CS450

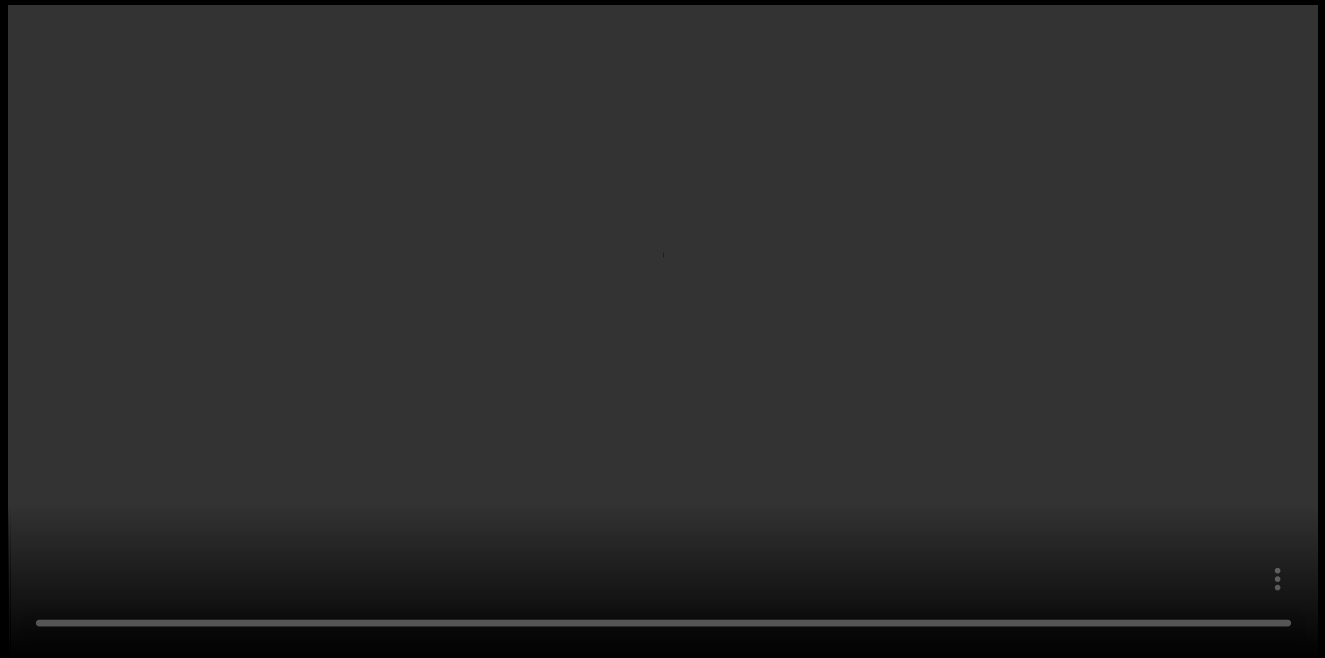
Structure of Higher Level Languages

Lecture 36: JavaScript intro; LambdaJS intro

Tiago Cogumbreiro

Today we will learn...

Hint: `(!+[]+[]+![]).length == 9`



Source: Gary Bernhardt, 2012. `www.destroyallsoftware.com/talks/wat`

Today we will learn...

- Introduce Object-Oriented Programming (OOP)
- Introduce "The Essence of Javascript"
- Implement JavaScript in Racket

Why JavaScript?

- JavaScript is everywhere
- JavaScript's object system is simple to describe
- JavaScript introduces a minimal yet powerful OOP model

Technically, ECMAScript

> However, the umbrella term "JavaScript" as understood in a web browser context contains several very different elements. One of them is the core language (ECMAScript), another is the collection of the Web APIs, including the DOM (Document Object Model).

Source: MDN

What are we learning?

UMASS BOSTON

The ***language mechanics*** of JavaScript

JavaScript **semantics**

1. What is an object?
2. How does variable binding work?
3. How does inheritance work?
4. How does mutation work?
5. How do functions interact with objects?

What are we **not** covering?

1. JavaScript API's (including the standard library)
2. `promises`/`async`/`await` (partially covered in Lecture 19)
3. Control flow (`for`, `while`, `if`, can be represented as λ-calculus)
4. JavaScript best practices
5. Differences between ECMAScript versions
6. A faithful implementation of ECMAScript (simplicity trumps fidelity)

How are we learning JavaScript?

1. We will study a research paper that introduces the core JavaScript semantics
2. We will implement these semantics in Racket

The Essence of JavaScript

Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi

Brown University

Abstract. We reduce JavaScript to a core calculus structured as a small-step operational semantics. We present several peculiarities of the language and show that our calculus models them. We explicate the desugaring process that turns JavaScript programs into ones in the core. We demonstrate faithfulness to JavaScript using real-world test suites. Finally, we illustrate utility by defining a security property, implementing it as a type system on the core, and extending it to the full language.

Essence of JavaScript. Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. ECOOP. 2010.

λ_{JS}: the essence of JavaScript

UMASS BOSTON

Paper summary

1. **Describes a core language:** λ_{JS} is a λ-calculus extended with shared memory (references), and with an object system
2. **Describes a translation function:** translates JavaScript into λ_{JS}

Our study

- We will learn how to implement the object system of JavaScript
- Our final project consists of implementing a translation function from SimpleJS into λ_{JS}

Paper URL: cs.brown.edu/research/plt/dl/jssem/v1/gsk-essence-javascript-r6.pdf

Some definitions on translators

- **Translator** (or translation function): the process of converting terms of one language into terms of another language (possibly the same language)
- **Compiler:** translate a source language into a target language; generally the target language is at the machine-level (e.g., assembly, bytecode, intermediate-representation)
- **Syntactic desugaring:** the process of converting syntactic abbreviations into more fundamental terms of the same language
- **Source-to-source translator:** unlike the compiler, a source-to-source compiler, the target language of a source-to-source translator is a higher-level language
- **Transpiler:** source-to-source translator. Term popularized by the JavaScript community.
- **Polyfill:** provides modern functionality on older browsers (eg, via syntactic desugaring).

Let us learn JavaScript and λ_{JS}

Following, I will:

1. introduce the basics of JavaScript
2. introduces the basics of λ_{JS}
3. relate the functionalities found in JavaScript to those of λ_{JS}
4. list our λ_{JS} AST written in Racket

Constants

Constants

$$c ::= n \mid s \mid b \mid \texttt{undef}$$

```
// Numbers
100
-100
// Strings
"fooo"
'fooo'
// Booleans
true
false
// Undefined
undefined
```

Expressions in JavaScript

Expressions

1. Objects
2. Functions

> In JavaScript functions are also objects! We will learn

Objects in JavaScript

Objects are mutable maps

UMASS BOSTON

1. **Object declaration:** `{"str1" : expr1, "str2": expr2, ... }`
2. **Field lookup:** `obj["field"]` or `obj.field`
3. **Field update:** `obj["field"] = expr` or `obj.field = expr`
4. **Field deletion:** `delete obj["field"]` or `delete obj.field`

```
var x = {} // Creates a new, empty object
// Reading an unset field returns undefined
console.assert(x["foo"] == undefined)
// Assignment works as expected
x["foo"] = 20
console.assert(x["foo"] == 20);
// Deleting a field is equivalent to assigning to undefined
delete x["foo"];  // x["foo"] = undefined ← in our semantics
console.assert(x["foo"] == undefined);
```

Using the dot notation

UMASS BOSTON

- In JavaScript we can use the dot notation.

```
var x = {} // Creates a new, empty object
// Reading an unset field returns undefined
console.assert(x.foo == undefined)
// Assignment works as expected
x.foo = 20
console.assert(x.foo == 20);
// Deleting a field is equivalent to assigning to undefined
delete x.foo;  // x.foo = undefined ← in our semantics
console.assert(x.foo == undefined);
```

JavaScript and creating objects

```
{} // The empty object
> {"a key": 100, "another key": true, "": null, "00": undefined }
{ 'a key': 100, 'another key': true, '': null, '00': undefined }
>{100: "numbers are converted to strings", true: "booleans are left as is",
  undefined: "as are undefined", null: "and also null"}
{ '100': 'numbers are converted to strings',
  true: 'booleans are left as is', undefined: 'as are undefined', null: 'and also null' }
> {"foo": 1, "foo": 2 + 100} // Only the left-most key sets its value
{ foo: 102 }  // Evaluation works on values (but not on not keys)
```

- In our implementation, we will assume the keys can **only** be strings

Functions in JavaScript

JavaScript and functions

Nameless functions

We can write lambda abstraction in JavaScript with the following syntax

```
(x ...) => e
```

where `x ...` is a comma-separated list of identifiers.

Example in JS

```
console.assert(
  ((x) =>
    (y) => x - y)(1)(2)
  ==
  -1);
```

Racket

```
(check-equals?
  (((lambda (x)
     (lambda (y) (- x y))) 1) 2)
  -1)
```

Named functions in JavaScript

Named functions in JavaScript have the following syntax.

```
function x(x...) { stmt ... }
```

JavaScript

```
function add(x, y) {
  return x + y;
}
console.assert(add(1, 2) == 3);
```

Racket

```
(define (add x y)
  (+ x y))
(check-equals? (add 1 2) 3)
```

Syntactic sugar

Version 1

```
p1 = {};
p1["x"] = 13
p1["y"] = 7;
console.assert(p1["x"] == 13);
console.assert(p1["y"] == 7);
```

Syntactic sugar

Version 1

```
p1 = {};
p1["x"] = 13
p1["y"] = 7;
console.assert(p1["x"] == 13);
console.assert(p1["y"] == 7);
```

Version 2

```
// Version 2: Syntactic sugar 1
p2 = {"x": 13, "y": 7};
console.assert(p1 != p2);
console.assert(p1["x"] == p2["x"]);
console.assert(p1["y"] == p2["y"]);
```

Syntactic sugar

Version 1

```
p1 = {};
p1["x"] = 13
p1["y"] = 7;
console.assert(p1["x"] == 13);
console.assert(p1["y"] == 7);
```

Version 2

```
// Version 2: Syntactic sugar 1
p2 = {"x": 13, "y": 7};
console.assert(p1 != p2);
console.assert(p1["x"] == p2["x"]);
console.assert(p1["y"] == p2["y"]);
```

Version 3

```
p3 = new Object();
p3.x = 13
p3.y = 7;
console.assert(p1.x == p3.x)
console.assert(p1.y == p3.y);
```

Syntactic sugar

Version 1

```
p1 = {};
p1["x"] = 13
p1["y"] = 7;
console.assert(p1["x"] == 13);
console.assert(p1["y"] == 7);
```

Version 2

```
// Version 2: Syntactic sugar 1
p2 = {"x": 13, "y": 7};
console.assert(p1 != p2);
console.assert(p1["x"] == p2["x"]);
console.assert(p1["y"] == p2["y"]);
```

Version 3

```
p3 = new Object();
p3.x = 13
p3.y = 7;
console.assert(p1.x == p3.x)
console.assert(p1.y == p3.y);
```

Summary

1. The usual dot-notation to get/set fields
2. == tests **references**; not structural
3. `new Object()` is a synonym for {}

   More on `new` in Lecture 24!

Expressions in λ_{JS}

Expressions in λ_{JS}

$$e ::= \{\} \mid e[e] \mid e[e] \leftarrow e \mid e(e) \mid \lambda x.e$$

- **Objects are immutable!**
- **Only anonymous functions.**
- $\{\}$ creates an "empty" object, that is, all fields are assigned to `undefined`
- The notation $e_o[e_f]$ reads the field e_f from object e_o
- The notation $e_o[e_f] \leftarrow e_v$ is an immutable extension of a map. Recall (`hash-add ht x v`).

Runtime-only values

Runtime-only values

$$v = (E, \lambda x.e) \mid O$$

- Closures are **run-time-only** values, so no way to show it in JavaScript
- Objects O are also **run-time-only** values and represents a map from strings to values (recall environments in λ_E)

λ_E-calculus

Revisiting Lesson 12

$$v \Downarrow_E v \qquad \texttt{(E-val)}$$

$$x \Downarrow_E E(x) \qquad \texttt{(E-var)}$$

$$\lambda x.e \Downarrow_E (E, \lambda x.e) \qquad \texttt{(E-clos)}$$

$$\frac{e_f \Downarrow_E (E_b, \lambda x.e_b) \qquad e_a \Downarrow_E v_a \qquad e_b \Downarrow_{\mathbf{E_b}[\mathbf{x} \mapsto \mathbf{v_a}]} v_b}{(e_f\ e_a) \Downarrow_E v_b} \qquad \texttt{(E-app)}$$

Rules for objects

$$\frac{\forall s.O(s) = \texttt{undef}}{\{\} \Downarrow_E O}\ \texttt{E-empty}$$

$$\frac{e_o \Downarrow_E O \qquad e_f \Downarrow_E s}{e_o[e_f] \Downarrow_E O(s)}\ \texttt{(E-get)}$$

$$\frac{e_o \Downarrow_E O \qquad e_f \Downarrow_E s \qquad e_v \Downarrow_E v}{e_o[e_f] \leftarrow e_v \Downarrow_E O[s \mapsto v]}\ \texttt{(E-set)}$$

Implementation

Syntax

$$c ::= n \mid s \mid b \mid \texttt{undef}$$

Syntax

$$c ::= n \mid s \mid b \mid \textbf{undef}$$

```
(define (k:const? v)
  (or (k:number? v)
      (k:string? v)
      (k:bool? v)
      (k:undef? v)))
(struct k:number (value) #:transparent)
(struct k:string (value) #:transparent)
(struct k:bool (value) #:transparent)
(struct k:undef () #:transparent)
```

Values

$$v = c \mid (E, \lambda x.e) \mid O$$

Values

$$v = c \mid (E, \lambda x.e) \mid O$$

```
(define (j:value? v)
  (or (k:const? v)
      (j:closure? v)
      (j:object? v)))
(struct j:closure (env decl) #:transparent)
(struct j:object (data) #:transparent)
```

Expressions

$$e ::= \lambda x.e \mid x \mid e(e) \mid e[e] \mid e[e] \leftarrow e$$

Expressions

$$e ::= \lambda x.e \mid x \mid e(e) \mid e[e] \mid e[e] \leftarrow e$$

```
(struct j:lambda (params body) #:transparent)
(struct j:variable (name) #:transparent)
(struct j:apply (func args) #:transparent)
;; Object-related operations
(struct j:get (obj field) #:transparent)
(struct j:set (obj field value) #:transparent)
```