# Programming Language Concepts

## Abstract Machines

Adrien Champion
adrien.champion@email.com

# Contents

1. Languages, programs and machines
2. Organic example: mictyris guinotae
3. Hardware example
4. Abstract machine: implementation

# Notions

*programming language*:

a formalism with some "instructions"

*program* in language $\mathcal{L}$:

a finite sequence of instructions in $\mathcal{L}$

# Notions

*programming language*:

a formalism with some “instructions”

*program* in language $\mathcal{L}$:

a finite sequence of instructions in $\mathcal{L}$

*abstract machine* for $\mathcal{L}$:

any set of data structures and algorithms which can perform the storage and execution of programs in $\mathcal{L}$

# Abstract machines

Abstract machines come in many flavors, but they share some traits.

They can

- process primitive data:

  integers, floats, . . .    addition, multiplication, . . .

- control the sequence of execution of operations:

  next instruction, *jump*, . . .

- control data transfers:

  from memory to registers, *addressing modes*, . . .

- manage memory:

  allocation, *garbage collection*, *stack*, . . .

# Abstract machines

Abstract machines come in many flavors, but they share some traits.

They can

- process primitive data:

  integers, floats, . . .  addition, multiplication, . . .

- control the sequence of execution of operations:

  next instruction, *jump*, . . .

- control data transfers:

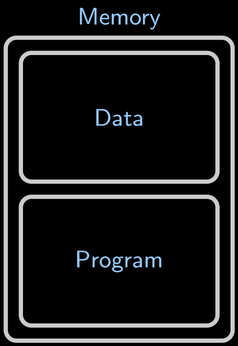
  from memory to registers, *addressing modes*, . . .

- manage memory:

  allocation, *garbage collection*, *stack*, . . .

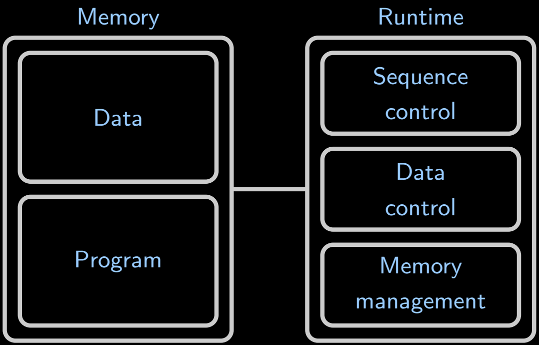
Also, they can run

# Runtime: execution cycle

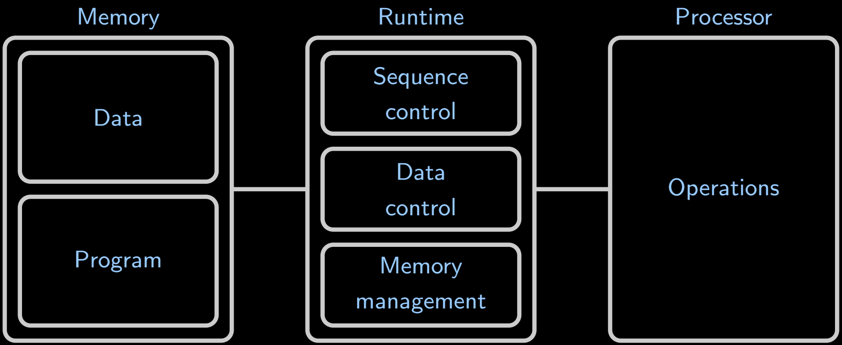


What is the difference between program and data?

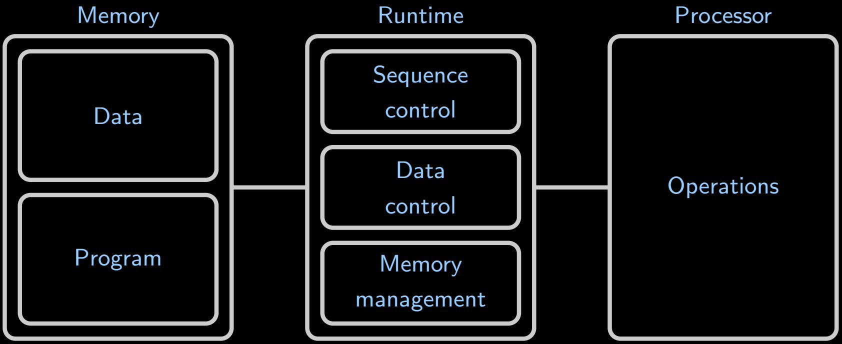
# Runtime: execution cycle



# Runtime: execution cycle



# Runtime: execution cycle



Memory

Data

Program

Runtime

Sequence control

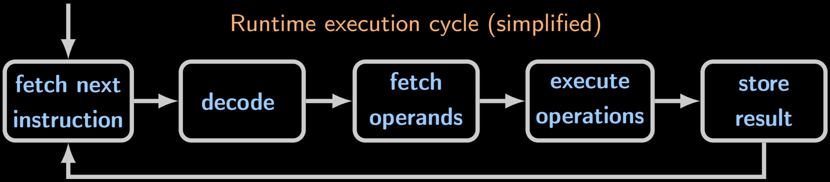
Data control

Memory management

Processor

Operations

Runtime execution cycle (simplified)



**fetch next instruction**

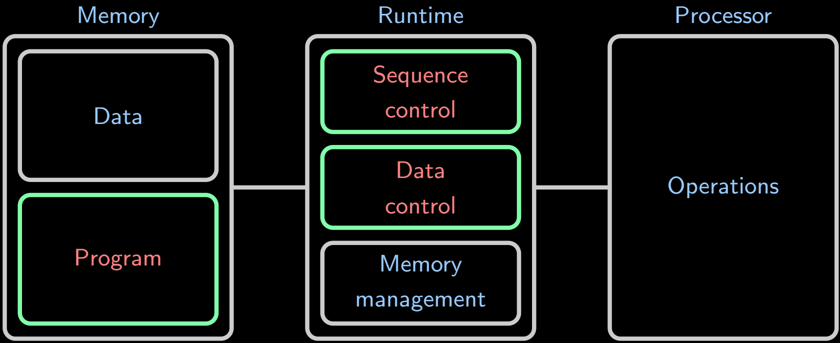
**decode**

**fetch operands**

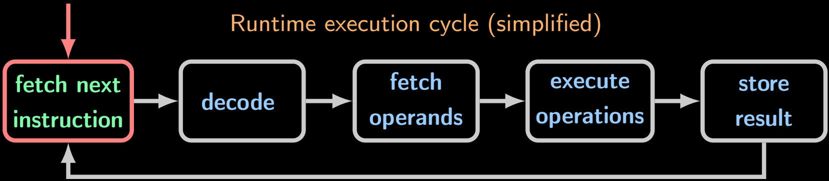
**execute operations**

**store result**

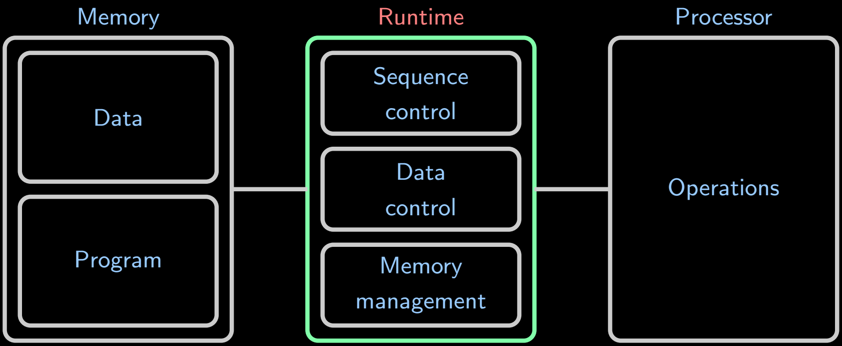
# Runtime: execution cycle



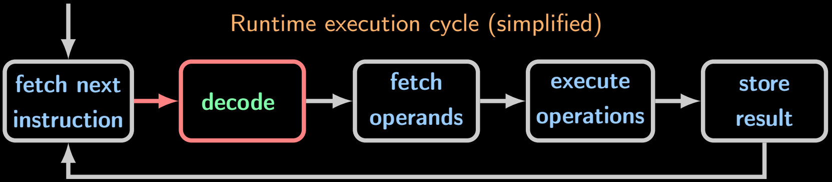
Runtime execution cycle (simplified)



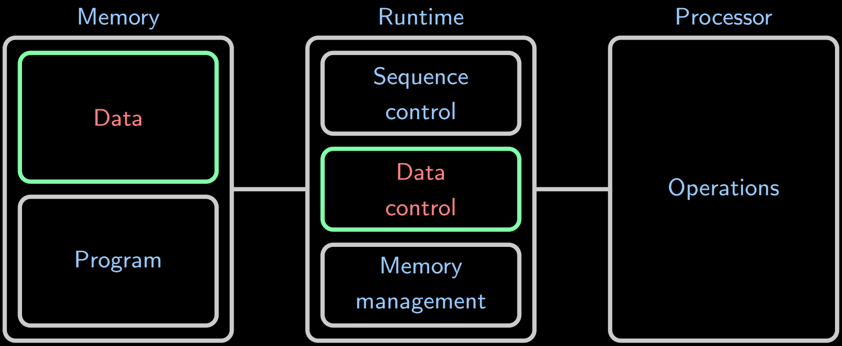
# Runtime: execution cycle

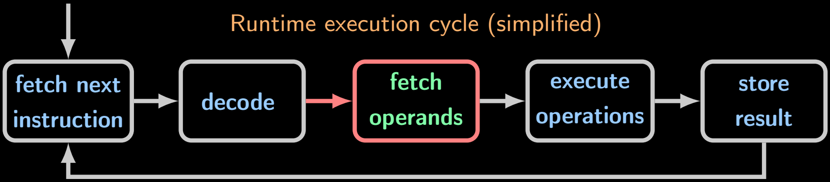


Runtime execution cycle (simplified)

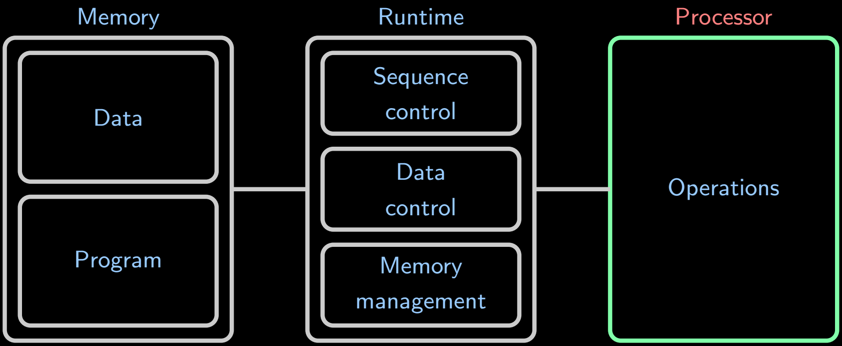


# Runtime: execution cycle




Runtime execution cycle (simplified)

# Runtime: execution cycle



Memory

Data

Program

Runtime

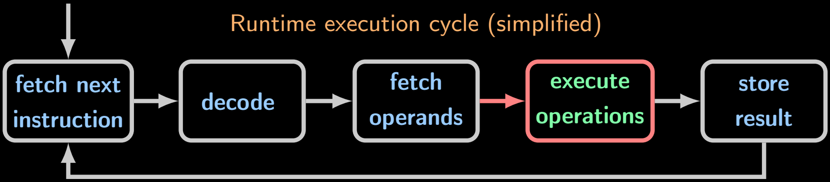
Sequence control

Data control

Memory management

Processor

Operations



Runtime execution cycle (simplified)

fetch next instruction

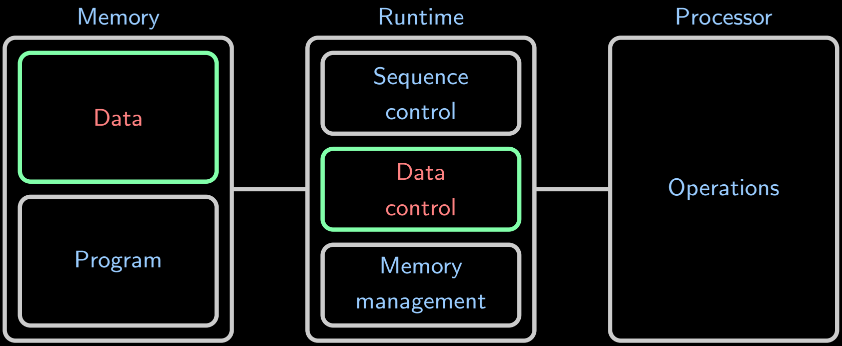
decode

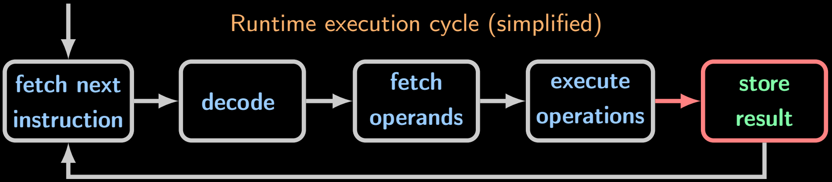
fetch operands

execute operations

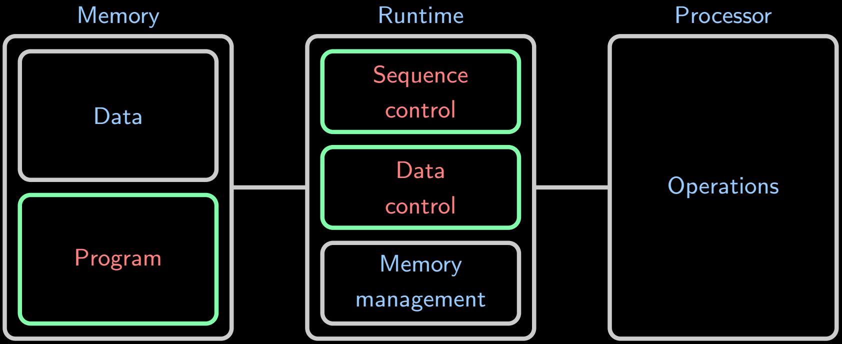
store result

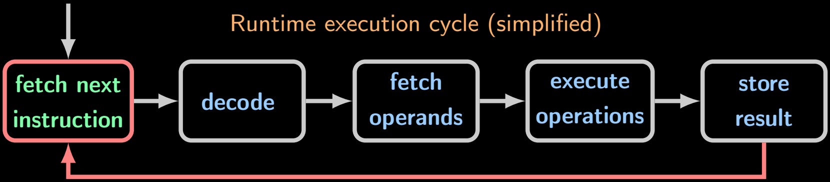
# Runtime: execution cycle





# Runtime: execution cycle




Runtime execution cycle (simplified)

# Runtime

As we saw, the runtime

- decides which instruction to execute next,
- fetches the instruction,
- decodes it in terms of primitive operations,
- fetches the operands,
- executes the primitive operations,
- stores the result.

# Runtime

As we saw, the runtime

- decides which instruction to execute next,
- fetches the instruction,
- decodes it in terms of primitive operations,
- fetches the operands,
- executes the primitive operations,
- stores the result.

What about *memory management*?

# Memory management

It can

- *allocate* / *free* memory,
- handle the *heap* / *stack* (if any),
- do garbage collection,
- *suspend* the execution of a program.

# Memory management

It can

- *allocate* / *free* memory,
- handle the *heap* / *stack* (if any),
- do garbage collection,
- *suspend* the execution of a program.

Ranges from simple to very complex depending on the abstract machine.

# Contents

1. Languages, programs and machines
2. Organic example: mictyris guinotae
3. Hardware example
4. Abstract machine: implementation

# Soldier crabs

Organic example: mictyris guinotae

Soldier crabs crossing path have a deterministic behavior

We can build circuits as lanes:

- $\top$ — at least a crab
- $\bot$ — no crab

http://www.gizmag.com/crab-computer-kobe/22145/

http://arxiv.org/pdf/1204.1749v1.pdf

# Soldier crabs

Organic example: mictyris guinotae

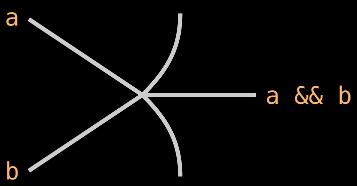
Soldier crabs crossing path have a deterministic behavior

We can build circuits as lanes:

- $\top$ — at least a crab
- $\bot$ — no crab

http://www.gizmag.com/crab-computer-kobe/22145/

http://arxiv.org/pdf/1204.1749v1.pdf



# Soldier crabs

Organic example: mictyris guinotae

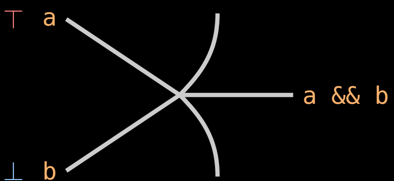
Soldier crabs crossing path have a deterministic behavior

We can build circuits as lanes:

- $\top$ — at least a crab
- $\bot$ — no crab

http://www.gizmag.com/crab-computer-kobe/22145/

http://arxiv.org/pdf/1204.1749v1.pdf



# Soldier crabs

Organic example: mictyris guinotae

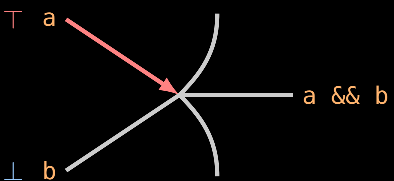
Soldier crabs crossing path have a deterministic behavior

We can build circuits as lanes:

- $\top$ — at least a crab
- $\bot$ — no crab

http://www.gizmag.com/crab-computer-kobe/22145/

http://arxiv.org/pdf/1204.1749v1.pdf



# Soldier crabs

Organic example: mictyris guinotae

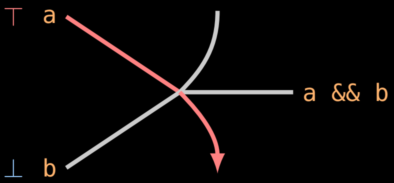
Soldier crabs crossing path have a deterministic behavior

We can build circuits as lanes:

- $\top$ — at least a crab
- $\perp$ — no crab

http://www.gizmag.com/crab-computer-kobe/22145/

http://arxiv.org/pdf/1204.1749v1.pdf



# Soldier crabs

Organic example: mictyris guinotae

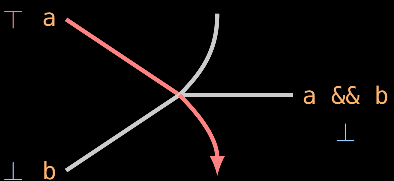
Soldier crabs crossing path have a deterministic behavior

We can build circuits as lanes:

- ⊤ — at least a crab
- ⊥ — no crab

http://www.gizmag.com/crab-computer-kobe/22145/

http://arxiv.org/pdf/1204.1749v1.pdf



# Soldier crabs

Organic example: mictyris guinotae

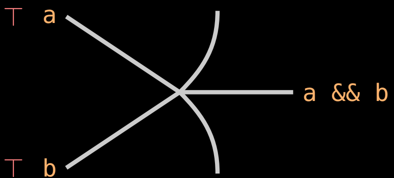
Soldier crabs crossing path have a deterministic behavior

We can build circuits as lanes:

- $\top$ — at least a crab
- $\bot$ — no crab

http://www.gizmag.com/crab-computer-kobe/22145/

http://arxiv.org/pdf/1204.1749v1.pdf



# Soldier crabs

Organic example: mictyris guinotae

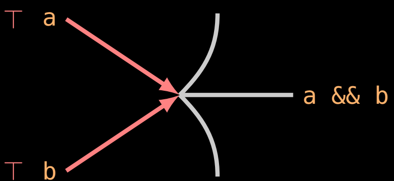
Soldier crabs crossing path have a deterministic behavior

We can build circuits as lanes:

- $\top$ — at least a crab
- $\bot$ — no crab

http://www.gizmag.com/crab-computer-kobe/22145/

http://arxiv.org/pdf/1204.1749v1.pdf



# Soldier crabs

Organic example: mictyris guinotae

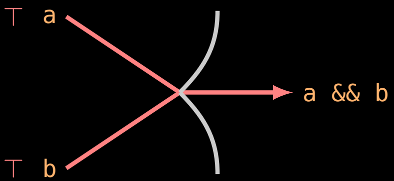
Soldier crabs crossing path have a deterministic behavior

We can build circuits as lanes:

- $\top$ — at least a crab
- $\bot$ — no crab

http://www.gizmag.com/crab-computer-kobe/22145/

http://arxiv.org/pdf/1204.1749v1.pdf



# Soldier crabs

Organic example: mictyris guinotae

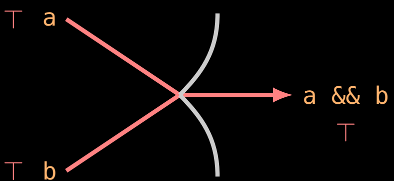
Soldier crabs crossing path have a deterministic behavior

We can build circuits as lanes:

- $\top$ — at least a crab
- $\bot$ — no crab

http://www.gizmag.com/crab-computer-kobe/22145/

http://arxiv.org/pdf/1204.1749v1.pdf



# Soldier crabs (runtime)

Sequence control / data transfer: humans put crabs in lanes

Processing primitive data: only `bool` is supported

- `true`: a least a crab in the lane
- `false`: no crab in the lane
- processor: lanes (circuit) powered by crab legs and brain

# Soldier crabs (runtime)

Organic example: mictyris guinotae

Sequence control / data transfer: humans put crabs in lanes

Processing primitive data: only `bool` is supported

- `true`: a least a crab in the lane
- `false`: no crab in the lane
- processor: lanes (circuit) powered by crab legs and brain

Although it is silly, can we (at least in theory)

- do arithmetic? How?
- implement sequence control / data transfer?

# Contents

1. Languages, programs and machines
2. Organic example: mictyris guinotae
3. Hardware example
4. Abstract machine: implementation

# Memory

Memory:

- composed of *RAM*, *caches* (L1, L2, L3), ...
- stores *words* of 32 / 64 bits
- recognizes primitive types (*a.k.a.* predefined types):
  booleans, integers, floats, characters, fixed-length sequences, ...

# Language

Composed of simple instructions:

```
OpCode Operand1 Operand2
```

For instance:

- add the contents of registers `R0` and `R5`, store result in `R5`:

  ```
  ADD R5 R0
  ```

- add the contents of the memory cells whose addresses are stored in registers `R0` and `R5`, store result in the cell `R5` points to:

  ```
  ADD (R5) (R0)
  ```

# Runtime

Sequence control: *Program Counter (PC)* (special register)

- contains the address of the next instruction to execute
- supports operations like *increment*, *jump*, . . .

Processing primitive data: *Arithmetic and Logic Unit (ALU)*

- arithmetic (`int`, `float`) and logical (`bool`) operations

Data transfer:

- special registers (MAR / MDR) bridge the memory / CPU gap
- handles different *addressing modes*
- operations to load data in the CPU's registers are provided

# Contents

1. Languages, programs and machines
2. Organic example: mictyris guinotae
3. Hardware example
4. Abstract machine: implementation

# Trade-off

Implementing a machine for a language is a trade-off between

- performance
- flexibility

  when the language evolves
- portability

  diffusion of *executables*

Eventually, code will run on a *physical* machine, the hardware.

## Trade-off

Implementing a machine for a language is a trade-off between

- performance
- flexibility — when the language evolves
- portability — diffusion of *executables*

Eventually, code will run on a *physical* machine, the hardware.

Hardware-level is the reference performance-wise:
can't go faster than hardware by definition

# Hardware

- very fast
- need to build new machines when language changes
- *in general*, can't share executable with different machines

By definition, CPU instructions are implemented in hardware

# Software

- slow *compared to hardware* (why?)
- easy to propagate changes in the language
- can share with different machines (if runtime installed)

Pretty much all programming languages, *to various degrees*

# Firmware

Compromise between hardware and software

- *microcode*, *a.k.a.* *microprogramming*
- happens at *firmware* level:
  - microcode/firmware is stored in fast, special memory
  - requires special equipment to write
- level directly above the circuitery

# Firmware

Compromise between hardware and software

- *microcode*, *a.k.a.* *microprogramming*
- happens at *firmware* level:
  - microcode/firmware is stored in fast, special memory
  - requires special equipment to write
- level directly above the circuitery

- almost as fast as hardware
- *usually* updatable
- not portable, tied to hardware

# Firmware

Compromise between hardware and software

- *microcode*, *a.k.a.* *microprogramming*
- happens at *firmware* level:
  - microcode/firmware is stored in fast, special memory
  - requires special equipment to write
- level directly above the circuitery

- almost as fast as hardware
- *usually* updatable
- not portable, tied to hardware

For instance, *GPU*s, *HDD*s, remote controls, embedded systems, . . .