LINGUAGENS E PARADIGMAS DE PROGRAMAÇÃO
TRABALHO EM GRUPO
2022/2

O trabalho se baseia na implementação de um interpretador para uma linguagem orientada a objetos (bol – *bruno's object-oriented language*).

A linguagem permite a definição de classes, com seus métodos e atributos. O programa inicia a execução no bloco principal. Exemplo da linguagem:

```
class Base
  vars id

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

class Pessoa
  vars num

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

begin
  vars p, b

  b = new Base
  p = new Pessoa
  p._prototype = b

  b.id = 111

  p.num = 123
  p.id  = 321

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

Execução do interpretador

O seu interpretador receberá um arquivo como parâmetro, contendo um programa na linguagem definida e deverá executá-lo. Por exemplo:

```
$ lua interpretador.lua  prog.bol
```

Considerações sobre a sintaxe da linguagem

- Nomes de métodos, variáveis, atributos e classes não podem utilizar palavras reservadas da linguagem:
  - class, method, begin, self, vars, end, if, return, eq, ne, lt, le, gt ge, new, io[1]
- Pode haver espaços no início da linha, no final das linhas ou entre os elementos da linguagem.
- Pode haver linhas em branco, que devem ser ignoradas.

Considerações sobre a semântica da linguagem

- Variáveis e atributos são inicializadas com o valor inteiro 0 (zero).
- Atributos, variáveis e parâmetros podem armazenar números inteiros ou referência para objetos.
- O valor dos parâmetros pode ser alterado (como se fossem variáveis).
- Um método sempre deve retornar um valor (inteiro ou objeto).
- Um método pode acessar seus parâmetros e variáveis diretamente, mas os atributos do objeto devem ser acessados via variável *self*.

Classe

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

Exemplo:

```
class Base
   vars a, b, c

   method calc()
      io.print(self.a)
      return 0
  end-method

end-class
```

```
class Frutas

   method info(a, b)
      io.print(a)
      b = a + 10
      return b
  end-method

end-class
```

Expressão de atribuição

A linguagem permite atribuição para alterar parâmetros, variáveis 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.

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
```

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”.

Condicional

A linguagem possui o seletor “if-then” / “if-then-else”. O teste pode ser igual eq, ne, gt, ge, lt, le (igual, não-igual, maior, maior-igual, menor, menor-igual). O teste só pode conter a comparação entre variáveis ou parâmetros (não atributos). O corpo do bloco do “if” pode conter atribuições, chamada de método, meta-ação ou retorno (não permite “if” aninhado).

Exemplo:

```
a = 1024
b = 128

if a eq b then
   c = b + a
else
   c = 123
end-if
```

```
a = 1024
b = obj.num

if a gt b then
   c = obj.calc(a, b)
   obj.func()
   io.print(c)
end-if
```

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 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, o objeto apontado pelo atributo “_prototype” deve ser inspecionado e o método invocado, caso exista. Senão, o atributo “_prototype” deve ser inspecionado no novo objeto. O processo continua até localizar um objeto que implemente o método (assuma que sempre haverá um objeto que implemente o método).

Quando um atributo for acessado em um objeto (leitura ou alteração), o atributo primeiro é procurado no objeto. Caso afirmativo, 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.

Variável *self*

Todo método possui um 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.

Objeto *io*

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

- io.print(n): recebe uma variável inteira, mostrando seu valor na tela, seguido de ‘\n’. Esse método retorna 0 (zero).
- io.dump(o): recebe um objeto e exibe na tela o código fonte atual da classe desse objeto. Esse método returna 0 (zero).

O objeto “io” não pode ser alterado (tanto por atribuição, meta-ações ou herança), sendo usado apenas para exibir valores na tela.

Meta-ações

As *meta-ações* permitem alterar o corpo de métodos da classe. A primeira linha dos métodos é considerada como a linha 1.

As meta-ações são:

- _insert(n): <string>

  Insere uma nova linha contendo <string> na posição “n” do método, deslocando as linhas para baixo. Se “n” for 0 (zero), insere a linha no final do método.

- _replace(n): <string>

  Substitui a linha “n” do método por <string>.

- _delete(n):

  Apaga a linha “n” do método, deslocando as linhas para cima.

Exemplo:

```
class Config
   method teste()
      vars x
      x = 123
   end-method
end-class

begin
   vars k
   k = new Config
   io.dump(k)
   Config.teste._replace(2):  x = 1024
   Config.teste._insert(0):   io.print(x)
   io.dump(k)
end
```

Saída:

```
class Config
  method teste()
    vars x
    x = 123
  end-method
end-class

class Config
  method teste()
    vars x
    x = 1024
    io.print(x)
  end-method
end-class
```

Bloco principal

O bloco principal é definido após as classes, sendo é o primeiro trecho de código a ser executado. O bloco pode definir variáveis e o código pode conter condicional, atribuição, chamada de método e meta-ações. O bloco não retorna valor.

```
begin
   vars a, b, c

   a = new Base
   b = 1023
   c = b * a.num

   io.print(c)
end
```

```
begin
   vars obj
   Base.showid._insert(2): a = x * 2
   obj = new Base
   io.dump(obj)
end
```

Regra do Trabalho

- Entrega do trabalho: 22/jan/2022, 23:55 (via Moodle)
  - Código fonte Lua (utilizem Lua 5.4 para divisão inteira).
- Trabalho deve feito em grupo de 3 alunos.
- Caso tenha dúvida sobre alguma parte, consulte o professor.
  - Não vá inventar da sua cabeça.
  - Pode ser que a especificação esteja incompleta ou inconsistente.
- Plágio significa nota 0 (zero) para todos os envolvidos.
- A nota irá considerar boas práticas de programação: qualidade, organização do código, comentários (relevantes), etc.
- O trabalho, depois de entregue, deve ser apresentado ao professor, em horário que será agendado com o grupo.
  - Não apresentar o trabalho significa nota 0 (zero) para o grupo.
- A implementação do interpretador poderá utilizar somente será permitido o uso dos módulos que são oferecidos por padrão na linguagem Lua (string, math, io, os, coroutine).

BNF da Linguagem

```
<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>
            | <meta-action>
            | 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> → <attr>
          | <method-call>
          | <meta-action>
          | return <name>

<meta-action> → <name>.<name>.<meta-op> (<line-number>) : ‘\n’
              | <name>.<name>.<meta-op> (<line-number>) : <string-no-nl> ‘\n’

<meta-op> → _insert | _replace | _delete

<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> → <attr>
            | <if>
            | <method-call>
            | <meta-action>
```

<name> → *sequência de letras maiúsculas ou minúsculas (sem números ou caracteres especiais)*

<number> → *número inteiro (positivo/negativo)*

<string-no-nl> → *sequência de caracteres sem* new line *(‘\n’)*