This is a draft! To be completed incrementally 2018-2019

Selfie: Introduction to the Implementation of Programming Languages, Operating Systems, and Processor Architecture

Christoph Kirsch and Sara Seidl
Department of Computer Sciences
University of Salzburg
2018

selfie.cs.uni-salzburg.at/slides

What?

- This presentation is part of a 3-semester introduction to architecture, compilers, and operating systems using the selfie system.

- Selfie is a minimal, fully self-contained 64-bit implementation of a self-compiling C compiler, RISC-V emulator, and RISC-V hypervisor. Selfie can compile, execute, and virtualize itself any number of times!

- Everything needed is implemented in a single, 10-KLOC file of C code. There are no includes and no external libraries. For bootstrapping, a C compiler will do.

- The purpose of selfie is to teach the implementation of programming languages (C subset), operating systems (virtual memory, concurrent processes), and processor architecture (RISC-V subset), all using the same code!

Why?

- Selfie shows, from first principles, how a minimal but still realistic hardware and software stack works.

- Anyone understanding elementary arithmetic and Boolean logic may be able to follow!

- The goal is to see and understand how the semantics of programming languages and the concurrent execution of programs is constructed using nothing but bits.

- The self-referential nature of the construction is an important part of selfie. Seeing how to resolve it establishes a well-founded understanding of basic computer science principles.

How?

- We use the selfie code to explain everything. Students will need to read, understand, and modify the code to follow the course.

- Instead of copying code here we provide clickable links to actual selfie code snippets on github.com

- Standard terminology is introduced with clickable links to wikipedia.org

Course Material

- Website: selfie.cs.uni-salzburg.at

- Slides (draft): selfie.cs.uni-salzburg.at/slides

- Book (draft, outdated): leanpub.com/selfie

- Sources (code, slides, book): github.com/cksystemsteaching/selfie

- Install selfie on your machine or in the cloud using the instructions provided in the selfie repository on github.com

- Please note that the slides are incomplete as of 2018 and published incrementally as they become available.

Syllabus

1. Programming in C*, the C subset in which selfie is written and compiles.

2. Introduction to RISC-U, the RISC-V subset targeted, emulated, and virtualized by selfie.

3. Introduction to starc, the selfie compiler (scanner, parser, type checker, register allocator, code generator).

4. Introduction to mipster, the selfie emulator (virtual and physical memory, machine contexts).

5. Introduction to hypster, the selfie hypervisor (virtual memory, context switching).

6. Introduction to monster, the selfie symbolic execution engine (planned).

Programming in C*

- C* is a tiny subset of the programming language C

- C* supports only 2 data types: unsigned integer, `uint64_t`, and pointer to unsigned integer, `uint64_t*`. There are no signed integers and no composite data types.

- C* features the unary * operator as the only means to access heap memory hence the name C*. There are no arrays and no structs in C*.

- C* features 5 statements (assignment, if-else, while loop, procedure call, return).

- C* has 3 types of literals (signed decimal number, character, string).

- C* supports 5 arithmetic operators (+, −, *, /, %) and 6 comparison operators (==, != ,< ,<= ,> , >=). There are no bitwise operators and no Boolean operators.

atoi in C*

(proper character and exception handling removed)

```c
uint64_t atoi(uint64_t* s) {
  uint64_t n;

  n = 0;

  // loop until s is terminated
  while (*s != 0) {
    // use base 10, offset by '0'
    n = n * 10 + *s - '0';

    // go to next digit
    s = s + 1;
  }

  return n;
}
```

atoi **stands for**
__ASCII__ **to integer**

Given a string s **of**
decimal digits,
the atoi **code computes**
the numerical value n
represented by s

Click atoi to see the actual code in selfie.
We provide such links throughout the presentation.

Programming Language C

TIOBE Programming Community Index
Source: www.tiobe.com

Legend: Java, C, C++, Python, C#, Visual Basic .NET, PHP, JavaScript, Ruby, SQL
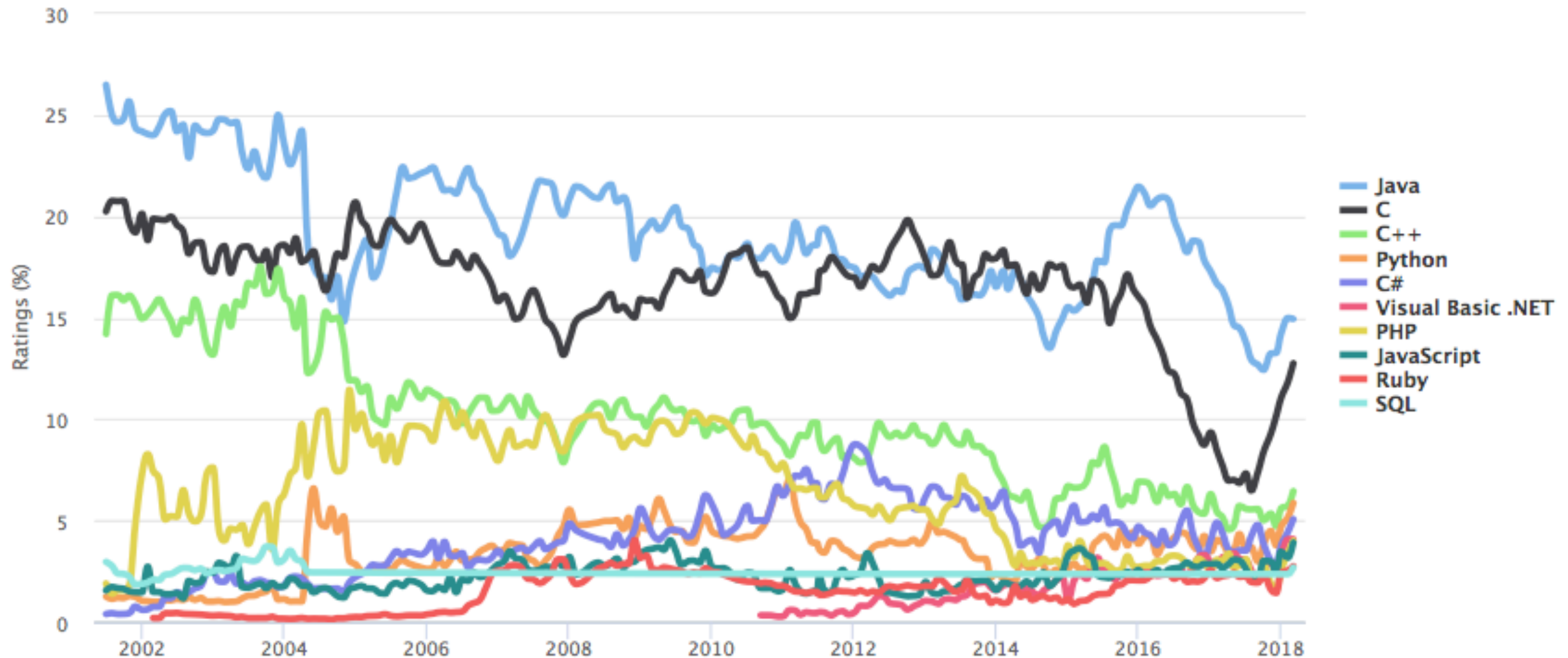
C with all its dialects is still the most popular programming language

https://www.tiobe.com/tiobe-index

Why C?

- This course aims at much more than just learning how to code.

- The choice of programming language is important but not more important than many other choices made here.

- C is relevant, C* is simple.

- With <u>imperative programming</u> the notion of program <u>state</u> maps one-to-one to the notion of machine state.

- Seeing the pitfalls of imperative programming enables students truly appreciate other paradigms such as <u>functional programming</u>.

- In fact, awareness of state makes students understand the value of functional programming. This is nevertheless left to others to teach.

C* by Example

procedure

pointer type

formal parameter

return type

```
uint64_t atoi(uint64_t* s) {
    uint64_t n;
```

integer type

local variable

integer literal

dereferencing

```
    n = 0;
```

assignment

comparison

```
    // loop until s is terminated
    while (*s != 0) {
        // use base 10, offset by '0'
        n = n * 10 + *s - '0';
```

while loop

expression

integer arithmetic

```
        // go to next digit
        s = s + 1;
    }
```

character literal

```
    return n;
}
```

pointer arithmetic

return statement

C* Integers and Pointers

- C* integers are unsigned 64-bit integers

- In C* there are five arithmetic operators: +, -, *, /, %

- And six comparison operators: ==, !=, <, <=, >, >=

- C* pointers are 64-bit pointers to C* integers

- And pointer arithmetic: +, -

C* versus C Integer Literals

- C* integer literals are unsigned 64-bit

- C integer literals are signed 32-bit

- For example, `1/-1==0` in C* but `1/-1==-1` in C

- And, `1%-1==1` in C* but `1%-1==0` in C

- Also, `1<-1` and `1<=-1` hold in C* but `1>-1` and `1>=-1` do not whereas the opposite is true in C

- The semantics of `/` and `%` as well as `<`, `<=`, `>`, and `>=` is different for signed and unsigned integers!

C* Characters and Strings

- C* characters are <u>ASCII</u>-encoded.

- C* character literals are characters in code like `'c'`.

- C* strings are stored as null-terminated sequences of characters.
  Alternatively the end of a string could be identified by storing the number of characters at its beginning.

- C* string literals are strings in code like `"this"`.

- The difference between `'a'` and `"a"`.

ascii representation
(**numerical value**)
in memory

pointer to first
word of where the
string is stored

C* versus C Strings

- C* strings are arrays of unsigned 64-bit integers

- C strings are arrays of characters, that is, of type `char`

- For example, `*"Hello World!"` is equal to `0x6F57206F6C6C6548` in C* but `0x48` in C

- Note that `0x48` is ASCII for `H` while `0x65`, `0x6C`, `0x6F`, `0x20`, and `0x57` are ASCII for `e`, `l`, `o`, `space`, and `W`, respectively.

Selfie Code Structure
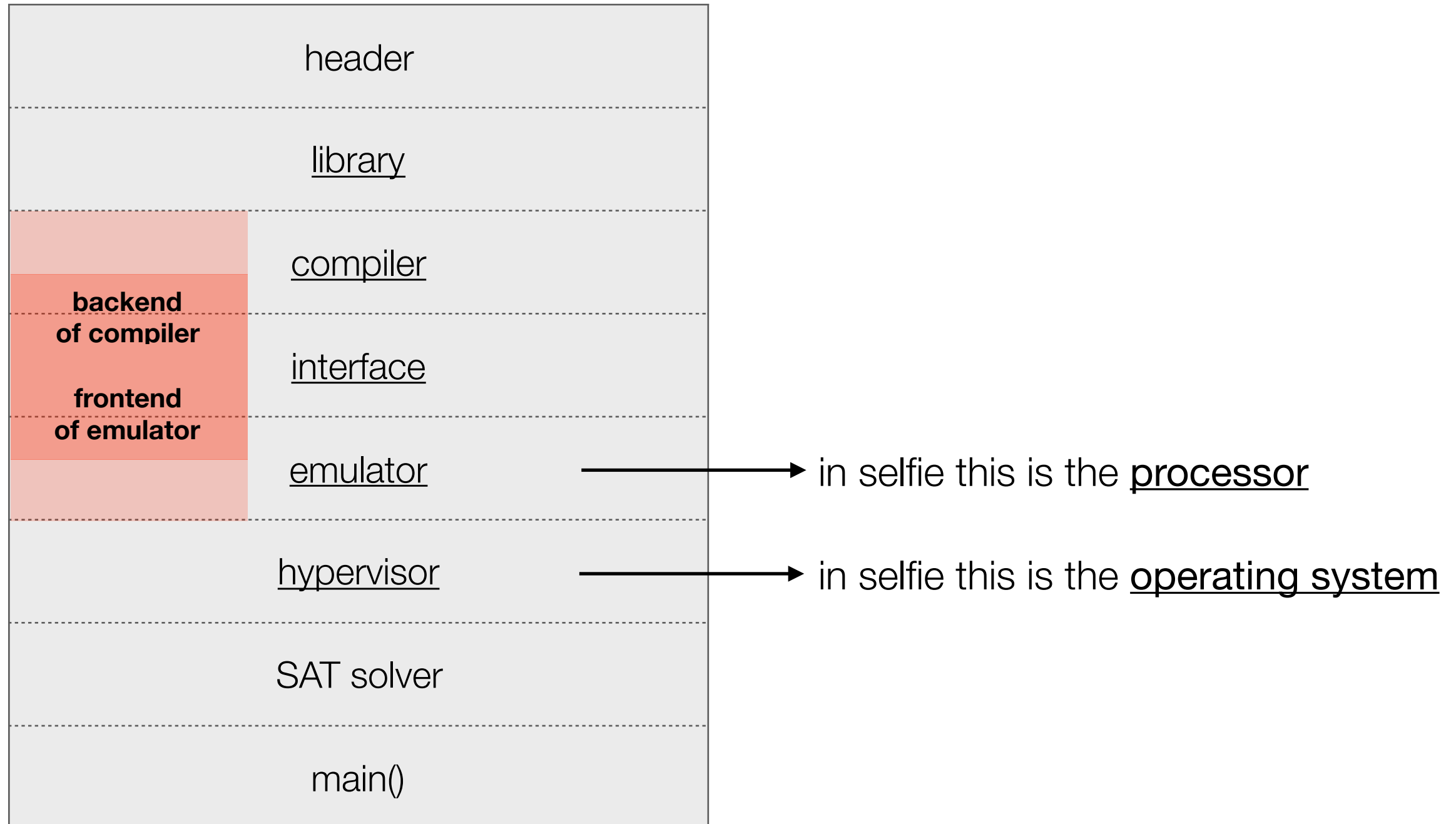
The Selfie Library

```
uint64_t leftShift(uint64_t n, uint64_t b);
uint64_t rightShift(uint64_t n, uint64_t b);

uint64_t getBits(uint64_t n, uint64_t i, uint64_t b);
uint64_t getLowWord(uint64_t n);
uint64_t getHighWord(uint64_t n);

uint64_t abs(uint64_t n);

uint64_t signedLessThan(uint64_t a, uint64_t b);
uint64_t signedDivision(uint64_t a, uint64_t b);

uint64_t isSignedInteger(uint64_t n, uint64_t b);
uint64_t signExtend(uint64_t n, uint64_t b);
uint64_t signShrink(uint64_t n, uint64_t b);

uint64_t  loadCharacter(uint64_t* s, uint64_t i);
uint64_t* storeCharacter(uint64_t* s, uint64_t i, uint64_t c);

uint64_t stringLength(uint64_t* s);
void     stringReverse(uint64_t* s);
uint64_t stringCompare(uint64_t* s, uint64_t* t);

uint64_t  atoi(uint64_t* s);
uint64_t* itoa(uint64_t n, uint64_t* s, uint64_t b, uint64_t a, uint64_t p);
```

A Compiler

translation

input

high-level programming language A

output
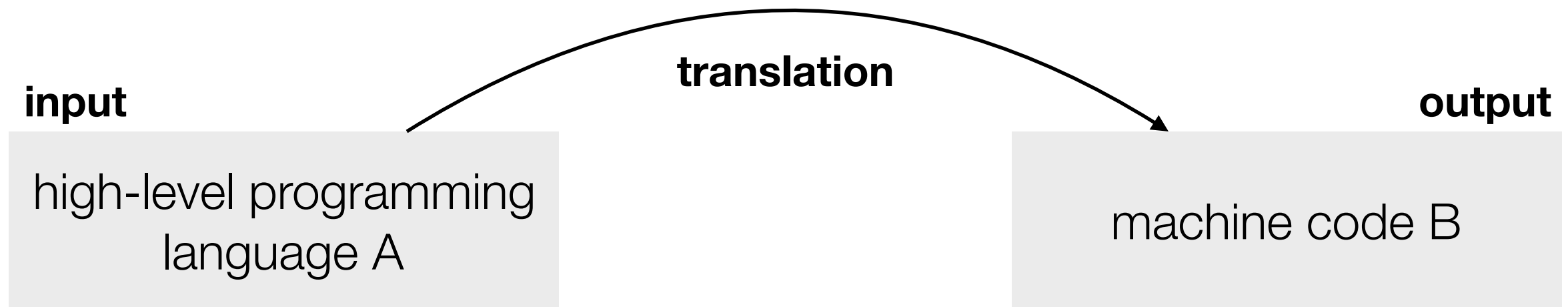
machine code B

- The selfie compiler written in C* translates C* code (self-referential).

- <u>High-level languages</u> have a structure that defines the control flow.

- <u>Machine code</u> has no structure, it is just a sequence of instructions.

- The compiler reads an input program, which is a sequence of characters (ASCII, UTF-8-encoded), and writes machine code.

Using the Selfie Compiler

disassemble binary code

compile to binary code

./selfie -c example.c

-s example.s

-o example.m

output binary code

Von Neumann Machine

CPU

ALU

| reg 31 |
| … |
| reg 0 |

pc

ir

63   0

bottleneck

MAX

word-aligned

16

8

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

63   0

byte-addressed

boot loader

ROM

63   0
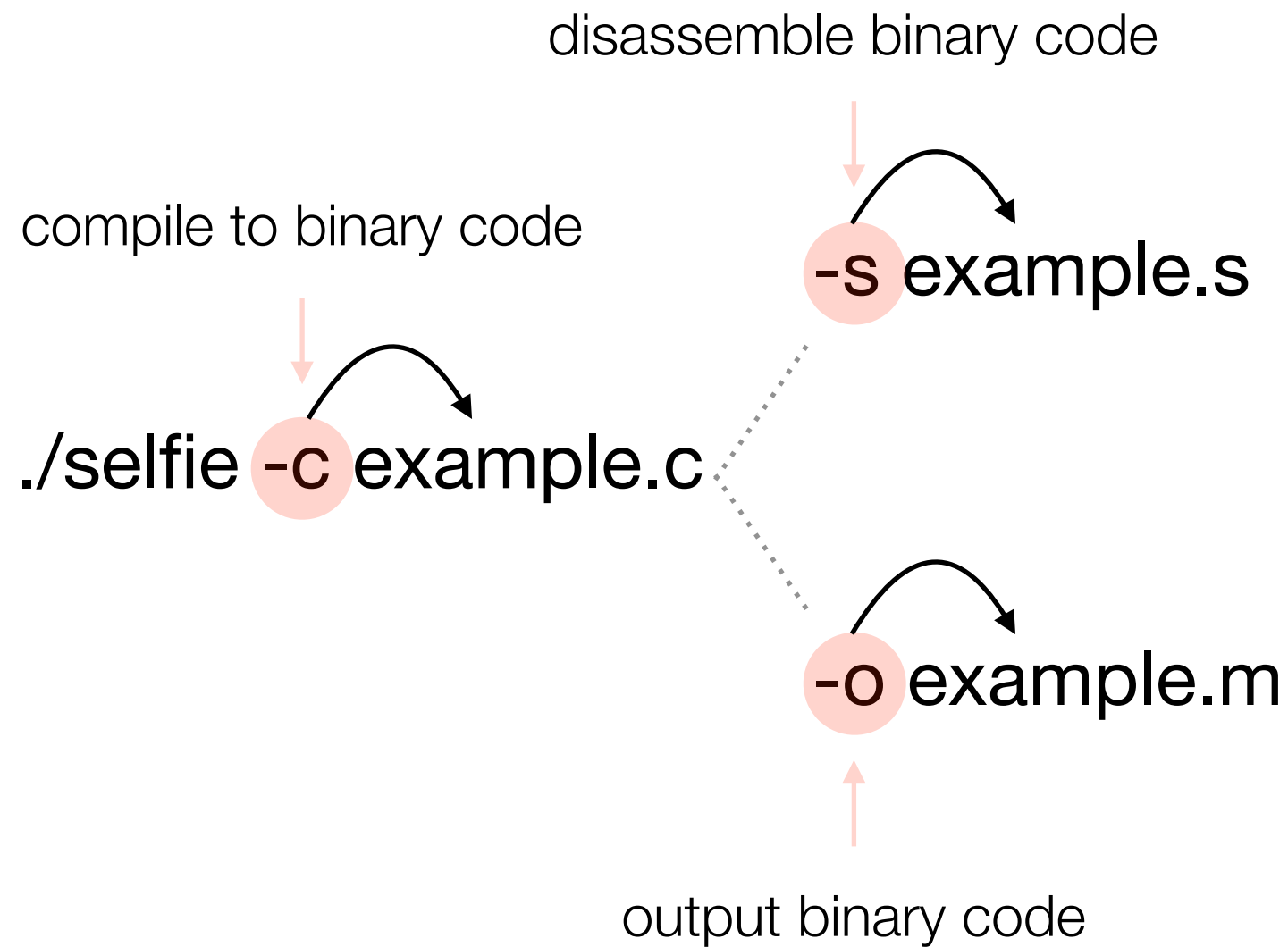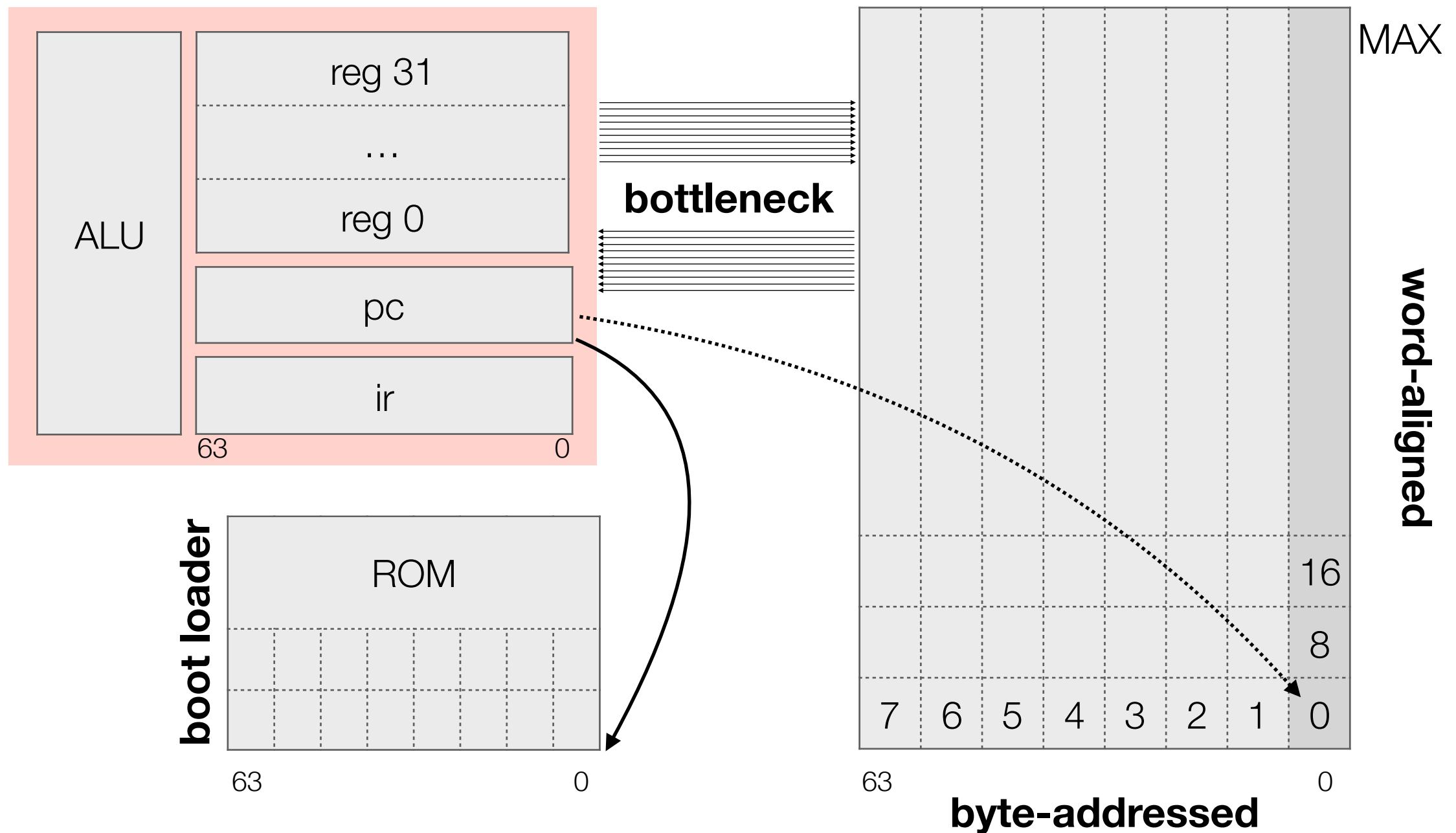
More about the Von Neumann architecture

Von Neumann Machine
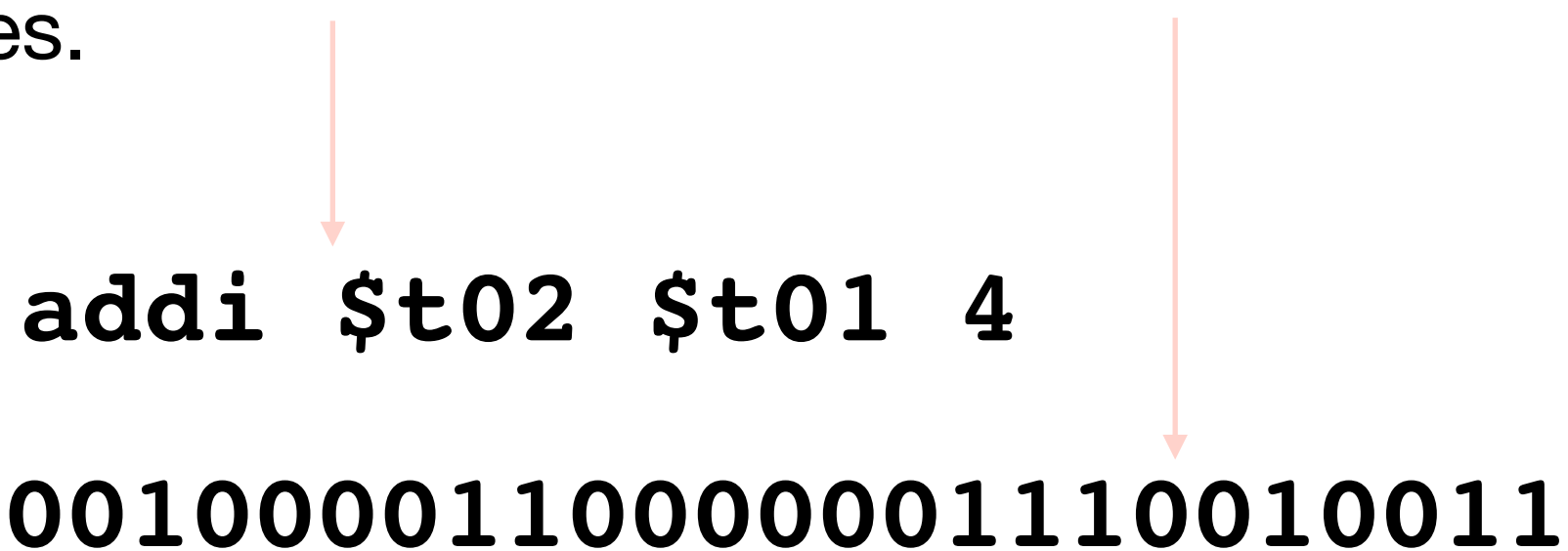
- **Key idea** - code and data in same memory

- How do we know what code is and what data is?
  - The **program counter** points to an instruction in memory making it code. This instruction may instruct the processor to load bits from memory and modify them making these bits data.

- **Bootstrapping** - Loading the first program

  - At the very beginning hardware support is needed to set the PC to address 0 of the memory where the boot-loader code is stored.

  - The boot-loader is code stored in non-volatile memory (ROM) that instructs the processor to load code.

  - Last instruction sets the PC to address 0 of main memory.

Introduction to RISC-U

- RISC-U is the RISC-V subset targeted, emulated, and virtualized by selfie.

- There are 14 instructions, each 32-bit wide, the processor knows and the compiler generates code for.

- When talking about formal languages it is important to distinguish between the syntax and the semantics of that language.

Syntax of RISC-U

- At machine code level we distinguish further between the human readable <u>assembly</u> code and the <u>binary</u> code the CPU executes.

```
addi $t02 $t01 4
```

```
00000000010000110000001110010011
```

Syntax of RISC-U

- Different instructions have different binary encodings. Instructions are encoded using special formats.

- The format of an instruction specifies how the 32 bits are interpreted. It is designed in a way that allows fast decoding of the instructions.

- Selfie always stores binary code which can then be <u>disassembled</u> to obtain the assembly code.

Semantics of RISC-U

- The semantics of an instruction is determined by how the processor implements it.

- In selfie this implementation is done in the `do_..` 's, like `do_addi()` or `do_sltu()`.

Decoding Instructions

- Each instruction can be uniquely identified by certain parts of the 32 bits called **opcode**, **funct3** and **funct7** code.

- When the instruction is known, the meaning of the remaining bits of the instruction becomes clear.

Level 1

opcode

looking further

Level 2

funct3

looking further

Level 3

funct7

The opcode specifies the format in which the instruction is encoded.

The opcode might already uniquely identify the instruction that has to be decoded. If not, the funct3 code is looked at next.

The opcode in combination with the funct3 code might already identify the instruction. Not part of all instruction formats.

The instruction is definitely known at this point. Not part of all instruction formats.

Instructions and Machine State

- Selfie uses special procedures (`_before()` & `_after()`) that show on which part of the machine state they depend, which part they modify and the modification itself.

rs1	**|-** rd	**->** rd
read only dependency in control flow	will be modified	new value after modification
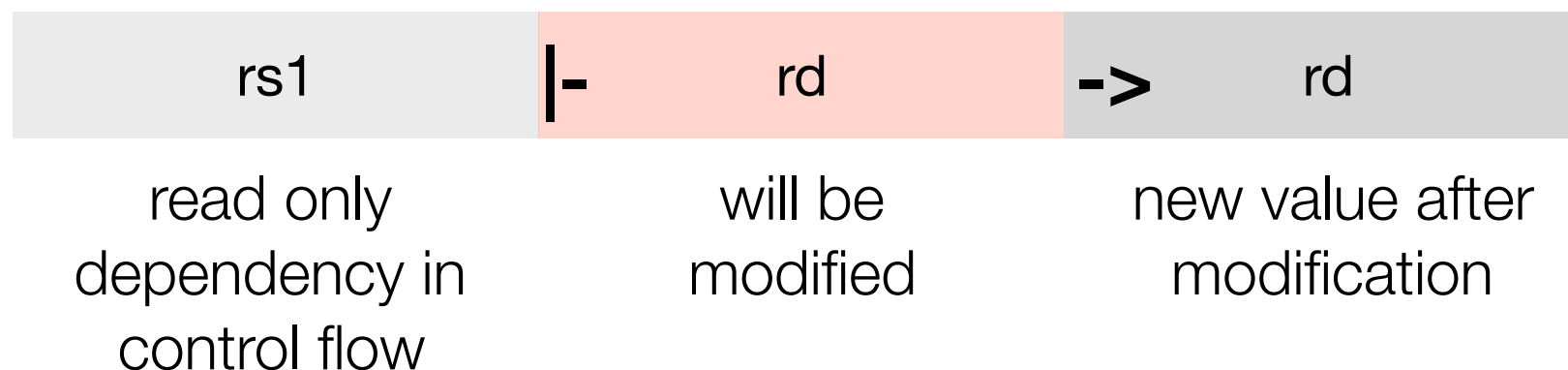
- This information is enough to determine the machine state at any point of execution (completely deterministic).

- The only way to inject information from outside that is not known beforehand is through the read call.

Instructions and Formats

RISC-U

lui	beq
addi	jal
add	jalr
sub	ld
mul	sd
divu	sltu
remu	ecall

special case of `addi`

nop

Formats

R-format
I-format
S-format
B-format
J-format
U-format

Immediate Arithmetic Instructions

lui	$rd	imm	
addi	$rd	$rs1	imm

- **Load upper immediate** loads `imm` value shifted by 12 bits into `$rd` and **add immediate** adds `imm` to the content of $rs1 and stores the result in `$rd`.

- These two instructions are used to **initialize registers** (`$rs1 = $zero`) and to **load addresses** into a register in order to read values from memory.

- `lui` is used to load the upper and `addi` to load the lower bits - see `load_integer(uint64_t value)`.

- A special case of `addi` is `nop`, with `$zero = $zero + 0`.
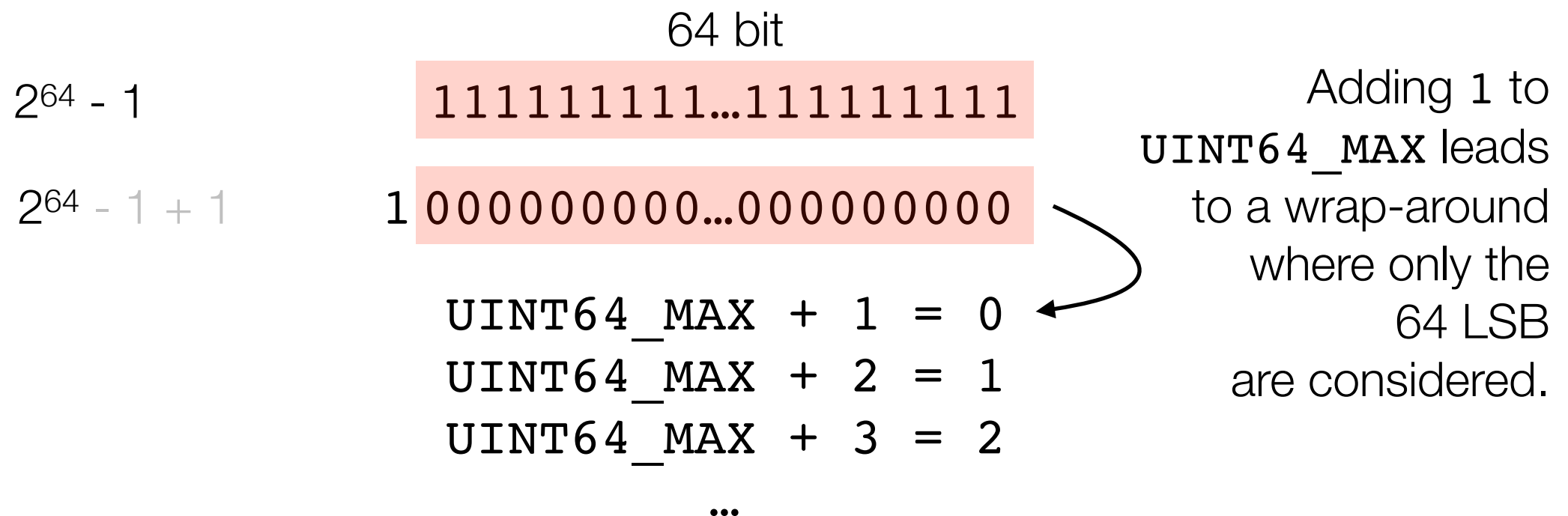
Arithmetic Instructions

add	$rd	$rs1	$rs2
sub	$rd	$rs1	$rs2
mul	$rd	$rs1	$rs2
Dive	$rd	$rs1	$rs2
remu	$rd	$rs1	$rs2

$rd = $rs1 +,-,*,/,% $rs2

- The processor executes these instructions using <u>unsigned integer arithmetic</u> with <u>wrap-around semantics</u>.

Wrap-around Semantics

- Cause of unbelievably expensive bugs, e.g. the Ariane 5 Flight 501.

- 2^{64} - 1 is the largest value that can be represented by 64 bits. In selfie this value is denoted `UINT64_MAX`.
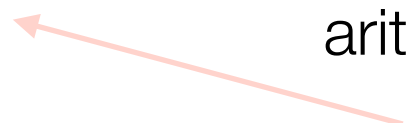
64 bit

2^{64} - 1   111111111...111111111

2^{64} - 1 + 1   1 000000000...000000000

Adding 1 to `UINT64_MAX` leads to a wrap-around where only the 64 LSB are considered.

```
UINT64_MAX + 1 = 0
UINT64_MAX + 2 = 1
UINT64_MAX + 3 = 2
```
...

Arithmetic Instructions

sltu	$rd	$rs1	$rs2

- Set `$rd` to `1` if `$rs1 < $rs2`.

- This is the only instruction needed to implement <, >, <=, >=, == and `!=`.

- How this is done:

  - == is implemented using `b - a < 1`.

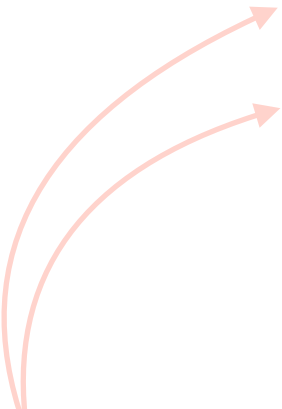In unsigned arithmetic only 0 satisfies this condition.

Memory Instructions

ld	$rd	offset($rs1)
sd	$rs2	offset($rs1)

- **Load** into `$rd` the value that is stored at the address that is obtained by adding the immediate offset to the content of `$rs1`.

- **Store** the content of `$rs2` at the address that is obtained by adding the immediate offset to the content of `$rs1`.

- The <u>addressing mode</u> used for these instructions is called register-relative addressing.

Control-Flow Instructions

- <u>Control flow</u>, at machine code level, is the order in which instructions are executed.

- All previous instructions feature implicit **trivial control flow**, that is, they simply set the program counter to the next instruction. Their main purpose is **data manipulation.**

- The following instructions have a more sophisticated effect on control flow.

Control-Flow Instructions

beq	$rs1	$rs2	imm
jal	$rd	imm	
jalr	$rd	$rs1	imm

- The first two instructions use a different addressing mode called pc-relative addressing at the resolution of 12 bit.

- **Branch on equal** sets the `pc` to `pc + imm` if the content of `$rs1` matches `$rs2`.

- **Jump and link** is used for procedure calls and stores the return address (address of next instruction) in `$rd`.

- **Jump and link register** is similar to `jal`, except that it uses register-relative addressing.

Link Register

```
uint64_t increment(uint64_t inc) {
    return inc + 1;
}

uint64_t main() {
    uint64_t a;

    a = 0
    a = increment(a);

    return a;
}
```

The next **instruction** that is linked is the assignment into a!

Operating System Support

```
ecall
```

- This is not a standard machine instruction but rather a **programmable instruction** that allows you to request services from the <u>operating system</u>, also known as <u>system call</u>.

- The arguments that specify what action we need the OS to take are provided in `$a7`.

- Selfie supports <u>these</u> ecalls: `read`, `write`, `open`, `malloc`, and `exit`.

- The implementation of each of these ecalls is in <u>handleSystemCall()</u>.

Introducing Language Operators

- In selfie there are no <u>bitwise operators</u> and no native machine instructions for bitwise operations.

- The next section shows how new language operators can be **systematically** implemented.

Bitwise Operators

- Bitwise operators in C perform operations on bit level. There are:

  - logic bitwise operators: & (bitwise AND), | (bitwise OR) and ˜ (bitwise NOT).

  - shift bitwise operators: << (bitwise left shift) and >> (bitwise right shift).

- Logical shifts (zeros are shifted in) or arithmetic shifts (sign bit is shifted in).

…in selfie

- The semantics of the bitwise operators << and >> on the `uint64_t` type is that of a *logical shift*.

- In selfie there are library functions that perform shift operations (see <u>here</u>):

    - `leftshift()`: n << b = n * 2^b

    - `rightshift()`: n >> b = n / 2^b

Language Operators

- The compiler has to recognize the operator and generate code for it.

- The processor has to understand the instructions encoded by the compiler and executes them.

- We first expand the processor by implementing a new machine instruction that can then be used by the compiler to generate code.

gets called by the **compiler**

encode

need to match

decode

gets called by the **processor**
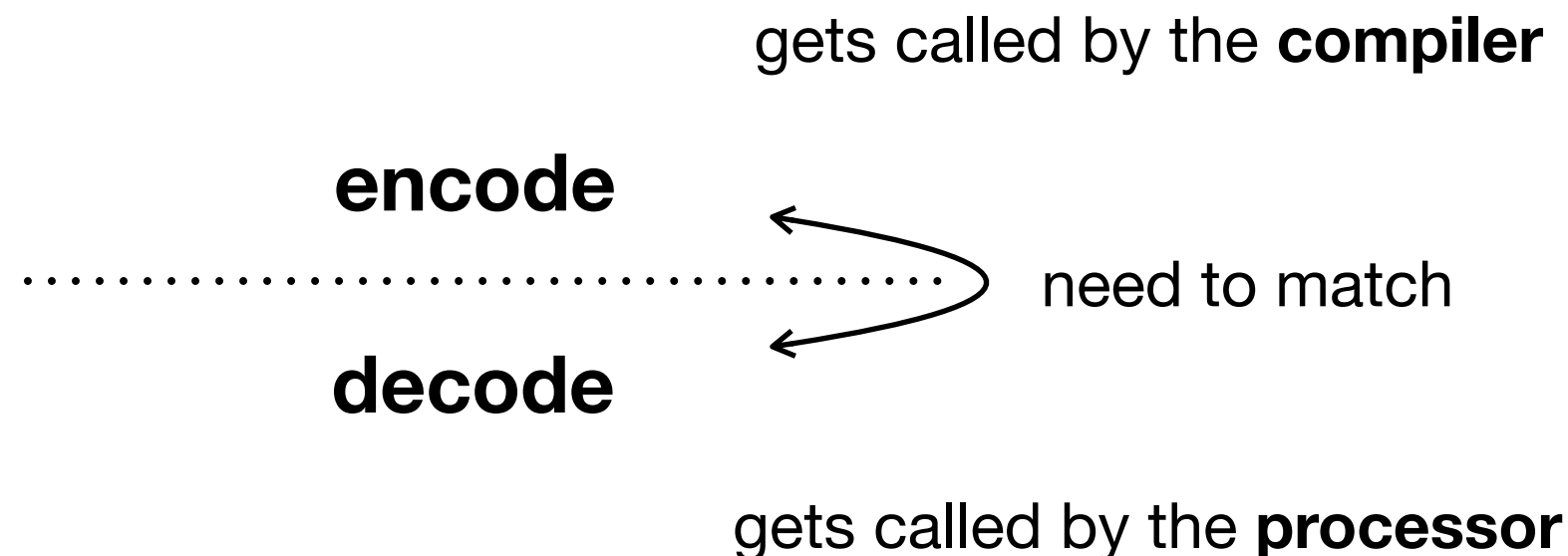
Implementing Machine Instructions

- Extending the Processor - Implementing machine instructions for `SLL` and `SRL`.

- Before implementing an instruction the **semantics** it is supposed to have must be clear.

- The implementation of an instruction determines its semantics. Truly understanding each instruction is key to understanding the target language generated by the compiler and executed by the processor.

The Processor

- The heart of the selfie emulator is the procedure <u>runUntilException()</u>

<u>fetch</u>

<u>decode</u>

<u>execute</u>
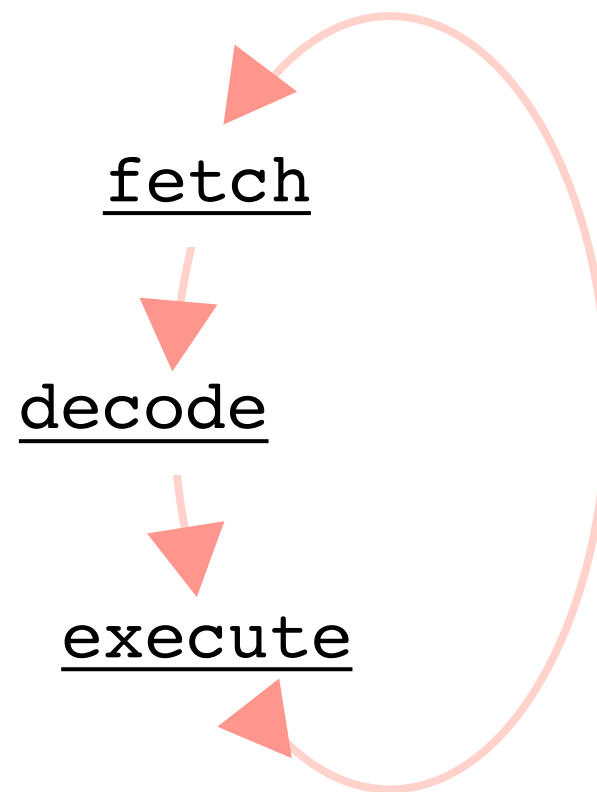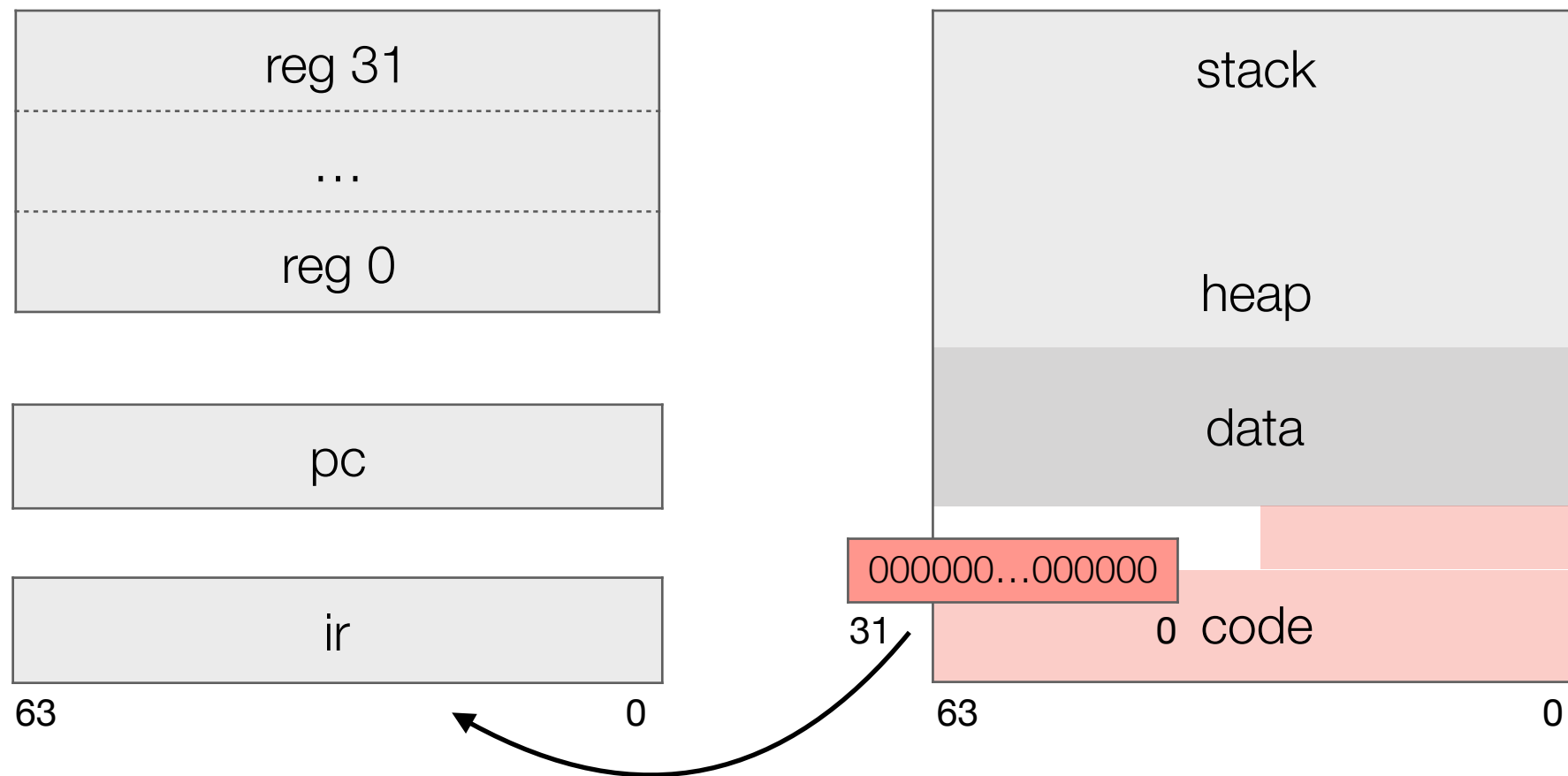
Fetch

- A 32-bit instruction gets fetched from memory and is then stored in the instruction register (`ir`)

- 2 instructions stored in memory using hi/low word

Decode-Remember

- The opcode specifies the instruction format in which the instruction is encoded.

- After decoding the binary code of the instruction, the processor knows what instruction and parameters it is dealing with.

- Example **ADDI**:

    - Immediate instructions contain bits that are interpreted as numerical value that is retrieved using shifting operations.

    - 5 bit are enough to encode 32 registers.

immediate	rs1	funct3	rd	opcode
12 bit	5 bit	3 bit	5 bit	7 bit

Execute

- The execution of every RISC-U instruction has a well-defined effect. It changes the state of the machine only at a specific location involving little data.

- At most two registers or one register and one memory location are modified by an instruction.

  - Every instruction modifies the `PC`.

  - Most instructions which modify **data** (another register or memory location) have trivial control flow (`PC` to next instruction).

  - Control-flow instructions have a more sophisticated **control flow**, that is, they may change the `PC` using relative or absolute addressing.

Execute

- Example **ADDI**:

  semantics: 64-bit unsigned addition with wrap-around semantics

  - Bits in `$rd` are overwritten with `$rs1 + imm`.

  - `PC = PC + INSTRUCTIONSIZE.`

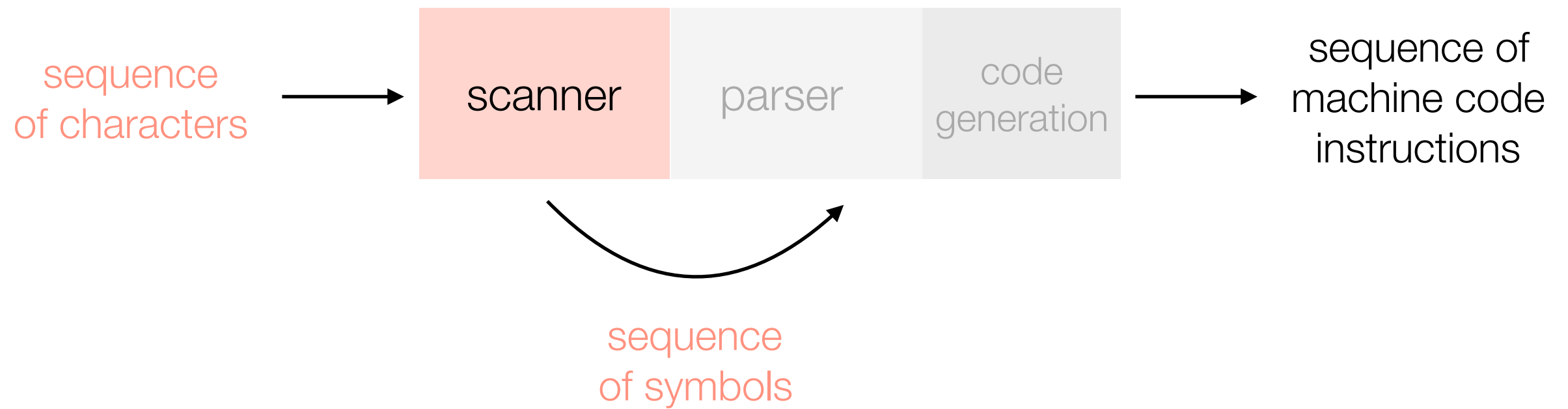 - Used for initialization - loading constants into registers.

Realize…

- This is how virtually every general-purpose processor operates.

- Instructions only have a tiny effect on the overall machine state.

- What makes modern computation so powerful is:

  - the incredible **speed**.

  - the enormous size of **memory**.

The Compiler

sequence of characters → **compiler**

scanner	parser	code generation

→ sequence of machine code instructions

The Scanner

sequence of characters →

scanner | parser | code generation

→ sequence of machine code instructions

sequence of symbols

The Scanner

*The scanner performs a **membership test** on a sequence of characters. It checks whether the symbols it sees are **valid symbols** of the programming language.*
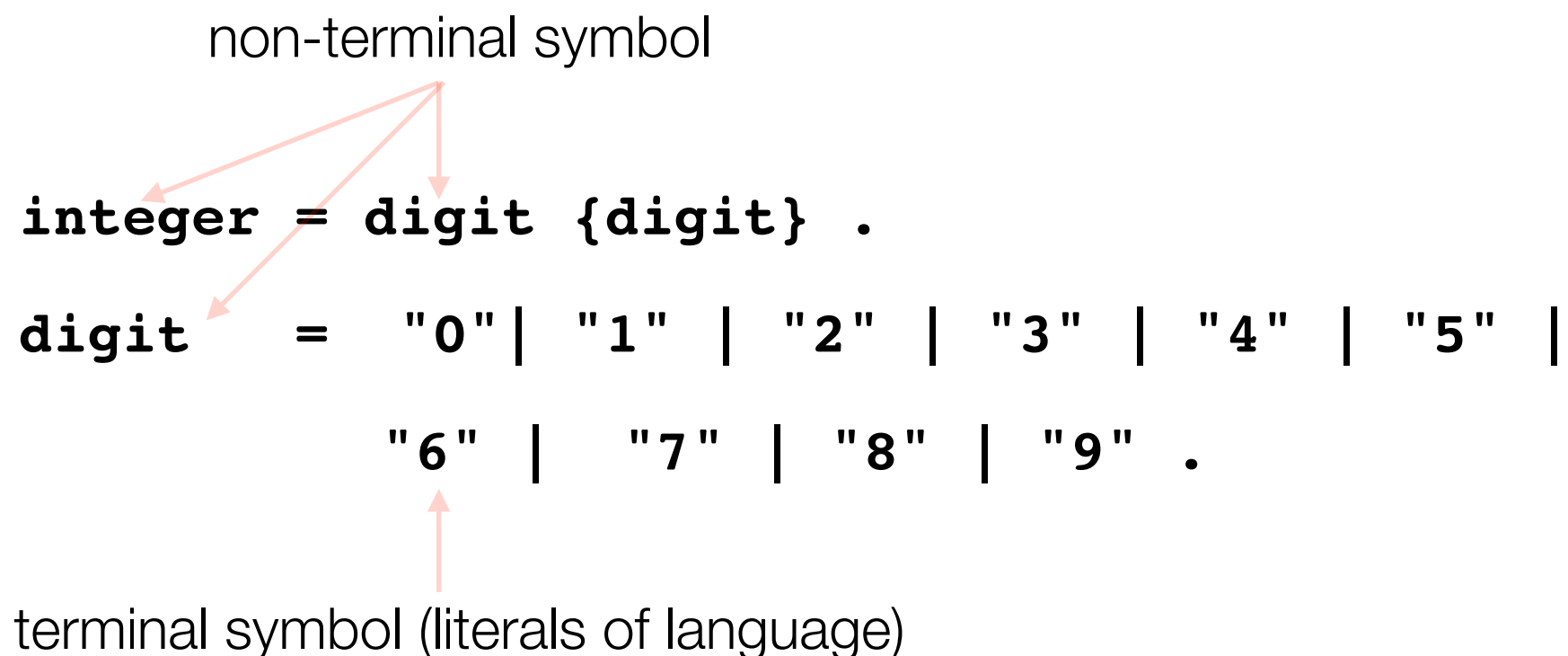
- Reading only **one character** at a time the scanner transforms a sequence of characters into a sequence of symbols that has **no structure**.

- There are single-character symbols (`'<'`, `x`) and multi-character symbols (`'<='`,`'variable'`). All symbols are uniquely identified by a token (integer).

- The whole sequence of symbols is not stored. Only the last read symbol is remembered.

What are valid symbols?

- **What is needed:**
  A specification of valid symbols -> <u>regular expression</u> and a formalism to write such a specification.

- **In selfie…**
  the chosen formalism is <u>EBNF</u> (see in the <u>grammar.md</u> file).

Regular Expression

- Regular expression is a formalism that defines a <u>regular language</u> (= a set of symbols defined by regular expression)

- The production rules consists of <u>terminal symbols, non-terminal symbols</u> and operators (repetition{}, concatination ⌴, xor│ , …).

- A regular expression can be reduced to **a single rule** by replacing every non-terminal symbol with its right-hand side until no non-terminal symbols are left.

non-terminal symbol

```
integer = digit {digit} .
digit   =  "0"| "1" | "2" | "3" | "4" | "5" |
           "6" |  "7" | "8" | "9" .
```

terminal symbol (literals of language)

EBNF<u>-Niklaus Wirth</u>

```
ebnf = { production } .

production = identifier "=" expression "." .

expression = term { "|" term } .

term = factor { factor } .

factor = identifier | string | "(" expression ")" |
         "[" expression "]" | "{" expression "}" .

string = """ printableCharacter { printableCharacter }""" .

printableCharacters = "a" | … | "!" .

identifier =  "ebnf" | "production" | … | "identifier .
```

The Scanner

- **The problem statement:**
  Given a specification of valid symbols where symbols may consist of multiple characters…

  …Is a given input (source code) in this specification, or in other words do the characters in the source file form valid symbols according to the specification?

Abstraction

- A scanner for the English language accepts this sentence.

- A sequence of characters gets assembled symbols (words and punctuation).

This␣sense␣makes␣!␣no␣sentence

Specification and Implementation

- **Specification**
  The specification of valid symbols is a regular expression written in EBNF. It is easier to reason about the correctness of regular expression and it is expressive enough. The number of valid symbols is infinite and their size is unbounded.

- **Implementation**
  The scanner implements this specification using a <u>finite state machine</u> (FSM), a computational model, that recognizes/accepts the set of regular expressions. The implementation is simple and can be done very efficiently.

FSM
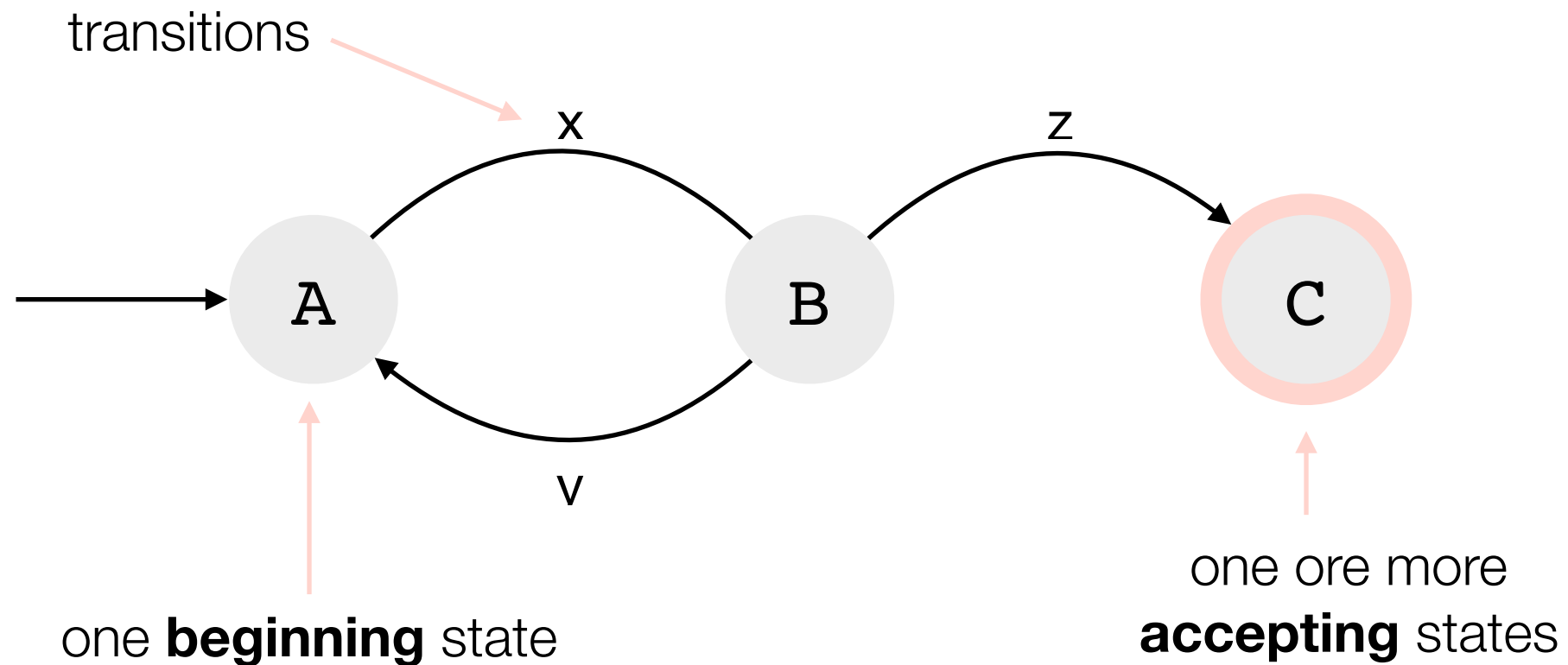
- Transitions describe how to get from one state to another.

- E.x. When in state B the condition z (may be "reading z") is true, a move to state C happens.

The Scanner

getSymbol()

calls

getCharacter()

findNextCharacter()
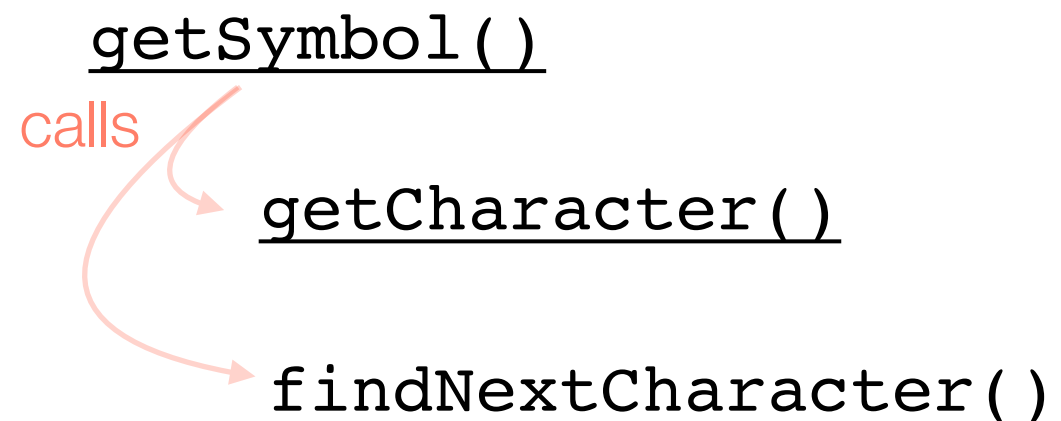
- The heart of the scanner is the procedure `getSymbol()`, which finds the next valid symbol in a sequence of characters and stores it in a global variable (`symbol`).

- It internally uses two procedures `getCharacter()` and `findNextCharacter()`.

- `getCharacter()` simply reads the next character of the input stream and stores it into a global variable (`character`).

- `findNextCharacter()` is the implementation of whitespace.

Sequence of Characters

- The procedure `getCharacter()` is used to read the next byte into a buffer.

- The return value is either the number of read bytes or 0, which indicates end of file (EOF).

- Any input file is nothing more than a **sequence of characters** to the scanner. It will even accept meaningless sequences as long as it consists of valid symbols in the specified language.

Whitespace

- Whitespace and comments do not change the semantics of code and could even be omitted (minification). Their purpose is to make the code more readable.

- The scanner implements whitespace and comments by simply ignoring them using the procedure `findNextCharacter()`.

The Scanner - getSymbol()
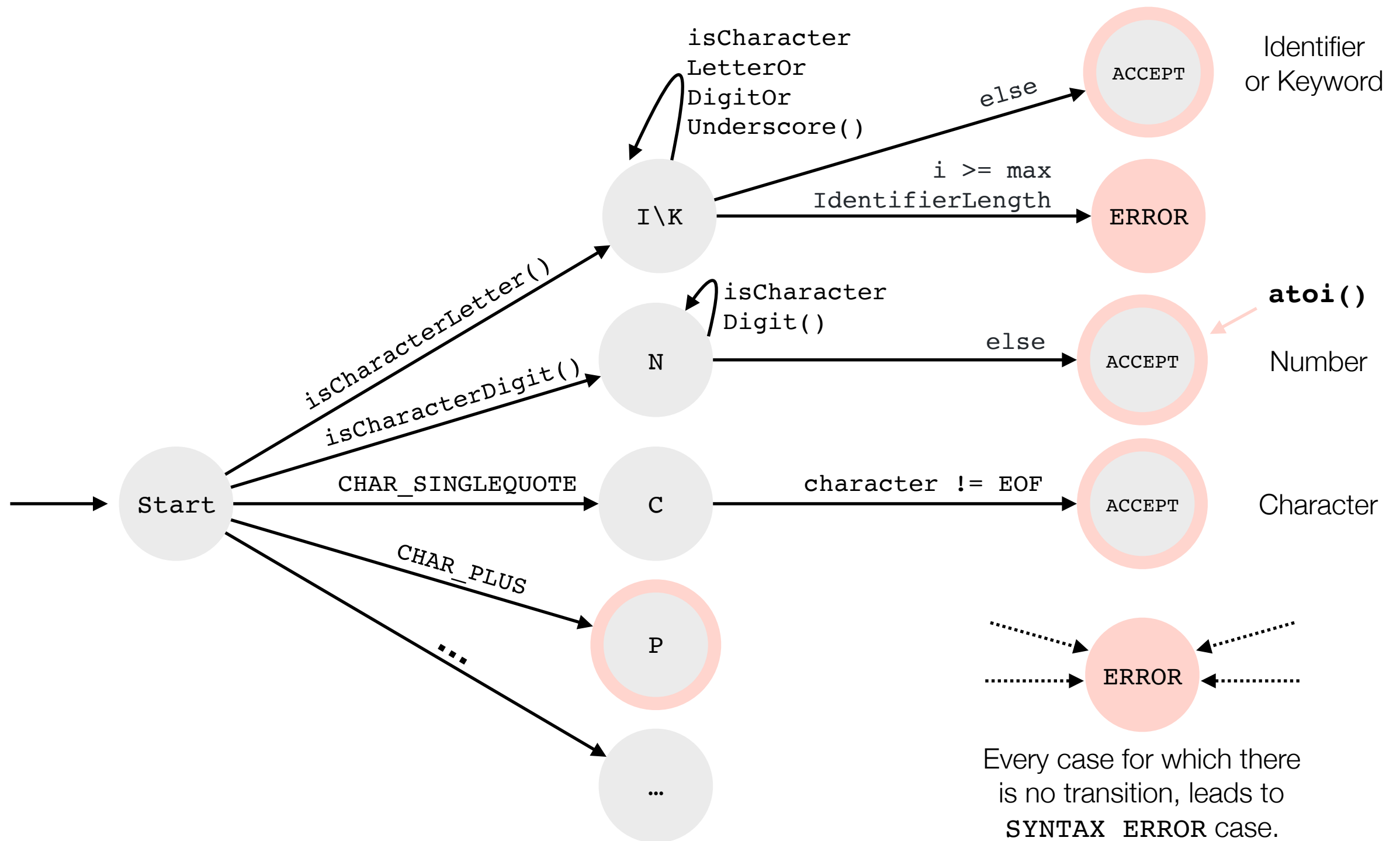
Properties

- **The scanner never crashes and always terminates:**
  In code the FSM is simply a huge `if/else if/else` statement that covers every possible case. Syntax errors fall through to the `else` case.

- **The scanner is good at forgetting:**
  As soon as the scanner reaches a state it has been in before everything in between is forgotten.

- **Correctness:**
  The state space of a machine is usually huge or even infinite and therefore reasoning about it is hard. An FSM however has only very few states (finitely many) and reasoning about them is much easier.

Properties

- **The scanner can not recognize structure:**
  It just operates on a sequence of characters and turns it into a sequence of symbols.
  To recognize structure, the scanner would need to be able to **count and remember** (e.g. the number of braces).

  Therefore we need something more powerful called a parser.