Compilers and intrinsics

Lectures on Modern Scientific Programming
Wigner RCP
23-25 November 2015

llvm.org/Logo.html

Compilers

We do not write assembly instructions because:

- It simply does not scale
  Large problems require proportionally larger work...
- Not safe, hard to comprehend, understand, maintain
- Not portable
  between architectures

llvm.org/Logo.html

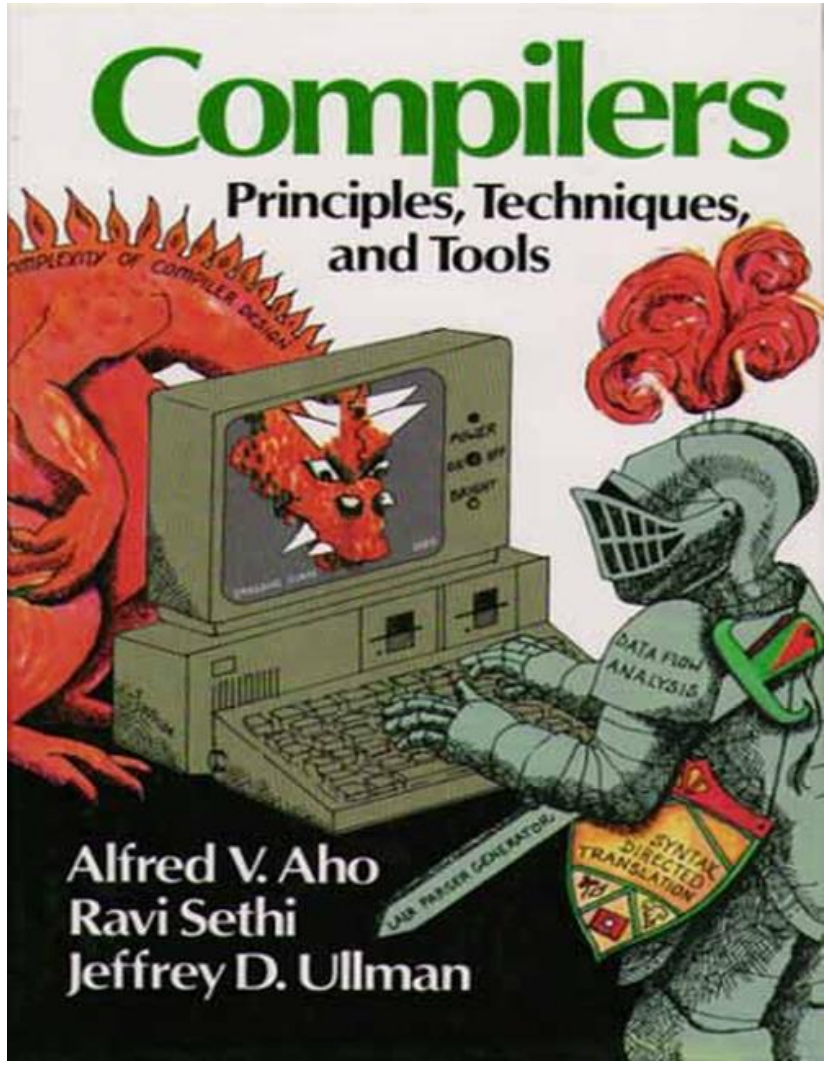
Compilers

Instead:

we use higher-level languages and a compiler, that translates this language to assembly instructions / binary

llvm.org/Logo.html

Compilers



„This picture is well-known in the compiler field

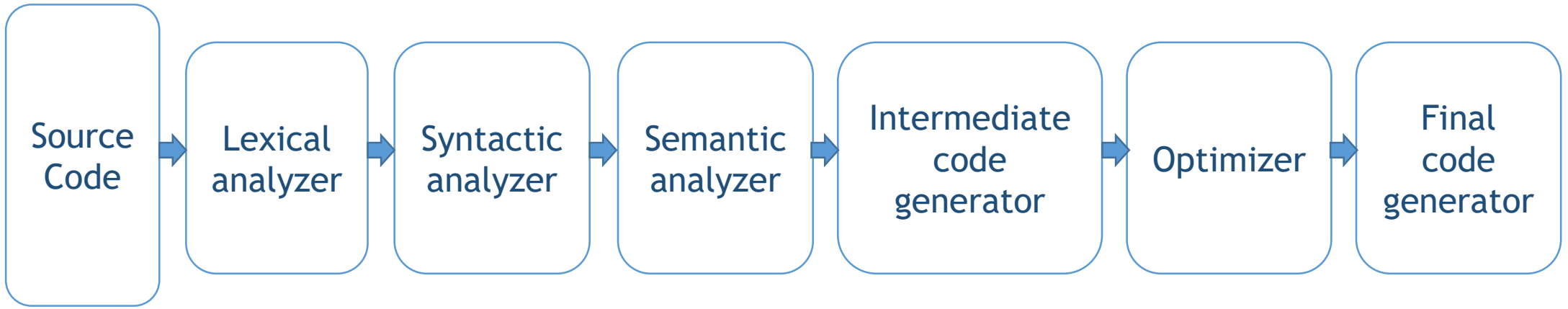
The dragon is the problem of compiling, the knight uses algorithms and data structures to ‚slay the dragon' "

„Dragons have connotations of power, speed and intelligence, and can also be sleek, elegant, and modular (err, maybe not)."

llvm.org/Logo.html

Compilers

Compilers are a complex piece of software
and to achieve performance and precision we should have an idea how they work



llvm.org/Logo.html

Compilers

Compiler development is complicated, but every day a new programming language is invented

How is it possible?

Modularity!

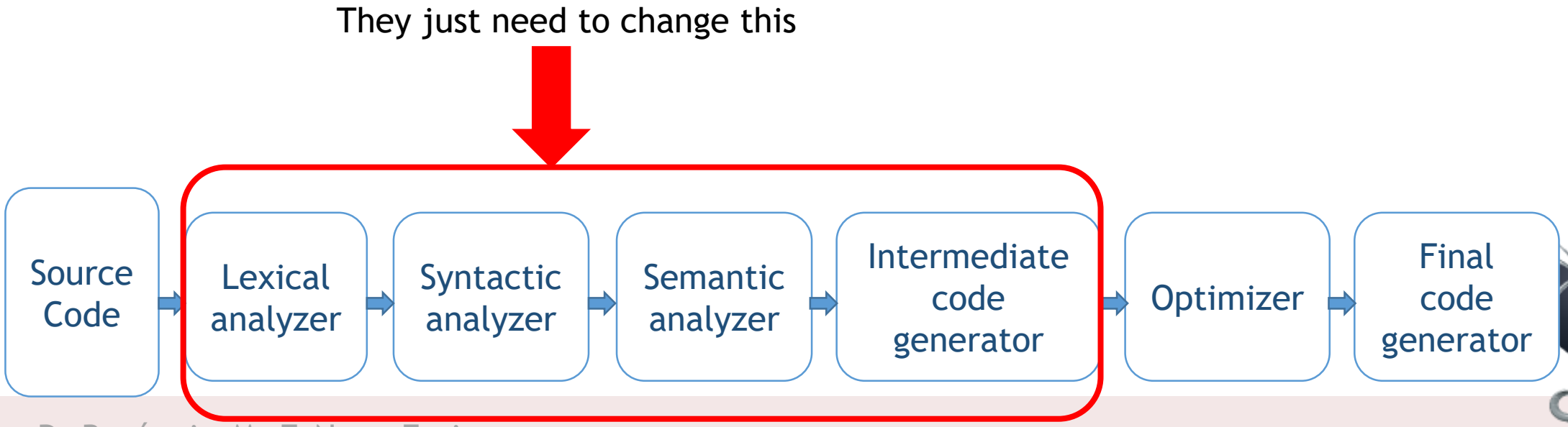
llvm.org/Logo.html

Compilers

High-level programming languages abstract from hardware details

register count, instruction set, etc.

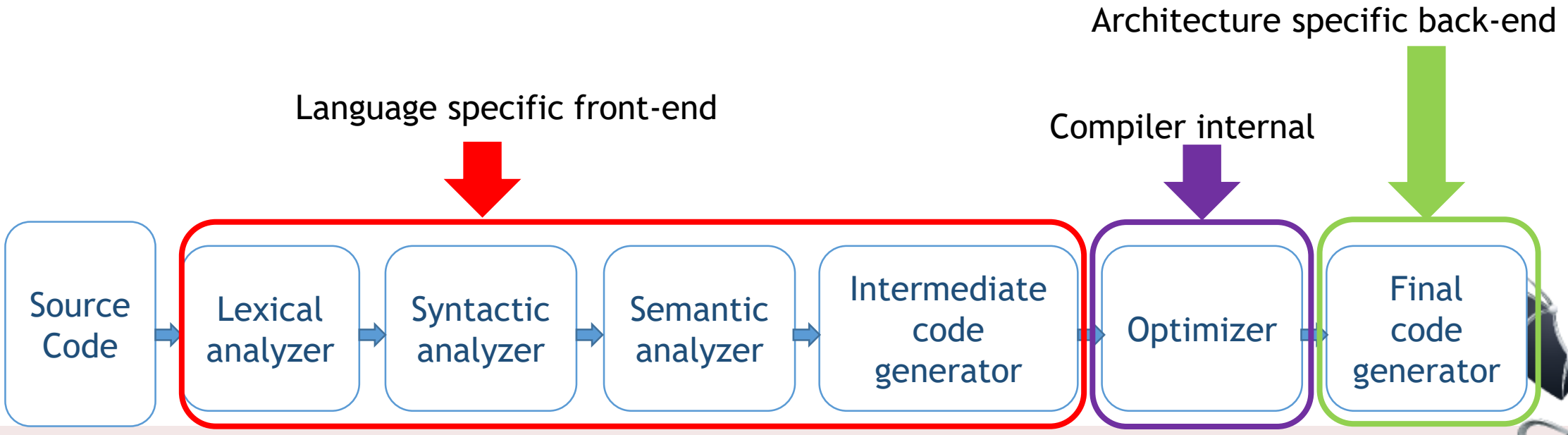
So a language is just a collection of syntax + semantics



Compilers

This means that the optimizer can be the same for multiple languages

And for multiple architectures!



Compilers

This means that the level of optimization

(i.e. how efficient code is generated)

only differs between languages because of language expressivity!

llvm.org/Logo.html

Optimizations

Some typical optimizations that the compilers carries out:

- Loop optimizations
  induction analysis, fission, fusion, inversion, interchange, invariant extraction, nest optimization, reversal, unrolling, splitting
- Data-flow optimizations
  common subexpression elimination, constant evaluation, reordering of load/store ops
- Algebraic / Floating-Point optimizations
  add, mul associativity, distributivity, factoring, division by multiplicative inverse, commutativity
- Instruction-level parallelization
  generating SSE/AVX automatically

Optimizations

So usually, you don’t need to know about details, just enable optimizations in your compiler...

Nevertheless, the compiler can only reason about constructs that it knows about!

llvm.org/Logo.html

Optimizations

Things to be aware of:

- The compiler can completely see through classes, struct and complex objects initialization, member selection, destruction etc. as far as there are no pointers/heap access is involved!

- This means that you should not sacrifice code readability for speed!

llvm.org/Logo.html

Optimizations

Things to be aware of:

- Pointers, new/delete/dynamic allocations, volatile variables, virtual methods, input from peripherals / streams / files breaks the compilers sight and -as such- optimizations.

  Although, the first part - especially virtual method optimizations - are constantly being improved!

llvm.org/Logo.html

Optimizations

Things to be aware of:

- The compiler can only reason about code, that it knows what it means!

Intrinsics!

Intrinsics

An intrinsic is usually a function, that the compiler knows specifically how to handle.

- SSE/AVX instructions as intrinsics
- Special mathematical functions
- Data movement and bitwise manipulation

The special mathematical function intrinsics may highly overlap with standard built-in function support. Check with your compilers manual!

llvm.org/Logo.html

Intrinsics

What is the difference between a built-in version and a hand made one?

- The compiler knows how to efficiently transcode it into low-level instructions
- Guaranteed precision up to the last digit
- May know how to handle special values
- May handle algebraic relations for it
- Have special implementations for different data types
  int, float, double, ...

In the hand made case none of these apply!

llvm.org/Logo.html

Math Intrinsics

Partial list of what is usually supported:

```
abs, acos, asin, atan, atan2, ceil, cos, cosh, exp,
fabs, floor, fmod, ln, log10,
memory {compare, copy, set},
pow, rot, sin, sinh, sqrt,
string {concat, cmp, cpy, length},
tan, tanh
```


llvm.org/Logo.html

Static Analysis

Static analysis consist of a special compilation, that runs specific algorithms to find some complex error types, that are usually made by developers.

Many compilers support some form of static analysis.

Some things they can catch:

- Uninitialized variables, dangling references
- Null- and some invalid pointer errors,
- double delete, use-after-delete, memory leaks

llvm.org/Logo.html

Flags...

At least once, take the time and read through the flags of your compiler:

- How to enable optimizations?
- How to enable vectorization and special instructions (SSE, AVX)
- What intrinsics are supported?
- How does optimizations affect numerics? (fast vs. precise)
- How to enable static analysis?

llvm.org/Logo.html