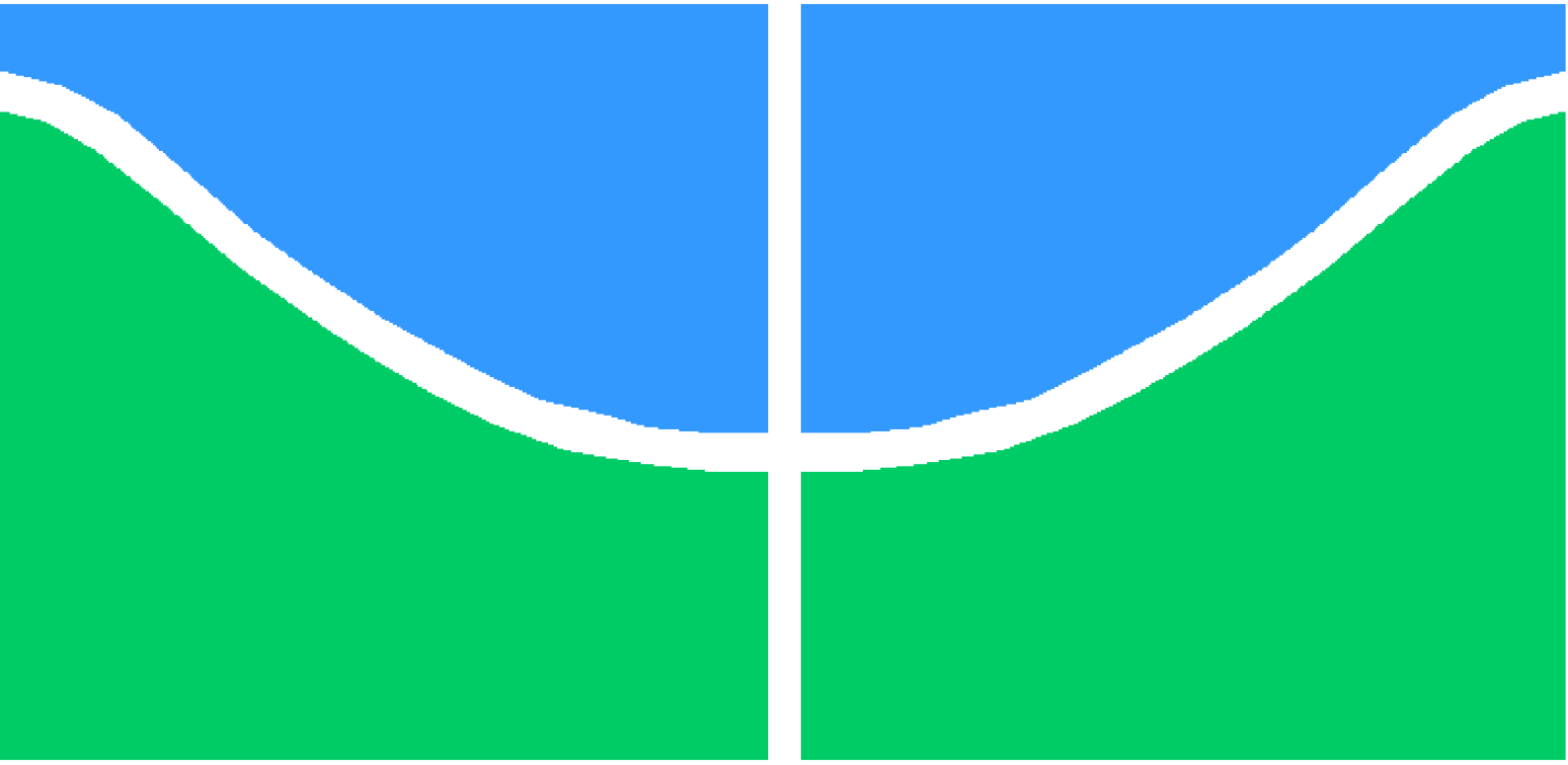

Universidade de Brasília – UnB
Faculdade UnB Gama – FGA
Engenharia de Software

FRED: Um formato para representação e troca de dados

Autor: Matheus de Sousa Bernardo
Orientador: Prof. Dr. Fábio Macêdo Mendes

Brasília, DF
2019

Matheus de Sousa Bernardo

FRED: Um formato para representação e troca 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: Prof. Dr. Fábio Macêdo Mendes

Brasília, DF

2019

Matheus de Sousa Bernardo
FRED: Um formato para representação e troca de dados/ Matheus de Sousa Bernardo. – Brasília, DF, 2019-
79 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Fábio Macêdo Mendes

Trabalho de Conclusão de Curso – Universidade de Brasília – UnB
Faculdade UnB Gama – FGA , 2019.

1. representação de dados. 2. troca de dados. I. Prof. Dr. Fábio Macêdo Mendes. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. FRED: Um formato para representação e troca de dados

CDU 02:141:005.6

Matheus de Sousa Bernardo

FRED: Um formato para representação e troca 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, 12 de Julho de 2019 – Data da aprovação do trabalho:

Prof. Dr. Fábio Macêdo Mendes
Orientador

Prof. Dr. Renato Coral Sampaio
Convidado 1

Prof. Dr. Bruno César Ribas
Convidado 2

Brasília, DF
2019

Resumo

Este projeto propõe a especificação e a implementação de um formato para troca e representação de dados chamado FRED. Essa notação foi especificada com uma sintaxe inspirada em JSON, porém evolui o modelo de dados com tipos adicionais (ex: dados representado data, hora e blobs binários). Além disso, a especificação prevê um mecanismo de extensão inspirado em XML e também com influência de outros formatos correlatos, e inclui a possibilidade de anotar valores com tags e metadados. Já a implementação do formato FRED foi realizada no projeto em duas linguagens, Haskell e JavaScript, e também foi desenvolvida uma suíte de testes para validar a implementação de acordo com a especificação. Por fim foi realizada uma comparação simples entre FRED e outros formatos.

Palavras-chave: representação de dados. troca de dados. json. xml.

Abstract

This project proposes the specification and implementation of a format for data exchange and representation called FRED. This notation has a syntax influenced by JSON, although it extends the data model of JSON with more types. It is possible to represent Date, Time and Binary Blobs in FRED. Furthermore it has an extension mechanism inspired by XML and other data formats, it also includes the ability to associate a value with tags and metadata. FRED implementation was done in two languages, Haskell and JavaScript. A test suite was also created with the objective of validating the implementation according to the specfication. Finally, a simple comparison was executed between FRED and other formats.

Keywords: data representation. data exchange. json. xml.

Lista de ilustrações

Lista de códigos

Lista de tabelas

Lista de abreviaturas e siglas

XML	*Extensible Markup Language*
SGML	*Standard Generalized Markup Language*
JSON	*JavaScript Object Notation*
FRED	*Flexible REpresentation of Data*

Lista de símbolos

:	Expressão deriva no contexto de gramáticas livre do contexto
$\Rightarrow$	Substituição de um não terminal por um terminal

Sumário

1 Introdução

Formatos utilizados para representar dados possuem muita importância em vários sistemas computacionais, pois a maneira em que eles são organizados e o poder de representação influem tanto na leitura e escrita por humanos como também na interpretação pelos computadores.

Com o advento da internet, a comunicação entre sistemas aumentou. Por isso, foi necessária a criação de formatos padronizados para representar os dados utilizados na comunicação de sistemas que frequentemente adotam tecnologias diferentes e diferentes linguagens de programação. Os dois formatos mais utilizados hoje em dia são XML e JSON.

O XML foi concebido a partir de uma linguagem de marcação, SGML. Um de seus objetivos é ser de fácil utilização na internet e representam documentos utilizando a metáfora de marcações de texto (BRAY et al., 2008).

O formato JSON foi criado com base na representação de objetos da linguagem JavaScript e é um subconjunto da linguagem, que possui uma sintaxe simples e tem como objetivo ser fácil tanto para humanos ler e escrever como para máquinas interpretarem (Ecma International, 2017).

Nesse projeto especificamos o FRED (*Flexible REpresentation of Data*), um formato para representação e troca de dados. Este formato possui uma sintaxe inspirada no JSON porém incrementa o modelo de dados utilizando algumas abstrações encontradas no XML.

Além disso, o FRED possui um modelo de dados um pouco mais completo que JSON, permitindo por exemplo dados sobre data e hora, entre outros definidos na especificação.

Visto que, XML representa dados com o foco em marcação e JSON possui um modelo de dados muito simples. O FRED possui uma sintaxe inspirada em JSON porém aumenta o modelo de dados e também permite extender este modelo mapeando naturalmente os dados representados para a forma em que eles são usados frequentemente. Isto é, facilitar a representação de dados por exemplo quando usados como classes, enums, entre outros.

1.1 Objetivos

1.1.1 Objetivo Geral

O objetivo principal deste trabalho é a construção de uma linguagem para representação e troca de dados, com base na sintaxe de JSON e utilizando de alguns conceitos do XML para expandir as abstrações desta nova linguagem.

1.1.2 Objetivos Específicos

- Especificar a linguagem de maneira formal.
- Implementar a linguagem com base na especificação.
- Implementar o *parser* em mais de uma linguagem de programação.
- Analisar e discutir a linguagem implementada.

1.2 Organização do trabalho

Este trabalho está organizado em três partes principais:

Referencial Teórico: Esta seção trata da fundamentação teórica utilizada no trabalho e tem o objetivo de fornecer a base para o desenvolvimento do projeto.

Metodologia: Esta seção define como será feito o trabalho e também o escopo do projeto.

Resultados: Esta seção aborda os resultados que foram alcançados no decorrer do projeto.

2 Referencial Teórico

2.1 Linguagens Formais

Na maior parte das linguagens naturais (ex. português, inglês, etc), letras, palavras e sentenças são entidades distintas. Também é verdade que existem agrupamentos, isto é, conjuntos de letras formam palavras e conjuntos de palavras formam sentenças. Contudo, nem todos esses conjuntos são válidos (COHEN, 1986).

Esse tipo de relação também é válido para linguagens em computadores. Por exemplo, na linguagem de programação *C*, conjuntos de caracteres formam comandos e um conjunto desses comandos formam programas. Estes também podem ser válidos ou não.

O estudo de linguagens formais busca entender como funcionam as linguagens de maneira unificada e utilizando regras bem definidas. Isto implica em definir sem ambiguidades, o que é uma linguagem e como validar uma linguagem além de propor algoritmos específicos para realizar esta análise. O formalismo também pressupõe que esta área não busca compreender a comunicação ou a interpretação subjetiva de textos.

Para definirmos uma linguagem de maneira formal, é necessário definir algumas abstrações. Uma destas são os Símbolos que são as unidades fundamentais de uma linguagem. Um conjunto de símbolos válidos é o chamado alfabeto. Uma sequência de símbolos é chamado *string* e um conjunto específico dessas *strings* quando consideradas bem formadas definem uma linguagem.

Deste modo, podemos definir uma linguagem como um conjunto contável de *strings* que usa símbolos de um alfabeto especificado. Usando essa definição o português é definido como o conjunto de todos os textos válidos em português (COHEN, 1986).

Entretanto, esse conjunto é infinito e portanto para definir esta linguagem de maneira precisa é necessário utilizar as regras gramaticais e definir a linguagem de maneira generativa utilizando uma descrição finita de regras.

Utilizando essa estratégia, frases em português que estão semanticamente erradas porém gramaticalmente corretas são consideradas válidas. Por exemplo, "Os dinossauros tem sua ética definida na tangente do verbo", são válidas para a linguagem, visto que linguagens formais não estudam o significado dos textos.

Para compor novas linguagens, a definição permite construir linguagens complexas a partir de outras linguagens mais simples. Para tal, é possível fazer algumas operações em linguagens e gerar outras mais abrangentes.

Por exemplo, dado duas linguagens: L definida por duas palavras **aa** e **bb** e D o

conjunto de dígitos de **0** a **9**. Podemos definir as operações nessas linguagens da seguinte maneira:

- A União entre linguagens é similar à operação em conjuntos, isto é a união de L e D significa um novo conjunto composto pelas palavras da linguagem L mais os dígitos da linguagem D, isto é, o conjunto com **aa**, **bb** e os dígitos de **0** a **9**.
- A Concatenação de linguagens é uma operação similar ao produto cartesiano em conjuntos, isto é, o novo conjunto é formado utilizando todas as palavras de L e combinando com as palavras da linguagem D. No caso as *strings* da nova linguagem possuem três caracteres onde os dois primeiros pertencem ao alfabeto de L e o terceiro ao alfabeto de D, por exemplo *aa4*.
- A operação *kleene closure* em uma linguagem consiste no conjunto de *strings* criadas concatenando a linguagem zero ou mais vezes, por exemplo aplicar esta operação na linguagem L significa construir uma nova linguagem onde é válido *strings* compostas por **aa** e **bb** de maneira concatenada, por exemplo *aabbaaaa*.

Existem outras operações mais complexas baseadas em recursão que serão discutidas adiante.

Todo esse formalismo permitiu o desenvolvimento de técnicas para interpretar linguagens por meio de computadores. Um exemplo disso é a área de compiladores. Duas partes fundamentais serão exploradas nas próximas seções, análise léxica e análise sintática, estas utilizam o formalismo para construir sistemas capazes de reconhecer linguagens e transformar em representações para o uso do computador.

2.2 Análise Léxica

O objetivo da análise léxica é ler uma sequência de caracteres como entrada, agrupar essa entrada em *strings* chamadas lexemas e então produzir como saída um conjunto de tokens para cada lexema construído. O analisador léxico também pode limpar espaços em brancos, comentários e outros caracteres que podem ser ignorados da entrada (AHO et al., 2006).

Um token é uma estrutura que possui um tipo, uma referência ao lexema e podem ter atributos como número de linha e/ou coluna. O tipo do token é um símbolo abstrato que representa uma unidade léxica, isto é, uma palavra chave ou uma sequência de caracteres que definem um identificador e o valor é algum atributo associado àquele token.

Podemos fazer uma analogia com o português, os lexemas são as palavras da língua e os tokens são a classe gramatical.

Outro conceito importante é o padrão de formação, este é uma descrição da forma em que lexemas de um token podem ocorrer na entrada. Portanto, para construir um lexema é necessário encontrar uma sequência de caracteres na entrada que corresponde a algum padrão predefinido. Por exemplo, um token que representa números inteiros pode corresponder a um padrão que agrupa qualquer sequência de dígitos em um único lexema.

Para construir um analisador léxico é necessário portanto especificar os padrões de formação para cada token. Isto é, descrever como devem ser os lexemas para cada token. Esse padrão é comumente especificado utilizando expressões regulares.

2.2.1 Expressões Regulares

Expressões regulares são uma notação utilizada para descrever um tipo específico de linguagem, as linguagens regulares. Linguagens regulares usam as operações descritas em 2.1 e associam uma sintaxe específica a cada uma destas três operações. As implementações encontradas na maioria das linguagens de programação são baseadas na sintaxe proposta pelo Perl e em alguns casos adicionam extensões não regulares à linguagem.

Essas expressões são construídas recursivamente a partir de expressões menores. A unidade básica dessa expressão são os símbolos de um alfabeto. Por exemplo, se o caractere a pertence a algum alfabeto então **a** é uma expressão regular que representa uma linguagem definida apenas pela palavra a.

Supondo que **r** e **s** são expressões regulares. Então, as operações básicas são representadas pela seguinte notação:

- (**r**)|(**s**) expressão regular que denota união entre as linguagens L(**r**) e L(**s**).
- (**r**)(**s**) expressão regular que denota concatenação entre as linguagens L(**r**) e L(**s**).
- (**r**)* expressão regular que representa a operação *kleene closure* na linguagem L(**r**).
- (**r**) expressão regular que representa a linguagem L(**r**). Os parentêses não mudam a linguagem representada, e servem apenas para desambiguar expressões. Por exemplo, na expressão ((**r**)|(**s**))* os últimos parentêses deixam explícito para qual expressão o operador *kleene* está sendo aplicado.

Por exemplo, dado que l representa todas as letras e o *underscore* aceitas pela linguagem C e que d representa todos as os dígitos aceitos pela linguagem C, então os identificadores da linguagem C podem ser descritos utilizando a expressão regular:

l(l|d)*

Isto significa que identificadores em C sempre começam com uma letra ou *underscore* e podem ter zero ou mais letras, *underscores* e dígitos subsequentes.

2.3 Análise Sintática

A análise léxica lê vários caracteres da entrada e os agrupa em uma sequência de tokens. Essa sequência de tokens não necessariamente define uma entrada válida e portanto é importante ter alguma maneira de validar a sintaxe dessa entrada.

A análise sintática é o processo onde é verificada a sintaxe da entrada. Para fazer essa verificação são utilizadas regras precisas que especificam uma estrutura sintática válida (AHO et al., 2006).

A especificação dessas regras pode ser feita utilizando uma notação chamada gramática livre de contexto, nela as regras de sintaxe são definidas de maneira precisa e fácil de entender.

Mais formalmente, uma gramática livre de contexto consiste em quatro partes. Sendo elas o conjunto dos terminais, o conjunto dos não terminais, o símbolo inicial e as produções.

Terminais: São os símbolos básicos que formam a sequência de caracteres da entrada. O analisador léxico retorna tokens que possuem terminais.

Não Terminais: Chamados de variáveis sintáticas denotam conjuntos de sequências de caracteres. Também impõe uma estrutura hierárquica na linguagem.

Símbolo Inicial: Diferencia um não terminal para ser considerado como inicial na interpretação da gramática. Corresponde à raiz da árvore sintática resultante.

Produção: Declara a maneira como os terminais e não terminais podem ser combinados para formar uma entrada válida. É dividido em três partes, como em a : β, o lado esquerdo que possui um não terminal, e o lado direito ou corpo da produção que, consiste de zero ou mais terminais e/ou não terminais e especifica a forma de construir uma sequência de caracteres para o não terminal referente a seu lado esquerdo.

A notação utilizada é uma *Backus Normal Form* extendida baseada na biblioteca Lark[1]. Nesta notação os não terminais são sempre escritos em caixa baixa e os terminais podem ser escritos de duas maneiras, literais simples utilizando aspas duplas ou terminais complexos escritos em caixa alta (LARK. . . , 2011).

[1] <https://github.com/lark-parser/lark>

A gramática exemplificada no Código 1 define uma linguagem capaz de representar expressões aritméticas simples. Os não terminais são: expr, term e factor. Os terminais são: "+", -", "*", "/", "(", ")"e ID (representa um identifador que é definido usando uma expressão regular). Por fim, o símbolo inicial é expr.

```
1  expr : expr "+" term
2       | expr "-" term
3       | term
4
5  term : term "*" factor
6       | term "/" factor
7       | factor
8
9  factor : "(" expr ")"
10         | ID
```

Código 1 – Exemplo de gramática livre do contexto que descreve expressões aritméticas como (20 + 1) * 2.

Para a construção de um analisador sintático (*parser*) é importante entender o conceito de derivação. A partir do símbolo inicial são feitos passos de reescrita onde um não terminal é substituído pelo corpo da sua produção.

Por exemplo, considere a primeira produção da gramática na linha 1 do Código 1. Essa linha pode ser lida da seguinte maneira "expr **deriva** expr + term". E isso significa que o não terminal expr pode ser substituído pela produção expr + term.

Por exemplo, a notação utilizada para representar essa substituição é.

$$\text{expr} \Rightarrow 40 + 2$$

Além disso, essas substituições podem ser aplicadas de forma sequencial. Por exemplo,

$$\text{expr} \Rightarrow \text{expr} + \text{term} \Rightarrow \text{term} + \text{factor} \Rightarrow \text{factor} + 2 \Rightarrow 40 + 2$$

Essa sequência de substituições é denominada derivação e possui papel fundamental para entender como construir um analisador sintático (AHO et al., 2006).

Outro conceito importante para a construção de um analisador sintático são as *parse trees* que representam uma derivação utilizando a estrutura de dados árvore, pode-se dizer que essa árvore também representa a entrada.

Utilizando a gramática especificada no Código 1, porém com uma pequena alteração onde os não terminais são representados apenas pelas suas primeiras letras, teríamos a Fig. 1. Essa árvore equivale à derivação a seguir:

$$\text{E} \Rightarrow \text{E} + \text{T} \Rightarrow \text{T} + \text{T} \Rightarrow \text{F} + \text{T} \Rightarrow 40 + \text{T} \Rightarrow 40 + \text{F} \Rightarrow 40 + 2$$

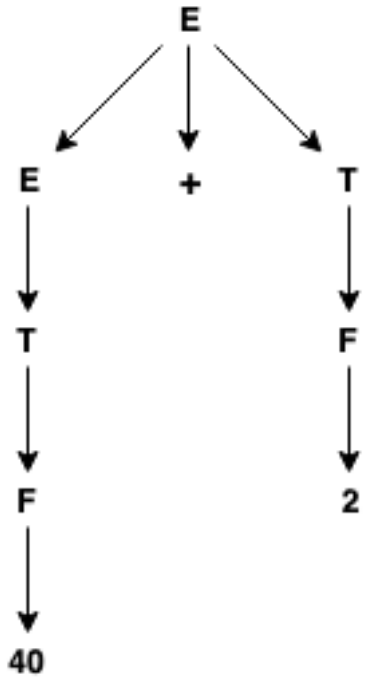

Figura 1 – Exemplo de Parse Tree.

Existem três tipos genéricos de *parsers* para gramáticas livre de contexto: universal, *top-down* e *bottom-up*. Os métodos de *parsing* universais podem analisar qualquer gramática. Porém, muitas vezes sua complexidade é exponencial, por exemplo o algoritmo Earley, um dos mais eficientes nesta categoria, tem no pior caso complexidade assintótica $O(n^3)$.

Normalmente os métodos mais utilizados são *top-down* ou *bottom-up*. Onde, no primeiro, a *parse tree* é construída a partir do topo, da raiz até as folhas. E no segundo é o inverso, das folhas até a raiz.

Vale ressaltar que os analisadores mais eficientes tanto *top-down* ou *bottom-up* só funcionam para subclasses de gramáticas livres do contexto. Este documento foca nos analisadores do tipo *top-down*. Visto que, um dos os objetivos do projeto é definir um formato de fácil construção utilizando técnicas *top-down*.

2.3.1 Analisadores *Top-Down*

A construção de analisadores do tipo *top-down* é como o processo de construir a *parse tree* a partir do topo e isto é equivalente a encontrar a derivação mais à esquerda para a entrada (AHO et al., 2006).

Para exemplificar, considere a gramática descrita no Código 2. Esta representa uma linguagem para expressões aritméticas parecida com a definida no Código 1, porém sofreu mudanças para adequar ao processo de *parsing top-down*.

Uma dessas mudanças foi retirar recursões à esquerda da gramática, pois esta leva a problemas durante a análise (AHO et al., 2006).

```
1  e  : t e'
2
3  e' : "+" t e'
4     | ϵ
5
6  t  : f t'
7
8  t' : "*" f t'
9     | ϵ
10
11 f  : "(" e ")"
12    | ID
```

Código 2 – Exemplo de gramática para analisadores top-down onde o símbolo epsilon representa uma produção vazia.

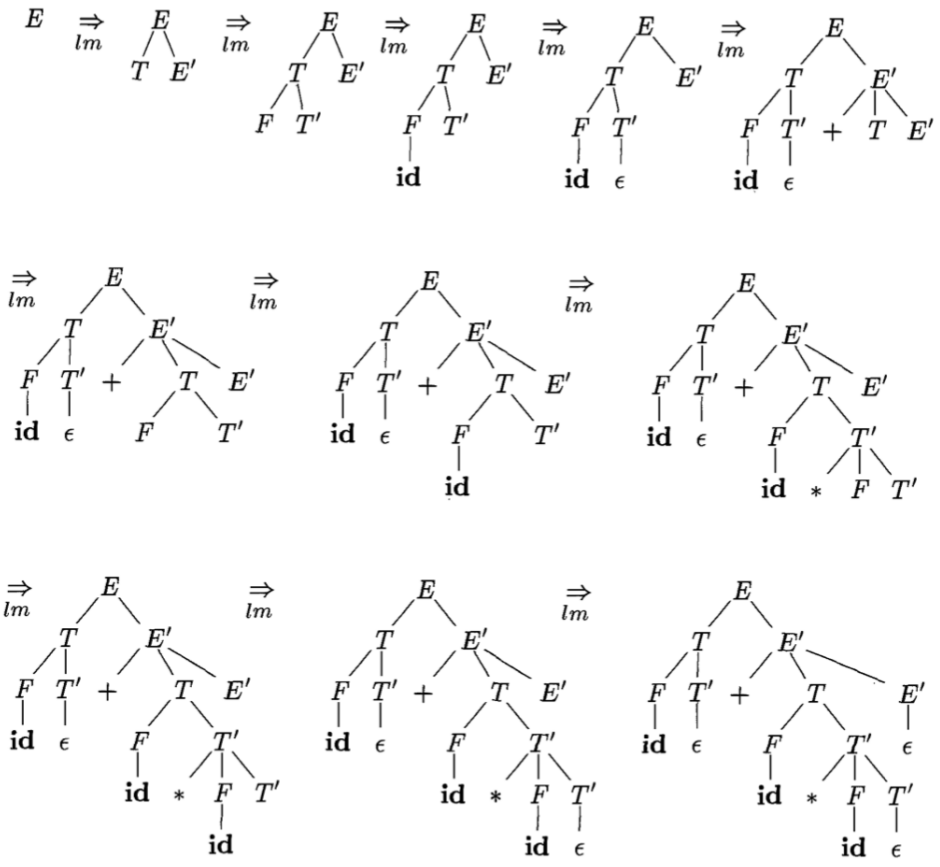

Figura 2 – Exemplo de parse top-down (AHO et al., 2006).

A análise sintática a partir do topo consiste em cada etapa determinar qual produção aplicar para o respectivo não terminal. Depois que uma produção for escolhida

o processo resume-se a combinar os terminais no corpo da produção com os tokens da entrada.

Um exemplo de análise *top-down* pode ser visualizado na Figura 2, onde existem várias árvores de derivação representando cada etapa do processo de análise para a entrada **id + id * id**. Este processo utiliza a gramática definida no Código 2.

Descida recursiva é uma maneira simples e genérica para construir um *parser top-down*. Nesse tipo de analisador existe um conjunto de procedimentos para cada não terminal (AHO et al., 2006).

A execução começa chamando recursivamente o procedimento para o símbolo inicial e então termina a execução com sucesso se a execução ler toda a entrada.

Esse tipo de analisador pode necessitar de *backtracking*. Isto é, ler a entrada toda novamente repetidas vezes. Caso o caminho tomado durante o momento de escolher uma produção falhar, é necessário reiniciar a execução usando outra produção. A necessidade de recomeçar a execução pode elevar o tempo de execução para valores exponenciais (AHO et al., 2006).

Um exemplo de procedimento para um não terminal pode ser visto no Código 3.

```
1  def E(tokens):
2      return [T(tokens), E_(tokens)]
3
4  def E_(tokens):
5      if tokens[0] == "+":
6          plus = tokens.pop(0)
7          return [plus, T(tokens), E_(tokens)]
8      else:
9          return None
10
11 ...
```

Código 3 – Exemplo de procedimento em pseudo-python para um não terminal

2.3.2 *Parser Combinators*

Outro método para construir *parsers top-down* bastante comum, principalmente em linguagens funcionais, são os chamados *parsers combinators*, esse modelo é construído a partir de um analisador com descida recursiva (HUTTON; MEIJER, 1996).

Essa técnica utiliza funções como *parsers* e define outras funções de alta ordem que implementam abstrações das gramáticas livres de contexto, isto é, sequências, escolha e repetição.

O método consiste em criar pequenos *parsers* para as produções da gramática

como funções. E utilzar as funções que representam as abstrações das gramáticas livres de contexto para combinar esses parsers.

Podemos definir um parser como uma função que recebe como entrada uma lista de *tokens* e tem como saída uma árvore e o restante da entrada. Um exemplo simples de parser pode ser visto no Código 4, nele o primeiro item da lista é consumido apenas se o primeiro item for um espaço em branco. Caso isso não ocorra o parser falha.

```
1 def whitespace(tokens):
2     if not tokens or tokens[0].type != SPACE:
3         raise SyntaxError
4     else:
5         return tokens[0], tokens[1:]
```

Código 4 – Exemplo de parser em pseudo-python

Um exemplo de combinador pode ser o sequenciamento, isto é aplicar um parser após o outro. Em gramática livre do contexto a notação para uma sequência é apenas colocar um item ao lado do outro na produção. O código para representar esse combinador está no Código 5.

```
1 def seq(parser1, parser2):
2     def seq_parser(tokens):
3         (result1, tokens) = parser1(tokens)
4         (result2, tokens) = parser2(tokens)
5         return ((result1, result2), tokens)
6     return seq_parser
```

Código 5 – Exemplo de combinador em pseudo-python

Parser combinators implementam gramáticas a partir da composição de funções deste tipo com funções primitivas.

2.3.3 *Parser* LL(1)

Existe uma classe de *parsers* chamada preditiva que evitam a necessidade de *backtracking*. Ela consegue determinar a próxima produção a ser executada de maneira determinística analisando os próximos tokens da entrada.

Um exemplo concreto são os analisadores LL(1). Neste tipo de *parsing* o processo é parecido com a descida recursiva. Entretanto, no momento de escolher a produção o token atual da entrada é analisado e pode-se determinar a produção correta a ser escolhida. Isto evita a necessidade de fazer *backtracking*.

Para construir esse tipo de *parser* é necessário um algoritmo que permita escolher qual produção a ser executada de maneira determinística.

Este algoritmo utiliza uma *predictive parsing table*. Essa tabela consiste de linhas representando não terminais, colunas representando o símbolo de entrada e o valor dentro da célula diz qual produção escolher durante o *parsing*.

O método para construir essa tabela necessita antes de duas operações comumente chamadas *FIRST* e *FOLLOW*.

A definição de FIRST(A), onde A é qualquer sequência de símbolos da gramática, é o conjunto dos terminais que iniciam as sequências derivadas a partir de A. De maneira genérica, dado uma derivação A ⇒ ... ⇒ cy onde c é um terminal, logo c pertence ao conjunto FIRST(A). Por exemplo utilizando a gramática definida no Código 2, o FIRST(F) é o conjunto com dois terminais **(** e **ID**. Como F não deriva string vazia o FIRST(T) e FIRST(E) são iguais.

A definição de FOLLOW(A) é o conjunto de terminais x que podem aparecer imediatamente à direita de um não terminal A. Isto é, o conjunto de terminais x em derivações na forma S ⇒ ... ⇒ wAxy onde w e y podem ser qualquer sequência de símbolos. Por exemplo utilizando a gramática definida no Código 2, o FOLLOW(E) é composto pelo terminal **)**.

Tabela 1 – Tabela preditiva para analisador sintático LL(1)

Não Terminal	**Terminais**				
	id	+	*	(	)
E	1	-	-	1	-
E'	-	3	-	-	4
T	6	-	-	6	-
T'	-	9	8	-	9
F	12	-	-	11	-

Um exemplo de tabela preditiva pode ser visto na Tabela 1, os números referenciam as linhas do Código 2 onde estão as produções.

Neste capítulo foi discutida a fundamentação teórica necessária para entender como um computador pode interpretar uma linguagem. O próximo capítulo discute vários formatos utilizados para serialização de dados.

3 Formatos para serialização de dados

No capítulo anterior, foram discutidas maneiras para transformar um texto em uma representação computacional. Para tal, foram definidas algumas abstrações como linguagens, gramáticas e *parsers*.

Como dito na Seção 1.1 o objetivo deste trabalho é definir uma representação para troca de dados. Portanto, é essencial discutir outras notações existentes. Especialmente XML (Extensible Markup Language) e JSON (JavaScript Object Notation).

3.1 XML (*Extensible Markup Language*)

Um formato muito conhecido é o XML (Extensible Markup Language). Essa notação define os chamados documentos XML, e também define parcialmente o comportamento dos processadores de XML (BRAY et al., 2008).

A origem do XML é a linguagem de marcação SGML e o XML é também portanto uma linguagem de marcação. Isto significa que a especificação foi pensada como uma analogia ao ato de marcar um texto físico. Como por exemplo em um processo de revisão.

A abstração principal é o chamado elemento e um documento XML contém um ou mais elementos. Um elemento consiste de tags, que são textos no formato **<...> ou** </...>, tanto de início e fim ou no caso de um elemento vazio consiste de um *empty-element tag*, que tem o formato <... />. Esses elementos delimitam uma marcação em um texto e possuem um tipo definido pelo seu nome além de permitirem a presença de atributos.

```
1 <termdef id="dt-dog" term="dog">duke</termdef>
2 <foo>bar</foo>
3 <br />
```

Código 6 – Exemplo de elementos XML

Entretanto, XML possui alguns problemas para representar dados. Um deles é que não mapeia de maneira natural algumas estruturas de dados como listas. A sintaxe é carregada e existe uma ambiguidade no uso de atributos e filhos. Além disso o texto é misturado com os elementos.

3.2 JSON (*JavaScript Object Notation*)

JSON é uma notação em formato de texto, mais nova que o XML, criada para facilitar a troca de dados de um servidor com o navegador web. Foi baseada na linguagem JavaScript (ECMAScript) e formalizada pela Ecma International (2017) após a proposta inicial do Douglas Crawford encontrada em Crockford (2006). Tem o objetivo de ser fácil de ler e escrever e também ser fácil para máquinas interpretar e gerar.

Um valor JSON pode ser do tipo *object*, *array*, *number*, *string*, *true*, *false* ou *null*, mapeando naturalmente para JavaScript e várias outras linguagens dinâmicas.

```
{
    "name": "Eberth",
    "age": 45,
    "friends": [
        "Richard",
        "Matthew"
    ],
    "studying": false
    "sleeping": true
    "bank_balance": null
}
```

Código 7 – Exemplo de JSON

A especificação do JSON propositalmente é simples e foi concebida explicitamente com um subconjunto seguro implementado com o eval no JavaScript.

O objetivo é definir apenas a sintaxe e não a semântica sobre como um valor JSON pode ser convertido para uma estrutura de dado de uma linguagem de programação, ainda que muitas vezes a semântica implícita pelo eval do JavaScript seja favorecida.

Apesar de ser baseado em JavaScript as estruturas utilizadas em JSON podem ser encontradas nativamente nas principais linguagens de programação dinâmicas e implementações do *parser* existem em praticamente todas linguagens de programação.

A troca de dados entre sistemas usando JSON necessita de um acordo entre as partes envolvidas, e o JSON possui limitações como ser um subconjunto do JavaScript e não possuir nenhum mecanismo de extensão. Além disso não permite a entrada de comentários e a regra para vírgulas é restrita. O sistema de tipos é relativamente pobre e a sintaxe não permite o uso em modo *stream* de maneira fácil. Por exemplo, dado um arquivo JSON muito grande o ideal é utilizar esse arquivo como uma *stream* de dados e não colocar ele todo em memória. Entretanto, devido a notação de objetos e listas isto se torna mais custoso em JSON.

3.3 Outros Formatos

Existem também outros formatos de texto e binários com foco na troca de dados ou representação de arquivos de configuração. O FRED também foi influenciado pelos formatos *Amazon Ion, TOML, edn e YAML*, estes trazem vários conceitos que evoluem os modelos JSON e XML.

- **Amazon Ion**: Possui representação tanto em texto como binário. O formato de texto é compátivel com JSON porém possui um modelo de dados com mais tipos, por exemplo blobs binários (arquivos binários grandes que podem ser codificados para algum formato de texto), data e hora (AMAZON. . . , 2016).
- **TOML**: O formato TOML tem um foco em arquivos de configuração e em ser simples, entretanto o modelo de dados também é mais diverso que o do JSON com tipos para datas e binários por exemplo. TOML é bastante influenciado pelo formato INI, que é amplamente utilizado, apesar de não possuir uma especificação formal (TOML. . . , 2013).
- **edn**: Este formato possui um modelo de dados rico e suporta um mecanismo de extensibilidade para indicar mais semântica aos dados, esta abstração possui alguma semelhança com o XML. Também suporta *streaming* facilmente diferente de JSON. Visto que vem da comunidade Clojure que é bastante focada em dados (EDN. . . , 2012).
- **YAML**: É um formato com um modelo de dados baseado em linguagens dinâmicas como Perl, Ruby, Python. Tem o foco em ser fácil de ler por humanos e também possui suporte para *streaming* (BEN-KIKI; EVANS; INGERSON, 2009).

4 FRED

Este capítulo discute como o projeto foi desenvolvido, especificamente para cumprir com os objetivos definidos na Seção 1.1. Para esse fim foi definido um escopo para o projeto, que consiste na especificação da linguagem FRED e como foram feitas as implementações e a coleta de resultados.

4.1 Especificação do FRED

A solução construída consistiu em implementar a linguagem FRED (Flexible REpresentation of Data). Portanto é necessário antes definir esta linguagem, tanto do ponto de vista de sintaxe como de modelo de dados.

O FRED é uma notação criada para a troca de dados. Os objetivos da linguagem são:

- Facilidade para ler e escrever por humanos.
- Facilidade para construir *parsers*.
- Suportar extensão do modelo de dados atráves de tags.
- Permitir a entrada de metadados.
- Sintaxe familiar, mas sem exigir compatibilidade com JSON.

O FRED possui um modelo de dados baseados em tipos frequentemente encontrados em linguagens dinâmicas, semelhante ao JSON. Porém, com algumas adições e abstrações inspiradas em XML e outras linguagens.

Existem também características que permitem escrever FRED com facilidade. As vírgulas em FRED são consideradas espaço em branco e isso permite por exemplo escrever objetos e arrays com maior facilidade. Também permite dentro dos objetos escrever as chaves sem aspas.

A linguagem FRED foi construída pensando tanto na facilidade para ler e escrever, como na simplicidade para construir *parsers*. Também possui abstrações que permitem incluir mais significado semântico aos dados e, no futuro, incluir sistemas de validação e conversão automática de tipo.

O FRED foi desenvolvido pensando em alguns casos de uso. Inicialmente a linguagem tem o uso principal para troca de dados, comunicação entre sistemas e outras formas

de representação de dados que devem ser legíveis para seres humanos como arquivos de configuração.

Outro uso interessante é a facilidade de representar árvores sintáticas em FRED. Pois, devido as tags *union types* são facilmente representadas e portanto construir uma árvore sintática fica simples como no exemplo **Mul [$x, Add [40, 2]]**.

Essa representação possui vantagens em relação as *S-expressions*, que é uma notação utilizada para representar listas aninhadas e árvores, popular em linguagens da família LISP. Visto que, o papel do nó da árvore é destacado e possui um local natural para a inclusão de meta-informação.

4.1.1 Comentários e Espaço em branco

Comentários são permitidos em FRED e utilizam o caractere **;**. É ignorada toda a linha depois do caractere até a quebra de linha. Em FRED a regra de espaço em branco é parecida com JSON exceto que o caractere vírgula também é considerado espaço em branco.

4.1.2 *Null* e Booleanos

Os valores atômicos são iguais ao JSON e possuem a mesma semântica. Estão representados na gramática definida no Código 8, onde algumas produções foram omitidas para facilitar a leitura.

```
1 atom : "null"
2      | "true"
3      | "false
```

Código 8 – Gramática para bools e null

4.1.3 *Strings*

Strings são representadas com aspas duplas como em **"Valid string"** e aceitam qualquer caracter unicode exceto os que devem ser escapados, que são:

\b, \t, \n, \f, \r, \", \\, \xXX, \uXXXX, \UXXXXXXXX

Diferentemente do JSON, existe uma notação para adicionar *code points* unicode com dois ou oito hexadecimais. A gramática está especificada no Código 9 e utiliza expressões regulares.

```
1 string : STRING_LITERAL
2 STRING_LITERAL : /"(?:[^\\"]|\\(?:[bfnrtv"\\/]
```

```
3                     |x[0-9a-fA-F]{2}
4                     |u[0-9a-fA-F]{4}
5                     |U[0-9a-fA-F]{8}))*"/
```

Código 9 – Gramática para strings

4.1.4 Números

Os números podem ser tanto decimais como inteiros e existe notação especial para entrar números em diferentes bases (binário, octal, hexadecimal). Exemplo de inteiros são **42, -42, 1_000_000, 0xBEEF_00E9, 0o7823, 0b1010** e exemplos de números do tipo float são **42.12, -42.12, 4.32e-19, -2E-2**.

A gramática que especifica estes tipos está no Código 10. As definições das expressões regulares foram omitidas.

```
1 number : NUMBER_LITERAL
2     | HEX_LITERAL
3     | OCT_LITERAL
4     | BIN_LITERAL
```

Código 10 – Gramática para números

4.1.5 Data e Hora

Um tipo que não existe em JSON mas que está definido na especificação do FRED é o *DateTime*. Isto é, Datas e Horas podem ser representadas em FRED. Utilizando uma notação parecida com a RFC 3339.

Datas não associadas a horas e nem a fusos, a representação é **YYYY-MM-DD**. Para representar Tempo sem estar associado a datas nem a fusos a representação seria **HH:MM:SS.SS**.

Datas com Horas podem ser representadas com seguinte notação **YYYY-MM-DDTHH:MM:SS.SSZ** ou **YYYY-MM-DDTHH:MM:SS.SS+-HH:MM**, onde o fuso é opcional e **T** pode ser substituido por __.

A gramática para estes tipos está exposta no Código 11.

```
1 atom : date_time
2      | TIME_FORMAT
3
4 date_time : date
5           | TIME_FORMAT
6
7 date : DATE_FORMAT [ ("_" | "T") TIME_FORMAT [TIME_OFFSET]]
```

Código 11 – Gramática para datas e horas

```
atom : symbol

symbol : "$" name
```

Código 13 – Gramática para symbols

4.1.6 *Blobs* Binários

Outro tipo de dado que não tem especificação em JSON porém tem em FRED são os blobs binários. Eles representam dados binários codificados de alguma forma. A notação é similar a uma string precedida por um , como no Código 12.

```
atom : blob

blob : "#" BLOB_LITERAL
```

Código 12 – Gramática para blobs

4.1.7 *Symbols*

Symbols representam semanticamente constantes ou variáveis e estão presentes nos modelos de dados das linguagens do tipo Lisp e em Ruby. Eles podem ser representados em FRED usando o caracter **$**. Por Exemplo, **$var1**.

A gramática que especifica esse tipo está no Código 13.

4.1.8 Listas e Arrays

A linguagem FRED também permite a representação de listas e/ou arrays, utilizando uma notação parecida com JSON. Por exemplo, [**1 2 3**]. Vale notar que o separador é pelo menos um espaço e que vírgulas em FRED são consideradas espaço em branco. Logo, listas também podem ser escritas desta forma [**1, 2, 3**] ou até mesmo de maneira misturada como em [**1, 2 3**].

A gramática que define esse tipo está no Código 14.

```
atom : array

array : "[" value* "]"
```

Código 14 – Gramática para listas

4.1.9 Objetos

Também é possível representar objetos em FRED. A sintaxe é muito parecida com JSON. Exemplo de dicionário em FRED pode ser visto no Código 15. Basicamente é um conjunto de pares chave e valor separados por pelo menos um espaço.

```
{
    foo : "bar"
    `test foo` : "bar"
}
```

Código 15 – Exemplo de dicionário em FRED chaves com espaços são escapados com backticks

A gramática que define o tipo objeto está no Código 16.

```
atom : object

object : "{" pair* "}"

pair : name ":" value

name : VARIABLE
     | QUOTED_VARIABLE
```

Código 16 – Gramática para objetos

4.1.10 Tags e Metadados

Existem duas abstrações importantes em FRED que não existem em JSON. São as *tags* e a possibilidade de entrada de metadados.

```
person {
    name: "eric"
    age: 25
}
```

Código 17 – Exemplo de dicionário com Tag em FRED

As *tags* são uma maneira de indicar significados específicos para os valores em FRED. Isto permite mecanismos de extensão do modelo de dados base. No Código 17 temos um objeto com tag em FRED.

Valores com tag em FRED podem ser associados a metadados. A sintaxe consiste em entrar atributos dentro de parentêses. Um exemplo de um valor FRED com tag e metadados é **phone (country="Brazil") "32131123"**.

Também é possível representar tags e metadados sem estar associado a nenhum elemento. Por exemplo: **(tag attr=1)**. Isto é semanticamente igual a uma tag associada ao valor null e corresponde às *empty tags* do XML.

A gramática que define a sintaxe das Tags está no Código 18.

```
value : tagged
```

```
---
person "Jhon"
---
person "Mary"
---
person "James"
---
```

Código 19 – Exemplo de documento com streaming em FRED

```
      | atom

tagged : name [attrs] atom
       | "(" name attr* ")"

attrs : "(" attr* ")"

attr : name "=" atom
```

Código 18 – Gramática para tags e metadados

4.1.11 Suporte para *Streaming* e *Append Only*

FRED possui suporte para *streaming*, isto é funciona em arquivos grandes que não são lidos inteiramente em memória mas sim como um fluxo de dados. No Código 19 está um exemplo de documento FRED para *streaming*. Isto permite utilizar documentos FRED em modo *append only*, o que é muito útil para *logs* e em várias aplicações onde os dados são tratados como imutáveis.

A gramática para suportar *streaming* está no Código 20.

```
document : stream
         | value

stream : "---" (value  "---")*
```

Código 20 – Gramática para streaming

4.1.12 Gramática completa e especificação

A gramática do FRED foi especificada formalmente no Apêndice A. Ela foi desenvolvida de forma a ser fácil de criar *parsers* usando técnicas simples como o LL(1) e descida recursiva. Entretando, isto não significa que as implementações devam utilizar estas técnicas.

No Apêndice B a gramática especificada pode ser visualizada por meio de diagramas de sintaxe.

Toda essa especificação está divulgada de maneira aberta no endereço <https://github.com/fred-format/fred>.

4.2 Atividades do Projeto de Implementação

Com a especificação formal da linguagem FRED, é necessário descrever o escopo do projeto de implementação. As implementações foram feitas em duas linguagens: Haskell e JavaScript. Também foi discutido a linguagem FRED e como ela se relaciona com outros formatos.

As atividades que foram feitas nesse projeto são:

- Implementação do FRED em Haskell
  - Estudo de arquitetura e tecnologias.
  - Implementar a linguagem em Haskell.
  - Documentar a implementação.
  - Distribuir a implementação no Hackage.
- Implementação do FRED em JavaScript
  - Estudo de arquitetura e tecnologias.
  - Implementar a linguagem em JavaScript.
  - Documentar a implementação.
  - Distribuir a implementação no NPM.

A implementação da linguagem foi desenvolvida acompanhada de uma suíte de testes para garantir que as implementações de FRED estão conforme a especificação. As atividades que foram feitas nessa etapa são:

- Estudar como construir testes agnósticos à linguagens de programação.
- Construir uma suíte de testes capaz de testar implementações de FRED.
- Documentar a suíte de testes.

Por fim foi feito uma análise simples da linguagem FRED baseada em algumas métricas de memória. Essa análise consiste de uma comparação com linguagens semelhantes a partir de métricas definidas.

As atividades desenvolvidas nessa etapa desenvolvidas foram coletar dados e comparar FRED com linguagens semelhantes.

5 Resultados

Este capítulo aborda os resultados que foram alcançados no projeto. Assim como os insumos desenvolvidos

Na Seção 4.1 foi especificado a linguagem de maneira formal. No decorrer deste capítulo serão explorados os outros objetivos do projeto.

5.1 Implementação do FRED em Haskell

Haskell é uma linguagem puramente funcional que possui uma presença forte na área de compiladores. Boa parte das biblioteca de *parsing* em Haskell utiliza a técnica de *Parsers Combinators*. A implementação utilizou a biblioteca *parsec*[1] devido à maturidade e boa documentação (LEIJEN; MEIJER, 2001).

A partir da gramática formalizada para a linguagem FRED e utilizando os conceitos de *parsers combinators* foram criadas funções para fazer o processo de *parsing*. Estas funções possuem relação direta com os termos da gramática.

```
1 value :: Parser FredValue
2 value = tagged
3         <|> (NonTag <$> atom)
```

Código 21 – Parser para o não terminal value

Por exemplo o não terminal **value** que foi especificado na Seção 4.1. Foi mapeado na função descrita no Código 21.

O tipo dessa função mostra que ela retorna um **FredValue**. Este tipo representa um valor FRED em Haskell utilizando um *Union Type* definido no módulo **Value.hs** mostrado no Código 22.

```
1 data FredValue =
2     Tag (String, [(String, FredAtom)], FredValue)
3     | NonTag FredAtom
```

Código 22 – Definição do tipo FredValue

Um *Union Type* é um tipo que representa um dado que pode assumir diferentes representações na mesma memória, por exemplo um dado que pode ser inteiro ou *float*.

Um **FredValue** sempre é associado à um **FredAtom**. Este define os tipos que a linguagem FRED possui e está definido no módulo **Value.hs** mostrado no Código 23

[1] https://hackage.haskell.org/package/parsec

```haskell
data FredAtom =
    B Bool
    | S String
    | A [FredAtom]
    | O [(String, FredValue)]
    | N (Either Integer Float)
    | Symbol String
    | Blob B.ByteString
    | LDate Day
    | LTime TimeOfDay
    | LDateTime LocalTime
    | DateTime ZonedTime
    | NULL
```

Código 23 – Definição do tipo FredAtom

Com esses tipos, a implementação continua com outros *parsers* criados a partir da gramática especificada.

Vale notar que foram feitas modularizações para facilitar a manutenabilidade do projeto. O projeto está dividido em 5 arquivos principais.

- **Fred.hs**: Define o módulo principal com as principais funções e expõe a função parse que inicia o processo de parsing.
- **Value.hs**: Define os tipos que representam um documento FRED em Haskell.
- **Number.hs**: Possui as funções auxiliares para construir os parsers para representações numéricas em FRED.
- **String.hs**: Possui as funções auxiliares para construir os parsers para representações de texto em FRED.
- **DateTime.hs**: Possui as funções auxiliares para construir os parsers para representações Data e Hora em FRED.

A implementação do FRED em Haskell está atualmente disponível como *package candidate* no *Hackage*[2] e pode ser encontrada em <https://hackage.haskell.org/package/fred-haskell-0.1.1.1/candidate>. O pacte foi documentado utilizando *Haddock*[3].

O processo para distribuir a biblioteca começou com um pedido para criar uma conta no Hackage e após isso uma confirmação, feita por um humano, do interesse de publicar e manter a biblioteca. O procedimento recomendado foi publicar a biblioteca

[2] Gerenciador de pacotes da comunidade Haskell
[3] Ferramenta para gerar documentação de códigos Haskell

primeiro no repositório para pacotes candidatos e o avaliador também passou as melhores práticas utilizadas pela comunidade para publicação de pacotes, por exemplo, a forma de versionamento utilizado pela comunidade.

5.2 Implementação do FRED em JavaScript

A implementação da linguagem FRED foi feita em JavaScript utilizando a biblioteca Chevrotain[4]. Essa biblioteca foi escolhida devido a dois fatores, performance e documentação (CHEVROTAIN. . . , 2015).

A técnica implementada é do tipo LL(k) e a biblioteca notadamente escolhe separar a análise léxica, sintática e semântica. Isso acabou refletindo na arquitetura do *parser*.

O módulo *lexer* é baseado em expressões regulares e seu objetivo é definir e capturar os tokens a partir da entrada. Consiste de várias expressões regulares que representam os terminais da gramática. Por exemplo a representação do tipo data em FRED pode ser capturada como um token utilizando o código em 24

```
1 const DateFormat = createToken({
2     name: "DateFormat",
3     pattern: /\d{4}-\d{2}-\d{2}/
4 })
```

Código 24 – Definição do token para data

O outro módulo essencial tem o papel de analisar a sintática. A biblioteca utiliza uma *Domain Specific Language (DSL)* para especificar a gramática e funciona analisando um vetor de token gerado pelo módulo de análise léxica. As regras seguem a gramática especificada em A. Como exemplo temos o não terminal **value** no código 25

```
1 $.RULE("value", () => {
2     $.OR([
3         { ALT: () => $.SUBRULE($.tagged) },
4         { ALT: () => $.SUBRULE($.atom) }
5     ])
6 })
```

Código 25 – Definição da regra value em JavaScript

Com a definição das regras sintáticas, é necessário definir a semântica, isto é como transformar essa entrada em uma estrutura de dados em JavaScript.

A biblioteca Chevrotain recomenda a separação entre a análise sintática e a semântica. Para tal, a partir da análise sintática a biblioteca retorna uma árvore sintática

[4] <https://github.com/SAP/chevrotain>

concreta, e permite, a partir dessa árvore, a construção de uma estrutura de dados nova que irá representar o significado semântico.

É necessário percorrer essa árvore e executar ações nos nós para construir essa estrutura. Para esse processo a biblioteca utiliza o padrão *Visitor* onde existe um método referente a cada regra e é responsável por gerar a estrutura final.

Na implementação essa classe está no arquivo **visitor.js** e um exemplo de método está no código 26.

```
class FREDToAstVisitor extends BaseCstVisitor {
    tagged(ctx) {
        if (ctx.tag) {
            return this.visit(ctx.tag)
        }
        else {
            return this.visit(ctx.voidTag);
        }
    }
}
```

Código 26 – Exemplo de ação semântica

Por fim, o arquivo **index.js** implementa uma função que inicia o processo de *parsing* utilizando os módulos anteriores. A ávore de arquivos do projeto está em 3.

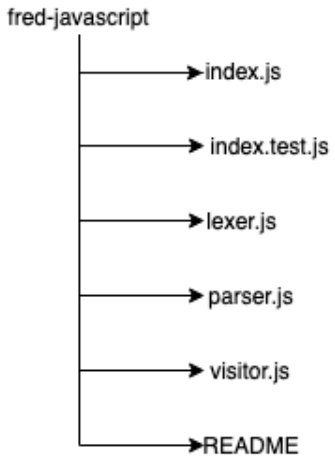

Figura 3 – Árvore de arquivos do projeto fred-javascript.

Essa implementação está publicada no npm[5] e pode ser encontrada em <https://www.npmjs.com/package/fred-js>. Também possui uma documentação simples no seu README.

[5] Node Package Manager

5.3 Suíte de Testes do FRED

Dado a complexidade tanto de especificar como de implementar uma linguagem, nem sempre as implementações estão de acordo com a especificação. Para garantir que ocorra isto foi criado uma suíte de testes da linguagem FRED, inspirada nas suítes de teste para a linguagem TOML[6] e Sass[7].

Essa suíte determina dois grupos de testes. Os que testam documentos FRED inválidos e os que testam documentos FRED válidos.

- Os testes inválidos consistem em arquivos FRED com valores inválidos para a linguagem e as implementações devem retornar erro quando tentar analisar esse documento.
- Os testes válidos consistem em arquivos de entrada com FRED válido e arquivos de sáida do tipo JSON, esses arquivos JSON estão num formato específico que codificam FRED a partir de uma transformação que não produz perda de informação.

Os arquivos de saída do tipo JSON precisam representar o documento de entrada FRED de maneira clara e específica pois as implementações irão fazer o *parsing* desta entrada e transformar num JSON de acordo com esse formato. Portanto, especificar esse formato é extremamente importante. Um valor FRED representado em JSON segue o seguinte padrão.

- Valores FRED com tag são representados como um objeto JSON com três atributos, tag, meta, value.
- Valores FRED sem tag são representados da mesma maneira como em JSON se for possível, exceto objetos.
- Valores FRED sem tag que não existem em JSON são representados como um objeto com dois campos type e value. Value normalmente consiste em uma serialização do objeto como string.
- Objetos FRED são representados como um objeto com campo type igual a *object* e no campo value um objeto seguindo as regras anteriores.

As tags tem seu valor representado como string e metadados são representados como um objeto onde a chave representa o nome do atributo FRED e o valor representa o valor do metadado FRED.

[6] <https://github.com/BurntSushi/toml-test>
[7] <https://github.com/sass/sass-spec>

```
Person (source=facebook) {
    name : "Richard"
    birth-date : 1997-12-08T12:32:45Z
    age : 21
    accomplishments : [
        "high school"
        "Chess champion"
    ]
}
```

Código 27 – Exemplo de FRED

Exemplo de documento FRED e respectiva representação em JSON, nos códigos 27 e 28.

```
{
    "tag" : "Person",
    "meta" : {
        "source" : "facebook"
    },
    "value" : {
        type : "object",
        value : {
            "name" : "Richard",
            "birth-date" : {
                "type" : "date",
                "value" : "1997-12-08T12:32:45Z"
            },
            "age" : "21",
            "accomplishments" : [
                "high school",
                "Chess champion"
            ]
        }
    }
}
```

Código 28 – Exemplo de JSON no formato de testes

Definida essa especificação, durante o trabalho foi criado um repositório onde foram construídos vários casos de testes, sendo 25 testes para FRED válidos e 25 para FRED inválidos.

O fluxo base para testar as implementações com esta suíte de testes é importar repositório de testes, ler os testes dos arquivos, criar casos de teste dinamicamente e executá-los.

Este fluxo foi utilizado tanto em Haskell como em JavaScript. Respectivamente

com a biblioteca hspec[8] e jest[9]. Ambas possuem uma API muito parecida e criar os testes dinamicamente segue um algoritmo parecido.

A suíte de testes está descrita no repositório <https://github.com/fred-format/fred-test>

5.4 Análise de uso de memória

Essa seção discute o uso de memória da linguagem. E também como ela se relaciona com outros formatos, especialmente JSON e XML. É difícil avaliar aspectos subjetivos como legibilidade e precisão. Por isso vamos focar em uma métrica fácil de analisar que é o tamanho dos documentos, além disso essa medida é fundamental em sistemas que se comunicam por uma rede distribuída para diminuir a latência de comunicação.

5.4.1 Comparação com JSON e XML

FRED e JSON se assemelham porém o FRED possui uma notação especial relacionada as tags e metadados. O XML por construção, entretanto tem uma sintaxe totalmente diferente baseada no SGML, que é uma linguagem de marcação e não de representação de dados.

Os cenários utilizados para a coleta das métricas, foram desenvolvidos baseados em três casos de uso que visam abranger de maneira imparcial as notações comparadas.

O primeiro cenário atinge a representação de dados em comunicação entre sistemas, o outro cenário abrange representação de HTML e documentos de marcação e por fim o último cenário foi focado em representar arquivos de configuração que provavelmente são escritos e lidos por humanos.

Os documentos utilizados para comparação e coleta de métricas estão no Apêndice C. Foram retiradas métricas relacionadas a quantidade de caracteres e bytes. Para tal, utilizou-se a seguinte metodologia:

- As métricas foram calculadas a partir de três exemplos representados de maneira mais similar possível em cada formato.
- Para calcular a quantidade de caracteres e bytes utilizou-se o comando *wc*.
- As métricas foram calculadas tanto nos documentos puros como nos minificados e comprimidos, para minificar foi utilizado minificadores simples que trabalham apenas com identação e espaços em branco. A forma de compressão foi Zip e Brotli.

[8] <https://github.com/hspec/hspec>
[9] <https://github.com/facebook/jest>

Retirar as métricas em arquivos comprimidos é importante, pois o tráfego de dados feito com o protocolo HTTP é normalmente comprimido.

Para o primeiro cenário foi analisado as linguagens XML, JSON e FRED quando representando dados utilizados em comunicação entre sistemas. Foi criado um cenário onde é representado uma pessoa em uma rede social. Como no Código 29.

```
Person (source=facebook) {
    name : "Richard"
    birth-date : 1997-12-08T12:32:45Z
    age : 21
    accomplishments : [
        "high school"
        "Chess champion"
    ]
}
```

Código 29 – Exemplo de dado sobre uma pessoa

As métricas foram coletadas sobre os documentos 33, 34 e 35 que estão no Apêndice C. Os resultados estão na Tabela 2.

Tabela 2 – Tabela com as métricas de tamanho em bytes

Tipo do Arquivo	**Tamanho do Arquivo (bytes)**		
	FRED	**JSON**	**XML**
Puro	1331	2077 (156%)	1732 (130%)
Minificado	703 (53%)	801 (113%)	1127 (160%)
Minificado Zip	463 (34%)	473 (102%)	490 (106%)
Minificado Brotli	243 (18%)	253 (104%)	265 (109%)

O documento puro significa o texto identado e com todos os espaços em branco. O minificado passou por um processo de retirar caracteres desnecessários visando diminuir o tamanho do arquivo. Por fim a compressão foi feita usando duas técnicas Zip e Brotli. As porcentagens na primeira linha significam o tamnho relativo ao FRED e assim subsequentemente nas outras linhas. Já a porcentagem na primeira coluna é referente ao FRED puro.

Para o segundo cenário, está sendo representado um arquivo HTML. Com isto, podemos avaliar como as linguagens comportam-se em relação a marcação de texto. Para tal, foi utilizado no FRED uma notação parecida com o exemplo exposto no Código 30.

As métricas foram coletadas nos documentos 36, 37, 38 que estão no Apêndice C e os resultados estão na Tabela 3.

Vale ressaltar que a representação em JSON utiliza a forma como a biblioteca React representa a DOM pois é o mais próximo de um consenso pragmático de como

```
div (class="card") [
    h1 "Card h1"
    h2 "Card h2"
    ul [
        li "Element 1"
        li "Element 2"
    ]
]
```

Código 30 – Exemplo de dado representando HTML

Tabela 3 – Tabela com as métricas de tamanho em bytes

Tipo do Arquivo	**Tamanho do Arquivo (bytes)**		
	FRED	**JSON**	**XML**
Puro	789	6503 (824%)	919 (116%)
Minificado	497 (63%)	1575 (317%)	628 (126%)
Minificado Zip	290 (37%)	359 (124%)	314 (108%)
Minificado Brotli	127 (16%)	178 (140%)	139 (109%)

representar HTML em JSON.

Por fim, foi desenvolvido um cenário que abrange documentos provavelmente lidos e escritos por humanos, no caso arquivos de configuração. Para o FRED foi utilizado a notação descrita no Código 31.

```
{
    name : "node-js-sample"
    version : "0.2.0"
    main : "index.js"
    scripts : {
        start : "node index.js"
        test : "node index.test.js"
    }
}
```

Código 31 – Exemplo de documento para arquivo de configuração

As métricas foram coletadas sobre os documentos 39, 40 e 41 que estão no Apêndice C. Os resultados estão na Tabela 4.

É necessário utilizar documentos maiores para avaliar as formas de compressão entre Zip e Brotli. Visto que o *header* do Zip parece ser bem maior que o de Brotli.

Neste Capítulo são mostrados os insumos que foram feitos para alcançar os objetivos do Projeto. Portanto, as implementações e a análise da linguagem foram realizadas com sucesso.

Mais especficamente, a linguagem FRED foi mais compacta em todos os casos e

Tabela 4 – Tabela com as métricas de tamanho em bytes

Tipo do Arquivo	**Tamanho do Arquivo (bytes)**		
	FRED	**JSON**	**XML**
Puro	611	646 (106%)	755 (124%)
Minificado	423 (69%)	457 (108%)	611 (144%)
Minificado Zip	458 (75%)	466 (102%)	501 (109%)
Minificado Brotli	256 (42%)	255 (99%)	278 (109%)

talvez seja mais legível, entretanto como exposto anteriormente é difícil retirar métricas sobre características subjetivas de linguagens.

6 Conclusão

O projeto consistiu em especificar e implementar uma linguagem para troca de dados, chamada FRED. A especificação foi construída com o objetivo de facilitar a implementação e portanto utiliza recursos simples.

A primeira implementação foi feita em Haskell e utiliza a técnica de *parsers combinators*. Neste analisador, a separação entre léxico, sintático e semântico não ficou clara. Algo que é comum neste tipo de técnica. Entretanto, o código foi modularizado por meio das funções e módulos que estão associados a partes da linguagem.

O segundo analisador foi escrito em JavaScript. Esta implementação utilizou uma biblioteca que constrói parsers LL(k). A arquitetura foi dividida de maneira clara entre o analisador léxico, sintático e semântico, pois a biblioteca também separa essas responsabilidades.

Com as implementações desenvolvidas foi necessário uma forma de verificar se estas condizem com a especificação. Para tanto, uma suíte de testes agnóstica a linguagem de implementação foi desenvolvido. Ela consiste de dois conjuntos de testes que verificam documentos FRED válidos e inválidos de acordo com a especificação. As implementações portanto utilizam essa suíte e verificam se estão de acordo com a especificação.

Durante o desenvolvimento do projeto, realizamos uma análise a respeito da linguagem FRED e suas características. Entretanto, não é posível afirmar que as características diferenciais melhoram a usabilidade ou a legibilidade. Para tal, é necessário desenvolver melhor o ecossistema e realizar estudos comparativos mais formais.

Também fizemos uma comparação simples com outros formatos correlatos, a métrica coletada foi o tamanho dos arquivos, percebe-se que os arquivos FRED são menores nos casos de uso explorados. Porém estudos mais formais e amplos são necessários visto que não é parte do escopo do projeto um estudo comparativo formal.

O formato FRED foi especificado e implementado em apenas duas linguagens de programação, portanto são necessários mais estudos sobre as vantagens e desvantagens de utilizar este format em outras linguagens. Para tal, é preciso mais realizar mais projetos utilizando FRED e portanto implementar FRED em outras linguagens é essencial.

Faltam também estudos relacionados a performance de *parsing*. Já que, não foi realizado pois não existem ainda *parsers* de FRED implementados manualmente e calibrados para performance máxima. Isto distorceria os resultados já que as alternativas para XML e JSON existem e são extremamente maduras.

6.1 Trabalhos Futuros

Como sequência a este projeto, percebe-se a necessidade de um estudo comparativo entre as linguagens correlatas, especialmente JSON e XML.

Não era escopo do projeto fazer esta comparação. Entretanto, este estudo pode fornecer informações úteis para futuras evoluções da linguagem e do seu ecossistema.

Também para melhorar o formato FRED, é necessário estudos sobre verificação e validação de modelo de dados e com isso definir uma especificação de um formato para *schemas*.

É necessário um estudo de performance de *parsing*, porém primeiro é necessário implementar JSON e XML numa biblioteca de *parsing* comum, como a utilizada nas implementações deste trabalho, ou implementar FRED de forma otimizada em uma linguagem como C.

Em relação a performance vale a pena mencionar que JSON é compatível com LL(1) a nível de caractere individual e isto permite implementações mais eficientes pois não é necessário fazer a análise léxica separadamente. Já FRED provavelmente não é devido a ambiguidades, por exemplo com o ínicio de null e uma tag que começa com n.

Implementar o formato FRED em outras linguagens de programação é extremamente necessário para possibilitar mais estudos e experimentos em diferentes casos de uso.

Uma dessas implementações a serem feitas deve utilizar as técnicas de descida recursiva e LL(1) para garantir que a gramática é de fato compatível com essas premissa.

Também é necessário estabilizar a sintaxe, pois a partir da opinião da comunidade e de trabalhos futuros, possíveis melhorias e mudanças serão sugeridas.

Referências

AHO, A. V. et al. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN 0321486811. Citado 7 vezes nas páginas 9, 24, 26, 27, 28, 29 e 30.

AMAZON Ion. 2016. Acessado em 2019-07-13. Disponível em: <https://amzn.github.io/ion-docs/>. Citado na página 35.

BEN-KIKI, O.; EVANS, C.; INGERSON, B. *YAML Ain't Markup Language (YAML) (tm) Version 1.2*. 2009. Disponível em: <http://www.yaml.org/spec/1.2/spec.html>. Citado na página 35.

BRAY, T. et al. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. 2008. W3C Recommendation. Citado 2 vezes nas páginas 21 e 33.

CHEVROTAIN: Parser Building Toolkit for JavaScript. 2015. Acessado em 2019-07-06. Disponível em: <https://github.com/SAP/chevrotain>. Citado na página 47.

COHEN, D. I. *Introduction to Computer Theory*. New York, NY, USA: John Wiley & Sons, Inc., 1986. ISBN 0-471-80271-9. Citado na página 23.

CROCKFORD, D. *The application/json Media Type for JavaScript Object Notation (JSON)*. RFC Editor, 2006. RFC 4627. (Request for Comments, 4627). Disponível em: <https://rfc-editor.org/rfc/rfc4627.txt>. Citado na página 34.

Ecma International. *The JSON Data Interchange Format*. 2017. Standard ECMA-404. Citado 2 vezes nas páginas 21 e 34.

EDN: extensible data notation. 2012. Acessado em 2019-07-13. Disponível em: <https://github.com/edn-format/edn>. Citado na página 35.

HUTTON, G.; MEIJER, E. Monadic parser combinators. School of Computer Science and IT, 1996. Citado na página 30.

LARK Grammar Reference. 2011. Acessado em 2019-06-12. Disponível em: <https://lark-parser.readthedocs.io/en/latest/grammar/>. Citado na página 26.

LEIJEN, D.; MEIJER, E. Parsec: Direct style monadic parser combinators for the real world. 2001. Citado na página 45.

TOML: Tom's Obvious, Minimal Language. 2013. Acessado em 2019-07-13. Disponível em: <https://github.com/toml-lang/toml>. Citado na página 35.

Apêndices

APÊNDICE A – Gramática da linguagem FRED

```
document : stream
            | value

stream : "---" (value  "---")*

value : tagged
        | atom

tagged : name [attrs] atom
        | "(" name attr* ")"

attrs : "(" attr* ")"

attr : name "=" atom

atom : object
        | array
        | date_time
        | symbol
        | number
        | string
        | bool
        | "null"

object : "{" pair* "}"

pair : name ":" value

array : "[" value* "]"

bool : "true" | "false"

symbol : "$" name

number : NUMBER_LITERAL
        | HEX_LITERAL
        | OCT_LITERAL
        | BIN_LITERAL

date_time : date
```

```
41              | TIME_FORMAT
42
43 date : DATE_FORMAT [ ("_" | "T") TIME_FORMAT [ TIME_OFFSET ] ]
44
45 string : STRING_LITERAL
46         | blob
47
48 blob : "#" BLOB_LITERAL
49
50 name : VARIABLE
51         | QUOTED_VARIABLE
52
53 VARIABLE : /[^#\"`$:;{}\[\]=\(\)\t\r\n ,0-9]{1}[^#\"`$:;{}\[\]=\(\)\t\r\
       n ,]*/
54
55 QUOTED_VARIABLE : /`(?:[^\\`]|\\(?:[bfnrtv`\\/]|x[0-9a-fA-F]{2}|u[0-9a-
       fA-F]{4}|U[0-9a-fA-F]{8}))*`/
56
57 STRING_LITERAL : /"(?:[^\\"]|\\(?:[bfnrtv"\\/]|x[0-9a-fA-F]{2}|u[0-9a-fA
       -F]{4}|U[0-9a-fA-F]{8}))*"/
58
59 BLOB_LITERAL : /"(?:[^\\"\u\U]|\\(?:[bfnrtv"\\/]|x[0-9a-fA-F]{2}))*"/
60
61 NUMBER_LITERAL : /-?(0|[1-9]\d*)(\.\d+)?([eE][+-]?\d+)?/
62
63 HEX_LITERAL : /0x[0-9a-fA-F]{1}([0-9a-fA-F]{1}|_[0-9a-fA-F]{1})*/
64
65 OCT_LITERAL : /0o[0-8]{1}([0-8]{1}|_[0-8]{1})*/
66
67 BIN_LITERAL : /0b[01]{1}([01]{1}|_[01]{1})*/
68
69 DATE_FORMAT : /\d{4}-\d{2}-\d{2}/
70
71 TIME_FORMAT : /\d{2}:\d{2}:\d{2}(\.\d+)?/
72
73 TIME_OFFSET : /Z|[+-]\d{2}:\d{2}/
74
75 WHITESPACE : /[ \t\n\r,]+/
76
77 COMMENT : /;.*/
```

Código 32 – Gramática em notação LARK para linguagem FRED

APÊNDICE B – Diagrama da Gramática da linguagem FRED

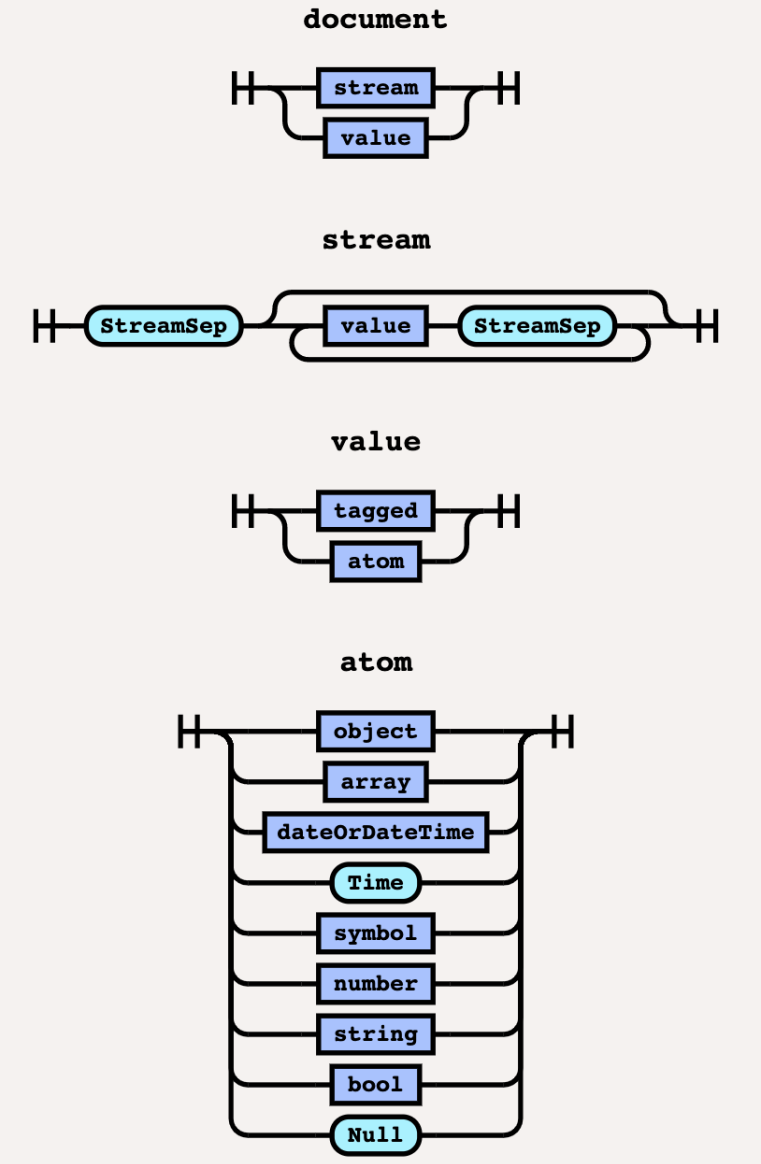

Figura 4 – Diagrama da Sintaxe do formato FRED (document até atom).

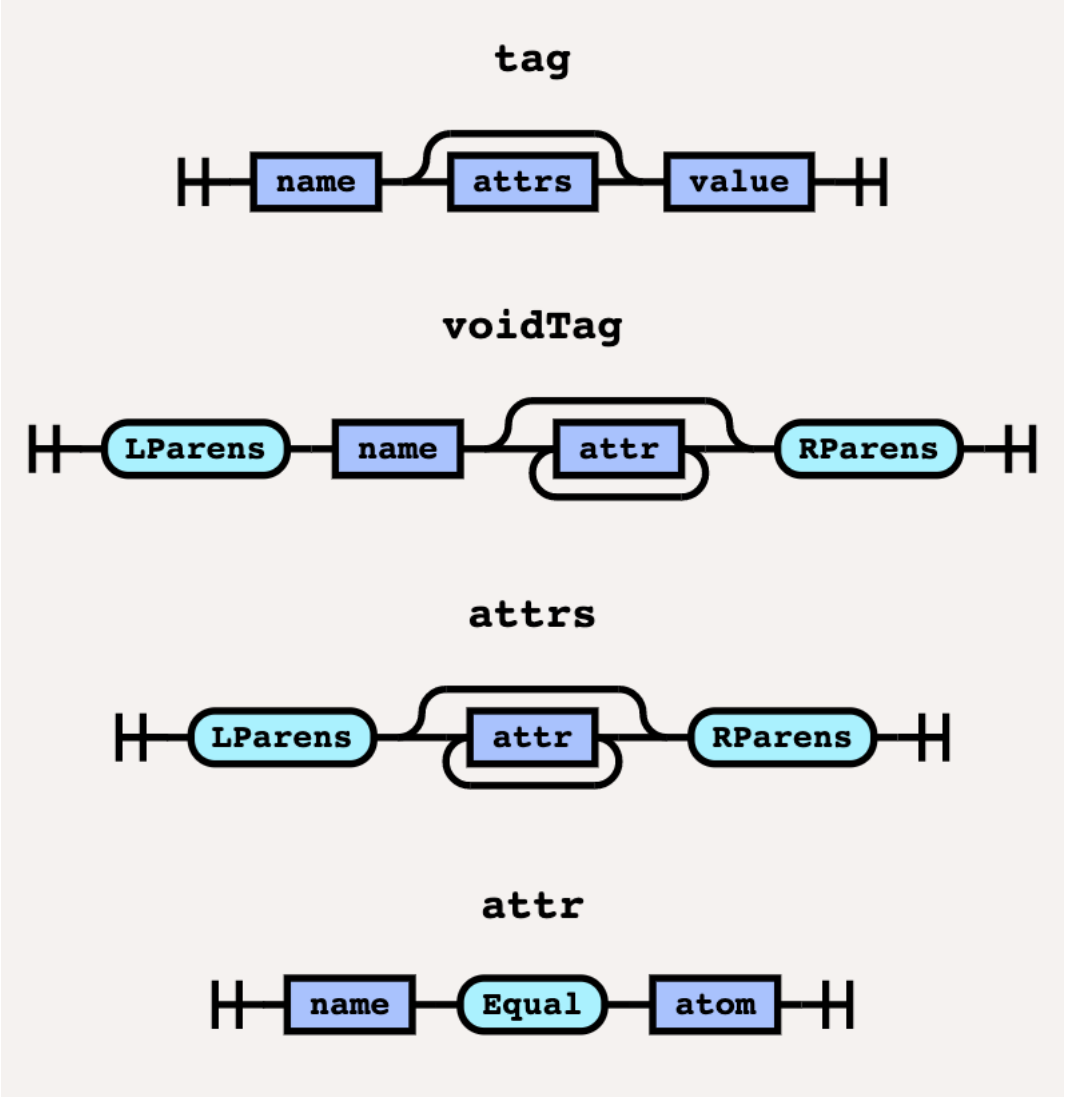

Figura 5 – Diagrama da Sintaxe do formato FRED (tag até attr).

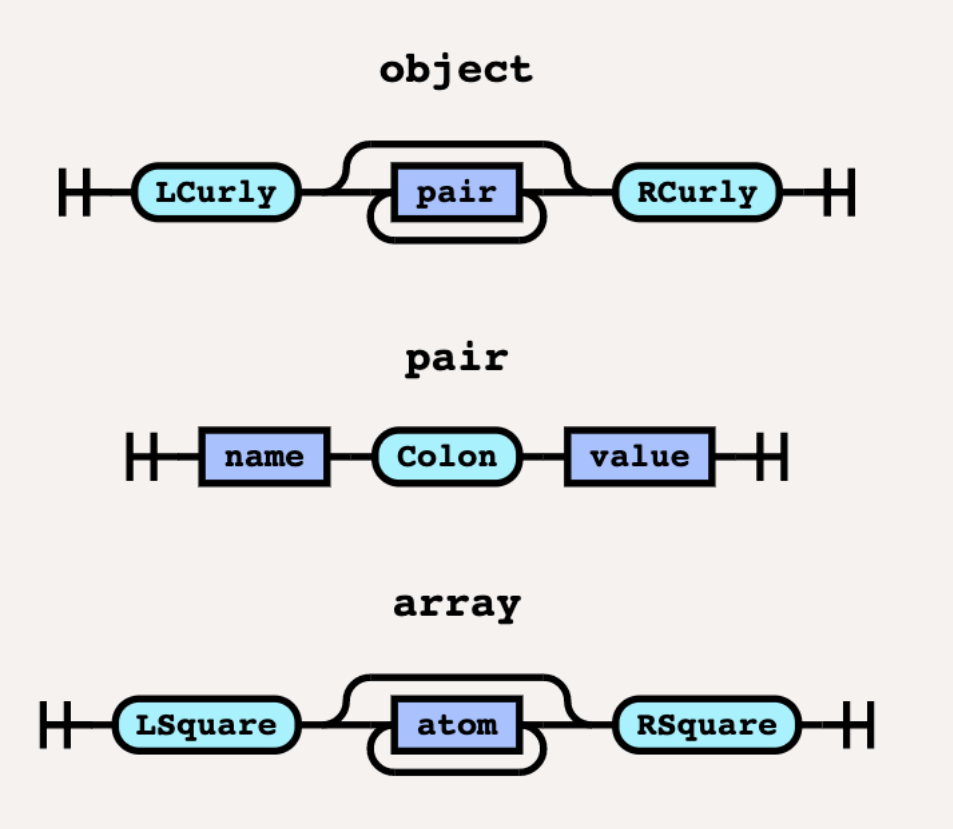

Figura 6 – Diagrama da Sintaxe do formato FRED (object até array).

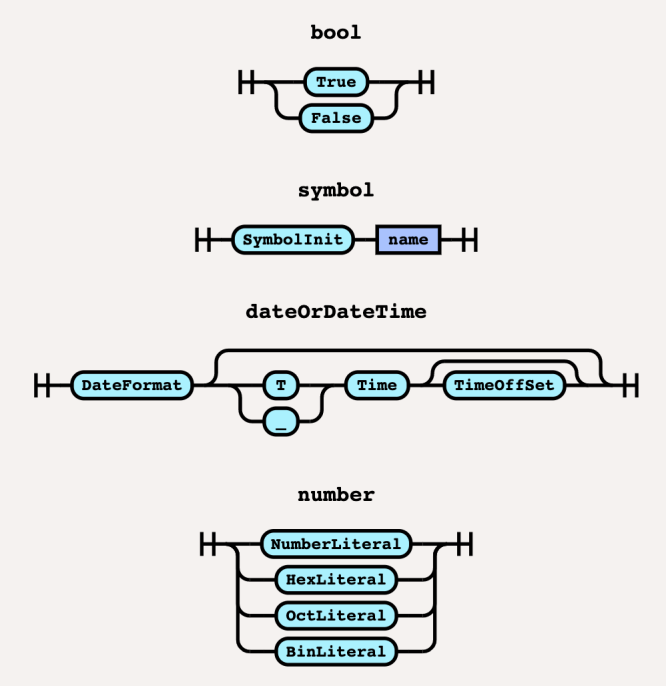

Figura 7 – Diagrama da Sintaxe do formato FRED (bool até number).

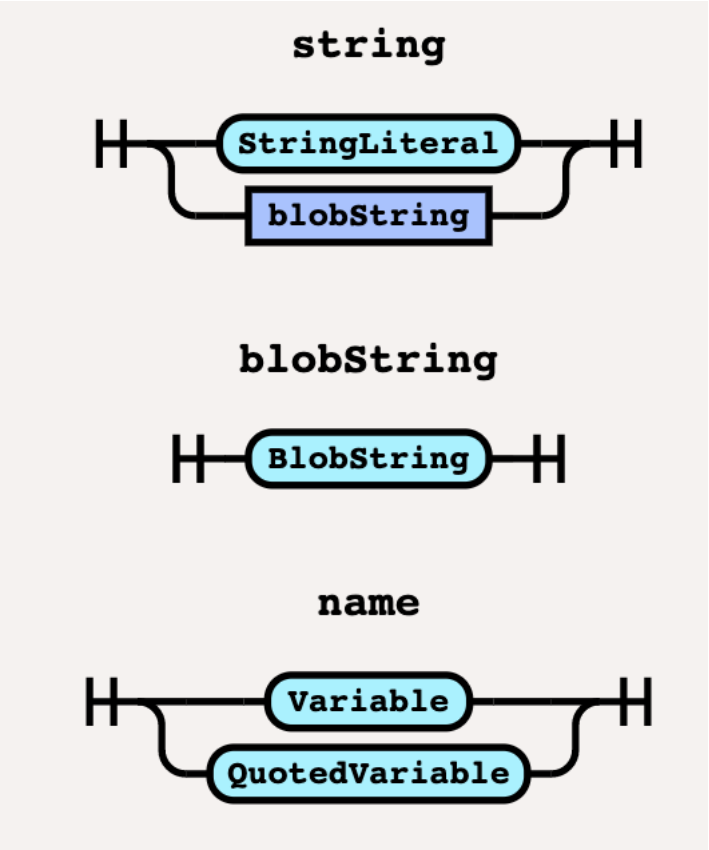

Figura 8 – Diagrama da Sintaxe do formato FRED (string até name).

APÊNDICE C – Documentos FRED, JSON e XML utilizados para comparação

```xml
<person source="facebook">
    <name>Richard</name>
    <birth-date>1997-12-08</birth-date>
    <age>21</age>
    <accomplishments>
        <value>High School</value>
        <value>Chess champion</value>
    </accomplishments>
    <friends>
        <person source="twitter">
            <name>Lucas Bernardo</name>
            <birth-date>1897-12-08</birth-date>
            <age>121</age>
            <accomplishments>
                <value>Middle School</value>
                <value>Soccer MVP</value>
            </accomplishments>
        </person>
        <person source="facebook">
            <name>Matheus</name>
            <birth-date>1987-12-08</birth-date>
            <age>31</age>
            <accomplishments>
                <value>PhD</value>
                <value>Chess champion</value>
            </accomplishments>
        </person>
        <person source="facebook">
            <name>Joao</name>
            <birth-date>1950-10-03</birth-date>
            <age>50</age>
            <accomplishments>
                <value>Best Soccer Player at School</value>
            </accomplishments>
        </person>
        <person source="orkut">
            <name>Maria</name>
            <birth-date>1998-12-08</birth-date>
            <age>20</age>
            <accomplishments>
```

```
41                 <value>College Degree</value>
42             </accomplishments>
43         </person>
44         <person source="myspace">
45             <name>John</name>
46             <birth-date>2000-12-08</birth-date>
47             <age>19</age>
48             <accomplishments>
49                 <value>High School</value>
50                 <value>Checkers champion</value>
51             </accomplishments>
52         </person>
53     </friends>
54 </person>
```

Código 33 – Exemplo de documeto XML

```
1 {
2     "Person": {
3         "source": "facebook",
4         "name": "Richard",
5         "birth-date": "1997-12-08",
6         "age": 21,
7         "accomplishments": [
8             "High School",
9             "Chess champion"
10        ],
11        "friends": [
12            {
13                "Person": {
14                    "source": "twitter",
15                    "name": "Lucas Bernardo",
16                    "birth-date": "1897-12-08",
17                    "age": 121,
18                    "accomplishments": [
19                        "Middle School",
20                        "Soccer MVP"
21                    ]
22                }
23            },
24            {
25                "Person": {
26                    "source": "facebook",
27                    "name": "Matheus",
28                    "birth-date": "1987-12-08",
29                    "age": 31,
30                    "accomplishments": [
31                        "PhD",
```

```
32                          "Chess champion"
33                      ]
34                  }
35              },
36              {
37                  "Person": {
38                      "source": "facebook",
39                      "name": "Joao",
40                      "birth-date": "1950-10-03",
41                      "age": 50,
42                      "accomplishments": [
43                          "Best Soccer Player at School"
44                      ]
45                  }
46              },
47              {
48                  "Person": {
49                      "source": "orkut",
50                      "name": "Maria",
51                      "birth-date": "1998-12-08",
52                      "age": 20,
53                      "accomplishments": [
54                          "College Degree"
55                      ]
56                  }
57              },
58              {
59                  "Person": {
60                      "source": "myspace",
61                      "name": "John",
62                      "birth-date": "2000-12-08",
63                      "age": 19,
64                      "accomplishments": [
65                          "High School",
66                          "Checkers champion"
67                      ]
68                  }
69              }
70          ]
71      }
72  }
```

Código 34 – Exemplo de documeto JSON

```
1  Person (source=facebook) {
2      name : "Richard"
3      birth-date : 1997-12-08
4      age : 21
```

```
    accomplishments : [
        "High School"
        "Chess champion"
    ]
    friends : [
        Person (source=twitter) {
            name : "Lucas Bernardo"
            birth-date : 1897-12-08
            age : 121
            accomplishments : [
                "Middle School"
                "Soccer MVP"
            ]
        }
        Person (source=facebook) {
            name : "Matheus"
            birth-date : 1987-12-08
            age : 31
            accomplishments : [
                "PhD"
                "Chess champion"
            ]
        }
        Person (source=facebook) {
            name : "Joao"
            birth-date : 1950-10-03
            age : 50
            accomplishments : [
                "Best Soccer Player at School"
            ]
        }
        Person (source=orkut) {
            name : "Maria"
            birth-date : 1998-12-08
            age : 20
            accomplishments : [
                "College Degree"
            ]
        }
        Person (source=myspace) {
            name : "John"
            birth-date : 2000-12-08
            age : 19
            accomplishments : [
                "High School"
                "Checkers champion"
            ]
```

```
        }
    ]
}
```

Código 35 – Exemplo de documeto FRED

```
<div class="card">
    <h1>Card Title</h1>
    <h2>Card Sub Title</h2>
    <ul>
        <li>Element Test</li>
        <li>Element Lorem Ipsum</li>
    </ul>
    <ul>
        <li>Lorem Element Ipsum</li>
        <li>Ipsum Element Lorem</li>
    </ul>
    <ul>
        <li>Test Element</li>
        <li>Lorem Element</li>
    </ul>
    <ul>
        <li>Ipsum Element</li>
        <li>Lorem Ipsum</li>
    </ul>
    <ul>
        <li>Lorem Test</li>
        <li>Lorem Element Test</li>
    </ul>
    <ul>
        <li>Element Test</li>
        <li>Element Lorem Ipsum</li>
    </ul>
    <ul>
        <li>Lorem Element Ipsum</li>
        <li>Ipsum Element Lorem</li>
    </ul>
    <ul>
        <li>Test Element</li>
        <li>Lorem Element</li>
    </ul>
    <ul>
        <li>Ipsum Element</li>
        <li>Lorem Ipsum</li>
    </ul>
    <ul>
        <li>Lorem Test</li>
        <li>Lorem Element Test</li>
```

```
    </ul>
</div>
```

Código 36 – Exemplo de documeto XML

```
{
    "type": "div",
    "props": {
        "class": "card",
        "children": [
            {
                "type": "h1",
                "props": {
                    "children": "Card Title"
                }
            },
            {
                "type": "h2",
                "props": {
                    "children": "Card Sub Title"
                }
            },
            {
                "type": "ul",
                "props": {
                    "children": [
                        {
                            "type": "li",
                            "props": {
                                "children": "Element Test"
                            }
                        },
                        {
                            "type": "li",
                            "props": {
                                "children": "Element Lorem Ipsum"
                            }
                        }
                    ]
                }
            },
            {
                "type": "ul",
                "props": {
                    "children": [
                        {
                            "type": "li",
                            "props": {
```

```
                                "children": "Lorem Element Ipsum"
                            }
                        },
                        {
                            "type": "li",
                            "props": {
                                "children": "Ipsum Element Lorem"
                            }
                        }
                    ]
                }
            },
            {
                "type": "ul",
                "props": {
                    "children": [
                        {
                            "type": "li",
                            "props": {
                                "children": "Test Element"
                            }
                        },
                        {
                            "type": "li",
                            "props": {
                                "children": "Lorem Element"
                            }
                        }
                    ]
                }
            },
            {
                "type": "ul",
                "props": {
                    "children": [
                        {
                            "type": "li",
                            "props": {
                                "children": "Ipsum Element"
                            }
                        },
                        {
                            "type": "li",
                            "props": {
                                "children": "Lorem Ipsum"
                            }
                        }
```

```
                    ]
                }
            },
            {
                "type": "ul",
                "props": {
                    "children": [
                        {
                            "type": "li",
                            "props": {
                                "children": "Lorem Test"
                            }
                        },
                        {
                            "type": "li",
                            "props": {
                                "children": "Lorem Element Test"
                            }
                        }
                    ]
                }
            },
            {
                "type": "ul",
                "props": {
                    "children": [
                        {
                            "type": "li",
                            "props": {
                                "children": "Element Test"
                            }
                        },
                        {
                            "type": "li",
                            "props": {
                                "children": "Element Lorem Ipsum"
                            }
                        }
                    ]
                }
            },
            {
                "type": "ul",
                "props": {
                    "children": [
                        {
                            "type": "li",
```

```
                            "props": {
                                "children": "Lorem Element Ipsum"
                            }
                        },
                        {
                            "type": "li",
                            "props": {
                                "children": "Ipsum Element Lorem"
                            }
                        }
                    ]
                }
            },
            {
                "type": "ul",
                "props": {
                    "children": [
                        {
                            "type": "li",
                            "props": {
                                "children": "Test Element"
                            }
                        },
                        {
                            "type": "li",
                            "props": {
                                "children": "Lorem Element"
                            }
                        }
                    ]
                }
            },
            {
                "type": "ul",
                "props": {
                    "children": [
                        {
                            "type": "li",
                            "props": {
                                "children": "Ipsum Element"
                            }
                        },
                        {
                            "type": "li",
                            "props": {
                                "children": "Lorem Ipsum"
                            }
```

```
                        }
                    ]
                }
            },
            {
                "type": "ul",
                "props": {
                    "children": [
                        {
                            "type": "li",
                            "props": {
                                "children": "Lorem Test"
                            }
                        },
                        {
                            "type": "li",
                            "props": {
                                "children": "Lorem Element Test"
                            }
                        }
                    ]
                }
            }
        ]
    }
}
```

Código 37 – Exemplo de documeto JSON

```
div (class="card") [
    h1 "Card Title"
    h2 "Card Sub Title"
    ul [
        li "Element Test"
        li "Element Lorem Ipsum"
    ]
    ul [
        li "Lorem Element Ipsum"
        li "Ipsum Element Lorem"
    ]
    ul [
        li "Test Element"
        li "Lorem Element"
    ]
    ul [
        li "Ipsum Element"
        li "Lorem Ipsum"
    ]
```

```
    ul [
        li "Lorem Test"
        li "Lorem Element Test"
    ]
    ul [
        li "Element Test"
        li "Element Lorem Ipsum"
    ]
    ul [
        li "Lorem Element Ipsum"
        li "Ipsum Element Lorem"
    ]
    ul [
        li "Test Element"
        li "Lorem Element"
    ]
    ul [
        li "Ipsum Element"
        li "Lorem Ipsum"
    ]
    ul [
        li "Lorem Test"
        li "Lorem Element Test"
    ]

]
```

Código 38 – Exemplo de documeto FRED

```
<package name="node-js-sample">
    <version>0.2.0</version>
    <description>A sample Node.js app using Express 4</description>
    <main>index.js</main>
    <scripts>
        <start>node index.js</start>
    </scripts>
    <dependencies>
        <express version="^4.13.3" />
    </dependencies>
    <engines>
        <node version="4.0.0" />
    </engines>
    <repository type="git" url="https://github.com/heroku/node-js-sample
    "/>
    <keywords>
        <value>node</value>
        <value>heroku</value>
        <value>express</value>
```

```
    </keywords>
    <author>Mark Pundsack</author>
    <contributors>
        <value>Zeke Sikelianos zeke@sikelianos.com (http://zeke.
    sikelianos.com)</value>
    </contributors>
    <license>MIT</license>
</package>
```

Código 39 – Exemplo de documeto XML

```
{
    "name": "node-js-sample",
    "version": "0.2.0",
    "description": "A sample Node.js app using Express 4",
    "main": "index.js",
    "scripts": {
        "start": "node index.js"
    },
    "dependencies": {
        "express": "^4.13.3"
    },
    "engines": {
        "node": "4.0.0"
    },
    "repository": {
        "type": "git",
        "url": "https://github.com/heroku/node-js-sample"
    },
    "keywords": [
        "node",
        "heroku",
        "express"
    ],
    "author": "Mark Pundsack",
    "contributors": [
        "Zeke Sikelianos <zeke@sikelianos.com> (http://zeke.sikelianos.
    com)"
    ],
    "license": "MIT"
}
```

Código 40 – Exemplo de documeto JSON

```
{
    name : "node-js-sample"
    version : "0.2.0"
    description: "A sample Node.js app using Express 4"
```

```
5     main : "index.js"
6     scripts : {
7         start: "node index.js"
8     }
9     dependencies:  {
10        express : "^4.13.3"
11    }
12    engines: {
13        node : "4.0.0"
14    }
15    repository : {
16        type : "git"
17        url : "https://github.com/heroku/node-js-sample"
18    }
19    keywords : [
20        "node"
21        "heroku"
22        "express"
23    ]
24    author : "Mark Pundsack"
25    contributors: [
26        "Zeke Sikelianos <zeke@sikelianos.com> (http://zeke.sikelianos.
      com)"
27    ]
28    license : "MIT"
29 }
```

Código 41 – Exemplo de documeto FRED