Modules I

15-150

Lecture 16: October 29, 2024

Stephanie Balzer
Carnegie Mellon University

An important idea in computer science

An important idea in computer science

Abstraction

An important idea in computer science

Abstraction

What is abstraction? What does it entail?

An important idea in computer science

Abstraction

- What is abstraction? What does it entail?
- Separation of **specification** from **implementation**.

An important idea in computer science

Abstraction

- What is abstraction? What does it entail?
- Separation of **specification** from **implementation**.

Specification: externally visible promise to deliver

An important idea in computer science

Abstraction

What is abstraction? What does it entail?

Separation of **specification** from **implementation**.

Specification: externally visible promise to deliver

Implementation: internal choice of how to deliver promise

An important idea in computer science

Abstraction

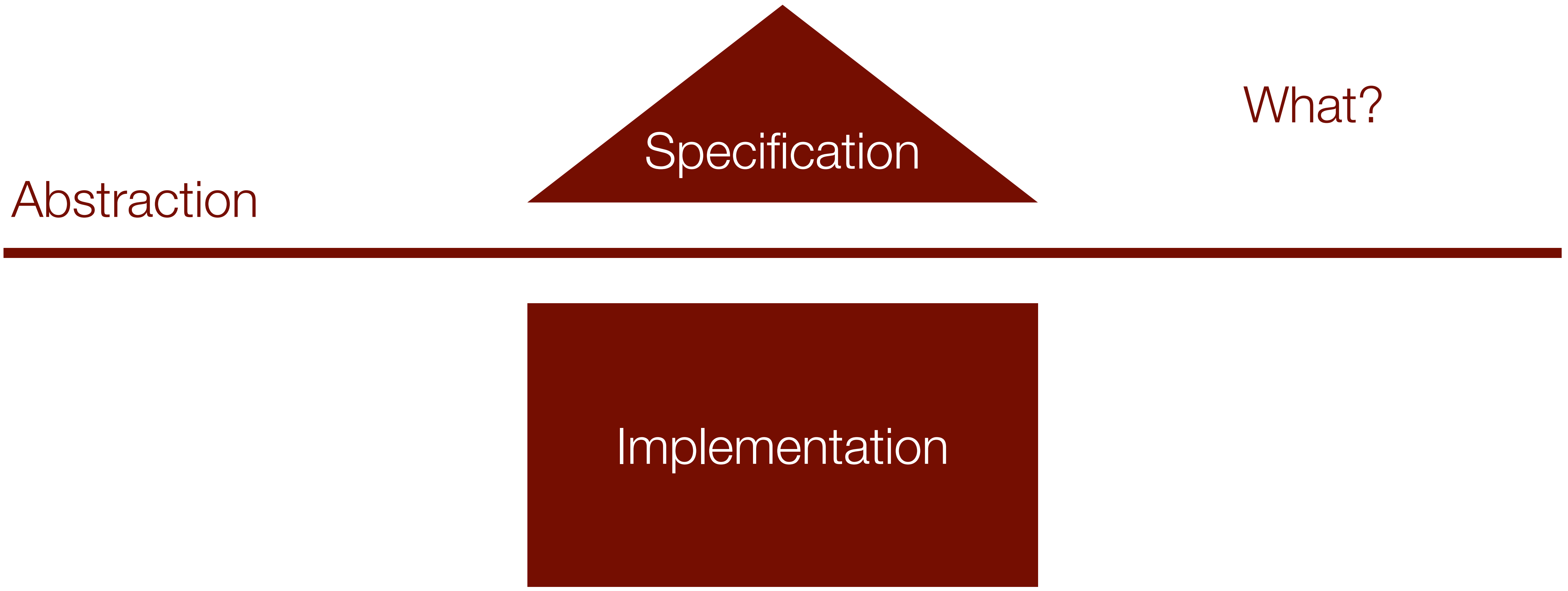
An important idea in computer science

Abstraction

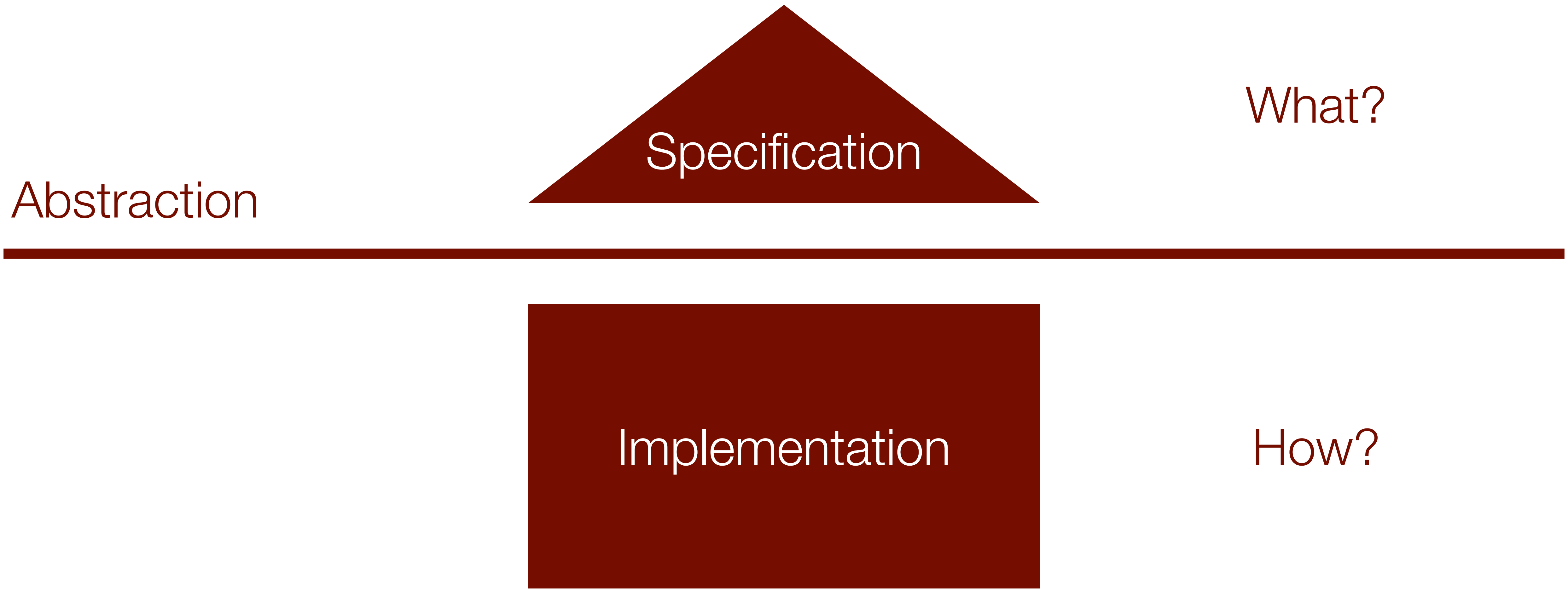
An important idea in computer science

An important idea in computer science

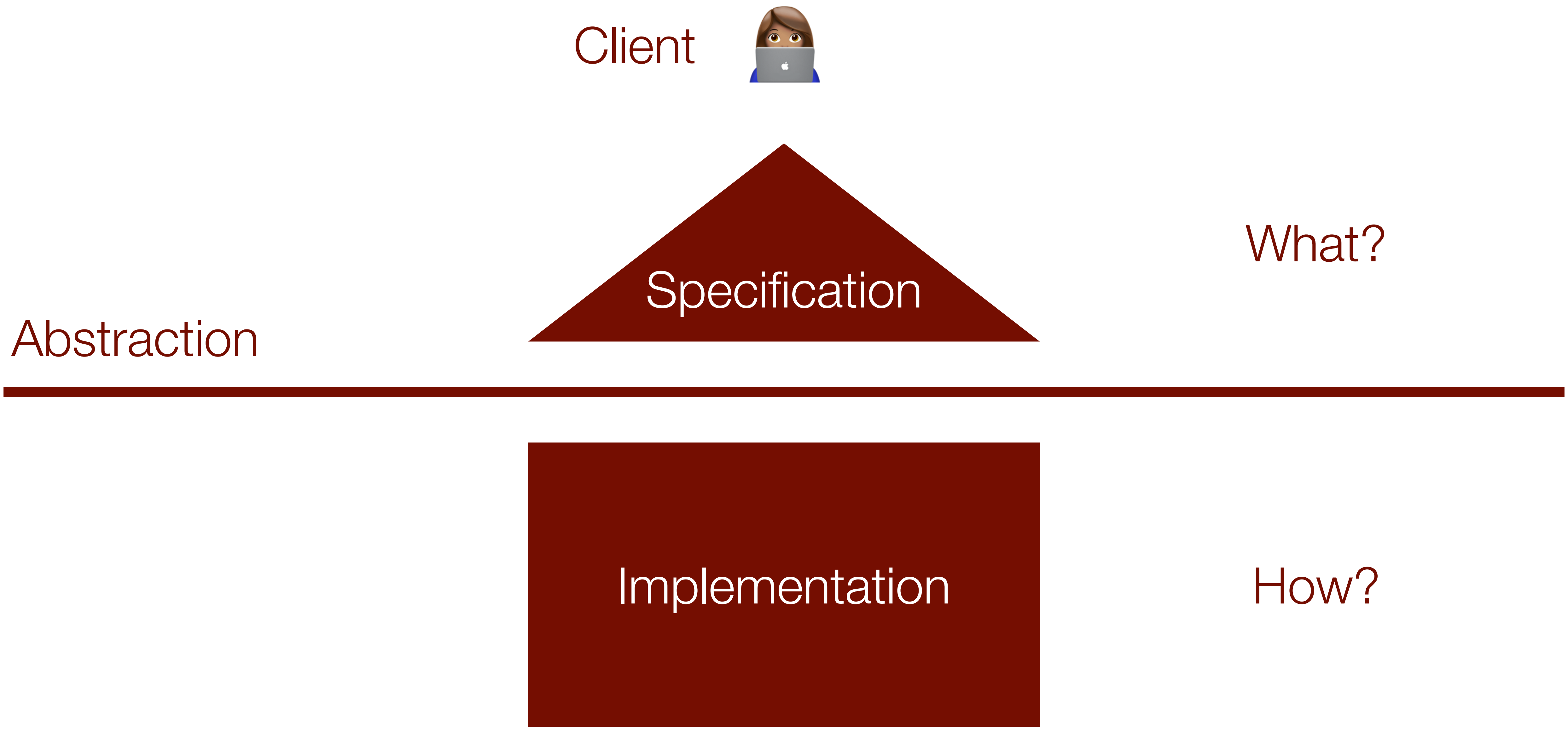
An important idea in computer science



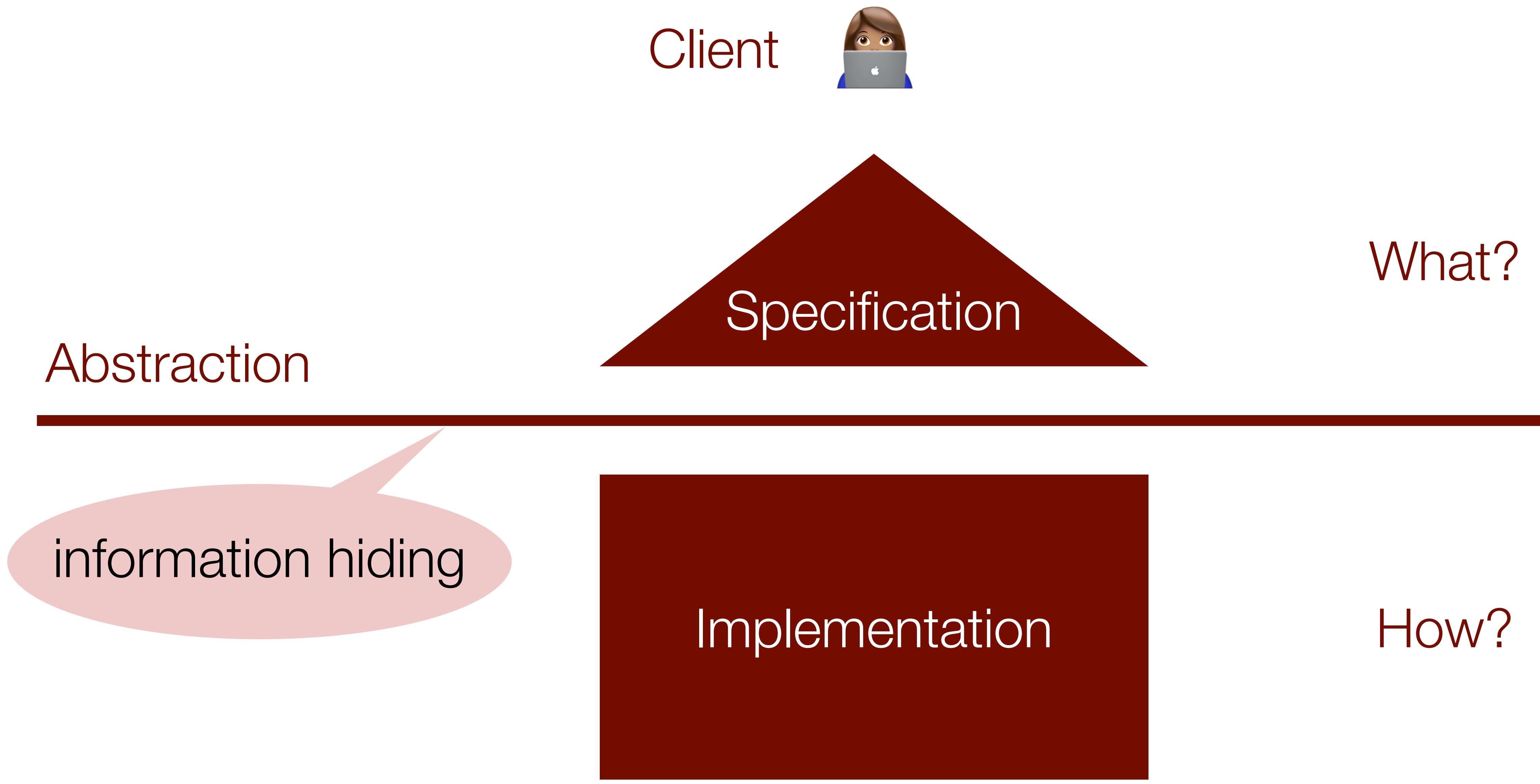
An important idea in computer science



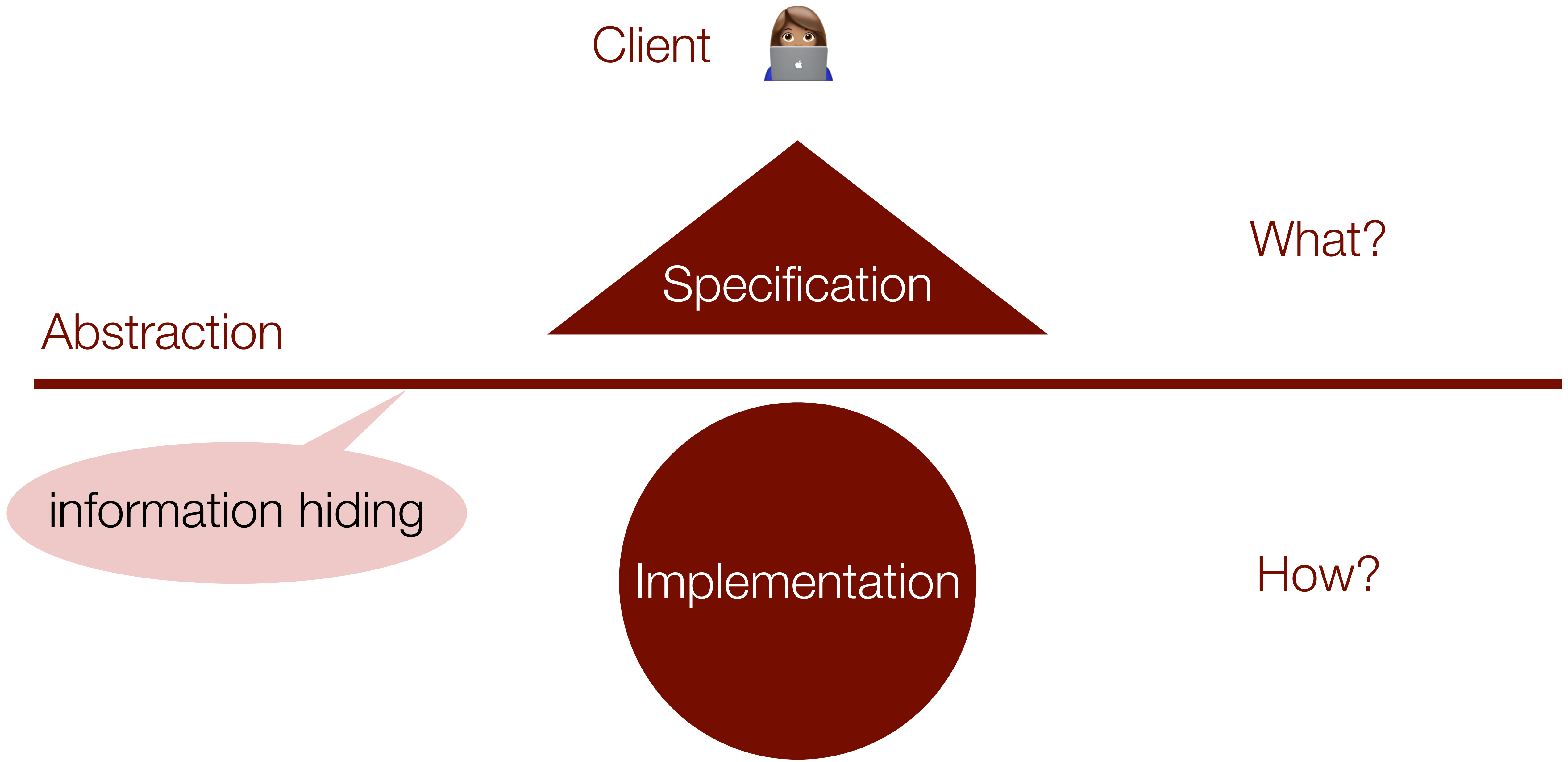
An important idea in computer science



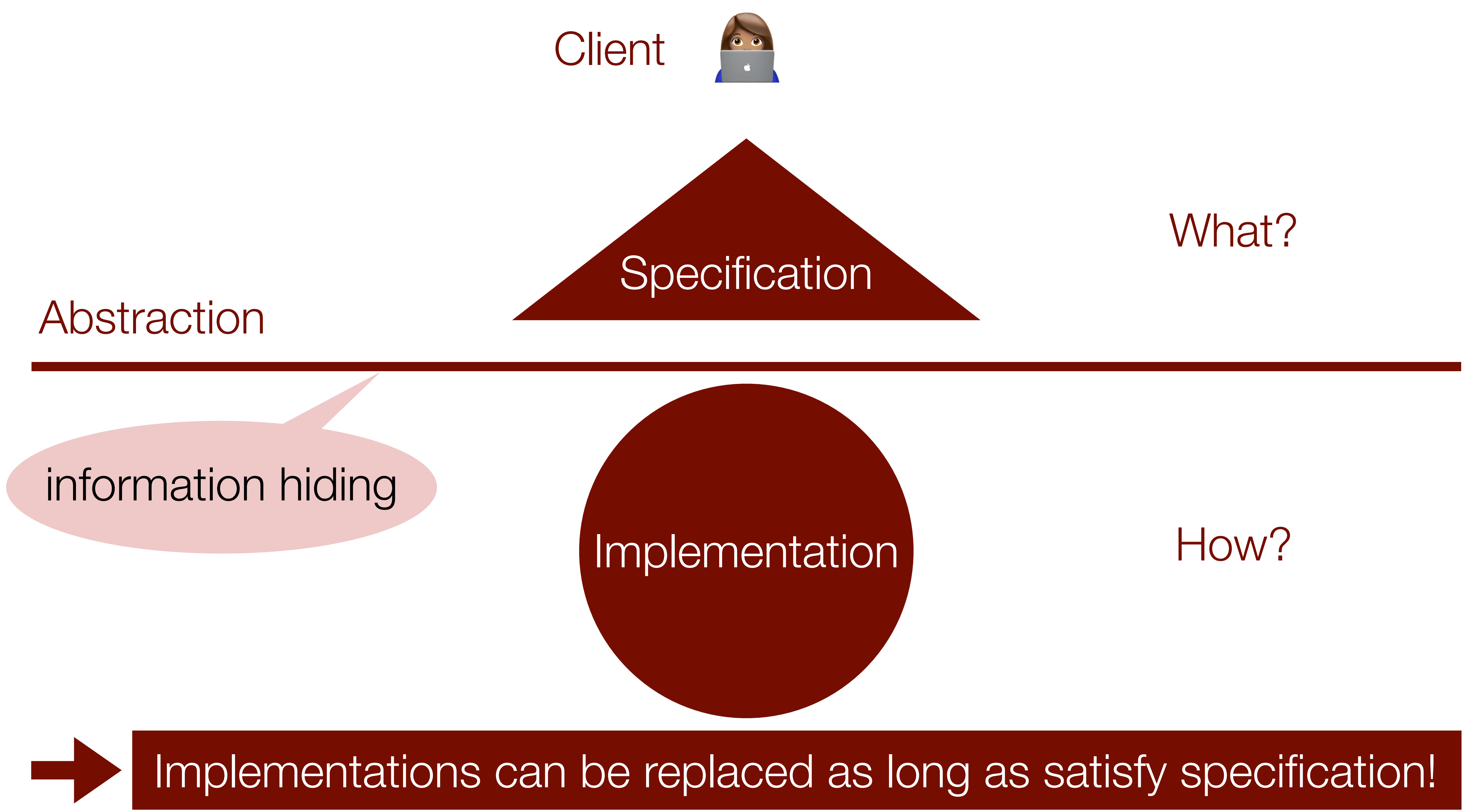
An important idea in computer science



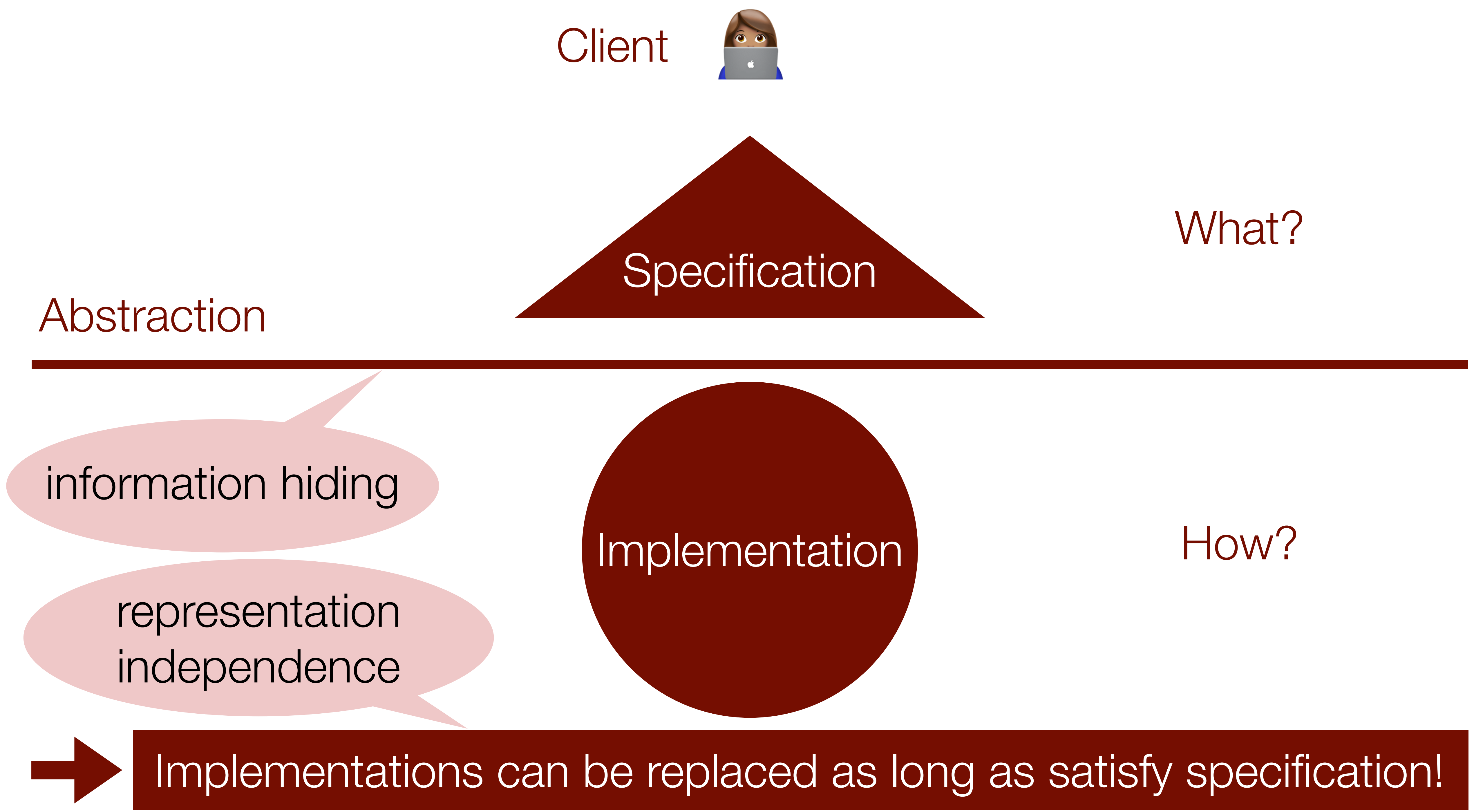
An important idea in computer science



An important idea in computer science



An important idea in computer science



An important idea in computer science

An important idea in computer science

Benefits of abstraction:

An important idea in computer science

Benefits of abstraction:

Code evolution without disturbing client code.

An important idea in computer science

Benefits of abstraction:

- Code evolution without disturbing client code.
- Reasoning: client only needs to consider specification.

An important idea in computer science

Benefits of abstraction:

- Code evolution without disturbing client code.
- Reasoning: client only needs to consider specification.
- Separate development: specifications as blueprint.

An important idea in computer science

Benefits of abstraction:

- Code evolution without disturbing client code.
- Reasoning: client only needs to consider specification.
- Separate development: specifications as blueprint.
- The only means by which programs become scalable.

Abstractions in SML

Abstractions in SML

Abstraction at small: functions

Abstractions in SML

Abstraction at small: functions

➡ **specification: function type with pre-/post-condition**

Abstractions in SML

Abstraction at small: functions

- specification: function type with pre-/post-condition
- implementation: function body

Abstractions in SML

Abstraction at small: functions

- specification: function type with pre-/post-condition
- implementation: function body

Abstraction at large: modules

Abstractions in SML

Abstraction at small: functions

- specification: function type with pre-/post-condition
- implementation: function body

Abstraction at large: modules

- Offer a way to combine what belongs together in one unit.

Abstractions in SML

Abstraction at small: functions

- specification: function type with pre-/post-condition
- implementation: function body

Abstraction at large: modules

- Offer a way to combine what belongs together in one unit.
- Specification: **signature**.

Abstractions in SML

Abstraction at small: functions

- specification: function type with pre-/post-condition
- implementation: function body

Abstraction at large: modules

- Offer a way to combine what belongs together in one unit.
- Specification: **signature**.
- Implementation: **structure**.

Abstractions in SML

Abstraction at small: functions

- specification: function type with pre-/post-condition
- implementation: function body

Abstraction at large: modules

- Offer a way to combine what belongs together in one unit.
- Specification: **signature**.
- Implementation: **structure**.
- Composing structures using **functors**.

Abstractions in SML

Abstraction at large: modules

- Offer a way to combine what belongs together in one unit.
- Specification: **signature**.
- Implementation: **structure**.
- Composing structures using **functors**.

Abstractions in SML

Abstraction at large: modules

- Offer a way to combine what belongs together in one unit.
- Specification: **signature**.
- Implementation: **structure**.
- Composing structures using **functors**.

SML Basis Library makes use of modules. E.g.,

Abstractions in SML

Abstraction at large: modules

- Offer a way to combine what belongs together in one unit.
- Specification: **signature**.
- Implementation: **structure**.
- Composing structures using **functors**.

SML Basis Library makes use of modules. E.g.,

```
Int.toString
```

Abstractions in SML

Abstraction at large: modules

- Offer a way to combine what belongs together in one unit.
- Specification: **signature**.
- Implementation: **structure**.
- Composing structures using **functors**.

SML Basis Library makes use of modules. E.g.,

```
Int.toString
```

is a function inside a **structure** called `Int`. The structure `Int` has the **signature** called `INTEGER` ascribed.

Abstractions in SML

Abstraction at large: modules

- Offer a way to combine what belongs together in one unit.
- Specification: **signature**.
- Implementation: **structure**.
- Composing structures using **functors**.

SML Basis Library makes use of modules. E.g.,

`Int.toString`

is a function inside a **structure** called `Int`. The structure `Int` has the **signature** called `INTEGER` ascribed.

- Let's unravel…

Modules at an example: queue

Modules at an example: queue

Queue, a **first-in first-out** (FIFO) data structure.

Modules at an example: queue

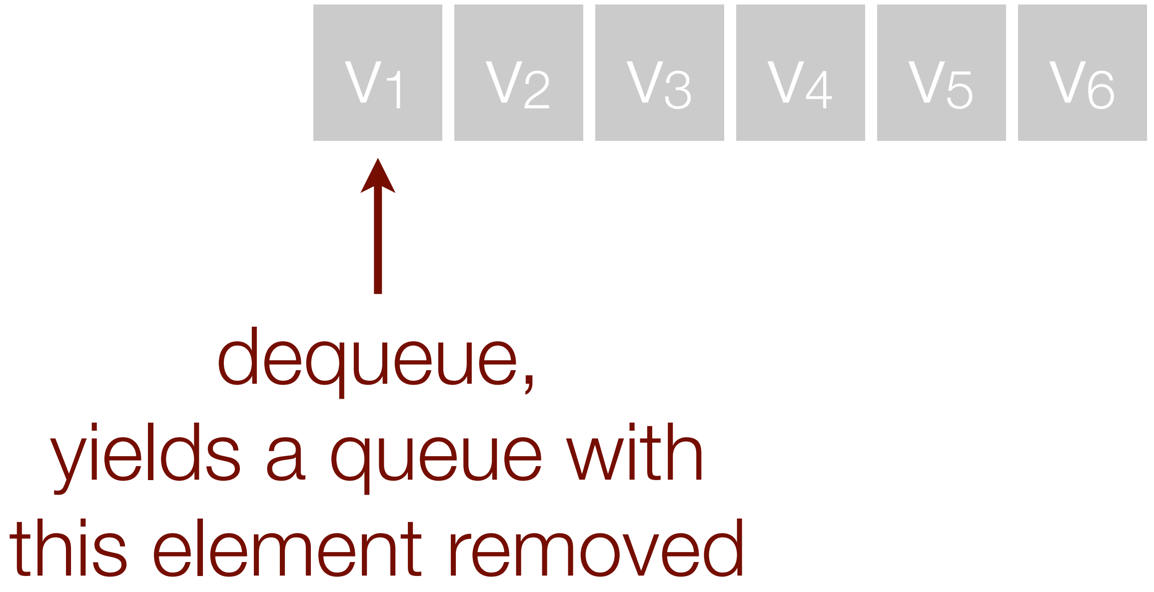
Queue, a **first-in first-out** (FIFO) data structure.

Modules at an example: queue

Queue, a **first-in first-out** (FIFO) data structure.

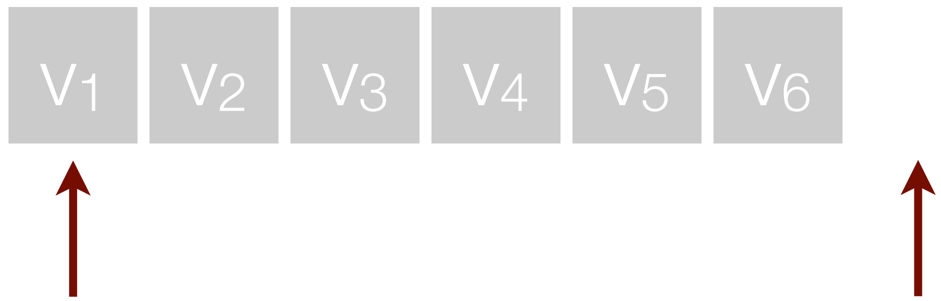
Modules at an example: queue

Queue, a **first-in first-out** (FIFO) data structure.



Modules at an example: queue

Queue, a **first-in first-out** (FIFO) data structure.



dequeue,
yields a queue with
this element removed

Modules at an example: queue

Queue, a **first-in first-out** (FIFO) data structure.

dequeue,
yields a queue with
this element removed

enqueue v_7,
yields a queue with
element added at end

Modules at an example: queue

Queue, a **first-in first-out** (FIFO) data structure.

dequeue,
yields a queue with
this element removed

enqueue v_7,
yields a queue with
element added at end

We can describe queue abstractly by specifying a new queue type, along with operations on that type.

Modules at an example: queue

Queue, a **first-in first-out** (FIFO) data structure.

dequeue,
yields a queue with
this element removed

enqueue v_7,
yields a queue with
element added at end

We can describe queue abstractly by specifying a new queue type, along with operations on that type.

That's a **signature**.

Modules at an example: queue

Queue, a **first-in first-out** (FIFO) data structure.

dequeue,
yields a queue with
this element removed

enqueue v_7,
yields a queue with
element added at end

- We can describe queue abstractly by specifying a new queue type, along with operations on that type.
  - That's a **signature**.
- Now we implement it in a **structure**.

Queue signature

Queue signature

```
signature QUEUE =
sig










end
```

Queue signature

```
signature QUEUE =
sig



end
```

Queue signature

name

```
signature QUEUE =
sig



end
```

Queue signature

name

```
signature QUEUE =
sig

end
```

Queue signature

name

```
signature QUEUE =
sig

end
```

specification

Queue signature

```
signature QUEUE =
sig










end
```

Queue signature

```
signature QUEUE =
sig
  type 'a queue              (* abstract type *)




end
```

Queue signature

```
signature QUEUE =
sig
  type 'a queue          (* abstract type *)





end
```

Queue signature

```
signature QUEUE =
sig
  type 'a queue            (* abstract type *)
```

type name

```
end
```

Queue signature

```
signature QUEUE =
sig
  type 'a queue            (* abstract type *)








end
```

type name

Queue signature

```
signature QUEUE =
sig
  type 'a queue          (* abstract type *)




end
```

polymorphic

type name

Queue signature

Postulate existence of an alpha queue

```
signature QUEUE =
sig
  type 'a queue          (* abstract type *)
```

polymorphic

type name

```
end
```

Queue signature

```
signature QUEUE =
sig
  type 'a queue          (* abstract type *)



end
```

Postulate existence of an alpha queue

polymorphic

type name

No details on how represented.

Queue signature

```
signature QUEUE =
sig
  type 'a queue          (* abstract type *)



end
```

Postulate existence of an alpha queue

Queue signature

```
signature QUEUE =
sig
  type 'a queue          (* abstract type *)



end
```

Postulate existence of an alpha queue

Operations on an alpha queue

Queue signature

```
signature QUEUE =
sig
  type 'a queue            (* abstract type *)




end
```

Queue signature

```
signature QUEUE =
sig
  type 'a queue         (* abstract type *)
  val empty : 'a queue





end
```

Queue signature

```
signature QUEUE =
sig
  type 'a queue            (* abstract type *)
  val empty : 'a queue
  val enq : 'a queue * 'a -> 'a queue



end
```

Queue signature

```
signature QUEUE =
sig
  type 'a queue          (* abstract type *)
  val empty : 'a queue
  val enq : 'a queue * 'a -> 'a queue
  val null : 'a queue -> bool




end
```

Queue signature

```
signature QUEUE =
sig
  type 'a queue          (* abstract type *)
  val empty : 'a queue
  val enq : 'a queue * 'a -> 'a queue
  val null : 'a queue -> bool
  exception Empty
  (* deq (q) raises Empty if q is empty *)
  val deq : 'a queue -> 'a * 'a queue
end
```

Let’s implement QUEUE in a structure

Let’s implement QUEUE in a structure

Implementation I: **represent** alpha queue as a single list.

Let's implement QUEUE in a structure

Implementation I: **represent** alpha queue as a single list.

Let’s implement QUEUE in a structure

Implementation I: **represent** alpha queue as a single list.

List represents alpha queue elements in arrival order.

Single list representation of alpha queue

Single list representation of alpha queue

```
signature QUEUE =
sig
 type 'a queue        (* abstract type *)
 val empty : 'a queue
 val enq : 'a queue * 'a -> 'a queue
 val null : 'a queue -> bool
 exception Empty
 val deq : 'a queue -> 'a * 'a queue
end
```

Single list representation of alpha queue

```
signature QUEUE =
sig
 type 'a queue         (* abstract type *)
 val empty : 'a queue
 val enq : 'a queue * 'a -> 'a queue
 val null : 'a queue -> bool
 exception Empty
 val deq : 'a queue -> 'a * 'a queue
end
```

```
structure Queue1 : QUEUE =
struct




end
```

Single list representation of alpha queue

```
signature QUEUE =
sig
 type 'a queue         (* abstract type *)
 val empty : 'a queue
 val enq : 'a queue * 'a -> 'a queue
 val null : 'a queue -> bool
 exception Empty
 val deq : 'a queue -> 'a * 'a queue
end
```

```
structure Queue1 : QUEUE =
struct



end
```

Single list representation of alpha queue

```
signature QUEUE =
sig
 type 'a queue         (* abstract type *)
 val empty : 'a queue
 val enq : 'a queue * 'a -> 'a queue
 val null : 'a queue -> bool
 exception Empty
 val deq : 'a queue -> 'a * 'a queue
end
```

```
structure Queue1 : QUEUE =
struct



end
```

name

Single list representation of alpha queue

```
signature QUEUE =
sig
 type 'a queue          (* abstract type *)
 val empty : 'a queue
 val enq : 'a queue * 'a -> 'a queue
 val null : 'a queue -> bool
 exception Empty
 val deq : 'a queue -> 'a * 'a queue
end
```

```
structure Queue1 : QUEUE =
struct

end
```

name

Single list representation of alpha queue

```
signature QUEUE =
sig
 type 'a queue        (* abstract type *)
 val empty : 'a queue
 val enq : 'a queue * 'a -> 'a queue
 val null : 'a queue -> bool
 exception Empty
 val deq : 'a queue -> 'a * 'a queue
end
```

Ascribe signature **QUEUE** to structure **Queue1**

```
structure Queue1 : QUEUE =
struct



end
```

name

Single list representation of alpha queue

```
signature QUEUE =
sig
 type 'a queue        (* abstract type *)
 val empty : 'a queue
 val enq : 'a queue * 'a -> 'a queue
 val null : 'a queue -> bool
 exception Empty
 val deq : 'a queue -> 'a * 'a queue
end
```

```
structure Queue1 : QUEUE =
struct

end
```

Ascribe signature **QUEUE** to structure **Queue1**

name

Ascribe means:
The structure provides all the items specified in the signature. (It may contain additional items, e.g., helper functions, which will not be visible outside the structure, e.g., to a client.)

Single list representation of alpha queue

```
signature QUEUE =
sig
 type 'a queue         (* abstract type *)
 val empty : 'a queue
 val enq : 'a queue * 'a -> 'a queue
 val null : 'a queue -> bool
 exception Empty
 val deq : 'a queue -> 'a * 'a queue
end
```

```
structure Queue1 : QUEUE =
struct




end
```

Single list representation of alpha queue

```
signature QUEUE =
sig
 type 'a queue        (* abstract type *)
 val empty : 'a queue
 val enq : 'a queue * 'a -> 'a queue
 val null : 'a queue -> bool
 exception Empty
 val deq : 'a queue -> 'a * 'a queue
end
```

```
structure Queue1 : QUEUE =
struct

end
```

Single list representation of alpha queue

```
signature QUEUE =
sig
 type 'a queue         (* abstract type *)
 val empty : 'a queue
 val enq : 'a queue * 'a -> 'a queue
 val null : 'a queue -> bool
 exception Empty
 val deq : 'a queue -> 'a * 'a queue
end
```

```
structure Queue1 : QUEUE =
struct

end
```

declarations

Single list representation of alpha queue

```
structure Queue1 : QUEUE =
struct





end
```

```
signature QUEUE =
sig
 type 'a queue        (* abstract type *)
 val empty : 'a queue
 val enq : 'a queue * 'a -> 'a queue
 val null : 'a queue -> bool
 exception Empty
 val deq : 'a queue -> 'a * 'a queue
end
```

Single list representation of alpha queue

```
signature QUEUE =
sig
 type 'a queue        (* abstract type *)
 val empty : 'a queue
 val enq : 'a queue * 'a -> 'a queue
 val null : 'a queue -> bool
 exception Empty
 val deq : 'a queue -> 'a * 'a queue
end
```

```
structure Queue1 : QUEUE =
struct
  type 'a queue = 'a list



end
```

Single list representation of alpha queue

```
structure Queue1 : QUEUE =
struct
  type 'a queue = 'a list









end
```

```
signature QUEUE =
sig
 type 'a queue        (* abstract type *)
 val empty : 'a queue
 val enq : 'a queue * 'a -> 'a queue
 val null : 'a queue -> bool
 exception Empty
 val deq : 'a queue -> 'a * 'a queue
end
```

Single list representation of alpha queue

```
signature QUEUE =
sig
 type 'a queue         (* abstract type *)
 val empty : 'a queue
 val enq : 'a queue * 'a -> 'a queue
 val null : 'a queue -> bool
 exception Empty
 val deq : 'a queue -> 'a * 'a queue
end
```

```
structure Queue1 : QUEUE =
struct
  type 'a queue = 'a list
```

representation type

```
end
```

Single list representation of alpha queue

```
signature QUEUE =
sig
 type 'a queue         (* abstract type *)
 val empty : 'a queue
 val enq : 'a queue * 'a -> 'a queue
 val null : 'a queue -> bool
 exception Empty
 val deq : 'a queue -> 'a * 'a queue
end
```

```
structure Queue1 : QUEUE =
struct
  type 'a queue = 'a list





end
```

representation type

Single list representation of alpha queue

```
signature QUEUE =
sig
 type 'a queue         (* abstract type *)
 val empty : 'a queue
 val enq : 'a queue * 'a -> 'a queue
 val null : 'a queue -> bool
 exception Empty
 val deq : 'a queue -> 'a * 'a queue
end
```

transparent ascription

```
structure Queue1 : QUEUE =
struct
  type 'a queue = 'a list



end
```

representation type

Single list representation of alpha queue

```
signature QUEUE =
sig
 type 'a queue         (* abstract type *)
 val empty : 'a queue
 val enq : 'a queue * 'a -> 'a queue
 val null : 'a queue -> bool
 exception Empty
 val deq : 'a queue -> 'a * 'a queue
end
```

transparent ascription

```
structure Queue1 : QUEUE =
struct
  type 'a queue = 'a list
```

representation type

Transparent ascription means:
The representation type of the abstract type alpha queue will be visible outside the structure, e.g., to the client.

```
end
```

Single list representation of alpha queue

```
signature QUEUE =
sig
 type 'a queue        (* abstract type *)
 val empty : 'a queue
 val enq : 'a queue * 'a -> 'a queue
 val null : 'a queue -> bool
 exception Empty
 val deq : 'a queue -> 'a * 'a queue
end
```

```
structure Queue1 : QUEUE =
struct
  type 'a queue = 'a list






end
```

Single list representation of alpha queue

```
signature QUEUE =
sig
 type 'a queue        (* abstract type *)
 val empty : 'a queue
 val enq : 'a queue * 'a -> 'a queue
 val null : 'a queue -> bool
 exception Empty
 val deq : 'a queue -> 'a * 'a queue
end
```

```
structure Queue1 : QUEUE =
struct
  type 'a queue = 'a list

  val empty = nil








end
```

Single list representation of alpha queue

```
signature QUEUE =
sig
 type 'a queue         (* abstract type *)
 val empty : 'a queue
 val enq : 'a queue * 'a -> 'a queue
 val null : 'a queue -> bool
 exception Empty
 val deq : 'a queue -> 'a * 'a queue
end
```

```
structure Queue1 : QUEUE =
struct
  type 'a queue = 'a list

  val empty = nil

  fun enq (q,x) = q @ [x]






end
```

Single list representation of alpha queue

```
signature QUEUE =
sig
 type 'a queue        (* abstract type *)
 val empty : 'a queue
 val enq : 'a queue * 'a -> 'a queue
 val null : 'a queue -> bool
 exception Empty
 val deq : 'a queue -> 'a * 'a queue
end
```

```
structure Queue1 : QUEUE =
struct
  type 'a queue = 'a list

  val empty = nil

  fun enq (q,x) = q @ [x]

  val null = List.null




end
```

Single list representation of alpha queue

```
signature QUEUE =
sig
 type 'a queue          (* abstract type *)
 val empty : 'a queue
 val enq : 'a queue * 'a -> 'a queue
 val null : 'a queue -> bool
 exception Empty
 val deq : 'a queue -> 'a * 'a queue
end
```

```
structure Queue1 : QUEUE =
struct
  type 'a queue = 'a list

  val empty = nil

  fun enq (q,x) = q @ [x]

  val null = List.null

  exception Empty

 fun deq (x::q) = (x,q)
    | deq (nil) = raise Empty

end
```

Single list representation of alpha queue

```
structure Queue1 : QUEUE =
struct
  type 'a queue = 'a list
  val empty = nil
  fun enq (q,x) = q @ [x]
  val null = List.null
  exception Empty
 fun deq (x::q) = (x,q)
    | deq (nil) = raise Empty
end
```

```
signature QUEUE =
sig
 type 'a queue        (* abstract type *)
 val empty : 'a queue
 val enq : 'a queue * 'a -> 'a queue
 val null : 'a queue -> bool
 exception Empty
 val deq : 'a queue -> 'a * 'a queue
end
```

Single list representation of alpha queue

```
structure Queue1 : QUEUE =
struct
  type 'a queue = 'a list
  val empty = nil
  fun enq (q,x) = q @ [x]
  val null = List.null
  exception Empty
 fun deq (x::q) = (x,q)
    | deq (nil) = raise Empty
end
```

```
signature QUEUE =
sig
 type 'a queue        (* abstract type *)
 val empty : 'a queue
 val enq : 'a queue * 'a -> 'a queue
 val null : 'a queue -> bool
 exception Empty
 val deq : 'a queue -> 'a * 'a queue
end
```

Let’s interact with **`Queue1`**:

Single list representation of alpha queue

```
structure Queue1 : QUEUE =
struct
  type 'a queue = 'a list
  val empty = nil
  fun enq (q,x) = q @ [x]
  val null = List.null
  exception Empty
 fun deq (x::q) = (x,q)
    | deq (nil) = raise Empty
end
```

```
signature QUEUE =
sig
 type 'a queue        (* abstract type *)
 val empty : 'a queue
 val enq : 'a queue * 'a -> 'a queue
 val null : 'a queue -> bool
 exception Empty
 val deq : 'a queue -> 'a * 'a queue
end
```

Let's interact with **Queue1**:

```
val q = Queue1.enq(Queue1.enq(Queue1.empty, 1), 2)
```

Single list representation of alpha queue

```
structure Queue1 : QUEUE =
struct
  type 'a queue = 'a list
  val empty = nil
  fun enq (q,x) = q @ [x]
  val null = List.null
  exception Empty
 fun deq (x::q) = (x,q)
    | deq (nil) = raise Empty
end
```

```
signature QUEUE =
sig
 type 'a queue        (* abstract type *)
 val empty : 'a queue
 val enq : 'a queue * 'a -> 'a queue
 val null : 'a queue -> bool
 exception Empty
 val deq : 'a queue -> 'a * 'a queue
end
```

Let's interact with **`Queue1`**:

```
val q = Queue1.enq(Queue1.enq(Queue1.empty, 1), 2)
```

What is the type of **`q`**?

Single list representation of alpha queue

```
structure Queue1 : QUEUE =
struct
  type 'a queue = 'a list
  val empty = nil
  fun enq (q,x) = q @ [x]
  val null = List.null
  exception Empty
 fun deq (x::q) = (x,q)
    | deq (nil) = raise Empty
end
```

```
signature QUEUE =
sig
 type 'a queue        (* abstract type *)
 val empty : 'a queue
 val enq : 'a queue * 'a -> 'a queue
 val null : 'a queue -> bool
 exception Empty
 val deq : 'a queue -> 'a * 'a queue
end
```

Let's interact with **Queue1**:

```
val q = Queue1.enq(Queue1.enq(Queue1.empty, 1), 2)
```

What is the type of **q**?

```
int Queue1.queue
```

Single list representation of alpha queue

Let’s interact with **`Queue1`**:

```
val q = Queue1.enq(Queue1.enq(Queue1.empty, 1), 2)
```

What is the type of **`q`**?

```
int Queue1.queue
```

Single list representation of alpha queue

Let's interact with **`Queue1`**:

```
val q = Queue1.enq(Queue1.enq(Queue1.empty, 1), 2)
```

What is the type of **`q`**?

```
int Queue1.queue
```

Why? Because:

First, the signature specifies that queues have type **`‘a queue`**, with **`‘a`** representing the element type. That is **`int`** here.

Second, we have implemented queues using a structure called **`Queue1`**. The type is defined inside the structure, so the type has the qualified name **`‘a Queue1.queue`**, here with **`‘a`** instantiated with **`int`**.

Single list representation of alpha queue

Let's interact with **`Queue1`**:

```
val q = Queue1.enq(Queue1.enq(Queue1.empty, 1), 2)
```

What is the type of **`q`**?

```
int Queue1.queue
```

Single list representation of alpha queue

Let's interact with **`Queue1`**:

```
val q = Queue1.enq(Queue1.enq(Queue1.empty, 1), 2)
```

What is the type of **`q`**?

```
int Queue1.queue
```

Also, **`q`** will be bound to:

Single list representation of alpha queue

Let's interact with **`Queue1`**:

```
val q = Queue1.enq(Queue1.enq(Queue1.empty, 1), 2)
```

What is the type of **`q`**?

```
int Queue1.queue
```

Also, **`q`** will be bound to:

```
[1, 2]
```

Single list representation of alpha queue

Let's interact with **`Queue1`**:

```
val q = Queue1.enq(Queue1.enq(Queue1.empty, 1), 2)
```

What is the type of **`q`**?

```
int Queue1.queue
```

Also, **`q`** will be bound to:

```
[1, 2]
```

We can see the representation type list because of transparent ascription (and because list is defined in the Basis Library and thus in the top-level scope).

Single list representation of alpha queue

Let's interact with `Queue1`:

```
val q = Queue1.enq(Queue1.enq(Queue1.empty, 1), 2)
```

What is the type of `q`?

```
int Queue1.queue
```

Single list representation of alpha queue

Let's interact with **`Queue1`**:

```
val q = Queue1.enq(Queue1.enq(Queue1.empty, 1), 2)
```

What is the type of **`q`**?

```
int Queue1.queue
```

Next, let's consider:

Single list representation of alpha queue

Let's interact with **`Queue1`**:

```
val q = Queue1.enq(Queue1.enq(Queue1.empty, 1), 2)
```

What is the type of **`q`**?

```
int Queue1.queue
```

Next, let's consider:

```
val (a, b) = Queue1.deq q
val (c, _) = Queue1.deq q
val (d, _) = Queue1.deq b
```

Single list representation of alpha queue

Let's interact with **`Queue1`**:

```
val q = Queue1.enq(Queue1.enq(Queue1.empty, 1), 2)
```

What is the type of **`q`**?

```
int Queue1.queue
```

Next, let's consider:

```
val (a, b) = Queue1.deq q
val (c, _) = Queue1.deq q
val (d, _) = Queue1.deq b
```

What are the bindings for **`a`**, **`c`**, and **`d`**?

Single list representation of alpha queue

Let's interact with **`Queue1`**:

```
val q = Queue1.enq(Queue1.enq(Queue1.empty, 1), 2)
```

What is the type of **`q`**?

```
int Queue1.queue
```

Next, let's consider:

```
val (a, b) = Queue1.deq q
val (c, _) = Queue1.deq q
val (d, _) = Queue1.deq b
```

What are the bindings for **`a`**, **`c`**, and **`d`**?

```
[1/a, 1/c, 2/d]
```

Single list representation of alpha queue

Let's interact with **Queue1**:

```
val q = Queue1.enq(Queue1.enq(Queue1.empty, 1), 2)
```

What is the type of **q**?

```
int Queue1.queue
```

Next, let's consider:

```
val (a, b) = Queue1.deq q
val (c, _) = Queue1.deq q
val (d, _) = Queue1.deq b
```

What are the bindings for **a**, **c**, and **d**?

```
[1/a, 1/c, 2/d]
```

no mutation!

Single list representation of alpha queue

Single list representation of alpha queue

How long does enqueuing take?

Single list representation of alpha queue

How long does enqueing take?

```
fun enq (q,x) = q @ [x]
```

Single list representation of alpha queue

How long does enqueuing take?

```
fun enq (q,x) = q @ [x]
```

→ **O**(n), with n the number of elements in **q**.

Single list representation of alpha queue

How long does enqueuing take?

```
fun enq (q,x) = q @ [x]
```

O(n), with n the number of elements in **q**.

Can we improve that?

Single list representation of alpha queue

How long does enqueuing take?

```
fun enq (q,x) = q @ [x]
```

- **O**(n), with n the number of elements in **q**.
- Can we improve that?
  - Yes, let's choose a different representation type!

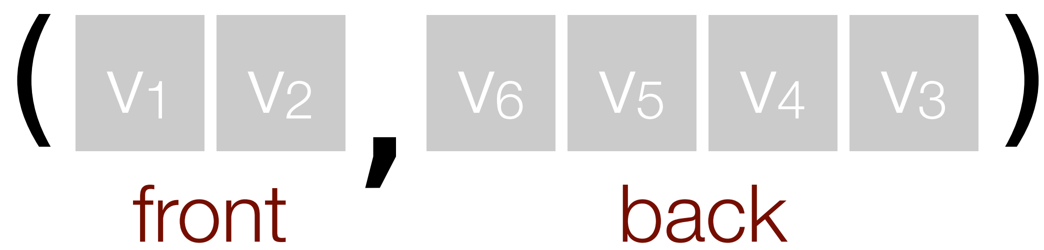
Pair of list representation of alpha queue

Pair of list representation of alpha queue

Implementation II: **represent** alpha queue as a pair of list.

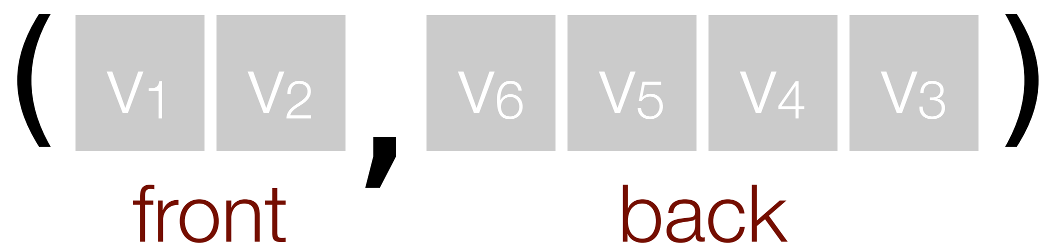
Pair of list representation of alpha queue

Implementation II: **represent** alpha queue as a pair of list.



Pair of list representation of alpha queue

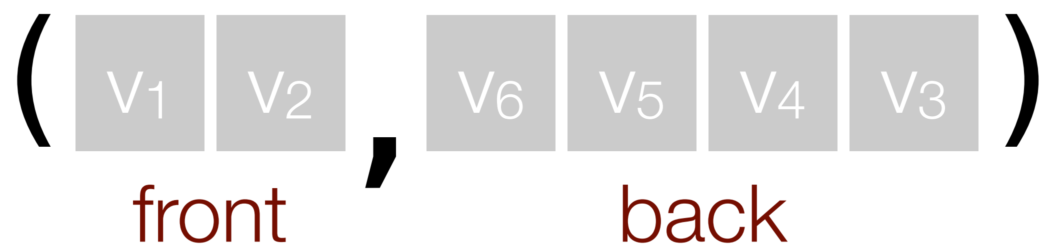
Implementation II: **represent** alpha queue as a pair of list.



Abstraction function:

Pair of list representation of alpha queue

Implementation II: **represent** alpha queue as a pair of list.



Abstraction function:

```
front @ (rev back)
```

Pair of list representation of alpha queue

Implementation II: **represent** alpha queue as a pair of list.

Abstraction function:

```
front @ (rev back)
```

Represents alpha queue elements in arrival order.

Pair of list representation of alpha queue

Pair of list representation of alpha queue

```
signature QUEUE =
sig
 type 'a queue        (* abstract type *)
 val empty : 'a queue
 val enq : 'a queue * 'a -> 'a queue
 val null : 'a queue -> bool
 exception Empty
 val deq : 'a queue -> 'a * 'a queue
end
```

Pair of list representation of alpha queue

```
structure Queue2 :> QUEUE =
struct
  type 'a queue =
    'a list * 'a list
```

```
signature QUEUE =
sig
 type 'a queue        (* abstract type *)
 val empty : 'a queue
 val enq : 'a queue * 'a -> 'a queue
 val null : 'a queue -> bool
 exception Empty
 val deq : 'a queue -> 'a * 'a queue
end
```

```
end
```

Pair of list representation of alpha queue

```
structure Queue2 :> QUEUE =
struct
  type 'a queue =
    'a list * 'a list









end
```

```
signature QUEUE =
sig
 type 'a queue        (* abstract type *)
 val empty : 'a queue
 val enq : 'a queue * 'a -> 'a queue
 val null : 'a queue -> bool
 exception Empty
 val deq : 'a queue -> 'a * 'a queue
end
```

Pair of list representation of alpha queue

opaque ascription

```
structure Queue2 :> QUEUE =
struct
  type 'a queue =
    'a list * 'a list




end
```

```
signature QUEUE =
sig
 type 'a queue        (* abstract type *)
 val empty : 'a queue
 val enq : 'a queue * 'a -> 'a queue
 val null : 'a queue -> bool
 exception Empty
 val deq : 'a queue -> 'a * 'a queue
end
```

Pair of list representation of alpha queue

opaque ascription

```
structure Queue2 :> QUEUE =
struct
  type 'a queue =
    'a list * 'a list




end
```

```
signature QUEUE =
sig
 type 'a queue        (* abstract type *)
 val empty : 'a queue
 val enq : 'a queue * 'a -> 'a queue
 val null : 'a queue -> bool
 exception Empty
 val deq : 'a queue -> 'a * 'a queue
end
```

Pair of list representation of alpha queue

opaque ascription

```
structure Queue2 :> QUEUE =
struct
   type 'a queue =
     'a list * 'a list
```

representation type

```
signature QUEUE =
sig
 type 'a queue          (* abstract type *)
 val empty : 'a queue
 val enq : 'a queue * 'a -> 'a queue
 val null : 'a queue -> bool
 exception Empty
 val deq : 'a queue -> 'a * 'a queue
end
```

```
end
```

Pair of list representation of alpha queue

opaque ascription

```
structure Queue2 :> QUEUE =
struct
   type 'a queue =
     'a list * 'a list
```

representation type

```
signature QUEUE =
sig
 type 'a queue        (* abstract type *)
 val empty : 'a queue
 val enq : 'a queue * 'a -> 'a queue
 val null : 'a queue -> bool
 exception Empty
 val deq : 'a queue -> 'a * 'a queue
end
```

Opaque ascription means that the representation type of the abstract type alpha queue will be hidden outside the structure, e.g., from the client. ML will only print a dash.

```
end
```

Pair of list representation of alpha queue

opaque ascription

```
structure Queue2 :> QUEUE =
struct
   type 'a queue =
     'a list * 'a list
```

representation type

```
signature QUEUE =
sig
 type 'a queue          (* abstract type *)
 val empty : 'a queue
 val enq : 'a queue * 'a -> 'a queue
 val null : 'a queue -> bool
 exception Empty
 val deq : 'a queue -> 'a * 'a queue
end
```

Opaque ascription means that the representation type of the abstract type alpha queue will be hidden outside the structure, e.g., from the client. ML will only print a dash.

→ Ensures information hiding and integrity of data structure.

```
end
```

Pair of list representation of alpha queue

opaque ascription

```
structure Queue2 :> QUEUE =
struct
   type 'a queue =
    'a list * 'a list
```

representation type

```
signature QUEUE =
sig
 type 'a queue          (* abstract type *)
 val empty : 'a queue
 val enq : 'a queue * 'a -> 'a queue
 val null : 'a queue -> bool
 exception Empty
 val deq : 'a queue -> 'a * 'a queue
end
```

Opaque ascription means that the representation type of the abstract type alpha queue will be hidden outside the structure, e.g., from the client. ML will only print a dash.

→ Ensures information hiding and integrity of data structure.

→ Not guaranteed with transparent ascription!

```
end
```

Pair of list representation of alpha queue

```
structure Queue2 :> QUEUE =
struct
  type 'a queue =
    'a list * 'a list
```

```
signature QUEUE =
sig
 type 'a queue        (* abstract type *)
 val empty : 'a queue
 val enq : 'a queue * 'a -> 'a queue
 val null : 'a queue -> bool
 exception Empty
 val deq : 'a queue -> 'a * 'a queue
end
```

```
end
```

Pair of list representation of alpha queue

```
structure Queue2 :> QUEUE =
struct
  type 'a queue =
    'a list * 'a list

  val empty = (nil, nil)

  fun enq ((front, back), x) = (front, x::back)




end
```

```
signature QUEUE =
sig
 type 'a queue          (* abstract type *)
 val empty : 'a queue
 val enq : 'a queue * 'a -> 'a queue
 val null : 'a queue -> bool
 exception Empty
 val deq : 'a queue -> 'a * 'a queue
end
```

Pair of list representation of alpha queue

```
structure Queue2 :> QUEUE =
struct
  type 'a queue =
     'a list * 'a list

  val empty = (nil, nil)

  fun enq ((front, back), x) = (front, x::back)

end
```

```
signature QUEUE =
sig
 type 'a queue          (* abstract type *)
 val empty : 'a queue
 val enq : 'a queue * 'a -> 'a queue
 val null : 'a queue -> bool
 exception Empty
 val deq : 'a queue -> 'a * 'a queue
end
```

Pair of list representation of alpha queue

```
signature QUEUE =
sig
 type 'a queue          (* abstract type *)
 val empty : 'a queue
 val enq : 'a queue * 'a -> 'a queue
 val null : 'a queue -> bool
 exception Empty
 val deq : 'a queue -> 'a * 'a queue
end
```

```
structure Queue2 :> QUEUE =
struct
  type 'a queue =
    'a list * 'a list

  val empty = (nil, nil)

  fun enq ((front, back), x) = (front, x::back)
```

Satisfies requirement that `f @ (rev(x::b))` constitutes queue elements in arrival order.

```
end
```

Pair of list representation of alpha queue

```
structure Queue2 :> QUEUE =
struct
  type 'a queue =
     'a list * 'a list

  val empty = (nil, nil)

  fun enq ((front, back), x) = (front, x::back)






end
```

```
signature QUEUE =
sig
 type 'a queue        (* abstract type *)
 val empty : 'a queue
 val enq : 'a queue * 'a -> 'a queue
 val null : 'a queue -> bool
 exception Empty
 val deq : 'a queue -> 'a * 'a queue
end
```

Pair of list representation of alpha queue

```
structure Queue2 :> QUEUE =
struct
  type 'a queue =
     'a list * 'a list

  val empty = (nil, nil)

  fun enq ((front, back), x) = (front, x::back)

  fun null (nil,nil) = true
    | null _ = false






end
```

```
signature QUEUE =
sig
 type 'a queue         (* abstract type *)
 val empty : 'a queue
 val enq : 'a queue * 'a -> 'a queue
 val null : 'a queue -> bool
 exception Empty
 val deq : 'a queue -> 'a * 'a queue
end
```

Pair of list representation of alpha queue

```
signature QUEUE =
sig
 type 'a queue        (* abstract type *)
 val empty : 'a queue
 val enq : 'a queue * 'a -> 'a queue
 val null : 'a queue -> bool
 exception Empty
 val deq : 'a queue -> 'a * 'a queue
end
```

```
structure Queue2 :> QUEUE =
struct
  type 'a queue =
     'a list * 'a list

  val empty = (nil, nil)

  fun enq ((front, back), x) = (front, x::back)

  fun null (nil,nil) = true
    | null _ = false

  exception Empty

  fun deq (x::front, back) = (x, (front, back))
    | deq (nil, nil) = raise Empty
    | deq (nil, back) = deq(rev back, nil)

end
```

Pair of list representation of alpha queue

Pair of list representation of alpha queue

Let's interact with `Queue2`:

Pair of list representation of alpha queue

Let's interact with `Queue2`:

```
val q = Queue2.enq(Queue2.enq(Queue2.empty, 1), 2)
```

Pair of list representation of alpha queue

Let’s interact with **`Queue2`**:

```
val q = Queue2.enq(Queue2.enq(Queue2.empty, 1), 2)
```

What is the type of **`q`**?

Pair of list representation of alpha queue

Let's interact with **`Queue2`**:

```
val q = Queue2.enq(Queue2.enq(Queue2.empty, 1), 2)
```

What is the type of **`q`**?

```
int Queue2.queue
```

Pair of list representation of alpha queue

Let's interact with `Queue2`:

```
val q = Queue2.enq(Queue2.enq(Queue2.empty, 1), 2)
```

What is the type of `q`?

```
int Queue2.queue
```

Why? Because:

First, the signature specifies that queues have type `‘a queue`, with `‘a` representing the element type. That is `int` here.

Second, we have implemented queues using a structure called `Queue2`. The type is defined inside the structure, so the type has the qualified name `‘a Queue2.queue`, here with `‘a` instantiated with `int`.

Pair of list representation of alpha queue

Let's interact with **`Queue2`**:

```
val q = Queue2.enq(Queue2.enq(Queue2.empty, 1), 2)
```

What is the type of **`q`**?

```
int Queue2.queue
```

Pair of list representation of alpha queue

Let's interact with `Queue2`:

```
val q = Queue2.enq(Queue2.enq(Queue2.empty, 1), 2)
```

What is the type of `q`?

```
int Queue2.queue
```

However, the binding for `q` will not be revealed because of opaque ascription:

Pair of list representation of alpha queue

Let's interact with **`Queue2`**:

```
val q = Queue2.enq(Queue2.enq(Queue2.empty, 1), 2)
```

What is the type of **`q`**?

```
int Queue2.queue
```

However, the binding for **`q`** will not be revealed because of opaque ascription:

```
val q = - : int Queue2.queue
```

Pair of list representation of alpha queue

Let's interact with **`Queue2`**:

```
val q = Queue2.enq(Queue2.enq(Queue2.empty, 1), 2)
```

What is the type of **`q`**?

```
int Queue2.queue
```

Pair of list representation of alpha queue

Let's interact with **`Queue2`**:

```
val q = Queue2.enq(Queue2.enq(Queue2.empty, 1), 2)
```

What is the type of **`q`**?

```
int Queue2.queue
```

Next, let's consider:

Pair of list representation of alpha queue

Let's interact with **`Queue2`**:

```
val q = Queue2.enq(Queue2.enq(Queue2.empty, 1), 2)
```

What is the type of **`q`**?

```
int Queue2.queue
```

Next, let's consider:

```
val (a, b) = Queue2deq q
val (c, _) = Queue2.deq q
val (d, _) = Queue2.deq b
```

Pair of list representation of alpha queue

Let's interact with **`Queue2`**:

```
val q = Queue2.enq(Queue2.enq(Queue2.empty, 1), 2)
```

What is the type of **`q`**?

```
int Queue2.queue
```

Next, let's consider:

```
val (a, b) = Queue2deq q
val (c, _) = Queue2.deq q
val (d, _) = Queue2.deq b
```

What are the bindings for **`a`**, **`c`**, and **`d`**?

Pair of list representation of alpha queue

Let's interact with **Queue2**:

```
val q = Queue2.enq(Queue2.enq(Queue2.empty, 1), 2)
```

What is the type of **q**?

```
int Queue2.queue
```

Next, let's consider:

```
val (a, b) = Queue2deq q
val (c, _) = Queue2.deq q
val (d, _) = Queue2.deq b
```

What are the bindings for **a**, **c**, and **d**?

```
[1/a, 1/c, 2/d]
```

Pair of list representation of alpha queue

Pair of list representation of alpha queue

How long does enqueuing take?

Pair of list representation of alpha queue

How long does enqueing take?

```
fun enq ((front, back), x) = (front, x::back)
```

Pair of list representation of alpha queue

How long does enqueuing take?

```
fun enq ((front, back), x) = (front, x::back)
```

O(1)!

Pair of list representation of alpha queue

How long does enqueuing take?

```
fun enq ((front, back), x) = (front, x::back)
```

- **O**(1)!
- Dequeuing can now take **O**(n) time.

Pair of list representation of alpha queue

How long does enqueuing take?

```
fun enq ((front, back), x) = (front, x::back)
```

- **O**(1)!
- Dequeuing can now take **O**(n) time.
- However, enqueuing and dequeuing n items will on take **O**(n) time total, so on average it is **O**(1).

Pair of list representation of alpha queue

How long does enqueuing take?

```
fun enq ((front, back), x) = (front, x::back)
```

O(1)!

Dequeuing can now take **O**(n) time.

However, enqueuing and dequeuing n items will on take **O**(n) time total, so on average it is **O**(1).

The amortized cost is **O**(1).

Comparing the two implementations

Comparing the two implementations

Operation:	Queue1:	Queue2:

Comparing the two implementations

Operation:	Queue1:	Queue2:
`empty`	`[]`	`([], [])`

Comparing the two implementations

Operation:	Queue1:	Queue2:
`empty`	`[]`	`([], [])`
`enq 1`	`[1]`	`([], [1])`

Comparing the two implementations

Operation:	Queue1:	Queue2:
`empty`	`[]`	`([], [])`
`enq 1`	`[1]`	`([], [1])`
`enq 2`	`[1,2]`	`([], [2,1])`

Comparing the two implementations

Operation:	Queue1:	Queue2:
`empty`	`[]`	`([], [])`
`enq 1`	`[1]`	`([], [1])`
`enq 2`	`[1,2]`	`([], [2,1])`
`deq`	`[2]`	

Comparing the two implementations

Operation:	Queue1:	Queue2:
`empty`	`[]`	`([], [])`
`enq 1`	`[1]`	`([], [1])`
`enq 2`	`[1,2]`	`([], [2,1])`
`deq`	`[2]`	*briefly this:* `([1,2], [])` *then this:* `([2], [])`

Comparing the two implementations

Operation:	Queue1:	Queue2:
`empty`	`[]`	`([], [])`
`enq 1`	`[1]`	`([], [1])`
`enq 2`	`[1,2]`	`([], [2,1])`
`deq`	`[2]`	`([1,2], [])`
		`([2], [])`
`enq 3`	`[2,3]`	`([2], [3])`

Comparing the two implementations

Operation:	Queue1:	Queue2:
`empty`	`[]`	`([], [])`
`enq 1`	`[1]`	`([], [1])`
`enq 2`	`[1,2]`	`([], [2,1])`
		briefly this:
`deq`	`[2]`	`([1,2], [])`
		`([2], [])`
`enq 3`	`[2,3]`	`([2], [3])`

Comparing the two implementations

Operation:	Queue1:	Queue2:
`empty`	`[]`	`([], [])`
`enq 1`	`[1]`	`([], [1])`
`enq 2`	`[1,2]`	`([], [2,1])`
		briefly this:
`deq`	`[2]`	`([1,2], [])`
		then this:
		`([2], [])`
`enq 3`	`[2,3]`	`([2], [3])`

Another example: dictionary

Another example: dictionary

A dictionary is a collection of pairs of the form

Another example: dictionary

A dictionary is a collection of pairs of the form

```
(key, value)
```

Another example: dictionary

A dictionary is a collection of pairs of the form

```
(key, value)
```

where all keys must be unique within a dictionary.

Another example: dictionary

A dictionary is a collection of pairs of the form

```
(key, value)
```

where all keys must be unique within a dictionary.

```
signature DICT =
sig
  type key = string                     (* concrete type *)
  type 'a entry = key * 'a              (* concrete type *)
  type 'a dict                          (* abstract type *)



end
```

Another example: dictionary

A dictionary is a collection of pairs of the form

```
(key, value)
```

where all keys must be unique within a dictionary.

```
signature DICT =
sig
  type key = string                       (* concrete type *)
  type 'a entry = key * 'a                (* concrete type *)
  type 'a dict                            (* abstract type *)



end
```

Another example: dictionary

A dictionary is a collection of pairs of the form

```
(key, value)
```

where all keys must be unique within a dictionary.

Let’s use strings as keys for now

```
signature DICT =
sig
  type key = string                      (* concrete type *)
  type 'a entry = key * 'a               (* concrete type *)
  type 'a dict                           (* abstract type *)


end
```

Another example: dictionary

A dictionary is a collection of pairs of the form

```
(key, value)
```

where all keys must be unique within a dictionary.

Let's use strings as keys for now

```
signature DICT =
sig
  type key = string                (* concrete type *)
  type 'a entry = key * 'a         (* concrete type *)
  type 'a dict                     (* abstract type *)



end
```

Another example: dictionary

A dictionary is a collection of pairs of the form

```
(key, value)
```

where all keys must be unique within a dictionary.

```
signature DICT =
sig
  type key = string                       (* concrete type *)
  type 'a entry = key * 'a                (* concrete type *)
  type 'a dict                            (* abstract type *)


end
```

Let's use strings as keys for now

Permit value type to be polymorphic

Another example: dictionary

A dictionary is a collection of pairs of the form

`(key, value)`

where all keys must be unique within a dictionary.

```
signature DICT =
sig
  type key = string                    (* concrete type *)
  type 'a entry = key * 'a             (* concrete type *)
  type 'a dict                         (* abstract type *)



end
```

Another example: dictionary

A dictionary is a collection of pairs of the form

```
(key, value)
```

where all keys must be unique within a dictionary.

```
signature DICT =
sig
  type key = string                     (* concrete type *)
  type 'a entry = key * 'a              (* concrete type *)
  type 'a dict                          (* abstract type *)
  val empty : 'a dict
  val lookup : 'a dict -> key -> 'a option
  val insert : 'a dict * 'a entry -> 'a dict
end
```

Another example: dictionary

A dictionary is a collection of pairs of the form

```
(key, value)
```

where all keys must be unique within a dictionary.

```
signature DICT =
sig
  type key = string                    (* concrete type *)
  type 'a entry = key * 'a             (* concrete type *)
  type 'a dict                         (* abstract type *)
  val empty : 'a dict
  val lookup : 'a dict -> key -> 'a option
  val insert : 'a dict * 'a entry -> 'a dict
end
```

Another example: dictionary

A dictionary is a collection of pairs of the form

```
(key, value)
```

where all keys must be unique within a dictionary.

```
signature DICT =
sig
  type key = string                (* concrete type *)
  type 'a entry = key * 'a         (* concrete type *)
  type 'a dict
  val empty : 'a dict
  val lookup : 'a dict -> key -> 'a option
  val insert : 'a dict * 'a entry -> 'a dict
end
```

Replace entry, if key already exists

Search tree representation of dictionary

Search tree representation of dictionary

Implementation: **represent** dictionary as a binary search tree, where

Search tree representation of dictionary

Implementation: **represent** dictionary as a binary search tree, where

`(key, value)`

Search tree representation of dictionary

Implementation: **represent** dictionary as a binary search tree, where

```
(key, value)
```

are stored in nodes.

Search tree representation of dictionary

Implementation: **represent** dictionary as a binary search tree, where

```
(key, value)
```

are stored in nodes.

Representation **invariant**:

Search tree representation of dictionary

Implementation: **represent** dictionary as a binary search tree, where

`(key, value)`

are stored in nodes.

Representation **invariant**:

Tree must be sorted.

Search tree representation of dictionary

Implementation: **represent** dictionary as a binary search tree, where

`(key, value)`

are stored in nodes.

- Representation **invariant**:
  - Tree must be sorted.
  - Keys must be unique.

Search tree representation of dictionary

Implementation: **represent** dictionary as a binary search tree, where

`(key, value)`

are stored in nodes.

- Representation **invariant**:
  - Tree must be sorted.
  - Keys must be unique.
- All functions declared by structure

Search tree representation of dictionary

Implementation: **represent** dictionary as a binary search tree, where

`(key, value)`

are stored in nodes.

- Representation **invariant**:
  - Tree must be sorted.
  - Keys must be unique.
- All functions declared by structure
  - may **assume** that received tree is sorted,

Search tree representation of dictionary

Implementation: **represent** dictionary as a binary search tree, where

```
(key, value)
```

are stored in nodes.

- Representation **invariant**:
  - Tree must be sorted.
  - Keys must be unique.
- All functions declared by structure
  - may **assume** that received tree is sorted,
  - and must **assert** that returned tree is sorted.

Search tree representation of dictionary

Implementation: **represent** dictionary as a binary search tree, where

```
(key, value)
```

are stored in nodes.

- Representation **invariant**:
  - Tree must be sorted.
  - Keys must be unique.
- All functions declared by structure
  - may **assume** that received tree is sorted,
  - and must **assert** that returned tree is sorted.
  - (Similarly for key uniqueness.)

Search tree representation of dictionary

Search tree representation of dictionary

```
signature DICT =
sig
  type key = string           (* concrete type *)
  type 'a entry = key * ‘a (* concrete type *)
  type 'a dict                (* abstract type *)
  val empty : 'a dict
  val lookup : 'a dict -> key -> 'a option
  val insert : 'a dict * 'a entry -> 'a dict
end
```

Search tree representation of dictionary

```
signature DICT =
sig
  type key = string           (* concrete type *)
  type 'a entry = key * ‘a (* concrete type *)
  type 'a dict                (* abstract type *)
  val empty : 'a dict
  val lookup : 'a dict -> key -> 'a option
  val insert : 'a dict * 'a entry -> 'a dict
end
```

```
structure BST : DICT =
struct
  type key = string
  type 'a entry = key * 'a

  datatype 'a tree =
    Empty | Node of 'a tree * 'a entry * 'a tree

  type 'a dict = 'a tree




end
```

Search tree representation of dictionary

```
signature DICT =
sig
  type key = string          (* concrete type *)
  type 'a entry = key * ‘a (* concrete type *)
  type 'a dict               (* abstract type *)
  val empty : 'a dict
  val lookup : 'a dict -> key -> 'a option
  val insert : 'a dict * 'a entry -> 'a dict
end
```

```
structure BST : DICT =
struct
  type key = string
  type 'a entry = key * 'a

  datatype 'a tree =
    Empty | Node of 'a tree * 'a entry * 'a tree

  type 'a dict = 'a tree




end
```

Search tree representation of dictionary

```
signature DICT =
sig
  type key = string                  (* concrete type *)
  type 'a entry = key * ‘a           (* concrete type *)
  type 'a dict                       (* abstract type *)
  val empty : 'a dict
  val lookup : 'a dict -> key -> 'a option
  val insert : 'a dict * 'a entry -> 'a dict
end
```

Explore :>

```
structure BST : DICT =
struct
  type key = string
  type 'a entry = key * 'a
  datatype 'a tree =
    Empty | Node of 'a tree * 'a entry * 'a tree
  type 'a dict = 'a tree




end
```

Search tree representation of dictionary

```
signature DICT =
sig
  type key = string              (* concrete type *)
  type 'a entry = key * ‘a (* concrete type *)
  type 'a dict                   (* abstract type *)
  val empty : 'a dict
  val lookup : 'a dict -> key -> 'a option
  val insert : 'a dict * 'a entry -> 'a dict
end
```

Explore :>

```
structure BST : DICT =
struct
  type key = string
  type 'a entry = key * 'a
  datatype 'a tree =
    Empty | Node of 'a tree * 'a entry * 'a tree
  type 'a dict = 'a tree
```

Transparent ascription can be useful for debugging purposes.

```
end
```

Search tree representation of dictionary

```
signature DICT =
sig
  type key = string             (* concrete type *)
  type 'a entry = key * ‘a (* concrete type *)
  type 'a dict                  (* abstract type *)
  val empty : 'a dict
  val lookup : 'a dict -> key -> 'a option
  val insert : 'a dict * 'a entry -> 'a dict
end
```

```
structure BST : DICT =
struct
  type key = string
  type 'a entry = key * 'a

  datatype 'a tree =
    Empty | Node of 'a tree * 'a entry * 'a tree

  type 'a dict = 'a tree




end
```

Search tree representation of dictionary

```
signature DICT =
sig
  type key = string              (* concrete type *)
  type 'a entry = key * ‘a (* concrete type *)
  type 'a dict                   (* abstract type *)
  val empty : 'a dict
  val lookup : 'a dict -> key -> 'a option
  val insert : 'a dict * 'a entry -> 'a dict
end
```

```
structure BST : DICT =
struct
  type key = string
  type 'a entry = key * 'a
  datatype 'a tree =
    Empty | Node of 'a tree * 'a entry * 'a tree
  type 'a dict = 'a tree



end
```

Search tree representation of dictionary

```
signature DICT =
sig
  type key = string          (* concrete type *)
  type 'a entry = key * ‘a (* concrete type *)
  type 'a dict               (* abstract type *)
  val empty : 'a dict
  val lookup : 'a dict -> key -> 'a option
  val insert : 'a dict * 'a entry -> 'a dict
end
```

```
structure BST : DICT =
struct
  type key = string
  type 'a entry = key * 'a
  datatype 'a tree =
    Empty | Node of 'a tree * 'a entry * 'a tree
  type 'a dict = 'a tree



end
```

no other choice

Search tree representation of dictionary

```
signature DICT =
sig
  type key = string             (* concrete type *)
  type 'a entry = key * ‘a (* concrete type *)
  type 'a dict                  (* abstract type *)
  val empty : 'a dict
  val lookup : 'a dict -> key -> 'a option
  val insert : 'a dict * 'a entry -> 'a dict
end
```

```
structure BST : DICT =
struct
  type key = string
  type 'a entry = key * 'a

  datatype 'a tree =
    Empty | Node of 'a tree * 'a entry * 'a tree

  type 'a dict = 'a tree

end
```

Search tree representation of dictionary

```
signature DICT =
sig
  type key = string          (* concrete type *)
  type 'a entry = key * ‘a (* concrete type *)
  type 'a dict               (* abstract type *)
  val empty : 'a dict
  val lookup : 'a dict -> key -> 'a option
  val insert : 'a dict * 'a entry -> 'a dict
end
```

```
structure BST : DICT =
struct
  type key = string
  type 'a entry = key * 'a
  datatype 'a tree =
    Empty | Node of 'a tree * 'a entry * 'a tree
  type 'a dict = 'a tree


end
```

Search tree representation of dictionary

```
signature DICT =
sig
  type key = string          (* concrete type *)
  type 'a entry = key * 'a   (* concrete type *)
  type 'a dict               (* abstract type *)
  val empty : 'a dict
  val lookup : 'a dict -> key -> 'a option
  val insert : 'a dict * 'a entry -> 'a dict
end
```

```
structure BST : DICT =
struct
  type key = string
  type 'a entry = key * 'a
  datatype 'a tree =
    Empty | Node of 'a tree * 'a entry * 'a tree
  type 'a dict = 'a tree
```

Because datatype is not declared in signature, constructors (and thus pattern matching) are not available outside signature.

```
end
```

Search tree representation of dictionary

```
signature DICT =
sig
  type key = string             (* concrete type *)
  type 'a entry = key * ‘a (* concrete type *)
  type 'a dict                  (* abstract type *)
  val empty : 'a dict
  val lookup : 'a dict -> key -> 'a option
  val insert : 'a dict * 'a entry -> 'a dict
end
```

```
structure BST : DICT =
struct
  type key = string
  type 'a entry = key * 'a
  datatype 'a tree =
    Empty | Node of 'a tree * 'a entry * 'a tree
  type 'a dict = 'a tree
```

Because datatype is not declared in signature, constructors (and thus pattern matching) are not available outside signature.

```
end
```

But bindings externally visible due to transparent ascription.

Search tree representation of dictionary

```
signature DICT =
sig
  type key = string            (* concrete type *)
  type 'a entry = key * ‘a (* concrete type *)
  type 'a dict                 (* abstract type *)
  val empty : 'a dict
  val lookup : 'a dict -> key -> 'a option
  val insert : 'a dict * 'a entry -> 'a dict
end
```

```
structure BST : DICT =
struct
  type key = string
  type 'a entry = key * 'a

  datatype 'a tree =
    Empty | Node of 'a tree * 'a entry * 'a tree

  type 'a dict = 'a tree




end
```

Search tree representation of dictionary

```
signature DICT =
sig
  type key = string         (* concrete type *)
  type 'a entry = key * ‘a (* concrete type *)
  type 'a dict              (* abstract type *)
  val empty : 'a dict
  val lookup : 'a dict -> key -> 'a option
  val insert : 'a dict * 'a entry -> 'a dict
end
```

```
structure BST : DICT =
struct
  type key = string
  type 'a entry = key * 'a

  datatype 'a tree =
    Empty | Node of 'a tree * 'a entry * 'a tree

  type 'a dict = 'a tree

  val empty = Empty

  fun lookup ...
  fun insert ...
end
```

Search tree representation of dictionary

Search tree representation of dictionary

```
(* ins : 'a dict * 'a entry -> 'a dict *)
fun insert (Empty, e) = Node(Empty, e, Empty)
  | insert (Node(lt, e’ as (k’,_), rt),
            e as (k,_)) =
```

Search tree representation of dictionary

```
(* ins : 'a dict * 'a entry -> 'a dict *)
fun insert (Empty, e) = Node(Empty, e, Empty)
  | insert (Node(lt, e’ as (k’,_), rt),
            e as (k,_)) =
```

Search tree representation of dictionary

```
(* ins : 'a dict * 'a entry -> 'a dict *)
fun insert (Empty, e) = Node(Empty, e, Empty)
  | insert (Node(lt, e’ as (k’,_), rt),
              e as (k,_)) =
```

Layered pattern matching

Search tree representation of dictionary

```
(* ins : 'a dict * 'a entry -> 'a dict *)
fun insert (Empty, e) = Node(Empty, e, Empty)
  | insert (Node(lt, e’ as (k’,_), rt),
            e as (k,_)) =
```

Search tree representation of dictionary

```
(* ins : 'a dict * 'a entry -> 'a dict *)
fun insert (Empty, e) = Node(Empty, e, Empty)
  | insert (Node(lt, e’ as (k’,_), rt),
            e as (k,_)) =
      (case String.compare(k,k’) of
```

Search tree representation of dictionary

```
(* ins : 'a dict * 'a entry -> 'a dict *)
fun insert (Empty, e) = Node(Empty, e, Empty)
  | insert (Node(lt, e’ as (k’,_), rt),
            e as (k,_)) =
      (case String.compare(k,k’) of
        EQUAL => Node(lt, e, rt)
```

Search tree representation of dictionary

```
(* ins : 'a dict * 'a entry -> 'a dict *)
fun insert (Empty, e) = Node(Empty, e, Empty)
  | insert (Node(lt, e’ as (k’,_), rt),
            e as (k,_)) =
      (case String.compare(k,k’) of
        EQUAL => Node(lt, e, rt)
```

Search tree representation of dictionary

```
(* ins : 'a dict * 'a entry -> 'a dict *)
fun insert (Empty, e) = Node(Empty, e, Empty)
  | insert (Node(lt, e' as (k',_), rt),
            e as (k,_)) =
      (case String.compare(k,k') of
        EQUAL => Node(lt, e, rt)
```

Replace existing entry with new one

Search tree representation of dictionary

```
(* ins : 'a dict * 'a entry -> 'a dict *)
fun insert (Empty, e) = Node(Empty, e, Empty)
  | insert (Node(lt, e’ as (k’,_), rt),
            e as (k,_)) =
     (case String.compare(k,k’) of
       EQUAL => Node(lt, e, rt)
```

Search tree representation of dictionary

```
(* ins : 'a dict * 'a entry -> 'a dict *)
fun insert (Empty, e) = Node(Empty, e, Empty)
  | insert (Node(lt, e’ as (k’,_), rt),
            e as (k,_)) =
      (case String.compare(k,k’) of
        EQUAL => Node(lt, e, rt)
        | LESS => Node(insert(lt,e), e’, rt)
```

Search tree representation of dictionary

```
(* ins : 'a dict * 'a entry -> 'a dict *)
fun insert (Empty, e) = Node(Empty, e, Empty)
  | insert (Node(lt, e’ as (k’,_), rt),
            e as (k,_)) =
      (case String.compare(k,k’) of
        EQUAL => Node(lt, e, rt)
        | LESS => Node(insert(lt,e), e’, rt)
        | GREATER => Node(lt, e’, insert(rt,e)))
```

Search tree representation of dictionary

Search tree representation of dictionary

```
(* lookup : 'a dict -> key -> 'a option *)
fun lookup tree key =
 let
    fun lk (Empty) = NONE
    | lk (Node(left, (k,v), right)) =
      (case String.compare(key,k) of
        EQUAL => SOME(v)
        | LESS => lk left
        | GREATER => lk right)
  in
    lk tree
  end
```

Search tree representation of dictionary

Search tree representation of dictionary

Let's interact with **BST**:

Search tree representation of dictionary

Let’s interact with **BST**:

```
val d = BST.insert(BST.insert(BST.insert(
          BST.empty,(“a",1)),("b",2)),("c",3))
```

Search tree representation of dictionary

Let’s interact with **BST**:

```
val d = BST.insert(BST.insert(BST.insert(
          BST.empty,(“a",1)),("b",2)),("c",3))
```

What is the type of **d**?

Search tree representation of dictionary

Let’s interact with **BST**:

```
val d = BST.insert(BST.insert(BST.insert(
          BST.empty,(“a",1)),("b",2)),("c",3))
```

What is the type of **d**?

```
int BST.dict
```

Search tree representation of dictionary

Let’s interact with **BST**:

```
val d = BST.insert(BST.insert(BST.insert(
          BST.empty,(“a",1)),("b",2)),("c",3))
```

What is the type of **d**?

```
int BST.dict
```

The binding for **d** will be revealed because of opaque ascription. However, because the tree datatype is not declared in the signature, a client cannot pattern match on its constructors.

Search tree representation of dictionary

Let's interact with **BST**:

```
val d = BST.insert(BST.insert(BST.insert(
          BST.empty,(“a",1)),("b",2)),("c",3))
```

What is the type of **d**?

```
int BST.dict
```

The binding for **d** will be revealed because of opaque ascription. However, because the tree datatype is not declared in the signature, a client cannot pattern match on its constructors.

Now consider:

Search tree representation of dictionary

Let's interact with **BST**:

```
val d = BST.insert(BST.insert(BST.insert(
          BST.empty,(“a",1)),("b",2)),("c",3))
```

What is the type of **d**?

```
int BST.dict
```

The binding for **d** will be revealed because of opaque ascription. However, because the tree datatype is not declared in the signature, a client cannot pattern match on its constructors.

Now consider: `val look = BST.lookup d`

Search tree representation of dictionary

Let's interact with **BST**:

```
val d = BST.insert(BST.insert(BST.insert(
          BST.empty,(“a",1)),("b",2)),("c",3))
```

What is the type of **d**?

```
int BST.dict
```

The binding for **d** will be revealed because of opaque ascription. However, because the tree datatype is not declared in the signature, a client cannot pattern match on its constructors.

Now consider: `val look = BST.lookup d`

What is the type of **look**?

Search tree representation of dictionary

Let's interact with **BST**:

```
val d = BST.insert(BST.insert(BST.insert(
          BST.empty,(“a",1)),("b",2)),("c",3))
```

What is the type of **d**?

```
int BST.dict
```

The binding for **d** will be revealed because of opaque ascription. However, because the tree datatype is not declared in the signature, a client cannot pattern match on its constructors.

Now consider: `val look = BST.lookup d`

What is the type of **look**?

```
BST.key -> int option
```

Search tree representation of dictionary

Let's interact with **BST**:

```
val d = BST.insert(BST.insert(BST.insert(
          BST.empty,(“a",1)),("b",2)),("c",3))
```

What is the type of **d**?

```
int BST.dict
```

Now consider: `val look = BST.lookup d`

What is the type of **look**?

```
BST.key -> int option
```

Search tree representation of dictionary

Let’s interact with **BST**:

```
val d = BST.insert(BST.insert(BST.insert(
          BST.empty,(“a",1)),("b",2)),("c",3))
```

What is the type of **d**?

```
int BST.dict
```

Now consider: `val look = BST.lookup d`

What is the type of **look**?

```
BST.key -> int option
```

Now consider:

Search tree representation of dictionary

Let's interact with **BST**:

```
val d = BST.insert(BST.insert(BST.insert(
          BST.empty,("a",1)),("b",2)),("c",3))
```

What is the type of **d**?

```
int BST.dict
```

Now consider: `val look = BST.lookup d`

What is the type of **look**?

```
BST.key -> int option
```

Now consider:

```
val x = look "e"
val y = look "a"
```

Search tree representation of dictionary

Let's interact with **BST**:

```
val d = BST.insert(BST.insert(BST.insert(
          BST.empty,(“a",1)),("b",2)),("c",3))
```

What is the type of **d**?

```
int BST.dict
```

Now consider: `val look = BST.lookup d`

What is the type of **look**?

```
BST.key -> int option
```

Now consider:

```
val x = look “e”
val y = look “a”
```

Bindings:

Search tree representation of dictionary

Let's interact with **BST**:

```
val d = BST.insert(BST.insert(BST.insert(
          BST.empty,(“a",1)),("b",2)),("c",3))
```

What is the type of **d**?

```
int BST.dict
```

Now consider: `val look = BST.lookup d`

What is the type of **look**?

```
BST.key -> int option
```

Now consider:

```
val x = look “e”
val y = look “a”
```

Bindings: `[NONE/x, (SOME 1)/y]`

That's all for today.
Next time we discuss functors!