Computation versus Programming

Last time, we talked about **computation**

```
(+ 1 (* 2 3)) → (+ 1 6) → 7
```

Programming?

Make a wanted poster... ➡

```
(define (maybe-wanted who wanted-who)
  (cond
   [(image=? who wanted-who)
    (above (text "WANTED" 32 "black") who)]
   [else
    who]))
```

We somehow wrote the function in one big, creative chunk.

Programming

Today: ***How to Design Programs***

`http://www.htdp.org/`

- Programming always requires creativity
- But a design rules can guide and focus creativity

HtDP is one presentation of the core of functional programming

The Design Recipe

Design Recipe I

Data

- Understand the input data: **`num`**, **`bool`**, **`sym`**, or **`image`**

Contract, Purpose, and Header

- Describe (but don’t write) the function

Examples

- Show what will happen when the function is done

Body

- The most creative step: implement the function body

Test

- Run the examples

Data

Choose a representation suitable for the function input

- Fahrenheit degrees ➡ `num`
- Grocery items ➡ `string`
- Faces ➡ `image`
- Wages ➡ `num`
- ...

Handin artifact: **none** for now

Contract, Purpose, and Header

Contract

Describes input(s) and output data

- **`f2c : num -> num`**
- **`is-milk? : string -> bool`**
- **`wearing-glasses? : image image image -> bool`**
- **`netpay : num -> num`**

Handin artifact: a comment

```
; f2c : num -> num
; is-milk? : string -> bool
```

Contract, Purpose, and Header

Purpose

Describes, in English, what the function will do

- Converts F-degrees `f` to C-degrees
- Checks whether `s` is a string for milk
- Checks whether `p2` is `p1` wearing glasses `g`
- Computes net pay (less taxes) for `n` hours worked

Handin artifact: a comment after the contract

```
; f2c : num -> num
; Converts F-degrees f to C-degrees
```

Contract, Purpose, and Header

Header

Starts the function using variables that are metioned in purpose

- `(define (f2c f) ....)`
- `(define (is-milk? s) ....)`
- `(define (wearing-glasses? p1 p2 g) ....)`
- `(define (netpay n) ....)`

Check: function name and variable count match contract

Handin artifact: as above, but absorbed into implementation

```
; f2c : num -> num
; Converts F-degrees f to C-degrees
(define (f2c f) ....)
```

Examples

Show example function calls an result

```
(check-expect (f2c 32) 0)
(check-expect (f2c 212) 100)

(check-expect (is-milk? "milk") true)
(check-expect (is-milk? "apple") false)
```

Check: function name, argument count and types match contract

Handin artifact: as above, after header/body

```
; f2c : num -> num
; Converts F-degrees f to C-degrees
(define (f2c f) ....)
(check-expect (f2c 32) 0)
(check-expect (f2c 212) 100)
```

Body

Fill in the body under the header

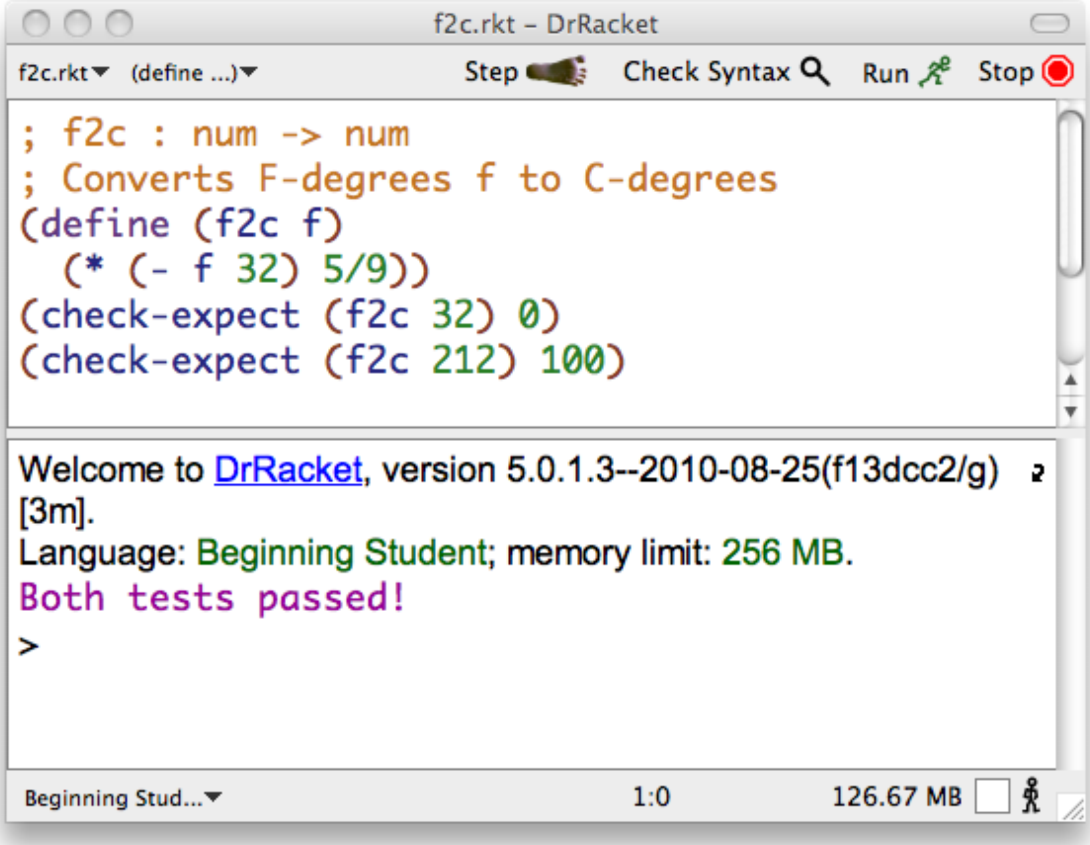
```
(define (f2c f)
  (* (- f 32) 5/9))

(define (is-milk? s)
  (string=? s "milk"))
```

Handin artifact: complete at this point

```
; f2c : num -> num
; Converts F-degrees f to C-degrees
(define (f2c f)
  (* (- f 32) 5/9))
(check-expect (f2c 32) 0)
(check-expect (f2c 212) 100)
```

Test

Click **Run** — examples serve as tests



```
; f2c : num -> num
; Converts F-degrees f to C-degrees
(define (f2c f)
  (* (- f 32) 5/9))
(check-expect (f2c 32) 0)
(check-expect (f2c 212) 100)
```

```
Welcome to DrRacket, version 5.0.1.3--2010-08-25(f13dcc2/g)
[3m].
Language: Beginning Student; memory limit: 256 MB.
Both tests passed!
>
```

Design Recipe — Each Step Has a Purpose

Data

- Shape of input data will drive the implementation

Contract, Purpose, and Header

- Provides a first-level understanding of the function

Examples

- Gives a deeper understanding and exposes specification issues

Body

- The implementation is the whole point

Test

- Evidence that it works

Compound Data

Compound Data So Far

A `posn` is

`(make-posn X Y)`

where `X` is a `num` and `Y` is a `num`

- `(make-posn 1 2)` is a value
- `(posn-x (make-posn 1 2))` → `1`
- `(posn-y (make-posn 1 2))` → `2`

So much for computation... how about program design?

Body

If the input is compound data, start the body by selecting the parts

Body

If the input is compound data, start the body by selecting the parts

```
; max-part : posn -> num
; Return the X part of p is it's bigger
; than the Y part, otherwise the Y part
(define (max-part p)
  ...)


(check-expect (max-part (make-posn 10 11)) 11)
(check-expect (max-part (make-posn 7 5)) 7)
```

Body

If the input is compound data, start the body by selecting the parts

```
; max-part : posn -> num
; Return the X part of p is it's bigger
; than the Y part, otherwise the Y part
(define (max-part p)
  ... (posn-x p) ... (posn-y p) ...)


(check-expect (max-part (make-posn 10 11)) 11)
(check-expect (max-part (make-posn 7 5)) 7)
```

Body

If the input is compound data, start the body by selecting the parts

```
; max-part : posn -> num
; Return the X part of p is it's bigger
; than the Y part, otherwise the Y part
(define (max-part p)
  (cond
   [(> (posn-x p) (posn-y p)) (posn-x p)]
   [else (posn-y p)]))
(check-expect (max-part (make-posn 10 11)) 11)
(check-expect (max-part (make-posn 7 5)) 7)
```

Body

If the input is compound data, start the body by selecting the parts

```
; max-part : posn -> num
; Return the X part of p is it's bigger
; than the Y part, otherwise the Y part
(define (max-part p)
  (cond
   [(> (posn-x p) (posn-y p)) (posn-x p)]
   [else (posn-y p)]))
(check-expect (max-part (make-posn 10 11)) 11)
(check-expect (max-part (make-posn 7 5)) 7)
```

Since this guideline applies before the usual body work, let’s split it into an explicit step

Design Recipe II

Data

- Understand the input data

Contract, Purpose, and Header

- Describe (but don’t write) the function

Examples

- Show what will happen when the function is done

Template

- Set up the body based on the input data (and *only* the input)

Body

- The most creative step: implement the function body

Test

- Run the examples

~~Body~~ Template

If the input is compound data, start the body by selecting the parts

```
; max-part : posn -> num
; ...
(define (max-part p)
  ... (posn-x p) ... (posn-y p) ...)
```

Check: number of parts in template =
number of parts data definition named in contract

A `posn` is

```
(make-posn X Y)
```

where `X` is a `num` and `Y` is a `num`

~~Body~~ Template

If the input is compound data, start the body by selecting the parts

Handin artifact: a comment

```
; max-part : posn -> num
; Return the X part of p is it's bigger
; than the Y part, otherwise the Y part
; (define (max-part p)
;    ... (posn-x p) ... (posn-y p) ...)
(define (max-part p)
  ... (posn-x p) ... (posn-y p) ...)
(check-expect (max-part (make-posn 10 11)) 11)
(check-expect (max-part (make-posn 7 5)) 7)
```

Other Kinds of Data

We wanted to represent snakes:

- name
- weight
- favorite food

Data

Deciding to define `snake` is in the first step of the design recipe

Handin artifact: a comment and/or `define-struct`

```
; A snake is
;    (make-snake sym num sym)

(define-struct snake (name weight food))
```

Now that we've defined `snake`, we can use it in contracts

Expanding the Zoo

An ant has

- a weight
- a location in the zoo

```
; An ant is
;   (make-ant num posn)
(define-struct ant (weight loc))

(make-ant 0.001 (make-posn 4 5))

(make-ant 0.007 (make-posn 3 17))
```

Programming with Ants

Define **`ant-at-home?`**, which takes an ant and reports whether it is at the origin

Contract, Purpose, and Header

```
; ant-at-home? : ant -> bool
```

Contract, Purpose, and Header

```
; ant-at-home? : ant -> bool
; Check whether ant a is home
```

Contract, Purpose, and Header

```
; ant-at-home? : ant -> bool
; Check whether ant a is home
(define (ant-at-home? a)
  ...)
```

Examples

```
; ant-at-home? : ant -> bool
; Check whether ant a is home
(define (ant-at-home? a)
  ...)
```

```
(check-expect (ant-at-home? (make-ant 0.001 (make-posn 0 0)))
              true)
(check-expect (ant-at-home? (make-ant 0.001 (make-posn 1 1)))
              false)
```

Template

```
; ant-at-home? : ant -> bool
; Check whether ant a is home
(define (ant-at-home? a)
  ... (ant-weight a)
  ... (ant-loc a) ...)
```

```
(check-expect (ant-at-home? (make-ant 0.001 (make-posn 0 0)))
              true)
(check-expect (ant-at-home? (make-ant 0.001 (make-posn 1 1)))
              false)
```

Template

```
; ant-at-home? : ant -> bool
; Check whether ant a is home
(define (ant-at-home? a)
  ... (ant-weight a)
  ... (posn-at-home? (ant-loc a)) ...)
```

New template rule: data-defn reference ⇒ template reference

Add templates for referenced data, if needed, and
implement body for referenced data

```
(check-expect (ant-at-home? (make-ant 0.001 (make-posn 0 0)))
              true)
(check-expect (ant-at-home? (make-ant 0.001 (make-posn 1 1)))
              false)
```

Template

```
; ant-at-home? : ant -> bool
; Check whether ant a is home
(define (ant-at-home? a)
  ... (ant-weight a)
  ... (posn-at-home? (ant-loc a)) ...)

(define (posn-at-home? p)
  ... (posn-x p) ... (posn-y p) ...)
```

```
(check-expect (ant-at-home? (make-ant 0.001 (make-posn 0 0)))
              true)
(check-expect (ant-at-home? (make-ant 0.001 (make-posn 1 1)))
              false)
```

Body

```
; ant-at-home? : ant -> bool
; Check whether ant a is home
; (define (ant-at-home? a)
;    ... (ant-weight a)
;    ... (posn-at-home? (ant-loc a)) ...)
; (define (posn-at-home? p)
;    ... (posn-x p) ... (posn-y p) ...)
(define (ant-at-home? a)
  (posn-at-home? (ant-loc a)))
(define (posn-at-home? p)
  (and (= (posn-x p) 0) (= (posn-y p) 0)))
```

```
(check-expect (ant-at-home? (make-ant 0.001 (make-posn 0 0)))
              true)
(check-expect (ant-at-home? (make-ant 0.001 (make-posn 1 1)))
              false)
```

Shapes of Data and Templates

The shape of the template matches the shape of the data

```
; An ant is
;   (make-ant num posn)

; A posn is
;   (make-posn num num)
```

```
(define (ant-at-home? a)
  ... (ant-weight a)
  ... (posn-at-home? (ant-loc a)) ...)

(define (posn-at-home? p)
  ... (posn-x p) ... (posn-y p) ...)
```

Variants

Animals

All animals need to eat...

Define **`feed-animal`**, which takes an animal (snake, dillo, or ant) and feeds it (5 lbs, 2 lbs, or 0.001 lbs, respectively)

What is an **`animal`**?

Animal Data Definition

```
; An animal is either
;   - snake
;   - dillo
;   - ant
```

The “either” above makes this a new kind of data definition:

data with ***varieties***

Examples:

```
(make-snake 'slinky 10 'rats)

(make-dillo 2 true)

(make-ant 0.002 (make-posn 3 4))
```

Feeding Animals

```
; feed-animal : animal -> animal
; To feed the animal a
(define (feed-animal a)
  ...)
```

```
(check-expect (feed-animal (make-snake 'Slinky 10 'rats))
              (make-snake 'Slinky 15 'rats))

(check-expect (feed-animal (make-dillo 2 true))
              (make-dillo 4 true))

(check-expect (feed-animal (make-ant 0.002 (make-posn 3 4)))
              (make-ant 0.003 (make-posn 3 4)))
```

Template for Animals

For the template step...

```
(define (feed-animal a)
  ...)
```

- Is **`a`** compound data?
- Technically yes, but the definition **`animal`** doesn't have **`make-`***something*, so we don't use the compound-data template rule

Template for Varieties

Choice in the data definition

```
; An animal is either
;   - snake
;   - dillo
;   - ant
```

means **`cond`** in the template:

```
(define (feed-animal a)
  (cond
   [... ...]
   [... ...]
   [... ...]))
```

Three data choices means three **`cond`** cases

Questions for Varieties

```
(define (feed-animal a)
  (cond
   [... ...]
   [... ...]
   [... ...]))
```

How do we write a question for each case?

It turns out that

```
    (define-struct snake (name weight food))
```

provides `snake?`

```
 (snake? (make-snake 'slinky 5 'rats)) → true
 (snake? (make-dillo 2 true)) → false
 (snake? 17) → false
```

Template

```
(define (feed-animal a)
  (cond
   [(snake? a) ...]
   [(dillo? a) ...]
   [(ant? a) ...]))
```

New template rule: varieties ⇒ **`cond`**

Now continue template case-by-case...

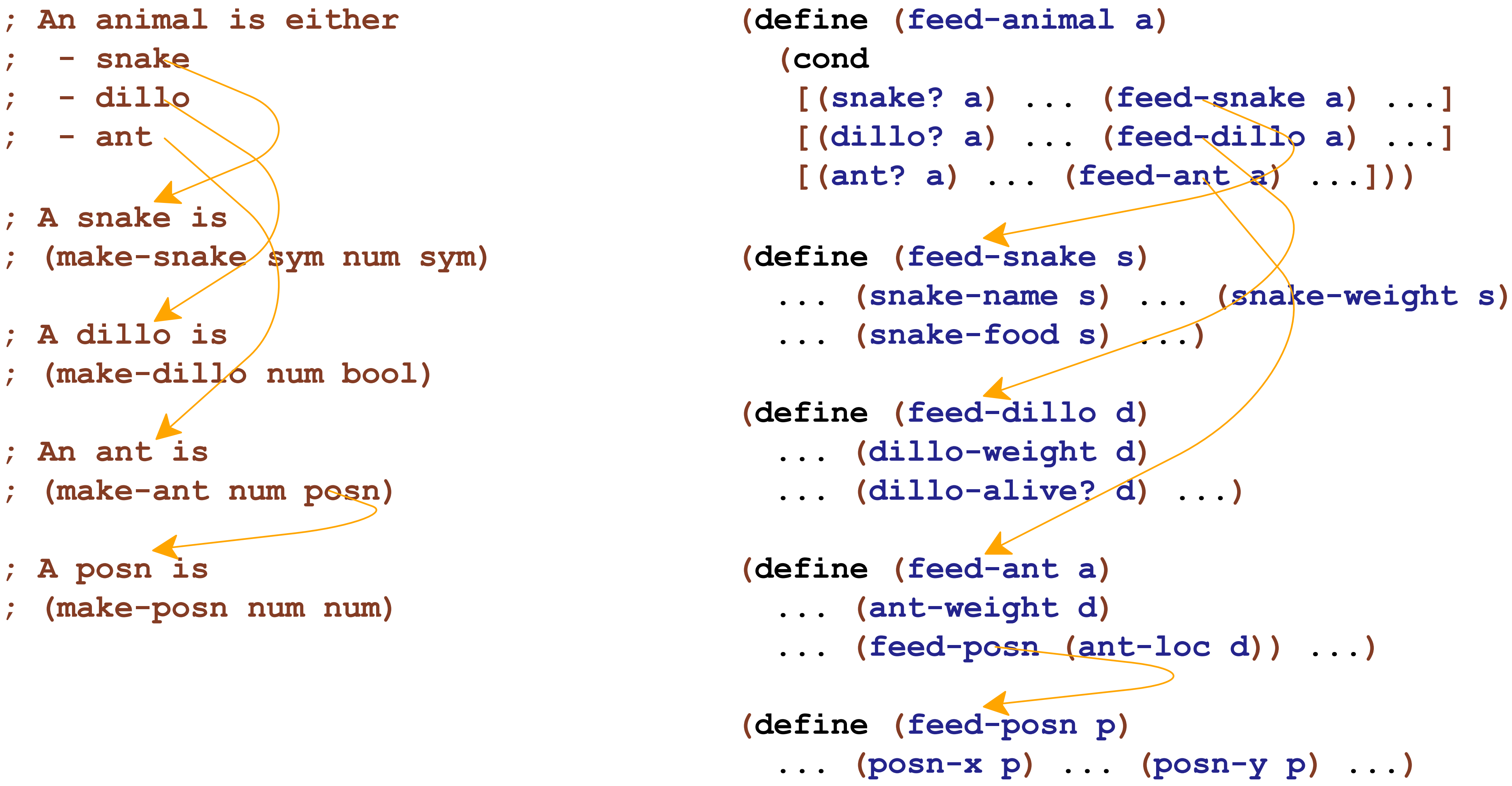
Template

```
(define (feed-animal a)
  (cond
   [(snake? a) ... (feed-snake a) ...]
   [(dillo? a) ... (feed-dillo a) ...]
   [(ant? a) ... (feed-ant a) ...]))
```

Remember: references in the data definition ⇒ template references

```
; An animal is either
;  - snake
;  - dillo
;  - ant
```

Shapes of Data and Templates



```
; An animal is either
;   - snake
;   - dillo
;   - ant

; A snake is
; (make-snake sym num sym)

; A dillo is
; (make-dillo num bool)

; An ant is
; (make-ant num posn)

; A posn is
; (make-posn num num)
```

```
(define (feed-animal a)
  (cond
   [(snake? a) ... (feed-snake a) ...]
   [(dillo? a) ... (feed-dillo a) ...]
   [(ant? a) ... (feed-ant a) ...]))

(define (feed-snake s)
  ... (snake-name s) ... (snake-weight s)
  ... (snake-food s) ...)

(define (feed-dillo d)
  ... (dillo-weight d)
  ... (dillo-alive? d) ...)

(define (feed-ant a)
  ... (ant-weight d)
  ... (feed-posn (ant-loc d)) ...)

(define (feed-posn p)
  ... (posn-x p) ... (posn-y p) ...)
```

Design Recipe III

Data

- Understand the input data

Contract, Purpose, and Header

- Describe (but don’t write) the function

Examples

- Show what will happen when the function is done

Template

- Set up the body based on the input data (and *only* the input)

Body

- The most creative step: implement the function body

Test

- Run the examples

Data

When the problem statement mentions **N** different varieties of a thing, write a data definition of the form

```
; A thing is
;   - variety1
;   ...
;   - varietyN
```

Examples

When the input data has varieties, be sure to pick each variety at least once.

```
; An animal is either
;   - snake
;   - dillo
;   - ant


(check-expect (feed-animal (make-snake 'Slinky 10 'rats))
              (make-snake 'Slinky 15 'rats))

(check-expect (feed-animal (make-dillo 2 true))
              (make-dillo 4 true))

(check-expect (feed-animal (make-ant 0.002 (make-posn 3 4)))
              (make-ant 0.003 (make-posn 3 4)))
```

Template

When the input data has varieties, start with **`cond`**

- **`N`** varieties ⇒ **`N`** **`cond`** lines
- Formulate a question to match each corresponding variety
- Continue template steps case-by-case

```
(define (feed-animal a)
  (cond
   [(snake? a) ...]
   [(dillo? a) ...]
   [(ant? a) ...]))
```

Template

When the input data has varieties, start with **`cond`**

- **`N`** varieties ⇒ **`N`** **`cond`** lines
- Formulate a question to match each corresponding variety
- Continue template steps case-by-case

When the data definition refers to a data definition, make the template refer to a template

```
(define (ant-at-home? a)
  ... (ant-weight a)
  ... (posn-at-home? (ant-loc a)) ...)

(define (posn-at-home? p)
  ... (posn-x p) ... (posn-y p) ...)
```

Template

When the input data has varieties, start with **`cond`**

- **`N`** varieties ⇒ **`N`** **`cond`** lines
- Formulate a question to match each corresponding variety
- Continue template steps case-by-case

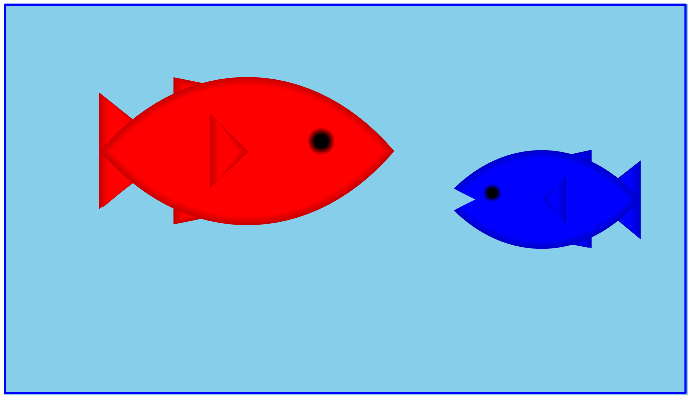
When the data definition refers to a data definition, make the template refer to a template

```
(define (feed-animal a)
  (cond
   [(snake? a) ... (feed-snake a) ...]
   [(dillo? a) ... (feed-dillo a) ...]
   [(ant? a) ... (feed-ant a) ...]))
```

Lists

Aquarium

Our zoo was so successful, let's start an aquarium



For a fish, we only care about its weight, so for two fish:

```
; An aquarium is
;   (make-aq num num)
(define-struct aq (first second))
```

Aquarium Template

```
; An aquarium is
;   (make-aq num num)
```

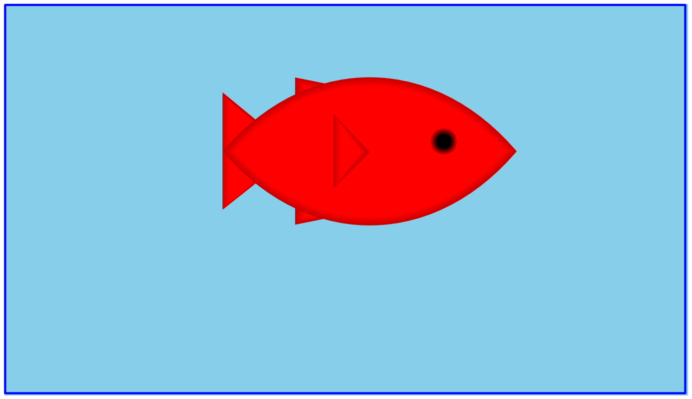
Generic template:

```
; func-for-aq : aquarium -> ...
; (define (func-for-aq a)
;    ... (aq-first a) ... (aq-second a) ...)


; aq-weight : aquarium -> num
(define (aq-weight a)
  (+ (aq-first a) (aq-second a)))

(check-expect (aq-weight (make-aq 7 8)) 15)
```

And so on, for many other simple aquarium functions...

Tragedy Strikes the Aquarium

Poor blue fish... now we have only one



Worse, we have to re-write all our functions...

```
; An aquarium is
;   (make-aq num)
(define-struct aq (first))
```

Aquarium Template, Revised

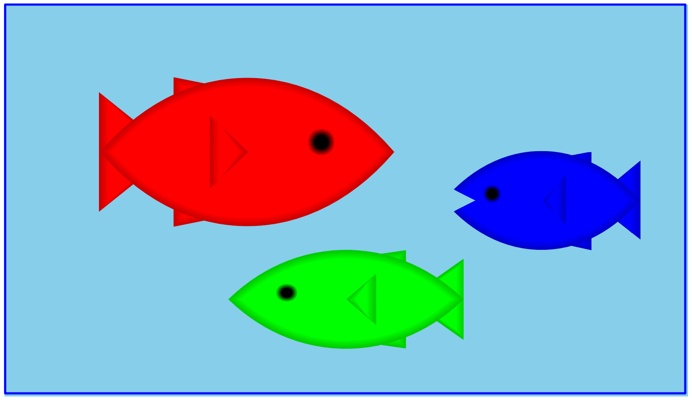
```
; An aquarium is
;   (make-aq num)

; func-for-aq : aquarium -> ...
; (define (func-for-aq a)
;    ... (aq-first a) ...)
```

```
; aq-weight : aquarium -> num
(define (aq-weight a)
  (aq-first a))

(check-expect (aq-weight (make-aq 7)) 7)
```

And so on, for **all** of the aquarium functions...

The Aquarium Expands

Hooray, we have two new fish!



Unfortunately, we have to re-re-write all our functions...

```
; An aquarium is
;   (make-aq num num num)
(define-struct aq (first second third))
```

A Flexible Aquarium Representation

Our data choice isn’t working

- An aquarium isn’t just 1 fish, 2 fish, or 100 fish—it’s a collection containing an arbitrary number of fish
- No data definition with just 1, 2, or 100 numbers will work

To represent an aquarium, we need a ***list*** of numbers

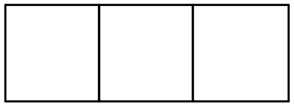
We don’t need anything new in the language, just a new idea

Structs as Boxes

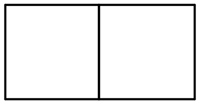
Pictorially,

- **`define-struct`** lets us define a new kind of box
- The box can have as many compartments as we want, but we have to pick how many, once and for all

```
(define-struct snake (name weight food))
```

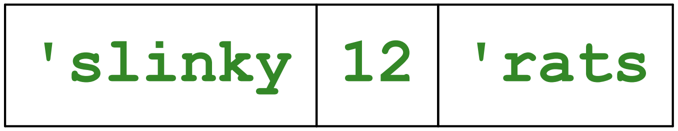
⇒ 

```
(define-struct ant (weight loc))
```

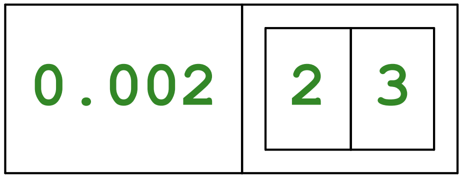
⇒ 

Boxes Stretch

The boxes stretch to fit any one thing in each slot:


'slinky 12 'rats

Even other boxes:


0.002 2 3

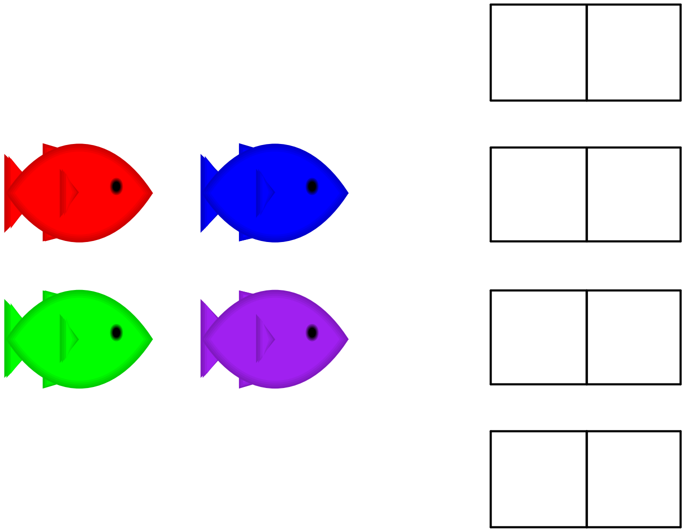
Still, the number of slots is fixed

Packing Boxes

Suppose that

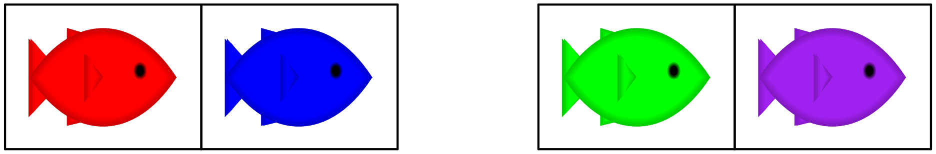
- You have four things to pack as one
- You only have 2-slot boxes
- Every slot must contain exactly one thing

How can you create a single package?



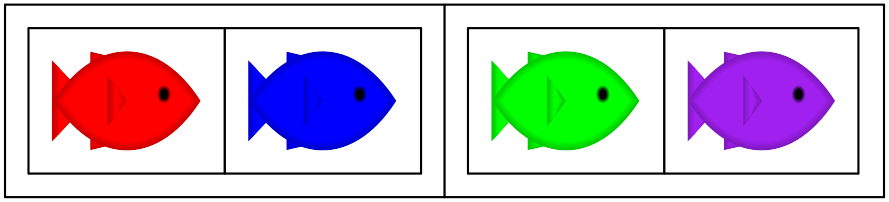
Packing Boxes

This isn’t good enough



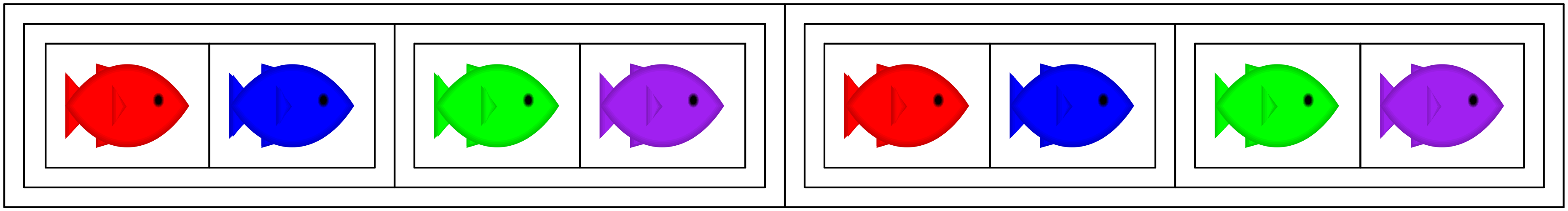
because it’s still two boxes...

But this works!

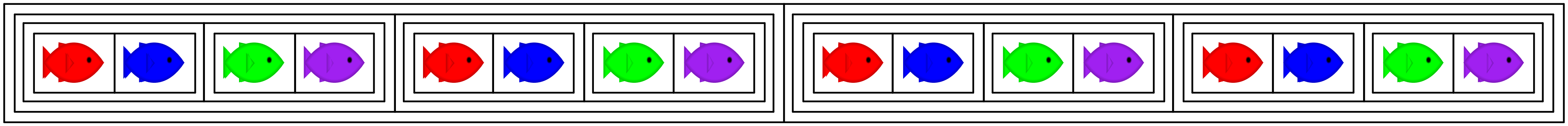


Packing Boxes

And here's 8 fish:



And here's 16 fish!



But what if we just add 1 fish, instead of doubling the fish?

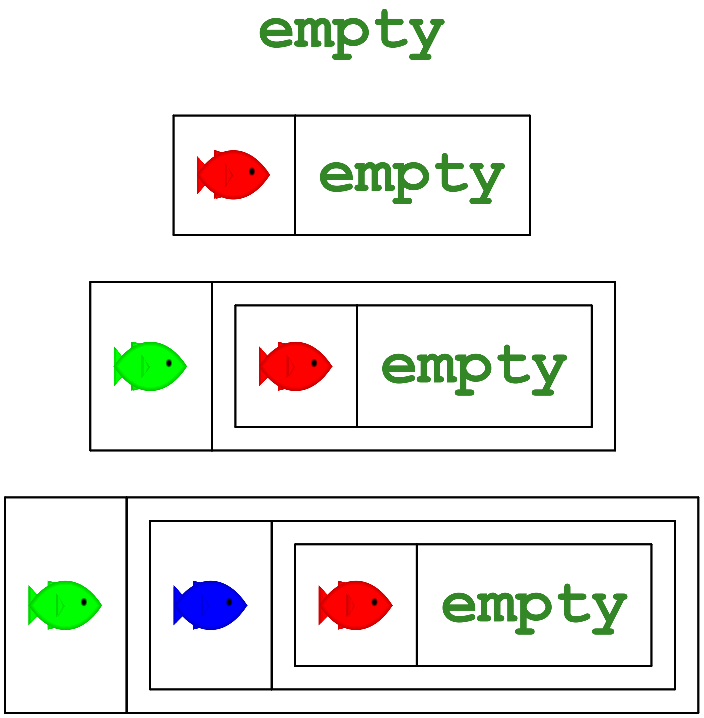
But what if we have 0 fish?

General Strategy for Packing Boxes

Here’s a general strategy:

- For 0 fish, use `empty`
- If you have a package and a new fish, put them together

To combine many fish, start with `empty` and add fish one at a time



`empty`

`empty`

`empty`

`empty`

General Strategy for a List of Numbers

To represent the aquarium as a list of numbers, use the same idea:

- For 0 fish, use `empty`
- If you have a list and a number, put them together with `make-bigger-list`

```
empty

(make-bigger-list 10 empty)

(make-bigger-list 5 (make-bigger-list 10 empty))

(make-bigger-list 7 (make-bigger-list 5 (make-bigger-list 10 empty)))
```

List of Numbers

```
; A list-of-num is either
;   - empty
;   - (make-bigger-list num list-of-num)
(define-struct bigger-list (first rest))
```

List of Numbers

```
; A list-of-num is either
;   - empty
;   - (make-bigger-list num list-of-num)
(define-struct bigger-list (first rest))
```

Generic template:

```
; func-for-lon : list-of-num -> ...
(define (func-for-lon l)
  ...)
```

List of Numbers

```
; A list-of-num is either
;   - empty
;   - (make-bigger-list num list-of-num)
(define-struct bigger-list (first rest))
```

Generic template:

```
; func-for-lon : list-of-num -> ...
(define (func-for-lon l)
  (cond
   [(empty? l) ...]
   [(bigger-list? l) ...]))
```

List of Numbers

```
; A list-of-num is either
;   - empty
;   - (make-bigger-list num list-of-num)
(define-struct bigger-list (first rest))
```

Generic template:

```
; func-for-lon : list-of-num -> ...
(define (func-for-lon l)
  (cond
   [(empty? l) ...]
   [(bigger-list? l)
    ... (bigger-list-first l)
    ... (bigger-list-rest l)
    ...]))
```

List of Numbers

```
; A list-of-num is either
;   - empty
;   - (make-bigger-list num list-of-num)
(define-struct bigger-list (first rest))
```

Generic template:

```
; func-for-lon : list-of-num -> ...
(define (func-for-lon l)
  (cond
   [(empty? l) ...]
   [(bigger-list? l)
    ... (bigger-list-first l)
    ... (bigger-list-rest l)
    ...]))
```

List of Numbers

```
; A list-of-num is either
;   - empty
;   - (make-bigger-list num list-of-num)
(define-struct bigger-list (first rest))
```

Generic template:

```
; func-for-lon : list-of-num -> ...
(define (func-for-lon l)
  (cond
   [(empty? l) ...]
   [(bigger-list? l)
    ... (bigger-list-first l)
    ... (func-for-lon (bigger-list-rest l))
    ...]))
```

Aquarium Weight

```
; aq-weight : list-of-num -> num
;  Sums the fish weights in l
(define (aq-weight l)
  ...)
```

Aquarium Weight

```
; aq-weight : list-of-num -> num
;  Sums the fish weights in l
(define (aq-weight l)
  ...)
```

```
(check-expect (aq-weight empty) 0)
```

Aquarium Weight

```
; aq-weight : list-of-num -> num
;  Sums the fish weights in l
(define (aq-weight l)
  ...)
```

```
(check-expect (aq-weight empty) 0)

(check-expect (aq-weight (make-bigger-list 2 empty))
              2)
```

Aquarium Weight

```
; aq-weight : list-of-num -> num
;  Sums the fish weights in l
(define (aq-weight l)
  ...)
```

```
(check-expect (aq-weight empty) 0)

(check-expect (aq-weight (make-bigger-list 2 empty))
              2)
```

```
(check-expect (aq-weight (make-bigger-list 5 (make-bigger-list 2 empty)))
              7)
```

Aquarium Weight

```
; aq-weight : list-of-num -> num
;  Sums the fish weights in l
(define (aq-weight l)
  (cond
   [(empty? l) ...]
   [(bigger-list? l)
    ... (bigger-list-first l)
    ... (aq-weight (bigger-list-rest l))
    ...]))


(check-expect (aq-weight empty) 0)

(check-expect (aq-weight (make-bigger-list 2 empty))
              2)

(check-expect (aq-weight (make-bigger-list 5 (make-bigger-list 2 empty)))
              7)
```

Aquarium Weight

```
; aq-weight : list-of-num -> num
;  Sums the fish weights in l
(define (aq-weight l)
  (cond
   [(empty? l) 0]
   [(bigger-list? l)
    (+ (bigger-list-first l)
       (aq-weight (bigger-list-rest l)))]))
```

```
(check-expect (aq-weight empty) 0)

(check-expect (aq-weight (make-bigger-list 2 empty))
              2)

(check-expect (aq-weight (make-bigger-list 5 (make-bigger-list 2 empty)))
              7)
```

Aquarium Weight

```
; aq-weight : list-of-num -> num
;   Sums the fish weights in l
(define (aq-weight l)
   (cond
    [(empty? l) 0]
    [(bigger-list? l)
     (+ (bigger-list-first l)
        (aq-weight (bigger-list-rest l)))]))
```

Try examples in the stepper

```
(check-expect (aq-weight empty) 0)

(check-expect (aq-weight (make-bigger-list 2 empty))
              2)

(check-expect (aq-weight (make-bigger-list 5 (make-bigger-list 2 empty)))
              7)
```

Shortcuts

The name `make-bigger-list` is awfully long

DrRacket has built-in shorter versions

`make-bigger-list`	⇒	`cons`
`bigger-list-first`	⇒	`first`
`bigger-list-rest`	⇒	`rest`
`bigger-list?`	⇒	`cons?`

```
(first (cons 1 empty)) → 1

(rest (cons 1 empty)) → empty

(cons? empty) → false
```

Lists using the Shortcuts

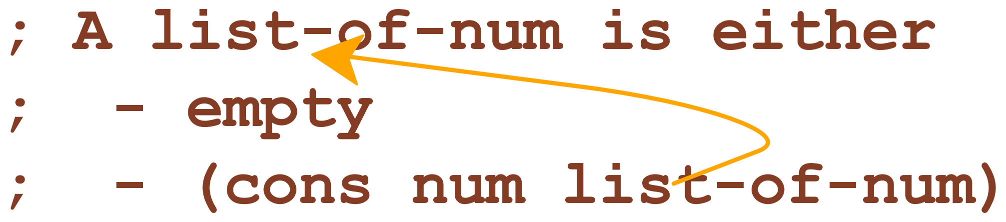
```
; A list-of-num is either
;   - empty
;   - (cons num list-of-num)

; aq-weight : list-of-num -> num
(define (aq-weight l)
  (cond
   [(empty? l) 0]
   [(cons? l) (+ (first l)
                 (aq-weight (rest l)))]))

(check-expect (aq-weight empty) 0)

(check-expect (aq-weight (cons 5 (cons 2 empty)))
              7)
```

Design Recipe for Lists

Design recipe changes for lists:

None

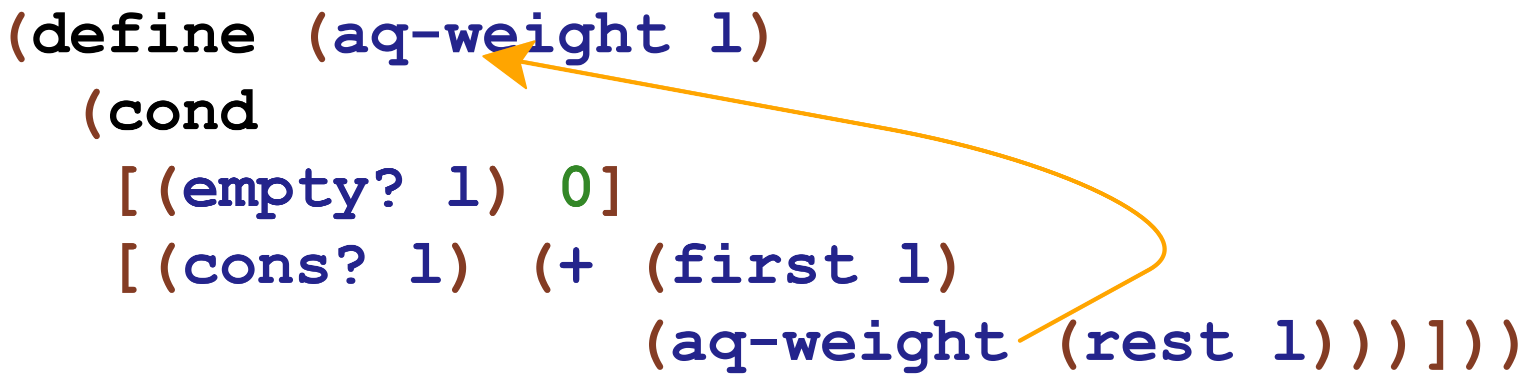
Granted, the self-reference was slightly novel...



```
; A list-of-num is either
;   - empty
;   - (cons num list-of-num)
```

Recursion

A self-reference in a data definition leads to a ***recursive*** function—one that calls itself



```
(define (aq-weight l)
  (cond
   [(empty? l) 0]
   [(cons? l) (+ (first l)
                 (aq-weight (rest l)))]))
```

Trees

More Pipes

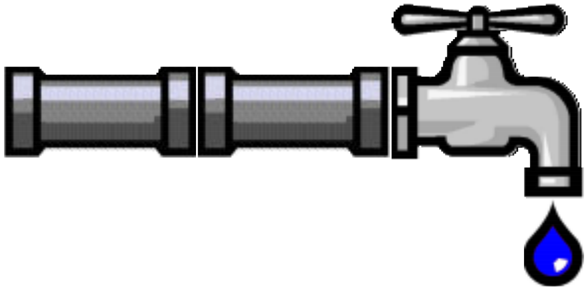
A pipeline has faucets (opened or closed), straight parts (copper or lead), and branches

More Pipes

A pipeline has faucets (opened or closed), straight parts (copper or lead), and branches

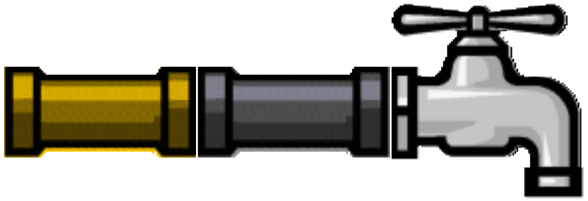
More Pipes

A pipeline has faucets (opened or closed), straight parts (copper or lead), and branches



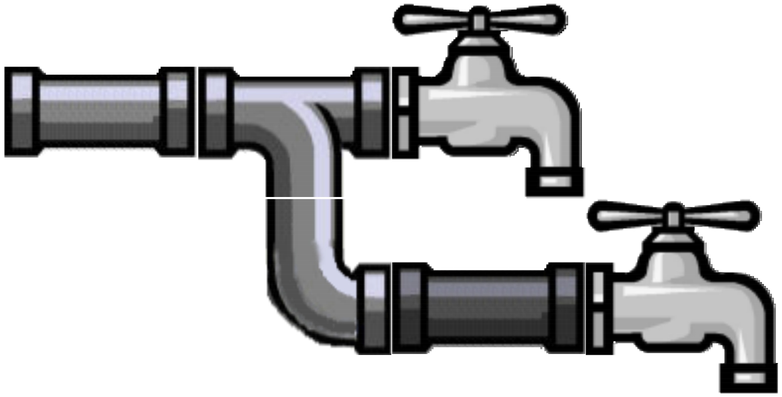
More Pipes

A pipeline has faucets (opened or closed), straight parts (copper or lead), and branches



More Pipes

A pipeline has faucets (opened or closed), straight parts (copper or lead), and branches



```
; A pipeline is either
;   - bool
;   - (make-straight sym pipeline)
;   - (make-branch pipeline pipeline)
(define-struct straight (kind next))
(define-struct branch (next1 next2))
```

Example Pipelines

```
; A pipeline is either
;   - bool
;   - (make-straight sym pipeline)
;   - (make-branch pipeline pipeline)
```

`false`

Example Pipelines

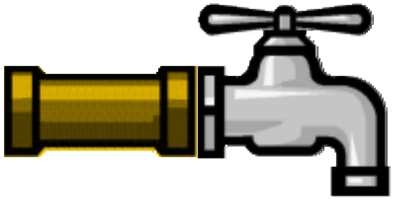
```
; A pipeline is either
;   - bool
;   - (make-straight sym pipeline)
;   - (make-branch pipeline pipeline)
```

`true`

Example Pipelines

```
; A pipeline is either
;   - bool
;   - (make-straight sym pipeline)
;   - (make-branch pipeline pipeline)
```

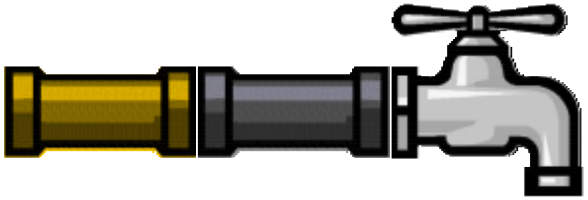
```
(make-straight 'copper false)
```



Example Pipelines

```
; A pipeline is either
;   - bool
;   - (make-straight sym pipeline)
;   - (make-branch pipeline pipeline)
```

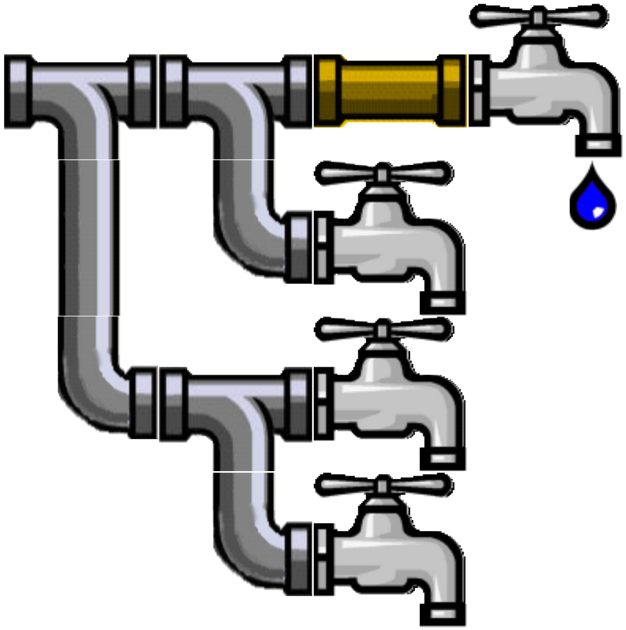
```
(make-straight 'copper
               (make-straight 'lead false))
```



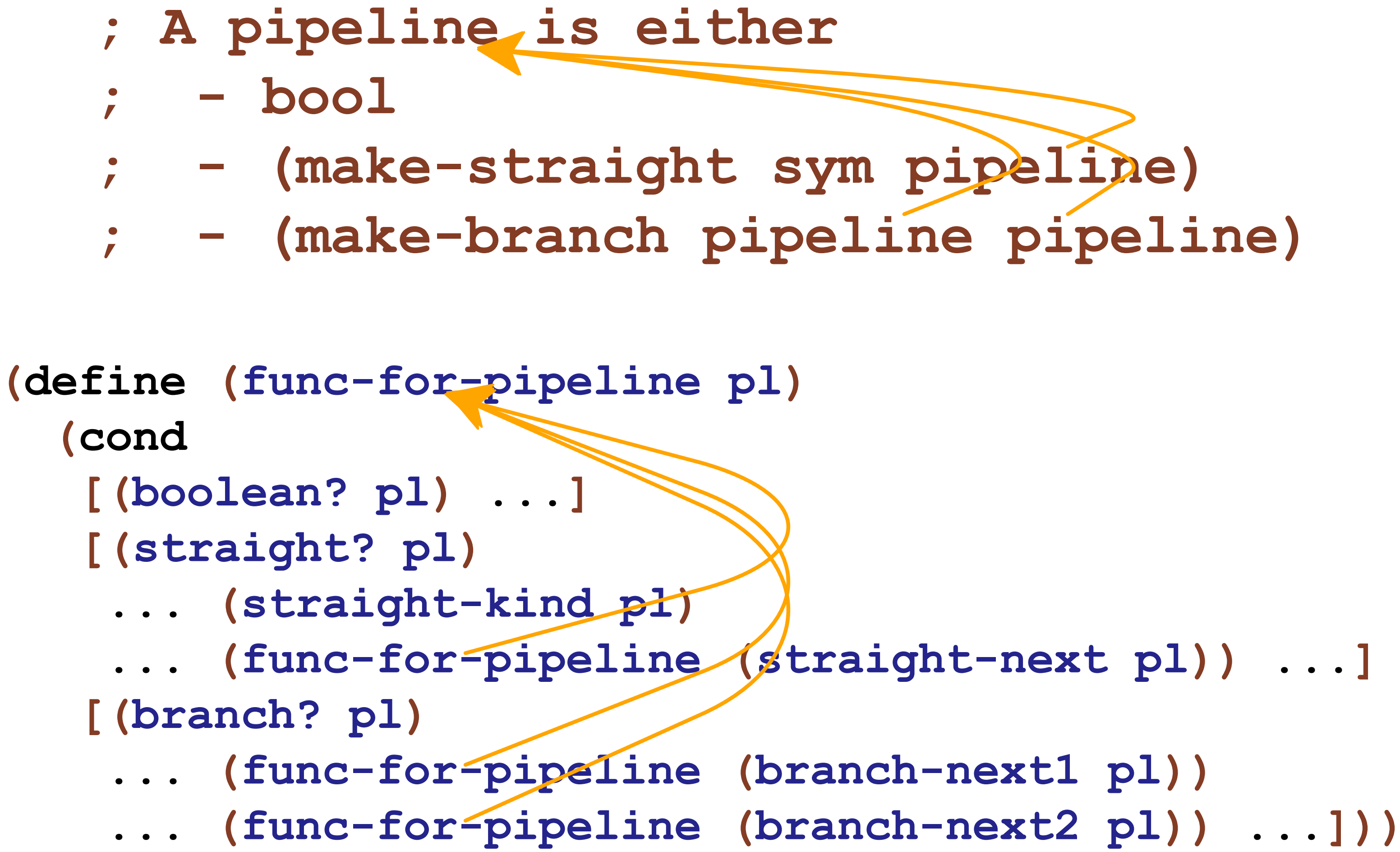
Example Pipelines

```
; A pipeline is either
;   - bool
;   - (make-straight sym pipeline)
;   - (make-branch pipeline pipeline)
```

```
(make-branch
 (make-branch (make-straight 'copper true)
              false)
 (make-branch false
              false))
```



Programming with Pipelines



```
; A pipeline is either
;   - bool
;   - (make-straight sym pipeline)
;   - (make-branch pipeline pipeline)
```

```
(define (func-for-pipeline pl)
  (cond
   [(boolean? pl) ...]
   [(straight? pl)
    ... (straight-kind pl)
    ... (func-for-pipeline (straight-next pl)) ...]
   [(branch? pl)
    ... (func-for-pipeline (branch-next1 pl))
    ... (func-for-pipeline (branch-next2 pl)) ...]))
```

Pipeline Examples

Implement the function **`water-running?`** which takes a pipeline and determines whether any faucets are open

Implement the function **`modernize`** which takes a pipeline and converts all **`'lead`** straight pipes to **`'copper`**

Implement the function **`off`** which takes a pipeline and turns off all the faucets

Implement the function **`twice-as-long`** which takes a pipeline and inserts a **`'copper`** straight pipe before every existing piece of the pipeline