Spezielle Features von Forth und Postscript

M. Anton Ertl
TU Wien

Debugging

- Stepping Debugger

  `dbg word`

  Funktioniert nur mit `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`
- nach einem Backtrace
  `( u ) tt nt bt`
- Decompiler
  `see word`

Interpretation, Compilation, und Ausführung

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

\ Interpretation
.( hello, world)

\ Interpretation und Ausführung
hello
```

Interpretation geht

- in Forth, Postscript, Lisp, Prolog, ...
- nicht in Fortran, C, C++, Java, ...

Keine Trennung zwischen Compilationszeit und Laufzeit

... durch die Sprache, nur im Kopf des Programmierers.
Kein Executable, bei vielen Systemen ein Image.

- Initialisierung von Datenstrukturen
- Macros: Ausführung während der Compilation
- Run-time code generation: Optimierung oder Vereinfachung

C++ hat eigene Programmiersprache für Macros (Templates)

Initialisierung

```
/* 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 ;
```

Codeerzeugung: LITERAL COMPILE,

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

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

Codeerzeugung: 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

Programme, die sich selbst drucken, möglichst kurz.

```
source type
```

Varianten, wenn man verschiedene Features von Forth nicht benutzt:
`http://www.complang.tuwien.ac.at/forth/quines.html`

Namensbindung

- Was passiert bei Wiederdefinition eines Namens?
- Algol (C, Java, ...): Compilationsfehler
- Lisp, Postscript: Dynamic Name Binding
  Neue Definition ersetzt alte
- Forth: Static name binding
  Alte Definition existiert weiter
  Alte Verwendungen beziehen sich auf alte Definition
  Neue Verwendungen beziehen sich auf neue Definition

Namensbindung: Kollision

- Häufiger Fall: Programm definiert Name x, library wird um x erweitert
- Algol: Fehler zur Compilezeit
- Lisp: Fehler zur Laufzeit (Falsche Funktion wird ausgeführt)
- Forth: Warnung zur Compilezeit, Programm funktioniert

Namensbindung: Probleme und Lösungen

- Kollisionen: Konventionen (C), Namespaces (C++)

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

- Kollisionen: Dictionaries (Postscript)

- Vorwärtsdeklarationen: `defer` (Forth)

Lokale Kontrollstrukturen

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

Kontrollstrukturen: Fundament

	Vorwärts	Rückwärts
Unbedingter Sprung	`AHEAD ( -- orig )`	`AGAIN ( dest -- )`
Bedingter Sprung	`IF ( -- orig )`	`UNTIL ( dest -- )`
Sprungziel	`THEN ( orig -- )`	`BEGIN ( -- dest )`

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

Kontrollstrukturen: Erweiterungen

```
: 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
```

Kontrollstrukturen: Erweiterungen

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

Kontrollstrukturen: unkonventionell

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

- Mit `cs-roll` beliebige Kontrollstrukturen

- Lesbarkeit?

Typen

Wer weiß den Typ eines Datums?

		Laufzeitsystem	
		nein	ja
Compiler	nein	Forth	Postscript, Python, Lisp
	ja	C, Pascal	C++, Java

- Typwissen des Programmierers (z.B. sortiertes Array)

- Uniformität (Lisp) vs. Differenzierung (Java)

Typen: Überprüfung

```
'a' * 5
```

- Wie geht man in Forth damit um? Testen!
- Mit etwas Erfahrung findet man Typfehler schnell.
- Intensiveres Testen $\Rightarrow$ man findet auch andere Fehler.

> 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*

Typen: 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<
```

Typen: Vorteile und Nachteile von Forth

\+ Mehr Uniformität: Bessere Wiederverwendbarkeit

\+ Erweiterungen möglich, die Typsystemerweiterung brauchen (OOP)

\+ Weniger Komplexität z.B. bei Containern

– Kein Typwissen für Garbage Collection, Marshalling etc.

– Bei weitgreifenden Änderungen wäre statische Überprüfung praktisch

Speicherverwaltung

- Statische Allokation
  `create allot`

- Dynamische Allokation mit expliziter Deallokation
  `allocate free`

- Dynamische Allokation mit automatischer Deallokation

Speicherverwaltung und 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

Objekt-Orientierte Erweiterung: Features

- Dynamic Dispatch (Virtuelle Funktionen)
- Instance Variables
- Einfachvererbung
- Statische Bindung (C++: `A::m`)
- Basisklasse `object`
- `new`

Objekt-Orientierte Erweiterung: Verwendung (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
```

Objekt-Orientierte Erweiterung: Verwendung (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
```

Objekt-Orientierte Erweiterung: Source Code

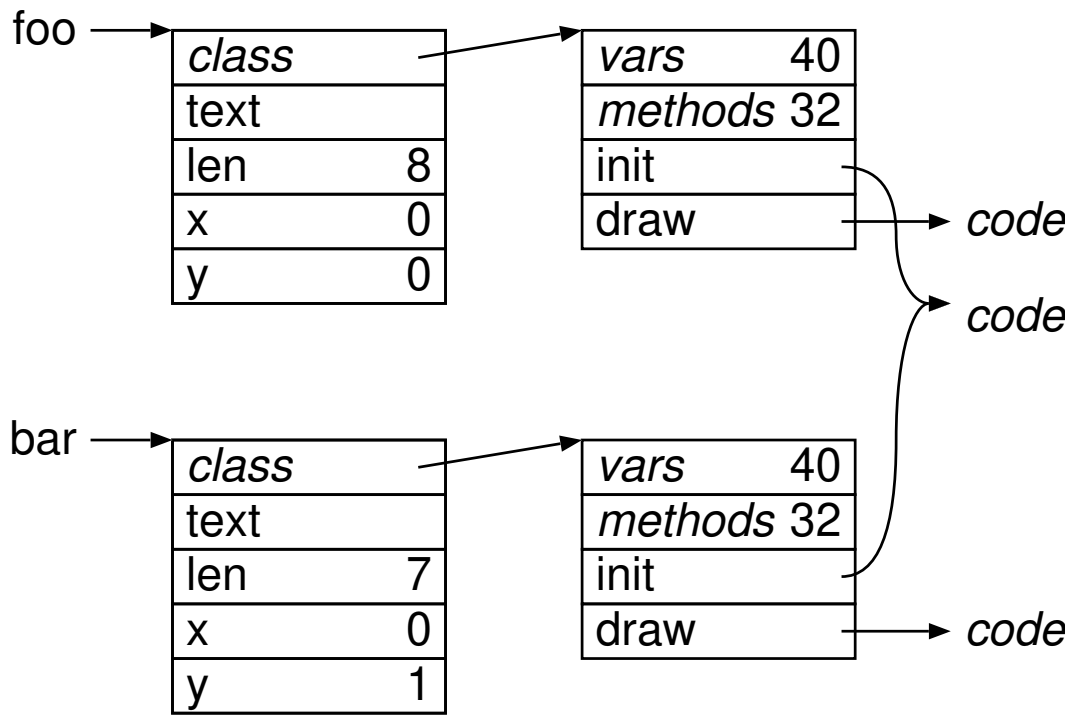
```
: method ( m v -- m' v ) Create  over , swap cell+ swap
  DOES> ( ... o -- ... ) @ over @ + @ execute ;
: var ( m v size -- m v' ) Create  over , +
  DOES> ( o -- addr ) @ + ;
: class ( class -- class methods vars ) dup 2@ ;
: end-class  ( class methods vars -- )
  Create  here >r , dup , 2 cells ?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 ,
```

Erklärung: `https://bernd-paysan.de/mini-oof.html`

Objekt-Orientierte Erweiterung: Erklärung

```
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
```

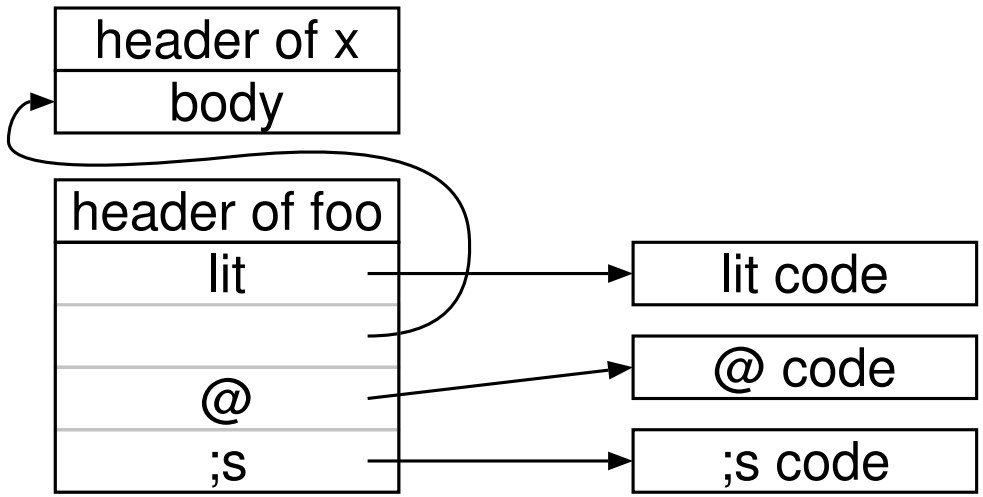


Forth-Philosophie

- eigentlich Chuck Moore's Philosophie
- *Keep it simple!*
- *Do not speculate!*
  keine Verallgemeinerung
  keine Wiederverwendung
  Erweiterungen/Hooks erst, wenn nötig
- *Do it yourself!*
  keine Libraries
- *Do not bury your tools!*

Implementierung

```
variable x
: foo x @ ;
```



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

Postscript

- Syntax
  - `( ) < > [ ] { } / %` sind Delimiter
  - `<< >>` sind Lexeme
  - `( )` umgeben Strings; Klammern ausbalancieren, oder `\) \(`
- Typen
  - Typen zur Laufzeit bekannt
  - Operatoren auf verschiedene Typen anwendbar
  - Typprüfung zur Laufzeit
  - Begrenzt nicht-statische Stacktiefe beim Aufbau von Arrays

Postscript: Typen

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	

- Referenz-Semantik (shallow copying)

Literale und ausführbare Objekte

Typ	Beispiel-Executable	Beispiel-Literal
Name	`name`	`/name`
Operator	`//add`	`/add load`
Array	{`dup mul`}	`[ 1 2 ]`

- Attribut von Objekten zusätzlich zum Typ
- Ausführung: Literale Objekte werden auf den Stack gelegt.
- Ausführung: bei ausführbaren Objekte typabhängiger Effekt
- Unterschied zwischen direkter und indirekter Ausführung
- Bei Betrachtung als Daten (z.B. Vergleich) sind Attribute egal.
- Weiteres Attribut: Access

Prozeduren und Arrays

- Prozeduren sind ausführbare Arrays
- Syntaktisch unterschiedlich
- `{ 1 2 add }`
- `[ 1 2 add ]`
- Elemente einer Prozedur werden *direkt* ausgeführt
  Prozeduren werden auf den Stack gelegt: `{ { 1 2 add } }`

Kontrollstrukturen

Operatoren mit Prozedur als Parameter

```
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
```

Namen und Dictionaries

- `/squared {dup mul} def`
- Über `squared` wird Prozedur indirekt ausgeführt
  $\Rightarrow$ die Elemente werden (direkt) ausgeführt
- Definition im aktuellen Dictionary
- Dictionary entspricht wordlist in Forth
- Dictionary stack entspricht search order

Namensbindung

- Namensbindung zur Laufzeit
  `/foo {bar} def  /bar {1} def`

- Verwendung als lokale Variablen
  `/foo { << /a rot >> begin ... a ... end } def`

- Dynamisches Scoping
  `<< /a 5 >> begin { a } end exec` liefert Fehler
  Viele Sprachen unterstützen statisches Scoping

- Ähnlich Environment-Variables in Unix

Vergleich

```
/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
```

Encapsulated Postscript

- Üblicherweise für Graphiken

- Wird in andere Dokumente eingebaut

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