NTNU | Norwegian University of Science and Technology

# Compiler Construction

## Lecture 1: Motivation and History

Michael Engel

# whoami?

- Michael Engel
  (michael.engel@ntnu.no, http://folk.ntnu.no/michaeng/)
- Studied computer engineering and applied mathematics (Univ. Siegen)
- PhD (Univ. Marburg) 2005
- Assist. Prof. TU Dortmund 2007–14
- Leeds Beckett U., Oracle Labs UK 2014–16
- Assoc. Prof. Coburg Univ. 2016–19
- Assoc. Prof. NTNU 2020–…


- Research Interests

  Compilers, operating systems, parallelization, dependability, embedded systems

# .org

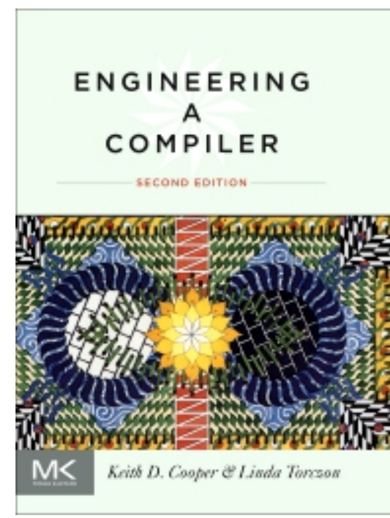
## Timetable

| Day | Time | Location | Type |
|---|---|---|---|
| Tue | 14:15-15:00 | Geologi G1 | Lecture/Forelesning |
| Tue | 15:15-16:45 | Realfagbygget R8 | Recitation/Øving |
| Fr | 12:15-14:00 | Sentralbygg 1 S4 | Lecture/Forelesning |

## Literature

| | |
|---|---|
| Authors | Keith Cooper, Linda Torczon |
| Title | Engineering a Compiler (Second Edition) |
| ISBN | 9780120884780 (hardcover)<br>9780080916613 (ebook) |

+ additional papers, articles, … on my web page



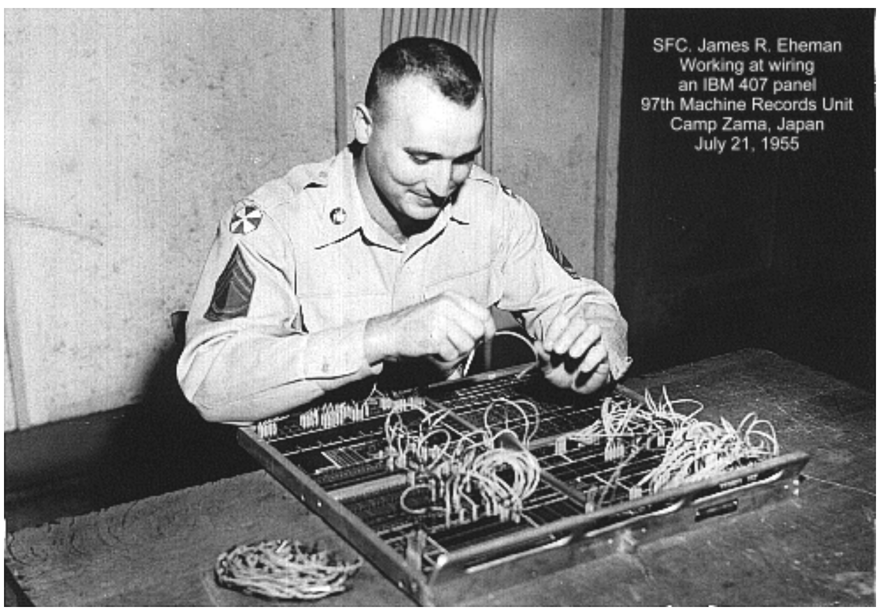
# Overview

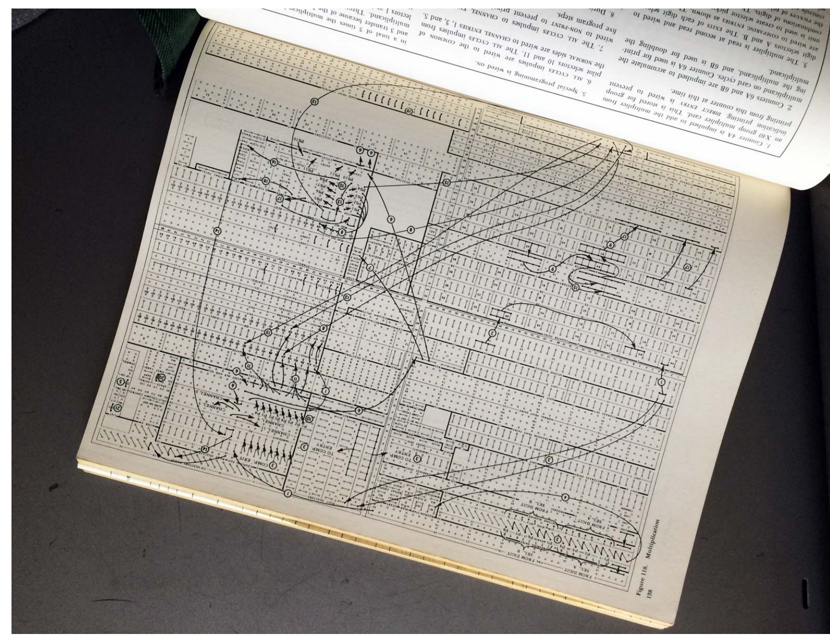
- History: the evolution of programming
  - from plugboards to compilers
- History of compilers
- The compilation process
- Semester overview

- Recitation (15:15–16:45): C crash course

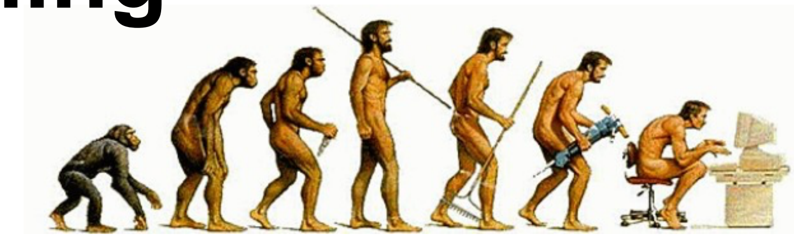
# Evolution of programming

- Early "computers" were electric calculating machines
- "Programming" meant creating a machine configuration using a plugboard
  - Bugs/changes => rewire...

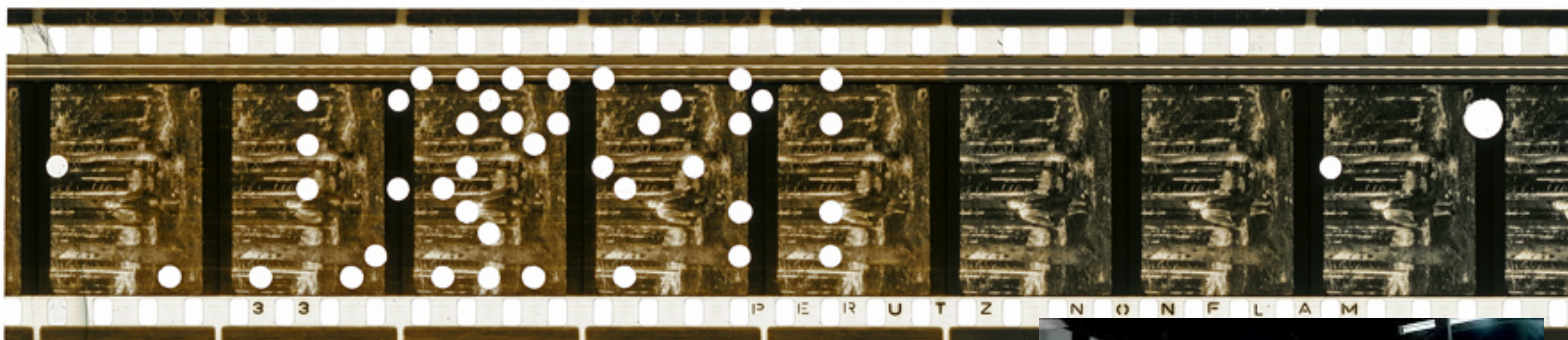


# Evolution of programming



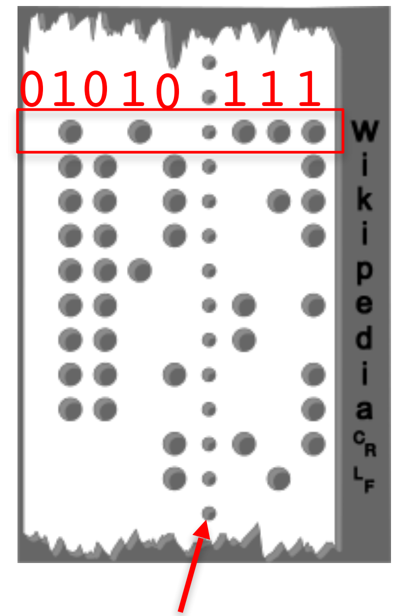
- Early programmable computers: "make bits by hand"
  - Zuse Z3 punched tape (1943): holes stamped in old cinema film rolls
  - later: paper tape



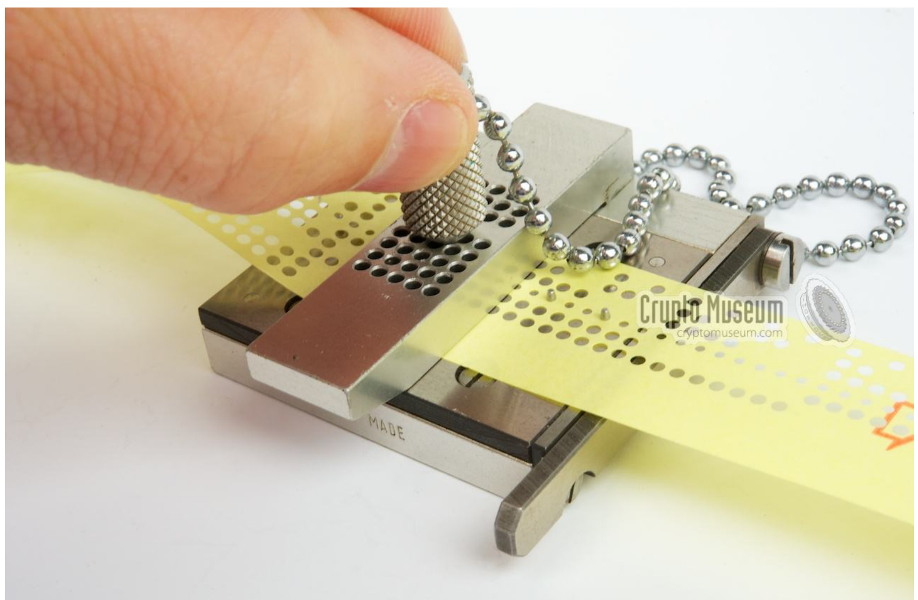

  - One word (set of bits) encoded per column
  - "hole" = log. 1, "no hole" = 0
  - e.g. 8 bits (one byte) per column

# What's on the tape?

- "…it depends"
- Data (text, numbers, …)
  - e.g. ASCII characters: 01010111 = 0x57 = "W"
- but also instructions

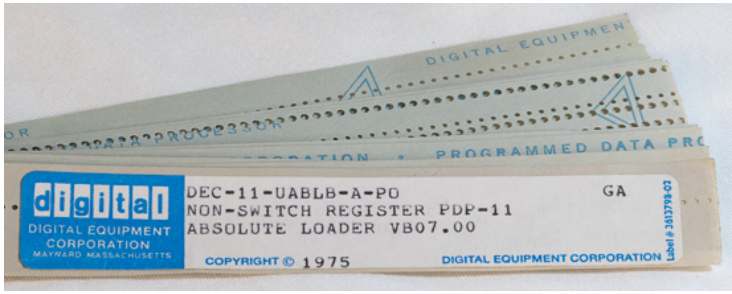



transport holes
(don't encode data)



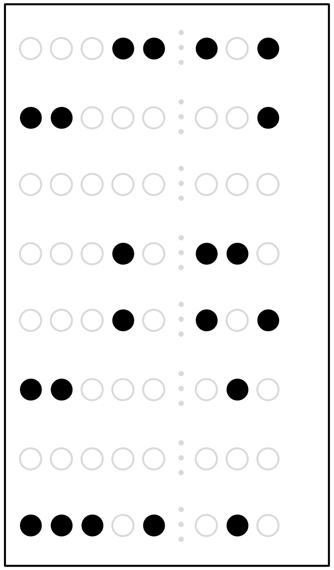

Manual tape punch

# Instructions on tape



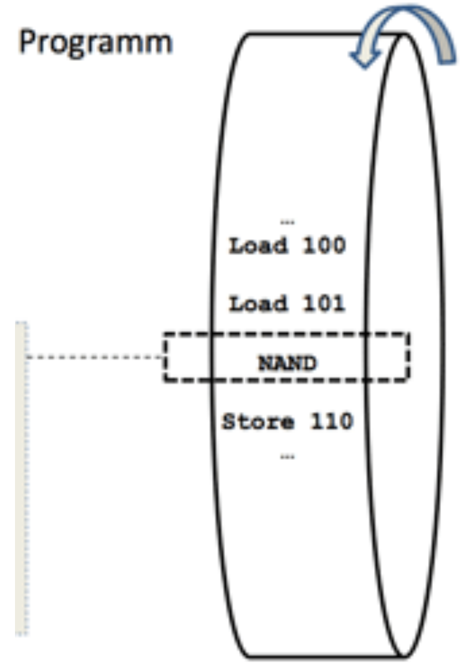

- Early computers (like the Z3) had no program storage
- The computer reads one instruction after the other from tape
- Later: load program from tape into memory
- Example: part of DEC PDP-11 boot loader on paper tape (1975)

00011 101
11000 001
00000 000
00010 110
00010 101
11000 010
00000 000
11101 010



# Building program structures

- Machine instruction on paper tape
- Columns (e.g. bytes) read one after the other
  - PDP-11 puts bytes into consecutive memory locations
  - Z3 reads **and executes** instructions from tape one after the other




- How can sequences of instructions be repeated?
  - Simply tape the end of the paper tape to the start: create a **loop**
- How could one implement conditional execution of code (if/then/else)?

# A manually created loop

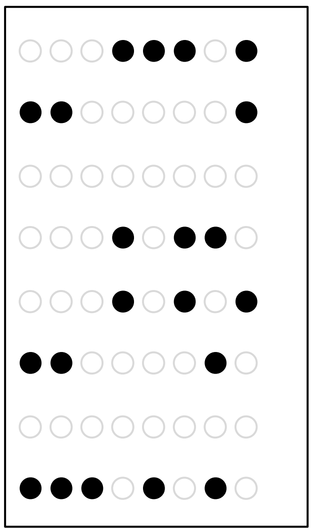
# Programs in memory

- Running code from paper tape is inconvenient
- John von Neumann invented the stored program concept (late 1940s)
  - Code and data share the same memory
- Until the 1970s, computers had **front panels** with switches and lights that enabled the operator to view and change every bit in the system
- Without boot ROM: boot loader had to be “toggled” in by hand…


DEC PDP11/70 front panel replica (3D printed) connected to a Raspberry Pi running a PDP11 emulator

# Programs in memory

- PDP11 instruction words are always multiples of 16 bits

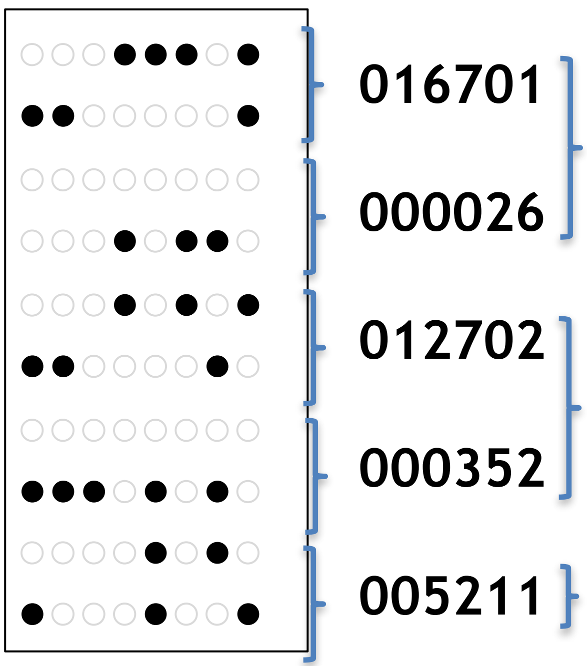


| byte | | octal | | binary (16 bit word) |
|---|---|---|---|---|
| 00011101 | | | | |
| 11000001 | | **016701** | = | 0 001 110 111 000 001 |
| 00000000 | | | | |
| 00010110 | | **000026** | = | 0 000 000 000 010 110 |
| 00010101 | | | | |
| 11000010 | | **012702** | = | 0 001 010 111 000 010 |
| 00000000 | | | | |
| 11101010 | | **000352** | = | 0 000 000 011 101 010 |

- Would you want to program a computer this way?

# From machine code to assembly

- Assembler: human readable machine instructions
- Common: 1:1-equivalence of
  assembler instruction to binary machine instruction
  - Some assemblers use “pseudo instructions” (ARM, MIPS, RISC-V)

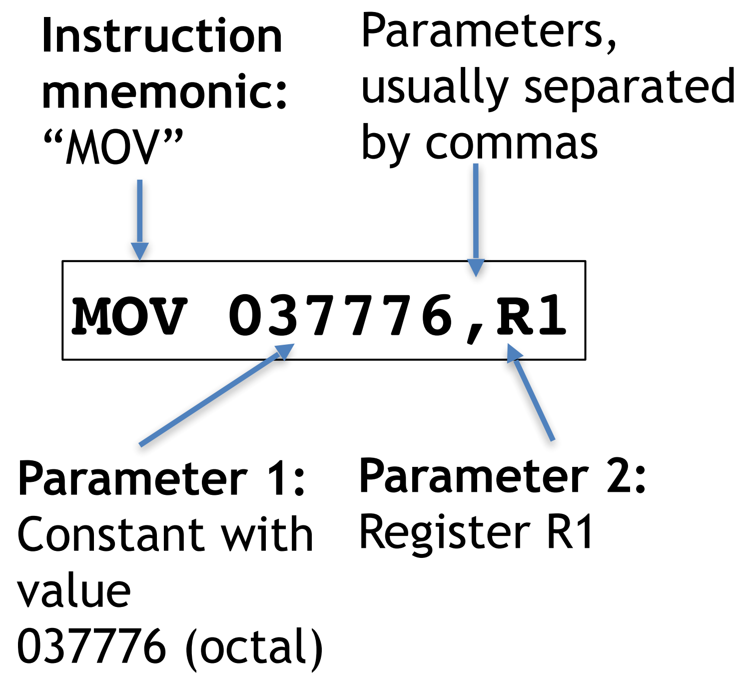



| octal encoding of machine instr. | equivalent assembler instruction |
|---|---|
| 016701 000026 | **MOV 037776,R1** |
| 012702 000352 | **MOV #352,R2** |
| 005211 | **INC @R1** |

# From binary to assembler

- Assembler instructions consist of instruction name (*mnemonic*) and optional parameters
- Parameters can be constants, register numbers, addresses

| octal encoding of machine instr. | assembler instruction with numeric constants |
|---|---|
| `016701 000026` | `MOV 037776,R1` |
| `012702 000352` | `MOV #352,R2` |
| `005211` | `INC @R1` |
| `105711` | `TSTB @R1` |
| `100376` | `BPL 037756` |
| `116162 000002` | |
| `037400` | `MOVB 2(R1),37400(R2)` |
| `005267 177756` | `INC 037752` |
| `000765` | `BR 037750` |
| `177550` | `.WORD 177550` |



# Making assembler (better) readable

- Using "magic numbers" is still quite inconvenient
- Most assemblers support the use of **symbolic names** for constants and memory addresses ("**labels**")
- In addition, comments are supported (and ignored 😉)

**memory address** | **machine instr.** | **assembler instr. using numbers**

```
037744: 016701 000026 MOV 037776,R1
037750: 012702 000352 MOV #352,R2
037754: 005211        INC @R1
037756: 105711        TSTB @R1
037760: 100376        BPL 037756
037762: 116162 000002
        037400        MOVB 2(R1),37400(R2)
037770: 005267 177756 INC 037752
037774: 000765        BR 037750
037776: 177550        .WORD 177550
```

**labels** **symbolic name**

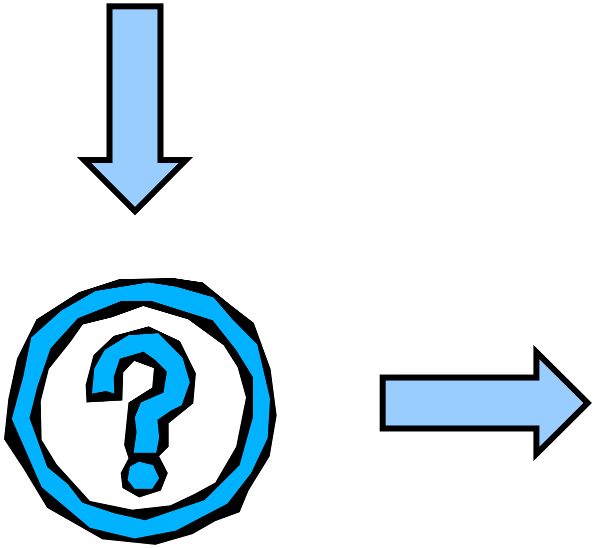
```
        mov device,r1@      // get csr address
loop:   mov #352,r2         // get offset
offset: inc (r1)            // read frame
wait:   tstb (r1)           // wait for ready
        bpl wait

        movb 2(r1),bnk(r2)  // store data
        inc loop+2          // bump address
        br loop
device: HSR         // csr, or 177560 for teletype
```

# From assembler to high-level languages

- Assembler helps (humans) to read machine-language programs
- What's missing compared to higher-level languages?
  - Constructs to enable program structure:
    **loops** (for, while, do) and **conditions** (if, switch)
  - **Variables**
    - Labels and symbolic names in assembler are just direct aliases for memory addresses resp. constants
  - **Data types, structures and objects**
    - Assembler only knows about machine data types
  - **Functions/methods**
    - Declaring, passing and returning of parameters
  - **Classes and objects**…
- **Compilers** can translate these constructs to machine language

# The compilation process black box

```
int main()
{
   . . .
   sum = num1 + num2;
   . . .
}
```



```
. . .
0xE59F1010
0xE59F0008
0xE0815000
0xE59F5008
. . .
```

# Example: from C to assembler

**C program: convert upper case to lower case letters**

- implemented as C function

- Uses ASCII character encoding:
  - 'A' = 0x41, 'B' = 0x42, ...
    'a' = 0x61, 'b' = 0x62, …

- If character in c is an upper case letter (c in ['A', 'B', … 'Z']), then the code adds the difference between lower case 'a' and upper case 'A' to variable c
- otherwise, c is returned unchanged

```
char tolower(char c)
{
  if (c >= 'A' && c <= 'Z')
     c += 'a' - 'A';

  return c;
}
```

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| **0** | NUL | DLE | space | 0 | @ | P | ` | p |
| **1** | SOH | DC1 XON | ! | 1 | A | Q | a | q |
| **2** | STX | DC2 | " | 2 | B | R | b | r |
| **3** | ETX | DC3 XOFF | # | 3 | C | S | c | s |
| **4** | EOT | DC4 | $ | 4 | D | T | d | t |
| **5** | ENQ | NAK | % | 5 | E | U | e | u |
| **6** | ACK | SYN | & | 6 | F | V | f | v |
| **7** | BEL | ETB | ' | 7 | G | W | g | w |
| **8** | BS | CAN | ( | 8 | H | X | h | x |
| **9** | HT | EM | ) | 9 | I | Y | i | y |
| **A** | LF | SUB | * | : | J | Z | j | z |
| **B** | VT | ESC | + | ; | K | [ | k | { |
| **C** | FF | FS | , | < | L | \ | l | \| |
| **D** | CR | GS | - | = | M | ] | m | } |
| **E** | SO | RS | . | > | N | ^ | n | ~ |
| **F** | SI | US | / | ? | O | _ | o | del |

# C to assembler: control structures

**Simplification of the C program**

- Assembler does not support complex "if" instructions
  - Only comparison of values and conditional jumps
- Compiler changes "and" (&&) operator into consecutive "if"s
  - Shown as simplified C code
- Complex expressions ("c += …") are also broken down
  - Three address code (two operands, one result)

```
char tolower(char c)
{
  if (c >= 'A' && c <= 'Z')
     c += 'a' - 'A';

  return c;
}
```

```
char tolower(char c)
{
  char temp;

  if (c >= 'A') {
    if (c <= 'Z') {
      temp = 'a';
      temp = temp - 'A';
      c = c + temp;
    }
  }

  return c;
}
```

# C to assembler transformation

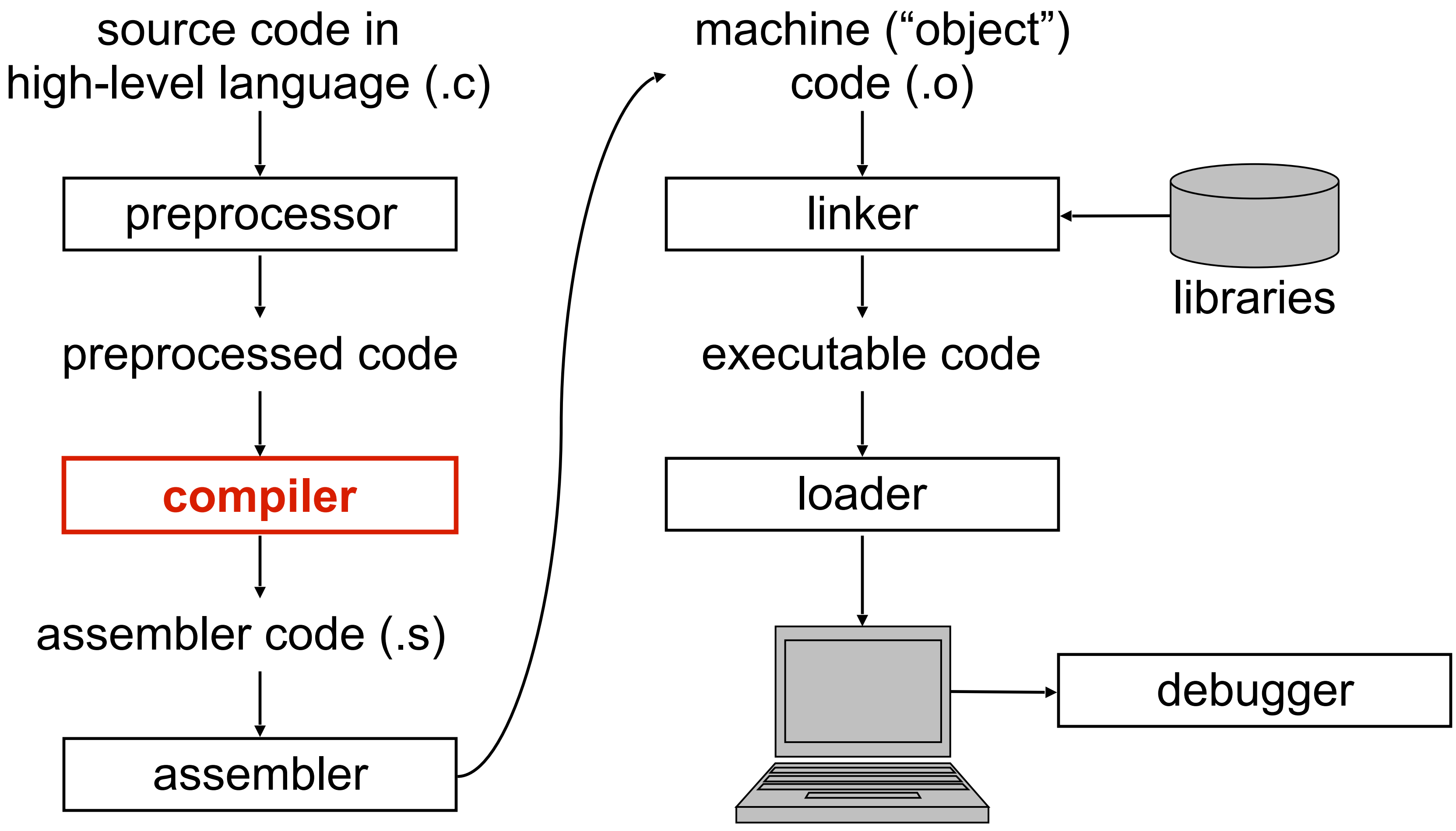
**Convert simplified C program to ARM (Thumb) assembler**

- No variables in assembler: variables in C assigned to processor registers
- c = r0, temp = r1

```c
char tolower(char c)
{
  char temp;

  if (c >= 'A') {
    if (c <= 'Z') {
      temp = 'a';
      temp = temp - 'A';
      c = c + temp;
    }
  }
  return c;
}
```

```
    AREA    text, CODE, READONLY

    EXPORT  tolower

tolower
    CMP     r0, #0x41
    BLT     lowerCase
    CMP     r0, #0x5a
    BGT     lowerCase
    MOV     r1, #0x61
    SUB     r1, #0x41
    ADD     r0, #r1
lowerCase
    BX      lr

    END
```

# Compilation process in detail



# Transpilers and other fun things

- Compilers do not always transform high-level languages to low-level machine code
- Source-to-source-compiler ("transpiler")
  - C-to-C, f2c (Fortran to C)
  - emscripten: C/C++ to Javascript
- Static binary transformation [3]
  - Dynamo optimization
- Just-in-time (JIT) compilation
  - Java VM, Android Dalvik/ART JIT
  - Transmeta Crusoe

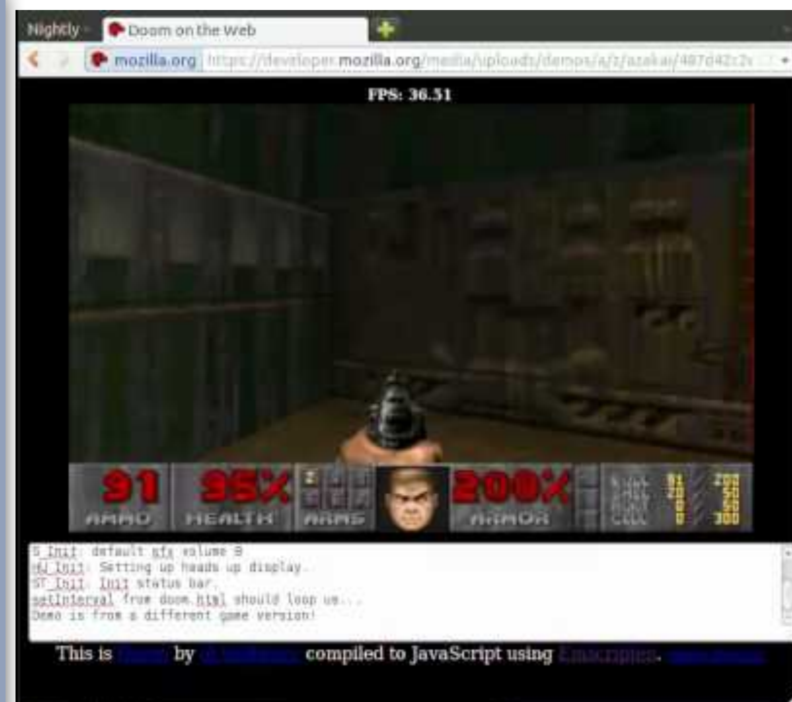
# Example: emscripten

- Source-to-source compiler [1]
  - Can transform languages with LLVM compiler frontend (C, C++, ...)
  - Runs as LLVM back end, produces JavaScript subset (wasm)
- Example use case: run Doom / Quake (written in C) in browser

```c
#include <stdio.h>
int main() {
  float fact = 1.0;
  int c;
  for (c=1; c<13; ++c) {
      fact *= c;
  }
  printf("%f\n", fact);
}
```

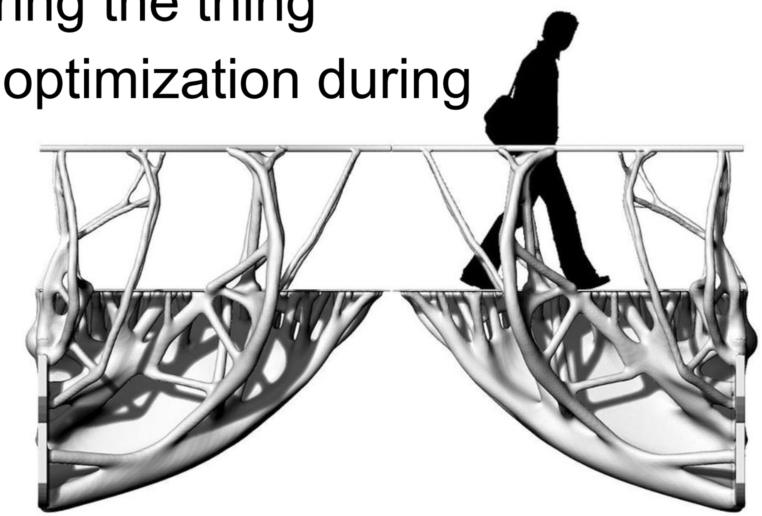
⇒ Emscripten ⇒

```
(loop $label$2
 (block $label$3
  (local.set $4
   (local.get $3)
  )
  (local.set $5
   (i32.lt_s
    (local.get $4)
    (i32.const 13)
   )
  )
  (if
   (i32.eqz
    (local.get $5)
   )
   (br $label$3)
  ...
```



# A different view of code

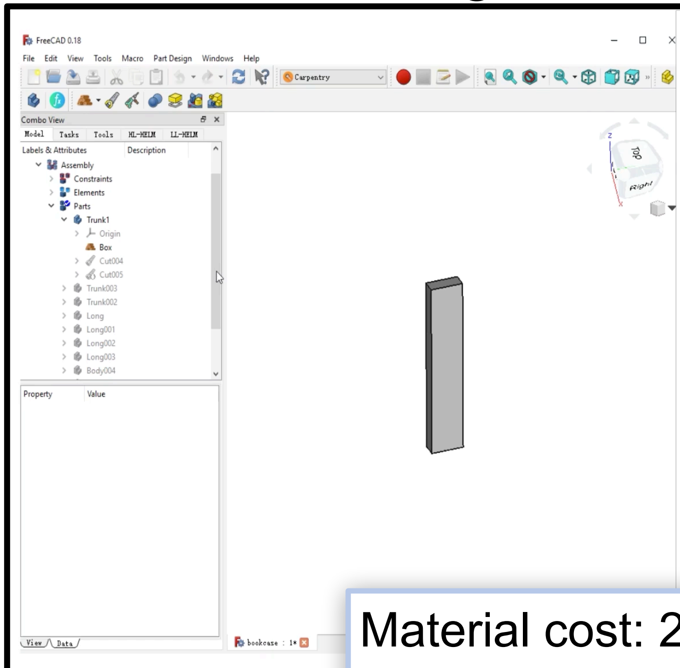
- Compilers can also be used in *very* different domains [5]
- Current research: "matter compiler"
  - Map high-level description (design) of a physical thing to instructions for machines manufacturing the thing
  - Check impossible requirements and optimization during compilation
- Example: 3D printing [5]
  - Compiler-generated 3D-printed bridge [6]
  - Output: "G code" to control 3D printer




```
T1 M6
G90 G94
G54 X0 Y0
G00 X0 Y0
G00 X0 Y-100
G01 X-59 Y81 F200 M3
G01 X95 Y-31
G01 X-95 Y-31
G01 X59 Y81
G01 X0 Y-100
G02 X100 Y0 I0 J100
G02 X0 Y-100 I-100 J0
M30;
```

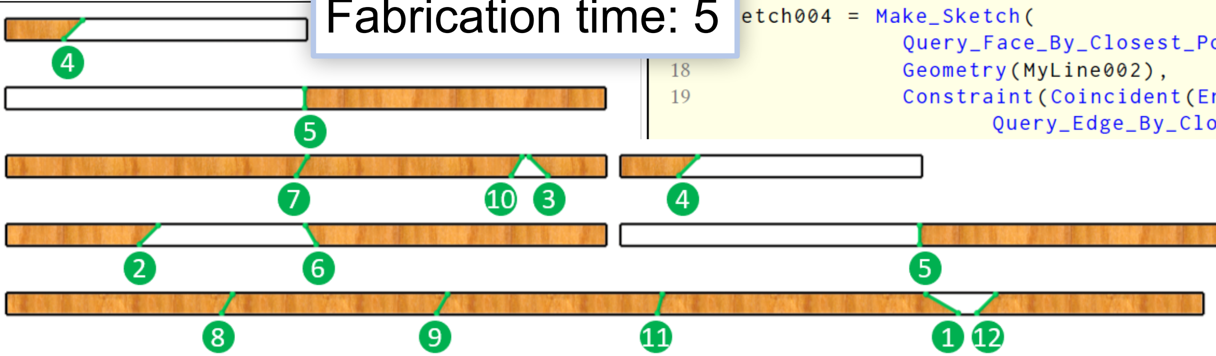
# Example: carpentry compiler

- Convert design of thing as 3D view to manufacturing code [4]




```
Box001 = Make_Stock(457.2, 38.1, 88.9);
MyLine000 = Line(457.2, 0, 435.203, 38.1);
Sketch = Make_Sketch(
        Query_Face_By_Closest_Point(Box001, 228.6, 19.05, 88.9),
        Geometry(MyLine000),
        Constraint(Coincident(Start(MyLine000), End(
              Query_Edge_By_Closest_Point(Box001, 228.6, 0, 88.9))),
              PointOnObject(End(MyLine000), Query_Edge_By_Closest_Point(
              Box001, 228.6, 38.1, 88.9)), Angle(Start(
              Query_Edge_By_Closest_Point(Box001, 457.2, 19.05, 88.9)), Start
              (MyLine000), 30)));
Cut = Make_Cut(Box001, Sketch, 0);
MyLine001 = Line(0, 38.1, 21.997, 0);
Sketch001 = Make_Sketch(
        Query_Face_By_Closest_Point(Cut, 228.6, 19.05, 88.9),
        Geometry(MyLine001),
        Constraint(Coincident(Start(MyLine001), End(
              Query_Edge_By_Closest_Point(Cut, 0, 19.05, 88.9))),
              PointOnObject(End(MyLine001), Query_Edge_By_Closest_Point(
              Cut, 228.6, 0, 88.9)), Angle(End(Query_Edge_By_Closest_Point
              (Cut, 0, 19.05, 88.9)), Start(MyLine001), 30)));
t001 = Make_Cut(Cut, Sketch001, 1);
x = Make_Stock(457.2, 38.1, 88.9);
Line002 = Line(435.203, 38.1, 457.2, 0);
etch004 = Make_Sketch(
        Query_Face_By_Closest_Point(Box, 228.6, 19.05, 88.9),
        Geometry(MyLine002),
        Constraint(Coincident(End(MyLine002), End(
              Query_Edge_By_Closest_Point(Box, 228.6, 0, 88.9))),
              t(MyLine002), Query_Edge_By_Closest_Point(
              88.9)), Angle(MyLine002, -60)));

              int(Cut004, 228.6, 19.05, 88.9),

              art(MyLine003), End(
              Query_Edge_By_Closest_Point(Cut004, 0, 19.05, 88.9))),
              PointOnObject(End(MyLine003), Query_Edge_By_Closest_Point(
              Cut004, 228.6, 0, 88.9)), Angle(End(
              Query_Edge_By_Closest_Point(Cut004, 0, 19.05, 88.9)), Start(
```

Material cost: 2.95
Fabrication time: 5



# Semester overview (tentative)

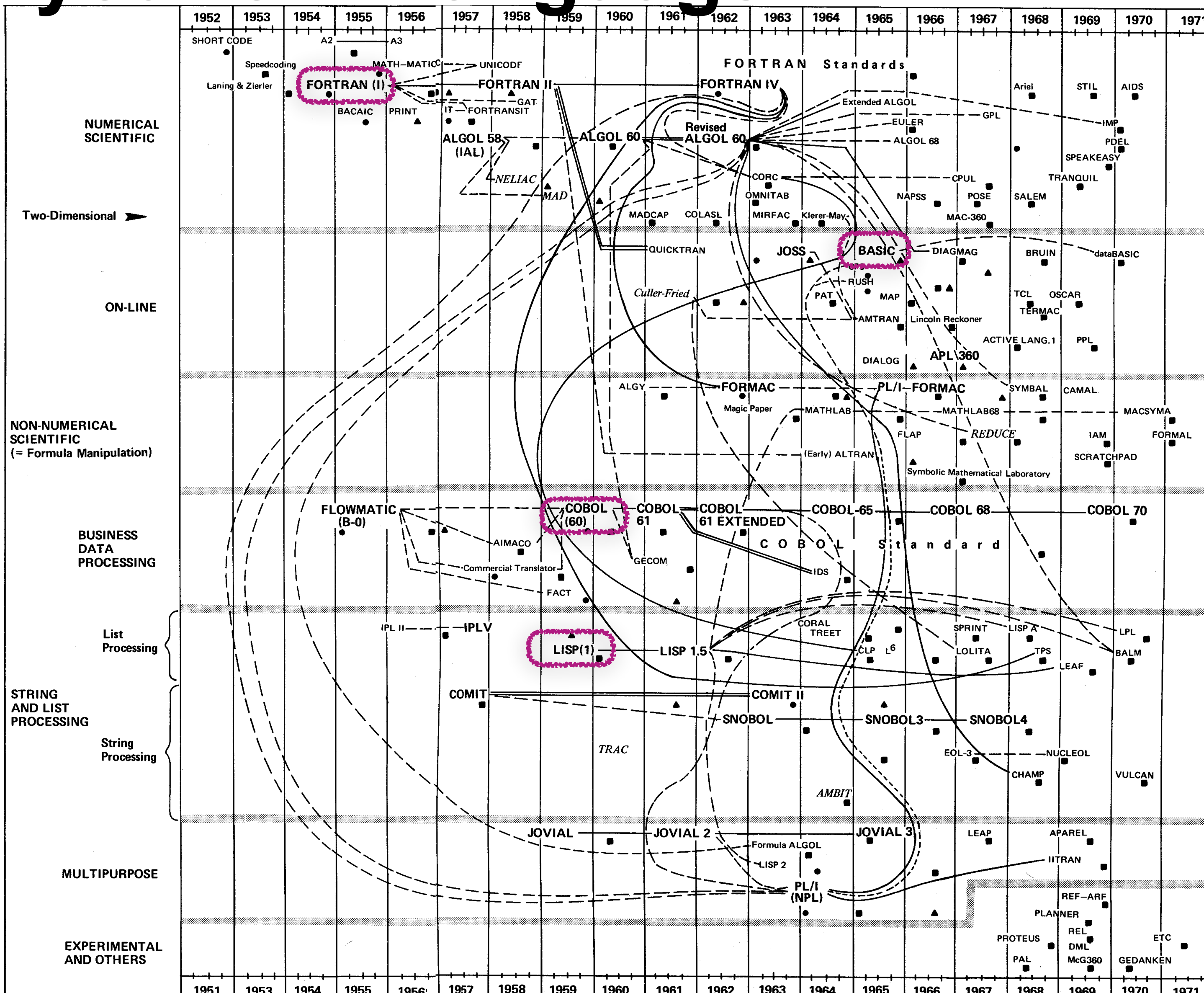
- Structure of a typical compiler
- Frontend
  - Scanning
  - Parsing and grammars
- Intermediate representations
  - Abstract syntax trees (ASTs) and SSA form
- Backend
  - Code generation
  - Code optimization
  - Linking
- Static code analysis

# Design your own language?

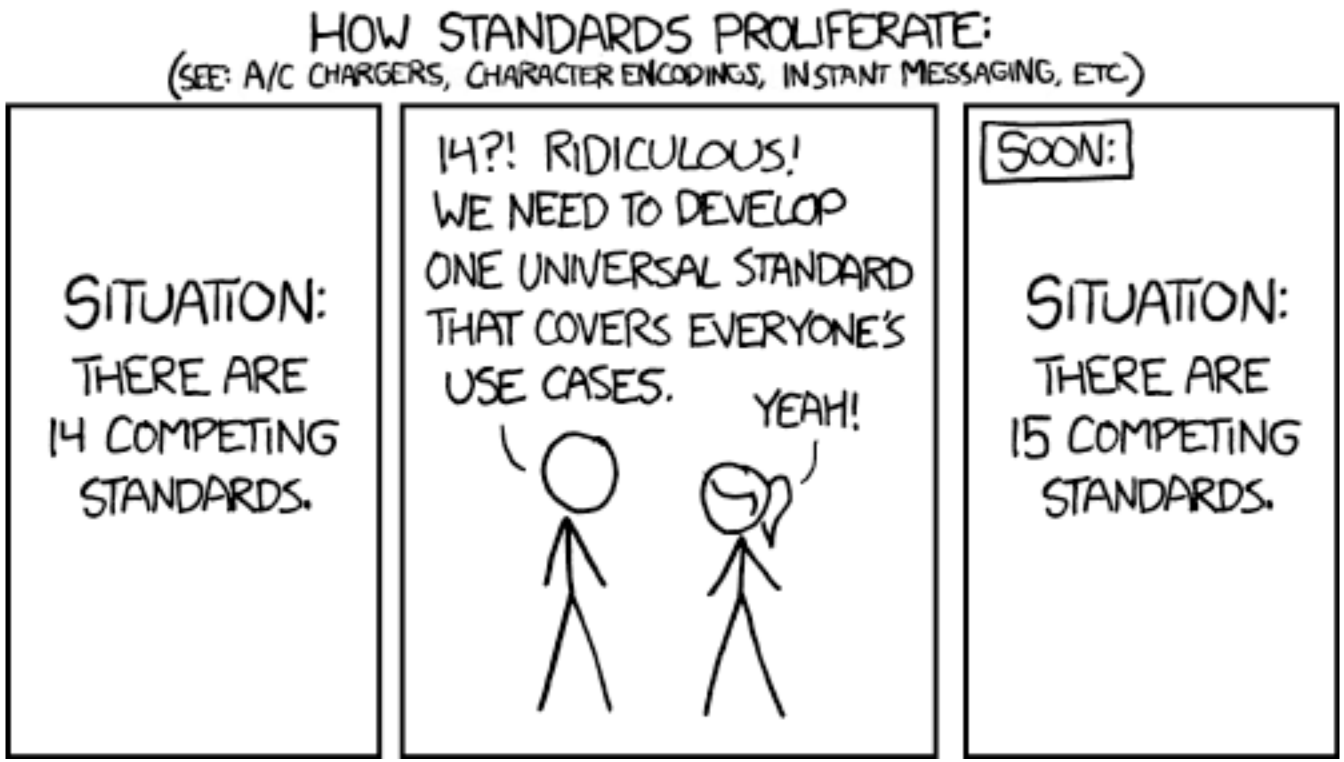
20 years of development [2]

Which languages are still widely used?

- FORTRAN
- COBOL
- LISP
- BASIC



# Design your own language?




xkcd by Randall Munroe: https://imgs.xkcd.com/comics/standards.png

# References


1. Alon Zakai, **Emscripten: an LLVM-to-JavaScript compiler**, Proceedings of OOPSLA'11
2. Jean E. Sammet, **Programming languages: history and future**, Communications of the ACM, July 1972, https://doi.org/10.1145/361454.361485
3. C. Cifuentes and V. Malhotra, **Binary translation: static, dynamic, retargetable?**, Proceedings of the International Conference on Software Maintenance 1996
4. Chenming Wu, Haisen Zhao, Chandrakana Nandi, Jeffrey I. Lipton, Zachary Tatlock and Adriana Schulz, **Carpentry Compiler**, ACM Transactions on Graphics 38(6), 2019
5. Hod Lipson and Melba Kurman, **Fabricated: The New World of 3D Printing**, Wiley 2013, ISBN: 978-1-118-35063-8, p.
6. "**3D Printing And The Complexity Of Compiling Matter"** https://www.forbes.com/sites/valleyvoices/2015/09/02/3d-printing-and-the-complexity-of-compiling-matter/