Compilers 101: intermediate representations, machine code, assembler

(a prelude to the JIT presentation)

What is a compiler

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


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

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

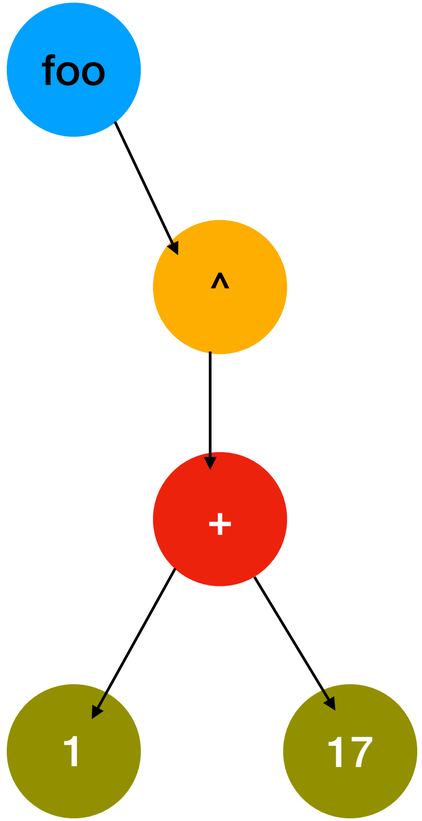
Overview of a compiler internals

An example with bytecode

Source code

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

Parse



Intermediate Representation

generate

Target Code

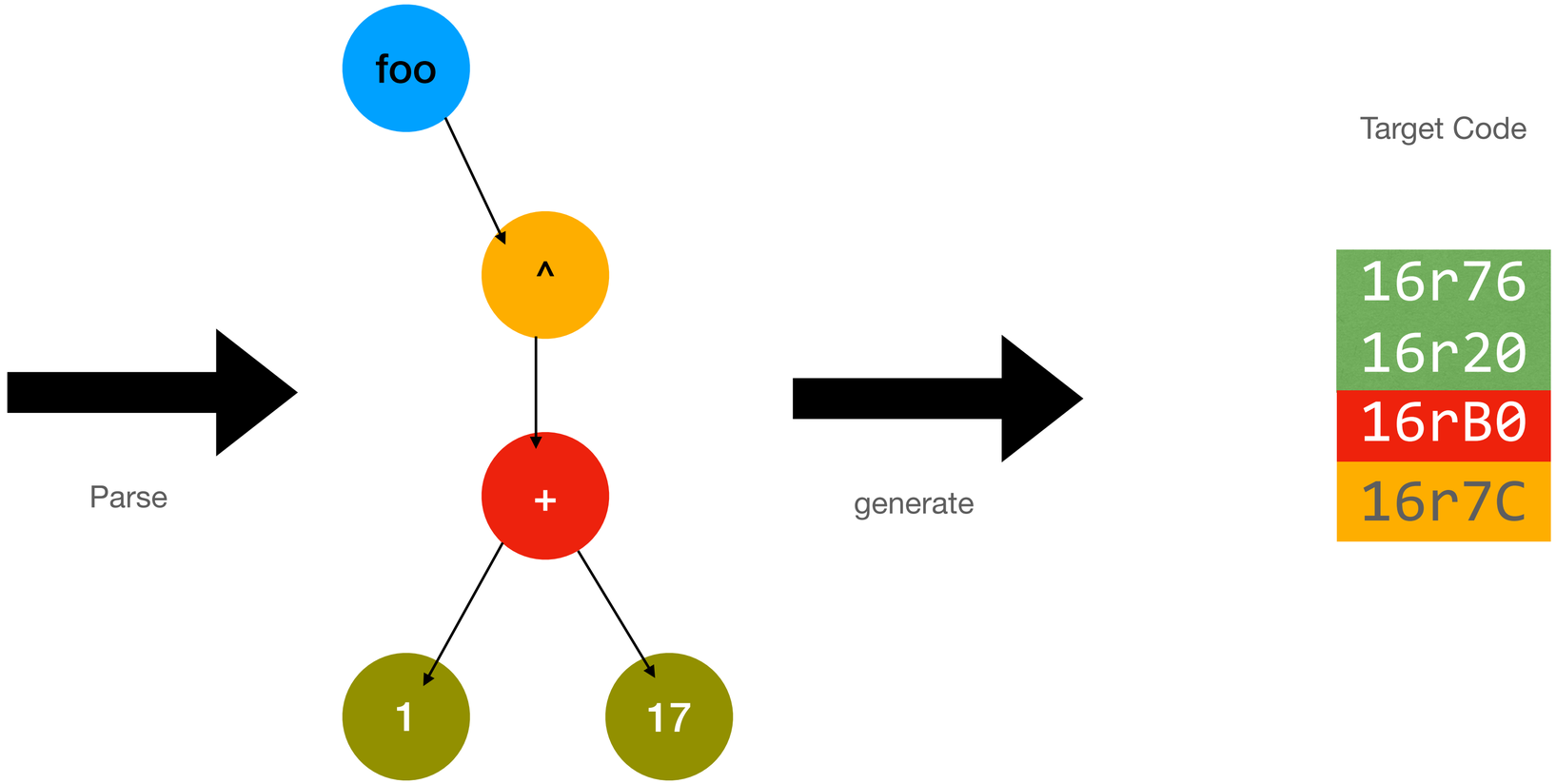
```
16r76
16r20
16rB0
16r7C
```

Example 1: The old Pharo Compiler

AST as intermediate representation

Source code

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

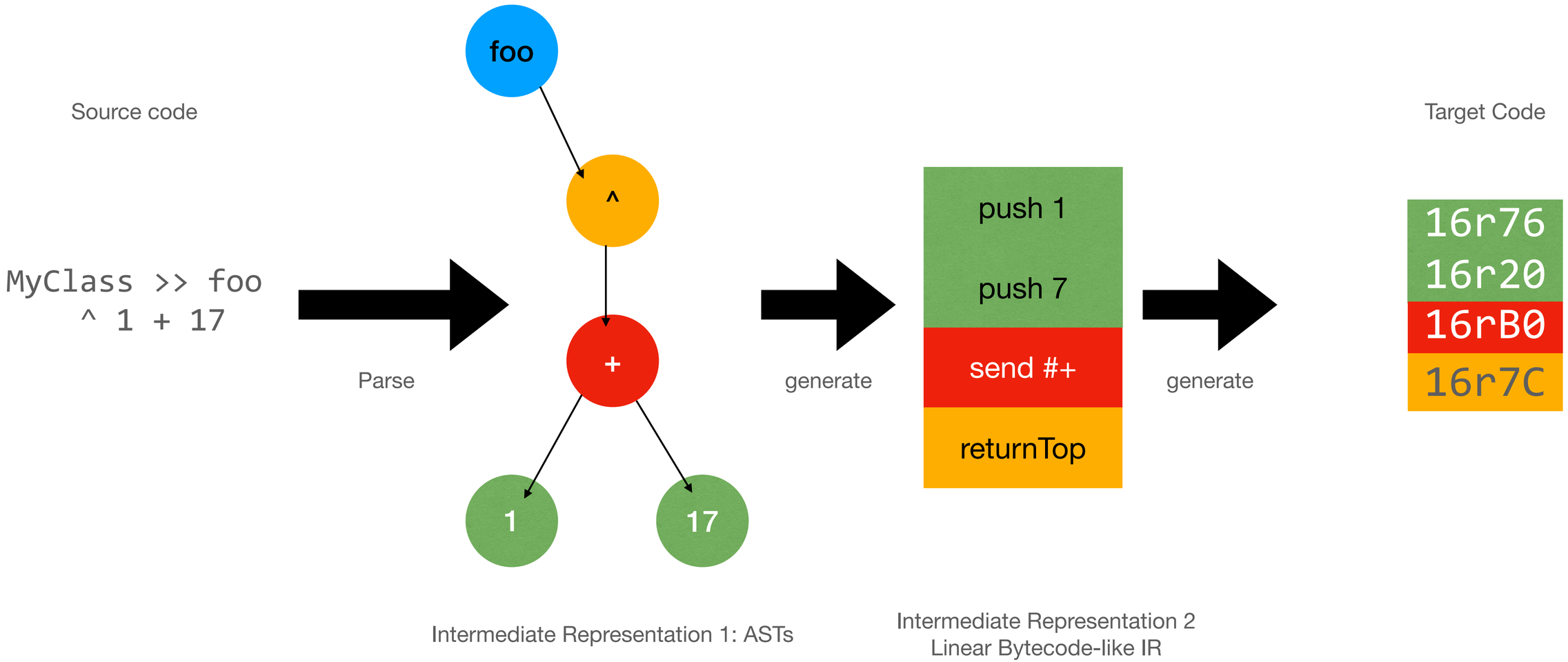


Target Code

Intermediate Representation

Example 2: The Opal Compiler

Introducing linear representations



What is the target code?

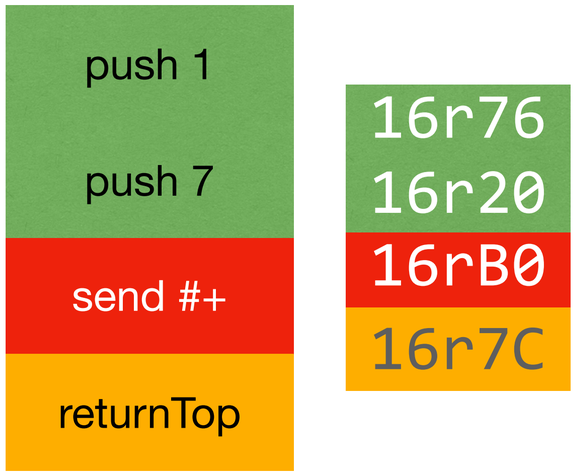
- Another programming language => we talk about transpilation
  e.g., Pharo to C translation

- 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

Target code: bytecode as virtual "machine code”

- The 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

- The real binary code executed by the machine
- 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

Before going deep: 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.,

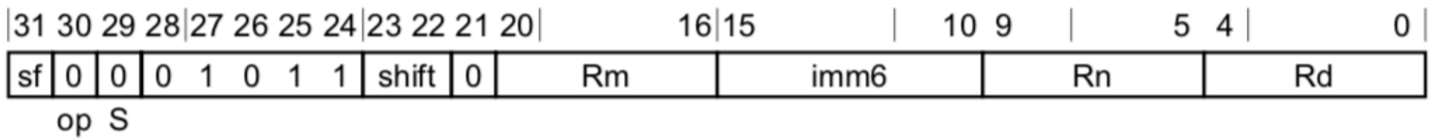
```
load r1 [r1, #40]
add r3 r2 r1
store [r1, #40] r3
```

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

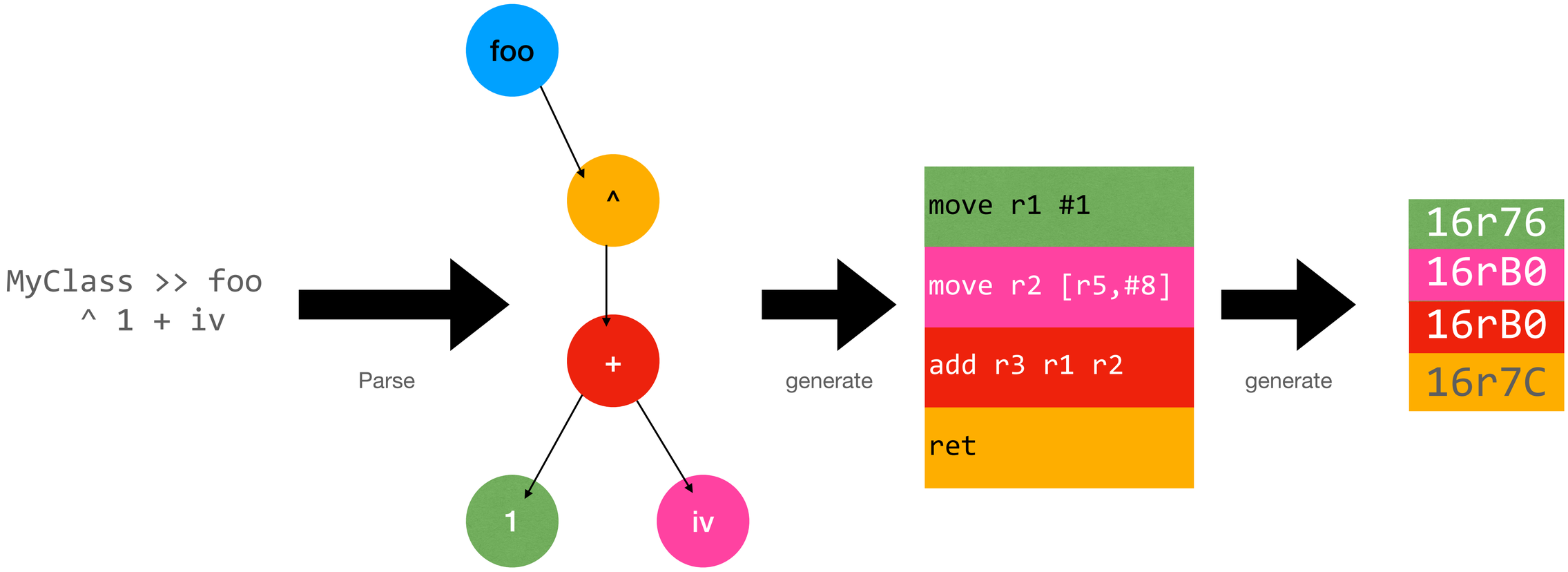
Applies when `sf == 1`.

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

Arm® Architecture Reference Manual

Armv8, for Armv8-A architecture profile

Conclusion



Source code

Tree IR
Source-code like

Linear IR
Bytecode, Assembly like

Target Code
Binary bytecode or machine code