INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DO RIO GRANDE DO NORTE

ANA MARIA FERREIRA DE ABREU GUEDES

NILO: UM MODELO DE PROCESSO DE SOFTWARE PARA O DESENVOLVIMENTO DE LINGUAGENS DE PROGRAMAÇÃO

NATAL

2026

ANA MARIA FERREIRA DE ABREU GUEDES

NILO: UM MODELO DE PROCESSO DE SOFTWARE PARA O DESENVOLVIMENTO DE LINGUAGENS DE PROGRAMAÇÃO

Trabalho de Conclusão de Curso apresentado ao Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas do Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Norte, em cumprimento às exigências legais como requisito parcial à obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas.

Orientadora: Dr. Jorgiano Márcio Bruno Vidal.
Coorientador: Dra. Marília Aranha Freire.

NATAL
2026

Instituto Federal de Educação, Ciência e Tecnologia do Estado do Rio Grande do Norte – IFRN
Sistema Integrado de Bibliotecas – SIBi

Guedes, Ana Maria Ferreira de Abreu.
G924n NILO : um modelo de processo de software para o desenvolvimento de linguagens de programação / Ana Maria Ferreira de Abreu Guedes. – 2026.
54 f. : il. color.

Trabalho de Conclusão de Curso (Graduação) – Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Norte, Natal, 2026.
Orientador: Prof. Dr. Jorgiano Márcio Bruno Vidal.
Coorientador: Dra. Marília Aranha Freire.

1. Linguagem de programação. 2. Modelo de processo de software. 3. Desenvolvimento de linguagens de programação. 4. Engenharia de software. I. Vidal, Jorgiano Márcio Bruno. II. Freire, Marília Aranha. III. Título.

SIBi/IFRN CDU 004.43

Elaborada pela Bibliotecária
Maria Ilza da Costa – CRB-15/412

Ana Maria Ferreira de Abreu Guedes

NILO: UM MODELO DE PROCESSO DE SOFTWARE PARA O DESENVOLVIMENTO DE LINGUAGENS DE PROGRAMAÇÃO

Trabalho de Conclusão de Curso apresentado ao Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas do Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Norte, em cumprimento às exigências legais como requisito parcial à obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas.

Trabalho de Conclusão de Curso aprovado em 09/07/2026, pela seguinte Banca Examinadora:

gov.br
Documento assinado digitalmente
JORGIANO MARCIO BRUNO VIDAL
Data: 13/07/2026 10:35:10-0300
Verifique em https://validar.iti.gov.br

Dr. Jorgiano Marcio Bruno Vidal – Orientador

Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Norte

gov.br
Documento assinado digitalmente
MARILIA ARANHA FREIRE
Data: 13/07/2026 09:26:26-0300
Verifique em https://validar.iti.gov.br

Dra. Marília Aranha Freire - Coorientadora

Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Norte

gov.br
Documento assinado digitalmente
GALILEU BATISTA DE SOUSA
Data: 11/07/2026 11:13:13-0300
Verifique em https://validar.iti.gov.br

Me. Galileu Batista de Sousa

Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Norte

gov.br
Documento assinado digitalmente
MARCIO EDUARDO KREUTZ
Data: 10/07/2026 11:43:00-0300
Verifique em https://validar.iti.gov.br

Dr. Marcio Eduardo Kreutz

Universidade Federal do Rio Grande do Norte

AGRADECIMENTOS

Cada enunciação ou axioma é insuficiente e diminuto para exprimir tamanha gratidão e carinho por todos que prestaram apoio desde o princípio. Para correr, assim que aprendi a caminhar, foi fastidioso, mas cada pilar, que posso chamar de família, sendo consanguínea ou não, foi sustento vital.

“*Eu te digo: estou tentando captar a quarta dimensão do instante-já que de tão fugidio não é mais porque agora tornou-se um novo instante-já que também não é mais. Cada coisa tem um instante em que ela é. Quero apossar-me do é da coisa*.” (Clarice Lispector)

RESUMO

Na Engenharia de Software, abordagens baseadas em métodos, metodologias ou processos são amplamente empregadas para guiar a definição, concepção, avaliação e manutenção de sistemas. Contudo, quando o foco se volta para as linguagens de programação, o panorama é diferente: embora a literatura reconheça o caráter evolutivo dessas linguagens, nota-se uma escassez de processos sistematizados equivalentes aos da engenharia de software tradicional. Essa ausência de modelos estruturados acarreta desafios significativos, sobretudo na manutenção e evolução de longo prazo das linguagens. Para mitigar esse problema, este trabalho propõe o NILO, um modelo de processo iterativo e incremental projetado para guiar o desenvolvimento de linguagens de programação de maneira sistemática. Adicionalmente, a viabilidade de suas fases e artefatos é validada por meio de um estudo de caso.

Palavras-chave: linguagens de programação; modelo de processo de software; desenvolvimento de linguagens de programação.

ABSTRACT

In Software Engineering, approaches based on methods, methodologies, or processes are widely employed to guide the definition, design, evaluation, and maintenance of systems. However, when the focus shifts to programming languages, the landscape is different: although the literature recognizes the evolutionary nature of these languages, there is a noticeable scarcity of systematized processes equivalent to those found in traditional software engineering. This absence of structured models entails significant challenges, particularly in the long-term maintenance and evolution of languages. To mitigate this problem, this work proposes NILO, an iterative and incremental process model designed to systematically guide the development of programming languages. Additionally, the feasibility of its phases and artifacts is validated through a case study.

Keywords: programming languages; software process models; language development.

LISTA DE ILUSTRAÇÕES

SUMÁRIO

1 INTRODUÇÃO

No âmbito da Engenharia de Software, o desenvolvimento de sistemas se utiliza frequentemente de abordagens técnicas particulares para a especificação, desenvolvimento, validação e evolução de sistemas (Sommerville, 2019). Essas abordagens geralmente são métodos, metodologias, técnicas e modelos de processo que estabelecem fases e artefatos que buscam fornecer orientação para realizar o trabalho de selecionar as ações e tarefas apropriadas para o projeto, com o objetivo de entregar o software com qualidade e conformidade com os requisitos (Pressman, 2021). Dentre as abordagens mais difundidas na engenharia de software destacam-se o modelo de Processo Unificado (UP), o modelo em Cascata e *frameworks* ágeis como o Scrum, os quais possuem a mesma finalidade de planejar, criar e manter um sistema garantindo o controle, a qualidade e a manutenibilidade do artefato resultante.

Contudo, quando o objeto de desenvolvimento se desloca para linguagens de programação, o cenário metodológico difere. Para compreender esse contexto, é necessário reconhecer tanto a natureza dessas linguagens quanto sua relevância. De acordo com Aho et al. (2008), as linguagens de programação são notações utilizadas para descrever instruções para pessoas e para máquinas. Sua relevância reside na resolução de problemas em diferentes domínios (Cuevas; Zaldivar; Perez, 2025), sendo essenciais no cenário tecnológico atual por serem utilizadas na construção de todos os *softwares* de um computador (Aho et al., 2008). A importância das linguagens evidencia a necessidade de refletir não apenas sobre sua aplicação, mas também sobre os processos que orientam sua concepção e evolução.

Nesse sentido, assim como os sistemas de *software* em geral, as linguagens de programação são consideradas evolutivas, sendo construídas e mantidas de forma cíclica. Autores como Favre (2005), Voelter et al. (2013) e Mernik, Heering e Sloane (2005) destacam a necessidade de uma abordagem iterativa que permita que uma linguagem evolua, além de reforçarem que atualmente há uma limitação de trabalhos relacionados a metodologias de desenvolvimento de linguagens de programação de domínio específico.

No entanto, mesmo com a compreensão da natureza cíclica e evolutiva das linguagens pela literatura acadêmica, ainda existe uma lacuna na intersecção entre

o desenvolvimento de linguagens e as práticas formais de Engenharia de *Software*. Isso se justifica, pois, diferentemente dos modelos de processo tradicionais aplicados a outros tipos de software, a concepção de linguagens carece de formalização de fases associadas à geração de artefatos, comuns em engenharia de *software*, tais como especificações de requisitos de linguagem e documentação de decisões de projeto. Consequentemente, seu projeto e implementação frequentemente não ocorrem de forma a contribuir para sua evolução (Favre, 2005).

Além disso, não há técnicas universais para a engenharia de *software*, pois os diferentes tipos de *software* exigem abordagens distintas (Sommerville, 2019). Isso significa que, apesar de as metodologias, métodos e *frameworks* tradicionais de desenvolvimento de software não especificarem restrições quanto aos tipos de sistemas produzidos — permitindo que a concepção de linguagens também se beneficie dessas abordagens — seus artefatos não são totalmente adequados para a produção de linguagens de programação, por sua característica mais voltadas para o desenvolvimento de sistemas. Diante disso, um processo específico deve ser utilizado para conceber linguagens de programação da forma mais apropriada às características desse domínio. Entretanto, na literatura não foram identificadas propostas que definem etapas e artefatos documentais e de código para a concepção e manutenção de linguagens.

Dessa forma, observou-se a necessidade de elaboração de um recurso capaz de guiar o planejamento, desenvolvimento e evolução de linguagens de programação. Por isso, este trabalho propõe um modelo de processo, denominado NILO, para construção e manutenção de linguagens de programação.

1.1 OBJETIVOS

1.1.1 Objetivo Geral

Especificar um modelo de processo para a criação de linguagens de programação.

1.1.2 Objetivos Específicos

- Elicitar os requisitos necessários para a construção do modelo de processo a partir da análise de trabalhos correlatos;
- Estruturar o modelo de processo NILO em etapas e artefatos definidos que orientem o desenvolvimento de uma linguagem de programação de forma iterativa e incremental;
- Demonstrar e avaliar a viabilidade do NILO por meio de sua aplicação prática na construção de uma linguagem de programação.

1.2 ORGANIZAÇÃO DO TRABALHO

Este trabalho está estruturado em 7 (sete) seções que discutem desde a contextualização até a explicação dos resultados e trabalhos futuros.

Primeiramente, a **introdução** apresenta o contexto e a motivação que cercam a criação do trabalho, além de explicar seus principais objetivos.

Em seguida, a seção de **metodologia** expõe as etapas que envolveram desde a identificação da lacuna até a avaliação da viabilidade do fluxo NILO.

Enquanto isso, a seção de **desenvolvimento** adentra na elicitação dos requisitos e na tomada de decisões de projeto para suprir a lacuna observada.

A seção de **modelo de processo NILO** tem como objetivo apresentar o modelo, assim como suas fases, artefatos e ciclo de vida.

O **estudo de caso**, quinta seção, discute a criação de uma linguagem de programação. Cada subseção corresponde à execução de uma fase específica, com o objetivo de validar empiricamente o NILO. Ao final, são discutidos os problemas identificados durante a execução e as soluções implementadas para cada um deles.

Na seção de **trabalhos relacionados**, são apresentados os estudos que influenciaram a concepção do modelo, destacando-se as similaridades preservadas e os diferenciais considerados necessários para sua evolução.

Por fim, na seção de **conclusão final** é demonstrada a intenção de alcançar diferentes contextos de utilização, incremento de práticas comuns na engenharia de *software* e de realizar melhorias no modelo.

2 METODOLOGIA

A pesquisa que compreendeu a concepção e a validação do NILO foi conduzida segundo a metodologia *Design Science Research Methodology* (DSRM), concebida por Peffers *et al.* (2007), cuja finalidade é a criação e avaliação de artefatos como solução de um determinado problema. Os autores definem a metodologia com as etapas de: (1) identificação e motivação do problema; (2) definição dos objetivos para uma solução; (3) design e desenvolvimento; (4) demonstração; (5) avaliação; e (6) comunicação. O fluxo de produção adotado foi composto pelas fases definidas por Peffers *et al.* (2007) e expandidas de acordo com as necessidades desse projeto, conforme apresentado na Figura 1.

Primeiramente, a etapa de **identificação e motivação do problema** foi composta pelo levantamento bibliográfico de materiais científicos com o enfoque em projeto de linguagem, metodologias de *software*, compiladores, engenharia de requisitos e teste de *software*. O levantamento objetivou a compreensão de conceitos de linguagens e da importância da formalização do seu processo de planejamento e desenvolvimento, investigando lacunas existentes na sua sistematização.

O passo subsequente foi a **definição dos objetivos para uma solução** que consistiu na avaliação de formas para resolver o problema identificado na etapa anterior por meio de um produto. Para isso, foram definidos requisitos essenciais para que o produto se tornasse uma solução viável para o problema. Assim, considerando esses requisitos, foi estabelecido, como solução, um modelo de processo de software para criação de linguagens.

Ademais, a etapa de **design e desenvolvimento** foi responsável pela construção do modelo de processo para a resolução do problema encontrado. Para isso, foi empregado o conhecimento teórico e prático obtidos a partir dos estudos e implementações realizadas anteriormente.

Posteriormente, a etapa de **demonstração** teve como finalidade aplicar o modelo n a construção de uma linguagem de programação. Dessa forma, foi possível implementar o código e elaborar a documentação por meio dos artefatos sugeridos pelo modelo de processo, possibilitando a avaliação preliminar do fluxo criado pelo produto.

Quanto à **avaliação**, ocorreram a validação, análise e a aplicação dos ajustes observados na etapa anterior. Isto é, houve alterações no modelo de processo a partir dos ajustes percebidos como necessários ao longo da execução prática do NILO.

Por fim, a etapa de **comunicação** objetivou a divulgação deste trabalho, o que compreendeu atividades como a sua elaboração e submissão para publicação em uma revista científica. A principal finalidade foi a disseminação dos resultados entre indivíduos interessados no desenvolvimento de linguagens de programação para uso do modelo de processo NILO.

Figura 1 - Fluxograma das etapas do desenvolvimento e da aplicação do NILO.

Fonte: A autora (2026) adaptado de Peffers *et al.* (2007).

3 DESENVOLVIMENTO

A etapa metodológica de identificação e motivação do problema realizada através da análise de trabalhos relacionados, elencados na Seção 6, destacou a lacuna relativa à inexistência de uma sistematização de um ciclo de vida iterativo para o planejamento, implementação e manutenção de linguagens de programação.

Por conseguinte, na etapa de definição dos objetivos para uma solução, foram delimitados os requisitos essenciais para que um produto fosse apresentado como a resolução do problema definido anteriormente (Quadro 1). Os requisitos foram derivados por meio da análise comparativa dos trabalhos correlatos, considerando as limitações recorrentes identificadas, em que os requisitos estabelecidos foram: (1) organizar fases iterativas e incrementais para remediar a ausência de formalização de fases associadas a artefatos padronizados de engenharia; (2) flexibilizar as fases e os artefatos, para que os indivíduos utilizadores possam adaptar o produto às suas necessidades, mas preservando ainda uma organização formal; (3) sugerir artefatos em cada fase; (4) abranger diferentes tipos de utilizadores podendo ser utilizado em contexto profissional e acadêmico; (5) incentivar a análise do problema que envolve a necessidade de desenvolvimento da linguagem de programação, identificando as demandas que justificam a sua existência antes de iniciar sua concepção; (6) instigar a definição das principais propriedades da linguagem de programação, utilizando um planejamento que facilite a compreensão das decisões por indivíduos diversos; (7) trazer opções de ferramentas que podem ser utilizadas na construção de linguagens; (8) possuir um enfoque também na documentação e planejamento de equipe; (9) dispor de uma fase destinada ao teste do tradutor da linguagem; (10) incluir exemplos práticos de como realizar cada fase; e (11) instruir a implementação prática da linguagem de programação, abstendo-se de impor detalhes operacionais e sem limitar o projeto a um tipo específico de processador de linguagem. Esses foram os principais requisitos entendidos como essenciais para que um produto resolvesse o problema apontado e se diferenciasse dos materiais já existentes.

Quadro 1 - Listagem de requisitos que devem estar presentes no produto.

N°	Nome	Descrição
R01	Iteratividade	Organizar em ciclos iterativos e incrementais.
R02	Flexibilidade	Adaptar fases e artefatos conforme a necessidade.
R03	Sugestão de Artefatos	Indicar modelos de documentos de saída para cada etapa.
R04	Abrangência	Abranger diferentes tipos de utilizadores.
R05	Alinhamento com o domínio	Incentivar a análise prévia do problema e da demanda pela linguagem.
R06	Definição das Características	Estimular o planejamento claro das propriedades fundamentais da linguagem.
R07	Recursos de Desenvolvimento	Trazer opções de ferramentas que podem ser utilizadas na construção de linguagens.
R08	Documentação e Planejamento	Possuir um enfoque também na documentação e planejamento de equipe.
R09	Testes	Incluir uma fase exclusiva para validação do tradutor da linguagem.
R10	Aplicação Prática	Adicionar exemplos práticos de como realizar cada fase.
R11	Orientação à implementação	Disponibilizar diretrizes de codificação sem imposição de detalhes operacionais.

Fonte: A autora (2026).

Posteriormente, com a definição dos requisitos necessários, buscou-se um produto que os contemplasse, como estratégia para suprir a carência de formalização de um ciclo iterativo para a criação de linguagens. O produto proposto foi um modelo de processo de *software* em virtude de sua orientação sobre as fases e incrementos que um projeto deve executar para realizar suas tarefas (Boehm, 1988). Portanto, a estruturação de suas fases foi disposta de maneira sistemática, se assemelhando ao que ocorre em outros contextos de desenvolvimento de *software*, apenas trazendo as especificidades necessárias. Dessa maneira,

tornou-se possível atender aos requisitos definidos e propor uma solução adequada para suprir a lacuna existente.

Após a definição dos requisitos fundamentais para resolução da lacuna observada, e a determinação de um modelo de processo como produto, foram tomadas decisões de projeto. Essas decisões estabelecem como cada requisito será atendido pelo modelo. Cabe destacar que alguns requisitos foram associados a decisões de projeto que implicaram na criação de fases, enquanto outros foram mapeados e incorporados como propriedades estruturais do modelo.

O Quadro 2 mapeia cada requisito definido (Quadro 1) para uma decisão de projeto: (1) na **iteratividade** (R01) definir um ciclo de vida dinâmico com fluxos de retorno interconectados para a garantia de fases iterativas e incrementais; (2) na **flexibilidade** (R02) orientar de forma clara os contextos em que cada artefato é mais adequado; (3) na **sugestão de artefatos** (R03) disponibilizar como devem ser os detalhes de cada artefato de saída, estabeleceu-se como necessário; (4) na **abrangência** (R04) explicar conceitos importantes de linguagens de programação, como gramática, tabelas de símbolos e técnicas de tradução de linguagens, visando o auxilio à compreensão e aplicação prática do modelo, considerando diferentes perfis de utilizadores; (5) no **alinhamento com o domínio** (R05), criar uma fase, denominada "Reconhecimento", com o enfoque na análise do domínio de um problema, foi definida como estratégia para incentivar a análise do problema; (6) na **definição das características** (R06) criar uma fase, denominada "Definição", com o enfoque na delimitação das principais características da linguagem, procurou instigar a definição das principais propriedades da linguagem; (7) no requisito **recursos de desenvolvimento** (R07) incentivar a autonomia do utilizador, evitando a imposição de um fluxo específico, foi pretendido ao trazer opções de ferramentas para a construção da linguagem; (8) na **documentação e planejamento** (R08), criar uma fase, denominada "Estruturação", com o enfoque na definição de características técnicas da linguagem, foi delimitada para organização do fluxo de desenvolvimento; (9) no requisito de **testes** (R09) criar uma fase, denominada "Teste", com o enfoque na avaliação e validação da linguagem, foi essencial para garantir a sua corretude; (10) na **aplicação prática** (R10) adicionar guias anexados a cada fase, foi estabelecido para fornecer um apoio na compreensão de como produzir cada artefato sugerido; e (11) na **orientação a implementação** (R11) criar uma fase,

denominada "Codificação", com o enfoque na implementação da linguagem, foi definido para instruir a implementação prática da linguagem de programação.

Quadro 2 - Mapeamento dos requisitos para decisões de projeto.

Nome do Requisito	**Decisão de Projeto**
Iteratividade	Definir um ciclo de vida iterativo.
Flexibilidade	Orientar os contextos em que cada artefato é mais adequado.
Sugestão de Artefatos	Detalhar os artefatos de saída de cada fase.
Abrangência	Apresentar conceitos importantes da criação de linguagens de programação.
Alinhamento com o domínio	Criar uma fase para análise do domínio de um problema que a linguagem busca resolver.
Definição das Características	Criar uma fase para definir a linguagem.
Recursos de Desenvolvimento	Incentivar a autonomia do utilizador, evitando a imposição de um fluxo específico.
Documentação e Planejamento	Criar uma fase para definição de características técnicas da linguagem.
Testes	Criar a fase para avaliação e validação da linguagem.
Aplicação Prática	Adicionar guias anexados a cada etapa.
Orientação à implementação	Criar a fase de implementação da linguagem.

Fonte: A autora (2026).

Com base nos requisitos e decisões de projeto, foi proposto o modelo de processo de desenvolvimento de linguagens de programação NILO. Nas subseções seguintes será discutida a estrutura da versão atual do modelo de processo, as fases definidas e a validação.

4 MODELO DE PROCESSO NILO

O NILO é um modelo de processo para o desenvolvimento de linguagens de programação, composto por cinco fases: reconhecimento, definição, estruturação, codificação e teste (Figura 2). Cada uma das fases apresenta sua finalidade, os artefatos a serem produzidos e uma atividade prática ilustrada por um exemplo de aplicação. As etapas iniciais — de reconhecimento, definição e estruturação — possuem o enfoque no projeto da linguagem, enquanto as de codificação e teste concentram-se na implementação. Mas, em conjunto, essas fases buscam apoiar o planejamento, a concepção e a evolução de linguagens de programação por meio de um processo iterativo e incremental. Além disso, é importante ressaltar que caso haja a necessidade de adição ou retirada de algum artefato, durante a realização de alguma fase, o modelo de processo é flexível.

Figura 2 - Fases do modelo de processo.

Fonte: A autora (2026).

4.1 FASES DO NILO

Nesta seção, cada uma das 5 (cinco) fases, listadas anteriormente, será descrita em detalhes, abordando seus objetivos, as atividades desenvolvidas e os artefatos produzidos.

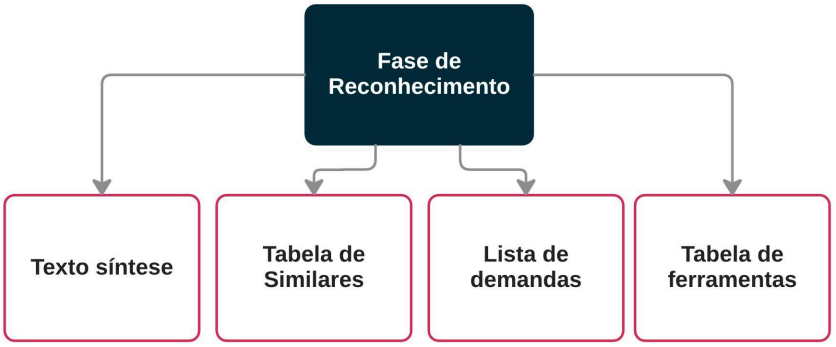
4.1.1 Reconhecimento

A primeira fase é chamada de **reconhecimento**, na qual se realiza a análise de um problema e/ou necessidade que justifica o desenvolvimento da linguagem de programação. Seu principal objetivo é a geração de artefatos que reúnem informações sobre as necessidades mapeadas para o projeto de linguagem.

Inicialmente, a equipe ou indivíduo, juntamente com os interessados na linguagem, deve responder a quatro questionamentos indicados pelo modelo de processo, com a finalidade de compreender o domínio e as necessidades do projeto. Esses questionamentos são: (1) que problema/necessidade busca suprir?; (2) o que sua linguagem teria de diferença crítica? não há algo já existente que supra suas necessidades?; (3) como essa problemática será resolvida?; (4) de que forma?.

Cada pergunta gera um artefato documental, que serve como registro formal das decisões tomadas, ajudando a assegurar que o desenvolvimento ocorra de forma alinhada às necessidades observadas e que elas possam ser gerenciadas durante o ciclo de vida da linguagem. A Figura 3 apresenta os 4 (quatro) artefatos a serem produzidos no reconhecimento: (1) texto síntese; (2) tabelas de similares; (3) lista de demandas; e (4) tabela das ferramentas.

Figura 3 - Artefatos gerados na fase de reconhecimento.

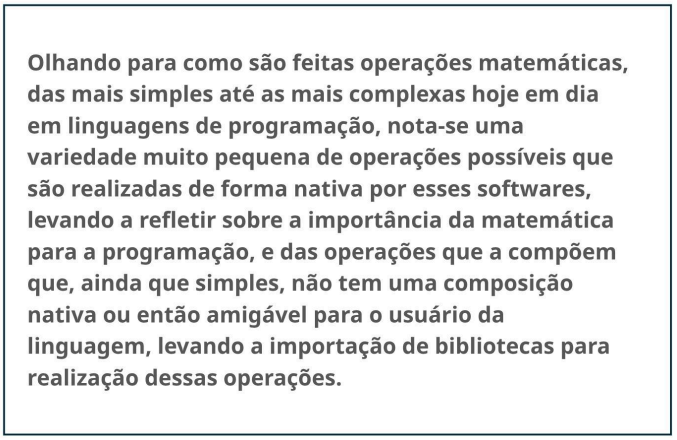


Fonte: A autora (2026).

O **Texto Síntese** é responsável pela descrição textual do problema que motivou a criação da linguagem. O artefato possui a finalidade de compreender os obstáculos gerados pela ausência de uma linguagem que supra as necessidades pontuadas. A Figura 4 demonstra uma versão de um texto síntese, que delimita um

problema que a linguagem a ser criada busca solucionar: “variedade muito pequena de operações matemáticas que são realizadas de forma nativa”.

Figura 4 - Exemplo de artefato de texto síntese simplificado.



Fonte: A autora (2026) adaptado de Vasconcelos (2026).

A **Tabela de Similares** possui o propósito de explicar a razão da formulação de uma nova linguagem, compreendendo se não existe algo que supra as necessidades. Para isso deve ser realizada a busca de linguagens similares, pontuando os diferenciais e os pontos em comum que o novo projeto de linguagem deve possuir para resolver o problema apresentado, assim como exemplificado na Figura 5.

Figura 5 - Exemplo de tabela de similares simplificada.

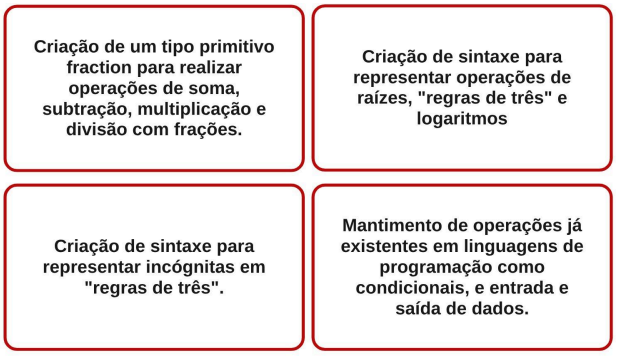
Linguagem	Descrição	Diferenciais
Julia	Linguagem moderna e eficiente focada na resolução de problemas matemáticos e científicos.	Soluções matemáticas propostas por Matty são feitas através de pacotes importados ou conjunto de operações primitivas.

Fonte: A autora (2026) adaptado de Vasconcelos (2026).

A **Lista de Demandas** é o artefato responsável por conter as definições das características que a linguagem deve ter, ou seja, quais as demandas que são necessárias para a linguagem cumprir seu propósito. A Figura 6 exibe um exemplo

de demandas listadas como necessárias para a resolução do problema delimitado na Figura 5.

Figura 6 - Exemplo de lista de demandas simplificada.



Fonte: A autora (2026) adaptado de Vasconcelos (2026).

Por fim, o artefato de **Tabela de Ferramentas** contém as ferramentas que serão empregadas para o desenvolvimento da linguagem, detalhando o papel de cada uma ao longo da implementação. Nesse sentido, a Figura 7 apresenta uma forma de organização da tabela, em que as colunas “ferramentas” e “funcionalidade”, em conjunto, cumprem o papel de listar as ferramentas a serem utilizadas para a implementação da linguagem.

Figura 7 - Exemplo de tabela de ferramentas simplificada.

Ferramenta	Funcionalidade
ANTLR	Elaboração da gramática em .g4 para operações existentes e novas propostas para em seguida gerar o lexer e o parser.
LLVM	Criação da representação intermediária da linguagem.

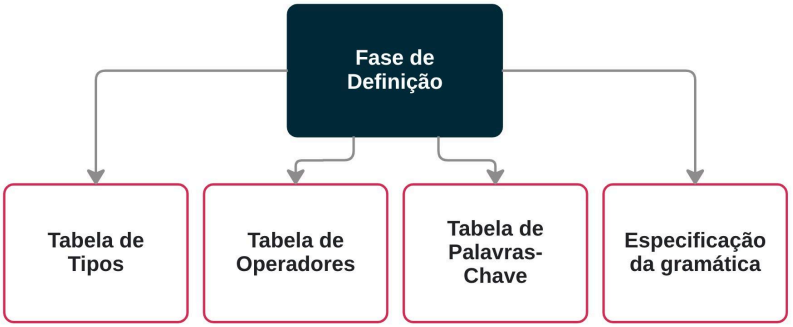
Fonte: A autora (2026) adaptado de Vasconcelos (2026).

4.1.2 Definição

A segunda fase do modelo de processo é a de Definição, na qual a principal finalidade é a tomada de decisões de projeto. Nessa etapa, partindo dos artefatos gerados na fase anterior, são definidas as principais características da linguagem,

tais como suas funcionalidades, tipos, palavras-chave e operadores. Ao fim da definição devem ser gerados artefatos que documentam todas as estruturas da linguagem (Figura 8): (1) tabela de tipos; (2) tabela de operadores; (3) tabela de palavras-chave; e (4) especificação da gramática.

Figura 8 - Artefatos gerados na fase de definição.



Fonte: A autora (2026).

A **Tabela de Tipos** descreve os tipos de dados contidos e uma descrição sobre o que deve ser armazenado. Porém, o artefato deve ser produzido apenas em casos que a linguagem necessite de uma separação explícita dos diferentes tipos de dados existentes. A Figura 9 mostra uma tabela com a definição dos tipos Decimal, Fraction e Integer.

Figura 9 - Exemplo de tabela de tipos simplificada.

Nome do tipo	Descrição
Decimal	Tipo numérico usado para representar números que contém partes fracionárias separadas por "."
Fraction	Tipo numérico usado para representar números fracionários.
Integer	Tipo numérico usado para representar números sem partes fracionárias.

Fonte: A autora (2026) adaptado de Vasconcelos (2026).

A **Tabela de Operadores** descreve quais os operadores e seus respectivos usos na linguagem (como +, -, * e /). Nessa perspectiva, os operadores “///”, “^” e “(|-|)”, representados na Figura 10, são descritos de tal forma a contribuir na documentação e compreensão da linguagem, sendo eles um exemplo de construção do artefato Tabela de Operadores.

Figura 10 - Exemplo de tabela de operadores simplificada.

Operador	Descrição
///	Declaração de uma fração.
^	Potenciação de tipos numéricos.
(| - |)	Operação de "regra de três" diretamente proporcional com tipos numéricos.

Fonte: A autora (2026) adaptado de Vasconcelos (2026).

O artefato **Tabela de Palavras-Chave** estabelece as palavras, para além dos tipos e operadores já definidos anteriormente, que possuem um significado particular na linguagem. Essas palavras não podem ser usadas em lugares como nome de variáveis, listas e funções (como *input*, *print*, *for* e *if*). Conforme a Figura 11, as colunas “palavras-chave”, “descrição” e “exemplo de uso”, são necessárias para a construção do artefato, sendo elas exemplificadas pelos termos “if/else”, “read” e “log”.

Figura 11 - Exemplo de tabela de palavras-chave simplificada.

Palavra-chave	Descrição	Exemplo de Uso
if/else	Declaração de um bloco condicional	if(x == y){...}else{...}
read	Realiza entrada de dados por parte do usuário	read;
log	Realiza a operação de logaritmo	log100

Fonte: A autora (2026) adaptado de Vasconcelos (2026).

A definição das funcionalidades completas da linguagem (como funções, condicionais e laços) são realizadas no artefato de **Especificação da Gramática**. A gramática representa a estrutura sintática das construções de uma linguagem de programação, estabelecendo regras que descrevem como os programas podem ser escritos. Sua construção frequentemente é representada por meio da notação Backus–Naur Form (BNF) ou de suas variações, amplamente utilizadas na especificação de linguagens de programação (Aho et al., 2008). Para a definição da

gramática, são utilizados os elementos definidos nos artefatos demonstrados anteriormente (como os tipos, operadores e palavras-chave). A Figura 12 demonstra a especificação de uma gramática, representada na notação BNF. Porém, o exemplo mostra apenas uma das formas de definir as regras de uma linguagem, sendo possível a utilização de outras abordagens.

Figura 12 - Exemplo de especificação da gramática na notação BNF.

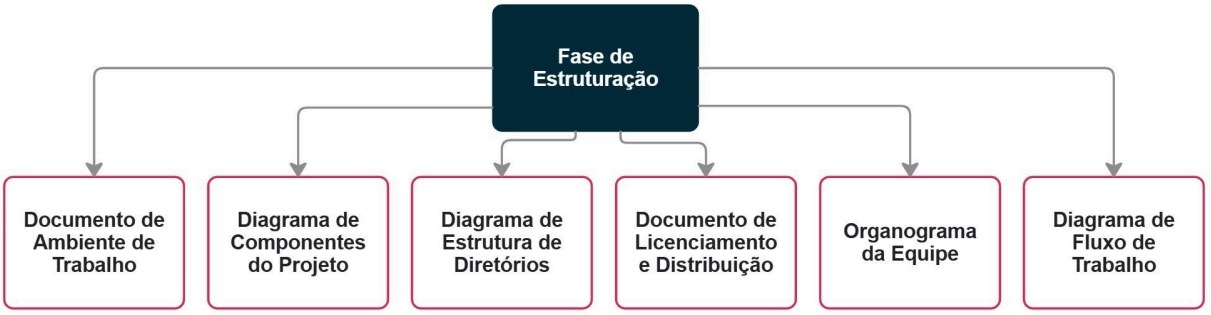
```
<regra_tres> ::= <identificador> "=" "(" <valor> "|" <valor> "-" "x"
"|" <valor> ")" ";"

<valor> ::= <numero> | <identificador>

<numero> ::= <inteiro> | <real>

<inteiro> ::= <digito> | <inteiro> <digito>

<real> ::= <inteiro> "." <inteiro>

<identificador> ::= <letra> | <identificador> <letra> |
<identificador> <digito> | <identificador> "_"

<letra> ::= "A" | ... | "Z" | "a" | ... | "z"

<digito> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

Fonte: A autora (2026).

4.1.3 Estruturação

A fase de estruturação é responsável pela organização da equipe, dos fluxos de desenvolvimento e das características técnicas que cercam a criação de uma linguagem, sendo a sua principal finalidade a manutenção a longo prazo. A Figura 13 mostra os 6 (seis) artefatos que podem ser produzidos nessa fase: (1) documento de ambiente de trabalho; (2) diagrama de componentes do projeto; (3) diagrama de estrutura de diretórios; (4) documento de licenciamento e distribuição; (5) organograma de equipe; e (6) diagrama de fluxo de trabalho.

Figura 13 - Artefatos gerados na fase de estruturação.



Fonte: A autora (2026).

O **Documento de Ambiente de Trabalho** possui o objetivo central de delimitar quais os recursos utilizados para estruturar o projeto. Isso significa que ele deve conter informações como o local de armazenamento e versionamento de artefatos, incluindo código, ambiente de desenvolvimento integrado (IDE), o sistema operacional utilizado e qualquer informação relevante para a construção do ambiente de produção da linguagem. A Figura 14 demonstra uma versão do artefato, exemplificando de que forma pode ser construída sua estrutura, a partir da definição da ferramenta e do seu propósito de utilização.

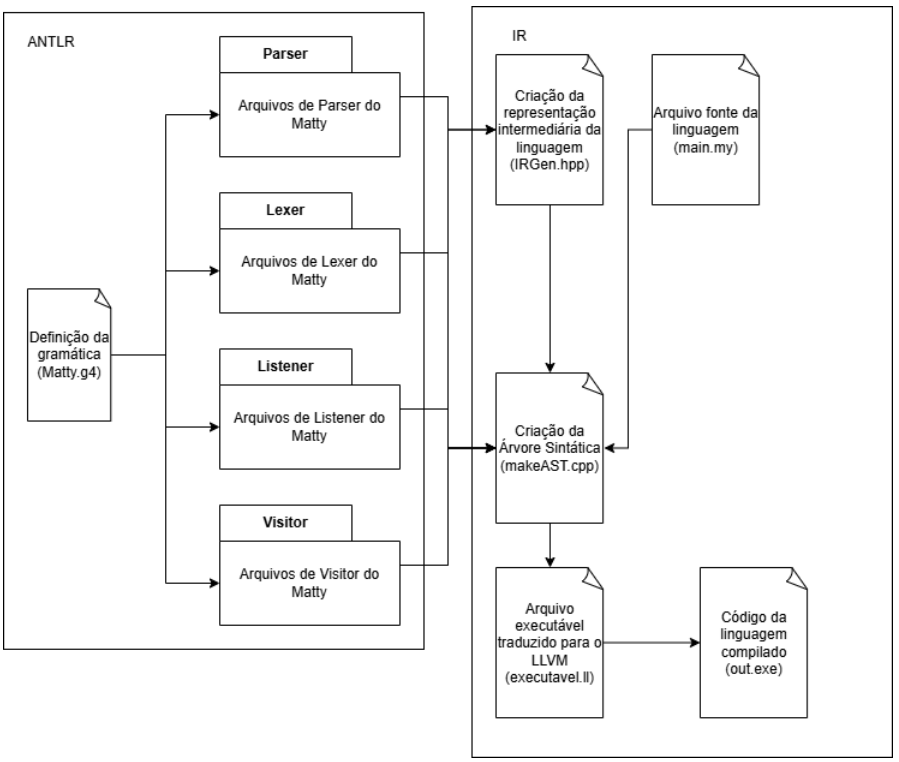
Figura 14 - Exemplo de documento de ambiente de trabalho simplificado.

Utilidade	Ferramenta
Ferramenta de versionamento e armazenamento de código	Github
Ambiente de Desenvolvimento Integrado (IDE)	Visual Studio Code

Fonte: A autora (2026) adaptado de Vasconcelos (2026).

O artefato **Diagrama de Componentes do Projeto** deve demonstrar o papel dos módulos da linguagem, esclarecendo suas funções e atribuições dentro do projeto de linguagem. A Figura 15, demonstra a comunicação entre os componentes responsáveis pelo funcionamento de uma linguagem compilada, em que as ferramentas e códigos internos, trocam informações para garantir o funcionamento da linguagem.

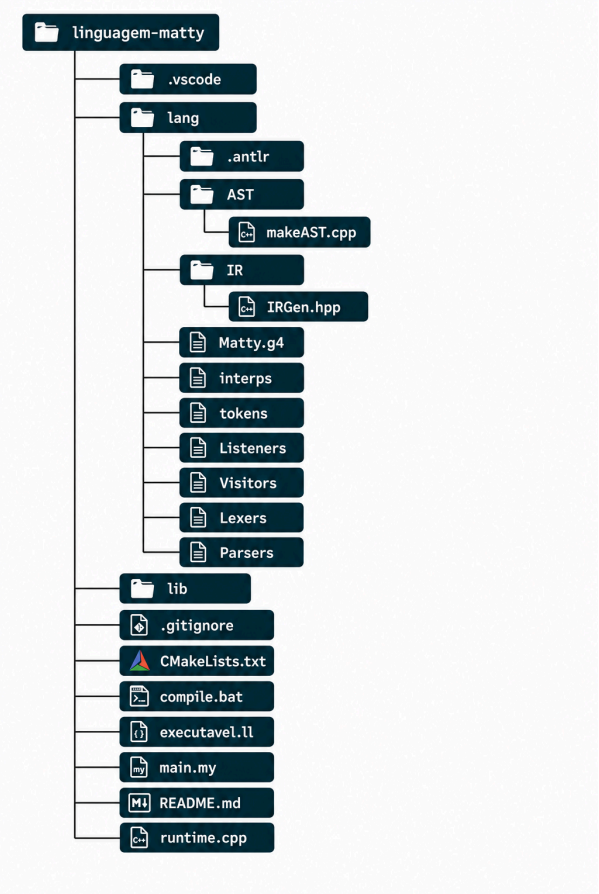
Figura 15 - Exemplo de diagrama de componentes do projeto.



Fonte: Vasconcelos (2026).

O artefato **Diagrama de Estrutura de Diretórios** tem o propósito de estruturar as pastas de arquivos, sua respectiva funcionalidade e o que ele irá armazenar. A Figura 16 demonstra uma versão do diagrama, apresentando os diretórios do projeto de forma ordenada, em que cada pasta e arquivo são partes constituintes da implementação da linguagem

Figura 16 - Exemplo de diagrama de estrutura de diretórios simplificado.



Fonte: A autora (2026) adaptado de Vasconcelos (2026).

O **Documento de Licenciamento e Distribuição** é indicado para a formalização de projetos de linguagem de grande escala. Nele, são definidas questões jurídicas e de disponibilidade do projeto, tal qual exemplificado na Figura 17.

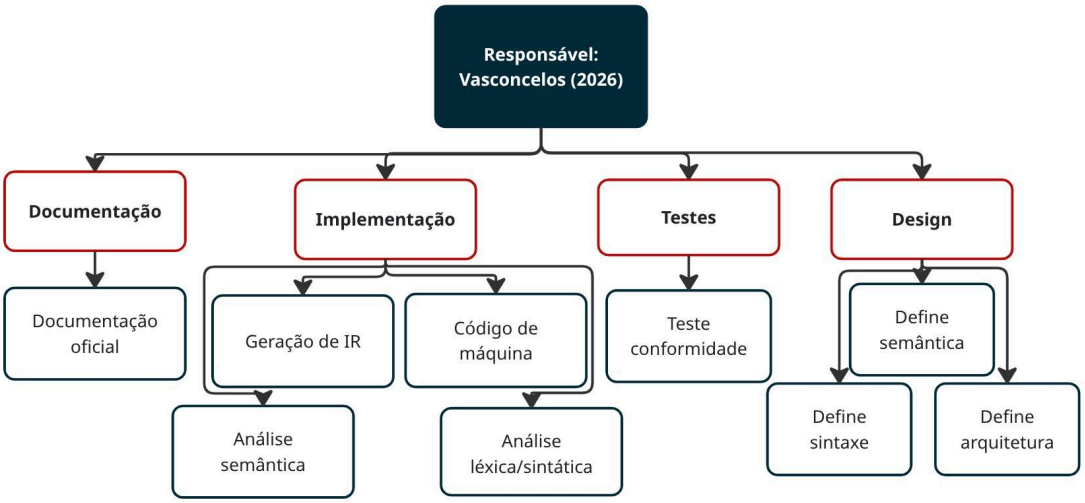
Figura 17 - Exemplo de documento de licenciamento e distribuição simplificado.

Identificação do Projeto
Nome: Matty
Versão: 1.0
Contexto de aplicação
Linguagem de programação voltada para a resolução de problemas matemáticos.
Modelo de Licenciamento
Código aberto com restrições. O código-fonte poderá ser consultado e utilizado para fins de estudo. Não é permitida a modificação direta do repositório principal.
Direitos e Restrições de Uso
Uso permitido para fins acadêmicos, educacionais e de pesquisa. Não é permitida a comercialização da linguagem sem autorização dos responsáveis.
Política de Distribuição
Não há previsão de distribuição comercial ou disponibilização de versões proprietárias.
Estratégia de Manutenção e Suporte
- Correção de erros identificados durante a evolução do projeto.
Fonte de Financiamento
O projeto é desenvolvido de forma independente, sem financiamento externo, patrocínio, comercialização ou recebimento de doações.

Fonte: A autora (2026).

O artefato **Organograma de Equipe** define o grupo de indivíduos e suas respectivas responsabilidades dentro do fluxo de construção da linguagem. O artefato se mostra ideal para projetos com equipes robustas, não indicado para linguagens desenvolvidas individualmente, uma vez que não há uma divisão de tarefas a ser formalizada. Algumas responsabilidades (documentação, implementação, testes e design) foram representadas na Figura 18, exemplificando uma alternativa de organograma de equipe.

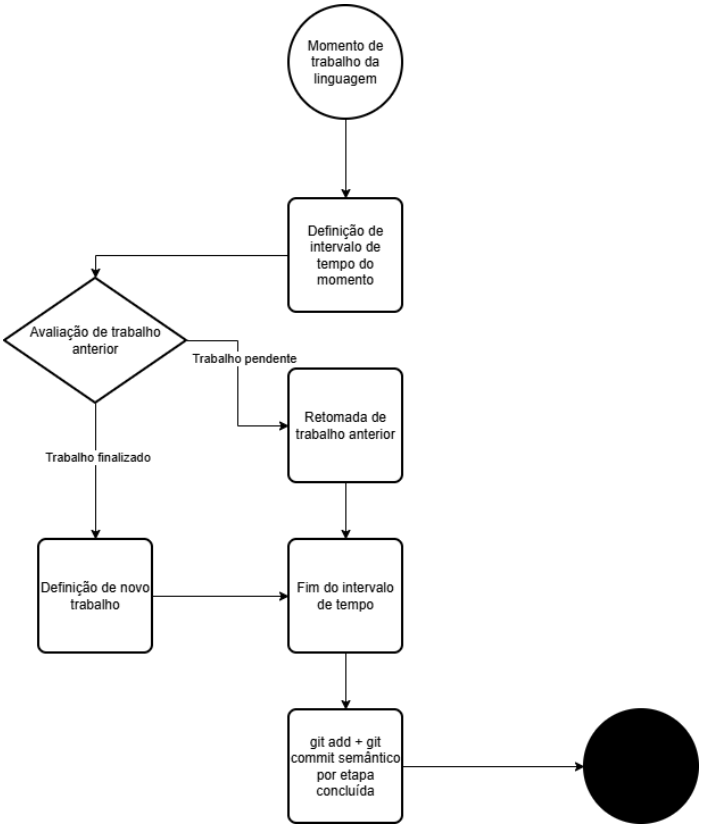
Figura 18 - Exemplo de Organograma de Equipe simplificado.



Fonte: A autora (2026).

O **Diagrama de Fluxo de Trabalho** define estratégias para a criação e o acréscimo de novas funcionalidades na linguagem, considerando desde a criação de uma tarefa até o encaminhamento dela para revisão por outro membro da equipe. A Figura 19 mostra um possível fluxo para o diagrama, em que durante um determinado período de tempo são trabalhadas funcionalidades já existentes ou novas implementações, até o momento de inserir o código no repositório de armazenamento e versionamento.

Figura 19 - Exemplo de diagrama de fluxo de trabalho simplificado.



Fonte: Vasconcelos (2026).

4.1.4 Codificação

A fase de codificação corresponde à transformação dos artefatos produzidos durante o planejamento em uma implementação funcional da linguagem, para a concretização dos requisitos e estruturas definidas previamente. O fluxo de desenvolvimento é definido pelas decisões documentadas na fase de estruturação

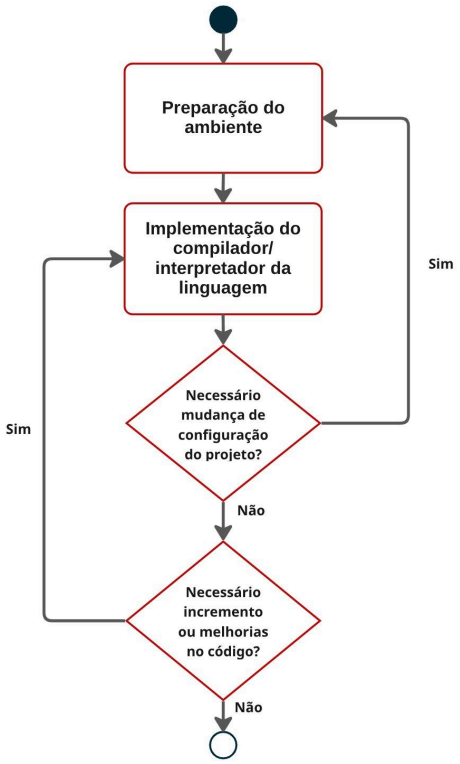
ou por processos previamente estabelecidos pela equipe responsável. O artefato produzido nessa fase é: (1) compilador/interpretador da linguagem (Figura 20).

Figura 20 - Artefatos gerados na fase de codificação.

Fonte: A autora (2026).

Nessa fase, realiza-se a implementação do compilador ou interpretador responsável pelo processamento da linguagem. Conforme representado na Figura 21, essa atividade é dividida em duas etapas principais. A primeira etapa é a preparação do ambiente, que consiste na instalação e configuração das ferramentas, bibliotecas e dependências necessárias para a implementação da linguagem. A segunda etapa consiste no desenvolvimento das funcionalidades da linguagem e caracteriza-se por um processo iterativo. Visto que, durante sua execução, podem surgir demandas que tornem necessário o retorno a etapas anteriores dessa fase. A necessidade de incorporar novas ferramentas, atualizar dependências ou reconfigurar o ambiente implica o retorno à etapa de preparação do ambiente. De forma semelhante, a correção de defeitos ou a implementação de novas funcionalidades demandam o retorno à etapa de desenvolvimento.

Figura 21 - Atividades para implementação de funcionalidades.

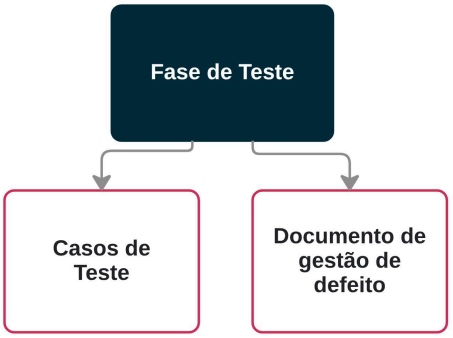


Fonte: A autora (2026).

4.1.5 Teste

A fase de teste tem como objetivo avaliar e validar as funcionalidades propostas para a linguagem, mapeando os erros encontrados para preservar a integridade da linguagem após implementações. Dessa forma, a etapa busca assegurar conformidade e completude funcional do projeto. Os artefatos produzidos são de código e documentais, que são: (1) Casos de Teste; e (2) Documento de Gestão de Defeito (Figura 22).

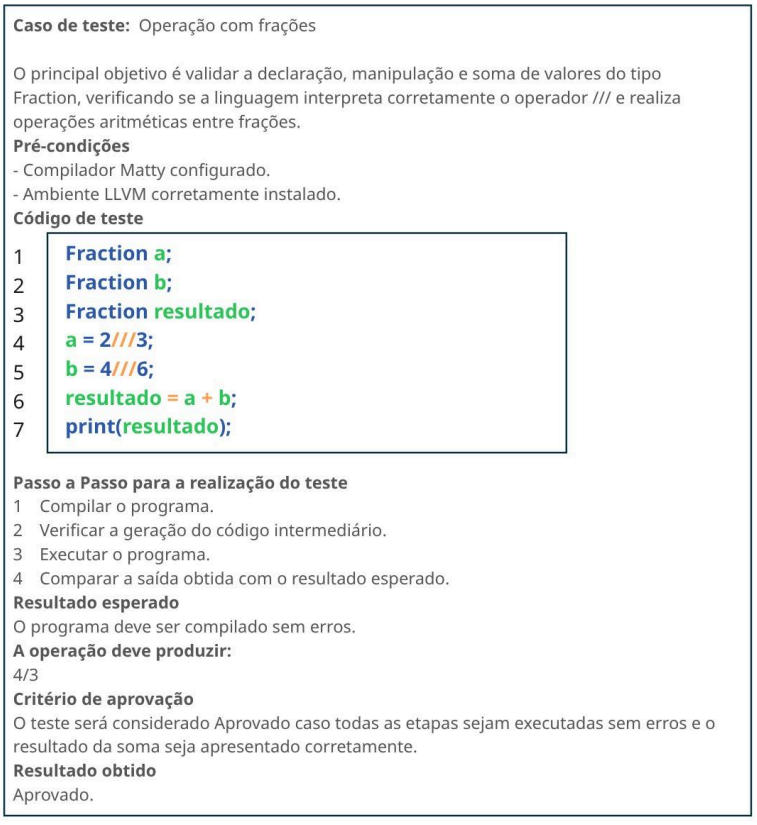
Figura 22 - Artefatos gerados na fase de teste.



Fonte: A autora (2026).

Os **Casos de Teste** devem ser produzidos de acordo com os requisitos e as funcionalidades previstas, de modo a orientar o desenvolvimento e garantir que os comportamentos esperados sejam validados. Esses casos podem ser representados por programas escritos na própria linguagem, executados manualmente, ou automatizados por meio de ferramentas de teste. Recomenda-se a elaboração de casos que contemplem tanto cenários de sucesso quanto cenários de exceção, permitindo verificar não apenas a correta execução das funcionalidades, mas também a capacidade da linguagem de identificar e tratar entradas inválidas e demais condições excepcionais. A Figura 23 exemplifica uma possível estrutura para a formalização documental de casos de testes, que devem ser refletidos em código e executados na linguagem. Dessa forma, é possível validar se o resultado esperado, explicitado no documento, foi alcançado.

Figura 23 - Exemplo de caso de teste.



Caso de teste: Operação com frações

O principal objetivo é validar a declaração, manipulação e soma de valores do tipo Fraction, verificando se a linguagem interpreta corretamente o operador /// e realiza operações aritméticas entre frações.

Pré-condições

- Compilador Matty configurado.
- Ambiente LLVM corretamente instalado.

Código de teste

```
1  Fraction a;
2  Fraction b;
3  Fraction resultado;
4  a = 2///3;
5  b = 4///6;
6  resultado = a + b;
7  print(resultado);
```

Passo a Passo para a realização do teste

1 Compilar o programa.
2 Verificar a geração do código intermediário.
3 Executar o programa.
4 Comparar a saída obtida com o resultado esperado.

Resultado esperado

O programa deve ser compilado sem erros.

A operação deve produzir:

4/3

Critério de aprovação

O teste será considerado Aprovado caso todas as etapas sejam executadas sem erros e o resultado da soma seja apresentado corretamente.

Resultado obtido

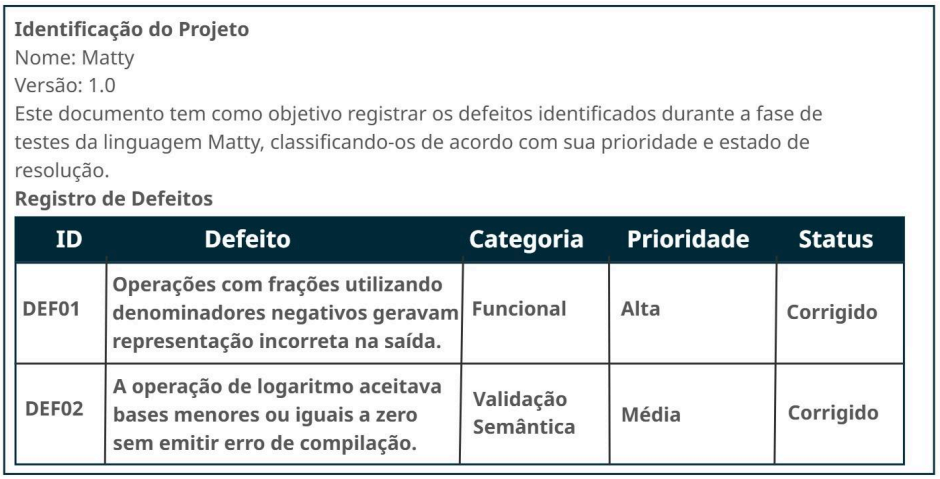
Aprovado.

Fonte: A autora (2026).

Enquanto isso, o **Documento de Gestão de Defeito** possui o propósito de mapear os erros encontrados, pois a gestão incorreta deles pode acarretar na

omissão da resolução do problema, gerando possíveis inconvenientes futuros. A Figura 24 demonstra o registro de problemas por meio do artefato, em que cada defeito tem um identificador, sua descrição, categoria, prioridade e estado atual.

Figura 24 - Exemplo de documento de gestão de defeito simplificado.



Identificação do Projeto
Nome: Matty
Versão: 1.0
Este documento tem como objetivo registrar os defeitos identificados durante a fase de testes da linguagem Matty, classificando-os de acordo com sua prioridade e estado de resolução.

Registro de Defeitos

ID	Defeito	Categoria	Prioridade	Status
DEF01	Operações com frações utilizando denominadores negativos geravam representação incorreta na saída.	Funcional	Alta	Corrigido
DEF02	A operação de logaritmo aceitava bases menores ou iguais a zero sem emitir erro de compilação.	Validação Semântica	Média	Corrigido

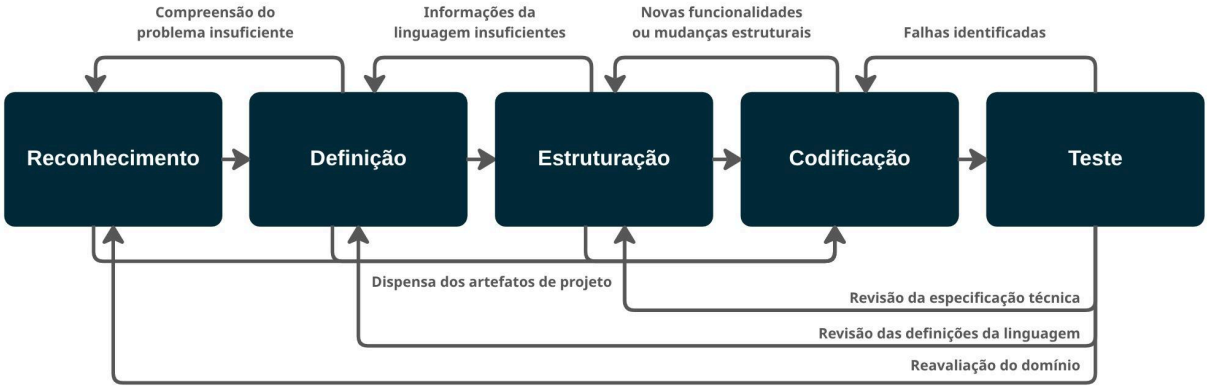
Fonte: A autora (2026).

4.3 CICLO DE VIDA NILO

Embora as fases sejam apresentadas em uma sequência lógica de execução, o modelo estabelece um fluxo cíclico. O processo permite retornos entre fases sempre que necessário, possibilitando revisões e refinamentos decorrentes de novas informações, decisões de projeto ou resultados obtidos durante o desenvolvimento.

A Figura 25 demonstra o ciclo de vida do Nilo, em que: (1) as **falhas identificadas** na execução dos testes permitem o retorno para a codificação; (2) a **criação de novas funcionalidades ou mudanças estruturais** cria a necessidade de voltar da fase de codificação para a estruturação; (3) as **informações da linguagem insuficientes** despertam demandas de regresso da estruturação para a definição; (4) a **compreensão insuficiente do problema** requer o retorno da definição para o reconhecimento; (5) as fases de reconhecimento, definição e estruturação permitem o avanço direto para a codificação em casos que o projeto de linguagem **dispense os artefatos sugeridos**; (6) a **revisão da especificação técnica**, a **verificação das definições da linguagem** e a **reavaliação do domínio** demandam o retorno da fase de teste para as de reconhecimento, definição e estruturação respectivamente.

Figura 25 - Ciclo de vida de uma linguagem no NILO.



Fonte: A autora (2026) adaptado de Glinz *et al.* (2024).

5 ESTUDO DE CASO

Um estudo de caso auxilia na realização da avaliação dos benefícios de métodos e ferramentas, mostrando os efeitos de uma tecnologia em uma situação típica — que não pode ser generalizada para todos os casos (Kitchenham, 1995). Com o objetivo de avaliar as fases e artefatos do modelo de processo NILO, adotou-se, por meio da concepção de uma linguagem de programação, um estudo de caso para permitir uma investigação empírica sobre a viabilidade do modelo de processo proposto dentro de um contexto real e delimitado. Dessa forma, a seguir discute-se como foi realizado o estudo e quais os resultados obtidos.

A linguagem de programação utilizada para a realização do estudo de caso foi o NiloScript[1], que se caracteriza como uma linguagem de propósito geral (GPL - *general purpose language*), ou seja, ela presta suporte a diferentes domínios. O NiloScript possui uma estrutura sintática em português, que adota a compilação como forma de tradução. Isso significa que ele possui estruturas que suprem necessidades em diferentes domínios e se utiliza de uma linguagem familiar para o programador para facilitar o processo de compreensão.

A criação do NiloScript, assim representado na Figura 26, ocorreu em um ciclo de vida iterativo. A linguagem foi desenvolvida de forma incremental, impactando também em melhorias para o modelo de processo. Isso significa que, após a construção dos artefatos do NiloScript, eram documentados problemas observados ou melhorias desejadas e, ao fim, eram aplicadas as alterações necessárias no NILO. Após a aplicação das mudanças observadas, as melhorias eram empregadas na linguagem NiloScript, de forma que as correções no modelo de processo fossem sempre validadas.

Figura 26 - Ciclo de criação da linguagem NiloScript.

[1] Disponível em: https://github.com/namariaa/Nilo

Fonte: A autora (2026).

Portanto, nas subseções seguintes será abordado como foi a execução das fases do NILO. Além disso, será discutido um exemplo de artefato produzido para cada uma das fases do modelo de processo. Mesmo com apenas cinco artefatos exibidos, todos foram produzidos e estão disponíveis no repositório GitHub[2] da linguagem. Ao fim, serão listadas as melhorias e problemas encontrados durante o desenvolvimento, explicando assim como se refletiram os incrementos no NILO.

5.1 FASE DE RECONHECIMENTO

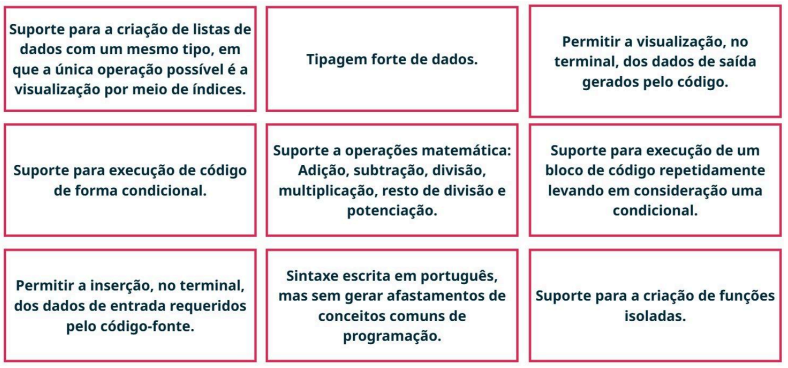
Na fase de reconhecimento, é fundamental a realização da análise de um problema/necessidade observada que leva a imperatividade de desenvolvimento de uma linguagem de programação. Desse modo, a execução do reconhecimento foi iniciada com a delimitação da necessidade que envolveu a criação da linguagem de programação NiloScript. No caso do NiloScript, a necessidade observada foi avaliar a operabilidade do modelo de processo NILO para a criação de linguagens. Além disso, foi estabelecido, como propriedade da linguagem, a utilização de

[2] Disponível em: https://github.com/namariaa/Nilo/tree/main/NiloScript/docs/NILO

palavras-chave em português, mantendo as funcionalidades comumente presente em outras linguagens de programação como laços de iteração e condicionais.

A Figura 27 demonstra o artefato de listagem de demandas, visto na subseção 4.1.1, em que foram apresentadas características necessárias para a linguagem ser utilizada como estudo de caso.

Figura 27 - Listagem de demandas do NiloScript.



Fonte: A autora (2026).

5.2 FASE DE DEFINIÇÃO

A tomada de decisões de projeto é a principal finalidade da fase de definição, sendo nela delimitada quais as características presentes na linguagem para cumprir a necessidade identificada no reconhecimento. Por isso, no NiloScript foram produzidos todos os artefatos sugeridos no modelo de processo com o objetivo de validá-los. A Figura 28 demonstra a especificação da gramática NiloScript, no formato reconhecido pela ferramenta ANTLR, que será explicada posteriormente na subseção 5.4. O elemento base (raiz) da gramática é um programa (*program*), uma sequência de declarações (*stmt*) que inclui: saída de dados (print); atribuição (*assignment*); laço de iteração (*loop*); condicional (*inCase*); função (*function*); chamada de funções (*functionCall*); lista de dados (*list*); expressões matemáticas (*expression*); e comentários. Em conjunto, elas estruturam o NiloScript e garantem o seu funcionamento na etapa de construção do seu código.

Por fim, é importante ressaltar que durante a execução desta fase, foi observada a ausência de uma forma de listar todas as palavras que possuíam um significado específico na linguagem, sendo essa lacuna documentada como ajuste necessário.

Figura 28 - Especificação da gramática NiloScript simplificada.

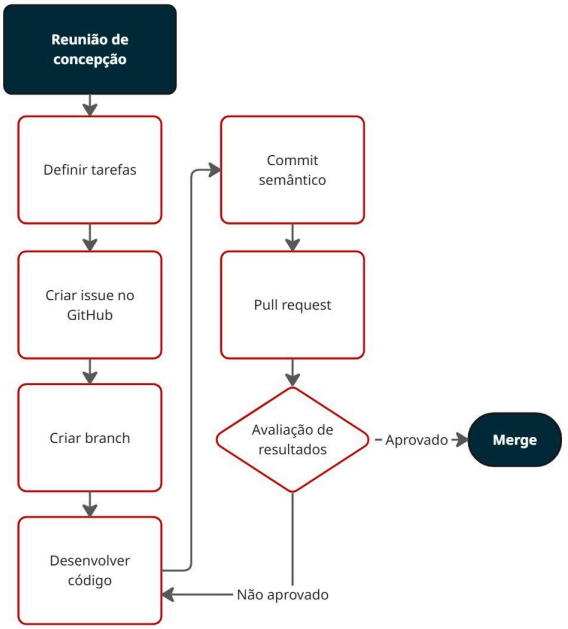
```
1   grammar NiloScript;
2
3   program : (stmt)+ EOF;
4   stmt : print SC
5        | assignment SC
6        | loop
7        | inCase
8        | function
9        | functionCall SC
10       | list SC
11       | expression SC
12       | COMMENT;
```

Fonte: A autora (2026).

5.3 FASE DE ESTRUTURAÇÃO

Para que seja possível ordenar os indivíduos e as características técnicas para a criação de uma linguagem, a fase de estruturação foi empregada no NiloScript. Assim, os seis artefatos da estruturação, especificados na seção 4.1.3, foram produzidos e avaliados. O diagrama de fluxo de trabalho (Figura 29) foi um desses artefatos, em que nele a partir da reunião de concepção são definidas atividades necessárias para incrementos da linguagem. Essas atividades, no NiloScript, eram registradas e adicionadas no repositório de armazenamento e versionamento, para que ao fim fosse possível avaliar os resultados e incrementá-los ao todo. Ao fim da criação de todos os artefatos, foi notada a restrição de meios de formalização do licenciamento da linguagem, que limitam a documentação de projetos de linguagens confidenciais ou reservadas. Além disso, a manutenção do código da linguagem, em casos de mudança de equipe de desenvolvimento, se tornava mais difícil pela ausência de uma representação visual explicativa da arquitetura de pastas de armazenamento de código.

Figura 29 - Diagrama de fluxo de trabalho do NiloScript.



Fonte: A autora (2026).

5.4 FASE DE CODIFICAÇÃO

A fim de gerar o código necessário para a produção da linguagem, a fase de codificação foi executada na construção do NiloScript. Nessa etapa, foi construído o compilador, para traduzir e processar os códigos-fonte da linguagem.

Todavia, antes de explicar sobre os artefatos de código produzidos nessa fase, é necessário compreender quais as etapas envolvem a concepção de um compilador. Um compilador, programa que recebe como entrada um código-fonte em uma linguagem de programação de alto nível e o traduz para uma linguagem objeto de mais baixo nível, possui seis etapas (Figura 30) no total: análise léxica, análise sintática, análise semântica, geração de código intermediário, otimização e geração de código (Aho et al., 2008). Juntas, as etapas são responsáveis por subdividir o código-fonte em partes indivisíveis, chamadas de tokens, e impor uma estrutura gramatical a eles. Em seguida, o fluxo cria uma representação intermediária que, ao fim, é transformada em um programa objeto, que no caso do NiloScript, é um executável. A análise léxica é responsável pela transformação do código-fonte em *tokens*, os quais são usados pela análise sintática para criar uma representação em árvore. A representação em árvore mostra a estrutura gramatical da sequência de *tokens*, que são verificados pela análise semântica se possuem sentido. Logo após, o gerador de código intermediário transforma a estrutura em árvore em um programa

para uma máquina abstrata utilizada para facilitar a tradução para a máquina alvo, que na etapa seguinte da compilação é otimizada. Por fim, o gerador de código de máquina transforma a representação intermediária otimizada em instruções de máquina de alguma arquitetura (Aho et al., 2008). Dessa forma, ao optar pela escolha de um compilador como processador do NiloScript, foi necessária a implementação de todas essas etapas. Assim, serão discutidos a seguir a construção do compilador Niloscript.

Figura 30 - Etapas de um compilador.

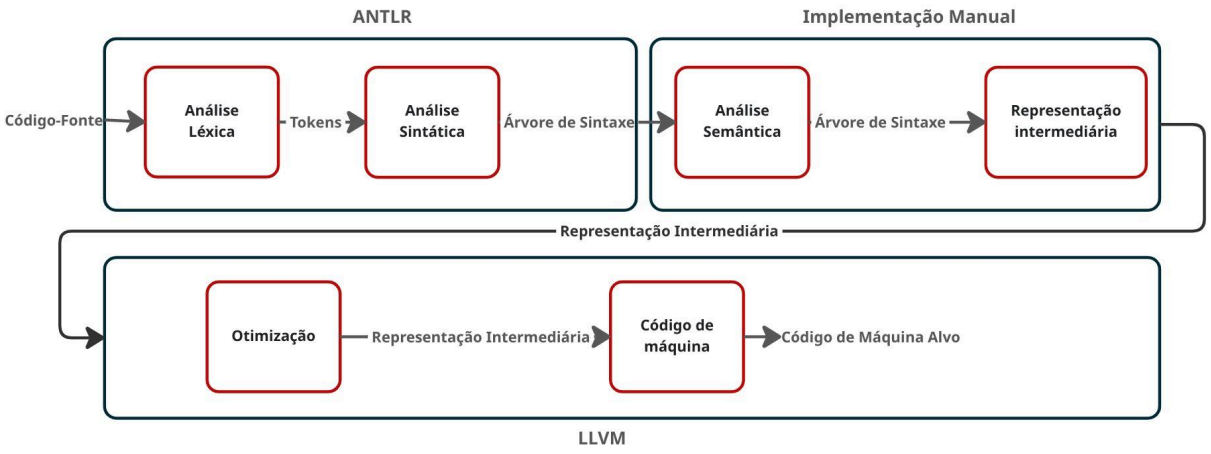
Fonte: A autora (2026) adaptado de Aho et al. (2008).

Inicialmente, para a construção do compilador, foram desenvolvidos os componentes responsáveis pelas análises léxica e sintática. Para isso, utilizou-se a gramática do NiloScript, previamente especificada na fase de definição e documentada no artefato de especificação da gramática, apresentado na subseção 5.2 e exemplificado na Figura 28. Nessa fase, a especificação foi utilizada como

entrada para o ANTLR, ferramenta capaz de gerar automaticamente analisadores léxicos e sintáticos a partir de uma gramática e de fornecer os mecanismos necessários à construção de árvores de análise sintática durante o processamento do código-fonte (PARR, 2013). Dessa forma, as regras gramaticais previamente definidas foram empregadas na geração dos componentes responsáveis pelo reconhecimento da estrutura léxica e sintática do NiloScript, concretizando, no compilador, as definições estabelecidas anteriormente.

Para as etapas seguintes, utilizou-se outra tecnologia, o LLVM. O LLVM é uma infraestrutura de compiladores reutilizáveis, produzido em C++. Ele foi projetado com o propósito de desenvolver técnicas de compilação estática e dinâmica (LLVM Project, 2026). No contexto do NiloScript, a infraestrutura foi utilizada para automatizar as etapas de otimização de representação intermediária e geração de código de máquina para diferentes arquiteturas. Para isso, foi necessária a construção de um gerador de representação intermediária que possibilitasse a associação da árvore sintática, construída pelo ANTLR, e a infraestrutura LLVM. A construção do gerador de representação intermediária foi realizada por meio de instruções de suporte fornecidas pelo LLVM, possibilitando a construção de uma ponte de comunicação clara. Assim, o compilador foi desenvolvido tornando possível o processamento de códigos-fonte escritos em NiloScript e sua tradução para uma representação executável no ambiente de destino. A Figura 31 apresenta o fluxo de construção do compilador, destacando o papel das ferramentas empregadas nas diferentes etapas de seu desenvolvimento.

Figura 31 - Fluxo das etapas de construção do compilador do NiloScript.

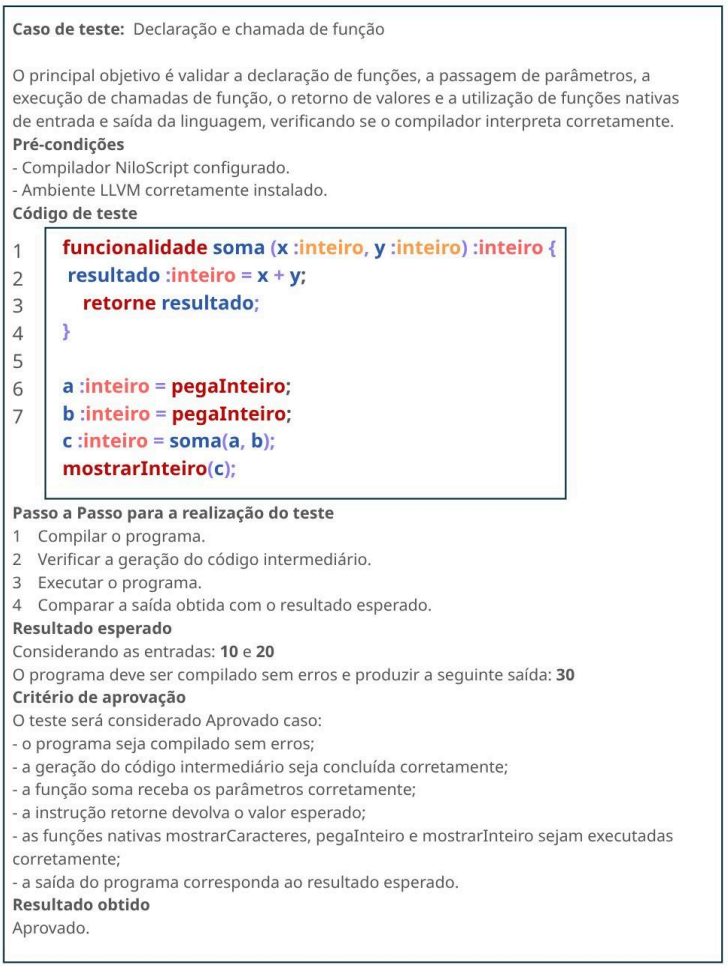


Fonte: A autora (2026).

5.5 FASE DE TESTE

Com o propósito de validar as funcionalidades da linguagem, executou-se a última fase do modelo de processo, o teste. Inicialmente, foram criados casos de testes para a linguagem, que envolveram a construção de códigos-fonte que foram documentados de forma que indicassem o resultado esperado, os critérios de aprovação e o passo a passo para realização do teste, representado na Figura 32. Para garantir a confiabilidade da linguagem foram criados casos de testes que envolvessem cenários de sucesso e falha para cada funcionalidade. Ao fim, foram obtidos 49 casos de testes que avaliaram e validaram o funcionamento de todas as funcionalidades da linguagem.

Figura 32 - Caso de teste de declaração e chamada de função do NiloScript.



Caso de teste: Declaração e chamada de função

O principal objetivo é validar a declaração de funções, a passagem de parâmetros, a execução de chamadas de função, o retorno de valores e a utilização de funções nativas de entrada e saída da linguagem, verificando se o compilador interpreta corretamente.

Pré-condições

- Compilador NiloScript configurado.
- Ambiente LLVM corretamente instalado.

Código de teste

```
1  funcionalidade soma (x :inteiro, y :inteiro) :inteiro {
2   resultado :inteiro = x + y;
3     retorne resultado;
4  }
5
6  a :inteiro = pegaInteiro;
7  b :inteiro = pegaInteiro;
   c :inteiro = soma(a, b);
   mostrarInteiro(c);
```

Passo a Passo para a realização do teste

1 Compilar o programa.
2 Verificar a geração do código intermediário.
3 Executar o programa.
4 Comparar a saída obtida com o resultado esperado.

Resultado esperado

Considerando as entradas: **10** e **20**
O programa deve ser compilado sem erros e produzir a seguinte saída: **30**

Critério de aprovação

O teste será considerado Aprovado caso:
- o programa seja compilado sem erros;
- a geração do código intermediário seja concluída corretamente;
- a função soma receba os parâmetros corretamente;
- a instrução retorne devolva o valor esperado;
- as funções nativas mostrarCaracteres, pegaInteiro e mostrarInteiro sejam executadas corretamente;
- a saída do programa corresponda ao resultado esperado.

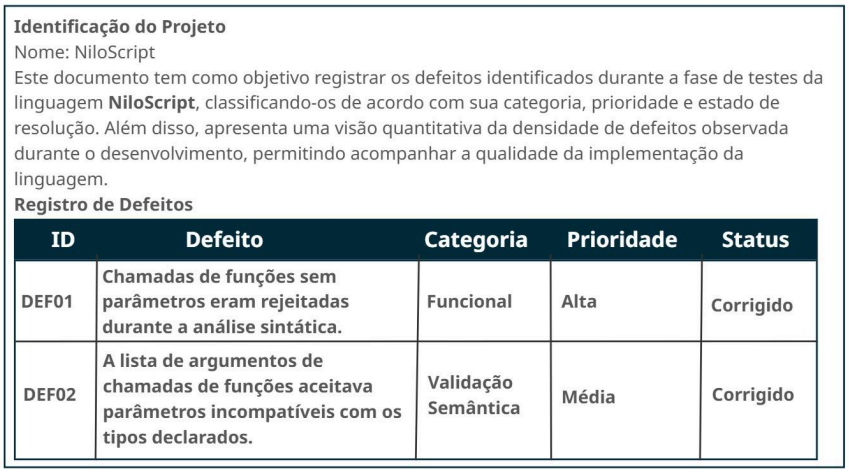
Resultado obtido

Aprovado.

Fonte: A autora (2026).

Durante a execução dos testes foram observados problemas no NiloScript. Então, para garantir que após a execução dos testes todos os defeitos encontrados fossem corrigidos e não se repetissem, criou-se o segundo e último artefato indicado na fase de teste, o documento de gestão de defeito. Estruturalmente, tal como representado na Figura 33, o documento relatou os defeitos e os registrou em uma tabela. Assim foi possível garantir a rastreabilidade de cada defeito, permitindo, de forma incremental, a melhoria da linguagem.

Figura 33 - Documento de gestão de defeitos simplificado do NiloScript.



Identificação do Projeto
Nome: NiloScript
Este documento tem como objetivo registrar os defeitos identificados durante a fase de testes da linguagem **NiloScript**, classificando-os de acordo com sua categoria, prioridade e estado de resolução. Além disso, apresenta uma visão quantitativa da densidade de defeitos observada durante o desenvolvimento, permitindo acompanhar a qualidade da implementação da linguagem.

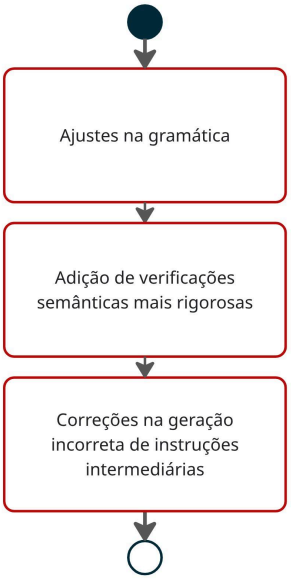
Registro de Defeitos

ID	Defeito	Categoria	Prioridade	Status
DEF01	Chamadas de funções sem parâmetros eram rejeitadas durante a análise sintática.	Funcional	Alta	Corrigido
DEF02	A lista de argumentos de chamadas de funções aceitava parâmetros incompatíveis com os tipos declarados.	Validação Semântica	Média	Corrigido

Fonte: A autora (2026).

Com a criação e execução dos casos de testes foram encontrados e documentados um total de 25 erros relativos a problemas na gramática da linguagem, problemas semânticos e de geração de representação intermediária. Isso tornou necessário a execução de 3 etapas de correção, que se centraram em: ajustes na gramática, adição de verificações semânticas mais rigorosas e correções na geração de algumas instruções intermediárias, assim representado na Figura 34. Após as etapas de correção, todos os casos de teste passaram a produzir os resultados esperados.

Figura 34 - Etapas de correções realizadas no NiloScript após testes.



Fonte: A autora (2026).

5.6 APLICAÇÃO DE MELHORIAS

A execução prática do estudo de caso permitiu avaliar a eficácia do modelo de processo NILO. A aplicação do modelo evidenciou o caráter iterativo e incremental do modelo, por meio da construção incremental das funcionalidades e documentações do NiloScript. Problemas observados na linguagem foram corrigidos com o retorno ou avanço nas etapas do processo. Além disso, o estudo de caso demonstrou a utilidade de todos os artefatos, ao contemplar desde a análise do problema/necessidade e planejamento das funcionalidades até a implementação e validação da linguagem através de testes. No entanto, o processo revelou também lacunas que prejudicavam a eficiência da metodologia em cenários reais, motivando rodadas de aprimoramento técnico.

A análise crítica dos dados colhidos no desenvolvimento do NiloScript evidenciou que: (1) a tabela de estruturas da linguagem não englobava de forma total todas as palavras reservadas da linguagem; (2) em casos de projetos de linguagem comerciais, ou até mesmo de licenças específicas, não havia um artefato que tratava sobre o licenciamento e disponibilização dessa linguagem; e (3) não havia artefato que simbolizasse estruturalmente o projeto para facilitar a adição de novas funcionalidades no local correto. Por isso, viu-se a necessidade de adição de três artefatos ao modelo: a tabela de palavras-chave, documento de licenciamento e distribuição e o diagrama de estrutura de diretórios. Além disso, com a refatoração

de código e a realização de testes, observou-se a necessidade de propor a criação de códigos-fonte escritos na linguagem em desenvolvimento, com o objetivo de testar cenários de erro e sucesso de cada funcionalidade da linguagem, garantindo que o código produzido consiga tratar diferentes fluxos, aumentando assim a segurança e confiabilidade da linguagem. Vale destacar que todas as requisições de mudanças observadas e analisadas foram aplicadas na versão atual do modelo de processo, trazendo melhorias à sua completude, pelo incremento de novos artefatos.

Por fim, é importante ressaltar que a linguagem evoluiu ao longo da pesquisa, desempenhando o papel de uma implementação de referência. Segundo Hogan e Newton (2015), uma implementação de referência pode ser usada para verificar a viabilidade do processo e servir como padrão de como utilizar o produto. Com a finalidade de melhorar a compreensão de como utilizar o modelo de processo, o NiloScript foi usado como exemplo prático de implementação. Dessa forma, além de validar o modelo, a linguagem NiloScript passou a constituir um exemplo de aplicação prática do NILO.

6 TRABALHOS RELACIONADOS

Com o avanço contínuo do desenvolvimento de software, diversos artefatos foram concebidos com o objetivo de apoiar e organizar o fluxo de criação e implementação. Todavia, esses trabalhos, geralmente, não são específicos para linguagens de programação. Mesmo quando focados nesse campo não contemplam a especificação, documentação, implementação e avaliação. A seguir são apresentados quatro trabalhos relacionados à criação de linguagens de programação com a finalidade de auxiliar na tomada de decisões para resolver o problema anteriormente identificado, a ausência de organização sistemática no fluxo de planejamento e implementação.

Inicialmente, Mernik, Heering e Sloane (2005) no seu trabalho *When and How to Develop Domain-Specific Languages* desenvolvem e associam padrões metodológicos, que são possíveis soluções testadas e reutilizáveis para resolução de problemas recorrentes, para cada fase, identificadas por eles, presentes no desenvolvimento de linguagens de domínio específico (DSL). As etapas delimitadas foram: (1) **decisão**, que partindo da listagem de padrões é determinada se há uma necessidade indispensável de criar uma nova DSL; (2) **análise** do domínio do problema e do conhecimento reunido acerca desse domínio; (3) **projeto** da linguagem a ser desenvolvida, partindo de padrões definidos pelos autores para a decisão de aproveitar recursos de linguagens já existentes, estender uma linguagem já existente ou criar uma do zero; (4) **implementação** que envolve a escolha da abordagem de desenvolvimento da linguagem de forma mais adequada para o contexto. O trabalho dos autores se assemelha com esse trabalho em sua abordagem de considerar a concepção de uma linguagem para além apenas de sua implementação, mas também levando em conta a análise e decisão que antecede a implementação. Porém, duas principais diferenças se evidenciam: a falta da especificação de artefatos a serem produzidos e a restrição para linguagens de domínio específico. O modelo de processo NILO sugere a produção de artefatos ao longo de suas fases e é mais abrangente quanto ao tipo de linguagens, pois também pode ser utilizada na implementação de linguagens de propósito geral (GPL).

Em *Crafting Interpreters*, Nystrom (2021) aborda de forma prática o passo a passo para a criação e a implementação de linguagens de programação.. Ao longo das páginas do livro, o autor, à medida que explica sobre a construção de cada parte

da linguagem, apresenta os conceitos que envolvem cada etapa, focando logo em seguida na implementação. Como ponto de convergência com o presente trabalho, se encontram demonstrações práticas que exemplificam os conceitos e as exposições dos conteúdos. Entretanto, o livro se distancia por sua essência de guia com etapas sistemáticas, não sendo tão flexível na escolha estrutural da linguagem. Além disso, o maior enfoque é na implementação, havendo uma distância na documentação, análise e planejamento da linguagem.

No terceiro trabalho relacionado, *Notable design patterns for domain-specific languages*, Spinellis (2001) discute acerca dos padrões de projeto mais recorrentes para projetar e implementar linguagens de domínios específicos. O trabalho do autor incentiva o projetista de DSLs a compreender as consequências e as alternativas de implementações disponíveis ao se optar pela escolha de um padrão de projeto de linguagem específico. Esse incentivo ao planejamento da linguagem, possibilita a escolha da forma de implementação partindo da apresentação de alternativas, sendo esse o ponto em comum compartilhado com o NILO. Apesar disso, a proposta de Spinellis (2001) se difere do NILO pela ausência de etapas iterativas e incrementais para a construção de linguagens. Há também a falta de sugestão de artefatos documentais e de código. Além disso, o NILO traz ideias para o projeto da linguagem, abordando sua implementação e avaliação, aspecto que o autor do trabalho relacionado em discussão não aborda.

Em *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*, Voelter et al. (2013) discute a respeito da criação de linguagens específicas de domínio, relacionando aspectos de projeto, implementação e engenharia de *software* com DSLs. Como semelhanças podemos destacar a abordagem de temas como definição, desenvolvimento e testes de linguagens de programação. Todavia, o trabalho, embora apresente técnicas e práticas para o desenvolvimento de DSLs, não propõe um modelo de processo estruturado.

Os trabalhos apresentados nesta seção mostram que os autores reconhecem que a concepção e manutenção de linguagens de programação devem ser conduzidas em um fluxo evolutivo focado na organização, planejamento e execução de atividades de forma contínua. Todavia, os trabalhos relacionados não propõem fases associadas a artefatos, comumente padronizados na área de engenharia de software. Essa limitação dificulta a padronização do processo de desenvolvimento, justificando a necessidade de um modelo que associe fases a artefatos

documentais. Diante disso, o NILO busca estruturar e formalizar fases e artefatos documentais e de código, propondo etapas abrangentes que atendam às diferentes necessidades e equipes envolvidas.

As diferenças entre os trabalhos relacionados e o NILO não indicam que não houve o aproveitamento de algumas de suas características. Mernik, Heering e Sloane (2005), contribuíram na melhoria e incremento de formas de decisões. Enquanto Nystrom (2021) e Voelter et al. (2013) contribuíram na definição estrutural do projeto. Por último, Spinellis (2001) contribuiu na geração de importantes contribuições para as formas de desenvolvimento. Portanto, esse estudo reforça a necessidade de desenvolvimento de uma formalização de etapas de desenvolvimento, inspirando-se nos trabalhos trazidos da área de desenvolvimento de linguagens de programação.

7 CONSIDERAÇÕES FINAIS

O modelo de processo NILO foi criado para prestar suporte a indivíduos e/ou equipes no desenvolvimento de linguagens de programação. Com o intuito de auxílio de implementação, a última versão do modelo de processo conta com fases delimitadas e artefatos de suporte. Dessa forma, o modelo de processo atua como recurso complementar estruturado, corroborando para a criação e evolução de linguagens de programação.

Mediante o exposto, é importante ressaltar que os principais resultados obtidos são: (1) modelo de processo NILO com estrutura adaptável; e (2) linguagem NiloScript como validador da eficácia do modelo de processo e como referência de implementação documental e de código.

Portanto, pode-se concluir que o trabalho contribuiu para a formalização das etapas que envolvem a concepção de uma linguagem. Além disso, sua estruturação permite flexibilidade, sugerindo caminhos possíveis para o desenvolvimento. É válido destacar também que o trabalho colaborou com o aprendizado sobre linguagens de programação, modelos de processo de software e requisitos.

Como trabalhos futuros, pode-se elencar a aplicação do modelo de processo em outros contextos e linguagens, além do emprego de melhorias nos artefatos, recursos sugeridos e fluxos demonstrados no NILO. Tais melhorias observadas envolvem o incremento de técnicas e passos comuns no levantamento de requisitos, como: (1) mapeamento dos interessados no projeto de linguagem antes da definição concreta do escopo; e (2) identificação do conhecimento dos envolvidos na concepção da linguagem acerca do seu domínio. Outros incrementos que perpassam os requisitos também foram observados, como: (1) demonstração, na fase de teste, da utilização de alguma das ferramentas de testes, que independe da forma escolhida para implementação de código; (2) adição de mais artefatos e perguntas na fase de definição que possam ajudar na delimitação mais detalhada do domínio e das necessidades; e (3) incremento de artefatos na fase de codificação, de forma a prestar um suporte mais concreto ao desenvolvimento de tradutores de linguagem. Assim, o modelo de processo NILO poderá apresentar melhorias consideráveis na sua constituição, contribuindo mais fortemente no projeto e na implementação de linguagens de programação.

REFERÊNCIAS

MERNIK, Marjan; HEERING, Jan; SLOANE, Anthony M.. When and how to develop domain-specific languages. **Acm Computing Surveys**, [S.L.], v. 37, n. 4, p. 316-344, dez. 2005. Association for Computing Machinery (ACM). http://dx.doi.org/10.1145/1118890.1118892.

TRIVEDI, Munesh Chandra. Role Of Context In Usability Evaluations: a review. **Advanced Computing**: An International Journal, [S.L.], v. 3, n. 2, p. 69-78, 31 mar. 2012. Academy and Industry Research Collaboration Center (AIRCC). http://dx.doi.org/10.5121/acij.2012.3208.

AHO, Alfred V.; LAM, Monica S.; SETHI, Ravi; ULLMAN, Jeffrey D. **Compiladores: princípios, técnicas e ferramentas**. 2. ed. São Paulo: Pearson Education do Brasil, 2008.

BOEHM, B. W.. A spiral model of software development and enhancement. **Computer**, [S.L.], v. 21, n. 5, p. 61-72, maio 1988. Institute of Electrical and Electronics Engineers (IEEE). http://dx.doi.org/10.1109/2.59.

NYSTROM, Robert. **Crafting Interpreters**. [S. L.]: Genever Benning, 2021. 639 p.

VOELTER, Markus et al. **DSL Engineering**: designing, implementing and using domain-specific languages. North Charleston: Createspace Independent Publishing Platform, 2013. 558 p.

FAVRE, J.-M.. Languages evolve too! Changing the Software Time Scale. **Eighth International Workshop On Principles Of Software Evolution (Iwpse'05)**, [S.L.], p. 33-44, 2005. IEEE. http://dx.doi.org/10.1109/iwpse.2005.22.

SOMMERVILLE, Ian. **Engenharia de software**. 10. ed. São Paulo: Pearson Education do Brasil, 2019. 744 p.

PRESSMAN, Roger S. **Engenharia de Software**: uma abordagem profissional. 9. ed. Porto Alegre: Amgh, 2021.

PEFFERS, Ken; TUUNANEN, Tuure; ROTHENBERGER, Marcus A.; CHATTERJEE, Samir. A Design Science Research Methodology for Information Systems Research. **Journal Of Management Information Systems**, [S.L.], v. 24, n. 3, p. 45-77, dez. 2007. Informa UK Limited. http://dx.doi.org/10.2753/mis0742-1222240302.

SPINELLIS, Diomidis. Notable design patterns for domain-specific languages. **Journal Of Systems And Software**, [S.L.], v. 56, n. 1, p. 91-99, fev. 2001. Elsevier BV. http://dx.doi.org/10.1016/s0164-1212(00)00089-3.

KITCHENHAM, B.; PICKARD, L.; PFLEEGER, S.L. Case studies for method and tool evaluation. **IEEE Software**, v. 12, n. 4, p. 52–62, 1995. Disponível em: http://ieeexplore.ieee.org/document/391832. Acesso em: 2 jul. 2026.

CUEVAS, Erik Hombre; ZALDIVAR, Daniel; PEREZ, Marco. Impact of Programming Languages on Learning Performance. **International Journal Of Information And Communication Technology Education**, [S.L.], v. 21, n. 1, p. 1-17, 22 mar. 2025. IGI Global. http://dx.doi.org/10.4018/ijicte.371419.

VASCONCELOS, Felipe Alves. **Linguagem-Matty**. 2026. Disponível em: https://github.com/FelipeAlves14/linguagem-Matty. Acesso em: 14 jun. 2026.

Hogan, M. and Newton, E. (2015), Supplemental Information for the Interagency Report on Strategic U.S. Government Engagement in International Standardization to Achieve U.S. Objectives for Cybersecurity, **NIST Interagency/Internal Report** (NISTIR), National Institute of Standards and Technology, Gaithersburg, MD, https://doi.org/10.6028/NIST.IR.8074v2. Disponível em: https://nvlpubs.nist.gov/nistpubs/ir/2015/NIST.IR.8074v2.pdf. Acesso em: 16 maio 2026.

GLINZ, Martin; VAN LOENHOUD, Hans; STAAL, Stefan; BÜHNE, Stan. **Nível Fundamental**: Guia de Estudo para o Certified Professional for Requirements Engineering (CPRE) de acordo com o padrão IREB. Versão 1.2.2. [S.l.]: International Requirements Engineering Board (IREB), 2024.

PARR, Terence. **The Definitive ANTLR 4 Referenc**e. S. L: Pragmatic Bookshelf, 2013. 328 p.

LLVM Project (org.). **The LLVM Compiler Infrastructure**. 2026. Disponível em: https://llvm.org/. Acesso em: 4 abr. 2025.