1 Programação Relacional em Scheme

Uma pessoa só tem certeza
daquilo que constrói

Giambattista Vico -
Historiador italiano, século XVIII

Neste capítulo, veremos uma linguagem de programação lógica com um sabor diferente do Prolog que vimos no início, chamada *miniKanren*, e veremos como implementá-la, por meio da linguagem *Scheme*.

Usaremos *Scheme* porque é uma linguagem que é pequena, o que significa que não será gasto muito esforço para apresentar o seu funcionamento. E é, ao mesmo tempo, poderosa, o que significa que não precisaremos de muito código para fazer o que nos propomos.

Esse capítulo tem um sabor diferente dos demais. Essa diferença pode ser visto rapidamente, pela cara do código, mas também de várias outras formas. Redefiniremos alguns termos usados anteriormente. Essas redefinições terão semelhanças e diferenças às definições originais, mas escolhemos não notar essas diferenças aqui, por acreditarmos serem claras o suficiente. Outro ponto que vale nota é que, aqui, buscamos apenas um maior entendimento e, para tanto, tentamos deixar o código como "implementado em primeiros princípios", evitando detalhes que poderiam deixá-lo mais eficiente, mas elevando os prerrequisitos para a compreensão do código, ou exigindo uma discussão mais completa sobre ele (o que preferimos evitar).

1.1 Introdução ao Scheme

Scheme é um *Lisp*[*]. O termo *Lisp* é às vezes usado para ser referir a uma linguagem de programação, mas o mais correto seria ser referida como a uma família de linguagens (de fato, dezenas de linguagens), todas com algumas características em comum, notadamente:

- São linguagens multi-paradigma, mas como um foco no paradigma de programação funcional, isto é, funções são "cidadãos de primeira classe";
- "Todo"[†] código *Lisp* (que não tenha erro de sintaxe) é avaliado para algum valor;
- Programas são expressos em notação polonêsa, em formato de listas[‡](listas

[*] De *LISt Processing*
[†] Com uma excessão bem natural, que será mencionada mais adiante.
[‡] Também conhecidas como *S-Expressions*, ou *Sexps*

em *Lisps* são delimitadas por parênteses)*:
`(+ 1 2)` ⇒ 3;

- O que nos leva a outro ponto: não existe diferença sintática entre a forma como programas *Lisp* são escritos e a forma como suas estruturas de dados são representadas. Diz-se, assim, que *Lisps* são homoicônicas (vale dizer, Prolog também é uma língua homoicônica), o que significa que a diferença entre dados e programa é "borrada", e programas podem, e frequentemente são, manipulados livremente.

Agora, vamos rapidamente introduzir a sintaxe principal de *Scheme*, com alguns exemplos†:

- Listas são representadas como `(`el_1 el_2 ... el_n`)`, em que el_i é o iésimo elemento da lista. No entanto, se escrevêssemos uma lista simplesmente assim, a lista seria confundida com uma aplicação de função (aplicação da função el_1 aos argumentos el_2 a el_n), então, para fins de desambiguação, é usada uma aspa simples, e a lista é escrita como '($el_1 el_2 ... el_n$), que é equivalente a `(list` $'el_1$ ... $'el_n$`)` . Essa aspa simples também pode ser usada para "evitar que um objeto seja avaliado"‡:
  A linha
  `a` ⇒ erro
  resulta em erro se `a` não for uma variável, já que tentará ser avaliada, sendo que `a` não tem valor associado, mas a linha
  $'a \;\Rightarrow\; 'a$
  é avaliada como ela mesma.

- Uma estrutura de dados mais geral do que lista em *Scheme* é o que é chamado *cons pair* (que nós chamaremos daqui para frente simplesmente de "par").. A lista `'(a b c d)` é equivalente a `(cons a (cons b (cons c (cons d '()))))`, em que `'()` é a lista vazia Assim *cons* constrói uma estrutura de dados formada por um par. Para obter o primeiro elemento do par, usa-se *car* e, para obter o segundo, *cdr*§. Temos, por exemplo,
  `(car (cons 1 (cons 2 3)))` ⇒ 1
  `(cdr (cons 1 (cons 2 3)))` ⇒ `(cons 2 3)` = `'(2 . 3)`
  `(car '(1 2 3 4))` ⇒ 1
  `(cdr '(1 2 3 4))` ⇒ `'(2 3 4)`

*Usaremos daqui em diante a notação **codigo** ⇒ **valor** para denotar que o código **código** avalia para o valor **valor**.

†Note que só introduziremos a parte da linguagem que nos será relevante, o que não é a linguagem toda.

‡Uma colocação mais correta seria "tornar objetos auto-avaliantes", mas não precisamos entrar em muitos detalhes de como isso funciona. Para nós é suficiente dizer que 'a é um símbolo.

§Esses nomes têm uma motivação histórica: eram nomes de registradores quando os primeiros *Lisps* estavam sendo criados.

Com isso, podemos definir um lista indutivamente como sendo ou a lista vazia, `'()`, ou um par, cujo *cdr* é uma lista*.

- Para definir funções, use **lambda**, ou λ†:
  `((λ (a b) (/ a b)) 1 3)` ⇒ 1/3

- Para definir constantes, use **define**:
  `(define divide (λ (a b) (/ 1 3)))`‡
  `(divide 1 3)` ⇒ 1/3
  `(define c (divide 1 3))`
  `(divide (divide 1 9) c)` ⇒ 1/3

- Um ponto importante, que usaremos muito logo mais é que, se quisermos criar listas com os valores das variáveis, no lugar de nomes simbólicos, podemos usar, no lugar da aspa simples a crase e preceder o nome da variável com uma vírgula:
  `(define x 10)`
  `` `(1 2 'x ,x) `` ⇒ `'(1 2 'x 10)`

- Para realizar execuções condicionais, use **cond**:

```
(cond
  ((< 1 0) (+ 3 4))
  ((< 0 1) (- 3 4))
  (else 0))
```

  ⇒ -1

  Podem ser adicionadas quantas cláusulas do tipo `((condicao)(efeito))` se quiser (vale notar que elas são avaliadas sequencialmente), sendo que a última pode opcionalmente ser como `(else (efeito))`, ou `(#t (efeito))`§.

- Para adicionar variáveis locais, use *let*:

*Note que `(cons 2 3)`, por exemplo, não é uma lista. Esse tipo de estrutura é chamada *dotted list*, porque, para distingui-la de uma lista, é costumeiramente impressa como `'(2 . 3)`, mas, assim como com Prolog, é uma estrutura de dados diferente, que tem o nome *dotted **list*** como que por uma aparência acidental.

†Editores de texto atuais aceitam os dois tipos de entrada, mas optamos por usar λ. Esse uso do símbolo tem a seguinte origem: Bertrand Russel e Alfred Whitehead buscaram, no início do século XX lançar as bases lógicas da matemática eu seu trabalho *Principia Mathematica*. Lá, para denotar que uma variável é livre, ela recebia chapéu, como em $\hat{a}(a + y)$. Mais tarde, Alonzo Church achou que seria mais conveniente ter fórmulas crescendo linearmente na horizontal, então decidiu mover o chapéu para o lado, obtendo $\hat{}a(a + y)$. Mas o chapéu flutuando parece engraçado, então Church o trocou pelo o símbolo não usado mais próximo que tinha, um Λ, como em $\Lambda a(a + y)$. Mas Λ tem uma grafia muito parecida com outra letra comum, então ele acabou eventualmente trocando para λ em sua teoria, que acabou se chamando *cálculo* λ [2].

‡Um *açucar sintático* para essa construção é `(define (divide a b) (/ 1 3))`.

§Preferiremos a segunda opção. O *#t* é de *true* e existe, analogamente, o *#f*, de *false*.

```
(let ((a (+ 3 4))
      (b (cons 1 2)))
  (+ a (car b)))
```

$\Rightarrow$ 8

O *let* tem duas partes, a de definições, da forma `((variavel valor)(variavel valor) ... (variavel valor))`*. em seguida, a parte de valor, que nos dá o valor que *let* assume.

Como exemplo, o seguinte programa calcula o comprimento de uma lista (lembre que, como

Dada essa introdução, faremos uso dessas e outras construções da linguagem sem maiores comentários (exceto quando uma construção especialmente difícil ou complexa o justificar). Para uma introdução mais compreensiva à linguagem, veja [1].

1.2 A linguagem miniKanren

Nosso objetivo aqui é implementar miniKanren, uma linguagem de programação relacional. No lugar de descrever toda a linguagem e depois implementá-la, seguimos pelo caminho de mostrar um pequeno exemplo do que esperamos conseguir e, então, implementamos a linguagem. A esperança é que essa abordagem ofereça maior entendimento dos conceitos explorados.

O tipo de coisa que queremos poder fazer com miniKanren é como o seguinte†:

```
(defrel (teacupᵒ t)
  (disj₂ (≡ 'tea t) (≡ 'cup t)))
(run* x
  (teacupᵒ x))
```

$\Rightarrow$ `'(tea cup)`

*Podem ser adicionadas quantas variáveis se quiser. As atribuições são feitas "paralelamente" (o que significa que a atribuição de valor a uma variável não influi no da outra, o que pode ser feito de forma paralela, ou não).

†Usaremos as convenções da literatura de usar subescritos e sobrescritos e símbolos matemáticos, como $\equiv$, para representar as relações, na esperança de que isso clarifique a notação e deixe o texto menos pesado. Em particular, para diferenciar um objeto relacional de um funcional, o relacional terá um "o" sobrescrito (como em $relacao^o$). Ao escrever o programa para o computador ler, os sobrescritos e subescritos que forem alfa-numéricos ou "*" podem ser escritos normalmente, na frente do termo (como em $relacaoo$, ou $run*$). O símbolo $\equiv$ é escrito "==" e, termos como $termo^\infty$, "$termo - inf$". Ademais, #u e #s devem ser trocados por *fail* e *succeed.*, respectivamente.

Veremos mais exemplos enquanto o construímos. A construção a seguir é em grande parte baseada em [3]. Para conferir detalhes de implementações completas, veja [4].

Como visto no primeiro exemplo, não seguimos, como em Prolog, uma convenção de nomeação de variáveis (em Prolog, a convenção era de que variáveis são capitalizadas). Assim, precisamos de algo para discerni-las e, para tal fim, o que usamos é o *fresh*, informando que a variável é "fresca".

Lembre-se que uma variável relacional não é a a mesma coisa que uma variável em uma linguagem de programação tradicional (não relacional). Para uma variável única, usamos*

```
(define (var name)(vector name))
```

usamos também†

```
(define (var? name)(vector? name))
```

Para evitar problemas como os de colisão de variáveis, as variáveis são locais, assim como em *Scheme*. Precisamos, então, de uma forma de modelar isso (note que a definição acima não reflete isso) e, o que usamos é o seguinte:

```
(define (call/fresh name f)
  (f (var name)))
```

Essa função espera, como segundo argumento, uma expressão λ, que recebe como argumento uma variável e produz como valor um goal, que tem acesso à variável criada.

Precisamos, agora, saber como associar um valor a uma variável. Diremos que o par `` `(,z . a) `` é uma associação de a à variável z. Mais em geral, um par é uma associação quando o seu *car* é uma variável.

A uma lista de associações chamaremos *substituição*. Na substituição `` `((,x . ,z)) ``, a variável x é "fundida" (ou, na nossa linguagem anterior, unificada) à variável z. A substituição vazia é simplesmente uma lista vazia: `(define empty-s '())`. Nem toda lista de associações é uma substituição, no entanto. Isso porque, não aceitamos, em nossas substituições, associações com o mesmo *car*. Então, o seguinte não é uma substituição: `` `((,z . a)(,x . c)(,z . b)) ``.

Precisamos de dois procedimentos importantes para lidar com substituições: um para extênde-las e um para obter o valor de uma variável presente nela.

Para obter o valor associado a x, usamos *walk*, que deve se comportar como o seguinte:

*Usamos *vector* para que a unicidade da variável seja definida por sua posição de memória. Outra opção, seria distingui-las por valor, se nos assegurássemos que seu valor é único.

†Símbolos como "?" podem ser usados no meio do código asssim como outros, tais como "a" ou "b".

```
(walk y
  `((,z . a)(,x . ,w)(,y . ,z)))
```

⇒ *a*

porque *y* está fundido a *z*, que está associado a *a*. O *walk* é como se segue*:

```
(define (walk v s)
  (let ((a (and (var? v)(assv v s))))
    (cond
      ((pair? a)(walk (cdr a) s))
      (else v))))
```

Esse código faz uso de *assv*, que ou produz *#f*, se não há associação cujo *car* seja *v* na substituição *s*, ou produz o *cdr* de tal associação. Perceba que, se *walk* produz uma variável como valor, ela é necessariamente fresca (isto é, que não foi associada).

Para extender uma substituição, usamos *ext-s*, que faz uso de *occurs?*:

```
(define (ext-s x v s)
  (cond
    ((occurs? x v s) #f)
    (else (cons '(,x . ,v) s))))

(define (occurs? x v s)
  (let ((v (walk v s)))
    (cond
      ((var? v) (eqv? v x))
      ((pair? v)
        (or (occurs? x (car v) s)
            (occurs? x (cdr v) s)))
      (else #f))))
```

Esse *occurs?* realiza o "teste de ocorrência" (aquele que, como mencionamos anteriormenet, não é feito por padrão no Prolog por razões de eficiência, e que faz com que substituições do tipo `` `((,y . (,x))(,x . (a . ,y)) `` sejam inválidas).

Com isso em mãos, podemos definir nosso unificador:

```
(define (unify u v s)
  (let ((u (walk u s))(v (walk v s)))
    (cond
      ((eqv? u v) s)
      ((var? u) (ext-s u v s))
      ((var? v) (ext-s v u s))
```

*Lembre-se que `(and a b)` ⇒ b, se $a \neq \#f$.

```
((and (pair? u) (pair? v))
 (let ((s (unify (car u)(car v) s)))
   (and s
     (unify (cdr u)(cdr v) s))))
(else #f))))
```

1.3 Streams

Antes de continuarmos, precisamos tocar no modelo de avaliação de Scheme. Scheme é uma linguagem de "ordem aplicativa", o que significa que os argumentos de uma função são todos avaliados no momento em que a função é aplicada. Por esse motivo, os *and* e *or* usuais, por exemplo, não podem ser funções em Scheme. Uma alternativa à ordem aplicativa é a "ordem normal", que algumas outras linguagens de programação funcional adotaram. Linguagens de ordem normal só avaliam o argumento de uma função quando esse argumento for usada, atrasando a avaliação do mesmo (no que é chamado "avaliação tardia", ou "avaliação preguiçosa").

Avaliação preguiçosa é conveniente para diversas operações e pode ser emulada em linguagens de programação funcional de ordem aplicativa pelo uso de **streams**. *Streams* são definidos como sendo ou a lista vazia, ou um par cujo **streams**
cdr é um *stream*, ou uma suspenção.

Uma **suspenção** é uma função formada por `(λ () corpo)`, em que `((λ ()` **suspenção**
`corpo))` é uma *stream*. Agora, se definirmos

```
(define (≡ u v)
  (λ (s)
    (let ((s (unify u v s)))
      (if s `(,s) '())))))
```

≡ produz um goal. Dois outros goals, *sucesso* e *falha*, são denotamos #s e #u:

```
(define #s
  (λ (s)
    `(,s)))

(define #u
  (λ (s)
    '()))
```

Um goal é uma função que recebe uma substituição e que, se retorna, retorna uma *stream* de substituições.

Como um exemplo, temos que `((≡x y) empty-s)` ⇒ `‘(((,x . ,y)))`, uma lista com uma substituição (com uma associação).

Ao lidar com *Streams*, precisamos de funções especiais, já que não são simples listas. *Streams* são uteis para a representação de estruturas de dados infinitas, então, por isso, funções e variáveis para lidar com elas terão um ∞ sobrescrito, para diferenciá-las das funções para listas comuns. Assim, podemos definir $append^\infty$:

```
(define (append∞ s∞ t∞)
  (cond
    ((null? s∞) t∞)
    ((pair? s∞)
      (cons (car s∞
        (append∞ (cdr s∞) t∞))))
    (else (λ ()
            (append∞ t∞ (s∞))))))
```

Note que, na suspensção, a ordem dos argumentos é trocada. Com essa função, podemos fazer

```
(define (disj2 g1 g2)
  (λ (s)
    (append∞ (g1 s) (g2 s))))
```

em que $disj_2$ é uma disjunção (um *ou* lógico).

Veja agora a seguinte definição:

```
(define (nevero)
  (λ (s)
    (λ ()
      ((nevero) s))))
```

Esse é um goal que não sucede nem falha. Para entender porque a ordem dos argumentos é trocada na suspenção de $append^\infty$, compare o valor de s^∞ em

```
(let ((s∞ ((disj2
                         (nevero)
                         (≡ 'olive x))
                         empty-s)))
  s∞)
```

com o valor de s^∞ em

```
(let ((s∞ ((disj2
                         (≡ 'olive x))
```

```
                          (never^o)
                          empty-s)))
      s^∞)
```

Em contraste com *never*^{o}, aqui está *always*^{o}:

```
(define (always^o)
  (λ (s)
    (λ ()
      ((disj_2 #s (always^o)) s))))
```

Antes de continuar, será útil conhecer a função *map*:
`(map f '(`el_1 ... el_n`))` $\Rightarrow$ `'((f `e_1`) ...  (f `el_n`))`

A lista construída por *map* é constrúida por *cons*. Mas existe também *map-append*, análoga a *map*, mas em que a lista resultante é construída por *append*. Usaremos um *append-map*, mas para *streams*, isto é, um $append - map^\infty$:

```
(define (append-map^∞ g s^∞)
  (cond
    ((null? s^∞) '())
    ((pair? s^∞)
      (append^∞ (g (car s^∞))
        (append-map^∞ g (cdr s^∞))))
  (else (λ ()
          (append-map^∞ g (s^∞))))))
```

Com isso, podemos definir a conjunção de dois goals, assim como definimos a disjunção:

```
(define (conj_2 g_1 g_2)
  (λ (s)
    (append-map^∞ g_2 (g_1 s))))
```

1.3.1 Voltando ao problema das variáveis

"Reificação" é um termo muito usado em contextos de programação relacional, assim como nos da teoria de Marx, e com significado semelhante. Notadamente, é "o ato de tratar algo abstrato como algo concreto, ou algo concreto como algo abstrato". No nosso contexto, pode-se argumentar que uma variável é algo "concreto", na medida que tem algum significado para quem programa. Apesar disso, temos a liberdade de, quando conveniente, tratá-la como algo abstrato, um *place-holder*, e, quando não apresentar valor, podemos denotá-la simplesmente como um "_" seguido de um número, que serve para diferi-lo de outras variáveis*.

Isso é útil, por exemplo, para mostrarmos os valores das variáveis (e, quando a variável for fresca, podemos simplesmente mostrá-la como "$_n$"). Mais em geral, quando uma variável é criada fresca, não tem valor. Na verdade, "não ter valor" significa que a variável é fresca, e a forma de representarmos isso convenientemente é reificando-a. Para realizar isso, precisamos primeiro do *reify-name**:

```
(define (reify-name n)
  (string→symbol
    (string-append ''_''
      (number→string n))))
```

Com *reify-name*, podemos criar o *reify-s*, que recebe uma variável e uma substituição, inicialmente vazia, r:

```
(define (reify-s v r)
  (let ((v (walk v r)))
    (cond
      ((var? v)
       (let ((n (length r)))
         (let ((rn (reify-name n)))
           (cons `(,v . ,rn) r))))
      ((pair? v)
       (let ((r (reify-s (car v) r)))
         (reify-s (cdr v) r)))
      (else r))))
```

Aqui usamos *length* para produzir um número único em cada uso de *reify-name*.

Para continuar com nosso esquema de reificação, vamos precisar de uma versão ligeiramente diferente de *walk*, o qual chamaremos *walk**†. Veja a definição:

```
(define (walk* v s)
  (let (v (walk v s))
    (cond
      ((var? v) v)
      ((pair? v)
        (cons
          (walk* (car v) s)
          (walk* (cdr v) s)))
```

*Nos programas, usaremos $_n$, em que n é um número natural.

**string → symbol* é escrito *string− > symbol*.

†Leia *walk star*.

```
        (else v))))
```

Note que *walk* e *walk** só diferem se *walk** *caminhar* a um par que contém alguma variável com associação na substituição *s*. Além disso, note que, se *walk** produz um valor *v* ao *caminhar* por uma substituição *s*, temos garantia de que as variáveis em *v* são frescas.

Com isso em mãos, podemos substituir cada variável fresca por sua reificação, com

```
(define (reify s)
  (λ (s)
    (let ((v (walk* v s)))
      (let ((r (reify-s v empty-s)))
        (walk* v r)))))
```

1.4 Finalizando

Na seção anterior, definimos a coluna dorsal do miniKanren. Para fechar precisaremos usar as macros do *Scheme*. A palavra "macro", no geral, é usada (neste contexto de programação) para se referir a código que "escreve código", isto é, que realiza transformações no código a ser "enviado"ao compilador ou interpretador. Várias linguagens têm macros de tipos diferentes, mas poucas são tão poderosas como as macros que costumam estar presentes em linguagens Lisp. *Scheme* não tem na própria linguagem mecanismos de iteração, por exemplo (como um laço *for*, ou *while*), mas esses (assim como vários outros mecanismos de iteração) podem ser facilmente definidos por meio de macros.

Para começar, notamos que temos $disj_2$ e $conj_2$, que são a disjunção e conjunção, mas para apenas dois argumentos. Gostaríamos de realizar disjunções e conjunções com vários goals. A disjunção de n termos é definida induvivamente (a conjunção é análoga):

$$(disj) \Rightarrow \#u$$
$$(disj\ g) \Rightarrow g$$
$$(disj\ g_0\ g\ ...) \Rightarrow (disj_2\ g_0\ (disj\ g\ ...))$$

que se traduz como

```
(define-syntax disj
  (syntax-rules ()
    ((disj) #u)
    ((disj g) g)
    ((disj g0 g ...) (disj2 g0 (disj g ...)))))
```

Cada *defrel* define uma nova função:

```
(define-syntax defrel
  (syntax-rules ()
    ((defrel (name x  ...) g ...)
     (define (name x ...)
       (λ (s)
         (λ ()
           ((conj g ...) s)))))))
```

Váriaveis frescas são criadas com:

```
(define-syntax fresh
  (syntax-rules ()
    ((fresh () g ...)(conj g ...))
    ((fresh (x0 x ...) g ...)
     (call/fresh 'x0
       (λ (x0)
         (fresh (x ...) g ...))))))
```

Para executar o goal, definimos run^*, que recebe uma lista de variáveis e um goal e, se terminar de executar, assume como valor uma lista com os valores de associção a tais variáveis de modo que o goal tenha sucesso (vale lembrar, tal valor pode ser uma variável reificada). Definimos também run, que recebe um número natural n, uma lista de variáveis e um goal, e se terminar de executar, assume como valor os n primeiros elementos de run^*.

Para termos essas definições, precisamos de $take^\infty$, que, quando dado um número n e uma stream s^∞, se algo, produz uma lista de, no máximo, n valores:

```
(define (take∞ n s∞)
  (cond
    ((and n (zero? n)) '())
    ((null? s∞) '())
    ((pair? s∞)
     (cons (car s∞)
       (take∞ (and n (sub1 n))
         (cdr s∞))))
    (else (take∞ n (s∞)))))
```

note que, se $n = false$, $take^\infty$, se retornar, produz uma lista de *todos* os valores (pergunta: valores de que?).

Agora podemos definir

```
(define (run-goal n g)
  (take∞ n (g empty-s)))
```

e

```
(define-syntax run
  (syntax-rules ()
    ((run n (x0 x ...) g ...)
     (run n q (fresh (x0 x ...)
                (≡ `(,x0 ,x ...) q) g ...)))
    ((run n q g ...)
     (let ((q (var 'q)))
       (map (reify q)
         (run-goal n (conj g ...)))))))

 (define-syntax run*
   (syntax-rules ()
     ((run* q g ...)(run #f q g ...))))
```

Com isso, temos um implementação mínima de *miniKanren*. Algumas construções importantes foram deixadas de lado em favor da brevidade, como os $cond^e$, $cond^a$ e $cond^u$, mas a leitora interessada é convidada a checar [3] ou [4] para mais detalhes.

Referências

[1] R. Kent Dybvyg, “The Scheme Programming Language - Fourth Edition”, disponível em https://www.scheme.com/tspl4/, acesso em Setembro de 2018.

[2] Peter Norvig, “Paradigms of Artificial Inteligence Programming - Case Studies in Common Lisp”, Morgan Kauffman Publishers, 1992.

[3] Daniel P. Friedman, William E. Byrd, Oleg Kiselyov, Jason Hemann, “The Reasoned Schemer - Second Edition”, The MIT Press, 2018.

[4] Site do miniKanren http://minikanren.org/.