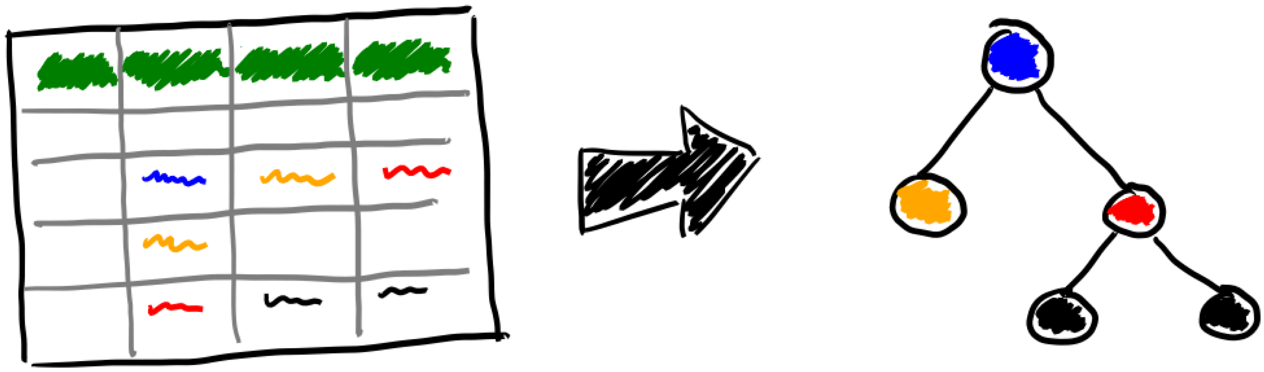

From Spreadsheets to Programs

Joe Gibbs Politz (UC San Diego)
Benjamin S. Lerner (Northeastern University)
Kathi Fisler (Brown University)
Shriram Krishnamurthi (Brown University)

With material and format remixed from…

BOOTSTRAP
www.bootstrapworld.org

Part 1 – Warmup

As a warmup, design a table (think spreadsheet layout) to represent…

An animal shelter's adoption list

A grade spreadsheet for a programming assignment with partners and multiple parts

Social networking account profiles

Genealogy (people and their biological parental relationships)

Numbers and Strings

Make sure you've loaded the **Getting to Know Pyret** starter file, and clicked "Run". We'll get to the table – first a few simple things to try:

1. Try typing `42` into the Interactions Area and hitting "Enter". What happens?

2. Try typing in other Numbers. What happens if you try a decimal like `0.5`? A fraction like `1/3`? Try really big Numbers, and really small ones.

3. String values are always in quotes. Try typing your name in quotes, and see what happens when you hit "Enter".

4. Try typing your name *without* the closing quote. What happens? Now try typing it without *any* quotes.

5. Just like in math, Pyret has *operators* like `+` and `-`. Try typing in `4 + 2`, and then `4+2` (without the spaces). What can you conclude from this?

6. Try typing in `4 + 2 + 6`, `4 + 2 * 6`, and `4 + (2 * 6)`. What can you conclude from this?

7. Try typing in `4 + "cat"`, and then `"dog" + "cat"`. What can you conclude from this? Write your answer below:

Booleans

What will each of the shaded expressions below evaluate to? Type them into Pyret if you're not sure.

`3 <= 4`	______________	`"a" > "b"`	______________
`3 == 2`	______________	`"a" <> "b"`	______________
`2 <> 4`	______________	`"a" == "b"`	______________
`3 <> 3`	______________	`"a" <> "a"`	______________

Boolean Operators

Pyret also has operators that work on *Booleans*. For each expression below, *write down your guess* about what it will evaluate to.

`(3 <= 4) and (3 == 2)`	_______________
`("a" == "b") and (3 <> 4)`	_______________
`(3 <= 4) or (3 == 2)`	_______________
`("a" == "b") or (3 <> 4)`	_______________

Tables, Functions, and Contracts

1. The table in the starter file defines several rows of shapes. Try typing `shapes` in the interactions window to see the table that was created.

2. Try adding a row for `rectangle`

3. We'll use **Contracts** to describe functions. The **Contract** for circle is:

   ```
   circle :: (radius :: Number), (style :: String), (color :: String) -> Image
   ```

   It describes what arguments the function consumes and what type of value it produces. We'll use contracts to describe all the built-in functions we use, and when we define our own functions. The last two pages of this handout have a *bunch* of contracts that you can use.

   The names on the parameters are optional, so we could also write circle as:

   ```
   circle :: Number, String, String -> Image
   ```

   What's the contract for rectangle?

4. Try adding a row for star and a row for ellipse

5. Try adding a column for corners, which holds the number of corners in each shape

6. We can get an individual row from a table with the function get-row:

   ```
   get-row :: (t :: Table), (index :: Number) -> Row
   ```

   Try out get-row on the shapes table at interactions!

7. To access the individual values within a row, we can use **bracket lookup**. Try running these expressions at interactions:

   ```
   > r = get-row(shapes, 0)
   > r["sample"]
   > r["name"]
   ```

Part 2 – Functions over Rows

Open the **Animals Dataset** starter file.

- We will manipulate tables by writing and using functions.
- Programming languages let us **define our own functions**.
- We use the **Design Recipe** to help us define functions without making mistakes.
- The first step is to write a **Contract** and **Purpose Statement** for the function, which specify the Name, Domain and Range of the function and give a summary of what it does.
- The second step is to **write at least two examples**, which show how the function should work for specific inputs. These examples help us see patterns, and we express those patterns by **circling and labeling** what changes.
- The final step is to **define the function**, which generalizes our examples and gives us an implementation.

(If you like the idea of the design recipe, you might like the book *How to Design Programs*)

The Design Recipe

Define a function called `is-fixed`, which tells us whether or not an animal is fixed

is-fixed :: *(animal :: Row)* → *Boolean*

name domain range

\# *Consumes an animal, and produces the value in the fixed column*

examples:

is-fixed (*sasha*) **is** *true*

________ (______) **is** ________

end

fun ________ (______) **:** ________

end

Define a function called `gender`, which consumes a Row of the animals table tells us the gender of that animal

________ :: ________ → ________

name domain range

\# ________

examples:

________ (______) **is** ________

________ (______) **is** ________

end

fun ________ (______) **:** ________

end

Define a function called `is-cat`, which consumes a Row of the animals table and produces `true` if it's a cat.

is-cat	::	*(animal :: Row)*	→	*Boolean*
name		domain		range

\# *Consumes an animal, and return true if the species is "cat"*

examples:

is-cat (*sasha*) **is** ______________________

______ (______) **is** ______________________

end

fun ______ (______) **:** ______________________

end

Define a function called `is-young`, which consumes a Row of the animals table and produces `true` if it's an animal that is less than two years old.

______	::	______	→	______
name		domain		range

\# ______________________

examples:

______ (______) **is** ______________________

______ (______) **is** ______________________

end

fun ______ (______) **:** ______________________

end

Define a function called `nametag`, which creates an image with the given animal's name in big red letters.

nametag :: *(animal :: Row)* → *Image*

name domain range

\# *Consumes an animal, and produces an image of their name in big, red letters*

examples:

nametag (*sasha*) **is** ______________________________

______________ (__________) **is** ______________________________

end

fun ______________ (__________) **:** ______________________________

end

Define a function called `is-kitten`, which consumes a Row of the animals table and produces true if it's an

______________ :: ______________________ → ______________

name domain range

\# ____

examples:

______________ (__________) **is** ______________________________

______________ (__________) **is** ______________________________

end

fun ______________ (__________) **:** ______________________________

end

Part 3 – Manipulating Tables

- **Methods** are special functions that are attached to pieces of data. We use them to manipulate Tables.

- They are different from functions in several ways:
  1. Their names can't be used alone: they can only be used as part of data, separated by a dot. (For example, `animals.order-by)`
  2. Their contracts are different: they include the type of the data as part of their names. (eg, `<Table>.order-by :: (column :: String) → Table`)
  3. They have a "secret" argument, which is the data they are attached to

- We will use three **Table Methods** to manipulate our datasets:
  1. <Table>.order-by – order the rows of a table based on a column
  2. <Table>.filter – create a **subset** of the data, with only certain rows
  3. <Table>.build-column – use the columns of a table to compute a new one

- There are more table methods on the contracts page at the back of this handout

- Note that method contracts are different, effectively containing a **Type** as part of their name. For example, row-n is the method version of the get-row function:

  `<Table>``.row-n :: (index :: Number) -> Row`

Table Plan

On Kitten Day, the shelter prints up a list of all the cats in their database that are less than 2 years old, and makes nametags for them. They need a function that will help them out! Define a function called `kittens-tags`, which takes in the dataset and produces a table of kittens with the nametags column of images.

Contract and Purpose

kittens-tags :: *(animals :: Table)* → *Table*

 # Consume a table of animals, and produce a table containing kittens with nametags, sorted by name

Example Tables

Make a Start Table and a result based on that table.

`animals-table` → `get-kittens-tags(animals-table)`

name	species	age	fixed	legs	weight	adopt
Sasha	cat	1	FALSE	4	6.5	4
Toggle	dog	3	TRUE	4	48	3
Buddy	lizard	2	FALSE	4	0.3	12
Wade	cat	1	FALSE	4	3.2	4
Mittens	cat	2	TRUE	4	7.4	5

name	species	age	fixed	legs	weight	adopt	tag
Sasha	cat	1	FALSE	4	6.5	4	Sasha
Wade	cat	1	FALSE	4	3.2	4	Wade

Define the function

Use the relevant methods (circle your helper functions!), then produce a result with the new table.

```
fun kittens-tags ( animals ) :
  t = pets
  .build-column(                                   )
  .filter(                                         )
  .order-by(                                       )
  t
end
```

Define the table

Are there more columns?

Are there fewer rows?

Are the rows ordered?

Produce the result

Table Plan

It's important for animals to stay healthy, especially when they get older. The veterinarians at the shelter want to put some of the dogs on a diet! They need a regular report of all the older dogs, sorted from heaviest-to-lightest. Define a function `old-dogs-diet`, which does just that!

Contract and Purpose

____________________ :: _________________________ → __________________

____

Examples

Make a Start Table and a result based on that table.

`animals-table` → `old-dogs-diet(animals-table)`

name	species	age	fixed	legs	weight	adopt
Snowcone	cat	2	TRUE	4	6.1	5
Lucky	dog	3	TRUE	3	45.4	9
Hercules	cat	3	FALSE	4	13.4	7
Toggle	dog	3	TRUE	4	48	3
Snuggles	tarantula	2	FALSE	8	0.1	1

name	species	age	fixed	legs	weight	adopt
Lucky	dog	3	TRUE	3	45.4	9
Snowcone	cat	2	TRUE	4	6.1	5
Toggle	dog	3	TRUE	4	48	3

Define the function

Use the relevant methods (circle your helper functions!), then produce a result with the new table.

fun ____________________ (_________) :

t = ____ *Define the table*

.build-column(____________________________________ *)* *Are there more columns?*

.filter(____ *)* *Are there fewer rows?*

.order-by(______________________________________ *)* *Are the rows ordered?*

____ *Produce the result*

end

Table Plan

The shelter is tracking birth-years for all the animals who've been fixed. They need a function that takes in their database and returns a table that contains the birth-year for each one. Define `get-fixed-by-age` that will do this for them.

Contract and Purpose

_____________________ :: __________________________ → ___________________

____

Examples

Make a Start Table and a result based on that table.

`animals-table` → `get-fixed-by-age(animals-table)`

name	species	age	fixed	legs	weight	adopt
Snowcone	cat	2	TRUE	4	6.1	5
Lucky	dog	3	TRUE	3	45.4	9
Hercules	cat	3	FALSE	4	13.4	7
Toggle	dog	3	TRUE	4	48	3
Snuggles	tarantula	2	FALSE	8	0.1	1

name	species	age	fixed	legs	weight	adopt	year
Snowcone	cat	2	TRUE	4	6.1	5	2015
Lucky	dog	3	TRUE	3	45.4	9	2014
Toggle	dog	3	TRUE	4	48	3	2014

Define the function

Use the relevant methods (circle your helper functions!), then produce a result with the new table.

```
fun ______________________ (_________) :
  t = __________________________________________________   Define the table
  .build-column(_________________________________________)  Are there more columns?
  .filter(_______________________________________________)  Are there fewer rows?
  .order-by(_____________________________________________)  Are the rows ordered?
  _______________________________________________________   Produce the result
end
```

Bad Sample Tables!

For each word problem, a Sample Table must have (1) all the columns that matter, (2) a representative sample of the rows, and be in (3) random order. For each problem below, check the boxes to determine if the Sample Table meets those criteria.

1. The shelter wants to know the median age of all the cats

name	species	age	fixed	legs	pounds	weeks
Sasha	cat	1	FALSE	4	6.5	3
Mittens	cat	2	TRUE	4	7.4	5
Sunfower	cat	5	TRUE	4	8.1	10

- ☐ Relevant columns
- ☐ Representative sample of rows
- ☐ Random order

2. The shelter wants a pie chart showing all the dogs' weight

name	species	age
Fritz	dog	4
Wade	cat	2
Nibblet	rabbit	6
Daisy	dog	5

- ☐ Relevant columns
- ☐ Representative sample of rows
- ☐ Random order

3. Sort all the animals alphabetically by name

name	species	age	fixed	legs	pounds	weeks
Ada	dog	2	TRUE	4	32	3
Bo	dog	4	TRUE	4	76.1	10
Boo-boo	dog	11	TRUE	4	123	10

- ☐ Relevant columns
- ☐ Representative sample of rows
- ☐ Random order

4. Make a bar chart for all the fixed animals

name	species	age	fixed	legs	pounds	weeks
Sasha	cat	1	FALSE	4	6.5	3

- ☐ Relevant columns
- ☐ Representative sample of rows
- ☐ Random order

My Dataset

What questions do you have about your dataset? List at least three

1) ____

____

2) ____

____

3) ____

____

Table Plan – more than returning t

Dogs are generally a lot bigger heavier than cats, so the shelter wants to look at a chart of *only* the dogs to determine who needs more exercise time. Define a function `pie-dog-weight`, which will make a pie chart showing the relative weights of all the dogs in the shelter.

Contract and Purpose

______________________ :: ____________________________ → ____________________

____

Examples

Make a Start Table and a result based on that table.

`animals-table` → `pie-dog-weight(animals-table)`

name	...	weight
Snowcone	…	6.1
Lucky	…	45.4
Hercules	…	13.4
Toggle	…	48
Snuggles	…	0.1

Define the function

Use the relevant methods (circle your helper functions!), then produce a result with the new table.

```
fun ________________________ (__________) :
  t = ________________________________________________________
      ________________________________________________________
      ________________________________________________________
      ________________________________________________________
      ________________________________________________________
end
```

Define the table

Are there more columns?

Are there fewer rows?

Are the rows ordered?

Produce the result

Part 4 – Tables with (more) Structure

Open up the Dog Pedigree Dataset starter file. Let's answer a few questions!

1. Which breed has the highest mean weight?
2. Which dogs have no known parents?
3. How many generations of Ramphar Floss's pedigree have some known ancestor?
4. What is the weight of the heaviest ancestor of Oakwin Junior?

In addition to tables, Pyret (and most programming languages), have ways of defining **structured data**. Here's a **data definition** that captures the essence of a dog's pedigree, along with an example:

```
data Pedigree:
  | unknown
  | dog(name :: String,
        sex :: String,
        breed :: String,
        weight :: Number,
        dam :: Pedigree,
        sire :: Pedigree)
end
```

Write code to express the following dogs as examples (the first one is given to you):

```
milan = dog("Milan Melba", "female", "Silky Terrier", 9.6, unknown, unknown)

spy =


yorker =


thonock-pharoah =
```

We can also use **dot accessors** to get the values of particular fields. Try running these expressions:

```
thonock-pharoah.sire

milan.breed

thonock-pharoah.sire.dam

unknown.weight
```

These dot accessors give access to the fields in a particular instance. How many total instances of **dog** are in the code that defines the sample instances above?

There are a few pieces here:

- The data definition has a **type name** and several **constructors**, which in this case are unknown and dog
- We can use the constructors as functions to create **instances** of the data definition
- Each constructor has zero or more **fields**, which hold data of a particular type
- Each constructor comes with a **predicate** that recognizes instances built with that constructor

Try running the following at interactions:

```
is-dog(milan)

is-unknown(milan)

is-dog(spy)

is-dog(unknown)

is-unknown(unknown)
```

summarize :: (p :: Pedigree) → String

name | domain | range

\# Consumes a Pedigree and produces a description summarizing the dog

examples:

summarize (milan) **is** "Milan Melba is a Silky Terrier (F)"

summarize (unknown) **is** ____________

summarize (________) **is** ____________

end

fun summarize (p):

if is-unknown(p): ____________

else if is-dog(p): ____________

end

end

Try running the example inputs at interactions to see the results:

```
summarize(thonock-pharoah)
summarize(milan)
summarize(unknown)
```

generations :: (p :: Pedigree) → Number

name | domain | range

\# Consumes a Pedigree, produces the number of generations with some known ancestor

examples:

generations (milan) **is** 1

generations (unknown) **is** ____________

generations (________) **is** ____________

end

fun generations (p):

if is-unknown(p): ____________

else if is-dog(p):

____

end

end

This structure of computing the answer for the dam and the answer for the sire, and then combining them somehow, is going to come up again and again when working with Pedigrees. It's going to be so frequent that we can write up a **template** for functions that work over Pedigrees:

```
fun a-pedigree-fun(p):
  if is-unknown(p): ...
  else if is-dog(p):
    ... p.name ... p.sex ... p.breed ... p.weight ...
    ... a-pedigree-fun(p.dam) ...
    ... a-pedigree-fun(p.sire) ...
  end
end
```

heaviest :: *(p :: Pedigree)* → *Number*

name — domain — range

\# *Consumes a Pedigree, produces the weight of the heaviest ancestor or 0 if unknown*

examples:

heaviest (*milan*) **is** *9.6*

heaviest (*unknown*) **is** ________

heaviest (________) **is** ________

end

fun *heaviest* (p):

if is-unknown(p): ________

else if is-dog(p):

____

end

end

has-ancestor :: *(p :: Pedigree), (name :: String)* → *Boolean*

name domain range

\# *Consumes a Pedigree and a name, produces true if the name is an ancestor of p*

examples:

____________(__________________) **is** __________________

____________(__________________) **is** __________________

____________(__________________) **is** __________________

end

fun ____________ (_________):

____

____

____

end

Design Recipes

____________________ : __________________________ → __________________

name domain range

\# ____

examples:

______________(__________) **is** _________________________________

______________(__________) **is** _________________________________

end

fun ______________ (_________) **:** _________________________________

end

____________________ : __________________________ → __________________

name domain range

\# ____

examples:

______________(__________) **is** _________________________________

______________(__________) **is** _________________________________

end

fun ______________ (_________) **:** _________________________________

end

Design Recipes

_________________________ :: _______________________________ → ______________________
name domain range

____

examples:

    ______________(__________) **is** ________________________________
    ______________(__________) **is** ________________________________

end

fun ______________ (_________) **:** ________________________________

end

_________________________ :: _______________________________ → ______________________
name domain range

____

examples:

    ______________(__________) **is** ________________________________
    ______________(__________) **is** ________________________________

end

fun ______________ (_________) **:** ________________________________

end

Table Plan

Contract and Purpose

______________________ :: ____________________________ → ____________________

____

Examples

Make a Start Table and a result based on that table.

__________________________________ → __________________________________

Define the function

Use the relevant methods (circle your helper functions!), then produce a result with the new table.

```
fun ________________________ (__________) :
  t = _______________________________________________________
      _______________________________________________________
      _______________________________________________________
      _______________________________________________________
      _______________________________________________________
end
```

Define the table

Are there more columns?

Are there fewer rows?

Are the rows ordered?

Produce the result

Table Plan

Contract and Purpose

______________________ :: ____________________________ → ____________________

____

Examples

Make a Start Table and a result based on that table.

__________________________________ → __________________________________

Define the function

Use the relevant methods (circle your helper functions!), then produce a result with the new table.

```
fun ________________________ (__________) :
  t = ____________________________________________________
  ________________________________________________________
  ________________________________________________________
  ________________________________________________________
  ________________________________________________________
end
```

Define the table

Are there more columns?

Are there fewer rows?

Are the rows ordered?

Produce the result

Contracts

Name	Domain		Range
triangle	:: (side :: *Number*, style :: *String*, color :: *String*)	→	*Image*
circle	:: (radius :: *Number*, style :: *String*, color :: *String*)	→	*Image*
star	:: (radius :: *Number*, style :: *String*, color :: *String*)	→	*Image*
rectangle	:: (width :: *Num*, height :: *Num*, style :: *Str*, color :: *Str*)	→	*Image*
ellipse	:: (width :: *Num*, height :: *Num*, style :: *Str*, color :: *Str*)	→	*Image*
square	:: (size :: *Number*, style :: *String*, color :: *String*)	→	*Image*
text	:: (str :: *String*, size :: *Number*, color :: *String*)	→	*Image*
overlay	:: (img1 :: *Image*, img2 :: *Image*)	→	*Image*
rotate	:: (degree :: *Number*, img :: *Image*)	→	*Image*
scale	:: (factor :: *Number*, img :: *Image*)	→	*Image*
string-repeat	:: (text :: *String*, repeat :: *Number*)	→	*String*
string-contains	:: (text :: *String*, search-for :: *String*)	→	*Boolean*
num-sqr	:: (n :: *Number*)	→	*Number*
num-sqrt	:: (n :: *Number*)	→	*Number*
num-min	:: (a :: *Number, b :: Number*)	→	*Number*
num-max	:: (a :: *Number, b :: Number*)	→	*Number*
get-row	:: (t :: *Table, index :: Number*)	→	*Row*

Contracts

Name	Domain		Range
<*Table*>.row-n	:: (n :: *Number*)	→	*Row*
<*Table*>.filter	:: (test :: *(Row* → *Boolean*))	→	*Table*
<*Table*>.build-column	:: (col :: *String*, *builder* :: *(Row* → *Value)*)	→	*Table*
<*Table*>.order-by	:: (col :: *String*, ascending :: *Boolean*)	→	*Table*
mean	:: (t :: *Table*, col :: *String*)	→	*Number*
median	:: (t :: *Table*, col :: *String*)	→	*Number*
modes	:: (t :: *Table*, col :: *String*)	→	*List<Number>*
bar-chart	:: (t :: *Table*, labels :: *String*, values :: *String*)	→	*Image*
pie-chart	:: (t :: *Table*, labels :: *String*, values :: *String*)	→	*Image*
box-plot	:: (t :: *Table*, col:: *String*)	→	*Image*
freq-bar-chart	:: (t :: *Table*, values :: *String*)	→	*Image*
histogram	:: (t :: *Table*, values :: *String*, bin-width :: *Number*)	→	*Image*
scatter-plot	:: (t :: *Table*, xs :: *String*, ys :: *String*)	→	*Image*
labeled-scatter-plot	:: (t :: *Table*, labels :: *String*, xs :: *String*, ys :: *String*)	→	*Image*
lr-plot	:: (t :: *Table*, xs :: *String*, ys :: *String*)	→	*Image*
labeled-lr-plot	:: (t :: *Table*, labels :: *String*, xs :: *String*, ys :: *String*)	→	*Image*