Terceiro trabalho prático de Linguagens de Programação

Implementação da linguagem CLASSES na linguagem Racket

Sob orientação do Professor Leonardo Vieira dos Santos Reis

Departamento de Ciência da Computação
Universidade Federal de Juiz de Fora
Minas Gerais - Brasil
ERE 2021.3

Sumário

1 Especificação técnica do trabalho - Linguagem CLASSES

O presente documento tem por objetivo apoiar o projeto desenvolvido durante o terceiro trabalho da disciplina DCC019 - Linguagens de Programação.

O programa foi implementado na linguagem *Racket* a partir da linguagem de exemplo *IREF* mostrada pelo professor em aula durante o curso. Todo o código fonte foi inserido em um arquivo único nomeado *classes.rkt*.

As citações dentro do texto estão organizadas de forma que as referências internas encontram-se entre parênteses e as externas entre colchetes. Todas elas são clicáveis de forma a encaminhar a navegação para o ponto exato do texto onde se encontram.

As caixas contendo código fonte foram feitas utilizando em suas representações a sintaxe descrita em [3] e com a forma de colorização (*highlighting*) baseada no manual [2] e no github [6] da linguagem *Racket* (as cores utilizadas diferem em prol de uma melhor visualização e leitura deste documento).

1.1 Execução do programa

Abaixo seguem detalhes importantes para a correta execução do programa.

1.1.1 Pré-requisitos

Abaixo estão listados os pré-requisitos para o correto funcionamento do programa. Por favor, tenha certeza de que os atende para a correta execução do programa.

- Sistema operacional GNU/Linux ou Windows (10 ou superior)
- Compilador REPL[5] (versão 8.3 ou superior)
- IDE DrRacket[1] (opcional, para edição de código)

1.1.2 Como carregar o programa

O processo de carregamento do programa no sistema é bem simples: basta obter uma cópia do arquivo *classes.rkt* e então abrí-lo usando alguma IDE ou o ambiente interativo, como por exemplo pelo comando abaixo (via terminal):

```
$ racket
```

Então, para carregar o arquivo no REPL, executar:

```
> (enter! "classes.rkt")
```

E o programa iniciará sua execução.

1.2 Funções, módulos e bibliotecas padronizadas

Esta seção descreve, muito brevemente, algumas das funções *Racket built-in* provenientes do compilador REPL[7] e outras que foram aproveitadas da linguagem *IREF* que foram utilizadas no programa.

1.2.1 Funções do pacote *racket*

Algumas funções como *define*, *cons*, *make-vector*, *if*, *list*, entre outras, foram importadas diretamente do pacote *racket*[4], que já inclui a implementação de várias funções e as disponibiliza para programas escritos na linguagem *Racket*.

1.2.2 Funções de *IREF*

A linguagem *IREF* (*indirect reference* ou referência indireta, em tradução livre) foi apresentada pelo professor durante o curso. Ela foi mostrada com o objetivo de demonstrar como poderia ser implementada uma linguagem de referência indireta utilizando a sintaxe de *Racket*.

De *IREF*, foram aproveitadas as funções relacionadas à funcionalidade que implementa uma ideia de "memória"no programa como, por exemplo, *empty-store*, *newref*, *deref* e *setref!* .

2 Aspectos de implementação

Durante o processo de desenvolvimento do programa, alguns aspectos e características mais importantes foram especificados pelos desenvolvedores e, após serem destacados, foram então representados e detalhados nesta seção.

2.1 Decisões de projeto

Nesta seção estão descritas algumas das decisões de projeto tomadas durante a implementação da linguagem CLASSES no programa implementado em *Racket*.

2.1.1 Representação da memória

Por meio da estrutura de vetor (*vector*) de *Racket*, foi criada a noção de memória de estado, onde as variáveis dos programas da linguagem CLASSES foram alocadas em tempo de execução. Para esse fim, foi alocado um vetor com tamanho igual a 1000 unidades de comprimento, ou seja, poderiam ser criados e salvos até 1000 elementos (atributos ou métodos) na "memória", além é claro de abrir-se a possibilidade de se alterar algum estado dentro dela.

2.1.2 Representação das classes

As classes são representadas na forma de uma lista onde cada elemento é uma classe. Por sua vez, o primeiro elemento de uma classe é o par (*NomeDaClasse NomeDaClassePai*) e o segundo elemento é uma outra lista com cada atributo da classe associado ao valor 0.

2.1.3 Representação dos atributos

Os atributos das classes são inseridos numa lista, onde todos eles são associados ao valor 0 por padrão. Nas instâncias, o valor de cada atributo é colocado na memória de estados e é guardada a posição na memória, onde há o valor do atributo na lista que é retornada pela avaliação da palavra chave *new*.

2.1.4 Representação dos objetos

Os objetos são representados nos estados da memória. Cada variável de um objeto tem seu valor definido na memória também. Para manipular os atributos do objeto dentro do programa principal, retornamos uma lista com cada atributo do objeto seguido de seu índice na memória de estados.

2.1.5 Representação dos métodos dos objetos

Os métodos foram criados de forma semelhante ã criação dos atributos das classes. Como ocorreram problemas ao fazer os métodos reconhecerem por qual instância eles foram chamados, essa funcionalidade foi removida da versão final do projeto.

2.1.6 Representação dos construtores dos objetos

Ocorreram dificuldades ao definir e executar os métodos. Pelo mesmo motivo descrito na subseção acima, não foram criados construtores para as classes.

2.1.7 Representação dos ambientes (*environments*)

Apesar do professor ter mostrado exemplos usando a abordagem de *environments* implementados de forma procedural, a representação dos *environments* de execução do programa foi feita por meio da abordagem de operações com listas. O ambiente, então, é constituído a partir da lista das classes no momento que a função *value-of* (3) avalia a expressão contendo o tipo *classes*.

2.1.8 Representação dos programas

Os programas na linguagem CLASSES foram pensados para serem escritos da seguinte forma:

(define progName '(main_prog (classes (class progClassA extends progClassB (**fields**) *[...]*)) (**progBody**)))

Seguindo a mesma representação, é possível relacionar o programa com a avaliação do *type main_prog* feita pela função *value-of* (3):

[(equal? type 'main_prog) (value-of (caddr exp) (value-of (cadr exp) Δ))]

Legenda das cores na representação dessa subseção:

- verde: representa a *keyword* da linguagem *Racket*
- vermelho: representa os nomes (do tipo *string*) das variáveis
- **roxo**: representa o conteúdo variável (onde um ou mais de um componenete deve entrar)
- azul: representa os tipos reconhecidos pela linguagem CLASSES
- fundo amarelo: representa a definição das classes do programa (sempre terminando por estender da classe *object*)
- fundo rosa: representa o corpo do programa a ser executado

2.2 Modelagem do problema

Nesta subseção é possível comentar que: de acordo com todo o conteúdo da seção 2.1, destaca-se que foram usados alguns elementos da linguagem *IREF*, outros da linguagem *Racket*, e, por fim, os demais foram personalizados para atender os requisitos da linguagem CLASSES. Assim, o objetivo de representar objetos, classes e a execução de programas em CLASSES foi alcançado.

2.3 Dificuldades enfrentadas

Durante o processo de desenvolvimento da solução, como era de se esperar, algumas dificuldades foram encontradas. As três maiores dificuldades enfrentadas neste trabalho, em ordem de tempo despendido para sua solução, foram:

1. Decidir como implementar a criação e a chamada dos métodos das classes;
2. Decidir sobre como representar as instâncias de maneira a encontrar a classe pai de forma rápida e, assim, encontrar os métodos da classe pai com *super*; e
3. Implementar a forma de representar as instâncias das classes para possibilitar a alteração dos valores dos atributos de forma eficiente.

A criação dos métodos e a herança entre as classes não foram implementadas. Ambos estavam demandando muito tempo e as tentativas não retornaram bons resultados. Por isso, na versão final, optou-se por não colocar métodos e herança entre as classes.

A forma de representar as instâncias foi inicialmente feita utilizando listas, mas a impossibilidade de alterar valores em uma lista nos fez utilizar a abordagem de estados da memória para criar os objetos.

3 Função auxiliar *value-of*

Nesta seção está descrita a implementação da função *value-of*, que avalia uma expressão dentro de um ambiente de execução e retorna seu valor. Ela funciona na prática, assim como uma função reconhecedora de tipos para a linguagem CLASSES e, então, possui a chamada e/ou assinatura das funções de *Racket* adaptadas para o contexto do programa, pensado para a linguagem CLASSES.

A função recebe o corpo do programa (expressão) e seu ambiente (*environment*). E, então, faz a avaliação do corpo por meio das funções *built-in* de *Racket*, ou outras personalizadas definidas de forma customizada.

Mais detalhes da execução dos programas de CLASSES podem ser vistos na seção (2.1.8)

Parâmetros de entrada da função:

- exp: Expressão a ser avaliada
- Δ: *Environment* (ambiente) de execução

Tipos identificados pela função:

- lit: Representa um número do tipo literal (int)
- var: Representa uma variável das funções a serem avaliadas
- dif: Representa uma função que calcula a diferença entre dois números
- zero?: Representa uma função que determina se um número é igual a zero ou não
- let: Representa a criação de um novo *environment*
- if: Representa uma função que implementa um condicional
- proc: Representa uma função que avalia um procedimento
- call: Representa uma função que aplica um procedimento (*procedure*) em um determinado valor
- letrec: Representa a criação de um novo *environment* de forma recursiva
- set: Representa uma função que altera o valor de uma posição da memória de estados
- begin: Representa uma função que permite executar múltiplas expressões de forma aninhada e retorna o valor da última expressão
- cons: Aplica a mesma operação feita pelo cons do *Racket* aos dois elementos passados como parâmetro

Tipos relacionados à linguagem CLASSES identificados pela função:

- classes: Representa uma função que adiciona todas as novas classes do programa, definindo o ambiente de execução do programa
- main_prog: Representa uma função que avalia o corpo do programa no ambiente definido pelo tipo *classes*
- new: Representa um construtor de um objeto
- set-val: Representa uma função que altera o valor de um campo (*field*) de objeto
- send: Representa uma função que, ou executa um método de um objeto, ou recupera o valor de um atributo do objeto na "memória"
- display: Representa uma função que passa um determinado elemento para ser impresso na tela

Código implementado em *Racket*:

```
(define (value-of exp Δ)
  (define type (car exp))
  (cond [(equal? type 'lit) (cadr exp)]
        [(equal? type 'var) (apply-env Δ (cadr exp))]
        [(equal? type 'dif) (- (value-of (cadr exp) Δ) (value-of (caddr exp) Δ))]
        [(equal? type 'zero?) (= (value-of (cadr exp) Δ) 0)]
        [(equal? type 'let) (value-of (cadddr exp)
                                      (extend-env (cadr exp) (value-of (caddr exp) Δ)  Δ))]
        [(equal? type 'if) (if (value-of (cadr exp) Δ)
                               (value-of (caddr exp) Δ) (value-of (cadddr exp) Δ))]
        [(equal? type 'proc) (proc-val (cadr exp) (caddr exp) Δ)]
        [(equal? type 'call) (apply-proc (value-of (cadr exp) Δ) (value-of (caddr exp) Δ))]

        [(equal? type 'letrec) (value-of (car (cddddr exp))
                                         (extend-env-rec (cadr exp) (caddr exp)
                                                         (cadddr exp) Δ))]

        [(equal? type 'set) (let ([v (value-of (caddr exp) Δ)])
                              (setref! (apply-env Δ (cadr exp)) v)
                              v)]

        [(equal? type 'begin) (foldl (lambda (e acc)
                                       (value-of e Δ))
                                     (value-of (cadr exp) Δ)
                                     (cddr exp))]
        [(equal? type 'classes) (add-classes (cdr exp) Δ)]
        [(equal? type 'main_prog) (value-of (caddr exp) (value-of (cadr exp) Δ))]
        [(equal? type 'new) (new-instance (cadr exp) Δ (get-addr-free))]
        [(equal? type 'set-val) (set-field (cadr exp) (caddr exp) Δ)]
        [(equal? type 'send) (send-field (cadr exp) Δ)]
        [(equal? type 'display) (display (cadr exp))]
        [(equal? type 'cons) (cons (value-of (cadr exp) Δ) (value-of (caddr exp) Δ))]

        [else (error "operação não implementada")]
        )
)
```

Listing 1: Código da função *value-of*

4 Detalhamento do código implementado

Nesta seção estão descritas as funções alvo do processo de avaliação dentro do programa, e que foram pensadas, implementadas e usadas para a completude dos objetivos presentes no documento de requisitos do trabalho.

4.1 Função *get-pai*

A função calcula e retorna qual é a classe pai de uma determinada classe. Essa função seria utilizada como parte da palavra *super* para buscar métodos na classe pai.

Parâmetros de entrada da função:

- nome: Representa o nome da classe filha
- env: Representa o ambiente (*environment*) de execução composto pela definição de todas as classes.

Código implementado em *Racket*:

```
(define (get-pai nome env)
(if (empty? env)
  (error "A classe pai não está definida!")
  (if (equal? nome (car env))
      (cadr env)
      (get-pai nome (cddr env))
   )
 )
)
```

Listing 2: Código da função *get-pai*

4.2 Função *empty-env-class*

A função cria um novo ambiente (*environment*) para os programas da linguagem CLASSES.

Parâmetros de entrada da função:

(Não possui parâmetros de entrada)

Código implementado em *Racket*:

```
(define empty-env-class empty)
```

Listing 3: Código da função *empty-env-class*

4.3 Função *extend-env-class*

A função estende o ambiente de execução para suportar mais uma definição de valor de variável.

Parâmetros de entrada da função:

- var: Representa a variável que será passada ao ambiente
- value: Representa o valor associado com a variável
- env: Representa o ambiente (*environment*) de execução

Código implementado em *Racket*:

```
(define (extend-env-class var value env)
  (list var value env)
  )
```

Listing 4: Código da função *extend-env-class*

4.4 Função *add-classes*

A função adiciona todas as classes de um programa no *envrironment* de execução vazio. Este ambiente vazio é definido na seção 4.3. E então, a classe *object* é criada.

Parâmetros de entrada da função:

- cls: Representa a lista de classes definidas pelo usuário
- env: Representa o ambiente (*environment*) de execução em que as classes serão criadas.

Código implementado em *Racket*:

```
(define (add-classes cls env)
  (if (empty? cls) env
  (list (value-of-class (car cls) env) (add-classes (cdr cls) env) '(object)) ))
```

Listing 5: Código da função *add-classes*

4.5 Função *value-of-class*

A função associa o nome da classe e seu pai com as definições da classe passadas pelo usuário. Ela é chamada uma vez para cada classe definida pelo usuário. Essa função associa o par (n*omeDaClasse nomeDaClassePai*) à lista de definições retornada pela função *value-of-definition*.

Parâmetros de entrada da função:

- cls: Representa uma das classes definidas pelo usuário
- env: Representa o ambiente (*environment*) de execução em que serão criadas as classes.

Código implementado em *Racket*:

```
(define (value-of-class cls env)
  (if (and (equal? (car cls) 'class) (equal? (caddr cls) 'extends))
      (extend-env-class (list (cadr cls) (cadddr cls))
                        (value-of-definition (cddddr cls) (cadddr cls) env) env)
      (error "Erro nas palavras chave classe ou extends"))
  )
```

Listing 6: Código da função *value-of-class*

4.6 Função *value-of-definition*

A função associa todos os atributos definidos com o valor nulo e retorna essa lista para a função *value-of-class*. Associa a palavra *fields* com a lista de atributos da classe.

Parâmetros de entrada da função:

- cls: Lista dos atributos da classe
- pai: Nome da classe pai
- env: Representa o ambiente (*environment*) de execução

Código implementado em *Racket*:

```
(define (value-of-definition cls pai env)
(if (empty? cls) env
  (extend-env-class 'fields (value-of-fields (car cls) pai env) env))
  )
```

Listing 7: Código da função *value-of-definition*

4.7 Função *value-of-fields*

A função passa por cada atributo da classe e estende o ambiente, colocando nele o atributo associado ao valor 0. Esse processo é executado de forma recursiva até que se acabem os atributos da classe.

Parâmetros de entrada da função:

- cls: Lista com os atributos da classe
- pai: Nome da classe pai
- env: Representa o ambiente (*environment*) de execução

Código implementado em *Racket*:

```
(define (value-of-fields cls pai env)
  (if (empty? cls) '()
      (extend-env-class (car cls) 0 (value-of-fields (cdr cls) pai env)) ))
```

Listing 8: Código da função *value-of-fields*

4.8 Função *new-instance*

A função cria uma nova instância de uma classe na memória de estados do programa. A função percorre o ambiente em que estão definidas as classes, e passa o vetor de atributos da nova classe de modo a criar uma nova instância para as funções *insert-in-memory* e *return-instance*. A função, então, retorna para o corpo principal do programa o que for obtido da função *return-instance*.

Parâmetros de entrada da função:

- obj: Representa o nome da classe para a qual deve ser criada uma nova instância
- env: Representa o ambiente (*environment*) de execução que contém as classes já definidas
- addr-free: Primeira posição livre da memória de estados

Código implementado em *Racket*:

```
(define (new-instance obj env addr-free)
  (if (empty? env) (error "Classe inexistente!\nNao foi possivel instanciar o objeto")
      (if (equal? obj (caaar env))
          (begin (insert-in-memory (cadr (cadar env)))
                 (return-instance (cadr (cadar env)) addr-free))
          (new-instance obj (cadr env) addr-free))
  )
)
```

Listing 9: Código da função *new-instance*

4.9 Função *return-instance*

A função retorna para *new-instance* uma lista com o mapeamento de cada atributo e sua posição na memória de estados. É esta função que torna possível a manipulação dos atributos no corpo principal do programa. Cada chamada desta função inclui o par *nomeDoAtributo posiçãoNaMemoria* numa lista. O valor da primeira posição livre na memória é calculado antes da execução da função *insert-in-memory*, logo, ambas as funções *insert-in-memory* e *return-instance* vão apontar para as mesmas posições durante a criação e possíveis manipulações das referências.

Parâmetros de entrada da função:

- cls: Vetor de atributos da classe para a qual deve ser criada uma nova instância
- addr-free: Primeira posição livre da memória de estados

Código implementado em *Racket*:

```
(define (return-instance cls addr-free)
  (if (empty? cls) '()
      (list (car cls) addr-free (return-instance (caddr cls) (+ addr-free 1)))
  )
)
```

Listing 10: Código da função *return-instance*

4.10 Função *insert-in-memory*

A função insere os atributos passados como parâmetro na memória de estados definida no escopo global.

Parâmetros de entrada da função:

- cls: Vetor de atributos da classe para a qual deve ser criada uma nova instância, que será incluído na memória por esta função

Código implementado em *Racket*:

```
(define (insert-in-memory cls)
  (if (empty? cls) (println "Instancia criada com sucesso!")
      (begin (newref (cadr cls)) (insert-in-memory (caddr cls)) )
    )
 )
```

Listing 11: Código da função *insert-in-memory*

4.11 Função *set-field*

A função altera o valor do campo *fld*, do objeto *env* para o valor *val*. O valor desse campo está armazenado em uma determinada posição na memória de estados. Essa posição é o valor associado a *fld* na lista *obj*.

Parâmetros de entrada da função:

- fld: Representa o campo (*field*) da instância *obj* a ser alterado
- val: Representa o valor a ser inserido na referência a *fld*
- obj: Instância de uma classe. Esse parâmetro é uma lista criada pela função *return-instance*, que mapeia os atributos da instância de acordo com a posição da memória em que cada atributo está.

Código implementado em *Racket*:

```
(define (set-field fld val obj)
  (if (empty? obj) (error "Valor nao existe nessa instancia")
      (if (equal? fld (car obj)) (setref! (cadr obj) val)
          (set-field fld val (caddr obj))
      )
  )
)
```

Listing 12: Código da função *set-field*

4.12 Função *send-field*

De forma bem semelhante à função *set-field* definida em 4.11, essa função retorna o valor de *fld* na memória de estados.

Parâmetros de entrada da função:

- fld: Representa o campo (*field*) do qual se deseja saber o valor
- cls: Representa o objeto que contém a referência da posição da memória

Código implementado em *Racket*:

```
(define (send-field fld cls)
  (if (empty? cls) (error "O campo escolhido nao é uma variável nem um método")
      (if (equal? fld (car cls)) (deref (cadr cls))
          (send-field fld (caddr cls))
      )
  )
)
```

Listing 13: Código da função *send-field*

5 Exemplos de funcionamento

Esta seção contém exemplos de funcionamento do programa que utiliza a linguagem CLASSES e de algumas de suas funções.

5.1 Exemplo 1 - Execução de um programa simples

Neste exemplo, é executado um programa que: cria duas classes (*classe1* e *classe2*) que possuem três parâmetros cada uma; cria uma nova instância de um objeto (*c1*) pertencente à classe *classe1*; e executa uma função que exibe uma mensagem na tela.

Código implementado em *Racket*:

```
(define exemploDisplay '(main_prog
                         (classes (class classe1 extends classe2 (a b c))
                                  (class classe2 extends object (d e f)))
                                  (let c1 (new classe1) (display "\nFim do primeiro exemplo\n") )
                        )
)
```

Listing 14: Código do primeiro exemplo

Resposta esperada:

```
"Instancia criada com sucesso!"

Fim do primeiro exemplo
```

5.2 Exemplo 2 - Execução da criação de um objeto numa classe inexistente

Neste exemplo, é executado um programa que: cria duas classes (*classe1* e *classe2*) que possuem três parâmetros cada uma; cria uma nova instância de um objeto (*c1*) pertencente à classe *classeNaoDeclarada*; e executa uma função que altera o valor de *a* de *c1* para 2.

Código implementado em *Racket*:

```
(define exemploDeErroClasseNaoDeclarada '(main_prog
                                          (classes (class classe1 extends classe2 (a b c))
                                                   (class classe2 extends object (d e f)))
                                                   (let c1 (new classeNaoDeclarada) (set-val a 2 c1)
                                                   )))
```

Listing 15: Código do segundo exemplo

Resposta esperada:

```
(error "Nao existe essa classe!!")
```

Comentário sobre o erro:

O erro acima foi proposital e ocorreu pois o programa tentou criar a instância do objeto *c1* na classe *classeNaoDeclarada*, que era uma classe inexistente naquele contexto, e por isso a tentativa de criar uma instância do objeto falhou, culminando com a falha de execução do programa.

5.3 Exemplo 3 - Execução da criação de um objeto numa classe

Neste exemplo, é executado um programa que: cria duas classes (*classe1* e *classe2*) que possuem três parâmetros cada uma; cria uma nova instância de um objeto (*c1*) pertencente à classe *classe1*; e executa uma função que altera o valor de *a* de *c1* para 2.

Código implementado em *Racket*:

```
(define mudarCampo '(main_prog
                     (classes (class classe1 extends classe2 (a b c))
                              (class classe2 extends object (d e f)))
                              (let c1 (new classe1) (set-val a 2 c1)
                              )))
```

Listing 16: Código do terceiro exemplo

Resposta esperada:

```
"Instancia criada com sucesso!"
```

5.4 Exemplo 4 - Execução da criação e recuperação de um campo de um objeto

Neste exemplo, é executado um programa que: cria duas classes (*classe1* e *classe2*) que possuem três parâmetros cada uma; cria uma nova instância de um objeto (*c1*) pertencente à classe *classe1*; executa uma função que altera o valor de *a* de *c1* para 5; e então consulta essa alteração e imprime o resultado na tela.

Código implementado em *Racket*:

```
(define pegaCampo
  '(main_prog (classes (class classe1 extends classe2 (a b c))
                       (class classe2 extends object (d e f)))
              (let c1 (new classe1)
                (begin (set-val a 5 c1)
                       (display "\nValor de a na intancia c1: ")
                       (send a c1) ))))
```

Listing 17: Código do quarto exemplo

Resposta esperada:

```
"Instancia criada com sucesso!"

Valor de a na intancia c1: 5
```

5.5 Exemplo 5 - Execução da criação de uma classe extendida de outra

Neste exemplo, é executado um programa que: cria uma classe *classe1* que estende a classe *objeto* e possui três atributos.

Código implementado em *Racket*:

```
(define criacaoDeClasse '(classes (class classe1 extends object (a b c)) ))
```

Listing 18: Código do quinto exemplo

Resposta esperada:

```
'(((classe1 object) (fields (a 0 (b 0 (c 0 ())))) ()) ()) () (object))
```

5.6 Exemplo 6 - Execução da subtração entre dois números armazenados em *fields*

Neste exemplo, é executado um programa que: cria duas classes (*classe1* e *classe2*) que possuem três parâmetros cada uma; cria uma nova instância de um objeto (*c1*) pertencente à classe *classe1*; executa uma função que altera os valores de *a* e de *b* de *c1* para 5 e 1, respectivamente; e então subtrai esses valores e imprime o resultado na tela.

Código implementado em *Racket*:

```
(define subtraiCampos
  '(main_prog (classes (class classe1 extends classe2 (a b c))
                       (class classe2 extends object (d e f)))
              (let c1 (new classe1)
                (begin (set-val a 5 c1) (set-val b 1 c1)
                       (display "\nDiferença entre os valores 'a' e 'b' na intancia c1: ")
                       (dif (send a c1) (send b c1))
                       ))))
```

Listing 19: Código do sexto exemplo

Resposta esperada:

```
"Instancia criada com sucesso!"

Diferença entre os valores 'a' e 'b' na intancia c1: 4
```

5.7 Exemplo 7 - Execução da criação de instâncias de dois objetos

Neste exemplo, é executado um programa que: cria duas classes (*classe1* e *classe2*) que possuem três parâmetros cada uma; cria uma nova instância de um objeto pertencente à classe *classe1*; cria uma nova instância de um objeto pertencente à classe *classe2*; e então imprime na tela o conteúdo da última expressão avaliada.

Código implementado em *Racket*:

```
(define visualizaObjeto
  '(main_prog (classes (class classe1 extends classe2 (a b c))
                       (class classe2 extends object (d e f)))
              (cons (new classe1) (new classe2)) ))
```

Listing 20: Código do sétimo exemplo

Resposta esperada:

```
"Instancia criada com sucesso!"
"Instancia criada com sucesso!"
'((a 12 (b 13 (c 14 ()))) d 15 (e 16 (f 17 ())))
```

Comentário sobre a execução:

O exemplo acima cria duas classes, por esse motivo ele imprime duas vezes a mensagem de criação de instância bem sucedida.

6 Referências

[1] Tim Colburn. *DrRacket Installation and Setup*. Acesso em 17 de Fevereiro de 2022. URL: https://www.d.umn.edu/~tcolburn/cs1581/labs/setup/.

[2] Racket Docs. *Syntax highlighting*. Acesso em 10 de Fevereiro de 2022. URL: https://docs.racket-lang.org/pollen/mini-tutorial.html.

[3] Racket Docs. *Syntax Model*. Acesso em 10 de Fevereiro de 2022. URL: https://docs.racket-lang.org/reference/syntax-model.html.

[4] Racket Docs. *The Racket Reference*. Acesso em 10 de Fevereiro de 2022. URL: https://docs.racket-lang.org/reference/.

[5] Racket Language. *Download Racket*. Acesso em 10 de Fevereiro de 2022. URL: https://download.racket-lang.org/all-versions.html.

[6] Racket. *syntax-color family of packages*. Acesso em 10 de Fevereiro de 2022. URL: https://github.com/racket/syntax-color.

[7] Beautiful Racket. *The REPL*. Acesso em 10 de Fevereiro de 2022. URL: https://beautifulracket.com/explainer/repl.html.