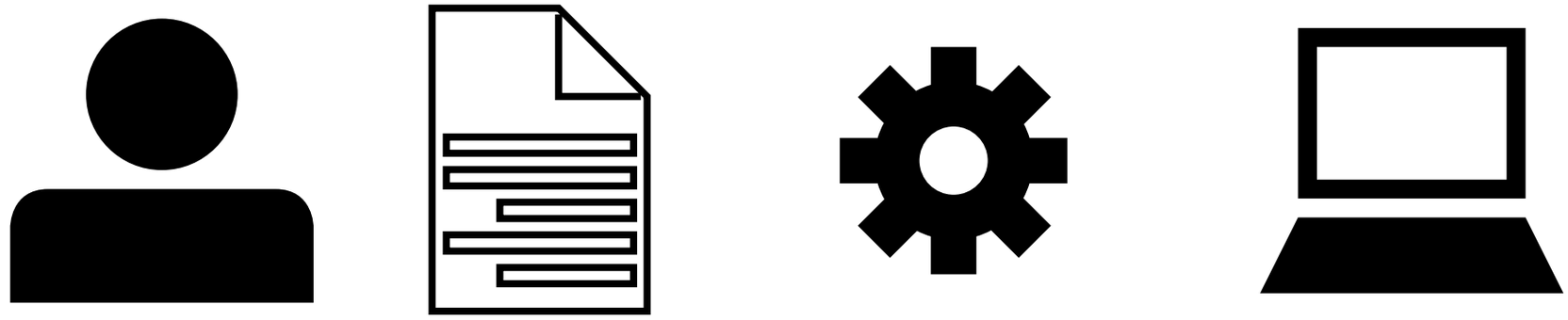
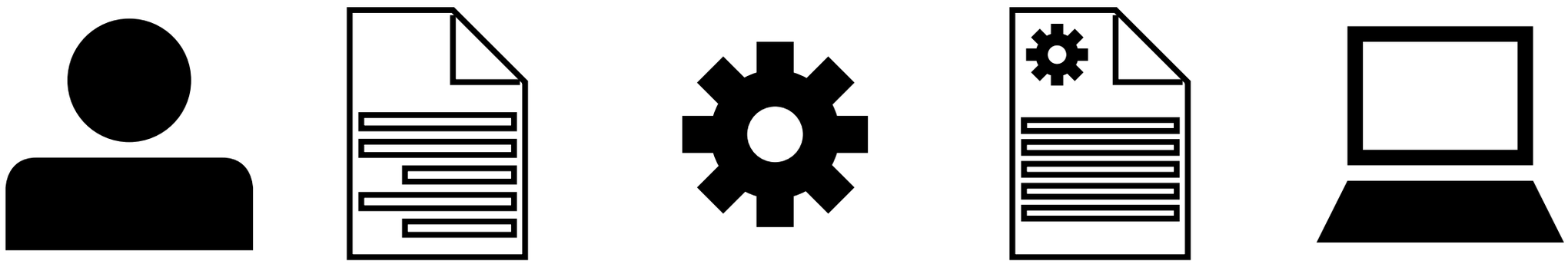
Programming Languages

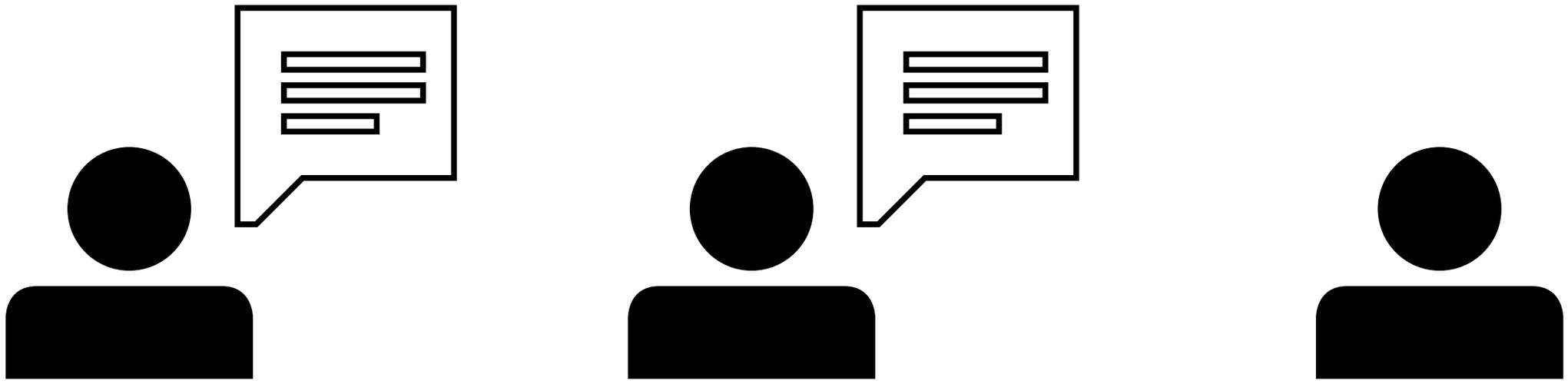
imperative & object-oriented

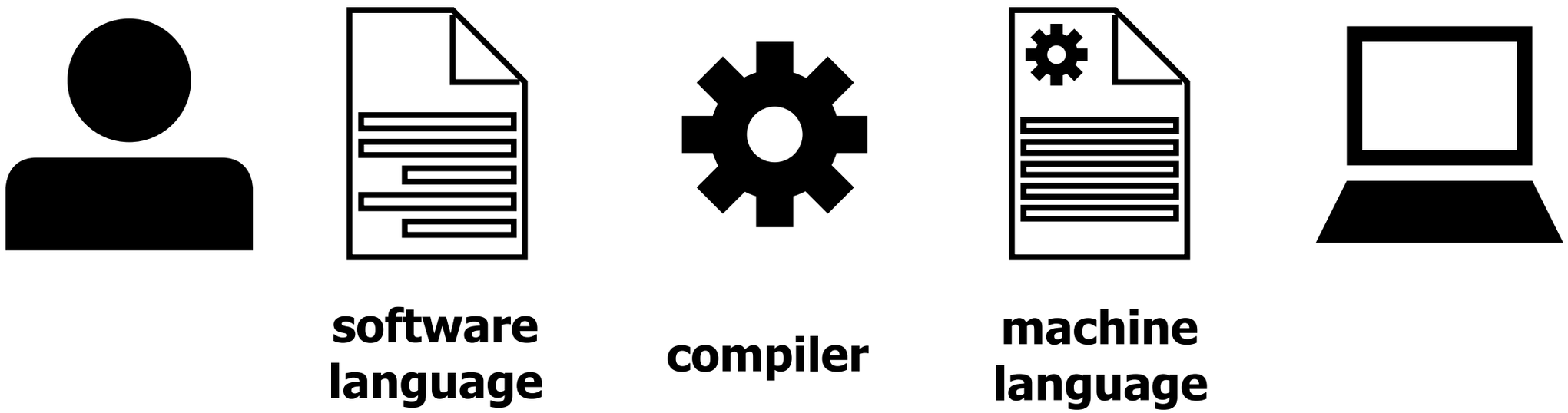
Guido Wachsmuth, Eelco Visser


BIGGER PICTU


software
language
interpreter

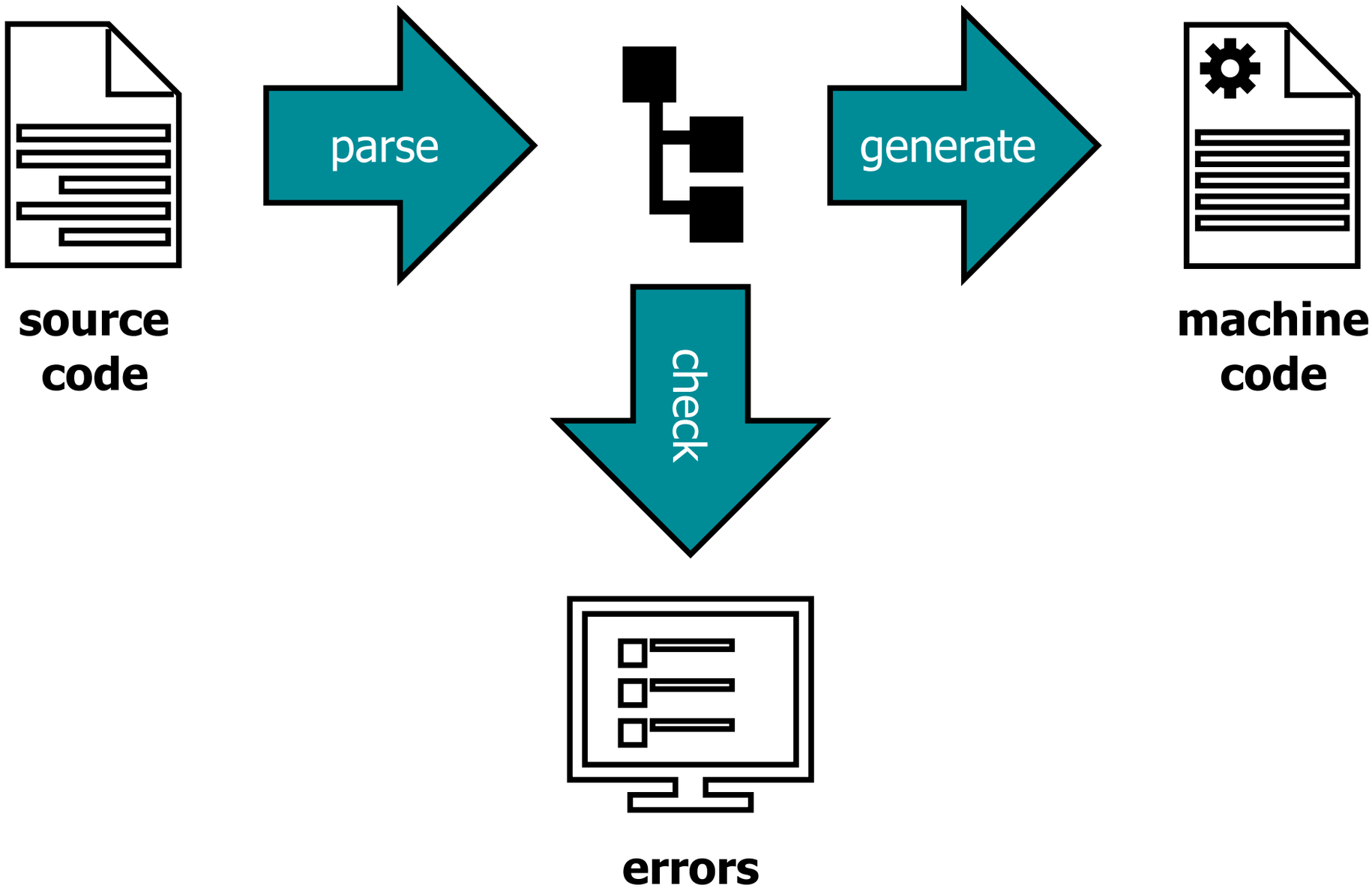

software
language
compiler
machine
language


natural language
translator
natural language


software language
compiler
machine language

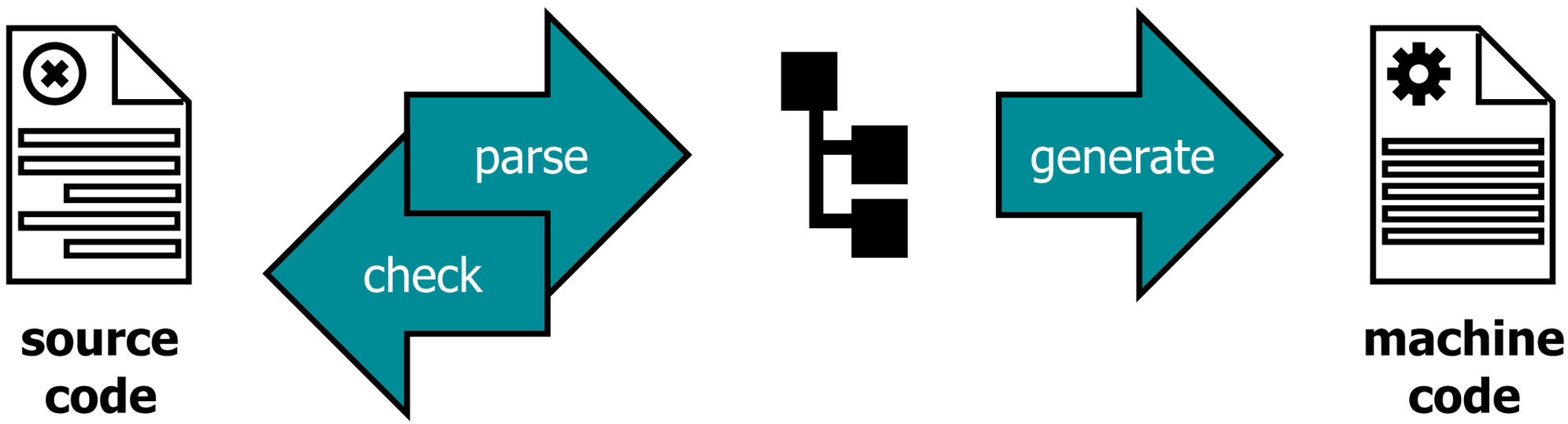
Traditional compilers

architecture



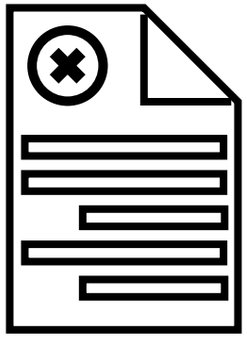
Modern compilers in IDEs

architecture

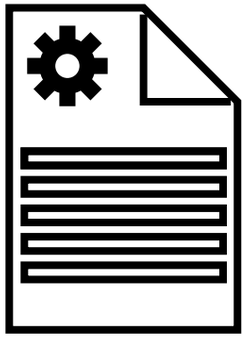


Compilers are inverted abstractions

architecture



source code



machine code

Compilers Invert Abstractions

Computers speak machine language

- basic instructions to move around and combine small things

Humans do not speak machine language very well

- “mov; loadi; jump” anyone?

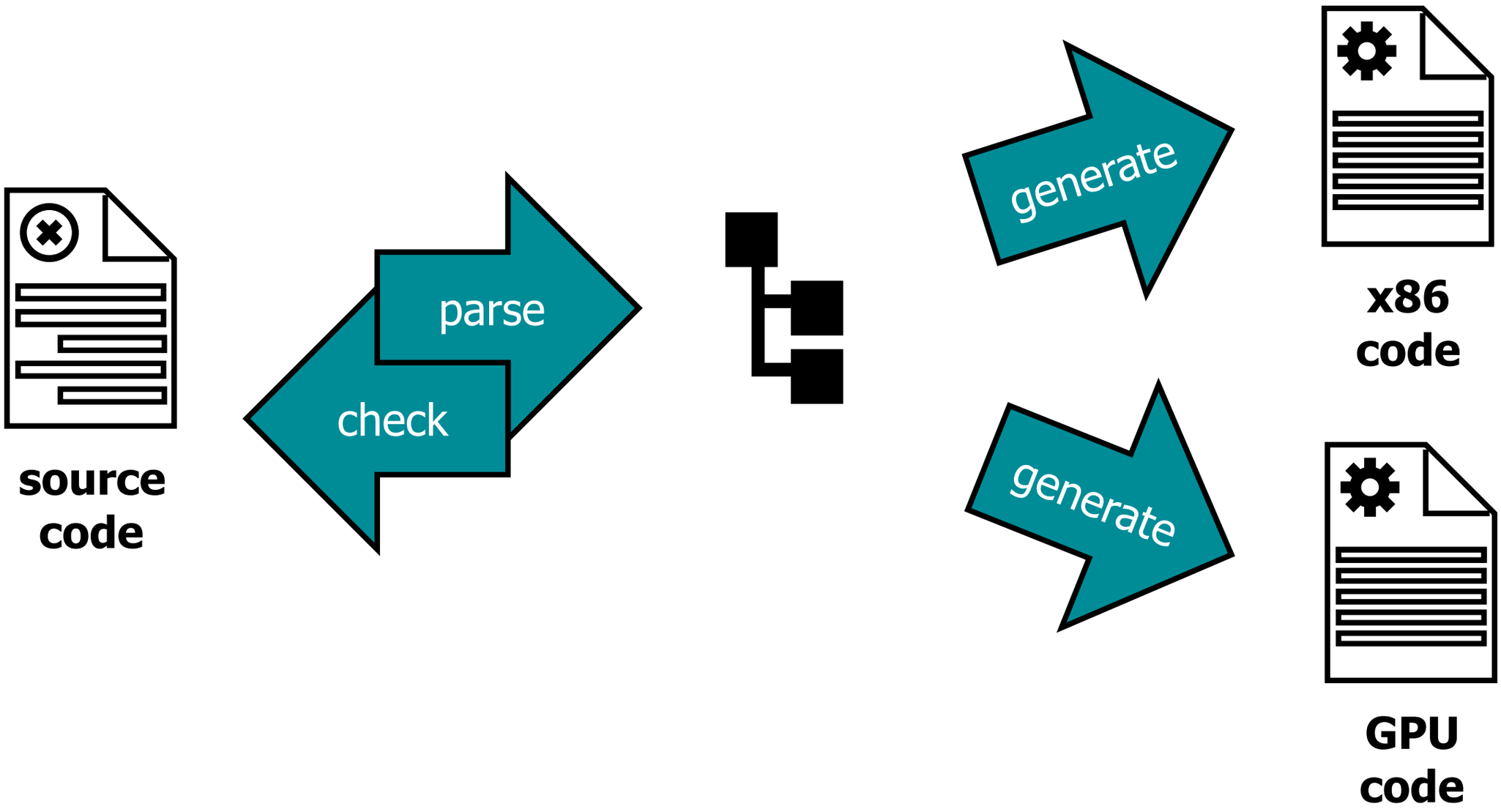
Programming languages abstract from machine language

- capture common programming patterns in language constructs
- abstract from uninteresting variation in machines

Compilers invert the abstractions of programming languages

Portability: Platform Independence

architecture



Understanding Compilers Requires …

Understanding machine languages

- machine architecture, instruction sets

Understanding programming language (abstraction)s

- memory, control-flow, procedures, modules, …
- safety mechanisms: type systems & other static analyses

Understanding how to define mapping from PL to ML

- semantics of such mappings
- techniques to implement such mappings

Imperative Languages

state & statements

procedures

types

Object-Oriented Languages

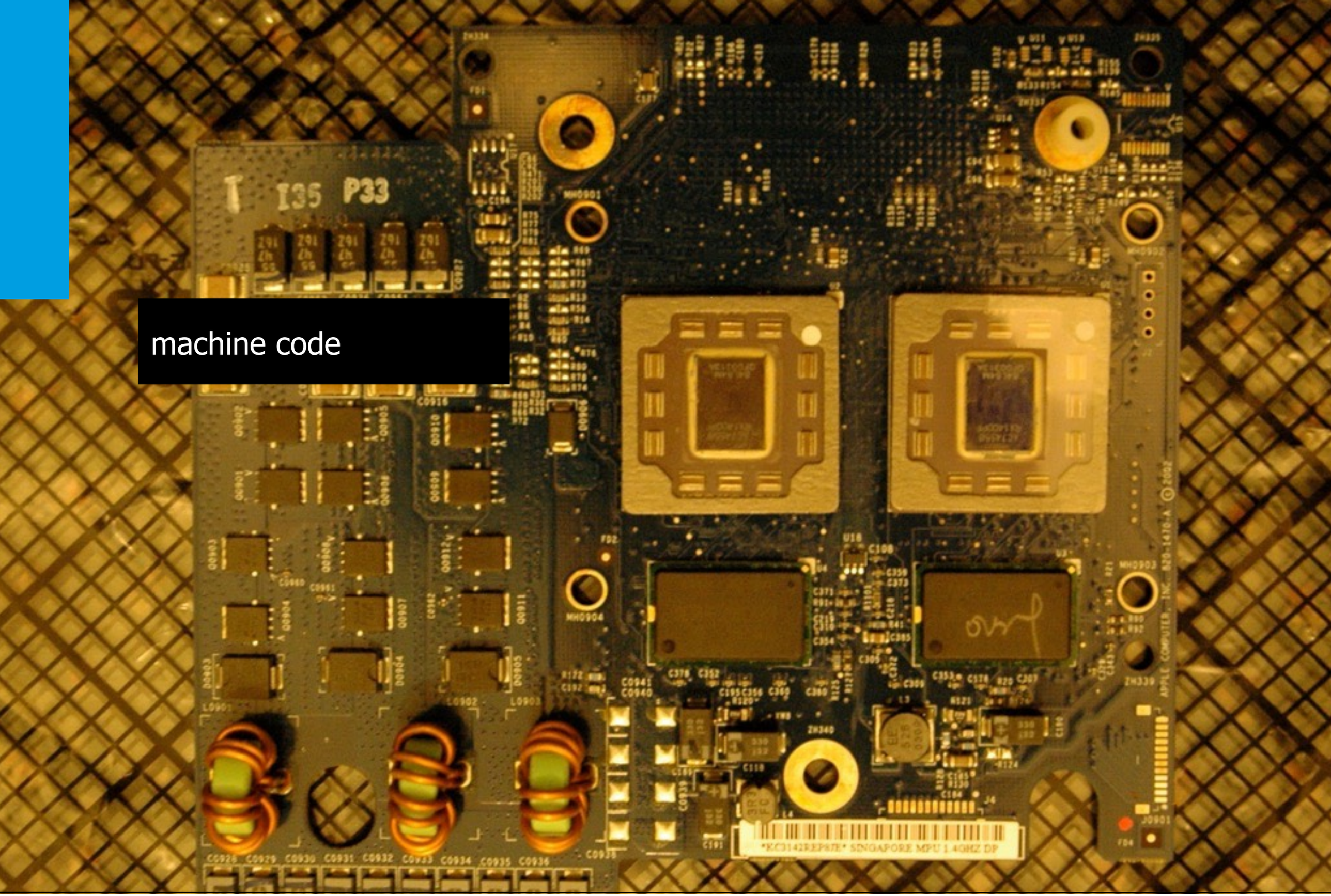
objects & messages

types

Imperative Languages

state & statements

machine code



State

basic concepts

Machine state

- a pile of data stored in memory
- memory hierarchy: registers, RAM, disk, network, …

Imperative program

- computation is series of changes to memory
- basic operations on memory (increment register)
- controlling such operations (jump, return address, …)
- control represented by state (program counter, stack, …)

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

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

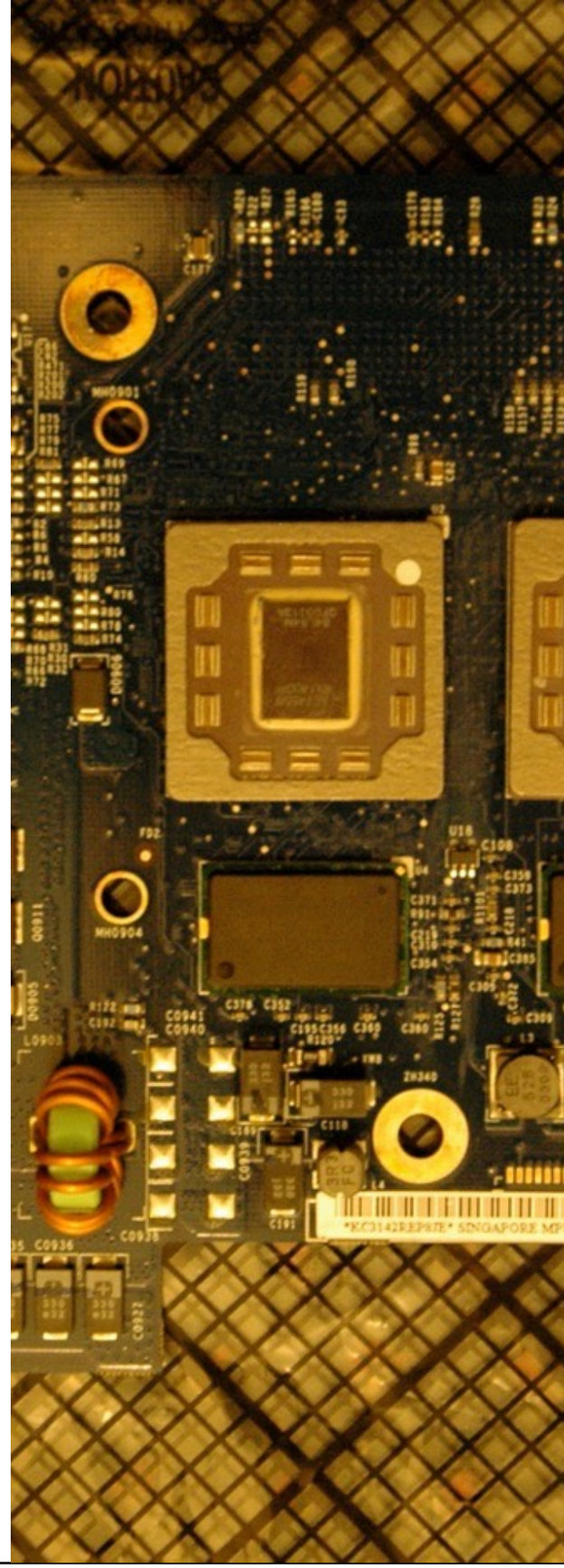
special purpose registers

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

Example: x86 Assembler

basic concepts

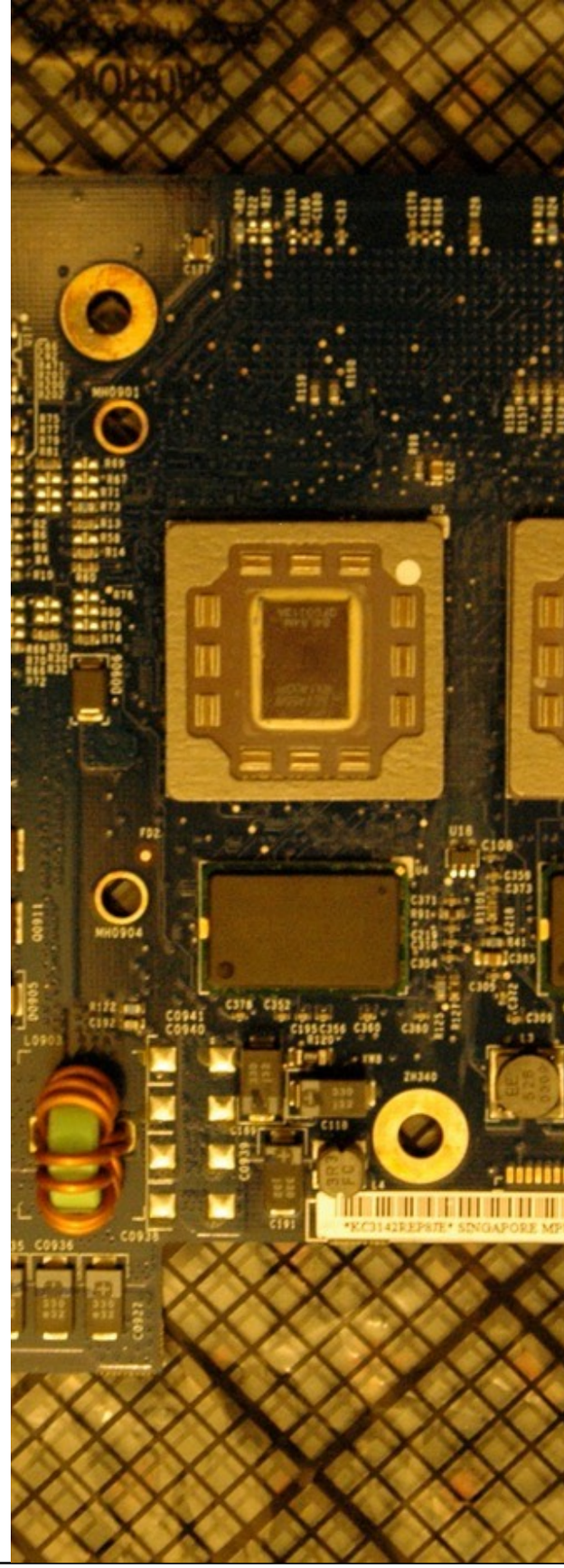
```
    mov AX [1]

    mov CX AX

L:  dec CX
    mul CX
    cmp CX 1
    ja  L

    mov [2] AX
```



Example: x86 Assembler

basic concepts

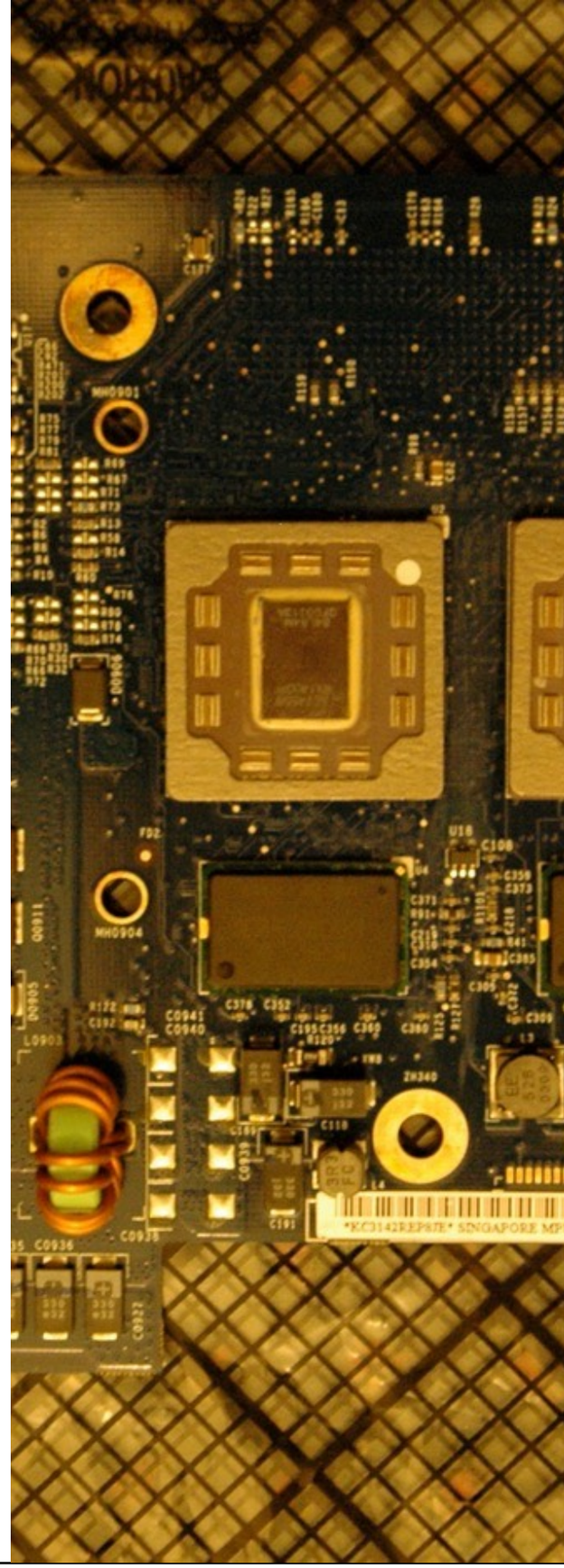
```
    mov AX [1]          read memory

    mov CX AX

L:  dec CX
    mul CX
    cmp CX 1
    ja  L

    mov [2] AX
```



Example: x86 Assembler

basic concepts

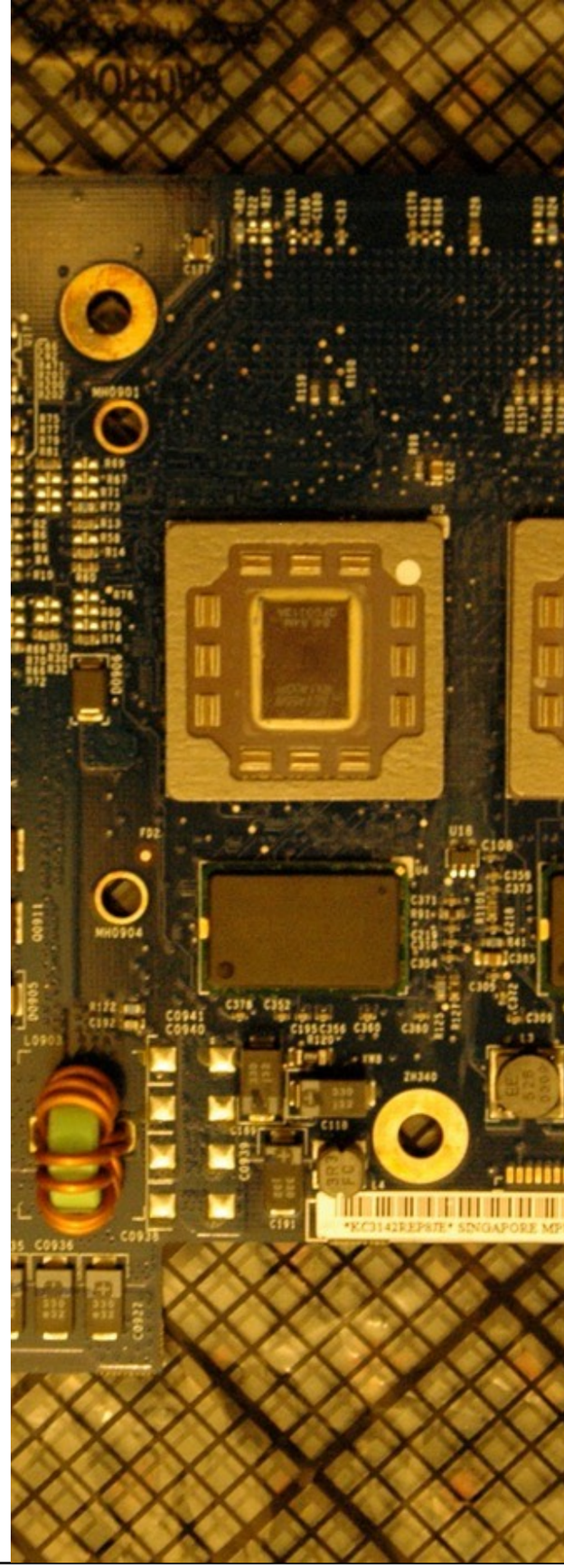
```
    mov AX [1]          read memory

    mov CX AX

L:  dec CX
    mul CX
    cmp CX 1
    ja  L

    mov [2] AX          write memory
```



Example: x86 Assembler

basic concepts

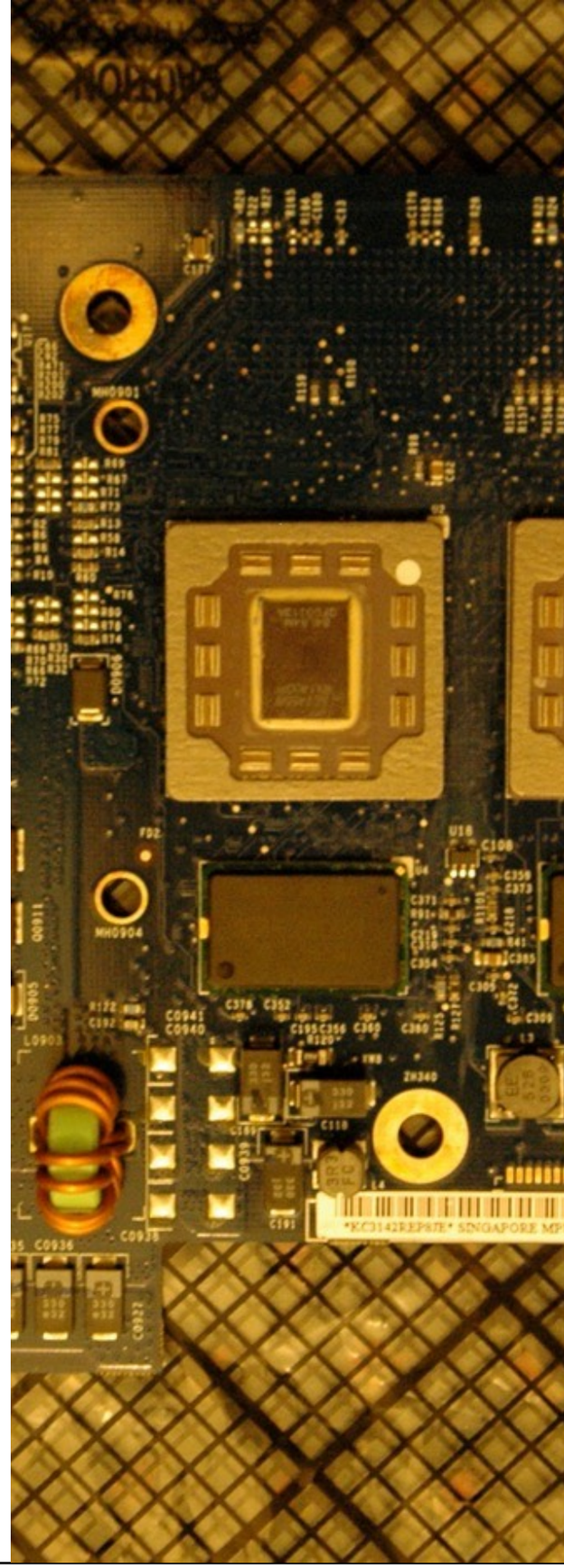
```
    mov AX [1]          read memory

    mov CX AX

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

    mov [2] AX          write memory
```



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
            iconst_1
            ireturn

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

Example: Java Bytecode

basic concepts

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

          iload 1
          ifne else
          iconst_1
          ireturn

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

read memory

Example: Java Bytecode

basic concepts

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

        iload 1
        ifne else
        iconst_1
        ireturn

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

read memory

calculation

Example: Java Bytecode

basic concepts

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

        iload 1
        ifne else
        iconst_1
        ireturn

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

jump

read memory

calculation

Memory & Control Abstractions

basic concepts

Memory abstractions

- variables: abstract over data storage
- expressions: combine data into new data
- assignment: abstract over storage operations

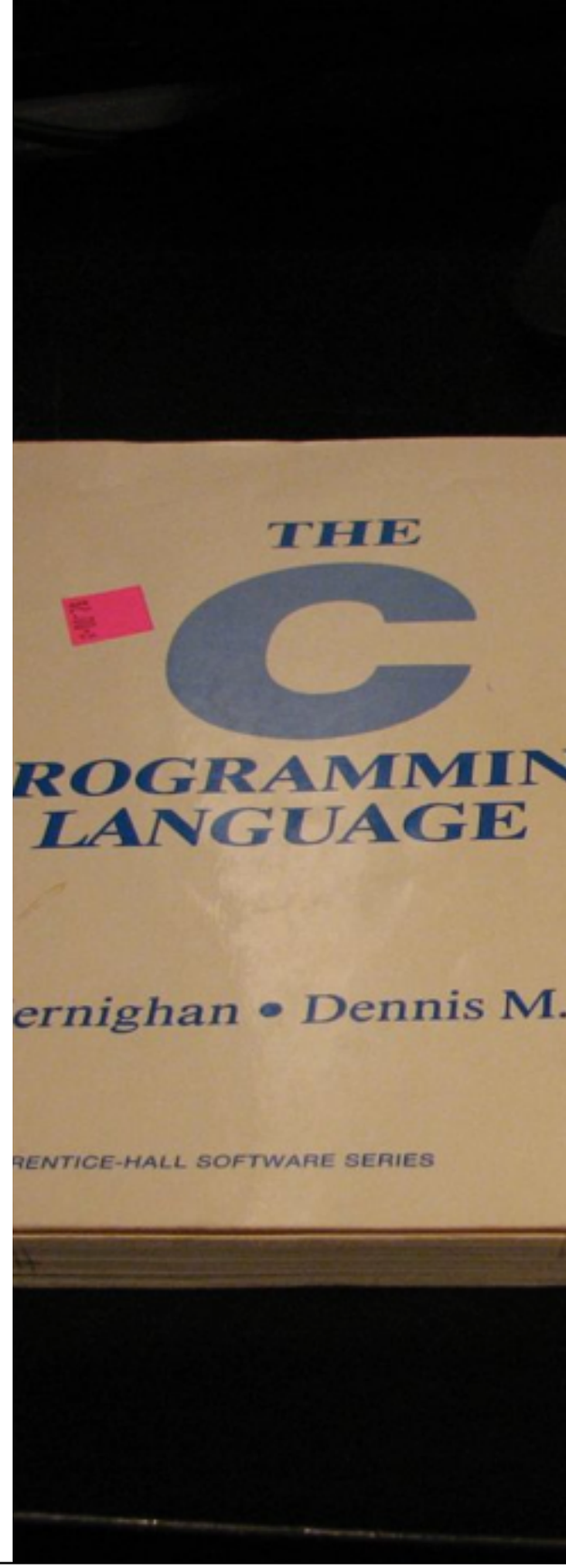
Control-flow abstractions

- structured control-flow: abstract over unstructured jumps
- ‘go to statement considered harmful’ Edgser Dijkstra, 1968

Example: C

states & statements

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

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

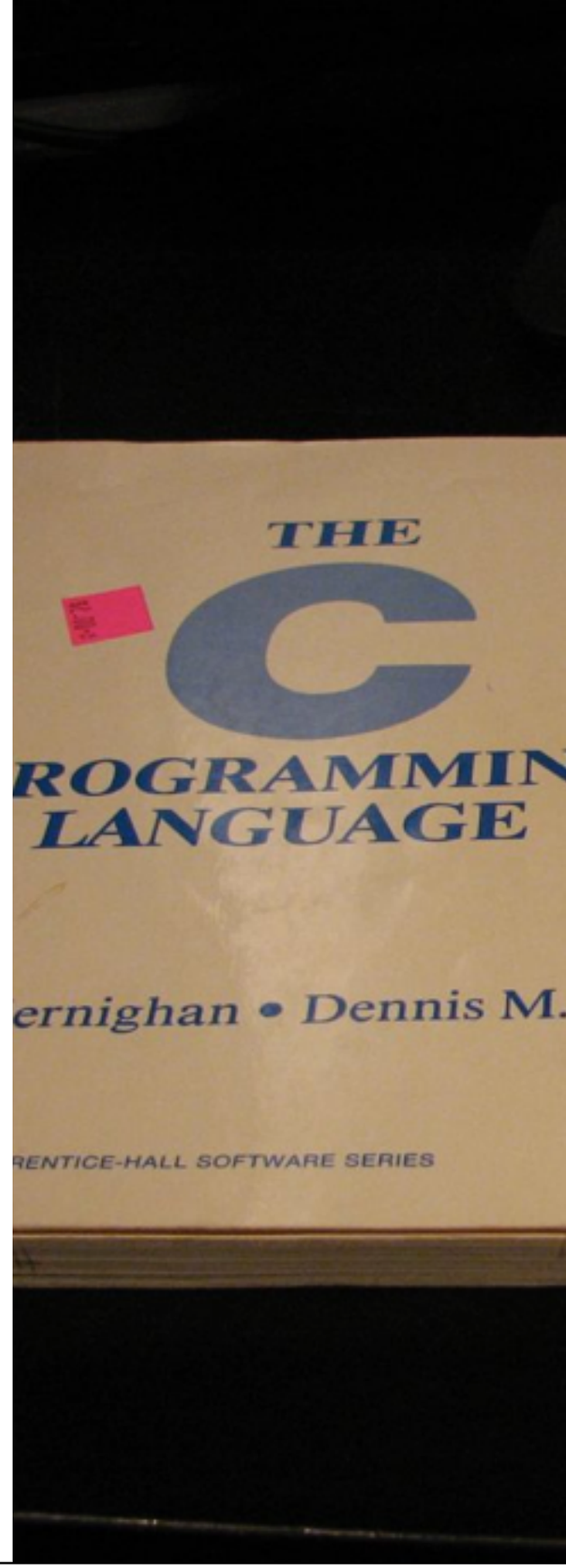


Example: C

states & statements

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

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

variable



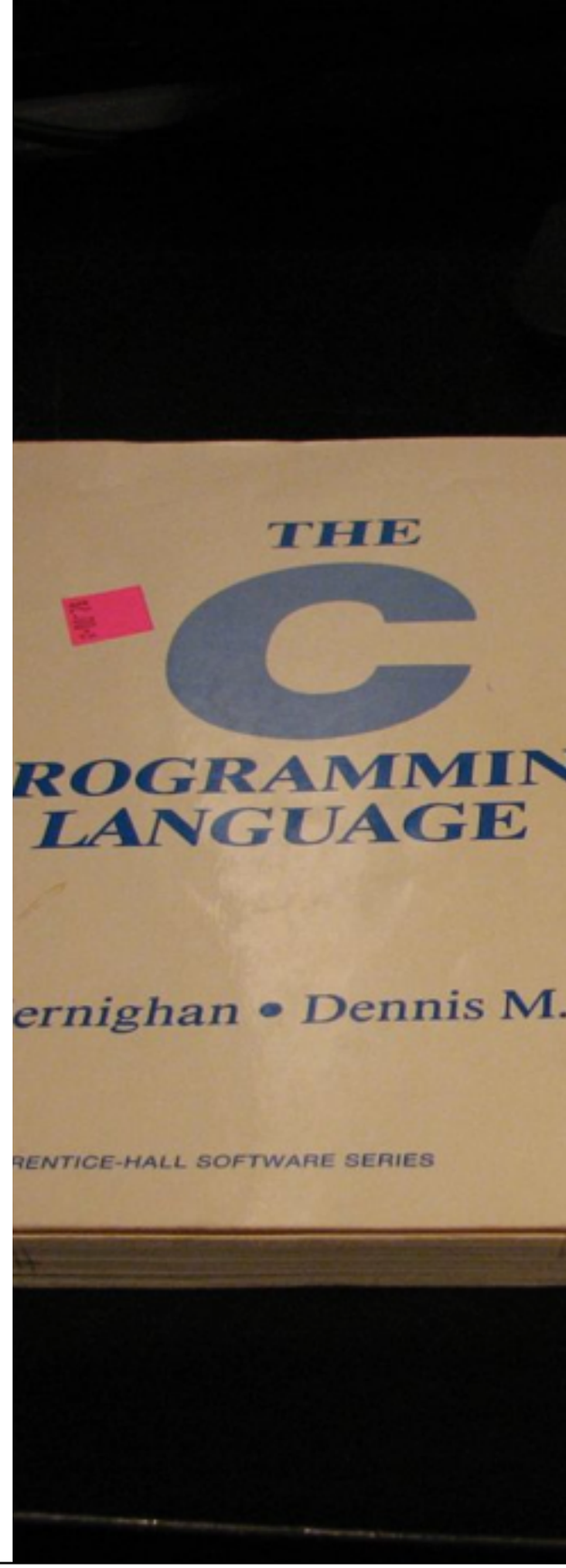
Example: C

states & statements

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

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

variable

expression



Example: C

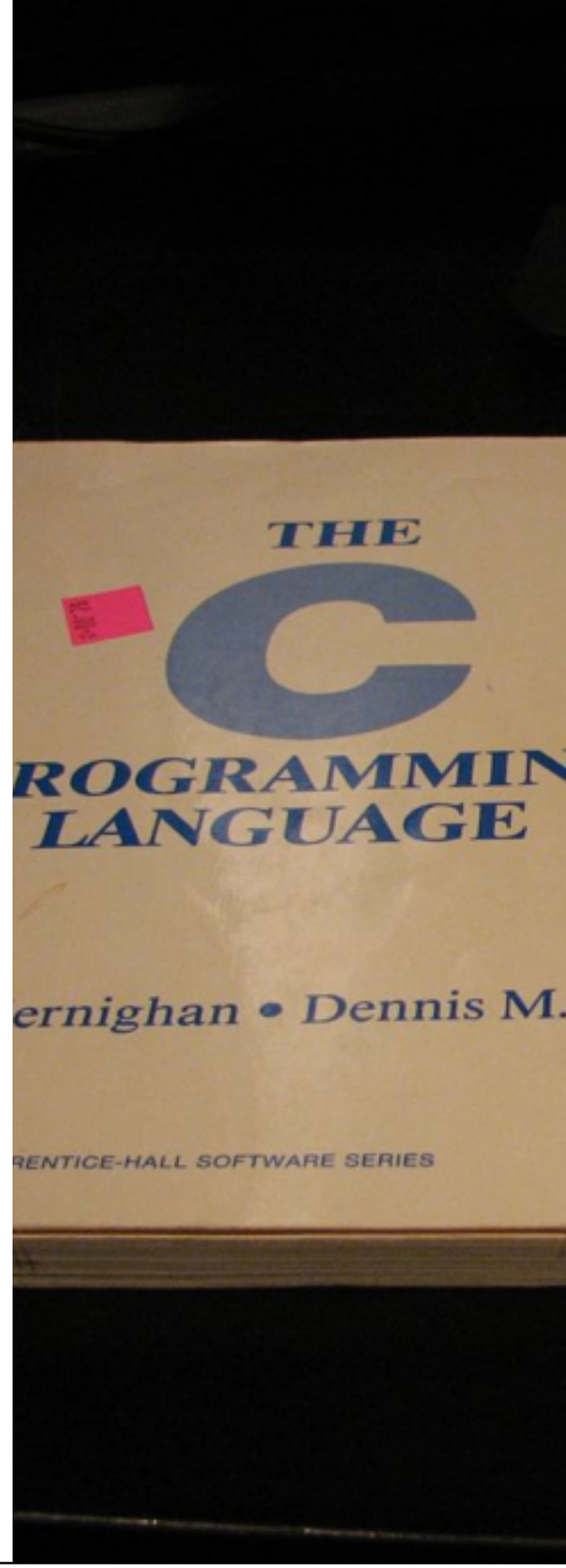
states & statements

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

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

variable

expression

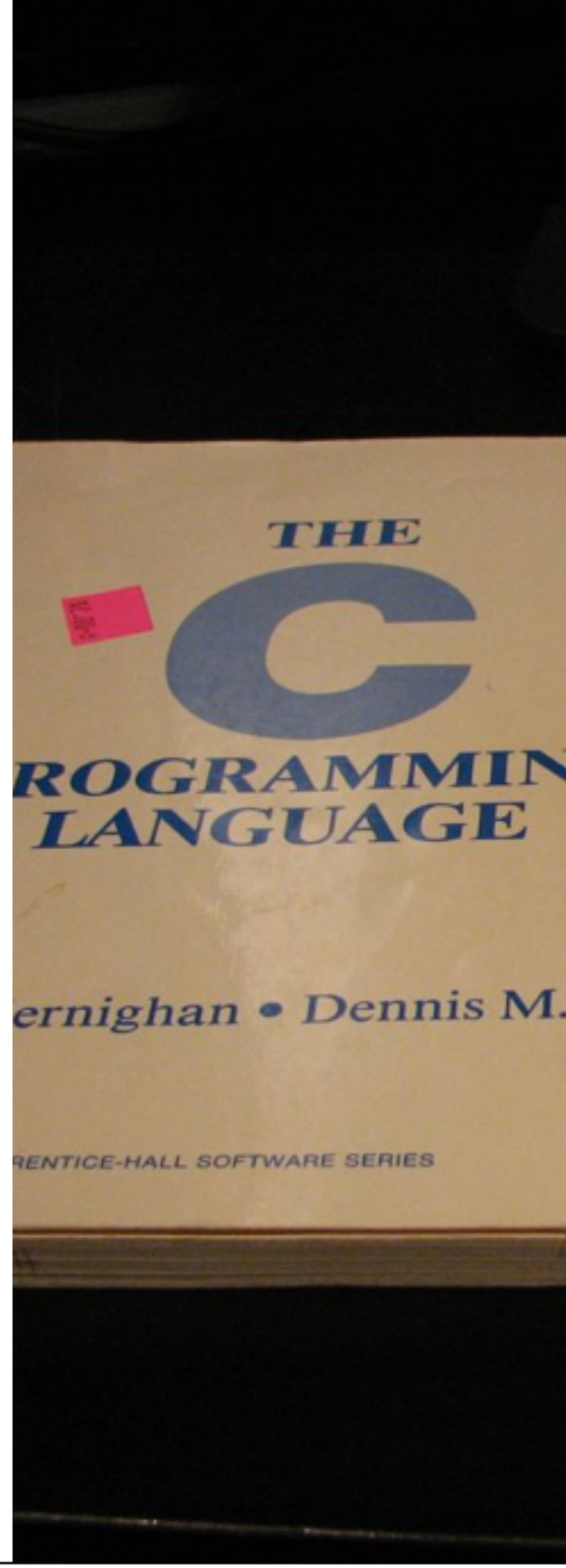
assignment



Example: C

states & statements

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

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



Example: Tiger

states & statements

```
/* factorial function */

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

Example: Tiger

states & statements

```
/* factorial function */

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

variable

Example: Tiger

states & statements

```
/* factorial function */

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

Example: Tiger

states & statements

```
/* factorial function */

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

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
```

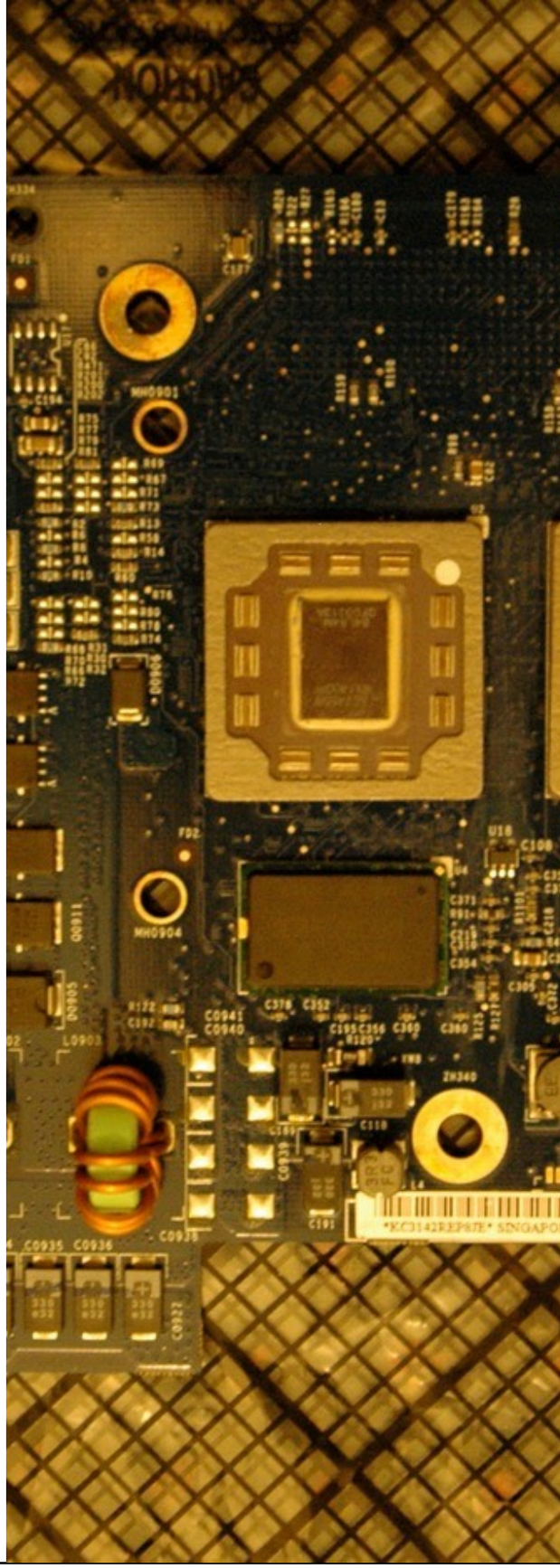
Imperative Languages

procedures

Example: x86 Assembler
modularity

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

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



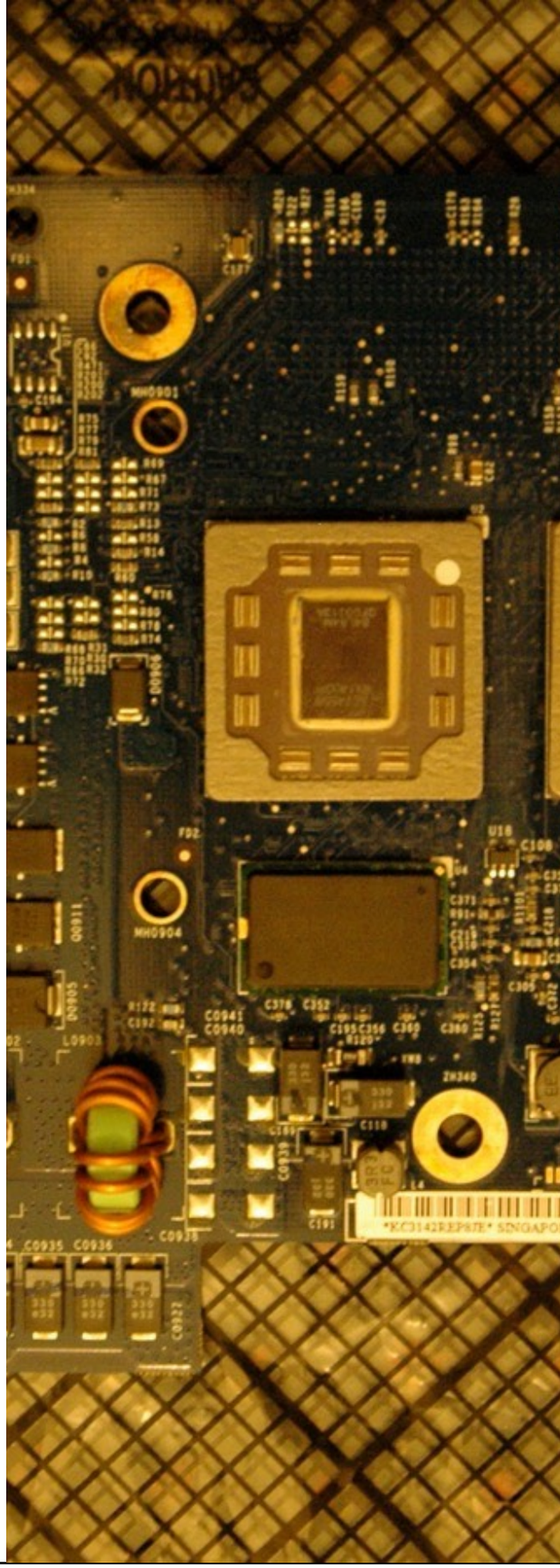
Example: x86 Assembler

modularity

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

pass parameter

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



Example: x86 Assembler

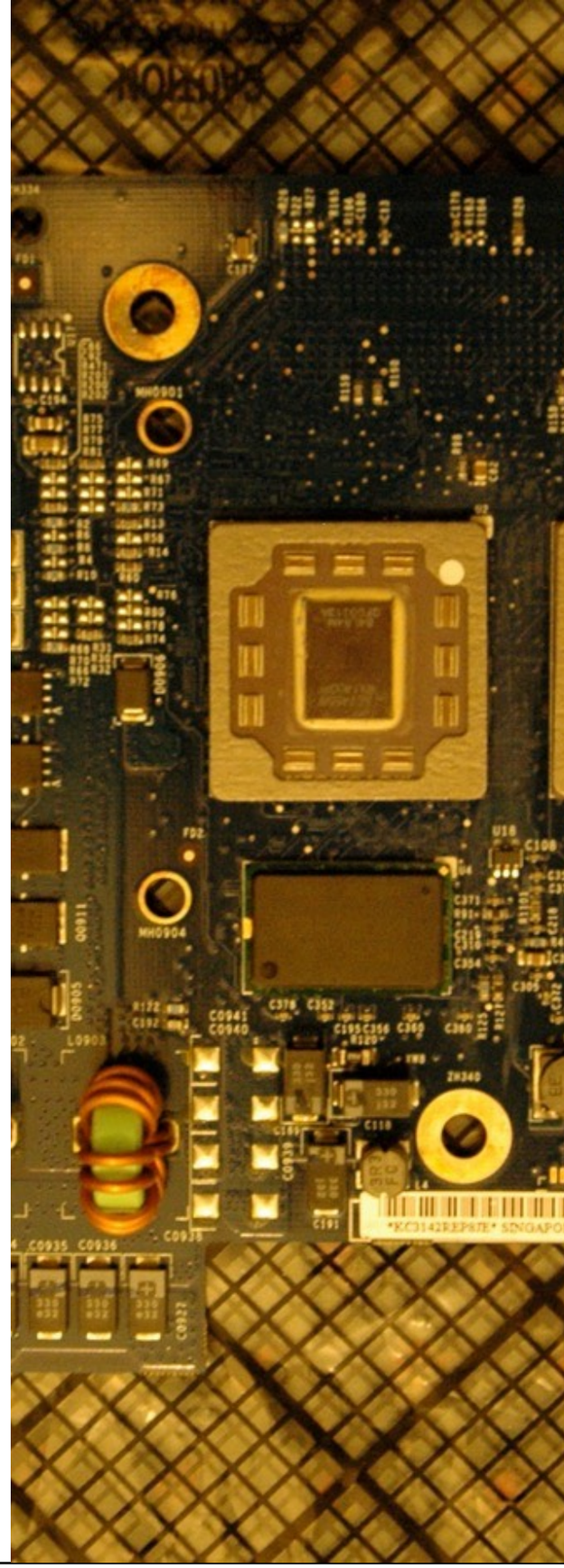
modularity

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

pass parameter

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

new stack frame



Example: x86 Assembler

modularity

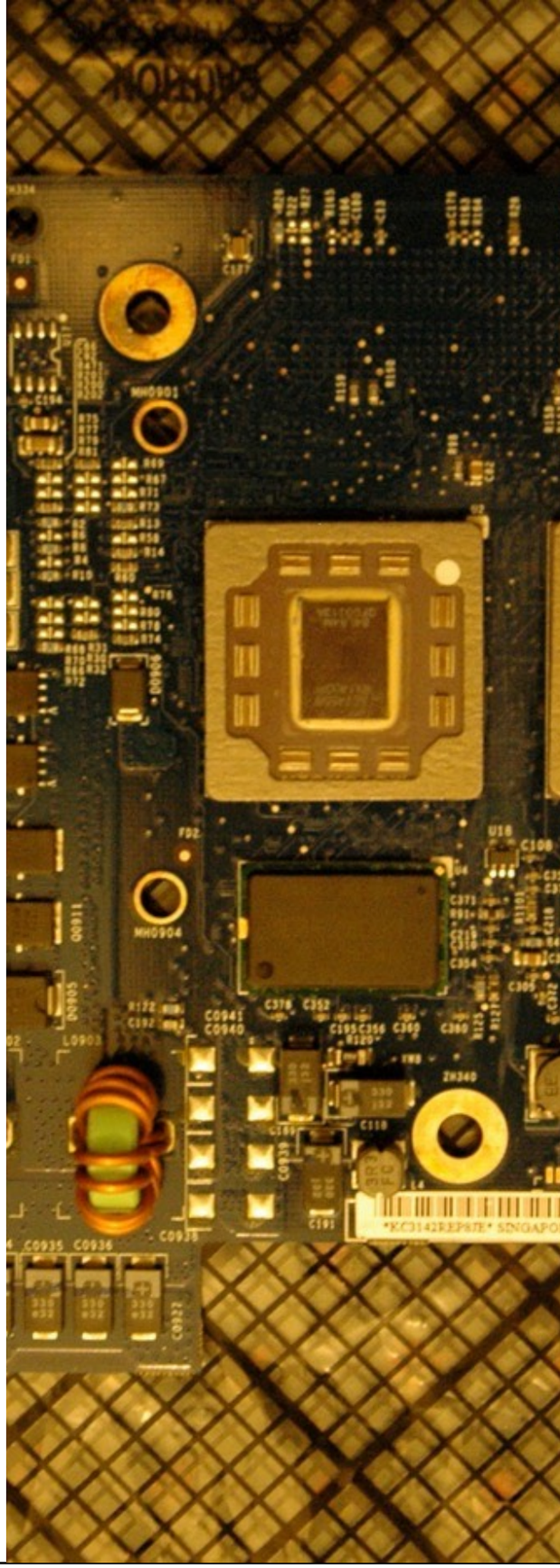
```
push 21
push 42
call _f
add  SP 8
```

pass parameter

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

new stack frame

access parameter



Example: x86 Assembler

modularity

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

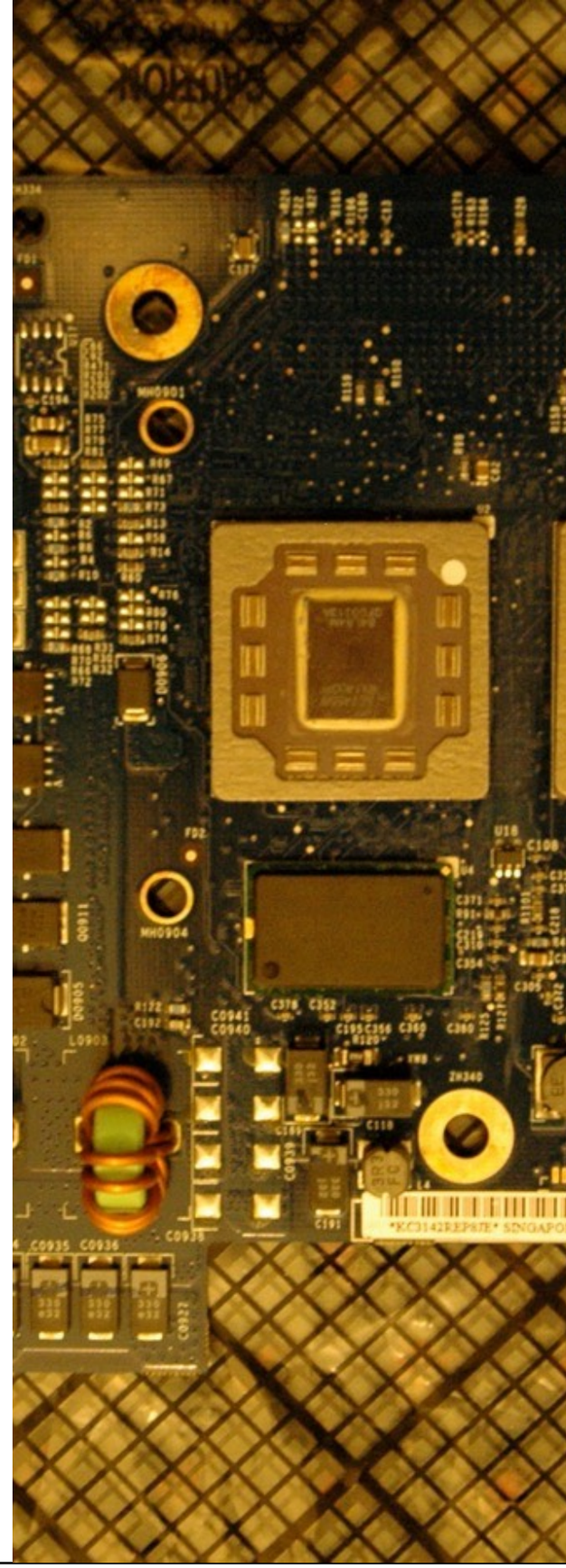
pass parameter

```
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: x86 Assembler

modularity

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

pass parameter

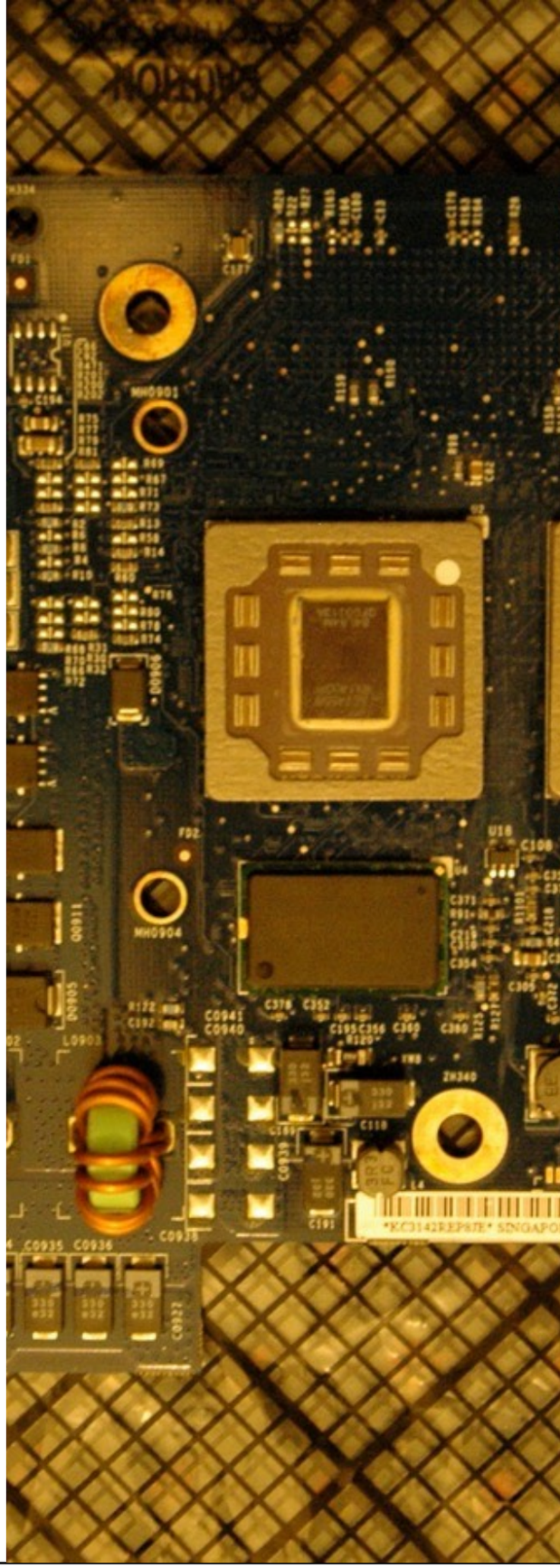
free parameters

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

new stack frame

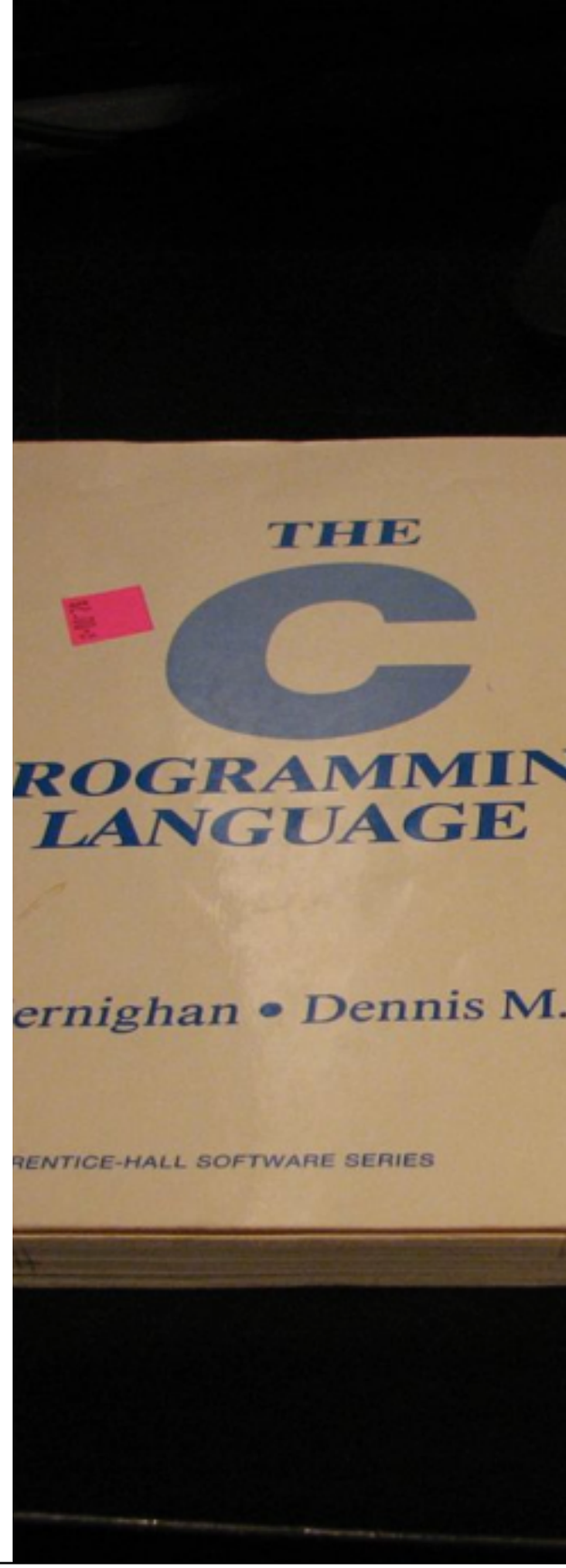
access parameter

old stack frame



Example: C
procedures

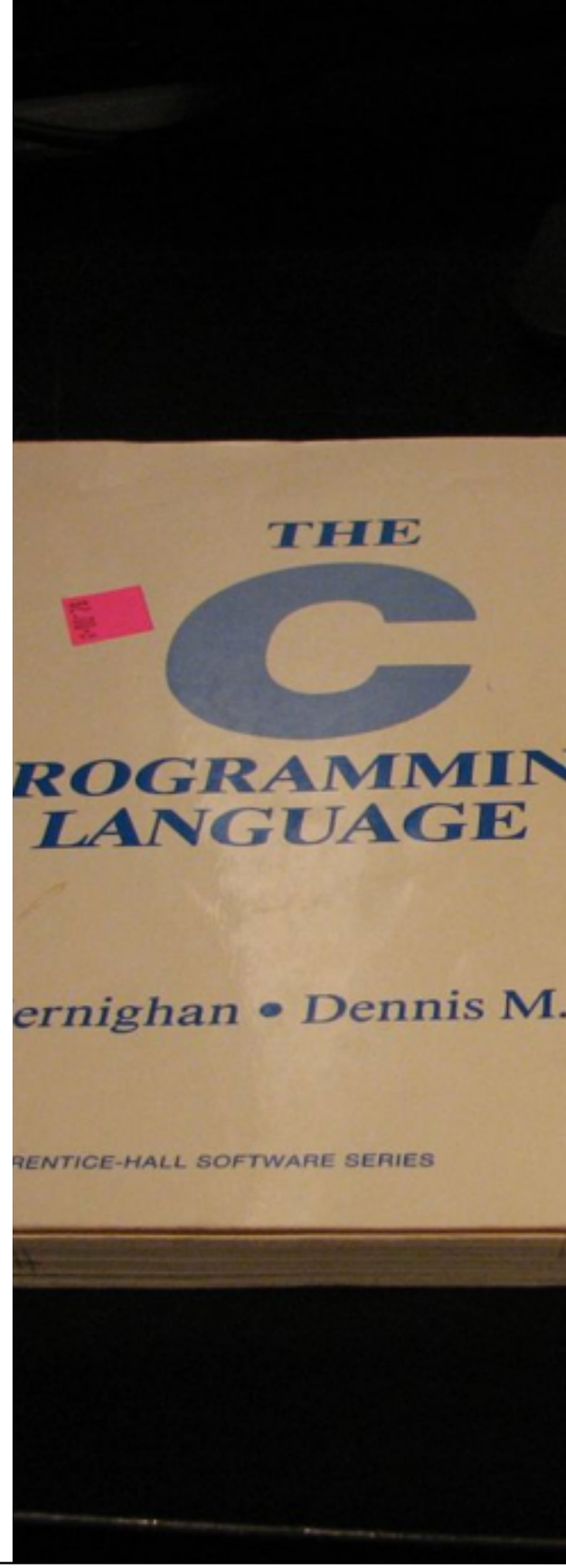
```c
#include <stio.h>

/* 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;
}
```



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

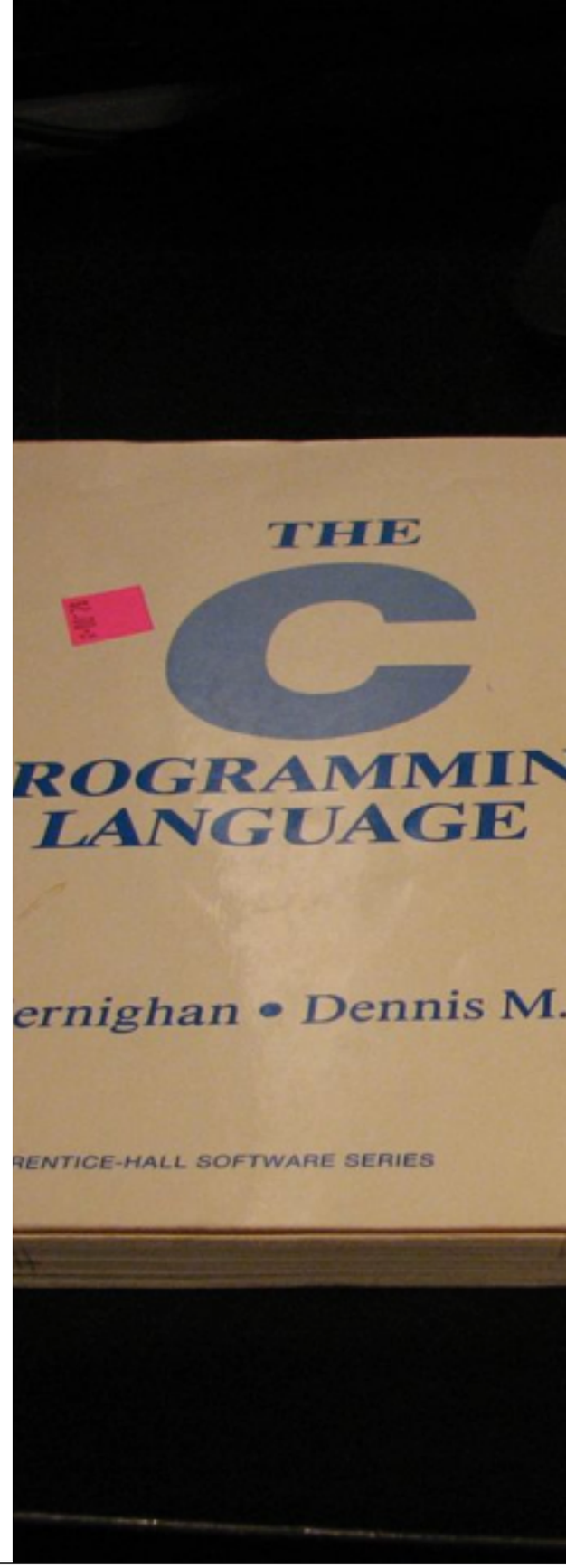


Example: C
procedures

```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

actual parameter



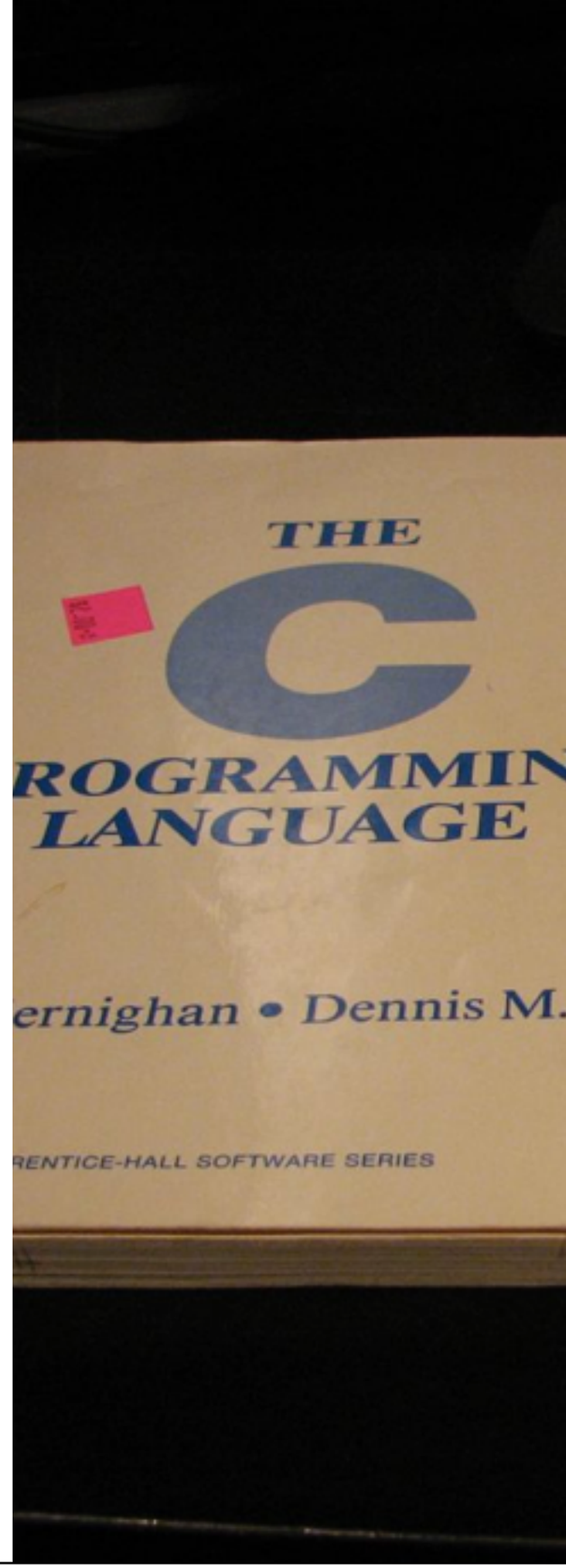
Example: C
procedures

```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

local variable

actual parameter



Example: C
procedures

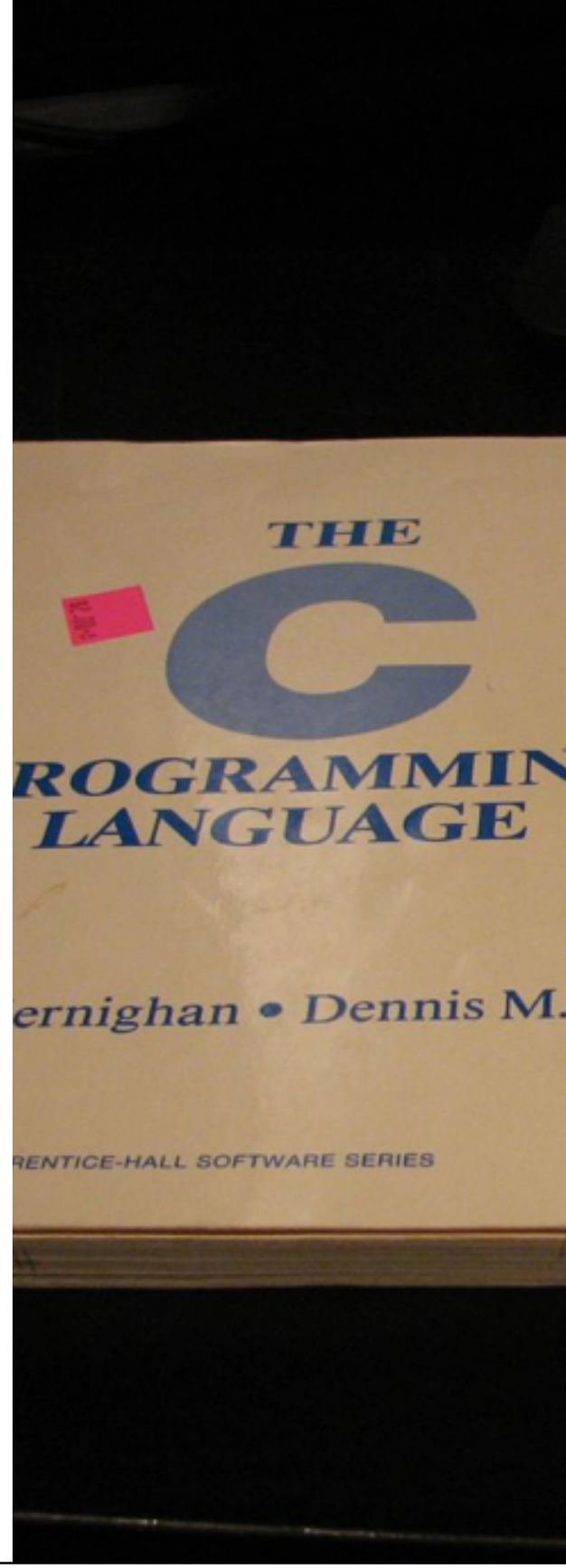
```c
#include <stio.h>

/* 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
```

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

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

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

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
```

Imperative Languages

types

Types & Type Checking

Subtitle Text

Type

- type is category of values
- operation can be applied to all values in some category

Type checking

- safety: ensure that operation is applied to right input values
- static: check during compile time
- dynamic: check during execution

Type Systems
dynamic & static typing

Machine code

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

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

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 Systems
compatibility

Type compatibility

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

Type equivalence

- structural type systems
- nominal type systems

Type Systems
compatibility

Type compatibility

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

Type equivalence

- structural type systems
- nominal 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

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

Polymorphism

biology

Polymorphism

biology

Polymorphism

biology

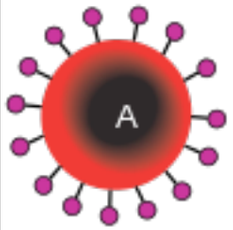
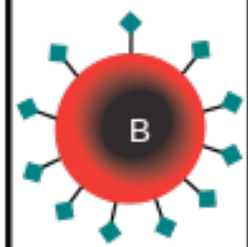
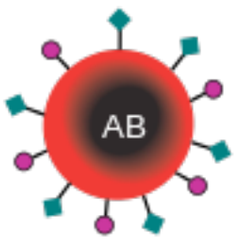
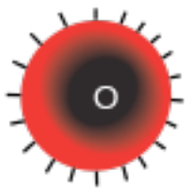
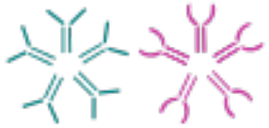
Polymorphism

biology

the occurrence of more than one form

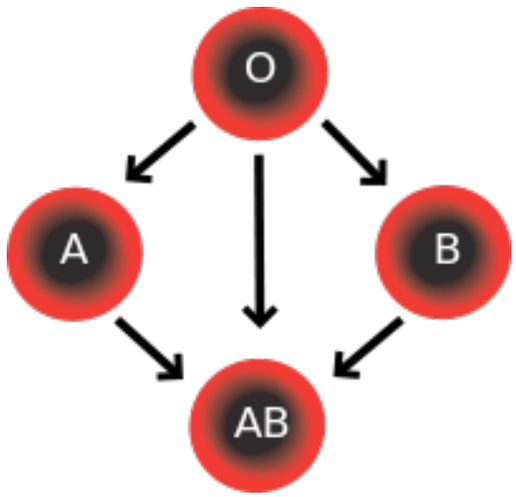
Polymorphism

biology

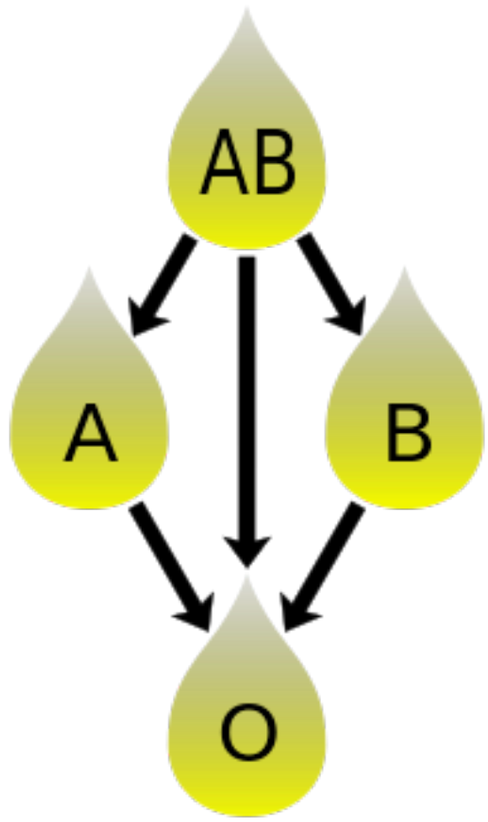
	Group A	Group B	Group AB	Group O
Red blood cell type				
Antibodies in Plasma	 Anti-B	 Anti-A	None	 Anti-A and Anti-B
Antigens in Red Blood Cell	A antigen	B antigen	 A and B antigens	None

Polymorphism

biology



can give blood to



can give plasma to

Polymorphism

programming languages

Polymorphism

programming languages

21 + 21

Polymorphism

programming languages

```
21 + 21
```

```
21.0 + 21.0
```

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"
```

Object-Oriented Languages

objects & messages

Modularity

objects & messages

Objects

- generalization of records
- identity
- state
- behavior

Modularity

objects & messages

Objects

- generalization of records
- identity
- state
- behavior

Messages

- objects send and receive messages
- trigger behavior
- imperative realization: method calls

Object-Oriented Languages types

Modularity
classes

Classes

- generalization of record types
- characteristics of objects: attributes, fields, properties
- behavior of objects: methods, operations, features

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

Modularity
classes

Classes

- generalization of record types
- characteristics of objects: attributes, fields, properties
- behavior 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 behavior
- nominal 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'() {…}
}
```

Modularity

inheritance vs. interfaces

Inheritance

- inherit attributes & methods
- additional attributes & methods
- override behavior
- nominal subtyping

Interfaces

- avoid multiple inheritance
- interface: contract for attributes & methods
- class: provides attributes & methods
- nominal 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

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

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

Type Systems

static vs. dynamic dispatch

Dispatch

- link method call to method

Static dispatch

- type information at compile-time

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);
```

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 G {
   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);
```

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