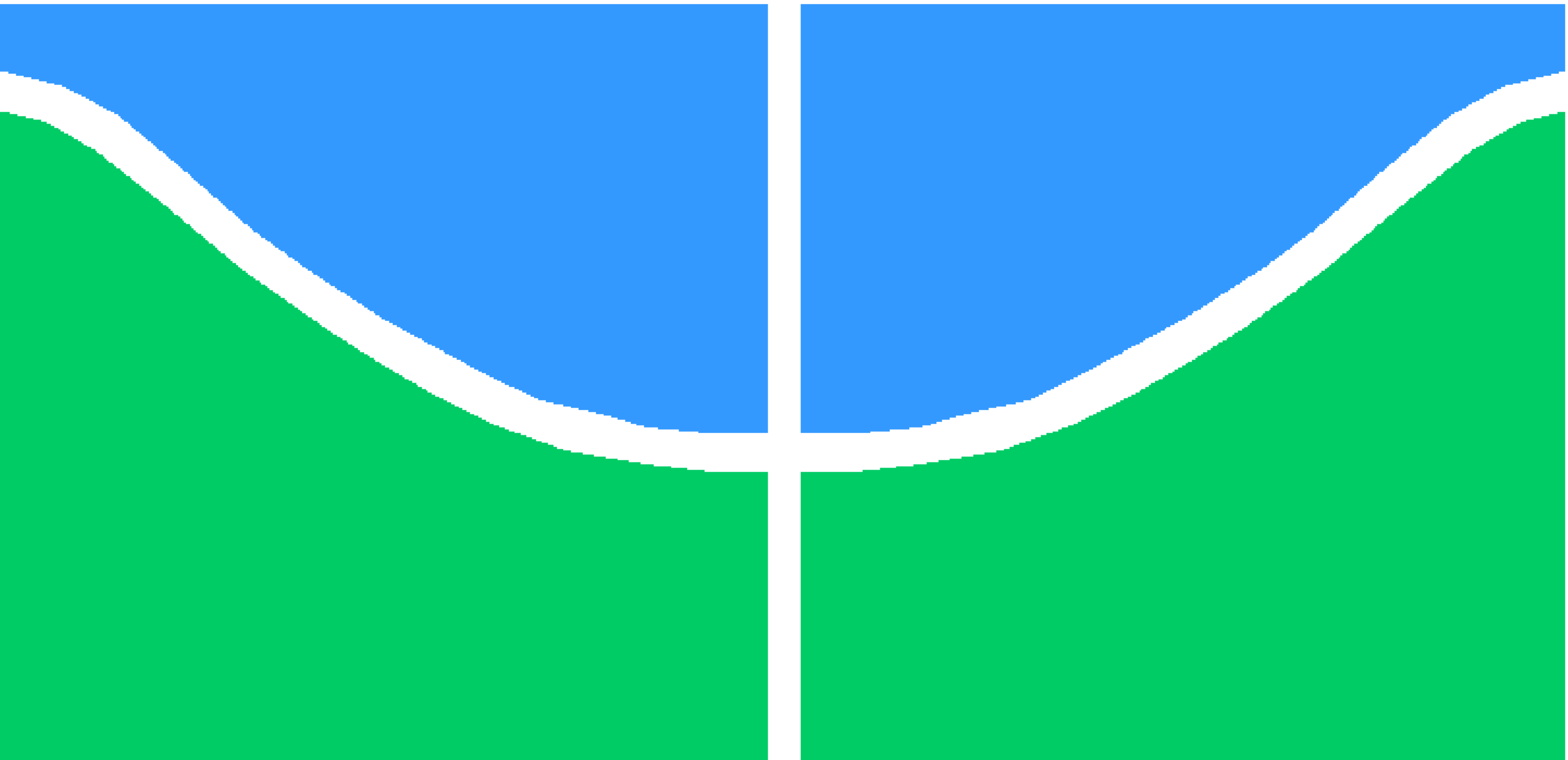

Universidade de Brasília – UnB
Faculdade UnB Gama – FGA
Engenharia de Software

Plai: Linguagem de programação para manipulação de dados

Autor: Matheus Batista Silva
Orientador: Professor Doutor Fábio Macedo Mendes

Brasília, DF
2022

Matheus Batista Silva

Plai: Linguagem de programação para manipulação de dados

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Universidade de Brasília – UnB

Faculdade UnB Gama – FGA

Orientador: Professor Doutor Fábio Macedo Mendes

Brasília, DF

2022

Matheus Batista Silva
Plai: Linguagem de programação para manipulação de dados/ Matheus Batista Silva. – Brasília, DF, 2022-
52 p. : il. (algumas color.) ; 30 cm.

Orientador: Professor Doutor Fábio Macedo Mendes

Trabalho de Conclusão de Curso – Universidade de Brasília – UnB
Faculdade UnB Gama – FGA , 2022.

1. Pipelines de dados. 2. DSL. I. Professor Doutor Fábio Macedo Mendes. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Plai: Linguagem de programação para manipulação de dados

CDU 02:141:005.6

Matheus Batista Silva

Plai: Linguagem de programação para manipulação de dados

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Trabalho aprovado. Brasília, DF, 09 de março de 2022 – Data da aprovação do trabalho:

Professor Doutor Fábio Macedo Mendes
Orientador

Professora Doutora Carla Silva Rocha Aguiar
Convidado 1

Convidado 2

Brasília, DF
2022

Resumo

Os desafios de desenvolvimento e manutenção de aplicações de coleta e manipulação de dados torna esses sistemas fontes de dívidas técnicas, parte devido ao uso de linguagens de programação de propósito geral (GPLs) em seu desenvolvimento. Pela natureza generalista das GPLs, esse tipo de linguagem possui dificuldade em expressar soluções para problemas de domínios complexos. Em comparação a linguagens de domínio específico (DSLs), as GPLs são mais difíceis de entender e demandam mais tempo de análise. Esse trabalho apresenta uma DSL para o desenvolvimento de *pipelines* de manipulação de dados, sendo uma linguagem interpretada são necessários os seguintes componentes básicos de um interpretador: gramática, analisador sintático, representação intermediária e ambiente de interpretação. Além disso, a linguagem apresentada provê ferramentas para realizar a validação estrutural dos dados manipulados em tempo de execução.

Palavras-chave: manipulação de dados. pipelines de dados. DSL. interpretador.

Abstract

The challenges of developing and maintaining data collection and manipulation applications make these systems a source of technical debt, in part due to the usage of general purpose programming languages (GPLs) in its development. Because of the generalist nature of GPLs, this kind of programming language faces difficulties in expressing solutions to complex domain problems. In comparison with domain specific programming languages (DSLs), the GPLs are harder to understand and demand more time of analysis. This work presents a DSL for development of data manipulation pipelines, being an interpreted language it is necessary the following basic components of an interpreter: grammar, parser, intermediate representation and interpretation environment. Furthermore, the presented language provides tools for executing data schema validation in runtime.

Key-words: data manipulation. data pipelines. DSL. interpreter.

Lista de ilustrações

Figura 1 – Visão geral das fases do compilador 23
Figura 2 – *Top-Down* parser da expressão 1100 26
Figura 3 – *Bottom-Up* parser da expressão 1100 26
Figura 4 – Árvore sintática abstrata da expressão $y = 42$ 27
Figura 5 – Ciclo *Red-Green-Refactor* 29
Figura 6 – Processo de integração contínua 33
Figura 7 – Fluxo de análise léxica e sintática 36
Figura 8 – Relatório de cobertura de testes 44

Lista de tabelas

Tabela 1 – Número de linhas de código por módulo. 44

Lista de abreviaturas e siglas

API	*Application programming interface*
BNF	*Backus-Naur Form*
CSV	*Comma-separated values*
DSL	*Domain-specific language*
EBNF	*Extended Backus-Naur Form*
ETL	*Extract, transform and load*
GPL	*General-purpose language*
JSON	*JavaScript Object Notation*
LALR	*Look ahead left to right, rightmost derivation*
PyPI	*Python Package-Index*
SQL	*Structured Query Language*
TDD	*Test-driven development*

Sumário

1 INTRODUÇÃO . . . 17
1.1 Objetivo . . . 19

2 FUNDAMENTAÇÃO TEÓRICA . . . 21
2.1 Mineração de dados . . . 21
2.1.1 Obtenção dos dados . . . 21
2.1.2 Manipulação dos dados . . . 21
2.1.3 Exploração dos dados . . . 22
2.2 Teoria de compiladores . . . 22
2.2.1 Análise léxica . . . 23
2.2.2 Análise sintática . . . 23
2.2.3 Representação intermediária . . . 26
2.2.4 Interpretador . . . 27

3 METODOLOGIA . . . 29
3.1 Desenvolvimento . . . 29
3.2 Ambiente de execução . . . 30
3.3 Versionamento . . . 30
3.4 Análise léxica e sintática . . . 31
3.5 Manipulação de dados . . . 32
3.6 Testes . . . 32

4 RESULTADOS . . . 34
4.1 Exemplo de uso . . . 34
4.2 Arquitetura . . . 35
4.3 Análise léxica e sintática . . . 35
4.3.1 Gramática . . . 36
4.3.2 Representação intermediária . . . 39
4.4 Interpretação . . . 41
4.5 Validações . . . 43
4.6 Métricas do projeto . . . 43
4.6.1 Cobertura de testes . . . 44

5 CONCLUSÃO . . . 45

REFERÊNCIAS . . . 46

APÊNDICES 49

APÊNDICE A – GRAMÁTICA EBNF DA LINGUAGEM PLAI . . . 50

1 Introdução

O crescimento expressivo da coleta de dados nos últimos anos, impulsado pelo fenômeno das redes sociais e internet das coisas, levou diversas organizações a enxergarem as oportunidades de análise inerentes a esse volume de informações. A exploração desses conjuntos de dados permite identificar possibilidades de otimização de processos internos, redução de custos e até mesmo aprimorar o engajamento dos clientes nos produtos oferecidos (LEE, 2017).

Para utilizar essas análises como insumo no processo de tomada de decisão, é necessário garantir que as fontes utilizadas possuam um nível de qualidade aceitável. Visto que as organizações podem encontrar-se em um estado inicial de maturidade em relação ao gerenciamento de dados, é comum a falta de uma estratégia de administração da informação, resultando em bases incompletas ou com dados desestruturados.

As principais dificuldades ao trabalhar com dados nesse cenário encontra-se principalmente na agregação e interoperabilidade, exigindo profissionais qualificados para lidar com essa demanda, além de questões referentes a inconsistência dos dados, integração com sistemas legados e monitoramento (KADADI et al., 2014).

É comum aplicar o processo conhecido como ETL (*Extract, Transform and Load*) para gerar fontes de dados homogêneas. Esse é um tipo de *pipeline* de dados, ou seja, um conjunto de operações sequencias, onde o resultado de uma operação serve como insumo para a próxima (BEYER et al., 2016).

A primeira etapa do ETL é a extração, onde os dados são coletados das diferentes fontes. Na segunda etapa, chamada transformação, são executadas operações de agregação e limpeza, aplicando também um conjunto de regras definidas a partir do contexto negocial. Finalmente na última etapa, conhecida como carregamento, as informações resultantes das transformações são enviadas para uma base de dados onde as aplicações clientes poderão consumir o resultado (VASSILIADIS; SIMITSIS; SKIADOPOULOS, 2002).

As fontes de dados resultantes desse processo servem também como insumo para o treinamento de algoritmos de aprendizado de máquina. Esses sistemas diferem dos sistemas tradicionais, principalmente pela capacidade de aprimorar as predições e categorizações com base nos dados providos, sem a necessidade de reprogramação manual (DAVENPORT; RONANKI, 2018).

A etapa de preparação das informações que alimentarão o algoritmo é essencial para garantir que o modelo terá o desempenho esperado, por exemplo, em um problema de classificação espera-se que o modelo receba uma quantidade balanceada de informações

de cada uma das classes que devem ser identificadas. Além disso, alguns algoritmos exigem estruturas de dados específicas e não podem ser treinados caso as informações de entrada não satisfaçam as condições estabelecidas, por exemplo, de não possuir valores nulos ou que os dados de entrada estejam normalizados (AKKIRAJU et al., 2018).

No ciclo de vida de aplicações que dependem de fontes de dados heterogêneas, é comum que as *pipelines* de processamento de dados se tornem origem de dívidas técnicas. Essas dívidas acabam surgindo quando novas fontes de dados são incorporadas a *pipeline* ou novas parametrizações são sugeridas tornando-se necessário alterar o código inicial para suportar essas mudanças. Isto eleva o custo de implementação de novas funcionalidades e testes de integração de ponta a ponta (SCULLEY et al., 2014).

As *pipelines* de processamento de dados normalmente são construídas utilizando linguagens de programação de propósito geral (GPLs), que têm como principais vantagens o ecossistema de ferramentas e as fontes de conhecimento disponíveis para os usuários da linguagem. Devido à necessidade de atender aos problemas dos mais variados domínios, as GPLs têm se tornado mais complexas, o que prejudica o processo de desenvolvimento (KOSAR et al., 2012).

Linguagens de domínio específico (DSLs) são uma das alternativas para abordar problemas de domínios complexos, oferecendo sintaxe e funcionalidades que melhor correspondem ao âmbito da solução. Aplicações desenvolvidas utilizando DSLs geralmente possuem menos falhas e são mais facilmente compreendidas em comparação a aplicações que utilizam GPLs(KOSAR et al., 2012).

Para o desenvolvimento de aplicações de manipulação de dados, as GPLs sofrem com a falta de estruturas de dados e sintaxe generalista, que tornam ineficiente o desenvolvimento e manutenção dessas aplicações.

Esse trabalho propõe uma DSL para o desenvolvimento eficiente de *pipelines* de manipulação de dados, que permite montar de forma declarativa as operações necessárias para agregação e limpeza de um conjunto de dados. O trecho 1.1 apresenta um exemplo de *pipeline* escrita em Plai, que realiza agregação, criação de novas colunas, filtragem e exportação do resultado em formato de arquivo.

```
pipeline(dataframe) as 'pct_issues_by_name.csv':
    $.groupby(.name, as_index=False).sum()
    (.count/.count.sum()) * 100 as pct
    {.name, .pct}
```

Código fonte 1.1 – Exemplo de código Plai

1.1 Objetivo

Este trabalho propõe uma linguagem de programação de domínio específico (DSL) para manipulação de dados em lote. A linguagem apresenta uma estrutura para definição de *pipelines* de dados, além de validação de esquema dos dados de entrada e saída.

A proposta é desenvolver uma linguagem interpretada que segue um fluxo tradicional de desenvolvimento de interpretadores, isso inclui as seguintes etapas:

1. desenvolver a sintaxe da linguagem e a correspondente gramática livre de contexto que suporte declaração de *pipelines* que realizam manipulação e tipagem de dados de entrada e saída, além de outras operações básicas;
2. reduzir a árvore sintática para uma estrutura intermediária;
3. desenvolver um interpretador para executar a estrutura de dados intermediária;
4. implementar etapa de validação estrutural das informações de entrada e saída das *pipelines* em tempo de execução.

2 Fundamentação teórica

2.1 Mineração de dados

Sistemas de mineração de dados são utilizados para a obtenção de conhecimento sobre produtos, aplicações ou processos. Um desafio comum no processamento desses dados coletados é a existência de problemas de formatação, valores ausentes, duplicações, contradições, valores discrepantes e outros tipos de inconsistências. Todos esses problemas afetam a qualidade das análises desenvolvidas posteriormente e o pré-processamento desses dados se torna uma parte considerável do esforço de desenvolvimento de um sistema de mineração de dados.

O processo de mineração de dados é composto de diferentes etapas que buscam assegurar a qualidade das informações analisadas. Assim, a *pipeline* de mineração de dados é a integração de todas as etapas desse processo, sendo que as principais fases incluem obtenção, manipulação, exploração, modelagem e interpretação (LI, 2019). Este trabalho tem como foco as três primeiras etapas desse processo.

2.1.1 Obtenção dos dados

A obtenção representa a etapa de extração dos dados das diferentes fontes disponíveis. Essa etapa é fundamental, pois aqui são definidas quais informações serão utilizadas. Além das fontes tradicionais de dados estruturados como banco de dados, é comum serem utilizadas informações pouco estruturados ou desestruturados provenientes de diferentes categorias de sensores. As informações coletadas podem ser armazenadas como arquivos, ou reestruturadas e inseridas em um banco de dados.

2.1.2 Manipulação dos dados

A fase de manipulação consiste tanto na limpeza dos dados para mitigar os problemas de formatação, quanto na transformação visando modificar a representação das informações para criar uma base padronizada. (LI, 2019).

Nesta etapa é comum lidar com valores ausentes, detectar discrepâncias, textos mal formatados e com codificação diferente do padrão esperado, entre outros problemas. A fase de transformação tipicamente lida com a normalização de valores, conversão entre valores categóricos e numéricos, discretização, entre outras transformações (LI, 2019).

2.1.3 Exploração dos dados

A etapa de exploração busca identificar padrões presentes nos dados, bem como as correlações entre as variáveis que constituem o conjunto. Nessa etapa são levantadas hipóteses que orientarão o desenvolvimento das análises para melhor interpretar as informações disponíveis. Para construir o estudo de exploração são utilizadas análises estatísticas e ferramentas de visualização de dados (LI, 2019).

2.2 Teoria de compiladores

Para que o computador consiga efetuar as instruções implementadas em uma linguagem de programação, é necessário traduzi-las para uma representação que a máquina consiga executar. Para isso, foram elaborados sistemas de software chamados compiladores, preparados para traduzir linguagens de programação de alto nível para uma representação equivalente em linguagem de máquina ou em estruturas de dados que podem ser posteriormente interpretadas (AHO et al., 2006).

A arquitetura geral de um compilador consiste essencialmente de duas partes: análise e síntese. A etapa análise consiste em identificar a estrutura sintática e semântica do código-fonte, resultando em uma representação intermediária, caso essa estrutura seja validada. A síntese consiste em otimizar a solução e gerar o código de máquina a partir da representação intermediária criada pela etapa de análise (AHO et al., 2006).

Outro tipo conhecido de sistema de processamento de linguagem são os interpretadores. A principal diferença é que os interpretadores não geram uma tradução de todo o código-fonte para linguagem de máquina, o interpretador executa as instruções implementadas com base na representação intermediária gerada, podendo ser a árvore sintática, código-fonte ou *bytecode* que executa em uma máquina virtual (PARR, 2010). Por mais que exista uma separação entre compilador e interpretador, ambas as abordagens são estudadas pela teoria de compiladores e compartilham técnicas em comum, assim dependendo da linguagem pode existir uma sobreposição dos métodos.

A Figura 1 apresenta os componentes que atuam tanto nos compiladores, quanto interpretadores. A etapa de análise faz parte de ambas as abordagens, a diferença ocorre na segunda etapa. Enquanto o compilador realiza a síntese para produzir um executável, o interpretador realiza a execução da representação intermediária.

Este trabalho tem como foco uma linguagem interpretada, logo nas seções seguintes serão discutidos os componentes básicos necessários para construir um interpretador.

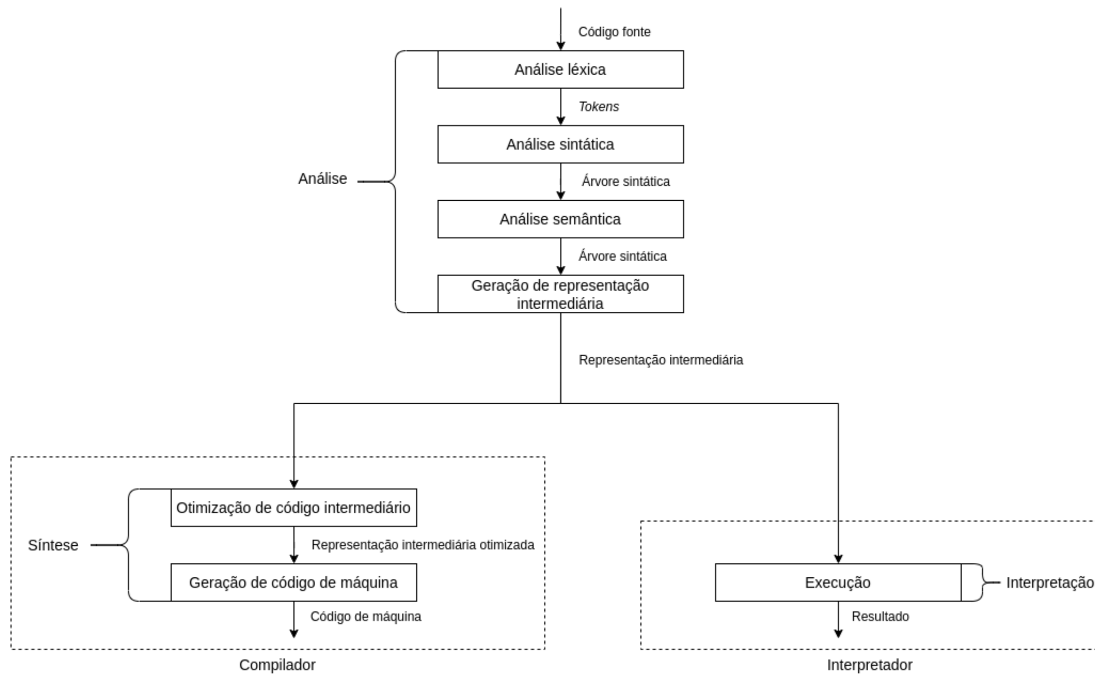

Figura 1 – Visão geral das fases do compilador

2.2.1 Análise léxica

A análise léxica é responsável por transformar o código-fonte, normalmente recebido como uma sequência de caracteres, para uma sequência de *tokens*. O *token* é uma representação que consiste em um par de valores contendo um símbolo que caracteriza aquela unidade léxica, e o valor equivalente encontrado no código de entrada (AHO et al., 2006).

Considerando que diferentes palavras podem ser associadas a uma mesma categoria de *token*, é necessário identificar o padrão que descreve adequadamente todos os seus lexemas. Esses padrões são tipicamente descritos por expressões regulares. Esta é uma representação conveniente já que lida com operações de concatenação, repetição e união a partir de padrões e frases simples (AHO et al., 2006).

Padrões identificados por expressões regulares tipicamente são expressos de maneiras mais sucintas do que outras abordagens. Por exemplo, pode-se descrever uma operação de concatenação pela expressão $[a-z][0-9]$, que descreve uma letra do alfabeto seguido de um dígito de 0 a 9. Valores que contém uma letra ou um dígito podem ser exemplificados pela expressão $[a-z] \mid [0-9]$. Já palavras quem contém múltiplas letras podem ser representadas pela expressão regular $[a-z]*$.

2.2.2 Análise sintática

Ao criar uma determinada linguagem nem toda sequência de *tokens* pode ser considerada uma expressão gramaticalmente correta. Para isso, é necessário definir uma estru-

tura capaz de descrever quais as formas gramaticais podem ser consideradas bem formadas para a linguagem implementada. Na teoria formal da linguagem essa estrutura é conhecida como **gramática generativa**, e consiste em um conjunto de descrições especificadas por regras de produção recursivas que definem que tipos de substituições podem ser feitas para produzir frases conforme a gramática. (HAUSSER, 1999).

Considere, por exemplo, a linguagem muito simples que produz frases com k repetições do número 1, seguidas de k repetições do número 0, por exemplo: 10, 1100, 111000, 11110000, etc. A Equação 2.1 apresenta como essa linguagem pode ser descrita utilizando o formato *phrase-structure*[1]

$$\begin{aligned} S &\rightarrow 1S0 \\ S &\rightarrow 10. \end{aligned} \tag{2.1}$$

Essa notação consiste em substituir o símbolo à esquerda da seta ($\rightarrow$) pelos valores equivalentes à direita. Nesse caso, para derivar a expressão 111000, a primeira regra é aplicada ($S \rightarrow 1S0$). Em seguida aplicando a primeira regra novamente substituindo o S da expressão pelos valores à direita da primeira regra, ficamos com $11S00$. Finalmente, aplicando a segunda regra e realizando a substituição do S temos como resultado 111000.

Com a natureza recursiva da primeira regra é possível derivar um conjunto infinito de expressões válidas para a linguagem utilizando um conjunto finito de regras e *tokens*. As gramáticas então são estruturas capazes de definir as regras sintáticas de uma linguagem de uma maneira precisa (HAUSSER, 1999).

No contexto da construção de uma linguagem de programação é comum utilizar as **gramáticas livres de contexto**, que consistem em uma sequência de regras de produção. Cada regra de produção especifica como símbolos terminais e não-terminais podem ser combinados para formar frases usando as propriedades recursivas da gramática generativa, um exemplo é a Equação 2.1. Cada produção é composta pelo elemento não-terminal definido pela produção, o símbolo identificador ($\rightarrow, :, =, ::=, etc$) e o corpo que descreve como as frases são formadas a partir de símbolos terminais e não-terminais (HOPCROFT; MOTWANI; ULLMAN, 2007).

Os terminais são elementos primários da linguagem que formam as frases, esses correspondem aos valores carregados pelos *tokens* extraídos na etapa de análise léxica, então é comum que os elementos terminais também sejam chamados de *tokens* (HOPCROFT; MOTWANI; ULLMAN, 2007).

Os não-terminais são variáveis sintáticas compostas por valores que descrevem um determinado padrão para um conjunto de palavras. Cada símbolo não-terminal representa

[1] Gramática de estrutura frasal

um subconjunto da linguagem implementada. Um dos símbolos não-terminais possui um caráter especial, pois determina qual a regra que dará início ao processo de derivação, esse símbolo é chamado de *start* (HOPCROFT; MOTWANI; ULLMAN, 2007).

A definição de uma gramática tipicamente utiliza uma metalinguagem para descrever que regras a compõe. A ISO/IEC 14977:1996[2] apresenta uma definição formal da metalinguagem sintática EBNF, que estende a notação BNF (*Backus-Naur Form*) primeiramente utilizada para a definição da linguagem de programação ALGO60 (BACKUS et al., 1963).

A notação EBNF foi produzida por Niklaus Wirth, e apresenta algumas mudanças simples de sintaxe e operadores utilizados para representar repetições e valores opcionais, permitindo criar gramáticas claras evitando o uso de recursão nos casos mais simples. Propõem também uma distinção clara entre terminais e não-terminais, permitindo o uso de metas símbolos como parte dos símbolos da linguagem. A notação EBNF evita o uso explícito de um símbolo para representar frases vazias (ϵ), já que possui operadores específicos para lidar com repetições e valores opcionais, que normalmente requisitariam tal símbolo (WIRTH, 1977).

As extensões meta sintáticas apresentadas pela EBNF incluem o uso dos metas símbolos:

- Chaves ($\{\}$) para expressar repetição de zero ou mais elementos, onde $\{a\}$ significa $\epsilon|a|aa|aaa|aaaa...$;
- Colchetes ($[]$) são utilizados para indicar símbolos opcionais como, por exemplo, $[a]$ expressa $\epsilon|a$;
- Parênteses ($()$) indicam agrupamento, assim $(a|b)c$ representa $ac|bc$;
- Operadores de expressões regulares ($*$?) como sintaxe alternativa para as operações descritas nos itens anteriores;
- Os valores de literais são expressos por aspas, por exemplo: $a = "a"$.

Os *tokens* encontrados na etapa de análise léxica são organizados conforme as regras da gramática, geralmente em uma forma de árvore, conhecida como árvore sintática. Existem duas abordagens típicas para a construção da árvore, ambas utilizam operações de derivação para construir a estrutura hierárquica de forma que reflita o conteúdo do código-fonte (AHO et al., 2006).

A primeira abordagem conhecida como *Top-Down* constrói a árvore a partir da produção que define o símbolo não-terminal de *start* da gramática, aplicando operações

[2] https://www.iso.org/standard/26153.html

de derivação até as folhas da árvore que correspondem aos símbolos terminais. Tomando como exemplo a expressão 1100 e gramática definida na Equação 2.1, pode-se observar as etapas de construção da árvore sintática usando a abordagem *Top-Down* na Figura 2.

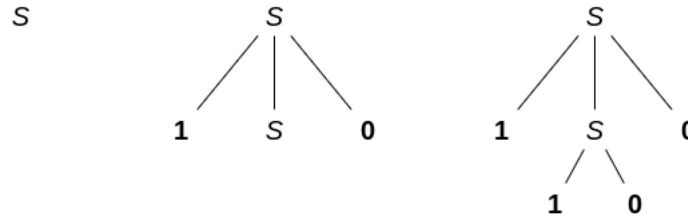

Figura 2 – *Top-Down* parser da expressão 1100

Já a abordagem *Bottom-Up*, realiza o caminho inverso, construindo a árvore das folhas até a raiz (regra de *start*) aplicando regras de redução (AHO et al., 2006). Pode-se observar as etapas desse processo no exemplo anterior na Figura 3.

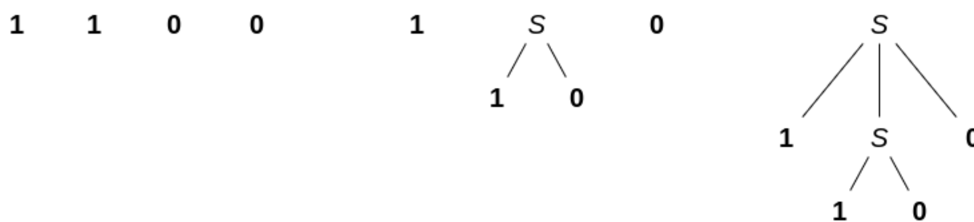

Figura 3 – *Bottom-Up* parser da expressão 1100

2.2.3 Representação intermediária

O compilador/interpretador normalmente constrói uma representação que resume as instruções presentes no código-fonte para facilitar a etapa de geração do programa final ou interpretação (AHO et al., 2006). A árvore sintática gerada ao fim da etapa de análise sintática contém a descrição da sequência de regras que o *parser* aplica durante a avaliação da entrada textual, apesar de ser útil para avaliar a execução do *parser*, essa estrutura pode conter informações desnecessárias para a etapa de execução.

É possível realizar a execução do programa diretamente a partir da árvore sintática, mas isso gera um acoplamento entre a estrutura de avaliação e a gramática, que faz com que mudanças de sintaxe possam impactar diretamente em como o programa é avaliado.

Uma forma comum de representação intermediária é a árvore sintática abstrata, essa estrutura tem como enfase três princípios de design: não deve conter nós desnecessários; deve ser uma estrutura simples de avaliar; tem como foco os operadores, operandos e a relação entre eles (PARR, 2010). Pode ser observado um exemplo desse tipo de árvore para a expressão $y = 42$ na Figura 4.

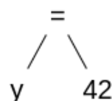

Figura 4 – Árvore sintática abstrata da expressão $y = 42$

A representação intermediária depende diretamente de como será seu uso na etapa de avaliação e das características da linguagem em que foi implementada. É possível criar essa representação utilizando classes, estruturas de dados genéricas e em muitas situações pode-se utilizar a árvore sintática abstrata diretamente como representação intermediária.

2.2.4 Interpretador

Interpretadores são responsáveis por realizar a execução das instruções codificadas na representação intermediária mediante um sistema de software que simula um computador. Essencialmente, esse sistema possui um componente responsável por extrair as instruções de código da memória, decodifica-las e executa-las. Além disso, possui componentes responsáveis por realizar o gerenciamento de memória para permitir operações de leitura, escrita e execução.

Os interpretadores podem ser divididos em alto e baixo nível. A diferença entre essas abordagens é qual tipo de entrada o sistema utiliza para avaliar o programa implementado. Em interpretadores de alto nível são utilizadas estruturas mais simples, como o próprio código-fonte ou a árvore sintática gerada pela etapa de análise sintática. Em interpretadores de baixo nível são utilizadas representações intermediárias do código-fonte, como *bytecode* (PARR, 2010).

Normalmente, a escolha entre um interpretador de alto ou baixo nível é balanceada pelos fatores de facilidade de implementação e desempenho. Enquanto interpretadores de alto nível demandam menos esforço para sua implementação, tipicamente possuem uma eficiência reduzida em comparação a interpretadores de baixo nível. Este trabalho tem como foco a facilidade de implementação, por esse motivo escolheu-se trabalhar com um interpretador de alto nível.

3 Metodologia

Esse capítulo descreve as ferramentas, técnicas, algoritmos e processos utilizados para o desenvolvimento deste trabalho. Bem como o ambiente de execução que utiliza como base a linguagem de programação Python.

3.1 Desenvolvimento

O desenvolvimento utiliza a metodologia *Test-driven development* (TDD), que consiste em um ciclo de desenvolvimento iniciado pela criação dos testes correspondentes a funcionalidade que se deseja implementar. O TDD é orientado por duas regras:

1. A adição de um novo trecho de código deve ter como principal motivação a correção de um teste automatizado falho;
2. Duplicações devem ser eliminadas.

O processo de desenvolvimento proposto pelo TDD é composto pelas etapas: *Red*, onde um teste simples é implementado, que a princípio irá falhar; em seguida na etapa *Green*, o código que faz o teste passar é implementado; enfim, na etapa de *Refactor*, toda duplicação criada na fase de implementação da funcionalidade deve ser removida ou refatorada. Esse processo é cíclico e acontece durante todo o desenvolvimento do sistema, como pode ser observado na Figura 5 (BECK, 2003).

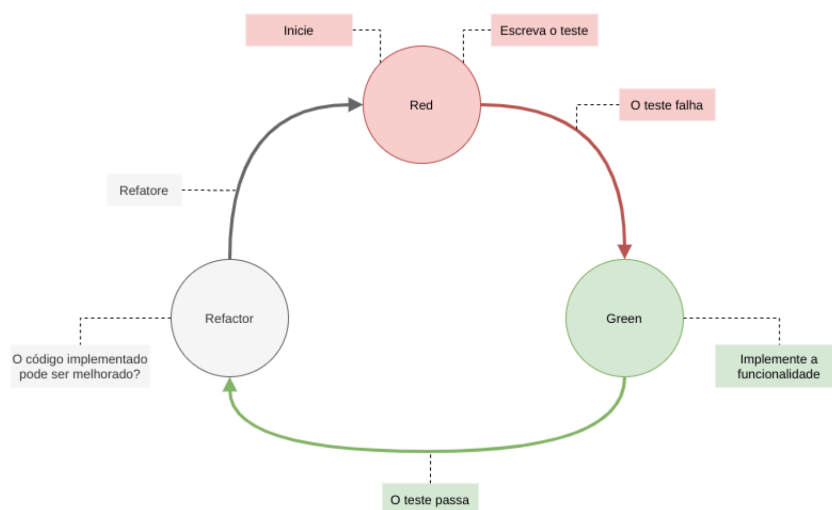

Figura 5 – Ciclo *Red-Green-Refactor*

O desenvolvimento do projeto ocorreu de forma cíclica para cada nova funcionalidade desenvolvida, sendo que ao final de cada ciclo uma versão funcional da linguagem foi

gerada. Para isso, cada funcionalidade foi decomposta em três etapas principais: escrita da gramática, redução da árvore sintática e interpretação. Para cada uma dessas etapas foi aplicado a metodologia TDD durante o processo de desenvolvimento.

A motivação para a escolha da metodologia TDD para o desenvolvimento desse trabalho foi devido à capacidade da mesma de auxiliar na cobertura de casos excepcionais que geram comportamentos inesperados. Ao desenvolver um compilador ou interpretador, existem casos atípicos que causam erros de execução que podem ser ignorados durante a implementação, além de que é importante garantir que variações de sintaxe sejam validadas corretamente.

3.2 Ambiente de execução

O projeto foi desenvolvido e testado no ambiente Python versão 3.6 à 3.9. Todas as dependências necessárias para executar o ambiente de desenvolvimento estão listadas no projeto e podem ser instaladas utilizando o *Python Package Index* (PyPI), um repositório de *software* para a linguagem de programação Python.

A versão estável do projeto está disponível também no PyPI e pode ser instalada como um pacote Python chamado plai[1].

3.3 Versionamento

O projeto segue o padrão de versionamento semântico, definido pelo formato: "Maior.Menor.Correção". Onde alterações na API incompatíveis com a versão anterior do projeto incrementam a "Maior", adições ou alterações compatíveis com a versão anterior incrementam a "Menor"e correções de falhas que não afetam a API acrescentam a "Correção"(PRESTON-WERNER, 2013).

O versionamento do código-fonte utiliza as ferramentas Git e Github. O Git[2] é um sistema de controle de versão distribuído, responsável por manter o rastro das diferentes mudanças realizadas no código-fonte de projetos durante o processo de desenvolvimento. Já o Github é um serviço de hospedagem na nuvem de projetos versionados pelo Git, permitindo que diferentes desenvolvedores colaborem remotamente no processo de desenvolvimento.

[1] https://pypi.org/project/plai
[2] https://git-scm.com/

3.4 Análise léxica e sintática

A etapa de divisão da entrada textual em *tokens* e a criação da árvore sintática foi implementada utilizando a biblioteca Lark[3]. O Lark suporta a definição de gramáticas livres de contexto através de sintaxe baseada na notação EBNF e implementa os seguintes algoritmos de análise sintática: Earley, LALR(1) e CYK. Diferente de outras ferramentas como Bison e Lex, o Lark realiza a construção automática da árvore sintática com base na declaração da gramática e é possível escolher o algoritmo de análise sintática no momento de inicialização do código.

O algoritmo escolhido para a etapa de análise sintática do projeto foi o LALR(1), que significa: *Look ahead left to right, rightmost derivation*. Esse analisador sintático utiliza a técnica conhecida como *Lookahead*, que consiste em observar um *token* a frente do que está sendo analisado, no caso do LALR(1), para decidir qual regra da gramática deve ser aplicada naquele trecho. Esse algoritmo é eficiente em uso de memória e tempo de execução, além de suportar a gramática de linguagens de propósito geral como Python e Java (SHINAN, 2021b).

A biblioteca Lark também oferece mecanismos para realizar o processamento da árvore sintática, facilitando a aplicação de transformações ou implementação de lógicas específicas de redução da árvore gerada. Um desses mecanismos é o *Transformer*, que facilita visitar cada nó da árvore sintática e aplicar a transformação apropriada conforme o tipo do nó. Os *Transfomers* do Lark executam de maneira bottom-up, ou seja, das folhas para a raiz da árvore (SHINAN, 2021c).

O processo de redução da árvore sintática procura criar uma estrutura concisa para a interpretação. No contexto desse projeto, a representação intermediária é uma *S-expression*, uma solução simples que oferece estrutura suficiente para execução dos programas declarados. *S-expressions*, ou *Symbolic Expressions*, são expressões prefixas agrupadas por parênteses utilizadas como notação para a declaração de programas e estrutura de dados da linguagem Lisp. Por exemplo, a expressão $2+4*3$ é representada na notação *S-expression* por $(+\ 2\ (*\ 4\ 3))$(LEITÃO, 2008).

Para esse trabalho foi utilizada uma adaptação da notação *S-expression* no contexto da linguagem Python. As expressões são representadas por listas seguindo um formato inspirado na notação original. Ou seja, a expressão $2 + 4 * 3$, seria convertida para $[''+'',\ 2,\ [''*'',\ 4,\ 3]]$.

[3] https://github.com/lark-parser/lark

3.5 Manipulação de dados

Plai utiliza Pandas como motor de manipulação de dados, que é uma biblioteca Python capaz de trabalhar com operações de leitura e escrita de diferentes formatos de dados, como CSV, Excel, JSON, SQL, entre outros. Além disso, o Pandas possui um motor otimizado para busca e manipulação de dados, que permite realizar agrupamentos, fusões, transformações e cálculos estatísticos (MCKINNEY, 2021). A biblioteca possui uma documentação extensa e rica, mas a sintaxe confusa costuma ser um problema durante o desenvolvimento.

A principal estrutura de dados disponibilizada pela biblioteca Pandas é o *DataFrame*, é uma estrutura bidimensional orientada a colunas que permite armazenar diferentes categorias de informações. Os *DataFrames* se assemelham a outras estruturas de dados tabulares como planilhas ou tabelas de bancos relacionais (TEAM, 2021b). O *DataFrame* é a principal estrutura de dados da linguagem Plai, possibilitando armazenar informações e realizar operações de manipulação.

3.6 Testes

O ambiente de execução dos testes utiliza o framework Pytest[4], que oferece um sistema de descoberta automatizada dos testes implementados e recursos de visualização de erros, facilitando o processo de configuração e execução.

Este trabalho tem como foco uma cobertura de teste mínima de 90% nos módulos que compõem a linguagem, para garantir uma maior confiabilidade na solução. Para realizar a análise de cobertura de código, foi utilizada a ferramenta Coverage.py[5] em conjunto com o Codecov[6] para apresentação do resultado.

O Coverage.py é uma ferramenta de análise de cobertura de testes para programas Python, que monitora quais trechos do código-fonte foram exercitados durante a execução dos testes. Provendo um relatório em diferentes formatos da cobertura geral de testes do sistema analisado. Já o Codecov consolida esses relatórios em uma plataforma *web*, permitindo analisar em detalhe os trechos de código cobertos ou não pelos testes.

O repositório de código do projeto foi configurado com um fluxo de integração contínua para assegurar uma maior qualidade do sistema. O sistema de integração contínua foi implementado utilizando a ferramenta *Github Actions*[7] e executa os testes unitários com diferentes versões do Python de forma automatizada a cada nova submissão ao repositório, como pode ser observado na Figura 6. Os testes são executados no ambiente Python

[4] https://pytest.org/
[5] https://coverage.readthedocs.io/
[6] https://codecov.io
[7] https://github.com/features/actions

em quatro versões diferentes: 3.6, 3.7, 3.8 e 3.9. Caso pelo menos uma das execuções falhe, a versão da aplicação com a nova alteração é considerada defeituosa.

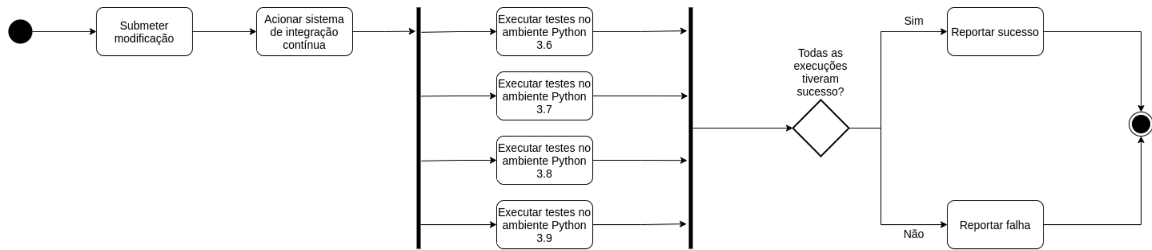

Figura 6 – Processo de integração contínua

4 Resultados

Esse capítulo apresenta os resultados atingidos com esse trabalho. As seções seguintes expõem a arquitetura geral da linguagem e como os componentes foram desenvolvidos e integrados, métricas do projeto, além de uma demonstração de uso da linguagem em comparação com métodos atuais de desenvolvimento.

4.1 Exemplo de uso

Tomando como exemplo um conjunto de dados que descreve as *issues* abertas no GitHub por linguagem de programação de 2011 a 2021, vamos demonstrar a implementação de uma sequência de operações de manipulação que resultem em um conjunto de dados que contenha a porcentagem de *issues* por linguagem relação ao total.

O trecho 4.1 demonstra como realizar a implementação utilizando a biblioteca Pandas na linguagem Python. É possível notar que a implementação se torna verbosa devido à maneira como as colunas do *dataframe* são acessadas e as atribuições realizadas para alterar os dados manipulados.

```
import pandas as pd

df = pd.read_csv('issues.csv')

df['name'] = df['name'].str.lower()
group_by_year = df.groupby('name', as_index=False).sum()
group_by_year['pct_issues'] = (
    group_by_year['count']/group_by_year['count'].sum()
) * 100

group_by_year.to_csv(
    'gh_issues_by_language.csv',
    columns=['name', 'count', 'pct_issues']
)
```

Código fonte 4.1 – Manipulação de dados em Python utilizando Pandas

Utilizando dos operadores especiais para acessar as colunas e a sintaxe de atribuição implementada pela linguagem Plai, as mesmas operações de manipulação podem ser feitas de forma concisa, como demonstra o trecho 4.2.

```
df = read_file('examples/issues.csv')

pipeline(df) as 'gh_issues_by_language.csv':
    .name.str.lower() as name
    $.groupby(.name, as_index=False).sum()
    (.count/.count.sum()) * 100 as pct_issues
    {.name, .count, .pct_issues}
```

Código fonte 4.2 – Manipulação de dados em Plai

4.2 Arquitetura

O interpretador de Plai foi desenvolvido em Python utilizando a biblioteca Lark e o código-fonte está organizado nos seguintes módulos:

plai.modules : contém a implementação das funções embutidas da linguagem, além das interfaces necessárias para estruturar o núcleo do sistema;

plai.environment : implementa a criação do ambiente de execução, incluindo as variáveis globais, tipos e funções;

plai.parser : engloba a declaração da gramática, a função que aplica o algoritmo LALR(1) no código de entrada, além das classes necessárias para realizar a redução da árvore sintática para a forma esperada pelo interpretador;

plai.symbol : inclui a classe utilizada para gerar objetos que representam os nomes de variáveis e funções;

plai.interpreter : declara a função responsável por executar a representação intermediária gerada a partir da etapa de análise sintática;

plai.validation : nesse módulo é implementada a validação dos dados de entrada e saída do processo de manipulação, com base na estrutura definida pelo usuário da linguagem.

4.3 Análise léxica e sintática

O processo de análise léxica e sintática possui duas etapas, como pode ser observado na Figura 7. A primeira consiste em aplicar o algoritmo LALR(1) no código-fonte de entrada, utilizando a gramática declarada no formato Lark. O resultado desse processo é uma árvore sintática abstrata que representa o conteúdo do código-fonte. Finalmente,

essa árvore sintática é reduzida utilizando a interface *Transformer* do Lark para gerar uma representação intermediária na forma de uma *S-expression*.

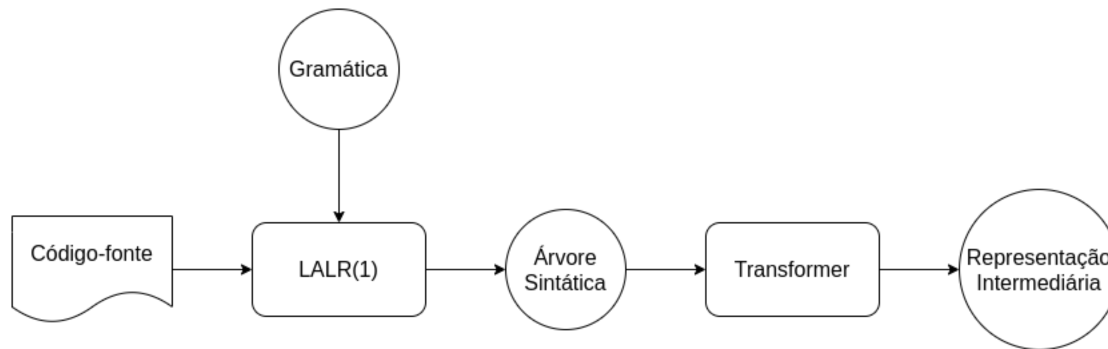

Figura 7 – Fluxo de análise léxica e sintática

4.3.1 Gramática

A gramática da linguagem foi definida utilizando a versão da notação EBNF implementada pela biblioteca Lark. A estrutura da gramática segue o padrão exemplificado no trecho 4.3, onde cada linha contém uma definição ou diretiva. As definições podem ser produções, declaradas em letras minúsculas, ou terminais, definidos em letra maiúscula. Já as diretivas são comandos especiais identificados pelo caractere inicial "%", que permitem, por exemplo, ignorar terminais, estender ou importar regras de outras gramáticas, entre outros (SHINAN, 2021a).

```
rule: <EBNF EXPRESSION>
    | etc.

TERMINAL: <EBNF EXPRESSION>

%ignore TERMINAL
```

Código fonte 4.3 – Exemplo de gramática aceita pela biblioteca Lark(SHINAN, 2021a)

A gramática estabelecida nesse trabalha busca apresentar uma sintaxe simples, focada na definição de tipos e operações em *DataFrames*, suportando também a avaliação de expressões, atribuição de variáveis e execução de funções. Nos parágrafos seguintes serão exploradas as declarações das principais regras da gramática.

A regra inicial da gramática especificada pelo nome *start* descrita no trecho 4.4, permite que a linguagem suporte uma sequência de declarações separadas por quebra de linha (*_NL*), sendo um caso comum quando a entrada é um arquivo. Essa regra também permite a avaliação de declarações únicas.

```
?start : (_NL | typed_stmt)*
       | typed_single_stmt _NL*
```

Código fonte 4.4 – Regra *start* da gramática

As produções *typed_stmt* e *typed_single_stmt* permitem especificar tipos para as instruções da linguagem, a gramática descrita no trecho 4.5, demonstra como as produções foram declaradas. A declaração é composta pelo terminal *NAME*, seguida dos caracteres " :: " e a instrução a qual o tipo será avaliado.

```
?typed_stmt : NAME "::" stmt
            | stmt
?typed_single_stmt : NAME "::" single_stmt
                   | single_stmt
```

Código fonte 4.5 – Regras que permitem definir tipos para a instruções da linguagem

A regra *stmt*, observada no trecho 4.6, é composta pelos diferentes tipos de instruções que a linguagem disponibiliza: expressões (*expr*), atribuição (*assignment*), operações de *alias* (*alias_expr*), declaração de tipos (*type_declaration*) e *pipelines* (*pipeline*).

```
?stmt : expr _NL
      | alias_expr _NL
      | assignment _NL
      | type_declaration _NL
      | pipeline
```

Código fonte 4.6 – Regra *stmt* da gramática

O trecho 4.7 expressa o conjunto de regras que formam as expressões. Na gramática a regra *atom_expr* representa os elementos básicos da linguagem, como números, *strings*, funções, variáveis, constantes, entre outros. Na cadeia de precedência dos termos, são primeiramente declaradas as regras das expressões aritméticas fundamentais: *term* para multiplicação e divisão; *arith_expr* para soma e subtração. Em seguida, operações de comparação, *comparison*. Por fim, expressões booleanas, *not_expr*, *and_expr* e *or_expr*. A natureza recursiva dessas regras permite compor expressões diversas resultando em uma árvore sintática que respeita a precedência e associatividade das operações.

```
?expr : or_expr

?or_expr : or_expr "or" and_expr
         | and_expr

?and_expr : and_expr "and" not_expr
```

```
          | not_expr

?not_expr : "not" not_expr
          | comparison

?comparison : arith_expr /(>=|<=|==|!=|>|<)/ arith_expr
            | arith_expr

?arith_expr : arith_expr /[+-]/ term
            | term

?term : term /(\*|\/{1,2})/ atom_expr
      | atom_expr
```

Código fonte 4.7 – Regras das expressões da gramática

Uma regra importante que faz parte dos elementos básicos da linguagem é a *sugar_column*. Uma expressão que permite acessar os valores de uma determinada coluna de um *DataFrame* pelo nome. Por exemplo, é possível recuperar as informações da coluna *bar* através da sintaxe *.bar* ou *."bar"*. O trecho 4.8 expõe a definição da regra.

```
sugar_column : "." NAME
             | "." string
```

Código fonte 4.8 – Definição da regra *sugar_column*

A regra *type_declaration*, descrita no trecho 4.9, especifica como a declaração de tipo do *DataFrame* deve ser expressa. Sendo uma regra simples, onde a palavra-chave *type* precede a atribuição do tipo para uma variável.

```
type_declaration : "type" NAME "=" expr
```

Código fonte 4.9 – Declaração de tipo do DataFrame

A instrução de *alias_expr* permite realizar operações com colunas de *DataFrames* atribuindo o resultado da operação a uma nova coluna. Por exemplo, a declaração 4.10, carrega na coluna foo o resultado da soma de 1 na coluna bar.

```
.bar + 1 as foo
```

Código fonte 4.10 – Exemplo de uso de instrução *alias*

As *alias_expr* são formadas por expressões declaradas em conjunto com a palavra-chave *as* e o terminal *NAME*, como descreve trecho 4.11.

```
alias_expr : expr "as" NAME
```

Código fonte 4.11 – Regra gramatical para *alias_expr*

A principal instrução da linguagem é a *pipeline*, que especifica um bloco código-fonte para manipulação de dados. A instrução é identificada pela palavra-chave *pipeline*, recebendo como argumento o *DataFrame* alvo das manipulações declaradas no bloco. O bloco deve possuir uma indentação em relação à declaração da instrução principal, seguindo o mesmo princípio dos blocos na linguagem Python. O trecho 4.12 apresenta um exemplo de declaração utilizando a instrução *pipeline*.

```
pipeline(dataframe):
    .bar + 1 as foo
```

Código fonte 4.12 – Exemplo de uso da instrução *pipeline*

A gramática que forma essa expressão possui uma variação que permite que o resultado da *pipeline* seja exportado para uma variável ou um arquivo, como será explorado na seção 4.4. A exportação é identificada pela palavra-chave *as* logo após a declaração dos argumentos da *pipeline*. O trecho 4.13 demonstra a regra gramatical para as duas variações dessa instrução.

```
pipeline: "pipeline" "("pipeline_args")" ":" suite
        | "pipeline" "("pipeline_args")" "as" (var|string) ":" suite
```

Código fonte 4.13 – Gramática da instrução *pipeline*

A declaração de todas as regras e terminais da linguagem podem ser analisados no apêndice A. Para sumarizar a sintaxe da linguagem, o trecho 4.14 apresenta uma sequência de declarações que realizam a leitura de um arquivo utilizando uma função e a criação de uma *pipeline*.

```
filepath = 'file.csv'
df = pd.read_csv(filepath)

pipeline(df) as 'new_file.csv':
    .name + '_foo' as foo_name
    .number + 1 as new_number
```

Código fonte 4.14 – Exemplo de sintaxe para manipulação de dados

4.3.2 Representação intermediária

Para gerar a estrutura de S-expression com base na árvore sintática resultante da etapa de análise sintática, foi definida a classe *PlaiTransformer* a partir da interface

Transformer disponibilizada pela biblioteca Lark. É possível dividir os nós da árvore sintática em dois grupos gerais: terminais e não terminais.

Os nós da árvore sintática que representam as regras terminais são avaliados no início da execução, pois compõem as folhas da árvore. Essas regras são: definição de nome, string, constantes booleanas e de valor nulo. Os tokens que simbolizam nomes, ou seja, variáveis ou funções são transformados em objetos da classe Symbol, contendo como atributo o valor do token original. Já as demais regras terminais são convertidos para suas representações equivalentes na linguagem Python.

Regras não terminais são avaliadas respeitando a ordem de precedência definida na gramática, retornando um conjunto de listas aninhadas, equivalente à estrutura S-expression. Em termos gerais, a lista retornada por cada tipo de transformação possui um operador identificador daquele tipo como primeiro elemento da lista, e os argumentos seguintes como os demais elementos.

Para exemplificar o fluxo de transformação da árvore sintática para a representação intermediária, considere a declaração do trecho 4.15. A partir dessa declaração a árvore sintática 4.16 é gerada. Por fim, a representação intermediária apresentada no trecho 4.17 é resultado do processo de redução da árvore sintática utilizando a classe *PlaiTransformar*.

```
foo = 40
foo + 2
```

Código fonte 4.15 – Exemplo de declaração na linguagem plai

```
start
  assignment
    foo
    number      40
  bin_op
    var  foo
    +
    number      2
```

Código fonte 4.16 – Exemplo de árvore sintática

```
[begin, [=, foo, 40], [+, foo, 2]]
```

Código fonte 4.17 – Exemplo de representação intermediária

4.4 Interpretação

A interpretação é realizada pela função recursiva *eval*, que percorre a representação intermediária de maneira Top-Down. Esse processo pode ser dividido nas seguintes categorias de operações: avaliação de funções e variáveis, atribuições, operações tipadas, manipulação de colunas e declaração de *pipelines*. A estrutura básica da interpretação dessas diferentes categorias é semelhante, primeiro a função *eval* identifica a operação a partir do primeiro elemento da lista que forma a representação intermediária e realiza o desempacotamento de iteráveis (BRANDL, 2007) para avaliar os demais elementos que compõem a estrutura.

Para compreender o fluxo considere a operação $2 + x$ que gera a representação intermediária descrita em 4.18. A interpretação dessa expressão inicia com a atribuição do operando à variável *head* e seus argumentos à $args$, como pode ser observado no trecho 4.19. O interpretador então executa a função *eval* no operador para determinar o tipo da operação, em seguida cada um dos argumentos são avaliados, por fim a operação é executada com os valores correspondentes a cada argumento.

```
[+, 2, x]
```

Código fonte 4.18 – S-expression da expressão $2 + x$

```
head, *args = sexpr

proc = eval(head, environment, **kwargs)
vals = [eval(arg, environment, **kwargs) for arg in args]

return proc(*vals)
```

Código fonte 4.19 – Avalição do caso geral da linguagem plai.

Existem formas especiais da linguagem que não podem ser avaliadas utilizando a mesma estratégia apresentada no trecho 4.19, pois produzem efeitos colaterais ou são operações que exigem uma implementação específica, tal como uma atribuição de variável, declaração de *pipelines*, entre outros.

É importante destacar três tipos de operações que se alinham ao propósito desse trabalho: operações tipadas, definidas seguindo a gramática do trecho 4.5; manipulação de colunas, realizada principalmente através das instruções *sugar_column* (4.8) e *alias_expr* (4.11); e declaração de *pipelines*, como descrito no trecho 4.13.

As operações tipadas asseguram que o *DataFrame* resultante da operação corresponde ao esquema de dados esperado, tanto em relação ao tipo do dado quanto a presença

da informação. Para isso, o interpretador separa os argumentos desse tipo de operação em dois: a declaração do esquema e o *DataFrame* a ser checado.

O esquema se trata de um dicionário, onde as chaves correspondem a coluna a ser checada e o valor é o tipo de dado esperado para aquela coluna. A validação acontece na chamada da função *validate_schema*, que será explorada na seção 4.5 desse documento. No caso de um esquema inválido a exceção *ValueError* é retornada com a mensagem de erro que descreve as colunas inválidas, como pode ser observado no trecho 4.20.

```
head, *args = sexpr

if head == Symbol.TYPED:
    type_def, args = args
    dataframe = eval(args, environment, **kwargs)

    schema = eval(type_def, environment, **kwargs)
    validation = validate_schema(dataframe, schema)

    if 'errors' in validation:
        msg = '\n'.join(validation['errors'])
        raise ValueError(msg)

    return dataframe
```

Código fonte 4.20 – Avaliação da validação de esquema dos DataFrames.

A avaliação da *pipeline* segue o princípio de não alterar o *DataFrame* original, retornando uma nova referência a cada modificação, o trecho 4.21 demonstra a declaração de uma *pipeline* em Plai. A interpretação consiste em avaliar em sequência cada instrução que compõe o bloco, tendo como argumento o *DataFrame* resultante da instrução anterior, como pode ser observado no trecho 4.22.

```
pipeline(df):
    .foo + 'bar' as foo_bar_column
```

Código fonte 4.21 – Exemplo de *pipeline*

```
if head == Symbol.PIPELINE:
    pipeline_args, block = args
    dataframe = eval(*pipeline_args, environment, **kwargs)

    for stmt in block:
        dataframe = eval(stmt, environment,
```

```
                              dataframe=dataframe, **kwargs)
    return dataframe
```

Código fonte 4.22 – Interpretação das pipelines na linguagem plai.

No bloco que compõe a *pipeline* é possível realizar operações em colunas utilizando a sintaxe *.column_name* ou *."column_name"*. Essas declarações criam um objeto interno do tipo *Col* que realiza operações com funções ou expressões que alterem a coluna alvo.

Outra variação de declaração de *pipeline* permite que o resultado da instrução seja exportado para uma variável ou um arquivo, conforme o exemplo 4.23. Caso o alvo da exportação seja uma variável, o resultado será carregado no ambiente de execução na variável de destino. Já para a saída no formato de arquivo, primeiro é avaliado qual o tipo do arquivo de destino conforme a extensão especificada, em seguida é utilizada a função de exportação adaptada ao formato. No momento, a linguagem Plai suporta somente exportações para arquivos no formato CSV.

```
pipeline(df) as 'result.csv':
    .foo + 'bar' as foo_bar_column
```

Código fonte 4.23 – Exemplo de *pipeline* com exportação

4.5 Validações

A validação estrutural dos *DataFrames* em tempo de execução acontece através da função *validate_schema* que faz parte do módulo *plai.validation*. São utilizadas as funções de introspecção de tipo de dados disponibilizadas pela biblioteca Pandas, que possibilita aferir se uma estrutura de dados é do tipo esperado (TEAM, 2021a). A função de instrospecção correspondente ao tipo esperado é executada para cada coluna a ser checada. Caso a coluna não exista ou não seja do tipo esperado, uma mensagem de erro apropriada é retornada pela função.

4.6 Métricas do projeto

O projeto possui um total de 1290 linhas de código Python, divididas conforme a Tabela 1, os testes automatizados representam cerca de 68% desse total. Foram feitos 179 *commits* e criados 35 *Pull Requests*.

No módulo *plai.parser* a gramática está no formato lark, logo não foi contabilizada na tabela 1, mas possui um total de 71 linhas.

Módulo	Número de linhas de código
plai.modules	76
plai.parser	93
plai.environment	45
plai.interpreter	110
plai.symbol	28
plai.validation	20
plai.__main__	37
tests/	881
Total	1290

Tabela 1 – Número de linhas de código por módulo.

4.6.1 Cobertura de testes

A cobertura de testes atual do projeto é de 89,19%, foram um total de 153 casos de testes implementados. A Figura 8 apresenta o relatório gerado pelo Codecov[1], onde é possível observar que os principais módulos da linguagem possuem uma cobertura superior a 96%. Apesar disso, o módulo *__main__* não possui testes. Esse módulo contem funções que compõem a ferramenta de linha de comando da linguagem, permitindo avaliar arquivos ou interagir com um terminal iterativo para executar instruções da linguagem.

/ plai

Files	☰	●	●	●	Coverage
modules	74	72	0	2	97.30%
parser	91	89	0	2	97.80%
__init__.py	2	2	0	0	100.00%
__main__.py	33	0	0	33	0.00%
environment.py	18	18	0	0	100.00%
interpreter.py	109	107	0	2	98.17%
symbol.py	28	27	0	1	96.43%
validation.py	15	15	0	0	100.00%
Folder totals (14 files)	370	330	0	40	89.19%
Project totals (14 files)	370	330	0	40	89.19%

Figura 8 – Relatório de cobertura de testes

[1] https://app.codecov.io/gh/matheusbsilva/plai

5 Conclusão

Dado a natureza volátil dos dados, é comum aplicações responsáveis por realizar manipulações de dados se tornarem origens de dívidas técnicas. O uso de linguagens de programação de propósito geral para o desenvolvimento desses sistemas contribui para esse cenário, devido à ineficiência desse tipo de linguagem em expressar soluções para domínios complexos.

A linguagem de domínio específico apresentada nesse trabalho possui os componentes básicos de uma linguagem interpretada para o desenvolvimento de *pipelines* de manipulação de dados. A gramática desenvolvida apresenta uma sintaxe concisa para a implementação dessas operações, o ambiente de interpretação desenvolvido em Python utiliza como representação intermediária uma adaptação da notação *S-expression* e a biblioteca Pandas como motor de manipulação de dados. A linguagem permite realizar a validação estrutural conforme os tipos de dados suportados pela biblioteca Pandas, para os dados de entrada e também os resultantes de operações de manipulação.

A sintaxe sucinta e alinhada ao âmbito de manipulação de dados, busca tornar mais eficiente a implementação e manutenção desse tipo de sistema, além de que a estrutura de validação de esquema dos dados de entrada e saída permite criar aplicações mais previsíveis.

Referências

AHO, A. V. et al. In: *Compilers: principles, techniques, & tools*. 2nd. ed. [S.l.]: Pearson/Addison Wesley, 2006. ISBN 978-0-321-48681-3. Citado 4 vezes nas páginas 22, 23, 25 e 26.

AKKIRAJU, R. et al. Characterizing machine learning process: A maturity framework. *CoRR*, abs/1811.04871, 2018. Disponível em: <http://arxiv.org/abs/1811.04871>. Citado na página 18.

BACKUS, J. W. et al. Revised report on the algorithm language algol 60. *Commun. ACM*, Association for Computing Machinery, New York, NY, USA, v. 6, n. 1, jan. 1963. ISSN 0001-0782. Disponível em: <https://doi.org/10.1145/366193.366201>. Citado na página 25.

BECK, K. *Test-driven development: by example*. [S.l.]: Addison-Wesley, 2003. Citado na página 29.

BEYER, B. et al. *Site Reliability Engineering: How Google Runs Production Systems*. [s.n.], 2016. Disponível em: <http://landing.google.com/sre/book.html>. Citado na página 17.

BRANDL, G. *PEP 3132 – Extended Iterable Unpacking*. 2007. <https://www.python.org/dev/peps/pep-3132/>. Citado na página 41.

DAVENPORT, T. H.; RONANKI, R. Artificial intelligence for the real world. 2018. ISSN 0017-8012. Disponível em: <https://hbr.org/2018/01/artificial-intelligence-for-the-real-world>. Citado na página 17.

HAUSSER, R. Foundations of Computational Linguistics: Man-Machine Communication in Natural Language. In: . Berlin, Heidelberg: Springer Berlin Heidelberg, 1999. ISBN 978-3-662-03920-5. Disponível em: <http://public.ebookcentral.proquest.com/choice/publicfullrecord.aspx?p=3097950>. Citado na página 24.

HOPCROFT, J. E.; MOTWANI, R.; ULLMAN, J. D. Introduction to automata theory, languages, and computation. In: . 3rd. ed. [S.l.]: Pearson/Addison Wesley, 2007. ISBN 978-0-321-45536-9 978-0-321-46225-1 978-0-321-45537-6. Citado 2 vezes nas páginas 24 e 25.

KADADI, A. et al. Challenges of data integration and interoperability in big data. In: *2014 IEEE International Conference on Big Data (Big Data)*. [S.l.: s.n.], 2014. p. 38–40. Citado na página 17.

KOSAR, T. et al. Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments. v. 17, p. 276–304, 2012. Citado na página 18.

LEE, I. Big data: Dimensions, evolution, impacts, and challenges. *Business Horizons*, v. 60, n. 3, p. 293–303, maio 2017. ISSN 00076813. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S0007681317300046>. Citado na página 17.

LEITÃO, A. M. From lisp s-expressions to Java source code. *Computer Science and Information Systems*, v. 5, n. 2, p. 19–38, 2008. ISSN 18200214. Citado na página 31.

LI, C. Preprocessing Methods and Pipelines of Data Mining: An Overview. *arXiv:1906.08510 [cs, stat]*, jun. 2019. ArXiv: 1906.08510. Disponível em: <http://arxiv.org/abs/1906.08510>. Citado 2 vezes nas páginas 21 e 22.

MCKINNEY, W. *pandas: a python data analysis library*. 2021. <https://pandas.pydata.org/>. Citado na página 32.

PARR, T. In: *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. 1. ed. [S.l.]: Pragmatic Bookshelf, 2010. ISBN 978-1-934356-45-6. Citado 3 vezes nas páginas 22, 26 e 27.

PRESTON-WERNER, T. *Semantic versioning 2.0.0*. 2013. <=>https://semver.org/. Citado na página 30.

SCULLEY, D. et al. Machine learning: The high interest credit card of technical debt. In: *SE4ML: Software Engineering for Machine Learning (NIPS 2014 Workshop)*. [S.l.: s.n.], 2014. Citado na página 18.

SHINAN, E. *Grammar Reference*. 2021. <https://lark-parser.readthedocs.io/en/latest/grammar.html>. Citado na página 36.

SHINAN, E. *Parsers*. 2021. <https://lark-parser.readthedocs.io/en/latest/parsers.html#lalr-1>. Citado na página 31.

SHINAN, E. *Transformers and Visitors*. 2021. <https://lark-parser.readthedocs.io/en/latest/visitors.html#transformer>. Citado na página 31.

TEAM, P. development. *General utility functions*. 2021. <https://pandas.pydata.org/docs/reference/general_utility_functions.html#dtype-introspection>. Citado na página 43.

TEAM, P. development. *Intro to data structures*. 2021. <https://pandas.pydata.org/pandas-docs/stable/user_guide/dsintro.html>. Citado na página 32.

VASSILIADIS, P.; SIMITSIS, A.; SKIADOPOULOS, S. Conceptual modeling for ETL processes. In: *Proceedings of the 5th ACM international workshop on Data Warehousing and OLAP - DOLAP*. ACM Press, 2002. Disponível em: <https://doi.org/10.1145%2F583890.583893>. Citado na página 17.

WIRTH, N. What can we do about the unnecessary diversity of notation for syntactic definitions? *Communications of the ACM*, v. 20, n. 11, p. 822–823, nov. 1977. ISSN 00010782. Disponível em: <http://portal.acm.org/citation.cfm?doid=359863.359883>. Citado na página 25.

Apêndices

APÊNDICE A – Gramática EBNF da linguagem Plai

```
?start : (_NL | typed_stmt)*
       | typed_single_stmt _NL*

?typed_stmt : NAME "::" stmt -> typed_stmt
      | stmt

?stmt : expr _NL
      | alias_expr _NL
      | assignment _NL
      | type_stmt _NL
      | pipeline

?typed_single_stmt : NAME "::" single_stmt -> typed_stmt
                   | single_stmt

?single_stmt : (expr | alias_expr | assignment | type_stmt)

assignment : NAME "=" expr

type_stmt : "type" NAME "=" expr

arguments : expr("," expr)* "," kwargs -> mix_args
          | expr("," expr)* -> posargs
          | kwargs

kwargs : argpair("," argpair)*

argpair : NAME "=" expr

pipeline : "pipeline" "(" pipeline_args ")" ":" suite
         | "pipeline" "(" pipeline_args ")" "as" (var | string) ":" suite

pipeline_args : expr
```

```
suite : _simple_stmt | _NL _INDENT typed_stmt+ _DEDENT

_simple_stmt : typed_single_stmt(";" typed_single_stmt)*

alias_expr : expr ("as" var)

?expr: or_expr

?or_expr : or_expr "or" and_expr
         | and_expr

?and_expr : and_expr "and" not_expr
          | not_expr

?not_expr : "not" not_expr -> not_op
          | comparison

?comparison : arith_expr _comp_op arith_expr -> bin_op
            | arith_expr

?arith_expr : arith_expr _sum_op term -> bin_op
            | term

?term : term _mult_op atom_expr -> bin_op
      | atom_expr

sugar_column : "." NAME
             | "." string

?atom_expr : atom_expr "(" arguments? ")" -> function_call
           | atom_expr "." NAME -> attr_call
           | "$" "." NAME -> dataframe_attr_call
           | sugar_column
           | atom


?atom : "(" expr ")"
      | "[" [expr ("," expr)*] "]" -> list_expr
```

```
      | "{" key_content "}"
      | number
      | string
      | var
      | "True" -> const_true
      | "False" -> const_false
      | "None" -> const_none

?key_content : _key_value_list? -> dict_expr
             | [expr ("," expr)*] -> slice_df_expr

_key_value_list : key_value("," key_value)*[","]
key_value : expr ":" expr

number : DEC_NUMBER
       | HEX_NUMBER
       | BIN_NUMBER
       | OCT_NUMBER
       | FLOAT_NUMBER
       | IMAG_NUMBER

var : NAME

string: STRING | LONG_STRING

!_sum_op :  "+" | "-"
!_mult_op : "*" | "/" | "//"
!_comp_op : "<" | ">" | "==" | ">=" | "<=" | "!="

_NL: /(\r?\n[\t ]*)+/

%import common.WS_INLINE
%import python (COMMENT, NAME, STRING, LONG_STRING)
%import python (DEC_NUMBER, FLOAT_NUMBER)
%import python (HEX_NUMBER, OCT_NUMBER, BIN_NUMBER, IMAG_NUMBER)

%declare _INDENT _DEDENT
%ignore WS_INLINE
%ignore COMMENT
```