Lecture 1: What is a Compiler?

CS4200 Compiler Construction

Eelco Visser

TU Delft

September 2018

TU Delft

Course Organization

No use of electronic devices during lectures

Eelco Visser About Research Teaching Contact Blog

Eelco Visser

News

- August 24, 2018: three papers accepted at SLE 2018:
  - Declarative specification of indentation rules: A tooling perspective on parsing and pretty-printing layout-sensitive languages by Eduardo Amorim, Michael Steindorfer, Sebastian Erdweg, and Eelco Visser
  - Migrating custom DSL implementations to a language workbench: An industrial tool demonstration by Jasper Denkers, Louis van Gool, and Eelco Visser
  - Migrating business logic to an incremental computing DSL: A case study by Daco Harkes, Elmer van Chastelet, and Eelco Visser
- August 16, 2018: Paper Scopes as Types by Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser accepted for OOPSLA 2018. The artifact accompanying the paper has also been accepted as Functional and Reusable
- July 23, 2018: The video of my talk at Curry On 2018 in Amsterdam is now online
- July 18, 2018: Congratulations to Robbert Krebbers for his NWO Veni grant for Verified programming language interaction
- July 6, 2018: Paper "Specializing a Meta-Interpreter" by Vlad Vergu and Eelco Visser accepted for ManLang 2018
- July 5, 2018: Paper Scopes as Types by Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser conditionally accepted for OOPSLA 2018
- July 3, 2018: Paper "Scalable Incremental Building with Dynamic Task Dependencies" by Gabriël Konat, Sebastian Erdweg, and Eelco Visser accepted for ASE 2018
- February 5, 2018: Paper PIE: A Domain-Specific Language for Interactive Software Development Pipelines by Gabriël Konat, Michael J. Steindorfer, Sebastian Erdweg and Eelco Visser accepted for ‹Programming› 2018
- February 5, 2018: Paper Towards Zero-Overhead Disambiguation of Deep Priority Conflicts by Luís Eduardo de Souza Amorim, Michael J. Steindorfer and Eelco Visser accepted for ‹Programming› 2018
- January 1, 2018: I have been invited to talk at Curry On 2018 in Amsterdam
- December 19, 2017: Paper PixieDust: Declarative Incremental User Interface Rendering through Static Dependency Tracking by Nick ten Veen, Daco Harkes and Eelco Visser accepted for WWW 2018 track on Web Programming
- September 26, 2017: Paper Intrinsically Typed Definitional Interpreters for Imperative Languages by Casper Bach Poulsen, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers, and Eelco Visser accepted at POPL 2018 [video] [paper]

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

CS4200: Two Courses

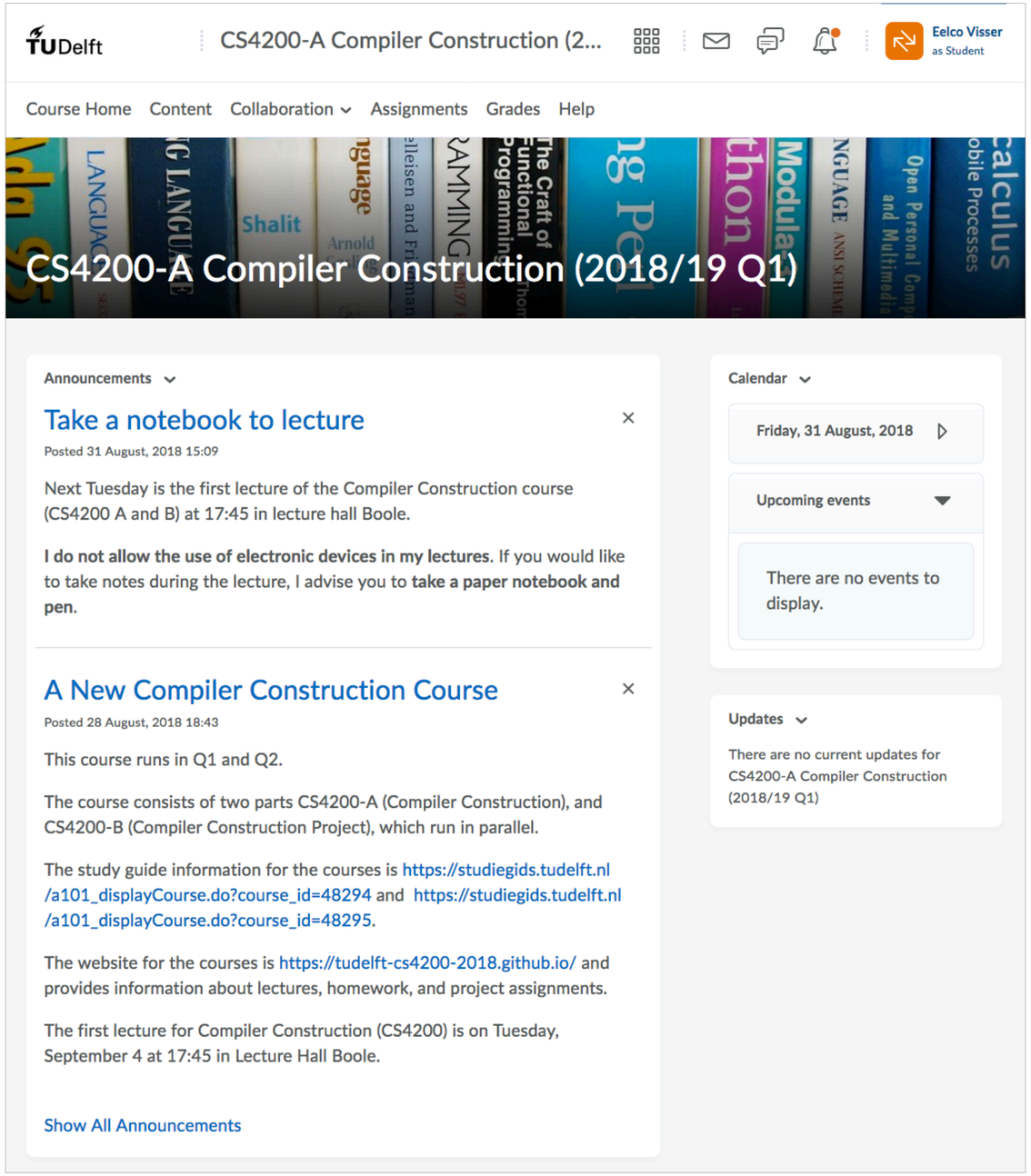
CS4200-A: Compiler Construction (5 ECTS)

- 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

Brightspace: Announcements



Course Website

TUDELFT CS4200 LECTURES ASSIGNMENTS PROJECT 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 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:

https://tudelft-cs4200-2018.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 2018-2019

We offer walk-in hours on Wednesday mornings 9:30-11:00 in room E4.420 of Building 28

Team Members

Eelco Visser

responsible instructor

Building 28. E4.340 Email Twitter

Jasper Denkers bio photo

Jasper Denkers

graduate teaching assistant

Building 28. E4.420 Email

Jeff Smits bio photo

Jeff Smits

graduate teaching assistant

Building 28. E4.420 Email

Hendrik van Antwerpen bio photo

Hendrik van Antwerpen

graduate teaching assistant

Building 28. E4.420 Email

Taico Aerts bio photo

Taico Aerts

student teaching assistant

Email

TUDELFT CS4200

LECTURES ASSIGNMENTS PROJECT CONTACT

Architecture Syntax Transformation Statics Dynamics

Compiler Construction

Twitter

Github

Edit on GitHub

Lectures

(Tentative) Schedule 2018-2019

Q1

- Tue Sep 4, 17:45-19:30, EWI Bool: What is a compiler?
- Fri Sep 7, 17:45-19:30, EWI Chip: Syntax Definition
- Tue Sep 11, 17:45-19:30, EWI Bool: Parsing
- Tue Sep 18, 17:45-19:30, EWI Bool: Syntactic (Editor) Services
- Tue Oct 25, 17:45-19:30, EWI Bool: Transformation
- Tue Oct 2, 17:45-19:30, EWI Bool: Name Resolution
- Tue Oct 9, 17:45-19:30, EWI Bool: Type Constraints
- Tue Oct 16, 17:45-19:30, EWI Bool: Constraint Resolution I
- Tue Oct 23, 17:45-19:30, EWI Bool: Constraint Resolution II
- Thu, Nov 1, 13:30-16:30, LR-CZ J: midterm exam

Q2

- Tue Nov 13, 10:45-12:30, CT-CZ D: Data-flow Analysis
- Tue Nov 20, 10:45-12:30, CT-CZ D: Virtual Machines & Code Generation
- Tue Nov 27, 10:45-12:30, CT-CZ D: Optimization
- Tue Dec 4, 10:45-12:30, CT-CZ D: Garbage Collection
- Tue Dec 11, 10:45-12:30, CT-CZ D: Dynamic Semantics & Interpreters
- Tue Dec 18, 10:45-12:30, CT-CZ D: Applications
- Tue Jan 8, 10:45-12:30, CT-CZ D: Reserve
- Tue Jan 15, 10:45-12:30, CT-CZ D: Overview
- Tue, Jan 29, 18:30-21:30, DW-IZ 1: final exam

CS4200-A: Homework Assignments

CS4200-B: Project

TUDELFT CS4200

LECTURES ASSIGNMENTS PROJECT CONTACT

Syntactic Analysis Semantic Analysis Code Generation Documentation

Compiler Construction

Twitter
Github

Edit on GitHub

OVERVIEW
Submission
Grades
Deadlines 2018-2019
Academic Misconduct
Retry

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

Submission

We rely on GitLab for assignment submissions. We assign a private GitLab 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.

CS4200-B: Project

TUDELFT CS4200

LECTURES ASSIGNMENTS PROJECT CONTACT

Syntactic Analysis Semantic Analysis Code Generation Documentation

lab 1 *lab 2* *lab 3* *FAQ*

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.

Note: The deadlines for Lab 1 and Lab 2 are at the same date so that you can develop tests and syntax definition together.

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`.

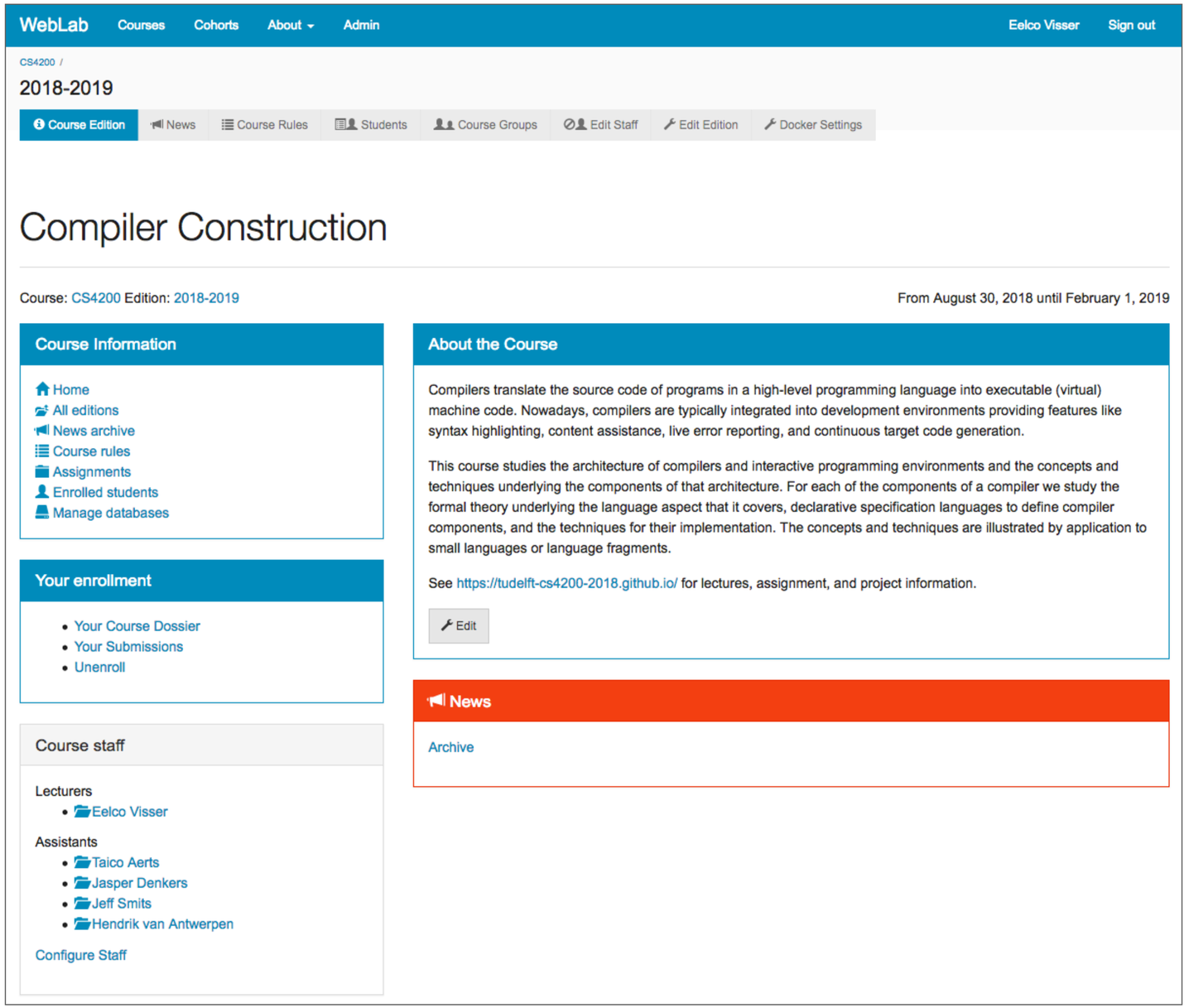
   Test cases for `FieldDecl` and `ParamDecl` should be covered in the tests for `ClassDecl` and `MethodDecl`, respectively.

   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.

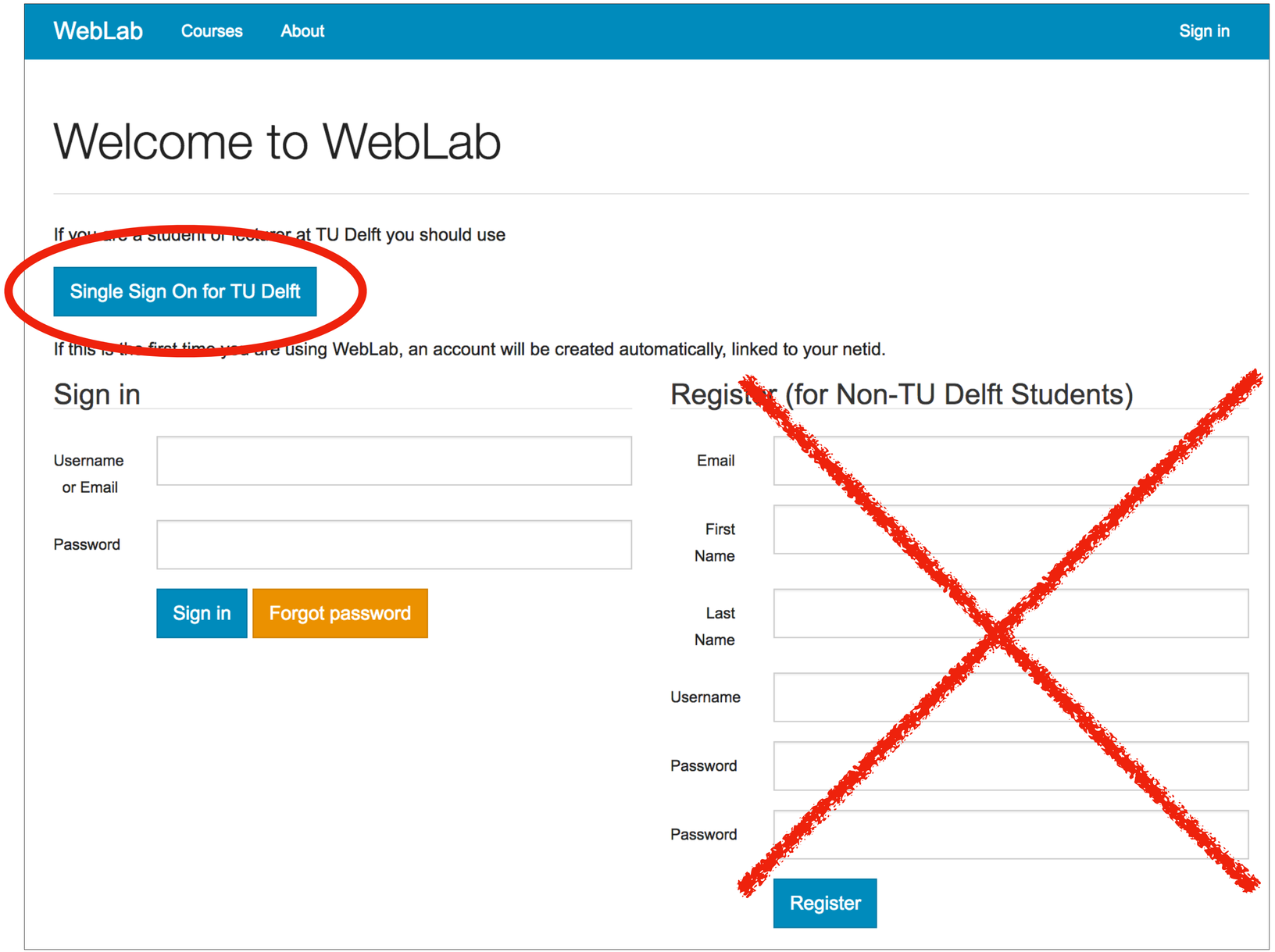
OVERVIEW
- Overview
  - Objectives
  - Submission
  - Grading
  - Early Feedback
- Detailed Instructions
  - Preliminaries
    - Git Repository
    - Importing projects into Eclipse
  - Anatomy of a Test Suite
  - MiniJava Syntax Definition
  - Test Cases
    - Lexical and Context-free Syntax
    - Disambiguation
    - Layout

WebLab for Grade Registration

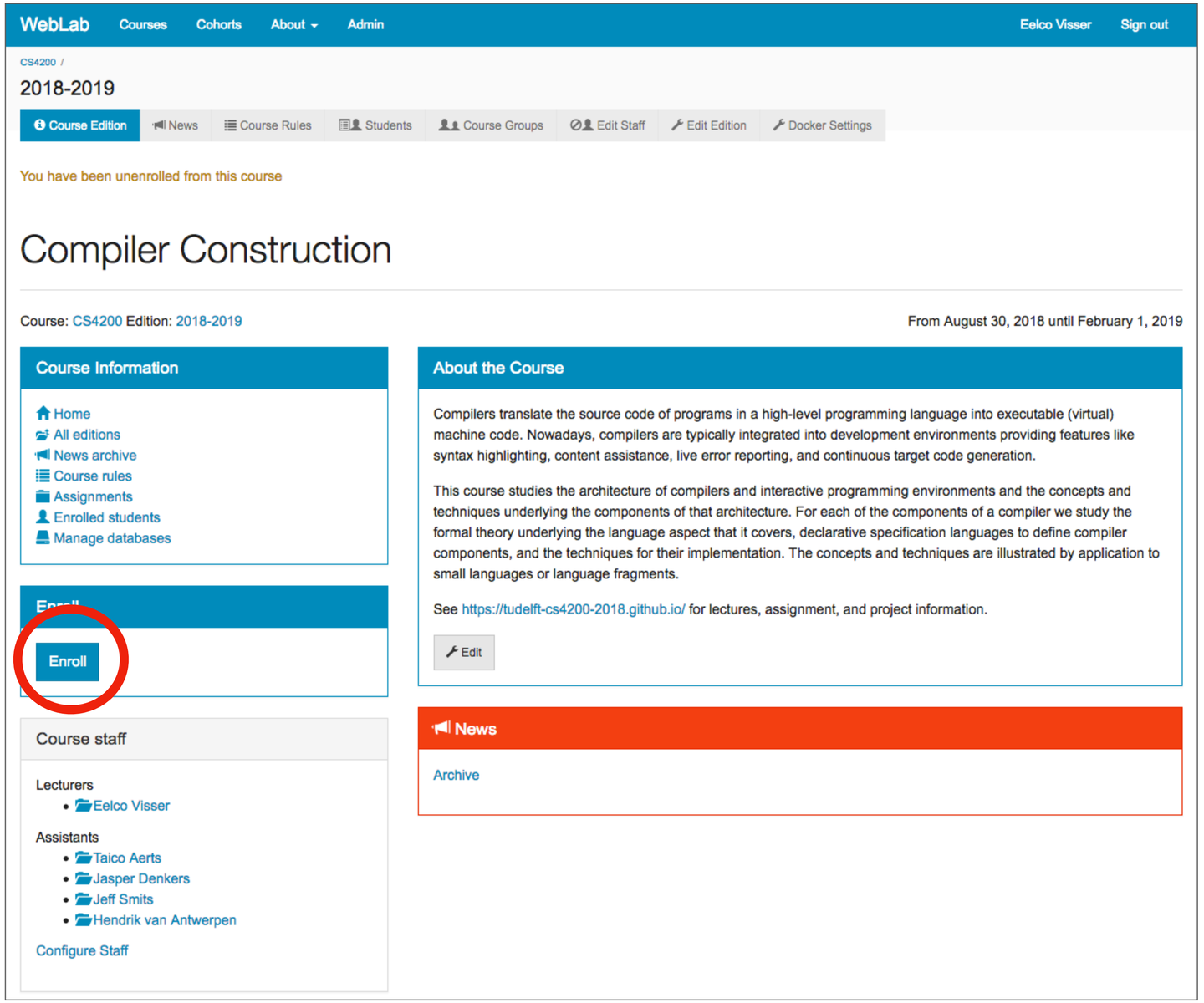


https://weblab.tudelft.nl/cs4200/2018-2019/

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

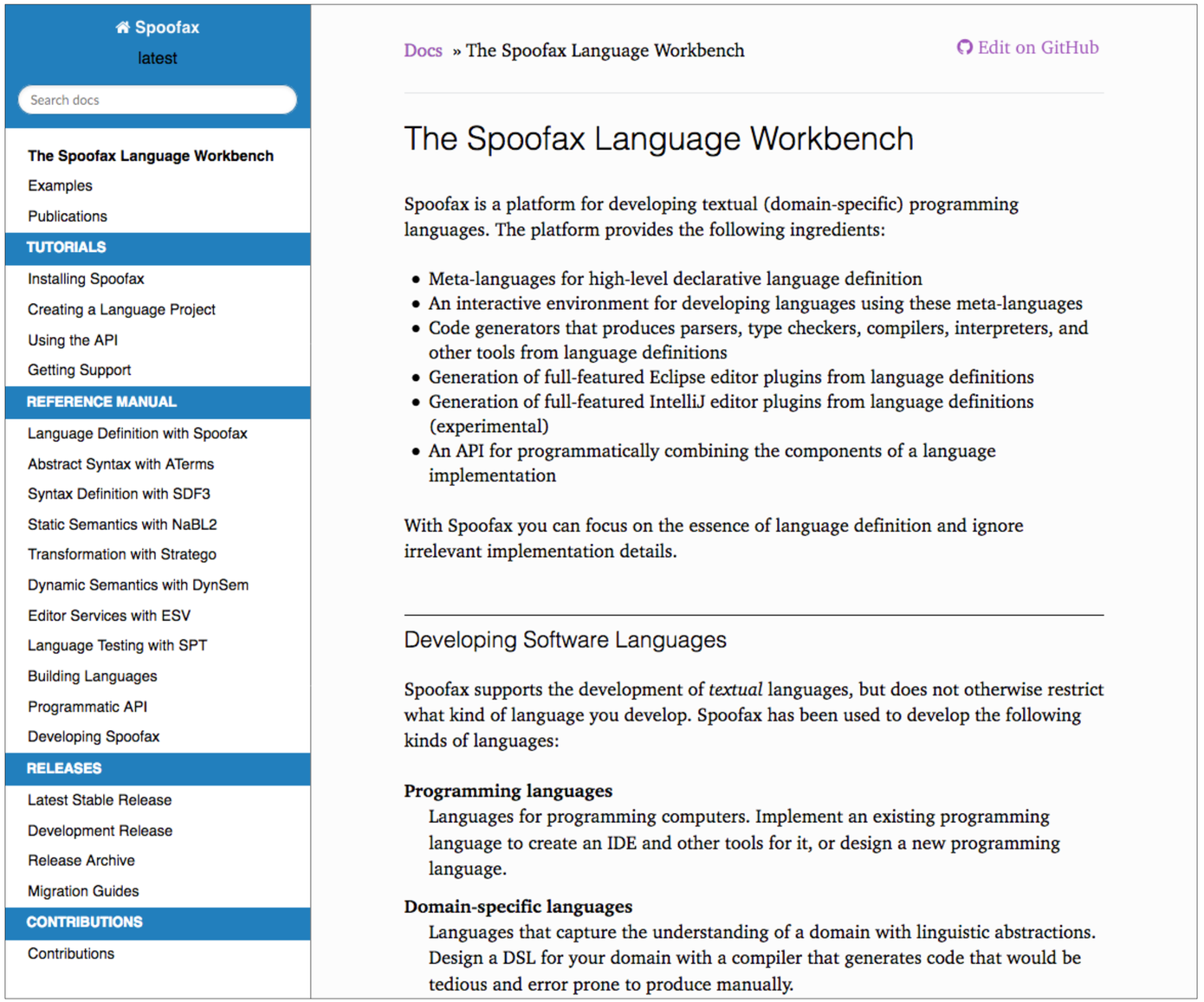


Enroll in WebLab



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

Spoofax Documentation



http://www.metaborg.org

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.

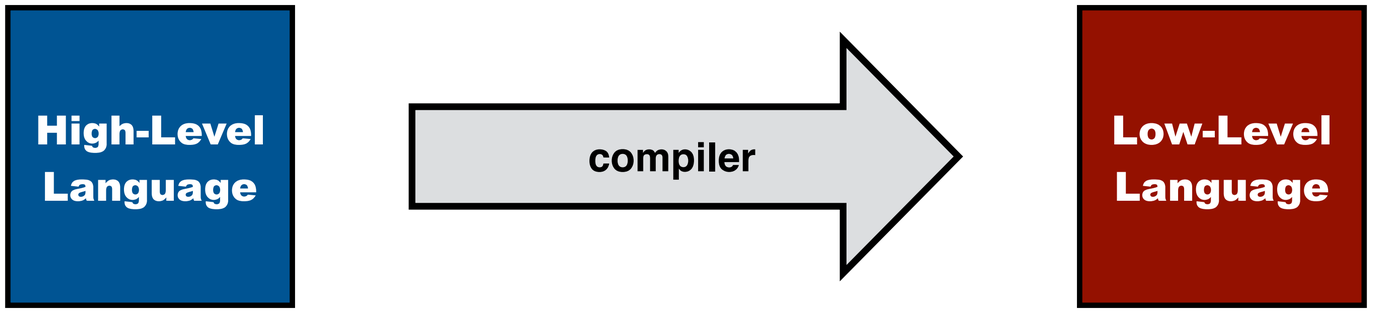
https://en.wiktionary.org/wiki/compile

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.

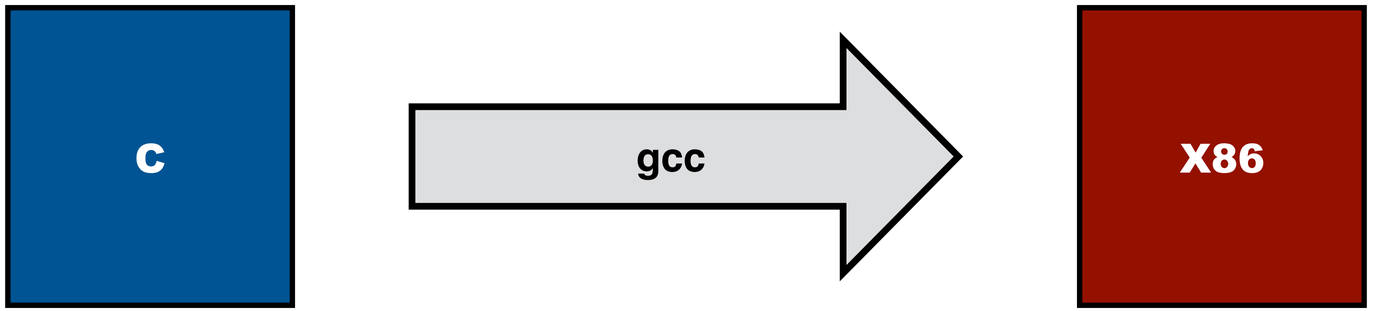
https://en.wikipedia.org/wiki/History_of_compiler_construction

Compiling = Translating



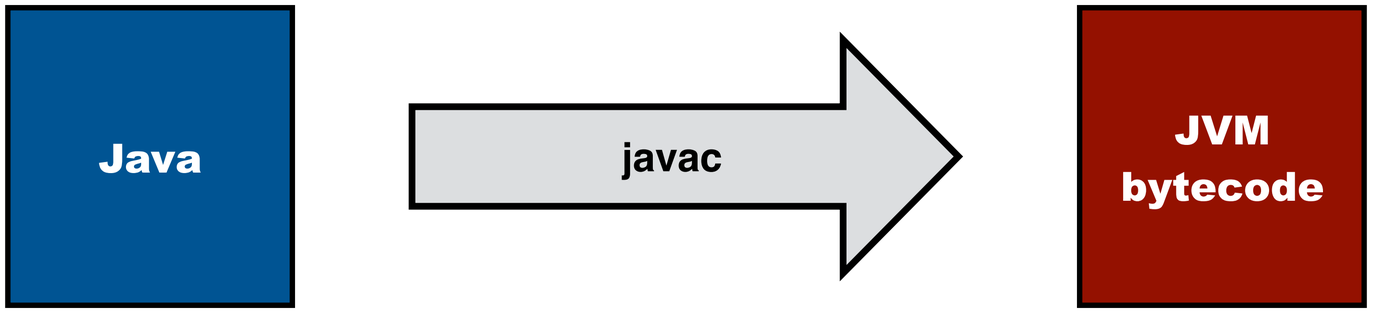
A compiler translates high-level programs to low-level programs

Compiling = Translating



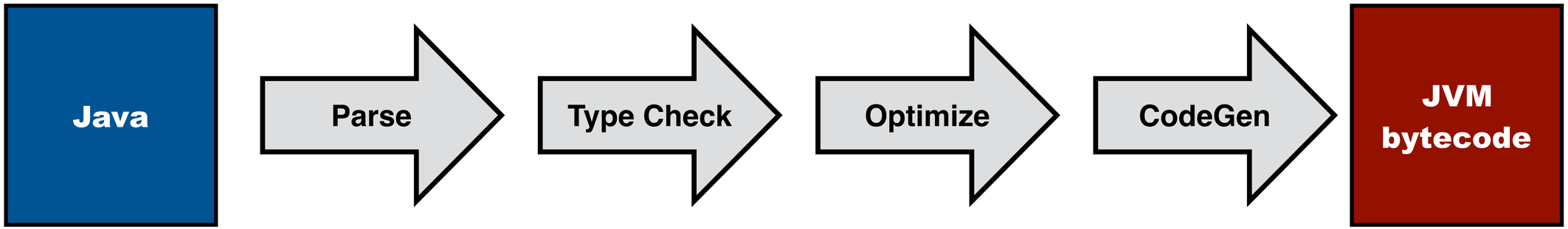
GCC translates C programs to object code for X86 (and other architectures)

Compiling = Translating



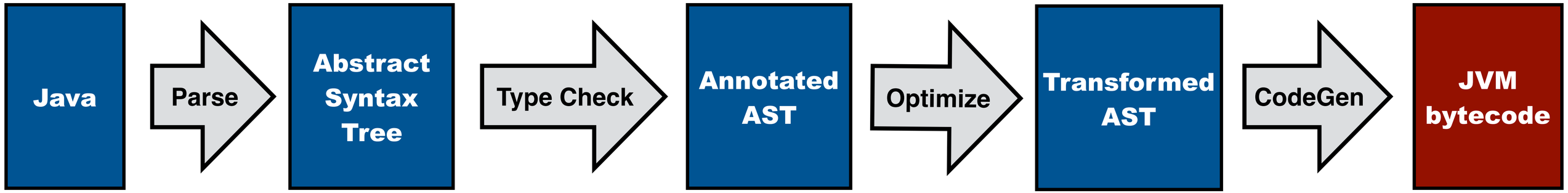
A Java compiler translates Java programs to bytecode instructions for Java Virtual Machine

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
- Produces an abstract syntax tree
- Represents the underlying (syntactic) structure of the program.

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

Optimizer

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

Compiler Components (2)

Code generator

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

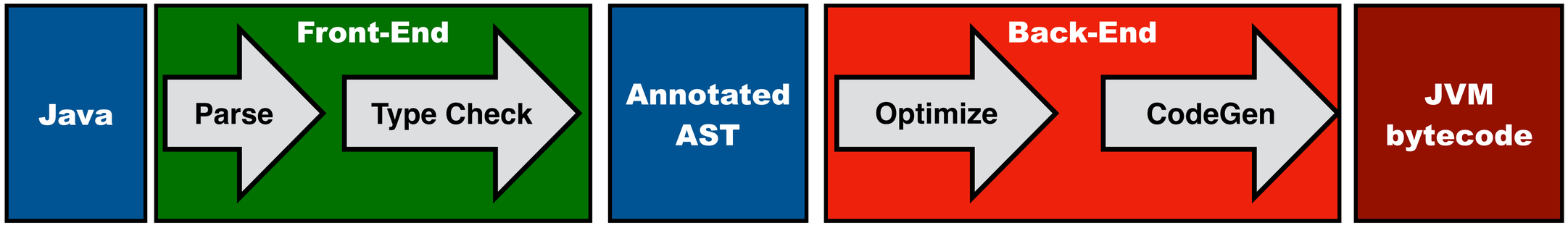
Register allocator

- Assigns physical registers to symbolic registers in the generated instructions

Linker

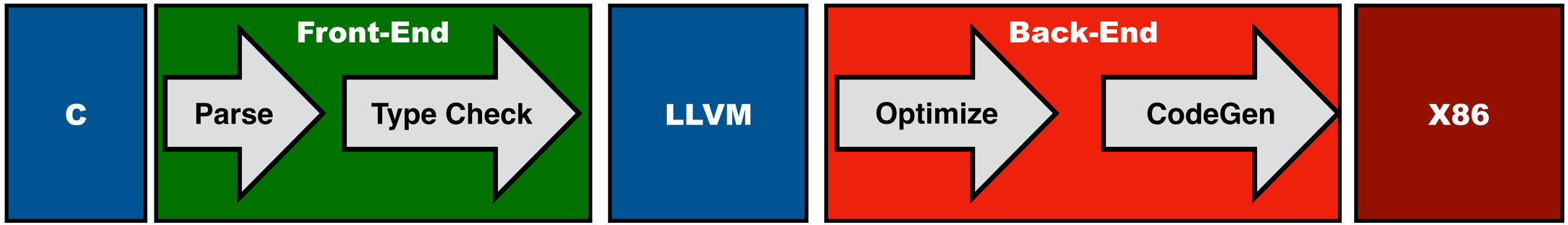
- Most modern languages support some form of modularity in order to divide programs into units
- When also supporting separate compilation, the compiler produces code for each program unit separately
- The linker takes the generated code for the program units and combines it into an executable program

Compiler = Front-end + Back-End



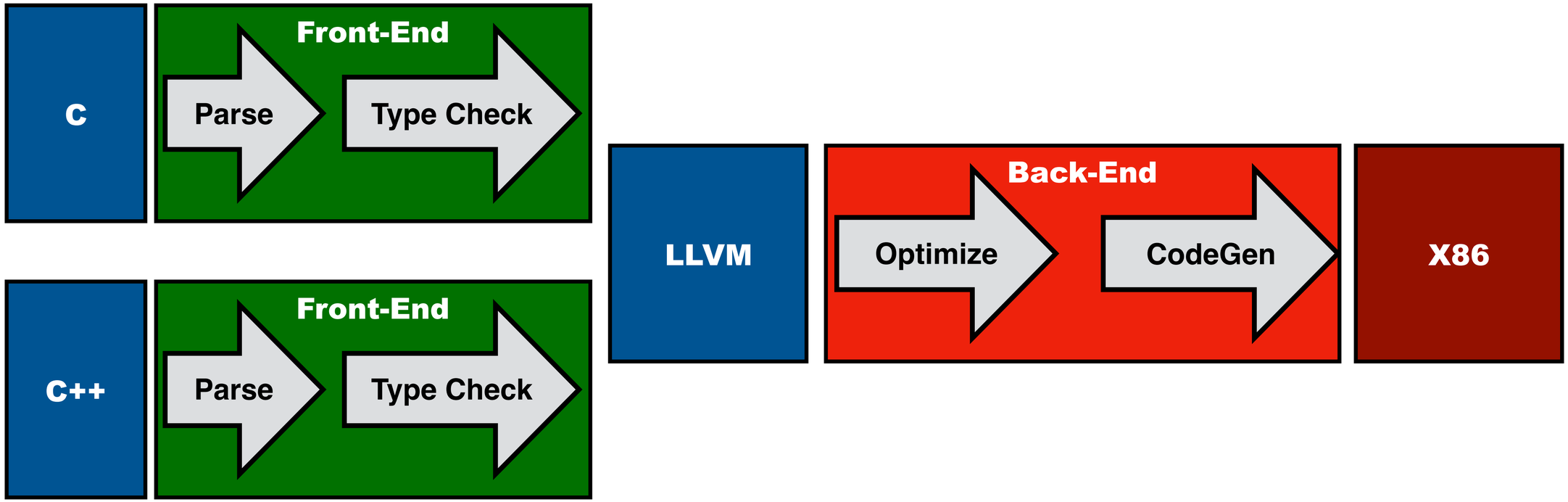
A compiler can typically be divided in a front-end (analysis) and a back-end (synthesis)

Compiler = Front-end + Back-End



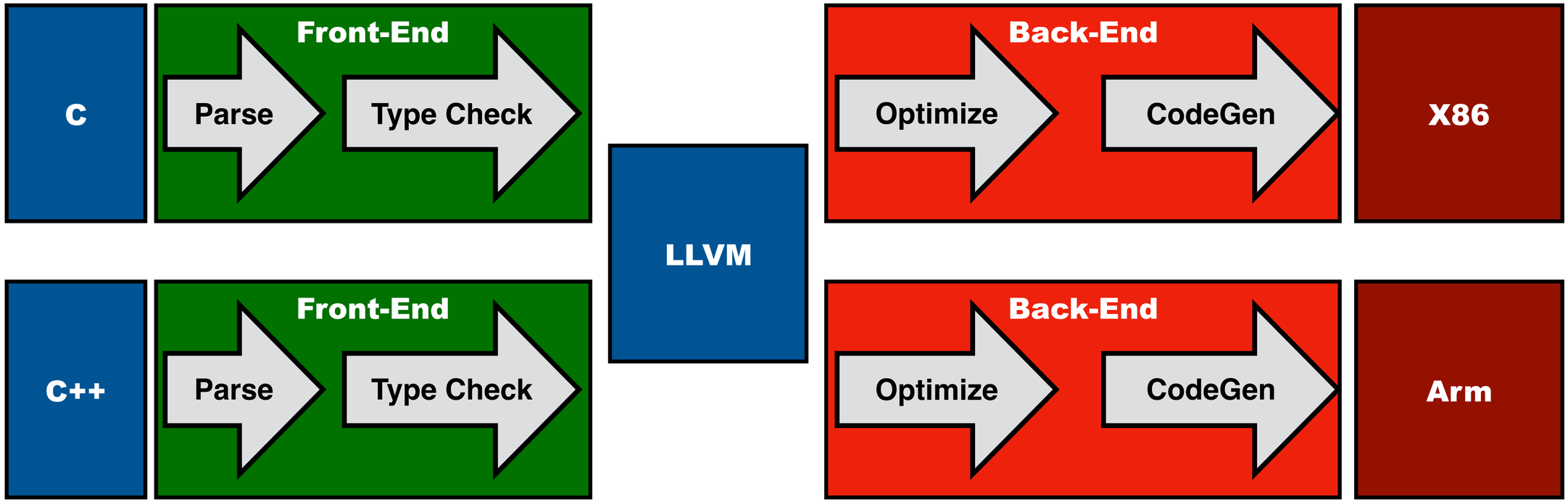
A compiler can typically be divided in a front-end (analysis) and a back-end (synthesis)

Repurposing Back-End

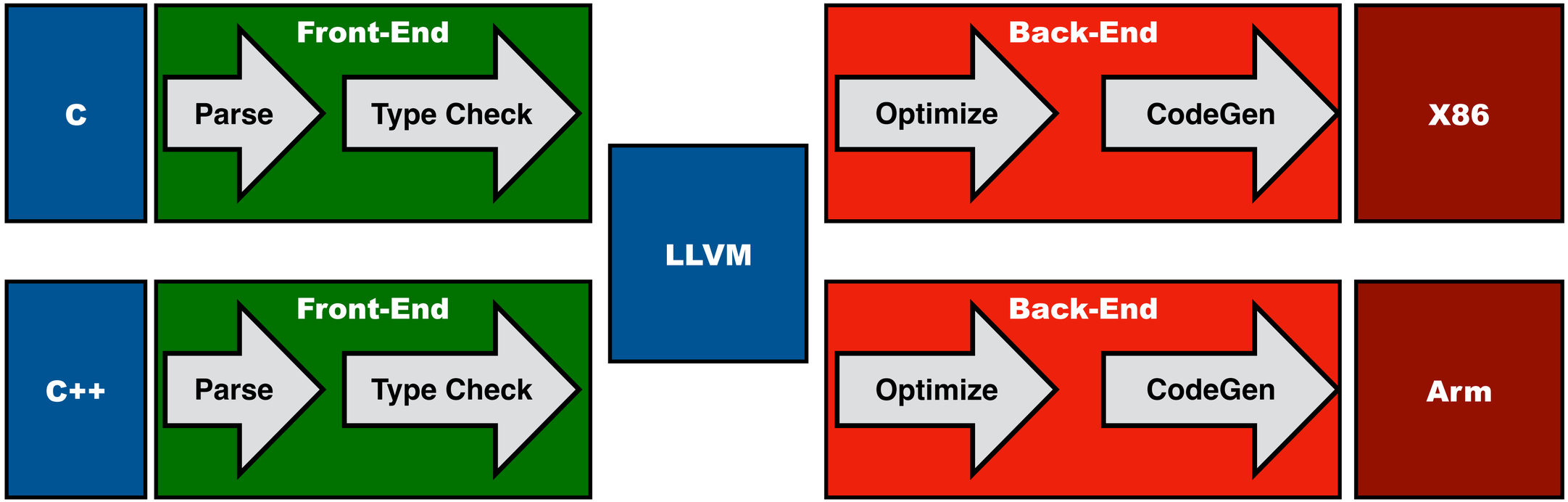


Repurposing: reuse a back-end for a different source language

Retargeting Compiler

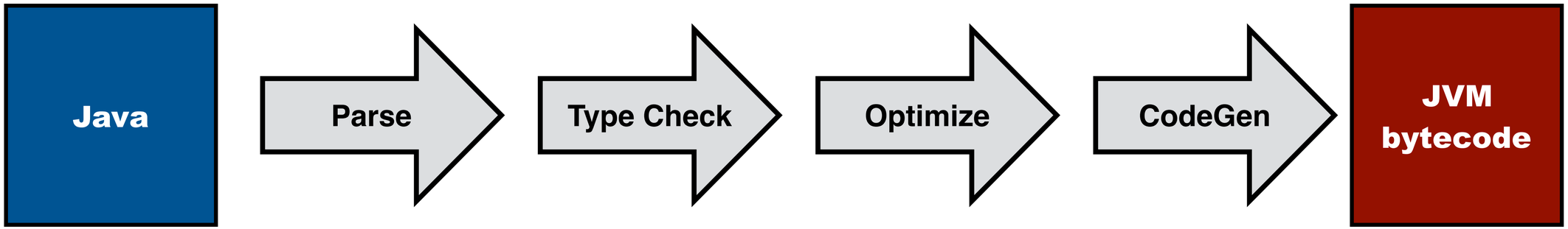


Retargeting: compile to different hardware architecture


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

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

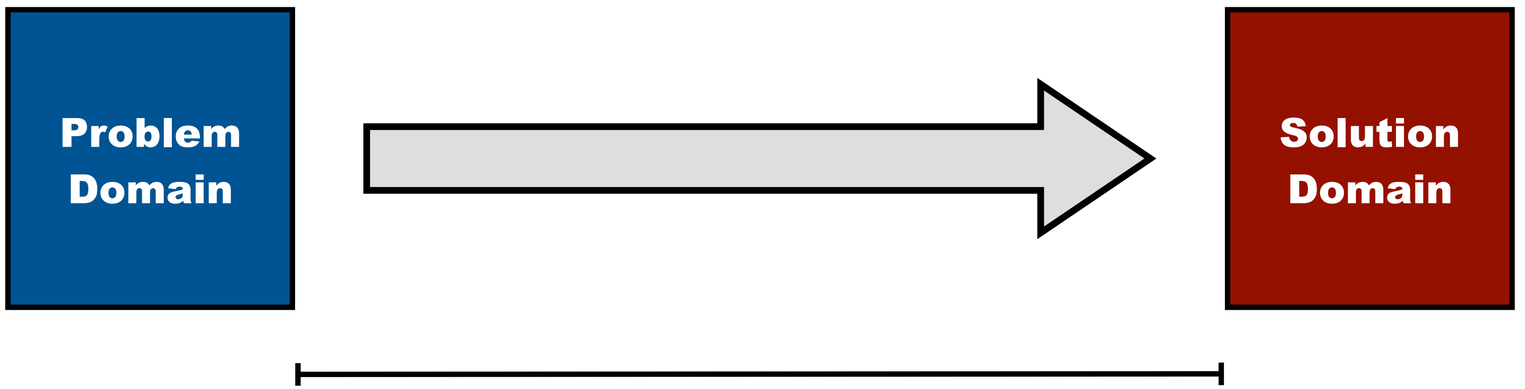
Why Compilers?

Programming = Instructing Computer

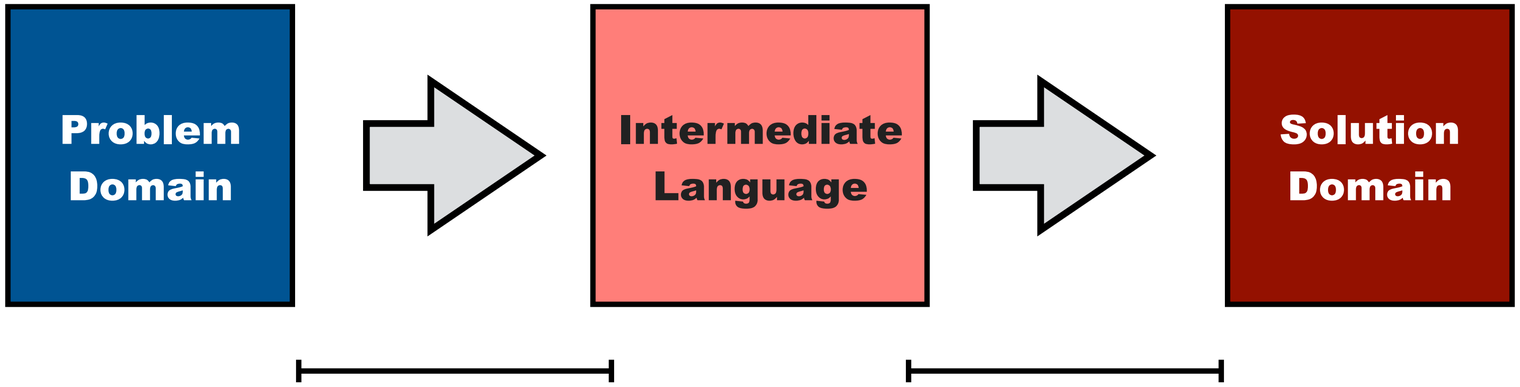
- fetch data from memory
- store data in register
- perform basic operation on data in register
- fetch instruction from memory
- update the program counter
- etc.

"*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

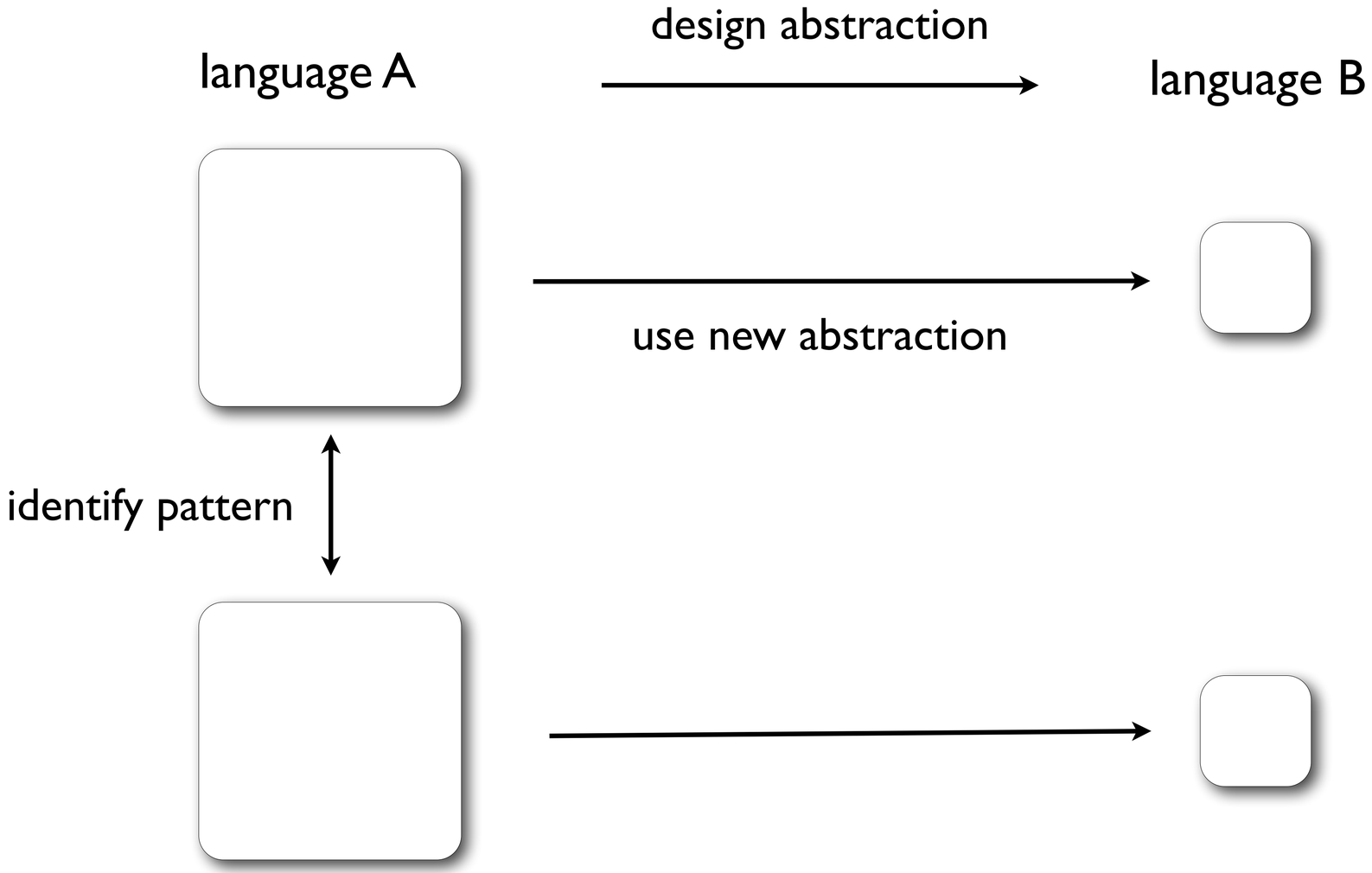
From Malloc to Garbage Collection

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

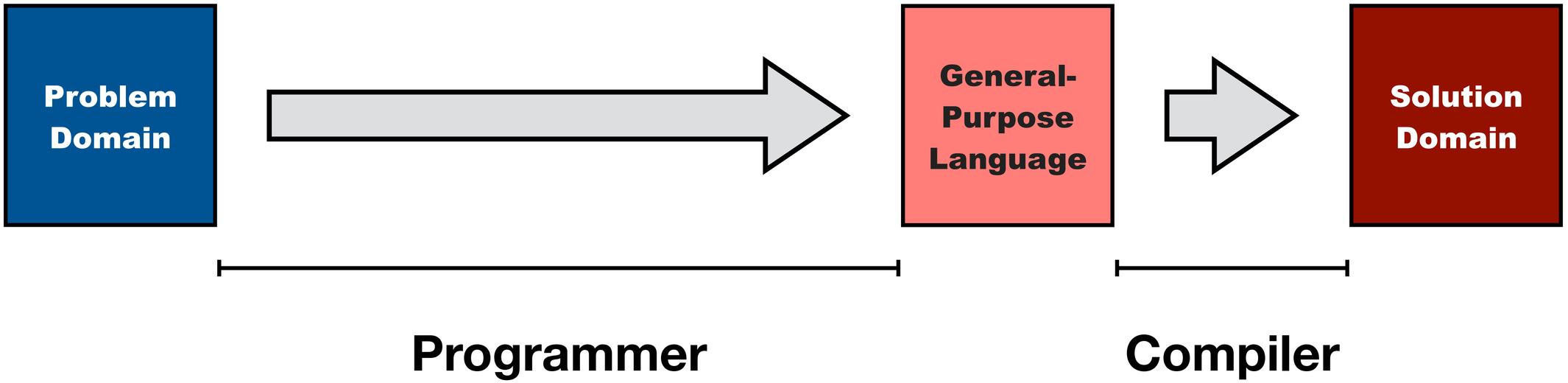
http://en.wikipedia.org/wiki/Malloc

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

Linguistic Abstraction



Compiler Automates Work of Programmer

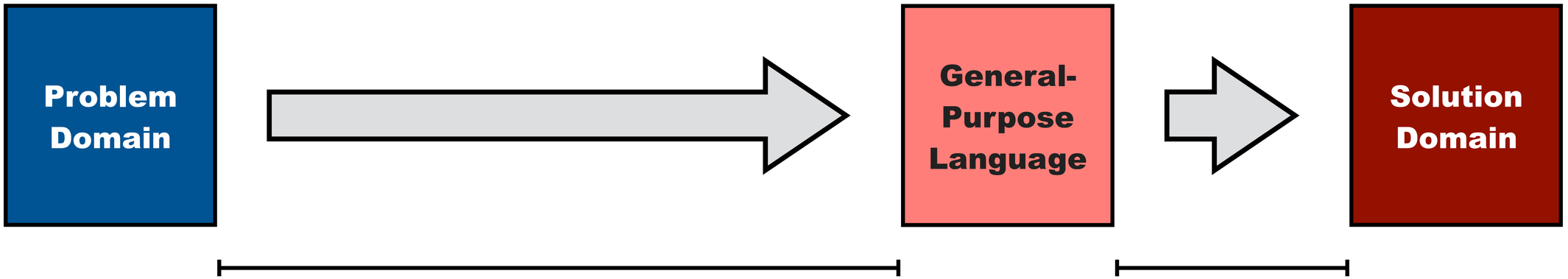


Compilers for modern high-level languages

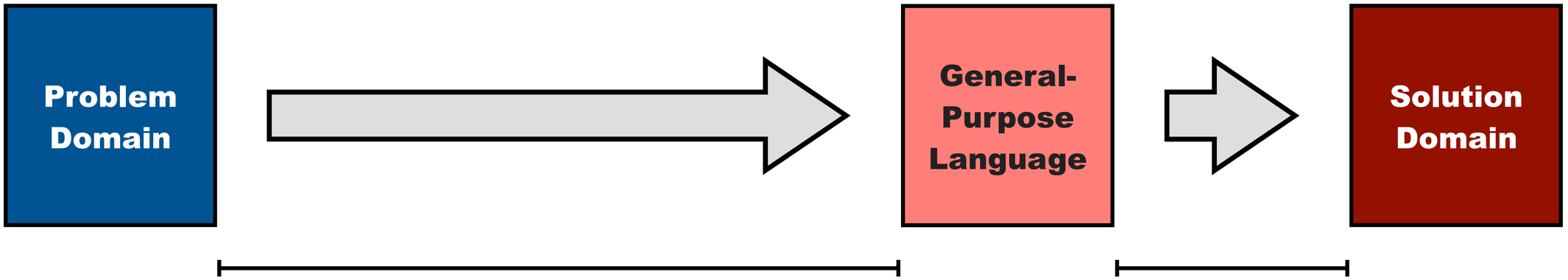
- Reduce the gap between problem domain and program
- Support programming in terms of computational concepts instead of machine concepts
- Abstract from hardware architecture (portability)
- Protect against a range of common programming errors

Domain-Specific Languages

Domains of Computation

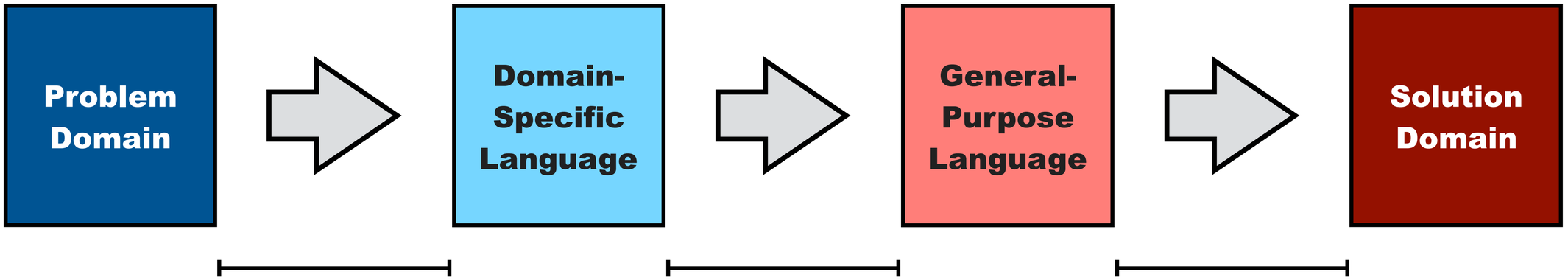


- Systems programming
- Embedded software
- Web programming
- Enterprise software
- Database programming
- Distributed programming
- Data analytics
- ...



"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

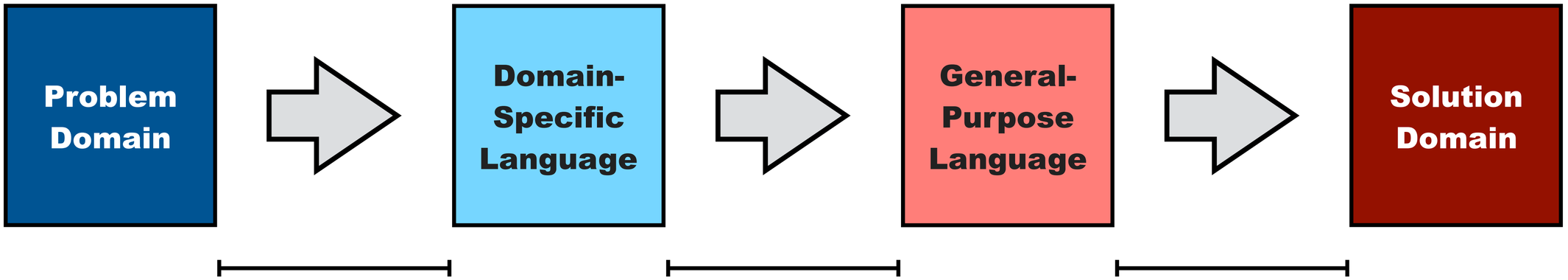
- What is the semantics of such definitions?
- How can we verify the correctness / consistency of language definitions?

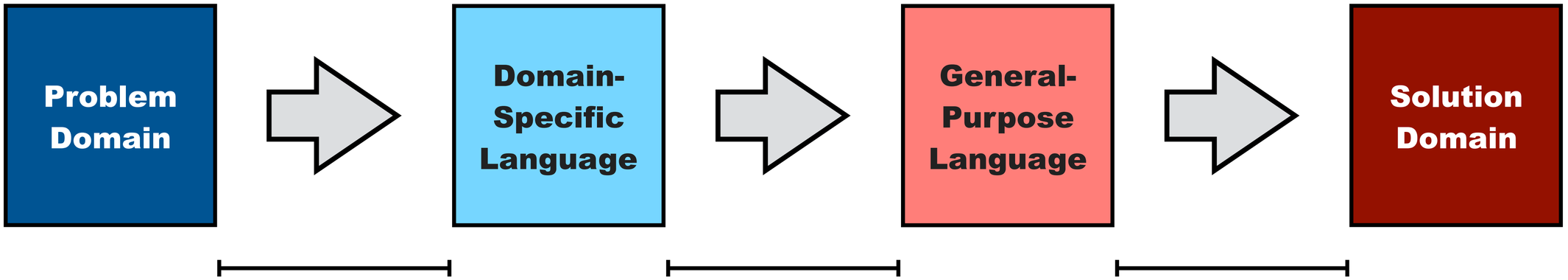
Implementation

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

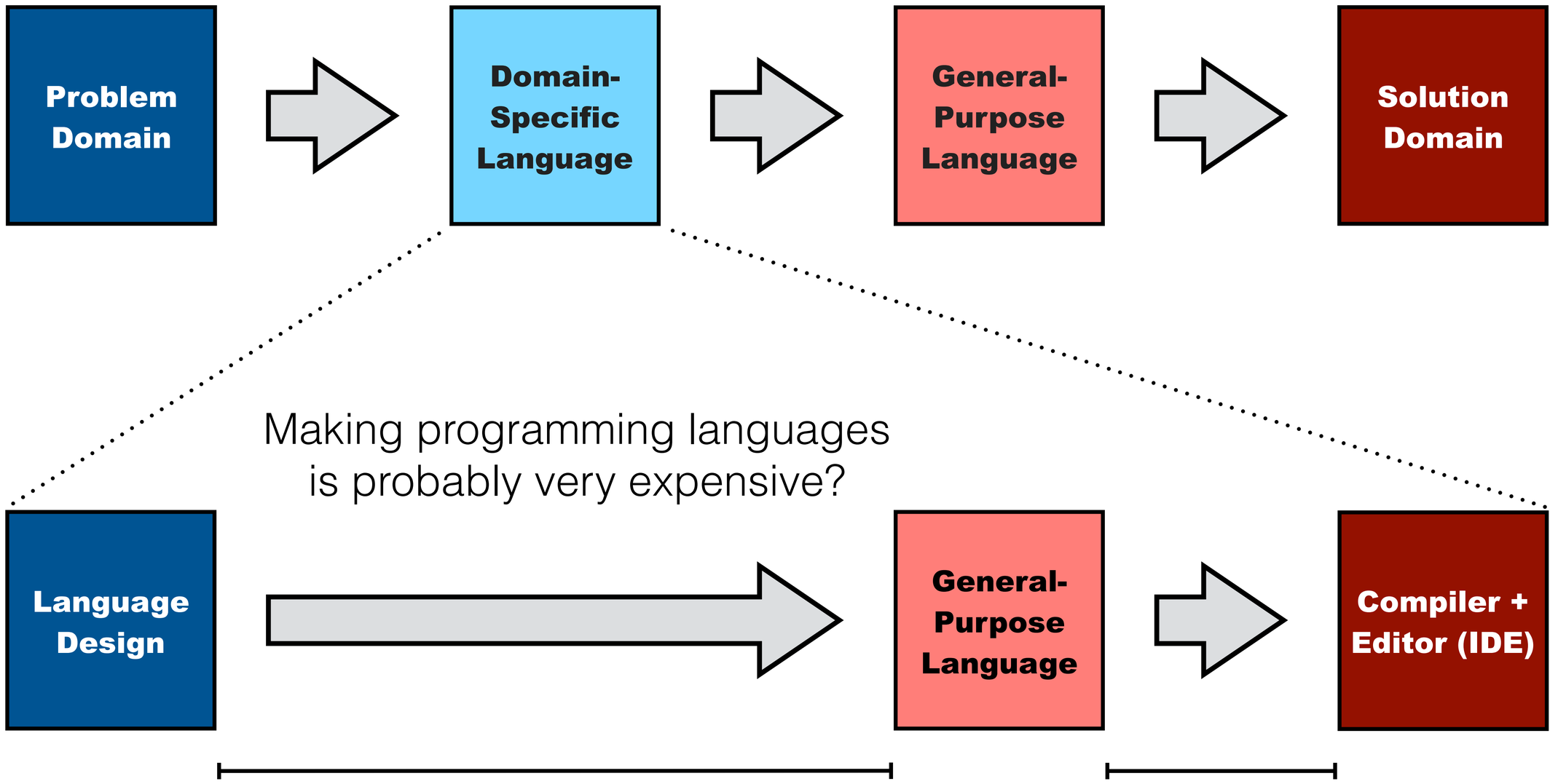
Evaluation

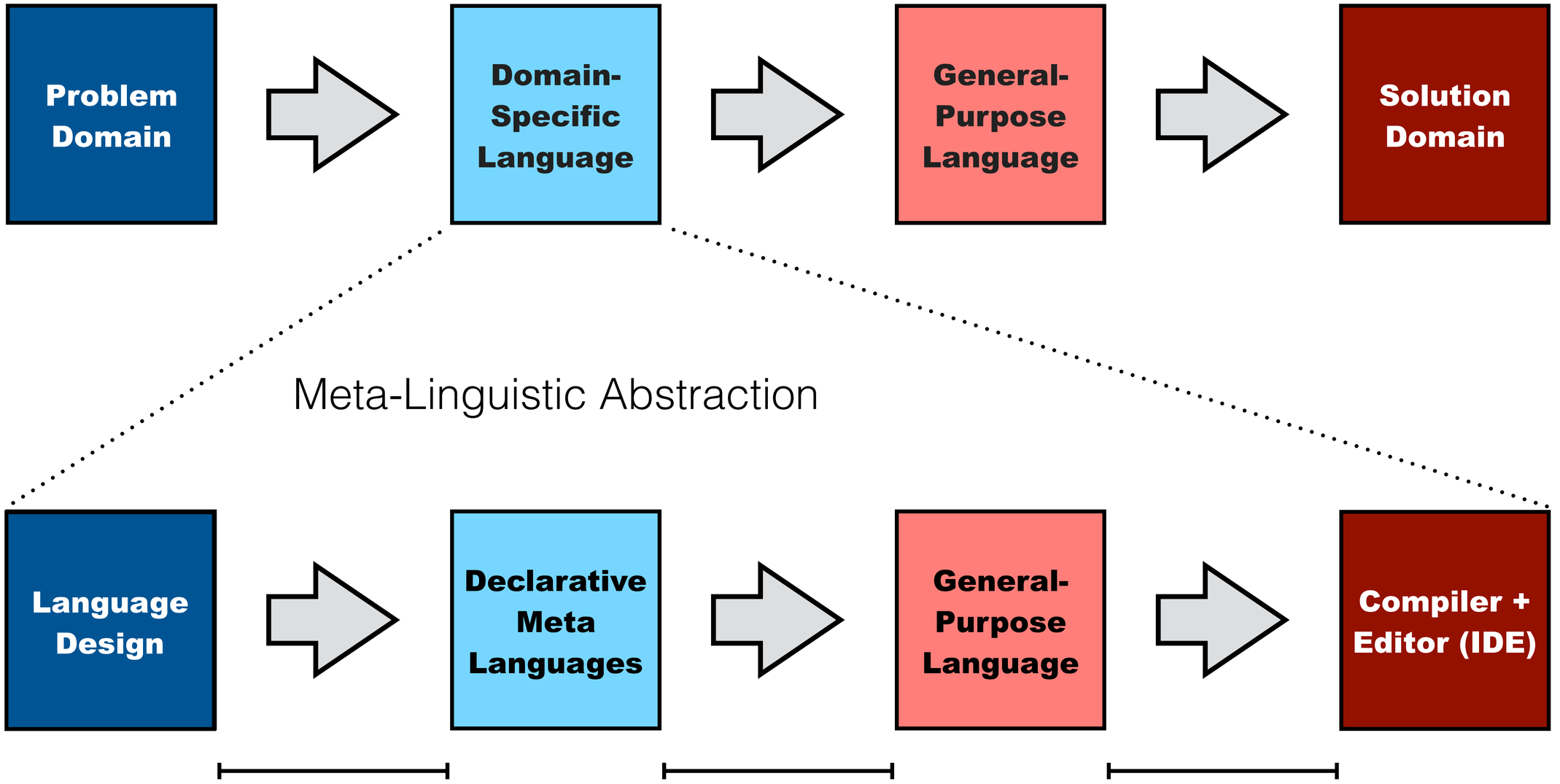
- Apply to new and existing languages to determine adequacy


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

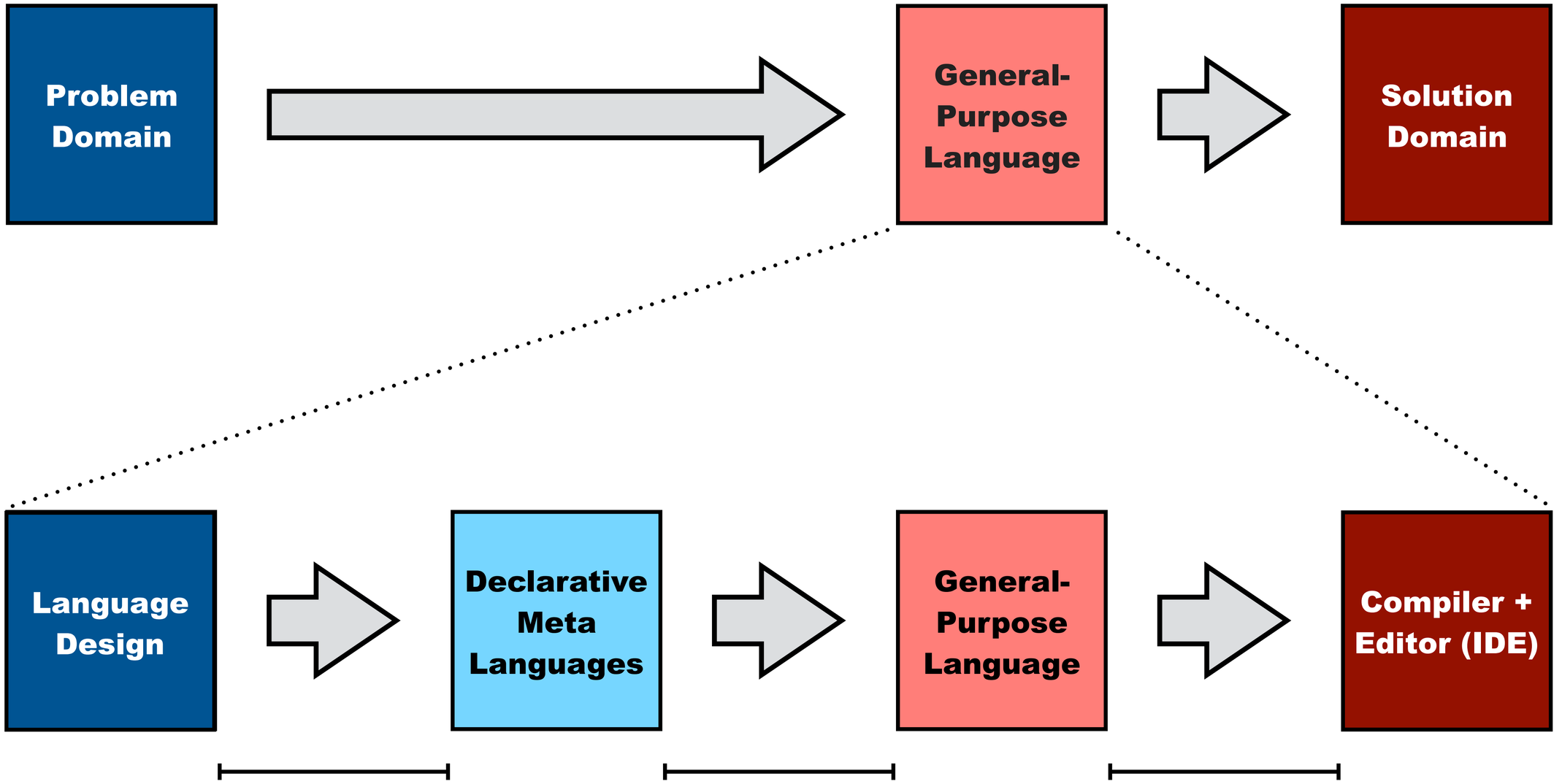


Making programming languages
is probably very expensive?


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)

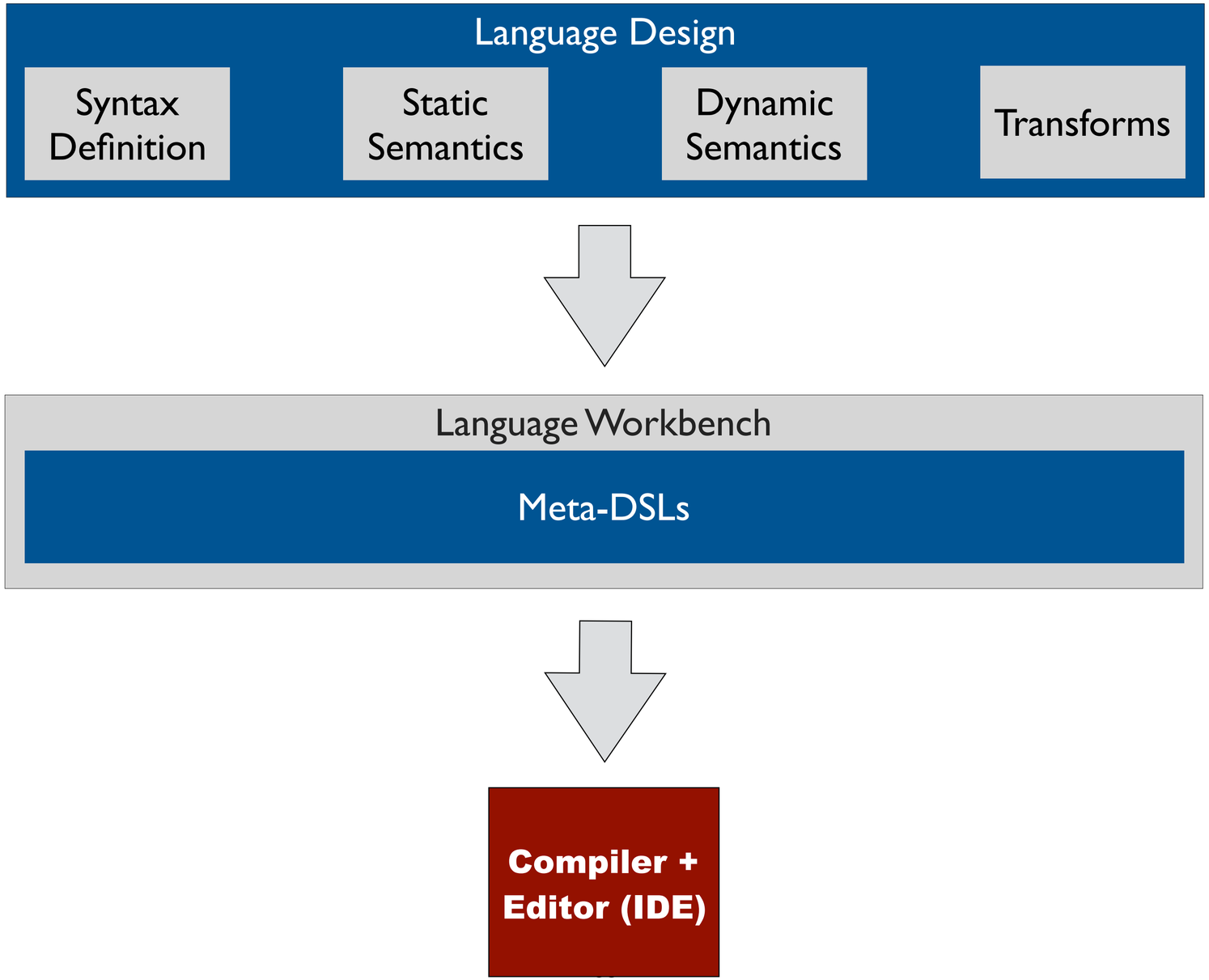


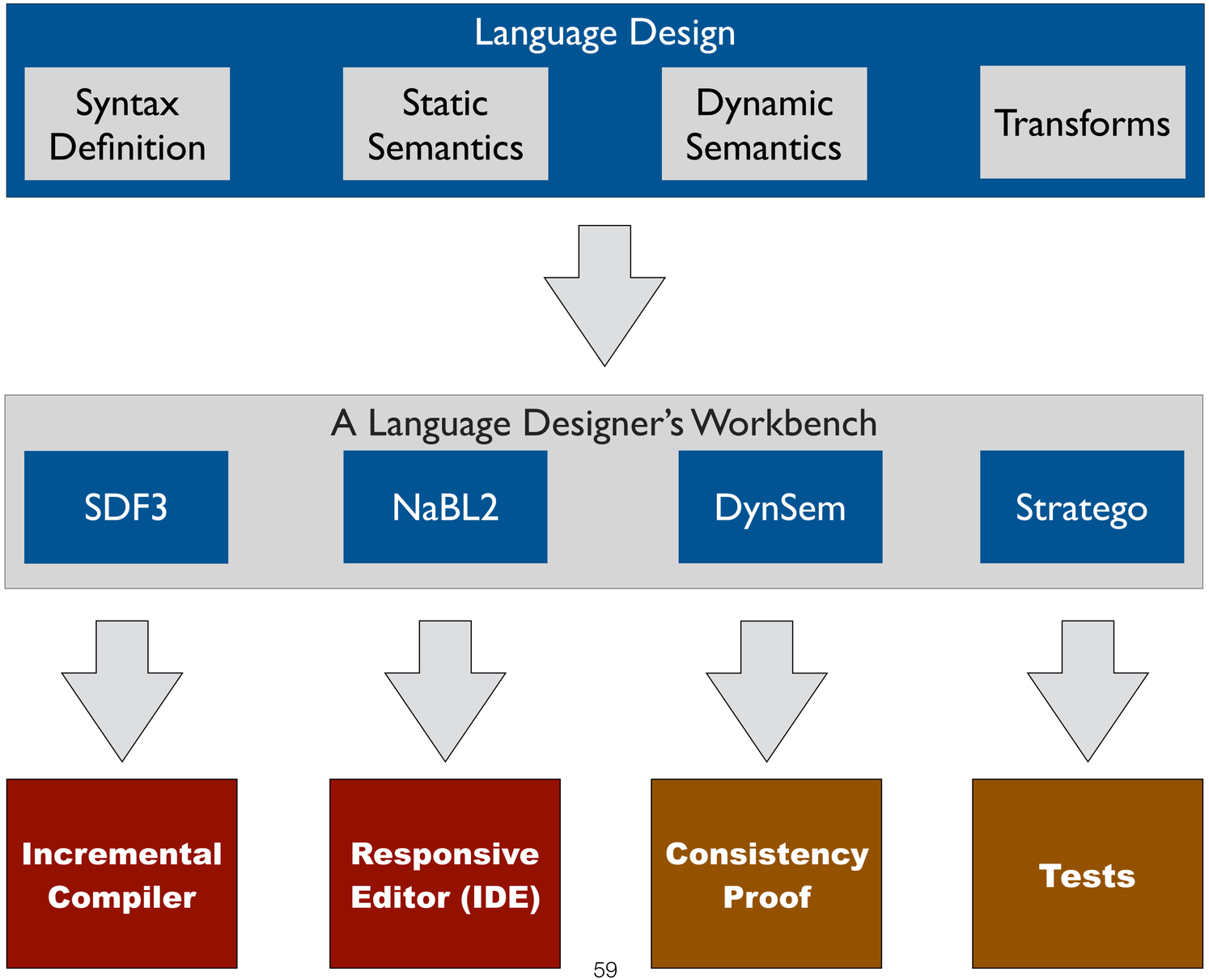
Applying compiler construction to the domain of compiler construction



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

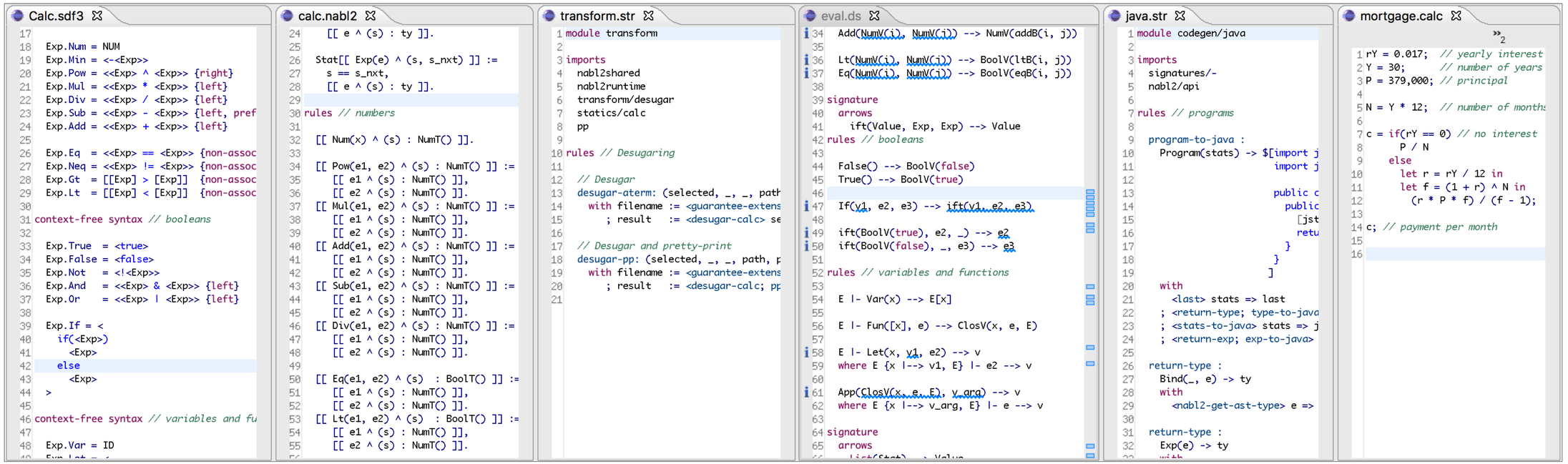
A Language Designer's Workbench

- 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

TU Delft

Language Definition in Spoofax Language Workbench



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) ]] :
//     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
```

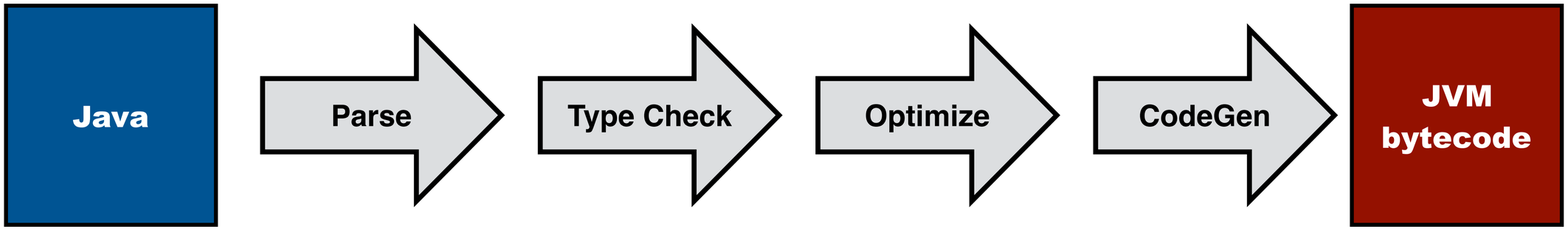
queens.tig

```
/* 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

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
- Basic Parsing
- Syntactic Editor Services
- Transformation
- Static Semantics & Name Resolution
- Type Constraints
- Constraint Resolution
- More Constraint Resolution

Q2

- Data-Flow Analysis
- Virtual Machines & Code Generation
- Optimization
- Garbage Collection
- Dynamic Semantics & Interpreters
- Applications
- Reserve
- Overview

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

Lectures: Tuesday, 10:45 in Lecture Hall CT-CZ D

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

https://doi.org/10.1145/1932682.1869497

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].

OOPSLA/SPLASH'10, October 17–21, 2010, Reno/Tahoe, Nevada, USA.

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 engineers 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.*

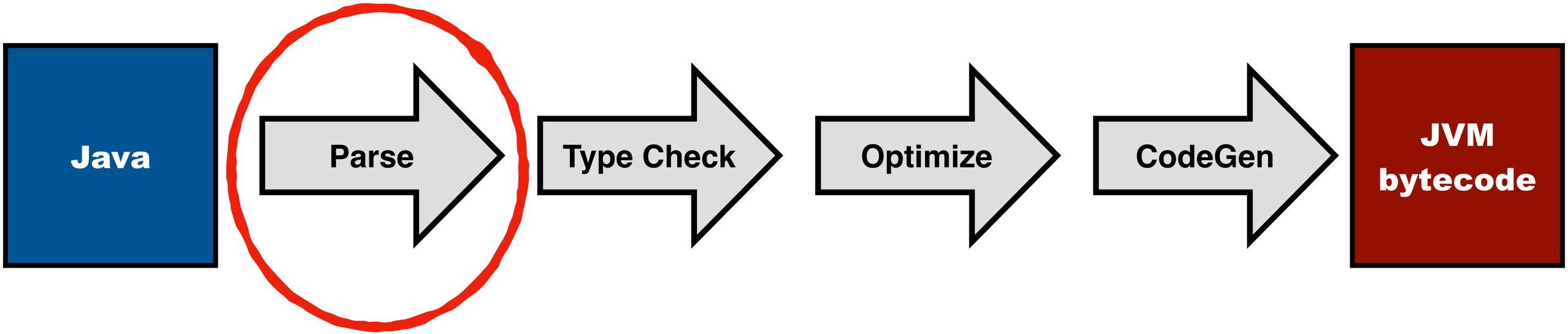
So when the language engineers saw that restricted parsers were good for efficiency, and that they were a delight to the benchmarks, they made software to make efficient parsers and began to make efficient parsers by making *parser definitions*. Those days, the language engineers went out from the garden 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-

1, 2010, Reno/Tahoe, Nevada, USA.
1-4503-0236-4/10/10…$10.00

Next Lecture

Next: Syntax Definition

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



Specification of syntax definition from which we can derive parsers