Linguagens e Ambientes de Programação (Aula Teórica 18)

LEI - Licenciatura em Engenharia Informática

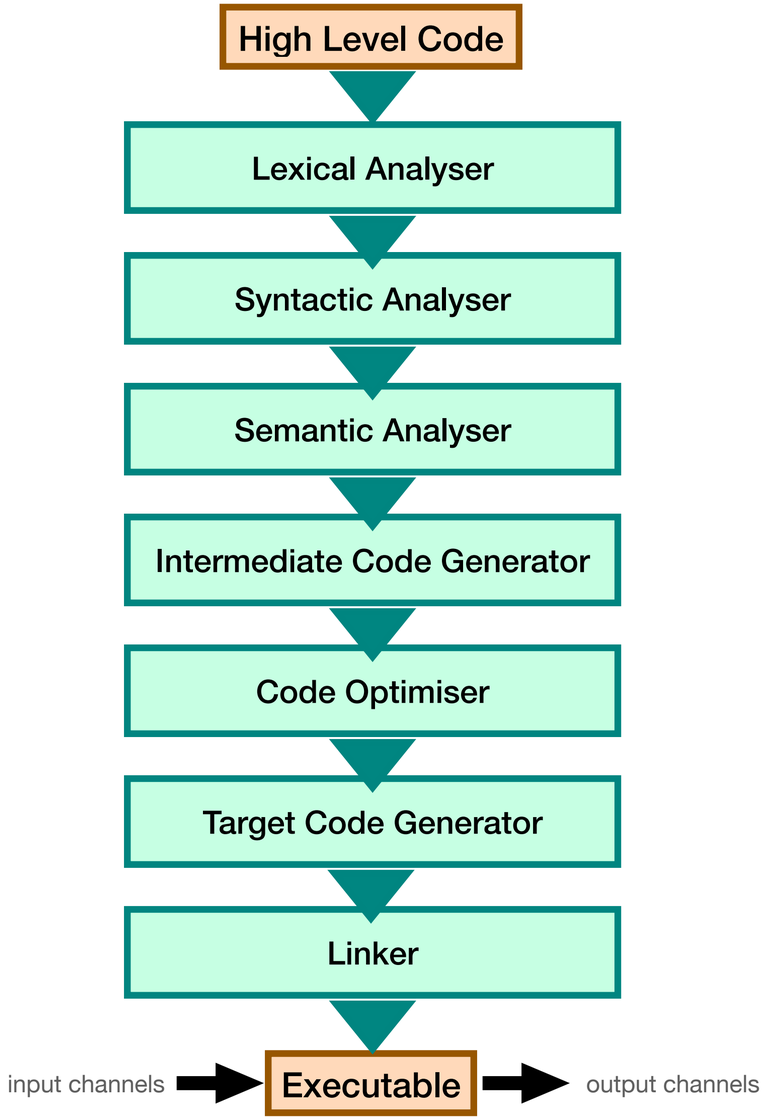
João Costa Seco (joao.seco@fct.unl.pt)

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.

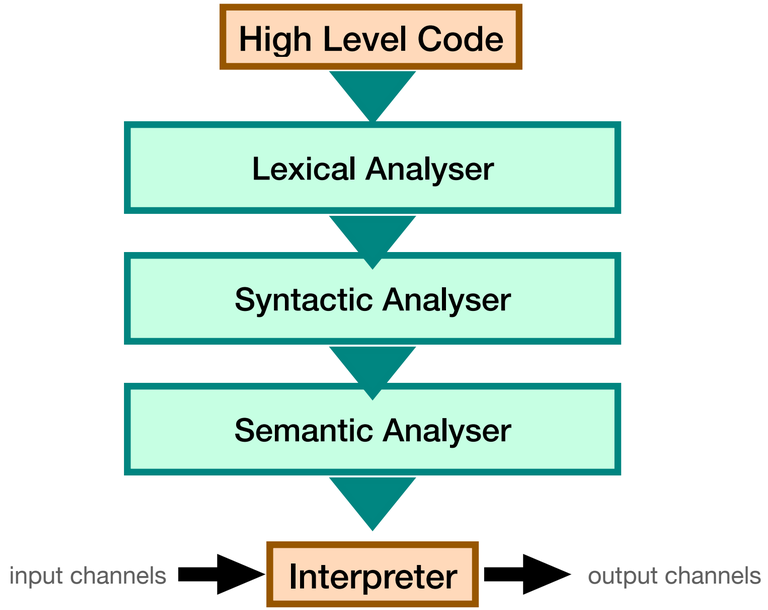
Compiladores



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

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

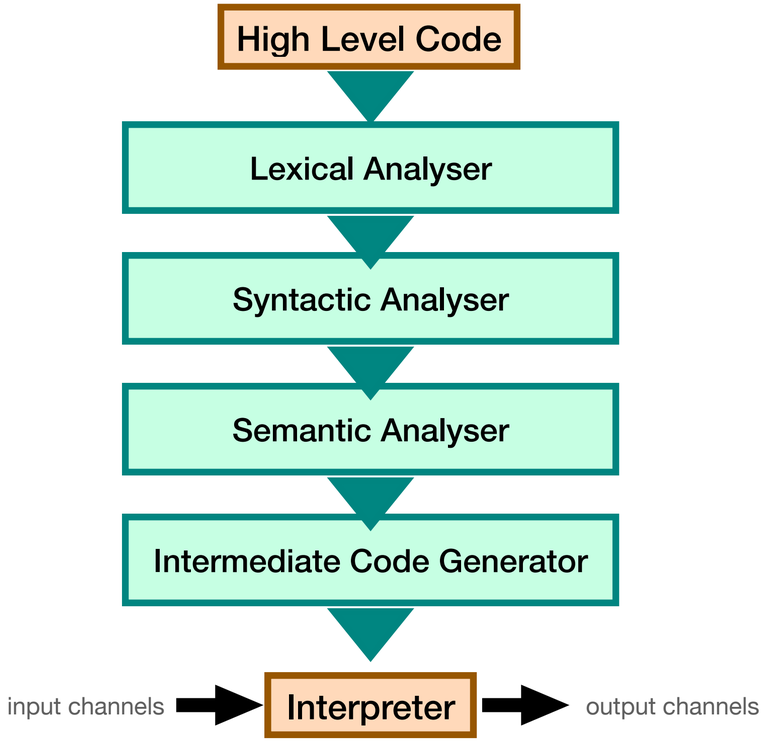
Interpretadores



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

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

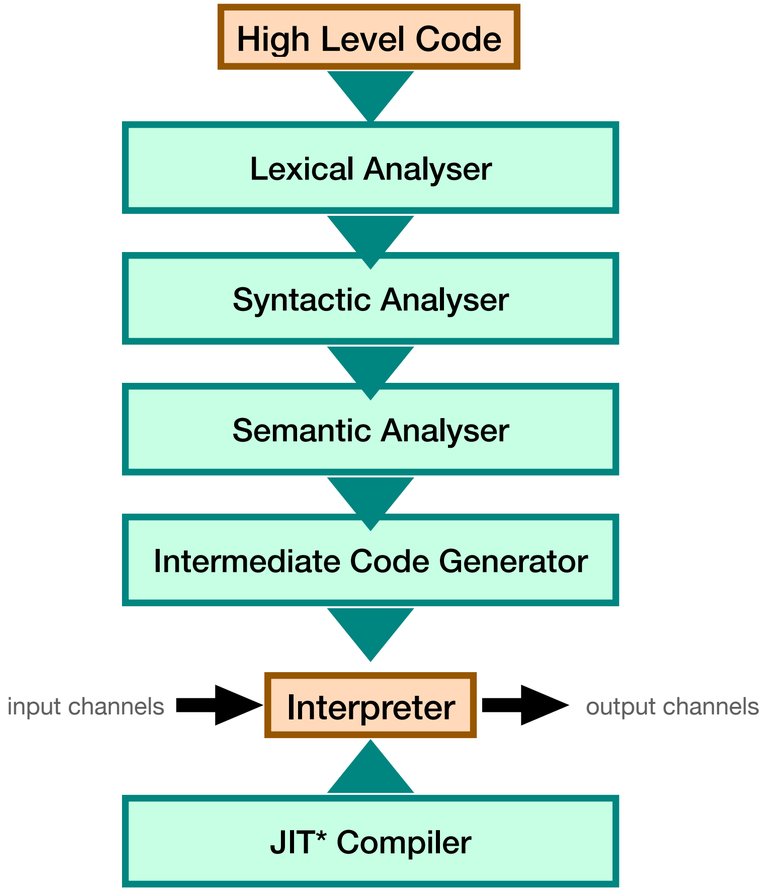
Interpretadores com código intermédio



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

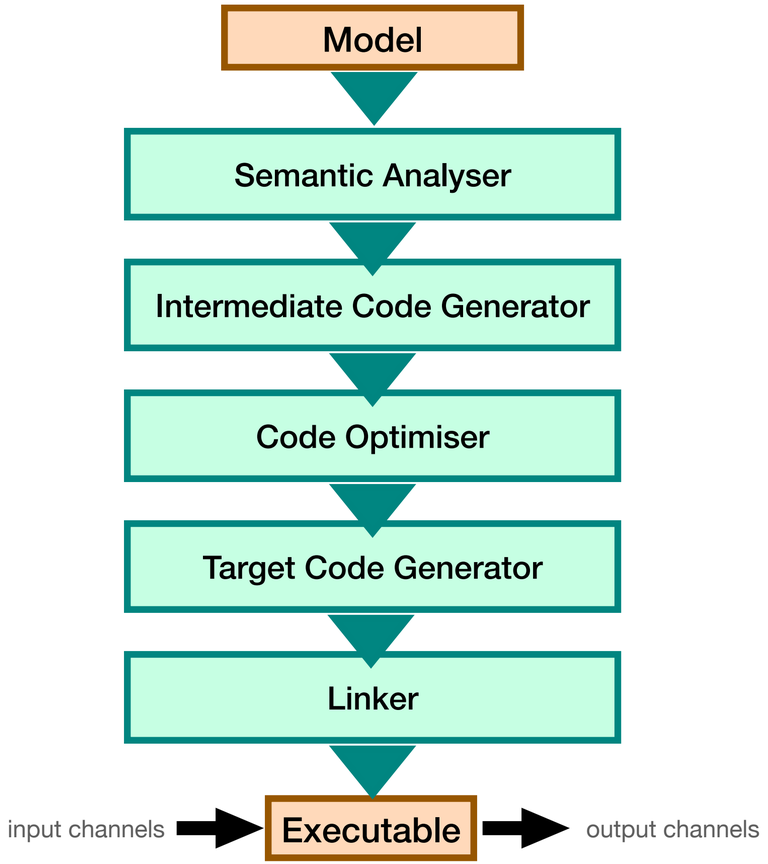


https://www.geeksforgeeks.org/phases-of-a-compiler/

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

*Just in time

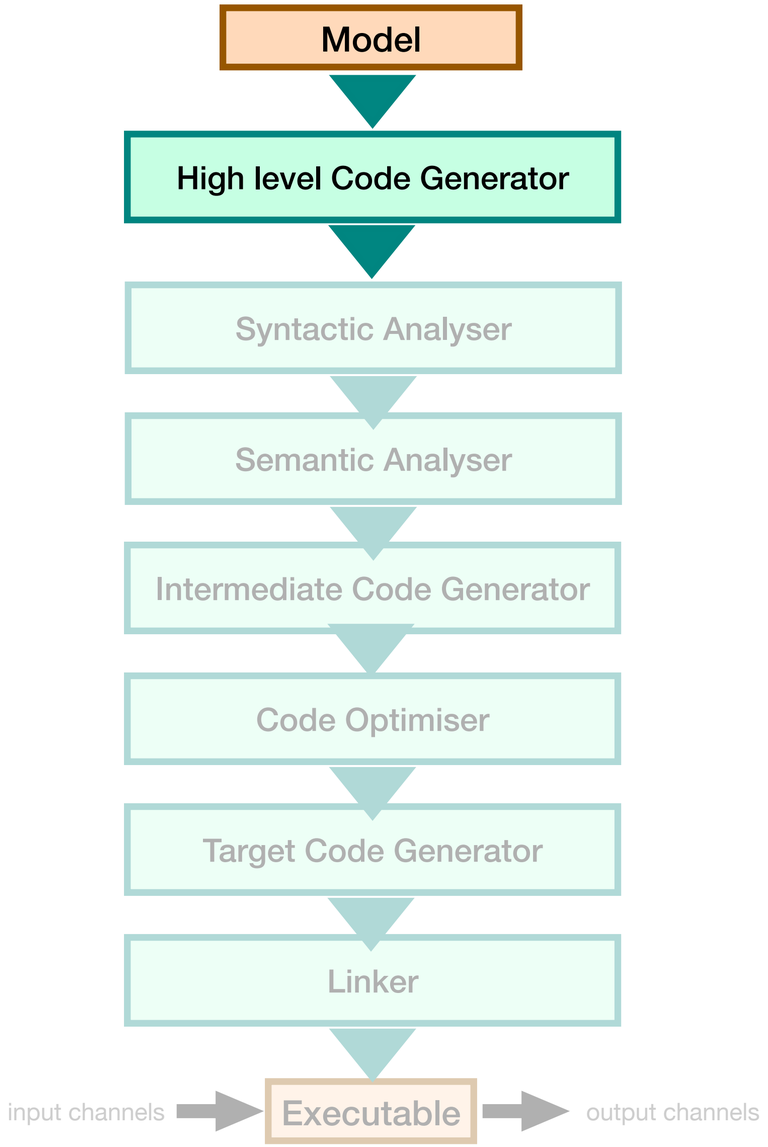
Plataformas baseadas em modelos



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

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

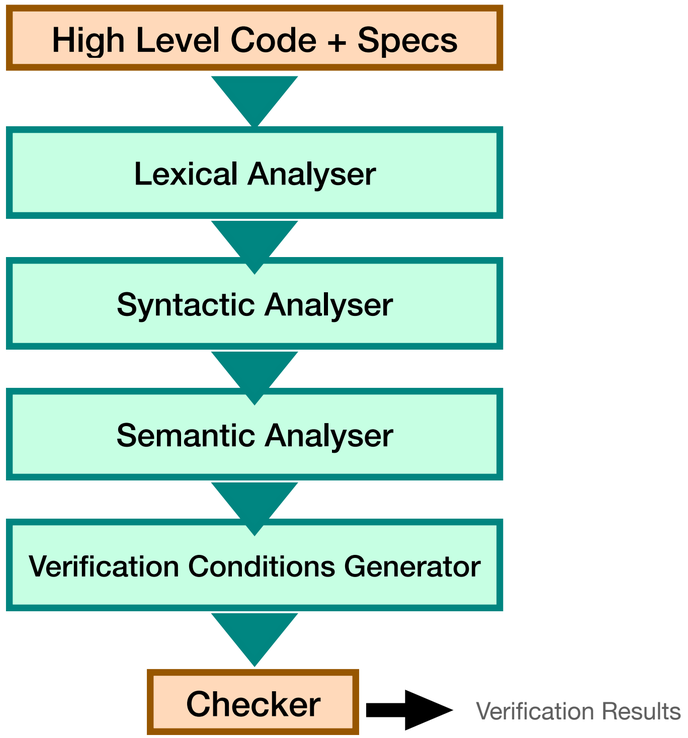
Plataformas baseadas em modelos



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 program
chama-se a sintaxe concreta.
  - `(1+2)*3`
  - `(1+2)`
  - `let x`
- A repres amenta ão
por algo
  - `Mul( Add( Num(1), Num(2) ), Num(3))`
  - `And( Equal( Mul( Add( Num(1), Num(2) )`
  - `Let("x", Add( Num(1), Num(2)), Mul( Us`

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

- 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>
```

Máquina de Pilha - ver segundo trabalho

- Relembre a estrutura das instruções do segundo trabalho.

```
type instruction =
  | SAdd
  | SSub
  | SMul
  | SDiv
  | SPush of int
  | SPop
  | SDup
  | SOver
  | SJmp of string
  | SJz of string
  | SJnz of string
  | SCmp
  | SSwp
  | SReturn
  | SLabel of string
```

```
let unparse_s = function
  | SAdd → "add"
  | SSub → "sub"
  | SMul → "mul"
  | SDiv → "div"
  | SPush n → "push " ^ string_of_int n
  | SPop → "pop"
  | SDup → "dup"
  | SOver → "over"
  | SJmp l → "jmp " ^ l
  | SJz l → "jz " ^ l
  | SJnz l → "jnz " ^ l
  | SCmp → "cmp"
  | SSwp → "swp"
  | SReturn → "return"
  | SLabel l → l ^ ":"

let unparse_l l = List.map unparse_s l
```

Função de representa a tradução para máquina de pilha

- Cada expressão tem uma condição invariante, deixa sempre o valor que denota no topo da pilha. Essa condição é a hipótese de indução para a composição de várias sub expressões.

```
let rec compile = function
  | Num n → [SPush n]
  | Add (e1, e2) → compile e1 @ compile e2 @ [SAdd]
  | Sub (e1, e2) → compile e1 @ compile e2 @ [SSub]
  | Mul (e1, e2) → compile e1 @ compile e2 @ [SMul]
  | Div (e1, e2) → compile e1 @ compile e2 @ [SDiv]
  | IfNZero (b, e1, e2) →
      let l = new_label () in
      compile b @ [SJz (l ^ "else")] @
      compile e1 @ [SJmp (l ^ "end"); SLabel (l ^ "else")] @ compile e2 @ [SLabel (l ^ "end")]
```

[15] ✓ 0.0s

```
val compile : ast → instruction list = <fun>
```

```
compile (Add (Num 1, Mul (Num 2, Num 3))) = [SPush 1; SPush 2; SPush 3; SMul; SAdd]
```

Função de representa a tradução para máquina de pilha

- Cada expressão tem uma condição in
denota no topo da pilha. Essa condiçã
composição de várias sub expressões

```
let rec compile = function
  | Num n → [SPush n]
  | Add (e1, e2) → compile e1 @ compile e2 @ [SA
  | Sub (e1, e2) → compile e1 @ compile e2 @ [SS
  | Mul (e1, e2) → compile e1 @ compile e2 @ [SM
  | Div (e1, e2) → compile e1 @ compile e2 @ [SD
  | IfNZero (b, e1, e2) →
      let l = new_label () in
      compile b @ [SJz (l ^ "else")] @
      compile e1 @ [SJmp (l ^ "end"); SLabel (l ^ "else")] @ compile e2 @ [SLabel (l ^ "end")]
```

✓ 0.0s

```
val compile : ast → instruction list = <fun>
```

```
compile (Add (Num 1, Mul (Num 2, Num 3))) = [SPush 1; SPush 2; SPush 3; SMul; SAdd]
```

```
compile (IfNZero (Num 1, Num 3, Num 4)) =
[
 SPush 1;
 SJz "L0_else";
 SPush 3;
 SJmp "L0_end";
 SLabel "L0_else";
 SPush 4;
 SLabel “L0_end”
 (* Return *)
]
```

Alonzo Church (1903 - 1995)

- A linguagem de programação fundamental
- O sistema formal cálculo Lambda

$$E ::= \; x \mid \; \lambda x.E \; \mid \; E \; E$$

$$(\lambda x.E) \; E' \longrightarrow E\{E'/x\} \qquad \frac{E \longrightarrow E''}{E \; E' \longrightarrow E'' \; E'}$$

Recordar da Aula Teórica #1

Lambda Calculus - Call by name

```
type lambda =
  | Num of int
  | Add of lambda * lambda

  | Var of string
  | App of lambda * lambda
  | Fun of string * lambda

let f = Fun ("x", Add (Var "x", Num 1))
let e = App (f, Num 2)
```

```
(fun x → x + 1) 2
```

```
let rec eval_lambda = function
  | Num n → IntValue n
  | Add (e1, e2) →
    (match eval_lambda e1, eval_lambda e2 with
      | IntValue n1, IntValue n2 → IntValue (n1 + n2)
      | _ → failwith "Not an integer value")
  | Fun (x, e) → Closure (x, e)
  | App (e1, e2) →
    (match eval_lambda e1 with
      | Closure (x, e) → eval_lambda (subst x e2 e)
      | _ → failwith "Not a function")
  | Var x → failwith "Not a value"
```

Lambda Calculus - Call by name

```
type lambda =
  | Num of int
  | Add of lambda * lambda

  | Var of string
  | App of lambda * lambda
  | Fun of string * lambda

let f = Fun ("x", Add (Var "x", Num 1))
let e = App (f, Num 2)
```

```
type value =
  | IntValue of int
  | Closure of string * lambda
```

```
let rec eval_lambda = function
  | Num n → IntValue n
  | Add (e1, e2) →
    (match eval_lambda e1, eval_lambda e2 with
      | IntValue n1, IntValue n2 → IntValue (n1 + n2)
      | _ → failwith "Not an integer value")
  | Fun (x, e) → Closure (x, e)
  | App (e1, e2) →
    (match eval_lambda e1 with
      | Closure (x, e) → eval_lambda (subst x e2 e)
      | _ → failwith "Not a function")
  | Var x → failwith "Not a value"
```

```
(fun x → x + 1) 2
```

Lambda Calculus - Call by name

```
type lambda =
  | Num of int
  | Add of lambda * lambda

  | Var of string
  | App of lambda * lambda
  | Fun of string * lambda

let f = Fun ("x", Add (Var "x", Num 1))
let e = App (f, Num 2)
```

```
type value =
  | IntValue of int
  | Closure of string * lambda
```

```
let rec eval_lambda = function
  | Num n → IntValue n
  | Add (e1, e2) →
    (match eval_lambda e1, eval_lambda e2 with
     | IntValue n1, IntValue n2 → IntValue (n1 + n2)
     | _ → failwith "Not an integer value")
  | Fun (x, e) → Closure (x, e)
  | App (e1, e2) →
    (match eval_lambda e1 with
     | Closure (x, e) → eval_lambda (subst x e2 e)
     | _ → failwith "Not a function")
  | Var x → failwith "Not a value"
```

```
(fun x → x + 1) 2
```

Lambda Calculus - Call by name

```
type lambda =
  | Num of int
  | Add of lambda * lambda

  | Var of string
  | App of lambda * lambda
  | Fun of string * lambda

let f = Fun ("x", Add (Var "x", Num 1))
let e = App (f, Num 2)
```

```
type value =
  | IntValue of int
  | Closure of string * lambda
```

```
let rec eval_lambda = function
  | Num n → IntValue n
  | Add (e1, e2) →
      (match eval_lambda e1, eval_lambda e2 with
       | IntValue n1, IntValue n2 → IntValue (n1 + n2)
       | _ → failwith "Not an integer value")
  | Fun (x, e) → Closure (x, e)
  | App (e1, e2) →
      (match eval_lambda e1 with
       | Closure (x, e) → eval_lambda (subst x e2 e)
       | _ → failwith "Not a function")
  | Var x → failwith "Not a value"
```

Lambda Calculus - Call by name

```
type lambda =
  | Num of int
  | Add of lambda * lambda

  | Var of string
  | App of lambda * lambda
  | Fun of string * lambda

let f = Fun ("x", Add (Var "x", Num 1))
let e = App (f, Num 2)
```

```
type value =
  | IntValue of int
  | Closure of string * lambda
```

```
let rec eval_lambda = function
  | Num n → IntValue n
  | Add (e1, e2) →
    (match eval_lambda e1, eval_lambda e2 with
      | IntValue n1, IntValue n2 → IntValue (n1 + n2)
      | _ → failwith "Not an integer value")
  | Fun (x, e) → Closure (x, e)
  | App (e1, e2) →
    (match eval_lambda e1 with
      | Closure (x, e) → eval_lambda (subst x e2 e)
      | _ → failwith "Not a function")
  | Var x → failwith "Not a value"
```

Lambda Calculus - Call by name

```
type lambda =
  | Num of int
  | Add of lambda * lambda

  | Var of string
  | App of lambda * lambda
  | Fun of string * lambda

let f = Fun ("x", Add (Var "x", Num 1))
let e = App (f, Num 2)
```

```
type value =
  | IntValue of int
  | Closure of string * lambda
```

```
let rec eval_lambda = function
  | Num n → IntValue n
  | Add (e1, e2) →
    (match eval_lambda e1, eval_lambda e2 with
      | IntValue n1, IntValue n2 → IntValue (n1 + n2)
      | _ → failwith "Not an integer value")
  | Fun (x, e) → Closure (x, e)
  | App (e1, e2) →
    (match eval_lambda e1 with
      | Closure (x, e) → eval_lambda (subst x e2 e)
      | _ → failwith "Not a function")
  | Var x → failwith "Not a value"
```

Lambda Calculus - Call by name

```
eval_lambda App (Fun ("x", Add (Var "x", Num 1)), Num 2)
  eval_lambda Fun ("x", Add (Var "x", Num 1))
    = Closure ("x", Add (Var "x", Num 1))
  eval_lambda Num 2
    = IntValue 2

  eval_lambda subst "x" (Num 2) (Add (Var "x", Num 1))
    = eval_lambda (Add (Num 2, Num 1))
    = IntValue 3

= IntValue 3
```

Lambda Calculus - Call by name

- Call-by-name quer dizer que o argumento de uma função não é avaliado antes da chamada de função, mas sim expandido no corpo da função antes de este ser avaliado.
- Alternativas são:
- Call-by-value, avaliar antes de chamar a função.
- Call-by-need, avaliar apenas quando usado, mas apenas uma vez.

```
let rec eval_lambda = function
  | Num n → IntValue n
  | Add (e1, e2) →
      (match eval_lambda e1, eval_lambda e2 with
        | IntValue n1, IntValue n2 → IntValue (n1 + n2)
        | _ → failwith "Not an integer value")
  | Fun (x, e) → Closure (x, e)
  | App (e1, e2) →
      (match eval_lambda e1 with
        | Closure (x, e) → eval_lambda (subst x e2 e)
        | _ → failwith "Not a function")
  | Var x → failwith "Not a value"
```

Alonzo Church (1903 - 1995)

- A linguagem de programação fundamental
- O sistema formal cálculo Lambda

$$E ::= \; x \mid \; \lambda x.E \; \mid \; E \; E$$

$$(\lambda x.E) \; E' \longrightarrow E\{E'/x\} \qquad \frac{E \longrightarrow E''}{E \; E' \longrightarrow E'' \; E'}$$

The End!