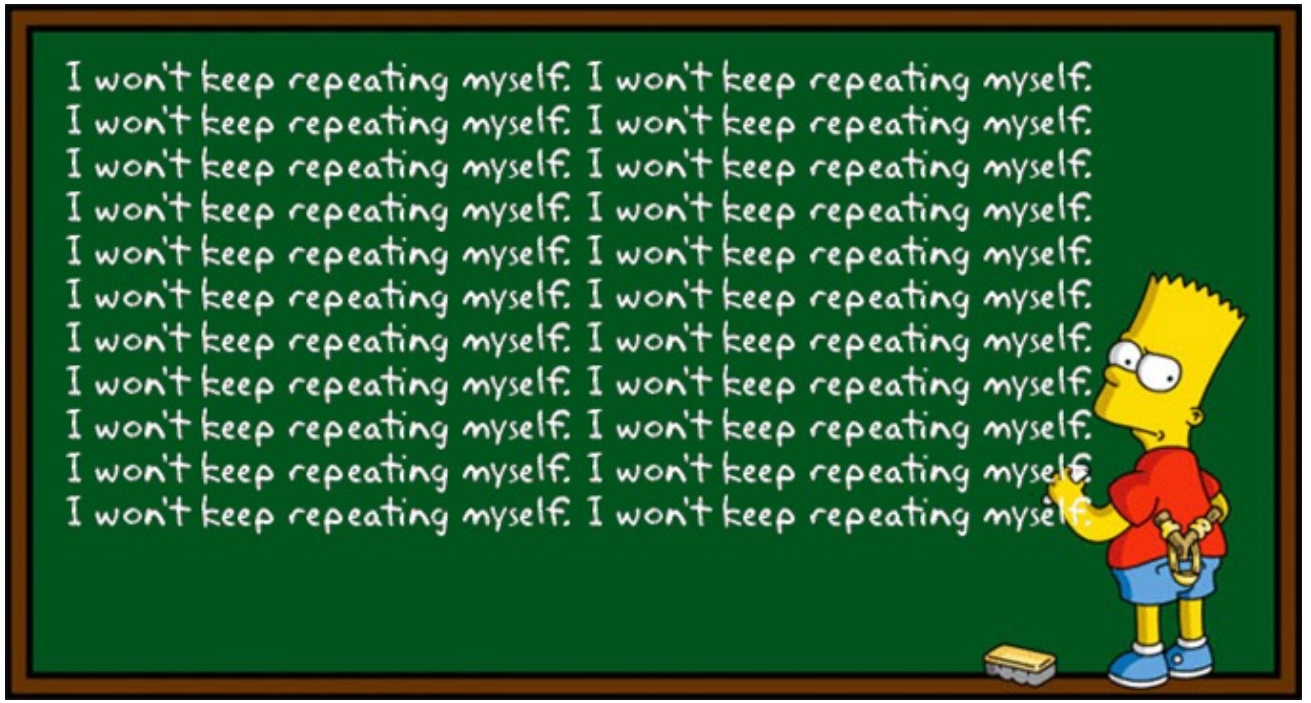
IN3040 – Repetition / Exam from 2017

Exercise 1, Runtime systems, scoping and types (40 %)



Eyvind W. Axelsen - eyvinda@ifi.uio.no

Task 1

Consider the program below, which is written in some new, hitherto unknown language (i.e., it is made up for this exam). The language supports interfaces and structs as the main units of decomposition, as well as variables and functions ("methods") within such structs.

Assume that the program is correct in the given language, that is, it contains no syntax errors or type errors, and its execution does not result in a runtime error.

Furthermore, assume that the program starts executing the Main function, and that print(<expr>) will print the string representation of the expression (akin to toString() in Java) to the console/terminal.

Task 1a

```
interface I {
  int function1();
}

struct A   {
  int function1() {
    int x = 21;
    int y = x +
       function2();
    return y;
  }
  int function2() {
    return x  ;
} }
```

```
struct B {
  int thisOneGoesTo = 11;
  int function1() {
    return thisOneGoesTo;
  }
}
```

```
void Main() {
  I myI = new A();
  int result = getValue(myI);

  myI = new B();
  result = result + getValue(myI);
  printResult();
}

int getValue(I i) {
  return i.function1();
}

void printResult() {
  print("The result is: ");
  print(result); // HERE!
}
```

1a) With a reasonable semantics, and a corresponding implementation of the language, what will be printed by the statement labeled "HERE!" in the program text above? Explain your reasoning briefly.

Task 1b

```
interface I {
  int function1();
}

struct A   {
  int function1() {
    int x = 21;
    int y = x +
       function2();
    return y;
  }
  int function2() {
    return x  ;
} }
```

```
struct B {
  int thisOneGoesTo = 11;
  int function1() {
    return thisOneGoesTo;
  }
}
```

```
void Main() {
  I myI = new A();
  int result = getValue(myI);

  myI = new B();
  result = result + getValue(myI);
  printResult();
}

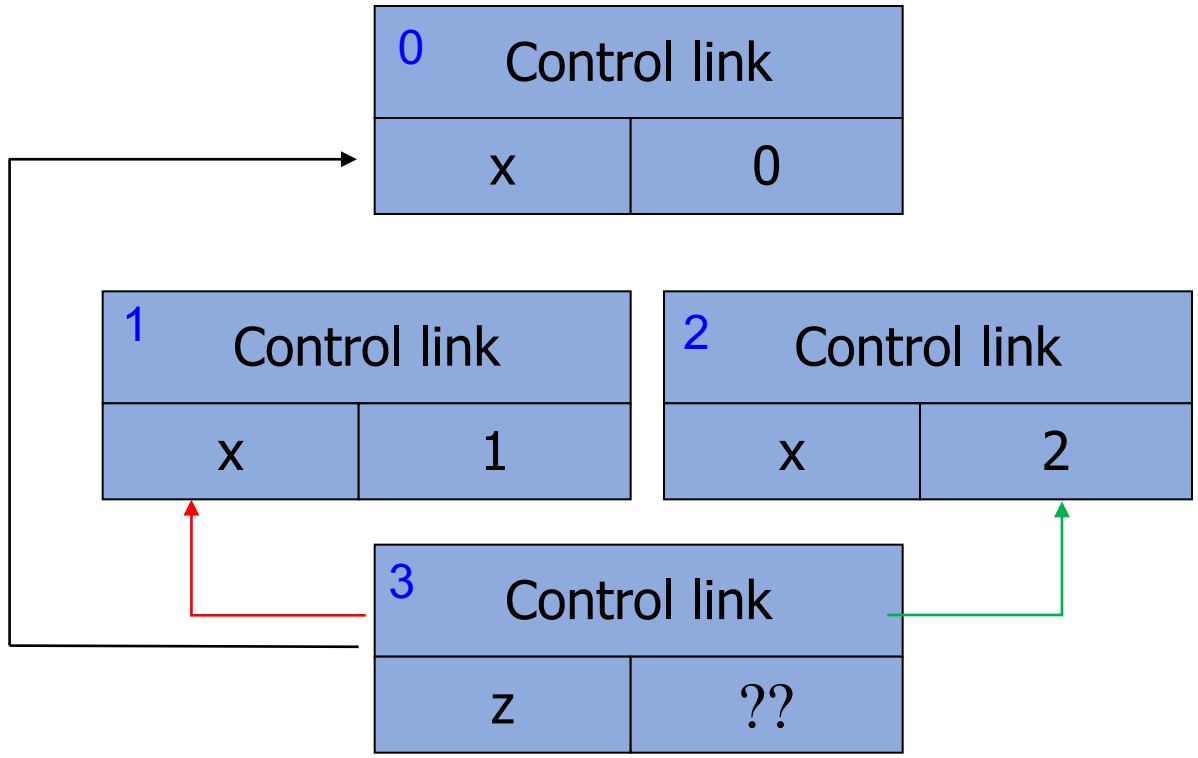
int getValue(I i) {
  return i.function1();
}

void printResult() {
  print("The result is: ");
  print(result); // HERE!
}
```

1b) It is reasonable to assume that this language has some properties that differ a bit from most "mainstream" languages such as e.g. Java and C#. Discuss briefly (and informally) your interpretation of the semantics of the language, focusing on topics discussed in class.

Lecture: Static/dynamic scoping

```
{  -- block 0
    int x = 0;
    {   -- block 1
        int x = 1;
    };
    {   -- block 2
        int x = 2;
    };
    {   -- block 3, called
        int z = x;
    }
}
```



If block 0 executes block 3
Static scope: z = ?
Dynamic scope: z = ?

If block 1 executes block 3
Static scope: z = ?
Dynamic scope: z = ?

If block 2 executes block 3
Static scope: z = ?
Dynamic scope: z = ?

Lecture: Type compatibility – is this OK?

```
class A {
  x: number;
}

class B {
  x: number;
}

var ab: A = new B();
ab.x = 42;
```

It depends!

- In Java?
- In Python?
- In JavaScript?
- In TypeScript?
- In C#?

Lecture: Type compatibility

- Nominally compatible
  - Values of types with the same name are compatible
- Structurally compatible
  - Types T1 and T2 are compatible
    - If T1 is nominally compatible with T2, or
    - T1 and T2 have the same signature (functions, variables, including names of such)

```
struct Position {int x, y, z; };

Position pos;

struct Date { int m, d, y; };
Date today;

void show(Date d);

...; show(today); ...

...; show(pos); ...
```

```
struct Complex { real x, y; };

struct Point { real x, y; };
```

Task 1c

```
interface I {
  int function1();
}

struct A   {
  int function1() {
    int x = 21;
    int y = x +
       function2();
    return y;
  }
  int function2() {
    return x  ;
} }
```

```
struct B {
  int thisOneGoesTo = 11;
  int function1() {
    return thisOneGoesTo;
  }
}
```

```
void Main() {
  I myI = new A();
  int result = getValue(myI);

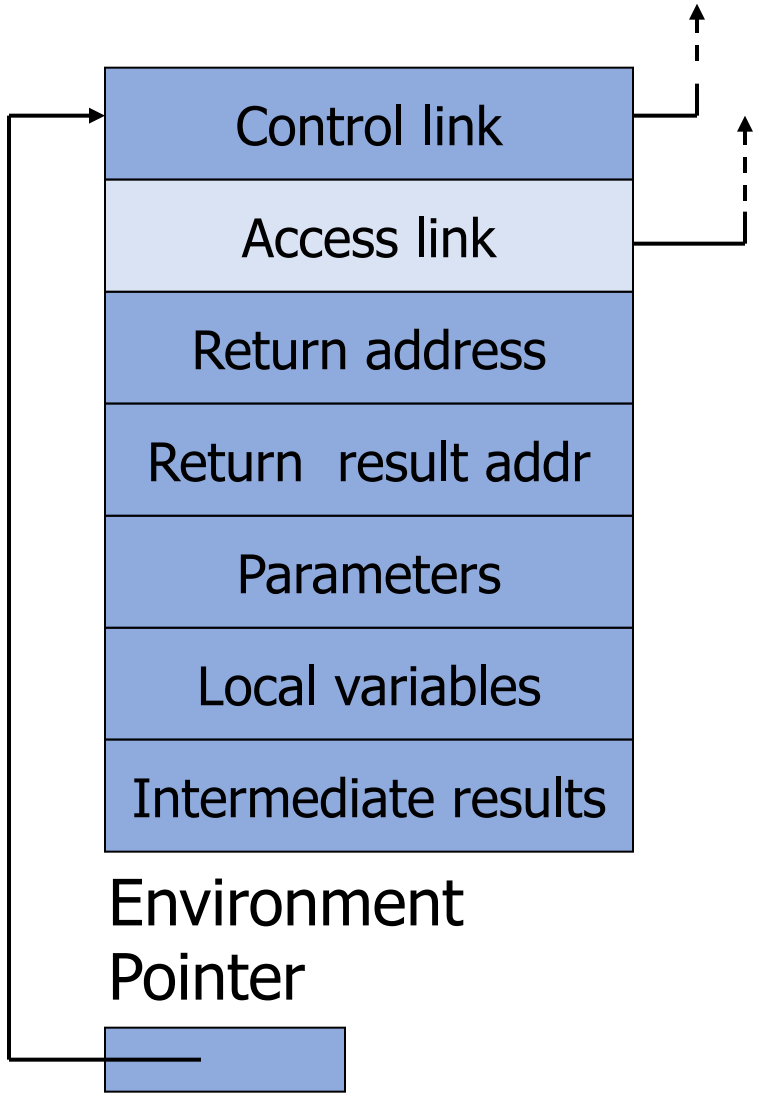
  myI = new B();
  result = result + getValue(myI);
  printResult();
}

int getValue(I i) {
  return i.function1();
}

void printResult() {
  print("The result is: ");
  print(result); // HERE!
}
```

1c) Draw the runtime stack at the point labeled "HERE!" in the program text above. That is, right after the call to print(result). Utilize your interpretation of the language's semantics from question 1b. Include objects and closures in the drawing as necessary.

Lecture: Activation record for static scope



- Control link (dynamic link)
  - Link to activation record of previous (calling) block
  - Why is it called *dynamic?*
- Access link (static link)
  - Link to activation record corresponding to the closest enclosing block in program text
  - Why is it called *static*?
- Difference
  - Control link depends on dynamic behavior of program
  - Access link depends on static form of program text

Lecture: Static scope with access links

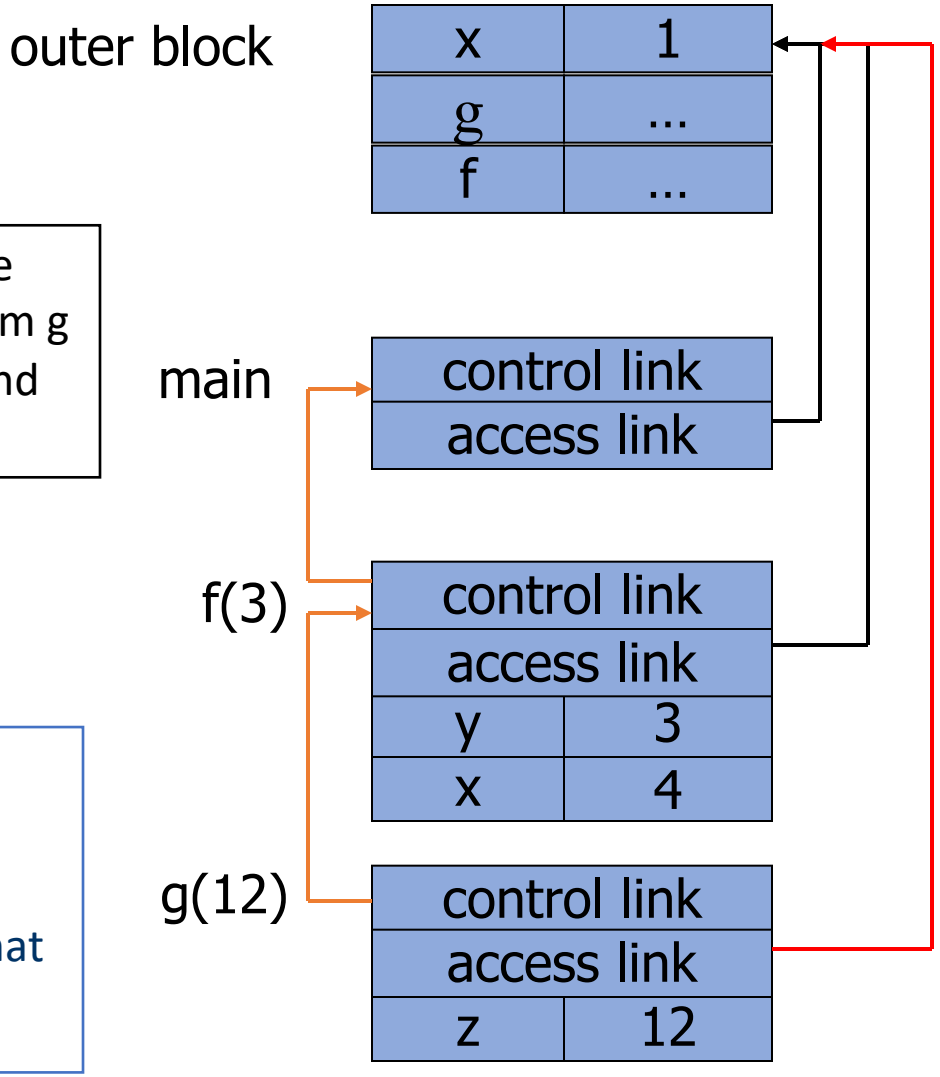
```
{
  int x = 1;

  int function g(z) { return x+z };

  int function f(y) {
      int x = y+1;
      return g(y*x)
   };

  main() {
    f(3);
  }
}
```

What will be the return value from g with dynamic and static scoping?



Use access link to find global variable:

- Access link is always set to frame of closest enclosing *lexical* block
- For function body, this is the block that contains function declaration

Task 1c

```
interface I {
  int function1();
}

struct A   {
  int function1() {
    int x = 21;
    int y = x +
       function2();
    return y;
  }
  int function2() {
    return x  ;
} }
```

```
struct B {
  int thisOneGoesTo = 11;
  int function1() {
    return thisOneGoesTo;
  }
}
```

```
void Main() {
  I myI = new A();
  int result = getValue(myI);

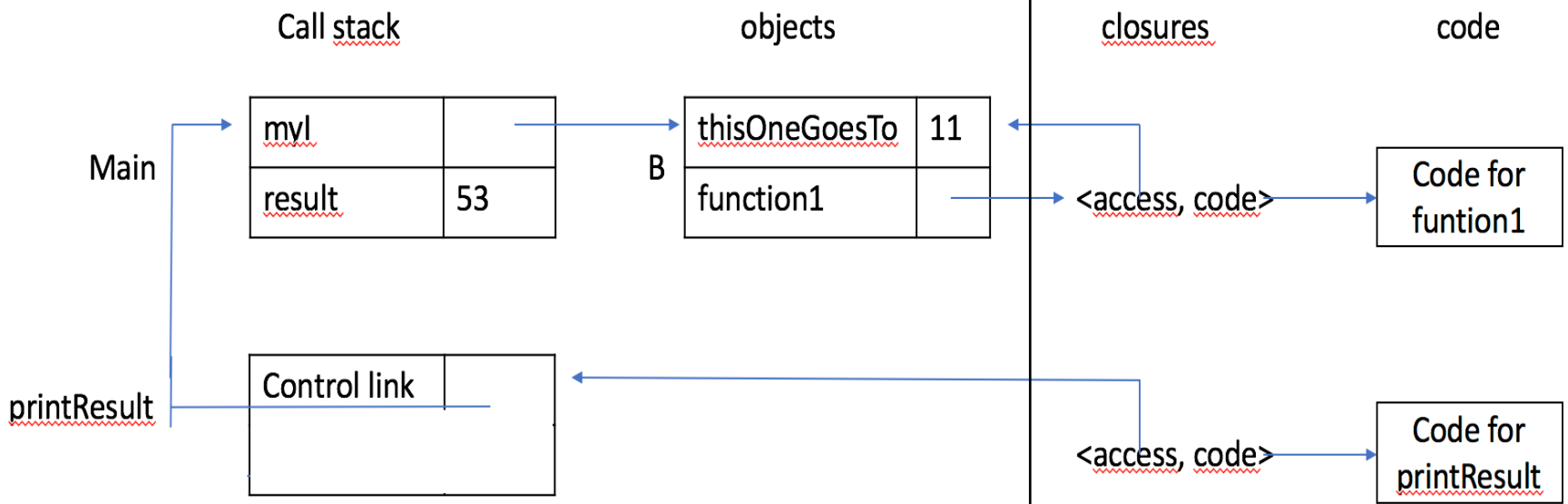
  myI = new B();
  result = result + getValue(myI);
  printResult();
}

int getValue(I i) {
  return i.function1();
}

void printResult() {
  print("The result is: ");
  print(result); // HERE!
}
```

1c) Draw the runtime stack at the point labeled "HERE!" in the program text above. That is, right after the call to print(result). Utilize your interpretation of the language's semantics from question 1b. Include objects and closures in the drawing as necessary.

Not strictly necessary for this task



Task 1d

Consider the following Java program snippet:

```
public class Program {
    public static void main(String[] args) {

        Object[] myArgs = args;

        myArgs[0] = 42;
    }
}
```

Explain why this piece of code is unsafe (there might be more than one reason!). Explain briefly what, from a language designer standpoint, can be done to amend the situation, should you be the creator of Java from scratch.

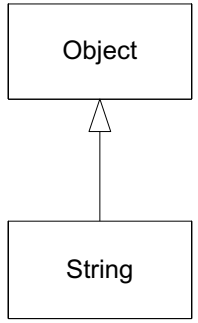
Lecture: Generics and subtyping

- String subtype of Object ⇏ List<String> subtype of List<Object> ?

```
List<String> ls = new ArrayList<String>();
List<Object> lo = ls;
lo.add(new Object());
String s = ls.get(0);
```

compile-time error

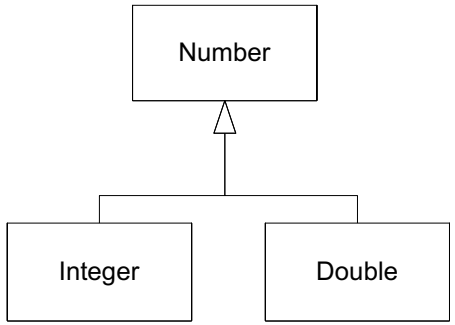
attempts to assign an Object to a String



- Integer subtype of Number ⇏ List<Integer> subtype of List<Number> ?

```
List<Integer> ints = Arrays.asList(1,2);
List<Number> nums = ints;
nums.add(3.14);
```

compile-time error



But look out! Arrays and subtyping

String subtype of Object ➔ String[ ] subtype of Object[ ]?

```
String[] myStrings = new String [10];
myStrings[0] = "Hello";
myStrings[1] = "World!";

Object[] myObjects = myStrings; // ???
myObjects[3] = new Object(); // !!!
```

Try it out in Java and/or C#!

Task 1d

Consider the following Java program snippet:

```
public class Program {
    public static void main(String[] args) {
        Object[] myArgs = args;
        myArgs[0] = 42;
    }
}
```

Explain why this piece of code is unsafe (there might be more than one reason!). Explain briefly what, from a language designer standpoint, can be done to amend the situation, should you be the creator of Java from scratch.

Task 1e

Assume now, that our made-up language should support generic function definitions (parametric polymorphism). Propose a syntax for this. You should do this by providing an EBNF notation for function signatures. (You do not need to include a definition of the function body.) The definition should support generic parameters with both covariant and contravariant constraints. Explain how this is solved with your suggestion.

You can assume that standard definitions of identifiers, strings, parameter lists, etc are provided in the grammar for you.

Use the following notational convention when writing your EBNF grammar:

<non-terminal>
“terminal”
[optional]
alternative1 | alternative2
(grouping)
zero-or-more-repetitions*
one-or-more-repetitions+

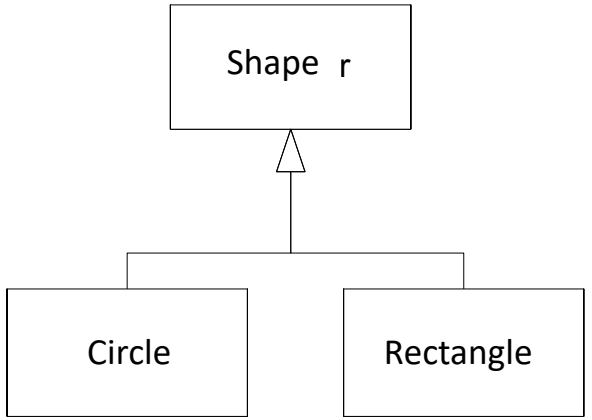
Lecture: Bounded polymorhpism - Wildcards

```
public abstract class Shape {
  public abstract void draw(Canvas c);
}

public class Circle extends Shape {
  private int x, y, radius;
  public void draw(Canvas c) { ... }
}

public class Rectangle extends Shape {
  private int x, y, width, height;
  public void draw(Canvas c) { ... }
}

public class Canvas {
  public void draw(Shape s) { s.draw(this);}
}
```



Goal: Write code to draw a list of any kind of shape →

Lecture: Bounded polymorhpism - Wildcards

```
// in class Canvas:
public void drawAll(List<Shape> shapes) {
  for (Shape s: shapes)
     s.draw(this);
}
```

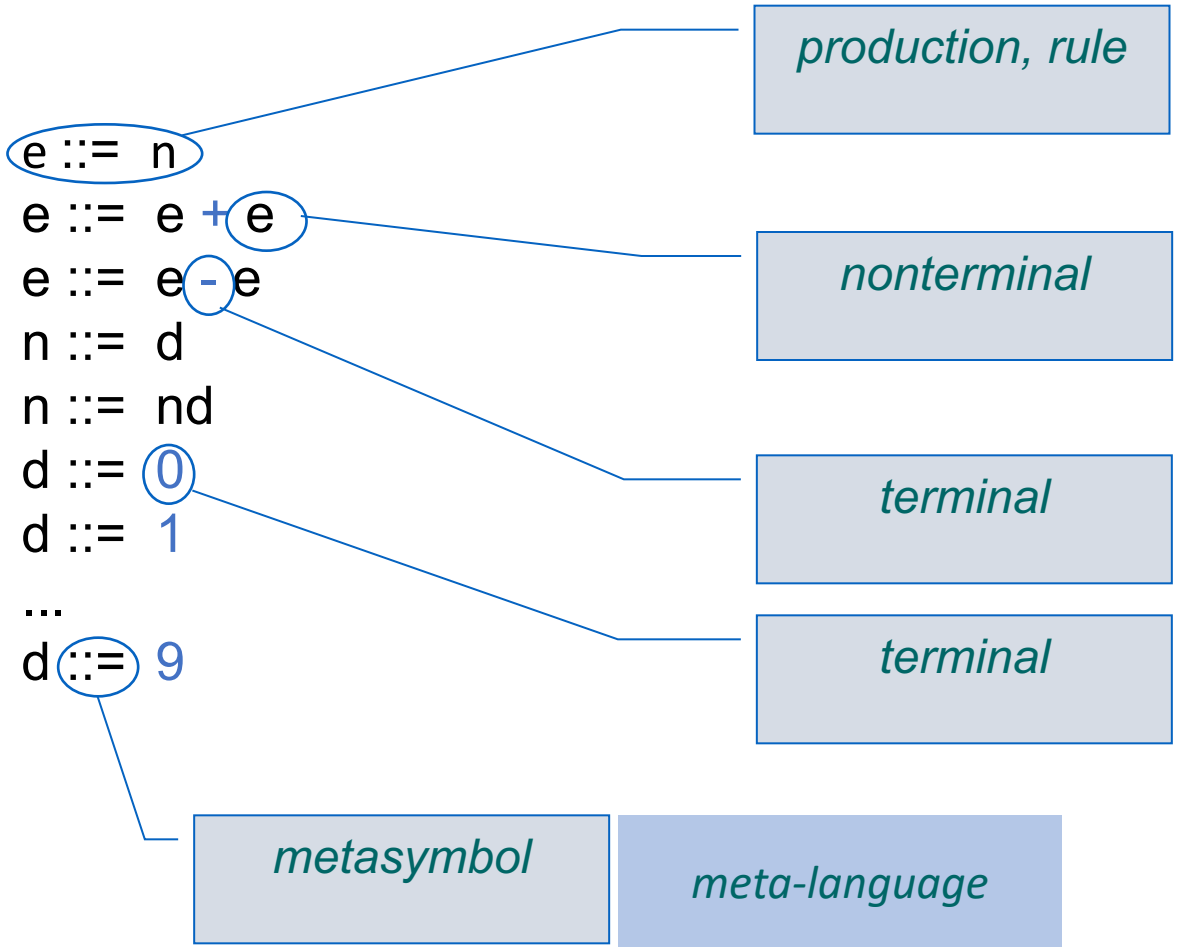
Cannot draw e.g. a list of rectangles or circles

```
public void drawAll(List<? extends Shape> shapes) {
  for (Shape s: shapes)
     s.draw(this);
}
```

- `List<S>` subtype of `List<? extends Shape >` for every `S` being a subtype of the (concrete) type `Shape`

- `List<S>` subtype of `List<? extends T >` for every `S` being a subtype of (the generic parameter) `T`

Syntax: described by BNF grammars



Terminals are found in the program text

Non-terminals are not

Extended BNF

- In Extended BNF (eBNF) we can use the following *metasymbols* on the righthand side:

|	alternatives
[…]	optionality (0 or 1 time)
*	zero or more times (from regular expressions – alternatively {...})
+	one or more times (from regular expressions)
(…)	grouping symbols (sometimes {...} is used)

Grammar from previous slide expressed more concisely with eBNF

```
e ::=  n | e + e | e - e
n ::=  d | nd
d ::=  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Task 1e - answer

The key point here is to make a grammar that allows for extends-relations and super-relations on the generic parameters.

The super-relations will be used for contravariant definitions, e.g. f<T super SomeStruct>(T t) { … }, and correspondingly for extends and covariant definitions.

```
function-def ::= <type-identifier> <identifier>
                 [ <generic-params> ]
                "(" [ <param-list> ] ")"
                "{" <func-body> "}"

generic-params ::= "<" <generic-param>
                   ("," <generic-param> )* ">"

generic-param ::= <identifier>
                  [ ("super" | "extends") <identifier> ]
```

Task 1f

In this question, we will allow structs to be generic, and use this to define function parameters. We will assume that there is a predefined struct called Func which has two generic parameters, representing a function's (single) formal parameter type (U), and return type (T), respectively. Example:

struct Func of T and U { … }

We will in this question use this struct to define formal function parameters to other functions. With this in mind, consider the following program:

```
int f( (Func of int and int) g) {
    int x = g(42);
    return x;
}

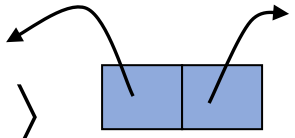
int dummy(int val) { return val; }

void Main() {
    f(dummy);
}
```

Draw the runtime stack for the program above, right before the line "return val" is executed in the function dummy.

Lecture: Closures

- Function value is pair *closure* = ⟨*env*, *code* ⟩



- A closure “closes” around its environment
  - A function value needs to take its environment along:

```
var createGreeter = function() {
    var msg = "Hello World";

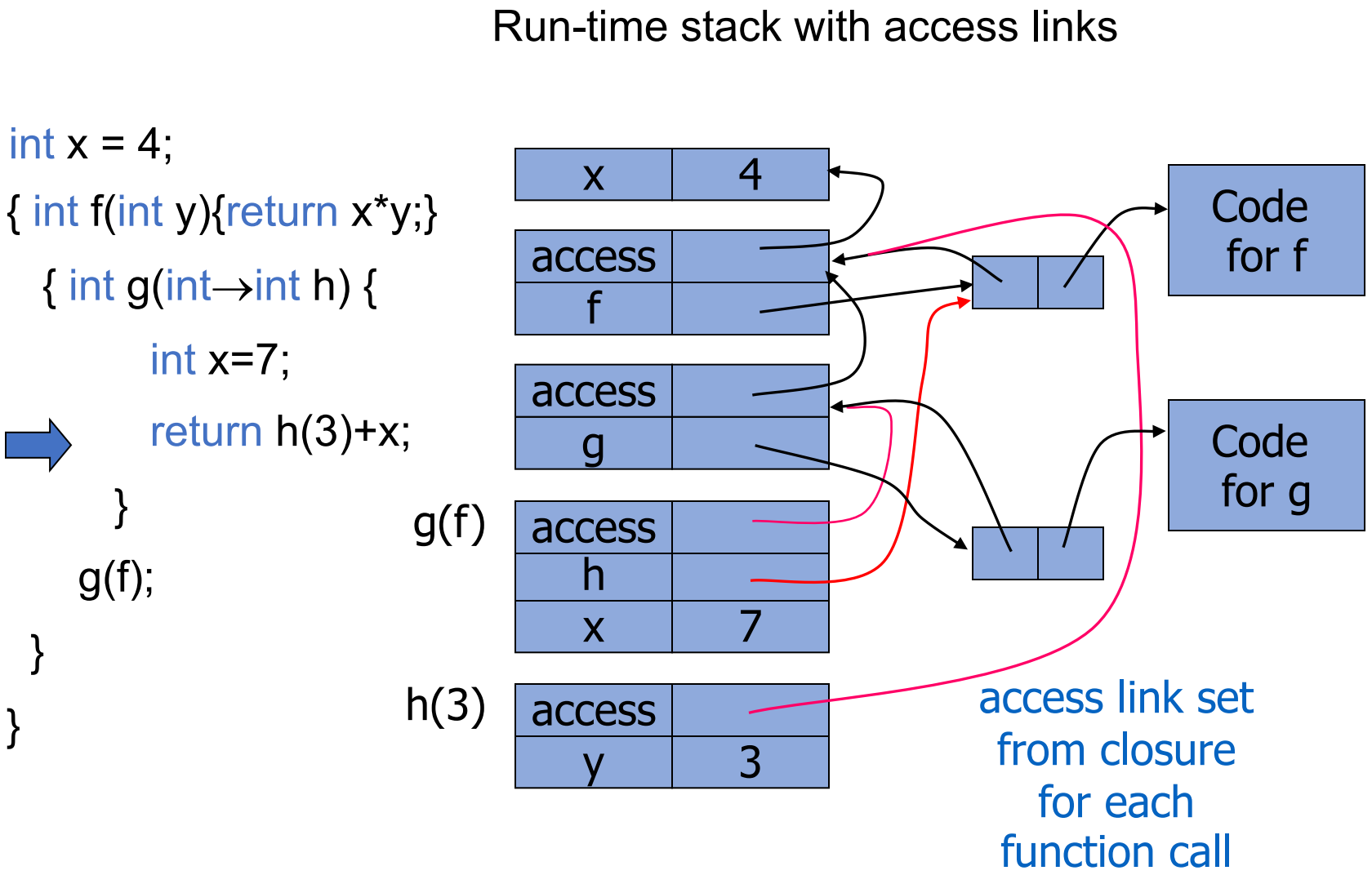
    var greeter = function() {
        console.log(msg);
    }

    return greeter;
}
```

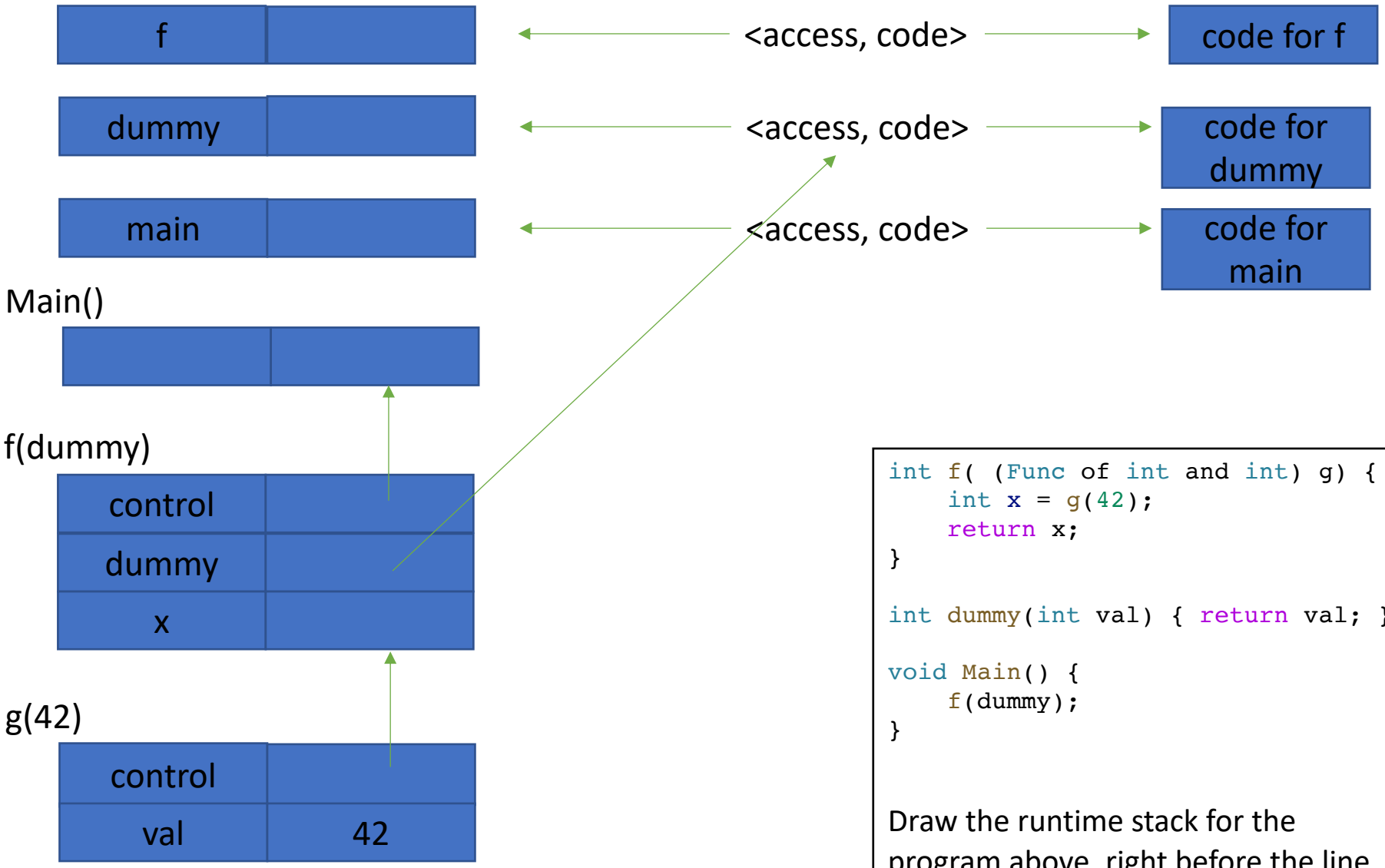
- When a function represented by a closure is called
  - Allocate activation record for call (as always)
  - Set the access link in the activation record using the environment pointer from the closure

Function Argument and Closures

```
{ int x = 4;
  { int f(int y){return x*y;}
    { int g(int→int h) {
          int x=7;
          return h(3)+x;
      }
    g(f);
  }
}
```



1f Solution




```
int f( (Func of int and int) g) {
    int x = g(42);
    return x;
}

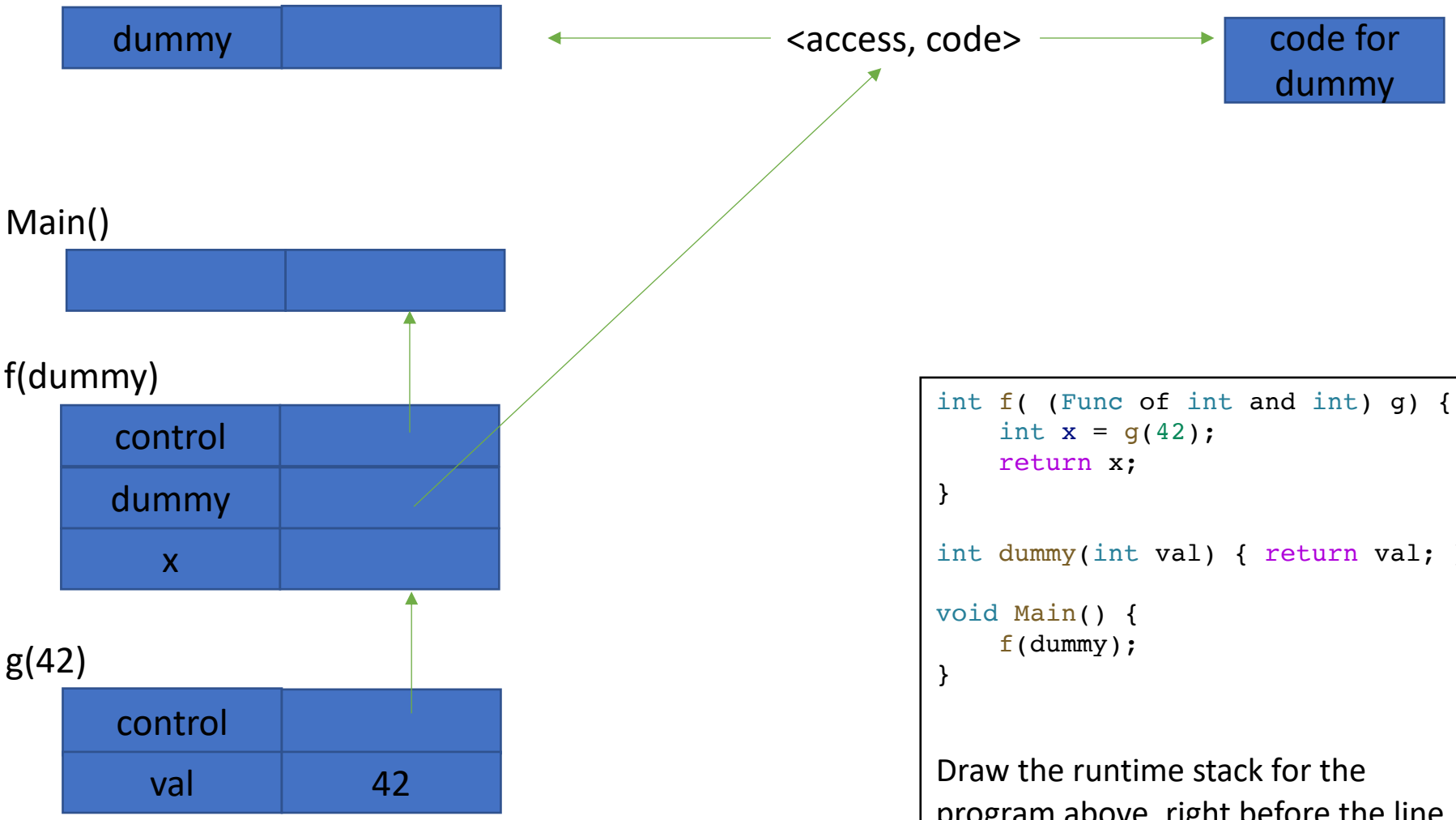
int dummy(int val) { return val; }

void Main() {
    f(dummy);
}
```

Draw the runtime stack for the program above, right before the line "return val" is executed in the function dummy.

1f Solution

DISCLAIMER: This is a simplified drawing, that skips func decl blocks – OK in this case, but not always




```
int f( (Func of int and int) g) {
    int x = g(42);
    return x;
}

int dummy(int val) { return val; }

void Main() {
    f(dummy);
}
```

Draw the runtime stack for the program above, right before the line "return val" is executed in the function dummy.

Summing up

- The upcoming exam:
  - December 14, 4 hours at Silurveien 2
  - All written and printed material allowed
- Some main topics from today's repeption/exam task
  - Runtime stacks and activation records
  - Dynamic vs static scope
  - Generics, variance
  - Functions/methods as parameters
  - EBNF
- Remember: there are more topics in this course!
  - Syntax/semantics in general – don't take specific semantics for granted!
  - (Abstract) syntax/parse trees
  - Scoping rules
  - Object orientation, multiple inheritance, virtual classes
  - And of course SML/Haskell, Prolog and all the rest of Volker's lectures
- Thank you, and GOOD LUCK!
  - Feel free to email me questions – eyvinda@ifi.uio.no

