The background image shows a top-down view of a wooden desk. On the desk is a spiral-bound notebook with several hand-drawn sketches of user interface elements, including login forms, navigation bars, and content layouts. The sketches are drawn with black and green markers. The word '#Content' is written in large, bold, black cursive at the top of the notebook. A hand is visible in the bottom right corner, holding a black pen and pointing towards the sketches. A white computer keyboard is partially visible in the top right corner. The overall scene suggests a design or development workshop.

Semana 5 – Programação Funcional e Lógica

Prof. Cassiano Moralles

Programação Funcional

```
def set_mirror_object_to_mirror(modifier_obj, mirror_mod, mirror_object):  
    operation == "MIRROR_X":  
    mirror_mod.use_x = True  
    mirror_mod.use_y = False  
    mirror_mod.use_z = False  
    operation == "MIRROR_Y":  
    mirror_mod.use_x = False  
    mirror_mod.use_y = True  
    mirror_mod.use_z = False  
    operation == "MIRROR_Z":  
    mirror_mod.use_x = False  
    mirror_mod.use_y = False  
    mirror_mod.use_z = True
```

```
selection at the end -add  
mirror_ob.select= 1  
modifier_ob.select=1
```

```
by.context.objects.active  
by.context.selected_objects
```

```
mirror_ob.select = 0  
by.context.selected_objects
```

```
by.context.objects[one.name].select
```

```
print("please select exactly
```

```
--- OPERATOR CLASSES ---
```

```
types.Operator):
```

```
    X mirror to the selected
```

```
object.mirror_mirror_x"
```

```
mirror X"
```

```
context):
```

```
object is not
```

Programação Funcional



O paradigma de programação funcional, baseado em funções matemáticas, é a base de projeto para um dos estilos de linguagem não imperativos mais importantes.



Esse estilo de programação é suportado por linguagens de programação funcional (ou linguagens aplicativas).



A programação funcional emergiu como um paradigma distinto no início da década de 1960.

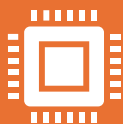


criação foi motivada pela necessidade dos pesquisadores no desenvolvimento de inteligência artificial e em seus subcampos – computação simbólica, prova de teoremas, sistemas baseados em regras e processamento de linguagem natural.



Essas necessidades não eram particularmente bem atendidas pelas linguagens imperativas da época.

Programação Funcional



A linguagem funcional original era a Lisp, desenvolvida por John McCarthy (McCarthy, 1960) e descrita no LISP 1.5 Programmer's Manual (McCarthy et al., 1965).

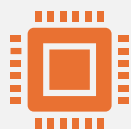


A descrição é notável tanto pela sua clareza quanto pela sua brevidade; o manual tem apenas 106 páginas!



A linguagem Lisp serve primariamente para processamento simbólico de dados. Ela tem sido usada para cálculos simbólicos em cálculo diferencial e integral, projeto de circuitos elétricos, lógica matemática, jogos e outros campos da inteligência artificial.

Programação Funcional

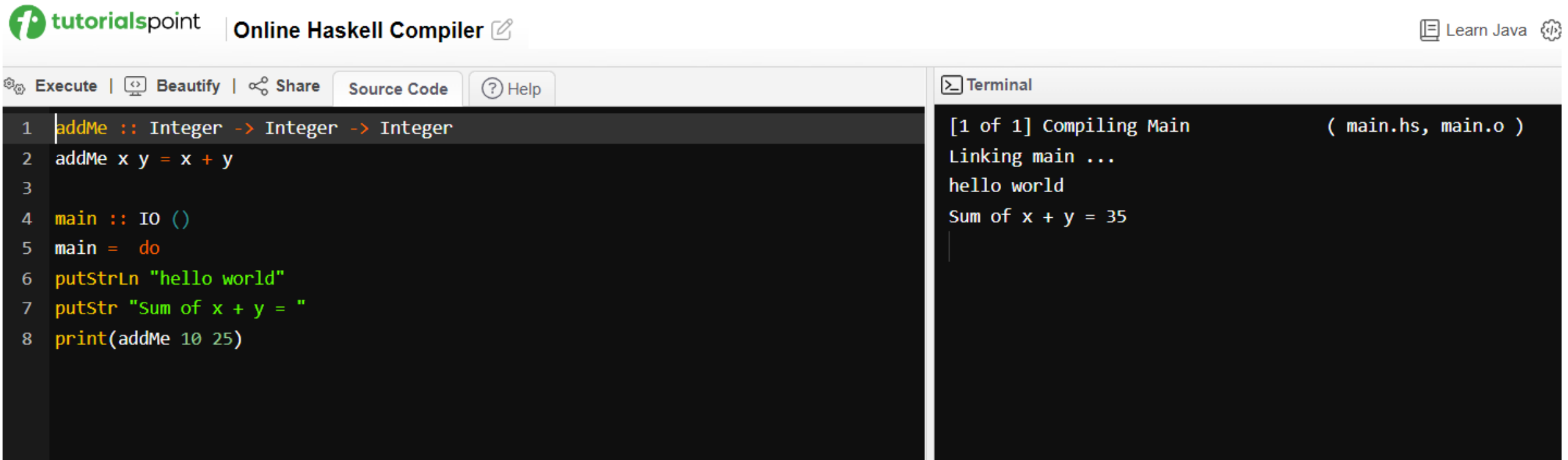


Em sua palestra do Prêmio Turing, Backus (1978) argumentou que linguagens de programação puramente funcionais são melhores do que linguagens imperativas porque resultam em programas mais legíveis, mais confiáveis e mais propensos a serem corretos.



Os programas puramente funcionais eram mais fáceis de serem entendidos, tanto durante quanto após o desenvolvimento, em grande parte porque os significados das expressões são independentes de seu contexto (um recurso característico de uma linguagem de programação funcional pura é que nem expressões, nem funções, têm efeitos colaterais).

Programação Funcional



The screenshot displays the 'Online Haskell Compiler' interface on the tutorialspoint website. The interface is divided into two main sections: a source code editor on the left and a terminal window on the right.

Source Code Editor: The editor contains the following Haskell code:

```
1 addMe :: Integer -> Integer -> Integer
2 addMe x y = x + y
3
4 main :: IO ()
5 main = do
6   putStrLn "hello world"
7   putStr "Sum of x + y = "
8   print(addMe 10 25)
```

Terminal Window: The terminal shows the output of the compilation and execution:

```
[1 of 1] Compiling Main           ( main.hs, main.o )
Linking main ...
hello world
Sum of x + y = 35
```

The interface includes a top navigation bar with the 'tutorialspoint' logo, the title 'Online Haskell Compiler', and a 'Learn Java' link. Below the title bar, there are tabs for 'Execute', 'Beautify', 'Share', 'Source Code', and 'Help'.

Programação Funcional

- A base da programação funcional é o cálculo lambda, desenvolvido por Church (1941). Uma expressão lambda especifica os parâmetros e a definição de uma função, mas não seu nome. Por exemplo, veja a seguir uma expressão lambda que define a função $\text{square} = x * x$.

$(\lambda x . x * x)$

- O identificador x é um parâmetro usado no corpo (sem nome) da função $x * x$.
- A aplicação de uma expressão lambda a um valor é representada por: $((\lambda x . x * x) 2)$ que dá como resultado 4

Programação Funcional

Este exemplo é uma ilustração de um cálculo lambda aplicado. O que Church realmente definiu foi um cálculo lambda *puro* ou *não-interpretado*, da seguinte maneira:

- 1 Qualquer identificador é uma expressão lambda.
- 2 Se M e N forem expressões lambda, então a *aplicação* de M a N , escrito como $(M\ N)$, é uma expressão lambda.
- 3 Uma *abstração*, escrita como $(\lambda x \cdot M)$, na qual x é um identificador e M é uma expressão lambda, é também uma expressão lambda.

Um conjunto simples de regras gramaticais BNF para a sintaxe desse cálculo lambda puro pode ser escrito como:

$$\text{ExpressãoLambda} \rightarrow \text{variable} \mid (M\ N) \mid (\lambda\ \text{variable} \cdot M)$$
$$M \rightarrow \text{ExpressãoLambda}$$
$$N \rightarrow \text{ExpressãoLambda}$$

Programação Funcional



Um aumento na produtividade



Eficiência de execução é outra base para comparação. Quando programas funcionais são interpretados, eles são mais lentos do que seus correspondentes imperativos.



Entretanto, existem agora compiladores para a maioria das linguagens funcionais, então as disparidades de velocidade de execução entre funcionais e imperativas compiladas não são mais tão grandes.



Potencial das linguagens funcionais é a legibilidade

Programação Funcional

Em linguagem C:

```
int sum_cubes(int n) {  
    int sum = 0;  
    for(int index = 1; index <= n; index++)  
        sum += index * index * index;  
    return sum;  
}
```

Em Haskell, a função poderia ser:

```
sumCubes n = sum (map (^3) [1..n])
```

Essa versão simplesmente especifica três passos:

1. Construa a lista de números ([1..n]).
2. Crie uma nova lista mapeando uma função que computa a terceira potência de cada um dos números da lista.
3. Some a nova lista.

Por causa dos detalhes de variáveis e de controle de iteração, essa versão é mais legível do que a em C*.

Programação Lógica

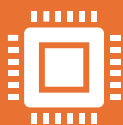
The image features a close-up, high-angle view of a complex electronic circuit board. The board is dark, with intricate patterns of glowing blue and white lines representing traces and components. On the right side, a large, rectangular area is filled with glowing binary code (0s and 1s) in a light blue font. The text 'Programação Lógica' is prominently displayed in the center in a white, sans-serif font. The overall aesthetic is futuristic and technological, with a strong emphasis on light and shadow.

Programação Lógica

A programação lógica (declarativa) surgiu como um paradigma distinto nos anos 70. A programação lógica é diferente dos outros paradigmas porque ela requer que o programador declare os objetivos da computação, em vez dos algoritmos detalhados por meio dos quais esses objetivos podem ser alcançados.

Os objetivos são expressos como uma coleção de asserções, ou regras, sobre os resultados e as restrições da computação. Por essa razão, a programação lógica, às vezes, é chamada programação baseada em regras.

Programação Lógica



As aplicações de programação declarativa se classificam em dois domínios principais: inteligência artificial e acesso de informações em bancos de dados.



No campo da inteligência artificial, Prolog tem sido influente. Alguns subcampos da inteligência artificial usam outras linguagens declarativas, como MYCIN, para modelar sistemas especializados.



Na área de bancos de dados, a Structured Query Language (SQL) tem sido bem popular.

Programação Lógica



Duas características interessantes e diferenciadas dos programas lógicos são não-determinismo e backtracking.



Um programa lógico não-determinístico pode encontrar várias soluções para um problema em vez de apenas uma, como seria normal em outros domínios de programação.




Além disso, o mecanismo backtracking que possibilita o não-determinismo está dentro do interpretador Prolog, e, portanto, é implícito em todos os programas Prolog. Ao contrário, o uso de outras linguagens para escrever programas backtracking requer que o programador defina o mecanismo de backtracking explicitamente.

Programação Lógica

- O mecanismo de backtracking é uma técnica utilizada em linguagens de programação lógica, como Prolog, para resolver problemas que envolvem busca e exploração de espaços de solução.
- A ideia principal do backtracking é tentar construir uma solução passo a passo, abandonando um caminho assim que se percebe que ele não leva a uma solução viável, e retrocedendo para tentar alternativas diferentes.
- O processo de backtracking envolve as seguintes etapas:
 - 1.Escolha:** Selecione uma opção ou faça uma escolha que parece ser promissora.
 - 2.Exploração:** Tente expandir essa escolha e ver se leva a uma solução completa.
 - 3.Verificação:** Verifique se a escolha atual levou a uma solução viável.
 - 4.Retrocesso:** Se a escolha atual não leva a uma solução viável, retroceda para a escolha anterior e tente uma alternativa diferente.

Programação Lógica


```
filho_geral(Y,X) :- progenitor(X,Y).
mãe(X,Y) :- progenitor(X,Y),mulher(X).
avô_geral(X,Z) :- progenitor(X,Y), progenitor(Y,Z).
irmão(X,Y) :- progenitor(Z,X), progenitor(Z,Y), homem(X).
ancestral(X,Z) :- progenitor(X,Z).
ancestral(X,Z) :- progenitor(X,Y), ancestral(Y,Z).
progenitor(sara,isaque).
progenitor(abraão,isaque).
progenitor(abraão,ismael).
progenitor(isaque,esaú).
progenitor(isaque,jacó).
progenitor(jacó,josé).
mulher(sara).
homem(abraão).
homem(isaque).
homem(ismael).
homem(esaú).
homem(jacó).
homem(josé).
```

 `ancestral(ismael, who).`

false

 `ancestral(who, ismael).`

false

 `ancestral(who, ismael).`

false

 `progenitor(who, ismael).`

false

 `progenitor(abraão, ismael).`

true

 `progenitor(abraão, X).`

X = isaque

Next

10

100

1,000

Stop

?- `progenitor(abraão, X).`