Special Properties of Forth and Postscript

M. Anton Ertl
TU Wien

Debugging

- Stepping Debugger

  `dbg word`

  Only works with `gforth-itc`

- Tracer (`printf` debugging)

  `~~`

- Backtrace

IDE features

- `locate word`
  `n b g l`
  `edit word`
- `help word`
  `n b g l`
- `where word`
  `( u ) ww`
  `nw bw`
- after a Backtrace
  `( u ) tt nt bt`
- Decompiler
  `see word`
  `simple-see word`
  `see-code word`

Interpretation, Compilation, and Execution

```
\ Compilation
: hello ( -- )
  ." hello, world" ;

\ Interpretation
.( hello, world)

\ Interpretation and execution
hello
```

Interpretation

- exists in Forth, Postscript, Lisp, Prolog, Python, ...
- does not exist in Fortran, C, C++, Java, Rust, ...

No hard boundary between compile time and run time

... in the languge; there may be one in the head of the programmer.
No executable file; many systems have an image file.

- Initialize data structures
- Macros: Execution during compilation
- Run-time code generation: optimization or simplification

C++ has a separate programming language for macros (template language)

Initialization

```
/* C */ int a[] = {235,1857};

/* C, alternative */
int a[2];  a[0]=foo(2); a[1]=bar(3);

\ Forth
create a 2 foo , 3 bar ,
```

Macros

```
: endif POSTPONE then ; immediate
: foo ... if ... endif ... ;

: (map) ( a n -- a a')   cells over + swap ;
: map<   postpone (map)  postpone ?do postpone i postpone @ ; immediate
: >map   1 cells postpone literal  postpone +loop ; immediate
: step  0  array 1000 map< + >map drop ;
```

Code generation: LITERAL COMPILE,

```
: foo [ 5 cells ] literal ;
: ]cells ] cells POSTPONE literal ;
: foo [ 5 ]cells ;

: twice ( xt -- )
  dup compile, compile, ;
: 2+ [ ' 1+ twice ] ;
: 4* [ ' 2* twice ] ;
```

Code generation: Gray

```
: compile-test \ set -- )
 postpone literal
 test-vector @ compile, ;

: generate-alternative1 \ -- )
 operand1 get-first compile-test
 postpone if
 operand1 generate
 postpone else
 operand2 generate
 postpone endif ;
```

Quines

Programs that print themselves; the shorter the better.

```
source type
```

By avoiding the use of certain Forth features, we get a variety of quines:
`http://www.complang.tuwien.ac.at/forth/quines.html`

Name Binding

- What happens if a name is redefined?
- Algol (C, Java, ...): Compilation error
- Lisp, Postscript: Dynamic Name Binding
  The new definition replaces the old one completely
- Forth: Static name binding
  The old definition still exists
  Old uses use the old definition
  New uses use the new definition

Name binding: Collision

- Frequent case: programmer defines name x, library is extended by x:

```
( in main ) require lib1.fs
  ( in lib1.fs ) require lib2.fs
    ( in lib2.fs ) : x ." x1" ;
  ( in lib1.fs ) : y x ;
( in main ) : x ." x2" ;
( in main ) x \ outputs "x2"
( in main ) y \ outputs "x1"
```

- Algol: Compile-time error
- Lisp: Library 1 executes the wrong function x; run-time error or worse
- Forth: Warning at compile time, program works

Namensbindung: problems and solutions

- Collisions: conventions (C), namespaces (C++)

```
#define _POSIX_C_SOURCE 199506L
#include <unistd.h>
```

- Collisions: dictionaries (Postscript)

- Forward declarations: `defer` (Forth)

```
defer foo
: bar ... foo ... ;
:noname ... bar ... ; is foo
```

Stateless control structures

```
... IF ... THEN
... IF ... ELSE ... THEN
BEGIN ... WHILE ... REPEAT
BEGIN ... UNTIL
CASE ... OF ... ENDOF ... OF ... ENDOF ... ENDCASE
?DO ... LOOP
```

Control structures: foundation

	forwards	backwards
Unconditional branch	`AHEAD ( -- orig )`	`AGAIN ( dest -- )`
Conditional branch	`IF ( -- orig )`	`UNTIL ( dest -- )`
branch target	`THEN ( orig -- )`	`BEGIN ( -- dest )`

```
: foo
  BEGIN ( C: dest )
    ... IF ( C: dest orig )
      ... THEN ( C: dest )
    ... UNTIL ( C: ) ;
```

Control Structures: ground floor

```
: ELSE ( compilation: orig1 -- orig2 ; run-time: -- ) \ core
    POSTPONE ahead
    1 cs-roll
    POSTPONE then ; immediate restrict

: WHILE ( compilation: dest -- orig dest ; run-time: f -- ) \ core
    POSTPONE if
    1 cs-roll ; immediate restrict

: REPEAT ( compilation: orig dest -- ; run-time: -- ) \ core
    POSTPONE again
    POSTPONE then ; immediate restrict
```

Control structures: Usage

```
: foo
  ... if ( C: orig1 )
    ...
  else ( C: orig2 )
    ... begin ( C: orig2 dest )
      ... while ( C: orig2 orig3 dest )
      ... repeat ( C: orig2 )
  then ;
```

Control structures: unconventional

```
: foo
  begin ( C: dest )
    ... while ( C: orig1 dest )
    ... while ( C: orig1 orig2 dest )
    ... repeat ( C: orig1 )
  ... else ( C: orig3 )
    ... then ;
```

- Arbitrary stateless control structures with `cs-roll`

- Readability?

Gforth: generalized guarded commands

```
case
    ... ?of ... endof
    ... ?of ... contof
    ...
next-case \ or "0 endcase"
```

```
: gcd ( n1 n2 -- n )
    case
        2dup > ?of tuck - contof
        2dup < ?of over - contof
    ( n n' ) endcase ;

: collatz ( u -- )
    case
        dup .
        1 of endof
        dup 1 and ?of 3 * 1+ contof
        2/
    next-case ;
```

Types

Who knows the type of data?

		run-time system	
		no	yes
compiler	no	Forth	Postscript, Python, Lisp
	yes	C, Pascal	C++, Java

- Type knowledge of the programmer (e.g., sorted array)

- uniformity (Lisp) vs. specialization (Java)

Types: Checking

```
'a' 5 *
```

- How do you find type errors in Forth? Testing!
- With a little experience type errors are easy to find.
  experience in interpreting program output
  experience in writing test cases
  experience in writing programs for easy testability
- More intensive testing $\Rightarrow$ you also find other errors

Types: Checking

As programmers learned C with Classes or C++, they lost the ability to quickly find the "silly errors" that creep into C programs through the lack of checking. Further, they failed to take the precautions against such silly errors that good C programmers take as a matter of course. After all, "such errors don't happen in C with Classes." Thus, as the frequency of run-time errors caused by uncaught argument type errors goes down, their seriousness and the time needed to find them goes up.
Bjarne Stroustroup

Types: Overloading

```
int n;          n*3             n @ 3 *
float r;        r*3.0           r f@ 3.0e f*
int n;          n<3             n @ 3 <
unsigned u;     u<3             u @ 3 u<
```

Types: Advantages and disadvantages of Forth

\+ More uniformity: Better reusability

\+ Extensions as libraries that would need type system extension in other languages (OOP)

\+ Less complexity, e.g., for containers

– No type knowledge for garbage collection, marshalling etc.

– For changes affecting many lines static type checking would be useful
Poor man's checking: Use new names for changed interfaces
New names allow gradual changes

Storage management

- Static allocation
  `create allot`
  Small systems ($<$ 64KB)

- Dynamic allocation with explicit deallocation
  `allocate free`
  Keep track of allocations: Memory leaks and double `free`
  Medium systems

- Dynamic allocation with automatic deallocation
  `allocate` and garbage collection
  Large systems ($>$ 16MB)

Storage management and APIs

`read-line ( c-addr u1 wfileid -- u2 flag wior )`

Reads a line from wfileid into the buffer at c-addr u1. Gforth supports all three common line terminators: LF, CR and CRLF. A non-zero wior indicates an error. A false flag indicates that 'read-line' has been invoked at the end of the file. u2 indicates the line length (without terminator): u2<u1 indicates that the line is u2 chars long; u2=u1 indicates that the line is at least u1 chars long, the u1 chars of the buffer have been filled with chars from the line, and the next slice of the line with be read with the next 'read-line'. If the line is u1 chars long, the first 'read-line' returns u2=u1 and the next read-line returns u2=0.

`slurp-fid ( fid -- c-addr u )`

c-addr u is the content of the file fid

Object-oriented extension: features

- Dynamic Dispatch (virtual functions/methods)
- Instance Variables
- Single inheritance
- Static binding (C++: `A::m`)
- Basic class `object`
- `new`

Object-oriented extension: Usage (1)

```
object class
  cell var text
  cell var len
  cell var x
  cell var y
  method init
  method draw
end-class button

:noname ( o -- ) >r
 r@ x @ r@ y @ at-xy  r@ text @ r> len @ type ;
 button defines draw
:noname ( addr u o -- ) >r
 0 r@ x ! 0 r@ y ! r@ len ! r> text ! ;
 button defines init
```

Object-oriented extension: Usage (2)

```
button class
end-class bold-button

: bold   27 emit ." [1m" ;
: normal 27 emit ." [0m" ;

:noname bold [ button :: draw ] normal ; bold-button defines draw

button new Constant foo
s" thin foo" foo init
page
foo draw
bold-button new Constant bar
s" fat bar" bar init
1 bar y !
bar draw
```

Object-oriented extension: source code

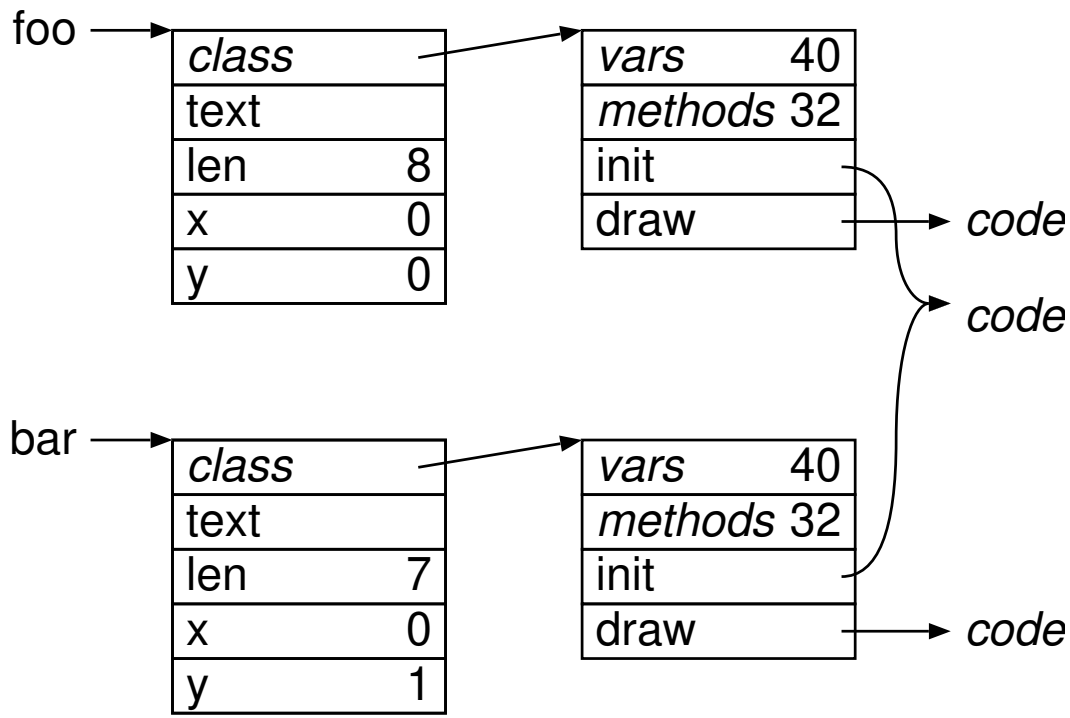
```
: method ( m v -- m' v ) Create  over , swap cell+ swap
  DOES> ( ... o -- ... ) @ over @ + @ execute ;
: var ( m v size -- m v' ) Create  over , +
  DOES> ( o -- addr ) @ + ;
: class ( class -- class methods vars ) dup 2@ ;
: end-class  ( class methods vars -- )
  Create  here >r , dup , 2 cells ?DO ['] noop , 1 cells +LOOP
  cell+ dup cell+ r> rot @ 2 cells /string move ;
: defines ( xt class -- ) ' >body @ + ! ;
: new ( class -- o )  here over @ allot swap over ! ;
: :: ( class "name" -- ) ' >body @ + @ compile, ;
Create object  1 cells , 2 cells ,
```

Explanation: https://bernd-paysan.de/mini-oof.html

Object-oriented extension: explanation

```
object class
  cell var text
  cell var len
  cell var x
  cell var y
  method init
  method draw
end-class button
button new Constant foo

button class
end-class bold-button
bold-button new Constant bar
```

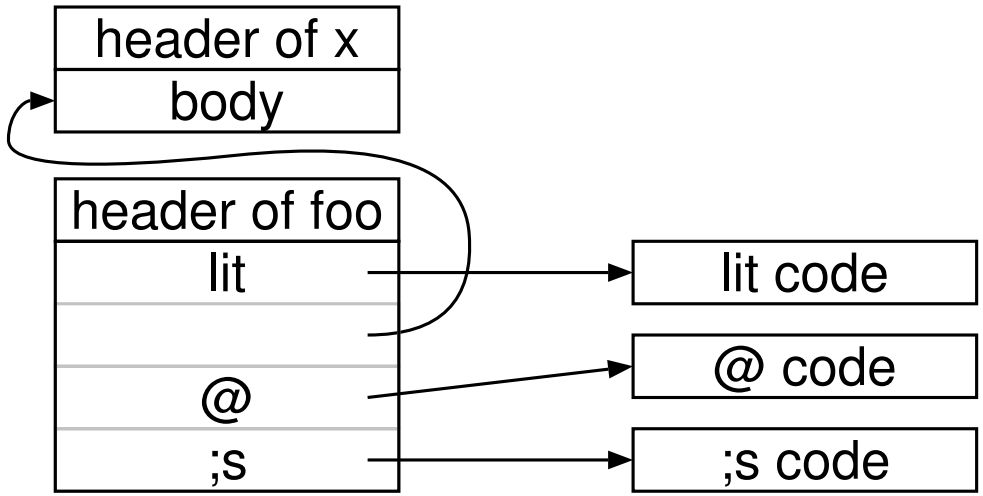


Forth-philosophy

- actually Chuck Moore's philosophy
- *Keep it simple!*
- *Do not speculate!*
  do not generalize
  do not design for reuse
  only take steps when necessary
- *Do it yourself!*
  no libraries
- *Do not bury your tools!*

Implementation

variable x
: foo x @ ;



- `see word`
- `simple-see word`
- `see-code word`

Postscript

- Syntax
  - `( ) < > [ ] { } / %` are delimiters
  - `<< >>` are lexemes
  - `( )` surround strings; balance parentheses, or use `\) \(`
- Types
  - Types known at run-time
  - Operators work on several types
  - Type checking at run-time
  - Limited non-static stack depth when building an array

Postscript: types

Typ	Beispiel-Literal
integer	`1`
real	`1.0`
boolean	`true`
name	`/name`
mark	`[`
null	`null`
operator	`/add load`
fontID	
save	
array	`[1 2]`
string	`(string)`
dictionary	`<< index1 wert1 index2 wert2 ... >>`
file	

- simple types: value semantics
- composite types: reference semantics (shallow copying)

Attributes: literal und executable objects

Typ	executable example	executable literal
Name	`name`	`/name`
Operator	`//add`	`/add load`
Array	{`dup mul`}	`[ 1 2 ]`

- Attribute of object in addition to type
- Executing a literal object pushes it on the stack
- Executing an executable object has a type-specific effect
- Difference between direct and indirect execution
- Attributes are ignored when the object is treated as data (most operations)
- Further attribute: access

Procedures and arrays

- Procedures are executable arrays
- Syntactically different
  `{ 1 2 add }`
  `[ 1 2 add ]`
- Binding a procedure to a name
  and executing the name executes the procedure indirectly
  Also with `exec`, `if` etc.
- Indirect execution of a procedure:
  Elements of the procedure are executed *directly*
- Direct execution of a procedure:
  push it on the stack

```
/x { { 1 2 add } } def
x exec
```

Control Structures

Operators that take (usually) procedures as parameters

```
a 0 lt { 1 == } if
a 0 lt { 1 == } { 2 == } ifelse
5 1 10 { == } for
5 2 10 { == } for
[ 1 7 3 ] { == } forall
<< /c 1 /a 2 /b 3 >> { == == } forall
4 { (abc) = } repeat
5 { dup 0 lt { exit } if dup = 1 sub } loop pop
```

Names and dictionaries

- `/squared {dup mul} def`
- By executing `squared`, the procedure is executed indirectly
  $\Rightarrow$ its elements are executed directly
- Definition is stored in current dictionary
- dictionary corresponds to Forth's wordlist
- Dictionary stack corresponds to Forth's search order

Name binding

- name binding at run-time
  `/foo {bar} def  /bar {1} def`
- Usage for local variables
  `/foo { << /a rot >> begin ... a ... end } def`
- Dynamic scoping
  `<< /a 5 >> begin { a } end exec` gives an error
  Most languages support static scoping only
- Similar to environment-variables in Unix

```
/foo { ... conf ... } def
<< /conf { 5 } >> begin foo end
<< /conf { x } >> begin foo end
```

Comparison

```
/v [ 0 1 999 {} for ] def
/step {0 v { add } forall} def
100000 {step pop} repeat
```

```
: (map) ( a n - a a')   cells over + swap ;
: map[   postpone (map)  postpone ?do postpone I postpone @ ; immediate
: ]map   1 cells postpone literal  postpone +loop ; immediate

create array 1000 cells allot
: init  1000 0 DO I  array I cells + !  LOOP ;
init
: step  0  array 1000 map[ + ]map drop ;
: bench  100000 0 DO step LOOP ;
bench
```

Document Structuring Conventions

```
%!PS-Adobe-2.0
%%Creator: dvips(k) 5.95a Copyright 2005 Radical Eye Software
%%Title: slides.dvi
%%Pages: 4 0
%%PageOrder: Ascend
%%Orientation: Landscape
%%BoundingBox: 0 0 595 842
%%DocumentFonts: LCMSS8 CMTT8 CMSY8 CMMI8 LCMSSI8 LCMSSB8 CMSY10
%%DocumentPaperSizes: a4
%%EndComments
... ProcSets ...
... Fonts ...
... Setup ...
%%Page: (0,1,2,3,4,5,6,7) 1
... Postscript Code ...
```

```
%%Page: (8,9,10,11,12,13,14,15) 2
... unabhängig von anderen Seiten
%%Trailer
...
%%EOF
```

Example:
Source: `https://www.complang.tuwien.ac.at/anton/lvas/ertl%26pirker97.txt`
Result: `https://www.complang.tuwien.ac.at/papers/ertl%26pirker97.ps.gz`

Encapsulated Postscript

- Usually for graphics
- Is included in other documents
- Examples: `https://www.complang.tuwien.ac.at/anton/eps-gallery`

```
%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: 90 651 387 709
... Prolog ...
%%Page: 1 1
... Nur eine Seite, meist ohne showpage ...
```