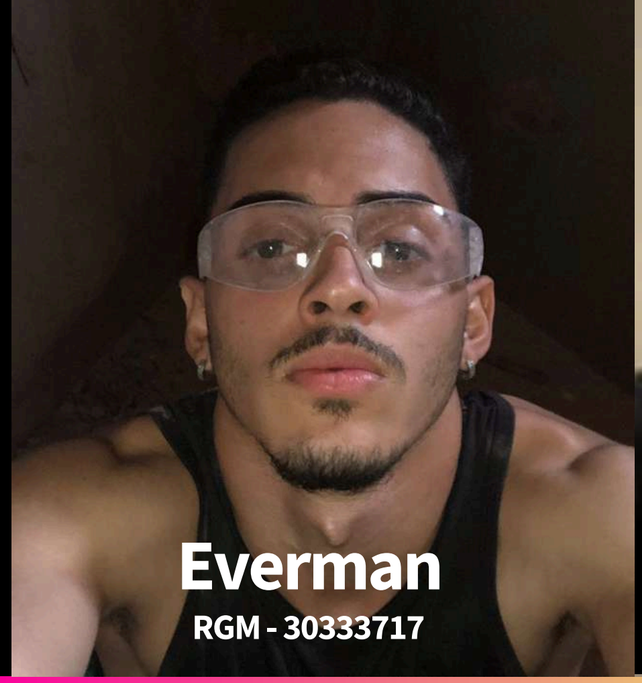
Grupo Preto
D1 - CC - 6º SEMESTRE

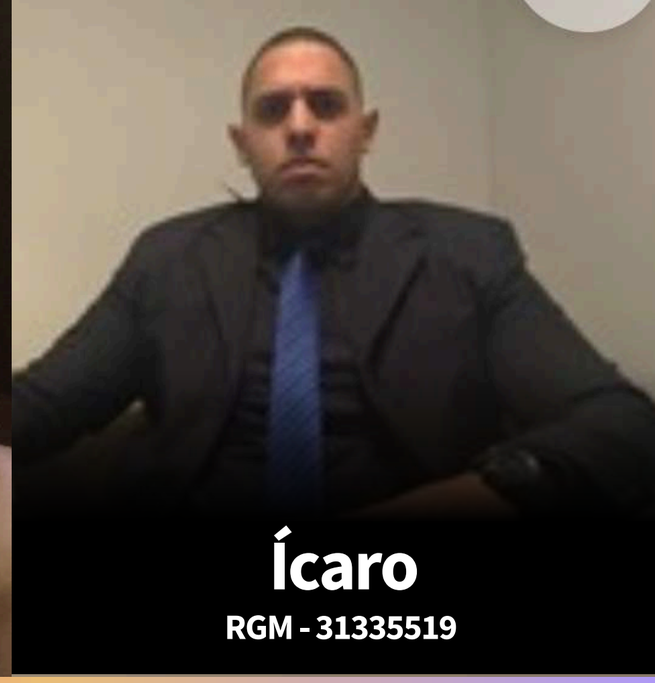
Haskell e Paradigma Funcional

Paradigmas de Linguagem de Programação
2024

Grupo Preto



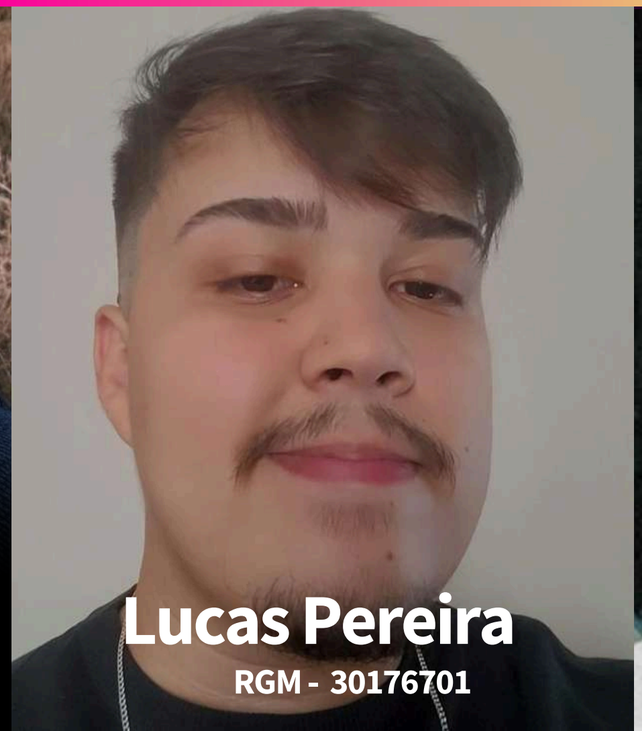
Everman
RGM - 30333717



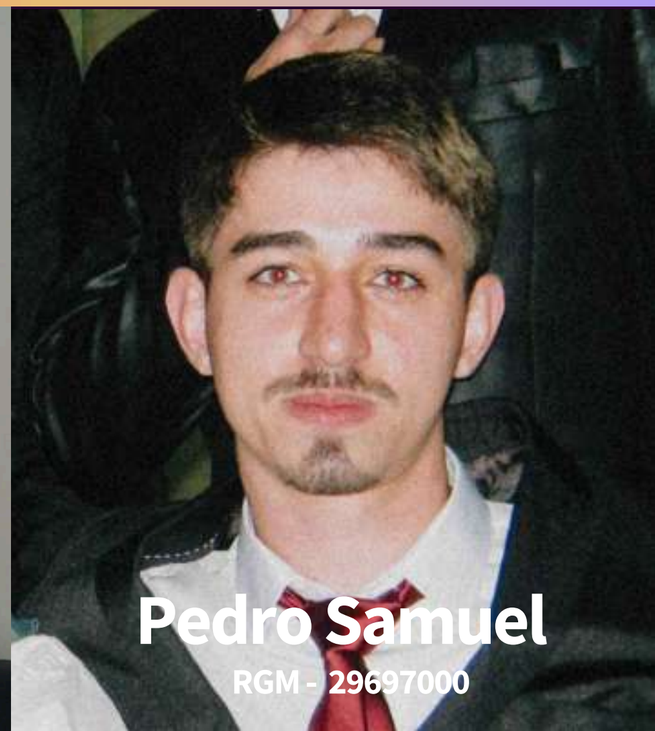
Ícaro
RGM - 31335519

José Henrique
RGM - 30525187

Julio Augusto
RGM - 30132401



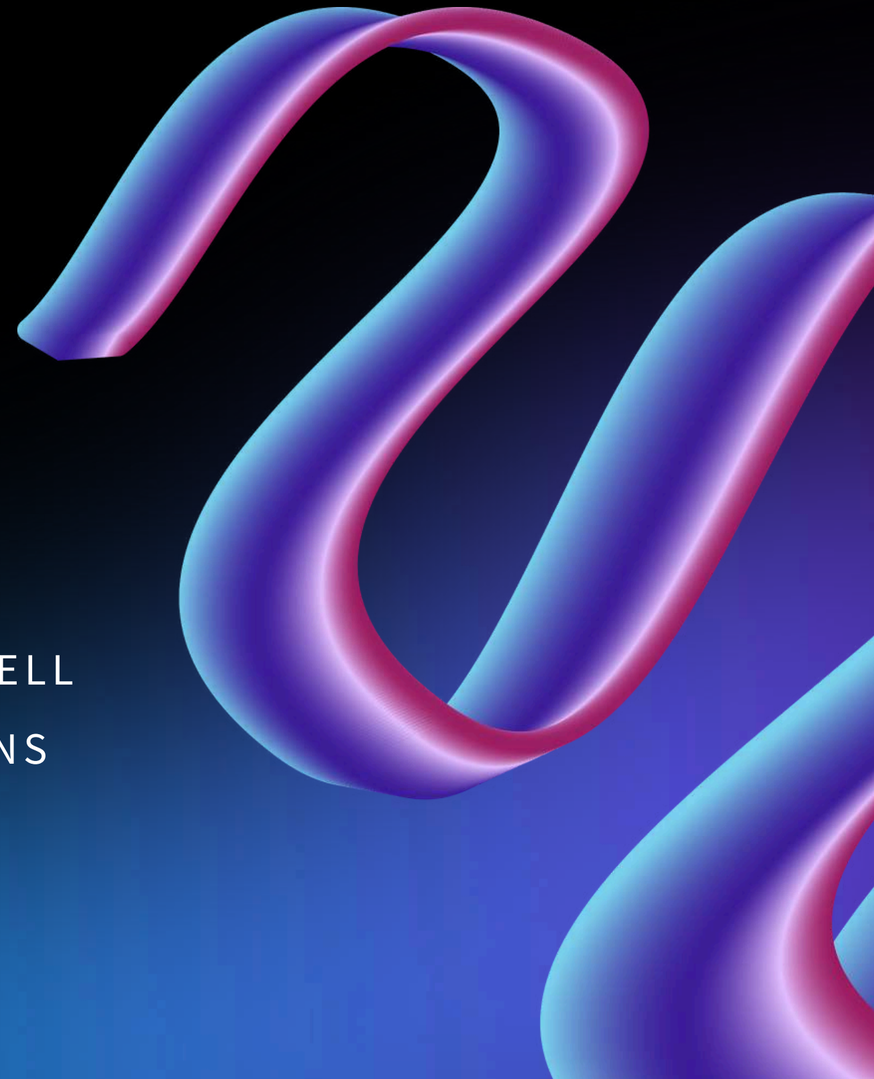
Lucas Pereira
RGM - 30176701



Pedro Samuel
RGM - 29697000

Grupo Preto
D1 - CC - 6º SEMESTRE

Índice.



Definições

PARADIGMA

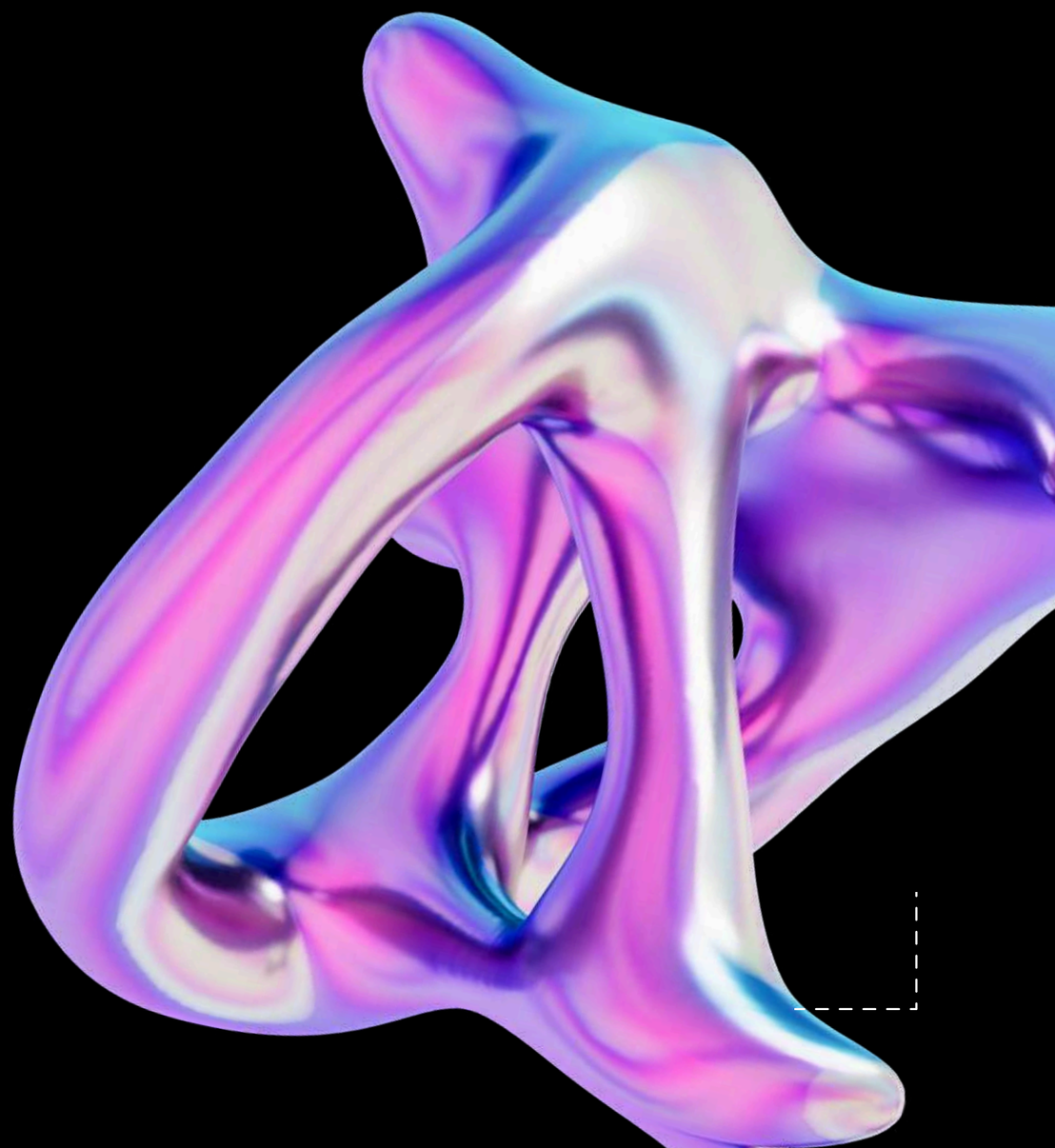
um paradigma nada mais é do que uma estrutura mental – composta por teorias, experiências, métodos e instrumentos – que serve para o pensamento organizar, de determinado modo, a realidade e os seus eventos.

Revista de Ciências Humanas - UFU

FUNÇÃO

Uma função, na programação, é um "bloco de código" que resolve um dado problema específico, que possui um nome associado e que pode ser chamado quando necessário. Ao final da função a linha de execução retorna para quem chamou a função.

IFSC



Paradigma Funcional.

Programação funcional é um paradigma de programação declarativo que trata a computação como uma avaliação de funções matemáticas e que evita estados ou dados mutáveis. Ele enfatiza a aplicação de funções, em contraste da programação imperativa, que enfatiza mudanças no estado do programa.

Paradigma Funcional.

O crescente interesse na programação funcional começou como uma resposta para lidar com o crescimento da concorrência em softwares multi-threaded.

O paradigma funcional permitiu ter paralelismo sem perder a integridade, garantindo imutabilidade.

Atualmente, o paradigma funcional provou-se como a melhor forma para lidar com a concorrência.

Cálculo Lambda

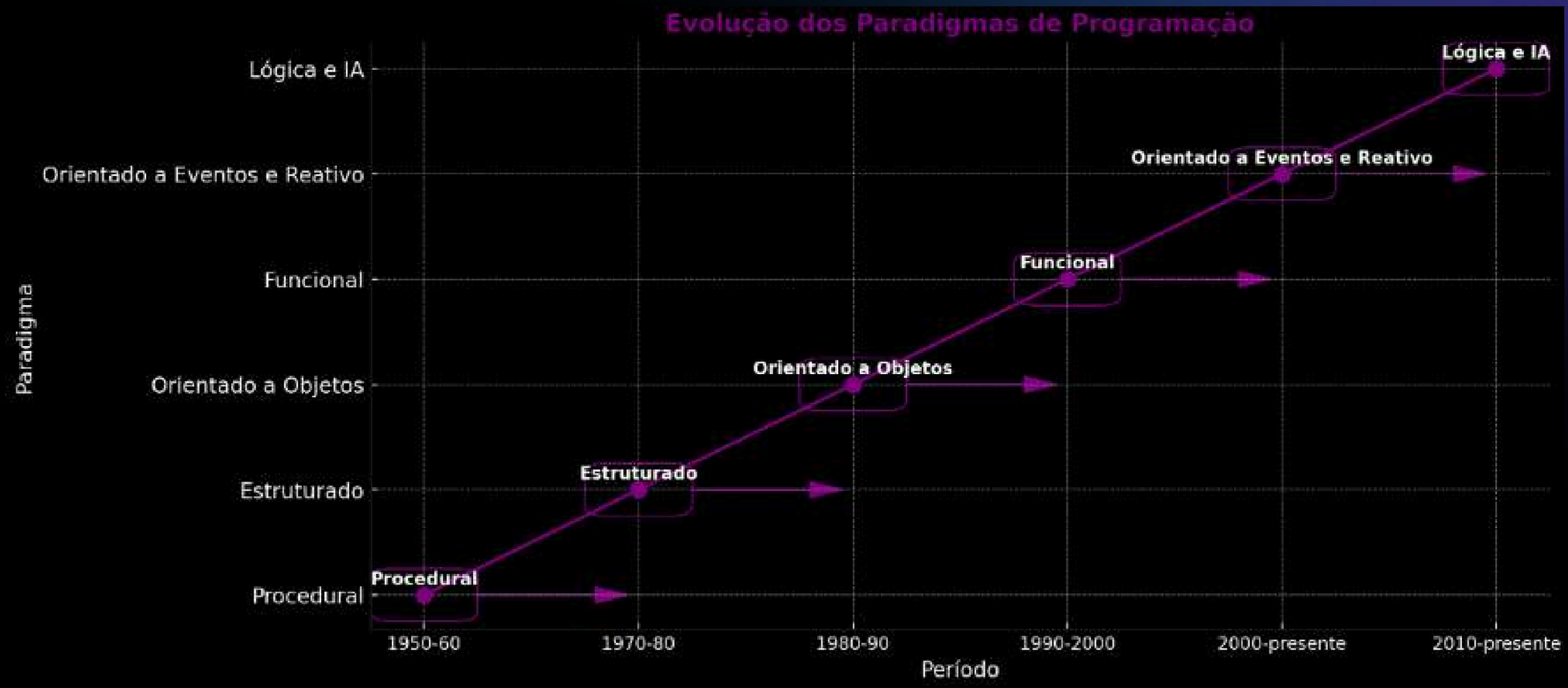
A base para o paradigma funcional baseia-se no Cálculo Lambda não tipado, desenvolvido por Alonzo Church em 1936.

Alonzo Church era um matemático na Universidade de Princeton, na Inglaterra. Church era o supervisor de doutorado de Alan Turing. Ele criou um modelo matemático de funções chamado cálculo lambda, um sistema formal que estuda funções recursivas computáveis. O cálculo lambda captura a essência da computação.

Alonzo Church

Universidade de Glasgow - Future Learn

Evolução dos Paradigmas



Anos 1990-2000: Paradigma Funcional

Contexto: Valoriza funções como blocos principais da programação.

Exemplos: Haskell, Lisp, Scheme.

Características: Funções de ordem superior, imutabilidade e transparência referencial. Foco na matemática e na eliminação de estados.

Características

- **Imutabilidade:** Uma vez que um dado é criado, ele não pode ser alterado. Isso evita muitos problemas comuns na programação, como efeitos colaterais indesejados.

- **Funções de Primeira Classe:** Em programação funcional, funções são tratadas como cidadãos de primeira classe, ou seja, podem ser passadas como argumentos, retornadas de outras funções e atribuídas a variáveis.

- **Composição de Funções:** Você pode criar novas funções a partir da combinação de funções existentes, facilitando a reutilização de código.

- **Recursão:** Em vez de loops tradicionais, a programação funcional utiliza a recursão para iterar sobre dados.

Vantagens

- **Código mais limpo e legível:** A programação funcional tende a produzir código mais conciso, previsível e fácil de ler, já que evita modificações de estado e minimiza efeitos colaterais.

- **Facilidade de testes e depuração:** Funções puras, que sempre retornam o mesmo resultado para os mesmos parâmetros, tornam mais fácil testar e rastrear bugs. A ausência de efeitos colaterais facilita isolar problemas.

- **Paralelismo mais simples:** Como não há modificações de estado compartilhado, o paralelismo é mais seguro e fácil de implementar, o que melhora o desempenho em sistemas distribuídos.

- **Reutilização de código:** Funções de alta ordem e composição permitem maior modularidade, facilitando o reaproveitamento de código entre diferentes partes do sistema.

Desvantagens

- **Uso de memória:** Como o paradigma funcional enfatiza a imutabilidade, ele tende a criar muitas cópias dos dados para evitar modificações.

- **Desempenho:** O uso excessivo de memória e a criação de muitas cópias do mesmo dado acabam consumindo um tempo a mais comparado a outros paradigmas.

- **Integração com código imperativo:** Já que muitas empresas possuem programas em bases de código legado, escritas em paradigmas imperativos, o paradigma funcional nem sempre é compatível com o estilo existente, dificultando a integração.

LINGUAGENS QUE SEGUEM ESSE PARADIGMA

Haskell

Erlang

Lisp

Scala

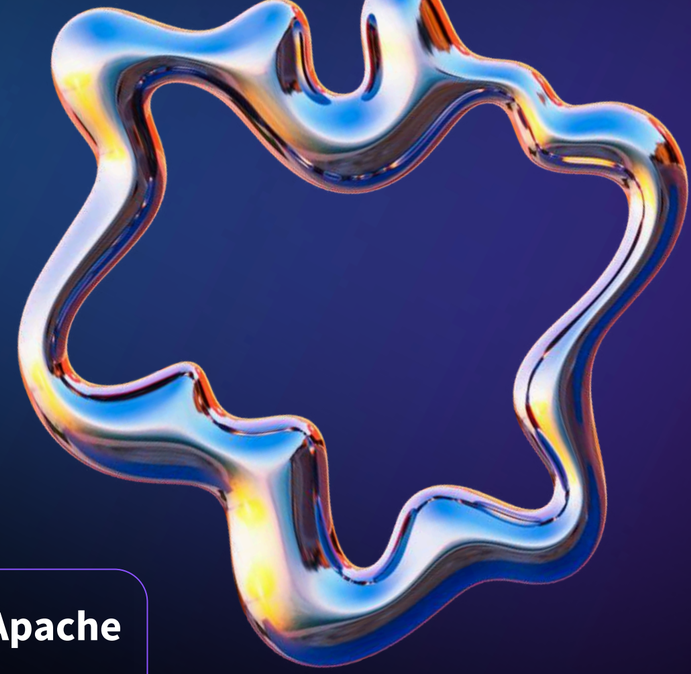
F#

Racket

Elixir

Standard ML

Casos de uso do Paradigma Funcional



- **Processamento e análise de dados :** Ferramentas de processamento de dados, como Apache Spark e Hadoop, usam linguagens funcionais (ou inspiradas em funcional) como Scala .

- **Cálculos Matemáticos e Científicos:** O uso de funções puras e imutabilidade é comum em cálculos matemáticos e científicos, garantindo reprodutibilidade e maior segurança contra efeitos colaterais indesejados.

- **Fintech:** Empresas com fins financeiros como bancos utilizam linguagens funcionais para realizar serviços como pagamentos e empréstimos.

Cronologia Haskell

Na década de 1930 tivemos a criação do Cálculo Lambda e avançamos rapidamente para o desenvolvimento inicial das linguagens de programação na década de 1950. Uma das primeiras linguagens de programação de alto nível foi LISP (List Processing). LISP adotou um estilo funcional, permitindo que funções de usuário fossem definidas e passadas como valores.

Já na década de 1980, diversos pesquisadores estavam inventando e estendendo várias linguagens de programação funcional. Diante das diversas pesquisas fragmentadas e sem código aberto, um grupo de acadêmicos formou um comitê para projetar e implementar uma nova linguagem que seria usada como veículo para pesquisa.

Relatório da Linguagem Haskell

Após vários anos de trabalho e discussões, o comitê publicou o primeiro **Relatório da Linguagem Haskell** em 1990. Este foi um marco importante: finalmente havia uma **linguagem funcional comum** em torno da qual a **comunidade de pesquisa** poderia se unir.

Neste relatório foi apresentado informações relativas à sua história, mostrado a sintaxe básica da linguagem, exemplos de utilizações, e um comparativo de vantagens e desvantagens do mesmo.

Haskell

Haskell é uma linguagem de programação puramente funcional, nomeada em homenagem ao lógico Haskell Curry, conhecido por seus trabalhos na lógica combinatória e no projeto ENIAC.

Haskell é a linguagem funcional sobre a qual mais se realizam pesquisas atualmente. Muito utilizada no meio acadêmico, é uma linguagem relativamente nova, derivada de outras linguagens funcionais, como por exemplo Miranda e ML.

Possui foco no alcance de soluções para problemas matemáticos, clareza, e de fácil manutenção nos códigos, e possui uma variedade de aplicações e apesar de simples é muito poderosa.

Logo

Haskell

O logotipo do Haskell tem um significado interessante e bem alinhado à sua filosofia de linguagem. O símbolo é formado por duas partes principais: um **“lambda” (λ)** e o **sinal de igual (=)**.

O lambda é um símbolo historicamente associado ao cálculo lambda. Esse conceito é essencial na programação funcional, que é o paradigma no qual Haskell se baseia. Haskell, sendo uma linguagem de programação puramente funcional, usa o símbolo para refletir a ideia de que cada expressão é uma função e não há efeitos colaterais, como acontece em paradigmas imperativos.

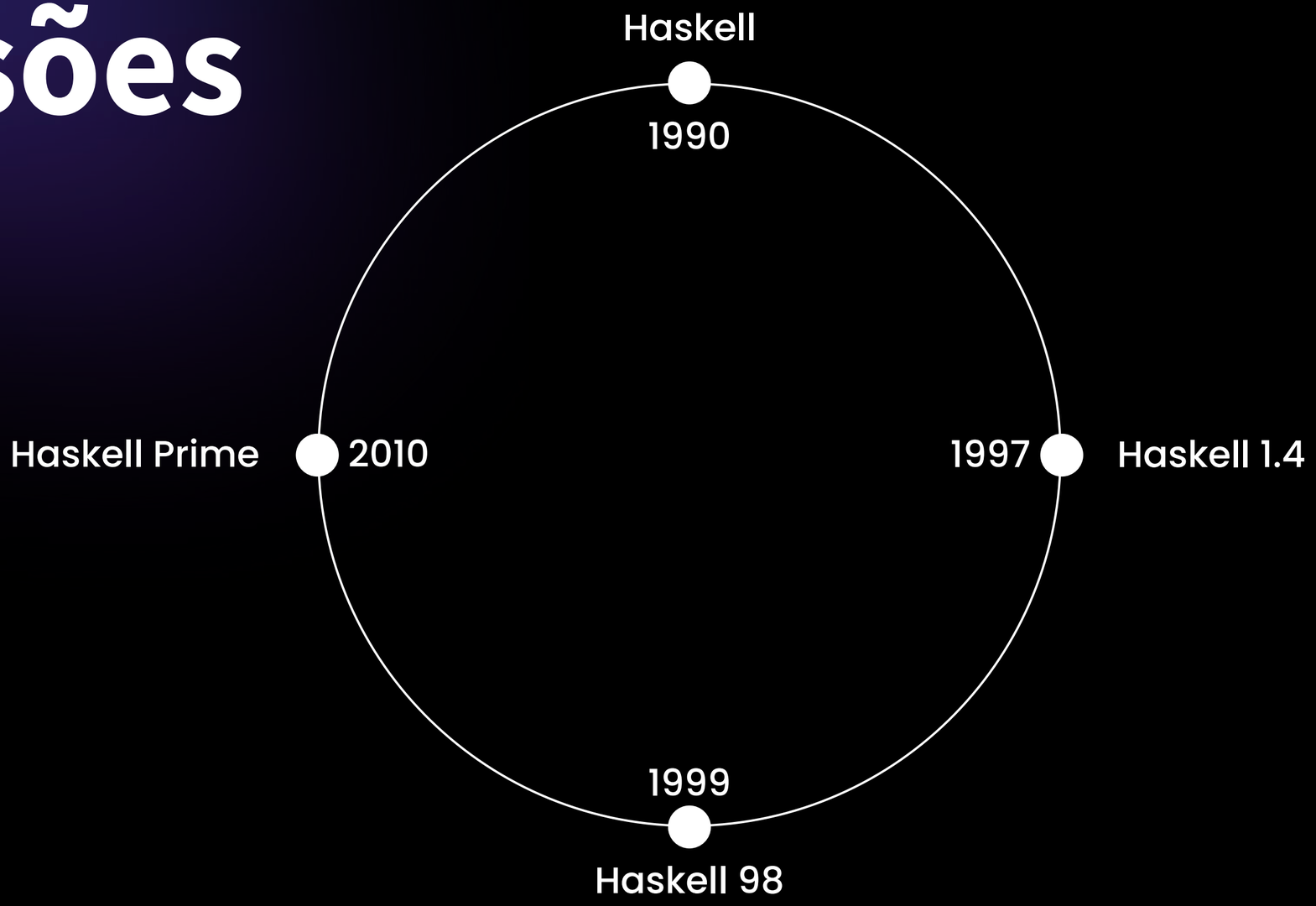
Já o símbolo de igual (=) na logo do Haskell representa o conceito de definição e atribuição imutável na linguagem, onde os valores não mudam de estado. Essa combinação de símbolos na logo do Haskell representa a essência da linguagem: uma **base matemática forte** (lambda) e um foco na **pureza e na imutabilidade** (igualdade).

Quem criou Haskell?

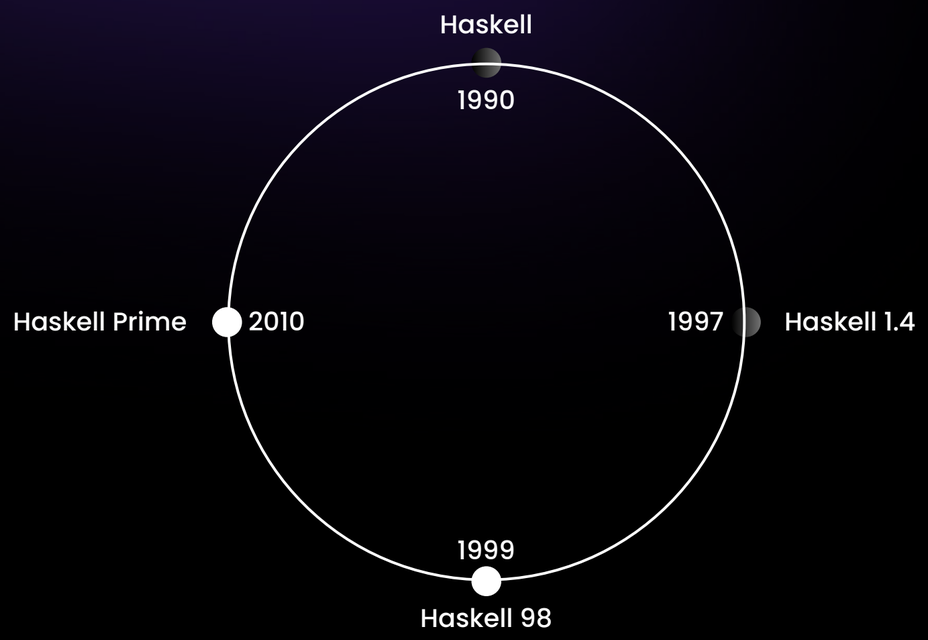
Em setembro de 1987 foi realizada a conferência Linguagens de Programação Funcional e Arquitetura de Computadores, no Oregon, onde foi criado o comitê citado anteriormente.
Esse comitê era composto por pesquisadores e cientistas da computação, incluindo nomes importantes para a criação de Haskell como Simon Peyton Jones, Paul Hudak, Philip Wadler, e John Hughes.

Simon Peyton Jones | Paul Hudak | Philip Wadler | John Hughes

Versões


Haskell
1990
Haskell 1.4
1997
Haskell 98
1999
Haskell Prime
2010

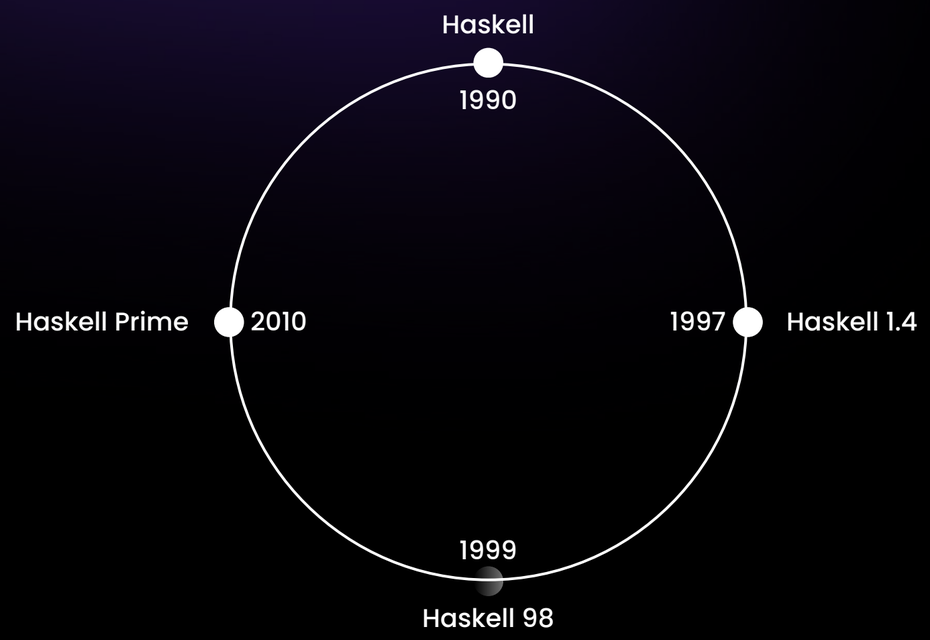
Versões



A primeira versão do Haskell (“Haskell 1.0”) foi definida em 1990, na criação e publicação da linguagem.

Os esforços do comitê resultaram em uma série de definições da linguagem (1.0, 1.1, 1.2, 1.3 e 1.4) que perdurou até 1997.

Versões



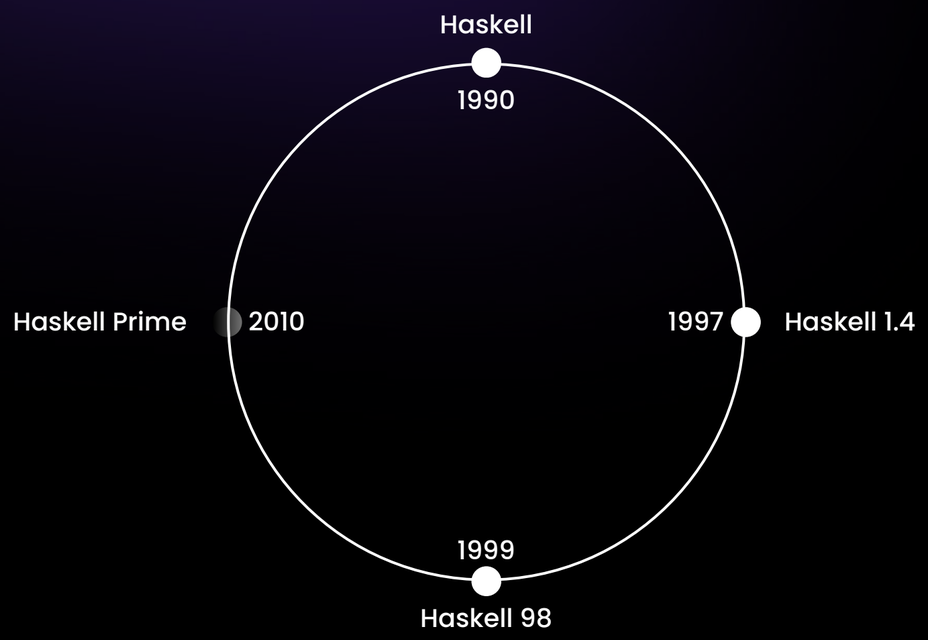
No final de 1997, as séries culminaram no Haskell 98, com a intenção de especificar uma versão **mais estável** e **portátil** da linguagem acompanhada de uma biblioteca padrão para ensino, e como base para extensões futuras.

O comitê expressou satisfação em criar extensões e variantes de Haskell 98 a partir da adição e incorporação de atributos experimentais.

Em fevereiro de 1999, o padrão da linguagem foi originalmente publicado em um relatório chamado de **The Haskell 98 Report**. Em janeiro de 2003, uma versão revisada foi publicada como **Haskell 98 Language and Libraries: The Revised Report**.

A linguagem continuou a evoluir rapidamente, com a implementação do **Compilador Glascow Haskell** representando o padrão atual.

Versões

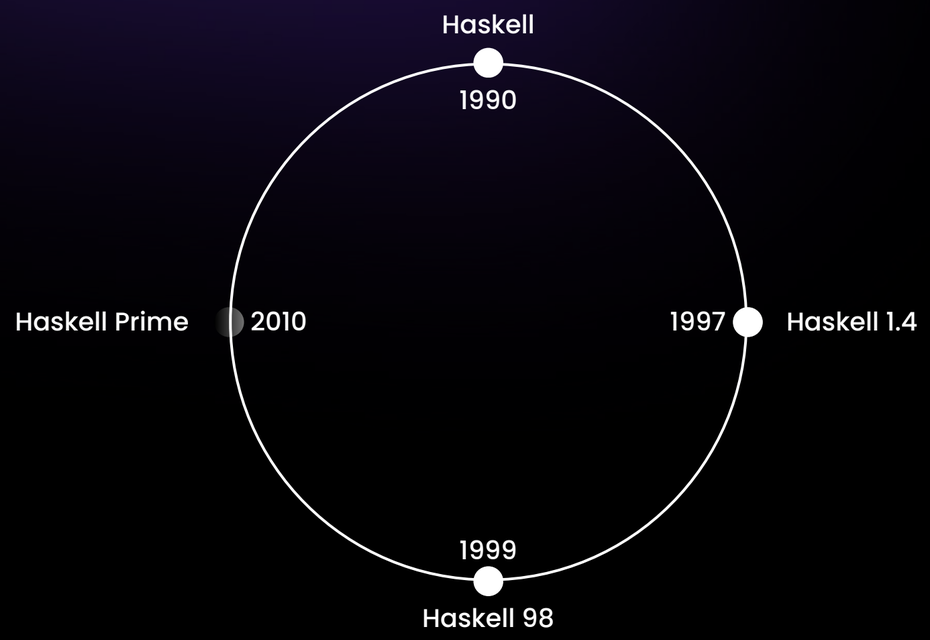


No começo de 2006, iniciou-se o processo de definir um sucessor para o padrão Haskell 98, informalmente chamado de Haskell Prime.

A intenção era de que viesse a ser um processo incremental para revisar a definição da linguagem, produzindo uma revisão nova anualmente.

A primeira revisão, chamada de Haskell 2010, foi anunciada em novembro de 2009, e publicado em julho de 2010.
Haskell 2010 é uma atualização incremental da linguagem, incorporando atributos muito utilizados e não controversos previamente ativados por bandeiras específicas do compilador.

Versões



Algumas das implementações do Haskell 2010 foram:

Módulos hierárquicos: Módulos que consistem de sequências de identificadores em maiúsculo separadas por pontos, ao invés de apenas um identificador. Dessa maneira é possível nomear os módulos de maneira hierárquica (Data.List ao invés de List).

A interface de função estrangeira (FFI), que permitiu ligações para outras linguagens de programação. Apenas ligações para C são especificadas no relatório, mas o design permite para outras ligações de linguagens.

As regras de inferência de tipo foram afrouxadas para permitir que mais programas chequem o tipo.

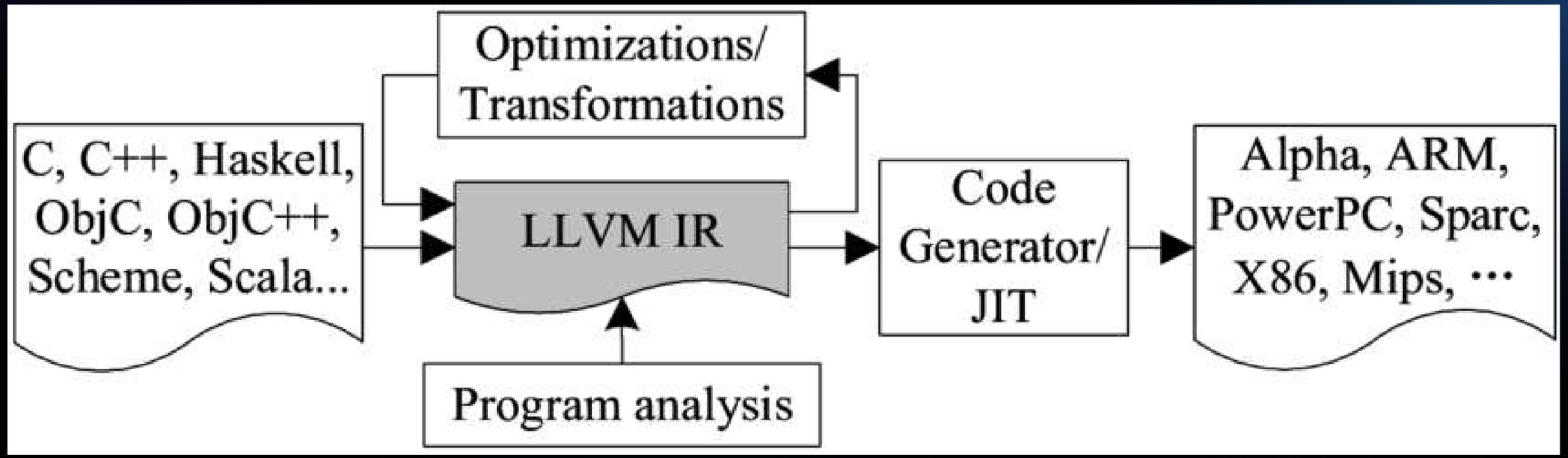
Compilador GHC

GHC (Glasgow Haskell Compiler) - Compilador para Haskell

- Origem e Evolução:
  - Criado na Universidade de Glasgow, em 1989, inicialmente em Lazy ML.
  - Reescrito em Haskell, com exceção do analisador sintático, sob a liderança de Simon Peyton Jones.
  - Lançado em versão beta em 1991, vem sendo aprimorado por uma comunidade global, com mais de 60 desenvolvedores contribuindo.
  - Simon Peyton Jones e Simon Marlow continuaram seu desenvolvimento na Microsoft Research.

- Etapas de Compilação:
  a. Análise Léxica e Sintática: Divide o código em tokens e verifica a sintaxe.
  b. Análise Semântica: Confere a validade do código em termos de tipos e variáveis.
  c. Otimização: Melhora o desempenho do código intermediário com várias técnicas.
  d. Geração de Código: Traduz o código otimizado para linguagem de máquina ou LLVM.

Processo de Compilação



Porquê o GHC?

- **Código Aberto:** Software livre que permite contribuições da comunidade.
- **Portabilidade:** Compatível com várias plataformas, como Windows, Linux e macOS.
- **Suporte Completo a Haskell:** Suporta o padrão Haskell 2010 e extensões, facilitando o desenvolvimento.
- **Desempenho:** Gera código eficiente, ideal para programas concorrentes.
- **Comunidade Ativa:** Grande rede de usuários e desenvolvedores para suporte e troca de conhecimento.

Características de Haskell

- **Funções puras:** Elas sempre produzem o mesmo resultado para os mesmos argumentos e não têm efeitos colaterais.

- **Imutabilidade:** As variáveis em Haskell são imutáveis por padrão, o que facilita a depuração e a compreensão do fluxo de dados.

- **Verificação de tipos estática e inferência de tipos:** Erros de tipagem são capturados em tempo de compilação, aumentando a segurança. Além disso, o sistema de inferência de tipos permite que o compilador deduza automaticamente o tipo de variáveis e funções, simplificando o código.

- **Avaliação preguiçosa:** Permite a criação de estruturas de dados infinitas e o uso eficiente de memória.

Vantagens

- **Segurança:** O sistema de tipagem estática ajuda a detectar erros em tempo de compilação.

- **Expressividade:** Haskell permite que os desenvolvedores escrevam código de maneira concisa e clara.

- **Manutenção:** O código tende a ser mais fácil de manter, pois as funções puras e a imutabilidade ajudam a prevenir erros complexos.

Desvantagens

- **Curva de aprendizado íngreme:** Para desenvolvedores que não estão familiarizados com o paradigma funcional, aprender Haskell pode ser desafiador.

- **Pouca adoção em produção:** Apesar de seu uso acadêmico e em nichos industriais, Haskell não é amplamente utilizado na indústria como um todo.

- **Problemas que envolvam muitas variáveis** (ex. banco de dados) ou muitas atividades sequenciais.

- **Implementações ineficientes:** considerando que as funções têm um caráter recursivo em suas definições, elas tendem a ser mais ineficientes computacionalmente que as linguagens imperativas, pois tem maior chance de ocorrer erros e possuem um grande uso de memória.

Casos de Uso

- **Desenvolvimento web:** Haskell tem sido aplicada no desenvolvimento de aplicações web, fornecendo uma alternativa robusta e eficiente para outras linguagens mais tradicionais.

- **Processamento de dados:** A funcionalidade pura de Haskell a torna especialmente adequada para casos de uso que envolvem processamento de grandes volumes de dados.

- **Criptografia e segurança:** Haskell oferece um alto nível de segurança e confiabilidade, o que a torna uma escolha popular para aplicações críticas de segurança, especialmente na área de criptografia.

- **Sistemas Web3 (Blockchain) :** Haskell é usado em ambientes muito sensíveis, onde o resultado de uma função tem que ser demonstrável e imutável antes de ser executado. Por essa razão, Haskell é uma opção muito segura para sistemas blockchain e Contratos inteligentes (Smart Contracts).

- **Setor Bancário :** Pelo fator de segurança e confiabilidade que Haskell provê, empresas no setor bancário vem adotando Haskell em automatizações de processos, já que é uma linguagem que garante segurança.

Uso na Indústria

- **Desenvolvimento web:** Projetos como o Snap Framework e o Yesod Framework têm feito uso de Haskell para construir aplicações web escaláveis e de alto desempenho.

- **Processamento de dados:** Empresas como a Facebook e a Standard Chartered têm utilizado Haskell para implementar soluções de processamento de dados eficientes e confiáveis.

- **Criptografia e segurança:** Projetos como o Cryptol e o Eta-Crypto têm demonstrado como Haskell pode ser aplicada com sucesso no desenvolvimento de sistemas seguros.

- **Sistemas Web3 :** Cardano que é uma criptomoeda, e seu sistema de construção de dAPPs (Atlas)

Outros Exemplos

- **AT&T -** Divisão de Segurança de Rede para automatizar o processamento de Abuso e uso ilegal na rede AT&T
- **Intel -** Compilador Haskell para sua pesquisa sobre paralelismo multicore em escala.
- **Bank of America Merril Lynch -** Transformação e carregamento de dados de backend
- **Sistemas web3 (Cardano) -** Cardano que é uma criptomoeda, e seu sistema de construção de dAPPs (Atlas)
  Dish Network (Contrato)
- **Microsoft -** Usa Haskell para seu sistema de serialização de produção, Bond. Bond é usado na Microsoft em serviços de alta escala.

Sigma

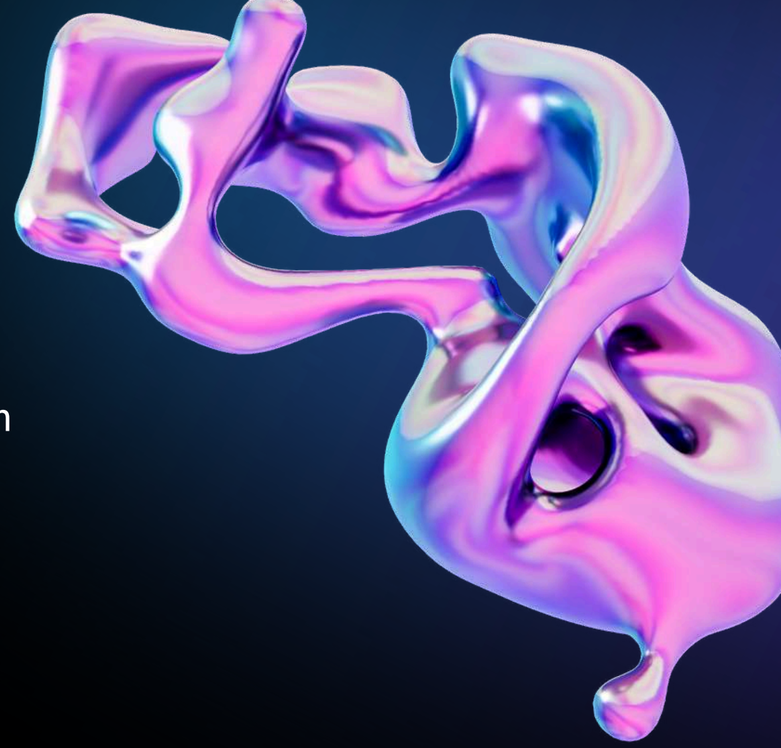
ANTISPAM - FACEBOOK

Uma das armas na luta contra spam, malware e outros abusos no Facebook é um sistema chamado **Sigma**. O trabalho dele é identificar proativamente ações maliciosas no Facebook, como spam, ataques de phishing, postagem de links para malware, etc.

O Facebook concluiu o redesenho do Sigma, que envolveu a substituição da linguagem FXL interna, usada anteriormente para programar o Sigma, por Haskell . O Sigma com tecnologia Haskell agora roda em produção, atendendo a mais de um milhão de solicitações por segundo.

Além disso, o Facebook fez várias melhorias no GHC (o compilador Haskell) e as alimentaram de volta ao upstream, e foram capazes de obter melhor desempenho do Haskell em comparação com a implementação anterior.

Por quê Haskell?



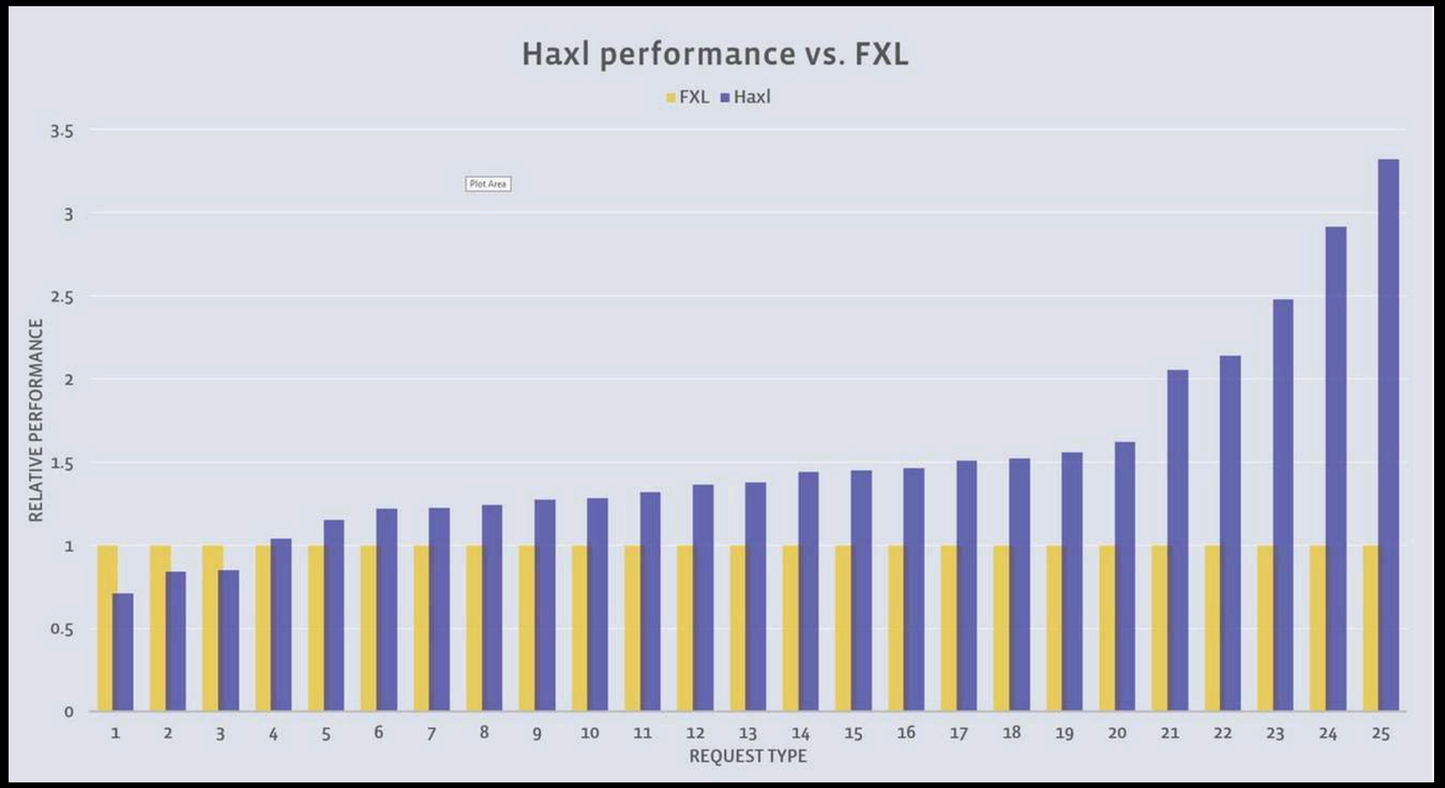
O Facebook definiu os seguintes recursos que a linguagem deveria ofertar quando estavam escolhendo uma substituição:

- **Ser puramente funcional e fortemente tipada**
- **Possibilitar buscas de dados em lote e sobreposição automáticas**
- **Enviar alterações de código para produção em minutos**
- **Desempenho**
- **Suporte para desenvolvimento interativo**

Segundo o próprio Facebook o motivo da escolha de Haskell foi que Haskell se sai muito bem nos recursos definidos: é uma linguagem **puramente funcional** e **fortemente tipada**, e tem um **compilador otimizador maduro** e um **ambiente interativo** (GHCi). Também tem todos os recursos de abstração que eles precisavam, com um rico conjunto de bibliotecas disponíveis e é apoiado por uma comunidade de desenvolvedores ativa.

Desempenho

O Haskell tem desempenho até **três vezes mais rápido** que o FXL para determinadas solicitações. Em uma combinação típica de carga de trabalho, mediu-se uma melhoria de **20% a 30%** no rendimento geral, o que significa que pode ser atendendido de 20% a 30% a mais de tráfego com o mesmo hardware



Sintaxe e Semântica

- Exemplos da sintaxe Haskell
- Exemplos da semântica Haskell
- Comparação de código com outra linguagem funcional
- Código de manipulação de listas

Sintaxe

GRAMÁTICA

Na gramática de uma língua natural, como o português, a sintaxe define como palavras e frases devem ser combinadas para criar sentenças que fazem sentido e seguem normas da língua.

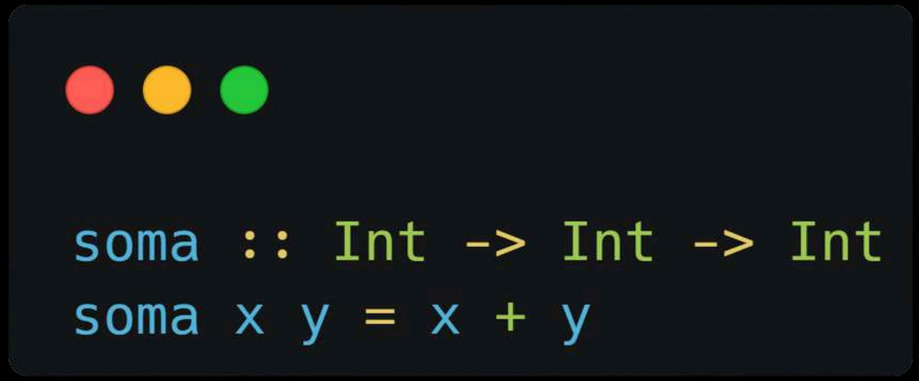
LP

Em linguagens de programação, a sintaxe é o conjunto de regras que define a forma correta de escrever instruções para que possam ser interpretadas ou compiladas pelo computador.

OU SEJA, AMBAS DEFINEM REGRAS QUE ESTRUTURAM A FORMAÇÃO DE SENTENÇAS VÁLIDAS EM SEUS RESPECTIVOS CONTEXTOS.

EXEMPLO

Função simples para somar dois números:



Semântica

A semântica trata do significado de palavras e frases, tanto em uma língua natural quanto em linguagens de programação. Enquanto a sintaxe se preocupa com a forma e a estrutura correta, a semântica é sobre o sentido e a interpretação.

A semântica na linguagem Haskell refere-se ao significado dos programas e como eles são executados, além da forma como os dados são manipulados e transformados. Haskell é uma linguagem funcional pura, e sua semântica é baseada em conceitos matemáticos como funções, tipos e equações.

EXEMPLO

Função simples para somar dois números:

```haskell
-- Definindo uma função que soma dois números
soma :: Int -> Int -> Int
soma x y = x + y
-- Aplicando a função
resultado = soma 3 5 -- resultado é 8
```

Comparação de código

A programação funcional permite o uso de operações de ordem superior, que podem receber outras funções como argumentos ou retornar as funções como resultados. Isso torna o código mais modular e reutilizável.

EXEMPLO
Números de 1 a 10 em Haskell

```haskell
-- main.hs
main :: IO()
main = mapM_ print [1..10]
```

EXEMPLO
Números de 1 a 10 em Lisp

```lisp
;; main.lisp
(defun print-numbers ()
  (loop for i from 1 to 10 do
  (print i)))

(print-numbers)
```

Manipulação de listas

As listas são um conjunto de sequências finitas de elementos de um dado tipo A, ou seja, diferentemente das tuplas as listas em Haskell são homogêneas.

EXEMPLO
criação de uma lista

```haskell
main :: IO ()
main = do
    let lostNumbers = [4, 8, 15, 16, 23, 48]
    print lostNumbers
```

Um aspecto importante das listas em Haskell é que elas podem ser definidas por compreensão, de maneira muito semelhante ao que estamos habituados com a definição de conjuntos na matemática.

EXEMPLO
obter o comprimento da lista

```haskell
main :: IO ()
main = do
    let lostNumbers = [4, 8, 15, 16, 23, 48]
    let comprimento = length lostNumbers
    print comprimento
```

List Comprehension

É uma maneira concisa de construir listas. A construção é similar à notação de conjuntos em matemática.

```haskell
-- List comprehension
squares :: [Int]
squares = [x * x | x <- [1..10]]

main :: IO ()
main = putStrLn $ "squares: " ++ show squares
```

Expressão case

O case permite a correspondência de padrões e é útil para trabalhar com dados de maneira condicional.

```haskell
-- Expressão case para condicionais
describeNumber :: Int -> String
describeNumber x = case x of
    0 -> "Zero"
    1 -> "One"
    _ -> "Another number"

main :: IO ()
main = do
    putStrLn $ "describeNumber 0: " ++ describeNumber 0
    putStrLn $ "describeNumber 2: " ++ describeNumber 2
```

Expressão let

Expressão let pode ser utilizada para definições temporárias. Ela permite definir variáveis locais em uma expressão.

```haskell
-- Expressão let para definições temporárias
calculateArea :: Float -> Float
calculateArea radius =
    let piValue = 3.14159
    in piValue * radius * radius

main :: IO ()
main = putStrLn $ "calculateArea 5: " ++ show (calculateArea 5)
```

Muito Obrigado!