

FIT2102

Programming Paradigms

Lecture 3

Functional Programming

Pure functions

TypeScript

Faculty of Information Technology



MONASH
University

Learning Outcomes

- Create programs in JavaScript in a functional style
- Explain the role of pure functional programming style in managing side effects
- 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
- Create linked-lists and functions that operate over them, using:
 - Objects and classes
 - Only functions (Church encoded lists)

Recap: Closures and higher-order functions

A function and the set of variables it accesses from its enclosing scope is called a *closure*

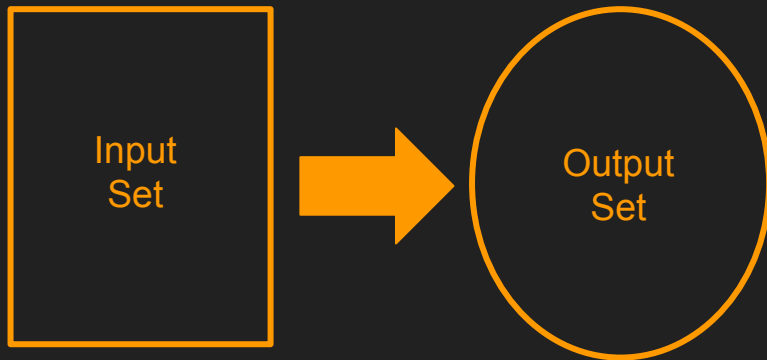
```
const multiply = x=> y=> x*y
const double = multiply(2)
[5,8,3,1,7,6,2].map(double)

> [ 10, 16, 6, 2, 14, 12, 4 ]
```

Functions that take other functions as parameters or that return functions are called *higher-order functions*.

Recap: Arrow Function Syntax

A function is a transform from one set of things to another:



```
function(x) {  
  return x / 3  
}  
  
x => x/3
```

This function is a transform from
number to number

In math we would be more precise:
e.g. the set of integers \mathbb{Z}
to the set of rationals \mathbb{Q}

Arrow function syntax - don't panic

```
const multiply = x=> y=> x*y
```

```
function multiply(x) {  
  return y => y * x;  
}
```

```
function multiply(x) {  
  return function(y) {  
    return x * y;  
  }  
}
```

Semantically all the same!
(unless they refer to `this`)

```
const operationOnTwoNumbers = f => x => y => f(x,y)
```

Semantically the same!

```
function operationOnTwoNumbers(f) {  
  return function(x) {  
    return function(y) {  
      return f(x,y);  
    }  
  }  
}
```


Currying

translating a function that takes multiple arguments into a sequence of unary functions.

Named after the mathematician Haskell Curry.

```
const operationOnTwoNumbers = f => x => y => f(x,y)
```

```
const multiply = operationOnTwoNumbers((x,y) => x*y)
```

```
const double = multiply(2)
```

Multiply is a
curried function

```
[5,8,3,1,7,6,2].map(double)
```


TypeScript Type Annotations

TypeScript is a superset of JavaScript that introduces *type annotations*

Any time you declare a variable, function, parameter, etc you can add a type annotation:

```
let x : number;
```

The compiler can often infer types from the context, i.e. *type inference*.

```
let x = 1;
```

Here `x` is obviously a `number` so an annotation is redundant.

When a type cannot be inferred, the type is `any`

Thus, TypeScript's default approach to type checking is *gradually typed*.

TypeScript Type Errors

```
let x = 1; // type number is inferred  
x = 'hello'
```

```
> error TS2322: Type '"hello"' is not assignable to type 'number'.
```

TypeScript Foo

```
const operationOnTwoNumbers =  
  (f: (x: number, y: number) => number) =>  
    (x: number) =>  
      (y: number) =>  
        f(x, y)
```

```
function operationOnTwoNumbers(f: (x: number, y: number) => number) {  
  return function(x: number) {  
    return function(y: number) {  
      return f(x, y);  
    }  
  }  
}
```

Type Aliases

```
type BinaryNumberFunc = (x:number, y:number) => number
```

```
type CurriedNumberFunc = (x:number) => (y:number) => number
```

```
const operationOnTwoNumbers: (f:BinaryNumberFunc)=>CurriedNumberFunc  
    = f=>x=>y=>f(x,y)
```

```
function operationOnTwoNumbers(f:BinaryNumberFunc):CurriedNumberFunc {  
    return function(x) {  
        return function(y) {  
            return f(x,y)  
        }  
    }  
}
```

Generic Types

```
type BinaryFunction<T> = (x:T, y:T) => T
```

```
type CurriedFunction<T> = (x:T) => (y:T) => T
```

```
const curry: <T>(f:BinaryFunction<T>)=>CurriedFunction<T>  
    = f=>x=>y=>f(x,y)
```

```
function curry<T>(f:BinaryFunction<T>):CurriedFunction<T> {  
    return function(x) {  
        return function(y) {  
            return f(x,y)  
        }  
    }  
}
```

Generic Types

```
type BinaryFunction<T> = (x:T, y:T) => T
```

```
type CurriedFunction<T> = (x:T) => (y:T) => T
```

```
const curry: <T>(f:BinaryFunction<T>)=>CurriedFunction<T>  
    = f=>x=>y=>f(x,y)
```

```
const twoToThe = curry(Math.pow)(2)
```

```
twoToThe(8)
```

```
> 256
```

⇨ This still type checks

```
const curriedConcat = curry(Array.prototype.concat.bind([]))
```

```
[[1,2],[1,2],[1,2]].map(curriedConcat([0]))
```

```
> [[0,1,2],[0,1,2],[0,1,2]]
```

⇨ But now
so does this!

Generic Types

```
type BinaryFunction<T,U,V> = (x:T, y:U) => V
```

```
type CurriedFunction<T,U,V> = (x:T) => (y:U) => V
```

```
const curry: <T,U,V>(f:BinaryFunction<T,U,V>)=>CurriedFunction<T,U,V>  
  = f=>x=>y=>f(x,y);
```

```
const twoToThe = curry(Math.pow)(2)
```

```
twoToThe(8)
```

```
> 256
```

⇐ This still type checks

```
const repeat = (n: number, s: string) => s.repeat(n)
```

```
console.log(['a','b','c'].map(curry(repeat)(3)))
```

```
> ["aaa", "bbb", "ccc"]
```

⇐ But now
so does this!

Lecture Activity 1

... to be announced

Linked Lists

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

```
const l:IListNode<number> = {  
  data: 1,  
  next: {  
    data: 2,  
    next: {  
      data: 3,  
      next: undefined  
    }  
  }  
}
```


```
function length<T>(l?:IListNode<T>): number {  
  return !l ? 0 : 1+length(l.next);  
}  
> length(l)  
3
```

```
function forEach<T>(f: (_:T)=>void, l?:IListNode<T>): void {  
  if (l) {  
    f(l.data);  
    forEach(f, l.next);  
  }  
}  
> forEach(console.log, l);  
1  
2  
3
```

Linked Lists - a type encoding list termination

```
interface IListNode<T> {  
  data: T;  
  next: ListPtr<T>;  
}  
type ListPtr<T> = IListNode<T> | undefined;
```

Union Type



```
const l:IListNode<number> = {  
  data: 1,  
  next: {  
    data: 2,  
    next: {  
      data: 3,  
      next: undefined  
    }  
  }  
}
```

```
function length<T>(l:ListPtr<T>): number {  
  return !l ? 0 : 1+length(l.next);  
}  
> length(l)  
3
```

```
function forEach<T>(f: (_:T)=>void, l:ListPtr<T>): void {  
  if (l) {  
    f(l.data);  
    forEach(f, l.next);  
  }  
}  
> forEach(console.log, l);  
1  
2  
3
```

Linked Lists - map

```
interface IListNode<T> {  
    data: T;  
    next: ListPtr<T>;  
}  
type ListPtr<T> = IListNode<T> | undefined;
```

```
function map<T,V>(f: (_:T)=>V, l: ListPtr<T>) : ListPtr<V>  
{  
    return l ? <IListNode<V>> {  
        data: f(l.data),  
        next: map(f, l.next)  
    } : undefined;  
}
```

```
> forEach(console.log, map(x=>2*x, l))  
2  
4  
6
```

Linked Lists - map with a class constructor

```
interface IListNode<T> {  
    data: T;  
    next: ListPtr<T>;  
}  
type ListPtr<T> = IListNode<T> | undefined;  
  
class ListNode<T> implements IListNode<T> {  
    constructor(  
        public data: T,  
        public next: ListPtr<T>) {}  
}
```

```
function map<T,V>(f: (_:T)=>V, l: ListPtr<T>) : ListPtr<V>  
{  
    return l ? new ListNode<V>(f(l.data), map(f, l.next))  
        : undefined;  
}  
  
> forEach(console.log, map(x=>2*x, l))  
2  
4  
6
```

Linked Lists - concat

```
interface IListNode<T> {  
    data: T;  
    next: ListPtr<T>;  
}  
type ListPtr<T> = IListNode<T> | undefined;  
  
class ListNode<T> implements IListNode<T> {  
    constructor(  
        public data: T,  
        public next: ListPtr<T>) {}  
}
```

? = Optional parameter



```
function concat<T>(a:ListPtr<T>, b?:ListPtr<T>): ListPtr<T> {  
    return a ? new ListNode(a.data, concat(a.next, b))  
        : (  
            b ? concat(b.next)  
            : undefined  
        )  
}  
  
> forEach(console.log, concat(1,map(x=>3+x, 1)))  
1  
2  
3  
4  
5  
6
```

Cons Lists (Church encoding)

Can we create lists with only lambda (anonymous) functions?

```
const cons = (head, rest) => f => f(head, rest)
```



The diagram illustrates the flow of the lambda function 'f'. A box around 'f' in the 'cons' definition has two arrows pointing to boxes around 'list' in the 'head' and 'rest' definitions, showing that 'f' is the function argument passed to both.

```
const aList = cons('Lists', cons("don't", cons("get", cons('any',  
    cons('simpler', cons('than', cons('this',  
        undefined)))))))
```

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

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

```
head(rest(rest(aList))) ⇒ "get"
```

Types for Cons Lists

Make the shapes of functions we expect on the previous slide explicit

```
/**
 * A ConsList is either a function created by cons, or empty (undefined)
 */
type ConsList<T> = Cons<T>|undefined;

/**
 * The return type of the cons function, is itself a function
 * which can be given a selector function to pull out either the head or rest
 */
type Cons<T> = (selector: Selector<T>) => T|ConsList<T>;

/**
 * a selector will return either the head or rest
 */
type Selector<T> = (head:T, rest:ConsList<T>)=> T|ConsList<T>;
```

Typed Cons List Functions

```
/**
 * cons "constructs" a list node, if no second argument is specified it is the last node in the list */
function cons<T>(head:T, rest: ConsList<T>): Cons<T> {
    return (selector: Selector<T>) => selector(head, rest);
}

/**
 * head returns the first element in the list
 * @param list is a Cons (note, not an empty ConsList) */
function head<T>(list:Cons<T>):T {
    return <T>list((head, rest) => head);
}

/**
 * rest everything but the head
 * @param list is a Cons (note, not an empty ConsList) */
function rest<T>(list:Cons<T>):ConsList<T> {
    return <Cons<T>>list((head, rest) => rest);
}
```


How do we get stuff out?

```
const aList = cons('Lists', cons("don't", cons("get",  
    cons('any', cons('simpler',  
    cons('than', cons('this'))))))))
```

```
function listToString(l:ConsList<string>): string {  
    if (!l) {  
        return ''  
    } else {  
        return head(l) + ' ' + listToString(rest(l))  
    }  
}
```

How do we get stuff out?

```
const aList = cons('Lists', cons("don't", cons("get",  
  cons('any', cons('simpler',  
    cons('than', cons('this'))))))))
```

```
const listToString: (l:ConsList<string>) => string  
  = l => l ? head(l) + ' ' + listToString(rest(l)) : ''
```

```
console.log(listToString(aList));
```

```
> Lists don't get any simpler than this!
```

Lecture Activity 2

... to be announced

Conclusions: Avoid hand-coded loops

```
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.

Conclusions: Function Purity

Every function that we've looked at today (except `console.log`) has been *pure*.

A pure function does not change the state of the surrounding program, or the world.

More than that, within the functions we did not reassign variables.

We say that these functions had *referential transparency*.

Referential transparency gives simple, reusable, safe functions.

Conclusions: Functions as a model of computation

The cons lists created only through functions are an example of “Church Encoding”

In the 1930s Alonzo Church created a model of computation which used only anonymous functions.

This model of computation became known as the “Lambda Calculus”

We will meet the lambda calculus properly in coming weeks.