Declare Your Language

Chapter 1: What is a Compiler?

Eelco Visser

IN4303 Compiler Construction
TU Delft
September 2017

TU Delft

Course Organization

No use of electronic devices during lectures

Eelco Visser About Research Teaching Contact Blog Admin

Eelco Visser

Full Professor of Computer Science at TU Delft | Chair Programming Languages Group | Antoni van Leeuwenhoek Professor

News

- August 11, 2017: Paper "The Semantics of Name Resolution in Grace" with Vlad Vergu accepted at DLS 2017
- August 4, 2017: Paper: "Deep Priority Conflicts in the Wild — A Pilot Study" with Eduardo Amorim and Michael Steindorfer accepted at SLE 2017
- August 4, 2017: Paper "FlowSpec: Declarative Dataflow Analysis Specification" with Jeff Smits and Guido Wachsmuth accepted at SLE 2017
- July 10, 2017: I will serve on the OOPSLA 2018 program committee; submission deadline is April 16, 2017; submit interesting papers!
- April 26, 2017: I will give a talk about scope graphs at Curry On 2017 in Barcelona (video)
- April 12, 2017: Paper IceDust 2: Derived Bidirectional Relations and Calculation Strategy Composition with Daco Harkes accepted at ECOOP 2017
- March 6, 2017: I will be on the Program Committee of ‹Programming› 2018; submit interesting papers!
- February 28, 2017: I will be lecturing on Declarative Language Definition at the ECOOP 2017 Summer School

Archive

Open Positions

- I have an opening for a PhD student in software language engineering
- Postdoc candidates: contact me if you are interested in submitting a proposal to the LEaDing Fellows program for Marie Curie postdoc positions; next deadline is December 31, 2017

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

http://eelcovisser.org/

Brightspace: Announcements

TU Delft | IN4303 Compiler Construction (2017/1... | Eelco Visser

Course Home | Content | Collaboration | Assignments | Grades | Course Admin | Help

Updates

There are no current updates for IN4303 Compiler Construction (2017/18 Q1)

Calendar

Thursday, 7 September, 2017

Upcoming events

There are no events to display. Create an event.

Announcements

Take a notebook to lecture

Posted 07 September, 2017 23:01

Tomorrow is the first lecture of the Compiler Construction (IN4303) course at 13:45 at **3mE (IZ G) (not at EWI!)**.

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

Posted 18 August, 2017 21:31

This course runs in Q1 and Q2

The website with information about the the course and the lab is https://tudelft-in4303-2017.github.io/

We will use WebLab for grade administration. Please enroll in the course https://weblab.tudelft.nl/in4303/2017-2018/

The first lecture is on Friday September 8 at 13:45 at 3mE - IZ G.

After that lectures are on Tuesday at 17:45 in lecture hall Pi at EWI starting on September 12.

Show All Announcements

Course Website

TUDELFT IN4303 LECTURES ASSIGNMENTS CONTACT

Photo Credit: Wikimedia Commons

Compiler Construction

- Twitter
- Github

Edit on GitHub

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 is about the efficient construction of compilers and their integration in the IDE. Its lectures are organized in three parts:

The first part focuses on declarative specification of compiler components as supported by state-of-the-art tools for compiler construction, including lexical syntax, context-free syntax, static semantics, and code generation.

In the second part, we address techniques for the compilation of imperative and object-oriented languages, including activation records, memory management, register allocation, and optimization techniques in detail.

The third part takes a closer look on the inside working of compiler components and their generators. In particular, we study parsing algorithms and parser generation.

In the practical work, students construct a compiler for a small object-oriented language using the tools introduced in the first part of the lectures.

https://tudelft-in4303-2017.github.io/

Compiler Construction

Twitter
Github

Edit on GitHub

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 2017-2018

We offer walk-in hours on Tuesday mornings 10:15-11:45 in room HB 08.280.

Team Members

Eelco Visser
responsible instructor
HB 08.270 Email Twitter

Hendrik van Antwerpen
graduate teaching assistant
HB 08.280 Tue 10:15-11:45 Email

Eduardo de Souza Amorim
graduate teaching assistant
HB 08.280 Tue 10:15-11:45 Email

Gabriël Konat
graduate teaching assistant
HB 08.280 Tue 10:15-11:45 Email

Jeff Smits
graduate teaching assistant
HB 08.040 Email

Photo Credit: Delft University of Technology

Compiler Construction

- Twitter
- Github

Edit on GitHub

Lectures

The lectures will be renewed this year; the schedule will be updated as we go. For your information, last year's schedule is included below.

Schedule 2017-2018

topic schedule is tentative

Q1

- Fri Sep 8, 13:45-15:30, 3mE - IZ G: What is a compiler? (Introduction)
- Tue Sep 12, 17:45-19:30, EWI Pi: Syntax Definition
- Tue Sep 19, 17:45-19:30, EWI Pi: Basic Parsing
- Tue Sep 26, 17:45-19:30, EWI Pi: Term Rewriting
- Tue Oct 3, 17:45-19:30, EWI Pi: Static Semantics & Name Resolution
- Tue Oct 10, 17:45-19:30, EWI Pi: Type Constraints
- Tue Oct 17, 17:45-19:30, EWI Pi: Constraint Resolution
- Tue Oct 24, 17:45-19:30, EWI Pi: (overflow)

Q2

- Tue Nov 14, 17:45-19:30, EWI Pi: Dynamic Semantics
- Tue Nov 21, 17:45-19:30, EWI Pi: Virtual Machines & Code Generation
- Tue Nov 28, 17:45-19:30, EWI Pi: Just-in-Time Compilation (Interpreters & Partial Evaluation)
- Tue Dec 5, 17:45-19:30, EWI Pi: Data-Flow Analysis
- Tue Dec 12, 17:45-19:30, EWI Pi: Garbage Collection
- Tue Dec 19, 17:45-19:30, EWI Pi: Advanced Parsing
- Tue Jan 9, 17:45-19:30, EWI Pi: Overview
- Tue Jan 16, 17:45-19:30, EWI Pi: (overflow)
- Tue Jan 23, 17:45-19:30, EWI Pi: (overflow)

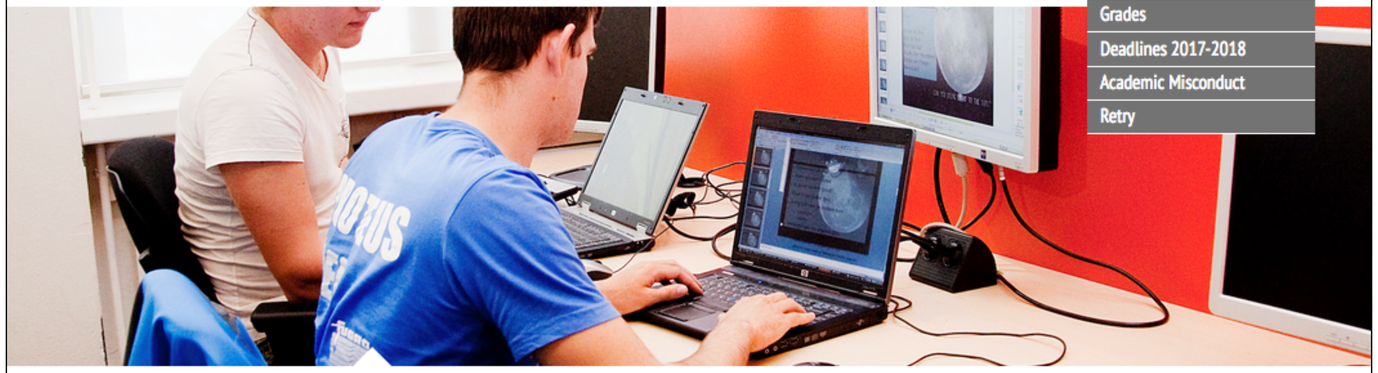


Photo Credit: Delft University of Technology

Compiler Construction

Twitter
Github

Edit on GitHub

Assignments

In the practical part of the course, you build an IDE for MiniJava, a small subset of Java. We split this task into three milestones of (roughly) weekly individual assignments:

- 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
- Milestone 3: Code Generation
  - Compiling Minimal Programs
  - Compiling Expressions and Statements
  - Compiling, Fields, Parameters, and Variables

Submission

We rely on GitHub for assignment submissions. We assign a private GitHub repository to each student and encourage students to commit and push their work frequently. To submit your assignment, you need to file a pull request.

To check your progress on an assignment, you can submit a *preliminary solution*. We will provide limited early feedback on preliminary solutions. This feedback typically comes with a tentative grade and points out areas where your solution is incomplete or insufficient, without giving any details on the reasons. We do *not give any feedback on the day of a deadline*, since this will not be early at all.

Compiler Construction

Twitter
Github

Edit on GitHub

Lab 1: Testing Syntax Analysis

This lab is under construction. Proceed at own risk.

In this lab, you develop a test suite for syntax analysis. The test suite consists of positive and negative test cases.

Overview

Objectives

Develop a test suite for syntax analysis. The test suite should provide

1. Syntactically valid and invalid test cases for sorts
   - `Program`,
   - `MainClass`,
   - `ClassDecl`,
   - `VarDecl`,
   - `MethodDecl`,
   - `Type`,
   - `Statement`,
   - `Exp`,
   - `ID` and
   - `INT`.

   Test cases for `FieldDecl` and `ParamDecl` should be covered in the tests for `ClassDecl` and `MethodDecl`, respectively.

   For grading, it is required to comply with these sort names literally.

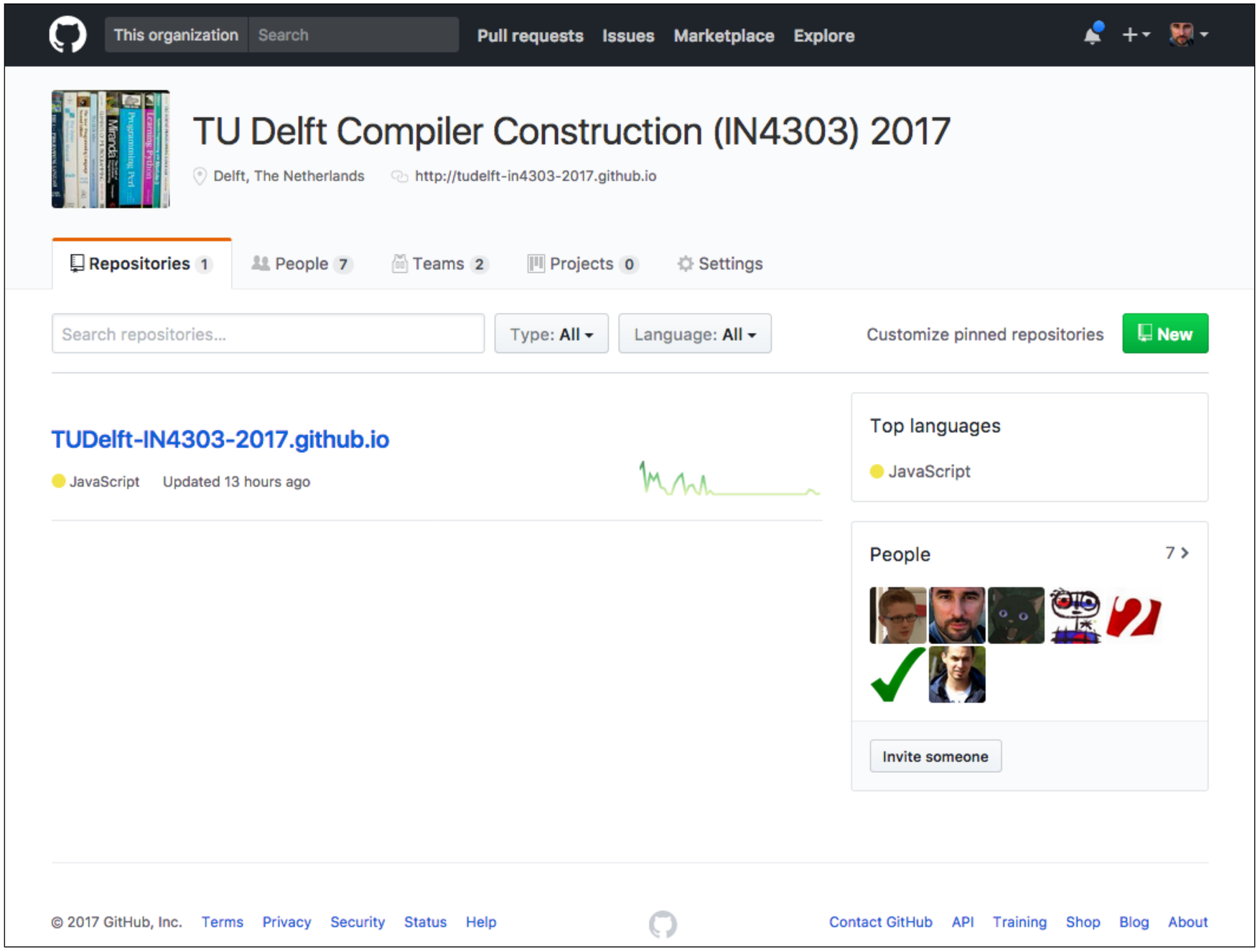
2. Disambiguation tests for associativity and precedence in expressions.
3. Test cases for mandatory and optional whitespace.

Submission

You need to submit your test project with a pull request against branch `assignment1` on GitHub. The Git documentation explains how to file such a request. We expect to find your Spoofax project `minijava` and your test project `minijava.test.syntax` next to each other in the root of the repository.

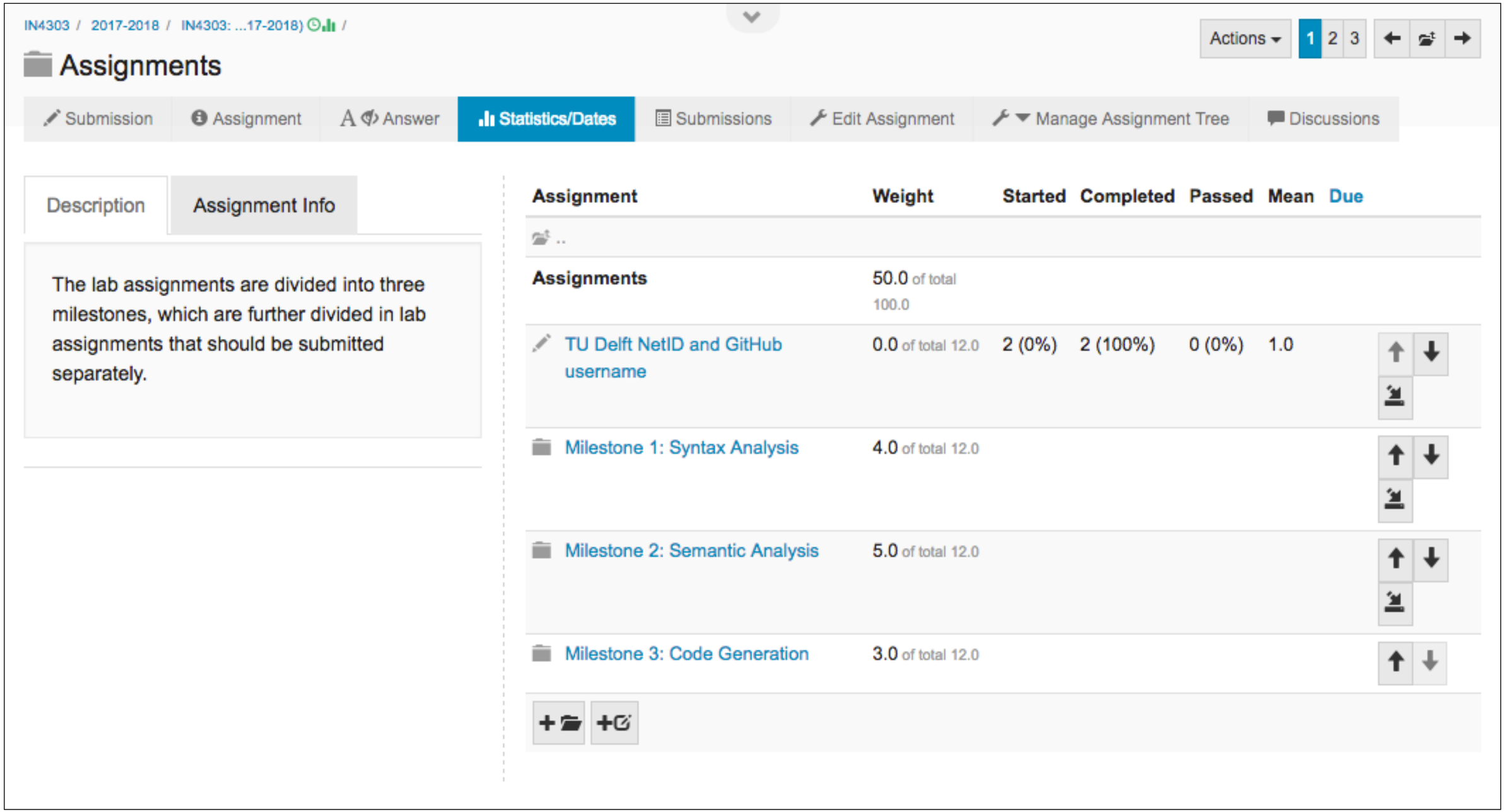
The deadline for submission is 20th September 2017, 23:59.

GitHub for Code



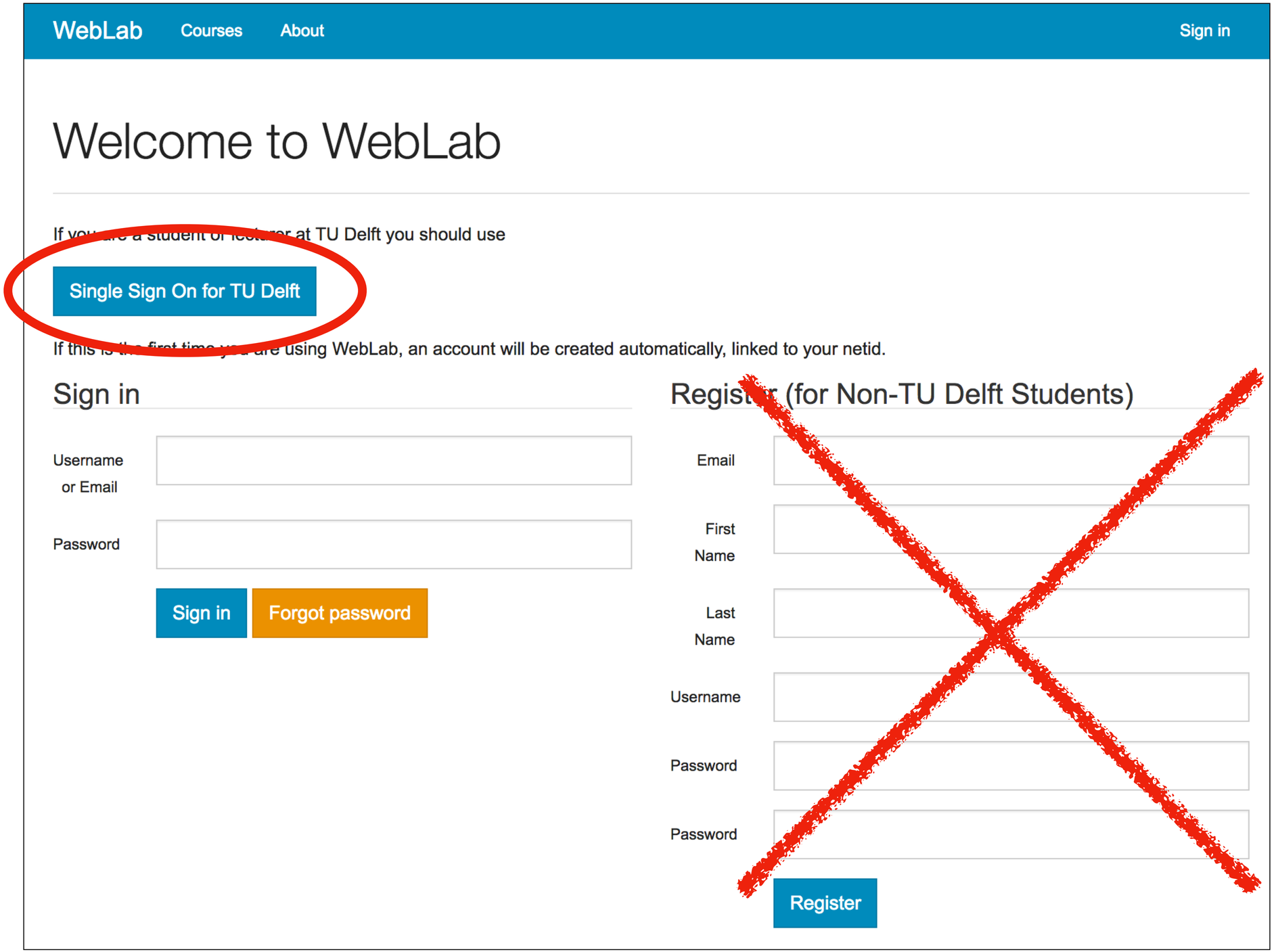
https://github.com/TUDelft-IN4303-2017

WebLab for Grade Registration

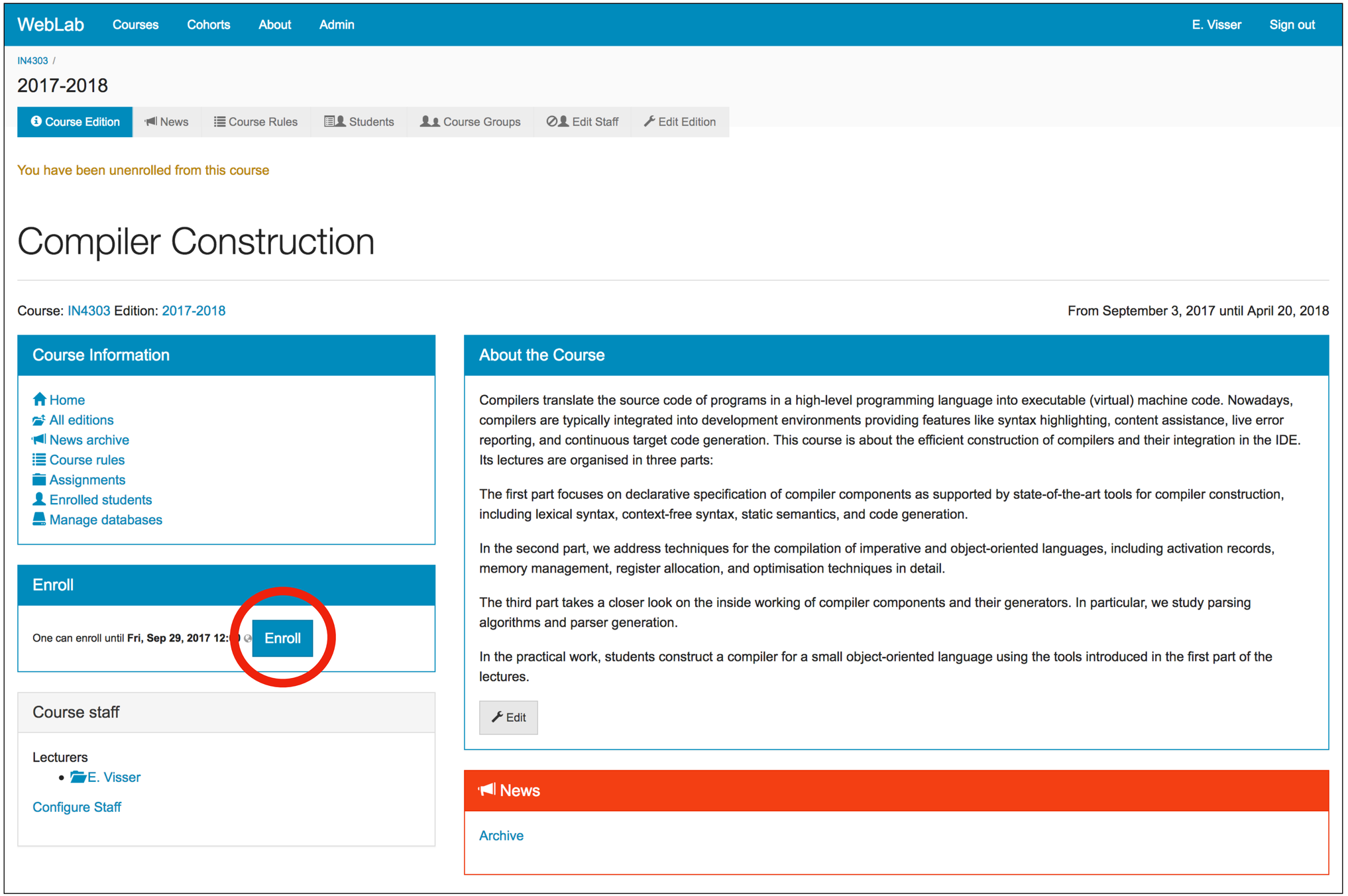


https://weblab.tudelft.nl/in4303/2017-2018/

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

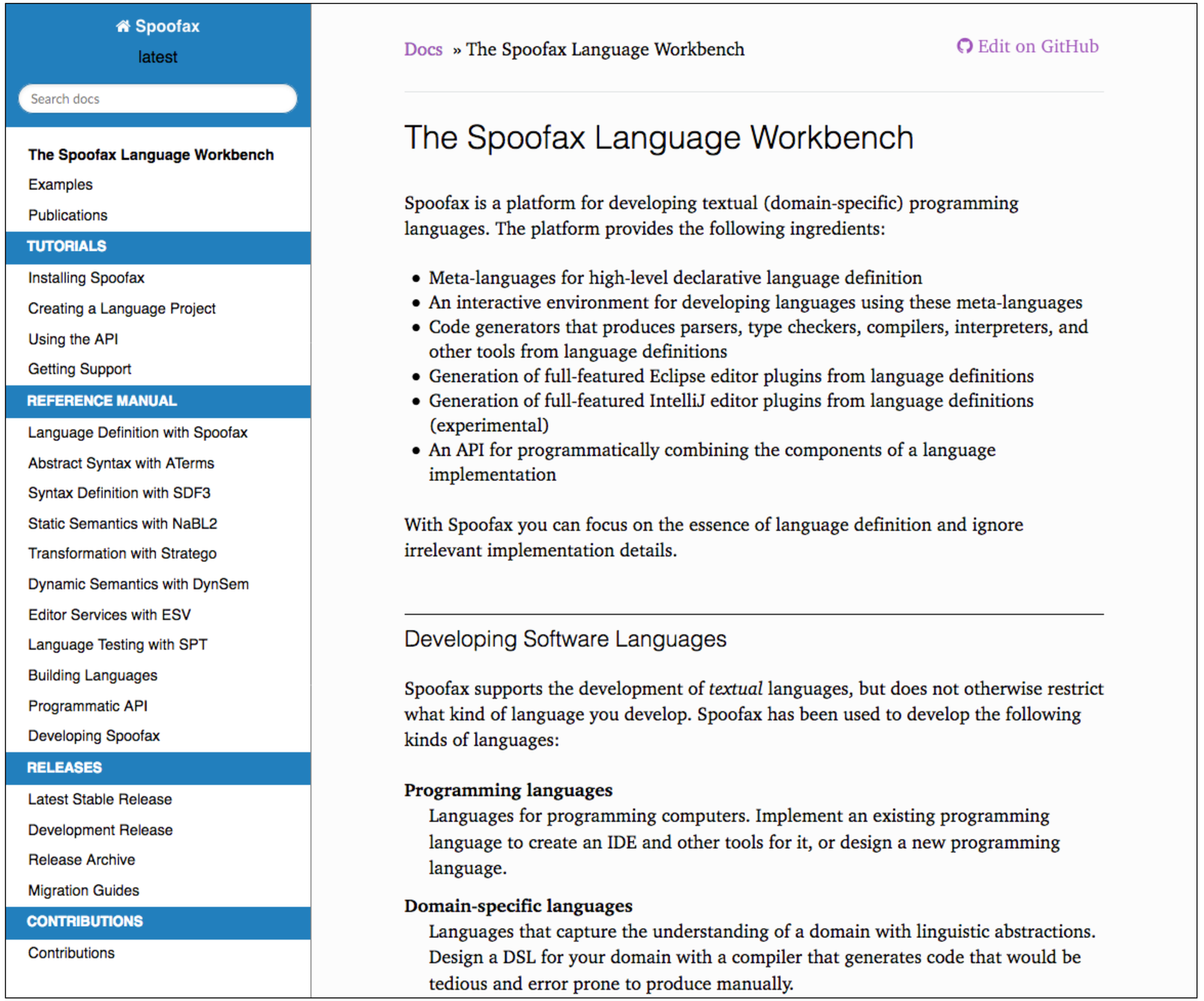


Enroll in WebLab



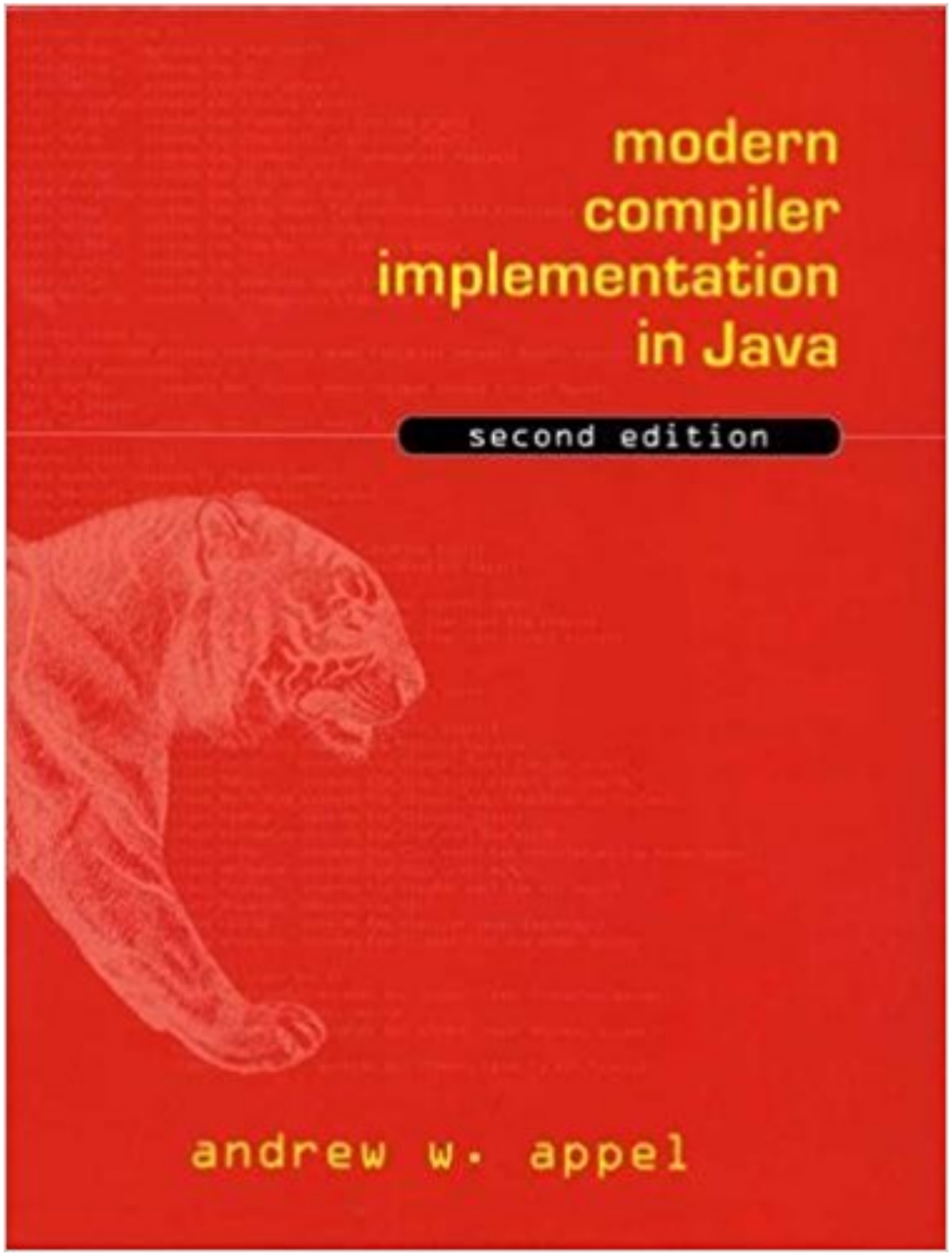
https://weblab.tudelft.nl/in4303/2017-2018/

Spoofax Documentation

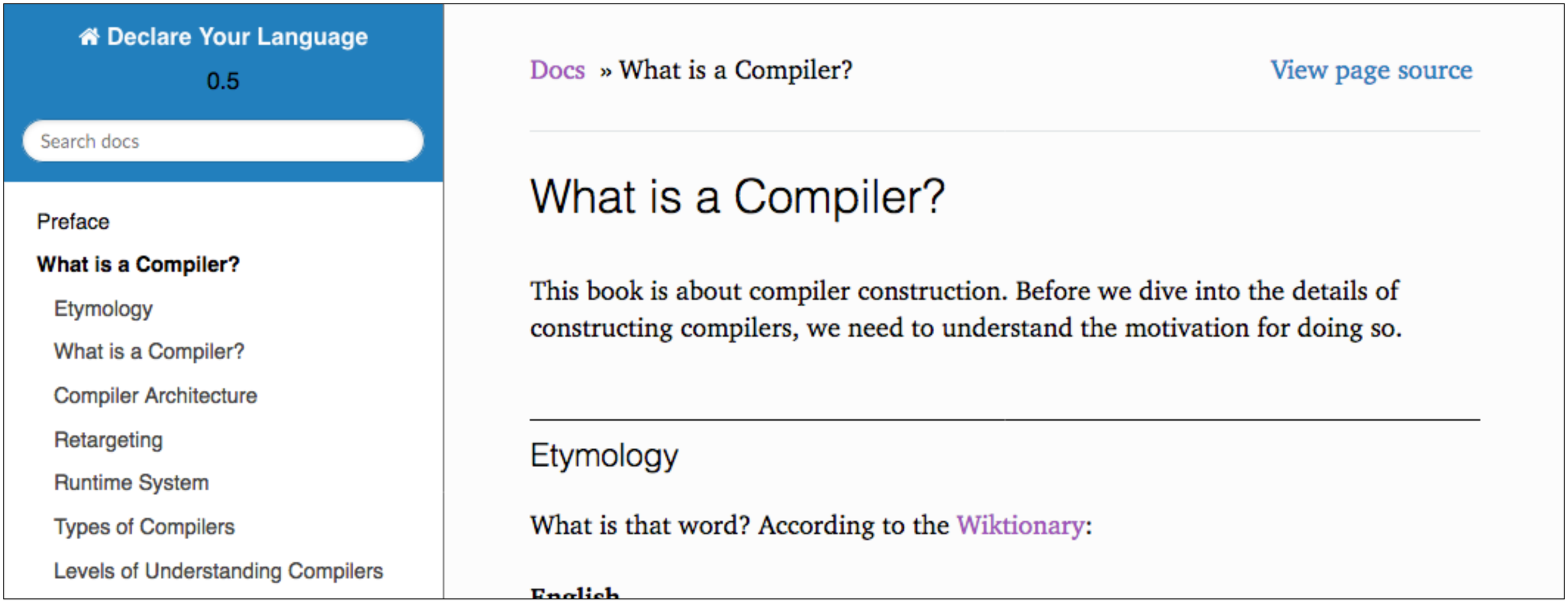


http://www.metaborg.org

Recommended Book



Lecture Notes (Under Construction)



http://www.declare-your-language.org

Grades for Lab Assignments

Grades

We grade the first final solution of each of your assignments. Grades reflect on correctness, clarity, shown programming skills, and readability of your submission. To pass the practical part of the course, you need to meet all of the following criteria:

1. You completed each assignment with a grade of 4.0 or better.
2. You completed the assignments of each milestone with an average grade of 5.0 or better.
3. You completed all the assignments with an average grade of 6.0 or better.

These rules allow you to compensate for lower grades in single assignments, but ensure a minimal quality in all your milestones as well as in your overall practical work.

https://tudelft-in4303-2017.github.io/assignments/

Deadlines

Deadlines 2017-2018

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 9 have a 24 hour deadline extension with a penalty of 2 points. Lab 9 has an extension until 12/01 with a penalty of 1 point.

These dates are subject to change.

- Milestone 1: Syntax Analysis
  - 20/09: Lab 1: Testing Syntax Analysis
  - 27/09: Lab 2: Syntax Definition
  - 04/10: Lab 3: Simple Term Rewriting
- Milestone 2: Semantic Analysis
  - 18/10: Lab 4: Testing Name Analysis
  - 27/10: Lab 5: Name Analysis
- Milestone 2: Semantic Analysis (continued)
  - 22/11: Lab 6: Testing Type Analysis
  - 06/12: Lab 7: Type Analysis
- Milestone 3: Code Generation
  - 13/12: Lab 8: Compiling Minimal Programs
  - 22/12: Lab 9: Compiling Expressions and Statements
  - 19/01: Lab 10: Compiling, Fields, Parameters, and Variables

https://tudelft-in4303-2017.github.io/assignments/

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 form 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!

What is a Compiler?

Etymology

Latin

Etymology

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

Pronunciation

- (Classical) IPA(key): /komˈpiː.loː/, [kɔmˈpiː.ɬoː]

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.

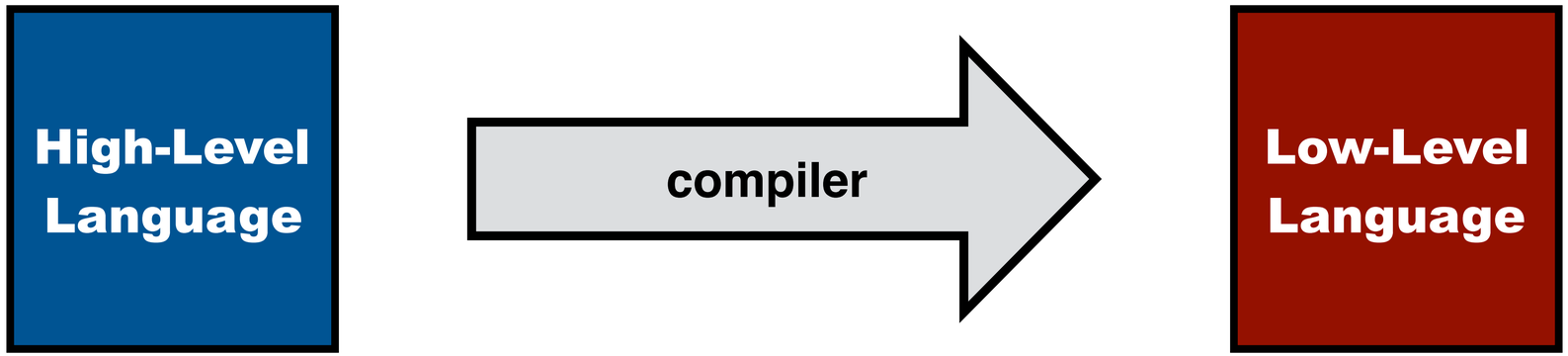
https://en.wiktionary.org/wiki/compile

Etymology

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.

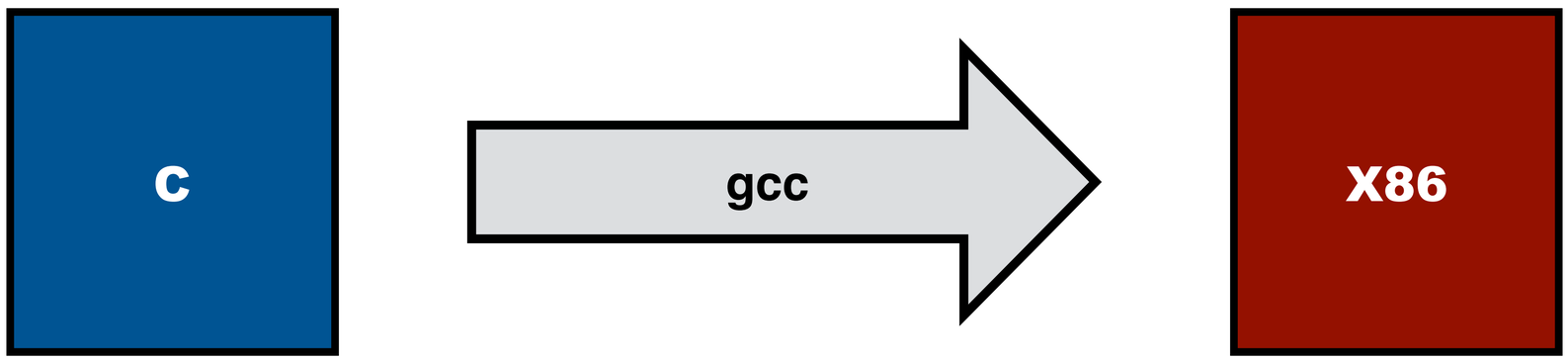
https://en.wikipedia.org/wiki/History_of_compiler_construction

Compiling = Translating



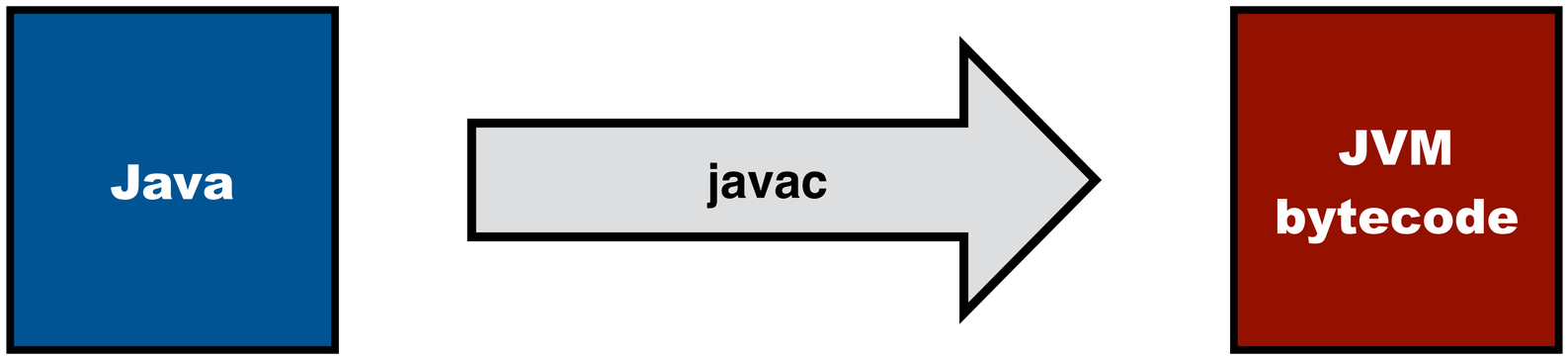
A compiler translates high-level programs to low-level programs

Compiling = Translating



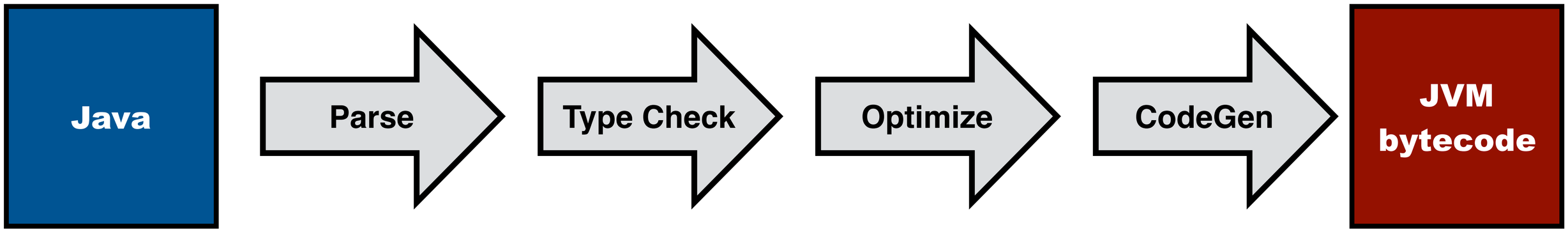
GCC translates C programs to object code for X86 (and other architectures)

Compiling = Translating



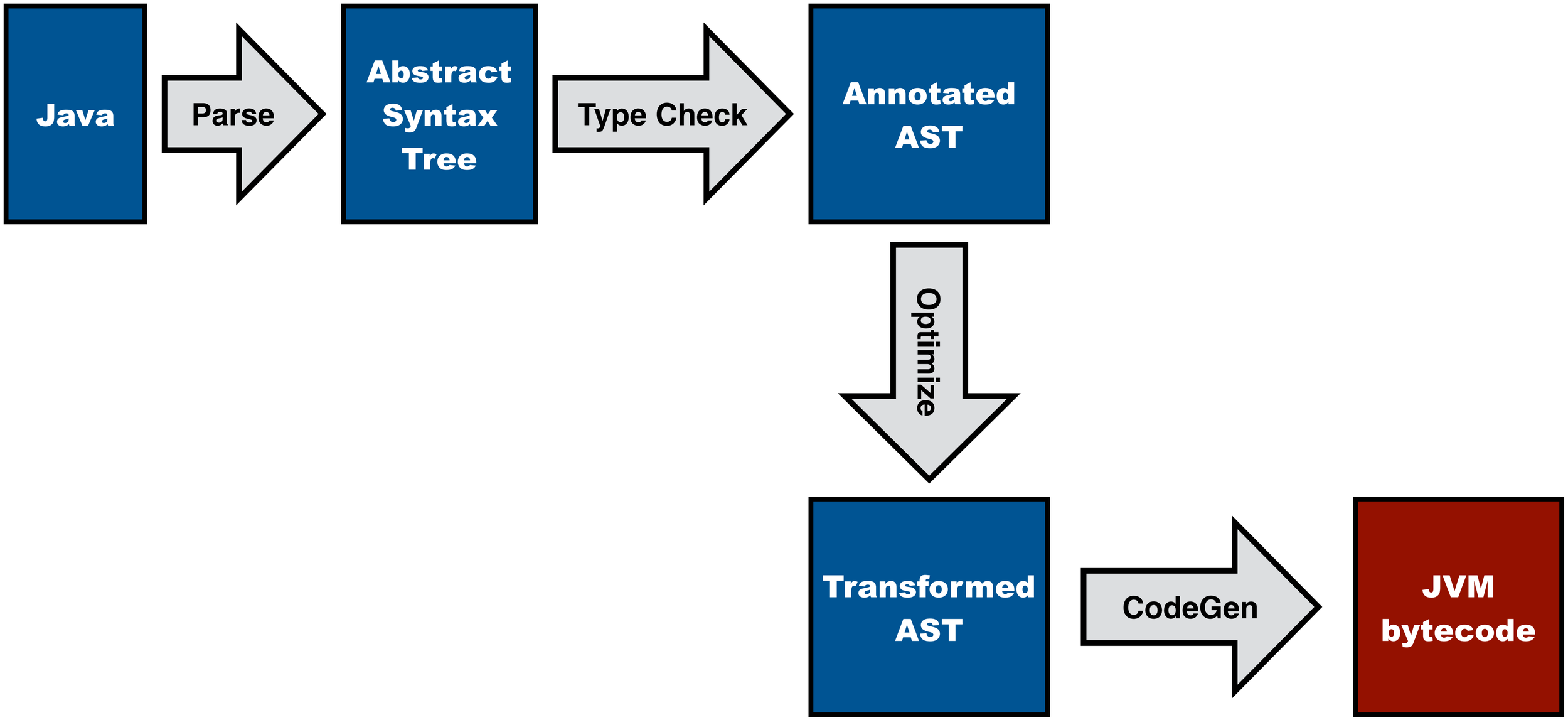
A Java compiler translates Java programs to bytecode instructions for Java Virtual Machine

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 (1)

Parser

- Reads in program text, checks that it complies with the syntactic rules of the language, and produces an abstract syntax tree, which represents the underlying (syntactic) structure of the program.

Type checker

- Consumes an abstract syntax tree and checks that the program complies with the static semantic rules of the language. To do that it needs to perform name analysis, relating uses of names to declarations of names, and checks that the types of arguments of operations are consistent with their specification.

Optimizer

- Consumes a (typed) abstract syntax tree and applies transformations that improve the program in various dimensions such as execution time, memory consumption, and energy consumption.

Code generator

- Transforms the (typed, optimized) abstract syntax tree to instructions for a particular computer architecture. (aka instruction selection)

Compiler Components (2)

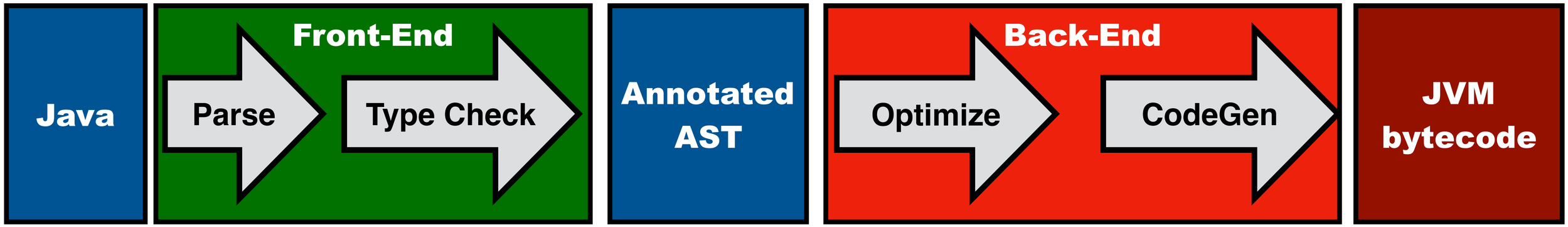
Register allocator

- Assigns physical registers to symbolic registers in the generated instructions.

Linker

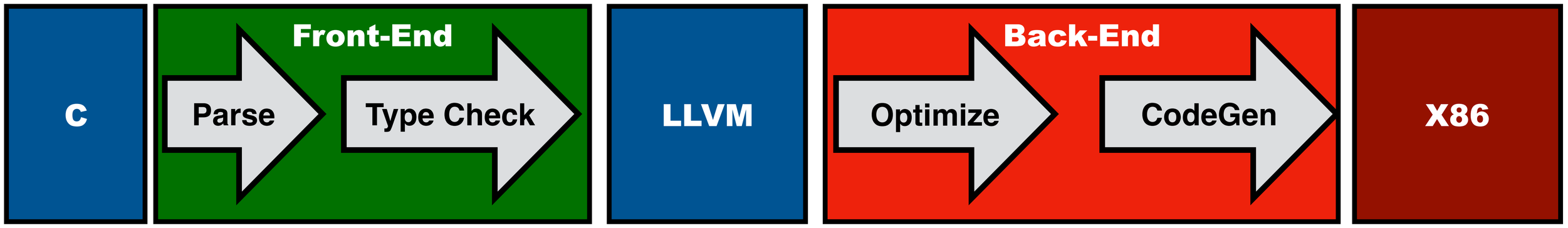
- Most modern languages support some form of modularity in order to divide programs into units. When also supporting separate compilation, the compiler produces code for each program unit separately. The linker takes the generated code for the program units and combines it into an executable program.

Compiler = Front-end + Back-End



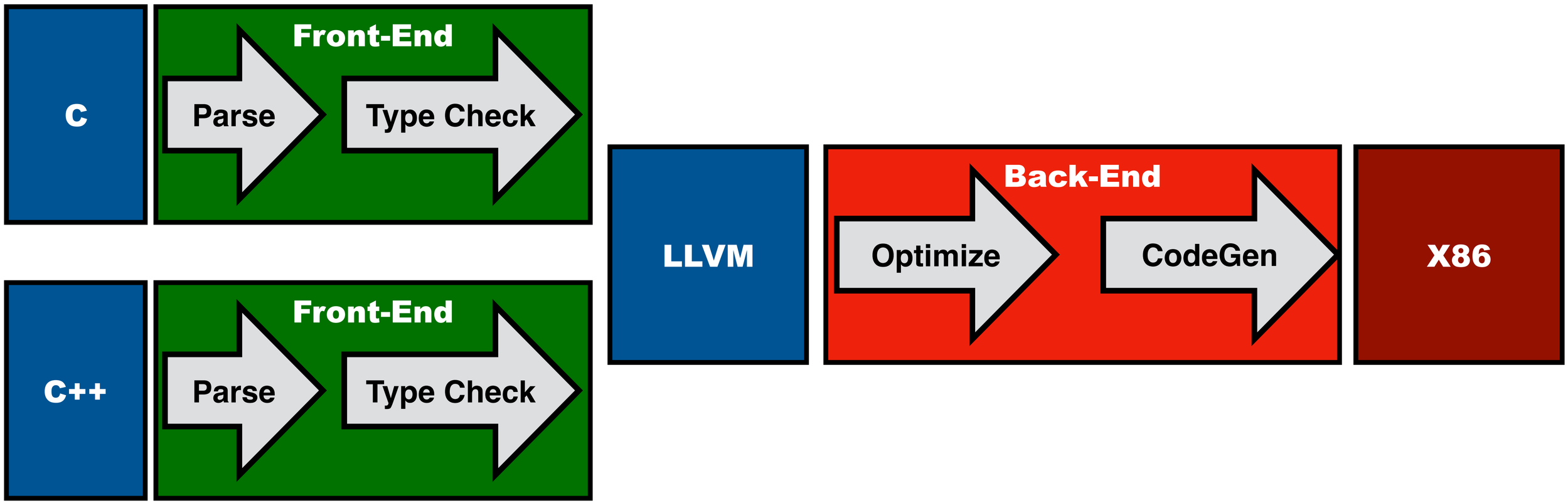
A compiler can typically be divided in a front-end (analysis) and a back-end (synthesis)

Compiler = Front-end + Back-End



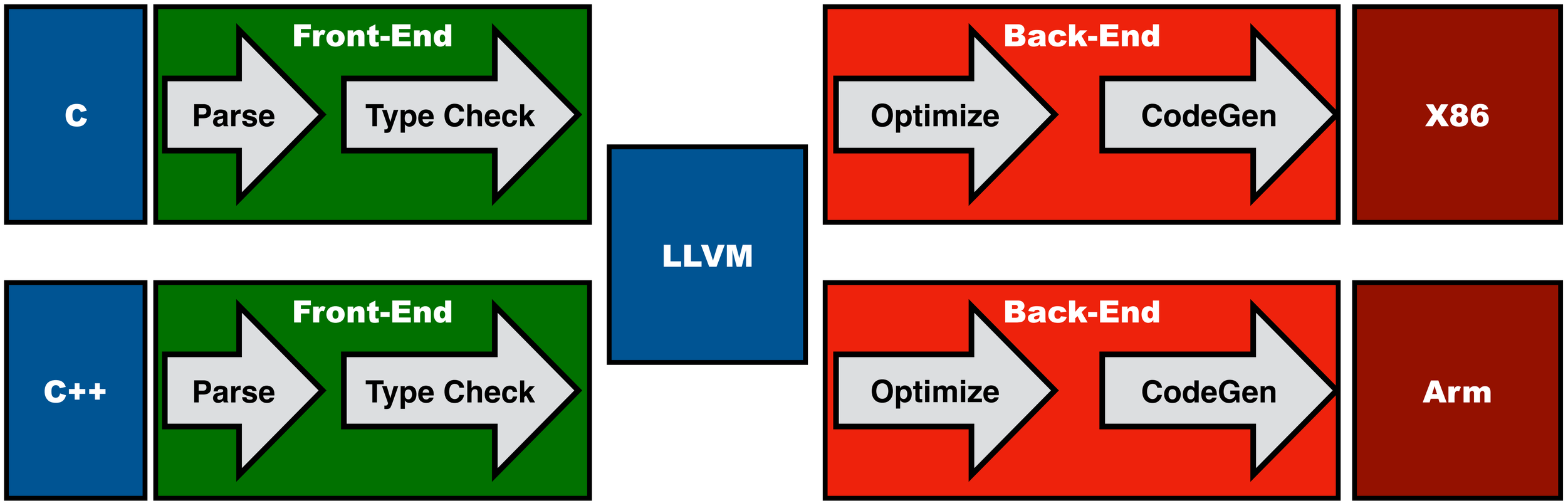
A compiler can typically be divided in a front-end (analysis) and a back-end (synthesis)

Repurposing Back-End



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

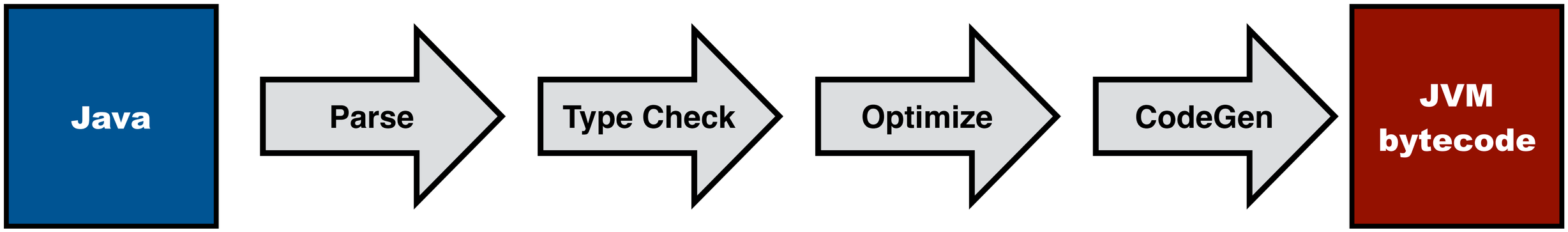
Retargeting Compiler



Retargeting: compile to different hardware architecture

What is a Compiler?

A bunch of components for translating programs



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

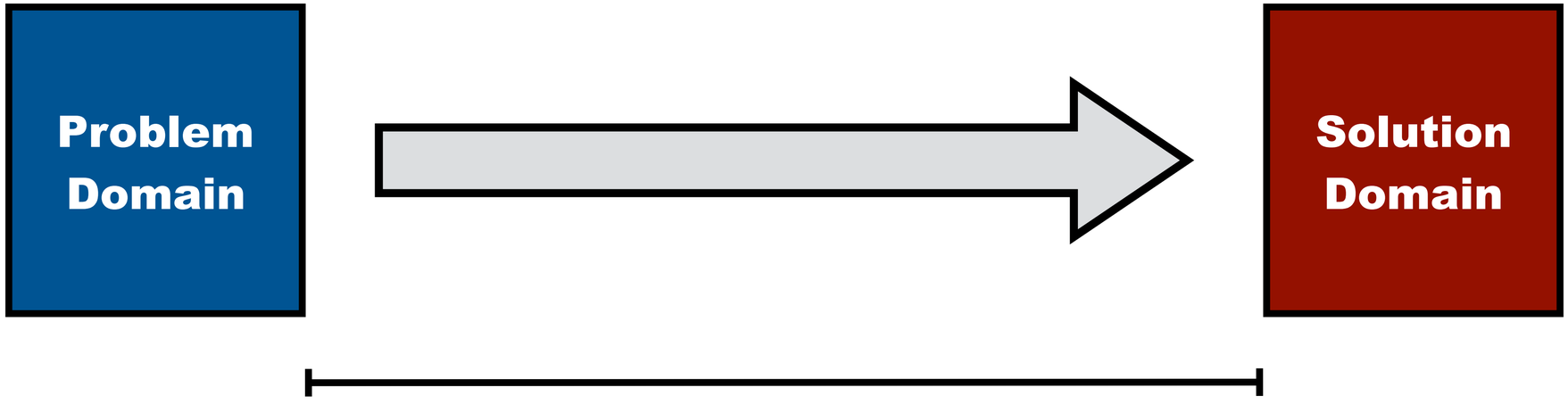
Why Compilers?

Programming = Instructing Computer

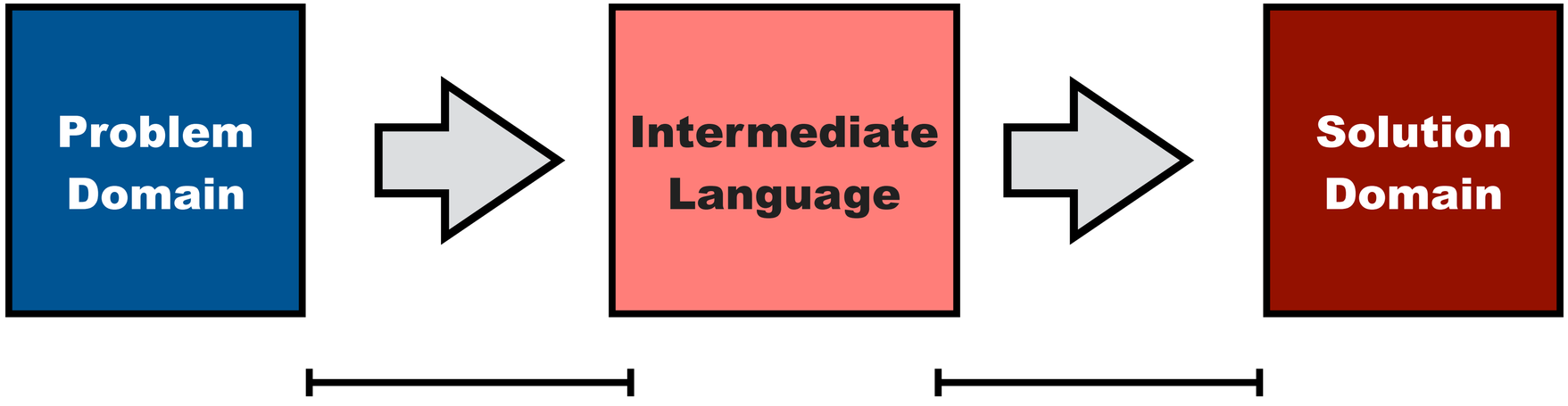
- fetch data from memory
- store data in register
- perform basic operation on data in register
- fetch instruction from memory
- update the program counter
- etc.

"*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
   - *the linguistic abstraction saved a lot of programming effort*
   - *he introduced a linguistic abstraction for page navigation in web programming*
2. the process of introducing linguistic abstractions
   - *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)
```

Source: http://sites.google.com/site/arch1utep/home/course_outline/translating-complex-expressions-into-assembly-language-using-expression-trees

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

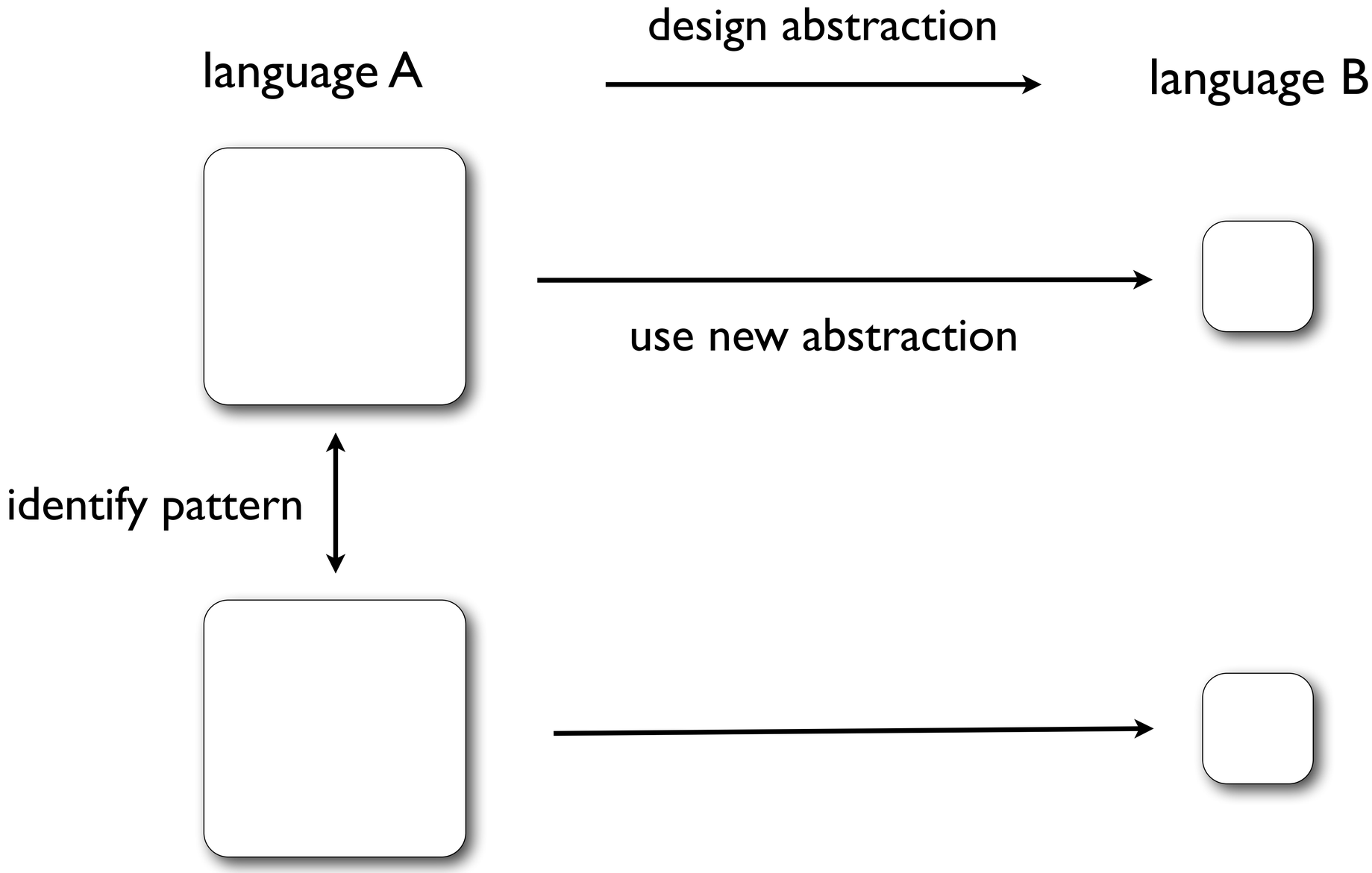
From Malloc to Garbage Collection

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

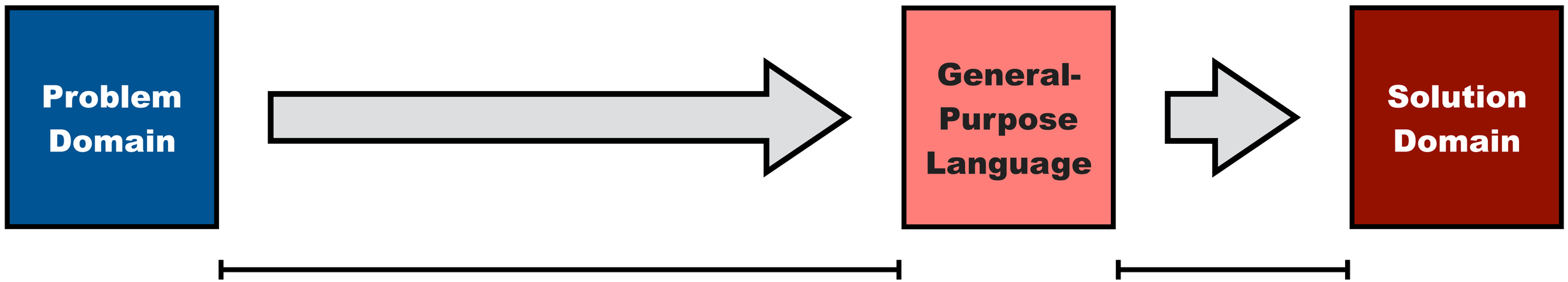
http://en.wikipedia.org/wiki/Malloc

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

Linguistic Abstraction



Compiler Automates Work of Programmer

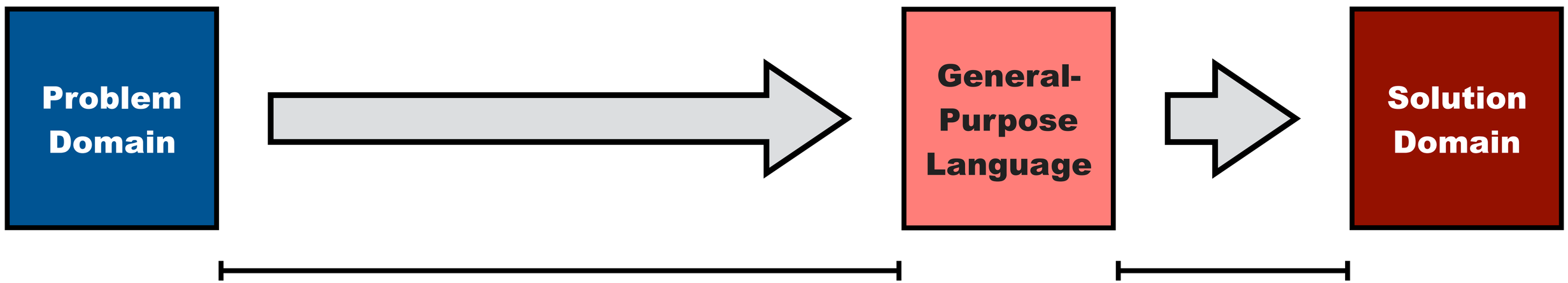


Compilers for modern high-level languages

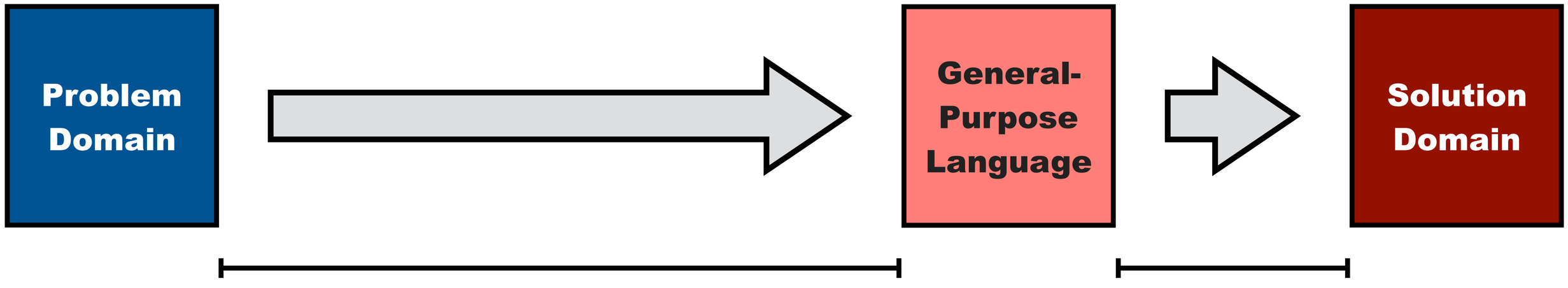
- Reduce the gap between problem domain and program
- Support programming in terms of computational concepts instead of machine concepts
- Abstract from hardware architecture (portability)
- Protect against a range of common programming errors

Domain-Specific Languages

Domains of Computation

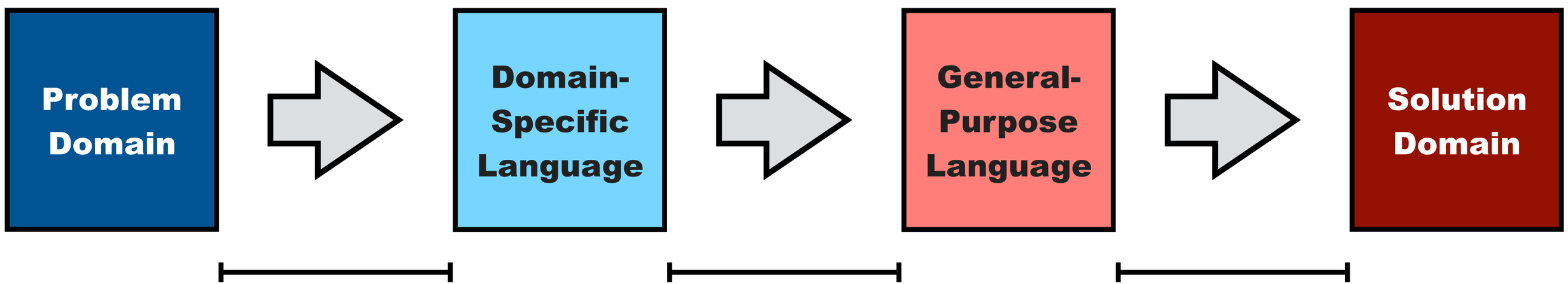


- Systems programming
- Embedded software
- Web programming
- Enterprise software
- Database programming
- Distributed programming
- Data analytics
- ...



"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

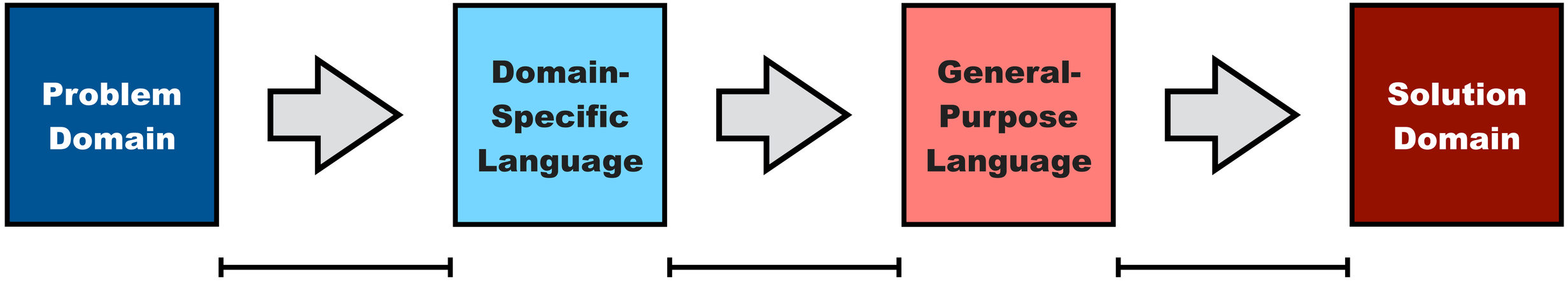
- What is the semantics of such definitions?
- How can we verify the correctness / consistency of language definitions?

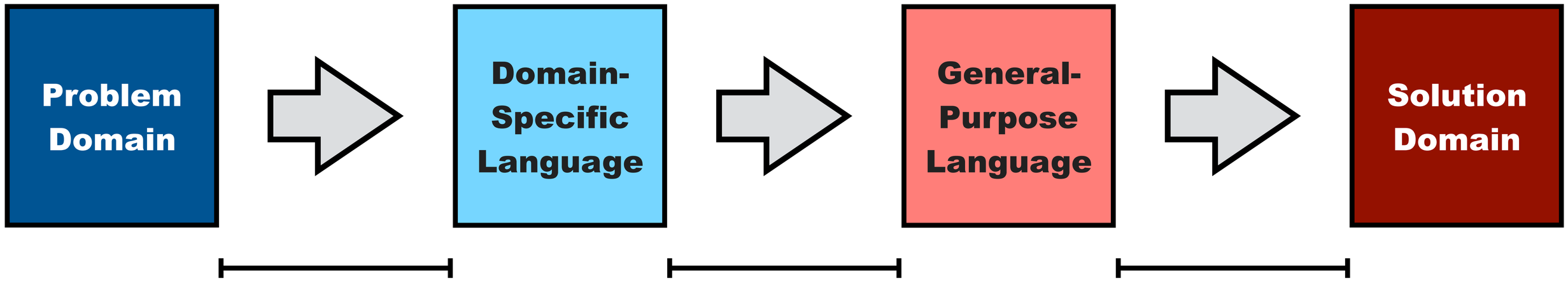
Implementation

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

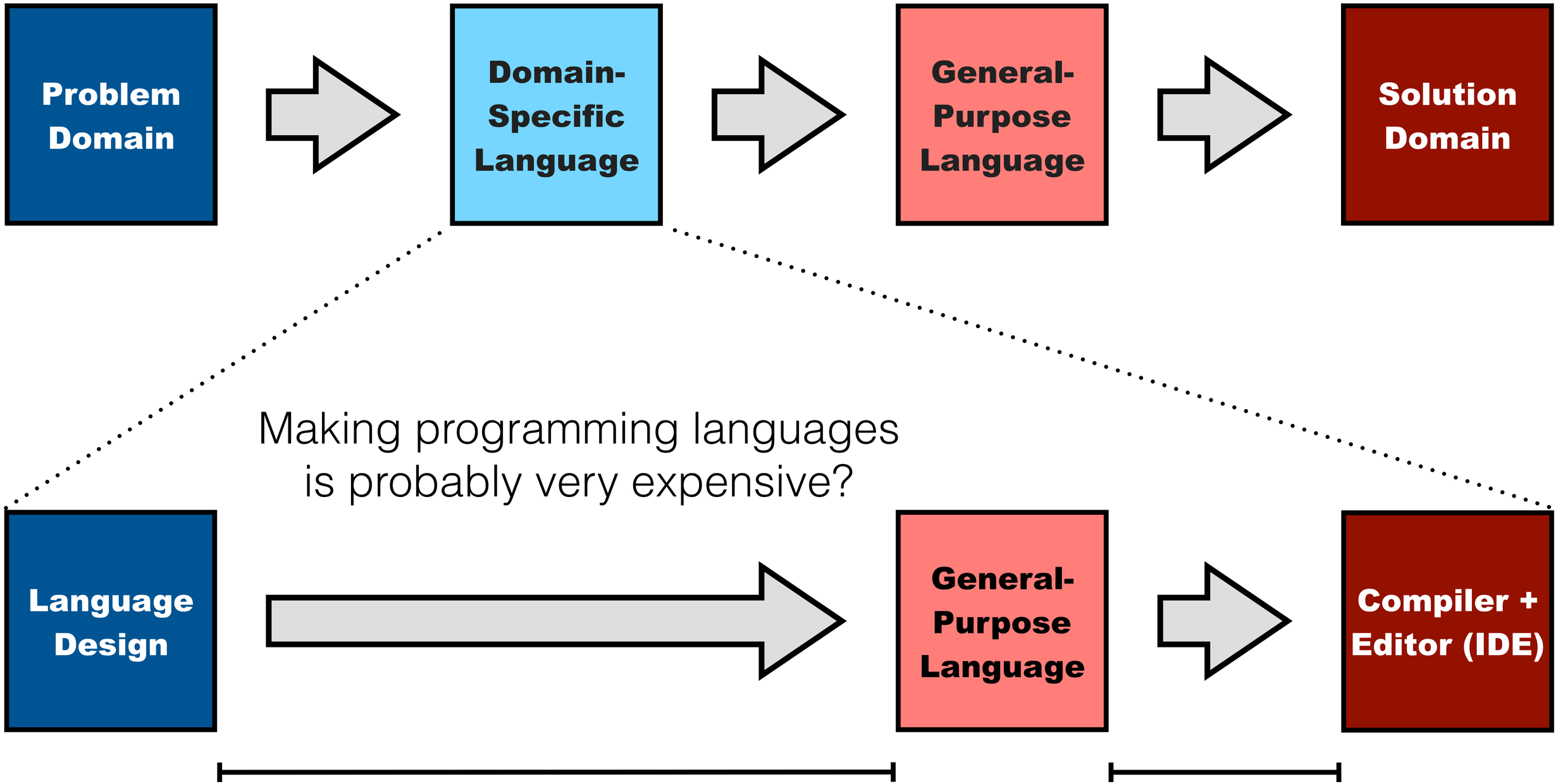
Evaluation

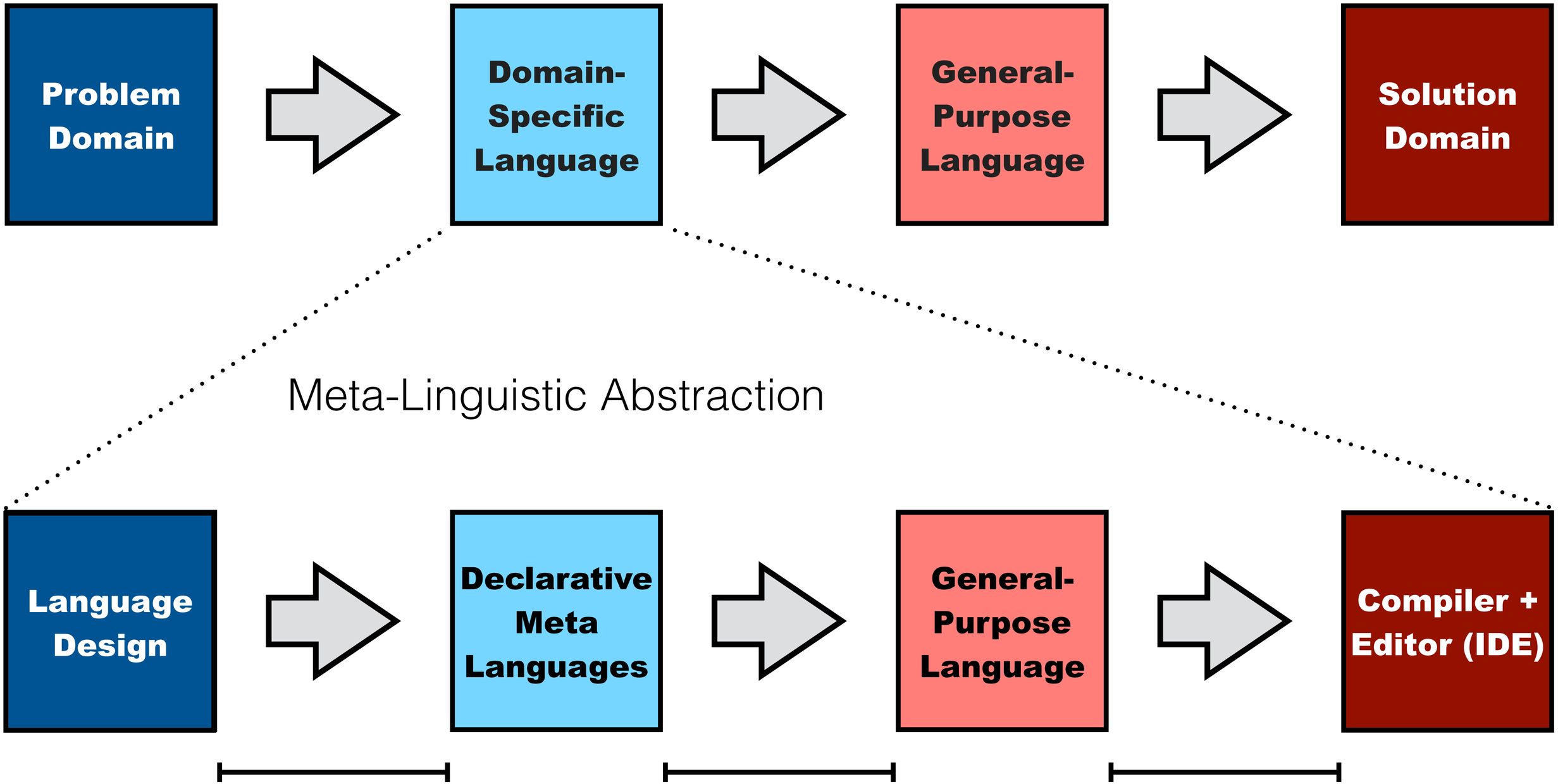
- Apply to new and existing languages to determine adequacy


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

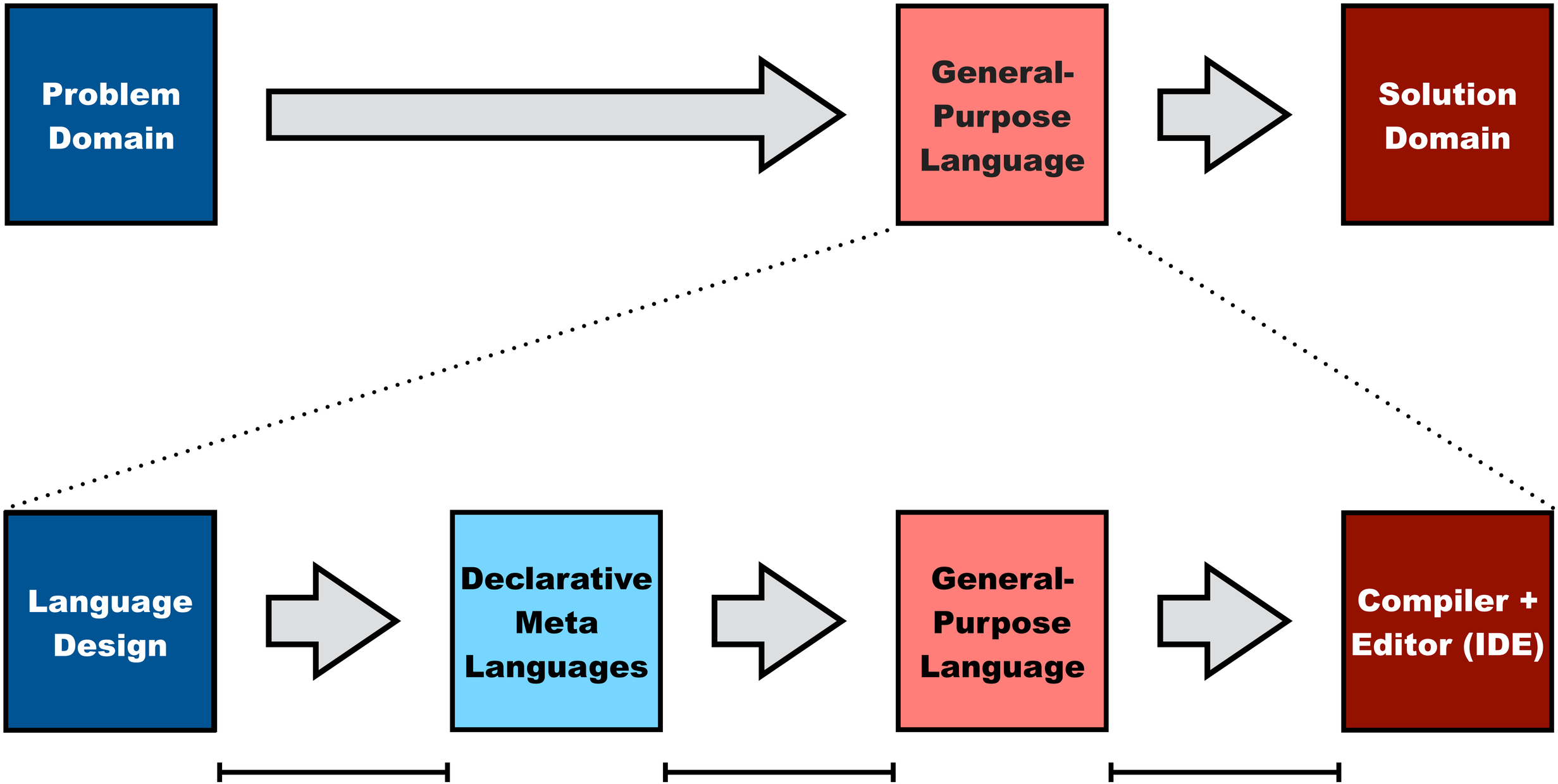


Making programming languages is probably very expensive?


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)



Applying compiler construction to the domain of compiler construction



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


Language Design
Declarative Meta Languages
Compiler + Editor (IDE)

Language Design

Syntax Definition

Static Semantics

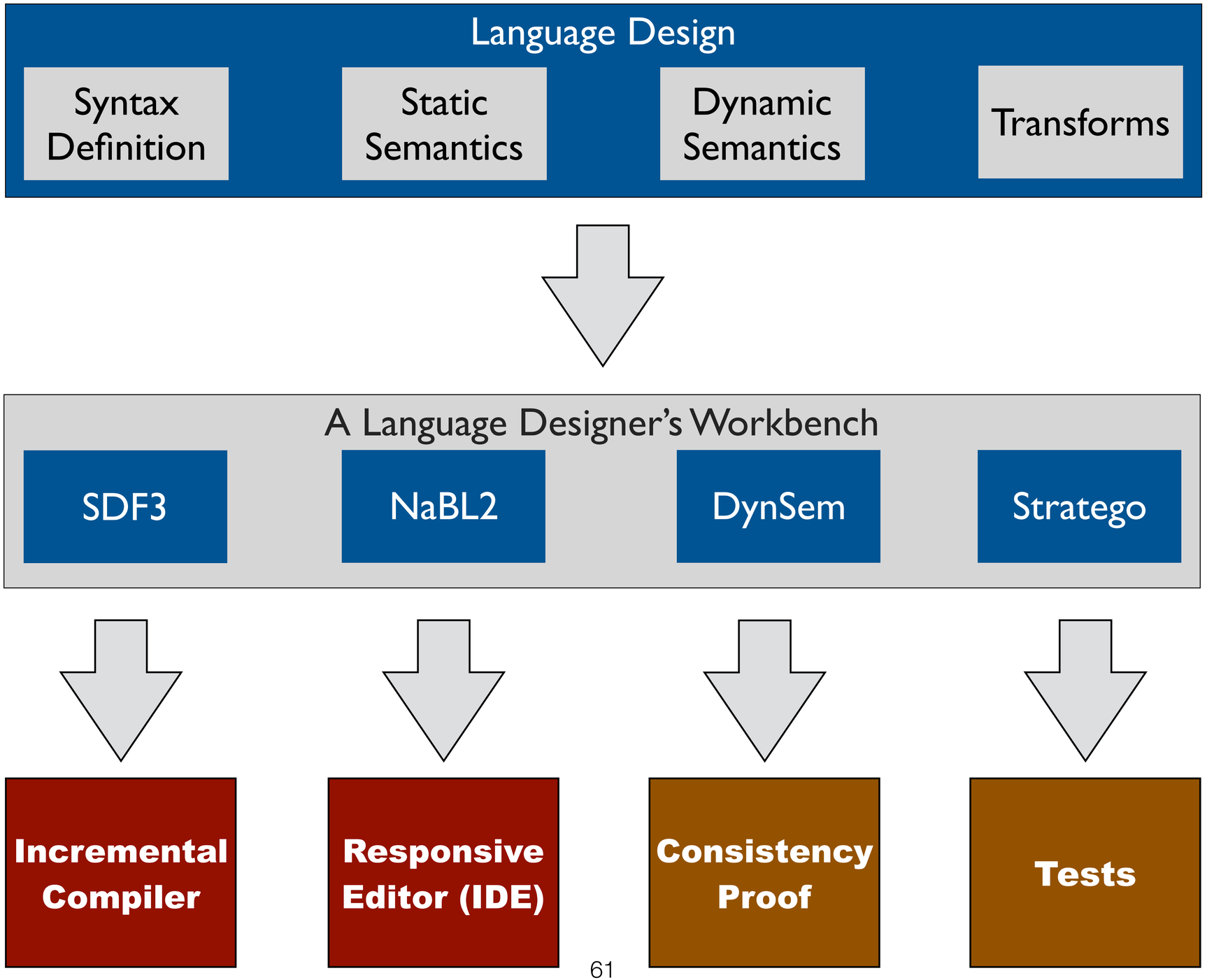
Dynamic Semantics

Transforms

Language Workbench

Meta-DSLs

Compiler + Editor (IDE)


Language Design
Syntax Definition
Static Semantics
Dynamic Semantics
Transforms
A Language Designer's Workbench
SDF3
NaBL2
DynSem
Stratego
Incremental Compiler
Responsive Editor (IDE)
Consistency Proof
Tests

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 Spoofax Language Workbench

SDF3: Syntax definition

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

NaBL2: 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 Spoofax 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 Spoofax Language Workbench

Calc.sdf3

```
  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, pref
  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 // booleans

  Exp.True  = <true>
  Exp.False = <false>
  Exp.Not   = <!<Exp>>
  Exp.And   = <<Exp> & <Exp>> {left}
  Exp.Or    = <<Exp> | <Exp>> {left}

  Exp.If = <
    if(<Exp>)
      <Exp>
    else
      <Exp>
  >

context-free syntax // variables and fu

  Exp.Var = ID
```

calc.nabl2

```
    [[ e ^ (s) : ty ]].

  Stat[[ Exp(e) ^ (s, s_nxt) ]] :=
    s == s_nxt,
    [[ e ^ (s) : ty ]].

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() ]].
  [[ Sub(e1, e2) ^ (s) : NumT() ]] :=
     [[ e1 ^ (s) : NumT() ]],
     [[ e2 ^ (s) : NumT() ]].
  [[ Div(e1, e2) ^ (s) : NumT() ]] :=
     [[ e1 ^ (s) : NumT() ]],
     [[ e2 ^ (s) : NumT() ]].

  [[ Eq(e1, e2) ^ (s)  : BoolT() ]] :=
     [[ e1 ^ (s) : NumT() ]],
     [[ e2 ^ (s) : NumT() ]].
  [[ Lt(e1, e2) ^ (s)  : BoolT() ]] :=
     [[ e1 ^ (s) : NumT() ]],
     [[ e2 ^ (s) : NumT() ]].
```

transform.str

```
module transform

imports
  nabl2shared
  nabl2runtime
  transform/desugar
  statics/calc
  pp

rules // Desugaring

  // Desugar
  desugar-aterm: (selected, _, _, path
    with filename := <guarantee-extens
       ; result   := <desugar-calc> se

  // Desugar and pretty-print
  desugar-pp: (selected, _, _, path, p
    with filename := <guarantee-extens
       ; result   := <desugar-calc; pp
```

eval.ds

```
  Add(NumV(i), NumV(j)) --> NumV(addB(i, j))

  Lt(NumV(i), NumV(j)) --> BoolV(ltB(i, j))
  Eq(NumV(i), NumV(j)) --> BoolV(eqB(i, j))

signature
  arrows
    ift(Value, Exp, Exp) --> Value
rules // booleans

  False() --> BoolV(false)
  True() --> BoolV(true)

  If(v1, e2, e3) --> ift(v1, e2, e3)

  ift(BoolV(true), e2, _) --> e2
  ift(BoolV(false), _, e3) --> e3

rules // variables and functions

  E |- Var(x) --> E[x]

  E |- Fun([x], e) --> ClosV(x, e, E)

  E |- Let(x, v1, e2) --> v
  where E {x |--> v1, E} |- e2 --> v

  App(ClosV(x, e, E), v_arg) --> v
  where E {x |--> v_arg, E} |- e --> v

signature
  arrows
```

java.str

```
module codegen/java

imports
  signatures/-
  nabl2/api

rules // programs

  program-to-java :
    Program(stats) -> $[import j
                        import j

                       public c
                         public
                           [jst
                           retu
                         }
                       }
                      ]
    with
      <last> stats => last
    ; <return-type; type-to-java
    ; <stats-to-java> stats => j
    ; <return-exp; exp-to-java>

  return-type :
    Bind(_, e) -> ty
    with
      <nabl2-get-ast-type> e =>

  return-type :
    Exp(e) -> ty
```

mortgage.calc

```
rY = 0.017;  // yearly interest
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
```

SDF3: Syntax Definition + **NaBL2: Static Semantics** + **DynSem: Dynamic Semantics** + **Stratego: Program Transformation** → **Programming Environment**

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) instOf ty_fun.
```

Calc: Dynamic Semantics

```
rules // numbers

  Num(n) --> NumV(parseB(n))

  Pow(NumV(i), NumV(j)) --> NumV(powB(i, j))
  Mul(NumV(i), NumV(j)) --> NumV(mulB(i, j))
  Div(NumV(i), NumV(j)) --> NumV(divB(i, j))
  Sub(NumV(i), NumV(j)) --> NumV(subB(i, j))
  Add(NumV(i), NumV(j)) --> NumV(addB(i, j))

  Lt(NumV(i), NumV(j)) --> BoolV(ltB(i, j))
  Eq(NumV(i), NumV(j)) --> BoolV(eqB(i, j))

rules // variables and functions

  E |- Var(x) --> E[x]

  E |- Fun([x], e) --> ClosV(x, e, E)

  E |- Let(x, v1, e2) --> v
  where E {x |--> v1, E} |- e2 --> v

  App(ClosV(x, e, E), v_arg) --> v
  where E {x |--> v_arg, E} |- e --> 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

Functions.sdf3 | Records.sdf3 | Arrays.sdf3

```
module Functions

imports Identifiers
imports Types

context-free syntax

  Dec.FunDecs = <<{FunDec "\n"}+>> {longest-

  FunDec.ProcDec = <
    function <Id>(<{FArg ", "}*>) =
      <Exp>
  >

  FunDec.FunDec = <
    function <Id>(<{FArg ", "}*>) : <Type> =
      <Exp>
  >

  FArg.FArg = <<Id> : <Type>>

  Exp.Call = <<Id>(<{Exp ", "}*>)>
```

functions.nabl2

```
module statics/functions

imports signatures/Functions-sig
imports statics/nabl-lib
imports statics/base

rules // function declarations

  // Parameters: In [function id(... id1: id
  // scope of the parameter id1 lasts throug
  // body exp

  Dec[[ FunDecs(fdecs) ^ (s, s_outer) ]] :=
     Map2[[ fdecs ^ (s, s_outer) ]].

  [[ ProcDec(f, args, e) ^ (s, s_outer) ]] :
  [[ ProcDec(f, args, e) ^ (s, s_outer) ]] :
//     Args[[ args ^ (s_fun, s_outer) : tys
     new s_fun,
     s_fun -P-> s,
     distinct/name D(s_fun) | error $[duplic
     MapTs2[[ args ^ (s_fun, s_outer) : tys
//
     Var{f} <- s,
     Var{f} : FUN(tys, UNIT()) !,
     [[ e ^ (s_fun) : UNIT() ]].

//  [[ FunDec(f, args, t, e) ^ (s, s_outer)
//     new s_fun,
//     s_fun -P-> s,
//     distinct/name D(s_fun) | error $[dupl
//     MapTs2[[ args ^ (s_fun, s_outer) : ty
```

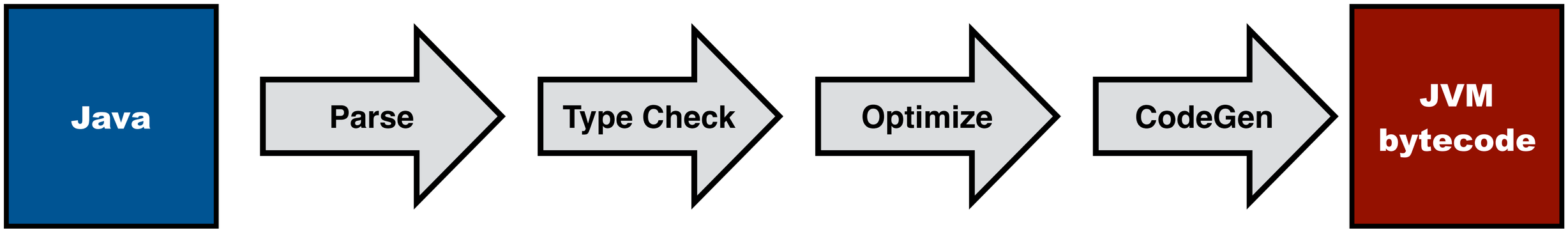
queens.tig

```
/* A program to solve the 8-queens problem */

let
    var N := 8

    type intArray = array of int

    var row := intArray [ N ] of 0
    var col := intArray [ N ] of 0
    var diag1 := intArray [N+N-1] of 0
    var diag2 := intArray [N+N-1] of 0

    function printboard() =
       (for i := 0 to N-1
     do (for j := 0 to N-1
          do print(if col[i]=j then " O" else " .");
        print("\n"));
        print("\n"))

    function try(c:int) =
(    if c=N
     then printboard()
     else for r := 0 to N-1
       do if row[r]=0 & diag1[r+c]=0 & diag2[r+7-c]=0
          then (row[r]:=1; diag1[r+c]:=1; diag2[r+7-c]:=1;
            col[c]:=r;
                try(c+1);
            row[r]:=0; diag1[r+c]:=0; diag2[r+7-c]:=0)
)
 in try(0)
end
```

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

Lectures (Tentative)

Q1

- What is a compiler? (Introduction)
- Syntax Definition
- Basic Parsing
- Term Rewriting
- Static Semantics & Name Resolution
- Type Constraints
- Constraint Resolution

Q2

- Dynamic Semantics
- Virtual Machines & Code Generation
- Just-in-Time Compilation (Interpreters & Partial Evaluation)
- Data-Flow Analysis
- Garbage Collection
- Advanced Parsing
- Overview

Lectures: Tuesday, 17:45 in Lecture Hall Pi at EWI

Lectures are recorded with Collegerama; but only if attendance is sufficient