DESENVOLVIMENTO ORIENTADO À TIPOS

Tipos de Dados Algébricos (ADT)

TÓPICOS

01 Definição de Tipos de Dados Algébricos

02 Tipos de Soma (União)

03 Tipos de Produto (interseção)

04 Tipos Recursivo

05 Tipos Exponencial

06 Exemplos práticos

DEFINIÇÃO

Tipos de dados algébricos são estruturas compostas por combinações de tipos simples usando operações algébricas como soma e produto

POR QUE TIPOS DE DADOS ALGÉBRICOS?

Expressividade

Permitem modelar estruturas de dados complexas de forma clara e concisa.

Segurança de Tipos

Detectam erros em tempo de compilação, garantindo manipulação correta dos dados.

Abstração de Dados

Encapsulam detalhes de implementação, facilitando a interação com os dados.

Padrões de Programação

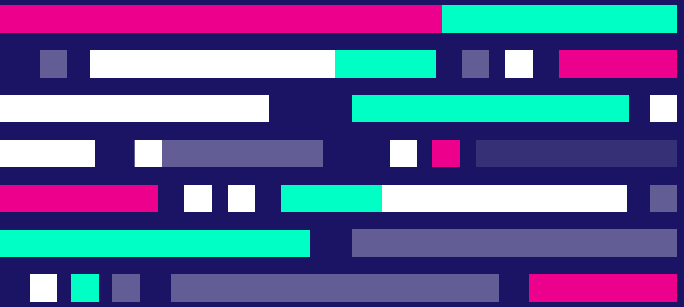
Promovem práticas que deixam o código mais limpo e modular.

Adaptabilidade

Sua flexibilidade permite abordar uma variedade extensa de problemas, tornando-os fundamentais no código.

Semântica

Permitem a criação de tipos de dados que refletem diretamente a semântica do problema, facilitando a compreensão e manutenção do código ao longo do tempo.



CATEGORIAS

Os tipos de dados algébricos podem ser divididos em quatro categorias principais: **tipos de soma** (ou tipos de união), **tipos de produto** (ou tipos de interseção), **tipos recursivos** e **tipos exponenciais**.

TIPOS DE SOMA

SOMA

representam uma escolha entre diferentes alternativas, onde cada opção é definida por um construtor de dados.

Alternativas

permitem representar uma escolha entre diferentes alternativas ou casos. Cada caso é definido por um construtor de dados distinto.

Construtores

Cada alternativa é representada por um construtor de dados específico, que pode ter parâmetros associados para expressar informações adicionais sobre o caso.

Disjunção de Casos

Os diferentes construtores de dados representam casos disjuntos entre si. A escolha de um caso exclui automaticamente as outras alternativas.

Flexibilidade e Expressividade

Esses tipos de dados fornecem uma maneira flexível e expressiva de modelar problemas que envolvem escolhas discretas ou múltiplas possibilidades, tornando-os valiosos em situações onde a lógica de negócios envolve alternativas claras.

EXEMPLO (C/C++)

```
enum DiaDaSemana {
    Segunda,
    Terca,
    Quarta,
    Quinta,
    Sexta,
    Sabado,
    Domingo
};
```

EXEMPLO (Python)

```python
from enum import Enum

class DiaDaSemana(Enum):
    SEGUNDA = 1
    TERCA = 2
    QUARTA = 3
    QUINTA = 4
    SEXTA = 5
    SABADO = 6
    DOMINGO = 7
```

EXEMPLO (Haskell)

```
data DiaDaSemana = Segunda | Terca | Quarta | Quinta
 | Sexta | Sabado | Domingo
```

EXEMPLO (Haskell)

```
data DiaDaSemana = Segunda | Terca | Quarta | Quinta
  | Sexta | Sabado | Domingo
```

Declaração de um novo tipo de dados
chamado DiaDaSemana

EXEMPLO (Haskell)

```
data DiaDaSemana = Segunda | Terca | Quarta | Quinta
  | Sexta | Sabado | Domingo
```

Construtor de dado, também chamado de caso. Cada um desses construtores representa um dia da semana específico. Cada construtor representa uma alternativa ou caso distinto e não tem parâmetros associados, indicando que são constantes sem informações adicionais.

TIPOS DE PRODUTO

PRODUTO

combinam múltiplos valores em uma única estrutura, onde cada valor está associado a um campo particular.

Combinação de Valores

permitem combinar múltiplos valores em uma única estrutura.

Campos Distintos

Cada valor na estrutura está associado a um campo específico, que pode conter diferentes tipos de dados.

Estruturação de Informações

organizam as informações em uma estrutura que permite a representação de entidades complexas e estruturadas, cada campo contendo dados distintos e específicos.

Flexibilidade Estrutural

Eles oferecem flexibilidade na definição de estruturas de dados complexas, permitindo a organização eficiente e acessível de informações relacionadas em um único objeto.

EXEMPLO (C/C++)

```cpp
#include <iostream>
#include <cmath>
using namespace std;

struct Circulo {
    float raio;
};

struct Retangulo {
    float largura;
    float altura;
};

float calcularArea(const Circulo& circulo) {
    return M_PI * pow(circulo.raio, 2);
}

float calcularArea(const Retangulo& retangulo) {
    return retangulo.largura * retangulo.altura;
}

int main() {
    Circulo circulo = {2.0};
    Retangulo retangulo = {3.0, 4.0};

    return 0;
}
```

EXEMPLO (Python)

```python
import math

class Circulo:
    def __init__(self, raio):
        self.raio = raio

    def calcular_area(self):
        return math.pi * self.raio ** 2

class Retangulo:
    def __init__(self, largura, altura):
        self.largura = largura
        self.altura = altura

    def calcular_area(self):
        return self.largura * self.altura

circulo = Circulo(2.0)
retangulo = Retangulo(3.0, 4.0)
```

EXEMPLO (Haskell)

```haskell
data Forma = Circulo Float | Retangulo Float Float

calcularArea :: Forma -> Float
calcularArea (Circulo raio) = pi * raio ^ 2
calcularArea (Retangulo largura altura) = largura *
altura

circulo = Circulo 2.0
retangulo = Retangulo 3.0 4.0
```

EXEMPLO (Haskell)

```haskell
data Forma = Circulo Float | Retangulo Float Float

calcularArea :: Forma -> Float
calcularArea (Circulo raio) = pi * raio ^ 2
calcularArea (Retangulo largura altura) = largura *
altura

circulo = Circulo 2.0
retangulo = Retangulo 3.0 4.0
```

Declaração de um novo tipo de dados chamado Forma

EXEMPLO (Haskell)

```
data Forma = Circulo Float | Retangulo Float Float

calcularArea :: Forma -> Float
calcularArea (Circulo raio) = pi * raio ^ 2
calcularArea (Retangulo largura altura) = largura *
altura

circulo = Circulo 2.0
retangulo = Retangulo 3.0 4.0
```

Este é um construtor de dados para o tipo Forma, que representa um círculo. O Float é um parâmetro que representa o raio do círculo.

EXEMPLO (Haskell)

```
data Forma = Circulo Float | Retangulo Float Float

calcularArea :: Forma -> Float
calcularArea (Circulo raio) = pi * raio ^ 2
calcularArea (Retangulo largura altura) = largura *
altura

circulo = Circulo 2.0
retangulo = Retangulo 3.0 4.0
```

Este é outro construtor de dados para o tipo Forma, que representa um retângulo. Os dois Floats são parâmetros que representam a largura e altura do retângulo, respectivamente.

EXEMPLO (Haskell)

```haskell
data Forma = Circulo Float | Retangulo Float Float

calcularArea :: Forma -> Float
calcularArea (Circulo raio) = pi * raio ^ 2
calcularArea (Retangulo largura altura) = largura *
altura

circulo = Circulo 2.0
retangulo = Retangulo 3.0 4.0
```

Esta é uma assinatura de tipo para a função calcularArea. Ela recebe um valor do tipo Forma e retorna um valor Float.

EXEMPLO (Haskell)

```
data Forma = Circulo Float | Retangulo Float Float

calcularArea :: Forma -> Float
calcularArea (Circulo raio) = pi * raio ^ 2
calcularArea (Retangulo largura altura) = largura *
altura

circulo = Circulo 2.0
retangulo = Retangulo 3.0 4.0
```

Esta é uma definição de padrão para a função calcularArea quando é fornecido um círculo. Ela calcula a área do círculo usando a fórmula pi * raio ^ 2.

EXEMPLO (Haskell)

```haskell
data Forma = Circulo Float | Retangulo Float Float

calcularArea :: Forma -> Float
calcularArea (Circulo raio) = pi * raio ^ 2
calcularArea (Retangulo largura altura) = largura * altura

circulo = Circulo 2.0
retangulo = Retangulo 3.0 4.0
```

Esta é outra definição de padrão para a função calcularArea quando é fornecido um retângulo. Ela calcula a área do retângulo multiplicando a largura pela altura.

EXEMPLO (Haskell)

```haskell
data Forma = Circulo Float | Retangulo Float Float

calcularArea :: Forma -> Float
calcularArea (Circulo raio) = pi * raio ^ 2
calcularArea (Retangulo largura altura) = largura *
altura

circulo = Circulo 2.0
retangulo = Retangulo 3.0 4.0
```

Isso cria um valor do tipo Forma representando um círculo com raio 2.0.

TIPOS RECURSIVOS

RECURSIVOS

Tipos recursivos são tipos de dados que se referem a si mesmos em suas definições, comuns em estruturas hierárquicas como listas e árvores.

Referência a Si Mesmo

Os tipos recursivos referem-se a si mesmos em suas próprias definições, criando uma estrutura auto-referencial.

Estruturas Hierárquicas

São comumente usados para representar estruturas de dados hierárquicas, como listas, árvores e grafos.

Recorrência

A definição do tipo é recursiva, o que significa que a estrutura pode conter instâncias de si mesma como parte de sua definição.

Flexibilidade

Permitem a modelagem de dados de forma flexível e aninhada, adequada para representar problemas que envolvem repetição ou aninhamento de informações.

EXEMPLO (C/C++)

```
struct No {
    int valor;
    No* proximo;
};
```

EXEMPLO (Python)

```python
class No:
    def __init__(self, valor):
        self.valor = valor
        self.proximo = None
```

EXEMPLO (Haskell)

```
data No = No Int No | Vazio
```

EXEMPLO (Haskell)

```
data No = No Int No | Vazio
```

Define um novo tipo de dados chamado No.

EXEMPLO (Haskell)

```
data No = No Int No | Vazio
```

Este é um construtor de dados para o tipo No. Ele representa um nó em uma lista encadeada, onde cada nó contém um valor inteiro (Int) e uma referência para o próximo nó na lista (No). Isso cria uma estrutura recursiva, pois No é usado dentro de sua própria definição.

EXEMPLO (Haskell)

```
data No = No Int No | Vazio
```

Este é outro construtor de dados para o tipo No, que representa o final da lista encadeada ou uma lista vazia. Ele é usado quando não há mais nós para referenciar.

TIPOS EXPONENCIAL

EXPONENCIAL

Os tipos exponenciais permitem representar uma quantidade exponencial de valores distintos, utilizando múltiplos construtores e/ou argumentos.

Representação Abundante

Permitem representar uma vasta gama de valores distintos, muitas vezes exponencialmente mais do que outros tipos de dados.

Múltiplos Construtores

Podem ter múltiplos construtores, cada um representando uma alternativa distinta, aumentando ainda mais a diversidade de valores possíveis.

Variedade de Argumentos

Podem ter múltiplos argumentos em cada construtor, permitindo uma combinação flexível de informações para representar diferentes estados ou opções.

Expressividade

São úteis para modelar situações complexas onde há uma grande quantidade de alternativas ou estados possíveis, proporcionando uma representação concisa e expressiva.

EXEMPLO PRÁTICO DE USO DE TIPOS DE DADOS ALGÉBRICOS

```haskell
data Funcionario = Gerente { nome :: String, idade :: Int, salario :: Float }
                 | Desenvolvedor { nome :: String, idade :: Int, linguagem :: String, salario :: Float }
                 | Estagiario { nome :: String, idade :: Int, curso :: String, salario :: Float }

salarioTotal :: Funcionario -> Float
salarioTotal (Gerente _ _ salario) = salario
salarioTotal (Desenvolvedor _ _ _ salario) = salario
salarioTotal (Estagiario _ _ _ salario) = salario

main = do
    let gerente = Gerente "João" 35 5000.0
    let desenvolvedor = Desenvolvedor "Maria" 28 "Python" 4000.0
    let estagiario = Estagiario "Pedro" 22 "Ciência da Computação" 1500.0

    putStrLn $ "Salário total do gerente: " ++ show (salarioTotal gerente)
    putStrLn $ "Salário total do desenvolvedor: " ++ show (salarioTotal desenvolvedor)
    putStrLn $ "Salário total do estagiário: " ++ show (salarioTotal estagiario)
```

EXEMPLO PRÁTICO DE USO DE TIPOS DE DADOS ALGÉBRICOS

```haskell
data Midia = Livro { titulo :: String, autor :: String, numPaginas :: Int }
           | Filme { titulo :: String, diretor :: String, duracao :: Int }
           | Musica { titulo :: String, artista :: String, duracao :: Int }

duracaoTotal :: [Midia] -> Int
duracaoTotal [] = 0
duracaoTotal (m:ms) = case m of
    Livro {} -> duracaoTotal ms
    Filme {duracao = d} -> d + duracaoTotal ms
    Musica {duracao = d} -> d + duracaoTotal ms

main = do
    let biblioteca = [ Livro "A Revolução dos Bichos" "George Orwell" 300
                     , Filme "Interestelar" "Christopher Nolan" 136
                     , Musica "Stairway to Heaven" "Led Zeppelin" 354 ]

    putStrLn $ "Duração total da biblioteca: " ++ show (duracaoTotal biblioteca) ++ " minutos"
```

GITHUB

Os exemplos estão disponíveis no Github: https://github.com/DanielCreeklear/HaskellUFABC/tree/main/Trabalhos