Parsing/RPN calculator algorithm

Parsing/RPN calculator algorithm
You are encouraged to solve this task according to the task description, using any language you may know.
Task

Create a stack-based evaluator for an expression in reverse Polish notation (RPN) that also shows the changes in the stack as each individual token is processed *as a table*.

- Assume an input of a correct, space separated, string of tokens of an RPN expression
- Test with the RPN expression generated from the Parsing/Shunting-yard algorithm task:

```
3 4 2 * 1 5 - 2 3 ^ ^ / +
```

- Print or display the output here

Notes

- ^ means exponentiation in the expression above.
- / means division.

See also

- Parsing/Shunting-yard algorithm for a method of generating an RPN from an infix expression.
- Several solutions to 24 game/Solve make use of RPN evaluators (although tracing how they work is not a part of that task).
- Parsing/RPN to infix conversion.
- Arithmetic evaluation.

Contents

- 1 Ada
- 2 ALGOL 68
- 3 ANTLR
- 3.1 Java
- 4 AutoHotkey
- 5 BBC BASIC
- 6 C
- 7 C++
- 8 C#
- 9 Ceylon

- 10 Clojure
- 11 Common Lisp
- 12 EchoLisp
- 13 Ela
- 14 D
- 15 Erlang
- 16 F#
- 17 Fortran
- 18 FunL
- 19 Go
- 20 Groovy
- 21 Haskell
- 22 Icon and Unicon
- 23 J
    - 23.1 Alternate Implementation
- 24 Java
- 25 JavaScript
    - 25.1 With checks and messages
- 26 Julia
- 27 Liberty BASIC
- 28 Mathematica
- 29 NetRexx
- 30 Nim
- 31 Objeck
- 32 OCaml
- 33 Oforth
- 34 ooRexx
- 35 Perl
- 36 Perl 6
- 37 Phix
- 38 PHP
- 39 PicoLisp
- 40 PL/I
- 41 PowerShell
- 42 Prolog
- 43 Python
    - 43.1 Version 1
    - 43.2 Version 2
- 44 Racket
- 45 REXX
    - 45.1 version 1
    - 45.2 version 2
    - 45.3 version 3 (error checking)
- 46 Ruby
- 47 Run BASIC
- 48 Scala

- 49 Sidef
- 50 Swift
- 51 Tcl
- 52 VBA
- 53 zkl

Ada

```
with Ada.Text_IO, Ada.Containers.Vectors;

procedure RPN_Calculator is

  package IIO is new Ada.Text_IO.Float_IO(Float);

   package Float_Vec is new Ada.Containers.Vectors
     (Index_Type => Positive, Element_Type => Float);
   Stack: Float_Vec.Vector;

   Input: String := Ada.Text_IO.Get_Line;
   Cursor: Positive := Input'First;
   New_Cursor: Positive;

begin
   loop
      -- read spaces
      while Cursor <= Input'Last and then Input(Cursor)=' ' loop
         Cursor := Cursor + 1;
      end loop;

      exit when Cursor > Input'Last;

      New_Cursor := Cursor;
      while New_Cursor <= Input'Last and then Input(New_Cursor) /= ' ' loop
         New_Cursor := New_Cursor + 1;
      end loop;

      -- try to read a number and push it to the stack
      declare
         Last: Positive;
         Value: Float;
         X, Y: Float;
      begin
         IIO.Get(From => Input(Cursor .. New_Cursor - 1),
                 Item => Value,
                 Last => Last);
         Stack.Append(Value);
         Cursor := New_Cursor;

      exception -- if reading the number fails, try to read an operator token
         when others =>
            Y := Stack.Last_Element; Stack.Delete_Last; -- pick two elements
            X := Stack.Last_Element; Stack.Delete_Last; -- from the stack
            case Input(Cursor) is
               when '+' => Stack.Append(X+Y);
               when '-' => Stack.Append(X-Y);
               when '*' => Stack.Append(X*Y);
               when '/' => Stack.Append(X/Y);
               when '^' => Stack.Append(X ** Integer(Float'Rounding(Y)));
               when others => raise Program_Error with "unecpcted token '"
                  & Input(Cursor) & "' at column" & Integer'Image(Cursor);
            end case;
            Cursor := New_Cursor;
      end;

      for I in Stack.First_Index .. Stack.Last_Index loop
         Ada.Text_IO.Put(" ");
         IIO.Put(Stack.Element(I), Aft => 5, Exp => 0);
      end loop;
      Ada.Text_IO.New_Line;
   end loop;

   Ada.Text_IO.Put("Result = ");
   IIO.Put(Item => Stack.Last_Element, Aft => 5, Exp => 0);


end RPN_Calculator;
```

Output:

```
3 4 2 * 1 5 - 2 3 ^ ^ / +
  3.00000
  3.00000  4.00000
  3.00000  4.00000  2.00000
  3.00000  8.00000
  3.00000  8.00000  1.00000
  3.00000  8.00000  1.00000  5.00000
  3.00000  8.00000 -4.00000
  3.00000  8.00000 -4.00000  2.00000
  3.00000  8.00000 -4.00000  2.00000  3.00000
  3.00000  8.00000 -4.00000  8.00000
  3.00000  8.00000 65536.00000
  3.00000  0.00012
  3.00012
Result =  3.00012
```

ALGOL 68

Works with: ALGOL 68G version Any - tested with release 2.8.win32

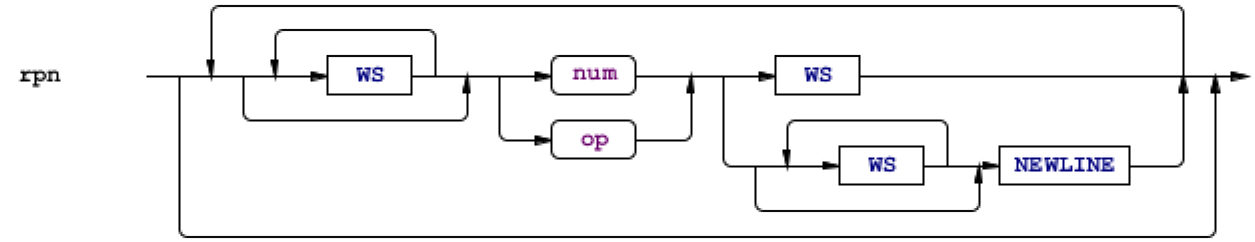
```
# RPN Expression evaluator - handles numbers and + - * / ^    #
#     the right-hand operand for ^ is converted to an integer #

# expression terminator #
CHAR end of expression character = REPR 12;

# evaluates the specified rpn expression #
PROC evaluate = ( STRING rpn expression )VOID:
BEGIN

    [ 256 ]REAL   stack;
    INT           stack pos := 0;
    # pops an element off the stack #
    PROC pop = REAL:
    BEGIN
        stack pos -:= 1;
        stack[ stack pos + 1 ]
    END; # pop #

    INT rpn pos := LWB rpn expression;

    # evaluate tokens from the expression until we get the end of expression
#
    WHILE

        # get the next token from the string #

        STRING token type;
        REAL   value;

        # skip spaces #
        WHILE rpn expression[ rpn pos ] = " "
        DO
            rpn pos +:= 1
        OD;

        # handle the token #
        IF rpn expression[ rpn pos ] = end of expression character
        THEN
            # no more tokens #
            FALSE

        ELSE
            # have a token #

            IF  rpn expression[ rpn pos ] >= "0"
            AND rpn expression[ rpn pos ] <= "9"
            THEN
                # have a number #

                # find where the nmumber is in the expression #
                INT  number start = rpn pos;
                WHILE (   rpn expression[ rpn pos ] >= "0"
                      AND rpn expression[ rpn pos ] <= "9"
                      )
                   OR rpn expression[ rpn pos ] = "."
                DO
                    rpn pos +:= 1
                OD;

                # read the number from the expression #
                FILE number f;
                associate( number f
                         , LOC STRING := rpn expression[ number start : rpn
pos - 1 ]
                         );
                get( number f, ( value ) );
                close( number f );
                token type := "number"

            ELSE
                # must be an operator #
                CHAR op      = rpn expression[ rpn pos ];
                rpn pos    +:= 1;

                REAL arg1   := pop;
                REAL arg2   := pop;
                token type  := op;

                value := IF   op = "+"
                         THEN
                             # add the top two stack elements #
                             arg1 + arg2
                         ELIF op = "-"
                         THEN
                             # subtract the top two stack elements #
                             arg2 - arg1
                         ELIF op = "*"
                         THEN
                             # multiply the top two stack elements #
                             arg2 * arg1
                         ELIF op = "/"
                         THEN
                             # divide the top two stack elements #
                             arg2 / arg1
                         ELIF op = "^"
                         THEN
                             # raise op2 to the power of op1 #
                             arg2 ^ ENTIER arg1
                         ELSE
                             # unknown operator #
                             print( ( "Unknown operator: """ + op + """",
newline ) );
                             0
                         FI

            FI;

            TRUE
        FI
    DO
        # push the new value on the stack and show the new stack #

        stack[ stack pos +:= 1 ] := value;

        print( ( ( token type + "           " )[ 1 : 8 ] ) );
        FOR element FROM LWB stack TO stack pos
        DO
            print( ( " ", fixed( stack[ element ], 8, 4 ) ) )
        OD;
        print( ( newline ) )

    OD;

    print( ( "Result is: ", fixed( stack[ stack pos ], 12, 8 ), newline ) )
```

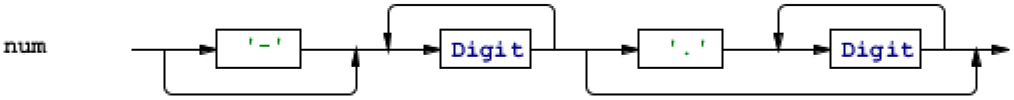
```
END; # evaluate #

main: (

    # get the RPN expresson from the user #

    STRING rpn expression;

    print( ( "Enter expression: " ) );
    read( ( rpn expression, newline ) );

    # add a space to terminate the final token and an expression terminator #
    rpn expression +:= " " + end of expression character;

    # execute the expression #
    evaluate( rpn expression )

)
```

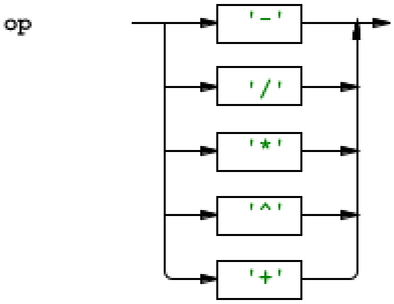
Output:

```
Enter expression: 3 4 2 * 1 5 - 2 3 ^ ^ / +
number    +3.0000
number    +3.0000  +4.0000
number    +3.0000  +4.0000  +2.0000
*         +3.0000  +8.0000
number    +3.0000  +8.0000  +1.0000
number    +3.0000  +8.0000  +1.0000  +5.0000
-         +3.0000  +8.0000  -4.0000
number    +3.0000  +8.0000  -4.0000  +2.0000
number    +3.0000  +8.0000  -4.0000  +2.0000  +3.0000
^         +3.0000  +8.0000  -4.0000  +8.0000
^         +3.0000  +8.0000 +65536.0
/         +3.0000  +0.0001
+         +3.0001
Result is:  +3.00012207
```

ANTLR







Java

```
grammar rpnC ;
//
//  rpn Calculator
//
//  Nigel Galloway - April 7th., 2012
//
@members {
Stack<Double> s = new Stack<Double>();
}
rpn     :       (WS*  (num|op)  (WS | WS* NEWLINE
{System.out.println(s.pop());}))*;
num     :       '-'? Digit+ ('.' Digit+)?
{s.push(Double.parseDouble($num.text));};
Digit   :       '0'..'9';
op      :       '-' {double x = s.pop(); s.push(s.pop() - x);}
        |       '/' {double x = s.pop(); s.push(s.pop() / x);}
        |       '*' {s.push(s.pop() * s.pop());}
        |       '^' {double x = s.pop(); s.push(Math.pow(s.pop(), x));}
        |       '+' {s.push(s.pop() + s.pop());};
WS      :       (' ' | '\t'){skip()};
NEWLINE :       '\r'? '\n';
```

Produces:

```
>java Test
3 4 2 * 1 5 - 2 3 ^ ^ / +
^Z
3.0001220703125
```

AutoHotkey

Works with: AutoHotkey_L

Output is in clipboard.

```
evalRPN("3 4 2 * 1 5 - 2 3 ^ ^ / +")
evalRPN(s){
        stack := []
        out := "For RPN expression: '" s
"'`r`n`r`nTOKEN`t`tACTION`t`t`tSTACK`r`n"
        Loop Parse, s
                If A_LoopField is number
                        t .= A_LoopField
                else
                {
                        If t
                                stack.Insert(t)
                                , out .= t "`tPush num onto top of stack`t"
stackShow(stack) "`r`n"
                                , t := ""
                        If InStr("+-/*^", l := A_LoopField)
                        {
                                a := stack.Remove(), b := stack.Remove()
                                stack.Insert(    l = "+" ? b + a
                                               :l = "-" ? b - a
                                               :l = "*" ? b * a
                                               :l = "/" ? b / a
                                               :l = "^" ? b **a
                                               :0      )
                                out .= l "`tApply op " l " to top of stack`t"
stackShow(stack) "`r`n"
                        }
                }
        r := stack.Remove()
        out .= "`r`n The final output value is: '" r "'"
        clipboard := out
        return r
}
StackShow(stack){
        for each, value in stack
                out .= A_Space value
        return subStr(out, 2)
}
```

Output:

```
For RPN expression: '3 4 2 * 1 5 - 2 3 ^ ^ / +'

TOKEN           ACTION                  STACK
3       Push num onto top of stack      3
4       Push num onto top of stack      3 4
2       Push num onto top of stack      3 4 2
*       Apply op * to top of stack      3 8
1       Push num onto top of stack      3 8 1
5       Push num onto top of stack      3 8 1 5
-       Apply op - to top of stack      3 8 -4
2       Push num onto top of stack      3 8 -4 2
3       Push num onto top of stack      3 8 -4 2 3
^       Apply op ^ to top of stack      3 8 -4 8
^       Apply op ^ to top of stack      3 8 65536
/       Apply op / to top of stack      3 0.000122
+       Apply op + to top of stack      3.000122
```

```
 The final output value is: '3.000122'
```

BBC BASIC

```
@% = &60B
RPN$ = "3 4 2 * 1 5 - 2 3 ^ ^ / +"

DIM Stack(1000)
SP% = 0

FOR i% = 1 TO LEN(RPN$)
  Token$ = MID$(RPN$,i%,1)
  IF Token$ <> " " THEN
    PRINT Token$ " :";
    CASE Token$ OF
      WHEN "+": PROCpush(FNpop + FNpop)
      WHEN "-": PROCpush(-FNpop + FNpop)
      WHEN "*": PROCpush(FNpop * FNpop)
      WHEN "/": n = FNpop : PROCpush(FNpop / n)
      WHEN "^": n = FNpop : PROCpush(FNpop ^ n)
      WHEN "0","1","2","3","4","5","6","7","8","9":
        PROCpush(VALMID$(RPN$,i%))
        WHILE ASCMID$(RPN$,i%)>=48 AND ASCMID$(RPN$,1)<=57
          i% += 1
        ENDWHILE
    ENDCASE
    FOR j% = SP%-1 TO 0 STEP -1 : PRINT Stack(j%); : NEXT
    PRINT
  ENDIF
NEXT i%
END

DEF PROCpush(n)
IF SP% > DIM(Stack(),1) ERROR 100, "Stack full"
Stack(SP%) = n
SP% += 1
ENDPROC

DEF FNpop
IF SP% = 0 ERROR 100, "Stack empty"
SP% -= 1
= Stack(SP%)
```

Output:

```
3 :          3
4 :          4          3
2 :          2          4          3
* :          8          3
1 :          1          8          3
5 :          5          1          8          3
- :         -4          8          3
2 :          2         -4          8          3
3 :          3          2         -4          8          3
^ :          8         -4          8          3
^ :      65536          8          3
/ : 0.00012207          3
```

```
+ :    3.00012
```

C

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

void die(const char *msg)
{
        fprintf(stderr, "%s", msg);
        abort();
}

#define MAX_D 256
double stack[MAX_D];
int depth;

void push(double v)
{
        if (depth >= MAX_D) die("stack overflow\n");
        stack[depth++] = v;
}

double pop()
{
        if (!depth) die("stack underflow\n");
        return stack[--depth];
}

double rpn(char *s)
{
        double a, b;
        int i;
        char *e, *w = " \t\n\r\f";

        for (s = strtok(s, w); s; s = strtok(0, w)) {
                a = strtod(s, &e);
                if (e > s)              printf(" :"), push(a);
#define binop(x) printf("%c:", *s), b = pop(), a = pop(), push(x)
                else if (*s == '+')     binop(a + b);
                else if (*s == '-')     binop(a - b);
                else if (*s == '*')     binop(a * b);
                else if (*s == '/')     binop(a / b);
                else if (*s == '^')     binop(pow(a, b));
#undef binop
                else {
                        fprintf(stderr, "'%c': ", *s);
                        die("unknown oeprator\n");
                }
                for (i = depth; i-- || 0 * putchar('\n'); )
                        printf(" %g", stack[i]);
        }
        if (depth != 1) die("stack leftover\n");

        return pop();
}

int main(void)
{
        char s[] = " 3 4 2 * 1 5 - 2 3 ^ ^ / + ";
        printf("%g\n", rpn(s));
        return 0;
}
```

It's also possible to parse RPN string backwards and recursively; good luck printing out your token stack *as a table*: there isn't one.

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <math.h>

#define die(msg) fprintf(stderr, msg"\n"), abort();
double get(const char *s, const char *e, char **new_e)
{
        const char *t;
        double a, b;

        for (e--; e >= s && isspace(*e); e--);
        for (t = e; t > s && !isspace(t[-1]); t--);

        if (t < s) die("underflow");

#define get2(expr) b = get(s, t, (char **)&t), a = get(s, t, (char **)&t), a = expr
        a = strtod(t, (char **)&e);
        if (e <= t) {
                if      (t[0] == '+') get2(a + b);
                else if (t[0] == '-') get2(a - b);
                else if (t[0] == '*') get2(a * b);
                else if (t[0] == '/') get2(a / b);
                else if (t[0] == '^') get2(pow(a, b));
                else {
                        fprintf(stderr, "'%c': ", t[0]);
                        die("unknown token");
                }
        }
#undef get2

        *(const char **)new_e = t;
        return a;
}

double rpn(const char *s)
{
        const char *e = s + strlen(s);
        double v = get(s, e, (char**)&e);
        while (e > s && isspace(e[-1])) e--;
        if (e == s) return v;

        fprintf(stderr, "\"%.*s\": ", e - s, s);
        die("front garbage");
}

int main(void)
{
        printf("%g\n", rpn("3 4 2 * 1 5 - 2 3 ^ ^ / +"));
        return 0;
}
```

C++

```
#include <vector>
#include <string>
#include <sstream>
#include <iostream>
#include <cmath>
#include <algorithm>
#include <iterator>
#include <cstdlib>

double rpn(const std::string &expr){
  std::istringstream iss(expr);
  std::vector<double> stack;
  std::cout << "Input\tOperation\tStack after" << std::endl;
  std::string token;
  while (iss >> token) {
    std::cout << token << "\t";
    double tokenNum;
    if (std::istringstream(token) >> tokenNum) {
      std::cout << "Push\t\t";
      stack.push_back(tokenNum);
    } else {
      std::cout << "Operate\t\t";
      double secondOperand = stack.back();
      stack.pop_back();
      double firstOperand = stack.back();
      stack.pop_back();
      if (token == "*")
        stack.push_back(firstOperand * secondOperand);
      else if (token == "/")
        stack.push_back(firstOperand / secondOperand);
      else if (token == "-")
        stack.push_back(firstOperand - secondOperand);
      else if (token == "+")
        stack.push_back(firstOperand + secondOperand);
      else if (token == "^")
        stack.push_back(std::pow(firstOperand, secondOperand));
      else { //just in case
        std::cerr << "Error" << std::endl;
        std::exit(1);
      }
    }
    std::copy(stack.begin(), stack.end(),
std::ostream_iterator<double>(std::cout, " "));
    std::cout << std::endl;
  }
  return stack.back();
}

int main() {
  std::string s = " 3 4 2 * 1 5 - 2 3 ^ ^ / + ";
  std::cout << "Final answer: " << rpn(s) << std::endl;

  return 0;
}
```

Output:

```
Input   Operation      Stack after
3       Push           3
4       Push           3 4
2       Push           3 4 2
*       Operate        3 8
1       Push           3 8 1
5       Push           3 8 1 5
-       Operate        3 8 -4
2       Push           3 8 -4 2
3       Push           3 8 -4 2 3
^       Operate        3 8 -4 8
^       Operate        3 8 65536
/       Operate        3 0.00012207
+       Operate        3.00012
Final answer: 3.00012
```

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Globalization;
using System.Threading;

namespace RPNEvaluator
{
    class RPNEvaluator
    {
        static void Main(string[] args)
        {
            Thread.CurrentThread.CurrentCulture =
CultureInfo.InvariantCulture;

            string rpn = "3 4 2 * 1 5 - 2 3 ^ ^ / +";
            Console.WriteLine("{0}\n", rpn);

            decimal result = CalculateRPN(rpn);
            Console.WriteLine("\nResult is {0}", result);
        }
        static decimal CalculateRPN(string rpn)
        {
            string[] rpnTokens = rpn.Split(' ');
            Stack<decimal> stack = new Stack<decimal>();
            decimal number = decimal.Zero;

            foreach (string token in rpnTokens)
            {
                if (decimal.TryParse(token, out number))
                {
                    stack.Push(number);
                }
                else
                {
                    switch (token)
                    {
                        case "^":
                        case "pow":
                            {
                                number = stack.Pop();

stack.Push((decimal)Math.Pow((double)stack.Pop(), (double)number));
                                break;
                            }
                        case "ln":
                            {

stack.Push((decimal)Math.Log((double)stack.Pop(), Math.E));
                                break;
                            }
                        case "sqrt":
                            {

stack.Push((decimal)Math.Sqrt((double)stack.Pop()));
                                break;
                            }
                        case "*":
                            {
                                stack.Push(stack.Pop() * stack.Pop());
                                break;
                            }
                        case "/":
                            {
                                number = stack.Pop();
                                stack.Push(stack.Pop() / number);
                                break;
                            }
                        case "+":
                            {
                                stack.Push(stack.Pop() + stack.Pop());
                                break;
                            }
                        case "-":
                            {
                                number = stack.Pop();
                                stack.Push(stack.Pop() - number);
                                break;
                            }
                        default:
                            Console.WriteLine("Error in CalculateRPN(string)
Method!");
                            break;
                    }
                }
                PrintState(stack);
            }

            return stack.Pop();
        }

        static void PrintState(Stack<decimal> stack)
        {
            decimal[] arr = stack.ToArray();

            for (int i = arr.Length - 1; i >= 0; i--)
            {
                Console.Write("{0,-8:F3}", arr[i]);
            }

            Console.WriteLine();
        }
    }
}
```

Output:

```
3 4 2 * 1 5 - 2 3 ^ ^ / +

3.000
3.000   4.000
3.000   4.000   2.000
3.000   8.000
3.000   8.000   1.000
3.000   8.000   1.000   5.000
3.000   8.000   -4.000
3.000   8.000   -4.000  2.000
3.000   8.000   -4.000  2.000   3.000
3.000   8.000   -4.000  8.000
3.000   8.000   65536.000
3.000   0.000
3.000

Result is 3.0001220703125
```

Ceylon

```
import ceylon.collection {

        ArrayList
}

shared void run() {
        value ops = map {
                "+" -> plus<Float>,
                "*" -> times<Float>,
                "-" -> ((Float a, Float b) => a - b),
                "/" -> ((Float a, Float b) => a / b),
                "^" -> ((Float a, Float b) => a ^ b)
        };

        void printTableRow(String|Float token, String description, {Float*}
stack) {

        print("``token.string.padTrailing(8)````description.padTrailing(30)```
`stack``");
        }

        function calculate(String input) {

                value stack = ArrayList<Float>();
                value tokens = input.split().map((String element)
                        => if(ops.keys.contains(element)) then element else
parseFloat(element));

                print("Token   Operation                     Stack");

                for(token in tokens.coalesced) {
                        if(is Float token) {
                                stack.push(token);
                                printTableRow(token, "push", stack);
                        } else if(exists op = ops[token], exists first =
stack.pop(), exists second = stack.pop()) {
                                value result = op(second, first);
                                stack.push(result);
                                printTableRow(token, "perform ``token`` on
``formatFloat(second, 1, 1)`` and ``formatFloat(first, 1, 1)``", stack);
                        } else {
                                throw Exception("bad syntax");
                        }
                }
                return stack.pop();
        }

        print(calculate("3 4 2 * 1 5 - 2 3 ^ ^ / +"));
}
```

Output:

```
Token   Operation                     Stack
3.0     push                          { 3.0 }
4.0     push                          { 3.0, 4.0 }
2.0     push                          { 3.0, 4.0, 2.0 }
*       perform * on 4.0 and 2.0      { 3.0, 8.0 }
1.0     push                          { 3.0, 8.0, 1.0 }
5.0     push                          { 3.0, 8.0, 1.0, 5.0 }
-       perform - on 1.0 and 5.0      { 3.0, 8.0, -4.0 }
2.0     push                          { 3.0, 8.0, -4.0, 2.0 }
3.0     push                          { 3.0, 8.0, -4.0, 2.0, 3.0 }
^       perform ^ on 2.0 and 3.0      { 3.0, 8.0, -4.0, 8.0 }
^       perform ^ on -4.0 and 8.0     { 3.0, 8.0, 65536.0 }
```

```
/       perform / on 8.0 and 65536.0  { 3.0, 1.220703125E-4 }
+       perform + on 3.0 and 0.0      { 3.0001220703125 }
3.0001220703125
```

Clojure

This would be a lot simpler and generic if we were allowed to use something other than ^ for exponentiation. ^ isn't a legal clojure symbol.

```
(ns rosettacode.parsing-rpn-calculator-algorithm
  (:require clojure.math.numeric-tower
            clojure.string
            clojure.pprint))

(def operators
  "the only allowable operators for our calculator"
  {"+" +
   "-" -
   "*" *
   "/" /
   "^" clojure.math.numeric-tower/expt})

(defn rpn
  "takes a string and returns a lazy-seq of all the stacks"
  [string]
  (letfn [(rpn-reducer [stack item] ; this takes a stack and one item and
makes a new stack
            (if (contains? operators item)
              (let [operand-1 (peek stack) ; if we used lists instead of
vectors, we could use destructuring, but stacks would look backwards
                    stack-1 (pop stack)]   ;we're assuming that all the
operators are binary
                (conj (pop stack-1)
                      ((operators item) (peek stack-1) operand-1)))
              (conj stack (Long. item))))] ; if it wasn't an operator, we'll
assume it's a long. Could choose bigint, or even read-line
    (reductions rpn-reducer [] (clojure.string/split string #"\s+"))))
;reductions is like reduce only shows all the intermediate steps

(let [stacks (rpn "3 4 2 * 1 5 - 2 3 ^ ^ / +")] ;bind it so we can output the
answer separately.
  (println "stacks: ")
  (clojure.pprint/pprint stacks)
  (print "answer:" (->> stacks last first)))
```

Output:

stacks: ([]

```
[3]
[3 4]
[3 4 2]
[3 8]
```

```
[3 8 1]
[3 8 1 5]
[3 8 -4]
[3 8 -4 2]
[3 8 -4 2 3]
[3 8 -4 8]
[3 8 65536]
[3 1/8192]
[24577/8192])
```

answer: 24577/8192

Common Lisp

```
(setf (symbol-function '^) #'expt)  ; Make ^ an alias for EXPT

(defun print-stack (token stack)
    (format T "~a: ~{~a ~}~%" token (reverse stack)))

(defun rpn (tokens &key stack verbose )
  (cond
    ((and (not tokens) (not stack)) 0)
    ((not tokens) (car stack))
    (T
      (let* ((current (car tokens))
             (next-stack (if (numberp current)
                           (cons current stack)
                           (let* ((arg2 (car stack))
                                  (arg1 (cadr stack))
                                  (fun (car tokens)))
                             (cons (funcall fun arg1 arg2) (cddr stack))))))
        (when verbose
          (print-stack current next-stack))
        (rpn (cdr tokens) :stack next-stack :verbose verbose)))))
```

Output:

```
>(defparameter *tokens* '(3 4 2 * 1 5 - 2 3 ^ ^ / +))

*TOKENS*
> (rpn *tokens*)

24577/8192
> (rpn *tokens* :verbose T)
3: 3
4: 3 4
2: 3 4 2
*: 3 8
1: 3 8 1
5: 3 8 1 5
-: 3 8 -4
2: 3 8 -4 2
3: 3 8 -4 2 3
^: 3 8 -4 8
^: 3 8 65536
/: 3 1/8192
+: 24577/8192
```

```
24577/8192
```

EchoLisp

```
;; RPN (postfix) evaluator

(lib 'hash)

(define OPS (make-hash))
(hash-set OPS "^" expt)
(hash-set OPS "*" *)
(hash-set OPS "/" //) ;; float divide
(hash-set OPS "+" +)
(hash-set OPS "-" -)

(define (op? op) (hash-ref OPS op))

;; algorithm :
https://en.wikipedia.org/wiki/Reverse_Polish_notation#Postfix_algorithm

(define (calculator rpn S)
        (for ((token rpn))
        (if (op? token)
                (let [(op2 (pop S)) (op1 (pop S))]
                        (unless (and op1 op2) (error "cannot calculate
expression at:" token))
                (push S ((op? token) op1 op2))
                (writeln op1 token op2 "→" (stack-top S)))
            (push S (string->number token))))
        (pop S))

(define (task rpn)
 (define S (stack 'S))
 (calculator (text-parse rpn) S ))
```

Output:

```
(task "3 4 2 * 1 5 - 2 3 ^ ^ / +")

4       *     2     →     8
1       -     5     →     -4
2       ^     3     →     8
-4      ^     8     →     65536
8     /     65536     →     0.0001220703125
3     +     0.0001220703125     →     3.0001220703125

    → 3.0001220703125

;; RATIONAL CALCULATOR
(hash-set OPS "/" /) ;; rational divide
(task "3 4 2 * 1 5 - 2 3 ^ ^ / +")

4       *     2     →     8
1       -     5     →     -4
2       ^     3     →     8
```

```
-4     ^     8     →     65536
8     /     65536     →     1/8192
3     +     1/8192     →     24577/8192

→ 24577/8192
```

Ela

```
open string generic monad io

type OpType = Push | Operate
  deriving Show

type Op = Op (OpType typ) input stack
  deriving Show

parse str = split " " str

eval stack []      = []
eval stack (x::xs) = op :: eval nst xs
  where (op, nst)  = conv x stack
        conv "+"@x = operate x (+)
        conv "-"@x = operate x (-)
        conv "*"@x = operate x (*)
        conv "/"@x = operate x (/)
        conv "^"@x = operate x (**)
        conv x     = \stack ->
          let n = gread x::stack in
          (Op Push x n, n)
        operate input fn (x::y::ys) =
          let n = (y `fn` x) :: ys in
          (Op Operate input n, n)

print_line (Op typ input stack) = do
  putStr input
  putStr "\t"
  put typ
  putStr "\t\t"
  putLn stack

print ((Op typ input stack)@x::xs) lv = print_line x `seq` print xs (head
stack)
print [] lv = lv

print_result xs = do
  putStrLn "Input\tOperation\tStack after"
  res <- return $ print xs 0
  putStrLn ("Result: " ++ show res)

res = parse "3 4 2 * 1 5 - 2 3 ^ ^ / +" |> eval []
print_result res ::: IO
```

Output:

```
Input   Operation      Stack after
```

```
3       Push            [3]
4       Push            [4,3]
2       Push            [2,4,3]
*       Operate         [8,3]
1       Push            [1,8,3]
5       Push            [5,1,8,3]
-       Operate         [-4,8,3]
2       Push            [2,-4,8,3]
3       Push            [3,2,-4,8,3]
^       Operate         [8,-4,8,3]
^       Operate         [65536,8,3]
/       Operate         [0.0001220703f,3]
+       Operate         [3.000122f]
Result: 3.000122f
```

D

Translation of: Go

```
import std.stdio, std.string, std.conv, std.typetuple;

void main() {
    auto input = "3 4 2 * 1 5 - 2 3 ^ ^ / +";
    writeln("For postfix expression: ", input);
    writeln("\nToken            Action            Stack");
    real[] stack;
    foreach (tok; input.split()) {
        auto action = "Apply op to top of stack";
        switch (tok) {
            foreach (o; TypeTuple!("+", "-", "*", "/", "^")) {
                case o:
                    mixin("stack[$ - 2]" ~
                          (o == "^" ? "^^" : o) ~ "=stack[$ - 1];");
                    stack.length--;
                    break;
            }
            break;
            default:
                action = "Push num onto top of stack";
                stack ~= to!real(tok);
        }
        writefln("%3s    %-26s  %s", tok, action, stack);
    }
    writeln("\nThe final value is ", stack[0]);
}
```

Output:

```
For postfix expression: 3 4 2 * 1 5 - 2 3 ^ ^ / +

Token            Action            Stack
  3    Push num onto top of stack  [3]
  4    Push num onto top of stack  [3, 4]
  2    Push num onto top of stack  [3, 4, 2]
  *    Apply op to top of stack    [3, 8]
  1    Push num onto top of stack  [3, 8, 1]
  5    Push num onto top of stack  [3, 8, 1, 5]
  -    Apply op to top of stack    [3, 8, -4]
  2    Push num onto top of stack  [3, 8, -4, 2]
  3    Push num onto top of stack  [3, 8, -4, 2, 3]
  ^    Apply op to top of stack    [3, 8, -4, 8]
  ^    Apply op to top of stack    [3, 8, 65536]
  /    Apply op to top of stack    [3, 0.00012207]
  +    Apply op to top of stack    [3.00012]

The final value is 3.00012
```

Erlang

```
-module(rpn).
-export([eval/1]).

parse(Expression) ->
    parse(string:tokens(Expression," "),[]).

parse([],Expression) ->
    lists:reverse(Expression);
parse(["+"|Xs],Expression) ->
    parse(Xs,[fun erlang:'+'/2|Expression]);
parse(["-"|Xs],Expression) ->
    parse(Xs,[fun erlang:'-'/2|Expression]);
parse(["*"|Xs],Expression) ->
    parse(Xs,[fun erlang:'*'/2|Expression]);
parse(["/"|Xs],Expression) ->
    parse(Xs,[fun erlang:'/'/2|Expression]);
parse(["^"|Xs],Expression) ->
    parse(Xs,[fun math:pow/2|Expression]);
parse([X|Xs],Expression) ->
    {N,_} = string:to_integer(X),
    parse(Xs,[N|Expression]).

%% The expression should be entered as a string of numbers and
%% operators separated by spaces. No error handling is included if
%% another string format is used.
eval(Expression) ->
    eval(parse(Expression),[]).

eval([],[N]) ->
    N;
eval([N|Exp],Stack) when is_number(N) ->
    NewStack = [N|Stack],
    print(NewStack),
    eval(Exp,NewStack);
eval([F|Exp],[X,Y|Stack]) ->
    NewStack = [F(Y,X)|Stack],
    print(NewStack),
    eval(Exp,NewStack).

print(Stack) ->
    lists:map(fun (X) when is_integer(X) -> io:format("~12.12b ",[X]);
                  (X) when is_float(X) -> io:format("~12f ",[X]) end, Stack),
    io:format("~n").
```

Output:

```
145> rpn:eval("3 4 2 * 1 5 - 2 3 ^ ^ / +").
           3
           4            3
           2            4            3
           8            3
           1            8            3
           5            1            8            3
          -4            8            3
           2           -4            8            3
           3            2           -4            8            3
    8.000000           -4            8            3
65536.000000            8            3
    0.000122            3
    3.000122
3.0001220703125
```

F#

Translation of: OCaml

As interactive script

```
let reduce op = function
  | b::a::r -> (op a b)::r
  | _ -> failwith "invalid expression"

let interprete s = function
  | "+" -> "add",    reduce ( + ) s
  | "-" -> "subtr",  reduce ( - ) s
  | "*" -> "mult",   reduce ( * ) s
  | "/" -> "divide", reduce ( / ) s
  | "^" -> "exp",    reduce ( ** ) s
  | str -> "push", (System.Double.Parse str) :: s

let interp_and_show s inp =
  let op,s'' = interprete s inp
  printf "%5s%8s " inp op
  List.iter (printf " %-6.3F") (List.rev s'')
  printf "\n";
  s''

let eval str =
  printfn "Token  Action  Stack";
  let ss = str.ToString().Split() |> Array.toList
  List.fold interp_and_show [] ss
```

Output:

```
> eval "3 4 2 * 1 5 - 2 3 ^ ^ / +";;
Token  Action  Stack
    3    push  3.000
    4    push  3.000  4.000
    2    push  3.000  4.000  2.000
    *    mult  3.000  8.000
    1    push  3.000  8.000  1.000
    5    push  3.000  8.000  1.000  5.000
    -   subtr  3.000  8.000  -4.000
    2    push  3.000  8.000  -4.000 2.000
    3    push  3.000  8.000  -4.000 2.000  3.000
    ^     exp  3.000  8.000  -4.000 8.000
    ^     exp  3.000  8.000  65536.000
    /  divide  3.000  0.000
    +     add  3.000
val it : float list = [3.00012207]
```

Fortran

Since the project is to demonstrate the workings of the scheme to evaluate a RPN text sequence, and the test example contains only single-digit numbers and single-character operators, there is no need to escalate to reading full integers or floating-point numbers, the code for which would swamp the details of the RPN evaluator. As a result, it is easy to scan the text via a DO-loop that works one character at a time since there is no backstepping, probing ahead, nor multi-symbol items that must be combined into a single "token" with states that must be remembered from one character to the next. With multi-character tokens, the scan would be changed to invocations of NEXTTOKEN that would lurch ahead accordingly.

The method is simple (the whole point of RPN) and the function prints a schedule of actions at each step. Possibly this semi-tabular output is what is meant by "as a table". Conveniently, all the operators take two operands and return one, so the SP accountancy can be shared. Unlike ! for example.

The source style is essentially F77 except for the trivial use of the PARAMETER statement, and CYCLE to GO TO the end of the loop when a space is encountered. With the introduction of unfixed-format source style came also the possible use of semicolons to cram more than one statement part on a line so that the CASE and its action statement can be spread across the page rather than use two lines in alternation: for this case a tabular layout results that is easier to read and check. Because the F90 MODULE protocol is not used, the function's type should be declared in the calling routine but the default type suffices.

```
      REAL FUNCTION EVALRP(TEXT)      !Evaluates a Reverse Polish string.
Caution: deals with single digits only.
       CHARACTER*(*) TEXT      !The RPN string.
       INTEGER SP,STACKLIMIT          !Needed for the evaluation.
       PARAMETER (STACKLIMIT = 6)     !This should do.
       REAL*8 STACK(STACKLIMIT)               !Though with ^ there's no upper
limit.
       INTEGER L,D             !Assistants for the scan.
       CHARACTER*4 DEED               !A scratchpad for the annotation.
       CHARACTER*1 C           !The character of the moment.
        WRITE (6,1) TEXT       !A function that writes messages... Improper.
    1   FORMAT ("Evaluation of the Reverse Polish string ",A,//      !Still,
it's good to see stuff.
     1   "Char Token Action  SP:Stack...")    !Such as a heading for the
trace.
        SP = 0                 !Commence with the stack empty.
        STACK = -666           !This value should cause trouble.
        DO L = 1,LEN(TEXT)     !Step through the text.
          C = TEXT(L:L)                       !Grab a character.
          IF (C.LE." ") CYCLE         !Boring.
          D = ICHAR(C) - ICHAR("0")   !Uncouth test to check for a digit.
          IF (D.GE.0 .AND. D.LE.9) THEN       !Is it one?
            DEED = "Load"                     !Yes. So, load its value.
            SP = SP + 1                               !By going up one.
            IF (SP.GT.STACKLIMIT) STOP "Stack overflow!"     !Or, maybe not.
            STACK(SP) = D                     !And stashing the value.
           ELSE                       !Otherwise, it must be an operator.
            IF (SP.LT.2) STOP "Stack underflow!"      !They all require two
operands.
            DEED = "XEQ"              !So, I'm about to do so.
            SELECT CASE(C)            !Which one this time?
             CASE("+"); STACK(SP - 1) = STACK(SP - 1) + STACK(SP)    !A + B =
B + A, so it is easy.
             CASE("-"); STACK(SP - 1) = STACK(SP - 1) - STACK(SP)    !A is in
STACK(SP - 1), B in STACK(SP)
             CASE("*"); STACK(SP - 1) = STACK(SP - 1)*STACK(SP)
        !Again, order doesn't count.
             CASE("/"); STACK(SP - 1) = STACK(SP - 1)/STACK(SP)
        !But for division, A/B becomes A B /
             CASE("^"); STACK(SP - 1) = STACK(SP - 1)**STACK(SP)     !So, this
way around.
             CASE DEFAULT             !This should never happen!
              STOP "Unknown operator!"!If the RPN script is indeed correct.
            END SELECT                !So much for that operator.
            SP = SP - 1               !All of them take two operands and make
one.
          END IF               !So much for that item.
          WRITE (6,2) L,C,DEED,SP,STACK(1:SP) !Reveal the state now.
    2     FORMAT (I4,A6,A7,I4,":",66F14.6)    !Aligned with the heading of
FORMAT 1.
        END DO                 !On to the next symbol.
        EVALRP = STACK(1)      !The RPN string being correct, this is the
result.
      END      !Simple enough!

      PROGRAM HSILOP
      REAL V
      V = EVALRP("3 4 2 * 1 5 - 2 3 ^ ^ / +") !The specified example.
      WRITE (6,*) "Result is...",V
      END
```

Output...

```
Evaluation of the Reverse Polish string 3 4 2 * 1 5 - 2 3 ^ ^ / +

Char Token Action  SP:Stack...
   1     3   Load   1:      3.000000
   3     4   Load   2:      3.000000      4.000000
   5     2   Load   3:      3.000000      4.000000      2.000000
   7     *   XEQ    2:      3.000000      8.000000
   9     1   Load   3:      3.000000      8.000000      1.000000
  11     5   Load   4:      3.000000      8.000000      1.000000
5.000000
  13     -   XEQ    3:      3.000000      8.000000     -4.000000
  15     2   Load   4:      3.000000      8.000000     -4.000000
2.000000
  17     3   Load   5:      3.000000      8.000000     -4.000000
2.000000      3.000000
  19     ^   XEQ    4:      3.000000      8.000000     -4.000000
8.000000
  21     ^   XEQ    3:      3.000000      8.000000  65536.000000
  23     /   XEQ    2:      3.000000      0.000122
  25     +   XEQ    1:      3.000122
 Result is...   3.000122
```

FunL

```
def evaluate( expr ) =
  stack = []

  for token <- expr.split( '''\s+''' )
    case number( token )
      Some( n ) ->
        stack = n : stack
        println( "push $token: ${stack.reversed()}" )
      None ->
        case {'+': (+), '-': (-), '*': (*), '/': (/), '^': (^)}.>get( token )
          Some( op ) ->
            stack = op( stack.tail().head(), stack.head() ) :
stack.tail().tail()
            println( "perform $token: ${stack.reversed()}" )
          None -> error( "unrecognized operator '$token'" )

  stack.head()

res = evaluate( '3 4 2 * 1 5 - 2 3 ^ ^ / +' )
println( res + (if res is Integer then '' else " or ${float(res)}") )
```

Output:

```
push 3: [3]
push 4: [3, 4]
push 2: [3, 4, 2]
perform *: [3, 8]
push 1: [3, 8, 1]
push 5: [3, 8, 1, 5]
perform -: [3, 8, -4]
push 2: [3, 8, -4, 2]
push 3: [3, 8, -4, 2, 3]
perform ^: [3, 8, -4, 8]
perform ^: [3, 8, 65536]
perform /: [3, 1/8192]
perform +: [24577/8192]
24577/8192 or 3.0001220703125
```

Go

No error checking.

```
package main

import (
    "fmt"
    "math"
    "strconv"
    "strings"
)

var input = "3 4 2 * 1 5 - 2 3 ^ ^ / +"

func main() {
    fmt.Printf("For postfix %q\n", input)
    fmt.Println("\nToken            Action            Stack")
    var stack []float64
    for _, tok := range strings.Fields(input) {
        action := "Apply op to top of stack"
        switch tok {
        case "+":
            stack[len(stack)-2] += stack[len(stack)-1]
            stack = stack[:len(stack)-1]
        case "-":
            stack[len(stack)-2] -= stack[len(stack)-1]
            stack = stack[:len(stack)-1]
        case "*":
            stack[len(stack)-2] *= stack[len(stack)-1]
            stack = stack[:len(stack)-1]
        case "/":
            stack[len(stack)-2] /= stack[len(stack)-1]
            stack = stack[:len(stack)-1]
        case "^":
            stack[len(stack)-2] =
                math.Pow(stack[len(stack)-2], stack[len(stack)-1])
            stack = stack[:len(stack)-1]
        default:
            action = "Push num onto top of stack"
            f, _ := strconv.ParseFloat(tok, 64)
            stack = append(stack, f)
        }
        fmt.Printf("%3s    %-26s  %v\n", tok, action, stack)
    }
    fmt.Println("\nThe final value is", stack[0])
}
```

Output:

```
For postfix "3 4 2 * 1 5 - 2 3 ^ ^ / +"

Token            Action            Stack
  3    Push num onto top of stack  [3]
  4    Push num onto top of stack  [3 4]
  2    Push num onto top of stack  [3 4 2]
  *    Apply op to top of stack    [3 8]
  1    Push num onto top of stack  [3 8 1]
  5    Push num onto top of stack  [3 8 1 5]
  -    Apply op to top of stack    [3 8 -4]
  2    Push num onto top of stack  [3 8 -4 2]
  3    Push num onto top of stack  [3 8 -4 2 3]
  ^    Apply op to top of stack    [3 8 -4 8]
  ^    Apply op to top of stack    [3 8 65536]
  /    Apply op to top of stack    [3 0.0001220703125]
  +    Apply op to top of stack    [3.0001220703125]

The final value is 3.0001220703125
```

Groovy

```
def evaluateRPN(expression) {
    def stack = [] as Stack
    def binaryOp = { action -> return { action.call(stack.pop(), stack.pop())
} }
    def actions = [
        '+': binaryOp { a, b -> b + a },
        '-': binaryOp { a, b -> b - a },
        '*': binaryOp { a, b -> b * a },
        '/': binaryOp { a, b -> b / a },
        '^': binaryOp { a, b -> b ** a }
    ]
    expression.split(' ').each { item ->
        def action = actions[item] ?: { item as BigDecimal }
        stack.push(action.call())

        println "$item: $stack"
    }
    assert stack.size() == 1 : "Unbalanced Expression: $expression ($stack)"
    stack.pop()
}
```

Test

```
println evaluateRPN('3 4 2 * 1 5 - 2 3 ^ ^ / +')
```

Output:

```
3: [3]
4: [3, 4]
2: [3, 4, 2]
*: [3, 8]
1: [3, 8, 1]
5: [3, 8, 1, 5]
-: [3, 8, -4]
2: [3, 8, -4, 2]
3: [3, 8, -4, 2, 3]
^: [3, 8, -4, 8]
^: [3, 8, 65536]
/: [3, 0.0001220703125]
+: [3.0001220703125]
3.0001220703125
```

Haskell

Pure RPN calculator

```
calcRPN :: String -> [Double]
calcRPN = foldl interprete [] . words

interprete s x
  | x `elem` ["+","-","*","/","^"] = operate x s
  | otherwise = read x:s
  where
    operate op (x:y:s) = case op of
      "+" -> x + y:s
      "-" -> y - x:s
      "*" -> x * y:s
      "/" -> y / x:s
      "^" -> y ** x:s
λ> calcRPN "3 4 +"
[7.0]

λ> calcRPN "3 4 2 * 1 5 - 2 3 ^ ^ / +"
[3.0001220703125]
```

Calculation logging

Pure logging. Log as well as a result could be used as a data.

```
calcRPNLog :: String -> ([Double],[(String, [Double])])
calcRPNLog input = mkLog $ zip commands $ tail result
  where result = scanl interprete [] commands
        commands = words input
        mkLog [] = ([], [])
        mkLog res = (snd $ last res, res)
λ> calcRPNLog "3 4 +"
([7.0],[("3",[3.0]),("4",[4.0,3.0]),("+",[7.0])])

λ> mapM_ print $ snd $ calcRPNLog "3 4 2 * 1 5 - 2 3 ^ ^ / +"
("3",[3.0])
("4",[4.0,3.0])
("2",[2.0,4.0,3.0])
("*",[8.0,3.0])
("1",[1.0,8.0,3.0])
("5",[5.0,1.0,8.0,3.0])
("-",[-4.0,8.0,3.0])
("2",[2.0,-4.0,8.0,3.0])
("3",[3.0,2.0,-4.0,8.0,3.0])
("^",[8.0,-4.0,8.0,3.0])
("^",[65536.0,8.0,3.0])
("/",[1.220703125e-4,3.0])
("+",[3.0001220703125])
```

Logging as a side effect. Calculator returns result in IO context:

```
import Control.Monad (foldM)

calcRPNIO :: String -> IO [Double]
calcRPNIO = foldM (verbose interprete) [] . words
```

```
verbose f s x = write (x ++ "\t" ++ show res ++ "\n") >> return res
  where res = f s x
λ> calcRPNIO "3 4 +"
3       [3.0]
4       [4.0,3.0]
+       [7.0]
[7.0]

λ> calcRPNIO "3 4 2 * 1 5 - 2 3 ^ ^ / +"
3       [3.0]
4       [4.0,3.0]
2       [2.0,4.0,3.0]
*       [8.0,3.0]
1       [1.0,8.0,3.0]
5       [5.0,1.0,8.0,3.0]
-       [-4.0,8.0,3.0]
2       [2.0,-4.0,8.0,3.0]
3       [3.0,2.0,-4.0,8.0,3.0]
^       [8.0,-4.0,8.0,3.0]
^       [65536.0,8.0,3.0]
/       [1.220703125e-4,3.0]
+       [3.0001220703125]
[3.0001220703125]
```

Or even more general (requires `FlexibleInstances` and `TypeFamilies` extensions).

Some universal definitions:

```
class Monad m => Logger m where
  write :: String -> m ()

instance Logger IO where write = putStr
instance a ~ String => Logger (Writer a) where write = tell

verbose2 f x y = write (show x ++ " " ++
                        show y ++ " ==> " ++
                        show res ++ "\n") >> return res
  where res = f x y
```

The use case:

```
calcRPNM :: Logger m => String -> m [Double]
calcRPNM = foldM (verbose interprete) [] . words
```

Output:

in REPL

```
λ> calcRPNM "3 4 2 * 1 5 - 2 3 ^ ^ / +"
[] "3" ==> [3.0]
[3.0] "4" ==> [4.0,3.0]
[4.0,3.0] "2" ==> [2.0,4.0,3.0]
[2.0,4.0,3.0] "*" ==> [8.0,3.0]
[8.0,3.0] "1" ==> [1.0,8.0,3.0]
[1.0,8.0,3.0] "5" ==> [5.0,1.0,8.0,3.0]
[5.0,1.0,8.0,3.0] "-" ==> [-4.0,8.0,3.0]
[-4.0,8.0,3.0] "2" ==> [2.0,-4.0,8.0,3.0]
```

```
[2.0,-4.0,8.0,3.0] "3" ==> [3.0,2.0,-4.0,8.0,3.0]
[3.0,2.0,-4.0,8.0,3.0] "^" ==> [8.0,-4.0,8.0,3.0]
[8.0,-4.0,8.0,3.0] "^" ==> [65536.0,8.0,3.0]
[65536.0,8.0,3.0] "/" ==> [1.220703125e-4,3.0]
[1.220703125e-4,3.0] "+" ==> [3.0001220703125]
[3.0001220703125]

λ> runWriter $ calcRPNM "3 4 +"
([7.0],"[] \"3\" ==> [3.0]\n[3.0] \"4\" ==> [4.0,3.0]\n[4.0,3.0] \"+\" ==>
[7.0]\n")
```

Icon and Unicon

```
procedure main()
   EvalRPN("3 4 2 * 1 5 - 2 3 ^ ^ / +")
end

link printf
invocable all

procedure EvalRPN(expr)          #: evaluate (and trace stack) an RPN string

   stack := []
   expr ? until pos(0) do {
      tab(many(' '))                         # consume previous seperator
      token := tab(upto(' ')|0)              # get token
      if token := numeric(token) then {      # ... numeric
         push(stack,token)
         printf("pushed numeric   %i : %s\n",token,list2string(stack))
         }
      else {                                 # ... operator
         every b|a := pop(stack)             # pop & reverse operands
         case token of {
            "+"|"-"|"*"|"^"   : push(stack,token(a,b))
            "/"               : push(stack,token(real(a),b))
            default           : runerr(205,token)
            }
         printf("applied operator %s : %s\n",token,list2string(stack))
         }
   }
end

procedure list2string(L)         #: format list as a string
   every (s := "[ ") ||:= !L || " "
   return s || "]"
end
```

Library: Icon Programming Library

printf.icn provides formatting

Output:

```
pushed numeric   3 : [ 3 ]
pushed numeric   4 : [ 4 3 ]
pushed numeric   2 : [ 2 4 3 ]
```

```
applied operator * : [ 8 3 ]
pushed numeric   1 : [ 1 8 3 ]
pushed numeric   5 : [ 5 1 8 3 ]
applied operator - : [ -4 8 3 ]
pushed numeric   2 : [ 2 -4 8 3 ]
pushed numeric   3 : [ 3 2 -4 8 3 ]
applied operator ^ : [ 8 -4 8 3 ]
applied operator ^ : [ 65536 8 3 ]
applied operator / : [ 0.0001220703125 3 ]
applied operator + : [ 3.0001220703125 ]
```

J

Offered operations are all dyadic - having two arguments. So on each step we may either "shift" a number to the stack or "reduce" two topmost stack items to one.

The final verb is monad - it takes single argument, which contains both the input and accumulated stack. First, create initial state of the input:

```
   a: , <;._1 ' ' , '3 4 2 * 1 5 - 2 3 ^ ^ / +'
┌┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┐
││3│4│2│*│1│5│-│2│3│^│^│/│+│
└┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┘
```

As an example, let's add monadic operation _ which inverses the sign of the stack top element.

We're going to read tokens from input one by one. Each time we read a token, we're checking if it's a number - in this case we put the number to the stack - or an operation - in this case we apply the operation to the stack. The monad which returns 1 for operation and 0 otherwise is "isOp". Dyad, moving input token to the stack, is "doShift", and applying the operation to the stack is "doApply".

There are 6 operations - one monadic "_" and five dyadic "+", "-", "*", "/", "^". For operation, we need to translate input token into operation and apply it to the stack. The dyad which converts the input token to the operation is "dispatch". It uses two miscellaneous adverbs, one for monadic operations - "mo" - and another for dyadic - "dy".

The RPN driver is monad "consume", which handles one token. The output is the state of the program after the token was consumed - stack in the 0th box, and remaining input afterwards. As a side effect, "consume" is going to print the resulting stack, so running "consume" once for each token will produce intermediate states of the stack.

```
   isOp=: '_+-*/^' e.~ {.@>@{.
   mo=: 1 :'(}: , u@{:) @ ['
   dy=: 1 :'(_2&}. , u/@(_2&{.)) @ ['
   dispatch=: (-mo)`(+dy)`(-dy)`(*dy)`(%dy)`(^dy)@.('_+-*/^' i. {.@>@])
   doShift=: (<@, ".@>@{.) , }.@]
   doApply=: }.@] ,~ [ <@dispatch {.@]
   consume=: [: ([ smoutput@>@{.) >@{. doShift`doApply@.(isOp@]) }.
   consume ^: (<:@#) a: , <;._1 ' ' , '3 4 2 * 1 5 - 2 3 ^ ^ / +'
```

```
3
3 4
3 4 2
3 8
3 8 1
3 8 1 5
3 8 _4
3 8 _4 2
3 8 _4 2 3
3 8 _4 8
3 8 65536
3 0.00012207
3.00012
┌───────┐
│3.00012│
└───────┘
   consume ^: (<:@#) a: , <;._1 ' ' , '3 _ 4 +'
3
_3
_3 4
1
┌─┐
│1│
└─┘
```

Alternate Implementation

```
rpn=: 3 :0
  queue=. |.3 :'|.3 :y 0'::]each;: y
  op=. 1 :'2 (u~/@:{.,}.)S:0 ,@]'
  ops=. +op`(-op)`(*op)`(%op)`(^op)`(,&;)
  choose=. ((;:'+-*/^')&i.@[)
  ,ops@.choose/queue
)
```

Example use:

```
   rpn '3 4 2 * 1 5 - 2 3 ^ ^ / +'
3.00012
```

To see intermediate result stacks, use this variant (the only difference is the definition of 'op'):

```
rpnD=: 3 :0
  queue=. |.3 :'|.3 :y 0'::]each;: y
  op=. 1 :'2 (u~/@:{.,}.)S:0 ,@([smoutput)@]'
  ops=. +op`(-op)`(*op)`(%op)`(^op)`(,&;)
  choose=. ((;:'+-*/^')&i.@[)
  ,ops@.choose/queue
)
```

In other words:

```
   rpnD '3 4 2 * 1 5 - 2 3 ^ ^ / +'
┌──────┐
```

```
│2 4 3│
└─────┘
5 1 8 3
3 2 _4 8 3
8 _4 8 3
65536 8 3
0.00012207 3
3.00012
```

Note that the seed stack is boxed while computed stacks are not. Note that top of stack here is on the left. Note also that adjacent constants are bundled in the parsing phase. Finally, note that the result of rpn (and of rpnD - lines previous to the last line in the rpnD example here are output and not a part of the result) is the final state of the stack - in the general case it may not contain exactly one value.

Java

Works with: Java version 1.5+

Supports multi-digit numbers and negative numbers.

```
import java.util.LinkedList;

public class RPN{
        public static void evalRPN(String expr){
                String cleanExpr = cleanExpr(expr);
                LinkedList<Double> stack = new LinkedList<Double>();
                System.out.println("Input\tOperation\tStack after");
                for(String token:cleanExpr.split("\\s")){
                        System.out.print(token+"\t");
                        Double tokenNum = null;
                        try{
                                tokenNum = Double.parseDouble(token);
                        }catch(NumberFormatException e){}
                        if(tokenNum != null){
                                System.out.print("Push\t\t");
                                stack.push(Double.parseDouble(token+""));
                        }else if(token.equals("*")){
                                System.out.print("Operate\t\t");
                                double secondOperand = stack.pop();
                                double firstOperand = stack.pop();
                                stack.push(firstOperand * secondOperand);
                        }else if(token.equals("/")){
                                System.out.print("Operate\t\t");
                                double secondOperand = stack.pop();
                                double firstOperand = stack.pop();
                                stack.push(firstOperand / secondOperand);
                        }else if(token.equals("-")){
                                System.out.print("Operate\t\t");
                                double secondOperand = stack.pop();
                                double firstOperand = stack.pop();
                                stack.push(firstOperand - secondOperand);
                        }else if(token.equals("+")){
                                System.out.print("Operate\t\t");
                                double secondOperand = stack.pop();
                                double firstOperand = stack.pop();
                                stack.push(firstOperand + secondOperand);
                        }else if(token.equals("^")){
                                System.out.print("Operate\t\t");
                                double secondOperand = stack.pop();
                                double firstOperand = stack.pop();
                                stack.push(Math.pow(firstOperand,
secondOperand));
                        }else{//just in case
                                System.out.println("Error");
                                return;
                        }
                        System.out.println(stack);
                }
                System.out.println("Final answer: " + stack.pop());
        }

        private static String cleanExpr(String expr){
                //remove all non-operators, non-whitespace, and non digit
chars
                return expr.replaceAll("[^\\^\\*\\+\\-\\d/\\s]", "");
        }

        public static void main(String[] args){
                evalRPN("3 4 2 * 1 5 - 2 3 ^ ^ / +");
        }
}
```

Output:

```
Input   Operation       Stack after
3       Push            [3.0]
4       Push            [4.0, 3.0]
2       Push            [2.0, 4.0, 3.0]
*       Operate         [8.0, 3.0]
1       Push            [1.0, 8.0, 3.0]
5       Push            [5.0, 1.0, 8.0, 3.0]
-       Operate         [-4.0, 8.0, 3.0]
2       Push            [2.0, -4.0, 8.0, 3.0]
3       Push            [3.0, 2.0, -4.0, 8.0, 3.0]
^       Operate         [8.0, -4.0, 8.0, 3.0]
^       Operate         [65536.0, 8.0, 3.0]
/       Operate         [1.220703125E-4, 3.0]
+       Operate         [3.0001220703125]
Final answer: 3.0001220703125
```

JavaScript

```
var e = '3 4 2 * 1 5 - 2 3 ^ ^ / +'
var s=[], e=e.split(' ')
for (var i in e) {
        var t=e[i], n=+t
        if (n == t)
                s.push(n)
        else {
                var o2=s.pop(), o1=s.pop()
                switch (t) {
                        case '+': s.push(o1+o2); break;
                        case '-': s.push(o1-o2); break;
                        case '*': s.push(o1*o2); break;
                        case '/': s.push(o1/o2); break;
                        case '^': s.push(Math.pow(o1,o2)); break;
                }
        }
        document.write(t, ': ', s, '<br>')
}
```

Output:

```
3: 3
4: 3,4
2: 3,4,2
*: 3,8
1: 3,8,1
5: 3,8,1,5
-: 3,8,-4
2: 3,8,-4,2
3: 3,8,-4,2,3
^: 3,8,-4,8
^: 3,8,65536
/: 3,0.0001220703125
+: 3.0001220703125
```

With checks and messages

```
var e = '3 4 2 * 1 5 - 2 3 ^ ^ / +'
eval: {
        document.write(e, '<br>')
        var s=[], e=e.split(' ')
        for (var i in e) {
                var t=e[i], n=+t
                if (!t) continue
                if (n == t)
                        s.push(n)
                else {
                        if ('+-*/^'.indexOf(t) == -1) {
                                document.write(t, ': ', s, '<br>', 'Unknown
operator!<br>')
                                break eval
                        }
                        if (s.length<2) {
                                document.write(t, ': ', s, '<br>',
'Insufficient operands!<br>')
                                break eval
                        }
                        var o2=s.pop(), o1=s.pop()
                        switch (t) {
                                case '+': s.push(o1+o2); break
                                case '-': s.push(o1-o2); break
                                case '*': s.push(o1*o2); break
                                case '/': s.push(o1/o2); break
                                case '^': s.push(Math.pow(o1,o2))
                        }
                }
                document.write(t, ': ', s, '<br>')
        }
        if (s.length>1) {
                document.write('Insufficient operators!<br>')
        }
}
```

Output:

```
3 4 2 * 1 5 - 2 3 ^ ^ / +
3: 3
4: 3,4
2: 3,4,2
*: 3,8
1: 3,8,1
5: 3,8,1,5
-: 3,8,-4
2: 3,8,-4,2
3: 3,8,-4,2,3
^: 3,8,-4,8
^: 3,8,65536
/: 3,0.0001220703125
+: 3.0001220703125
```

Julia

(This code takes advantage of the fact that all of the operands and functions in the specified RPN syntax are valid Julia expressions, so we can use the built-in `parse` and `eval` functions to turn them into numbers and the corresponding Julia functions.)

```
function rpn(s)
    stack = Any[]
    for op in map(eval, map(parse, split(s)))
        if isa(op, Function)
            arg2 = pop!(stack)
            arg1 = pop!(stack)
            push!(stack, op(arg1, arg2))
        else
            push!(stack, op)
        end
        println("$op: ", join(stack, ", "))
    end
    length(stack) != 1 && error("invalid RPN expression $s")
    return stack[1]
end
rpn("3 4 2 * 1 5 - 2 3 ^ ^ / +")
```

Output:

```
3: 3
4: 3, 4
2: 3, 4, 2
*: 3, 8
1: 3, 8, 1
5: 3, 8, 1, 5
-: 3, 8, -4
2: 3, 8, -4, 2
```

```
3: 3, 8, -4, 2, 3
^: 3, 8, -4, 8
^: 3, 8, 65536
/: 3, 0.0001220703125
+: 3.0001220703125
```

(The return value is also `3.0001220703125`.)

Liberty BASIC

```
global stack$

expr$ = "3 4 2 * 1 5 - 2 3 ^ ^ / +"
print "Expression:"
print expr$
print

print "Input","Operation","Stack after"

stack$=""
token$ = "#"
i = 1
token$ = word$(expr$, i)
token2$ = " "+token$+" "

do
    print "Token ";i;": ";token$,
    select case
    'operation
    case instr("+-*/^",token$)<>0
        print "operate",
        op2$=pop$()
        op1$=pop$()
        if op1$=""  then
            print "Error: stack empty for ";i;"-th token: ";token$
            end
        end if

        op1=val(op1$)
        op2=val(op2$)

        select case token$
        case "+"
            res = op1+op2
        case "-"
            res = op1-op2
        case "*"
            res = op1*op2
        case "/"
            res = op1/op2
        case "^"
            res = op1^op2
        end select
        call push str$(res)
    'default:number
    case else
        print "push",
        call push token$
    end select
    print "Stack: ";reverse$(stack$)
    i = i+1
    token$ = word$(expr$, i)
    token2$ = " "+token$+" "
loop until token$ =""

res$=pop$()
print
print "Result:" ;res$
extra$=pop$()
if extra$<>"" then
    print "Error: extra things on a stack: ";extra$
end if
end

'---------------------------------------
function reverse$(s$)
    reverse$ = ""
    token$="#"
    while token$<>""
        i=i+1
        token$=word$(s$,i,"|")
        reverse$ = token$;" ";reverse$
    wend
end function
'---------------------------------------
sub push s$
    stack$=s$+"|"+stack$    'stack
end sub

function pop$()
    'it does return empty on empty stack
    pop$=word$(stack$,1,"|")
    stack$=mid$(stack$,instr(stack$,"|")+1)
end function
```

Output:

```
Expression:
3 4 2 * 1 5 - 2 3 ^ ^ / +

Input         Operation     Stack after
Token 1: 3    push          Stack:  3
Token 2: 4    push          Stack:  3 4
Token 3: 2    push          Stack:  3 4 2
Token 4: *    operate       Stack:  3 8
Token 5: 1    push          Stack:  3 8 1
Token 6: 5    push          Stack:  3 8 1 5
Token 7: -    operate       Stack:  3 8 -4
Token 8: 2    push          Stack:  3 8 -4 2
Token 9: 3    push          Stack:  3 8 -4 2 3
Token 10: ^   operate       Stack:  3 8 -4 8
Token 11: ^   operate       Stack:  3 8 65536
Token 12: /   operate       Stack:  3 0.12207031e-3
Token 13: +   operate       Stack:  3.00012207

Result:3.00012207
```

Mathematica

(This code takes advantage of the fact that all of the operands and functions in the specified RPN syntax can be used to form valid Mathematica expressions, so we can use the built-in ToExpression function to turn them into numbers and the corresponding Mathematica functions. Note that we need to add braces around arguments, otherwise "-4^8" would be parsed as "-(4^8)" instead of "(-4)^8".)

```
calc[rpn_] :=
  Module[{tokens = StringSplit[rpn], s = "(" <> ToString@InputForm@# <> ")"
&, op, steps},
   op[o_, x_, y_] := ToExpression[s@x <> o <> s@y];
   steps = FoldList[Switch[#2, _?DigitQ, Append[#, FromDigits[#2]],
        _, Append[#[[;; -3]], op[#2, #[[-2]], #[[-1]]]]
        ] &, {}, tokens][[2 ;;]];
   Grid[Transpose[{# <> ":" & /@ tokens,
      StringRiffle[ToString[#, InputForm] & /@ #] & /@ steps}]]];
Print[calc["3 4 2 * 1 5 - 2 3 ^ ^ / +"]];
```

Output:

```
3:   3

4:   3 4

2:   3 4 2

*:   3 8

1:   3 8 1

5:   3 8 1 5

-:   3 8 -4

2:   3 8 -4 2

3:   3 8 -4 2 3

^:   3 8 -4 8

^:   3 8 65536

/:   3 1/8192

+:   24577/8192
```

NetRexx

Translation of: Java

```
/* NetRexx */
options replace format comments java crossref symbols nobinary

numeric digits 20

rpnDefaultExpression = '3 4 2 * 1 5 - 2 3 ^ ^ / +'
EODAD = '.*'

parse arg rpnString

if rpnString = '.' then rpnString = rpnDefaultExpression
if rpnString = '' then do
  say 'Enter numbers or operators [to stop enter' EODAD']:'
  loop label rpnloop forever
    rpnval = ask
    if rpnval == EODAD then leave rpnloop
    rpnString = rpnString rpnval
    end rpnloop
  end

rpnString = rpnString.space(1)
say rpnString':' evaluateRPN(rpnString)

return

-- ---------------------------------------------------------------------------
---
method evaluateRPN(rpnString) public static returns Rexx

  stack = LinkedList()
  op = 0
  L = 'L'
  R = 'R'
  rpnString = rpnString.strip('b')
  say 'Input\tOperation\tStack after'
  loop label rpn while rpnString.length > 0
    parse rpnString token rest
    rpnString = rest.strip('b')
    say token || '\t\-'
    select label tox case token
      when '*' then do
        say 'Operate\t\t\-'
        op[R] = Rexx stack.pop()
        op[L] = Rexx stack.pop()
        stack.push(op[L] * op[R])
        end
      when '/' then do
        say 'Operate\t\t\-'
        op[R] = Rexx stack.pop()
        op[L] = Rexx stack.pop()
        stack.push(op[L] / op[R])
        end
      when '+' then do
        say 'Operate\t\t\-'
        op[R] = Rexx stack.pop()
        op[L] = Rexx stack.pop()
        stack.push(op[L] + op[R])
        end
      when '-' then do
        say 'Operate\t\t\-'
        op[R] = Rexx stack.pop()
        op[L] = Rexx stack.pop()
        stack.push(op[L] - op[R])
        end
      when '^' then do
        say 'Operate\t\t\-'
        op[R] = Rexx stack.pop()
        op[L] = Rexx stack.pop()
        -- If exponent is a whole number use Rexx built-in exponentiation
operation, otherwise use Math.pow()
        op[R] = op[R] + 0
        if op[R].datatype('w') then stack.push(op[L] ** op[R])
        else stack.push(Rexx Math.pow(op[L], op[R]))
        end
      otherwise do
        if token.datatype('n') then do
          say 'Push\t\t\-'
          stack.push(token)
          end
        else do
          say 'Error\t\t\-'
          end
        end
      end tox
      calc = Rexx
      say stack.toString
    end rpn
    say
    calc = stack.toString
  return calc
```

Output:

```
Input   Operation       Stack after
3       Push            [3]
4       Push            [4, 3]
2       Push            [2, 4, 3]
*       Operate         [8, 3]
1       Push            [1, 8, 3]
5       Push            [5, 1, 8, 3]
-       Operate         [-4, 8, 3]
2       Push            [2, -4, 8, 3]
3       Push            [3, 2, -4, 8, 3]
^       Operate         [8, -4, 8, 3]
^       Operate         [65536, 8, 3]
/       Operate         [0.0001220703125, 3]
+       Operate         [3.0001220703125]

3 4 2 * 1 5 - 2 3 ^ ^ / +: [3.0001220703125]
```

Nim

Translation of: Python

```
import math, rdstdin, strutils, tables

type Stack = seq[float]

proc lalign(s, x): string =
  s & repeatChar(x - s.len, ' ')

proc opPow(s: var Stack) =
  let b = s.pop
  let a = s.pop
  s.add a.pow b

proc opMul(s: var Stack) =
  let b = s.pop
  let a = s.pop
  s.add a * b

proc opDiv(s: var Stack) =
  let b = s.pop
  let a = s.pop
  s.add a / b

proc opAdd(s: var Stack) =
  let b = s.pop
  let a = s.pop
  s.add a + b

proc opSub(s: var Stack) =
  let b = s.pop
  let a = s.pop
  s.add a - b

proc opNum(s: var Stack, num) = s.add num

let ops = toTable({"^": opPow,
                   "*": opMul,
                   "/": opDiv,
                   "+": opAdd,
                   "-": opSub})

proc getInput(inp = ""): seq[string] =
  var inp = inp
  if inp.len == 0:
    inp = readLineFromStdin "Expression: "
  result = inp.strip.split

proc rpnCalc(tokens): auto =
  var s: Stack = @[]
  result = @[@["TOKEN","ACTION","STACK"]]
  for token in tokens:
    var action = ""
    if ops.hasKey token:
      action = "Apply op to top of stack"
      ops[token](s)
    else:
      action = "Push num onto top of stack"
      s.opNum token.parseFloat
    result.add(@[token, action, s.map(proc (x: float): string = $x).join("
")])

let rpn = "3 4 2 * 1 5 - 2 3 ^ ^ / +"
echo "For RPN expression: ", rpn
let rp = rpnCalc rpn.getInput

var maxColWidths = newSeq[int](rp[0].len)
for i in 0 .. rp[0].high:
  for x in rp:
    maxColWidths[i] = max(maxColWidths[i], x[i].len)

for x in rp:
  for i, y in x:
    stdout.write y.lalign(maxColWidths[i]), " "
  echo ""
```

Output:

```
For RPN expression: 3 4 2 * 1 5 - 2 3 ^ ^ / +
TOKEN ACTION                     STACK
3     Push num onto top of stack 3.0
4     Push num onto top of stack 3.0 4.0
2     Push num onto top of stack 3.0 4.0 2.0
*     Apply op to top of stack   3.0 8.0
1     Push num onto top of stack 3.0 8.0 1.0
5     Push num onto top of stack 3.0 8.0 1.0 5.0
-     Apply op to top of stack   3.0 8.0 -4.0
2     Push num onto top of stack 3.0 8.0 -4.0 2.0
3     Push num onto top of stack 3.0 8.0 -4.0 2.0 3.0
^     Apply op to top of stack   3.0 8.0 -4.0 8.0
^     Apply op to top of stack   3.0 8.0 65536.0
/     Apply op to top of stack   3.0 0.0001220703125
+     Apply op to top of stack   3.0001220703125
```

Objeck

```
use IO;
use Struct;

bundle Default {
  class RpnCalc {
    function : Main(args : String[]) ~ Nil {
      Caculate("3 4 2 * 1 5 - 2 3 ^ ^ / +");
    }

    function : native : Caculate(rpn : String) ~ Nil {
      rpn->PrintLine();

      tokens := rpn->Split(" ");
      stack := FloatVector->New();
      each(i : tokens) {
        token := tokens[i]->Trim();
        if(token->Size() > 0) {
          if(token->Get(0)->IsDigit()) {
            stack->AddBack(token->ToFloat());
          }
          else {
            right := stack->Get(stack->Size() - 1); stack->RemoveBack();
            left := stack->Get(stack->Size() - 1); stack->RemoveBack();
            select(token->Get(0)) {
              label '+': {
                stack->AddBack(left + right);
              }

              label '-': {
                stack->AddBack(left - right);
              }

              label '*': {
                stack->AddBack(left * right);
              }

              label '/': {
                stack->AddBack(left / right);
              }

              label '^': {
                stack->AddBack(right->Power(left));
              }
            };
          };
          PrintStack(stack);
        };
      };
      Console->Print("result: ")->PrintLine(stack->Get(0));
    }

    function : PrintStack(stack : FloatVector) ~ Nil {
      "  ["->Print();
      each(i : stack) {
        stack->Get(i)->Print();
        if(i + 1< stack->Size()) {
          ", "->Print();
        };
      };
      ']'->PrintLine();
    }
  }
}
```

Output:

```
3 4 2 * 1 5 - 2 3 ^ ^ / +
  [3]
  [3, 4]
  [3, 4, 2]
  [3, 8]
  [3, 8, 1]
  [3, 8, 1, 5]
  [3, 8, -4]
  [3, 8, -4, 2]
  [3, 8, -4, 2, 3]
  [3, 8, -4, 8]
  [3, 8, 65536]
  [3, 0.00012207]
  [3.00012]
result: 3.00012
```

OCaml

```
(* binop : ('a -> 'a -> 'a) -> 'a list -> 'a list *)
let binop op = function
  | b::a::r -> (op a b)::r
  | _ -> failwith "invalid expression"

(* interp : float list -> string -> string * float list *)
let interp s = function
  | "+" -> "add",    binop ( +. ) s
  | "-" -> "subtr",  binop ( -. ) s
  | "*" -> "mult",   binop ( *. ) s
  | "/" -> "divide", binop ( /. ) s
  | "^" -> "exp",    binop ( ** ) s
  | str -> "push", (float_of_string str) :: s

(* interp_and_show : float list -> string -> float list *)
let interp_and_show s inp =
  let op,s' = interp s inp in
  Printf.printf "%s\t%s\t" inp op;
  List.(iter (Printf.printf "%F ") (rev s'));
  print_newline ();
  s'

(* rpn_eval : string -> float list *)
let rpn_eval str =
  Printf.printf "Token\tAction\tStack\n";
  let ss = Str.(split (regexp_string " ") str) in
  List.fold_left interp_and_show [] ss
```

Evaluation of the test expression:

```
# rpn_eval "3 4 2 * 1 5 - 2 3 ^ ^ / +";;
Token   Action  Stack
3       push    3.
4       push    3. 4.
2       push    3. 4. 2.
*       mult    3. 8.
1       push    3. 8. 1.
5       push    3. 8. 1. 5.
-       subtr   3. 8. -4.
2       push    3. 8. -4. 2.
3       push    3. 8. -4. 2. 3.
^       exp     3. 8. -4. 8.
^       exp     3. 8. 65536.
/       divide  3. 0.0001220703125
+       add     3.00012207031
```

```
- : float list = [3.0001220703125]
```

Oforth

Oforth uses RPN and natively parse RPN.

```
"3 4 2 * 1 5 - 2 3 ^ ^ / +" eval println
```

Output:

```
3
```

To show the changes in the stack, we can use .l after evaluating each word :

```
: rpn(s) { s words apply(#[ eval .l ]) }

rpn("3 4 2 * 1 5 - 2 3 ^ ^ / +")
```

Output:

```
3 |
3 | 4 |
3 | 4 | 2 |
3 | 8 |
3 | 8 | 1 |
3 | 8 | 1 | 5 |
3 | 8 | -4 |
3 | 8 | -4 | 2 |
3 | 8 | -4 | 2 | 3 |
3 | 8 | -4 | 8 |
3 | 8 | 65536 |
3 | 0 |
3 |
```

ooRexx

```
/* ooRexx ***************************************************************
* 10.11.2012 Walter Pachl  translated from PL/I via REXX
**********************************************************************/
fid='rpl.txt'
ex=linein(fid)
Say 'Input:' ex
/* ex=' 3 4 2 * 1 5 - 2 3 ^ ^ / +' */
Numeric Digits 15
expr=''
st=.circularqueue~new(100)
Say 'Stack contents:'
do While ex<>''
  Parse Var ex ch +1 ex
  expr=expr||ch;
  if ch<>' ' then do
    If pos(ch,'0123456789')>0 Then      /* a digit goes onto stack    */
      st~push(ch)
    Else Do                             /* an operator                */
      op=st~pull                        /* get top element            */
      select                            /* and modify the (now) top el*/
        when ch='+' Then st~push(st~pull +  op)
        when ch='-' Then st~push(st~pull -  op)
        when ch='*' Then st~push(st~pull *  op)
        when ch='/' Then st~push(st~pull /  op)
        when ch='^' Then st~push(st~pull ** op)
        end;
      Say st~string(' ','L')           /* show stack in LIFO order   */
      end
    end
  end
Say 'The reverse polish expression = 'expr
Say 'The evaluated expression = 'st~pull
```

Output:

```
Input: 3 4 2 * 1 5 - 2 3 ^ ^ / +
Stack contents:
3 8
3 8 -4
3 8 -4 8
3 8 65536
3 0.0001220703125
3.0001220703125
The reverse polish expression = 3 4 2 * 1 5 - 2 3 ^ ^ / +
The evaluated expression = 3.0001220703125
```

Perl

```
# RPN calculator
#
# Nigel Galloway April 2nd., 2012
#
$WSb = '(?:^|\s+)';
$WSa = '(?:\s+|$)';
$num = '([+-/]?(?:\.\d+|\d+(?:\.\d*)?))';
$op = '([-+*/^])';
sub myE {
  my $a = '('.$1.')'.$3.'('.$2.')';
  $a =~ s/\^/**/;
  return eval($a);
}
while (<>)  {
  while (s/$WSb$num\s+$num\s+$op$WSa/' '.myE().' '/e)  {}
  print ($_, "\n");
}
```

Produces:

```
>rpnC.pl
3 4 2 * 1 5 - 2 3 ^ ^ / +
 3.0001220703125
```

Perl 6

Works with: rakudo version 2015-09-25

```
my $proggie = '3 4 2 * 1 5 - 2 3 ^ ^ / +';

class RPN is Array {

    method binop(&op) { self.push: self.pop R[&op] self.pop }

    method run($p) {
        for $p.words {
            say "$_ ({self})";
            when /\d/ { self.push: $_ }
            when '+'  { self.binop: &[+] }
            when '-'  { self.binop: &[-] }
            when '*'  { self.binop: &[*] }
            when '/'  { self.binop: &[/] }
            when '^'  { self.binop: &[**] }
            default   { die "$_ is bogus" }
        }
        say self;
    }
}

RPN.new.run($proggie);
```

Output:

```
3 ()
4 (3)
2 (3 4)
* (3 4 2)
1 (3 8)
5 (3 8 1)
- (3 8 1 5)
2 (3 8 -4)
3 (3 8 -4 2)
^ (3 8 -4 2 3)
^ (3 8 -4 8)
/ (3 8 65536)
+ (3 0.0001220703125)
3.0001220703125
```

Phix

```
procedure evalRPN(string s)
sequence stack = {}
sequence ops = split(s)
    for i=1 to length(ops) do
        string op = ops[i]
        switch op
            case "+": stack[-2] = stack[-2]+stack[-1]; stack = stack[1..-2]
            case "-": stack[-2] = stack[-2]-stack[-1]; stack = stack[1..-2]
            case "*": stack[-2] = stack[-2]*stack[-1]; stack = stack[1..-2]
            case "/": stack[-2] = stack[-2]/stack[-1]; stack = stack[1..-2]
            case "^": stack[-2] = power(stack[-2],stack[-1]); stack =
stack[1..-2]
            default : stack = append(stack,scanf(op,"%d")[1][1])
        end switch
        ?{op,stack}
    end for
end procedure
evalRPN("3 4 2 * 1 5 - 2 3 ^ ^ / +")
```

Output:

```
"started"
{"3",{3}}
{"4",{3,4}}
{"2",{3,4,2}}
{"*",{3,8}}
{"1",{3,8,1}}
{"5",{3,8,1,5}}
{"-",{3,8,-4}}
{"2",{3,8,-4,2}}
{"3",{3,8,-4,2,3}}
{"^",{3,8,-4,8}}
{"^",{3,8,65536}}
{"/",{3,0.0001220703125}}
{"+",{3.00012207}}
```

PHP

```
<?php
function rpn($postFix){
    $stack = Array();
    echo "Input\tOperation\tStack\tafter\n" ;
        $token = explode(" ", trim($postFix));
        $count = count($token);

    for($i = 0 ; $i<$count;$i++)
        {
        echo $token[$i] ." \t";
        $tokenNum = "";

        if (is_numeric($token[$i])) {
            echo  "Push";
                       array_push($stack,$token[$i]);
        }
        else
        {
            echo "Operate";
            $secondOperand = end($stack);
                       array_pop($stack);
            $firstOperand = end($stack);
            array_pop($stack);

            if ($token[$i] == "*")
                               array_push($stack,$firstOperand *
$secondOperand);
            else if ($token[$i] == "/")
                array_push($stack,$firstOperand / $secondOperand);
            else if ($token[$i] == "-")
                array_push($stack,$firstOperand - $secondOperand);
            else if ($token[$i] == "+")
```

```
                array_push($stack,$firstOperand + $secondOperand);
            else if ($token[$i] == "^")
                array_push($stack,pow($firstOperand,$secondOperand));
            else {
                die("Error");
            }
        }
               echo "\t\t" . implode(" ", $stack) .  "\n";
    }
    return end($stack);
}

echo "Compute Value: " . rpn("3 4 2 * 1 5 - 2 3 ^ ^ / + ");
?>
```

Output:

```
Input   Operation       Stack   after
3       Push            3
4       Push            3 4
2       Push            3 4 2
*       Operate         3 8
1       Push            3 8 1
5       Push            3 8 1 5
-       Operate         3 8 -4
2       Push            3 8 -4 2
3       Push            3 8 -4 2 3
^       Operate         3 8 -4 8
^       Operate         3 8 65536
/       Operate         3 0.0001220703125
+       Operate         3.0001220703125
Compute Value: 3.0001220703125
```

PicoLisp

This is an integer-only calculator:

```
(de rpnCalculator (Str)
   (let (^ **  Stack)   # Define '^' from the built-in '**'
      (prinl "Token  Stack")
      (for Token (str Str "*+-/\^")
         (if (num? Token)
            (push 'Stack @)
            (set (cdr Stack)
               ((intern Token) (cadr Stack) (pop 'Stack)) ) )
         (prin Token)
         (space 6)
         (println Stack) )
      (println (car Stack)) ) )
```

Test (note that the top-of-stack is in the left-most position):

```
: (rpnCalculator "3 4 2 * 1 5 - 2 3 \^ \^ / +")
Token  Stack
3      (3)
```

```
4      (4 3)
2      (2 4 3)
*      (8 3)
1      (1 8 3)
5      (5 1 8 3)
-      (-4 8 3)
2      (2 -4 8 3)
3      (3 2 -4 8 3)
^      (8 -4 8 3)
^      (65536 8 3)
/      (0 3)
+      (3)
3
-> 3
```

PL/I

```
Calculator: procedure options (main);            /* 14 Sept. 2012 */
   declare expression character (100) varying initial ('');
   declare ch character (1);
   declare (stack controlled, operand) float (18);
   declare in file input;

   open file (in) title ('/CALCULAT.DAT,type(text),recsize(100)');
   on endfile (in) go to done;

   put ('Stack contents:');
main_loop:
   do forever;
      get file (in) edit (ch) (a(1));
      expression = expression || ch;
      if ch = ' ' then iterate;
      select (ch);
         when ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9')
            do; allocate stack; stack = ch; iterate main_loop; end;
         when ('+') do; operand = stack; free stack; stack = stack +
operand; end;
         when ('-') do; operand = stack; free stack; stack = stack -
operand; end;
         when ('*') do; operand = stack; free stack; stack = stack *
operand; end;
         when ('/') do; operand = stack; free stack; stack = stack /
operand; end;
         when ('^') do; operand = stack; free stack; stack = stack **
operand; end;
      end;
      call show_stack;
   end;

done:
   put skip list ('The reverse polish expression = ' || expression);
   put skip list ('The evaluated expression = ' || stack);

end Calculator;
```

```
Stack contents:
      3.0000000000      8.0000000000
      3.0000000000      8.0000000000     -4.0000000000
      3.0000000000      8.0000000000     -4.0000000000      8.0000000000
      3.0000000000      8.0000000000  65536.0000000000
      3.0000000000      0.0001220703
      3.0001220703
The reverse polish expression = 3 4 2 * 1 5 - 2 3 ^ ^ / +
The evaluated expression =  3.00012207031250000E+0000
```

The procedure to display the stack:

```
/* As the stack is push-down pop-up, need to pop it to see what's inside. */
show_stack: procedure;
   declare ts float (18) controlled;

   do while (allocation(stack) > 0);
      allocate ts; ts = stack; free stack;
   end;
   put skip;
   do while (allocation(ts) > 0);
      allocate stack; stack = ts; free ts; put edit (stack) (f(18,10));
   end;
end show_stack;
```

PowerShell

```
function Invoke-Rpn
{
  <#
    .SYNOPSIS
        A stack-based evaluator for an expression in reverse Polish notation.
    .DESCRIPTION
        A stack-based evaluator for an expression in reverse Polish notation.

        All methods in the Math and Decimal classes are available.
    .PARAMETER Expression
        A space separated, string of tokens.
    .PARAMETER DisplayState
        This switch shows the changes in the stack as each individual token
is processed as a table.
    .EXAMPLE
        Invoke-Rpn -Expression "3 4 Max"
    .EXAMPLE
        Invoke-Rpn -Expression "3 4 Log2"
    .EXAMPLE
        Invoke-Rpn -Expression "3 4 2 * 1 5 - 2 3 ^ ^ / +"
    .EXAMPLE
        Invoke-Rpn -Expression "3 4 2 * 1 5 - 2 3 ^ ^ / +" -DisplayState
#>
    [CmdletBinding()]
    Param
    (
        [Parameter(Mandatory=$true)]
        [AllowEmptyString()]
        [string]
        $Expression,

        [Parameter(Mandatory=$false)]
        [switch]
        $DisplayState
    )
    Begin
    {
        function Out-State ([System.Collections.Stack]$Stack)
        {
            $array = $Stack.ToArray()
            [Array]::Reverse($array)
            $array | ForEach-Object -Process { Write-Host ("{0,-8:F3}" -f $_)
-NoNewline } -End { Write-Host }
        }

        function New-RpnEvaluation
        {
            $stack = New-Object -Type System.Collections.Stack

            $shortcuts = @{
                "+" = "Add"; "-" = "Subtract"; "/" = "Divide"; "*" =
"Multiply"; "%" = "Remainder"; "^" = "Pow"
            }

            :ARGUMENT_LOOP foreach ($argument in $args)
            {
                if ($DisplayState -and $stack.Count)
                {
                    Out-State $stack
                }

                if ($shortcuts[$argument])
                {
                    $argument = $shortcuts[$argument]
                }

                try
                {
                    $stack.Push([decimal]$argument)
                    continue
                }
                catch
                {
                }

                $argCountList = $argument -replace "(\D+)(\d*)",‘$2’
                $operation = $argument.Substring(0, $argument.Length –
$argCountList.Length)

                foreach($type in [Decimal],[Math])
                {
                    if ($definition = $type::$operation)
                    {
                        if (-not $argCountList)
                        {
                            $argCountList = $definition.OverloadDefinitions |
                                Foreach-Object { ($_ -split ", ").Count } |
                                Sort-Object -Unique
                        }

                        foreach ($argCount in $argCountList)
                        {
                            try
                            {
                                $methodArguments =
$stack.ToArray()[($argCount-1)..0]
                                $result =
$type::$operation.Invoke($methodArguments)

                                $null = 1..$argCount | Foreach-Object {
$stack.Pop() }

                                $stack.Push($result)

                                continue ARGUMENT_LOOP
                            }
                            catch
                            {
                                ## If error, try with the next number of
arguments
                            }
                        }
                    }
                }
            }

            if ($DisplayState -and $stack.Count)
            {
                Out-State $stack
                if ($stack.Count)
                {
                    Write-Host "`nResult = $($stack.Peek())"
                }
            }
            else
            {
                $stack
            }
        }
    }
    Process
    {
        Invoke-Expression -Command "New-RpnEvaluation $Expression"
    }
    End
    {
    }
}

Invoke-Rpn -Expression "3 4 2 * 1 5 - 2 3 ^ ^ / +" -DisplayState
```

Output:

```
3.000
3.000   4.000
3.000   4.000   2.000
3.000   8.000
3.000   8.000   1.000
3.000   8.000   1.000   5.000
3.000   8.000   -4.000
3.000   8.000   -4.000  2.000
3.000   8.000   -4.000  2.000   3.000
3.000   8.000   -4.000  8.000
3.000   8.000   65536.000
3.000   0.000
3.000

Result = 3.0001220703125
```

Prolog

Works with SWI-Prolog.

```
rpn(L) :-
        writeln('Token  Action                             Stack'),
        parse(L, [],[X] ,[]),
        format('~nThe final output value is ~w~n', [X]).

% skip spaces
parse([X|L], St) -->
        {char_type(X, white)},
        parse(L, St).

% detect operators
parse([Op|L], [Y, X | St]) -->
        { is_op(Op, X, Y, V),
          writef('    %s', [[Op]]),
          with_output_to(atom(Str2), writef('Apply %s on top of stack',
[[Op]])),
          writef('  %35l', [Str2]),
          writef('%w\n', [[V | St]])},
        parse(L, [V | St]).

% detect number
parse([N|L], St) -->
        {char_type(N, digit)},
        parse_number(L, [N], St).

% string is finished
parse([], St) --> St.

% compute numbers
parse_number([N|L], NC, St) -->
        {char_type(N, digit)},
        parse_number(L, [N|NC], St).
parse_number(S, NC, St) -->
        { reverse(NC, RNC),
          number_chars(V, RNC),
          writef('%5r', [V]),
          with_output_to(atom(Str2), writef('Push num %w on top of stack', 
[V])),
          writef('  %35l', [Str2]),
          writef('%w\n', [[V | St]])},
        parse(S, [V|St]).

% defining operations
is_op(42, X, Y, V) :-  V is X*Y.
is_op(43, X, Y, V) :-  V is X+Y.
is_op(45, X, Y, V) :-  V is X-Y.
is_op(47, X, Y, V) :-  V is X/Y.
is_op(94, X, Y, V) :-  V is X**Y.
```

Output:

```
5 ?- rpn("3 4 2 * 1 5 - 2 3 ^ ^ / +").
Token  Action                             Stack
    3   'Push num 3 on top of stack'       [3]
    4   'Push num 4 on top of stack'       [4,3]
    2   'Push num 2 on top of stack'       [2,4,3]
    *   'Apply * on top of stack'          [8,3]
    1   'Push num 1 on top of stack'       [1,8,3]
    5   'Push num 5 on top of stack'       [5,1,8,3]
    -   'Apply - on top of stack'          [-4,8,3]
    2   'Push num 2 on top of stack'       [2,-4,8,3]
    3   'Push num 3 on top of stack'       [3,2,-4,8,3]
    ^   'Apply ^ on top of stack'          [8,-4,8,3]
    ^   'Apply ^ on top of stack'          [65536,8,3]
    /   'Apply / on top of stack'          [0.0001220703125,3]
    +   'Apply + on top of stack'          [3.0001220703125]

The final output value is 3.0001220703125
true .
```

Python

Version 1

```
def op_pow(stack):
    b = stack.pop(); a = stack.pop()
    stack.append( a ** b )
def op_mul(stack):
    b = stack.pop(); a = stack.pop()
    stack.append( a * b )
def op_div(stack):
    b = stack.pop(); a = stack.pop()
    stack.append( a / b )
def op_add(stack):
    b = stack.pop(); a = stack.pop()
    stack.append( a + b )
def op_sub(stack):
    b = stack.pop(); a = stack.pop()
    stack.append( a - b )
```

```
def op_num(stack, num):
    stack.append( num )

ops = {
 '^': op_pow,
 '*': op_mul,
 '/': op_div,
 '+': op_add,
 '-': op_sub,
 }

def get_input(inp = None):
    'Inputs an expression and returns list of tokens'

    if inp is None:
        inp = input('expression: ')
    tokens = inp.strip().split()
    return tokens

def rpn_calc(tokens):
    stack = []
    table = ['TOKEN,ACTION,STACK'.split(',')]
    for token in tokens:
        if token in ops:
            action = 'Apply op to top of stack'
            ops[token](stack)
            table.append( (token, action, ' '.join(str(s) for s in stack)) )
        else:
            action = 'Push num onto top of stack'
            op_num(stack, eval(token))
            table.append( (token, action, ' '.join(str(s) for s in stack)) )
    return table

if __name__ == '__main__':
    rpn = '3 4 2 * 1 5 - 2 3 ^ ^ / +'
    print( 'For RPN expression: %r\n' % rpn )
    rp = rpn_calc(get_input(rpn))
    maxcolwidths = [max(len(y) for y in x) for x in zip(*rp)]
    row = rp[0]
    print( ' '.join('{cell:^{width}}'.format(width=width, cell=cell) for
(width, cell) in zip(maxcolwidths, row)))
    for row in rp[1:]:
        print( ' '.join('{cell:<{width}}'.format(width=width, cell=cell) for
(width, cell) in zip(maxcolwidths, row)))

    print('\n The final output value is: %r' % rp[-1][2])
```

Output:

```
For RPN expression: '3 4 2 * 1 5 - 2 3 ^ ^ / +'

TOKEN           ACTION                 STACK
3     Push num onto top of stack 3
4     Push num onto top of stack 3 4
2     Push num onto top of stack 3 4 2
*     Apply op to top of stack   3 8
1     Push num onto top of stack 3 8 1
5     Push num onto top of stack 3 8 1 5
```

```
-      Apply op to top of stack   3 8 -4
2      Push num onto top of stack 3 8 -4 2
3      Push num onto top of stack 3 8 -4 2 3
^      Apply op to top of stack   3 8 -4 8
^      Apply op to top of stack   3 8 65536
/      Apply op to top of stack   3 0.0001220703125
+      Apply op to top of stack   3.0001220703125

 The final output value is: '3.0001220703125'
```

Version 2

```
a=[]
b={'+': lambda x,y: y+x, '-': lambda x,y: y-x, '*': lambda x,y: y*x,'/':
lambda x,y:y/x,'^': lambda x,y:y**x}
for c in '3 4 2 * 1 5 - 2 3 ^ ^ / +'.split():
    if c in b: a.append(b[c](a.pop(),a.pop()))
    else: a.append(float(c))
    print c, a
```

Output:

```
3 [3.0]
4 [3.0, 4.0]
2 [3.0, 4.0, 2.0]
* [3.0, 8.0]
1 [3.0, 8.0, 1.0]
5 [3.0, 8.0, 1.0, 5.0]
- [3.0, 8.0, -4.0]
2 [3.0, 8.0, -4.0, 2.0]
3 [3.0, 8.0, -4.0, 2.0, 3.0]
^ [3.0, 8.0, -4.0, 8.0]
^ [3.0, 8.0, 65536.0]
/ [3.0, 0.0001220703125]
+ [3.0001220703125]
```

Racket

```
#lang racket

(define (calculate-RPN expr)
  (for/fold ([stack '()]) ([token expr])
    (printf "~a\t -> ~a~N" token stack)
    (match* (token stack)
     [((? number? n) s) (cons n s)]
     [('+ (list x y s ___)) (cons (+ x y) s)]
     [('- (list x y s ___)) (cons (- y x) s)]
     [('* (list x y s ___)) (cons (* x y) s)]
     [('/ (list x y s ___)) (cons (/ y x) s)]
     [('^ (list x y s ___)) (cons (expt y x) s)]
     [(x s) (error "calculate-RPN: Cannot calculate the expression:"
                   (reverse (cons x s)))])))
```

Test case

```
-> (calculate-RPN '(3.0 4 2 * 1 5 - 2 3 ^ ^ / +))
3.0      -> ()
4        -> (3.0)
2        -> (4 3.0)
*        -> (2 4 3.0)
1        -> (8 3.0)
5        -> (1 8 3.0)
-        -> (5 1 8 3.0)
2        -> (-4 8 3.0)
3        -> (2 -4 8 3.0)
^        -> (3 2 -4 8 3.0)
^        -> (8 -4 8 3.0)
/        -> (65536 8 3.0)
+        -> (1/8192 3.0)
3.0001220703125
```

Reading from a string:

```
(calculate-RPN (in-port read (open-input-string "3.0 4 2 * 1 5 - 2 3 ^ ^ /
+")))
```

REXX

version 1

```
/* REXX ***************************************************************
* 09.11.2012 Walter Pachl  translates from PL/I
**********************************************************************/
fid='rpl.txt'
ex=linein(fid)
Say 'Input:' ex
/* ex=' 3 4 2 * 1 5 - 2 3 ^ ^ / +' */
Numeric Digits 15
expr=''
st.=0
Say 'Stack contents:'
do While ex<>''
  Parse Var ex ch +1 ex
  expr=expr||ch;
  if ch<>' ' then do
    select
      When pos(ch,'0123456789')>0 Then Do
        Call stack ch
        Iterate
        End
      when ch='+' Then do; operand=getstack(); st.sti = st.sti +  operand;
end;
      when ch='-' Then do; operand=getstack(); st.sti = st.sti -  operand;
end;
      when ch='*' Then do; operand=getstack(); st.sti = st.sti *  operand;
```

```
end;
      when ch='/' Then do; operand=getstack(); st.sti = st.sti /  operand;
end;
      when ch='^' Then do; operand=getstack(); st.sti = st.sti ** operand;
end;
      end;
    call show_stack
    end
  end
Say 'The reverse polish expression = 'expr
Say 'The evaluated expression = 'st.1
Exit
stack: Procedure Expose st.
/* put the argument on top of the stack */
  z=st.0+1
  st.z=arg(1)
  st.0=z
  Return
getstack: Procedure Expose st. sti
/* remove and return the stack's top element */
  z=st.0
  stk=st.z
  st.0=st.0-1
  sti=st.0
  Return stk
show_stack: procedure Expose st.
/* show the stack's contents */
  ol=''
  do i=1 To st.0
    ol=ol format(st.i,5,10)
    End
  Say ol
  Return
```

Output:

```
Input: 3 4 2 * 1 5 - 2 3 ^ ^ / +
Stack contents:
     3.0000000000     8.0000000000
     3.0000000000     8.0000000000    -4.0000000000
     3.0000000000     8.0000000000    -4.0000000000     8.0000000000
     3.0000000000     8.0000000000 65536.0000000000
     3.0000000000     0.0001220703
     3.0001220703
The reverse polish expression = 3 4 2 * 1 5 - 2 3 ^ ^ / +
The evaluated expression = 3.0001220703125
```

version 2

This REXX version handles tokens (not characters)   so that the RPN could be   (for instance):

3.0   .4e1   2e0   *   +1.   5   -   2   3   **   **   /   +

which is the essentially the same as the default used by the REXX program.

```
/*REXX program  evaluates  a   ═════ Reverse Polish notation  (RPN) ═════
expression. */
parse arg x                                       /*obtain optional arguments
from the CL*/
if x=''  then x= "3 4 2 * 1 5 - 2 3 ^ ^ / +"     /*Not specified?  Then use
the default.*/
tokens=words(x)                                   /*save the  number  of
tokens   "  ". */
showSteps=1                                       /*set to 0 if working steps
not wanted.*/
ox=x                                              /*save the  original  value
of  X.     */
            do i=1  for tokens;   @.i=word(x,i)  /*assign the input tokens to
an array. */
            end   /*i*/
x=space(x)                                        /*remove any superfluous
blanks in  X. */
L=max(20, length(x))                              /*use 20 for the minimum
display width.*/
numeric digits L                                  /*ensure enough decimal
digits for ans.*/
say center('operand', L, "─")        center('stack', L+L, "─")
/*display title*/
$=                                                /*nullify the stack
(completely empty).*/
       do k=1  for tokens;   ?=@.k;   ??=?       /*process each token from
the  @. list.*/
       #=words($)                                 /*stack the count (the
number entries).*/
       if datatype(?,'N')  then do;  $=$ ?;   call show  "add to───►stack";
iterate;  end
       if ?=='^'           then ??= "**"         /*REXXify    ^ ───► **
(make legal).*/
       interpret 'y='word($,#-1)  ??  word($,#)  /*compute via the famous
REXX INTERPRET*/
       if datatype(y,'N')  then y=y/1            /*normalize the number with
÷ by unity.*/
       $=subword($, 1, #-2)     y                /*rebuild the stack with the
answer.   */
       call show ?                                /*display steps (tracing
into),  maybe.*/
       end   /*k*/
say                                               /*display a blank line,
better perusing*/
say ' RPN input:'  ox;   say "  answer───►"$      /*display original input;
display ans.*/
parse source upper . y .                          /*invoked via  C.L.  or via
a REXX pgm?*/
if y=='COMMAND' | \datatype($,"W")  then exit    /*stick a fork in it,  we're
all done. */
                                     else exit $  /*return the answer  ───►
the invoker.*/
/*──────────────────────────────────────────────────────────────────────────────
───────────*/
show: if showSteps  then say center(arg(1), L)            left(space($), L);
return
```

output when using the default input:

```
─────────operand───────── ──────────────────────stack───────────────────────
     add to───►stack      3
     add to───►stack      3 4
     add to───►stack      3 4 2
            *             3 8
     add to───►stack      3 8 1
     add to───►stack      3 8 1 5
            -             3 8 -4
     add to───►stack      3 8 -4 2
     add to───►stack      3 8 -4 2 3
            ^             3 8 -4 8
            ^             3 8 65536
            /             3 0.0001220703125
            +             3.0001220703125

 RPN input: 3 4 2 * 1 5 - 2 3 ^ ^ / +
 answer───► 3.0001220703125
```

version 3 (error checking)

This REXX version is the same as above, but also checks for various errors and allows more operators:

- checks for illegal operator
- checks for illegal number
- checks for illegal bit (logical) values
- checks for malformed RPN expression
- checks for division by zero
- allows alternative exponentiation symbol **
- allows logical operations & && |
- allows alternative division symbol ÷
- allows integer division %
- allows remainder division //
- allows concatenation ||

```
/*REXX program  evaluates  a   ═════ Reverse Polish notation  (RPN) ═════
expression. */
parse arg x                                      /*obtain optional arguments
from the CL*/
if x=''  then x= "3 4 2 * 1 5 - 2 3 ^ ^ / +"     /*Not specified?  Then use
the default.*/
tokens=words(x)                                  /*save the  number  of
tokens   "  ". */
showSteps=1                                      /*set to 0 if working steps
not wanted.*/
ox=x                                             /*save the  original  value
of  X.     */
            do i=1  for tokens;   @.i=word(x,i)  /*assign the input tokens to
an array. */
```

```
            end   /*i*/
x=space(x)                                       /*remove any superfluous
blanks in  X. */
L=max(20, length(x))                             /*use 20 for the minimum
display width.*/
numeric digits L                                 /*ensure enough decimal
digits for ans.*/
say center('operand', L, "—")        center('stack', L+L, "—")
/*display title*/
Dop= '/ // % ÷';             Bop='& | &&'        /*division operators;
binary operands.*/
Aop= '- + * ^ **'  Dop Bop;  Lop=Aop "||"        /*arithmetic operators;
legal operands.*/
$=                                               /*nullify the stack
(completely empty).*/
       do k=1  for tokens;   ?=@.k;   ??=?       /*process each token from
the  @. list.*/
       #=words($);  b=word($, max(1, #) )        /*the stack count;  the last
entry.    */
                    a=word($, max(1, #-1) )      /*stack's  "first"  operand.
*/
       division  =wordpos(?, Dop)\==0            /*flag:  doing a some kind
of division.*/
       arith     =wordpos(?, Aop)\==0            /*flag:  doing arithmetic.
*/
       bitOp     =wordpos(?, Bop)\==0            /*flag:  doing some kind of
binary oper*/
       if datatype(?, 'N')   then do; $=$ ?;  call show  "add to───►stack";
iterate;  end
       if wordpos(?, Lop)==0 then do; $=e 'illegal operator:' ?;      leave;
end
       if w<2                then do; $=e 'illegal RPN expression.';  leave;
end
       if ?=='^'             then ??= "**"       /*REXXify  ^ ──► **   (make
it legal). */
       if ?=='÷'             then ??= "/"        /*REXXify  ÷ ──► /    (make
it legal). */
       if division  &  b=0   then do; $=e 'division by zero.'      ;  leave;
end
       if bitOp & \isBit(a)  then do; $=e "token isn't logical: " a;  leave;
end
       if bitOp & \isBit(b)  then do; $=e "token isn't logical: " b;  leave;
end
       interpret 'y='   a   ??   b               /*compute with two stack
operands*/
       if datatype(y, 'W')   then y=y/1          /*normalize the number with
÷ by unity.*/
       _=subword($, 1, #-2);      $=_ y          /*rebuild the stack with the
answer.   */
       call show ?                               /*display (possibly)  a
working step.  */
       end   /*k*/
say                                              /*display a blank line,
better perusing*/
if word($,1)==e  then $=                         /*handle the special case of
errors.   */
say ' RPN input:'  ox;   say " answer───►"$      /*display original input;
```

```
display ans.*/
parse source upper . y .                        /*invoked via  C.L.  or via
a REXX pgm?*/
if y=='COMMAND' | \datatype($,"W")  then exit    /*stick a fork in it,  we're
all done. */
                                    else exit $  /*return the answer  ───►
the invoker.*/
/*────────────────────────────────────────────────────────────────────────────
───────────*/
isBit: return arg(1)==0 | arg(1)==1             /*returns   1   if arg1 is a
binary bit*/
show:  if showSteps  then say center(arg(1), L)          left(space($), L);
return
```

output is identical to the 2nd REXX version.

Ruby

See Parsing/RPN/Ruby

```
rpn = RPNExpression("3 4 2 * 1 5 - 2 3 ^ ^ / +")
value = rpn.eval
```

Output:

```
for RPN expression: 3 4 2 * 1 5 - 2 3 ^ ^ / +
Term    Action  Stack
3       PUSH    [3]
4       PUSH    [3, 4]
2       PUSH    [3, 4, 2]
*       MUL     [3, 8]
1       PUSH    [3, 8, 1]
5       PUSH    [3, 8, 1, 5]
-       SUB     [3, 8, -4]
2       PUSH    [3, 8, -4, 2]
3       PUSH    [3, 8, -4, 2, 3]
^       EXP     [3, 8, -4, 8]
^       EXP     [3, 8, 65536]
/       DIV     [3, 0.0001220703125]
+       ADD     [3.0001220703125]
Value = 3.0001220703125
```

Run BASIC

```
prn$ = "3 4 2 * 1 5 - 2 3 ^ ^ / + "

j = 0
while word$(prn$,i + 1," ") <> ""
i = i + 1
  n$ = word$(prn$,i," ")
  if n$ < "0" or n$ > "9" then
    num1    = val(word$(stack$,s," "))
    num2    = val(word$(stack$,s-1," "))
    n       = op(n$,num2,num1)
    s       = s - 1
    stack$ = stk$(stack$,s -1,str$(n))
   print "Push Opr ";n$;" to stack:  ";stack$
 else
  s = s + 1
  stack$ = stack$ + n$ + " "
  print "Push Num ";n$;" to stack:  ";stack$
end if
wend

function stk$(stack$,s,a$)
for i = 1 to s
  stk$ = stk$ + word$(stack$,i," ") + " "
next i
stk$ = stk$ + a$ + " "
end function

FUNCTION op(op$,a,b)
if op$ = "*" then op = a * b
if op$ = "/" then op = a / b
if op$ = "^" then op = a ^ b
if op$ = "+" then op = a + b
if op$ = "-" then op = a - b
end function
Push Num 3 to stack:  3
Push Num 4 to stack:  3 4
Push Num 2 to stack:  3 4 2
Push Opr * to stack:  3 8
Push Num 1 to stack:  3 8 1
Push Num 5 to stack:  3 8 1 5
Push Opr - to stack:  3 8 -4
Push Num 2 to stack:  3 8 -4 2
Push Num 3 to stack:  3 8 -4 2 3
Push Opr ^ to stack:  3 8 -4 8
Push Opr ^ to stack:  3 8 65536
Push Opr / to stack:  3 1.22070312e-4
Push Opr + to stack:  3.00012207
```

Scala

```
object RPN {
  val PRINT_STACK_CONTENTS: Boolean = true

  def main(args: Array[String]): Unit = {
    val result = evaluate("3 4 2 * 1 5 - 2 3 ^ ^ / +".split(" ").toList)
    println("Answer: " + result)
  }

  def evaluate(tokens: List[String]): Double = {
    import scala.collection.mutable.Stack
    val stack: Stack[Double] = new Stack[Double]
    for (token <- tokens) {
      if (isOperator(token)) token match {
        case "+" => stack.push(stack.pop + stack.pop)
        case "-" => val x = stack.pop; stack.push(stack.pop - x)
        case "*" => stack.push(stack.pop * stack.pop)
        case "/" => val x = stack.pop; stack.push(stack.pop / x)
        case "^" => val x = stack.pop; stack.push(math.pow(stack.pop, x))
        case _ => throw new RuntimeException( s""""$token" is not an
operator""")
      }
      else stack.push(token.toDouble)

      if (PRINT_STACK_CONTENTS) {
        print("Input: " + token)
        print(" Stack: ")
        for (element <- stack.seq.reverse) print(element + " ");
        println("")
      }
    }

    stack.pop
  }

  def isOperator(token: String): Boolean = {
    token match {
      case "+" => true; case "-" => true; case "*" => true; case "/" => true;
case "^" => true
      case _ => false
    }
  }
}
```

Output:

```
Input: 3 Stack: 3.0
Input: 4 Stack: 3.0 4.0
Input: 2 Stack: 3.0 4.0 2.0
Input: * Stack: 3.0 8.0
Input: 1 Stack: 3.0 8.0 1.0
Input: 5 Stack: 3.0 8.0 1.0 5.0
Input: - Stack: 3.0 8.0 -4.0
Input: 2 Stack: 3.0 8.0 -4.0 2.0
Input: 3 Stack: 3.0 8.0 -4.0 2.0 3.0
Input: ^ Stack: 3.0 8.0 -4.0 8.0
Input: ^ Stack: 3.0 8.0 65536.0
Input: / Stack: 3.0 1.220703125E-4
Input: + Stack: 3.0001220703125
Answer: 3.0001220703125
```

Sidef

Translation of: Perl 6

```
var proggie = '3 4 2 * 1 5 - 2 3 ^ ^ / +';

class RPN(arr=[]) {

    method binop(op) {
        var x = arr.pop
        var y = arr.pop
        arr << y.(op)(x)
    }
    method run(p) {
        p.each_word { |w|
            say "#{w} (#{arr})";
            given (w) {
                when (/\d/) {
                    arr << w.to_f
                }
                when (<+ - * />) {
                    self.binop(w)
                }
                when ('^') {
                    self.binop('**')
                }
                default {
                    die "#{w} is bogus"
                }
            }
        }
        say arr[0]
    }
}

RPN.new.run(proggie);
```

Output:

```
3 ()
4 (3)
2 (3 4)
* (3 4 2)
1 (3 8)
5 (3 8 1)
- (3 8 1 5)
2 (3 8 -4)
3 (3 8 -4 2)
^ (3 8 -4 2 3)
^ (3 8 -4 8)
/ (3 8 65536)
+ (3 0.0001220703125)
3.0001220703125
```

Swift

Translation of: Go

```
let opa = [
    "^": (prec: 4, rAssoc: true),
    "*": (prec: 3, rAssoc: false),
    "/": (prec: 3, rAssoc: false),
    "+": (prec: 2, rAssoc: false),
    "-": (prec: 2, rAssoc: false),
]

func rpn(tokens: [String]) -> [String] {
    var rpn : [String] = []
    var stack : [String] = [] // holds operators and left parenthesis

    for tok in tokens {
        switch tok {
        case "(":
            stack += [tok] // push "(" to stack
        case ")":
            while !stack.isEmpty {
                let op = stack.removeLast() // pop item from stack
                if op == "(" {
                    break // discard "("
                } else {
                    rpn += [op] // add operator to result
                }
            }
        default:
            if let o1 = opa[tok] { // token is an operator?
                for op in stack.reverse() {
                    if let o2 = opa[op] {
                        if !(o1.prec > o2.prec || (o1.prec == o2.prec &&
o1.rAssoc)) {
                            // top item is an operator that needs to come off
                            rpn += [stack.removeLast()] // pop and add it to
the result
                            continue
                        }
                    }
                    break
                }

                stack += [tok] // push operator (the new one) to stack
            } else { // token is not an operator
                rpn += [tok] // add operand to result
            }
        }
    }

    return rpn + stack.reverse()
}

func parseInfix(e: String) -> String {
    let tokens = e.characters.split{ $0 == " " }.map(String.init)
    return rpn(tokens).joinWithSeparator(" ")
}

var input : String

input = "3 + 4 * 2 / ( 1 - 5 ) ^ 2 ^ 3"
"infix: \(input)"
"postfix: \(parseInfix(input))"
```

Output:

```
"postfix: 3 4 2 * 1 5 - 2 3 ^ ^ / +"
```

Tcl

```
# Helper
proc pop stk {
    upvar 1 $stk s
    set val [lindex $s end]
    set s [lreplace $s end end]
    return $val
}

proc evaluate rpn {
    set stack {}
    foreach token $rpn {
        set act "apply"
        switch $token {
            "^" {
                # Non-commutative operation
                set a [pop stack]
                lappend stack [expr {[pop stack] ** $a}]
            }
            "/" {
                # Non-commutative, special float handling
                set a [pop stack]
                set b [expr {[pop stack] / double($a)}]
                if {$b == round($b)} {set b [expr {round($b)}]}
                lappend stack $b
            }
            "*" {
                # Commutative operation
                lappend stack [expr {[pop stack] * [pop stack]}]
            }
            "-" {
                # Non-commutative operation
                set a [pop stack]
                lappend stack [expr {[pop stack] - $a}]
            }
            "+" {
                # Commutative operation
                lappend stack [expr {[pop stack] + [pop stack]}]
            }
            default {
                set act "push"
                lappend stack $token
            }
        }
        puts "$token\t$act\t$stack"
    }
    return [lindex $stack end]
}

puts [evaluate {3 4 2 * 1 5 - 2 3 ^ ^ / +}]
```

Output:

```
3       push    3
4       push    3 4
2       push    3 4 2
*       apply   3 8
1       push    3 8 1
5       push    3 8 1 5
```

```
-       apply   3 8 -4
2       push    3 8 -4 2
3       push    3 8 -4 2 3
^       apply   3 8 -4 8
^       apply   3 8 65536
/       apply   3 0.0001220703125
+       apply   3.0001220703125
3.0001220703125
```

VBA

Translation of: Liberty BASIC

```
Global stack$

Function RPN(expr$)
Debug.Print "Expression:"
Debug.Print expr$
Debug.Print "Input", "Operation", "Stack after"

stack$ = ""
token$ = "#"
i = 1
token$ = Split(expr$)(i - 1) 'split is base 0
token2$ = " " + token$ + " "

Do
    Debug.Print "Token "; i; ": "; token$,
    'operation
    If InStr("+-*/^", token$) <> 0 Then
        Debug.Print "operate",
        op2$ = pop$()
        op1$ = pop$()
        If op1$ = "" Then
            Debug.Print "Error: stack empty for "; i; "-th token: "; token$
            End
        End If

        op1 = Val(op1$)
        op2 = Val(op2$)

        Select Case token$
        Case "+"
            res = CDbl(op1) + CDbl(op2)
        Case "-"
            res = CDbl(op1) - CDbl(op2)
        Case "*"
            res = CDbl(op1) * CDbl(op2)
        Case "/"
            res = CDbl(op1) / CDbl(op2)
        Case "^"
            res = CDbl(op1) ^ CDbl(op2)
        End Select

        Call push2(str$(res))
    'default:number
    Else
        Debug.Print "push",
        Call push2(token$)
    End If
    Debug.Print "Stack: "; reverse$(stack$)
    i = i + 1
    If i > Len(Join(Split(expr, " "), "")) Then
        token$ = ""
    Else
        token$ = Split(expr$)(i - 1) 'base 0
        token2$ = " " + token$ + " "
    End If
Loop Until token$ = ""

Debug.Print
Debug.Print "Result:"; pop$()
'extra$ = pop$()
If stack <> "" Then
    Debug.Print "Error: extra things on a stack: "; stack$
End If
End
End Function

'----------------------------------------
Function reverse$(s$)
    reverse$ = ""
    token$ = "#"
    While token$ <> ""
        i = i + 1
        token$ = Split(s$, "|")(i - 1) 'split is base 0
        reverse$ = token$ & " " & reverse$
    Wend
End Function
'----------------------------------------
Sub push2(s$)
    stack$ = s$ + "|" + stack$ 'stack
End Sub

Function pop$()
    'it does return empty on empty stack
    pop$ = Split(stack$, "|")(0)
    stack$ = Mid$(stack$, InStr(stack$, "|") + 1)
End Function
```

Output:

```
?RPN("3 4 2 * 1 5 - 2 3 ^ ^ / +")
Expression:
3 4 2 * 1 5 - 2 3 ^ ^ / +
Input         Operation     Stack after
Token  1 : 3  push          Stack:  3
Token  2 : 4  push          Stack:  3 4
Token  3 : 2  push          Stack:  3 4 2
Token  4 : *  operate       Stack:  3  8
Token  5 : 1  push          Stack:  3  8 1
Token  6 : 5  push          Stack:  3  8 1 5
Token  7 : -  operate       Stack:  3  8 -4
Token  8 : 2  push          Stack:  3  8 -4 2
Token  9 : 3  push          Stack:  3  8 -4 2 3
Token  10 : ^ operate       Stack:  3  8 -4  8
Token  11 : ^ operate       Stack:  3  8  65536
Token  12 : / operate       Stack:  3  .0001220703125
Token  13 : + operate       Stack:   3.0001220703125

Result: 3.0001220703125
```

zkl

```
var ops=D("^",True,  "*",'*,  "/",'/,  "+",'+,  "-",'-);

fcn parseRPN(e){
   println("\npostfix: ", e);
   stack:=L();
   foreach tok in (e.split()){
      op:=ops.find(tok);
      if(op){
         y := stack.pop(); x := stack.pop();
         if(True==op) x=x.pow(y);
         else         x=op(x,y);
         stack.append(x);
      }
      else stack.append(tok.toFloat());
      println(tok," --> ",stack);
   }
   println("result: ", stack[0])
}
tests:=T("3 4 2 * 1 5 - 2 3 ^ ^ / +");
foreach t in (tests) { parseRPN(t) }
```

Output:

```
postfix: 3 4 2 * 1 5 - 2 3 ^ ^ / +
3 --> L(3)
4 --> L(3,4)
2 --> L(3,4,2)
* --> L(3,8)
1 --> L(3,8,1)
5 --> L(3,8,1,5)
- --> L(3,8,-4)
2 --> L(3,8,-4,2)
3 --> L(3,8,-4,2,3)
^ --> L(3,8,-4,8)
^ --> L(3,8,65536)
/ --> L(3,0.00012207)
+ --> L(3.00012)
result: 3.00012
```

Categories:

- Programming Tasks
- Solutions by Programming Task
- Ada
- ALGOL 68
- ANTLR

- AutoHotkey
- BBC BASIC
- C
- C++
- C sharp
- Ceylon
- Clojure
- Common Lisp
- EchoLisp
- Ela
- D
- Erlang
- F Sharp
- Fortran
- FunL
- Go
- Groovy
- Haskell
- Icon
- Unicon
- Icon Programming Library
- J
- Java
- JavaScript
- Julia
- Liberty BASIC
- Mathematica
- NetRexx
- Nim
- Objeck
- OCaml
- Oforth
- OoRexx
- Perl
- Perl 6
- Phix
- PHP
- PicoLisp
- PL/I
- PowerShell
- Prolog
- Python
- Racket
- REXX
- Ruby
- Run BASIC

- Scala
- Sidef
- Swift
- Tcl
- VBA
- Zkl

- What links here
- Related changes
- Special pages
- Printable version
- Permanent link
- Page information
- Browse properties

- This page was last modified on 25 August 2016, at 03:05.

- Privacy policy
- About Rosetta Code
- Disclaimers