UNIVERSIDADE DE CAXIAS DO SUL
ÁREA DO CONHECIMENTO DE CIÊNCIAS EXATAS E ENGENHARIAS

FELIPE MIOTTO

DESENVOLVIMENTO DE UM COMPILADOR E UMA MÁQUINA VIRTUAL DE PYTHON PARA O AMBIENTE WEBALGO

BENTO GONÇALVES
2019

FELIPE MIOTTO

DESENVOLVIMENTO DE UM COMPILADOR E UMA MÁQUINA VIRTUAL DE PYTHON PARA O AMBIENTE WEBALGO

Trabalho de conclusão de curso apresentado como requisito parcial à obtenção do título de Bacharel em Ciência da Computação na Área do Conhecimento de Ciências Exatas e Engenharias da Universidade de Caxias do Sul.

Orientador: Prof. Dr. Ricardo Vargas Dorneles

BENTO GONÇALVES
2019

FELIPE MIOTTO

DESENVOLVIMENTO DE UM COMPILADOR E UMA MÁQUINA VIRTUAL DE PYTHON PARA O AMBIENTE WEBALGO

Trabalho de conclusão de curso apresentado como requisito parcial à obtenção do título de Bacharel em Ciência da Computação na Área do Conhecimento de Ciências Exatas e Engenharias da Universidade de Caxias do Sul.

Aprovado em ___/___/_____

Banca examinadora:

_____
Prof. Dr. Ricardo Vargas Dorneles
Universidade de Caxias do Sul – UCS

_____
Prof. Me. Alexandre Erasmo Krohn Nascimento
Universidade de Caxias do Sul – UCS

_____
Prof. Me. Joacir Giaretta
Universidade de Caxias do Sul – UCS

RESUMO

A linguagem Python vem ganhando cada vez mais espaço no meio acadêmico, devido tanto à produtividade quanto à facilidade na escrita de código providas aos alunos iniciantes em programação de computadores. Este trabalho é o primeiro passo no sentido de munir a ferramenta WebAlgo com o suporte a Python, visando seguir a iminente tendência de aplicação da linguagem às disciplinas introdutórias de programação. O trabalho apresenta importantes conceitos pertinentes à área de estudo, como compiladores, interpretadores e máquinas virtuais, seguido por uma breve introdução à linguagem Python. Após, são apresentados o funcionamento das instruções de baixo nível e o gerenciamento de memória. Ao final, são detalhados os subconjuntos de Python e *bytecode* escolhidos, bem como a gramática implementada e, por fim, as estruturas presentes no *software* implementado.

Palavras-chave: WebAlgo. Python. Compiladores. Interpretadores. Máquinas virtuais.

ABSTRACT

The Python language has been gaining substantial space in academia due to both productivity and ease in code writing it provides to beginning students of computer programming. This work is the first step in order to equip WebAlgo with Python support, aiming to follow the imminent trend of the language application in the introductory programming disciplines. The work presents important concepts pertaining to the area of study, such as compilers, interpreters and virtual machines, followed by a brief introduction of the Python language. Afterwards, the operation of low-level instructions and memory management are discussed. At the end, the Python and bytecode subsets chosen are detailed, as well as the grammar implemented, and, finally, the structures present in the implemented software.

Palavras-chave: WebAlgo. Python. Compilers. Interpreters. Virtual machines.

LISTA DE FIGURAS

LISTA DE SIGLAS

AST	*Abstract Syntax Tree*
CLR	*Common Language Runtime*
CWI	*Centrum Wiskunde & Informatica*
JIT	*Just-in-time*
JVM	*Java Virtual Machine*
LL	*Left-to-Right, Leftmost*
MIT	*Massachusetts Institute of Technology*
PC	*Program Counter*
TDS	Tradução Dirigida por Sintaxe
TOS	*Top of the stack*
TOS1	*Second top-most stack*
UCS	Universidade de Caxias do Sul
USP	Universidade de São Paulo

SUMÁRIO

1 INTRODUÇÃO

Disciplinas introdutórias a programação possuem um grau inerente de dificuldade, resultando num alto nível de evasão e reprovação (BERGIN; REILLY, 2005). Em sala de aula, estudantes iniciantes em programação enfrentam adversidades em comum no aprendizado, tais como: dificuldades para entender o problema proposto; dificuldades para pensar em uma solução para o problema; dificuldades para identificar os passos necessários para a solução do problema; dificuldades de se expressar de forma algorítmica; dificuldades para testar o algoritmo (DORNELES; PICININ JR; ADAMI, 2010). Com o intuito de mitigar esses problemas são de grande valia a construção e a utilização de ferramentas pedagógicas que auxiliem o aluno a superar as dificuldades iniciais.

Estudos vêm sendo feitos com o intuito de desenvolver ferramentas para auxílio do aprendizado de programação, das quais pode-se citar Scratch, Droplet e Repl.it, todas ferramentas online. Scratch foi desenvolvido pelo *Massachusetts Institute of Technology* (MIT) com o foco na faixa etária dos 8 aos 16 anos, onde o aluno pode interagir, através de blocos, com projetos como criação de vídeos musicais, histórias animadas e diversos jogos interativos (MALONEY et al., 2010). Droplet foi desenvolvido pela *Phillips Exeter Academy* e, assim como o Scratch, possui o recurso de arrastar e soltar blocos, porém o mescla com a forma tradicional de escrita de código. Baseia-se numa linguagem real, CoffeeScript, e busca desta forma evitar lacunas no conhecimento do aluno que aprende com ferramentas de arrastar e soltar blocos e posteriormente precisa migrar para o modo tradicional de programação (BAU, 2015). Repl.it é um projeto pessoal fundado por Amjad Masad que possui suporte para diversas linguagens. Essa ferramenta, além de poder ser usada individualmente por programadores, possui suporte para o uso em sala de aula. Nela, o professor pode criar exercícios e compartilhar um link com os alunos, que funciona como um convite para a sala de aula virtual. Há, inclusive, a possibilidade de se criar casos de teste e acompanhar o desempenho individual dos alunos.

Em 2009 foi desenvolvida por professores da Universidade de Caxias do Sul (UCS) uma ferramenta de apoio ao aprendizado de programação chamada WebAlgo, onde atualmente há suporte para as linguagens Português Estruturado e C. Nela, o aluno encontra um acervo de exercícios disponibilizados pelo docente da disciplina e pode submeter sua resolução quantas vezes for necessário, tendo seu código analisado de acordo com casos de teste predefinidos pelo professor, obtendo um retorno adequado ao caso. Para o professor são

oferecidas ferramentas que auxiliam no acompanhamento do progresso de cada aluno dentro da própria ferramenta.

Nos últimos anos uma grande variedade de linguagens de programação foi criada com propósitos diversos. Dentre elas, destaca-se Python, uma linguagem de alto nível, com uma sintaxe simples e fácil de entender, orientada a objetos e de tipagem dinâmica. A linguagem Python vêm sendo cada vez mais adotada em diversos campos, inclusive no meio acadêmico. Visando seguir essa tendência, viu-se necessário o desenvolvimento de uma ferramenta pedagógica que dê aos professores da UCS a possibilidade de aplicar Python a disciplinas relacionadas à programação de computadores.

Ao longo dos anos, várias implementações de Python foram criadas. PyPy, por exemplo, é uma implementação de um subconjunto de Python escrita usando o próprio Python, que possui como foco a velocidade de execução, já que utiliza o conceito de *Just-In-Time* (JIT). Jython, uma implementação da linguagem escrita em Java, mescla funcionalidades do Python com o Java. A implementação oficial de Python é escrita em C, sendo o seu interpretador chamado de CPython. Ao longo de todo este trabalho, quando for citada a linguagem Python, seu interpretador e seu conjunto de instruções, referir-se-á à implementação oficial da versão 3.6 da linguagem.

1.1 PROBLEMA DE PESQUISA

Python é uma linguagem conceitualmente interpretada, onde o código fonte passa pelas análises de *front-end* (análises léxica, sintática e semântica) para após, na etapa de *back-end*, ter seu código intermediário (*bytecode*) executado por uma máquina virtual de pilha (ao contrário de linguagens compiladas, em que o código objeto final tem como alvo uma arquitetura física). Nesse quesito, o tratamento interno do WebAlgo ao Python será semelhante às duas linguagens já existentes na ferramenta, uma vez que o código fonte em C/Português Estruturado é convertido para uma representação intermediária, sendo em seguida executado por uma máquina virtual.

Algumas funcionalidades presentes no Python não existem nas linguagens presentes no WebAlgo. Dentre elas pode-se citar listas, tuplas, conjuntos e dicionários. Adicionalmente, Python trata internamente de maneira diferenciada as representações numéricas: um número inteiro, por exemplo, pode ser arbitrariamente grande, já que é representado por uma estrutura em C, não sendo regido pelo tamanho da palavra da arquitetura da máquina ou do sistema operacional. Tendo em vista essas diferenças, inclusive o fato de Python ser inteiramente

orientado a objetos, foi necessário precaver-se a fim de evitar ao máximo imprevistos no decorrer da implementação.

A documentação de Python é vasta, muito bem detalhada e a implementação original é de código aberto. Decidiu-se a partir disso que, em detrimento de se utilizar a máquina virtual já implementada no WebAlgo, ou a própria *Java Virtual Machine* (JVM) para a execução do *bytecode* gerado, fosse implementado um subconjunto das instruções *bytecode* da máquina virtual do Python, resultando no interpretador (máquina virtual) do código intermediário gerado pelas análises do *front-end*. Isso resultou em maior controle e segurança durante a implementação, já que quaisquer imprevistos que viessem a ocorrer poderiam ser facilmente solucionados consultando o material oficial da linguagem.

1.2 OBJETIVOS DO TRABALHO

O objetivo geral deste trabalho foi o desenvolvimento de um compilador e de uma máquina virtual de Python para o ambiente de ensino WebAlgo da UCS, consistindo nas seguintes etapas:

- Implementação das análises do *front-end* (analisador léxico, sintático e semântico);
- Implementação do *back-end* (máquina virtual).

1.3 ESTRUTURA DO TRABALHO

O trabalho é dividido em três partes principais: uma breve introdução sobre compiladores, interpretadores e máquinas virtuais; uma breve descrição sobre a linguagem Python e seu funcionamento; e o detalhamento do funcionamento e das estruturas internas do *software* implementado.

Na primeira parte são discutidas as principais diferenças entre compiladores e interpretadores, qual o propósito de uso de cada um, bem como suas vantagens e desvantagens. É feita, também, uma explicação sobre máquinas virtuais no contexto de linguagens de programação.

Na segunda parte é discorrido sobre a linguagem Python: sua história e especificações técnicas; geração de código intermediário (*bytecodes*); gerenciamento de memória; e o funcionamento do *Garbage Collector* (coletor de lixo).

Na terceira parte são abordados os assuntos pertinentes à implementação do *software*

proposto. Estudos sobre tópicos abordados em disciplinas introdutórias de programação serviram como base para a definição dos subconjuntos de Python e *bytecode* implementados. A partir das funcionalidades selecionadas para fazerem parte da implementação, foi definida a gramática que serviu como estrutura formal do mecanismo sintático do compilador. Ao final, são tratados assuntos referentes às estruturas internas do *software* implementado.

2 REFERENCIAL TEÓRICO

Os capítulos subsequentes englobam as informações técnicas referentes às áreas de estudo pertinentes à monografia.

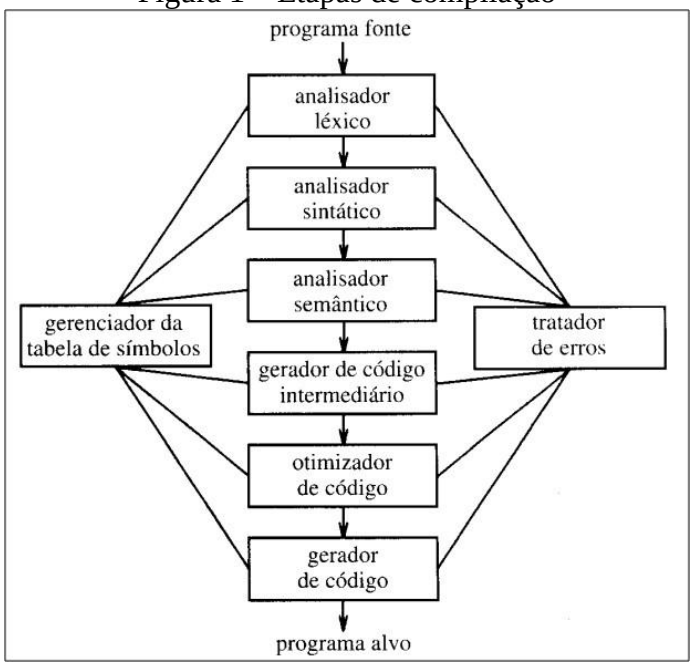
2.1 COMPILADORES E INTERPRETADORES

As linguagens de programação geralmente possuem uma sintaxe projetada para fazer sentido a seres humanos. Entretanto, dispositivos eletrônicos não processam informação de forma similar a qual percebemos a realidade. Em suas estruturas internas, um processador interpreta dados em forma de pulsos elétricos. Os dados que transitam no *hardware* são codificados em notação binária, portanto, a linguagem utilizada por seres humanos para a escrita de programas de computador precisa passar por uma tradução para poder ser entendida pelas estruturas internas do computador. Os artefatos responsáveis por fazer essa tradução de código legível para humanos para código executável pelo *hardware* são chamados de compiladores e interpretadores.

Compiladores são *softwares* que traduzem um código fonte para uma arquitetura alvo. Durante o processo de compilação, o compilador fica encarregado de reportar erros presentes no código fonte. As etapas de compilação de um programa fonte são divididas em análise e síntese. A análise (também chamada de *front-end*) consiste em três etapas: análise léxica (onde o código fonte é fragmentado em *tokens*); análise sintática (onde os *tokens* são verificados por uma implementação de uma gramática que valida a sintaxe do programa); e análise semântica (onde ocorrem as verificações semânticas – variáveis não declaradas ou duplicadas, erro de tipos, etc.). O resultado do *front-end* é geralmente uma estrutura intermediária do código fonte, bem como uma tabela de símbolos (uma estrutura que contém informações sobre o código), que servirão como entrada para a etapa de síntese (também chamada de *back-end*). Na síntese, a estrutura intermediária é convertida para uma representação que faça sentido para a arquitetura alvo – alguns compiladores envolvem uma etapa de otimização antes de gerar código para a arquitetura alvo (AHO et al., 2007). A Figura 1 ilustra as etapas de compilação de um código fonte.

Interpretadores, ao invés de converterem o código fonte para uma representação intermediária ou de baixo nível, simplesmente executam o código intermediário juntamente com os dados de entrada do usuário. Vantagens e desvantagens entre o conceito de compilação e interpretação podem ser constatadas.

Figura 1 – Etapas de compilação



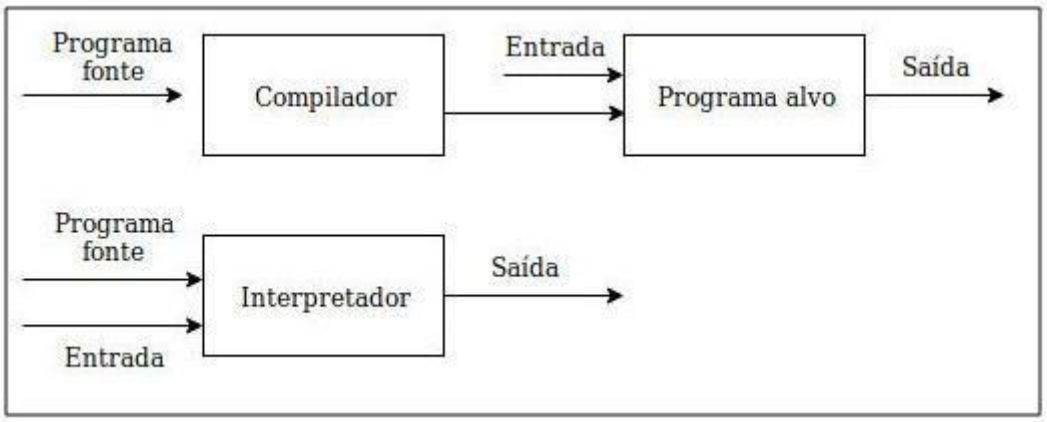
Fonte: AHO; SETHI; ULLMAN, 1995.

Compiladores tornam a execução do código muito mais veloz que interpretadores, já que o código objeto gerado é otimizado para ser executado por uma arquitetura física – ao contrário de interpretadores, em que para uma única instrução a ser executada, várias instruções do processador podem ser exigidas. Em contrapartida, interpretadores permitem melhores diagnósticos de erros, já que executam o código comando a comando.

Tome-se como exemplo uma divisão por zero: a detecção do erro pode ser facilmente feita por uma interpretador, visto que ele terá controle sobre a linha de código onde ocorreu a falha. Já o código objeto gerado por um compilador geralmente não possui informações sobre o código fonte (apesar de ser possível incluir esses dados), fazendo com que uma divisão por zero gere uma interrupção de exceção com informações sobre endereços de memória, tornando a identificação de erros menos amigável do que normalmente seria através do processo de interpretação (MAK, 2009).

O compilador recebe como entrada o programa fonte, analisando-o por completo e convertendo para o programa alvo. O interpretador recebe o programa fonte, executando-o sem converter para uma representação intermediária (Figura 2).

Figura 2 – Processos de compilação e interpretação



Fonte: Adaptado de AHO et al., 2007.

2.2 MÁQUINAS VIRTUAIS

A virtualização vem tendo um papel cada vez mais importante no mundo da computação. Há não muito tempo, a maioria dos *softwares* era projetada para rodar em uma arquitetura física, tornando-os amarrados à plataforma. A portabilidade era dificultada, uma vez que é necessário recompilar (e muitas vezes reescrever) um *software* para torná-lo compatível com uma arquitetura diferente. A necessidade de tornar sistemas portáveis pode ser resolvida ou mitigada com o uso de máquinas virtuais.

Máquinas virtuais, no contexto de linguagens de programação, são implementações de *software* que simulam uma arquitetura física real ou não. Diversas linguagens de programação utilizam essa estratégia para adquirir portabilidade. Como as máquinas virtuais das linguagens de programação possuem uma especificação bem definida, cada sistema operacional pode receber uma implementação da especificação da máquina virtual. O resultado final é uma linguagem de alto nível independente do sistema operacional, já que a execução é feita pela máquina virtual e não pela plataforma. A grande vantagem desse tipo de abordagem é a possibilidade de se escrever uma única versão de código fonte em alto nível e rodá-lo em qualquer plataforma, geralmente sem a necessidade de alterações (SMITH et. al., 2005).

Apesar de algumas implementações de máquinas virtuais serem baseadas em registradores, as principais (como a JVM da Oracle, a *Common Language Runtime* (CLR) da Microsoft e o CPython do Python) são baseadas em pilha – máquinas que utilizam esse tipo de arquitetura são chamadas de máquinas de pilha (CRAIG, 2006). Máquinas de pilha são máquinas abstratas em que as instruções lógicas e aritméticas são efetuadas sobre valores em uma pilha, e os resultados armazenados na mesma. As instruções de uma máquina de pilha

podem ser: instruções aritméticas; instruções de manipulação de pilha; e instruções de fluxo de controle. Uma máquina virtual geralmente opera sobre um conjunto de *bytecodes* (código intermediário) que determinam o que a máquina deve fazer a cada ciclo de execução.

2.3 PYTHON

Python é uma poderosa linguagem de *script*[1] de alto nível, puramente orientada a objetos, *case-sensitive* e que possui uma sintaxe fácil de aprender e entender. Implementa nativamente estruturas de alto nível, isentando o programador de construir suas próprias estruturas, como é comum em linguagens como o C. Possui como objetivo delegar ao programador apenas a lógica da solução do problema, dispensando o foco em detalhes técnicos como estruturas de baixo nível e ponteiros.

A velocidade média de escrita de código é um de seus grandes atrativos. Justamente por possuir o intuito de facilitar o trabalho do programador, frequentemente códigos que levariam horas para serem escritos em outras linguagens, levam minutos para serem desenvolvidos em Python. Por possuir uma sintaxe enxuta, a curva de aprendizado é curta. Segundo seu próprio criador, "uma sintaxe rica mais atrapalha do que ajuda" (198-? ROSSUM apud LESSA, 2000).

Python é derivado da ABC, uma linguagem para fins didáticos da década de 1980. Python nasceu no final de 1989, criado pelo holandês Guido van Rossum na *Centrum Wiskunde & Informatica* (CWI), um instituto de pesquisa em matemática e ciência da computação, na cidade de Amsterdam. Seu nome foi escolhido em homenagem ao seriado britânico de comédia *Monty Python's Flying Circus*, do qual van Rossum é um grande fã.

A execução de Python se dá com o auxílio de um interpretador, possuindo implementações em diversos sistemas operacionais, tornando Python uma linguagem multiplataforma. Por ser interpretado, Python (assim como outras linguagens interpretadas) possui algumas diferenças em relação às linguagens compiladas: como desvantagem há uma execução mais lenta do código final; como vantagem, a linguagem possui uma portabilidade bastante sólida (a maior parte dos programas escritos em uma plataforma poderá ser executada nas demais plataformas) (LESSA, 2000).

[1] Uma das definições sobre linguagens de *script* refere-se à não presença das etapas de compilação-linkedição-carga, fazendo com que o processo de execução do código fonte seja mais simplificado. Frequentemente linguagens de *script* são associadas a linguagens "cola" – utilizadas como integradoras de dois ou mais programas. Adicionalmente, linguagens de *script* são consideradas linguagens que possuem um ciclo rápido de escrita de código (BARRON, 2000).

No princípio Python foi concebido para fins didáticos, porém sua sintaxe simples e legível a tornou uma linguagem bastante apreciada por programadores. Devido à sua similaridade com pseudocódigos (frequentemente utilizados como uma forma de facilitar o entendimento de uma rotina), Python é chamado de "pseudocódigo executável" (VANDERPLAS, 2016).

Antes de se iniciar qualquer descrição do funcionamento de Python, uma nota faz-se necessária: em Python tudo é considerado objeto (tipos escalares, *strings*, funções, classes, instâncias de classes, etc.), e tudo suporta atributos e métodos. Python não funciona de acordo com o modelo clássico de estruturação de memória de linguagens de programação, onde uma variável refere-se a uma posição física de memória. Variáveis em Python possuem uma relação diferente com os valores, e o equivalente de "variável x recebe valor y" pode ser considerado, em Python, como "variável x recebe uma referência ao objeto que armazena o valor y". A fins de simplificação, ao se falar que uma variável recebe um determinado valor, ficar-se-á subentendido que internamente a variável recebe uma referência ao objeto que contém aquele valor, a menos que alguma explicação adicional se faça necessária.

Neste trabalho não será feita uma descrição das funcionalidades básicas de Python, podendo esse tipo de material ser encontrado de forma bastante detalhada e completa na obra de Vanderplas (2016).

2.3.1 *Bytecode*

Assim como diversas linguagens de programação como Java e C#, Python também é executado por uma máquina virtual de pilha. Para que um código fonte seja executado pela máquina virtual, faz-se necessária a execução das etapas de *front-end* e *back-end*, onde a partir do código fonte é gerada uma representação intermediária, denominada *bytecode*.

A máquina virtual do Python opera sobre um conjunto de *bytecodes* que determinam o que a máquina deve fazer a cada ciclo de execução. A implementação oficial de Python implementa mais de 100 instruções *bytecode* (119 na versão 3.6), das quais apenas um subconjunto será implementado nesse trabalho (o subconjunto será discutido mais adiante no texto).

Internamente Python representa as instruções *bytecode* como uma sequência de *bytes* em memória. Pode-se verificar a estrutura dessa sequência digitando apenas alguns comandos no terminal (modo interativo). Para um melhor entendimento, pode-se representar a sequência de *bytes* de uma maneira um pouco mais amigável, utilizando a função "dis", nativa do

Python. A função “dis” toma como entrada uma função qualquer em Python, imprimindo as instruções *bytecode* equivalentes de uma forma perfeitamente legível. A Figura 3 ilustra a sequência de *bytecodes* gerados para a função “foo” (através do comando “foo.__code__.co_code”), e a representação das instruções *bytecode* de uma maneira mais legível através do comando “dis.dis(foo)”.

Figura 3 – Instruções *bytecode* da função “foo”

```
>>> def foo():
...     a = 42
...     b = 7
...     c = a + b
...     print("Lorem ipsum")
...
>>> foo.__code__.co_code
b'd\x01}\x00d\x02}\x01|\x00|\x01\x17\x00}\x02t\x00d\x03\x83\x01\x01\x00d\x00S\x00'
>>> dis.dis(foo)
  2           0 LOAD_CONST               1 (42)
              2 STORE_FAST               0 (a)

  3           4 LOAD_CONST               2 (7)
              6 STORE_FAST               1 (b)

  4           8 LOAD_FAST                0 (a)
             10 LOAD_FAST                1 (b)
             12 BINARY_ADD
             14 STORE_FAST               2 (c)

  5          16 LOAD_GLOBAL              0 (print)
             18 LOAD_CONST               3 ('Lorem ipsum')
             20 CALL_FUNCTION            1
             22 POP_TOP
             24 LOAD_CONST               0 (None)
             26 RETURN_VALUE
```

Fonte: Próprio autor.

Pode-se interpretar o código desmontado da Figura 3 da seguinte maneira: os números da primeira coluna referem-se à linha equivalente no código da função “foo”; os números da segunda coluna são os índices das instruções *bytecode* (cada *bytecode* ocupa dois *bytes*, fazendo com que a próxima instrução possua o índice dois *bytes* após o atual); a terceira coluna é a instrução *bytecode*; a quarta coluna, quando presente, é o argumento (índice) para a instrução; a quinta coluna é uma “dica” do que a instrução significa (KAPTUR, 201-?). Algumas instruções-chave da Figura 3 podem ser descritas da seguinte forma:

- LOAD_CONST: empilha uma constante com base no índice;
- STORE_FAST: desempilha o elemento e o armazena na variável;
- LOAD_FAST: empilha o valor da variável com base no índice;

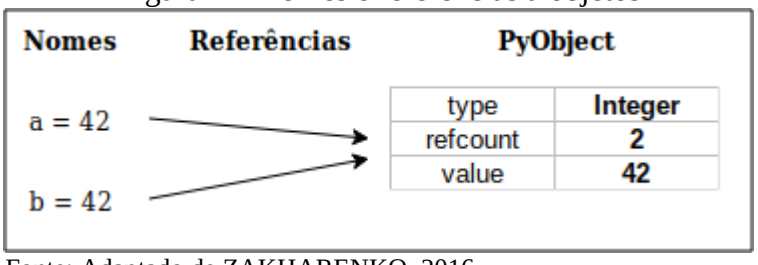
- BINARY_ADD: desempilha o topo e o subtopo, soma-os e empilha o resultado;
- CALL_FUNCTION: efetua uma chamada de função;
- POP_TOP: desempilha um elemento e descarta-o;
- RETURN_VALUE: retorna o topo da pilha para o contexto *caller*.

2.3.2 Gerenciamento de memória

O interpretador do CPython possui um gerenciador de memória para um eficiente gerenciamento dos objetos e estruturas em memória de um programa em execução. Juntamente com o sistema operacional, o gerenciador de memória manipula uma estrutura *heap* para o armazenamento de dados de processos que rodam sobre a máquina virtual do Python. Cabe somente à máquina virtual gerenciar a memória, não sendo possível que o programador interfira na estrutura interna da área *heap* (PYTHON SOFTWARE FOUNDATION, 2019a).

Diferentemente de linguagens como C e Pascal, onde as variáveis possuem tipo e tamanho fixos armazenados em memória, Python é uma linguagem dinamicamente tipada, onde nomes podem representar diferentes tipos de dados durante a execução do programa. Quando uma atribuição de valores é feita através do operador de atribuição (“=”), uma ligação é feita entre o nome e o objeto (PYTHON SOFTWARE FOUNDATION, 2019a). Cada objeto em memória geralmente armazena seu tipo, seu valor e seu contador de referências (ZAKHARENKO, 2016). A Figura 4 ilustra a relação entre nomes (variáveis), referências e objetos.

Figura 4 – Nomes e referências a objetos



Fonte: Adaptado de ZAKHARENKO, 2016.

Nomes são rótulos para objetos em memória, onde cada objeto pode ser referenciado por vários nomes. Outra diferença em relação a linguagens como C e Pascal é o relacionamento das variáveis com seu respectivo valor. Enquanto em C e Pascal uma variável

é o nome de uma estrutura fixa em memória, em Python uma variável aponta para uma estrutura de tipo e tamanho mutáveis, não possuindo um objeto amarrado a ela: ao mudar o valor de uma variável, a mesma passa a apontar para outro objeto/endereço (ZAKHARENKO, 2016). Conforme pode ser observado na Figura 5, a função "id" retorna o endereço do objeto na memória. Pode-se notar, nas duas primeiras atribuições, que "a" e "b" referenciam a mesma posição de memória. O rótulo "a" passa a referenciar outra posição de memória ao ter seu valor alterado.

Figura 5 – Objetos em memória

```
>>> a = 42
>>> id(a)
139900061292896
>>> b = 42
>>> id(b)
139900061292896
>>> a = 7
>>> id(a)
139900061291776
```

Fonte: Próprio autor.

Conforme visto na Figura 5, variáveis podem apontar para o mesmo objeto. Entretanto, essa característica demanda alguns cuidados por parte do programador. Ao atualizar o valor de um objeto, todas as variáveis que apontam para ele serão afetadas. A Figura 6 ilustra o processo onde primeiramente atribui-se a "x" uma lista. Em seguida "y" aponta para o mesmo objeto que "x". Após, adiciona-se um elemento à lista através da função "append". Ao verificar o conteúdo de "y", percebe-se que os efeitos aplicam-se também a ele.

Figura 6 – Comportamento de listas

```
>>> x = [1, 2, 3]
>>> y = x
>>> x.append(4)
>>> x
[1, 2, 3, 4]
>>> y
[1, 2, 3, 4]
```

Fonte: Adaptado de VANDERPLAS, 2016.

Cabe salientar que nem todas as variáveis com o mesmo valor apontam para o mesmo objeto em memória. Para números inteiros, por exemplo, a faixa de valores em que

duas variáveis com o mesmo valor referenciam o mesmo objeto é de -5 até 256. Para valores fora dessa faixa, mesmo variáveis com o mesmo valor apontam para objetos diferentes na memória (PYTHON SOFTWARE FOUNDATION, 2019b), conforme ilustra a Figura 7.

Figura 7 – Referência à faixa de -5 até 256

```
>>> a = -5
>>> b = -5
>>> a is b # isto é, 'a' aponta para o mesmo objeto que 'b'?
True
>>> a = -6
>>> b = -6
>>> a is b # isto é, 'a' aponta para o mesmo objeto que 'b'?
False
>>>
>>> a = 256
>>> b = 256
>>> a is b # isto é, 'a' aponta para o mesmo objeto que 'b'?
True
>>> a = 257
>>> b = 257
>>> a is b # isto é, 'a' aponta para o mesmo objeto que 'b'?
False
```

Fonte: Próprio autor.

2.3.3 *Garbage Collector*

A linguagem Python possui um gerenciador de memória denominado *Garbage Collector* (coletor de lixo, em tradução literal), cuja função consiste em desalocar objetos desnecessários da memória. De maneira resumida, o coletor de lixo verifica, para cada objeto em memória, seu contador de referências. Quando o objeto possuir nenhuma referência a ele, o coletor de lixo desaloca-o da memória, liberando o espaço. A Figura 8 ilustra o conceito de desalocação de objetos baseada em um contador de referências (ZAKHARENKO, 2016).

Figura 8 – Referências a objetos

```
var1 = Obj()        # objeto Obj possui uma referência
var2 = var1         # objeto Obj possui duas referências
var1 = AnotherObj() # objetos Obj e AnotherObj possuem uma referência cada
var2 = var1         # objeto AnotherObj possui duas referências;
                    # objeto Obj possui nenhuma referência
                    # objeto Obj será desalocada pelo coletor de lixo
```

Fonte: Adaptado de PYTHON BRASIL, 2008.

O coletor de lixo possui algumas desvantagens, como *overhead* de espaço (já que para armazenar o contador de referências é necessário armazená-lo juntamente com cada objeto) e *overhead* de processamento (a cada ação que afeta o objeto, faz-se necessário o incremento ou decremento do contador de referências). Além dessas desvantagens, o coletor de lixo possui o ônus de não ser totalmente seguro quando se trabalha com *threads* – duas *threads* que tentam alterar o contador de referências ao mesmo tempo podem ocasionar problemas (ZAKHARENKO, 2016).

3 ESPECIFICAÇÕES DO PROJETO

As seções subsequentes discutem os assuntos pertinentes à implementação do trabalho, como os subconjuntos de Python e *bytecode*s que foram utilizados, bem como a gramática implementada para validação sintática do código fonte informado pelo usuário.

3.1 SUBCONJUNTO DE PYTHON

Apesar de Python oferecer ao programador um enorme número de funcionalidades, apenas algumas foram contempladas na implementação do presente trabalho, tendo em vista a limitação de tempo imposta para elaboração de um trabalho de conclusão. Outro motivo que favoreceu a seleção de apenas algumas funcionalidades da linguagem é o propósito de uso do WebAlgo, isto é, o auxílio do ensino introdutório de programação, fazendo com que algumas funcionalidades de Python fujam do escopo do projeto. Deste modo, o subconjunto implementado ficou limitado a funcionalidades essenciais ao ensino introdutório de programação, podendo, após esse trabalho de conclusão, ser acrescido em funcionalidades.

Toda a construção do *software* foi baseada o máximo possível na documentação oficial de Python (uma documentação bastante completa), tendo fornecido uma relativa segurança no decorrer do andamento do projeto.

Disciplinas introdutórias de programação geralmente abordam conceitos relativamente previsíveis, podendo haver algumas variações: variáveis e constantes; entrada e saída de dados; operadores aritméticos, de atribuição, relacionais e lógicos; estruturas de condição; estruturas de repetição; vetores e matrizes.

Em 2016, a Universidade de São Paulo (USP) disponibilizou um curso *online* chamado “Introdução à Ciência da Computação com Python”, direcionado principalmente, porém não exclusivamente, aos alunos de graduação da USP. O projeto foi uma iniciativa da Pró-Reitoria de Pesquisa da USP junto ao Departamento de Ciência da Computação do Instituto de Matemática e Estatística, ministrado por Fabio Kon, professor titular do Departamento de Ciência da Computação da Universidade de São Paulo, nele sendo abordados os seguintes assuntos pertinentes ao ensino introdutório de programação (NAOE, 2016):

- Variáveis (inteiros, *floats*, *strings* e booleanos);
- Operadores aritméticos, de atribuição, relacionais e lógicos;
- Entrada e saída de dados;

- Estruturas de condição e repetição;
- Funções;
- Vetores (listas);
- Matrizes (listas de listas);
- Módulos (importação de códigos externos);
- Programação orientada a objetos.

A companhia de ensino americana *Codecademy* disponibiliza um curso introdutório a programação em Python abordando os seguintes tópicos:

- Variáveis (inteiros, *floats*, *strings* e booleanos);
- Entrada e saída de dados;
- Estruturas de condição e repetição;
- Funções;
- Listas e dicionários;
- Operadores aritméticos, de atribuição, relacionais e lógicos;
- Classes;
- Entrada e saída de arquivos.

De acordo com informações obtidas junto à professora Maria de Fatima Webber do Prado Lima, que ministra na UCS disciplinas introdutórias de programação utilizando Python, é comum tratar-se nesse tipo de disciplina os seguintes tópicos:

- Variáveis, constantes, tipos de dados;
- Entrada e saída de dados;
- Operadores e expressões: operador de atribuição, operadores aritméticos, relacionais e lógicos;
- Estruturas de condição;
- Estruturas de repetição;
- Vetores e matrizes;
- Bibliotecas para geração de gráficos;
- Manipulação de arquivos.

Com base nos três casos, o subconjunto implementado foi:

- Variáveis simples: inteiros, *floats*, *strings*, booleanos e *NoneType*;

- Variáveis container: listas e tuplas;
- Métodos de lista: *append* e *len*;
- Operadores aritméticos: adição, subtração, multiplicação, divisão, divisão inteira, módulo, exponenciação;
- Operadores aritméticos reduzidos (“+=”, “-=”, “*=”, “/=”, “%=”, “**=” e “//=”);
- Operadores relacionais: igualdade, diferença, menor que, maior que, menor ou igual a, maior ou igual a;
- Operadores booleanos: *and*, *or* e *not*;
- Operadores *prose-like* (*is*, *is not*, *in*, *not in*);
- Entrada de dados: função *input*;
- Saída de dados: função *print*;
- Estruturas de condição: *if*, *elif* e *else*;
- Estruturas de repetição: *for* e *while*;
- Sentença *break*;
- Funções;
- Iterador *range*;
- Funções para conversão de tipos: *int()*, *float()* e *str()*;

Não foram incluídas bibliotecas para geração de gráficos e manipulação de arquivos devido à sua inerente dificuldade de implementação. Além disso, a implementação de classes, dicionários e conjuntos não foi contemplada. Tais funcionalidades não foram inclusas no escopo do projeto devido ao fato de o trabalho em si ter apresentado um grau elevado de complexidade por contemplar tanto as etapas de compilação quanto de interpretação. Contudo, cabe o destaque de que adições de funcionalidades são aplicáveis em futuras implementações.

3.2 SUBCONJUNTO DE *BYTECODES*

Conforme descrito anteriormente, todo o código fonte passa por algumas etapas de compilação para ser convertido em instruções *bytecode* que, por fim, são executadas pela máquina virtual (a execução do *bytecode* é frequentemente chamada de interpretação). Na implementação da ferramenta deste trabalho de conclusão de curso, o tratamento do código

fonte é feito de forma similar ao da implementação oficial de Python, ou seja, o código fonte é convertido para instruções *bytecode* equivalentes, e após é feita a execução através de uma máquina virtual.

Como a máquina virtual utilizada para a interpretação das instruções de baixo nível teve de ser implementada, isto é, não foi utilizada algum tipo de máquina virtual já existente, se fez necessária a definição de um subconjunto de instruções *bytecode* a ser implementado.

Assim como descrito anteriormente, na definição do subconjunto de Python, a implementação de todo o conjunto de *bytecodes* da versão oficial é inviável para um trabalho de conclusão de curso. Considerando, também, que a partir do subconjunto de Python previamente definido, somente algumas instruções *bytecode* são necessárias para uma representação fiel entre *bytecode* e código fonte, a implementação de um subconjunto estritamente necessário para a correta execução do código fonte mostrou-se conveniente.

Tendo como base o subconjunto de Python previamente citado, o subconjunto de instruções *bytecode* implementado foi – TOS é uma abreviação de "topo da pilha" (*top of the stack*) e TOS1 é uma abreviação de "subtopo da pilha" (*second top-most stack*):

- LOAD_CONST: empilha uma constante;
- LOAD_NAME: empilha o valor de uma variável global (no contexto global);
- LOAD_FAST: empilha o valor de uma variável local (no contexto de uma função);
- LOAD_GLOBAL: empilha o valor de uma variável global;
- STORE_NAME: desempilha e armazena o valor em uma variável global (no contexto global);
- STORE_FAST: desempilha e armazena o valor em uma variável local (no contexto de uma função);
- BINARY_ADD: implementa TOS = TOS1 + TOS;
- BINARY_SUBTRACT: implementa TOS = TOS1 – TOS;
- BINARY_TRUE_DIVIDE: implementa TOS = TOS1 / TOS;
- BINARY_MULTIPLY: implementa TOS = TOS1 * TOS;
- BINARY_FLOOR_DIVIDE: implementa TOS = TOS1 // TOS;
- BINARY_POWER: implementa TOS = TOS1 ** TOS;
- INPLACE_ADD: implementa TOS = TOS1 + TOS;
- INPLACE_SUBTRACT: implementa TOS = TOS1 – TOS;
- INPLACE_TRUE_DIVIDE: implementa TOS = TOS1 / TOS;

- INPLACE_MULTIPLY: implementa TOS = TOS1 * TOS;
- INPLACE_FLOOR_DIVIDE: implementa TOS = TOS1 // TOS;
- INPLACE_POWER: implementa TOS = TOS1 ** TOS;
- COMPARE_OP: operador de comparação;
- UNARY_POSITIVE: implementa TOS = +TOS;
- UNARY_NEGATIVE: imeplementa TOS = -TOS;
- POP_JUMP_IF_FALSE: se TOS é falso, realiza um salto para outra instrução da lista de instruções *bytecode*;
- JUMP_FORWARD: salto relativo para outra instrução da lista de instruções *bytecode*;
- JUMP_ABSOLUTE: salto absoluto para outra instrução da lista de instruções *bytecode*;
- SETUP_LOOP: empilha um bloco referente a um *loop*;
- GET_ITER: torna TOS um iterável;
- FOR_ITER: assumindo que TOS possui um objeto iterável, empilha o próximo elemento;
- POP_BLOCK: remove um bloco da pilha;
- POP_TOP: remove e descarta TOS;
- BUILD_LIST: desempilha elementos e cria uma lista;
- BINARY_SUBSCR: implementa TOS = TOS1[TOS];
- BUILD_SLICE: aplica um *slice* em uma lista empilhando o resultado;
- MAKE_FUNCTION: realiza os tratamentos necessários para construção de uma função, empilhando o objeto necessário para execução da mesma.
- CALL_FUNCTION: realiza a chamada da função que está em TOS;
- RETURN_VALUE: retorna o topo da pilha ao context *caller*.

A definição do subconjunto previamente definido baseou-se numa pesquisa sistemática sobre quais instruções *bytecode* implementam as funcionalidades do subconjunto de Python definido para este trabalho. Utilizando o comando nativo “dis”, é possível verificar como determinado bloco de código é mapeado para as instruções *bytecode*. Com o uso dessa ferramenta, cada funcionalidade do subconjunto de Python teve sua estrutura de baixo nível identificada e inserida no subconjunto de *bytecodes.*

As instruções *LOAD_CONST* e *LOAD_GLOBAL* podem ser verificadas sempre que

uma constante local e um objeto global, respectivamente, são empilhados. As instruções *LOAD_FAST* e *STORE_FAST* também são comumente utilizadas, já que se tratam, respectivamente, do empilhamento de valores de variáveis locais e do armazenamento de valores em variáveis locais.

Em Python, mesmo quando uma função não tem seu retorno explicitado, internamente Python retorna um objeto do tipo *NoneType*. Através do *bytecode* de funções, abordado adiante, verificar-se-á nas últimas instruções o empilhamento de uma variável *None* e o seu retorno através da instrução *RETURN_VALUE*.

Pode-se verificar através da Figura 9 que valores de qualquer tipo simples (inteiros, *floats* e *strings*) são manipulados de maneira idêntica, sem o uso de instruções específicas para determinado tipo. A instrução *LOAD_CONST* empilha o número 42 e *STORE_FAST* armazena-o na variável "a". Ao fim do bloco verifica-se o retorno de um *None* através da instrução *RETURN_VALUE*.

Figura 9 – Instruções *LOAD_CONST*, *LOAD_FAST* e *RETURN_VALUE*

```
>>> def foo():
...     a = 42
...     b = 3.14
...     c = "Lorem ipsum"
...
>>> dis.dis(foo)
  2           0 LOAD_CONST               1 (42)
              2 STORE_FAST               0 (a)

  3           4 LOAD_CONST               2 (3.14)
              6 STORE_FAST               1 (b)

  4           8 LOAD_CONST               3 ('Lorem ipsum')
             10 STORE_FAST               2 (c)
             12 LOAD_CONST               0 (None)
             14 RETURN_VALUE
```

Fonte: Próprio autor.

A instrução *LOAD_GLOBAL* é utilizada tanto para empilhar uma variável global quanto o objeto de uma função, onde, no segundo caso, faz-se necessário o uso da instrução *CALL_FUNCTION* para a execução do procedimento. Verifica-se na Figura 10 que após *CALL_FUNCTION* há um valor que indica o número de parâmetros que a função irá receber. A instrução *CALL_FUNCTION* faz com que sejam desempilhados os argumentos, mais a chamada da função em si, onde após a execução da mesma, o retorno é colocado na pilha.

Figura 10 – Instruções *LOAD_GLOBAL* e *CALL_FUNCTION*

```
>>> def foo():
...     a = upper("lorem ipsum")
...     b = lower("LOREM IPSUM")
...
>>> dis.dis(foo)
  2           0 LOAD_GLOBAL              0 (upper)
              2 LOAD_CONST               1 ('lorem ipsum')
              4 CALL_FUNCTION            1
              6 STORE_FAST               0 (a)

  3           8 LOAD_GLOBAL              1 (lower)
             10 LOAD_CONST               2 ('LOREM IPSUM')
             12 CALL_FUNCTION            1
             14 STORE_FAST               1 (b)
             16 LOAD_CONST               0 (None)
             18 RETURN_VALUE
```

Fonte: Próprio autor.

As instruções aritméticas (*BINARY_ADD*, *BINARY_SUBTRACT*, *BINARY_TRUE_DIVIDE*, *BINARY_FLOOR_DIVIDE*, *BINARY_MULTIPLY* e *BINARY_POWER*) possuem funcionamento semelhante entre si: desempilham TOS e TOS1, aplicando a operação específica de cada instrução, empilhando o resultado (Figura 11).

Figura 11 – Instruções aritméticas

```
>>> def foo():
...     a = 7
...     b = 42
...     c = a + b
...     d = a - b
...     e = a * b
...     f = a / b
...     g = a // b
...     h = a % b
...     i = a ** b
...
>>> dis.dis(foo)
  2           0 LOAD_CONST               1 (7)
              2 STORE_FAST               0 (a)

  3           4 LOAD_CONST               2 (42)
              6 STORE_FAST               1 (b)

  4           8 LOAD_FAST                0 (a)
             10 LOAD_FAST                1 (b)
             12 BINARY_ADD
             14 STORE_FAST               2 (c)

  5          16 LOAD_FAST                0 (a)
             18 LOAD_FAST                1 (b)
             20 BINARY_SUBTRACT
             22 STORE_FAST               3 (d)

  6          24 LOAD_FAST                0 (a)
             26 LOAD_FAST                1 (b)
             28 BINARY_MULTIPLY
             30 STORE_FAST               4 (e)

  7          32 LOAD_FAST                0 (a)
             34 LOAD_FAST                1 (b)
             36 BINARY_TRUE_DIVIDE
             38 STORE_FAST               5 (f)

  8          40 LOAD_FAST                0 (a)
             42 LOAD_FAST                1 (b)
             44 BINARY_FLOOR_DIVIDE
             46 STORE_FAST               6 (g)

  9          48 LOAD_FAST                0 (a)
             50 LOAD_FAST                1 (b)
             52 BINARY_MODULO
             54 STORE_FAST               7 (h)

 10          56 LOAD_FAST                0 (a)
             58 LOAD_FAST                1 (b)
             60 BINARY_POWER
             62 STORE_FAST               8 (i)
             64 LOAD_CONST               0 (None)
             66 RETURN_VALUE
```

Fonte: Próprio autor.

As instruções de operadores aritméticos reduzidos (*INPLACE_ADD*, *INPLACE_SUBTRACT*, *INPLACE_TRUE_DIVIDE*, *INPLACE_FLOOR_DIVIDE*, *INPLACE_MULTIPLY* e *INPLACE_POWER*), diferem das instruções aritméticas usuais somente no nível de gerenciamento de memória. Em termos práticos, possuem funcionamento

e aplicações semelhantes às instruções aritméticas: desempilham TOS e TOS1, empilhando o resultado.

As instruções *COMPARE_OP*, *POP_JUMP_IF_FALSE* e *JUMP_FORWARD*, são verificadas ao se desmontar códigos com as estruturas de condição (*if*, *elif* e *else*). *COMPARE_OP* toma como parâmetro qualquer operador relacional, desempilhando TOS e TOS1, comparando e empilhando o resultado (verdadeiro ou falso). A instrução *POP_JUMP_IF_FALSE* define o *bytecode counter* (que possui função similar ao *program counter* ou PC da arquitetura de computadores clássica) para determinado valor caso o TOS seja falso. A instrução *JUMP_FORWARD* incrementa o *bytecode counter* em "n", onde "n" é o valor do parâmetro (Figura 12).

Figura 12 – Instruções de estruturas de condição

```
>>> def foo():
...     a = 7
...     b = 42
...     if a == b:
...             c = 0
...     elif a != b:
...             c = 1
...     elif a < b:
...             c = 2
...     elif a > b:
...             c = 3
...     elif a <= b:
...             c = 4
...     elif a >= b:
...             c = 5
...     else:
...             c = 6
...
>>> dis.dis(foo)
  2           0 LOAD_CONST               1 (7)
              2 STORE_FAST               0 (a)

  3           4 LOAD_CONST               2 (42)
              6 STORE_FAST               1 (b)

  4           8 LOAD_FAST                0 (a)
             10 LOAD_FAST                1 (b)
             12 COMPARE_OP               2 (==)
             14 POP_JUMP_IF_FALSE       22

  5          16 LOAD_CONST               3 (0)
             18 STORE_FAST               2 (c)
             20 JUMP_FORWARD            74 (to 96)

  6     >>   22 LOAD_FAST                0 (a)
             24 LOAD_FAST                1 (b)
             26 COMPARE_OP               3 (!=)
             28 POP_JUMP_IF_FALSE       36

  7          30 LOAD_CONST               4 (1)
             32 STORE_FAST               2 (c)
             34 JUMP_FORWARD            60 (to 96)

  8     >>   36 LOAD_FAST                0 (a)
             38 LOAD_FAST                1 (b)
             40 COMPARE_OP               0 (<)
             42 POP_JUMP_IF_FALSE       50

  9          44 LOAD_CONST               5 (2)
             46 STORE_FAST               2 (c)
             48 JUMP_FORWARD            46 (to 96)

 10     >>   50 LOAD_FAST                0 (a)
             52 LOAD_FAST                1 (b)
             54 COMPARE_OP               4 (>)
             56 POP_JUMP_IF_FALSE       64

 11          58 LOAD_CONST               6 (3)
             60 STORE_FAST               2 (c)
             62 JUMP_FORWARD            32 (to 96)

 12     >>   64 LOAD_FAST                0 (a)
             66 LOAD_FAST                1 (b)
             68 COMPARE_OP               1 (<=)
             70 POP_JUMP_IF_FALSE       78

 13          72 LOAD_CONST               7 (4)
             74 STORE_FAST               2 (c)
             76 JUMP_FORWARD            18 (to 96)

 14     >>   78 LOAD_FAST                0 (a)
             80 LOAD_FAST                1 (b)
             82 COMPARE_OP               5 (>=)
             84 POP_JUMP_IF_FALSE       92

 15          86 LOAD_CONST               8 (5)
             88 STORE_FAST               2 (c)
             90 JUMP_FORWARD             4 (to 96)

 17     >>   92 LOAD_CONST               9 (6)
             94 STORE_FAST               2 (c)
        >>   96 LOAD_CONST               0 (None)
             98 RETURN_VALUE
```

Fonte: Próprio autor.

As instruções *BUILD_LIST*, *BINARY_SUBSCR* e *POP_TOP* são verificadas ao se desmontar operações sobre listas (operadores *prose-like* e indexação de listas). A instrução *BUILD_LIST* desempilha "n" elementos, sendo "n" o parâmetro da instrução, construindo uma lista, empilhando-a. A instrução *BINARY_SUBSCR* é verificada quando faz-se um acesso a um elemento da lista, através da indexação, desempilhando TOS e TOS1, executando

TOS1[TOS], e empilhando o resultado. A instrução *POP_TOP* descarta o TOS quando o mesmo não é mais necessário (Figura 13).

Figura 13 – Instruções *BUILD_LIST*, *BINARY_SUBSCR* e *POP_TOP*

```
>>> def foo():
...     a = [1, 2, 3]
...     b = [1, 2, 3]
...     c = a[0]
...     a is b
...     a is not b
...     1 in a
...     1 not in a
...
```

```
>>> dis.dis(foo)
  2     0 LOAD_CONST      1 (1)
        3 LOAD_CONST      2 (2)
        6 LOAD_CONST      3 (3)
        9 BUILD_LIST      3
       12 STORE_FAST      0 (a)

  3    15 LOAD_CONST      1 (1)
       18 LOAD_CONST      2 (2)
       21 LOAD_CONST      3 (3)
       24 BUILD_LIST      3
       27 STORE_FAST      1 (b)

  4    30 LOAD_FAST       0 (a)
       33 LOAD_CONST      4 (0)
       36 BINARY_SUBSCR
       37 STORE_FAST      2 (c)

  5    40 LOAD_FAST       0 (a)
       43 LOAD_FAST       1 (b)
       46 COMPARE_OP      8 (is)
       49 POP_TOP

  6    50 LOAD_FAST       0 (a)
       53 LOAD_FAST       1 (b)
       56 COMPARE_OP      9 (is not)
       59 POP_TOP

  7    60 LOAD_CONST      1 (1)
       63 LOAD_FAST       0 (a)
       66 COMPARE_OP      6 (in)
       69 POP_TOP

  8    70 LOAD_CONST      1 (1)
       73 LOAD_FAST       0 (a)
       76 COMPARE_OP      7 (not in)
       79 POP_TOP
       80 LOAD_CONST      0 (None)
       83 RETURN_VALUE
```

Fonte: Próprio autor.

As instruções *SETUP_LOOP*, *GET_ITER*, *FOR_ITER*, *JUMP_ABSOLUTE* e *POP_BLOCK* são verificadas ao se desmontar o comando de repetição *for*. A instrução *SETUP_LOOP* define que o bloco de repetição se estende da instrução atual até “n” *bytes*, onde “n” é o parâmetro. A instrução *GET_ITER* desempilha o TOS, torna-o um objeto iterável, empilhando-o. A instrução *FOR_ITER* toma o TOS, que é um objeto iterável, acessando seu próximo elemento, empilhando-o. A instrução *JUMP_ABSOLUTE* define o *bytecode counter* em “n”, sendo “n” o parâmetro da instrução. A instrução *POP_BLOCK* remove todo o bloco da pilha, baseando-se na prévia definição de *SETUP_LOOP* (Figura 14).

Figura 14 – Instruções de estruturas de repetição

```
>>> def foo():
...     for n in range(10):
...             print(n)
...
>>> dis.dis(foo)
  2           0 SETUP_LOOP              24 (to 26)
              2 LOAD_GLOBAL              0 (range)
              4 LOAD_CONST               1 (10)
              6 CALL_FUNCTION            1
              8 GET_ITER
        >>   10 FOR_ITER                12 (to 24)
             12 STORE_FAST               0 (n)

  3          14 LOAD_GLOBAL              1 (print)
             16 LOAD_FAST                0 (n)
             18 CALL_FUNCTION            1
             20 POP_TOP
             22 JUMP_ABSOLUTE           10
        >>   24 POP_BLOCK
        >>   26 LOAD_CONST               0 (None)
             28 RETURN_VALUE
```

Fonte: Próprio autor.

A instrução *BREAK_LOOP* é verificada ao se desmontar um comando de repetição contendo um comando *break*, onde a instrução é seguida pela instrução *JUMP_ABSOLUTE*, resultando na quebra do laço (Figura 15).

Figura 15 – Instrução *BREAK_LOOP*

```
>>> def foo():
...     for n in range(10):
...             print(n)
...             if n == 5:
...                     break
...
>>> dis.dis(foo)
  2           0 SETUP_LOOP              34 (to 36)
              2 LOAD_GLOBAL              0 (range)
              4 LOAD_CONST               1 (10)
              6 CALL_FUNCTION            1
              8 GET_ITER
        >>   10 FOR_ITER                22 (to 34)
             12 STORE_FAST               0 (n)

  3          14 LOAD_GLOBAL              1 (print)
             16 LOAD_FAST                0 (n)
             18 CALL_FUNCTION            1
             20 POP_TOP

  4          22 LOAD_FAST                0 (n)
             24 LOAD_CONST               2 (5)
             26 COMPARE_OP               2 (==)
             28 POP_JUMP_IF_FALSE       10

  5          30 BREAK_LOOP
             32 JUMP_ABSOLUTE           10
        >>   34 POP_BLOCK
        >>   36 LOAD_CONST               0 (None)
             38 RETURN_VALUE
```

Fonte: Próprio autor.

Os comandos *while*, *continue*, *print* e *input* não resultam em nenhuma adição em relação às instruções previamente detalhadas.

3.3 DEFINIÇÃO DA GRAMÁTICA

Uma das etapas de compilação é a análise sintática que, a partir de uma gramática, verifica se a sequência de *tokens* obedece às regras sintáticas da linguagem. A implementação oficial de Python implementa uma gramática LL(1)[2] (IKE-NWOSU, 2017), com 85 produções no total.

[2] Uma gramática LL(1) pode ser implementada por um *parser* descendente recursivo, onde o primeiro "L" refere-se à entrada ser analisada da esquerda (L) para a direita; o segundo "L" refere-se à derivação mais à esquerda (L); e o "1" refere-se à análise de um *token* por vez para ser feita a verificação de qual produção deve-se utilizar (AHO, 2007).

Assim como as definições dos subconjuntos de Python e *bytecode* foram limitadas segundo um critério, a gramática implementada contemplou apenas as funcionalidades condizentes com o subconjunto de Python. Com base na gramática oficial de Python, que pode ser encontrada na documentação oficial (PYTHON SOFTWARE FOUNDATION, 2019c), tendo sido realizadas algumas adaptações, a gramática implementada foi:

- *file_input*: (NEWLINE | **stmt**)* ENDMARKER
- *funcdef*: 'def' NAME **parameters** ':' **suite**
- *parameters*: '(' [**typedargslist**] ')'
- *typedargslist*: **tfpdef** [',' **resto_typedargslist**]
- *resto_typedargslist*: **tfpdef** [',' **resto_typedargslist**] | vazio
- *tfpdef*: NAME
- *stmt*: **simple_stmt** | **compound_stmt**
- *simple_stmt*: **small_stmt** (';' **small_stmt**)* [';'] NEWLINE
- *small_stmt*: **expr_stmt** | **print_stmt | flow_stmt**
- ex*pr_stmt*: **testlist_star_expr** (**augassign testlist** | '=' **resto_expr_stmt**)
- *resto_expr_stmt*: **testlist_star_expr** '=' **resto_expr_stmt | vazio**
- *print_stmt*: 'print' '(' **test** (',' **test**)* ')'
- *testlist_star_expr*: **test** [',' **resto_testlist_star_expr**] [',']
- *resto_testlist_star_expr*: **test** [',' **resto_testlist_star_expr**]
- *augassign*: '+=' | '-=' | '*=' | '/=' | '%=' | '**=' | '//='
- *flow_stmt*: 'break' | 'continue' | 'return' [**testlist**]
- *compound_stmt*: **if_stmt** | **while_stmt** | **for_stmt** | **funcdef**
- *if_stmt*: 'if' **test** ':' **suite** ('elif' **test** ':' **suite**)* ['else' ':' **suite**]
- *while_stmt*: 'while' **test** ':' **suite** ['else' ':' **suite**]
- *for_stmt*: 'for' **exprlist** 'in' **testlist** ':' **suite** ['else' ':' **suite**]
- *suite*: **simple_stmt** | NEWLINE INDENT **stmt**+ DEDENT
- *test*: **or_test**
- *or_test*: **and_test** ('or' **and_test**)*
- *and_test*: **not_test** ('and' **not_test**)*
- *not_test*: 'not' **not_test** | **comparison**
- *comparison*: **expr** (**comp_op expr**)*
- *comp_op*: '<'|'>'|'=='|'>='|'<='|'<>'|'!='|'in'|'not' 'in'|'is'|'is' 'not'

- *expr*: **term** (('+'|'-') **term**)*
- *term*: **factor** (('*'|'/'|'%'|'//') **factor**)*
- *factor*: ('+'|'-'|'~') **factor** | **power**
- *power*: **atom_expr** ['**' **factor**]
- *atom_expr*: **atom trailer***
- *atom*: '(' [**testlist**] ')' | '[' [**testlist**] ']' | NAME | NUMBER | STRING+ | 'None' | 'True' | 'False'
- *trailer*: '(' [**arglist**] ')' | '[' **subscript** ']' | '.' NAME
- *subscript*: **test** | [**test**] ':' [**test**] [**sliceop**]
- *sliceop*: ':' [**test**]
- *exprlist*: **expr** [',' **resto_exprlist**] [',']
- *resto_exprlist*: **expr** [',' **resto_exprlist**]
- *testlist*: **test** [',' **resto_testlist**] [',']
- *resto_testlist*: **test** [',' **resto_testlist**]
- *arglist*: **argument** [',' **resto_arglist**] [',']
- *resto_arglist*: **argument** [',' **resto_arglist**]
- *argument*: **test**

Onde: nomes em itálico são símbolos não-terminais utilizados no lado esquerdo das produções em que são definidos; nomes em negrito são símbolos não-terminais utilizados no lado direito das produções; nomes entre apóstrofos são símbolos terminais; *pipes* separam produções pertinentes ao mesmo não-terminal; asteriscos após uma sentença entre parênteses significam a operação de fecho, isto é, zero ou mais ocorrências da sentença; o símbolo de soma após a sentença entre parênteses significa fecho positivo, isto é, uma ou mais ocorrências da sentença; sentenças entre colchetes são sentenças opcionais; e símbolos com todos os caracteres em maiúsculo são *tokens* especiais utilizados internamente pelo interpretador de Python. É importante salientar que a gramática previamente definida está em notação de expressão regular.

A produção **file_input** é o símbolo inicial da gramática, a partir do qual toda a árvore de derivação será gerada. A produção **funcdef** é a parte da gramática onde toda a estrutura de uma declaração de função será montada. As produções **parameters** e **typedargslist** possuem relação direta com a possível sequência de parâmetros que uma declaração de função pode ter. Situação semelhante à produção **tfpdef**, que relaciona-se diretamente com **typedargslist**,

sendo um dos blocos de montagem de declarações de funções.

A produção **stmt** deriva duas importantes produções da gramática: **simple_stmt** e **compound_stmt**. A produção **simple_stmt** deriva **small_stmt** (onde são geradas as expressões de todos os tipos) e **flow_stmt** (que direciona a geração para os comandos *break*, *continue* e *return*). As produções **if_stmt**, **while_stmt** e **for_stmt** servem de formalismo sintático para os comandos *if-elif-else*, *while* e *for*, respectivamente.

A gramática definida é completa o suficiente para abranger grande parte dos cenários possíveis utilizando-se o subconjunto previamente apresentado. Dentre as funcionalidades não inclusas na gramática deste trabalho estão os comandos *import* (importação de módulos/códigos externos), *async* e *wait* (utilizados para programação assíncrona), *del* (utilizado para deletar referências a objetos), *raise* e *exception* (para tratamento de exceções), *yield* (retorno especial de funções), anotação de funções (explicitação de quais tipos de parâmetro e retorno uma função terá) e os parâmetros **args* e ***kwargs* (um tipo especial de passagem de argumentos para uma função). Optou-se por não incluir tais funcionalidades devido aos seus propósitos de utilização não serem condizentes com os assuntos abordados em disciplinas introdutórias de programação.

Do subconjunto implementado, de uma forma geral, as funcionalidades não possuem limitações. Entretanto, alguns comandos possuem restrições quando comparados à implementação oficial de Python. As funcionalidades implementadas, no que tange à sua restrição de uso, podem ser descritas da seguinte forma:

- Variáveis simples: uso irrestrito;
- Listas: uso irrestrito;
- Métodos *append* e *len*: uso irrestrito;
- Operadores aritméticos: uso irrestrito;
- Operadores aritméticos reduzidos: uso irrestrito;
- Operadores relacionais: uso irrestrito;
- Operadores booleanos: uso irrestrito;
- Função *input*: uso irrestrito;
- Função *print*: aceita somente um argumento (necessária a concatenação de elementos caso queria-se imprimir vários dados);
- Estruturas de condição *if*, *elif*, e *else*: uso irrestrito;
- Estrutura *for*: uso irrestrito, exceto pela impossibilidade de inserção dos comandos *if* ou *while* em seu corpo;

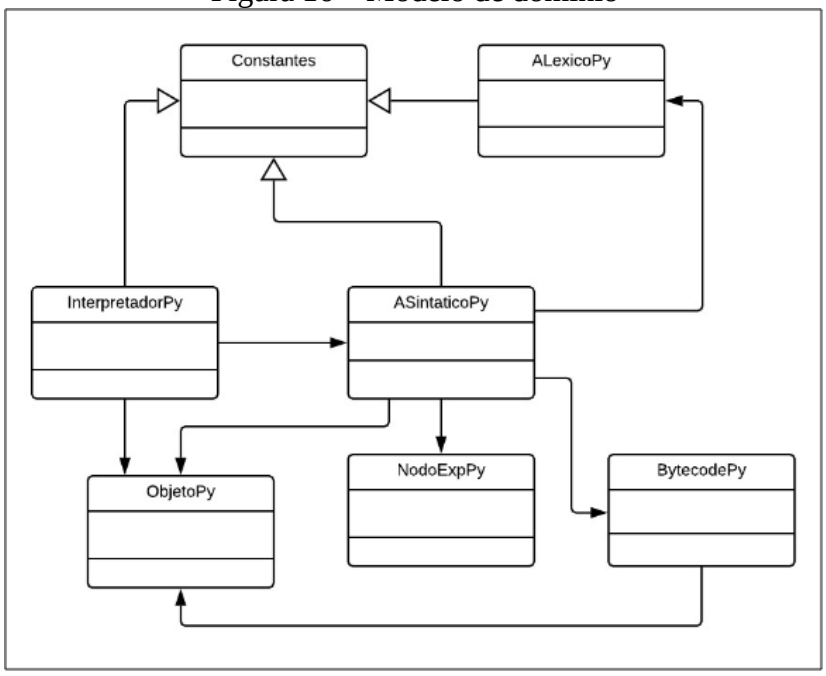
- Estrutura *while*: uso irrestrito, exceto pela impossibilidade de inserção dos comandos *if* ou *for* em seu corpo;
- Sentença *break*: uso irrestrito;
- Funções: uso irrestrito, exceto pela impossibilidade de utilização de recursividade e do retorno explícito de valores;
- Iterador *range*: uso irrestrito, exceto pela impossibilidade da inserção de expressões como argumentos (são aceitos somente números inteiros);
- Funções *int()*, *float()* e *str()*: uso irrestrito.

4 DESENVOLVIMENTO

Tendo em consideração os assuntos abordados nos capítulos anteriores, as funcionalidades pertinentes foram desenvolvidas de modo complementar ao WebAlgo. Alterações pontuais tiverem de ser aplicadas, mas, na maioria dos aspectos, foi possível seguir o escopo proposto no primeiro trabalho de conclusão. O desenvolvimento dividiu-se em duas etapas principais: o desenvolvimento do *front-end* (análises léxica, sintática, semântica e geração de instruções *bytecode*) e o desenvolvimento do *back-end* (máquina virtual).

Ao todo foram acrescentadas seis novas classes ao WebAlgo. Todas as classes serão abordadas em detalhes ao longo dessa seção, onde serão feitos os detalhamentos de seus atributos. A Figura 16 ilustra o modelo de domínio correspondente a essas classes, com a adição da classe **Constantes**, discutida adiante.

Figura 16 – Modelo de domínio



Fonte: Próprio autor.

4.1 DESENVOLVIMENTO DO *FRONT-END*

O *front-end* foi construído com o uso de duas novas classes, **ALexicoPy**, onde está codificado o analisador léxico e **ASintaticoPy**, onde estão codificados os analisadores sintático e semântico e o gerador de instruções *bytecode*.

4.1.1 Analisador léxico

O analisador léxico recebe como entrada uma *string* contendo todo o código fonte em Python informado pelo usuário. Em seguida, o código fonte é analisado por uma implementação de um autômato finito determinístico, onde os *tokens* são requisitados sob demanda pelo analisador sintático. Caso o código informado pelo usuário contenha inconsistências léxicas, um erro é emitido pelo analisador. Todos os *tokens* necessários para atender o subconjunto definido da linguagem são suportados pelo analisador. Para a definição de constantes equivalentes a cada *token*, a classe **Constantes**, já existente no WebAlgo, foi reaproveitada (herdada) considerando o fato de que a maioria dos *tokens* utilizados na linguagem C também estão presentes na linguagem Python.

Como editor do código fonte foi utilizada a classe **RSyntaxArea**, classe já presente no WebAlgo, sendo este o editor utilizado tanto para o compilador de Português Estruturado quanto o da linguagem C. Neste projeto, alguns ajustes nos atributos do objeto de **RSyntaxArea** foram realizados para que o editor funcionasse de uma forma mais adequada à linguagem Python.

4.1.2 Analisadores sintático e semântico

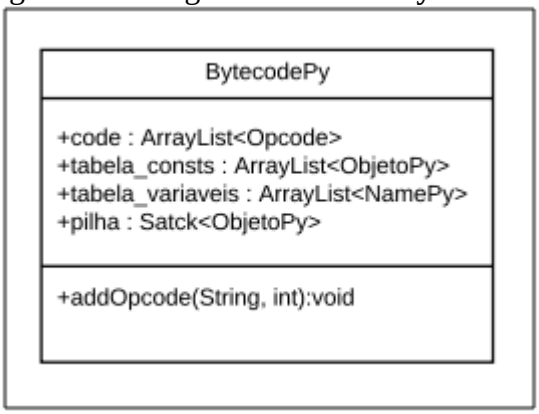
Os analisadores sintático e semântico foram desenvolvidos em conjunto, na mesma classe (**ASintaticoPy**). Toda a gramática proposta foi implementada, tendo sido utilizado um método em Java para representar cada produção. O processo de reconhecimento sintático inicia na produção inicial (**file_input**), onde a partir desse ponto é feito o reconhecimento de todo o programa. Sempre que há a necessidade de obter o próximo *token*, é chamado um método da classe **ALexicoPy** que devolve o próximo elemento do código fonte. Durante pontos específicos em cada método, foram adicionadas funcionalidades para emissão de mensagens de erros, construção de estruturas pertinentes à geração de código intermediário e detecção de erros semânticos.

Para armazenamento das estruturas necessárias para a interpretação do programa pela máquina virtual foi criada a classe **BytecodePy**. Fazem parte da lista de atributos um *ArrayList* de **Opcode** que armazena as instruções *bytecode* (cada objeto de **Opcode** armazena uma instrução) e um objeto do tipo *Stack* de **ObjetoPy**, que faz o papel da pilha de execução utilizada pela máquina virtual (a classe **ObjetoPy** é discutida em detalhes ainda nesta seção). Além disso, outros dois objetos de *ArrayList* constituem os atributos dessa classe: a tabela de

variáveis (*ArrayList* de **NamePy**, que estende de **ObjetoPy**), responsável por armazenar todas as variáveis reconhecidas pelo analisador sintático; e a tabela de constantes (*ArrayList* de **ObjetoPy**), que armazena todas as constantes identificadas no reconhecimento do código fonte. Construtores e outros métodos simples fazem parte do conjunto de métodos implementados, entretanto, não são dignos de nota.

A Figura 17 ilustra o diagrama de classes da classe **BytecodePy**. No que tange às classes **ObjetoPy** e **NamePy**, por ora, basta o conhecimento de que objetos de ambas as classes representam objetos em memória. As duas classes serão detalhadas mais adiante, ainda nesta seção. A classe **Opcode** apenas armazena uma *string* e um inteiro, servindo para armazenamento de instruções *bytecode* individuais. A fim de simplificação, sua presença foi omitida do diagrama.

Figura 17 – Diagrama da classe **BytecodePy**



Fonte: Próprio autor.

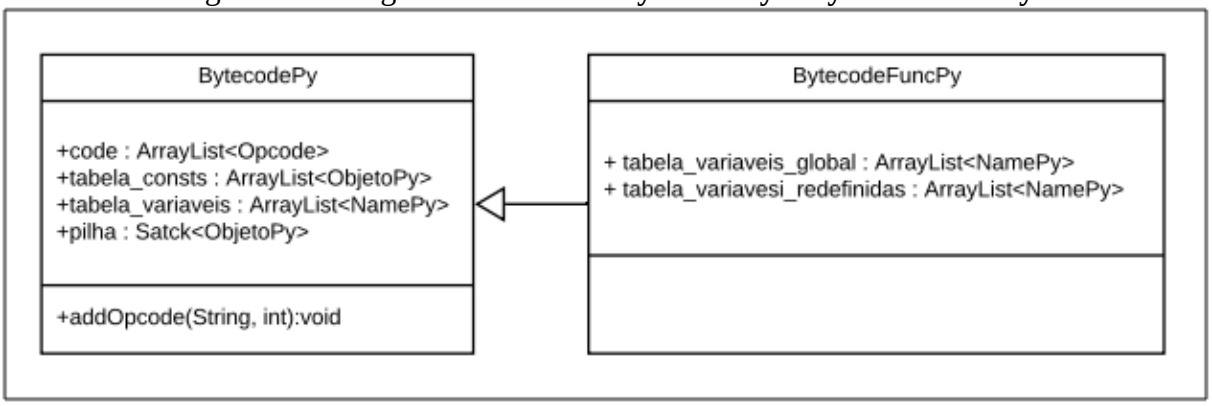
Já no início do processo de reconhecimento, ao chamar o primeiro método da gramática, é passado por referência um objeto da classe **BytecodePy**, responsável por armazenar tanto as instruções *bytecode* quanto as estruturas necessárias à execução do programa.

Na chamada do método da produção **funcdef**, onde definições de funções são reconhecidas, um novo objeto da classe **BytecodeFuncPy** (que herda de **BytecodePy)** é criado e passado por referência. Esse novo objeto é responsável por armazenar todas as instruções *bytecode* da função, bem como as tabelas de variáveis e constantes. Visando ser o mais fiel possível ao comportamento da implementação oficial de Python, o objeto *bytecode* de funções é preenchido de forma independente do objeto *bytecode* do contexto global, resultando em um único objeto *bytecode* para o contexto global e outro para cada definição de função. Em outras palavras, para cada função definida, há um objeto de **BytecodeFuncPy**

onde as informações ficam armazenadas de forma independente entre si e em relação ao objeto *bytecode* do contexto global. Isso facilita tanto o processo de compilação quanto o de interpretação, pois tem-se as funções e o contexto global estruturados de forma separada, podendo ser gerados e executados sem interferência mútua.

Os objetos de **BytecodeFuncPy** que representam funções, conforme dito anteriormente, herdam da classe **BytecodePy**, onde foi feita a adição de duas novas estruturas: um *ArrayList* de **NamePy** que mantém uma cópia das variáveis globais que serão utilizadas naquela função (necessário para a máquina virtual) e outro *ArrayList* (também de **NamePy**) responsável por armazenar uma lista com todas as variáveis redefinidas naquela função, ou seja, variáveis que serão tratadas como locais (informação necessária apenas no reconhecedor sintático/semântico, para geração de instruções *bytecode*). A Figura 18 ilustra os atributos e relacionamentos entre as classes **BytecodePy** e **BytecodeFuncPy**.

Figura 18 – Diagrama das classes **BytecodePy** e **BytecodeFuncPy**



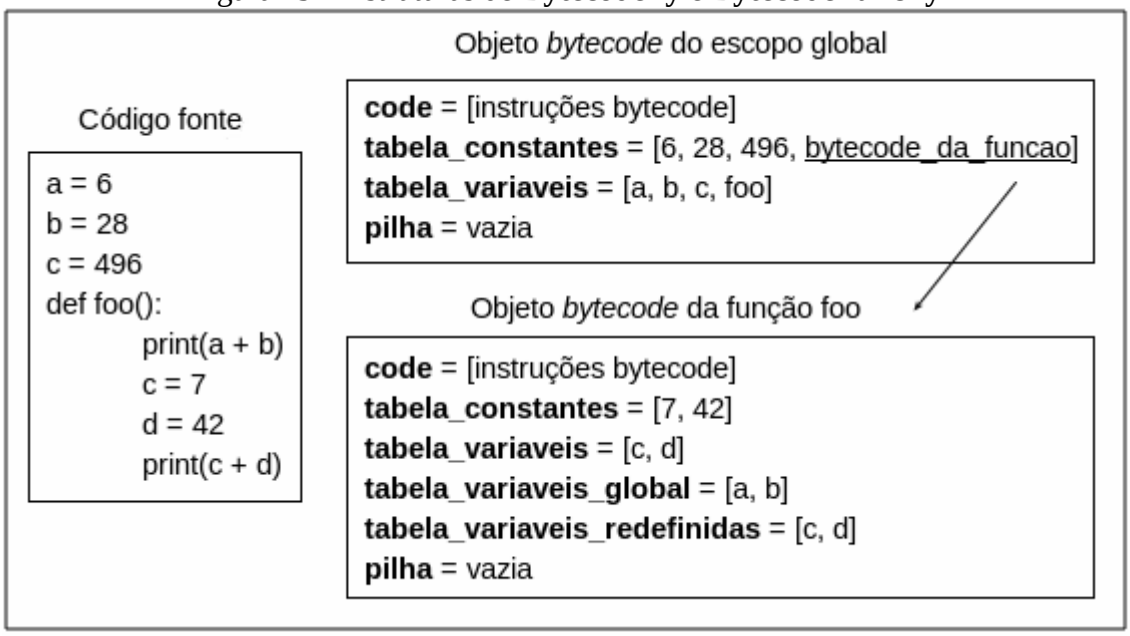
Fonte: Próprio autor.

Com base nas informações dos dois novos atributos da classe **BytecodeFuncPy**, as instruções *bytecode* são geradas conforme tem-se a necessidade de acesso a uma variável local ou global. Na geração de instruções *bytecode*, uma verificação é feita, sendo gerada a instrução *LOAD_FAST* caso queira-se o acesso a uma variável local, ou *LOAD_GLOBAL* caso esteja-se acessando uma variável global.

Ao término do reconhecimento sintático da função, o objeto *bytecode* gerado (objeto da classe **BytecodeFuncPy**) é armazenado na tabela de constantes do escopo em que a função foi definida (presente no objeto da classe **BytecodePy**), comportamento similar à implementação oficial de Python. A Figura 19 ilustra as estruturas resultantes de um código fonte qualquer, onde pode-se identificar as estruturas anteriormente citadas. Percebe-se que o objeto *bytecode* da função é armazenado na tabela de constantes do objeto *bytecode* do

contexto global. Além disso, é possível verificar que, no objeto *bytecode* da função, duas novas estruturas estão presentes, a tabela de variáveis globais e a tabela de variáveis redefinidas. As pilhas de cada objeto serão contextualizadas na Seção 4.2. A fim de simplificação, a constante *None* foi ocultada da tabela de constantes.

Figura 19 – Estruturas de **BytecodePy** e **BytecodeFuncPy**

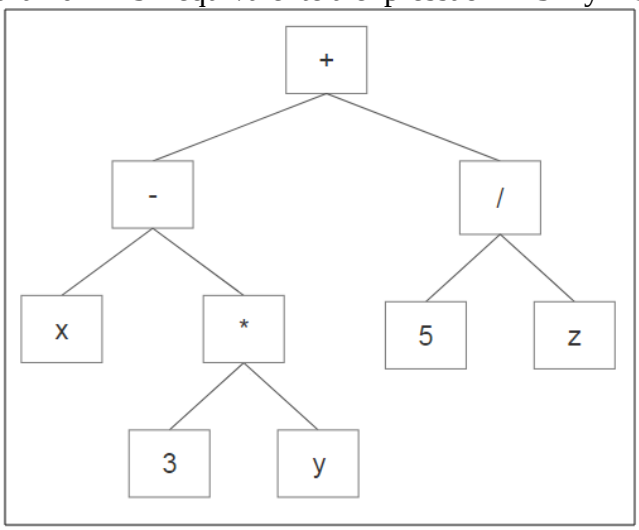


Fonte: Próprio autor.

Para representação de expressões e de trechos de código onde há a possibilidade de derivação de símbolos terminais (identificadores, inteiros, *floats* ou *strings*) foi utilizado o conceito de árvore sintática abstrata, ou *Abstract Syntax Tree* (AST), onde cada nodo é representado pela classe **NodoExpPy**. Conforme a expressão é reconhecida, são criados os nodos correspondentes a cada elemento da expressão.

A classe **NodoExpPy** tem o papel de representar nodos de expressões, onde cada nodo pode representar tanto um operando quanto um operador: constantes e variáveis como operandos; operadores aritméticos, reduzidos, relacionais, booleanos, etc., como operadores. Os dois principais atributos da classe **NodoExpPy** são os nodos filhos da esquerda e da direita, responsáveis pela estruturação da árvore. Ambos os filhos são objetos dessa mesma classe, **NodoExpPy**. Além destes, outros atributos estão presentes: um inteiro para controle do tipo daquele nodo (caso esteja sendo representado um nodo de soma, de multiplicação, de um *token*, etc.); uma *string* para armazenamento do valor do nodo (caso seja um nodo de uma variável, por exemplo); e um inteiro utilizado como identificador único para aquele nodo. A Figura 20 ilustra a AST gerada para uma expressão qualquer.

Figura 20 – AST equivalente à expressão x – 3 * y + 5 / z

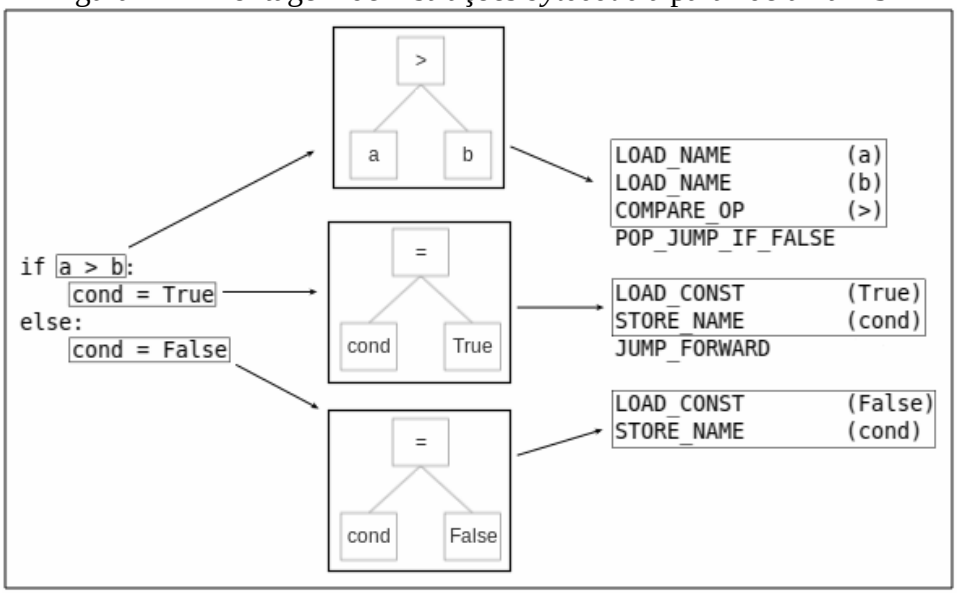


Fonte: Próprio autor.

O uso do conceito de AST foi adotado devido à complexidade apresentada pela geração de instruções *bytecode* diretamente no reconhecedor sintático. Desta forma, sempre que há a montagem do objeto *bytecode* (passo executado sempre ao final do reconhecimento sintático/semântico), ao se encontrar um nodo da AST, um método para conversão é chamado, onde passa-se a AST e recebe-se um objeto de **BytecodePy**.

Ao final da montagem do objeto *bytecode*, têm-se uma estrutura com todas as instruções *bytecode* daquele código, inclusive as instruções geradas a partir das estruturas AST. A Figura 21 ilustra a montagem das instruções *bytecode* de um comando condicional *if*, tendo ciência de que a condição e o corpo do *if* e do *else* são representados por uma AST. As duas instruções não destacadas são geradas de forma direta no analisador sintático/semântico, sem o auxílio de uma AST.

Figura 21 – Montagem de instruções *bytecode* a partir de uma AST

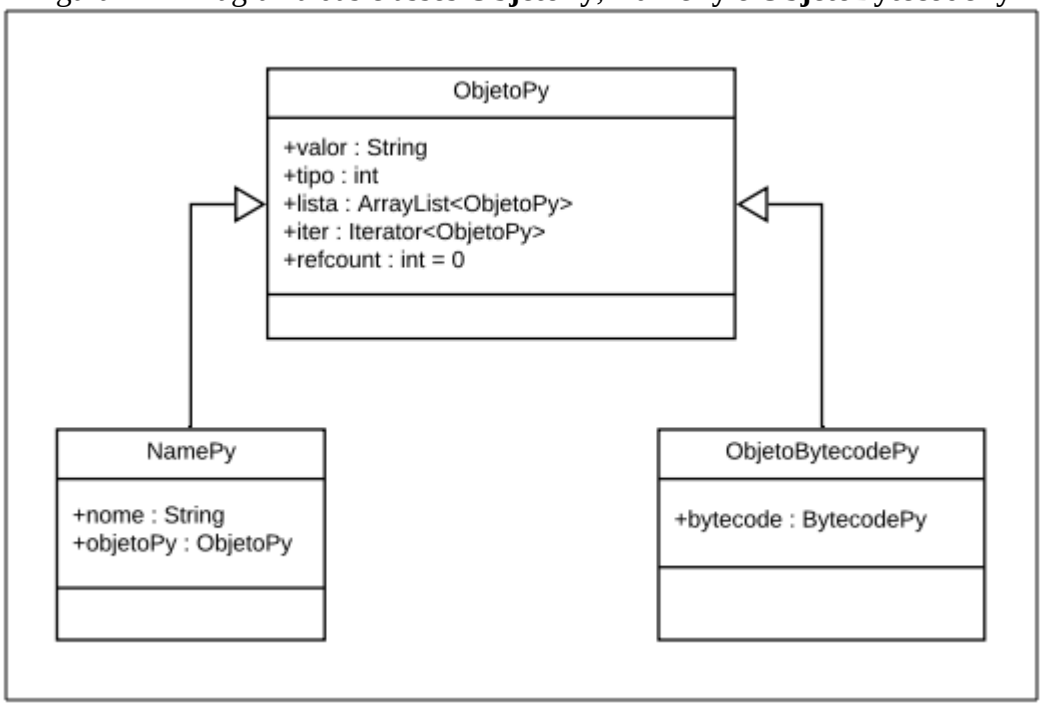


Fonte: Próprio autor.

Baseando-se no funcionamento da implementação oficial de Python, onde todos os dados são representados por objetos, foi criada a classe **ObjetoPy**, responsável por armazenar as estruturas pertinentes aos objetos resultantes do código fonte. Os dois principais atributos dessa classe são uma *string* para armazenamento do lexema referente ao objeto (caso o objeto tiver), e um inteiro, responsável por armazenar a constante que identifica o tipo do objeto (inteiro, variável, lista, etc.). Além desses atributos, um *ArrayList* existe para os casos em que um objeto tenha a possibilidade de possuir dois ou mais valores, como é o caso de listas e tuplas. Por fim, destaca-se um atributo inteiro, que funciona como contador de referências, necessário para o correto funcionamento do *Garbage Collector*, que será discutido na Seção 4.2.1.

Duas classes herdam de **ObjetoPy**: as classes **NamePy** e **ObjetoBytecodePy**. A classe **NamePy** é responsável por representar variáveis. Já a classe **ObjetoBytecodePy** é utilizada para representação de objetos *bytecode* nas tabelas de constantes. Essa abordagem foi escolhida devido ao fato de em Python todo tipo de dado ser representado como um objeto. Desta forma, tanto variáveis (nomes) quanto objetos *bytecode* em nível de execução herdam de **ObjetoPy**, mantendo um único tipo de objeto como padrão durante a interpretação. A Figura 22 ilustra a relação entre as classes **ObjetoPy**, **NamePy** e **ObjetoBytecodePy**.

Figura 22 – Diagrama das classes **ObjetoPy**, **NamePy** e **ObjetoBytecodePy**

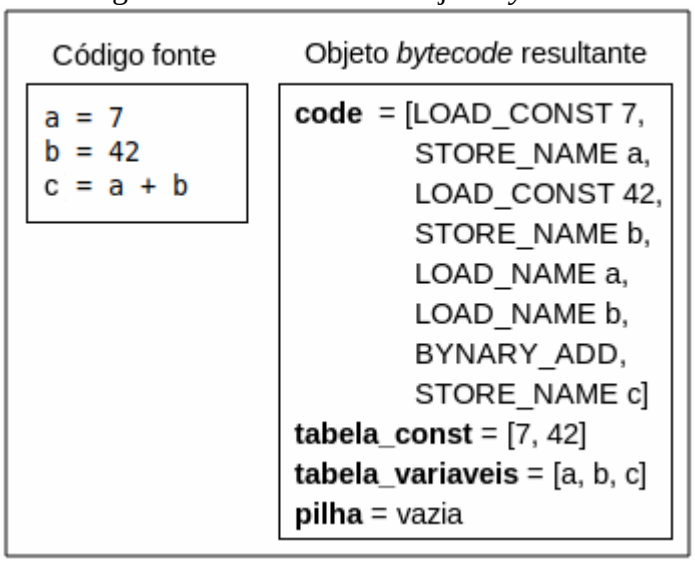


Fonte: Próprio autor.

Durante toda a análise sintática, conforme se faz necessário o armazenamento de uma variável ou constante, as tabelas de variáveis e de constantes, presentes na classe **BytecodePy**, são alimentadas. Os constituintes dessas duas tabelas são objetos do tipo **ObjetoPy**, que possuem todos os atributos necessários para representação de variáveis e de constantes.

Ao final das análises sintática e semântica, têm-se um único objeto *bytecode* contendo toda a estrutura necessária para alimentar a máquina virtual (etapa onde é realizada a execução do programa informado pelo usuário). A Figura 23 ilustra um código em Python qualquer, bem como a estrutura do objeto *bytecode* equivalente. A constante *None*, a fim de simplificações, foi ocultada da tabela de constantes.

Figura 23 – Estrutura do objeto *bytecode*



Fonte: Próprio autor.

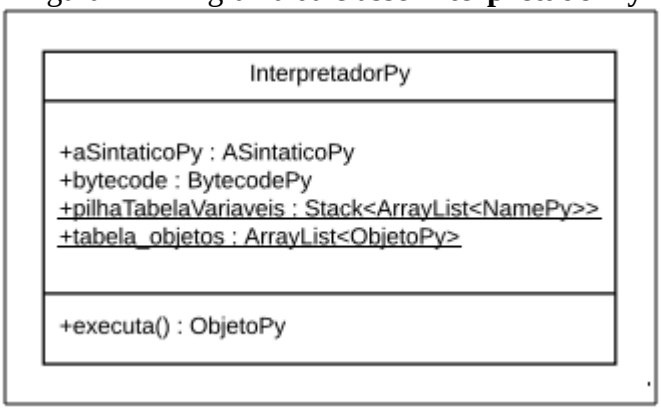
4.2 DESENVOLVIMENTO DO *BACK-END*

A segunda etapa do projeto foi o desenvolvimento do *back-end*, consistindo na máquina virtual, responsável pela execução do objeto *bytecode* gerado pelas análises do *front-end*. Pode-se entender a máquina virtual como um *software* à parte, que apenas recebe os objetos da etapa anterior e realiza as operações pertinentes, sendo que nenhuma estrutura em nível de código é gerada nessa etapa (algumas estruturas em nível de execução são geradas nessa etapa e serão discutidas nas seções posteriores). De maneira resumida, a máquina virtual é responsável por obter as entradas do usuário, executar as instruções *bytecode* e gerar as saídas necessárias.

4.2.1 Máquina virtual

Para o desenvolvimento da máquina virtual foi criada a classe **InterpretadorPy**, cujos atributos são um objeto de **BytecodePy** (que será a cópia do objeto *bytecode* recebido pelo construtor), um *ArrayList* estático de **ObjetoPy** utilizado como tabela para armazenamento de objetos em tempo de execução, e um objeto do tipo *Stack*, também estático, onde são empilhadas as tabelas de variáveis durante as chamadas de funções (Figura 24).

Figura 24 – Digrama da classe **InterpretadorPy**



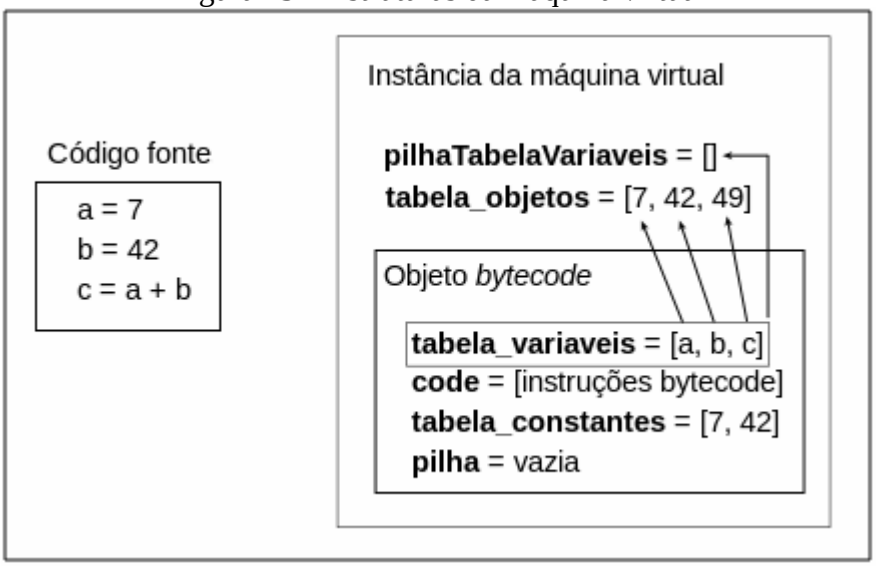
Fonte: Próprio autor.

No momento inicial da etapa de *back-end*, uma instância da classe **InterpretadorPy** é criada para execução do contexto global. Nessa instanciação é feita uma cópia do objeto *bytecode* do contexto global, bem como a instanciação do *ArrayList* global de objetos e da pilha de tabelas de variáveis.

O *ArrayList* de objetos, conforme dito, é estático, sendo único e compartilhado entre todas as possíveis instâncias da classe. Essa estrutura serve como área de objetos durante a execução do programa, podendo ser comparada à memória *heap*, onde os objetos ficam armazenados, podendo ser apontados por variáveis de diferentes contextos, tanto o global quanto o de alguma função.

A pilha de tabelas de variáveis possui a função de manter em uma única estrutura as tabelas de variáveis globais e tabelas de variáveis de possíveis funções. Esse tipo de estrutura é necessário para que qualquer contexto de execução tenha acesso tanto às variáveis locais quanto às globais. Assim, dependendo da instrução *bytecode* a ser executada, faz-se uma busca da informação na tabela de variáveis local ou na tabela global. A Figura 25 ilustra a relação entre a tabela de variáveis de um objeto *bytecode* e a tabela global de objetos (*heap*) ao final da terceira atribuição. Os objetos da tabela de variáveis referenciam objetos da tabela global de objetos. Além disso, a pilha de tabelas de variáveis da máquina virtual contém a tabela de variáveis do objeto *bytecode*.

Figura 25 – Estruturas da máquina virtual



Fonte: Próprio autor.

O principal método da máquina virtual, responsável pela execução do programa, contém um laço infinito que executa as instruções *bytecode* uma a uma. Um inteiro é utilizado como *bytecode counter*, cuja função é definir qual a próxima instrução a ser executada. A cada iteração, avança-se uma posição no *ArrayList* de instruções (incrementa-se o *bytecode counter*), a menos que alguma instrução de salto faça com que o *bytecode counter* aponte para outra posição.

Conforme definido no escopo do projeto, a máquina virtual utiliza uma pilha como área de trabalho, empilhando valores e resultados, sendo enquadrada, portanto, como uma máquina virtual de pilha. Conforme as instruções são executadas, a pilha de execução cresce e diminui, ficando, ao final da execução, vazia.

Além da pilha de execução, outras estruturas também vão sendo manipuladas durante a interpretação. Os objetos da tabela de variáveis, presentes no objeto *bytecode*, sofrem alterações durante a execução, podendo ter suas referências alteradas ou removidas, conforme a instrução executada. Todas as tabelas de variáveis ficam na pilha de tabelas de variáveis, sendo que esta também sofre alterações conforme funções são chamadas. A tabela de objetos (*heap*), outra estrutura própria da máquina virtual, também cresce e diminui conforme objetos são criados e removidos (a remoção do objeto é realizada pelo *Garbage Collector*, discutido mais adiante, ainda nesta seção).

Sempre que a máquina virtual executar a instrução *CALL_FUNCTION* (para uma chamada de função), uma nova instância da máquina virtual é criada. Essa nova instância possui todos os atributos necessários para que seja feita a execução da função de forma

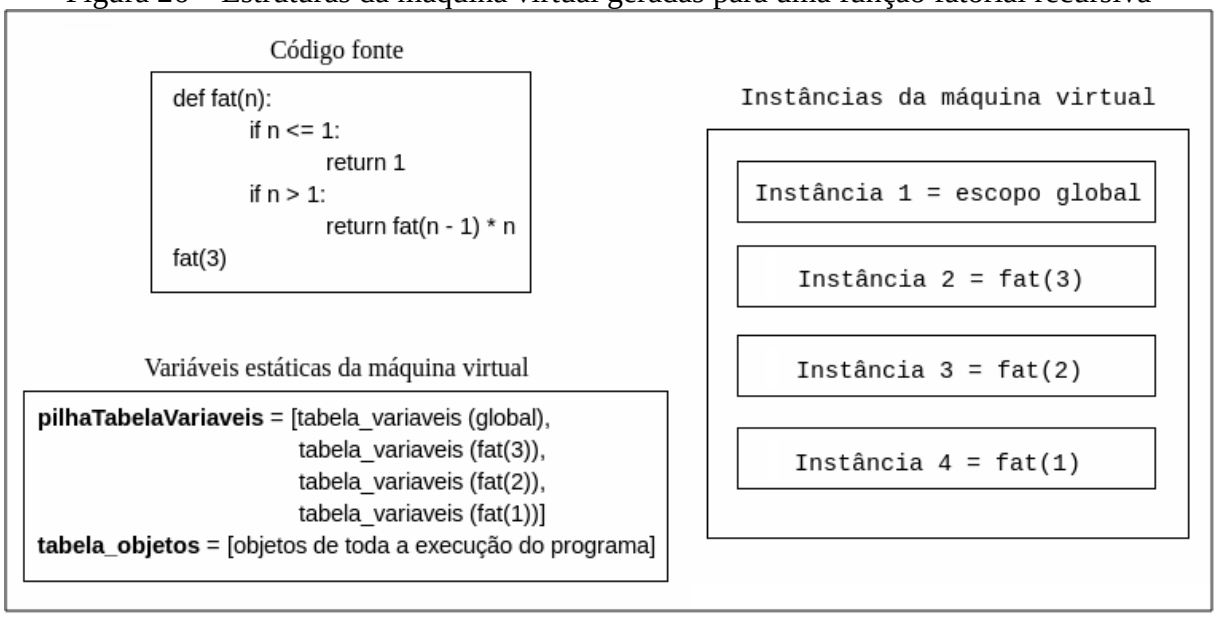
independente do contexto global. Como o objeto *bytecode* da função é passado como argumento para o construtor da máquina virtual, têm-se uma nova instância de **InterpetadorPy** com acesso tanto ao objeto *bytecode* da função (que engloba as instruções e as tabelas) quanto às demais estruturas (como a tabela de objetos e a pilha de tabelas de variáveis, ambas estáticas).

Sempre que há uma chamada de função, além da nova instanciação da máquina virtual, empilha-se a tabela de variáveis da função na pilha de tabelas de variáveis. Desta forma, a função a ser executada terá acesso tanto à tabela de variáveis local quanto à global.

Para manter uma independência entre as pilhas de execução, cada instância da máquina virtual executa as instruções com o auxílio de uma pilha independente, isto é, tanto o contexto global quanto o de cada função executa as instruções em uma pilha separada – a pilha de execução encontra-se na classe **BytecodePy**. Essa abordagem facilita tanto a geração de código intermediário quanto a interpretação do programa, já que os objetos *bytecode* não possuem uma dependência direta entre si.

Ao término da execução da função, invariavelmente, será executada a instrução *RETURN_VALUE*, que desempilha o topo da pilha e o devolve ao contexto que chamou a função, que, por sua vez, empilha esse retorno na pilha de execução local. A Figura 26 ilustra a execução de um programa fatorial recursivo. A imagem simplifica as estruturas da execução da máquina virtual logo após a última chamada recursiva da função. Na figura percebe-se a pilha de tabelas de variáveis (estática), que armazena, nesse momento particular da execução, as quatro tabelas de variáveis: a tabela do escopo global e a tabela de cada chamada de função. Além da pilha de tabelas de variáveis, há a tabela global de objetos (*heap*), também uma estrutura estática. Ao lado estão ilustradas as quatro instâncias da máquina virtual criadas para a execução tanto do contexto global quanto de cada chamada de função.

Figura 26 – Estruturas da máquina virtual geradas para uma função fatorial recursiva



Fonte: Próprio autor.

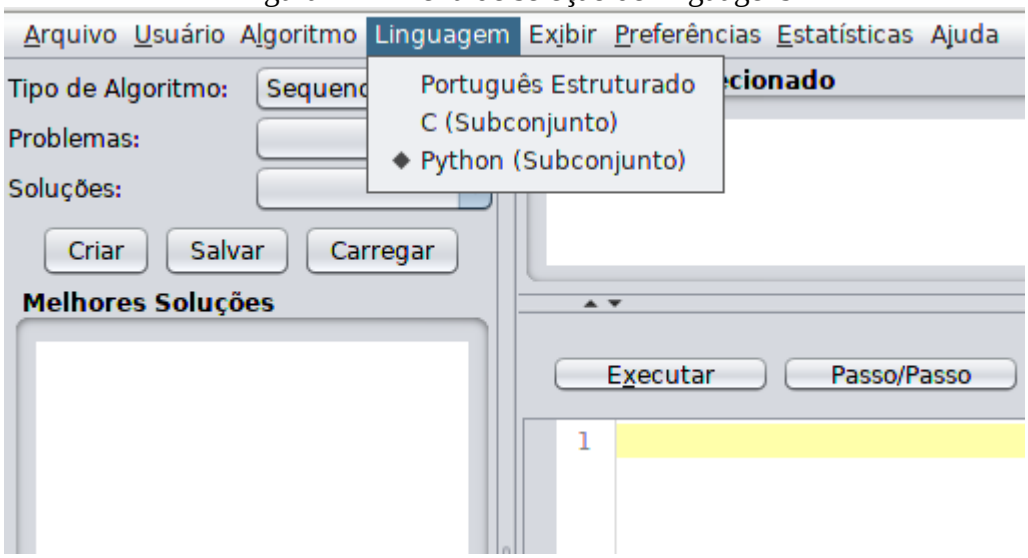
O *Garbage Collector* entra em ação sempre que a instrução *STORE_NAME* (atribuição de um objeto a uma variável) é executada. O contador de referências do objeto a ser atribuído é incrementado, e o contador de referências do objeto que perderá aquela referência em questão é decrementado. A partir disso, verifica-se se o objeto que perdeu a referência possui o contador de referências igual a zero. Caso positivo, significa que já não há mais nenhuma variável referenciando o objeto, podendo ser feita a sua remoção da tabela global de objetos (*heap*).

4.3 INTEGRAÇÃO COM O WEBALGO

Tendo como vantagem o fato de todo o ambiente do WebAlgo já estar devidamente implementado e estável, nenhuma funcionalidade além das propostas neste trabalho precisou ser implementada. Elementos de interface gráfica e de comunicação com o servidor da UCS foram reaproveitados, tendo sido utilizados de maneira a se comunicar apropriadamente com a implementação do compilador/interpretador de Python.

No menu de seleção da linguagem foi acrescida a opção para Python, bastando o usuário selecionar dessa listagem a linguagem em que irá escrever seu código. A Figura 27 ilustra o menu principal do WebAlgo onde a seleção da linguagem Python foi adicionada.

Figura 27 – Menu de seleção de linguagens

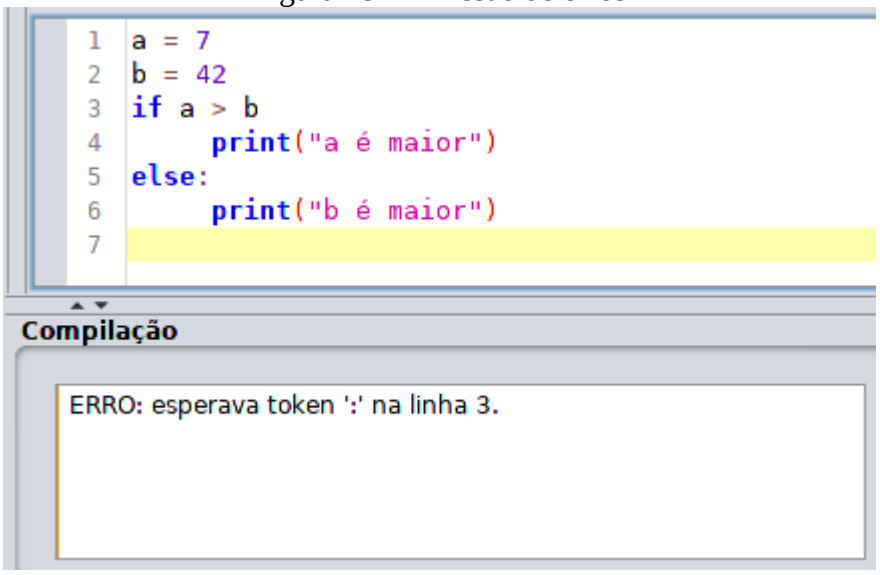


Fonte: Próprio autor.

Para que o aluno possa escrever seu programa em Python, basta que selecione a linguagem no menu ilustrado na Figura 27 e digite seu algoritmo no editor. Após escrever o programa, ao clicar no botão de execução é iniciado o processo de reconhecimento do código fonte informado no editor.

Quando, no momento das análises ou da interpretação, um erro sintático, semântico ou em tempo de execução for identificado, é emitida uma mensagem ao usuário e o processo de análise ou execução é interrompido (Figura 28).

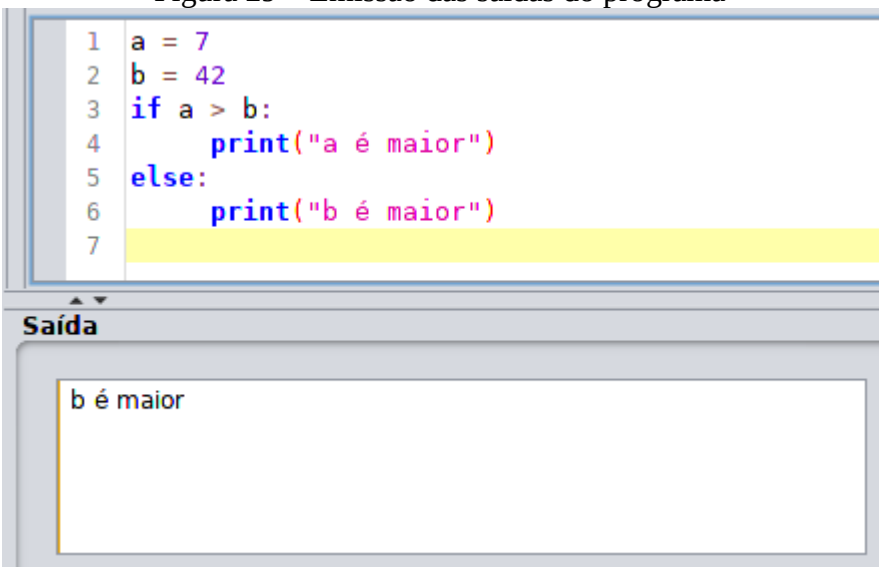
Figura 28 – Emissão de erros



Fonte: Próprio autor.

Quando o código informado pelo usuário não possuir inconsistências léxicas, sintáticas ou semânticas, o programa é executado pela máquina virtual, sendo exibidas ao aluno as saídas resultantes (Figura 29).

Figura 29 – Emissão das saídas do programa

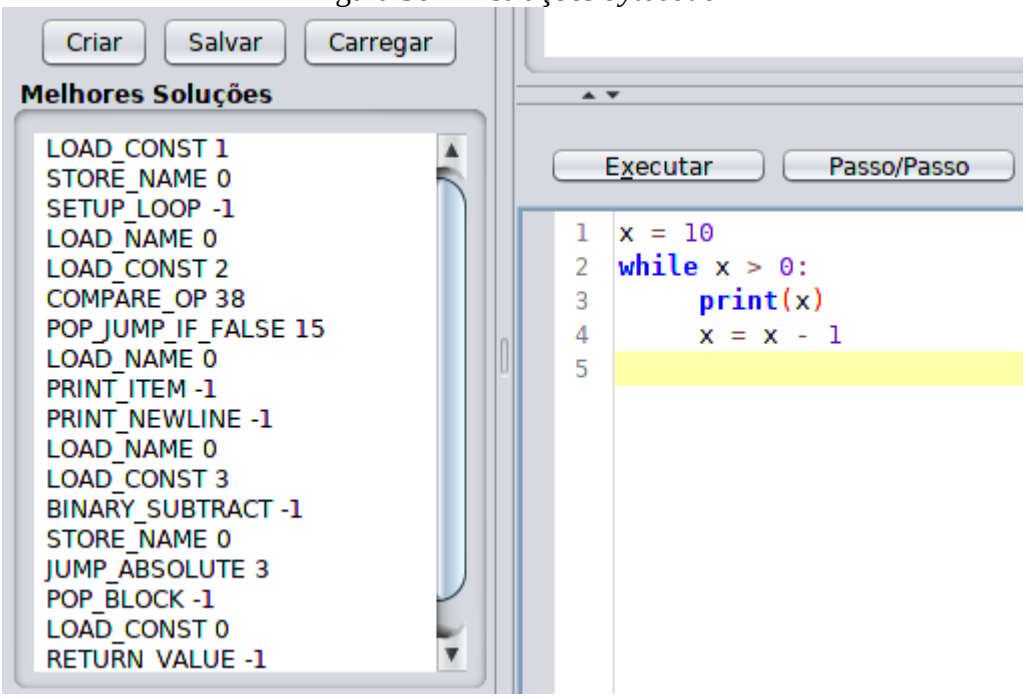


Fonte: Próprio autor.

Durante o desenvolvimento da ferramenta, foi utilizada a caixa de texto das melhores soluções para a exibição das instruções *bytecode* geradas para o código informado. Após a conclusão do *software* esse recurso foi retirado, voltando a ter somente a utilidade originalmente concebida (Figura 30).

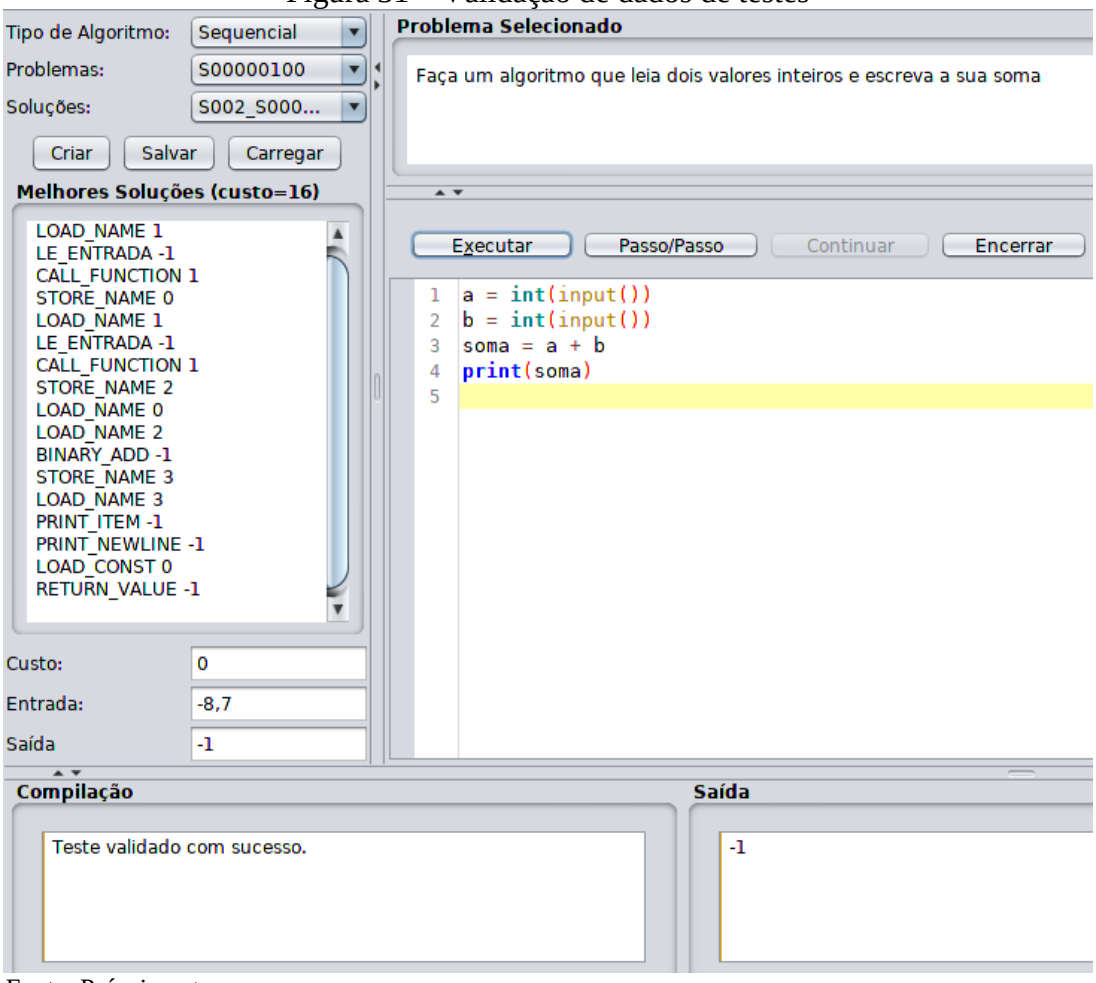
Outro recurso do WebAlgo que foi integrado ao compilador/máquina virtual foi o validador de casos de testes. O usuário carrega um exercício da lista de problemas, escreve seu programa em Python, insere os dados de entrada e de saída, e faz-se uma validação daquele caso de testes. Uma mensagem apropriada é exibida ao usuário conforme validação (Figura 31). A integração com a lista de exercícios (incluindo a carga e gravação de soluções) foi também realizada, operando de forma perfeitamente funcional com a versão de Python do WebAlgo.

Figura 30 – Instruções *bytecode*



Fonte: Próprio autor.

Figura 31 – Validação de dados de testes



Fonte: Próprio autor.

5 CONSIDERAÇÕES FINAIS

Durante este trabalho foi realizada a implementação de um compilador e de uma máquina virtual de um subconjunto da linguagem Python. A linguagem foi escolhida devido à sua crescente presença no meio acadêmico. Em detrimento do desenvolvimento de um novo ambiente, decidiu-se a reutilização do *software* WebAlgo, tirando proveito das funcionalidades pedagógicas já existentes neste sistema.

Foi optado pelo desenvolvimento de um novo interpretador em vez da utilização de uma máquina virtual já existente, como a JVM, por exemplo. A abordagem foi adotada tendo em vista o vasto conteúdo disponível sobre o funcionamento do interpretador oficial da linguagem. A equivalência entre comandos em Python e instruções *bytecode* pôde ser facilmente identificada. Com a vantagem de cada instrução de baixo nível ter seu funcionamento especificado na documentação oficial, a escolha pela implementação de uma nova máquina virtual mostrou-se conveniente.

A escolha pelo desenvolvimento de uma máquina virtual baseada na implementação oficial de Python, além de ter servido como uma segurança extra durante o desenvolvimento do projeto, facilita futuras inclusões de novas funcionalidades. Tendo em vista que cada funcionalidade de Python é relativamente independente das demais, novos recursos da linguagem podem ser implementados ao WebAlgo com relativa facilidade.

Apesar de alterações pontuais terem sido realizadas em relação ao que foi proposto no primeiro trabalho de conclusão, de um modo geral foi possível seguir o escopo do projeto. Durante o decorrer do trabalho verificou-se a necessidade de alterações específicas na gramática e no subconjunto de *bytecodes,* ficando as alterações concentradas principalmente nesses dois pontos. Entretanto, nenhuma mudança drástica precisou ser realizada, tendo sido o trabalho desenvolvido com relativa segurança do início ao fim.

Os recursos implementados neste trabalho de conclusão são abrangentes o suficiente para serem utilizados em disciplinas introdutórias à programação. A integração com algumas funcionalidades pedagógicas do WebAlgo também está operacional. A comunicação com a lista de problemas, incluindo a possibilidade de o usuário carregar e salvar seu exercício no servidor, está operando de maneira satisfatória. Outra funcionalidade do WebAlgo que foi integrada ao compilador/máquina virtual de Python foi o teste automatizado, conforme descrito na Seção 4.3, funcionando devidamente tanto para valores únicos de entrada e saída, quanto para conjuntos de “n” elementos.

Como a base do compilador e da máquina virtual estão implementadas, diversas possibilidades de trabalhos futuros se mostram plausíveis. Num primeiro momento, os recursos que mais se mostram necessários são o suporte a dicionários e conjuntos. A implementação do suporte a alguma biblioteca específica também é uma possibilidade viável. Outra possibilidade seria a implementação do suporte à importação de módulos externos com o uso da palavra reservada "*import*". Esse tipo de funcionalidade expandiria de maneira significativa os recursos suportados pela versão de Python do WebAlgo.

Após o término deste trabalho, pôde-se constatar que a implementação completa de um compilador e uma máquina virtual pode apresentar um grau elevado de complexidade. As dificuldades e os riscos de imprevistos tendem a aumentar caso o desenvolvimento não seja baseado em alguma implementação que já tenha se mostrado viável. Como neste projeto foi seguido de modo o mais fiel possível ao CPython, possíveis imprevistos ou dificuldades se mostraram consideravelmente menos prováveis.

Vale o destaque, também, de que uma implementação desse tipo de *software* só se torna possível caso se faça uma abordagem sistemática, utilizando regras e técnicas já estabelecidas. Neste trabalho a estratégia utilizada foi a Tradução Dirigida por Sintaxe (TDS) bem estabelecida na obra de Aho, Sethi e Ullman (1995). Além disso, para validação sintática foi utilizada a gramática oficial de Python, resultando numa maior segurança durante a implementação.

Espera-se que com essa primeira versão do suporte a Python, o WebAlgo seja utilizado de maneira cada vez mais frequente no ensino básico de programação de computadores. Com o acréscimo de funcionalidades através de trabalhos futuros, será possível a utilização da ferramenta não somente em disciplinas introdutórias de programação, mas também em disciplinas que possuam uma necessidade mais específica de recursos.

REFERÊNCIAS

AHO, Alfred V.; SETHI, Ravi; ULLMAN, Jeffrey D.. **Compiladores: Princípios, Técnicas e Ferramentas.** [S. l.]: LTC, 1995. 344 p.

AHO, Alfred V. et al. **Compilers: Principles, Techniques, and Tools.** 2. ed. Boston: Addison-wesley, 2007.

BARRON, David W.. **The World of Scripting Languages.** [S. l.]: John Wiley & Sons, Inc., 2000. 506 p.

BAU, David. Droplet, a blocks-based editor for text code. **Journal Of Computing Sciences In Colleges**, [s. L.], v. 30, n. 6, p.138-144, jun. 2015. Disponível em: <https://dl.acm.org/citation.cfm?id=2753052>. Acesso em: 26 maio 2019.

BERGIN, Susan; REILLY, Ronan. The influence of motivation and comfort-level on learning to program. In: BERGIN, Susan; REILLY, Ronan. **Proceedings of the 17th Workshop of the Psychology of Programming Interest Group.** Brighton: Psychology Of Programming Interest Group, PPIG 05, 2005. p. 293-304. Disponível em: <http://mural.maynoothuniversity.ie/8685/1/SB-Influence-2005.pdf>. Acesso em: 26 maio 2019.

CRAIG, Iain D.. **Virtual Machines.** [S. l.]: Springer, 2005. 288 p.

DORNELES, Ricardo Vargas; PICININ JR, Delcino; ADAMI, André Gustavo. ALGOWEB: a web-based environment for learning introductory programming. In: DORNELES, Ricardo Vargas; PICININ JR, Delcino; ADAMI, André Gustavo. **2010 10th IEEE International Conference on Advanced Learning Technologies.** Sousse: IEEE, 2010. p. 83-85. Disponível em: <https://ieeexplore.ieee.org/abstract/document/5571154/>. Acesso em: 26 maio 2019.

IKE-NWOSU, Obi. **Inside The Python Virtual Machine.** [S. l.]: Leanpub, 2017. 125 p.

KAPTUR, Allison. **A Python Interpreter Written in Python.** 201-?. Disponível em: <https://www.aosabook.org/en/500L/a-python-interpreter-written-in-python.html>. Acesso em: 28 maio 2019.

LESSA, Andre. **Python Developer's Handbook.** [S. l.]: Sams Publishing, 2000. 960 p.

MALONEY, John et al. The Scratch Programming Language and Environment. **Acm Transactions On Computing Education**, [S. l.], v. 10, n. 4, p. 1-15, 1 nov. 2010. Association for Computing Machinery (ACM). http://dx.doi.org/10.1145/1868358.1868363.

MAK, Ronald. **Writing Compilers and Interpreters: A Software Engineering Approach.** 3. ed. [S. l.]: Wiley Publishing, 2009. 864 p.

NAOE, Aline. **Curso on-line gratuito da USP ensina conceitos básicos de computação.** 2016. Disponível em: <jornal.usp.br/?p=58176>. Acesso em: 31 maio 2019.

PYTHON BRASIL, **FuncionamentoGarbageCollector**. 2008. Disponível em <https://wiki.python.org.br/FuncionamentoGarbageCollector> Acesso em 28 maio 2019.

PYTHON SOFTWARE FOUNDATION. **Memory Management.** 2019a. Disponível em: <https://docs.python.org/3.6/c-api/memory.html>. Acesso em: 28 maio 2019.

PYTHON SOFTWARE FOUNDATION. **byteplay Module Documentation.** 2019b. Disponível em: <https://wiki.python.org/moin/ByteplayDoc>. Acesso em: 28 maio 2019.

PYTHON SOFTWARE FOUNDATION. **10. Full Grammar specification.** 2019c. Disponível em: <https://docs.python.org/3.6/reference/grammar.html>. Acesso em: 01 jun. 2019.

SMITH, J. E.; NAIR, Ravi. **Virtual Machines: Versatile Platforms for Systems and Processes.** [S. l.]: Morgan Kaufmann, 2005. 656 p.

VANDERPLAS, Jake. **A Whirlwind Tour of Python.** [S. l.]: O'Reilly Media, Inc., 2016. 91 p.

ZAKHARENKO, N., **Nina Zakharenko - Memory Management in Python - The Basics – PyCon 2016**. 2016. (26m59s). Disponível em: <https://www.youtube.com/watch?v=F6u5rhUQ6dU>. Acesso em: 28 maio 2018.