



# INF3110 – Repetition / Exam from 2017

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

Eyvind W. Axelsen

[eyvinda@ifi.uio.no](mailto:eyvinda@ifi.uio.no) | [@eyvindwa](https://twitter.com/eyvindwa) 

<http://eyvinda.at.ifi.uio.no>

# Today

- Part 1: Runtime systems, scoping and types
  - Exam 2017 task 1, with repetition of central points in order to solve it
  - Questions/answers
- Part 2: SML/Prolog
  - Daniel could unfortunately not make it today
  - Morten to the rescue!



# 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!  
}
```

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!  
}
```

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.

## Task 1c

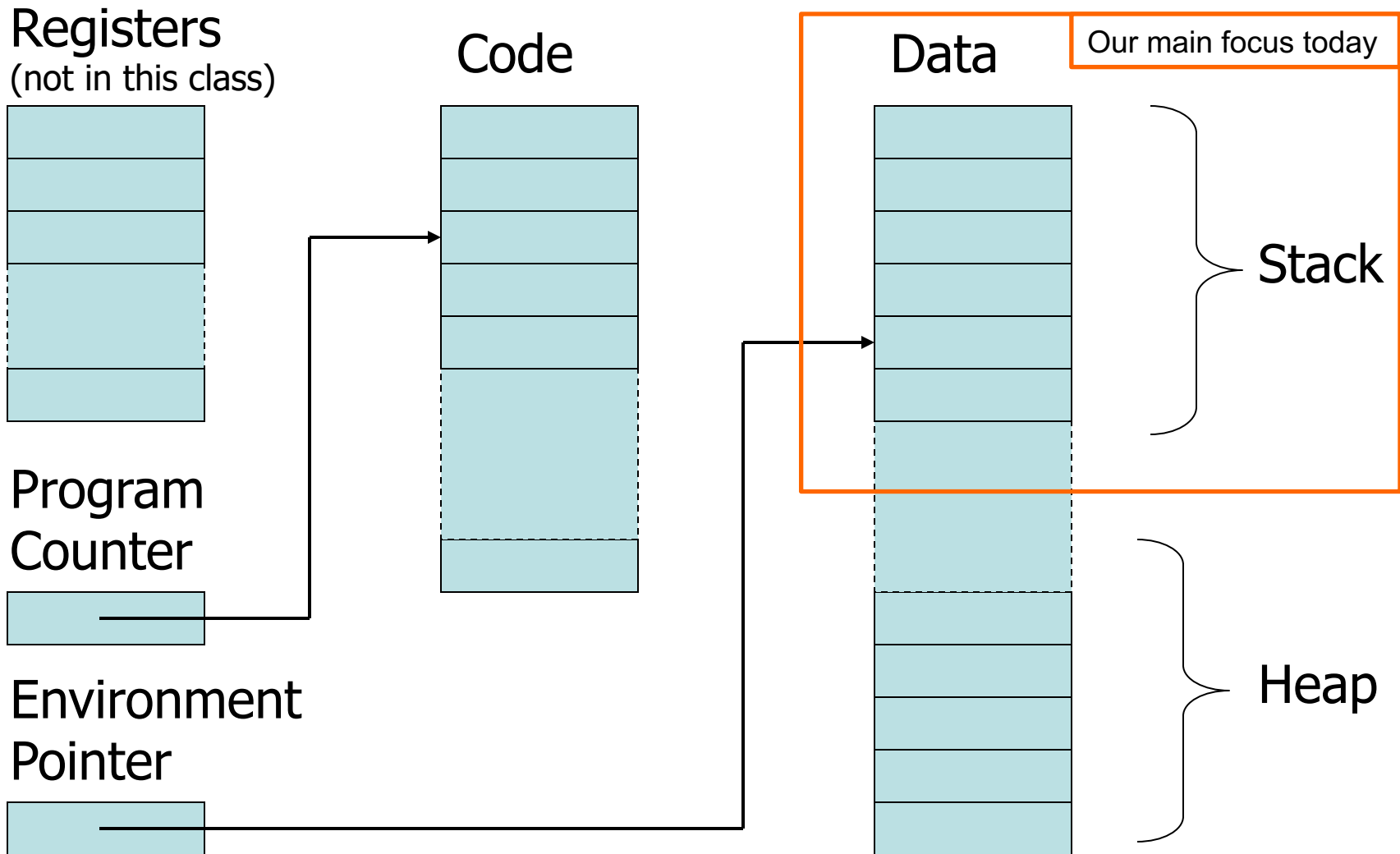
```
interface I {  
    int function1();  
}  
  
struct A {  
    int function1() {  
        int x = 21;  
        int y = x +  
            function2();  
        return y;  
    }  
    int function2() {  
        return x ;  
    }  
}
```

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

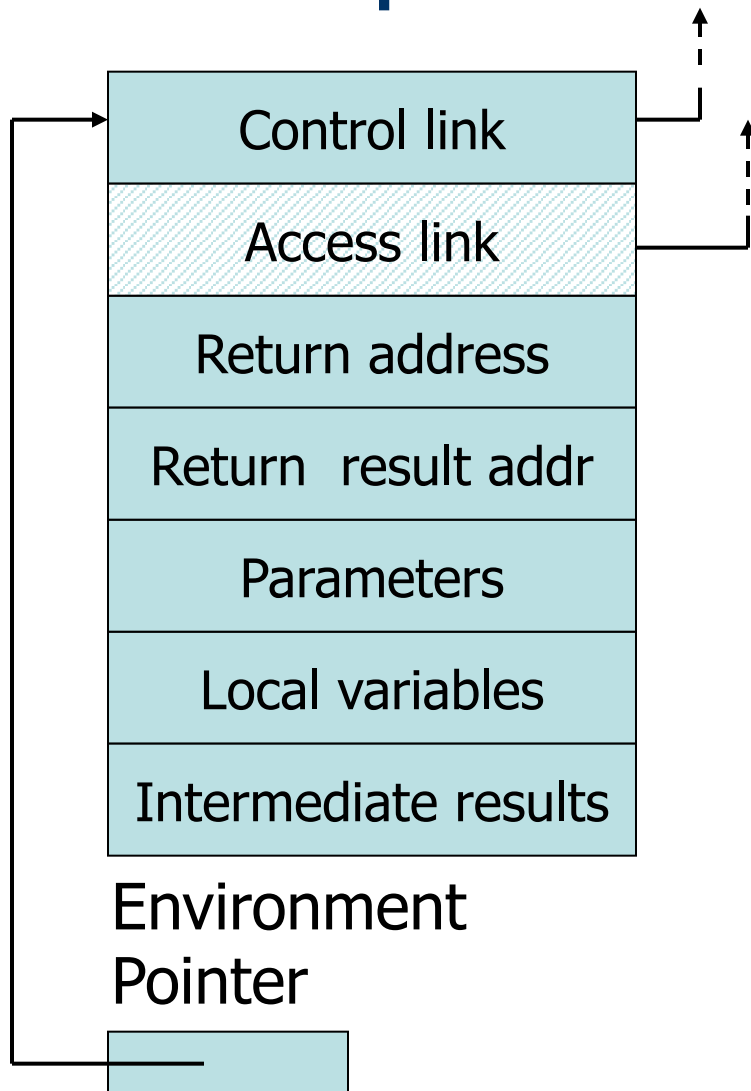
```
void Main() {  
    I myI = new A();  
    int result = getValue(myI);  
  
    myI = new B();  
    result = result + getValue(myI);  
    printResult();  
}  
  
int getValue(I i) {  
    return i.function1();  
}  
  
void printResult() {  
    print("The result is: ");  
    print(result); // HERE!  
}
```

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 Runtime org. 1 - Simplified Reference Model of a Machine - used to understand memory management



# Lecture Runtime org. 1 - 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 Runtime org. 1 - Static scope with access links (C-like notation)

```
{
  int x = 1;

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

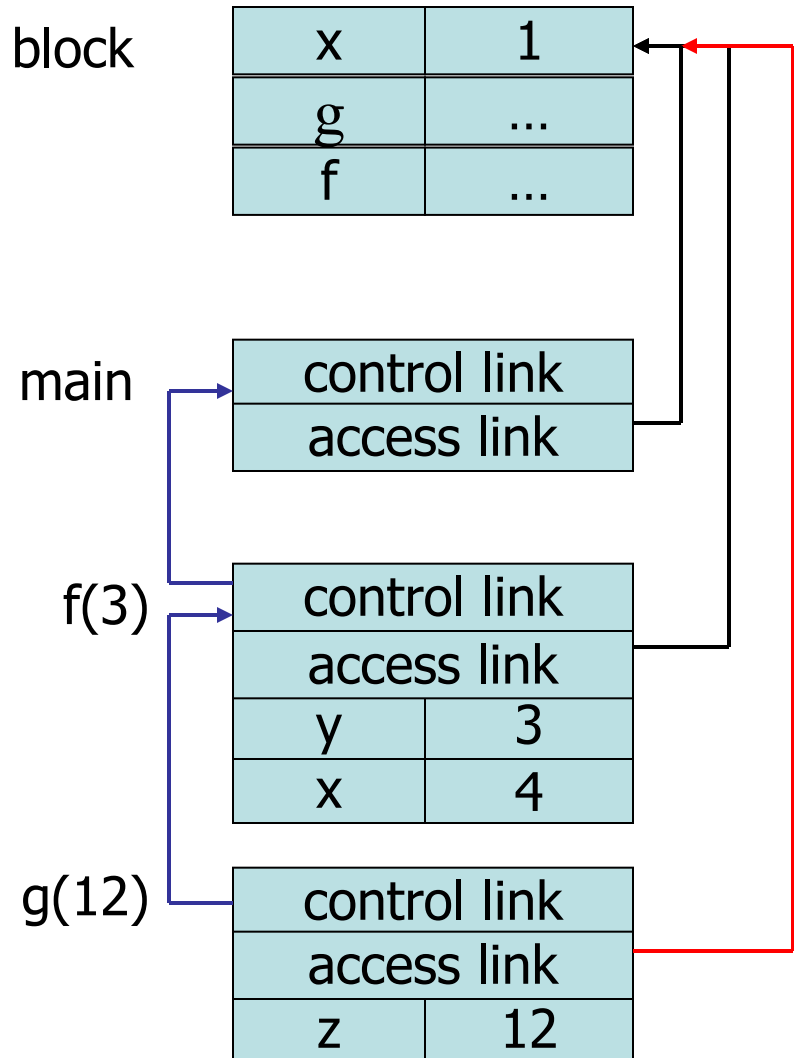
  int function f(y) {
    int x = y+1;
    return g(y*x)
  };

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

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

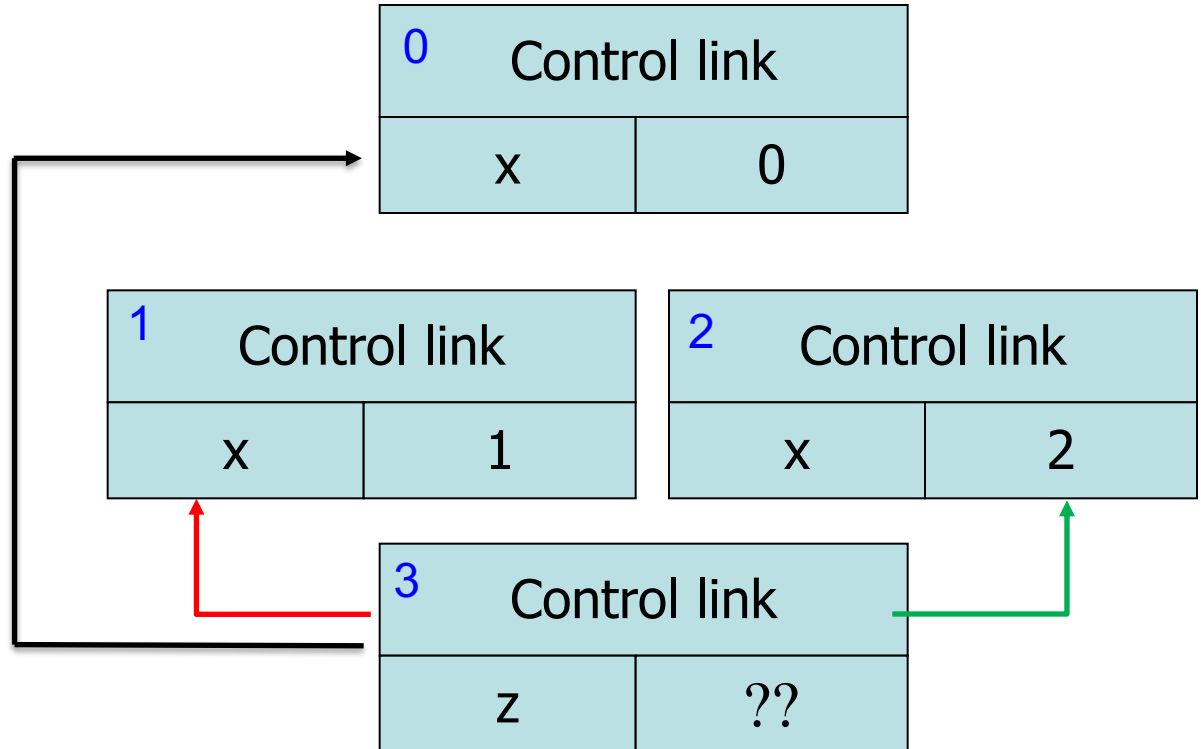
outer block



# Example – static/dynamic scoping

```

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

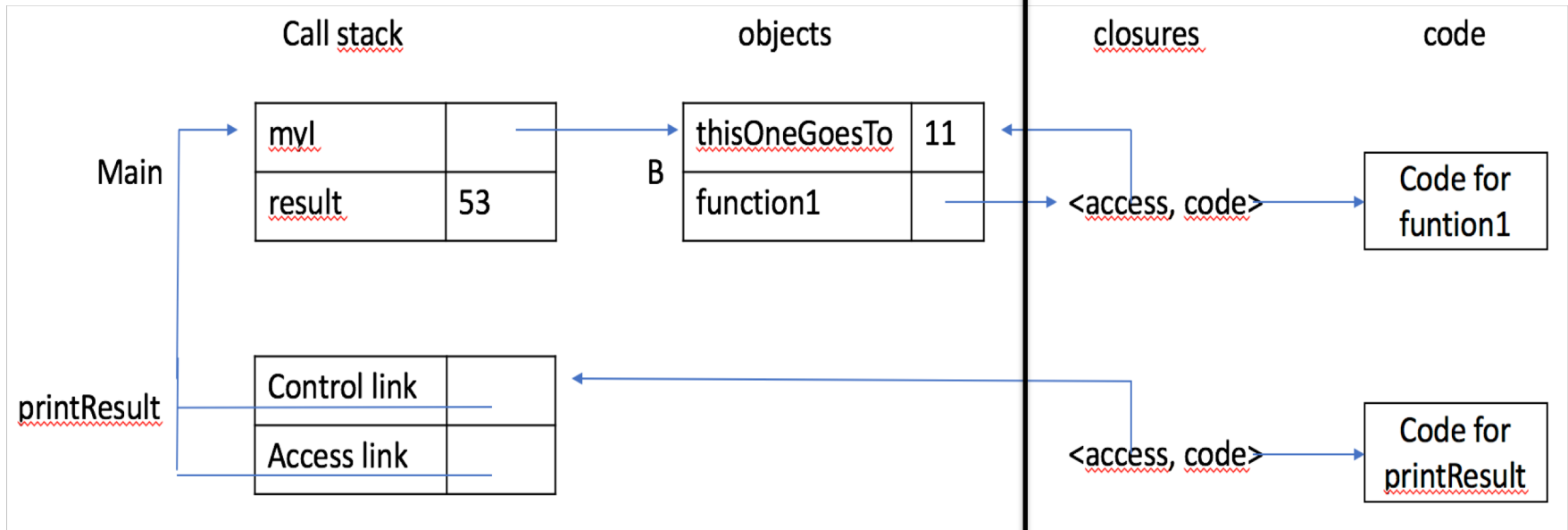


If block 0 executes block 3  
 Static scope: z = ?  
 Dynamic scope: z = ?

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

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

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.

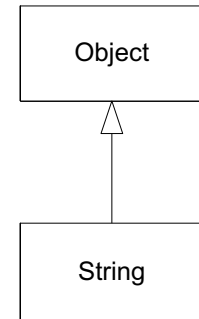
# Lecture OO II - Generics and subtyping

- 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

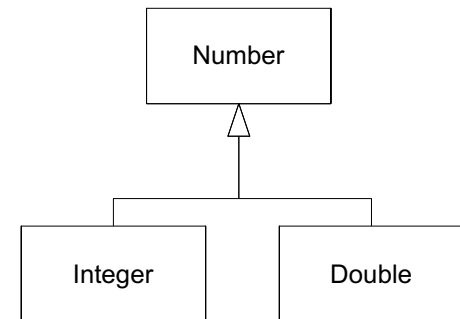
attempts to assign  
an Object to a String



- 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 OO II - Bounded polymorphism - Wildcards - II

```
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); }  
}
```

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

# Lecture OO II - Bounded polymorphism - Wildcards - III

```
// in class Canvas:
```

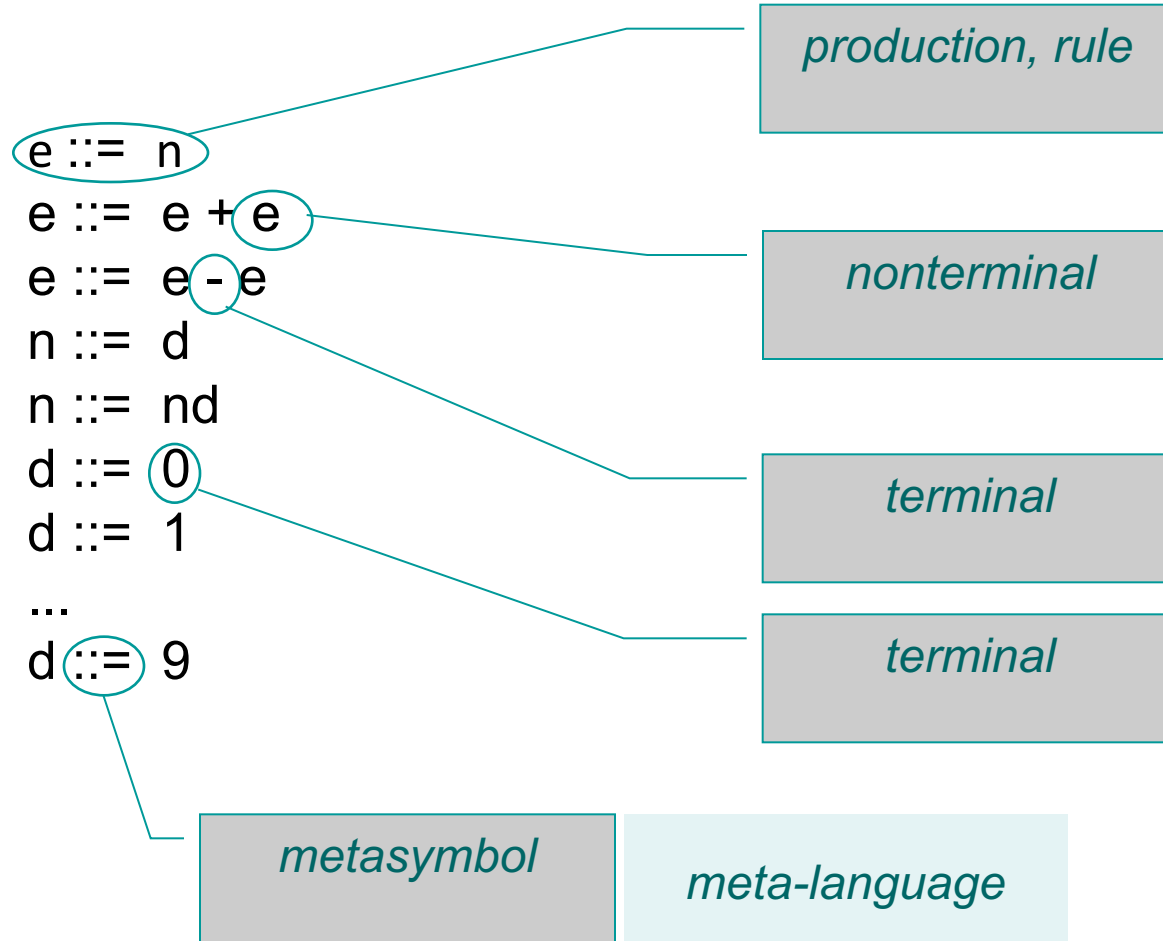
```
public void drawAll(List<Shape> shapes) {  
    for (Shape s: shapes)  
        s.draw(this);  
}
```

```
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`

# Lecture: Syntax and Semantics

## Syntax: Described by BNF-grammars



*Terminals* are found in the program text

*Non-terminals* are not

# Lecture: Syntax and Semantics

## 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`.

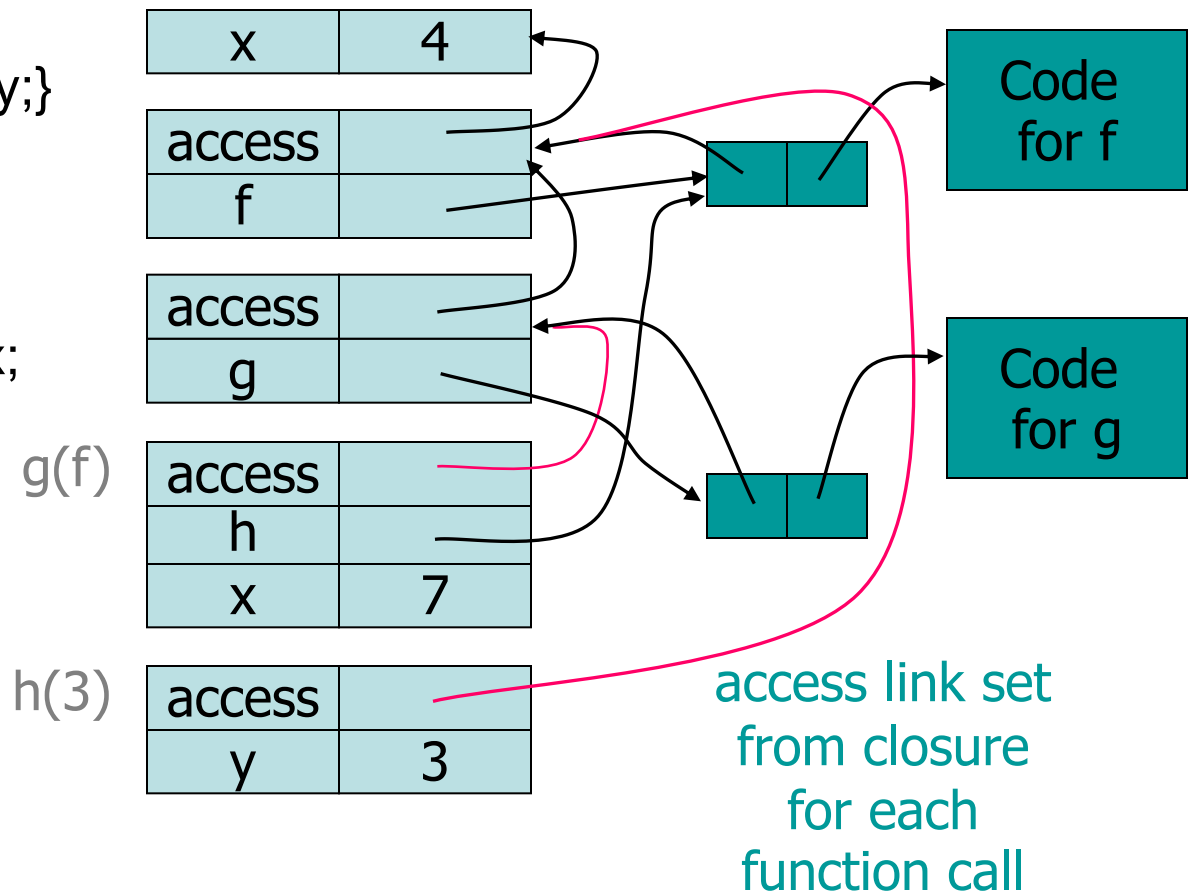
# Lecture 00 II - Closures

- Function value is pair *closure* =  $\langle env, code \rangle$
- 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

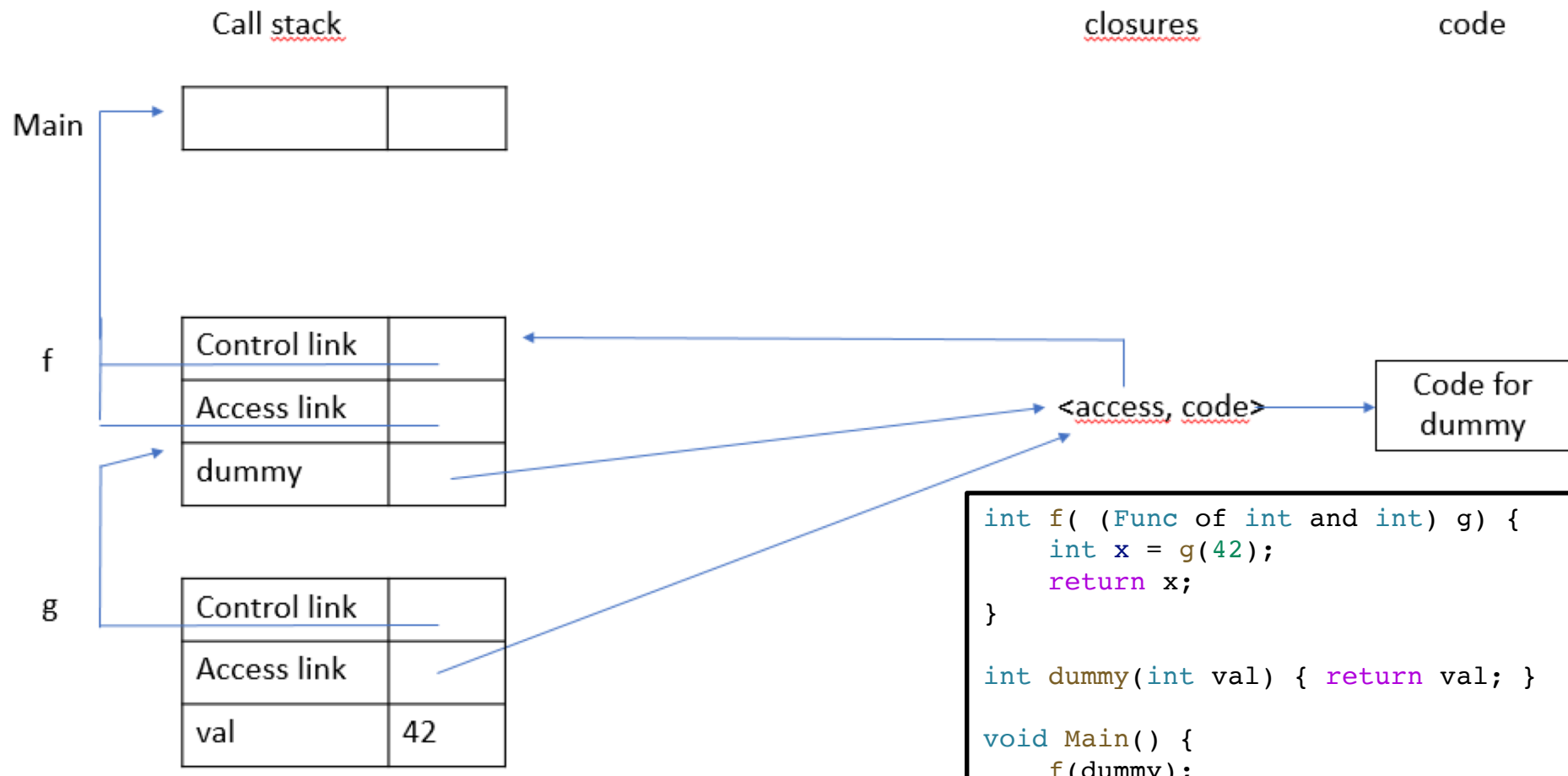
# Lecture OO II - Function Argument and Closures

Run-time stack with access links

```
{ 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.

# Summing up

- The upcoming exam:
  - November 28, 09:00 (4 hours).
  - All written and printed material allowed
    - That includes the textbook and slides from the lectures, as well as your own notes
  - Digital exam – check your logon to Inspera the day before!
- Some main topics of today's (2017) 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, Prolog and all the rest of Daniel's lectures (coming up!)
  - More exams on the course page – take a look at them!
- Thank you, and GOOD LUCK!
  - Feel free to email me questions – [eyvinda@ifi.uio.no](mailto:eyvinda@ifi.uio.no)