UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS DE RUSSAS

CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ALYSSON LUCAS BRAGA PINHEIRO

GRAMÁTICA LIVRE DE CONTEXTO PARA TIPAGEM DE GRAFOS EM LINGUAGEM DE PROGRAMAÇÃO

RUSSAS

2025

ALYSSON LUCAS BRAGA PINHEIRO

GRAMÁTICA LIVRE DE CONTEXTO PARA TIPAGEM DE GRAFOS EM LINGUAGEM DE PROGRAMAÇÃO

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação do Campus de Russas da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Ciência da Computação.

Orientador: Prof. Dr. Cenez Araújo de Rezende.

RUSSAS

2025

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

P718g Pinheiro, Alysson.
Gramática livre de contexto para tipagem de grafos em linguagem de programação / Alysson Pinheiro. – 2025.
63 f.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Russas, Curso de Ciência da Computação, Russas, 2025.
Orientação: Prof. Dr. Cenez Araújo de Rezende.

1. gramática livre de contexto. 2. ipagem de grafos. 3. sistemas de tipos. 4. antlr4. 5. verificação semântica. I. Título.

CDD 005

ALYSSON LUCAS BRAGA PINHEIRO

GRAMÁTICA LIVRE DE CONTEXTO PARA TIPAGEM DE GRAFOS EM LINGUAGEM DE PROGRAMAÇÃO

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação do Campus de Russas da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Ciência da Computação.

Aprovada em: 30/07/2025.

BANCA EXAMINADORA

Prof. Dr. Cenez Araújo de Rezende (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. Eurinardo Rodrigues Costa
Universidade Federal do Ceará (UFC)

Prof. Dr. Alexandre Matos Arruda
Universidade Federal do Ceará (UFC)

À minha família, por toda confiança e apoio. Mãe, Socorro Morais, sua dedicação e cuidado foram essenciais nos momentos difíceis — nunca esquecerei. À minha irmã, Layza Nascimento, pelo companheirismo e leveza nos dias mais pesados.

AGRADECIMENTOS

Agradeço, primeiramente, ao Prof. Dr. Cenez Araújo de Rezende, meu orientador e professor na disciplina de Compiladores, cuja orientação criteriosa e inspiradora foi essencial para a realização deste projeto. Suas aulas, sua abordagem didática e seu vasto conhecimento foram fundamentais na construção deste trabalho. Foi, sobretudo, sua dedicação que despertou e alimentou minha curiosidade pelo universo das gramáticas formais.

Expresso minha sincera gratidão aos meus amigos discentes do Laboratório de Tecnologias Inovadoras — Samuel Lima, Mateus Daniel, Naum Josafá, Paulo Henrique, Pedro Ítalo e Yan Rodrigues — pelo apoio constante e incentivo ao longo da graduação. Em uma cidade nova e em um ambiente universitário repleto de desafios, vocês foram pilares na construção de uma rede de amizade que tornou esse percurso mais leve, significativo e inesquecível.

Aos meus pais e à minha irmã, agradeço pelo amor incondicional, pelo apoio contínuo e pela confiança em cada etapa da minha jornada. À minha mãe, em especial, sou grato pelos ensinamentos que me guiaram a reconhecer aquilo pelo qual realmente vale a pena lutar.

Estendo meus agradecimentos a todos os professores que contribuíram para a minha formação, não apenas transmitindo conhecimento, mas também sendo exemplos de ética, humanidade e dedicação. Entre eles, agradeço de forma especial aos professores da banca: Prof. Dr. Alexandre Matos Arruda — também coordenador do Laboratório de Tecnologias Inovadoras, cuja estrutura e visão tornaram este ambiente de pesquisa ainda mais fértil — e ao Prof. Dr. Eurinardo Rodrigues Costa, cujo ensino em Linguagens Formais e Autômatos foi essencial para consolidar os fundamentos teóricos que sustentam este trabalho.

A todos que, de forma direta ou indireta, contribuíram para essa trajetória, meu mais sincero obrigado.

RESUMO

Este Trabalho de Conclusão de Curso investiga, formaliza e valida uma *Gramática Livre de Contexto* (GLC) dedicada à tipagem estática de grafos, com vistas a suprir lacunas metodológicas em linguagens formais e compiladores orientados a grafos. A gramática foi especificada em *Extended Backus–Naur Form* (EBNF), convertida para *Another Tool for Language Recognition* (ANTLR 4) e acompanhada de uma semântica executável em Python, implementada via *visitor pattern*. Essa infraestrutura permite, em tempo de compilação, detectar usos de vértices e arestas sem declaração prévia, violação de cardinalidade, incompatibilidades típicas de multigrafos direcionados e ausência de atributos obrigatórios, além de identificar ciclos ou identificadores duplicados quando tais restrições são impostas. Do ponto de vista acadêmico, o trabalho contribui ao oferecer uma base teórica rigorosa que pode ser empregada tanto no ensino de Teoria de Grafos quanto em disciplinas de Compiladores e Linguagens Formais, servindo como caso de estudo completo — da definição lexical e sintática ao ciclo de análise semântica — para estudantes e pesquisadores. Ademais, a proposta estabelece um *framework* extensível que pode ser reutilizado em futuras linhas de pesquisa sobre sistemas de tipos para grafos, transformação de grafos baseada em regras, integração de grafos a DSLs e geração automática de código. A validação com um conjunto de onze casos de teste sintéticos evidenciou cobertura integral das regras gramaticais, reforçando a robustez do método e seu potencial como artefato de pesquisa replicável.

Palavras-chave: gramática livre de contexto; tipagem de grafos; sistemas de tipos; antlr4; verificação semântica.

ABSTRACT

This work proposes the development of a context-free grammar for graph typing, aiming to formalize the characteristics of these structures in computational systems. The increasing complexity of graph-based systems, used in domains such as social networks, logistics systems, and data science, demands tools that ensure consistency and safety in handling these structures. The proposed grammar defines types for vertices and edges, as well as establishes composition rules and semantic validation. The project focuses on planning the grammar and modeling an interpreter capable of validating and inferring properties of graphs. It is expected that the future implementation of this proposal (in a subsequent thesis) will result in a formal and efficient approach to graph typing, contributing to the reliability of computational systems that rely on these structures.

Keywords: context-free grammar; graph typing; type systems; antlr4; semantic validation.

LISTA DE QUADROS

LISTA DE CÓDIGOS-FONTE

LISTA DE ABREVIATURAS E SIGLAS

ANTLR 4	*Another Tool for Language Recognition*
API	*Application Programming Interface* (Interface de Programação de Aplicações)
AST	*Abstract Syntax Tree* (Árvore Sintática Abstrata)
BFS	*Breadth-First Search* (Busca em Largura)
BNF	*Backus–Naur Form*
DFS	*Depth-First Search*
DOT	Linguagem de descrição de grafos do Graphviz
DSL	*Domain-Specific Language* (Linguagem de Domínio Específico)
EBNF	*Extended Backus–Naur Form*
GLC	*Gramática Livre de Contexto*
GNU Bison	*GNU Bison Parser Generator* (Gerador de Analisadores Bison do Projeto GNU)
IDE	*Integrated Development Environment* (Ambiente de Desenvolvimento Integrado)
JSON	*JavaScript Object Notation*

LL(1)	Parser LL com *look-ahead* de 1 símbolo
LR	*Left-to-Right, Rightmost derivation* (Análise Sintática Esquerda-para-Direita com Derivação mais à Direita)
SVG	*Scalable Vector Graphics*
UML	*Unified Modeling Language*
XML	*Extensible Markup Language*

LISTA DE SÍMBOLOS

G	Grafo representado como um par ordenado $G = (V, E)$.
V	Conjunto de vértices do grafo.
E	Conjunto de arestas do grafo.
v_i	Vértice específico, com i como identificador único.
T_V	Tipo associado a um vértice.
A_V	Conjunto de atributos de um vértice.
e_i	Aresta específica, com i como identificador único.
T_E	Tipo associado a uma aresta.
A_E	Conjunto de atributos de uma aresta.
v_{origem}	Vértice de origem de uma aresta.
v_{destino}	Vértice de destino de uma aresta.
Γ	Contexto de tipagem.
$\rightarrow$	Representação de mapeamento ou direção.
$\times$	Produto cartesiano entre conjuntos.
$\rightsquigarrow$	Implicação de compatibilidade (“resulta em”).
$*$	Fecho de Kleene (zero ou mais repetições).
$\mathscr{L}$	Linguagem formal gerada pela gramática.
Σ	Alfabeto de símbolos terminais da gramática.
$\mathscr{G}$	Gramática formal $\mathscr{G} = (N, \Sigma, P, S)$.
N	Conjunto de símbolos não-terminais da gramática.
P	Conjunto de regras de produção da gramática.
S	Símbolo inicial da gramática.
λ	Função ou expressão anônima.
V_T	Conjunto de *tipos de vértices*
E_T	Conjunto de *tipos de arestas*
Card	Anotação de cardinalidade mínima–máxima $[\, m..n \,]$

SUMÁRIO

1 INTRODUÇÃO

Os grafos são estruturas matemáticas fundamentais para a modelagem de relações entre entidades e têm sido amplamente utilizados em domínios como Ciência da Computação, engenharia, redes sociais, logística e biologia computacional West (2001) e Bondy e Murty (1976). Essa ampla aplicabilidade decorre de sua capacidade de representar sistemas complexos por meio de uma estrutura relacional abstrata, que pode modelar desde rotas de transporte e fluxos de dados até interações sociais e biológicas.

Com o avanço das aplicações baseadas em grafos, cresce a necessidade de mecanismos que assegurem a consistência, segurança e reutilização semântica dessas estruturas em sistemas computacionais Cormen *et al.* (2009). Nesse contexto, este trabalho propõe o desenvolvimento de uma GLC dedicada à tipagem de grafos, cuja função é conferir uma semântica formal aos seus elementos — vértices, arestas e atributos — possibilitando a validação sintática e semântica, a inferência de tipos de propriedades e a integração com compiladores Pierce (2002).

Diferentemente de tipos estruturados clássicos como registros e classes, que assumem uma organização hierárquica, os grafos modelam relações arbitrárias e dinâmicas. Assim, por exemplo, em um grafo representando uma rede social, seria possível definir vértices dos tipos "Usuário", "Empresa"e "Evento", e arestas como "Segue", "TrabalhaEm"ou "Participa", cujas conexões seriam validadas automaticamente pela gramática segundo regras de tipagem declaradas previamente. Tal mecanismo favorece a segurança semântica, reduz erros lógicos e aprimora a robustez de sistemas complexos.

A principal motivação deste trabalho é propor uma gramática de tipagem de grafos que possa contribuir para uma linguagem de programação voltada ao ensino e pesquisa, como em Teoria dos Grafos, na qual os grafos sejam tratados como entidade de primeira classe. Essa abordagem oferece uma ferramenta didática inovadora que permite ao discente experimentar diretamente os conceitos teóricos de grafos direcionados, ponderados, regras de conectividade, subgrafos e inferência de tipos. A interação prática com grafos fortemente tipados promove um aprendizado mais ativo, intuitivo e contextualizado.

Apesar da difusão de sistemas de tipos em linguagens de programação modernas, ainda há uma lacuna quanto ao suporte nativo à tipagem estática de grafos com regras semânticas formais. A gramática proposta neste trabalho visa preencher essa lacuna, aliando a expressividade dos grafos à segurança de um sistema de tipos robusto e extensível Milner (1978) e Hindley (1969).

Espera-se, portanto, que esta gramática constitua uma base formal e reutilizável tanto para o desenvolvimento de ferramentas computacionais quanto para a criação de uma linguagem acadêmica especializada, incentivando a inovação didática e a evolução teórica na área de grafos Diestel (2017).

Além desta Introdução, o **Capítulo 2** apresenta a *Fundamentação Teórica*, revisitando os principais conceitos de Grafos, Sistemas de Tipos e Gramáticas em Linguagens Formais que embasam a proposta. O **Capítulo 3** discute os *Trabalhos Relacionados*, situando esta pesquisa frente às soluções mais relevantes da literatura. No **Capítulo 4**, a *Metodologia* é detalhada, desde a definição dos elementos fundamentais até a formalização da gramática em EBNF e o pipeline de validação semântica. O **Capítulo 5** apresenta os *Resultados*, destacando a gramática final, os testes executados e as contribuições obtidas. Por fim, o **Capítulo 6** expõe as *Conclusões e Trabalhos Futuros*, sintetizando as principais realizações, limitações e perspectivas de continuidade desta pesquisa.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta os principais fundamentos teóricos que sustentam a proposta desta pesquisa. Primeiramente, revisitam-se os conceitos essenciais da *Teoria dos Grafos*, abordando sua origem histórica, classificações estruturais, representações computacionais e algoritmos basilares. Em seguida, discute-se o arcabouço de *Gramáticas em Linguagens Formais* e de sistema de tipos, ressaltando como tais ferramentas fornecem o rigor necessário para especificar e validar estruturas relacionais complexas. Por fim, introduz-se a *Gramática Livre de Contexto para Tipagem de Grafos* proposta, destacando seus recursos expressivos, motivações didáticas e benefícios práticos. Essa fundamentação estabelece o pano de fundo conceitual indispensável para compreender, no restante do trabalho, a construção, a aplicação e a avaliação da gramática desenvolvida.

2.1 Grafo

2.1.1 Origem Histórica e Definições Básicas

O conceito de grafo surgiu formalmente com o trabalho de Euler sobre o problema das *Pontes de Königsberg* em 1736, considerado o marco inaugural da combinatória moderna Euler (1736). A generalização e sistematização desses estudos deu origem à *Teoria dos Grafos*, área da Matemática Discreta dedicada ao estudo de estruturas compostas por vértices (ou nós) e Arestas (ou arcos), representadas formalmente como um par ordenado $G = (V, E)$ Harary (1969) e West (2001). Cada vértice $v \in V$ modela uma entidade abstrata, enquanto cada aresta $e = (v_i, v_j) \in E$, onde $v_i, v_j \in V$, modela uma relação binária entre duas entidades.

2.1.2 Classificações Estruturais

- **Direcionamento.** Se as Arestas possuem orientação, o grafo é *direcionado* (digrafo); caso contrário, é *não-direcionado*. A distinção determina se cada aresta é ordenado ou não. grafos direcionados são essenciais em modelagem de fluxos, precedências e autômatos Diestel (2017).
- **Multiplicidade e laços.** Em grafos simples não há laços ($v_i = v_j$) nem múltiplas Arestas paralelas entre dois vértices, enquanto multigrafos permitem ambas as situações Bondy e Murty (1976). multigrafos são comuns em redes de transporte, onde diferentes rotas

conectam as mesmas cidades.

- **Ponderação.** Um grafo ponderado associa um peso $w : E \to \mathbb{R}^+$ a cada Aresta, permitindo expressar custo, capacidade ou distância — condição indispensável a algoritmos de caminho mínimo, como Dijkstra e Bellman–Ford Cormen *et al.* (2009).
- **Conectividade.** Um é grafo conexo se existe pelo menos um caminho entre quaisquer $v_i, v_j \in V$; caso contrário, apresenta componentes conexas disjuntas. A conectividade é verificada em $O(|V| + |E|)$ via *Depth-First Search* (DFS) Tarjan (1972).
- **Planaridade.** grafos planares podem ser desenhados no plano sem cruzamento de Arestas; o Teorema de Kuratowski caracteriza essa propriedade via subgrafos homeomorfo (em grafos) a K_5 ou $K_{3,3}$ Gross e Yellen (2014).

2.1.3 Representações Computacionais

Para uso algorítmico, um grafo $G = (V, E)$ pode ser tipicamente codificados como:

1. **Matrizes de adjacência:** estrutura $|V| \times |V|$ densa, com ponto flutuante para grafos completos; acesso $O(1)$ para consulta de Aresta, porém $O(|V|^2)$ de memória.
2. **Listas de adjacência:** vetor de listas $A[v]$ contendo vizinhos de v; requer $O(|V| + |E|)$ de memória e é a representação preferencial para grafos esparsos Cormen *et al.* (2009).
3. **Listas de incidência** ou **edge-list**, útil em algoritmos baseados em ordenação de Arestas, como Kruskal.

Essas representações afetam a complexidade de muitos algoritmos tais como: travessal, cobertura mínima, fluxo máximo e isomorfismo.

2.1.4 Algoritmos Fundamentais

- ***Breadth-First Search* (Busca em Largura) (BFS).** Explora o grafo camada a camada, identificando o menor número de Arestas num caminho entre um vértice fonte e os demais; base para provas de bipartição, cálculo de diâmetro e construção de árvores geradoras Cormen *et al.* (2009).
- **Depth-First Search (Busca em Profundidade) DFS.** Fundamenta algoritmos de detecção de ciclos, ordenação topológica e componentes fortemente conexas (Kosaraju, Tarjan).
- **Dijkstra e A*.** *Dijkstra* resolve o problema de caminhos mínimos de fonte única em grafos ponderados com pesos não negativos (com fila de prioridade: $O((|V| + |E|) \log |V|)$). A* generaliza Dijkstra com busca informada, priorizando por $g(n) + h(n)$; com heurística

admissível/consistente, mantém otimalidade Hart *et al.* (1968).

Esses algoritmos ilustram como a estrutura de grafo é explorada computacionalmente, reforçando a necessidade de representações e tipagens rigorosas para garantir corretude.

2.2 Gramática para Tipagem de Grafos

2.2.1 Gramáticas em Linguagens Formais

No contexto de linguagens formais, uma gramática é definida como uma 4-upla $GM = (N, \Sigma, P, S)$, em que N é um conjunto finito de *símbolos não-terminais*, Σ é um conjunto de *terminais*, P é um conjunto finito de *regras de produção* e $S \in N$ é o *símbolo inicial* Hopcroft *et al.* (2006). As *GLCs* caracterizam a classe $\mathscr{L}_{\mathrm{CF}}$ de linguagens reconhecíveis por autômatos de pilha determinísticos.

2.2.2 Tipagem Estática e Sistema de Tipos

Um *sistema de tipos* associa a cada expressão de um programa um construtor semântico capaz de restringir operações inválidas (*soundness*) e, idealmente, de não rejeitar programas corretos (*completeness*) Pierce (2002). Extender o sistema de tipos às instâncias de grafos implica:

- atribuir **conjuntos de tipos** aos vértices e às arestas: V_T (tipos de vértice) e E_T (tipos de aresta), o que permite verificar propriedades de domínio (ex.: `Aluno` conecta-se apenas a `Curso`);
- estabelecer **regras de formação** P que descrevem as conexões válidas entre tipos, por exemplo como triplas $P \subseteq V_T \times E_T \times V_T$ do tipo (τ_s, τ_e, τ_t) (origem, aresta, destino);
- manter um **contexto de tipagem** Γ (um mapeamento de identificadores para tipos), e usar a notação de **julgamento de tipagem** $\Gamma \vdash e : T$, lida como: "sob as hipóteses de Γ, a expressão e tem tipo T", durante a análise semântica Milner (1978).

2.2.3 GLC para Tipagem de Grafos

A gramática proposta neste trabalho utiliza EBNF para modelar estruturas tipadas de grafos. A seguir, apresenta-se um extrato simplificado, adequado a leitores iniciantes:

$$S \rightarrow \texttt{GraphDecl}^{+}$$

$$\texttt{GraphDecl} \rightarrow \texttt{graph}\ ID\ \{\ \texttt{TypeSec}\ \texttt{VertSec}\ \texttt{EdgeSec}\ \}$$

$$\texttt{TypeSec} \rightarrow \texttt{types}\ \{\ \texttt{VTypeDecl}^{*}\ \texttt{ETypeDecl}^{*}\ \}$$

$$\texttt{VTypeDecl} \rightarrow \texttt{vertex}\ ID\ [\texttt{extends}\ ID]\ [\texttt{Card}]\ [\texttt{AttrBlock}]$$

$$\texttt{ETypeDecl} \rightarrow \texttt{edge}\ ID\ (ID, ID)\ [\texttt{directed} \mid \texttt{undirected}]\ [\texttt{Card}]\ [\texttt{AttrBlock}]$$

Onde `Card` descreve a Cardinalidade $[m..n]$ e `AttrBlock` encapsula atributos tipados. A escolha de GLC permite:

1. **Parsing determinístico.** Ferramentas Parser LL com *look-ahead* de 1 símbolo (LL(1)), como ANTLR 4, geram analisadores compatíveis com compiladores educacionais.
2. **Validação semântica incremental.** Após a derivação sintática, um *visitor pattern* percorre a árvore anotando tipos e verificando restrições declaradas em P.
3. **Extensibilidade modular.** Novos tipos, atributos ou restrições podem ser adicionados sem modificar o núcleo da gramática, respeitando o princípio de *open-closed*.

2.2.4 Exemplo Ilustrativo de código

O código-fonte completo que exemplifica a declaração de um grafo de rotas aéreas encontra-se no Apêndice A. A listagem ilustra (i) herança de vértices (`Hub extends Airport`); (ii) atributos obrigatórios (`code`, `city`); e (iii) arestas direcionadas (`Flight`), todos validados pela gramática proposta. Durante a fase semântica, o analisador assegura que `Flight` conecta unicamente vértices do tipo `Airport` (ou seus subtipos), reportando erro caso contrário.

2.2.5 Recursos Expressivos da Gramática Proposta

– **Herança (entre tipos) entre tipos de vértices** (`extends`) — um tipo especializado reaproveita atributos e restrições do supertipo, promovendo reutilização Cardelli (1985).
– **Cardinalidade** $[\,m..n\,]$ — especifica o número mínimo m e máximo n de ocorrências permitidas, inspirado na notação *Unified Modeling Language* (UML) Rumbaugh *et al.* (2004).
– **Atributos tipados, obrigatórios e com valor-padrão** — cada elemento declara `atributo:Tipo` seguido de `required` ou `default = valor`, evitando *nullability* Pierce (2002).

– **Arestas direcionadas** (`directed`) *ou* **não** (`undirected`) — definem orientação semântica explícita; a escolha impacta algoritmos de travessia Diestel (2017).
– **Múltiplos grafos no mesmo programa** — várias declarações `graph` compartilham arquivo, mas mantêm *namespace*s isolados Mens e Gorp (2006).

Exemplo completo dos recursos

Para uma demonstração integrada de herança, cardinalidade, atributos opcionais, múltiplos tipos de aresta e validação semântica, consulte o Apêndice B, que apresenta o grafo `socialNet`. Nessa listagem observa-se, por exemplo, a herança de `Student` a `Person`, arestas simétricas (`Knows`) e assimétricas (`Follows`), além do uso de valores-padrão de atributos.

Validação semântica do exemplo

– O vértice `b` herda atributos obrigatórios de `Person`.
– A restrição `[1..*]` é satisfeita (há ao menos um `Person`).
– O atributo `age` usa o valor padrão `18` quando omitido.
– `Follows` é assimétrico; `Knows` é simétrico.
– Os grafos `socialNet` e `logistics` têm *namespace*s distintos.

2.2.6 Benefícios Didáticos e Práticos

– **Segurança estática.** Erros de modelagem são detectados em tempo de compilação, antecipando inconsistências lógicas difíceis de depurar em tempo de execução.
– **Abstração de alto nível.** grafos deixam de ser simulados por matrizes ou listas; passam a ser entidade de primeira classes, capazes de receber operações semânticas próprias (subgrafo, contração, *pattern-matching*).
– **Integração com *Domain-Specific Language* (Linguagem de Domínio Específico)s (DSLs).** A gramática pode ser embutida em linguagens específicas de domínio, oferecendo suporte nativo a problemas de roteamento, dependências ou fluxos de dados.
– **Ferramenta de ensino.** Em ambientes acadêmicos, fornece um laboratório seguro onde estudantes experimentam transformações de grafos com garantias formais de corretude.

A combinação de Teoria dos Grafos clássica e Sistemas de Tipos modernos resulta, portanto, em um arcabouço robusto para modelagem, validação e manipulação de estruturas relacionais complexas, alinhado às demandas de aplicações contemporâneas em Ciência da Computação.

3 TRABALHOS RELACIONADOS

Este capítulo revisita as principais pesquisas que abordam a tipagem e a transformação de grafos, situando o estado-da-arte em relação à proposta deste trabalho. Inicia-se com um *Panorama das Pesquisas*, que delineia as linhas teóricas e aplicadas mais relevantes. Em seguida, cada subseção analisa criticamente estudos representativos — desde abordagens baseadas em *type graphs*, sistemas de tipos para transformações, formalismos algébricos e bancos de dados orientados a grafos, até propostas acadêmicas emergentes. Por fim, apresenta-se uma tabela comparativa que sintetiza os aspectos avaliados e evidencia como a gramática livre de contexto aqui desenvolvida preenche lacunas identificadas na literatura. Essa revisão fundamenta e justifica as escolhas metodológicas adotadas ao longo do TCC.

3.1 Panorama das Pesquisas

A tipagem formal de grafos tem sido explorada em diversas linhas de pesquisa, sobretudo no contexto da transformação de grafos, especificações baseadas em regras e engenharia de software model-driven. No entanto, permanece uma lacuna metodológica significativa quanto à construção de gramáticas formais — especialmente gramáticas livres de contexto (GLCs) — voltadas à definição textual e tipada de grafos como estruturas de primeira classe em linguagens de programação. Esta seção examina trabalhos de base teórica e técnica que influenciam a proposta deste TCC, apresentando suas metodologias, restrições e distinções em relação ao sistema gramatical aqui desenvolvido.

Specifying Graph Languages with Type Graphs
(Corradini et al., 1997)

Corradini et al. Corradini *et al.* (1997) introduzem os *type graphs* como construtos formais para definição de linguagens de grafos. A proposta parte do paradigma categórico de grafos e homomorfismos, onde grafos válidos são definidos como instâncias de um grafo-tipo, respeitando restrições estruturais como tipos de vértices e aridade de arestas.

O método adotado consiste na composição de morfismos entre grafos usando técnicas de categorização em diagramas comutativos. As operações de substituição, extensão e restrição são validadas através de preservação da tipagem e consistência estrutural. O trabalho enfatiza a modelagem em ambientes visuais, com foco em editores gráficos de modelagem e Engenharia

de Software.

Contudo, a ausência de uma linguagem textual declarativa impede a aplicação direta desse modelo em linguagens de programação, dificultando sua integração com compiladores, interpretadores ou ambientes de ensino. A proposta deste TCC avança ao propor uma gramática formal textual — escrita em EBNF e implementada em ANTLR — que permite definir grafos com tipos e atributos diretamente na linguagem, realizando validações sintáticas e semânticas durante o parsing.

Type Systems for Graph Transformation Systems
(Heckel e Taentzer, 2006)

Heckel e Taentzer Heckel e Taentzer (2006) apresentam um sistema de tipos para transformação de grafos baseado em regras formais de correspondência e preservação de propriedades. A abordagem parte da definição de grafos-tipo e operações de reescrita, propondo um mecanismo para garantir que transformações (como inserção, exclusão e substituição de subgrafos) não violem restrições semânticas estabelecidas previamente.

O método baseia-se na identificação de pré-condições e pós-condições para regras de transformação, com foco na consistência de tipagem durante a aplicação dessas regras. O sistema admite subtipagem e inferência parcial de tipos, e é validado por meio de provas formais e implementação prototípica em ferramentas baseadas em visualização de grafos.

Apesar de sua robustez teórica, a proposta não contempla uma gramática formal textual nem oferece suporte a linguagens com análise sintática tradicional (ex: análise LL ou LR). Em contrapartida, a gramática desenvolvida neste TCC é explicitamente textual, escrita em EBNF, permitindo que grafos sejam definidos como declarações tipadas, com validação semântica embutida e suporte a atributos, cardinalidade, herança e multigrafos, diretamente integráveis a compiladores.

Graph Transformation: A Specification Technique and Its Applications
(Ehrig et al., 1991)

Ehrig et al. Ehrig *et al.* (1991) sistematizam a teoria de transformação de grafos com um enfoque em especificação algébrica. A técnica apresentada define regras formais para reescrita de grafos, caracterizadas por tripletos (L, K, R), onde L representa o subgrafo a ser substituído, R o grafo resultante, e K os elementos preservados.

A metodologia inclui o uso de condições aplicáveis (*application conditions*), controle de conflitos e critérios de aplicabilidade baseados em homomorfismos. A proposta é eficaz para transformação computacional, e influenciou diversos formalismos como AGG, GROOVE e Henshin.

Entretanto, sua ênfase é prescritiva e operacional, voltada à reescrita e execução, não abordando a tipagem como elemento central. A estrutura tipada, quando presente, é embutida nos grafos, e não gerada por gramática textual formal com regras sintáticas explícitas. Diferente disso, a proposta deste TCC define uma GLC textual que permite, por exemplo, especificar que um vértice do tipo `Manager` herda atributos de `Person` e só pode conectar-se a vértices do tipo `Company`, sendo essas restrições verificadas automaticamente por meio de análise semântica estruturada.

Introduction to Automata Theory, Languages, and Computation (Hopcroft et al., 2006)

O trabalho de Hopcroft, Motwani e Ullman Hopcroft *et al.* (2006) constitui o alicerce teórico de linguagens formais, autômatos e gramáticas. A obra cobre com profundidade a definição de GLCs, suas propriedades de derivação, algoritmos de normalização (ex: forma normal de Chomsky) e técnicas de análise como *parsing* descendente e ascendente.

O método didático da obra oferece algoritmos detalhados para conversão entre notações (BNF e EBNF), construção de autômatos de pilha (PDA) e tabelas de *parsing*. Embora não trate de grafos, fornece o embasamento técnico fundamental que permite a este TCC construir uma gramática textual com estrutura EBNF validada via ANTLR, realizando análise sintática descendente com *look-ahead* (LL(1)).

A gramática proposta deste TCC aplica esses conceitos diretamente a grafos: define blocos para tipos de vértices e arestas, instâncias de elementos, regras de ligação e atribuição semântica, utilizando mecanismos de *parsing* formal e *visitors* para validação contextual. O uso de EBNF permite ainda transcrição automatizada para compiladores, diferentemente de abordagens exclusivamente visuais.

Webber, Robinson e Eifrem (2012) — Modelo de Grafos com Tipagem Implícita no Neo4j

Webber *et al.* (2012) apresentam o Neo4j como um sistema de gerenciamento de banco de dados orientado a grafos que adota o modelo *Property Graph*. Nesse modelo, os

vértices e arestas podem conter rótulos (*labels*) e atributos arbitrários no formato chave-valor. A linguagem de consulta Cypher é utilizada para expressar padrões de correspondência entre nós e relacionamentos de forma declarativa, permitindo navegação, filtragem e atualização das estruturas de grafo com sintaxe semelhante ao SQL.

Embora o Neo4j permita representar grafos ricos e heterogêneos, seu sistema de tipos é implícito e não formalizado. Os rótulos servem apenas como convenções semânticas, não havendo mecanismos formais para herança entre tipos, verificação de cardinalidade, obrigatoriedade de atributos ou restrições entre conexões. A validação de consistência estrutural deve ser feita manualmente, e erros de modelagem só são detectados em tempo de execução, caso afetem a lógica das consultas.

Em contraste, a proposta deste TCC define uma gramática livre de contexto textual para grafos tipados, com suporte a atributos, cardinalidade, herança e validação semântica automática. Enquanto o Neo4j prioriza flexibilidade e execução dinâmica, a abordagem deste TCC foca em rigor sintático e semântico, permitindo que estruturas de grafo sejam analisadas e validadas formalmente em tempo de compilação, com aplicação em ambientes acadêmicos, compiladores e linguagens específicas de domínio.

Jaguar-Lang: Linguagem Acadêmica com Perspectiva de Tipagem Gráfica (Rezende, 2025)

O projeto *Jaguar-Lang*, atualmente em andamento sob coordenação do Prof. Dr. Cenez Araújo de Rezende, tem como objetivo o desenvolvimento de uma linguagem de programação acadêmica com fins didáticos e de pesquisa na área de compiladores e linguagens formais. Segundo Rezende (2025), a proposta visa à construção de uma infraestrutura para experimentação de técnicas de compilação em contextos de alto desempenho (*High-Performance Computing*), com especial atenção à paralelização e distribuição computacional.

Embora o foco principal do Jaguar-Lang seja a experimentação em ambientes HPC, o projeto contempla a inclusão de recursos expressivos de linguagem voltados ao ensino de compiladores e linguagens específicas de domínio. Entre os tópicos prospectivos, está a introdução de mecanismos de representação de grafos como estruturas de primeira classe (tipos nativos e manipuláveis diretamente na linguagem), incluindo a possibilidade futura de definição de uma tipagem estática para essas estruturas no próprio núcleo da linguagem.

A relação com este TCC se estabelece na convergência conceitual: ambos visam a representação e manipulação de grafos em linguagens de programação com respaldo formal.

No entanto, enquanto o Jaguar-Lang se estrutura como uma plataforma ampla, este trabalho se concentra especificamente na definição e validação de uma gramática livre de contexto para tipagem de grafos, que poderá futuramente se integrar ou inspirar a arquitetura do Jaguar-Lang como subsistema ou módulo de linguagem fortemente tipada orientada a grafos. Como o projeto está em adamento, não será adicionado a tabela de comparação a este trabalho.

3.2 Comparação entre os Trabalhos

Quadro 1 – Comparação entre trabalhos relacionados e a proposta deste TCC

Aspecto Avaliado	**Corr.**	**Heck.**	**Hopc.**	**Neo4j**	**Este Trabalho**
Tipagem formal de vértices e arestas	✓	✓			✓
Tipagem nativa de grafos					✓
Transformação de grafos	✓	✓		✓	
Uso de GLC			✓		✓
Gramática textual com EBNF					✓
Validação semântica automática		✓			✓
Integração com compiladores			✓		✓
Atributos e herança com verificação estática		✓			✓
Proposta deste trabalho	**Uma gramática textual formal para grafos tipados com EBNF, validação semântica e sintática, herança, atributos, cardinalidade e integração com ferramentas de análise e compilação.**				

Legenda: Corr. = Corradini et al. (1997); Heck. = Heckel e Taentzer (2006); Hopc. = Hopcroft et al. (2006).
Fonte: Elaborado pelo autor.

Os trabalhos analisados fornecem contribuições valiosas em suas respectivas áreas, mas nenhum deles propõe uma solução formal textual que integre tipagem estática, análise sintática e semântica, e regras gramaticais específicas para grafos como entidades de primeira classe. A proposta deste trabalho avança ao combinar fundamentos de linguagens formais com requisitos estruturais e semânticos dos grafos, preenchendo uma lacuna metodológica na literatura atual.

4 METODOLOGIA

Este capítulo descreve, passo a passo, o processo empregado para conceber, formalizar, implementar e validar a GLC proposta para a tipagem de grafos. Inicialmente, são delineados os elementos fundamentais (vértices, Arestas, atributos e Cardinalidade) que servem de base ao modelo. Em seguida, explicita-se a escolha da notação EBNF e a estrutura modular da gramática, destacando como cada bloco (`types`, `vertices` e `edges`) contribui para a expressividade e a segurança do sistema de tipos. Na sequência, apresentam-se a especificação completa em EBNF, exemplos ilustrativos e os recursos avançados oferecidos (herança, múltiplos grafos, comentários, etc.). Por fim, detalham-se a arquitetura de validação, o conjunto de testes, as ferramentas utilizadas (ANTLR 4, Python e `anytree`) e o fluxo de execução automatizado que garante a correção sintática e semântica dos grafos definidos.

4.1 Construção da Gramática

A construção de uma GLC para *tipagem de grafos* exige uma rigorosa abordagem sistemática, de modo a permitir que grafos sejam tratados como entidades de primeira classe em linguagens de programação, com validação sintática e semântica garantidas. Esta metodologia é dividida em quatro fases:

1. **Definição dos elementos fundamentais da gramática**;
2. **Formalização das regras de formação** (em EBNF, sem recursão à esquerda, com equivalência *Backus–Naur Form* (BNF) assegurada);
3. **Elaboração da gramática completa e robusta** — com suporte a herança de tipos, cardinalidade, atributos com metadados, múltiplos grafos e direcionalidade explícita de arestas;
4. **Validação da gramática** por meio de testes formais e ferramentas automáticas.

4.2 Definição dos Elementos Fundamentais

A gramática considera os seguintes elementos estruturais de grafos:

- **Vértices** V, com:
  - Identificador único (*id*);
  - Tipo associado (T_V);
  - Atributos (A_V);

  - Cardinalidade mínima e máxima.
- **Arestas** E, com:
  - Identificador único (*id*);
  - Tipo (T_E);
  - Origem e destino ($v_{\text{origem}}, v_{\text{destino}}$);
  - Direcionalidade (direcionada ou não);
  - Cardinalidade;
  - Atributos (A_E).

4.3 Formalização da Gramática

4.3.1 Escolha da Notação EBNF

A notação EBNF (*Extended Backus-Naur Form*) foi adotada para representar a gramática com maior clareza e concisão. Diferente da BNF tradicional, a EBNF permite o uso direto de operadores de repetição e opcionais, como `*`, `+` e `?`, além de suportar agrupamentos explícitos.

Essa notação facilita:

- A escrita formal das regras de produção;
- A transcrição direta para ferramentas como ANTLR 4;
- A compreensão por parte de leitores com familiaridade com linguagens formais.

A escolha da EBNF baseia-se em sua ampla utilização em especificações formais modernas, conforme discutido em Wirth (1996) e Hopcroft *et al.* (2006).

4.3.2 Estrutura da Gramática

A gramática foi projetada com três blocos principais, seguindo a estrutura lógica dos grafos:

1. **Bloco** `types`: define os tipos de vértices e arestas. Suporta herança entre vértices (`extends`), atributos tipados com metadados (obrigatoriedade e valores padrão), além de cardinalidade explícita.
2. **Bloco** `vertices`: instancia os vértices do grafo. Cada vértice é associado a um tipo previamente declarado, e seus atributos devem obedecer ao contrato do tipo.
3. **Bloco** `edges`: define as conexões entre vértices. Cada aresta possui tipo, orientação

(via `->`), e atributos opcionais. A verificação de compatibilidade entre tipos de vértices conectados é realizada durante a análise semântica.

4.3.3 Especificação em EBNF

A seguir apresenta-se a gramática completa na notação EBNF:

O código-fonte completo da gramática em EBNF foi transferido para o Apêndice C, evitando sobrecarregar esta seção.

4.3.4 Recursos Expressivos da Gramática

A gramática suporta os seguintes recursos formais:

- Herança entre tipos de vértices com *`extends`*;
- Cardinalidade mínima e máxima com notação `[m..n]`;
- Atributos com tipo, obrigatoriedade e valor padrão;
- Arestas direcionadas (*`directed`*) ou não (*`undirected`*);
- Definição de múltiplos grafos no mesmo programa;
- Comentários em estilo `//` ou `#`.

4.3.5 Exemplo de Grafo Válido

O grafo usado como estudo de caso (`companyNetwork`) encontra-se integralmente descrito no Apêndice D.

4.3.6 Validação da Gramática

A gramática formal desenvolvida foi implementada diretamente em ANTLR 4, a partir da tradução sistemática da especificação em EBNF. O objetivo da validação foi garantir que:

- A sintaxe da linguagem fosse corretamente interpretada por analisadores lexicais e sintáticos gerados automaticamente;
- A semântica de grafos tipados fosse rigorosamente verificada via análise personalizada;
- A estrutura e coerência dos dados fossem passíveis de inspeção via geração de árvores sintáticas.

Para isso, foi desenvolvido um projeto completo disponível publicamente no GitHub[1], contendo:

- Arquivo `Graph.g4`: especificação da gramática em ANTLR 4;
- Implementações auxiliares em `Python` e `anytree` para validação semântica e renderização das árvores sintáticas;
- Script `main.py`: responsável por realizar *parsing*, análise semântica e geração de relatórios;
- Diretório `tests/`[2]: contém todos os arquivos de entrada utilizados na validação, incluindo os grafos válidos (`arquivos de texto simples`) e inválidos, cobrindo as regras gramaticais e semânticas;
- Diretório `tree_tests/`: armazena as árvores sintáticas geradas automaticamente, facilitando a depuração.
- `Resultados dos testes/`[3]: apresenta a saída gerada durante a execução dos testes, indicando quais grafos foram aceitos ou rejeitados pelo validador, de acordo com as regras gramaticais e semânticas definidas.

Essa estrutura modular permite a replicação completa dos experimentos conduzidos neste trabalho, favorecendo a reprodutibilidade acadêmica e a futura extensão da proposta por outros pesquisadores.

4.3.7 Arquitetura de Validação e Execução

A validação foi conduzida com base no projeto `GramaticaTipagemGrafosGLC`, disponível publicamente em Pinheiro (2025), cuja arquitetura encontra-se estruturada da seguinte forma:

- `Graph.g4` — arquivo principal contendo a gramática formal para grafos tipados em ANTLR 44;
- `GraphLexer.py`, `GraphParser.py`, `GraphVisitor.py` — arquivos gerados automaticamente a partir de `Graph.g4` usando ANTLR 4;
- `GraphValidator.py` — módulo de validação semântica desenvolvido manualmente, derivado do `GraphVisitor`;

[1] https://github.com/alyssonlcss/GramaticaTipagemGrafosGLC
[2] https://github.com/alyssonlcss/GramaticaTipagemGrafosGLC/tree/main/tests
[3] https://github.com/alyssonlcss/GramaticaTipagemGrafosGLC/tree/main?tab=readme-ov-file#resultados-dos-testes---tests

- `main.py` — script que percorre todos os arquivos de teste, realiza parsing, validação semântica e geração da árvore sintática;
- `tests/` — conjunto de arquivos de entrada representando grafos válidos e inválidos;
- `tree_tests/` — diretório gerado automaticamente contendo as árvores sintáticas de cada grafo analisado, em formato de texto.

4.3.8 Fluxo de Execução dos Testes

A verificação da conformidade sintática e semântica da gramática foi conduzida de forma automatizada por meio do script `main.py`, responsável por orquestrar todo o *pipeline* de análise. O fluxo completo de execução dos testes pode ser descrito conforme as etapas abaixo:

1. **Leitura dos arquivos de teste:** todos os arquivos presentes no diretório `tests/` são percorridos de forma recursiva. Cada arquivo contém uma definição textual de um ou mais grafos com diferentes configurações de tipos, atributos e conexões.
2. **Análise léxica e sintática:** utilizando o *parser* gerado pelo ANTLR 4 4 (via os arquivos `GraphLexer.py` e `GraphParser.py`), a entrada textual é transformada em uma *Abstract Syntax Tree* (Árvore Sintática Abstrata) (AST), representando a estrutura hierárquica do grafo conforme definido pela gramática.
3. **Geração da árvore sintática:** com apoio da biblioteca `anytree`, a árvore sintática gerada é renderizada e exportada para o diretório `tree_tests/`. Isso permite inspeção visual da árvore para depuração ou documentação.
4. **Análise semântica com `GraphValidator.py`:** um *visitor pattern* customizado percorre a árvore sintática verificando invariantes semânticos do grafo:
   - Se todos os tipos utilizados estão previamente declarados;
   - Se todos os atributos obrigatórios estão presentes;
   - Se os tipos de origem e destino nas arestas são compatíveis com as assinaturas;
   - Se não há identificadores duplicados para vértices ou arestas;
   - Se estruturas inválidas como ciclos proibidos (*self-loops*) ocorrem.
5. **Geração do relatório de validação:** ao final de cada verificação, o resultado é impresso no terminal. Casos válidos são confirmados com uma mensagem positiva, enquanto erros semânticos são listados de forma detalhada para cada elemento inválido detectado.
6. **Cobertura total:** esse processo é repetido automaticamente para todos os arquivos da pasta de testes, permitindo cobertura sistemática de todas as produções e regras semânticas

da gramática.

Esse processo garante que tanto a estrutura gramatical quanto as propriedades de integridade dos grafos definidos sejam rigorosamente testadas. A arquitetura modular, baseada em `ANTLR`, `Python` e `anytree`, provê flexibilidade para expandir o conjunto de testes ou adaptar as regras semânticas conforme novas funcionalidades sejam incorporadas à gramática.

4.3.9 Ferramentas Utilizadas

- **ANTLR 4** Parr (2013): geração de analisadores LL(1) e estrutura básica de `GraphParser`, `GraphLexer` e `GraphVisitor`;
- **Python**, Python Software Foundation (2021) + ***anytree*** Brunner (2023): geração de árvores sintáticas com indentação hierárquica por meio de *`render tree`*;
- **Casos de teste sintéticos**: mais de 10 arquivos foram projetados cobrindo todas as possibilidades da gramática.

Embora ferramentas como o *GNU Bison Parser Generator* (Gerador de Analisadores Bison do Projeto GNU) (GNU Bison) (baseado em análise *Left-to-Right, Rightmost derivation* (Análise Sintática Esquerda-para-Direita com Derivação mais à Direita) (LR)) também sejam compatíveis com o domínio, optou-se por **ANTLR 44** devido à maior legibilidade e facilidade na construção de analisadores descendentes para linguagens com estrutura hierárquica como grafos.

4.3.10 Exemplo de Saída

```
Testando: tests\invalido_multiplo_erro.txt
Erros semânticos encontrados:
 - Vértice 'x1' falta atributo obrigatório 'val'.
 - Identificador duplicado: 'x1'.
 - Vértice 'x2' usa tipo indefinido 'Y'.
 - Aresta 'e1' espera conexão (X -> X), recebeu (X -> Y).
 - Aresta 'e2' usa tipo indefinido 'Z'.
```

4.4 Conclusão da Metodologia

A metodologia adotada permitiu a construção de uma gramática livre de contexto robusta e extensível, validada de forma sistemática. A estrutura modular da linguagem, separando

tipos e instâncias, aliada à análise semântica automatizada, garantiu consistência e precisão.

A gramática demonstrou capacidade de representar grafos complexos, com suporte a herança, cardinalidade, atributos opcionais e múltiplos grafos. A próxima etapa consiste na integração da linguagem com um ambiente de compilação experimental para permitir inferência de tipos e geração de código orientado a grafos.

5 RESULTADOS

Este capítulo apresenta os principais resultados obtidos e as contribuições consolidadas ao final da implementação da gramática livre de contexto para tipagem de grafos. O projeto alcançou plenamente os objetivos estabelecidos, com a criação de uma linguagem formal validada sintática e semanticamente, capaz de representar grafos com propriedades avançadas como herança, multigrafos, atributos opcionais e verificação de restrições de tipo.

5.1 GLC para Tipagem de Grafos

O principal resultado do trabalho é a definição e implementação de uma GLC para grafos tipados, expressa inicialmente em notação EBNF e posteriormente traduzida para ANTLR 4 4. A gramática é:

- **Completa:** Cobre todas as construções essenciais de grafos: vértices, arestas, tipos, herança, cardinalidade, atributos opcionais, conexões múltiplas e grafos compostos;
- **Consistente:** A semântica é verificada por um *visitor pattern* customizado que impõe verificações de integridade, como compatibilidade de tipos e ausência de ciclos, com mensagens de erro claras e informativas;
- **Extensível:** Permite a introdução de novos tipos e regras sem modificar a estrutura principal da linguagem.

5.1.1 Quadro-Resumo de Cobertura dos Testes

Quadro 2 – Cobertura semântica da gramática por teste executado

Arquivo	**Objetivo e saída semântica**	**Resultado**
valido01.txt	Grafo com herança simples, atributos e conexões válidas. Nenhum erro semântico encontrado.	Aceito
valido02_undirected.txt	Aresta não-direcionada entre vértices do mesmo tipo. Nenhum erro semântico encontrado.	Aceito
valido03_multigrafo.txt	Dois grafos em uma só entrada. Nenhum erro semântico encontrado.	Aceito
valido04_heranca_profunda.txt	Herança em múltiplos níveis (C $\rightarrow$ B $\rightarrow$ A). Erro: Aresta 'l1' espera conexão (A $\rightarrow$ A), recebeu (C $\rightarrow$ A).	Rejeitado
valido05_atributos_opcionais.txt	Tipos com atributos opcionais e valores padrão. Nenhum erro semântico encontrado.	Aceito
invalido_atributo_ausente.txt	Atributo obrigatório ausente em vértice. Erro: Vértice 'u1' falta atributo obrigatório 'name'.	Rejeitado
invalido_ciclo_nao_permitido.txt	Aresta com *self-loop* explícito (x $\rightarrow$ x). Erro: Aresta 'c1' forma ciclo (*self-loop*) não permitido.	Rejeitado
invalido_duplicata_vertice.txt	Identificador de vértice repetido. Erro: Identificador duplicado: 'n1'.	Rejeitado
invalido_multiplo_erro.txt	Caso extremo com múltiplos erros. Erros: Vértice 'x1' falta atributo obrigatório 'val'; Identificador duplicado: 'x1'; Vértice 'x2' usa tipo indefinido 'Y'; Aresta 'e1' espera (X $\rightarrow$ X), recebeu (X $\rightarrow$ Y); Aresta 'e2' usa tipo indefinido 'Z'.	Rejeitado
invalido_tipo_aresta.txt	Tipos invertidos na assinatura da aresta. Erro: Aresta 'e1' espera conexão (B $\rightarrow$ A), recebeu (A $\rightarrow$ B).	Rejeitado
invalido_vertice_tipo_nao_declarado.txt	Uso de tipo de vértice não declarado. Erro: Vértice 'p1' usa tipo indefinido 'Human'.	Rejeitado

Fonte: Elaborado pelo autor.

Como complemento à **Quadro-Resumo de Cobertura dos Testes**, o **Apêndice E** apresenta a listagem integral de cada arquivo de entrada — um por página — permitindo a inspeção detalhada das construções sintáticas e dos cenários de validação empregados. Desse modo, o

leitor pode replicar todos os experimentos e verificar, linha a linha, como cada caso de teste exercita as produções gramaticais e as regras semânticas descritas neste trabalho.

5.1.2 Relatório de Testes Realizados

- `valido01.txt`

  Contém um grafo com vértices e arestas tipados, atributos obrigatórios e opcionais corretamente atribuídos, além de herança simples. Nenhuma inconsistência foi encontrada. Resultado: **aceito**.
- `valido02_undirected.txt`

  Testa a correta interpretação de uma aresta simétrica entre dois usuários. A estrutura foi corretamente interpretada e aceita. Resultado: **aceito**.
- `valido03_multigrafo.txt`

  Apresenta dois grafos distintos no mesmo arquivo. O *parser* separa corretamente os escopos e nenhuma colisão de identificadores foi detectada. Resultado: **aceito**.
- `valido04_heranca_profunda.txt`

  Embora atributos sejam herdados corretamente, a semântica das conexões exige tipos exatos, e não subtipos. Resultado: **rejeitado**, conforme esperado.
- `valido05_atributos_opcionais.txt`

  A gramática permite atributos não obrigatórios e com valores padrão. Nenhum erro semântico foi detectado. Resultado: **aceito**.
- `invalido_atributo_ausente.txt`

  O vértice omite um campo `name` marcado como `required`. O verificador semântico rejeitou corretamente. Resultado: **rejeitado**.
- `invalido_ciclo_nao_permitido.txt`

  Uma aresta direcionada conecta um vértice a ele mesmo. A gramática não proíbe, mas a semântica rejeita explicitamente *self-loop*s. Resultado: **rejeitado**.
- `invalido_duplicata_vertice.txt`

  O identificador do vértice aparece duas vezes. O validador detecta a duplicidade e rejeita. Resultado: **rejeitado**.
- `invalido_multiplo_erro.txt`

  Caso extremo com cinco erros: ausência de atributo, tipo inexistente, duplicação, assinatura de aresta incompatível e uso de aresta indefinida. Todos os erros foram reportados.

Resultado: **rejeitado**.

– `invalido_tipo_aresta.txt`

Aresta com assinatura invertida em relação à especificação da gramática. Detectado e rejeitado corretamente. Resultado: **rejeitado**.

– `invalido_vertice_tipo_nao_declarado.txt`

Tipo `Human` não declarado. O verificador rejeitou como esperado. Resultado: **rejeitado**.

A gramática mostrou-se adequada para modelar estruturas complexas, sendo potencialmente aplicável em domínios como linguagens orientadas a grafos, DSLs acadêmicas e compiladores.

5.2 Validação Formal da Gramática

A validação foi realizada com o uso de ANTLR 4 4, Python e bibliotecas auxiliares como `anytree`. A seguir, os resultados consolidados:

– **Testes de Conformidade:** Todos os casos positivos (ex: herança simples, multigrafos, atributos opcionais) foram corretamente aceitos;

– **Testes de Rejeição:** Casos negativos como atributos ausentes, tipos indefinidos, duplicação e assinaturas incompatíveis foram todos corretamente identificados e rejeitados;

– **Robustez:** Foram testados grafos com estruturas complexas, subgrafos independentes, *self-loop*s proibidos, erros combinados e carga elevada de elementos. Todos os erros semânticos esperados foram reportados.

Além disso, a ferramenta `main.py` gera automaticamente árvores sintáticas para cada grafo testado, com formatação legível, armazenadas na pasta `tree_tests/`. A análise dessas árvores ajudou a depurar a estrutura da gramática.

5.3 Contribuições Consolidadas

Com base na execução bem-sucedida dos testes e nos arquivos de validação gerados, as contribuições deste trabalho podem ser assim sintetizadas:

– **Gramática validada e funcional:** com cobertura de propriedades semânticas ricas e implementada em ferramentas modernas de análise sintática;

– **Ferramenta de análise de grafos:** capaz de aceitar entradas textuais, gerar árvores e validar restrições de integridade;

- **Repositório funcional e reutilizável:** O projeto encontra-se publicado em (PINHEIRO, 2025), permitindo sua replicação e extensão por pesquisadores e estudantes;
- **Base para experimentação acadêmica:** linguagem clara, modular e segura, que pode ser estendida para aplicações pedagógicas e de pesquisa;
- **Integração com compiladores:** a estrutura de tipos, atributos e herança reflete abstrações comuns em linguagens modernas, facilitando a integração com estágios como análise semântica e geração de código.

Em resumo, o trabalho entregou uma infraestrutura formal sólida e validada, com resultados consistentes entre os testes planejados e os comportamentos esperados da gramática, contribuindo diretamente para avanços no uso de grafos em linguagens formais e compiladores. Para ter acesso aos resultados, todos os arquivos testados e ao projeto de validação da gramática acesse o projeto feito pelo autor, Pinheiro (2025).

6 CONCLUSÕES E TRABALHOS FUTUROS

Este capítulo encerra o trabalho apresentando uma síntese dos resultados alcançados com a GLC proposta, destacando sua contribuição para a tipagem estática de grafos e a validação automática de estruturas complexas. Em seguida, são delineadas perspectivas de evolução que incluem o suporte a grafos temporais, a incorporação de metadados e anotações avançadas, a integração com *Integrated Development Environment* (Ambiente de Desenvolvimento Integrado)s (IDEs), a geração de bytecode ou artefatos intermediários e a interoperabilidade com bibliotecas externas como NetworkX e Neo4j. Por fim, as considerações finais reforçam a relevância acadêmica e prática da gramática, bem como seu potencial de servir de base para futuras DSLs, compiladores e *pipeline*s de análise orientados a grafos.

6.1 Conclusões

Este trabalho apresentou a construção de uma GLC para tipagem de grafos, com foco em garantir consistência sintática e semântica, validada formalmente por meio de ferramentas como ANTLR 4 e análise baseada em `visitor patterns`. A gramática foi implementada, testada e avaliada por uma série de arquivos que simulam aplicações reais e cenários extremos.

Os principais resultados obtidos foram:

- **Gramática formal completa:** Capaz de representar grafos com herança de tipos, múltiplas arestas, atributos obrigatórios e opcionais, cardinalidade e subgrafos distintos.
- **Validação automatizada:** A gramática foi integrada a um *pipeline* de testes estruturado, com suporte à geração de árvores sintáticas e relatórios de erro precisos.
- **Repositório funcional e reutilizável:** O projeto encontra-se publicado em (PINHEIRO, 2025), permitindo sua replicação e extensão por pesquisadores e estudantes.

A estrutura modular e extensível da linguagem torna a gramática útil tanto para fins acadêmicos quanto para aplicações práticas em DSLs, ambientes de compilação e sistemas de análise orientados a grafos.

6.2 Trabalhos Futuros

As possibilidades de evolução deste trabalho são amplas. Algumas propostas viáveis incluem:

- **Generalização para Grafos Temporais e Dinâmicos:** Adicionar suporte à modelagem

de grafos que variam ao longo do tempo, com atributos temporais ou históricos.

- **Suporte a Metadados e Anotações:** Permitir comentários, documentação inline ou restrições avançadas por meio de expressões lógicas associadas aos tipos.
- **Integração com Ambientes de Desenvolvimento:** Construção de editores visuais e IDEs com validação sintática e semântica em tempo real.
- **Backend para Compiladores:** Estender a gramática com instruções operacionais que possibilitem sua tradução para bytecode, código intermediário ou estruturas de execução (ex: *JavaScript Object Notation* (JSON), *Extensible Markup Language* (XML), AST).
- **Visualização e Exportação:** Ferramentas para exportar os grafos validados para formatos de visualização (Linguagem de descrição de grafos do Graphviz (DOT), *Scalable Vector Graphics* (SVG)) ou interoperabilidade com bibliotecas externas (ex: NetworkX, Neo4j).

6.3 Considerações Finais

Este trabalho representou um avanço importante na formalização da tipagem de grafos por meio de uma gramática robusta, validada e implementada em ferramentas modernas. Combinando clareza sintática e expressividade semântica, a proposta se mostra viável como base para DSLs, modelos de compiladores e aplicações acadêmicas.

A gramática proposta atendeu aos critérios de completude, consistência e extensibilidade. Os testes realizados comprovaram sua capacidade de detectar erros, validar restrições complexas e lidar com múltiplas topologias. Com isso, ela se consolida como uma ferramenta formal de alto potencial para representar grafos em linguagens de programação estruturadas.

Espera-se que as futuras extensões e aplicações derivadas desta base fortaleçam o uso de grafos como estruturas de primeira classe em ambientes formais e produtivos.

REFERÊNCIAS

BONDY, J.; MURTY, U. **Graph Theory with Applications**. [*S. l.*]: Elsevier, 1976.

BRUNNER, C. **anytree: Powerful and Lightweight Tree Data Structures in Python**. 2023. https://anytree.readthedocs.io/. Biblioteca para representação hierárquica de árvores sintáticas em Python.

CARDELLI, L. A semantics of multiple inheritance. **Information and Computation**, v. 76, n. 2-3, p. 138–164, 1985.

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. **Introduction to Algorithms**. [*S. l.*]: MIT Press, 2009.

CORRADINI, A.; EHRIG, H.; MONTANARI, U.; RIBEIRO, L.; ROSSI, F. Specifying graph languages with type graphs. In: **Handbook of Graph Grammars and Computing by Graph Transformation**. Singapore: World Scientific, 1997. v. 1, p. 1–61.

DIESTEL, R. **Graph Theory**. [*S. l.*]: Springer, 2017.

EHRIG, H.; ENGELS, G.; KREOWSKI, H.-J.; ROZENBERG, G. Graph grammars and their application to computer science. **Lecture Notes in Computer Science**, Springer, v. 532, p. 1–20, 1991.

EULER, L. Solutio problematis ad geometriam situs pertinentis. **Commentarii Academiae Scientiarum Imperialis Petropolitanae**, v. 8, p. 128–140, 1736.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. **Design Patterns: Elements of Reusable Object-Oriented Software**. Reading, MA: Addison–Wesley, 1995. ISBN 978-0201633610.

GROSS, J. L.; YELLEN, J. **Graph Theory and Its Applications**. 3. ed. [*S. l.*]: CRC Press, 2014.

HARARY, F. **Graph Theory**. [*S. l.*]: Addison–Wesley, 1969.

HART, P. E.; NILSSON, N. J.; RAPHAEL, B. A formal basis for the heuristic determination of minimum cost paths. **IEEE Transactions on Systems Science and Cybernetics**, v. 4, n. 2, p. 100–107, 1968.

HECKEL, R.; TAENTZER, G. Type systems for graph transformation. **Electronic Notes in Theoretical Computer Science**, Elsevier, v. 148, n. 1, p. 19–40, 2006.

HINDLEY, R. The principal type-scheme of an object in combinatory logic. **Transactions of the American Mathematical Society**, JSTOR, v. 146, p. 29–60, 1969.

HOPCROFT, J. E.; MOTWANI, R.; ULLMAN, J. D. **Introduction to Automata Theory, Languages, and Computation**. 3rd. ed. [*S. l.*]: Pearson, 2006. Seção 4.2 discute transformações de EBNF para BNF.

MENS, T.; GORP, P. V. A taxonomy of model transformation. In: **ICMT 2006**. [*S. l.*]: Springer, 2006, (LNCS, v. 4066). p. 125–142.

MILNER, R. A theory of type polymorphism in programming. **Journal of Computer and System Sciences**, Elsevier, v. 17, n. 3, p. 348–375, 1978.

PARR, T. **ANTLR (Another Tool for Language Recognition)**. [*S. l.*], 2013. Versão 4. Disponível em: https://www.antlr.org/.

PIERCE, B. C. **Types and Programming Languages**. [*S. l.*]: MIT Press, 2002.

PINHEIRO, A. **GramaticaTipagemGrafosGLC**. 2025. https://github.com/alyssonlcss/GramaticaTipagemGrafosGLC. Projeto de validação semântica de grafos com ANTLR4 e Python.

Python Software Foundation. **Python Language Reference, version 3.10**. 2021. https://www.python.org/. Linguagem de programação de alto nível, interpretada e de tipagem dinâmica.

REZENDE, C. A. d. R. **Jaguar-Lang: Uma Linguagem de Programação para Fins de Ensino e Pesquisa em Compiladores: Uma Abordagem Paralela e Distribuída**. 2025. https://cadproj.ufc.br/projects/592. Projeto em andamento na Universidade Federal do Ceará. Aprovado em 08/05/2025.

RUMBAUGH, J.; JACOBSON, I.; BOOCH, G. **The Unified Modeling Language Reference Manual**. 2. ed. [*S. l.*]: Addison–Wesley, 2004.

TARJAN, R. Depth-first search and linear graph algorithms. **SIAM Journal on Computing**, v. 1, n. 2, p. 146–160, 1972.

WEBBER, J.; ROBINSON, I.; EIFREM, E. **Graph Databases: New Opportunities for Connected Data**. 1st. ed. [*S. l.*]: O'Reilly Media, 2012. ISBN 978-1449356262.

WEST, D. B. **Introduction to Graph Theory**. [*S. l.*]: Prentice Hall, 2001.

WIRTH, N. **Compiler Construction**. [*S. l.*]: Addison–Wesley, 1996.

GLOSSÁRIO

anytree	Biblioteca Python que oferece estruturas de árvore genéricas e utilitários para percorrê-las, manipulá-las e renderizá-las de forma simples
directed	Qualificador de aresta que especifica orientação explícita, definindo dígrafos em que cada aresta possui vértice de origem e vértice de destino distintos
extends	Palavra-chave da gramática que indica Herança (entre tipos) entre tipos de vértice, permitindo que um subtipo reutilize atributos e restrições do supertipo
namespace	Espaço de nomes isolado que evita colisões entre identificadores em diferentes grafos ou contextos
parser	módulo de análise sintática que, a partir da sequência de tokens produzida pelo lexer e de uma gramática, decide se a entrada pertence à linguagem e constrói uma representação estrutural (árvore de derivação ou AST). Exemplos de famílias: LL, LR, Earley e CYK; pode incluir estratégias de recuperação de erros
pipeline	Sequência ordenada de etapas de processamento (*stages*) que transformam dados ou artefatos de software de forma incremental, favorecendo paralelismo, modularidade e automação — por exemplo, o *pipeline* de validação que executa *parsing*, visitação da AST e geração de relatórios

render tree	Função da biblioteca *anytree* que imprime uma árvore hierárquica em formato textual, facilitando a visualização da AST gerada após o *parsing*
self-loop	Aresta que conecta um vértice a si mesmo; sua validade depende das regras semânticas vigentes
undirected	Qualificador de aresta que especifica ausência de orientação, caracterizando grafos não direcionados onde as conexões são bidirecionais
visitor pattern	Padrão de projeto orientado a objetos que permite aplicar operações a estruturas de árvore (como ASTs) sem alterar suas classes (GAMMA *et al.*, 1995)
A*	algoritmo de busca informada para caminhos mínimos que usa a função $f(n) = g(n) + h(n)$, combinando o custo já percorrido g com uma heurística h até o objetivo. Com heurística admissível garante otimalidade; com heurística consistente evita reaberturas. Para $h \equiv 0$, reduz-se ao Dijkstra's algorithm
analisador sintático	sinônimo de *parser*
aresta	elemento de um grafo que conecta dois vértices, podendo ser direcionada ou não

bytecode	Código intermediário de baixo nível, independente de arquitetura, gerado por compiladores e executado por uma máquina virtual; combina portabilidade com desempenho, como o *bytecode* produzido pela JVM ou pelo interpretador Python
caminho	sequência finita de vértices $(v_0, \ldots, v_k)$ tal que, para todo i, existe $\{v_i, v_{i+1}\} \in E$ (ou $(v_i, v_{i+1}) \in E$ no caso dirigido). O comprimento do caminho é k, isto é, o número de arestas percorridas
cardinalidade	Restrição que especifica a quantidade mínima e máxima de ocorrências de um elemento (como vértice ou aresta), inspirada na notação da UML (RUMBAUGH *et al.*, 2004)
Dijkstra's algorithm	Algoritmo de caminhos mínimos de fonte única em grafos ponderados com pesos não negativos, proposto por Edsger W. Dijkstra em 1959; utiliza fila de prioridade e tem complexidade $O\big((\lvert V\rvert + \lvert E\rvert)\log \lvert V\rvert\big)$.
entidade de primeira classe	Elemento que pode ser criado, atribuído a variáveis, passado como argumento e retornado por funções. Tratado como valor nativo pelas construções da linguagem (PIERCE, 2002)
grafo	estrutura matemática $G = (V, E)$ que representa relações entre vértices por meio de Arestas
grafo conexo	grafo no qual existe ao menos um caminho entre quaisquer dois vértices; caso contrário, o grafo é desconexo e decompõe-se em componentes conexas

grafo direcionado	grafo cujas Arestas possuem orientação, isto é, cada aresta representa um par ordenado (v_i, v_j)
grafo planar	grafo que pode ser desenhado no plano sem cruzamento de Arestas; pela caracterização de Kuratowski, grafos contendo um subgrafo homeomorfo a K_5 ou $K_{3,3}$ não são planares
grafo ponderado	grafo no qual a cada Aresta é associado um peso $w : E \to \mathbb{R}$ (p. ex. custo, capacidade ou distância)
grafo simples	grafo que não admite laços $(v_i = v_j)$ nem múltiplas Arestas paralelas entre o mesmo par de vértices
herança (entre tipos)	Recurso que permite que um tipo derivado reutilize atributos e regras de um tipo base (CARDELLI, 1985)
homeomorfo (em grafos)	dois grafos são homeomorfos se um pode ser obtido do outro por uma sequência de subdivisões de arestas e supressões de vértices de grau 2; de forma equivalente, possuem subdivisões isomorfas. A noção é central na caracterização de planaridade de Kuratowski (DIESTEL, 2017; GROSS; YELLEN, 2014)
inferência de tipos	Mecanismo que permite ao compilador deduzir automaticamente os tipos das expressões sem anotações explícitas (HINDLEY, 1969; MILNER, 1978)
lexer	fase de compilador que converte o fluxo de caracteres de entrada em uma sequência de *tokens*, a ser consumida pelo analisador sintático

lista de adjacência	vetor de listas $A[v]$ contendo os vizinhos de cada $v \in V$; ocupa $O(\|V\| + \|E\|)$ de memória e é preferível para grafos esparsos
lista de incidência	representação que armazena explicitamente todas as Arestas como pares (ou triplas com peso) de vértices incidentes; útil em algoritmos que ordenam/percorrrem arestas, como Kruskal
matriz de adjacência	matriz $\|V\| \times \|V\|$ em que a célula (i, j) indica presença (e opcionalmente o peso) da Aresta entre v_i e v_j; acesso $O(1)$ e custo de memória $O(\|V\|^2)$ (simétrica em grafos não-direcionados)
metadado	Dados que descrevem outros dados, oferecendo contexto semântico ou estrutural adicional (por exemplo, rótulos de atributos, unidades de medida ou restrições de validação)
multigrafo	grafo que admite múltiplas Arestas distintas entre o mesmo par de vértices; laços podem ser permitidos conforme a definição adotada
Neo4j	Sistema de gerenciamento de banco de dados orientado a grafos baseado no modelo *Property Graph*; utiliza a linguagem declarativa Cypher para consultar e manipular vértices e arestas com rótulos e propriedades
NetworkX	Biblioteca de análise de grafos em Python que fornece estruturas de dados flexíveis e algoritmos para criação, manipulação, visualização e estudo de grafos complexos

Python	Linguagem de programação de alto nível, interpretada e multiparadigma, criada por Guido van Rossum em 1991; amplamente utilizada em ciência de dados, automação, *Application Programming Interface* (Interface de Programação de Aplicações)s (APIs) e desenvolvimento web
sistema de tipos	Conjunto de regras formais que associa tipos às expressões de uma linguagem de programação, prevenindo operações inválidas e proporcionando garantias de segurança como *soundness* e *completeness*
subgrafo	subconjunto de vértices e Arestas de um grafo original que preserva incidência
tipagem estática	Sistema no qual os tipos são verificados em tempo de compilação, permitindo detectar erros antes da execução (PIERCE, 2002; MILNER, 1978)
vértice	elemento de um grafo que representa uma entidade ou objeto; também chamado de nó

APÊNDICE A – DECLARAÇÃO COMPLETA DO GRAFO `FLIGHTNETWORK`

Código-fonte 1 – Código-fonte completo do grafo `flightNetwork`

```
graph flightNetwork {
  types {
    vertex Airport [1..*] attributes {
      code : string required;
      city : string required;
    };
    vertex Hub extends Airport;

    edge Flight (Airport, Airport) directed
         attributes { duration : int; };
  }

  vertices {
    gru : Hub     [ code="GRU", city="Sao Paulo" ];
    jfk : Airport [ code="JFK", city="New York" ];
  }

  edges {
    f1 : Flight (gru -> jfk) [ duration = 540 ];
  }
}
```

APÊNDICE B – DECLARAÇÃO COMPLETA DO GRAFO `SOCIALNET`

Código-fonte 2 – Código-fonte completo do grafo `socialNet`

```
graph socialNet {
  types {
    vertex Person [1..*] attributes {
      id   : int    required;
      name : string required;
      age  : int    default = 18;
    };

    vertex Student extends Person
                   attributes { university : string; };

    edge Knows   (Person, Person) undirected;
    edge Follows (Person, Person) directed;
  }

  vertices {
    a : Person  [ id=1, name="Ana",   age=21 ];
    b : Student [ id=2, name="Bruno", university="UFC" ];
  }

  edges {
    e1 : Knows   (a -- b);
    e2 : Follows (a -> b);
  }
}
```

APÊNDICE C – GRAMÁTICA COMPLETA EM EBNF PARA TIPAGEM DE GRAFOS

Código-fonte 3 – Gramática EBNF para tipagem robusta de grafos

```
<program> ::= { <graph> }

<graph> ::= "graph" <id> "{" <type_section> <vertex_section> <edge_section> "}"

<type_section> ::= "types" "{" { <vertex_type_decl> | <edge_type_decl> } "}"

<vertex_type_decl> ::= "vertex" <id> [ "extends" <id> ]
                       [ <cardinality> ]
                       [ <attribute_block> ] ";"

<edge_type_decl> ::= "edge" <id> "(" <id> "," <id> ")"
                     [ "directed" | "undirected" ]
                     [ <cardinality> ]
                     [ <attribute_block> ] ";"

<cardinality> ::= "[" <min_card> ".." <max_card> "]"
<min_card> ::= <int>
<max_card> ::= <int> | "*"

<attribute_block> ::= "attributes" "{" { <attribute_decl> } "}"
<attribute_decl> ::= <id> ":" <type> [ "required" | "optional" ] [ "=" <value> ] ";"

<vertex_section> ::= "vertices" "{" { <vertex_instance> } "}"
<vertex_instance> ::= <id> ":" <id> [ <attribute_assign_block> ] ";"

<edge_section> ::= "edges" "{" { <edge_instance> } "}"
<edge_instance> ::= <id> ":" <id> "(" <id> "->" <id> ")" [ <attribute_assign_block> ] ";"

<attribute_assign_block> ::= "[" { <attribute_assignment> "," } <attribute_assignment> "]"
<attribute_assignment> ::= <id> "=" <value>

<type> ::= "int" | "float" | "string" | "bool" | "date" | <id>
<value> ::= <int> | <float> | <string> | "true" | "false" | <date>

<id> ::= (letter | "_") { letter | digit | "_" }
<int> ::= digit { digit }
<float> ::= <int> "." <int>
<string> ::= '"' { character - '"' } '"'
<date> ::= <int> "-" <int> "-" <int>

<comment> ::= "//" { character } | "#" { character }
```

APÊNDICE D – EXEMPLO COMPLETO DO GRAFO `COMPANYNETWORK`

Código-fonte 4 – Código-fonte completo do grafo `companyNetwork`

```
graph companyNetwork {
  types {
    vertex Person [1..*] attributes {
      name: string required;
      age: int;
    };

    vertex Manager extends Person attributes {
      level: string = "senior";
    };

    vertex Company [1..*] attributes {
      name: string required;
    };

    edge WorksAt (Person, Company) directed [0..*];
    edge Manages (Manager, Person) directed [0..*];
  }

  vertices {
    p1 : Person  [ name="Alice",  age=30 ];
    p2 : Person  [ name="Bob",    age=25 ];
    m1 : Manager [ name="Carol",  age=40, level="executive" ];
    c1 : Company [ name="OpenAI" ];
  }

  edges {
    e1 : WorksAt (p1 -> c1);
    e2 : WorksAt (p2 -> c1);
    e3 : Manages (m1 -> p2);
  }
}
```

APÊNDICE E – ARQUIVOS DE TESTE DA GRAMÁTICA

Códigos-fonte 5 até 15

Código-fonte 5 – `invalido_vertice_tipo_nao_declarado.txt`

```
1 graph invalidVertexType {
2   types {
3     vertex Person;
4   }
5
6   vertices {
7     x : Human;          // tipo Human nao declarado
8   }
9 }
```

Código-fonte 6 – `valido01.txt`

```
graph companyNetwork {
  types {
    vertex Person [1..*] attributes {
      name: string required;
      age: int;
    };

    vertex Manager extends Person attributes {
      level: string = "senior";
    };

    vertex Company [1..*] attributes {
      name: string required;
    };

    edge WorksAt (Person, Company) directed [0..*];
    edge Manages (Manager, Person) directed [0..*];
  }

  vertices {
    p1 : Person  [ name="Alice",  age=30 ];
    p2 : Person  [ name="Bob",    age=25 ];
    m1 : Manager [ name="Carol",  age=40, level="executive" ];
    c1 : Company [ name="OpenAI" ];
  }

  edges {
    e1 : WorksAt (p1 -> c1);
    e2 : WorksAt (p2 -> c1);
    e3 : Manages (m1 -> p2);
  }
}
```

Código-fonte 7 – `valido02_undirected.txt`

```
graph friendship {
  types {
    vertex Person;
    edge Knows (Person, Person) undirected;
  }

  vertices {
    a : Person;
    b : Person;
  }

  edges {
    e1 : Knows (a -- b);
  }
}
```

Código-fonte 8 – `valido03_multigrafo.txt`

```
graph transport {
  types {
    vertex City;
    edge Road  (City, City) undirected;
    edge Train (City, City) undirected;
  }

  vertices {
    s : City;
    p : City;
  }

  edges {
    r1 : Road  (s -- p);
    t1 : Train (s -- p);   // multigrafo: duas arestas paralelas
  }
}
```

Código-fonte 9 – `valido04_heranca_profunda.txt`

```
graph academic {
  types {
    vertex Person;
    vertex Student extends Person;
    vertex PhD     extends Student;
  }

  vertices {
    x : PhD;
  }
}
```

Código-fonte 10 – `valido05_atributos_opcionais.txt`

```
graph sensors {
  types {
    vertex Device attributes {
      id   : int    required;
      temp : float  optional;
    };
  }

  vertices {
    d1 : Device [ id = 10 ];
    d2 : Device [ id = 11, temp = 22.5 ];
  }
}
```

Código-fonte 11 – `invalido_atributo_ausente.txt`

```
graph missingAttr {
  types {
    vertex User attributes {
      name : string required;
    };
  }

  vertices {
    u1 : User;              // falta atributo obrigatorio 'name'
  }
}
```

Código-fonte 12 – `invalido_ciclo_nao_permitido.txt`

```
graph noLoops {
  types {
    vertex Node;
    edge Link (Node, Node) directed;
  }

  vertices {
    x : Node;
  }

  edges {
    l1 : Link (x -> x);     // self-loop proibido
  }
}
```

Código-fonte 13 – `invalido_duplicata_vertice.txt`

```
1 graph dupVertex {
2   vertices {
3     x : vertex;
4     x : vertex;             // identificador duplicado
5   }
6 }
```

Código-fonte 14 – `invalido_multiplo_erro.txt`

```
graph multiError {
  types {
    vertex X attributes { val : int required; };
    edge   E (X, X);
  }

  vertices {
    x1 : X;                 // falta atributo 'val'
    x1 : X [ val = 1 ];     // duplicado
    x2 : Y;                 // tipo Y nao existe
  }

  edges {
    e1 : E (x1 -> x2);      // destino de tipo incompativel
    e2 : Z (x1 -> x1);      // aresta de tipo indefinido
  }
}
```

Código-fonte 15 – `invalido_tipo_aresta.txt`

```
graph badEdgeType {
  types {
    vertex A;
    vertex B;
    edge   E (A, A) directed;
  }

  vertices {
    a : A;
    b : B;
  }

  edges {
    e1 : E (a -> b);        // origem/destino nao batem com assinatura de E
  }
}
```