Introdução a Linguagem R

Escopo

Delermando Branquinho Filho

A desvio sobre valores de ligação para símbolo

Como R sabe qual valor atribuir a qual símbolo? Quando eu digito

```
m <- function(x = 0) {
        x * x
}

m(2)
```

```
## [1] 4
```

Como R sabe qual valor atribuir ao símbolo `lm` ? Por que ele não dá o valor de `lm` que está no pacote * stats *?

O desvio sobre valores de ligação para símbolo

Quando R tenta vincular um valor a um símbolo, ele procura através de uma série de 'ambientes' para encontrar o valor apropriado. Quando você está trabalhando na linha de comando e precisa recuperar o valor de um objeto R, a ordem é

1. Pesquise no ambiente global um nome de símbolo correspondente ao solicitado.
2. Pesquise os espaços para nome de cada um dos pacotes na lista de pesquisa

A lista de pesquisa pode ser encontrada usando a função `search` .

```
search()
```

```
## [1] ".GlobalEnv"        "package:stats"     "package:graphics"
## [4] "package:grDevices" "package:utils"     "package:datasets"
## [7] "package:methods"   "Autoloads"         "package:base"
```

Vincular valores ao símbolo

- O *global environment* ou o espaço de trabalho do usuário é sempre o primeiro elemento da lista de pesquisa eo pacote * base * é sempre o último.
- A ordem dos pacotes na lista de pesquisa é importante!
- O usuário pode configurar quais pacotes são carregados na inicialização para que você não possa assumir que haverá uma lista de conjuntos de pacotes disponíveis.
- Quando um usuário carrega um pacote com `library` o namespace desse pacote é posto na posição 2 da lista de pesquisa (por padrão) e tudo o resto é deslocado para baixo na lista.

- Observe que R tem namespaces separados para funções e não-funções para que seja possível ter um objeto chamado c e uma função chamada c.

Regras de definição do escopo

As regras de escopo para R são a principal característica que a tornam diferente da linguagem S original.

- As regras de escopo determinam como um valor está associado a uma variável livre em uma função
- R usa 'escopo' ou 'escopo estático'. Uma alternativa comum é *dynamic scoping*.
- Relacionado às regras de escopo é como R usa o *list* de pesquisa para vincular um valor a um símbolo
- O escopo lexical revela-se particularmente útil para simplificar cálculos estatísticos

---

Lexical Scoping

Considere a seguinte função.

```
f <- function(x, y) {
        x ^ 2 + y / z
}
z <- 2
f(2,3)
```

```
## [1] 5.5
```

Esta função tem 2 argumentos formais `x` e `y`. No corpo da função há outro símbolo "z". Nesse caso, `z` é chamado de variável *free*. As regras de escopo de uma linguagem determinam como os valores são atribuídos a `variáveis livres`. As `variáveis livres` não são argumentos formais e não são variáveis `locais` (atribuídas no corpo da função).

Lexical Scoping

O escopo lexical em R significa que

Os valores das `variáveis livres` são pesquisados no ambiente em que a função foi definida.

O que é um ambiente?

- Um *environment* é uma coleção de pares (de símbolo, valor), ou seja, `x` é um símbolo e `3.14` pode ser seu valor.
- Todo ambiente tem um ambiente pai; É possível que um ambiente tenha múltiplos "filhos"
- o único ambiente sem um pai é o ambiente vazio
- Uma função + um ambiente = a_closure_ ou *function closure*.

Lexical Scoping

Procurando o valor para uma variável livre:

- Se o valor de um símbolo não for encontrado no ambiente em que uma função foi definida, a pesquisa será continuada no ambiente *parent*.

- A busca continua abaixo da sequência de ambientes pai até atingir o *top-level environment*; Geralmente, o ambiente global (espaço de trabalho) ou o namespace de um pacote.
- Após o ambiente de nível superior, a pesquisa continua abaixo da lista de pesquisa até atingir o ambiente *empty*. Se um valor para um dado símbolo não pode ser encontrado uma vez que o ambiente vazio é chegado, em seguida, um erro é lançado.

Lexical Scoping

Por que isso tudo importa?

- Normalmente, uma função é definida no ambiente global, de modo que os valores das `variáveis livres` são apenas encontrados no espaço de trabalho do usuário
- Este comportamento é lógico para a maioria das pessoas e é geralmente a “coisa certa” para fazer
- No entanto, em R você pode ter funções definidas dentro de outras `funções`
- Linguagens como C não permitem que você faça isso
- Agora as coisas ficam interessantes - Neste caso, o ambiente em que uma função é definida é o corpo de outra função!

Lexical Scoping

```
make.power <- function(n) {
        pow <- function(x) {
        x^n
        }
        pow
}
```

Esta função retorna outra função como seu valor.

```
cube <- make.power(3)
square <- make.power(2)
cube(3)
```

```
## [1] 27
```

```
square(3)
```

```
## [1] 9
```

Explorando um encerramento de função

O que há no ambiente de uma função?

```
ls(environment(cube))
```

```
## [1] "n"   "pow"
```

```
get("n", environment(cube))
```

```
## [1] 3
```

```
ls(environment(square))
```

```
## [1] "n"   "pow"
```

```
get("n", environment(square))
```

```
## [1] 2
```

Lexical vs. Escopo Dinâmico

```
y <- 10

f <- function(x) {
        y <- 2
        y^2 + g(x)
}

g <- function(x) {
        x*y
}
```

Qual é o valor de

```
f(3)
```

```
## [1] 34
```

Lexical vs. Escopo Dinâmico

- Com o escopo lexical o valor de `y` na função `g` é procurado no ambiente em que a função foi definida, neste caso o ambiente global, então o valor de `y` é 10.
- Com o escopo dinâmico, o valor de `y` é procurado no ambiente a partir do qual a função foi *called* (às vezes referida como o ambiente *calling*).
- Em R o ambiente de chamada é conhecido como *parent frame*
- Então o valor de `y` seria 2.

Quando uma função é *definida* no ambiente global e é subsequentemente *called* do ambiente global, então o ambiente de definição e o ambiente de chamada são iguais. Isso às vezes pode dar a aparência de escopo dinâmico.

```
g <- function(x) {
        a <- 3
        x+a+y
        }
g(2)
# Error in g(2) : object "y" not found
y <- 3
g(2)
```

Outras línguas

Outros idiomas que suportam o escopo lexical - Scheme - Perl - Python - Lisp comum (todas as línguas convergem para Lisp)

Consequências do Lexical Scoping

- Em R, todos os objetos devem ser armazenados na memória
- Todas as funções devem levar um ponteiro para seus respectivos ambientes de definição, que poderiam estar em qualquer lugar
- No S-PLUS, as variáveis livres são sempre procuradas no espaço de trabalho global, para que tudo possa ser armazenado no disco porque o “ambiente de definição” de todas as funções é o mesmo.

Aplicação: Otimização

Por que alguma dessas informações é útil? - As rotinas de otimização em R como `optim` , `nlm` e `optimize` exigem que você passe uma função cujo argumento é um vetor de parâmetros (por exemplo, uma log-verossimilhança) - No entanto, uma função de objeto pode depender de uma série de outras coisas além de seus parâmetros (como *data*) - Ao escrever software que faz otimização, pode ser desejável permitir que o usuário mantenha determinados parâmetros fixos

Maximizando uma Probabilidade Normal

Escrever uma função “construtor”

```
make.NegLogLik <- function(data, fixed=c(FALSE,FALSE)) {
        params <- fixed
        function(p) {
                params[!fixed] <- p
                mu <- params[1]
                sigma <- params[2]
                a <- -0.5*length(data)*log(2*pi*sigma^2)
                b <- -0.5*sum((data-mu)^2) / (sigma^2)
                -(a + b)
        }
}
```

Nota: Funções de otimização em funções R *minimize*, você precisa usar a probabilidade log negativa.

```
set.seed(1); normals <- rnorm(100, 1, 2)
nLL <- make.NegLogLik(normals)
nLL
```

```
## function(p) {
##                 params[!fixed] <- p
##                 mu <- params[1]
##                 sigma <- params[2]
##                 a <- -0.5*length(data)*log(2*pi*sigma^2)
##                 b <- -0.5*sum((data-mu)^2) / (sigma^2)
##                 -(a + b)
##         }
## <environment: 0x000000000a3e10e8>
```

```
function(p) {
                params[!fixed] <- p
                mu <- params[1]
                sigma <- params[2]
                a <- -0.5*length(data)*log(2*pi*sigma^2)
                b <- -0.5*sum((data-mu)^2) / (sigma^2)
                -(a + b)
        }
```

```
## function(p) {
##                 params[!fixed] <- p
##                 mu <- params[1]
##                 sigma <- params[2]
##                 a <- -0.5*length(data)*log(2*pi*sigma^2)
##                 b <- -0.5*sum((data-mu)^2) / (sigma^2)
##                 -(a + b)
##         }
```

```
ls(environment(nLL))
```

```
## [1] "data"   "fixed"  "params"
```

Estimando Parâmetros

```
optim(c(mu = 0, sigma = 1), nLL)$par
```

```
##       mu    sigma
## 1.218239 1.787343
```

Fixing σ = 2

```
nLL <- make.NegLogLik(normals, c(FALSE, 2))
optimize(nLL, c(-1, 3))$minimum
```

```
## [1] 1.217775
```

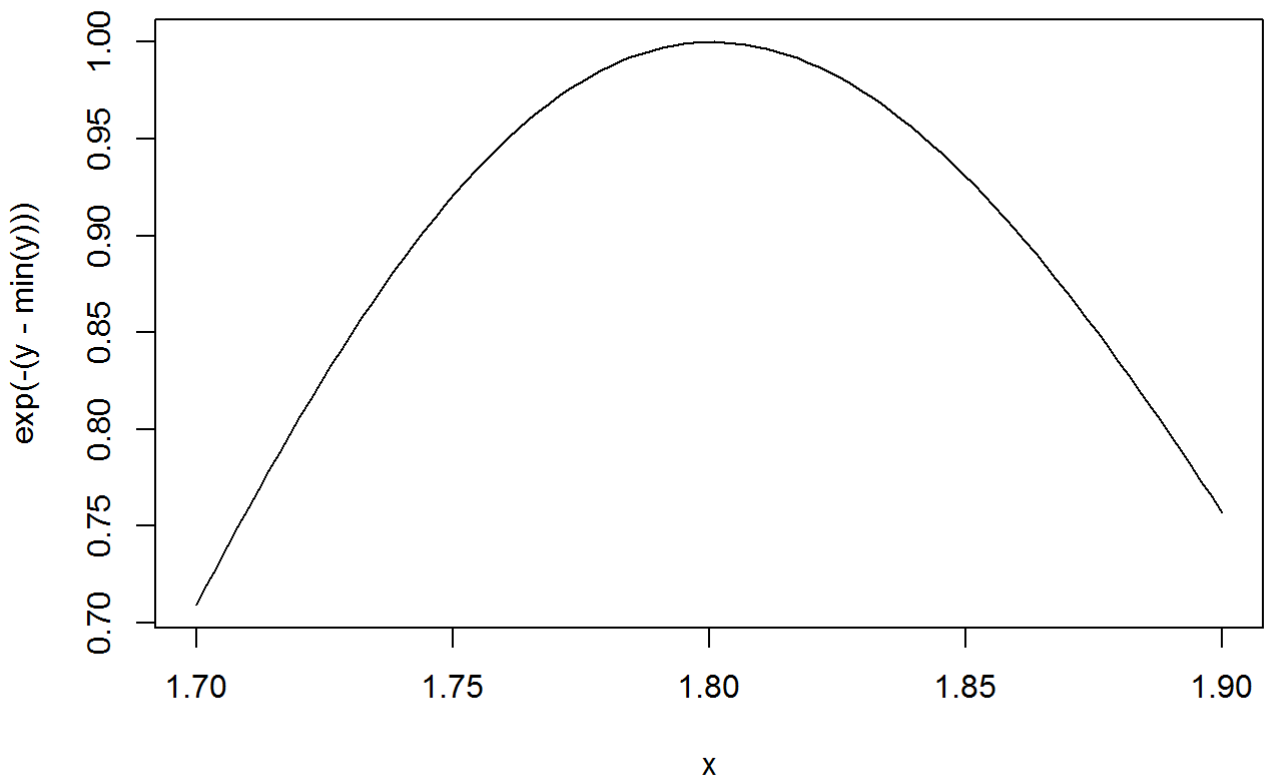
Fixing μ = 1

```
nLL <- make.NegLogLik(normals, c(1, FALSE))
optimize(nLL, c(1e-6, 10))$minimum
```

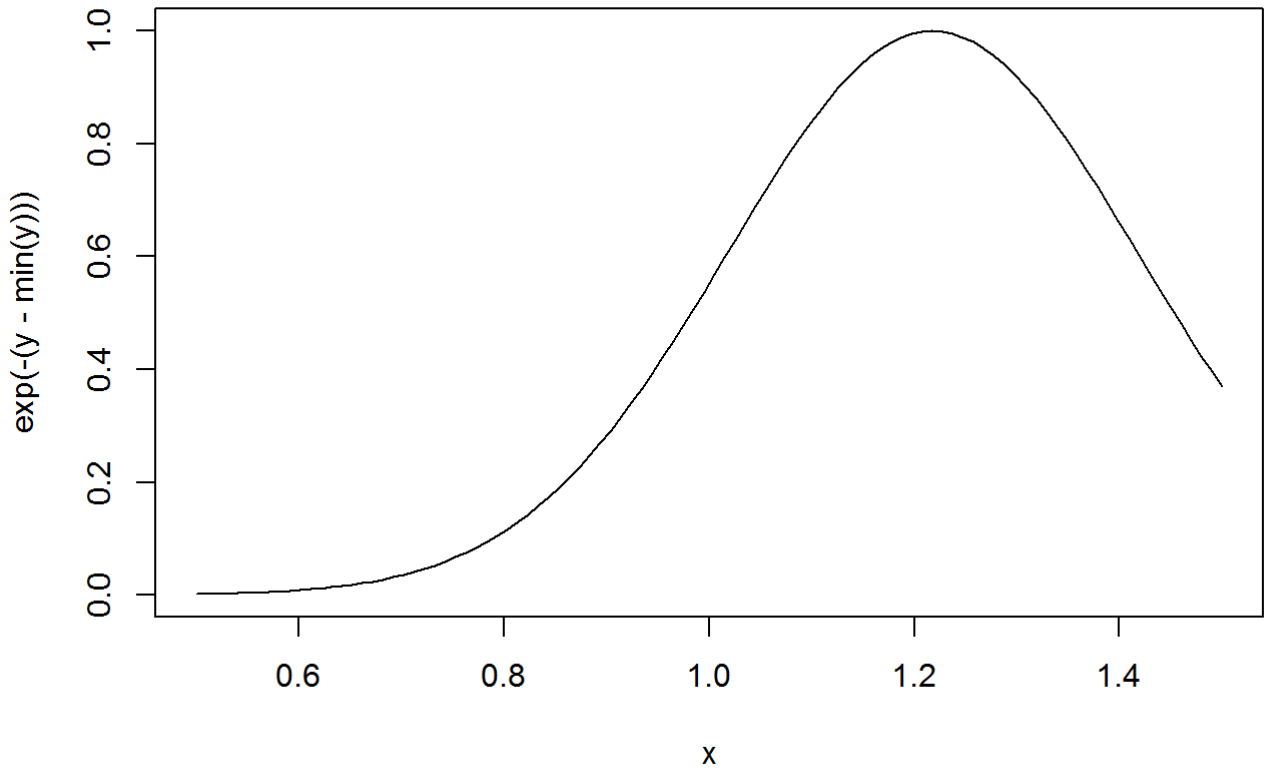
```
## [1] 1.800596
```

Plotando a Verossimilhança

```
nLL <- make.NegLogLik(normals, c(1, FALSE))
x <- seq(1.7, 1.9, len = 100)
y <- sapply(x, nLL)
plot(x, exp(-(y - min(y))), type = "l")
```




```
nLL <- make.NegLogLik(normals, c(FALSE, 2))
x <- seq(0.5, 1.5, len = 100)
y <- sapply(x, nLL)
plot(x, exp(-(y - min(y))), type = "l")
```



Lexical Scoping Resumo

- As funções objetivas podem ser “construídas” que contêm todos os dados necessários para avaliar a função
- Não há necessidade de transportar longas listas de argumentos - útil para o trabalho interativo e exploratório.
- Código pode ser simplificado e cleand up
- Referência: Robert Gentleman e Ross Ihaka (2000). “Âmbito lexical e computação estatística”, *JCGS*, 9, 491-508.

.