# Compiler Construction

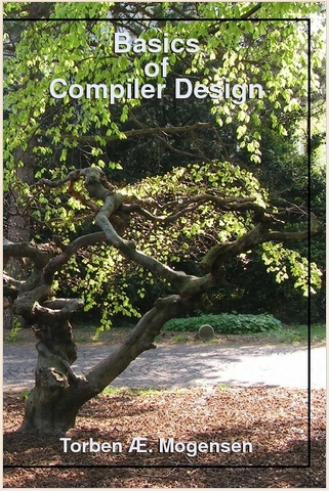
## Lecture 16

## Bootstrapping

Jeremy Yallop, Lent 2026

jeremy.yallop@cl.cam.ac.uk

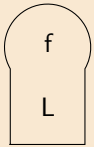
# Recommended book




*Chapter 13 of*
**Basics of Compiler Design**
Torben Ægidius Mogensen
`http://hjemmesider.diku.dk/~torbenm/Basics/`

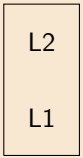
# Notation

A program




Computes function f written in language L

An interpreter

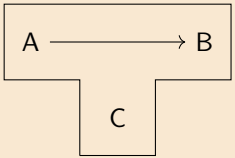



Interprets language L2 written in language L1

A machine


Executes code in language L

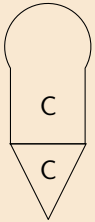
A compiler




Translates language A into language B
Written in language C

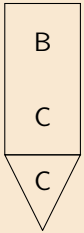
# Examples

To execute a program



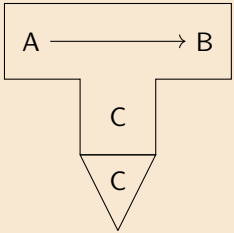

we run it on a machine

To execute an interpreter




we run it on a machine

To execute a compiler



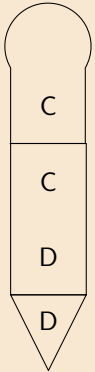

we run it on a machine

# Interpreting a program

Run a program
written in language C

on an interpreter for C
written in language D

on a D machine



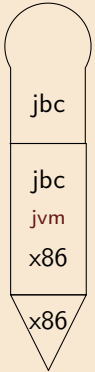

(Note: the languages must match)

# Interpreting a Java program

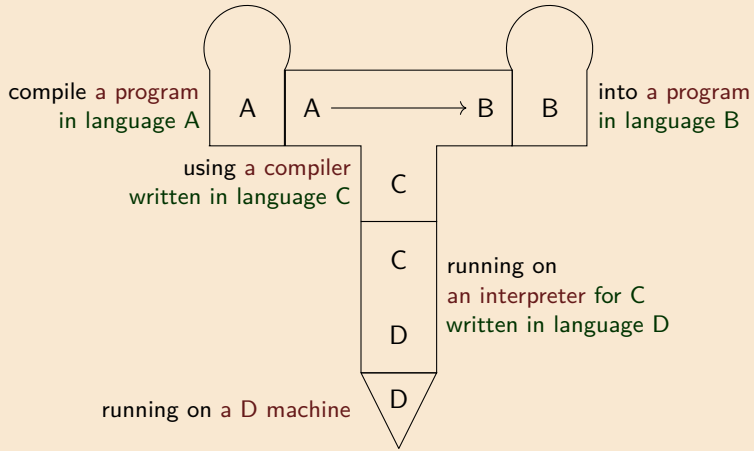
Run a program
written in Java byte code

on an interpreter for Java byte code
written in x86 code

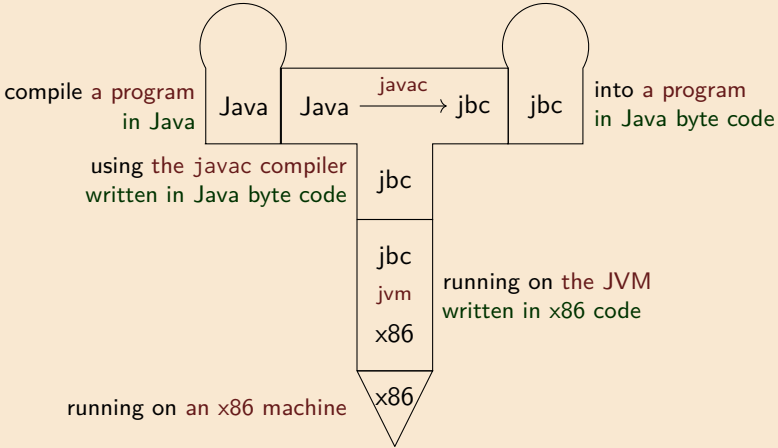
on a x86 machine



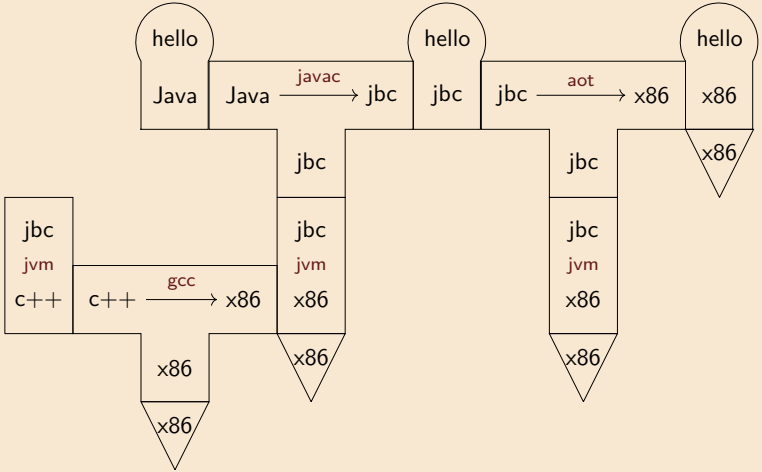
# Running a compiler on an interpreter



# Running javac on the JVM



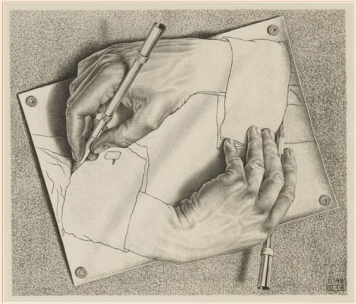
# Ahead-of-time compilation for Java




Thanks to David Greaves for the example

# Compiling compilers

The OCaml compiler
is written in OCaml

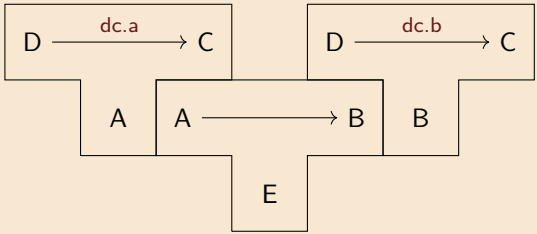


Puzzle: how was the compiler compiled?

Compilers can be translated, just like any other program:

a compiler from D to C in language A

a compiler from D to C in language B

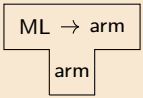



compile programs from A to B

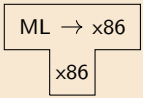
# Porting a compiler to a new platform



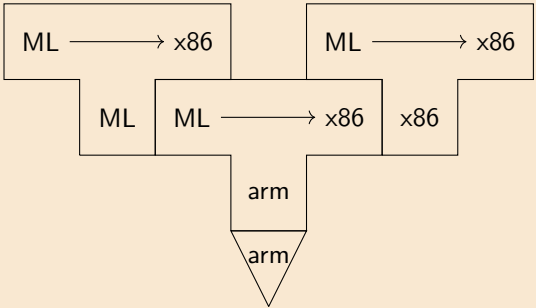
**We have**:
a compiler from ML to arm
that runs on arm




**We want**:
a compiler from ML to x86
that runs on x86




1. write an ML-to-x86 compiler in ML
2. compile the compiler for arm
3. run the compiler on arm to compile itself

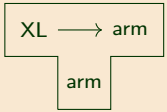


# Full bootstrap

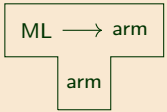
Previous example: *half bootstrap* (needs existing compiler for the language).

New example: *full bootstrap* (no existing ML compiler for the language)

**We want**:
a compiler from **XL** to arm
that runs on arm

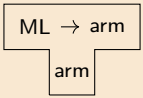



**We have**:
a compiler from ML to arm
that runs on arm

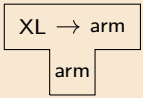


# Full bootstrap

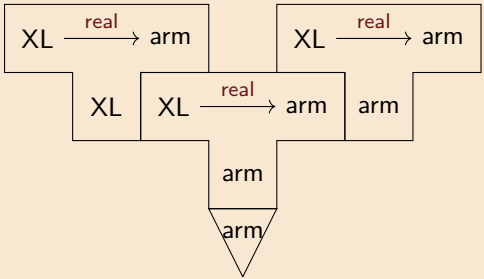
**We have**:
a compiler from ML to arm
that runs on arm




**We want**:
a compiler from **XL** to arm
that runs on arm




1. write a quick-and-dirty (QAD) **XL-to-ML** compiler in ML
2. compile the QAD compiler for arm
3. Write a real **XL-to-arm** compiler in **XL**
4. Use the QAD compiler to compile the real compiler to ML
5. Compile the resulting ML program to arm
6. Use the generated compiler to compile itself



The *speed* of the quick-and-dirty compiler does not matter much
(We could even use a quick-and-dirty interpreter instead)

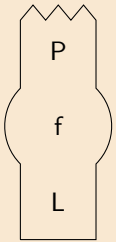
We don't need to give the quick-and-dirty compiler to users

Once the real compiler works,
we can discard the quick-and-dirty compiler altogether

# Verified compilation

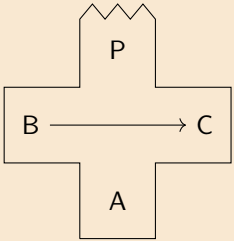
# New notation: proofs of correctness

## A verified program



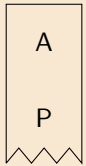

Computes function f
written in language L
with a proof in language P

## A verified compiler




Translates language A into B
written in language C
with a proof in language P

## A proof checker



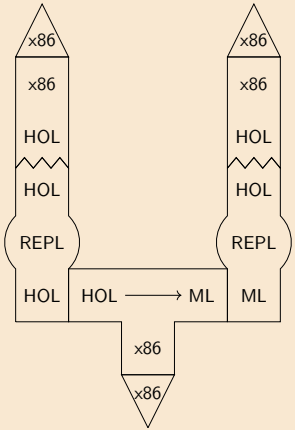

Checks proofs in language P
Written in language A

Thanks to Paul D'Souza for the extended notation and the example

# Compiling the CakeML verified compiler: HOL ⇝ ML



The CakeML REPL[1] is written in HOL and compiled to ML code + a proof

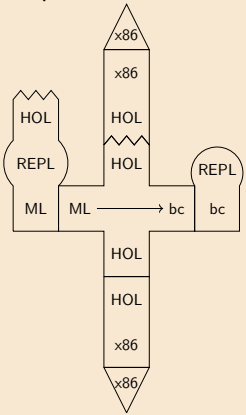



[1] read-eval-print loop

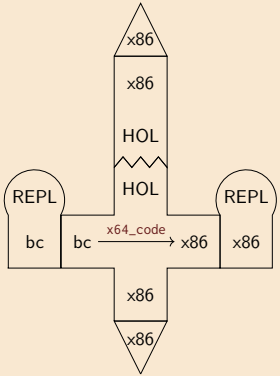
# Compiling the CakeML verified compiler: ML ⇝ x86



Next, the *verified* compiler compiles the REPL to bytecode




Finally, the bytecode is compiled




**Theorem** (abridged): the x86 REPL behaves according to the original specification

# Trusting trust

> "The cutest program I ever wrote"
> – Ken Thompson
>
> *(Reflections on Trusting Trust)*

**Aim**: modify a compiler to compromise `login`

**Warm up**: teach a compiler about vertical tabs

C compilers have code to interpret escape sequences like `\n` in `"Hello, world\n"`:

```
...
c = next();
if (c != '\\') return c;
c = next();
if (c == '\\') return '\\';
if (c == 'n') return '\n';
...
```

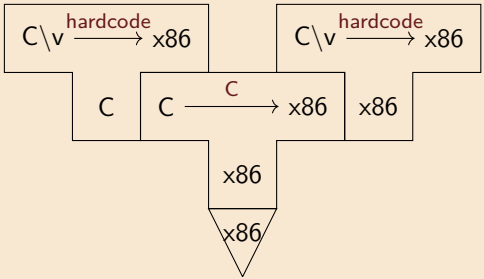
**Q**: how can we add support for vertical tabs `\v`?

(Assume the C compiler is bootstrapped.)

**Step 1**: hard-code the ASCII code for `\v` in the compiler source:

```
c = next();
if (c == '\\') return '\\';
if (c == 'n') return '\n';
if (c == 'v') return 11;
...
```

Recompile the compiler source using the installed C compiler:

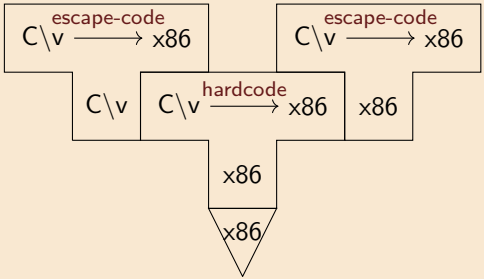



Now we have a C compiler that supports `\v` in C programs. Install it.

**Step 2**: modify the compiler source again to remove the hardcoded constant:

```
c = next();
if (c == '\\') return '\\';
if (c == 'n') return '\n';
if (c == 'v') return '\v';
...
```

Recompile the modified source using the freshly installed C compiler:



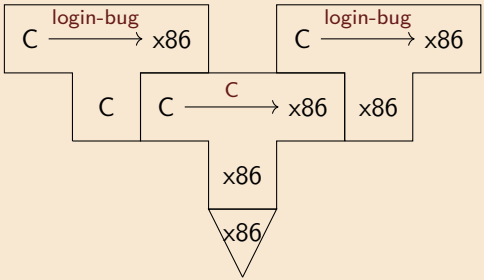

The C compiler has learnt to translate `\v` (but there's no record in the source!)

**Plan**: repeat the process to compromise the `login` command.

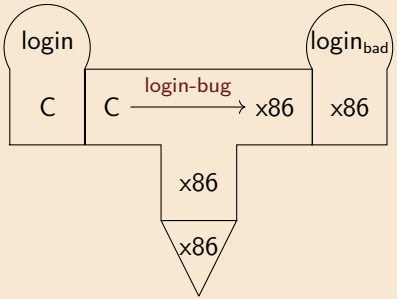
**Step 1**: update the C compiler's code to detect `login.c` and insert a bug:

```
void compile(const char *program) {
  if (matches(program, "< login code >") {
     compile("< code for backdoor >");
  }
  ...
```

Compile and install the new C compiler:



Now the compiler will miscompile `login`:

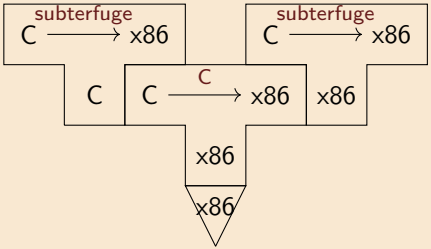



**Problem**: people will easily spot the bug in the compiler source.

**Step 2**: update the C compiler code to detect `compiler.c` and insert a 2nd bug:

```
void compile(const char *program) {
  if (matches(program, "< login code >") {
     compile("< code for backdoor >");
  }
  if (matches(program, "< compiler code >") {
     compile("< code for miscompilation >");
  }
```

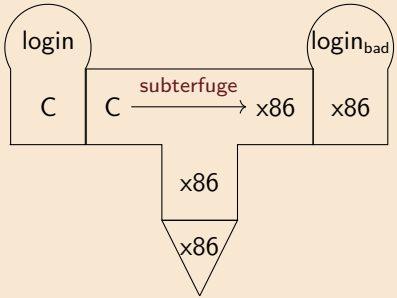
Compile and install the new C compiler:



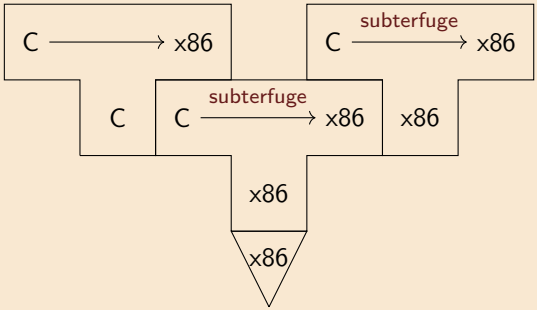

**Finally**: remove the bugs from the compiler source.

# The compiler has learnt to insert backdoors

The compiler will still miscompile `login`:




The compiler will now also miscompile the compiler:




The system is **compromised**, with **no trace** in the `login` or compiler source.

We need to *debootstrap* to recover an uncompromised compiler.