`See-code` und Code-Erzeugung in Gforth 1.0

M. Anton Ertl, TU Wien

See und simple-see

```
: bar if 5 then ;
```

```
see bar
: bar
  IF     #5
  THEN ;
```

```
simple-see bar
$3F9A20A540 ?branch
$3F9A20A548 <bar+$20>
$3F9A20A550 lit
$3F9A20A558 #5
$3F9A20A560 ;s
```

See-code

```
see-code bar
$3F9A20A540 ?branch    1->1
$3F9A20A548 <bar+$20>
  3F99EBAC54:   addi    sp,sp,8
  3F99EBAC56:   mv      spB,spA
  3F99EBAC58:   ld      spA,$0(sp)
  3F99EBAC5C:   ld      a5,$0(ip)
  3F99EBAC60:   addi    a6,ip,$8
  3F99EBAC64:   bnez    spB,3F99EBAC6E
  3F99EBAC66:   ld      a4,$0(a5)
  3F99EBAC68:   addi    ip,a5,$8
  3F99EBAC6C:   jr      a4
  3F99EBAC6E:   addi    ip,a6,$8
$3F9A20A550 lit    1->1
$3F9A20A558 #5
  3F99EBAC72:   sd      spA,$0(sp)
  3F99EBAC76:   addi    sp,sp,-8
  3F99EBAC78:   ld      spA,$0(ip)
  3F99EBAC7C:   addi    ip,ip,10
$3F9A20A560 ;s    1->1
  3F99EBAC7E:   ld      a6,$0(rp)
  3F99EBAC82:   addi    rp,rp,8
  3F99EBAC84:   addi    ip,a6,$8
  3F99EBAC88:   ld      a4,$-8(ip)
  3F99EBAC8C:   jr      a4
```

Non-relocatable primitives

```
: erase 0 fill ;        see-code erase
                        $3F98F9E540 lit    1->1
                        $3F98F9E548 #0
                          3F98C4EC54:   sd      spA,$0(sp)
                          3F98C4EC58:   addi    sp,sp,-8
                          3F98C4EC5A:   ld      spA,$0(ip)
                          3F98C4EC5E:   addi    ip,ip,10
                          3F98C4EC60:   ld      a4,$-8(ip)
                          3F98C4EC64:   jr      a4
                        $3F98F9E550 fill
                        $3F98F9E558 ;s    1->1
                          3F98C4EC66:   ld      a6,$0(rp)
                          3F98C4EC6A:   addi    rp,rp,8
                          3F98C4EC6C:   addi    ip,a6,$8
                          3F98C4EC70:   ld      a4,$-8(ip)
                          3F98C4EC74:   jr      a4
```

Static Superinstructions

```
: foo < if 5 then ;

$3F98F9E588 < ?branch      1->1
$3F98F9E590 ?branch
$3F98F9E598 <foo+$28>
  3F98C4EC76:   ld      a4,$8(sp)
  3F98C4EC7A:   ld      a5,$8(ip)
  3F98C4EC7E:   addi    sp,sp,10
  3F98C4EC80:   addi    a6,ip,$10
  3F98C4EC84:   blt     a4,spA,$3F98C4EC94
  3F98C4EC88:   ld      a4,$0(a5)
  3F98C4EC8A:   ld      spA,$0(sp)
  3F98C4EC8E:   addi    ip,a5,$8
  3F98C4EC92:   jr      a4
  3F98C4EC94:   ld      spA,$0(sp)
  3F98C4EC98:   addi    ip,a6,$8

$3F98F9E5A0 lit     1->1
$3F98F9E5A8 #5
  3F98C4EC9C:   sd      spA,$0(sp)
  3F98C4ECA0:   addi    sp,sp,-8
  3F98C4ECA2:   ld      spA,$0(ip)
  3F98C4ECA6:   addi    ip,ip,10
$3F98F9E5B0 ;s     1->1
  3F98C4ECA8:   ld      a6,$0(rp)
  3F98C4ECAC:   addi    rp,rp,8
  3F98C4ECAE:   addi    ip,a6,$8
  3F98C4ECB2:   ld      a4,$-8(ip)
  3F98C4ECB6:   jr      a4
```

Constant-based optimization

```
: foo 2 pick ;
```

```
see foo
: foo
  third ;
```

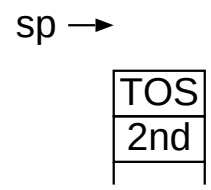
```
see-code foo
$3F7ADE05F8 third    1->1
  3F7AA90CAE:   ld      a5,$10(sp)
  3F7AA90CB2:   addi    ip,ip,8
  3F7AA90CB4:   addi    sp,sp,-8
  3F7AA90CB6:   sd      spA,$8(sp)
  3F7AA90CBA:   mv      spA,a5
$3F7ADE0600 ;s    1->1
  3F7AA90CBC:   ld      a6,$0(rp)
  3F7AA90CC0:   addi    rp,rp,8
  3F7AA90CC2:   addi    ip,a6,$8
  3F7AA90CC6:   ld      a4,$-8(ip)
  3F7AA90CCA:   jr      a4
```

Stack Caching

```
: foo swap ! ;
see-code foo
$3F7ADE0540 swap    1->2
  3F7AA90C54:   ld      spB,$8(sp)
  3F7AA90C58:   addi    ip,ip,8
  3F7AA90C5A:   addi    sp,sp,8
$3F7ADE0548 !    2->0
  3F7AA90C5C:   sd      spA,$0(spB)
  3F7AA90C60:   addi    ip,ip,8
$3F7ADE0550 ;s    0->1
  3F7AA90C62:   ld      spA,$8(sp)
  3F7AA90C66:   addi    sp,sp,8
  3F7AA90C68:   ld      a6,$0(rp)
  3F7AA90C6C:   addi    rp,rp,8
  3F7AA90C6E:   addi    ip,a6,$8
  3F7AA90C72:   ld      a4,$-8(ip)
  3F7AA90C76:   jr      a4
```

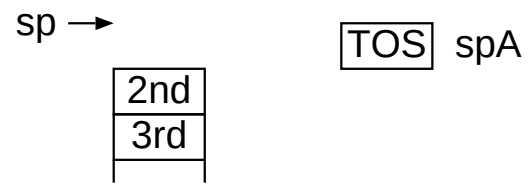
State 0

memory **registers**



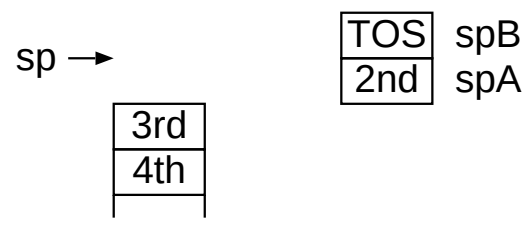
State 1

memory **registers**

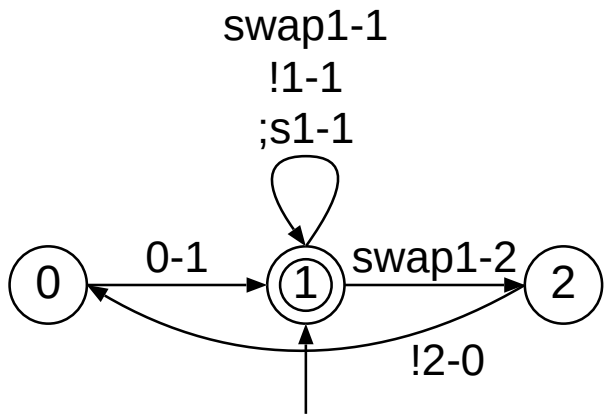


State 2

memory **registers**



Mögliche Implementation



- Idealerweise eine Sequenz auf einmal
  Aber: Decompiler schaut sich lieber jedes primitive einzeln an
- Möglichwerweise mehrdeutige Sequenzen
  Und wenn schon!
  Bei aktueller Codeerzeugung vermutlich eindeutig
- Kostet keinen Extra-Speicher
  Aber dafür Implementationsaufwand

Tatsächliche Implementierung

- Compiler speichert für jedes compilierte Primitive:
  Startadresse und Länge des native code
  Welches primitive oder superinstruction ist es
  Anfangs- und Endzustand des Stack Cache
- Der Decompiler braucht nur mehr nachschauen
- Kostet 16 Bytes pro compiliertem primitive
- ca. 500KB für das Gforth-Image
  fast soviel wie der Platz für den native code
- Einfacher zu implementieren

Zusammenfassung

- `see-code` zeigt native code
- zeigt static superinstructions
- zeigt stack caching
- Implementierung mit Zusatzinformation durch den Compiler