Forth: A New Synthesis
Progress Report
Growing Forth with seedForth

Ulrich Hoffmann <uho@xlerb.de>

Overview

Growing Forth

- introduction
- preForth (simpleForth, Forth)
- seedForth
- summary

Forth: A New Synthesis

- EuroForth 2016

  Implementing the Forth Inner Interpreter in High Level Forth

- Forth 2017

  Stack of Stacks strings on the data stack

- EuroForth 2017

  handler based outer interpreter

Forth: A New Synthesis

- Forth everywhere (as much as possible)
- bootstrap-capable self-generating system
- completely transparent
- simple to understand
- quest for simplicity
- biological analogy
- disaggregation
- Can Forth emerge from less than Forth?

preForth

- Can Forth emerge from less than Forth?
- What can be omitted?
  - no DOES>
  - no BASE
  - no STATE
  - no pctured numerical output <# # #>
  - no CATCH/THROW

preForth

- Can Forth emerge from less than Forth?
- What else can be omitted?
  - no immediate words, i.e.
    - no control structures IF ELSE THEN BEGIN WHILE REPEAT UNTIL
  - no defining words - but   :
  - no memory @ ! CMOVE ALLOT ,
  - no input stream
  - no dictionary, no EXECUTE nor EVALUATE
  - not interactive

preForth

- What remains?
    - stack
    - return stack
    - just **`?EXIT`** and recursion as control structures
    - colon definitions
    - optional tail call optimization
    - in- and output via **`KEY`**/**`EMIT`**
    - decimal positive und negative numbers (single cell)
    - character literals in 'x'-notation
    - decimal number output (single cell)

preForth Programs

How do they look like?

```
: countdown ( n -- )
   dup .
   ?dup 0= ?exit
   1-  tail countdown ;
```

```
5 countdown

5 4 3 2 1 0
```

preForth Programs

How do they look like?

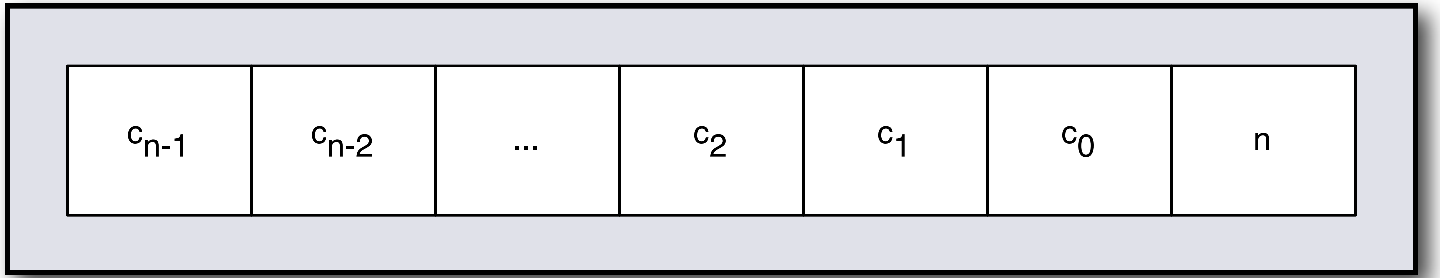
```
: dashes ( n -- )
  ?dup 0= ?exit
    '-' emit 1- tail dashes ;
```

```
5 dashes

-----
```

preForth Programs

How do they look like?

```
\ show displays topmost string

: show ( S -- )
   ?dup 0= ?exit  swap >r 1- show
   r> emit ;
```



preForth Programs

How do they look like?

```
: ."Hello,_world!" ( -- )
    'H' 'e' 'l' 'l' 'o' ',' bl
    'w' 'o' 'r' 'l' 'd' '!' 13 show ;
```

```
Hello world!
```

preForth Operations for Stack Strings

- `_dup ( S -- S S )`
- `_swap ( S1 S2 -- S2 S1 )`
- `_drop ( S -- )`
- `_show ( S -- )`
- Patterns
  - `dup pick  ( S -- c )` first character
  - `swap 1+ ( S1 c -- S2 )` append character

Pick and Roll ?!

```
: pick ( xn-1 ... x0 i -- xn-1 ... x0 xi )
   over swap ?dup 0= ?exit nip swap
   >r 1- pick r> swap ;

: roll ( xn-1 ... x0 i -- xn-1 ... xi-1 xi+1 ... x0 xi )
   ?dup 0= ?exit swap >r 1- roll r> swap ;
```

```
: ?dup ( x -- x x | 0 )
   dup dup ?exit drop ;
```

Primitives

- Forth everywhere (as much as possible)
- the must be some basis:
  - 13 primitives:

```
emit key
dup swap drop
0< -
?exit
>r r>
nest unnest
lit
```

Defintion of Primitives

Formulate in the plattform target language (here i386-Asm)

```
code ?exit ( f -- )
        pop eax
        or eax, eax
        jz qexit1
        mov esi, [ebp]
        lea ebp,[ebp+4]
qexit1: next
;
```

Describing Target Code

Formulate in the plattform target language (here i386-Asm)

```
prefix
format ELF


...

macro next  {
       lodsd
       jmp dword [eax]
}
...
;
```

pre

prelude

prefix

preamble

preformatted

preForth compiler

- accepts preForth programs from stdin
- writes plattform programs to stdout
  - here i386 assembler
  - more backends very easy (C, planned AMD64, stm8, NIGE)
- formulated itself in preForth
- can reproduce itself
- first bootstrap via gForth or SwiftForth
- machine code generated by plattform assembler

preForth compiler

- outer interpreter and compiler based on handlers
- Handler `( S -- i*x 0 | S )`

```
\ ?'x' detects and compiles a character literal
: ?'x' ( S -- 0 | S )
       dup 0= ?exit
       dup 3 - ?exit
       over   ''' - ?exit
       3 pick ''' - ?exit
       2 pick >r _drop r>
       ,lit 0 ;
```

- Handlers are combined in colon definitions.

preForth compiler

- Handlers are combined in colon definitions.
- preForth compiler loop:

```
: ] ( -- )
    token                   \ get next token
    \ run compilers
    ?; ?dup 0= ?exit        \ ;  leave compiler loop
    ?\                      \ comment
    ?tail                   \ marked as tail call
    ?'x'                    \ character literal
    ?lit                    \ number
    ?word                   \ word
    _drop  tail ] ;         \ ignore unhandled token and cycle
```

generated plattform code

?exit

```
; ?exit
_Qexit: DD _QexitX
_QexitX:pop eax
        or eax, eax
        jz qexit1
        mov esi, [ebp]
        lea ebp,[ebp+4]
qexit1: next
```

?dup

```
; ?dup
_Qdup:  DD _nest
_QdupX:
        DD _dup
        DD _dup
        DD _Qexit
        DD _drop
        DD _unnest
```

simpleForth

- preForth is turing complete

  Writing a complete Forth in preForth is possible...
  ... but cumbersome.

- extending preForth: simpleForth

simpleForth

- simpleForth is like preForth
- preForth ⊂ simpleForth

- in addition:
  - control structures: IF ELSE THEN BEGIN WHILE REPEAT UNTIL
  - definitions with and without Header in generated code
  - memory: @ ! c@ c! allot c, ,
  - variable constant
  - ['] execute
  - immediate definitions

Bootstrapping Forth

- full, interactive Forth ("*Forth*") in simpleForth

- new synthesis:
    - handler based text/interpretierer
    - dual words
    - dynamic memory management
    - ...

- works - but not really satisfying

Observations / Dislikes

- "double" description
  - control structures
  - header structures
  1. for the generated Forth image
  2. for use in the interactuce system

- continue quest

The Birth of seedForth

seedForth

- eliminates the issue of double described structures
- further simplifies the basis even further
- very small (potentially) interactive Forth system
- 460 LOC
- has dictionary extensible by colon definitions
- can be extended to a full-featured interactive Forth

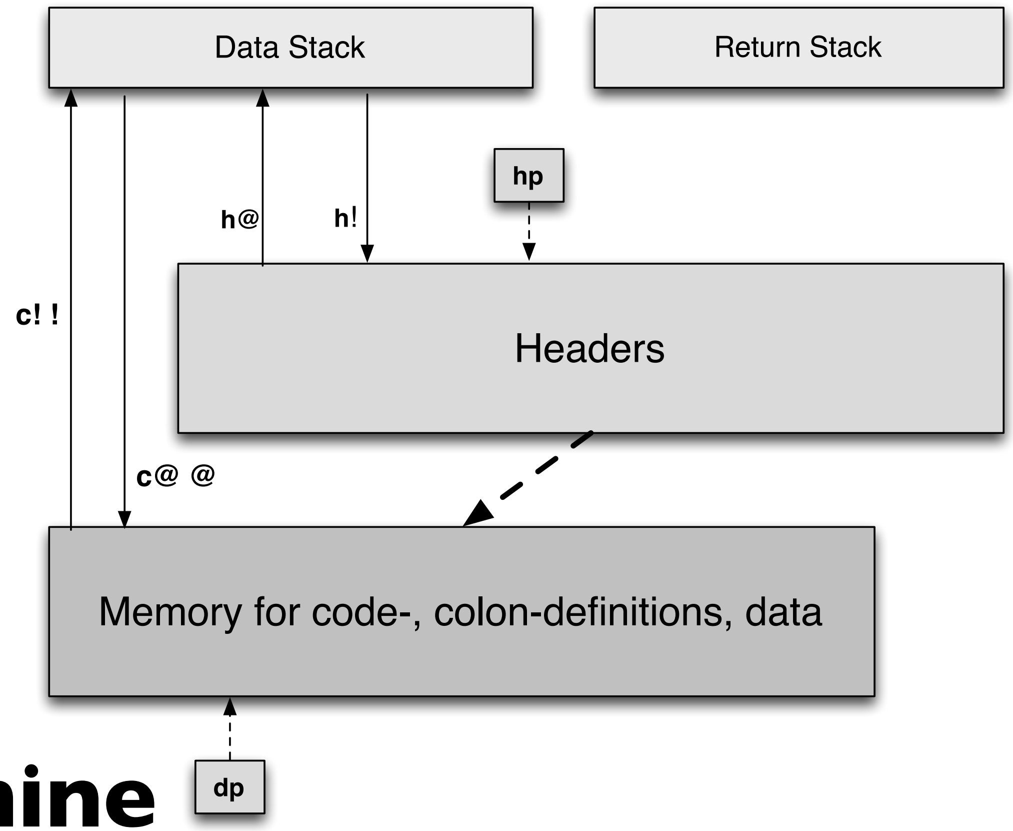
- *accepts source code in byte tokenized form*

- seedForth for i386 and AMD64

The Birth of seedForth

simplify names:

names are just numbers



seedForth virtual machine

- data stack, return stack
- dictionary
  addressable memory for code, colon defs, data
- headers
  array mapping word indices to start adresses

seedForth words

```
  $00 #FUN: bye        $01 #FUN: emit          $02 #FUN: key        $03 #FUN: dup
  $04 #FUN: swap       $05 #FUN: drop          $06 #FUN: 0<         $07 #FUN: ?exit
  $08 #FUN: >r         $09 #FUN: r>            $0A #FUN: -          $0B #FUN: unnest
  $0C #FUN: lit        $0D #FUN: @             $0E #FUN: c@         $0F #FUN: !
  $10 #FUN: c!         $11 #FUN: execute       $12 #FUN: branch     $13 #FUN: ?branch
  $14 #FUN: negate     $15 #FUN: +             $16 #FUN: 0=         $17 #FUN: ?dup
  $18 #FUN: cells      $19 #FUN: +!            $1A #FUN: h@         $1B #FUN: h,
  $1C #FUN: here       $1D #FUN: allot         $1E #FUN: ,          $1F #FUN: c,
  $20 #FUN: fun        $21 #FUN: interpreter   $22 #FUN: compiler   $23 #FUN: create
  $24 #FUN: does>      $25 #FUN: cold          $26 #FUN: depth      $27 #FUN: compile,
  $28 #FUN: new        $29 #FUN: couple        $2A #FUN: and        $2B #FUN: or
  $2C #FUN: catch      $2D #FUN: throw         $2E #FUN: sp@        $2F #FUN: sp!
  $30 #FUN: rp@        $31 #FUN: rp!           $32 #FUN: $lit
```

```
: interpreter ( -- )
   key execute   tail interpreter ;
```

```
: compiler ( -- )
   key ?dup 0= ?exit compile, tail compiler ;
```

seedForth Tokenizer

- convert human readable source code to byte tokenized source code ("*editor task*")
- about 100 LOC

demo.seedsource

```
program demo.seed

'H' # emit  'e' # emit  'l' # dup emit emit  'o' # emit  10 # emit

': 1+ ( x1 -- x2 ) 1 #, + ;'

'A' #  1+  emit   \ outputs B

end
```

↓

demo.seed

```
00000000  02 48 01 02 65 01 02 6c  03 01 01 02 6f 01 02 0a  |.H..e..l....o...|
00000010  01 20 0c 00 02 01 1e 22  15 0b 00 02 41 33 01 00  |. ....."....A3..|
```

seedForth Tokenizer

- convert human readable source code to byte tokenized source code ("*editor task*")
- about 100 LOC

demo.seedsource

```
program demo.seed

'H' # emit  'e' # emit  'l' # dup emit emit  'o' # emit  10 # emit

': 1+ ( x1 -- x2 ) 1 #, + ;'

'A' #  1+  emit   \ outputs B

end
```

demo.seed

```
00000000  02 48 01 02 65 01 02 6
00000010  01 20 0c 00 02 01 1e 2
```

```
...o...|
...A3..|
```

```
$ cat demo.seed | bin/seedForth
seed
Hello
B
```

seedForth grows

Planned extensions toward full-featured interactive Forth

- ✓ dynamic memory allocation with allocate, resize and free
- ✓ defining words including DOES>
- headers with dictionary search and DUAL behaviour word support
- text interpreter and compiler that work on non tokenized source using a handler based approach with string descriptors and regular expressions.
- compiling words
- a Forth assembler for the target platform and additional primitives,
- multitasking
- OOP
- file and operating system interface
- access to hardware
- the tokenizer and preForth can eventually also be expressed in seedForth and so it will be self contained.

Summary

preForth

- bootstrap capable, self-generating system
- complete transparency
- simple to understand

seedForth

- byte tokenized source code
- initially word names are number indices into the header array
- extensible to full-featured interactive Forth
- simple to understand

Can Forth emerge from less than Forth?

Yes - with preForth and seedForth ☺