Expeditiously yours, C211/A591

a lickety-split summary for the very impatient

Dan-Adrian German
dgerman@indiana.edu

aiming 2 s t r e a m l i n e
the transition to c212/a592

Last content update: January 7th, 2018 @6:25pm.
Some typos/links may have been fixed/updated since.
http://silo.cs.indiana.edu:8346/c211/impatient

Contents

Foreword

"The world has no hold on you. Whatever has a hold on you comes from your mind."

— Bill Porter, *Salesman*

Three are the intended purposes of this document: first, and foremost, it can and should serve as a brief refresher for those students that join C212/A592, whether or not[1] they have gone through C211/A591 first. Second, it can serve as a road map to students of C211/A591 at the start of the semester before they start reading thoroughly through the actual textbook[2]. Finally, it could also serve as a roadmap for any language independent HtDP2e-based course[3].

Please don't print this document as it is likely to change often and without notice, furthermore it is readily available on-line[4].

For the overall philosophy of the approach please read this[5]. From that point of view it is also hoped that these notes might serve as a good introduction for new and incoming UIs and TAs as they get started with teaching design strategies[6] instead programming for the sake of programming in a very specific (and ultimately randomly chosen) programming language[7].

Reminder: HtDP2e is language independent[8][9].

---

[1] And especially then.

[2] `http://htdp.org/2018-01-06/Book/`

[3] For example the one we've put together in Javascript

[4] `https://www.cs.indiana.edu/~dgerman/c211-for-the-impatient/evening-star.pdf`

[5] Just go ahead and click on the word "this" for now

[6] From the document online: "HtDP aims for a paradigm shift in the teaching of programming. While traditional texts introduce the expressive power of the chosen programming language, HtDP presents systematic design strategies. Its exercises do not ask students to practice the use of some linguistic features but to apply design strategies. Put differently, HtDP ignores the expressive power of programming languages in favor of the constructive power of its design strategies. This idea deserves a close look, from instructors as well as students. [...] HtDP aims to discourage students from their 'tinker until it works' habits and imbue them with a 'design attitude' from the very beginning."

[7] "Lisp isn't a language, it's a building material." – Alan Kay.

[8] Here's Alan Kay's explanation (click to go there) of his own oft-quoted statement.

[9] Extremely important clarification from Matthias Felleisen: "I [...] don't like to see Racket mentioned as the language with which we teach CS1. As a matter of fact, I have seen many students who struggle with (real) Racket after they are thru with CS1. The teaching languages (BSL, ISL, ISL/+) are carefully designed to match the cognitive stage of their development as [apprentice] coders. The key is to *enforce* these languages, to make

them real languages. Simply picking a subset and teaching the subset of a language without enforcing fails. We have seen this dozens of times as people picked subsets for text books and lectures of Algol, Simula, Pascal, Fortran. Ratfor, C, C++, Java, Python, JavaScript and you name it. Don't make co-instructors believe that they can simply list the features they will use and everything is hunky-dory after that. [...] Some people run ACL2 out of Eclipse. Does this make it the Eclipse language? Some people run Java in Emacs. Does this make it the Emacs language? DrRacket is an IDE for languages that run on the Racket VM just like Eclipse is an IDE for JVM languages. And both accommodate languages that run elsewhere. *SL is neither a subset of Racket not a superset. Due to Racket's nature, one can create a module that runs code that mixes both, but that's because of Racket's nature."

Chapter 1

Syntax

Consider the following expression:

$$3 - 4 + 5$$

Clearly it means the following:

$$((3 - 4) + 5)$$

While the use of parentheses in the second expression makes it completely unambiguous, there is an implicit assumption in the first one. It assumes that you are aware of (and will correctly apply) the known order of operations.

Here's a similar expression where you are also expected to know the exact order in which the operators should be applied[1]:

$$3 - 4 * 5$$

This expression is[2] the same as:

$$(3 - (4 * 5))$$

The mathematical notation used above, whether used with or without parentheses, is called infix[3] notation because the operators are placed in between their operands. You are probably quite familiar with the infix notation.

In 1924 the Polish logician and philosopher Jan Łukasiewicz came upon the idea of an unambiguous, parentheses-free[4] mathematical notation[5] in which operators precede the operands.

The programming language used in C211/A592 is Racket, which is a type of Scheme, which is a kind of Lisp. It is a distinctive feature of this family of languages that they use Łukasiewicz's notation in which operators will precede

[1] The asterisk denotes multiplication.
[2] Multiplication has higher precedence.
[3] `https://en.wikipedia.org/wiki/Infix_notation`
[4] As long as all operands have the same arity
[5] `https://en.wikipedia.org/wiki/Polish_notation`

their operands. Parentheses will in fact be part of our notation in Racket because humans need a bit of redundancy, and because operators in Scheme don't all have the same arity[6]. As an example, the first expression on this page translated in Polish notation looks like this:

```
+ - 3 4 5
```

The same expression in Racket looks like this[7]:

```
(+ (- 3 4) 5)
```

Likewise the second expression in Polish notation becomes:

```
- 3 * 4 5
```

In Racket, Scheme, Lisp this is written as:

```
(- 3 (* 4 5))
```

In the first lab you will be asked to convert several arithmetical and/or algebrical expressions from infix to prefix notation[8]. If you want to practice you can type converted expressions into Dr. Racket's interactions panel:

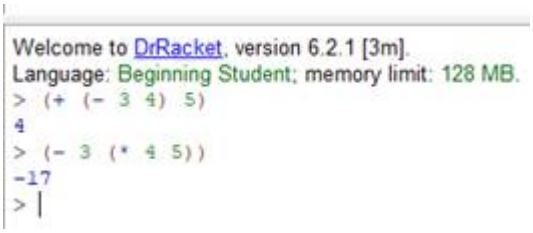

Figure 1.1: Please select Beginner Student Language before you start.

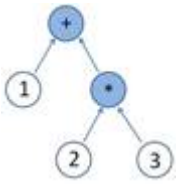

Figure 1.2: Alternative view of how `1 + 2 * 3` is in fact `(+ 1 (* 2 3))`

[6]But ask us about `https://en.wikipedia.org/wiki/Currying`

[7]Hopefully you agree parens add clarity.

[8]Given any such expression in infix notation you should first add parentheses to it to make it unambiguous. You will need to use only binary operators. Then for each expression you need to swap the operator with the first operand.

Chapter 2

Pictures

Here's a boat designed by Joseph Stahl during the Summer of 2014:

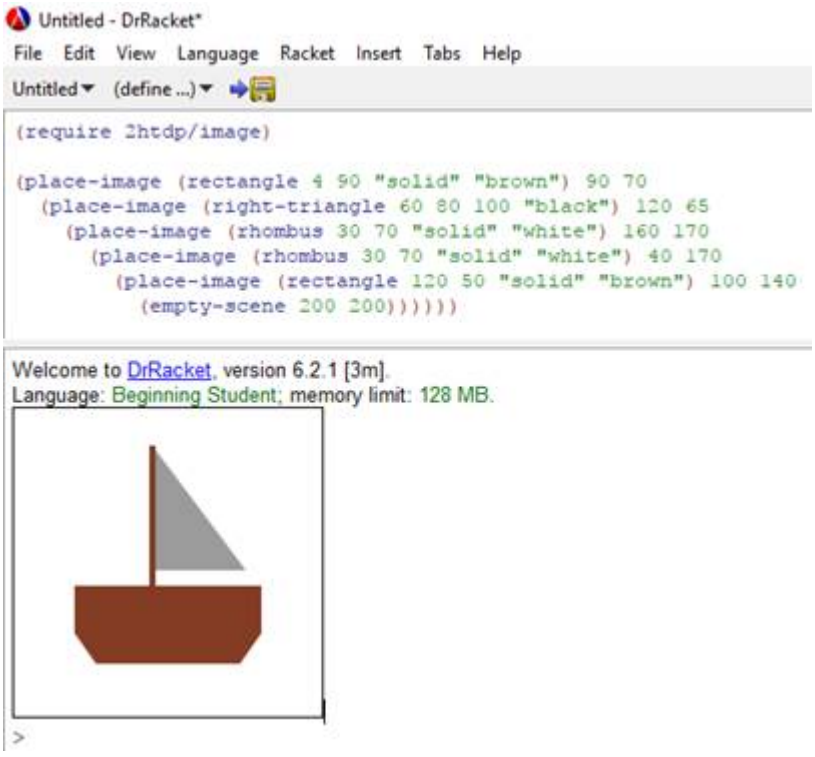

Figure 2.1: Notice the `require` at the very top.

A type is simply a set of values[1] plus a set of associated operators. Example of types: integers, strings, pictures, etc. Importing the right set of operators (the

[1] Also called literals.

way we do it with `2htdp/image` above) allows you to use them in expressions. Your program is simply an expression. What is an expression? A literal is an expression. A variable is an expression. An operator applied to a number of arguments (expressions) is also an expression. Here's the same program shown before, this time with variables being used to store the results of expressions of intermediate purpose:

```
(require 2htdp/image)

(define part1 (rectangle 120 50 "solid" "brown"))
(define part2 (rhombus 30 70 "solid" "white"))
(define part3 (rhombus 30 70 "solid" "white"))
(define part4 (right-triangle 60 80 100 "black"))
(define part5 (rectangle 4 90 "solid" "brown"))
(define backdrop (empty-scene 200 200))

(define boat
  (place-image part5 90 70
    (place-image part4 120 65
      (place-image part3 160 170
        (place-image part2 40 170
          (place-image part1 100 140 backdrop))))))
```

```
Welcome to DrRacket, version 6.2.1 [3m].
Language: Beginning Student; memory limit: 128 MB.
> boat
```

```
Beginning Student ▾                                    4:
```

Figure 2.2: The boat picture step by step.

At the beginning of this class you will practice writing a variety of expressions in BSL (which is a carefully crafted strategic subset of Racket). Some of the expressions you will need to design will ask you to create pictures like the ones shown below[2].

[2] I will develop the penguin in class

As you develop expressions please take some time to examine how they're evaluated[3]. Also, don't forget that you can include expectations[4] in your programs and we shall soon see examples.

Figure 2.3: A penguin.

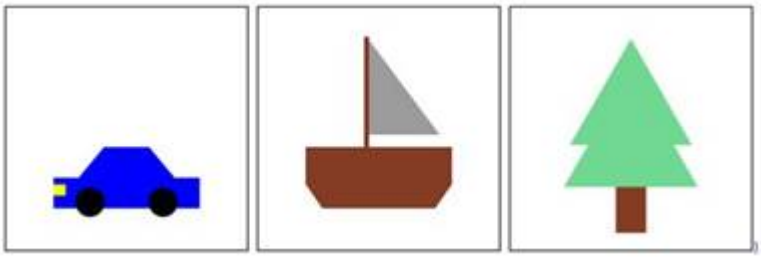

Figure 2.4: A car, a boat and a tree.

Throughout this document I will include jokes and puzzles[5] at various places in the text. Here's an example: what word becomes shorter when you add two letters to it[6]? Here are some more: what word becomes shorter when you add

[3]The Stepper tool in Dr. Racket mimics a student in a pre-algebra course
[4]Via `check-expect` for test-driven development.
[5]Let me know if you think you have an answer.
[6]Try evaluating `(string-append "short" "er")`

three letters to it? Can you prove[7] that seven is half of twelve? And finally (for now): what is the number of letters in the correct answer to this question? We end this brief chapter by showcasing some of the TDD and debugging/tracing tools existing in Dr. Racket and BSL:

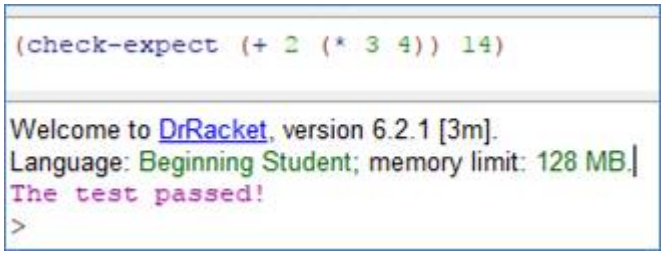

Figure 2.5: Including expectations in your code.

Figure 2.6: Stepping through any expressions starts here.

Figure 2.7: Pressing Step opens up the Stepper window.

Figure 2.8: Notice we're always being told where in the process we are.

Get comfortable working with these two tools because they will be absolutely essential to the rest of your work throughout this course.

[7] There are at least two solutions to this question.

Chapter 3

Conditionals

The following expression produces a number:

```
(+ 2 4 3)
```

The follwing expression produces a truth value:

```
(< 2 4 3)
```

Boolean[1] values can be combined with `and`, `or` and `not`. From now on we can use `if` statements as part of our expressions. This will allow your programs to branch out, with each test they take.

As an example, the code below determines if `year` is a leap year[2] or not. Note how the expression and its sub-expressions are being put together:

```
(define year 1900)

(string-append (number->string year)
               (if (integer? (/ year 4))
                   (if (zero? (remainder year 100))
                       (if (integer? (/ year 400))
                           " leap year"
                           " not a leap year")
                       " leap year")
                   " not leap year"))
```

A meaningful exercise[3] would be to simplify the expression above, reducing it to (perhaps) just a boolean expression that evaluates to `true` if the `year` is

[1] This is a new type for us.

[2] Before 1582 if a year was divisible by 4 it was a leap year. After 1582, a year is leap if it's divisible by 4 and is not divisible by 100, unless it's also divisible by 400.

[3] Another meaningful exercise would be to write code that turns a GPA (essentially a number in [0, 4]) into the letter grade whose numeric value is closest to it; break ties in favor of the better grade, for example 2.85 should be reported as B.

leap, and `false` otherwise. Alternative ways of using conditionals in your code involve using a `cond` form with clauses that more clearly indicate the cases under consideration. Consider the same code fragment from before, rewritten as follows:

```
(define year 1900)

(string-append
  (number->string year)
  (cond ((not (integer? (/ year 4))) " not a leap year")
        ((not (integer? (/ year 100))) " leap year")
        ((zero? (remainder year 400)) " leap year")
        (else " not a leap year")))
```

Simplify the following expressions[4]:

- `(if a true false)`
- `(if a false true)`
- `(false?  a)`
- `(and a false)`
- `(and a true)`
- `(or a false)`
- `(or a true)`
- `(not (false?  a))`
- `(or (< n 5) (< n 25))`
- `(and (< n 5) (< n 25))`
- `(or (> n 3) (< n 5))`
- `(and (> n 3) (< n 5))`
- `(or (< n 3) (> n 5))`
- `(if (< n 20) true (> n 10))`
- `(if (< n 10) true (> n 20))`
- `(if (< n 10) (if (> n 5) true false) (< n 20))`

Two expressions are equivalent if they have the same truth tables. An expression is a simplification of the other if it's shorter.

[4]Assume `a` is a boolean variable and `n` is a whole number.

Chapter 4

Functions

Assume you wrote code to draw a picture, for example the car we showed a while ago. Now write a function that takes the scale information from the user and draws your picture to that scale:

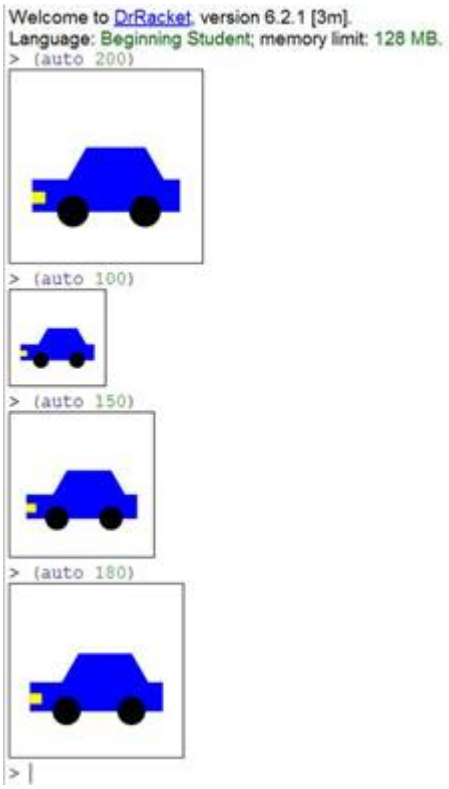

Figure 4.1: Notice the scale information here is the width in pixels.

In class I will do the same with my Penguin code:

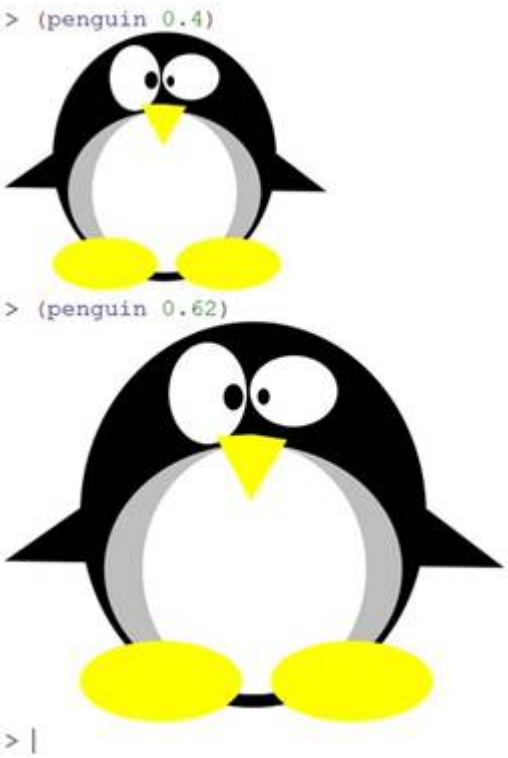

Figure 4.2: Notice scale information here is a percent of original size.

My top-level function looks like this:

```
(define (penguin scale)
  (overlay/xy ; image number number image
   (overlay/xy                          ; place this image
    (beak scale)        ; place beak    ;
    (adjust scale -200) ; here (x, y)   ;
    (adjust scale -165) ;               ;
    (skeleton scale))   ; over this one ;
   (adjust scale  -110)                 ; here (x, y)
   (adjust scale    90)                 ;
   (wings scale)))                      ; over this image
```

However, that's likely to not tell you very much at this stage. It's time for us to get organized in a serious way.

Chapter 5

The Design Recipe

Every time we need to design a function we go through these steps:

1. Choose data representation
2. Give some examples (of your data representation choice)
3. Name your function, write the signature (its contract)
4. Write purpose statement (for your function)
5. Give some examples (describing how your function will work)
6. Choose a template (but see below[1] for more accuracy)
7. Write the code (based on the template)
8. Add check-expect statements (tests)

The design recipe is a way of structuring (not eliminating) your creative process. It relies heavily on the templates used, and therein lies the tremendous appeal of this approach[2]: there are only 10 (ten) function templates in all. Six of these templates are very basic[3], one is powerful, intuitive and extremly general[4] (though very basic and simple) and the other three[5] do require a bit of practice and study but are rooted in one and the same powerful yet elementary concept: recursion[6]. Furthermore, with these 10 templates and the design

[1] From Matthias Felleisen: "It is incorrect to write that developers pick a template. Theres only one template that's fixed, the one for generative recursion. All other templates are constructed in a step by step fashion from the data definition. In the same spirit, I prefer to say we develop a data representation for the information that our program will process (with the implication we may have to backtrack and fix it). The result is a data definition. [...] Finally, I'd love to see some prose explaining how each step builds on all preceding ones."

[2] `http://htdp.org/`

[3] Atomic, intervals, enumerations, itemizations, structures, and union of structures.

[4] Functional decomposition

[5] Structural recursion, generative recursion and accumulator-passing style.

[6] An old joke says that to understand recursion you first need to understand recursion. If you understand this joke you're probably ready: reference is made to a circular, instead of a fully recursive, concept. Recursion implies circularity. However, circularity does not have a fixed point (or base case) as we shall explain soon. Recursion and induction are closely related: all recursive programs admit a proof by induction.

recipe you can design a solution to any problem, any function, regardless of what it asks. This is our promise to you: if you can write a program at all you can write it in one of these 10 ways, of which $6 + 1$ are very extremely basic.

5.1 Recursive Data

What do you call what is shown in this picture:

Figure 5.1: This is (as far as I can tell) a pile of construction rocks.

How do you describe this, then:

Figure 5.2: To me this looks like a serving of peanuts.

Here's one more:

Figure 5.3: A bunch of coins.

What do these three examples have in common? Each one of them admits a recursive definition. Let's examine that statement closely by asking a question:

what do you get if you add a rock to a pile of rocks? Answer: an(other) pile[7]. What do you get if you add a peanut to a serving of peanuts? Answer: another (slightly bigger) serving. What do you get when you add a coin to a bunch of coins? Answer: another bunch of coins, that has one more coin in it.

We have thus shown you three examples of discrete[8] structures: we call them discrete because one can count the rocks, peanuts and/or coins in them, one by one. And we've seen that in every pile of rocks there is a smaller pile that the bigger one was built on. Likewise, in every serving of peanuts there is a slightly smaller serving and in every bunch of coins there's a slightly smaller bunch, that you can get to if you remove just one coin (or peanut, for the serving).

In fact it is this operation of taking things out (as opposed to adding them in) that will help finalize the type of definitions that we are seeking. Consider this:

Figure 5.4: What is the purpose of the holes?

Many currencies (coins) have holes in them. You see a sample above, but there are others. What could be the purpose of those holes? One possibility is that in the process some of the material the coin is made of is saved. Another one is that coins can be grouped together in various arrangements[9] such as the one shown a bit later in Figure 5.5 that can be found on the next page. Perhaps you

[7]Slightly bigger, but for the time being size is not important.
[8]`https://en.wikipedia.org/wiki/Discrete_space`
[9]Purely decorative, I suppose.

can think of several other reasons for these holes. However the most obvious purpose for me is: ease of transport. This should be by and large intuitive but if you're still unsure take a look at Figure 5.6 below.

Figure 5.5: Feng Shui coins you can purchase on Amazon.

The caption below is taken from Quora[10]:

Figure 5.6: Zeni or Kozeni is Japanese word for coins or small change. Kozeni have hole in the center of the coins, sometimes round or square. They are kept using a straw rope which is called Zenisashi.

So now let's imagine a bunch of Kozeni on a Zenisashi. What if we add one coin to the bunch? We get a slightly bigger bunch[11]. Let's try going the other way now: what if we take a coin from the bunch instead of adding a new one? The answer is immediate (or, rather, easy to determine) if the bunch contains a reasonably large amount of coins. What if it contains only three[12] coins? It is still a bunch and if we take one coin out it turns into a bunch of just two coins. You see where this is going... if we take one more coin out we're left with just one coin on the Zenisashi. Is that a bunch[13]? Is it the smallest[14]

[10] `https://www.quora.com/When-was-the-Zenny-first-used-as-currency`
[11] We've gone over this already.
[12] It's a very small bunch.
[13] I don't see why not.
[14] This depends on your point of view.

bunch? What if we also remove the last coin from the thread (rope)? We end up with 0 (zero[15]) coins on a Zenisashi rope. Is the rope itself a bunch? Is there such a thing as the empty bunch? Again, the answer depends on your point of view[16] and unless explicitly stated otherwise the empty bunch is considered a very natural concept.

5.2 Recursive Definitions

Let's see if you accept this definition:

```
A Bunch-of-Coins is one of:
  -- Empty
  -- (Add-to Bunch-of-Coins Coin)
```

This is our first recursive definition. It relies on three things:

1. The existence of `Empty` as a primitive concept[17].
2. The existence of a definition for `Coin`, a user-defined type.
3. The induction/recursive step of building a bigger bunch[18].

It is now easy to define non-empty bunches:

```
A non-empty-Bunch-of-Coins is one of:
  -- Coin
  -- (Add-to non-empty-Bunch-of-Coins Coin)
```

The longest of walks begins with a step[19]. In our terms:

```
A Walk is one of:
  -- Step
  -- (Add-to Walk Step)
```

Lots of things in nature, if not all of them, admit recursive definitions.

I include for your information a couple of examples below, with pictures. A (binary) tree is one such example. A fern[20] is another, and probably the most common of these examples. L-system[21] trees also form realistic models of natural patterns. Self-similar (recursive) subsets of the two-dimensional plane are also known as fractals, since their dimension is between 1 (that of a line)

[15] The concept of zero, both as a placeholder and as a symbol for nothing, is a relatively recent development. See `https://www.livescience.com/27853-who-invented-zero.html`

[16] `https://en.wikipedia.org/wiki/Empty_set`

[17] That is also the base case we mentioned earlier.

[18] By adding one `Coin` to an existing `Bunch-of-Coins`. What this really means is that we can rely on `Add-to` as an existing, already defined operation.

[19] The empty walk makes no sense to me.

[20] `https://en.wikipedia.org/wiki/Barnsley_fern`

[21] `https://en.wikipedia.org/wiki/L-system`

and 2 (the plane itself). However the fractal dimension aspect of these objects is not the main point of interest here, their basic structure is, so please don't get distracted by this interesting yet secondary aspect.

Figure 5.7: The Barnsley Fern is one of the basic examples of self-similar sets.

Here's another definition and its associated picture:

```
A Leaf is one of:
  -- Number

A BinaryTree is one of:
  -- Leaf
  -- (Make-Tree-from-Branches BinaryTree BinaryTree)
```

Try to see that into this picture:

Figure 5.8: Each branch in this binary tree is a (smaller) binary tree.

Chapter 6

Structures

Let's start by defining a `Coin`:

```
; A Coin is a (make-coin String Number)
(define-struct coin (country value))
```

The purpose is to attach a value (in USD) to a foreign currency.

```
(define euro (make-coin "Europe" 1.14))
(define yen (make-coin "Japan" 0.0083))
```

With this (and the ability to define structures) we have:

```
; A Bunch (of Coin) is one of
; -- (make-mt)
; -- (make-bunch Coin Bunch)
(define-struct mt ())
(define-struct bunch (first rest))
```

Let's see some examples now:

```
(define a (make-mt))
(define b (make-bunch yen a))
(define c (make-bunch yen b))
(define d (make-bunch euro c))
(define f (make-bunch yen d))
```

We want to write a program that counts the number of coins in a given bunch.

Discussion about structures normally starts with `define-struct`. Up until now (and still a little bit more) we consider strings and numbers as atomic pieces of data[1]. So if we wanted to design some type of data that is made

[1] That's not exactly true, because strings are made up of individual characters and numbers have digits.

up of several pieces we need to use `define-struct` as shown here. Consider that every basketball team is made up of five players. We could define a `team` datatype that consists of five names (strings[2]) put together, for the five positions in a basketball team:

```
(define-struct team (guard forward center small shooting))
; team is (make-team String String String String String)

; team?
; make-team
; team-guard, team-forward, team-center, team-small, team-shooting

(define pacers (make-team "George" "Roy" "Lance" "Paul" "David"))
```

Recall[3] that each `struct` definition with n fields comes with $n + 2$ built-in functions: n of them are selectors, one for each of the enumerated fields, one is a constructor and the last one is a predicate (used to identify instances of the type). As an example here's how we can implement and test the `Coin` data definition that started this chapter:

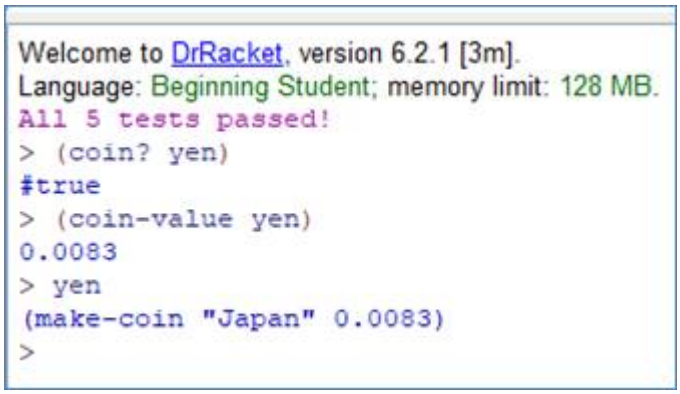

Figure 6.1: Taking the definitions for a ride

Here's how we can define each one of the starters[4] on the Cleveland Cavaliers 2016-18 team. We also can define the Cavs as a team (with a logo and individual pictures) as shown here[5]. You can click and visit the pics but I will insert them here as well, and their definitions, for your convenience.

```
(define-struct player (name age photo))
; player is a (make-player String Number Image)
; player? player-name player-age player-photo
```

[2] We could also put actual pictures.
[3] Go ahead an practice with these and other definitions to make sure all is clear so far.
[4] `http://silo.cs.indiana.edu:8346/c211/impatient/images/cavs-001.jpg`
[5] `http://silo.cs.indiana.edu:8346/c211/impatient/images/cavs-002.jpg`

Here's the first picture:

```
(define-struct player (name age photo))
; player is a (make-player String Number Image)
; player? player-name player-age player-photo
```

```
(define sf (make-player "LeBron James" 32 ))
```

```
(define pf (make-player "Kevin Love" 28 ))
```

```
(define pg (make-player "Kyrie Irving" 25 ))
```

```
(define sg (make-player "James Korver" 36 ))
```

```
(define ct (make-player "Tristan Thompson" 26 ))
```

Figure 6.2: The starters for 2016-17 Cleveland Cavaliers.

Clearly `sf` means small forward, `pf` means power forward, `pg` means point guard, `sg` means shooting guard, `ct` means center[6], as you will also see below.

Now here's how we can define a team:

```
(define-struct team (logo point-guard shooting-guard center power-forward small-forward)
; team is a (make-team Image Player Player Player Player Player)
```

This definition relies on the previous one. Now we can query our little database: [1] show me the photo of the Cavaliers' shooting guard; [2] what's the name

[6]This isn't a test of my basketball knowledge, so if not accurate I gladly plead guilty.

of the Cavalier's center; [3] how old is the small forward for the Cavs? The questions and the answers are shown below.

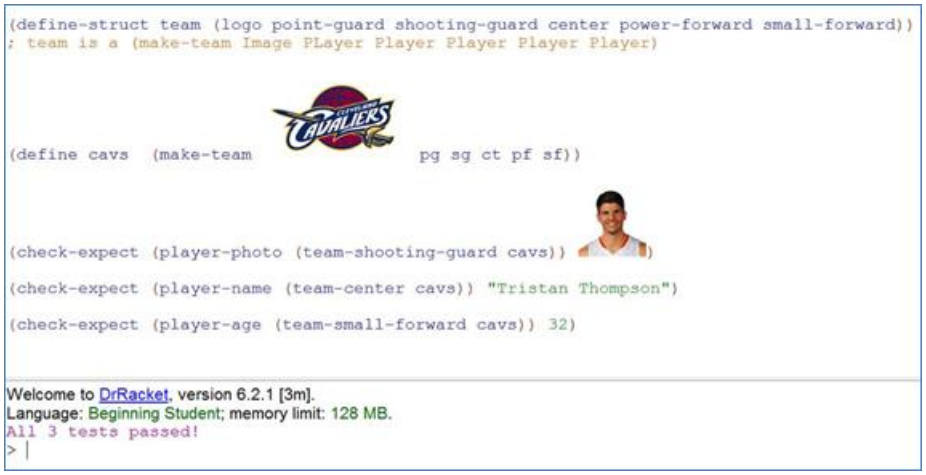

Figure 6.3: Cleveland Cavaliers: definition and some queries.

Wouldn't it be nice if we could endow these structures (as we define and design them) with functions of our own design[7], to make them a little bit more autonomous, self-aware and useful? It certainly would (if you ask me). We'll show how that can be done in C212/A592.

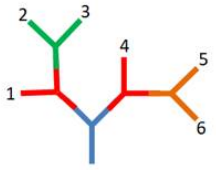

Figure 6.4: A Binary Tree, like the one shown in the previous chapter.

And the BSL definition of the picture above:

```
; A BinaryTree is one of:
;   -- Number

;   -- (make-bt BinaryTree BinaryTree)
(define-struct bt (left right))

(define ex01 (make-bt (make-bt 1 (make-bt 2 3)) (make-bt 4 (make-bt 5 6))))
```

[7]Points would be able to calculate distances to other points, Fractions would be able to add each other, Lines would be able to report their length and Triangles and Circles (and other Shapes) could report their area...

Chapter 7

Templates

Whenever we say to "design a function", we mean that you need to follow the design recipe. Any other time that you write a function in this class, you *also* need to follow the design recipe.

7.1 Atomic Template

We are going to illustrate each template with a problem. For this template consider the following: given two integers calculate the largest of the two. You can use the built-in `abs` function, addition, subtraction, division, the number 2 as well as the two numbers (your inputs, as well you should). Similar problems are those that calculate various things using a formula in closed-form (e.g., converting to Fahrenheit from Celsius, to miles from kilometers, to inches from centimeters, to meters from feet and/or yards and so on). Here's the problem and its solution:

```
; A Number is an Integer
; examples:
(define a 3)
(define b 5)
; largest : Number Number -> Number
; returns the largest of its two inputs
; examples: given 3 5 expected 5
;           given 5 4 expected 5
;           given -1 -9 expected -1
; (define (largest a b)
;   ( ... a ... b ... ) ; template says: use your inputs
(define (largest a b)
  (/ (+ (abs (- a b)) a b) 2)) ; see a and b being used?
(check-expect (largest 3 5) 5)
(check-expect (largest 5 4) 5)
(check-expect (largest -1 -9) -1)
```

7.2 Functional Decomposition

The functional decomposition template is similar to the atomic template in the sense that the atomic template also uses a creative composition of operators into a formula to solve the initial problem. The difference is that this template includes a wishlist which could lead to a set of helper functions you need to define. Here's the simplest most basic example:

```
; A [Maybe Character] is one of:
;   -- false
;   -- Character

; random-char: String -> [Maybe Character]
; pulls out and returns a random character from the input
; examples: given "" expected false
;           given "what" expected one of 'w' 'h' 'a' 't'
; (define (random-char word)
;   ( ... word ... ) ; resembles atomic template
; wishlist: string-length : String -> Number
;           random : Number -> Number
;           string-ref : String Number -> Character
; for testing:
;           string-contains? : String String -> Boolean
;           determines if first arguments appears in second
(define (random-char word)
  (if (zero? (string-length word))
      false
      (string-ref word (random (string-length word)))))

(check-expect (random-char "") false)
(check-expect (string-contains?
               (make-string 1 (random-char "something"))
               "something")
              true)
(check-expect (string-contains?
               (make-string 1 (random-char "some"))
               "thing")
              false)
```

Notice that the template says nothing about the specific problem and otherwise resembles the atomic template that simply advises the designer to just be sure to use all the function's inputs (or be aware of them). The moment you write the wishlist you start making commitments to a specific creative process[1] that is aimed at solving a specific problem. You will see this again

[1] Your own.

later when we compare the kind of recursion that stems from your thoughts (generative recursion) with the type of recursion that is dictated by a recursive data definition (structural recursion). In this respect it is worth pointing out that templates don't just help solve a problem they also help document the solution to the problem so it can be checked and understood with greater ease later. The last thing I point here is that I used a parametric type[2] in this example. If this type of parametric definition is known, I am using it here to encourage its use. If it's not, you are encouraged to check it out.

7.3 Intervals

This is the first template that uses conditionals. Consider the following problem: letter grades are A, B, C, D and F. Each letter is worth a number of points as follows: 4 points for an A, 3 points for a B, C is worth 2 points, D is worth 1 point and F is worth 0 (zero) points[3]. A + (plus) adds 0.3 points and a (minus) takes 0.3 points away from the original value; as an example B+ is worth 3.3 points, C- is worth 1.7 points and D- is worth 0.7 points. There is no F+, no A- and for the purpose of this problem, no A+. The problem asks that you design a function that receives a number and returns the letter grade that is closest to it in numeric value. This is a perfect example for the intervals template. Here are the first 5 steps of the design recipe applied to this problem:

```
; A Grade is one of:  ; data representation
;   -- "F"
;   -- "D-"
;       ...
;   -- "B+"
;   -- "A-"
;   -- "A"

; A GPA is a Number   ; data representation
; examples: 3.14 -0.16 4.97 2.85

; gpa->letter : GPA -> Grade    ; signature
; purpose statement: see problem above
; examples: given 2.85 expected B
;                 0.40          D-
;                 4.01          A+
;                -0.57          F
;                 2.7           B-
```

We can now move on to the template part:

[2] See 14.3 Similarities in Data Definitions in the text.
[3] We mentioned this problem earlier, when we discussed Conditionals.

```
; (define (gpa->letter gpa)
;   (if (<= ... gpa ...) "A+"
;     (if (<= ... gpa ...) "A")
;       ...
;         (if (<= ... gpa ...) "D-")
;             "F") ... )))
```

Clearly a `cond` would work better here. However the point is to illustrate the intervals. Here's the code:

```
(define (gpa->letter gpa)
  (if (< 4 gpa) "A+"
      (if (<= 3.85 gpa) "A"
          (if (<= 3.5 gpa) "A-"
              (if (<= 3.15 gpa) "B+"
;       2.7   3.0   3.3   3.7      4.0
;    2.5   2.85  3.15  3.5   3.85
;-----+-----+-----+-----+-----+-----+----------
;  C+ B- B- B  B  B+ B+ A- A- A  A  A  A+ A+ A+
                  (if (<= 2.85 gpa) "B"
                      (if (<= 2.5 gpa) "B-"
                          (if (<= 2.15 gpa) "C+"
                              (if (<= 1.85 gpa) "C"
                                  (if (<= 1.5 gpa) "C-"
                                      (if (<= 1.15 gpa) "D+"
                                          (if (<= .85 gpa) "D"
                                              (if (<= .35 gpa) "D-"
                                                  "F")))))))))))))
(check-expect (gpa->letter 2.85   ) "B")
(check-expect (gpa->letter 2.7    ) "B-")
(check-expect (gpa->letter 3.1499 ) "B")
```

Reminding you that the design recipe and the template are meant to structure, document and support your reasoning, not eliminate it, there is additional information you can and should include in your code in this case (as an indication of how you determined the intervals). As a reminder, given three numbers in ascending order a, b and c and x a number defined as follows:

$$x \in \left[\frac{(a+b)}{2}, \frac{(b+c)}{2}\right)$$

Then, with this definition x is closer to b than it is to a (or c) and we break ties in favor of the better grade (in this case b for the value of x equal to $\frac{a+b}{2}$).

7.4 Enumerations

This template and the next are very similar to the one just presented. Consider the key event handler that moves the spaceship[4] left, right, up or down:

```
; process-key : World Key -> World
; create new world that includes suggested change
; (basically updates the x and y according to Key)
; (define (process-key world key)
;   (cond ((key=? "up") ...)
;         ((key=? "down") ...)
;         ...
;         (else ...)))
(define (process-key world key)
  (cond ((key=? key "up"   ) (make-world (+  0 (worlc
        ((key=? key "down" ) (make-world (+  0 (worlc
        ((key=? key "left" ) (make-world (+ -1 (worlc
        ((key=? key "right") (make-world (+  1 (worlc
        (else (make-world (+ 0 (world-x world)) (+ 0
```

Figure 7.1: Basic example of enumeration template in action.

I am not including the code in its entirety here (the image is cropped) since I'd like us to just focus on the enumeration template. If you want to see and run the entire program you can find it here[5]. You can also replace the circle used in the program with an actual image, and I include here two options.

Figure 7.2: The Spaceship image from Realm of Racket.

You can find the picture above here[6][7], and the image below here[8].

Figure 7.3: Another spaceship image.

We will discuss this entire program in great detail soon.

[4] Recall this is just a super-fast summary; events are next and by the time you read this you might have seen them in class anyway.

[5] `http://silo.cs.indiana.edu:8346/c211/impatient/e001.phps`

[6] `http://silo.cs.indiana.edu:8346/c211/impatient/ufo.rkt`

[7] `http://silo.cs.indiana.edu:8346/c211/impatient/images/ufo.jpg`

[8] `http://silo.cs.indiana.edu:8346/c211/impatient/images/rocket.png`

7.5 Itemizations

Data definitions that use itemizations generalize intervals and enumerations. Because they combine more than one type we also call itemizations "unions". The distinction[9] is perhaps a bit subtle:

```
; A [Maybe X] is one of:
;   -- false
;   -- X

; add3 : [Maybe Number] -> Number
; adds 3 or leaves input unchanged
; (define (add3 input)
;   (cond ((false? input) ...)
;         ((number? input) ...)
;         (else ...)))

(define (add3 input)
  (cond ((false? input) false)
        ((number? input) (+ 3 input))
        (else (error "Bad input type."))))

(check-expect (add3 false) false)
(check-expect (add3 6) 9)
```

An itemization is therefore an enumeration of types.

7.6 Structures

This template is very simple and resembles templates 1 (Atomic) and 2 (Functional Decomposition). Recall this example from (roughly) two sections ago:

```
; A World is a (make-world Number Number)
(define-struct world (x y)) ; world? make-world world-x world-y
; examples (make-world 200 200)

; render : World -> Image
; shows the world to the user
; (draws a circle at that position)
; (define (render world)
;   (... world ... world? ... world-x ... world-y ... make-world ...))
```

The template simply wants you to remember the $n + 2$ functions defined by a structure definition with n fields.

[9] This example is also in the text (almost).

Here's what the template turns into:

```
(define (render world)
  (place-image (circle 30 "solid" "red")
               (world-x world) ; world-x
               (world-y world) ; world-y
               (empty-scene 400 400)))
```

This is something we're very familiar with by now.

7.7 Union of Structures

This is a combination of templates 5 (Itemizations) and 6 (Structures) and is what led us into the ability to build, and work with, data of arbitray size. As an illustration consider the problem where a point is a pair of two numbers, a circle is a point with a radius, a line is a pair of two points and a triangle is a triplet of points. If a shape is a circle or a triangle (union of types) we need to design the function that is able to calculate the area of a given shape.

```
; A Point is a (make-point Number Number)
(define-struct point (x y))

; A Circle is a (make-circl (center radius)
(define-struct circl (center radius))

; A Line is a (make-line Point Point)
(define-struct line (a b))

; A Triangle is a (make-triangle Point Point Point)
(define-struct triangle (a b c))

; A Shape is one of:
;   -- Circle
;   -- Triangle
```

This being the data representation take a look at our template:

```
; area : Shape -> Number
; see problem statement for purpose statement
; (define (area shape)
;   (cond ((circl? shape) ...)
;         ((triangle? shape) ...)
```

Use Heron's formula[10] for the area of a triangle.

[10] http://mathworld.wolfram.com/HeronsFormula.html

7.8 Structural Recursion

For this and the next two templates we have separate chapters. They are the three pillars of the approach that we advocate for and promote here. We include them here for the sake of completeness, so as to have them all in one place[11]. This is a book for the impatient, remember?

The Structural Recursion template is by far the most important/complex template so far. It is a direct consequence of a recursive data definition: processing follows the data structure and if data is recursively described, processing becomes naturally recursive.

Let's illustrate it through an example: design a function that calculates the sum of the numbers in a list of numbers. Then design a function that calculates the average of a list of numbers. Again, templates are meant to help you structure your thoughts, not replace or eliminate them.

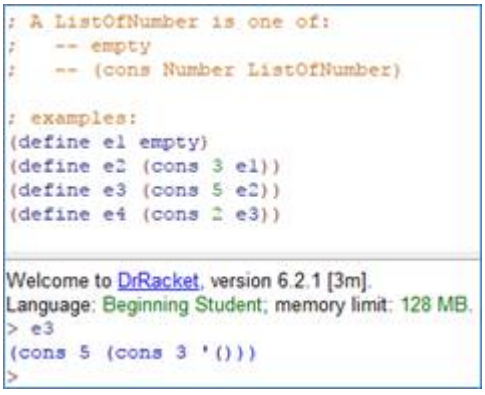

Figure 7.4: Data definition for our problem.

This might be a good time to switch to BSL with List Abbreviations:

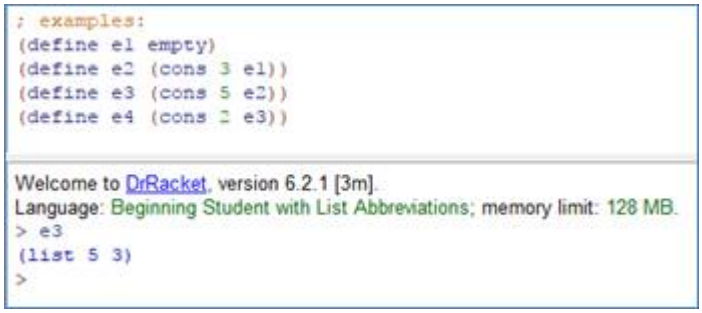

Figure 7.5: The same example run in BSL with List Abbreviations.

[11] If that is important.

Now it's time for our template:

```
; sum : ListOfNumber -> Number
; returns the sum of all numbers
; (define (sum lon)
;   (cond ((empty? lon) ...) ; some immediate answer
;         (else ( ... (first lon) ... (sum (rest lon)) ... ))))
;                                      ^ what's the promise here?
```

Repetitio est mater studiorum[12]. Practice writing these templates as you solve your homework, if you want to become an expert programmer. If you don't, the impact will be close to irrelevant. Here's the code the template becomes:

```
(define (sum lon)
  (cond ((empty? lon) 0)
        (else (+ (first lon) (sum (rest lon))))))
```

Everything in blue was already in the template. The only application specific code we added is in red. Here now is the battery of tests:

```
(check-expect (sum empty) 0)
(check-expect (sum (list 4)) 4)
(check-expect (sum (list 1 3 2)) 6)
(check-expect (sum (list -1 4 2 -3 5)) (+ -1 4 2 -3 5))
(define a (list 4 2 3 5 1))
(check-expect (sum a) (+ (first a) (sum (rest a))))
```

The last `check-expect` illustrates what we call a one step expectation. One step expectations will in fact lead us into the generative recursion approach. But before we get any further I want to provide the answer to the second question: design the function that calculates the average of a list of numbers.

Again, you can

(a) use a functional decomposition template directly since you can use `length` (or even better you can write your own) and you already have `sum`, or,

(b) you can try the structural recursion template directly.

Let's do both. With start with functional decomposition:

```
; avg : Non-Empty-ListOfNumber -> Number
; produce the average of a non-empty list of numbers
(define (avg nelon)
  (/ (sum nelon) (length nelon)))

(check-expect (avg (list 3 4 2 5)) (/ (+ 3 4 2 5) 4))
```

[12] Free translation: practice makes perfect.

It's true: in the preceding example I took a shortcut and skipped the template. I wouldn't do this in class and I most certainly won't skip that step for the next example: structural recursion. Using the structural recursion directly is certainly harder but it is a very worthwhile exercise, and one that I strongly reccomend:

```
; avg : Non-Empty-ListOfNumber -> Number
; produce the average of a non-empty list of numbers
; (define (avg nelon)
;   (cond ((empty? (rest nelon)) ...)
;         (else ( ... (first nelon) ... (avg (rest nelon)) ... ))))

(define (avg nelon)
  (cond ((empty? (rest nelon)) (first nelon))
        (else (/ (+ (first nelon)
                    (* (length (rest nelon))
                       (avg (rest nelon))))
                 (length nelon)))))
```

We again emphasize what the template brings to the completed code with color coding, in red and blue, like we already did before.

As an exercise you should try to write a `sort` function. You will need a helper function. Start by realizing that [1] the empty list is sorted. Now, if we can `(sort (rest lon))`, as our induction hypothesis, all we need to complete sorting is to [2] insert the first element into the sorted rest of the list:

```
(insert (first lon) (sort (rest lon)))
```

So you need to write `insert` which is pretty simple.

7.9 Generative Recursion

As we have seen in the previous section one can get quite creative within the structural recursion pattern. As we increase the level of creativity we might find out that our ideas (or thoughts) suggest a novel recursive approach. Sree[13] for example suggested (with respect to sorting a list of numbers) the following (general) one step expectation: if you have to sort a `(list 4 2 3 1 5)` start by determining the smallest of the list (in this case 1) and remove it from the list. Then sort the remaining numbers[14] and add the number you removed earlier, with `cons`:

```
(cons (smallest lon) (sort (remove (smallest lon) lon))))
```

[13] Sree Vankineni, H211 Fall 2015 and H212 Spring 2016

[14] This is the induction hypothesis.

Clearly now we need to write `remove` and `smallest` in addition to `sort`. There is also a second aspect worth mentioning: one does not need to provide any proof that a program written in structural recursion style is terminating. The fact follows from the earlier fact that the data structure has arbitrary size, but is nonetheless finite[15]. One however will need to attach a proof of termination for any function written in generative recursive style.

7.10 Accumulators

Using accumulators your programs become context-sensitive: they will be able to remember. Let's just give you a flavor of what that looks like[16]. Assume you're asked to write a function that calculates the n-th Fibonacci number. You could use Binet's formula[17] which is already pretty astounding, and an atomic template. Here it is:

$$F_n = \frac{\Phi^n - (-\Phi)^{-n}}{\sqrt{5}} = \frac{(1+\sqrt{5})^n - (1-\sqrt{5})^{-n}}{2^n\sqrt{5}}$$

You could use the very definition:

$$f(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ f(n-1) + f(n-2) & \text{otherwise} \end{cases}$$

Here's what happens though when we use this formula directly:

```
; f : Number -> Number
; f(4) is 5
; f(0) is 1
(define (f n)
  (cond ((zero? n) 1) ; f(0)
        ((= n 1) 1)   ; f(1)
        (else (+ (f (- n 1)) (f (- n 2))))))

(check-expect (f 5) 8)
(check-expect (f 4) 5)
```

On my machine at home[18], `f(30)` takes 906 ms (milliseconds) to compute, `f(35)` takes 9303 ms, `f(37)` takes 21217 ms and `f(40)` a full minute and a half (96924 ms). Clearly beyond 40 this process becomes prohibitive. Essentially if I wait for `f(38)` and `f(39)` to calculate I can then produce `f(40)` and all

[15] And our program emulates it.

[16] Now in addition to the termination proof one needs to provide an invariant for each accumulator. See Floyd-Hoare logic.

[17] `http://mathworld.wolfram.com/BinetsFibonacciNumberFormula.html`

[18] I have a rather old computer.

remaining Fibonacci numbers far faster than the computer will ever be able to (if it uses the definition above).

So we might come up with the following approach[19]:

```
; fibo : Number -> Number
; calculates and reports the n-th Fibonacci number
(define (fibo number)
    (local (
        ;      fold1    fold2    count    number    result
        ;  ------------------------------------------------
        ;        1        1        3       10
        ;        1        2        4
        ;        2        3        5
        ;        3        5        6
        ;        5        8        7
        ;        8       13        8
        ;       13       21        9
        ;       21       34       10
        ;                                          55 (end)
        ; helper : Number Number Number -> Number
        ; third argument is an accumulator that tells us when to stop
        ; the first two arguments are consecutive Fibonacci numbers
        ; invariants: fold1 is (fibo (- count 2))
        ;             fold2 is (fibo (- count 1))
            (define (helper fold1 fold2 count)
              (cond ((= count number) (+ fold1 fold2))
                    (else (helper fold2 (+ fold1 fold2) (add1 count)))))
        )
      (cond ((= 1 number) 1)
            ((= 2 number) 1)
            (else (helper 1 1 3)))))
```

To run this code you need to switch to ISL. For all we've seen so far there seem to be three reasons to switch to ISL: [1] you can use `time` to time your code[20]; [2] ISL comes with its own (very powerful) `sort` function, and [3] the ability to define `local` functions as seen above. This last point hints at a far-reaching truth[21]: ISL is a more powerful language than BSL. We will see soon all the why and how.

Please don't think that every problem admits a fast solution via accumulators. It's true that accumulator passing style is a very powerful technique, but some problems are intrinsically hard; the Tower of Hanoi, for example, has an exponential complexity lower bound.

---

[19] Feel free to debate the numbering in my invariants.

[20] `(time (fibo 12345))` now only takes 147 ms on my old laptop even though the resulting number (basically the 12345-th Fibonacci number) has 2580 digits.

[21] Though what we said here only represents the tip of the iceberg.

Chapter 8

Events

Go ahead and download[1] and play Joe Topp's Fishay! game. Joe was a student in Fall 2014 C211 and Spring 2015 H212. His game is a one player game in which the player controls a fish with the keyboard. The player's avatar needs to avoid all incoming fish that are bigger[2] while aiming to collide (if possible) with smaller fish. Every time you hit a smaller fish your avatar gets a little bit bigger, and you have an increased chance to survive. Here's a picture of the game in session:

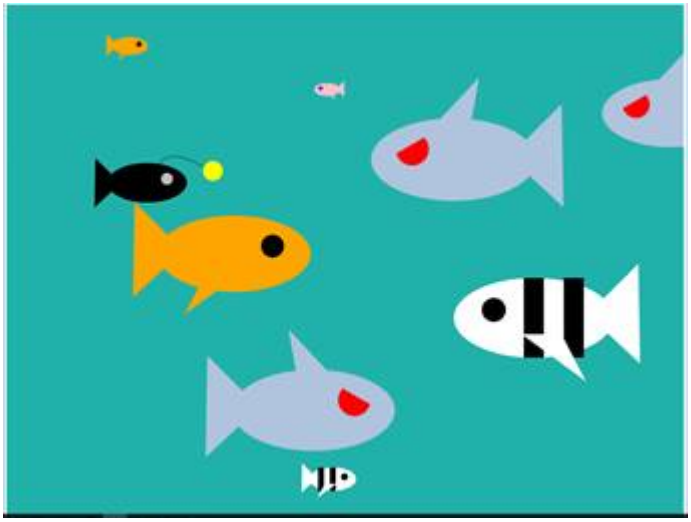

Figure 8.1: Joe Topp's Fishay! game: 100% BSL.

We start by explaining how the avatar moves in this game.

[1] `http://silo.cs.indiana.edu:8346/c211/impatient/Fishay!.rkt`

[2] The player dies and the game ends when if you run into a bigger fish.

8.1 Keyboard

We need to start, as always, with data representation:

```
(require 2htdp/image)
(require 2htdp/universe)

; A World is a (make-world Number Number)
(define-struct world (x y)) ; world? make-world world-x world-y
; examples (make-world 200 200)
```

We then design a way to render this world:

```
; render : World -> Image
; shows the world to the user
; (define (render world)
;   (... world ... world? ... world-x ... world-y ... make-world ...))
(define (render world)
  (place-image (circle 30 "solid" "red")
               (world-x world)
               (world-y world)
               (empty-scene 400 400)))
```

What would you want to happen if the user hits a key?

```
; process-key : World Key -> World
; create new world that includes suggested change
; (define (process-key world key)
;   (cond ((key=? "up") ...)
;         ((key=? "down") ...)
;         ...
;         (else ...)))
(define (process-key world key)
  (cond ((key=? key "up"   ) (make-world (+  0 (world-x world))
                                         (+ -1 (world-y world))))
        ((key=? key "down" ) (make-world (+  0 (world-x world))
                                         (+  1 (world-y world))))
        ((key=? key "left" ) (make-world (+ -1 (world-x world))
                                         (+  0 (world-y world))))
        ((key=? key "right") (make-world (+  1 (world-x world))
                                         (+  0 (world-y world))))
        (else                (make-world (+  0 (world-x world))
                                         (+  0 (world-y world))))))
```

The answer (above) is that the world changes (the circle's position in it, in fact) depending on which arrow key is hit. As of right now it's not clear who will call this function and when.

BSL comes with a form that has the following template:

```
; (big-bang ...
;           (on-tick ...)
;           (to-draw ...)
;           (on-key ...))
```

The `big-bang` form works as follows: the first argument is an initial world. Of the three clauses presented only `to-draw` is mandatory. It must be given a function of one argument (the world) that knows how to render (draw) the world. The `big-bang` form takes the initial world and renders it immediately. It then waits for user (and system) events and when such an event happens the corresponding event handler is called.

```
(big-bang (make-world 200 200)
          ; (on-tick nothing)
          (to-draw render)
          (on-key process-key))
```

The `on-tick` clause is optional, but if it were not:

```
; nothing : World -> World
; in this case nothing happens
; (define (nothing world)
;   (... world ...))
(define (nothing world)
  world)
```

As an exercise change the program so the circle has inertia: an initial velocity (with projections along the axes v_x and v_y). When the arrow keys are pressed a change in velocity (or, rather, its components) happens. The change in position should not be a direct result of pressing the keys[3], the change in velocity should. You can find a possible solution to this challenge here[4].

8.2 Mouse

Just like the `big-bang` watches for keyboard and timer events it also watches what the user is doing with the mouse. Here's `ripples` that lets user create circles at various locations in the world, and then it grows the circles as if they were ripples on the water surface of a pond[5].

[3] So if the circle is moving to the right with a certain velocity, pressing the left arrow key should slow it down, even though it will continue to move in that direction until the velocity reaches 0 (zero) and, furthermore, changes sign.

[4] `http://silo.cs.indiana.edu:8346/c212/spr2015/0422a.phps`

[5] I am very fond of this program and to the best of my recollection I trace the idea of it to the Fall 2014 C211, and I am going to say it was proposed by either Eric Wennstrom or Sam Tobin-Hochstadt, though I can't recall now which one of them actually proposed it.

We start by importing the necessary packages, then stating our data representation. A world is a (possibly empty) bunch of circles. This is the first time our world is of arbitrary size.

```
(require 2htdp/image)
(require 2htdp/universe)

; A Point is a (make-point Number Number)
(define-struct point (x y))

; A Circle is a (make-circl Point Number)
(define-struct circl (center radius))

; A ManyCircl is one of:
;   -- empty

;   -- (cons Circle ManyCircl)

; A World is a ManyCircl

; (big-bang initial
;           (to-draw ...)
;           (on-tick ...)
;           (on-mouse ...))
```

We user the structural recursion template to render the world.

```
(define initial empty)

; render : World -> Image
; (define (render world)
;   (cond ((empty? world) ...)
;         (else (... (first world) ... (render (rest world)) ...))))
(define (render world)
  (cond ((empty? world) (empty-scene 400 400))
        (else (place-image
                (circle (circl-radius (first world)) "outline" "red")
                (point-x (circl-center (first world)))
                (point-y (circl-center (first world)))
                (render (rest world))))))
```

We also define another sample initial world.

```
(define sample-world (cons (make-circl (make-point 300 300) 30)
                       (cons (make-circl (make-point 200 100) 80)
                        (cons (make-circl (make-point  50  50) 10) empty))))
```

We now need to describe what happens every time a unit of time goes by.

```
; enlarge : circl -> circl
; increases the radius by 1 pixel
(define (enlarge circl)
  (make-circl (circl-center circl) (add1 (circl-radius circl))))

; update : world -> world
;
(define (update world)
  (cond ((empty? world) world)
        (else (cons (enlarge (first world))
                    (update (rest world))))))
```

Finally, here's the mouse event handler.

```
; meh : world x y event -> world
;
(define (meh world x y event)
  (cond ((mouse=? "button-down" event)
         (cons (make-circl (make-point x y) 1)
               world))
        (else world)))
```

And we put it all together like this:

```
(big-bang sample-world
          (to-draw render)
          (on-tick update)
          (on-mouse meh))
```

8.3 Timers

Consider the following implementation of the Snake game[6]. The end scene of the game relies on this[7] function reproduced here without annotations:

```
(define (say-goodbye world)
  (place-image (text "GAME" 70 "black")
               200 150
               (place-image (text "OVER" 70 "black")
                            200 250
                            (render-world world))))
```

[6] `http://silo.cs.indiana.edu:8346/c211/impatient/snake.phps`

[7] Drawing a string as `text` (click on it) with given font size and color.

The `big-bang` form contains, starts and manages a timer. You can control it by indicating the number of seconds (represented as a number with decimals, or floating-point) between the ticks. You also define a time event handler and place its name (or, later on, its definition, if you are working in ISL+) in the `on-tick` clause. The `big-bang` form[8] starts by invoking the renderer, then with every event (be it a mouse, keyboard or timer generated event) the relevant timer will be invoked on the world, then the renderer is invoked again.

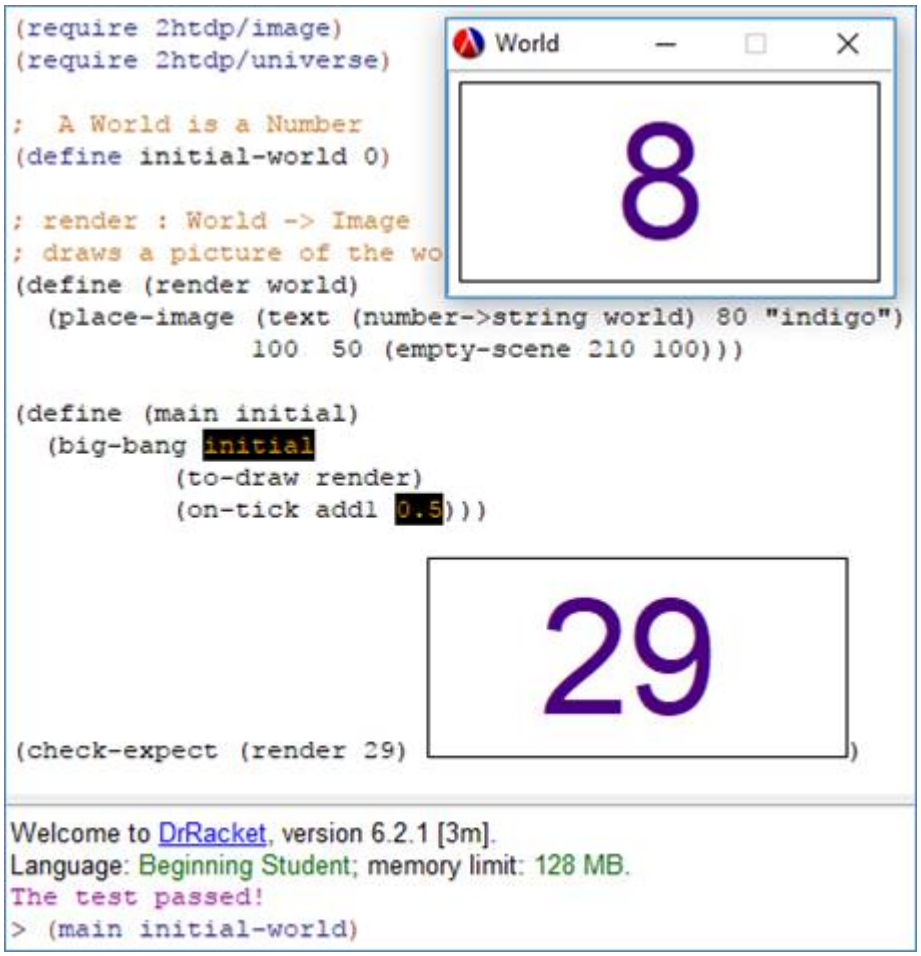

Figure 8.2: A BSL implementation of a graphical counter.

Note how `check-expect`s can include graphics. Note the parts not tested yet are shown in some sort of reversed video: initial and 0.5 (if it's hard to read).

[8] It's a macro, for those Let Over Lambda fans out there, if any...

Chapter 9

Animation

Problem solving means creatively fitting your purpose into the type of things that can be achieved with your tools. Tools have syntax associated (e.g., a hammer), that is, they can be used only as intended. There are special tools in BSL that, while mysterious, appear to be quite interesting and powerful. One such mechanism is represented by the function `animate`. The other goes by the name of `big-bang`. They both implement a modern, event-driven, type of programming. Our focus in this class remains programming by design. We now design functions that can be used in the various clauses of the `big-bang` form: `on-tick`, `to-draw`, `on-mouse`, `on-key`, etc. Templates for intervals and itemizations naturally fit the mouse and key event handlers. In class we have defined/discussed the following terms: constant, input, output and (most importantly of them) state. `animate` is a form that repeatedly calls its argument[1] on points in a discrete timeline that starts at 0 (zero) and proceeds indefinitely to plus infinity. Consider the following example:

```
(require 2htdp/image)
(require 2htdp/universe)

(define a 600) ; a is really the size of the frame

; picture : Number -> Image
; picture plots a circle at successive locations on a sinusoide
; parametric equations essentially are: x = t and y = (sin t)
(define (picture t)
  (place-image (circle 30 "solid" "red")
               (modulo t a) ; keep the circle in the frame
               (+ 300 (* 200 (sin (/ (* 5 pi t) 600)))) ; adjust the y
               (empty-scene a a)))

; (animate picture)
```

[1] This is why we need macros... this is still BSL!

If we keep x constant we get a simple harmonical oscillation as if on a spring:

```
; picture : Number -> Image
; picture plots a circle at successive locations on a sinusoide
; parametric equations essentially are: x = t and y = (sin t)
(define (picture t)
  (place-image (circle 30 "solid" "red")
               200 ; keep the circle on a vertical line
               (+ 300 (* 200 (sin (/ (* 5 pi t) 600)))) ; adjust the y
               (empty-scene a a)))
```

Essentially I'd like you to rewrite this using `big-bang` instead. Here's an alternate program to consider converting:

```
(require 2htdp/image)
(require 2htdp/universe)

(define a 400) ; a is really the size of the frame

; can you document and simplify this function
(define (y t)
  (cond ((<= 0 (modulo t (* 2 a)) (/ a 2)) (- (/ a 2) (modulo t (* 2 a))))
        ((<= (/ a 2) (modulo t (* 2 a)) a) (- (modulo t (* 2 a)) (/ a 2)))
        ((<= a (modulo t (* 2 a)) (* 3 (/ a 2))) (+ (/ a 2) (modulo t a)))
        (else (+ (/ a 2) (- (* 2 a) (modulo t (* 2 a)))))))

; same question as above (not as hard though)
(define (x t)
  (cond ((< (modulo t (* 2 a)) a) (modulo t (* 2 a)))
        (else (- a (modulo t a)))))

; picture : Number -> Image
; plotting a circle at (x(t), y(t)) where x(t) and y(t) are defined above
(define (picture t)
  (place-image (circle 30 "solid" "red")
               (x (* 2 t))
               (y (* 2 t))
               (empty-scene a a)))

; (animate picture)
```

Run it, think about it, convert it, submit it.

Chapter 10

Games

We've seen examples of games: Snake[1][2], Fishay![3] and Tetris[4]. In this section I want to briefly explain how you can add inertia to your avatar (something we discussed briefly before). Switch to ISL/+ for this, please.

```
(require 2htdp/image)
(require 2htdp/universe)

(define spaceship (circle 30 "solid" "red"))

(define (place-spaceship-at y)
  (place-image spaceship 150 y (empty-scene 400 400)))

; (place-spaceship-at 300)

(define (main initial)
  (big-bang initial
            (on-tick sub1)
            (to-draw place-spaceship-at)
            (stop-when zero?)
            (on-key (lambda (world key) 0))))
```

Can you determine how the program behaves before you run it[5]?

In this chapter I am going to be a little wild. If I decide it's not working maybe I will rewrite it. However, until then, I am going to do two things: (a) I am going to use one feature of ISL/+ that I believe to be very intuitive but that I have not explained yet[6] and (b) I am going to be a little loose with the

[1] `http://silo.cs.indiana.edu:8346/c211/impatient/snake.rkt` in ISL+
[2] By C211/A591 during the 6W1 Summer 2017 session.
[3] `http://silo.cs.indiana.edu:8346/c211/impatient/Fishay!.rkt` in BSL by Joe Topp
[4] `http://silo.cs.indiana.edu:8346/c211/impatient/tetris.rkt` by David van Horn.
[5] It hurts, without annotations, doesn't it?
[6] Though I will explain it here.

annotations[7]. So now pay attention to the data representation for the world below, so we can understand how the entire program works.

```
(require 2htdp/image)
(require 2htdp/universe)

(define spaceship (circle 30 "solid" "red"))
                  ; or you can put an image[8] instead
```

Here's an example:

Figure 10.1: The spaceship I am using in my code.

Here's another picture:

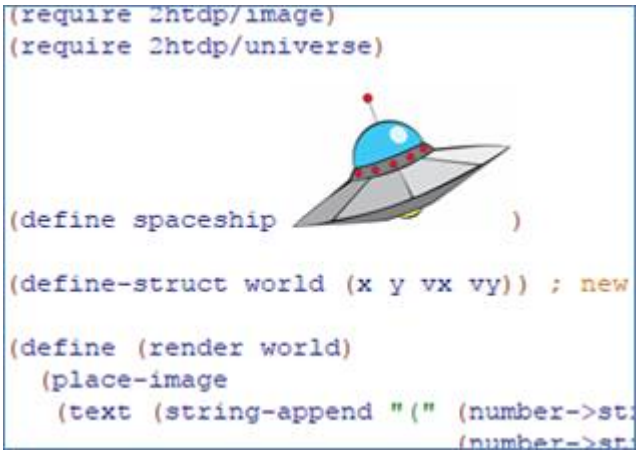

Figure 10.2: How easy it is to switch to a(nother) picture.

Finally, it's important to understand how `render` works.

The goal is to represent the spaceship at (and along with) the x and y coordinates that it has at the moment. We are talking only rendering here, not what happens when a time interval ends and a clock tick is detected[9]. We are also not talking about any of the key events[10]. We're just talking about the

[7] You can provide what I do not, as an exercise, as you study the code.

[8] `http://silo.cs.indiana.edu:8346/c211/impatient/images/image036.jpg`

[9] In that case the position of the spaceship is amended with the values of v_x and v_y as they appear in the world state at the moment. Notice that if at least one of them is $\neq 0$ the spaceship moves by itself and you may need several key events to stop it on either axis.

[10] The arrow keys, when detected, modify the values of v_x or v_y by ± 1.

rendering function.

Here's how the program looks at a certain moment frozen in time:

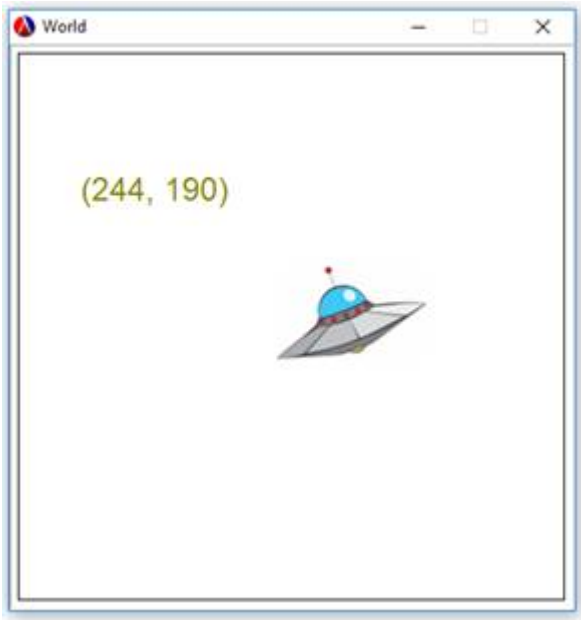

Figure 10.3: Both the spaceship and it coordinates are shown, in real time.

So now you can understand how the function below works:

```
(define-struct world (x y vx vy)) ; new state

(define (render world)
  (place-image
   (text
    (string-append
     "(" (number->string (world-x world)) ", "
         (number->string (world-y world))
     ")"
    ) ; end of string-append call
    24 ; size of font
    "olive") ; end of text call (font color is olive)
   100 100 ; x and y coordinates for text
   (place-image ;; place the spaceship image on the empty scene
     spaceship (world-x world) (world-y world) (empty-scene 400 400))))
               ; ^ x and        ^ y of the spaceship
```

The world is taken apart to get the coordinates of the spaceship. The space ship is placed on the empty scene and on the resulting image the coordinates

are shown as an ordered pair. Their location is (fixed) at: $(100, 100)$.

We can now take a look at the three[11] remaining function definitions:

```
(define (update world)
  (make-world (+ (world-x world) (world-vx world))
              (+ (world-y world) (world-vy world))
              (world-vx world)
              (world-vy world)))

(define (main initial)
  (big-bang initial
            (on-tick update)
            (to-draw render)
            (stop-when never)
            (on-key (lambda (world key)
                      (cond ((equal? key "up"   ) (make-world
                                                    (world-x world)
                                                    (world-y world)
                                                    (world-vx world)
                                                    (sub1 (world-vy world))))
                            ((equal? key "down" ) (make-world
                                                    (world-x world)
                                                    (world-y world)
                                                    (world-vx world)
                                                    (add1 (world-vy world))))
                            ((equal? key "left" ) (make-world
                                                    (world-x world)
                                                    (world-y world)
                                                    (sub1 (world-vx world))
                                                    (world-vy world)))
                            ((equal? key "right") (make-world
                                                    (world-x world)
                                                    (world-y world)
                                                    (add1 (world-vx world))
                                                    (world-vy world)))
                            (else world))))))

(define (never world) false)

(define initial (make-world 200 200 1 1))

; (main initial)
```

The key events are handled by an anonymous, inlined function.

[11] Which one is handling the key events, do you think?

Chapter 11

Structural Recursion

Consider the following data[1]:

```
(define jungle-patrol
  (cons (make-elephant "Babar")
    (cons (make-elephant "Tantor")
      (cons (make-elephant "Horton")
        empty))
```

Here's the picture as seen in the movie:

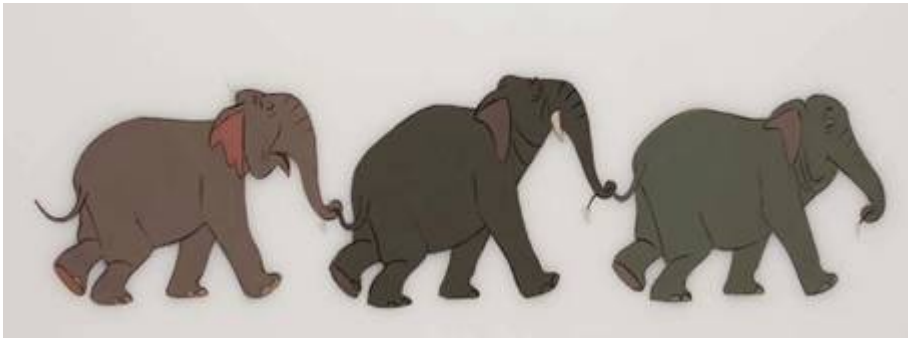

Figure 11.1: Notice that Horton is holding (that's a `cons`) onto `empty`'s tail!

The question is: will we be able to recognize (and relate to) this type of structure a lot, in our travails[2], or is this more like an exceptional, whimsical[3] case? Take numbers (and strings) for example. So far we have considered them atomic. But: are they? And if they're not, why aren't they and what are they, precisely? Intuitively we know strings are made of characters (because we have to type them one by one) and numbers are made of digits.

[1] Example worked out with Aining Wang in LH204 on 05/18/2017.
[2] Struggles, trials and tribulations, or simply put: in problem-solving.
[3] An outlier.

So what do you think of this[4]:

```
; A Digit is one of:
;     -- 0
;     -- 1
;     -- 2
;     -- 3
;     -- 4
;     -- 5
;     -- 6
;     -- 7
;     -- 8
;     -- 9

; A Number is one of:
;     -- Digit
;     -- (+ (* 10 Number) Digit)

; Examples: ...
```

Clearly this indicates that numbers are built recursively from digits. We can then design a function `lengde`[5] using the design recipe and the structural recursion template.

```
; lengde : Number -> Number
; does what (string-length (number->string ...)) does
; (define (lengde number)
;   (cond ((= 0 number) ...)
;         ((= 1 number) ...)
;         ((= 2 number) ...)
;         ((= 3 number) ...)
;         ((= 4 number) ...)
;         ((= 5 number) ...)
;         ((= 6 number) ...)
;         ((= 7 number) ...)
;         ((= 8 number) ...)
;         ((= 9 number) ...)
;         (else ( ...
;                 (modulo number 10) ; last digit
;                 ...
;                 (lengde (floor (/ number 10)))
;                 ; the part up to the last digit (like a reversed rest)
;                 ...
;                 ))))
```

[4] The first step in the design recipe is always: data representation.

[5] Means "length", in Dutch.

Here's how we flesh out the template into code:

```
(define (lengde number)
  (cond ((= 0 number) 1)
        ((= 1 number) 1)
        ((= 2 number) 1)
        ((= 3 number) 1)
        ((= 4 number) 1)
        ((= 5 number) 1)
        ((= 6 number) 1)
        ((= 7 number) 1)
        ((= 8 number) 1)
        ((= 9 number) 1)
        (else (+ 1 (lengde (floor (/ number 10)))))))

(check-expect (lengde       6) 1)
(check-expect (lengde      23) 2)
(check-expect (lengde     102) 3)
(check-expect (lengde 8723642) 7)
```

Is this the only way we can see recursive structure in numbers?

```
; A Size is a Number
;
; A Number is one of:
;   -- 0 (zero)
;   -- (+ 1 Number)

; (define (build-list size)
;   (cond ((zero? size) ...) ; the fixed point
;         (else ( ...
;                 1
;               ; ^ resembles (first lon)
;                 ...
;                (build-list (sub1 size))
;               ; ^ what is the promise here?
;               ; (sub1 size) is like (rest lon)
;                 ... ))))
```

Clearly it all always starts with the data representation. In addition, you can probably tell we're trying to design a function that builds a list of numbers of a given size. The only input here[6] is the `size` and if we can find some recursive structure in it we can apply the structural recursion template to our design[7].

[6] Besides the requirements on the kind of numbers we accept in the resulting list.

[7] ISL already provides a `build-list` function that takes a size and a generating procedure as inputs. Ours is a less ambitious project, for now.

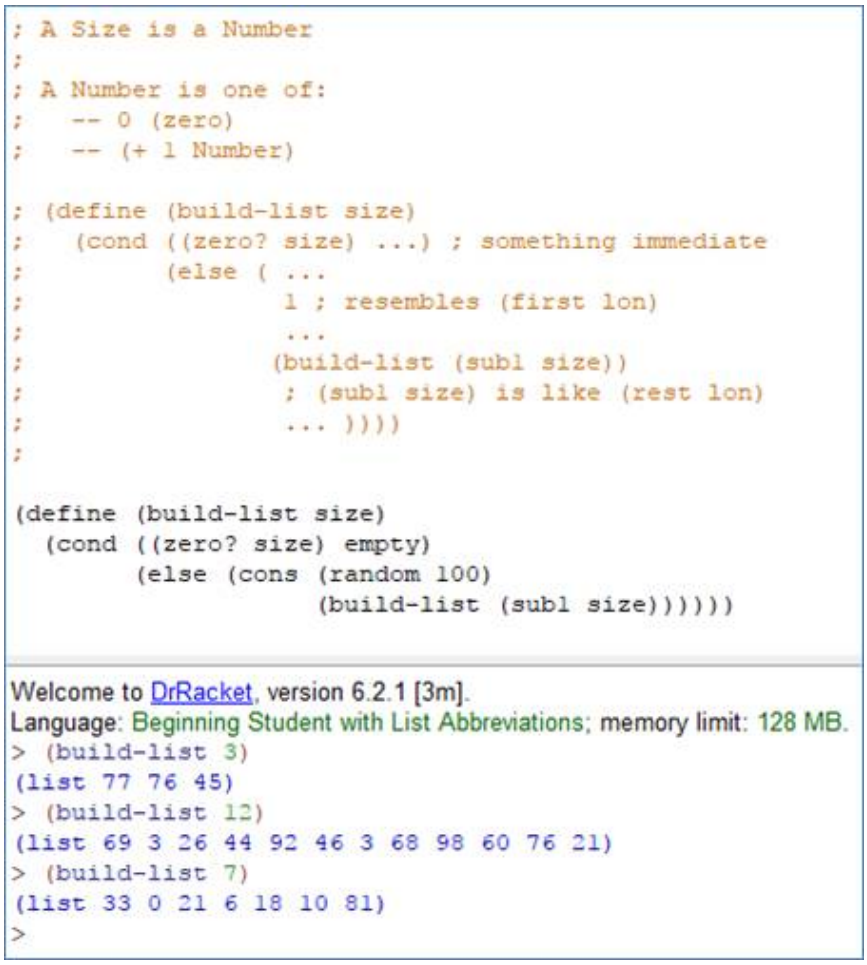

Figure 11.2: The structural recursion template is far more natural, ubiquitous and, ultimately useful than it may have seemed at first. To quote Marcel Proust: "The only true voyage would be not to travel through a hundred different lands with the same pair of eyes, but to see the same land through a hundred different pairs of eyes."

Proust exercise: There's a dolphin in this picture[8]. Can you see it?

Next I will present a number of pictures and discuss them. Daniel Brady was the UI in Summer 2014 for both C211/A591 and C212/A592. He developed a short manual for tackling the C211/A591 semester project based on the design recipe. We then tried to make sure the semester project and the approach in C212/A592 were consistent[9]. In the process he developed a program that he

[8] https://i.pinimg.com/originals/c0/72/0d/c0720da9a3ad64165c1b981d7bc7dc88.jpg

[9] Incidentally in C212/A592, which is in Java, students develop their own BigBang abstraction, first for individual games, then using the MVC pattern for multiplayer games.

called SuperPuffs because it showed Superman flying around to get pancakes[10] (in typical Snake/Worm fashion).

Figure 11.3: Daniel Brady's (Summer 2014 UI) Superpuffs game.

Figure 11.4: Sad Snake (Homework 06 in Summer 2017) ending.

[10]As Superman got to the pancakes his trail of little clouds would increase, and new pancakes would appear randomly in the scene. Daniel used to say the meaning of the trail of clouds was a bit ambiguous, but that it made sense why Superman would die if, as the trail would get longer, Superman would run into his own trail of (presumably noxious) clouds.

A snake, heading down, with a grid showing its individual segments.

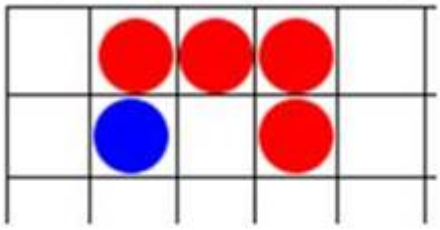

Figure 3: The previous snake, now heading left.

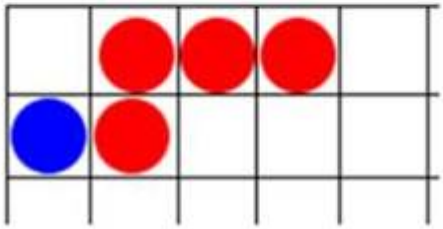

Figure 11.5: How the Snake moves: the head moves, following the direction it has, the tail vanishes and becomes neck (where the head used to be). With structural recursion we can take care of worlds that contain objects of arbitrary size, such as a hungry snake.

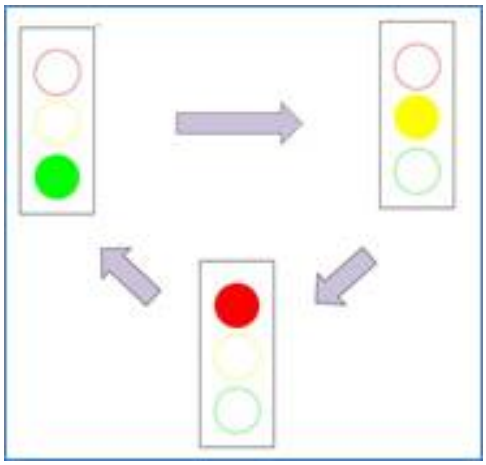

Figure 11.6: Traffic lights work differently here in the US and in England (not sure why). Here's an older exercise that initially deals with timers. Assuming this state transition diagram (US) can you write a program to either cycle between these states by timers and/or (as an extension) via key events? Either way the purpose here is for you to please compare the world data representations for Snake and this problem.

Chapter 12

BSL

In Computer Science we seek algorithmic solutions to common problems. We encode the solutions using a programming language. Our first language is called BSL. It uses a prefix notation and it has numbers, strings, characters and booleans in it, as well as a number of primitive functions (such as `random`, `substring`, `+`, `*`, `<`, `even?`, `zero?` and many others). With `define` we bound values to names. With `define-struct` we define structures. When we design functions (or operators) we use the design recipe. The design recipe starts with a choice for data representation. The heart of the design recipe is a function template. Templates are classified according to the data they're processing: atomic, intervals, itemizations (unions), structures, unions of structures and uni-dimensional lists (so far). (In this context `big-bang` and `animate` programs provide a framework for event-driven programming.) `cons` and `empty` along with `first` and `rest` appear in all list processing templates. For very large lists (such as the list of words in "Green Eggs and Ham", or even "Macbeth") we need BSL with list abbreviations. We've seen how to abstract over two functions with an additional parameter. An example will serve as reminder:

```
; insert-ascending : Number [ListOf Number] -> [ListOf Number]
; inserts number in its place in list of numbers in ascending order
(define (insert-ascending num lon)
  (cond ((empty? lon) (list num))
        ((<= (first lon) num)
         (cons (first lon)
               (insert-ascending num (rest lon))))
        (else (cons num lon))))

(check-expect (insert-ascending 2 empty) (list 2))
(check-expect (insert-ascending 3 (list 1 2 4 5)) (list 1 2 3 4 5))
(check-expect (insert-ascending 3 (list 1 2 3 3 5)) (list 1 2 3 3 3 5))
(check-expect (insert-ascending 1 (list 2 3 4 5)) (list 1 2 3 4 5))
(check-expect (insert-ascending 5 (list 2 3 4 5)) (list 2 3 4 5 5))
```

Here's the corresponding `sort-ascending` procedure:

```
; sort-ascending : [ListOf Number]
; sorts a possibly empty list of numbers in ascending order
(define (sort-ascending lon)
  (cond ((empty? lon) lon)
        (else
         (insert-ascending (first lon) (sort-ascending (rest lon))))))

(check-expect (sort-ascending empty) empty)
(check-expect (sort-ascending (list 5 4 6 3 5 1 2))
              (list 1 2 3 4 5 5 6))
```

If we want to sort in descending order:

```
; insert-descending : Number [ListOf Number] -> [ListOf Number]
; inserts number in its place in list of numbers in descending order
(define (insert-descending num lon)
  (cond ((empty? lon) (list num))
        ((>= (first lon) num)
         (cons (first lon)
               (insert-descending num (rest lon))))
        (else (cons num lon))))

(check-expect (insert-descending 2 empty) (list 2))
(check-expect (insert-descending 3 (list 5 4 2 1)) (list 5 4 3 2 1))
(check-expect (insert-descending 3 (list 5 3 3 2 1)) (list 5 3 3 3 2 1))
(check-expect (insert-descending 1 (list 5 4 3 2)) (list 5 4 3 2 1))
(check-expect (insert-descending 5 (list 5 4 3 2)) (list 5 5 4 3 2))

; sort-descending : [ListOf Number]
; sorts a possibly empty list of numbers in descending order
(define (sort-descending lon)
  (cond ((empty? lon) lon)
        (else
         (insert-descending (first lon) (sort-descending (rest lon))))))

(check-expect (sort-descending empty) empty)
(check-expect (sort-descending (list 5 4 6 3 5 1 2))
              (list 6 5 5 4 3 2 1))
```

It is hoped that this exercise becomes boring quickly, since there's so much code that is repeated. Every time we need to sort in a new way[1] we need to

[1]Examples: sort a list of numbers such that even numbers are listed first, in descending order, followed by the odd numbers in ascending order, and variations of this. Another example: sort a list of random numbers from the interval $[10, 30]$ such that all the numbers

write two new functions: this is not good. We need to find a way to use our initial two functions over and over again. So we abstract with one additional parameter, as advertised:

```
; insert : String Number [ListOf Number] -> [ListOf Number]
; inserts number into ordered list according to first param
(define (insert how num lon)
  (cond ((empty? lon) (list num))
        ((string=? how "ascending")
         (cond ((<= (first lon) num)
                (cons (first lon) (insert how num (rest lon))))
               (else (cons num lon))))
        ((string=? how "descending")
         (cond ((>= (first lon) num)
                (cons (first lon) (insert how num (rest lon))))
               (else (cons num lon))))
        (else (error "Not sure what you want."))))

(check-expect (insert "ascending" 2 empty) (list 2))
(check-expect (insert "ascending" 3 (list 1 2 4 5)) (list 1 2 3 4 5))
(check-expect (insert "ascending" 3 (list 1 2 3 3 5)) (list 1 2 3 3 3 5))
(check-expect (insert "ascending" 1 (list 2 3 4 5)) (list 1 2 3 4 5))
(check-expect (insert "ascending" 5 (list 2 3 4 5)) (list 2 3 4 5 5))

(check-expect (insert "descending" 2 empty) (list 2))
(check-expect (insert "descending" 3 (list 5 4 2 1)) (list 5 4 3 2 1))
(check-expect (insert "descending" 3 (list 5 3 3 2 1)) (list 5 3 3 3 2 1))
(check-expect (insert "descending" 1 (list 5 4 3 2)) (list 5 4 3 2 1))
(check-expect (insert "descending" 5 (list 5 4 3 2)) (list 5 5 4 3 2))

; sort-ascending : [ListOf Number]
; sorts a possibly empty list of numbers in ascending order
(define (sort how lon)
  (cond ((empty? lon) lon)
        (else (insert how (first lon) (sort how (rest lon))))))

(check-expect (sort "ascending" empty) empty)
(check-expect (sort "ascending" (list 5 4 6 3 5 1 2))
              (list 1 2 3 4 5 5 6))

(check-expect (sort "descending" empty) empty)
(check-expect (sort "descending" (list 5 4 6 3 5 1 2))
              (list 6 5 5 4 3 2 1))
```

---

less than or equal to 21 appear first, in descending order, followed by the numbers above 21, in ascending order (this is the Blackjack order).

It doesn't look too pretty but at least we only have two functions. It's true, they have an additional parameter that helps them dispatch the right code according to the intended purpose. So we have definitely learned to abstract over two (or more) functions using an additional parameter[2].

When that parameter refers to a function we need to graduate to the next language in hierarchy: ISL (Intermediate Student Language). It allows functions to be values. It also provides a set of powerful primitives that abstract over the types of functions we can now write in BSL (`sort`, `filter`, `map`, `foldr`, `apply`, `build-list`, etc.) We call these higher order functions. We've also heard in lab about `lambda` and we continue to practice test-driven development with `check-expect` and such.

An example will make this clear:

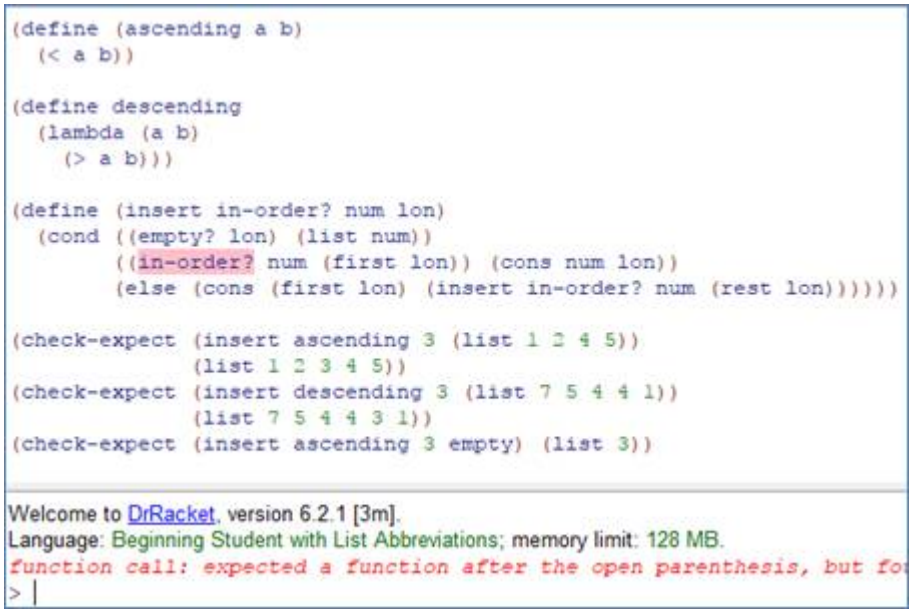

Figure 12.1: This code would work as intended in ISL.

BSL does not allow functions to be passed around to other functions. How meaningful would that be? How natural? In mathematics this has been common place since about 1687 when Newton published his *Principia*. The textbook actually says: "If you have taken a calculus course, you have encountered the differential operator and the indefinite integral. Both of those are functions that consume and produce functions. But we do not assume that you have taken a calculus course."

We shall see shortly what is new in ISL (and its variation ISL/+) and what the additional expressive power of the new language can do for us and our programs.

[2]See also 14.1 Similarities in Functions (click to go there)

Chapter 13

ISL

Around the 1670s, Sir Isaac Newton's conceptual understanding of physics prompted him to invent the complicated math known as calculus[1].

In ancient times (around 580-212 BCE), three notable philosophers–Pythagoras, Euclid, and Archimedes–created what we know now as algebra and geometry, but they struggled to unify them. In the late 1500s, Rene Descartes unified algebra, which was used as an analytical tool, along with the geometric shapes. He found that a point on a plane can be described using two numbers, and from that information, equations of geometric figures were born.

Newton was foremost a physicist, and in his day, he tackled many difficult issues in physics, the most famous of which was gravity. Newton is known for developing the laws of motion and gravitation, which undoubtedly led to his work in calculus. When trying to describe how an object falls, Newton found that the speed of the object increased every split second and that no mathematics currently used could describe the object at any moment in time.

Another tale of Newton's reasoning is that a colleague of his asked why the orbits of the planets were in an ellipse[2]. Newton took some time to ponder the question and came back with the fact that the ellipses are actually sections of cones. His proofs[3] were entirely geometric and based on obscure[4] properties of conic sections, which in his times everybody knew. Armed with calculus, which he invented, he could describe exactly how those sections behaved.

Calculus provided a more expressive mathematical language for Physics in its quest to more effectively describe Nature. It is also what ISL and ISL/+ will try to do for us as we ponder over the set of all programs, with the stated goal of increasing the set of intellectually manageable programs[5]

[1] `https://futurism.com/how-and-why-did-newton-develop-such-a-complicated-math/`

[2] Which today is known as one of Kepler's laws.

[3] You can find an absolutely breathtaking account of that in *Feynman's Lost Lecture* a 1996 book (plus CD) that essentially contains Richard Feynman's lecture of 03/13/1962 at Caltech. The book was put together by David Goodstein.

[4] For us, today.

[5] Cf. Dijkstra.

In the rest of the chapter we will introduce and illustrate through various examples ISL's new features. The chapter that follows deals with ISL/+.

Let us now proceed to examine the new features of ISL:

13.1 `apply`

The following tests all pass:

```
(check-expect (+ 1 2 3 4) 10)
(check-expect (+ 5 3) 8)

(check-expect (apply + (list 1 2 3 4)) 10)
(check-expect (apply + (list 5 -1 2)) 6)


(check-expect (max -3 4) 4)
(check-expect (max 6 21 -12 23 -14) 23)

(check-expect (apply max (list 4 2 3 6 5 1))
              (max 4 2 3 6 5 1))
```

The absolutely novel situation here is that `max` and `+` are functions that admit a variable number of arguments. We don't know how to define functions that have a variable number of arguments[6], but we can collect the arguments in a list. Furthermore if while the program is executing these arguments (collected in a list) do become available we couldn't possibly write code that would result in the invocation of either one of these functions (`max`, `+`) with the arguments we collected: we'd need to write code at execution time, and execute it[7].

So `apply` allows us to take a function of variable arity and apply it to a list of values we collected. This is a fundamentally new feature, one that you can't define in BSL.

Exercise: try writing the signature of `apply`.

13.2 `build-list`

```
(check-expect (build-list 10 add1)
              (list 1 2 3 4 5 6 7 8 9 10))
```

This works a bit like `animate`: the numbers from 0 to 10 are given to `add1` and the results are being collected in a list. This gets even better in ISL/+.

[6]The point I am trying to make here is: if (as your program is running) you get a list of numbers of unknown length and you are being told to find the largest of them you can't possibly use `max` without `apply`. You could write your own `max` function, using structural recursion, but you could not possibly use the `max` that we all know and love (above).

[7]Another thing we have no idea, nor any significant interest at the moment, in doing.

13.3 `compose`

In ISL/+ we could define a binary version of this function as follows:

```
(define (compose f g)
  (lambda (x)
    (f (g x))))
```

But `compose` has variable arity, so no need to try to compete with it.

```
(check-expect ((compose sub1 sqr abs) -2)
              (- (* (abs -2) (abs -2)) 1))
```

The common denominator has been, thus far, that functions can now be passed as arguments. However in addition to this new aspect some of these new features truly broke away into a new dimension. This will not always be the case; for example most of the remaining features simply abstract the design patterns that we already know. In this regard when we start using these new features it will feel similar to what riding a bike, driving a car or, at times, flying an airplane, feels to walking[8].

13.4 `filter`

The next function implements a simple selection[9] mechanism:

```
(check-expect (filter even? (list 3 2 4 5 1 2 6 6))
                            (list   2 4     2 6 6))
```

The signature for this function is as follows:

```
[X -> Boolean] [List-of X] -> [List-of X]
```

Let's read it out loud to make sure all is clear: `filter` takes a predicate on `X` (which is some type) and a list of `X` values and returns a list that contains only those values (if any) for which the predicate holds[10].

This function would be easy to define using structural recursion.

---

[8] You need to learn how to ride a bike, but you can go farther and faster with it than by walking. It's harder to learn how to drive, and the risks of accidents are bigger, but I would not want to have to go by bike, let alone to walk, to Indy. I will never learn to fly an airplane, but I prefer it if I go to Los Angeles, Australia or Europe; however, on campus I absolutely can't use an airplane and depending where on campus I am and where I want to go I might prefer the bike to the car (or the other way around).

[9] See Relational Model later.

[10] This description includes the purpose statement.

13.5 `fold*`

There are two such functions: `foldl` and `foldr`.

```
(check-expect (foldr + 0 (list 1 2 3 4 5 6)) 21)
(check-expect (foldr cons empty (list 1 2 3 4 5 6))
                                (list 1 2 3 4 5 6))
(check-expect (foldl cons empty (list 1 2 3 4 5 6))
                                (list 6 5 4 3 2 1))
(check-expect (foldl - 0 (list 1 2 3 4 5 6))  3)
(check-expect (foldr - 0 (list 1 2 3 4 5 6)) -3)
```

Both have the same signature:

```
[ [ X Y -> Z ] X [ListOf Y] -> [ListOf Z] ]
```

The signature says that there are three inputs:

1. a function `[ X Y -> Z ]`
2. a(n initial) value of type `X`
3. a list of items of type `Z`

The first function takes two values (of types `X` and `Y` that don't necessarily need to be different, but can be) and produce a value of type `Z`. Again, these could all be the same type, but don't need to be. The idea is that the function will be repeatedly applied to the elements of the list using the individual value as a seed. The previous tests could be rewritten as follows:

```
(check-expect
  (foldr + 0 (list 1 2 3 4 5 6))
  (+ 1 (+ 2 (+ 3 (+ 4 (+ 5 (+ 6 0)))))))

(check-expect
  (foldr cons empty (list 1 2 3 4 5 6))
  (cons 1 (cons 2 (cons 3 (cons 4 (cons 5 (cons 6 empty)))))))

(check-expect
  (foldl cons empty (list 1 2 3 4 5 6))
  (cons 6 (cons 5 (cons 4 (cons 3 (cons 2 (cons 1 empty)))))))

(check-expect (foldl - 0 (list 1 2 3 4 5 6))
              (- 6 (- 5 (- 4 (- 3 (- 2 (- 1 0)))))))

(check-expect (foldr - 0 (list 1 2 3 4 5 6))
              (- 1 (- 2 (- 3 (- 4 (- 5 (- 6 0)))))))
```

So if you understand the signature it's just the order that differs.

13.6 map

Continue to practice writing signatures:

```
(check-expect (map sub1 (list 4 2  0 6 1 -2))
              (list           3 1 -1 5 0 -3))
```

Here's why it won't be easy to write `map`'s signature:

```
(check-expect (map list '(1 2 3 4)
                        '(a b c d))
              '((1 a) (2 b) (3 c) (4 d)))
```

In any event a `map` function of given arity is easy to define with structural recursion. As an exercise, design a function named `cross` that takes two lists and produces their cartesian product. As a reminder, for any two[11] sets A and B their cartesian product is defined as follows: $A \times B = \{(x, y) | x \in A, y \in B\}$.

13.7 sort

The signature of this function is:

```
[ListOf X] [X X -> Boolean] -> [ListOf X]
; the resulting list is sorted accordingly
```

To clarify the outcome consider the following:

```
(sort (list 1 3 5 2 3 6) <)

; sorted? : [ListOf X] [X X -> Boolean] -> Boolean
; determines if the list is sorted according to pred
(define (sorted? lon pred)
  (cond ((empty? lon) true)
        ((empty? (rest lon)) true)
        ((pred (first lon) (second lon)) (sorted? (rest lon) pred))
        (else false)))

(check-expect
 (let ((lon (list 4 3 2 3 1 4 6 5)))
   (sorted? (sort lon <) <=))
 true)
```

With what we know now we can also write:

```
(check-expect (apply <= (sort (list 4 2 3 1 5 5 2 4) <)) true)

(check-expect (apply >= (sort (list 4 2 3 1 5 5 2 4) >)) true)
```

[11] This can be easily generalized to: $\times_{i=1}^{n} A_i = \{(x_1, \ldots, x_n) | x_i \in A_i, \forall 1 \leq i \leq n\}$

13.8 `procedure?`

This, in some sense, is the very hallmark of this (ISL) new language: we are now able to pass functions around as if they were data. We can examine them and if needed we can invoke them. Clearly this predicate is necessary, it would have made no sense in BSL though.

13.9 `local`

Consider the following implementation of `cross` in ISL:

```
; A [PairOf X Y] is a:
;   -- (list X Y)
; (or we could have defined a struct for this)
;
; cross : [ListOf X] [ListOf Y] -> [ListOf [PairOf X Y]]
; the entire work is done by helper, cross just sets the ranges
(define (cross initial-a initial-b)
;          helper : [ListOf X] [ListOf Y] -> [ListOf [PairOf X Y]]
;          helper simulates a pair of nested loops
  (local ((define (helper x y)
            (cond ((empty? x) empty)
                  ; when the outer loop ends
                  ((empty? y) (helper (rest x) initial-b))
                  ; when the outer loop advances
                  (else (cons
                         (list (first x) (first y))
                         (helper x (rest y)))))))
                  ; when the inner loop advances
    (helper initial-a initial-b)))

(check-expect (cross '(a b) '(1 2 3))
              '((a 1) (a 2) (a 3) (b 1) (b 2) (b 3)))
```

Can you write it without `local`? Would it be easier to write using a `map`?

Chapter 14

ISL+

Higher-order functions available in ISL+ shortcut through our design recipe and templates: `(sum lon)` is the same as `(apply + lon)`, and `sorted?` for which we worked a bit (in a very obscure, arcane context) can be simply defined as `(apply < lon)` while `(append a b)` is the same as `(foldr cons b a)`, apparently. ISL/+ primitives abstract (encapsulate in a general fashion) the most important, powerful features of our template(s) thus far. In addition we can use anonymous functions and closures. Here is ISL's `sort` that only needs a predicate to do its job:

```
(sort
 (build-list 10 (lambda (x) (random 40)))
 (lambda(a b)
   (cond ((and (<= a 21) (<= b 21)) (>= a b))
         ((and (> a 21) (> b 21)) (<= a b))
         ((and (<= a 21) (> b 21)) true)
         (else false))))
```

Here's how short (and compact) the shortest big-bang becomes:

```
(big-bang 16
          (on-tick (lambda (world) world))
          (on-key (lambda (world key)
                    (cond ((key=? key "up") (+ 1 world))
                          (else world))))
          (to-draw (lambda (world)
                     (place-image (text (number->string world)
                                        30
                                        "olive")
                                  100
                                  100
                                  (empty-scene 200 200)))))
```

I am sure you miss the annotations so please add them.

14.1 Closures

Identify the one closure[1] in the example below[2]:

```
; A Bit is one of:
;   -- 0
;   -- 1

; lining : Number Number -> [ListOf Bit]
; produces a list of size bits all zero except a one at index target
(define (lining target size)
  ; build-list : Number [Number -> Bit] -> [ListOf Bit]
  ; turns list of size nums in2 list of bits cf. given conversion procedure
  (build-list size
              ; [Number -> Bit]
              ; converts an index (a position) into a bit
              ; identifies location of "on" bit (like a predicate)
              (lambda (index) (if (= index target) 1 0))))

; eight : Number -> [ListOf [ListOf Bit]]
; (eight size) produces the identity matrix of size size
(define (eight size)
  ; [[Number -> [ListOf Bit]] [ListOf Number] -> [ListOf [ListOf Bit]]]
  ; maps numbers into their corresponding index rows in the identity matrix
  (map
   ; [Number -> [ListOf Bit]]
   ; turns row into the the row-th row in the identity matrix of size size
   (lambda (row)
     (lining row size))
   ; build-list : Number [Number -> Number] -> [ListOf Number]
   ; produces the list of integers from 0 to (sub1 number)
   (build-list size
               ; [Number -> Number]
               ; this is the identity function
               (lambda(x) x))))
```

Here's a test:

```
(check-expect (eight 3)
              (list
               (list 1 0 0)
               (list 0 1 0)
               (list 0 0 1)))
```

We notice that the code is true to promise.

[1] https://www.cs.indiana.edu/classes/c211-dgerman/EdwardsColumbiaLambda.pdf

[2] Worked out in class with Lining Jiang on 05/26/2016.

Here's another test, formatting is ours:

```
(check-expect (eight 10)
              (list
               (list 1 0 0 0 0 0 0 0 0 0)
               (list 0 1 0 0 0 0 0 0 0 0)
               (list 0 0 1 0 0 0 0 0 0 0)
               (list 0 0 0 1 0 0 0 0 0 0)
               (list 0 0 0 0 1 0 0 0 0 0)
               (list 0 0 0 0 0 1 0 0 0 0)
               (list 0 0 0 0 0 0 1 0 0 0)
               (list 0 0 0 0 0 0 0 1 0 0)
               (list 0 0 0 0 0 0 0 0 1 0)
               (list 0 0 0 0 0 0 0 0 0 1)))
```

And here's the same function without annotations and using only one `define`:

```
(define (eight size)
  (map (lambda (row)
         (build-list size (lambda (index) (if (= index row) 1 0))))
       (build-list size (lambda(x) x))))
```

Hard to say which version is more legible[3]. Please take a look here[4] to see a derivation of `cross` without any ISL/+ special features except closures[5]. Here's `cross` again this time using closures and `map`'s:

```
;  cross : [ListOf [ListOf Atom]] -> [ListOf [ListOf Atom]]
;  produces the cartesian product of its inputs
; (define (cross lol)
;   (cond ((empty? lol) ...)
;         ((list? lol) (... (first lol) ... (cross (rest lol)) ...))
;                                           ^ what's the promise here?
;         (else "Bad input.")))
(define (cross lol)
  (cond ((empty? lol) lol)
        ((empty? (rest lol)) (map list (first lol)))
                             ; one extra wrinkle when dealing with the fixed point
        ((list? lol) (apply append ; a-ha! did you see that?
                            (map (lambda (promise-elem)
                                   (map (lambda (first-elem)
                                          (cons first-elem promise-elem))
                                        (first lol)))
                                 (cross (rest lol))))) ; see how I build on the promise?
                      (else "Bad input")))
```

[3] Probably together, as a pair, they work best.

[4] `https://www.cs.indiana.edu/classes/c211-dgerman/sum2016/forloops.html`

[5] Closures (the `lambda`'s) are the real strength here.

Here are the tests:

```
(check-expect (cross empty) empty)

(check-expect (cross '((1 2 3)))
              (list (list 1) (list 2) (list 3)) )

(check-expect (cross '((1 2 3)  (a b)))
              (list (list 1 'a) (list 2 'a) (list 3 'a) (list 1 'b)
(list 2 'b) (list 3 'b)))

(check-expect
 (cross '((1 2 3) (a b) (+ -)))
 (list
  (list 1 'a '+)
  (list 2 'a '+)
  (list 3 'a '+)
  (list 1 'b '+)
  (list 2 'b '+)
  (list 3 'b '+)
  (list 1 'a '-)
  (list 2 'a '-)
  (list 3 'a '-)
  (list 1 'b '-)
  (list 2 'b '-)
  (list 3 'b '-)) )
```

Now, please accept the following challenge[6] (also see below): design a function that is able to draw a scalable pattern like the one shown in the picture. The answer is here[7] if you want to check.

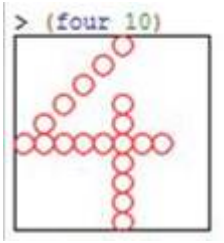

Figure 14.1: A scalable pattern.

[6] http://silo.cs.indiana.edu:8346/c211/impatient/images/image039.jpg
[7] http://silo.cs.indiana.edu:8346/c211/sum2016/for-001.phps

Chapter 15

Abstraction

A few years ago John F. Duncan[1] and I started an experiment in our classes for our monthly SOTL group meetings[2]. Students would have to come to class with a minute paper (this was to be their ticket to the lecture) that they would turn in before class started. We used to refer to this exercise as "Previously on C211" and we compared it with those "three seasons of Lost in eight minutes" summaries on DVD. Here's such a summary from a student[3] in 6W1 Summer 2015 for where we are in this book, at this time:

> Understanding the basic structure of functions, enforced by the ever increasing number of template definitions, the numbers of tools at our disposal has increased dramatically. The use of the unnamed function, `lambda`, and the recursive structure has allowed us to work through arbitrary sized data. Furthermore, our ability to construct higher [order] functions has led to a new area of coding: abstraction. I can see that the reuse of code not only cuts down on the size of the file, increasing [maintainability], but also higher order functions increase the efficiency with which data is processed, also increasing speed. We're approaching a zone of high eloquence and paradigmicity with our design[4].

There's an exercise in the book (in Part III Abstraction) that asks us to design a data representation for finite and infinite sets where the set itself can determine if a certain element is in it or not. (Exercise changes its number often but can be easily found through a quick scan of 17.5 Representing with `lambda`).

In this chapter I want to discuss an extension of a solution to that problem, so to narrow down the number of possibilities let's get started by quickly reviewing a reasonably good answer. In the end we will develop an entirely new

[1] `https://www.sice.indiana.edu/all-people/profile.html?profile_id=193`

[2] The experiment is still under way.

[3] I truly don't remember the name of the student, or at least I'm not sure, so I am going to say it was either Tori Milhoan or William Bobe.

[4] Such a refreshing summary!

solution and approach that will be positioned between this class (C211) and the next (C212) and will open the path up to Object-Oriented Programming.

We start by realizing someone already defined the set of all evens for us:

```
(define all-evens even?)
; a set is an effective procedure for determining membership
```

We then define the empty set and the ability to add an element to a set:

```
; [ empty-set: X -> Boolean ]
; the empty set has no elements in it
(define empty-set
  (lambda (elem) false))

; add : X [ X -> Boolean ] -> [ X -> Boolean ]
; new element is added to the set
(define (add elem set)
  ; X -> Boolean
  ; given request we determine if it's in the set or not
  (lambda (request)
    (if (equal? request elem)
        true
        (set request))))
```

This immediately gives us two set operations:

```
; union : [X -> Boolean] [X -> Boolean] -> [X -> Boolean]
; takes two decision procedures and combines them into one
(define (union a b)
  ; X -> Boolean
  ; given request we determine if it's in the set or not
  (lambda (request)
    (or (a request) (b request))))

; subtract : [X -> Boolean] [X -> Boolean] -> [X -> Boolean]
; takes two decision procedures and combines them into one
(define (subtract a b) ; can be defined but not part of problem as stated
  ; X -> Boolean
  ; decision procedure: an element is in a - b if it's in a and not in b
  (lambda (request)
    (and (a request) (not (b request)))))
```

It should be easy for you to implement `intersection` now. We already have `cross` although not in this representation[5] and `filter` (in the most general

[5]It can't even be written in this representation, which is most suitable for environments, however the solution that we will discuss will admit a `cross`. Don't want to be distracted

sense). This means we can easily define a relational algebra (selection, projection and join) so look forward to that when the time comes, in a few chapters.

To better understand what we say here we have some slides[6] for you[7].

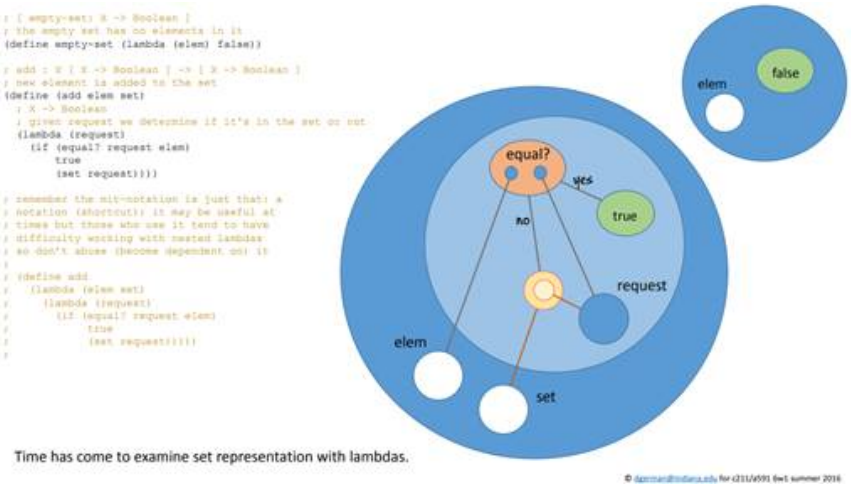

Figure 15.1: The slides online explain in slow motion how things work.

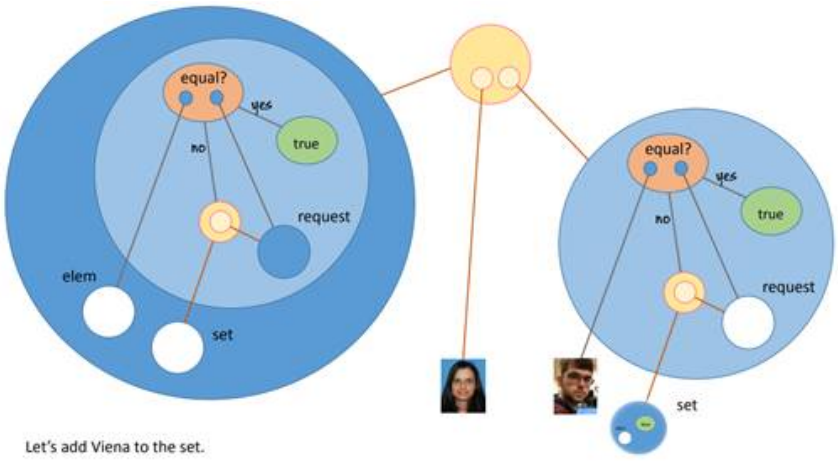

Figure 15.2: Some of the examples are illustrated step by step.

(side-tracked) into a huge deliberation here, so I will say we only consider finite sets. For infinite sets we should be able to work a solution out with infinite streams and lazy evaluation, along the lines of infinite continuing fractions, but we should be clear we're thinking finite sets, as in actual databases.

[6] `http://silo.cs.indiana.edu:8346/c211/impatient/lambdas.pdf`

[7] Some of the students in the slides: Viena Sharma, Griffin Park and Swaraj Patankar.

Based on what you have seen thus far we define a new data representation:

```
; A [SetOf X] is a [ [SearchRequestFor X] -> [SearchResultFor X] ]
; A set is a function that receives a search request and returns a search result.
```

This definition, in turn, requires us to define two new data representations:

```
; A [SearchRequestFor X] is one of:
;   -- (make-select-all)
; this is the request to see everything
(define-struct select-all ())
;   -- (make-see-top)
; this request is for the element most recently added to the set (like a stack)
(define-struct see-top ())
;   -- (make-retrieve-kernel)
; this request is for the set minus the top element (if any)
(define-struct retrieve-kernel ())
;   -- (make-singleton-request X)
; this requests is for a specific element (whether it is in the set or not)
(define-struct singleton (element))
```

Notice how the second data representation choice does not require any new data definition:

```
; A [SearchResultFor X] is one of (maps types of requests above in some order):
;   -- Boolean
; if we're searching for a specific element we may or may not find it
;  -- [Maybe X]
; if we're being asked to retrieve the top element (most recently added to the set)
;   -- [ [SearchRequestFor X] -> [SearchResultFor X] ]
; if we're being asked to return the set minus its top element
;   -- [ListOf X]
; the other kind of request we implement just returns all elements in the set
```

And we can finally make our first real definition:

```
; empty-set : [SearchRequestFor X] -> [SearchResultFor X]
; the empty set does not contain any elements
(define empty-set
  (letrec ((this (lambda (request)
                   (cond ((see-top? request) false)
                         ((select-all? request) empty)
                         ((retrieve-kernel? request) this)
                         ; minor hack so we don't write code twice
                         (else false)))))
    this)) ; or I could have just defined empty-set recursively at the top level
```

Note that things are a bit more complicated on account that we now have an enumeration of messages (or commands) the sets can receive (and process).

Here are, now, `add`, `union` and `subtract`:

```
; add : X [SetOf X] -> [SetOf X]
; adds the new element to the set
(define (add elem set)
  (cond ((set (make-singleton elem)) set) ; so as to not duplicate
        (else
         ; [ [SearchRequestFor X] -> [SearchResultFor X] ]
         (lambda (request)
           (cond ((see-top? request) elem) ; similar to first for lists
                 ((retrieve-kernel? request) set) ; similar to rest for lists
                 ((and (singleton? request) ; we found it
                       (equal? elem (singleton-element request))) true)
                 ((select-all? request) (cons elem (set request)))
                                        ; keep building answer
                 (else (set request))))))) ; keep looking for element

; subtract : X [SetOf X] -> [SetOf X]
; returns a set without the specified element in it
(define (subtract elem set)
  (if (set (make-singleton elem)) ; if it's in there we have to take it out
      (let ((top (set (make-see-top))) ; let's get our hands on the top element
            (bottom (set (make-retrieve-kernel)))) ; and the set it was added to
        (if (equal? elem top) ; if top is what we want to remove
            bottom ; we have the answer
            (add top (subtract elem bottom)))) ; otherwise we have to keep working
      set)) ; if the element is not in the set we're done

; union : [SetOf X] [SetOf X] -> [SetOf X]
; returns a set that has all elements in it (no duplicates)
(define (union a b)
  (cond ((empty? (a (make-select-all))) b) ; a is the empty-set
        (else (let ((e (a (make-see-top)))) ; let me see what a has on top
                (union (subtract e a)
                       ; similar to (+ n m) = (+ (- n 1) (+ m 1)) until (zero? n)
                       (add e b))))))

; one example with a set named search-for

(define search-for (add 3 (add 1 (add 3 (add -1 (add 5 (add -1 empty-set)))))))
                   ; the set 1, 3, 5, -1

(check-expect (search-for (make-singleton 3)) true)
(check-expect (search-for (make-select-all)) (list 1 3 5 -1))
                                             ; notice the order of elements
```

Procedural data representation is sometimes used with environments (also known as associative arrays, hashtables, etc.) and for that their main goal is to act as oracles: is such and such variable defined? The approach we took recreates the set with every request[8] in a true object-oriented fashion, and the sets become aware, enlightened[9].

```
; two more examples with sets a and b
(define a (add 1 (add 2 (add 1 (add 3 (add 5 (add 3 (add 2 empty-set)))))))
(check-expect (a (make-select-all)) (list 1 5 3 2))
              ; again, order is a result of not accepting duplicates

(define b (add 1 (add 2 (add 7 (add 7 (add 6 (add 6 (add 1 empty-set)))))))
(check-expect (b (make-select-all)) (list 2 7 6 1))
(check-expect ((subtract 2 a) (make-select-all)) (list 1 5 3))
              ; test of subtracting innermost element of a

(define c (union a b)) ; test of union below
(check-expect (c (make-select-all)) (list 3 5 2 7 6 1))
                                    ; order easy to anticipate
(check-expect (a (make-select-all)) (list 1 5 3 2))
               ; silly test to prove a is unchanged
(check-expect (b (make-select-all)) (list 2 7 6 1))
               ; silly test to prove b is unchanged

(define danny empty-set) ; new tests, we define a mini-roster for lh008
(check-expect (danny (make-singleton "Menghan")) false)

(define menghan (add "Menghan"  danny))    ; so we build the mini-roster
(define brandon (add "Brandon"  menghan))  ; by adding students
(define hehe    (add "He He"    brandon))  ; one by one
(define yuanyuan(add "Yuanyuan" hehe))
(define chuhan  (add "Chuhan"   yuanyuan))
(define arif    (add "Arif"     chuhan))
(define baojie  (add "Baojie"   arif))
(define norman  (add "Norman"   baojie))
(define mark    norman)

(check-expect
 (mark (make-select-all))
 (list "Norman" "Baojie" "Arif" "Chuhan" "Yuanyuan" "He He" "Brandon" "Menghan"))

(check-expect
 (chuhan (make-select-all)) (list "Chuhan" "Yuanyuan" "He He" "Brandon" "Menghan"))
```

[8] So at least conceptually one can implement a `set!` feature easily in BSL in a purely functional manner (i.e., not relying on any pre-existing side-effects) we described.

[9] A Zen master once told me: "Do the opposite of what I tell you." So, I didn't.

15.1 Intermission

"Second, we want to provide you with a little distraction. We know how frustrating the subject matter can be, and a little distraction will help you keep your sanity."
— Friedman & Felleisen, *The Little Lisper*

Where do otters come from?	Otter Space.
Why is an elephant big, gray and wrinkled?	Because if he was small, white and round he'd be an aspirin.
Can an elephant jump higher than a lamppost?	Yes. Lampposts can't jump.
What goes ha, ha, ha, plop?	Someone laughing his head off.
The ancient Egyptian children were rather confused when their daddies became mummies.	The elevator didn't feel well. It felt it was coming down with something.
What did the bee say to the flower?	"Hey, Bud, when do you open?"
Excuse me, does this bus go to Duluth?	No, this bus goes beep beep.
How do you catch up a unique rabbit?	Unique up on it.
How do you catch up a tame rabbit?	Tame way, unique up on it.
How do crazy people go catch a rabbit?	You wait until it comes down the psycho path and unique up on it.
Somebody said you look like an owl.	Who?

What's green, leafy and sings the blues?	Elvis Parsley.
What do you call a boomerang that doesn't work?	A stick.
Why do the elephants paint their toenails red?	So they can hide in cherry trees.
Have you ever seen an elephant in a cherry tree?	No, because they're hiding so well!
Why shouldn't you go in the jungle on Thursdays between 1-3pm?	Because it's when the elephants are taking skydiving lessons.
Why are aligators/crocodiles so flat?	They didn't stay out of the jungle on Thursdays between 1-3pm.
Why do fire departments have Dalmatians?	To help them find the hydrants.
What do Alexander the Great and Winnie the Pooh have in common?	They have the same middle name.
How do you scramble an egg?	Ne gag.
What does a dog do that a man steps into?	Pants.

```
{ let f = function(n, f) { return n == 1 ? 1 : n + f(n-1, f); };
  { let f = function(n, f) { return n == 1 ? 1 : n * f(n-1, f); };
    console.log( f(10, f) );
  }
  console.log( f(10, f) );
}
```

Chapter 16

Grammars

Every data representation is a grammar. Every example is a derivation in the grammar. The structural recursion template describes basic acceptors and/or translators over the language defined by chosen data representation. It doesn't matter the data is now more complex: structural recursion template means our programs continue to emulate their inputs. Case in point, this example[1]:

```
; A [ListOf X] is one of:
;   -- empty
;   -- (cons X [ListOf X])

; A SEx[2] is one of:
;  -- Atom
;  -- [ListOf SEx]

; An Atom is one of:
;  -- Number
;  -- String
;  -- Symbol
; example: 3 is a Number and therefore an Atom
; example: "whassup" is a String and therefore an Atom
; example: in (define a 3) a is Symbol and therefore an Atom
; example: '(kyle viena skylar) is a list of Symbols and therefore a SEx
```

Now the acceptors we want to write:

```
; sex? : ... -> Boolean
; determines if the input qualifies as a SEx
(define (sex? input)
  (or (atom? input)
      (losex? input))) ; changed it
```

[1] See 19. The Poetry of S-Expressions in your text.
[2] Cf. Laurent Siklóssy

One down, two more to go:

```
; atom? : ... -> Boolean
; determines if the input qualifies as an atom
(define (atom? input)
  (or (symbol? input)
      (string? input)
      (number? input)))

; losex? : ... -> Boolean
; determines if the input qualifies as a [ListOf SEx]
(define (losex? input)
  (cond ((empty? input) true)
        (else ( (and (sex? (first input))
                     (losex? (rest input)))))))
```

Now the tests[3]:

```
; (define sex1 (cons 4
;                (cons "dog"
;                  (cons 'cat
;                    (cons (cons 5
;                            (cons "cat"
;                              (cons 'dog empty)))
;                          empty)))))
; the example is good but we need to take small steps

(check-expect (sex? (list 1 2 3)) true)

(check-expect (sex? '(1 (2 3) ((4) 5))) true)

(check-expect (sex? '(((((((()))) ((()))) ((()))) 4)) true)

(check-expect
 (sex? '(cons 4
              (cons "dog"
                    (cons 'cat
                          (cons
                           (cons 5
                                 (cons "cat"
                                       (cons 'dog empty)))
                           empty)))))
 true)
```

Can you give an example of input that does not qualify as an SEx(pression)?

[3]Worked in class with Kyle Palmer on 05/30/2016.

Now in the remaining of the chapter I'd like to do three things:

(a) I'd like to develop a simple lexical analyzer (tokenizer)

(b) I'd like to develop a simple top-down (recursive descent) parser

(c) I'd like to develop a simple expression evaluator (based on (a))

Let's start by building a tokenizer for simple arithmetic expressions:

```
; A Token is a String without space(s)

; tokenize : [ListOf Char] -> [ListOf Token]
; basically works like take/bundle
(define (tokenize input)
  (let ((input (trim input)))
    (if (empty? input)
        empty
        (cons (leading input)
              (tokenize (remaining input))))))

(check-expect
 (tokenize (explode "   good   to    see    you     "))
 '(("g" "o" "o" "d") ("t" "o") ("s" "e" "e") ("y" "o" "u")))
```

So that's all of it, and we need to provide: `trim`, `leading` and `remaining`.

```
; trim : [ListOf Char] -> [ListOf Char]
; if there are leading spaces they're removed
(define (trim loc)
  (cond ((empty? loc) empty)
        ((string=? " " (first loc)) (trim (rest loc)))
        (else loc)))

(check-expect
 (trim (explode "    how are you?   "))
          (explode "how are you?   "))

; leading : [ListOf Char] -> [ListOf Char]
; collects the list of leading non-space chars
(define (leading loc)
  (cond ((empty? loc) empty)
        ((string=? " " (first loc)) empty)
        (else (cons (first loc) (leading (rest loc))))))

(check-expect (leading (explode "this is it"))
              (explode "this"))
```

You've seen two of the functions, here's the third:

```
; remaining : [ListOf Char] -> [ListOf Char]
; collects what's left after leading is applied
(define (remaining loc)
  (cond ((empty? loc) empty)
        ((string=? " " (first loc)) loc)
        (else (remaining (rest loc)))))

(check-expect
 (remaining (explode "Fischers Fritze fischt frische Fische."))
 (explode  " Fritze fischt frische Fische."))
```

So now the top level function just builds an interface (wrapper) around the tokenizer so we get back strings as tokens instead of lists:

```
; string-split: String -> [ListOf Token]

(define (string-split str)
  (map
   (lambda (loc) (foldr string-append "" loc))
   (tokenize (explode str))))

(check-expect
 (string-split "    what  is  going   on    ")
 (list "what" "is" "going" "on"))

(check-expect
 (string-split " ( 1 + 2 ) * 3 ")
 (list "(" "1" "+" "2" ")" "*" "3"))
```

That concludes (a) and we will use this soon, at (c).

Starting the second part of our project, the parser, I need to clarify that the approach taken here is on purpose (relatively) very simple. The theory of parsing, translating and compiling is as old as the study of programming languages and therefore very well put together[4].

To write a successful recursive descent parser we need to make sure our grammar is not ambiguous. A grammar is ambiguous if it is both left- and right-recursive[5]. Furthermore, parsers usually guide their actions by paying attention to a set of so-called look-ahead symbols. Things can get quickly out of hand, so to keep everything simple we: will give our example, we'll keep it as simple and intuitive as possible, and then refer you to your friendly compilers course. This example is very similar to what's in 19.3 S-expressions in your

[4]See the original two volume set published by Prentice Hall in 1972 by Aho and Ullman.

[5]To get things going seriously we actually need a lot of definitions, and we don't have the time, space or, frankly, any mandate to do that now.

textbook. We get started by reiterating that our data representation will be a grammar, with productions. Here it is:

```
; Expression   -> Number
;              |  Calculation
; Calculation  -> Term * Term
;              |  Term + Term
; Term         -> ( Calculation )
;              -> Expr

(define e01        3                ) ; yes
(define e02   '(   2 *   4 )        ) ; yes
(define e03-a '( ( 1 +   2 )      ) ) ; no
(define e03-b '(   1 +   2 )        ) ; yes
(define e04-a '( ( 1 +   2 ) * 3   ) ) ; yes
(define e04-b '(   1 +   2   * 3   ) ) ; no
(define e04-c '(   1 + ( 2   * 3 ) ) ) ; yes

(check-expect (expr? e01  ) true)
(check-expect (expr? e02  ) true)
(check-expect (expr? e03-a) false)
(check-expect (expr? e03-b) true)
(check-expect (expr? e04-a) true)
(check-expect (expr? e04-b) false)
(check-expect (expr? e04-c) true)
```

Note that not all the examples are in the language generated by our grammar.

```
; expr? : ... -> Boolean
; determines if input is a well formed expression
(define (expr? input)
  (or (number? input)
      (calculation? input)))

(define (calculation? input)
  (and (list? input)
       (= (length input) 3)
       (or (and (term? (first input))
                (symbol=? (second input) '+)
                (term? (third input)))
           (and (term? (first input))
                (symbol=? (second input) '*)
                (term? (third input))))))

(define (term? input)
  (or (calculation? input)
      (number? input)))  ; that's it
```

Though I took out all annotations you can tell this is working the same way: every non-terminal is a procedure (function) and the entire program is a collection of mutually recursive procedures that closely emulates the grammar (hence, structural recursion, even when data is so tightly intertwined).

For my third example I'd like to use the `string-split` that I have developed a little earlier[6]. The evaluator presented below uses a Markov algorithm[7], so that's another example of a generatively recursive approach:

```
(check-expect
 (string-split " ( 1 + 2 ) * 3 ")
 (list "(" "1" "+" "2" ")" "*" "3")) ; this is where we start

(define (elim-parens input)
  (cond ((empty? input) input)
        ((not (left-paren? (first input)))
         (cons (first input) (elim-parens (rest input))))
        (else (cond ((and (nummer? (second input))
                          (right-paren? (third input)))
                     (cons (second input)
                           (elim-parens (rest (rest (rest input))))))
                    (else (cons (first input)
                                (elim-parens (rest input)))))))))

(define (clean-parens input)
  (let ((result (elim-parens input)))
    (if (equal? input result)
        result
        (clean-parens result))))

(check-expect (clean-parens (string-split " ( ( ( 3 ) ) ) ")) (list "3"))
```

The type of expressions we want to be able to evaluate are those made of whole numbers, `+`, `*`, parentheses and that are well-formed in the usual sense. We also ask (for `string-split`'s sake) that all tokens be separated by space, as can be shown in the examples we present.

The basic procedure is based on three passes that are repeated until there's no change[8]. First the redundant parentheses are eliminated, if any. That's the code we already saw. It relies on three predicates that are presented shortly. The second pass is to reduce all multiplications; in the absence of redundant parens this is the operation of next precedence. Finally, if there are any additions that can be performed, they are; this step takes into account that

[6] It is true that the derivation presented for `string-split` is not structural, but generative. It is similar, in fact to the `take` and `bundle` exercises in that chapter. Recall that this is a document that groups topics in categories that may differ a little from order in which they are presented in class, although we have tried to stay true to that as well.

[7] `https://en.wikipedia.org/wiki/Markov_algorithm`

[8] A little bit like bubble sorting.

unfinished multiplications might still block additions that (in a near-sighted way) look good (but aren't). Here are the three predicates:

```
(define (left-paren?  input) (string=? "(" input))
(define (right-paren? input) (string=? ")" input))
(define (nummer?      input) (number? (string->number input)))
```

Here are some example of one-step parenthesis removal:

```
(check-expect (elim-parens (string-split " 2 ")) (list "2"))
(check-expect (elim-parens (string-split " ( 2 ) ")) (list "2"))
(check-expect (elim-parens (string-split " ( ( 2 ) ) ")) (list "(" "2" ")"))
```

This is how we reduce all multiplications that can be reduced:

```
(define (elim-stars input)
  (cond ((empty? input) input)
        ((and (>= (length input) 3)
              (nummer? (first input))
              (star? (second input))
              (nummer? (third input)))
         (cons (number->string
                (* (string->number (first input))
                   (string->number (third input))))
               (rest (rest (rest input)))))
        (else (cons (first input) (elim-stars (rest input))))))

(define (star? input) (string=? "*" input))
(define (plus? input) (string=? "+" input))

(define (clean-stars input)
  (let ((result (elim-stars input)))
    (if (equal? input result)
        result
        (clean-stars result))))
(check-expect (clean-stars (string-split " 2 ")) (list "2"))
(check-expect (clean-stars (string-split " 2 * 3 ")) (list "6"))
```

Here's the slightly more involved procedure for reducing additions:

```
(define (clean-pluses input)
  (let ((result (elim-pluses input)))
    (if (equal? input result)
        result
        (clean-pluses result))))
```

In many ways these are end of semester exercises because the strategy used here is (clearly, and on purpose) accumulator-passing style.

```
; elim-plus-at : Number [ListOf Token] -> [ListOf Token]
; reduces plus at index, assumes green light has been given
(define (elim-plus-at index input)
  (cond ((= index 1) (cons (number->string (+ (string->number (first input))
                                              (string->number (third input))))
                           (rest (rest (rest input)))))
        (else (cons (first input)
                    (elim-plus-at (sub1 index) (rest input))))))

(check-expect (elim-plus-at 1 (string-split " 2 + 3 ")) (list "5"))
(check-expect (elim-plus-at 2 (string-split " ( 1 + 2 ) * 3 "))
              (list "(" "3" ")" "*" "3"))

(check-expect (elim-plus-at 8 (string-split " ( 1 + 3 ) * ( 4 + 5 ) "))
              (list "(" "1" "+" "3" ")" "*" "(" "9" ")"))

(define (elim-pluses input)
  (travel-from 0 input))

(define (travel-from index input)
  (cond ((= index (length input)) input)
        ((checks-out index input) (elim-plus-at index input))
        (else (travel-from (add1 index) input))))

(define (checks-out index input)
  (and (plus? (list-ref input index))
       (and (>= (sub1 index) 0) (nummer? (list-ref input (sub1 index))))
       (and (< (add1 index) (length input)) (nummer? (list-ref input (add1 index))))
       (or  (< (- index 2) 0) (not (star? (list-ref input (- index 2)))))
       (or (>= (+ index 2) (length input)) (not (star? (list-ref input (+ index 2)))))))

(check-expect (checks-out 1 (string-split " 1 + 2 + 3 + 4 ")) true)

(define (eval input)
  (let ((result (clean-pluses (clean-stars (clean-parens input)))))
    (if (equal? result input)
        result
        (eval result))))

(check-expect (eval (string-split " 1 + 3 * 4 + 5 ")) (list "18"))
(check-expect (eval (string-split " ( 1 + 3 ) * 4 + 5 ")) (list "21"))
(check-expect (eval (string-split " ( 1 + 3 ) * ( 4 + 5 ) ")) (list "36"))
(check-expect (clean-pluses (string-split " ( 1 + 3 ) * ( 4 + 5 ) "))
                            (string-split " ( 4 ) * ( 9 ) "))
(check-expect (eval (string-split " ( 1 + 3 ) * 2 ")) (list "8"))
(check-expect (elim-pluses (string-split " 1 + 2 * 3 ")) (list "1" "+" "2" "*" "3"))
(check-expect (elim-pluses (string-split " ( 1 + 2 ) * 3 ")) (list "(" "3" ")" "*" "3"))
```

Chapter 17

Trees

Trees are an important data model for representing hierarchical information. Many data structures involving combinations of arrays and pointers can be used to implement trees, and the data structure of choice depends on the operations performed on the tree. Two of the most important representations for tree nodes are the leftmost-child-right-sibling representation and the trie (array of pointers to children). Recursive algorithms and proofs are well suited for trees. A variant of our basic induction scheme, called structural induction, is effectively a complete induction on the number of nodes in the tree.

The binary tree is a variant of the tree model in which each node has (optional) left and right children. A binary search tree is a labeled binary tree with the "binary search property" that all labels in the left subtree precede the label at a node, and all labels in the right subtree follow the label at the node. The dictionary abstract data type is a set upon which we can perform the operations `insert`, `delete`, and `lookup` (recall our `lambda` representation of sets). The binary search tree efficiently implements dictionaries. A priority queue is another abstract data type, a set upon which we can perform the operations `insert` and `deleteMax`. A partially ordered tree is a labeled binary tree with the property that the label at any node is at least as great as the label at its children. Balanced partially ordered trees, where the nodes fully occupy levels from the root to the lowest level, where only the leftmost positions are occupied, can be implemented by an array structure called a heap. This structure provides an $\mathcal{O}(\log n)$ sorting algorithm called heapsort.

```
; A BinaryTree is one of:
;   -- Number

;   -- (make-bt BinaryTree BinaryTree)
(define-struct bt (left right))

(define ex01 (make-bt (make-bt 1 (make-bt 2 3))
                      (make-bt 4 (make-bt 5 6))))
```

You have seen this data definition and example before.

```
; sum : BinaryTree -> Number
; sums the leaves of a binary tree
(define (sum bt)
  (cond ((number? bt) bt)
        (else (+ (sum (bt-left bt))
                 (sum (bt-right bt))))))

(check-expect (sum ex01) 21)
```

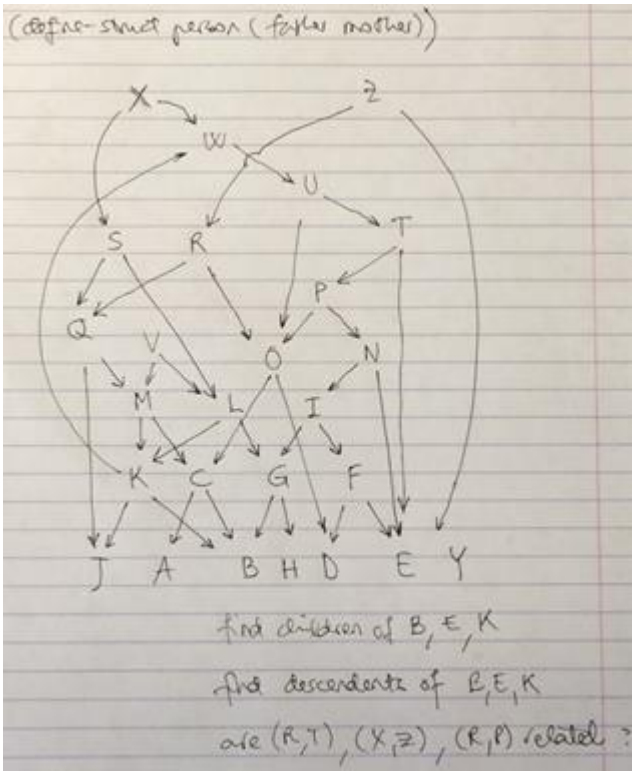

Figure 17.1: This diagram is about (a specific set of) scientists. A friend drew it for you. Things to know: Two mentors can raise a scientist (from an apprentice/student). A scientist can mentor as soon as it reaches that stage. A scientist remembers her/his mentors. You can become scientist only once. If you have the same mentor you are "related". If you have the same exact mentors you are "siblings". Design a data representation then design a function that would answer questions like those listed. Additional questions: are there any siblings in it? How many unrelated people are there? What's the largest set of unrelated people in there?

Chapter 18

Sets

The set is the most fundamental data model of mathematics. Every concept in mathematics, from trees to real numbers, is expressible as a special kind of set. Thus, it should not be surprising that sets are als a fundamental model of computer science. The common operations on sets, such as union, intersection, and difference can be visualized in terms of Venn diagrams.

Algebraic laws can be used to manipulate and simplify expressions involving sets and operations on sets.

(Binary) relations are sets of pairs. A function is a relation in which there is at most one tuple with a given first component. A one-to-one correspondence between two sets is a function that associates a unique element of the second set with each element of the first, and viceversa. There are a number of significant properties of binary relations: reflexivity, transitivity, symmetry, and asymmetry are among the most important.

Partial orders, total orders, and equivalence relations are important special cases of binary relations.

Infinite sets are those sets that have a one to one correspondence with one of their proper subsets. Some infinte sets are "countable," that is, they have a one to one correspondence with the integers. Other infinite sets, such as the reals, are not countable.

18.1 Relational Algebra

One of the most important applications for computers is storing and managing information. The manner in which information is organized can have a profound impact on how easy it is to access and manage. Perhaps the simplest but most versatile way to organize information is to store it in tables.

The relational model is centered on this idea: the organization of data into collections of two-dimensional tables called "relations". We can also think of the relational model as generalization of the set data model. Originally, the relational data model was developed for databases. Other kinds of software

besides database systems can make good use of tables of information as well, for example, tables are used in compilers etc.

Relational algebra is a high-level notation for specifiying queries about one or more relations. It's principal operations are union, intersection, difference, selection, projection and join (which defined as a natural subset of the cartesian product). There are several kinds of joins that can be defined and are useful.

Optimizations of expressions in relational algebra can make significant improvements in the running time for the evaluation of expressions and is therefore essential if languages based on relational algebra are to be used in practice to express queries.

18.2 NoSQL

At the core of most large-scale applications and services is a high-performance data storage solution. The back-end data store is responsible for storing important data such as user account information, product data, accounting information, blogs. Good applications require the capability to store and retrieve data with accuracy, speed, and reliability. Therefore, the data storage mechanism we choose must be capable of performing at a level that satisfies your application's demand. Several data storage solutions are available to store and retrieve the data your applications need. The three most common are: direct file system storage in files, relational databases, and NoSQL databases.

The realization that enterprise data can be stored in structures other than RDBMSs is a major turning point for people who enter the NoSQL space. For our purpose, NoSQL is a set of concepts that allows the rapid and efficient processing of data sets with a focus on performance, reliability, and agility. NoSQL is more than rows in tables, it's free of joins, it's schema-free, it works on many processors, it supports linear scalability and it's innovative. NoSQL is not about the SQL language, it's not only open source, it's not only big data, it's not about a clever use of RAM and SSD and it's not an elite group of products. It's just a new reality.The motivation behind NoSQL is: simplified design, horizontal scaling and finer control over the availability of data.

The rise of NoSQL databases marks the end of the era of relational database dominance. However the NoSQL databases will not become the new dominators. While relational database management systems will still be popular, and probably used in the majority of situations, they will no longer be the automatic choice. The era of polyglot persistence has begun. Although Big Data remains the main driver for the NoSQL rise, it is not the only reason to use NoSQL. Many NoSQL databases are designed to run well on large clusters, which makes them more attractive for large data volumes, but often people select NoSQL due to easier database interaction in their applications.

Chapter 19

Generative Recursion

Here's the template for generative recursion:

```
(define (generative-recursive-fun problem)
  (cond
    [(trivially-solvable? problem)
     (determine-solution problem)]
    [else
     (combine-solutions
       ... problem ...
       (generative-recursive-fun (generate-problem-1 problem))
       ...
       (generative-recursive-fun (generate-problem-n problem)))]))
```

Here's a comment[1] from the book:

> Is there a real difference between structural recursive design and the one for generative recursion? Our answer is"it depends" Of course, we could say that all functions using structural recursion are just special cases of generative recursion. This "everything is equal" attitude, however, is of no help if we wish to understand the process of designing functions. It confuses two kinds of design that require different forms of knowledge and that have different consequences. One relies on a systematic data analysis and not much more; the other requires a deep, often mathematical, insight into the problem-solving process itself. One leads programmers to naturally terminating functions; the other requires a termination argument. Conflating these two approaches is simply unhelpful.

Let's go through some examples to clarify this.

[1] 26.3 Structural vs. Generative Recursion

19.1 merge

Here's an idea for a sort based on `merge` (a function that takes two sorted sequences and merges them into a(nother, larger, still) sorted sequence) and two helper functions, `longest` and `rest`:

```
(sort seq) = (merge (longest seq)
                    (sort (rest seq)))
```

Here `longest` returns the longest prefix already sorted and `rest` returns what's left. Here's how this works step by step with annotations as to why we got there:

```
(sort [4, 5, 2, 3, 1, 7, 6]) is...
   because (longest [4, 5, 2, 3, 1, 7, 6]) is [4, 5]
       and (rest    [4, 5, 2, 3, 1, 7, 6]) is [2, 3, 1, 7, 6]
(merge [4, 5] (sort [2, 3, 1, 7, 6])) which is...
   because (longest [2, 3, 1, 7, 6]) is [2, 3]
       and (rest    [2, 3, 1, 7, 6]) is [1, 7, 6]
(merge [4, 5] (merge [2, 3] (sort [1, 7, 6])) which is...
   because (longest [1, 7, 6]) is [1, 7]
        and (rest    [1, 7, 6]) is [6]
(merge [4, 5] (merge [2, 3] (merge [1, 7] (sort ([6])))))) which is...
   because    (sort [6]) is [6]

(merge [4, 5] (merge [2, 3] (merge [1, 7] [6]))) which is...

(merge [4, 5] (merge [2, 3] [1, 6, 7])) which is...

(merge [4, 5] [1, 2, 3, 6, 7]) which is...

[1, 2, 3, 4, 5, 6, 7]
```

Implement this idea. Clearly this looks so much like insertion sort (which was classified as having a structural recursive design). It should be now clear that the distinction between structural and generative recursion design is sometimes very subtle, somewhat subjective and/or often times fuzzy.

19.2 quickSort

Quicksort (sometimes called partition-exchange sort) is an efficient sorting algorithm, serving as a systematic method for placing the elements of an array in order. Developed by Tony Hoare in 1959 and published in 1961, it is still a commonly used algorithm for sorting. It is probably the fastest sorting algorithm in practice, but has some pathological cases (that don't happen often in practice, since real-world data is not shuffled, it usually has some hidden order in it) where it underperforms severely.

Here's the code we will develop in class:

```
(define (q-sort lon)
  (cond ((empty? lon) lon)
        ; empty list is sorted already
        ((empty? (rest lon)) lon)
        ; one element list sorted too
        (else (let ((pivot (list-ref lon (quotient (length lon) 2))))
                  ; choose a pivot
                (let ((leq (remove pivot (filter (lambda (elem) (<= elem pivot))
                                                 lon)))
                   ; create lists
                      (gt (filter (lambda (elem) (> elem pivot)) lon)))
                   ; via filtering (thanks ISL!)
                  (append (q-sort leq) (list pivot) (q-sort gt)))))))
                   ; then recur and combine

(check-expect (q-sort '(4 2 6 4 1 5 3 3 7 2)) (list 1 2 2 3 3 4 4 5 6 7))
```

Here's how `quicksort` runs, step by step:

```
(qsort (list 4 2 3 1 5 4 3 6 2 3)) =

(append (qsort (list 4 2 3 1 2 3))
        (list 4)
        (qsort (list 5 6)))        =
```

As a reminder, the pivots are chosen randomly. We continue:

```
(append (append (qsort empty)
                (list 1)
                (qsort (list 4 2 3 2 3)))
        (list 4)
        (append (qsort (list 5))
                (list 6)
                (qsort empty)))    =

(append (append empty
                (list 1)
                (append (qsort (list 2 2 3))
                        (list 3)
                        (qsort (list 4))))
        (list 4)
        (append (list 5)
                (list 6)
                empty))            =
```

```
(append (append empty
                (list 1)
                (append (append (qsort (list 2))
                                (list 2)
                                (qsort (list 3)))
                        (list 3)
                        (list 4)))
        (list 4)
        (append (list 5)
                (list 6)
                empty))            =

(append (append empty
                (list 1)
                (append (append (list 2)
                                (list 2)
                                (list 3))
                        (list 3)
                        (list 4)))
        (list 4)
        (append (list 5)
                (list 6)
                empty))            =

(append (append empty
                (list 1)
                (append (list 2 2 3)
                        (list 3)
                        (list 4)))
        (list 4)
        (list 5 6))                =

(append (append empty
                (list 1)
                (list 2 2 3 3 4))
        (list 4)
        (list 5 6))                =

(append (list 1 2 2 3 3 4)
        (list 4)
        (list 5 6))                =

(list 1 2 2 3 3 4 4 5 6)
```

19.3 permutations

There's a very interesting problem early in your text that goes like this:

> Louise, Jane, Laura, Dana, and Mary decide to run a lottery that assigns one gift recipient to each of them (Secret Santa). Since Jane is a developer, they ask her to write a program that performs this task in an impartial manner[2]. Of course, the program must not assign any of the sisters to herself.

The key challenge here may be to produce the set of all permutations[3]. However, in reality you need to be ready for a *tour de force* through several chapters of the book.

Here's how my program works:

```
Welcome to DrRacket, version 6.2 [3m].
Language: Intermediate Student with lambda; memory limit: 256 MB.
> (make-pairs '(Robert Joel Taylor Kristopher Yibo))
  ; Team Four
(list
 (list 'Taylor 'buys 'gift 'for 'Robert)
 (list 'Yibo 'buys 'gift 'for 'Joel)
 (list 'Kristopher 'buys 'gift 'for 'Taylor)
 (list 'Joel 'buys 'gift 'for 'Kristopher)
 (list 'Robert 'buys 'gift 'for 'Yibo))
> (make-pairs '(Robert Joel Taylor Kristopher Yibo))
  ; notice each call generates a random solution
(list
 (list 'Yibo 'buys 'gift 'for 'Robert)
 (list 'Taylor 'buys 'gift 'for 'Joel)
 (list 'Kristopher 'buys 'gift 'for 'Taylor)
 (list 'Robert 'buys 'gift 'for 'Kristopher)
 (list 'Joel 'buys 'gift 'for 'Yibo))
> (make-pairs '(Chang Tao John Eric Aining Liping))
  ; Team Two
(list
 (list 'Aining 'buys 'gift 'for 'Chang)
 (list 'John 'buys 'gift 'for 'Tao)
 (list 'Eric 'buys 'gift 'for 'John)
 (list 'Liping 'buys 'gift 'for 'Eric)
 (list 'Chang 'buys 'gift 'for 'Aining)
 (list 'Tao 'buys 'gift 'for 'Liping))
> (make-pairs '(dgerman caudrets ruifzhen jwmeert))
```

[2] All assignments should be equally likely.

[3] Two methods will be presented. One of them is shown here. You will have to design and implement the other one.

```
  ; instructors
 (list
  (list 'caudrets 'buys 'gift 'for 'dgerman)
  (list 'ruifzhen 'buys 'gift 'for 'caudrets)
  (list 'jwmeert 'buys 'gift 'for 'ruifzhen)
  (list 'dgerman 'buys 'gift 'for 'jwmeert))
> (make-pairs '(dgerman caudrets ruifzhen jwmeert))
  ; instructors
 (list
  (list 'jwmeert 'buys 'gift 'for 'dgerman)
  (list 'ruifzhen 'buys 'gift 'for 'caudrets)
  (list 'caudrets 'buys 'gift 'for 'ruifzhen)
  (list 'dgerman 'buys 'gift 'for 'jwmeert))
> (make-pairs '(dgerman caudrets ruifzhen jwmeert))
  ; instructors
 (list
  (list 'ruifzhen 'buys 'gift 'for 'dgerman)
  (list 'jwmeert 'buys 'gift 'for 'caudrets)
  (list 'dgerman 'buys 'gift 'for 'ruifzhen)
  (list 'caudrets 'buys 'gift 'for 'jwmeert))
>
```

Here's a key part of it:

```
; subtract-from : [ListOf Atom] Number -> [ListOf Atom]
; returns the list without the element at valid index index
(define (subtract-from los index)
  (cond ((zero? index) (rest los))
        (else (cons (first los) (subtract-from (rest los) (sub1 index))))))

(check-expect (subtract-from '(a b c d e f) 2) '(a b d e f))
(check-expect (subtract-from '(a b c) 0) '(b c))
(check-expect (subtract-from '(a b c d) (sub1 (length '(a b c d)))) '(a b c))

; perm : [ListOf Atom] -> [ListOf [ListOf Atom]]
; calculates and returns all permutations of its input
(define (perm los)
  (cond ((empty? los) (list empty))
        ((list? los) (apply append
                            (map (lambda (index)
                                   (map (lambda (x)
                                          (cons (list-ref los index) x))
                                        (perm (subtract-from los index))))
                                 (build-list (length los)
                                             (lambda (x) x)))))
        (else (error "Bad input."))))
```

Chapter 20

Backtracking

Backtracking is a general algorithm for finding all (or some) solutions to some computational problems[1], that incrementally builds candidates to the solutions, and abandons each partial candidate ("backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution.

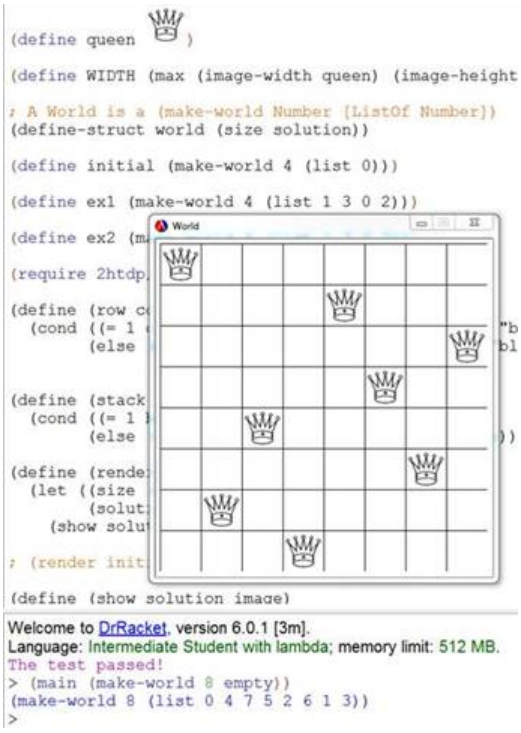

Figure 20.1: We will develop in class a program to solve the n-Queen problem.

[1] Notably constraint satisfaction problems.

The classic textbook example of the use of backtracking is the eight (more generally n) queens puzzle, that asks for all arrangements of queens on an $n \times n$ chessboard so that no queen attacks each other. The partial candidates are arrangements of k queens in the first k rows of the board, all in different rows and columns. Any partial solution that contains two mutually attacking queens can be abandoned.

Backtracking can be applied only for problems which admit the concept of a "partial candidate solution" and a relatively quick test of whether it can possibly be completed to a valid solution.

Everything starts with data representation. Given the n-queen problem we define a data representation for the solution we're building. If the space of potential solutions is a tree it can be explored easily within a relatively simple generative recursive design.

20.1 Super-Polynomial Time

Here's a solution to the Towers of Hanoi problem worked out with Jerry Haoxuan Shen on June 8, 2016 in class. Switch to ASL briefly for this program:

```
; tower of hanoi

(define (hanoi size start middle end)
  (cond ((= size 1) (begin (display (string-append "move disk from " start " to " end)) (newline)))
        (else (begin
                (hanoi (- size 1) start end middle)
                (begin (display (string-append "move disk from " start " to " end)) (newline))
                (hanoi (- size 1) middle start end)))))
```

Figure 20.2: Such a short program, such a (computationally) hard problem.

Figure 20.3: The Tower of Hanoi is $\mathcal{O}(2^n)$ (intractably hard).

Chapter 21

Accumulators

```
; 0. sum the numbers in a list of numbers
;
; [ListOf Number] -> Number
; sums the numbers in the given list
;
(define (sum lon)
  (local (
          ; [ListOf Number] Number -> Number
          ; adds the numbers in the list one by one to the second argument
          ;
          ; accumulator: remaining-numbers, numbers still left to be processed
          ; accumulator: sum-thus-far, the answer being built iteration by iteration
          ;
          ; example:
          ;   remaining-numbers    sum-thus-far
          ; ------------------------------------------------------
          ;  (1 2 3 4 5)                 0
          ;    (2 3 4 5)              0 + 1 = 1
          ;      (3 4 5)              1 + 2 = 3
          ;        (4 5)              3 + 3 = 6
          ;          (5)              6 + 4 = 10
          ;        empty             10 + 5 = 15
          ;
          (define (helper remaining-numbers sum-thus-far)
            (cond ((empty? remaining-numbers) ; termination condition
                   sum-thus-far) ; sum-thus-far becomes the final answer
                  (else (helper (rest remaining-numbers) ; fewer numbers remain
                                (+ (first remaining-numbers) ; update sum-thus-far
                                   sum-thus-far)))))
          )
    (helper lon 0))) ; initial call to the helper function
    ; (notice the initial values of the accumulators)

(check-expect (sum '(1 2 3 4 5)) (+ 0 1 2 3 4 5))
```

The following is a variant of `reverse` that is very fast. It's meant to represent an alternative solution[1], to the exercise at the end of *31.1 A Problem with Structural Processing*. Here the accumulator is a procedure (continuation):

```
; We decided to trade space for time and proposed the following version
; of reverse written with the design recipe of Part II (Fig. 38) in mind
; but using a very important (and convenient) aspect of ISL+ (you know it!)
;
;  [ListOf Number] -> [ListOfNumber]
;  reverses a list of numbers
(define (umkehren lon) ; umkehren means reverse in German
   (local (
          ;  [ListOf Number] -> [ [ListofNumber] -> [ListOfNumber] ]
          ;  builds and returns the actual computation
          ;  (define (rev lon)
          ;    (cond ((empty? ...) ...)
          ;          (else ( ... (rev (rest lon)) ... (first lon) ... ))))
           (define (rev lon)
             (cond ((empty? (rest lon)) (lambda (x) (cons (first lon) x)))
                   (else (lambda (x)
                           ((rev (rest lon))
                            ((lambda (x) (cons (first lon) x)) x))))))
          )
          ;  we invoke 19.4 for the two lines below
     (cond ((empty? lon) empty)
           (else ((rev lon) empty)))))

(umkehren empty)
(umkehren (list 1))
(umkehren (list 1 2))
(umkehren (list 1 2 3))
(umkehren (list 1 2 3 4))
(umkehren (list 1 2 3 4 5))

(time
  (let ((a (reverse (build-list 1000000 add1))))
    'done)) ; cpu time: 94 real time: 94 gc time: 63

(time
  (let ((a (umkehren (build-list 1000000 add1))))
    'done)) ; cpu time: 2777 real time: 2807 gc time: 935

(define (old-rev lon)
   (cond ((empty? lon) lon)
         (else (append (old-rev (rest lon)) (list (first lon))))))
(time
  (let ((a (old-rev (build-list 100000 add1)))) ; ten times shorter
    'done)) ; cpu time: 116330 real time: 116901 gc time: 51531
```

[1] Worked out with Mardin Yadegar, in Center Grove and B'ton on 11/18/2014.

Chapter 22

Graphs

A graph is, in a sense, nothing more than a binary relation.

However, it has a powerful visualization as a set of points (called nodes) connected by lines (called edges) or by arrows (called arcs). In this regard the graph is a generalization of the tree data model.

Figure 22.1: Ticket to Ride: map of the North American train routes.

In class we will introduce most of the key concepts of graph theory. These include: paths and shortest paths, spanning trees, depth-first search trees and

forests, graph coloring and the chromatic number, cliques and clique numbers and planar graphs. Some algorithms that are usually discussed: minimal spanning tree, detecting cycles, topological order, single-source reachability, connected components, single-source shortest path, and all-pairs shortest path. In class we will try to understand roughly what these are and we will work out in detail only one of them. Here's a data representation for the map shown earlier (related to the incidence matrix of the graph):

```
(define
  ticket-to-ride
  '(
    ("vancouver" ("calgary" "seattle"))
    ("calgary" ("vancouver" "seattle" "helena" "winnipeg"))
    ("winnipeg" ("calgary" "helena" "duluth" "sault ste. marie"))
    ("sault ste. marie" ("winnipeg" "duluth" "toronto" "montreal"))
    ("montreal" ("boston" "new york" "toronto" "sault ste. marie"))
    ("boston" ("montreal" "new york"))
    ("new york" ("washington" "pittsburgh" "montreal" "boston"))
    ("toronto" ("sault ste. marie" "montreal" "pittsburgh" "duluth" "chicago"))
    ("pittsburgh" ("toronto" "new york" "washington" "raleigh" "nashville" "saint louis" "chicago"))
    ("washington" ("new york" "pittsburgh" "raleigh"))
    ("raleigh" ("charleston" "atlanta" "nashville" "pittsburgh" "washington"))
    ("charleston" ("raleigh" "atlanta" "miami"))
    ("miami" ("charleston" "atlanta" "new orleans"))
    ("atlanta" ("raleigh" "charleston" "miami" "new orleans" "nashville"))
    ("nashville" ("saint louis" "little rock" "atlanta" "raleigh" "pittsburgh"))
    ("chicago" ("pittsburgh" "saint louis" "toronto" "duluth" "omaha"))
    ("saint louis" ("chicago" "pittsburgh" "nashville" "little rock" "kansas city"))
    ("little rock" ("nashville" "new orleans" "dallas" "oklahoma city" "saint louis"))
    ("new orleans" ("houston" "little rock" "atlanta" "miami"))
    ("houston" ("el paso" "dallas" "new orleans"))
    ("dallas" ("little rock" "houston" "el paso" "oklahoma city"))
    ("oklahoma city" ("denver" "kansas city" "little rock" "dallas" "el paso" "santa fe"))
    ("kansas city" ("omaha" "saint louis" "oklahoma city" "denver"))
    ("omaha" ("helena" "duluth" "chicago" "kansas city" "denver"))
    ("duluth" ("winnipeg" "sault ste. marie"  "toronto" "chicago" "omaha" "helena"))
    ("helena" ("seattle" "calgary" "winnipeg" "duluth" "omaha" "denver" "salt lake city"))
    ("salt lake city" ("portland" "helena" "denver" "las vegas" "san francisco"))
    ("denver" ("salt lake city" "helena" "omaha" "kansas city" "oklahoma city" "santa fe" "phoenix"))
    ("santa fe" ("denver" "oklahoma city" "el paso" "phoenix"))
    ("el paso" ("los angeles" "phoenix" "santa fe" "oklahoma city" "dallas" "houston"))
    ("phoenix" ("los angeles" "denver" "santa fe" "el paso"))
    ("las vegas" ("salt lake city" "los angeles"))
    ("san francisco" ("portland" "salt lake city" "los angeles"))
    ("los angeles" ("san francisco" "las vegas" "phoenix" "el paso"))
    ("portland" ("seattle" "salt lake city" "san francisco"))
    ("seattle" ("vancouver" "calgary" "helena" "portland"))
   )
)
```

Chapter 23

ASL

ASL has: `display`, `newline`, `set!`. The real world too has lots of things, some of which are tempting (perhaps) in the immediate context but questionable for their long term effect. In this regard assignment statements can be abused, like alcohol, various other substances including prescription drugs and/or tobacco or tobacco-like products (that may even be legal in some states, but that's beside the point here). They all need to be used with care and in some sense that's why we placed them in ASL: you kinda need to be 21 (or an adult programmer) to use them. It is our hope that by now you have formed all good habits that will protect you as you gain access to adult programming material like `set!` and `display`. Think `big-bang` and accumulators if you feel the need to show things (`to-render` them) or modify them as you create new instances in a new world (`on-tick`).

Using the data representation in the previous chapter we need to write a program that finds the shortest path[1] between two cities on the map:

```
Welcome to DrRacket, version 6.5 [3m].
Language: Intermediate Student with lambda; memory limit: 128 MB.
Both tests passed!
> (find-shortest-path "los angeles" "montreal" ticket-to-ride)
(list
 "los angeles"
 "san francisco"
 "salt lake city"
 "helena"
 "winnipeg"
 "sault ste. marie"
 "montreal")
> (find-shortest-path "los angeles" "toronto" ticket-to-ride)
(list "los angeles" "san francisco" "salt lake city" "helena" "duluth" "toronto")
```

It doesn't matter if your program is breadth-first or depth-first.

[1] As you will see, there will be many ways to define "shortest".

Here's a potential solution (using ASL for `begin` and `display`).

```
(define (find-shortest-path start end in-graph)
    (local (
           ;  Node -> [ListOf Node]
           ;  determines the list of in-graph successors of node
            (define (neighbors-of node)
              (let ((pair (assq node in-graph))) ; note the scope
                (if (false? pair)
                    empty
                    (second pair))))
           ;  Node Node [ListOf Node] [ListOf Node] -> [ListOf Node]
           ;  returns the best path between start and end
            (define (find-paths start end current best)
              (cond ((equal? start end) (choose-shorter (cons start current) best))
                    ((and (not (empty? best))
                          (> (length current) (length best))) best)
                    (else (let ((neighbors (neighbors-of start)))
                            (let ((neighbors (filter (lambda (node)
                                                       (not (member? node current)))
                                                     neighbors)))
                              (helper neighbors end (cons start current) best))))))
           ;  [ListOf Node] Node [ListOf Node] [ListOf Node] -> [ListOf Node]
            (define (helper options target current best)
              (cond ((empty? options) best)
                    (else (let ((answer (find-paths (first options) target
                    current best)))
                            (helper (rest options) target current
                            (choose-shorter answer best))))))
           ;  [ListOf Node] [ListOf Node]
           ;  selects the shorter of the two according to length
            (define (choose-shorter a b)
              (cond ((or (empty? b)
                         (< (length a) (length b))) a)
                    (else b)))
           ) ; end of local definitions
      (reverse (find-paths start end empty empty))))

(check-expect
 (find-shortest-path "los angeles" "montreal" ticket-to-ride)
 (list "los angeles" "san francisco" "salt lake city"
       "helena" "winnipeg" "sault ste. marie" "montreal"))

(check-expect
 (find-shortest-path "seattle" "miami" ticket-to-ride)
 (list "seattle" "helena" "denver" "oklahoma city"
       "little rock" "new orleans" "miami"))
```

I guess this is not the ASL program I wanted to show, this is a regular ISL/+ a-ps program that solves the problem via BFS. I'll add the other one here soon.

Chapter 24

Universe

The extent of this exercise is significantly larger during the regular academic semesters. In 6W1 Summer we don't get to work on it as much, save for what we show below. This chapter could have also been entitled "Servers and ports: using `silo` to deploy". Below, I am following own notes from 12/13/2014[1] and 04/13/2016[2].

Assume you have a Windows machine. You want to connect to `silo`[3]. You must have `PuTTY` (get it from IUWare for free, just your IUB Network ID is needed. When you start it, you need to watch for two things: setting X11 forwarding and starting Xming (if you have it installed[4]). Both Xming and `PuTTY` are free and installed on all Windows machines on campus already.

Once Xming starts you should see this on the taskbar:

Figure 24.1: Xming is a very discreet program.

Connect to `silo`. At the prompt start `xeyes` or `xclock`, as a test. Once you're sure you have a reliable connection and X11 forwarding happens correctly stop them and load racket, which then becomes available to you at the prompt on silo. We discuss modules. They are instrumental in producing standalone executables[5] In class we will compare this with Python executables. Does it work with world and universe programs too? You bet it does.

And at this point we should run a simple BigBang program through our X11 connection just to be sure all works well (like with the `xeyes`).

[1] `https://www.cs.indiana.edu/classes/h211-dgerman/fall2014/mo.html`

[2] `http://silo.cs.indiana.edu:8346/h212/spr2016/0413a.phps`

[3] `silo` runs Linux, so once you connect you need to learn and start using various Unix commands. Unix may appear cryptic at first, it is said if you mistype a Unix command you probably get another Unix command, but it is actually beautiful, friendly and powerful. Learn to love it. It will love you back.

[4] If you don't, it's free and there are complete instructions in the IUB Knowledge Base.

[5] And Matthias suggested an even better approach, need to document it here soon.

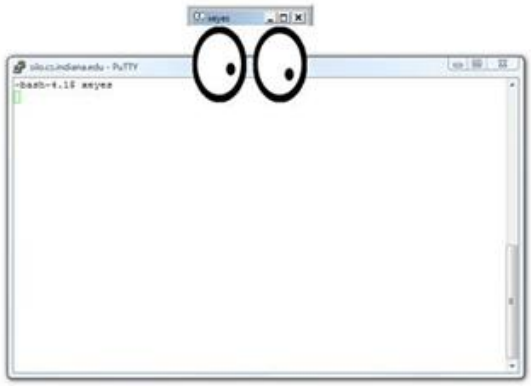

Figure 24.2: `xeyes` running through our connection.

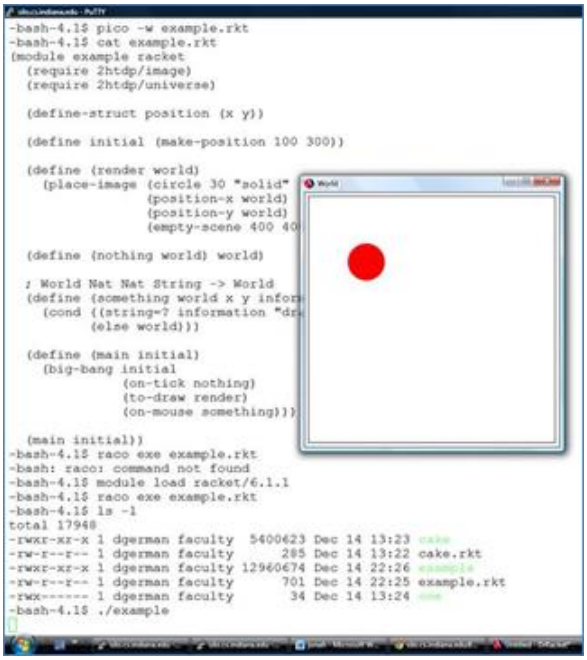

Figure 24.3: A simple `big-bang` program running through our connection.

Now deployment means that:

- we turn the server into a module
- indicate a port, compile and start it
- then ask the clients to connect to the server

First determine a free port:

```
-bash-4.1$ pwd
/u/dgerman/deployment
-bash-4.1$ nslookup silo.cs.indiana.edu
Server:         129.79.1.1
Address:        129.79.1.1#53

silo.cs.indiana.edu     canonical name = silo.soic.indiana.edu.
Name:   silo.soic.indiana.edu
Address: 129.79.247.5

-bash-4.1$ netstat -a | grep 31415
-bash-4.1$
```

Here's the relevant part of the server:

```
-bash-4.1$ cat server.rkt
(module server racket

(define-struct posn (x y))

(require 2htdp/image)
(require 2htdp/universe)

; A Client is (make-world IWorld Nat Nat)
(define-struct client (world x y))

;
; the rest of the code the same as what you developed in LOCALHOST mode
;
;          ...
           (else (make-ball (+ x vx) (+ y vy) vx vy)))))
           ; keep moving according to current momentum

; take out many worlds, keep the universe

(universe (list empty (make-ball 200 200 1 2))
           [port 31415] ; please use an own port that's still available
           [on-msg update-clients]
           [on-new register-client]
           [on-tick move-ball-and-broadcast]) ; last piece of the puzzle
) ; this closes the module up top
```

Compile the server and start it on silo.

Then here's the relevant part of the client:

```
(require 2htdp/image)
(require 2htdp/universe)

; A Client is (make-world IWorld Nat Nat)
(define-struct client (world x y))

; the rest of the code the same as what you developed in LOCALHOST mode
;
;          ...
           (else (make-ball (+ x vx) (+ y vy) vx vy)))))
           ;; keep moving according to current momentum

; take the many worlds out, run a big-bang by itself
(big-bang
 (make-world (make-posn 0 200 ) (make-posn 400 200) (make-ball 200 200 1 2))
 [register "129.79.247.5"]
 [port 31415]
 [on-receive message-from-server]
 [on-key send-message]
 [to-draw draw-world])
```

Now load a client in DrRacket on a PC, and another client on another PC, then run them. Here's how it looks for me (I ran both clients on my laptop):

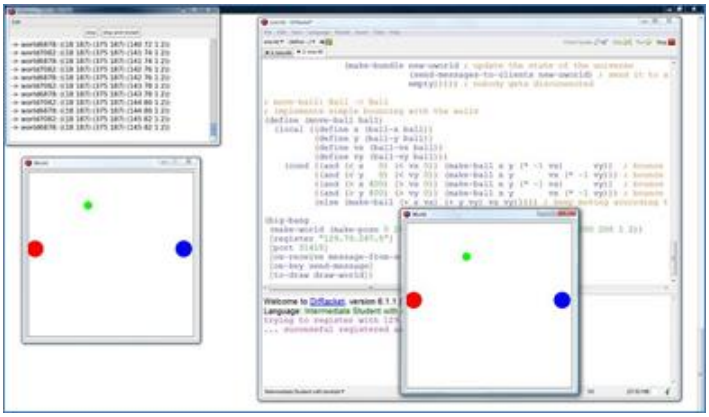

Figure 24.4: Simple multiplayer shared environment relying on Universe teachpack. In C212/A592 you will design all of this by yourself, from the ground up, in Java (w/ MVC); we'll also discuss additional types of network and web programming then.

Chapter 25

Racket

Consider the following document[1]. Compare with Realm of Racket[2] (RoR) and even Land of Lisp[3]. Chapter 2 of RoR[4] presents a game that runs like this:

```
Welcome to DrRacket, blah-blah-blah.
50
> (bigger)
75
> (smaller)
62
> (bigger)
68
> (smaller)
65
> (smaller)
63
>
```

Essentially, I think of a number[5], then run the program. After each guess I indicate to the program if it should try higher or lower. Here's how you can implement this program without `set!`:

```
; this still needs ASL for begin and display (but no set!)
(define (guess low high)
  (lambda (input)
    (let ((half (quotient (+ low high) 2)))
      (cond ((eq? input 'guess) half)
            ((eq? input 'higher) (guess half high))
            ((eq? input 'lower) (guess low half))
            (else 'done)))))
```

[1] http://silo.cs.indiana.edu:8346/c211/impatient/cacm-draft.pdf

[2] http://realmofracket.com/

[3] http://landoflisp.com/

[4] Realm of Racket is not a textbook, but instead a book that bridges the gap between the programming languages used in this course and Racket programming.

[5] Here my secret number was 63.

That was the key procedure, the rest of the program follows.

```
(define (conversation monad)
  (cond ((eq? monad 'done) "I knew it.")
        (else (begin
                (map display (list "is it " (monad 'guess) " ? "))
                (conversation (monad (read)))))))

(conversation (guess 1 100))
```

Here's the code from Chapter 2 of Realm of Racket for your reference:

```
#lang racket ; that's how realm of racket does it
(define lower 1) ; since they would not exist otherwise
(define upper 100)
(define (guess)
  (quotient (+ lower upper) 2))
(define (smaller)
  (set! upper (max lower (sub1 (guess))))
  (guess))
(define (bigger)
  (set! lower (min upper (add1 (guess))))
  (guess))
(define (start n m)
  (set! lower (min n m))
  (set! upper (max n m))
  (guess))
(start 1 100)
```

Run each one, annotate them, then please let me know what you think. Here[6]'s a game (a version of Nim) we developed with Lindsay Koons and Scott Babbitt during office hours on June 21, 2014. If you look at the URL posted below you will see the code was based on the following:

```
#lang racket

(define game
  (lambda()
    (display "prompt> ")
    (let ((input (read)))
      (display "You entered: ")
      (display input)
      (newline)
      (if (equal? input 'bye)
          'bye
          (game)))))
```

Can you tell me what the program above does[7]?

[6] https://www.cs.indiana.edu/classes/c211-dgerman/sum2014/lectureTwentyOne.html

[7] Furthermore, what is its design type?

Chapter 26

OOP

Consider the following code:

```
#lang racket

(define (make-counter name initial)
  (let ((balance initial)
        (owner name))
    (lambda ()
      (set! initial (add1 initial))
      (display (string-append
                owner ": " (number->string initial)))
      (newline))))
```

Here's how it runs:

```
Welcome to DrRacket, version 6.2.1 [3m].
Language: racket; memory limit: 128 MB.
> (define dgerman (make-counter "Adrian" 10))
> (dgerman)
Adrian: 11
> (dgerman)
Adrian: 12
> (define mardin (make-counter "Mardin" 8))
> (mardin)
Mardin: 9
> (mardin)
Mardin: 10
> (mardin)
Mardin: 11
> (dgerman)
Adrian: 13
>
```

It looks like we created two entities, each one remembers its owner and the value it had last time we looked at it. Every time you invoke such an entity it charges you, then reports the current total. We have created two objects, both of the same type: a counter. Racket provides[1] the ability to define classes, create objects, but we won't use that *per se*.

Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods. A feature of objects is that an object's procedures can access and often modify the data fields of the object with which they are associated (objects have a notion of `this` or `self`). There is significant diversity of OOP languages, but the most popular ones are class-based, meaning that objects are instances of classes, which typically also determine their type. In C212/A592 you will learn about such things as: classes and objects, encapsulation, composition, delegation, abstract classes and interfaces, the class extension mechanism, inheritance, polymorphism, dynamic method lookup, shadowing of variables, constructor chaining and a significant number of design patterns[2].

In class-based languages the classes are defined beforehand and the objects are instantiated based on the classes. If two objects `apple` and `orange` are instantiated from the class `Fruit`, they are inherently of the same type and it is guaranteed that you may handle them in the same way; e.g. a programmer can expect the existence of the same attributes such as `color` or `sugarContent` or `isRipe`.

In prototype-based languages[3] the objects are the primary entities. No classes even exist. The prototype of an object is just another object to which the object is linked. Every object has one prototype link (and only one). New objects can be created based on already existing objects chosen as their prototype. You may call two different objects apple and orange a fruit, if the object fruit exists, and both apple and orange have fruit as their prototype. The idea of the fruit class doesn't exist explicitely, but as the equivalence class of the objects sharing the same prototype. The attributes and methods of the prototype are delegated to all the objects of the equivalence class defined by this prototype. The attributes and methods owned individually by the object may not be shared by other objects of the same equivalence class; e.g. the attributes sugar content may be unexpectedly not present in apple. Only single inheritance can be implemented through the prototype.

From the link below: "While this [confusion] is often considered to be one of JavaScript's weaknesses, the prototypal inheritance model itself is, in fact, more powerful than the classic model. It is, for example, fairly trivial to build a classic model on top of a prototypal model."

---

[1] `http://www.ccs.neu.edu/home/matthias/Thoughts/Programming_with_Class_in_Racket.html`

[2] Factory, Singleton, Observer/Observable, Template Method, Strategy etc.

[3] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain`

Chapter 27

Commencement

Richard Matheson's *What Dreams May Come* might have been a better title[1] for this chapter. Below I chose to just include some quotes[2] that I consider important because I believe them to be either true, or funny, or both.

> "Syntactic sugar causes cancer of the semicolon" – Alan Perlis[3]

> "[Javascript] has a lot of stupid in it... The good parts of [Javascript] go back to Scheme and Self." – Brendan Eich, via[4] M. Butterick[5]

> "Lisp is still #1 for key algorithmic techniques such as recursion and condescension[6]." – Verity Stob[7]

> "That language is an instrument of human reason, and not merely a medium for the expression of thought, is a truth generally admitted." – George Boole, quoted[8] in Iverson's Turing Award Lecture

> "Programs must be written for people to read, and only incidentally for machines to execute." – Hal Abelson[9]

> "Be patient. Everything comes to you in the right moment[10]."

> "The mind is everything. What you think you become."

[1] But it would have been too long.
[2] For your enjoyment and/or reflection.
[3] Also see `https://www.cs.indiana.edu/~dswise/perlis.gif`
[4] `https://practicaltypography.com/why-racket-why-lisp.html`
[5] `https://practicaltypography.com/end-credits.html#bio`
[6] The part about condescension not entirely true, but definitely funny.
[7] `https://en.wikipedia.org/wiki/Verity_Stob`
[8] `http://amturing.acm.org/lectures.cfm`
[9] Reading Assignment: Blown to Bits, Addison Wesley Professional, June 16, 2008.
[10] This and the next by `https://en.wikipedia.org/wiki/Gautama_Buddha`

There's more[11] (also here[12]) but for now this would have to suffice. I would have also wanted to use a different title for this document[13], but in the end I decided that would have sounded way too pretentious. I also had a third possible title in mind but it seemed to me that it would have sounded too disrespectful[14] although if it had, it would have done so in a very specific Tim Minchin kind of way. If you want to listen to a song as we part ways you could try this[15] (to remind us of our journey together, to this very point) or this[16] (this always sounds good after the last lecture, last lab or the final exam).

Remember: a miracle is a shift in perception from fear to love. Success is getting what you want. Happiness is wanting what you get. There is no way to happiness—happiness *is* the way.


recipes and iterative refinement. The design recipes are a creation of the authors, and were they usable the use of the latter.

1. From Problem Analysis to Data Definitions

Identify the information that must be represented and how it is represented in the chosen programming language. Formulate data definitions and illustrate them with examples.

2. Signature, Purpose Statement, Header

State what kind of data the desired function consumes and produces. Formulate a concise answer to the question what the function computes. Define a stub that lives up to the signature.

3. Functional Examples

Work through examples that illustrate the function's purpose.

4. Function Template

Translate the data definitions into an outline of the function.

5. Function Definition

Fill in the gaps in the function template. Exploit the purpose statement and the examples.

6. Testing

Articulate the examples as tests and ensure that the function passes all. Doing so discovers mistakes. Tests also supplement examples in that they help others read and understand the definition when the need arises—and it will arise for any serious program.

Figure 1: The basic steps of a function design recipe

Design Recipes apply to both complete programs and individual functions. This book deals with just two recipes for complete programs: one for programs with a graphical user interface (GUI) and one for batch programs. In contrast, design recipes for functions come in a wide variety of flavors: for atomic forms of data such as numbers; for enumerations of different kinds of data; for data that compounds other data in a fixed manner; for finite

Figure 27.1: Education is what's left when you forget everything you learned in school. We hope this will become second nature to you and will become so good at it that you won't even remember where you learned it first. (At least I know I do.)

[11] `http://www.paulgraham.com/quotes.html`
[12] `http://www.azquotes.com/quotes/topics/lisp.html`
[13] To count the stars and call them all by name...
[14] Susie in a shoeshine shop.
[15] `https://www.youtube.com/watch?v=xAKZ4Ppiy94`
[16] `https://www.youtube.com/watch?v=6AEO21i-oNO`