ^ []Ca2d*2d
--> A3A2 0A[] ^ Ca2d*2d
--> A3A2 0A[]C ^ a2d*2d
...
--> A3A2A ^ a2d*2d
--> A3A2 ^ 2c1 3c-1c1=3c[]Ca2d*2d*2d
--> A3A2 2 ^ c1 3c-1c1=3c[]Ca2d*2d*2d
--> A3A2A ^ 1 3c-1c1=3c[]Ca2d*2d*2d
--> A3A2A1 ^ 3c-1c1=3c[]Ca2d*2d*2d
--> A3A2A1 3 ^ c-1c1=3c[]Ca2d*2d*2d
--> A3A2A1 2 ^ -1c1=3c[]Ca2d*2d*2d
--> A3A2A1 ^ 1c1=3c[]Ca2d*2d*2d
```

```
--> A3A2A1 1 ^ c1=3c[]Ca2d*2d*2d
--> A3A2A1 1 ^ 1=3c[]Ca2d*2d*2d
--> A3A2A1 1 1 ^ =3c[]Ca2d*2d*2d
--> A3A2A1 1 ^ 3c[]Ca2d*2d*2d
--> A3A2A1 1 3 ^ c[]Ca2d*2d*2d
--> A3A2A1 1A ^ []Ca2d*2d*2d
--> A3A2A1 1A[] ^ Ca2d*2d*2d
--> A3A2A1 1A[]C ^ a2d*2d*2d
...
--> A3A2A1[] ^ a2d*2d*2d
--> A3A2A1 ^ 2d*2d*2d
--> A3A2A1 2 ^ d*2d*2d
--> A3A2 1 ^ *2d*2d
--> A3A2 ^ 2d*2d
--> A3A2 2 ^ d*2d
--> A3 2 ^ *2d
--> A6 ^ 2d
--> A6 2 ^ d
--> 6 ^
```

User interface

Write a user interface for the calculator. Users shall be able to type in commands (simple arithmetic calculations as well as complete programs to be executed) through the input stream and get results through the output stream. The input format shall not be overly restrictive, and the output shall be easily readable. At the begin and the end of execution appropriate greeting messages shall be displayed. All parts of the user interface must be written solely in the language of the calculator. On startup the calculator simulator shall automatically load the user interface into the code stack.

Testing

To test the calculator and its user interface write a prime number test in the language of the calculator. This is, develop a string understood by the user interface that instructs the calculator to ask for a number, decide if this number is prime and show the result in a readable form (not just 0 or 1) on the display. The user interface shall not implement any part of this test program directly, but the user interface shall be powerful enough to accept corresponding input. Please try the prime number test out with several numbers (up to at least 1000).