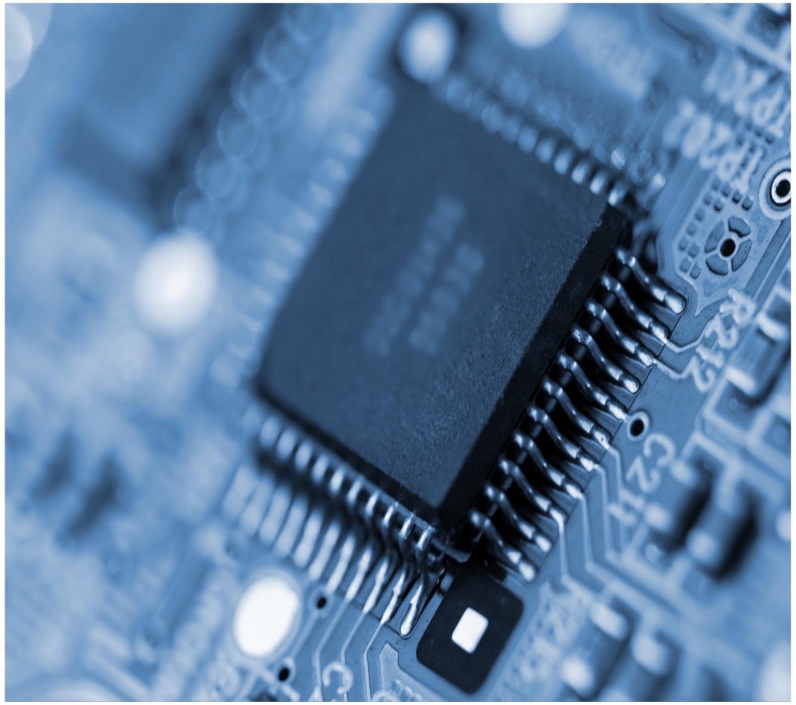
Computer Processors

Virtual Machines: Overview



This lecture is based on the excellent course *Nand to Tetris* by Noam Nisam and Shimon Schocken, and we reuse here many of the slides provided at www.nand2tetris.org

What is a virtual machine (VM)

VM is a used in two different contexts but they share a similar idea:

- *OS virtualization* - allows multiple operating systems to be installing and run concurrently on a single physical machine.

- *Abstract virtual machines* - allow for technical details to be abstracted away from an implementation. This is the technology that underpins modern compiler tool chains.

We are interested in Abstract virtual machines

Why VM?

```
// First example in Programming 101
class Main {
    function void main() {
        do Output.printString("Hello World!");
        do Output.println(); // New line.
        return;
    }
}
```

Why VM?

```
// First example in Programming 101
class Main {
    function void main() {
        do Output.printString("Hello World!");
        do Output.println(); // New line.
        return;
    }
}
```

Issues:

- Program execution
- Writing on the screen
- Handling class, function ...
- Handling do, while, ...
- function call and return
- operating system
- ...

Why VM?

```
// First example in Programming 101
class Main {
    function void main() {
        do Output.printString("Hello World!");
        do Output.println(); // New line.
        return;
    }
}
```

abstraction

Q: How can high-level programmers ignore all these issues?

A: They treat the high-level language as an ***abstraction***.

Why VM?

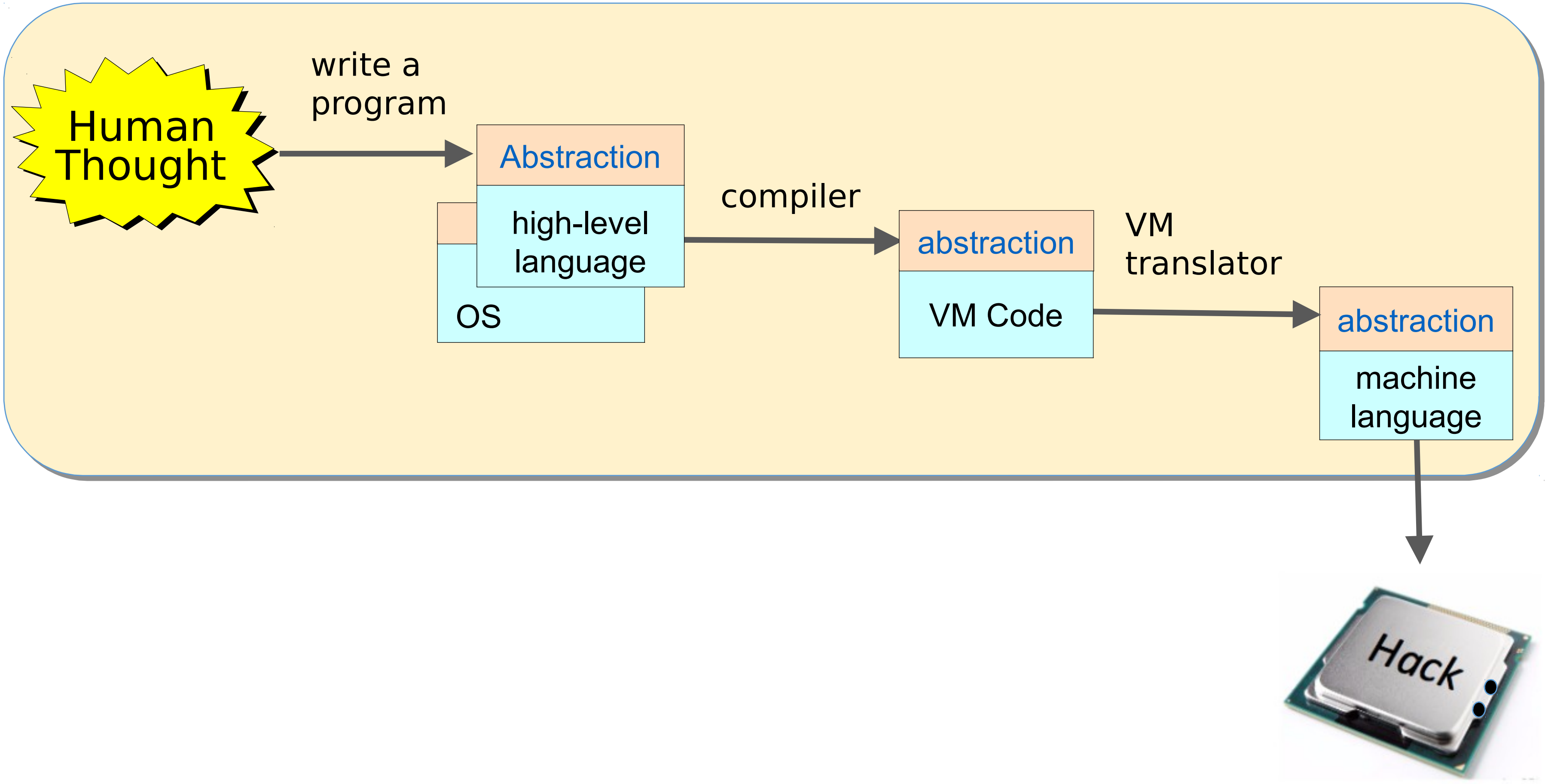
```
// First example in Programming 101
class Main {
    function void main() {
        do Output.printString("Hello World!");
        do Output.println(); // New line.
        return;
    }
}
```

abstraction

Q: What makes the abstraction work?

A:
- □ Assembler
- □ Virtual machine
- □ Compiler
- □ Operating system

Why VM?



Human Thought

write a program

Abstraction

high-level language

OS

compiler

abstraction

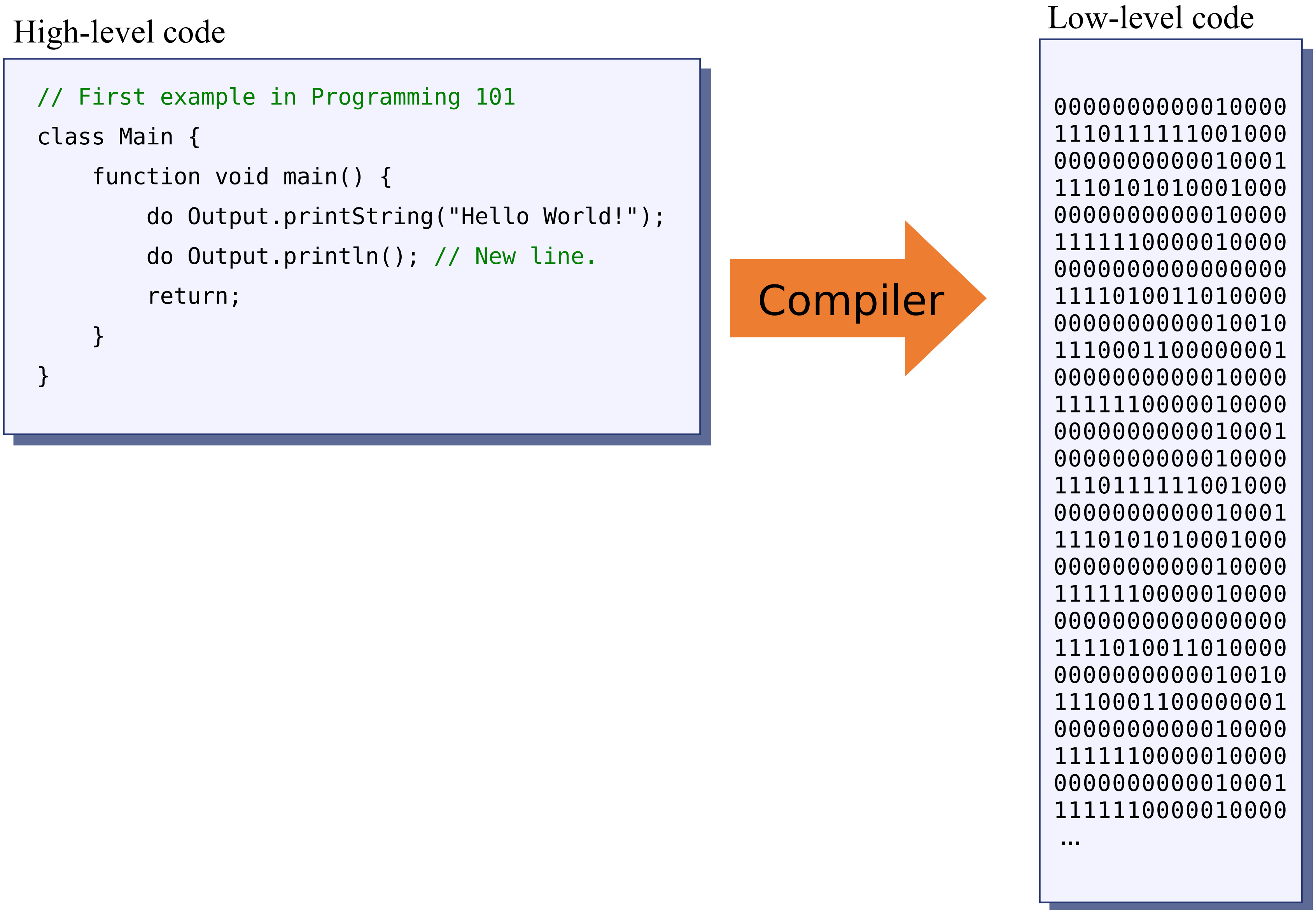
VM Code

VM translator

abstraction

machine language

Why VM? 1-tier compilation:



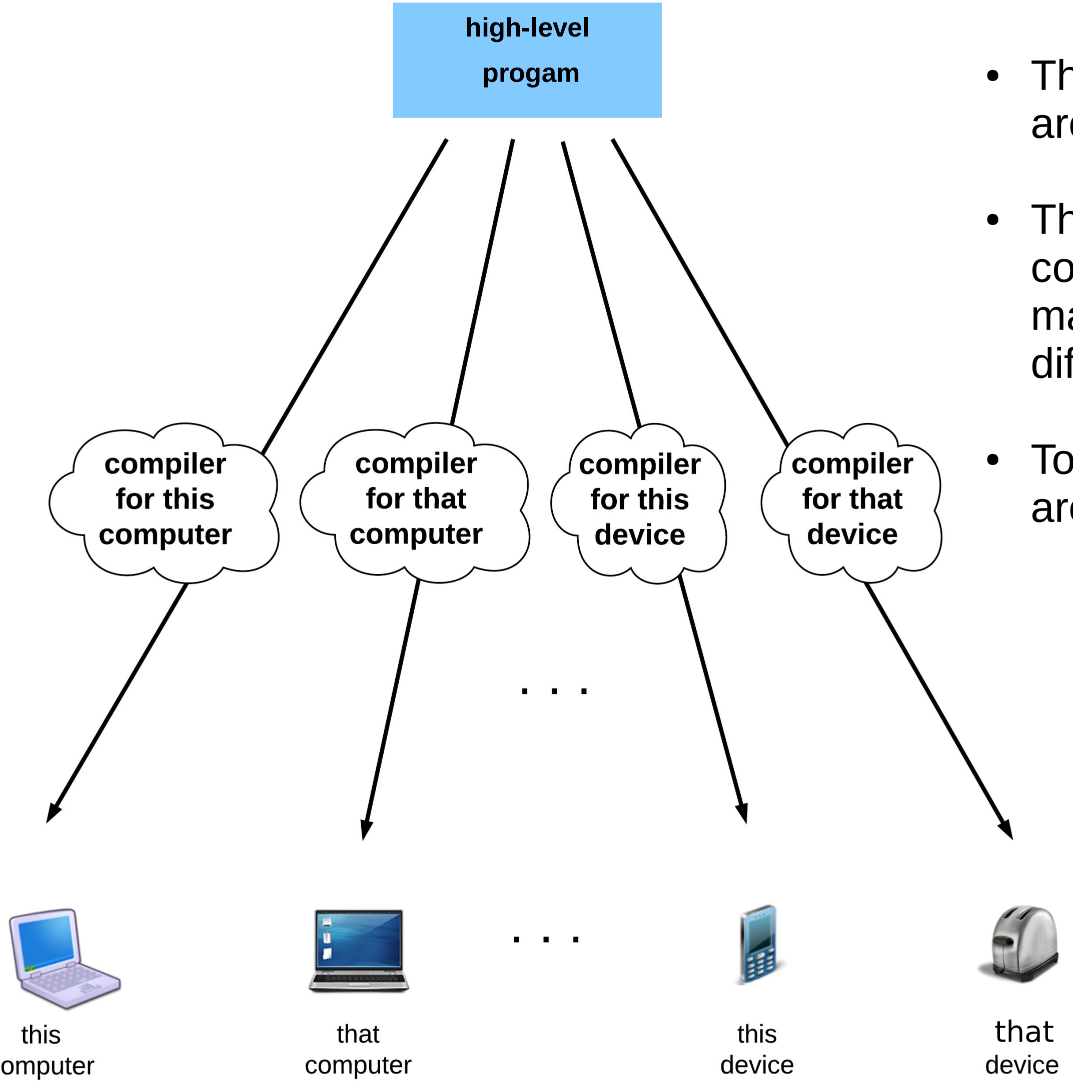
High-level code

```
// First example in Programming 101
class Main {
    function void main() {
        do Output.printString("Hello World!");
        do Output.println(); // New line.
        return;
    }
}
```

Low-level code

```
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
1111110000010000
...
```

Why VM? 1-tier compilation:

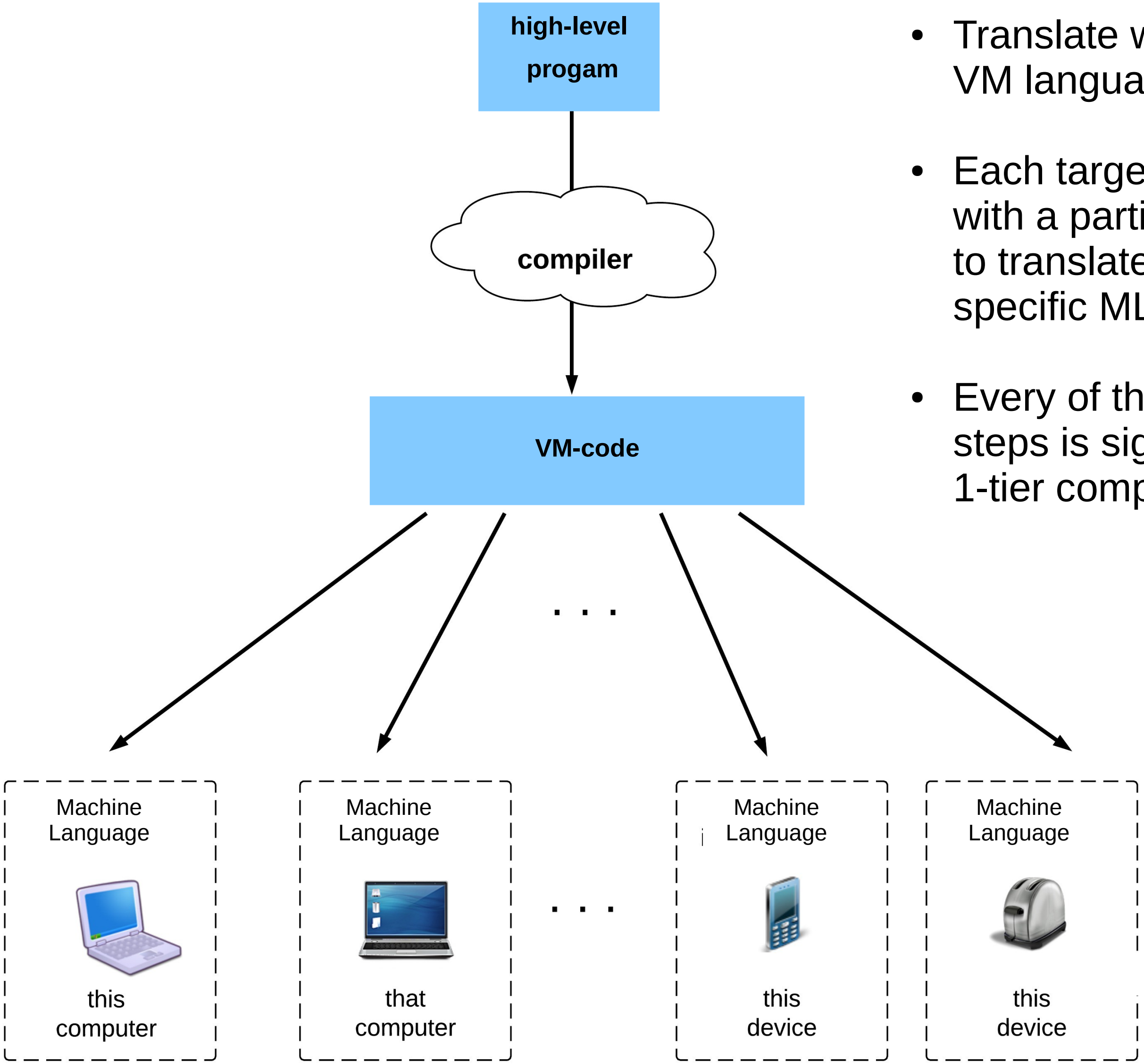


- There are many different computer architectures

- Thus, it's not enough to write one compiler only, you have to develop many compilers, one for every different architecture.

- To circumvent this issue VM are used

Why VM: 2-tier compilation:



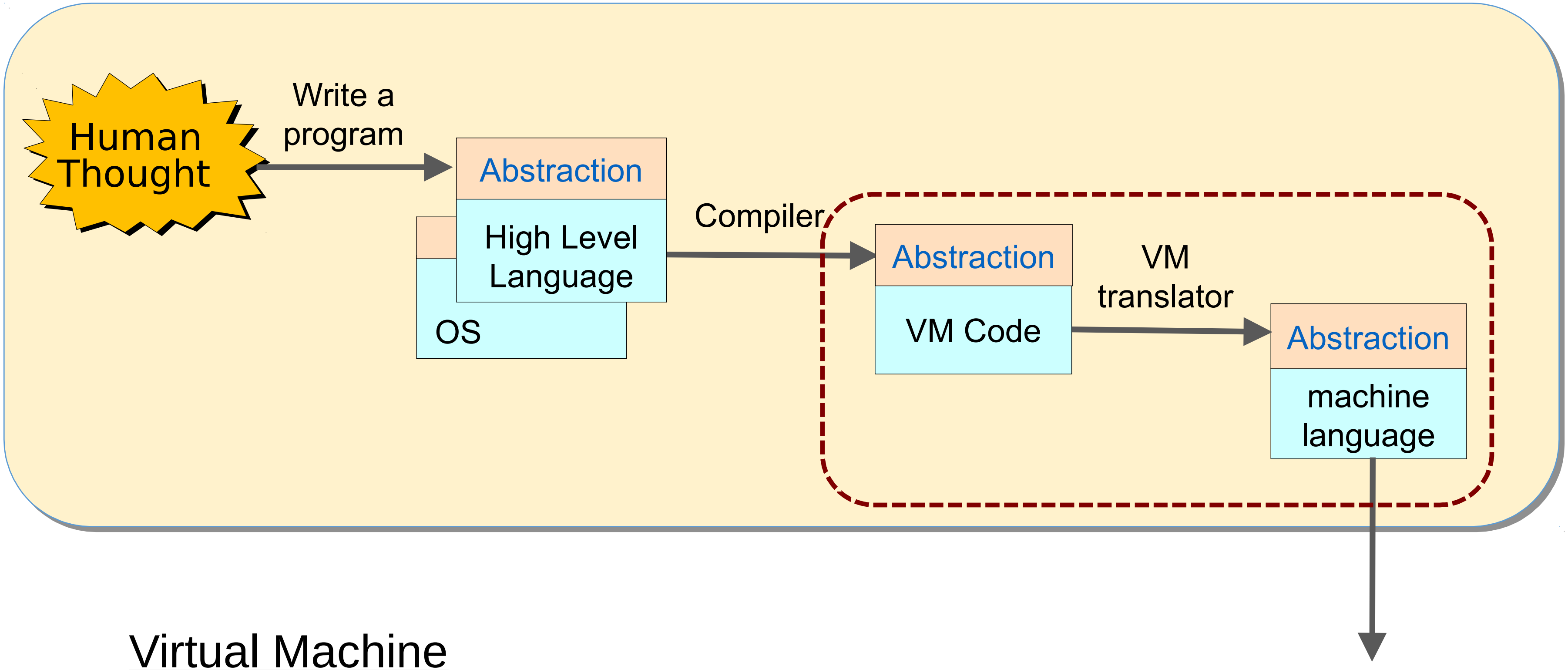
- Translate with compiler to VM language

- Each target architecture is equipped with a particular compiler that allows to translate the VM-code into the specific ML code

- Every of these single “translation” steps is significantly simpler than 1-tier compilation

Virtual Machine (VM)

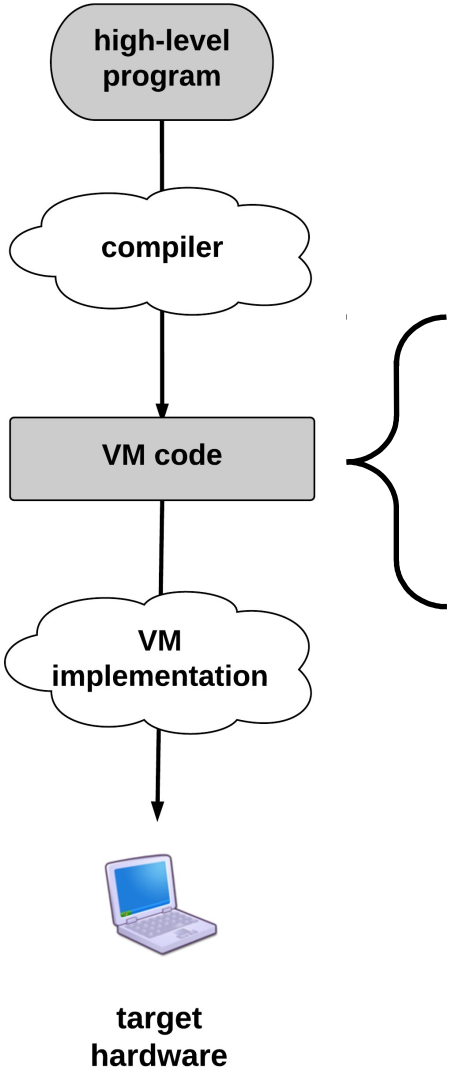
UNIVERSITY OF LEEDS



Virtual Machine

- Understanding the VM abstraction
- Building a VM implementation

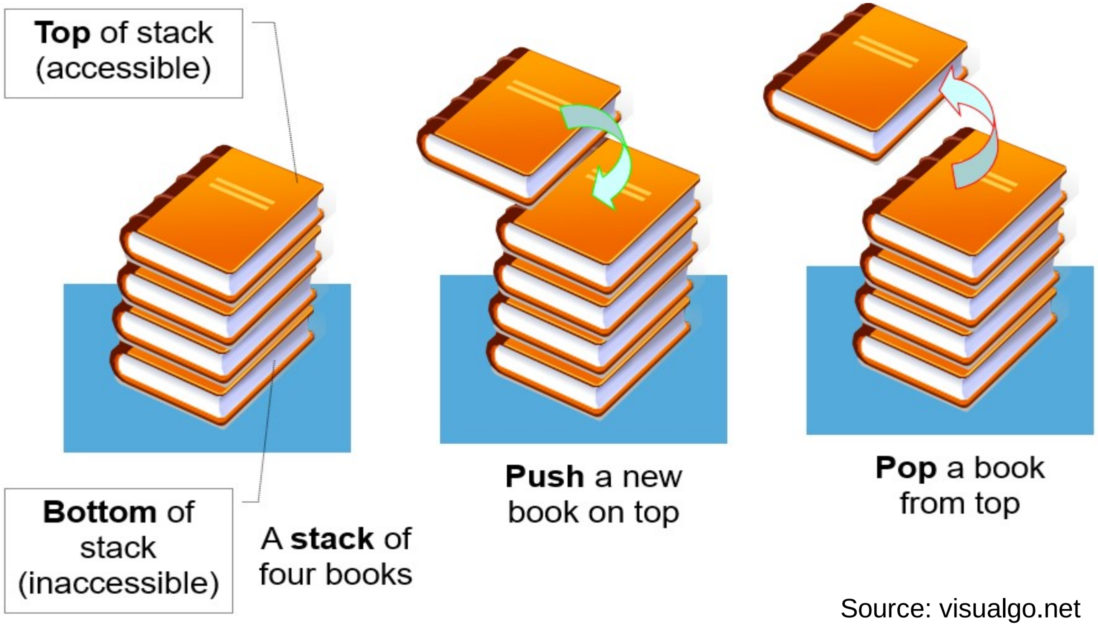
Virtual Machine (VM)



Stack machine abstraction

- Architecture
- Commands

Stack


Top of stack (accessible)
Bottom of stack (inaccessible)
A **stack** of four books
Push a new book on top
Pop a book from top

Source: visualgo.net

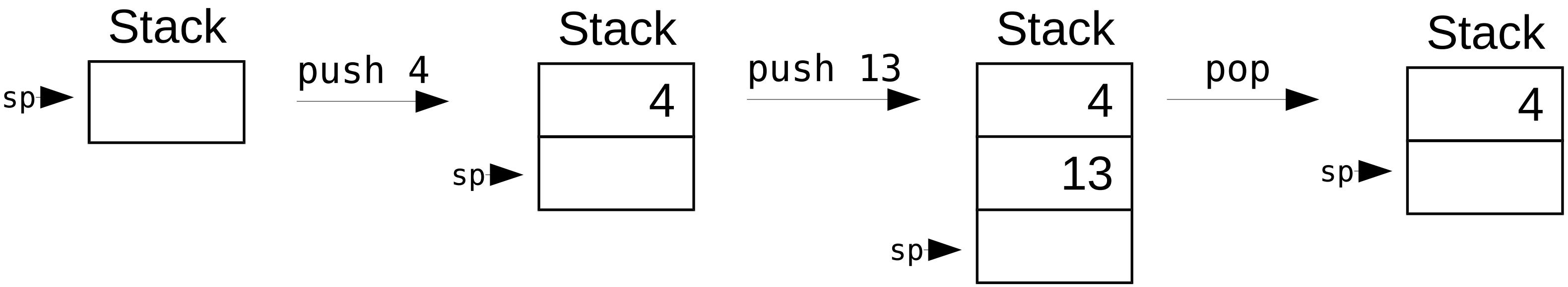
<u>Stack</u> = data structure with two operations to manipulate the stack:

- push: add an element at the stack's top
- pop: remove the top element

Known as: **L**ast-**i**n-**F**irst-**o**ut (LiFo) stack

Stack

UNIVERSITY OF LEEDS



`sp` = stack pointer which “points” to location where next element will be added (top)

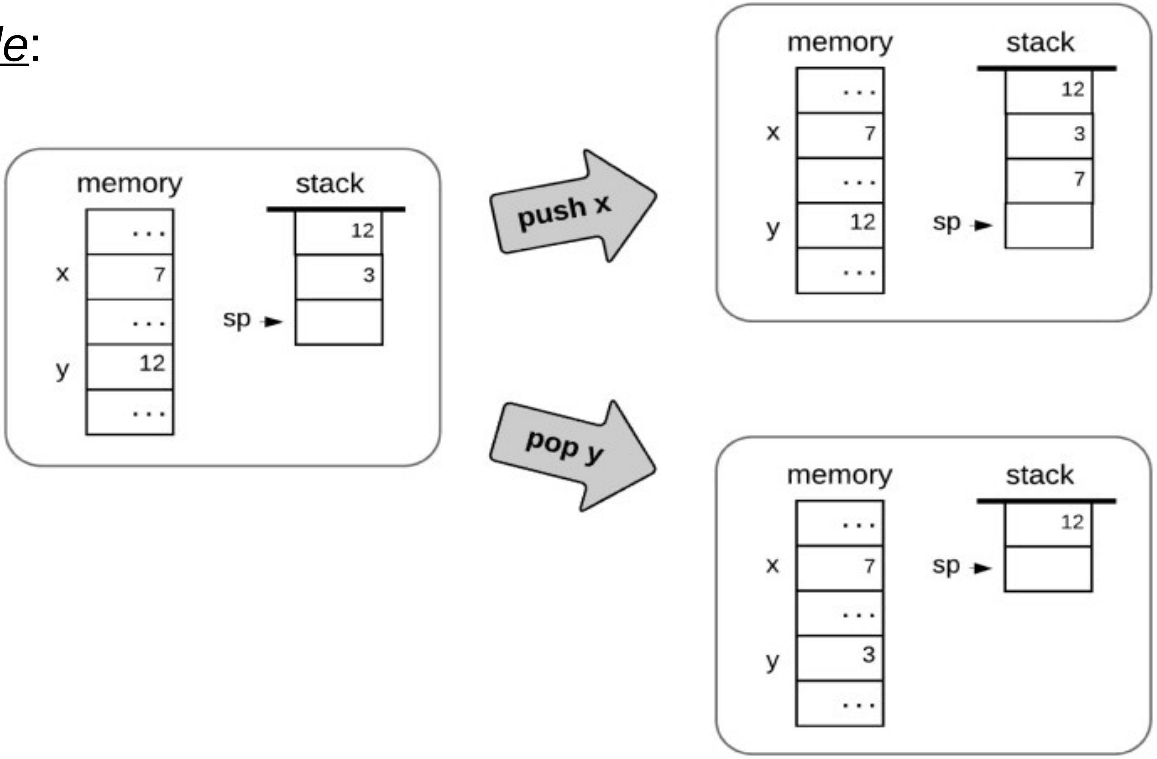
VM and stack machines

- The VM we use is stack based, will support functions and memory (=stack machine model) which is a common way to represent VMs
- The elements that we push to / pop from the stack are *operands*

VM and stack machines

- The VM we use is **stack** based, will support functions and **memory** (=stack machine model) which is a common way to represent VMs
- The elements that we push to / pop from the stack are *operands*

Example:

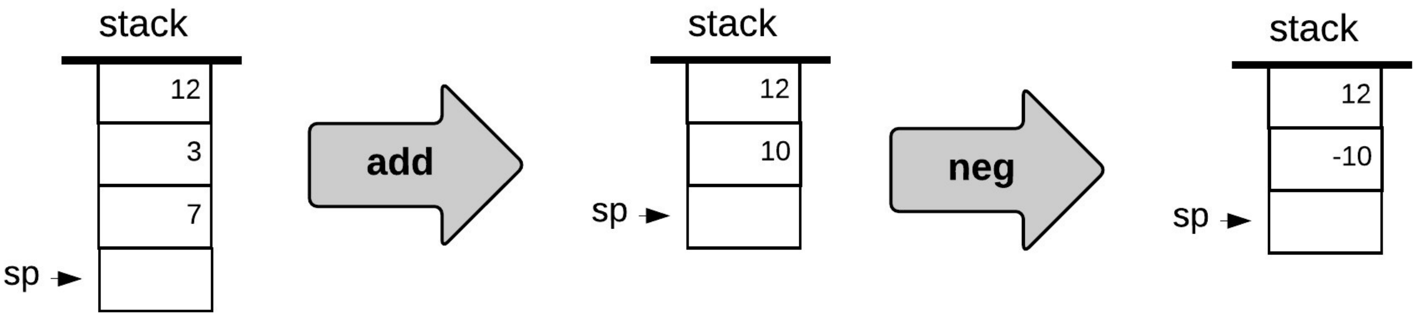


VM and stack machines

UNIVERSITY OF LEEDS

- The VM we use is stack based, will support **functions** and memory (=stack machine model) which is a common way to represent VMs
- The elements that we push to / pop from the stack are *operands*

Example:



Applying a function *f* on the stack:

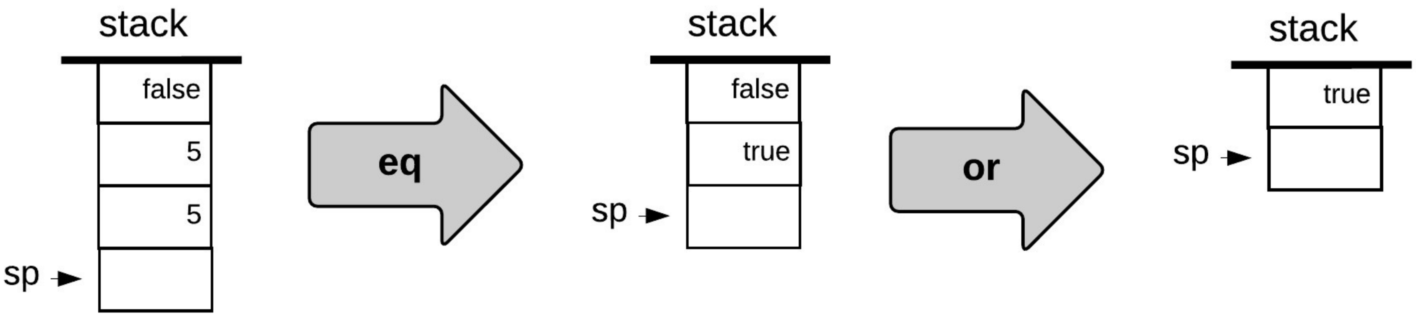
- pops the argument(s) from the stack
- Computes *f* on the arguments
- Pushes the result onto the stack.

VM and stack machines

UNIVERSITY OF LEEDS

- The VM we use is stack based, will support **functions** and memory (=stack machine model) which is a common way to represent VMs
- The elements that we push to / pop from the stack are *operands*

Example:



Applying a function *f* on the stack:

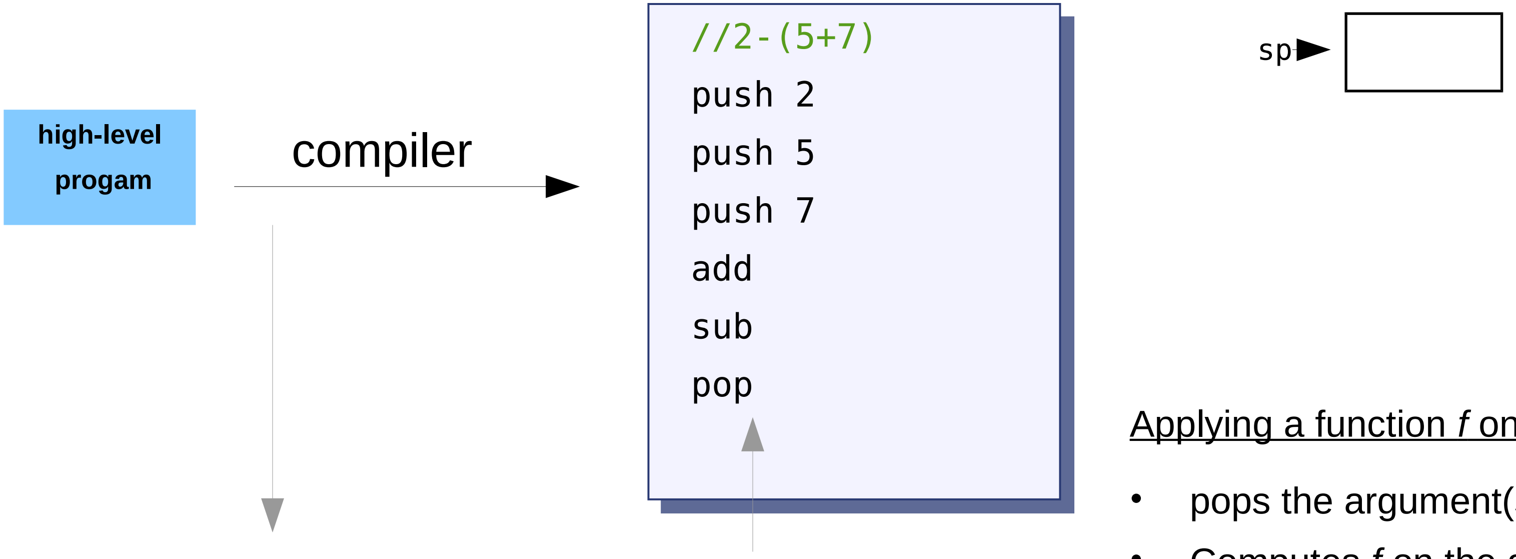
- pops the argument(s) from the stack
- Computes *f* on the arguments
- Pushes the result onto the stack.

VM and stack machines

- The VM we use is stack based, will support functions and memory (=stack machine model) which is a common way to represent VMs
- The elements that we push to / pop from the stack are *operands*

Example:

VM code



```
//2-(5+7)
push 2
push 5
push 7
add
sub
pop
```

2-(5+7) is "translated to"

Applying a function *f* on the stack:

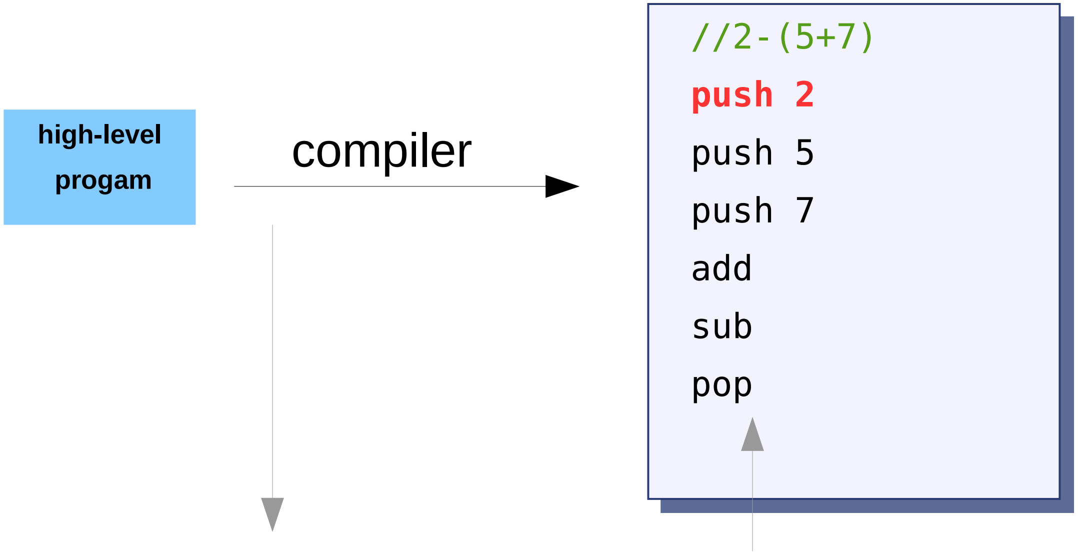
- pops the argument(s) from the stack
- Computes *f* on the arguments
- Pushes the result onto the stack.

VM and stack machines

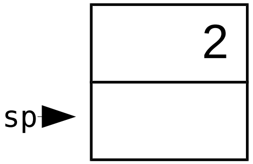
- The VM we use is stack based, will support functions and memory (=stack machine model) which is a common way to represent VMs
- The elements that we push to / pop from the stack are *operands*

Example:

VM code



```
//2-(5+7)
push 2
push 5
push 7
add
sub
pop
```



2-(5+7) is “translated to” 2

Applying a function *f* on the stack:

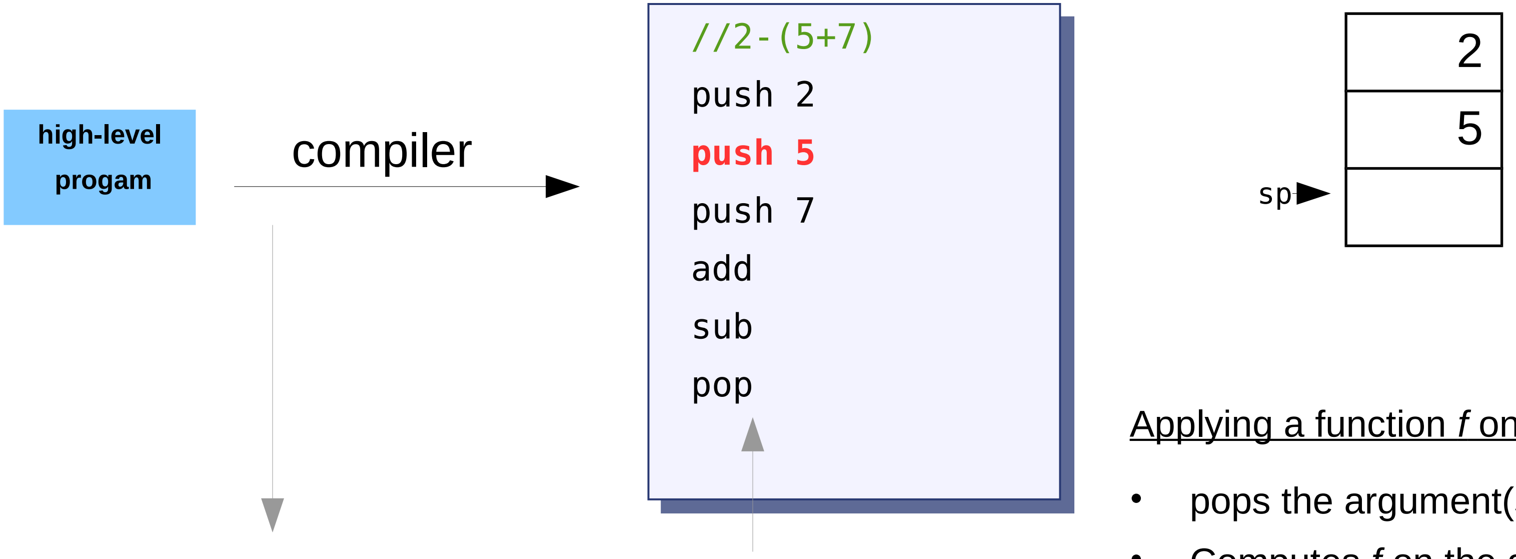
- pops the argument(s) from the stack
- Computes *f* on the arguments
- Pushes the result onto the stack.

VM and stack machines

- The VM we use is stack based, will support functions and memory (=stack machine model) which is a common way to represent VMs
- The elements that we push to / pop from the stack are *operands*

Example:

VM code



```
//2-(5+7)
push 2
push 5
push 7
add
sub
pop
```

2-(5+7) is "translated to" 2 5

Applying a function *f* on the stack:

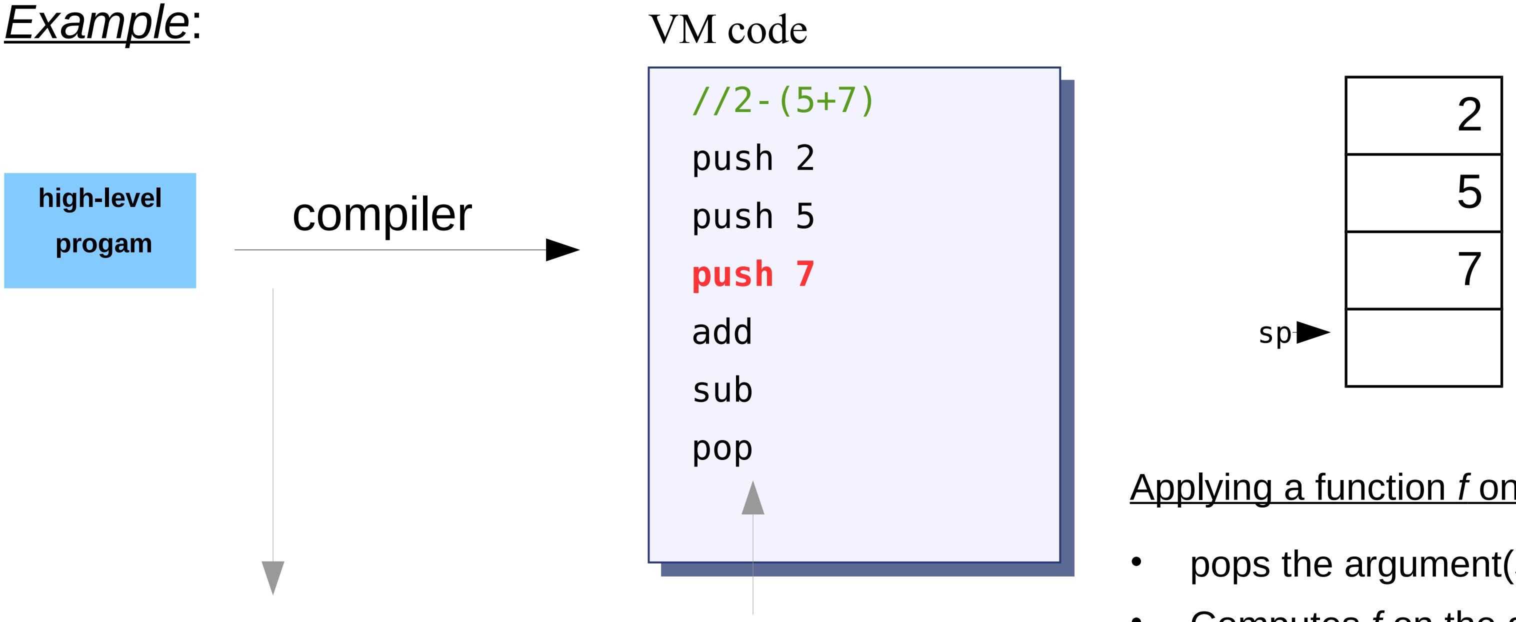
- pops the argument(s) from the stack
- Computes *f* on the arguments
- Pushes the result onto the stack.

VM and stack machines

UNIVERSITY OF LEEDS

- The VM we use is stack based, will support functions and memory (=stack machine model) which is a common way to represent VMs
- The elements that we push to / pop from the stack are *operands*

Example:



```
//2-(5+7)
push 2
push 5
push 7
add
sub
pop
```

`2-(5+7)` is “translated to” `2 5 7`

Applying a function *f* on the stack:

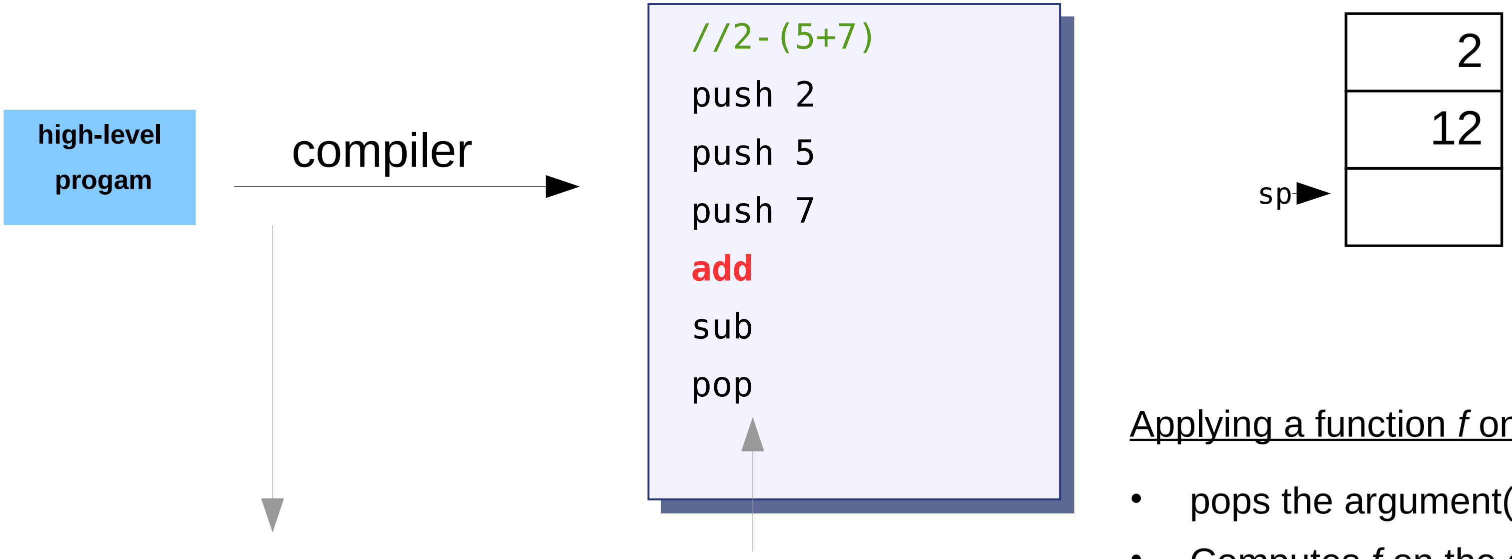
- pops the argument(s) from the stack
- Computes *f* on the arguments
- Pushes the result onto the stack.

VM and stack machines

- The VM we use is stack based, will support functions and memory (=stack machine model) which is a common way to represent VMs
- The elements that we push to / pop from the stack are *operands*

Example:

VM code



```
//2-(5+7)
push 2
push 5
push 7
add
sub
pop
```

2-(5+7) is "translated to" 2 5 7 +

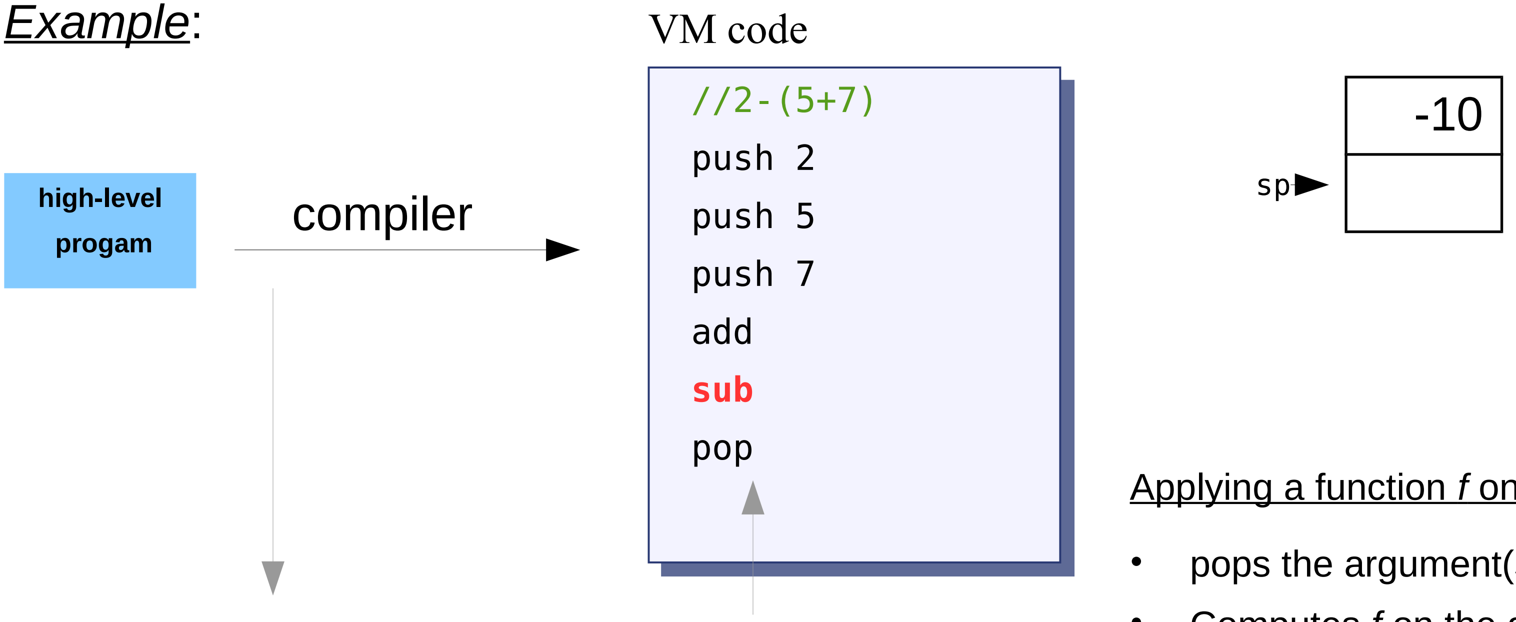
Applying a function *f* on the stack:

- pops the argument(s) from the stack
- Computes *f* on the arguments
- Pushes the result onto the stack.

VM and stack machines

- The VM we use is stack based, will support functions and memory (=stack machine model) which is a common way to represent VMs
- The elements that we push to / pop from the stack are *operands*

Example:



```
//2-(5+7)
push 2
push 5
push 7
add
sub
pop
```

`2-(5+7)` is “translated to” `2 5 7 + -`

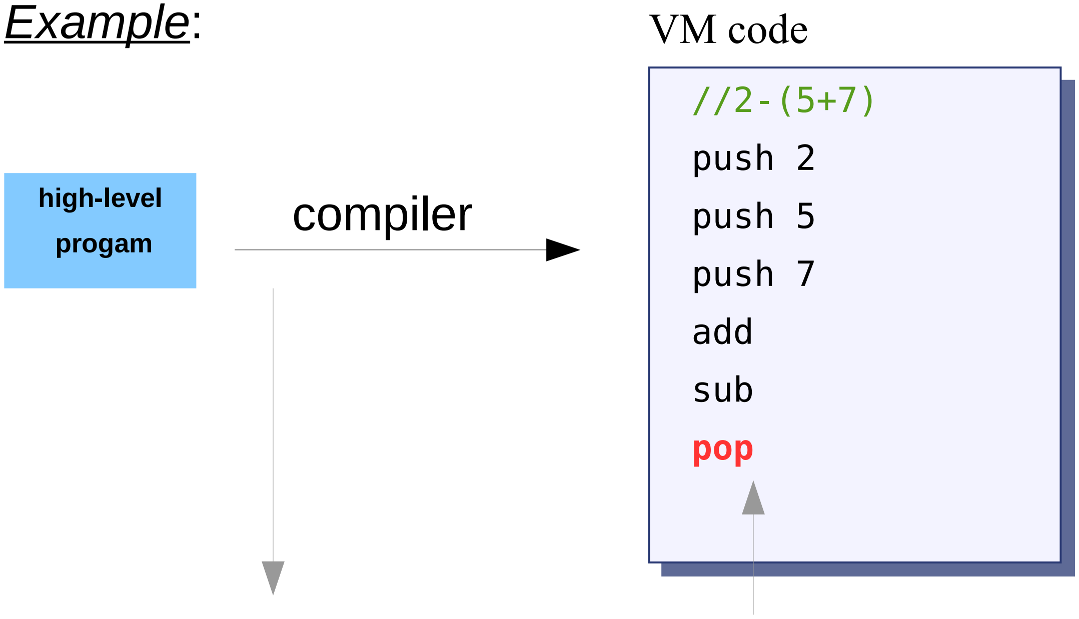
Applying a function *f* on the stack:

- pops the argument(s) from the stack
- Computes *f* on the arguments
- Pushes the result onto the stack.

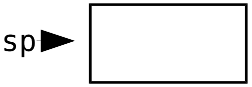
VM and stack machines

- The VM we use is stack based, will support functions and memory (=stack machine model) which is a common way to represent VMs
- The elements that we push to / pop from the stack are *operands*

Example:



```
//2-(5+7)
push 2
push 5
push 7
add
sub
pop
```



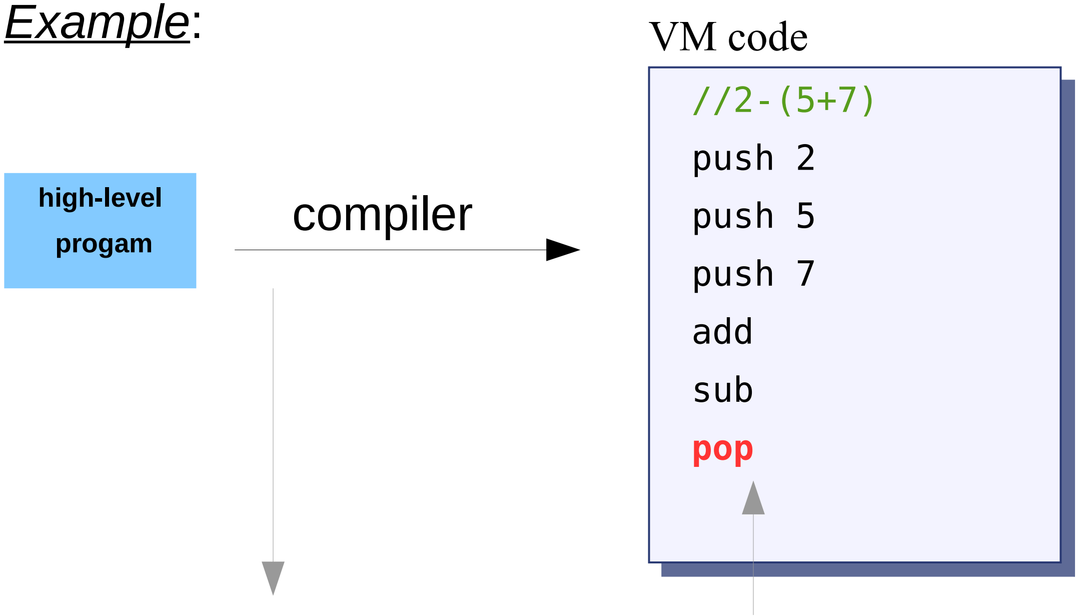
2-(5+7) is “translated to” 2 5 7 + -

Applying a function *f* on the stack:

- pops the argument(s) from the stack
- Computes *f* on the arguments
- Pushes the result onto the stack.

VM and stack machines

Example:



```
//2-(5+7)
push 2
push 5
push 7
add
sub
pop
```

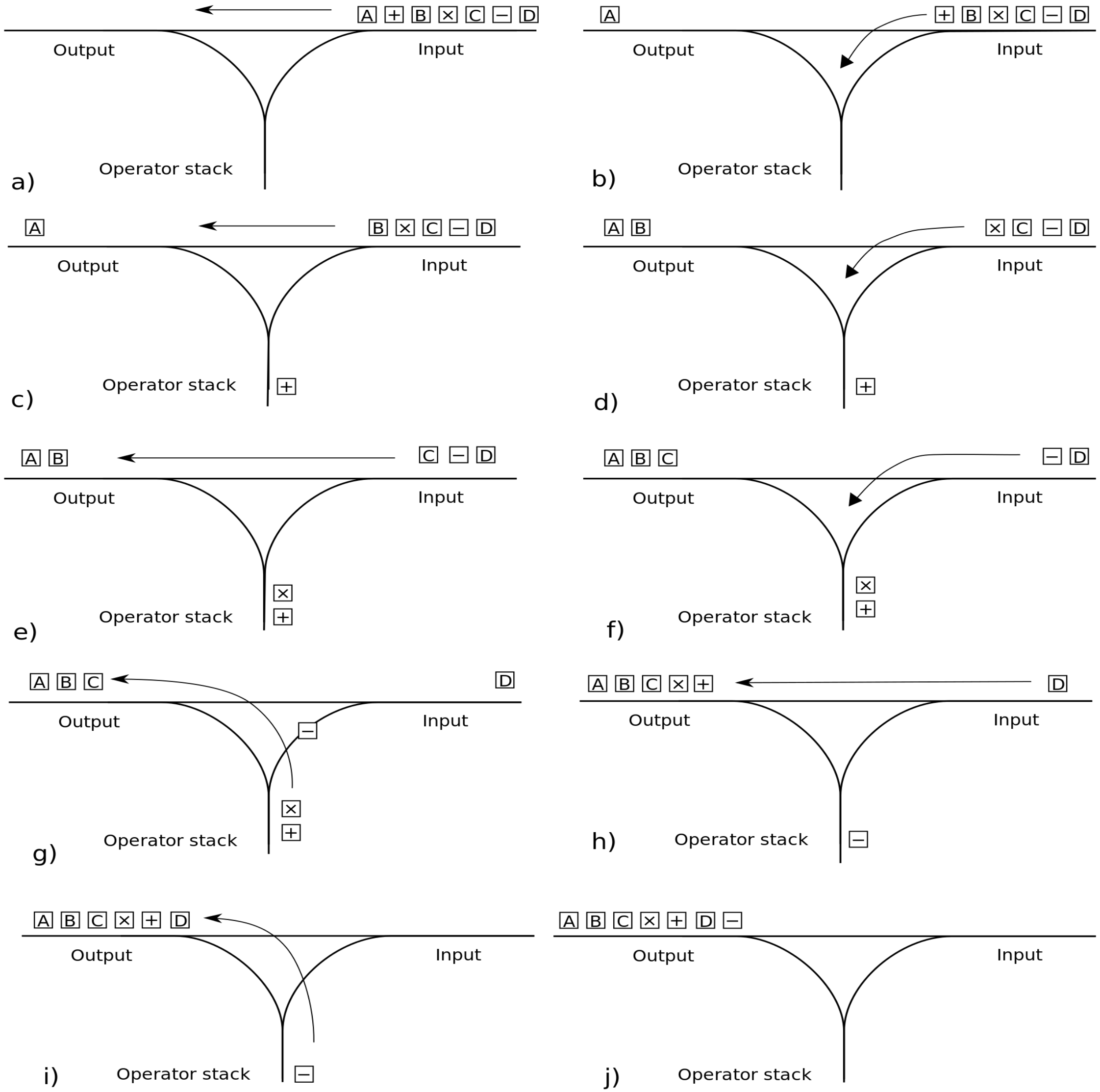
`2-(5+7)` is “translated to” `2 5 7 + -`

- Arithmetic or logical expressions are in *infix* notation
- The latter translation is known as *postfix* or *reverse polish notation* and this notation can then be translated to VM-code (done by compiler)
- To compute *reverse polish notation* algorithms exists e.g. Dijkstra's *Shunting-yard-Algorithmus*

Shunting-yard algorithm in a nutshell

The input is processed one symbol at a time:

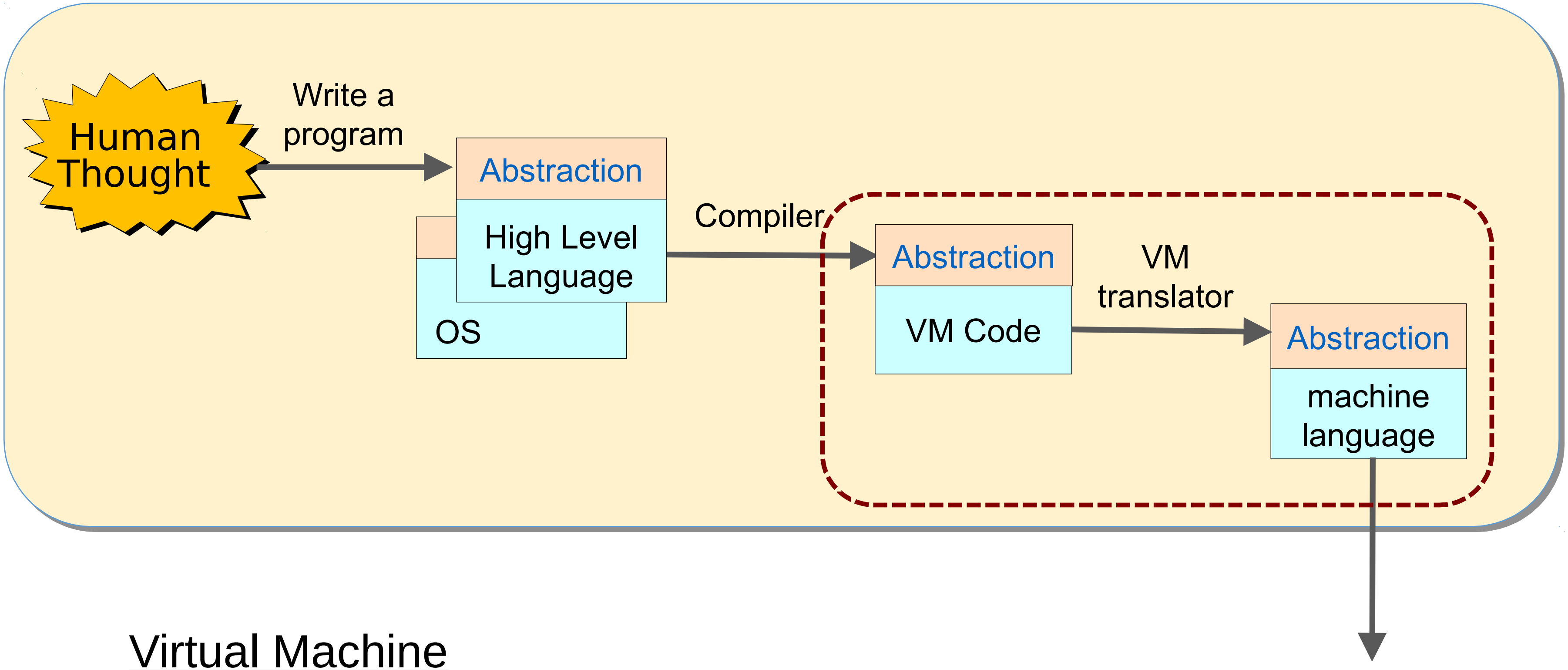
- if a variable or number is found, it is outputted a), c), e), h).
- If the symbol is an operator, it is pushed onto the operator stack b), d), f).
- If the operator's precedence is less than that of the operators at the top of the stack or the precedences are equal and the operator is left associative, then that operator is popped off the stack and added to the output g).
- Finally, any remaining operators are popped off the stack and added to the output i).



source: commons.wikimedia.org/wiki/File:Shunting_yard.svg

Virtual Machine (VM)

UNIVERSITY OF LEEDS



Virtual Machine

- Understanding the VM abstraction
- Building a VM implementation