Fields of Factor

A journey into unknown fields

by Jack Lucas

Contents

Chapter 1

In Which the Seed is Planted...

It is a typical drudgery that assails the front matter of most programming books. Factor, however, is not at typical language. Almost everything about it quite literally turns any common concept of how programming ought to be done into a matter of wondering which way is up. That is to say Factor turns some things backwards and other things forwards in a way not unlike a proverbial cheese grater where our mind becomes a substitute for the cheese.

As for drudgery, it would be a shame to litter the front page with the classic recollection of language history, reasons to bother, and grandiose overviews that leave you with more questions than you arrived with. Many view languages as each possessing a particular character that can positively impress on your mind, but if we're going to go with that analogy I think we'll find Factor to possess a depth it may take quite awhile to appreciate. So I think it best that in the spirit of Factor, we Factor! Here is our goal. The purpose of this book is threefold.

:

- We first propose this as a fun way of learning Factor through making games.
- We secondly propose this as a way of learning how to make games by using the raylib library. Since raylib is really just a graphics API abstraction we can safely have some fun in designing our own miniature game engine, thus learning about mak-

ing games from the ground up.

- We thirdly propose this as an introductory text on programming concepts common to almost all languages. ;

On our way through the Field we will view the following landmarks –

To begin with we will create two examples that are extremely simple, so we can try to get our bearings with Factor. In-between these examples we will start considering the basics of the game loop, and what components would make it easier for us to make games with.

We will then start implementing our so called *"entity system"* and interleave this with more examples. In other words we imagine a string of examples that gets progressively more impressive as we build our game engine iteratively. The idea of iterative design is an important concept that forms the core of this book.

And now with the drudgery fully out of the way, we take our first step into the silky sun-kissed landscape of Factor.

Chapter 2

And Watered...

The Field we intend to walk through does not come without a friendly guidepost laying out the trail. For this we will briefly go over the statement, *"Factor is a concatenative, stack-based language with higher level features."* in order to gain some simple insight into what we may expect as we venture further. After this, we will spend a brief amount of time making sure your Factor development environment is setup.

2.1 Factor is Concatenative

The first trait of factor we endeavour to understand is the nature of concatenation. Experience from other languages may tell you I'm about to talk about Strings, but that would not go far enough. The property of concatenation lies in the realm of functions, and would be easiest to explain by starting from familiar territory. We first consider the C programming language and how we might define a function 'Square'.

```
1    int square (int n){
2        return n * n ;
3    }
```

Armed with our simple function we might want the square of a square of a number, and begin with the most straightforward solution.

```
1    int main () {
2        int n = 3;
3        n = square(square(n));
```

```
        return 0;
    }
```

In C, a function like Square in the manner we have just used is utilizing functional application. That is, the application of SQUARE(N) to SQUARE() results in the desired double squaring. Concatenation views functions as a compositional effort rather than an applicational one. Let's see how we might do the same thing in Factor.

```
: square ( n -- n2 )
    dup * ;
```

In Factor functions are referred to as words. Defining a word is done by using the colon, followed by the name of the word. The parenthesis that come after define the stack effects, and the word body follows after ending with a semi-colon. Such that we might say a word is defined like...

```
:   NAME ( INPUT – OUTPUT )
        BODY                    ;
```

When we learn about the stack this definition will make more sense but for now lets engage in a trust fall exercise and pretend that square actually does what we expect it to do. To get the square of a square in Factor we would do the following.

```
3 square square
```

Which would result in an answer of 27. But you will say, how is this any different? The difference lies in understanding that we have the power to compose words into new ones. The definition of our double-square is clearly the result of combining two square functions into one.

```
: double-square ( n -- n2 )
    square square ;
```

That is the composition of two words results in a new word that defines the effect we need. In C we would not be able to compose, rather, we would still be stuck with defining application on our double-square function.

```
    int double_square (int n){
        return square(square(n));
    }
```

The important part to realize from this is that we are not necessarily dealing with arguments, but rather with functions themselves. In application the focus is on arguments, in concatenation our focus is on nothing but the composition of functions.

"But programs are more than just a SQUARE function!" a man holding up the rear of our tour will shout. And he'd be right! So why do we care about all this concatenative nonsense anyhow? It just so happens that concatenation is what gives us the ability to factor code well. We might consider a problem where we want a program to open and modify a file.

```
    int main(int argc, char** argv){
        FD file = open_file(argv[2])
        modify_file_formula(file);
    return 0;
    }
    void modify_file_formula(FD file){
        seek_file_line(file, DESIRED_LINE);
        for(int i = 0; i < DATA_LEN; i++){
            write(FILE, data[i]);
        }
    return;
    }
```

Though incomplete, the illustration here is to focus on what we're focusing on! In C we stand between this bridge where variables exist on one side and functions exist on the other. Our goal then is to make sure we apply our functions to our variables in order to get the result we want. Let's contrast this to a Factor example.

```
: modify-file-with ( file line data -- )
    open-file seek-file-line
    write-file-data close-file ;
! Macro version
: modify-file-with (file line data -- )
    [ seek-file-line write-file-data ]
    with-open-file ;
```

The first thing to note is that the program itself is a composition of func-

tions. Variables don't really exist in a sense, as it's just data laying around on the stack (we'll get to that!). Rather by taking words we may have had, or defined, and "stringing" them together we have created a general word that expresses what we want to achieve. This is the concatenative spirit. If we agree to the idea that we should factor out code that does a specific task then in turn we receive a collection (Vocabulary) of words that can then be combined to express new meaning. Smaller building blocks create more complex ones! This topic will be a central point of the text, and I don't expect this section to be a "WOW!" factor just yet. However, this methodology of explicitly abstracting code into simpler blocks of functionality that you can then combine in any way you see fit is a central benefit that Factor will impress upon you. Concatenative properties are the means which make this method extremely easy. For now we continue on to the next idea Factor has...

2.2 Factor is Stack-Based

If you have never been introduced to the concept of a stack; you have nothing to be afraid of. The Stack is a linear data structure which we can interact with through two primary methods. Namely, we PUSH or POP the stack. To push a stack is to place something new onto it, and to pop a stack is to (depending on the stack type) take the first value off the top. This is the essence of what is called a **FIFO** stack, or first-in first-out stack. We imagine the stack like an infinite staircase with steps large enough to hold a small table on each. Before we have done anything all the tables are empty, and we start at the bottom (not on a step) in possession of an empty stack. But now lets move upwards; if we type a number into Factor we can place something on the next table up.

```
2
```

We are now standing on the first step with a table that has the number 2. We can place any sort of thing on these tables, be it numbers, objects, strings, or even anonymous functions. If we want to grab the number again we can use the . word which will cause Factor to take an item off the stack and print it as output.

```
2 .
! . ( -- a )
! Output :> 2
! Stack is now empty
```

If our arms are long enough we can introduce some new words that let us manipulate what is on the nearby tables.

```
2 3
! We are standing on the 2nd step and the stack is
 ↪ now: 2 3 (current table has a 3 on it)
swap
! We grab the current thing off the table we're at
 ↪ and swap it with the thing from the previous
 ↪ table.
! ( a b -- b a )
! Stack: 3 2
```

Before we continue we should introduce what is called STACK NOTATION. As shown before stack notation looks like (INPUT – OUTPUT) where INPUT or OUTPUT can be any number of things we desire. For instance to describe a word that takes two numbers and will output a third we might say (a b – c). One matter of confusion we should clear up immediately is the order in which stack notation should be read. Stack notation for the INPUT side is read as the item closest to the – (the rightmost side) as being on-top of the stack. In other words we are describing the steps we'll walk up and which one we'll end on. The OUTPUT side works the same way, we describe the step we start on and which one we end at. Keep in mind that it doesn't matter at all that you keep track of *"which step you're on"*; the position is relative. All that matters is understanding which way is up.

```
2 3
! Stack is: 2 3
dup
! ( a -- a a )
! Make a copy of the current step you're on.
! Stack is now: 2 3 3
```

DUP, SWAP, and . are stack manipulation words. These all modify the stack in some way as shown by their stack notation. But you're probably wondering, if the stack is replacing variables (not completely the truth yet) then wouldn't it be annoying to have to modify the placement of the variables so that they are correctly used by the words we're calling? And you'd be right, it's super annoying. Luckily, Factor has higher order functions (Combinators) that will let us eliminate a lot of this inconvenience.

Let us consider some other uses of the stack

```
2 3 +
! + ( a b -- c )
! Stack is now: 5
! + has grabbed the object off our current step
 ↪ moved down a step.  And added the object we
 ↪ took to the object on the table below.

5 *
! * ( a b -- c )
! Stack is now: 25
! 5 was left over from our addition (we stayed on
 ↪ the step) and so we moved up to the next step
 ↪ and placed another 5 on the table. When * was
 ↪ called the same process as + occurred.
```

Later we will see that Factor provides dynamic and lexical variables to us if we choose to use them. However, and quite surprisingly, this method of using the stack (we'll get more in-depth in the next chapter) can adequately provide for most of our needs. In a philosophical sense, it also forces us to factor our code into smaller pieces. At most you probably want a word to consume around 3 items off the stack. Anymore than that and you will indubitably end up *"Stack shuffling"*, that is, trying to shuffle everything into place so you can use it. Factor forces us to learn to break our code up into bite size chunks and really focus on the core question we're trying to answer with each word. Gone are the days of writing massive functions that do everything, break it up! This mindset is invaluable no matter which programming language you choose to use. This will be hard at first, but as the mindset develops you will find yourself breaking up problems into smaller pieces that generally solve a problem. With those definitions in place the answer to the whole problem them becomes a trivial composition of simpler pieces.

For the moment, this is a brief taste of the stack. We will soon see how this sole data structure can be used to write any manner of program we desire. For now we consider that if bad programming is writing a function as a whole page, then good programming is writing it as a paragraph. Esoteric programming is writing it as a sentence (looking at you Perl!), and Factor programming asks you to express yourself in a single word. Factor may be a man of few words, but when he speaks, he says quite a lot.

2.3 Factor is Higher Order

As we begin our journey through the Field we witness a most peculiar property; there exists a beach. Somehow in the compression of cosmological space there exists a beachhead adoring a vast and immeasurable ocean. No one is sure how the beach exists, and many have tried to swim out to it's end. The oddness begins where our trail continues in that for one reason or another as one steps just a little farther through the Field the ocean disappears. All manner of those prone to idleness (philosophers) have attempted to solve the problem in vain.

To this day there is one living creature who inhabits the beachhead. He is an ancient sea tortoise of unbearably great wisdom; blessed with the noble rite of *"being too smart for his own good"*. Every morning the crabs of the sea make him a Kelp latte, and in the sudden burst of energy inspired from this, he aids them in understanding the best way to gather their shells for the day. Being quite old he never seems to remember their names and so a little crab remains on-hand to remind him. We call this crab the Caller, and he frees his people from the rather ignoble prospect of remaining anonymous. Til then they are only a bracket.

```
1 [ crab ]
```

Understandably, this tortoise takes all day to wake up, and at the last moment when all the crabs are ready he calls out their name with a command.

```
1 [ crab ] tortoise-command
```

You see the tortoise is rather annoyed from all the shells laying around on his beach; he has soft wrinkly feet after all! So what commands are there for him to say besides those that relate to the gathering of that nefarious sea litter? He doesn't seem to differentiate them either, but looks at the collection of shells as a whole.

```
1 { shell-1 shell-2 shell-3 } [ crab ]
   ↪ tortoise-command
```

Now the tortoise is aware that the crabs will use the shells if only he gets them near it, he must simply decide how.

```
{ shell-1 shell-2 } [ crab ] map
! The Crabs visit each shell and return something
 ↪ new!

{ shell-1 shell-2 } [ crab ] each
! The Crabs visit each shell and run back into the
 ↪ ocean without saying goodbye

{ shell-1 shell-2 } 0 [ crab ] reduce
! The Crabs visit each shell but take the result of
 ↪ the previous shell with them as a tool to use
 ↪ at the next shell.
```

Now before the Crabs unionize (and try to get some wages for their work) the Tortoise decides he better have an actual value for each of the shells. This can be anything, but he decides to number them for now.

```
{ 1 2 3 } [ 2 + ] map

! 1 [ 2 + ] caller-crab -> 1 2 + -> 3
! 2 [ 2 + ] caller-crab ->  2 2 + -> 4
! 3 [ 2 + ] caller-crab -> 3 2 + -> 5
! Results collected in a new sequence: { 3 4 5 }

{ 1 2 3 } [ 2 + ] each

! { 1 2 3 } -> { 3 4 5 } (Crabs run away)
! (No value is returned)

{ 1 2 3 } 0 [ + ] reduce

! { 1 2 3 } -> [ 1 0 + ] -> [ 1 2 + ] -> [ 3 3 + ]
 ↪ -> 6
! (Crabs return a single value)
```

We will make heavy use of higher-order functions throughout the book, but for now the Tortoise is tired. What we need to remember is that a higher order function is a lot like the cranky old Tortoise. The shells are whatever value we have viewed as a collection (or sequence). The Crabs are words that we wrap in brackets because we don't want to have the Caller Crab use them yet. And we, are the Tortoise. By wrapping what we want to do in brackets we create a QUOTATION and the higher order word we choose to use is responsible for determining how these quotations are used when

we are presented with a sequence. Such that higher order words are meta in a sense, we don't care about what is actually being done, but how we do it! We are using a Combinator to determine how a word is applied to a sequence. For now we consider this a rather intoxicating gulp of Kelp latte, but soon we will see how this simple concept can help us perform great wonders.

Chapter 3

Preemptive Stretching

At this point we have some vague notion of Factor's traits. We know it is Concatenative, Stack-based, and has Higher-Order words, but we haven't really explored the depths of what this means, nor can we see it quite clearly. We have a marble block that is slowly being chiseled into a Greek statue. This is also helpful to our understanding however. The way we are learning about Factor is an iterative process itself! And that is the essence of Factor. From unclear things, we chisel away until the simplicity of the problem is before us clearly. Before we commit ourselves to walking through this Field we should ensure that our shoelaces are tied properly.

3.1 Getting some Shoes

Shoes may be found at Factor's website where you can download a binary version of Factor. Factor provides a very nice development environment called "The Listener" which you are free to use for the purposes of this book. Since the listener is so straightforward, for the present time I will only recommend glancing over the reference for key shortcuts. Some other fancy shoes can be found for users of Emacs. The shoes for Emacs are called Fuel-mode and can be installed by running M-x *package-install* **fuel**. You will also want to add the following to your .emacs.

```
1 (setq fuel-listener-factor-binary
   ↪  "/home/*username*/factor/factor")
2 (setq fuel-listener-factor-image
   ↪  "/home/*username*/factor/factor.image")
```

After this you can simply run M-X *run-factor* and you'll be ready to continue on the journey!

3.2 Limber up

We'll assume that you now either have the listener or fuel-mode running on your screen. It's time we laid down some basics before we embark on our tour. The first thing we are greeted with by Factor is the prompt for the REPL which should look like this.

```
IN: scratchpad
```

Factor begins by telling us which Vocabulary we are in. The default Vocabulary is *scratchpad* which is an initial testing ground we can play around with. We remember that what we would call *functions* in other languages are called *words* in Factor. A Vocabulary is simply a collection of particular words. If we want to use words from another Vocabulary we simply type **USE:** and the name of the Vocabularly we want.

```
IN: scratchpad USE: rot13
Loading resource:extra/rot13/rot13.factor
```

This loads the *rot13* Vocabulary which gives us access to a few new words to play around with. Programming with Factor is a lot like a conversation, and what we nerds like to call the **REPL**. The REPL stands for Read-Evaluate-Print-Loop.

3.2.1 Read

In this stage Factor is waiting for input from us. It will then Read or Parse our input. We can input a great number of things into Factor. Lets look at some of the ways Factor can understand us.

```
2 3 4 4.5 10
! We can type numbers of any sort (int or float)
 ↪ and Factor will read them.

"Hello, Factor!"
! Strings are typed how you would expect.

{ 1 2 "Hello" }
! Sequences which you can think of as lists or
 ↪ arrays (for now) are typed with { } on either
 ↪ side.

+ / - * append
! We can also type words.
```

3.2.2 Evaluate

Factor takes our input and evaluates it. It will search our Vocabularies to see if what we've typed is a Word. If that fails it'll look to data literals like strings, numbers, or sequences.

3.2.3 Print

After reading and evaluating what we've input, Factor will then Print out it's results. Factor will also keep us informed of what's on the "infinite staircase" (or Stack). Let's see how that looks.

```
IN: scratchpad 1 2 3 ! enter

--- Data stack:
1
2
3
IN: scratchpad
```

Factor displays the top of the stack as the last thing we read. In this case, 3 is at the top of the stack (staircase). If we were to call + right now what would happen?

```
--- Data stack:
1
2
3
IN: scratchpad + ! enter

--- Data stack:
1
5
IN: scratchpad
```

If you guessed that + would take two items off the stack, add them together, and then put that result back on the stack (5); then you're right! Let's try calling -.

```
--- Data stack:
1
5
```

```
IN: scratchpad - ! enter

--- Data stack:
-4
IN: scratchpad
```

Once again two items are taken off the stack, used in some way, and then the result is placed on the top of the stack. You may wonder, does every word only take two items and place one? No, each word can declare how many items it will take and how many it'll give. This declaration is actually the notation we introduced early for **Stack checker notation**. Let's try defining a new word called **Super-Add** that'll show us how this works.

```
IN: scratchpad : Super-Add ( n n n -- x )
                 + + ;

! And again but expanded
IN: scratchpad

! We begin with a colon to say we want to define a
 ↪ new word.
: Super-Add
! The name of the new word.  This is case sensitive.

( n n n -- x )
! A stack effect declaration.  This one says that
 ↪ we are intending to take three items off the
 ↪ stack and place one item on the stack. Factor
 ↪ will check if our declaration matches up with
 ↪ the word's definition.

+ +
! Super-Add is simply the combination of two
 ↪ additions.

;
! We end our definition with the semi-colon
```

Factor now has a new word **Super-Add** that is entirely useless but helpful for our simple examples. Let's try calling Super-Add for the first time.

```
--- Data stack:
-4
IN: scratchpad Super-Add ! enter
Data stack underflow

Type :help for debugging help.

--- Data stack:
-4
```

Uh-oh! We tried calling our shiny new Super-Add word but Factor denied us the privilege of a successful execution. What's wrong? The problem is that our stack simply doesn't have enough data on it. If we were living in our staircase analogy, we just asked Factor if we can face-plant off the first step. The problem is, Factor cares. If a word is going to use more data than is currently on the Stack then Factor will notice this and tell us we've had a data underflow. The solution to this in a real program would be to figure out why your dataflow is incorrect, but for now we can simply put more items on the stack. Factor won't touch the data that is already there if we run into an underflow.

```
--- Data stack:
-4
IN: scratchpad 2 2 ! enter
--- Data stack:
-4
2
2
IN: scratchpad Super-Add ! enter
--- Data stack:
0
IN: scratchpad
```

Perfect! Our Super-Add word works and we've been left with quite literally nothing. Perhaps it's time we moved on to something a little more meaty? But before we begin we should call **clear**.

```
IN: scratchpad clear ! enter
```

This word is particularly useful when you're developing in the REPL as it will clear all data off the stack. With that we're ready to party like it's 23 A.D.!

3.3 Unto Caesar

The wheat stalks sway around us, reflecting the olive stained Field's comforting glow onto every wrinkle we've accrued over our lifetime of memories. We bask for a timeless moment in the recollection of distant optimism dancing in the fig trees nearby. Our new shoes embrace our legs like we've known them forever and yet, have only recently met. Their soles swipe on the dirt of the Field as if to say, *"go on"*. And in this intrepid air of sensible peace we suddenly hear words of profound dignity. Words which break the silence with an authority that says something like *"Hey you, carry my pack"* And so we listen...

> **Hey you, carry my pack!**

It's Quotaus! Roman Soldier. Precarious Praetorian. And most importantly, the guy who's hefty pack is now festooned on your shoulders. Quotaus beckons the group to follow him through the winding olive groves, and pulls out a shiny wine glass as if to celebrate a toast to your discomfort. His babbling soon becomes incoherent; drooping to a lowly level more befitting of a teenage girl.

> **How come Maximinus Thrax gets all the glory? It's not fair. Just because he's 8 feet tall and can wrestle 30 men at once. I bet he can't even write poetry like I can.**

Oh, Severus Alexander
Surely lost his candor
The giant leaped
And chased the wreath
Made slow the horses cantor

With a raised eyebrow you meet the sudden sobbing of the lousy legate. But it's no use trying to get out of the situation with his pack strapped onto you like a harness. Perhaps we can cheer him up instead? As luck would have it, a humble shop appears in the distance. There may be hope yet! We march nearer towards the abode as a sign that reads *Cicero's Cafe* displays itself outside. With a consolatory pat, we direct our crybaby centurion inside. Assorted in all arrays on the shelves within are numerous confectionery delights.

> **Ave Friend! Let me know what you want. Our menu is right over here.**

We look at the menu handed to us by the proprietor and see a lovely selection.

Praetorian Praline	$5
Chariot Crunch	$6
Maximinus Munch	$12

Surely a Praetorian Praline would do the trick! We reach for our wallet and take out–

> **Oh and by the way...the typesetter put all the numbers in Arabic, but I only accept Roman Denaraii. He's busy learning his lesson.**

The cawking outside alerts us to the presence of a horrified typesetter chained up and surrounded by a bunch of toga wearing chickens. They appear to be giving a stern discourse on the rights of chicken suffrage due to the stunning rhetoric of Diogenes. If we wish to absolve ourselves from this fate, we better pay with Roman Denaraii. Luckily, inside the pack that Quotaus gave us there is plenty of it. We better figure out how to convert Arabic to Roman numerals. Maybe we can make a new menu and save Quotaus from further emotional distraught.

3.3.1 Saving Quotaus

There are several ways to think of a how to solve a problem like this. Our first approach could be to try to clearly identify our current state, and the state we want to get to.[1] Our current state is that we possess a menu with names and values. The state we want to reach is a menu that has the same names but converted values. If we think back and remember the cranky tortoise on the beach we can remember that Factor has higher order operations. So perhaps, if we figure out how to change one value to what we want, then we can use those higher order operations to effect all the values in turn. In other words, we can simplify the problem down to a smaller problem. How do we change a line into the line we want? Maybe we should represent a line in Factor and see if that makes things clearer.

```
1 ! We'll put a line on the menu into a sequence
2 { "Praetorian Praline" 5 }
3 ! And we want to end up with
4 { "Praetorian Praline" "v denarii" }
```

[1] By the way, this will be the only non-graphical example we'll use. After this we'll be introduced to Raylib and start making some graphical programs!

It's clear from this then that when we possess a line, all we need to do is call a word on the 2nd item that can change our arabic numeral into a roman numeral string. Can we write such a word? We sure can.

```
IN: scratchpad 5 >roman
--- Data stack:
"v"
IN: scratchpad
```

This is our current goal then. We need to write a word **>roman** that takes one number off the stack and puts it's roman equivalent onto the stack. Let's fire up our REPL and create a new Vocabulary that can hold our definitions. Factor itself has a tool we can use for this.

```
IN: scratchpad USE: tools.scaffold
! load scaffold tools
IN: scratchpad "fof-roman" scaffold-work
! The name of the Vocabulary will be "fof-roman"
Creating scaffolding for P"
 ↪ resource:work/fof-roman/fof-roman.factor"
Loading resource:work/fof-roman/fof-roman.factor
```

This will cause Factor to create a new directory to hold our Vocabulary in the *"work/"* folder in the Factor directory. If we look at the new *fof-roman* directory we see a single file named *fof-roman.factor*.[2] We can now begin writing definitions in this file and load them in Factor.

```
IN: scratchpad USE: fof-roman
! load a Vocabulary
IN: scratchpad "fof-roman" reload
! Reload a Vocabulary. If you are using fuel mode
 ↪ you can hit C-c C-k in the file buffer
```

If we look in the fof-roman.factor file we can see that it contains the following.

[2]The original code for the number conversion is in basis/roman. The code was written by Doug Coleman and will be the only introductory code written by someone else. In a sense this is a tribute because the roman vocabulary helped me understand how powerful concatenative programming could be. Ave, Doug?

```
USING: ;
IN: fof-roman
```

These two lines are sort of like headers for our Vocabulary. The *USING:* line will be where we can say that we require words from other Vocabularies in order to use the Vocabulary we are defining. Factor's documentation is very good, so while you may not know what each of the Vocabularies we're about to ask for in *USING:* do; it is very good practice to go look up what Words they include. Most of the time it is very easy to see what a Vocabulary does. We'll begin with some simple Vocabularies. Factor is smart enough to tell us what we need to add to our *USING:* if we end up using a word from a Vocabulary we don't have in our file.

```
USING: unicode math.order kernel sequences
math strings grouping splitting.monotonic ;
! Don't freak out.
IN: fof-roman
```

There are times where we want to define Words that should not be usable simply by loading the Vocabulary. In other words, we want **private** words that are only internally defined.[3] For instance, we obviously want people using our Vocabulary to have access to words like **>roman** but that doesn't mean they need access to all the internal definitions we used to compose that word. Using **private** words helps Vocabularies stay cleaner, and presents a very clear interface that you expect other people to use meaningfully. Let's begin by defining some private words for ourselves. The first thing we could start with is creating a translations between the symbolic representation of roman numerals, and their values in arabic.

```
<PRIVATE
! until PRIVATE> is reached everything inside will
 ↪ be defined as a private word
CONSTANT: roman-digits
{ "m" "cm" "d" "cd" "c" "xc" "l" "xl" "x" "ix"  "v"
 ↪ "iv" "i" }
! We define a constant.  You could think of this
 ↪ like a word defined with the following stack
 ↪ notation ( -- n ) in that it takes nothing off
 ↪ the stack and places an constant value onto it.
CONSTANT: roman-values
```

[3] If you really want to use private words you still can though.

```
{ 1000 900 500 400 100 90 50 40 10 9 5 4 1 }
! For some words we do not need to put an ending
  ↪ semi-colon.  CONSTANT: is one of them.
PRIVATE>
```

Although we aren't sure how yet, we want to end up with a word **>roman** that might have this type of stack declaration.

```
: roman ( n -- str ) ! Take a int output a string
;
```

Let's think of how we might convert a number to roman numerals on it's own.

: We have 11.

Is 11 divisible by 10?

Yes!

Is there a roman numeral for 10?

Yes it's **x**.

How many times can we divide 10 from 11?

Once.

That gives us a remainder of 1. Is there a roman numeral for that?

Yes, it's **i**.

How many times can we divide 1 from 1?

Once.

So we divided the number by 10 once, and by 1 once. The values for that correspond to **x** for the number of times we divided by 10, and **i** for the number of times we divided by 1.

Right, so we're left with **xi** as the answer. ;

So it seems like if we start at the highest number (in this case 1000 for M) and ask how many times we can divide the Arabic numeral we have by it, we can also figure out how many times that symbol would appear in the roman numeral. Then we simply take the remainder and ask how many times we can divide with a lower number until we hit 1. We might imagine a simple example like this.

```
IN: scratchpad 11
--- Data stack:
11 ! Initial Value

IN: scratchpad 10
--- Data stack:
11
10 ! Divisor

IN: scratchpad /mod
! Divide the top number on the stack by the next
 ↪ number ( 11 / 10 ) and place the quotient and
 ↪ the remainder on the stack.
--- Data stack:
1 ! Quotient
1 ! Remainder

IN: scratchpad swap
! Swap the top value with the
! value below it
--- Data stack:
1 ! Remainder
1 ! Quotient is now on top of the stack.

IN: scratchpad "x"
--- Data stack:
1   ! Remainder
1   ! Quotient
"x" ! Equivalent to 10

IN: scratchpad <repetition>
! Take the top element on the stack and make a
 ↪ sequence with X repetitions decided by the 2nd
 ↪ element on the stack ( len elt -- repetition )
--- Data stack:
1 ! Remainder
```

```
T{ repetition f 1 "x" }  ! A sequence containing 1 repetition of "x"

IN: scratchpad concat
! Concatenate sequences together. In this case the repetitions are put together in a single string. We could get "xxx" for 3 repetitions.
--- Data stack:
1   ! Remainder
"x" ! Result

! Now all together for when the number is 1
IN: scratchpad swap
! Get the previous result out of the way.
IN: scratchpad 1 /mod swap "i"
<repetition> concat
--- Data stack:
"x"
0
"i"

IN: scratchpad nip
! nip removes the second element on the stack ( x y -- y ).
--- Data stack:
"x"
"i"
```

We have a little bit of a clearer picture of the process for conversion now. But if we remember the cranky tortoise perhaps we can think of a higher level to do this. What if we mapped over the *roman digits* and *roman-values* with the value we want to convert? Then we'd be able to divide, and pass that remainder to the next lower value. We already know the process for dividing a number, getting the number of repetitions it produces and turning that into a string. If we choose to map over the sequences maybe our definition of **>roman** would look something like this.

```
: >roman ( n -- str )
    roman-values
    roman-digits
    ! Put both sequences we defined on the stack and ontop of our "n" we want to convert
    [ fof-convert ]
```

```
6       ! Define our crabs as a word that takes a n,
          ↪ and a pair of elements from roman-digits
          ↪ and roman-values. We don't want to call it
          ↪ yet, so we wrap it in brackets to make a
          ↪ quotation.  A pair would look like 1000
          ↪ and "M" pushed to the stack at the time
          ↪ that word is called.
7       2map
8       ! The tortoise takes two sequences from the
          ↪ stack and tells the crabs to use their
          ↪ word on the pair of elements. If S{ }
          ↪ denotes a sequence then S{ "m" "x" "i" }
          ↪ and S{ 1000 10 1 } would mean "m" and 1000
          ↪ are on the stack when fof-convert is
          ↪ called, then "x" and 10 and so on.
9       ;
10
11      ! And without comments
12
13  : >roman ( n -- str )
14      roman-values roman-digits
15      [ fof-convert ] 2map ;
```

While we're trying to simplify, maybe we should think about what **fof-convert** should do?

```
1   : fof-convert
2       ( n r-digit r-value -- rem value-string )
3       [ /mod swap ] dip <repetition> concat ;
```

We've seen most of these words before, but what does **dip** do? **dip** is defined as *(x quot – x)* and what it does is very simple. Imagine we have a *Data stack* that looks like:

```
1   --- Data stack:
2   2
3   3
4   [ 1 + ]
```

If we were to call the quotation on top, **[1 +]**, then we would end up with 4, because the + would need two items off the data stack, and the only one it can grab is the 3 right beneath it. However, what if we want it to use the 2 instead? Perhaps, we actually want to *dip* under a value on the stack so

we can get to other values, and then restore that value we dipped under when we're done. This is what **dip** does.

```
--- Data stack:
2
3
[ 1 + ]

IN: scratchpad dip
--- Data stack:
3
3

! In essence what happened was...

--- Data stack:
2
[ 1 + ]

--- Data stack:
3

--- Data stack:
3
3 ! Original 3 brought back from the dead.
```

We can test out our new **fof-convert** word in the REPL.

```
IN: scratchpad 11 10 "x" fof-convert
--- Data stack
1   ! The remainder
"x" ! Our string for what was divisble by 10
```

Great! We get back a string representing the value that was converted into roman numerals and the remainder of that operation. This means each crab is going to return that value to the tortoise. Such that we could imagine... **REDO THIS TYPESETTING**

```
! Crab 10 gets:    11 10 "x"
! Crab 10 returns: 1 "x"
! Tortoise takes: "x" as return value
! Crabs 9 5 and 4 can't divide 1 so they return ""
! Tortoise hands 1 and "i" to the last crab.
```

```
! Crab 1 returns: 0 "i"
! Tortoise takes the "i" as the return value
! Tortoise is left with a sequence
{ "" ... "x" ... "i" }
! and his data stack looks like

--- Data stack:
0
{ "" ... "x" ... "i" }
```

Now all we have to do is get rid of the original value (0) and then concatenate all the values we have in our sequence together.

```
IN: scratchpad nip
--- Data stack:
{ "" ... "x" ... "i" }

IN: scratchpad "" concat-as
! to concatenate everything in the sequence into a
 ↪ certain type. In this case we provided "" so
 ↪ the result will be a string.
--- Data Stack:
"xi"
```

Let's look at what we should have in our *fof-roman.factor* file now.

```
USING: unicode math.order kernel sequences
math grouping splitting.monotonic strings ;
IN: fof-roman


<PRIVATE

CONSTANT: roman-digits
    { "m" "cm" "d" "cd" "c" "xc" "l" "xl" "x"  "ix"
     ↪ "v" "iv" "i" }

CONSTANT: roman-values
    { 1000 900 500 400 100 90 50 40 10 9 5 4 1 }

PRIVATE>

: fof-convert ( n r-digit r-value -- rem
 ↪ value-string )
```

```
    [ /mod swap ] dip
    <repetition> concat ;

: >roman ( n -- str )
    roman-values roman-digits
    [ fof-convert ]
    2map nip "" concat-as ;
```

The first step of our problem is done! Even Caesar would be jealous. But now we have to remember the overall problem we were trying to solve. We wanted to turn.

Praetorian Praline	$5
Chariot Crunch	$6
Maximinus Munch	$12

Into:

Praetorian Praline	v denarii
Chariot Crunch	vi denarii
Maximinus Munch	xii denarii

Let's start by putting a single line on the stack.

```
{ "Praetorian Praline" 5 }
```

Now we just need a word that applies our new **>roman** function to the second element and returns the result.[4] Let's try to define one.

```
: menuline-to-roman ( line -- line' )
    dup
    ! Duplicate the line on the stack ( x -- x x )
    1 swap
    ! We're going to change the sequence at index
     ↪ 1.  We need the number to be above the
     ↪ sequence before we call change-nth.
    [ >roman " denarii" append ]
    ! Quotation that will take our number from the
     ↪ sequence when change-nth is called, convert
     ↪ it to roman, and append " denarii" to it.
    change-nth
```

[4] This would be much cleaner by using objects instead of sequences. For now we keep it simple.

```
    ! Applies the quotation to an index in a
     ↪ sequence and changes the value with the
     ↪ result.  The index, the duplicate
     ↪ sequence, and the quotation will be
     ↪ consumed, leaving the changed sequence on
     ↪ the stack.  The change to the duplicate
     ↪ affects the original.

! Without comments
: menuline-to-roman ( line -- line' )
    dup 1 swap
    [ >roman " denarii" append ]
    change-nth ;

! { "Praetorian Praline" 5 } -> { " Praetorian
 ↪ Praline" "v denarii" }
```

At this point it's entirely trivial to convert the menu to roman numerals. Let's define our final word to solve this problem. All we have to do is map over the menu with our new word, **menu-to-roman**.

```
: menu-to-roman ( menu -- menu' )
    ! If the word modifies a stack item,  put a '
     ↪ on it.
    [ menuline-to-roman ] map ;
    ! Go through each menuline calling
     ↪ menuline-to-roman on each item.
```

Because we have access to higher order functions (or wisdom of the great Tortoise) we notice that almost immediately we could discard most of the problem. We didn't have to think about changing one menu into another. We didn't even have to focus that hard on how to change a single line from one to another. Once we had the power to solve the core problem (changing arabics into roman numerals) all we really had to do was wrap the rest of our program in higher order functions. This sort of compositional mindset is incredibly useful! At this point we should create a new constant to hold our menu.

```
CONSTANT: arabic-menu
{
    { "Praetorian Praline" 5 }
    { "Chariot Crunch" 6 }
```

```
    { "Maximinus Munch" 12 }
}
```

Make sure you have all the words we've just defined in the *fof-roman.factor* file and call *"fof-roman" reload* in the REPL. We should see if it works!

```
IN: scratchpad arabic-menu
--- Data stack:
{ ~array~ ~array~ ~array~ }

IN: scratchpad menu-to-roman
--- Data stack:
{ ~array~ ~array~ ~array~ }

IN: scratchpad . ! call the dot operator
{
    { "Praetorian Praline" "v denarii" }
    { "Chariot Crunch" "vi denarii" }
    { "Maximinus Munch" "xii denarii" }
}
IN: scratchpad
```

We present the new menu to the proprietor.

> **Whats this!? Shows what that good for nothing typesetter knew! I'm giving Praetorian Pralines to all of you. Please help yourself.**

You leap for joy tasting the delicious crunch of a rare roman Praline cookie. Nearby, you see Quotaus slightly cheering up but still distraught.

> **Ave, Quotaus was it? I saw you at a tavern recently. I think you were doing poetry?**
>
> **I-i was...I didn't think anyone would remember.**
>
> **Remember? I bought your scrolls. Far better to spend a day reading the great poet Quotaus than hanging around talking about that dumb Maximinus Thrax.**
>
> **But, isn't there a cookie here named after him?**

> **There was. But, maybe we could add a Quotaus Qurabiya to the menu? I'll go get that stupid typesetter again. Maybe he's learned his lesson.**

You all move to the window but the only thing out there is some dormant feathers puckered all over the ground. To this day no one knows exactly what happened to the typesetter. Some say, that after being convinced of the chicken's anti-platonic discord, he joined the Chicken-suffragette movement as a pamphlet designer until his subsequently expulsion on the grounds of *"being an egg-head"*.

3.4 Exercises

The Twostep

Write a word that takes two numbers off the stack, adds them, and places a roman numeral on the stack.

Caesar's Demise

It is also possible to write a word, **roman>** which converts a Roman numeral to an Arabic numeral. Think about how the process would look for doing this. If you think you can implement it, go for it! Should curiosity overtake you, the answer can be found in *extra/roman/roman.factor*. It is well worth reading the whole file, even if you can't figure everything out yet!

Chapter 4

Meet Ray

Your group continues on through new scenery that looks much like the previous scenery, except now there's *two of you.* Or at least, someone acting like they've reached the precipice of lateral thought might say something like that. Although, you might accept that annoyance. You might accept the attempted depth of a distracting observation. It might help drown out the choir of cascading sine waves some people refer to as a *Natural River*. Choirs are nice though. At least when they come to your house and fill your ears with melodic music you can have the decent pleasure of waving them off when they're done. But not with this river. No, this river is when the conductor decides to snap his baton and whip out a train horn.

You grasp your ears. It's just all too much. Too much to know, too much to do, and too little to hear.

Can you hear me? Can you hear me? Can you hear me?
Can you hear me? Can you hear me? Can you hear me?

Can you hear me?

Can you hear me? Can you hear me? Can you hear me?
Can you hear me? Can you hear me? Can you hear me?

You turn to see a French looking fellow not unlike a musketeer. Your eyes focus in on two blurs rapidly approaching your face.

SLAP!
SLAP!
SLAP!

Voilà mon pain pour la journée

He bows as if expecting a show of gratitude but your eyebrows betray you yet again.

> **Désolé monsieur! My native tongue makes offense to your patronage. I said, there goes my bread for the day.**

You feel your ears but your hands fill with an unfamiliar sense of texture. Bread? You consider the valid uses of a baguette until your brain realizes it'd rather have dignified hearing over any actual dignity.

> **Oui Monsieur. My domicile is nearby, and soundproof!**

Within the duration of a cloak's rustling you find yourself flicking bits of bread out of your ears and into the gold plated depository of a high class lounge. An aura of French pretensions and noble sensibilities rush in to perform triage on your aural canal. A wine glass rings out with a kingly vitality as if being worshipped by the presence of it's edible subjects.

> **You must be famished Monsieur. I regret to precipitate that we seem have run out of bread. We surely cannot eat without first breaking the baguette. But, mon ami, I would invite you to stay in the guest room tonight. We will dine tomorrow.**

You thank him for his generosity and inquire where the guest room might be.

> **Anglais drôle. It is where any respectable guestroom should be. Just a short swim over the river!**

You wait for the sarcasm to be indicated, but the man's face remains steadfast. Before you can even consider stealing a grape an army of butlers escort your group out through large wooden doors laced with the intrepid nature of an inconvenient host.

You find yourself near the river again. Silence adorns its surface and hums with the fresh light of a full moon. A shadowy figure in the guise of a simple townsman walks into view.

> **How many times do we have to tell Auburt to turn his speakers down? And river noises? Really? His front door is 10 steps from a river for goodness sake! The whole town is going to go deaf from this nonsense.**

Shunning reluctance you inquire for any answers from the man regarding the guest house.

> **The guest house? Ah, just a short step across the river it is. Although I wager you wouldn't get more than a few feet in those shoes. You have to walk on-top of the water after all. But I've cooked up a secret formula that can turn any shoe into a buoyant masterpiece.**

He hands you an indescribable thing. So indescribable that it is only describable through the babbling of loafing gamblers who have just lost their bobbles. So we describe it here.

```
1 USING: raylib.ffi
```

> **That ought to get you across. Just use that, oh, and also you're gonna need this for the frogs.**

```
1 USING: rot13
```

Before you can ask about the frogs the man has disappeared. With upgraded shoes you look at the waters surface. Your optimising is already lying in the bed on the other side. But what about the frogs?

4.1 Shiny New Shoes

Our shoes have been upgraded! Coincidentally, we've also finally reached the main topic this book is concentrated on. Making games! Before we begin, we should probably check out what this shoe upgrade actually is.

Raylib is a library set to make the process of creating a video game easier. It abstracts over OpenGL to give us easy drawing functions, provides audio processing, window and input handling, animations, and runs on almost any platform. Wasn't that a mouthful? I sure think so. Perhaps, we should stick to the Factor spirit and learn as we go. The journey we take will be firstly, iterative, as an attempt to build an appreciation for the full picture. This full picture process will help us appreciate the journey it took to get there, and will also help sharpen our design skills. Let's try to draw something on the screen!

4.1.1 Moonlit Crustacean

The moonlight betrays the figure of a highly appendaged creature dancing atop the surface of the water. For a moment, you fear the omen of perfidious frogs but the conclusion isn't all that ribbeting. A tentacle forms in the serene spectacle. The scene begs to be inscribed on a canvas, but during this thought it hits you, and were it not for the biting pain of being hit in the face you would wonder over the irony of the situation. The shadowy octopus leaps onto the shore near you, and the canvas from your recent recapitulation. You see the raiment of soft moonlight over his rubbery body and in a moment of awestruck serendipity you fail to question why this octopus is holding paint brushes. He gives you a look like the condescending gaze of a artistic coffeshop patron annoyed by your presence, and in the same breath holds up a brush to his mouth as if to say, *"Quiet!"*.

He turns his contemplation towards the thistle scraped landscape and in a furious rush of passionate inclinations waves his tendrils before the canvas as if the murder of a blank frame attacked his conscience with vitriolic intent. In a flash, the canvas stands imprinted with a moment in time now trapped for eternity. Adorned before you is a scene of crickets with reed fashioned leashes leading rabbits through dunes of sand like Arabian merchants. It is a true testament to the nocturnally creative instincts of an otherwise unknown artistic troupe, but just before you can steal the canvas away to the Louvre, the Octopus chucks it into the water.

A new canvas arrives on the ground before explanation for it's origins can explain themselves. In another whirlwind of appendages a new masterpiece bearing the testament of dusty Mercantilism lays before you, except the crickets have moved a step forward this time. The octopus continues unfettered; surrounded by a glow wrought from passion and the smell of slightly singed paint brushes. You watch as he creates, tosses his work away, and begins anew. For a moment you focus on the canvas' and begin to see the crickets with their rabbit steads come to life. They appear as if they're moving!

The Octopus continues in his furious pursuits and as you stare at his work it is almost as if the figures in his canvas have come to life. Within a few minutes the crickets have progressed quite a bit further than they originally were, and a few minutes after that they have completely walked off the canvas! The Octopus tilts his head as if to indicate superiority right as he callously makes one final stroke against a blank canvas. In one swing of his tentacle the consummation of passion and labor burst forth into a meritorious ignition. The paintbrush is on fire! He chucks the paintbrush

into your hands and within a moment it becomes the only memory of his existence. You gaze out onto the still waters of the river, truly inspired by his silent servitude towards the unfathomable beauty of nature. You think. *"Maybe I can do that?"*.

4.1.2 The Torch of the Octopus

Well we can do that! Armed with our new torch let's try to see if we can create a canvas.

```
IN: scratchpad USE: raylib.ffi
! Load raylib if you haven't already

IN: scratchpad 640 480 "Ode to Octopi"
! We're going to create a canvas.  First we have to
 ↪ put some data relating to what type of canvas
 ↪ we want on the stack.  In this case we want a
 ↪ 640x480 canvas with a title "Ode to Octopi"

IN: scratchpad init-window
! Create the canvas with raylib!
```

After you type this a new blank window should pop up on your screen. This is your new canvas! We can create anything we want with this canvas but we should also go through some drudgery to understand where this canvas comes from.

Firstly we learned earlier that Raylib handles windowing and input, but how does it do this? The answer to that is **GLFW**! GLFW is another library that specifically handles windowing and input. Windowing is essentially the ability for a program to use the host OS' graphical environment in order to register a new window on the screen. We also need a way to interface with the window and draw things on it. The standardized way to do this is to use a *graphics API* of which we might mention things like OpenGL, Vulkan, DirectX, etc. Raylib handles abstracting an OpenGL interface for us so we can draw in comfort.[1]

init-window is a function in the Raylib library that has been wrapped into being a word through the *Raylib.ffi* Vocabulary. We're going to be using a whole lot of Raylib functions but they will luckily look and act like words to us because of Factor's splendid FFI.

[1] This book is not about using OpenGL however. It's about using games to learn concatenative programming concepts from Factor. In this way, we use Raylib so we can hit the ground running quickly.

Before we make our game, perhaps we should try to get something drawn on the canvas? To tell Raylib that we would like to draw, we simply have to call **begin-drawing** and whenever we've finished we must end with **end-drawing**.

```
IN: scratchpad begin-drawing
IN: scratchpad RAYWHITE clear-background
! We want to clear the canvas  and set it to
 ↪  whatever color we want.

IN: scratchpad 320 240 20 RED draw-circle
! Factor should show the names of the arguments in
 ↪  fuel-mode.  In this case we need a x-pos,
 ↪  y-pos, radius, and color on the stack before
 ↪  we call draw-circle.

IN: scratchpad end-drawing
```

As soon as **end-drawing** is called our screen should update and contain a red circle near the middle of the screen. Using the *begin-drawing – end-drawing* scheme is much like asking raylib to give you a buffer you can write all your draws to. These draws will be made sequentially. So for instance if we wanted to put a green circle ontop of our red circle we could think of it like this...

```
IN: scratchpad begin-drawing
IN: scratchpad 320 240 10 GREEN draw-circle
IN: scratchpad 320 240 20 RED draw-circle
IN: scratchpad end-drawing
! If you do it this way you will not be able to see
 ↪  the green circle because the red circle is
 ↪  larger and gets drawn later.

! Correct version
IN: scratchpad begin-drawing
IN: scratchpad 320 240 20 RED draw-circle
IN: scratchpad 320 240 10 GREEN draw-circle
IN: scratchpad end-drawing
```

For now we try to get slightly acquainted with some of the words Raylib gives to us. We can't make a game without getting a feel for it after all! Raylib can also handle text, perhaps we should put a fitting epigram for our first venture into the world of graphics.

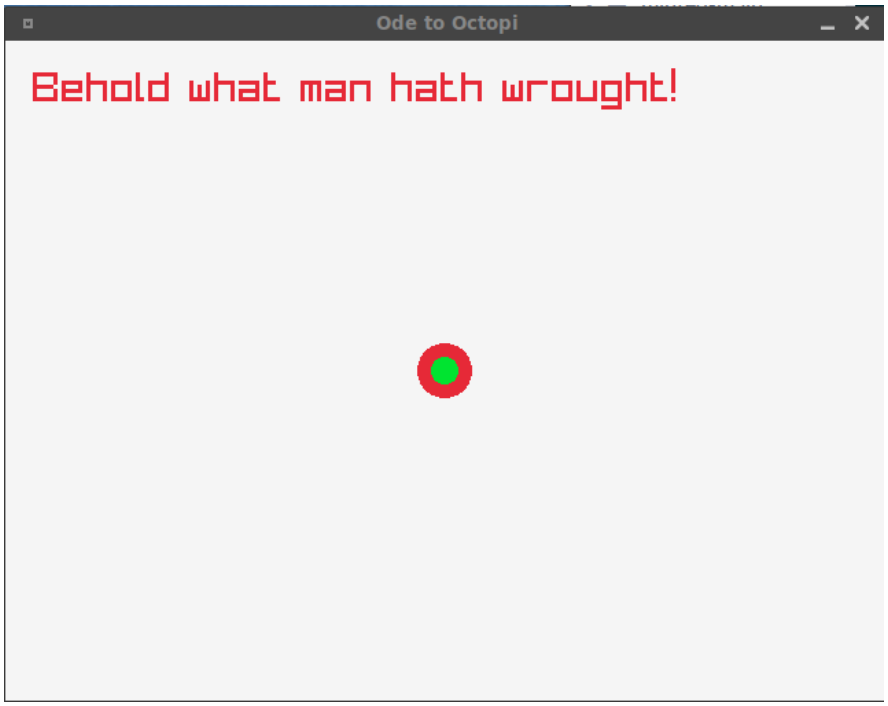

Figure 4.1: An Ode to Octopi

```
! Full version
IN: scratchpad USE: raylib.ffi
IN: scratchpad 640 480 "Ode to Octopi"
 ↪ init-window
IN: scratchpad begin-drawing
IN: scratchpad RAYWHITE clear-background
IN: scratchpad 320 240 10 GREEN draw-circle
IN: scratchpad 320 240 20 RED draw-circle
IN: scratchpad "Behold what man hath wrought!" 20
 ↪ 20 30 RED draw-text
IN: scratchpad end-drawing
```

Now that we have a seminal understanding of our new shoes; we should setup a new Vocabulary to hold our future game. If anything seems unexplained, rest assured, we will be going over everything more in depth later. We also will be thinking about how we can make the process more conve-

nient, what words we can define to do so, and what components we could make to facilitate building a game easier. In our introduction to this book we talked about an interleaving and iterative approach. That means our first game is going to look rather primitive code-wise, but as we learn more about Factor we will slowly build better and clearer versions of our games in order to get the most out of Factor and our time.

4.2 Crossing the Thames

Before we discuss the game design lets quickly setup a new Vocabulary to hold our game. If you remember that looks like this.

```
IN: scratchpad USE: tools.scaffold
IN: scratchpad "fof-thames" scaffold-work
```

This will create our *work/fof-thames* directory and we can once again store our work in *work/fof-thames/fof-thames.factor*. Til now we've lived in the *scratchpad* Vocabulary but it'd probably be better if we set our listener to live in the *fof-thames* Vocabulary. This way if we defined any new words in the listener, they will end up living in our new Vocabulary. We will still have to put the definition in our file if we want to save it for next time however. To set the current Vocabulary we use **IN:** like so.

```
IN: scratchpad IN: fof-thames
IN: fof-thames
```

Taking heed from our previous method of solving the Centurion problem, we should try to define what our current state is and the state we want to get to. Right now, we are only aware of a few Raylib words. We can draw text and circles to the screen, setup a window, and we have setup our new Vocabulary. To figure out what state we want to get to let's consider a two pronged approach.

One prong is that we want a Vocabulary filled with words that adequately describe whatever our game design is. We also want to be able to call a single word to jumpstart the whole program. This single word should handle setting up any variables, creating the initial game world, running the game loop, and dealing with input. Perhaps we'd want a word like this.

```
: run-thames-game ( -- )
    create-game-world
```

```
    run-render-and-input-loop
    close-window ;
```

We haven't introduced any conditionals yet like IF, CASE, or WHILE, nor have we seen any looping mechanisms like LOOP or regular recursion. We will add these to our mental Vocabulary shortly. However, this process of conjecturing over some helpful property we wish existed is called *Wishful Thinking* and it's an entirely helpful process in design. Programming is really about conjuring forth whatever we find most helpful with whatever tools we have available. Factor, just so happens to give us a plethora of useful tools (as we shall see). Let us define this prong then as the following. We want to end up with a file that looks like this.

```
USING: raylib.ffi and other vocabularies ;
IN: fof-thames

: create-game-world ( -- world )
; ! define

: run-render-and-input-loop ( world -- world )
; ! define

: run-thames-game ( -- )
    create-game-world
    run-render-and-input-loop
    close-window ;
    ! close-window is a Raylib word.
MAIN: run-thames-game
```

MAIN: is a word that defines which word will run first when the program is compiled. In Factor we can essentially throw a Vocabulary at it and ask it to give us a deployed version of that Vocabulary. This is much like compiling a C program in that we receive back a binary file we can then run. **MAIN:** is necessary to tell the compiler which word it should run when the program first starts. In this case we want *run-thames-game* to run first.

One other noteworthy topic is that Factor uses a single pass parser. This means that the parser starts reading words from start to finish and only in that order. So, if we're going to use a word we have to define it before it's used.

```
! Works!  eat-sushi is defined before it's used.
: eat-sushi ( -- )
    eat ;

: octopus ( -- )
    eat-sushi ;

! No good!  eat-sushi is used in octopus before
 ↪ it's defined.
: octopus ( -- )
    eat-sushi ;

: eat-sushi ( -- )
    eat ;

! This however, works.  We can use the DEFER: word
 ↪ to declare that we will define a word later
 ↪ but we want to use it in a word definition now.
 ↪ DEFER: ends up being really helpful if you
 ↪ want to do mutual recursion.  Consider a  task
 ↪ like this...
DEFER: eat-sushi

: octopus ( -- )
    eat-sushi ;

: eat-sushi ( -- )
    eat ;

! The definitions for go-fishing and eat-sushi are
 ↪ mutually recursive.  That is,  they depend on
 ↪ each other. We can define words like this in
 ↪ Factor by making use of DEFER:
DEFER: eat-sushi

: go-fishing ( -- )
    fish eat-sushi ;

: eat-sushi ( -- )
    eat go-fishing ;
```

The other prong of our game is the design itself. Or in other words, what the heck are we trying to make? Good question!

Crossing the Thames, Design	
Story & Gameplay	Notes
• The player must cross the Thames and reach King Frog. • The player walks on-top of the water dodging frogs as they surf down the stream. • Letters flow down the stream too which the player must collect. • The player is aware of how many letters he needs because of an indicator on the screen. • When the player reaches King Frog with all the letters, King Frog will ask for the password. • The player must use the *rot13* cipher to decrypt the the password. • When the password is accepted, King Frog lets the player pass and he wins the game.	• The player will use WASD to move across the water. • Frogs will move in different rows in a straight line. The row number will be determined by the difficulty. • We need a way for the player to collide with frogs and letters. If he collides with a frog he gets put back at the beginning. If he touches a letter it gets removed from the screen and added to the indicator. • For giving the password to King Frog the player should use Q and E to run the *rot13* cipher on the word.

Table 4.1: Design Sheet

run-thames-game	Main word that creates a game world and begins the render/input loop.
run-render-and-input-loop	Consumes a *world* and places a new world on the stack based on received input. Iterates over the *world* data to draw entities.
create-game-world	Takes a pre-defined initial state and returns a *world* ready to be run.
handle-input	Handles input by taking player data and modifying it according to whatever has been predefined.
handle-render	Takes a *world* and maps over the values therein. Automatically calls *begin-drawing* and *end-drawing* so all rendering is done in one pass.
check-collisions	Checks collisions between the player and anything else on the screen. Calls a word if a collision happens. The actual word called depends on what was collided with.

Table 4.2: Collection of Useful Words

Whew! What a multitude of words we have to implement! And this isn't even all the ones we're going to need. Oh bother! Just as we about to give up a poet who looks like he spent 6 weeks in a mental institution (as good poets do) barrels up to the shore. He sneaks a peek at the frogs floating down surrounded by magical letters. Without thought, (again as good poets do) he launches atop the water in a masterful weave of rhyme and kinestetics.

I would bathe myself in strangeness:
These comforts heaped upon me, smother me!
I burn, I scald so for the new,
New friends, new faces,
Places!
Oh to be out of this,
This that is all I wanted
- save the new.[2]

As the last note of his rhapsody falls from his lips he lands a final triumphant leap onto the opposite shore. A jumble of letters lie in his hand and within moments he has deciphered the password. He turns back to you, clearly noticing your difficulty.

> **Cheer up chap, nothing is all that insurmountable with the ideogrammatic method!**

In a final bow of genuine good-will the poet disappears into the tree line. Ideogrammatic? What's all that rubbish about?

Just as much as this book is about programming, it is also about inspiring philosophies of solving problems in your mind such that no matter what you end up using your mental Vocabulary will aid you through it. One such philosophy that underlies this book is the Ideogrammatic method. This method makes us consider ideas as their component sum. For instance, a red ball and a tree has nothing to do with the Sun, but we can use it to describe the Sun. A tree rises up, and a red ball looks hot. When these components are placed together a new picture is formed, and if we were to keep adding components eventually we would have a symbol that describes the Sun that was wrought from more primitive parts we already had access to. In the same way we want to make a game but we don't have a word that expresses our game. We need more primitive parts. Those parts seem to

[2] Ezra Pound, The Plunge. Inspiration for the Ideogrammatic method.

be rendering, input, and some sort of data structure. However, even these are foggy ideas. This is why we understandably spend the time trying to understand what our current state is. From our current state we reason into a more advanced state, or in other words, we form a more complicated ideogram from simpler components.

There are two important statements that come from this. The first is that we have to learn how to go backwards just as much as forwards. We must be able to break a single symbol into it's more primitive parts, and vice versa. For our current game, we have already done this to some degree.

The next question that comes up is, what do we work on first? It feels like everything needs to exist at once for the final idea to come into being. To some extent, that is where we want to get, but it does not define the journey in getting there. We can make do with using incomplete ideograms to make more complicated ones. Just as a lens starts off unfocused until it is adjusted, so too we can make primitive input, rendering, and game logic and slowly improve each until they come together into one final idea. That being said, why don't we start with input?

4.3 The Ancient Rite of Irrigation

Input in Raylib is really simple. We can ask if a key is being pressed, has been pressed, isn't pressed, or has been released.[3] Why don't we try to make a little game where we can move our red circle? If we build that ideogram, then when it comes time to make the game for this chapter we'll already have some idea of what we need to do when we define the *handle-input* word. You can keep all the words we define in our **fof-thames.factor** file if you'd like. We are going to be changing it's contents constantly!

First, we define some helper functions.

```
1 USING: raylib.ffi kernel sequences locals
    ↪ alien.enums ;
2 IN: fof-thames
3
4 : make-window ( -- )
5     640 480 "Irrigation" init-window
6     ! Create our window.
7     60 set-target-fps ;
8     ! Tell Raylib what fps we want.
9
```

[3] Raylib also supports game-pads, but for this book we'll stick with the good old mouse and keyboard.

```
: clear-window ( -- )
    RAYWHITE clear-background ;
! This'll be a cleaner way to clear our canvas.
```

Now we're going to need some way to store where the player is on the screen. The easiest way to do this seems like a global variable. Let's try that method first.

```
SYMBOL: player
! Creates a symbol named player
```

Symbols are a lot like global variables. We can put things in them and get things out. To do either of these we use **set** and **get**.

```
IN: fof-thames 2 player set
! Set the value of player to 2

IN: fof-thames player get
! Place the value of the symbol player on the
 ↪ stack.  In this case,  2.
```

That seems serviceable for our small example. But what are we going to store in the *player* symbol? What if we used vector coordinates? This is a 2D game after all. Luckily, Raylib has just what we need in the form of **Vector2**. Vector2 is a lot like the sequence:

```
! Template { x y }
{ 2 3 }
```

Except instead of being purely a sequence, it is actually a structure. The difference with a structure is that we have **named slots** that let us access exactly what we want to from that data type. In Factor we have **Vector2** defined like this.

```
STRUCT: Vector2
    { x float }
    { y float } ;

! Template
! STRUCT: name
!       { slot-name type }
```

```
!       ...
!       { slot-name type } ;
```

Later when we learn about **OOP** in Factor we will see the **STRUCT:**'s more powerful brother, the **TUPLE:**. In Factor, no matter which one you use to define your data type, Factor will define extra words for free for your convenience. The important one we want to know about now is the **slot-name»** and **slot-name«** words. Now those are just templates for this example, the actual names will be whatever the slot is named followed by « or ». What do these do? They're analogous to *set* and *get*.

```
! Assume we have a Vector2 struct on the stack.  If
 ↪ you have raylib.ffi loaded you can literally
 ↪ type the following line into Factor and it
 ↪ will create a Vector2 for you.
S{ Vector2 f 2.0 3.0 }
! Now we know that Vector2 has two slots, x and y.
 ↪ We can use the << and >> words to access and
 ↪ change the values.

IN: fof-thames dup x>>

--- Data stack:
S{ Vector2 f 2.0 3.0 }
2.0

! This gives us the x slot from the structure.  We
 ↪ use dup so the original structure doesn't go
 ↪ away.  Let's try changing the value of x.

IN: fof-thames 3 +
! Add 3 to 2.0 resulting in 5.0

IN: fof-thames [ dup ] dip
! Make a copy of the structure.  Dip under the 5.0.
--- Data stack:
S{ Vector2 f 2.0 3.0 }
S{ Vector2 f 2.0 3.0 }
5.0
```

```
IN: fof-thames swap x<<
! Change the value of x in our Vector2 structure to
  ↪ 5.0.  We swap because we need the value 5.0
  ↪ below our copy of the structure. ( val struct
  ↪ -- )

--- Data stack:
S{ Vector2 f 5.0 3.0 }
```

Typing in a Vector2 literally, however, is not so fun. Is there a better way? Absolutely, we can use **<struct-boa>** to easily create any structure. **boa** stands for *by order of arguments* and it simply takes the data we have on our stack and matches it up with the slot-names in our structure.

```
IN: fof-thames 2 3 Vector2 <struct-boa>
--- Data stack:
S{ Vector2 f 2.0 3.0 }
```

You can see the order of arguments in play from **boa** if you remember how we defined our *Vector2* struct. That is, the top value of the stack will be used for *x*, the next value for *y*, and so on for however many slots we need to fill. The only downside to a *boa* operation is that we need to ensure we have a value on the stack for all of the slots. When we get into **TUPLE:**'s later we will see more ways to make structures. For now, *boa* suffices.

Now that we have the power of vectors we can both, setup the player symbol when the game starts, and write a function to have the player display himself during the render loop.

```
: show-player-circle ( -- )
    player get ! Get the player symbol's value
    25.0 RED   ! 25.0 will be the radius of our
      ↪ circle and the color will be red.
    draw-circle-v ;
```

Here we have a word that we can call during the **rendering** loop that will handle drawing the player to the screen. It will get the coordinates from the *player* symbol and pass the actual *Vector2* structure to the word *draw-circle-v*. The *-v* in this case is the vector version of the Raylib circle drawing functions. It lets us pass the whole structure instead of individual *x* and *y*'s. Now to setup the player's initial coordinates when the game starts. How about putting him in the middle of the screen? Raylib gives

us two words for getting the screen size with **get-screen-width** and **get-screen-height**.

```
: setup-player ( -- )
    get-screen-width 2 / ! Half the width
    get-screen-height 2 / ! Half the height
    ! Width/2 and Height/2 on the stack
    Vector2 <struct-boa>
    ! Place it in a Vector2.
    player set ;
    ! Place the Vector2 in the player symbol.

! Uncommented version.
: setup-player ( -- )
    get-screen-width 2 /
    get-screen-height 2 /
    Vector2 <struct-boa>
    player set ;
```

Since we're only drawing one thing, the player, we might as well define the word for our render-loop now. As we're dealing with a really small number of things to render we can use the simplest definition possible. However, we will have to improve this for our later games!

```
: render ( -- )
    begin-drawing
    clear-window
    show-player-circle
    end-drawing ;
```

Now we're only missing the input portion of our game, and a main word to tie everything together. For the input, let's think about what we really need to do.

: We want to get input from the keyboard and have it let the player move.

That sounds like multiple problems. First we need to define what input from the keyboard looks like. What is a key?

A key is a value in an *enumeration*, or in other words a

key has a numeric value attached to it.

How can we get a keys value?

In Raylib we type *KEY_* followed by the key we want. So 1 would be *KEY_ONE*, W would be *KEY_W* and so on. To get the keys value we use the word *enum>number*

Ok. We have the keys value now how do we know if it's being pressed?

We would use the word *is-key-pressed* with the key's numeric value on top of the stack.

Good! And what does this return?

It returns a *Boolean* value *true* or *false*. In Factor that would be *t* or *f*.

Then what do we need to do?

Well, if one of the key's we are watching returns *t* then we need to perform an action. In this game if WASD is pressed then we need to move the player.

What does movement mean in this case?

Movement is a change of coordinates. So if a movement key is pressed all we have to do is update the coordinates by the speed of the player. After that, the render loop will draw it at it's new location. This will look animated because we are rendering new canvas' so quickly.

Perfect! Let's do that.

Broken down our process in processing input is composed of these primitive elements. We can use something like the following in Factor to see if a key is being pressed.

```
KEY_W enum>number is-key-down
```

But we also need to do something if this is true. Namely, we need to update the coordinates for the player data. What we need is a conditional! We can decide what to do based on if something is true or false using the **if** word.

```
--- Data Stack:
potatoes

have-potatoes? [ eat-potatoes ] [ cry ] if
```

It may be a little strange at first that the **if** comes last, but it's really no different than everything else we've been doing. In this case **have-potatoes?** will take an item off the stack and decide if it's a potato or not. If it is true it'll place a **t** on the stack. Then see what happens...

```
--- Data stack:
t
[ eat-potatoes ]
[ cry ]

IN: fof-thames if
! if's stack notation is ( ? true false -- )
```

Because we have a **t** the first quotation will be called. (Remember that anything in brackets is called a **quotation**). If **have-potatoes?** had returned **f** or false then we would be suffering from prolonged fasting and **cry**.

We can now model how we could perform our input requirement. We check if the key is being pressed. If the key is not being pressed we simply do nothing, so our false statement will be **[ ]**, an empty quotation.

```
KEY_W  enum>number is-key-down
[ update-player ] [ ] if
```

Without implementing **update-player** yet, we have essentially solved the problem. However, there is one big problem with this situation, we would be repeating ourselves constantly for every different key we want to press. Let's try to factor out the bits that don't have anything to do with a specific key.

```
enum>number is-key-down
[ update-action ] [ ] if
```

We could even define this as a word.

```
: input-check ( key -- )
    enum>number is-key-down
    [ update-action ] [ ] if
```

Now we can make use of higher-order words to check whatever keys we want. What if threw all the keys we wanted to check into a sequence?

```
{ KEY_W KEY_A KEY_S KEY_D } [ input-check ] each
! Remember that each doesn't want a return value
```

Well this seems quite nice, but we're missing one crucial ingredient. How are we going to define what action should take place for each key? If we press **W** we want the red circle to move upwards, but for **S** we want it to move down.

It seems like we need to somehow marry the key and it's action together. What if we did a sequence of sequences? Maybe something like...

```
{
    { KEY ACTION }
    { KEY ACTION }
}
```

That should work. We can figure out how we'll map over that later. Let's figure out what the action should be. Remember, we are looking to have some word that updates the players coordinates. The simplest way to do this would be to add two new words to our Vocabulary. Each of them will take an increment and change the player's coordinates on a certain axis.

```
: move-x ( increment -- )
    player get x>> +
    ! Get the current x value and add the increment
     ↪ to it
    player get x<< ;
    ! Set the new value in the player symbol
```

```
6
7  : move-y (increment -- )
8      player get y>> +
9      player get y<< ;
```

You're probably thinking, 'Hey you're repeating yourself too!'. And yeah, we could Factor this even more. When we get more advanced and comfortable we will try to be more clever about it, for now these simple definitions will do.

Now we can define our sequence as such.

```
1  {
2      { KEY_W [ 2.0 move-x  ] }
3      { KEY_S [ -2.0 move-x ] }
4      { KEY_A [ -2.0 move-y ] }
5      { KEY_D [ 2.0 move-y  ] }
6  }
```

We can now write a word to handle our input. Let's see what we can do.

```
1  : check-individual-action ( keypair -- )
2      dup first ! Get the key
3      enum>number is-key-down
4      [ second call( -- ) ]
5      [ drop ] if ;
6
7  : check-input ( -- )
8  {
9      { KEY_W [ -2.0 move-y  ] }
10     { KEY_S [  2.0 move-y  ] }
11     { KEY_A [  2.0 move-x  ] }
12     { KEY_D [ -2.0 move-x  ] }
13 }
14 ! We could also put this in a symbol if we wanted.
15
16 [ check-individual-action ] each ;
```

Because it's a little complicated we should take a closer look at **check-individual-action**. It is helpful to remember that it is looking for a *{ KEY ACTION }* pair on the stack when it is called.

```
--- Data stack:
{ KEY_W [ -2.0 move-y ] }

IN: fof-thames dup first

--- Data stack:
{ KEY_W [ -2.0 move-y ] }
KEY_W

IN: fof-thames enum>number is-key-down

! We're going to take the true path so lets
  ↪ simulate that without the if.

--- Data stack:
{ KEY_W [ -2.0 move-y ] }

IN: fof-thames second

--- Data stack:
[ -2.0 move-y ]

IN: fof-thames call( -- )
```

The reason for **call(** is very primitive. We're going to call a *quotation* at run-time and Factor needs to know what the stack effect is going to be. **run-time** is when the program is actual running so the compiler will not have a chance to check if the stack effect will keep the stack consistent. Factor, being the nice guy he is, doesn't want us to *blow the stack* or *stack underflow*. Blowing the stack, by the way, is when we keep putting data on that never gets used and our computer ends up running out of memory. In other words, nothing too complicated is going on, we're simply declaring what our quotation is going to do at run-time. In this case it does nothing so we put **(–)**.

It dawns on us that we've forgot one last thing. We've gotta write a **main** word to tie this all together!

```
: main ( -- )
    make-window setup-player
        [ check-input render
        window-should-close not ]
    loop
```

```
6       close-window ;
```

window-should-close is a Raylib word that will check if a condition has occurred that is asking the window to close. By default this is, hitting the *x* in the corner of the window, or hitting escape.

Also new, is **loop** which takes a quotation and keeps running it indefinitely as long as the last value it returns is **t**. This occurs when we call **window-should-close not**. Raylib will return **f** if nothing has happened to signal a window close. We then call **not** to flip the value to true. **loop** will receive the true value and call the entire quotation again from the beginning. This is the early form of our *gameloop*!

Finally, putting everything together we should end up with the following in our **fof-thames.factor** file.

```
1  USING: raylib.ffi kernel sequences locals
    ↪ alien.enums namespaces math
    ↪ classes.struct accessors combinators ;
2  IN: fof-thames
3
4  : make-window ( -- )
5      640 480 "Irrigation" init-window
6      60 set-target-fps ;
7
8  : clear-window ( -- )
9      RAYWHITE clear-background ;
10
11 SYMBOL: player
12
13 : show-player-circle ( -- )
14     player get 25.0 RED draw-circle-v ;
15
16 : setup-player ( -- )
17     get-screen-width 2 /
18     get-screen-height 2 /
19     Vector2 <struct-boa>
20     player set ;
21
22 : render ( -- )
23     begin-drawing
24     clear-window
25     show-player-circle
26     end-drawing ;
```

```
27
28 : move-x ( increment -- )
29     player get x>> + player get x<< ;
30
31 : move-y ( increment -- )
32     player get y>> + player get y<< ;
33
34 : check-individual-action ( keypair -- )
35     dup first enum>number is-key-down
36         [ second call( -- ) ]
37         [ drop ]
38     if ;
39
40 : check-input ( -- )
41     {
42         { KEY_W [ -2.0 move-y  ] }
43         { KEY_S [ 2.0  move-y  ] }
44         { KEY_A [ -2.0 move-x  ] }
45         { KEY_D [ 2.0  move-x  ] }
46     }
47     [ check-individual-action ] each ;
48
49 : main ( -- )
50     make-window setup-player
51         [ check-input render
52         window-should-close not ]
53     loop
54     close-window ;
55
56 MAIN: main
```

You can now pat yourself on the back and call **main** from the Factor listener. A window should pop up that lets us move a red circle around the screen. On a side note, one benefit of the way we handled input is that we can handle multiple keys being pressed at once. This means we get diagonal movement for free!

We take a breather and settle down on the shore. The moon lies timelessly in a gaze above us, and for a second we consider taking a nap amongst the lush grass near our feet. That is, until a warning rings out in the night.

> **Hey buddy! You better hurry up. Auburt is getting up early today, and trust me when I say, his music is nothing compared to his alarm clock.**

4.4 Exercises

To irrigate, concatenate!

You may wonder why that section was called, The ancient rite of irrigation. That's because you could effectively make a little program that lets you draw crude water canals on a primitive field. You can do this by changing only two words. One of these words is the color of the background. Factor's bindings for Raylib contain most of the common colors; usable by typing the name of the color in all caps. The other aspect is making the player circle leave a trail. Think about the Octopus. He throws away his work and starts with a new canvas each time, much like we just did. So what would you have to change about a painting to create a trail? What would you physically be doing if you worked on only one canvas? How can we emulate this in our example program? You only have to move one word, and change the color of another.

Chapter 5

The East London Poet Society

Uncommonly known, but tucked into the corroding relics of ancient Rome, is the proof of a distinct underclass powered by a lyrical witticism married to insanity. And in these fine annals of history we find only whispers that laid quite dormant til the 20th century. Propagating the chaotic intent of raw imagination was the vagrant spirit of an inverted conscience fighting against a mass of stone pamphlets we refer to as **Respectable Sensibilities**. It was quite nobly said that the validity of your work was not sealed with the approval of future nods:

Unless you were locked up for being insane.

Yesteryear called them all manner of inane maladies for threatening to breach the immutable and intangible existence of, **The Social Contract**. Today we call them by a simpler nomenclature of which can only mask an insufferable revolutionary motive.

As such, we ascribe to them the name,

Poet.

At least those are the kind of delirious thoughts that occasionally accompany the feeble soul in his journey towards an unplanned nap. That however, is what has just happened to you, as you plopped down into blissful unconsciousness in a grass bed near the shore. Oh Dear! How are you ever going get across the river in time now? The answer seems out of reach, as if locked in a footnote somewhere.[1]

[1] You're probably wondering why the heck we gave a little game design for the Cross the Thames game and now seem to be moving on. We're going to make use of the ideogram-

You slip into the crevices of your mind containing the motion picture department, and begin to dream.

Oi! You bloke. Where's your poetry liocense?

Silence me but you'll never silence England, copper!

Oh Great England, Free the speech!
Inkwells spare not chance to reach
Respond! oh pen; abandon brawn

Oh, I think I know how this one ends.

Apply to face, police baton!

WACK!

You witness the brutal tyranny as the face of a nefarious poet rings out in a baton induced drum solo. You stand considering the validity of street justice as your eyes gaze over the quaint surroundings of a London street. Trees drift in the wind with a pretension that spells indifference to the violence. Across the street you see a towering building that looks like the shambling attempt of the physically inept in creating a hidden fortress. You look down at your hands to see a map with a large X on it. The moment hits you that your destination is in exactly the same spot as that unkempt fortress. But that can't be right, it says on the map that it's a, **Poet's United** meeting spot. The clothing around you reeks of unusual fiber. You witness yourself and in a moment realize the preposterous idea that you are dressed like someone who hasn't seen the same bed in 10 years. Near the still wet stains of a now late latte you spot a name tag stitched to your lapel.

Sir George Cann Stanza

It would not be entirely unlikely, but also not altogether imprudent to assume that you must be a member of the poets. You witness the gangs of altruistically challenged flatfoots prowling with their weapons of choice, the fearful baton. Perhaps, poet or not, it would be best to make a dash for the fortress before you. But how with all the guards around?

matic method to make a simpler version of the Thames game, the Poet game. When we awake from our nap the Thames game will only be a matter of combining all the things we've learned into a straight-forward solution. Solve the simple core problem first!

5.1 Mad Dash til Balderdash

We set out with a goal to make the Crossing the Thames game but quickly realized we were inadequately prepared. It makes sense then that we should try to cross a street before crossing a river. Earlier, we saw how to do input in Raylib and Factor, but it would be wise now to see what we need to know in order to make our new games. There are namely 6 components to our game that we need to implement. With these in place it will simply be a matter of tying our components together to form a game. Our components then are:

- Input handling
- A grid system for movement
- Rendering
- Basic AI and NPC entities
- States (Start, Restart, End)
- Collisions between entities

Let's start with our grid and build up from there. Our game will consist of a static screen, and all the player has to do is move to the other side of the screen while avoiding the enemies. When the player does reach the other side we'll increase the difficulty (add more enemies) and place the player at the bottom of the new screen. The player will continue until he reaches the final door and makes his escape. Customarily we'll start by setting up our vocabulary.

```
IN: scratchpad USE: tools.scaffold
IN: scratchpad "fof-poet" scaffold-work
IN: scratchpad USE: fof-poet
IN: fof-poet
```

5.1.1 The Grid

Where's this grid? Coming out of nowhere like it owns the place? Well, it does own the place. The grid will be the backbone for the organization of our game. Eventually, it'd be nice to have a *world* view of our game; or a way we can view the state of our game through a data structure. Currently, this role will be fulfilled by the grid. We begin as we always do by

recognizing our current state and the one we want to get to. Right now, we have nothing. We'd like to get to a place where we have some words that describe the creation of a grid. Let's use some of that **wishful thinking** and imagine what our grid will look like first.

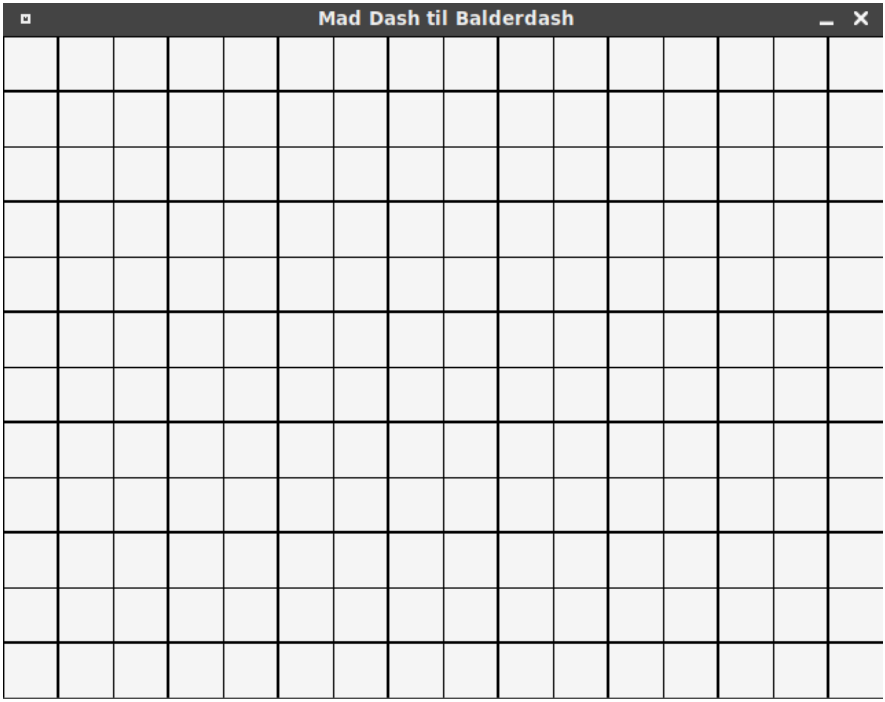

Figure 5.1: The Grid

Looks pretty nifty! But why do we want this? One good reason is that we want to handle player movement in a uniform manner. So instead of having the player move around at a certain speed we simply have him move to a new square. If the player can only move around on squares then we also don't have to worry about him walking off the screen. Furthermore, we can setup certain squares to contain different things. Perhaps, we want a certain square to trigger a *win* if the player steps on it. We also need to model the guards to walk around and possibly *'collide'* with the player. In this case, all we have to check is if the player and the guard are currently on the same square. Our grid will be the simplest way to implement all of these things in our first bigger Factor game.

That's some great wishful thinking, you say, but get your head out of

the clouds! How're we going to implement this? Let's start by simplifying the problem. What can we realize from our dreams to create reality? The first thing that strikes out obviously is the fact that each grid is really just a collection of squares. A collection? That sounds like a sequence; maybe we should keep them all in one?

```
{ square1 square2 ... squareN }
```

Because we're opting to keep them in a sequence this way, we are also guaranteeing ourselves that we'll be able to work all that higher-order goodness on our grid. We'll be able to use *map, reduce,* and *each* to interact with our grid. With that in mind, we can break the problem up into a question that haunts the halls of kindergarten classrooms.

What is a Square?

Difficult questions like this often lead to existential dread or pragmatism. A whole book could be written on the depth of this one innocent inquisitive. ~~And so we revert our original direction and begin our thesis on the substance and form of the rectangular phantom known as the squa...~~

Or! We could take a moment to look at where we're going. Perhaps, we are brave enough to admit that we know all there is to know about this homogeneous fiend. How would we even draw one? Raylib provides an answer for us. **draw-rectangle-lines** is a word we can use to draw a square on the screen. Here is how it is used.

```
IN: fof-poet 0 0 50 50 BLACK

--- Data stack:
0   ! X coordinate
0   ! Y coordinate
50  ! Width
50  ! Height
S{ Color f 0 0 0 255 }  ! Color

IN: fof-poet draw-rectangle-lines
```

All **draw-rectangle-lines** needs from us on the stack is a position for the *x and y coordinates*, a *width and height*, and a *raylib color*. There's a few things that can remain the same between all the squares in our grid. Color is one since we want all the lines to be the same color. Another is height and width since every square should be the same size. That just leaves us

with an *x and y position.* All of our squares can be modeled this way. From the previous section we remember mentioning a word called **TUPLE:** that was much like **STRUCT:**. Now seems as good a time as ever to use it. We can define our square tuple like this.

```
TUPLE: square posx posy ;
```

square is the name of our tuple class. We also have slots named **posx** and **posy** that we can store coordinates in. Tuples form the backbone of the *OOP* system in Factor that we'll be learning about. The cool part is that we can use a tuple as a sort of structure to hold our data without having to really do any *OOP* at all. Factor splits up the object system into a couple components we can then combine together any way we see fit, but that's for a later chapter.

For now, we have achieved the prospect of having a square type. But our definition isn't actually complete. How are we going to know if something is on the square? We should add a slot that will let us store the data of what is on the square so we can use it later.

```
! Complete square definition
TUPLE: square posx posy container ;
```

Any grid worth it's salt is uniform. To do this we need all the squares to agree to have a certain *width* and *height.* We can create a symbol and a word to set this up for us.

```
! Symbol to hold our width and height.  Since width
 ↪  = height in a square we only need one number.
SYMBOL: grid-size

! Set the grid size to be squares with 40 pixels of
 ↪  width and height.  We can change this value
 ↪  here to make our grid different sizes.
: setup-grid ( -- )
    40 grid-size set ;
```

Perfect! However, we still don't have a *sequence* that contains all our squares. Maybe we should find a way to bring them into existence? How would we go about doing that?

:

We want a grid composed of squares. What is a square?

A square is an *x and y coordinate* with a height and width determined by the grid size.

Do we know anything else?

We know that our grid should fill the screen. And we also know the size of this screen. We can use **get-screen-width** and **get-screen-height** to see what that is.

What if we had a screen size of 640x480 and a grid size of 40. Where would our first square go?

Our first square would be at (0, 0).

And the last?

Our last square would be at (640 - gridsize, 480 - gridsize).

What if our **x** coordinate never changed? What squares would we have?

Our squares would change their **y-coordinate** by the *grid-size*. So we would have squares at (0, 0), (0, 40), (0, 80) ... (0, 440).

What if we increased **x** by it's grid-size?

We'd get (40, 0), (40, 40), (40, 80) ... (40, 440).

When would our **x-coordinate** go off the screen?

When it exceeds the max width of 640.

How do we fill the screen with squares then?

We know that we will only visit points in increments of the grid-size. We could have a **range** that expresses the increments **x** will visit like (0, 40, ... 640) and another range that has **y** like (0, 40, ... 480). We should pick a list to start with and then iterate through the other one til we return a sequence.

Let's try it.

;

The core of our problem has to deal with **ranges**. We can create a range in Factor by using the **<range>** word. Let's look at some quick uses of **<range>**.

```
IN: fof-poet USE: math.ranges
IN: fof-poet 0 10 1
--- Data stack:
0   ! Starting Point
10  ! Ending Point
1   ! Step

IN: fof-poet <range> dup ! Make a copy for more
 ↪ examples

--- Data stack:
T{ range f 0 11 1 }
T{ range f 0 11 1 }

IN: fof-poet [ ] map
! [ ] is an empty quotation,  so this map will
 ↪ simply return the entire sequence unmodified.

--- Data stack:
T{ range f 0 11 1 }
{ 0 1 2 3 4 5 6 7 8 9 10 }

IN: fof-poet drop ! Get rid of the sequence
IN: fof-poet [ dup * ] map ! Square the entire
 ↪ range
--- Data stack:
{ 0 1 4 9 16 25 36 49 64 81 100 }
```

Likewise we can also create a range to express all the points we need to hit in our xy coordinates. Let's try defining a new word to generate all our points.

```
: setup-grid-squares ( -- )
    0 get-screen-width grid-size get <range>
    0 get-screen-height grid-size get <range> ;
```

This definition won't compile but it does give us a good starting point. Both of our axis start at 0 and end at their maximum determined by the current screen size. They also *step* over an amount determined by the grid

size. In other words our ranges would both look like this if they were a sequence for our current grid-size of 40.

```
{ 0 40 80 ... 640 }
{ 0 40 80 ... 480 }
```

Essentially what we want to do is have one range be the outer range and another the inner range. This is purely arbitrary, and it doesn't matter if it's x or y. Imagine we have x as the outer range. This means we want to pick one value from x and marry it to **all** the values in y. We should update our word with this idea.

```
: setup-grid-squares ( -- )
    0 get screen-width grid-size get <range>
    [ 0 get-screen-height grid-size get <range> ]
    map ;
```

Our next problem is to consider what we actually do with these values. We have our higher level structure but we haven't defined what word (or crab) is going to be driving the actual solution.

```
: setup-grid-squares ( -- )
    range-1
    [ range-2 make-grid-line ]
    map ;
```

The problem shifts to this, what happens when we are given a single value from one range (0, 40, or 80, etc) and an entire range? How do we marry all the proper values together and turn them into squares? This actually turns out to be fairly easy to do! I'm going to show the whole thing and we'll work out some of the trickier bits.

```
: make-grid-line ( x range -- list )
    [ length ] keep swap
    [ swap <repetition> ] dip
    zip
    [ [ 40 H{ } square boa ] with-datastack ] map ;

! Commented

: make-grid-line ( x range -- list )
    [ length ] keep
```

```
11      ! We want the length of the range here,  we use
         ↪ keep to retain the original range.  Stack
         ↪ will look like ( x range len -- )
12
13      [ swap <repetition>] dip
14      ! Dip below the range and place the length
         ↪ above the x on the stack.  We then call
         ↪ <repetition> to make a sequence of x's that
         ↪ is the same length as the range. Stack now
         ↪ looks like ( repetition range -- )
15
16      zip
17      ! Zip takes two sequences and "zips" them
         ↪ together. For a sequence X and Y we would
         ↪ get { { x1 y1 } { xn yn } } and so on.
18
19      ! Now for the magic.
20
21      [
22          [ grid-size get H{ } square boa ]
23      ! This quotation holds our grid-size, and a hash
         ↪ table.  We'll explain that in a moment! It
         ↪ then uses boa to make the square.
24
25          with-datastack
26      ! We're going to explain this in it's own
         ↪ section next.
27      ] map ;
```

You might find that most of this is fairly simple except for one part.[2] Where are we actually getting the x and y values before **boa** is called by our inner quotation? The answer to that is found in a friendly little word called, **with-datastack**.

Consider that right after **zip** is called we essentially have a stack that looks like this.

[2] I'm going to post this several times but it is a very good idea to try to run words part by part in the REPL. For this one all you need to do is put a single value like 40 and a sequence like { 40 80 120 }. You could then type only **[length] keep**, and then the next line, and so on. In this manner you can see exactly what the stack looks like as this word is run. If you are using the listener you can use the stepper to do the same thing!

```
--- Data stack:
{ { 0 40 } { 0 80 } { 0 120 } ... { 0 640 } }
```

Furthermore we're going to be mapping over this sequence so there will only be a single part (like { 0 40 }) on the stack when the quotation needed by **with-datastack** is called.

```
{ 0 40 }
```

We remember that we're trying to create a **square** which needs to have certain values on the stack before we call **boa**.

```
posx posy size container square boa
```

The problem here is we have { 0 40 }, not *0 and 40* or *0 40*. This is a problem because **boa** will interpret that as one argument only and complain that we're lacking data. The solution to this is found in **with-datastack**. **with-datastack** will take a sequence and place it's values on the stack individually, it will then run a quotation in that context for us. Such that when our inner quotation is run we end up with something like this.

```
! { 0 40 } [ grid-size get H{ } square boa ]
 ↪ with-datastack
! with-datastack places the { 0 40 } on the stack
 ↪ for us.

--- Data stack:
0
40
[ grid-size get f square boa ]

! with-datastack then calls the quotation, and
 ↪ since boa will take 2 more values off the
 ↪ stack than are contained within it's quotation
 ↪ we end up with.

--- Data stack:
T{ square }
```

Easy! Our definition of **make-grid-line** is complete. Now we just have to double check that **setup-grid-square** is doing what we asked. What we want is a single sequence that holds all our squares. However, our current

definition won't give us that because **make-grid-line** returns a sequence itself.

```
! map from make-grid-line returns { square ...
 ↪ square }
! map from setup-grid-squares ends up with { {
 ↪ square ... square } { square ... square } ... }
```

All we need to add is the handy **flatten** word. **flatten** will, as the name implies, remove any inner sequences and splice them into the outer sequence. It looks like this.

```
IN: fof-poet USE: sequences.deep
IN: fof-poet { { 0 0 } { 1 1 } { 2 2 } }
--- Data stack:
{ { 0 0 } { 1 1 } { 2 2 } }

! This is a sequence that contains 3 sequences.

IN: fof-poet flatten
--- Data stack:
{ 0 0 1 1 2 2 }

! Our sequence after calling flatten.  Notice it is
 ↪ all only one sequence now.
```

Our final version of **setup-grid-squares** then looks like this.

```
: setup-grid-squares ( -- )
    0 get-screen-width grid-size get <range>
    [ 0 get-screen-height grid-size get <range>
     ↪ make-grid-line ]
    map flatten grid set ;
```

Our only other addition here is **grid set** which will set the symbol **grid** to hold the sequence of squares we have just generated. Our grid system is essentially almost done at this point. What are we missing though? A way to render it all of course! By this point you might be able to guess which word we need next. We have a sequence of squares who's arguments mostly map to what we need for **draw-rectangle-lines**. How are we going to render it all?

```
: draw-grid ( -- )
    grid get [ draw-square ] each ;
```

Higher order functions save the day once again! Now we only need to focus on how to draw an individual square. To understand how we procede from here let's consider what our tuple, **square** looks like compared to the arguments to **draw-rectangle-lines**.

```
! square
TUPLE: square posx posy size container ;

! draw-rectangle-lines stack effect
( posx posy width height color -- )
```

On first glance they seem to be quite close in their mapping. *Size* is both the *height* and *width*, so all we need to do is drop the *container* slot and add a *color*. If only we could turn our **square** into a sequence. Then all we'd have to do is make some minor modifications before calling **draw-rectangle-lines**. Luckily, Factor provides a word called **tuple-slots** for exactly this purpose.

```
--- Data stack:
T{ square f 0 0 40 f }

IN: fof-poet tuple-slots
--- Data stack:
{ 0 0 40 f }
```

Try to pause for a moment and consider what you would have to do in order to call **draw-rectangle-lines** if your stack looked like this.

```
0  ! posx
0  ! posy
40 ! size
f  ! container
```

Here's how our version will look.

```
: draw-square ( grid-square -- )
    tuple-slots
    [ drop dup BLACK draw-rectangle-lines ]
    with-datastack drop ;
```

```
5
6   ! Commented
7   : draw-square ( grid-square -- )
8       tuple-slots ! Convert square to sequence
9       [
10          drop dup
11          ! Drop container slot, duplicate size since
              ↪ width = height.
12          BLACK
13          ! Add a color to the stack
14          draw-rectangle-lines
15          ! Use our raylib word to draw the square.
16          ]
17      with-datastack drop ;
18      ! with-datastack will essentially put posx,
          ↪ posy, size, and container on the stack
          ↪ before the quotation is called.
```

With that we've finished our grid and have set some groundwork to build upon. Our input, rendering, and AI words will all make use of this grid for our game. One thing we did not touch on is the mysterious **H{ }** in our **make-grid-line**. That is the literal syntax for a hash-table in Factor. If you don't know what a hash table is, don't worry, we're going to introduce it when we actaully start working with our grid. Before we close out this section we should recap on what should currently be in **fof-poet.factor**.

```
1   USING: raylib.ffi kernel sequences locals
      ↪ alien.enums namespaces math
      ↪ classes.struct accessors combinators
      ↪ math.ranges sequences.deep continuations
      ↪ assocs classes.tuple ;
2   IN: fof-poet
3
4   TUPLE: square posx posy size container ;
5   SYMBOL: grid-size
6   SYMBOL: grid
7
8   : make-window ( -- )
9       640 480 "Mad Dash til Balderdash" init-window
10      60 set-target-fps ;
11
12  : clear-window ( -- )
13      RAYWHITE clear-background ;
```

```
14
15 : setup-grid ( -- )
16     40 grid-size set ;
17
18 : make-grid-line ( x range -- list )
19     [ length ] keep
20     [ swap <repetition> ] dip
21     zip
22     [ [ 40 H{ } square boa ] with-datastack ] map ;
23
24 : setup-grid-squares ( -- )
25     0 get-screen-width grid-size get <range>
26     [ 0 get-screen-height grid-size get <range>
       ↪ make-grid-line ]
27     map flatten grid set ;
28
29 : draw-square ( grid-square -- )
30     tuple-slots
31     [ drop dup BLACK draw-rectangle-lines ]
32     with-datastack drop ;
33
34 : draw-grid ( -- )
35     grid get [ draw-square] each ;
36
37 : render ( -- )
38     begin-drawing
39     clear-window
40     draw-grid
41     end-drawing ;
42
43 : setup ( -- )
44     setup-grid
45     setup-grid-squares ;
46
47 : main ( -- )
48     make-window setup
49     [ render window-should-close not ] loop
50     close-window ;
```

5.1.2 Inspiring Input

It is the subtle irony of the iterative process that in building anything we typically end up with a wooden bridge when we imagine a glorious gilded one. When we do our input system this emphasis will become clear, as

we forgo the proverbial power-tools Factor gives us and stick it out with a humble screwdriver. Why not use the power tools first? For one, we'd have to pause to explain the *OOP* system and secondly we'd never realize why we appreciate the easier tools. Tis good that we should see how to do things in a more primitive way and then find a contrast between the other components Factor offers. In other words, the point is for you to see the **why** of the higher level components instead of me simply saying, '*These are better use them!*'. That being said, this section might be a doozy. If you don't feel like you have the **stack manipulation** words down, well, we'll try to explain everything.[3]

We begin as we always do questioning the validity of our origins. Since we've already done some input before we have a good starting point.

```
: process-input ( keypair -- )
    dup first ! Get the key
    enum>number is-key-down
    [ second call( -- ) ] [ drop ] if ;

: check-input ( -- )
    {
        { KEY_W [ move-up-square     ] }
        { KEY_S [ move-down-square   ] }
        { KEY_A [ move-left-square   ] }
        { KEY_D [ move-right-square  ] }
    }
    [ process-input ] each ;
```

This wasn't a horrible first attempt at handling input, but it's possible we could do better. The problem is, how? The answer seems to lie in a bundle of questions.

How do we move from one square to the other?
How do we represent the player?
How do we write our input words to use the grid?

The secret lies in that weird looking **H{ }** we defined earlier in **make-grid-line**.

Hashing it out

A **Hash-Table** is a lot like a conversation with someone. You bring up a topic, and if they have something to say about it they'll respond with a

[3] There is a certain balance between explaining absolutely everything and trying to get on to our next game.

rambling sentence containing their views.[4] That is, what we ask could be called a **key** and what they respond with is a **value**. Hash-tables could be viewed this way, but with one important difference; we have to put the key-values in ourselves.[5] One way to create a hash table in Factor is to use the literal syntax. That would look like so...

```
H{ }
```

But wait, didn't we put that for each square that gets made? That's right! Each square has an internal hash-table that we can hold a conversation with.[6] Let's try to store some data in a hash-table using a key-pair. The word we can use to do this is **set-at** which takes a *value*, a *key* for that value, and finally, at the top of the stack, a *hash-table*. Since the hash-table implements the *assoc* protocol, which we'll learn about later, we can also use **set-at** for any type of associative data structure. For now our eyes are fixed on the hash-table, let's fill it with something.

```
IN: scratchpad H{ }
--- Data Stack:
H{ } ! Hash table

IN: scratchpad 12042 "lucky numbers"
--- Data Stack:
H{ }            ! Hash table
12042           ! value
"lucky numbers" ! key

IN: scratchpad pick
! Pick will take the 3rd item down on the stack and
 ↪ place it on top.  In this case it will copy our
 ↪ hash table to the top of the stack. ( x y z --
 ↪ x y z x )
--- Data Stack:
H{ }
12042
```

[4] Especially if you bring up politics!

[5] Although we won't get into the technicals of how a hash-table is implemented or it's algorithmic properties, it will suffice to say that a hash-table is often one of the faster data structures you can use in a programming language.

[6] So many footnotes...is it a good idea to use a hash-table instead of an list or array for each square? There's probably no noticeably difference in speed with the amount of data we're using, but we're deciding to use them anyway.

```
"lucky numbers"
H{ }  ! Same as the hash table at the bottom of the
  ↪  stack.

IN: scratchpad set-at
--- Data Stack:
H{ { "lucky numbers" 12042 } }
```

Our hash table holds a singular key. We can now have a primitive conversation with it, much like we would with a shady bartender. We could slide our arm onto the proverbial counter and whisper,

Hey, you know the lucky numbers?

I might, but that's not the way you ask buddy!

We can ask properly in Factor in numerous ways but the simplest would be to use **at**. To use it, we need to supply a key and a hash-table we want to ask for that key from. Some versions of **at** exist like **?at** or **at*** which will return extra values if the key is false, doesn't exist, or the value is false or missing. Since we'll be using **at*** later, we'll explain it here too.

```
--- Data Stack:
H{ { "lucky numbers" 12042 } }

IN: scratchpad dup "lucky numbers" swap
--- Data Stack:
H{ { "lucky numbers" 12042 } }
"lucky numbers"
H{ { "lucky numbers" 12042 } }

IN: scratchpad at
--- Data Stack:
H{ { "lucky numbers" 12042 } }
12042
```

Aha! The lucky numbers! For posterities sake perhaps we should ask what the unlucky numbers are so we don't accidentally use them.

```
--- Data Stack:
H{ { "lucky numbers" 12042 } }

```

```
IN: scratchpad dup "unlucky numbers" swap at
--- Data Stack:
H{ { "lucky numbers" 12042 } }
f
```

If the key we ask for doesn't exist in the hash-table, then Factor will return a **f** value for us to show this. However, if we want to know the difference between a value actually being **f** and a key not existing we can use **at***.

```
--- Data Stack:
H{ { "lucky numbers" 12042 } }

IN: scratchpad dup "unlucky numbers" swap at*

--- Data Stack:
H{ { "lucky numbers" 12042 } }
f    ! Value of the key.  Since it's missing we get
  ↪  a f or false value.
f    ! Boolean that say if the key exists in the
  ↪  hash table.  Since it doesn't exist we get a f
  ↪  or false value.
```

The last thing we need to know is how to remove a key from our table. This can be achieved using **delete-at** which takes a *key* to delete and the *hash-table* to delete it from.

```
--- Data Stack:
H{ { "lucky numbers" 12042 } }

IN: scratchpad dup "lucky numbers" swap delete-at

--- Data Stack:
H{ }
```

Now, how will all this help us? Well, let's think about this in terms of the grid. Each square essentially can act as a box we can put things in since we they all hold an internal hash-table we can get with the **container»** word. If that's the case then what if we put the following key in one of the boxes.

```
{ "player" t }
```

Where *"player"* is our key and **t** is our value. If that's the case then we've essentially boiled our player down to a *boolean* value. If we want to move the player all we have to do is remove his key from the square he's at and add a key to the square he's going to. Thus, only one player key will ever exist in the entire grid. We can then write a word to find which square the player is at, which will give us the coordinates of the square, and hence, the location we need to draw the player at. Another cool thing about this hash-table method is that we can also describe enemies in the same way.

```
{ "enemy" "up" }
```

In this case the key is now "enemy" and the value is a string describing the direction the enemy is headed. Note that even though this method is simple it comes at a cost as only one enemy can inhabit a square at any time. However, this will give us the chance to write a word that will let enemies bounce off each other too. Another property of this hash-table based grid is we get collisions for essentially free since all we have to check is if a *"player"* and *"enemy"* key exist in the same square. With that we have one part of our problem solved. We just need an input word that handles setting and moving these key-values around the grid.

5.1.3 Exercises

The power of keys

If you'd like, it would be immensely profitable to go look at some of the other words Factor has in regards to using data structures modeled after a key-value pairing. Take a few minutes to view the Assoc and Hash-table documentation. You can move on when you can answer the question...

> **How would you get all the keys in an associative data structure like a hash-table?**

There is a word for it.

5.2 Re-inspired Input

Each square in our grid is how many pixels away from each other? If you thought, 'by the **grid-size**' then you are correct! We can use this to find other squares from a square we currently posess. So for instance if we have the square at **(0, 0)** then if we increment x or y by the *grid-size*, we'll end

up with the square in that direction. We can model this idea by writing four words to handle incrementing from a current x or y to reach another square.

```
: up-square ( x y -- square )
    grid-size get - find-square  ;

: down-square ( x y -- square )
    grid-size get + find-square ;

: left-square ( x y -- square )
    [ grid-size get - ] dip find-square  ;

: right-square ( x y -- square )
    [ grid-size get + ] dip find-square ;
```

When we go up or down we need to modify the y value, and for left or right we modify the x value. Hence the usage of **dip** in the left or right words. This seems easy enough, but you can test it out in the REPL if you'd like to confirm that these do indeed modify the coordinates correctly. But what is this **find-square** word? Well, that's how we search our grid. It's also a somewhat complicated word, but nothing was ever won without some effort. Here's **find-square** which takes coordinates and searches the grid to find the square that matches them.

```
: find-square ( x y -- square )
    [ coordinates=? ] 2curry grid get
    swap
    filter ;
```

We shouldn't fear the presence of a foreign vocabulary. Let's assume that **coordiantes=?** is a word that takes a *square, x, and y* and returns a *bool* that is true if the coordinates are the same as the square, and false otherwise. In a moment we'll write that word ourselves, but what the heck is **2curry**?

Currying is a lot like taking a word that takes multiple arguments, for instance **coordinates=?** takes three, and turning it into a word that takes less arguments. Consider a very simple case like adding two numbers.

```
IN: scratchpad 2 3 [ + ]
--- Data Stack:
2
```

```
3
[ + ] ! Quotation that essentially will take two
  ↪  arguments.

IN: scratchpad curry
--- Data Stack:
2
[ 3 + ] ! New quotation that takes one argument

IN: scratchpad call
--- Data Stack:
5
```

Curry is useful when you have data you want to be used as an argument to a word, especially in a higher order context. So for instance if we wanted to write a word that takes a number and a sequence and maps over the sequence, adding the number to each member, we could write this.

```
: n-map ( sequence n -- sequence' )
    [ + ] curry
    ! Curry [ + ] with n such that we receive [ n +
     ↪  ] as our quotation.
    map  ;
```

Currying is a lot like the tortoise painting the crabs with a certain value before sending them out to do their work. In this case we want our n added to each number so we simply 'curry' it into the quotation we're going to apply to each member of the sequence. Now we return to our original word.

```
: find-square ( x y -- square )
    [ coordinates=? ] 2curry grid get
    swap
    filter ;
```

So what's happening here? Really try to figure it out! What's happening is that we are using **2curry** which takes two items off the stack and curries them into our quotation. Hence we will end up with something like.

```
[ x y coordinates=? ]
```

On our stack, but of course the values will be the actual coordinates we have put on the stack before we called the word. Such that we could have something like.

```
[ 40 40 coordinates=? ]
```

Next we call...

```
grid get swap
```

Which simply gets our sequence containing all the squares, and then swaps places with the quotation such that the quotation is now ontop of the stack. We then call...

```
--- Data Stack:
Sequence
Curried-Quotation

IN: scratchpad filter
```

filter as we may know will apply a quotation to each member of the sequence and return any members that returned *true* when the quotation was applied. In this way we are asking what square has the correct coordinates; if one exists we will get back something like this on the stack.

```
{ square }
```

If not we get an empty sequence back. Next we need to actually define **coordinate=?**. But we can't compare a square to x and y values. We need someway to get the **posx** and **posy** values from the square first. Let's write a word to do that.

```
! This version returns the original square too
: with-square-coordinates ( square -- x y square )
    [ posx>> ] keep
    [ posy>> ] keep ;

! If we purely want the coordinates we can use this
 ↪ word instead
: square-coordinates ( square -- x y )
    with-square-coordinates drop ;
```

In the interest of brevity we're going to continue on. We have 450 lines of code to get through after all! If these words don't make sense, double check what **keep** does, and how you get values from a **TUPLE:**.

Finally, here is **coordinates=?**

```
: coordinates=? ( square x y -- bool )
    [ square-coordinates ] 2dip
    swap [ = ] dip
    swap [ = ] dip and ;
```

We use **2dip** to jump under the *x and y* values to reach the square. After *square-coordinates* is called there will be four items on the stack.

```
x0 y0 ! Square
x1 y1 ! Values we provided
```

Some **stack-juggling** is done to compare both values against each other. As an exercise can you think of a better way to do this? Remember you can test this by putting four numbers on the stack and experimenting yourself. We use = to ask if numbers are equivalent, if they are we receive a **t** and if not a **f** is returned. We then use **and** to make sure that both of these cases returned **true**. Now that we have **find-square** defined our *up-square*, *down-square*, etc... words will work. That's great! But what are we going to do with them?

Well what we can do right now is, if we have a *square* we have four words that let us find it's neighboring squares. So maybe, when the player presses a key we should have a string returned saying which direction the press is headed, then we simply find the square in that direction and move the player onto it. All of the player's possible inputs could go in a word like, **player-inputs**.

```
: player-inputs ( -- inputs )
    {
        { KEY_W "up"    }
        { KEY_S "down"  }
        { KEY_A "left"  }
        { KEY_D "right"  }
    } ;
```

This word simply returns a sequence of sequences containing a *key-value* pair. The key is a *raylib* key and the value is a direction assigned to

that key. Let's imagine that we have a word that takes a *key-pair* and if is currently being pressed it returns which direction corresponds to the key, else it returns an empty string. This could be our new **process-input**

```
: process-input ( keypair -- result/bool )
    dup first ! Get the key
    enum>number is-key-down
    [ second ] [ drop "" ] if ;
```

Instead of the old version that used *call(* to call a quotation, we simply return a string describing the direction or the empty string. However, we haven't tied these together yet. We also need a word that takes a pair of coordinates, and a direction, and returns a square that corresponds to the square in that direction. So for instance if our starting point is (0, 0) and we press **S** then we would want to call **down-square** and receive the square at those coordinates.

```
: which-square ( x y direction -- square' )
    {
        { "up"    [ up-square    ] }
        { "down"  [ down-square  ] }
        { "left"  [ left-square  ] }
        { "right" [ right-square ] }
    } case ;
```

which-square will do this for us. Notice that it needs an x and y value, we'll have to write another word in a moment to provide that. Remember that *case* checks the top item on the stack against a the first item of each sequence. If any of the items are equal then the second value in the sequence will be called. In this case, **which-square** ends up returning the square that we would be moving to based on our key-press. We aren't done writing the words we'll need for our input, but let's try to come at this from the other direction. Here's the one higher level word that we'll handle all of the players input.

```
: check-input ( -- )
    player-inputs
    [ process-input dup
      "" equal?
      [ drop ]
      [ attempt-to-move ] if
    ] each ;
```

This seems complicated but it's really pretty simple. We first call **player-inputs** to put the sequence of possible inputs on the stack. Then we put a quotation on the stack that we are going to call on each *key-pair* in that sequence. The problem has been reduced to how do we handle a single key-press. So how do we?

Let's get our origin absolutely crystal. When that quotation is called this is an example of what we'd have on the stack.

```
{ KEY_W "up" }
```

Now we can step through our quotation. The first thing we do is ask,

> **Is this key being pressed?**

This is accomplished by *process-input*. Now we'll either have a direction or an empty string on the stack. So we ask,

> **Is there an empty string on the stack?**

This is handled by *equal?* which returns a Boolean for us. Now we have two possible answers.

> **The string is empty so drop our dup off the stack and end the quotation.**

Or

> **The string is a direction so attempt to move the player to a new square.**

It seems like we just need to define **attempt-to-move** then. This word will take a direction and try to move the player to the square in that direction.

```
: attempt-to-move ( direction -- )
    { "player" t } find-player-square rot
    move-square ;
```

Oh dear, almost everything here is new. Will it ever end? Probably not, but we'll continue on anyway.

```
{ "player" t }
```

Is the key-pair we want to pass onto the square we decide to move to. Remember all movement is done by changing the hash-tables, so we provide exactly which key-pair we are trying to change here. You might wonder why we have to include the **t**. That's because we are unifying enemy movement under some of the same words, and they don't use **t** but instead give the direction they are currently moving.

```
find-player-square
```

This is the word we'll use to locate the square holding the player at any time. We'll define it here, but we'll need to define some other words for it to work. These will defined in a moment.

```
: find-player-square ( -- square )
    "player" find-square-by-container first ;
```

We imagine that we have a word that will allow us to find a square if it contains a certain key in it's hash-table.

```
rot
```

We've seen this word before but to reiterate. All it does is takes the 3rd item on the stack and rotates it to the top.

```
1 2 3
rot
2 3 1
```

```
move-square
```

This will be the higher level word that performs all the steps and decisions in moving a square. We already have more than half of it defined.

```
: move-square ( tag square direction -- )
    [ dup square-coordinates ] dip
    which-square
    pick-square ;
```

The data stack when this is called would look something like.

```
{ "player" t } square "up"
```

We dip under the direction to get the coordinates for the square. Then we send the coordinates (x, y) and the direction to **which-square** which gives us back the square we are proposing to move to. Now we just have to pick if we can move to that square or have to stay. There are a couple reasons we might be forced to stay.

> We are hitting a wall (i.e. the end of the screen).
>
> For enemies, there is another enemy in the square we want to move to.

We're almost at the homestretch. Now we just need to define a few more words and our movement portion will be complete. **pick-square** has to decide if the new square is a valid move. Here is it's definition.

```
: pick-square ( tag square square2 -- )
    dup empty?
    [ 3drop ]
    [ first square-decision ] if ;
```

All we do in *pick-square* is check if the new square returned as an empty sequence {}. If it did, we just drop the whole thing and continue on with our merry life on the same square. If it did return a square we have to do some very scary things...First we call **first** to take square out of it's sequence.

```
{ square } first -> square
```

Next we need to write **square-decision**. This might be the most heinous word in the whole game. Are you ready?

```
:: square-decision ( tag square square2 -- )
    square2 tag first container=?
    [ square square2 tag ?change-direction ]
    [ square2 tag second tag first set-square-key
      square tag first remove-square-key ] if ;
```

Now that's a big boy. The first thing to note is that we used,

```
::
! instead of
:
```

in our definition of *square-decision*. This means we are using *lexical variables* instead of data laying around on the stack. When this word is called the first three items[7] will come off the stack and be bound to the names that were declared in the *stack notation*. Basically, in the context of only this word, we have the variables *tag, square, and square2*. What does this mean? It means if you write something like...

```
tag
```

...In the body of this word, then the value that was taken off the stack when the word was called and assigned to the symbol 'tag' will be thrown onto the stack. In other words, the stack is empty and when we initially call the symbol 'square2', then the value that was bound to 'square2' is placed on the stack for us to use. We can call any of these symbols as many times as we'd like. Without lexical variables this word looked even more horrifying then it does now. But hey, we said we're going to start primitively. Later, when we get access to *OOP*, words like this will be a distant memory.

```
square2 tag first container=?
```

The first thing we do is throw the new square we want to move to and the tag's key on the stack (in this case "player"). We then call **container=?** which we'll build on later but is also a word that our enemies will be using to see if there is another one of them on the square they want to move to. Still, let's still what **container=?** does.

```
: container=? ( square tag -- bool )
    swap container>> at* nip ;
```

Pffft! Easy. All we're doing is getting the hash-table from the square and using **at*** to ask if a key exists in the hash-table for the tag we've provided. Notice we use *nip* because *at** returns the value and then a Boolean to indicate if the key exists. We only need the Boolean so we *nip* the value below it.

[7] It's three in this case, but the actual amount is dependant on the stack-notation

Next we reach an **if** statement. This *if* is purely for enemy logic, so we're going to skip the next line and go to the case that will always be true since we're a player.

```
[ square square2 tag ?change-direction ]
! Skipping this
```

```
[ square2 tag second tag first set-square-key
      square tag first remove-square-key ]
```

Ok. This quotation is trying to accomplish two things. By now we've decide that the new square is a valid move, so all we need to do is set our key on that square, and remove it from the square we're currently on. This is a two-step process. First we summon up the new square, the value of the tag, and the key for the tag such that our stack looks like this now.

```
new-square value key
```

Now we can define **set-square-key** which simply takes this data and adds it into the square's hash-table. This is purely a convenience word.

```
: set-square-key ( square val key -- )
    pick container>> set-at drop ;
```

This one should be self-explanatory by now. If you need help try reading the documentation on **pick** and simulating what would happen. All we're doing is adding a key-value pair to a hash-table.

Lastly, we remove the key-value pair from our old square.

```
square key
```

Is on the stack from **square tag first** and now we need to define **remove-square-key**. However, try to take a moment to figure out how it might be defined. You can do it!

```
: remove-square-key ( square key -- )
    swap container>> delete-at ;
```

Are we done? Yes. We did it! At least as far as the player is concerned, we now have words that describe how the player can move. Let's recap on

what we did really quick. The basic gist is that we use *raylib* to figure out which keys have been pressed, when we find out that one has we return the direction that the key corresponds to. We then use that direction to find a new square to possibly move to, we figure out if there is a square there, and then simply move our key-value pair from our current square to the new square. Enemies essentially do the exact same thing, however they have a few extra components that let them check if a space is occupied. Let's try to define some enemies now. Rendering everything is going to be extremely simple, so don't worry, we're doing the hardest part first. It's all downhill, eventually!

5.2.1 Paving the Road

Finishing our implementation of both the player and enemies will go a lot smoother if we finish up defining our **grid utilities**. We've already defined the majority of the words we need. All that's really left is defining some words to add some convenience to working with containers. Our first one is to define a word that is a lot like **find-square** but lets us ask which square's contain a particular key.

```
: find-square-by-container ( name -- square )
    [ container=? ] curry
    grid get swap
    filter ;
```

We curry the tag onto our quotation and then filter through the grid to see which squares have the particular key we're looking for. This uses the **container=?** word we defined in the last section.

We'll also find it useful to find all squares that have a particular *x or y*. We can define those in much the same way.

```
: find-square-by-coordinate ( n -- squares )
    grid get swap filter ; inline

: find-square-by-y ( y -- squares )
    [ swap posy>> = ] curry
    find-square-by-coordinate ;

: find-square-by-x ( x -- squares )
    [ swap posx>> = ] curry
    find-square-by-coordinate ;
```

Building off **find-square-by-coordinate** we create two other words

that let us get all squares with a certain *x or y* value. We'll use this later to draw sidewalks, and the end screen. The only tricky part here is we swap the **posy»** word so the square is on top of the sequence. Think about how we're mapping over this if you're confused. Try to write down what the stack would look like when the quotation is called. We also need a word to give us the actual value of a key. We can describe this more concisely by writing it.

```
: get-square-key ( square key -- val )
    swap container>> at* drop ;
```

Although you can't see it yet, all of our **Vector2**'s will end up being generated from the square's coordinates. But this is not correct, we want anything we draw to have it's origin in the middle of the square, not on it's corner point. Thus we define two more words that take a vector and offset it to the proper coordinates.

```
: center-square ( -- x )
    grid-size get 2 / ;

: offset ( vector2 -- vector2' )
    [ x>> center-square + ] keep
    y>> center-square +
    Vector2 <struct-boa> ;
```

center-square simply halves our *grid-size*. The center point for any square is the coordinates incremented by the grid-size divided by 2. **offset** performs this function for us, and generates a new *Vector2* for our render word to use.

Last but kinda least is a function that gives us the middle square of our screen. So for instance, if we have a height of 480, then dividing by the *grid-size* merits 12 actual squares going height-wise. This word is used to determine what square the player should start on since we want him in the middle and at the bottom of the screen.

```
: middle-square ( n -- n' )
    grid-size get / 2 / floor
    grid-size get * ;
```

If we supply a *screen-height* or *screen-width* to this word we will get the coordinates for the middle square. Notice that we use **floor** which removes

the decimal part from the floating point number. This happens because our grid will never have a square coordinate at anything other than an integer.

```
15.5 floor --> 15
```

This effectively completes all the utility words we'll need.

5.2.2 The Enemy of My Enemies Enemy is the British Empire

Enemies in our game will be dependant on the *difficulty*. This will essentially be a number we keep track of that determines how many enemies should be spawned. As the player advances through the levels the enemy count will increase making the game much harder. Difficulty won't be so difficult to implement, we just need a symbol and a few simple words.

In-fact here is our entire difficulty system.

```
SYMBOL: difficulty
: set-difficulty ( n -- )
    difficulty set ;

: increase-difficulty ( -- )
    difficulty get
    1 + difficulty set ;

: reset-difficulty ( -- )
    0 difficulty set ;
```

Technically this is a lie, there are three more words we'll need to define later but they are going to be for our difficulty selection screen. For now, we can implement enemies with this. Let's recognize our current state. We have a number, and we need that many enemies to be generated each screen change. Now there are several ways we can do this, but overall we still want to distill the problem down to the act of setting up only one enemy. Then we just do it as many times as we need to. Let's imagine we have a word called **setup-enemy** that will place an enemy onto the grid. Here is how our higher level word could look.

```
: setup-enemies ( -- )
    difficulty get setup-enemy drop ;
```

In the interest of trying to hit a few different ways of doing things, we're going to implement *setup-enemy* without any higher order words. Instead we're going to rely on good old fashion *recursion*. Recursion is

one of those amazingly simple concepts that can sometimes be difficult to internalize. You could easily write a whole book about it.[8]. However, we have a limited amount of space with all the other items on our agenda. Simply put recursion is about this, we have a thing we want to keep doing but slightly different, and we have a reason to stop. This is typically referred to as the 'base case'. Without thinking about Factor for a moment let's try to think through the process.

: **What do we have?**

We have a number denoting the difficulty.

And what do we want to do with it?

Well, whatever that number is, is the number of enemies we'd like.

So if the number is 0 what would you do?

If the number is 0 we'd do nothing. Or if we were doing something we'd stop because there would be nothing more to do.

What if you had 2?

If we had two then I would setup an enemy. Then I would decrement the number I had because I would have 1 enemy and 1 enemy left to setup.

And then what would you do?

I'd setup another enemy nearly the same as I did before, and then decrement my number again to get 0. Now I have 2 enemies setup and 0 left to setup.

And then?

And now I stop because I'm at 0, and hope they'll be my friends.

Doubtful. ;

[8] And they did. The little schemer is a fantastic book for learning and internalizing recursion

Let's see how to implement recursion in Factor. We'll define a word, **setup-enemy**, that takes a **n**.

```
: setup-enemy ( n -- n' )
    grid-width grid-height find-square first
    directions random "enemy" set-square-key
    dup 0 =
    [ ] [ 1 - setup-enemy ] if ; recursive
```

The first two words are new aren't they? Here they are.

```
: grid-enemy-map ( n -- n' )
    grid-size get / 1 - random
    grid-size get * ;

: grid-height ( -- height )
    game-height grid-enemy-map ;

: grid-width ( -- width )
    game-width grid-enemy-map ;
```

All these words do is give us a random valid coordinate from our grid.

```
grid-width grid-height find-square first
```

We then call **find-square** to get the square we have randomly selected. We use first because *find-square* returns it's value as a sequence.

```
directions random "enemy" set-square-key
```

directions is simply a sequence of our directions.

```
: directions ( -- directions )
   { "up" "down" "left" "right" } ;
```

Now what's very cool is that **random** is smart enough to determine the type of value that was given to it. Give *random* a number? It'll give you a random number up to that value. Give *random* a sequence? It'll give you a random member of that sequence. We use this to get a random direction. We then place the key *"enemy"* on the stack. Now our stack looks like.

```
square-we-chose random-direction "enemy"
```

It's a trivial matter to call **set-square-key** which will add our enemy to the square's container and hence, the enemy effectively now exists in the game world. What's left? We never used that *n* that we placed on the stack, did we?

```
dup 0 =
   [ ] [ 1 - setup-enemy ] if ; recursive
```

First we duplicate the *n* and check if it's value is equal to 0. If it is we simply do nothing and the chain reaction will end. But if it isn't 0; then we still have more enemies to setup. In this case we'll decrease the *n* value to *n - 1* and call *setup-enemy* again. *setup-enemy* will keep calling itself until the number reaches 0. This is the essence of recursion. It may be strange that the definition of *setup-enemy* is defined in terms of itself but it's really a remarkably simply thought pattern that can apply to numerous different domains. Some solutions are easier to think about iteratively, some recursively, and others with higher order combinators. Factor gives you the freedom to use whatever you'd like. We do need to let Factor know we intend to do this however, and that is why the **recursive** word is used right after the semi-colon. Factor requires this word for defining recursive words. There are actually two types of recursion. One of them is called iterative recursion, which is the one that **setup-enemy** uses. This is a whole chapter in itself but it's good to think about the basics. *setup-enemy* does not depend on future results of it's recursion. That might be a definition that makes some people mad but it's a simple way to think about it. We could write a word that looks partially like...

```
: new-result ( n -- n' )
    dup 0 =
    [ 0 ]
    [ dup new-result + ] if ; recursive
```

This word (that doesn't do anything) would not be iterative recursion. Why? We could imagine that at some point in time when **new-result** is called in the word body that there now exists two *new-result*'s. We could call the original *new-result0* and the one called by *new-result0*, *new-result1*. When *new-result0* calls *new-result1* it can't actually exit. It's has to wait for the value from *new-result1* to return so it can end with it's final word (+).

This is already a bit too much of a detour for our purposes, but if this sort of thing interests you I leave a valuable book in the footnotes for you.[9]

Well is that it for our enemies? We still have to implement how they'll move. Let's do that now. Immediately, we can turn the problem into the problem of moving only a single enemy. How do we do this?

```
! This will return a sequence of all squares
  ↪ containing enemies
: enemies ( -- squarelst )
    "enemy" find-square-by-container ;

: move-enemies ( -- )
    enemies
    [ move-enemy ] each ;
```

Ok now we just need to define **move-enemy**. We know we're going to need to use **move-square** but what else is special about an enemy moving?

```
: move-enemy ( square -- )
    [ check-enemy-direction ] keep
    [ enemy-direction ] keep
    [ { "enemy" } swap suffix ] dip
    over second move-square ;
```

This'll definitely require a line-by-line! But first let's consider why this definition looks so complicated. There's one simple reason. The enemy needs to be able to turn around when it hits a wall. If we didn't implement this the enemy would hit a wall and stay stuck in nihilistic limbo for an eternity until we swiftly destroyed it with the ESC key. And we don't want that, I think.

```
[ check-enemy-direction ] keep
```

We use keep to hand the square to **check-enemy-direction**. What does this do?

```
: enemy-direction ( square -- direction )
    container>> "enemy" swap at ;

: check-enemy-direction ( square -- )
```

[9] First chapter of S.I.C.P. or Structure and Interpretation of Computer Programs.

```
    dup enemy-direction
    [ dup square-coordinates ] dip
    which-square { } equal?
    [ dup enemy-direction flip-direction ]
    [ drop ] if ;
```

Another big one. We can handle this. All this word implements is a solution to the problem just proposed. First we use *enemy-direction* to figure out what direction we're headed. Then we grab the coordinates of the square on the stack and send that, along with the direction we want to go, to **which-square**. If we receive an empty list from this then we need to flip the direction we're heading before we move. This is accomplished with the next few words.

```
: opposite-direction ( direction -- direction )
    {
        { "up"    [ "down" ] }
        { "down"  [ "up" ] }
        { "left"  [ "right" ] }
        { "right" [ "left" ] }
        [ drop f ]
    } case ;

: flip-direction ( square direction -- )
    opposite-direction "enemy" set-square-key ;
```

Imagine that we have this on the stack.

```
square "up"
```

However, we know that we can't go up. So we call **flip-direction** which reverses our direction and then sets that value into the square we're on. This is all **check-enemy-direction** is ultimately doing. We can now return to *move-enemy*.

```
! move-enemy ...
    [ enemy-direction ] keep
    [ { "enemy" } swap suffix ] dip
```

Once we're sure of our direction we get it again, the next part is rudimentary. Our stack looks like.

```
"down" square
```

But we need to add the correct tag to send to *move-square*. Hence our next line dips under the square and appends "down" to the sequence resulting in.

```
{ "enemy" "down" } square
```

Our last line to look at from *move-enemy* is...

```
over second move-square ;
```

Try to find out what this does. Remember that, *move-square* needs a direction on-top of the stack. And now altogether again you should hopefully, be able to understand what this word does.

```
: move-enemy ( square -- )
    [ check-enemy-direction ] keep
    [ enemy-direction ] keep
    [ { "enemy" } swap suffix ] dip
    over second move-square ;
```

You've almost made it through all of the input functions. But there's one final beast we have to conquer. You might eventually forgive me for starting so primitive. On the bright side, you're really gonna appreciate **OOP** and some of the other more advanced things we can do later on. We recall the former beastly definition of **square-decision**.

```
:: square-decision ( tag square square2 -- )
    square2 tag first container=?
    [ square square2 tag ?change-direction ]
    [ square2 tag second tag first set-square-key
      square tag first remove-square-key ] if ;
```

The line in question is...

```
[ square square2 tag ?change-direction ]
```

Here's the scoop. We needed this line because there may be a instance where two randomly generated enemies are placed on the same row or column and could potentially meet each other. This poses a problem because

if they ever land on the same square than one of the *key-value* pairs is going to be destroyed since our system only recognizes one. Now we could rewrite our system, but that means making it even more complicated. We'd lose a lot of the simpler definitions we had and some of our convenience words would go right out the window. Later on we can build a more complicated system but for our first game it seems hardly proper. So instead of that, we opt to put up with two rather boisterous words. Our other word **?change-direction** will take two squares and a tag. It will then look at the square we propose to move to, to see if there is a tag of the same type already occupying the space. If there is then *?change-description* will flip the direction of the tags on both squares. In other words it will look like the enemies bounced off each other because they reverse direction at the same time. Here is our definition, we relate no more description of it's working but rather occupy our current time with a moment of silence over the programmers of old who, through blood and sweat, programmed in a way more bespoken to cavemen and this present word before us.

```
! Since it was only explained once, DEFER: simply
 ↪  tells the compiler we want to use the word
 ↪  before we define it, but will define it later.
! Change the tag of a square into it's
 ↪  opposite-direction
: change-former ( square tag -- )
    dup first [ second opposite-direction ] dip
    set-square-key ;

Change the
: change-later ( square tag -- )
    dup [ second ] dip first set-square-key ;

:: ?change-direction ( square square2 tag -- )
    square2 tag first get-square-key
    tag second opposite-direction equal?
    [ square tag change-former
      square2 tag change-later ] when ;
```

We survived input. I'll admit if you haven't built up a mindset for the stack yet, then this previous section was probably pretty challenging. However, there really is no other way to build up the mindset other than bashing your head into it. Hopefully, I at least make the pain a little more enjoyable. Eventually, it will recede and you'll find it second nature to deal with the stack. But not only that, while feeling freer you'll still grow to appre-

ciate the nature of Factoring out both the problems you wish to solve and the way you go about solving them. It's a little appreciated side-benefit of Factor to build such a mindset in you. I guarantee it will help you in any other language and you will find yourself breaking problems down easier while building more reusable components with simpler inner workings. Of course, it won't be as enjoyable as Factor though!

5.3 Baking Crumpets

You may not believe it, but we're actually almost done with the whole game. We have only a few things left. Notably, we need to have a way to select difficulty, a way to win, a way to lose, and our game loop. None of these things are as complicated as input was. Let's begin!

Let's do collisions and a way to lose since they are mostly the same thing in our program. Ready for how we detect collisions? I bet you're thinking there's a bunch of **Vector2**'s involved and it's going to be this long drawn out and complicated thing. Here it is in 5 words.

```
: detect-collision ( -- )
    find-player-square
    enemies
    member?
    [ game-over-screen ] when ;
```

That's it we did collisions. All this does is gets the player's square, and the sequence containing all the squares with enemies. Then we just use **member?** to ask if the player is a part of any of these squares. If he is, it's game over bud. Let's see how we can handle a game over screen.

```
: game-over-screen ( -- )
    [ draw-game-over
     game-over-input window-should-close not ]
     loop ;
```

We basically have a miniature version of our future game loop include in this word. Drawing is exceptionally simple, all we want to do is alert the player of his failure and give him an option to redeem himself.

```
: game-over-text ( -- )
    "Press Space to Continue \nEscape to Exit"
    40 get-screen-height 2 /
    30 BLACK draw-text ;
```

```
5
6  : draw-game-over ( -- )
7      begin-drawing
8      clear-window
9      game-over-text
10     end-drawing ;
```

We've seen this sort of render loop before. The only notable component is the introduction of the **draw-text** word. This word takes the following.

```
1  string x-pos y-pos font-size color
```

For this we elect to draw the text a little off to the side. Exquisite! If you'd like to berate or console the player, now would be the time to edit the string. I won't judge you. Just remember that you can use '\n' to break the line like you regularly would with text modifiers. The only other word we need to wrap up this beautiful bundle of abject failure we want to present to the player is a word to detect if the player wants to restart the game.

```
1  : game-over-input ( -- )
2      {
3          { KEY_SPACE "space" }
4      }
5      [ process-input
6        "space" equal?
7        [ reset-difficulty setup game-loop ] when ]
          ↪ each ;
```

This looks much like our previous input word only this time we are only going to check space. If *process-input* returns "space" then we restart the game, else we keep looping until the player makes up his mind. We can restart the game in a very horrible way. The way we did it. One of the exercises later on will be investigating how we did this wrong and considering solutions, but for now you only get a hint! How do we restart the game exactly? Well that's easy we simply call **setup** to reinitialize the grid and player (we'll define it at the end since it's super simple), and then call **game-loop** to restart the game with freshly initialized data.

While we're here let's also throw in a little number in the top right screen that'll let him keep track of what level/difficulty he's on.

```
: draw-level-num ( -- )
    difficulty get number>string
    10 10 20 RED draw-text ;
```

This simply gets the difficulty and converts it to the string before drawing it. We'll add it to our render loop later.

Now we have to work on how a player wins, and how a player selects the difficulty. Both are somewhat related but I'll let you pick which one we do first.

...

Really? If you say so; that one hasn't gone first since the incident but alright.

5.3.1 The Incident

When our game first starts the player is greeted with a difficulty selection screen. He can choose from the following difficulties.

```
: difficulties ( -- n )
    { { "one" 3 } { "two" 5 } { "three" 10 } } ;
```

Each screen the player beats increases the difficulty by 1. When the final screen is reached a ending zone will be drawn instead, and when the player steps inside he'll be greeted with the **game-won** screen. Here's how we can implement the difficulty screen.

```
: choose-difficulty ( -- )
    [ render-difficulty-screen
      check-difficulty-input
      max-level get number? not ] loop ;
```

This is not so unlike our **game-over-screen** word. Rendering the difficulty screen can be handled like so.

```
: draw-difficulty-box ( -- )
    "Press 1, 2 or 3 to select\n
    (1) 3 Levels\n
    (2) 5 Levels\n
    (3) 10 Levels\n"
    40 40
    30 BLACK draw-text ;
```

```
: render-difficulty-screen ( -- )
    begin-drawing
    clear-window
    draw-difficulty-box
    end-drawing ;
```

Nothing surprising here either. We're becoming pro's already. But this sort of familiarity should also tip us off to the need for better abstractions. We are 'almost' repeating ourselves in this way. Future games will need to have some improvements built on the lessons learned from this one.

```
SYMBOL: max-level
: check-difficulty-input ( -- )
    {
        { KEY_ONE "one" }
        { KEY_TWO "two" }
        { KEY_THREE "three" }
    } [ process-input dup
        "" equal?
        [ drop ]
        [ difficulties at max-level set ]
        if ] each ;
```

We use our **at** word with the sequence of sequence we get from **difficulties**. This is one example of **at** being available for different data types. Our new visitor here is the **max-level** symbol. This will be used to compare the current difficulty level and see if we're on the final screen or not. Otherwise, this should hopefully be understandable since we've seen this pattern before.

```
max-level get number? not
```

You might have noticed in our **choose-difficulty** that we have this line. *max-level* is initially set to to *f* when our game starts. Since that is not a number, **number?** will return false. We then use **not** to turn this into a true value and keep the *loop* running. When we do end up setting a number this value will become false and the loop will end. Although we don't have all the words yet we can finally describe what our game loop will look like.

```
: game-loop ( -- )
    [ check-input render move-enemies
      detect-collision ?next-screen
      window-should-close not ] loop ;
```

We'll leave rendering for the absolute end since it won't take very long at all. For now let's try to understand what **?next-screen**'s purpose is. Every time the game loop goes around we're going to run this word and it's purpose is twofold. It will be responsible for checking if the player has reached the top of the current screen (and thus it'll need to load a new screen), and it also checks if the player is on the last screen. If he is and has reached the top row then we can move to the **game-won** screen.

```
: ?next-screen ( -- )
    find-player-square
    posy>> 0 =
    [ difficulty get max-level get =
      [ game-won ]
      [ increase-difficulty setup game-loop ] if
    ] when ;
```

We use **posy»** because any coordinate with a y = 0 will be at the very top of the screen. After comparing the *difficulty* to the **max-level** we can determine if the player has won or should move to a new screen. All that's left to define is the how it looks when we win.

```
: draw-game-won-text ( -- )
    "You reached the\nPoet Society!\nSpace to play
     ↪ again."
    40 get-screen-height 2 /
    30 BLACK draw-text ;

: draw-game-won ( -- )
    begin-drawing
    clear-window
    draw-grid
    draw-ground
    draw-game-won-text
    end-drawing ;

: game-won ( -- )
    [ draw-game-won game-over-input
```

```
16        window-should-close not ] loop ;
```

One surprising fact here is that we use the **game-over-input** word to allow the player to restart the game from the *game-won* screen. And that's that. Finally, we implement some rendering. But we'd like our game to look somewhat nice, so why don't we include some textures? We can use textures to draw nice images on-top of our squares, implement sidewalks, and add a door to the final screen for the player to exit out of. *Raylib* facilitates this incredibly deftly.

```
1  SYMBOL: textures
2  : setup-textures ( -- )
3      { "ground.png" "sidewalk.png" "end.png"
       ↪  "door.png" }
4      [ dup load-image load-texture-from-image
5        [ { } swap suffix ] dip suffix ]
6      map
7      textures set ;
8
9  : get-textures ( key -- texture )
10     textures get at ;
```

We first create a sequence composed of filenames for the current directory. Then we map over this sequence using **load-image** which takes a file-name and returns an image, and **load-texture-from-image** which takes a loaded image and converts it to a texture. Afterwards we create a sequence that looks like...

```
1  { file-name texture-struct }
```

Which will turn into something like...

```
1  {
2      { file-name0 texture-struct0 }
3      { file-name1 texture-struct1 }
4  }
```

And be placed into our *textures* symbol. Now we can use **get-textures** to ask for any texture we've loaded. We can use the ground texture and map over all the squares in our grid to create a nice floor.

```
: draw-ground ( -- )
    grid get
    [ square-coordinates [ "ground.png"
     ↪ get-textures ] 2dip
      RAYWHITE draw-texture ] each ;
```

The same can be done with making sidewalks. We simply ask for any squares that exist where x = 0 or x = game-width, and then map over them with the same **draw-texture** word. **draw-texture** will place a texture we've loaded onto a coordinate point. It takes the following values.

```
texture posx posy color
```

```
: draw-ground ( -- )
    grid get
    [ square-coordinates [ "ground.png"
     ↪ get-textures ] 2dip
      RAYWHITE draw-texture ] each ;
```

Enemies can be drawn with a few simple words too.

```
: enemy-vector ( square -- vector )
    [ posx>> ] keep
    posy>> Vector2 <struct-boa> ;

: draw-enemy ( vector -- )
    offset 20.0 BLUE draw-circle-v

: draw-enemies ( -- )
    enemies
    [ enemy-vector draw-enemy ] each ;
```

enemy-vector will take a square an enemy is on and generate a *Vector2* for it. We then send this vector to **draw-enemy** which in turn will update it with **offset** so it is drawn correctly in the square. Drawing the player is mostly the same.

```
: player-vector ( -- vector )
    find-player-square
    [ posx>> ] keep
    posy>> Vector2 <struct-boa> ;
```

```
: draw-player ( -- )
    player-vector offset
    20.0 RED draw-circle-v ;
```

We can also draw a fancy final row with a door when the player reaches the final screen. We'll call this **draw-winning-end** and it'll check if it's on the final level before letting itself draw.

```
: draw-ending-door ( -- )
    "door.png" get-textures game-width
     ↪ middle-square 0
    RAYWHITE draw-texture ;

: draw-winning-end ( -- )
    difficulty get max-level get =
    [ 0 find-square-by-y
      [ square-coordinates [ "end.png" get-textures
       ↪ ] 2dip
        RAYWHITE draw-texture ] each
    draw-ending-door ] when ;
```

And with that we've easily finished our rendering words. After tying it all together we'll finally have our first game!

```
! String all our rendering words together.  Is
 ↪ there a better way to do this?
: render ( -- )
    begin-drawing
    clear-window
    draw-grid
    draw-ground
    draw-sidewalks
    draw-winning-end
    draw-player
    draw-enemies
    draw-level-num
    end-drawing ;

! The grid, the squares, the player, and all
 ↪ enemies get reset on any screen change or game
 ↪ restart.
: setup ( -- )
```

```
    setup-grid
    setup-grid-squares
    setup-player
    setup-enemies ;

! words that only run once when the game starts and not during a restart
: pre-setup ( -- )
    setup-textures
    choose-difficulty
    reset-difficulty ;

! for REPL development
: reset-max-level ( -- )
    f max-level set ;

: main ( -- )
    make-window
    pre-setup
    setup
    game-loop
    reset-max-level ! While developing you'll want this
    close-window ;

MAIN: main
```

Whoosh! What a journey. Before closing this section out let's post the full version of what we should have in **fof-poet.factor**.

```
! Copyright (C) 2019J Jack Lucas
! See http://factorcode.org/license.txt for BSD license.
USING: raylib.ffi kernel sequences locals alien.enums
namespaces math classes.struct accessors combinators math.ranges
sequences.deep continuations assocs classes.tuple math.functions random
    math.parser ;
IN: fof-poet

TUPLE: square posx posy size container ;
```

```
10
11  SYMBOL: grid-size
12  SYMBOL: grid
13
14  : make-window ( -- )
15      800 600 "Mad Dash til Balderdash" init-window
16      10 set-target-fps ;
17
18  : clear-window ( -- )
19      RAYWHITE clear-background ;
20
21  ! The Grid
22  : setup-grid ( -- )
23      40 grid-size set ;
24
25  ! Remove the points that would lie outside the
     ↪ canvas
26  : game-height ( -- height )
27      get-screen-height grid-size get - ;
28
29  : game-width ( -- width )
30      get-screen-width grid-size get - ;
31
32  : make-grid-line ( x range -- list )
33      [ length ] keep
34      [ swap <repetition> ] dip
35      zip
36      [ [ grid-size get H{ } clone square boa ]
         ↪ with-datastack ] map ;
37
38  : setup-grid-squares ( -- )
39      0 game-width grid-size get <range>
40      [ 0 game-height grid-size get <range>
         ↪ make-grid-line ]
41      map flatten grid set ;
42
43  : draw-square ( grid-square -- )
44      tuple-slots
45      [ drop dup BLACK draw-rectangle-lines ]
46      with-datastack drop ;
47
48  : draw-grid ( -- )
49      grid get [ draw-square ] each ;
50
```

```
! Grid Utilities

! Extract the square coordinates
: with-square-coordinates ( square -- x y square )
    [ posx>> ] keep
    [ posy>> ] keep ;

: square-coordinates ( square -- x y )
    with-square-coordinates drop ;

! Check if a pair (x,y) matches a square
: coordinates=? ( square x y -- bool )
    [ square-coordinates ] 2dip
    swap [ = ] dip
    swap [ = ] dip and ;

: find-square ( x y -- square )
    [ coordinates=? ] 2curry grid get
    swap
    filter ;

: container=? ( square name -- bool )
    swap container>> at* nip ;

: find-square-by-container ( name -- square )
    [ container=? ] curry
    grid get swap
    filter ;

: find-square-by-coordinate ( n -- squares )
    grid get swap filter ; inline

! Find all squares that have a certain x or y
: find-square-by-y ( y -- squares )
    [ swap posy>> = ] curry
    find-square-by-coordinate ;

: find-square-by-x ( x -- squares )
    [ swap posx>> = ] curry
    find-square-by-coordinate ;

: set-square-key ( square val key -- )
    pick container>> set-at drop ;

```

```
: remove-square-key ( square key -- )
    swap container>> delete-at ;

: get-square-key ( square key -- val )
    swap container>> at* drop ;

: center-square ( -- x )
    grid-size get 2 / ;

: offset ( vector2 -- vector2' )
    [ x>> center-square + ] keep
    y>> center-square +
    Vector2 <struct-boa> ;

: middle-square ( n -- n' )
    grid-size get / 2 / floor
    grid-size get * ;

! Player
: find-player-square ( -- square )
    "player" find-square-by-container first ;

: player-vector ( -- vector )
    find-player-square
    [ posx>> ] keep
    posy>> Vector2 <struct-boa> ;

: draw-player ( -- )
    player-vector offset
    20.0 RED draw-circle-v ;

: setup-player ( -- )
    t "player"
    game-width middle-square
    game-height find-square first container>>
    set-at ;

! Movement Words

! Square2 Doesnt exist stay
! Square2 occupied with same tag Stay
! Else move to square2
DEFER: opposite-direction

```

```
: change-former ( square tag -- )
    dup first [ second opposite-direction ] dip
    set-square-key ;

: change-later ( square tag -- )
    dup first pick swap get-square-key
    opposite-direction [ first ] dip
    swap set-square-key ;

:: ?change-direction ( square square2 tag -- )
    square2 tag first get-square-key
    tag second opposite-direction equal?
    [ square tag change-former
      square2 tag change-later ] when ;

:: square-decision ( tag square square2 -- )
    square2 tag first container=?
    [ square square2 tag ?change-direction ]
    [ square2 tag second tag first set-square-key
      square tag first remove-square-key ] if ;

: pick-square ( tag square square2 -- )
    dup empty?
    [ 3drop ]
    [ first square-decision ] if ;

: up-square ( x y -- square )
    grid-size get - find-square  ;

: down-square ( x y -- square )
    grid-size get + find-square ;

: left-square ( x y -- square )
    [ grid-size get - ] dip find-square  ;

: right-square ( x y -- square )
    [ grid-size get + ] dip find-square ;

: which-square ( x y direction -- square' )
    {
        { "up"    [ up-square    ] }
        { "down"  [ down-square  ] }
        { "left"  [ left-square  ] }
        { "right" [ right-square ] }
```

```
183     } case ;
184
185 : move-square ( tag square direction -- )
186     [ dup square-coordinates ] dip
187     which-square
188     pick-square ;
189
190 : process-input ( keypair -- result/bool )
191     dup first ! Get the key
192     enum>number is-key-down
193     [ second ] [ drop "" ] if ;
194
195 : player-inputs ( -- inputs )
196     {
197         { KEY_W "up"    }
198         { KEY_S "down"  }
199         { KEY_A "left"  }
200         { KEY_D "right"  }
201     } ;
202
203 : attempt-to-move ( direction -- )
204     { "player" t } find-player-square rot
      ↪ move-square ;
205
206 : check-input ( -- )
207     player-inputs
208     [ process-input dup
209       "" equal?
210       [ drop ]
211       [ attempt-to-move ] if
212     ] each ;
213
214 ! AI Entities
215 : enemy-vector ( square -- vector )
216     [ posx>> ] keep
217     posy>> Vector2 <struct-boa> ;
218
219 : draw-enemy ( vector -- )
220     offset 20.0 BLUE draw-circle-v ;
221
222 : enemies ( -- squarelst )
223     "enemy" find-square-by-container ;
224
225 : draw-enemies ( -- )
```

```
226     enemies
227     [ enemy-vector draw-enemy ] each ;
228
229 : enemy-direction ( square -- direction )
230     container>> "enemy" swap at ;
231
232 : opposite-direction ( direction -- direction )
233     {
234         { "up"    [ "down" ] }
235         { "down"  [ "up" ] }
236         { "left"  [ "right" ] }
237         { "right" [ "left" ] }
238         [ drop f ]
239     } case ;
240
241 : flip-direction ( square direction -- )
242     opposite-direction "enemy" set-square-key ;
243
244 : check-enemy-direction ( square -- )
245     dup enemy-direction
246     [ dup square-coordinates ] dip
247     which-square { } equal?
248     [ dup enemy-direction flip-direction ]
249     [ drop ] if ;
250
251 : move-enemy ( square -- )
252     [ check-enemy-direction ] keep
253     [ enemy-direction ] keep
254     [ { "enemy" } swap suffix ] dip
255     over second move-square ;
256
257 : move-enemies ( -- )
258     enemies
259     [ move-enemy ] each ;
260
261 : directions ( -- directions )
262     { "up" "down" "left" "right" } ;
263
264 : grid-enemy-map ( n -- n' )
265     grid-size get / 1 - random
266     grid-size get * ;
267
268 : grid-height ( -- height )
269     game-height grid-enemy-map ;
```

```
270
271 : grid-width ( -- width )
272     game-width grid-enemy-map ;
273
274 ! Difficulty
275 SYMBOL: difficulty
276 : set-difficulty ( n -- )
277     difficulty set ;
278
279 : increase-difficulty ( -- )
280     difficulty get
281     1 + difficulty set ;
282
283 : reset-difficulty ( -- )
284     0 difficulty set ;
285
286 ! Enemies
287 : setup-enemy ( n -- n' )
288     grid-width grid-height find-square first
289     directions random "enemy" set-square-key
290     dup 0 =
291     [ ] [ 1 - setup-enemy ] if ; recursive
292
293 : setup-enemies ( -- )
294     difficulty get setup-enemy drop ;
295
296 : game-over-text ( -- )
297     "Press Space to Continue \nEscape to Exit"
298     40 get-screen-height 2 /
299     30 BLACK draw-text ;
300 ! States (End, Restart, Start)
301 : draw-game-over ( -- )
302     begin-drawing
303     clear-window
304     game-over-text
305     end-drawing ;
306
307 DEFER: game-loop
308 DEFER: setup
309
310 : game-over-input ( -- )
311     {
312         { KEY_SPACE "space" }
313     }
```

```
314     [ process-input
315       "space" equal?
316       [ reset-difficulty setup game-loop ] when ]
            ↪ each ;
317
318 : game-over-screen ( -- )
319     [ draw-game-over
320      game-over-input window-should-close not ] loop
          ↪ ;
321
322 ! Collisions
323
324 : detect-collision ( -- )
325     find-player-square
326     enemies
327     member?
328     [ game-over-screen ] when ;
329
330 ! Textures and scenery
331 SYMBOL: textures
332 : setup-textures ( -- )
333     { "ground.png" "sidewalk.png" "end.png"
          ↪ "door.png" }
334     [ dup load-image load-texture-from-image
335       [ { } swap suffix ] dip suffix ]
336     map
337     textures set ;
338
339 : get-textures ( key -- texture )
340     textures get at ;
341
342 : draw-ground ( -- )
343     grid get
344     [ square-coordinates [ "ground.png"
          ↪ get-textures ] 2dip
345       RAYWHITE draw-texture ] each ;
346
347 : draw-sidewalks ( -- )
348     0 find-square-by-x
349     game-width find-square-by-x append
350     [ square-coordinates [ "sidewalk.png"
          ↪ get-textures ] 2dip
351       RAYWHITE draw-texture ] each ;
352
```

```
353 SYMBOL: max-level
354
355 : draw-ending-door ( -- )
356     "door.png" get-textures game-width
        ↪ middle-square 0
357     RAYWHITE draw-texture ;
358
359 : draw-winning-end ( -- )
360     difficulty get max-level get =
361     [ 0 find-square-by-y
362       [ square-coordinates [ "end.png" get-textures
          ↪ ] 2dip
363         RAYWHITE draw-texture ] each
364     draw-ending-door ] when ;
365
366 : draw-level-num ( -- )
367     difficulty get number>string
368     10 10 20 RED draw-text ;
369
370 : draw-game-won-text ( -- )
371     "You reached the\nPoet Society!\nSpace to play
        ↪ again."
372     40 get-screen-height 2 /
373     30 BLACK draw-text ;
374
375 : draw-game-won ( -- )
376     begin-drawing
377     clear-window
378     draw-grid
379     draw-ground
380     draw-game-won-text
381     end-drawing ;
382
383 : game-won ( -- )
384     [ draw-game-won game-over-input
385       window-should-close not ] loop ;
386
387 : render ( -- )
388     begin-drawing
389     clear-window
390     draw-grid
391     draw-ground
392     draw-sidewalks
393     draw-winning-end
```

```
394     draw-player
395     draw-enemies
396     draw-level-num
397     end-drawing ;
398
399 : setup ( -- )
400     setup-grid
401     setup-grid-squares
402     setup-player
403     setup-enemies ;
404
405 : ?next-screen ( -- )
406     find-player-square
407     posy>> 0 =
408     [ difficulty get max-level get =
409       [ game-won ]
410       [ increase-difficulty setup game-loop ] if
411     ] when ;
412
413 : game-loop ( -- )
414     [ check-input render move-enemies
415       detect-collision ?next-screen
416       window-should-close not ] loop ;
417
418 : draw-difficulty-box ( -- )
419     "Press 1, 2 or 3 to select\n
420     (1) 3 Levels\n
421     (2) 5 Levels\n
422     (3) 10 Levels\n"
423     40 40
424     30 BLACK draw-text ;
425
426 : render-difficulty-screen ( -- )
427     begin-drawing
428     clear-window
429     draw-difficulty-box
430     end-drawing ;
431
432 : difficulties ( -- n )
433     { { "one" 3 } { "two" 5 } { "three" 10 } } ;
434
435 : check-difficulty-input ( -- )
436     {
437         { KEY_ONE "one" }
```

```
        { KEY_TWO "two" }
        { KEY_THREE "three" }
    } [ process-input dup
        "" equal?
        [ drop ]
        [ difficulties at max-level set ]
        if ] each ;

: choose-difficulty ( -- )
    [ render-difficulty-screen
      check-difficulty-input
      max-level get number? not ] loop ;

! These only need to be initialized once
: pre-setup ( -- )
    setup-textures
    choose-difficulty
    reset-difficulty ;

: reset-max-level ( -- )
    f max-level set ;

: main ( -- )
    make-window
    pre-setup
    setup
    game-loop
    reset-max-level ! While developing you'll want
     ↪ this
    close-window ;

MAIN: main
```

We make our way into the poet society building, narrowly dodging the blows of ruthless illiterates. Inside you hear the muttering of tongues seasoned with guile; a muttering that sounds much like...

Brothers you have heard it said
Lift not the sword but wield the pen
Until today that fact was true
A epithet of mores and rules
The sword did rust inside your sheathe
But now today; the clumsy Keith

He knocked our well of words from ink
And now to might we all must sink
Avast your swords! Avast your bucklers!
All out of ink we curl our knuckles!
Once again, we all blame Keith
Now attack with rage unsheathed!

A myriad of miraculous mysteries confound you as the paltry poets embrace an unfamiliar steel and charge out the front door. But one pair of eyes divert from their battle rage, if only for a moment.

Oh hey, Sir George. Fancy meeting you here.

A jab, a tussle, and a prod all lash out in your general direction; a rather impressive byproduct of fervor emanating from the rather unseemly militia rushing by. Battle cries sound out from shrill voices that have never heard war. A march of shoes only recently conscripted into the Romanticism of revolution thud through the floor.

So the lads and I are headed out. You wouldn't happen to need anything? Paper, ink, the King of England's head?

You nod in frantic disapproval underscored by a confusion that seems much like a mute man trying to teach a phonics course.

Ah, alright then. Take it easy Sir George!

For a moment, peace returns, as the last hasty minded word juggler passes through the doors. Ink lies all over the floor. A mix of paper and latte cups litter the tables contemplating their final destination in the glow of a soft fireplace crackling nearby. Just then the door busts open again.

Oi! That's their leader! Beat him til he becomes an ablative thesaurus of agony. [10]

Capt. Dennis, that was beautiful sir. We'll beat that poet, right we will. We'll be having none of that fancy word play around 'ere!

Their batons lift up before you. You clench your eyes shut hoping to wince away the incoming pain. But just like the mercy of a mild convenience your eyes peer open again; only to see the playful lush of innocent grass dancing by the river.

[10] Try again

5.4 Exercises

Life is hard

Add a new difficulty level to the game. It can be any number, and if you're feeling particularly spiffy you could even add an "unlimited mode" but that might require some extra work. Perhaps you'd have to scale the number of enemies added? In any case add a new selection to the menu when the game starts.

But sometimes easy

Perhaps we'd like to change the difficulty when we fail. Add an option to the game over screen to allow the level to be changed. Here is how the process might look. First we need to be able to recognize a key being input for it, let's say we use 'd' to indicate we want to change it. Then we also need to set the max-level to **f** so the difficulty code will run. After we change the difficulty we have to restart the game, what words have we used to do that?

It's a feature!

In this next chapter we're going to revisit this game again briefly and spend some time looking at what we did right, what we did wrong, and what we can learn for next time. Before we do this it would be a good idea to try to decide what you think about the code. I'll tell you right now, it purposefully does a few things wrong. See if you can find them. What could be cleaned up? What words could be factored better? Is there an architectural issue? What limits does our system have? Learning from your mistakes (or I guess my purposeful ones) is part of the iterative process. We are now much closer to understanding the true solution to making *Mad Dash to Balderdash*, but we only got there by being willing to fall on our own face a bit first. We want solutions that emerge after hacking something together; not solutions that are hacked together til they work.

Chapter 6

The Man in the Teacup

In light of the view that we have only had a brief time with Factor, our primitive attempt at a first game seemed to go rather well. Does it run? Yes it runs, and that would be enough for most people. Some would even be content to stitch snippets of various bits of code together until a game magically popped out. But we don't find any joy in not being able to express exactly how our program works. In-fact if we do say a property of our code includes the ability to say, *"I have no idea how this part works"* then something has failed us. Either the language has not been molded to the problem domain, we have not properly thought out our abstractions or architecture, or we were focused on a pragmatism that evades all understanding. Now I don't look down on people who do program this way. But we can do better can't we? Shouldn't we always be looking up toward a goal of getting better? This chapter will be our introspection towards the greater goal of programming as an art form. An introspection that is the capstone of the iterative process.

Mad Dash Til Balderdash really includes three main architectural issues. Our first issue is in the nature of the *game-loop* itself. Another one is the abstraction handling our input and the contextual response to certain keys. Lastly, there are some limits to our system that don't really need to exist.

6.1 And again and again and again

An issue exists with our game-loop, although not necessarily with the loop itself. Let's look at some of the times we *restart* the game.

```
: game-over-input ( -- )
    {
        { KEY_SPACE "space" }
    }
    [ process-input
      "space" equal?
      [ reset-difficulty setup game-loop ] when ]
       ↪  each ;

: ?next-screen ( -- )
    find-player-square
    posy>> 0 =
    [ difficulty get max-level get =
      [ game-won ]
      [ increase-difficulty setup game-loop ] if
    ] when ;
```

Or in other words we can simplify both cases down to these quotations.

```
[ reset-difficulty setup game-loop ]
! and
[ increase-difficulty setup game-loop ]
```

The difficulty begins where the realization hits that we are currently in a *game-loop* when one of those quotations is called. Instead of finishing the original loop we immediately reset the data and jump into a brand new loop. Technically speaking the more loops we jump into the more old loops we have sitting around, waiting for their chance to close up shop. Now, this is in a way, a method of recursion since the game-loop is defined with potential paths it can take that involve calling itself. Because of our simplistic method of having global symbols, and because each loop refers to those symbols, we are technically only wasting the most minute amount of *RAM* in using this method. But we are still wasting memory. In the future what we really want is a word that takes in a *game-world* on the stack; being a sequence containing the data for our world, and that outputs a new modified game-world. This would give us one singular loop that is always running, and technically make our game more functional. In this case we speak of a program being functional if it limits side effects and is guaranteed to produce the same output with the same input. In this we mean there exists a mapping from input to output that does not change based on external modifiers like we might see in a more imperative program. We

will expound on this in later sections, but for now we want a small taste of what sort of improvements might lie in our road-map.

6.2 The Broken Record

check-input, **check-difficulty-input**, and **game-over-input** are one example of our *broken record.*[1] What does this mean? Well, it means we're *almost* repeating ourselves a bit too much. Here is the pattern we've formed.

```
: check-input-generic ( -- )
    {
        { RAYLIB_KEY  "keystring" }
    } [ process-input
        ! Space where we do something with the
         ↪ input based on what string we receive
    ] each ;
```

First of all, what are the upsides of this pattern? Well the upside is we've expressed our input in a uniformed way such that we can process everything through *process-input* and then deal with strings instead of Raylib keys. It's also very simple, and makes use of higher-order words. That's all well and good, but what is the downside?

It's a pattern.

Before this whole book incites the furor of the numerous orthodox programmers that would view this statement as iconoclasm, I'd like to throw my 2 cents on why I would say this. The issue lies in the nature of programming itself. Languages exist as bridges between our mind and the computer; a sort of isomorphic conversation we are intending to add meaning to. As a disclaimer, surely some patterns exist that are good, some patterns simply tell us there is a reliable way to solve a particular problem. But other patterns, like the one before us, ask a question instead. They ask,

"Is this a pattern I'm using to solve a class of problem, or a limitation of the language?".

If it's to solve a class of problem then perhaps we have decided to place our explorers cap on the nightstand and stick with a well known method. There isn't much to say against this. But if it is a problem of the language, then the bridge between us and the computer has been violated. That is to

[1] There is one other major one too, see if you can find it after reading this section.

say there are two things to make note of here. One is to say that a pattern can quickly turn into the laziness expressed from the failure to properly abstract a problem, and the other is to question if an iron cage has been placed upon us by the language designers. For the former, our job is to recognize when something could be abstracted in a clearer way, and for the later we are at the mercy of the language itself. Luckily, in Factor's case we can circumvent this, as Factor belongs to the class of languages with true macros. That is we can build the language up toward our solution by using macros. Factor will allow us to modify Factor itself in all sorts of ways. One of the simplest ways we could do this is to imagine we had a macro called **with-input**. We could then express the following for our input words.

```
! with-input defined somewhere

: check-input ( sequence logic -- )
    with-input ;

: check-game-over-input ( -- )
    { { KEY_SPACE "space" } }
    [ "space" equal?
      [ restart-game ] when ]
    check-input ;
```

Now we can define *with-input* such that it rearranges a sequence and a quotation containing some logic for handling that sequence into the following.

```
{ input-sequence }
[ process-input
    input-logic ] each
```

In other words, we can Factor out even the scaffolding we want to use and then only create words that focus on the unique data applicable to that particular area of our code. But don't worry if you don't understand this yet, we're going to use macros much more fully in a future game. For now, what we really want to ponder is how we can get out of the trap of *"almost"* repeating ourselves. Later, we will learn that macros can be used in a much more powerful way than simply abstracting patterns out of our code.

6.3 Malnourished Gymnast

6.4 Exercises

Notwithstanding the issues we have just gone over, we're going to create the *Crossing the Thames* game. Or rather, you're going to create it and I'm going to try to guide you through the process. We've already done the much harder game after all! But you'll say, *"What about all these issues?"*. And to that I say, if you think you can solve them in the simpler game, go for it. Else, I think we've realized that the main problem has to do with our abstractions. What we really want is a sort of foundational Vocabulary that lets us express these patterns more sufficiently. This will be our goal going into the next game, and the game after that. We're going to absolutely need to grow Factor to accommodate our way of expressing things in the realm of games so that when we come to our final and most complicated game we are prepared. For now, this is the chance to get more acquainted with the basics of Factor, to play with the stack, and to try to think about how you can Factor out complicated problems into simpler ones. Let's begin!

Half of the battle is a good name

Start off by creating a new Vocabulary to hold the *Crossing the Thames Game*. A name like *fof-thames* would work but you can pick whatever you want. In the interests of brevity, copy the **USING:** line from *fof-poet* to get all the Vocabularies you will need for this game. It is partly up to you to you look at these Vocabularies and see what they do. In-fact it's probably one of the best ways to learn Factor in and of itself.

Trapped in the grid

Crossing the Thames will use the same grid system from our last game. Try to implement it yourself if you can. It's only 7 words. Think about the nature of the grid and how we needed to iterate over the possible points. We also needed a *grid-size*. Finally, there are only two words needed in rendering the grid. Reduce the problem into a problem of rendering one square and you'll be well on your way.

Since we haven't touched on *OOP* yet I'll give you the **TUPLE:** the grid system uses to represent squares.

```
1 TUPLE: square posx posy size container ;
```

Three Tiered with no tears

It's helpful in languages like Factor like give you a REPL to be able to incrementally see your progress as you work on your code. We can do that with this game by implementing our three core words, **setup**, **render**, and **game-loop**. Here's an idea of what each should do.

```
: setup ( -- )
    ! Set global symbols to what their value should
     ↪ be before the game runs
    ;

: render ( -- )
    begin-drawing
    clear-window
    ! Code that handles what we specifically want
     ↪ to draw
    end-drawing  ;

: game-loop ( -- )
    ! Keep the game running perpetually.  Remove
     ↪ the input word until we define it.
     ↪ window-should-close will still allow the
     ↪ 'esc' key as input.
    [ input render
    window-should-close not ] loop ;

: main ( -- )
    setup
    game-loop
    close-window ;
```

If you have completed the last exercise then you should have some words like **setup-grid** and **draw-grid** available to you. Where can you put these in the core words? Once you have these core words available you can slowly add each new section of the game that you complete. For instance, if we were to complete the exercises up to this point and run **main** we should be presented with a *grid* on the screen. This lets us visually see our progress and check if everything is working correctly. When we're done we can hit **ESC** and we'll be right back in the REPL. Try to get the core words defined so you can visibly see your progress in the rest of the exercises.

Where's our map?

With Mad dash til balderdash still in our mind we may forget to contrast the differences between the two games.

Design	**Mad Dash**	**Thames**
Screen changes?	Many, heading up	One single screen
Enemies?	Moving up and down, left and right randomly	Moving from one side of the screen to the other and disappearing
Collectibles?	None	Letters to solve a password
Difficulty?	Changeable	No difficulty system

We see then, that there are some changes. We don't have to worry about the difficulty screen, and we can simplify enemy movement code. We do have to add collectibles and a way to indicate to the player what letters he has collected. Take a moment to try to visualize what the game might look like simply based on these elements.

Chapter 7

Time, Thyme, and Taeme Again

There are those among us who would shun any frank discussion of our origins. An editor of Dopha-mental magazine once spoke on the sordid state of affairs with his keystone argument in regards to history.

"Is that it?"

And it seems thus, that a rather particular public opinion is formed from outlets like The merry melancholy and Blast magazine, who inform us that any attempt at discerning history is merely a glancing blow from the folly of trying to apprehend an intangible past. But since you have just exasperated all your energy in crossing the thames and descended into apathetic helplessness, we relate to you the origins of this world now. You may think this to be the rambling of an illegal narrator, but my only intent is to instill in you an appreciation for the rare event you are about to witness. For as you settle down onto a rashly determined stone seat and glance at your surroundings, you notice the rather intrepid abundance of Thyme waving around you.

It seems you are about to meet, him.

But who is he? That question brings us concurrently back to the beginning of this chapter, and more importantly the beginning of time.

In the beginning, explosive vectors of Chaos tumbled through the night sky. Although this was problematic in the fact that a night sky was not yet around to witness it. However, if it were there it would also witness the cataclysmic forces forming the world below. In due time, Chaos brought forth Order, and Order brought forth Man, which was well and good until

Man brought forth decisions and the subsequent return of Chaos. There is a certain group of men referred to only as the **Butler's of Levram**[1] who choicely consider this the only apparent explanation for universal origins. And while we may never meet them, it suffices to say that their motto is

Share first, Consider later.

which has been both their greatest virtue, and only excuse for why they can't afford their mortgages. But then maybe their Landlords are just uneducated peasants. Perhaps they neglect the distant echos of powerful Chaos dancing through the leaves the Butler's so famously tend to.

Yet there is one more sound among the clamor. The sound of a thousand tongues licking the *Mangy Mane*; a true sign of piety among feline sensibilities. But when they aren't busy dividing themselves over spoiled fish or scratching up the wood flooring, the **Cats of Kalypso** may offer another explanation for our origins.

The Mangy Mane did it.

And that was as far as they got before balkanizing their kingdoms over which side a tuna can should be opened from.

The true explanation for the origin of all things is much less glamorous. It lies rooted in the deepest pit man can mine for meaning. It tugs at the very bindings of all philosophies. Men have been throttled, bashed, abraised, rejected, confuddled, and enchanted by it's very occurence. A thing which can never be caught yet is always seen by those observing closely. An object powering all the hindsight provided by an ounce of reflection. And please do not act unreasonably when I tell you the true explanation for the origin of all things is the awe inspiring tasset of,

Coincidence.

Shame overcomes me at the very mention. It rattles my bones to ponder that fate has a sense of humor. Perhaps, over the sound of life's cackling, you may find your last drink of sanity at the door leading in to the rabbit hole. The dreadfully disappointing thing is, in this case, the truth. For the land of Ez was called forth, reared and raised by an Original Act wrought from love, unmitigated classical physics, and a mildly annoying case of heartburn.

[1] This book does not take place on earth, nor even in this time period, but from the land of Ez. The Author risked life and limb, violating the time-space continuum to snatch this book out of a previous universe and commit cross-world plaguarism.

You may now ask. Ok, but what about the *him* we're supposed to be meeting. You see he was once a being from Ezreal. Although, it would be incorrect to use *was* for he neither *was* nor *will be* but always is. His true name is in the tongue of the Ezrealians, and would take nearly a whole lifetime to speak incomprehensibly. Foreseeing this problem, the ancients provided us with a name for our own manner of tongue, and in English[2] he is called, *Taeme*, but some refer to him as *Kevin*.

To truly understand *Taeme* it is necessary to remember the **Original Act**. The issue with this, is that English only provides three tenses. A past, present, and future. But all of these are properly bound to time and are insufficient to describe *before time*. Yet, we will try.

Before time began, *Taeme* existed in Ezreal. Ezreal was a noble place, and *Taeme* was a noble creature, but subservient to a higher good. Much like a mirror can reflect light, but is not the source of the light itself, so was *Taeme* also a reflection of his master. It is helpful to remind ourselves that outside of time, one does not look forward to, nor regret anything, for one is always experiencing simultaneously all the things he *has done and will do*. Inconveniently, *Taeme* was currently experiencing this fact in regards to all the spicy food he had ever and will ever ingest. But while there is no time for regret in Ezreal; heartburn was another matter entirely. It was in this state thus that *Taeme* ventured to find some solitude, only to inadvertently wander into his Master's gaze. It was here that we can first adequately describe the statement, *in a moment*. For, in a moment, *Taeme* caught, not a reflection of his master, but an unmitigated dose of untamed beauty. Emanating from his Master, waves of the purest beauty that will ever exist raptured themselves into Taeme's eyes. It was a moment of exquisite delicacy that would make even Plato and his ideal forms blush. Yet, an undeniably unique and colossal rift begat itself into existence as Taeme's heart skipped not one, but two beats. His heart, leaving the synchronous nature of the absence of time was cast back. It landed exactly one second, not before time began for Taeme already existed in this state, but before the void of un-time began. Of course, our endeavour into this matter quickly becomes hampered by the fact that no one really understands Ezrealian metaphysics.[3] But it will suffice to say that *Taeme* is locked in the eternal marathon of trying to catch up to a lost moment in un-time. And while Taeme finds it somewhat pleasant to eternally chase the *Original Act*,

[2] Also know as Ez Common

[3] There are those who claim to though. Their commentary can be found in another book the Author has plagiarized from their world.

we experience his actions in the shocking realization that

> **Time exists.**

For it would not, if Taeme were not running behind it and pushing it forward a moment at a time. However, while we have generated a slight rapport and understanding of Taeme's predicament, we explain all this only to relate that Taeme is,

> really annoying to talk to.

Successfully warned, you begin to realize that you are not alone in this patch of the forest. A small man emerges from the thrush in an energetic hop and with boundless joy frolics around the Thyme dotted around the area.

> **Ah, the guest house? That old thing got torn down last year.**

He looks friendly so you ask him if he knows where the guest house might be. Wait?

> **My name? Why I'm Taeme of course. Inevitable. Never late for a party, although I am always running late.**

A little confused over his previous answer, you ask him what his name is.

> **Whats going on? Ah where are my manners. It's rather hard for me not to answer your question before you say it, what with the, being locked behind time and all.**

Your head starts to hurt, and in your tired demeanor you blot out a question about what the heck is going on.

> **Sleep? Why, why would you want to pass the time that way? But if you really insist fellow, I suppose I could run backwards.**

You rub your eyes and tell him that you really don't understand what's going on but perhaps if you got some sleep your head would stop hurting.

> **What do you mean what does it have to do with anything? You see if I run backwards we can go back to when the guest house existed.**

You ask him what that has to do with anything.

> **Destroying the world? Nonsense. I mean I've never done it before but it'd make sense to me. It'll just be like time had a little hiccup along the way.**

In your exhaustion you inadvertently tell him that reversing time could have dire consequences for the world.

> **Off I go! Wait, gotta make sure I do this right. Carry the one, divide by zero. Ah, ah! Got it. Happy meeting you, catch you later! Or I suppose, catch you before!**

You yell at him to stop before he runs off but it's too late. It was always too late. And now it's too early. For in a moment you watch time quite literally rewind the atmosphere around you like a hamster working his calves on a treadmill connected to a film projector. The trees dance across the landscape, as they get brought in by lumberjacks, set up, shrink, bloom, and finally disappear into the ground. Buildings appreciate in value as they quickly repair themselves only to witness deconstruction men tear them down again. You see a Kingdom appear with elderly men growing younger as Kings are decrowned, rethroned, and unassassinated. Wise men are ignored as you witness the culture return to primitive sensibilities. You consider if time has returned to normal. One last blink and before you stands a rather imposing offense to all social mores. Time has finally resumed normally, but when? You wonder just how far back Taeme just threw you. Before you a large dump of various trash lies erected, being held up by a run down structure you assume to be the main office to a recycling center. That is until your eyes refocus and spot the sign hanging from one nail on the transom.

The Clever Bean Bistro

As your mind tries to make sense of the disgusting prospect of running a restaurant next to a dump, your stomach rumbles.

7.1 Clever Bean Bistro

The next game were going to make is Clever Bean which tells the rather unfortunate story of an unreasonable chef.