TU Berlin, Scientific Computing
Winter Semester 2021/2022

Slide lecture 3

Jürgen Fuhrmann

juergen.fuhrmann@wias-berlin.de

Some general remarks on computer languages

Machine code

- Detailed instructions for the actions of the CPU are provided as binary code (mostly written in hexadecimal form)
- Not human readable, but programming started with hand-coding instructions like this:

```
534c 29e5 31db 48c1 fd03 4883 ec08 e85d
feff ff48 85ed 741e 0f1f 8400 0000 0000
4c89 ea4c 89f6 4489 ff41 ff14 dc48 83c3
0148 39eb 75ea 4883 c408 5b5d 415c 415d
415e 415f c390 662e 0f1f 8400 0000 0000
f3c3 0000 4883 ec08 4883 c408 c300 0000
0100 0200 4865 6c6c 6f20 776f 726c 6400
011b 033b 3400 0000 0500 0000 20fe ffff
8000 0000 60fe ffff 5000 0000 4dff ffff
```

- Sample types of instructions:
  - Transfer data between memory location and register
  - Perform arithmetic/logic operations with data in register
  - Check if data in register fulfills some condition
  - Conditionally change the memory address from where instructions are fetched $\equiv$ "jump" to address
  - Save all register context and take instructions from different memory location until return $\equiv$ "call"

My first programmable computer

SER2d by VEB Elektronische Rechenmaschinen Karl-Marx-Stadt (around 1962)
My secondary school owned one around 1975

- I started programming this way
- Instructions were supplied on punched tape
- Output was printed on a typewriter
- The magnetic drum could store 127 numbers and 127 instructions

Assembler code

- Human readable representation of CPU instructions
- Some write it by hand ...
  - Code close to abilities and structure of the machine
  - Handle constrained resources (embedded systems, early computers)
- Translated to machine code by a programm called *assembler*

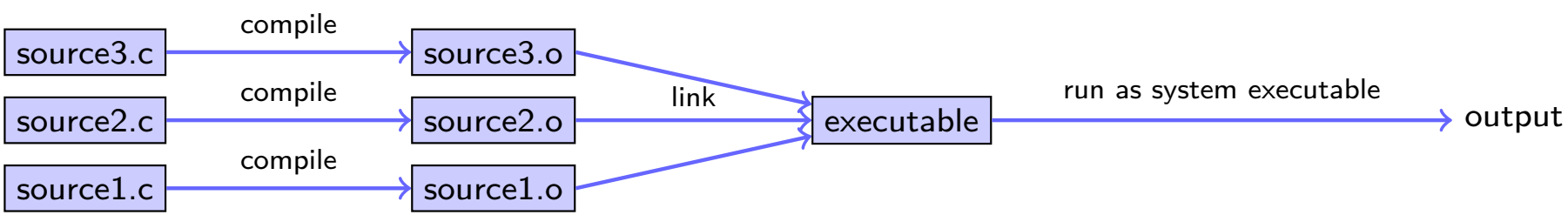
```
.file   "code.c"
.section    .rodata
.LC0:
.string "Hello world"
.text
...
pushq   %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq    %rsp, %rbp
.cfi_def_cfa_register 6
subq    $16, %rsp
movl    %edi, -4(%rbp)
movq    %rsi, -16(%rbp)
movl    $.LC0, %edi
movl    $0, %eax
call    printf
```

Compiled high level languages

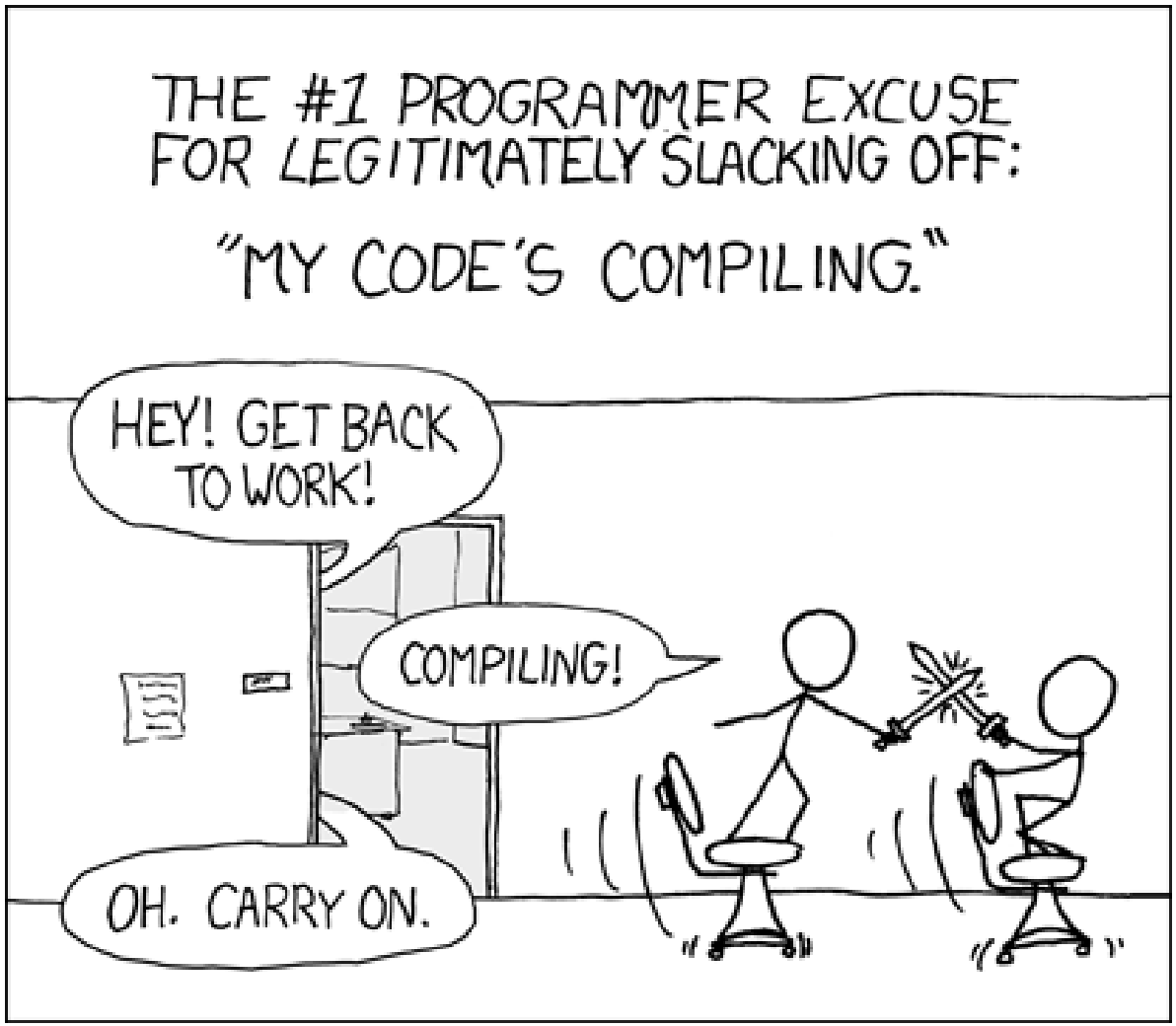
- Algorithm description using mix of mathematical formulas and statements inspired by human language
- Translated to machine code (resp. assembler) by *compiler*

```
#include <stdio.h>
int main (int argc, char *argv[])
{
  printf("Hello world");
}
```

- "Far away" from CPU $\Rightarrow$ the compiler is responsible for creation of optimized machine code
- Fortran, COBOL, C, Pascal, Ada, Modula2, C++, Go, Rust, Swift ...
- Strongly typed
- Tedious workflow: compile - link - run



Compiling. . .



. . . from xkcd

Compiled languages in Scientific Computing

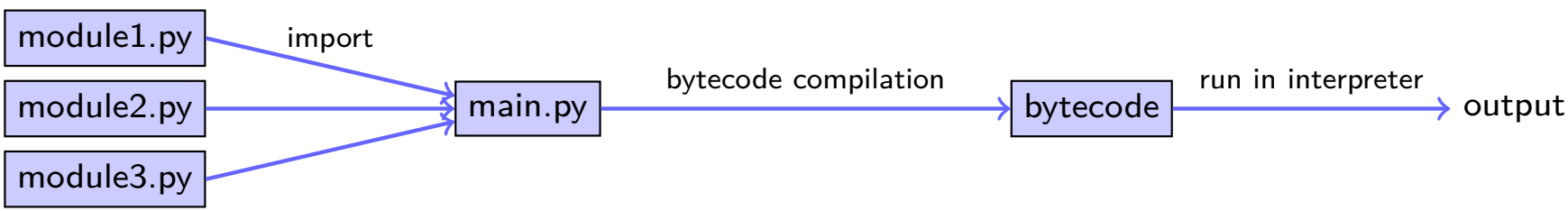
- Fortran: FORmula TRANslator (1957)
  - Fortran4: really dead
  - Fortran77: large number of legacy libs: BLAS, LAPACK, ARPACK ...
  - Fortran90, Fortran2003, Fortran 2008
    - Catch up with features of C/C++ (structures,allocation,classes,inheritance, C/C++ library calls)
    - Lost momentum among new programmers
    - In many aspects very well adapted to numerical computing
    - Well designed multidimensional arrays
    - Still used in several subfields of scientific computing
- C: General purpose language
  - K&R C (1978) weak type checking
  - ANSI C (1989) strong type checking
  - Had structures and allocation early on
  - Numerical methods support via libraries
  - Fortran library calls possible
- C++: *The* powerful general purpose object oriented language used (not only) in scientific computing
  - Superset of C (in a first approximation)
  - Classes, inheritance, overloading, templates (generic programming)
  - C++11: $\approx$ 2011 Quantum leap: smart pointers, threads, lambdas, anonymous functions
  - Since then: C++14, C++17, C++20 – moving target ...
  - With great power comes the possibility of great failure...

High level scripting languages

- Algorithm description using mix of mathematical formulas and statements inspired by human language
- Often: simpler syntax, less "boiler plate"

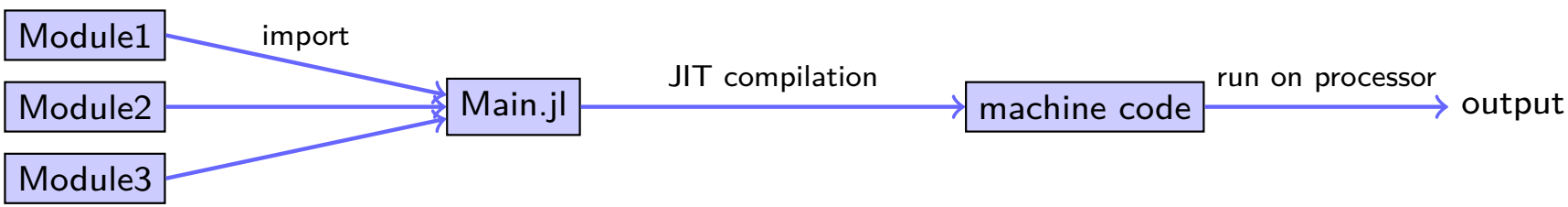
```
print("Hello world")
```

- Need intepreter in order to be executed
- Very far away from CPU $\Rightarrow$ usually significantly slower compared to compiled languages
- Matlab, Python, R, Lua
- Less strict type checking, powerful introspection capabilities
- Immediate workflow: "just run"
  - in fact: first compiled to *bytecode* which can be interpreted more efficiently



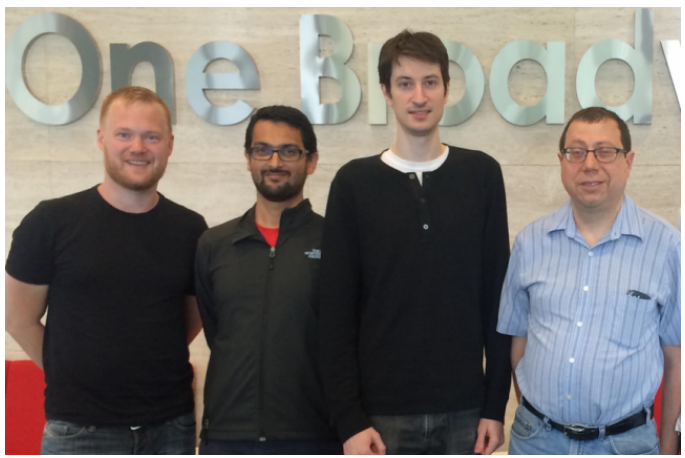
JIT based languages

- Byte code interpretation is a perfomance bottleneck.
  - Common practice: use compiled language for performance critical parts - e.g. use C for performance critical part of Python
  - "two language problem":
    - need for three skill sets: two languages + interfacing
    - complex tooling for installation, porting and maintenance
- **Just In Time compilation (JIT)**: compile to **machine code** instead of byte code "on the fly"
  - Many languages try to add JIT technology after they have been designed: javascript, Lua, Java, Smalltalk, Python/NUMBA
  - **Julia** (v1.0 since August, 2018) was **designed** for JIT



Julia History & Resources

- 2009-02: V0.1 Development started in 2009 at MIT (S. Bezanson, S. Karpinski, V. Shah, A. Edelman)
- 2012: V0.1
- 2016-10: V0.5 experimental threading support
- 2017-02: SIAM Review: Julia - A Fresh Approach to Numerical Computing
- 2018-08: **V1.0**
- 2018 Wilkinson Prize for numerical software



- Homepage incl. download link: https://julialang.org/
- Wikibook: https://en.wikibooks.org/wiki/Introducing_Julia

Julia - a first characterization

"**Like matlab, but faster**"

"**Like matlab, but open source**"

"**Like python + numpy, but faster and counting from 1**"

- Main purpose: performant numerics
- Multidimensional arrays as first class objects
  (like Fortran, Matlab; unlike C++, Swift, Rust, Go . . .)
- Array indices counting from 1
  (like Fortran, Matlab; unlike C++, python) - but it seems this becomes more flexible
- Array slicing etc.
- Extensive library of standard functions, linear algebra operations
- Growing package ecosystem

. . . there is more to the picture

- Developed from scratch using modern knowledge in language development
- Strongly typed $\Rightarrow$ JIT compilation to performant code
- Multiple dispatch: all functions are essentialy templates
- Parallelization: SIMD, threading, distributed memory
- Reflexive: one can loop over struct elements
- Module system, module precompilation
- REPL (Read-Eval-Print-Loop)
- Ecosystem:
  - Package manager with github integration
  - Foreign function interface to C, Fortran, wrapper methods for C++
  - PyCall.jl: loading of python modules via reflexive proxy objects (e.g. plotting)
  - Intrinsic tools for documentation, profiling, testing
  - Code inspection of LLVM and native assembler codes
  - IDE integration with Visual Studio Code
  - Jupyter, Pluto notebooks

Conclusion

. . . R. Hamming (of "Hamming code" and "Hamming distance" fame, who started his carrier programming in Los Alamos) in 1968:

"Indeed, one of my major complaints about the computer field is that whereas Newton could say,"If I have seen a little farther than others, it is because I have stood on the shoulders of giants," I am forced to say, "Today we stand on each other's feet." **Perhaps the central problem we face in all of computer science is how we are to get to the situation where we build on top of the work of others rather than redoing so much of it in a trivially different way.** Science is supposed to be cumulative, not almost endless duplication of the same kind of things." (1968)

- In addition to it's take on performance, Julia focuses on composability of packages from different authors, moreover, it allows to interface existing codes in C,C++, Python, R and other languages
- Of all "new" languages (like Rust, Swift, Go etc.) Julia is the only one designed for numerical computing
- C++ still lacks standardizes basic infrastructure like versatile multidiemensional arrays
- So let us give Julia a try...