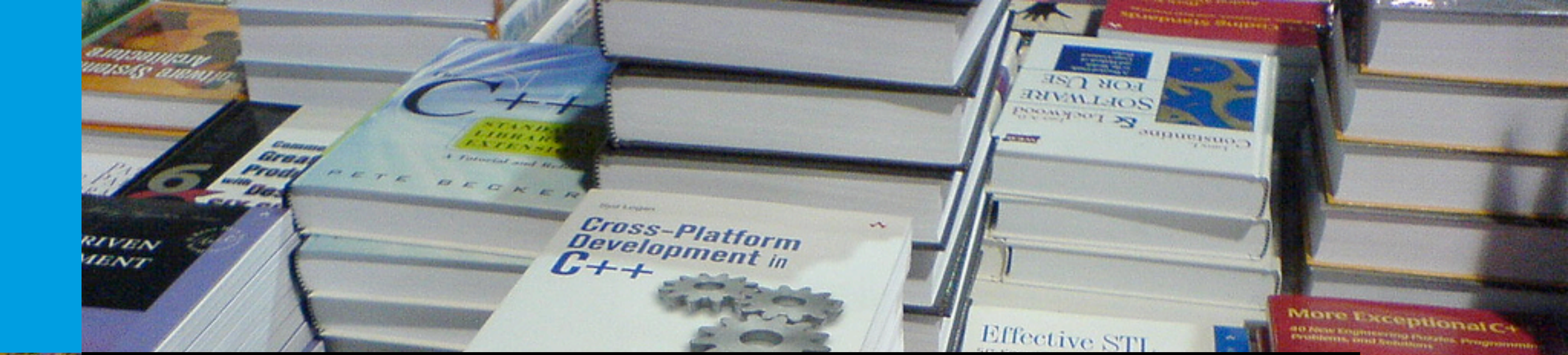
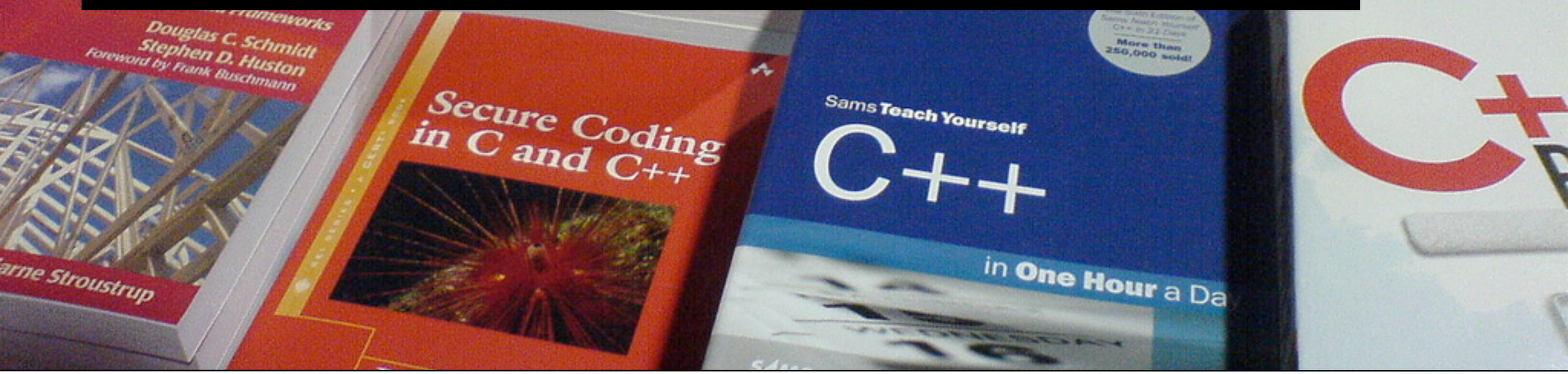


Imperative and Object-Oriented Languages

Guido Wachsmuth



Assessment

last lecture

Explain the properties of language using the examples of English and MiniJava.

- arbitrary
- symbolic
- systematic
- productive
- non-instinctive
- conventional
- modifiable

Assessment
last lecture

What is a software language?

- computer-processable artificial language used to engineer software
- piece of software

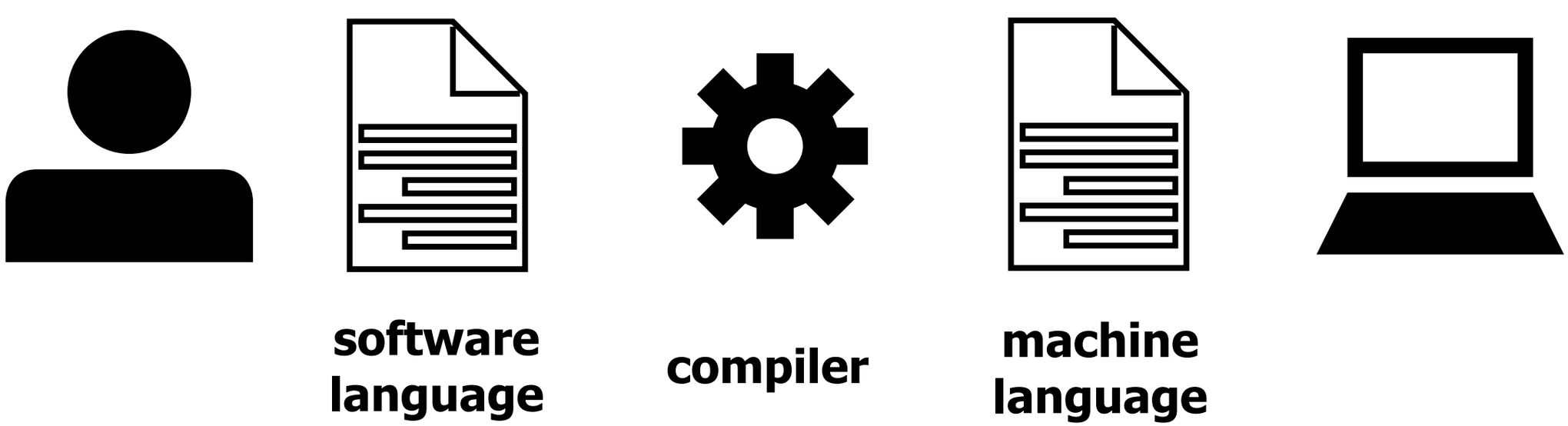
Why is MiniJava a software language?

- computer-processable artificial language
- programming language, can be used to engineer software
- MiniJava compiler = piece of software

Why is English not a software language?

- not computer-processable, not artificial

Recap: Compilers



Overview
today's lecture

I
imperative languages

II
object-oriented languages

I

imperative languages

machine code

x86 family
registers

general purpose registers

- accumulator AX - arithmetic operations
- counter CX - shift/rotate instructions, loops
- data DX - arithmetic operations, I/O
- base BX - pointer to data
- stack pointer SP, base pointer BP - top and base of stack
- source SI, destination DI - stream operations

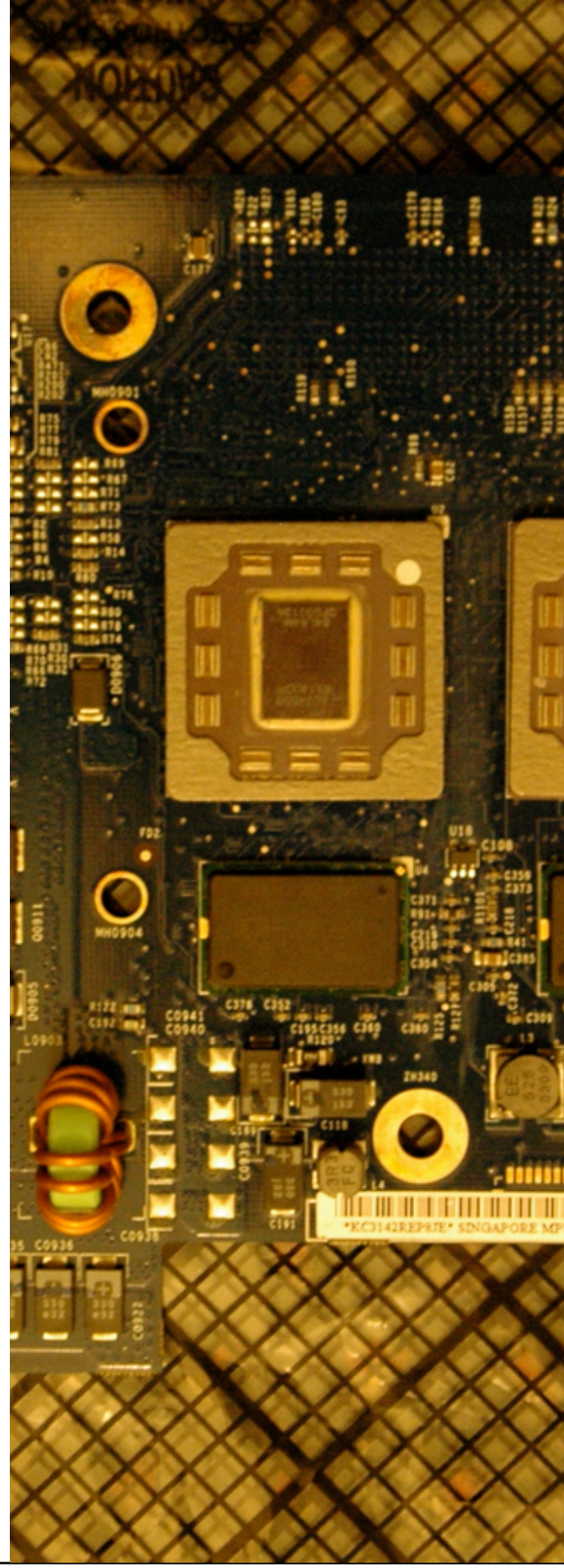
special purpose registers

- segments SS, CS, DS, ES, FS, GS
- flags EFLAGS

Example: x86 Assembler

basic concepts

```
    mov AX [1]          read memory

    mov CX AX

L:  dec CX
    mul CX              calculation
    cmp CX 1
    ja  L               jump

    mov [2] AX          write memory
```



Example: Java Bytecode

basic concepts

```
.method static public m(I)I

        iload 1
        ifne else                    jump
        iconst_1
        ireturn

else:   iload 1                      read memory
        dup
        iconst_1
        isub                         calculation
        invokestatic Math/m(I)I
        imul
        ireturn
```

Example: C

states & statements

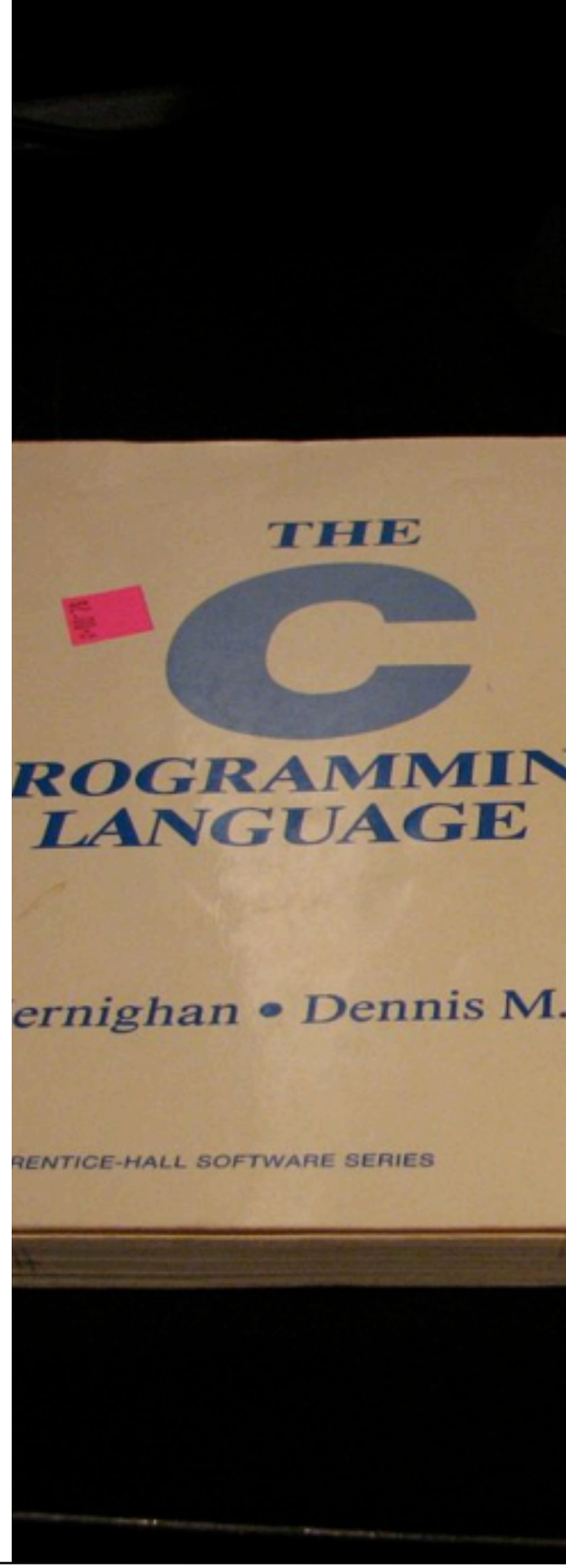
```
int f = 1
int x = 5
int s = f + x

while (x > 1) {
   f = x * f ;
   x = x - 1
}
```

variable

expression

control flow

assignment



Example: Tiger

states & statements

```
/* factorial function */

let
   var f := 1                    variable
   var x := 5
   var s := f + x                expression
in
   while x > 1 do (              control flow
      f := x * f ;
      x := x - 1                 assignment
   )
end
```

Example: x86 Assembler

modularity

```
push 21
push 42
call _f
add  SP 8
```

pass parameter

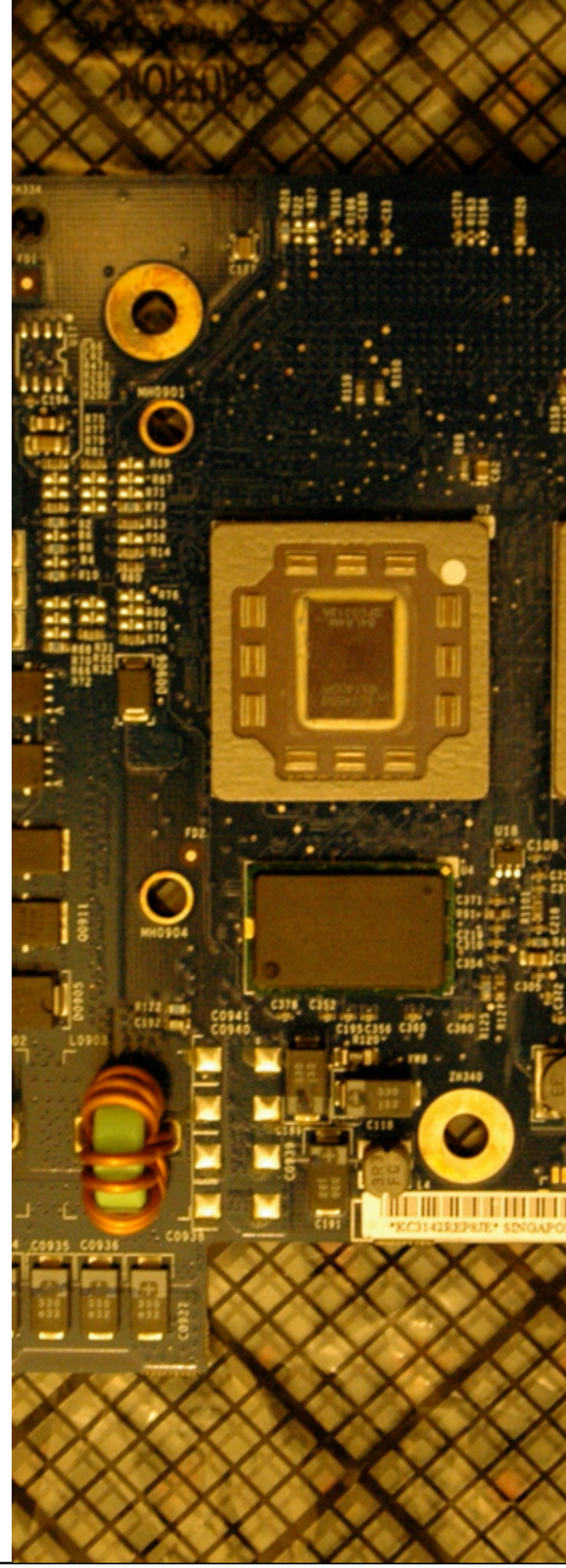
free parameters

```
push BP
mov  BP SP
mov  AX [BP + 8]
mov  DX [BP + 12]
add  AX DX
pop  BP
ret
```

new stack frame

access parameter

old stack frame



Example: C
procedures

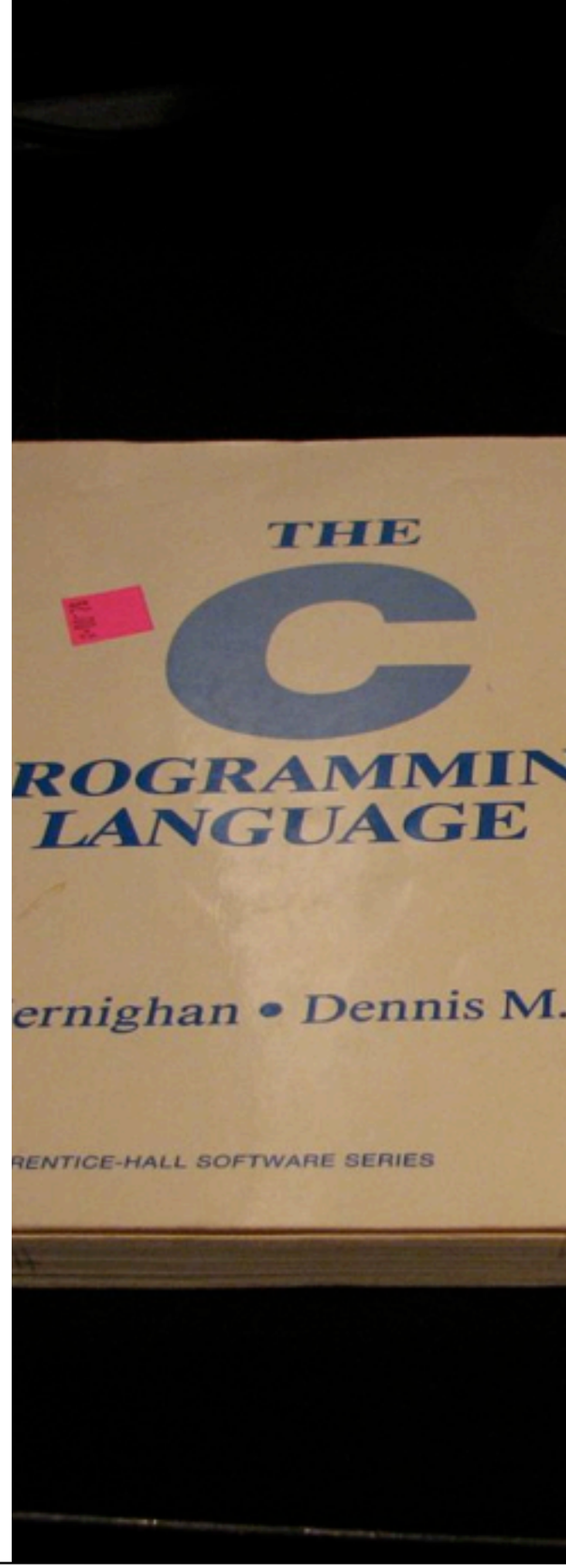
```c
#include <stio.h>

/* factorial function */
int fac( int num ) {
    if (num < 1)
        return 1;
    else
        return num * fac(num - 1) ;
}

int main() {
    int x = 10 ;

    int f = fac( x ) ;
    int x printf(“%d! = %d\n”, x, f);
    return 0;
}
```

formal parameter

recursive call

local variable

actual parameter



Example: Tiger
procedures

```
/* factorial function */

let
    function fac( n: int ) : int =
        let
            var f := 1
        in
            if n < 1 then
                f := 1
            else
                f := (n * fac(n - 1) );
            f
        end
    var f := 0
    var x := 5
in
    f := fac( x )
end
```

formal parameter

local variable

recursive call

actual parameter

Example: Tiger

call by value vs. call by reference

```
let
    type vector = array of int

    function init(v: vector) =
        v := vector[5] of 0

    function upto(v: vector, l: int) =
        for i := 0 to l do
            v[i] := i

    var v : vector := vector[5] of 1
in
    init(v) ;
    upto(v, 5)
end
```

Type Systems
dynamic & static typing

machine code

- memory: no type information
- instructions: assume values of certain types

dynamically typed languages

- typed values
- run-time checking & run-time errors

statically typed languages

- typed expressions
- compile-time checking & compile-time errors

Type Systems
compatibility

type compatibility

- value/expression: actual type
- context: expected type

type equivalence

- structural type systems
- nominative type systems

subtyping

- relation between types
- value/expression: multiple types

Example: Tiger

type compatibility

```
let
type A = int
type B = int
type V = array of A
type W = V
type X = array of A
type Y = array of B

    var a: A := 42
    var b: B := a
    var v: V := V[42] of b
    var w: W := v
    var x: X := w
    var y: Y := x
in
    y
end
```

Type Systems
record types

record

- consecutively stored values
- fields accessible via different offsets

record type

- fields by name, type, position in record
- structural subtyping: width vs. depth

```
type R1 = {f1 : int, f2 : int}
type R2 = {f1 : int, f2 : int, f3 : int}
type R3 = {f1 : byte, f2 : byte}
```

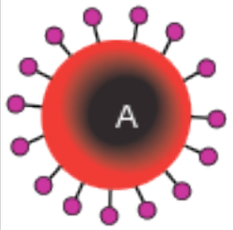
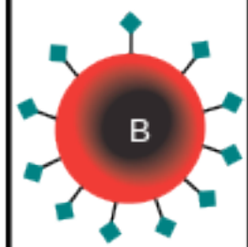
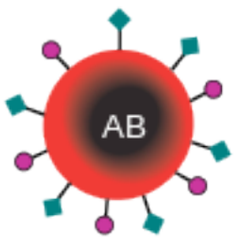
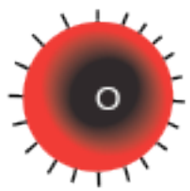
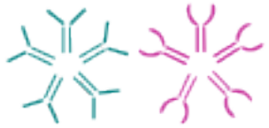
Polymorphism

biology

the occurrence of more than one form

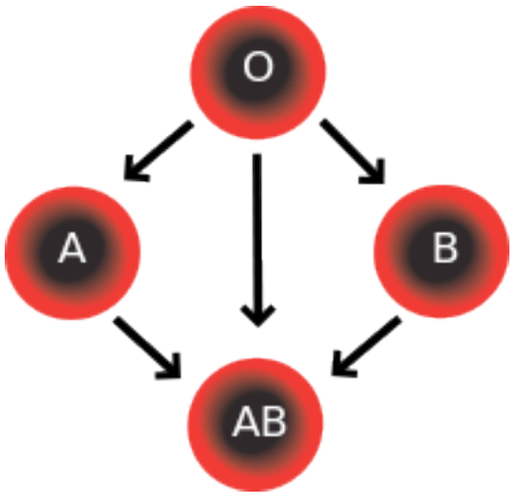
Polymorphism

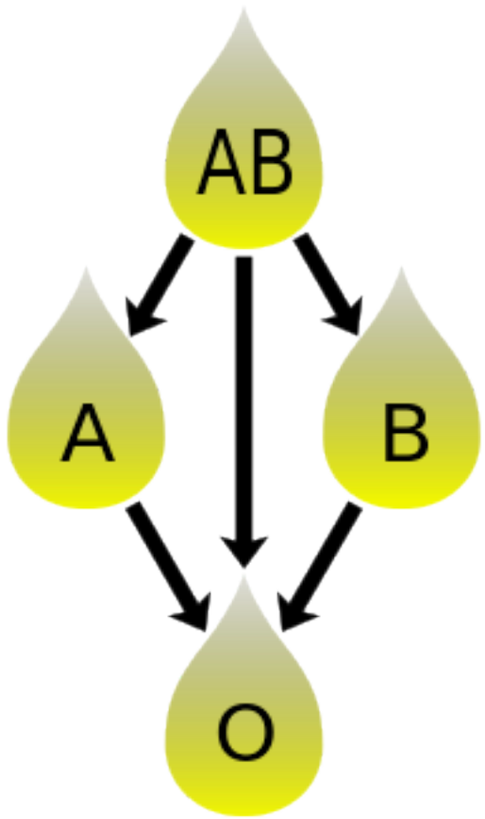
biology

	Group A	Group B	Group AB	Group O
Red blood cell type				
Antibodies in Plasma	 Anti-B	 Anti-A	None	 Anti-A and Anti-B
Antigens in Red Blood Cell	A antigen	B antigen	 A and B antigens	None

Polymorphism

biology





Polymorphism

programming languages

```
21 + 21
```

```
21.0 + 21.0
```

```
"foo" + "bar"
```

Polymorphism

programming languages

```
print(42)

print(42.0)

print("foo")
```

Polymorphism

programming languages

```
21 + 21
```

```
21.0 + 21.0
```

```
21 + 21.0
```

```
21 + "bar"
```

Type Systems
polymorphism

ad-hoc polymorphism

overloading

- same name, different types, same operation
- same name, different types, different operations

type coercion

- implicit conversion

universal polymorphism

subtype polymorphism

- substitution principle

parametric polymorphism

```
21 + 21
21.0 + 21.0

print(42)
print(42.0)

"foo" + "bar"

21 + "bar"
```

coffee break

II

object-oriented languages

Modularity

objects & messages

objects

- generalisation of records
- identity
- state
- behaviour

messages

- objects send and receive messages
- trigger behaviour
- imperative realisation: method calls

Modularity
classes

classes

- generalisation of record types
- characteristics of objects: attributes, fields, properties
- behaviour of objects: methods, operations, features

encapsulation

- interface exposure
- hide attributes & methods
- hide implementation

```
public class C {
   public int f1;
   private int f2;
   public void m1() { return; }
   private C m2(C c) { return c; }
}
```

Modularity

inheritance vs. interfaces

inheritance

- inherit attributes & methods
- additional attributes & methods
- override behaviour
- nominative subtyping

interfaces

- avoid multiple inheritance
- interface: contract for attributes & methods
- class: provide attributes & methods
- nominative subtyping

```
public class C {
   public int f1;
   public void m1() {…}
   public void m2() {…}
}

public class D extends C {
   public int f2;
   public void m2() {…}
   public void m3() {…}
}

public interface I {
   public int f;
   public void m();
}

public class E implements I {
   public int f;
   public void m() {…}
   public void m'() {…}
}
```

Type Systems
polymorphism

ad-hoc polymorphism

overloading

- same method name, independent classes
- same method name, same class, different parameter types

overriding

- same method name, subclass, compatible types

universal polymorphism

subtype polymorphism

- inheritance, interfaces

parametric polymorphism

Type Systems

static vs. dynamic dispatch

dispatch

- link method call to method

static dispatch

- type information at compile-time

dynamic dispatch

- type information at run-time
- single dispatch: one parameter
- multiple dispatch: more parameters

Example: Java

single dispatch

```
public class A {} public class B extends A {} public class C extends B {}

public class D {
   public A m(A a) { System.out.println("D.m(A a)"); return a; }
   public A m(B b) { System.out.println("D.m(B b)"); return b; }
}

public class E extends D {
   public A m(A a) { System.out.println("E.m(A a)"); return a; }
   public B m(B b) { System.out.println("E.m(B b)"); return b; }
}
```

```
A a = new A(); B b  = new B(); C c  = new C(); D d = new D(); E e  = new E();
               A ab = b;       A ac = c;                      D de = e;

d. m(a); d. m(b); d. m(ab); d. m(c); d. m(ac);
e. m(a); e. m(b); e. m(ab); e. m(c); e. m(ac);
de.m(a); de.m(b); de.m(ab); de.m(c); de.m(ac);
```

Type Systems

variance

A <: B

C **?** D

Type Systems

covariance

A <: B

C <: D

Type Systems

contravariance

A <: B

C :> D

Type Systems

invariance

$A <: B$

$C = D$

Type Systems
overriding

methods

- parameter types
- return type

covariance

- method in subclass
- return type: subtype of original return type

contravariance

- method in subclass
- parameter types: supertypes of original parameter types

Example: Java

overloading vs. overriding

```
public class F {
   public A m(B b) { System.out.println("F.m(B b)"); return b; }
}

public class G extends F {
   public A m(A a) { System.out.println("G.m(A a)"); return a; }
}

public class H extends F {
   public B m(B b) { System.out.println("H.m(B b)"); return b; }
}
```

```
A a = new A(); B b  = new B(); F f = new F(); G g = new G(); H h = new H();
               A ab = b;

f.m(b);
g.m(a); g.m(b); g.m(ab);
h.m(a); h.m(b); h.m(ab);
```

Example: Java

invariance

```
public class X {
   public A a;
   public A    getA()    { return a ; }
   public void setA(A a) { this.a = a ; }
}

public class Y extends X {
   public B a;
   public B    getA()    { return a ; }
   public void setA(B a) { this.a = a ; }
}
```

```
A a = new A(); B b  = new B(); X y = new Y();

y.getA(); y.setA(b); y.setA(a);

String[] s = new String[3] ; Object[] o = s ; o[1] = new A();
```

III

summary

Summary
lessons learned

imperative languages

- state & statements
- abstraction over machine code
- control flow & procedures
- types

object-oriented languages

- objects & messages
- classes
- inheritance
- types

Literature

learn more

Imperative Languages

Carl A. Gunter: Semantics of Programming Languages: Structures and Techniques. MIT Press, 1992

Kenneth C. Louden: Programming Languages: Principles and Practice. Course Technology, 2002

Object-Oriented Languages

Martin Abadi, Luca Cardelli: A Theory of Objects. Springer, 1996.

Kim B. Bruce: Foundations of Object-Oriented Programming Languages: Types and Semantics. MIT Press, 2002.

Timothy Budd: An Introduction to Object-Oriented Programming. Addison-Wesley, 2002.

Outlook

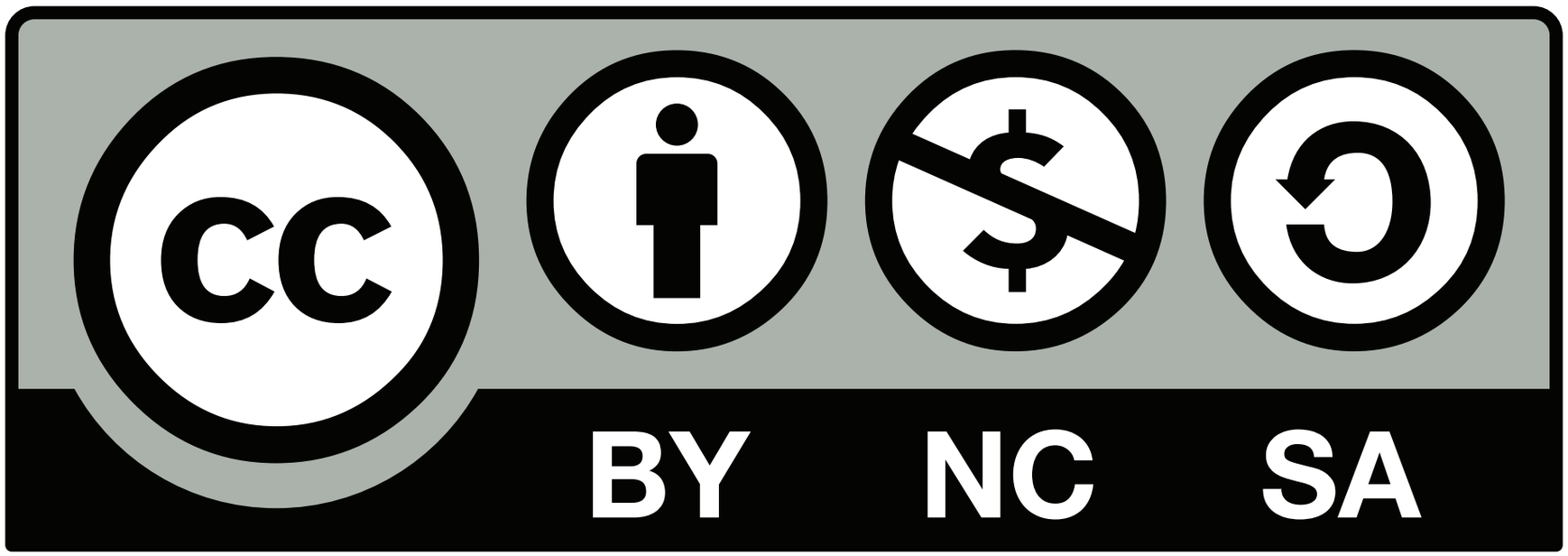
coming next

declarative language definition

- Lecture 1: Grammars and Trees
- Lecture 2: SDF and ATerms
- Lecture 3: Name Binding and Type Systems
- Lecture 4: Term Rewriting
- Lecture 5: Static Analysis and Error Checking
- Lecture 6: Code Generation

Lab Sep 12

- get used to Eclipse, Spoofax, and MiniJava


BY
NC
SA

Pictures

copyrights

Slide 1: Popular C++ by Itkovian, some rights reserved

Slide 4: PICOL icons by Melih Bilgil, some rights reserved

Slides 7, 9, 13: Dual Processor Module by roobarb!, some rights reserved

Slides 11, 14: The C Programming Language by Bill Bradford, some rights reserved

Slides 12, 15, 16, 19: Tiger by Bernard Landgraf, some rights reserved

Slide 21: Adam and Eva by Albrecht Dürer, public domain

Slide 22: ABO blood type by InvictaHOG, public domain

Slide 23: Blood Compability and Plasma donation compatibility path by InvictaHOG, public domain

Slide 28: Delice de France by Dominica Williamson, some rights reserved

Slide 46: Nieuwe Kerk by Arne Kuilman, some rights reserved