MAC0316/5754
Exercício Programa 3
Data de entrega: 09/12/2024

Instruções:

1. Você deve entregar seu programa pelo PACA em um **único arquivo** *.rkt* contendo as definições do interpretador.
2. **Programas atrasados terão uma penalidade de 20% na pontuação <u>POR DIA</u>.**

Interpretador (10 pontos)

Nesta parte vamos implementar uma versão da linguagem **Smalltalk**. As grandes modificações serão na **linguagem de entrada** e no **processo de resolução de métodos**. Utilizaremos resolução *dinâmica* de métodos mas usaremos um padrão "hibrido" onde inteiros e suas operações são primitivos.
ATENÇÃO: este programa é BEM mais extenso que os anteriores, você deve começar bem antes para que consiga entregar.

Sintaxe da linguagem (descrita como uma gramática):

```
<input>      --> <expression> | <class-def>


<class-def>  --> (class <class-id> <inst-var> <method-def> <method-def>)
       => nome da classe adicionado com "let"
       => Classe "Object" inicalizada no ambiente global
<class-id>   --> um identificador (para nome da super classe)
<inst-var>   --> um identificador (para nome da variável de instância)


<method-def> --> <regularMethodDef> | <primitiveMethodDef>
<regularMethodDef> -> (method <method-id> <method-arg> <exp>)
<primitiveMethodDef> -> (primitiveMethod <method-id> <integer>)
       ==>sem parametro, argumento vai direto ao código plai-typed
<method-id>  --> um símbolo identificador (para nome do método)
```

```
<method-arg> --> um símbolo identificador (para nome do parâmetro do
método)



<expression> --> <arith-exp> | <if-exp> | <seq-exp> | <set-exp> | <let-
exp> |

                 <new-exp> | <send-exp> | <value>

<arith-exp>  --> (<arith-op> <expression> <expression>)

<arith-op>   --> + | - | * | /

<if-exp>     --> (if <expression> <expression> <expression>)

<seq-exp>    --> (seq <expression> <expression)

<set-exp>    --> (set! <inst-var> <value>) | (set! <method-arg> <value>)

<let-exp>    --> (let <id>  <value>)

<new-exp>    --> (new <class-id> <value>)

<send-exp>   --> (send <expression> <method-id> <value>)



<value>      --> <integer> | <object> | <class> | <method>
```

a) **(2 pontos)**: Acrescente o valor `classV` no interpretador. Implemente o comando `class`, **parte da linguagem**, que possui 4 parâmetros: nome da classe pai, variável de instância, definição de método 1, e definição de método 2. Note que o `class` devolve um `classV`, cujo nome poderá ser registrado no ambiente usando o comando `let`.

b) **(4 pontos):** Substitua o valor `closV` pelo valor `methodV`, que é semelhante aos fechamentos mas possui um nome (para envio de mensagens) e **não guarda um ambiente**. Troque `lamC` e `lamS` por primitveMethodC, primitiveMethodS, regularMethodC e regularMethodS, para substituir o comando lambda por method. Note que `methodV` será um componente de `classV` quando o usuário definir uma classe. `methodV` terá acesso a um ambiente (que incluirá o atributo de instância da classe) apenas quando o método for chamado. **O valor methodV conterá dois campos, o nome do método e a sua definição**. Você deve definir o tipo MessageDefinition para poder lidar com mensagens regulares e primitivas:

```
(define-type MethodDefinition
```

[regularMethod (par : symbol) (body : ExprC)]
[primitiveMethod (num : number)]
)

Note que isso é um valor do INTERPRETADOR não ermos primitiveMethodV e regularMethodV, apenas methodV, que contém um MethodDefinition.

c) **(1 pontos):** Acrescente o valor `objetcV`. Pense bem quais devem ser os campos deste valor (lembrando que você deve fazer a busca de métodos e que você pode ter várias variáveis de instância…).

d) **(2 Pontos)** Você dever inicializar o ambiente global com a classe *Object*. Esta classe deve implementar o método "mensagemDesconhecida", que tem um simV como parâmetro. Esta mensagem deve ser "primitiva", chamando um erro de interpretador com a mensagem: "Erro: mensagemDesconhecida:…". Onde "…" indica o nome da mensagem, ou seja, é o conteúdo do simV.

e) **(2 pontos)**: Implemente o operador primitivo `new` , que recebe como parâmetro o **nome de uma classe** e um **valor**, e cria um `objetcV` daquela classe cujo atributo é inicializado com o valor. Lembre-se que a criação de um objeto da classe filha deve envolver um objeto com variáveis de instância da classe em questão bem as variáveis associadas a classes ancestrais (cujos atributos podem ser inicializados com o valor nullV). Além disso, lembre que o objeto filho usará esses objetos para acessar atributos de instância e métodos das classes ancestrais. Em minha solução fiz uma função auxiliar para gerar um ambiente de variáveis de instância que recebia uma classe como parâmetro.

f) (4 pontos) Você deve substituir a antiga appC por sendC. Esta nova expressão deve fazer o "methodLookup", tentando percorrer as classes e superclasses e verificar se existe um método com o nome indicado na expressão. Na minha implementação eu fiz uma rotina auxiliar separada, "methodLookup", que recebe uma classe e um nome de função (um símbolo) e procura a definição do método. Ela pode retornar um methodV, caso encontre, ou um nullV, caso não encontre. Para isso uso uma funçãozinha recursiva interna definida co letrec. Na interpretação de sendC, quando não encontramos um método devemos então interpretar um envio da mensagem 'mensagemDesconhecida ao

objeto original, utilizando como argumento um quoteC com o nome da mensagem. **(CUIDADO, este é uma das partes mais complexas do EP!!)**

g) (2 pontos). Implemente "métodos primitivos". Neste caso a expressão da mensagem deve ser (primitiveMethod XXX ZZZ), onde XXX é um símbolo com o nome da mensagem e ZZZ é um número *que identifica o índice do método no vetor de métodos primitivos*. No seu interpretador iremos registrar primitivas por número, onde cada primitiva será uma expressão lambda DO PLAI-TYPED (ou seja do interpretador) com um código arbitrário. Esta expressão terá um argumento que será o argumento usado no envio da mensagem). Note que este tipo de método permitiria que fizéssemos uma linguagem mais "pura", pois poderíamos implementar todas as operações como aritmética e if na forma de primitivas. Mas vocês devem se limitar à mensagem primitiva "mensagemDesconhecida", descrita abaixo. No meu interpretador eu coloquei uma variável global "PrimitiveMessageVector", um vetor com as definições de funções primitivas (lambdas que recebem um argumento e fazem alguma ação). Veja abaixo a definição de meu vetor:

```
(define PrimitiveMethodVector
      (make-vector 2 (lambda ([ x : Value] ) : Value
                        (error 'primitive "invalid primitive method")))); 0
;add primitive 1 for 'mensagemDesconhecida
(vector-set! PrimitiveMethodVector 1
             (lambda ([methodName : Value])
               (type-case Value methodName
                 [symV (symbolValue)
                  (error 'messaging
                         (string-append "mensagemDesconhecida:"
                                        (symbol->string symbolValue)))]
                 [else (error 'wrongArgument
                              "Wrong Argument: primitive 1 should receive a symV")])))
```

((vector-ref PrimitiveMethodVector 1) parametro)

h) Você devem implementar uma variável implícita (não declarada) ***self.*** Esta variável pode ser utilizada no código dos programas e se refere ao objeto que está recebendo a mensagem. Isso permite que um método da classe chame outro método da mesma classe, ou de uma classe derivada. Para isso, toda vez que executar um método, além de

acrescentar ao ambiente do objeto o valor do argumento, deve também acrescentar ‘self, apontando para o próprio objeto.

Note que toda classe possui 2 métodos. Isso permitirá testar a redefinição de métodos da classe pai.

IMPORTANTE: Agora seu interpretador deve ter uma fase de inicialização, onde é criado o ambiente global para classes, a classe Object e seu método primitivo, e uma estrutura de “métodos primitivos” que associa um número a um código de plai-typed com um argumento (veja acima meu exemplo).

IMPORTANTE: Não será dado **crédito parcial** aos itens acima. Apenas aqueles que funcionarem receberão pontos.

IMPORTANTE: Você deve usar a versão `plai-typed` (**não** `plai`) do interpretador *ep3Esqueleto.rkt* (disponibilizado com este enunciado). Neste interpretador já implementei o let e o valor nullV, para inicializar variáveis.

DICA: Pense bem nas características que você precisa implementar e teste todas elas. Acesso a variáveis de instância tanto da classe do objeto como associadas a super classes, troca de valores de variáveis de instância, redefinição de métodos simples, envio de mensagem inexistente, definição do método “mensagemDesconhecida” em uma classe de usuário para ver se código executado é o da nova classe e não de “Object”. O desenho dos testes faz parte do EP, mas vocês podem trazer dúvidas nas aulas.

EXEMPLO DE CODIGO:

```
(interpS '(let classe1 (class Object i
                             (regularMethod m1 x i)
                             (regularMethod m2 x (send self m1 x)))
    (let classe2 (class classe1 j
                        (regularMethod m1 x (quote subclassregularMethod))
                        (regularMethod m3 y y))
        (let object2 (new classe2 200)
            (send object2 m2 55)))))
```