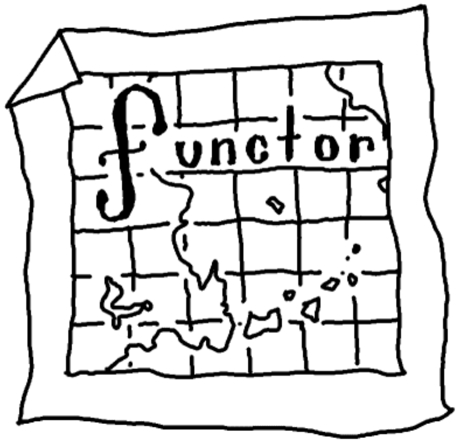
Modules

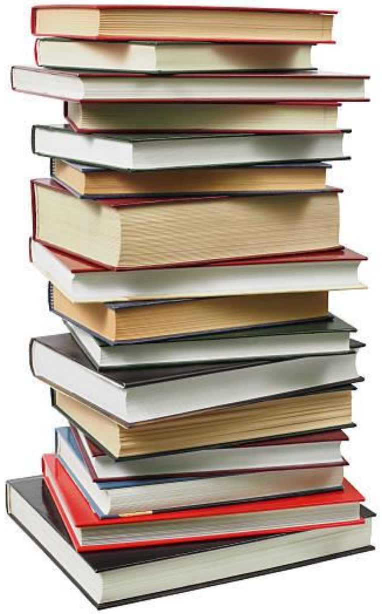
Password: “kind”

Modules
=
Structures & Functors



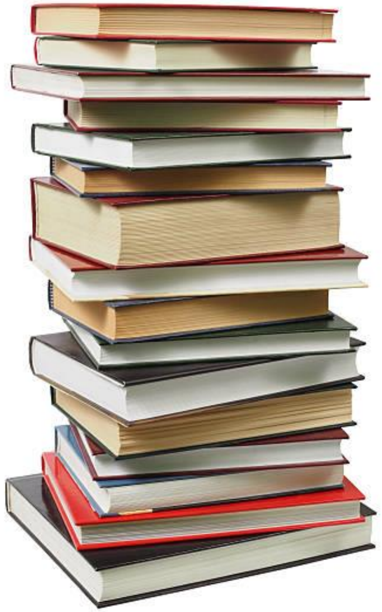
Back to the beginning: Interfaces (in 15-122)

```
//typedef ______* stack_t;

stack_t empty_stack();
int size(stack_t S);
void push(stack_t S, int i);
int pop(stack_t S);
```



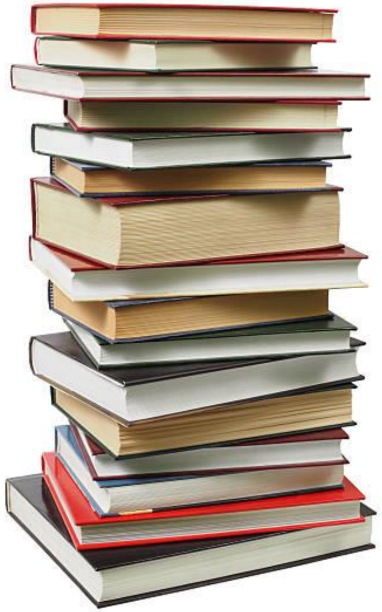
Back to the beginning: Interfaces (in 15-122)

```
struct stack {
   int value;
   struct stack* next;
};

typedef struct stack* stack_t;

stack_t empty_stack() {

…
```



Back to the beginning: Interfaces (in 15-122)

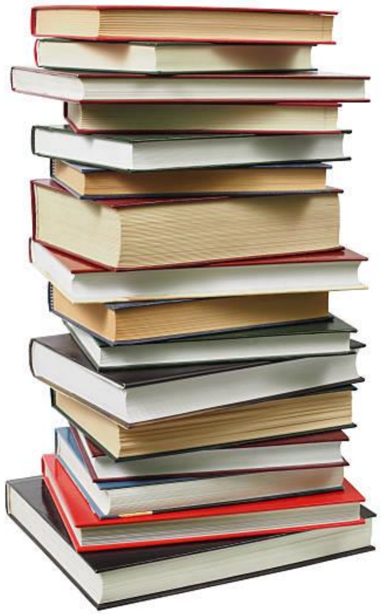
```
int client_function(stack_t S) {
    return S->value;
}
```



Doesn’t respect the interface

SML is better

```
signature STACK = sig
   type stack
   val empty_stack : unit -> stack
   val size : stack -> int
   val push : stack -> int -> unit
   val pop : stack -> unit
end
structure Stack :> STACK = struct
   type stack = list ref
   fun empty_stack () =
   …
end

fun client_function(s : Stack.stack) = List.hd (!s)  (* Type error *)
```



How are modules typechecked?

To check `M :> S`,

1. Find the type that the dynamic portion of S represents. Call this tau’.
2. Find the type of the dynamic portion of M, rearranging, deleting, or monomorphizing fields as necessary to get it to be similar to tau’. Call this tau.

3. Get the *least general kind* of the static portion of M. Call this *k.*
4. Get the kind that the static portion of S represents. Call this *k’.*

5. Check if *k* is a subkind of *k’.*
6. Check if tau is equal to tau’.

Steps 1 & 2

1. Find the type of the dynamic portion of M, rearranging, deleting, or monomorphizing fields as necessary. Call this tau.
2. Find the type that the dynamic portion of S represents. Call this tau’.

Signatures, Signatures, Signatures

A signature specifies

1. Abstract types that must be defined
2. Concrete type definitions
3. Exception declarations
4. The types of values that must be defined

A signature not only gives the types of *values* in a structure, it gives the types of *types* in that structure.

Anatomy of a signature

```
signature S = sig
     type t
     type s = int
     type v = t

     val x : t
     val h : t -> int
     val g : t -> s
     val f : t -> s -> v
end
```

Static Component (types(?) of types) — applies to the type declarations (`type t`, `type s = int`, `type v = t`)

Dynamic Component (types of values) — applies to the value declarations (`val x`, `val h`, `val g`, `val f`)

Type of the dynamic component

Think of it like a record

```
type S_dynamic = {
   x : t,
   h : t -> int,
   g : t -> s,
   f : t -> s -> v
}
```

Depends on the type of the static component!

Quick review of records

```
type r = {
   x : int,
   f : bool -> int,
   y : unit
}

val v : r = {x=10,
             f=(fn _ => 10),
             y=()}
```

Steps 3 & 4

3. Get the *least general kind* of the static portion of M. Call this *k.*
4. Get the kind that the static portion of S represents. Call this *k'.*

Anatomy of a signature

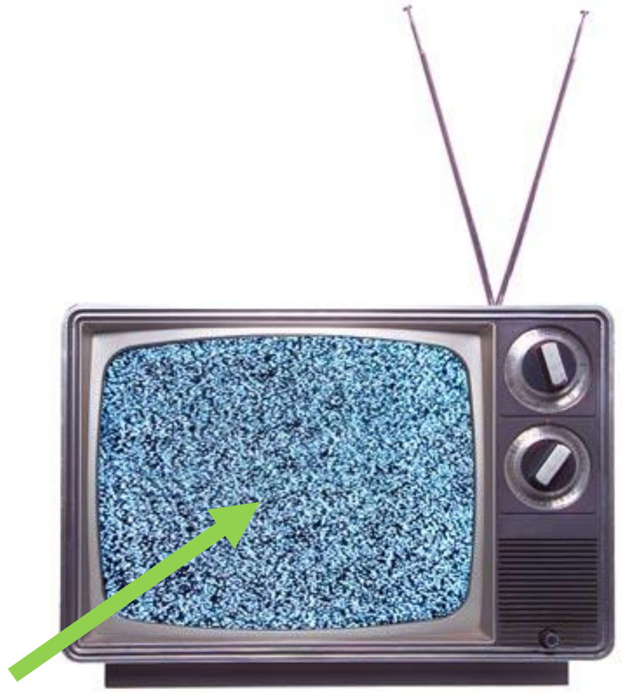
```
signature S = sig
     type t
     type s = int
     type v = t

     val x : t
     val h : t -> int
     val g : t -> s
     val f : t -> s -> v
end
```

Static Component (types(?) of types) — applies to the type declarations (type t, type s = int, type v = t)

Dynamic Component (types of values) — applies to the val declarations (val x, val h, val g, val f)

Kind of the static component

A type of types!



The “static” component

Kinds

Kinds describe

- Types `int :: Type`
- Functions on types `list :: Type -> Type`
- Tuples of types `(int, char) :: Type * Type`
- Records of types `{t=int, b=bool} :: {t :: Type, b :: Type}`
- Singletons (more on this later) `int :: S(int)`
- Dependent Records `{t=int, b=t} :: {t :: Type, b :: S(t)}`

Kinds

Here are some types:

Types	Values
int	10, 98317, 15312, 15417
int * bool	(10, true), (0, false)
bool list	[true, false, true], [true], []
(char, string) either	INR “hype for types”, INL #”c”

Here are some kinds:

Kinds	Types
Type (usually written T)	int, int list, (char, string) either, int * bool
Type -> Type	list, option, tree
Types * Type	(char, string), (int, int)
Type * Type -> Type	either

In a signature

```
type t
type ‘a s
type v
```

Describes the kind

```
{ t :: Type,
  s :: (Type -> Type),
  v :: Type }
```

In a structure

```
type t = int
type ‘a s = ‘a list
type v = bool
```

Describes the type

```
{t=int, s=list, v=bool}
```

What about this?

```
signature S = sig
    type t
    type s = int
    type v = t
end
```

Dependent Kinds

A *dependent kind* is a kind that depends on a type.

Motivation:

```
signature S = sig
   type t = int
end
```

The *kind* of t should depend on int. The kind of t should state that *t has to be int.*

Singleton Kinds

Kinds that specify what (single) type inhabits them

Denoted `S(type)`

Examples:

```
int :: S(int)
bool :: S(bool)
int list :: S(int list)
```

Invented by Bob Harper and Chris Stone

Dependent Record* Kinds

Later fields in the record type and kind can reference earlier fields

```
{t=int, s=t} :: {t :: Type, s :: S(t)}
```

* (Usually called dependent pairs and denoted $\prod(\alpha::k_1).k_2$)

In a signature

```
type t
type u = int
type v = t
```

Describes the kind

```
{ t :: Type,
  u :: S(int),
  v :: S(t) }
```

In a structure

```
type t = int
type u = bool
type v = t
```

Describes the type

```
{t=int, u=bool, v=t}
```

In a signature

```
type t
type u = int
type v = t
```

Describes the kind

```
{ t :: Type,
  u :: S(int),
  v :: S(t) }
```

In a structure

```
type t = int
type u = bool
type v = t
```

Describes the type

```
{t=int, u=bool, v=t}
```

The least general kind of this type is

```
{t :: S(int), u :: S(bool), v ::
S(t)}
```

This is a subkind of the signature's kind, since every type with kind S(int) also has kind Type.

Steps 5 & 6

5. Check if *k* is a subkind of *k'*.
6. Check if tau is equal to tau'.

Check it

```
signature S = sig
    type t
    type s = int
    val f : t -> s -> bool
end

structure M = struct
    type t = bool
    type s = int
    fun f b _ = b
end
```

S_{static} = {t :: Type, s :: S(int)}

$S_{dynamic}$ = {f : t -> s -> bool}

M_{static} = {t :: S(bool), s :: S(int)}

$M_{dynamic}$ = {f : bool -> int -> bool}

M_{static} <:: S_{static}

$S_{dynamic}$ = $M_{dynamic}$

Using a structure

```
signature S = sig
     type t
     type s = int
     type v = t

     val x : t
     val h : t -> int
     val g : t -> s
     val f : t -> v
end
```

Existential type

```
∃{t::Type,
  s::S(int),
  v::S(t)}.
  {x : t,
   h : t -> int,
   g : t -> s,
   f : t -> v}
```

Static Component

Dynamic Component

Using a structure

```
M : ∃{t::Type,
      s::S(int),
      v::S(t)}.
      {x : t,
       h : t -> int,
       g : s -> int,
       f : t -> v}
```

```
M.x
M.g 10
M.h 10
```