Pnut: A C Transpiler Targeting Human Readable POSIX Shell

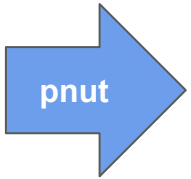
Laurent Huberdeau, Cassandre Hamel, Marc Feeley, Stefan Monnier
Université de Montréal
October 20th 2024

Structure

1. What is pnut?
2. Motivation
3. Implementation and challenges
4. Does it work?

Human Readable POSIX Shell

```c
void puts(char *s) {
  while (*s != 0) {
    putchar(*s);
    s += 1;
  }
}
```



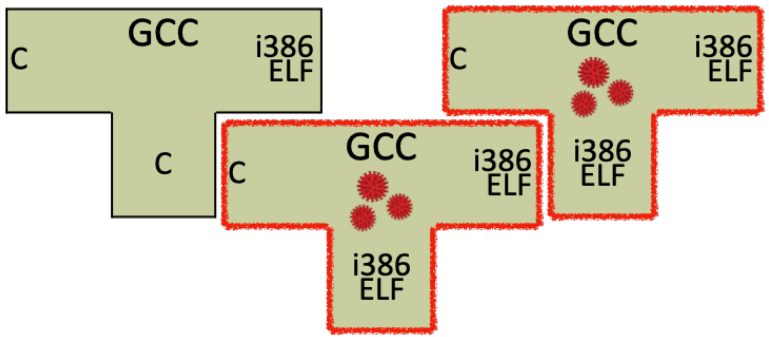

```
_puts() { let s $2
  while [ $((_$s)) != 0 ]; do
    _putchar __ $((_$s))
    : $((s += 1))
  done
  endlet $1 s
}
```

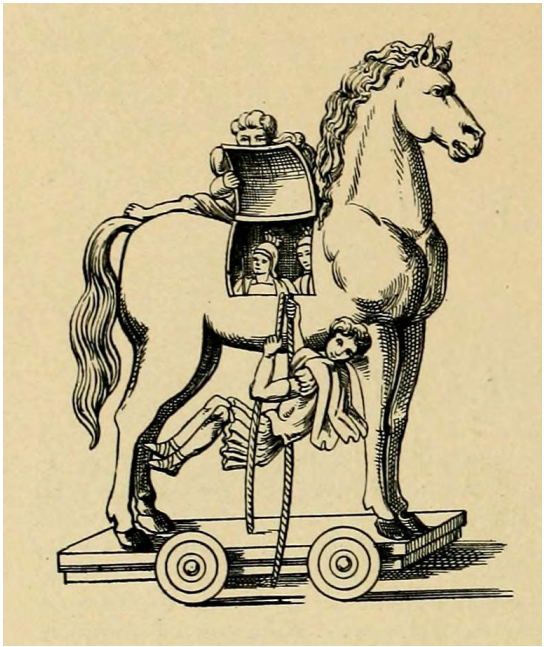
POSIX Shell

But why?

Reproducible Builds and Bootstrapping!

- Ken Thompson Turing Award lecture
  - Reflections on Trusting Trust
- What good is source code if you can’t trust the build tools?

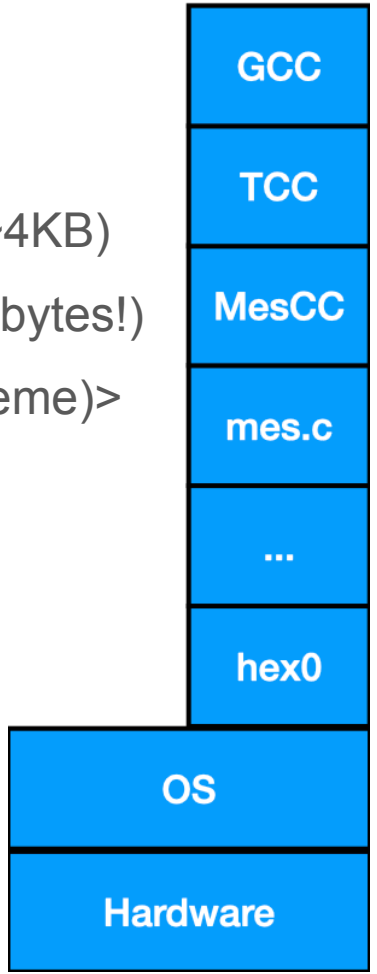




Related Work

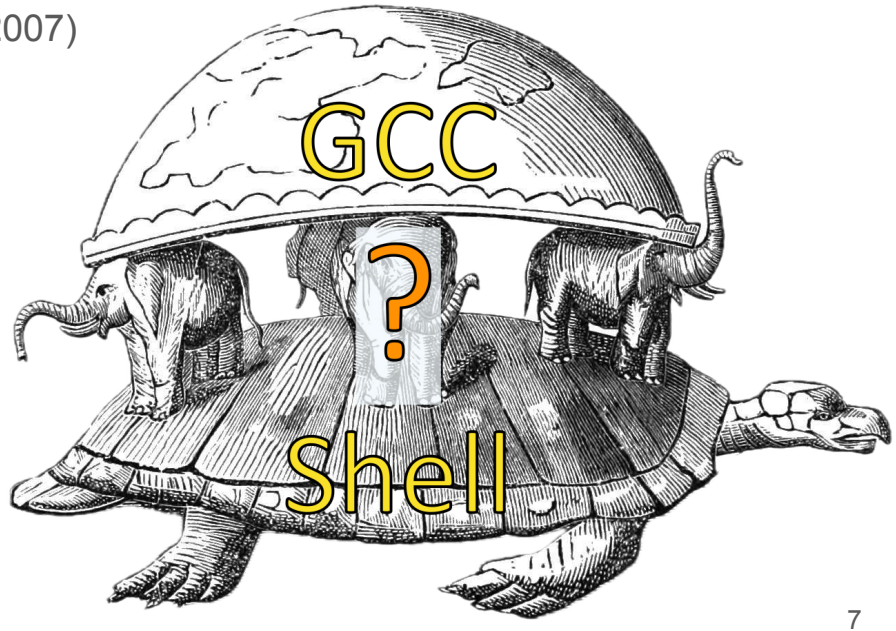
- **Live-bootstrap**: Minimal bootloader, kernel, shell (~4KB)
- **hex0**: minimal and auditable binary seed (Only 190 bytes!)
- **MesCC**: <Scheme interpreter (C), C Compiler (Scheme)>

Multiple independent bootstrap paths are good!



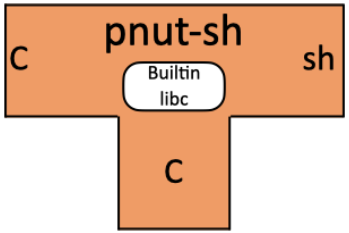
POSIX Shell for Bootstrapping

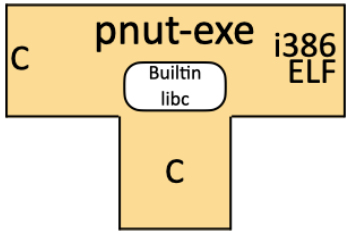
- Cross platform and diverse implementations
  - Many implementations: bash (1989), ksh (1983), ash (1989), dash (1997), zsh (1990), yash (2007)
  - On almost all platforms
  - Likely developed independently
  - ⇒ **Diverse Double-Compilation!**
- Auditable seed
  - We can use an untrusted compiler as long as we check its output.

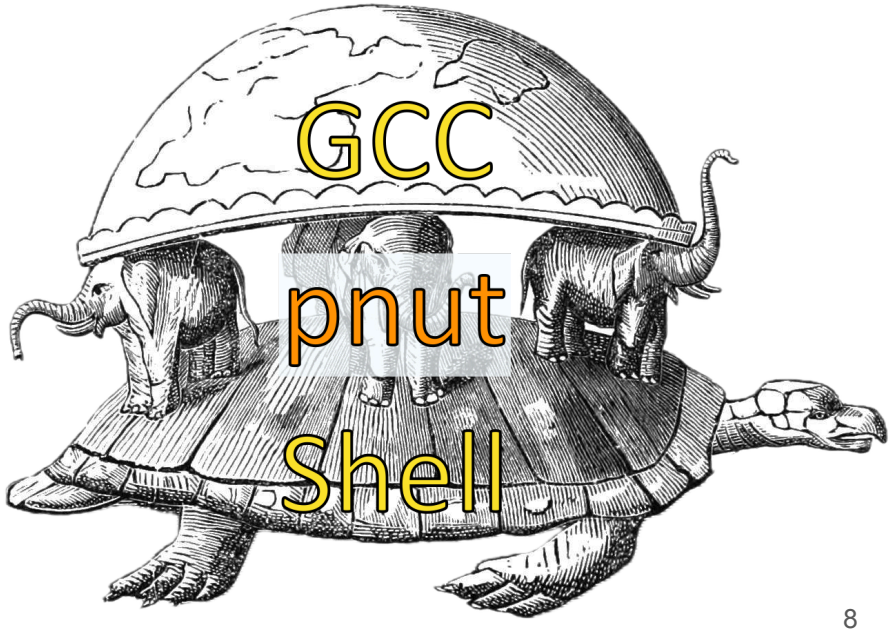


The plan

1. **pnut-sh**: C to **Human readable** POSIX Shell
2. **pnut-exe**: C to binary compiler







Compilation Steps

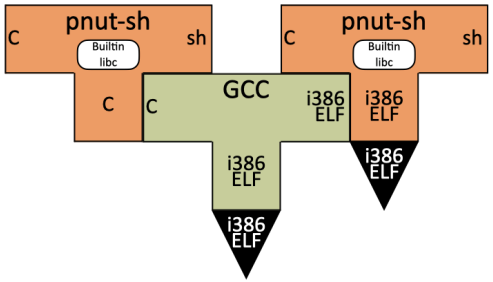
1. `gcc.exe pnut-sh.c > pnut-sh.exe`
2. `pnut-sh.exe pnut-sh.c > pnut-sh.sh`

3. `pnut-sh.sh pnut-exe.c > pnut-exe.sh`
4. `pnut-exe.sh pnut-exe.c > pnut-exe.exe`
5. `pnut-exe.exe tcc.c > tcc.exe`
6. `tcc.exe gcc.c > gcc.exe`

Compilation Steps

1. `gcc.exe pnut-sh.c > pnut-sh.exe`
2. `pnut-sh.exe pnut-sh.c > pnut-sh.sh`

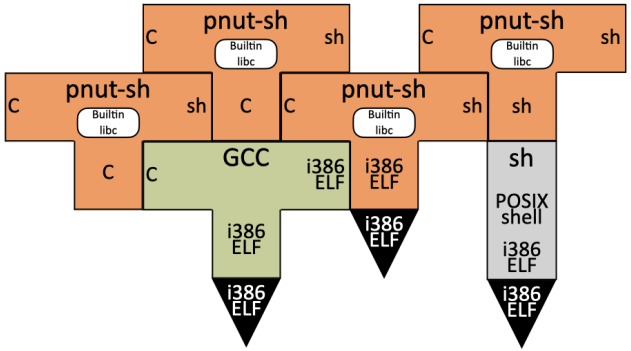
3. `pnut-sh.sh pnut-exe.c > pnut-exe.sh`
4. `pnut-exe.sh pnut-exe.c > pnut-exe.exe`
5. `pnut-exe.exe tcc.c > tcc.exe`
6. `tcc.exe gcc.c > gcc.exe`



Compilation Steps

1. `gcc.exe pnut-sh.c > pnut-sh.exe`
2. `pnut-sh.exe pnut-sh.c > pnut-sh.sh`

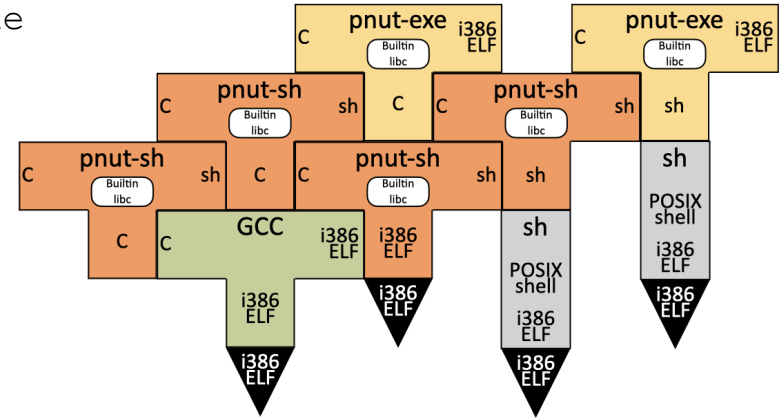
3. `pnut-sh.sh pnut-exe.c > pnut-exe.sh`
4. `pnut-exe.sh pnut-exe.c > pnut-exe.exe`
5. `pnut-exe.exe tcc.c > tcc.exe`
6. `tcc.exe gcc.c > gcc.exe`



Compilation Steps

1. `gcc.exe pnut-sh.c > pnut-sh.exe`
2. `pnut-sh.exe pnut-sh.c > pnut-sh.sh`

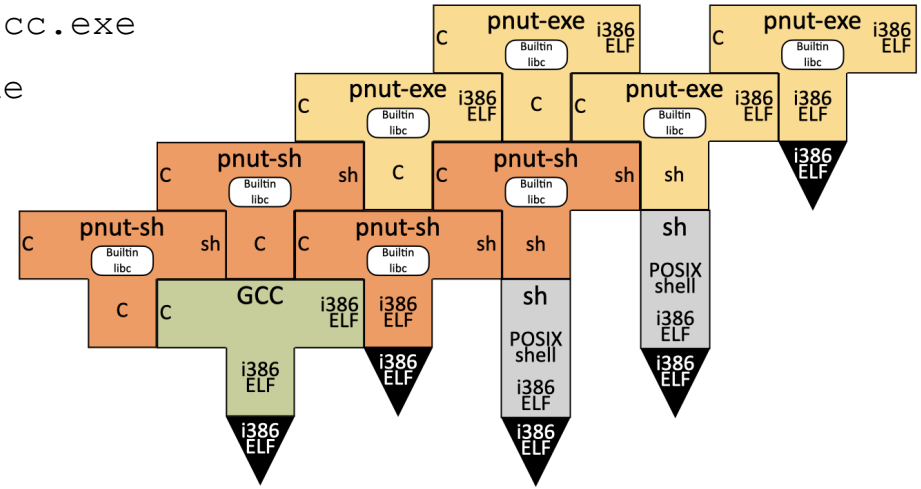
3. `pnut-sh.sh pnut-exe.c > pnut-exe.sh`
4. `pnut-exe.sh pnut-exe.c > pnut-exe.exe`
5. `pnut-exe.exe tcc.c > tcc.exe`
6. `tcc.exe gcc.c > gcc.exe`



Compilation Steps

1. `gcc.exe pnut-sh.c > pnut-sh.exe`
2. `pnut-sh.exe pnut-sh.c > pnut-sh.sh`

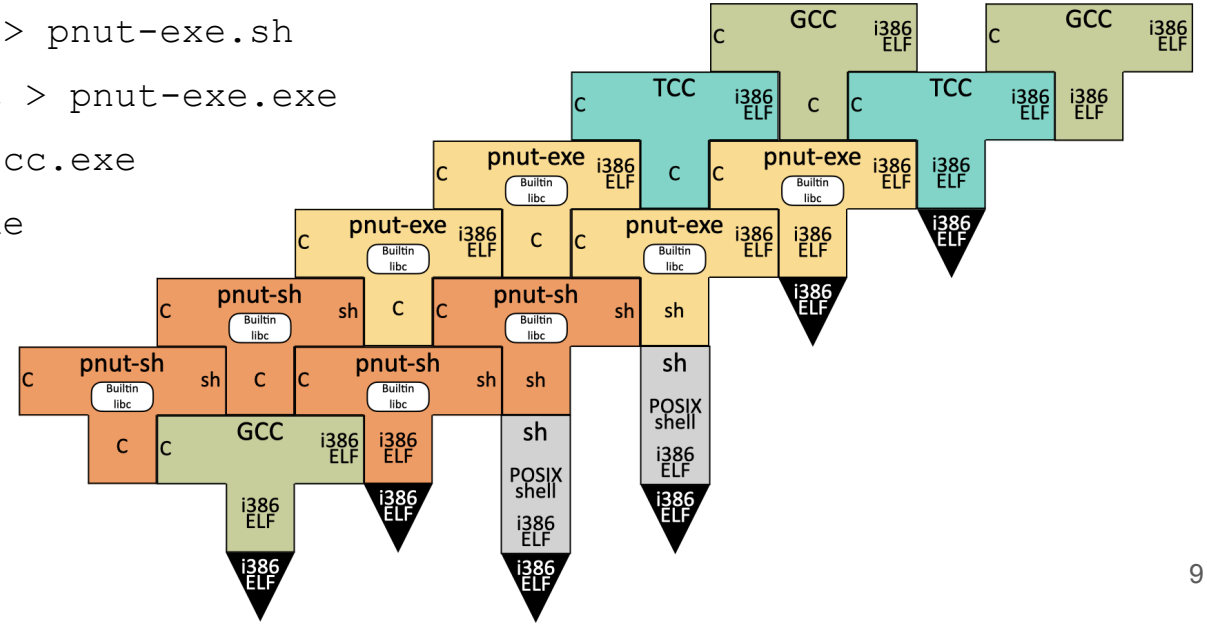
3. `pnut-sh.sh pnut-exe.c > pnut-exe.sh`
4. `pnut-exe.sh pnut-exe.c > pnut-exe.exe`
5. `pnut-exe.exe tcc.c > tcc.exe`
6. `tcc.exe gcc.c > gcc.exe`



Compilation Steps

1. `gcc.exe pnut-sh.c > pnut-sh.exe`
2. `pnut-sh.exe pnut-sh.c > pnut-sh.sh`

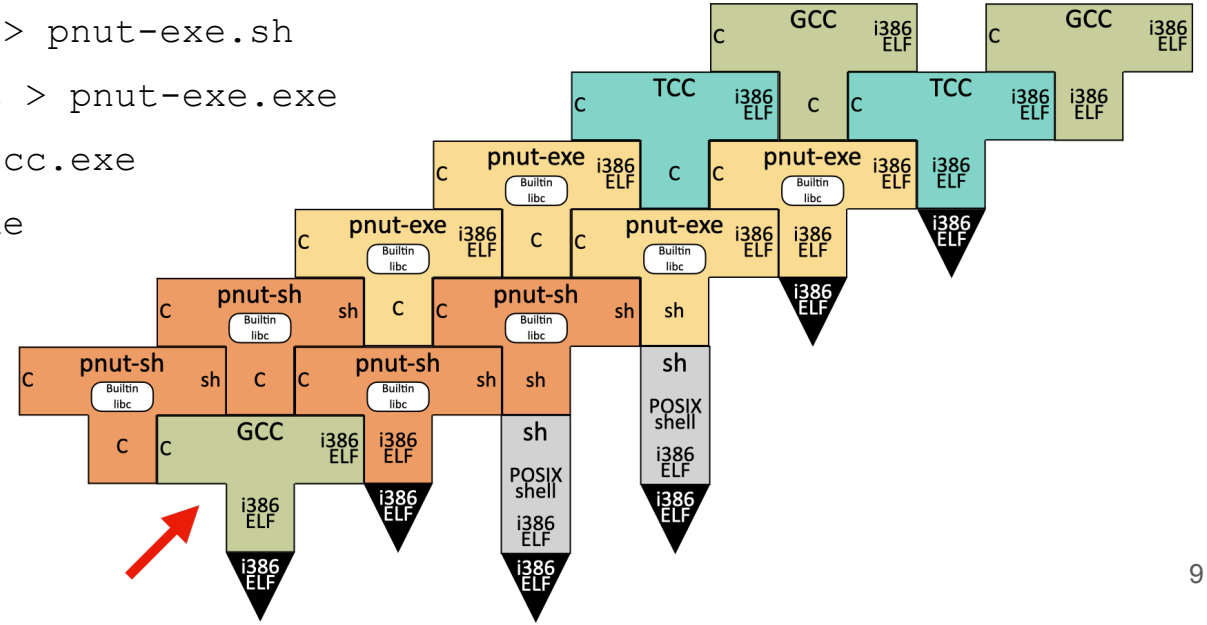
3. `pnut-sh.sh pnut-exe.c > pnut-exe.sh`
4. `pnut-exe.sh pnut-exe.c > pnut-exe.exe`
5. `pnut-exe.exe tcc.c > tcc.exe`
6. `tcc.exe gcc.c > gcc.exe`



Compilation Steps

1. gcc.exe pnut-sh.c > pnut-sh.exe
2. pnut-sh.exe pnut-sh.c > pnut-sh.sh

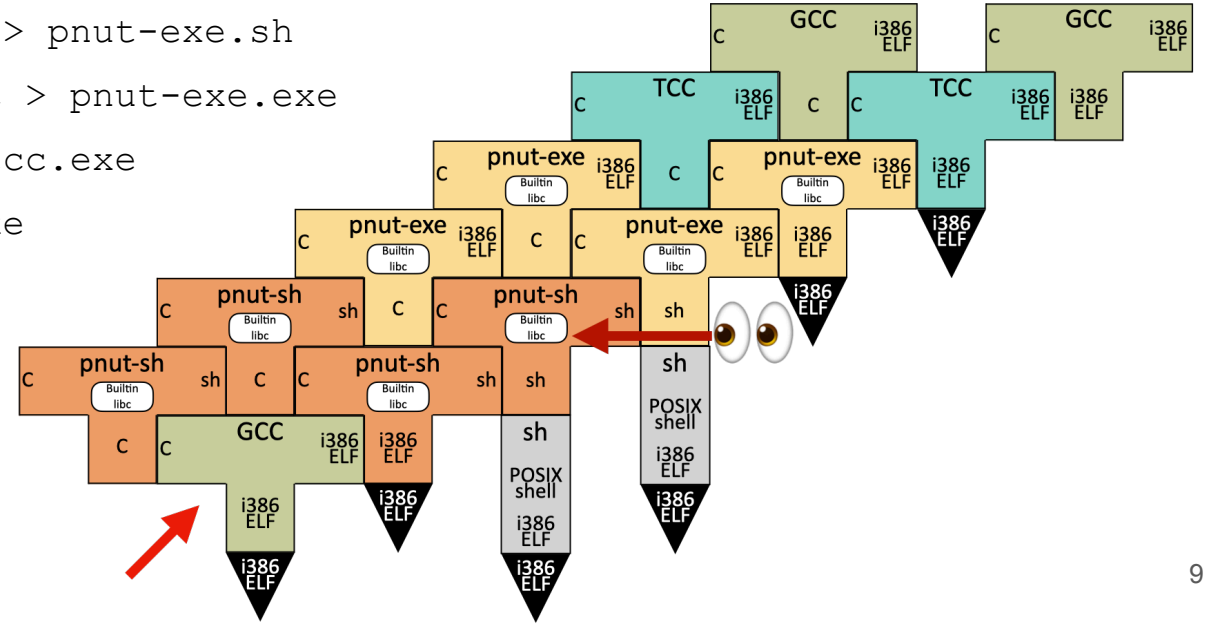
3. pnut-sh.sh pnut-exe.c > pnut-exe.sh
4. pnut-exe.sh pnut-exe.c > pnut-exe.exe
5. pnut-exe.exe tcc.c > tcc.exe
6. tcc.exe gcc.c > gcc.exe



Compilation Steps

1. `gcc.exe pnut-sh.c > pnut-sh.exe`
2. `pnut-sh.exe pnut-sh.c > pnut-sh.sh`

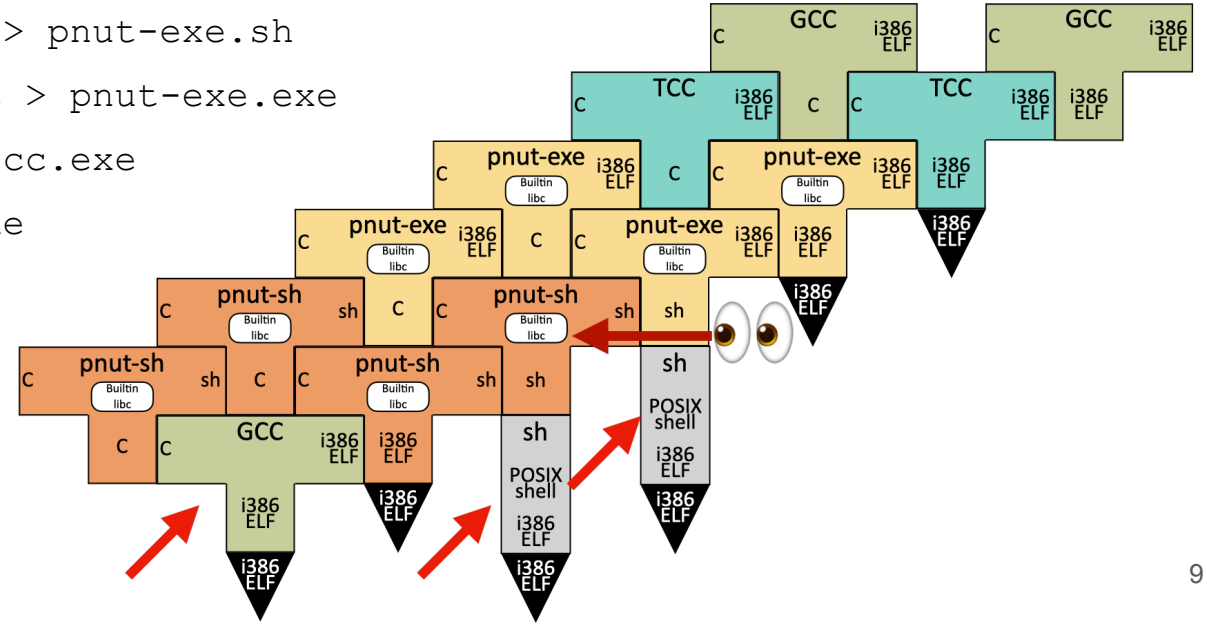
   Read and compare pnut.sh to pnut-sh.c
3. `pnut-sh.sh pnut-exe.c > pnut-exe.sh`
4. `pnut-exe.sh pnut-exe.c > pnut-exe.exe`
5. `pnut-exe.exe tcc.c > tcc.exe`
6. `tcc.exe gcc.c > gcc.exe`



Compilation Steps

1. `gcc.exe pnut-sh.c > pnut-sh.exe`
2. `pnut-sh.exe pnut-sh.c > pnut-sh.sh`

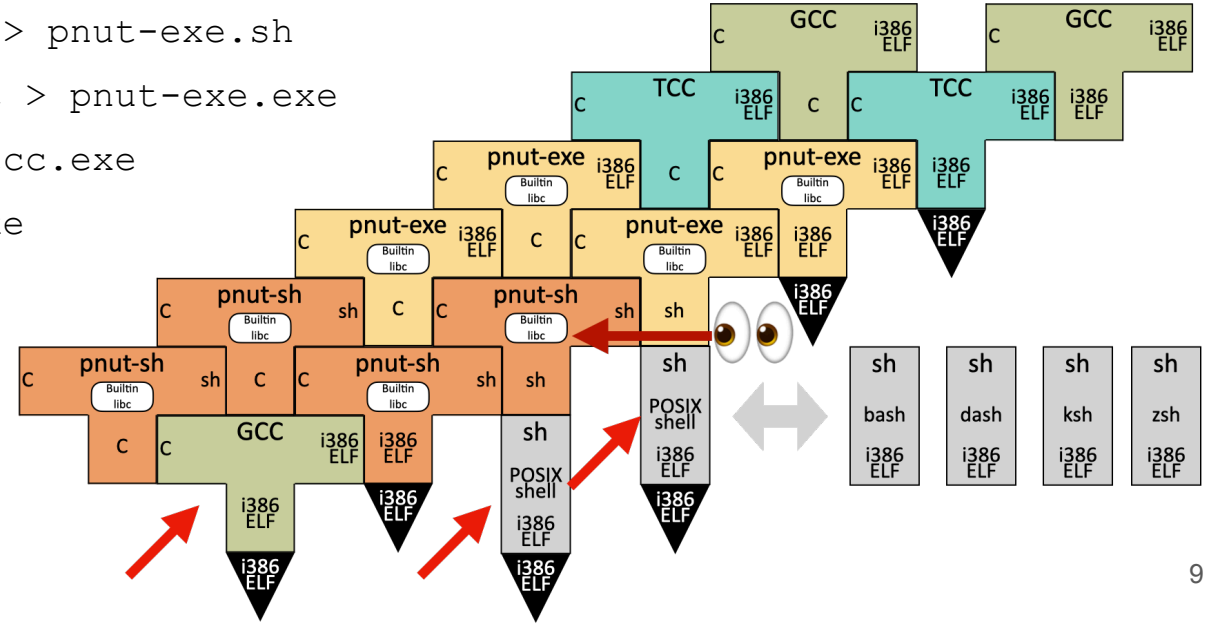
   Read and compare pnut.sh to pnut-sh.c
3. `pnut-sh.sh pnut-exe.c > pnut-exe.sh`
4. `pnut-exe.sh pnut-exe.c > pnut-exe.exe`
5. `pnut-exe.exe tcc.c > tcc.exe`
6. `tcc.exe gcc.c > gcc.exe`



Compilation Steps

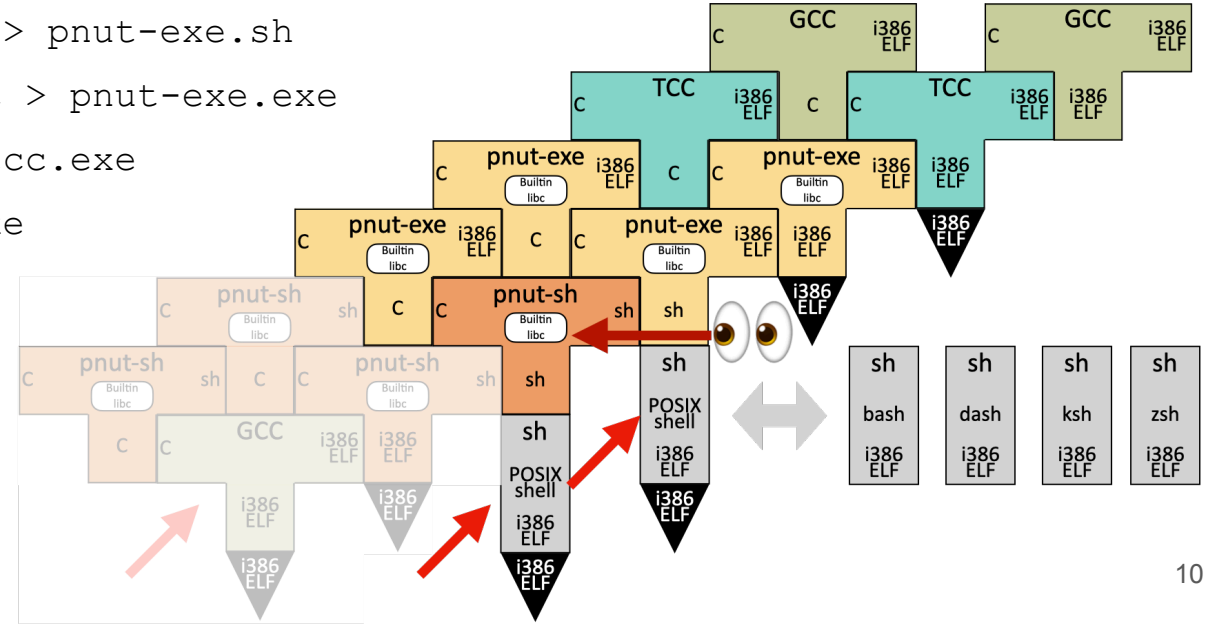
1. `gcc.exe pnut-sh.c > pnut-sh.exe`
2. `pnut-sh.exe pnut-sh.c > pnut-sh.sh`

   Read and compare pnut.sh to pnut-sh.c
3. `pnut-sh.sh pnut-exe.c > pnut-exe.sh`
4. `pnut-exe.sh pnut-exe.c > pnut-exe.exe`
5. `pnut-exe.exe tcc.c > tcc.exe`
6. `tcc.exe gcc.c > gcc.exe`



Compilation Steps

1. `pnut-sh.sh pnut-exe.c > pnut-exe.sh`
2. `pnut-exe.sh pnut-exe.c > pnut-exe.exe`
3. `pnut-exe.exe tcc.c > tcc.exe`
4. `tcc.exe gcc.c > gcc.exe`



Implementation Challenges

Implementation Challenges

- No arrays/data structures
- Only procedures, not functions
  - No return value
  - No local variables
- Character-based I/O

Design Goals

- Auditable and human readable:
  - Preserve the structure of the C source code
  - No use of **eval**
  - Avoid constants and magic numbers
- No external utilities (such as bc, od, …)
- Sufficiently fast for bootstrapping

Arithmetic Expansions and Arrays

Arithmetic Expansions and Arrays

- Evaluates signed integer expressions: `$(( 5 * 10 ))`

Arithmetic Expansions and Arrays

- Evaluates signed integer expressions: `$(( 5 * 10 ))`
- Can refer to variables: `$(( a + 10 ))`

Arithmetic Expansions and Arrays

- Evaluates signed integer expressions: `$(( 5 * 10 ))`
- Can refer to variables: `$(( a + 10 ))`
- Supports all C operators, including assignments: `$(( a += b ))`

Arithmetic Expansions and Arrays

- Evaluates signed integer expressions: `$(( 5 * 10 ))`
- Can refer to variables: `$(( a + 10 ))`
- Supports all C operators, including assignments: `$(( a += b ))`
- Memory locations are variables _1, _2, _3, …

Arithmetic Expansions and Arrays

- Evaluates signed integer expressions: `$(( 5 * 10 ))`
- Can refer to variables: `$(( a + 10 ))`
- Supports all C operators, including assignments: `$(( a += b ))`
- Memory locations are variables _1, _2, _3, …
- Index with nesting: `$(( x = _$((1 + 1)) )) # x = *(2)`

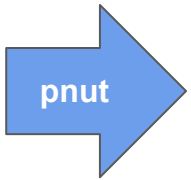
Arithmetic Expansions and Arrays

- Evaluates signed integer expressions: `$(( 5 * 10 ))`
- Can refer to variables: `$(( a + 10 ))`
- Supports all C operators, including assignments: `$(( a += b ))`
- Memory locations are variables _1, _2, _3, …
- Index with nesting: `$(( x = _$((1 + 1)) )) # x = *(2)`
- `: $(( expr ))` ignores the result of the expr

Arithmetic Expansions and Arrays

- Evaluates signed integer expressions: `$(( 5 * 10 ))`
- Can refer to variables: `$(( a + 10 ))`
- Supports all C operators, including assignments: `$(( a += b ))`
- Memory locations are variables _1, _2, _3, …
- Index with nesting: `$(( x = _$((1 + 1)) )) # x = *(2)`
- `: $(( expr ))` ignores the result of the expr

```
void puts(char *s) {
  while (*s != 0) {
    putchar(*s);
    s += 1;
  }
}
```




```
_puts() { let s $2
  while [ $((_$s)) != 0 ]; do
    _putchar __ $((_$s))
    : $((s += 1))
  done
  endlet $1 s
}
```

Implementation Challenges

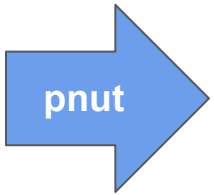
- No arrays/data structures ✅
- Only procedures, not functions
  - No return value
  - No local variables
- Character-based I/O

Returning values from functions

C-like solution

```
int meaning_of_life() {
  return 42;
}

answer = meaning_of_life();
```




```
_meaning_of_life() {
  : $(($1 = 42))
}

_meaning_of_life answer
```

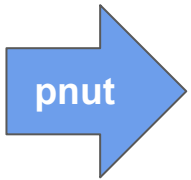
POSIX Shell

Local Variables and Parameters

- Callee-save calling convention
  - **let** = push
  - **endlet** = pop

```
int add2(int a, int b) {
  return a + b;
}

result = add2(10, 20);
```



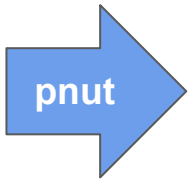

```
_add2() { let a $2; let b $3
  : $(($1 = a + b))
  endlet $1 b a
}

_add2 result 10 20
```

Faster Local variables and Parameters

- Use function positional parameters to store values
- **2x** faster but harder to read

```c
int add2(int a, int b) {
  return a + b;
}
```



POSIX Shell

```sh
_add2() { # a: $2, b: $3
  set $@ $a $b
  a=$2
  b=$3
  : $((__tmp = a + b))
  : $((a = $4)) $((b = $5))
  : $(($1 = __tmp))
}
```

Implementation Challenges

- No arrays/data structures ✅
- Only procedures, not functions ✅
  - No return value
  - No local variables
- Character-based I/O

Character-based I/O

- Shell has file descriptors
- **read** builtin reads lines until ‘\n’
  - Shell strings are delimited with \0 => cannot read NUL bytes
- Output any byte with **printf** builtin
- UNIX `open/read/write`

```
exec 3< a.txt # Open a.txt in read mode with fd 3

IFS=                          # Read until \n
while read -r line <&3; do # Read line from a.txt
  printf "%s\n" "$line"     # Write line to stdout
done;

exec 3<&-  # Close file a.txt
```

Implementation Challenges

- No arrays/data structures ✅
- Only procedures, not functions ✅
  - No return value
  - No local variables
- Character-based I/O ✅

Design Goals

Design Goals

- Auditable and human readable:
  - Preserve the structure of the C source code ✅
  - No use of **eval** ✅
  - Avoid constants and magic numbers

Design Goals

- Auditable and human readable:
  - Preserve the structure of the C source code ✅
  - No use of **eval** ✅
  - Avoid constants and magic numbers
- No external utilities (such as bc, od, …) ✅

Design Goals

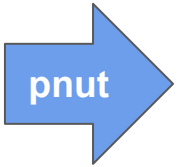
- Auditable and human readable:
  - Preserve the structure of the C source code ✅
  - No use of **eval** ✅
  - Avoid constants and magic numbers
- No external utilities (such as bc, od, …) ✅
- Sufficiently fast for bootstrapping

Avoiding magic numbers

- From character constants and structures

```
struct Pair {
  char fst;
  int snd;
};

struct Pair *p;
p = malloc(sizeof(struct Pair));
p->fst = 'a';
p->snd = 10;
```



POSIX Shell

```
# Pair struct member declarations
readonly __fst=0
readonly __snd=1
readonly __sizeof__Pair=2

# Character constants
readonly __a__=97

let p
_malloc p $((__sizeof__Pair))
: $((_$((p + __fst)) = __a__))
: $((_$((p + __snd)) = 10))
```

String literals

- Shell string unpacked to memory on first use at site of usage

```
void print_hex_digit(int digit) {
  putchar("0123456789abcdef"[digit & 15]);
}
```


```
_print_hex_digit() { let digit $2
  defstr __str_1 "0123456789abcdef"
  _putchar __ $((_$((__str_1 + (digit & 15)))))
  endlet $1 digit
}
```

Does it work?

C language support

Required for TCC:

- More numeric types (float, unsigned)
- Bitfields
- Functions pointers
- Variadic functions

pnut-exe:

- `goto`
- `switch` fallthrough
- Variable shadowing
- Address of (`&`) operator
- Stack allocated arrays and structures

pnut-sh:

- Function declarations
- `if/switch`
- `for/while/do_while`
- `struct/enum`
- Pointer arithmetic
- `malloc/free`
- `read/write/exit`
- C Preprocessor

Results

- Output shell script are ~35% larger than C code
- Can compile small programs such as the **C4 compiler**
- Compatible with versions of bash back to 2.05b (2002)

pnut-exe.c (5k LOC) compilation times with pnut-sh.sh

Shell	Bootstrap time
ksh	24.6 s
dash	43.8 s
bash	60.7 s
yash	61.9 s
zsh	664.2 s

Conclusion

- POSIX shell is a viable foundation for reproducible builds
- Work on bootstrapping TCC (and then GCC) is ongoing
- **Try pnut: https://pnut.sh**

Fibonacci Compile Copy

```c
/*
  Print the first 20 numbers of the Fibonacci sequence
*/

#include <stdio.h>

int fib(int n) {
  if (n < 2) {
    return n;
  } else {
    return fib(n - 1) + fib(n - 2);
  }
}

void main() {
  int n;
  int i = 0;
  while (i < 20) {
    n = fib(i);
    printf("fib(%d) = %d\n", i, n);
```

Shell Copy

```sh
#!/bin/sh
set -e -u

: $((__t2 = __t1 = n = 0))
_fib() { let n $2
  let __t1; let __t2
  if [ $n -lt 2 ] ; then
    : $(($1 = n))
  else
    _fib __t1 $((n - 1))
    _fib __t2 $((n - 2))
    : $(($1 = __t1 + __t2))
  fi
  endlet $1 __t2 __t1 n
}

: $((i = n = 0))
_main() {
  let n; let i
  i=0
```

End

Environment size (1)

Table 1. Time in seconds to initialize an array of size N

N	ksh	dash	bash	yash	zsh
10000	0.03	0.03	0.04	0.05	0.05
100000	0.25	1.27	0.44	0.50	0.44
500000	1.21	34.96	2.14	2.62	2.26
1000000	2.47	312.49	4.24	5.09	4.60

Environment size (2)

Table 2. Time in seconds to convert characters using a subshell compared to the fast method with different environment size.

	ksh	dash	bash	yash	zsh
subshell (1000)	0.06	3.12	6.23	4.99	5.21
subshell (10000)	0.06	3.67	8.80	5.70	5.68
subshell (100000)	0.06	6.77	22.77	11.59	9.53
fast (1000)	0.01	0.01	0.04	0.15	0.03
fast (10000)	0.02	0.03	0.04	0.15	0.04
fast (100000)	0.02	0.03	0.04	0.15	0.05

Shell character to ASCII code

```
__c2i_0=48; ...; __c2i_z=122

__c="A" # Convert character to character code
case $__c in
 [a-zA-Z0-9]) __code=$((__c2i_$__c)) ;;
 " ") __code=32 ;;
 *) __code=$(LC_CTYPE=C printf %d "'$__c") ;;
esac
```

pnut-sh.sh benchmarks

	ksh	dash	bash	yash	zsh	pnut.exe (GCC)	pnut.exe (pnut-exe)
hello.c (5 LOC)	0.028s	0.019s	0.059s	0.059s	0.085s	0.001s	0.004s
fib.c (19 LOC)	0.089s	0.059s	0.190s	0.197s	0.314s	0.001s	0.005s
cat.c (41 LOC)	0.183s	0.119s	0.407s	0.427s	0.666s	0.001s	0.018s
wc.c (64 LOC)	0.281s	0.188s	0.655s	0.673s	1.246s	0.001s	0.021s
sha256sum.c (233 LOC)	1.512s	1.105s	4.058s	4.078s	9.007s	0.001s	0.034s
c4.c (529 LOC)	5.494s	5.274s	13.956s	14.259s	67.350s	0.002s	0.085s
repl.c (814 LOC)	13.361s	13.650s	23.543s	19.614s	105.406s	0.002s	0.079s
pnut.c (6698 LOC)	31.008s	72.997s	76.548s	76.455s	1094.586s	0.006s	0.469s
pnut-exe.c (6296 LOC)	24.991s	44.610s	61.606s	62.309s	660.516s	0.005s	0.368s

Reverse Polish Notation Calculator in Shell


```
stack=""                                # Initialize empty stack
push() { stack="$1 $stack"; }           # String concat operand to stack
pop() { # $@ = stack, uses word splitting to extract first 2 elements
  val2="$1"; val1="$2"; shift 2
  stack="$@"
}

while IFS= read -r opnd; do             # Read operand from stdin line by line
  case $opnd in
    '+'|'-'|'*'|'/')
      pop $stack;                       # Pop 2 values from stack
      push $(($val1 $opnd $val2))       # Perform operation and push result
      ;;
    *) push "$opnd" ;;
  esac
done

echo "Result: $stack"
```

let and endlet definition

```
__SP=0
let() { # $1: variable name, $2: value
 : $((__SP += 1)) $((__$__SP=$1)) # Push
 : $(($1=${2-0}))                 # Init
}

endlet() { # $1: return variable
           # $2...: function local
variables
  __ret=$1 # Don't overwrite return value
  : $((__tmp = $__ret))
  while [ $# -ge 2 ]; do
    : $(($2 = __$__SP)) $((__SP -= 1));
   shift;
 done
 : $(($__ret=__tmp))  # Restore return val
}
```

```
foo() { let argA $2; let argB $3
 let x; let y
 # {do something}
 endlet $1 y x argB argA
}
```