A Theoretician's Programming Language

Do we need...

- **Function definition?**

A Theoretician's Programming Language

Do we need...

- **Function definition?** granted!

A Theoretician's Programming Language

Do we need...

- **Function definition?** granted!
- **Function application?**

A Theoretician's Programming Language

Do we need...

- **Function definition?** granted!
- **Function application?** granted!

A Theoretician's Programming Language

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!

A Theoretician's Programming Language

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?**

A Theoretician's Programming Language

Do we need...

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

A Theoretician's Programming Language

Do we need...

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

A Theoretician's Programming Language

Do we need...

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

A Theoretician's Programming Language

Do we need...

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

Some Examples

```
function square(x) {
    return x * x;
}
square(13);
```

Do we need multiple arguments?

```
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);
```

Another Example

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

Do we need multiple arguments?

```
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);
```

Do we need numbers?

Representing 0 using Church numerals:

```
function zero(f) {
    return function(x) {
                return x;
            }
}
zero('something')('somethingelse')
```

Do we need numbers?

Representing 1 using Church numerals:

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

Do we need numbers?

Representing 2 using church numerals:

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

Getting the number back

```
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 `js2church` that takes a JediScript number as argument and returns its Church numeral?

Multiplication

```
function times(x) {
    return function(y) {
                return function(f) {
                            return x(y(f));
                        }
                }
}
function two(f) {
    return function(x) {
                return f(f(x));
            }
}
```

Multiplication

```
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

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

Conditional expressions

```
(20 < 10) ? 5 : 7
```

Do we need conditionals?

Idea

Represent booleans with functions

The function “true”

```
function True(x) {
    return function(y) {
                return x;
    }
}
```

Do we need conditionals?

Idea

Represent booleans with functions

The function “false”

```
function False(x) {
    return function(y) {
                return y;
    }
}
```

Do we need conditionals?

Conditional in JediScript

```
True ? 5 : 7;
```

Conditional using Encoding

```
True(5)(7);
```

Factorial using Conditional Expressions

```
function factorial(x) {
    return (x === 0) ? 1
                     : x * factorial(x - 1);
}
factorial(5);
```

Step 1: Eliminate Recursive Call

```
function F(f) {
    return function(x) {
                return (x === 0) ? 1
                                 : x * f(x - 1);
            };
}
```

Step 2: Find a Fix-Point Function (aka Y-Combinator)

We need a function Y with the following properties:

$$Y(F) \equiv F(Y(F))$$

Step 2: A Y-Combinator

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

The Pure (Untyped) Lambda Calculus

As a sublanguage of JediScript, the Lambda Calculus looks like this:

$$\mathbf{L} ::= \langle id \rangle \mid (\mathbf{L})(\mathbf{L}); \mid \texttt{function}(\langle id \rangle)\ \{\ \texttt{return}\ \mathbf{L};\ \}$$

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

Summary

- Simplicity is an important and highly useful driving force behind science and engineering

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

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

Homework

Write a lambda expression `EXP` such that

```
lambda2jediscript((EXP)(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 `lambda2jediscript` and `jediscript2lambda`
- the Y combinator shown above
- addition, multiplication
- predecessor “n - 1” is the hardest