Programming Languages and Environments (Lecture 16)

LEI - Licenciatura em Engenharia Informática

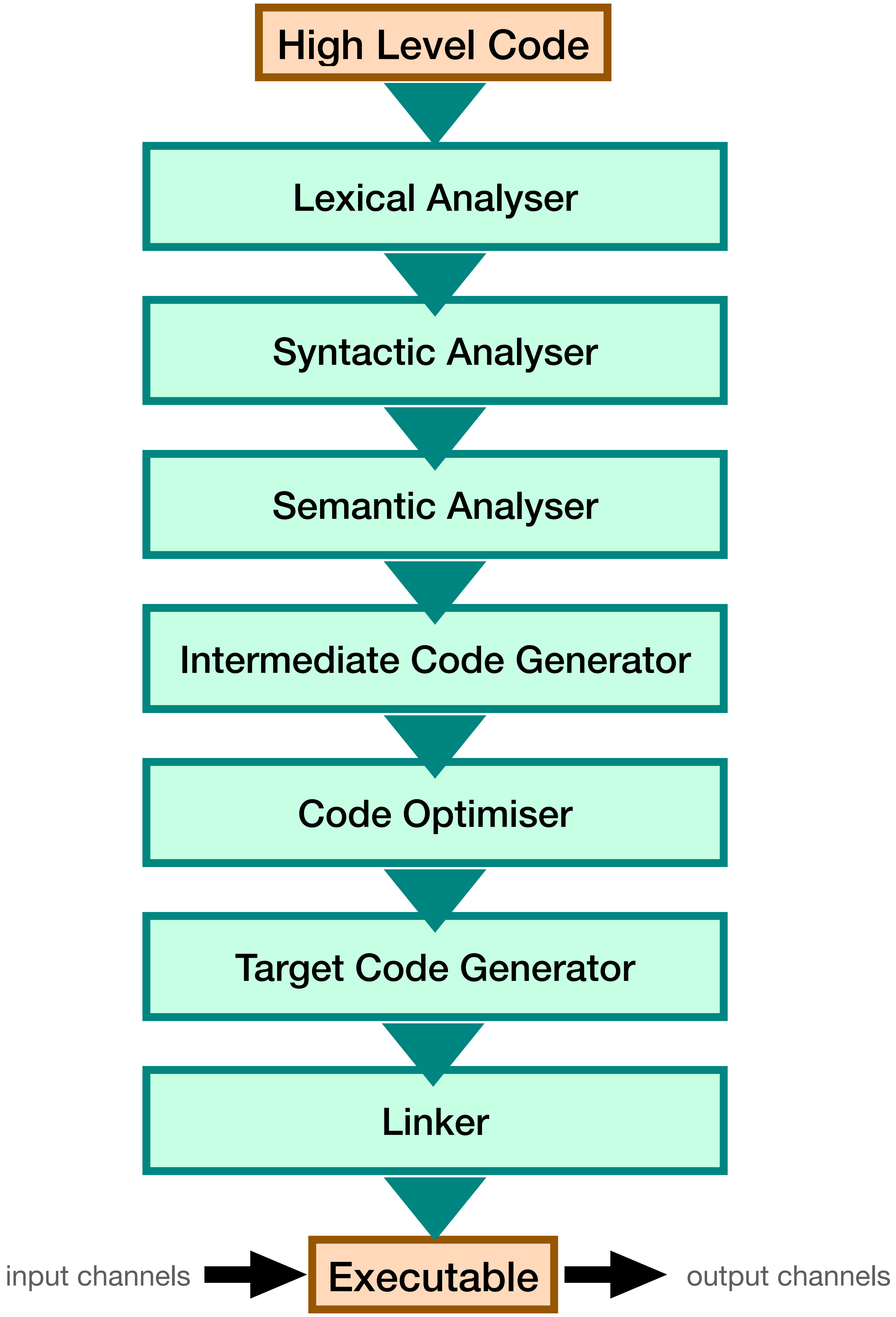
João Costa Seco (joao.seco@fct.unl.pt)

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

Code as data (simplified)

- Compilers (from source code to machine code, executable)
- Interpreters (execution of source code)
- Code generators (from specifications to source code)
- Model-driven platforms (from models to source code or execution)
- Static code analyzers (from source code and specifications to property verification)
- Correctness, security, performance, etc.

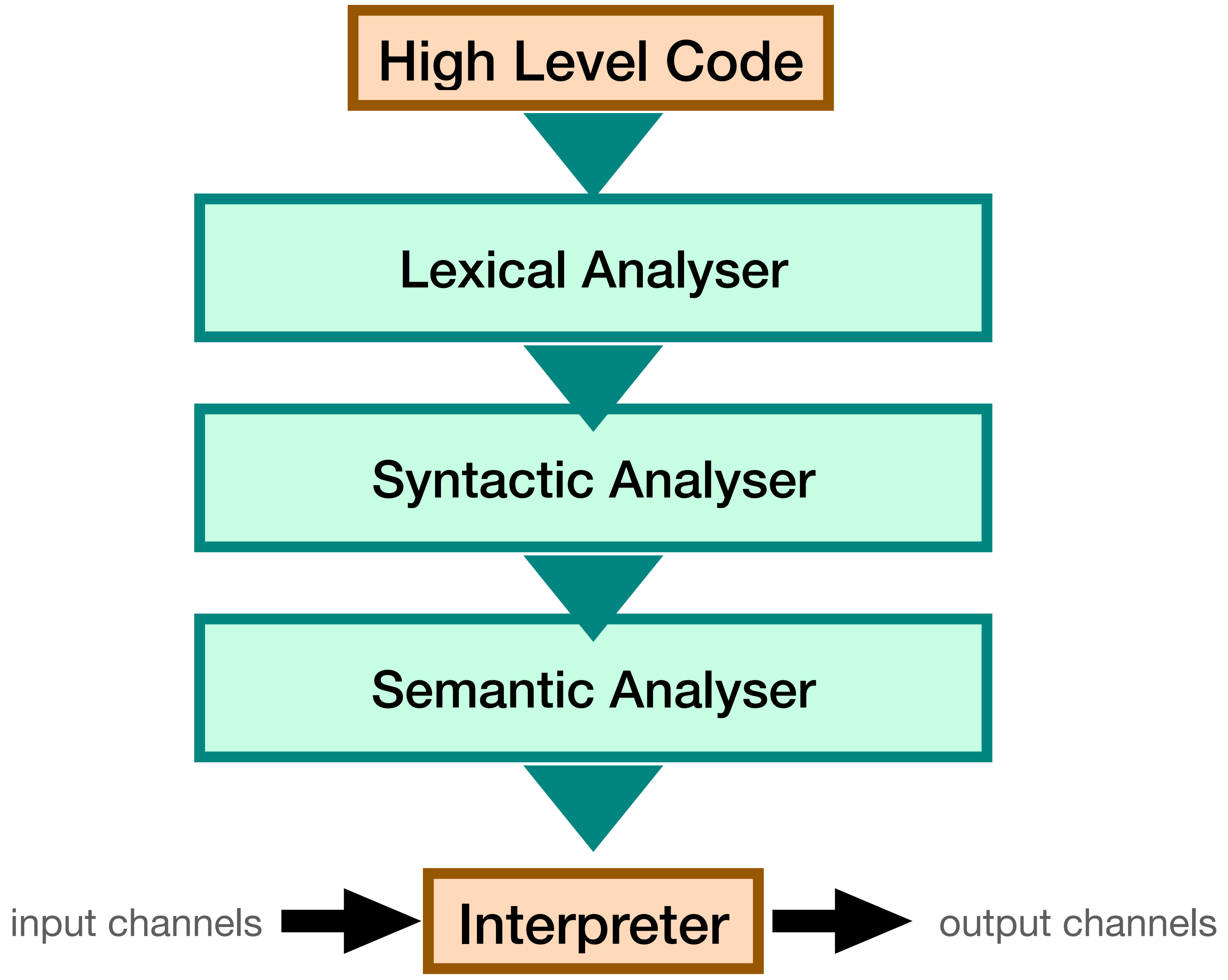
Compilers



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

https://www.cs.cmu.edu/~fp/courses/15411-f14/lectures/01-overview.pdf

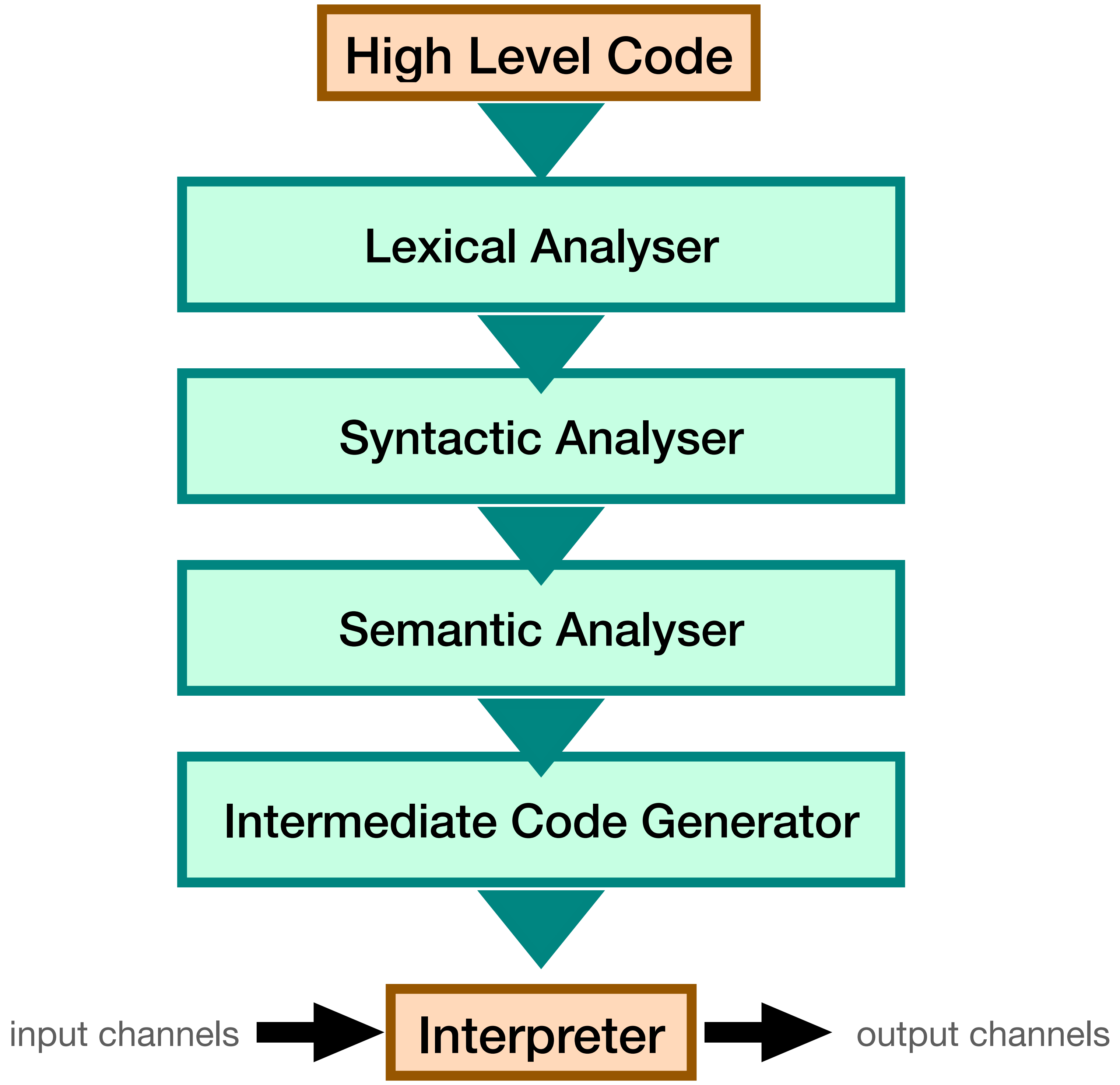
Interpreters



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

https://www.cs.cmu.edu/~fp/courses/15411-f14/lectures/01-overview.pdf

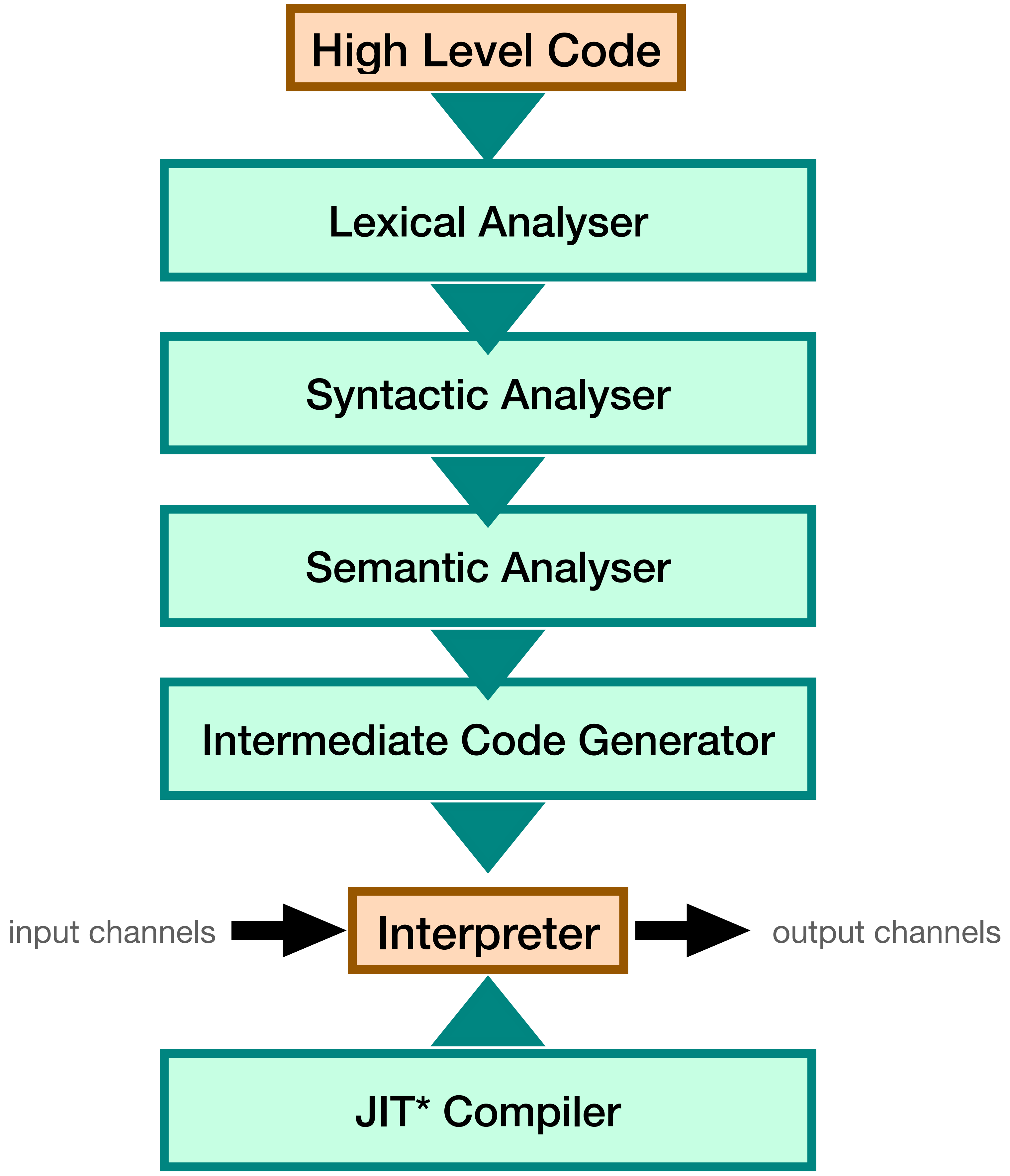
Interpreters with intermediate code



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

https://www.cs.cmu.edu/~fp/courses/15411-f14/lectures/01-overview.pdf

Interpreters with intermediate code and JIT

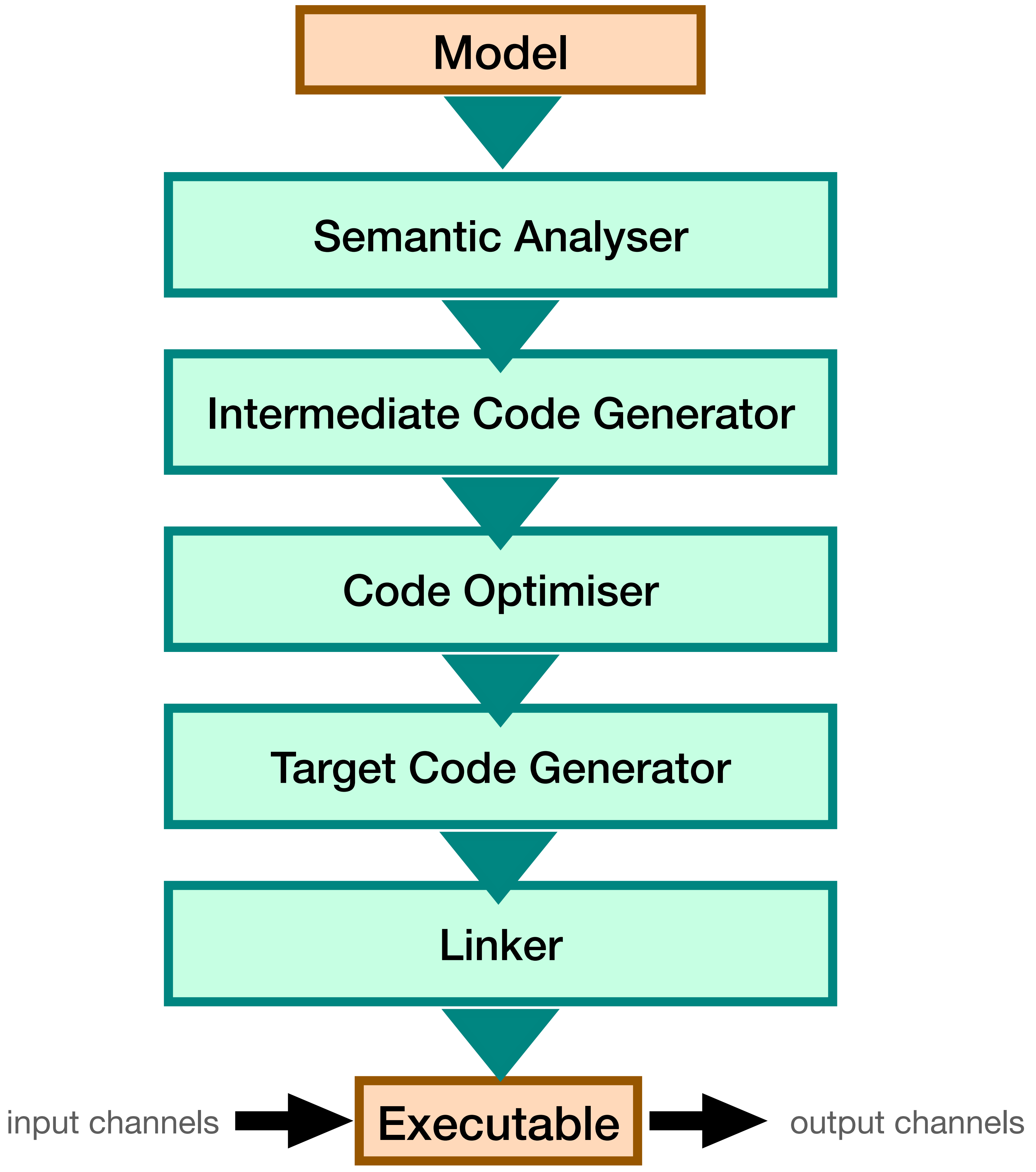


https://www.geeksforgeeks.org/phases-of-a-compiler/

https://www.cs.cmu.edu/~fp/courses/15411-f14/lectures/01-overview.pdf

*Just in time

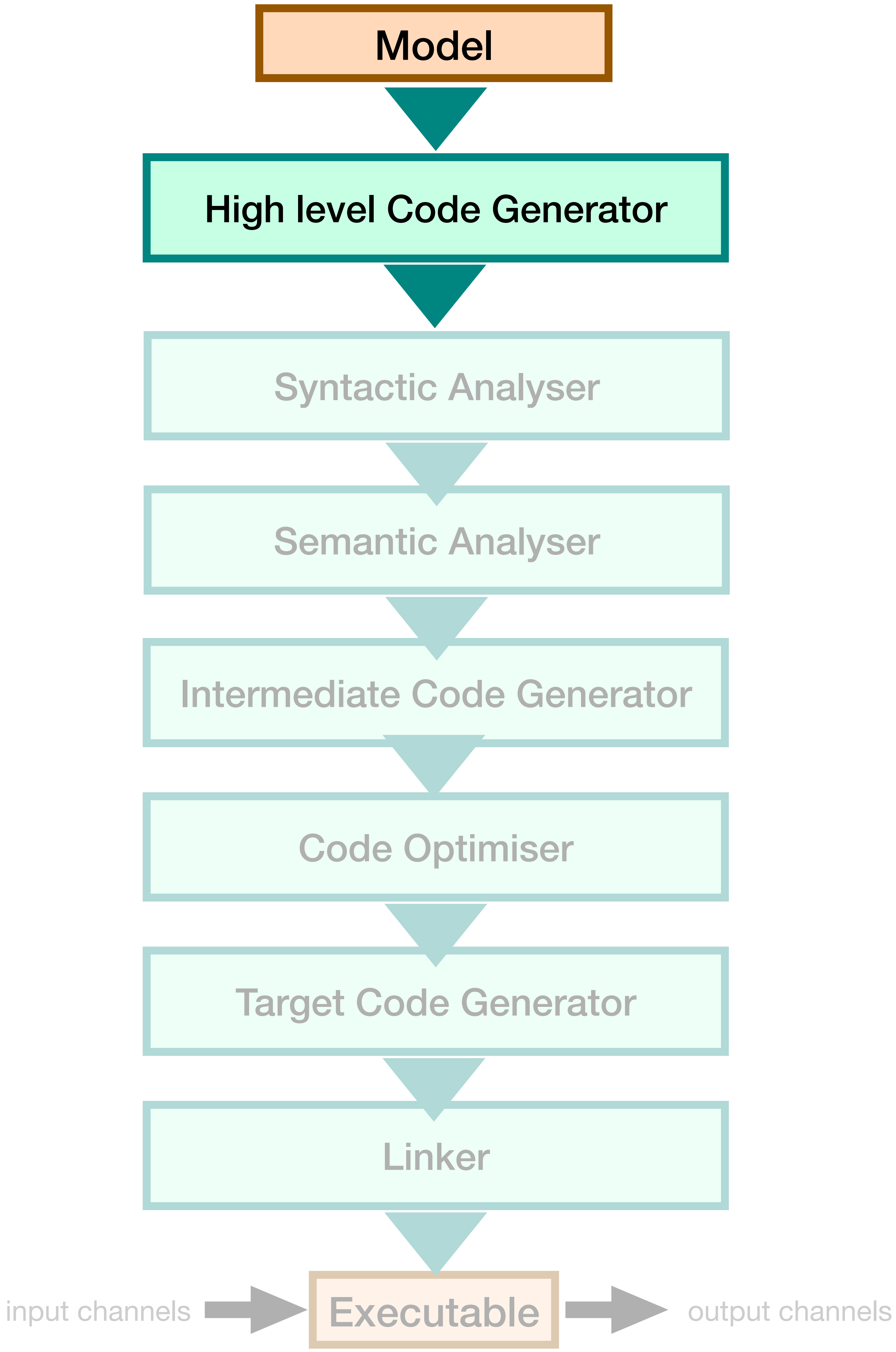
Platforms based on models



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

https://www.cs.cmu.edu/~fp/courses/15411-f14/lectures/01-overview.pdf

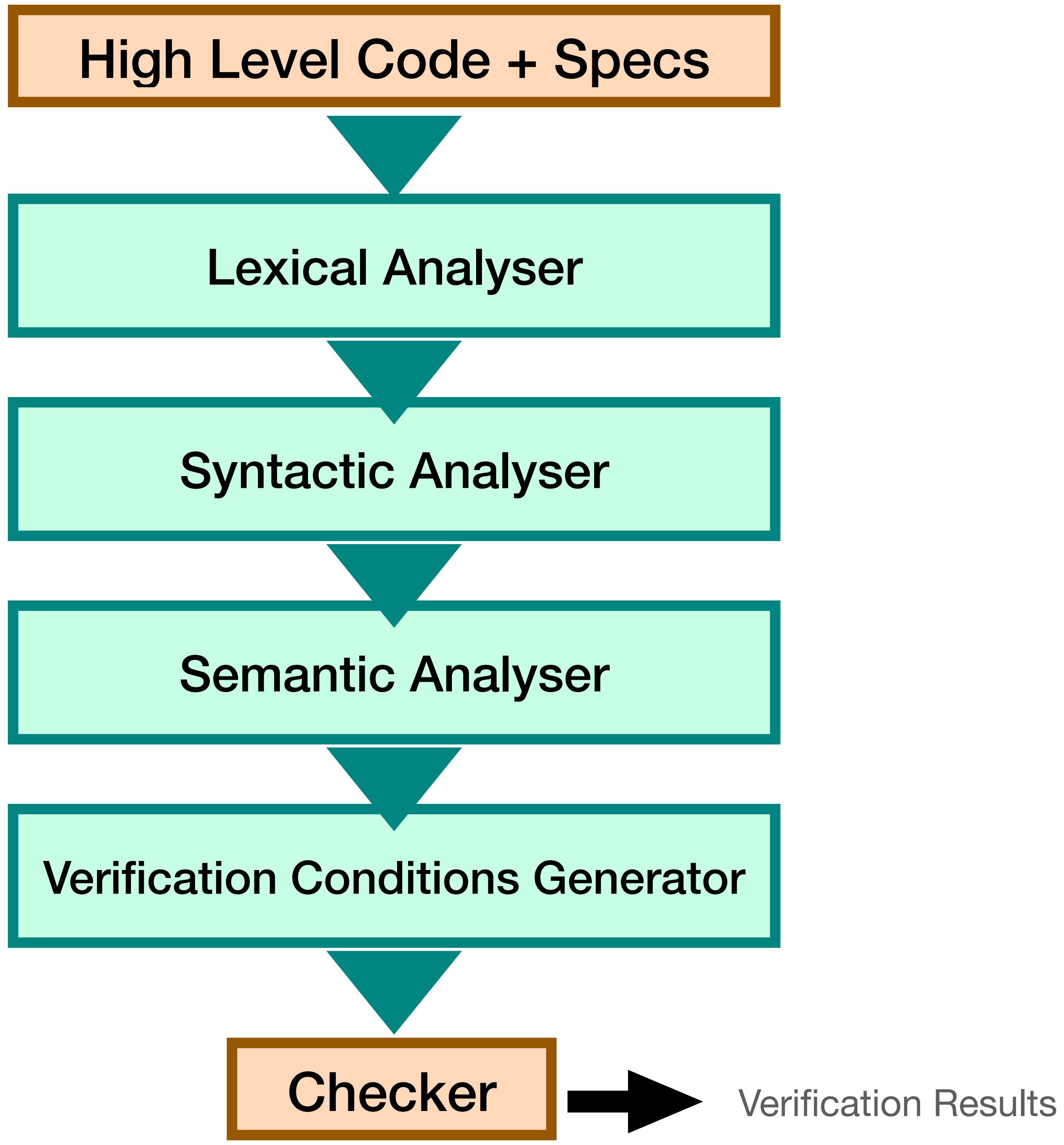
Platforms based on models



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

https://www.cs.cmu.edu/~fp/courses/15411-f14/lectures/01-overview.pdf

Verification tools



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

https://www.cs.cmu.edu/~fp/courses/15411-f14/lectures/01-overview.pdf

Concrete syntax vs Abstract syntax vs Models

- The textual representation of programs that programmers need to understand is called the concrete syntax.
  - (1+2)*3
  - (1+2)*3 = 6 && 2 <= 3
  - let x = 1+2 in x*3
- The internal representation used by compilers and analysis tools enables manipulation by verification and transformation algorithms.
  - 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)))

A simple calculator

- Expressions are composed of binary operators, organized into a tree of heterogeneous elements.
- Algorithms over programs are now algorithms over a tree of elements of various kinds.
- An algebraic data type allows representing any valid expression in an expression language.

```
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

Concrete syntax vs Abstract syntax vs

- The textual representation of programs that progra
called the concrete syntax.
  - (1+2)*3
  - (1+2)
  - let x =
- The inte s and
manipu on alg
  - Mul(Add(Num(1), Num(2)), Num(3))
  - And(Equal(Mul(Add(Num(1), Num(2)), Num(3)
  - Let(“x”, Add(Num(1), Num(2)), Mul(Use(“x”), Nu

Models are abstract representations typically edited directly using specialized tools. They are usually serialized in databases, JSON, or XML.


```json
{
  "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

- Languages that are built compositionally, using well-defined blocks and functions, and without unstructured jump instructions, allow for the definition of efficient compilation and code analysis processes.
- In structured languages, we can interpret and compile a program compositionally, handling each part of an expression or command individually.
- The semantics of a language is a function from a syntactic element to a specific result (value/code/type).
- Evaluation, compilation, and type-checking algorithms are typically inductive algorithms over trees of syntactic elements.

Evaluation of an expression

- The evaluation of an expression is our calculator is given by the `eval` function, where `[eval e]` is the value denoted by the expression.

```
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
```

```
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>
```

Now with booleans

- Quando temos valores de tipos diferentes
  a AST permite a criação de expressões heterógeneas
  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
```

Stack machine code

```
type code =
  | Push of int
  | Add
  | Sub
  | Mul
  | Div
  | Ge
  | JmpNZ of string
```

```
module CodeMap = Map.Make(String)

let c0 = CodeMap.empty
      |> CodeMap.add "main" [Push 99; Push 0; Ge; JmpNZ "label"; Push 0; Push 99; Sub]
      |> CodeMap.add "label" [Push 99; Push 1; Add]

let _ = loop (CodeMap.find "main" c0) [] c0
```

```
let rec loop insts stack code =
  match insts, stack with
  | [], _ -> stack
  | Push n :: next, stack' -> loop next (n::stack') code
  | Add :: next, x::y::stack' -> loop next ((x+y)::stack') code
  | Sub :: next, x::y::stack' -> loop next ((y-x)::stack') code
  | Mul :: next, x::y::stack' -> loop next ((x*y)::stack') code
  | Div :: next, x::y::stack' -> loop next ((y/x)::stack') code
  | Ge :: next, x::y::stack' -> loop next ((if y >= x then 1 else 0)::stack') code
  | JmpNZ label :: next, x::stack' -> if x <> 0 then loop (CodeMap.find label code) stack' code
                                      else loop next stack' code
  | _ -> failwith "Bad Program!!!"
```

Function that represents the translation to stack machine code

- Each expression has an invariant condition: it always leaves the value it denotes on top of the stack. This condition serves as the induction hypothesis for the composition of multiple sub-expressions.

```
let rec compile e =
  match e with
  | ENum n -> [Push n]
  | EAdd (e1,e2) -> (compile e1)@(compile e2)@[Add]
  | EMul (e1,e2) -> (compile e1)@(compile e2)@[Mul]

let e5 = EMul(EAdd(ENum(1),ENum(1)),ENum(3))

let _ = assert ([Push 1; Push 1; Add; Push 3; Mul] = compile e5)
```