lang5
a new programming language for OpenVMS (and more)

Prof. Dr. Bernd Ulmann

27-OCT-2011

Hochschule fuer Oekonomie und Management, Frankfurt

Foreword

Foreword

This is. . .

- . . . not the first talk about the programming language **lang5**. Similar talks were given at the Connect symposium in 2010 etc. already.
- . . . no introduction into programming **lang5**. These slides try to give an impression of this language and to wet your appetite to learn more.

How is this talk structured?

- First **lang5** is introduced and its installation on OpenVMS is described.
- The following section shows some simple but typical programming examples.
- Following this some rather complex examples are shown but without a detailed analysis.

Introduction

First of all. . .

Do you miss VAX APL?

No: You obviously missed something!

Yes: Maybe **lang5** will make you happy again!

lang5 at a glance

What is **lang5**?

- **lang5** is a dynamic programming language (dynamic typing and memory management).
- **lang5** incorporates the main features of APL and Forth.
- The **lang5**-interpreter is written in Perl and easily portable.
- **lang5** runs out of the box on OpenVMS (...and UNIX, Windows...)
- **lang5** may be used to replace the much missed VAX APL interpreter.
- **lang5** is free software.
- **lang5**'s home is located at `http://lang5.sf.net`.
- **lang5**-development began in 2009 and the current release is V1.0.

lang5 at a glance

What is noteworthy about the language itself?

- It is a stack oriented language just like Forth but...
- ...the stack can hold scalars as well as nested data structures!
- The language can be **extended by itself** making use of so called *User Defined Words*.
- User defined words can act as unary or binary operators and are thus equivalent to builtin operators.
- Unary and binary operators are automatically applied in an element wise fashion on the elements of nested data structures which makes most of the explicit loops unnecessary that one is used to in traditional languages.
- Data structures can be *dressed* to denote their structural type (like *matrix*, *set* etc.) – in addition to that operators can be overloaded to act on such structures[1].

[1] New feature as of V1.0.

Installing lang5

How to install **lang5** on your OpenVMS system?

- Make sure you have a Perl interpreter running on your system.
- Download the **lang5** distribution kit from `https://sourceforge.net/projects/lang5/files/`. This kit contains the interpreter, a lot of examples and the complete documentation in PDF format.
- Unpack the distribution kit at a location suitable for your environment as the following example shows:

```
$ SET DEF DISK$SOFTWARE:[000000]
$ UNZIP DISK$SCRATCH:[SYSTEM]LANG5.ZIP
```

Installing lang5

- Define a foreign command for invoking **lang5** by adding a line like this to your `SYS$MANAGER:SYLOGIN.COM`:

  ```
  $ LANG5 :== PERL DISK$SOFTWARE:[LANG5]LANG5
  ```

- Now users can invoke the **lang5**-interpreter like that:

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

First Steps

First steps

At a first glance, **lang5** can be used like a stack based calculator (so owners of HP calculators have a slight advantage :-)) as the following example shows:

Simple scalar computations

scalar

```
trillian$ lang5
loading mathlib.5: Const..Basics..Set..Stat..Cplx..
                   P..LA..Graph..NT..
loading stdlib.5:  Const..Misc..Stk..Struct..
lang5> 3 2 1 + * .
9
lang5> exit
```

scalar

First steps

But **lang5** is much more powerful since it supports a wealth of array operations (too many to show here). Let us have a look at computing the sum of all integers running from 1 to 100.

In a traditional language like C this can be accomplished like this:

Example 1 – Gauss sum in C

gauss.c

```c
#include <stdio.h>

int main()
{
  int i, sum = 0;

  for (i = 1; i < 101; sum += i++);
  printf("Result: %d\n", sum);
}
```

gauss.c

First steps

The same problem can be solved in **lang5** much more easily:

Example 2 – calculating $\sum_{i=1}^{100} i$ in **lang5**:

$\sum_{i=1}^{100} i$ can be interpreted a bit differently as the sum of the elements of a vector with unit stride, running from 1 to 100.

Sum and factorial

```
100 iota 1 + '+ reduce .
```

Sum and factorial

Sum and factorial

How does this work? Let us have a look at the calculation of the sum:

```
100 iota 1 + '+ reduce .
```

- `100 iota` generates a vector `[0 1 2 ... 99]`.
- Adding 1 to this vector yields `[1 2 3 ... 100]`.
- `'+` pushes the operator "+" onto the stack.
- The `reduce`-function expects an operator on the top of the stack (*TOS* for short) and a vector below. It then applies this operator between all successive vector elements yielding `1 + 2 + 3 + ... + 100` in this case.
- The `.`-function prints the TOS.
- That's all – no explicit loops, nothing...

User defined words

This can be written better by introducing a user defined word (UDW) to calculate the Gauss sum for any number found on the top of the stack (so this UDW acts like a traditional function or subroutine):

Example 3 – simple user defined words

gauss.5

```
# Define a new word "gauss":
: gauss iota 1 + '+ reduce ;

# Use this new user defined word:
100 gauss .
```

gauss.5

This UDW gauss operates directly on the values found on the top of the stack (TOS for short). This means that this word is neither an unary nor a binary operator but acts more like a function (or subroutine).

Unary/binary operators

lang5 can do better – one particular strength is its builtin mechanism of applying unary and binary operators (including unary/binary UDWs) automatically to all elements of nested data structures without any need for explicit loops as the following example shows:

Example 4 – some builtin binary operators

Operators
```
lang5> [1 2 3] [4 5 6] + .
[    5     7     9   ]
lang5> [1 2 3] 2 ** .
[    1     4     9   ]
lang5>
```
Operators

Unary/binary UDWs

This mechanism can also be used in the case of user defined words which must be declared as unary or binary words respectively:

Example 5 – unary user defined words

gauss_factorial.5

```
# Define two unary words:
: gauss(*)     iota 1 + '+ reduce ;
: factorial(*) iota 1 + '* reduce ;

10 iota 1 + dup gauss . factorial .
```

gauss_factorial.5

```
trillian$ lang5 gauss_factorial_unary.5
loading mathlib.5: Const..Basics..Set..Stat..Cplx..P..LA..Graph..NT..
loading stdlib.5:  Const..Misc..Stk..Struct..
loading gauss_factorial_unary.5
[    1     3     6    10    15    21    28    36    45    55  ]
[    1     2     6    24   120   720  5040  40320  362880  3628800  ]
alberich$
```

Calculating Fibonacci numbers

lang5 allows recursive calls of words, too:

Example 6 – the ubiquituous Fibonacci series:

fibr_unary.5

```
: fib(*)
  dup 2 < if drop 1 break then
  dup 1 - fib swap 2 - fib +
;

10 iota fib .
```

fibr_unary.5

```
trillian$ lang5 fibr_unary.5
loading mathlib.5: Const..Basics..Set..Stat..Cplx..P..LA..Graph..NT..
loading stdlib.5:  Const..Misc..Stk..Struct..
loading fibr_unary.5
[    1     1     2     3     5     8    13    21    34    55  ]
```

Calculating Fibonacci numbers

How does this work?

- First of all, $\boxed{\texttt{10 iota fib .}}$ places a vector `[0 1 2 ... 9]` onto the TOS, calls the unary UDW `fib` and prints the resulting vector.
- `fib` is executed once for each element of the nested data structure it is applied since it is a unary UDW.
- The first step is to check if the value found on the TOS is less than 2 – in this case `fib` will just drop the value and return `1`.
- Otherwise `fib` calls itself twice with new arguments smaller by one and two respectively and returns the sum of the results of these calls.

Creating nested structures

lang5 offers a rich complement of functions and operators which help generating and restructuring nested data structures. The two main functions for this are shape and reshape:

Example 7 – shape and reshape

shape and reshape

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

shape and reshape

Throwing dice

Suppose you have to simulate throwing a six sided dice 100 times and calculate the arithmetic mean of the results you get:

Example 8 – throwing dice

throw_dice.5

```
: throw_dice
  6 over reshape
  ? int 1 +
  '+ reduce swap /
;

100 throw_dice .
```

throw_dice.5

```
lang5> 'throw_dice.5 load
3.47
```

Throwing dice

How does this work?

- $\boxed{\texttt{100 throw\_dice}}$ pushes 100 onto the stack and calls the word `throw_dice`.
- $\boxed{\texttt{6 over}}$ yields `100 6 100` on the stack.
- The `reshape`-function expects a dimension vector (or a scalar in the one-dimensional case) on the TOS and rearranges the object found below accordingly. In this case the result is a vector of the form `[6 6 6 ... 6]`.
- The unary `?`-operator generates a pseudo random number between 0 and the number found on the TOS. Since it is unary it is automatically applied to all elements of the vector we just created.
- $\boxed{\texttt{int 1 +}}$ gets rid of the fractional part of the resulting vector elements and makes sure they are between 1 and 6.
- $\boxed{\texttt{'+ reduce}}$ then computes the sum of the vector elements.
- $\boxed{\texttt{swap /}}$ swaps this sum and the 100 from the beginning and divides, yielding the arithmetic mean.

Sum of cubes

Recently I found the following Fortran-example program[2] which prints all numbers between 1 and 999 which are equal to the sum of the cubes of their digits:

sum_of_cubes.for

```fortran
program sum_of_cubes
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
end program sum_of_cubes
```

sum_of_cubes.for

Horrible, isn't it? Let's do it in **lang5**:

[2]Cf. [Adams et al. 09][p. 41]

Sum of cubes

The **lang5**-solution is a bit shorter (”Look Mom, no Loops!”):

Example 9 – sum of cubes

sum_of_cubes.5
```
: cube_sum(*)
  "" split 3 ** '+ reduce
;

999 iota 1 + dup dup cube_sum == select .
```
sum_of_cubes.5

```
lang5> 'sum_of_cubes.5 load
[    1   153   370   371   407  ]
```

Sum of cubes

How does this work?

- $\boxed{\texttt{cube\_sum(*)}}$ defines an unary word.
- This word pushes an empty string onto the stack and splits the element found below yielding a vector of the individual digits of the number which was found on the stack before.
- It then calculates the cubes of the vector elements by $\boxed{\texttt{3 **}}$.
- This vector of cubed digits is then summed using $\boxed{\texttt{'+ reduce}}$. The word thus transforms a number found on the TOS into the sum of its digit cubes.
- $\boxed{\texttt{999 iota 1 +}}$ yields `[1 2 3 ... 999]`.
- Since we need three of these vectors, it is duplicated twice.
- Then $\boxed{\texttt{cube\_sum}}$ is applied element wise to this vector.
- $\boxed{\texttt{==}}$ compares the result of this operation with the first copy of the original vector yielding something like `[1 0 0 ...]`.
- $\boxed{\texttt{select}}$ selects elements from a vector controlled by a corresponding boolean vector.

More complex examples

List of primes

The following program implements a form of the sieve of Eratosthenes which is quite popular in the APL community. The basic ideas for generating a list of primes between 2 and a given value `n` are these:

- Generate a vector `[1, 2, 3, ..., n]`.
- Drop the first vector element yielding `[2, 3, 4, ..., n]`.
- Compute the outer product of two such vectors yielding a matrix like this:

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

List of primes

- Obviously this matrix contains everything but prime numbers, so the next step is to determine which number contained in the original vector `[2, 3, ..., n]` is *not* contained in this matrix which can be done using the set operation `in`.
- The result of `in` is a vector with `n-1` elements each being `0` (its corresponding vector element was not found in matrix and is thus not prime) or `1`.
- After inverting this binary vector it can be used to select all prime numbers from the initial vector `[2, 3, ..., n]`.

All of this is accomplished by the following **lang5**-program:

List of primes

Example 10 – list of primes

primes.5

```
: prime_list
  1 - iota 2 + dup dup dup '* outer swap in not select
;

100 prime_list .
```

primes.5

This program yields the following output:

```
[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 ]
```

ASCII plotting

The following example shows how functions can be plotted on a dumb ASCII terminal by employing `reshape` to generate vectors consisting of blank characters which are then used for indentation.

To plot a simple sine curve the following code could be used:

Example 11 – ASCII plotting

sine_curve.5

```
: print_dot(*)
  " " 1 compress swap reshape "*\n" append "" join .
;

21 iota 10 / 3.14159265 * sin 20 * 25 + int
```

sine_curve.5

This yields the following output:

ASCII plotting

```
lang5> 'sine_curve.5 load
                         *
                               *
                                    *
                                         *
                                            *
                                             *
                                            *
                                         *
                                    *
                               *
                         *
                  *
             *
        *
     *
     *
     *
        *
             *
                  *
                        *
```

Matrix-Vector-Multiplication

Matrix-vector-multiplication is a good example to show operator overloading in **lang5**. As an example multiply

$$\begin{pmatrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 9\end{pmatrix} \text{ by } \begin{pmatrix}10\\11\\12\end{pmatrix}:$$

Example 12 – matrix vector multiplication

matrix_vector.5

```
lang5> 9 iota 1 + [3 3] reshape 'm dress
lang5> 3 iota 10 + 'v dress
lang5> .s
vvvvvvvvvvvvvvvvvvvv Begin of stack listing vvvvvvvvvvvvvvvvvvvv
Stack contents (TOS at bottom):
[
  [    1     2     3  ]
  [    4     5     6  ]
  [    7     8     9  ]
](m)
[   10    11    12  ](v)
^^^^^^^^^^^^^^^^^^^^^ End of stack listing ^^^^^^^^^^^^^^^^^^^^^
```

matrix_vector.5

Matrix-Vector-Multiplication

The multiplication operator is overloaded like this:

matrix_vector.5

```
# 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
;
```

matrix_vector.5

Applying it to the matrix and vector defined before yields this:

matrix_vector.5

```
lang5> * .
[   68   167   266   ](v)
```

matrix_vector.5

Ulam spiral

In 1963 Stanisław Marcin Ulam discovered the today so called *Ulam spiral* while playing with numbers:

```
73                      79
    43              47
71              23
    41      7
        19          2   11
            5       3       29
67      17              13
    37                      31
                61      59
```

- Create a spiral out of integer numbers, starting at 1.
- Remove all non-prime numbers.

This can be done in **lang5** without any explicit loops or variables like this:

Ulam spiral

ulam.5

```
: 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
```

ulam.5

Other examples

The **lang5**-distribution contains a wealth of additional examples which include the examples shown here. In addition the following **lang5**-programs are contained in the kit:

`apple.5`:	Compute a Mandelbrot set (ASCII displayed).
`cantor.5`:	Generate a Cantor set.
`cosine.5`:	Cosine approximation using a MacLaurin series.
`gol.5`:	Conway's Game-Of-Live implemented in **lang5**.
`perfect.5`:	Find perfect numbers in an interval.
`sort.5`:	Sorting external data.

Conclusion

Conclusion

- **lang5** is a powerful tool and can be used for a wide range of (mostly mathematical) applications, ranging from rapid prototyping to ad hoc data analysis, experimental mathematics and the like.
- The interpreter is rather stable – the existing features are extremely unlikely to change (although new features will be added).
- If you are interested in using or even extending the **lang5**-interpreter do not hesitate to join the project team at `http://lang5.sf.net` or contact the author dirctly at `ulmann@vaxman.de`
- The author would like to thank Mr. Thomas Kratz who wrote most of the current incarnation of the **lang5**-interpreter.

Bibliography

Bibliography

- [Adams et al. 09] Jeanne C. Adams, Walter S. Brainerd, Richard A. Hendrickson, Richard E. Maine, Jeanne T. Martin, Brian T. Smith, *The Fortran 2003 Handbook*, Springer, 2009
- [Brodie 04] Leo Brodie, *Thinking Forth – A Language and Philosophy for Solving Problems*, 2004
- [Conklin 07] Edward K. Conklin, Elizabeth D. Rather, *Forth Programmer's Handbook*, FORTH, Inc., 2007
- [Giloi 77] Wolfgang K. Giloi, *Programmieren in APL*, deGruyter, Berlin, 1977
- [Katzan 70] Harry Katzan Jr., *APL Programming and Computer Techniques*, Van Nostrand Reinhold Company, 1970