

Linguagens e Ambientes de Programação

(Aula Teórica 15)

LEI - Licenciatura em Engenharia Informática

João Costa Seco (joao.seco@fct.unl.pt)

Agenda

- Concorrência
- I/O assíncrono, Promessas e Futuros

Concorrência

- Capacidade de realizar várias computações que se sobrepõem no tempo, e não exigir que sejam feitas em sequência.
 - Interfaces gráficos; para assegurar respostas rápida a acções do utilizador.
 - Folhas de cálculo; para recalcular todos os cálculos sem interrupções.
 - Web Browser; para carregar e mostrar páginas de forma incremental.
 - Servidores; para atender vários clientes sem os fazer esperar.
- Como?
 - Entrelaçamento: comutando entre as várias computações ativas rapidamente.
 - Paralelismo: utilizando vários processadores físicos (multi-core).
- Preemptive / Collaborative

Não determinismo

- Um programa que pode ter resultados diferentes de cada vez que é executado é um programa não determinista.
- A introdução de concorrência causa não determinismo, ao não fixar a ordem pela qual operações são executadas.
- Quando dois programas imperativos partilham variáveis de estado, as possibilidades de alteração desse mesmo estado são indeterminadas. Logo, não é fácil prever o comportamento exacto dos programas.
- As interações/interferências entre programas são benignas ou malignas (race conditions)
- As linguagens funcionais tornam mais fácil o raciocínio sobre programas porque uma expressão pura denota sempre o mesmo valor.

Promessas ou Futuros

- Representam computações que ainda não acabaram mas que irão denotar um valor algures num instante futuro (diferido).
- Normalmente associadas a uma computação concorrente ao thread principal.

```
async function getNames() {  
    const response = await fetch('https://server.com/users')  
    const data = await response.json()  
    return data.map(user => user.name)  
}
```

- Async (Jane Street) e Lwt (Ocsigen) são duas bibliotecas populares para implementar computação assíncrona em OCaml.

Promessas ao estilo Lwt

- São abstrações de dados para um modelo de computação assíncrona.
- As promessas são referências, o seu valor pode mudar.
- Quando é criada não contém nada.
- Uma promessa pode ser cumprida e preenchida por um valor
- Uma promessa pode ser rejeitada (preenchida por uma exceção)
- Em ambos os casos diz-se resolvida.

Assinatura do Módulo Promessa

```
▷ ▾ (** A signature for Lwt-style promises, with better names *)
module type PROMISE = sig
  type 'a state =
    | Pending
    | Fulfilled of 'a
    | Rejected of exn

  type 'a promise

  type 'a resolver

  (** [make ()] is a new promise and resolver. The promise is pending. *)
  val make : unit → 'a promise * 'a resolver

  (** [return x] is a new promise that is already fulfilled with value
  | | [ x ]. *)
  val return : 'a → 'a promise

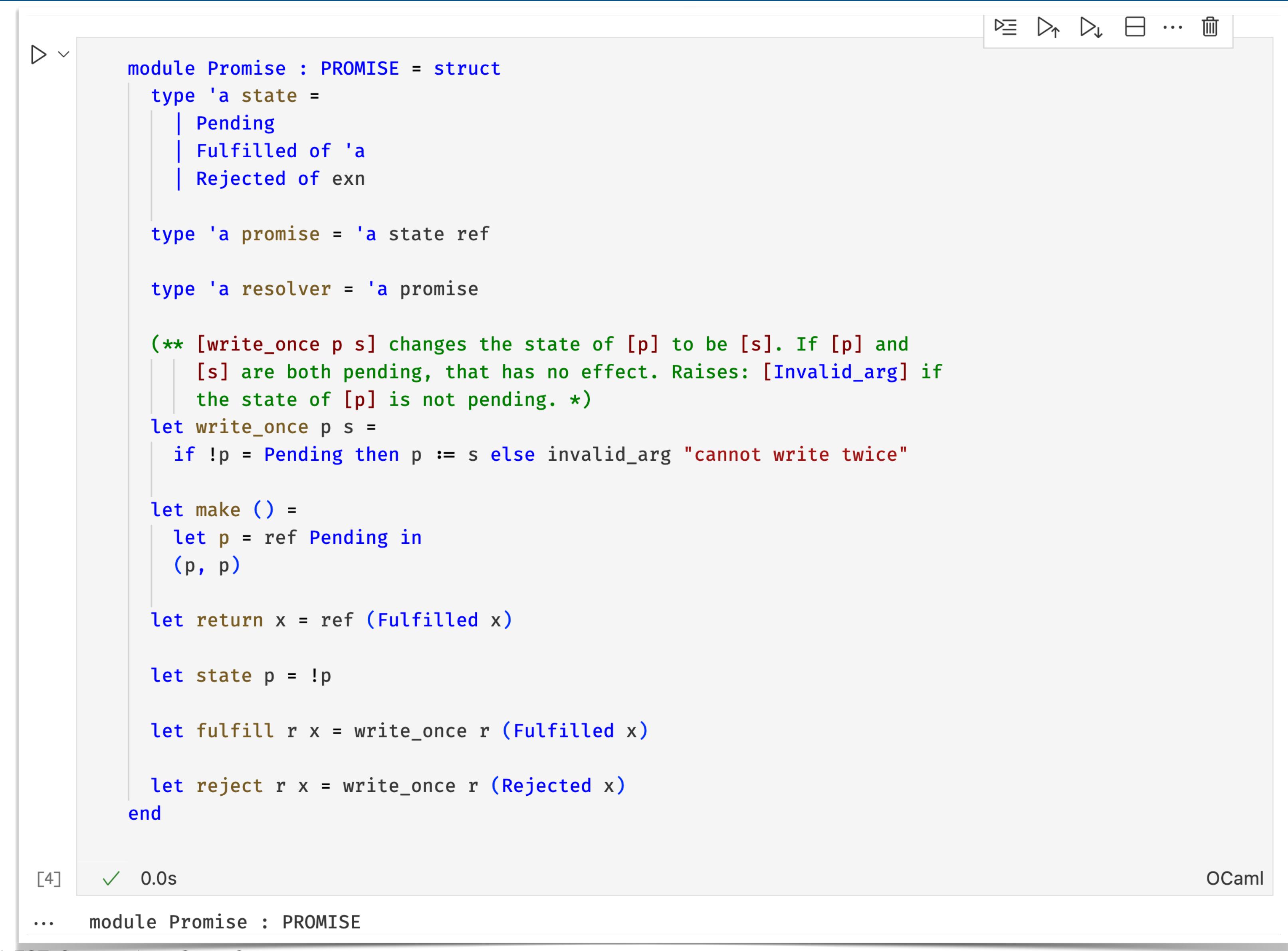
  (** [state p] is the state of the promise *)
  val state : 'a promise → 'a state

  (** [fulfill r x] fulfills the promise [p] associated with [r] with
  | | value [ x ], meaning that [state p] will become [Fulfilled x].
  | | Requires: [p] is pending. *)
  val fulfill : 'a resolver → 'a → unit

  (** [reject r x] rejects the promise [p] associated with [r] with
  | | exception [ x ], meaning that [state p] will become [Rejected x].
  | | Requires: [p] is pending. *)
  val reject : 'a resolver → exn → unit
end

[1] ✓ 0.0s OCaml
```

Implementação do Módulo Promessa



The screenshot shows an OCaml code editor window with the following code:

```
module Promise : PROMISE = struct
  type 'a state =
    | Pending
    | Fulfilled of 'a
    | Rejected of exn

  type 'a promise = 'a state ref

  type 'a resolver = 'a promise

  (** [write_once p s] changes the state of [p] to be [s]. If [p] and
   * [s] are both pending, that has no effect. Raises: [Invalid_arg] if
   * the state of [p] is not pending. *)
  let write_once p s =
    if !p = Pending then p := s else invalid_arg "cannot write twice"

  let make () =
    let p = ref Pending in
    (p, p)

  let return x = ref (Fulfilled x)

  let state p = !p

  let fulfill r x = write_once r (Fulfilled x)

  let reject r x = write_once r (Rejected x)
end
```

The code defines a module `Promise` with the following structure:

- Type definitions:
 - `'a state`: A discriminated union with three cases: `Pending`, `Fulfilled of 'a`, and `Rejected of exn`.
 - `'a promise`: A reference to a `'a state`.
 - `'a resolver`: A function that takes a `'a promise` as an argument.
- Implementation functions:
 - `write_once`: Changes the state of a promise. If the current state is `Pending`, it updates it to the new state `s`. Otherwise, it raises `invalid_arg "cannot write twice"`.
 - `make`: Creates a new promise by initializing its state to `Pending` and returning a tuple of the promise and its resolver.
 - `return`: Creates a fulfilled promise by initializing its state to `Fulfilled x`.
 - `state`: Returns the current state of a promise.
 - `fulfill`: Sets the state of a promise to `Fulfilled x`.
 - `reject`: Sets the state of a promise to `Rejected x`.

The code is annotated with a multi-line comment explaining the behavior of `write_once`.

At the bottom of the editor, there is a status bar showing [4], a green checkmark icon, 0.0s, and OCaml.

I/O Síncrona



```
let file = "example.dat"
let message = "Hello!"

let () =
  let oc = open_out file in
  Printf.printf oc "%s\n" message;
  close_out oc;

  let ic = open_in file in
  try
    let line = input_line ic in
    print_endline line;
    flush stdout;
    close_in ic
  with e →
    close_in_noerr ic;
    raise e
```



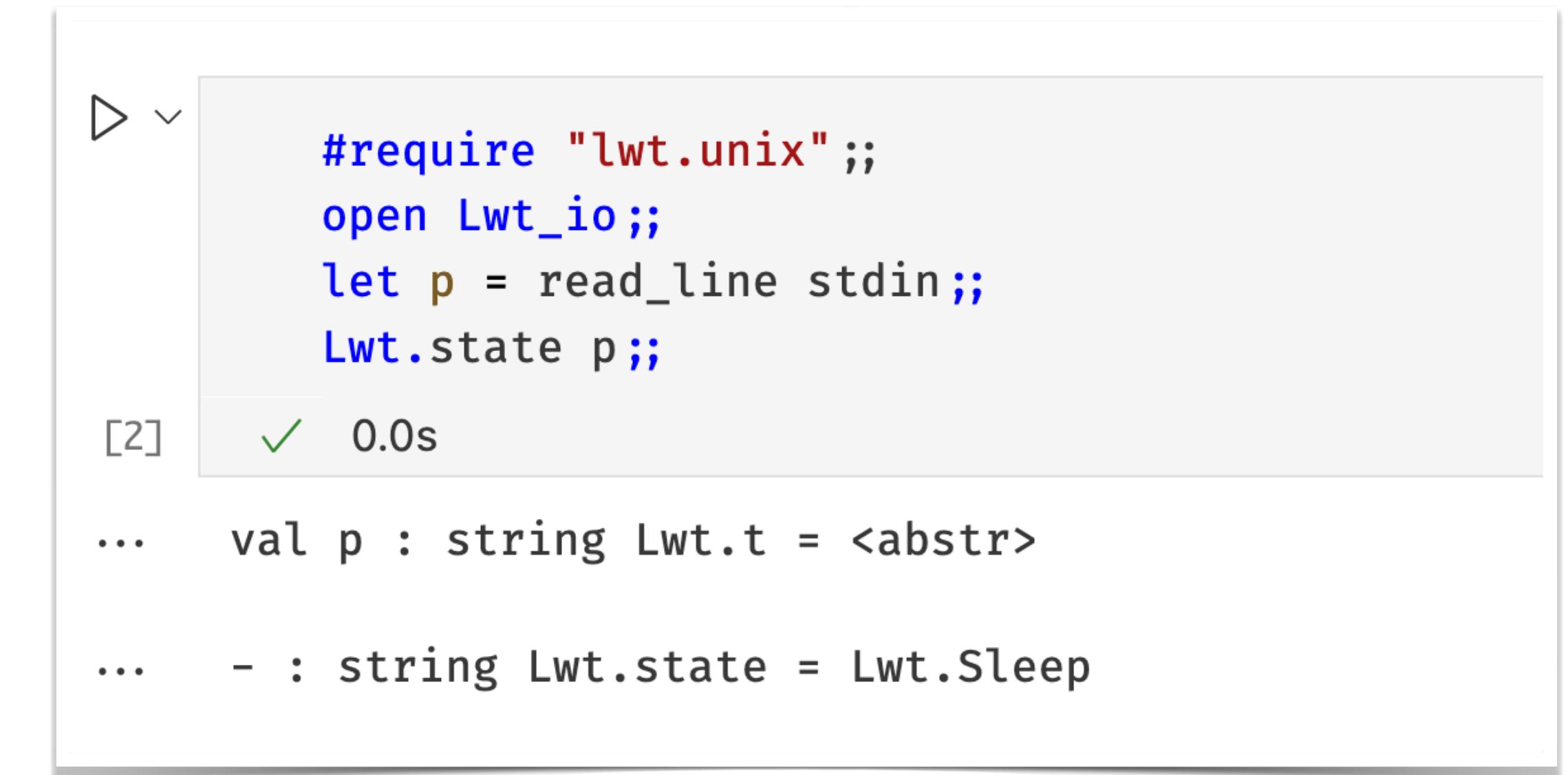
```
# ignore(input_line stdin); print_endline "done";;
<type your own input here>
done
- : unit = ()
```

<https://ocaml.org/docs/file-manipulation>

<https://cs3110.github.io/textbook/chapters/ds/promises.html>

I/O Assíncrona

- As primitivas de I/O não bloqueiam à espera que termine a operação de I/O.
- O valor que denotam é uma promessa.
- O tipo da promessa é mascarado pelo utop mas é de facto **string Lwt.t**



A screenshot of an OCaml utop session. The code is:

```
#require "lwt.unix";;
open Lwt_io;;
let p = read_line stdin;;
Lwt.state p;;
```

The output shows the promise has been fulfilled with a string value:

[2] ✓ 0.0s

... val p : string Lwt.t = <abstr>

... - : string Lwt.state = Lwt.Sleep

A notação desta imagem é a notação nativa do módulo lwt

Callbacks

- Uma função callback de uma promessa é uma função que é chamada aquando da realização da mesma.
- Liga-se uma função callback a uma promessa através da função **bind**.

```
▶ let print_the_string str = Lwt_io.printf "The string is: %S\n" str;;
      Lwt.bind p print_the_string
[3]
... val print_the_string : string → unit Lwt.t = <fun>
```

```
▶ Lwt.bind
[3] ✓ 0.0s
... - : 'a Lwt.t → ('a → 'b Lwt.t) → 'b Lwt.t = <fun>
```

Callbacks 2

- Várias operações assíncronas podem ser encadeadas, para manter o determinismo/sequentialidade.
- A função **bind** combina a função **run** espera pelo total resultado e devolve o valor final.

```
open Lwt_io

let p =
  Lwt.bind (read_line stdin) (fun s1 →
    Lwt.bind (read_line stdin) (fun s2 →
      Lwt_io.printf "%s\n" (s1^s2)))

let _ = Lwt_main.run p
```

[4]

```
▷ Lwt.bind;;
Lwt_main.run
[5] ✓ 0.0s
... - : 'a Lwt.t → ('a → 'b Lwt.t) → 'b Lwt.t = <fun>
... - : 'a Lwt.t → 'a = <fun>
```

```
utop # #use "teoricas/LAP2024-15/promises.ml";;
val p : unit Lwt.t = <abstr>
one
two
- : unit = ()
onetwo
```

Callbacks 2

- Várias operações assíncronas podem ser encadeadas, para manter o determinismo/sequentialidade.
- A função **bind** combina a função **run** espera pelo total resultado e devolve o valor final.

```
▷ Lwt.bind;;
[5]   Lwt_main.run
      ✓ 0.0s
...   - : 'a Lwt.t → ('a → 'b Lwt.t) → 'b Lwt.t = <fun>
...   - : 'a Lwt.t → 'a = <fun>
```

```
>>=
▷ open Lwt_io
open Lwt.Infix

let p =
  read_line stdin ≫ fun s1 →
  read_line stdin ≫ fun s2 →
  Lwt_io.printf "%s\n" (s1^s2)

let _ = Lwt_main.run p
[]
```

```
utop # #use "teoricas/LAP2024-15/promises.ml";;
val p : unit Lwt.t = <abstr>
one
two
- : unit = ()
onetwo
```

Callbacks 2

- Várias operações assíncronas podem ser encadeadas, para manter o determinismo/sequentialidade.
- A função `bind` combina a função `read_line` com a função `getNames`, esperando pelo total resultado e devolvendo o valor final.

```
▶ Lwt.bind;;
  Lwt_main.run
[5]   ✓ 0.0s
...   - : 'a Lwt.t → ('a → 'b Lwt.t) → 'b Lwt.t = <fun>
...   - : 'a Lwt.t → 'a = <fun>
```

```
>>=
▶ open Lwt_io
open Lwt.Infix

let p =
  read_line stdin ≫ fun s1 →
  read_line stdin ≫ fun s2 →
    function getNames() {
      return fetch('https://server.com/users')
        .then(response ⇒ response.json())
        .then(data ⇒ data.map(user ⇒ user.name));
    }
  getNames().then(names ⇒ console.log(names));
-
```

JavaScript

Callbacks implemented

```
(** A signature for Lwt-style promises, with better names *)
module type PROMISE = sig
  type 'a state =
    | Pending
    | Fulfilled of 'a
    | Rejected of exn

  type 'a promise

  type 'a resolver

  (** [make ()] is a new promise and resolver. The promise is pending. *)
  unit -> 'a promise * 'a resolver
  val make : unit → 'a promise * 'a resolver

  (** [return x] is a new promise that is already fulfilled with value [ x ]. *)
  'a -> 'a promise
  val return : 'a → 'a promise

  (** [state p] is the state of the promise *)
  'a promise -> 'a state
  val state : 'a promise → 'a state

  (** [fulfill r x] resolves the promise [p] associated with [r] with
      value [ x ], meaning that [state p] will become [Fulfilled x].
      Requires: [p] is pending. *)
  'a resolver -> 'a -> unit
  val fulfill : 'a resolver → 'a → unit

  (** [reject r x] rejects the promise [p] associated with [r] with
      exception [ x ], meaning that [state p] will become [Rejected x].
      Requires: [p] is pending. *)
  'a resolver -> exn -> unit
  val reject : 'a resolver → exn → unit

  (** [p >= c] registers callback [c] with promise [p].
      When the promise is fulfilled, the callback will be run
      on the promises's contents. If the promise is never
      fulfilled, the callback will never run. *)
  'a promise -> ('a -> 'b promise) -> 'b promise
  val ( >= ) : 'a promise → ('a → 'b promise) → 'b promise
end
```

<https://cs3110.github.io/textbook/chapters/ds/promises.html>

Callbacks implemented

```
module Promise : PROMISE = struct
  type 'a state = Pending | Fulfilled of 'a | Rejected of exn

  (** RI (representation invariant): the input may not be [Pending] *)
  type 'a handler = 'a state → unit

  (** RI: if [state ◁ Pending] then [handlers = []]. *)
  type 'a promise = {
    mutable state : 'a state;
    mutable handlers : 'a handler list
  }
```

Callbacks implemented

```
'a state -> unit) -> 'a promise -> unit
let enqueue
  (handler : 'a state → unit)
  (promise : 'a promise) : unit
  =
  promise.handlers ← handler :: promise.handlers

type 'a resolver = 'a promise

(** [write_once p s] changes the state of [p] to be [s]. If [p] and [s]
   are both pending, that has no effect.
   Raises: [Invalid_arg] if the state of [p] is not pending. *)
'a resolver -> 'a state -> unit
let write_once p s =
  if p.state = Pending
  then p.state ← s
  else invalid_arg "cannot write twice"
```

Callbacks implemented

```
unit -> 'a resolver * 'a resolver
let make () =
  let p = {state = Pending; handlers = []} in
  p, p

'a -> 'a resolver
let return x =
  {state = Fulfilled x; handlers = []}

'a resolver -> 'a state
let state p = p.state

(** requires: [st] may not be [Pending] *)
'a resolver -> 'a state -> unit
let resolve (r : 'a resolver) (st : 'a state) =
  assert (st  $\neq$  Pending);
  let handlers = r.handlers in
  r.handlers  $\leftarrow$  [];
  write_once r st;
  List.iter (fun f  $\rightarrow$  f st) handlers
```

Callbacks implemented

```
'a resolver -> exn -> unit
let reject r x =
    resolve r (Rejected x)

'a resolver -> 'a -> unit
let fulfill r x =
    resolve r (Fulfilled x)

'a resolver -> 'a handler
let handler (resolver : 'a resolver) : 'a handler
    = function
        | Pending → failwith "handler RI violated"
        | Rejected exc → reject resolver exc
        | Fulfilled x → fulfill resolver x
```

Callbacks implemented

```
('a -> 'b resolver) -> 'b resolver -> 'a handler
let handler_of_callback
  (callback : 'a → 'b promise)
  (resolver : 'b resolver) : 'a handler
= function
  | Pending → failwith "handler RI violated"
  | Rejected exc → reject resolver exc
  | Fulfilled x →
    let promise = callback x in
    match promise.state with
    | Fulfilled y → fulfill resolver y
    | Rejected exc → reject resolver exc
    | Pending → enqueue (handler resolver) promise

'a resolver -> ('a -> 'b resolver) -> 'b resolver
let ( ≫= )
  (input_promise : 'a promise)
  (callback : 'a → 'b promise) : 'b promise
=
  match input_promise.state with
  | Fulfilled x → callback x
  | Rejected exc → {state = Rejected exc; handlers = []}
  | Pending →
    let output_promise, output_resolver = make () in
    enqueue (handler_of_callback callback output_resolver) input_promise;
    output_promise
```

end

Linguagens e Ambientes de Programação (Aula Teórica 16)

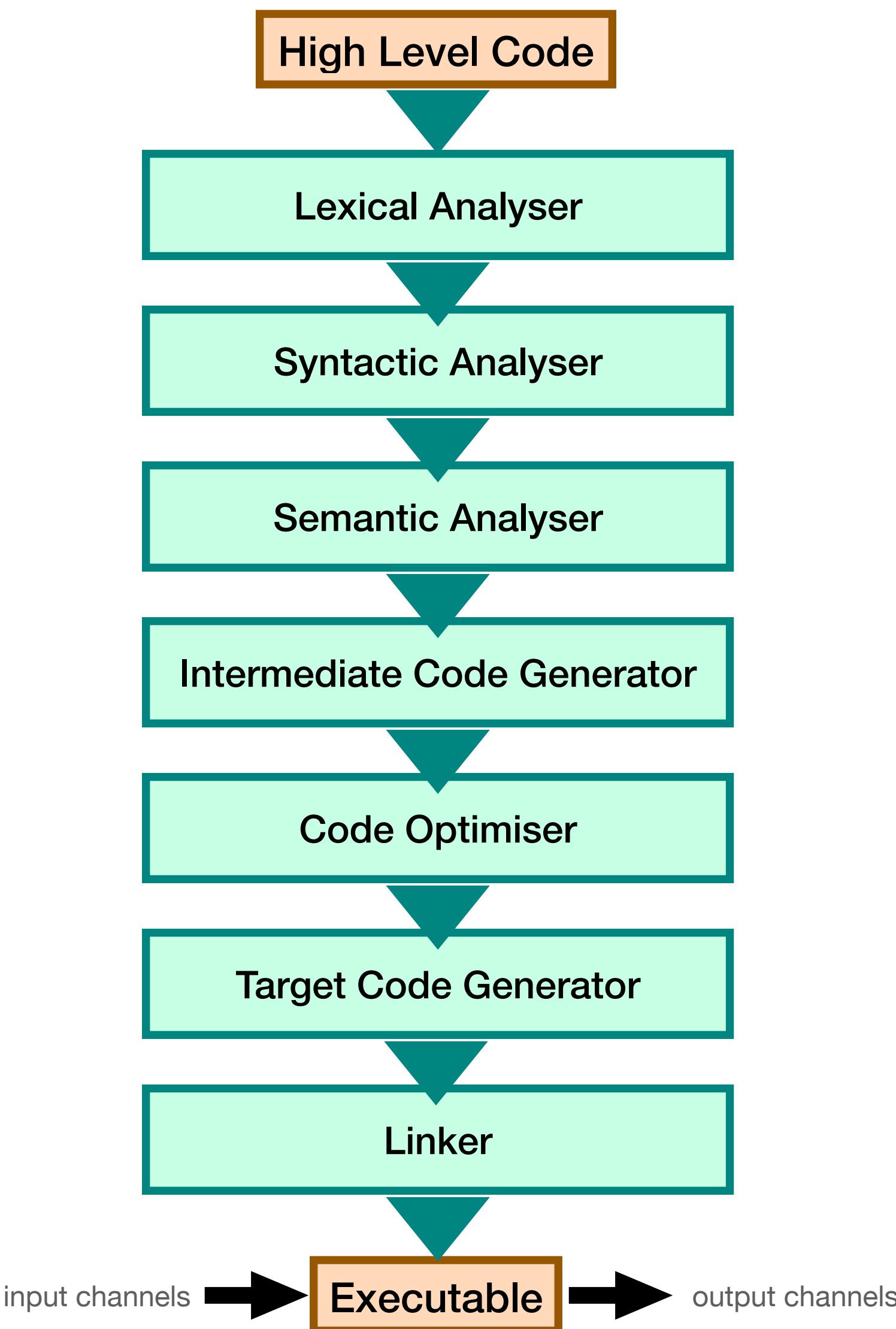
LEI - Licenciatura em Engenharia Informática

João Costa Seco (joao.seco@fct.unl.pt)

Código como dados (simplificado)

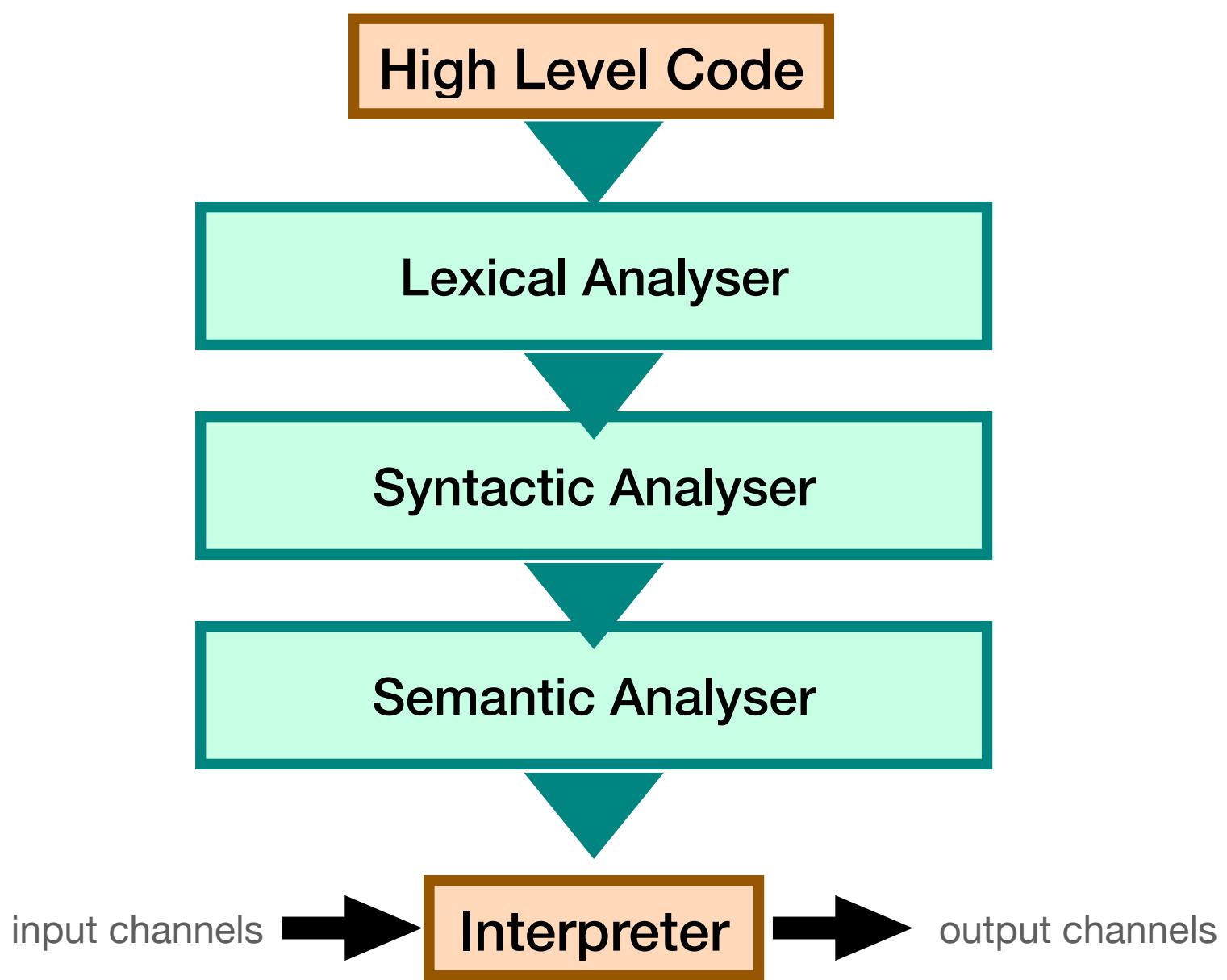
- Compiladores (de código fonte para código máquina, executável)
- Interpretadores (execução de código fonte)
- Geradores de código (de especificações para código fonte)
- Plataformas baseadas em modelos (modelos para código fonte ou execução)
- Analisadores estáticos de código (de código fonte e especificações para verificação de propriedades)
- Correção, segurança, performance, etc.

Compiladores



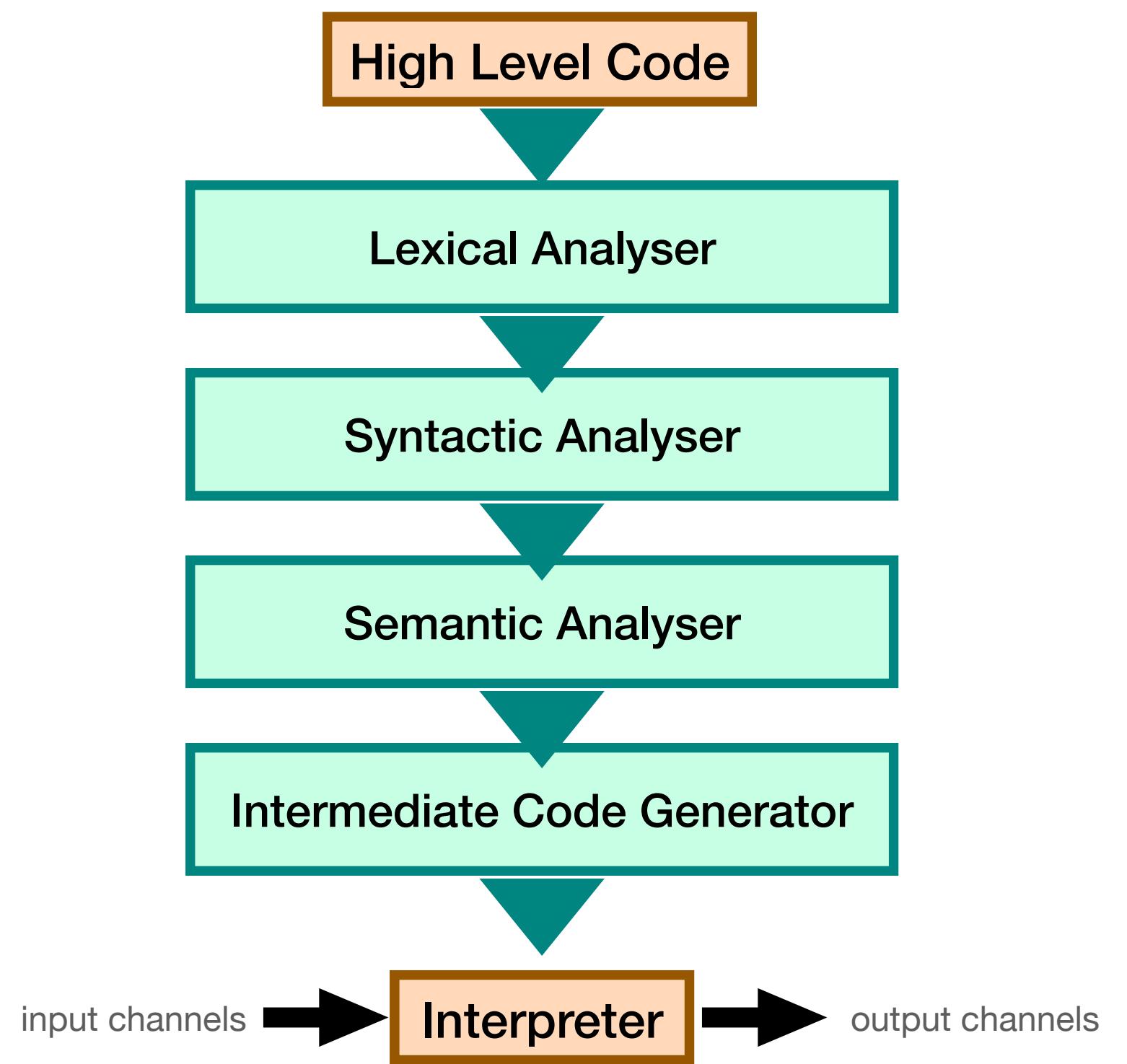
<https://www.geeksforgeeks.org/phases-of-a-compiler/>

Interpretadores



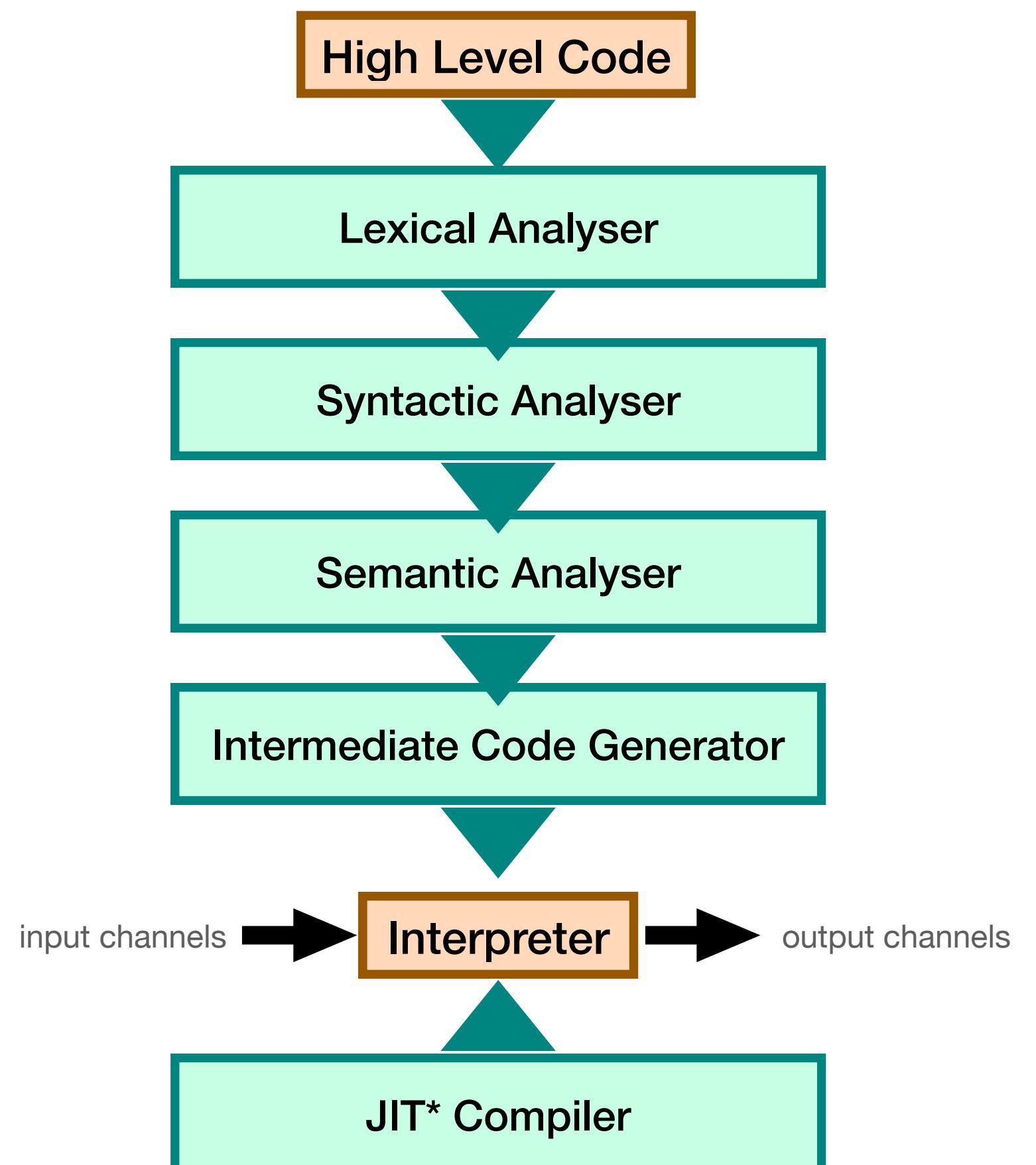
<https://www.geeksforgeeks.org/phases-of-a-compiler/>

Interpretadores com código intermédio



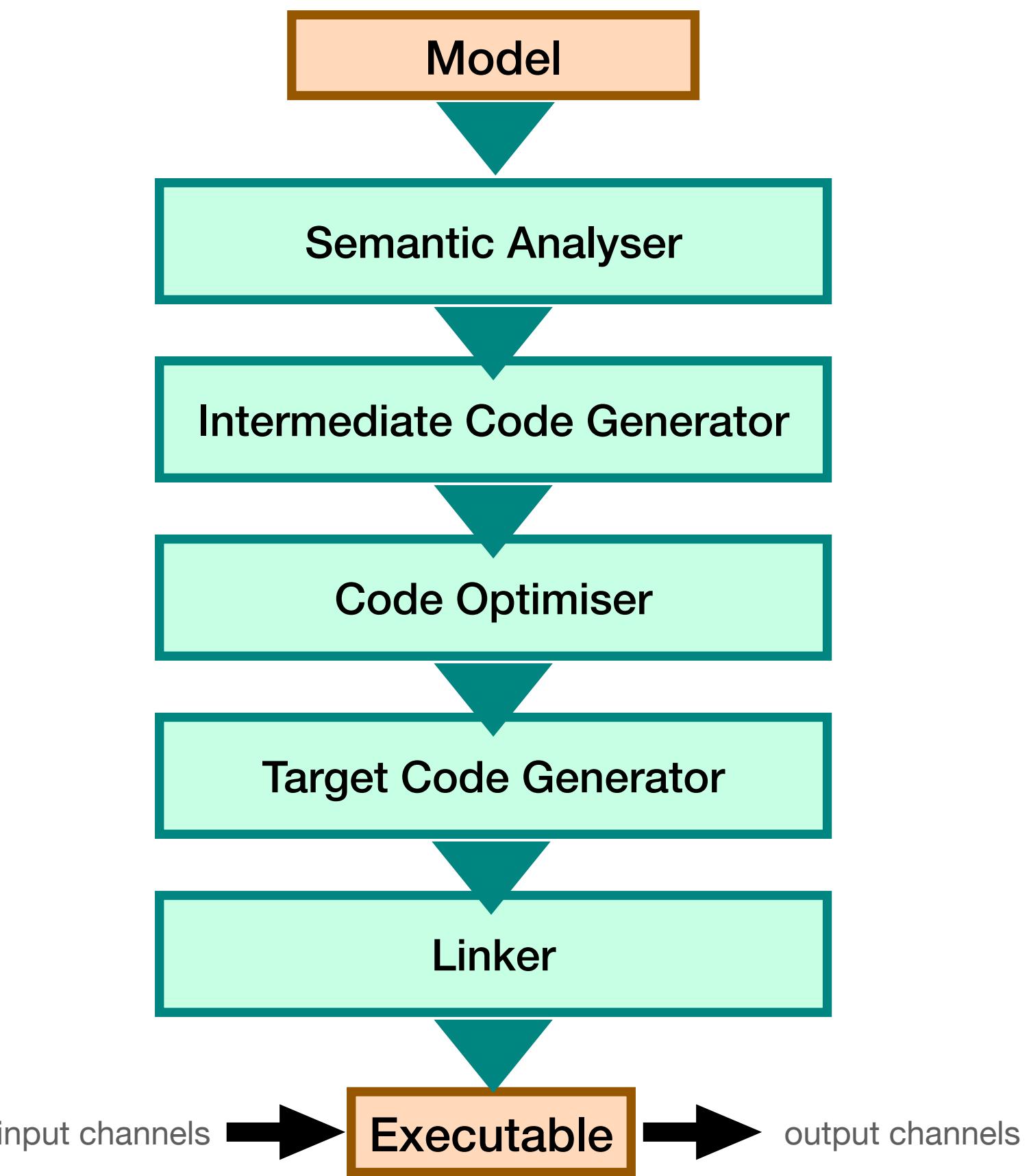
<https://www.geeksforgeeks.org/phases-of-a-compiler/>

Interpretadores com código intermédio com JIT



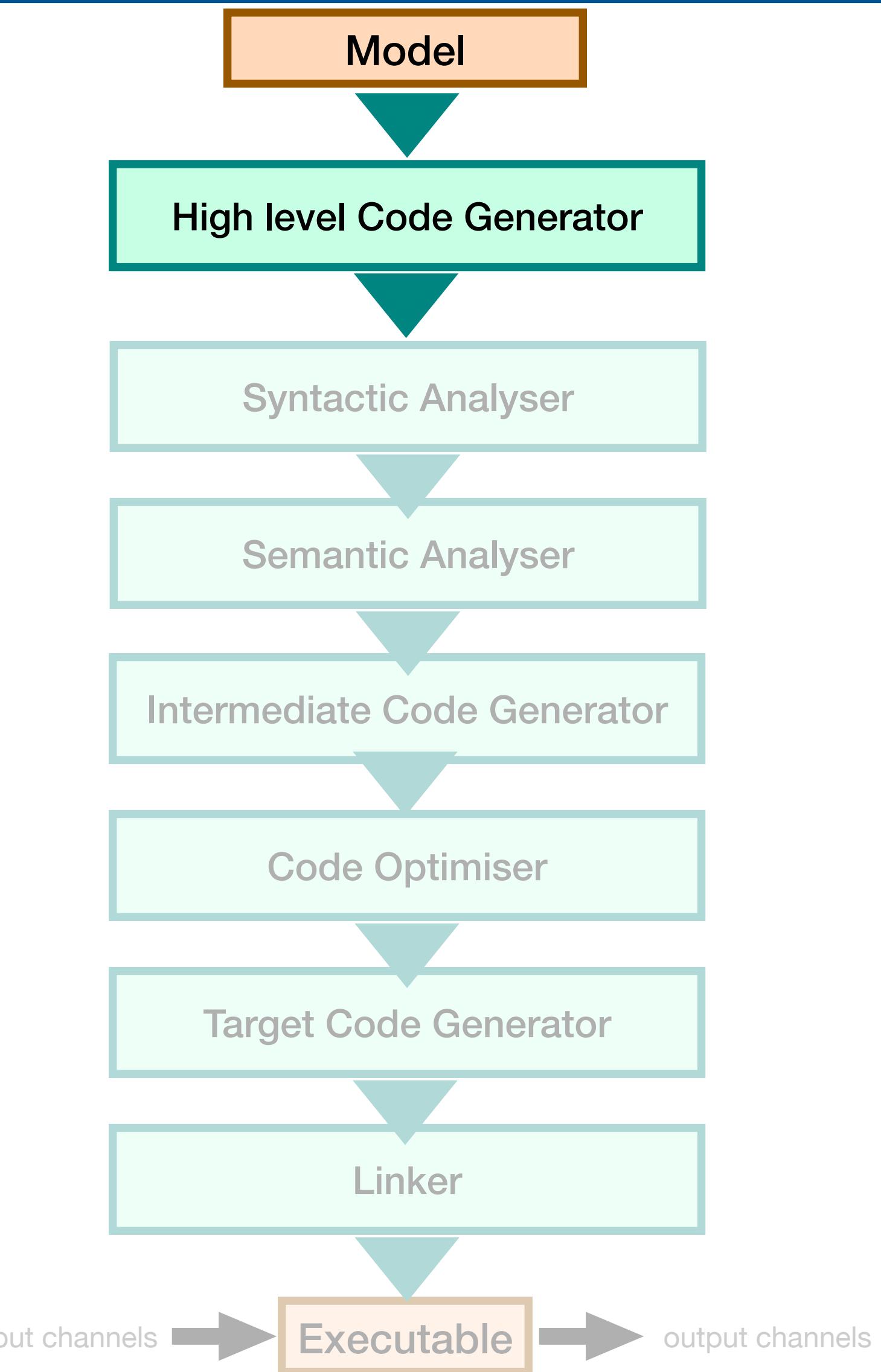
<https://www.geeksforgeeks.org/phases-of-a-compiler/>

Plataformas baseadas em modelos



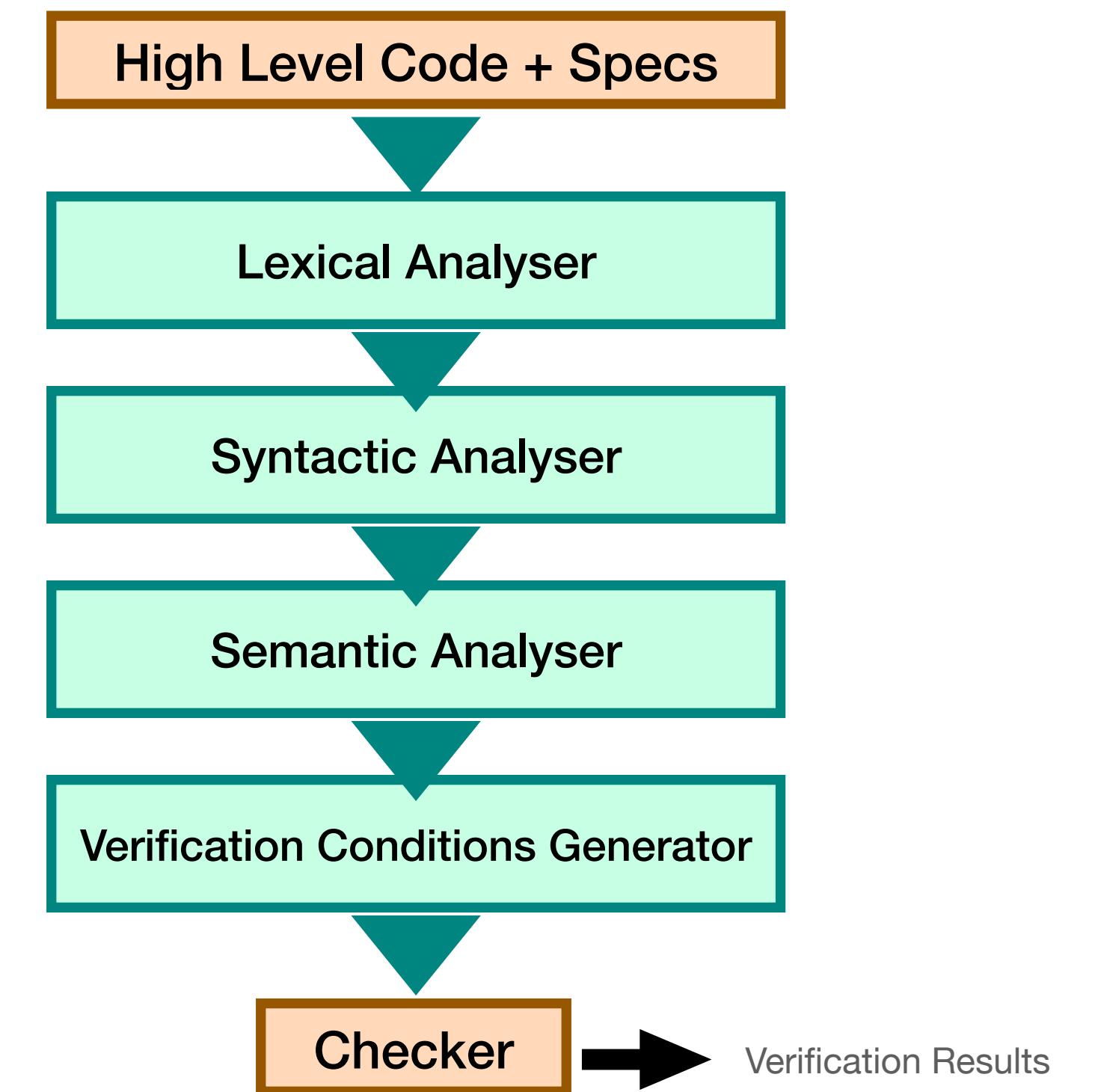
<https://www.geeksforgeeks.org/phases-of-a-compiler/>

Plataformas baseadas em modelos



<https://www.geeksforgeeks.org/phases-of-a-compiler/>

Ferramentas de verificação



<https://www.geeksforgeeks.org/phases-of-a-compiler/>

Sintaxe Concreta vs Sintaxe Abstrata vs Modelos

- A representação textual de programas que os programadores precisam de conhecer chama-se a sintaxe concreta.
 - $(1+2)^*3$
 - $(1+2)^*3 = 6 \&& 2 <= 3$
 - let $x = 1+2$ in x^*3
- A representação interna de compiladores e ferramentas de análise permite a manipulação por algoritmos de verificação e transformação.
 - `Mul(Add(Num(1), Num(2)), Num(3))`
 - `And(Equal(Mul(Add(Num(1), Num(2)), Num(3)) , Num(6)), ...)`
 - `Let("x", Add(Num(1), Num(2)), Mul(Use("x"), Num(3)))`

Tipo que representa um programa: uma calculadora simples

- As expressões são compostas por operadores binários, organizados numa árvore de elementos heterogéneos.
- Algoritmos sobre programas são agora algoritmos sobre uma árvore de elementos de várias naturezas.
- Um tipo algébrico permite representar qualquer expressão válida de uma linguagens de expressões.

```
▶ type ast =
| Num of int
| Add of ast * ast
| Sub of ast * ast
| Mul of ast * ast
| Div of ast * ast
| IfNZero of ast * ast * ast
[6] ✓ 0.0s
```

```
▶ let example_1 = IfNZero (Num 1, Num 3, Num 4)
let example_2 = Add (Num 1, Num 2)
let example_3 = Add (Num 1, IfNZero (Sub (Num 1, Num 1), Num 3, Num 4))
[7] ✓ 0.0s
```

Sintaxe Concreta vs Sintaxe Abstrata

- A representação textual de programas que os programadores conhecem chama-se a sintaxe concreta.
- $(1+2)*3$
- $(1+2)^3$
- `let x =`
- A representação abstrata é manipulada por ferramentas de transformação.
- `Mul(Add(Num(1), Num(2)), Num(3))`
- `And(Equal(Mul(Add(Num(1), Num(2)), Num(3)), Num(6)), LessThan(Num(6), Num(7)))`
- `Let("x", Add(Num(1), Num(2)), Mul(Use("x"), Num(3)))`

Modelos são representações abstratas geralmente editadas diretamente por ferramentas especializadas. Normalmente são serializados em bases de dados, JSON ou XML.

```
{  
  "type": "LogicalExpression",  
  "operator": "&&",  
  "left": {  
    "type": "BinaryExpression",  
    "operator": "=",  
    "left": {  
      "type": "BinaryExpression",  
      "operator": "*",  
      "left": {  
        "type": "BinaryExpression",  
        "operator": "+",  
        "left": {  
          "type": "Literal",  
          "value": 1  
        },  
        "right": {  
          "type": "Literal",  
          "value": 2  
        }  
      },  
      "right": {  
        "type": "Literal",  
        "value": 3  
      }  
    },  
    "right": {  
      "type": "Literal",  
      "value": 6  
    }  
  }  
}
```

Structured programming

- As linguagens que são construídas de forma compositinal, usando blocos bem definidos e funções, e sem instruções de salto indisciplinadas, permitem a definição de processos de compilação e análise de código eficientes.
- Em linguagens estruturadas podemos interpretar/compilar um programa de forma compositinal, tratando das partes de cada expressão/comando.
- A semântica de uma linguagem é uma função de um elemento sintático para um determinado resultado (valor/código/tipo).
- Os algoritmos de avaliação/compilação/tipificação são tipicamente algoritmos indutivos sobre árvores de elementos sintáticos.

Função de representa a avaliação de uma expressão

- A avaliação de uma expressão de uma calculadora é dada pela função `eval` onde `[eval e]` é o valor denotado pela expressão.

▷

```
let rec eval = function
| Num n → n
| Add (a, b) → eval a + eval b
| Sub (a, b) → eval a - eval b
| Mul (a, b) → eval a * eval b
| Div (a, b) → eval a / eval b
| IfNZero (a, b, c) → if eval a = 0 then eval c else eval b
```

[8]

✓ 0.0s

... val eval : ast → int = <fun>

```
eval (Add(Num 1, Mul (Num 2, Num 3))) =
eval (Num 1) + eval (Mul (Num 2, Num 3)) =
1 + eval (Mul (Num 2, Num 3)) =
1 + (eval (Num 2) * eval (Num 3)) =
1 + (2 * eval (Num 3)) =
1 + (2 * 3) =
1 + 6 =
7
```

Now with booleans

- Quando temos valores de tipos diferentes a AST permite a criação de expressões heterogêneas que denotam valores de diferentes naturezas.

```
type ast =
| Num of int
| True
| False
| Add of ast * ast
| Sub of ast * ast
| Mul of ast * ast
| Div of ast * ast
| And of ast * ast
| Or of ast * ast
| Not of ast
| Eq of ast * ast
| Ge of ast * ast
| Le of ast * ast
| Gt of ast * ast
| Lt of ast * ast
| If of ast * ast * ast
```

```
type result =
| ValI of int
| ValB of bool

let int_of v =
match v with
| ValI n -> n
| _ -> failwith "Expecting an Integer"

let bool_of v =
match v with
| ValB b -> b
| _ -> failwith "Expecting an Boolean"
```

```
let rec eval (e:ast) =
match e with
| Num n -> ValI n
| True -> ValB true
| False -> ValB false
| Add (e1,e2) -> ValI (int_of(eval e1) + int_of(eval e2))
| Sub (e1,e2) -> ValI (int_of(eval e1) - int_of(eval e2))
| Mul (e1,e2) -> ValI (int_of(eval e1) * int_of(eval e2))
| Div (e1,e2) -> ValI (int_of(eval e1) / int_of(eval e2))
| Eq (e1,e2) -> ValB (int_of(eval e1) = int_of(eval e2))
| Ge (e1,e2) -> ValB (int_of(eval e1) >= int_of(eval e2))
| Le (e1,e2) -> ValB (int_of(eval e1) <= int_of(eval e2))
| Gt (e1,e2) -> ValB (int_of(eval e1) > int_of(eval e2))
| Lt (e1,e2) -> ValB (int_of(eval e1) < int_of(eval e2))
| And (e1,e2) -> ValB (bool_of(eval e1) && bool_of(eval e2))
| Or (e1,e2) -> ValB (bool_of(eval e1) || bool_of(eval e2))
| Not e1 -> ValB (not (bool_of(eval e1)))
| If (c,e1,e2) -> if bool_of(eval c) then (eval e1) else (eval e2)
```

```
let e3 = If(Eq(Num(1),Num(2)),Num(0),False)
let e4 = Add(e3,Num(0))
```

Now with typing

```
type result_type = Int_ty | Bool_ty

let rec eval_type (e:ast) =
  match e with
  | Num n -> Int_ty
  | True -> Bool_ty
  | False -> Bool_ty
  | Add (e1,e2) -> if is_int e1 && is_int e2 then Int_ty else failwith("Error")
  | Sub (e1,e2) -> if is_int e1 && is_int e2 then Int_ty else failwith("Error")
  | Mul (e1,e2) -> if is_int e1 && is_int e2 then Int_ty else failwith("Error")
  | Div (e1,e2) -> if is_int e1 && is_int e2 then Int_ty else failwith("Error")
  | Eq (e1,e2) -> if eval_type e1 = eval_type e2 then Bool_ty else failwith("Error")
  | Ge (e1,e2) -> if eval_type e1 = eval_type e2 then Bool_ty else failwith("Error")
  | Le (e1,e2) -> if eval_type e1 = eval_type e2 then Bool_ty else failwith("Error")
  | Gt (e1,e2) -> if eval_type e1 = eval_type e2 then Bool_ty else failwith("Error")
  | Lt (e1,e2) -> if eval_type e1 = eval_type e2 then Bool_ty else failwith("Error")
  | And (e1,e2) -> if is_bool e1 && is_bool e2 then Bool_ty else failwith("Error")
  | Or (e1,e2) -> if is_bool e1 && is_bool e2 then Bool_ty else failwith("Error")
  | Not e1 -> if is_bool e1 then Bool_ty else failwith("Error")
  | If (c,e1,e2) -> if is_bool c then if eval_type e1 = eval_type e2 then eval_type e1 else failwith("Error") else failwith("Error")
  and
  | is_int e = eval_type e = Int_ty
  and
  | is_bool e = eval_type e = Bool_ty
```