Revista Brasileira de Estudos de Gestão e Desenvolvimento Regional
v. 08, n. 03, p. 1-11, 2026
Seção de Artigos

PROGRAMAÇÃO FUNCIONAL COM HASKELL E LISP

FUNCTIONAL PROGRAMMING WITH HASKELL AND LISP

PROGRAMACIÓN FUNCIONAL CON HASKELL Y LISP

Letícia Souza dos Santos
Universidade do Estado de Mato Grosso - UNEMAT
e-mail: leticiasouzasnp@gmail.com

Davi Alves Mares
Universidade do Estado de Mato Grosso - UNEMAT
e-mail: davialvesmares@gmail.com

Dra. Janecler Foppa
https://orcid.org/0000-0002-8906-4195
Universidade do Estado de Mato Grosso - UNEMAT
e-mail: janecler.foppa.snp@gmail.com

Yasmin Conceição de Lima Carvalho
Universidade do Estado de Mato Grosso - UNEMAT
e-mail: yasmin.carvalho@unemat.br

Submissão em: 08/11/2025 Aceito em: 19/01/2026

RESUMO
Este artigo analisa fundamentos conceituais e filosóficos das linguagens funcionais, com ênfase nas linguagens Lisp e Haskell, a partir do cálculo λ. Investiga elementos que as definem, como pureza funcional, tipagem, transparência referencial, homoiconicidade e estratégias de avaliação. Lisp destaca-se pela flexibilidade e metaprogramação, enquanto Haskell prioriza rigor formal e segurança por meio do sistema de tipos. A problemática central investiga os fundamentos de Haskell e Lisp e suas diferenças no contexto da programação funcional. O objetivo geral é explorar esses fundamentos e como objetivos específicos, compreender o sistema de tipagem e os conceitos que asseguram expressividade e rigor matemático, com abordagem qualitativa e método bibliográfico. Conclui-se que ambas fortalecem o paradigma funcional e contribuem para soluções robustas e expressivas.
Palavras-chave: Programação Funcional, Linguagens de Programação, Paradigmas Computacionais

ABSTRACT
This article analyzes the conceptual and philosophical foundations of functional languages, with an emphasis on Lisp and Haskell, based on λ calculus. It investigates elements that define them, such as functional purity, typing, referential transparency, homoiconicity, and evaluation strategies. Lisp stands out for its flexibility and metaprogramming, while Haskell prioritizes formal rigor and security through its type system. The central problem investigates the foundations of Haskell and Lisp and their differences in the context of functional programming. The general objective is to explore these foundations and, as specific objectives, to understand the typing system and the concepts that ensure expressiveness and mathematical rigor, using a

qualitative approach and bibliographic method. It concludes that both strengthen the functional paradigm and contribute to robust and expressive solutions.
Keywords: Functional Programming, Programming Languages, Computational Paradigms

RESUMEN
Este artículo analiza los fundamentos conceptuales y filosóficos de los lenguajes funcionales, con énfasis en Lisp y Haskell, basados en el cálculo lambda. Investiga los elementos que los definen, como la pureza funcional, el tipado, la transparencia referencial, la homoiconicidad y las estrategias de evaluación. Lisp destaca por su flexibilidad y metaprogramación, mientras que Haskell prioriza el rigor formal y la seguridad mediante su sistema de tipos. El problema central indaga los fundamentos de Haskell y Lisp y sus diferencias en el contexto de la programación funcional. El objetivo general es explorar estos fundamentos y, como objetivos específicos, comprender el sistema de tipado y los conceptos que garantizan la expresividad y el rigor matemático, utilizando un enfoque cualitativo y un método bibliográfico. Concluye que ambos fortalecen el paradigma funcional y contribuyen a soluciones robustas y expresivas.
Palabras clave: Programación Funcional, Lenguajes de Programación, Paradigmas Computacionales

1 INTRODUÇÃO

A programação funcional constitui uma corrente de pensamento na ciência da computação que organiza o ato de programar em torno da definição e transformação de funções matemáticas. Sua genealogia remonta ao trabalho de Alonzo Church sobre o cálculo λ, cuja formulação formalizou uma visão da computação baseada em aplicação e substituição de funções, uma alternativa ao modelo imperativo de máquinas de estados e instruções sequenciais. Essa base teórica não só forneceu uma linguagem formal para raciocínios sobre computação, mas também inspirou linguagens de programação cuja sintaxe e semântica privilegiavam a expressão de transformações em detrimento de mutações de estado (Alonzo Church, 1936).

Dois marcos históricos e intelectuais desempenham papel central na narrativa das linguagens funcionais: Lisp, concebido por John McCarthy no final da década de 1950, e Haskell, fruto de um esforço colaborativo acadêmico consolidado nos anos 1990. Lisp materializou cedo a ideia de tratar código como dados, a chamada homoiconicidade, e dotou os programadores de mecanismos de metaprogramação (macros) que ampliaram radicalmente a capacidade de abstração e extensão da linguagem. Por seu turno, Haskell nasceu com a ambição de servir como um laboratório de pesquisa e prática para ideias de pureza funcional, tipagem estática forte e avaliação não-estrita, procurando formalizar e tornar aplicáveis conceitos provenientes da teoria dos domínios e da semântica denotacional (Hudak *et al*., 1990).

A distinção entre Lisp e Haskell não é apenas histórica, mas filosófica. Lisp oferece uma plataforma de experimentação simbólica, flexível e pragmática. Haskell procura rigor matemático, segurança estática e modelos explícitos para efeitos computacionais. Apesar dessas diferenças, ambas as famílias de linguagens compartilham preocupações centrais, transparência referencial, modularidade através de funções, e uma tendência à programação declarativa, que as tornam relevantes

tanto para investigação teórica quanto para aplicações práticas contemporâneas (Hudak *et al*.,1990). Destaca-se como problemática deste artigo quais são os fundamentos das linguagens Haskell e Lisp, comparando-as na programação funcional.

O objetivo geral é explorar os fundamentos das linguagens Haskell e Lisp e como objetivos específicos destacar o sistema de tipagem formal e os conceitos que garantem sua expressividade e rigor matemático.

2 FUNDAMENTOS CONCEITUAIS

Nesta seção sintetizou-se, em termos conceituais e filosóficos, os alicerces que orientam o desenho e a adoção de linguagens funcionais. A intenção não é definir formalmente cada mecanismo, mas explicitar as motivações intelectuais e as consequências de projeto que emergem dessas motivações.

2.1 Cálculo λ como paradigma de referência

O cálculo λ introduz a noção de computação como manipulação de símbolos por meio de abstrações e aplicações funcionais. Essa visão transforma programas em objetos matemáticos sobre os quais se podem provar propriedades, raciocinar sobre equivalências e aplicar transformações formais. A adoção do cálculo λ como referência filosófica legitima práticas como a substituição semântica (transparência referencial) e inspira modelos formais de semântica operacional e denotacional que apoiam verificação e otimização (Alonzo Church, 1936).

A escolha por linguagens puramente funcionais (ou por estilos que privilegiam funções puras) é movida pela busca de previsibilidade e facilidade de raciocínio. Uma função pura, em termos conceituais, é uma transformação matematicamente determinística entre entradas e saídas, sem efeitos colaterais observáveis no ambiente. Esse ideal permite tratar programas como equações, aplicar substituições locais e construir provas locais de correção. Linguagens ou estilos que adotam essa filosofia também precisam propor mecanismos para integrar efeitos (E/S, estado, exceções) sem sacrificar a razão formal, daí a emergência de padrões de modelagem de efeitos (por exemplo, monads na literatura teórica) (Hudak *et al*.,1990).

2.2 Homoiconicidade e metaprogramação (o caso Lisp)

Uma das contribuições intelectuais mais notáveis de Lisp é a sua capacidade de representar código como estruturas de dados nativas (listas), o que facilita a escrita de programas que geram, analisam e transformam outros programas, isto é, metaprogramação. Filosoficamente, isso abre um espaço para experimentação e extensão do próprio ambiente de linguagem: DSLs internas, macros que elevam a abstração e ferramentas interativas de desenvolvimento (REPLs) tornam-se práticas naturais. A homoiconicidade altera a relação entre linguagem e programador, deslocando parte do poder técnico do compilador/interpreter para o próprio código do usuário (John McCarthy, 1958).

2.3 Tipagem, segurança e o projeto de linguagens (o caso Haskell)

Haskell representa uma visão em que a tipagem estática é usada como ferramenta para capturar invariantes e prevenir classes de erros em tempo de compilação. Em vez de confiar exclusivamente em testes dinâmicos, a tipagem serve como documentação executável e como restrição que guia o desenho de APIs e abstrações. Do ponto de vista filosófico, isso reflete uma confiança em métodos formais para garantir correção parcial do *software*, reduzindo a carga cognitiva do programador ao trabalhar com grandes sistemas (Hudak *et al*., 1990).

2.4 Estratégia de avaliação: implicações filosóficas e pragmáticas

A escolha da estratégia de avaliação (avaliada estrita vs. não-estrita/lazy) é uma decisão de projeto com profundas implicações conceituais. A avaliação não-estrita, ao adiar a computação até o momento de necessidade, favorece a definição de componentes de forma modular e permite raciocinar sobre produtores e consumidores de dados sem impor uma ordem de execução rígida. Filosoficamente, isso aproxima a programação de uma modelagem declarativa de fluxo de dados.

Pragmaticamente, contudo, impõe desafios de gestão de recursos (ex.: representações pendentes de computação) e exige modelos de análise de espaço/tempo distintos daqueles de linguagens eager (Hudak *et al*., 1990).

A ênfase na composição, construir programas a partir da combinação de funções pequenas e bem definidas, funciona como princípio organizador para o projeto de *software* funcional. Em termos filosóficos, composição é análoga à composição de funções matemáticas: permite provar propriedades locais e estender sistemas sem alterar os componentes originais. A modularidade funcional facilita reutilização e construção incremental de complexidade, reduzindo o acoplamento e aumentando a previsibilidade do comportamento do sistema (Hudak *et al*., 1990).

3 LINGUAGEM HASKELL

Haskell é uma linguagem de programação puramente funcional desenvolvida academicamente, na qual todas as construções são tratadas como funções, permitindo composições matemáticas elegantes e previsíveis. Essa característica de pureza funcional garante o princípio da Transparência Referencial (a substituição de uma expressão por seu valor sem alterar o resultado do programa), fundamental para a prova de correção de programas, tornando-a uma referência importante no paradigma funcional (Hudak *et al*., 1990). A linguagem distingue estruturas homogêneas (como listas) de heterogêneas (como tuplas), tratando a String como uma forma sintática abreviada (*syntactic sugar*) para listas de caracteres (Hudak; Fasel, 1992).

Haskell possui tipagem forte e estática, o que garante a segurança de tipos (que o programador não misture tipos de forma inadequada, como somar um caractere com um inteiro) e assegura que todos os erros de tipo sejam detectados em tempo de compilação (Hudak; Fasel, 1992; Silva, 2017).

O recurso de inferência de tipos permite deduzir automaticamente o tipo das expressões, mesmo que o programador omita as assinaturas de tipo (Hudak; Fasel, 1992).

O sistema de inferência de Haskell é baseado no modelo DamasMilner (DM), também conhecido como Hindley-Milner (HM), que é capaz de encontrar o tipo principal (*Principal Type*) de todas as expressões bem tipadas. Esse sistema é

profundamente fundamentado em conceitos da Lógica, Teoria dos Tipos e Teoria das Categorias (Silva, 2017; Silva; Roggia; Vasconcellos, 2018).

Essa relação formal é conhecida como Correspondência Curry-Howard Lambek (Silva; Roggia; Vasconcellos, 2018), que estabelece uma equivalência entre:

- Tipos e Proposições na Lógica construtivista (e.g., Lógica Intuicionista);
- Termos do Cálculo Lambda Tipado (i.e., programas/expressões) e Provas dessas proposições;
- Cálculo Lambda Tipado e Categorias Cartesianas Fechadas (CCC) na Teoria das Categorias.

A solidez matemática dessa correspondência garante as propriedades do sistema de tipos de Haskell (Silva, 2017), assegurando o rigor e a corretude em tempo de compilação.

3.1 Funções puras e composição

O núcleo da programação em Haskell reside nas funções puras, que não produzem efeitos colaterais. A composição é facilitada por construções como o *Pattern Matching*, que define o comportamento da função por meio de padrões de entrada, e o *Currying*, que realiza a aplicação parcial de argumentos e é nativo da linguagem (Hudak; Fasel, 1992).

Essa abordagem permite tratar funções como valores de primeira classe, viabilizando o uso de funções de ordem superior, fundamentais para a modularidade e reuso de código. A ausência de laços imperativos é suprida pela recursão como mecanismo de repetição. O fluxo condicional utiliza estruturas como guards (guarnições) ou *pattern matching*. A construção de listas é facilitada pelas *list comprehensions*, uma sintaxe concisa e declarativa semelhante à notação de conjuntos matemáticos (Hudak; Fasel, 1992).

Haskell incorpora dois principais tipos de polimorfismo: o paramétrico, que permite funções operarem sobre tipos genéricos (como [a], lista de qualquer tipo a), e o polimorfismo *ad hoc*, implementado pelas *Type Classes* (classes de tipos). Uma *Type Class* define um conjunto de funções (conhecidas como métodos) que um tipo deve suportar (Hudak; Fasel, 1992). Por exemplo, a classe Eq (igualdade) declara o método (==). Para que um tipo se torne comparável, ele precisa ser declarado como uma instância dessa classe, fornecendo uma implementação concreta para o método == (Hudak; Fasel, 1992).

A assinatura de uma função polimórfica *ad hoc* expressa uma restrição (*constraint*), indicando que a função só está definida para tipos que são instâncias da classe específica.

O sistema, contudo, apresenta complexidades. Uma dificuldade notória surge com as *Type Classes* que possuem múltiplos parâmetros. Problemas de ambiguidade na inferência de tipos ocorrem devido à falta de especialização durante o processo de dedução de tipos.

Tais ambiguidades motivaram revisões teóricas e o desenvolvimento de novos sistemas de tipos que permitem a definição de classes com múltiplos parâmetros, muitas vezes sem a necessidade de extensões como dependências funcionais (Ribeiro, 2013).

Haskell permite modelar dados complexos por meio da definição de novos tipos. O comando data define Tipos Algébricos de Dados (ADT), que representam

somas ou produtos de outros tipos, enquanto newtype encapsula tipos existentes (Hudak; Fasel, 1992).

Para manter a pureza funcional e, ainda assim, permitir operações com efeitos colaterais (como entrada/saída), Haskell utiliza abstrações de alto nível. Functors e Monads são essas estruturas que encapsulam e estruturam computações, controlando os efeitos. O embasamento teórico das Monads é rigoroso: elas são formalmente definidas como monoides na categoria dos endofuntores de Haskell (Silva, 2017; Silva, Roggia; Vasconcellos, 2018).

Essa definição, oriunda da Teoria das Categorias, demonstra como a Monad IO é capaz de encapsular operações de entrada e saída, garantindo que as ações impuras sejam contidas e sequenciadas dentro de um contexto funcional puro (Silva, Roggia; Vasconcellos, 2018).

Outro pilar da linguagem é a Avaliação Preguiçosa (*Lazy Evaluation*), também chamada de não-estrita (*non-strict*). Neste modelo de execução, as expressões não são avaliadas imediatamente, mas apenas quando o valor é estritamente necessário para o fluxo do programa (Hudak; Fasel, 1992). Isso resulta em duas grandes vantagens:

1. Otimização de Recursos: Evita o cálculo de valores que nunca serão usados, otimizando o uso de recursos.

2. Estruturas de Dados Infinitas: Permite a definição e o trabalho com estruturas de dados potencialmente infinitas (como a sequência de números de Fibonacci ou uma lista de todos os quadrados), das quais apenas uma porção finita será extraída para o cálculo real (Hudak; Fasel, 1992). No entanto, a não-estrita avaliação de Haskell complica a modelagem categórica. Embora se afirme que a categoria Hask existe (tipos como objetos, funções como morfismos), a presença de expressões não terminantes (como *undefined*) e a não estrita avaliação podem levar a casos em que a composição de morfismos falha em satisfazer as leis da categoria ou quebra a equivalência observacional em alguns contextos (Silva, 2017).

4 LINGUAGEM LISP

Lisp é uma linguagem de programação que surgiu no final dos anos 1950 e é utilizada até hoje em projetos que envolvem manipulação simbólica, inteligência artificial e metaprogramação (John McCarthy, 1958).

O nome “Lisp” é uma abreviação de *List Processing* (processador de listas), pois, internamente, as estruturas de dados mais comuns são listas, e a linguagem é especialmente eficiente em manipulá-las como se fossem programas ou dados (John McCarthy, 1960).

Lisp foi criado por John McCarthy, em 1958, no Instituto de Tecnologia de Massachusetts (MIT - EUA), como uma linguagem para manipular símbolos e listas, com aplicações em pesquisas de inteligência artificial. McCarthy propôs o uso do cálculo lambda, de Alonzo Church, como base formal para funções que tratam dados como funções e vice-versa (Mccarthy, 1960).

Nas décadas seguintes, Lisp evoluiu com várias implementações e dialetos (como Lisp 1.5, MacLisp, Interlisp, entre outros), que trouxeram melhorias em eficiência, coleta de lixo, ambientes interativos e outras características importante do Lisp (John McCarthy, 1962).

Lisp é uma linguagem de programação de alto nível. Apesar de antiga, é a segunda linguagem mais antiga ainda em uso, depois de Fortran. Entre as décadas de

1970 e 1980, era amplamente utilizada em projetos de inteligência artificial (Sebesta, 2012).

O Lisp foi fundamental para o avanço da computação, pois introduziu diversas ideias hoje comuns na maioria das linguagens de programação. Entre elas está a recursão, que ocorre quando uma função chama a si mesma para resolver um problema (Sebesta, 2012).

A sintaxe de Lisp é baseada em *s-expressions* (expressões simbólicas). Basicamente, tudo em Lisp é escrito como uma lista entre parênteses, em que o primeiro elemento é a função ou operador e os demais são os argumentos. Como essa notação é chamada de notação prefixa (o operador vem antes dos operandos). Ela diferencia Lisp de linguagens com notação infixa (como "3 + 4") (John McCarthy, 1958).

Para definir novas funções em Lisp, é preciso criar uma combinação de quatro elementos. O primeiro é o defun, que serve para avaliar a definição da função. O Segundo é o nome da função a ser definida; o terceiro é a lista de parâmetros; e o quarto é a expressão que determina o valor retornado pela função (John McCarthy, 1958).

O *lambda* é uma função que possui várias características, mas não é associada a nenhum símbolo, ou seja, é uma função anônima. A sintaxe do lambda é semelhante à de defun, omitindo-se apenas o nome da função. A utilidade do *lambda* em Lisp é fundamental, pois qualquer função corresponde, internamente, a um *lambda*. O símbolo da letra grega λ (*lambda*) vem da matemática, onde é usado no estudo de funções e aplicações funcionais. O cálculo *lambda* é uma ferramenta essencial para o estudo da semântica de linguagens de programação (Church, 1941).

As estruturas de dados são uma lista, um dos componentes básicos da linguagem Lisp, o próprio nome vem de *list processing*. Se o segundo elemento for outro cons, o resultado será uma lista, como no exemplo: Quando o último elemento for constante nil, será considerada uma lista vazia (John McCarthy, 1958).

Em Lisp, os átomos são os elementos mais simples, eles não podem ser decompostos em partes menores. Exemplos de átomos incluem números e símbolos que representam nomes de variáveis ou funções (John McCarthy, 1958). Esses elementos são chamados de atômicos porque não podem ser divididos. Para verificar se um dado é atômico, usa-se a função atom. Exemplo: Para testar se "xpto" é atômico e escrever essa expressão acima, o resultado será um erro, pois o Lisp interpretará "xpto" como uma variável sem valor atribuído. Para que o símbolo seja tratado como um valor literal e não como uma variável, utiliza-se a forma quote, que impede a avaliação do símbolo (John McCarthy, 1958).

Os macros são estruturas que permitem criar formas de expressar instruções na linguagem. Elas são utilizadas para gerar outras expressões Lisp a partir de seus argumentos antes da execução do programa. Além de transformar o código antes da avaliação, os macros são úteis porque permitem definir novas construções na linguagem, por exemplo, o comando *if*, que não poderia ser aplicado como uma função comum. (Graham, 1993). Na prática, muitas das "formas especiais" do Lisp são implementadas a partir de macros, cuja sintaxe é semelhante à de defun, usada para definir funções.

Com o surgimento de diversos dialetos, como FranzLisp, ZetaLisp, MacLisp, Scheme, entre outros, houve dificuldade de comunicação entre os membros da comunidade Lisp (John McCarthy, 1970).

Para resolver esse problema, foi criado o Common Lisp, um *standard* que visava unificar os dialetos, facilitar a troca de ideias e programas, e servir como uma versão padrão da linguagem (Common Lisp standard, 1984).

O *Scheme* é um dos dialetos mais importantes do Lisp. Foi criado por Guy L. Steele Jr. e Gerald Jay Sussman, no MIT, na década de 1970, com o objetivo de simplificar a linguagem e torná-la mais coerente do ponto de vista matemático. (Steele; Sussman, 1975).

Diferente de outros dialetos, o Scheme possui uma estrutura mínima e utiliza escopo léxico, o que torna o comportamento das variáveis mais previsível. Ele também trata as funções como elementos de primeira classe, permitindo passá-las como argumentos ou retorná-las como resultados, o que reforça o paradigma funcional (Steele; Sussman, 1975).

O *Scheme* é amplamente usado em pesquisas e no ensino de programação, sendo conhecido por sua clareza e simplicidade. Além disso, serviu de base para outras linguagens modernas, como o Racket (Abelson; Sussman, 1996).

5 RESULTADOS ENTRE COMPARAÇÕES CONCEITUAIS: LISP X HASKELL

Nesta seção são apresentadas comparações que destacam as diferenças de filosofia e de orientação de projeto, sem abordar detalhes específicos de implementação.

Objetivo de projeto: Lisp foi idealizada como uma ferramenta para pesquisa em inteligência artificial e para experimentação com representação simbólica; Haskell foi projetada como uma linguagem padrão para explorar e consolidar conceitos de programação funcional pura com garantia de tipos (Hudak *et al*.,1990).

Extensão e metaprogramação: Lisp encoraja a extensão da linguagem pelo próprio usuário (macros, DSLs internas), apoiada por sua natureza homoicônica; Haskell favorece extensão através do sistema de tipos e de metaprogramação em tempo de compilação (*Template* Haskell), resultando em *trade-offs* distintas entre flexibilidade dinâmica e segurança estática (Hudak *et al*.,1990).

Estilo cognitivo: programar em Lisp frequentemente estimula um estilo exploratório e iterativo (REPL, mutabilidade opcional), adequado à prototipação e à manipulação simbólica; programar em Haskell estimula a especificação formal de invariantes e desenho de interfaces tipadas, adequado a cenários onde segurança e correção têm prioridade (Hudak *et al*.,1990).

Relação com a teoria: ambas as linguagens estão firmemente enraizadas no legado teórico do cálculo λ, mas Haskell incorpora diretamente instrumentos teóricos (tipos algébricos, semânticas formais, técnicas para modelar efeitos), enquanto Lisp canalizou o legado teórico em poderosas práticas de metaprogramação aplicadas (Alonzo Church, 1936).

6 DISCUSSÃO

A reflexão sobre as linguagens funcionais no nível conceitual revela que se trata sobretudo de uma maneira de pensar: programar por transformações, privilegiar transparência referencial, e estruturar sistemas por composição (Alonzo Church, 1936). Lisp destaca-se por sua flexibilidade e metaprogramação, enquanto Haskell

valoriza rigor formal e segurança por tipagem. Lisp e Haskell encarnam, cada uma a seu modo, diferentes respostas a esse ideal, uma orientada à plasticidade e à metaprogramação; a outra, ao rigor formal e à segurança estática.

Compreender estes fundamentos é condição necessária para avaliar quando e como aplicar ideias funcionais em problemas concretos, seja adotando um estilo funcional em linguagens híbridas, seja escolhendo linguagens funcionais puras para domínios que beneficiem de garantias formais (Hudak *et al*.,1990).

7 CONCLUSÃO

A realização deste estudo permitiu compreender de forma aprofundada a implementação e os fundamentos das linguagens funcionais Haskell e Lisp, evidenciando suas contribuições históricas e conceituais para a evolução da ciência da computação. Ambas as linguagens, embora distintas em sua origem e estrutura, compartilham o paradigma funcional como base, oferecendo abordagens declarativas que influenciaram significativamente o desenvolvimento de linguagens modernas e de novas formas de pensar a programação.

No contexto histórico em que Haskell e Lisp foram desenvolvidas revela não apenas os avanços tecnológicos da época, mas também a consolidação de ideias que permanecem relevantes até os dias atuais. A linguagem Lisp, pioneira na aplicação de conceitos matemáticos e na manipulação simbólica, estabeleceu as bases para a inteligência artificial e para técnicas modernas de metaprogramação.

Já Haskell, com seu forte embasamento teórico em lógica e cálculo lambda, trouxe inovações na forma de tratar funções puras, avaliação preguiçosa e sistema de tipos, tornando-se referência em ambientes acadêmicos e em aplicações que exigem alta confiabilidade. A comparação entre Haskell e Lisp ao longo deste trabalho demonstrou que, apesar de ambas seguirem o paradigma funcional, elas apresentam diferenças estruturais e filosóficas que impactam diretamente em sua aplicabilidade e na forma como os problemas são modelados. Essa análise comparativa reforça a importância de conhecer as características específicas de cada linguagem para a escolha adequada em projetos de *software*.

Por fim, este estudo reafirma a relevância do paradigma funcional como uma abordagem sólida e atual na programação, sendo essencial tanto para a formação acadêmica quanto para o desenvolvimento de soluções computacionais robustas. O aprofundamento nos conceitos, estruturas e significados das linguagens Haskell e Lisp oferece não apenas um resgate histórico, mas também uma base conceitual importante para compreender os rumos da programação contemporânea.

REFERÊNCIAS

ABELSON, H.; SUSSMAN, G. J. **Structure and Interpretation of Computer Programs.** 2. ed. Cambridge: MIT Press. 1996. Disponível em: https://mitpress.mit.edu/9780262510875/structure-and-interpretation-of-computer-programs/. Acesso em: 6 nov. 2025.

HUDAK, P.; JONES, S. P.; WADLER, P.; HUGHES, J. et al. **Report on the Programming Language Haskell:** a non-strict, purely functional language. Version 1.0. Yale University. 1990. Disponível em:

https://www.mat.uc.pt/~pedro/lectivos/docs/haskell98-report.pdf. Acesso em: 6 nov. 2025.

CHURCH, A. An Unsolvable Problem of Elementary Number Theory. **American Journal of Mathematics**, v.58, n.2, p.345–363. 1936. Disponível em: https://doi.org/10.2307/2371045. Acesso em: 6 nov. 2025.

CHURCH, A. **The Calculi of Lambda-Conversion.** Princeton: Princeton University Press. 1941. Disponível em: https://press.princeton.edu/books/hardcover/9780691079872/the-calculi-of-lambda-conversion. Acesso em: 6 nov. 2025.

DUARTE, R. M.; DU BOIS, A. R.; CAVALHEIRO, G. G. H.; PILLA, M. L. (2019). Verificando a interferência do escalonador do Glasgow Haskell Compiler em aplicações usando STM Haskell. **Revista de Informática Teórica e Aplicada – RITA**, v.26, n.2, p.116–134. 2019. Disponível em: https://seer.ufrgs.br/rita/article/view/94038. Acesso em: 6 nov. 2025.

FRIEDMAN, D. P.; FEELEY, M. **The Scheme Programming Language**. 3. ed. Cambridge: MIT Press. 1996. Disponível em: https://mitpress.mit.edu/9780262258166/the-scheme-programming-language/. Acesso em: 6 nov. 2025.

GRAHAM, P. **On Lisp: Advanced Techniques for Common Lisp**. Englewood Cliffs: Prentice Hall.1993. Disponível em: https://paulgraham.com/onlisp.html. Acesso em: 6 nov. 2025.

HUDAK, P.; FASEL, J. H. A gentle introduction to Haskell. **ACM SIGPLAN Notices**, v. 27, n.5, T-1–T-53. 1992. Disponível em: https://dl.acm.org/doi/10.1145/130697.130698. Acesso em: 6 nov. 2025.

HUGHES, J. Why Functional Programming Matters. **The Computer Journal**, v.32, n.2, 1989/1990. 1989. Disponível em: https://www.cs.kent.ac.uk/people/staff/dat/miranda/whyfp90.pdf. Acesso em: 6 nov. 2025.

LAUNCHBURY, J. A Natural Semantics for Lazy Evaluation. In: **Proceedings of the 20th ACM SIGPLAN SIGACT Symposium on Principles of Programming Languages** *(POPL)*. 1993. Disponível em: https://dl.acm.org/doi/10.1145/158511.158618. Acesso em: 6 nov. 2025.

McCARTHY, J. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. **Communications of the ACM,** v.3, n.4, p.184–195. 1960. Disponível em: https://www-formal.stanford.edu/jmc/recursive.pdf. Acesso em: 6 nov. 2025.

McCARTHY, J. **LISP — Notes on its Past and Future**. Stanford University. 1980. Disponível em: https://www-formal.stanford.edu/jmc/lisp20th.pdf. Acesso em: 6 nov. 2025.

MOGGI, E. Notions of computation and monads. **Information and Computation.** 1991. Disponível em: https://www.cs.cmu.edu/~crary/819-f09/Moggi91.pdf. Acesso em: 6 nov. 2025.

RIBEIRO, R. G. **Classes de tipos com múltiplos parâmetros e opcionais em Haskell** (Tese de doutorado). Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Belo Horizonte, MG, Brasil. 2013. Disponível em: https://repositorio.ufmg.br/handle/1843/ESBF-9GMJLU. Acesso em: 6 nov. 2025.

SEBESTA, R. W. **Conceitos de Linguagens de Programação.** 10. ed. São Paulo: Pearson. 2012. Disponível em: https://www.pearson.com/en-us/subject-catalog/p/conceitos-de-linguagens-de-programacao/P200000009169. Acesso em: 6 nov. 2025.

SILVA, R. C. G. **Visão categórica do sistema de tipos de Haskell** *(*Trabalho de conclusão de curso). Universidade do Estado de Santa Catarina, Joinville, SC, Brasil. 2017. Disponível em: https://repositorio.udesc.br/handle/11311/1001. Acesso em: 6 nov. 2025.

SILVA, R. C. G.; ROGGIA, K. G.; VASCONCELLOS, C. D. Haskell type system analysis: Análise do sistema de tipos de Haskell. **Revista de Informática Teórica e Aplicada – RITA,** v. 25, n.3, 75–88. 2018. Disponível em: https://seer.ufrgs.br/rita/article/view/79256. Acesso em: 6 nov. 2025.

STEELE, G. L. Jr.; SUSSMAN, G. J. **The Revised Report on Scheme:** A Dialect of Lisp. Cambridge: MIT Artificial Intelligence Laboratory. 1975. Disponível em: https://dspace.mit.edu/handle/1721.1/6094. Acesso em: 6 nov. 2025.