
**UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO**

**CENTRO TECNOLÓGICO**

**COLEGIADO DO CURSO DE CIÊNCIA DA COMPUTAÇÃO**

Ricardo Magalhães Santos Filho

# Tradução de programação funcional para grafos *Dataflow* executáveis

Vitória, ES

2025

Ricardo Magalhães Santos Filho

# Tradução de programação funcional para grafos *Dataflow* executáveis

Anteprojeto apresentado ao Colegiado do Curso de Ciência da Computação do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito para aprovação na Disciplina Projeto de Graduação I.

Universidade Federal do Espírito Santo – UFES

Centro Tecnológico

Colegiado do Curso de Ciência da Computação

Orientador: Prof. Dr. Alberto Ferreira de Souza

Coorientador: Prof. Dr. Tiago Assumpção de Oliveira Alves

Vitória, ES

2025

# Sumário

# 1 Introdução

O modelo *Dataflow* representa uma forma natural de exploração de paralelismo, permitindo que operações sejam executadas assim que seus operandos estejam disponíveis. Diferentemente do modelo sequencial tradicional, esse paradigma não exige que o programador gerencie diretamente mecanismos complexos de controle de concorrência, como semáforos ou travas (*locks*), para aproveitar o potencial de arquiteturas com múltiplos ou muitos núcleos. Em vez disso, exige apenas a especificação explícita das dependências entre os blocos de código, deixando que a ativação das operações seja guiada pela disponibilidade dos dados.

Este trabalho propõe a tradução de programas escritos em um subconjunto de Haskell (AL., 2010) para grafos *Dataflow*, conectando o paradigma funcional — baseado em expressões puras — com um modelo de execução paralela orientado a dados. O compilador desenvolvido viabiliza essa integração por meio da geração automática de grafos compatíveis com ambientes de execução distribuída, com suporte opcional à conversão para o *assembly* utilizado por arquiteturas baseadas no modelo *TALM* (MARZULO et al., 2010).

## 1.1 Motivação e Justificativa

Para que o *Dataflow* se torne mais acessível, é necessário integrá-lo a linguagens e paradigmas existentes. Neste projeto, propõe-se utilizar Haskell como linguagem de entrada por ser uma linguagem funcional de ampla adoção acadêmica. A ausência de estados e a natureza declarativa do modelo funcional facilita a tradução para o modelo *Dataflow*, historicamente implementado com maior sucesso em linguagens funcionais. Assim, o trabalho justifica-se pela importância do desenvolvimento de compiladores híbridos e para a adoção prática de arquiteturas baseadas em fluxo de dados.

## 1.2 Objetivos

**Objetivo Geral:** Desenvolver um compilador que traduza programas escritos em um subconjunto de Haskell para grafos *Dataflow*, representados no formato `.dot`, com opção de geração de código no *assembly THLL*.

**Objetivos Específicos:**

- Definir formalmente um subconjunto funcional de Haskell com expressividade Turing-completa.

- Implementar análise léxica, sintática e semântica para esse subconjunto.
- Construir uma representação intermediária (*AST*) e convertê-la em grafo *Dataflow*.
- Exportar o grafo para o formato `.dot`, compatível com o *Graphviz*.
- Opcionalmente, traduzir o grafo para código *THLL* executável no ambiente *TALM / Trebuchet*.
- Verificar a corretude através de uma análise formal da linguagem aceita pela gramática e uma inspeção estrutural das representações intermediárias geradas pelo compilador.
- Avaliar o desempenho estrutural e funcional dos programas traduzidos.

## 1.3 Método de Desenvolvimento do Trabalho

O desenvolvimento do trabalho será conduzido com base em uma arquitetura modular, com divisão clara entre fases, responsabilidades e produtos intermediários. *Scripts* complementares serão empregados para automatização de testes, coleta de métricas e visualização de resultados. Toda a infraestrutura do projeto será construída de forma a favorecer a rastreabilidade, a reprodutibilidade e a verificação incremental dos resultados.

### Fase 1 – Definição e Formalização do Subconjunto de Haskell

A primeira etapa consiste na definição de um subconjunto da linguagem Haskell que seja Turing-completa. Serão incluídas funções puras, inclusive de ordem superior, expressões *lambda* com aplicação funcional explícita, estruturas recursivas (inclusive chamadas recursivas mútuas), expressões condicionais tanto com `if-then-else` quanto com `case`, e um conjunto de tipos básicos: inteiros, números de ponto flutuante, valores booleanos, caracteres, *strings* e listas homogêneas. Também serão incluídos mecanismos de entrada e saída simples.

A formalização desse subconjunto será feita por meio de uma gramática parcial escrita em *EBNF*, oferecendo clareza na estrutura da linguagem. A escolha do subconjunto será justificada teoricamente por sua equivalência ao cálculo *lambda*, assegurando a completude.

### Fase 2 – Análise Léxica e Sintática

Com base na gramática formalizada, serão construídos analisadores léxicos e sintáticos capazes de reconhecer corretamente os programas escritos no subconjunto definido. Esses analisadores identificarão os *tokens* válidos e construirão a árvore de sintaxe abstrata (*AST*) correspondente à estrutura hierárquica do código. A *AST* será posteriormente serializada em formato textual intermediário, permitindo o desacoplamento entre as etapas de análise e síntese do compilador.

### Fase 3 – Representação Intermediária e Análise Semântica

Na terceira fase, a *AST* será processada por um módulo responsável por construir uma representação intermediária (*IR*) funcional, estruturada de modo a capturar o fluxo de dados, aplicações de função, chamadas recursivas e agrupamentos de expressões. Essa representação servirá de ponte entre o código funcional original e o modelo orientado a dados que será utilizado na fase seguinte.

Concomitantemente, será realizada uma verificação semântica, com o objetivo de garantir a integridade estrutural e o significado das construções presentes no programa. Essa verificação incluirá a detecção de variáveis livres — identificadores utilizados sem declaração visível no escopo atual —, a validação da aridade em aplicações de função e na construção de listas, assegurando que o número de argumentos ou elementos corresponda às regras esperadas, e a conformidade estrutural de funções recursivas, inclusive nos casos de recursão mútua entre definições interdependentes.

Também será realizada uma verificação de tipos básica, limitada ao subconjunto da linguagem suportado, que incluirá inteiros, números de ponto flutuante, valores booleanos, caracteres, *strings* e listas homogêneas. Essa análise deverá assegurar, por exemplo, que operadores sejam aplicados a operandos compatíveis e que expressões condicionais apresentem tipos coerentes nos ramos. Embora não se implemente um sistema completo de inferência de tipos, essa verificação será suficiente para prevenir erros semânticos.

### Fase 4 – Geração de Grafo *Dataflow*

A representação intermediária será convertida em um grafo orientado a dados, onde cada nó representa uma operação elementar (função, operador ou estrutura de controle) e as arestas indicam dependências de dados entre essas operações. Essa estrutura permitirá uma visualização explícita do paralelismo implícito presente no programa funcional. O grafo será exportado em um formato textual padronizado, permitindo sua renderização gráfica para inspeção visual, depuração e análise do comportamento estrutural.

### Fase 5 – Tradução para Código *THLL*

A partir do grafo orientado a dados, será gerado um código intermediário na linguagem *THLL*, uma notação projetada para descrever blocos computacionais com entradas e saídas explícitas. Cada operação identificada no grafo será mapeada para uma função anotada, respeitando o modelo *Dataflow*. O código resultante poderá ser posteriormente traduzido para uma representação executável por um ambiente baseado em grafos orientados a dados. A geração do *THLL* permitirá a experimentação com diferentes mapeamentos entre lógica funcional e execução paralela.

## Fase 6 – Verificação de Corretude

A verificação da corretude será feita estruturalmente. A primeira consiste na completude da linguagem aceita pela própria gramática. Depois, virá a inspeção estrutural da saída gerada pelo compilador. Isso inclui verificar se a árvore de sintaxe abstrata preserva a hierarquia esperada de chamadas e expressões, se a representação intermediária (*IR*) reflete corretamente o encadeamento de funções e dependências de dados, e se o grafo gerado contém todos os nós e conexões esperadas. Essa verificação será feita inicialmente por inspeção manual de casos simples, usando visualizações do grafo e impressões da *IR*, e posteriormente será automatizada por *scripts* que checam propriedades estruturais, como a existência de caminhos entre nós dependentes, ausência de ciclos em contextos não-recursivos e integridade do mapeamento entre nós e operações.

## Fase 7 – Avaliação de Desempenho

A avaliação de desempenho será realizada com base na estrutura do grafo gerado e em sua eficiência potencial de execução. Em compiladores, essa análise pode ser feita em duas frentes principais: a análise estática da estrutura intermediária gerada (neste caso, o grafo *Dataflow*), e a análise dinâmica da execução simulada, quando disponível.

Na análise estática, serão extraídas métricas diretamente da topologia do grafo, como:

- ***Fan-in*** e ***fan-out***: número de dependências de entrada e saída de cada nó, que indicam concentração de dados e possíveis gargalos.
- **Profundidade crítica**: o maior caminho de dependências entre entrada e saída, que representa a menor latência possível do programa sob execução paralela ideal.
- **Largura máxima de nível**: quantidade de nós executáveis simultaneamente em cada nível de avaliação, refletindo o paralelismo estrutural disponível.

Essas métricas serão obtidas por meio de algoritmos de análise de grafos, como ordenação topológica, contagem de predecessores, e marcação de níveis. O grafo será percorrido computando o tempo mínimo de ativação de cada nó com base em suas dependências. Isso permite estimar o limite inferior de tempo de execução e o grau de paralelismo efetivo do programa.

Na análise dinâmica, quando a execução simulada estiver disponível, será medida a duração real da execução, o número de nós ativados em paralelo por etapa e a ocupação de unidades computacionais simuladas. Essa medição permite validar os resultados da análise estrutural e identificar sobrecargas não visíveis apenas na estrutura, como contenção em canais ou escalonamento subótimo.

O conjunto de testes será composto por programas com diferentes padrões de paralelismo, incluindo composição funcional densa, recursão profunda, uso extensivo de listas, e *pipelines* com etapas independentes. A partir dos dados coletados, será possível caracterizar o impacto estrutural das decisões de compilação e identificar regiões do grafo que limitam o paralelismo. Isso, por sua vez, poderá fundamentar otimizações futuras no núcleo do compilador.

### Fase 8 – Organização e Artefatos do Projeto

O projeto será mantido sob controle de versão por meio de um sistema distribuído, com ramificações específicas para cada fase de desenvolvimento (análise léxica, análise sintática, análise semântica, *IR*, geração de grafo, geração de código, testes e documentação). Essa organização permitirá isolar alterações, facilitar *reverts* e manter um histórico auditável das decisões técnicas ao longo do tempo.

Como parte do desenvolvimento, serão produzidos e versionados os seguintes artefatos: o código-fonte completo dos módulos do compilador; um conjunto de testes representativos com suas entradas e saídas esperadas; os *logs* de execução e validação; os arquivos intermediários (como *ASTs* e grafos `.dot`); gráficos e métricas extraídas da análise estrutural e de desempenho dos grafos; além de documentação técnica e de uso descrevendo a arquitetura do sistema, instruções de compilação e execução, e os formatos adotados em cada etapa da *pipeline*.

## 1.4 Cronograma

- **Atividade 1:** Definir formalmente o subconjunto de Haskell com base funcional, incluindo tipos, operações, entrada e saída.
- **Atividade 2:** Desenvolver analisadores léxico e sintático com base na gramática do subconjunto, com geração da AST.
- **Atividade 3:** Implementar o frontend do compilador, incluindo a representação intermediária (IR).
- **Atividade 4:** Gerar o grafo Dataflow e exportá-lo no formato `.dot`, compatível com ferramentas de visualização.
- **Atividade 5:** Traduzir o grafo para código THLL (anotações em C), permitindo posterior compilação com Couillard para execução no ambiente TALM (opcional).
- **Atividade 6:** Realizar testes de corretude com programas escritos no subconjunto e verificação por equivalência de saída.

- **Atividade 7:** Avaliar o desempenho estrutural e funcional dos programas traduzidos por meio de benchmarks automatizados.
- **Atividade 8:** Documentar a metodologia, decisões de projeto, resultados e limitações encontradas.
- **Atividade 9:** Revisar o material, formatar e submeter a versão final do TCC II.

| **Atividade** | **Mai** | **Jun** | **Jul** | **Ago** | **Set** | **Out** | **Nov** | **Dez** |
|---|---|---|---|---|---|---|---|---|
| Definir subconjunto de Haskell | X | X | | | | | | |
| Desenvolver analisadores | | X | X | | | | | |
| Implementar frontend (AST + IR) | | | X | X | | | | |
| Gerar grafo Dataflow e exportar (.dot) | | | | X | X | | | |
| Traduzir grafo para THLL (opcional) | | | | | X | X | | |
| Testar corretude (I/O, estruturas) | | | | | X | X | | |
| Analisar desempenho (scripts) | | | | | | X | X | |
| Documentar metodologia e resultados | | | | | | | X | |
| Revisar, finalizar e submeter TCC 2 | | | | | | | | X |

Tabela 1 – Cronograma de execução das atividades do TCC (maio a dezembro de 2025).

# 2 Fundamentação Teórica e Tecnologias Utilizadas

## 2.1 Definição de uma Linguagem

A construção de um compilador requer, como ponto de partida, a definição formal da linguagem que ele será capaz de processar. Essa definição é expressa por meio de uma gramática, que estabelece as regras de formação das sentenças válidas. A análise sintática, etapa essencial do processo de compilação, utiliza essa gramática para verificar se a estrutura de um programa corresponde àquela especificada pela linguagem.

A partir dessa estrutura sintática, é possível acoplar ações semânticas durante o processo de reconhecimento — prática que dá origem à técnica conhecida como *tradução dirigida pela sintaxe*, na qual a própria derivação do programa orienta a geração de representações intermediárias ou de código alvo (AHO et al., 2006).

### 2.1.1 Gramáticas Livres de Contexto

As gramáticas livres de contexto são amplamente utilizadas na definição da sintaxe de linguagens de programação. Sua expressividade é suficiente para descrever estruturas fundamentais como expressões, comandos compostos, blocos de escopo e chamadas de função. Além disso, elas são compatíveis com algoritmos eficientes de análise sintática, o que as torna práticas tanto do ponto de vista teórico quanto da implementação (HOPCROFT; MOTWANI; ULLMAN, 2007).

Formalmente, uma gramática livre de contexto é uma quádrupla $G = (V, \Sigma, P, S)$, onde:

- $V$ é um conjunto finito de variáveis (ou símbolos não-terminais);
- $\Sigma$ é um conjunto finito de símbolos terminais, tal que $V \cap \Sigma = \emptyset$;
- $P$ é um conjunto finito de produções da forma $A \to \alpha$, com $A \in V$ e $\alpha \in (V \cup \Sigma)^*$;
- $S \in V$ é o símbolo inicial.

Cada produção indica que um símbolo não-terminal pode ser substituído por uma sequência finita de terminais e não-terminais. O processo de derivação consiste em aplicar essas substituições repetidamente até que se obtenha uma cadeia formada apenas por símbolos terminais.

### 2.1.2 EBNF: Uma Linguagem de Descrição de Linguagens

Embora as gramáticas livres de contexto sejam suficientemente expressivas para descrever a maioria das linguagens de programação, sua notação tradicional pode se tornar extensa e pouco legível em especificações mais complexas. A *EBNF* (Extended Backus-Naur Form) surge como uma alternativa mais concisa, sem perda de rigor formal (WIRTH, 1977).

A *EBNF* introduz operadores sintáticos que facilitam a escrita de produções frequentes:

- `[ ...]` indica elementos opcionais;
- `{ ...}` denota repetições arbitrárias (zero ou mais vezes);
- `( ...)` permite o agrupamento de expressões;
- `|` expressa alternativas.

Esses recursos tornam a *EBNF* particularmente adequada para uso em documentação técnica e ferramentas de geração automática de analisadores. Neste trabalho, ela será utilizada para especificar formalmente o subconjunto funcional da linguagem *Haskell* aceito pelo compilador.

## 2.2 Análise Léxica

A análise léxica é a primeira etapa da cadeia de compilação e tem por função segmentar a entrada textual em unidades léxicas significativas denominadas *tokens*. Essa segmentação é baseada em padrões regulares associados a cada categoria sintática da linguagem. A implementação dessa etapa é tradicionalmente feita por meio de autômatos finitos, cuja construção se baseia na equivalência com expressões regulares (HOPCROFT; MOTWANI; ULLMAN, 2007). O resultado é uma sequência linear de símbolos terminais que alimentará a fase de análise sintática.

### 2.2.1 Autômatos Finitos

Autômatos finitos são modelos formais de computação utilizados para reconhecer linguagens regulares. Eles constituem a base teórica da análise léxica, permitindo representar, de maneira eficiente, os padrões que definem os *tokens* da linguagem de entrada.

Existem dois tipos principais de autômatos finitos: os determinísticos (*AFD*) e os não determinísticos (*AFN*). No modelo não determinístico, a transição de um estado para

outro pode ocorrer de forma ambígua, com múltiplas alternativas possíveis para um mesmo símbolo, ou até mesmo sem consumir símbolos de entrada. Já no modelo determinístico, cada estado possui exatamente uma transição definida para cada símbolo do alfabeto, o que facilita a implementação eficiente de analisadores léxicos.

Apesar da diferença operacional, ambos os modelos possuem o mesmo poder expressivo: qualquer linguagem reconhecível por um *AFN* também pode ser reconhecida por um *AFD* equivalente. Essa equivalência foi demonstrada formalmente por Michael Rabin e Dana Scott em sua prova de 1959, na qual também apresentaram o algoritmo de conversão de *AFNs* para *AFDs* por meio do processo de determinização (RABIN; SCOTT, 1959).

A análise léxica geralmente utiliza autômatos determinísticos por serem mais eficientes na execução, com complexidade linear em relação ao tamanho da entrada. Ferramentas como `Alex` (COMMUNITY, 2023b) realizam internamente essa conversão: as expressões regulares fornecidas pelo programador são traduzidas para *AFNs* e, posteriormente, transformadas em *AFDs* minimizados antes da geração do código do analisador.

Um autômato finito determinístico é formalmente definido como uma 5-upla $M = (Q, \Sigma, \delta, q_0, F)$, onde:

- $Q$ é um conjunto finito de estados;
- $\Sigma$ é o alfabeto de entrada;
- $\delta : Q \times \Sigma \to Q$ é a função de transição;
- $q_0 \in Q$ é o estado inicial;
- $F \subseteq Q$ é o conjunto de estados finais (ou de aceitação).

O reconhecimento ocorre processando uma cadeia de entrada símbolo a símbolo, transitando entre estados conforme definido pela função $\delta$. Uma cadeia é aceita se, após consumir todos os símbolos, o autômato estiver em um estado final.

## 2.3 Análise Sintática

A análise sintática verifica se a sequência de *tokens* produzida pela análise léxica constitui uma sentença válida de acordo com a gramática da linguagem. Esse processo constrói uma estrutura hierárquica — geralmente uma árvore de derivação — que representa a forma como a entrada pode ser derivada a partir do símbolo inicial.

Neste trabalho, a análise sintática será implementada com o auxílio do gerador `Happy`, que utiliza técnicas de análise *bottom-up* baseadas em tabelas. A gramática da

linguagem, escrita em *EBNF*, é convertida internamente em tabelas de *parsing* para orientar o reconhecimento da estrutura do programa.

### 2.3.1 Análise Sintática *Bottom-up*

A análise sintática *bottom-up* consiste em reconstruir a derivação da entrada partindo dos símbolos terminais e aplicando as produções da gramática de forma inversa até alcançar o símbolo inicial. Esse processo, conhecido como *redução*, segue o princípio da derivação à direita.

Em *parsers bottom-up*, os símbolos da entrada são lidos e empilhados até que um padrão correspondente ao lado direito de uma produção seja identificado e reduzido ao seu lado esquerdo. Esse modelo é a base de algoritmos como os analisadores *LR*.

### 2.3.2 Analisadores Sintáticos *LR*

Analisadores *LR* (*Left-to-right, Rightmost derivation*) são algoritmos *bottom-up* que utilizam tabelas de transição para guiar o reconhecimento da entrada. São capazes de lidar com a maioria das construções sintáticas utilizadas em linguagens de programação e operam com eficiência linear.

Esses analisadores mantêm uma pilha de estados e *tokens*, consultando duas tabelas principais: `action` e `goto`. A primeira determina se deve haver um deslocamento, *redução* ou aceitação; a segunda orienta as transições entre estados após uma *redução*.

Dentre as variações, destacam-se os modelos *SLR*, *LALR* e *LR(1)*, que diferem em complexidade e poder de expressão. O analisador sintático deste projeto utiliza a variante *LALR(1)*, implementada pelo `Happy` (COMMUNITY, 2023c).

## 2.4 Análise Semântica

A análise semântica é a etapa do processo de compilação responsável por garantir que um programa, além de estar sintaticamente correto, também seja logicamente coerente. Essa fase opera sobre a árvore de sintaxe abstrata (*AST*) e introduz no processo informações como tipos, escopos, contextos e restrições semânticas. Também é nela que se estabelecem estruturas de símbolos e outras auxiliares utilizadas na geração de código ou em verificações posteriores (NIELSON; NIELSON, 2007).

### 2.4.1 Verificação de Tipos e Escopo

A verificação de tipos consiste em assegurar que cada expressão da linguagem respeita as regras de formação de tipos estabelecidas. Em linguagens funcionalmente

tipadas, como *Haskell*, isso inclui a compatibilidade de tipos em aplicações de funções, uso de operadores e construções condicionais.

Além dos tipos, é necessário garantir que cada identificador referenciado esteja devidamente declarado e visível no escopo atual. A verificação de escopo envolve a construção e manutenção de um ambiente de variáveis ou tabela de símbolos, que relaciona cada nome a sua declaração correspondente e ao contexto em que é válido.

### 2.4.2 Variáveis Livres, Aridade e Recursividade

A presença de variáveis livres — ou seja, identificadores utilizados sem definição local — compromete a corretude semântica do programa. Por isso, é necessário realizar uma varredura que identifique e rejeite expressões contendo referências externas não resolvidas.

Também é fundamental verificar a *aridade* de cada função, ou seja, o número de argumentos esperados por sua definição. Em linguagens funcionais com aplicação *curried*, como *Haskell*, a verificação de aridade deve levar em conta o modo de aplicação e a estrutura aninhada das funções.

Por fim, o suporte a recursão exige atenção adicional. A análise semântica precisa garantir que chamadas recursivas respeitam a assinatura da função original e não violam as regras de escopo ou dependência cíclica entre declarações. Estruturas de recursão mútua também devem ser reconhecidas e corretamente anotadas na representação intermediária.

## 2.5 Representações Intermediárias

As representações intermediárias (*IRs*) são estruturas que descrevem o programa após as fases de análise, servindo como base para otimizações e geração de código. Elas permitem separar a descrição da lógica do programa das particularidades de *hardware* ou de linguagens de destino. Em compiladores modernos, é comum a existência de múltiplos níveis de *IR*, variando entre formas mais próximas à linguagem-fonte (como a $AST$) e representações mais baixas, voltadas à execução.

Neste trabalho, são utilizadas duas *IRs* principais: a Árvore de Sintaxe Abstrata ($AST$), que representa a estrutura lógica do código após a análise sintática, e uma representação em grafo orientado, adequada à geração de programas para o modelo *Dataflow*.

### 2.5.1 Árvore de Sintaxe Abstrata (AST)

A $AST$ representa a hierarquia das construções do programa de forma simplificada, eliminando detalhes sintáticos irrelevantes (como parênteses e palavras-chave). Cada nó da árvore corresponde a uma construção da linguagem (como expressões, declarações ou

comandos), enquanto os filhos representam seus constituintes.

Diferente da árvore de derivação completa, a *AST* é projetada para refletir a semântica da linguagem e facilitar sua análise e transformação. Durante a análise semântica, informações adicionais podem ser acopladas aos nós da *AST*, como tipos inferidos, escopos, ou anotações para geração de código.

Neste projeto, a *AST* é construída automaticamente a partir do *parser* gerado por `Happy` (COMMUNITY, 2023c), com suporte adicional para acoplamento de ações semânticas e transformação posterior em grafo funcional.

### 2.5.2 Representando Fluxo de Dados por Grafos

A segunda forma de representação intermediária adotada é um grafo orientado a dados, no qual os nós correspondem a operações (funções, aplicações, testes condicionais) e as arestas representam dependências de dados entre essas operações. Essa estrutura é inspirada no modelo *Dataflow*, no qual a execução é ativada pela disponibilidade de operandos.

Cada subgrafo pode ser interpretado como uma unidade computacional autônoma, com entradas e saídas bem definidas. Funções compostas e estruturas de controle são transformadas em subgrafos interligados, permitindo a decomposição natural em *super-instruções*. Essa abordagem favorece a paralelização automática e a análise de dependências.

## 2.6 Corretude de um Compilador

A corretude de um compilador pode ser entendida como a garantia de que todas as transformações realizadas ao longo do processo de compilação preservam a estrutura e o significado interno do programa, conforme definido pela especificação da linguagem. Em contextos onde não há uma semântica operacional executável como referência — como no caso de linguagens definidas por subconjuntos ou traduções não padronizadas —, essa corretude é assegurada por métodos formais baseados na gramática da linguagem e pela inspeção das estruturas intermediárias geradas.

No compilador aqui proposto, a verificação de corretude não depende da comparação com a execução de um programa funcional de referência, mas sim da consistência interna de suas fases: a conformidade entre a entrada e a árvore de sintaxe abstrata gerada, a validade da transformação desta em uma representação intermediária orientada a dados, e a fidelidade dessa representação ao ser convertida em um grafo.

### 2.6.1 Verificação Estrutural

A verificação estrutural consiste na inspeção detalhada das representações intermediárias produzidas pelo compilador — *AST* e grafo — para garantir que mantenham a coerência e o encadeamento esperados conforme as regras da linguagem. Isso inclui validar que a hierarquia sintática foi corretamente refletida na árvore de análise, que as dependências e escopos foram respeitados na *IR*, e que as operações e fluxos de dados representados no grafo estão completos, conexos e semanticamente válidos.

Ao adotar esse enfoque estrutural, evita-se a necessidade de um modelo funcional de execução como oráculo de corretude, confiando em propriedades formais da linguagem e nas invariantes mantidas ao longo das etapas de tradução.

## 2.7 Otimização de um Compilador

A etapa de otimização em um compilador tem por objetivo transformar a representação intermediária do programa de forma a melhorar seu desempenho, reduzir o uso de recursos ou preparar o código para fases posteriores de geração eficiente. Essas transformações preservam a semântica original do programa, mas alteram sua estrutura interna visando eficiência computacional.

Entre as técnicas mais comuns estão: eliminação de código morto, propagação de constantes, movimentação de invariantes de laço, fusão de expressões redundantes e reescrita de padrões ineficientes. A maioria dessas otimizações é realizada sobre a Árvore de Sintaxe Abstrata (*AST*) ou uma forma intermediária controlada como três-endereços (*three-address code*).

Um dos fundamentos da otimização moderna é a análise de fluxo de dados, que examina como valores se propagam ao longo do programa. Essa análise global permite identificar com precisão quais variáveis são utilizadas, modificadas ou são constantes em cada ponto do código, viabilizando otimizações dependentes de contexto. Técnicas como análise de disponibilidade de expressões, *live variable analysis* e análise de alcance de definições (*reaching definitions*) são clássicos exemplos desse tipo de abordagem (AHO et al., 2006).

Embora a otimização seja considerada uma fase separada, seu impacto é cumulativo e afeta diretamente a qualidade do código gerado, a estrutura dos grafos de execução e a viabilidade de paralelismo ou reuso de blocos computacionais.

## 2.8 Definição do Modelo *Dataflow*

O modelo de execução *Dataflow* descreve programas como redes de operações conectadas por canais de dados, onde cada operação é ativada automaticamente quando todos os seus operandos estão disponíveis. Esse paradigma rompe com o controle sequencial de fluxo e favorece uma execução naturalmente paralela, já que múltiplas operações independentes podem ocorrer simultaneamente (DENNIS; MISUNAS, 1975).

Formalmente, um programa baseado em *Dataflow* pode ser representado por um grafo bipartido dirigido

$$G = (V_O \cup V_D, E),$$

onde:

- $V_O$ é o conjunto de *nós de operação*, cada um representando uma instrução ou função computável;
- $V_D$ é o conjunto de *nós de dados*, que representam valores intermediários produzidos ou consumidos;
- $E \subseteq (V_O \times V_D) \cup (V_D \times V_O)$ é o conjunto de arestas dirigidas, conectando operações às suas saídas (dados produzidos) e dados às operações seguintes (dados consumidos).

Essa separação entre dados e operações, típica de representações como grafos bipartidos, permite explicitar o fluxo de informações e identificar naturalmente oportunidades de paralelismo. A ativação de um nó de operação ocorre apenas quando todos os seus dados de entrada estão disponíveis.

Considere a computação da expressão $z = (a + b) \times (c - d)$. Seu grafo *Dataflow* correspondente é representado na Figura 1, em que nós circulares representam dados e nós retangulares representam operações.

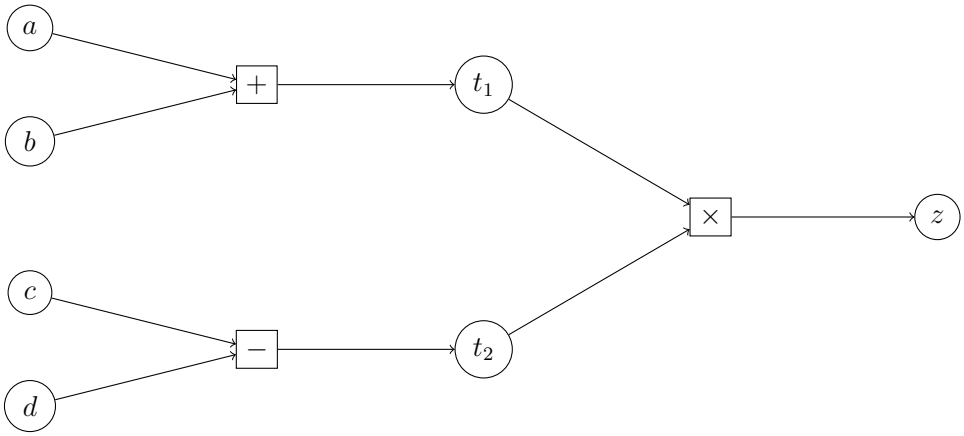


Figura 1 – Grafo *Dataflow* para a expressão $z = (a + b) \times (c - d)$.

Neste grafo, as arestas indicam o fluxo de dados, e as operações são executadas assim que recebem todos os seus operandos. Esse estilo de representação facilita a análise de paralelismo, visto que as operações de soma e subtração podem ser executadas em paralelo, antes da multiplicação final.

## 2.9 TALM

A arquitetura *TALM* (*Thread Architecture with Local Memories*) é uma plataforma de execução baseada no modelo *Dataflow* voltada à execução eficiente de programas paralelos em arquiteturas *multicore* convencionais. Ela foi proposta na tese de doutorado de Tiago Assumpção de Oliveira Alves (ALVES, 2014), com o objetivo de viabilizar o modelo *Dataflow* no espaço de usuário.

### 2.9.1 Instruções TALM

O conjunto de instruções *TALM* forma a linguagem de montagem da arquitetura e corresponde às operações executáveis de granularidade fina ou composta no grafo *Dataflow*. Cada instrução representa um nó que dispara sua execução assim que seus operandos de entrada estão disponíveis.

Abaixo está a lista completa das instruções *TALM*, conforme documentado por Alves (ALVES, 2014):

| **Instrução** | **Categoria** | **Descrição** |
|---|---|---|
| `const` | Constante | Gera um valor constante como saída. |
| `add`, `sub`, `mul`, `div` | Aritmética | Operações aritméticas básicas sobre inteiros ou ponto flutuante. |
| `addi`, `subi`, `muli`, `divi` | Aritmética com Imediato | Versões das operações aritméticas com um operando imediato. |
| `and`, `or`, `xor`, `not` | Lógica *bit a bit* | Operações lógicas entre bits. |
| `eq`, `neq`, `lt`, `leq`, `gt`, `geq` | Comparação | Testes relacionais entre operandos. |
| `steer` | Controle de Fluxo | Encaminha operandos a saídas distintas com base em uma condição booleana. |
| `merge` | Controle de Fluxo | Escolhe um entre múltiplos valores com base na disponibilidade. |
| `split` | Dados | Duplica um valor para múltiplos destinos. |
| `callgroup` | Controle de Execução | Dispara um grupo de instâncias de *super-instrução*. |
| `callsnd` | Controle de Execução | Envia argumentos para uma instância específica. |
| `retsnd` | Controle de Execução | Envia valores de retorno a partir de uma instância. |
| `ret` | Controle de Execução | Sinaliza a finalização de uma instância. |
| `inctag` | Controle de Instância | Incrementa o identificador de instância (*tag*). |
| `tagop` | Controle de Instância | Manipula diretamente valores de *tag*. |
| `super` | *Super-instrução* | Instancia uma *super-instrução* comum. |
| `specsuper` | *Super-instrução* | Instancia uma *super-instrução* especulativa. |
| `superinstmacro` | *Macro* | Define um atalho nomeado para uma instância de *super-instrução*. |

Tabela 2 – Instruções *TALM* conforme especificadas por Alves (ALVES, 2014).

### 2.9.2 Super Instruções

*Super-instruções* são o principal mecanismo para adaptação de granularidade dentro da arquitetura *TALM*. Elas consistem em blocos de código definidos pelo programador em uma linguagem nativa da máquina-alvo (como *C*), compilados como bibliotecas compartilhadas e utilizados como unidades atômicas de execução no grafo *Dataflow*.

Cada *super-instrução* encapsula uma operação de maior granularidade e é representada como um nó no grafo, disparando sua execução somente quando todos os operandos de entrada estão disponíveis, conforme as regras do modelo *Dataflow*. Diferente das instruções de granularidade fina do *TALM*, as *super-instruções* executam diretamente no processador hospedeiro, permitindo melhor desempenho e reduzindo o custo de interpretação.

A sintaxe para instanciar uma *super-instrução* no código *assembly TALM* segue a forma:

```
super <nome>, <id_super>, <#saídas>, [entradas]
```

| Campo | Descrição |
|---|---|
| `<nome>` | Identificador da instância, utilizado para nomear as saídas. |
| `<id_super>` | Identificador numérico que referencia a implementação da *super-instrução* na biblioteca compilada. |
| `<#saídas>` | Número de operandos de saída produzidos pela *super-instrução*. |
| `[entradas]` | Lista de operandos de entrada exigidos pela *super-instrução*. |

Tabela 3 – Campos da diretiva `super` para instanciar *super-instruções TALM*

Além disso, o *TALM* oferece a variante `specsuper`, com a mesma sintaxe, usada para instanciar *super-instruções* especulativas.

Para facilitar a legibilidade do código *assembly* e reutilização de padrões, existe a diretiva `superinstmacro`, que permite criar mnemônicos personalizados para *super-instruções*. A sintaxe é:

```
superinst(<mnemônico>, <id_super>, <#saídas>, <especulativa>)
```

Após sua definição, o mnemônico pode ser utilizado para instanciar a *super-instrução* com uma sintaxe reduzida:

```
<mnemônico> <nome_instância>, [entradas]
```

Essa abordagem facilita a construção de grafos complexos, melhora a clareza do código e permite rápida substituição ou reconfiguração de trechos funcionais inteiros. *Super-instruções* especulativas, por sua vez, são utilizadas em contextos onde a execução pode ser revertida, caso determinada condição não seja satisfeita, aumentando as possibilidades de paralelismo otimizável pela arquitetura (ALVES, 2014).

### 2.9.3 *THLL* (*TALM High-Level Language*)

A linguagem *THLL* (*TALM High-Level Language*) é um subconjunto da linguagem *C*, estendida por diretivas que indicam as entradas e saídas de cada *super-instrução*. Essa linguagem foi projetada para permitir que desenvolvedores expressem explicitamente a granularidade computacional dentro do modelo *TALM*, facilitando o mapeamento entre funções tradicionais e nós do grafo *Dataflow* (MARZULO et al., 2011).

Cada *super-instrução* em *THLL* corresponde a uma função que pode ser executada de forma independente, recebendo dados como entrada e produzindo resultados como saída. As diretivas `treb_superinput` e `treb_superoutput` são utilizadas para marcar variáveis de entrada e saída, respectivamente.

O exemplo a seguir ilustra uma *super-instrução* simples de soma em *THLL*, escrita como uma função *C* anotada:

Listagem 2.1 – Exemplo de super-instrução *THLL* para soma

```
1 treb_superinput(int a, int b);
2 treb_superoutput(int result);
3
4 void super_add() {
5     result = a + b;
6 }
```

Para compilar programas escritos em *THLL*, utiliza-se o **Couillard**, um compilador desenvolvido especificamente para a linguagem. O **Couillard** transforma o código anotado em três artefatos distintos: (i) código *assembly TALM* correspondente à *super-instrução*; (ii) um arquivo de metadados descrevendo sua posição e conexões no grafo de execução; e (iii) uma biblioteca compartilhada (`.so`) com a versão compilada da função.

Esses artefatos são posteriormente utilizados pelo ambiente de execução **Trebuchet**, que interpreta os grafos *TALM* e coordena a execução paralela das *super-instruções* com base nas dependências explicitadas. Essa separação entre código computacional e estrutura de controle permite paralelização eficiente, testes modulares e maior transparência na análise do comportamento do programa.

## 2.10 Tecnologias empregadas

A implementação do compilador proposto envolve diversas tecnologias, escolhidas por sua adequação ao modelo *Dataflow*, facilidade de integração e suporte ao desenvolvimento modular. A seguir, descrevem-se brevemente os papéis de cada uma:

- ***Haskell*** (AL., 2010): linguagem funcional com tipagem forte e sem efeitos colaterais. Utilizada para a entrada dos programas a serem compilados.
- ***Alex*** (COMMUNITY, 2023b): gerador de analisadores léxicos em *Haskell*, responsável pela identificação dos símbolos da linguagem.
- ***Happy*** (COMMUNITY, 2023c): gerador de analisadores sintáticos em *Haskell*, usado para validar a estrutura dos programas segundo a gramática.
- ***GHC*** (TEAM, 2023): compilador oficial de *Haskell*. Serve como referência de execução para verificação funcional dos resultados.
- ***C*** (KERNIGHAN; RITCHIE, 1988): linguagem imperativa utilizada no núcleo do compilador, para a verificação semântica e construção dos grafos.

- ***Python*** (FOUNDATION, 2023b): linguagem usada em *scripts* de teste, análise e geração de gráficos. Utiliza as bibliotecas ***networkx*** (HAGBERG; SCHULT; SWART, 2008), ***pandas*** (MCKINNEY, 2010) e ***matplotlib*** (HUNTER, 2007).
- ***Graphviz*** (LABS, 2023): ferramenta de visualização gráfica, usada para gerar representações dos grafos produzidos.
- ***TikZ*** (TANTAU, 2021): biblioteca do LaTeX para desenho de diagramas, usada em figuras explicativas.
- ***THLL*** (MARZULO et al., 2011): linguagem intermediária baseada em *C*, voltada à execução no modelo *Dataflow*.
- ***Couillard*** (MARZULO et al., 2011): compilador da linguagem *THLL*, que gera o código *TALM* e os metadados do grafo.
- ***Trebuchet*** (ALVES et al., 2011): interpretador de grafos *Dataflow*, que executa os programas paralelamente em máquinas *multicore*.
- ***Make*** (FOUNDATION, 2023a): ferramenta de automação de tarefas, responsável pela compilação e execução organizada dos componentes.
- ***Git*** (COMMUNITY, 2023a): sistema de controle de versões utilizado durante o desenvolvimento e organização do projeto.

# Referências

AHO, A. V. et al. *Compilers: Principles, Techniques, and Tools*. 2. ed. Boston: Pearson, 2006. Citado 2 vezes nas páginas 9 e 15.

AL., S. P. J. et. *The Haskell 2010 Language Report*. 2010. Language specification. Disponível em: <https://www.haskell.org/onlinereport/haskell2010/>. Citado 2 vezes nas páginas 3 e 20.

ALVES, T. A. d. O. *Dataflow Execution for Reliability and Performance on Current Hardware*. Tese (Tese (Doutorado)) — Programa de Engenharia de Sistemas e Computação, COPPE, UFRJ, Rio de Janeiro, 2014. Orientador: Felipe Maia Galvão França. Citado 3 vezes nas páginas 17, 18 e 19.

ALVES, T. A. O. et al. Trebuchet: Exploring tlp with dataflow virtualisation. *International Journal of High Performance Systems Architecture*, v. 3, n. 2/3, p. 137–148, 2011. ISSN 1751-6528. Citado na página 21.

COMMUNITY, G. *Git Documentation*. 2023. Disponível em: <https://git-scm.com/doc>. Citado na página 21.

COMMUNITY, T. H. *Alex: A Lexical Analyzer Generator for Haskell*. [S.l.], 2023. Ferramenta de análise léxica. Disponível em: <https://www.haskell.org/alex/>. Citado 2 vezes nas páginas 11 e 20.

COMMUNITY, T. H. *Happy: The Parser Generator for Haskell*. [S.l.], 2023. Gerador de analisador sintático LALR(1). Disponível em: <https://www.haskell.org/happy/>. Citado 3 vezes nas páginas 12, 14 e 20.

DENNIS, J. B.; MISUNAS, D. P. Data flow schemas. In: IEEE COMPUTER SOCIETY. *Proceedings of the 3rd Annual Symposium on Computer Architecture*. [S.l.], 1975. p. 9–14. Citado na página 16.

FOUNDATION, F. S. *GNU Make Manual*. 2023. Disponível em: <https://www.gnu.org/software/make/manual/>. Citado na página 21.

FOUNDATION, P. S. *Python 3.11 Documentation*. 2023. Disponível em: <https://docs.python.org/3/>. Citado na página 21.

HAGBERG, A. A.; SCHULT, D. A.; SWART, P. J. *NetworkX: Network Analysis in Python*. 2008. Proceedings of the 7th Python in Science Conference (SciPy 2008). Citado na página 21.

HOPCROFT, J. E.; MOTWANI, R.; ULLMAN, J. D. *Introduction to Automata Theory, Languages, and Computation*. 3. ed. Boston: Pearson, 2007. Citado 2 vezes nas páginas 9 e 10.

HUNTER, J. D. *Matplotlib: A 2D Graphics Environment*. 2007. 90–95 p. Citado na página 21.

KERNIGHAN, B. W.; RITCHIE, D. M. *The C Programming Language*. [S.l.]: Prentice Hall, 1988. Segunda Edição. Citado na página 20.

LABS, A. R. *Graphviz Documentation*. [S.l.], 2023. Disponível em: <https://graphviz.org/documentation/>. Citado na página 21.

MARZULO, L. A. J. et al. TALM: A hybrid execution model with distributed speculation support. In: *Proceedings of the International Symposium on Computer Architecture and High Performance Computing Workshops (CAHPCW)*. [S.l.: s.n.], 2010. p. 31–36. Citado na página 3.

MARZULO, L. A. J. et al. Couillard: Parallel programming via coarse-grained data-flow compilation. *CoRR*, abs/1109.4925, 2011. Disponível em: <http://dblp.uni-trier.de/db/journals/corr/corr1109.html#abs-1109-4925>. Citado 2 vezes nas páginas 19 e 21.

MCKINNEY, W. *Data Structures for Statistical Computing in Python*. 2010. Citado na página 21.

NIELSON, F.; NIELSON, H. R. *Semantics with Applications: An Appetizer*. [S.l.]: Springer, 2007. 2nd edition. Citado na página 12.

RABIN, M. O.; SCOTT, D. Finite automata and their decision problems. *IBM Journal of Research and Development*, IBM, v. 3, n. 2, p. 114–125, 1959. Citado na página 11.

TANTAU, T. *The TikZ and PGF Manual*. [S.l.]: TeX Users Group, 2021. Version 3.1.9a. Citado na página 21.

TEAM, G. *The Glasgow Haskell Compiler*. [S.l.], 2023. Compilador oficial da linguagem Haskell. Disponível em: <https://www.haskell.org/ghc/>. Citado na página 20.

WIRTH, N. What can we do about the unnecessary diversity of notation for syntactic definitions? *Communications of the ACM*, v. 20, n. 11, p. 822–823, 1977. Citado na página 10.