cc64 - Small C on the C64

EuroForth 2020

Philip Zembrod - pzembrod@gmail.com

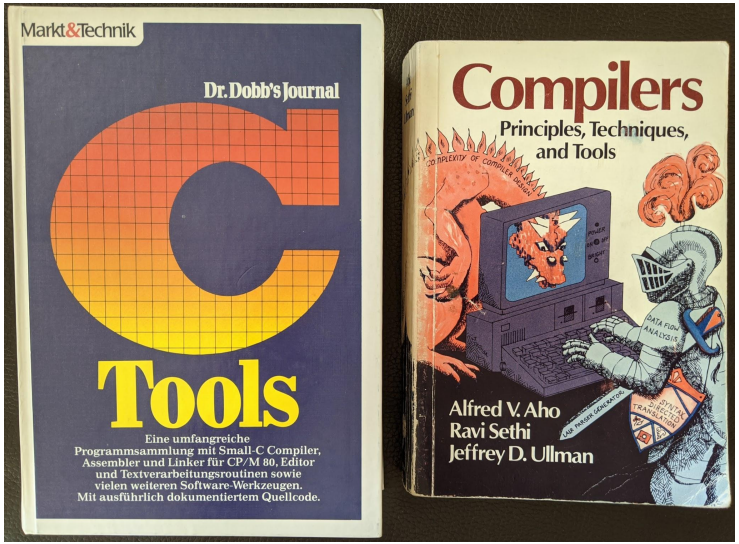
Why write a C compiler on the C64 in 1989?

I thought I could.

Long-standing interest in compilers

Inspirations: Dragon book +
Ron Cain's and James E. Hendrix'
8080 Small-C compiler

Existing C64 C & Pascal compilers
felt slow, opaque, impractical.



ASSI/M macros

“s65+”: macros + string functions
+ conditional assembly

Functions, local and global vars,
recursion, arrays, pointers

Syntax still assembly-ish
Next: real compiler

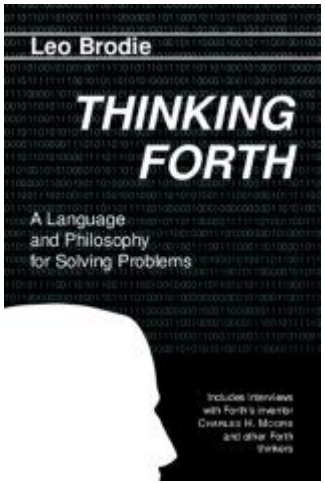
```
function findloc
char name^/start^
getargs name/start
 char cptr^
 move locptr,cptr
 decr cptr
 while "comp start,cptr",lt
  sub cptr^,cptr
  test stracmp(name/cptr/#maxnamlen)
  if eq
   sub #of_name,cptr
   return cptr
  endif
  sub #of_name+1,cptr
 endwhile
 return 0
endfunc
```

Why write it in Forth?

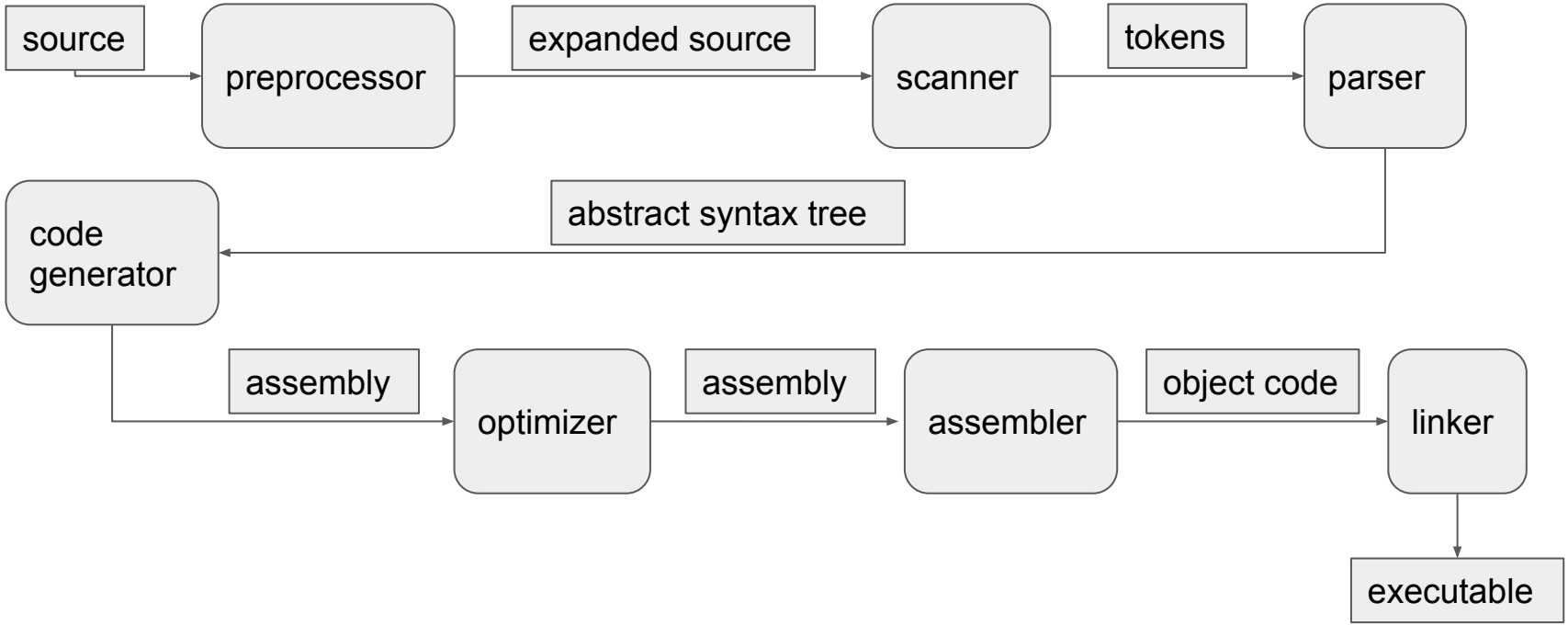
Why not in s65+? macro-expanding 100x: assembling slow

Encountered UltraForth

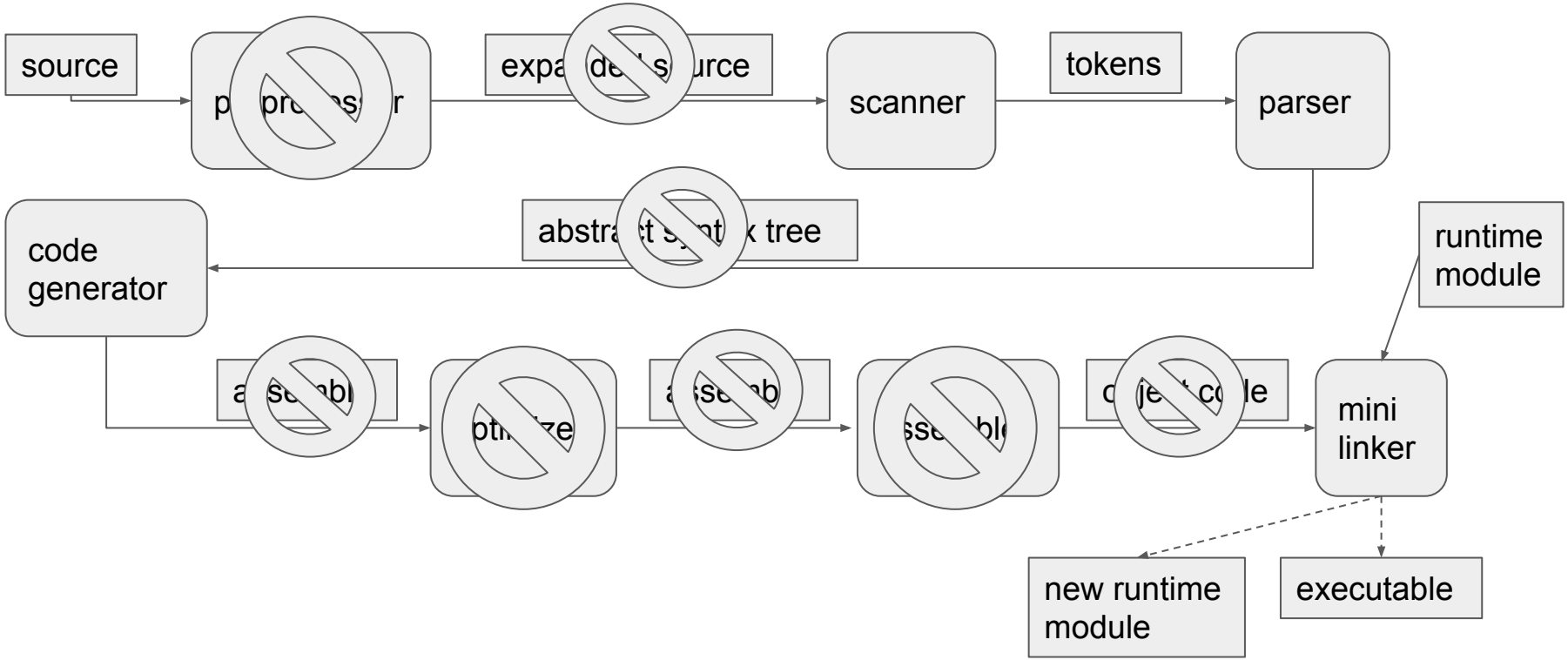
- compact code
- transparent, fast enough, practical
- fascinating
- … the known advantages of Forth ...



Typical classic C compiler



cc64 simplifications



cc64 parser

- Parsers
  - top-down vs bottom-up
  - hand-written vs grammar-generated
  - en.wikipedia.org/wiki/Comparison_of_parser_generators
  - Forth-generating parser generators are rare
- cc64: top-down recursive-descent parser
  - \+ easy to understand
  - – needs deep rstack
  - hand-crafted: more fun
- Limitations
  - K&R, int & char only, only 1 pointer level
  - not supported: unsigned, long, floats, void, struct, union, typedef, type casts, goto

Example: || and &&

```
: l-or ( -- obj )
     l-and
     BEGIN  <l-or> #oper# comes? WHILE
     do-l-or.1  l-and
     do-l-or.2  REPEAT ;

: l-and ( -- obj )
     bit-or
     BEGIN  <l-and> #oper# comes? WHILE
     do-l-and.1  bit-or
     do-l-and.2  REPEAT ;
```

Defining word

```
<or>  ' do-or
1  ' bit-xor  binary bit-or

<xor> ' do-xor
1  ' bit-and  binary bit-xor

<and> ' do-and
1  ' equal    binary bit-and

<==>  ' do-eq
<!=>  ' do-ne
2  ' comp     binary equal
```

```
<<>   ' do-lt
<<=>  ' do-le
<>>   ' do-gt
<>=>  ' do-ge
4  ' shift    binary comp

<<<>  ' do-shl
<>>>  ' do-shr
2  ' sum      binary shift

<+>   ' do-add
<->   ' do-sub
2  ' product  binary sum

<*>   ' do-mult
</>   ' do-div
<%>   ' do-mod
3  ' unary    binary product
```

Challenge: 6502

Only 3 registers: a, x, y, all 8 bit. 256 bytes hardware stack.
High-level languages be like :-(

Approach: virtual 16 bit stack machine with virtual a in 6502 a/x:

```
a&: .lda#     <& # lda  >& # ldx  ;a

a:  .pha      pha  txa  pha  ;a

a:  .sta-zp   $zp sta   $zp+1 stx ;a
```

Soft stack for local variables

Codegen do-xyz using code templates .xyz

```
   ' / ' .#div ' .div# ' .div
binop do-div

   ' mod ' .#mod ' .mod# ' .mod
binop do-mod

   ' * ' .mult# ' .mult# ' .mult
binop do-mult
```

```
\ divide a by zp, remainder in zp
a: (.divmod  $divmod jsr ;a

: .div#  .ldzp#          (.divmod ;
: .#div  .sta-zp .lda#   (.divmod ;
: .div   .sta-zp .pla    (.divmod ;

: .mod#  .div#   .lda-zp ;
: .#mod  .#div   .lda-zp ;
: .mod   .div    .lda-zp ;
```

Code templates

```
a&: .ldzp#  tay
  <& # lda  $zp   sta
  >& # lda  $zp+1 sta  tya  ;a

: (a: ( -- sys )
  create  here 0 c,  20
  assembler ;

: ;a ( sys -- )
  20 ?pairs
  current @ context !
  here over - 1- swap c! ;
```

```
: a, ( par I b -- step )
  dup $f and $7 =
    IF 2/ 2/ 2/ 2/ 2*
    atab + @ execute
    ELSE b, 2drop 1 THEN ;

: a&: (a: does> ( par pfa -- )
  count bounds DO
  dup I dup c@ a, +LOOP drop ;

: a: (a: does> ( pfa -- )
    count bounds
    DO 0 I dup c@ a, +LOOP ;
```

Other challenges

- Memory size
- Keep overview of source code
  - 3 disks 170 kB each
  - -> many printouts, 12 screens per page
- Moving screens around
- C64 charset: \^_{|}~ missing
- Text editor: missing
- Testing … 😓

… graduation … pause 1996-2019 ...

Restart with emulator

- From screen to stream sources
  - ufscr2file.c, ascii2petscii.c, petscii2ascii.c
  - Simple INCLUDE implementation
- VICE emulator with 4 disk drives
  - 1 Linux-dir-backed
  - 3 d64 disk images
- Automate build & tests
  - Script VICE with --keybuf param
  - Terminate VICE when file “notdone” deleted
  - Re-learn GnuMake
- Tests, tests, tests, some fixes, release ... github.com/pzembrod/cc64

VolksForth cross-pollination

- Ported to C64/C16 VolksForth:
  - VICE automation infra
  - INCLUDE
- Automated tests
- Automated target compile
- Integrated INCLUDE into core
- Binary flavours full and lite: with and without block words
- Fixed C16-32k
- Took lite binaries back to cc64
  - -> cc64 on c16-64k/Plus4!

Further ideas

- Port to Commander x16
- Port to Linux (host)
- cc64 unit/component tests
- ANSI function param syntax
- Profile memory and CPU. Why is so slow?
- Output assembler source
- Relocating symbolic linker
- C library

Thank you for listening!

Questions?