Linguagens e Ambientes de Programação (Aula Teórica 16)

LEI - Licenciatura em Engenharia Informática

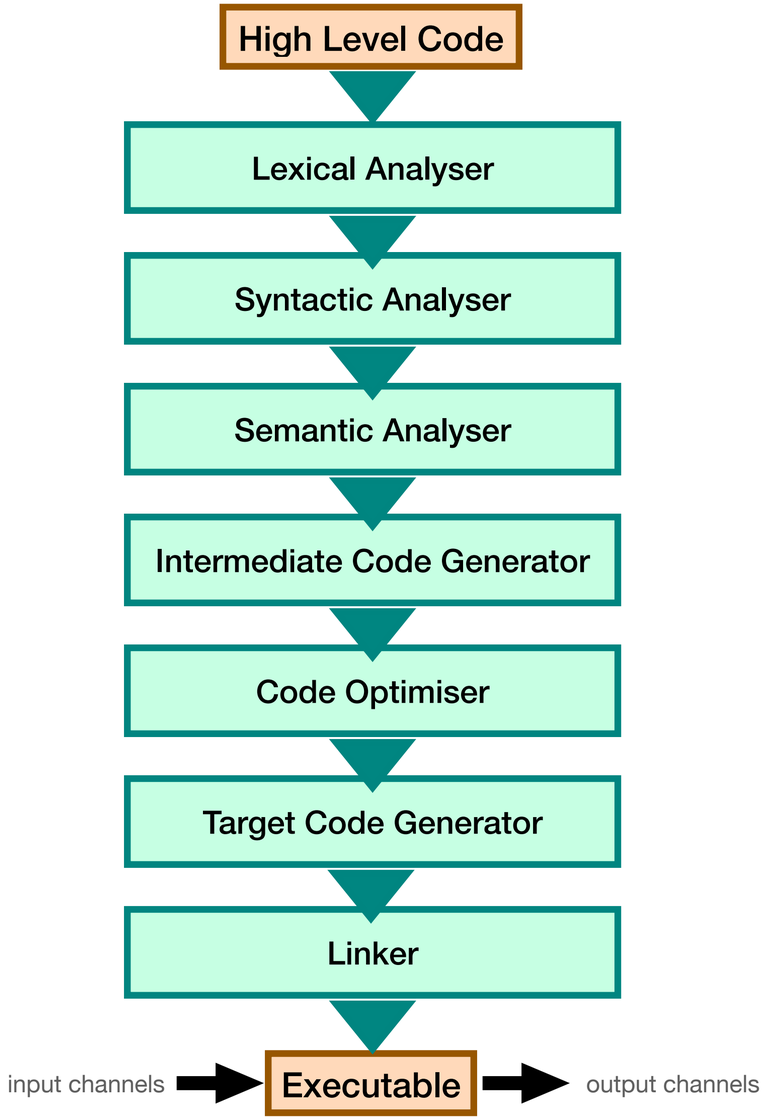
João Costa Seco (joao.seco@fct.unl.pt)

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

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.

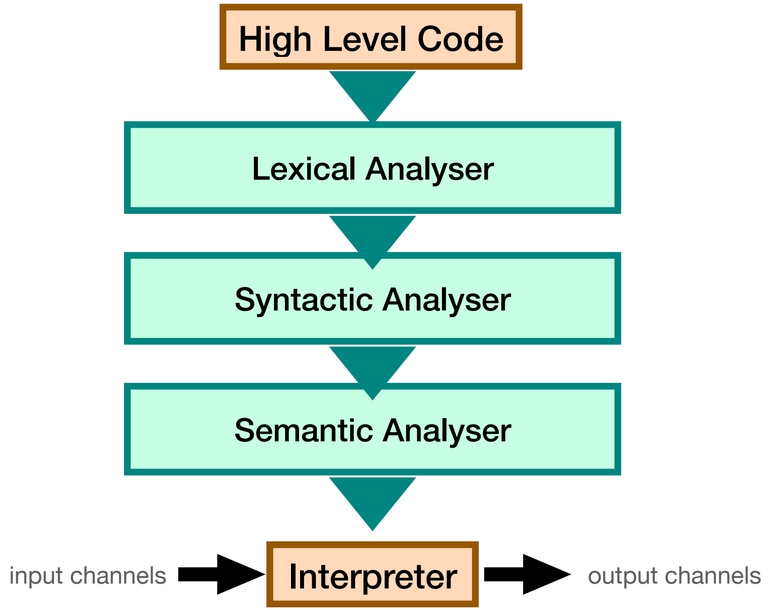
Compiladores



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

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

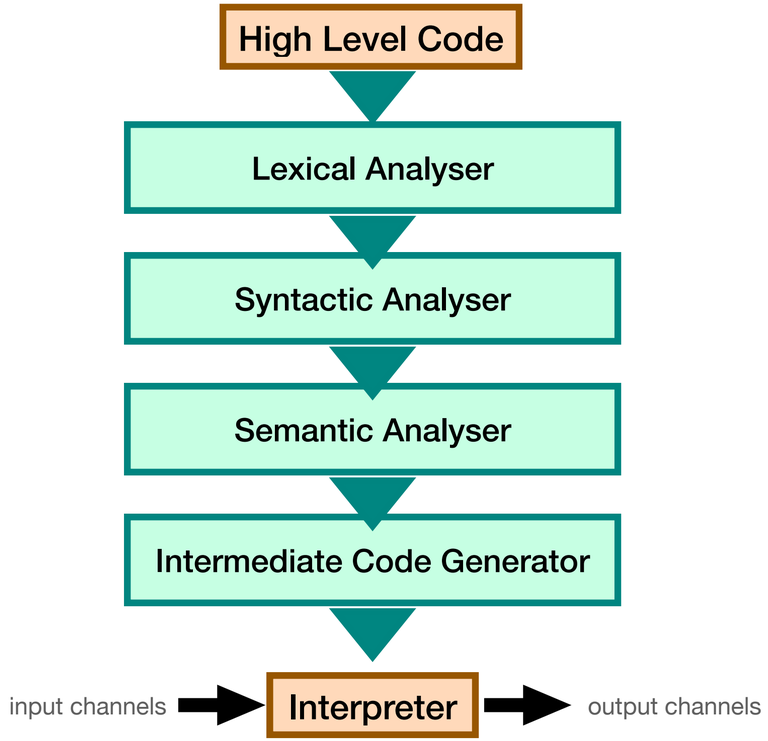
Interpretadores



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

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

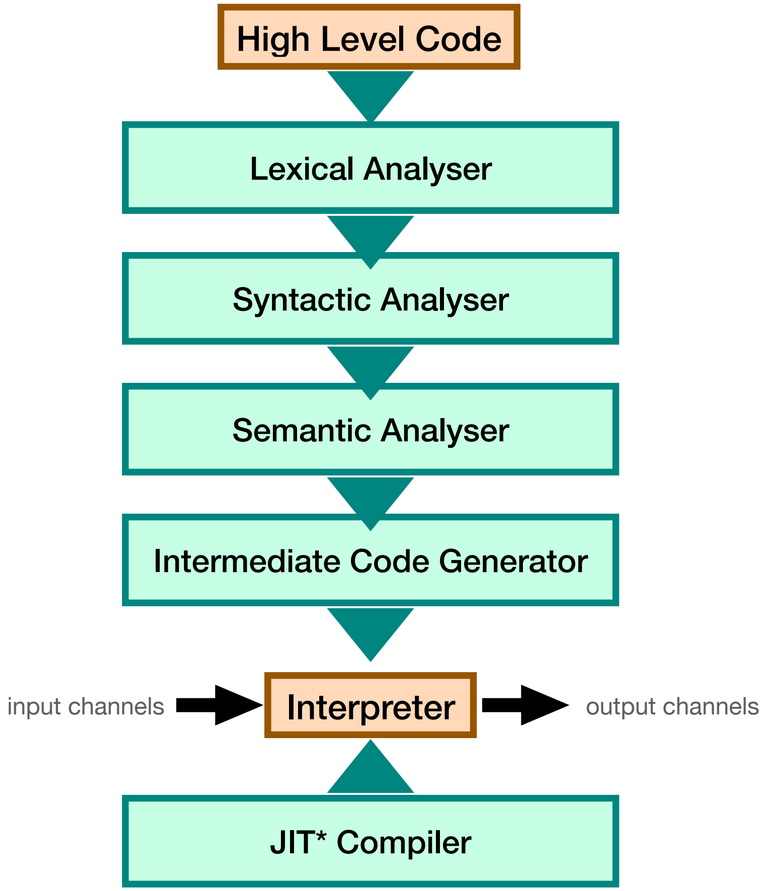
Interpretadores com código intermédio



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

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

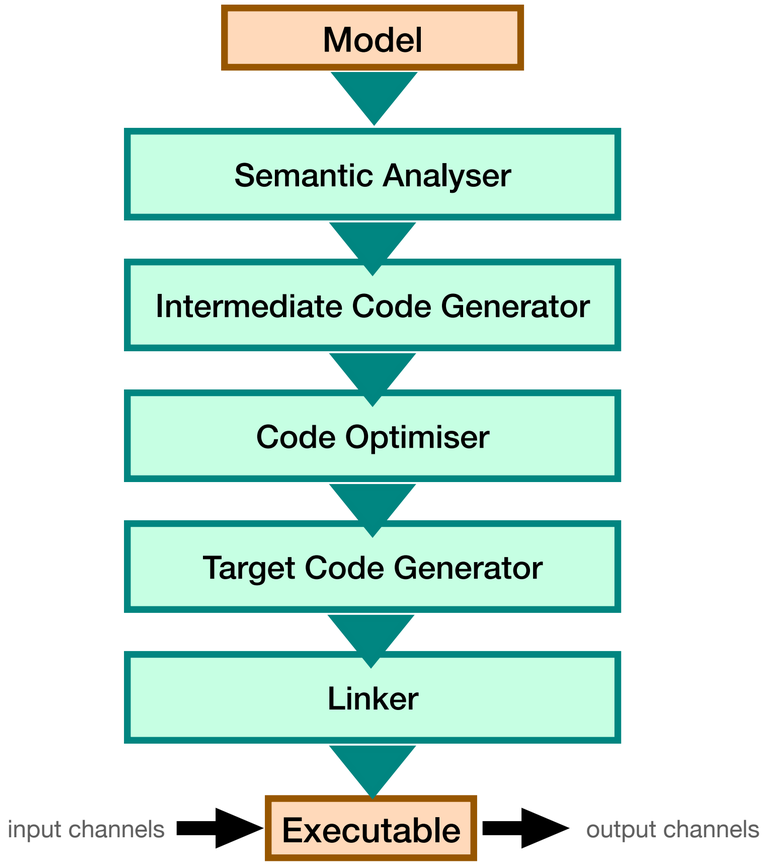
Interpretadores com código intermédio com JIT



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

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

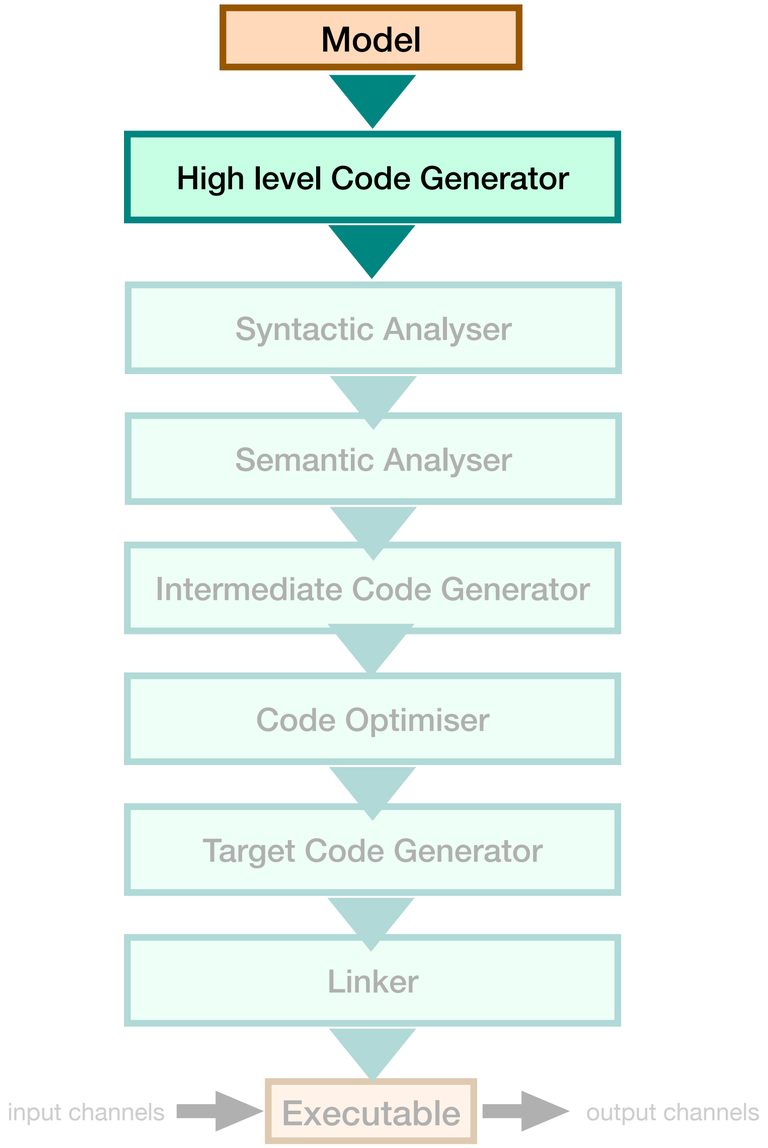
Plataformas baseadas em modelos



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

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

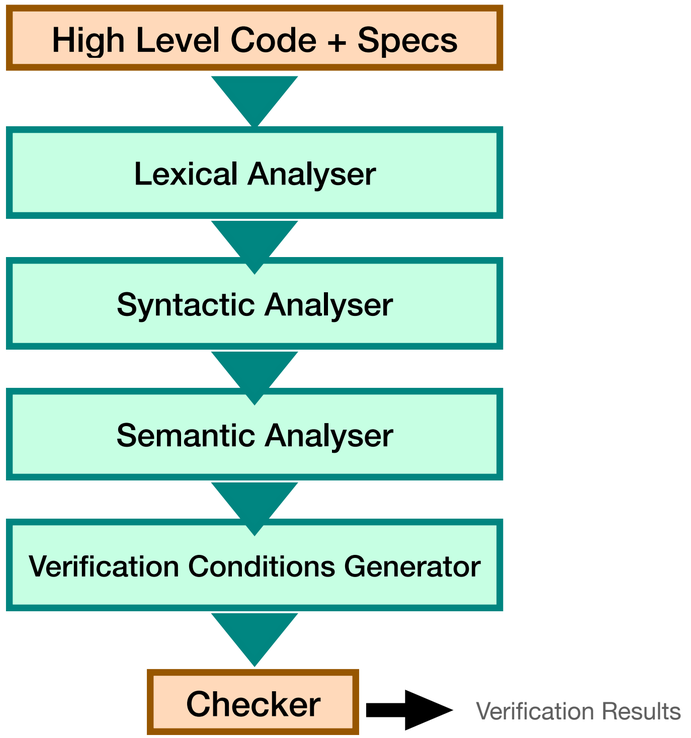
Plataformas baseadas em modelos



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

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

Ferramentas de verificação



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

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

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.

```
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

- Um tipo algébrico permite representar qualquer expressão válida de uma linguagens de expressões.

```
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 pro
conhecer chama-se a sintaxe concreta.
  - (1+2)*3
  - (1+2)*
  - let x =
- A repre
manipu
  - 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

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 composicional, 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 composicional, 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.

```
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
```