09 Effect Handlers

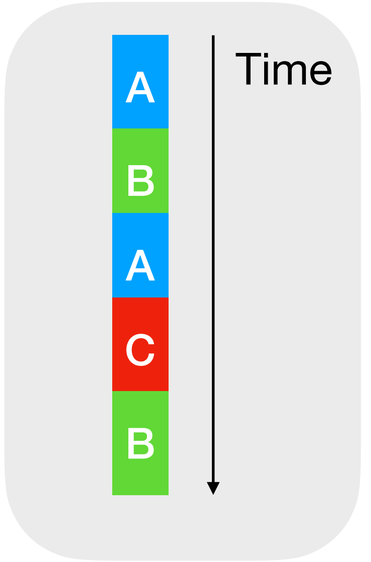
CS 6868: Concurrent Programming

KC Sivaramakrishnan

Spring 2026, IIT Madras

Concurrency

- ***Interleaved*** executions of tasks in time
- **Examples** — Asynchronous I/O, GUI in JavaScript, interrupt-driven embedded systems, etc.
- ***No need for multiple cores!***
- ***Challenges***
  - non-determinism
  - Many different schedules complicate reasoning
  - efficiency, etc.



Interleaved execution

Concurrent Programming

- Computations may be *suspended* and *resumed* later

Concurrent Programming

- Computations may be *suspended* and *resumed* later
- Many languages provide concurrent programming mechanisms as *primitives*
  - **async/await** — JavaScript, Python, Rust, C# 5.0, F#, Swift, …
  - **generators** — Python, Javascript, …
  - **coroutines** — C++, Kotlin, Lua, …
  - **futures & promises** — JavaScript, Swift, …
  - **Lightweight threads/processes** — Haskell, Go, Erlang

Concurrent Programming

- Computations may be *suspended* and *resumed* later
- Many languages provide concurrent programming mechanisms as *primitives*
  - **async/await** — JavaScript, Python, Rust, C# 5.0, F#, Swift, …
  - **generators** — Python, Javascript, …
  - **coroutines** — C++, Kotlin, Lua, …
  - **futures & promises** — JavaScript, Swift, …
  - **Lightweight threads/processes** — Haskell, Go, Erlang
- *Often include many different primitives in the same language!*
  - JavaScript has async/await, generators, promises, and callbacks

Effect Handlers

- Effect Handlers are a mechanism for programming with ***user-defined effects***
- They are the basis of all ***non-local control-flow mechanisms***
  - Exceptions
  - Generators
  - Iterators
  - Lightweight threads
  - Promises
  - Asynchronous IO
  - And generally, ***Concurrency***

This lecture

- Effect Handlers in OCaml 5
- Delimited Continuations and their basics
- Examples of using effect handlers

Effect handlers basics

Summing up numbers

```
let rec sum_up acc =
    let l = input_line stdin in
    acc := !acc + int_of_string l;
    sum_up acc

let _ =
  let r = ref 0 in
  try sum_up r with
  | End_of_file -> Printf.printf "Sum is %d\n" !r
```

```
$ ocaml test1.ml
```

User-defined exception

```
type exn += Conversion_failure of string
(* same as [exception Conversion_failure of string] *)

let int_of_string l =
  try int_of_string l with
  | Failure _ -> raise (Conversion_failure l)

let rec sum_up acc =
    let l = input_line stdin in
    acc := !acc + int_of_string l;
    sum_up acc

let _ =
  let r = ref 0 in
  try sum_up r with
  | End_of_file -> Printf.printf "Sum is %d\n" !r
  | Conversion_failure s ->
      Printf.fprintf stderr "Bad input: %s\n%!" s
```

```
$ ocaml test2.ml
```

Recovering with effect handlers

```
type _ Effect.t += Conversion_failure : string -> int Effect.t

let int_of_string l =
  try int_of_string l with
  | Failure _ -> perform (Conversion_failure l)

let rec sum_up acc =
    let l = input_line stdin in
    acc := !acc + int_of_string l;
    sum_up acc

let _ =
  let r = ref 0 in
  try sum_up r with
  | End_of_file -> Printf.printf "Sum is %d\n" !r
  | effect Conversion_failure s, k ->
      Printf.fprintf stderr "Bad input: %s\n%!" s;
      continue k 0
```

```
$ ocaml test3.ml
```

Effect handlers — Syntax

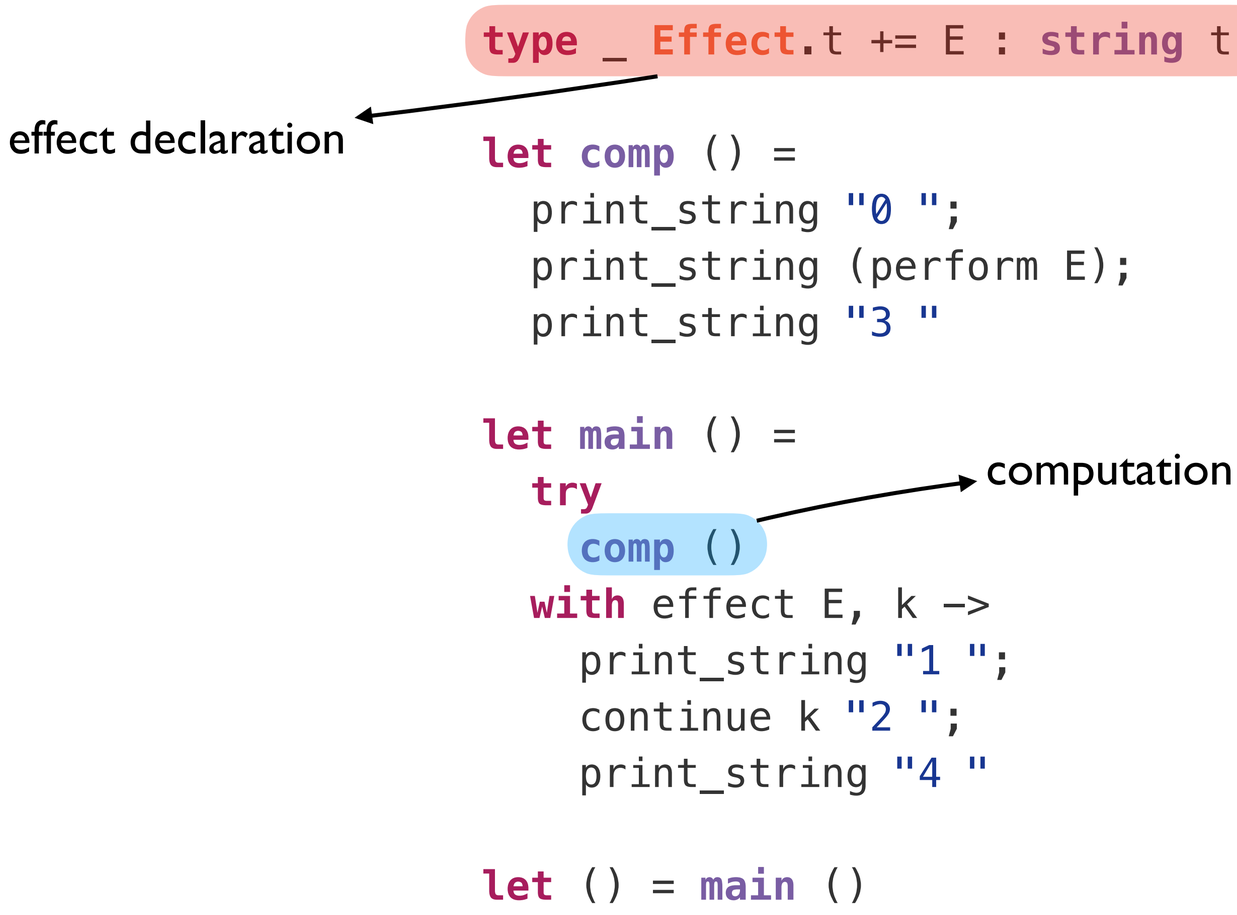
```
type _ Effect.t += E : string t

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E, k ->
    print_string "1 ";
    continue k "2 ";
    print_string "4 "

let () = main ()
```

Effect handlers — Syntax

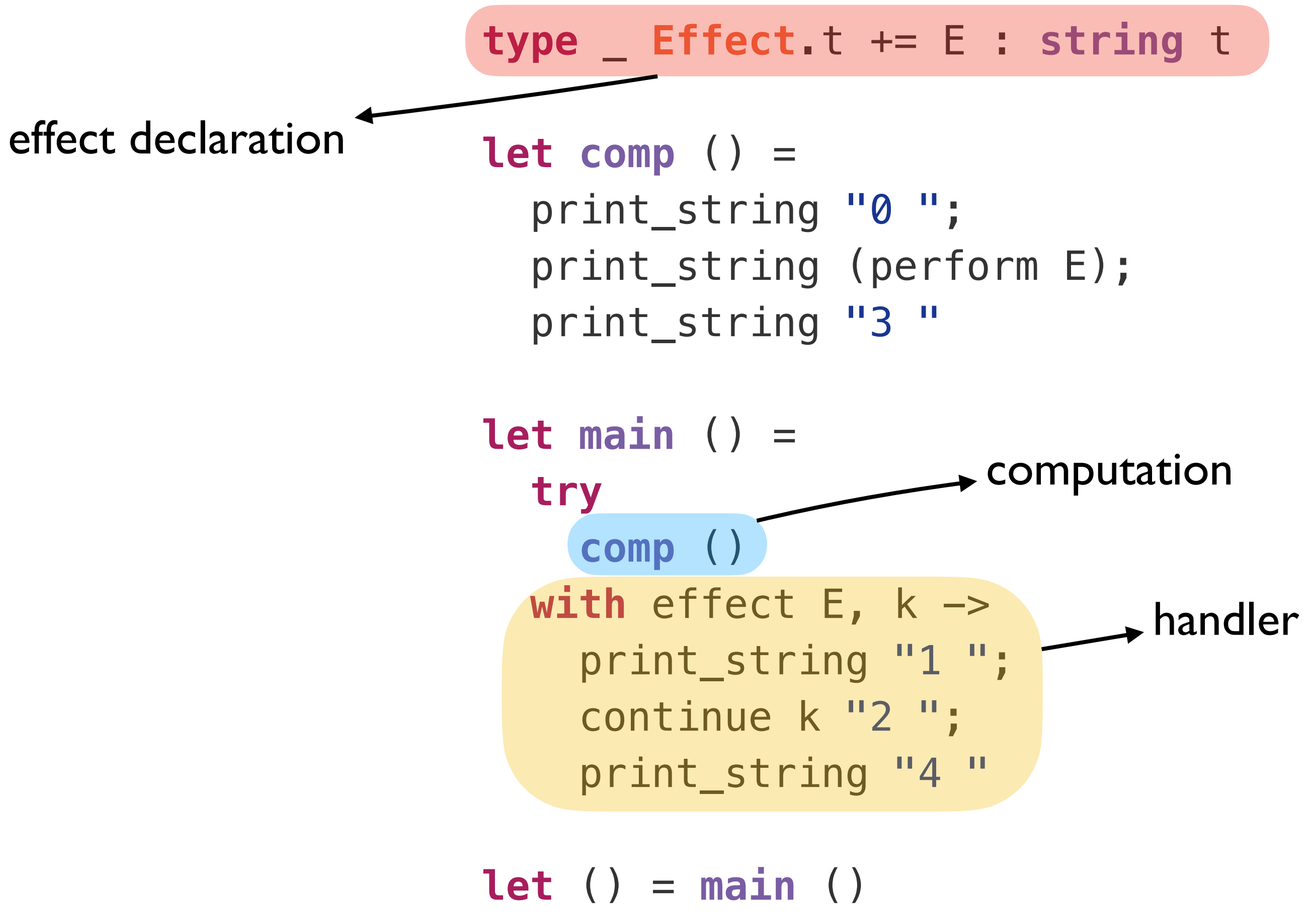
effect declaration

```
type _ Effect.t += E : string t

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E, k ->
    print_string "1 ";
    continue k "2 ";
    print_string "4 "

let () = main ()
```

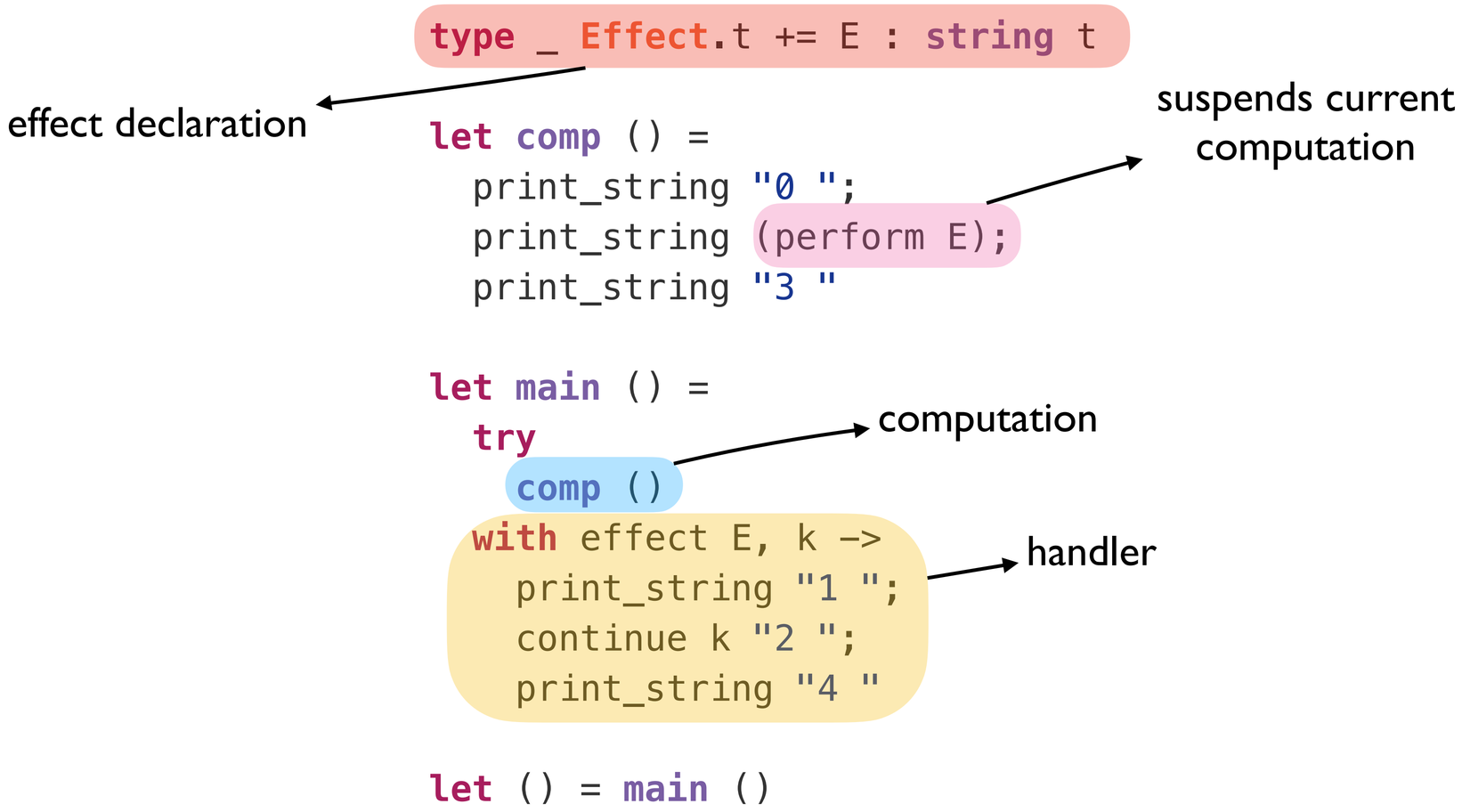
Effect handlers — Syntax



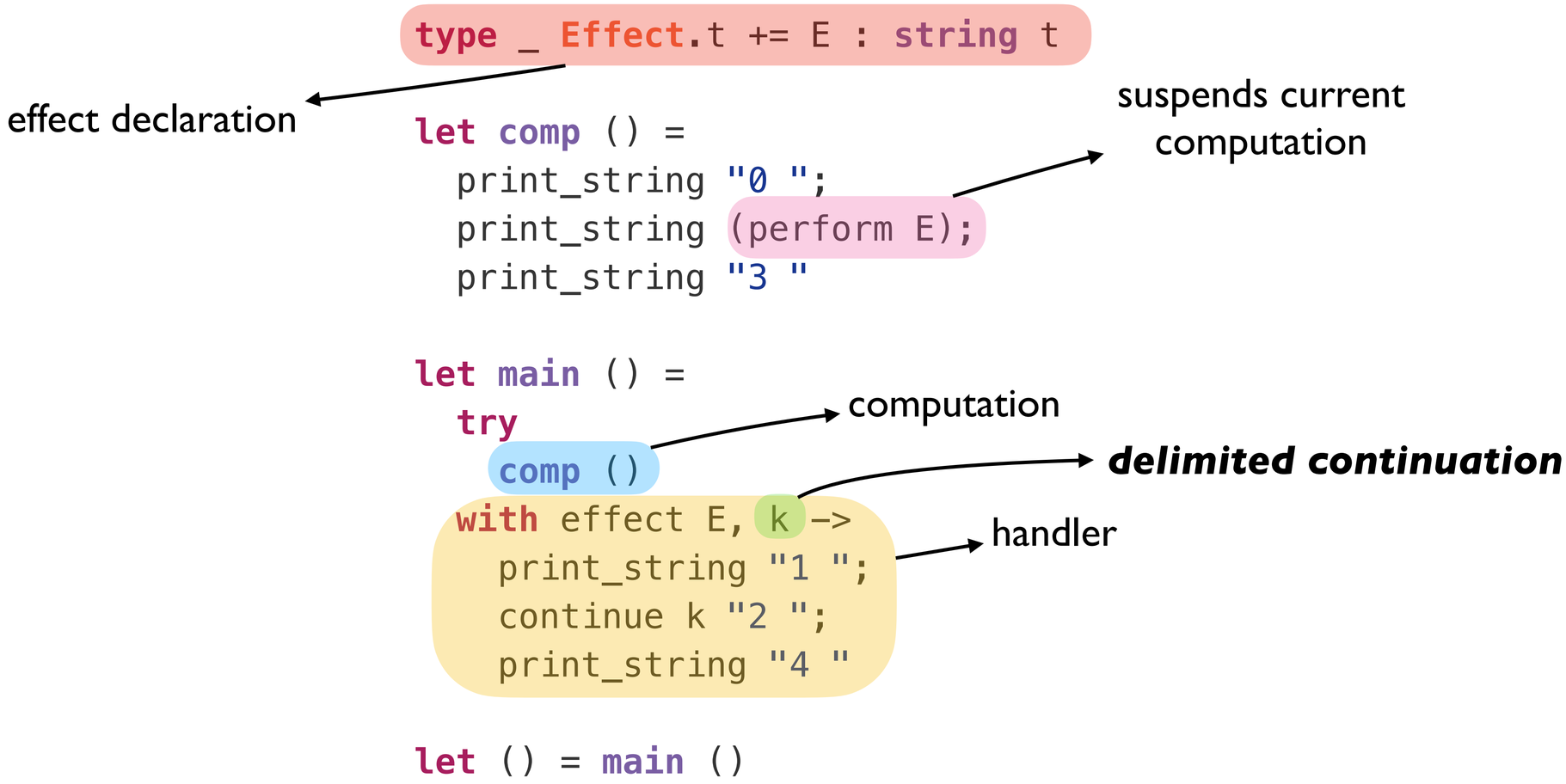
Effect handlers — Syntax



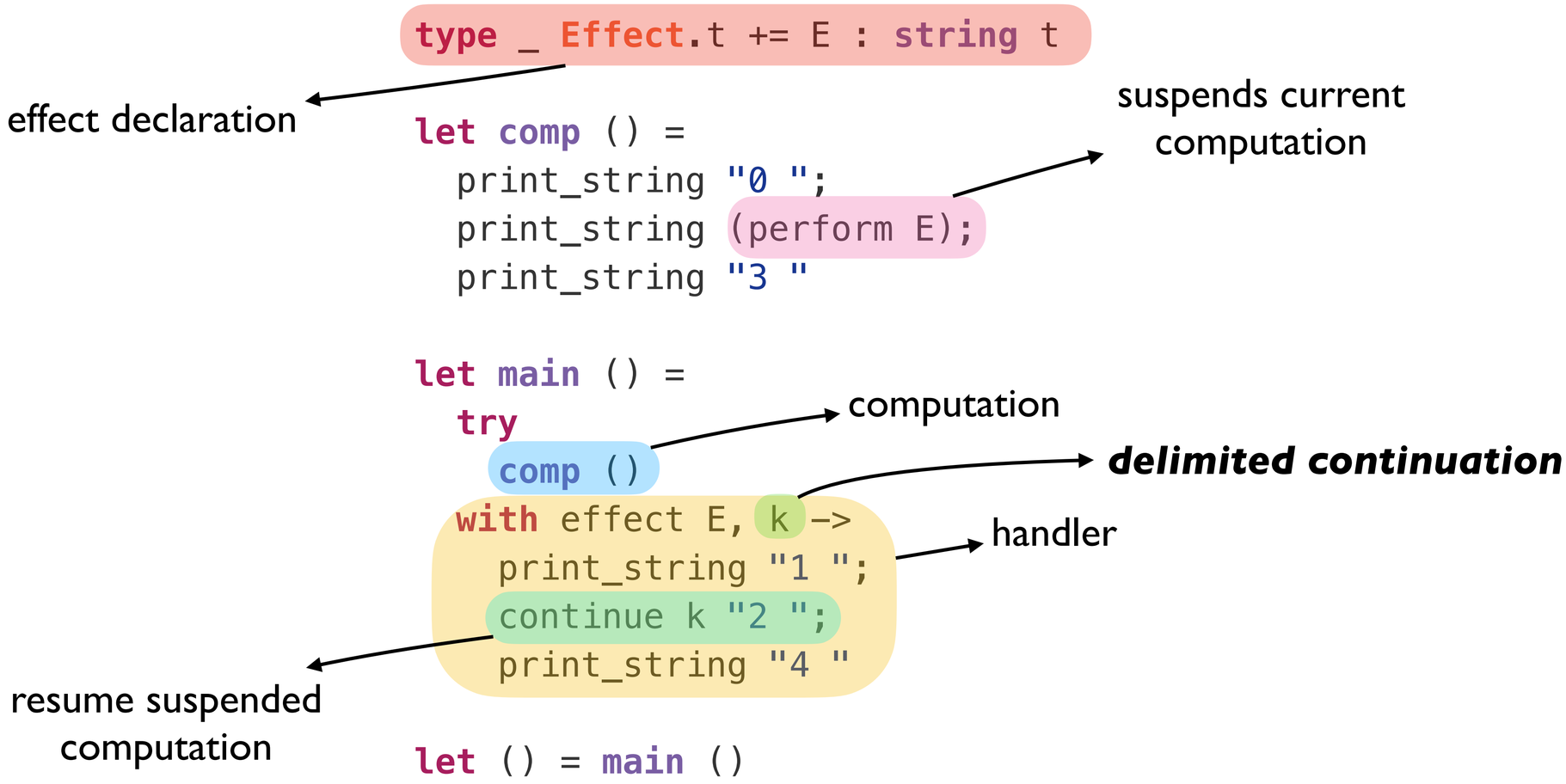
Effect handlers — Syntax



Effect handlers — Syntax



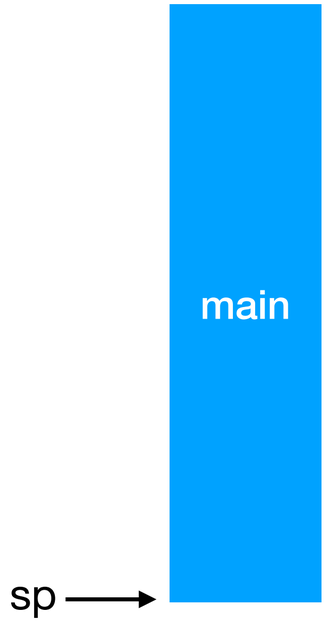
Effect handlers — Syntax



Effect Handlers — Tracing the execution

```
type _ Effect.t += E : string t

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E, k ->
    print_string "1 ";
    continue k "2 ";
    print_string "4 "
```

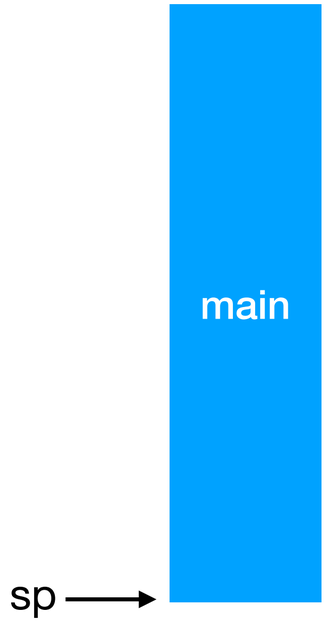
pc →



Effect Handlers — Tracing the execution

```
type _ Effect.t += E : string t

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E, k ->
    print_string "1 ";
    continue k "2 ";
    print_string "4 "
```

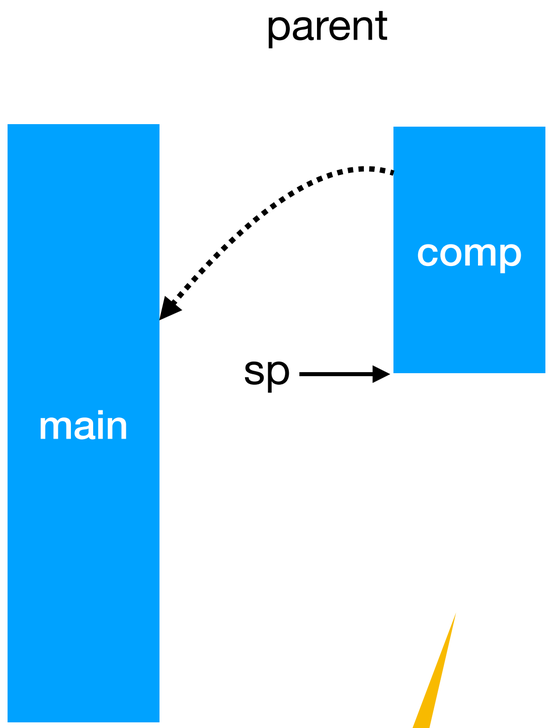
pc →



Effect Handlers — Tracing the execution

```
type _ Effect.t += E : string t

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E, k ->
    print_string "1 ";
    continue k "2 ";
    print_string "4 "
```

pc →



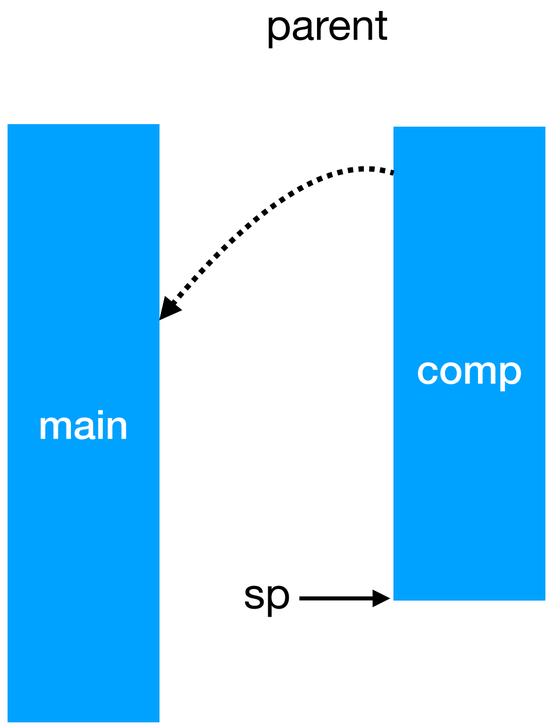
Fiber: A piece of stack + effect handler

Effect Handlers — Tracing the execution

```
type _ Effect.t += E : string t

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E, k ->
    print_string "1 ";
    continue k "2 ";
    print_string "4 "
```

pc →

0

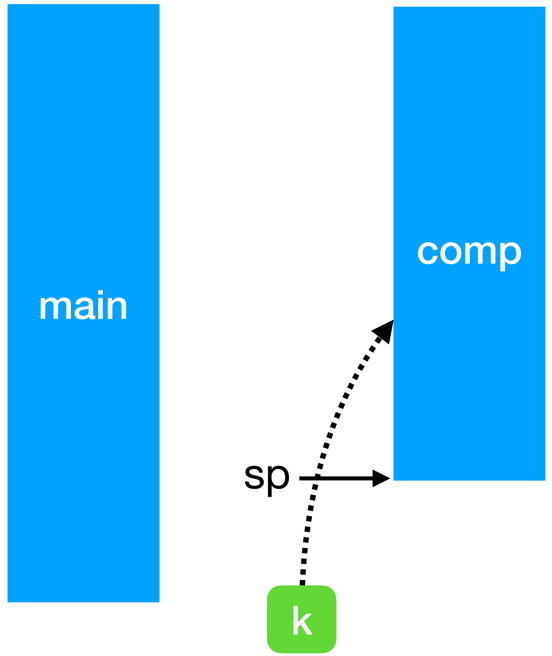


Effect Handlers — Tracing the execution

```
type _ Effect.t += E : string t

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E, k ->
    print_string "1 ";
    continue k "2 ";
    print_string "4 "
```

pc →

0

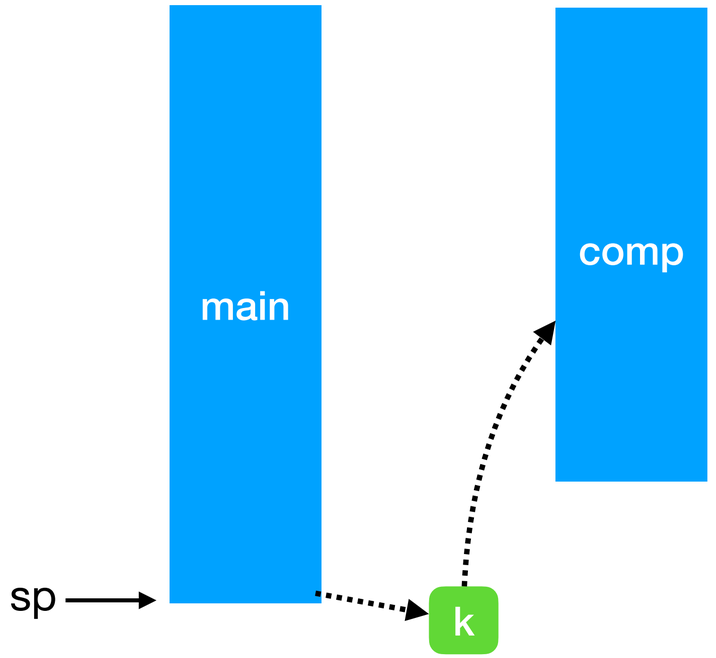


Effect Handlers — Tracing the execution

```
type _ Effect.t += E : string t

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E, k ->
    print_string "1 ";
    continue k "2 ";
    print_string "4 "
```

pc →

0

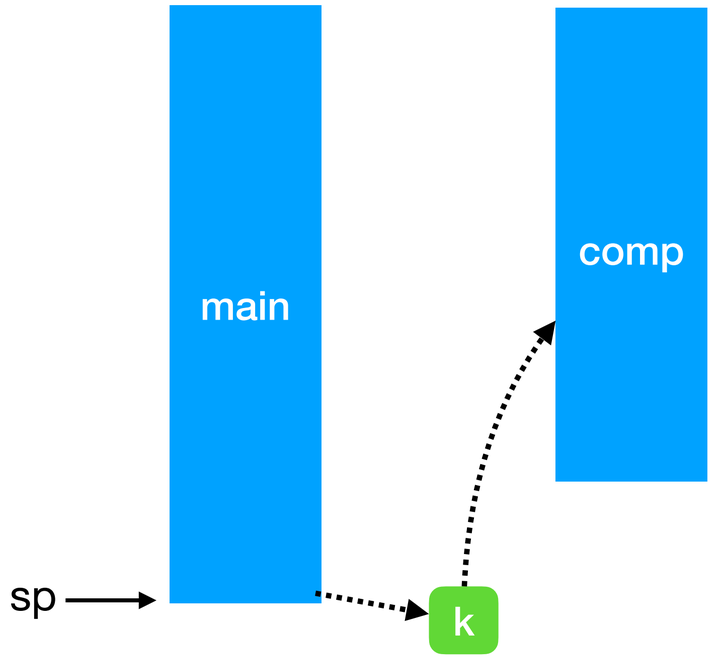


Effect Handlers — Tracing the execution

```
type _ Effect.t += E : string t

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E, k ->
    print_string "1 ";
    continue k "2 ";
    print_string "4 "
```

pc →

0

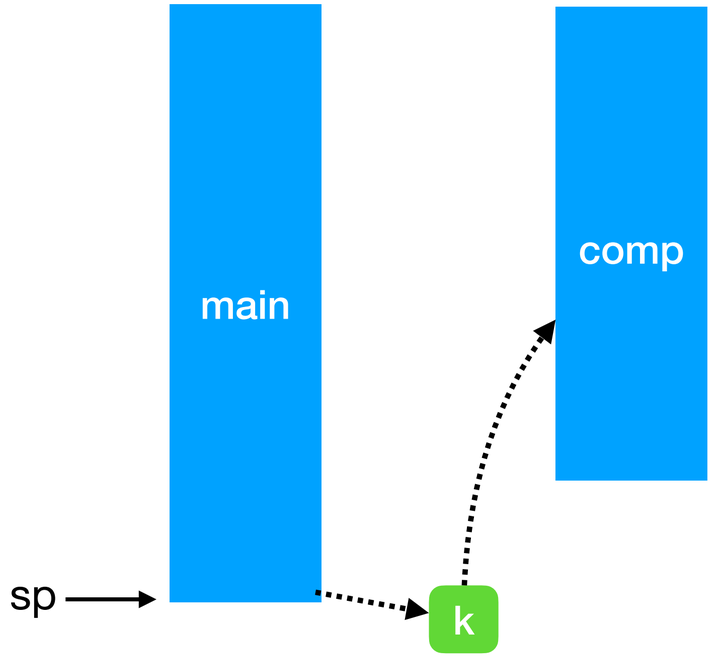


Effect Handlers — Tracing the execution

```
type _ Effect.t += E : string t

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E, k ->
    print_string "1 ";
    continue k "2 ";
    print_string "4 "
```

pc →

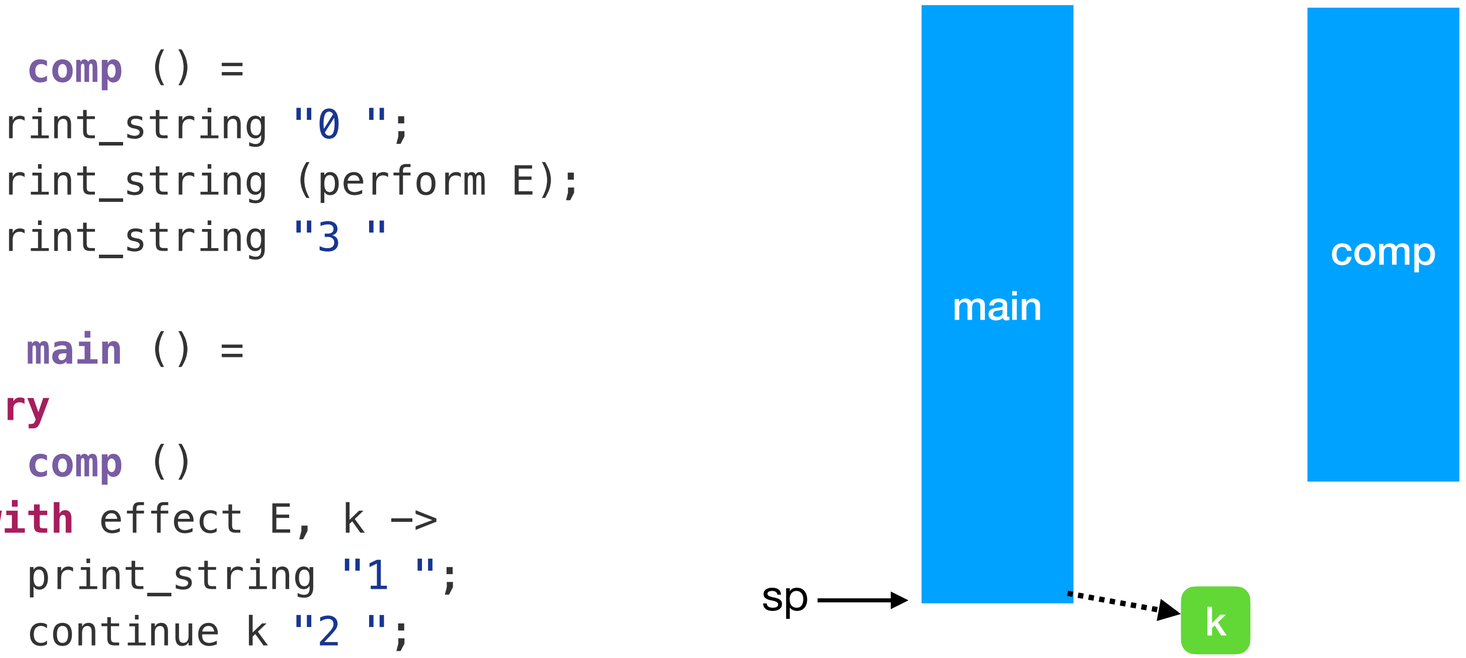
0 1



Effect Handlers — Tracing the execution

```
type _ Effect.t += E : string t

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E, k ->
    print_string "1 ";
    continue k "2 ";
    print_string "4 "
```

pc →



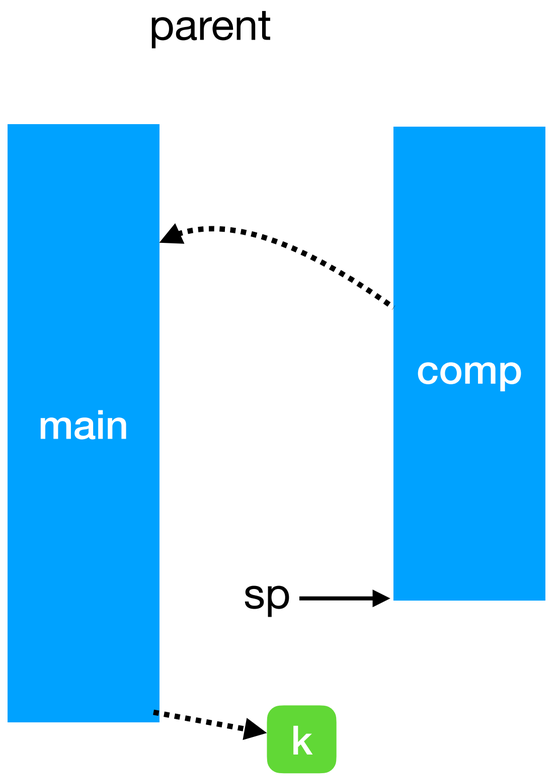
0 1

Effect Handlers — Tracing the execution

```
type _ Effect.t += E : string t

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E, k ->
    print_string "1 ";
    continue k "2 ";
    print_string "4 "
```

pc →

0 1

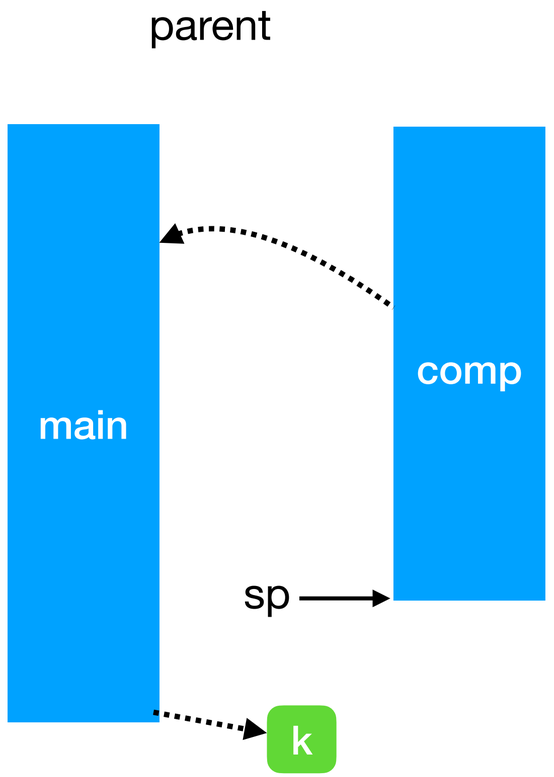


Effect Handlers — Tracing the execution

```
type _ Effect.t += E : string t

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E, k ->
    print_string "1 ";
    continue k "2 ";
    print_string "4 "
```

pc →

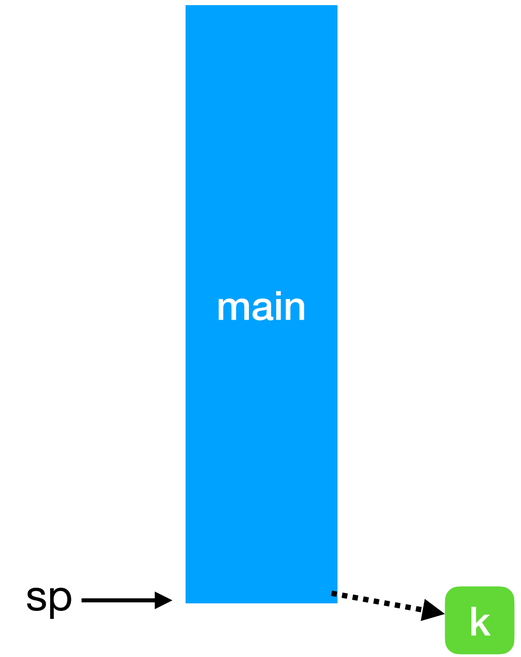
0 1 2



Effect Handlers — Tracing the execution

```
type _ Effect.t += E : string t

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E, k ->
    print_string "1 ";
    continue k "2 ";
    print_string "4 "
```

pc →



0 1 2 3

Effect Handlers — Tracing the execution

```
type _ Effect.t += E : string t

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E, k ->
    print_string "1 ";
    continue k "2 ";
    print_string "4 "
```

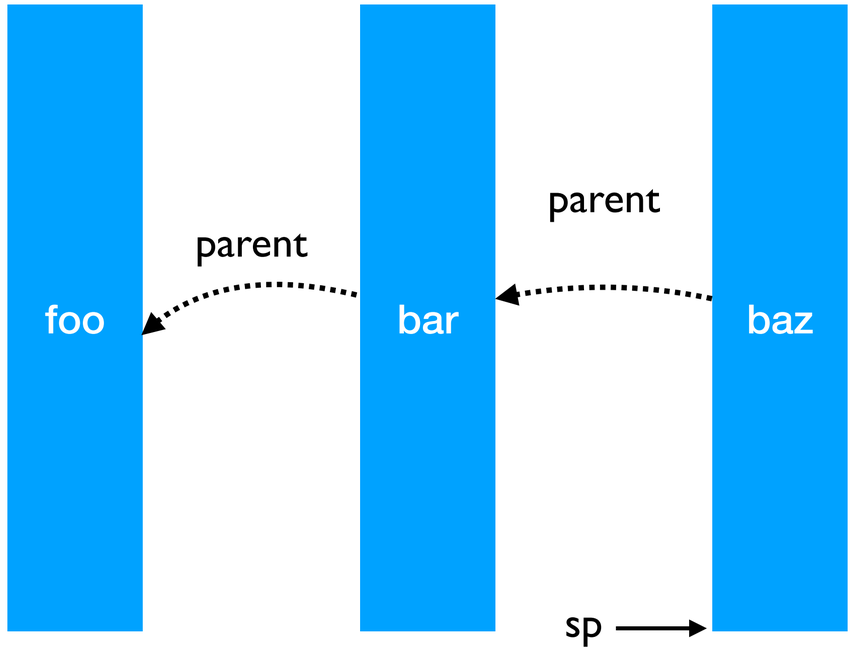
pc →

0 1 2 3 4

Handlers can be nested

```
type _ Effect.t += A : unit t
                 | B : unit t

let baz () =
  perform A

let bar () =
  try
    baz ()
  with effect B, k ->
    continue k ()

let foo () =
  try
    bar ()
  with effect A, k ->
    continue k ()
```

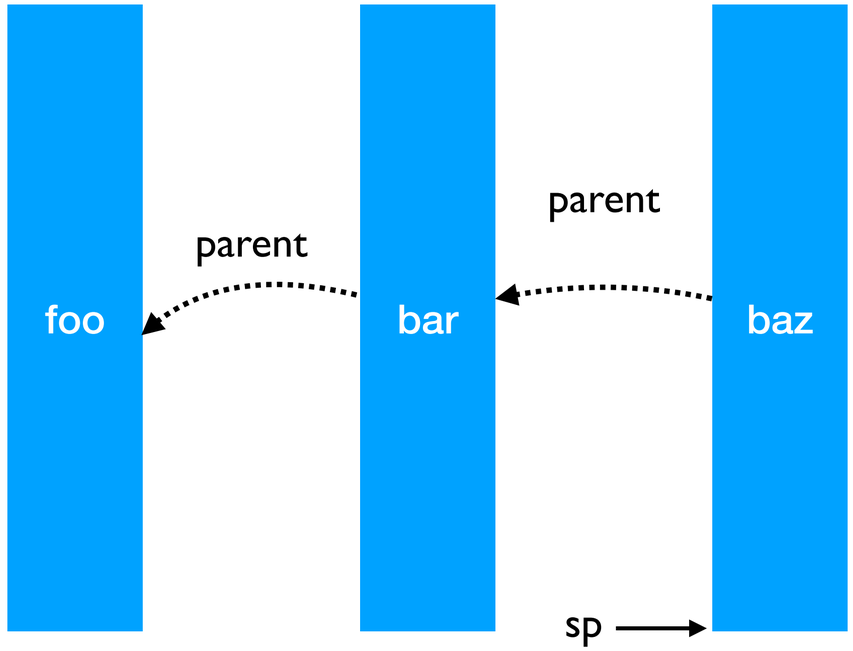
pc →



Handlers can be nested

```
type _ Effect.t += A : unit t
                 | B : unit t

let baz () =
  perform A

let bar () =
  try
    baz ()
  with effect B, k ->
    continue k ()

let foo () =
  try
    bar ()
  with effect A, k ->
    continue k ()
```

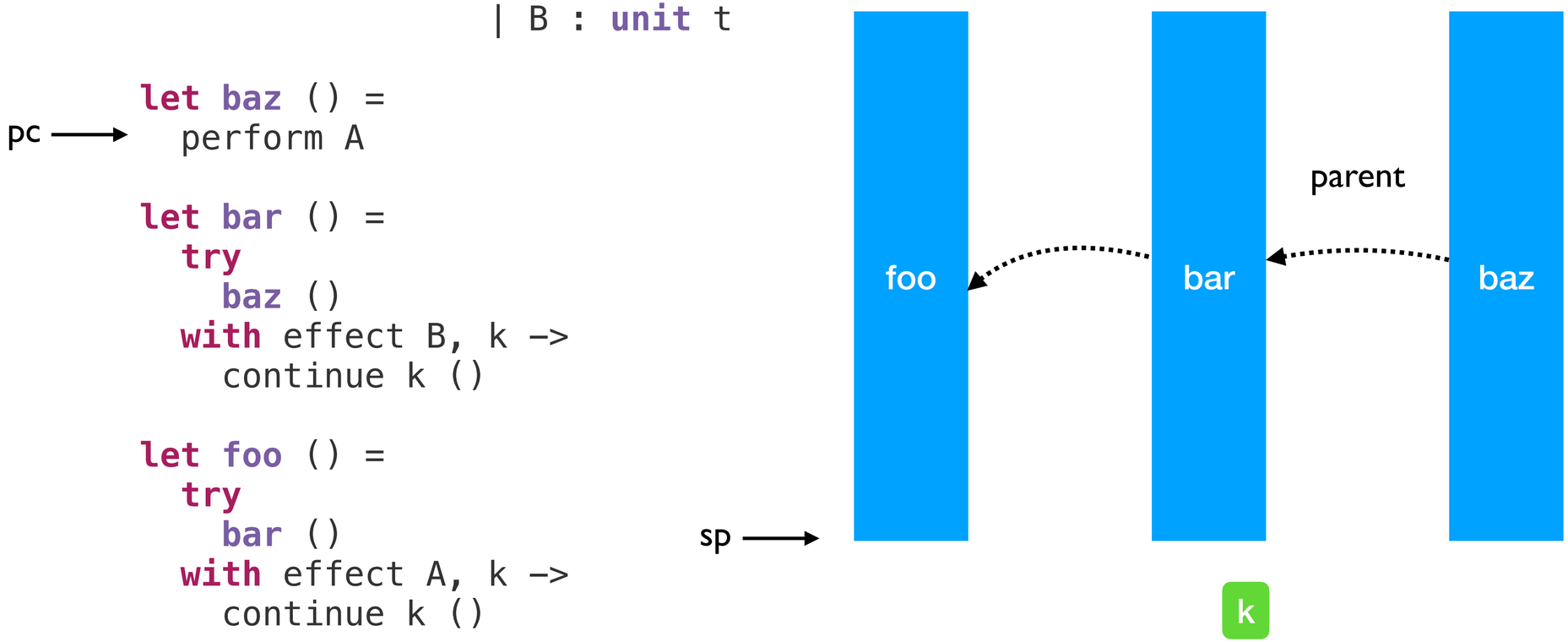
pc →



- Linear search through handlers
  - ✦ *Handler stacks shallow in practice*

Handlers can be nested

```
type _ Effect.t += A : unit t
                 | B : unit t

let baz () =
  perform A

let bar () =
  try
    baz ()
  with effect B, k ->
    continue k ()

let foo () =
  try
    bar ()
  with effect A, k ->
    continue k ()
```

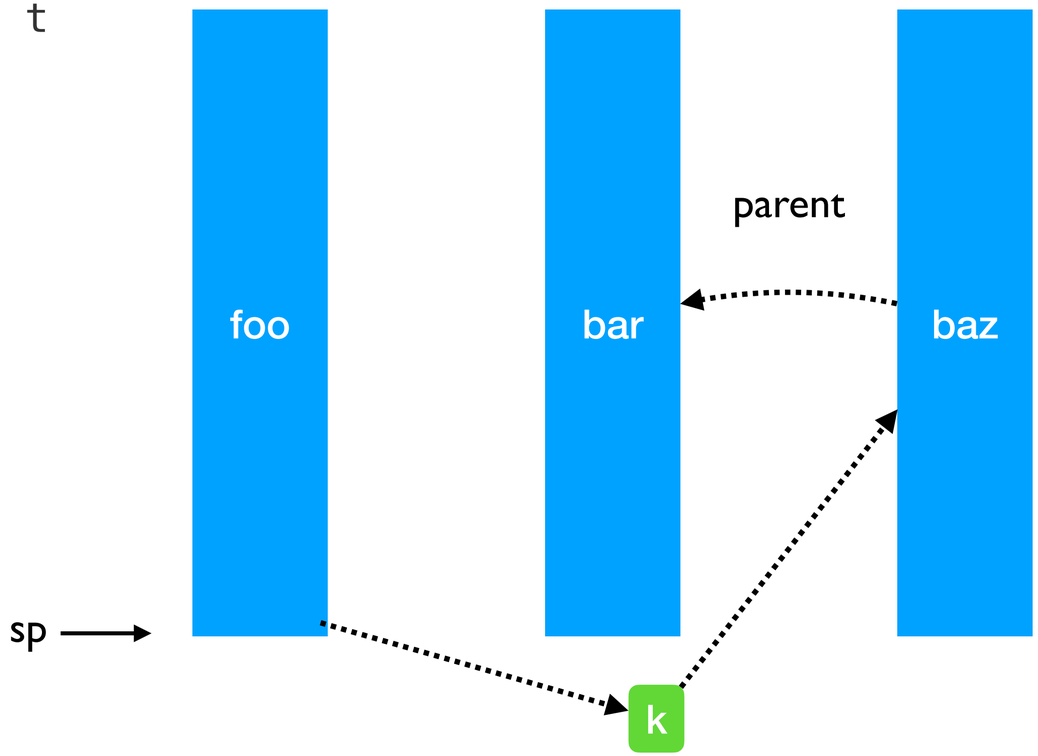


- Linear search through handlers
  - ✦ *Handler stacks shallow in practice*

Handlers can be nested

```
type _ Effect.t += A : unit t
                 | B : unit t

let baz () =
  perform A

let bar () =
  try
    baz ()
  with effect B, k ->
    continue k ()

let foo () =
  try
    bar ()
  with effect A, k ->
    continue k ()
```

pc →



- Linear search through handlers
  - ✦ *Handler stacks shallow in practice*

Handling effects — two ways

```
type _ Effect.t += Ask : string t

let greeting () =
  let name = perform Ask in
  Printf.printf "Hello, %s!\n" name;
  if name = "Bob" then failwith "Bob is not welcome"

(* Way 1: try..with -- handles effects and exceptions.
   The return value of the body is the return value of the whole expression. *)
let _ =
  Printf.printf "--- try..with ---\n";
  try greeting () with
  | effect Ask, k -> continue k "Alice"
  | Failure msg -> Printf.printf "caught exception: %s\n" msg
```

```
--- try..with ---
Hello, Alice!
```

Handling effects — two ways

```
type _ Effect.t += Ask : string t

let greeting () =
  let name = perform Ask in
  Printf.printf "Hello, %s!\n" name;
  if name = "Bob" then failwith "Bob is not welcome"

(* Way 2: match..with -- handles effects, exceptions, AND the return value.
   This gives you a chance to transform or inspect what the body returned. *)
let _ =
  Printf.printf "--- match..with ---\n";
  match greeting () with
  | () -> Printf.printf "greeting returned normally\n"
  | exception (Failure msg) ->
      Printf.printf "caught exception: %s\n" msg
  | effect Ask, k -> continue k "Bob"
```

```
--- match..with ---
Hello, Bob!
caught exception: Bob is not welcome
```

Types

```
module Effect : sig
  type _ t = ..
  (** The type of effects. An extensible variant type; can add more constructors. *)

  val perform : 'a t -> 'a
  (** [perform e] performs the effect [e], which is handled by the nearest enclosing handler for that
      effect. *)

  module Deep : sig
    type ('a, 'b) continuation
    (** The type of continuations for deep handlers.  A value of type [('a, 'b) continuation] represents
        the rest of the computation after the point where an effect is performed, with the ability to
        resume it by providing a value of type ['a] and yielding a result of type ['b]. *)

    val continue : ('a, 'b) continuation -> 'a -> 'b
    (** [continue k v] resumes the continuation [k] with value [v]. *)
    ...
  end
end
```

More Types — Unhandled effects

```
module Effect : sig
  ...
  module Deep : sig
   ...
   val discontinue : ('a, 'b) continuation -> exn -> 'b
    (** [discontinue k exn] resumes the continuation [k]
        by raising [exn]. *)
  end

  exception Unhandled: 'a t -> exn
  (** Raised by [perform] when no handler is found for
      the performed effect. *)

  exception Continuation_already_resumed
  (** Raised by [continue] when the continuation has
      already been resumed. *)
end
```

```
type _ Effect.t += E : unit t
                 | F : unit t

let f () =
  try perform E with
  | Unhandled E ->
      Printf.printf "Caught Unhandled E\n"

let g () =
  try f () with
  | effect F, k ->
      continue k ()

let _ = g ()
```

More Types — Linear Continuations

```
module Effect : sig
  ...
  module Deep : sig
   ...
   val discontinue : ('a, 'b) continuation -> exn -> 'b
    (** [discontinue k exn] resumes the continuation [k]
        by raising [exn]. *)
  end

  exception Unhandled: 'a t -> exn
  (** Raised by [perform] when no handler is found for
      the performed effect. *)

  exception Continuation_already_resumed
  (** Raised by [continue] when the continuation has
      already been resumed. *)
end
```

```
open Effect
open Effect.Deep

type _ Effect.t += E : unit t

let _ =
  try perform E with
  | effect E, k ->
    continue k ();
    continue k () (* Will raise exception *)
```

```
% ocaml test7.ml
Exception: Continuation_already_resumed.
```

Every continuation must be resumed exactly once

More Types — Discontinue with exception

```
module Effect : sig
  ...
  module Deep : sig
   ...
   val discontinue : ('a, 'b) continuation -> exn -> 'b
    (** [discontinue k exn] resumes the continuation [k]
        by raising [exn]. *)
  end

  exception Unhandled: 'a t -> exn
  (** Raised by [perform] when no handler is found for
      the performed effect. *)

  exception Continuation_already_resumed
  (** Raised by [continue] when the continuation has
      already been resumed. *)
end
```

```
type _ Effect.t += E : int t

let _ =
  match perform E with
  | v -> Printf.printf "returned: %d\n" v
  | exception (Invalid_argument msg) ->
      Printf.printf "discontinued with: %s\n" msg
  | effect E, k ->
      discontinue k (Invalid_argument "kapow!")
```

```
% ocaml test8.ml
discontinued with: kapow!
```

Typing the handler block

```
type _ Effect.t += E : 'c -> 'd Effect.t
                 | F : 'e -> 'f Effect.t

match e (* e: 'a *) with
| x (* x: 'a *) ->
   f x (* f: 'a -> 'b; f x: 'b *)
| exception e (* e: exn *) ->
   g e (* g: exn -> 'b; g e: 'b *)
| effect E v (* v: 'c; E v : 'd Effect.t *), k (* ('d,'b) continuation *) ->
   h v k (* h: 'c -> ('d,'b) continuation -> 'b; h v k: 'b *)
| effect F v (* v: 'e; F v : 'f Effect.t *), k (* ('f,'b) continuation *) ->
   i v k (* i: 'e -> ('f,'b) continuation -> 'b; i v k: 'b *)
```

This one takes a bit of getting used to

Effect handlers examples

User-defined effects

- Effect handlers can be used to simulate ***built-in effects*** using ***user-defined effects***
  - Built-in effects — exceptions, state, non-determinism, concurrency, probabilistic programming, etc.

Exceptions

```
(* Exceptions using built-in exceptions *)
exception Foo

let _ =
  try
    perform_some_computation ();
    raise Foo;            (* <-- raise = perform *)
    perform_more_computation ()
  with
  | Foo ->
      handle_foo () (* <-- handler; no continuation *)
```

```
(* Exceptions using effect handlers *)
type _ Effect.t += Foo : unit Effect.t

let _ =
  Try
    perform_some_computation ();
    perform Foo;          (* <-- perform = raise *)
    perform_more_computation ()
  with
  | effect Foo, _k ->
      (* <-- handler; drop continuation; leak! *)
      handle_foo ()
```

Reader effect

```
module Reader (T : sig type t end) : sig
  val get : unit -> T.t
  val run : T.t -> (unit -> 'a) -> 'a
end = struct
  type _ Effect.t += Get : T.t Effect.t

  let get () = perform Get

  let run (env : T.t) f =
    try f () with
    | effect Get, k -> continue k env
end
```

Reader effect

```
module Reader (T : sig type t end) : sig
  val get : unit -> T.t
  val run : T.t -> (unit -> 'a) -> 'a
end = struct
  type _ Effect.t += Get : T.t Effect.t

  let get () = perform Get

  let run (env : T.t) f =
    try f () with
    | effect Get, k -> continue k env
end
```

```
module IntReader = Reader (struct type t = int end)
module StrReader = Reader (struct type t = string end)

let greeting () =
  let name = StrReader.get () in
  let age = IntReader.get () in
  Printf.printf "%s is %d years old\n" name age

let () =
  IntReader.run 30 (fun () ->
    StrReader.run "Alice" greeting)
(* Alice is 30 years old *)

let () =
  IntReader.run 25 (fun () ->
    StrReader.run "Bob" greeting)
(* Bob is 25 years old *)
```

Reader effect

```
module Reader (T : sig type t end) : sig
  val get : unit -> T.t
  val run : T.t -> (unit -> 'a) -> 'a
end = struct
  type _ Effect.t += Get : T.t Effect.t

  let get () = perform Get

  let run (env : T.t) f =
    try f () with
    | effect Get, k -> continue k env
end
```

Reader effect

```
module Reader (T : sig type t end) : sig
  val get : unit -> T.t
  val run : T.t -> (unit -> 'a) -> 'a
end = struct
  type _ Effect.t += Get : T.t Effect.t

  let get () = perform Get

  let run (env : T.t) f =
    try f () with
    | effect Get, k -> continue k env
end
```

```
module Inner = Reader (struct type t = int end)
module Outer = Reader (struct type t = int end)

let comp () =
  let x = Inner.get () in
  let y = Outer.get () in
  Printf.printf "inner = %d, outer = %d\n" x y

let () =
  Outer.run 1 (fun () ->
    Inner.run 2 comp)
(* inner = 2, outer = 1 *)
```

State effect with built-in refs

```
module State (T : sig type t end) : sig
  val get : unit -> T.t
  val set : T.t -> unit
  val run : T.t -> (unit -> 'a) -> T.t * 'a
end = struct
  type _ Effect.t += Get : T.t Effect.t
                   | Set : T.t -> unit Effect.t
  let get () = perform Get
  let set v = perform (Set v)
  let run (init : T.t) f =
    let state = ref init in
    let res =
      try f () with
      | effect Get, k -> continue k !state
      | effect (Set v), k ->
          state := v; continue k ()
    in
    (!state, res)
end
```

State effect with built-in refs

```
module State (T : sig type t end) : sig
  val get : unit -> T.t
  val set : T.t -> unit
  val run : T.t -> (unit -> 'a) -> T.t * 'a
end = struct
  type _ Effect.t += Get : T.t Effect.t
                   | Set : T.t -> unit Effect.t
  let get () = perform Get
  let set v = perform (Set v)
  let run (init : T.t) f =
    let state = ref init in
    let res =
      try f () with
      | effect Get, k -> continue k !state
      | effect (Set v), k ->
          state := v; continue k ()
    in
    (!state, res)
end
```

```
module IS = State (struct type t = int end)

let comp () =
  Printf.printf "initial: %d\n" (IS.get ());
  IS.set 42;
  Printf.printf "after set: %d\n" (IS.get ());
  IS.set 100;
  Printf.printf "after second set: %d\n" (IS.get ())

let () =
  let final_state, () = IS.run 0 comp in
  Printf.printf "final state: %d\n" final_state

(* Output:
   initial: 0
   after set: 42
   after second set: 100
   final state: 100
*)
```

State effect with state-passing (pure)

```
(* State-passing style: purely functional *)
module StateFn (T : sig type t end) : sig
  val get : unit -> T.t
  val set : T.t -> unit
  val run : T.t -> (unit -> 'a) -> T.t * 'a
end = struct
  type _ Effect.t += Get : T.t Effect.t
                   | Set : T.t -> unit Effect.t
  let get () = perform Get
  let set v = perform (Set v)
  let run (init : T.t) f =
    let comp =
      match f () with
      | x -> (fun s -> (s, x))
      | effect Get, k ->
          (fun (s : T.t) -> (continue k s) s)
      | effect (Set v), k ->
          (fun _s -> (continue k ()) v)
    in
    comp init
end
```

See state_trace.md

State effect with state-passing (pure)

```
(* State-passing style: purely functional *)
module StateFn (T : sig type t end) : sig
  val get : unit -> T.t
  val set : T.t -> unit
  val run : T.t -> (unit -> 'a) -> T.t * 'a
end = struct
  type _ Effect.t += Get : T.t Effect.t
                   | Set : T.t -> unit Effect.t
  let get () = perform Get
  let set v = perform (Set v)
  let run (init : T.t) f =
    let comp =
      match f () with
      | x -> (fun s -> (s, x))
      | effect Get, k ->
          (fun (s : T.t) -> (continue k s) s)
      | effect (Set v), k ->
          (fun _s -> (continue k ()) v)
    in
    Comp init
end
```

State effect with state-passing (pure)

```
(* State-passing style: purely functional *)
module StateFn (T : sig type t end) : sig
  val get : unit -> T.t
  val set : T.t -> unit
  val run : T.t -> (unit -> 'a) -> T.t * 'a
end = struct
  type _ Effect.t += Get : T.t Effect.t
                   | Set : T.t -> unit Effect.t
  let get () = perform Get
  let set v = perform (Set v)
  let run (init : T.t) f =
    let comp =
      match f () with
      | x -> (fun s -> (s, x))
      | effect Get, k ->
          (fun (s : T.t) -> (continue k s) s)
      | effect (Set v), k ->
          (fun _s -> (continue k ()) v)
    in
    Comp init
end
```

```
module IS = StateFn (struct type t = int end)

let comp2 () =
  Printf.printf "initial: %d\n" (IS.get ());
  IS.set 42;
  Printf.printf "after set: %d\n" (IS.get ());
  IS.set 100;
  Printf.printf "after second set: %d\n" (IS.get ())

let () =
  let final_state, () = IS.run 0 comp2 in
  Printf.printf "final state: %d\n" final_state

(* Output:
   initial: 0
   after set: 42
   after second set: 100
   final state: 100
*)
```

Control Inversion

- ***Iterator*** and ***generator*** are two ways to traverse a data data structure
  - Iterator — the traversal is controlled by the callee
  - Generator — the traversal is controlled by the caller

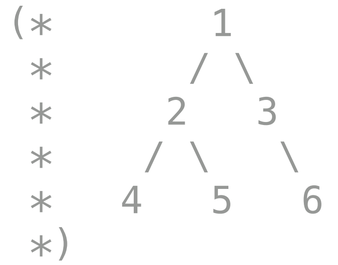
Control Inversion

- ***Iterator*** and ***generator*** are two ways to traverse a data data structure
  - Iterator — the traversal is controlled by the callee
  - Generator — the traversal is controlled by the caller
- Consider the binary tree and its iterator

```
type 'a tree =
  | Leaf
  | Node of 'a tree * 'a * 'a tree

let rec iter t f = match t with
  | Leaf -> ()
  | Node (l, x, r) ->
    iter l f;
    f x;
    iter r f
```

Control Inversion

- ***Iterator*** and ***generator*** are two ways to traverse a data data structure
  - Iterator — the traversal is controlled by the callee
  - Generator — the traversal is controlled by the caller



- Consider the binary tree and its iterator

```
type 'a tree =
  | Leaf
  | Node of 'a tree * 'a * 'a tree

let rec iter t f = match t with
  | Leaf -> ()
  | Node (l, x, r) ->
    iter l f;
    f x;
    iter r f
```

```
  Printf.printf "=== Internal iterator (push-based) ===\n";
  iter example_tree (fun x -> Printf.printf "  visited %d\n" x)
(* Output:
  === Internal iterator (push-based) ===
    visited 4
    visited 2
    visited 5
    visited 1
    visited 3
    visited 6
*)
```

Control Inversion

```
let to_gen (type a) (iter : (a -> unit) -> unit) =
  let module M =
    struct type _ Effect.t += Next : a -> unit Effect.t
  end in
  let open M in
  let rec step = ref (fun () ->
    try
      iter (fun x -> perform (Next x));
      None
    with effect (Next v), k ->
      step := (fun () -> continue k ());
      Some v)
  in
  fun () -> !step ()
```

Control Inversion

```
let to_gen (type a) (iter : (a -> unit) -> unit) =
  let module M =
    struct type _ Effect.t += Next : a -> unit Effect.t
  end in
  let open M in
  let rec step = ref (fun () ->
    try
      iter (fun x -> perform (Next x));
      None
    with effect (Next v), k ->
      step := (fun () -> continue k ());
      Some v)
  in
  fun () -> !step ()
```

```
(*        1
 *       / \
 *      2   3
 *     / \   \
 *    4   5   6
 *)
```

Control Inversion

```
let to_gen (type a) (iter : (a -> unit) -> unit) =
  let module M =
    struct type _ Effect.t += Next : a -> unit Effect.t
  end in
  let open M in
  let rec step = ref (fun () ->
    try
      iter (fun x -> perform (Next x));
      None
    with effect (Next v), k ->
      step := (fun () -> continue k ());
      Some v)
  in
  fun () -> !step ()
```

```
(*       1
 *      / \
 *     2   3
 *    / \   \
 *   4   5   6
 *)
```

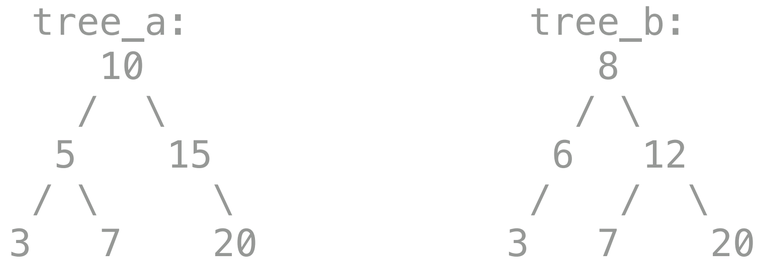
```
let next = to_gen (iter example_tree);;
next ();; (* Some 4 *)
next ();; (* Some 2 *)
next ();; (* Some 5 *)
next ();; (* Some 1 *)
next ();; (* Some 3 *)
next ();; (* Some 6 *)
next ();; (* None *)
```

Control Inversion — Same Fringe Problem

- Fringe — leaf values from left to right
- “Lazy” solution using control inversion

Control Inversion — Same Fringe Problem

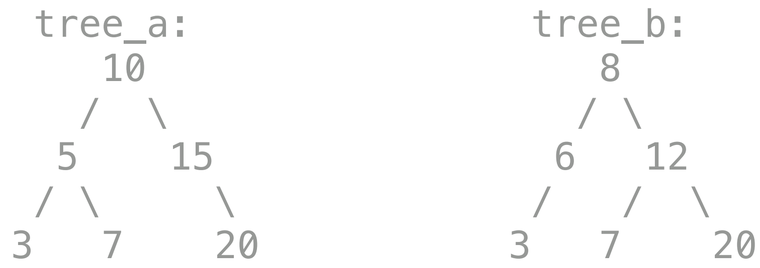
- Fringe — leaf values from left to right
- “Lazy” solution using control inversion




```
(* Iterate over fringe (leaf) elements only *)
let rec iter_leaves t f = match t with
  | Leaf -> ()
  | Node (Leaf, x, Leaf) -> f x
  | Node (l, _, r) ->
    iter_leaves l f;
    iter_leaves r f
```

Control Inversion — Same Fringe Problem

- Fringe — leaf values from left to right
- “Lazy” solution using control inversion




```
(* Iterate over fringe (leaf) elements only *)
let rec iter_leaves t f = match t with
  | Leaf -> ()
  | Node (Leaf, x, Leaf) -> f x
  | Node (l, _, r) ->
    iter_leaves l f;
    iter_leaves r f
```

```
let same_fringe t1 t2 =
  let next1 = to_gen (iter_leaves t1) in
  let next2 = to_gen (iter_leaves t2) in
  let rec go () =
    match next1 (), next2 () with
    | None, None -> true
    | Some v1, Some v2 -> v1 = v2 && go ()
    | _ -> false
  in
  go ()
```

Control Inversion

```
let to_gen (type a) (iter : (a -> unit) -> unit) =
  let module M =
    struct type _ Effect.t += Next : a -> unit Effect.t
  end in
  let open M in
  let rec step = ref (fun () ->
    try
      iter (fun x -> perform (Next x));
      None
    with effect (Next v), k ->
      step := (fun () -> continue k ());
      Some v)
  in
  fun () -> !step ()
```

```
  let numbers n yield =
    yield 0;
    for i = 1 to n do
      yield i; yield (-i)
    done;;
  let next = to_gen (numbers 10);;
  next ();;
  next ();;
  next ();;
  next ();;
  ...
(* Output:
  === Python-style generator ===
    Some 0
    Some 1
    Some -1
    Some 2
    ...
*)
```

Lightweight Concurrency

Lightweight Concurrency

- Concurrency is a program structuring mechanism
  - 1 (lightweight) thread per request in a web server
  - 1 (lightweight) thread per computational unit in a nested parallel program

Lightweight Concurrency

- Concurrency is a program structuring mechanism
  - 1 (lightweight) thread per request in a web server
  - 1 (lightweight) thread per computational unit in a nested parallel program
- Domains are too heavy

Lightweight Concurrency

- Concurrency is a program structuring mechanism
  - 1 (lightweight) thread per request in a web server
  - 1 (lightweight) thread per computational unit in a nested parallel program
- Domains are too heavy
- Describe concurrency and transparently run it on top of multiple domains
  - GHC Haskell, Go, Java Virtual Threads, Erlang OTP, etc.

Lightweight Concurrency

- Concurrency is a program structuring mechanism
  - 1 (lightweight) thread per request in a web server
  - 1 (lightweight) thread per computational unit in a nested parallel program
- Domains are too heavy
- Describe concurrency and transparently run it on top of multiple domains
  - GHC Haskell, Go, Java Virtual Threads, Erlang OTP, etc.
- Also known as M:N scheduling
  - M lightweight threads / fibers on N hardware threads / cores
  - M >> N

Lightweight Concurrency

```
(** A lightweight cooperative scheduler using effect handlers. *)

val fork : (unit -> unit) -> unit
(** [fork f] spawns [f] as a new concurrent task. *)

val yield : unit -> unit
(** [yield ()] suspends the current task and schedules the next one. *)

val run : (unit -> unit) -> unit
(** [run main] runs [main] and all forked tasks in round-robin order. *)
```

Lightweight Concurrency

```
type _ Effect.t
  += Fork : (unit -> unit) -> unit Effect.t
   | Yield : unit Effect.t

let fork f = perform (Fork f)
let yield () = perform Yield
```

```
(* A concurrent round-robin scheduler *)
let run main =
  let run_q = Queue.create () in
  let enqueue k = Queue.push k run_q in
  let dequeue () =
    if Queue.is_empty run_q then ()
    else continue (Queue.pop run_q) ()
  in
  let rec spawn f =
    match f () with
    | () -> dequeue ()
    | exception e ->
        print_string (Printexc.to_string e);
        dequeue ()
    | effect Yield, k -> enqueue k; dequeue ()
    | effect (Fork f), k -> enqueue k; spawn f
  in
  spawn main
```

Lightweight Concurrency

```
let () =
  run (fun () ->
    fork (fun () ->
      for i = 1 to 3 do
        Printf.printf "Task A: %d\n" i;
        yield ()
      done);
    fork (fun () ->
      for i = 1 to 3 do
        Printf.printf "Task B: %d\n" i;
        yield ()
      done);
    for i = 1 to 3 do
      Printf.printf "Main  : %d\n" i;
      yield ()
    done)
```

```
Output:
   Task A: 1
   Task B: 1
   Task A: 2
   Main  : 1
   Task B: 2
   Task A: 3
   Main  : 2
   Task B: 3
   Main  : 3
```

“Lightweight” Concurrency

- Comparing domain spawn+join vs fork
- Note, our current lightweight concurrency doesn’t have parallelism

```
% dune exec ./tests/domains_vs_fibers.exe
Entering directory '/Users/kc/teaching/cs6868/cs6868_s26'
Leaving directory '/Users/kc/teaching/cs6868/cs6868_s26'
Lightweight threads (50,000,000): 2.716 s
Domain spawn+join   (50,000): 2.293 s

Throughput:
  Lightweight: 18,411,419 threads/s
  Domains:     21,803 spawn-joins/s
  Ratio:       844.4x
```

Many other examples

https://github.com/ocaml-multicore/effects-examples

- Transactions
- Memoization
- Probabilistic programming
- Non-determinism
- …

Fin