UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Henry Rodrigues Da Silva

LINGUAGEM DE PROGRAMAÇÃO VES

Florianópolis

2019

Henry Rodrigues Da Silva

LINGUAGEM DE PROGRAMAÇÃO VES

Trabalho de Conclusão de Curso submetido ao curso de Ciências da computação para a obtenção do Grau de Bacharel em ciências da computação. Orientador: Prof. Dra. Eng. Jerusa Marchi

Florianópolis

2019

Henry Rodrigues Da Silva

LINGUAGEM DE PROGRAMAÇÃO VES

Este Trabalho de Conclusão de Curso foi julgado aprovado para a obtenção do Título de "Bacharel em ciências da computação", e aprovado em sua forma final pelo curso de Ciências da computação.

Florianópolis, 01 de julho 2019.

Prof. MEng. José Francisco Danilo De Guadalupe Correa Fletes
Coordenador do Curso

Banca Examinadora:

Prof. Dra. Eng. Jerusa Marchi
Orientador

Prof. Dr. Elder Rizzon Santos

Prof. Dr. Maicon Rafael Zatelli

AGRADECIMENTOS

Agradeço primeiramente a Deus por me dar vida, a possibilidade de iniciar e concluir minha graduação e a capacidade para realizar este trabalho.

A meu pai Dilceu, minha mãe Marisa e minha irmã Anne, por seu amor incondicional, apoio e incentivo, mesmo nos piores momentos.

Aos meus amigos e colegas, que batalharam ao meu lado até o final do curso. Em especial a meus grandes amigos, Alexandre Behling e Bruno Manica, por seu companheirismo e apoio. Também a Augusto Zwirtes, meu colega de apartamento e faculdade que hoje chamo de irmão.

Aos professores que durante o curso contribuíram para meu crescimento pessoal e profissional.

A minha orientadora, Jerusa Marchi, que como professora, inspirou em mim o desejo de estudar teoria da computação e linguagens formais. E como orientadora sempre me apoiou, direcionou e incentivou, tornando este trabalho possível.

One has no right to love or hate anything if one has not acquired a thorough knowledge of its nature. Great love springs from great knowledge of the beloved object, and if you know it but little you will be able to love it only a little or not at all.

Leonardo da Vinci

RESUMO

Existem diversas linguagens de programação no mercado, cada uma com seus pontos fortes e fracos. Este trabalho apresenta Ves, uma linguagem de programação focada em expressividade, legibilidade e flexibilidade, utilizando conceitos adaptados de outras linguagens de forma tentar melhorar funcionalidades presentes nas mesmas, ou ainda, facilitar o uso de técnicas que já existem mas são difíceis de se trabalhar nas linguagens atuais. Para que a linguagem possa cumprir com esses requisitos é dado foco em meta-programação e type-safety, dando preferência a uma sintaxe mais declarativa.

Palavras-chave: Semântica de programas. Linguagem de programação.

ABSTRACT

There are many programming languages in the market, each one of them has its strengths and weaknesses. This project introduces Ves, a programming language that aims to be expressive, legible and flexible, using concepts adapted from other languages intending to improve their features, or even ease the use of techniques that already exist, but are hard to work with in other languages. In order for the language to be able to fulfill these requirements the project focuses in meta-programming and type-safety, favoring a declarative syntax.

Keywords: Program semantics. Programming language.

LISTA DE FIGURAS

Figura 1 Exemplos de comentário em Ves 39
Figura 2 Trecho da gramática que reconhece identificadores 41
Figura 3 Exemplos de literal inteiro 42
Figura 4 Exemplos de literal de ponto flutuante 43
Figura 5 Exemplos de literal de caractere 43
Figura 6 Exemplos de literal de *string* 44
Figura 7 Exemplos de expresões em Ves. 45
Figura 8 Exemplos de tupla 46
Figura 9 Exemplo de listas 46
Figura 10 Trecho da gramática que descreve declarações const 46
Figura 11 Exemplos de declaração const 47
Figura 12 Techo da gramática que descreve declarações let 47
Figura 13 Exemplo de declarações let 47
Figura 14 Trecho da gramática que descreve declarações var 47
Figura 15 Exemplos de declaração de variável 48
Figura 16 Trecho da gramática que descreve comandos *if* 48
Figura 17 Exemplo do comando *if* 48
Figura 18 Trecho da gramática que descreve o comando *while* 49
Figura 19 Exemplo do comando *while* 49
Figura 20 Trecho da gramática que descreve o comando *loop* 49
Figura 21 Exemplo do comando *loop* 49
Figura 22 Exemplo de comando *loop* nomeado 50
Figura 23 Trecho da gramática que descreve o comando *for* 50
Figura 24 Exemplos de comando *for* 50
Figura 25 Trecho da gramática que descreve a declaração de *match* 51
Figura 26 Exemplo do comando *match* 51
Figura 27 Trecho da gramática que descreve o comando *try except/finally* 52
Figura 28 Exemplo de bloco do tipo *try-finally* 52
Figura 29 Exemplo de bloco do tipo *try-except* 53
Figura 30 Exemplo do comando *raise* 53
Figura 31 Trecho da gramática que descreve o comando *with* 53

Figura 32 Exemplo de comando *with* 53
Figura 33 Trecho da gramática que descreve a declaração de funções 54
Figura 34 Exemplo de função 54
Figura 35 Exemplo de sintaxe alternativa de funções 54
Figura 36 Exemplo de funções anônimas 55
Figura 37 Trecho da gramática que descreve a declaração de um tipo 56
Figura 38 Exemplo de declaração de registros 56
Figura 39 Exemplo de uniões simples 57
Figura 40 Exemplo de união com cláusula *with* 57
Figura 41 Exemplo da declaração de atributos 58
Figura 42 Trecho da gramática que descreve a declaração de um método 58
Figura 43 Exemplo de declaração de métodos 59
Figura 44 Exemplos de chamada de método 59
Figura 45 Trecho da gramática que descreve a declaração de uma propriedade 60
Figura 46 Exemplo de propriedades 60
Figura 47 Exemplo de propriedades com *getter* e *setter* 60
Figura 48 Trecho da gramática que descreve a declaração de um *trait* 61
Figura 49 Exemplo de implementação de *trait* 61
Figura 50 Diagrama de classes do modelo semântico 62
Figura 51 Exemplo de sobrecarga de operadores 63
Figura 52 Exemplo de construtor 64
Figura 53 Exemplo de tipos inferidos 65
Figura 54 Exemplo de linguagem fortemente tipada. 66
Figura 55 Formas diferentes de declarar uma função. 66
Figura 56 Exemplo de abstração em Ves. 67
Figura 57 Exemplo de *while* com só um comando em C++. 69
Figura 58 Exemplo de *while* com bloco vazio em Ves. 70
Figura 59 Declaração de propriedade em C#. 70
Figura 60 Declaração de propriedade em Ves. 70
Figura 61 Exemplo de nomes de tipos em C#. 71
Figura 62 Exemplo de nomes de tipos em Ves. 71

Figura 63 Exemplo de sintaxe de função em C#. 72
Figura 64 Exemplo de função com correspondência de padrões em Ves. 72
Figura 65 Exemplo de utilidade de *loop*. 73
Figura 66 Exemplo de função anônima em *Python*. 73
Figura 67 Exemplo de função anônima em Ves. 73

LISTA DE TABELAS

Tabela 1 Operadores de Ves ordenados por precedência 40
Tabela 2 Palavras reservadas 41
Tabela 3 Delimitadores 44

SUMÁRIO

1 INTRODUÇÃO 25
1.1 OBJETIVOS 25
1.1.1 Objetivos gerais 25
1.1.2 Objetivos específicos 26
1.2 DELIMITAÇÃO DO TRABALHO 26
1.3 MÉTODO DE PESQUISA 26
1.3.1 Fundamentação teórica 26
1.3.2 Trabalhos correlatos 27
1.3.3 Especificação formal 28
1.3.4 Definição das características 28
1.3.5 Exemplos de uso 28
2 TRABALHOS CORRELATOS 29
2.1 ADA 29
2.2 C++ 30
2.3 C# 31
2.4 DART 31
2.5 F# 32
2.6 PASCAL 33
2.7 PYTHON 34
2.8 RUBY 35
2.9 RUST 35
2.10 COMPARAÇÃO 36
2.10.1 Simplicidade e Ortogonalidade 37
2.10.2 Multiplicidade de funcionalidades 37
2.10.3 Sobrecarga de operadores 37
2.10.4 Expressividade 38
3 ESPECIFICAÇÃO FORMAL 39
3.1 ESPECIFICAÇÃO LÉXICA 39
3.1.1 Comentários 39
3.1.2 Operadores 39
3.1.3 Identificadores 40
3.1.4 Literais 42
3.1.4.1 Literais inteiros 42
3.1.4.2 Literais de número real 42
3.1.4.3 Literais de caractere 43
3.1.4.4 Literais de string 43
3.1.5 Delimitadores 44

3.2 ESPECIFICAÇÃO SINTÁTICA 44
3.2.1 Expressões 45
3.2.2 Tuplas 45
3.2.3 Listas 46
3.2.4 Const 46
3.2.5 Let 47
3.2.6 Var 47
3.2.7 If/else 48
3.2.8 While 48
3.2.9 Loop 49
3.2.10 For 50
3.2.11 Comandos de controle de laço 50
3.2.12 Match 51
3.2.13 Try 52
3.2.13.1 Finally 52
3.2.13.2 Except 52
3.2.14 With 53
3.2.15 Funções 54
3.2.16 Funções anônimas 55
3.2.17 Tipos 55
3.2.17.1 Registros 56
3.2.17.2 Uniões 57
3.2.18 Atributos 57
3.2.19 Métodos 58
3.2.20 Propriedades 59
3.2.21 Traits 61
3.3 ESPECIFICAÇÃO SEMÂNTICA 61
3.3.1 O tipo *unit* 62
3.3.2 Operadores 63
3.3.3 Construtor 63
4 DEFINIÇÃO DAS CARACTERÍSTICAS 65
4.1 TIPAGEM 65
4.2 PARADIGMAS DE PROGRAMAÇÃO 66
4.3 NÍVEL DE ABSTRAÇÃO 66
4.3.1 Propósito 67
5 EXEMPLOS DE USO 69
5.1 LEGIBILIDADE 69
5.1.1 Estruturas de controle 69
5.1.2 Declarações 70
5.2 EXPRESSIVIDADE 70
5.2.1 Anotações de tipo 71

5.2.2 Funções 71
5.3 FLEXIBILIDADE 72
5.3.1 Laços infinitos 72
5.3.2 Funções anônimas 73
6 CONCLUSÃO 75
REFERĘNCIAS 77

1 INTRODUÇÃO

O desenvolvimento de software é uma área em constante crescimento por parte de empresas e entidades públicas que visam atender cada vez mais a demanda do mercado por softwares de propósitos específicos e com alta qualidade. O mercado brasileiro por exemplo, movimentou 39,6 bilhões de dólares em 2016 o que representou 2,1% do PIB do país e 1,9% do total de investimentos de TI do mundo (ABES, 2018).

Para se obter a qualidade desejada em seus projetos de *software*, as empresas geralmente utilizam-se de paradigmas, padrões e linguagens, que auxiliam a alcançar melhores resultados quando aplicados em domínios específicos para os quais foram criados e também testados.

Cada linguagem foca em diferentes aspectos do desenvolvimento de *software*, sempre tentando ser ideal dentro de seu nicho. Tendo em vista a grande demanda no desenvolvimento de software, bem como a necessidade de linguagens que implementem de forma inovadora novas funcionalidades necessárias, este trabalho apresenta uma nova linguagem de programação de propósito geral, focada em legibilidade, expressividade e flexibilidade.

A linguagem será chamada Ves[1]. Utilizando-se conceitos adaptados de outras linguagens, Ves busca facilitar o uso de abordagens e paradigmas de desenvolvimento de *software*, que estão presentes em outras linguagens mas geralmente são difíceis de se trabalhar.

1.1 OBJETIVOS

A partir da contextualização do problema, os objetivos, geral e específicos deste trabalho são assim definidos:

1.1.1 Objetivos gerais

O presente trabalho tem como objetivo geral apresentar uma linguagem de programação multi-paradigma e de propósito geral chamada Ves.

[1] O nome Ves vem de uma língua fictícia criada pelo autor e significa água. Foi escolhido por ser simples e para denotar a fluidez da linguagem.

1.1.2 Objetivos específicos

Para a obtenção do objetivo geral, os seguintes objetivos específicos são requeridos:

1. Analisar características presentes em linguagens atuais buscando inspiração para o desenvolvimento de Ves;
2. Especificar formalmente a linguagem Ves;
3. Implementação parcial de um interpretador para prova de conceito;
4. Definir um conjunto de características desejadas para a linguagem descrita;
5. Exemplificar usos da linguagem para desenvolvimento de software.

1.2 DELIMITAÇÃO DO TRABALHO

A linguagem de programação Ves visa oferecer uma alternativa para desenvolvimento de *software* combinando diversas características presentes em outras linguagens e oferecendo uma sintaxe concisa, simples e legível. A implementação do interpretador será meramente para prova de conceito e seu uso em ambientes de desenvolvimento no mundo real está fora do escopo deste trabalho por conta das limitações de tempo.

1.3 MÉTODO DE PESQUISA

O desenvolvimento do trabalho é realizado por meio de 4 fases relacionadas aos objetivos propostos. Para que as fases do desenvolvimento possam ser melhor compreendidas uma breve fundamentação teórica será apresentada na subseção seguinte.

1.3.1 Fundamentação teórica

O projeto de uma linguagem de programação geralmente produz três especificações, a léxica, sintática e semântica. As três especificações são apresentadas em maior detalhe no Capitulo 3.

Quando um interpretador ou compilador lê o código fonte, esse código passa pelo analisador léxico. Durante essa fase as palavras e símbolos presentes no código fonte são segmentados em unidades léxicas chamadas de *tokens*. Segundo (AHO, 2003) um *token* é um par consistindo do nome do *token* e um atributo opcional chamado de valor. O nome do *token* é um símbolo abstrato que representa o tipo da unidade léxica, por exemplo, uma palavra reservada, ou uma sequencia de caracteres denotando um identificador. Os nomes de *tokens* sãos os símbolos processados pela fase seguinte, a analise sintática.

A fase de analise léxica também descarta comentários no código e verifica se os caracteres lidos se encaixam na especificação lexicográfica da linguagem, ou seja, se não existem caracteres inválidos no código.

Uma vez concluída a fase de análise léxica, os tokens são passados para a fase de analise sintática, que tem por objetivo verificar se os *tokens* estão ordenados de forma gramaticalmente correta. Para que essa verificação seja possível é necessário primeiro definir a gramática da linguagem, essa definição será apresentada em maior detalhe no capitulo 3.

Linguagens de programação geralmente têm sua sintaxe especificada através de gramáticas classificadas por (CHOMSKY, 1956) como livres de contexto. Esse tipo de gramática é menos poderosa do que as gramáticas de linguagens naturais, ou seja, as que são usadas por pessoas para se comunicar. Entretanto são suficientes para descrever a maior parte das linguagens de programação e são simples o suficiente para que um computador possa realizar a analise em um tempo relativamente pequeno, na maior parte dos casos.

A análise semântica de uma linguagem diz respeito ao significado do código e vai muito além da estrutura das frases, e por isso formalizar tais aspectos é bastante difícil. Neste trabalho a formalização da semântica se dará a partir de exemplos e descrições em formato de texto.

1.3.2 Trabalhos correlatos

Nesta fase é realizada a análise de outras linguagens que inspiram Ves e outras que buscam, assim como Ves, ser expressivas, eficientes e facilitarem meta-programação sem perder legibilidade. Os pontos fundamentais abordados são, um breve histórico de cada linguagem seus diferentes paradigmas de programação e como algumas de suas características influenciaram o desenvolvimento de Ves.

1.3.3 Especificação formal

O objetivo dessa fase é apresentar a especificação léxica, sintática e semântica da linguagem, detalhando os comandos e estruturas presentes, além de seus usos e exemplos de código.

Nesta seção serão apresentados também detalhes da implementação do interpretador referente ao terceiro objetivo específico.

1.3.4 Definição das características

Esta fase objetiva definir características desejadas na linguagem, tais como, seu sistema de tipos, paradigmas abordados, estratégia de execução, nível de abstração e propósito.

1.3.5 Exemplos de uso

Esta fase tem objetivo de apresentar os casos de uso da linguagem, comparando-a com outras linguagens atuais e destacando sua usabilidade.

2 TRABALHOS CORRELATOS

Legibilidade e usabilidade são características desejadas em quase todas as linguagens de programação mas nem todas têm isso como foco principal, este capítulo contém um breve histórico de cada uma das linguagens que influenciaram Ves, ou que possuem objetivos similares. As linguagens nas quais Ves se baseia são:

- Ada;
- C++;
- C#;
- Dart;
- F#;
- Pascal;
- Python;
- Rust.

2.1 ADA

Ada é uma linguagem criada pelo departamento de defesa dos Estados Unidos na década de 1970 para ser usada em sistemas embarcados e de tempo real. A principal motivação para sua criação era o fato de que na época diversas linguagens eram usadas pelo departamento de defesa para desenvolvimento de sistemas nesses ambientes e muitas delas eram obsoletas, dependentes de *hardware* ou não suportavam programação modular de uma forma segura (WHITAKER, 1993). Em 1975 um grupo de especialistas chamado de *High Order Language Working Group*(HOLWG) foi formado com o objetivo de escolher uma linguagem existente, ou criar uma nova, que fosse ideal para desenvolvimento desse tipo de *software*. Diversos documentos especificando as características desejadas na linguagem foram escritos e refinados de forma incremental nos anos seguintes, até que em 1978 a especificação final foi escrita e chamada de *Steelman language requirements* (HOLWG, 1978). Como não existiam linguagens no mercado que seguissem essas especificações, uma nova foi criada e batizada de Ada, em homenagem

à Augusta Ada, Condessa de Lovelace. A linguagem Ada é imperativa, fortemente tipada, possui suporte nativo para *design-by-contract*, tipos dependentes e concorrência, além de possuir diversas verificações em tempo de compilação que detectam até mesmo certos erros que só acontecem em tempo de execução. As principais contribuições de Ada para Ves são o comando *loop* e a inspiração para a versão condicional dos comandos de quebra de laço.

2.2 C++

Bjarne Stroustrup da *Bells Labs* começou a trabalhar na década de 80 em uma nova linguagem de programação, chamada de "*C with Classes*". Essa linguagem era uma versão melhorada de C, adicionando uma série de novos recursos, sendo o mais importante, classes. Esta linguagem ao longos dos anos foi melhorada, aumentada e, finalmente, tornou-se C++ (OUALLINE, 2003). O objetivo de Stroustrup era implementar uma versão distribuída do núcleo Unix, usando C++ como uma extensão de C, mantendo todos os seus recursos e adicionando recursos novos (WALLACE, 1993).

Alguns dos desafios incluíram simular a infraestrutura da comunicação entre processos em um sistema distribuído ou de memória compartilhada, além de escrever drivers para o sistema. Quando se fala de meta-programação, C++ se destaca por causa do desenvolvimento de uma técnica chamada *template metaprogramming*. Segundo Stroustrup (2018), *templates* foram criados com objetivo de facilitar programação genérica, mas sua ideia era que *templates* deveriam ser extremamente flexíveis, a ponto de permitir que outros programadores usassem essa funcionalidade de formas que que nem Stroustrup sozinho poderia imaginar durante sua concepção e tendo zero *overhead*. Muitas das ideias e características de Ves se baseiam em C++, usando uma sintaxe similar a dos templates de C++ mas buscando ser mais limpa e legível, uma vez que esse é um dos grandes problemas de técnicas de meta-programação como SFINAE (*Substitution failure is not an error*), que permite compilação condicional e especialização parcial de *templates*.

Quanto a sua contribuição para o desenvolvimento de Ves, C++ inspirou o uso de chaves para delimitar blocos de código, os operadores lógicos e os de *bitwise*.

2.3 C#

2.4 DART

A linguagem Dart (DART..., 2018) é uma linguagem que foi originalmente desenhada para a *web*, a qual foi concebida durante a conferência GOTO realizada na Dinamarca em outubro de 2011, em um projeto criado pelos desenvolvedores Lars Bark e Kasper Lund. Como toda linguagem *client side*, linguagem que é executada no lado cliente, Dart precisou passar por uma série de testes junto à ECMA (*European Computer Manufacturer's Association*) internacional para verificar seu funcionamento em *browsers* modernos, tendo assim sua primeira especificação aprovada e liberada para a comunidade. Dart foi desenhada para poder facilmente escrever ferramentas de desenvolvimento para aplicações *web* modernas e capacitadas para ambientes de alto desempenho. Como características da linguagem é possível mencionar:

- É baseada em compilação de código *JavaScript*;
- Baseada em classes;
- Orientada a objetos;
- Tem sintaxe baseada na linguagem C;
- Implementa herança simples.

Segundo (BRACHA, 2015) Dart é uma linguagem de programação orientada a objetos pura, baseada em classe, de herança única, dando suporte a tipos genéricos, interfaces e *mixins*[1], ainda segundo ele, a linguagem tem um escopo léxico e usa um único *namespace* para variáveis, funções e tipos. É considerado um erro em tempo de compilação se houver mais de uma entidade com o mesmo nome declarado no mesmo escopo. Os nomes em escopos mais profundos podem esconder nomes nos mais externos, no entanto, é um aviso estático se uma declaração apresentar um nome que está disponível em um escopo de inclusão léxica. Esses nomes ainda podem ser introduzidos em um escopo por declarações ou por outros mecanismos, como importações ou herança. Dart influenciou Ves com sua biblioteca padrão, o suporte para funções assíncronas e *mixins*. Por conta das limitações de tempo neste trabalho,

[1] Mixin age como uma superclasse, contendo a funcionalidade desejada. A subclasse pode então herdar ou reusar a funcionalidade, mas não especializá-la.

as especificações dessas funcionalidades não serão apresentadas, sendo deixadas para trabalhos futuros.

2.5 F#

No início da década de 1970 Robin Milner e seus colegas Lockwood Morris e Malcolm Newey iniciaram a especificação de uma linguagem de programação funcional sucinta totalmente baseada em tipos, adequada para manipular informações estruturadas (MILNER, 1978). Foi fortemente baseada nos princípios de LISP, e inicialmente implementada em LISP, a linguagem foi chamada de ML (*Meta language*).

Mais tarde ML inspirou diversas outras linguagens funcionais fortemente tipadas, que incluem Edinburgh ML, Miranda, Haskell, Standard ML, OCaml, F#, Elm, ReasonML e PureScript.

As linguagens da família ML são frequentemente associadas ao formalismo, no entanto, uma preocupação primária de Milner e seus colegas desde o início, foi a usabilidade, eles precisavam de uma linguagem para programar de forma sucinta e precisa as regras de prova e transformações de um sistema de demonstração de teoremas chamado LCF (*Logic for Computable Functions*), que rodava em máquinas PDP10 (MACQUEEN, 2015). Dentre as características escolhidas se destacam a inclusão do estado mutável (para permitir que o estado de prova fosse armazenado em um sistema interativo) e um sistema de inferência de tipos, que mais tarde ficou conhecido como inferência de tipo Hindley-Milner, permitindo que o código para táticas derivadas fosse sucinto e automaticamente generalizado.

As características presentes em *Standard ML*, são base para o design de F#. Assim como acontece em todas as linguagem da familia ML:

- O paradigma central suportado por F# ainda é uma programação funcional fortemente tipada;
- As principais construções em F# são declarações de tipo e de funções, essa ultima tem tipos inferidos e automaticamente generalizados;
- F# ainda tem como objetivo apoiar um modo de programação em que o foco está no domínio que está sendo manipulado, e não nos detalhes da programação em si.

A ideia inicial do F# era simples: trazer os benefícios do OCaml para o .Net, unindo programação funcional fortemente tipada com o

ambiente .Net através da reimplementação do compilador de OCaml e parte de sua biblioteca padrão, adaptando-a ao *.Net Common Language Runtime* e permitindo interoperabilidade com outras linguagem como C#, Visual Basic e C++.

A primeira implementação de F# foi iniciada em dezembro de 2001 com um *front-end* que reconhecia as principais construções usando sintaxe de OCaml e visando usar o ILX (*Extended intermediate language*), que foi criado com objetivo de facilitar integração da infraestrutura existente com linguagens funcionais (SYME, 2001). Em 2006, a linguagem foi *bootstrapped*, ou seja, o compilador e a biblioteca da linguagem foram escritos na própria linguagem.

Com relação a sua influencia sobre este trabalho, F# inspirou fortemente a sintaxe usada por Ves para declaração de tipos, incluindo a sintaxe usada para declaração de uniões, que em F# são chamados de *Discriminated unions*, a sintaxe para compreensões de lista, expressões condicionais e funções anonimas.

2.6 PASCAL

O desenvolvimento da linguagem Pascal é baseada em dois objetivos principais. O primeiro é fazer a linguagem disponível para ensinar programação como uma disciplina sistemática de um modo claro e natural, o qual é refletido pela linguagem. O segundo é desenvolver implementações dessa linguagem, de modo que seja confiável e eficiente nos computadores disponíveis da década de 1970.

> Linguagens existentes devem ser usadas como base para desenvolvimento sempre que se encaixarem em seus objetivos, como estrutura sistemática, flexibilidade do programa e estruturação dos dados, e implementação eficiente (WIRTH, 1971).

Partindo deste contexto, a linguagem Algol 60 foi usada como base para o Pascal, pois ela busca a cumprir com maioria dessas demandas, porém em um grau muito mais elevado que qualquer outra linguagem e portanto, os princípios de estruturação, e a forma de expressão são copiadas da linguagem Algol 60. A maioria das declarações começa com uma única palavra chave, essa propriedade facilita o entendimento da sintaxe para ambos humanos e computadores e, de fato, a sintaxe do código fonte de Pascal pode ser processada pelas mais simples técnicas de análise sintática. A principal contribuição de Pascal

para Ves é a sintaxe simples e os tipos que podem ser construídos a partir de intervalos, ou seja, assim como em Pascal é possível declarar uma versão customizada de tipos escalares como *char*, *int* ou *float* que é tratada pelo compilador como um novo tipo e é dada garantia que esses valores não excederão o limite definido, por conta das limitações deste trabalho essas funcionalidades não serão especificadas ou implementadas mas ficam definidas como características desejadas para trabalhos futuros.

2.7 PYTHON

Criada por Guido Van Rossum no final da década de 80, Python é uma linguagem de programação multi paradigma, sendo que programação orientada a objetos e programação estruturada são totalmente suportadas, além disso muitos de seus recursos suportam programação funcional, programação orientada a aspectos, meta-programação e meta-objetos (ROSSUM; DRAKE, 2011).

Python ainda possui tipagem dinâmica e combina *reference counting* com *cycle-detecting garbage collector* para gerenciamento de memória. Também possui resolução dinâmica de nomes (*late binding*), que vincula nomes de métodos e variáveis durante a execução do programa.

Segundo (HETTINGER, 2012), o design do Python oferece suporte para programação funcional na tradição Lisp, fazendo uso de funções como *filter*, *map* e *reduce*; compreensões para dicionários, conjuntos, listas e *generators*. Já a biblioteca padrão possui dois módulos (itertools e functools) que implementam ferramentas funcionais emprestadas de Haskell e Standard ML (HELLMANN, 2011).

Quando se compara Python com outras linguagens de programação, como por exemplo Java (ROSSUM, 1997), certos aspectos devem ser levados em conta. Normalmente, espera-se que os programas escritos em Python sejam executados mais lentamente que os escritos em Java, todavia, eles também demoram muito menos tempo para serem desenvolvidos. Essa diferença pode ser atribuída aos tipos de dados de alto nível internos que Python possui, e à sua tipagem dinâmica. Por exemplo, um programador Python não perde tempo declarando os tipos de argumentos ou variáveis, as estruturas de dados são todas polimórficas e os dicionários de Python, para os quais o rico suporte sintático é construído diretamente na linguagem, são usados em quase todos os programas.

Python é uma linguagem com uma sintaxe bastante limpa, entre-

tanto, o uso obrigatório de indentação, enquanto positivamente reforça a prática de indentar corretamente o código, se torna problemática quando há mistura de *tabs* e espaços, tal prática também torna mais difícil o desenvolvimento da gramática e a implementação do analisador sintático.

Python contribuiu para o desenvolvimento de Ves inspirando a sintaxe para definição de tuplas, sua filosofia de desenvolvimento, que prioriza simplicidade e sua biblioteca padrão.

2.8 RUBY

Concebida em 1993 por Yukihiro Matsumoto, Ruby é uma linguagem interpretada e orientada a objetos focada primariamente no conforto do programador (MATSUMOTO, 2008), por isso sua sintaxe busca ser simples, limpa e a linguagem busca causar o minimo possível de confusão no que diz respeito a sua semântica.

Enquanto Ves compartilha parte da filosofia de Ruby com relação ao programador esse não é o seu objetivo principal, sendo que expressividade e flexibilidade na linguagem são priorizados, além disso, Ves não é unicamente orientada a objetos, buscando obter o melhor de orientação a objetos, programação funcional e outros paradigmas atuais.

Quanto à sintaxe, Ves difere em grande medida de Ruby, um exemplo simples é o fato de que parênteses não são necessários para chamadas de função em Ruby, mas são em Ves, as palavras reservadas, para declaração de funções e tipos também diferem, enquanto Ruby usa "*def*"e "*class*", Ves usa "*fun*"e "*type*"respectivamente.

2.9 RUST

Rust é uma nova linguagem de programação para o desenvolvimento de sistemas capazes e eficientes, com o objetivo de ser uma alternativa mais segura para C/C++ (REED, 2015). Foi projetada para dar suporte a paralelismo na construção de aplicações e bibliotecas que tiram o máximo de *hardwares* modernos, assim como eliminar o uso de *garbage collector*, a segurança do Rust fornece forte garantia sobre isolamento, simultaneidade e segurança de memória.

Segundo (MATSAKIS; II, 2014), Rust também oferece um modelo claro para performance, implicando em uma predição mais fácil

sobre a eficiência dos programa e uma maneira de conseguir tais garantias é permitindo um controle refinado das representações de memória, com suporte direto para alocação de pilha e armazenamento de registro contíguo. A linguagem acaba então por equilibrar esses controles com o requisito absoluto de segurança: o sistema de tipos e o ambiente em tempo de execução de Rust garantem a ausência de condições de corrida, *buffer overruns*, *stack overflows* e acessos a memória não inicializada ou desalocada.

(LIGHT, 2015) afirma que o uso de Rust para desenvolvimento de sistemas em geral, garante benefícios em relação ao uso da linguagem C ou C++, por exemplo, pelo fato de possuir construções de linguagem de alto nível que programadores esperam de linguagens como C ++, Java, Python e outros. Rust fornece abstrações, como métodos anexados a objetos, permitindo uma associação mais clara de dados com seus métodos, sobrecarga de operadores e um sistema de módulo para permitir agrupar a funcionalidade relacionada em um único *namespace*.

Tratando-se de metaprogramação e extensibilidade, (LIGHT, 2015) cita outro beneficio ao se usar Rust ao invés de C, fato de Rust possuir um sistema de *macros* e *plugins* muito abrangente e poderoso, uma vez que os *macros* de Rust causam transformações diretas na sua AST(*Abstract syntax tree*), esses são muito diferentes e mais poderosos que os de transformação de texto puro que estão disponíveis através do pré-processador de C. Com *macros* de C deve-se estar sempre ciente de que a expansão se dá no mesmo contexto de uso, isso significa que, em C, eles podem sobrescrever ou redefinir variáveis se o programador não estiver atento. Em Rust, os *macros* nunca sobrescreverão variáveis, a não ser que essas sejam explicitamente passada para eles.

Quanto sua influência no desenvolvimento de Ves, Rust contribuiu com algumas palavras reservadas como "*pub*"e "*priv*", que são uma alternativa mais curta para "*public*"e "*private*", respectivamente e assim como em Rust não são necessários parênteses cercando as condições de estruturas dos comandos *if while* e *for*.

2.10 COMPARAÇÃO

A linguagem de programação Ves visa ser legível, expressíva e flexível, nesse contexto será apresentada uma comparação entre essas características nas linguagens que a inspiraram. Sebesta (2012) apresenta alguns critérios para comparação de linguagens de programação, entretanto tais critérios são difíceis de se avaliar e por vezes vagos. Al-

guns desses critérios não serão levados em conta neste trabalho, uma vez que não se encaixam nos objetivos da linguagem.

2.10.1 Simplicidade e Ortogonalidade

A simplicidade geral de uma linguagem afeta diretamente sua legibilidade. Linguagens como Pascal que são voltadas para o ensino geralmente são possuem menos estruturas básicas, o que permite uma curva menor no aprendizado. Ves busca inspiração em linguagens como Pascal e Ada para tornar-se mais legível e facilitar seu aprendizado.

Linguagens como C++ possuem diversas estruturas, que operam de diversas formas, tornando-a uma linguagem mais difícil de se ler. Apesar de ter uma sintaxe parecida com C++, Ves busca simplificar as estruturas complicadas que podem surgir em C++.

2.10.2 Multiplicidade de funcionalidades

Multiplicidade de funcionalidades pode ser definida como a capacidade de realizar a mesma operação de diversas formas diferentes. Evidentemente a possibilidade de fazer uma coisa de duas formas diferentes pode causar confusão, além de demandar que os programadores entendam e estejam cientes de todas essas funcionalidades.

Nesse quesito destaca-se a linguagem *Python*, que apesar de possuir diversas funcionalidades tem em sua filosofia de desenvolvimento a redução de funcionalidades equivalentes. *"There should be one – and preferably only one – obvious way to do it. (PETERS, 2004)"*.

2.10.3 Sobrecarga de operadores

Enquanto sobrecarga de operadores pode ser extremamente útil em certos casos, essa funcionalidade pode afetar grandemente a legibilidade se não usada de forma adequada. O uso indiscriminado dessa funcionalidade acaba tornando operações sobre tipos customizados imprevisíveis, aumentando a propensão a erros no código.

Linguagens funcionais geralmente permitem a definição de novos operadores. Essa funcionalidade pode causar os mesmos problemas que a sobrecarga de operadores. Entretanto como os operadores são novos na linguagem não existe uma convenção sobre como deveriam operar, sendo que é necessário verificar a documentação ou mesmo a própria

definição do operador para que se entenda seu funcionamento.

2.10.4 Expressividade

Em uma linguagem de programação expressividade pode se referir a diversas características. Neste trabalho expressividade diz respeito a capacidade da linguagem de representar comportamentos ou estruturas de forma simples. Esse conceito é retomado em maior detalhe no Capitulo 5.

3 ESPECIFICAÇÃO FORMAL

Este capítulo do trabalho refere-se ao cumprimento do objetivo número dois e três, descritos previamente. Apresentando as especificações léxica, sintática e semântica da linguagem de modo formal, além de prover exemplos de código demonstrando o uso das estruturas descritas e detalhes superficiais da implementação do interpretador para prova de conceito.

3.1 ESPECIFICAÇÃO LÉXICA

Quando um código fonte é lido pelo analisador ele é segmentado em *tokens*, que são usados mais tarde para a fase de análise sintática. Esta seção tem como objetivo descrever as regras usadas pelo analisador léxico da linguagem Ves. Os arquivos de código fonte da linguagem Ves usam codificação *Unicode UTF-8* por padrão, atualmente não há uma forma de utilizar outra codificação no arquivo.

3.1.1 Comentários

Comentários podem ser de linha ou de bloco, enquanto o primeiro só se estende até o fim de uma linha, o segundo pode se estender por múltiplas linhas, desde que o bloco seja terminado antes do fim do arquivo. A sintaxe dos comentários é a mesma usada em C/C++ como pode ser visto na Figura 1.

```
// Este comentário termina no fim da linha
/* Este comentário tem

    várias linhas.
*/
```

Figura 1 – Exemplos de comentário em Ves

3.1.2 Operadores

Na Tabela 1 estão dispostos os símbolos considerados operadores pelo analisador léxico, ordenados de forma decrescente por precedência.

Tipo	Operadores
Prefixo	+ - ! ~
Acesso	. ::
Sufixo	...
Aritméticos	+ - * / % **
Bitwise	& | ^<<>>
Range	.. ..<
Igualdade	== !=
Comparação	<><= >=
Lógico	&& ||
Tupla	,
Atribuição	= += -= *= /= %= **= &= |= ~= ^= >>= <<=

Tabela 1 – Operadores de Ves ordenados por precedência

3.1.3 Identificadores

Identificadores são palavras começando com uma letra ou *underscore*, opcionalmente seguidos de uma letra, número, *underscore* ou apóstrofo. Identificadores também podem começar com apóstrofo, nesse caso, o identificador deve ser seguido ao menos de uma letra, número ou *underscore* mas não pode conter outro apóstrofo. Caracteres *Unicode* também são aceitos como identificadores, ou parte deles. A Figura 2 mostra o trecho da gramática do analisador léxico que descreve os identificadores.

```
fragment NameChar : NameStartChar
                  | '0'..'9'
                  | '_'
                  | '\u00B7'
                  | '\u0300'..'\u036F'
                  | '\u203F'..'\u2040'
                  ;
fragment NameStartChar : 'A'..'Z' | 'a'..'z'
                       | '\u00C0'..'\u00D6'
                       | '\u00D8'..'\u00F6'
                       | '\u00F8'..'\u02FF'
                       | '\u0370'..'\u037D'
                       | '\u037F'..'\u1FFF'
                       | '\u200C'..'\u200D'
                       | '\u2070'..'\u218F'
                       | '\u2C00'..'\u2FEF'
                       | '\u3001'..'\uD7FF'
                       | '\uF900'..'\uFDCF'
                       | '\uFDF0'..'\uFFFD'
                       ;

ID : NameStartChar (NameChar | TICK)*
   | TICK NameChar+
   | '_' NameChar+
   ;
```

Figura 2 – Trecho da gramática que reconhece identificadores

Exemplos de identificadores válidos:
test, _test, jorge's, x1, x', a'b'c, 'name, α
Exemplos de identificadores inválidos:
0abc, 'test', 'john's;

Como na maioria das linguagens de programação alguns identificadores são reservados para denotar declarações ou estruturas, portanto não são considerados válidos para nomear variáveis, funções, tipos e etc. As palavras reservadas para a linguagem Ves podem ser vistos na Tabela 2. É importante notar que algumas palavras reservadas não terão sua função especificada, uma vez que são reservadas para implementações futuras, essas palavras aparecem em itálico na tabela 2.

as	break	const	continue	*del*	else	except
false	finally	for	*from*	fun	if	*import*
in	*is*	let	loop	match	*mod*	nil
of	priv	prop	prot	pub	raise	return
trait	true	try	type	*use*	var	when
where	while	with	*yield*			

Tabela 2 – Palavras reservadas

3.1.4 Literais

Literais são uma forma de denotar valores para constantes de um conjuntos de tipos, em Ves constantes podem ser dos tipos:

- Inteiros;
- Números reais;
- Caracteres;
- Strings;

3.1.4.1 Literais inteiros

Literais inteiros são números do tipo *int* e podem ser expressados de diversas formas, a figura 3 sumariza as formas de representação desses literais.

```
/* BINARIO */    /* OCTAL */    /* DECIMAL */    /* HEXADECIMAL */
0b0010;          0o777;          420;            0xf;
0B0011;          0O755;          23;             0X1DeeD;
0b0001_0111;     0o7_5_0;        1_000;          0xdEaD_bEeF;
0B1000_0000;     0O7_5_4;        0_3_0;          0X1_7_A;
```

Figura 3 – Exemplos de literal inteiro

É importante notar que o sinal não faz parte do literal.

3.1.4.2 Literais de número real

Literais desse tipo têm três formas básicas e assim como acontece com os inteiros existe uma versão *builtin* chamada de *float*, que é de ponto flutuante. A Figura 4 apresenta os formatos aceitos pelo *lexer* de Ves para literais desse tipo.

```
/* FLOAT */
2.22;
3.14_15_93;
.001;
.000_6;
// O 'e' pode ser maiusculo ou minusculo
6.023e23;       // Constante de Avogadro
299.792458E+6;  // Velocidade da luz(Km/s)
9.10938356e-31; // Massa de um elétron
```

Figura 4 – Exemplos de literal de ponto flutuante

3.1.4.3 Literais de caractere

Esses literais representam um caractere e estão associados ao tipo *char*. A Figura 5 sumariza os tipos de literais de caractere aceitos em Ves.

```
/* CHAR */    /* ESCAPE SIMPLES */    /* ESCAPE HEX */
'a';          '\n';                   '\x1b';
' ';          '\r';                   '\x04';

/* ESCAPE UTF-16 */    /* ESCAPE UTF-32 */
'\u30C0';              '\U22423221';
'\u03B5';
```

Figura 5 – Exemplos de literal de caractere

Nas sequências de escape o número de dígitos depois do caractere *backslash* é fixo, ou seja, uma sequência de escape UTF-16, por exemplo, deve ter exatamente quatro dígitos depois do marcador "\u".

3.1.4.4 Literais de string

Literais de *string* são uma forma diferente de especificar uma cadeia de caracteres, que em Ves aparece cercada por aspas duplas e assim como literais de caractere aceita sequências de escape. Literais desse tipo produzem objetos do tipo *str*, a linguagem ainda não dá suporte para a criação de literais de *string* customizados, portanto literais desse tipo são exclusivamente do tipo *str*. A figura 6 mostra exemplos de literais de *string*.

```
/* STRING */
"Hello World!";
"\u30C0 = 3.1415193...";
```

Figura 6 – Exemplos de literal de *string*

3.1.5 Delimitadores

Delimitadores são caracteres, ou sequências deles, usados para marcar blocos de código ou seções dentro de declarações. É importante notar que alguns desses operadores têm significado diferente quando presentes em expressões. Em Ves os caracteres usados como delimitadores estão presentes na Tabela 3.

(	)	#[	]#	[	]	:
{	}	->	=>	=	|	

Tabela 3 – Delimitadores

3.2 ESPECIFICAÇÃO SINTÁTICA

Como descrito brevemente no capitulo 1, a análise sintática é o processo de ler uma *string* de caracteres, com o objetivo de associar esses caracteres com unidades sintáticas de uma gramática formal. Uma vez que o código é agrupado em unidades chamadas *tokens*, como descrito na seção anterior a análise sintática se inicia, terminando com a produção de uma árvore sintática.

Para a construção do analisador sintático, foi usada uma ferramenta chamada ANTLR (*Another tool for language recognigtion*). Essa ferramenta lê um arquivo contendo a especificação sintática da linguagem, que se encontrada no Anexo 1, e gera o código fonte do analisador na linguagem alvo, nesse caso *Python*.

O analisador gerado pela ferramenta deriva as regras partindo da regra que define uma unidade de compilação (um arquivo). Esse tipo de analisador é chamado *top-down* pois constrói a árvore de sintática a partir da sua raiz.

O analisador lê o arquivo de código Ves da esquerda para a direita e tenta derivar as produções na mesma ordem, esse tipo de analisador é chamado de LL (*Left-to-right with Leftmost derivation*). Normalmente analisadores desse tipo usam um número fixo de *lookahead*, que é a

quantidade de símbolos necessária para distinguir uma regra de produção das outras. O analisador gerado pela ferramenta ANTLR é do tipo LL(*) adaptativo, analisadores desse tipo não possuem *lookahead* fixo, eles usam quantos *tokens* forem necessários para determinar qual regra de produção deve ser escolhida.

3.2.1 Expressões

Ves se baseia na sintaxe das linguagens derivadas de C/C++ para expressões e usa os mesmos operadores de Python, exceto para a versão *inline* dos comandos *if* e *for*, nesses Ves usa a mesma sintaxe de F#. Na Figura 7 é possível ver exemplos de expressões em Ves.

```
/* OPERADORES */
/* Unários */
-1;      /* Negação */
+23;     /* Positivo */
!false;  /* Negação lógica */
~0xfff0; /* Negação binária */

/* Aritméticos binários */
1 + 2;  /* Soma */
2 - 1;  /* Subtração */
2 * 2;  /* Multiplicação */
8 / 2;  /* Divisão */
5 % 3;  /* Módulo(resto de divisão) */
2 ** 3; /* Potenciação */

true == true;  // Igualdade
true != false; // Desigualdade

// Comparação
1 > 2;
2 < 1;
3 >= 3;
3 >= 4;
4 <= 3;
3 <= 3;
```

```
// Lógicos binários
true && false; // and lógico
true || false; // or lógico

// Bitwise binários
3 & 1; // Bitwise and
3 | 1; // Bitwise or
3 ^ 1; // Bitwise xor

// Shift
1 << 2; // Shift para a esquerda
8 >> 2; // Shift para a direita

// Alternação
if 1 > 0 -> true else false;
if false -> "Never" else "Always";
```

Figura 7 Exemplos de expresões em Ves.

Note que em Ves, a divisão de dois inteiros resulta em um número de ponto flutuante e como mencionado anteriormente, o sinal de um número é a aplicação de um operador unário.

3.2.2 Tuplas

Uma tupla é uma coleção de valores que podem ser, ou não, de tipos diferentes. Ves usa a mesma sintaxe que Python para especificação de tuplas, elas são denotadas por expressões separadas por vírgula e

opcionalmente cercadas por parênteses. Para criar tuplas de 1 elemento é necessário adicionar uma vírgula depois da expressão que computa o valor do elemento e cerca-la com parênteses, como pode ser visto na Figura 7.

```
/* TUPLA */
1, 2;
((1, 2), (3, 4));
true, 1, 1.0, '1', "1";
```

Figura 8 – Exemplos de tupla

3.2.3 Listas

Listas em Ves são denotados por expressões separadas por vírgulas e cercadas de colchetes. Existe também a possibilidade de se usar compreensões de lista, como pode ser visto na Figura 9.

```
/* LISTA */
[1, 2, 3];
[for i in 1..100 -> i];
[for x in 0..100 if x % 2 == 0 -> x ];
```

Figura 9 – Exemplo de listas

3.2.4 Const

A palavra reservada *const*, introduz uma nova constante que é computada em tempo de compilação, sendo o equivalente em C++ a uma *constexpr*. Na Figura 10 é possível ver o trecho da gramática que descreve esse tipo de declaração e na Figura 11 exemplos de declarações desse tipo.

```
// CONST DECLARATION
const_decl : CONST name=ident (COLON type_=type_expr)? EQUAL init=expr
           ;
```

Figura 10 – Trecho da gramática que descreve declarações const

```
// CONST
const pi = 3.141593;
const x: int = 23;
// x = 1; // <- Erro! não é possível modificar constantes
```

Figura 11 – Exemplos de declaração const

3.2.5 Let

A palavra reservada *let* cria uma ligação imutável entre o identificador especificado e o valor produzido pela expressão no lado direito da atribuição. A Figura 12 contém o trecho da gramática que descreve esse tipo de declaração e a Figura 13 mostra exemplos de uso.

```
// LET DECLARATION
let_decl : LET name=ident (COLON type_=type_expr)? EQUAL init=expr
         | LET name=ident COLON type_=type_expr
         ;
```

Figura 12 – Techo da gramática que descreve declarações let

```
// LET
let numero = 1;
let nome: str = read("Informe seu nome:");
```

Figura 13 – Exemplo de declarações let

3.2.6 Var

A palavra reservada *var*, seguida de um identificador declara uma nova variável. Se a variável for inicializada o tipo pode ser omitido, do contrário, é necessário especificar o tipo da nova variável. Na Figura 14 é possível ver o trecho da gramática que define esse tipo de declaração e na Figura 15, exemplos de declaração de variáveis.

```
// VAR DECLARATION
var_decl : VAR name=ident (COLON type_=type_expr)? EQUAL init=expr
         | VAR name=ident COLON type_=type_expr
         ;
```

Figura 14 – Trecho da gramática que descreve declarações var

```
// VAR
var i = 0;
var y: f32 = 1 / 3;
```

Figura 15 – Exemplos de declaração de variável

3.2.7 If/else

Como em diversas outras linguagens de programação Ves possui um comando de alternação denominado *if*. Geralmente esse comando tem duas formas, uma forma que cria um novo bloco de código depois da condicional e uma que aceita somente um outro comando. Historicamente a segunda versão da condicional é conhecida por causar problemas de ambiguidade na gramática e por isso foi removida de Ves (GNU, 2019). A Figura 16 apresenta o trecho da gramática que descreve o comando *If/else* e a Figura 17 apresenta alguns exemplos de seu uso.

```
// IF STATEMENT
if_stmt : IF condition=expr then_branch=stmt_body
          (ELSE else_branch=stmt_body)?
        ;
```

Figura 16 – Trecho da gramática que descreve comandos *if*

```
// IF
if true {
    print("always prints");
}

if false {
    print("never prints");
} else {
    print("always prints");
}
```

Figura 17 – Exemplo do comando *if*

3.2.8 While

O comando *while* cria um laço que executa os comandos do corpo do bloco associado a ele enquanto uma condição for verdadeira. Assim como acontece em outras estruturas de controle o *while* não possui uma versão que aceite somente um comando como corpo. A Figura 18 apresenta o trecho da gramática que descreve esse comando e a Figura

19 apresenta um exemplo de seu uso.

```
// WHILE STATEMENT
while_stmt: WHILE condition=expr body=stmt_body
          ;
```

Figura 18 – Trecho da gramática que descreve o comando *while*

```
// WHILE
var x = 0;
while x < 10 {
    x += 1;
    print(x);
}
```

Figura 19 – Exemplo do comando *while*

3.2.9 Loop

Esse comando foi adaptado da linguagem de programação Ada e facilita a criação de laços infinitos ou com condições complexas de parada. Na Figura 20 é possível ver o trecho gramática que descreve esee comado e na Figura 21 um exemplo de uso.

```
// LOOP STATEMENT
loop_stmt: LOOP name=ident? body=stmt_body
         ;
```

Figura 20 – Trecho da gramática que descreve o comando *loop*

```
// LOOP
var x = 0;
loop {
    break if x == 1000;
    x+=1;
}
```

Figura 21 – Exemplo do comando *loop*

Loops são a única estrutura de controle de execução em Ves que podem ser nomeados, dessa forma é possível navegar entre *loops* aninhados como pode ser visto na Figura 22.

```
var x = 0;
loop outer {
    loop inner {
        x += 1;
        break outer if x == 100000;
        break inner /*<-can be omitted*/ if x % 10 == 0;
    }
   x += 100;
}
```

Figura 22 – Exemplo de comando *loop* nomeado

3.2.10 For

Assim como em outras linguagens de programação Ves possui o laço *for*, esse laço pode ser usado para iterar sobre qualquer objeto que suporte iteração, como listas e intervalos. Na Figura 23 o trecho da gramática que descreve esse comando é apresentado e na Figura 24, exemplos de seu uso.

```
// FOR STATEMENT
for_stmt: FOR item=ident (COLON type_=type_expr)
          IN iterator=expr body=stmt_body
        ;
```

Figura 23 – Trecho da gramática que descreve o comando *for*

```
// FOR
// i de 0 até 10 (intervalo fechado)
for i in 0..10 {
    print(i);
}
// i de 0 até 9 (intervalo aberto)
for i in 0..<10 {
    print(i);
}
```

Figura 24 – Exemplos de comando *for*

3.2.11 Comandos de controle de laço

Dentro do corpo dos comandos que criam laços é possível usar *break* e *continue* para parar o laço e pular para a próxima iteração, respectivamente. Nos laços do tipo *loop*, que podem ser nomeados e não possuem condição de término, existe uma variação desses comandos que inclui uma condição e a possibilidade de definir qual laço será alvo do comando, como visto nas Figuras 21 e 22.

3.2.12 Match

Essa estrutura é usada para realizar correspondência de padrões (*pattern matching*), várias linguagens de programação possuem essa funcionalidade e no caso de Ves, a sintaxe foi fortemente inspirada em F# e Rust. Assim como nessas linguagens, é possível expandir tuplas, escolher casos de uma união e verificar valores de tipos escalares. A Figura 25 apresenta o trecho da gramática que descreve essa estrutura e a Figura 26 mostra um exemplo de seu uso.

```
union_unpack : type_access tuple_expansion?
             ;

pattern : union_unpack
        | tuple_expansion
        ;

match_clause_body : (AS alias=ident)? (WHEN condition=expr)? DASH_GREATER stmt
                  |
                  ;

match_clause : pattern body=match_clause_body            #pattern_clause
             | lit (body=match_clause_body|SEMICOLON)    #literal_clause
             | WILDCARD body=match_clause_body                #discard_clause
             ;

match_stmt : MATCH value=expr LBRACE clauses+=match_clause* RBRACE
           .
```

Figura 25 – Trecho da gramática que descreve a declaração de *match*

```
// MATCH
let n = 10;
match n {
    1;2;3 -> print("muito baixo");
    x when n % 23 == 0 -> print("23!");
    _ -> print("ok");
}

match tryParseInt("a") {
    Some(x) -> print(x);
    None -> print("invalido");
}
```

Figura 26 – Exemplo do comando *match*

3.2.13 Try

Esse comando introduz um novo escopo, que é delimitado por chaves como nos demais comandos, esse bloco deve ser seguido de um ou mais blocos *except* ou um bloco *finally* ou ambos. Na figura 27 é possível ver o trecho da gramática que descreve esse comando.

```
// TRY STATEMENT
except_block : EXCEPT exc_name=ident (AS alias=ident)? stmt_body
             ;

try_stmt : TRY body=stmt_body excepts+=except_block* (FINALLY finally_body=stmt_body)?
         ;
```

Figura 27 – Trecho da gramática que descreve o comando *try except/finally*

3.2.13.1 Finally

Quando o *try* é combinado com o comando *finally*, é garantida a execução do segundo bloco, independentemente da eventual ocorrência de exceções dentro do primeiro bloco. Na Figura 28 é possível ver um exemplo de estruturas desse tipo.

```
/* TRY FINALLY */
var file = open("./file.txt");
try {
    for line in file {
        print(line);
    }
} finally {
    file.close();
}
```

Figura 28 – Exemplo de bloco do tipo *try-finally*

3.2.13.2 Except

Quando o *try* é combinado com o comando *except*, é possível tratar uma exceção de forma bastante semelhante ao que acontece em outras linguagens de programação. A Figura 29 exemplifica o uso dessa estrutura.

```
/* TRY EXCEPT */
let text = read("número: ");
try {
    let i = int(text);
    print("O número é: " + text);
} except Exception {
    print("Esse não é um número");
}
```

Figura 29 – Exemplo de bloco do tipo *try-except*

O comando *raise* é usado para "levantar" uma exceção, que interrompe a execução do programa e sai dos escopos recursivamente até ser tratada, se não tratada, uma exceção interrompe a execução do programa. A Figura 30 exemplifica o uso desse comando.

```
/* RAISE */
if n < 10 {
    raise Exception("n muito pequeno");
}
print(n);
```

Figura 30 – Exemplo do comando *raise*

3.2.14 With

With é um comando que facilita o uso de recursos que precisam ser desalocados depois do seu uso, como por exemplo arquivos, mas pode ser usado como uma forma genérica de gerenciamento de contexto. Na Figura 31 é possível ver o trecho da gramática que descreve dessa estrutura e na Figura 32 um exemplo de seu uso.

```
// WITH STATEMENT
with_stmt : WITH expr (AS alias=ident)? body=stmt_body
          ;
```

Figura 31 – Trecho da gramática que descreve o comando *with*

```
/* WITH */
with open("./file.txt") as file {
    for line in file {
        print(line);
    }
}
```

Figura 32 – Exemplo de comando *with*

O código da Figura 32. é equivalente ao da Figura 28.

3.2.15 Funções

Funções em Ves são denotadas pela palavra reservada *fun* seguida do nome da função, uma lista de parâmetros entre parenteses, opcionalmente o tipo de retorno e o corpo da função. Quando o tipo de retorno não é informado a função retorna *unit*. A Figura 34 apresenta o trecho da gramática que descreve a declaração de funções e a Figura 34 apresenta exemplos desse tipo de declaração.

```
// FUNCTION DECLARATION
fun_decl : FUN (name=ident|LPAREN operator=overloadable_operators RPAREN)
           LPAREN params+=param_decl (COMMA params+=param_decl)* RPAREN (COLON type_expr)?
           body=fun_body
         ;

fun_body : EQUAL expr SEMICOLON
         | EQUAL match_stmt
         | stmt_body
         | SEMICOLON
         ;
```

Figura 33 – Trecho da gramática que descreve a declaração de funções

```
/* FUNÇÕES */
fun add(x: int, y: int): int {
    return x + y;
}
```

Figura 34 – Exemplo de função

Existe também uma sintaxe alternativa para funções cujo corpo consiste somente de um comando, quando esse comando for um *return* a palavra reservada pode ser omitida, como mostrado na Figura 35.

```
fun add(x: int, y: int): int = x + y;
fun say(what: str) = print(what);
```

Figura 35 – Exemplo de sintaxe alternativa de funções

3.2.16 Funções anônimas

Esse tipo de função foi introduzido por Alonzo Church (FERNÁNDEZ, 2009) e é geralmente usada como argumento para funções de ordem mais alta ou para compor comportamentos mais complexos. Assim como o nome indica, essas funções não são intrinsecamente associadas a um identificador e podem conter anotações de tipo ou não. A Figura 36 exemplifica a criação dessas funções.

```
/* LAMBDA */
let sum = fun(x: int, y: int): int -> x + y;
let sum': (int, int)=>int = fun(x, y)-> x + y;
```

Figura 36 – Exemplo de funções anônimas

3.2.17 Tipos

Assim como em muitas outras linguagens, Ves permite ao programador definir novos tipos, que podem ser de duas categorias, registros ou uniões. Na Figura 37 é possível ver o trecho da gramática que descreve declarações de tipo para ambas as categorias.

```
// TYPE DECLARATION
type_decl : TYPE name=ID (COLON bases+=type_expr (COMMA bases+=type_expr)*)?
            body=type_body
          ;
field_decl : name=ident COLON type_=type_expr SEMICOLON
           ;

record_body : LBRACE record_member* RBRACE
            ;

union_body : EQUAL PIPE? options+=union_option (PIPE options+=union_option)*
             (SEMICOLON | WITH with_body=record_body)
         ;
record_member : field_decl
              | fun_decl
              | type_decl
              | meth_decl
              | prop_decl
              | trait_decl
              ;
type_body : record_body
          | union_body
          ;

union_option : name=ident OF type_=type_expr
             | name=ident
             ;
```

Figura 37 – Trecho da gramática que descreve a declaração de um tipo

3.2.17.1 Registros

Tipos dessa categoria também são conhecidos como "*classes*", "*structs*" ou "*records*" e são úteis para agrupar dados em atributos, além de proverem propriedades e métodos.

```
type Point {
    prop x: float;
    prop y: float;

}

type Record {
    prop _.id: int { get; priv set; }
}
```

Figura 38 – Exemplo de declaração de registros

3.2.17.2 Uniões

Também conhecidas como "*Tagged unions*", "*Discriminated unions*", "*Variants*" ou "*Sum types*", são estruturas de dados usadas para armazenar um valor que pode ser de um dado conjunto de tipos, além de permitir a criação de enumerações, se dado conjunto for vazio. Uniões em Ves possuem sua sintaxe baseada na de F#, sendo que basta remover o ponto e vírgula no código presenta na Figura 39 e ele se torna válido em F#.

```
/* UNIÕES */
type MyBool = True | False ;
type DeviceState = On | Off ;
type Something =
    | Integer of int
    | Float of float
    | Bool of bool
    | State of DeviceState
    ;
```

Figura 39 – Exemplo de uniões simples

Assim como em F#, Ves possui a cláusula *with*, que substitui o ponto e vírgula no final da declaração de uma união e permite a definição de métodos e propriedades que serão compartilhados entre todas as opções definidas anteriormente, como mostrado na Figura 40.

```
type DeviceState =
    | On
    | Off
with {
    fun this.toBool(): bool = match this {
            On -> true;
            Off -> false;
        }
    }
}
```

Figura 40 – Exemplo de união com cláusula *with*

3.2.18 Atributos

Um atributo é um membro de uma classe que guarda dados para uma instância, diferente das propriedades, atributos não têm *getters* ou *setters*, eles são como variáveis declaradas no contexto de um tipo. Em Ves atributos são os membros mais simples de se definir em um tipo e são sempre privados, dessa forma, só é possível acessá-los através de

métodos ou propriedades, como pode ser visto na Figura 41.

```
type SaleItem {
    _id: int;
    fun this.id(): int = this._id;
    fun this.id(new_id: int) = this._id = new_id;
    _price: float;
    fun this.price(): float = this._price;
}
```

Figura 41 – Exemplo da declaração de atributos

3.2.19 Métodos

Métodos são funções que pertencem a um tipo e operam sobre objetos, ou valores desse tipo. Para seu correto funcionamento o método geralmente precisa de uma forma de se referir ao objeto com o qual está operando, a forma mais simples de fazer isso é introduzir um identificador que é usado para acessar a instância, e nos referiremos a esse identificar desse ponto em diante como "identificador *this*". Algumas linguagens usam como identificador *this* uma palavra reservada, comumente *this* ou *self*, outras como F#, Python e Ves apesar de possuírem uma convenção de nome dão ao programador a liberdade de usar qualquer identificador válido, ou mesmo descartá-lo usando o caractere *underscore*(_). Em Ves a convenção para o identificador *this* é "*this*". Métodos em Ves são por padrão públicos a não ser que algum modificador de visibilidade seja adicionado antes da palavra reservada *fun*. Na Figura 42 é possível ver o trecho da gramática que descreve a declaração de um método e na Figura 43, exemplos dessas declarações.

```
// METHOD DECLARATION
meth_decl : visibility=(PROT|PUB|PRIV)?
            FUN self_id=opt_id DOT (name=ident|LPAREN operator=overloadable_operators RPAREN)
            LPAREN (params+=param_decl (COMMA params+=param_decl)*)? RPAREN (COLON type_expr)?
            body=meth_body
          ;

meth_body : EQUAL expr SEMICOLON
          | EQUAL match_stmt
          | stmt_body
          | SEMICOLON
          ;
```

Figura 42 – Trecho da gramática que descreve a declaração de um método

```
/* TIPOS */
type Person {
    prop name: str;

    fun _.say(what: str) = print(what);
    fun this.introduce() = print("Hi, I am " + this.name);
}
```

Figura 43 – Exemplo de declaração de métodos

Internamente a referência da instância é passada como primeiro parâmetro da função, de forma que na Figura 44, ambas as chamadas para "*introduce*" são equivalentes.

```
let jorge = Person();
jorge.name = "Jorge";

// Equivalentes:
jorge.introduce();
Person::introduce(jorge);
```

Figura 44 – Exemplos de chamada de método

3.2.20 Propriedades

O conceito de propriedades é bastante variado, algumas linguagens como Java e C++ permitem somente a definição de *getters* e *setters* para atributos, outras como C# possuem esse conceito nativamente. Uma das vantagens da definição de propriedades através de *getters* e *setters* é que podemos optar por não ter, ou mesmo restringir o acesso a um dos dois métodos e obtemos uma melhor granularidade no controle de acesso aos dados do objeto, por outro lado, a definição de dois métodos exige que mais código seja escrito para se obter um comportamento relativamente simples portanto, em Ves propriedades podem ser declaradas com o uso da palavra reservada *prop* e sua sintaxe é fortemente inspirada em C#. Assim como métodos, propriedades são por padrão públicas a não ser que algum modificador de visibilidade seja adicionado antes da palavra reservada *prop*, além disso propriedades permitem a alteração da visibilidade de seu *setter*, sendo que esse tem a mesma visibilidade da propriedade, exceto quando a palavra reservada contextual *set* for precedida de um modificador de visibilidade. Na Figura 45 pode-se ver o trecho da gramática que define declarações desse tipo e na Figura 46 exemplos de uso.

```
// PROPERTY DECLARATION
getter : GET body=meth_body
       ;

setter : visibility=(PROT|PUB|PRIV)? SET body=meth_body
       ;

prop_body : LBRACE getter setter RBRACE
          | LBRACE getter RBRACE
          | LBRACE setter getter RBRACE
          | LBRACE setter RBRACE
          ;

prop_decl : visibility=(PROT|PUB|PRIV)? PROP
            (self_id=opt_id DOT)? name=ident COLON type_=type_expr
            (EQUAL value=expr SEMICOLON|body=prop_body|SEMICOLON)
          ;
```

Figura 45 – Trecho da gramática que descreve a declaração de uma propriedade

```
/* PROPRIEDADES */
type SaleItem {
    prop id: int;
    _price: float;
    fun this.price(): float = this._price;
}
```

Figura 46 – Exemplo de propriedades

Apesar de possuírem uma sintaxe própria propriedades permitem, de forma semelhante a C#, que *getters* e *setters* sejam definidos e para isso possuem um "corpo"onde esses métodos podem ser implementados usando dois identificadores que são palavras reservadas somente nesse contexto. Internamente propriedades que possuem *getter* e *setter* criam um método para cada e por isso é necessário definir juntamente com a propriedade o nome para o identificador *this*, que será usado em ambos os métodos. A Figura 47 é uma reimplementação da Figura 46, trocando o método *price* por uma propriedade com *getter* público e setter privado.

```
type SaleItem {
    prop id: int;
    prop _.price: float { get; priv set; }
}
```

Figura 47 – Exemplo de propriedades com *getter* e *setter*

3.2.21 Traits

Também conhecidos como *protocols* ou *interfaces*, são usados para declarar métodos e propriedades, que podem mais tarde ser implementadas dentro de tipos, dessa forma não é possível implementar métodos de tipo algum diretamente no *trait*, incluindo *getters* e *setters*. A Figura 48 apresenta o trecho da gramática que especifica esse tipo de declaração e a Figura 49 mostra exemplos da declaração e implementação de um *trait*.

```
// TRAIT DECLARATION
trait_decl : TRAIT name=ident
             (COLON bases+=type_expr (COMMA bases+=type_expr)*)?
             LBRACE members+=trait_member* RBRACE
           ;

trait_member : meth_decl
             | prop_decl
             ;
```

Figura 48 – Trecho da gramática que descreve a declaração de um *trait*

```
/* TRAITS */
trait 'Vehicle {
    fun _.startEngine();
}

type Car: 'Vehicle {
    fun this.startEngine {
        // Implementação ...
    }
}
```

Figura 49 – Exemplo de implementação de *trait*

Por convenção em Ves o nome de *traits* começa com o caractere apóstrofo (’),uma vez que esses são identificadores válidos em Ves.

3.3 ESPECIFICAÇÃO SEMÂNTICA

Quando o código de um arquivo é lido pelo interpretador de Ves ele passa pela análise léxica e sintática, uma árvore é então construída, essa árvore é percorrida e um modelo orientado a objetos é construído. Esse modelo guarda as informações de cada declaração para uso nas fases posteriores de análise. Na figura 50 é possível ver o diagrama de classes desse modelo.

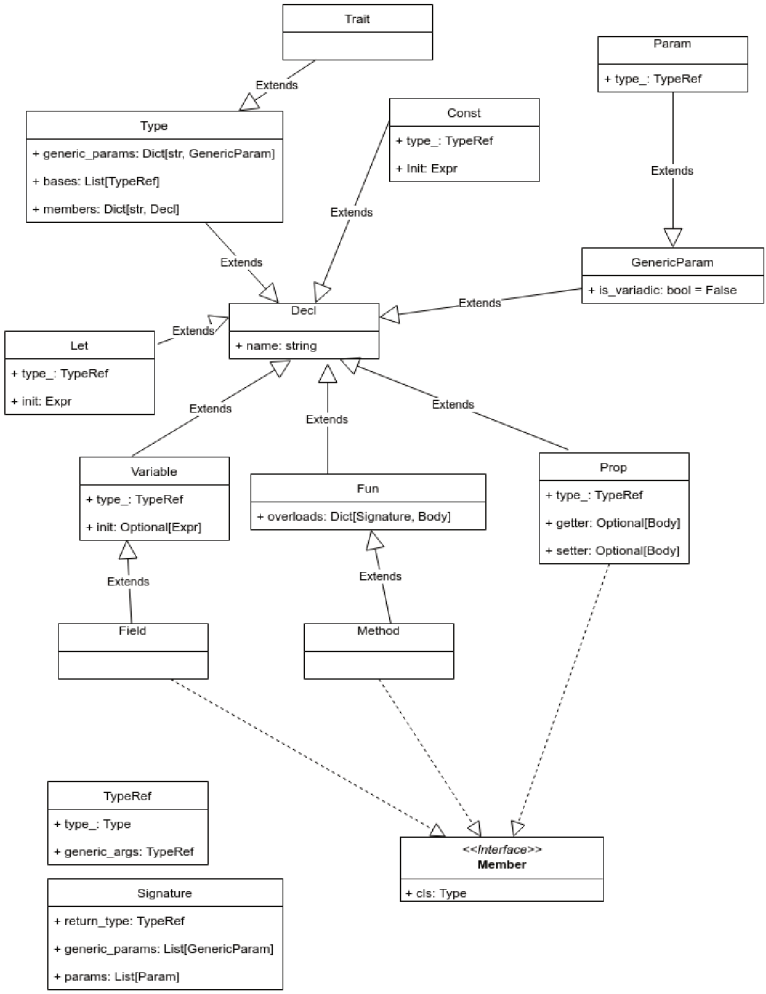

Figura 50 Diagrama de classes do modelo semântico

3.3.1 O tipo *unit*

O tipo *unit* é um tipo unitário, ou seja, só pode existir uma instancia dele em todo o programa, essa instancia é denotada por um par de parênteses vazio. Esse tipo é usado para denotar a ausência de valor no retorno de uma função. Em Ves todas as funções que não

têm anotação de tipo de retorno, retornam *unit* de forma implícita. Se o comando *return* aparecer em uma função que retorna *unit* então a expressão que segue a palavra reservada pode ser omitida.

3.3.2 Operadores

Em Ves os operadores são métodos de um dos objetos operados, dessa forma é possível sobrecarregar operadores de forma bastante simples em Ves. Para sobrecarregar operadores em Ves basta implementar um método cujo nome é substituído pelo simbolo entre parênteses. Na Figura 51, é possível ver um exemplo, onde do operador binário de soma é implementado para o tipo *Point*.

```
/* SOBRECARGA DE OPERADORES */
type Point {
    prop x: float;
    prop y: float;
    fun this.(+)(other: Point): Point {
        var p = Point();
        p.x = this.x + other.x;
        p.y = this.y + other.y;
        return p;
    }
}
```

Figura 51 – Exemplo de sobrecarga de operadores

3.3.3 Construtor

O construtor é um método que é chamado para iniciar os atributos de um objeto, em algumas linguagens como Pascal, C++ e C# construtores possuem uma sintaxe diferenciada e não podem ser referenciados diretamente como os outros métodos, em outras como *Python* e *Ruby* eles são métodos normais, que por convenção são chamados quando é necessário instanciar um objeto. Em Ves processo de instanciação começa com a chamada do operador de chamada (*call*) do tipo a ser instanciado, como operadores são métodos e tipos são objetos isso se dá de forma natural. A implementação do operador de chamada de um tipo aloca a memória necessária para uma instância e chama o método *init* da instância, que é implementado pelo programador. O método *init* pode ser sobrecarregado com diferentes parâmetros mas deve sempre retornar *unit*. Na Figura 52 é possível ver um exemplo de declaração e implementação de um construtor.

```
type Person {
    prop name: str;
    fun this.init(name: str) {
        print(name + " was just instantiated.");
        this.name = name;
    }
}
let jorge = Person("Jorge");
// output: Jorge was just instantiated.
```

Figura 52 – Exemplo de construtor

4 DEFINIÇÃO DAS CARACTERÍSTICAS

Assim como previsto no objetivo específico quatro, essa seção define as características da linguagem Ves. Linguagens de programação podem ser caracterizadas de diversas formas, neste trabalho serão levados em conta os seguintes aspectos:

- Tipagem;
- Paradigmas de programação;
- Nível de abstração;
- Propósito.

4.1 TIPAGEM

Em uma linguagem de programação o sistema de tipos é um conjunto de políticas que atribui a propriedade tipo a diversas estruturas, como expressões, variáveis e constantes (PIERCE; BENJAMIN, 2002), esses tipos permitem que validações sejam feitas e permitem a diferenciação de valores. Segundo (CARDELLI, 1997), o principal objetivo de um sistema de tipos é reduzir a possibilidade de *bugs* em um programa, através da definição de "interfaces"que conectam diferentes partes do sistema, em Ves anotações de tipo são obrigatórias em estruturas nas quais não é possível inferir o tipo durante a verificação de tipos, essas estruturas representam a grande maioria dos comandos em Ves, de forma que a linguagem pode ser considerada estaticamente tipada e com inferência em casos triviais, a Figura 53. exemplifica casos nos quais o tipo é trivial, e portanto, inferido.

```
var abool = true    // tipo: bool
var achar = 'a'     // tipo: char
var aint = 0;       // tipo: int
var afloat = 1.0    // tipo: float
var astr = "str"    // tipo: str
var atuple = 1, 'a' // tipo: (int, char)
var alist = [1, 2]  // tipo: [int]
var aunit = ()      // tipo: unit
```

Figura 53 Exemplo de tipos inferidos

Algumas linguagens como Ada, F# e Pascal são consideradas fortemente tipadas, o que quer dizer que, em geral, essas linguagens têm

regras mais estritas para tipos em tempo de compilação, por exemplo o código da Figura 54 não compila em F#, pois o tipo esperado para a função "f" é *float* e o tipo passado foi inferido como *int*.

```
let f (x: float) = x + 100.0 // float → float

let a = f 1 // float
```

Figura 54 – Exemplo de linguagem fortemente tipada.

Em Ves um código equivalente ao da Figura 54 compilaria pois os tipos numéricos são implicitamente convertidos uns para os outros.

4.2 PARADIGMAS DE PROGRAMAÇÃO

Ves suporta diversos paradigmas de programação, tentando extrair o melhor de cada abordagem para a resolução de problemas. Em Ves tudo é um objeto, o que caracteriza uma linguagem orientada a objetos, por outro lado a linguagem permite o uso de estratégias funcionais, que são caracterizadas por funções anônimas, declarações *let*, o comando *match* e uniões, independentemente da abordagem escolhida, Ves busca unir de forma harmoniosa esses paradigmas. Na Figura 55 estão exemplificadas duas formas equivalente de definir uma função, usando a palavra reservada *fun*, que explicitamente declara uma função ou ligando uma função anônima a um nome, demonstrando que funções em Ves são simplesmente objetos e podem ser guardadas em variáveis ou passadas como parâmetro em funções.

```
fun add(x: int, y: int): int = x + y;
let add: (int, int)=> int = fun(x, y)-> x + y;
```

Figura 55 – Formas diferentes de declarar uma função.

4.3 NÍVEL DE ABSTRAÇÃO

Ves é uma linguagem de programação de alto nível, isso quer dizer que as estruturas e conceitos da linguagem abstraem a maior parte das particularidades do *hardware* e sistema operacional no qual o programa está rodando. Abstração também permite de modo bastante simples representar o domínio do problema que o programa se propõe

a resolver. A Figura 56 mostra a implementação, em Ves, de uma calculadora simples, que percorre uma árvore de expressão de forma recursiva computando-a e retorna o seu resultado.

```
type Expr =
    | Add of (Expr, Expr)
    | Sub of (Expr, Expr)
    | Mul of (Expr, Expr)
    | Int of int
    ;


fun eval(expr: Expr): int = match expr {
    Add(lhs, rhs) -> eval(lhs) + eval(rhs);
    Sub(lhs, rhs) -> eval(lhs) - eval(rhs);
    Mul(lhs, rhs) -> eval(lhs) * eval(rhs);
    Int(x) -> x;
}

let expr = Add(Int(1), Int(2));
var ans = eval(expr);
```

Figura 56 – Exemplo de abstração em Ves.

4.3.1 Propósito

Ves é uma linguagem de programação de propósito geral e, portanto busca abstrair detalhes do domínio dos problemas em seu *design*. Para atingir tal objetivo, Ves se inspira em linguagens como Python, que possui uma vasta biblioteca padrão, permitindo à linguagem atender uma gama grande de domínios, mantendo sua sintaxe limpa e simples.

5 EXEMPLOS DE USO

Esta seção do trabalho se refere ao quinto objetivo específico, onde exemplos de código serão apresentados e comparados com outras linguagens de programação, a fim de destacar a expressividade, legibilidade e flexibilidade de Ves.

5.1 LEGIBILIDADE

Legibilidade diz respeito à facilidade do leitor de reconhecer palavras e estruturas na linguagem, por isso esse aspecto é bastante influenciado por preferencia pessoal e familiaridade com a linguagem. Nesta seção partes da sintaxe de Ves serão comparadas com as linguagens descritas no capítulo 1, buscando mostrar de um ponto e vista puramente pragmático as vantagens que Ves apresenta.

5.1.1 Estruturas de controle

Em Ves todas as estruturas de controle de execução introduzem um novo escopo delimitado por chaves, essa escolha foi feita numa tentativa de evitar enganos por parte do programador. No comando *while* da Figura 57 pode-se ter a impressão que a função *advance* é chamada a cada iteração, mas por conta do ponto e virgula depois da condição a função só é chamada quando o laço terminar.

```
while (matchChar('x'));
    advance();
```

Figura 57 – Exemplo de *while* com só um comando em C++.

Como os blocos em Ves são sempre explícitos, há uma menor chance de que o programador se engane, mesmo com indentação errada. Na Figura 58 é possível ver um código em Ves, equivalente ao da Figura 57, mantendo a indentação.

```
while matchChar('x') {}
    advance();
```

Figura 58 – Exemplo de *while* com bloco vazio em Ves.

5.1.2 Declarações

Em Ves todas as declarações começam com uma palavra reservada denotando exatamente o que está sendo declarado, isso facilita o entendimento do código e torna a curva de aprendizado da linguagem menos íngreme.

Como descrito no capítulo 2, Ves suporta a declaração de propriedades em tipos. A sintaxe dessa declaração foi inspirada na de C# mas buscando denotar mais claramente o que será declarado.

```
public class Person
{
    public string Name { get; set; }
}
```

Figura 59 – Declaração de propriedade em C#.

Para pessoas que estão habituadas é intuitivo ver que na Figura 59 é declarada uma propriedade chamada *Name*, porém não existe uma indicação explícita de que essa declaração cria uma propriedade, sendo que pode ser facilmente confundida com uma declaração de método ou atributo. Em Ves, declarações de propriedades começam com a palavra reservada *prop*, denotando explicitamente o que está sendo declarado. A Figura 60 apresenta como o código da Figura 59 seria escrito em Ves.

```
type Person {
    prop name: str;
}
```

Figura 60 – Declaração de propriedade em Ves.

5.2 EXPRESSIVIDADE

Expressividade diz respeito a capacidade da linguagem de representar comportamentos ou estruturas de forma simples. Nesta seção a

expressividade de Ves será demonstrada através de comparações com outras linguagens citadas no capítulo 1.

5.2.1 Anotações de tipo

Linguagens estaticamente tipadas usam anotações de tipo em declarações para permitir certas verificações durante a análise. A seguir algumas anotações de tipos em Ves serão comparadas com as de C#. Na Figura 61 é possível ver uma série de declarações de variáveis em C# demonstrando o uso de tipos da linguagem.

```
List<Person> people;
Dictionary<string, int> grades;
Tuple<int, int> point;
Func<List<float>, float> average;
```

Figura 61 – Exemplo de nomes de tipos em C#.

Em Ves alguns tipos possuem uma notação especial, isso permite que sejam definidos de forma mais simples. A Figura 62 é uma tradução do código da figura 61 para Ves.

```
var people: [People];
var grades: Dict[str, int];
var point: (int, int);
var average: ([float])=>float;
```

Figura 62 – Exemplo de nomes de tipos em Ves.

5.2.2 Funções

Como mostrado no capítulo 2, funções em Ves podem ser declaradas explicitamente de duas formas, uma que possui um corpo delimitado por chaves e outra na qual o corpo é trocado pelo operador de atribuição. C# possui uma sintaxe similar, entretanto em C# , como pode ser visto na Figura 63.

```
int add1(int x) {
    return x + 1;
}
// OU
int add1(int x)=> x + 1;
```

Figura 63 – Exemplo de sintaxe de função em C#.

Em Ves o operador (=>) é usado para denotar o tipo de uma função, por isso operador de atribuição é usado. Diferentemente de C#, em Ves o operador pode ser seguido não só de uma expressão, mas também do comando *match*, como visto na Figura .

```
fun StateToBool(state: DeviceState): bool =
    match state {
        On -> true;
        Off -> false;
    }
```

Figura 64 – Exemplo de função com correspondência de padrões em Ves.

5.3 FLEXIBILIDADE

Flexibilidade neste trabalho diz respeito a capacidade da linguagem de se adaptar a diferentes paradigmas e estratégias de resolução de problemas.

5.3.1 Laços infinitos

Como mostrado no capítulo 2, Ves possui uma estrutura específica para laços infinitos chamada *loop*. Essa estrutura não é comum nas linguagens de programação atuais, sendo que das citadas no capítulo 1, somente Ada possui uma estrutura similar, mostrando que Ves flexibiliza o uso de laços. Na Figura 65 é possível ver um exemplo de uso dessa estrutura, combinada com controle condicional do laço, que seria mais difícil de implementar em outras linguagens.

```
loop mainLoop {
    loop eventLoop {
        doThis();
        break mainLoop if state == State.Error;
        //...
        DoThat();
        continue if state == State.Retry;
        //...
        finalize();
        break if state == State.Success;
    }
}
```

Figura 65 – Exemplo de utilidade de *loop*.

5.3.2 Funções anônimas

Funções anônimas são bastante uteis e presentes em diversas linguagens. Na Figura 66 é apresentado um exemplo de função anônima em *Python*, que não suporta anotações de tipo.

```
# f(x: Person)-> None
f = lambda x: x.introduce()
```

Figura 66 – Exemplo de função anônima em *Python*.

Em Ves esse tipo de função pode conter anotações de tipo, ou não. A sintaxe para esse tipo de declaração foi pensada especialmente para flexibilizar seu uso, sem perder poder de expressividade. Como pode ser visto na Figura 67

```
let f = fun(x: Person): unit -> x.introduce();
let f: (Person)=>unit = fun(x)-> x.introduce();
```

Figura 67 – Exemplo de função anônima em Ves.

6 CONCLUSÃO

Este trabalho apresenta uma linguagem de programação multiparadigma e de propósito geral chamada Ves, como uma alternativa no desenvolvimento de software. Considerando a grande demanda nessa área e a constante busca por técnicas que permitam maior abstração sem perda de expressividade, a linguagem Ves busca atender a esta necessidade.

No capítulo inicial foram apresentadas linguagens que tiveram influência sobre Ves, contextualizando o desenvolvimento de cada uma delas, a fim de prover embasamento para escolhas de projeto feitas em fases posteriores.

Durante a especificação de Ves foi mostrado que algumas estruturas léxicas e sintáticas foram fortemente inspiradas nas linguagens previamente descritas, sempre buscando consistência e simplicidade. Os grandes desafios da especificação foram, manter a legibilidade, flexibilidade e expressividade equilibradas, de forma que o código seja fácil de entender, adaptável e conciso, além de prover uma semântica sã e bem definida.

Na fase de definição das características foram apresentados tópicos que aprofundam o modelo semântico de Ves, de forma a permitir uma compreensão melhor do funcionamento da linguagem e direcionar o desenvolvimento de trabalhos futuros.

No último capítulo uma comparação entre Ves e outras linguagens usadas na atualidade foi feita, demonstrando a legibilidade, flexibilidade e expressividade da linguagem descrita.

A base teórica provida pelas linguagens analisadas no Capitulo 1 foi essencial para o projeto de Ves, agregando estruturas e conceitos que são importantes para atingir os objetivos propostos. Assim, a partir das observações feitas conclui-se que, apesar de estar ainda em estágios iniciais de seu desenvolvimento, Ves cumpre os objetivos aos quais se propõe e pode sim ser usada futuramente para desenvolvimento de software.

Como trabalhos futuros sugere-se a implementação completa do interpretador, ou a implementação de um compilador, para a linguagem, a inclusão de suporte nativo para concorrência. Quanto à sintaxe, sugere-se a inclusão de herança para registros, suporte para tipos genéricos, a implementação de tipos escalares customizados, interpolação de texto e o enriquecimento do modelo semântico de forma a tornar a linguagem mais robusta.

REFERĘNCIAS

ABES, S. *Histórico do Estudo Mercado Brasileiro de Software*. 2018. <http://www.abessoftware.com.br/dados-do-setor/estudo-2018--dados-2017>. Acessado em 2019-07-13.

AHO, A. V. *Compilers: principles, techniques and tools (for Anna University), 2/e*. [S.l.]: Pearson Education India, 2003.

BRACHA, G. *The Dart Programming Language*. [S.l.]: Addison-Wesley Professional, 2015.

CARDELLI, L. *Handbook of Computer Science and Engineering, chapter Type Systems*. [S.l.]: CRC Press, 1997.

CHOMSKY, N. Three models for the description of language. *IRE Transactions on information theory*, IEEE, v. 2, n. 3, p. 113–124, 1956.

DART programming language. 2018. <https://www.dartlang.org>. Acessado em 2018-11-25.

FERNÁNDEZ, M. *Models of Computation: An Introduction to Computability Theory*. [S.l.]: Springer Science & Business Media, 2009.

GNU. *Shift-Reduce*. 2019. <https://www.gnu.org/software/bison/manual/html_node/Shift_002freduce.html>. Acessado em 2019-5-28.

HELLMANN, D. *The Python Standard Library by Example: PYTH STAND LIB BY EXAM _p1*. [S.l.]: Addison-Wesley Professional, 2011.

HETTINGER, R. Pep 289–generator expressions. *Python Enhancement Proposals*, 2012.

HOLWG. *Steelman online*. 1978. <https://dwheeler.com/steelman/steelman.htm>. Acessado em 2019-05-29.

LIGHT, A. *Reenix: Implementing a unix-like operating system in rust*. Tese (Doutorado) — Master's thesis, Brown University, Department of Computer Science, 2015.

MACQUEEN, D. *The history of Standard ML*: Ideas, principles, culture. 2015. <http://sml-family.org/history/ML2015-talk.pdf>. Acessado em 2018-11-25.

MATSAKIS, N. D.; II, F. S. K. The rust language. In: ACM. *ACM SIGAda Ada Letters*. [S.l.], 2014. v. 34, n. 3, p. 103–104.

MATSUMOTO, Y. *Ruby 1.9*. 2008. <https://www.youtube.com/watch?v=oEkJvvGEtB4>. Acessado em 2018-11-26.

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

OUALLINE, S. *Practical C++ programming*. [S.l.]: "O'Reilly Media, Inc.", 2003.

PETERS, T. *The Zen of Python*. 2004. <https://www.python.org/dev/peps/pep-0020/>. Acessado em 2019-07-13.

PIERCE, B. C.; BENJAMIN, C. *Types and programming languages*. [S.l.]: MIT press, 2002.

REED, E. Patina: A formalization of the rust programming language. *University of Washington, Department of Computer Science and Engineering, Tech. Rep. UW-CSE-15-03-02*, 2015.

ROSSUM, G. V. *Comparing Python to Other Languages*. 1997. <https://www.python.org/doc/essays/comparisons>. Acessado em 2018-11-24.

ROSSUM, G. V.; DRAKE, F. L. *The python language reference manual*. [S.l.]: Network Theory Ltd., 2011.

SEBESTA, R. W. *Concepts of programming languages*. [S.l.]: Boston: Pearson,, 2012.

STROUSTRUP, B. *CONCEPTS: The Future of Generic Programming (the future is here)*. 2018. <https://www.youtube.com/watch?v=HddFGPTAmtU&vl=en>. Acessado em 2018-12-02.

SYME, D. Ilx: Extending the .net common il for functional language interoperability. In: . Springer-Verlag, 2001. <https://www.microsoft.com/en-us/research/publication/ilx-extending-the-net-common-il-for-functional-language-interoperability/>.

WALLACE, C. *The semantics of the C++ programming language*. [S.l.]: Citeseer, 1993.

WHITAKER, W. A. Ada—the project: the dod high order language working group. In: ACM. *ACM SIGPLAN Notices*. [S.l.], 1993. v. 28, n. 3, p. 299–331.

WIRTH, N. The programming language pascal. *Acta informatica*, Springer, v. 1, n. 1, p. 35–63, 1971.

```
parser grammar Ves;

options { tokenVocab=VesLex; }



expr : op=(PLUS | MINUS | BANG | TILDE)          rhs=expr     #unary
     | lhs=expr op=STAR_STAR                     rhs=expr     #pow
     | lhs=expr op=(STAR|SLASH|PERCENT)          rhs=expr     #mul
     | lhs=expr op=(PLUS|MINUS)                  rhs=expr     #sum
     | lhs=expr op=AMPERSAND                     rhs=expr     #bin_and
     | lhs=expr op=PIPE                          rhs=expr     #bin_or
     | lhs=expr op=CARET                         rhs=expr     #xor
     | lhs=expr op=(GREATER_GREATER|LESS_LESS)   rhs=expr     #shift
     | low=expr ((DOT_DOT) step=expr)?
                op=(DOT_DOT|DOT_DOT_LESS)        high=expr    #range_expr
     | lhs=expr op=(EQUAL_EQUAL|BANG_EQUAL)      rhs=expr     #eq
     | lhs=expr op=(LESS|GREATER|
                    GREATER_EQUAL|LESS_EQUAL)    rhs=expr     #comp
     | lhs=expr AMPERSAND_AMPERSAND              rhs=expr     #bool_and // These two
     | lhs=expr PIPE_PIPE                        rhs=expr     #bool_or  // are short-
circuit
     | IF condition=expr DASH_GREATER then_branch=expr
       ELSE else_branch=expr                                  #expr_if
     | FUN LPAREN params+=lambda_param (COMMA params+=lambda_param) RPAREN
         (COLON return_t=type_expr)? DASH_GREATER expr #lambda_expr
     | lhs=expr op=COMMA                         rhs=expr     #comma
     | access                                                   #acc_expr
     | RAISE expr                                           #raise_expr
     ;



assignment_ops : PLUS_EQUAL | MINUS_EQUAL| STAR_EQUAL | SLASH_EQUAL
               | PERCENT_EQUAL | STAR_STAR_EQUAL | PIPE_EQUAL
               | AMPERSAND_EQUAL| CARET_EQUAL| GREATER_GREATER_EQUAL
               | LESS_LESS_EQUAL | EQUAL
               ;


assign : lhs=access op=assignment_ops match_stmt
       | lhs=access op=assignment_ops expr SEMICOLON
       ;

lambda_param : name=opt_id (COLON type_=type_expr)?
             ;

tuple_expansion : LPAREN items+=opt_id (COMMA items+=opt_id)* RPAREN
                | LPAREN items+=opt_id COMMA RPAREN
                ;

// MATCH EXPRESSION

union_unpack : type_access tuple_expansion?
             ;

pattern : union_unpack
        | tuple_expansion
        ;

match_clause_body : (AS alias=ident)? (WHEN condition=expr)? DASH_GREATER stmt
                  |
                  ;

match_clause : pattern body=match_clause_body             #pattern_clause
             | lit (body=match_clause_body|SEMICOLON)     #literal_clause
             | WILDCARD body=match_clause_body                 #discard_clause
             ;

match_stmt : MATCH value=expr LBRACE clauses+=match_clause* RBRACE
           ;

unit : LPAREN RPAREN
     ;


lit : INT_L
    | CHR_L
```

```
    | FLT_L
    | STR_L
    | TRUE
    | FALSE
    | NIL
    | unit
    ;


term : LPAREN expr RPAREN                                                    #grouping
     | LPAREN assign RPAREN                                                  #assgin_expr
     | LBRACKET expr RBRACKET                                                #list_lit
     | LBRACE dict_item (SEMICOLON dict_item)* SEMICOLON? RBRACE             #dic_lit
     | LBRACKET FOR ID IN expr (IF expr)? DASH_GREATER expr RBRACKET          #list_comp
     | lit                                                                   #lit_term
     | ident                                                                 #name_access
     ;
dict_item: key=expr COLON value=expr
         ;

access : term                                                                    #primary
       | COLON_COLON access
#global_access
       | lhs=access COLON_COLON rhs=ident
#static_access
       | lhs=access DOT rhs=ident
#instance_access
       | lhs=access LBRACKET (expr (COMMA expr)*)? RBRACKET
#item_access
       | callee=access LPAREN (args+=expr (COMMA args+=expr)*)? RPAREN
#call_access
       ;

generic_param : ident             #generic_name
              | ELLIPSIS          #unnamed_variadic
              | ident ELLIPSIS    #variadic
              ;



tuple_type : UNIT
#unit_type
           | LPAREN item=type_expr COMMA RPAREN
#tuple_of_one
           | LPAREN items+=type_expr (COMMA items+=type_expr)+ RPAREN         #tuple_t
           ;

list_type : LBRACKET type_expr RBRACKET
          ;

fun_type : LPAREN (args+=type_expr (COMMA args+=type_expr)*)? RPAREN EQUAL_GREATER
return_t=type_expr
         ;

dict_type : LBRACE type_expr COLON type_expr RBRACE
          ;

type_access : type_access COLON_COLON ident
            | type_access LBRACKET (type_expr (COMMA type_expr)*)? RBRACKET
            | ident
            ;

type_expr : type_access
          | tuple_type
          | list_type
          | fun_type
          | dict_type
          ;

param_decl : name=opt_id COLON type_=type_expr (EQUAL default=expr)?
           ;


// TYPE DECLARATION
type_decl : TYPE name=ID (COLON bases+=type_expr (COMMA bases+=type_expr)*)?
            body=type_body
          ;
field_decl : name=ident COLON type_=type_expr SEMICOLON
```

```
           ;

record_body : LBRACE record_member* RBRACE
            ;

union_body : EQUAL PIPE? options+=union_option (PIPE options+=union_option)*
             (SEMICOLON | WITH with_body=record_body)
         ;
record_member : field_decl
              | fun_decl
              | type_decl
              | meth_decl
              | prop_decl
              | trait_decl
              ;
type_body : record_body
          | union_body
          ;

union_option : name=ident OF type_=type_expr
             | name=ident
             ;


// WHERE DECLARATION
where_decl : param=ident COLON constraints+=type_expr (AMPERSAND
constraints+=type_expr)*;

// PROPERTY DECLARATION
getter : GET body=meth_body
       ;

setter : visibility=(PROT|PUB|PRIV)? SET body=meth_body
       ;

prop_body : LBRACE getter setter RBRACE
          | LBRACE getter RBRACE
          | LBRACE setter getter RBRACE
          | LBRACE setter RBRACE
          ;

prop_decl : visibility=(PROT|PUB|PRIV)? PROP
            (self_id=opt_id DOT)? name=ident COLON type_=type_expr
            (EQUAL value=expr SEMICOLON|body=prop_body|SEMICOLON)
          ;

// METHOD DECLARATION
meth_decl : visibility=(PROT|PUB|PRIV)?
            FUN self_id=opt_id DOT (name=ident|LPAREN operator=overloadable_operators
RPAREN)
            LPAREN (params+=param_decl (COMMA params+=param_decl)*)? RPAREN (COLON
type_expr)?
            body=meth_body
          ;

meth_body : EQUAL expr SEMICOLON
          | EQUAL match_stmt
          | stmt_body
          | SEMICOLON
          ;


overloadable_operators : AMPERSAND | AMPERSAND_AMPERSAND | AMPERSAND_EQUAL
                       | BANG | BANG_EQUAL | CARET | CARET_EQUAL | EQUAL | EQUAL_EQUAL
                       | GREATER | GREATER_GREATER | GREATER_GREATER_EQUAL |
GREATER_EQUAL | LESS | LESS_EQUAL
                       | LESS_LESS | LESS_LESS_EQUAL | MINUS | MINUS_EQUAL | PERCENT
| PERCENT_EQUAL
                       | PIPE | PIPE_EQUAL | PIPE_PIPE | PLUS | PLUS_EQUAL | SLASH |
SLASH_EQUAL | STAR
                       | STAR_EQUAL | STAR_STAR | STAR_STAR_EQUAL | TILDE |
TILDE_EQUAL
                       ;

// FUNCTION DECLARATION
fun_decl : FUN (name=ident|LPAREN operator=overloadable_operators RPAREN)
           LPAREN params+=param_decl (COMMA params+=param_decl)* RPAREN (COLON
type_expr)?
```

```
           body=fun_body
         ;

fun_body : EQUAL expr SEMICOLON
         | EQUAL match_stmt
         | stmt_body
         | SEMICOLON
         ;


// LOOP CONTROL
loop_ctrl_stmt : BREAK which=ident? (IF expr)? SEMICOLON              #break_stmt
               | CONTINUE which=ident? (IF expr)? SEMICOLON           #continue_stmt
               ;

// LOOP STATEMENT
loop_stmt: LOOP name=ident? body=stmt_body
         ;

// WHILE STATEMENT
while_stmt: WHILE condition=expr body=stmt_body
          ;

// FOR STATEMENT
for_stmt: FOR item=ident (COLON type_=type_expr)?
          IN iterator=expr body=stmt_body
        ;

// IF STATEMENT
if_stmt : IF condition=expr then_branch=stmt_body
          (ELSE else_branch=stmt_body)?
         ;

// WITH STATEMENT
with_stmt : WITH expr (AS alias=ident)? body=stmt_body
          ;


// TRY STATEMENT
except_block : EXCEPT exc_name=ident (AS alias=ident)? stmt_body
             ;

try_stmt : TRY body=stmt_body excepts+=except_block* (FINALLY finally_body=stmt_body)?
         ;

// CONST DECLARATION
const_decl : CONST name=ident (COLON type_=type_expr)? EQUAL init=expr
           ;

// LET DECLARATION
let_decl : LET name=ident (COLON type_=type_expr)? EQUAL init=expr
         | LET name=ident COLON type_=type_expr
         ;

// VAR DECLARATION
var_decl : VAR name=ident (COLON type_=type_expr)? EQUAL init=expr
         | VAR name=ident COLON type_=type_expr
         ;

// TRAIT DECLARATION
trait_decl : TRAIT name=ident
             (COLON bases+=type_expr (COMMA bases+=type_expr)*)?
             LBRACE members+=trait_member* RBRACE
           ;

trait_member : meth_decl
             | prop_decl
             ;

// USE DECLARATION
type_pattern : WILDCARD
             | ident ELLIPSIS?
             | ELLIPSIS
             | UNIT
             | LPAREN type_pattern COMMA RPAREN
             | LPAREN type_pattern (COMMA type_pattern)* RPAREN
             | (LBRACKET type_pattern (COMMA type_pattern)* RBRACKET)?
                LPAREN type_pattern (COMMA type_pattern)* RPAREN EQUAL_GREATER
```

```
type_pattern
             | ident LBRACKET type_pattern (COMMA type_pattern)* RBRACKET
             | LBRACKET type_pattern RBRACKET
             ;
use_decl : USE name=ident (LBRACKET patterns+=type_pattern (COMMA
patterns+=type_pattern)* RBRACKET)?
           EQUAL type_expr SEMICOLON
         ;

decl : const_decl
     | let_decl
     | var_decl
     | fun_decl
     | type_decl
     | trait_decl
     ;


stmt : expr SEMICOLON
     | assign
     | match_stmt
     | decl
     | if_stmt
     | for_stmt
     | while_stmt
     | loop_stmt
     | loop_ctrl_stmt
     | stmt_body
     | with_stmt
     | try_stmt
     | use_decl
     | RETURN expr? SEMICOLON
     | SEMICOLON
     ;

ident : ID
      | GET
      | SET
      ;

opt_id : ident
       | WILDCARD
       ;

stmt_body : LBRACE stmts+=stmt+ RBRACE      #stats_body
          | LBRACE RBRACE                   #empty_body
          ;

vesfile : stmts+=stmt* EOF
        ;
```

Linguagem de programação Ves

Henry Rodrigues Da Silva[1]

[1]Departamento de Informática e Estatística – Universidade Federal de Santa Catarina (UFSC)
Campus Universitário – Florianópolis – SC – Brasil

henry.rs@grad.ufsc.br

Abstract. *There are many programming languages in the market, each one of them has its strengths and weaknesses. This project introduces Ves, a programming language that aims to be expressive, legible and flexible, using concepts adapted from other languages intending to improve their features, or even ease the use of techniques that already exist, but are hard to work with in other languages. In order for the language to be able to fulfill these requirements the project focuses in meta-programming and type-safety, favoring a declarative syntax.*

Resumo. *Existem diversas linguagens de programação no mercado, cada uma com seus pontos fortes e fracos. Este trabalho apresenta Ves, uma linguagem de programação focada em expressividade, legibilidade e flexibilidade, utilizando conceitos adaptados de outras linguagens de forma tentar melhorar funcionalidades presentes nas mesmas, ou ainda, facilitar o uso de técnicas que já existem mas são difíceis de se trabalhar nas linguagens atuais. Para que a linguagem possa cumprir com esses requisitos é dado foco em meta-programação e type-safety, dando preferência a uma sintaxe mais declarativa.*

1. Introdução

O desenvolvimento de software é uma área em constante crescimento por parte de empresas e entidades públicas que visam atender cada vez mais a demanda do mercado por softwares de propósitos específicos e com alta qualidade. O mercado brasileiro por exemplo, movimentou 39,6 bilhões de dólares em 2016 o que representou 2,1% do PIB do país e 1,9% do total de investimentos de TI do mundo [ABES 2018].

Para se obter a qualidade desejada em seus projetos de *software*, as empresas geralmente utilizam-se de paradigmas, padrões e linguagens, que auxiliam a alcançar melhores resultados quando aplicados em domínios específicos para os quais foram criados e também testados.

Cada linguagem foca em diferentes aspectos do desenvolvimento de *software*, sempre tentando ser ideal dentro de seu nicho. Tendo em vista a grande demanda no desenvolvimento de software, bem como a necessidade de linguagens que implementem de forma inovadora novas funcionalidades necessárias, este trabalho apresenta uma nova linguagem de programação de propósito geral, focada em legibilidade, expressividade e flexibilidade.

A linguagem será chamada Ves[1]. Utilizando-se conceitos adaptados de outras linguagens, Ves busca facilitar o uso de abordagens e paradigmas de desenvolvimento

[1]O nome Ves vem de uma língua fictícia criada pelo autor e significa água. Foi escolhido por ser simples e para denotar a fluidez da linguagem.

de *software*, que estão presentes em outras linguagens mas geralmente são difíceis de se trabalhar.

2. Projeto de linguagens

O projeto de uma linguagem de programação geralmente produz três especificações, a léxica, sintática e semântica.

Quando um interpretador ou compilador lê o código fonte, esse código passa pelo analisador léxico. Durante essa fase as palavras e símbolos presentes no código fonte são segmentados em unidades léxicas chamadas de *tokens*. Segundo [Aho 2003] um *token* é um par consistindo do nome do *token* e um atributo opcional chamado de valor. O nome do *token* é um símbolo abstrato que representa o tipo da unidade léxica, por exemplo, uma palavra reservada, ou uma sequencia de caracteres denotando um identificador. Os nomes de *tokens* sãos os símbolos processados pela fase seguinte, a analise sintática.

A fase de analise léxica também descarta comentários no código e verifica se os caracteres lidos se encaixam na especificação lexicográfica da linguagem, ou seja, se não existem caracteres inválidos no código.

Uma vez concluída a fase de análise léxica, os tokens são passados para a fase de analise sintática, que tem por objetivo verificar se os *tokens* estão ordenados de forma gramaticalmente correta.

Linguagens de programação geralmente têm sua sintaxe especificada através de gramáticas classificadas por [Chomsky 1956] como livres de contexto. Esse tipo de gramática é menos poderosa do que as gramáticas de linguagens naturais, ou seja, as que são usadas por pessoas para se comunicar. Entretanto são suficientes para descrever a maior parte das linguagens de programação e são simples o suficiente para que um computador possa realizar a analise em um tempo relativamente pequeno, na maior parte dos casos.

A análise semântica de uma linguagem diz respeito ao significado do código e vai muito além da estrutura das frases, e por isso formalizar tais aspectos é bastante difícil. Neste trabalho a formalização da semântica se dará a partir de exemplos e descrições em formato de texto.

3. A linguagem Ves

Para alcançar os objetivos de projetar uma linguagem legível, expressiva e flexível algumas outras linguagens atuais foram usadas como base, provendo não só inpiração para a sintaxe mas também para algumas funcionalidades. Essas linguagens são, Ada, C++, C#, *Dart*, F#, Pascal, *Python*, *Ruby* e *Rust*.

3.1. Especificação léxica

Os arquivos de código fonte da linguagem Ves usam codificação *Unicode UTF-8* por padrão, e atualmente não há uma forma de utilizar outra codificação no arquivo.

Em Ves identificadores são palavras começando com uma letra ou *underscore*, opcionalmente seguidos de uma letra, número, *underscore* ou apóstrofo. Identificadores também podem começar com apóstrofo, nesse caso, o identificador deve ser seguido ao

menos de uma letra, número ou *underscore* mas não pode conter outro apóstrofo. Caracteres *Unicode* também são aceitos como identificadores, ou parte deles.

Assim como virtualmente todas as linguagens de programação, Ves possui um conjunto de identificadores reservados. As palavras reservadas de Ves estão dispostas na Tabela 1, as palavras que aparecem em itálico são reservadas para uso futuro.

as	break	const	continue	*del*	else	except
false	finally	for	*from*	fun	if	*import*
in	*is*	let	loop	match	*mod*	nil
of	priv	prop	prot	pub	raise	return
trait	true	try	type	*use*	var	when
where	while	with	*yield*			

Table 1. Palavras reservadas

Literais são uma forma de denotar valores para constantes de um conjuntos de tipos. Na Figura 1 é possível ver um sumário dos tipos de literais em Ves.

```
/* Literais */
'a', '\n'                       // Caractere
true, false                     // Booleano
0b0001_0111, 0o27, 23, 0x17     // Inteiro
0.32, 6.02e23, .5               // Ponto flutuante
"Hello World!"                  // String
```

Figure 1. Literais em Ves

3.2. Especificação sintática

Para a construção do analisador sintático, foi usada uma ferramenta chamada ANTLR (*Another tool for language recognigtion*). Essa ferramenta lê um arquivo contendo a especificação sintática da linguagem e gera o código fonte do analisador na linguagem alvo, nesse caso *Python*.

O analisador gerado pela ferramenta deriva as regras partindo da regra que define uma unidade de compilação (um arquivo). Esse tipo de analisador é chamado *top-down* pois constrói a árvore de sintática a partir da sua raiz.

O analisador lê o arquivo de código Ves da esquerda para a direita e tenta derivar as produções na mesma ordem, esse tipo de analisador é chamado de LL (*Left-to-right with Leftmost derivation*). Normalmente analisadores desse tipo usam um número fixo de *lookahead*, que é a quantidade de símbolos necessária para distinguir uma regra de produção das outras. O analisador gerado pela ferramenta ANTLR é do tipo LL(*) adaptativo, analisadores desse tipo não possuem *lookahead* fixo, eles usam quantos *tokens* forem necessários para determinar qual regra de produção deve ser escolhida.

Ves permite ligar valores a identificadores de três formas diferentes, eles podem ser uma constante em tempo de compilação, uma ligação imutável ou uma variável. Na Figura 2 é possível ver alguns exemplos dessas declarações.

```
const pi: float = 3.141592;
let name: str = read("Input your name: ");
var evens: [int] = [for i in 0..2..100 -> i];
var y: int = if x < 1 -> 1 else x * 2;
```

Figure 2. Exemplos de declarações em Ves

Ves também possui estruturas de controle de execução e laços, entretanto eles possuem suas particularidades. Como forma de eliminar problemas no comando de alternação *if* e facilitar a leitura do código essas estruturas não possuem uma versão que aceita só um comando, ou seja, em Ves o uso de um bloco delimitado por chaves é obrigatório. Na Figura 3 é possível ver exemplos dessas estruturas.

```
if x > 10 {
    //..
} else {
    //...
}
while n < 10 {
    //...
}
for i in 0..10 {
    //...
}
loop {
    break if x < 23;
}
```

Figure 3. Exemplos de estruturas de controle e laços.

Dentre os comandos de criação de laço destaca-se o comando *loop*. Esse comando foi adaptado da linguagem de programação Ada e facilita a criação de laços infinitos ou com condições complexas de parada. Além disso *loops* são a única estrutura de controle de execução em Ves que podem ser nomeados, dessa forma é possível navegar entre *loops* aninhados como pode ser visto na Figura 4.

```
loop mainLoop {
    loop eventLoop {
        doThis();
        break mainLoop if state == State.Error;
        //...
        DoThat();
        continue if state == State.Retry;
        //...
        finalize();
        break if state == State.Success;
    }
}
```

Figure 4. Exemplo de comando *loop* nomeado

Ves possui também o conceito de funções, elas são denotadas pela palavra reservada *fun* seguida do nome da função, uma lista de parâmetros entre parenteses, opcional-

mente o tipo de retorno e o corpo da função. Quando o tipo de retorno não é informado a função retorna *unit*. Existe também uma sintaxe alternativa para funções cujo corpo consiste somente de um comando, quando esse comando for um *return* a palavra reservada pode ser omitida. Funções podem ser nomeadas ou anonimas, no caso das anonimas anotações de tipo são opcionais. A Figura 5 exemplifica todos os tipos de função aceitos por Ves.

```
fun fact(n: int): int {
    if n <= 1 {
        return 1;
    }
    return n * fact(n - 1);
}

fun fact'(n: int): int = if n <= 1 -> 1 else n * fact'(n-1);

let add1 = fun(x: int): int -> x + 1;
let times2: (int)=>int = fun(x)-> x + 1;
```

Figure 5. Exemplos de funções em Ves

Assim como em muitas outras linguagens, Ves permite ao programador definir novos tipos, que podem ser de duas categorias, registros ou uniões. A Figura 6 apresenta exemplos de declarações de tipos em Ves.

```
type MyBool = True | False ;

type Value =
    | Int of int
    | Float of float
    | Nothing
with {
    fun this.toStr(): str = match this {
            Int(x) -> str(x);
            Float(x) -> str(x);
            Nothing -> ""
    }
}

type Person {
    prop name: str;
    fun this.init(name: str) {
        this.name = name;
    }
}
```

Figure 6. Exemplos de tipos em Ves

Assim como em F#, Ves possui a cláusula *with*, que substitui o ponto e vírgula no final da declaração de uma união e permite a definição de métodos e propriedades que serão compartilhados entre todas as opções definidas anteriormente.

3.3. Especificação semântica

Quando o código de um arquivo é lido pelo interpretador de Ves ele passa pela análise léxica e sintática, uma árvore é então construída, essa árvore é percorrida e um modelo orientado a objetos é construído. Esse modelo guarda as informações de cada declaração para uso nas fases posteriores de análise. Na figura 7 é possível ver o diagrama de classes desse modelo.

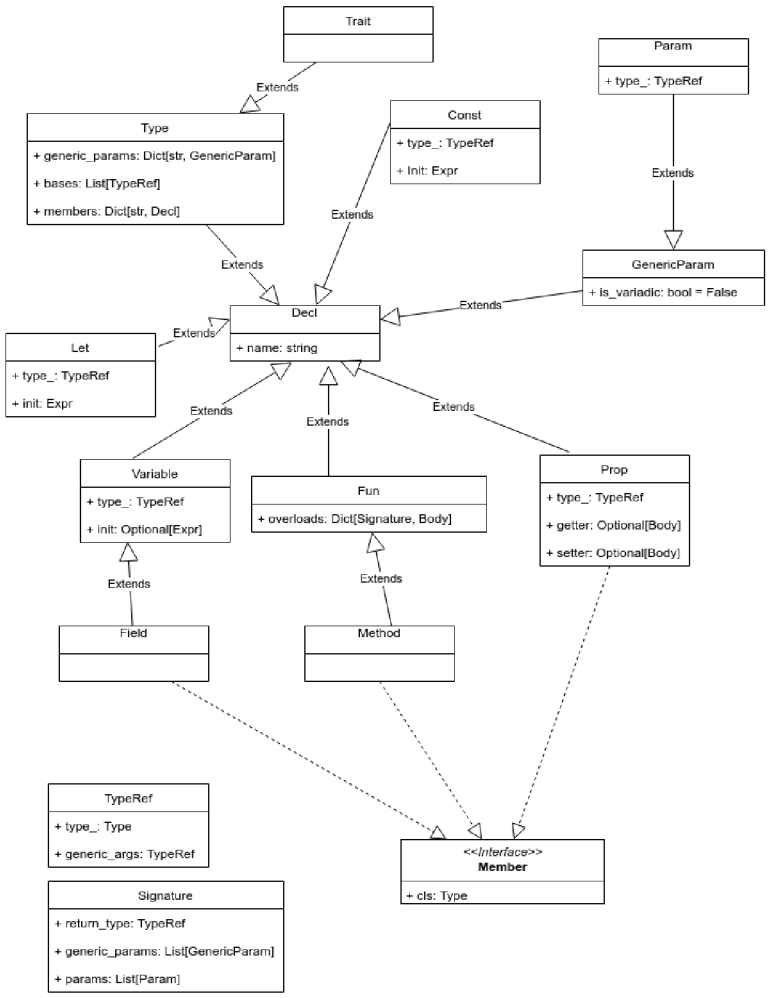

Figure 7. Diagrama de classes do modelo semântico

4. Características de Ves

Ves é uma linguagem interpretada com tipagem estática, mais fracamente tipada do que linguagens como Ada ou F#, mas mais fortemente tipada do que C++, que possui inúmeras conversões implícitas.

Ves é uma linguagem de programação de alto nível, isso quer dizer que as estruturas e conceitos da linguagem abstraem a maior parte das particularidades do *hardware* e sistema operacional no qual o programa está rodando. Abstração também permite de modo bastante simples representar o domínio do problema que o programa se propõe a resolver.

4.1. Legibilidade

Legibilidade diz respeito à facilidade do leitor de reconhecer palavras e estruturas na linguagem, por isso esse aspecto é bastante influenciado por preferencia pessoal e familiaridade com a linguagem, entretanto existem algumas características em Ves que estão presentes com a intenção clara de facilitar a legibilidade.

Em Ves todas as estruturas de controle de execução introduzem um novo escopo delimitado por chaves, essa escolha foi feita , em parte, numa tentativa de evitar enganos por parte do programador, que pode confundir uma estrutura cujo corpo é composto por somente um comando e tentar incluir novos comandos. Em Ves mesmo com a indentação errada é mais difícil cometer erros desse tipo.

Em Ves todas as declarações começam com uma palavra reservada denotando exatamente o que está sendo declarado, isso facilita o entendimento do código e torna a curva de aprendizado da linguagem menos íngreme.

4.2. Expressividade

Expressividade diz respeito a capacidade da linguagem de representar comportamentos ou estruturas de forma simples. Em Ves alguns tipos possuem uma notação especial, isso permite que sejam definidos de forma mais simples. A Figura 8 mostra exemplos de nomes de tipos em Ves.

```
var people: [Person];           // list of people
var grades: Dict[str, int];     // Dictionary
var point: (int, int);          // Tuple
var average: ([float])=>float; // Function
```

Figure 8. Exemplo de nomes de tipos em Ves.

4.3. Flexibilidade

Flexibilidade aqui diz respeito a capacidade da linguagem de se adaptar a diferentes paradigmas e estratégias de resolução de problemas.

Como mostrado anteriormente, Ves possui uma estrutura específica para laços infinitos chamada *loop*. Essa estrutura não é comum nas linguagens de programação atuais, sendo que das citadas como inspiração para Ves, somente Ada possui uma estrutura similar, mostrando que Ves flexibiliza o uso de laços. Na Figura 9 é possível ver um exemplo de uso dessa estrutura, combinada com controle condicional do laço, que seria mais difícil de implementar em outras linguagens.

4.4. Conclusão

Considerando a grande demanda na área de desenvolvimento de *software* e a constante busca por técnicas que permitam maior abstração sem perda de expressividade, a linguagem Ves surge como uma alternativa nesse contexto, permitindo integrar essas características de forma equilibrada.

A base teórica provida pelas linguagens que serviram de inspiração foi essencial para o projeto de Ves, agregando estruturas e conceitos que são importantes para atingir os

```
loop mainLoop {
    loop eventLoop {
        doThis();
        break mainLoop if state == State.Error;
        //...
        DoThat();
        continue if state == State.Retry;
        //...
        finalize();
        break if state == State.Success;
    }
}
```

Figure 9. Exemplo de utilidade de *loop*.

objetivos propostos. Assim, a partir das observações feitas conclui-se que, apesar de estar ainda em estágios iniciais de seu desenvolvimento, Ves cumpre os objetivos aos quais se propõe e pode sim ser usada futuramente para desenvolvimento de software.

Como trabalhos futuros sugere-se a implementação completa do interpretador, ou a implementação de um compilador, para a linguagem, a inclusão de suporte nativo para concorrência, um direcionamento maior da linguagem para ensino.

Quanto à sintaxe, sugere-se a inclusão de herança para registros, suporte para tipos genéricos, a implementação de tipos escalares customizados, interpolação de texto e o enriquecimento do modelo semântico de forma a tornar a linguagem mais robusta.

References

ABES, S. (2018). Histórico do estudo mercado brasileiro de software. `http://www.abessoftware.com.br/dados-do-setor/estudo-2018--dados-2017`. [Online; acessado em 14-Julho-2019].

Aho, A. V. (2003). *Compilers: principles, techniques and tools (for Anna University), 2/e*. Pearson Education India.

Chomsky, N. (1956). Three models for the description of language. *IRE Transactions on information theory*, 2(3):113–124.