# Lecture 1: What is a Compiler?

**CS4200 Compiler Construction**

**Eelco Visser**

**TU Delft**

**September 2018**

# Etymology

**Latin**

**Etymology**

From *con-* ("with, together") + *pīlō* ("ram down").

**Pronunciation**

• (Classical) IPA<sup>(key)</sup>: /komˈpiː.loː/, [kɔmˈpiː.ɫoː]

**Verb**

**compīlō** (*present infinitive* **compīlāre**, *perfect active* **compīlāvī**, *supine* **compīlātum**); *first conjugation*

1. I snatch together and carry off; plunder, pillage, rob, steal.

**https://en.wiktionary.org/wiki/compilo#Latin**

# Dictionary

**English**

**Verb**

**compile** (*third-person singular simple present* **compiles**, *present participle* **compiling**, *simple past and past participle* **compiled**)

1. (transitive) To put together; to assemble; to make by gathering things from various sources. *Samuel Johnson* **compiled** *one of the most influential dictionaries of the English language.*
2. (obsolete) To construct, build. quotations
3. (transitive, programming) To use a compiler to process source code and produce executable code. *After I* **compile** *this program I'll run it and see if it works.*
4. (intransitive, programming) To be successfully processed by a compiler into executable code. *There must be an error in my source code because it won't* **compile**.
5. (obsolete, transitive) To contain or comprise. quotations
6. (obsolete) To write; to compose.

**https://en.wiktionary.org/wiki/compile**

# Etymology

John McCarthy developed Lisp in 1958 at MIT. "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I". Information Processing Language  from 1955 or 1956, and already included many of the concepts, such as list-processing and recursion, which came to be used in Lisp.

The first compiler was written by [Grace Hopper], in 1952, for the [A-0 System] language. The term *compiler* was coined by Hopper.[1][2] The A-0 functioned more as a loader or [linker] than the modern notion of a compiler.
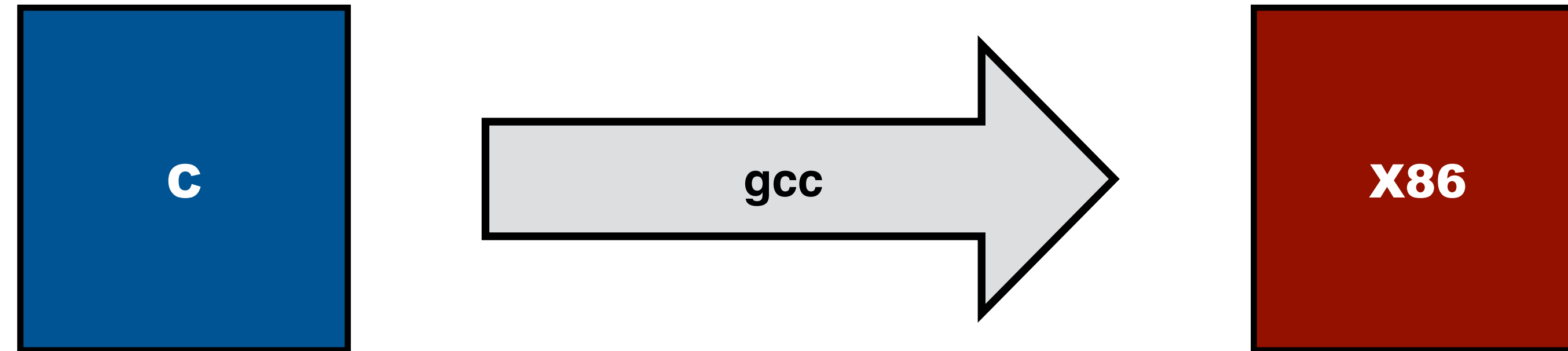
The FORTRAN team led by John W. Backus at IBM introduced the first commercially available compiler, in 1957, which took 18 person-years to create
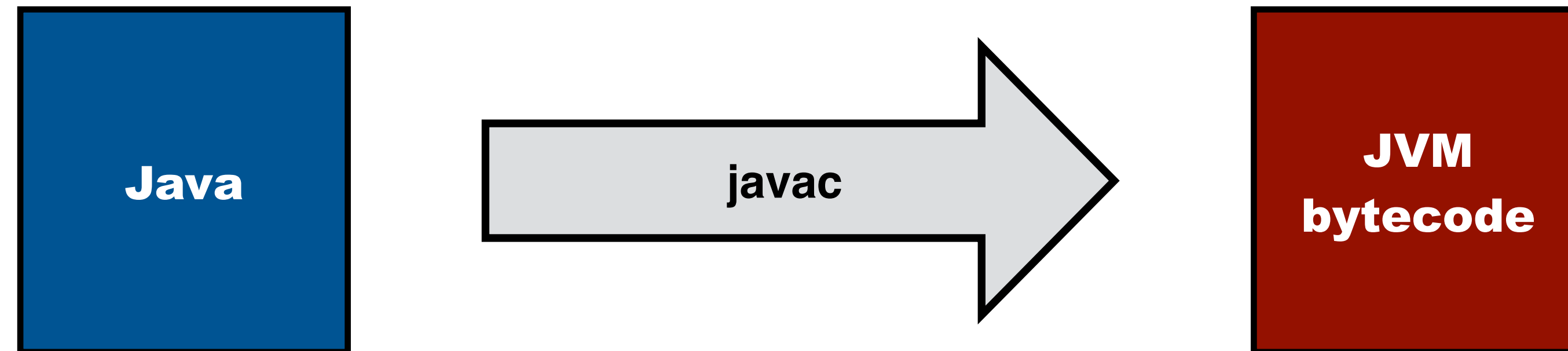
**https://en.wikipedia.org/wiki/History_of_compiler_construction**

# Compiling = Translating
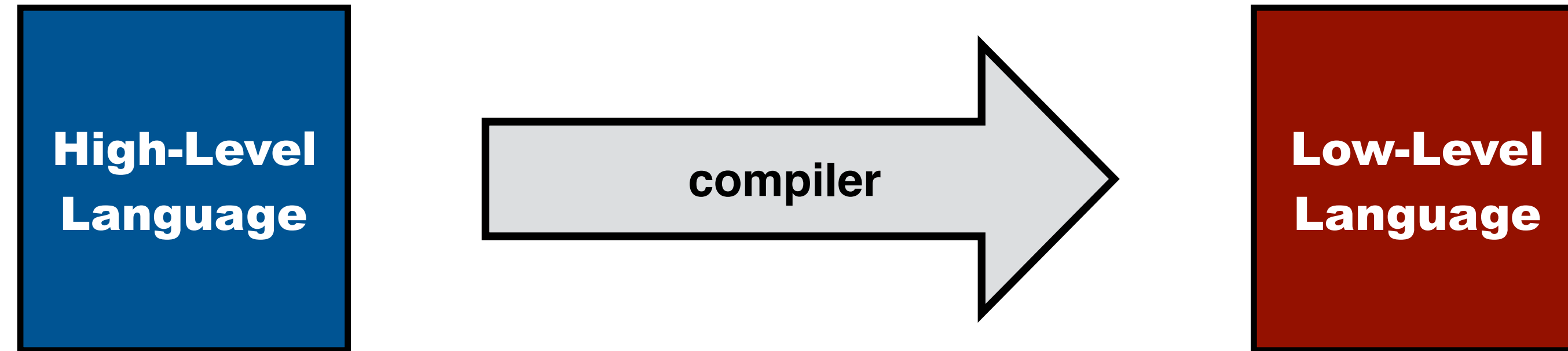


A compiler translates high-level programs to low-level programs
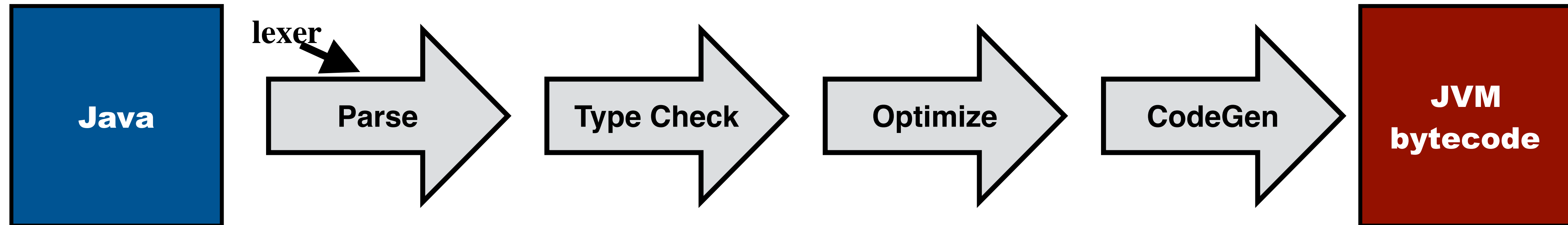
# Compiling = Translating



C → gcc → X86

**GCC translates C programs to object code for X86 (and other architectures)**

# Compiling = Translating



**A Java compiler translates Java programs to bytecode instructions for Java Virtual Machine**

# Architecture: Multi-Pass Compiler

| Java | lexer → | Parse → | Type Check → | Optimize → | CodeGen → | JVM bytecode |

**A modern compiler typically consists of sequence of stages or passes**

# Intermediate Representations



**A compiler is a composition of a series of translations between intermediate languages**

# Compiler Components



## Parser

- Reads in program text
- Checks that it complies with the syntactic rules of the language
- Produces an abstract syntax tree
- Represents the underlying (syntactic) structure of the program.

# Compiler Components



## Type checker

- Consumes an abstract syntax tree
- Checks that the program complies with the static semantic rules of the language
- Performs name analysis, relating uses of names to declarations of names
- Checks that the types of arguments of operations are consistent with their specification

# Compiler Components

| Java | → Parse → | Abstract Syntax Tree | → Type Check → | Annotated AST | → Optimize → | Transformed AST | → CodeGen → | JVM bytecode |

## Optimizer

- Consumes a (typed) abstract syntax tree
- Applies transformations that improve the program in various dimensions
  - execution time
  - memory consumption
  - energy consumption.

**Constant folding,
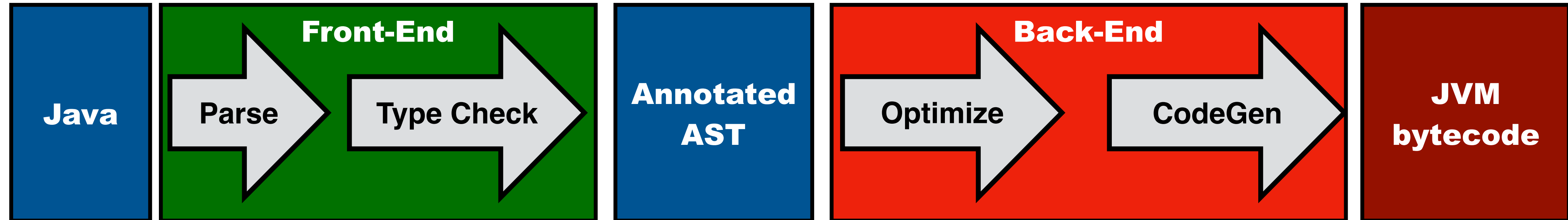Constant propagation,
...**

# Compiler Components



## Code generator

- Transforms abstract syntax tree to instructions for a particular computer architecture
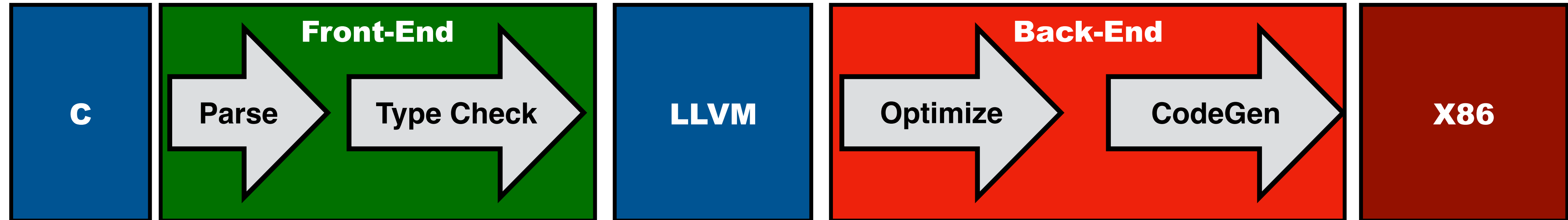- aka instruction selection

## Register allocator

- Assigns physical registers to symbolic registers in the generated instructions
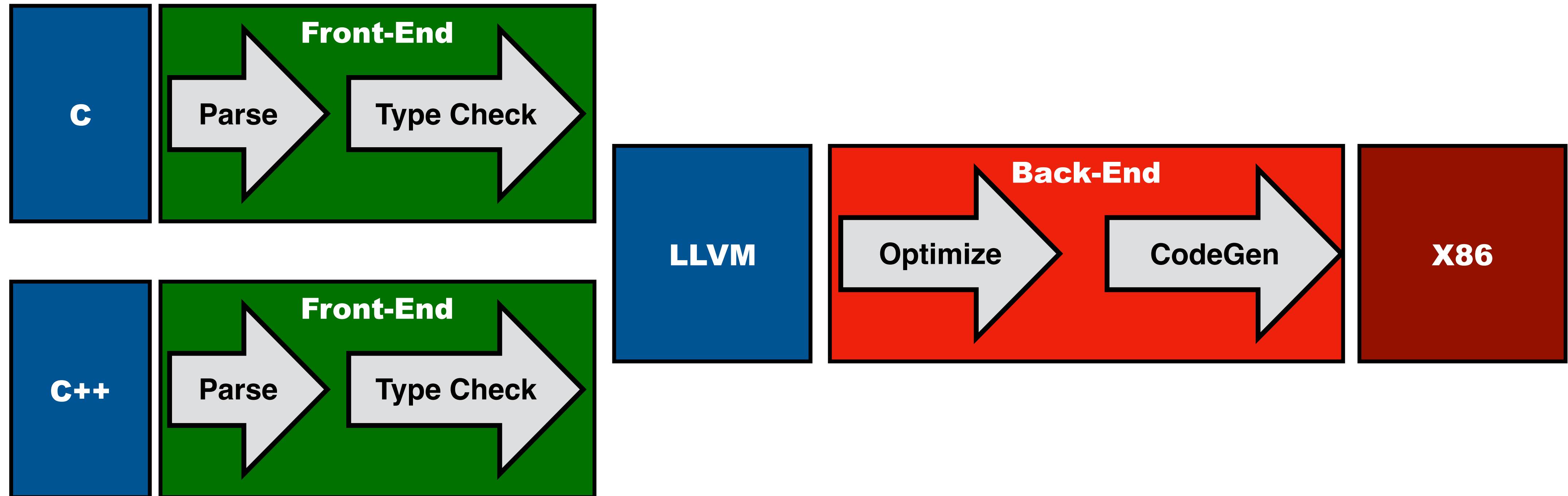
# Compiler = Front-end + Back-End



**A compiler can typically be divided in a front-end (analysis) and a back-end (synthesis)**
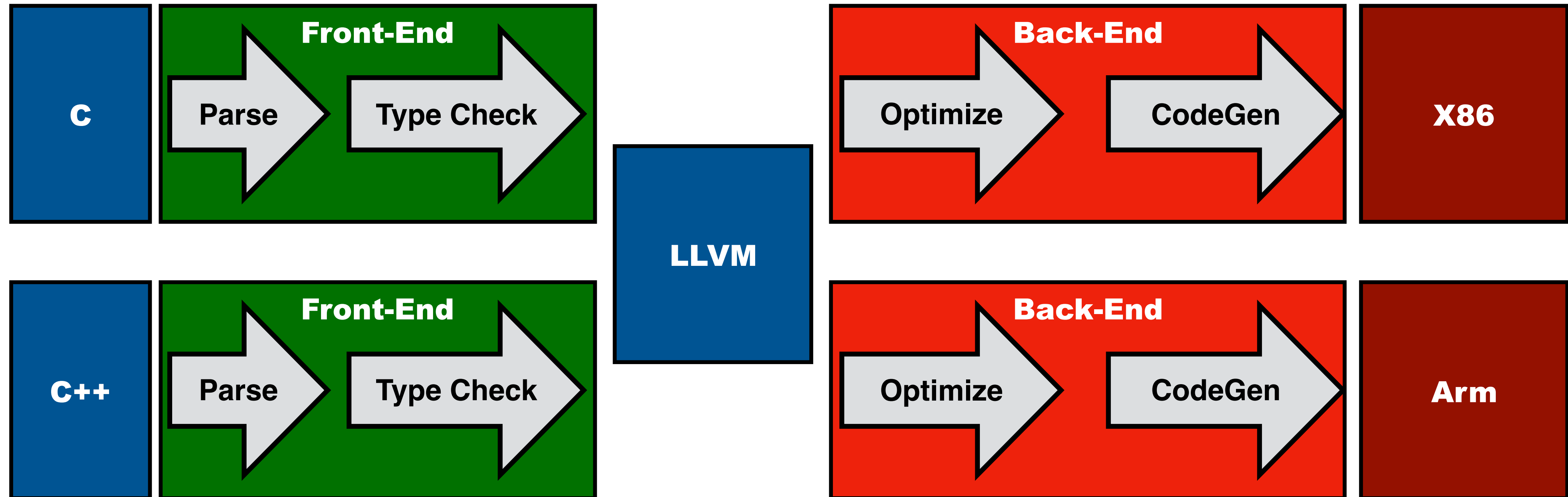
# Compiler = Front-end + Back-End



A compiler can typically be divided in a front-end (analysis) and a back-end (synthesis)

# Repurposing Back-End



**Repurposing: reuse a back-end for a different source language**

# Retargeting Compiler



| C | Front-End | | LLVM | Back-End | | X86 |
|---|---|---|---|---|---|---|
| | Parse | Type Check | | Optimize | CodeGen | |

| C++ | Front-End | | | Back-End | | Arm |
|---|---|---|---|---|---|---|
| | Parse | Type Check | | Optimize | CodeGen | |

**Retargeting: compile to different hardware architecture**

# What is a Compiler?

A bunch of components for translating programs

| Java | Parse | Type Check | Optimize | CodeGen | JVM bytecode |
|------|-------|------------|----------|---------|--------------|
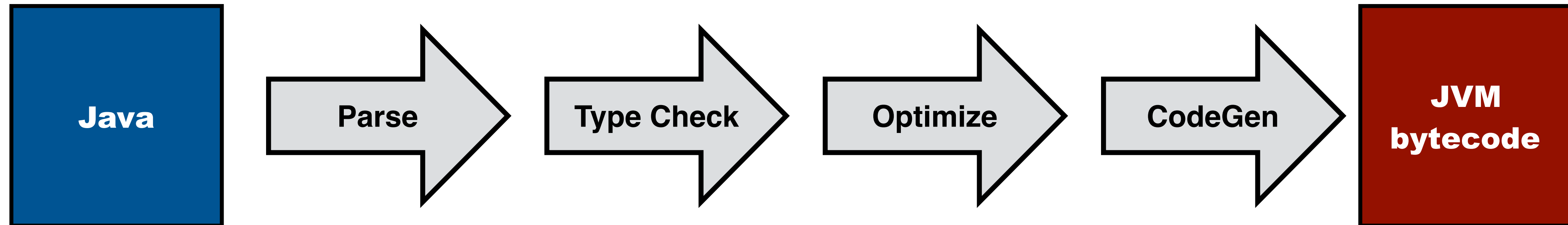
Compiler Construction = Building Variants of Java?

# Types of Compilers (1)

## Compiler

**–** translates high-level programs to machine code for a computer

## Bytecode compiler

**–** generates code for a virtual machine

## Just-in-time compiler

**–** defers (some aspects of) compilation to run time

## Source-to-source compiler (transpiler)

**–** translate between high-level languages

## Cross-compiler

**–** runs on different architecture than target architecture

# Types of Compilers (2)

## Interpreter

**–** directly executes a program (although prior to execution program is typically transformed)

## Hardware compiler

**–** generate configuration for FPGA or integrated circuit

## De-compiler

**–** translates from low-level language to high-level language

# Why Compilers?

# Programming = Instructing Computer

- fetch data from memory
- store data in register
- perform basic operation on data in register
- fetch instruction from memory
- update the program counter
- etc.

"*Computational thinking* is the thought processes involved in formulating a problem and expressing its solution(s) in such a way that a computer—human or machine—can effectively carry out."

Jeanette M. Wing. Computational Thinking Benefits Society.
In Social Issues in Computing. January 10, 2014.
http://socialissues.cs.toronto.edu/index.html
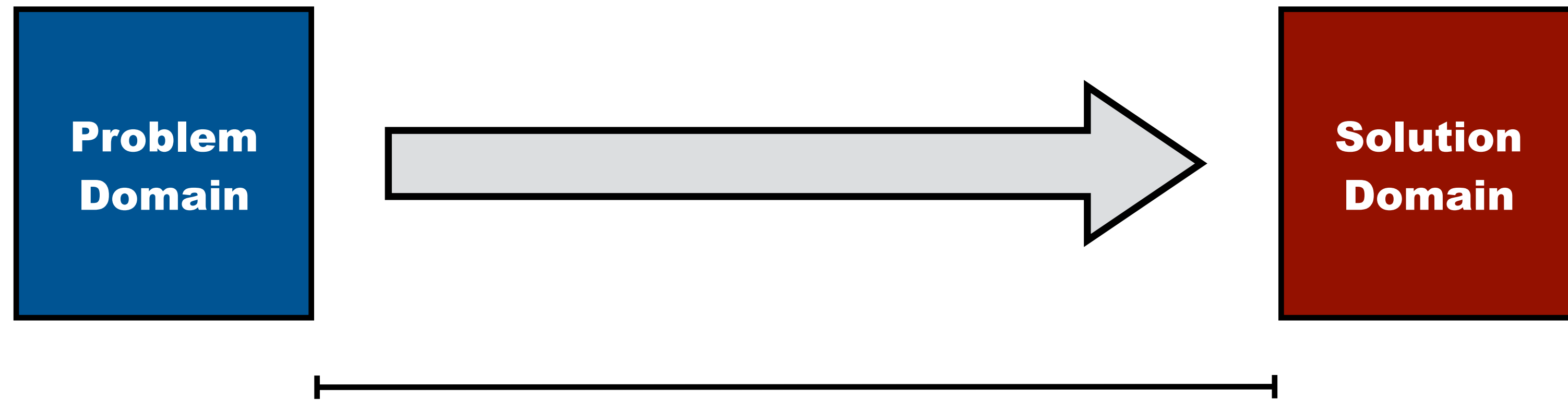
Programming is expressing intent
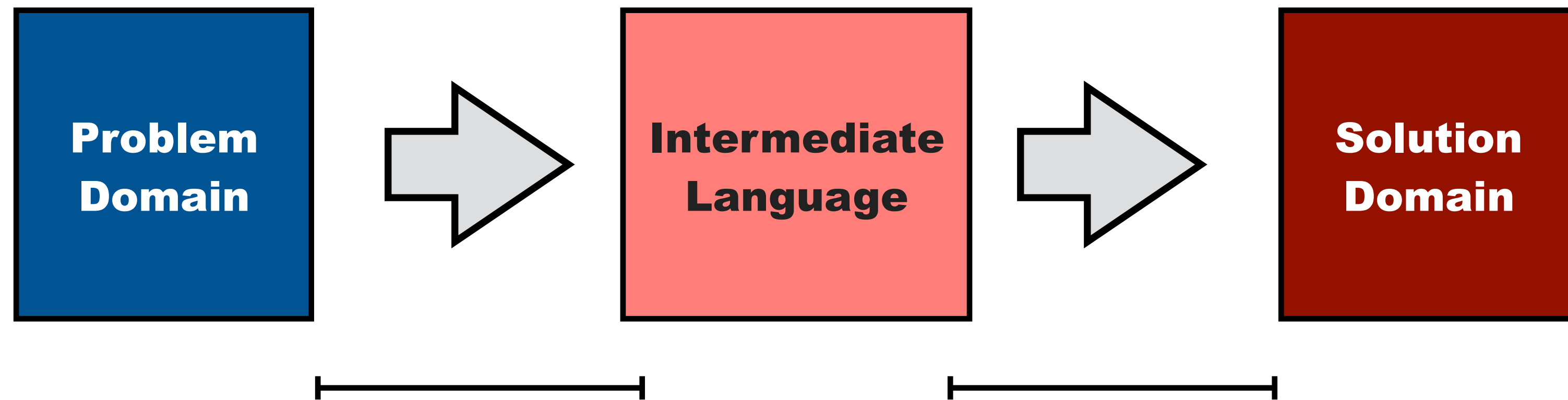
linguistic abstraction | liNG'gwistik ab'strakSHən |

noun

1.    a programming language construct that captures a programming design pattern

        o *the linguistic abstraction saved a lot of programming effort*

        o *he introduced a linguistic abstraction for page navigation in web programming*

2.    the process of introducing linguistic abstractions

        o *linguistic abstraction for name binding removed the algorithmic encoding of name resolution*

# From Instructions to Expressions

```
mov &a, &c
add &b, &c
mov &a, &t1
sub &b, &t1
and &t1,&c
```

```
c    = a
c   += b
t1   = a
t1  -= b
c   &= t1
```

```
c = (a + b) & (a - b)
```

# From Calling Conventions to Procedures

```
calc:
 push eBP              ; save old frame pointer
 mov eBP,eSP           ; get new frame pointer
 sub eSP,localsize     ; reserve place for locals
 .
 .                     ; perform calculations, leave result in AX
 .
 mov eSP,eBP           ; free space for locals
 pop eBP               ; restore old frame pointer
 ret paramsize         ; free parameter space and return
```

```
push eAX              ; pass some register result
push byte[eBP+20]     ; pass some memory variable (FASM/TASM syntax)
push 3                ; pass some constant
call calc             ; the returned result is now in eAX
```

Due to the small number of architectural registers, the x86 calling conventions mostly pass arguments on the stack, while the return value (or a pointer to it) is passed in a register.

```
def f(x)={ ... }        f(e1)
```

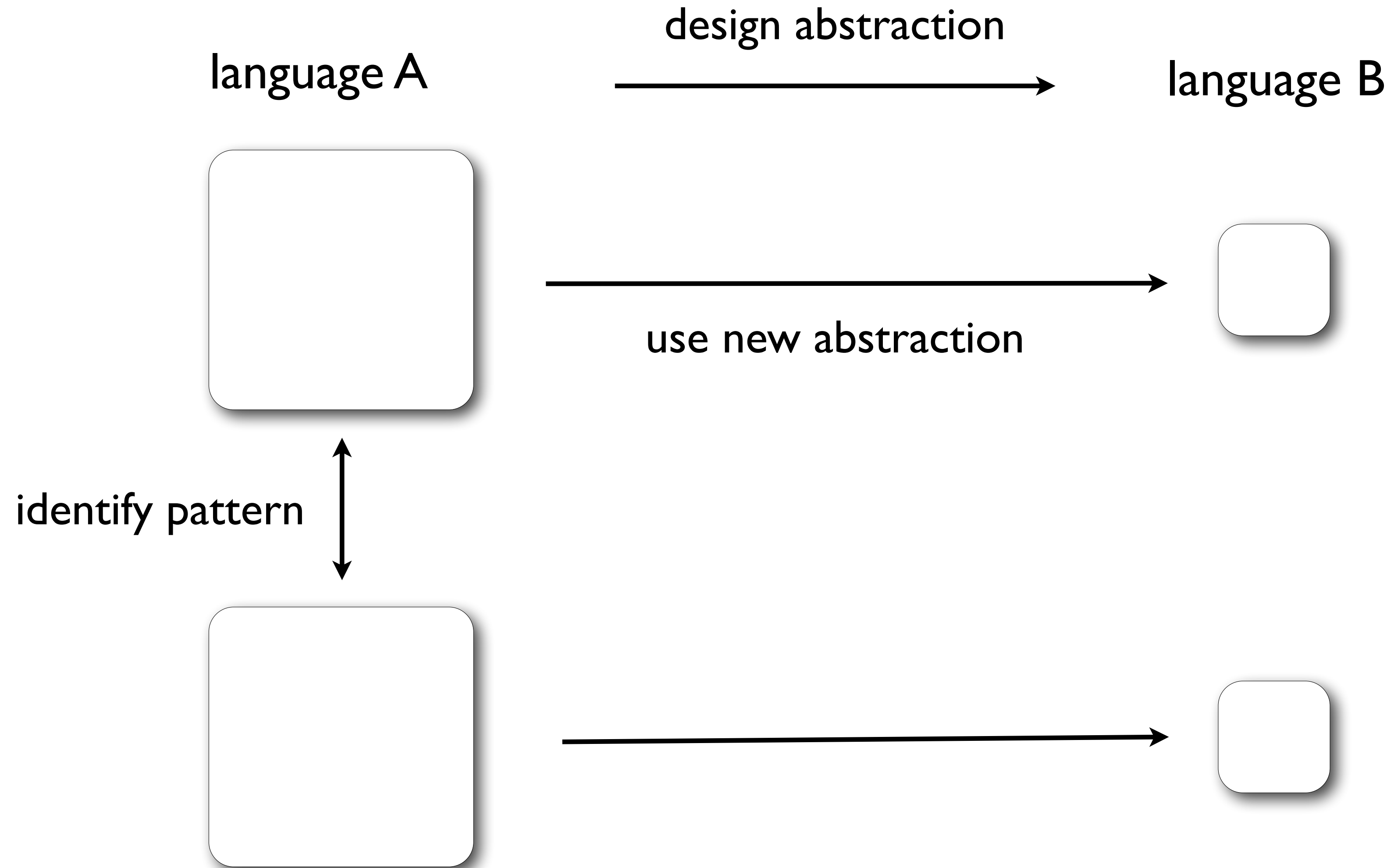function definition and call in Scala

# From Malloc to Garbage Collection

```
/* Allocate space for an array with ten elements of type int. */

int *ptr = (int*)malloc(10 * sizeof (int));
if (ptr == NULL) {
    /* Memory could not be allocated, the program
       should handle the error here as appropriate. */
} else {
    /* Allocation succeeded.  Do something.  */
    free(ptr);  /* We are done with the int objects,
                   and free the associated pointer. */
    ptr = NULL; /* The pointer must not be used again,
                   unless re-assigned to using malloc again. */
}
```
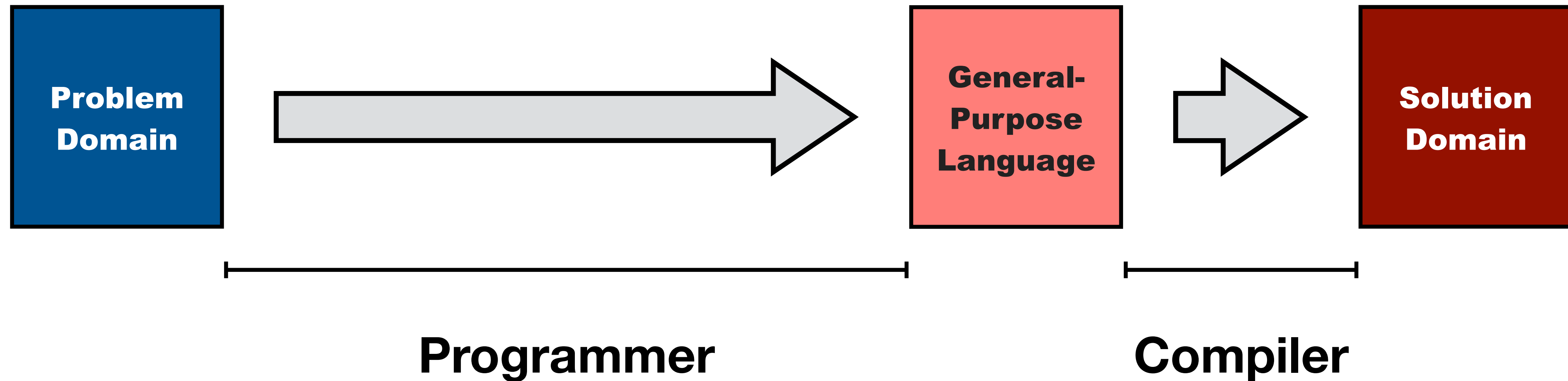
http://en.wikipedia.org/wiki/Malloc

```
        int [] = new int[10];
 /* use it; gc will clean up (hopefully) */
```

# Linguistic Abstraction

design abstraction

language A

language B

use new abstraction

identify pattern

# Compiler Automates Work of Programmer



Compilers for modern high-level languages

- Reduce the gap between problem domain and program
- Support programming in terms of computational concepts instead of machine concepts
- Abstract from hardware architecture (portability)
- Protect against a range of common programming errors

# Domain-Specific Languages

# Domains of Computation



Problem Domain → General-Purpose Language → Solution Domain
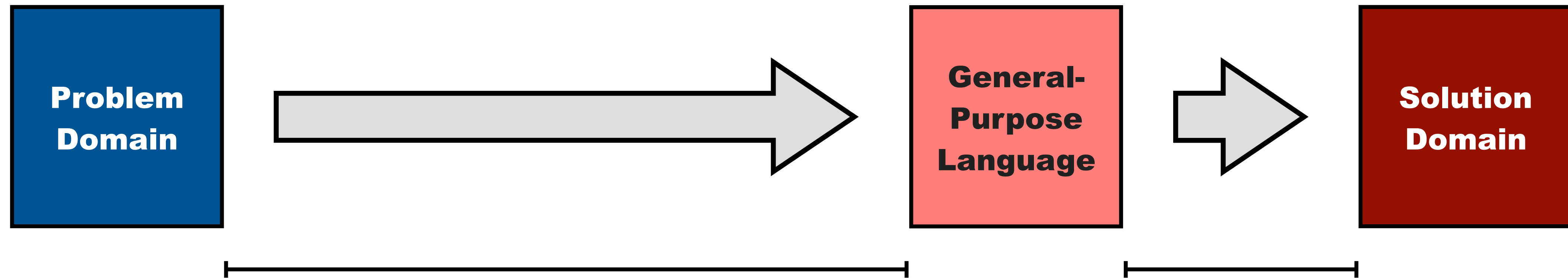
- Systems programming
- Embedded software
- Web programming
- Enterprise software
- Database programming
- Distributed programming
- Data analytics
- ...
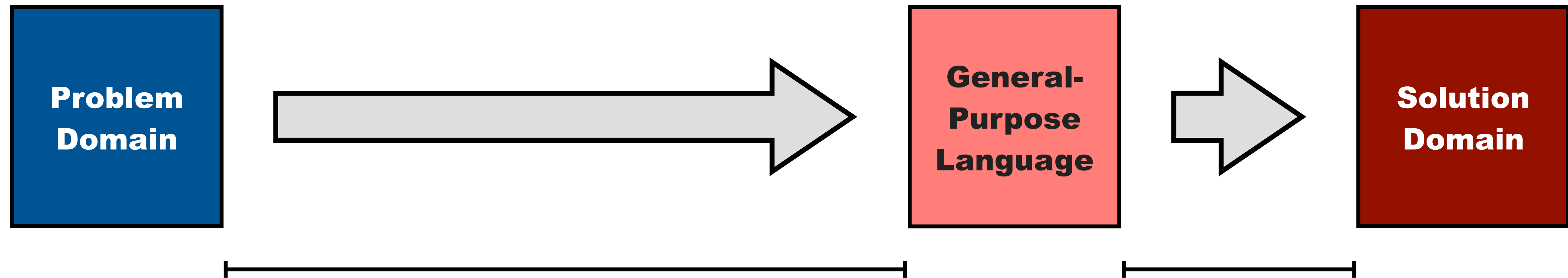
"A programming language is low level when its programs require attention to the irrelevant"

Alan J. Perlis. Epigrams on Programming.
SIGPLAN Notices, 17(9):7-13, 1982.

Problem Domain → Domain-Specific Language → General-Purpose Language → Solution Domain
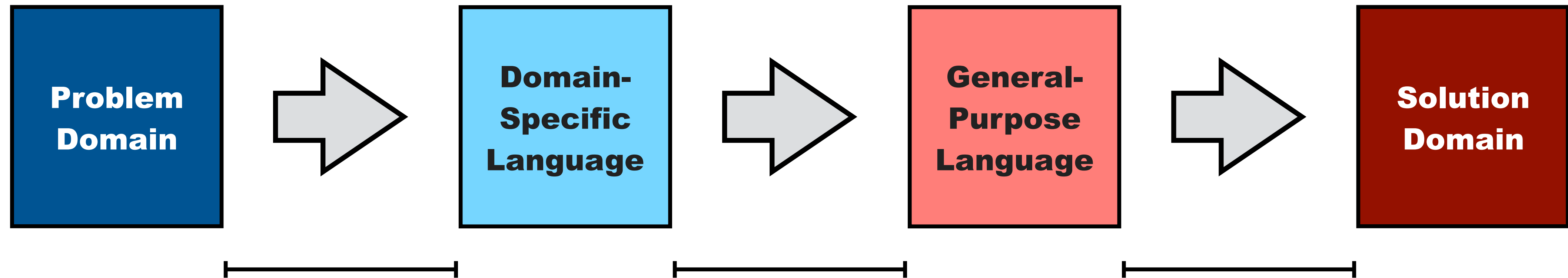
**Domain-specific language (DSL)**

noun

1. a programming language that provides notation, analysis, verification, and optimization specialized to an application domain

2. result of linguistic abstraction beyond general-purpose computation

# Language Design Methodology

## Domain Analysis

– What are the features of the domain?

## Language Design

– What are adequate linguistic abstractions?

– Coverage: can language express everything in the domain?

  ‣ often the domain is unbounded; language design is making choice what to cover

– Minimality: but not more

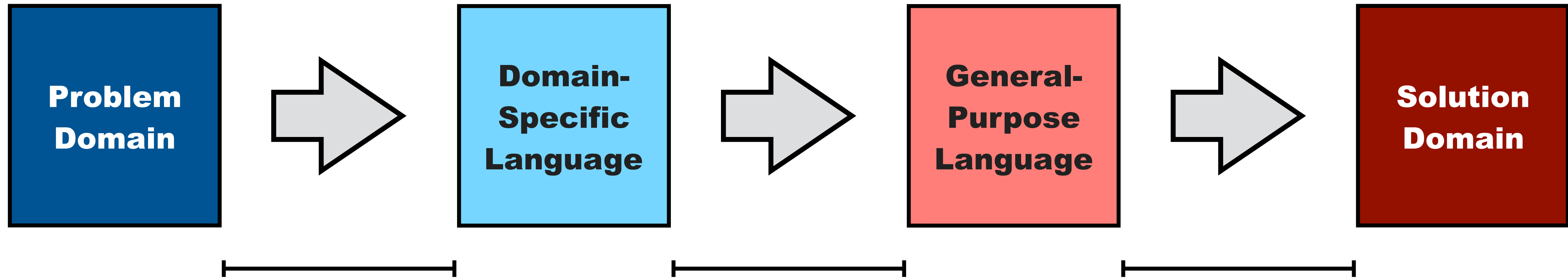  ‣ allowing too much interferes with multi-purpose goal

## Semantics

– What is the semantics of such definitions?

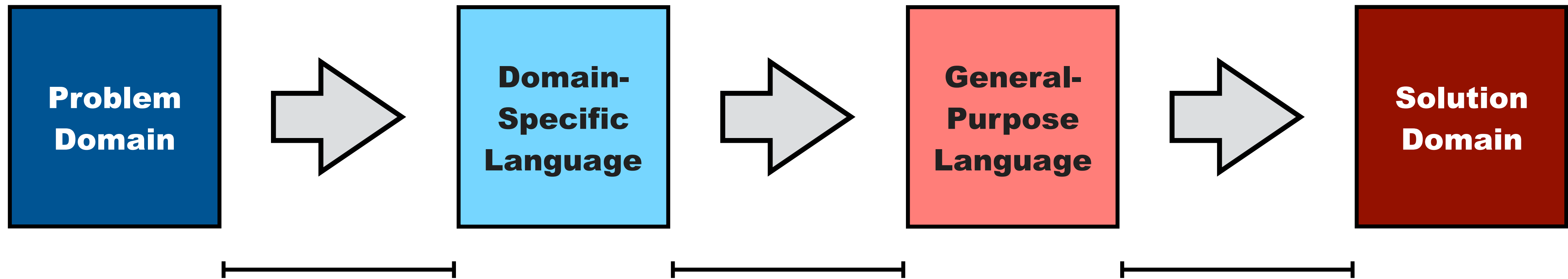– How can we verify the correctness / consistency of language definitions?

## Implementation

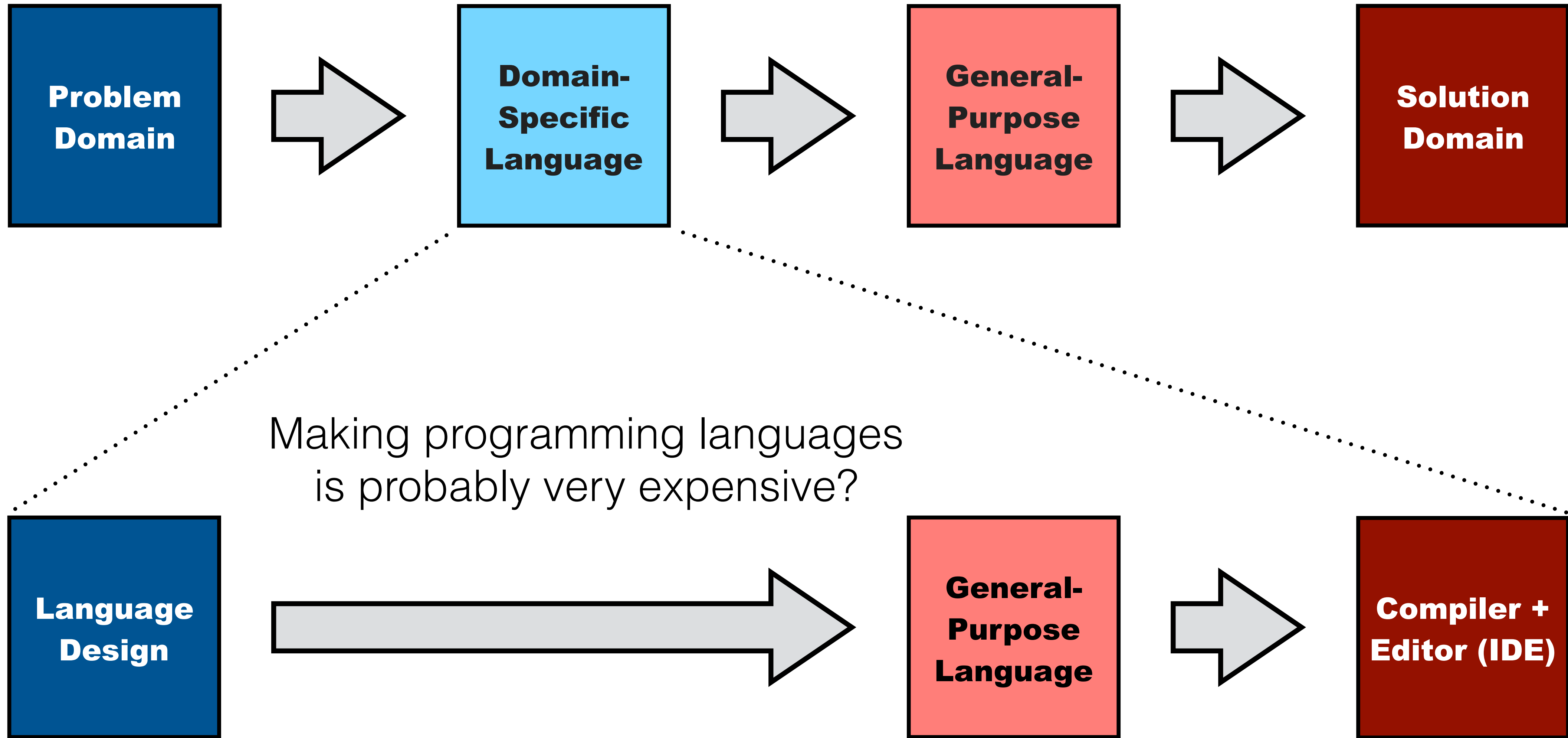– How do we derive efficient language implementations from such definitions?
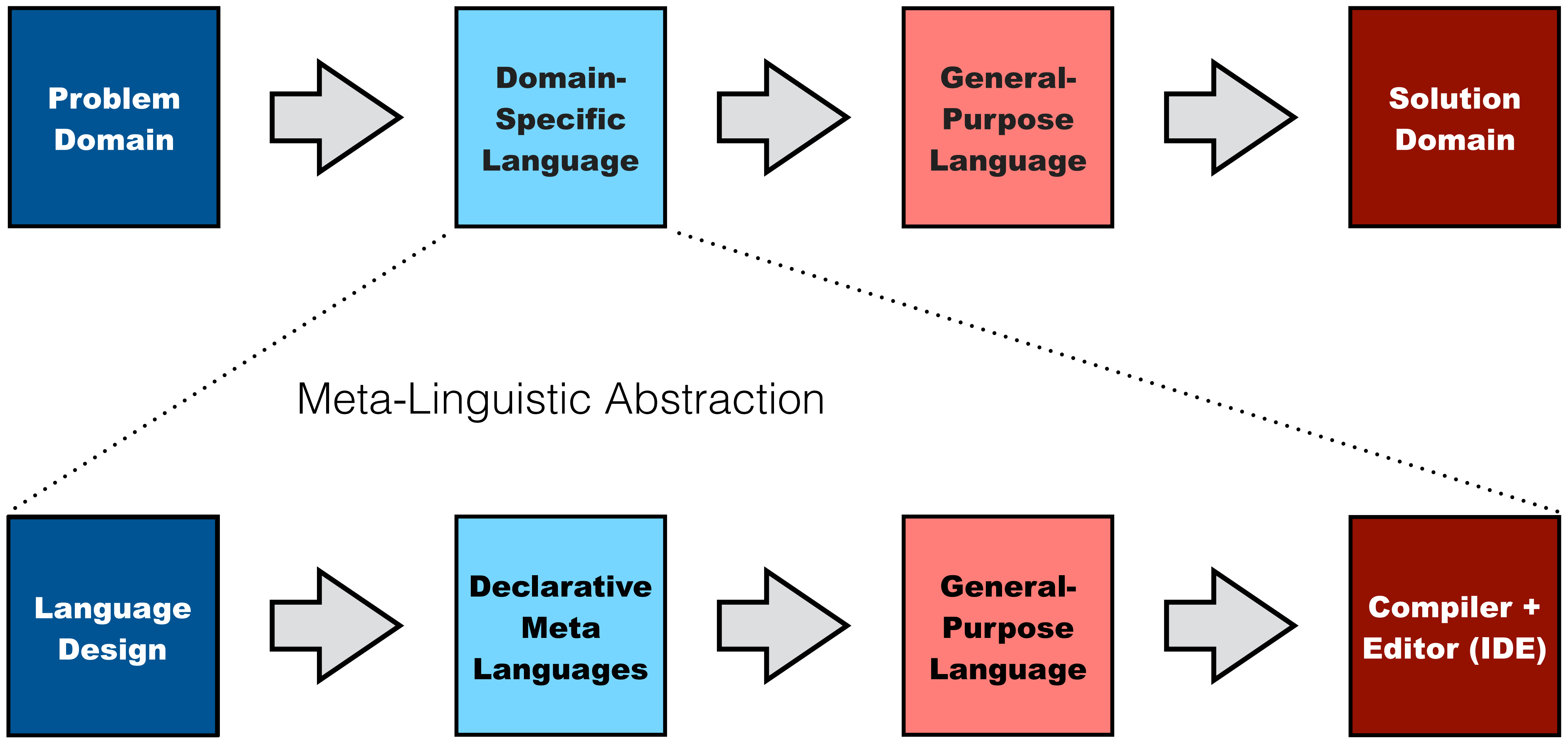
## Evaluation

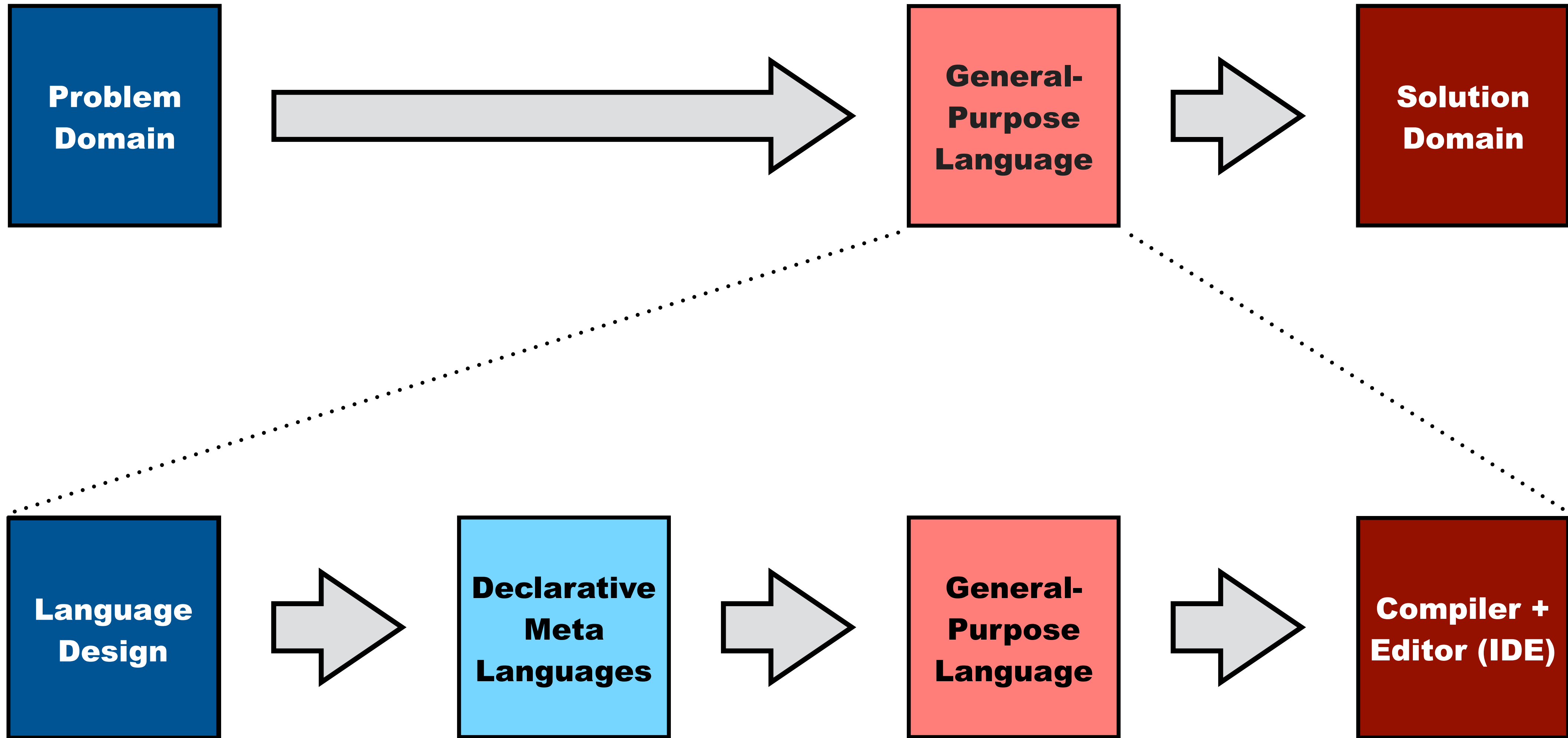– Apply to new and existing languages to determine adequacy

Problem Domain → Domain-Specific Language → General-Purpose Language → Solution Domain

Making programming languages
is probably very expensive?

Problem Domain → Domain-Specific Language → General-Purpose Language → Solution Domain

Making programming languages is probably very expensive?

Language Design → General-Purpose Language → Compiler + Editor (IDE)

64

**Problem Domain** → **Domain-Specific Language** → **General-Purpose Language** → **Solution Domain**

Meta-Linguistic Abstraction

**Language Design** → **Declarative Meta Languages** → **General-Purpose Language** → **Compiler + Editor (IDE)**

65

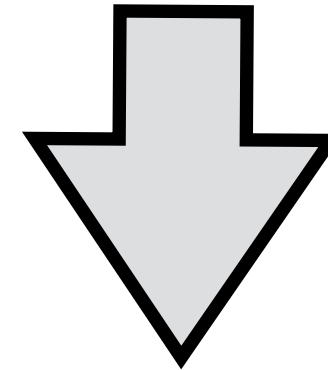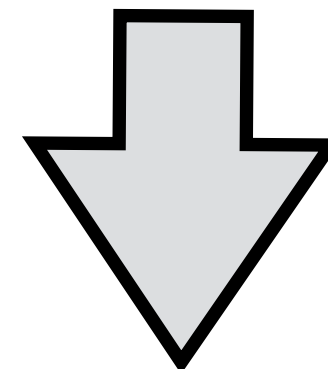**Applying compiler construction to the domain of compiler construction**

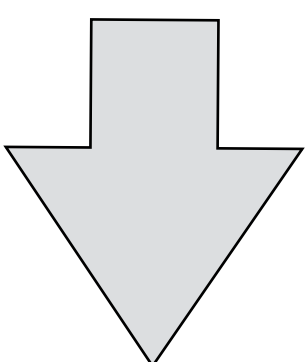That also applies to the definition of (compilers for) general purpose languages
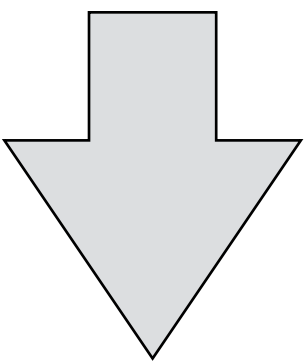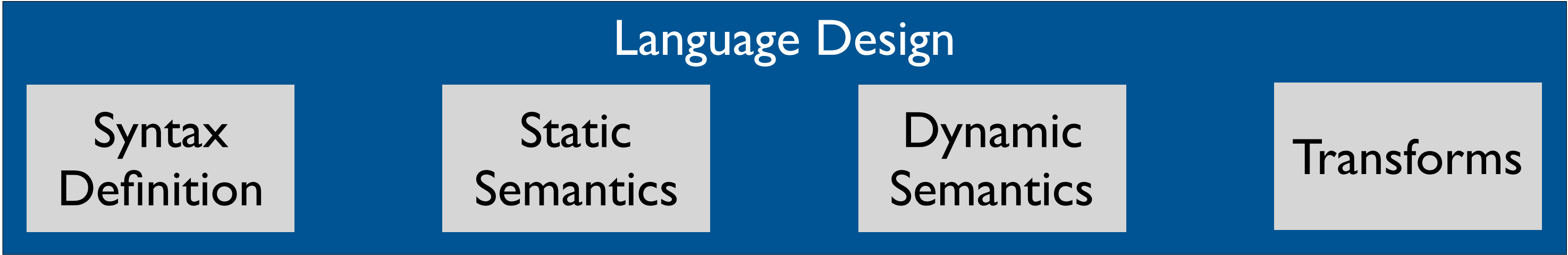
# Language Design

| Syntax Definition | Static Semantics | Dynamic Semantics | Transforms |

⬇

## Language Workbench

### Meta-DSLs

⬇

**Compiler + Editor (IDE)**

# Declarative Language Definition

## Objective

– A workbench supporting design and implementation of programming languages

## Approach

– Declarative multi-purpose domain-specific meta-languages

## Meta-Languages

– Languages for defining languages

## Domain-Specific

– Linguistic abstractions for domain of language definition (syntax, names, types, …)

## Multi-Purpose

– Derivation of interpreters, compilers, rich editors, documentation, and verification from single source

## Declarative

– Focus on what not how; avoid bias to particular purpose in language definition

# Separation of Concerns

## Representation

- Standardized representation for <aspect> of programs
- Independent of specific object language

## Specification Formalism

- Language-specific declarative rules
- Abstract from implementation concerns

## Language-Independent Interpretation

- Formalism interpreted by language-independent algorithm
- Multiple interpretations for different purposes
- Reuse between implementations of different languages