M0.2 - Compiler Architecture

Guillermo Polito

Goals

- Understand compiler architectures
  - Phases
  - Intermediate representations
  - Common terminology

Compilers

```
MyClass >> foo
    ^ 1 + 17
```


```
push 1
push 17
send +
returnTop
```

A program that translates a program in a *source* language to a *target* language

Target Code

- Another programming language => we talk about transpilation
  e.g., Pharo to C translation

- Some binary code for a virtual machine
  e.g., the Pharo bytecode

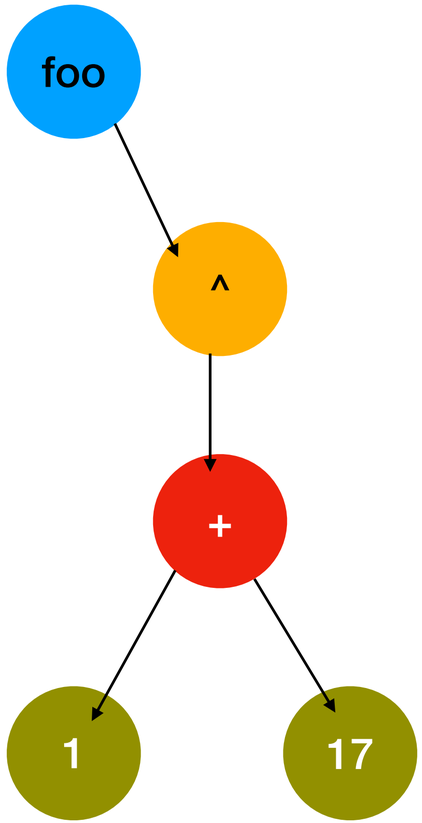
- Some binary code for a real machine
  e.g., machine code for x86, or ARMv8

Compiler Internals Example: Simple Compiler

Source code

```
MyClass >> foo
    ^ 1 + 17
```

Parsing



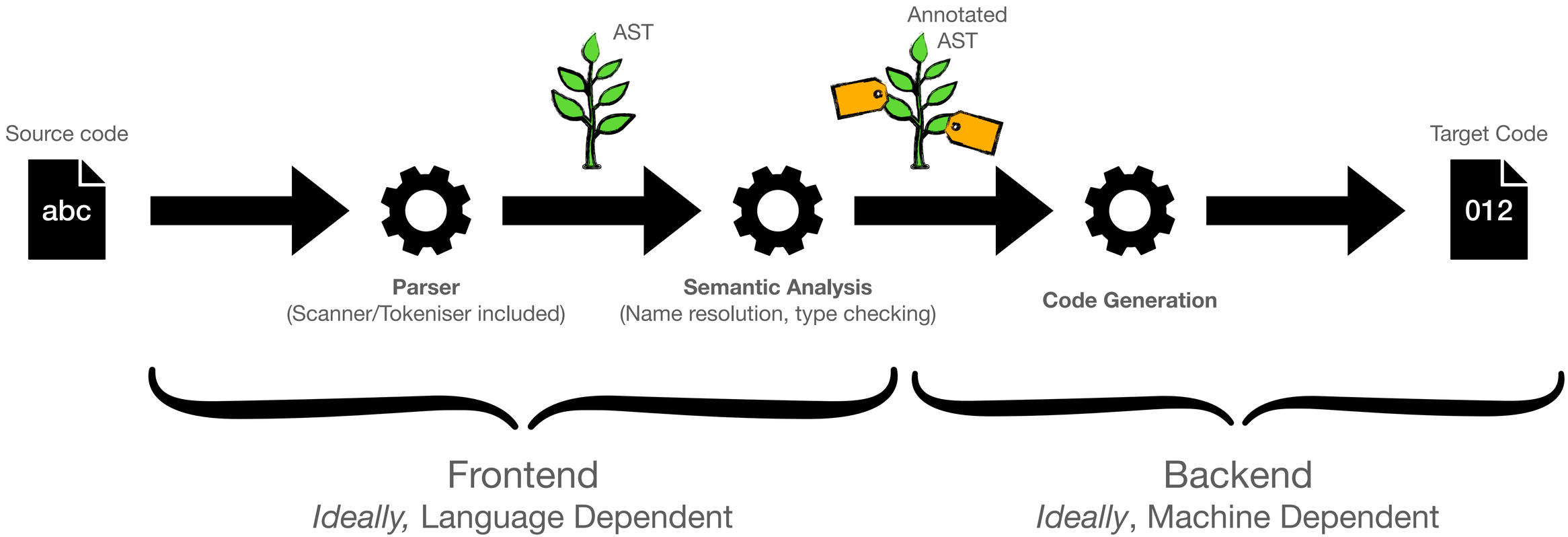
Intermediate Representation

Code Generation

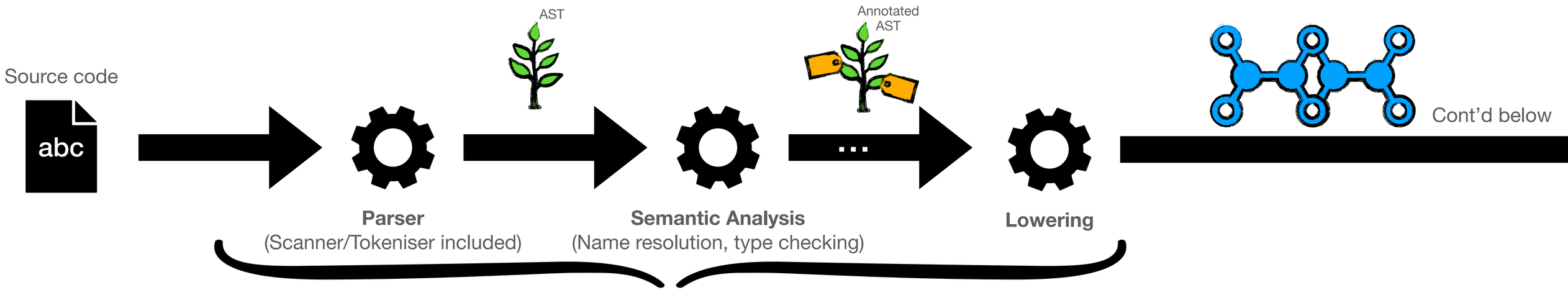
Target Code

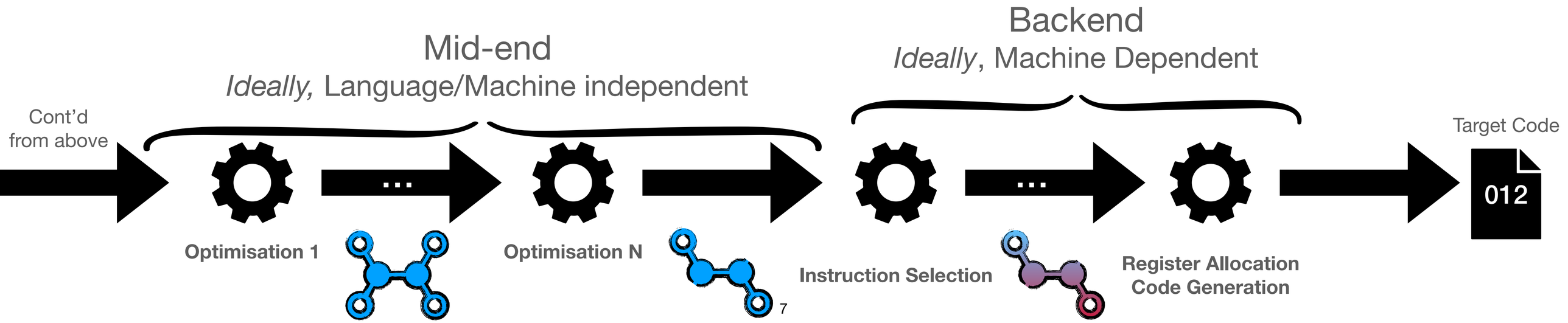
```
16r76
16r20
16rB0
16r7C
```

Compiler Phases by Example



Modern Optimizing Compiler Phases

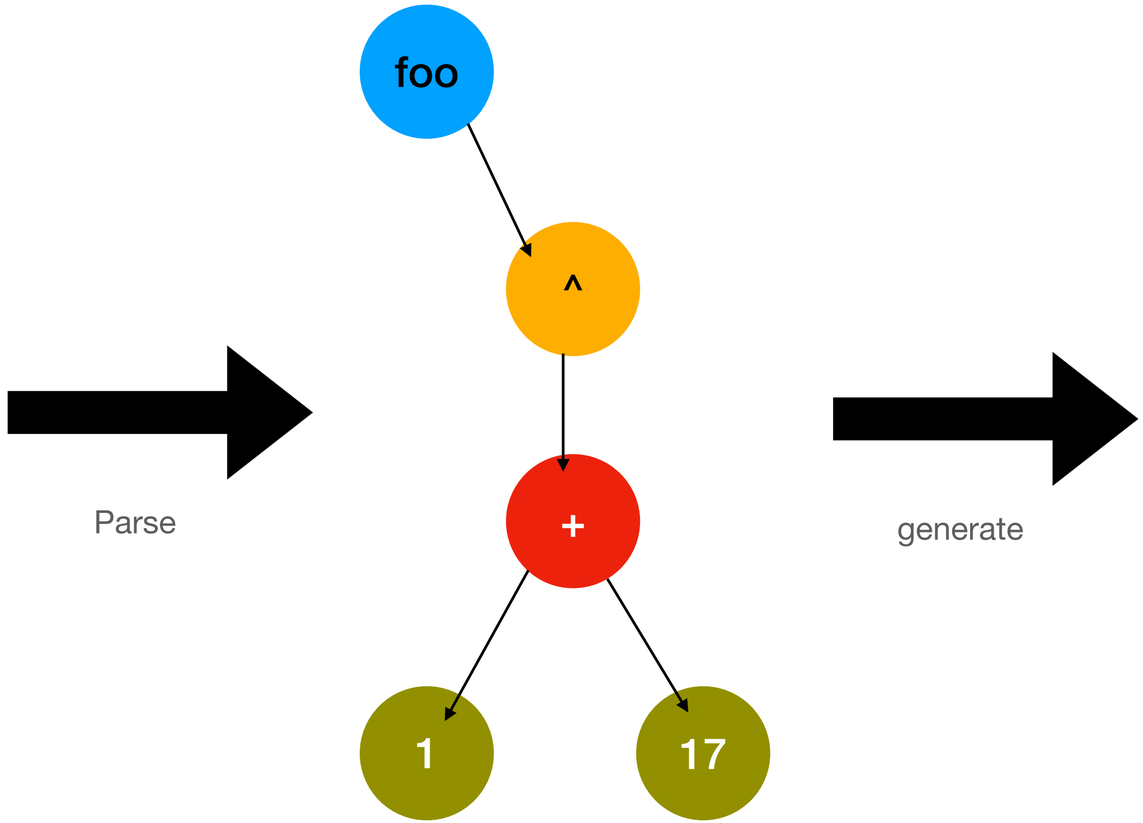




Real Example: The old Pharo Compiler

Source code

```
MyClass >> foo
    ^ 1 + 17
```



Parse

Intermediate Representation

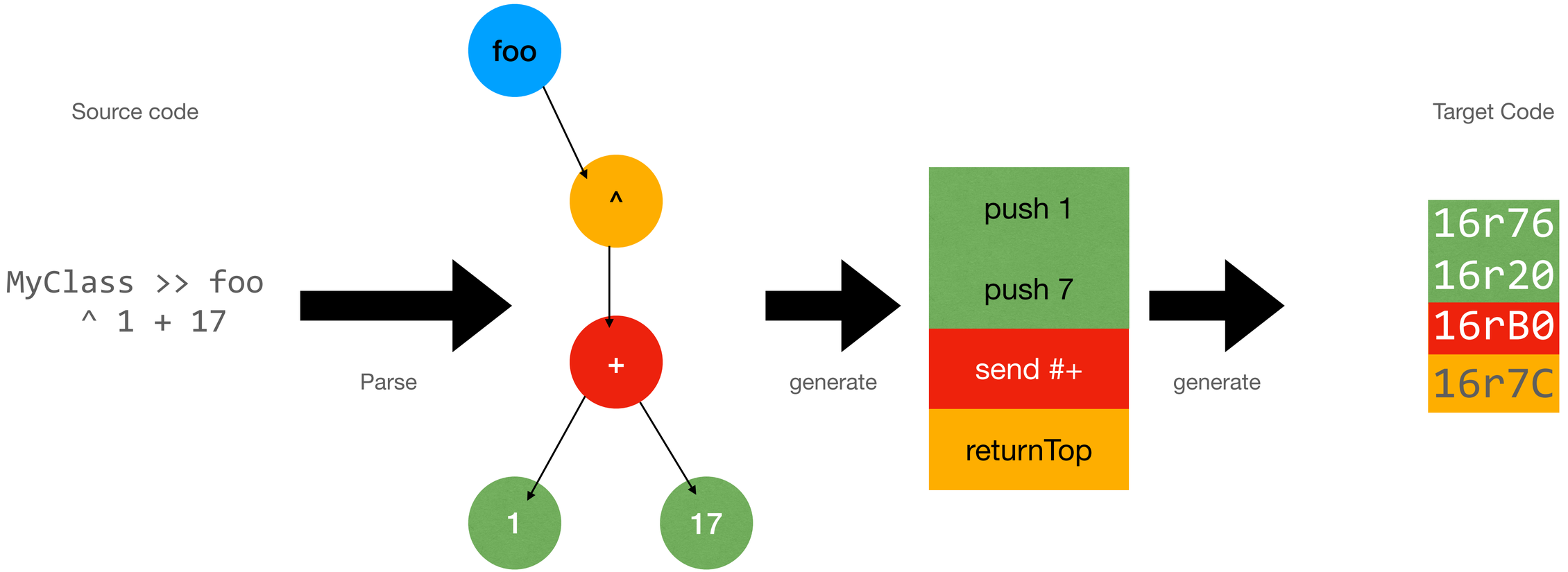
generate

Target Code

```
16r76
16r20
16rB0
16r7C
```

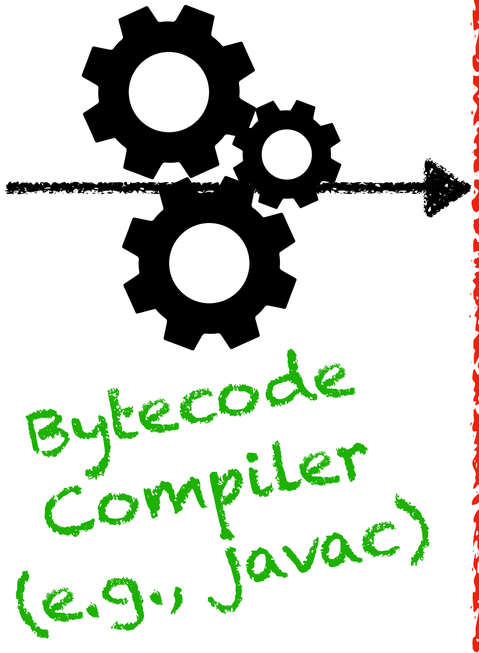
Real Example: The Pharo Opal Compiler

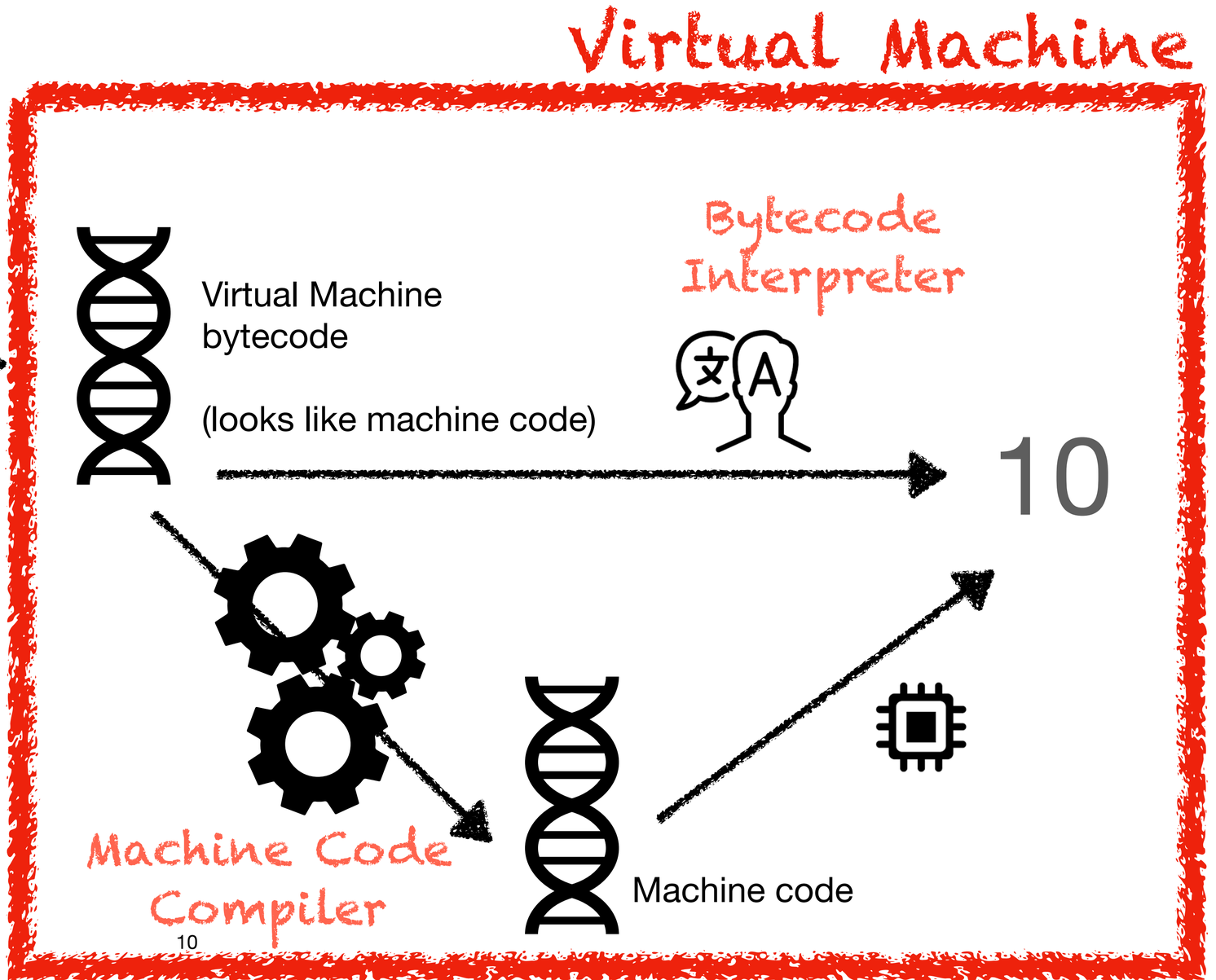
Introducing linear intermediate representations



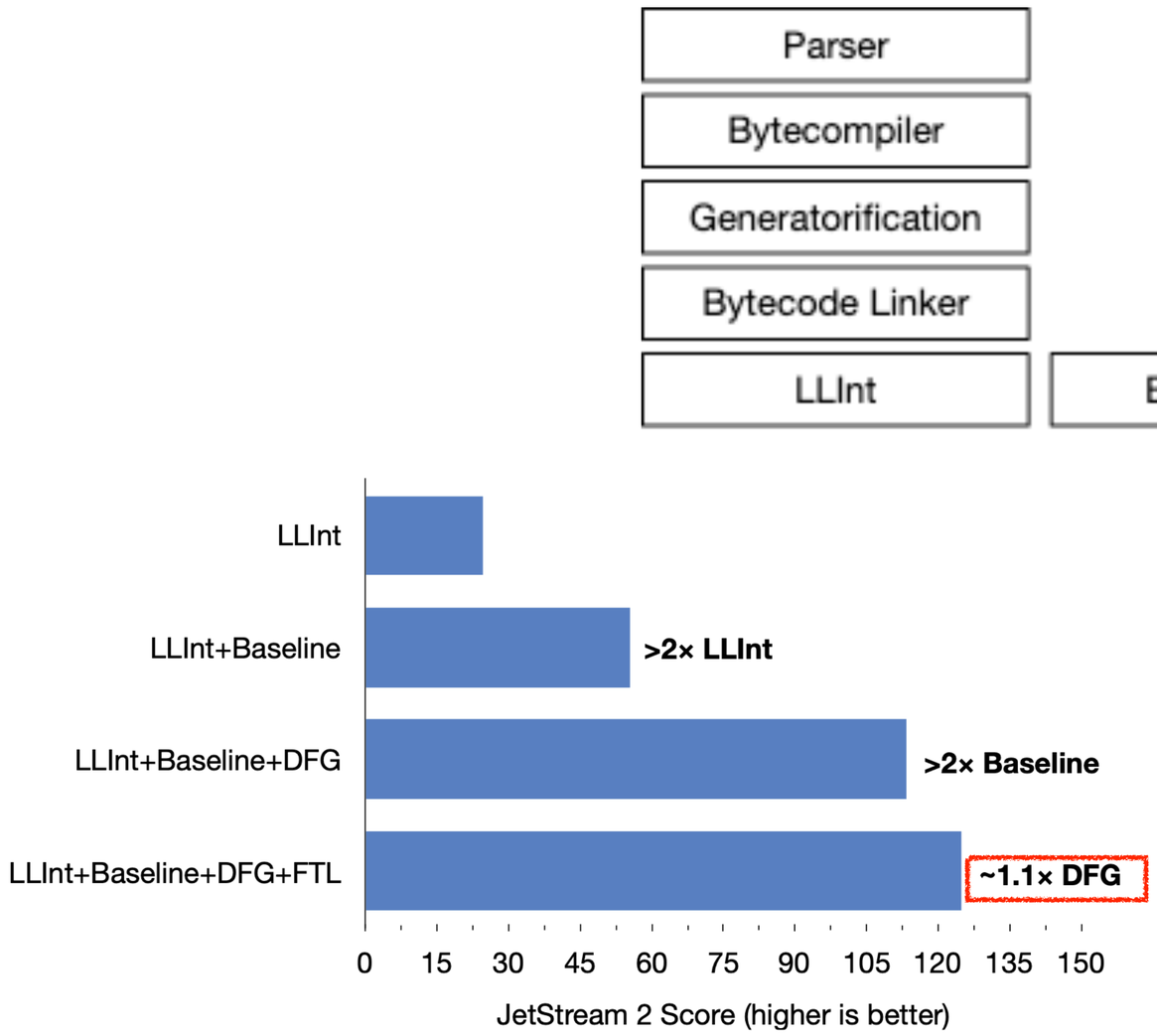
Modern Languages: Compiler*s* + Interpreters

```
a := 1;
if (condition){
  a := a + 7;
}
return a + 2;
```





Javascript Core JIT Compiler



Parser

Bytecompiler

Generatorification

Bytecode Linker

LLInt

Baseline JIT

DFG Compiler

DFG Bytecode Parser

DFG Optimizer

DFG Backend

FTL Compiler

DFG Bytecode Parser

Extended DFG Optimizer

Lower DFG to B3

B3 Optimizer

Instruction Selector

Air Optimizer

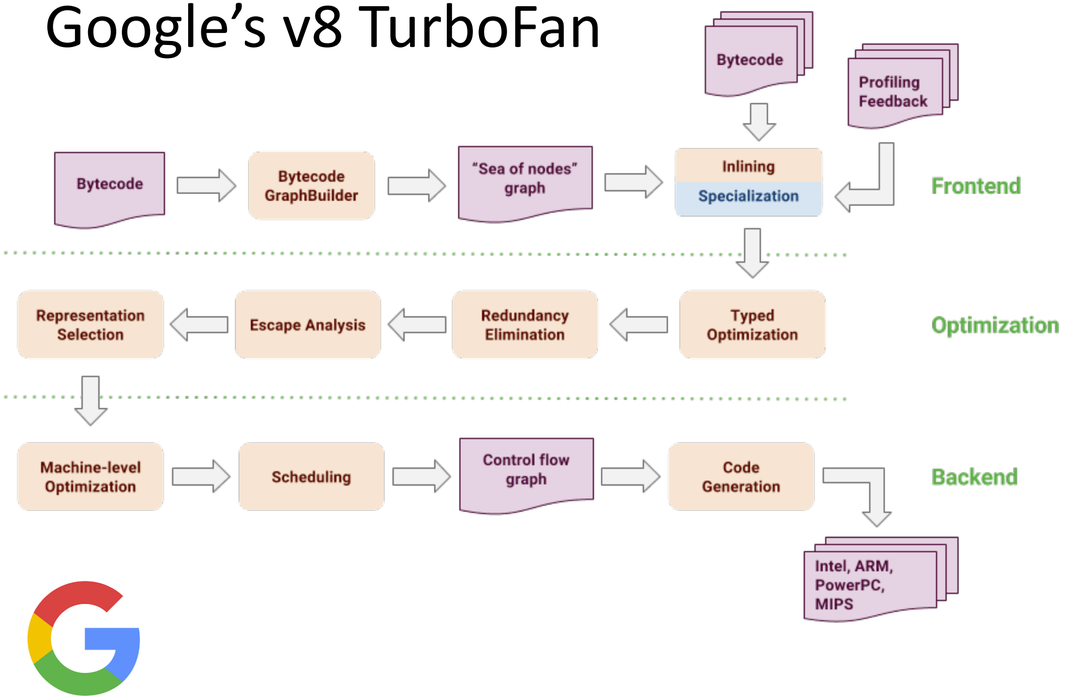
Air Backend

Multiple levels
Different representations
Engineering cost!

https://webkit.org/blog/10308/speculation-in-javascriptcore/

V8 and JSCore in depth

Google's v8 TurboFan

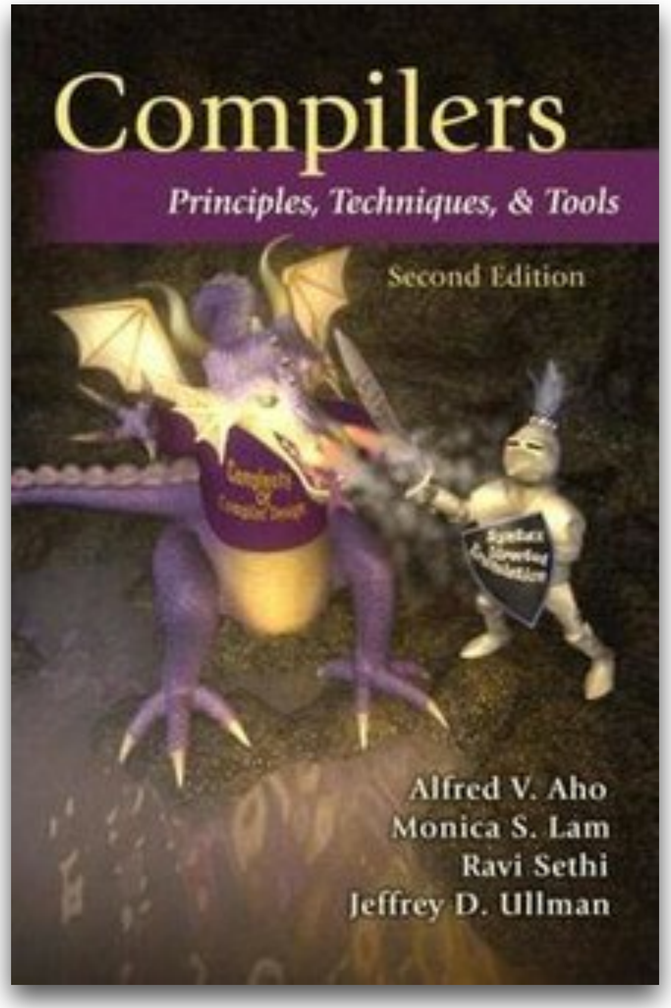


Apple's JavascriptCore[2021]

https://webkit.org/blog/10308/speculation-in-javascriptcore/
https://ponyfoo.com/articles/an-introduction-to-speculative-optimization-in-v8

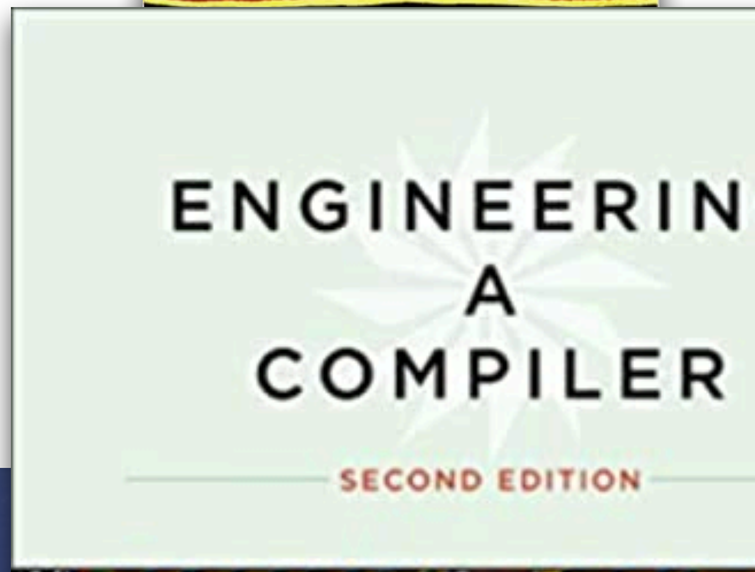
Compiler Frontends

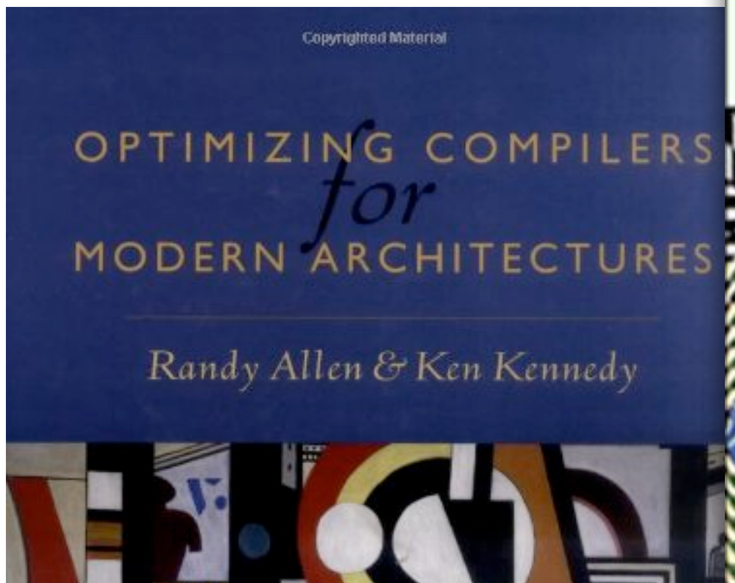
- ***Extensively*** studied in books
  - E.g., Parsers/Lexers/Scanners are >80% of the dragon book
  - Check it :)
  - The dragon book is not the only one



Compiler Mid- and Back- ends

- ***Fewer books*** covering these topics
  - Info is in research papers
  - Still an active area of research





Intermediate Representation design:

Notes and Terminology

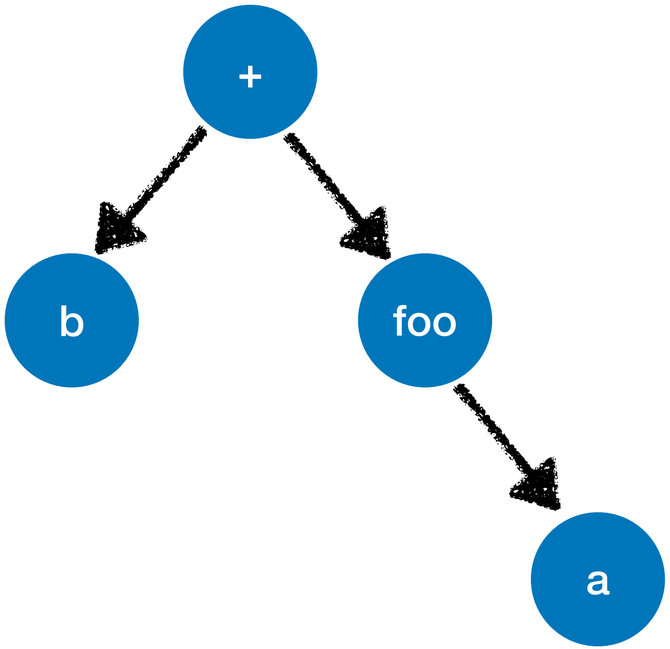
IRs: Data structures to represent code

```
b + foo(a);
```

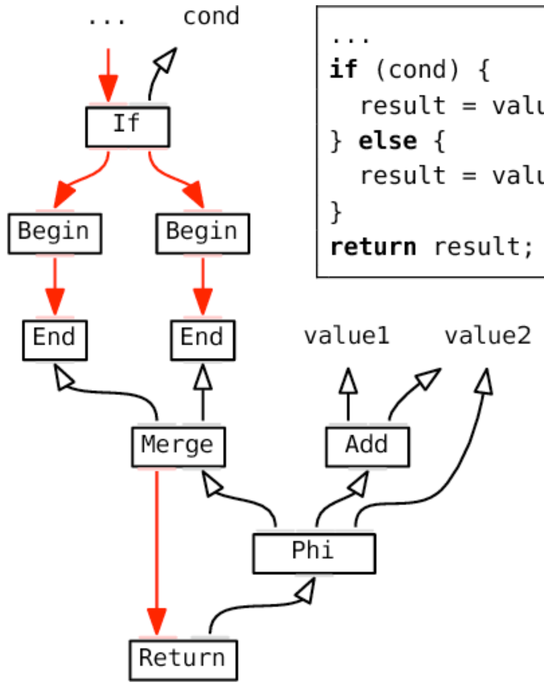
- Lists

LOAD R1, b
MOV R2, R1
LOAD R1, a
CALL FOO
ADD R1, R1, b

- Trees



- DAGs




```
...
if (cond) {
  result = value1 + value2;
} else {
  result = value2;
}
return result;
```


Graal IR : An Extensible Declarative Intermediate Representation, APPLC'13

Lists representing code

- Closer to “machine” code
- Simple to manipulate
- Relations between instructions become implicit
  - e.g., how many arguments does foo have?
  - e.g., Answer => sometimes, we need to see foo’s code
  - These become “conventions"

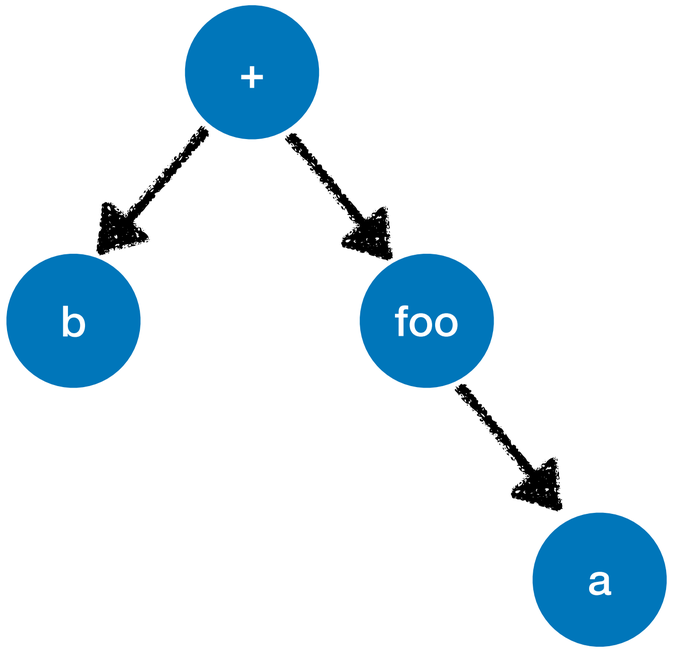
```
b + foo(a);
```

LOAD R1, b
MOV R2, R1
LOAD R1, a
CALL foo
ADD R1, R1, b

Trees representing code

- Closer to source code
- Often produced by a parser
- Relations are explicit
  - e.g., how many arguments does foo have?
  - e.g., Answer => look at foo's children!

```
b + foo(a);
```



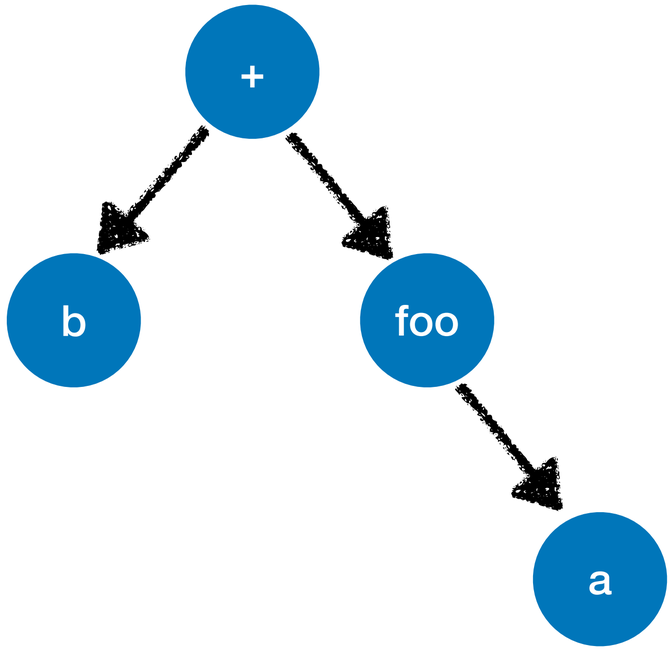
Abstract Syntax Trees (ASTs)

- Trees representing code
- Abstract, because they do not represent ALL elements in the grammar
  - i.e., parentheses, statement finalisers, indentation are **not** in the tree

```
b + foo(a);

b + (foo(a));

b +
  foo(a)
```



Other Possible IR Flavors

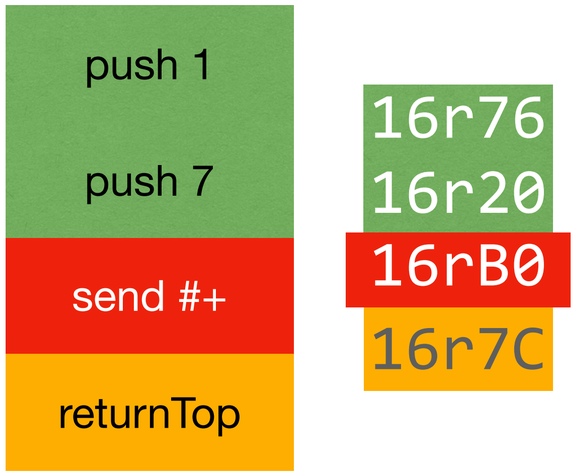
- Control Flow Graphs
  - Nodes represent uninterrupted sequences of instructions
  - Edges represent control-flow
- Sea of nodes
  - Flat representation: control flow and instructions are at the same level
- Single Static Assignment (SSA) form + variants
  - Each variable has max one assignment

On Code Generation:

Notes and Terminology

Target code: virtual "machine code”

- A virtual machine simulates a machine
- The instructions are called **bytecodes**
- Independent of the real machine
- Stack based: operands are exchanged through a stack
- Compact: Stack is implicit



Target code: machine code

- Binary code executed by the machine/CPU/GPU/…
- Bytes encode instructions
- The set of instructions + CPU is called **instruction set architecture** (ISA)
- Each machine/CPU has its own ISA and binary encoding of it
- Typically register machines:
  operands are exchanged through registers and memory

```
#[ 31 32 3 213 76 1 0 88 ]
```

some arm v8 instructions

Machine code VS Assembly code

- Assembly is a programming language
- (not machine code)
- That is translated to machine code using a compiler (aka an assembler)

```
nop
ldr x12, #40
```

```
#[ 31 32 3 213 76 1 0 88 ]
```

```
nop          ldr x12, #40
```

For simplicity: we will use assembly examples to represent machine code

Machine code instructions

- Basic instructions:
  - Write data to memory
  - Read data from memory
  - Arithmetic (+ , - , * …)
  - Bit instructions (shift, bitAnd …)
  - Control flow (jump, jump if)

```
mnemonic op1 op2 op3 …
```

```
e.g.,
  load r1 [r1, #40]
  add r3 r2 r1
  store [r1, #40] r3
```

Machine code: Instruction Operands

Basic operand types

- Immediate numbers: encoded directly in the instruction bytes

  e.g., `#40`

- Registers: small memories in the CPU

  e.g., `r1 r2 r3`

- Memory addresses: calculated from registers and/or immediates

  e.g., `[r1, #40]`

```
load r2 [r1, #40]
add r3 r2 r1
store [r1, #40] r3
```

Machine code: Registers

- Small memories in the CPU
- Some are for general use
- Some are specific (e.g., Floating point numbers)
- Some are for the CPU (and not for us)

=> And there are very few!!!

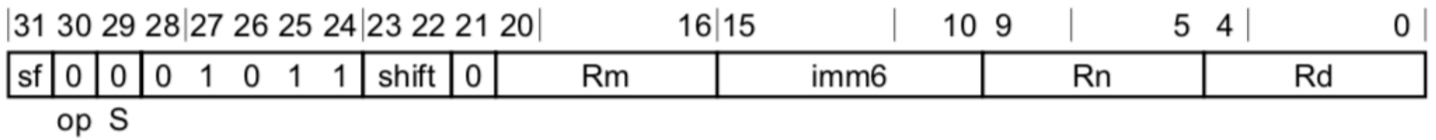
```
e.g.,
  load r1 [r1, #40]
  add r3 r2 r1
  store [r1, #40] r3
```

Machine code: binary encoding

Each ISA has its manual

C6.2.5 ADD (shifted register)

Add (shifted register) adds a register value and an optionally-shifted register value, and writes the result to the destination register.



32-bit variant

Applies when `sf == 0`.

```
ADD <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

64-bit variant

Applies when `sf == 1`.

```
ADD <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

Arm® Architecture Reference Manual

Armv8, for Armv8-A architecture profile

Takeaways

- Compilers are organized in Phases
- Compilers represent code using Intermediate Representations (IRs)
- Compilers are designed with different IRs to achieve different trade-offs
- Phases either
  - modify some code representation e.g., optimizations
  - change some code representation into another e.g., a parser

Material

- Books
  - Dragon Book
  - Advanced Compiler Design & Impl
  - Engineering a Compiler
- Blogs
  - V8, JSCore