Do we need...

- Function definition?
Do we need...

- **Function definition?** granted!

- Function application? granted!

- Functions with multiple parameters? no!

- Numbers? no!

- Conditionals? no!

- Recursive functions? no!
Do we need...

- **Function definition?** granted!
- **Function application?**
Do we need...

- **Function definition**? granted!
- **Function application**? granted!
Do we need...

- **Function definition?** granted!
- **Function application?** granted!
- **Functions with multiple parameters?**
A Theoretician’s Programming Language

Do we need...

- **Function definition?** granted!
- **Function application?** granted!
- **Functions with multiple parameters?** no!
Do we need...

- **Function definition?** granted!
- **Function application?** granted!
- **Functions with multiple parameters?** no!
- **Numbers?**
A Theoretician’s Programming Language

Do we need...

- **Function definition?** granted!
- **Function application?** granted!
- **Functions with multiple parameters?** no!
- **Numbers?** no!
A Theoretician’s Programming Language

Do we need...

- **Function definition?** granted!
- **Function application?** granted!
- **Functions with multiple parameters?** no!
- **Numbers?** no!
- **Conditionals?**
Do we need...

- Function definition? granted!
- Function application? granted!
- Functions with multiple parameters? no!
- Numbers? no!
- Conditionals? no!
Do we need...

- **Function definition?** granted!
- **Function application?** granted!
- **Functions with multiple parameters?** no!
- **Numbers?** no!
- **Conditionals?** no!
- **Recursive functions?**
Do we need...

- **Function definition?** granted!
- **Function application?** granted!
- **Functions with multiple parameters?** no!
- **Numbers?** no!
- **Conditionals?** no!
- **Recursive functions?** no!
Do we need...

- **Function definition?** granted!
- **Function application?** granted!
- **Functions with multiple parameters?** no!
- **Numbers?** no!
- **Conditionals?** no!
- **Recursive functions?** no!

Martin Henz

From the Lambda Calculus to JavaScript
function square(x) {
    return x * x;
}
square(13);
function plus(x,y) {
    return x + y;
}
plus(5,7);
Do we need multiple arguments?

function plus(x, y) {
    return x + y;
}

plus(5, 7);

becomes

function plus(x) {
    function plusx(y) {
        return x + y;
    }
    return plusx;
}

var plusfive = plus(5);

plusfive(7);
function power(x,y) {
    if (y === 0) return 1;
    return x * power(x,y-1);
}

power(2,4);
function power(x, y) {
    if (y === 0) return 1;
    return x * power(x, y - 1);
}

power(2, 4);

translates to:

function power(x) {
    return function(y) {
        if (y === 0) return 1;
        return x * power(x)(y - 1);
    };
}

power(2)(4);
Representing 0 using Church numerals:

```javascript
function zero(f) {
    return function(x) {
        return x;
    }
}

zero('something')('somethingelse')
```
Do we need numbers?

Representing 1 using Church numerals:

```javascript
function one(f) {
    return function(x) {
        return f(x);
    }
}

one(function(x) { return x*2; })(4)
```
Representing 2 using church numerals:

```javascript
function two(f) {
    return function(x) {
        return f(f(x));
    }
}
two(function(x) { return x*2; })(4)
```
function two(f) {
  return function(x) {
    return f(f(x));
  }
}

function church2js(c) {
  return c(function(x) { return x+1; })(0);
}

church2js(two);
Can you do the reverse?

Wanted:
Define a function \texttt{js2church} that takes a JediScript number as argument and returns its Church numeral?
function times(x) {
    return function(y) {
        return function(f) {
            return x(y(f));
        }
    }
}

function two(f) {
    return function(x) {
        return f(f(x));
    }
}
function three(f) {
    return function (x) {
        return f(f(f(x)));
    }
}

function church2js(c) {
    return c(function (x) { return x+1; })(0);
}

church2js(times(two)(three));
**Conditionals**

Conditional statements

```javascript
if (20 < 10) { return 5; } else { return 7; }
```

Conditional expressions

```javascript
(20 < 10) ? 5 : 7
```
Do we need conditionals?

Idea

Represent booleans with functions

The function “true”

```javascript
function True(x) {
    return function(y) {
        return x;
    }
}
```
Do we need conditionals?

Idea

Represent booleans with functions

The function “false”

```javascript
function False(x) {
    return function(y) {
        return y;
    }
}
```
Do we need conditionals?

Conditional in JediScript

True ? 5 : 7;

Conditional using Encoding

True(5)(7);
function factorial(x) {
    return (x === 0) ? 1
        : x * factorial(x - 1);
}
factorial(5);
function F(f) {
    return function(x) {
        return (x === 0) ? 1
        : x * f(x - 1);
    };  
}
We need a function $Y$ with the following properties:

$$Y(F) \equiv F(Y(F))$$
function (f) {
    return (function (x) {
        return f(function (y) {
            return x(x)(y);
        });
    })
    (function (x) {
        return f(function (y) {
            return x(x)(y);
        });
    });
}
As a sublanguage of JediScript, the Lambda Calculus looks like this:

\[ \text{L ::= } \langle \text{id} \rangle | (\text{L})(\text{L}); | \text{function}(\langle \text{id} \rangle) \{ \text{return L; } \} \]
So: Why don’t we program using the Lambda Calculus?

Answer

Other design goals are equally important!
So: Why don’t we program using the Lambda Calculus?

Answer
Other design goals are equally important!

Some design goals for full JavaScript
- Expressive
- Easy to learn
- Convenient to use
So: Why don’t we program using the Lambda Calculus?

**Answer**
Other design goals are equally important!

**Some design goals for full JavaScript**
- Expressive
- Easy to learn
- Convenient to use

**At the expense of...**
So: Why don’t we program using the Lambda Calculus?

Answer
Other design goals are equally important!

Some design goals for full JavaScript
- Expressive
- Easy to learn
- Convenient to use

At the expense of...
simplicity!
Lambda Calculus: Some History

- Introduced by Alonzo Church in 1930s as a minimal formal system for recursion theory
- Later found to be equivalent to other computing frameworks (Church-Turing thesis)
- Used extensively in programming language theory and theoretical computer science
Simplicity is an important and highly useful driving force behind science and engineering.
Simplicity is an important and highly useful driving force behind science and engineering.

Enables insights that would otherwise remain lost in a thicket of details.
Summary

- Simplicity is an important and highly useful driving force behind science and engineering.
- Enables insights that would otherwise remain lost in a thicket of details.
- In practice, simplicity competes with other goals; keep it in mind when thinking about complex systems.
Write a lambda expression $\text{EXP}$ such that

$$\text{lambda2jediscript}((\text{EXP})(\text{jediscript2lambda}(6)));$$

will result in the factorial of 6 when entered in the Web Console of Firefox.

You will need:

- conditionals using the shown encoding
- your implementation of $\text{lambda2jediscript}$ and $\text{jediscript2lambda}$
- the $Y$ combinator shown above
- addition, multiplication
- predecessor “$n - 1$” is the hardest