

Lecture 1: What is a Compiler?

CS4200 Compiler Construction

Eelco Visser

TU Delft

September 2019

This Lecture

Course organization

- Course team
- Study material
- Project

Introduction

- What is a compiler?
- Why do we need compilers?
- What do we study in this course?

Course Organization

Eelco Visser

Coordinates

- Full Professor
- Programming Languages Group
- (Formerly Software Engineering Research Group)
- Department of Software Technology
- Faculty of Electrical Engineering, Mathematics and Computer Science
- Delft University of Technology
- Delft, The Netherlands ([CEST/CET](#))
- Email: e.visser@tudelft.nl, visser@acm.org
- Web: <http://eelcovisser.org> (this page)

Research

I lead the [Software Language Design and Engineering](#) research program. Our mission is to enable software engineers to effectively design, implement, and apply domain-specific languages. We are doing research in three tracks:

- Language engineering: investigate the automatic derivation of efficient, scalable, incremental compilers and usable IDEs from high-level, declarative language definitions
- Semantics engineering: investigate the automatic verification of the consistency of language definitions in order to check properties such as type soundness and semantics preservation
- Language design: investigate the systematic design of domain-specific software languages with an optimal tradeoff between expressivity, completeness, portability, coverage, and maintainability.

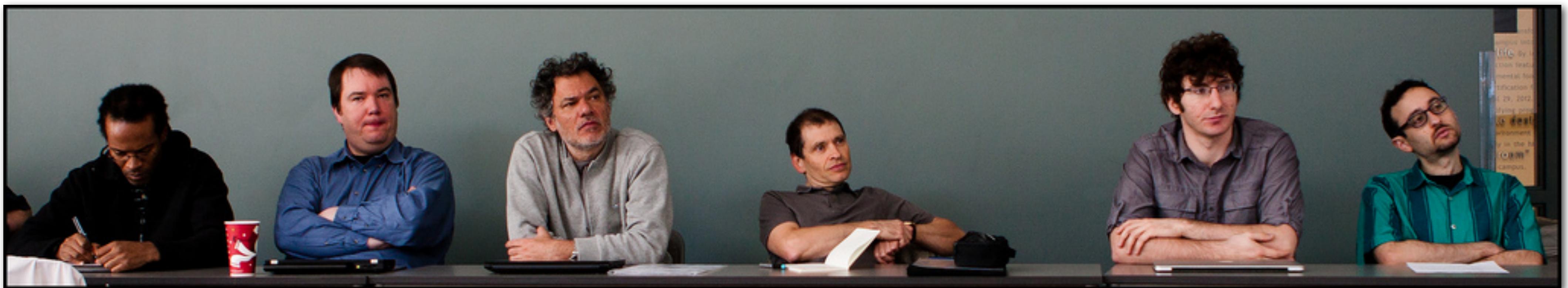
Research overview

Projects

- [The Language Designer's Workbench](#)
- [Spoofax](#): a language workbench
- [SDF3](#): a syntax definition formalism
- [NaBL](#): a name binding language
- [DynSem](#): a dynamic semantics specification language
- [Stratego](#): a program transformation language
- [WebDSL](#): a web programming language
- [researchr](#): a bibliography management system
- [researchr/conf](#): a domain-specific content management system for conferences
- [WebLab](#): a learning management system



Project overview



No use of electronic devices during lectures

Team Members



Eelco Visser

responsible instructor

Building 28. E4.340 Email Twitter



Jasper Denkers

graduate teaching assistant

Building 28. E4.420 Email



Jeff Smits

graduate teaching assistant

Building 28. E4.420 Email



Hendrik van Antwerpen

graduate teaching assistant

Building 28. E4.420 Email



Bram Crielaard

student teaching assistant

Email



Ana Šemrov

student teaching assistant

Email

Contact

Handling questions by e-mail is very inefficient, which is why we try to avoid it as much as possible. Lectures and lab sessions are natural points of contacts with instructors and teaching assistants. We also offer walk-in hours. For longer discussions, we prefer appointments outside our walk-in hours. To make an appointment, please send the responsible instructor an email with

- a detailed description of your problem: Where are you struggling? What questions do you have? How can we help you?
- some alternative dates: We prefer appointments in the morning. Keep in mind that we will need some time to prepare the appointment.

You will receive an email with

- some tasks for you to prepare the appointment,
- the date of our appointment, which you are asked to confirm.

Walk-in Hours 2019-2020

We offer walk-in hours on Wednesday mornings 9:30-11:00 on the fourth floor of Building 28, where you can find us next to the elevators.

**Slack #spoofax-users channel
for questions about Spoofax
=> send me email**

Brightspace: Announcements

The screenshot shows a Brightspace course page for "CS4200-A Compiler Construction (2019/20 Q1)". The top navigation bar includes the TU Delft logo, course title, user profile (Eelco Visser), and various icons for messaging, notifications, and settings.

Announcements

- Take a notebook to lecture** (Posted 04 September, 2019 21:55)

Tomorrow is the first lecture of the Compiler Construction course (CS4200 A and B) at 10:45 in lecture hall Boole.
I do not allow the use of electronic devices in my lectures. If you would like to take notes during the lecture, I advise you to take a paper notebook and pen.
- Compiler Construction Course in 2019-2020** (Posted 26 August, 2019 13:40)

This course runs in Q1 and Q2.
The course consists of two parts CS4200-A (Compiler Construction), and CS4200-B (Compiler Construction Project), which run in parallel.
The study guide information for the courses is https://studiegids.tudelft.nl/a101_displayCourse.do?course_id=48294 and https://studiegids.tudelft.nl/a101_displayCourse.do?course_id=48295.
The website for the courses provides information about lectures, homework, and project assignments. The site for this academic year is not yet online, but it will be a copy with adaptions of last year's site: <https://tudelft-cs4200-2018.github.io/>.
The first lecture for Compiler Construction (CS4200) is on Thursday, September 5 at 10:45 in Lecture Hall Boole.
The second lecture will be on Friday, September 6 at 13:45 in Lecture Hall Pi. This lecture is instead of the lab for CS4200-B on Friday afternoon.

Calendar

- Wednesday, 4 September, 2019
- Upcoming events
There are no events to display.
Create an event.

Updates

- There are no current updates for CS4200-A Compiler Construction (2019/20 Q1)

Course Website

TUDELFT CS4200 LECTURES ASSIGNMENTS PROJECT CONTACT



Photo Credit: Wikimedia Commons

Compiler Construction

Course Contents

Compilers translate the source code of programs in a high-level programming language into executable (virtual) machine code. Nowadays, compilers are typically integrated into development environments providing features like syntax highlighting, content assistance, live error reporting, and continuous target code generation.

This course studies the architecture of compilers and interactive programming environments and the concepts and techniques underlying the components of that architecture. For each of the components of a compiler we study the formal theory underlying the language aspect that it covers, declarative specification languages to define compiler components, and the techniques for their implementation. The concepts and techniques are illustrated by application to small languages or language fragments.

The course covers the following topics:



Compiler Construction

- [Twitter](#)
- [Github](#)

[Edit on GitHub](#)

CS4200: Two Courses

CS4200-A: Compiler Construction (5 ECTS)

- Study concepts and techniques
- Lectures
- Papers
- Homework assignments: apply to small problems
- Assessment: exams in November and January

CS4200-B: Compiler Construction Project (5 ECTS)

- Build a compiler and programming environment for a subset of Java
- Assessment: code submitted for lab assignments

CS4200-A: Compiler Construction

Concepts and Techniques



Compiler Construction

- [Twitter](#)
- [Github](#)

[Edit on GitHub](#)

Lectures

(Tentative) Schedule 2019-2020

Q1

- Thu Sep 5, 10:45-12:30, EWI Boole: [What is a compiler?](#)
- Fri Sep 6, 13:45-15:30, EWI Pi: [Syntax Definition](#)
- Thu Sep 12, 10:45-12:30, EWI Boole: [Parsing](#)
- Thu Sep 19, 17:45-19:30, EWI Boole: [Syntactic \(Editor\) Services](#)
- Thu Sep 26, 10:45-12:30, EWI Boole: [Transformation](#)
- Thu Oct 3, 10:45-12:30, EWI Boole: [Introduction to Static Analysis](#)
- Thu Oct 10, 10:45-12:30, EWI Boole: [Type Checking](#)
- Thu Oct 17, 10:45-12:30, EWI Boole: [Specifications with NaBL2](#)
- Thu Oct 24, 10:45-12:30, EWI Boole: [Constraint Resolution](#)
- Thu Oct 31, 13:30-16:30, IO-PC hall 2 (CTRL): Exam Q1

Q2

- Wed Nov 13, 15:45-17:30, EWI Boole: [Data-flow Analysis](#)
- Wed Nov 20, 15:45-17:30, EWI Boole: [Monotone Frameworks](#)
- Wed Nov 27, 15:45-17:30, EWI Boole: [Virtual Machines](#)
- Wed Dec 4, 15:45-17:30, EWI Boole: [Code Generation](#)
- Wed Dec 11, 15:45-17:30, EWI Boole: [Interpreters](#)
- Wed Dec 18, 15:45-17:30, EWI Boole: [Memory Management](#)
- Wed Jan 8, 15:45-17:30, EWI Boole: Reserve
- Wed Jan 15, 15:45-17:30, EWI Boole: [Beyond Compiler Construction](#)
- Fri Jan 31, 13:30-16:30, IO-PC hall 3 (SHIFT): Exam Q2

CS4200-A: Homework Assignments

TUDELFT CS4200

LECTURES ASSIGNMENTS PROJECT CONTACT

Syntax Transformation Statics Dynamics



Compiler Construction

[Twitter](#) [Github](#)

[Edit on GitHub](#)

Homework Assignments

To study the material of the lectures we provide homework assignments to be submitted to [WebLab](#).

Syntax

- Week 1.1
- Week 1.2
- Week 1.3

Transformation

- Week 1.4

Statics

- Week 1.5
- Week 1.6
- Week 1.7
- Week 1.8
- Week 2.1
- Week 2.2

Dynamics

- Week 2.3: Virtual Machines
- Week 2.4: Code Generation
- Week 2.5: Interpreters
- Week 2.6: Memory Management

WebLab for Homework, Exams, Grade Registration

The screenshot shows the WebLab interface for the course CS4200, 2019-2020. The top navigation bar includes links for WebLab, Courses, Cohorts, About, Admin, Elco Visser, and Sign out. Below the navigation is a header with the course code CS4200 and year 2019-2020. A toolbar below the header includes Course Edition (highlighted in blue), News, Course Rules, Students, Course Groups, Edit Staff, Edit Edition, and Docker Settings.

Compiler Construction

Course: CS4200 Edition: 2019-2020

From September 1, 2019

Course Information

- Home
- All editions
- News archive
- Course rules
- Assignments
- Enrolled students

Enroll

One can enroll until Mon, Sep 30, 2019 12:00

[Enroll](#)

Course staff

Lecturers

- Jeff Smits
- Eelco Visser
- Hendrik van Antwerpen

Assistants

- Bram Crielgaard
- Ana Šemrov

[Configure Staff](#)

About the Course

Compilers translate the source code of programs in a high-level programming language into executable (virtual) machine code. Nowadays, compilers are typically integrated into development environments providing features like syntax highlighting, content assistance, live error reporting, and continuous target code generation.

This course studies the architecture of compilers and interactive programming environments and the concepts and techniques underlying the components of that architecture. For each of the components of a compiler we study the formal theory underlying the language aspect that it covers, declarative specification languages to define compiler components, and the techniques for their implementation. The concepts and techniques are illustrated by application to small languages or language fragments.

See <https://tudelft-cs4200-2019.github.io/> for lectures, assignment, and project information.

[Edit](#)

Active Assignments

(*) This is only visible for course managers and administrators

Assignments with most submission edits from past 6 hours

[Load Assignments with most submission edits from past 6 hours](#)

Upcoming Deadlines

No upcoming deadlines

News

[Archive](#)

<https://weblab.tudelft.nl/cs4200/2019-2020/>

Sign in to WebLab using “Single Sign On for TU Delft”

The screenshot shows the WebLab login page. At the top, there is a blue header bar with the "WebLab" logo, "Courses", "About", and "Sign in" links. Below the header, the main content area has a white background. It features a large "Welcome to WebLab" heading. Underneath it, a message says: "If you are a student or lecturer at TU Delft you should use". A blue button labeled "Single Sign On for TU Delft" is highlighted with a red circle. Below this, another message states: "If this is the first time you are using WebLab, an account will be created automatically, linked to your netid." To the left, there is a "Sign in" section with fields for "Username or Email" and "Password", and buttons for "Sign in" and "Forgot password". To the right, there is a "Register (for Non-TU Delft Students)" section, which is crossed out with a large red X. This section includes fields for "Email", "First Name", "Last Name", "Username", "Password", and "Password" (repeated), and a "Register" button.

WebLab Courses About Sign in

Welcome to WebLab

If you are a student or lecturer at TU Delft you should use

Single Sign On for TU Delft

If this is the first time you are using WebLab, an account will be created automatically, linked to your netid.

Sign in

Username or Email

Password

Sign in Forgot password

Register (for Non-TU Delft Students)

Email

First Name

Last Name

Username

Password

Password

Register

Enroll for Course CS4200 in WebLab

The screenshot shows the WebLab interface for the CS4200 course in the 2019-2020 edition. The top navigation bar includes links for WebLab, Courses, Cohorts, About, Admin, Eelco Visser, and Sign out. The main content area displays the course title 'Compiler Construction'. A sidebar on the left contains 'Course Information' with links to Home, All editions, News archive, Course rules, Assignments, and Enrolled students. The 'Enroll' section features a message about enrollment closing on Sep 30, 2019, and a prominent blue 'Enroll' button, which is circled in red. Below this are sections for 'Course staff' (Lecturers: Jeff Smits, Eelco Visser, Hendrik van Antwerpen; Assistants: Bram Crielgaard, Ana Šemrov) and 'Configure Staff'. The main content area includes sections for 'About the Course' (describing compilers), 'Active Assignments' (with a note for course managers/administrators), 'Assignments with most submission edits from past 6 hours' (Load Assignments), 'Upcoming Deadlines' (No upcoming deadlines), and a 'News' section (Archive).

<https://weblab.tudelft.nl/cs4200/2019-2020/>

Academic Misconduct

Academic Misconduct

All actual, detailed work on assignments must be **individual work**. You are encouraged to discuss assignments, programming languages used to solve the assignments, their libraries, and general solution techniques in these languages with each other. If you do so, then you must **acknowledge** the people with whom you discussed at the top of your submission. You should not look for assignment solutions elsewhere; but if material is taken from elsewhere, then you must **acknowledge** its source. You are not permitted to provide or receive any kind of solutions of assignments. This includes partial, incomplete, or erroneous solutions. You are also not permitted to provide or receive programming help from people other than the teaching assistants or instructors of this course. Any violation of these rules will be reported as a suspected case of fraud to the Board of Examiners and handled according to the EEMCS Faculty's fraud procedure. If the case is proven, a penalty will be imposed: a minimum of exclusion from the course for the duration of one academic year up to a maximum of a one-year exclusion from all courses at TU Delft. For details on the procedure, see Section 2.1.26 in the faculty's Study Guide for MSc Programmes.

DON'T!

CS4200-B: Compiler Construction Project

Building a compiler and IDE for MiniJava

Project Assignments

TUDELFT CS4200

LECTURES

ASSIGNMENTS

PROJECT

CONTACT



Photo Credit: Wikimedia Commons



Compiler
Construction

Compiler Construction

Course Contents

Compilers translate the source code of programs in a high-level programming language into executable (virtual) machine code. Nowadays, compilers are typically integrated into development environments providing features like syntax highlighting, content assistance, live error reporting, and continuous target code generation.

<https://tudelft-cs4200-2019.github.io>

Download Special Spofax Version for the Course!

TUDELFT CS4200

LECTURES ASSIGNMENTS PROJECT CONTACT

Syntax Statics Dynamics Documentation

Git Spofax Jasmin



Compiler Construction

[Twitter](#) [Github](#)

[Edit on GitHub](#)

Spofax

We use the [Spofax language workbench](#) in the lab to develop a MiniJava editor.

The Spofax Language Workbench supports the definition of all aspects of textual languages, such as syntax, static analysis, and compilation, using high-level, declarative meta-languages. From a language definition using these meta-languages, Spofax generates full-featured Eclipse and IntelliJ editor plugins, as well as a command-line interface. The generated editors include syntax highlighting, syntax checking, parse error recovery, error markers for syntactic and semantic errors, and custom operations, such as invoking an interpreter or compiler.

Downloading

In this lab, we will be using the Spofax Eclipse plugin. You can download an Eclipse installation with the Spofax Eclipse plugin preinstalled from our build farm:

- [Windows 32-bits](#)
- [Windows 64-bits](#)
- [Linux 32-bits](#)
- [Linux 64-bits](#)
- [Mac OS X \(Intel only\)](#)

These are specifically tagged versions of Spofax, suited for the lab assignments. Please use these versions of Spofax, instead of the ones found on the Spofax website.

To install and run Spofax, read the [Installing Spofax](#) instructions, starting from the Unpack section.

OVERVIEW

Downloading
Updating
Documentation

<https://tudelft-cs4200-2019.github.io/documentation/spofax.html>



Compiler Construction

Twitter

Github

[Edit on GitHub](#)

Project

In the project part of the course (CS4200-B), you build a compiler and IDE for MiniJava, a small subset of Java. We split this task into three milestones, each consisting of smaller ‘labs’:

- Milestone 1: Syntax Analysis
 - Testing Syntax Analysis
 - Syntax Definition
 - Simple Term Rewriting
- Milestone 2: Semantic Analysis
 - Testing Name Analysis
 - Name Analysis
 - Testing Type Analysis
 - Type Analysis
 - Data-Flow Analysis
- Milestone 3: Code Generation
 - Compiling Minimal Programs
 - Compiling Expressions and Statements
 - Compiling Fields, Parameters, and Variables

OVERVIEW

[Submission](#)

[Grades](#)

[Deadlines 2018-2019](#)

[Academic Misconduct](#)

[Retry](#)



Compiler Construction

- [Twitter](#)
- [GitHub](#)

[Edit on GitHub](#)

Overview

Objectives

1. Specify rewrite rules `to-outline-label` which map AST nodes to labels in an outline view. You should include:
 - classes (class name and, if available, parent class name),
 - fields (field name and type),
 - methods (method name, parameter types, return type) and
 - local variables (variable name and type).
2. Define rewrite rules `desugar` which desugar
 - unary expressions into terms of the form `UnExp(op, exp)`,
 - binary expressions into terms of the form `BinExp(op, exp1, exp2)` and
 - octal numbers into decimal numbers.
3. Integrate `desugar` into a strategy `desugar-all` which desugars subtrees in an AST.

For grading, it is required to comply with all constructor names, rule names, and strategy names literally.

OVERVIEW

- Overview
- Objectives
- Submission
- Grading
- Early Feedback
- Detailed Instructions
- Preliminaries
- GitHub Repository
- Anatomy of a Spoofax Project
- Initial Editor Project
- Signature
- Outline View
- Rewrite Rules
- Naming Conventions
- String Interpolation
- Annotations
- Challenge
- Desugaring
- Signature
- Rewrite Rules
- Editor Integration
- Strategies
- Challenge

Spoofax Documentation

The screenshot shows the official Spoofax documentation website. The left sidebar contains a navigation menu with sections like 'The Spoofax Language Workbench', 'TUTORIALS', 'REFERENCE MANUAL', 'RELEASES', and 'CONTRIBUTIONS'. The main content area is titled 'The Spoofax Language Workbench' and describes it as a platform for developing textual (domain-specific) programming languages. It lists several key ingredients: meta-languages for declarative language definition, an interactive environment for developing languages, code generators for various tools, and generation of Eclipse and IntelliJ editor plugins. Below this, a section on 'Developing Software Languages' is introduced, mentioning that Spoofax supports textual languages and has been used for various kinds of languages. Two specific categories are highlighted: 'Programming languages' (for computer programming) and 'Domain-specific languages' (for capturing domain understanding). The footer of the page includes a link to the website's URL.

Docs » The Spoofax Language Workbench [Edit on GitHub](#)

The Spoofax Language Workbench

Spoofax is a platform for developing textual (domain-specific) programming languages. The platform provides the following ingredients:

- Meta-languages for high-level declarative language definition
- An interactive environment for developing languages using these meta-languages
- Code generators that produce parsers, type checkers, compilers, interpreters, and other tools from language definitions
- Generation of full-featured Eclipse editor plugins from language definitions
- Generation of full-featured IntelliJ editor plugins from language definitions (experimental)
- An API for programmatically combining the components of a language implementation

With Spoofax you can focus on the essence of language definition and ignore irrelevant implementation details.

Developing Software Languages

Spoofax supports the development of *textual* languages, but does not otherwise restrict what kind of language you develop. Spoofax has been used to develop the following kinds of languages:

Programming languages
Languages for programming computers. Implement an existing programming language to create an IDE and other tools for it, or design a new programming language.

Domain-specific languages
Languages that capture the understanding of a domain with linguistic abstractions. Design a DSL for your domain with a compiler that generates code that would be tedious and error prone to produce manually.

Assignment Submission

Private GitLab repository per student

- <https://gitlab.ewi.tudelft.nl/CS4200-B/2019-2020/>

Submit by Merge Request

Multiple submissions possible

Limited early feedback provided

Details online:

- <https://tudelft-cs4200-2019.github.io/project/#submission>

Assignment Deadlines

Deadlines 2019-2020

In Q1 and Q2 the lab is on Friday. The deadlines for the assignments (labs) are listed below and are at 23:59 on that date. All assignments except lab 10 have a 24 hour deadline extension with a penalty of 1 point (off the final grade). Lab 10 has an extension until 11/01 with a penalty of 1 point.

These dates are subject to change.

- Milestone 1: Syntax Analysis
 - 28/09: Lab 1: Testing Syntax Analysis
 - 28/09: Lab 2: Syntax Definition
 - 05/10: Lab 3: Simple Term Rewriting
- Milestone 2: Semantic Analysis
 - 26/10: Lab 4: Testing Name Analysis
 - 26/10: Lab 5: Name Analysis
- Milestone 2: Semantic Analysis (continued)
 - 23/11: Lab 6: Testing Type Analysis
 - 23/11: Lab 7: Type Analysis
 - 30/11: Lab 8: Data-flow Analysis
- Milestone 3: Code Generation
 - 07/12: Lab 9: Compiling Minimal Programs
 - 21/12: Lab 10: Compiling Expressions and Statements
 - 18/01: Lab 11: Compiling, Fields, Parameters, and Variables

Project grades

To pass you need to meet all of these criteria:

- Each assignment grade ≥ 4
- Each milestone average ≥ 5
- Overall average ≥ 6

Project Grades and Registration

The screenshot shows the assignment management interface for the CS4200-B course. The top navigation bar includes links for CS4200, 2018-2019, CS4200:...18-2019, Actions (with items 1 and 2 selected), and various tabs like Assignment, Answer, Statistics/Dates (which is active), Submissions, Edit Assignment, Manage Assignment Tree, and Discussions.

The main content area is divided into two sections: 'Assignment Info' on the left and 'Assignment Tree' on the right.

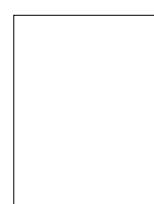
Assignment Info:

- Description:** Compiler Construction Project
- Assignment Info:** The lab assignments are divided into three milestones, which are further divided in lab assignments that should be submitted separately.

Assignment Tree:

Assignment	Weight	Started	Completed	Passed	Mean	Due
CS4200-B	50.0 of total 100.0					
TU Delft NetID and GitLab registration	0.0 of total 12.0					
Milestone 1: Syntax	4.0 of total 12.0					
Milestone 2: Semantic Analysis	5.0 of total 12.0					
Milestone 3: Code Generation	3.0 of total 12.0					

Each assignment row has up and down arrows for reordering, and a refresh icon.



<https://weblab.tudelft.nl/cs4200/2019-2020/assignment/20207/info>

Contact hours

Lab on Fridays. 13:45-17:45 DW-PC 2

Walk-in hours on Wednesdays. 9:30-11:00 E4.420 (Building 28)



<https://tudelft-cs4200-2018.github.io/team/>

What is a Compiler?

Etymology

Latin

Etymology

From [con-](#) (“with, together”) + [pīlō](#) (“ram down”).

Pronunciation

- ([Classical](#)) IPA^(key): /kɒm'piː.loʊ/, [kɒm'piː.tɔː]

Verb

compīlō (present infinitive [compīlāre](#), perfect active [compīlāvī](#), supine [compīlātūm](#)); [first conjugation](#)

1. I [snatch](#) together and [carry](#) off; [plunder](#), [pillage](#), [rob](#), [steal](#).

Dictionary

English

Verb

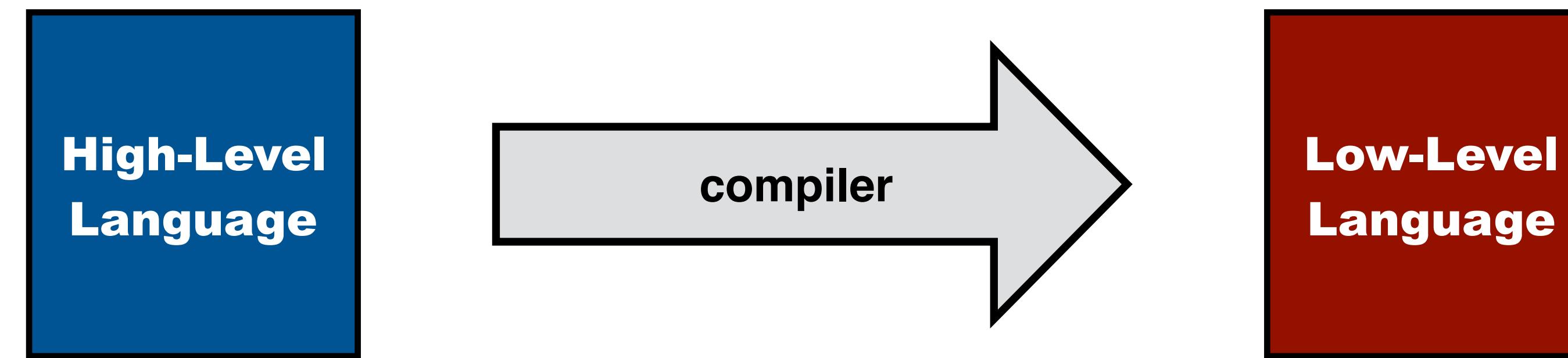
compile (*third-person singular simple present [compiles](#), present participle [compiling](#), simple past and past participle [compiled](#)*)

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](#).

Etymology

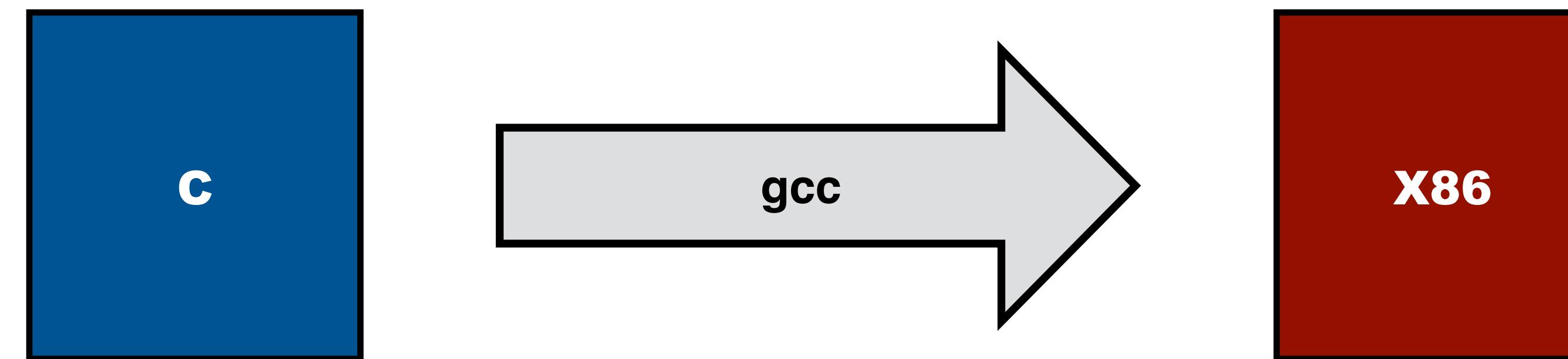
The first compiler was written by [Grace Hopper](#), in 1952, for the [A-0 System](#) language. The term *compiler* was coined by Hopper.^{[1][2]} The A-0 functioned more as a loader or [linker](#) than the modern notion of a compiler.

Compiling = Translating



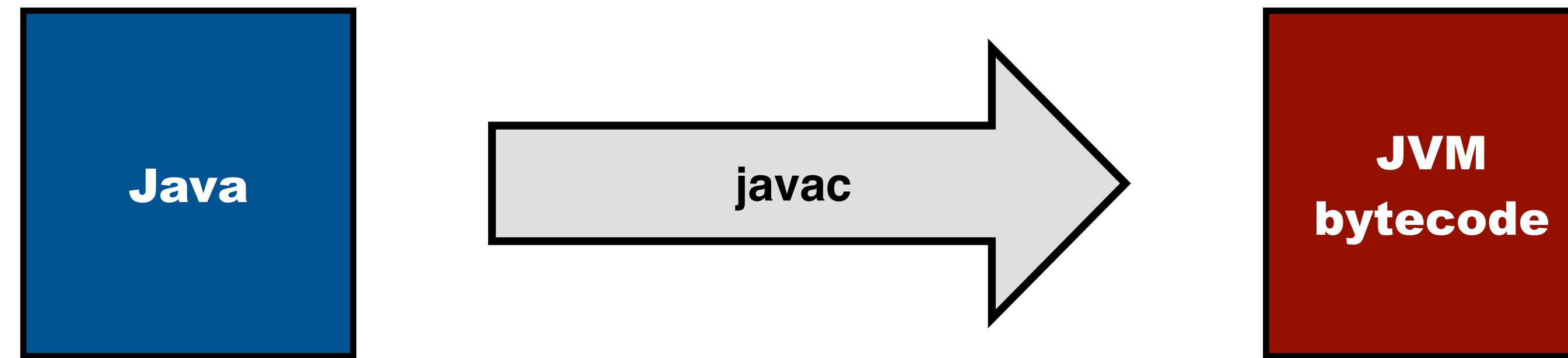
A compiler translates high-level programs to low-level programs

Compiling = Translating



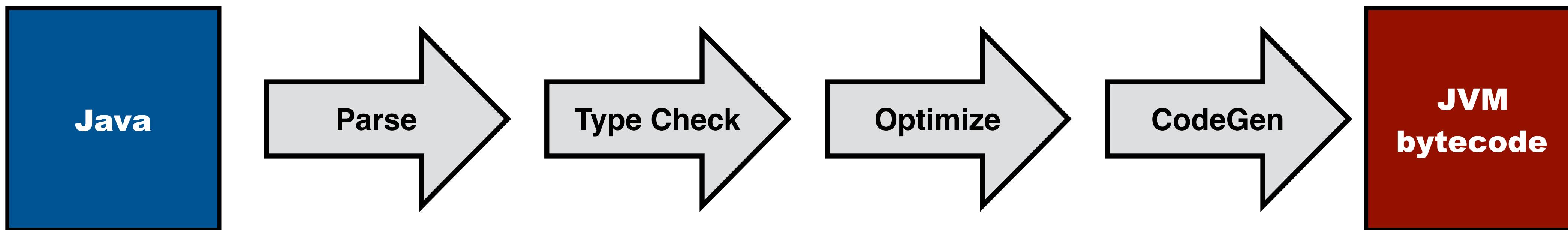
GCC translates C programs to object code for X86 (and other architectures)

Compiling = Translating



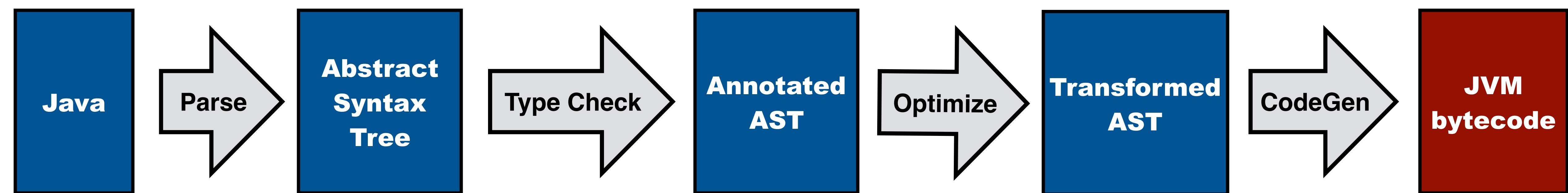
A Java compiler translates Java programs to bytecode instructions for Java Virtual Machine

Architecture: Multi-Pass Compiler



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



Optimizer

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

Compiler Components



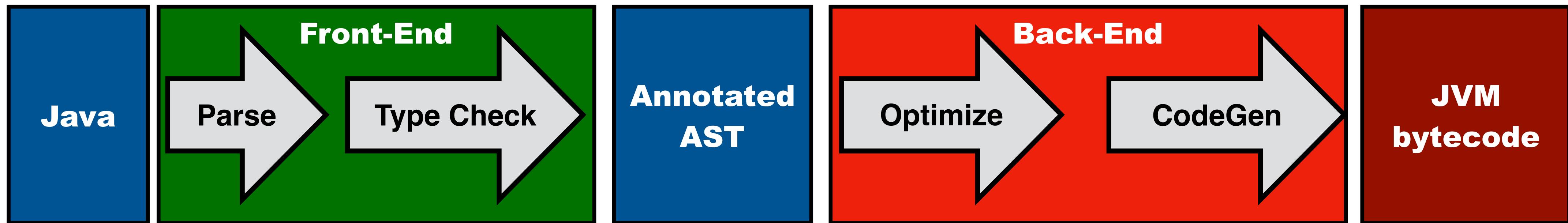
Code generator

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

Register allocator

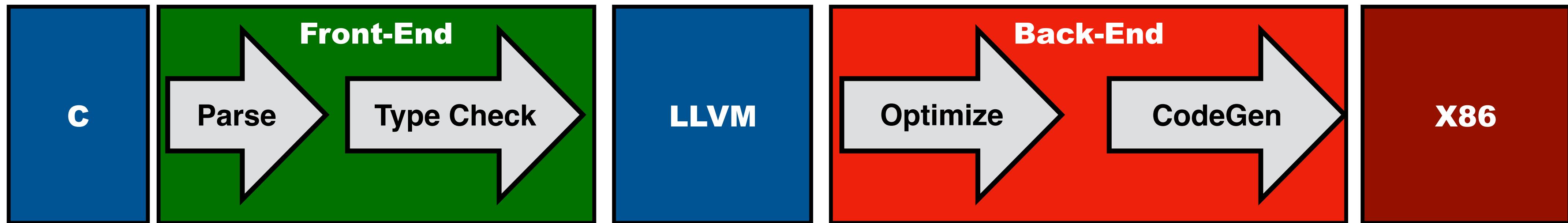
- Assigns physical registers to symbolic registers in the generated instructions

Compiler = Front-end + Back-End



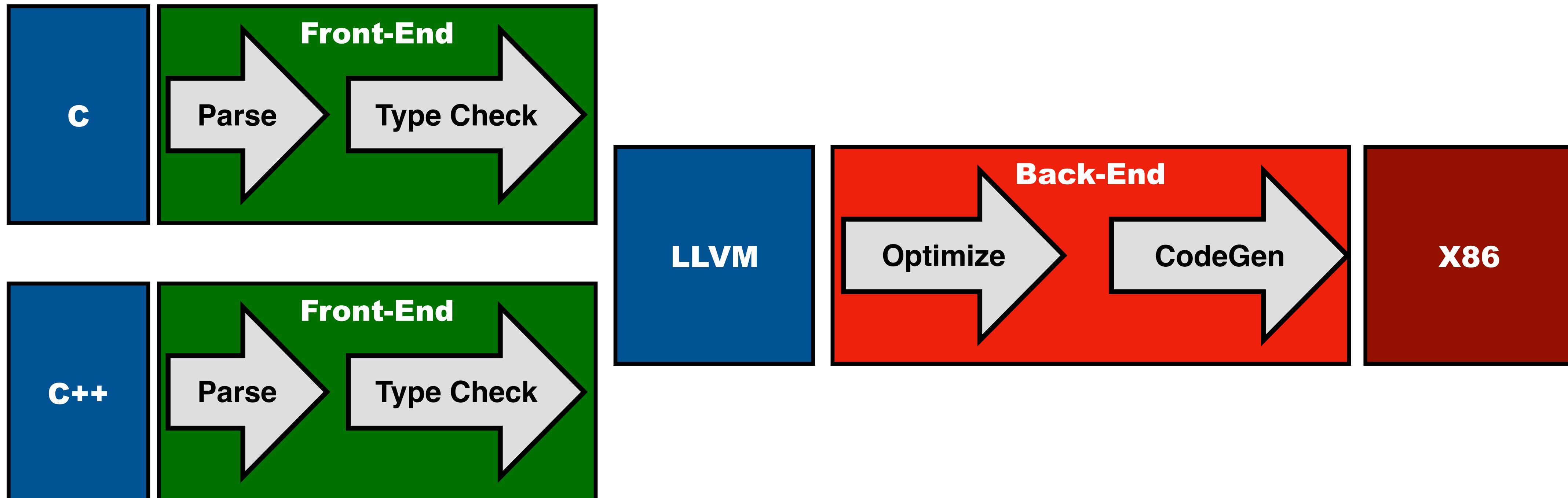
A compiler can typically be divided in a front-end (analysis) and a back-end (synthesis)

Compiler = Front-end + Back-End



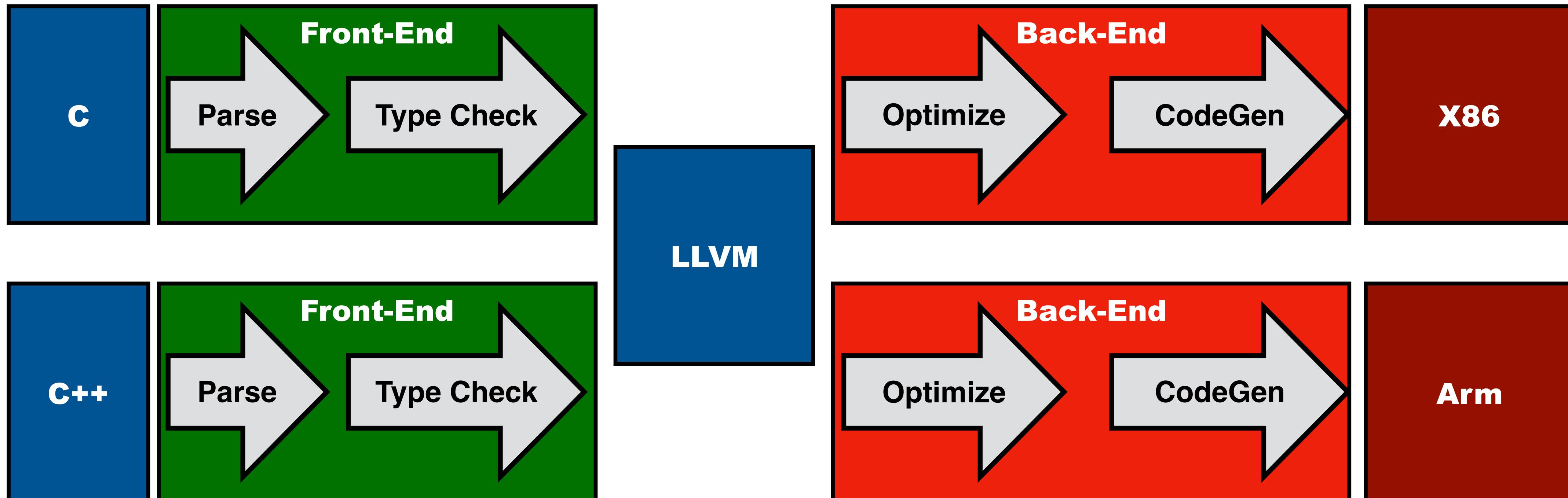
A compiler can typically be divided in a front-end (analysis) and a back-end (synthesis)

Repurposing Back-End



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

Retargeting Compiler



Retargeting: compile to different hardware architecture

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

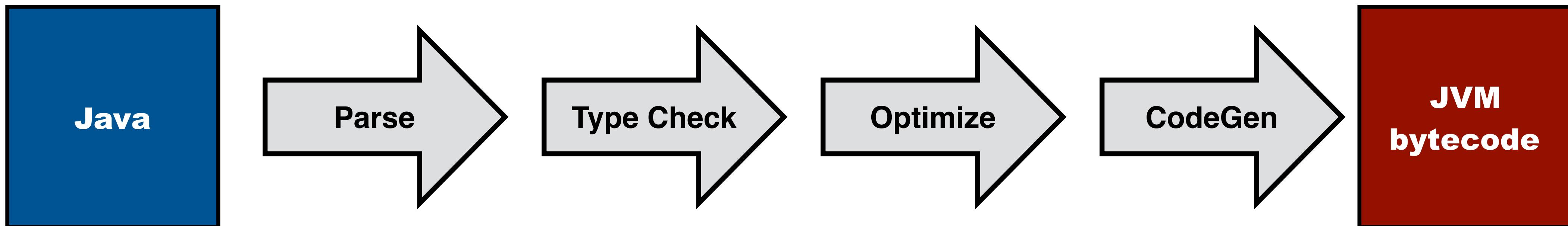
- generate configuration for FPGA or integrated circuit

De-compiler

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

What is a Compiler?

A bunch of components for translating programs



Compiler Construction = Building Variants of Java?

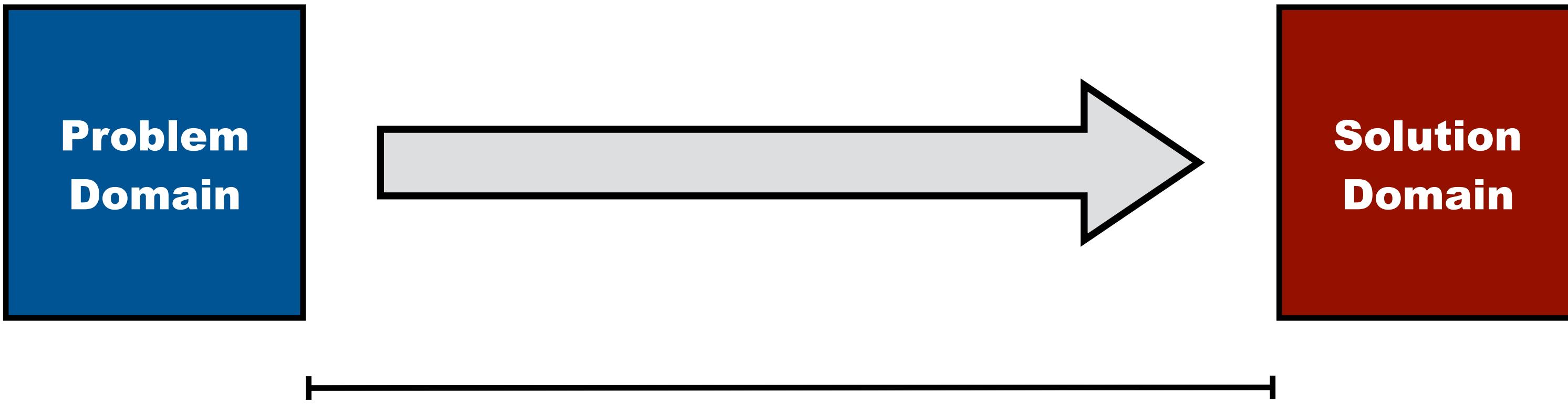
Why Compilers?

Programming = Instructing Computer

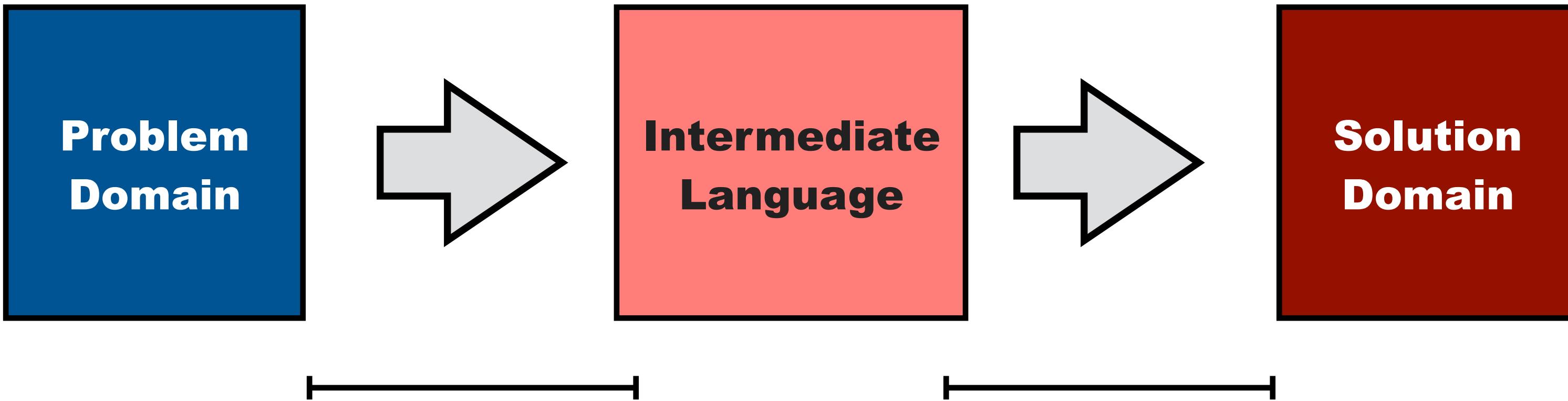
- fetch data from memory
- store data in register
- perform basic operation on data in register
- fetch instruction from memory
- update the program counter
- etc.

"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
```

http://en.wikipedia.org/wiki/Calling_convention

def f(x)={ ... }

f(e1)

function definition and call in Scala

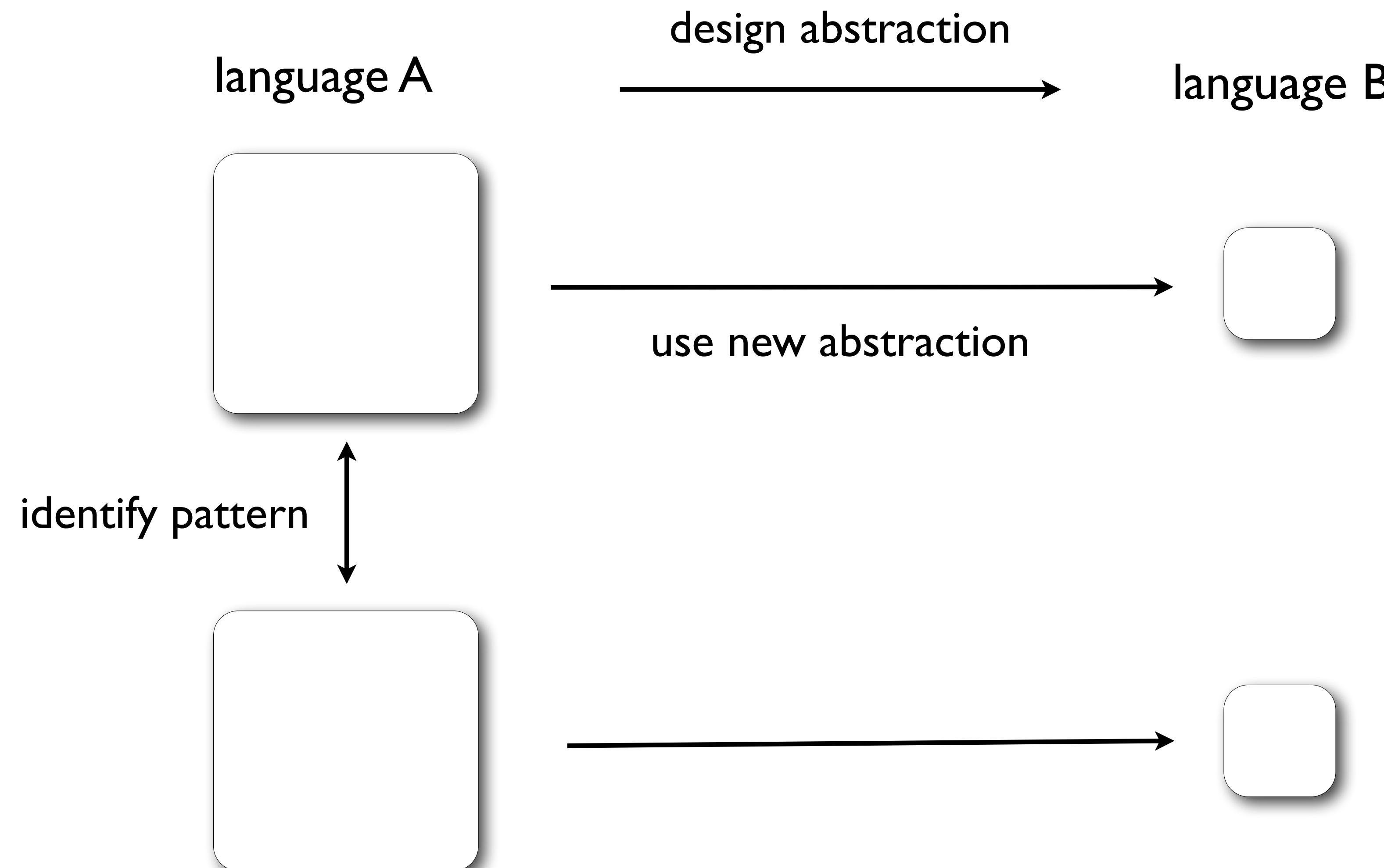
From Malloc to Garbage Collection

```
/* 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. */
}
```

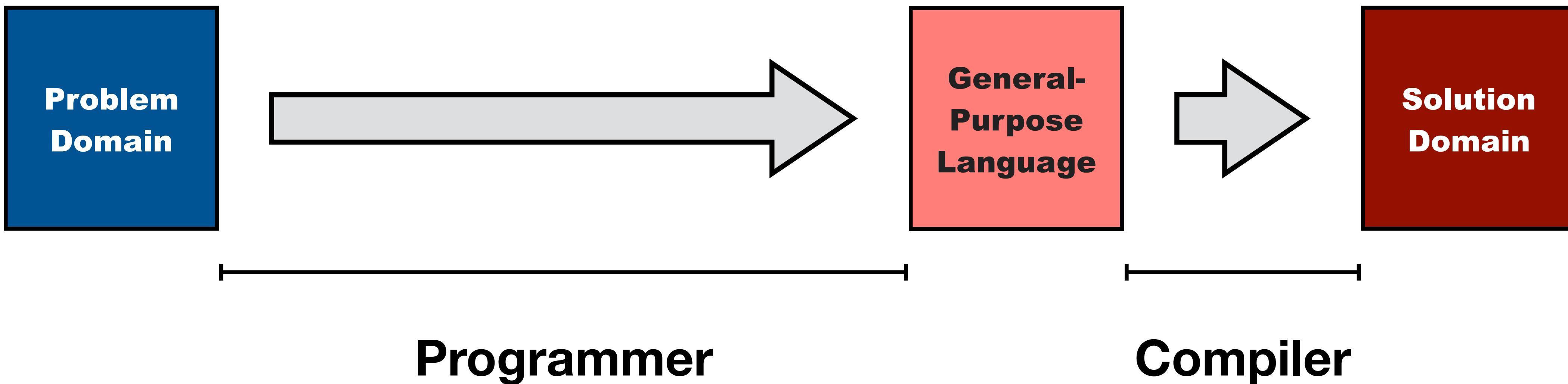
<http://en.wikipedia.org/wiki/Malloc>

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

Linguistic Abstraction



Compiler Automates Work of Programmer

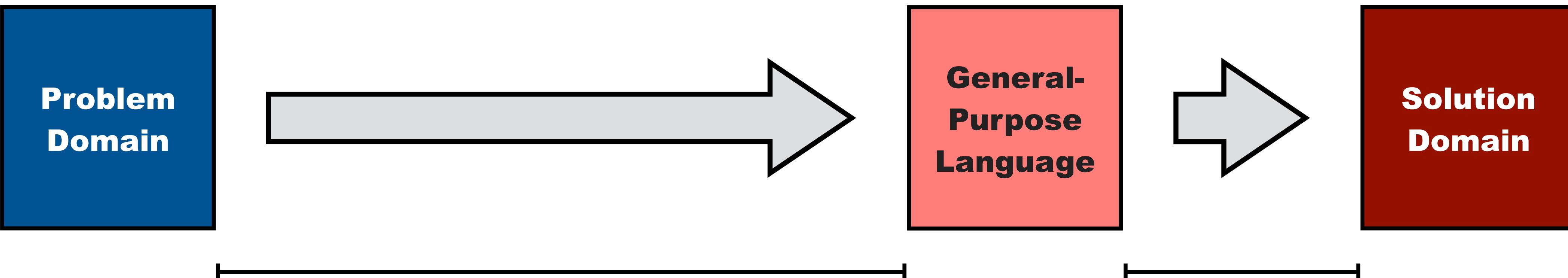


Compilers for modern high-level languages

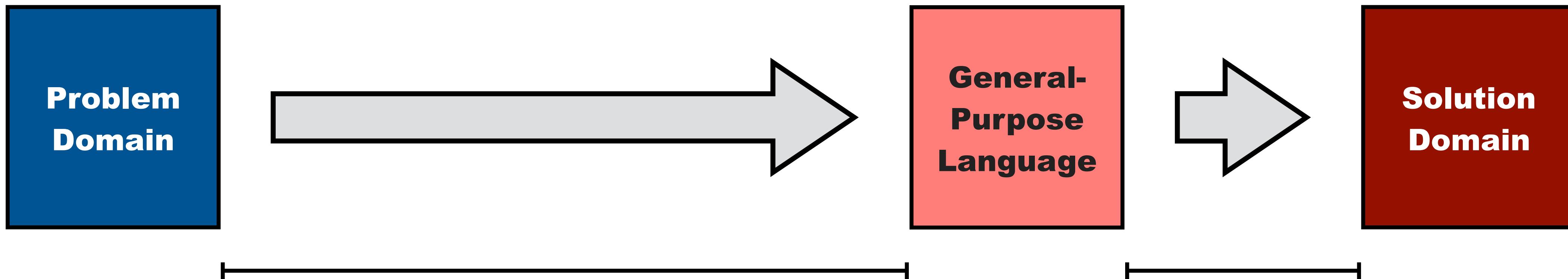
- Reduce the gap between problem domain and program
- Support programming in terms of computational concepts instead of machine concepts
- Abstract from hardware architecture (portability)
- Protect against a range of common programming errors

Domain-Specific Languages

Domains of Computation

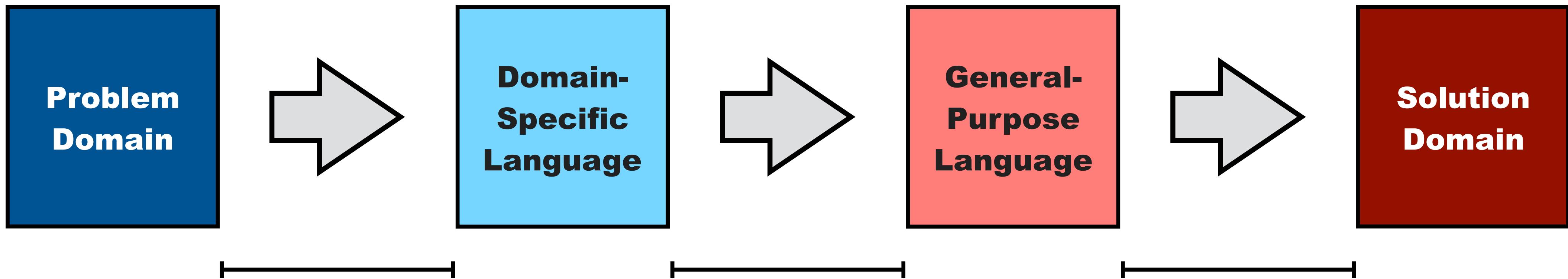


- Systems programming
- Embedded software
- Web programming
- Enterprise software
- Database programming
- Distributed programming
- Data analytics
- ...



“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.



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

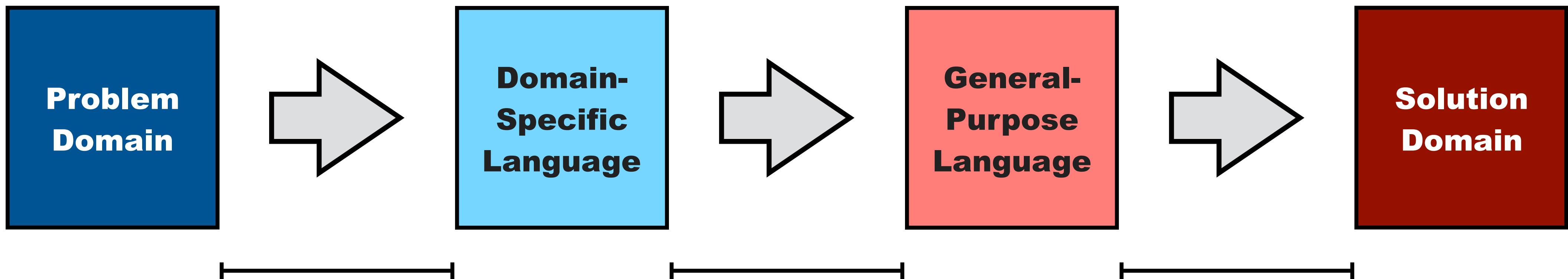
- What is the semantics of such definitions?
- How can we verify the correctness / consistency of language definitions?

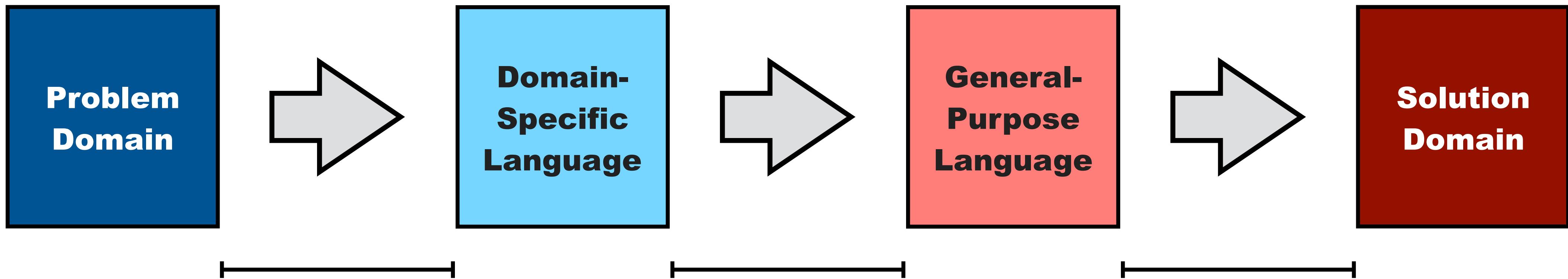
Implementation

- How do we derive efficient language implementations from such definitions?

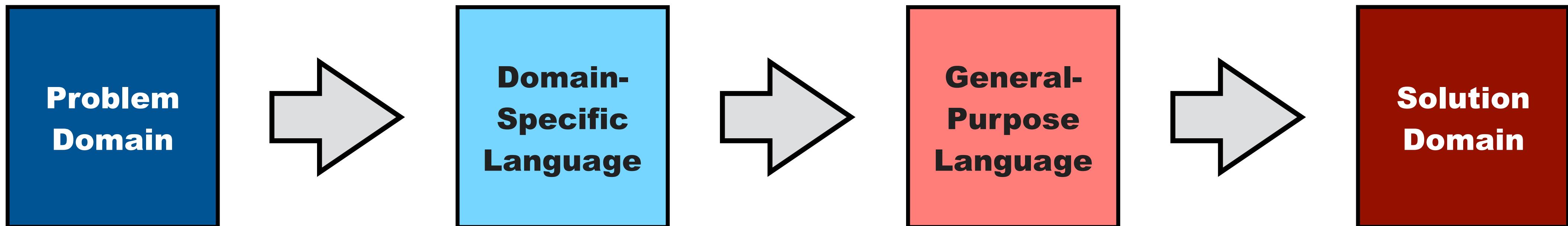
Evaluation

- Apply to new and existing languages to determine adequacy

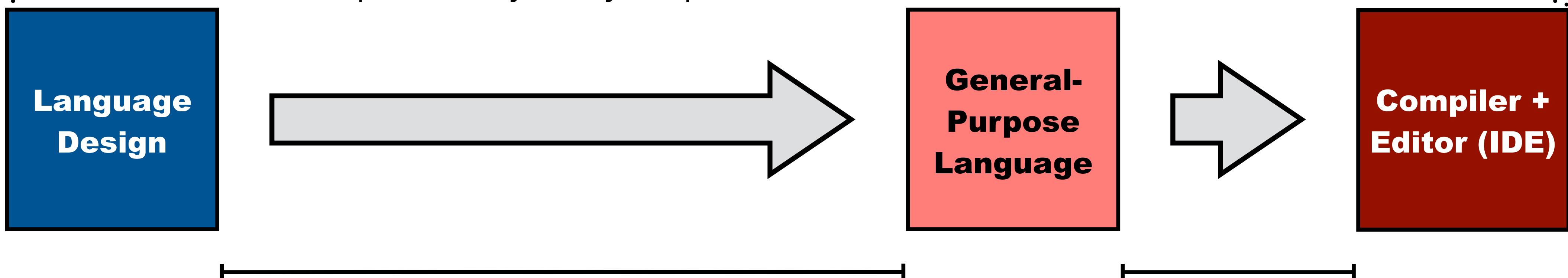


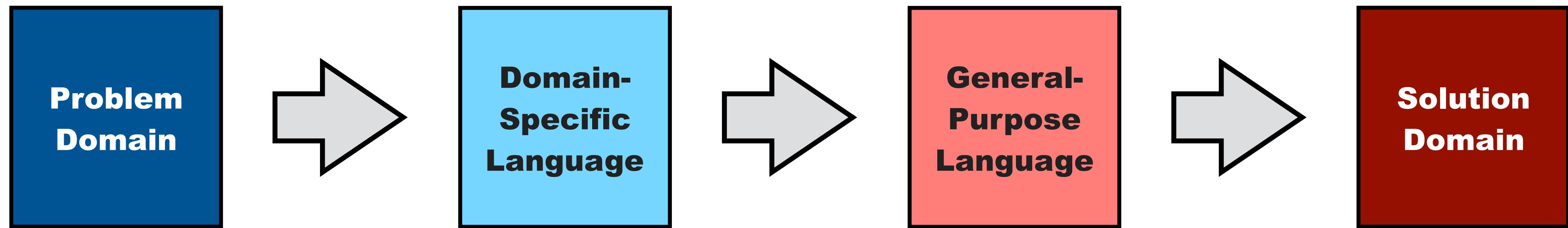


Making programming languages
is probably very expensive?

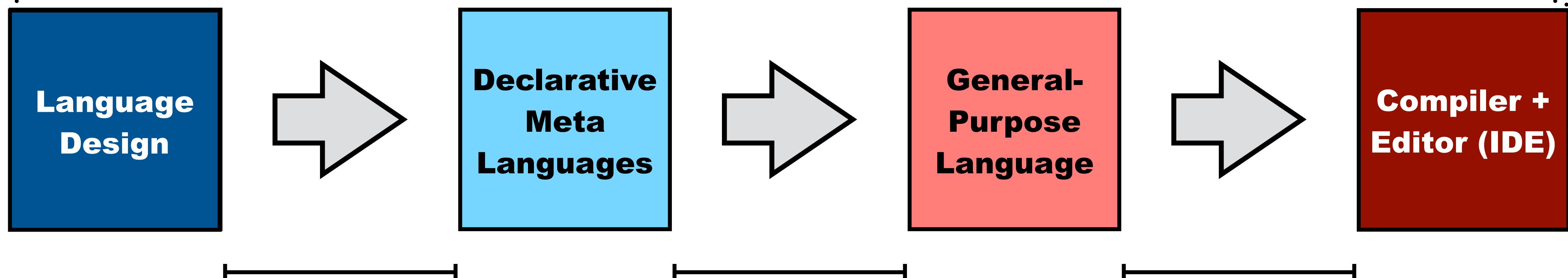


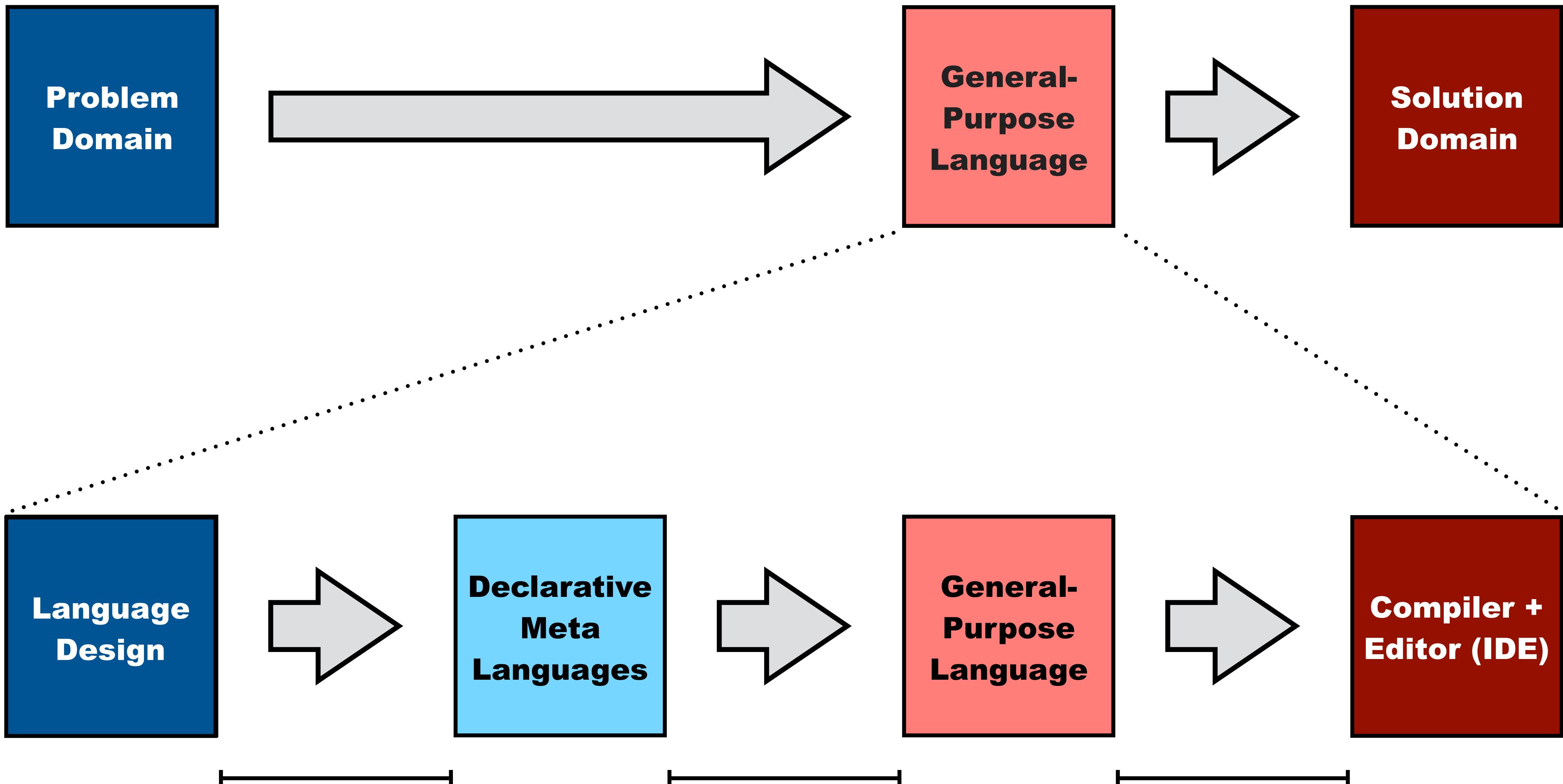
Making programming languages
is probably very expensive?



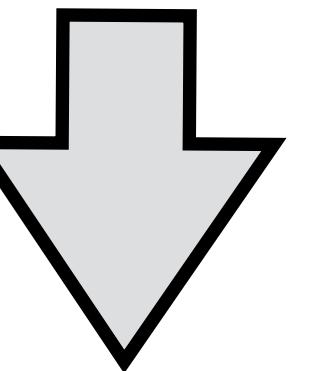


Meta-Linguistic Abstraction

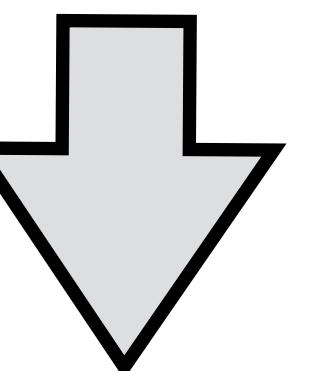




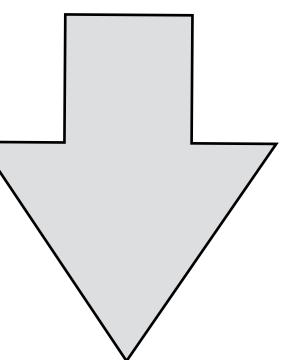
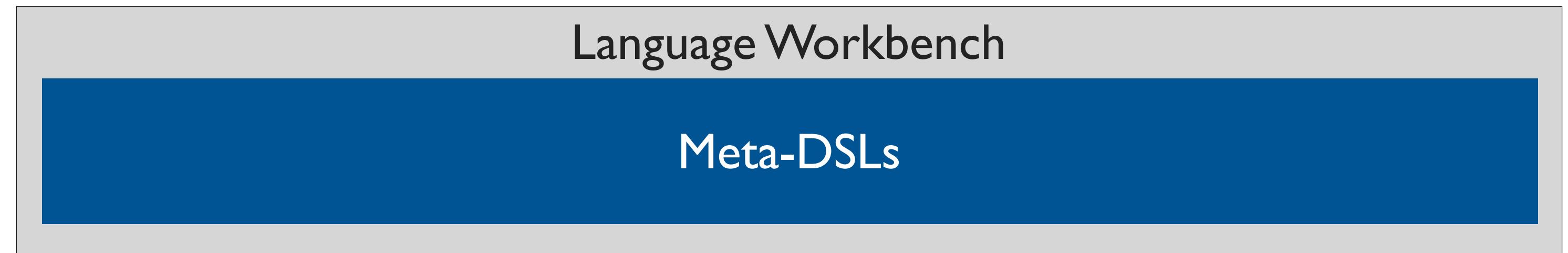
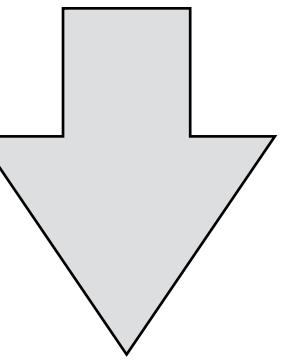
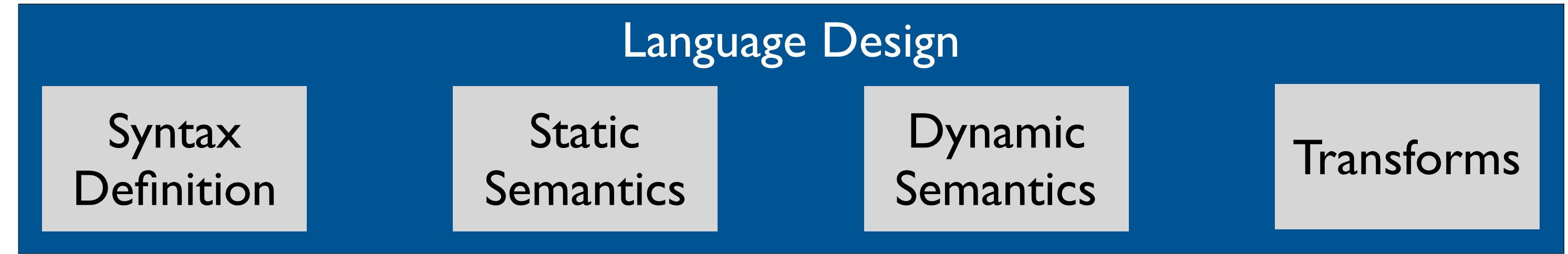
**Language
Design**

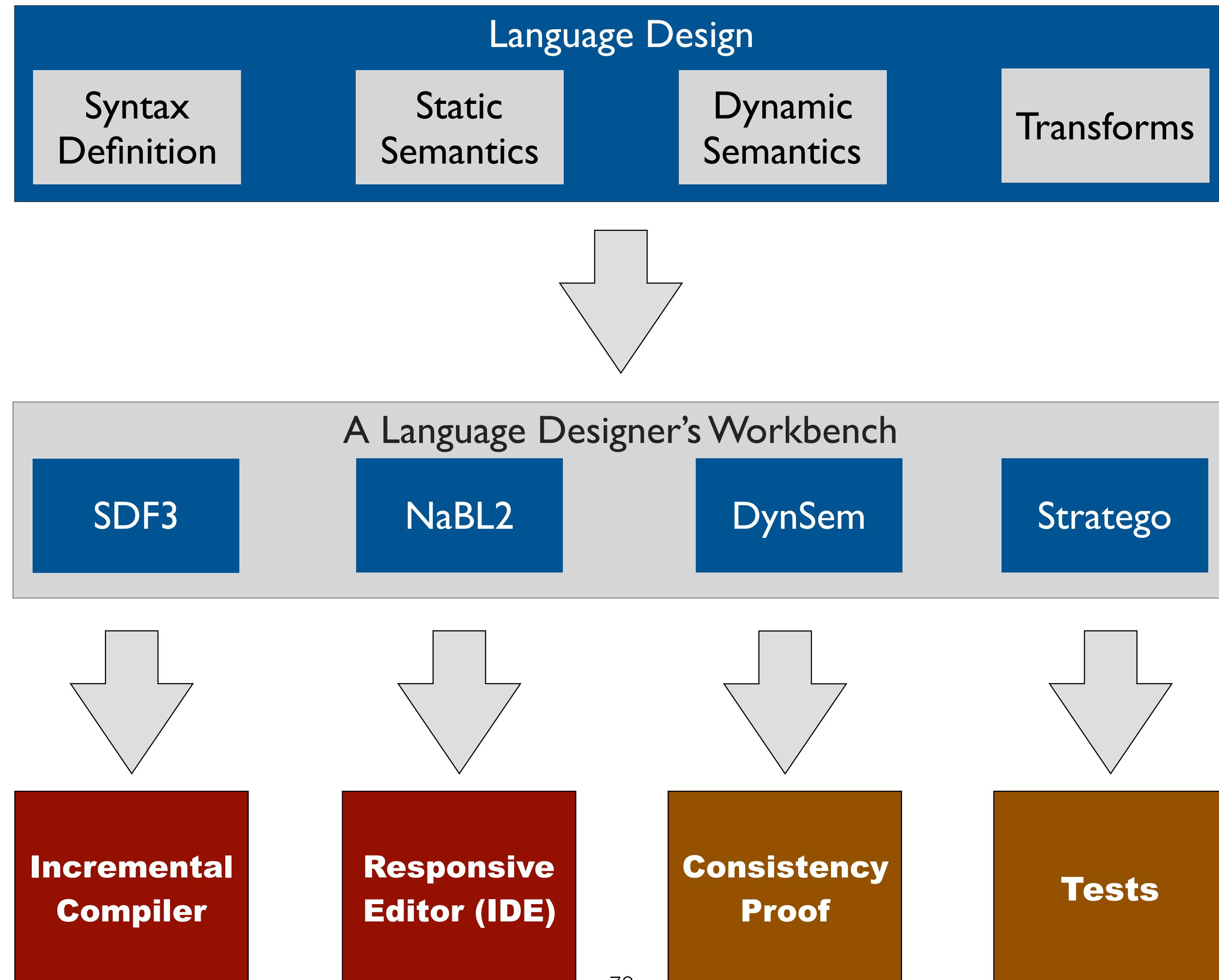


**Declarative
Meta
Languages**



**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

Meta-Languages in Spooftax Language Workbench

SDF3: Syntax definition

- context-free grammars + disambiguation + constructors + templates
- derivation of parser, formatter, syntax highlighting, ...

NaBL2 or Statix: Names & Types

- name resolution with scope graphs
- type checking/inference with constraints
- derivation of name & type resolution algorithm

Stratego: Program Transformation

- term rewrite rules with programmable rewriting strategies
- derivation of program transformation system

FlowSpec: Data-Flow Analysis

- extraction of control-flow graph and specification of data-flow rules
- derivation of data-flow analysis engine

DynSem: Dynamic Semantics

- specification of operational (natural) semantics
- derivation of interpreter

Literature

The Spooftax Language Workbench

- Lennart C. L. Kats, Eelco Visser
- OOPSLA 2010

A Language Designer's Workbench

- A one-stop-shop for implementation and verification of language designs
- Eelco Visser, Guido Wachsmuth, Andrew P. Tolmach, Pierre Neron, Vlad A. Vergu, Augusto Passalaqua, Gabriël D. P. Konat
- Onward 2014

A Taste of Compiler Construction

Language Definition in Spooftax Language Workbench

The screenshot shows the Spooftax Language Workbench interface with six tabs open:

- Calc.sdf3**: Syntax Definition (SDF3) code for a simple calculator language.
- calc.nabl2**: Static Semantics (NaBL2) code for the same language.
- transform.str**: Dynamic Semantics (DynSem) code for transforming the language.
- eval.ds**: Program Transformation (Stratego) code for evaluating expressions.
- java.str**: Program Transformation (Stratego) code for generating Java code.
- mortgage.calc**: A small application (Program) for calculating monthly mortgage payments.



Calc: A Little Calculator Language

```
rY = 0.017; // yearly interest rate
Y = 30;      // number of years
P = 379,000; // principal

N = Y * 12; // number of months

c = if(rY == 0) // no interest
    P / N
  else
    let r = rY / 12 in
      let f = (1 + r) ^ N in
        (r * P * f) / (f - 1);

c; // payment per month
```

<https://github.com/MetaBorgCube/metaborg-calc>

<http://www.metaborg.org/en/latest/source/langdev/meta/lang/tour/index.html>

Calc: Syntax Definition

```
context-free syntax // numbers
```

```
Exp = <(<Exp>)> {bracket}  
  
Exp.Num = NUM  
Exp.Min = <-<Exp>>  
Exp.Pow = <<Exp> ^ <Exp>> {right}  
Exp.Mul = <<Exp> * <Exp>> {left}  
Exp.Div = <<Exp> / <Exp>> {left}  
Exp.Sub = <<Exp> - <Exp>> {left, prefer}  
Exp.Add = <<Exp> + <Exp>> {left}  
  
Exp.Eq = <<Exp> == <Exp>> {non-assoc}  
Exp.Neq = <<Exp> != <Exp>> {non-assoc}  
Exp.Gt = [[Exp] > [Exp]] {non-assoc}  
Exp.Lt = [[Exp] < [Exp]] {non-assoc}
```

```
context-free syntax // variables and functions
```

```
Exp.Var = ID  
Exp.Let = <  
  let <ID> = <Exp> in  
  <Exp>>  
>  
Exp.Fun = <\ \ <ID+> . <Exp>>  
Exp.App = <<Exp> <Exp>> {left}
```

Calc: Type System

rules // numbers

```
[[ Num(x) ^ (s) : NumT() ]].  
  
[[ Pow(e1, e2) ^ (s) : NumT() ]] :=  
  [[ e1 ^ (s) : NumT() ]],  
  [[ e2 ^ (s) : NumT() ]].  
[[ Mul(e1, e2) ^ (s) : NumT() ]] :=  
  [[ e1 ^ (s) : NumT() ]],  
  [[ e2 ^ (s) : NumT() ]].  
[[ Add(e1, e2) ^ (s) : NumT() ]] :=  
  [[ e1 ^ (s) : NumT() ]],  
  [[ e2 ^ (s) : NumT() ]].
```

rules // variables and functions

```
[[ Var(x) ^ (s) : ty ]] :=  
  {x} -> s, {x} |-> d, d : ty.  
  
[[ Let(x, e1, e2) ^ (s) : ty2 ]] :=  
  new s_let, {x} <- s_let, {x} : ty, s_let -P-> s,  
  [[ e1 ^ (s) : ty ]],  
  [[ e2 ^ (s_let) : ty2 ]].  
  
[[ Fun([x], e) ^ (s) : FunT(ty1, ty2) ]] :=  
  new s_fun, {x} <- s_fun, {x} : ty1, s_fun -P-> s,  
  [[ e ^ (s_fun) : ty2 ]].  
  
[[ App(e1, e2) ^ (s) : ty_res ]] :=  
  [[ e1 ^ (s) : ty_fun ]],  
  [[ e2 ^ (s) : ty_arg ]],  
  FunT(ty_arg, ty_res) inst0f ty_fun.
```

Calc: Dynamic Semantics

rules // numbers

$\text{Num}(n) \rightarrow \text{NumV}(\text{parseB}(n))$

$\text{Pow}(\text{NumV}(i), \text{NumV}(j)) \rightarrow \text{NumV}(\text{powB}(i, j))$

$\text{Mul}(\text{NumV}(i), \text{NumV}(j)) \rightarrow \text{NumV}(\text{mulB}(i, j))$

$\text{Div}(\text{NumV}(i), \text{NumV}(j)) \rightarrow \text{NumV}(\text{divB}(i, j))$

$\text{Sub}(\text{NumV}(i), \text{NumV}(j)) \rightarrow \text{NumV}(\text{subB}(i, j))$

$\text{Add}(\text{NumV}(i), \text{NumV}(j)) \rightarrow \text{NumV}(\text{addB}(i, j))$

$\text{Lt}(\text{NumV}(i), \text{NumV}(j)) \rightarrow \text{BoolV}(\text{ltB}(i, j))$

$\text{Eq}(\text{NumV}(i), \text{NumV}(j)) \rightarrow \text{BoolV}(\text{eqB}(i, j))$

rules // variables and functions

$E |- \text{Var}(x) \rightarrow E[x]$

$E |- \text{Fun}([x], e) \rightarrow \text{ClosV}(x, e, E)$

$E |- \text{Let}(x, v1, e2) \rightarrow v$

where $E \{x \mapsto v1, E\} |- e2 \rightarrow v$

$\text{App}(\text{ClosV}(x, e, E), v_arg) \rightarrow v$

where $E \{x \mapsto v_arg, E\} |- e \rightarrow v$

Calc: Code Generation

```
rules // numbers
```

```
exp-to-java : Num(v) -> ${[BigDecimal.valueOf([v])]}
```

```
exp-to-java :  
Add(e1, e2) -> ${[je1].add([je2])}  
with  
<exp-to-java> e1 => je1  
; <exp-to-java> e2 => je2
```

```
exp-to-java :  
Sub(e1, e2) -> ${[je1].subtract([je2])}  
with  
<exp-to-java> e1 => je1  
; <exp-to-java> e2 => je2
```

```
rules // variables and functions
```

```
exp-to-java : Var(x) -> ${[x]}
```

```
exp-to-java :  
Let(x, e1, e2) -> ${([jty]) [x] -> [je2].apply([je1])}  
with  
<nabl2-get-ast-type> e1 => ty1  
; <nabl2-get-ast-type> e2 => ty2  
; <type-to-java> FunT(ty1, ty2) => jty  
; <exp-to-java> e1 => je1  
; <exp-to-java> e2 => je2
```

```
exp-to-java :  
f@Fun([x], e) -> ${([jty]) [x] -> [je])}  
with  
<nabl2-get-ast-type> f => ty  
; <type-to-java> ty => jty  
; <exp-to-java> e => je
```

```
exp-to-java:  
App(e1, e2) -> ${[e1].apply([e2])}  
with  
<exp-to-java> e1 => je1  
; <exp-to-java> e2 => je2
```

Lecture Language: Tiger

The screenshot shows a code editor with three tabs:

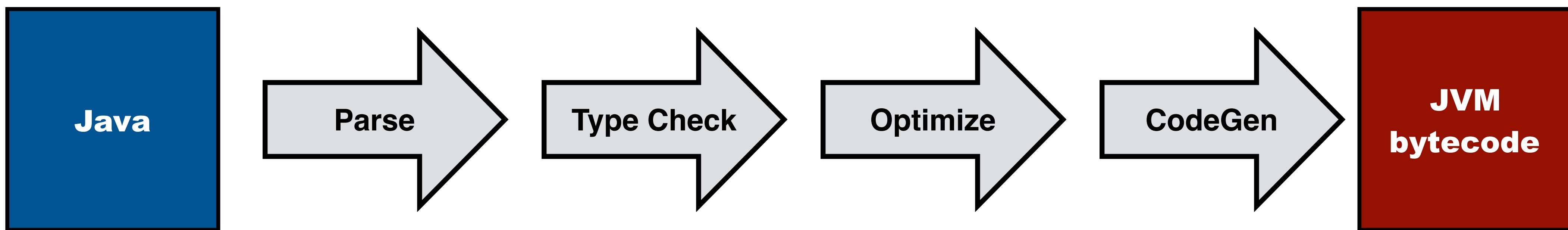
- Functions.sdf3**: Active tab. Contains Metaborg SDF3 grammar rules:

```
1 module Functions
2
3 imports Identifiers
4 imports Types
5
6 context-free syntax
7
8 Dec.FunDecls = <<{FunDec "\n"}+>> {longest-
9
10 FunDec.ProcDec = <
11   function <Id>(<{FArg ", "}*>) =
12     <Exp>
13 >
14
15 FunDec.FunDec = <
16   function <Id>(<{FArg ", ">*>) : <Type> =
17     <Exp>
18 >
19
20 FArg.FArg = <<Id> : <Type>>
21
22 Exp.Call = <<Id>(<{Exp ", ">*>)>
```
- functions.nabl2**: Shows Nabl2 code for function declarations and rules.
- queens.tig**: Shows Tiger code for solving the 8-queens problem.

<https://github.com/MetaBorgCube/metaborg-tiger>

Studying Compiler Construction

The Basis



**Compiler construction techniques
are applicable in a wide range of
software (development) applications**

Levels of Understanding Compilers

Specific

- Understanding a specific compiler
- Understanding a programming language (MiniJava)
- Understanding a target machine (Java Virtual Machine)
- Understanding a compilation scheme (MiniJava to Byte Code)

Architecture

- Understanding architecture of compilers
- Understanding (concepts of) programming languages
- Understanding compilation techniques

Domains

- Understanding (principles of) syntax definition and parsing
- Understanding (principles of) static semantics and type checking
- Understanding (principles of) dynamic semantics and interpretation/code generation

Meta

- Understanding meta-languages and their compilation

Course Topics

Syntax

- concrete syntax, abstract syntax
- context-free grammars
- derivations, ambiguity, disambiguation, associativity, priority
- parsing, parse trees, abstract syntax trees, terms
- pretty-printing
- parser generation
- syntactic editor services

Transformation

- rewrite rules, rewrite strategies
- simplification, desugaring

Statics

- static semantics and type checking
 - ▶ name binding, name resolution, scope graphs
 - ▶ types, type checking, type inference, subtyping
 - ▶ unification, constraints
- semantic editor services
- data-flow analysis
 - ▶ control-flow, data-flow
 - ▶ monotone frameworks, worklist algorithm

Dynamics

- dynamic semantics and interpreters
- operational semantics, program execution
- virtual machines, assembly code, byte code
- code generation
- memory management, garbage collection

Lectures (Tentative)

Q1

- What is a compiler? (Introduction)
- Syntax Definition
- Parsing
- Syntactic Editor Services
- Transformation
- Static Semantics & Name Resolution
- Type Checking
- Specification with Statix (or NaBL2)
- Constraint Resolution

Q2

- Data-Flow Analysis
- Monotone Frameworks
- Virtual Machines
- Code Generation
- Interpreters
- Memory Management
- Applications
- Reserve
- Overview

Lectures: Thursday, 10:45 in
Lecture Hall Boole at EWI

Lectures: Wednesday, 15:45
in Lecture Hall Boole at EWI

Homework Assignments

This award winning paper describes the design of the Spoofax Language Workbench.

It provides an alternative architecture for programming languages tooling from the compiler pipeline discussed in this lecture.

Read the paper and make the homework assignments on WebLab.

Note that the details of some of the technologies in Spoofax have changed since the publication. The

The Spoofax Language Workbench

Rules for Declarative Specification of Languages and IDEs

Lennart C. L. Kats

Delft University of Technology

l.c.l.kats@tudelft.nl

Eelco Visser

Delft University of Technology

visser@acm.org

Abstract

Spoofax is a language workbench for efficient, agile development of textual domain-specific languages with state-of-the-art IDE support. Spoofax integrates language processing techniques for parser generation, meta-programming, and IDE development into a single environment. It uses concise, declarative specifications for languages and IDE services. In this paper we describe the architecture of Spoofax and introduce idioms for high-level specifications of language semantics using rewrite rules, showing how analyses can be reused for transformations, code generation, and editor services such as error marking, reference resolving, and content completion. The implementation of these services is supported by language-parametric editor service classes that can be dynamically loaded by the Eclipse IDE, allowing new languages to be developed and used side-by-side in the same Eclipse environment.

Categories and Subject Descriptors D.2.3 [Software Engineering]: Coding Tools and Techniques; D.2.6 [Software Engineering]: Programming Environments

General Terms Languages

1. Introduction

Domain-specific languages (DSLs) provide high expressive power focused on a particular problem domain [38, 47]. They provide linguistic abstractions over common tasks within a domain, so that developers can concentrate on application logic rather than the accidental complexity of low-level implementation details. DSLs have a concise, domain-specific notation for common tasks in a domain, and allow reasoning at the level of these constructs. This allows them to be used for automated, domain-specific analysis, verification, optimization, parallelization, and transformation (AVOPT) [38].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

OOPSLA/SPLASH'10, October 17–21, 2010, Reno/Tahoe, Nevada, USA.
Copyright © 2010 ACM 978-1-4503-0203-6/10/10...\$10.00

For developers to be productive with DSLs, good integrated development environments (IDEs) for these languages are essential. Over the past four decades, IDEs have slowly risen from novelty tool status to becoming a fundamental part of software engineering. In early 2001, IntelliJ IDEA [42] revolutionized the IDE landscape [17] with an IDE for the Java language that parsed files as they were typed (with error recovery in case of syntax errors), performed semantic analysis in the background, and provided code navigation with a live view of the program outline, references to declarations of identifiers, content completion proposals as programmers were typing, and the ability to transform the program based on the abstract representation (refactorings). The now prominent Eclipse platform, and soon after, Visual Studio, quickly adopted these same features. No longer would programmers be satisfied with code editors that provided basic syntax highlighting and a “build” button. For new languages to become a success, state-of-the-art IDE support is now mandatory. For the production of DSLs this requirement is a particular problem, since these languages are often developed with much fewer resources than general purpose languages.

There are five key ingredients for the construction of a new domain-specific language. (1) A parser for the syntax of the language. (2) Semantic analysis to validate DSL programs according to some set of constraints. (3) Transformations manipulate DSL programs and can convert a high-level, technology-independent DSL specification to a lower-level program. (4) A code generator that emits executable code. (5) Integration of the language into an IDE.

Traditionally, a lot of effort was required for each of these ingredients. However, there are now many tools that support the various aspects of DSL development. Parser generators can automatically create a parsers from a grammar. Modern parser generators can construct efficient parsers that can be used in an interactive environment, supporting error recovery in case of syntax-incorrect or incomplete programs. Meta-programming languages [3, 10, 12, 20, 35] and frameworks [39, 57] make it much easier to specify the semantics of a language. Tools and frameworks for IDE development such as IMP [7, 8] and TMF [56], simplify the implementation of IDE services. Other tools, such as the Synthesizer

Assignment

Read this paper in preparation for Lecture 2

<https://doi.org/10.1145/1932682.1869535>

<http://swerl.tudelft.nl/twiki/pub/Main/TechnicalReports/TUD-SERG-2010-019.pdf>

Pure and Declarative Syntax Definition: Paradise Lost and Regained

Lennart C. L. Kats

Delft University of Technology
l.c.l.kats@tudelft.nl

Eelco Visser

Delft University of Technology
visser@acm.org

Guido Wachsmuth

Delft University of Technology
g.h.wachsmuth@tudelft.nl

Abstract

Syntax definitions are pervasive in modern software systems, and serve as the basis for language processing tools like parsers and compilers. Mainstream parser generators pose restrictions on syntax definitions that follow from their implementation algorithm. They hamper evolution, maintainability, and compositionality of syntax definitions. The pureness and declarativity of syntax definitions is lost. We analyze how these problems arise for different aspects of syntax definitions, discuss their consequences for language engineers, and show how the pure and declarative nature of syntax definitions can be regained.

Categories and Subject Descriptors D.3.1 [Programming Languages]: Formal Definitions and Theory — Syntax; D.3.4 [Programming Languages]: Processors — Parsing; D.2.3 [Software Engineering]: Coding Tools and Techniques

General Terms Design, Languages

Prologue

In the beginning were the *words*, and the words were *trees*, and the trees were words. All words were made through *grammars*, and without grammars was not any word made that was made. Those were the days of the garden of Eden. And there where language engineers strolling through the garden. They made languages which were sets of words by making grammars full of beauty. And with these grammars, they turned words into trees and trees into words. And the trees were natural, and pure, and beautiful, as were the grammars.

Among them were software engineers who made software as the language engineers made languages. And they dwelt with them and they were one people. The language en-

gineers were software engineers and the software engineers were language engineers. And the language engineers made *language software*. They made *recognizers* to know words, and *generators* to make words, and *parsers* to turn words into trees, and *formatters* to turn trees into words.

But the software they made was not as natural, and pure, and beautiful as the grammars they made. So they made software to make language software and began to make language software by making *syntax definitions*. And the syntax definitions were grammars and grammars were syntax definitions. With their software, they turned syntax definitions into language software. And the syntax definitions were language software and language software were syntax definitions. And the syntax definitions were natural, and pure, and beautiful, as were the grammars.

The Fall Now the serpent was more crafty than any other beast of the field. He said to the language engineers,

Did you actually decide not to build any parsers?

And the language engineers said to the serpent,

We build parsers, but we decided not to build others than general parsers, nor shall we try it, lest we loose our syntax definitions to be natural, and pure, and beautiful.

But the serpent said to the language engineers,

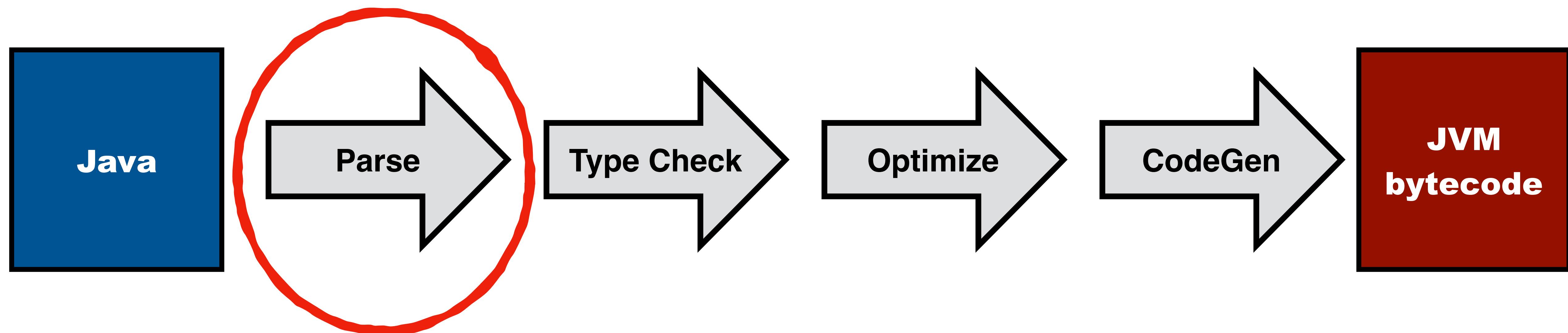
You will not surely loose your syntax definitions to be natural, and pure, and beautiful. For you know that when you build particular parsers your benchmarks will be improved, and your parsers will be the best, running fast and efficient.

So when the language engineers saw that restricted parsers were good for efficiency, and that they were a delight to the benchmarks, they made software to make efficient parsers and began to make efficient parsers by making *parser definitions*. Those days, the language engineers went out from the garden of Eden. In pain they made parser definitions all the days of their life. But the parser definitions were not grammars and grammars were not parser definitions. And by the sweat of their faces they turned parser definitions into effi-

Next Lecture

Next: Syntax Definition

Lecture: Friday, 13:45 in
Lecture Hall Pi at EWI



Specification of syntax definition from which we can derive parsers