# Interpretação e Compilação (de Linguagens de Programação)

# Unidade 2: Sintaxe Abstracta

**Os interpretadores e compiladores são algoritmos que aceitam programas sintaticamente válidos e produzem denotações (valores, efeitos, ou outros programas). Os programas são os dados de entrada para esta classe de algoritmos.**

**Como podemos representar os programas para que sejam processados?**

**Como podemos definir o processamento de programas.**

- Definição inductiva de tipos de dados
- Algoritmos recursivos sobre tipos de dados inductivos
- Representação da sintaxe abstracta como tipo inductivo
- Sintaxe concreta vs sintaxe abstracta (parsing)
- Um programa como dados de input para outro programa

# Sintaxe e Semântica

- **Conceito chave**: definição da semântica de uma linguagem por indução na estrutura sintática da linguagem
- **Conceito chave**: definição da semântica de uma linguagem por meio de um algoritmo interpretador

- Como definir a sintaxe de uma linguagem de programação como um tipo de dados ?

- Como definir a semântica de uma linguagem de programação como um algoritmo interpretador ?

# Tipos de Dados

- Um tipo de dados pode ser representado como um conjunto de valores decidível.
- Tal conjunto de valores pode ser infinito, mas cada valor é "finito".
- Exemplos:

  - Booleanos (**true** / **false**)
  - Números inteiros ( …, -2, -1, 0, 1, 2, … )
  - Listas
  - Árvores binárias
  - etc ...

# Definição de tipos de dados

- Quando o conjunto de valores possíveis é finito, podemos definir os valores do tipo por enumeração :
  - Boolean ≜ { **true**, **false** }
  - BasicColor ≜ { **red**, **green**, **blue** }
- Como definir o conjunto dos valores de um tipo quando o número de valores possível é infinito ?
  - NaturalNumber ≜ ?
  - List ≜ ?
  - Tree ≜ ?

# Definição indutiva de dados

- **O mecanismo de definição "universalmente" usado em informática é a definição indutiva.**

O tipo *NaturalNumber* é definido pelas seguintes regras :

1. ***0*** é um número natural (caso base)
2. se ***n*** é um número natural, então ***succ(n)*** é também um número natural
3. não existem mais números naturais, excepto os construídos de acordo com as regras 1 e 2 acima.

# Definição indutiva de dados

- **O mecanismo de definição "universalmente" usado em informática é a definição indutiva.**

O tipo *List* (de elementos de tipo T) é definido por:

1. ***nil*** é uma lista (a lista vazia)
2. se ***x*** é um valor de tipo T e ***L*** é uma lista, então ***cons***(***x***, ***L***) é também uma lista
3. não existem mais listas, excepto as construídas de acordo com as regras 1 e 2 acima.

```
type list = Nil | Cons of int * list
```

# Definição indutiva de dados

- **Todas as definições indutivas de tipos de dados obedecem ao mesmo padrão:**

O tipo *LabeledTree* (de valores de tipo T) é definido por:

1. ***empty*** é uma árvore (a árvore vazia)
2. se x é um valor de tipo T, e T1 e T2 são árvores, então ***node***(***T1***, ***x***, ***T2***) também é uma árvore, com raiz etiquetada por ***x***, cuja subárvore esquerda é ***T1*** e cuja subárvore direita é ***T2***.
3. Não existem mais … 1. e 2.

```
type tree = Empty | Node of tree * int * tree
```

# Definição indutiva de dados

- **Todas as definições indutivas de tipos de dados obedecem ao mesmo padrão:**

O tipo *Stack* (de valores de tipo T)

1. ***empty*** é uma pilha (a pilha vazia)
2. se ***x*** é um valor de tipo T, e ***S*** é uma pilha, então ***push***(***x***,***S***) também é uma pilha, cujo topo contém o valor x, por trás do qual está a pilha S.
3. Não existem mais … 1. e 2.

```
type stack = Empty | Push of int * stack
```

# Definição indutiva de dados

- Os ingredientes da definição indutiva de um tipo de dados são:

1. o **nome** do tipo T
2. um conjunto de **construtores**

- Um construtor é uma função que permite construir novos valores do tipo T a partir de valores já construídos do mesmo tipo ou de outros tipos.
- Cada construtor tem uma assinatura, que indica os tipos dos seus parâmetros.
- O tipo resultado de cada construtor do tipo T é o tipo T

# Definição indutiva de dados

- Elementos da definição indutiva do tipo *lista* de valores de tipo **integer**

- o nome do tipo: ***ListInt***
- um conjunto de construtores: **nil, cons**

**nil**: () → ListInt

**cons**: Integer × ListInt → ListInt

# Definição indutiva de dados

- Elementos da definição indutiva do tipo *lista* de valores de tipo **integer**

- o nome do tipo: ***ListInt***
- um conjunto de construtores: **nil**, **cons**

**nil**: () → ListInt

**cons**: Integer × ListInt → ListInt

```
/* ListInt.h */
typedef struct ListInt ListInt;
ListInt* nil();
ListInt* cons(int elem, ListInt *tail);
```

# Definição indutiva de dados

- Elementos da definição indutiva do tipo *lista* de valores de tipo **integer**

- o nome do tipo: ***ListInt***
- um conjunto de construtores: **nil**, **cons**

**nil**: () → ListInt

**cons**: Integer × ListInt → ListInt

```
class ListInt {
    static ListInt nil();
    static ListInt cons(int elem, ListInt tail);
};
```

# Definição indutiva de dados

- Elementos da definição indutiva do tipo *árvore etiquetada* de valores de tipo **integer**

- o nome do tipo: **TreeInt**
- um conjunto de construtores: **empty**, **node**

**empty**: () → TreeInt

**node**: TreeInt × Integer × TreeInt → TreeInt

# Definição indutiva de dados

- Elementos da definição indutiva do tipo *árvore etiquetada* de valores de tipo **integer**

- o nome do tipo: **Treeint**
- um conjunto de construtores: **empty, node**

**empty**: () → TreeInt

**node**: TreeInt × Integer × TreeInt → TreeInt

```
interface TreeInt {
    TreeInt();
    TreeInt(TreeInt l, int elem, TreeInt r);
};
```

# Definição indutiva de dados

- Elementos da definição indutiva do tipo *árvore etiquetada* de valores de tipo **integer**

- o nome do tipo: **Treeint**
- um conjunto de construtores: **empty, node**

**empty**: () → TreeInt

**node**: TreeInt × Integer × TreeInt → TreeInt

```
/* TreeInt.h */
typedef struct TreeInt TreeInt;
TreeInt* empty();
TreeInt* node(TreeInt *l, int elem, TreeInt *r);
```

# Definição indutiva de algoritmos

**The Smaller-Subproblem Principle**

*If we can reduce a problem to a smaller subproblem, we can call the procedure that solves the problem to solve the subproblem.*

- A definição de algoritmos que operam sobre tipos de dados de tipo indutivo, funcionam por **análise de casos** no construtores e decomposição do problema em problemas mais pequenos (valores mais pequenos).
- Dado um valor *v* qualquer de um tipo T, o algoritmo
  - verifica-se qual é o último construtor aplicado na construção de *v* (por exemplo, *v* é cons(3, cons(2,nil)));
  - extraem-se os componentes aos quais o construtor foi aplicado (neste caso, o inteiro 3 e a lista cons(2,nil));
  - aplica-se recursivamente aos componentes de tipo T. Assim, obtêm-se os resultados intermédios que permitem produzir o resultado final.

# Definição indutiva de algoritmos

- Tipo **ListInt**
- Construtores: **nil, cons**

- Algoritmo length(L) para calcular o **comprimento** de uma lista qualquer L:

se L é da forma **nil**

length(L) ≜ 0

se L é da forma **cons**(x,L')

length(L) ≜ 1+length(L')

```
type listInt = Nil
             | cons of int * listInt


let rec length l = match l with
            Nil -> 0
          | Cons(x,l) -> 1+(length l)
```

# Definição indutiva de algoritmos

- Tipo **ListInt**
- Construtores: **nil, cons**

- Algoritmo sum(L) para calcular a **soma** dos valores numa lista qualquer L:

se L é da forma **nil**

sum(L) ≜ 0

se L é da forma **cons**(x,L')

sum(L) ≜ x+sum(L')

```
type listInt = Nil
             | cons of int * listInt


let rec sum l = match l with
            Nil -> 0
          | Cons(x,l) -> 1+(sum l)
```

# Definição indutiva de algoritmos

- Tipo **TreeInt**
- Construtores: **empty, node**
- Algoritmo numnodes(T) para calcular o **número de nós** numa árvore qualquer T:

se T é da forma **empty**

numnodes(L) ≜ 0

se T é da forma **node**(L', x, L'')

numnodes(L) ≜ 1+numnodes(L')+numnodes(L'')

```
type treeInt = Empty
  | Node of treeInt * int * treeInt
let rec numnodes t = match t with
        Empty -> 0
      | Node(l,x,r) -> 1+(numnodes l)+(numnodes r)
```

# Definição indutiva de algoritmos

- Tipo **TreeInt**
- Construtores: **empty**, **node**
- Algoritmo depth(T) para calcular a altura de uma árvore qualquer T:

se T é da forma **empty**

depth(T) ≜ 0

se T é da forma **node**(L',x,L'')

depth(T) ≜ 1+**max**(depth(L'), depth(L''))

```
type treeInt = Empty
  | Node of treeInt * int * treeInt
let rec depth t = match t with
        Empty -> 0
      | Node(l,x,r) -> 1+(max (depth l) (depth r))
```

# Definição indutiva de programas

- O conjunto de todos os programas de uma linguagem de programação pode ser visto como um tipo de dados
- É fácil definir indutivamente o conjunto de todos os programas de uma linguagem de programação
- A definição indutiva de linguagens melhorou o desenho das mesmas (Fortran/spaghetti code vs. Pascal/estrutura em blocos)

- Discussão:
  definição indutiva de algoritmos versus definição indutiva de programas
  (a que se referem as duas ocorrências do adjectivo "indutivo"?)

# Definição indutiva de programas

## Copyright Notice

The following manuscript

EWD 215: A Case against the GO TO Statement

was published as a letter entitled

Go-to statement considered harmful

in *Commun. ACM 11* (1968), 3:
permission.

A Case against the GO TO Statement.

by Edsger W.Dijkstra
Technological University
Eindhoven, The Netherlands

Since a number of years I am familiar with the obse
quality of programmers is a decreasing function of the de
statements in the programs they produce. Later I discover
the go to statement has such disastrous effects and did I
that the go to statement should be abolished from all "hi
programming languages (i.e. everything except -perhaps- p

# Estruturação hierárquica de linguagens

```
10 for I=1 to 10 do
20 if I<10 goto 40
25 for J=I to 20 do
30 next I
40 next J
```

```
for i=1 to 10 do
    begin
       if i<10 then
            begin

            end
    end;
```

# Exemplo: A Linguagem CALC

- CALC é uma linguagem simples de expressões aritméticas.
- Cada programa da linguagem CALC é uma expressão algébrica construída com base em numerais inteiros, nos quatro operadores aritméticos básicos (+, −, ×, ÷ ) e nos parêntesis.
- Exemplos:

```
(21+32) * 42
2/(7-2)
```

- A semântica pretendida para a linguagem CALC é a esperada para uma linguagem de expressões aritméticas:

  a denotação de cada expressão da linguagem CALC é o seu valor.

# Semântica da linguagem CALC

Em geral, a semântica de uma linguagem pode ser caracterizada por uma função I que atribui um significado (ou denotação) a cada programa (ou fragmento de programa) sintacticamente correcto.

- No caso da linguagem CALC:

$$I : CALC \rightarrow Integer$$

CALC ≡ conjunto dos programas válidos

Integer = conjunto dos significados (denotações)

Como representar a linguagem CALC como um tipo de dados?

# A Linguagem CALC (como tipo indutivo)

- Tipo de dados CALC com os construtores: **num, add, mul, div, sub**

$$\begin{aligned}
\textbf{num}&: \text{Integer} \rightarrow \text{CALC}\\
\textbf{add}&: \text{CALC} \times \text{CALC} \rightarrow \text{CALC}\\
\textbf{mul}&: \text{CALC} \times \text{CALC} \rightarrow \text{CALC}\\
\textbf{div}&: \text{CALC} \times \text{CALC} \rightarrow \text{CALC}\\
\textbf{sub}&: \text{CALC} \times \text{CALC} \rightarrow \text{CALC}
\end{aligned}$$

- Cada valor do tipo indutivo CALC representa uma expressão na linguagem CALC. Diz-se que o tipo indutivo CALC define a sintaxe abstracta da linguagem CALC
- A sintaxe concreta de uma linguagem:
  - Caracteriza a forma como as suas expressões e programas são efectivamente escritos em termos de sequências de caracteres, formatação, etc …
- A sintaxe abstracta de uma linguagem:
  - Caracteriza a estrutura das suas expressões e programas, apresentando essa estrutura em termos de construtores abstractos.

# Sintaxe abstracta vs Sintaxe concreta

- Constantes inteiras
  - em decimal: 12
  - em hexadecimal: 0x0C
  - sintaxe abstracta: num(12)
- Variáveis
  - em Pascal: A
  - em bash: %A
  - sintaxe abstracta: var("A")
- Afectação
  - em C: x = 2
  - em Pascal: x := 2
  - sintaxe abstracta: assign(var("x"),num(2))

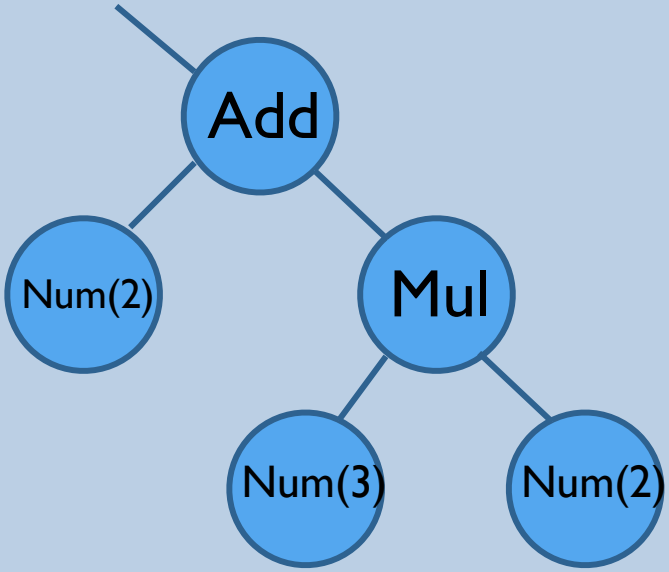
# Sintaxe abstracta vs Sintaxe concreta

- Expressões algébricas
  - em C: 2*3+2
  - em RPN: 2 3 * 2 +
  - em Lisp: (+ (* 2 3) 2)
  - sintaxe abstracta: add(mul(num(2),num(3)),num(2))
- Blocos
  - em C: {S1 S2 … Sn }
  - em Pascal: begin S1; S2; …; Sn end
  - sintaxe abstracta: block(S1,S2,…,Sn)
- Ciclo while:
  - em C: while (C) S
  - em Pascal: while C do S
  - sintaxe abstracta: while(C,S)

# Árvore Sintáctica Abstracta

- Representa a estrutura de uma frase da linguagem em termos dos seus constructores abstractos.
- Abstract Syntax Tree (AST)

2 + 3 * 2




(2 + 3)* 2