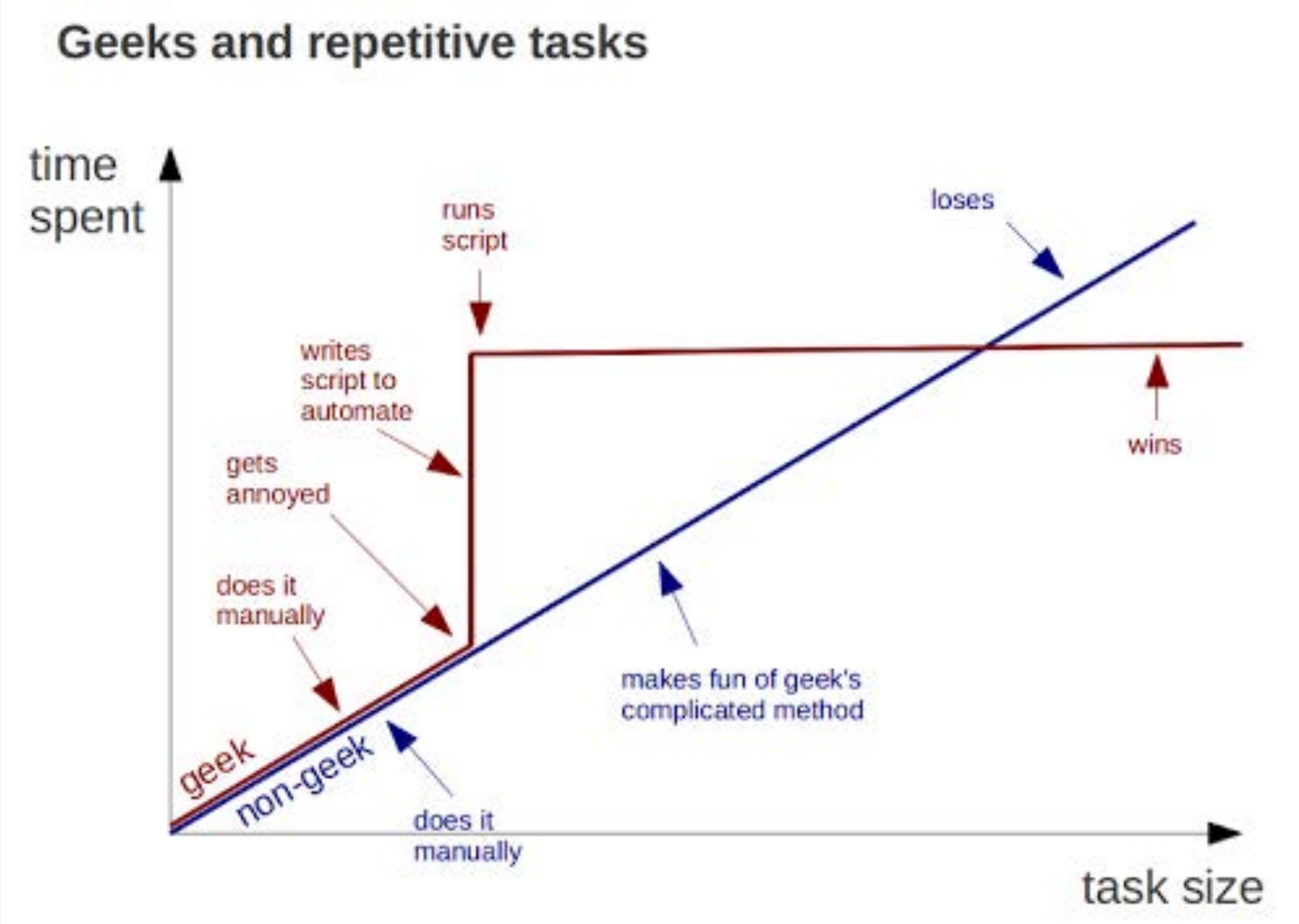
Practical Programming with R

Stuart Borrett
biol534



Why to write code

Bruno Olveira
http://nicercode.github.io/blog/2013-04-05-why-nice-code/

Review Lab 1

What were the most challenging Exercises? Why?

Remaining Questions?

Compare your results to the posted solutions

http://people.uncw.edu/borretts/courses/bio534/labs/solutions/Bio534-lab1-solutions.pdf

Practical Programming: Learning objectives

Students should be able to…

- Organize computational projects
- Identify and apply programming concepts such as loops and branching
- Recognize the computational savings of vectorizing tasks when possible
- Practice debugging
- Create functions in R.

1. Organizing Computational Projects
2. Program Flow Control
3. Neat Programming
4. Practice

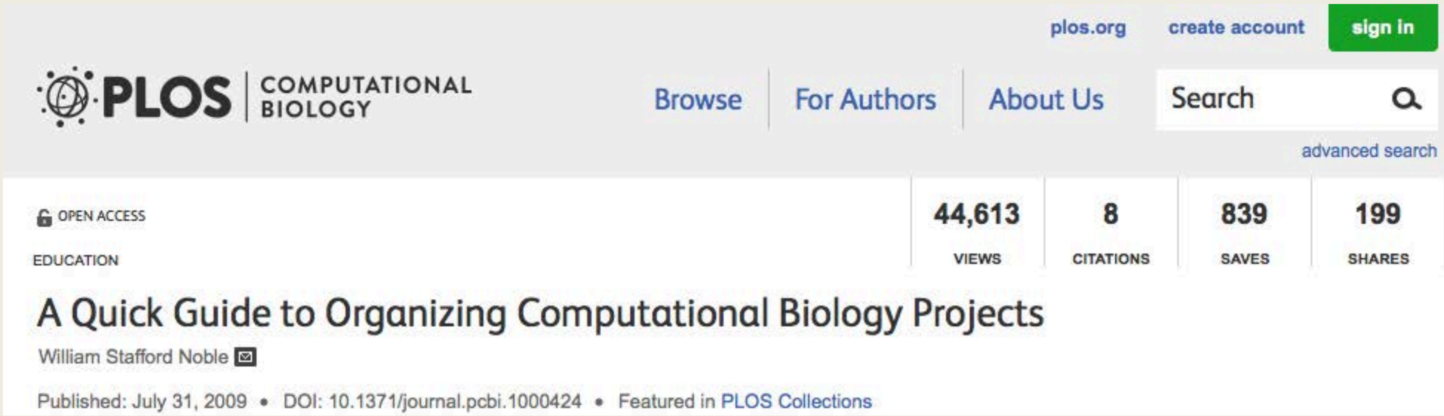
1 2 3 4

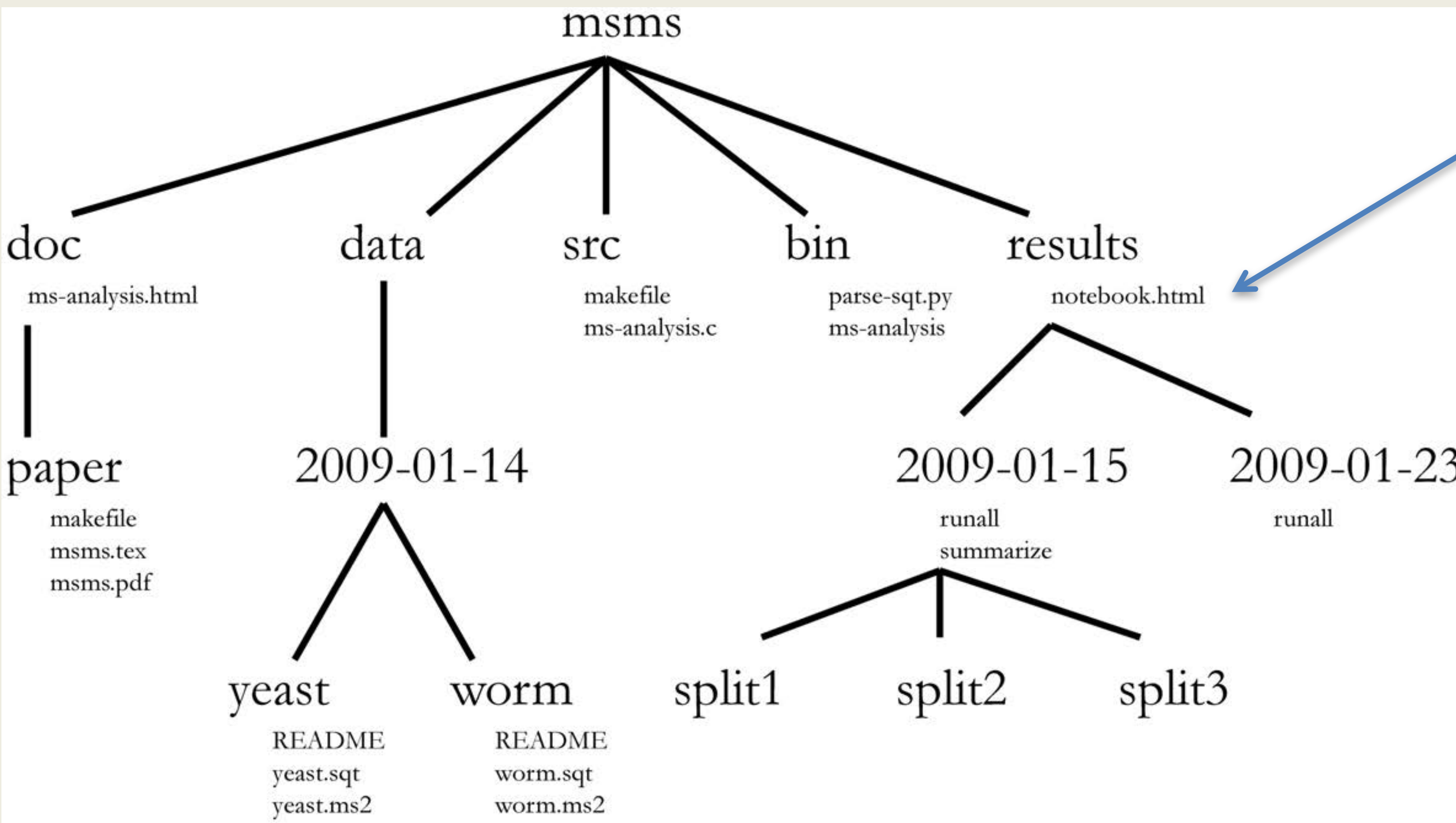
Organization of Computational Projects

Problem: Many files, bits, and pieces

Question: How do we keep it organized?

Nobel's Project Directory





Research Notebook

Simplified Project Organization

Project

- **bin** — Source code, programs, scripts
- **data** — Any and all data for the project
- **doc** — Documents/presentations you write for the project
- **refs** — Relevant papers
- **results** — Analytical results, figures, tables, and Project Notebook

Example: Throughflow Centrality

Directory Structure

```
tcent — borretts@152-20-221-131:~/research/tcent — b...

 borretts@152-20-221-131: tcent $ pwd
/Users/borretts/research/tcent
 borretts@152-20-221-131: tcent $ ll
total 12288
drwxr-xr-x   57 borretts  staff   1.9K Aug 26  2013 bin
drwxr-xr-x    5 borretts  staff   170B Nov 21  2012 data
drwx------   59 borretts  staff   2.0K May 20  2013 doc
-rwx------    1 borretts  staff   6.0M Aug  4  2011 eec.zip
drwxr-xr-x@   5 borretts  staff   170B Dec 23  2011 references
drwxr-xr-x  136 borretts  staff   4.5K Aug 17 10:44 results
 borretts@152-20-221-131: tcent $
```

Project Notebook

Throughflow Centrality

Laboratory Notebook

Stuart R. Borrett

Show files

1 2 3 4

Program Flow Control

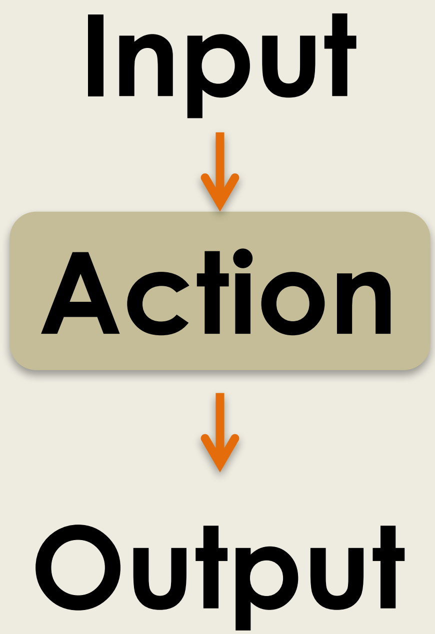
Loops and Branching

Basic Flow

By default, R reads scripts and executes them line by line.

- Replicates entering commands by hand at the command line

Ultrastructure



Basic Flow

By default, R reads scripts and executes them line by line.

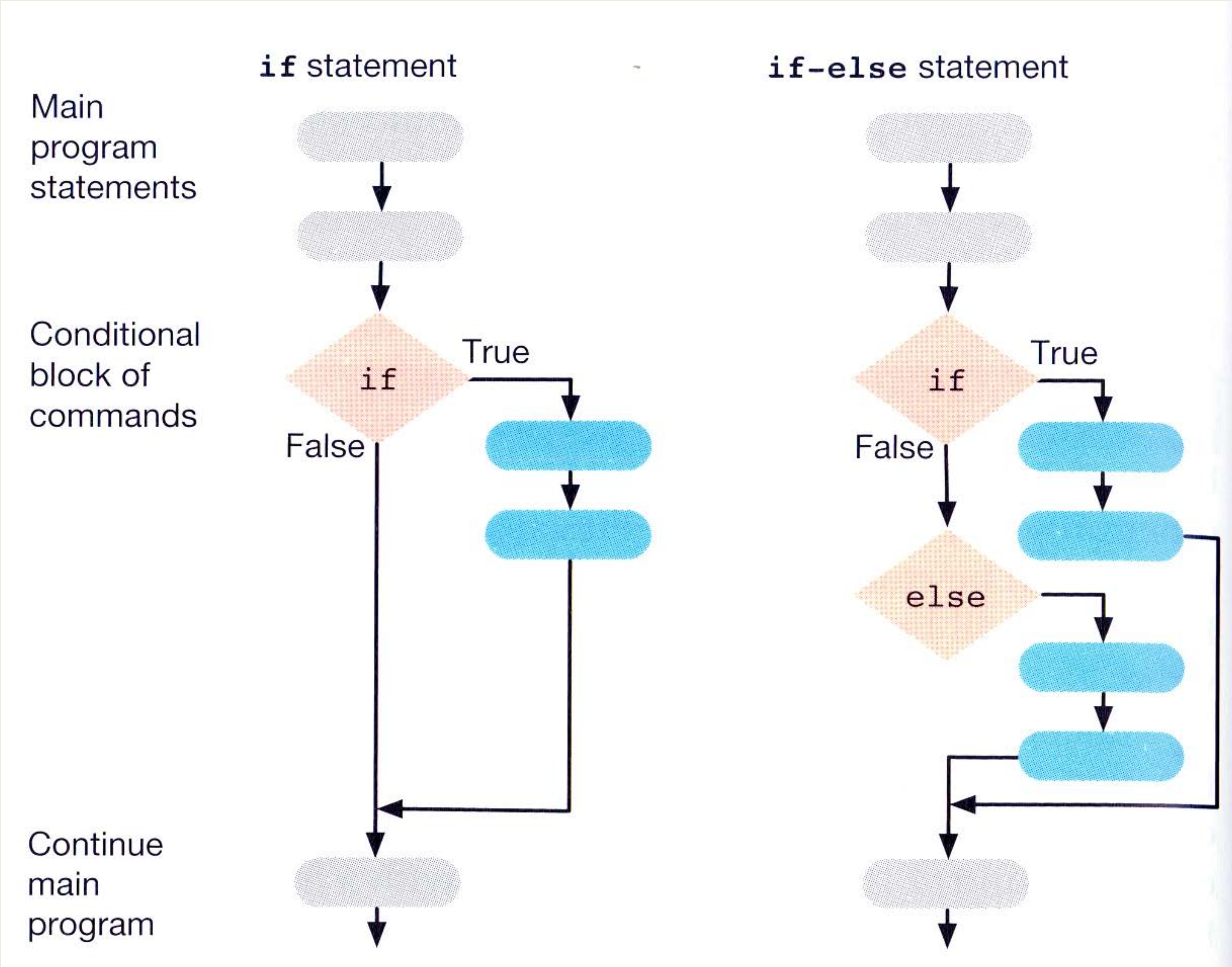
- Replicates entering commands by hand at the command line

Create and Execute the Following Script

```
# Example Script
# Borrett, Aug. 2011
# ------------------
setwd("~/teaching/biol534.f11/PracticalProgramming/code")  # change working
directory

# INPUT - create variables
a = runif(100) # creates a vector of 100 numbers drawn from a uniform random
distribution between 0 and 1

b = rnorm(100) # creates a vector of 100 numbers drawn form a normal distribution
with mean 0 and standard deviation 1.

# ACTION
c = a + b

# OUTPUT
hist(a)

quartz()  # creates new plot window on MAC; use win() on windows or x11() on
linux or mac
plot(a,b)
```

Branches – If-Then Statements

Sometimes we only want code to execute when certain conditions are met



Haddock and Dunn 2011, Fig 7.3

Conditional flow Control

Branching R Example

General Form

```
if(condition) {
    some commands
}else{
    some other commands
}
```

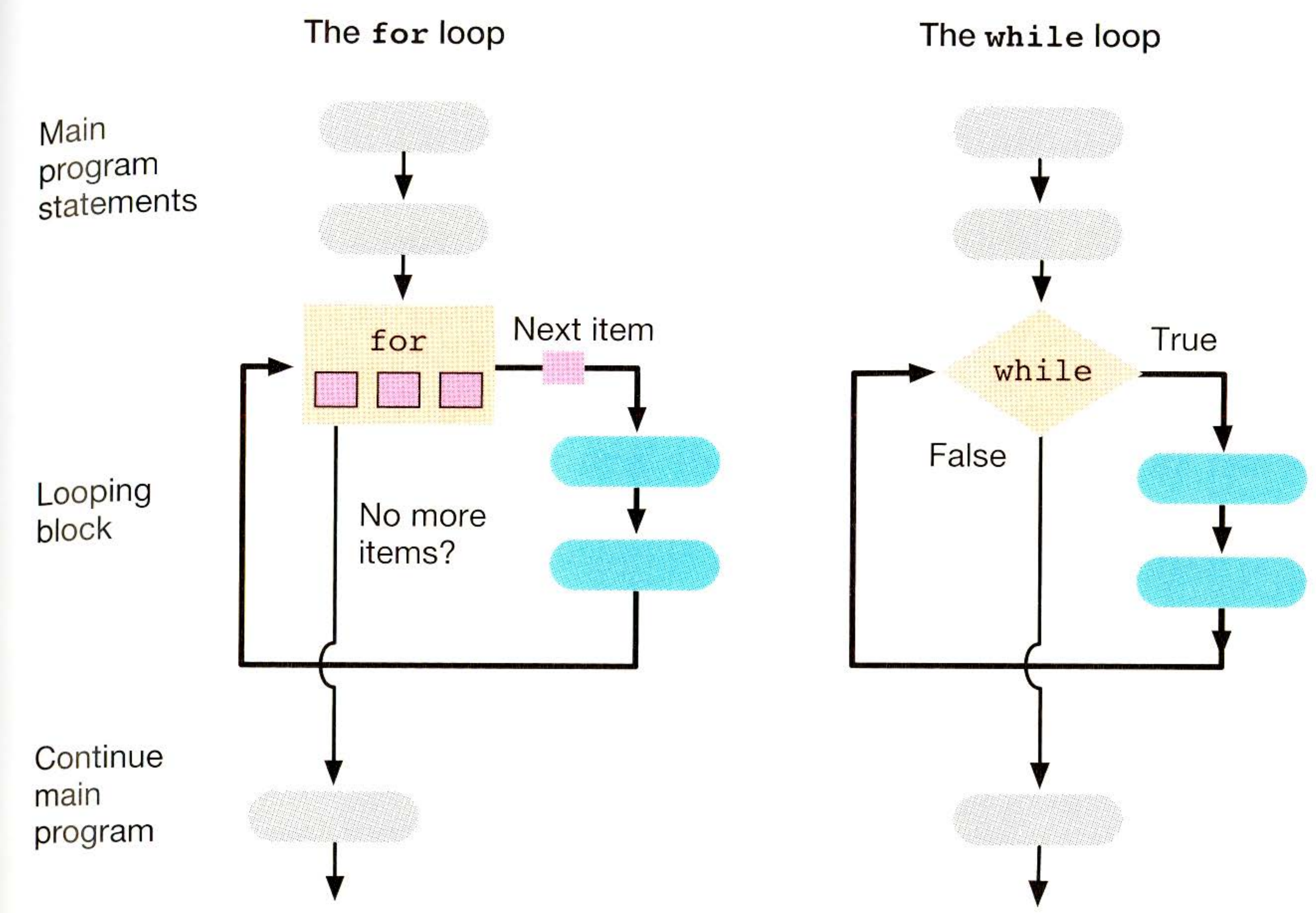
Jones, Maillardet, Robinson 2009

Example

```
# program spuRs/resources/scripts/quad2.r
# find the zeros of a2*x^2 + a1*x + a0 = 0

# clear the workspace
rm(list=ls())

# input
a2 <- 1
a1 <- 4
a0 <- 5

# calculate the discriminant
discrim <- a1^2 - 4*a2*a0
# calculate the roots depending on the value of the discriminant
if (discrim > 0) {
    roots <- c( (-a1 + sqrt(a1^2 - 4*a2*a0))/(2*a2),
                (-a1 - sqrt(a1^2 - 4*a2*a0))/(2*a2) )
} else {
    if (discrim == 0) {
        roots <- -a1/(2*a2)
    } else {
        roots <- c()
    }
}

# output
show(roots)
```

Iteration by Loops

Sometimes we want to perform the same action multiple times



Haddock and Dunn 2011, Fig 7.4

Example: Summing a Vector

General Form

```
for (var in seq) {
       commands
       }
```

Example

```
# Example: Summing a Vector
# Borrett, Aug 2011
# From Jones, Maillardet, and Robinson 2009, p33
# ---------------

x.list = seq(1,9, by=2)

sum.x = 0  # initialize sum.x

for (x in x.list){
    sum.x = sum.x + x  # incremental sum
    cat("The current loop element is ",x, "\n")
    cat("The cummulative total is ", sum.x, "\n")
}
```

Example: Pension

```
# program: spuRs/resources/scripts/pension.r
# Forecast pension growth under compound interest

# clear the workspace
rm(list=ls())

# Inputs
r <- 0.11                 # Annual interest rate
term <- 10                # Forecast duration (in years)
period <- 1/12            # Time between payments (in years)
payments <- 100           # Amount deposited each period

# Calculations
n <- floor(term/period)   # Number of payments
pension <- 0
for (i in 1:n) {
    pension[i+1] <- pension[i]*(1 + r*period) + payments
}
time <- (0:n)*period

# Output
plot(time, pension)
```

Jones, Maillardet, Robinson 2009

Example: Exponential Pop Growth

```
# Iteration Example: Exponential Population Growth
# Borrett, Aug 2011
# Haefner equation 2.5
# -------------------

# INPUTS
mx.time = 10  # number of time units to consider
N = rep(0,mx.time)  # initialize population vector
N0 = 10  # initial population size
r = 0.5  # per capita rate of population growth

# ACTION

for (i in 1:mx.time){  # note start at time 2
    cat("index is", i, "\n")
    if(i == 1){
        N[i] = N0
        cat("initial condition set")
        }
    N[i+1] = N[i] + r*N[i]  # main equation
}

# OUTPUT
time.vec = seq(0,mx.time,by=1)
plot(time.vec,N,
    type = "b",
    xlab = "time",
    ylab = "population size (individuals)",
    )
```

Charting Flow

Program (3plus1)

```
   # program: spuRs/resour
1  x <- 3
2  for (i in 1:3) {
3    show(x)
4    if (x %% 2 == 0) {
5      x <- x/2
6    } else {
7      x <- 3*x + 1
8    }
9  }
10 show(x)
```

Chart

Table 3.1 *Charting the flow for program* `threexplus1.r`

line	x	i	comments
1	3		i not defined yet
2	3	1	i is set to 1
3	3	1	3 written to screen
4	3	1	`(x %% 2 == 0)` is `FALSE` so go to line 7
7	10	1	x is set to 10
8	10	1	end of else part
9	10	1	end of for loop, not finished so back to line 2
2	10	2	i is set to 2
3	10	2	10 written to screen
4	10	2	`(x %% 2 == 0)` is `TRUE` so go to line 5
5	5	2	x is set to 5
6	5	2	end of if part, go to line 9
9	5	2	end of for loop, not finished so back to line 2
2	5	3	i is set to 3
3	5	3	5 written to screen
4	5	3	`(x %% 2 == 0)` is `FALSE` so go to line 7
7	16	3	x is set to 16
8	16	3	end of else part
9	16	3	end of for loop, finished so continue to line 10
10	16	3	16 written to screen

This is exactly what the computer does when it executes a program: it keeps track of its current position in the program and maintains a list of variables and their values. *Whatever line you are currently at, if you know all the variables then you always know which line to go to next.*

Jones, Maillardet, Robinson 2009

While Loops

When you don't know how many times you need to iterate

Example

```
# program: spuRs/resources/scripts/compound.r
# Duration of a loan under compound interest

# clear the workspace
rm(list=ls())

# Inputs
r <- 0.11                  # Annual interest rate
period <- 1/12             # Time between repayments (in years)
debt_initial <- 1000       # Amount borrowed
repayments <- 12           # Amount repaid each period

# Calculations
time <- 0
debt <- debt_initial
while (debt > 0) {
    time <- time + period
    debt <- debt*(1 + r*period) - repayments
}

# Output
cat('Loan will be repaid in', time, 'years\n')
```

Jones, Maillardet, Robinson 2009

Loops vs. Vectorizaiton

- Loops work
- Vectorized calculations are much faster.

Loop

```
ptm = proc.time()

n = 100000
s = 0
for (i in 1:n){
    s = s + i^2
}
s

proc.time() - ptm
```

```
> ptm = proc.time()
>
> n = 100000
> s = 0
> for (i in 1:n){
+   s = s + i^2
+ }
> s
[1] 3.333383e+14
>
> proc.time() - ptm
   user  system elapsed
  0.108   0.002   0.152
```

Vectorized

```
ptm = proc.time()

sum((1:n)^2)

proc.time() - ptm
```

```
> ptm = proc.time()
> sum((1:n)^2)
[1] 3.333383e+14
> proc.time() - ptm
   user  system elapsed
  0.004   0.001   0.028
>
```

Functions

- Functions are like scripts, but they can be used to break the actions into chunks
- Usually use a function for a task that will be repeated

General Form

```
function.name=function(argument1,argument2,...) {
      command;
      command;
        ...
      command;
      return(value)
}
```

Examples

```
mysquare=function(v,w) {
     u=v^2+w^2;
     return(u)
}
```

```
mysquare2=function(v,w) {
     q=v^2; r=w^2
     return(list(v.squared=q,w.squared=r))
}
```

1 2 3 4

Neat Programming

Neat and well documented code facilitates use and debugging

Good Programming Habits

Header

- Name of Program
- Name of Author
- Date
- Function Objectives
  - INPUT
  - OUTPUT

Variable Names

- Use descriptive or meaningful names when possible
- Avoid using reserved names [exists() function]

Use comments to describe analytical steps

Use blank lines to separate code into distinct parts

Use indenting for loops and branches

```
# program: spuRs/resources/scripts/compound.r
# Duration of a loan under compound interest

# clear the workspace
rm(list=ls())

# Inputs
r <- 0.11                  # Annual interest rate
period <- 1/12             # Time between repayments (in years)
debt_initial <- 1000       # Amount borrowed
repayments <- 12           # Amount repaid each period

# Calculations
time <- 0
debt <- debt_initial
while (debt > 0) {
    time <- time + period
    debt <- debt*(1 + r*period) - repayments
}

# Output
cat('Loan will be repaid in', time, 'years\n')
```

R Style Guide

https://google.github.io/styleguide/Rguide.xml

This is a set of useful code style guidelines.

Version Control

- Software that keeps track of file changes
  - Useful for software development, coding
  - Useful for paper/presentation preparation
    - Do you use a version numbering system in the file name (e.g., myfile_v1.docx)?

- Software Examples: Git, Mecurial, CVS, Subversion

- More Info @ http://git-scm.com/book/en/Getting-Started-About-Version-Control
- http://nicercode.github.io/git/

1 2 3 4

Practice

Complete Exercises {1, 2, 3, 4, 6, 7, 9a} Jones et al. 2009

Write a function "domeig"

- Takes as input a single vector and returns a list with components "average" (mean of the values of in the vector) and "variance" (the variance of the values in the vector). [DMB]