My goal with CS450 is to teach you ...

1. Fundamental concepts behind most programming languages
   - functional programming, delayed evaluation, control flow and exceptions, object oriented systems, monads, macros, pattern matching, variable scoping, immutable data structures

2. A framework to describe language concepts
   - \(\lambda\)-calculus and formal systems to specify programming languages
   - functional programming and monads to implement specifications

3. A methodology to understand complex systems
   - (formally) specify and implement each programming language feature separately
   - understand a complex system as a combination of smaller simpler systems
   - implement and test features independently
Today we will...

- Revise JavaScript's object system
- Introduce SimpleJS: S-Expression-based syntax and simpler JavaScript rules
- Introduce LambdaJS: \(\lambda\)-calculus + references + immutable objects
- Introduce translation from SimpleJS into LambdaJS

Why are we learning all SimpleJS and LambdaJS?

- You already know \(\lambda\)-calculus with references (heap)
- You already know how objects work (ie, a map with a lookup that work like frames and environments)
- I want to teach you the fundamentals of JavaScript by building it on top of concepts that you already know!
- I can introduce another kind of specifying the semantics of a system, by translating it into another system (denotational semantics)
Object prototypes

A.__proto__ = B links A object to B, if a field f is not available in A, then it is looked up in B (which works recursively until finding undefined).

Functions are constructors

If we call a function A with new, then A is called as the constructor of a new object.

Constructor's prototype

If A is a function, then A.prototype becomes the __proto__ of every object created using A with new.
Quiz

What is the name of the paper we are studying?
SimpleJS
Introducing SimpleJS

- SimpleJS is just a simplification of JavaScript with fewer corner case, which is easier to learn.
- SimpleJS was created by your instructor for CS450 (yet close to what you have in The Essence of JavaScript)
- SimpleJS has a formal syntax (below) and also an S-expression syntax (hw8-util.rkt)
- Today we will **formally** describe SimpleJS in terms of how we can represent it in LambdaJS (defined in The Essence of JavaScript).

\[
    e ::= x \mid \text{let } x = e \text{ in } e \mid x.y \mid x.y := e \mid x.y(e\cdots) \\
    \mid \text{function}(x\cdots)e \mid \text{new } e(e\cdots) \\
    \mid \text{class extends } e \{\text{constructor}(x\cdots)e\} \; m\cdots \}
\]

\[
m ::= x(x\cdots)e
\]
Writing Shape in SimpleJS

JavaScript

```javascript
function Shape(x, y) {
    this.x = x;
    this.y = y;
}
let p = new Shape(10, 20);
Shape.prototype.translate =
    function(x, y) {
        this.x = this.x + x;
        this.y = this.y + y;
    };
p.translate(1,2);
return p;
```

SimpleJS

```simplejs
(let Shape
    (function (x y)
        (begin (set! this.x x)
            (set! this.y y)))
    (let p (new Shape 10 20)
        (let Shape-proto Shape.prototype
            (begin
                (set! Shape-proto.translate
                    (function (x y)
                        (begin
                            (set! this.x (! + this.x x))
                            (set! this.y (! + this.y y))))))
            (p.translate 1 2)
            p)))
```

Writing Shape in SimpleJS

CS450 ☽ Deconstructing JavaScript ☽ Lecture 25 ☽ Tiago Cogumbreiro
Writing Rectangle in SimpleJS

JavaScript

```javascript
function Rectangle(width, height) {
    this.x = 0;
    this.y = 0;
    this.width = width;
    this.height = height;
}
Rectangle.prototype = Shape.prototype;
let r1 = new Rectangle(10, 20);
return r1;
```

SimpleJS

```simplejs
(let Rectangle
 (function (width height)
 (begin
 (set! this.x 0)
 (set! this.y 0)
 (set! this.width width)
 (set! this.height height)))
(set! Rectangle.prototype Shape.prototype)
(let r1 (new Rectangle 10 20)
 r1))
```
Writing Rectangle in SimpleJS

JavaScript

```javascript
function Rectangle(width, height) {
    this.x = 0;
    this.y = 0;
    this.width = width;
    this.height = height;
}
Rectangle.prototype = Shape.prototype;
let r1 = new Rectangle(10, 20);
return r1;
```

SimpleJS

```simplejs
(let Rectangle
    (function (width height)
        (begin
            (set! this.x 0)
            (set! this.y 0)
            (set! this.width width)
            (set! this.height height)))
    (set! Rectangle.prototype Shape.prototype)
    (let r1 (new Rectangle 10 20)
        r1))
```

What are the possible problems of this form of inheritance?
Writing Rectangle in SimpleJS

What are the possible problems of this form of inheritance?

How can we add a new method to Rectangle?
Writing Rectangle in SimpleJS

With the highlighted pattern we can safely mutate `Rectangle.prototype`. This is the same as `Rectangle.prototype = {'__proto__': Shape.prototype }`, but we have no syntax for such a pattern in SimpleJS.

**JavaScript**

```javascript
function Rectangle(width, height) {
  this.x = 0;
  this.y = 0;
  this.width = width;
  this.height = height;
}
let p = function () {};
p.prototype = Shape.prototype;
Rectangle.prototype = new p();
let r1 = new Rectangle(10, 20);
return r1;
```

**SimpleJS**

```javascript
(let Rectangle
(function (width height)
 (begin (set! this.x 0) (set! this.y 0)
 (set! this.width width)
 (set! this.height height)))
(let p (function () 0)
 (begin
 (set! p.prototype = Shape.prototype)
 (set! Rectangle.prototype (new p))
 (let r1 (new Rectangle 10 20)
  r1)))
```
LambdaJS
LambdaJS

Think **Racket** without `define`, without macros, with **objects**, and **heap** operations.

Expressions

\[
e ::= v \mid x \mid \lambda x.e \mid e(e) \mid \{s: e\} \mid e[e] \mid e[e] \leftarrow e \mid \text{alloc } e \mid e := e
\]
Concrete LambdaJS S-expression syntax

<table>
<thead>
<tr>
<th>Formal syntax</th>
<th>S-expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \lambda x.e )</td>
<td>(lambda (x) e)</td>
</tr>
<tr>
<td>( e_1(e_2) )</td>
<td>( (e_1 \ e_2) )</td>
</tr>
<tr>
<td>{ &quot;foo&quot; : 1 + 2, &quot;bar&quot; : x }</td>
<td>(object [&quot;foo&quot; (+ 1 2)] [&quot;bar&quot; x])</td>
</tr>
<tr>
<td>o[&quot;foo&quot;]</td>
<td>(get-field o &quot;foo&quot;)</td>
</tr>
<tr>
<td>alloc {}</td>
<td>(alloc (object))</td>
</tr>
<tr>
<td>( x := {} )</td>
<td>(set! x (object))</td>
</tr>
<tr>
<td>( x := 1; x )</td>
<td>(begin (set! x 1) x)</td>
</tr>
<tr>
<td>let ( x = 10 ) in ( x + 4 )</td>
<td>(let ([x 10]) (+ x 4))</td>
</tr>
</tbody>
</table>

In Racket you can actually allocate a reference with `(box e)`, which is equivalent to LambdaJS\(\text{alloc} \ e\)\), and update the contents of that reference with `(set-box! b e)`, which is equivalent to LambdaJS \(\text{set!} \ e\).
Translating SimpleJS into LambdaJS
Translating SimpleJS into LambdaJS

1. A SimpleJS object is represented as a reference to an immutable LambdaJS object
2. A SimpleJS function is represented as an object with two fields: (a) a lambda-function that represents the code, a prototype field which points to an empty SimpleJS object
3. Create an object with new expects a SimpleJS function as argument and must create a new object, initialize its prototype, and call the constructor function (see point 2)
4. Method invocation corresponds to accessing a SimpleJS function and passing the implicit this object to it (see 2)

Objectives of the translation

- Explicit this
- Functions are not objects: convert function into an object+lambda
- Explicit memory manipulation
- No method calls: use function calls
Translating a function

JavaScript

```javascript
function Shape(x, y) {
    this.x = x;
    this.y = y;
};
```

Step 1: only objects and lambdas

```javascript
Shape = {
    '$code': (obj, x, y) => {
        obj.x = x;
        obj.y = y;
    },
    'prototype' = {}
};
```
Translating a function

JavaScript

```javascript
function Shape(x, y) {
    this.x = x;
    this.y = y;
}
```

Step 1: only objects and lambdas

```javascript
Shape = {
    '$code': (obj, x, y) => {
        obj.x = x;
        obj.y = y;
    },
    'prototype' = {};
};
```

Step 2: explicit references

```javascript
Shape = alloc {'$code': (this, x, y) => {
    this = (deref this)['x'] ← x; // In LambdaJS we have to replace the whole object
    this = (deref this)['y'] ← y;},
    'prototype' = alloc {}};
```
Translating new

JavaScript

```javascript
p1 = new Shape(0, 1);
```

Step 1: only objects and lambdas; no implicit this

```javascript
p1 = {
  "__proto__": Shape.prototype;
Shape["$code"](p1, 0, 1);
```

```javascript
Shape["$code"](p1, 0, 1);
```
Translating new JavaScript

Step 1: only objects and lambdas; no implicit this

```javascript
p1 = {"__proto__": Shape.prototype};
Shape["$code"](p1, 0, 1);
```

Step 2: explicit references

```javascript
p1 = alloc {"__proto__": (deref Shape)["prototype"]};
(deref Shape)["$code"](p1, 0, 1);
```
Translating method invocation

JavaScript

```javascript
p1.translate(10, 20);
```

Step 1: only objects and lambdas; no implicit this

```javascript
m = p1["translate"];  // get object method
m["$code"](p1, 10, 20);  // get code for method
```
Translating method invocation

JavaScript

```javascript
p1.translate(10, 20);
```

Step 1: only objects and lambdas; no implicit this

```javascript
m = p1["translate"]; // get object method
m["$code"](p1, 10, 20); // get code for method
```

Step 2: explicit references

Formally

```javascript
m = (deref p1)["translate"]; (deref m)["$code"](p1, 10, 20);
```

SimpleJS

```javascript
(let ([m (get-field (deref p1) "translate")]) ((get-field (deref m) "$code") p1 10 20))
```
Translating SimpleJS into LambdaJS

Before

```javascript
Shape.prototype.translate = function(x, y) {
  this.x += x; this.y += y;
};
p1 = new Shape(0, 1);
p1.translate(10, 20);
```

After

```javascript
// 1. Function declaration
Shape = alloc {
  "$code": (this, x, y) => { ... },
  "prototype" = alloc {};
};
p = (deref Shape)["prototype"];
(deref p)["translate"] = alloc {
  "$code": (this, x, y) => { ... }
  "prototype": alloc {};
// 2. new
p1 = alloc {
  "__proto__":
    (deref Shape)["prototype"];
  (deref Shape)["$code"](p1, 0, 1);
// 3. method call
f = (deref p1)["translate"];
(deref f)["$code"](p1, 10, 20);
```
Field lookup

\[ J[x.y] = (\text{deref } x)["y"] \]
To be continued...