Tail recursion
Functional Programming

Jens Egholm Pedersen and Anders Kalhauge

Spring 2017

Outline

- Stacks
- Stacks in CPUs
  - Exercise 1
- Tail recursion
- Hand-in Tail recursion

Stacks

Stacks are LIFO queues

A stack is a **L**ast **I**n **F**irst **O**ut queue.

CPU and instructions

CPUs executes instructions sequentially

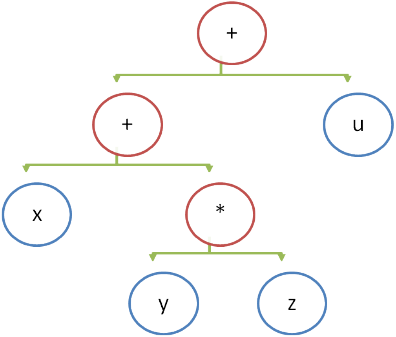
CPU and instructions

CPUs executes instructions sequentially

So how is `x + y * z + u` possible?

CPUs executes instructions sequentially

So how is `x + y * z + u` possible?



Call stack

Your computer *queues* instructions

Call stack

Your computer *queues* instructions in a **call stack**

Call stack

Your computer *queues* instructions in a **call stack**
Typically blocks of code

Call stack

Your computer *queues* instructions in a **call stack**
Typically blocks of code or **procedures / subroutines**

Call stack

Your computer *queues* instructions in a **call stack**
Typically blocks of code or **procedures / subroutines**

Stack content

A `call stack` basically
consists of calls into subroutines

Stack content

A `call stack` basically consists of calls into subroutines

They are called `stack frames`

Frame content

What does the frame contain? What do we need to know?

Frame content

What does the frame contain? What do we need to know?

Frame content

What does the frame contain? What do we need to know?

- Space for arguments and variables

Frame content

What does the frame contain? What do we need to know?

- Space for arguments and variables
- Return address

Frame content

What does the frame contain? What do we need to know?

- Space for arguments and variables
- Return address: Where do we go after we're done?

Frame content

What does the frame contain? What do we need to know?

- Space for arguments and variables
- Return address: Where do we go after we're done?

Disclaimer: Implementation specific

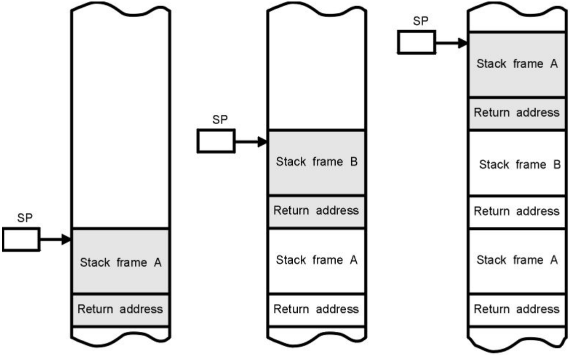
Stack frames

What happens every time we call a function?

Stack frames

What happens every time we call a function?

We create a new stack frame!



Recursion on the stack


```
public void count(int i) {
  if (i > 0) {
    return i + count(i - i);
  } else return i;
}
```

Recursion on the stack

```
public void count(int i) {
  if (i > 0) {
    return i + count(i - i);
  } else return i;
}
```

`count(3)`

Recursion on the stack

```
public void count(int i) {
  if (i > 0) {
    return i + count(i - i);
  } else return i;
}
```

```
count(3)
  count(2)
```

Recursion on the stack

```
public void count(int i) {
  if (i > 0) {
    return i + count(i - i);
  } else return i;
}
```

```
count(3)
  count(2)
    count(1)
```

```
public void count(int i) {
  if (i > 0) {
    return i + count(i - i);
  } else return i;
}
```

```
count(3)
  count(2)
    count(1)
      count(0)
```

```java
public void count(int i) {
  if (i > 0) {
    return i + count(i - i);
  } else return i;
}
```

```
count(3)
  count(2)
    count(1)
      count(0)
    // 1 + 0
```

```java
public void count(int i) {
  if (i > 0) {
    return i + count(i - i);
  } else return i;
}
```

```
count(3)
  count(2)
    count(1)
      count(0)
    // 1 + 0
  // 2 + 1
```

Recursion on the stack

```
public void count(int i) {
  if (i > 0) {
    return i + count(i - i);
  } else return i;
}
```

```
count(3)
  count(2)
    count(1)
      count(0)
    // 1 + 0
  // 2 + 1
// 3 + 3
```

Small exercise

```java
public long fibonacci(int n) {
  if (n <= 1) return n;
  else return fibonacci(n-1) + fibonacci(n-2);
}
```

```
public long fibonacci(int n) {
  if (n <= 1) return n;
  else return fibonacci(n-1) + fibonacci(n-2);
}
```

Small exercise

```
public long fibonacci(int n) {
  if (n <= 1) return n;
  else return fibonacci(n-1) + fibonacci(n-2);
}
```

1. How many stack frames do we create with `fibonacci(2)`?

Small exercise

```java
public long fibonacci(int n) {
    if (n <= 1) return n;
    else return fibonacci(n-1) + fibonacci(n-2);
}
```

1. How many stack frames do we create with `fibonacci(2)`?
2. How many stack frames do we create with `fibonacci(5)`?

Small exercise

```java
public long fibonacci(int n) {
  if (n <= 1) return n;
  else return fibonacci(n-1) + fibonacci(n-2);
}
```

1. How many stack frames do we create with `fibonacci(2)`?
2. How many stack frames do we create with `fibonacci(5)`?
3. How many stack frames do we create with `fibonacci(10)`?

```
public long fibonacci(int n) {
  if (n <= 1) return n;
  else return fibonacci(n-1) + fibonacci(n-2);
}
```

1. How many stack frames do we create with `fibonacci(2)`?
2. How many stack frames do we create with `fibonacci(5)`?
3. How many stack frames do we create with `fibonacci(10)`?
4. What is the general formula for how many stack frames the `fibonacci` function creates?

Exercise 1

In a new Java project:

Exercise 1

In a new Java project:

1. Implement a function for factorial using `BigInteger`
   ```
   public static BigInteger factorial(BigInteger i);
   ```

Exercise 1

In a new Java project:

1. Implement a function for factorial using `BigInteger`
   `public static BigInteger factorial(BigInteger i);`
2. Run factorial with `100'000` as input. What happens?

Exercise 1

In a new Java project:

1. Implement a function for factorial using `BigInteger`
   `public static BigInteger factorial(BigInteger i);`
2. Run factorial with `100'000` as input. What happens?
3. Try to run it with `10`. Better now?

In a new Java project:

1. Implement a function for factorial using `BigInteger`
   `public static BigInteger factorial(BigInteger i);`
2. Run factorial with `100’000` as input. What happens?
3. Try to run it with `10`. Better now?
4. In `Run -> Set Project Configuration -> Customize...`
   `-> VM Options` write ‘`-Xss20m`’

Exercise 1

In a new Java project:

1. Implement a function for factorial using `BigInteger`
   `public static BigInteger factorial(BigInteger i);`
2. Run factorial with `100’000` as input. What happens?
3. Try to run it with `10`. Better now?
4. In `Run -> Set Project Configuration -> Customize... -> VM Options` write ‘`-Xss20m`’
5. Run it with `100’000` as input. What happened?

What is recursion

Think about factorial. What is the input and what is the output?

What is recursion

Think about factorial. What is the input and what is the output?

Think about fibonacci. What is the input and what is the output?

What is recursion

Think about factorial. What is the input and what is the output?

Think about fibonacci. What is the input and what is the output?

Is there a pattern?

What is recursion

Think about factorial. What is the input and what is the output?

Think about fibonacci. What is the input and what is the output?

Is there a pattern?

Yes there is!

```java
public int count(int i) {
  if (i <= 0) return 0;
  else return count(i - 1)
}
```

Stack frame for counting

```java
public int count(int i) {
  if (i <= 0) return 0;
  else return count(i - 1)
}
```

What is in a stack frame?

Stack frame for counting

```java
public int count(int i) {
  if (i <= 0) return 0;
  else return count(i - 1)
}
```

What is in a stack frame?

$\boxed{i}$ $\boxed{i - 1}$ $\boxed{count(i - 1)}$

Simplifying stack frames

```java
public int count(int i) {
  if (i <= 0) return 0;
  else return count(i - 1)
}
```

```
public int count(int i) {
  if (i <= 0) return 0;
  else return count(i - 1)
}
```

call | i | i - 1 | count(i - 1)

```
public int count(int i) {
  if (i <= 0) return 0;
  else return count(i - 1)
}
```

call	$\boxed{\text{i}}$	$\boxed{\text{i - 1}}$	$\boxed{\text{count(i - 1)}}$
count(3)	$\boxed{3}$	$\boxed{2}$	$\boxed{\text{count(2)}}$

Simplifying stack frames

```
public int count(int i) {
  if (i <= 0) return 0;
  else return count(i - 1)
}
```

call	i	i - 1	count(i - 1)
`count(3)`	3	2	count(2)
`count(2)`	2	1	count(1)

Simplifying stack frames

```
public int count(int i) {
  if (i <= 0) return 0;
  else return count(i - 1)
}
```

call	i	i - 1	count(i - 1)
count(3)	3	2	count(2)
count(2)	2	1	count(1)
count(1)	1	0	count(0)

```
public int count(int i) {
  if (i <= 0) return 0;
  else return count(i - 1)
}
```

call	i	i - 1	count(i - 1)
count(3)	3	2	count(2)
count(2)	2	1	count(1)
count(1)	1	0	count(0)
count(0)	0	?	?

cphbusiness COPENHAGEN BUSINESS ACADEMY

```
public int count(int i) {
    if (i <= 0) return 0;
    else return count(i - 1)
}
```

call	i	i - 1	count(i - 1)
count(3)	3	2	count(2)
count(2)	2	1	count(1)
count(1)	1	0	count(0)
count(0)	0	?	?
count(1)	1	0	0

```java
public int count(int i) {
  if (i <= 0) return 0;
  else return count(i - 1)
}
```

call	i	i - 1	count(i - 1)
count(3)	3	2	count(2)
count(2)	2	1	count(1)
count(1)	1	0	count(0)
count(0)	0	?	?
count(1)	1	0	0
count(2)	2	1	0

```
public int count(int i) {
  if (i <= 0) return 0;
  else return count(i - 1)
}
```

call	i	i - 1	count(i - 1)
count(3)	3	2	count(2)
count(2)	2	1	count(1)
count(1)	1	0	count(0)
count(0)	0	?	?
count(1)	1	0	0
count(2)	2	1	0
count(3)	3	2	0

```java
public int count(int i) {
  if (i <= 0) return 0;
  else return count(i - 1)
}
```

call	i	i - 1	count(i - 1)
`count(3)`	3	2	count(2)

Simplifying stack frames

```java
public int count(int i) {
  if (i <= 0) return 0;
  else return count(i - 1)
}
```

call	i	i - 1	count(i - 1)
count(2)	2	1	count(1)

```
public int count(int i) {
  if (i <= 0) return 0;
  else return count(i - 1)
}
```

call	$\boxed{i}$	$\boxed{i - 1}$	$\boxed{\text{count}(i - 1)}$
count(1)	$\boxed{1}$	$\boxed{0}$	$\boxed{\text{count}(0)}$

```java
public int count(int i) {
  if (i <= 0) return 0;
  else return count(i - 1)
}
```

call	$\boxed{\text{i}}$	$\boxed{\text{i - 1}}$	$\boxed{\text{count(i - 1)}}$
count(0)	$\boxed{0}$	$\boxed{?}$	$\boxed{?}$

Tail call elimination

This is called **tail call elimination**.

Tail call elimination

This is called **tail call elimination**. Why?

Tail call elimination

This is called **tail call elimination**. Why?

Because there are **no recursive calls**.

Tail call elimination

This is called **tail call elimination**. Why?

Because there are **no recursive calls**.

We simply update the stack

Java analogy

Using tail call elimination we can go from

Java analogy

Using tail call elimination we can go from

```java
public int count(int i) {
  if (i <= 0) return 0;
  else return count(i - 1)
}
```

Using tail call elimination we can go from

```java
public int count(int i) {
  if (i <= 0) return 0;
  else return count(i - 1)
}
```

to

Java analogy

Using tail call elimination we can go from

```
public int count(int i) {
  if (i <= 0) return 0;
  else return count(i - 1)
}
```

to

```
public int count(int i) {
  while(i > 0) {
    i--;
  }
  return i;
}
```

Tail call elimination scope 1/2

Can you think of situations where **tail call elimination** *cannot* be used?

Tail call elimination scope 1/2

Can you think of situations where **tail call elimination** *cannot* be used?

Can this be optimised with tail call elimination?

```
public int count(int i) {
  if (count <= 0) return 0;
  else {
    int newCount = count(i - 1);
    return newCount;
  }
}
```

Tail call elimination scope 1/2

Can you think of situations where **tail call elimination** *cannot* be used?
Can this be optimised with tail call elimination?

```
public int count(int i) {
  if (count <= 0) return 0;
  else {
    int newCount = count(i - 1);
    return newCount;
  }
}
```

No.

Can you think of situations where **tail call elimination** *cannot* be used?
Can this be optimised with tail call elimination?

```
public int count(int i) {
  if (count <= 0) return 0;
  else {
    int newCount = count(i - 1);
    return newCount;
  }
}
```

No. The recursive call is not the last!

Tail call elimination scope 2/2

Can this be optimised with tail call elimination?

```java
public long sum(long n) {
  if (n <= 1) return n;
  else return n + sum(n - 1);
}
```

Can this be optimised with tail call elimination?

```
public long sum(long n) {
  if (n <= 1) return n;
  else return n + sum(n - 1);
}
```

No.

Tail call elimination scope 2/2

Can this be optimised with tail call elimination?

```
public long sum(long n) {
  if (n <= 1) return n;
  else return n + sum(n - 1);
}
```

No. The recursive call is not the last!

Tail call elimination scope 2/2

Can this be optimised with tail call elimination?

```
public long sum(long n) {
  if (n <= 1) return n;
  else return n + sum(n - 1);
}
```

No. The recursive call is not the last! Let's try that in Elm.

Can this be optimised with tail call elimination?

```
public long sum(long n) {
  if (n <= 1) return n;
  else return n + sum(n - 1);
}
```

No. The recursive call is not the last! Let's try that in Elm.
Let's try that in Java!

Tail recursion

This technique is called **tail recursion**

This technique is called **tail recursion**

- Replaces tail calls with stack frame manipulation

Tail recursion

This technique is called **tail recursion**

- Replaces tail calls with stack frame manipulation
- Saves stack frames

This technique is called **tail recursion**

- Replaces tail calls with stack frame manipulation
- Saves stack frames
- Note: Java does **not** have it!

Java bytecode example

See the example from `https://github.com/cphbus-functional-programming/java-exercises` in the `tailrecursion` project.

Fibonacci example

Can this be optimised with tail call elimination?

```
public int fibonacci(int i) \{
  if (i < 2) return i;
  else return fibonacci(i - 1) + fibonacci(i - 2)
\}
```

Fibonacci example

Can this be optimised with tail call elimination?

```
public int fibonacci(int i) \{
  if (i < 2) return i;
  else return fibonacci(i - 1) + fibonacci(i - 2)
\}
```

No.

Fibonacci example

Can this be optimised with tail call elimination?

```
public int fibonacci(int i) \{
  if (i < 2) return i;
  else return fibonacci(i - 1) + fibonacci(i - 2)
\}
```

No. There are two recursive calls

Can this be optimised with tail call elimination?

```
public int fibonacci(int i) \{
  if (i < 2) return i;
  else return fibonacci(i - 1) + fibonacci(i - 2)
\}
```

No. There are two recursive calls

Can we optimise it to eliminate tail calls?

Can this be optimised with tail call elimination?

```
public int fibonacci(int i) \{
  if (i < 2) return i;
  else return fibonacci(i - 1) + fibonacci(i - 2)
\}
```

No. There are two recursive calls

Can we optimise it to eliminate tail calls?

```
public int fibonacci(int i, int previous, int next) {
  if      (i == 0) return previous;
  else if (i == 1) return next;
  else             return fibonacci(i - 1, next,
                                    (next + previous));
}
```

Hand-in

1. Finish the virtual CPU

Hand-in

1. Finish the virtual CPU
2. Program a tail recursive version of factorial

Hand-in

1. Finish the virtual CPU
2. Program a tail recursive version of factorial

Hint: Use the factorial implementation from Anders on GitHub under **general/ 02 Assignment**