How to Compile Lambda Expressions
Java 8
From Recursion to Iteration

Lambda Expressions $\rightarrow$ Tail Form
$\rightarrow$ First-Order Form
$\rightarrow$ Imperative Form

Form	**Idea**	**Approach**
Tail	functions never return	continuations
First-order	functions are all top level	data structures
Imperative	functions take no arguments	register allocation

Recursion versus Iteration

Recursion:

```
static Function<Integer,Integer> // in class Test
fact =
    n ->
      n == 0 ?
        1
      : n * Test.fact.apply(n-1);
```

Iteration:

```
static int factIter(int n) {
  int a=1;
  while (n!=0) { a=n*a; n=n-1; }
  return a;
}
```

Low-level Iteration with goto

```
// global register n
// global register a

factIter:  a=1;
Loop:      if (n==0) { }
           else      { a=n*a; n=n-1; goto Loop }
```

The Stack

```
  fact.apply(4)
= 4 * fact.apply(3)
= 4 * 3 * fact.apply(2)
= 4 * 3 * 2 * fact.apply(1)
= 4 * 3 * 2 * 1 * fact.apply(0)
= 4 * 3 * 2 * 1 * 1
= 24
```

```
  factIter(4)
= 24
```

From General Programs to Tail Form

Recursion:

```
static Function<Integer,Integer> // in class Test
fact =
    n ->
      n == 0 ?
        1
      : n * Test.fact.apply(n-1);
```

Tail Form:

Uses continuation passing style (CPS).

```
static BiFunction<Integer, Function<Integer,Integer>,
                  Integer>
factCPS =
    (n,k) ->
      n == 0 ?
        k.apply(1)
      : Test.factCPS.apply(n-1, v -> k.apply(n * v));
```

How to call a function that is written in CPS

```
static BiFunction<Integer, Function<Integer,Integer>,
                           Integer>
factCPS =
    (n,k) ->
      n == 0 ?
        k.apply(1)
      : Test.factCPS.apply(n-1, v -> k.apply(n * v));


factCPS.apply(4, v -> v)
```

Calling a function that is written in CPS

```
    factCPS.apply(4, v1 -> v1)
=   factCPS.apply(3, v2 -> (v1 -> v1)
                              .apply(4 * v2))
=   factCPS.apply(2, v3 -> (v2 -> (v1 -> v1)
                              .apply(4 * v2))
                              .apply(3 * v3))
=   factCPS.apply(1, v4 -> (v3 -> (v2 -> (v1 -> v1)
                              .apply(4 * v2))
                              .apply(3 * v3))
                              .apply(2 * v4))
=   factCPS.apply(0, v5 -> (v4 -> (v3 -> (v2 -> (v1 -> v1)
                              .apply(4 * v2))
                              .apply(3 * v3))
                              .apply(2 * v4))
                              .apply(1 * v5))
```

Calling a function that is written in CPS

```
    factCPS.apply(0, v5 -> (v4 -> (v3 -> (v2 -> (v1 -> v1)
                                .apply(4 * v2))
                                .apply(3 * v3))
                                .apply(2 * v4))
                                .apply(1 * v5))
= (v5 -> (v4 -> (v3 -> (v2 -> (v1 -> v1)
                                .apply(4 * v2))
                                .apply(3 * v3))
                                .apply(2 * v4))
                                .apply(1 * v5))
                                .apply(1)
= (v4 -> (v3 -> (v2 -> (v1 -> v1)
                                .apply(4 * v2))
                                .apply(3 * v3))
                                .apply(2 * v4))
                                .apply(1)       // 1 * 1 = 1
```

Calling a function that is written in CPS

```
  (v4 -> (v3 -> (v2 -> (v1 -> v1)
                          .apply(4 * v2))
                          .apply(3 * v3))
                          .apply(2 * v4))
                          .apply(1)
= (v3 -> (v2 -> (v1 -> v1)
                          .apply(4 * v2))
                          .apply(3 * v3))
                          .apply(2)      // 2 * 1 = 2
= (v2 -> (v1 -> v1)
                          .apply(4 * v2))
                          .apply(6))     // 3 * 2 = 6
= (v1 -> v1)
                          .apply(24))    // 4 * 6 = 24
= 24
```

Grammar

Evaluation of a Tail Form expression (*TailForm*) has **one call** which is the last operation.

Evaluation of a Simple expression (*Simple*) has **no calls**.

$$
\begin{array}{rcl}
\textit{TailForm} & ::= & \textit{Simple} \\
& | & \textit{Simple}\ \texttt{.apply(}\ \textit{Simple}_1\ \texttt{...}\ \textit{Simple}_n\ \texttt{)} \\
& | & \textit{Simple}\ \texttt{?}\ \textit{TailForm}\ \texttt{:}\ \textit{TailForm} \\
\\
\textit{Simple} & ::= & \textit{Identifier} \\
& | & \textit{Constant} \\
& | & \textit{Simple PrimitiveOperation Simple} \\
& | & \text{Identifier}\ \texttt{->}\ \textit{TailForm}
\end{array}
$$

Examples

Tail Form (not Simple):

```
k.apply(1)
gcd(y, (x % y))
y == 0 ? x : gcd(y, (x % y))
```

Simple:

```
n
0
n == 0
v -> v
v2 -> k.apply(v1 + v2)
```

Not Tail Form:

```
n * Test.fact.apply(n-1)
Test.fib.apply(n-1) + Test.fib.apply(n-2)
a.member(b) ? 1 : 2
n == 0 ? 1 : n * Test.fact.apply(n-1)
n -> n == 0 ? 1 : n * Test.fact.apply(n-1)
```

The Transformation Rules (1/2)

```
        static Function<...> // in class Test
        foo =
            x -> ---
==>
        static BiFunction<...> // in class Test
        fooCPS =
            (x,k) -> k.apply( --- )


        k.apply( foo.apply(a, n-1) )
==>
        fooCPS.apply(a, n-1, k)


        k.apply(----- (foo.apply(a, n-1)) -----)
==>
        fooCPS.apply(a, n-1, v -> k.apply(----- v ----))
```

The Transformation Rules (2/2)

```
        k.apply(y ? --- : ---)
==>
        y ? k.apply(---) : k.apply(---))
```

```
        k.apply((foo x) ? --- : ---)
==>
        fooCPS(x, v -> k.apply(v ? ... : ...))
==>
        fooCPS(x, v -> v ? k.apply(---) : k.apply(---))
```

Example (1/3)

Recursion:

```
static Function<Integer,Integer> // in class Test
fact =
    n ->
      n == 0 ?
        1
      : n * Test.fact.apply(n-1);
```

After the first step:

```
static BiFunction<Integer, Function<Integer,Integer>,
                  Integer>
factCPS =
    (n,k) -> k.apply (
      n == 0 ?
        1
      : n * Test.fact.apply(n-1) );
```

Example (2/3)

After the first step:

```
static BiFunction<Integer, Function<Integer,Integer>,
                  Integer>
factCPS =
    (n,k) -> k.apply (
      n == 0 ?
        1
      : n * Test.fact.apply(n-1) );
```

After the second step:

```
static BiFunction<Integer, Function<Integer,Integer>,
                  Integer>
factCPS =
    (n,k) ->
      n == 0 ?
        k.apply(1)
      : k.apply( n * Test.fact.apply(n-1) );
```

Example (3/3)

After the second step:

```
static BiFunction<Integer, Function<Integer,Integer>,
                          Integer>
factCPS =
      (n,k) ->
        n == 0 ?
          k.apply(1)
        : k.apply( n * Test.fact.apply(n-1) );
```

After the third step, we have Tail Form:

```
static BiFunction<Integer, Function<Integer,Integer>,
                          Integer>
factCPS =
      (n,k) ->
        n == 0 ?
          k.apply(1)
        : Test.factCPS.apply(n-1, v -> k.apply(n * v));
```

CPS Transformation of simply-typed λ-terms (CBV)

$$\begin{array}{ll} \text{Types:} & t ::= t \to t \mid \mathsf{Int} \\ \text{Expressions:} & e ::= x \mid \lambda x.e \mid e_1 e_2 \end{array}$$

Here is a CPS transformation, defined by a function $[\![\cdot]\!]$:

$$\begin{array}{rcl} [\![x]\!] & = & \lambda k.kx \\ [\![\lambda x.e]\!] & = & \lambda k.k(\lambda x.[\![e]\!]) \\ [\![e_1 e_2]\!] & = & \lambda k.[\![e_1]\!](\lambda v_1.[\![e_2]\!](\lambda v_2.(v_1 v_2)k)) \end{array}$$

Let o be a type of answers. Define t^* inductively:

$$\begin{array}{rcl} \mathsf{Int}^* & = & \mathsf{Int} \\ (s \to t)^* & = & s^* \to (t^* \to o) \to o \end{array}$$

Define $A^*(x) = t^*$ if $A(x) = t$.

Theorem If $A \vdash e : t$, then $A^* \vdash [\![e]\!] : (t^* \to o) \to o$.

From Tail Form to First-Order Form

```
static BiFunction<Integer, Function<Integer,Integer>,
                  Integer>
factCPS =
    (n,k) ->
      n == 0 ?
        k.apply(1)
      : Test.factCPS.apply(n-1, v -> k.apply(n * v));

factCPS.apply(4, v -> v)
```

Continuations for `factCPS`

Continuation	::=	`v -> v`
	|	`v ->` *Continuation*`.apply(n * v)`

Continuations as a Datatype (1/2)

Continuations for `factCPS`

Continuation ::= `v -> v`
  | `v ->` *Continuation*`.apply(n * v)`

```
interface Continuation {
  Integer apply(Integer a);
}
```

Continuations as a Datatype (2/2)

```
class Identity implements Continuation {
  public Integer apply(Integer a) {
    return a;
  }
}

class FactRec implements Continuation {
  Integer n; Continuation k;
  public FactRec(Integer n_prime, Continuation k_prime) {
    n = n_prime; k = k_prime;
  }

  public Integer apply(Integer v) {
    return k.apply(n * v);
  }
}
```

From Tail Form to First-Order Form

```
static BiFunction<Integer, Function<Integer,Integer>,
                          Integer>
factCPS =
    (n,k) ->
      n == 0 ?
        k.apply(1)
      : Test.factCPS.apply(n-1, v -> k.apply(n * v));

factCPS.apply(4, v -> v)

static BiFunction<Integer,Continuation,Integer>
factCPSadt =
    (n,k) ->
      n == 0 ?
        k.apply(1)
      : Test.factCPSadt.apply(n-1, new FactRec(n,k));

factCPSadt.apply(4, new Identity())
```

Clever Representation of a Continuation

Continuations for `factCPS`

$$\begin{array}{lll} \textit{Continuation} & ::= & \texttt{v -> v} \\ & \mid & \texttt{v -> } \textit{Continuation}\texttt{.apply(n * v)} \end{array}$$

Claim: Every `factCPS` continuation is of the form

`v -> p * v` for some `p`.

Base case: `v -> v` = `v -> 1 * v`

Induction step: Suppose

```
       k  =  w -> p * w
  Now:
             v -> k.apply(n * v)
          =  v -> (w -> p * w).apply(n * v)
          =  v -> p * (n * v)
          =  v -> (p * n) * v
```

Represent a Continuation by a Number

```
static BiFunction<Integer, Function<Integer,Integer>,
                  Integer>
factCPS =
    (n,k) ->
      n == 0 ?
        k.apply(1)
      : Test.factCPS.apply(n-1, v -> k.apply(n * v));

factCPS.apply(4, v -> v)
```

Represent the continuation `v -> p * v` by the number `p`.

Suppose `k` represents `w -> p * w`
Notice: `k.apply(1)    =    p * 1    =    p`
Notice: `k.apply(n * v) = p * (n * v) = (p * n) * v`
Notice: `v -> k.apply(n * v)    =    v -> (p * n) * v`
which we can represent by `(p * n)`

Represent a Continuation by a Number

```
static BiFunction<Integer, Function<Integer,Integer>,
                  Integer>
factCPS =
    (n,k) ->
      n == 0 ?
        k.apply(1)
      : Test.factCPS.apply(n-1, v -> k.apply(n * v));

factCPS.apply(4, v -> v)
```

```
static BiFunction<Integer,Integer,Integer>
factCPSnum =
    (n,k) ->
      n == 0 ?
        k
      : Test.factCPSnum.apply(n-1, k*n);
```

From First-Order Form to Imperative Form

Get rid of the function parameters; instead use global registers.

```
static Integer n;
static Integer k;

static void factCPSimp() {
   if (n == 0) { }
   else        { k = k*n; n = n-1; factCPSimp(); }
}

n = 4; k = 1;
factCPSimp();
System.out.println(k);
```

From Imperative Form to Low-Level Code

```
static Integer n;
static Integer k;


n = 4; k = 1;
factCPSimp:
   if (n == 0) { }
   else        { k = k*n; n = n-1; goto factCPSimp }
System.out.println(k);
```

Example: Euclid's Algorithm

```
public static int gcd(int x, int y) {
  return
    y == 0 ?
      x
    : gcd(y, (x % y));
}
```

Example: Even-Odd Deciders

```
static Function <Integer,Boolean>
even =
    n ->
      n == 0 ?
        true
      : Test.odd.apply(n-1);

static Function <Integer,Boolean>
odd =
    n ->
      n == 0 ?
        false
      : Test.even.apply(n-1);
```

Example: Fibonacci (1/5)

```
static Function<Integer,Integer>
fib =
    n ->
      n <= 2 ?
        1
      : Test.fib.apply(n-1) + Test.fib.apply(n-2);
```

Example: Fibonacci (2/5)

```
static BiFunction<Integer,Function<Integer,Integer>,
                  Integer>
fibCPS =
    (n,k) ->
      n <= 2 ?
        k.apply(1)
      : Test.fibCPS.apply(
           n-1,
           v1 -> Test.fibCPS.apply(
                         n-2,
                         v2 -> k.apply(v1 + v2)));
```

```
fibCPS.apply(6, v -> v)
```

Example: Fibonacci (3/5)

```
class FibRec1 implements Continuation {
  Integer n;
  Continuation k;
  public FibRec1(Integer n_prime, Continuation k_prime) {
    n = n_prime;
    k = k_prime;
  }

  public Integer apply(Integer v) {
    return Test.fibCPSadt.apply(
                              n-2,
                              new FibRec2(v,k));
  }
}
```

Example: Fibonacci (4/5)

```
class FibRec2 implements Continuation {
  Integer v1;
  Continuation k;
  public FibRec2(Integer v1_prime, Continuation k_prime) {
    v1 = v1_prime;
     k = k_prime;
  }

  public Integer apply(Integer v) {
    return k.apply(v1 + v);
  }
}
```

Example: Fibonacci (5/5)

```
static BiFunction<Integer,Continuation,Integer>
fibCPSadt =
    (n,k) ->
      n <= 2 ?
        k.apply(1)
      : Test.fibCPSadt.apply(
           n-1,
           new FibRec1(n,k));


fibCPSadt.apply(6, new Identity())
```