FIT2102 Programming Paradigms

Tim Dwyer, 2018

Synopsis

Ability to code in differently constructed programming languages is analogous to speaking in different natural languages with varying grammars. Similar to natural languages, programming languages from different paradigms (styles) vary in their expressiveness and efficiency. One programming language may require many screens-full of complex code to accomplish a task for which another requires but a few expressive lines of code. Therefore, understanding the design principles of programming languages enables computational problems to be implemented in dramatically different and powerful ways; leading, in some cases, to solutions that are more elegant, correct, maintainable, efficient and/or extensible.

This unit examines a selection of programming languages and paradigms and explores the evolution of language design from low-level paradigms that are closer to the execution model of the machine, to more high-level declarative paradigms that allow programmers to model a problem precisely rather than specify its solution. The unit covers paradigms such as functional and logic programming, comparing and contrasting them to programming styles that students are already familiar with, including object-oriented, imperative and procedural programming paradigms. Topics include specification and data-modeling techniques (covering types and polymorphism, mutability-versus-purity, state management, and side-effects) and different models of execution such as strict-versus-lazy evaluation.

The unit provides practical experience with a variety of non-procedural, non-object-oriented programming languages and discusses the influence of programming language theory on the design of current mainstream computer languages, and how the theory translates to practice.

Outcomes

At the completion of this unit, students should be able to:

1. describe the major attributes used to describe programming languages;
2. describe the major features, strengths and weaknesses of important programming languages in the context of their historical development;
3. analyse and critique past, present and future programming languages;
4. evaluate the suitability of different paradigms for different problem types;
5. design and implement simple programs in several programming languages of different paradigms and demonstrate an ability to solve more complex problems in at least one non-procedural paradigm;
6. describe the theoretical aspects of modern programming paradigms and apply this theory to analysis and design of programs.

Schedule (Calendar)

Week 1:

- Intro
    - Syntax vs Semantics
    - Levels of Abstraction
- Assembler - the machine level
    - Von Neumann Architecture
    - Instruction Execution Cycle
- JavaScript and functions
    - Recursion
    - Immutable variables

Week 2:

- JavaScript, objects and functions
- Function purity

Week 3:

- Functional Programming
- TypeScript
- Linked lists and Church-encoding lists with functions

Week 4:

- Lazy lists
- FRP
- Assignment 1

Week 5:

- Higher-order Functions
- Combinators / Currying / Lambda Calculus

Week 6:

- Purescript and Haskell

Week 7:

- Assignment 1 due
- Haskell!

Week 8:

- Assignment 2
- Point-Free style
- Functor and Applicative

Week 9:

- Haskell
- Foldable
- Traversable

Week 10 (Guest lecture):

- Haskell

Week 11:

- Assignment 2 due Sunday, October 15th

Week 12:

- Constraint Programming - MiniZinc

1. Introduction: Levels of Abstraction

As a branch of Computer Science, the theory and practice of programming has grown from a very practical need: to create tools to help us get computers to perform useful and often complex tasks. Programming languages are tools, that are designed and built by people to make life easier in this endeavour. Furthermore, they are rapidly evolving tools that have grown in subtlety and complexity over the many decades to take advantage of changing computer hardware and to take advantage of different ways to model computation.

An important distinction to make when considering different programming languages, is between *syntax* and *semantics*. The syntax of a programming language is the set of *symbols* and rules for combining them (the *grammar*) into a correctly structured program. These rules are often arbitrary and chosen for historical reasons, ease of implementation or even aesthetics. An example of a syntactic design choice is the use of indentation to denote block structure or symbols such as “BEGIN” and “END” or “{” and “}”.

```python
# python code:
def sumTo(n):
    sum = 0
    for i in range(0,n):
        sum = sum + i
    return sum
```

```c
// C code:
int sumTo(int n) {
    int sum = 0;
    for(int i = 0; i < n; i++) {
        sum += i;
    }
    return sum;
}
```

Example: functions in python and C that are syntactically different, but semantically identical.

By contrast, the “semantics” of a programming language relate to the meaning of a program: how does it structure data? How does it execute or evaluate?

In this course we will certainly be studying the syntax of the different languages we encounter. It is necessary to understand syntax in order to correctly compose programs that a compiler or interpreter can make sense of. The syntactic choices made by language designers are also often interesting in their own right and can have significant impact on the ease with which we can create or read programs. However, arguably the more profound learning outcome from this course should be an appreciation for the semantics of programming and how different languages lend themselves to fundamentally different approaches to programming, with different abstractions for modelling problems and different ways of executing and evaluating. Hence, we will be considering several languages that support quite different programming *paradigms*.

For example, as we move forward we will see that C programs vary from the underlying machine language mostly syntactically. C abstracts certain details and particulars of the machine architecture and has much more efficient syntax than Assembly language, but the semantics of C are not so far removed from Assembler - especially modern assembler that supports procedures and conveniences for dealing with arrays.

By contrast, Haskell and MiniZinc (which we will be exploring in the second half of this course) represent quite different paradigms where the underlying machine architecture, and even the mode of execution, is significantly abstracted away. MiniZinc, in particular, is completely *declarative* in the sense that the programmer's job is to define and model the constraints of a problem. The approach to finding the solution, in fact any algorithmic details, are completely hidden.

One other important concept that we try to convey in this course is that while some languages are engineered to support particular paradigms (such as functional programming or logic programming), the ideas can be brought to many different programming languages. For example, we begin learning about functional programming in JavaScript (actually TypeScript and EcmaScript 2017) and we will demonstrate that functional programming style is not only possible in this language, but brings with it a number of benefits.

Later, we will pivot from JavaScript (briefly) to PureScript, a haskell-like language that compiles to JavaScript. We will see that, while PureScript syntax is very different to Javascript, the JavaScript generated by the PureScript compiler is not so different to the way we implemented functional paradigms manually in JavaScript.

Then we will dive a little-more deeply into a language that more completely "buys into" the functional paradigm: Haskell. As well as having a syntax that makes functional programming very clean, the haskell compiler strictly enforces *purity* and makes the interesting choice of being *lazy* by default.

In summary, the languages we will study (in varying degrees of depth) will be Assembler, C/C++, JavaScript (ES2017 and TypeScript), PureScript, Haskell and MiniZinc, with

JavaScript/TypeScript and Haskell being the main languages explored in problem sets and assignments. Thus, this course will be a tour through programming paradigms that represent different levels of abstraction from the underlying machine architecture. To begin, we spend just a little time at the level of least abstraction: the hardware itself.

2. The Machine Level

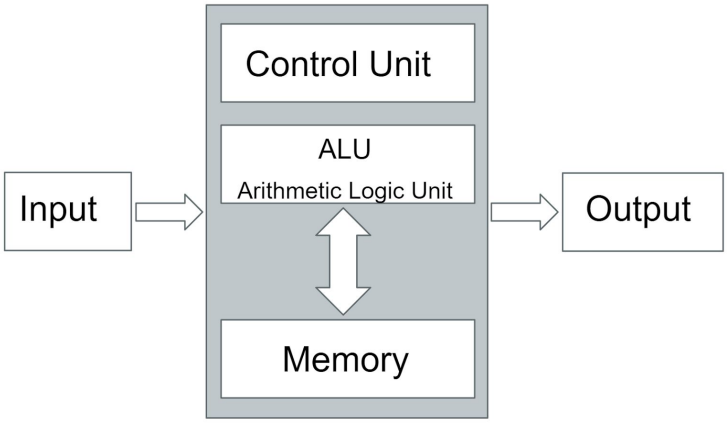
Learning Outcomes:

- Explain the need for abstraction from machine instructions to high-level languages
- Explain how assembly instructions work with registers and memory to perform computation
- Explain how assembly programs are structured into subroutines through jumps
- *[not for 2018 - Create a basic x86 assembly program to perform a computation]*

Conceptually, modern computer architecture deviates little from the *von Neumann model* proposed in 1945 by Hungarian-American computer scientist John von Neumann. The development of the von Neumann model occurred shortly after the introduction of the theoretical *Turing Machine* concept.

The von Neumann architecture was among the first to unify the concepts of *data* and *programs*. That is, in a von Neumann architecture, a program is just data that can be loaded into memory. A program is a list of instructions that read data from memory, manipulate it, and then write it back to memory. This is much more flexible than previous computer designs which had stored the programs separately in a fixed (read-only) manner.

The classic von Neumann model looks like so:



At a high-level, standard modern computer architecture still fits within this model. The following diagram is adapted from *Assembly Language for x86 Processors* by Irvine.

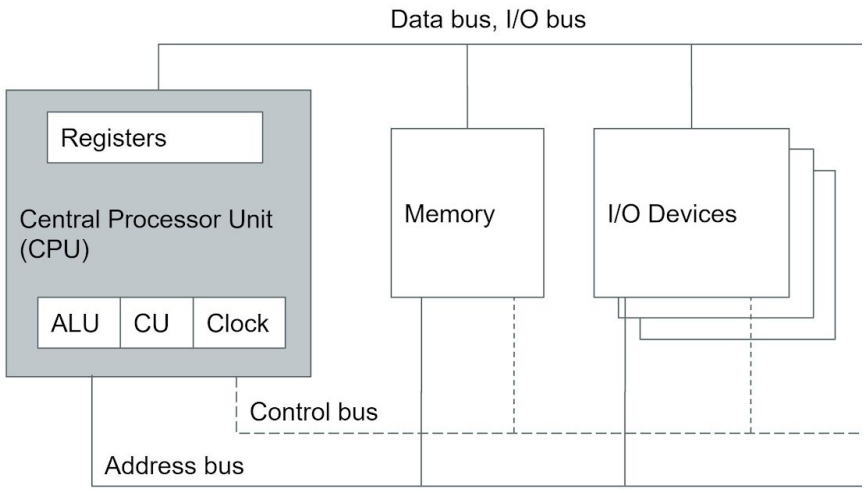
ALU - Arithmetic Logic Unit

CU - Control Unit

Clock - Synchronises CPU operations with other system components

Data bus - transfers instructions and operations between CPU and memory

Address bus - holds addresses of instructions and data when the currently executing instruction



Programs run on an x86 machine according to the *Instruction Execution Cycle*

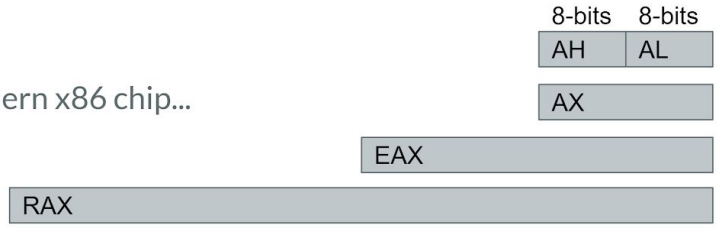
1. CPU fetches instruction from instruction queue, increments instruction pointer
2. CPU decodes instruction and decides if it has operands
3. If necessary, CPU fetches operands from registers or memory
4. CPU executes the instruction
5. If necessary, CPU stores result, sets status flags, etc.

Registers are locations on the CPU with very low-latency access due to their physical proximity to the execution engine. Modern x86 processors also have 2 or 3 levels of *cache memory* physically on-board the CPU. Accessing cache memory is slower than accessing registers (with all sorts of very complicated special cases) but still many times faster than accessing main memory. The CPU handles the movement of instructions and data between levels of cache memory and main memory automatically, cleverly and—for the most part—transparently. To cut a long story short, it can be very difficult to predict how long a particular memory access will take. Probably, accessing small amounts of memory repeatedly will be cached at a high-level and therefore fast.

NOTE: SECTIONS HIGHLIGHTED IN BLUE ARE NOT CORE OR EXAMINABLE IN FIT2102 IN 2019. READ THEM ONLY IF YOU ARE INTERESTED.

x86 Data Registers

Just some of the registers on a modern x86 chip...



Name	64-Bit	32-Bit	16-Bit	8-Bit (High)	8-Bit (Low)
Accumulator	RAX	EAX	AX	AH	AL
Base	RBX	EBX	BX	BH	BL
Counter	RCX	ECX	CX	CH	CL
Data	RDX	EDX	DX	DH	DL

We will work only in 32-bit mode using the registers highlighted.

The above are “general” purpose registers, although when programming in Assembler you need to be aware of how different instructions use them. Some additional registers are:

ESP - addresses data on the stack (a special memory structure for functions)
EBP - used by high-level languages to reference function parameters and local variables on the stack
EIP - instruction pointer
Flags - register, made up of individually flag bits that control modes or show status

Lots more available in special modes that we won’t worry about!
(MMX, XMM, FPU, etc...)

Let’s consider a small MASM Assembler program:

```
.386                                    ; choose 32 bit mode
.model flat, stdcall                    ; execution model (don't worry about it for now)
ExitProcess PROTO, dwExitCode:DWORD
                                        ; declares this special external procedure and the type of its parameter
```

```
.code                                   ; opens the code section
main PROC                               ; here's where our "main" procedure begins
        mov eax, 5                      ; move 5 into eax register
        add eax, 6                      ; add 6 to the value in the eax register
        invoke ExitProcess, eax         ; exit returning eax as the result code
main ENDP

END main
```

Text to the right of a ';' is my comments. The program really begins after the .code declaration. The instructions have the following structure:

[label:] mnemonic [operands] [; comment]

The optional label is used as the target for jump instructions. The mnemonic identifies the type of instruction. Different instructions expect different sets of operands (usually 1 or 2), these can be the numeric literals, names of registers, or labels corresponding to different memory locations.

There are many different instructions available on modern x86 processors. Here is a small subset that we will use to make some simple programs:

Mnemonic	**Operands**	**Description**
mov	dest, src	Move (copy) contents of src into dest
add	y, x	Add contents of x to y (result in y)
sub	y, x	Subtract x from y (result in y)
mul	x	Multiply eax by x (result in eax)
div	x	Divide eax by x, result in eax, remainder in edx
xor	y, x	Bitwise exclusive or (Tip, to zero a register: xor eax, eax)
jmp	label	Jump unconditionally to label
loop	label	If ecx > 0 jump to label and decrement ecx
cmp	x, y	Compare x and y and set the flags register accordingly
je	label	Jump to label if result of previous cmp operation was equal
j[ne|l|le|g|...]	label	Jump to label if ↑ not equal, <, <=, >, etc...

push	x	Push x onto the top of the stack, decrement esp
pop	x	Take top value off stack, put into x and increment esp

The final convenience functions of MASM assembler that we will use are declarations of named variables declared in a .data section (these just correspond to places in memory), and procedures. Here is a slightly more complicated example program using these features:

```
.386
.model flat, stdcall
ExitProcess PROTO, dwExitCode:DWORD

.data
        x DWORD 1                       ; a DWORD is a "double word"
        y DWORD 2                       ; a word is two bytes
        z DWORD 3                       ; thus a dword is 32 bits
.code
main PROC
        mov ebx, x
        mov ecx, y
        mov eax, z
        call sum3                       ; call the procedure
        invoke ExitProcess, eax         ; result is 1+2+3=6
main ENDP

sum3 PROC
        add eax, ebx
        add eax, ecx
        ret                             ; need to end procedures (other than main) in ret
sum3 ENDP

END main
```

You now have a sufficient subset of the 32-bit mode x86 instruction set to make some non-trivial programs.

Exercises:

2.1. Project Euler Problem 1 is:
Find the sum all of numbers divisible by three and five under 1000.
Write an assembler program to solve this problem, return the answer as the result code (as the program above is doing).

2.2. There is more than one way to solve this problem. Try to find a second way. Hint: one way requires fewer iterations than the other, but is slightly more complex.

2.3. How does programming in Assembly compare to higher-level languages you are familiar with? Are there both advantages and disadvantages? Make a list of both.

NOTE: SECTIONS HIGHLIGHTED IN BLUE ARE NOT CORE OR EXAMINABLE IN FIT2102 IN 2019. READ THEM ONLY IF YOU ARE INTERESTED.

3. C and C++ functions and memory management

Learning Outcomes:

- Explain in basic terms how C and C++ instructions are compiled to machine language and relate this to Assembler code
- Explain how C and C++ differentiate Stack and Heap memory allocation
- Apply strategies for memory management such as RAII and smart pointers
- Explain the basic principles of memory management used in higher-level languages such as reference counting and garbage collection

History of C

C was developed from 1969 to 1973 at Bell Labs by Dennis Ritchie (who was kind of a big deal computer scientist). We are not going to study C in much depth in this course, but if you are interested, the definitive text for C is (and probably will remain, because plain C has evolved little in decades) “The C Programming Language” by Kernighan and Ritchie.

The Unix operating system kernel was written in C making it among the first languages to be used for system level programming. C’s fortunes rose with the popularity of Unix and it lives on in many places, none-the-least-of-them being the Linux kernel and associated drivers on everything from toasters to tablet computers. Its syntax lives on in modern languages like
Java (meaning you will probably understand the program below even if this is your first time looking at C), C#, JavaScript (as we will see) and many others.

C is *imperative* in the sense of being constructed of a sequence of statements that change a program’s state. In the early 1970s C was not the most advanced language. LISP and Scheme were already demonstrating higher-level concepts that we will be discussing later in this unit. However, as we have seen, C outlived these languages in common usage. Perhaps a good part of its appeal, especially for systems-level programming, is that the translation to machine code (optimising compilers aside) is relatively straight-forward.

C has not changed much since the early days. It evolved a little to iron out some early kinks but was standardised by ANSI in the 80s. It is a small language but expressive enough to do some powerful things (such as build operating systems). On the other hand, engineers in the late 70s, while they loved its simplicity and closeness to the hardware, recognised the value of the *Object Oriented* programming style becoming popular in languages like Simula. OO, as you should be aware from Java, is a paradigm for modelling complex processes by structuring the functions and data hierarchically from abstract, general types (*classes*), to very concrete specific types. Thus, C++ was developed by Bjarne Stroustrup as a strict superset of C incorporating object-oriented features but retaining the ability to write code

that compiles without too many surprises to machine code. It is not necessary for this course, but if you want to be an expert C++ programmer, Stroustrup's book "The C++ Programming Language" remains the definitive text.

We are only going to look briefly at C and C++ as examples of the imperative and OO paradigms, examining only a few features, but with sufficient depth to gain some understanding of how their programs are transformed into machine instructions, the advantages they provide compared to coding directly in Assembly, but also some of the issues that cause C and C++ programmers angst.

Looking at simple C code under the microscope

We're not going to try to become expert C programmers.
Consider the following, straight-forward C solution for Project Euler Problem 1.

```
unsigned int EulerProb1(unsigned int max) {
    unsigned int sum = 0;
    for (
        unsigned int i = 0;// it's not standard to separate for statements across multiple
        i < max;           // lines like this but each of these sub-statements corresponds
        ++i                // to multiple  machine operations, as we'll see below.
   ) {
        if (i % 3 == 0 || i % 5 == 0) {
            sum += i;
        }
    }
    return sum;
}
int main()
{
    unsigned int max = 1000;
    return EulerProb1(max);
}
```

The C compiler translates this code into machine instructions which it is instructive to see *disassembled* in the Visual Studio debugger.

Viewing Dissassembly
Calling a __cdecl function
x86 Functions and Stack Frames

```
unsigned int EulerProb1(unsigned int max) {
                                    : the following 10 lines of code are just setting up the stack frame
```

```
01371A70       push    ebp             ; store the address of the calling function
01371A71       mov     ebp,esp         ; make the "base pointer" point to the top of the stack
01371A73       sub     esp,0D8h        ; move stack pointer past parameter and local variable declarations
01371A79       push    ebx             ; store these registers so we can overwrite them locally
01371A7A       push    esi
01371A7B       push    edi
01371A7C       lea     edi,[ebp-0D8h]  ; the following few instructions are initialising a block of memory
01371A82       mov     ecx,36h         ; at the start of the stack frame - not really necessary in this function
01371A87       mov     eax,0CCCCCCCCh
01371A8C       rep stos    dword ptr es:[edi]
      unsigned int sum = 0;
01371A8E       mov     dword ptr [sum],0 ; sum is at ptr [sum] (somewhere between esp and ebp, to find out: &sum)
      unsigned int i = 0;
01371A95       mov     dword ptr [ebp-14h],0          ; i is at [ebp-14h]
01371A9C       jmp     EulerProb1+37h (01371AA7h)     ; jump past the test (why did it reorder?)
      ++i
01371A9E       mov     eax,dword ptr [ebp-14h]        ; eax <- i
01371AA1       add     eax,1                          ; increment
01371AA4       mov     dword ptr [ebp-14h],eax        ; store incremented i
      i < max;
01371AA7       mov     eax,dword ptr [ebp-14h]        ; eax <- i
01371AAA       cmp     eax,dword ptr [max]            ; compare with max (use &max in watch window to get actual address)
01371AAD       jge     EulerProb1+68h (01371AD8h)     ; all done!
      if (i % 3 == 0 || i % 5 == 0) {
00B81AAF       mov     eax,dword ptr [ebp-14h]        ; eax <- i
00B81AB2       xor     edx,edx                        ; clear edx
00B81AB4       mov     ecx,3
00B81AB9       div     eax,ecx                        ; i%3 (result in edx)
00B81ABB       test    edx,edx                        ; == 0?  (Equivalent to: AND edx, edx)
00B81ABD       je      EulerProb1+5Fh (0B81ACFh)      ; do it!
00B81ABF       mov     eax,dword ptr [ebp-14h]        ; eax <- i
00B81AC2       xor     edx,edx                        ; clear edx
00B81AC4       mov     ecx,5
00B81AC9       div     eax,ecx                        ; i%5
00B81ACB       test    edx,edx
00B81ACD       jne     EulerProb1+68h (0B81AD8h)      ; don't do it!
       sum += i;
00B81ACF       mov     eax,dword ptr [sum]            ; eax <- sum
00B81AD2       add     eax,dword ptr [ebp-14h]        ; eax <- eax + i
00B81AD5       mov     dword ptr [sum],eax            ; sum <- eax
00B81AD8       jmp     EulerProb1+2Eh (0B81A9Eh)       ; continue loop
      return sum;
00B81ADA       mov     eax,dword ptr [sum]            ; return result in eax
}
00B81ADD       pop     edi                            ; restore registers from stack
00B81ADE       pop     esi
00B81ADF       pop     ebx
00B81AE0       mov     esp,ebp
00B81AE2       pop     ebp
00B81AE3       ret
```

Recall that the **esp** register is the *stack pointer*. The stack is used to keep track of function calls. Function calls can be nested many times, until the stack (whose size is set---by the compiler---in the executable program file's preamble) is full. When the stack fills the program will crash with a *stack overflow* error.

Before the function call the function args (e.g. max) are pushed onto the stack. In the function preamble (above) we see that the **ebp** register is used to point to the start of the function's stack frame while **esp** is advanced to the top of the stack so it can be used for local pushes and pops within the function. Function-level local variables and parameters to the function are dereferenced relative to **ebp**. At the end of the function, the result is returned in **eax** and the state of the registers prior to the function call is restored from the stack.

From the above, it should be clear that there is a bunch of stuff happening that is not strictly necessary for the operation of this function, but is there as boiler plate to support the stack frame structure. It might be tempting to assume from this that it is easy to write tighter code in assembly than the compiler generates. However, the above disassembly was generated when compiling and running in "Debug" mode. It may be instructive to repeat this exercise in "Release" mode which causes the compiler to apply optimisations - the simplest being to use registers for local variables wherever possible.

Exercises:
3.1. Compile the above code in release mode and look at the disassembly carefully in the debugger. What are the differences?

3.2. Try using the other approach (as implemented last week in Assembler) to solving this problem in C code. Try to understand the disassembly and document it line-by-line like the example above.

The point of all this, is that to program in Assembler, you need to understand how the processor works, you need to think like the machine. Switching to targeting programs for different platforms (such as the ARM processor on your phone) may require a whole different set of processor-specific expertise. Languages like C take you a step higher and the compiler takes care of the finicky machine-level details. You still need to be aware (vaguely) of how local variables are stored on the stack and that the stack is finite.

Exercises:
3.3. Write a simple recursive C function to compute the sum of positive integers up to a given n

3.4. What's the biggest n you can specify before the stack overflow crash occurs (make sure you are running in debug mode)?

3.5. If the function is not already tail-recursive then rearrange it to make it so (the recursive call must be the very last operation in the function). The tail-recursive version will still crash in debug mode for large n, but try it again in Release mode. What happens? Why?

Heap vs Stack: C++ Memory Management

We've seen in the disassembly above how the local variables live in a block at the start of the stack frame. Here's a really simple way to overflow the stack:

```
void fun() {
    int a[1000000];
}
```

We tried to create an array of int of size 1000000 on the stack. You can easily do the math to figure out precisely how many bytes this is. It's not a huge amount of memory by modern standards, so clearly the stack is not, by default, intended for this kind of use.

To avoid overflowing the stack you need to know how to dynamically request additional memory from the operating system (this is called *heap* memory). You also need to tell the OS when you are ready to release that heap memory. In C++, this is done with the **new** and **delete** operators.

```
void fun() {
    int* a = new int[1000000];
}
```

An int* is a memory address where an int (or the start of an array of ints) is stored. We also call this a *pointer* since it points to somewhere in memory. This program should not crash because we asked the operating system for the memory nicely using **new** and it gave us back a chunk of heap memory that it nothing better to do with. However, this program has a problem. After the function returns, the heap memory is still allocated to our program, even though the pointer a - which was created in the function's stack frame as four bytes to hold the memory address - is gone. Worse, if we call this function (or something less trivial but being equally sloppy with memory) repeatedly, we will grab a new chunk of memory every time and not give it back until the whole program completes. Called in a loop, it could easily consume all the memory available to the system and bad stuff will happen (don't try this!). This is called a *memory leak.*

The following is the polite way to return the memory when we are finished with it:

```
void fun(size_t n) {
    // arrays created on the stack must have a fixed size
    // i.e. the size of the array must be known at compile time.
    // new does not have this restriction, hence we can pass
    // in the size as a parameter to the
    // function as above.
    int* a = new int[n];
    // … do something with a, load some data into it,
```

```
    //sort it, write it back out to a file, etc.
    delete [] a;
}
```

It seems straightforward, but in real code of any complexity, it's very easy to forget to give back the memory when you are done with it. Also, even if we do try to be a good citizen and give delete the heap memory when we are finished with it, we may never get there due to an *exception* being thrown (C++ has no support for a **finally** clause in try-catch exception handling). Also, small memory leaks often go unnoticed for years, even in production systems. This may not be a serious problem, until one day the apparently robust code is reused in a slightly different way, e.g. called repeatedly in a long running background process, or applied to bigger data.

RAII pattern

Hence, in C++, good programmers typically use a pattern called *Resource Acquisition is Initialisation* (or *RAII*) to manage resources. To demonstrate it, we need to tap into a couple of C++'s object-oriented features to build our own array type that has some distinct advantages over the built in one.

```
class SafeArray {
    int* a;
public:
    SafeArray(size_t n) : a(new int[n]) {}
    int& operator[](size_t i) {
        return a[i];
    }
    ~SafeArray() {
        delete [] a;
    }
};
void fun(size_t n) {
    SafeArray sa(n);
    sa[0] = 1; // Do stuff with the array, just like it's on the stack
    std::cout << sa[0] << std::endl;
}  // When sa is removed from the stack, ~SafeArray() cleans the heap
```

So the above includes lots of idiomatic C++ and for the first time our code may not be immediately understandable to a Java programmer. First, SafeArray is a class with public and private *members.* The array pointer is private (since it appears above the **public** keyword). This is good because we don't want anyone outside our class members to mess with it directly. Next up we have the public *constructor* which creates the underlying array on the heap using **new**.

Next up, we overload the square-bracket operators that are usually used for arrays to allow us to both get and set values at specific positions in the array. The & in the return type means that we return a *reference* to the value, rather than the value itself. Unlike a pointer, this is transparent syntactically to the caller. However, it means that the reference can be used as an *lvalue*, i.e. it can be used on the left-hand side of an = assignment.

Exercise:
3.6. Make SafeArray safer still to avoid accessing beyond the end of the array.

Finally, we declare a *destructor* for the class: ~SafeArray. This is where the magic happens, we've told SafeArray how to clean up after itself!

Now, to use SafeArray, we create a SafeArray instance on the stack - but it occupies very little space there, the only data stored locally in a SafeArray object is the int pointer which is the size of a single memory address (in 32 bit mode this is 4 bytes). Behind the scenes, the constructor initialises the array of the requested size in heap memory. We can go ahead and use `sa` as if it's a local array on the stack - but actually reading and writing to heap memory. Then, (and this is the real magic), when sa goes out of scope (at the end of the function), as the stack frame is being cleaned up, the StackArray destructor will be called, which releases the heap memory and we have no memory leak.

The RAII pattern is used all the time by STL classes (the C++ standard library), so you can write fairly clean looking code that is actually managing memory effectively. On the other hand, the power to use pointers to manipulate memory locations remains in C++ as both a blessing and a curse.

Reference Counting and Garbage Collection
There are other methods of memory management available in C++. The STL provides other smart pointer types like std::shared_ptr. This is managed through *reference counting.* Any type may be wrapped in a shared_ptr. Copies of the shared_ptr are shallow (meaning all copies point to the same wrapped data object) and a count of all the copies is maintained. When the final copy is deleted (or goes out of scope somewhere) the underlying object is deleted.

Also, not part of the STL (yet) but available in various C++ libraries is *garbage collection*. Garbage collection is probably the most sophisticated memory management technique, although it was introduced ridiculously early, with LISP in 1958. Similar to reference counting, the number of copies of objects still in use is maintained. However, rather than deleting the object as soon as its reference count goes to 0, a separate run-time operation performs periodic sweeps to cleanup objects no longer in use. This may cause periodic pauses in execution of the main program. A recent trend is to use reference counting by default instead of garbage collection to achieve similarly transparent memory management with more predictable runtime behaviour, e.g. SWIFT does this.

Both *reference counting* and *garbage collection* are "set and forget" paradigms. In C++ they require using special library functions hence adding syntax to your program. An advantage of C++ is that you are not wedded to any one paradigm. You can choose one or do it all yourself as we have seen above. Languages like Java (and JavaScript) rely on garbage collection to keep memory management as simple (and hidden) as possible. Memory leaks are still possible though, and other resources (such as event handlers, threads, file handles, and so on) can also be leaked, so techniques like RAII are still applicable in higher-level languages.

C++ is a powerful, multi-paradigm programming language that allows system programs to have precise control, or complex application builders to use libraries like STL to rapidly build systems with relatively clean code and leave all the complexity hidden in the libraries. In the 90s C++ really had its heyday, being used to build Windows, office suites, games and everything in between. However, the power to “shoot yourself in the foot”, albeit with a beautiful and very clever gun, worried many people. In the real-world, not every programmer has the time to master a programming language as sophisticated as C++ had become and even the best programmers could still make mistakes. The other problem with C++ is that the compiled executable programs tend to be large. Even the simplest program that uses the STL will be large, due to the way C++ handles generic types (through templates). With the advent of the web people wanted more interactive web pages but embedding C++ code in web pages was obviously *a bad idea*.

4. JavaScript: functions are just objects

Learning Outcomes

- Explain the relationship between javascript functions and objects
- Compare arrow functions and regular function syntax
- Create and apply anonymous functions to fluent style code
- Explain JavaScript's prototype scheme for creating classes from functions
- Create ES6 style classes with constructors and getters
- Compare object oriented polymorphism to dependency injection through functions

Some Context

In the late 90s the mood was right for a language that was small and simple and with executable files small enough to be distributed over the web. Originally Java was meant to be that language but, while it quickly gained traction as a language for building general purpose applications and server-side middleware, it never really took off in the browser. Something even simpler, and better integrated with the Document Object Model (DOM) of HTML pages was required to add basic interaction to web pages.

Brendan Eich was hired by Netscape in 1995 to integrate a Scheme interpreter into their browser for this purpose. No messy deployment of Java bytecode bundles - the browser would have been able to run Scheme scripts imbedded directly into web pages. This would have been awesome. Unfortunately, for reasons that were largely political and marketing related, it was felt that something more superficially resembling Java was required. Thus, Eich created a prototype scripting language in 2 weeks that eventually became JavaScript. As we will see, it is syntactically familiar for Java developers. Under the hood, however, it follows quite a different paradigm.

The fact it was initially rushed to market, the fact that browser makers seemingly had difficulty early on making standards-compliant implementations, and a couple of regrettable decisions at first regarding things like scoping semantics, meant that JavaScript developed something of a bad name. It's also possible that there was some inherent snobbiness amongst computer science types that, since JavaScript was not a compiled language, it must inevitably lead to armageddon. Somehow, however, it survived and begat the "web 2.0" phenomenon of what we now refer to as rich, client-side "web apps". It has also matured and, with the EcmaScript 6 (ES6) and up versions, has actually become quite an elegant little multiparadigm language.

JavaScript 101

The following introduction to JavaScript assumes a reasonable knowledge of programming in another imperative language such as Python or Java.

Declaring Variables

We declare constant variables in JavaScript with the `const` keyword:

```
const z = 1;  // constant (immutable variable) at global scope
```

You can try this in the debug console in a browser such as Chrome. If we try to change the value of such a const variable, we get a run-time error:

```
z = 2
> Uncaught TypeError: Assignment to constant variable.
```

We define mutable variables in JavaScript with the `let` keyword:

```
let w = 1;
```

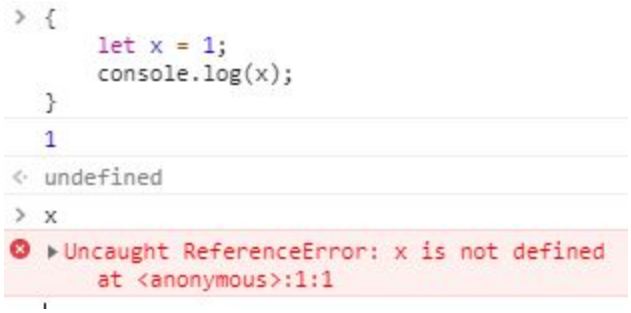
You can verify in the debugger that you are able to change the value of w.

```
console.log(w)
> 1
w = 2
console.log(w)
> 2
```

(Note: there is another legacy keyword for declaring variables in JavaScript "`var`" that has different scoping rules. Don't use it.)

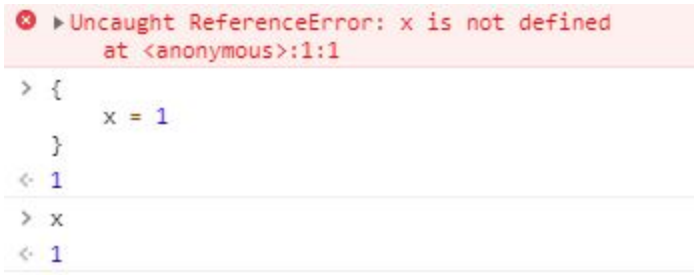
Variable scope

You can limit the visibility of a variable to a specific part of a program by declaring it inside a block of code delineated by curly braces:



The above console.log statement successfully output the value of x because it was inside the same scope (the same set of curly braces). The subsequent error occurs because we tried to look at x outside the scope of its definition. Variables declared outside of any scope are said to be "global" and will be visible to any code loaded on the same page and could clobber or be clobbered by other global definitions - so take care!

Be especially carefully to always declare variables with either `let` or `const` keywords. If you omit these keywords, a variable will be created at the global scope, like so:

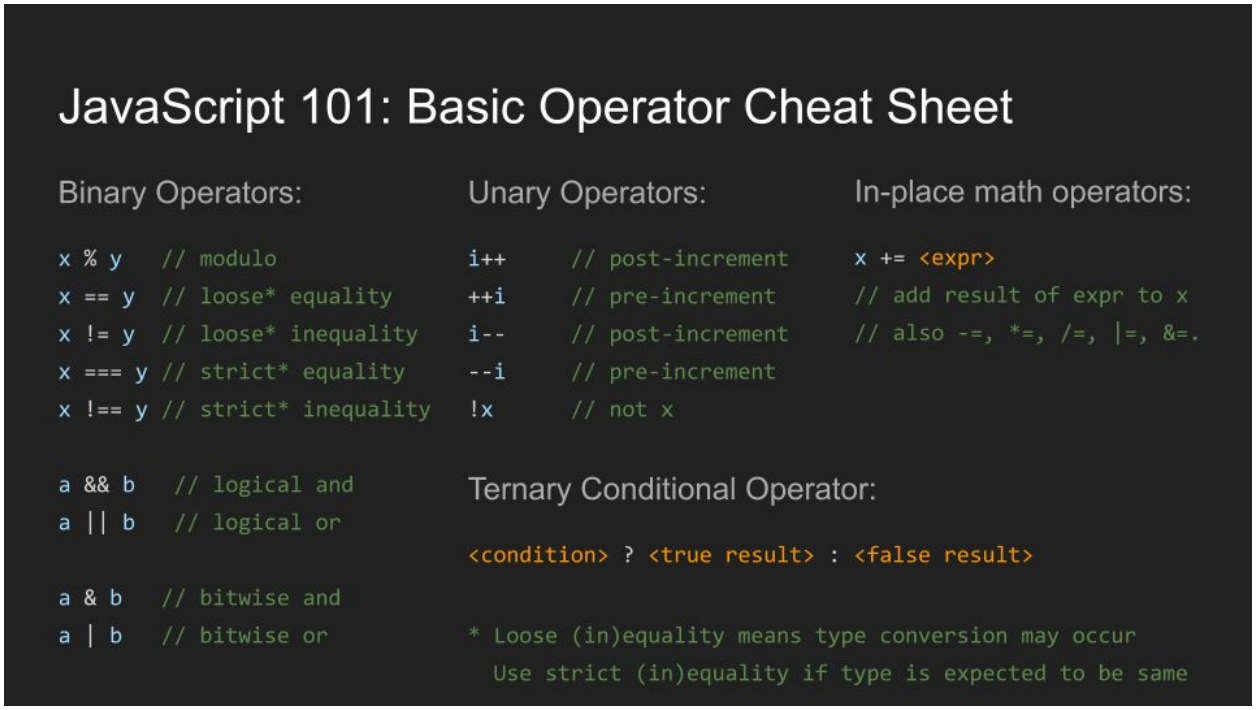


Functions

Functions are declared with the `function` keyword. You can give the function a name followed by a tuple of zero or more parameters. The scope of the function is marked by a matching pair of curly braces { … }. You return the result with the `return` keyword.

```
/**
* define a function called "myFunction" with two parameters, x and y
* which does some silly math, prints the value and returns the result
*/
function myFunction(x, y) {
  let t = x + y; // t is mutable
  t += z;  // += adds the result of the expression on the right to the value of t
  const result = t // semi colons are not essential (but can help to catch errors)
  console.log("hello world") // prints to the console
  return result; // returns the result to the caller
}
```

Above and below we are starting to use a few fancy operators like +=. Here's a cheat sheet:



You invoke (or call) a function like so:

```
myFunction(1,2)
> hello world
> 4
```

An if statement looks like so:

```
/**
* get the greater of x and y
*/
function maxVal(x, y) {
   if (x >= y) {
       return x;
   } else {
       return y;
   }
}
```

There is also a useful ternary expression syntax for if-then-else:

```
function maxVal(x, y) {
   return x >= y ? x : y;
}
```

We can loop with `while:`

```
/**
* sum the numbers up to and including n
*/
function sumTo(n) {
   let sum = 0;
   while (n) {
       sum += n--;
   }
   return sum;
}
sumTo(10)
> 55
```

Or `for`:

```
function sumTo(n) {
   let sum = 0;
   for (let i = 1; i <= n; i++) {
       sum += i;
   }
   return sum;
}
```

Or we could perform the same computation using a recursive loop:

```
function sumTo(n) {
   if (n === 0) return 0;  // base case
   return n + sumTo(n-1);  // inductive step
}
```

We can make this really succinct with a ternary if-then-else expression:

```
function sumTo(n) {
   return n ? n + sumTo(n-1) : 0;
}
```

We consider this recursive loop a more “declarative” coding style than the imperative loops.

It is closer to the ***inductive definition*** of sum than a series of steps for how to compute it.

- ***No mutable variables*** used
- Each expression in this code is "***pure***": it has no ***effects*** outside the expression.
- Therefore: this code has the property of ***referential transparency***.
- The code succinctly states the ***loop invariant.***

A rather serious caveat:
Too many levels of recursion will cause a ***stack overflow***.

```
sumTo(1000000)
> Uncaught RangeError: Maximum call stack size exceeded
```

We can make functions more versatile by parameterising them with other functions:

```
function sumTo(n, f) {
   return n ? f(n) + sumTo(n-1, f) : 0;
}

function square(x) {
   return x * x;
}
```

So we can compute the sum of the first 10 squares:

```
sumTo(10, square)
> 385
```

Objects
Like Java, everything in JavaScript is an object. You can construct an object populated with some data, essentially just with JSON syntax:

```
const myObj = {
    aProperty: 123,
    anotherProperty: "tim was here"
}
```

However, in JavaScript, objects are simply property bags, indexed by a hashtable:

```
// the following are equivalent and both involve a hashtable lookup:
console.log(myObj.aProperty)
console.log(myObj['aProperty'])
```

Note that when we declare an object with the `const` keyword as above, it is only ***weakly immutable***. This means that we cannot reassign myObj to refer to a different object, however, we can change the properties inside myObj. Thus, the `myObj` variable is constant/immutable,

but the object created by the declaration is mutable. So, after making the above **`const`** declaration, if we try the following reassignment of myObj we receive an error:

```
> myObj = { blah: 345 }
▶Uncaught TypeError: Assignment to constant variable.
    at <anonymous>:1:7
```

But we can reassign properties on **`myObj`**:

```
> myObj.anotherProperty = "tim was here again"
"tim was here again"
> myObj.aProperty = 456
456
> myObj
▶{aProperty: 456, anotherProperty: "tim was here again"}
```

We can also quickly declare variables that take the values of properties of an object, through **destructuring** syntax:

```
const {aProperty} = myObj
console.log(aProperty)
123
```

Which is equivalent to:

```
const aProperty = myObj.aProperty
```

This is most convenient to destructure objects passed as arguments to functions. It makes it clear from the function definition precisely which properties are going to be accessed. Consider:

```
const point = {x:123, y:456}
function showX({x}) {
   console.log(x)
}
```

Arrays

JavaScript has python-like syntax for array objects:

```
const a = [1,2,3]
a.length
3
a[0]
1
a[2]
3
```

You can also destructure arrays into local variables:

```
const [x,y,z] = a
z
3
```

See below for further functions for working with arrays.

Dynamic Typing

The members of myObj are implicitly typed as number and string respectively, and as we see in the console.log, conversion to string happens automatically. JavaScript is ***interpreted*** by a JavaScript engine rather than compiled into a static executable format. Originally, this had implications on execution speed, as interpreting the program line by line at run time could be slow. Modern JavaScript engines, however, feature Just in Time (JIT) compilation and optimisation - and speed is becoming comparable to execution of C++ code that is compiled in advance to native machine code. However, another implication remains. It is not type checked by a compiler. Thus, type errors cause run-time failures rather than being caught at compile time. JavaScript is dynamically typed in that types are associated with values rather than variables. That is, a variable that is initially bound to one type, can later be rebound to a different type, e.g.:

```
let i = 123;    // a numeric literal has type number
i = 'a string'; // a string literal has type string, but no error here!
```

The C compiler would spit the dummy when trying to reassign `i` with a value of a different type, but the JavaScript interpreter is quite happy to go along with your decision to change your mind about the type of `i`.

Functions

The nifty thing about JavaScript - one Scheme'ish thing that presumably survived from Eich's original plan - is that functions are also just objects. That is, given the following function:

```
function sayHello(person) {
    console.log('hello ' + person)
}
sayHello('tim')
> "hello tim"
```

We can easily bind a function to a variable[1]:

```
const hi = sayHello
hi('tim')
> "hello tim"
```

The sayHello function is called a ***named function***. We can also create an ***anonymous*** *function* to be bound immediately to a variable:

```
const hi = function(person) {
    console.log("hello " + person)
}
```

or to pass as a parameter into another function, for example, Array objects have a `forEach` member that expects a function as an argument, which is then applied to every member of the array:

```
['tim', 'sally', 'anne'].forEach(function(person) {
      console.log('hello ' + person)
})
> "hello tim"
> "hello sally"
> "hello anne"
```

This pattern is now so common in JavaScript that the EcmaScript 6 standard introduced some new ***arrow*** syntax (with slightly different semantics, as explained below) for anonymous functions:

```
['tim', 'sally', 'anne'].forEach(person=> console.log('hello ' + person))
```

Note that whatever value the expression on the right-hand side of the arrow evaluates to is implicitly returned:

```
['tim', 'sally', 'anne'].map(person=> "hello " + person)
> ["hello tim", "hello sally", "hello anne"]
```

---

[1] The original JavaScript syntax for declaring a variable used the `var` keyword. However, the scoping of variables declared in this way was strange for people familiar with C and Java scoping rules, and caused much angst. It has been fixed since ES6 with the `let` and `const` keywords, we prefer these to `var`.

Multiple ';' separated statements including local variable declarations can be enclosed in brackets with arrow syntax, but then an explicit return statement is required to return a value:

```
['tim', 'sally', 'anne'].map(person=> {
    const message = "hello " + person
    console.log(message)
    return message
})
```

Array Cheat Sheet: Pure Methods on Array

Where a is an array with elements of type U *(Note: these are not correct TS annotations, but a Haskelly "shorthand")*:

```
a.forEach(f: U=> void): void            // apply the function f to each element of the array

// Pure functions:
a.slice(): U[]                          // copy the whole array
a.slice(start: number): U[]             // copy from the specified index to the end of the array
a.slice(start: number, end: number): U[]// copy from start index up to (but not including) end index
a.map(f: U=> V): V[]                    // apply f to elements and return result in new array of type V
a.filter(f: U=> boolean): U[]           // returns a new array of the elements for which f returns true
a.concat(b: U[]): U[]                   // return a new array with the elements of b concatenated after
                                        // the elements of a (can take additional array arguments)
a.reduce(f: (V, U)=> V, V): V           // Uses f to combine elements of
                                        // the array into a single result of type V
```

All of the above are pure in the sense that they do not mutate a, but return the result in a new object.

What about Array.forEach?

Closures

Functions can be nested inside other function definitions and can access variables from the enclosing scope. A function and the set of variables it accesses from its enclosing scope is called a ***closure***. You can also have a function that creates and returns a closure that can be applied later.

```
function add(x) {
    return y => y+x; // we are going to return a function which includes
                     // the variable x from it's enclosing scope
                     // - "a closure"
}
let addNine = add(9)
addNine(10)
> 19
addNine(1)
```

```
> 10
```

Functions that take other functions as parameters (like `[].forEach`) or which return functions (like the above definition of `add`) are called ***higher-order functions***.

In JavaScript you can also create functions as members of objects:

```
let say = {
    hello: person => console.log('hello ' + person)
}
say.hello("tim")
> "hello tim"
```

But these objects are only single instances. JavaScript supports creating object instances of a certain type (i.e. having a set of archetypical members, like a Java class) through a function ***prototype*** mechanism. You create a constructor function:

```
function Person(name, surname) {
    this.name = name
    this.surname = surname
}
let author = new Person('tim', 'dwyer')
sayHello(author.name)
> "hello tim"
```

You can also add method functions to the prototype, that are then available from any objects of that type:

```
Person.prototype.hello = function() { console.log("hello " + this.name) }
author.hello()
> "hello tim"
```

Note that above we use the old-style verbose JavaScript anonymous function syntax instead of the arrow form. This is because there is a difference in the way the two different forms treat the `this` symbol. In the arrow syntax, `this` refers to the enclosing execution context. In the verbose syntax, `this` resolves to the object the method was called on.

It's very tempting to use the prototype editing mechanism for evil. For example, I've always wished that JS had a function to create arrays initialised over a range:

```
Array.prototype.range = (from, to)=>Array(to).fill()
      .map((_,i)=>i)
      .filter(v=> v >= from)

[].range(3,9)
> [3,4,5,6,7,8]
```

Of course, if you do something like this in your JS library, and it pollutes the global namespace, and one day EcmaScript 9 introduces an actual range function with slightly different semantics, and someone else goes to use the `[].range` function expecting the official semantics - well, you may lose a friend or two.

Exercises:

4.1. Amend the range function above to handle negative values in from or to

4.2. Hack a sum function onto the Array.prototype (you'll need to use an old style anonymous function to access the array through this).

4.3. Why might you lose friends doing this kind of thing to built-in types?

4.4. We are going to be dealing with linked-list like data structures a lot in this course. Implement a linked list using javascript objects as simply as you can, and create some functions for working with it, like length and map.

EcmaScript 6 Class Syntax

Consider another class created with a function and a method added to the prototype:

```
function Person(name, occupation) {
   this.name = name
   this.occupation = occupation
}
Person.prototype.sayHello = function() {
   console.log(`Hi, my name's ${this.name} and I ${this.occupation}!`)
}
const tim = new Person("Tim","lecture Programming Paradigms")
tim.sayHello()
> Hi, my name's Tim and I lecture Programming Paradigms!
```

ES6 introduced a new syntax for classes that will be more familiar to Java programmers:

```
class Person {
   constructor(name, occupation) {
       this.name = name
       this.occupation = occupation
```

```
    }
    sayHello() {
        console.log(`Hi, my name's ${this.name} and I ${this.occupation}!`)
    }
}
```

There is also now syntax for “getter properties”: functions which can be invoked without (), i.e. to look more like properties:

```
class Person {
    constructor(name, occupation) {
        this.name = name
        this.occupation = occupation
    }
    get greeting() {
        return `Hi, my name's ${this.name} and I ${this.occupation}!`
    }
    sayHello() {
        console.log(this.greeting)
    }
}
```

And classes of course support single-inheritance to achieve polymorphism:

```
class LoudPerson extends Person {
    sayHello() {
        console.log(this.greeting.toUpperCase())
    }
}

const tims = [
    new Person("Tim","lecture Programming Paradigms"),
    new LoudPerson("Tim","shout about Programming Paradigms")
]

tims.forEach(t => t.sayHello())
Hi, my name's Tim and I lecture Programming Paradigms!
HI, MY NAME'S TIM AND I SHOUT ABOUT PROGRAMMING PARADIGMS!
```

Dependency Injection

It's useful to compare the above style of polymorphism, to a functional approach to dependency injection:

```
class Person {
   constructor(name, occupation, voiceTransform = g => g) {
       this.name = name
       this.occupation = occupation
       this.voiceTransform = voiceTransform
   }
   get greeting() {
       return `Hi, my name's ${this.name} and I ${this.occupation}!`
   }
   sayHello() {
       console.log(this.voiceTransform(this.greeting))
   }
}
const tims = [
   new Person("Tim", "lecture Programming Paradigms"),
   new Person("Tim", "shout about Programming Paradigms", g => g.toUpperCase())
]
tims.forEach(t => t.sayHello())
Hi, my name's Tim and I lecture Programming Paradigms!
HI, MY NAME'S TIM AND I SHOUT ABOUT PROGRAMMING PARADIGMS!
```

So the filter property defaults to the identity function (a function which simply returns its argument), but a user of the Person class can inject a dependency on another function from outside the class when they construct an instance.

This is a "lighter-weight" style of code reuse or specialisation.

5. Functional Programming

Learning Outcomes

- Create programs in JavaScript in a functional style
- Explain the role of pure functional programming style in managing side effects

Introduction

The elements of JavaScript introduced above, specifically:

- Anonymous functions
- Binding functions to variables
- Higher-order functions

are sufficient for us to explore a paradigm called ***functional programming***. In the functional programming paradigm the primary model of computation is through the evaluation of functions. While JavaScript (and many---but not all, as we shall see---other languages inspired by the functional paradigm) do not enforce it, true functional programming mandates the functions be ***pure*** in the sense of not causing ***side effects***. Side effects are changes to state outside of the result returned by the function directly.

Function Purity vs Side Effects

Pure functions are perhaps most easily illustrated with some examples and counterexamples. The following function has no effect outside its return result and is therefore ***pure***:

```
function squares(a) {
    let b = new Array(a.length);
    for (let i = 0; i < a.length; i++) {
        b[i] = a[i]**2;
    }
    return b;
}
```

While the above function (viewed as a black box) is pure, its implementation is not very functional. Specifically, the code around variable `b` does not have the property of ***referential transparency***. That is, because the value of `b` is reassigned during execution figuring out the state at a given line is impossible, without looking at its context. True functional languages enforce referential transparency through ***immutable*** variables (which sounds like a tautology). That is, once a variable is ***bound*** to a value, it cannot be reassigned. In JavaScript we can opt-in to immutable variables by declaring them `const`.

A more functional way to implement the `squares` function would be more like the examples we have seen previously:

```
const squares = a=> a.map(x=> x**2)
```

By contrast, the following is considered impure because it modifies the memory referred to by `a` in-place:

```
function squares(a) {
    for (let i = 0; i < a.length; i++) {
```

```
        a[i] = a[i]**2;
    }
}
```

An impure function typical of something you may see in OO code:

```
let messagesSent = 0;
function send(message, recipient) {
   let success = recipient.notify(message);
   if (success) {
       ++messagesSent;
   } else {
       console.log("send failed! " + message);
   }
   console.log("messages sent " + messagesSent);
}
```

This function is impure in three ways:

- it **mutates** the state of the count variable `messagesSent` from the enclosing scope
- it (likely) does something (what exactly is unclear) to `recipient`
- finally, it sends output to the console. Outputting to a device (although only for display purposes) is definitely a side effect.

Side effects are bad for transparency (knowing everything about what a function is going to do) and maintainability. When state in your program is being changed from all over the place bugs become very difficult to track down.

Computation with Pure Functions

Pure functions may seem restrictive, but in fact pure function expressions and higher-order functions can be combined into powerful programs. In fact, anything you can compute with an imperative program can be computed through function composition. Side effects are required eventually, but they can be managed and the places they occur can be isolated. Let's do a little demonstration, although it might be a bit impractical, we'll make a little list processing environment with just functions:

```
const cons = (head, rest)=> selector=> selector(head, rest);
```

With just the above definition we can `cons`truct a list (the term ***cons*** dates back to LISP) with three elements, terminated with null, like so:

```
const list123 = cons(1, cons(2, cons(3, null)));
```

The data element, and the reference to the next node in the list are stored in the closure returned by the cons function. Created like this, the only side-effect of growing the list is creation of new cons closures. Mutation of more complex structures such as trees can be managed in a similarly 'pure' way, and surprisingly efficiently, as we will see later in this course.

So `cons` is a function that takes two parameters (`head` and `rest`), and returns a function that itself takes a function (`selector`) as argument. The `selector` function is then applied to `head` and `rest`. What might the `selector` function be and how do we apply it to a list element? Well we don't exactly apply it ourselves, we give it to the closure returned by the cons function and it applies it for us. There are the two selectors we need to work with the list:

```
const
    head = list=> list((head, rest)=> head),
    rest = list=> list((head, rest)=> rest);
```

Now, `head` gives us the first data element from the list, and `rest` gives us another list. Now we can access things in the list like so:

```
const one = head(list123), // ===1
    list23 = rest(list123),
    two = head(list23), // ===2
    … // and so on
```

Now, here's the ubiquitous `map` function:

```
const map = (f, list)=> !list ? null
                : cons(f(head(list)), map(f, tail(list)))
```

Exercise:

5.1. Implement a fromArray function to construct a list from an array

5.2. Implement a filter function, which takes a function and a list, and returns another list populated only with those elements of the input list for which the function returns true

5.3. Implement a reduce function for these functional lists, similar to javascript's Array.reduce

5.4. How can we update just one element in this list without mutating any data and what is the run-time complexity of such an operation?

Thus, with only pure function expressions and JavaScript conditional expressions (`?:`) we can begin to perform complex computations. We can actually go further and eliminate the conditional expressions with more functions! Here's the gist of it: we wrap list nodes with another function of two arguments, one argument, `whenempty`, is a function to apply when the list is empty, the other argument, `notempty`, is applied by all internal nodes in the list. An empty list node (instead of null) applies the `whenempty` function when visited, a non-empty node applies the `notempty` function. The implementations of each of these functions then form the two conditions to be handled by a recursive algorithm like `map` or `reduce`. See "Making Data out of Functions" by Braithwaite for a more detailed exposition of this idea.

These ideas, of computation through pure function expressions, are inspired by Alonzo Church's ***lambda calculus***. We'll be looking again at the lambda calculus later. Obviously, for the program to be at all useful you will need some sort of side effect, such as outputting the results of a computation to a display device. When we begin to explore PureScript and Haskell later in this course we will discuss how such languages manage this trick while remaining "pure".

6. TypeScript: because type checking makes it easier to work with Higher-Order Functions

Learning Outcomes

- Create programs in TypeScript using the types to ensure correctness at compile time.
- Explain how Generics allow us to create type safe but general and reusable code.
- Compare and contrast strongly, dynamically and gradually typed languages.
- Describe how compilers that support type inference assist us in writing type safe code.

As the Web 2.0 revolution hit in 2000s web apps built on JavaScript grew increasingly complex and today, applications like GoogleDocs are as intricate as anything built over the decades in C++. In the 90s I for one (though I don't think I was alone) thought that this would be impossible in a dynamically typed language. It is just too easy to make simple mistakes (as simple as typos) that won't be caught until run time. It's likely only been made possible due to increased rigour in testing. That is, instead of relying on a compiler to catch mistakes, you rely on a comprehensive suite of tests that evaluate running code before it goes live.

Part of the appeal of JavaScript is that the source code being the artefact that runs directly in a production environment gives an immediacy and comprehensibility to software deployment. However, in recent years more and more tools have been developed that introduce a build-chain into the web development stack. Examples include: minifiers, which compact and obfuscate JavaScript code before it is deployed; bundlers, which merge different JavaScript files and libraries into a single file to (again) simplify deployment; and also, new languages that compile to JavaScript, which seek to fix older versions of the JavaScript language's shortcomings and compatibility issues in different browsers. Examples of the latter include CoffeeScript, Clojure and (more recently) PureScript (which we will visit later in this unit). Right now, however, we will take a closer look at another language in this family called *TypeScript*. See here for some tutorials and deeper documentation.

TypeScript is interesting because it forms a relatively minimal augmentation, or superset, of EcmaScript syntax that simply adds type annotations. For the most part, the compilation process simply performs validation on the declared types and strips away the type annotations rendering just the legal JavaScript ready for deployment. This lightweight compilation into a language with a similar level of abstraction to the source is known also known as ***transpiling*** (as opposed to C++ or Java where the object code is much closer to the machine execution model).

An excellent free resource for learning the TypeScript language is the TypeScript Deep-dive book. Type annotations in TypeScript come after the variable name's declaration, like so:

```
let i: number = 123;
```

Actually, in this case the type annotation is completely redundant. The TypeScript compiler features sophisticated ***type inference***. In this case it can trivially infer the type from the type of the literal.

Previously, we showed how rebinding such a variable to a string in JavaScript is perfectly fine by the JavaScript interpreter. However, such a change of type in a variable a dangerous pattern that is likely an error on the programmer's part. The TypeScript compiler will generate an error:

```
let i = 123;
i = 'hello!';
//[ts] Type '"hello!"' is not assignable to type 'number'.
```

However, it is a fairly common practice in JavaScript to implicitly create overloaded functions by accepting arguments of different types and resolving them at run-time. For example, the following function can handle either a string in `value`, or a function that needs to be called to retrieve the string value:

```
function setAttr(elem, name, value) {
      const val = typeof value === 'function' ? value() : value;
      elem.setAttribute(name, val);
}
```

TypeScript can make this expectation explicit:

```
function setAttr(elem: Element, name: string,
      value: string | (()=>string))
{
      const val = typeof value === 'function' ? value() : value;
      elem.setAttribute(name, val);
}
```

Where the type annotation: `string | (()=>string)` is called a ***union type***, which provides a '|' separated list of possible types for the variable. The TypeScript typechecker also knows about typeof expressions (as used above) and will also typecheck the different clauses of if statements that use them for consistency with the expected types.

TypeScript also supports ***generics***, or parameterised types. For example, the following interface might be the basis of a linked list element:

```
interface IListNode<T> {
      data: T;
      next?: IListNode<T>;
}
```

The ? after the next means that this property is optional. Thus:

```
typeof this.next === 'undefined'
```

or simply:

```
!this.next
```

would indicate the end of the list.

A concrete implementation of a class for `IListNode<T>` can provide a constructor:

```
class ListNode<T> implements IListNode<T> {
   constructor(public data: T, public next?: IListNode<T>) {}
}
```

Then we can construct a list like so:

```
const list = new ListNode(1, new ListNode(2, new ListNode(3)));
```

Exercises:

6.1 Implement a class List<T> whose constructor takes an array parameter and creates a linked list of ListNode<T>.

6.2 Add methods to your List<T> class for:

forEach(f:(_:T)=>void):List<T> (returns the same list being operated on to support method chaining)

filter(f:(_:T)=>boolean):List<T>

map<V>(f:(_:T)=>V):List<V>

reduce(f:(accumulator:V,t:T)=>V, initialValue:V):V

Typing systems with different 'degrees' of strictness

C++ is considered a ***strongly*** typed language in the sense that all types of values and variables must match up on assignment or comparison. Further, it is "statically" typed in that the compiler requires complete knowledge (at compile-time) of the type of every variable. This can be overridden (type can be cast away and void pointers passed around) but the programmer has to go out of their way to do it (i.e. opt-out).

JavaScript, by contrast, as we have already mentioned, is ***dynamically*** typed in that types are only checked at run time. Run-time type errors can occur and be caught by the interpreter on primitive types, for example the user tried to invoke an ordinary object like a function, or refer to an attribute that doesn't exist, or to treat an array like a number.

TypeScript represents a relatively new trend in being a ***gradually*** *typed* language. Another way to think about this is that, by default, the type system is ***opt-in***. Unless declared otherwise, all variables have type `any`. The `any` type is like a wild card that always matches, whether the `any` type is the target of the assignment:

```
let value; // has implicit <any> type
value = "hello";
value = 123;
// no error.
```

Or the source (r-value):

```
let value: number;
value = "hello";
//[ts] Type '"hello"' is not assignable to type 'number'.
value = <any>"hello";
// no error.
```

While leaving off type annotations and forcing types with `any` may be convenient, for example, to quickly port legacy JavaScript into a TypeScript program, generally speaking it is good practice to use types wherever possible, and can actually be enforced with the `--noImplicitAny` compiler flag. The compiler's type checker is a sophisticated constraint satisfaction system and the correctness checks it applies are usually worth the extra effort - especially in modern compilers like TypeScript where type inference does most of the work for you.

7. Functional Patterns

Passing functions around, anonymous or not, is incredibly useful and pops up in many practical programming situations.

Eliminating Loops
Loops are the source of many bugs: fence-post errors, range errors, typos, incrementing the wrong counter, etc.

A typical for loop has four distinct places where it's easy to make errors that can cause critical problems:

for ([*initialization*]; [*condition*]; [*final-expression*])
  statement

- The ***initialization*** can initialise to the wrong value (e.g. n instead of n-1, 1 instead of 0) or initialise the wrong variable.
- The ***condition*** test can use = instead of ==, <= instead of < or test the wrong variable, etc.
- The ***final-expression*** can (again) increment the wrong variable
- The ***statement*** body might change the state of variables being tested in the termination condition since they are in scope.

For many standard loops, however, the logic is the same every time and can easily be abstracted into a function. Examples: `Array.map`, `Array.reduce`, `Array.forEach`, etc. The logic of the loop body is specified with a function which can execute in its own scope, without the risk of breaking the loop logic.

Callbacks
In JavaScript and HTML5 events trigger actions associated with all mouse clicks and other interactions with the page. You subscribe to an event on a given HTML element as follows:

```
element.addEventHandler('click',
      e=>{
      // do something when the event occurs,
      // maybe using the result of the event e
})
```

Note that callback functions passed as event handlers are a situation where the difference between the arrow syntax and regular anonymous function syntax really matters. In the body of the arrow function above this will be bound to the context of the caller, which is probably what you want if you are coding a class for a reusable component.

Continuations
Continuations are functions which, instead of returning the result of a computation directly to the caller, pass the result on to another function, specified by the caller.

We can rewrite basically any function to pass their result to a continuation function instead of returning the result directly.

```
function simplePlus(a: number, b: number): number {
   return a + b;
}
function continuationPlus(a:number, b:number, done:(result:number)=>void)
{
   done(a+b);
}
```

We can also rewrite tail-recursive functions to end with continuations, which specify some custom action to perform when the recursion is complete:

```
function tailRecFactorial(a: number, n: number): number {
   return n<=1 ? a : tailRecFactorial(n*a, n-1);
}
function continuationFactorial(
      a: number, n: number, finalAction: (result:number)=>void): void
{
   if (n<=1) finalAction(a);
   else continuationFactorial(n*a, n-1, finalAction);
}
```

Continuations are essential in asynchronous processing, because the function will return immediately after dispatching the job, e.g. to the JavaScript event loop:

```
setTimeout(()=>console.log('done.'), 0);
// the above tells the event loop to execute
// the continuation after 0 milliseconds delay.
// even with a zero-length delay, the synchronous code
// after the setTimeout will be run first...
console.log('job queued on the event loop...');
> job queued on the event loop...
> done.
```

NOTE: SECTIONS HIGHLIGHTED IN BLUE ARE NOT CORE OR EXAMINABLE IN FIT2102 IN 2018. READ THEM ONLY IF YOU ARE INTERESTED.

RAII with functions

Resource Acquisition Is Initialisation or (RAII), as discussed earlier in relation to C++ memory management, is not just for managing memory. There are other resources which need to be released after use - and that must be applied in all circumstances (for example, in the event of an exception). Examples include mutexes and file handles. In C++ we showed how this was achieved using constructors to Acquire the resource, and the C++ stack cleanup processes' reliable invocation of destructors to release the resource. In garbage collected languages we don't know precisely when an object will be disposed so the pattern is not usable in quite the same way. However, by wrapping use of the resource in a function and passing in functions for the acquisition, use and disposal of the resource, we can achieve something similar:

```
function using<T>(
    acquire: ()=>T,
    action: (_:T)=>void,
    error: (e:any)=>void,
    dispose: (_:T)=>void
) {
    let resource:T = null;
    try {
        resource = acquire();
        action(resource);
    }
    catch(e) {
        error(e);
    }
    finally {
        if(resource) dispose(resource);
    }
}
```

You see this pattern a lot in JavaScript. The Node.js filestream API, for example, works precisely this way.

Method Chaining

Chained functions are a common pattern. Assuming definitions for `map`, `filter` and `take` similar to:

```
// maps an Iterable (an interface like our IListNode with a way to get a value of
// the current node or iterate to the next element) containing elements of
// type T, to an Iterable with elements of type V
function map<T,V>(f: (_:T)=>V, l: Iterable<T>): Iterable<V> { ...
// create a new Iterable containing only the elements of l that
// satisfy the predicate f
function filter<T>(f: (t:T)=>boolean, l: Iterable<T>): Iterable<T> { ...
// create a new Iterable containing only the fist n elements of l
function take<T>(n: number, l: Iterable<T>): Iterable<T> { ...
```

We can chain calls to these functions like so:

```
take(3,
   filter(x=>x%2===0,
       map(x=>x+1, someIterable)
   )
)
```

But in the function call chain above you have to read them inside-out to understand the flow. Also, keeping track of how many brackets to close gets a bit annoying. Thus, you will often see class definitions that allow for method chaining, by providing methods that return an instance of the class itself, or another chainable class.

```
class List<T> {
   map<V>(f: (item: T) => V): List<V> {
       return new List(map(f, this.head));
   }
...
```

Then the same flow as above is possible without the nesting and can be read left-to-right, top-to-bottom:

```
someIterable
   .map(x=>x+1)
   .filter(x=>x%2===0)
```

```
    .take(3)
```

This is called “fluent” programming style.

NOTE: SECTIONS HIGHLIGHTED IN BLUE ARE NOT CORE OR EXAMINABLE IN FIT2102 IN 2018. READ THEM ONLY IF YOU ARE INTERESTED.

ES6 Iterable

This is just an FYI in case you see this style of code in the wild. You don't need (and I don't even encourage you) to use ES6 generators for your code in this unit.

EcmaScript 6 introduces a standard Iterable interface and so-called generator function syntax (`function*` and `yield`) that allow you to use imperative style for loops over data structures that implement the interface.

```
function* map<U,V>(f:(_:U)=>V, it: Iterable<U>) {
   for (const u of it) yield f(u);
}
class List<T> implements Iterable<T> {
   ...
   map<V>(f:(_:T)=>V) : List<V> {
        return new List<V>(map(f, this));
   }
   ...
}
```

Tap

The tap function can be used to “tap” into the flow of a chain of functions and do something with the value being passed into the next function.

Its type signature looks like this:

```
function tap<T,V>(f:(_:T)=>void, g:(_:T)=>V):(_:T)=>V
```

So tap creates a function in which function f is applied to x, before it returns the result of g(x).

Then, in a mysterious function chain, like the following:

```
somethingIterable
    .filter(someComplicatedFilter)
    .map(someComplicatedTransform)
```

We can harmlessly “tap in” to inspect the values coming out of the filter:

```
somethingIterable
    .filter(someComplicatedFilter)
    .map(tap(console.log, someComplicatedTransform))
```

Exercise:
7.1. Implement tap

7.2. Create a function called trace, that you can wrap around any unary function in order to output the name of the function and the value being passed in before actually invoking the function. Hint: in JavaScript, you can get the name of a function f simply as f.name.

Fluent Interfaces (pure vs impure)
Interfaces like the above in object-oriented languages are often called *fluent interfaces*. One thing to be careful about fluent interfaces in JavaScript is that the methods may or may not be *pure*. That is, the type system does not warn you whether the method mutates the object upon which it is invoked and simply returns `this`, or creates a new object, leaving the original object untouched. We can see, however, that `List.map` as defined above, creates a new list and is pure.

Exercise:
7.3. If someIterable above were declared const, would it protect you against mutations in someIterable due to impure methods?

Lazy evaluation
Usually, expressions in imperative languages are fully evaluated each step immediately after the previous step. This is called *strict* or *eager* evaluation. Functions, however, give us a way to execute code (or evaluate expressions) only when they are really required. This is called *lazy evaluation*. As an eager student at Monash University you will be unfamiliar with this concept, since you always do all your work as soon as possible because you love it so much. This is obviously the best strategy for most things in life (and definitely for studying for this course), but laziness as a programming paradigm can sometimes enable different ways to model problems. Early in our introduction to TypeScript we looked at a function `setAttr` that took as argument, either an immediate value or a function that returns a value:

```
function setAttr(elem: Element, name: string,
      value: string | (()=>string)) {//...
```

We didn't discuss this much at the time, but this potentially lets us delay the evaluation of an expression. We don't have to have the value ready at the time of the function call, rather we can provide a computation to obtain the value at a later time.

This is the essence of how functional languages can elevate laziness to a whole paradigm, and it is a powerful concept. For one thing, it allows us to define *infinite lists*[2]. Compare the following definition with the `ListNode` class we defined earlier:

```
interface LazySequence<T> {
   value: T;
   next():LazySequence<T>;
}
```

They both have two properties, the only real difference between the two is that `next` is defined here as a function (method) instead of a simple property.

```
function naturalNumbers() {
   return function _next(v:number):LazySequence<number> {
       return {
           value: v,
           next: ()=>_next(v+1)
       }
   }(1)
}
```

Note that _next is immediately invoked in the return. If you are like me you will need to look at this for a little to get your head around it. To summarise: we are defining a function in a returned expression and immediately invoking it with 1 as the starting value. This pattern is used so often in JavaScript it has an acronym: ***IIFE*** or ***Immediately Invoked Function Expression***. It was one way of achieving encapsulation before ES6 introduced proper classes.

```
const n = naturalNumbers();
n.value
> 1
n.next().value
> 2
n.next().next().value
> 3
```

And so on, as many times as you call next.

Exercise:

[2] A handy blogpost by Jeremy Fairbanks explains this idiom further.

7.4. Create a function take(n,seq) which returns a `LazySequence` of *the first n elements of an infinite* `LazySequence` *of the sort generated by naturalNumbers.* Leave the `next` property of the last element of the result sequence `undefined`, to indicate the end.

7.5. Create `map`, `filter` *and reduce functions (similar to those defined on Array.prototype) for such a sequence and use them along with* `take` *to create a solution for Project Euler Problem 1 (encountered earlier): sum of first n natural numbers not divisible by 3 or 5.*

7.6. Make a general purpose infinite sequence initialisation function that creates infinite lazy sequences. It will take as parameter a function to compute the next value from the current value. In other words, it should be a "factory" for functions like naturalNumbers. Thus, if we call our function initSequence, then initSequence(n=>n+1) will return a function equivalent to naturalNumbers.

7.7. Use your general purpose sequence generator to generate fibonacci numbers.

Functional Reactive Programming

Modern computer systems often have to deal with asynchronous processing. Examples abound:

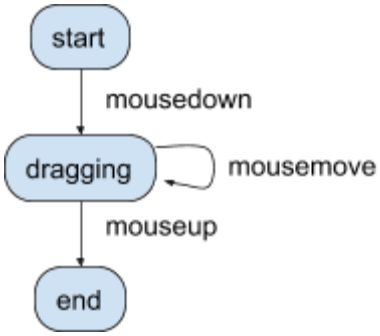
- In RESTful web services, where a client sends a non-blocking request (e.g. GET) with no guarantee of when the server will send a response.
- In user interfaces, events are triggered by user interaction with different parts of the interface, which may happen at any time.
- Robotics and other systems with sensors, the system must respond to events in the world.

Under the hood, most of these systems work on an *event model*, a kind of single-threaded multitasking where the program (after initialisation) polls a FIFO queue for incoming events in the so-called *event loop*. When an event is popped from the queue, any *subscribed actions* for the event will be applied.

In JavaScript the first event loop you are likely to encounter is the browser's. Every object in the DOM (Document Object Model - the tree data structure behind every webpage) has events that can be subscribed to, by passing in a callback function which implements the desired action. We saw a basic click handler earlier.

Handling a single event in such a way is pretty straightforward. Difficulties arise when events have to be nested to handle a (potentially-bifurcating) sequence of possible events.

A simple example that begins to show the problem is implementing a UI to allow a user to drag an object on (e.g.) and SVG canvas. The state machine that models this is pretty simple:



There are only three transitions, each triggered by an event. Here's an event-driven code fragment that provides such dragging for some SVG element `elem`, that is a child of an SVG canvas element referred to by the variable `svg`:

```
elem.addEventListener('mousedown', (e:MouseEvent)=>{
   const
     xOffset = Number(elem.getAttribute('x')) - e.clientX,
     yOffset = Number(elem.getAttribute('y')) - e.clientY,
     moveListener = (e:MouseEvent)=>{
       elem
         .setAttribute('x',e.clientX + xOffset)
         .setAttribute('y',e.clientY + yOffset);
     },
     done = ()=>{
       svg.removeEventListener('mousemove', moveListener);
     };
   svg.addEventListener('mousemove', moveListener);
   svg.addEventListener('mouseup', done);
   svg.addEventListener('mouseout', done);
 })
```

It's all a bit amorphous - the flow of control is not very linear or clear. We're declaring callback functions inside of callback functions, we have to manually unsubscribe from events when we're done with them (or potentially deal with weird behaviour when unwanted zombie events fire). The latter is not unlike the kind of resource cleanup that RAII is meant to deal with. Generally speaking, nothing about this function resembles the state machine diagram. The code sequencing has little sensible flow.

Functional Reactive Programming or FRP is another functional programming pattern (enabled by a data-structure called **`Observable`** that has much in common with the functional linked lists and trees we have looked at earlier.

```
const
  mousedown = Observable.fromEvent<MouseEvent>(elem, 'mousedown'),
  mousemove = Observable.fromEvent<MouseEvent>(svg, 'mousemove'),
  mouseup = Observable.fromEvent<MouseEvent>(svg, 'mouseup');

mousedown
  .map(({clientX, clientY}) => ({
    mouseDownXOffset: Number(elem.getAttribute('x')) - clientX,
    mouseDownYOffset: Number(elem.getAttribute('y')) - clientY
  }))
  .flatMap(({mouseDownXOffset, mouseDownYOffset}) =>
    mousemove
      .takeUntil(mouseup)
      .map(({clientX, clientY}) => ({
        x: clientX + mouseDownXOffset,
        y: clientY + mouseDownYOffset
      })))
  .subscribe(({x, y}) => {
    elem.setAttribute('x', x)
    elem.setAttribute('y', y)
  });
```

The Observable's `mousedown`, `mousemove` and `mouseup` are like streams which we can transform with familiar operators like `map` and `takeUntil`. The `flatMap` operator "flattens" the inner `mousemove` Observable stream back to the top level, then `subscribe` will apply a final action before doing whatever cleanup is necessary for the stream.

Inside Observable

The `Observable` interface hides a bit of complexity in order to present to the programmer using it something that looks like a functional list or lazy sequence like the ones we have explored earlier. Thus, to use observable you can ignore this complexity and simply chain map, filter, takeUntil and so on into sophisticated streams, each terminated by a subscribe as above.

It is interesting, however, to dig inside `Observable` a little further to see how it hides the complexity of asynchronous behaviour. The code snippets below are from a simple `Observable` implementation, but in practice Rx.js uses similar behaviour, and such `Observables` are slated eventually for inclusion in the EcmaScript standard.

Behind the scenes each `Observable` is backed by an `Observer`. `Observers` are like linked-list nodes, with a reference *destination* to another `Observer`. `Observer` has a *next* method that is given a value v by whatever invokes it, and then passes it to the destination via destination.next(v). The `Observer` interface looks like (note: a more complete `Observer` would also feature an error handler):

```
interface Observer<Input> {
  next(value: Input): void;
  complete(): void;
  unsub?: ()=>void;
}
```

So `Observers` are either notified of incoming stream values by `next` or the stream is closed by a call to `complete`. The `unsub` method is a stream-specific clean up operation (this is again, where we see the functional RAII pattern). For example, in the `Observable` stream created by `fromEvent` (below) `unsub` removes the event listener.

To explore the relationship between `Observable` and `Observer` further we trace the setup of the mousedown stream above to see what the `Observable` does internally. The mousedown `Observable` stream is constructed with the `Observable.fromEvent` - a static factory function that constructs a new `Observable`. When it's first created in this way the `Observable` is incomplete, it has no `Observer`. The `Observer` is not created until something subscribes to the `Observable` (calls `Observable`.subscribe). In the `fromEvent`, the `Observable` that is created has no `Observer`, but the constructor is provided with a function that knows how to wire up an `Observer` to an eventHandler, by calling the `Observer`'s next method when the event fires.

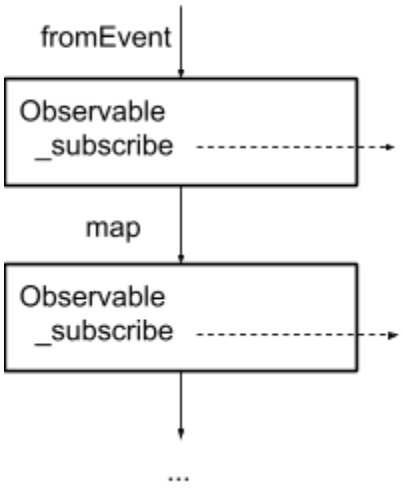
```
static fromEvent<E extends Event>(el: Node, name: string): Observable<E> {
  return new Observable<E>((observer: Observer<E>) => {
      const listener = (e:E) => observer.next(e);
      el.addEventListener(name, listener);
      return () => el.removeEventListener(name, listener);
  })
}
```

We then attach a new `Observable` to the mousedown `Observable` via its map method, which creates a new `Observable`. Still, this does not immediately create an `Observer`, rather a function is passed to the new `Observable`'s constructor that knows how to subscribe the first `Observable` to an observer (not yet specified). This delayed (lazy) subscription involves

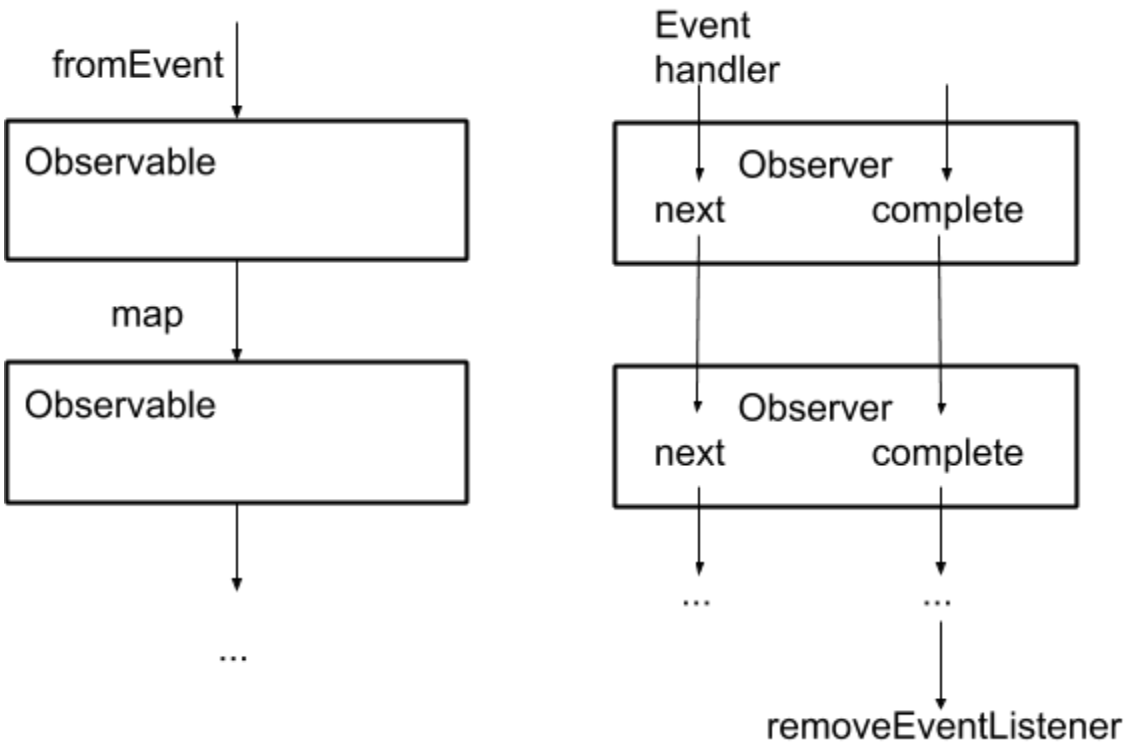
specifying a `next` function (that will eventually become the `Observer`'s `next`). The `next` function will invoke the listening `Observer`'s `next` on a projected value.

```
// create a new observable that observes this observable
// and applies the project function on next
map<R>(project: (_:Input)=>R): Observable<R> {
  return new Observable<R>(observer=>
    this.subscribe(e=>observer.next(project(e)), ()=>observer.complete())
  );
}
```

We continue chaining `Observable`s in this way through additional calls to map, filter, and so on. The following illustrates the chain at this stage when nothing is subscribed. The private _subscribe property on each `Observable` holds the function that has been passed in to create the `Observer`, but at the moment it is all potential - hence the dotted arrow.



No observers are created for any of the **`Observable`**s until a subscribe is called at the end of the chain. At that time, the `subscribe` methods are called, creating the **`Observer`** that backs each **`Observable`**, all the way up the chain and the **`Observable`** chain becomes "live" - ready to process events. The following shows the situation once the **`Observer`**s are live:



The fun thing is that after `Observable.subscribe` is called, really the `Observable` object's job is done. The stream is maintained by the chain of observers and all the delegate next methods that have been handed to them when set up by the `Observable`. Similarly, the complete methods are delegated to be invoked by the function returned by `Observable.subscribe`. This is done automatically, for example, by `takeUntil`:

```
// creates a child Observable that is detached when the given Observable fires
takeUntil<CutoffInput>(cutoff: Observable<CutoffInput>): Observable<Input> {
  return new Observable<Input>(observer => {
      cutoff.subscribe(_=>observer.complete());
      return this.subscribe(e=>observer.next(e), () => observer.complete())
  });
}
```

The subscription to the `cutoff Observable` that is passed in, completes the downstream `Observable`.

Related to `takeUntil`, is `flatMap`, in the sense that it also observes another `Observable`, which affects its downstream notifications. It has the following signature:

```
// when this Observable occurs, create an Observable downstream
// from the specified stream creator.
// output is "flattened" into the original stream
flatMap<Output>(streamCreator: (value: Input) => Observable<Output>)
  : Observable<Output>
```

In this case, however, `flatMap` passes through notifications from the observed `Observable` to its child stream.

The function `streamCreator` returns an `Observable` that is to be observed. In the mouse-dragging example above it is a stream based on mousemove. Every notification from `flatMap's` upstream `Observer` causes `streamCreator` to be called anew with the original stream's value, and subscribe to be called on the resulting stream. The subscription causes the value from the observed `Observable` to be passed downstream.

7.8. Implement a simple observable with methods including flatMap based on the description above.

8. Combining Higher-Order Functions

The really exciting aspect of higher-order function support in languages like JavaScript is that it allows us to combine simple reusable functions in sophisticated ways. We've already seen how functions like map, filter and reduce can be chained to flatten the control flow of data processing. In this section we will look at some tricks that allow us to use functions that work with other functions in convenient ways.

Curried Functions

Let's say we want a function for computing the volume of cylinders, parameterised by the approximation for π that we plan to use:

```
function cylinderVolume(pi: number, height: number, radius: number):
number {
   return pi * radius * radius * height;
}
```

And we invoke it like so:

```
cylinderVolume(Math.PI, 4, 2);
```

Now consider another version of the same function:

```
function cylinderVolume(pi: number) {
  return function(height: number) {
       return function(radius: number) {
       return pi * radius * radius * height;
       }
  }
}
```

This one, we can invoke like so:

```
cylinderVolume(Math.PI)(4)(2);
```

But we have some other options too. For example, we are unlikely to change our minds about what precision approximation of PI we are going to use between function calls. So let's make a local function that fixes PI:

```
const cylVol = cylinderVolume(Math.PI);
```

Which we can invoke when we are ready like so:

```
cylVol(4)(2)
```

What if we want to compute volumes for a whole batch of cylinders of fixed height of varying radii?

```
const radii = [1.2,3.1,4.5, ... ],
      makeHeight5Cylinder = cylVol(5),
      cylinders = radii.map(makeHeight5Cylinder);
```

Or we can make it into a handy function to compute areas of circles:

```
const circleArea = cylVol(1)
```

Such functions are called *curried functions* and they are named after a mathematician named Haskell Curry. This gives you a hint as to what functions look like in the Haskell programming language and its variants.

Composition

Generalises the kind of chained function calls we have being doing above:

```
function compose<U,V,W>(f:(x:V)=>W,g:(x:U)=>V) {
```

```
  return (x:U)=> f(g(x))
}
```

This function lets us combine two functions into a new reusable function. For example, given a messy list of strings representing numbers of various precision:

```
const grades = ['80.4','100.000','90','99.25']
```

We can define a function to parse these strings into numbers and then round them to the nearest whole number purely through composition of two existing functions:

```
const roundFloat = compose(Math.round, Number.parseFloat)
```

And then apply it to the whole set:

```
grades.map(roundFloat)
> [80, 100, 90, 99]
```

Note that `compose` let us define roundFloat without any messing around with anonymous functions and explicit wiring-up of return values to parameters. We call this *tacit* or *point-free style* programming.

Exercise:
8.1. Create a compose function in javascript that takes a variable number of functions as arguments and composes (chains) them. Using the spread operator (...) to take a variable number of arguments as an array and the Array.prototype.reduce method, the function should be very small. Note, you won't be able to give this a satisfactory type in TypeScript until TypeScript supports variadic kinds.

Identity (I-Combinator)
This may seem trivial:

```
function identity<T>(value: T): T {
   return value;
}
```

But it has some important applications:

- In order to wrap a value in a function that can be passed to other functions expecting an accessor function as input.
- For mocking in tests
- For extracting data from encapsulated types (e.g. by passing identity into map).

K-Combinator

The curried K-Combinator looks like:

```
const K = x=> y=> x
```

So it is a function that ignores its second argument and returns its first argument directly. Note the similarity to the head function of our cons list earlier. In fact, we can derive both the head and tail functions used earlier from K and I combinators:

```
const
   K = x=> y=> x,
   I = x=> x,
   cons = x=> y=> f=> f(x)(y),
   head = l=> l(K),
   tail = l=> l(K(I)),
   l = cons(1)(cons(2)(cons(3)(null))),
   forEach = f=> l=> l?(f(head(l)),forEach(f)(tail(l))):null;

forEach(console.log)(l);
1
2
3
```

FYI it has been shown that simple combinators like K and I (at least one other is required) are sufficient to create languages as powerful as lambda calculus without the need for lambdas, e.g. see SKI Combinator Calculus.

Alternation (OR-Combinator)

A function that applies a first function. If the first function fails (returns undefined, false or null), it applies the second function. The result is the first function that succeeded.

```
const or = f=> g=> v=> f(v) || g(v)
```

Basically, it's a curried if-then-else function with continuations. Imagine something like the following data for student names in a unit, then a dictionary of the ids of students in each class:

```
const students = ['tim','sally','sam','cindy'],
      class1 = { 'tim':123, 'cindy':456},
      class2 = { 'sally':234, 'sam':345};
```

We have a function that lets us lookup the id for a student in a particular class:

```
const lookup = class=> name=> class[name]
```

Now we can try to find an id for each student, first from class1 but fall back to class2 if it isn't there:

```
const ids = students.map(or(lookup(class1))(lookup(class2)))
```

Fork-Join Combinator

```
function fork(join, f, g) {
   return value => join(f(value), g(value));
}
```

Exercise:

8.4. Use the fork-join combinator to compute the average over a sequence of numeric values.

8.5. Add Type annotations to the above definition of the fork function. How many distinct type variables do you need?

Unary

Here's an interesting example:

```
['1','2','3'].map(parseInt);
```

We are converting an array of strings into an array of int. The output will be [1,2,3] right? WRONG!

```
['1','2','3'].map(parseInt);
> [1, NaN, NaN]
```

What the ...!

But:

```
parseInt('2')
> 2
```

Exercises:

8.6. From the docs for Array.map and parseInt can you figure out why the above is happening?

8.7. Write a function called unary that takes a binary function and a value to bind to its first argument, and returns a unary function. What is its fully specified TypeScript type signature?

- Flip - e.g. applied to map(Iterable,fn) to create mapApplyFn(Iterable).

- Compose (binary, n-ary)
- Pipe (reverse direction of compose)

9. Lambda Calculus

The Lambda Calculus is a model of computation developed in the 1930s by the mathematician Alonzo Church. You are probably aware of the more famous model for computation developed around the same time by Alan Turing: the *Turing Machine.* However, while the Turing Machine is based on a hypothetical physical machine (involving tapes from which instructions are read and written) the Lambda Calculus was conceived as a set of rules and operations for function abstraction and application. It has been proven that, as a model of computation, the Lambda Calculus is just as powerful as Turing Machines, that is, any computation that can be modelled with a Turing Machine can also be modeled with the Lambda Calculus.

The Lambda Calculus is also important to study as it is the basis of functional programming. The operations we can apply to Lambda Calculus expressions to simplify (or *reduce*) them, or to prove equivalence, can also be applied to pure functions in a programming language.

Lambda Calculus expressions are written with a standard system of notation. It is worth looking at this notation before studying haskell-like languages because it was the inspiration for Haskell syntax. Here is a simple *Lambda Abstraction* of a function:

$\lambda x. x$

The λ simply denotes the start of a function expression, then follows a list of parameters (in this case we have only a single parameter called x) terminated by "." Then follows the function body, an expression returned by the function when it is *applied*. The variable x is said to be *bound* to the parameter. Variables that appear in the function body but not in the parameter list are said to be *free*. The above lambda is equivalent to the JavaScript expression: `x => x`

We have already discussed combinators in JavaScript, now we can give them a more formal definition: *a combinator is a lambda expression (function) with no free variables*.

Exercise:

9.1. When we discussed Combinators, we gave this function a name. What was it?

Some things to note about such a lambda abstraction:

- It has no name, it is *anonymous*. Note that anonymous functions in languages like JavaScript are also frequently called *lambda expressions*, or just *lambdas*. Now you know why.

- The names of variables bound to parameters in a lambda expression are only meaningful within the context of that expression. Thus, $\lambda x.\, x$ is semantically equivalent (or *alpha equivalent*) to $\lambda y.\, y$ or any other possible renaming of the variable.
- Lambda functions can have multiple parameters in the parameter list, e.g.: $\lambda xy.\, x\, y$, but they are curried (e.g. a sequence of nested univariate functions) such that $\lambda x.\, \lambda y.\, x\, y = \lambda xy.\, x\, y$

What can we do with such a lambda expression? Well we can *apply*[3] it to another expression:
$(\lambda x.\, x)\, y$

We can reduce this expression to a simpler form by a substitution, indicated by a bit of intermediate notation:
$x\,[x := y]$
Now we perform the substitution in the body of the expression and throw away the head, since all the bound variables are substituted, leaving only:
y
This first reduction rule, substituting the arguments of a function application to all occurrences of that parameter inside the function body, is called *beta reduction*.
The next rule arises from the observation that, for some lambda term M that does not involve x:
$\lambda x\,.\, M\;\; x$
is just the same as M. This last rule is called *eta conversion*.

One thing to note about the lambda calculus is that it does not have any such thing as a global namespace. All variables must be:
- Parameters from some enclosing lambda expression (note, below we start using labels to represent expressions - these are not variables, just placeholders for an expression that can be substituted for the label).
- Immutable - there is no way to assign a new value to a variable from within a lambda expression.

This makes the language and its evaluation very simple. All we (or any hypothetical machine for evaluating lambda expressions) can do with a lambda is apply the three basic alpha, beta and eta reduction and conversion rules. Here's a fully worked example of applying the different rules to reduce an expression:

(λz.z) (λa.a a) (λz.z b)

⇒ (λz[z:=λa.a a].z)(λz.z b) => BETA Reduction

⇒ (λa.a a) (λz.z b)

[3] The same way we can apply anonymous functions to an argument in JavaScript.

⇒ λa[a:=λz.z b].a a => BETA Reduction

⇒ (λz.z b) (λz.z b)

⇒ (λz[z:=(λz.z b)].z b) => BETA Reduction

⇒ (λz.z b) b

⇒ λz [z:=b].z b => BETA Reduction

⇒ b b

Note that function application is *left-associative*. This means that BETA reduction is applied left to right, i.e. ((λz.z) (λa.a a)) (λz.z b).

And yet, this simple calculus is sufficient to perform computation. We can model any of the familiar programming language constructs with lambda expressions. For example, Booleans[4]:

TRUE = $\lambda xy.\ x$
FALSE = $\lambda xy.\ y$
Note that we are using the same trick here that we used with the head and rest functions for our cons list, i.e. returning either the first or second parameter to make a choice between two options. Now we can make an IF expression:

IF = $\lambda btf.\ b\ t\ f$

IF TRUE returns the expression passed in as t and IF FALSE returns the expression passed in as f. Now we can make Boolean operators:

AND = $\lambda xy.$ IF $x\ y$ FALSE
OR = $\lambda xy.$ IF x TRUE y
NOT = $\lambda x.$ IF x FALSE TRUE

These restrictions also make it a bit difficult to see how lambda calculus can be a general model for useful computation. For example, how can we have a loop? How can we have recursion if a lambda expression does not have any way to refer to itself?

The first hint to how loops might be possible with lambda calculus is the observation that some expressions do not simplify when beta reduced. For example:

1) $(\lambda x\ .\ x\ \ x)\ (\lambda y.\ y\ y)$
2) $(\lambda x\ [x := \lambda y.\ y\ y]\ .\ x\ \ x\ )$

[4] Here we are assigning labels (TRUE/FALSE/IF/etc) to expressions - these are not variables, just placeholders for an expression that can be substituted for the label.

3) $(\lambda y\,.\,y\ \ y)\,(\lambda y.\,y\,y)$ - which is alpha equivalent to what we started with, so goto (1).

Thus, the reduction would go on forever. Such an expression is said to be *divergent*. However, if a lambda function is not able to refer to itself it is still not obvious how recursion is possible.

The answer is due to the American mathematician Haskell Curry and is called the fixed-point or Y combinator:

$\lambda f.\,(\lambda x\,.f\,(x\,x)\,)\,(\lambda x.f\,(x\,x)\,)$

Exercise:

9.2. Try using Beta reduction on the application of the Y combinator to another function (e.g. g). A miracle occurs!

If we try to directly translate the above version of the Y-combinator into JavaScript we get the following:

```
const Y = f=> (x => f(x(x)))(x=> f(x(x))) // warning infinite recursion ahead!
```

Which we can then try to apply as follows:

```
// A simple function that recursively calculates 'n!'.
const FAC = f => n => n>1 ? n * f(n-1) : 1
const fac = Y(FAC)
console.log(fac(6))
... stack overflow
```

And bad things happen. Why? JavaScript uses *Eager* or *Strict Evaluation*. This means expressions are evaluated immediately they are encountered by the interpreter. Let's try doing beta reduction on the y-combinator applied to the above FAC function in lambda calculus, assuming strict evaluation:

λf[f := FAC]. (λx. f (x x)) (λx. f (x x))
λx[(λx. FAC(x x))]. FAC(x x)
FAC ((λx. FAC (x x)) (λx. FAC (x x)))
FAC (λx[(λx. FAC(x x))]. FAC (x x))
FAC (FAC ((λx. FAC (x x)) (λx. FAC (x x))))
FAC (FAC ((λx[(λx. FAC (x x))]. FAC (x x))))
FAC (FAC (FAC ((λx. FAC (x x)) (λx. FAC (x x)))))

It just goes on forever expanding nested expressions of FAC without actually invoking it and forcing evaluation of the expression involving n that would eventually force it to terminate. How do we restore the lazyness necessary to make progress in this recursion? (hint: it involves wrapping some part of Y in another lambda.

Exercise:
9.3. Write a version of the Y-Combinator in JavaScript such that Y(Fac)(6) successfully evaluates to 120.

10. The Haskell Family and PureScript

JavaScript is a multiparadigm language that due to its support for functions as objects, closures and, therefore, higher-order functions, is able to be used as in a functional programming style. However, if you are really enamored with currying and combining higher-order functions, then it really makes a lot of sense to use a language that is actually designed for it.

There are a number of purpose-built Functional Programming languages. Lisp (as we have already discussed) is the original, but there are many others. Scheme is a Lisp derivative, as is (more recently) Clojure. SML and its derivatives (e.g. OCAML, F#, etc) form another family of functional programming languages. However, the strongest effort to build a language that holds to the principles of lambda-calculus inspired functional programming such as immutability (purity) is the Haskell family. There are a number of efforts to bring haskell-like purity to web programming, inspired by the potential benefits the functional-style holds for managing complex state in asynchronous and distributed applications. Firstly, it is possible to compile haskell code directly to JavaScript (using GHCJ) although the generated code is opaque and requires a runtime. Another promising and increasingly popular haskell-inspired language for client-side web development is Elm, although this again requires a runtime. Also, Elm is rather specialised for creating interactive web apps.

The JavaScript targeting Haskell derivative we are going to look at now is *PureScript*. The reason for this choice is that PureScript generates standalone and surprisingly readable JavaScript. For a full introduction to the language, the PureScript Book, written by the language's creator, is available for free. However, in this unit we will only make a brief foray into PureScript as a segue from JavaScript to Haskell. To avoid overwhelming ourselves with minor syntactic differences we will also endeavor to stick to a subset of PureScript that is syntactically the same as Haskell.

Hello Functional Language!
Without further ado, here is some PureScript code[5]. Fibonacci number computation is often called the "hello world!" of functional programming:

```
fibs :: Int -> Int
fibs 0 = 1
fibs 1 = 1
```

[5] And here is a quick guide to setting up PureScript, Compiling and running this code.

```
fibs n = fibs (n-1) + fibs (n-2)
```

Woah! A function for Fibonacci numbers that is about as minimal as you can get! And the top line, which just declares the type of the function, is often optional - depending on whether the compiler can infer it from the context. Having said that, it's good practice to include a type declaration, especially for top-level functions. This function takes an Int parameter, and returns an Int. Note that the arrow shorthand for the function type definition is highly reminiscent of the JavaScript fat-arrow… though skinnier.

The next three lines define the actual logic of the function, which very simply gives a recursive definition for the nth Fibonacci number. This definition uses a feature common to many functional programming languages: *pattern matching*. That is, we define the fibs function three times, with the first two definitions handling the *base cases*. It says, literally: "the 0th and 1st fibs are both 1". The last line defines the general case, that the remaining fibonacci numbers are each the sum of their two predecessors. Note, this definition is not perfect. Calling:
`fibs -1`
would be a *bad idea*. Good practice would be to add some exceptions for incorrect input to our function. In a perfect world we would have a compiler that would check types dependent on values (actually, languages that support *dependent types* exist, e.g. the *Idris* language is an interesting possible successor to Haskell in this space).

One thing you will have noticed by now is that Haskell-like languages are light on syntax. Especially use of brackets is minimal, and typically to be avoided when evaluation order can be inferred correctly by the compiler's application of lambda-calculus inspired precedence rules for function and operator application.

We can define a main function for our program, that maps the `fibs` function to a ("Nil"-terminated) linked-list of numbers and displays them to the console like so:

```
main = log $ show $ map fibs $ 1..10
```

and here's the output when you run it from the command line:

```
(1 : 2 : 3 : 5 : 8 : 13 : 21 : 34 : 55 : 89 : Nil)
```

I'm omitting the type declaration for main because the type for functions that have input-output side-effects is a little more complicated, differs from haskell - and the compiler doesn't strictly need it yet anyway.

The above definition for main is a chain of functions and the order of evaluation (and hence how you should read it) is right-to-left. The `$` symbol is actually shorthand for brackets around everything to the symbol's right. In other words, the above definition for main is equivalent to:

```
main = log ( show ( map fibs ( 1..10 )))
```

The `$` is not just magic syntax (a keyword) but is actually an operator defined in the PureScript Prelude like so:

```
infixr 0 apply as $
```

That is, `$` is an infix, right associative operator with binding precedence `0` (the lowest) that invokes the apply function:

```
apply f x = f x
```

Woah! What is `f` and what is `x`? Well, in PureScript functions are generic by default - but we (and the compiler) can infer, since `f x` is a function call with argument `x`, that `f` is a function and `x` is… anything. So `apply` literally applies the function `f` to the argument `x`. Since the binding precedence of the `$` operator is so low compared to most things that could be placed to its right, brackets are (usually) unnecessary.

Exercise:
10.1. If one didn't happen to like the fact that function chaining with the $ operator reads right to left, how would one go about creating an operator that chains left to right? (Hint: infixl is a thing and you will need to make a slightly different apply function also).

So anyway, back to the chain of functions in main:

```
main = log $ show $ map fibs $ 1..10
```

`log` is a function that wraps JavaScript's console.log
`show` is a function that is overloaded to convert various types to strings. In this case, we'll be showing a List of Int.
`map` is (equivalent to our old friend from our JavaScript exercises) a function that applies a function to stuff inside a… let's call it a container for now… in this case our Container is a List.
`1..10` uses the .. (range) infix operator to create a List of Int between 1 and 10.

Peeking under the hood

So all this may seem pretty foreign, but actually, since we've already covered many of the functional programming fundamentals in JavaScript, let's take a look at the JavaScript code that the PureScript compiler generates for fibs and main and see if anything looks familiar. Here's fibs, exactly as it comes out of the compiler:

```
var fibs = function (v) {
   if (v === 0) {
       return 1;
   };
   if (v === 1) {
       return 1;
   };
   return fibs(v - 1 | 0) + fibs(v - 2 | 0) | 0;
};
```

Woah! It's pretty much the way a savvy JavaScript programmer would write it. At first glance the code generated for main is a bit denser. Here it is, again as generated by the compiler but I've inserted some line breaks so we can see it a little more clearly:

```
var main = Control_Monad_Eff_Console.log(
  Data_Show.show(
    Data_List_Types.showList(Data_Show.showInt)
  )(
    Data_Functor.map
      (Data_List_Types.functorList)(fibs)(Data_List.range(1)(10))
  )
);
```

Each of the functions lives in an object that encapsulates the module where it is defined. That's pretty standard JavaScript practice. The rest is just function calls (application). The call to the range function is interesting:

```
Data_List.range(1)(10)
```

Woah! It's a curried function! Data_List.range(1) returns a function that creates lists of numbers starting from 1. The second call specifies the upper bound.

Exercise:
10.2. What other functions called in the JavaScript code generated for the above definition of main are curried? Why?

Tail Call Optimisation

Our definition for fibs was recursive. This has a nice declarative style about it. The definition is very close to a mathematical definition. But at some point in your training for imperative programming you will have most likely been told that recursion is evil and inefficient. Indeed,

we've seen at the start of this course that there is overhead due to creating new stack frames for each function call. Looping recursively creates a new stack frame for each iteration and so our (finite) stack memory will be consumed linearly with the number of iterations. However, there are certain patterns of recursive function calls that are compiler can easily recognise and replace with an iterative loop. We can see this happening directly in PureScript if we reconfigure our fibs definition to use a *tail call*.

```
fibs n = f n 0 1
  where
    f 0 _ b = b
    f i a b = f (i-1) b (a+b)
```

In general, as we have seen with `$`, PureScript (and Haskell) have relatively few keywords, instead preferring functions and operators built with the language itself in the Prelude. The `where` keyword, however, is one of the exceptions. It allows us to make some local definitions inside the context of another function. Here we define `f` whose first parameter is an iteration counter, whose base case is 0. The key feature of `f` is that its recursive call is the very last thing to happen in the function body. That is, it is in the *tail position*.

The other important aspect of PureScript that we are encountering for the first time in the above definition is that indentation is used to determine scope (as in python).

Here's the JavaScript that is generated this time:

```
var fibs = function (n) {
   var f = function ($copy_v) {
       return function ($copy_v1) {
           return function ($copy_b) {
               var $tco_var_v = $copy_v;
               var $tco_var_v1 = $copy_v1;
               var $tco_done = false;
               var $tco_result;
               function $tco_loop(v, v1, b) {
                   if (v === 0) {
                       $tco_done = true;
                       return b;
                   };
                   $tco_var_v = v - 1 | 0;
                   $tco_var_v1 = b;
                   $copy_b = v1 + b | 0;
```

```
                    return;
                };
                while (!$tco_done) {
                    $tco_result = $tco_loop($tco_var_v, $tco_var_v1, $copy_b);
                };
                return $tco_result;
            };
        };
    };
    return f(n)(0)(1);
};
```

Obviously, it's a less direct translation than was generated for our previous version of fibs. However, you can fairly easily understand it still. Hint, the "tco_" prefix in many of the generated variable names stands for "Tail Call Optimisation" and the local function f is a curried function, as are all functions of more than one argument in PureScript. The important thing is that the recursive call is gone, replaced by a while loop.

We have seen all we need for now of PureScript. It's a small but nicely put together language. It takes the best features of Haskell and reinterprets some of them quite cleverly to achieve relatively seamless interop with JavaScript. However, it's still a bit niche. For the remainder of this unit we'll dive more deeply into Haskell, which has a long history and is supported by a very large and active community across academia and industry.

11. Introduction to Haskell

"Haskell Programming from First Principles" by Allen and Moronuki is a recent and excellent introduction to haskell that is quite compatible with the goals of this course. The ebook is not too expensive, but unfortunately, it is independently published and hence not available from our library. "Learn you a Haskell" by Miran Lipovaca is a freely available alternative that is also a useful introduction.

&

Both the simplest and tail-recursive versions of our PureScript **`fibs`** code are also perfectly legal haskell code. The main function will be a little different, however:

```
main :: IO ()
main = print $ map fibs [1..10]
```

I've included the type signature here although it's still not absolutely necessary, but it's an elephant in the room that we need to address sooner or later. So **`main`** is a function that takes no inputs (no need for **`->`** with something on the left) and it returns something in the **`IO`** *monad.* Without getting into it too much, monads are special functions that can also wrap some other value. In this case, the main function just does output, so there is no wrapped value and hence

the `()` (called *unit*) indicates this. You can think of it as being similar to the void type in C, Java or TypeScript.

What this tells us is that the main function produces an `IO` *side effect*. This mechanism is what allows Haskell to be a *pure* functional programming language while still allowing you to get useful stuff done. Side effects can happen, but when they do occur they must be neatly bundled up and declared to the type system, in this case through the `IO` monad. For functions without side-effects, we have strong, compiler checked guarantees that this is indeed so (that they are pure).

By the way, once you are in the `IO` monad, you can't easily get rid of it. Any function that calls a function that returns an `IO` monad, must have `IO` as its return type. Thus, effectful code is possible, but the type system ensures we are aware of it and can limit its taint. The general strategy is to use pure functions wherever possible, and push the effectful code as high in your call hierarchy as possible. Pure functions are much more easily reusable in different contexts.

The **`print`** function is equivalent to the PureScript **`log $ show`**. Haskell also defines **`show`** for many types in the Prelude, but **`print`** in this case invokes it for us. The other difference here is that square brackets operators are defined in the prelude for linked lists. In PureScript they were used for Arrays - which (in PureScript) don't have the range operator (..) defined so I avoided them.

Another thing to note about Haskell at this stage is that its evaluation is *lazy by default*. Laziness is of course possible in other languages (as we have seen in JavaScript), and there are many lazy data-structures defined and available for PureScript (and most other functional languages).

However, lazy by default sets Haskell apart. It has pros and cons, on the pro side:

- It can make certain operations more efficient, for example, we have already seen in JavaScript how it can make streaming of large data efficient
- It can enable infinite sequences to be defined and used efficiently (this is a significant semantic difference)
- It opens up possibilities for the compiler to be really quite smart about its optimisations.

But there are definitely cons:

- It can be hard to reason about run-time performance
- Mixing up strict and lazy evaluation (which can happen inadvertently) can lead to (for example) O(n^2) behaviour in what should be linear time processing.

By the way, since it's lazy-by-default, it's possible to transfer the version of the Y-combinator given in the previous section into haskell code almost as given in Lambda Calculus:

```
y = \f -> (\x -> f (x x)) (\x -> f (x x))
```

However, to get it to type-check one has to force the compiler to do some unsafe type coercion. The following (along with versions of the Y-Combinator that do type check in haskell) are from [an excellent Stack Overflow post]():

```
import Unsafe.Coerce
y :: (a -> a) -> a
y = \f -> (\x -> f (unsafeCoerce x x)) (\x -> f (unsafeCoerce x x))
main = putStrLn $ y ("circular reasoning works because " ++)
```

12. Creating Expressive Declarative Functions

Consider the following pseudocode for a simple recursive definition of the Quick Sort algorithm:

```
QuickSort list:
  Take head of list as a pivot
  Take tail of list as rest
  return
        QuickSort( elements of rest < pivot ) ++ (pivot : QuickSort( elements of rest >= pivot ))
```

We've added a bit of notation here: *a* : *l* inserts *a* ("cons"es) to the front of a list *l* ; *l1* ++ *l2* is the concatenation of lists *l1* and *l2*.

In JavaScript the fact that we have anonymous functions through compact arrow syntax and expression syntax if (with ? :) means that we can write pure functions that implement this recursive algorithm in a very functional, fully-curried style. However, the language syntax really doesn't do us any favours!

For example,

```
const
  sort = order=>
    list=> !list ? null :
      (pivot=>rest=>
        (lesser=>greater=>
          concat(sort(order)(lesser))
                (cons(pivot)(sort(order)(greater)))
        )(filter(a=> order(a)(pivot))(rest))
         (filter(a=> !order(a)(pivot))(rest))
      )(head(list))(tail(list))
```

Consider, the following, more-or-less equivalent haskell implementation:

```
sort [] = []
sort (pivot:rest) = lesser ++ [pivot] ++ greater
  where
    lesser = sort $ filter (<pivot) rest
    greater = sort $ filter (>=pivot) rest
```

Haskell helps with a number of language features. First, is *pattern matching*. Pattern matching is like function overloading that you may be familiar with from languages like Java or C++ - where the compiler matches the version of the function to invoke for a given call be matching the type of the parameters to the type of the call - except in Haskell the compiler goes a bit deeper to inspect the values of the parameters.

There are two declarations of the sort function above. The first handles the base case of an empty list. The second handles the general case, and pattern matching is again used to destructure the lead cons expression into the `pivot` and `rest` variables. No explicit call to `head` and `tail` functions is required.

The next big difference is the haskell style of function application - which has more in common with lambda calculus than JavaScript. The expression `f x` is application of the function `f` to whatever `x` is. This helps to cut down massively on bracket creep.

Another thing that helps with readability is *infix operators*. For example, ++ is an infix binary operator for list concatenation. The : operator for cons is another. There is also the aforementioned $ which gives us another trick for removing brackets, and finally, the < and >= operators. Note, that infix operators can also be curried and left only partially applied as in (<pivot).

Next, we have the `where` which lets us create locally scoped variables - without technically ending the expression and without the need for the trick I used in the JavaScript version of using the parameters of anonymous functions as locally scoped variables.

Finally, you'll notice that the haskell version of sort appears to be missing a parameterisation of the order function. Does this mean it is limited to number types? In fact, no - from our use of < and >= the compiler has inferred that it is applicable to any ordered type. More specifically, to any type in the *type class* Ord.

I deliberately avoided the type declaration for the above function because, (1) we haven't really talked about types properly yet, and (2) because I wanted to show off how clever Haskell type inference is. However, it is actually good practice to include the type signature. If one were to load the above code, without type definition, into GHCI (the Haskell REPL), one could interrogate the type like so:

> :t sort

```
sort :: Ord t => [t] -> [t]
```

Thus, the function `sort` has a generic type-parameter `t` which is *constrained* to be in the `Ord` type class. It's input parameter is a list of `t`, as is its return type. This is also precisely the syntax that one would use to declare the type explicitly. Usually, for all top-level functions in a Haskell file it is good practice to explicitly give the type declaration. Although, it is not always necessary, it can avoid ambiguity in many situations, and secondly, once you get good at reading Haskell types, it becomes useful documentation.

Here's another refactoring of the quick-sort code. This time with type declaration because I just said it was the right thing to do:

```
sort :: Ord t => [t] -> [t]
sort [] = []
sort (pivot:rest) = below pivot rest ++ [pivot] ++ above pivot rest
 where
   below p = partition (<p)
   above p = partition (>=p)
   partition comparison = sort . filter comparison
```

This version makes a point about point-free style and how it can lead to not only compact code, but also code that can read almost like a natural language declarative definition of the algorithm. Here, we use the . operator for function composition. Although it looks like the `comparison parameter` could also go away here with eta conversion, actually the low precedence of the . operator means there is (effectively) an implicit parenthesis around `filter comparison.`

Haskell has a number of features that allow us to express ourselves in different ways. Above we used a `where` clause to give a post-hoc, locally-scoped declaration of the `below` and `above` functions. Alternately, we could define them at the start of the function body with `let <variable declaration expression> in <body>.` Or we can use `let`, `in` and `where` all together, like so:

```
sort :: Ord t => [t] -> [t]
sort [] = []
sort (pivot:rest) = let
   below p = partition (<p)
   above p = partition (>=p)
 in
   below pivot rest ++ [pivot] ++ above pivot rest
 where
   partition comparison = sort . filter comparison
```

13. Data and Typeclasses

We can declare custom types for data in Haskell using the `data` keyword. Consider the following declaration of our familiar cons list[6]:

```
data ConsList = Nil | Cons Int ConsList
```

The | operator looks rather like the union type operator in TypeScript, and indeed it serves a similar purpose. Here, a `ConsList` is defined as being a composite type, composed of either Null or a Cons of an Int value and another `ConsList`. This is called an "***algebraic data type***" because | is like an "or", or algebraic "sum" operation for combining elements of the type while separating them with a space is akin to "and" or a "product" operation.

Note that neither `Null` or `Cons` are built in. They are simply labels for constructor functions for there different versions of a `ConsList`. You could equally well call them `EndOfList` and `MakeList` or anything else that's meaningful to you. `Nil` is a function with no parameters, `Cons` is a function with two parameters. `Int` is a built-in primitive type for limited-precision integers.

Now we can create a small list like so:

```
l = Cons 1 $ Cons 2 $ Cons 3 Nil
```

And we can create a function to determine a `ConsList`'s length using pattern matching; to not only create different definitions of the function for each of the possible instances of a ConsList, but also to destructure the non-empty Cons:

---

[6] Note that such a definition is made completely redundant by Haskell's wonderful built-in lists, where [] was the empty list, and : was an infix cons operator.

```
consLength :: ConsList -> Int
consLength Nil = 0
consLength (Cons _ rest) = 1 + consLength rest
```

Since we don't care about the head value in this function, we match it with _, which effectively ignores it.

Record Syntax

Consider the following simple record data type:

```
data Student = Student Int String Int
```

A Student has three fields, mysteriously typed Int, String and Int. Let's say my intention in creating the above data type was to store a student's id, name and mark. I would create a record like so:

```
> t = Student 123 "Tim" 95
```

Here's how one would search for the student with the best mark:

```
best :: [Student] -> Student -> Student
best [] b = b
best (a@(Student _ _ am):rest) b@(Student _ _ bm) =
  if am > bm
  then best rest a
  else best rest b
```

The `@` notation, as in `b@(Student _ _ bm)` stores the record itself in the variable `b` but also allows you to unpack its elements, e.g. `bm` is bound to mark.

To get the data out of a record I would need to either destructure using pattern matching, as above, every time, or create some accessor functions:

```
id (Student n _ _) = n
name (Student _ n _) = n
mark (Student _ _ n) = n
> name t
"Tim"
```

It's starting to look a bit like annoying boilerplate code. Luckily, Haskell has another way to define such record types, called *record syntax*:

```
data Student = Student { id::Integer, name::String, mark::Int }
```

This creates a record type in every way the same as the above, but the accessor functions `id`, `name` and `mark` are created automatically.

Typeclasses

Haskell uses "type classes" as a way to associate functions with types. A type class is like a promise that a certain type will have specific operations and functions available. You can think of it as being similar to a TypeScript interface. Despite the name however, it is not like a TypeScript class, since a type class is does not actually define the functions themselves. The functions are defined in "instances" of the type class. A good starting point for gaining familiarity with type classes is seeing how they are used in the standard Haskell prelude. From GHCi we can ask for information about a specific typeclass with the :i command, for example, Num is a typeclass common to numeric types:

```
GHCi> :i Num
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
  {-# MINIMAL (+), (*), abs, signum, fromInteger, (negate | (-)) #-}
        -- Defined in `GHC.Num'
instance Num Word -- Defined in `GHC.Num'
instance Num Integer -- Defined in `GHC.Num'
instance Num Int -- Defined in `GHC.Num'
instance Num Float -- Defined in `GHC.Float'
instance Num Double -- Defined in `GHC.Float'
```

This is telling us that for a type to be an instance of the Num typeclass, it must provide the operators +, * and the functions abs, signum and fromInteger, and either (-) or negate. The last is an option because a default definition exists for each in terms of the other. The last five lines (beginning with "instance") tell us which types have been declared as instances of Num and hence have definitions of the necessary functions. These are Word, Integer, Int, Float and Double. Obviously this is a much more finely grained set of types than JavaScript's universal "number" type. This granularity allows the type system to guard against improper use of numbers that might result in loss in precision or division by zero.

The main numeric type we will use in this course is `Int`, i.e. fixed-precision integers.

Note some obvious operations we would likely need to perform on numbers that are missing from the Num typeclass. For example, equality checking. This is defined in a separate type class Eq, that is also instanced by concrete numeric types like Int:

```
> :i Eq
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  {-# MINIMAL (==) | (/=) #-}
...
instance Eq Int
...
```

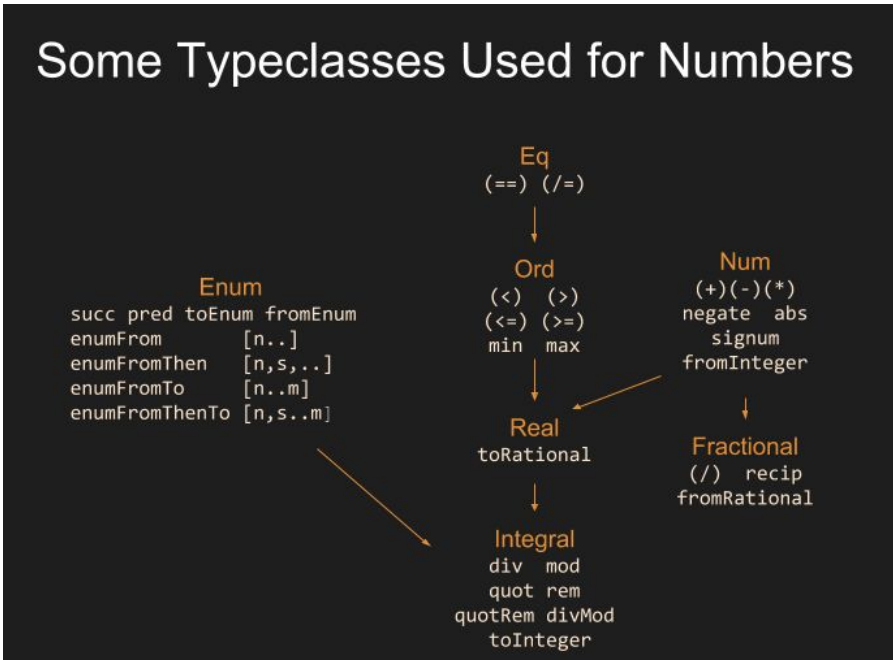
Note again that instances need implement only == or /= (not equal to), since each can be easily defined in terms of the other. Still we are missing some obviously important operations, e.g., what about inequalities? These are defined in the Ord type class:

```
> :i Ord
class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<) :: a -> a -> Bool
  (<=) :: a -> a -> Bool
  (>) :: a -> a -> Bool
  (>=) :: a -> a -> Bool
  max :: a -> a -> a
  min :: a -> a -> a
  {-# MINIMAL compare | (<=) #-}
```

The `compare` function returns an Ordering:

```
> :i Ordering
data Ordering = LT | EQ | GT
```

A custom data type can be made an instance of `Ord` by implementing either `compare` or `<=`. The definition `Eq a => Ord a` means that anything is an instance of `Ord` must also be an instance of `Eq`. Thus, typeclasses can build upon each other into quite rich hierarchies:



Creating custom instances of type classes
If we have our own data types, how can we make standard operations like equality and inequality testing work with them? Luckily, the most common type classes can easily be instanced automatically through the `deriving` keyword. example, if we want to define a `Suit` type for a card game.

```
data Suit = Spade|Club|Diamond|Heart
 deriving (Eq,Ord,Enum,Show)
```

```
> Spade < Heart
True
```

The `Show` typeclass allows the data to be converted to strings with the `show` function (e.g. so that GHCi can display it). The `Enum` typeclass allows enumeration, e.g.:

```
> [Spade .. Heart]
[Spade,Club,Diamond,Heart]
```

We can also create custom instances of typeclasses by providing our own implementation of the necessary functions, e.g.:

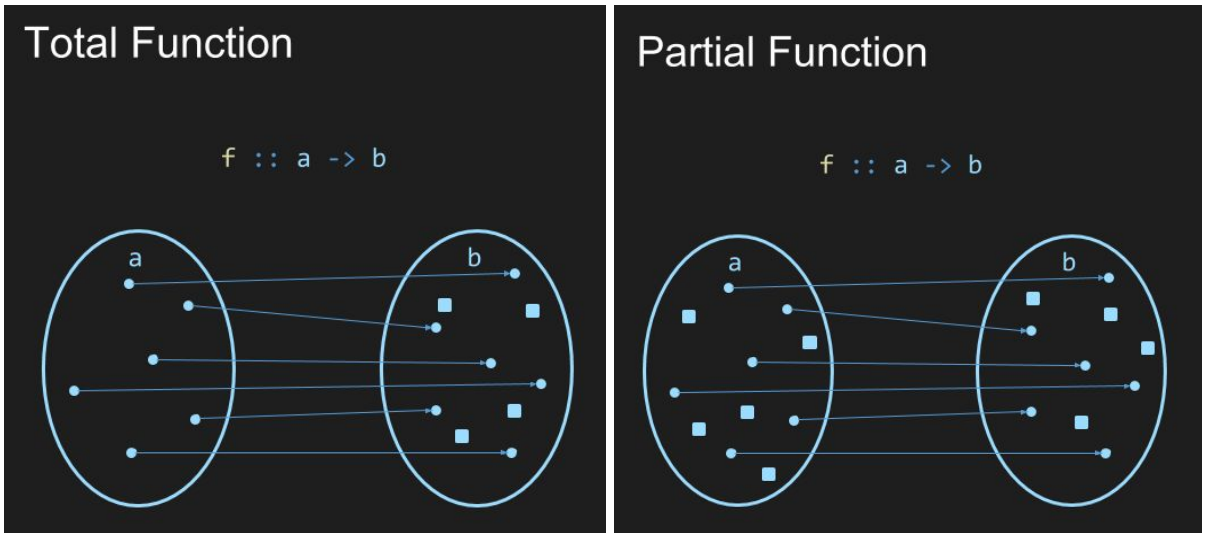
```
instance Show Suit where
 show Spade = "^"
 show Club = "&"
 show Diamond = "O"
 show Heart = "V"
```

```
> [Spade .. Heart]
[^,&,O,V]
```

Maybe
Another important built-in type is Maybe:

```
> :i Maybe
data Maybe a = Nothing | Just a
```

All the functions we have considered so far are assumed to be *total*. That is, the function provides a mapping for every element in the input type to an element in the output type. Maybe allows us to have a sensible return-type for *partial* functions. Functions which do not have a mapping for every input:



For example, the built-in `lookup` function can be used to search a list of key-value pairs, and fail gracefully by returning `Nothing` if there is no matching key.

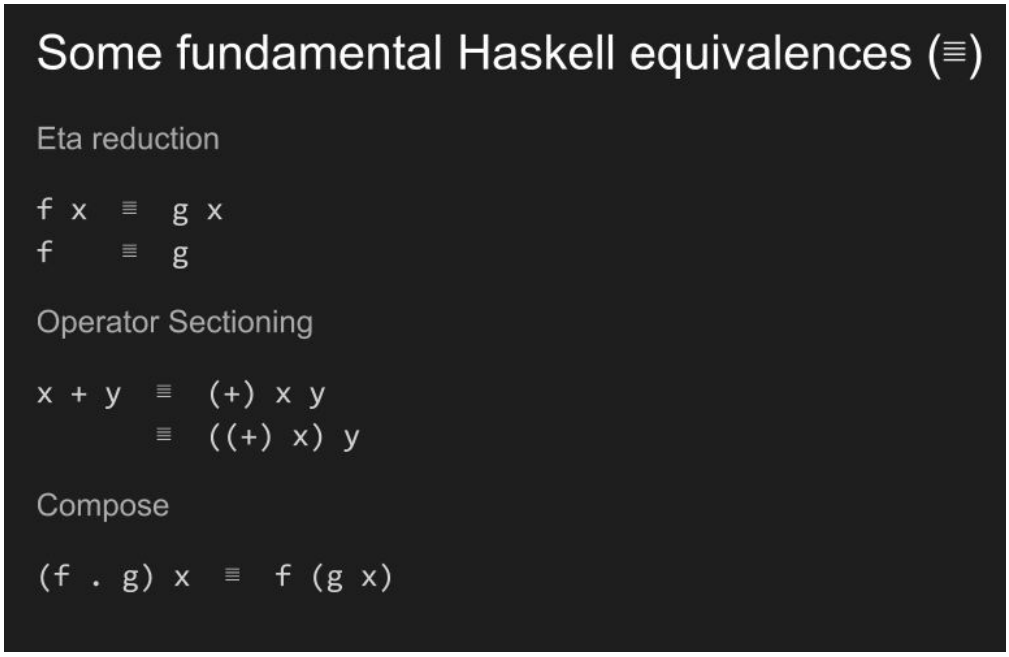
```
phonebook :: [(String, String)]
phonebook = [ ("Bob",   "01788 665242"), ("Fred",  "01624 556442"), ("Alice", "01889
985333") ]

> :t lookup
lookup :: Eq a => a -> [(a, b)] -> Maybe b

> lookup "Fred" phonebook
Just "01624 556442"

> lookup "Tim" phonebook
Nothing
```

We can use pattern matching to extract values from `Maybe` (when we have `Just` a value), or to perform some sensible default behaviour when we have `Nothing`.

```
printNumber name = msg $ lookup name phonebook
where
   msg (Just number)  = print number
   msg Nothing        = print $ name ++ " not found in database"

*GHCi> printNumber "Fred"
"01624 556442"
*GHCi> printNumber "Tim"
"Tim not found in database"
```

14. Refactoring Haskell Code to be Point-Free

The following equivalences make many refactorings possible in Haskell:



We have discussed point-free and tacit coding style earlier in these notes. In particular, eta-reduction works in Haskell the same as in lambda calculus and for curried JavaScript functions. It is easy to do and usually declutters code of unnecessary arguments that help to distill their essence, e.g.:

```
lessThanNum :: Num a => a -> [a] -> [a]
lessThanNum n aList = filter (<n) aList
```

The following is more concise, and once you are used to reading haskell type definitions, just as self evident:

```
lessThanNum :: Num a => a -> [a] -> [a]
lessThanNum n = filter (<n)
```

But the above still has an argument (a point), n. Can we go further?

It is possible to be more aggressive in refactoring code to achieve point-free style by using the compose operator (.):

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(f . g) x = f (g x)
```

To see how to use (.) in `lessThanNum` we need to refactor it to look like the right-hand side of the definition above, i.e. `f (g x)`. For `lessThanNum`, this takes a couple of steps, because the order we pass arguments to (<) matters. Partially applying infix operators like (<n) is called *operator sectioning*. Placing n after < means that it is being passed as the second argument to the operator, which is inconvenient for eta-reduction. Observe that (<n) is equivalent to (n>), so the following is equivalent to the definition above:

```
lessThanNum n = filter (n>)
```

Now we can use the non-infix form of (>):

```
lessThanNum n = filter ((>) n)
```

And we see that if we were to replace filter by f, (>) by g, and n by x, we would have exactly the definition of (.). Thus,

```
lessThanNum n = (filter . (>)) n
```

And now we can apply eta-reduction:

```
lessThanNum = filter . (>)
```

Between operator sectioning, the compose combinator (.), and eta-reduction it is possible to write many functions in point-free form. For example, the flip combinator:

```
flip :: (a -> b -> c) -> b -> a -> c
flip f a b = f b a
```

can also be useful in reversing the arguments of a function or operator in order to get them into a position such that they can be eta-reduced.

In code written by experienced haskellers it is very common to see functions reduced to point-free form. Does it make it for more readable code? To experienced haskellers, many times yes. To novices, perhaps not. When to do it is a matter of preference. Experienced haskellers tend to prefer it, they will argue that it reduces functions like the example one above "to their essence", removing the "unnecessary plumbing" of explicitly named variables. Whether you like it or not, it is worth being familiar with the tricks above, because you will undoubtedly see them used in practice. The other place where point free style is very useful is when you would otherwise need to use a lambda function.

Some more (and deeper) discussion is available on the Haskell Wiki.

Exercises

13.1 Refactor the following function to be point-free:

f a b c = (a+b)*c

(This is clearly an extreme example but is a useful - and easily verified - practice of operator sectioning, composition and eta-reduction.)

14. Functor

We've been mapping over lists and arrays many times, first in JavaScript:

```
console> [1,2,3].map(x=>x+1)
[2,3,4]
```

Now in haskell:

```
GHCi> map (\i->i+1) [1,2,3]
[2,3,4]
```

Or (eta-reduce the lambda to be point-free):

```
GHCi> map (+1) [1,2,3]
[2,3,4]
```

Here's the implementation of map for lists as it's defined in the GHC standard library (the links in the code below are navigable to the actual source):

```
map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs
```

It's easy to generalise this pattern to any data structure that holds one or more values: *mapping a function over a data structure creates a new data structure whose elements are the result of applying the function to the elements of the original data structure.*

In Haskell this pattern is captured in a type class called Functor, which defines a function called *fmap*.

```
class Functor (f :: * -> *) where
  fmap :: (a -> b) -> f a -> f b
```

Naturally, lists have an instance:

```
Prelude> :i []
...
instance Functor [] -- Defined in `GHC.Base'
```

We can actually look up GHC's implementation of fmap for lists and we see:

```
instance Functor [] where
    fmap = map
```

And here's the instance for Maybe:

```
instance  Functor Maybe  where
  fmap _ Nothing      = Nothing
  fmap f (Just a)     = Just (f a)
```

So we can fmap a function over a Maybe without having to unpackage it:

```
GHCi> fmap (+1) (Just 6)
Just 7
```

This is such a common operation that there is an operator alias for fmap: <$>

```
GHCi> (+1) <$> (Just 6)
Just 7
```

Which also works over lists:

```
GHCi> (+1) <$> [1,2,3]
[2,3,4]
```

Lists of Maybes frequently arise. For example, the "mod" operation on integers (e.g. mod 3 2 == 1) will throw an error if you pass 0 as the divisor:

> mod 3 0
*** Exception: divide by zero

We might define a safe modulo function:

```
safeMod :: Integral a => a-> a-> Maybe a
safeMod _ 0 = Nothing
safeMod numerator divisor = Just $ mod numerator divisor
```

This makes it safe to apply safeMod to an arbitrary list of Integral values:

```
> map (safeMod 3) [1,2,0,4]
[Just 0,Just 1,Nothing,Just 3]
```

But how do we keep working with such a list of Maybes? We can map an fmap over the list:

```
GHCi> map ((+1) <$>) [Just 0,Just 1,Nothing,Just 3]
[Just 1,Just 2,Nothing,Just 4]
```

Or equivalently:

```
GHCi> ((+1) <$>) <$> [Just 0,Just 1,Nothing,Just 3]
[Just 1,Just 2,Nothing,Just 4]
```

In addition to lists and Maybes, a number of other built-in types have instances of Functor:

```
GHCi> :i Functor
instance Functor (Either a) -- Defined in `Data.Either'
instance Functor [] -- Defined in `GHC.Base'
instance Functor Maybe -- Defined in `GHC.Base'
instance Functor IO -- Defined in `GHC.Base'
instance Functor ((->) r) -- Defined in `GHC.Base'
instance Functor ((,) a) -- Defined in `GHC.Base'
```

The definition for functions (->) might surprise:

```
instance Functor ((->) r) where
  fmap = (.)
```

So the composition of functions f and g: f . g, is equivalent to ‘mapping’ f over g, e.g. f <$> g.

```
GHCi> f = (+1)
GHCi> g = (*2)
GHCi> (f.g) 3
7
GHCi> (f<$>g) 3
7
```

We can formalise the definition of Functor with two laws:

1. The law of identity [7]

$$\forall x : (id \ <\$> \ x) \equiv x$$

2. The law of composition

$$\forall f, g, x : (f \circ g \ <\$> \ x) \equiv (f \ <\$> \ (g \ <\$> \ x))$$

Note that these laws are not enforced by the compiler when you create your own instances of Functor. You'll need to test them for yourself. Following these laws guarantees that general code (e.g. algorithms) using fmap will also work for your own instances of Functor.

15. Applicative

The applicative introduces a new operator <*> (pronounced "apply"), which lets us apply functions inside a computational context.

For example, a function inside a Maybe can be applied to a value in a Maybe.

```
GHCi> (Just (+3)) <*> (Just 2)
Just 5
```

Or a list of functions [(+1),(+2)]), to things inside a similar context (e.g. a list [1,2,3]).

```
> [(+1),(+2)] <*> [1,2,3]
[2,3,4,3,4,5]
```

Note that lists definition of <*> produces the cartesian product of the two lists, that is, all the possible ways to apply the functions in the left list, to the values in the right list. It is interesting to look at the source for the definition of Applicative for lists on Hackage:

```
instance Applicative [] where
      pure x    = [x]
      fs <*> xs = [f x | f <- fs, x <- xs]  -- list comprehension
```

The definition of <*> for lists uses a *list comprehension*. List comprehensions are a short-hand way to generate lists, using notation similar to mathematical set builder notation. The set builder notation here would be: $\{f(x) \mid f \in fs \wedge x \in xs\}$. In English it means: "the set (Haskell list) of all functions in `fs` applied to all values in `xs`".

[7] Where `id` is the identity function: `id = \i -> i` for example: `id 3 = 3.`

A common use-case for Applicative is applying a binary (two-parameter) function over two Applicative values, e.g.:

```
> (+) <$> Just 3 <*> Just 2
Just 5
```

This is called “lifting” a function over Applicative. Actually, it’s so common that Applicative also defines dedicated functions for lifting binary functions (in the GHC.Base module):

```
> GHC.Base.liftA2 (+) (Just 3) (Just 2)
Just 5
```

It’s also useful to lift binary data constructors over two Applicative values, e.g. for tuples:

```
> (,) <$> Just 3 <*> Just 2
Just (3, 2)
```

Or lifting a data constructor over lists:

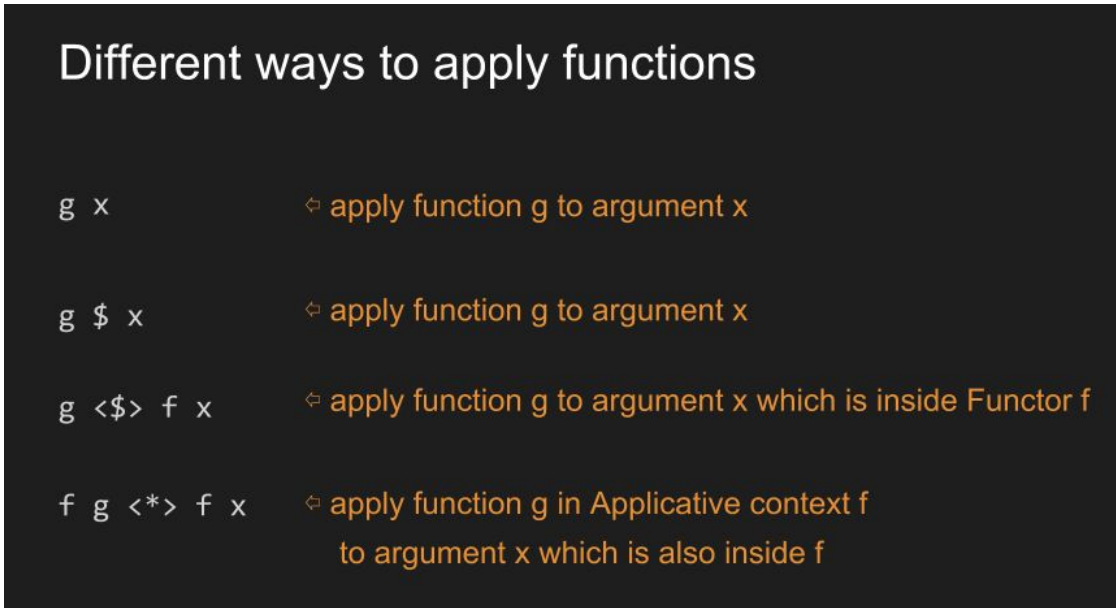
```
data Suit = Spade|Club|Diamond|Heart
 deriving (Eq,Ord,Enum,Bounded)

instance Show Suit where
 show Spade = "^"      -- ♠    (closest I could come in ASCII was ^)
 show Club = "&"       -- ♣
 show Diamond = "O"    -- ♦
 show Heart = "V"      -- ♥

data Rank =
Two|Three|Four|Five|Six|Seven|Eight|Nine|Ten|Jack|Queen|King|Ace
 deriving (Eq,Ord,Enum,Show,Bounded)

data Card = Card Suit Rank
 deriving (Eq, Ord, Show)
```

We can make one card using the Card constructor:

```
GHCi> Card Spade Ace
Card ^ Ace
```

Or, since both Suit and Rank derive Enum, we can enumerate the full lists of Suits and Ranks, and then lift the Card operator over both lists to create a whole deck:

```
GHCi> Card <$> [Spade ..] <*> [Two ..]
[Card ^ Two,Card ^ Three,Card ^ Four,Card ^ Five,Card ^ Six,Card ^
Seven,Card ^ Eight,Card ^ Nine,Card ^ Ten,Card ^ Jack,Card ^ Queen,Card ^
King,Card ^ Ace,Card & Two,Card & Three,Card & Four,Card & Five,Card &
Six,Card & Seven,Card & Eight,Card & Nine,Card & Ten,Card & Jack,Card &
Queen,Card & King,Card & Ace,Card O Two,Card O Three,Card O Four,Card O
Five,Card O Six,Card O Seven,Card O Eight,Card O Nine,Card O Ten,Card O
Jack,Card O Queen,Card O King,Card O Ace,Card V Two,Card V Three,Card V
Four,Card V Five,Card V Six,Card V Seven,Card V Eight,Card V Nine,Card V
Ten,Card V Jack,Card V Queen,Card V King,Card V Ace]
```



Applicative is a “subclass” of Functor, meaning that an instance of Applicative can be ‘fmap’ed, but Applicatives also declare (at least[8]) two additional functions, **`pure`** and **`(<*>)`** (pronounced ‘apply’ - but I like calling it “TIE Fighter”):

```
GHCi> :i Applicative
class Functor f => Applicative (f :: * -> *) where
  pure :: a -> f a
```

[8] pure and <*> are the minimal definition to instance Applicative but it also has liftA2, *> and <* defined in terms of pure and <*>.

```
  (<*>) :: f (a -> b) -> f a -> f b
...
```

As for Functor, many Base Haskell types are also Applicative, e.g. `[]`, `Maybe`, `IO` and `(->)`.

16. Foldable

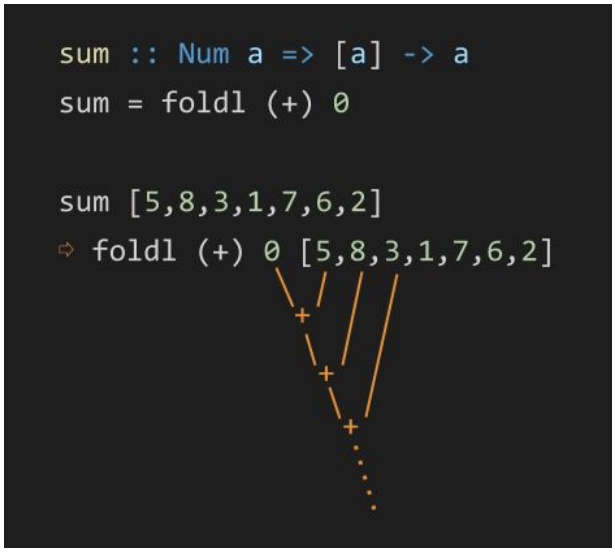
Recall the “reduce” function that is a member of JavaScript’s Array type, and which we implemented ourselves for linked and cons lists, was a way to generalise loops over enumerable types.
In Haskell, this is concept is once again generalised with a typeclass called Foldable - the class of things which can be “folded” over to produce a single value. The obvious Foldable instance is list. Although in JavaScript reduce always associates elements from left to right, haskell offers two functions foldl, which folds left to right) and foldr (which folds right to left):

```
> :t foldl
foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b

> :t foldr
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
```

Here’s how we right-fold over a list to sum its elements:

```
> foldr (\i a -> i + a) 0 [5,8,3,1,7,6,2]
32
```

Actually, this is a classic example where point-free coding style makes this expression very succinct: `foldr (+) 0 [5,8,3,1,7,6,2]`
...but the lambda above makes it clear which parameter is the accumulator and which is the list element. Here’s a left fold with a picture of the fold:



In the folds above, we provide the (+) function to tell foldl|r how to aggregate elements of the list. There is also a typeclass for things that are “automatically aggregatable” or “concatenatable” called Monoid which declares a general concatenation function for Monoid called mconcat. One instance of Monoid is Sum, since there is an instance of Sum for Num:

```
> :i Monoid
...
instance Num a => Monoid (Sum a)
...
> import Data.Monoid
Data.Monoid> mconcat $ Sum <$> [5,8,3,1,7,6,2]
Sum {getSum = 32}
```

So a sum is a data type with an accessor function getSum that we can use to get back the value:

```
Data.Monoid> getSum $ mconcat $ Sum <$> [5,8,3,1,7,6,2]
32
```

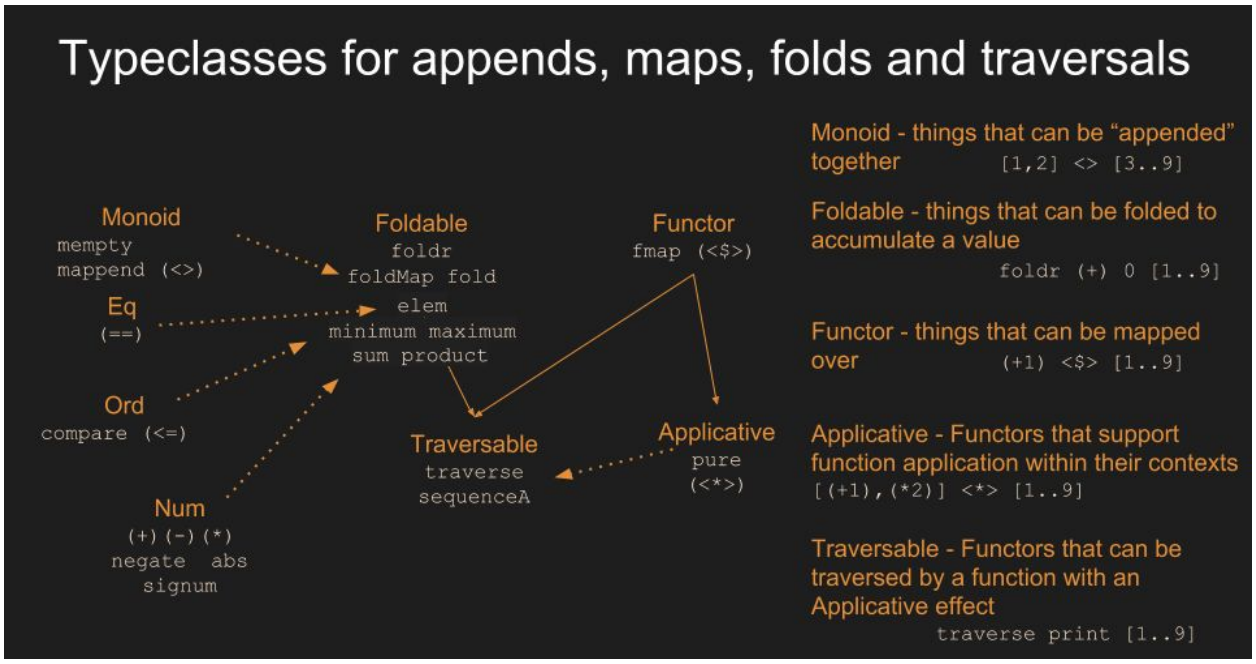
We make a data type aggregatable by instancing Monoid and providing definitions for the functions mappend and mempty. For Sum these will be (+) and 0 respectively.
Lists are also themselves Monoidal, with mappend defined as an alias for list concatenation (++), and mempty as []. Thus, we can:

```
Data.Monoid> mconcat [[1,2],[3,4],[5,6]]
[1,2,3,4,5,6]
```

Which has a simple alias “concat” defined in the Prelude:

```
> concat [[1,2],[3,4],[5,6]]
[1,2,3,4,5,6]
```

17. Traversable



Remember our safe modulo function:

```
safeMod :: Integral a => a-> a-> Maybe a
safeMod _ 0 = Nothing
safeMod numerator divisor = Just $ mod numerator divisor
```

Which we could use to map over a list of numbers without throwing divide-by-zero exceptions:

```
GHCi> map (safeMod 3) [1,2,0,2]
[Just 0,Just 1,Nothing,Just 1]
```

But what if 0s in the list really are indicative of disaster so that we should bail rather than proceeding? The traverse function of the Traversable type-class gives us this capability:

```
GHCi> traverse (safeMod 3) [1,2,0,2]
Nothing
```

Traverse applies a function with an Applicative return value (or Applicative *effect*) to the contents of a Traversable thing.

```
GHCi> :t traverse
traverse
  :: (Applicative f, Traversable t) => (a -> f b) -> t a -> f (t b)
```

What are some other functions with Applicative effects? Lots! E.g.:

- Any constructor of a data type that instances Applicative: e.g. `Just :: a -> Maybe a`
- Anything that creates a list: `(take 5 $ repeat 1) :: Num a => [a]`
- IO is Applicative, so a function like `print :: Show a => a -> IO ()`
- etc...

The print function converts values to strings (using show if available from an instance of Show) and sends them to standard-out. The print function wraps this effect (there is an effect on the state of the console) in an IO computational context:

```
GHCi> :t print
print :: Show a => a -> IO ()
```

The () is like void in TypeScript - it's a type with exactly one value (), and hence is called "Unit". There is no return value from print, only the IO effect, and hence the return type is IO (). IO is also an instance of Applicative. This means we can use traverse to print out the contents of a list:

```
GHCi> traverse print [1,2,3]
1
2
3
[(),(),()]
```

Where the return value is a list of the effects of each application of print, inside the IO Applicative.

```
GHCi> :t traverse print [1,2,3]
traverse print [1,2,3] :: IO [()]
```

There is no easy way to get rid of this IO return type - which protects you from creating IO effects unintentionally.

A related function defined in Traversable is sequenceA allows us to convert directly from Traversables of Applicatives, to Applicatives of Traversables:

```
> :t sequenceA
sequenceA :: (Applicative f, Traversable t) => t (f a) -> f (t a)
GHCi> sequenceA [Just 0,Just 1,Nothing,Just 1]
Nothing
```

Or on a "clean" list of Just values:

```
GHCi> sequenceA [Just 0,Just 1,Just 1]
Just [0,1,1]
```

18. Monad

As always, we can interrogate ghci to get a basic synopsis of the Monad typeclass:

```
> :i Monad
class Applicative m => Monad (m :: * -> *) where
  (>>=) :: m a -> (a -> m b) -> m b
  (>>) :: m a -> m b -> m b
  return :: a -> m a
...
  {-# MINIMAL (>>=) #-}
        -- Defined in `GHC.Base'
instance Monad (Either e) -- Defined in `Data.Either'
instance [safe] Monad m => Monad (ReaderT r m)
  -- Defined in `transformers-0.5.2.0:Control.Monad.Trans.Reader'
instance Monad [] -- Defined in `GHC.Base'
instance Monad Maybe -- Defined in `GHC.Base'
instance Monad IO -- Defined in `GHC.Base'
instance Monad ((->) r) -- Defined in `GHC.Base'
instance Monoid a => Monad ((,) a) -- Defined in `GHC.Base'
```

Things to notice:

- Monad is a subclass of Applicative (and therefore also a Functor)
- `return = pure`, from Applicative. `return` exists for historical reasons and you can safely use only `pure` (PureScript has only `pure`).
- the operator `(>>=)` (pronounced “bind”) is the minimal definition (the one function you must create--in addition to the functions also required for Functor and Applicative--to make a new Monad instance).
- `>>` is special case of bind (described below)
- lots of built-in types are already monads

There also exists a flipped version of bind:

```
(=<<) = flip (>>=)
```

Its type has a nice correspondence to the other operators we have already seen for function application in various contexts:

```
(=<<) :: Monad m       => (a -> m b) -> m a -> m b
(<*>) :: Applicative f => f (a -> b) -> f a -> f b
(<$>) :: Functor f     =>   (a -> b) -> f a -> f b
($)   ::                    (a -> b) -> a   -> b
```

IO

The haskell type which captures IO effects is called IO. As we demonstrated with the traverse function, it is possible to perform IO actions using fmap and applicative, (for example printing to the console), the challenge is taking values out of an IO context and using them to create further IO effects.

Here are some simple IO "actions":

```
sayHi :: IO ()
sayHi = putStrLn "Hi, what's your name?"
readName :: IO String
readName = getLine
greet :: String -> IO ()
greet name = putStrLn ("Nice to meet you, " ++ name ++ ".")
```

The following typechecks:

```
main = greet <$> getLine
```

When you run it from either GHCi or an executable compiled with ghc, it will pause and wait for input, but you will not see the subsequent greeting.
This is because the type of the expression is:

```
GHCi> :t greet <$> getLine
greet <$> getLine :: IO (IO ())
```

The IO action we want (greet) is nested inside another IO action. When it is run, only the outer IO action is actually executed. The inner IO computation (action) is not actually touched.
To see an output we somehow need to flatten the `IO (IO ())` into just a single level: `IO ()`.
(>>=) gives us this ability:

```
GHCi> :t getLine >>= greet
getLine >>= greet :: IO ()

GHCi> getLine >>= greet
Tim
Nice to meet you Tim!
```

The special case of bind (>>) allows us to chain actions without passing through a value:

```
GHCi> :t (>>)
(>>) :: Monad m => m a -> m b -> m b

GHCi> sayHi >> getLine >>= greet
Hi, what's your name?
Tim
Nice to meet you Tim!
```

Do notation

Haskell gives us syntactic sugar for bind in the form of “do blocks”:

```
main :: IO ()
main = do
   sayHi
   name <- readName
   greet name
```

Which is entirely equivalent to the above code, or more explicitly:

```
main =
   sayHi >>
   readName >>=
   \name -> greet name
```

Note that although <- looks like assignment to a variable named “name”, it actually expands to a parameter name for a lambda expression following the bind. Thus, the way I read the following do expression is “take the value (a string in this case) out of the Monad context resulting from the function (readName) and assign to the symbol (name)”:

```
do
  name <- readName
  greet name
```

You can also mix in actual variable assignments using let:

```
do
  name <- readName
  let greeting = "Hello " ++ name
  putStrLn greeting
```

Join

A function called “join” from Control.Monad also distills the essence of Monad nicely. Its type and definition in terms of bind is:

```
join :: Monad m => m (m a) -> m a
join = (>>=id)
```

We can apply join to “flatten” the nested IO contexts from the earlier fmap example:

```
GHCi>:t join $ greet <$> getLine :: IO ()
```

Which will now execute as expected:

```
GHCi> join $ greet <$> getLine
Tim
Nice to meet you Tim!
```