



LINGUAGENS E PARADIGMAS DE PROGRAMAÇÃO
TRABALHO EM GRUPO
2024/2



1. Histórico de mudanças no documento

Versão 1.0	- Versão inicial do trabalho
Versão 2.0	- Permite “prototype” dentro de “if-else” e da “main” - Adiciona a instrução “pop” - Atualiza a seção 14.2 sobre o uso da instrução “pop”
Versão 3.0	- Altera a instrução “set”, trocando a ordem do valor com o objeto

2. Regras do trabalho

- A data de entrega do trabalho será no dia 29 de outubro de 2024.
- O trabalho consiste na criação de dois programas:
 - Um compilador de BOOL para instruções de máquina virtual de pilha.
 - Um interpretador de máquina virtual de pilha.
- O compilador e interpretador devem ser escrito na linguagem Java. O seu trabalho será testado no ambiente Linux com Java versão 22.
 - Deve ser enviado um arquivo ZIP contendo o projeto IntelliJ.
- O trabalho deve ser feito em grupo de 3 alunos.
- O grupo deverá apresentar o trabalho para o professor, em horário agendado.
 - Caso o grupo não apresente, o trabalho terá nota zero.
- Qualquer plágio (total ou parcial) implicará em nota zero para todos os envolvidos.
- Não é permitido o uso de bibliotecas geradoras de gramática, apenas expressões regulares no reconhecimento do programa.
- Assuma que todo o programa do usuário estará correto, ou seja, não precisa ficar verificando o formato e argumentos das operações.
- Caso omissos devem ser tratados com o professor.
 - Não tente assumir qualquer coisa se estiver com dúvida.

3. Exemplo da linguagem

O trabalho se baseia na implementação de um interpretador para uma linguagem BOOL – *Bruno's Object-Oriented Language*.

A linguagem permite a definição de classes, com métodos e atributos. A linguagem também possui um mecanismo de herança e coletor de lixo. O programa inicia a execução na “main”.

Abaixo é apresentado um exemplo de um programa em BOOL:

```
class Base
  vars id

  method showid()
  vars x
  begin
    self.id = 10
    x = self.id
    io.print(x)
    x = 0
    return x
  end-method
end-class

class Pessoa
  vars num

  method calc(x)
  vars y, z
  begin
    z = self.num
    y = x + z
    io.print(y)
    y = new Base
    return y
  end-method
end-class

main()
vars p, b, x
begin
  b = new Base
  p = new Pessoa
  p._prototype = b

  b.id = 111

  p.num = 123
  p.id = 321

  x = 1024
  p.showid()
  p.calc(x)
end
```

4. Considerações sobre a sintaxe da linguagem

- Nomes de métodos, parâmetros, variáveis locais, atributos e classes serão sempre formados por sequência de letras (maiúsculas ou minúsculas).
- Esses nomes não podem utilizar palavras reservadas da linguagem:
 - `class`, `method`, `begin`, `self`, `vars`, `end`, `if`, `return`, `eq`, `ne`, `lt`, `le`, `gt`, `ge`, `new`, `main`, `io`¹.
- Pode haver espaços no início da linha, no final da linha ou entre os elementos da linguagem.
 - Pode haver linhas em branco, que devem ser ignoradas.
- Os números sempre serão inteiros 32-bits (positivos ou negativos).

5. Considerações sobre a semântica da linguagem

- Variáveis locais e atributos são inicializadas com o valor inteiro 0 (zero).
- Atributos, variáveis e parâmetros podem armazenar números inteiros ou referência para objetos.
- O parâmetro pode ser alterado como se fosse uma variável.
- Um método sempre deve retornar, seja um valor inteiro, seja uma referência a um objeto.
- Um método pode acessar seus parâmetros e variáveis locais diretamente, mas os atributos do objeto devem ser acessados via referência *self*.
- A linguagem possui coletor de lixo, que deve ser implementado pelo interpretador.

6. Classe

A linguagem pode ter (zero ou) várias definições de classe. Uma classe pode ter (zero ou) vários métodos e atributos. Os métodos podem ter (zero ou) vários parâmetros e (zero ou) várias variáveis locais. Os atributos, parâmetros e variáveis locais podem armazenar valores inteiros ou referências para objetos. Todo método deve retornar um valor (inteiro ou referência a um objeto).

Exemplo:

<pre>class Base vars a, b, c method calc() vars x begin x = self.a io.print(x) x = 0 return x end-method end-class</pre>	<pre>class Frutas method info(a, b) vars c begin io.print(a) c = 10 b = a + c return b end-method end-class</pre>
---	--

7. Objetos

Um objeto é criado utilizando o comando “new”, seguido do nome da classe. A linguagem segue o mesmo modelo de Java, onde o que é armazenado em variáveis, parâmetros ou atributos é uma referência ao objeto e não o objeto em si.

No código abaixo, só existe um objeto `Person`, tanto “x” como “y” apontam para o mesmo objeto.

<pre>x = new Person y = x</pre>

¹ “io” é reservado apenas para nome de variável ou parâmetro, pois há um objeto *built-in*. No entanto, deve ser possível criar um método com o nome “io”.

8. Expressão de atribuição

A linguagem permite atribuição para alterar parâmetros, variáveis locais e atributos de objetos. A atribuição pode ser simples, expressão aritmética, chamada de função ou criação de objeto. Operações aritméticas só podem ser realizadas sobre variáveis ou parâmetros, ou seja, deve-se atribuir os valores de atributos ou constantes para variáveis ou parâmetros para depois fazer a operação.

Exemplo:

```
a = 1024
b = 128
c = b + a
o = new Base
o.id = 128
o.id = c
o.num = p.calc(a)
pessoa.age = dog.age
obj = p.change()
obj.num = 10
```

9. Condicional

A linguagem possui o seletor “if” / “if-else”. A expressão de teste pode utiliza os operadores *eq*, *ne*, *gt*, *ge*, *lt*, *le* (respectivamente: igual, não-igual, maior, maior-igual, menor, menor-igual). O teste só pode conter a comparação entre variáveis ou parâmetros (sem atributos, constantes ou chamada de método). O corpo do bloco do “if” ou “else” podem conter atribuições, chamada de método ou retorno (não permite “if” aninhado).

Exemplo:

<pre>a = 1024 b = 128 if a eq b then c = b + a else c = 123 end-if</pre>	<pre>a = 1024 b = obj.num if a gt b then c = obj.calc(a, b) obj.func() io.print(c) end-if</pre>	<pre>a = 1024 b = obj.num if a gt b then return 10 end-if</pre>
---	--	--

10. Mecanismo de herança

Todo objeto possui um atributo especial chamado “_prototype”, que pode apontar para um outro objeto, mas não para o próprio objeto. Quando um é criado, “_prototype” aponta para nada. Note que a linguagem não possui “null”, isso é detalhe de implementação e resolvido internamente.

Quando um método é chamado em um objeto, primeiro é verificado se o objeto implementa o método. Caso afirmativo, o método é invocado. No entanto, se o método não for encontrado, se “_prototype” aponta para outro objeto, o objeto apontado deve ser inspecionado, e caso tenha o método, este é invocado. Senão, o atributo “_prototype” deve ser inspecionado no novo objeto, e o método invocado, se existir. O processo continua até localizar um objeto que implemente o método (assuma que sempre haverá um objeto que implementa o método).

Quando um atributo for acessado em um objeto (leitura ou alteração), o atributo primeiro é procurado no objeto. Caso exista, o atributo é lido ou alterado. No entanto, se o atributo não for encontrado, o atributo “_prototype” é utilizado para localizar (recursivamente) um objeto que contenha o atributo. Então, o valor do atributo é lido ou alterado.

11. Variável *self*

Todo método possui uma variável implícita chamada “self” que aponta para o objeto em que o método foi chamado. Mesmo utilizando o atributo “_prototype” para localizar o método, a variável “self” aponta para o objeto em que o método foi originalmente chamado.

12. Objeto io

Existe um objeto *built-in* chamado “io” que pode ser acessado de qualquer método ou do corpo principal. Esse objeto possui apenas um métodos:

- `io.print(n)`: recebe uma variável ou parâmetro inteiros, mostrando seu valor na tela, seguido de ‘\n’. Esse método retorna 0 (zero).

13. Função principal

A função principal “main” é definida após as classes, sendo o primeiro trecho de código a ser executado. Ela não recebe parâmetros e pode definir variáveis locais. Esta função não retorna valor ao final.

<pre>main() vars a, b, c, x begin a = new Base x = a.num b = 1023 c = b * x io.print(c) end</pre>	<pre>main() vars obj begin obj = new Base obj.run() end</pre>
--	---

14. Máquina de pilha

O interpretador deverá ser implementado como uma máquina de pilha cujas as instruções serão mostradas adiante. Isso significa que a execução das instruções supõe uma pilha onde os valores intermediários serão armazenados, por exemplo, a soma espera que os dois valores estejam nesta pilha. Além disso, a pilha serve como auxílio na passagem de parâmetros para chamada de métodos; por exemplo, para a chamada de um métodos, os valores de entrada da chamada do método serão colocados na pilha e depois transferidos para os parâmetros.

Os resultados das operações (soma ou chamada de método) também voltam para a pilha.

14.1. Instruções

A seguir está a lista de instruções que o compilador deve gerar e o interpretador deve reconhecer e executar.

Nos exemplos abaixo são usado comentários com “//”, isso não faz parte da linguagem, eles foram colocados nos exemplos apenas como forma de ajudar no entendimento. Os programas em BOOL não terão comentários.

Instrução	<code>const <number></code>
Descrição	Coloca o número inteiro (positivo ou negativo) no topo da pilha.
Exemplo	<pre>const 10 const -20 const 10283</pre>

Instrução	<code>load <name></code>
Descrição	Coloca o valor da variável ou parâmetro “name” no topo da pilha.
Exemplo	<pre>load a load x load self</pre>

Instrução	store <name>
Descrição	Retira o valor do topo da pilha e armazena na variável ou parâmetro “name”.
Exemplo	<pre>const 10 store a // a = 10 load a store b // b = a</pre>

Instrução	add
Descrição	<p>Soma dois valores do topo da pilha, colocando o resultado de volta no topo. Ela executa o seguinte pseudocódigo:</p> <pre>v2 = pop() v1 = pop() push(v1 + v2)</pre>
Exemplo	<pre>load a load b add // a + c store c // c = a + c</pre>

Instrução	sub
Descrição	<p>Subtrai dois valores do topo da pilha, colocando o resultado de volta no topo. Ela executa o seguinte pseudocódigo:</p> <pre>v2 = pop() v1 = pop() push(v1 - v2)</pre>
Exemplo	<pre>load a load b sub // a - b store c // c = a - b</pre>

Instrução	mul
Descrição	<p>Multiplica dois valores do topo da pilha, colocando o resultado de volta no topo. Ela executa o seguinte pseudocódigo:</p> <pre>v2 = pop() v1 = pop() push(v1 * v2)</pre>
Exemplo	<pre>load a load b mul // a * b store c // c = a * b</pre>

Instrução	div
Descrição	Divisão <u>inteira</u> de dois valores do topo da pilha, colocando o resultado de volta no topo. Ela executa o seguinte pseudocódigo: <pre> v2 = pop() v1 = pop() push(v1 / v2) </pre>
Exemplo	<pre> load a load b div // a / b store c // c = a / b </pre>

Instrução	new <class>
Descrição	Cria um novo objeto da classe e coloca uma referência para esse objeto no topo da pilha.
Exemplo	<pre> new Person store p // p = new Person </pre>

Instrução	get <name>
Descrição	Coloca o valor do atributo “name” de um objeto no topo da pilha. Executa o seguinte pseudocódigo: <pre> obj = pop() push(obj.name) </pre> É importante lembrar que esse instrução deve seguir o mecanismo de herança ao ser executada.
Exemplo	<pre> load p // Coloca referência de 'p' no topo da pilha get age // Pega o valor de p.age e coloca no topo da pilha store b // Armazena o valor de 'age' em 'b' (b = p.age) </pre>

Instrução	set <name>
Descrição	Atribui o valor do topo da pilha ao atributo “name” de um objeto. Executa o seguinte pseudocódigo: <pre> obj = pop() v = pop() obj.name = v </pre> É importante lembrar que esse instrução deve seguir o mecanismo de herança ao ser executada.
Exemplo	<pre> const 25 // Coloca o valor a ser atribuído na pilha load p // Coloca referência ao objeto na pilha set age // p.age = 25 </pre>

Instrução	call <name>
Descrição	<p>Chama o método “name” de um objeto. Os parâmetros devem ser colocados na pilha antes da chamada do método. O retorno do método deve ser colocado na pilha.</p> <p>Note que na chamada de método, os parâmetro passados são empilhados da esquerda para direita.</p> <p>É importante lembrar que esse instrução deve seguir o mecanismo de herança ao ser executada.</p>
Exemplo	<pre>// b = obj.soma(a, b) load a load b load obj // Coloca referência ao objeto na pilha call soma // Chama “soma” em obj store b</pre>

Instrução	gt
Descrição	<p>Compara dois valores do topo da pilha e coloca o resultado de volta no topo. Ela executa o seguinte pseudocódigo:</p> <pre>v2 = pop() v1 = pop() push(v1 > v2)</pre> <p>Note que é colocado um valor booleano no topo da pilha como resultado da comparação.</p>
Exemplo	<pre>load x load y gt // x > y</pre>

Instrução	ge
Descrição	<p>Compara dois valores do topo da pilha e coloca o resultado de volta no topo. Ela executa o seguinte pseudocódigo:</p> <pre>v2 = pop() v1 = pop() push(v1 >= v2)</pre> <p>Note que é colocado um valor booleano no topo da pilha como resultado da comparação.</p>
Exemplo	<pre>load x load y ge // x >= y</pre>

Instrução	lt
Descrição	<p>Compara dois valores do topo da pilha e coloca o resultado de volta no topo. Ela executa o seguinte pseudocódigo:</p> <pre>v2 = pop() v1 = pop() push(v1 < v2)</pre> <p>Note que é colocado um valor booleano no topo da pilha como resultado da comparação.</p>
Exemplo	<pre>load x load y lt // x < y ?</pre>

Instrução	le
Descrição	<p>Compara dois valores do topo da pilha e coloca o resultado de volta no topo. Ela executa o seguinte pseudocódigo:</p> <pre> v2 = pop() v1 = pop() push(v1 <= v2) </pre> <p>Note que é colocado um valor booleano no topo da pilha como resultado da comparação.</p>
Exemplo	<pre> load x load y le // x <= y </pre>

Instrução	eq
Descrição	<p>Compara dois valores do topo da pilha e coloca o resultado de volta no topo. Ela executa o seguinte pseudocódigo:</p> <pre> v2 = pop() v1 = pop() push(v1 == v2) </pre> <p>Note que é colocado um valor booleano no topo da pilha como resultado da comparação.</p>
Exemplo	<pre> load x load y eq // x == y </pre>

Instrução	ne
Descrição	<p>Compara dois valores do topo da pilha e coloca o resultado de volta no topo. Ela executa o seguinte pseudocódigo:</p> <pre> v2 = pop() v1 = pop() push(v1 != v2) </pre> <p>Note que é colocado um valor booleano no topo da pilha como resultado da comparação.</p>
Exemplo	<pre> load x load y ne // x != y </pre>

Instrução	pop
Descrição	Remove o elemento do topo da lista (descarta).
Exemplo	<pre> load obj call size pop // Remove o retorno de obj.size() da pilha </pre>

Instrução	if <n>
Descrição	Retira do topo da pilha um valor booleano e se esse valor for “false”, não executa as “n” (salta “n” instruções). Caso contrário, não faz nada, ou seja, executa as instruções do “if”.
Exemplo	<pre>// if (x == y) { b = obj.soma(a, b) } load x load y eq if 5 // Se pop() == false, // não execute as '5' próximas instruções load a load b load obj call soma store b // b = obj.soma(a, b)</pre>

Instrução	else <n>
Descrição	Quando presente, esta instrução trabalha em conjunto com a instrução “if”. Se o valor retirado da pilha por “if” for “true”, não executa as “n” instruções seguintes. Caso contrário, não faz nada, ou seja, executa as instruções do “else”.
Exemplo	<pre>// if (x > y) { b = 10 } eles { a = x + y} load x load y gt if 2 // tmp = pop() // Se (tmp == false), // não execute as '2' próximas instruções const 10 store b else 4 // Se (tmp == true), // não execute as '4' próximas instruções load x load y add store a</pre>

Instrução	ret
Descrição	Retorna o controle para quem chamou o método. Note que a instrução não possui valor de retorno, esse valor deve ser colocado na pilha antes de chamar esta instrução.
Exemplo	<pre>// return x load x ret</pre>

14.2. Chamada de método

Apenas para esclarecer a chamada de método utilizando a instrução “call”. O interpretador ao executar essa instrução deve olhar a assinatura do método que está sendo chamado para saber quantos parâmetros são necessários. Então, o interpretador deve retirar da pilha a quantidade de valores correspondentes e atribuir esses valores aos parâmetros. Lembrando que os parâmetros são empilhados da esquerda para a direita.

Por exemplo, suponha que tenhamos o seguinte métodos:


```
method process(x, y, z)
begin
end
```

Se fossemos chamar esse método com “obj.process(a, b, c)” – supondo a=10, b=20, c=30. Isso seria traduzido como:

```
load a
load b
load c
load obj
call process
```

Obs: Os argumentos para uma chamada de método só podem ser parâmetros ou variáveis locais (vide a descrição BNF).

Note que a pilha está da seguinte forma antes da chamada da instrução “call”:

obj	 Topo da pilha
30	
20	
10	

Então, o interpretador, ao executar a instrução “call process”, deve retirar a referência ao objeto da pilha, procurar pelo método “process” (utilizar herança se for o caso), identificar que ele precisa de 3 valores e então retirar 30, 20 e 10 da pilha, atribuindo x = 10, y = 20 e z = 30. Assim, a atribuição aos parâmetros é feita da direita para esquerda, com z recebendo o topo, depois y e por último x.

Além disso, a instrução “call” deve atribuir à variável “self” o objeto que foi retirado do topo da pilha.

O valor de retorno do método é deixado no topo da pilha. Se for uma atribuição, esse valor deve então ser atribuído à variável. Caso seja uma chamada sem atribuição, o valor deve ser retirado do topo da pilha utilizando a instrução “pop”.

14.3. Compilação de BOOL

A primeira parte do trabalho é criar um compilador que traduza as instruções dos métodos e da função principal para instruções de uma máquina de pilha. O formato de saída será parecido com BOOL a menos do corpo dos métodos e da função principal. Por exemplo:

Programa em BOOL	Programa BOOL compilado
<pre>class Base vars id method showid() vars x begin self.id = 10 x = self.id return x end-method end-class</pre>	<pre>class Base vars id method showid() vars x begin const 10 load self set id load self get id store x load x ret end-method end-class</pre>

O seu compilador deve receber o nome de um arquivo BOOL e o nome de um arquivo de saída. O compilador então lê e compila o corpo dos métodos e da *main* do arquivo BOOL (mantendo o restante das informações inalteradas) e salva o resultado no outro arquivo. Por exemplo:

```
java BoolCompiler file.bool file.boolc
```

15. Interpretação de BOOL

O segundo programa a ser desenvolvido deve ser o interpretador BOOL, que terá como entrada um arquivo BOOL compilado e deve executar o programa, começando da função *main*. Por exemplo:

```
java BoolInterpreter file.boolc
```

O interpretador deve ser responsável por criar uma representação para as classes, objetos, métodos, etc., de tal forma que o programa funcione como esperado.

15.1. Coletor de lixo

A linguagem BOOL só tem comando para criar objetos, assim a destruição do objeto deve ser realizada por um coletor de lixo quando o objeto que não estiver mais sendo utilizado. A tarefa de coletar os objetos é do interpretador.

No trabalho, a cada 5 instruções executadas, a rotina de coleta de lixo do interpretador deve ser chamada. Deve-se implementar uma versão simples de coletor *mark-and-sweep* como descrito à seguir.

Essa técnica consiste em dividir a rotina de coleta de lixo em duas fases, a de marcação (*mark*) e depois de coleta (*sweep*). Quando a rotina de coleta executar, primeiro ela vai marcar os objetos de uma cor e, ao final, os objetos que não estiverem sendo marcados na cor em questão serão apagados.

Na primeira fase de marcação é utilizada, por exemplo, a cor *vermelha*. A rotina vai passar pela pilha, parâmetros, variáveis, atributos, etc., verificando se eles são uma referência para algum objeto. Se for, o objeto é então marcado de vermelho. Ao final da fase de marcação, os objetos que não estiverem marcados de *vermelho* são coletados.

Na próxima vez que a rotina de coleta executar, ela irá utilizar a cor *preta*. Da mesma forma, a rotina passa pela pilha, parâmetros, variáveis, atributos, etc., marcando os objetos como preto. Ao final, os objetos que não estiverem marcados de preto, são coletados.

O processo continua indefinidamente, sempre alternando as cores vermelha e preta.

Apenas uma ressalva é feita na criação dos objetos. Para simplificar, quando um objeto é criado com “new”, ele é marcado como *cinza*. Isso evita em saber qual é a cor da vez. Note que se ao final da fase de marcação, se o objeto ainda estiver como cinza, ele é coletado. Apenas objetos marcados com a cor da vez (preto ou vermelho) não são coletados.

16. BNF da linguagem BOOL

A seguir é apresentado o BNF da linguagem BOOL, mostrando quais são as possíveis construções que a linguagem aceita.

```
<program> → <main-body>
           | <class-defs> <main-body>

<class-defs> | <class-def>
              | <class-def> <class-defs>

<class-def> → class <name> '\n' <attrs-def> <methods-def> end-class '\n'
           | class <name> '\n' <attrs-def> end-class '\n'
           | class <name> '\n' <methods-def> end-class '\n'

<attrs-def> → vars <name-list> '\n'
```

```

<name-list> → <name>
            | <name>, <name-list>
<methods-def> → <method-def>
            | <method-def> <method-defs>
<method-def> → <method-header> <vars-def> <method-body>
            | <method-header> <method-body>
<method-header> → method <name> () '\n'
            | method <name> ( <name-list> ) '\n'
<vars-def> → <attrs-def>
<method-body> → begin '\n' <body-stmts> end-method '\n'
<body-stmts> → <body-stmt>
            | <body-stmt> <body-stmts>
<body-stmt> → <prototype>
            | <attr>
            | <if>
            | <method-call>
            | return <name>
<prototype> → <name>._prototype = <name>
<attr> → <lhs> = <arg> '\n'
        | <lhs> = <arg-bin> <op> <arg-bin> '\n'
<op> → + | - | * | /
<lhs> → <name>
        | <name>.<name>
<arg> → <number>
        | <name>
        | <name>.<name>
        | <method-call>
        | <obj-creation>
<arg-bin> → <name>
<obj-creation> → new <name> '\n'
<method-call> → <name>.<name> () '\n'
            | <name>.<name> ( <name-list> ) '\n'
<if> → if <name> <cmp> <name> then '\n' <if-stmts> end-if '\n'
        | if <name> <cmp> <name> then '\n' <if-stmts> else '\n' <if-stmts> end-if '\n'
<cmp> → eq | ne | gt | ge | lt | le
<if-stmts> → <if-stmt>
            | <if-stmt> <if-stmts>
<if-stmt> → <prototype>
            | <attr>
            | <method-call>
            | return <name>
<main-body> → begin '\n' <vars-def> <main-stmts> end '\n'
            | begin '\n' <main-stmts> end '\n'
<main-stmts> → <main-stmt>
            | <main-stmt> <main-stmts>
<main-stmt> → <prototype>
            | <attr>
            | <if>
            | <method-call>
<name> → sequência de letras maiúsculas ou minúsculas (sem números ou caracteres especiais)
<number> → número inteiro 32-bits (positivo/negativo)

```