Design and Implementation of an Array Language

Bernd Ulmann

V. 1.0, 26-MAY-2013

To my beloved wife Rikka.

Acknowledgments

This book would not have been possible without the support and help of many people. First of all, I would like to thank my wife Rikka Mitsam who never complained about the many hours I spent writing instead of being with her and did a lot of proofreading.

I am also greatly indebted to Thomas Kratz who did most of the implementation of the Lang5-interpreter and the `Array::DeepUtils` module.

I am particularly grateful for the support and help of Jens Breitenbach and Hans Franke show did a magnificent job at proof reading and made many valuable suggestions which greatly enhanced this booklet.

In addition to that, I would like to thank Patrick Hedfeld for countless fruitful discussions about languages in general and programming languages and Lang5 in special. He also did a terrific job with writing the Lang5-Redbook. In addition to that he spotted many flaws and faults in this booklet which were corrected accordingly.

Contents

1 Introduction

1.1 Preliminaries

There is a plethora of introductory texts for various programming languages filling countless shelves but there are only a few texts covering the basic aspects of the design and implementation of a programming language. One reason for this is that most programming languages have grown to be far too complex to be usable for teaching their inner workings. Therefore this book first introduces a rather simple stack based *array language* called Lang5, which has been developed explicitly to serve as an example for introductory lectures about programming languages and interpreter design.

All of the source code used and described in the following is available for free at `http://lang5.sourceforge.net` and should be downloaded and installed on a LINUX, Mac OS X, Windows or even OpenVMS based computer since the remainder of this book makes heavy use of the interpreter and solving the exercises requires a Lang5-interpreter at hand.

It is further assumed that the reader already has some programming experience, ideally with the C and Perl, and knows about nested data structures built from arrays and hashes since this is not an introductory text about programming as such.

User input is always denoted by red color in the following examples to distinguish it clearly from output generated by the interpreter, the operating system etc.

1.2 Array languages

First of all, what is a so-called *array language*? Most languages, some of which even became part of our everyday vocabulary, like C, its derivatives C++ and C#, Java, Perl, Python etc. are either *imperative*[1] or *object oriented* languages as the "TIOBE Programming Community Index for April 2013", shows: The ten most used programming languages are either imperative (C being the most widely used language) or object oriented languages.[2]

Languages employing other programming paradigms like so-called *functional* or *array languages* are, unfortunately, used only rarely according to TIOBE. LISP, for example, is on position 13 being used in 0.095% of the projects taken into account while APL,

[1] Also called von Neumann *languages*.

[2] In April 2013, the top ten programming languages listed by TIOBE (see `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`) were C, Java, C++, Objective-C, C#, PHP, (Visual) Basic and Python.

the archetypal array programming language, has rank 39, being used in 0.222% of the projects.

This is a pity since both paradigms offer methods and chances for programming that exceed those of the more traditional languages taking up the first ranks. Especially array programming languages are highly powerful and yet extremely underestimated.

The idea of array languages is an old one: In the late 1950s, Ken Iverson, a mathematician who was not satisfied with the traditional form of mathematical notation, set out to develop a new notation while being an assistant professor in Harvard.[3] In 1960 Ken Iverson began working for IBM where he met Adin Falkoff who became interested in this new notation. He extended his notation to a degree which made it possible to be used for the description of algorithms and computer systems in general.[4]

Internally, this language was known as *Iverson's Better Math* but IBM did not like that name for obvious reasons, thus a new name was needed. In the following this seminal language became known as APL, short for *A Programming Language*. This name was used officially for the first time in the title of [Iverson 1962] where it was still used as a method for *"interpersonal communication"*.[5]

The first noteworthy aspect of APL is its unusual character set which was (and often still is) considered a major obstacle as the following question asked by R. A. Brooker shows:[6]

> *"Why do you insist on using a notation which is a nightmare for typist and compositor?"*

This question was even more important back in the early 1960s since there were no graphic display capable of displaying the special APL characters available. IBM overcame this problem with the introduction of the so-called *Selectric Typewriter* which could use special character balls containing the characters required for the APL system. Despite this particular nuisance, the extreme consistency and efficiency of APL quickly led to the development of interpreters for this language.[7]

APL encourages a rather unique style of programming. On a first glance APL programs are quite unreadable for the uninitiated but as soon as one gets used to its special character set and its basic ideas, it turns out to be one of the most powerful programming languages ever. Especially its array features which make APL an *array language* lead to unusual and often astonishingly short solutions compared with other programming languages.[8]

[3] See [Janko 1980].

[4] The most famous example for a computer system's description in this notation is that of the IBM /360 architecture in 1964 which is described in [Falkoff et al. 1964].

[5] See [Janko 1980][p. 1].

[6] See [McDonnel 1981][p. 11].

[7] Hellermann implemented an interpreter supporting a subset of APL on the IBM 1620 in 1963; an APL system running in batch mode was developed for the IBM 7090 mainframe in 1965 by M. L. Breed and P. S. Abrams who also implemented an APL system running under an experimental time sharing system on an IBM 7090 (see [Janko 1980][p. 1]).

[8] This is, by the way, a good example of how notation and languages shape the way of problem description and solving. Ken Iverson's famous publication [Iverson 1963] is a must read in this context.

APL in its very nature is an interpreter language with an unusually high degree of interactivity. It features dynamic typing, a feature also found in today's *dynamic languages* like Perl, Python etc., but its main feature is its use of the vector as its basic data structure. Because of this, APL programs seldomly use (explicit) loops, conditional execution and the like since wherever possible vector and matrix operations are employed.

The following example gives an impression of the power resulting from using vectors as basic data structures: The sum $\sum_{i=1}^{100} i$ is to be computed. Using a typical imperative language like C this could be accomplished as follows:

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int sum, i;
6
7      for (i = sum = 0; i <= 100; sum += i++);
8
9      printf("%d\n", sum);
10     return 0;
11 }
```

As straightforward as such a solution may seem, only a small portion of this program (a single line, in fact) deals with the actual problem of computing the sum of all natural numbers ≤ 100. The remaining lines deal with variable declaration, printing the result and terminating the program graciously. An APL solution of the same problem might look like this: +/⍳100

APL programs are read from right to left, so this program applies the ⍳ (*iota*) function to the scalar 100 which yields a vector with unit stride containing 100 elements. The next part is tricky: Using the reduction operator / the dyadic operator + is applied between each two successive elements of this vector thus yielding 1+2+3+...+99+100. This is typical idiom of an array language which saves many of the explicit loops required by other programming languages.

The next (and last) APL example[9] is more complicated but shows even more of the power of array languages. A list of prime numbers between 2 and 100 is to generated. A first, very simple, straightforward solution in C might look like this:

```
1  #include <stdio.h>
2
3  #define END 100
4
```

[9] Much more detailed information about APL can be found in [Iverson 1962], [Gilman et al. 1970], [Iverson 1963] etc.

```
int is_prime(int value)
{
    int divisor;

    if (!(value % 2))
        return value == 2;

    for (divisor = 3; divisor * divisor <= value; divisor += 2)
        if (!(value % divisor))
            return 0;

    return 1;
}

int main()
{
    int i;

    for (i = 2; i <= END; i++)
        if (is_prime(i))
            printf("%d ", i);

    printf("\n");
    return 0;
}
```

A function `is_prime()` is defined which tests be successive division if a number is prime. Using this function within a loop a list of prime numbers is printed. As simple as this approach is, a substantial amount of code is still not devoted to the problem itself but is owed to the language. A more elegant solution in C could implement the *sieve of* Eratosthenes which is based on a unit stride vector starting with the value 2. In the first run, the leftmost vector element must be prime, so all of its multiples can be marked as non-prime in the vector until the last element of the vector has been reached. The next loop will then determine the next element from the left which has not been marked as being non-prime. All multiples of this element will be marked in this run etc. In the end the vector will contain a list of prime numbers. A typical C implementation of this algorithm might look as follows:

```
#include <stdio.h>

#define END 100

int main()
{
    int i, j, v[END + 1];

```

```
9       for (i = 2; i <= END; v[i++] = 1); /* Initialize the array */
10
11      /* Successively mark all non-prime elements: */
12      for (i = 2; i * i <= END; i++)
13          for (j = 2; v[i] && i * j <= END; v[i * j++] = 0);
14
15      /* Print the list of primes generated by this operation: */
16      for (i = 2; i <= END; i++)
17          if (v[i])
18              printf("%d ", i);
19
20      printf("\n");
21      return 0;
22  }
```

This solution is much more elegant than the simple trial divisions shown before, but the algorithm itself is rather cluttered by the language itself. Solving this problem in an array language like APL might look like this: $(\sim E \in E \circ . \times E)/E \leftarrow 1 \downarrow \iota E \leftarrow 100$

How does this work? Reading from right to left, a scalar variable E is set to the value 100. Applying the ι function to E yields a unit stride vector containing 100 elements: $(1, 2, \ldots, 100)$. Dropping the first element of this vector by $1 \downarrow$ in turn yields a vector $(2, 3, \ldots, 100)$ which is then stored again in the variable E.

The next step is building a matrix by computing the outer product of two of these vectors: $E \circ . \times E$ thus yields a matrix of the form

$$\begin{pmatrix} 4 & 6 & 8 & 10 & \ldots \\ 6 & 9 & 12 & 15 & \ldots \\ 8 & 12 & 16 & 20 & \ldots \\ 10 & 15 & 20 & 25 & \ldots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

which obviously does not contain any prime numbers at all since every element of this matrix is the product of at least two prime numbers. By applying $E \in$ to this matrix, a binary vector is generated which corresponds to the vector E in that it contains a 1 at every place where E contains a non-prime number and a 0 otherwise. Inverting this vector by $\sim$ yields a binary vector containing a 1 at every location corresponding to a prime number in E and a 0 everywhere else.

The last step is the selection of all elements from the vector E based on this selection vector yielding a list of primes between 2 and 100.[10]

The most remarkable thing about this program is the observation that no explicit loops and only a minimal amount of variables were required. This is a characteristic feature

[10] As elegant as this solution is, it should be noted that a naive implementation (which holds true for Lang5) requires excessive amounts of memory and CPU time. So for practical applications it is often necessary to decide between elegance and performance.

of all array programming languages yielding terse and very expressive programs. Of course, there is much more about array programming and APL but these two examples might suffice in showing some of the power of the array programming paradigm. The following chapter describes the basic design of Lang5, the simple stack based array programming language used throughout the remaining portions of this book.

2 The design of Lang5

2.1 Reverse Polish notation

Most programming languages use an algebraic notation style which allows expressions like

```
z += dx * (y + 2 * (ya + yb) + yc) / 6.
```

Typically, the evaluation of such expressions is performed in a left-to-right fashion and takes operator precedences into account which seems useful and natural but often unclear or erroneous assumptions about these operator precedences on the side of the programmer turn out to be the source of either unnecessary parentheses or of plain programming errors. Accordingly, Ken Iverson decided to abandon this scheme for his APL system and employ a strict right-to-left scheme of evaluation. This seems odd at first sight but it turns out that expressions written like this tend to be shorter and not as error-prone as those written in the traditional style. Evaluating a polynomial $(a,b,c,d,e)\prod x$ using the Horner scheme looks like

```
y = a + x * (b + x * (c + x * (d + x * e)))
```

when written as a traditional algebraic expression. Using Iverson's notation it is simplified to

```
y = a + x * b + x * c + x * d + x * e.
```

This is due to the fact of right-to-left evaluation. The first term is thus xe, followed by $d + xe$, which is then multiplied by x again yielding $x(d + xe)$ etc.[1]

Still another notation was developed by the Polish logician and philosopher Jan Łukasiewicz.[2] He created what became known as *Polish notation* or *prefix notation* in which an operator always precedes its operands. If the arity of all operators is fixed, this style of notation does not require any parentheses at all since the precedence of operations is solely determined by their respective position in an expression.[3] A

[1] The implementation of a parser for algebraic expressions working in APL-style from right to left which is based on [Holub 1985][pp. 165 ff.] is shown in appendix A.

[2] 12/21/1878–02/13/1956

[3] As Jens Breitenbach points out, the Polish notation is more natural from a mathematical point of view than the common infix notation, since arithmetic operations are represented functions taking two arguments: $p : \mathbb{N} \times \mathbb{N} \to \mathbb{N}, (x,y) \mapsto p(x,y) := x + y$

variant of this notation, called *reverse Polish notation*, or *RPN* for short, was developed independently several times starting with [Burks et al. 1954]. The programming languages Forth, developed by Charles H. Moore beginning in the late 1950s is completely based on RPN as were most of HP's pocket calculators. RPN systems are based on a so-called *stack*, a data-structure based on a list which can be extended by *pushing* elements onto it. Retrieving elements is done by an operation called *pop* which removes one element from the end of the stack and returns this element. Stacks are ubiquitous in modern computing as they are used to store parameters, local variables and return addresses for subroutine or function calls. Therefore most modern processor architectures feature at least basic stack operations like push and pop. It is only in programming languages like Forth or Lang5 that stacks are exposed to the programmer.

Evaluating the polynomial shown above on an RPN system like a traditional HP pocket calculator could be done by the following sequence of operations:

```
e x * d + x * c + x * b + x * a +
```

This would, in effect, first push e and x onto the stack, execute the binary multiplication operator *, which in turn removes the two topmost elements from the stack and pushes the result of the multiplication back onto the stack. This value is then incremented by d, multiplied by x and so on.

When designing a programming language, the first thing to determine is the way in which expressions should be written. Lang5 employs the RPN notation which has several advantages: It is rather easily implemented in an interpreter or compiler and programming in RPN-style turns out to be quite powerful. In addition to that, it is especially useful to see the progression of a complex calculation. The following code-example shows a simple RPN-calculator written in Perl:

```
1  use strict;
2  use warnings;
3
4  my @stack;
5
6  my %functions = (
7      '.' => sub { print pop(@stack), "\n"; },
8      '+' => sub { push(@stack, pop(@stack) + pop(@stack)); },
9      '*' => sub { push(@stack, pop(@stack) * pop(@stack)); },
10     '-' => sub { push(@stack, -(pop(@stack) - pop(@stack))); },
11     '/' => sub { push(@stack, 1 / (pop(@stack) / pop(@stack))); },
12 );
13
14 while (my $input = <STDIN>)
15 {
16     $functions{$_} ? $functions{$_}->() : push(@stack, $_)
17         for (split(/\s+/, $input));
```

```
18 }
```

At its heart is the stack which is just a global array.[4] The basic functions offered are stored as subroutine references in a hash `%functions`. The user input is read from the standard input channel and stored in the scalar variable `$input`. This line of input is then split on white-space characters to yield individual tokens which are then either used to execute a particular function or to push a value onto the stack.

If a token contains the name of a known operator or function, it is a valid key for the hash `%functions`. In these cases, the subroutine addressed by this key is executed, `$functions{$_}->()`, in all other cases the token is pushed onto the stack. Using this simple program, an algebraic expression like `3 * 2 + 1` can be evaluated by entering `1 2 3 * + .` – the single dot removes the topmost stack element and prints it to the standard output.

Exercise 1:

1. Extend this simple RPN calculator by adding two binary operators for exponentiation (`**`), modulo (`%`) etc.
2. Extend it by adding a unary operator `!` for calculating factorials.
3. Implement some basic error checking to make sure that there are enough elements on the stack for the various operations implemented. Ideally these checking routines would be implemented in a general way – one for all binary operators etc. One approach to accomplish this would be to change the hash `%functions` to a two-dimensional structure in which each function name points to a two-element hash containing two keys `code` and `type`. The value of `code` contains the reference to the actual routine performing the desired operation while `type` contains information about the routine as being of type unary, binary etc. This information could then be used to control a central error checking routine.

Of these three methods, left-to-right and right-to-left evaluation of arithmetic expressions and RPN the latter one was selected as the basis for Lang5.

2.2 Dynamic typing

Another basic aspect of programming languages is the question of whether to employ static typing as found in C or to use dynamic typing as used by most *dynamic programming languages*[5] There is an ongoing discussion between proponents of these two

[4] Being an introductory example, no provision has been made for avoiding a global stack or error checking etc.

[5] Also known as *scripting languages*.

paradigms. In static typing a variable gets a type like being an *integer* or *float* assigned and will be capable of holding a value of this type only during its life-time. This has several advantages – among others, it speeds up execution of programs since the type of a variable is known in advance and does not have to be determined during run-time. In addition to that, static typing allows a variety of programming errors to be catched during compile-time.

Dynamic typing, on the other hand, offers a high degree of flexibility: Why should one explicitly restrict a variable to holding only a single type of value while the executing instance, normally an interpreter, knows best which type a variable contains at run-time. The often expressed fear that this might open Pandora's box and yield errors hard to catch has not been proven over the years, instead, the use of languages using dynamic typing often yields significant productivity gains compared with languages based on a static typing concept.[6]

It was decided to use dynamic typing in Lang5. In effect, most of Perl's dynamic typing capabilities are used in Lang5 since the interpreter is written entirely in Perl. So a scalar element on the stack can hold integer values, floating point numbers as well as strings.

2.3 Data structures

If it were just for the two characteristics of employing RPN in conjunction with dynamic typing, Lang5 would be nothing more than a slightly generalized Forth. Since Lang5 was intended to be an array language, the RPN concept was extended in a way that allows nested arrays to be pushed onto the stack and processed as whole data structures. So the two basic data-structures employed by Lang5 are scalar values like integers, floating point numbers, and strings, and nested arrays of arbitrary dimension using scalars as their data elements. In this respect, Lang5 works quite like RPL, short for *Reverse Polish LISP*, the programming language used in late HP pocket calculators like the HP28 or HP48 and later models.

In Lang5 it is possible to push a nested data structure like the two-dimensional array

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

onto the stack as

```
[[1 2 3] [4 5 6] [7 8 9]]
```

This, of course makes it necessary to extend the basic operators to work on whole data-structures like this array instead of working on scalar values only. This is done implicitly by the interpreter which traverses data-structures as necessary and applies basic operators like +, * etc. in an element-wise fashion to corresponding elements of such data-structures.

[6] See [Ousterhout 1998] and [Prechelt 2010].

The following chapter describes the process of installing and running the Lang5-interpreter which will be used throughout the remaining sections of this book. Following this, the language itself is introduced and described in detail.

3 Installing and running Lang5

3.1 Installation

The following section will describe the design and implementation of a simple array programming language called Lang5. Most examples require access to a Lang5-interpreter which can be installed easily on most operating systems.[1] The only prerequisite requirement is a `Perl` interpreter since the Lang5-interpreter itself is written in `Perl`. If there is no `Perl` interpreter already installed on the destination system, this should be done first.[2]

The easiest way to get started with Lang5 is to download the distribution kit `lang5.zip` from `http://lang5.sourceforge.net`[3] which can be unzipped to any suitable location in the directory structure of the destination system. There is no need to install Lang5 into a special location like `/usr/local` or something like that on a UNIX system as the following example shows.[4] Figure 3.1 shows the directory structure which created by unzipping the Lang5-distribution kit.

```
$ cd
$ unzip ../Downloads/lang5.zip
Archive:  lang5.zip
   creating: lang5/
   creating: lang5/doc/
  inflating: lang5/doc/introduction.pdf
   creating: lang5/examples/
  inflating: lang5/examples/apple.5
  inflating: lang5/examples/bargraph.5
  inflating: lang5/examples/cantor.5
  inflating: lang5/examples/cosine.5
  inflating: lang5/examples/fibr.5
  inflating: lang5/examples/fibr_apply.5
  inflating: lang5/examples/fibr_unary.5
  inflating: lang5/examples/gauss_factorial.5
  inflating: lang5/examples/gauss_factorial_unary.5
  inflating: lang5/examples/gol.5
```

[1] Currently supported are LINUX, Mac OS X, Windows and OpenVMS.

[2] Typical `Perl` distributions for most common operating systems can be obtained from `http://www.perl.org/get.html` or `http://www.activestate.com/activeperl`.

[3] The direct link is `https://downloads.sourceforge.net/project/lang5/lang5.zip`.

[4] For a more permanent installation the `PATH` variable should be extended accordingly to include the interpreter's base directory.

```
  inflating: lang5/examples/matrix_vector.5
  inflating: lang5/examples/perfect.5
  inflating: lang5/examples/prime.5
  inflating: lang5/examples/prime_2.5
  inflating: lang5/examples/sine_curve.5
  inflating: lang5/examples/sort.5
  inflating: lang5/examples/sort.data
  inflating: lang5/examples/sum_of_cubes.5
  inflating: lang5/examples/throw_dice.5
  inflating: lang5/examples/ulam.5
  inflating: lang5/INSTALL
  inflating: lang5/lang5
  inflating: lang5/lang5.vim
   creating: lang5/lib/
  inflating: lang5/lib/mathlib.5
  inflating: lang5/lib/stdlib.5
   creating: lang5/perl_modules/
   creating: lang5/perl_modules/Array/
  inflating: lang5/perl_modules/Array/DeepUtils.html
  inflating: lang5/perl_modules/Array/DeepUtils.pm
 extracting: lang5/perl_modules/Array/pod2htmd.tmp
 extracting: lang5/perl_modules/Array/pod2htmi.tmp
   creating: lang5/perl_modules/Lang5/
  inflating: lang5/perl_modules/Lang5/String.pm
  inflating: lang5/perl_modules/Lang5.pm
   creating: lang5/perl_modules/Term/
   creating: lang5/perl_modules/Term/ReadLine/
  inflating: lang5/perl_modules/Term/ReadLine/Perl.pm
  inflating: lang5/perl_modules/Term/ReadLine/readline.pm
  inflating: lang5/README
$ cd lang5
$ chmod 755 lang5
$ ~/lang5/lang5
loading mathlib.5: Const..Basics..Set..Stat..Cplx..P..LA..Graph..
                   Trig..NT..
loading stdlib.5:  Const..Misc..Stk..Struct..
lang5> exit
$
```

The installation on a Windows system is equally simple (provided that there is a `Perl` interpreter already installed): The distribution kit `lang5.zip` can be unzipped to any suitable location even something like `C:\Temp\lang5`. The interpreter is then started simply from within a command window[5] by typing `perl c:\Temp\lang5\lang5`.

[5] This can be opened by executing `cmd.exe`.

```
lang5/
 +- lang5                    The interactive interpreter wrapper.
 +- lang5.vim                Syntax highlighting for the vi.
 +- perl_modules/
                +- Array     Contains the array functionality.
                +- Lang5     Contains the string module.
                +- Lang5.pm  The main interpreter module.
                +- Term/     Pure Perl ReadLine implementation.
 +- INSTALL                  Installation information.
 +- README                   Release information.
 +- doc/                     This directory contains some PDF files.
 +- examples/                Examples written in lang5.
 +- lib/                     Contains libraries loaded at startup.
       +- mathlib.5          A collection of mathematical user words.
       +- stdlib.5           Miscellaneous additional word definitions.
```

Figure 3.1: Directory structure created by the Lang5-distribution kit

3.2 Starting the interpreter

In the simplest case the Lang5-interpreter is just started as shown above, but there are many cases which require the use of so-called "qualifiers" which control the overall behavior of the interpreter. The general format for calling the interpreter is as follows:

```
lang5 [<qualifier> [<qualifier> .. [<qualifier>] .. ]] [file1 [file2 ...]]
```

`file_1` etc. represent names of files containing Lang5-source code that is to be executed by the interpreter. The available qualifiers are those listed below:

`-b` or `--benchmark`: By specifying this qualifier, the interpreter will print a list of all called functions, operators and words ordered by the number of executions descending.

`-d` or `--debug_level`: This qualifier sets the debug level of the interpreter which is useful mainly during development of the interpreter itself. Possible values are `TRACE`, `DEBUG`, `INFO`, `WARN`, `ERROR` (this is the default value for this parameter), `FATAL`. The last value generates the least amount of output while `TRACE` generates extremely detailed output and renders the interpreter mostly useless apart for debugging due to the amount of output. `lang5 -d TRACE` would start the interpreter in its trace mode where every tiny operation within the interpreter itself is printed.

`-e` or `--evaluate`: Using this qualifier it is possible to evaluate a Lang5-expression during startup of the interpreter and before loading any files containing Lang5-source code. This is useful to place initial values onto the stack or to execute

so-called one-liners. This option forces the interpreter into batch mode:[6]

```
$ lang5 -e "1 2 + ."
loading mathlib.5: Const..Basics..Set..Stat..Cplx..P..LA..Graph..
                   Trig..NT..
loading stdlib.5:  Const..Misc..Stk..Struct..
3
$
```

-f or --format: By default, numerical values are printed by the Lang5-interpreter with at least four places to ensure a pretty formatted output of nested arrays. In cases where this default behavior is not suitable, the control string for all output operations can be modified by this parameter. To have output values printed with at least 15 places, the interpreter could be started as follows:

```
$ lang5 -format "%15s"
loading mathlib.5: Const..Basics..Set..Stat..Cplx..P..LA..Graph..
                   Trig..NT..
loading stdlib.5:  Const..Misc..Stk..Struct..
lang5> [1 2] .
[              1              2  ]
```

-i or --interactive: When the interpreter is started without any file name specified on the command line, it will automatically enter interactive mode. If one or more files containing Lang5-source code are specified, the interpreter will run in batch mode by default, so it will execute the code and then quit. In cases where files should be loaded and the interpreter should nevertheless run in interactive mode, this can be forced by specifying this qualifier on the command line.

-n or --nolibs: By default, the interpreter loads all libraries located in the `lib` directory of the interpreter's directory tree. Specifying this qualifier, loading these libraries can be suppressed:

```
$ lang5
loading mathlib.5: Const..Basics..Set..Stat..Cplx..P..LA..Graph..i
                   Trig..NT..
loading stdlib.5:  Const..Misc..Stk..Struct..
lang5> exit

$ lang5 -n
lang5>
```

-s or --statistics: The `-statistics` qualifier behaves quite like `-benchmark` with the difference that the output will not be sorted by the number of calls made to the various functions, operators and words.

[6] More than one `-e`-qualifier may be supplied at once. If there are any source code files specified on the command line, they will be executed after the statements following `-e` have been executed.

-t or --time: Specifying -t, the interpreter will print out the time consumed for each line executed when running in interactive mode. When running in batch mode, the total run time will be printed.

-v or --version: This qualifier will cause the interpreter to print out its version number before executing any program.[7]

-w or --width: Some commands are aware of the current terminal width which is set to 80 by default. Using -w this width can be changed.

3.3 First steps

The very first example is the unavoidable hello-world-program written in Lang5. The first example shows how to write and run a hello-world in interactive mode:

```
$ lang5
loading mathlib.5: Const..Basics..Set..Stat..Cplx..P..LA..Graph..
                   Trig..NT..
loading stdlib.5:  Const..Misc..Stk..Struct..
lang5> "Hello world!\n" .
Hello world!
lang5> exit
```

Now create a file called hello_world.5 with your preferred text-editor.[8] This file should contain only one line containing the program typed in in the above example. This program is the executed by running the Lang5-interpreter in batch mode which is entered automatically since a file name is specified when starting the interpreter:

```
$ cat > hello_word.5
"Hello world!\n" .
$ lang5 hello_word.5
loading mathlib.5: Const..Basics..Set..Stat..Cplx..P..LA..Graph..
                   Trig..NT..
loading stdlib.5:  Const..Misc..Stk..Struct..
loading hello_word.5
Hello world!
$
```

Now let us execute some of the examples supplied with the Lang5-interpreter:

```
$ lang5 -b lang5/examples/sine_curve.5
loading mathlib.5: Const..Basics..Set..Stat..Cplx..P..LA..Graph..Trig..NT..
```

[7] The libraries are loaded before printing the version number.
[8] The UNIX shell command cat is far from being an editor but is useful in short examples like this.

```
loading stdlib.5:  Const..Misc..Stk..Struct..
loading lang5/examples/sine_curve.5
                          *
                                *
                                     *
                                          *
                                             *
                                              *
                                             *
                                          *
                                     *
                                *
                          *
                    *
               *
          *
     *
     *
     *
          *
               *
                    *
                          *
================================================================================
Statistics:
--------------------------------------------------------------------------------
execute word             :        572 ! Function                :        438 !
Push data                :        275 ! Binary                  :        130 !
+                        :         64 ! If                      :         63 !
compress                 :         63 ! Word definitions        :         61 !
Max. stack depth         :         50 ! roll                    :         42 !
swap                     :         42 ! _roll                   :         42 !
dup                      :         42 ! expand                  :         42 !
type                     :         42 ! .                       :         39 !
ne                       :         21 ! append                  :         21 !
eq                       :         21 ! <                       :         21 !
depth                    :         21 ! join                    :         21 !
reshape                  :         21 ! Unary                   :          4 !
*                        :          2 ! /                       :          1 !
int                      :          1 ! iota                    :          1 !
print_dot                :          1 ! sin                     :          1 !
--------------------------------------------------------------------------------
$
```

4 Lang5 basics

Lang5 is a small interpreted programming language borrowing heavily from APL and Forth[1] The basic goal of Lang5 is to serve as a language for teaching the basics of array programming as well as to serve as an introduction to the implementation of interpreters. Lang5 attempts to combine the particular strengths of Forth (in essence its stack based structure and the ability to extend the language itself by defining so-called *words* which can then be used in exactly the same way as built-in functions or operators) as well as those of APL (especially its array handling capabilities).

4.1 Getting started

So the central data structure employed by Lang5 is a so-called *stack* which in essence is a data structure that mainly relies on *pushing* elements onto it or removing elements by a *pop* operation. Thus a stack is a *LIFO*[2] data structure. Such structures are used in a variety of applications ranging from keeping track of subroutine calls and their respective parameters and return addresses to the generation of machine code for arithmetic expressions etc. The most influential stack oriented programming language is Forth which was conceived by Charles H. Moore in the late 1950s. Using a stack oriented language feels a bit unusual at the beginning but one gets used to its rather quickly. As an example, the expression (2+3)*4 is to be evaluated with Lang5. One way to do this is to push all arguments onto the stack and the applying the necessary binary operators + and *:

```
lang5> 4 3 2 + * .
20
```

The interpreter will push anything to its stack as long as it is not an operator or any other thing that can be executed. Individual values, function names, operators etc. are delimited by white space like a blank character or a newline, so 4 3 2 pushes three scalar values onto the stack which will contain the value 2 in its topmost position afterwards.[3] Executing a binary operator will always fetch the two topmost elements from the stack, use them as operands for the operator and push the result back onto the stack. So executing the binary operator + will pop the two topmost elements of the stack (2 and 3), add them and push the resulting value 5 back onto the stack which now contains the values 4 and 5.

[1] Quite like RPL, the language used in HP's advanced pocket calculators, borrows from LISP and Forth.
[2] Short for *Last In First Out*.
[3] The topmost position of a stack is called *Top Of Stack*, *TOS* for short.

Executing the binary operator * will then remove these two values from the stack and push back the result of the multiplication which is 20 in this case. The function . finally removes the topmost element from the stack and prints it.

Exercise 2:

1. Calculate the area of a circle with a diameter of 1.25 meters using the Lang5-interpreter. (Hint: `pi` pushes an approximation for π onto the stack.)
2. Devise as many different ways as possible to compute the sum of the values 2, 3, 5, 8 and 13 using Lang5.

4.2 Basic data structures

So the basic data structure within Lang5 is a stack. In contrast to Forth this stack can hold not only scalar values as shown in the preceding section but arrays of arbitrary structure.[4] Thus Lang5 distinguishes between scalar values and arrays which are pushed onto the stack. Examples for scalar values are 3.1415926535, 17 or even a string like "Hello world!":[5]

```
lang5> 3.1415926535

lang5> 17

lang5> "Hello world!"

lang5> .s
vvvvvvvvvvvvvvvvvvv Begin of stack listing vvvvvvvvvvvvvvvvvvv
Stack contents (TOS at bottom):
3.1415926535
17
Hello world!
^^^^^^^^^^^^^^^^^^^^ End of stack listing ^^^^^^^^^^^^^^^^^^^^
```

The real power of Lang5 results from its ability to deal with arbitrarily deeply nested arrays on the stack. An array is denoted by enclosing its values into square brackets as the following examples of a one-, a two- and a three-dimensional array shows:

```
lang5> [1]

lang5> [[1 2][3 4]]
```

[4] The term *array* will always denote an array of arbitrary dimension in the following.

[5] Using .s the current contents of the stack can be displayed without destroying the stack's contents as a repetitive application of . would do.

```
lang5> [[[1 2][3 4]][[5 6][7 8]]]

lang5> .s
vvvvvvvvvvvvvvvvvvvv Begin of stack listing vvvvvvvvvvvvvvvvvvvv
Stack contents (TOS at bottom):
[    1  ]
[
  [    1     2  ]
  [    3     4  ]
]
[
  [
    [    1     2  ]
    [    3     4  ]
  ]
  [
    [    5     6  ]
    [    7     8  ]
  ]
]
^^^^^^^^^^^^^^^^^^^^ End of stack listing ^^^^^^^^^^^^^^^^^^^^
```

Sometimes it is necessary to push an operator onto the stack rather than executing it directly. In this case it can be enclosed either in double quotes to force the interpreter to accept it as a simple string or it can be preceded with a single quote like this:

```
lang5> "+" .
+
lang5> '- .
-
```

Many operators and functions built into Lang5 work on scalars as well as on arrays. If a unary function like the factorial ! is applied to a scalar it just returns the corresponding factorial. If its argument is an array it will be automatically applied to all array elements:

```
lang5> 5 ! .
120
lang5> [0 1 2 3 4 5] ! .
[    1     1     2     6    24   120  ]
```

The same mechanism applies to binary operators and functions, so arrays can be easily added, multiplied, divided, subtracted etc. in an element-wise fashion:

```
lang5> [1 2] [3 4] + .
[     4      6   ]
```

What happens if the data structures are not "compatible" regarding their *shape*? The Lang5-interpreter first determines which of the two structures supplied as operands to a binary operator or function is bigger and the transforms the shape of the smaller to match that of the bigger structure. If the smaller array does not contain enough elements for this transformation, its elements are used over and over again to fill the required intermediate data structure. This also applies to cases where a binary operator or function is applied to an array and a scalar:

```
lang5> [1 2 3] [4 5] + .
[     5      7      7   ]
lang5> [1 2 3] 1 + .
[     2      3      4   ]
```

Exercise 3:

1. Compute and print the sum of the two following two-dimensional matrices:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \text{ and } \begin{pmatrix} 7 & 6 & 2 \\ 1 & 9 & 5 \\ 3 & 8 & 4 \end{pmatrix}$$

2. Subtract the value 1 from all elements of the first matrix shown above.

4.3 Language elements

The basic language elements of Lang5 are divided into built-in operators, functions, control instructions and so-called *words* which are user-defined and be used exactly like built-ins after their definition. This makes it possible to extend the language itself by defining words which are then used in programs. The following terms are used in the remaining portion of this book:

4.3.1 Operators

Since Lang5 is a stack based language as Forth, everything has to be written in post-fix notation, so operators, which are grouped into *niladic*, *unary* and *binary* operators will fetch 0, 1 or 2 elements from the stack, perform some operations on these values and push the result back onto the stack. Examples for operators are the well known basic arithmetic operators like +, - and the like.

As shown in the previous examples, operators work on scalars only – if applied to arrays the Lang5-interpreter will traverse the nested data structure and apply the opera-

tor to the basic scalar elements forming the base of the array, so `[1 2 3] 1 +` will add one to every element of the array `[1 2 3]` yielding `[2 3 4]`. This holds true for unary as well as binary operators – if a binary operator is applied to two arrays these should be of the same shape and size – otherwise the dimensional larger operand determines how the smaller operand will be restructured automatically – and the operator will then be applied to corresponding elements from each of the arrays.

In addition to that, binary operators (and binary user-defined words, see section 4.3.5) can be used in conjunction with the array operation `outer` which creates data structures like outer products and the like (see below) as well as with `reduce` which applies an operator or word to elements of an array yielding a result with a dimension which is by one lower than the dimension of the input array.

4.3.2 Functions

While operators work on scalars only and always push exactly one result value back onto the stack, so-called *functions* act in a more general way as they can operate on a value on the stack as a whole and not in an element-wise fashion. An example for such a function is `dup` which duplicates the topmost stack element:

```
lang5> [1 2 3] dup .s
vvvvvvvvvvvvvvvvvvvv Begin of stack listing vvvvvvvvvvvvvvvvvvvv
Stack contents (TOS at bottom):
[    1    2    3  ]
[    1    2    3  ]
^^^^^^^^^^^^^^^^^^^^ End of stack listing ^^^^^^^^^^^^^^^^^^^^
```

4.3.3 Control instructions

Lang5 only features a few so-called *control instructions* which are used to implement conditional execution of program parts and loops.[6] The most basic control instruction is the `if-else-then` construction:[7]

```
lang5> 1 2 > if "greater" else "less or equal" then .
less or equal
```

The keyword `if` removes the topmost stack element and performs all actions up to the next `else` or `then`, whichever comes first, if this value does not evaluate to false. Typically everything which is not 0 or undefined is considered as being false.

The only built-in loop construction is `do-loop` which implements an endless loop which can be terminated with `break`:

[6] See section 5.5.
[7] The `else`-part is optional.

```
lang5> 1 do dup . 1 + dup 5 > if break then loop
1
2
3
4
5
```

Exercise 4:
Modify the program shown above so that the loop counts down from 5 to 1 and then exits.

4.3.4 I/O instructions

Such an instruction either reads or writes data from or to a device (normally `stdin` and `stdout` if no IO-redirection has been executed). One of the output instructions, `.`, has been used extensively in the preceding sections already, see section 5.3 for more information.

4.3.5 Words

A word is a user-defined collection of operators, functions, constants etc. which can be accessed later on by issuing the name of the word. Thus a word acts very much like a subroutine in a conventional programming language like Perl. User-defined words come in three flavors:

Simple words: Such a word acts like a function described above.

Unary words: A unary word can be used exactly like a built in unary operator.

Binary words: A binary word can be used like a binary operator (especially in conjunction with `reduce` and `outer` etc.).

The concept of user-defined words which has been pioneered in `Forth` is one of the main features of Lang5. Defining a simple word looks quite the same as it would in `Forth`: The word definition itself is started with a colon, followed by a white space and the name of the new word (or that of a word to be overridden). All following commands are grouped into this word until a white space separated semicolon is encountered. In the following example a word named `square` is defined which computes the square of the topmost stack element:

```
lang5> : square dup * ;

lang5> 5 square .
25
```

Such a *simple* word works on the stack elements as is – it will not be applied in an element-wise fashion as binary operators, although in some cases it will look like this.

Exercise 5:

The `square` word defined above is applied to the topmost stack element as a whole – explain the behavior of the interpreter in this example:

```
lang5> : square dup * ;

lang5> [1 2 3] square .
[     1     4     9   ]
```

Why is the result a three-element vector as the original operand and why have the elements of the operand been squared if the word itself has not been applied automatically to each vector element?

Things get more interesting when unary or binary words are defined since these behave exactly like other built-in unary and binary operators. In the first example a unary word `print` will be defined which will by applied automatically in an element-wise fashion by the Lang5-interpreter if its argument is an array:

```
lang5> : print(*) . ;

lang5> 1 print
1
lang5> [1 2 3] print
1
2
3
lang5> [[1 2][3 4]] print
1
2
3
4
```

The suffix `(*)` following the word's name denotes that this will be a unary word. The star denotes that the interpreter does not have to care about so called *dressed data structures* which are described in section 4.4.

A binary word is defined in the easiest case by specifying `(**)` as suffix of the word's name as the following example shows:[8]

```
lang5> : binary_print(*,*) . . "------\n" . ;

```

[8] The control sequence \n used within a string generates a newline character sequence.

```
lang5> [1 2 3] [4 5 6] binary_print
4
1
------
5
2
------
6
3
------
```

Exercise 6:
What will happen if this binary word `binary_print` is applied to a vector `[1 2 3]` and a scalar `1` instead of two vectors? Explain the behavior of the interpreter.

4.3.6 Variables

Although variables are required less often in Lang5 than in most other programming languages, their use often makes a program more readable and maintainable. Variables are set using `set` which expects the name of the variable to be set (and declared if it not already exists) on the topmost stack element and the value which is to be stored in the variable in the stack element just below. The value stored in a variable is pushed onto the stack by just entering the name of the variable:

```
lang5> 2 'x set

lang5> x x * .
4
```

Using the word `.v` a list of all variables currently defined can be displayed.[9]

Exercise 7:
Apply the simple word `square` defined above to a variable called `vector` which has been set to `[1 2 3 4]` before.

4.4 Dressed data structures

So-called *dressed data structures* are the basis for overloading words. As powerful as unary and binary user-defined words are, it is often necessary to make clear to what type of data such a word is to be applied by the interpreter. Therefore the data structures on the stack must be marked by *dressing* them. This operation assigns a meta information to a data structure without altering its contents.

[9] This list also contains some special purpose variables which control the operation of the Lang5-interpreter – see appendix B.

A data structure can be dressed in two ways: Either implicitly by writing the dress code enclosed in parentheses behind the structure or by using the `dress` function. In the following example two data structures are defined with the dress codes `foo` and `bar`:

```
lang5> [1 2](foo)

lang5> [3 4] 'bar dress

lang5> .s
vvvvvvvvvvvvvvvvvvvv Begin of stack listing vvvvvvvvvvvvvvvvvvvv
Stack contents (TOS at bottom):
[    1    2  ](foo)
[    3    4  ](bar)
^^^^^^^^^^^^^^^^^^^^ End of stack listing ^^^^^^^^^^^^^^^^^^^^
```

Applying a simple unary word like `print` as defined above will fail since the interpreter will only apply it in an element-wise fashion to the elements of an undressed array. In the case of a dressed data structure the interpreter will search for a unary or binary word with the proper dress code which fails in this example:

```
lang5> print
Error: no handler for type 'bar'
```

Using the dress code it is now possible to overload the word `print` to deal with data structures having the dress code `foo` (and `bar` respectively):

```
lang5> [1 2](foo)

lang5> : print(foo) "foo:" . .  ;
lang5> print
foo:[    1    2  ](foo)
lang5> .s
vvvvvvvvvvvvvvvvvvvv Begin of stack listing vvvvvvvvvvvvvvvvvvvv
Stack contents (TOS at bottom):
^^^^^^^^^^^^^^^^^^^^ End of stack listing ^^^^^^^^^^^^^^^^^^^^
```

Why is there an undefined value on top of the stack now? The reason is simple: The unary word `print` with dress code `(foo)` has been applied to the data structure as a whole. Whatever this word returns on the stack will be used as the result of this word. Since the last `.` consumes the dressed data structure by printing it, there is no return value from `print(foo)` and thus the stack contains an undefined element instead of the original dressed data structure.

This brings up the following question: What does `print(foo)` "see" on the stack when it is being executed?

```
lang5> 1

lang5> [2 3](foo)

lang5> : print(foo) .s ;

lang5> .s
vvvvvvvvvvvvvvvvvvv Begin of stack listing vvvvvvvvvvvvvvvvvvv
Stack contents (TOS at bottom):
1
[    2     3  ](foo)
^^^^^^^^^^^^^^^^^^^^ End of stack listing ^^^^^^^^^^^^^^^^^^^^
lang5> print
vvvvvvvvvvvvvvvvvvv Begin of stack listing vvvvvvvvvvvvvvvvvvv
Stack contents (TOS at bottom):
[    2     3  ](foo)
^^^^^^^^^^^^^^^^^^^^ End of stack listing ^^^^^^^^^^^^^^^^^^^^
lang5> .s
vvvvvvvvvvvvvvvvvvv Begin of stack listing vvvvvvvvvvvvvvvvvvv
Stack contents (TOS at bottom):
1
[    2     3  ](foo)
^^^^^^^^^^^^^^^^^^^^ End of stack listing ^^^^^^^^^^^^^^^^^^^^
```

This needs some explanation: At first, two values, 1 and `[2 3](foo)`, are pushed onto the stack as shown by the following `.s`. Another `.s` invoked as part of the definition of `print(foo)` only shows the dressed structure `[2 3](foo)` while a third `.s` invoked after the execution of `print(foo)` has completed shows both values again...

The reason for this is simple: All unary and binary words operate in a *stack jail*, i. e. they get a temporary stack which only contains the one (in the unary case) or two topmost elements (for binary words) of the main stack. This makes it impossible for a word to have side effects on the main stack, since every such word gets its own temporary stack to work in which is destroyed after completion of the word's execution. At this point the topmost element of this temporary stack is copied back to the main stack where it serves as the result of the word.

As complicated as dressed data structures may seem, they are very powerful and used in many places in the libraries of Lang5. A typical example are the basic arithmetic operators which are overloaded in the mathematical library to work transparently on complex numbers. By convention, a complex number, consisting of a real and an imaginary part, is denoted by the dress code `(c)` like this:

```
lang5> [1 2](c) [3 4](c) * .
[   -5    10  ](c)
```

Two undressed arrays would have been multiplied element-wise as the following example shows:

```
lang5> [1 2] [3 4] * .
[    3     8  ]
```

The definition of the overloaded multiplication word for complex numbers which can be found in the mathematical library[10] looks like this:[11]

```
# Multiplication of two complex numbers.
: *(c,c)
  strip swap strip swap
  [0 1 0 1] subscript swap [0 1 1 0] subscript
  * expand drop
  + rot rot - swap
  2 compress 'c dress
;
```

An essential feature of most words operating on dressed data structures can be seen at the beginning of this overloaded multiplication: Using the `strip` function the data structure is undressed – otherwise an endless recursion would occur if another multiplication becomes necessary in this word which would call the word again etc. `swap` is a typical function in stack oriented programming languages the exchanges the two topmost stack elements, so `strip swap strip swap` strips both operands and making sure that their order on the temporary stack remains unchanged.

> **Exercise 8:**
> Define overload definition of the binary word `*` which operates on two data structures dressed with `(baz)`. This new word shall multiply the elements of the two operand vectors element-wise but with a change of sign. The result should be a `(baz)` data structure again.

Currently, the five following dress codes are used in the Lang5-libraries:

c: Complex numbers

m: Matrices

p: Polar coordinate tupels

s: Sets

v: Vectors

[10] See section D.

[11] This serves just as an example since most of the functionality used here has not yet been introduced.

5 The Lang5 dictionary

Lang5 comes with a decent complement of built-in functions, operators and word (these are defined in so-called *libraries* which are loaded during startup and are contained in the `lib` directory of the Lang5-directory tree. The following sections list all of these elements grouped as follows:

- Stack manipulation and display
- Array manipulation and generation
- File handling
- Mathematical, logical and comparison operations
- Control structures
- Miscellaneous operators and functions
- Variable and word handling

The following sections describe all of the available functions, operators and words grouped as above and sorted alphabetically within each group.

5.1 Stack manipulation and display

The following functions and words build the backbone of the stack manipulation capabilities of Lang5. These functions and words operate directly on the stack and do not care about the structure of any element on the stack, so `swap` will just interchange the two topmost stack elements regardless of their dress code or structure.

5.1.1 . .

This function is normally used for debugging purposes only since it prints the stack's contents using the `Perl`'s `Data::Dumper` yielding not very readable output. `. .` does not affect the values on the stack:

```
lang5> 1 2 [3 4] [5 6](foo) ..
$1 = [
  1,
  2,
```

```
  [
    '3',
    '4'
  ],
  bless( [
    '5',
    '6'
  ], 'foo' )
];
```

5.1.2 .s

This word from the standard library[1] prints all elements on the stack in non-destructive way using the standard output format of Lang5 which is pretty useful during program development:

```
lang5> 1 2 [3 4] [5 6](foo) .s
vvvvvvvvvvvvvvvvvvv Begin of stack listing vvvvvvvvvvvvvvvvvvv
Stack contents (TOS at bottom):
1
2
[    3     4  ]
[    5     6  ](foo)
^^^^^^^^^^^^^^^^^^^^^ End of stack listing ^^^^^^^^^^^^^^^^^^^^^
```

5.1.3 clear

This word from the standard library removes all elements from the stack:

```
lang5> 1 2 [3 4] [5 6](foo) clear .s
Stack is empty!
```

5.1.4 depth

The depth function pushes the number of elements on the stack onto the stack:

```
lang5> clear 1 2 [3 4] [5 6](foo) depth .
4
```

[1] See section C.

5.1.5 drop

This function removes the topmost element of the stack. An empty stack will cause an error:

```
lang5> clear 1 drop

lang5> drop
Error: too few elements on stack, expected X
History: drop clear depth 0 > if ARRAY(0x93bb40) 1 drop drop
```

5.1.6 dup

The dup function duplicates the topmost stack element which is often useful for the conditional execution of program parts and the like:

```
lang5> clear [1 2] dup .s
vvvvvvvvvvvvvvvvvvvv Begin of stack listing vvvvvvvvvvvvvvvvvvvv
Stack contents (TOS at bottom):
[    1    2  ]
[    1    2  ]
^^^^^^^^^^^^^^^^^^^^ End of stack listing ^^^^^^^^^^^^^^^^^^^^
```

5.1.7 2dup

This word from the standard library duplicates the two topmost stack elements:

```
lang5> clear 1 2 2dup .s
vvvvvvvvvvvvvvvvvvvv Begin of stack listing vvvvvvvvvvvvvvvvvvvv
Stack contents (TOS at bottom):
1
2
1
2
^^^^^^^^^^^^^^^^^^^^ End of stack listing ^^^^^^^^^^^^^^^^^^^^
```

5.1.8 ndrop

ndrop which is also defined in the standard library drops the *n* topmost stack elements where *n* is contained in the topmost element which itself does not count:

```
lang5> clear 1 2 3 2 ndrop .s
vvvvvvvvvvvvvvvvvvvv Begin of stack listing vvvvvvvvvvvvvvvvvvvv
```

```
Stack contents (TOS at bottom):
1
^^^^^^^^^^^^^^^^^^^^^^ End of stack listing ^^^^^^^^^^^^^^^^^^^^^^
```

5.1.9 over

This function the second element from the top of stack to the top of stack:

```
lang5> clear 1 2 over .s
vvvvvvvvvvvvvvvvvvvv Begin of stack listing vvvvvvvvvvvvvvvvvvvv
Stack contents (TOS at bottom):
1
2
1
^^^^^^^^^^^^^^^^^^^^^^ End of stack listing ^^^^^^^^^^^^^^^^^^^^^^
```

Exercise 9:
Define a new word `twodup` that works like `2dup` using the `over` function.

5.1.10 pick

A generalized form of `over` is the word `pick` from the standard library. Controlled by a natural number on top of the stack it will pick the element n positions down the stack non-destructively and push a copy of this element on the stack:

```
lang5> clear 1 2 3 4 3 pick .s
vvvvvvvvvvvvvvvvvvvv Begin of stack listing vvvvvvvvvvvvvvvvvvvv
Stack contents (TOS at bottom):
1
2
3
4
1
^^^^^^^^^^^^^^^^^^^^^^ End of stack listing ^^^^^^^^^^^^^^^^^^^^^^
```

Exercise 10:
Define yet another word performing the same action as `2dup`, this time using `pick`.

5.1.11 _roll

This function implements a very generalized rotation on the elements on the stack. It expects the number of rotation steps in the topmost stack element and the depth the rotation shall cover in the element below:

```
lang5> clear 1 2 3 4 5 6 4 2 _roll .s
vvvvvvvvvvvvvvvvvvvv Begin of stack listing vvvvvvvvvvvvvvvvvvvv
Stack contents (TOS at bottom):
1
2
5
6
3
4
^^^^^^^^^^^^^^^^^^^^ End of stack listing ^^^^^^^^^^^^^^^^^^^^
```

5.1.12 roll

This standard library word is a variant of _roll. It requires only the depth of the rotation to be performed in the topmost stack element. Every execution of roll will then perform a single rotation step.

Exercise 11:
Define a word called myroll based on _roll that behaves like roll.

5.1.13 rot

This word defined in the standard library is the equivalent to the classic Forth function and performs a single rotation on the three topmost stack elements:

```
lang5> clear 1 2 3 rot .s
vvvvvvvvvvvvvvvvvvvv Begin of stack listing vvvvvvvvvvvvvvvvvvvv
Stack contents (TOS at bottom):
2
3
1
^^^^^^^^^^^^^^^^^^^^ End of stack listing ^^^^^^^^^^^^^^^^^^^^
```

5.1.14 swap

The swap function interchanges the two topmost stack elements:

```
lang5> 1 2 swap . .
1
2
```

Exercise 12:
Define a word `myswap` that implements the functionality of `swap` using only operations like `rot`, `over` and the like.

5.2 Array manipulation and generation

The functions, operators and words described in the following implement the array capabilities of Lang5. In contrast to the stack operations described above they are "aware" of the data structures they act upon.

5.2.1 append

The append word defined in the standard library expects two arrays or an array and a scalar in the two topmost stack elements and returns a larger array containing the concatenation of the elements of these two operands:

```
lang5> [1 2] [3 4] append .
[    1    2    3    4  ]
lang5> [1 2] 3 append .
[    1    2    3  ]
```

5.2.2 apply

The function `apply` applies a unary or binary operator or user-defined word along the first axis of an array (or two arrays) as the following example shows:

```
lang5> [[1 2][3 4]]

lang5> : print(*) . ;

lang5> dup print
1
2
3
4
lang5> 'print apply
[    1    2  ]
[    3    4  ]
```

In the first example, the unary word `print` is executed on a copy of the two-dimensional array `[[1 2][3 4]]`. Since this word has been defined as a unary word denoted by the suffix `(*)`, the interpreter implicitly applies the word to every single element

of the array. Using `apply` this unary word is only called twice – once for each one-dimensional sub-array of the original array yielding the output shown at the bottom of the example.

5.2.3 collapse

The `collapse` function "flattens" a nested array and returns a one-dimensional array containing all elements of the argument:

```
lang5> [[1 2][3 4]] collapse .
[    1    2    3    4  ]
```

5.2.4 compress

This function takes *n* elements from the stack and forms an array from these elements. It expects the number of elements to be fetched in the topmost stack element:

```
lang5> 1 2 3 4 4 compress .
[    1    2    3    4  ]
```

5.2.5 dreduce

This word from the standard library applies the `reduce` function (see section 5.2.16) repeatedly on a nested array structure until only a scalar value is left. It expects a binary operator on the topmost stack element and an array below. It then applies this operator between each two successive array elements:

```
lang5> [[1 2][3 4]] '+ dreduce .
10
```

> **Exercise 13:**
> Implement a word `mydreduce` that performs the same operation as `dreduce`, based on `reduce`.

5.2.6 dress

The `dress` function *dresses* an array with a name, a so-called *dress code*.[2] The explicit form looks like `[1 2] 'c dress` while the same effect can be achieved with implicit dressing as in `[1 2](c)`. This function returns the dress code of a dressed data structure – if applied to a non-dressed structure or a scalar yields an undefined valued:

[2] See section 4.4.

```
lang5> [1 2](c) dressed .
c
lang5> 1 dressed .
undef
```

It should be noted that dressed does not remove the structure its operates on from the stack, it just pushes the dress code as a string onto the stack.

5.2.7 expand

expand is a function that expects an array in the topmost stack element. It removes this element from the stack and places all array elements along the first axis back onto the stack. After expanding the array in this way, the number of elements placed onto the stack is pushed onto the stack:

```
lang5> clear [[1 2] [3 4] 5] expand .s
vvvvvvvvvvvvvvvvvvv Begin of stack listing vvvvvvvvvvvvvvvvvvv
Stack contents (TOS at bottom):
[    1    2  ]
[    3    4  ]
5
3
^^^^^^^^^^^^^^^^^^^^^ End of stack listing ^^^^^^^^^^^^^^^^^^^^^
```

5.2.8 extract

This word from the standard library extracts a particular element from an array removing it from the array and placing the element just removed onto the stack:

```
lang5> clear ['this 'is 'a 'test] 2 extract .s
vvvvvvvvvvvvvvvvvvv Begin of stack listing vvvvvvvvvvvvvvvvvvv
Stack contents (TOS at bottom):
[ this is test ]
a
^^^^^^^^^^^^^^^^^^^^^ End of stack listing ^^^^^^^^^^^^^^^^^^^^^
```

5.2.9 grade

The grade function expects a one-dimensional array (a vector) in the topmost stack element and generates a vector of the same size containing index values[3] which, if used as argument for a subscript function (see section 5.2.28) will yield a sorted vector:

[3] As in most programming languages, indices start at 0.

```
lang5> clear [3 1 4 1 5 9 2 6 5 3 5] grade .s
vvvvvvvvvvvvvvvvvvvv Begin of stack listing vvvvvvvvvvvvvvvvvvvv
Stack contents (TOS at bottom):
[    3    1    4    1    5    9    2    6    5    3    5  ]
[    3    1    6    9    0    2    8    4   10    7    5  ]

^^^^^^^^^^^^^^^^^^^^ End of stack listing ^^^^^^^^^^^^^^^^^^^^
lang5> subscript .
[    1    1    2    3    3    4    5    5    5    6    9  ]
```

5.2.10 in

`in` implements the element-of function from set theory. It is best demonstrated with a simple example:

```
lang5> [1 3 5] [1 2 3 4 5] in .
[    1    0    1    0    1  ]
```

Using the `select` function (see section 5.2.22) the vector resulting from executing `in` can be used to select only those elements of a data structure having a non-zero value in corresponding locations of this result vector.

5.2.11 index

This function generates an index vector which represents the location of the elements of the structure found on the topmost stack element in the structure just below it. If an element is not contained in this structure and empty index element is returned:

```
lang5> [3 1 4 1 5 9 2 6 5 3 5] [1 2 3 4 5 6 7] index .
[
  [    1  ]
  [    6  ]
  [    0  ]
  [    2  ]
  [    4  ]
  [    7  ]
  [ ]
]
```

5.2.12 iota

This unary built-in expects a positive integer value n on the top of the stack which will be removed. `iota` generates a vector with unit stride, starting at 0 and containing n elements:

```
lang5> 5 iota .
[    0    1    2    3    4  ]
```

5.2.13 join

The join function is the inverse of split (see section 5.2.25) – it concatenates the elements of a vector using a glue string and thus forms a single result string which is pushed onto the stack:

```
lang5> 4 iota " plus one equals " join .
0 plus one equals 1 plus one equals 2 plus one equals 3
```

5.2.14 length

The length function returns the number of elements along the first axis of the array found in the topmost stack element. This function does not remove the array from the stack:

```
lang5> [1 2 3] length .s
vvvvvvvvvvvvvvvvvvv Begin of stack listing vvvvvvvvvvvvvvvvvvv
Stack contents (TOS at bottom):
[    1    2    3  ]
3
^^^^^^^^^^^^^^^^^^^^ End of stack listing ^^^^^^^^^^^^^^^^^^^^
```

5.2.15 outer

outer is one of the most mighty functions in Lang5. It expects a binary operator on the top of the stack and two vectors in the stack elements just below. It will then create an outer "product"[4] of these two vectors based on the operator. Thus a simple multiplication table can be created as follows:

```
lang5> 10 iota 1 + dup '* outer .
[
  [    1    2    3    4    5    6    7    8    9   10  ]
  [    2    4    6    8   10   12   14   16   18   20  ]
  [    3    6    9   12   15   18   21   24   27   30  ]
  [    4    8   12   16   20   24   28   32   36   40  ]
  [    5   10   15   20   25   30   35   40   45   50  ]
  [    6   12   18   24   30   36   42   48   54   60  ]
  [    7   14   21   28   35   42   49   56   63   70  ]
```

[4] The type of which depends on the operator used.

```
  [     8     16     24     32     40     48     56     64     72     80   ]
  [     9     18     27     36     45     54     63     72     81     90   ]
  [    10     20     30     40     50     60     70     80     90    100   ]
]
```

5.2.16 reduce

This function also expects a binary operator in the topmost stack element and a vector just below. It will then apply the operator between all successive elements of the vector and push the result of this operation back to the stack. Using `reduce` the sum of the first 100 integers can be computes as follows:

```
lang5> 100 iota 1 + '+ reduce .
5050
```

`reduce` always operates along the first axis of the array to be processed, so here the + operator acts on the three one-dimensional components of the two-dimensional array:

```
lang5> [[1 2 3][4 5 6][7 8 9]] '+ reduce .
[    12     15     18   ]
```

If applied to an empty array it returns the value of the neutral element of the binary operator to which it was applied if this operator features a neutral element. This is often useful when dealing with boundary cases like the problem of handling the value zero in a factorial computation:

```
lang5> [] '+ reduce .
0
lang5> [] '* reduce .
1
```

Exercise 14:

Define a unary word named `myfactorial` to calculate the factorial of a positive integer value on the top of the stack.

5.2.17 remove

This function expects a scalar or a vector in the topmost stack element and an array at the element just below. It will then remove all elements from the first dimension of this array which are addressed by the elements of the vector or scalar found in the TOS:

```
lang5> 4 iota 1 + [2 3] remove .
[    1    2  ]
lang5> [[1 2 3][4 5 6][7 8 9]] 1 remove .
[
  [    1    2    3  ]
  [    7    8    9  ]
]
```

5.2.18 reshape

The reshape function allows the manipulation of the structure of arrays. It expects two arrays in the two topmost stack elements: a vector describing the structure of the resulting array and the array the structure of which is to be changed.[5] The following example shows how to build a three-dimensional array based on a two dimensional vector using reshape:

```
lang5> 8 iota 1 + [2 2 2] reshape .
[
  [
    [    1    2  ]
    [    3    4  ]
  ]
  [
    [    5    6  ]
    [    7    8  ]
  ]
]
```

If there are not enough elements in the source array to fill the structure of the destination array, its elements are reread from the beginning over and over again until the result has been populated with values:[6]

```
lang5> 1 [2 2] reshape .
[
  [    1    1  ]
  [    1    1  ]
]
```

Exercise 15:
Define a unary word that expects a natural number n on the top of the stack and will create an identity matrix having n columns and rows.

[5] Thus reshape resembles the binary case of APL's ρ.
[6] This behavior also applies if the source structure of the reshape operation is a simple scalar.

5.2.19 reverse

This function reverses the elements along the first axis of an array:

```
lang5> [1 2 3] reverse .
[    3    2    1  ]
lang5> [[1 2] [3 4]] reverse .
[
  [    3    4  ]
  [    1    2  ]
]
```

5.2.20 rotate

The `rotate` function rotates an *n*-dimensional array along its axes. It expects the structure to be rotate on the second to top element of the stack and a vector controlling to rotation in the topmost stack element. The following example first shows a single rotation along the first axis, followed by a rotation by -2 along the second axis and, finally, a combined rotation along both axes of a two-dimensional array:

```
lang5> 9 iota [3 3] reshape

lang5> dup [1 0] rotate .
[
  [    6    7    8  ]
  [    0    1    2  ]
  [    3    4    5  ]
]
lang5> dup [0 -2] rotate .
[
  [    2    0    1  ]
  [    5    3    4  ]
  [    8    6    7  ]
]
lang5> [1 1] rotate .
[
  [    8    6    7  ]
  [    2    0    1  ]
  [    5    3    4  ]
]
```

5.2.21 scatter

This function distributes (scatters) the values of a one dimensional array into a new data structure, controlled by an index vector as the following example shows:

```
lang5> ['a 'b 'c] [[0 0][0 1][1 0][1 1]] scatter .
[
  [ a b ]
  [ c a ]
]
```

As with other functions, the elements of the source array are reread from the beginning if there are not enough elements to fill the destination structure.

5.2.22 select

The `select` function selects elements from an array. It expects a one-dimensional array at the topmost stack element containing values evaluating to `true` (anything except 0) or `false` (the value 0) and an array in the element below. The values of the control vector determine which elements from the second array are to be included in the resulting data structure:

```
lang5> ['a 'b 'c] [1 0 1] select .
[ a c ]
```

5.2.23 shape

shape is the inverse function to `reshape` – it returns a vector describing the structure of an array. This result vector contains one element for each dimension of the source data structure which has been examined, containing the number of elements along this particular axis.[7] Obviously, applying shape twice to an array returns its dimensionality:

```
lang5> [[1 2][3 4]] shape .
[ 2 2 ]
lang5> [[1 2][3 4]] shape shape .
[ 2 ]
```

shape does not remove the array that it operated upon from the stack.

5.2.24 slice

The `slice` function expects a source array on the element below the TOS and a two-element vector in the topmost stack element. This array contains two coordinate tuples controlling the slicing operation by defining an upper left and lower right "corner" of the n-dimensional cube represented by the source array. The following example shows

[7] Thus shape resembles the unary case of APL's ρ.

this behavior: First, a three-dimensional cube containing 64 elements running from 0 to 63 is created. Then a sub-cube defined by the two coordinate vectors `[1 1 1]` and `[2 2 2]` describing the upper left front and lower left back corner is extracted from this structure and printed:

```
lang5> 64 iota [4 4 4] reshape [[1 1 1][2 2 2]] slice .
[
  [
    [  21  22 ]
    [  25  26 ]
  ]
  [
    [  37  38 ]
    [  41  42 ]
  ]
]
```

The two-element control vector containing the coordinate tupels can also be used to slice data from a higher perspective be omitting one or more dimensions:

```
lang5> 64 iota [4 4 4] reshape [[1] [3]] slice .
[
  [
    [  16  17  18  19 ]
    [  20  21  22  23 ]
    [  24  25  26  27 ]
    [  28  29  30  31 ]
  ]
  [
    [  32  33  34  35 ]
    [  36  37  38  39 ]
    [  40  41  42  43 ]
    [  44  45  46  47 ]
  ]
  [
    [  48  49  50  51 ]
    [  52  53  54  55 ]
    [  56  57  58  59 ]
    [  60  61  62  63 ]
  ]
]
```

Here the first two-dimensional planes from the three-dimensional 64-element cube have been sliced out.

5.2.25 split

The binary operator split, which is the inverse of join,[8] expects two scalars on the topmost stack elements: a regular expression and a string. The regular expression controls where the string is split into parts. These parts are the combined into a result array that is placed back onto the stack:

```
lang5> "this is a string" " " split .
[ this is a string ]
```

5.2.26 spread

The spread function applies a binary operator to the element of an array in a successive manner:

```
lang5> [1 2 3] '+ spread .
[    1     3     6  ]
```

So the first element of the result is the first element of the original array, the next element of the result is the sum of the first and second elements of the source, the third element is calculated by applying the binary operator to the first three elements of the source array etc.

Exercise 16:

Generate an array containing the squares of the first ten natural numbers without using multiplication etc. Just use spread and remember that squares can be easily generated by adding odd numbers with stride 2.

5.2.27 strip

The strip function removes the dress code of an array. It is the inversion function to dress:[9] If applied to a non-dressed array, nothing will happen.

```
lang5> [1 2](foo) strip .
[    1     2  ]
lang5> [1 2] strip .
[    1     2  ]
```

5.2.28 subscript

This function selects data from an array structure based on a vector containing coordinate vectors as the following example shows:

[8] See section 5.2.13.
[9] See section 5.2.6.

```
lang5> 64 iota [4 4 4] reshape [1 [1 1 1] [2 2 2]] subscript .
[
  [
    [   16    17    18    19  ]
    [   20    21    22    23  ]
    [   24    25    26    27  ]
    [   28    29    30    31  ]
  ]
    21    42
]
```

5.2.29 transpose

The transpose function performs a generalized matrix transposition. It expects a control value on the top of the stack and the array to be transposed in the stack element just below:

```
lang5> 9 iota [3 3] reshape dup 1 transpose .s
vvvvvvvvvvvvvvvvvvvv Begin of stack listing vvvvvvvvvvvvvvvvvvvv
Stack contents (TOS at bottom):
[
  [    0     1     2  ]
  [    3     4     5  ]
  [    6     7     8  ]
]
[
  [    0     3     6  ]
  [    1     4     7  ]
  [    2     5     8  ]
]
^^^^^^^^^^^^^^^^^^^^ End of stack listing ^^^^^^^^^^^^^^^^^^^^
```

5.3 File handling

The basic output functions, namely `.`, `..`, and `.s` have already been introduced in section 5.1 – output generated by those functions is directed to the standard output channel[10] normally, but there are a number of functions that allow reading from and writing to files as well.

[10] `stdout` for short. `stdout` has the file number 1.

5.3.1 close

This function closes a file which has been previously opened using the open function (see section 5.3.5). close expects the number of the file to be closed on the topmost stack element.

5.3.2 eof

The eof function expects the number of a previously opened file (see section 5.3.5) on the TOS and returns the value 1 if a following read operation (see section 5.3.6) would result in an error due to reaching the end of the file.[11] Otherwise 0 is returned.

5.3.3 fin

This function expects a valid file number in the topmost stack element and redirects the standard input channel[12] to that particular file.

5.3.4 fout

The fout function redirects stdout to the file specified by the file number in the topmost stack element.

5.3.5 open

This function opens a file for read, write, or append. open expects two scalar values on the two topmost stack elements: one describing the mode of operation for which the file will be opened (possible values are <, >, and >> for read, write, and append), the other one containing the name of the file.

open removes these two scalars from the stack and returns the file number of the file just opened. The following example shows how to create a file named test.dat if it does not exist already and write the string "Hello world!\n" to it:

```
1 lang5> '> 'test.dat open dup fout "Hello world!\n" . close
```

5.3.6 read

read reads a record from a stream[13] and pushes it onto the stack. Thus reading the record which was written to the file test.dat just before can be accomplished like this:

```
1 lang5> '< 'test.dat open dup fin read . close
2 Hello world!
```

[11] *EOF* for short.
[12] stdin for short. stdin has the file number 0.
[13] Be default this is stdin which can be changed by fin, see section 5.3.3.

5.3.7 STDIN, STDOUT, STDOUT

These three words push the respective file numbers of the three standard IO-streams `stdin`, `stdout` and `stderr`[14] onto the stack.

5.3.8 unlink

The `unlink` function actually deletes a file which name is specified in a scalar on the TOS. The following example shows how to delete the file `test.dat` created in the examples above:

```
lang5> 'test.dat unlink
```

5.3.9 slurp

If is often quite handy to read the content of a file in a single step. This is accomplished by the word `slurp` which expects the name of the file in the TOS element and returns an one-dimensional array containing the records of the file. Assume that there is a file `grades.dat` containing the following lines of data:

```
Name;grade
Student a;2
Student b;5
Student c;3
Student d;1
Student e;1
```

Reading this file's contents into an array can be accomplished like this:

```
lang5> 'grades.dat slurp .
[ Name;grade Student a;2 Student b;5 Student c;3 Student d;1
  Student e;1 ]
```

Exercise 17:

Define a word that expects the name of a file containing grades in the format shown above which reads in the file and returns the mean grade of all students.

5.4 Mathematical, logical and comparison operations

Lang5 supports quite a lot of mathematical, logical and comparison operators and words which are described in more detail in the following. In most cases these opera-

[14] The standard error stream.

tors and words are rather self-explanatory.

5.4.1 +, -, *, /

These are the traditional binary arithmetic operation plus, minus[15] Plus and minus have 0 as their neutral element while multiplication and division use 1 as neutral element. These four operators are already overloaded to work transparently on complex numbers.[16] In addition to this, * has been overloaded to multiply matrices by vectors or matrices.[17]

5.4.2 %, **

Binary modulus and power operators. Both have 1 as their neutral element.

5.4.3 &, |, ∧

Bit-wise and, or, and exclusive or operators.

5.4.4 ==, !=, >, <, >=, <=

Binary numerical comparison operators. == and != are overloaded to work on complex numbers[18] as well as on polar coordinates.[19]

Exercise 18:
Define a word `largest_only` that expects two one-dimensional arrays `a` and `b` on the top of the stack and returns an one-dimensional array that only contains those elements from `a` that are bigger than the corresponding elements from `b`. Test it with two arrays `[1 2 3]` and `[0 2 2]`. The result should be the array `[1 3]`.

5.4.5 ===

The binary operator === is an "exactly equals" operator. Applied to `undef` and 0 it will return 0 and not 1 as a simple == would do.

5.4.6 eq, ne, gt, lt, ge, le

These are the string-comparison equivalents to the numerical comparison operators described in section 5.4.4.

[15] Since this is a binary operator, it can not be used to change the sign of a value. To accomplish this the word neg (see section 5.4.34) is used.
[16] These are dressed with `(c)`.
[17] Matrices and vectors are dressed with `(m)` and `(v)` respectively.
[18] Dressed with `(c)`.
[19] Dressed with `(p)`.

5.4.7 eql

The binary string comparison operator `eql` is equivalent to === (see section 5.4.5), it will not treat empty strings and `undef` as being equal.

5.4.8 <=>, cmp

Generalized numerical and alphanumerical comparison operators. a b <=> yields -1 if a is less than b, 0 if both are equal and +1 if a is greater than b. `cmp` behaves accordingly but the actual comparison is performed in lexicographic mode.

5.4.9 ||, &&

Binary logical or and and operators.

5.4.10 !

This unary word implements the factorial function.

5.4.11 ?

This so-called unary "dice" operator represent a simple pseudo-random generator. The value *n*contained in the topmost stack element is used as the upper limit for the pseudo-random number p to be returned which will always satisfy $0 \leq p < n$.

5.4.12 atan2

The binary operator `atan2` returns the arc tangent function of the two topmost stack element.

5.4.13 abs

The unary `abs` word returns the absolute value of a numerical value. In case of a complex number, dressed with `(c)`, it will return $\sqrt{\mathtt{re}^2 + \mathtt{im}^2}$.

5.4.14 amean

This word returns the arithmetic mean of the element in a one-dimensional vector.

Exercise 19:
Implement your own version of this function and call it `myamean`.

5.4.15 and

Logical and operator.

5.4.16 cmean

This word returns the cubic mean of the elements of a one-dimensional vector.

5.4.17 complex

The unary word `complex` converts a value represented by polar coordinates (dressed with `(p)`) into a complex number (dressed with `(c)`).

5.4.18 cos

Calls the cosine function.

5.4.19 defined

Returns 1 if a value is not `undef` – if an element is not defined, the value `undef` is returned:

```
lang5> [1 undef 2] defined 1 == .
[    1     0     1  ]
```

5.4.20 distinct

This unary word removes all elements from a set, a one-dimensional vector dressed with `(s)`, which occur more than once:

```
lang5> [3 1 4 1 5 9 2 6 5 3 5](s) distinct .
[    1     2     3     4     5     6     9  ](s)
```

5.4.21 e

Places an approximation for Euler's constant onto the stack.

5.4.22 eps

Places a constant ε onto the stack. This value should be used when comparing floating point numbers together with `abs` instead of a simple `==`:[20]

```
lang5> 1 3 / .3333333333333 == .
0
lang5> 1 3 / .3333333333333 - abs eps < .
1
```

[20] Comparing floating point values without the use of such an ε value is never a good idea since such values suffer from various problems which make them only a mediocre substitute for real numbers from $\mathbb{R}$.

5.4.23 exp

This unary operator returns the exponential function of the argument found in the topmost stack element.

5.4.24 gcd

This word expects two integer numbers in the two topmost stack elements and returns their greatest common divisor.

5.4.25 gmean

The word gmean returns the geometric mean of the elements of an one-dimensional vector which is expected in the TOS.

5.4.26 hmean

Returns the harmonic mean of the elements of an one-dimensional array on the TOS.

5.4.27 hoelder

Computes the generalized mean, the so-called Hoelder-mean of the elements of an one-dimensional array expected on the top of the stack.

5.4.28 im

Returns the imaginary part of a complex number (dressed with (c)):

```
lang5> [1 2](c) im .
2
```

5.4.29 int

This unary operator returns the integer part of a value:

```
lang5> 1 3 / int .
0
lang5> 10 [2 3 4 5] / int .
[    5     3     2     2  ]
```

5.4.30 intersect

This word returns the intersection of two sets (arrays dressed with (s)). Duplicate elements will be removed from the result set!

5.4.31 max

The binary operator max returns the maximum of two values:

```
lang5> 2 3 max .
3
lang5> [1 2 3 4 5 6 7] [3 1 4 5 9 2 6] max .
[ 3 2 4 5 9 6 7 ]
```

Exercise 20:
Define a word set_max that will work on a set (dressed with s) containing numeric values which will return the maximum value of the set's elements. (Hint: You can use spread and extract to accomplish this task.)

5.4.32 median

This word returns the median of a one-dimensional numeric vector:

```
lang5> [3 1 4 5 9 2 6] median .
4
```

5.4.33 min

This binary operator returns the minimum of two values.

5.4.34 neg

The unary operator neg changes the sign of a numerical value.

5.4.35 not

Unary not operator.

5.4.36 or

Binary logical or operator.

5.4.37 polar

This unary word converts a complex number, dressed with (c), into a polar value (dressed with (p)).

5.4.38 prime

This unary word tests if an integer value is a prime number and returns a positive integer if true, otherwise 0 is returned.

> **Exercise 21:**
> Define a word `prime_list` that expects an integer value on the TOS and returns a list of prime numbers up to that number using the `prime` word.

5.4.39 qmean

This word returns the quadratic mean of the elements of an one-dimensional array which is expected in the topmost stack element.

5.4.40 re

This unary word returns the real part of a complex number (dressed with `(c)`):

```
lang5> [1 2](c) re .
1
```

5.4.41 sin

Returns the sine of its argument.

5.4.42 sqrt

Returns the square root of a positive value.

> **Exercise 22:**
> Define a new word `better_sqrt` that can work with positive values as well as with negative ones (returning a complex number in the latter case).

5.4.43 subset

This word expects two sets (dressed with `(s)`) on the stack and tests if the one on the TOS is a subset of the set just below. If this is true, 1 will be pushed onto the stack, 0 otherwise.

```
lang5> [1 2 3](s) [1 3](s) subset .
1
lang5> [1 2 3](s) [3 1 2](s) subset .
1
lang5> [1 2 3](s) [2 4](s) subset .
0
```

5.4.44 tan

This unary word return the tangent of its argument.

5.4.45 union

This word expects two sets (dressed with (s)) on the stack and returns the union of these two sets. The resulting set does not contain any duplicates:

```
lang5> [1 2 3](s) [3 4 5](s) union .
[    1    2    3    4    5  ](s)
```

5.5 Control structures

Although the available control structures and elements have already been described in section 4.3.3 they are listed here again as a reference:

5.5.1 break

break terminates the execution of a loop which is the only way to get out of a do-loop construction. It also terminates the execution of a word when it is executed.

5.5.2 do-loop

The two keywords implement an endless loop. In contrast to Forth, Lang5 does not support an implicit loop variable like Forth's I. The only way to exit such a loop is through executing break.

5.5.3 if-else-then

These three keywords implement the traditional control structure as in other languages. Due to the stack-based nature of Lang5 if interprets the topmost element of the stack as a logical value to control the execution of the following block of code until an else, which is optional, or the terminating then is found.

5.6 Miscellaneous functions and words

5.6.1 execute

This function expects a string of Lang5-instructions or an one-dimensional array of strings of such instructions in the topmost stack element and executes these instructions:

```
lang5> clear "1 2 .s swap" execute .s
vvvvvvvvvvvvvvvvvvv Begin of stack listing vvvvvvvvvvvvvvvvvvv
Stack contents (TOS at bottom):
1
2
^^^^^^^^^^^^^^^^^^^^ End of stack listing ^^^^^^^^^^^^^^^^^^^^
vvvvvvvvvvvvvvvvvvv Begin of stack listing vvvvvvvvvvvvvvvvvvv
Stack contents (TOS at bottom):
2
1
^^^^^^^^^^^^^^^^^^^^ End of stack listing ^^^^^^^^^^^^^^^^^^^^
```

A typical application for this function is to eliminate loops with a fixed, previously known number of iterations by explicit loop-unrolling:

```
lang5> "'test ." 5 reshape execute
testtesttesttesttest
```

5.6.2 exit

`exit` terminates the Lang5-interpreter immediately.

5.6.3 gplot

The `gplot` word is a simple interface to the `gnuplot` package.[21] It expects a one-dimensional array on the stack and generates a plot based on its individual elements.

5.6.4 help

`help` can be used to obtain a basic explanation of built-in functions and the like:

```
lang5> '+ help
+: Basic binary operator +, neutral element: 0.
```

5.6.5 load

This function loads (and executes) a Lang5-program from a file. The following example shows how to execute the example program generating an Ulam-spiral which can be found in the `examples`-directory of the Lang5-directory tree:

[21] This package must be installed separately.

```
lang5> 'lang5/trunk/examples/ulam.5 load
73                                          79
        43                          47
71                          23
        41              7
                19                  2       11              53
                        5               3               29
67              17                          13
        37                                          31
                                61              59
```

5.6.6 panic

The panic function prints the content of the topmost stack element and immediately leaves the Lang5-interpreter loop. When running in batch-mode this will terminate the Lang5-interpreter instantaneously, while it will just return to the command prompt when running in interactive mode.

5.6.7 save

The save word saves the current work-space to a file which name is expected in the topmost stack element. save makes use of dump (see section 5.7.4) as it loops over a list of all variable and word names and appends their respective definition to the file being written. So saving and restoring the current work-space can be done as follows:[22]

```
$ lang5
loading mathlib.5: Const..Basics..Set..Stat..Cplx..P..LA..Graph..
                   Trig..NT..
loading stdlib.5:  Const..Misc..Stk..Struct..
lang5> : square dup * ;

lang5> 25 square .
625
lang5> 'my_workspace.5 save
Saving workspace to my_workspace.5: done
lang5> exit

$ lang5 -i my_workspace.5
loading mathlib.5: Const..Basics..Set..Stat..Cplx..P..LA..Graph..
                   Trig..NT..
loading stdlib.5:  Const..Misc..Stk..Struct..
loading my_workspace.5
```

[22] The -i qualifier forces the Lang5-interpreter to enter interactive mode which it would not do otherwise, if called with a file like this.

```
lang5> 25 square .
625
lang5>
```

5.6.8 system

The unary operator executes a string at the shell level of the operating system. This is a potentially very dangerous function as it might actually destroy the operating system if not used with care! The results of the command executed in the shell are placed into a one-dimensional array on the TOS:

```
lang5> 'date system .
[ Sun Apr  7 20:46:11 CEST 2013 ]
```

5.6.9 type

This function returns the type of the element in the topmost stack element without destroying this element:

```
lang5> 5 type [1 2 3] type [1 2](c) type .s
vvvvvvvvvvvvvvvvvvvv Begin of stack listing vvvvvvvvvvvvvvvvvvvv
Stack contents (TOS at bottom):
5
S
[    1    2    3  ]
A
[    1    2  ](c)
D
^^^^^^^^^^^^^^^^^^^^ End of stack listing ^^^^^^^^^^^^^^^^^^^^
```

Possible results are:

B: Binary operator

U: Unary operator

N: Niladic operator

F: Built in function

V: Variable

W: User defined word

S: Scalar

A: Array

D: Dressed data structure

5.6.10 ver

ver pushes the version number of the Lang5-interpreter onto the stack.

5.7 Variable and word handling

5.7.1 .ofw

This built-in function prints a list of all operators, functions and words known the Lang5-interpreter at the current moment. Each name of such an element is preceded by a single character denoting its type:

B: Binary operator

U: Unary operator

N: Niladic operator

F: Built in function

V: Variable

W: User defined word

5.7.2 .v

This word displays a list of all defined variables, including the special variables explained in section B.

```
lang5> .v
Variables:
__log_level'    --->    'ERROR'
__number_format'        --->    '%4s'
__terminal_width'       --->    '80'
```

5.7.3 del

This function expects the name of a variable or user-defined word in the topmost stack element and deletes this variable or word.

5.7.4 dump

This function converts a user-defined word into a textual representation that is pushed as a string onto the stack.[23]

[23] This function is the heart of explain.

5.7.5 eval

This unary operator expects the name of a variable on the top of the stack and returns its content.

5.7.6 explain

This word expects the name of a user-defined word or variable on the topmost stack element and prints this word's or variable's definition to `stdout`:

```
lang5> 'prime explain
: prime(*)
  type "S" ne
  if
    "prime: TOS is not scalar!\n" panic
  then
  dup 1 ==
  if
    drop 0
  then
  dup 4 <
  if
    break
  then
  dup sqrt 2 / int iota 1 + 2 * 1 + [    2  ]
  swap append % "&&" reduce
;
```

5.7.7 set

Sets (and defines, if necessary) a variable. The name of the variable is expected in the topmost stack element and the value that variable is to be set to is expected in the stack element just below.[24]

Caution: User-defined words and variables share the same name-space, so trying to define a variable with a name already used for a user-defined word will either in an error message if the interpreter is running in interactive mode or in an abort if the interpreter is running in batch mode!

5.7.8 vlist

Pushed a one-dimensional array containing the names of all variables onto the stack:

```
lang5> vlist .
[ __log_level __number_format __terminal_width ]
```

[24] See section 4.3.6.

5.7.9 wlist

Similar to vlist this pushes a list of the names of all user-defined words onto the stack:

```
lang5> wlist .
[ ! != * + - .s .v / 2dup == STDERR STDIN STDOUT abs amean append
  clear cmean complex distinct dreduce e eps explain extract gcd
  gmean gplot hmean hoelder im intersect max median min ndrop neg
  pi pick polar prime qmean re roll rot save slurp subset tan union ]
```

6 Programming examples

6.1 Fibonacci numbers

The elements of the sequence defined by

$$f(0) = (1) = 1 \text{ and}$$
$$f(i) = f(i-1) + f(i-2)\ \forall i > 1$$

are called Fibonacci *numbers* in honor of Leonardi Pisani who discovered this sequence among many other (more important and influential) things.[1] Generating this sequence with Lang5 can be done like this[2]:

```
: fib           # Define a word named fib
    dup 2 <     # Handle first
    if          # two sequence
        drop 1  # elements.
    else        # All other
        dup     # elements end
        1 - fib # here.
        swap 2 - fib
        +
    then
;

0 do # Make a loop running from 0 to 10
    dup fib .  1 +
    dup 10 > if break then
loop
```

As straightforward as this solution is, it makes no use of any Lang5 features extending a traditional `Forth` interpreter. Defining a unary word `fib(*)` the explicit loop can be replaced by applying this word to a vector with unit stride as the following program shows:

```
: fib(*)
  dup 2 < if
```

[1] See [Lüneburg 1993].

[2] Remember that, as in Perl, there is more than one way to do it!

```
    drop 1 break
  then
  dup 1 - fib
  swap 2 - fib +
;

10 iota fib .
```

6.2 Throwing dice

More of the array language features of Lang5 are employed in the following example in which the arithmetic mean of the outcomes of throwing a six sided dice $n \in \mathbb{N}$ times is computed. Whereas non-array languages would require a loop to perform the repeated throwing of the dice this is accomplished in Lang5 by first creating a vector containing n times the value 6 representing the number of sides of the dice. Such a vector containing 100 elements can be created like this: `6 100 reshape` Applying the unary `?` operator to this vector will consume this vector and generate a new vector containing 100 pseudo-random numbers $0 \geq r < 6$. Since a dice always yields a natural number between 1 and 6 as a result, the unary `int` operator will be applied to this vector, truncating every floating point vector element. Adding one finally yields the desired vector containing 100 elements.

In the last step, the elements of this vector are summed by `'+ reduce` and then divided by the number of elements to return the desired arithmetic mean. The following listing shows a complete Lang5 program implementing this examples:

```
: throw_dice
  # Make a vector of the form [6 6 6 ... 6].
  6 over reshape

  # Throw dice n times, retain integer part and make sure
  # the results are between 1 and 6.
  ? int 1 +

  # Sum over all results and divide by the number of values.
  '+ reduce swap /
;

100 throw_dice .
```

6.3 Cosine approximation

Applying the very same principles, a simple cosine approximation using the well known MacLaurin series

$$\cos x \approx \sum_{i=0}^{n} (-1)^i \frac{x^{2i}}{(2i)!}$$

can be written in Lang5 without any explicit loops at all as the following example shows:

```
#
# Approximation of the cosine function using a MacLaurin
# series of 11 terms. The argument is expected on the TOS.
#
: mc_cos
    # Save x and the number of MacLaurin terms for future use.
    'x set 9 'terms set

    # Generate a vector containing x ** (2 * i).
    terms iota dup defined x * swap 2 * dup 'v2i set **

    # Generate a vector containing (2 * i)! and divide the
    # previous vector.
    v2i ! /

    # Generate a vector of the form [1 -1 1 -1 1 ...].
    terms iota 1 + 2 % 2 * 1 -

    # Multiply both vectors and compute the sum of the
    # result's elements.
    * '+ reduce
;

3.14159265 mc_cos .
```

6.4 List of primes

A typical way to generate a list of primes in an array language like APL has already been shown in section 1.2:

$$(\sim E \in E \circ . \times E)/E \leftarrow 1 \downarrow \iota E \leftarrow 100$$

This APL expression generates a vector with unit stride, starting with 2 and uses this as the base for creating a matrix by applying an outer product operation. This matrix

contains no prime numbers at all and can thus be used to select all values from a copy of this vector which are not contained in the matrix thus yielding a list of prime numbers. Its Lang5-implementation looks like this:

```
: prime_list
  1 - iota 2 +   # Generate a vector [2 .. TOS]
  dup dup dup    # Make sure there are four identical vectors.
  '* outer       # Outer product of the top two vectors.
  swap in        # Generate a selection vector based on vector
                 # and matrix.
  not            # Invert the elements of this vector
  select         # Use this vector to select elements from
                 # the vector [2 .. TOS].
;

do
  "Please enter a number between 2 and 100: " .
  read
  dup 2 < if
    "\tToo small!\n" . drop
  else
    dup 100 > if
      "\tToo large!\n" . drop
    else
      break
    then
  then
loop

prime_list .
```

Exercise 23:

Implement a unary word called `myprime` that returns 0 for a non-prime argument and 1 otherwise. This word should rely on trial divisions using the modulus operation without any explicit loop or conditional. To test if a given $n \in \mathbb{N}$ is prime, a vector containing $\sqrt{n} - 1$ elements n is generated in a first step. Then a vector with divisors is generated and both vectors are processed by %. If there is any value that divides n without remainder, n is not prime.

6.5 Printing a sine curve

Generating simple ASCII plots of functions is easy in an array language like Lang5. The basic idea is to mimic a strip recorder. Each line printed corresponds to one x-coordinate. The y-coordinate of a point to be printed is then set by generating a string of as many spaces as specified by the value of the y-coordinate.

```
# Generate a string consisting of n (from TOS) "-" and terminated
# by CR/LF.
: print_dot(*) " " 1 compress swap reshape "*\n" append "" join . ;

# Create a vector containing the width of the bargraph to be
# printed.
21 iota 10 / 3.14159265 * sin 20 * 25 + int

print_dot # Apply the word print_bar elementwise to this vector.
```

The output generated by this program has been shown in section 3.3.

6.6 Sorting external data

This example is just a variation of exercise number 17:

```
a ; 7
b ; 1
c ; 3
```

```
: get_upper(*) '; split expand drop swap drop ;
: get_lower(*) '; split expand drop drop ;

'sort.data slurp dup
get_lower swap get_upper
grade swap drop subscript
.
```

6.7 Matrix-vector-multiplication

Since the basic built-in multiplication operator * performs an element-wise multiplication of two data-structures, it must be explicitly overloaded to perform a matrix-vector-multiplication.

The definition : *(m,v) ...; overloads the multiplication operator for matrix-vector-operations. First a unary local word inner+ which computes the sum of the elements of a vector is defined before the dress codes of the arguments for this multiplication operator are stripped.

The multiplication performed in the following step is the basic multiplication which will work element-wise on the elements of the two- and the one-dimensional data-structure. The resulting two-dimensional matrix is then reduced to a vector by applying inner+. Dressing it as a vector completes the operation.

```
# Multiplication word:
: *(m,v)
  # Calculate the inner sum of a vector:
  : inner+(*) '+ reduce ;

  # Get rid of the dress codes:
  strip swap strip swap

  * 'inner+ apply
  'v dress
;

# Create a 3-by-3 matrix and a three-element vector:
9 iota 1 + [3 3]reshape 'm dress
3 iota 10 + 'v dress

# Multiply the matrix with the vector:
* .
```

6.8 Sum of cubes

[Adams et al. 2009][p. 41] contains a short FORTRAN example program which generates a list of all natural numbers between 1 and 999 which are equal to the sum of the cubes of their respective digits. The approach taken in this program is pretty straightforward: Three nested loops generate the three digits of the numbers to be tested. The number made up from these digits is then compared to the sum of the digit cubes and printed if both are equal.

```
program sum_of_cubes
! This program prints all 3-digit-numbers that
! equal the sum of the cubes of their digits.
implicit none
integer :: H, T, U
do H = 1, 9
  do T = 0, 9
    do U = 0, 9
      if (100*H + 10*T + U == H**3 + T**3 + U**3) &
        print "(3I1)", H, T, U
    end do
  end do
end do
```

Using an array language the same problem can be solved much more elegantly:

```
999 iota 1 +
```

generates an array containing all numbers to be tested. After creating a couple of copies of this vector, the unary word `cube_sum` is applied to this array. This word splits a number on an empty string which results in an array containing the individual digits of this number.[3] These digits are then taken to their third power and summed using `reduce`. The array resulting from applying `cube_sum` to the original value array is then compared with one of the copies for equality yielding a binary selection array. Using `select` only those elements which are equal to their digit-cube-sum are selected and printed.

```
# Print all natural numbers < 1000 which are equal to the sum
# of the cubes of their respective digits:

: cube_sum(*) "" split 3 ** '+ reduce ;
999 iota 1 + dup dup cube_sum == select .
```

6.9 Perfect numbers

A so called *perfect number* is a natural number which is equal to the sum of its positive divisors excluding the number itself. Using

```
% not
```

in the unary word `p` the divisors of a given natural number are identified by creating an array containing 1 at the position of every divisor and 0 in all other places. This array is then used to generate a list of the divisors by `select` which is then summed and compared to the value being tested.

```
: p(*)
  dup dup 1 - iota 1 + dup rot swap
  % not select '+ reduce ==
;
500 iota 1 + dup p select .
```

6.10 Mandelbrot set

The so called Mandelbrot *set* which was discovered by Benoit Mandelbrot is the archetypal fractal shape. It is generated by successive application of the iteration

$$z_{n+1} = z_n^2 + c \tag{6.1}$$

[3] It should be noted that a number is automatically treated as a string in this context.

to the points of a grid on the complex number plane where c represents the individual grid points. It can be shown that the absolute value of the elements of the sequence generated by this iteration formula for a given c is unbounded if it exceeds 2.

Normally the values c to be considered are generated by two nested loops covering the area of the complex plane in question. Then (6.1) is applied to each c in a third loop until it either diverges or until a maximum number of iterations has been reached. The number of iterations this last loop performed until it terminates is then used to select a color denoting the behavior of this particular c.

It is noteworthy that generating a `Mandelbrot` set can be done without any explicit loops in an array language. The program shown in the following first defines three word: `d2c` takes two scalar values and creates a complex number by compressing and dressing the resulting array. `iterate` is a unary word expecting a complex number. This word performs the actual iterations without any loop at all: The basic iteration step

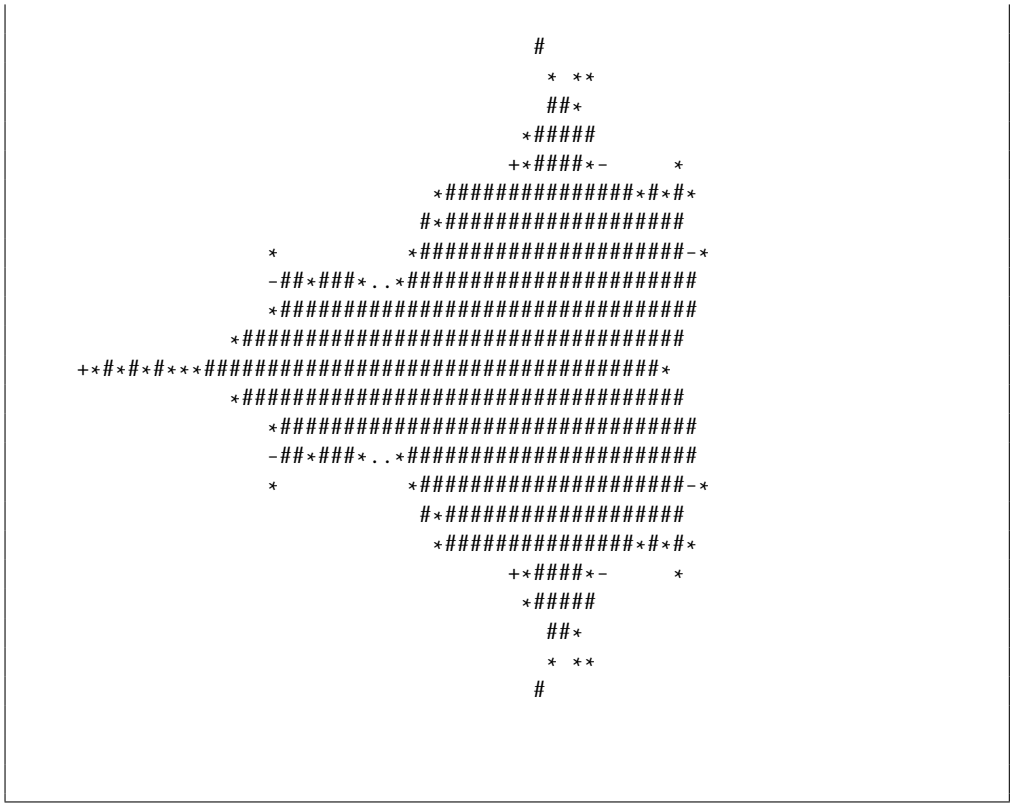
```
dup * over +
```

is defined as a string which is then repeatedly stored into an array using `reshape`. This array is then executed as a Lang5-program.[4] The third word, `print_line` prints one single line of the `Mandelbrot` set: The value generated by applying (6.1) to every c is used to select one character out of a string containing a number of characters of decreasing blackness. The main program sets up a two-dimensional array of complex numbers on which `iterate` operates.

```
1  : d2c(*,*) 2 compress 'c dress ;        # Make a complex number.
2
3  : iterate(c) [0 0](c) "dup * over +" steps reshape execute ;
4
5  : print_line(*) "#*+-. " "" split swap subscript "" join . "\n" . ;
6
7  75 iota 45 - 20 /                       # x coordinates
8  29 iota 14 - 10 /                       # y cordinates
9  'd2c outer                              # Make complex matrix.
10
11 10 'steps set                           # How many iterations?
12
13 iterate abs int 5 min 'print_line apply # Compute & print
```

```
1  $ lang5 apple.5
2  loading mathlib.5: Const..Basics..Set..Stat..Cplx..P..LA..Graph..Trig..NT..
3  loading stdlib.5:  Const..Misc..Stk..Struct..
4  loading apple.5
5
6
```

[4] Effectively this closely resembles the process of *loop unrolling* performed by optimizers in compilers.



6.11 Game of Life

In 1970, the British mathematician John Horton Conway developed a two-dimensional cellular automaton based on four simple rules which should become the icon of a decade. This automaton consists of cells on a toroidal surface which are connected to their eight direct neighbors each. A cell is in one of two states at every moment: *alive* or *dead* as determined by this set of rules:

1. A cell being alive dies if it has less than two neighbor cells being alive.
2. It also dies if there are more than three neighbor cells being alive.
3. A dead cell will change its state to alive when it has exactly three neighbor cells which are alive.
4. A living cell continues to live if it has two or three living neighbor cells.

The main program shown below makes use of the loop unrolling trick shown in section 6.10 by creating an instruction array containing elements of the form `print_field`

`iterate` and executing these instructions. The word `print_field` prints the field on which the cells live while `iterate` performs the actual iteration of the cellular automaton.

The number of neighbor cells being alive is determined by a trick: The current state of the automaton which is represented by a two-dimensional array is copied eight times. Each of these copies is then rotated by ± 1 horizontally, vertically and diagonally. The state of a cell is represented by the values 0 and 1 representing the states *dead* and *alive* respectively. Summing these eight matrices yields the number of neighbor cells being alive. This value is then used to determine the new state of every cell by applying the locally defined binary word `rule` which implements the rule set shown above.

```
#
# This is a 5-implementation of Conway's Game-of-Life.
#
#  The idea is to create eight matrices, based on the Game-of-Life matrix,
# where a 1 denotes a living cell while a 0 denotes a dead cell. These eight
# matrices are the result of eight matrix rotations (left, right, up, down,
# upper left, upper right, lower left, lower right). These eight matrices are
# then summed to determine the number of neighbours of each cell. After that
# the standard Game-of-Life-rules are applied to the original matrix and the
# neighbour sum matrix to determine the new population.
#
: print_field # Pretty print the field of cells with a frame.
  : print_line(*) [" " "*"] swap subscript "" join '! . . '! . "\n" . ;

  dup shape expand drop swap drop 2 + '- swap reshape "" join dup . "\n" .
  swap 'print_line apply drop . "\n" .
;

: iterate # Perform one Game-of-Life-iteration
  : rule(*,*) swap if dup 2 >= swap 3 <= && else 3 == then ;

  # Rotate the matrix in all eight directions and sum these eight matrices:
  dup [1 0]   rotate swap
  dup [-1 0]  rotate swap
  dup [0 1]   rotate swap
  dup [0 -1]  rotate swap
  dup [1 1]   rotate swap
  dup [-1 1]  rotate swap
  dup [1 -1]  rotate swap
  dup [-1 -1] rotate swap

  9 -1 _roll + + + + + + + rule
;

: create_matrix(*) "" split " " ne ;

# Setup the start matrix - in this case it only contains a glider:
```

```
[
  "                  "
  "                  "
  "                  "
  "          *       "
  "            *     "
  "         ***      "
  "                  "
  "                  "
  "                  "
]

# Perform 100 iterations:
create_matrix "print_field iterate" 100 reshape execute
```

The start configuration defined in the program shown above is the so called *glider*. This cell configuration exhibits a repetitive pattern and slowly moves through the living space of the automaton.[5]

```
$ lang5 gol.5
loading mathlib.5: Const..Basics..Set..Stat..Cplx..P..LA..Graph..Trig..NT..
loading stdlib.5:  Const..Misc..Stk..Struct..
loading gol.5
------------------
!                  !
!                  !
!                  !
!         *        !
!           *      !
!         ***      !
!                  !
!                  !
!                  !
!                  !
------------------
------------------
!                  !
!                  !
!                  !
!                  !
!        * *       !
!         **       !
!         *        !
!                  !
!                  !
!                  !
```

[5] It should be noted that the *Game-of-life* cellular automaton has been shown to be Turing complete, i.e. it is as powerful as a universal Turing machine from a computational point of view.

```
------------------
------------------
!                !
!                !
!                !
!                !
!          *     !
!        * *     !
!         **     !
!                !
!                !
!                !
------------------
------------------
!                !
!                !
!                !
!                !
!        *       !
!         **     !
!        **      !
!                !
!                !
!                !
------------------
```

6.12 Ulam spiral

The so called Ulam *spiral* is a simple yet interesting way to visualize the distribution of prime numbers. The idea is simple: Generate a rectangular spiral of natural numbers, starting with 2 in the middle. Then leave out the places of non-prime numbers so that only the prime numbers remain. The resulting structure shows intriguing patterns which are connected to polynomials generating rather long sequences of prime numbers.

The following program is a bit more tricky than the examples shown before. Figuring out its operation is left as an exercise to the reader.

```
: ulam_spiral
  : seq
    : zip(*,*) 2 compress " " join ;
    : subsubseq swap 2 2 compress reshape ;
    : subseq
      0 pick     [0 1]  subsubseq 1 pick     [1 0]  subsubseq
      2 pick 1 + [0 -1] subsubseq 3 pick 1 + [-1 0] subsubseq
      5 roll drop append append append
```

```
    ;

    dup 2 reshape 1 compress
    over iota 2 * 1 + "subseq append" 3 pick reshape zip execute
    over 2 * [0 1] subsubseq append '+ spread
  ;

  : print_line(*)
    : rpl(*) dup not if drop "" then ;
    rpl "\t" join . "\n" .
  ;

  seq swap 2 * 1 + 2 ** iota 1 + dup prime swap and swap scatter
  'print_line apply drop
;

4 ulam_spiral
```

The output of the program shown above is too small to show the patterns of mostly diagonal lines containing prime numbers, but it shows the basic form of such a Ulam spiral:

```
$ lang5 ulam.5
loading mathlib.5: Const..Basics..Set..Stat..Cplx..P..LA..Graph..Trig..
                   NT..
loading stdlib.5:  Const..Misc..Stk..Struct..
loading ulam.5
73                                          79
        43                          47
71                          23
        41              7
                19                  2       11              53
                        5           3               29
67              17                          13
        37                                          31
                            61              59
```

7 Interpreter anatomy

The following sections provide a short walk-through of the Lang5-interpreter's source code. This is by no means a thorough description of every implementation detail, it is intended to serve as a starting point for more thorough evaluation by the reader.

7.1 The wrapper `lang5`

Since the Lang5-interpreter itself is encapsulated in a Perl module `Lang5.pm`, the actual user-interface is implemented in the file `lang5` which can be found in the main directory of the interpreter directory tree. First of all, a Lang5-object is instantiated like this:

```
1  my $fip = Lang5->new(
2      log_level      => $opt{debug_level},
3      number_format  => $opt{format},
4      text_callback  => sub {
5          $line_count += tr/\n/\n/ for @_;
6          print $OUT @_;
7      },
8      libdir         => "$Bin/lib",
9      libautoload    => !$opt{nolibs},
10 );
```

All of the interpreter's functionality can now be accessed via `$fip`. Following this, a signal handler for catching `SIGINT` is registered and storing the current system time is stored.

The interpreter can operate in three modes which are partially mutual exclusive:

Evaluating a program specified on the command line: In this mode, a (short) program which is included in double quotes following the `-e` qualifier, is executed immediately. This is controlled by `$opt{evaluate}` which reflects `-e`.

Batch mode: If any source files are specified on the command line their contents will be read and executed by the interpreter. This is done by looping over the contents of `ARGV` which do no longer contain any qualifiers and other parameters since these are already removed by `GetOptions`.

Interactive mode: This mode is mutually exclusive with the `-e` option. In interactive mode, commands are read from the standard input using `readline`.

In either case a Lang5-program is executed by calling `execute` which expects a reference to the Lang5-object, a reference to an array containing the lines of source to be executed and a handle for output. It then performs the following actions:

```
$fip->add_source_line($_) # Build a program from the individual lines.
    for @$lines;
$fip->execute();          # Execute the program.
if ( $fip->error() ) {    # Handle any errors which occurred.
    ...
}
```

The remaining parts of `lang5` mostly deal with handling statistical data and printing a usage summary if the program has been called with invalid options.

The following sections are focused on the Lang5-interpreter itself which is contained in the file `perl_modules/Lang5.pm`.

7.2 Parsing

The first step for executing a Lang5-program is to build a nested data-structure representing the program in a form that makes it easy for the interpreter to perform the necessary actions.

The method `add_source_line` is called for every single line of source code. It skips empty lines and gets rid of invalid characters which are normally caused by sloppy typing such as holding the ALT key after typing a closing square bracket for to long so that a single space character will become an ALT-space. Other valid characters like single quotes, double quotes and quoted backslashes are escaped by replacing them by constants like `__CTSQ__`. This simplifies the following tasks of the parser.

Special treatment is necessary for word headers consisting of an initial colon and a word name with optional parentheses. The hash `%re` used to detect a word header contains some regular expressions which are used throughout the interpreter:

```
my %re = (
  float => qr/^([+-]?)(?=\d|\.\d)\d*(\.\d*)?([Ee]([+-]?\d+))?$/,
  whead => qr/\S+\{\[.+?\}/,
  strob => qr/\Qbless( do{\(my \E\$o = (’.*’)\Q)}, ’Lang5::String’ )/,
);
```

The regular expression stored in the entry `float` matches a floating point number, that stored in `whead` has been mentioned above and `strob` contains a regular expression which is used to detect `Lang5::String` objects.

Since keywords in Lang5-source code are mainly separated by white-space, it is necessary to protect strings which might contain white-spaces from further pro-

cessing. This is done using the function _secure_string. After removing any comments the line processed like this is finally appended to an array referenced by $self->{_line_buffer}.

To get an impression of the preprocessing operations performed by add_source_line the following short example program is considered:

```
: loop_example
  1 do dup . 1 + dup 5 > if break then loop
;

loop_example "Loop finished. Now print an array:" .
[[1 2] [3 4]] .
```

After processing every single line of this program with add_source_line the buffer $self->{_line_buffer} contains the following data:[1]

```
$1 = [
  ': loop_example',
  '  1 do dup . 1 + dup 5 > if break then loop ',
  ';',
  'loop_example  {4c6f6f702066696e69736865642e204e6f77207072696e742061
                  6e2061727261793a}  .',
  '[[1 2] [3 4]] .'
];
```

After all lines of a Lang5-program are processed like this, the method execute is called which in turn calls _parse_source. This function splits the lines contained in $self->{_line_buffer} on white-space and square brackets and takes care that square brackets are not removed during this operation. Applied to the example program above, the result of this call is this data-structure:

```
$1 = [
  ':',
  'loop_example',
  '1',
  'do',
  'dup',
  '.',
  '1',
  '+',
  'dup',
  '5',
  '>',
```

[1] Printed with Data::Dumper.

```
  'if',
  'break',
  'then',
  'loop',
  ';',
  'loop_example',
  bless( do{\(my $o = 'Loop finished. Now print an array:')},
            'Lang5::String' ),
  ',',
  '[',
  '[',
  '1',
  '2',
  ']',
  '[',
  '3',
  '4',
  ']',
  ']',
  '.'
];
```

Clearly the two-dimensional array to be printed at the end of the example program needs some further processing since the process of splitting the source into a stream of tokens just returned a sequence of square brackets and values which have to be assembled into an array for further processing. This is done in `_transmogrify_arrays`. Applied to the structure shown above it returns this:

```
$1 = [
  ':',
  'loop_example',
  '1',
...
  ',',
  [
    [
      '1',
      '2'
    ],
    [
      '3',
      '4'
    ]
  ],
  '.'
];
```

The array specified as `[[1 2][3 4]]` in the source code is now represented by a real two-dimensional array. The next step of the parser handles control structures like `if...else...then` and `do...loop`. This is done by calling `_if_do_structures` which transforms the data-structure in the line buffer into this:

```
$1 = [
  ':',
  'loop_example',
  '1',
  'do',
  [
    'dup',
    '.',
    '1',
    '+',
    'dup',
    '5',
    '>',
    'if',
    [
      'break'
    ]
  ],
  ';',
  'loop_example',
  bless( do{\(my $o = 'Loop finished. Now print an array:')},
            'Lang5::String' ),
  '.',
  [
    [
      '1',
      '2'
    ],
    [
      '3',
      '4'
    ]
  ],
  '.'
];
```

The program's structure is now represented by a nested data-structure which is then used by the central interpreter routine `_execute`.

7.3 _execute

_execute is the heart of the interpreter. It acts on a data-structure like that shown before and effectively executes a program by traversing this structure. It is implemented as a finite-state machine with the following states:

```
use constant {
    STATE_RUN                  => 0,
    STATE_START_WORD           => 1,
    STATE_EXPAND_WORD          => 2,
    STATE_SKIP_WORD_DEFINITION => 3,
    STATE_EXECUTE_IF           => 4,
    STATE_EXECUTE_ELSE         => 5,
    STATE_IF_COMPLETED         => 6,
    STATE_EXECUTE_DO           => 7,
    STATE_BREAK_EXECUTED       => 8,
};
```

Beginning with the initial state STATE_RUN, _execute loops over the elements of the array containing the preprocessed and parsed program. _execute calls itself recursively whenever a nested program part controlled by a conditional or a loop is encountered.

Exercise 24:

Draw a state diagram of _execute. Note that the sequence of the conditionals which control the state transitions in the source code is essential for correct operation (explain why).

7.4 Built-ins etc.

_execute relies on a nested data-structure quite similar to that shown in the simple RPN parser of section 2.1. This structure is contained in a hash %builtin and looks like this:

```
my %builtin = (

    ### niladic operators
    exit  => {
        desc => 'Leave the interpreter immediately.',
        type => 'niladic',
        code => sub { $_[0]->{_exit_called} = 1; },
    },
...
    ### unary operators
    '?' => {
        desc => 'Generate a pseudo random number.',
```

```
13          type => 'unary',
14          pop  => [qw/X/],
15          push => [qw/I/],
16          ntrl => 0,
17          code => sub { rand($_[1]); },
18      },
19  ...
20      ### binary operators
21      # direct mapping to perl operators
22      # with 0 as neutral element
23      ( map {
24          $_ => {
25              desc => "Basic binary operator $_, neutral element: 0.",
26              type => 'binary',
27              pop  => [qw/X X/],
28              push => [qw/S/],
29              ntrl => 0,
30              code => eval("sub { no warnings qw/numeric/;
31                                  \$_[2] $_ \$_[1] }"),
32          }
33      } qw(
34          + -
35      )),
36  ...
```

Every built-in operator/function is represented by an entry in this hash which references a nested hash containing the following keys:

desc: This entry contains a short help text describing the operation.[2]

type: Any language element is of a specific type which is stored in this entry. Possible values are: `niladic`, `unary`, `binary`, `function`, `variable`[3]

pop: This entry describes how many values are popped from the stack by an operation and the expected type of these values. Possible values are:

- **A:** Array
- **BO:** Binary operator
- **I:** Integer value
- **F:** Floating point value
- **PI:** Positive integer value
- **V:** A valid variable name
- **S:** Any scalar value

[2] See section 5.6.4.

[3] These types are normally referred to only by their first letter in the source code.

X: Any value

U: A user defined word

N: A name of a user defined word or variable

These values are used for automatic checks of the values before executing a built-in. Central element for this type checking is the dispatch table `%param_checks` which maps these type descriptors to check functions.

push: This describes what will be pushed back onto the stack.[4]

ntrl: Value of the neutral element which will be returned if one of the arguments of the operator/function is undefined.

code: The actual functionality of the built-in.

Since unary and binary operations are automatically applied in an element-wise way onto nested data-structures, the interpreter core `_execute` relies on two subroutines `_unary` and `_binary` which in turn call `unary` and `binary` from the Perl module `Array::DeepUtils` which contains the (sometimes rather complicated) array handling capabilities.

7.5 Stacks

The central data-structure of any Lang5-program is the stack onto which all operands are placed before they can be processed. Pushing a value onto the stack is straightforward as long as this value is a scalar. In this case

```
push \@\$stack, \$element + 0;
```

is executed.

Exercise 25:
Explain the rationale behind the `+ 0` operation in the statement above.

When an array is to be pushed onto the stack, things are a bit more complex. A simple

```
push \@\$stack, \$element
```

would push a reference to the array onto the stack. If this array was read from a variable this had the effect that any changed on the array on the stack would affect the value of the variable through the reference.[5] To avoid this problem any nested data-structure which is to be pushed onto the stack is first copied using `dclone` from `Array::DeepUtils`. `dclone` problem...

[4] This information is not currently used by the interpreter.

[5] In fact, this was a bug in the interpreter which was only detected and corrected recently (after more than two years of development).

7.6 Local stacks

Although all operators, functions and words could operate on a single central stack this would pose the risk of changing values on the central stack by side-effects. To avoid this risk, the Lang5-interpreter creates a local stack every time an operation is to be performed. In the case of a unary or binary operation, one or two values are fetched from the central stack and pushed onto a temporary stack on which the operation itself acts.

Whatever will be found in the top-most stack element of such a local stack after the execution of the operator or the like has finished will be pushed onto the central stack and the local stack will be destroyed.

Exercise 26:

Why isn't it necessary to use `dclone` when pushing the arguments onto the local stack for an unary or binary operator? Why does pushing a reference suffice and not cause unwanted side effects?

7.7 Questions

Exercise 27:

1. Follow the operation of the Lang5-interpreter by executing a simple program without loading the standard libraries[6] by specifying the `-n` qualifier. To trace the operation of the interpreter specify `-d TRACE` on the command line.
2. Analyze the structure which results from a user defined word. Therefore follow the program flow starting with the state `STATE_START_WORD` of the finite-state machine in `_execute`.
3. Explore how overloading user defined words actually works. Central to this functionality is the subroutine `_get_func`.

8 Extending Perl

All of the nice array-handling functionality of Lang5 are also available to pure-Perl by means of the Array::APX package which overloads some of the built-in operators of Perl and introduces some useful methods for dealing with nested data-structures. This module is in fact a wrapper around Array::DeepUtils.[1]

The following example shows how the element-wise product of two vectors can be computed using this module:

```
use strict;
use warnings;
use Array::APX qw(:all);

# Create two vectors [1 2 3] and [4 5 6]:
my $x = iota(3) + 1;
my $y = iota(3) + 3;

# Multiply both vectors and print the result:
print $x * $y;
```

A noteworthy feature of Array::APX is that some of Perl's standard operators are not only overloaded to operate on APX-objects but also act as reduce operator or as outer product. The reduce operator is represented by / as in APL:

```
use strict;
use warnings;
use Array::APX qw(:all);

# Create a vector [1 .. 100]:
my $x = iota(100) + 1;

my $adder = sub {$_[0] + $_[1]};
print 'The sum of all elements is ', $adder / $x, "\n";
```

Outer products are specified by enclosing a binary operator which in fact is a reference to a suitable subroutine in pipe symbols:

[1] Array::APX has been introduced at the YAPC::Europe 2012, see [Ulmann 2012]. The source of this module can be found at http://cpansearch.perl.org/src/VAXMAN/Array-APX-0.3/lib/Array/APX.pm.

```
use strict;
use warnings;
use Array::APX qw(:all);

my $x = iota(10) + 1;
my $m = sub {$_[0] * $_[1]};

print $x |$m| $x;
```

Implementing the algorithm for generating a list of primes shown in section 1.2 using Array::APX is quite straight-forward:

```
use strict;
use warnings;
use Array::APX qw(:all);

my $f = sub { $_[0] * $_[1] }; # We need an outer product
my $x;

print $x->select(!($x = iota(199) + 2)->in($x |$f| $x));
```

The Array::APX module and additional documentation can be found on CPAN.[2]

[2] http://search.cpan.org

A A simple arithmetic expression parser

```
/*
**  Simple parser for arithmetic integer expressions of the
** following form:
**
**       <expression> = <factor>
**                    | <factor> [+-/*] <factor>
**
**        <factor>    = (<expression>)
**                    | -(<expression>)
**                    | <constant>
**                    | -<constant>
**
**        <constant>  = [0-9]
**                    | <constant>[0-9]
**
*/

#include <stdio.h>
#include <string.h>
#include <ctype.h>

#define STRING_LENGTH 133

char *gbl$pos;     /* Pointer into the string to be parsed   */
int gbl$error = 0; /* Error flag - incremented by each error */

/* Get rid of an optional '\n' character at the end of a string */
void chomp(char *string)
{
    int last = strlen(string) - 1;
    if (last >= 0 && string[last] == '\n')
        string[last] = (char) 0;
}

/* Parse a constant and return its integer value */
int constant()
```

```
{
    int result = 0;

    /* Loop over the digits of the constant and update the result */
    while (*gbl$pos && isdigit(*gbl$pos))
        result = result * 10 + (*gbl$pos++ - '0');

    return result;
}

/* Evaluate a factor in an expression */
int factor()
{
    int minus, result;

    /* Check for an optional minus sign, remember and get rid of it */
    if (minus = (*gbl$pos == '-')) gbl$pos++;

    /* Does the factor start with a parenthesis? */
    if (*gbl$pos == '(')
    {
        gbl$pos++; /* Skip the parenthesis */
        result = expression();
        if (gbl$error) return 0;

        if (*gbl$pos == ')')
            gbl$pos++;
    }
    else /* No parenthesis, just a constant to follow */
        result = constant();

    return minus ? -result : result;
}

int expression() /* Evaluate an expression */
{
    char operator;
    int result, expr;

    /* If an expression ends with an operator not followed by another
    ** factor we will end up here with gbl$pos pointing to the
    ** trailing null character of the string, which denotes an error.
    */
    if (!*gbl$pos)
    {
        printf("\texpression(): Empty string!\n");
        gbl$error++;
```

```
84          return 0;
85      }
86
87      /* An expression consists of at least one factor */
88      result = factor();
89      if (gbl$error) return 0;
90
91      /* If there are chars left, the next one must be an operator */
92      if (*gbl$pos)
93      {
94          operator = *gbl$pos++;
95
96          /* Calculate <expression> <operator> <expression> */
97          switch(operator)
98          {
99              case '+':
100                 result += expression();
101                 break;
102             case '-':
103                 result -= expression();
104                 break;
105             case '*':
106                 result *= expression();
107                 break;
108             case '/':
109                 result /= expression();
110                 break;
111         }
112     }
113
114     return result;
115 }
116
117 int main()
118 {
119     char string[STRING_LENGTH];
120     int result;
121
122     for(;;) /* Endless loop for user input and processing */
123     {
124         printf("Expression: ");
125
126         /* Read user input - leave the loop on EOF */
127         if (!fgets(string, sizeof(string), stdin)) break;
128
129         chomp(string);
130         gbl$pos = string; /* All functions rely on this global ptr */
```

```
131
132         result = expression();
133         if (gbl$error) /* Any errors so far? */
134         {
135             printf("\tmain(): Error evaluating \"%s\"\n", string);
136             gbl$error = 0; /* Reset error counter */
137         }
138         else
139             printf("\tResult = %d\n", result);
140     }
141
142     printf("Parser ended.\n");
143     return 0;
144 }
```

Exercise 28:

1. Extend the parser by adding binary operators for exponentiation (**), modulo (%) etc.
2. Add a unary operator ! to calculate the factorial of a number.
3. Change the parser so that it evaluates expressions from left to right.

B Special purpose variables

There are a couple of variables that control the operation of the Lang5-interpreter. These variables can be set during run time like other variables which will affect the interpreter's behavior accordingly. Currently the following special variables are supported:

__**log_level:** Normally this variable will be set to "ERROR" causing the interpreter to log error messages only. Possible values for this special variable are "TRACE", "DEBUG", "INFO", "WARN", "ERROR" and "FATAL". This variable should not be changed unless one really needs to get more information about the operation of the interpreter. Especially the settings "DEBUG" and "INFO" will generate substantial amounts of output – and are only useful for debugging the interpreter itself.

__**number_format:** The main function to print values, ., uses the format description found in this variable for printing scalar values. By default this variable is set to "%4s".

__**terminal_width:** Some functions like dump need to know about the size of the terminal being used which is specified using this special variable.

C The standard library

```
###
### stdlib.5, the standard library for 5.
###
### Internal variables are always prefixed by '_<word>' to avoid collisions
### between different words.
###

"loading stdlib.5:  " .

#==============================================================================
"Const.." .
: STDIN  0 ;
: STDOUT 1 ;
: STDERR 2 ;

#==============================================================================
"Misc.." . # Housekeeping words.

# Stack pretty printer (non-desctructive).
: .s
  depth 0 == if "Stack is empty!\n" . break then
  "vvvvvvvvvvvvvvvvvvvv Begin of stack listing vvvvvvvvvvvvvvvvvvvv\n" .
  "Stack contents (TOS at bottom):\n" .
  depth compress dup
  do
    length 0 == if break then
    0 extract .
  loop
  drop expand drop
  "\n^^^^^^^^^^^^^^^^^^^^ End of stack listing ^^^^^^^^^^^^^^^^^^^^\n" .
;

# Print a list of all variables known to the interpreter.
: .v
  "Variables:\n" .
  vlist                         # Get list of all variable names.
  do                            # Process the list.
   length 0 == if break then    # Anything left to print?
   0 extract                    # Get name to be printed.
   dup "\t--->\t"                # Prepare string to be printed.
```

```
   rot eval                      # Get value of variable.
   "\n" 4 compress "'" join .      # Make string and print line.
  loop drop
;

# explain a word.
: explain dump . ;

# Save the current workspace - expects destination filename on TOS.
: save
  : uxplain(*) explain ;

  depth 1 < if "save: Not enough elements on stack!\n" panic then
  type 'S ne if "save: scalar as filename expected!\n" . break then
  "Saving workspace to " over ": " 3 compress "" join .
  '> swap open '_save_destination set
  _save_destination fout
  wlist vlist append uxplain drop
  STDOUT fout
  _save_destination close '_save_destination del
  "done\n" .
;

# Read a file (the filename is expected to be in TOS) and create an array
# containing one record of this file per element.
: slurp
  depth 1 < if "slurp: Not enough elements on stack!\n" panic then
  type 'S ne if "slurtp: Scalar as filename expected!\n" panic then
  '< swap open '__slurp_fh set __slurp_fh fin
  []
  do
    eof if break then
    read append
  loop
  __slurp_fh close '__slurp_fh del
  STDIN fin
;

#=============================================================================
"Stk.." .

# Duplicate the two topmost elements on the stack.
: 2dup
  depth 2 < if "2dup: Not enough elements on stack!\n" panic then
  over over
;

# Remove all elements from stack.
: clear
```

```
  depth 0 > if depth compress drop then
;

# Generalized drop, TOS = depth.
: ndrop
  depth 1 < if "ndrop: Not enough elements on stack!\n" panic then
  type 'S ne if "ndrop: TOS is not scalar!\n" panic then
  compress drop
;

# Generalized over - it expects the position of the element to be picked
# at the TOS.
: pick
  depth 1 < if "pick: Not enough elements on stack!\n" panic then
  type 'S ne if "pick: TOS is not scalar!\n" panic then
  compress swap dup rot rot 1 compress append expand drop
;

# Generalized rot, TOS = depth.
: roll 1 _roll ;

# rotate the topmost 3 elements
: rot 3 1 _roll ;

#==============================================================================
"Struct.." .

# Append a scalar or a vector to another vector.
: append
  depth 2 < if "append: Not enough elements on stack!\n" panic then
  type 'S eq if 1 compress then
  type 'A ne if "append: Not an array!\n" panic then
  swap
  expand dup 2 + roll
  expand dup 2 + roll
  + compress
;

# Deep reduce - this word will reduce a nested structure to a single scalar
# regardless if its depth.
: dreduce
  over type 'A ne if "dreduce: TOS-1 is not an array!\n" panic then drop
  swap collapse swap reduce
;

# Extract an element from an array (subscript and remove combined) - TOS
# contains the element's number while TOS-1 contains the array.
: extract
  depth 2 < if "extract: Not enough elements on stack!\n" panic then
```

```
  type 'S ne if "extract: TOS is not scalar!\n" panic then
  over
  type 'A ne if "extract: TOS-1 is not an array!\n" panic then drop
  2dup 1 compress subscript rot rot remove swap expand drop
;

"\n" .
```

D The mathematical library

```
#
#  mathlib.5 contains various word definitions to deal with sets,
# statistics or to plot data.
#
#  This module makes use of the following dresses:
#
# (c)        Complex numbers
# (m)        Matrix
# (p)        Polar coordinates
# (s)   Set
# (v)        Vector

"loading mathlib.5: " .

#=============================================================================
"Const.." .

# Useful constants:
: pi  1 1 atan2 4 * ;
: e   1 exp ;
: eps 1.e-10 ; # This is used in comparison operators etc.

#=============================================================================
"Basics.." .

# Calculate the factorial.
: !(*) iota 1 + '* reduce ;

# Absolute value.
: abs(*)dup 0 < if neg then ;

# Maximum of the two topmost stack elements:
: max(*,*) 2dup - 0 < if swap then drop ;

# Minimum of the two topmost stack elements:
: min(*,*) 2dup - 0 > if swap then drop ;

#=============================================================================
"Set.." .
```

```
# distinct removes all elements from a set which occur more than once. As a
# side effect the resulting distinct set will be sorted.
: distinct(s)
  strip
  length 2 < if 's dress break then # Nothing to do for an empty set.
  grade subscript                   # Sort the array representing the set.
  dup dup
  [-1] remove [undef] swap append   # Right shift the sorted array.
  == not select                     # Determine the duplicates, negate the
                                    # resulting boolean vector and select
  's dress                          # the unique elements.
;

# Return the intersection of two sets.
# The result is a set without duplicates.
: intersect(s,s)
  distinct strip swap distinct strip over in select 's dress
;

# subset expects two sets on the stack and tests if the one on the TOS is
# a subset of the one below it. In this case a 1 is left on the TOS,
# otherwise 0 is returned.
: subset(s,s) strip swap strip swap in '&& reduce ;

# Return the union of two sets without duplicates.
: union(s,s) strip swap strip append 's dress distinct ;

#============================================================================
"Stat.." .

# Calculate arithmetic mean of the elements of a vector.
: amean
  depth 1 < if "amean: Stack is empty!\n" panic then
  type 'A ne if "amean: TOS is not an array!\n" panic then
  length 0 == if drop 0 break then
  dup '+ reduce swap length swap drop /
;

# Compute the cubic mean of the elements of a vector:
# ((x ** 3 + x ** 3 + ... + x     ** 3) / n) ** (1 / 3)
#    0        1              n - 1
: cmean
  depth 1 < if "cmean: Stack is empty!\n" panic then
  type 'A ne if "cmean: TOS is not an array!\n" panic then
  length 0 == if drop 0 break then
  3 hoelder
;

# Compute the geometric mean of the elements of a vector:
```

```
# (x  * x  * ... * x     ) ** (1 / n)
#   0    1          n - 1
: gmean
  depth 1 < if "gmean: Stack is empty!\n" panic then
  type 'A ne if "gmean: TOS is not an array!\n" panic then
  length 0 == if drop 0 break then
  length swap '* reduce swap 1 swap / **
;

# Compute the harmonic mean of the elements of a vector:
# n / (1 / x + 1 / x  + ... + 1 / x     )
#          0       1              n - 1
: hmean
  depth 1 < if "hmean: Stack is empty!\n" panic then
  type 'A ne if "hmean: TOS is not an array!\n" panic then
  length 0 == if drop 0 break then
  -1 hoelder
;

# Compute the hoelder mean of the elements of a vector:
# ((x ** k + x ** k + ... + x     ** k) / n) ** (1 / k)
#    0        1              n - 1
: hoelder
  depth 2 <
    if "hoelder: This word needs two words on the stack!\n" panic then
  type 'S ne if "hoelder: TOS is no a scalar!\n"    panic then
  swap type 'A ne if "hoelder: TOS-1 is not an array!\n" panic then swap
  over length swap drop 0 == if drop drop 0 break then
  swap length swap 2 pick ** '+ reduce swap / 1 rot / **
;

# Compute the median of the elements of a vector. The result is computed
# like this for a sorted vector:
#                  / x            for an odd number of elements
#                  !  (n + 1) / 2
#       x       = <
#        median    ! (x      + x         ) / 2 for an even number of elts
#                  \   n / 2    n / 2 + 1
#
: median
  depth 1 < if "median: Stack is empty!\n" panic then
  type 'A ne if "median: TOS is not an array!\n" panic then
  length 0 == if drop 0 break then
  grade subscript  # Sort the vector elements.
  length dup 2 %
  0 == if              # The vector has an even number of elements.
    2 / 2dup
    1 - 1 compress subscript expand drop
    rot rot
```

```
    1 compress subscript expand drop
    + 2 /
  else                 # Odd number of vector elements.
    1 + 2 / 1 - 1 compress subscript expand drop
  then
;

# Compute the quadratic mean of the elements of a vector:
# sqrt((x ** 2 + x ** 2 + ... + x     ** 2) / n)
#       0       1             n - 1
: qmean
  depth 1 < if "qmean: Stack is empty!\n" panic then
  type 'A ne if "qmean: TOS is not an array!\n" panic then
  length 0 == if drop 0 break then
  2 hoelder
;

#==============================================================================
"Cplx.." . # Functionality for dealing with complex numbers.

# Overload 'abs to return the absolute value of a complex number.
: abs(c)
  strip 2 ** '+ reduce sqrt
;

# Overload 'neg to perform the complement operation on a complex number.
: neg(c)
  strip [1 -1] * 'c dress
;

# Addition of two complex numbers.
: +(c,c)
  strip swap strip + 'c dress
;

# Subtraction of two complex numbers.
: -(c,c)
  strip swap strip swap - 'c dress
;

# Multiplication of two complex numbers.
: *(c,c)
  strip swap strip swap
  [0 1 0 1] subscript swap [0 1 1 0] subscript
  * expand drop
  + rot rot - swap
  2 compress 'c dress
;
```

```
# Division of two complex numbers.
: /(c,c)
  strip dup 2 ** '+ reduce
  rot strip rot
  [0 1 0 1] subscript swap [0 1 1 0] subscript
  * reverse expand drop
  + rot rot swap - 2 pick / rot rot swap / swap
  2 compress 'c dress
;

# Return the real part of a complex number.
: re(c)
  strip expand drop drop
;

# Return the imaginary part of a complex number.
: im(c)
  strip expand drop swap drop
;

# Convert a complex number to a polar coordinate tuple.
: polar(c)
  strip dup
  2 ** '+ reduce sqrt # This yields the radius.
  swap
  dup [0 0] == '&& reduce
    if "Can not convert zero cplx to polar!\n" panic then
  expand drop atan2   # This yields phi.
  2 compress 'p dress # Make a polar coordinate tuple.
;

# Convert a polar coordinate tuple to a complex number.
: complex(p)
  strip expand drop 2dup
  cos * rot rot sin *
  2 compress 'c dress
;

# Overload == for comparing complex numbers.
: ==(c,c)
  strip swap strip - abs eps < '&& reduce
;

# Overload != for comparing complex numbers.
: !=(c,c)
  strip swap strip - abs eps > '|| reduce
;

#=============================================================================
```

```
"P.." .

# Overload == for polar tuples.
: ==(p,p)
  strip swap strip - abs eps < '&& reduce
;

# Overload != for polar tuples.
: !=(p,p)
  strip swap strip - abs eps > '|| reduce
;

#=============================================================================
"LA.." .

# Overload * for matrix-vector-multiplication.
: *(m,v)
  # Calculate the inner sum of a vector:
  : inner+(*) '+ reduce ;

  swap strip shape rot strip swap reshape *
  'inner+ apply
  'v dress
;

: *(m,m) # Overload '* for matrix-matrix-multiplication
  # If we multiply an n*m matrix (columns*rows) by an m*n matrix using the
  # already existing matrix-vector-multiplication, we will need m copies of
  # the first matrix. First of all, let us determine m (as a side effect,
  # this second matrix looses its matrix dress which will be useful soon):
  strip shape [1] subscript expand drop

  # Now we compress the first matrix into an array and reshape it so that
  # this array will contain m copies of the original matrix:
  rot 1 compress swap reshape

  # Now swap the two arrays
  swap

  # To apply the already existing matrix-vector-multiplication to these two
  # arrays we have to transpose the topmost two dimensional array and
  # transform it into a one dimensional array of vectors:
  : a2v(*) 'v dress ;
  strip 1 transpose 'a2v apply

  # Now let us apply the existing matrix-vector-multiplication:
  *

  # Since this yields a one dimensional array of vectors, we have to strip
```

```
  # the array elements and dress the array itself as being a matrix:
  : v2a(v) strip ;
  'v2a apply

  # The result is still transposed, so perform another transposition and
  # dress it:
  1 transpose 'm dress
;

#=============================================================================
"Graph.." .

# array (the name reflects the fact that only the y-coordinates are fed
# into gnuplot).
# gplot plots a graph based on the elements of a single, one dimensional
: gplot
  # _gplot_write_data is a unary word to be used with apply to write the
  # data to be plotted to the gnuplot scratch data file.
  : _gplot_write_data(*) . ;

  depth 1 < if "gplot: Stack is empty!\n" panic then
  type 'A ne if "gplot: TOS is not an array!\n" panic then

  "_5_gplot.data" '__gplot_data_name set
  "_5_gplot.cmd"  '__gplot_cmd_name  set

  '> __gplot_data_name open '__gplot_fh set
  __gplot_fh fout
  '_gplot_write_data apply drop
  __gplot_fh close

  '> __gplot_cmd_name open '__gplot_fh set
  __gplot_fh fout
  "set key off\n" .
  "plot \"" __gplot_data_name "\" with lines\n" 3 compress "" join .
  __gplot_fh close

  STDOUT fout

  'gnuplot __gplot_cmd_name 2 compress " " join system drop
  __gplot_data_name unlink
  __gplot_cmd_name  unlink

  '__gplot_data_name del
  '__gplot_cmd_name  del
  '__gplot_fh        del
;

#=============================================================================
```

```
"Trig.." .

: tan dup sin swap cos / ;

#============================================================================
"NT.." .

# Places 1 on TOS if TOS was prime, 0 otherwise.
: prime(*)
  type 'S ne if "prime: TOS is not scalar!\n" panic then
  dup 1 == if drop 0 then
  dup 4 < if break then
  dup sqrt 2 / int iota 1 + 2 * 1 + [2] swap append % '&& reduce
;

# Return the gcd of two integers
: gcd(*,*)
  do
    2dup 0 > swap 0 > && not if break then
    2dup <= if
      over -
    else
      swap over - swap
    then
  loop
  dup 0 == if drop else swap drop then
;

"\n" .
```

E Solutions to selected exercises

Solution 2:

1. ```
   lang5> 1.25 2 / 2 ** pi * .
   1.22718463030851
   ```

2. ```
   lang5> 2 3 + 5 + 8 + 13 + .
   31
   lang5> 2 3 5 8 13 + + + + .
   31
   ```

   Using a more sophisticated function, this problem could be solved like this:

   ```
   lang5> [2 3 5 8 13] '+ reduce .
   31
   ```

Solution 3:

1. ```
   lang5> [[1 2 3] [4 5 6] [7 8 9]] [[7 6 2] [1 9 5] [3 8 4]] + .
   [
     [     8     8     5  ]
     [     5    14    11  ]
     [    10    16    13  ]
   ]
   ```

2. ```
   lang5> [[1 2 3] [4 5 6] [7 8 9]] 1 - .
   [
     [     0     1     2  ]
     [     3     4     5  ]
     [     6     7     8  ]
   ]
   ```

Solution 4:

```
lang5> 5 do dup . 1 - dup 1 < if break then loop
5
4
3
2
1
```

Solution 5:

`: square dup * ;` applied to a vector returns a vector of the same size with the squares of the elements of the original vector. This is due to the fact that although the word `square` itself is *not* applied in an element-wise fashion to the vector, the `dup` duplicates the vector as a whole and the multiplication operator `*` is then applied to the elements of these two identical vectors.

Solution 6:

```
lang5> : binary_print(*,*) . . "------\n" . ;
lang5> [1 2 3] 1 binary_print
1
1
------
1
2
------
1
3
------
```

Since `binary_print` is a binary word, the interpreter automatically expands the smaller data structure (the scalar in this case) to match the structure of the larger argument (the vector). So `binary_print` is implicitly applied to two vectors `[1 2 3]` and `[1 1 1]` respectively.

Solution 7:

```
lang5> : square dup * ;
lang5> [1 2 3 4] 'vector set
lang5> vector square .
[    1    4    9    16  ]
```

Solution 8:

```
: *(baz,baz)
  strip swap strip
  * -1 *
  'baz dress
;

[1 2 3](baz) [4 5 6](baz) * .
```

Solution 9:

```
lang5> : twodup over over ;
lang5> 1 2 twodup .s
vvvvvvvvvvvvvvvvvvvv Begin of stack listing vvvvvvvvvvvvvvvvvvvv
Stack contents (TOS at bottom):
1
2
1
2
^^^^^^^^^^^^^^^^^^^^ End of stack listing ^^^^^^^^^^^^^^^^^^^^
```

Solution 10:

```
lang5> : twodup 1 pick 1 pick ;
lang5> 1 2 twodup .s
vvvvvvvvvvvvvvvvvvvv Begin of stack listing vvvvvvvvvvvvvvvvvvvv
Stack contents (TOS at bottom):
1
2
1
2
^^^^^^^^^^^^^^^^^^^^ End of stack listing ^^^^^^^^^^^^^^^^^^^^
```

Solution 11:

```
: myroll 1 _roll ;
```

Solution 12:

```
lang5> : myswap over rot drop ;
lang5> 1 2 myswap . .
1
2
```

Solution 13:

```
lang5> : mydreduce swap collapse swap reduce ;
lang5> [[1 2] [3 4]] '+ mydreduce .
10
```

Solution 14:

```
lang5> : myfactorial iota 1 + '* reduce ;
lang5> 5 myfactorial .
120
```

Solution 15:

To create a n times n identity matrix, a vector looking like `[1 0 ...0]` containing n+1 elements is created. This vector is then `reshape`d into a n times n matrix. Since `reshape` reads the source data structure over and over again to fill the destination structure, this will eventually produce the identity matrix.

```
: identity_matrix
  dup
  1 + iota 0 ==
  swap dup 2 compress
  reshape
;

3 identity_matrix .
```

Solution 16:

```
lang5> 10 iota 2 * 1 + '+ spread .
[    1     4     9    16    25    36    49    64    81   100  ]
```

Solution 17:

```
: compute_mean_grade
  # Define a support word to get rid of the student names:
  : get_second_element(*) '; split 0 remove ;

  slurp                        # Read in file
  0 remove                     # Remove header line
  get_second_element collapse # Create a one-dimensional structure
  amean                        # Compute the mean value
;
```

```
'grades.dat compute_mean_grade .
```

Solution 18:

```
lang5> : largest_only over < select ;
lang5> [1 2 3] [0 2 2] largest_only .
[    1    3  ]
```

Solution 19:

```
lang5> : myamean length swap '+ reduce swap / ;
lang5> [1 2 3 4] myamean .
2.5
```

Solution 20:

```
lang5> : setmax 'max spread length 1 - extract ;
lang5> [3 1 4 1 5 9 2 6 5 3 5](s) setmax .
9
lang5>
```

Solution 21:

```
lang5> : prime_list 1 - iota 2 + dup prime select ;
lang5> 100 prime_list .
[    2    3    5    7   11   13   17   19   23   29   31
    37   41   43   47   53   59   61   67   71   73   79
    83   89   97  ]
```

Solution 22:

```
: better_sqrt
  dup abs sqrt
  swap 0 < if
    1 2 compress 'c dress
  then
;

2  better_sqrt .
-2 better_sqrt .
```

Another variant, yielding a complex number in every case might look like this:

```
: better_sqrt
  dup abs sqrt
  swap 0 < 2 compress 'c dress
;

2  better_sqrt .
-2 better_sqrt .
```

Solution 23:

```
: myprime(*)
  dup sqrt int 1 - dup rot swap reshape
  swap iota 2 +
  % 0 == '+ reduce
  0 ==
;

# Test the unary word myprime:
99 iota 2 + dup myprime select .
```

Solution 26:

The values which are pushed onto the local stack are automatically removed from the main stack. Thus, it is sufficient to work with references throughout since there is only one instance of these values and side-effects are impossible.

Bibliography

[Adams et al. 2009] Jeanne C. Adams, Walter S. Brainerd, Richard A. Hendrickson, Richard E. Maine, Jeanne T. Martin, Brian T. Smith, *The Fortran 2003 Handbook*, Springer, 2009

[Burks et al. 1954] Arthur W. Burks, Don W. Warren, Jesse B. Wright, "An Analysis of a Logical Machine Using Parenthesis-Free Notation" in *Mathematical Tables and Other Aids to Computation*, Vol. 8, No. 46, Apr., 1954, pp. 53–57

[Dr. Dobb's Journal] Dr. Dobb's Journal, *C Tools*, Markt & Technik Verlag, 1986

[Falkoff et al. 1964] A. D. Falkoff, K. E. Iverson, E. H. Sussenguth, "A formal description of SYSTEM/360", in *IBM Systems Journal*, Vol 3, No. 3, 1964, pp. 198–261

[Gilman et al. 1970] Leonard Gilman, Allen J. Rose, *APL\360 an interactive approach*, John Wiley & Sons, Inc., 1970

[Holub 1985] Allen Holub, "Wie Compiler arbeiten", in [Dr. Dobb's Journal][pp. 153–167]

[Iverson 1962] Kenneth E. Iverson, *A Programming Language*, J. Wiley & Sons, New York, 1962

[Iverson 1963] Kenneth E. Iverson, "Notation as a Tool of Thought", in [N. N. 1981][pp. 105–128]

[Janko 1980] Wolfgang H. Janko, *APL 1 – Eine Einführung in die Elemente der Sprache und des Systems*, Athenaeum Verlag, Königstein/Ts., 1980

[Lüneburg 1993] Heinz Lüneburg, *Leonardi Pisani Liber Abaci oder Lesevergnügen eines Mathematikers*, BI Wissenschaftsverlag, 1993

[McDonnel 1981] Eugene E. McDonnell, "Introduction", in [N. N. 1981][p. 11–14]

[N. N. 1981] N. N., *A Source Book in APL*, APL PRESS, Palo Alto, 1981

[Ousterhout 1998] John K. Ousterhout, "Scripting: Higher Level Programming for the 21st Century", in *IEEE Computer Magazine*, March 1998, `http://www.eve.cmu.edu/~ganger/712.fall02/papers/scripting-computer98.ps`, retrieved 01/14/2010

[Prechelt 2010] Lutz Prechelt, *Are Scripting Languages Any Good? A Validation of Perl, Python, Rexx and Tcl against C, C++, and Java*, `http://page.mi.fu-berlin.de/prechelt/Biblio/jccpprt2_advances2003.pdf`, retrieved 01/14/2010

[Ulmann 2012] Bernd Ulmann, "Array programming for mere mortals", in *Proceedings YAPC::Europe 2012*, pp. 37–41

Index