Introduction

Here I list a couple of examples of the entire (expected) flow of my project. Including inputs, and outputs.

overview The examples are set out in the following order:

- library module (javascript restricts one file per module)
- library test-suite
- expected generated d.ts file

I am currently planning on generating the d.ts file in as simple a manner as possible. I flesh out a more complex example at the end, showing how functions can have different type signatures depending on how the test-suite accesses the library.

0.1 Basic Object

```
// foo.js
module.exports = {
  a: true,
  b: 1
}
```

```
// test-suite foo.js
var Coverage = require('../build/coverage.js')
var foo = Coverage.wrap(require('./foo'), 'foo')
foo.a === true
var x = foo.b + 3
```

```
// d.ts declaration
export var a: boolean
export var b: number
```

0.2 Function Object

```
// func.js
module.exports = {
  x: 2,
  f: function(a) { return a + 1; }
};
```

```
// test-suite func.js
var Coverage = require('../build/coverage.js')
var func = Coverage.wrap(require('./func.js'), 'func')
x = func.x
f = func.f
f(x)
```

```
// d.ts declaration
export var x: number
export function f(a: number): number
```

0.3 Nested Object

```
// obj.js
module.exports = {
  o: {
    a: 1,
    f: function(n) { return n + 'cm'; }
  },
  v: 'hello'
}
```

```
// test-suite obj.js
var Coverage = require('../build/coverage.js')
var obj = Coverage.wrap(require('./obj.js'), 'obj')

// console.log(obj)
console.log(obj.o.f(obj.v))
console.log(obj.o.f(obj.o.a))
```

```
// d.ts declaration

export var v: string
export var o: {
    a: number,
    f(a: string|number): string
}
```

0.4 Nested Object Goal

```
// obj_final.js
module.exports = {
  Length: {
    toCm: function(n) { return n + 'cm'; },
    toIn: function(n) { return n + 'Inches'; },
    Const: {
      one: 1,
      two: 2
    }
  },
  supported: ["cm", "inches"]
}
```

```
// test-suite obj_final.js
var obj = require('./examples/obj_final.js');

obj.Length.toCm(obj.Length.Const.one);
obj.Length.toCm('hello');


// note no toIn never applied to a string,
// so we don't know if it will work.
obj.Length.toIn(obj.Length.Const.one);


obj.supported[0];
```

```
// d.ts declaration
declare interface LengthConst {
    one: number
    two: number
}
declare interface Length {
    toCm(a: string|number): string
    toIn(a: number): string
    Const: LengthConst
}

export var Length: Length
export var supported: [string]
```

Reasoning I have structured the interfaces (which describe an object's properties) such that an interface that is accessible at *foo.bar* will have the

interface name $FooBar$. This means that there is a guarantee that no interface will clash. This also means that there may be duplications in interfaces. Currently I'm unsure whether or not this is a serious issue (or an issue at all). It should be possible to squash interfaces if they have the same internal structure.