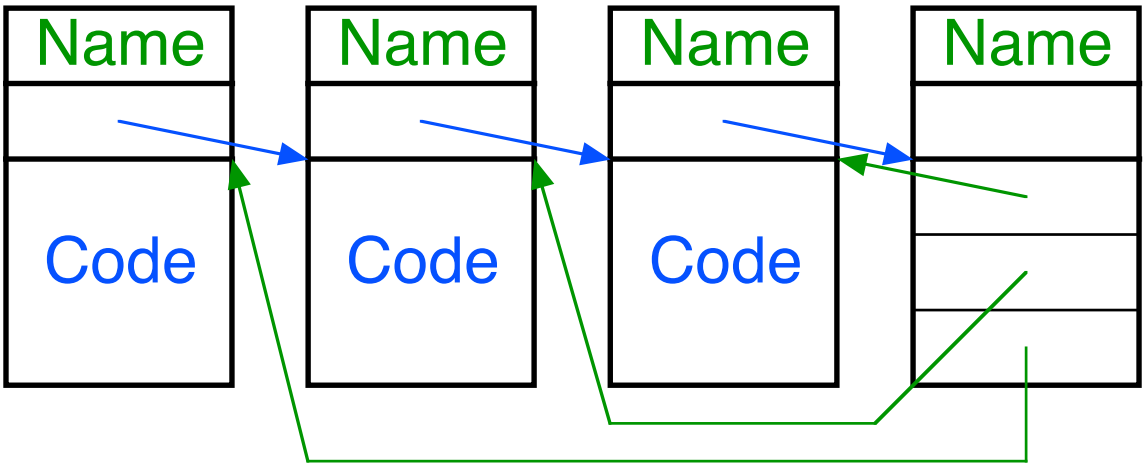
Threaded Code

threaded = aufgefädelt

Die interne Darstellung eines threaded Interpreters ist eine Liste von Adressen vorher definierter interner Darstellungen (Unterprogrammen). Diese Darstellungen sind in einer linearen Liste aufgefädelt.

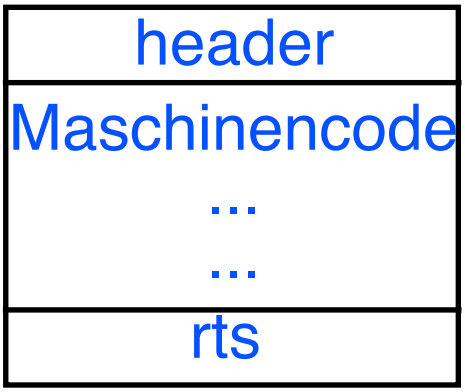


Die einzelnen Elemente werden übersetzt. Die abstrakte Maschine ist meistens eine Stack-maschine.

FORTH

subroutine threaded code

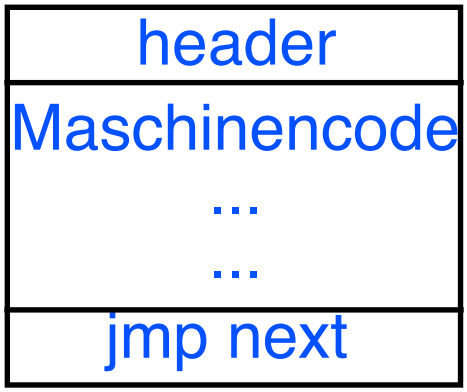
Maschinencode



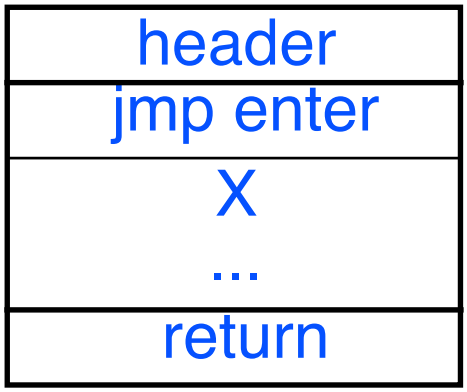
Zwischencode

direct threaded code

Maschinencode

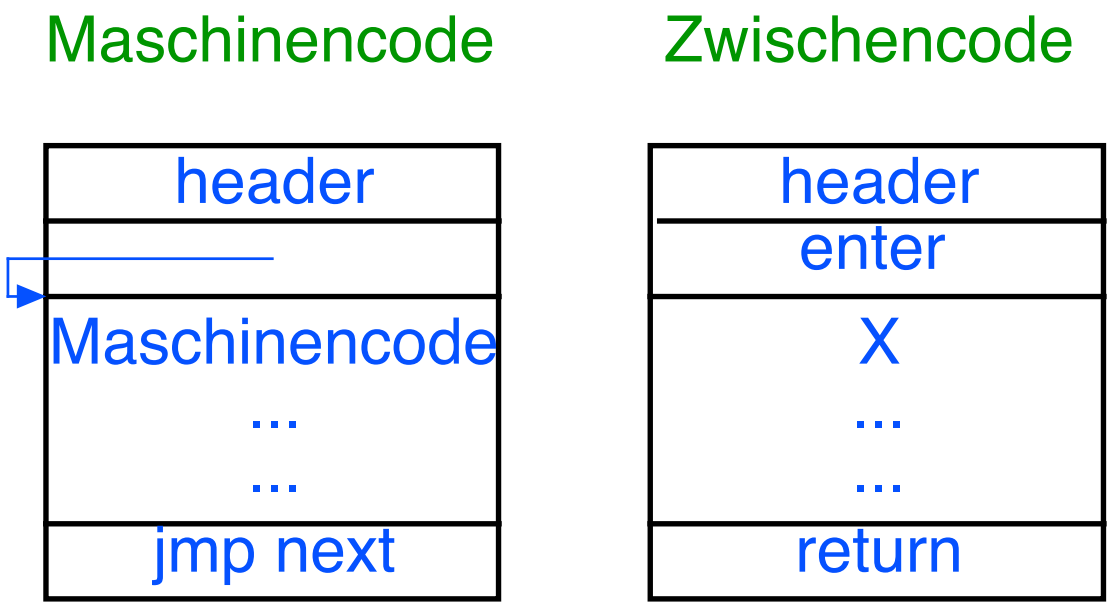


Zwischencode




```
enter:    push    ic
          move    ic = instr + jmp_len
next:     move    instr = (ic)+
          jmp     (instr)

return:   pop     ic
          jmp     next
```

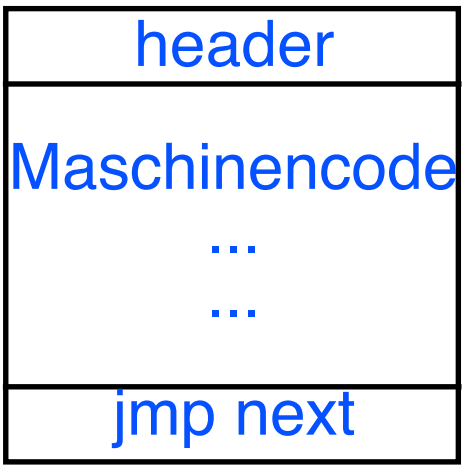
indirect threaded code



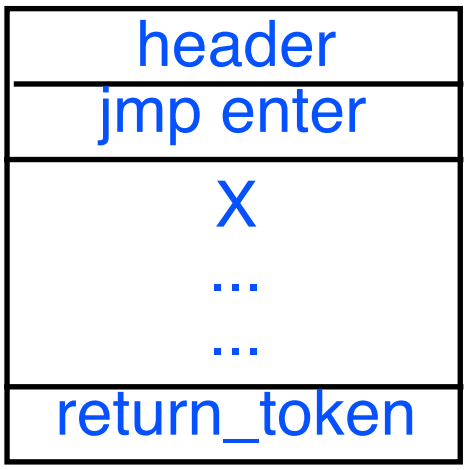

```
enter:    push    ic
          move    ic = instr
next:     move    instr = (ic)+
          move    ind = (instr)+
          jmp     (ind)

return:   pop     ic
          jmp     next
```

token threaded code

Maschinencode



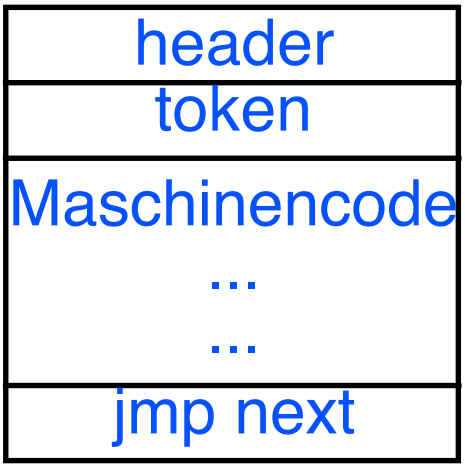
Zwischencode



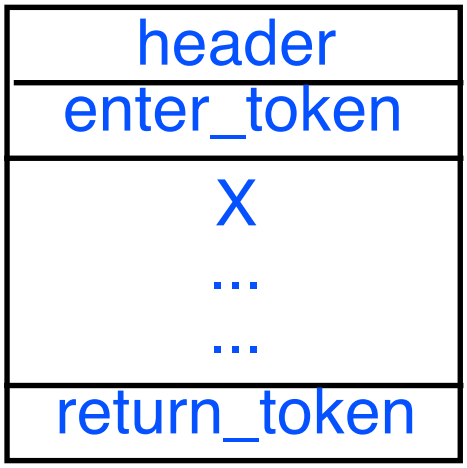

```
enter:    push    ic
          move    ic = addr + jmp_len
next:     move    instr = (ic)+
          move    addr = tab(instr)
          jmp     (addr)

return:   pop     ic
          jmp     next
```

indirect token threaded code

Maschinencode



Zwischencode




```
enter:    push   ic
          move   ic = addr + token_len
next:     move   instr = (ic)+
          move   addr = tab(instr)
          move   addr = (addr)
          move   addr = tab(addr)
          jmp    (addr)

return:   pop    ic
          jmp    next
```

Kosten auf dem MC68000

subroutine threaded 220 Zyklen 4 byte

```
enter:
next:    bsr + rts
return:  rts
```

enter = 0 next = 34 return = 16

direct threaded 162 Zyklen 2 byte

```
enter:   move.w  a6,-(a7)
         lea     a6,#4(a5)
next:    move.w  (a6)+,a5
         jmp     (a5)
return:  move.w  (a7)+,a6
         move.w  (a6)+,a5
         jmp.w   (a5)
```

enter = 42 next = 16 return = 24

indirect threaded 212 Zyklen 2 byte

```
enter:   move.w  a6,-(a7)
         move    a5,a6
next:    move.w  (a6)+,a5
         move.w  (a5)+,a4
         jmp     (a4)
return:  move.w  (a7)+,a6
         move.w  (a6)+,a5
         move.w  (a5)+,a4
         jmp     (a4)
```

enter = 36 next = 24 return = 32

token threaded 248 Zyklen 1 byte

```
enter:   move.b  a6,-(a7)
         lea     a6,#4(a4)
next:    move.b  (a6)+,a5
         move.w  tab(a5),a4
         jmp     (a4)
return:  move.w  (a7)+,a6
         move.b  (a6)+,a5
         move.w  tab(a5),a4
         jmp     (a4)
```

enter = 44 next = 28 return = 36

indirect token threaded 408 Zyklen 1 byte

```
enter:   move.b  a6,-(a7)
         lea     a6,#1(a4)
next:    move.b  (a6)+,a5
         move.w  tab(a5),a4
         move.b  (a4)+,a5
         move.w  tab(a5),a3
         jmp     (a3)
return:  move.w  (a7)+,a6
         move.b  (a6)+,a5
         move.w  tab(a5),a4
         move.b  (a4)+,a5
         move.w  tab(a5),a3
         jmp     (a4)
```

enter = 64 next = 48 return = 56

Kosten auf dem MIPS R3000

subroutine threaded 17 Zyklen 8 byte

Ein delay-slot für Lade- und Sprungbefehle

`r31` return address register

```
enter:   add    r30,r30,-4
         sw     r31,(r30)

next:    bal label + j r31
return:  lw     r31,(r30)
         add    r30,r30,4
         j      r31
```

enter = 2 next = 2 return = 3

direct threaded 30 Zyklen 4 byte

```
enter:   sw       r1,-4(r30)
         add      r1,r2,8
         lw       r2,4(r2)
         add      r30,r30,-4
         j        r2
         nop
next:    lw       r2,(r1)
         delay slot
         jmp      r2
         add      r1,r1,4
return:  lw       r1,(r30)
         add      r30,r30,4
         next
```

enter = 6 next = 3 return = 6

Die Sprache ZIP
(Z80 Interpretative Processor)

Zip ist eine FORTH artige Sprache

Wichtigste Merkmale:

Indirekt threaded code

Stack orientiert

linear verkettete 3 Zeichen lange Namen

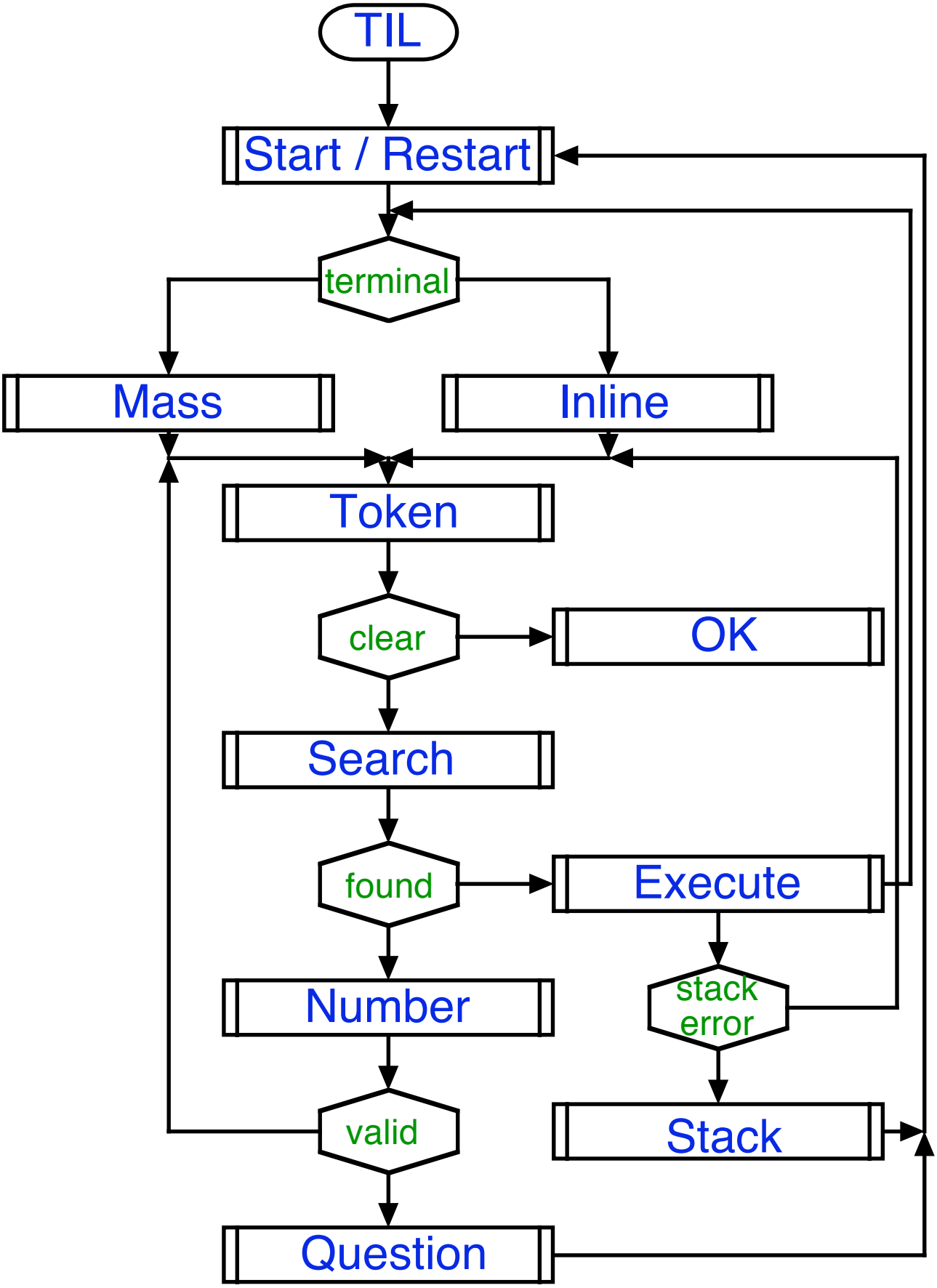
Z80 Maschinencode und ZIP

4 Kb Memory

integer Arithmetik

Loop-Anweisungen (kein goto)

einfacher Line Editor


TIL
Start / Restart
terminal
Mass
Inline
Token
clear
OK
Search
found
Execute
Number
stack
error
valid
Stack
Question

ZIP Funktionen

Start/Restart: initialisiert Stack pointer, System Variable und Execute-Modus

Mass/Inline: liest eine Zeile in Zeilenbuffer

Token: liest den nächsten token aus dem Zeilenbuffer und stellt ihn in Dictionary

Ok: gibt eine Ok-Meldung aus

Search: durchsucht Dictionary nach dem token, gibt die Adresse und ein gefunden Flag auf den Stack

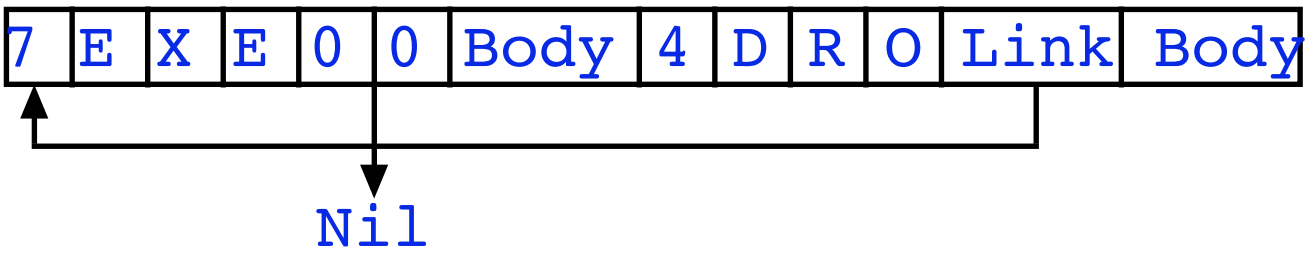
Execute: Wenn execute-Mode, dann Kommando ausführen, wenn compile-Mode, dann wenn immediate-Kommando, dann ausführen, sonst übersetzen

Stack: gibt Fehlermeldung aus, geht zu Start

Number: stellt Nummer auf Stack oder übersetzt sie

Question: gibt token und Fehlermeldung aus

Dictionary Format



Nummern: Integer, 2 byte auf Stack, im Body zuerst literal handler und dann der Wert

Strings: Auf Stack letzter Buchstabe zuunterst mit high-byte gesetzt, in Body zuerst Länge, dann die einzelnen Buchstaben

Boolesche Werte: 1..true, 0..false,
bei Abfragen jeder Wert ≠ 0 true

Konstanten und Variablen

Konstante: n CONSTANT name

DECIMAL 255 CONSTANT max

: ... max ...;
: ... 255 ...;
Name und Wert mit Zeiger auf Konstanten-aktivierungscode gespeichert

Variable: n VARIABLE name

Bei Aktivierung des Namens wird Adresse auf den Stack gelegt.

Arrays: n VARIABLE name size DP +!

Variable wird angelegt, dann wird der Dictionaryzeiger um size erhöht. Bei Indizierung muß der Index zur Adresse addiert werden.

Users: USERS n

Addiert n zur Adresse vom User-Speicher-block. Dieser Speicherblock kann beliebig im Speicher stehen.

Systemparameter

Können als Variable oder als Register verwendet werden

IR:	Instruction Register
WA:	Word-Address Variable
SP:	Daten Stackpointer
RSP:	Return Stackpointer
MODE:	Execute/Compile Mode false/true
STATE:	Immediate Wort wenn MODE=STATE
DP:	Dictionary Zeiger Variable
CONTEXT:	Suchvokabular
CURRENT:	Definitionsvokabular
START:	true, wenn erster Start
LPB:	Zeilenbuffer Zeigervariable
BASE:	Zahlenbasis Variable

Stackbefehle

DROP:	Entferne oberstes Stackelement
DUP:	verdopple oberstes Stackelement
SWAP:	vertausche obere 2 Stackelemente
OVER:	kopiere 2. Stackelement auf den Stack
2DUP:	verdreifache oberstes Stackelement
2SWAP:	vertausche oberstes und 3. Stack-element
2OVER:	kopiere 3. Stackelement auf den Stack
RROT:	ABC ⇒ CAB
LROT:	ABC ⇒ BCA
!:	*stack[top] = stack[top-1]; top-=2;
+!:	*stack[top] += stack[top-1]; top-=2;
0SET:	*stack[top--] = 0;
1SET:	*stack[top--] = 1;
@:	stack[top] = *stack[top];
<R:	Pop Daten- und Push Returnstack
R>:	Pop Return- und Push Datenstack

Rechenbefehle

ABS, MINUS, +, -, *, /, /MOD, MOD/, */, */MOD

MAX, MIN, 2*, 2/, 1+, 2+, 1- 2-

AND, OR, XOR, NOT

=, >, <, 0=, 0<

KEY, ECHO, CLEAR, CRETURN, SPACE, TYPE, DISPLAY

#: pop x, y=x mod base, chr(y), x div base
#S: führt solange # aus, bis stack[top] == 0
#>: zeigt die Zahl mittels DISPLAY an
.: gibt stack[top] aus
?: gibt *stack[top] aus
,: Speichert oberstes Stackelement auf
    Adresse des DP ab und erhöht DP
HERE: legt DP auf den Stack
?SP: legt Stackpoin
?RS: legt Returnstackpointer auf den Stack
TOKEN: Interpreter
´: sucht den token im Vokabular und gibt
    Adresse auf den Stack
ABORT, ASPACE, SEARCH
ENTRY: legt Adresse des zuletzt definierten
    Wertes auf den Stack
CA!: legt die Adresse, die auf dem Stack ist
    und speichert es an der WA des letzten
    Wortes
WAIT, FILL, ERASE, DUMP, ADUMP

Kontrollstrukturen

BEGIN ... END Schleife

BEGIN ... flag END

flag IF ... ELSE THEN

flag IF ... THEN

BEGIN ... flag IF ... WHILE

BEGIN ... flag IF ... ELSE ... WHILE

end start DO ... LOOP

end start DO ... inc +LOOP

Schleifenindex steht auf dem Returnstack
und wird um 1 oder inc erhöht

LEAVE: Schleifenexit für DO-Loops

switch mit Sprungtabelle implementieren

Compiling Keywords

CREATE: stellt den nächsten token in den Dictionary

:: startet Compilemode, macht ein CREATE und setzt die Adresse von enter in den Dictionary

;: setzt Adresse von return in den Dictionary und beendet Compilermode

;CODE: setzt Adresse von SCODE in Dictionary und beendet Compilermode

Assembler
Screens
andere Dictionaryformate
module threaded code

```
%!PS-Adobe-2.0 EPSF-1.2
%%BoundingBox: 0 0 192 150
%%Pages: 0
%%DocumentFonts: Times-Roman
%%EndComments
1 setlinejoin
0 0 translate
1 1 scale
0.3 setlinewidth

%string functions
newpath /Times-Roman findfont 9 scalefont setfont
/xjright {dup stringwidth pop neg 0 rmoveto} def
/xjcenter {dup stringwidth pop 2 div neg 0 rmoveto} def
/yjtop {0 -7 rmoveto} def
/yjcenter {0 -3.5 rmoveto} def

%transformation
/xscale 150 def
/xoff  30 def
/yscale 6 def
/yoff  17 def

/tf      {exch 1 sub xscale mul xoff add exch yscale mul yoff
         add} def
/mt      {tf moveto} def
/lt      {tf lineto} def

%def ticks
/xtick {1 sub xscale mul xoff add yoff moveto 0 -4 rlineto
         0 -2 rmoveto xjcenter yjtop show} def
/ytick {yscale mul yoff add xoff exch moveto -4 0 rlineto
         -2 0 rmoveto yjcenter xjright show} def

%axes
xoff yoff moveto
xoff yoff yscale 20 mul add lineto
stroke
xoff yoff moveto
xoff xscale add yoff lineto
stroke
```

```
(1.25)  1.25 xtick
(1.5)   1.50 xtick
(1.75)  1.75 xtick

(5%)    5    ytick
(10%)   10   ytick
(15%)   15   ytick

1 20 mt 0 4 rmoveto xoff 2 sub neg 0 rmoveto
        (misprediction rate) show
2 2 mt 10 0 rmoveto (code size) xjright show

  1       13.2648      mt
  2       13.2648      lt
stroke

 1.0000 17.2086  mt
 1.0063 16.1894  lt
 1.0125 15.0748  lt
 1.0188 14.6029  lt
 1.0251 14.4566  lt
 1.0314 14.2689  lt
 1.0376 14.1226  lt
 1.0804 14.1226  lt

 stroke
```