First-Class Continuations: What and Why

Arjun Guha

University of Massachusetts Amherst

Plan

Today's topic: An introduction to *first-class continuations*

Tomorrow's topic: How to implement first-class continuations by source-to-source translation

Plan

Today's topic: An introduction to *first-class continuations*

Tomorrow's topic: How to implement first-class continuations by source-to-source translation

Today's meta-topic: The *expressiveness* of programming language features

Tomorrow's meta-topic: How to conduct research on a full-fledged programming language, without losing your sanity

We will use basic JavaScript for all code samples. Even if you don't know the language, it should be straightforward to follow. Please ask questions if you have trouble with the syntax.

Plan

Today's topic: An introduction to *first-class continuations*

Tomorrow's topic: How to implement first-class continuations by source-to-source translation

Today's meta-topic: The *expressiveness* of programming language features

Tomorrow's meta-topic: How to conduct research on a full-fledged programming language, without losing your sanity

We will use basic JavaScript for all code samples. Even if you don't know the language, it should be straightforward to follow. Please ask questions if you have trouble with the syntax.

Please interrupt and ask questions!

We are going to see a few code examples that are very challenging to understand.

Ask questions of the form, *"What if we change the example to do X instead?"* We will change the example and run it to see how it behaves.

Control Operators

Definition

A *control operator* is a programming language construct that changes the normal flow of execution in a program.

First-class continuations are a kind of control operator.

Some other control operators that are more familiar:

- *Conditionals*: **`if`** and **`switch`**
- *Loops*: **`for`** and **`while`**
- *Structured jumps*: **`break`** and **`continue`**
- *Exceptions*: **`try catch`** and **`try finally`**

Control Operators

Definition

A *control operator* is a programming language construct that changes the normal flow of execution in a program.

First-class continuations are a kind of control operator.

Some other control operators that are more familiar:

- *Conditionals*: **`if`** and **`switch`**
- *Loops*: **`for`** and **`while`**
- *Structured jumps*: **`break`** and **`continue`**
- *Exceptions*: **`try catch`** and **`try finally`**

Recent versions of JavaScript support some additional control operators:

- *Asynchronous functions:* **`async function`** and **`await`**
- *Generator functions:* **`function*`**, **`yield`**, and **`yield*`**

None of this is JavaScript specific: you can find all the control operators above in several other programming languages.

Why New Control Operators?

- The right control operator can make programs much easier to read and write. We will show how, using JavaScript's asynchronous functions and generator functions.
- First-class continuations make it possible to build several control operators as a library (i.e., without building them into the language).

Synchronous I/O

Definition

A *synchronous* or *blocking* I/O operation blocks execution while performing I/O and then resumes execution with the result of the I/O operation.

For example, a function with the following type would synchronously download an image:

```
loadImage(url: string) => Image
```

We could then display the image:

```
// drawImage(image: Image) => void
function drawImage(image) {
  document.body.appendChild(image);
}

let image = loadImage("example.jpg");
drawImage(image);
```

Synchronous I/O

Definition

A *synchronous* or *blocking* I/O operation blocks execution while performing I/O and then resumes execution with the result of the I/O operation.

For example, a function with the following type would synchronously download an image:

```
loadImage(url: string) => Image
```

We could then display the image:

```
// drawImage(image: Image) => void
function drawImage(image) {
  document.body.appendChild(image);
}

let image = loadImage("example.jpg");
drawImage(image);
```

In JavaScript, (almost) all I/O is asynchronous. It is impossible to write a synchronous `loadImage` function in JavaScript.

Asynchronous I/O

Definition

Asynchronous or *nonblocking* I/O performs the operation in the background and applies *callback function* when the I/O result is available.

We can implement an asynchronous function to load images:

```
loadImage(url: string, callback: Image => void) => void
```

Asynchronous I/O

Definition

Asynchronous or *nonblocking* I/O performs the operation in the background and applies *callback function* when the I/O result is available.

We can implement an asynchronous function to load images:

```
loadImage(url: string, callback: Image => void) => void
```

So, to load two images in sequence (link):

```
function callback2(image2) {
  drawImage(image);
}

function callback1(image1) {
  drawImage(image);
  loadImage("example2.jpg", callback2);
}

loadImage("example1.jpg", callback1);
```

Asynchronous I/O

Definition

Asynchronous or *nonblocking* I/O performs the operation in the background and applies *callback function* when the I/O result is available.

We can implement an asynchronous function to load images:

```
loadImage(url: string, callback: Image => void) => void
```

So, to load two images in sequence (link):

```
function callback2(image2) {
  drawImage(image);
}

function callback1(image1) {
  drawImage(image);
  loadImage("example2.jpg", callback2);
}

loadImage("example1.jpg", callback1);
```

We need a new callback function to wait for the result of each new asynchronous operation.

Async Functions

Definition

An *async function* can start an asynchronous task, suspend its own execution, and then resume with the result of the task when the task completes.

(link)

```
function loadImage(url) {
  // implementation omitted. Turns callbacks into Promises.
}

async function F() {
  let image1 = await loadImage("example1.jpg");
  drawImage(image1);
  let image2 = await loadImage("example2.jpg");
  drawImage(image2);
}
```

- Within an async function, we use the **`await`** keyword to call another async function.
- We can read the program from top to bottom (contrast to the callback-based approach).

Generator Functions

Definition

A *generator function* is a special kind of function that suspends its execution when it produces a value for the caller. The caller may then resume the generator function to make it produce the nex value (if any).

Ordinary functions *do not* work this way: they run to completion and cannot be suspended.

```
function* makeThreeGen() {
  yield 1;
  yield 2;
  yield 3;
}
```

```
let gen = makeThreeGen();
console.log(gen.next().value); // displays 1
console.log(gen.next().value); // displays 2
```

Unbounded Generators

The following example helps illustrate that the **yield** statement truly suspends execution of the generator function:

```
function* genNats() {
  let i = 0;
  while (true) {
    yield i;
    i = i + 1;
  }
}

let gen = genNats();

// Displays 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
for (let i = 0; i < 10; i++) {
  console.log(gen.next().value);
}
```

If **yield** did not suspend the generator function, the loop in the generator would run forever.

Some More Control Operators

There are several other kinds of control operators that JavaScript does not have:

- *Sampling executions from probability distributions*: WebPPL implements a compiler from a small subset of JavaScript with extensions to support probabilistic programming.
- *Backtracking search:* Tau Prolog implements a Prolog interpreter in JavaScript.
- *(Green) Threads:* Concurrent JavaScript

Some More Control Operators

There are several other kinds of control operators that JavaScript does not have:

- *Sampling executions from probability distributions*: WebPPL implements a compiler from a small subset of JavaScript with extensions to support probabilistic programming.
- *Backtracking search:* Tau Prolog implements a Prolog interpreter in JavaScript.
- *(Green) Threads:* Concurrent JavaScript

Question

Should JavaScript implement these natively too?

Some More Control Operators

There are several other kinds of control operators that JavaScript does not have:

- *Sampling executions from probability distributions*: WebPPL implements a compiler from a small subset of JavaScript with extensions to support probabilistic programming.
- *Backtracking search:* Tau Prolog implements a Prolog interpreter in JavaScript.
- *(Green) Threads:* Concurrent JavaScript

Question

Should JavaScript implement these natively too?

Using first-class continuations, we can implement each control operator above as a small library. In essence, first-class continuations subsume a wide variety of other control operators and language features.

Semantics, Informally

We need to understand three concepts before we get to first-class continuations:

1. Environments
2. Active Expressions
3. Continuations

This will be an quick, informal introduction via examples. For a rigorous, formal approach, see Semantis Engineering with PLT Redex.

Environments

Definition

The *environment* of a program maps variable names to values.

```
let x = 100;
let y = 200;
console.log(x + y);
```

At the last line, the environment holds the values of x and y.

```
function F(x) {
  console.log(x);
}

F(100);
F(200);
```

Each application evaluates the body of F in a different environment.

Active Expressions

Definition

The *active expression* in a program is the smallest part of the program that the language will evaluate next.

Examples:

```
console.log(2 + 3 * 4);
```

Active Expressions

Definition

The *active expression* in a program is the smallest part of the program that the language will evaluate next.

Examples:

```
console.log(2 + 3 * 4);
```

```
console.log(2 + 12);
```

Active Expressions

Definition

The *active expression* in a program is the smallest part of the program that the language will evaluate next.

Examples:

```
console.log(2 + 3 * 4);
```

```
console.log(2 + 12);
```

```
console.log(14);
```

Active Expressions

Definition

The *active expression* in a program is the smallest part of the program that the language will evaluate next.

Examples:

```
console.log(2 + 3 * 4);
```

```
console.log(2 + 12);
```

```
console.log(14);
```

```
if (5 > 10) {
  console.log("oops");
}
else {
  console.log("hi");
}
```

Two things to note:

- A statement or the whole program can be the active expression.
- Values cannot be active expressions.

Active Expressions

Definition

The *active expression* in a program is the smallest part of the program that the language will evaluate next.

Examples:

```
console.log(2 + 3 * 4);
```

```
console.log(2 + 12);
```

```
console.log(14);
```

Two things to note:

- A statement or the whole program can be the active expression.
- Values cannot be active expressions.

```
if (5 > 10) {
  console.log("oops");
}
else {
  console.log("hi");
}
```

```
if (false) {
  console.log("oops");
}
else {
  console.log("hi");
}
```

Active Expressions

Definition

The *active expression* in a program is the smallest part of the program that the language will evaluate next.

Examples:

```
console.log(2 + 3 * 4);
```

```
console.log(2 + 12);
```

```
console.log(14);
```

Two things to note:

- A statement or the whole program can be the active expression.
- Values cannot be active expressions.

```
if (5 > 10) {
  console.log("oops");
}
else {
  console.log("hi");
}
```

```
if (false) {
  console.log("oops");
}
else {
  console.log("hi");
}
```

```
console.log("hi");
```

Continuations

Definition

The *continuation* of an active expression is the rest of the program (i.e., excluding the active expression).

Therefore, the continuation is an expression with a single "hole", where the active expression used to be. We will write that hole as $\square$.

For example:

Program	Active Expression	Continuation
`console.log(2 + 3 * 4)`	`3 * 4`	

Continuations

Definition

The *continuation* of an active expression is the rest of the program (i.e., excluding the active expression).

Therefore, the continuation is an expression with a single "hole", where the active expression used to be. We will write that hole as $\square$.

For example:

Program	Active Expression	Continuation
`console.log(2 + 3 * 4)`	`3 * 4`	`console.log(2 + □)`
`console.log(2 + 12)`	`2 + 12`	

Continuations

Definition

The *continuation* of an active expression is the rest of the program (i.e., excluding the active expression).

Therefore, the continuation is an expression with a single "hole", where the active expression used to be. We will write that hole as $\square$.

For example:

Program	Active Expression	Continuation
`console.log(2 + 3 * 4)`	`3 * 4`	`console.log(2 + □)`
`console.log(2 + 12)`	`2 + 12`	`console.log(□)`
`console.log(14)`	`console.log(14)`	

Continuations

Definition

The *continuation* of an active expression is the rest of the program (i.e., excluding the active expression).

Therefore, the continuation is an expression with a single "hole", where the active expression used to be. We will write that hole as □.

For example:

Program	Active Expression	Continuation
`console.log(2 + 3 * 4)`	`3 * 4`	`console.log(2 + □)`
`console.log(2 + 12)`	`2 + 12`	`console.log(□)`
`console.log(14)`	`console.log(14)`	□

Continuations

Definition

The *continuation* of an active expression is the rest of the program (i.e., excluding the active expression).

Therefore, the continuation is an expression with a single "hole", where the active expression used to be. We will write that hole as $\square$.

For example:

Program	Active Expression	Continuation
`console.log(2 + 3 * 4)`	`3 * 4`	`console.log(2 + □)`
`console.log(2 + 12)`	`2 + 12`	`console.log(□)`
`console.log(14)`	`console.log(14)`	□

Note

All programming languages have continuations. They are fundamental for evaluation.

To run a program, we:

1. Identify the active expression and the continuation,
2. Evaluate the active expression,
3. Plug the result of evaluation into the hole that the active expression,
4. Repeat, until the program does not have an active expression.

Another Example of Continuations

	Program	`if (10 > 5) { log("o" + "k"); } else { log("oops"); }`
Step 1		

Another Example of Continuations

Step 1	Program	`if (10 > 5) { log("o" + "k"); } else { log("oops"); }`
	Active Expression	`10 > 5`
	Continuation	`console.log(2 + □)`
Step 2	Program	`if (true) { log("o" + "k"); } else { log("oops"); }`

Another Example of Continuations

Step 1	Program	`if (10 > 5) { log("o" + "k"); } else { log("oops"); }`
	Active Expression	`10 > 5`
	Continuation	`console.log(2 + □)`
Step 2	Program	`if (true) { log("o" + "k"); } else { log("oops"); }`
	Active Expression	`if (true) { log("o" + "k"); } else { log("oops"); }`
	Continuation	□
Step 3	Program	`log("o" + "k");`

Another Example of Continuations

Step 1	Program	`if (10 > 5) { log("o" + "k"); } else { log("oops"); }`
	Active Expression	`10 > 5`
	Continuation	`console.log(2 + □)`
Step 2	Program	`if (true) { log("o" + "k"); } else { log("oops"); }`
	Active Expression	`if (true) { log("o" + "k"); } else { log("oops"); }`
	Continuation	□
Step 3	Program	`log("o" + "k");`
	Active Expression	`"o" + "k"`
	Continuation	`log(□);`
Step 4	Program	`log("ok");`

Another Example of Continuations

Step 1	Program	`if (10 > 5) { log("o" + "k"); } else { log("oops"); }`
	Active Expression	`10 > 5`
	Continuation	`console.log(2 + □)`
Step 2	Program	`if (true) { log("o" + "k"); } else { log("oops"); }`
	Active Expression	`if (true) { log("o" + "k"); } else { log("oops"); }`
	Continuation	`□`
Step 3	Program	`log("o" + "k");`
	Active Expression	`"o" + "k"`
	Continuation	`log(□);`
Step 4	Program	`log("ok");`
	Active Expression	`log("ok");`
	Continuation	`□`

First-Class Continuations

Definition

In programming languages that support *first-class continuations*, it is possible to turn a continuation into a value.

A continuation value stored in a variable:

```
let k = (□ + 100);
```

First-Class Continuations

Definition

In programming languages that support *first-class continuations*, it is possible to turn a continuation into a value.

A continuation value stored in a variable:

```
let k = (□ + 100);
```

An array of continuation values:

```
let arr = [□ + 100, □ + 200];
```

A continuation passed as an argument to a function:

```
f(□ + 100);
```

First-Class Continuations

Definition

In programming languages that support *first-class continuations*, it is possible to turn a continuation into a value.

A continuation value stored in a variable:

```
let k = (□ + 100);
```

An array of continuation values:

```
let arr = [□ + 100, □ + 200];
```

A continuation passed as an argument to a function:

```
f(□ + 100);
```

Note

The □ notation is pseudocode. It is not possible to directly write a continuation in a program, but it helps understand what is happening under the hood. We will resolve this in a moment.

Applying a Continuation Value

A continuation value can be applied to an argument. (Applying a continuation value looks like applying a unary function.) When a continuation `k` is applied to a value `v`:

1. We discard the current continuation, and
2. The continuation `k` is restored and the hole gets filled with `v`.

Step 1	Program	**let** k = (□ + 1); log(100 + k(10))
	Environment	
	Active Expression	**let** k = (□ + 1)
	Continuation	□; log(100 + k(10))
Step 2	Program	log(100 + k(10));
	Environment	k = (□ + 1)
	Active Expression	k(10)
	Continuation	log(100 + □);
Step 3		

Applying a Continuation Value

A continuation value can be applied to an argument. (Applying a continuation value looks like applying a unary function.) When a continuation `k` is applied to a value `v`:

1. We discard the current continuation, and
2. The continuation `k` is restored and the hole gets filled with `v`.

Step 1	Program	**let** `k = (□ + 1); log(100 + k(10))`
	Environment	
	Active Expression	**let** `k = (□ + 1)`
	Continuation	`□; log(100 + k(10))`
Step 2	Program	`log(100 + k(10));`
	Environment	`k = (□ + 1)`
	Active Expression	`k(10)`
	Continuation	`log(100 + □);`
Step 3	Program	`10 + 1;` *This is `k` with `10` plugged in.*
	Environment	`k = (□ + 1)`
	Active Expression	`10 + 1`
	Continuation	□ *Continuation from Step 2 has been discarded*
Step 4	Result	`11;`
	Environment	`k = (□ + 1)`

Note on `call/cc`

Note

If you have see continuations before, you may know about `call/cc`. We are going to introduce an operator that is similar, but not identical to `call/cc`.

Capturing Continuations

Languages with first-class continuations provide a primitive operator that can turn its own continuation into a value ("capturing the current continuation").

We are going to introduce a new unary operator called **`control`**, which takes a unary function as an argument. When applied to a function `f`, **`control`**`(f)`:

1. Turns its own continuation into a continuation value `k`, and
2. Applies the function to the continuation value `f(k)` in the empty continuation.

The following example is fairly simple, since `F` does not use `k`:

Step 1	Program	`function F(k) { log(200); }; log(100 + control(F));`
	Environment	
	Active Expression	`function F(k) { log(200); }`
	Continuation	`□; log(100 + control(F));`
Step 2	Program	`log(100 + control(F));`
	Environment	`function F(k) { log(200); };`
	Active Expression	`control(F)`
	Continuation	`log(100 + □);`
Step 3		

Capturing Continuations

Languages with first-class continuations provide a primitive operator that can turn its own continuation into a value ("capturing the current continuation").

We are going to introduce a new unary operator called **`control`**, which takes a unary function as an argument. When applied to a function `f`, **`control`**`(f)`:

1. Turns its own continuation into a continuation value `k`, and
2. Applies the function to the continuation value `f(k)` in the empty continuation.

The following example is fairly simple, since `F` does not use `k`:

Step 1	Program	`function F(k) { log(200); }; log(100 + control(F));`
	Environment	
	Active Expression	`function F(k) { log(200); }`
	Continuation	`□; log(100 + control(F));`
Step 2	Program	`log(100 + control(F));`
	Environment	`function F(k) { log(200); };`
	Active Expression	`control(F)`
	Continuation	`log(100 + □);`
Step 3	Program	`log(200);`
	Environment	`let k = log(100 + □); function F(k) { log(200); };`
	Active Expression	`log(200);`
	Continuation	`□`

Applying a Captured Continuation

In the following example, the **throw** statement is never reached:

```
function F(k) {
  k(200);
  throw "bad";
}
log(100 + control(F));
```

Step 1	Program	`function F(k) { k(200); throw "bad"; }; log(100 + control(F));`
	Environment	
	Active Expression	`function F(k) { k(200); throw "bad"; };`
	Continuation	`□; log(100 + control(F));`
Step 2	Program	`log(100 + control(F));`
	Environment	`function F(k) { k(200); throw "bad"; };`
	Active Expression	`control(F)`
	Continuation	`log(100 + □);`

Step 3

Applying a Captured Continuation

In the following example, the **throw** statement is never reached:

```
function F(k) {
  k(200);
  throw "bad";
}
log(100 + control(F));
```

Step 1	Program	`function F(k) { k(200); throw "bad"; }; log(100 + control(F));`
	Environment	
	Active Expression	`function F(k) { k(200); throw "bad"; };`
	Continuation	`□; log(100 + control(F));`
Step 2	Program	`log(100 + control(F));`
	Environment	`function F(k) { k(200); throw "bad"; };`
	Active Expression	`control(F)`
	Continuation	`log(100 + □);`
Step 3	Program	`k(200); throw "bad"`
	Environment	`let k = log(100 + □);`
	Active Expression	`k(200)`
	Continuation	`□; throw "bad";`
Step 4		

Applying a Captured Continuation

In the following example, the **throw** statement is never reached:

```
function F(k) {
  k(200);
  throw "bad";
}
log(100 + control(F));
```

Step 1	Program	`function F(k) { k(200); throw "bad"; }; log(100 + control(F));`
	Environment	
	Active Expression	`function F(k) { k(200); throw "bad"; };`
	Continuation	`□; log(100 + control(F));`
Step 2	Program	`log(100 + control(F));`
	Environment	`function F(k) { k(200); throw "bad"; };`
	Active Expression	`control(F)`
	Continuation	`log(100 + □);`
Step 3	Program	`k(200); throw "bad"`
	Environment	`let k = log(100 + □);`
	Active Expression	`k(200)`
	Continuation	`□; throw "bad";`
Step 4	Program	`log(100 + 200);`
	Environment	
	Active Expression	`100 + 200`
	Continuation	`log(□);`

...

Applying Captured Continuations (Example 2)

Trace the execution of this program:

```
let i = 0;
let saved = "nothing";
function handler(k) {
  saved = k;
  saved("Start");
}

console.log(control(handler));
if (i < 3) {
  i = i + 1;
  saved(i);
}
```

Building New Control Operators with First-Class Continuations

- Code that uses first-class continuations directly is usually very hard to read.
- But, we can use first-class continuations to build other control operators that are more straightforward.

A Countdown Program

Programming Challenge

Write a program that counts down seconds, displaying: "Three", "Two", "One", "Liftoff!".

A Countdown Program

Programming Challenge

Write a program that counts down seconds, displaying: “Three”, “Two”, “One”, “Liftoff!”.

Solution:

```
function callback3() {
  console.log("Liftoff!");
}

function callback2() {
  console.log("One");
  setTimeout(callback3, 1000);
}

function callback1() {
  console.log("Two");
  setTimeout(callback2, 1000);
}

console.log("Three");
setTimeout(callback1, 1000);
```

A Countdown Program

Programming Challenge

Write a program that counts down seconds, displaying: “Three”, “Two”, “One”, “Liftoff!”.

Solution:

```
function callback3() {
  console.log("Liftoff!");
}

function callback2() {
  console.log("One");
  setTimeout(callback3, 1000);
}

function callback1() {
  console.log("Two");
  setTimeout(callback2, 1000);
}

console.log("Three");
setTimeout(callback1, 1000);
```

WTF

We cannot write a blocking sleep function in JavaScript (without busy-waiting).

Application: The Sleep Operator

Using first-class continuations, we can simulate a synchronous `sleep` function:

```
function sleep(n) {
  function sleeper(k) {
    setTimeout(k, n);
  }
  control(sleeper);
}
```

Application: The Sleep Operator

Using first-class continuations, we can simulate a synchronous `sleep` function:

```
function sleep(n) {
  function sleeper(k) {
    setTimeout(k, n);
  }
  control(sleeper);
}
```

The countdown program, refactored:

```
console.log("Three");
sleep(1000);
console.log("Two");
sleep(1000);
console.log("One");
sleep(1000);
console.log("Liftoff!");
```

Application: The Sleep Operator

Using first-class continuations, we can simulate a synchronous `sleep` function:

```
function sleep(n) {
  function sleeper(k) {
    setTimeout(k, n);
  }
  control(sleeper);
}
```

The countdown program, refactored:

```
console.log("Three");
sleep(1000);
console.log("Two");
sleep(1000);
console.log("One");
sleep(1000);
console.log("Liftoff!");
```

We can take any asynchronous function and simulate synchronous execution following this recipe. This is effectively what **`async function`** and **`await`** do.

Application: The Sleep Operator

JavaScript does not support shared-memory concurrency (for now). Using first-class continuations, we can simulate a synchronous `sleep` function:

```
function sleep(n) {
  function sleeper(k) {
    setTimeout(k, n);
  }
  control(sleeper);
}
```

The countdown program, refactored:

```
console.log("Three");
sleep(1000);
console.log("Two");
sleep(1000);
console.log("One");
sleep(1000);
console.log("Liftoff!");
```

We can take any asynchronous function and simulate synchronous execution following this recipe. This is effectively what **`async function`** and **`await`** do.

Application: Cooperative Threads

Definition

Cooperative threads run several logical threads on a single physical thread, thus there is no parallelism and only one thread is running at a time (other threads are suspended in background). Moreover, the running thread has to explicitly yield control of the physical thread for another thread to start running.

Application: Cooperative Threads

Definition

Cooperative threads run several logical threads on a single physical thread, thus there is no parallelism and only one thread is running at a time (other threads are suspended in background). Moreover, the running thread has to explicitly yield control of the physical thread for another thread to start running.

Key ideas in the implementation:

1. A global array of continuation values, where each continuation value is a suspended thread.
2. To yield, we capture the continuation of the active thread, add it to the array, and apply one of the other continuations.

Application: Cooperative Threads

Definition

Cooperative threads run several logical threads on a single physical thread, thus there is no parallelism and only one thread is running at a time (other threads are suspended in background). Moreover, the running thread has to explicitly yield control of the physical thread for another thread to start running.

Key ideas in the implementation:

1. A global array of continuation values, where each continuation value is a suspended thread.
2. To yield, we capture the continuation of the active thread, add it to the array, and apply one of the other continuations.

```
let threads = [ ];

function createThread(f) {
  function threadFunc() {
    f();
    start();
  }
  threads.push(threadFunc);
  yieldThread();
}
```

```
function start() {
  if (threads.length > 0) {
    let nextThread = threads.shift();
    nextThread();
  }
}

function yieldThread() {
  function switcher(k) {
    threads.push(k);
    let kOther = threads.shift();
    kOther("resumed");
  };

  control(switcher);
}
```

Using Cooperative Threads

```
function threadA() {
  for (let i = 0; i < 10; i++) {
      yieldThread();
      console.log(i);
  }
})


function threadB() {
  for (let i = 100; i < 110; i++) {
      yieldThread();
      console.log(i);
  };
})

createThread(threadA);
createThread(threadB);
start(); // needed to activate other threads
```

Conclusion

- I apologize if your head hurts
- Control operators can make certain kinds of programs much easier to write (e.g., generator functions and async functions)
- First-class continuations are a powerful primitive for building new control operators
- All examples online (including some that I have not covered)
- Tomorrow: How to implement continuations by source-to-source translation