What is a Compiler?

Eelco Visser

TU Delft

CS4200 | Compiler Construction | September 2, 2021

This Lecture

Introduction

- What is a compiler?
- Why compilers?
- Meta-languages for language definition
- Language workbenches
- Project language

Course organization

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.

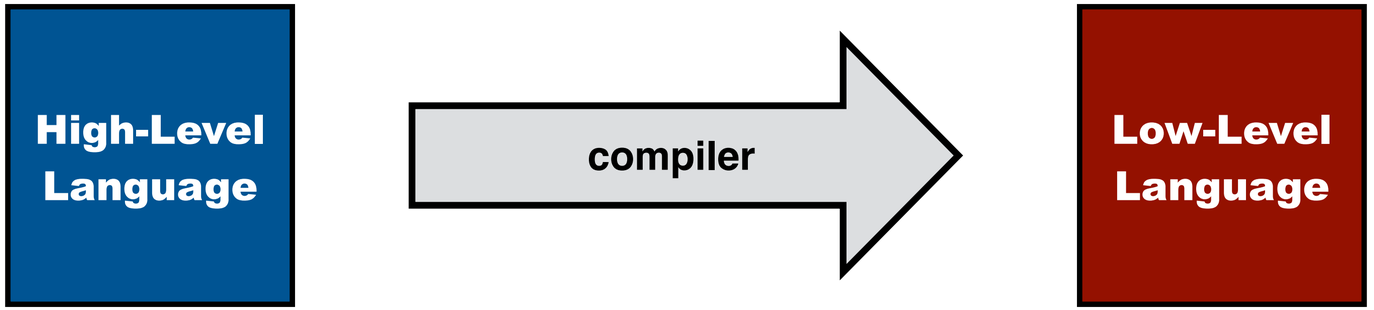
https://en.wiktionary.org/wiki/compile

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.

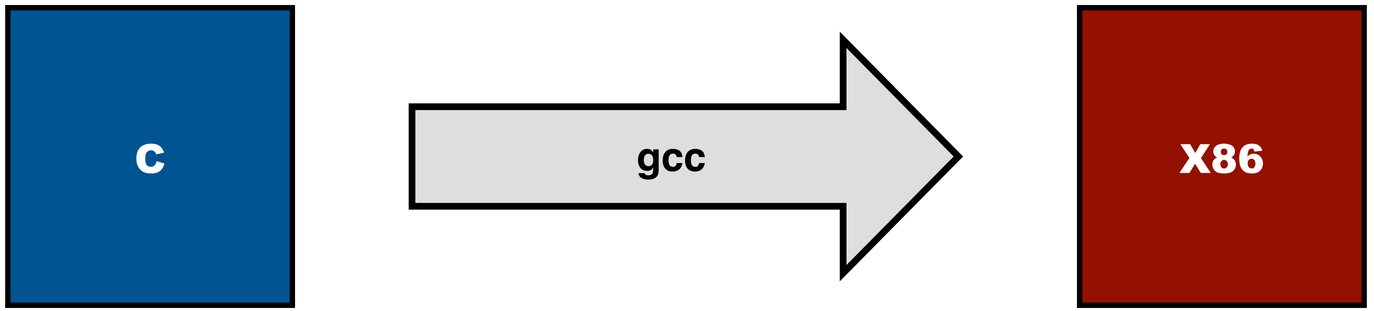
https://en.wikipedia.org/wiki/History_of_compiler_construction

Compiling = Translating



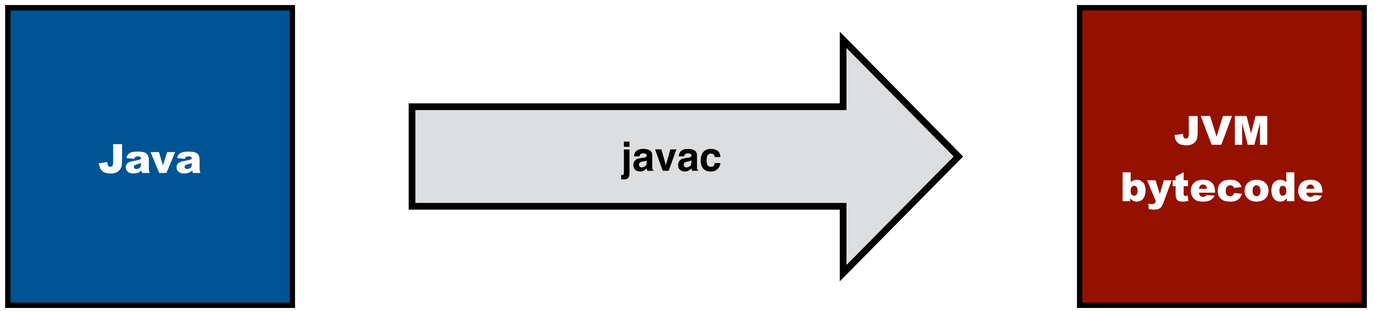
A compiler translates high-level programs to low-level programs

Compiling = Translating



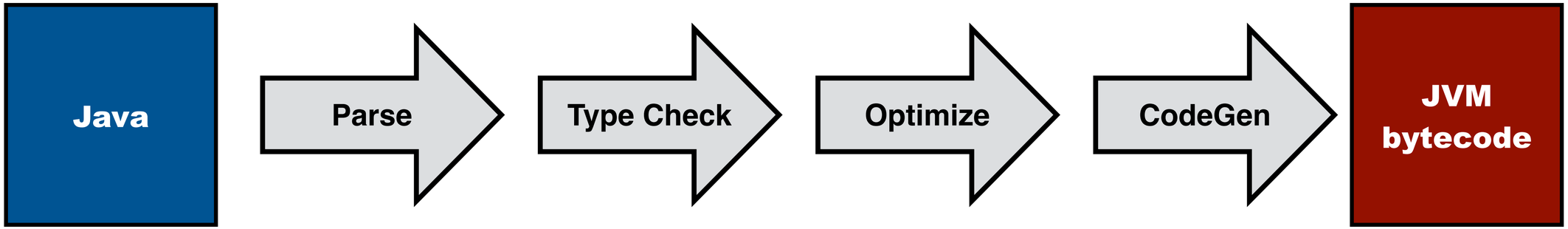
GCC translates C programs to object code for X86 (and other architectures)

Compiling = Translating



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

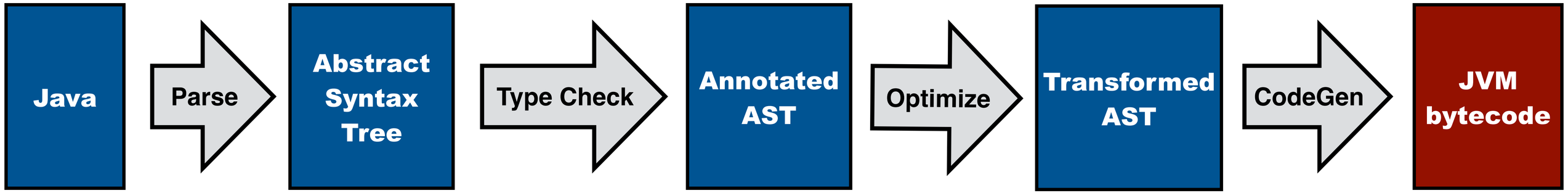
Architecture: Multi-Pass Compiler



A modern compiler typically consists of sequence of stages or passes

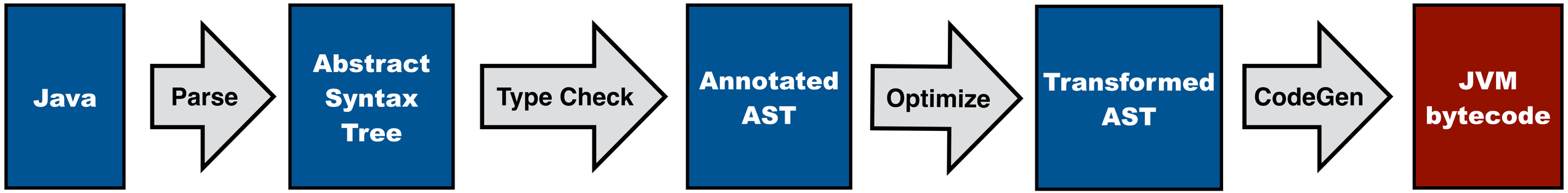
Intermediate Representations

component descriptions on next slides



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

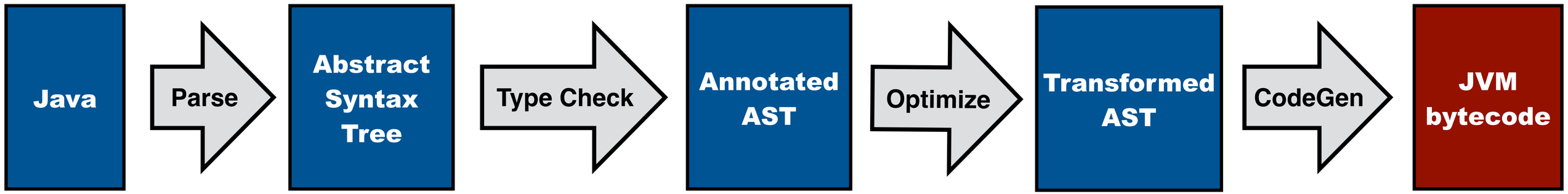
Compiler Components



Parser

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

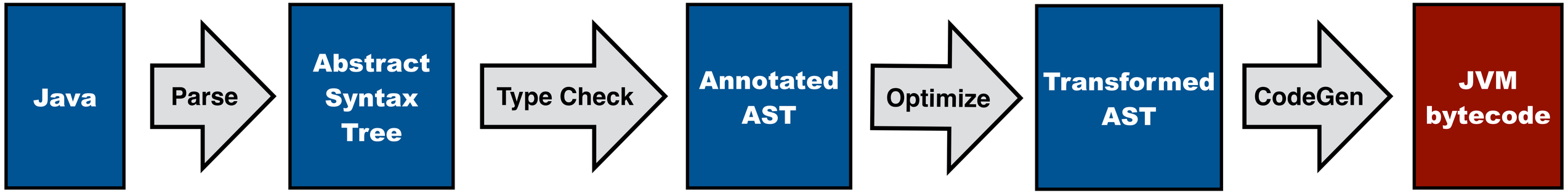
Compiler Components



Type checker

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

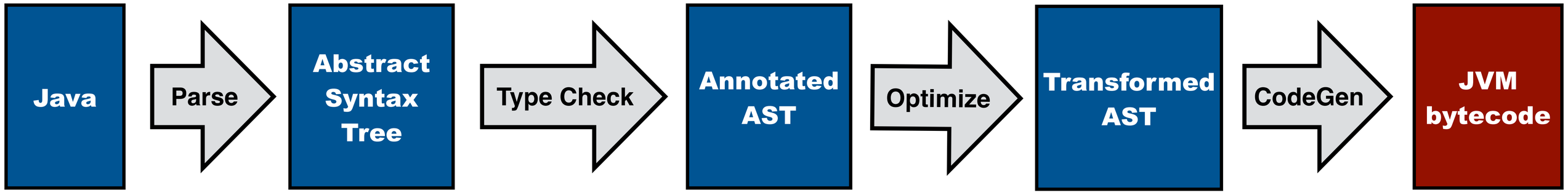
Compiler Components



Optimizer

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

Compiler Components



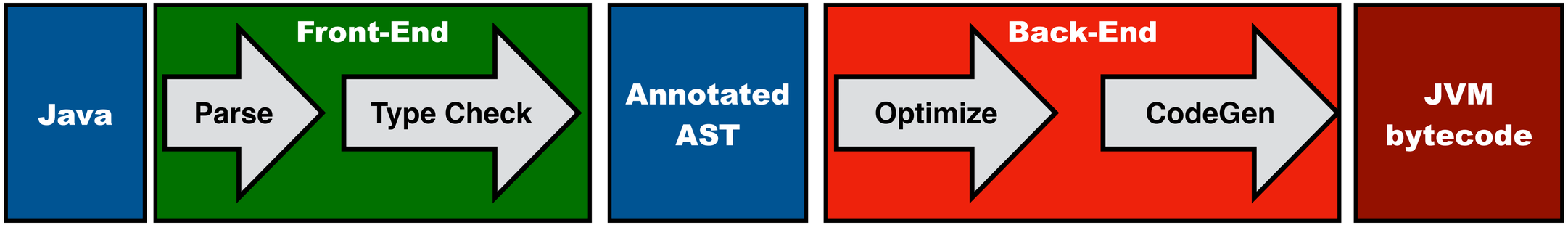
Code generator

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

Register allocator

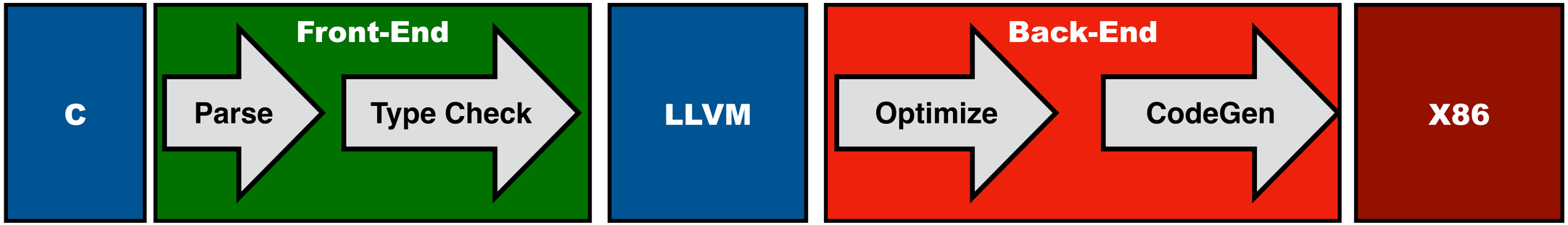
- Assigns physical registers to symbolic registers in the generated instructions

Compiler = Front-end + Back-End



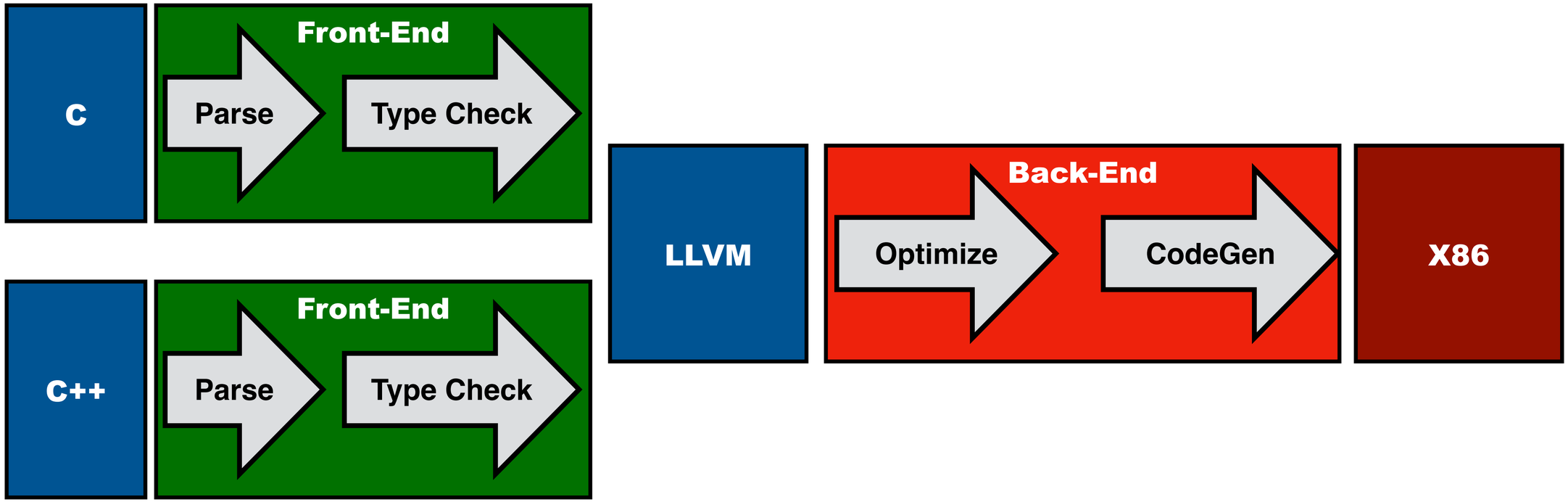
A compiler can typically be divided in a front-end (analysis) and a back-end (synthesis)

Compiler = Front-end + Back-End



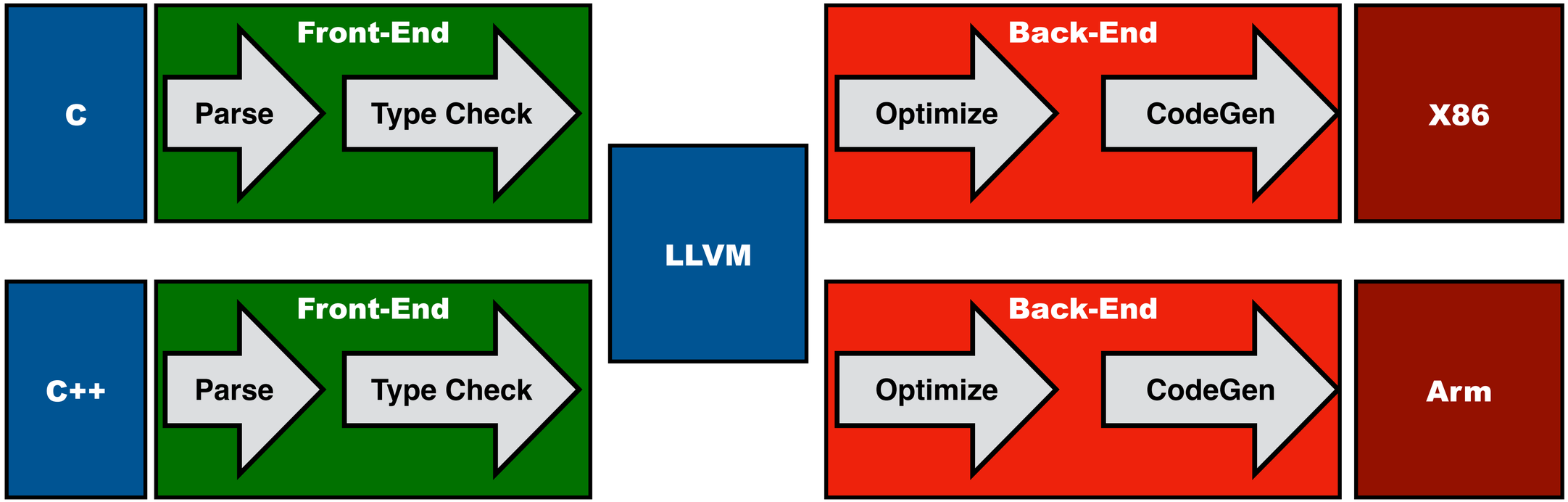
A compiler can typically be divided in a front-end (analysis) and a back-end (synthesis)

Repurposing Back-End



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

Retargeting Compiler



Retargeting: compile to different hardware architecture

Types of Compilers (1)

Compiler

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

Bytecode compiler

– generates code for a virtual machine

Just-in-time compiler

– defers (some aspects of) compilation to run time

Source-to-source compiler (transpiler)

– translate between high-level languages

Cross-compiler

– runs on different architecture than target architecture

Types of Compilers (2)

Interpreter

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

Hardware compiler

- generate configuration for FPGA or integrated circuit

De-compiler

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

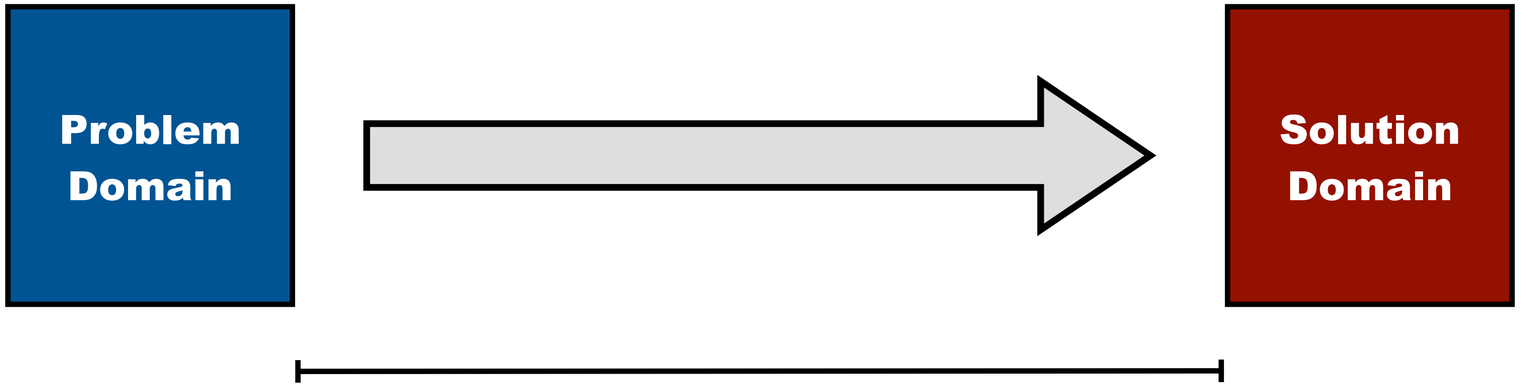
Why Compilers?

Programming = Instructing Computer

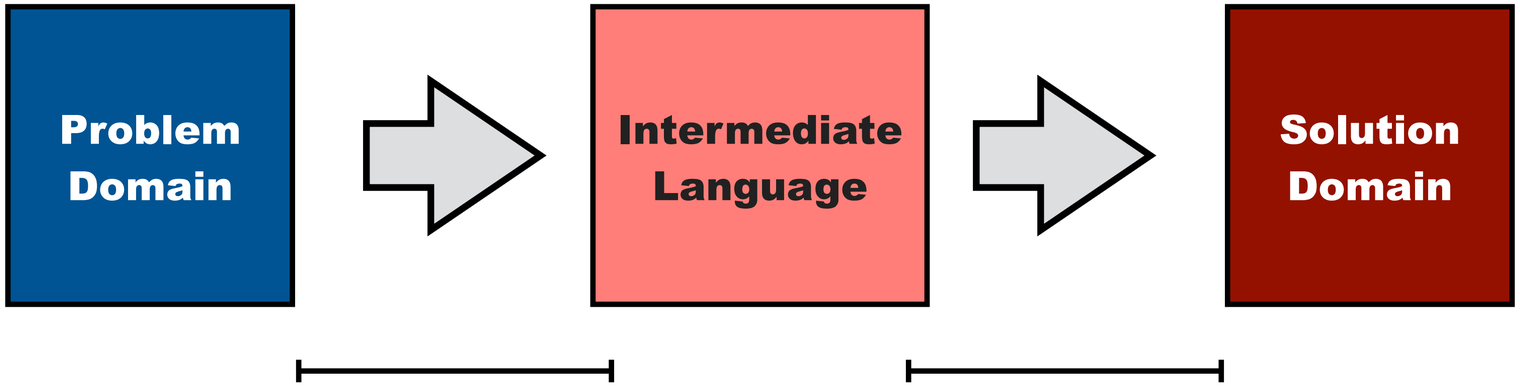
- fetch data from memory
- store data in register
- perform basic operation on data in register
- fetch instruction from memory
- update the program counter
- etc.

"*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

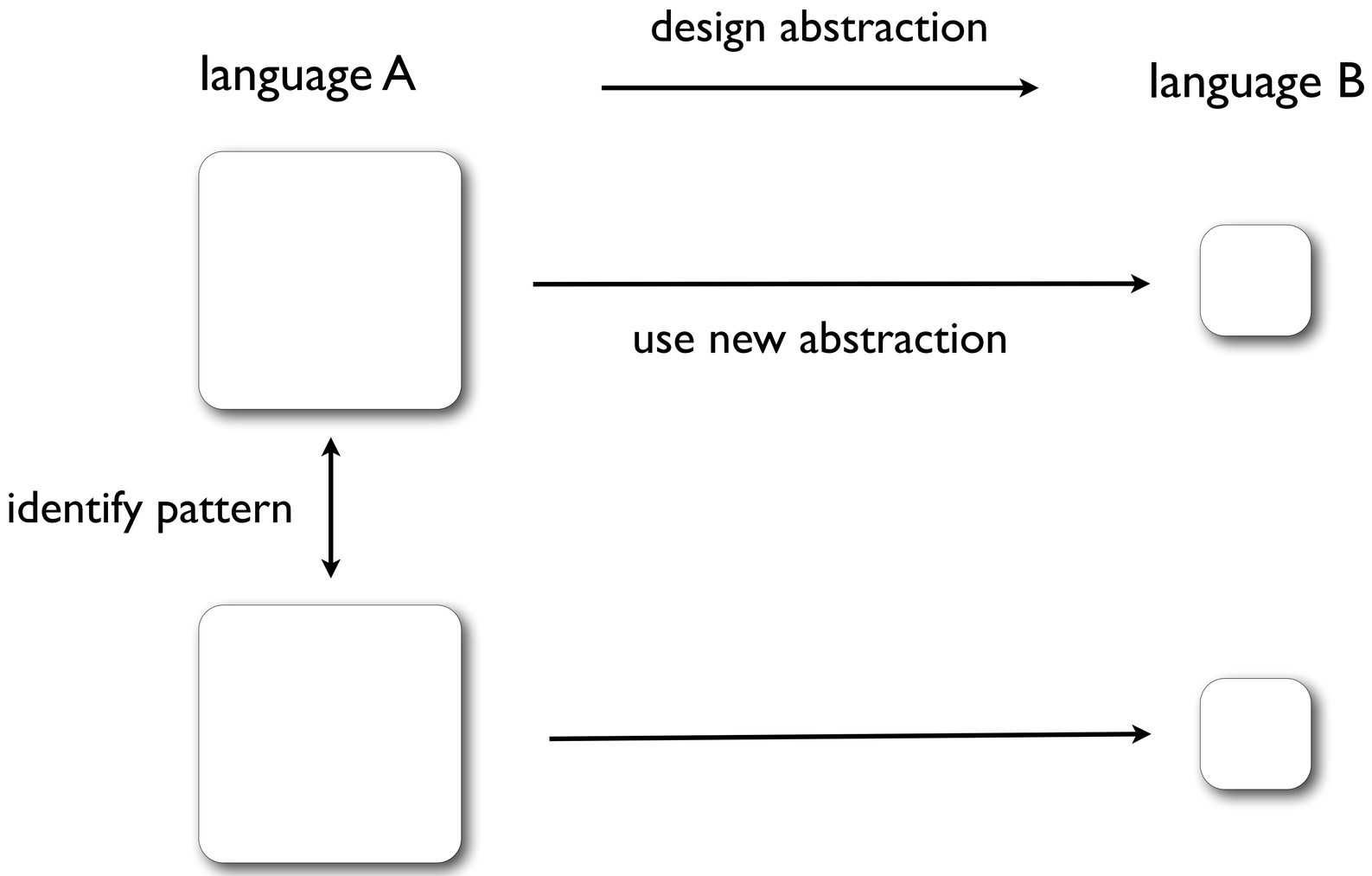
From Malloc to Garbage Collection

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

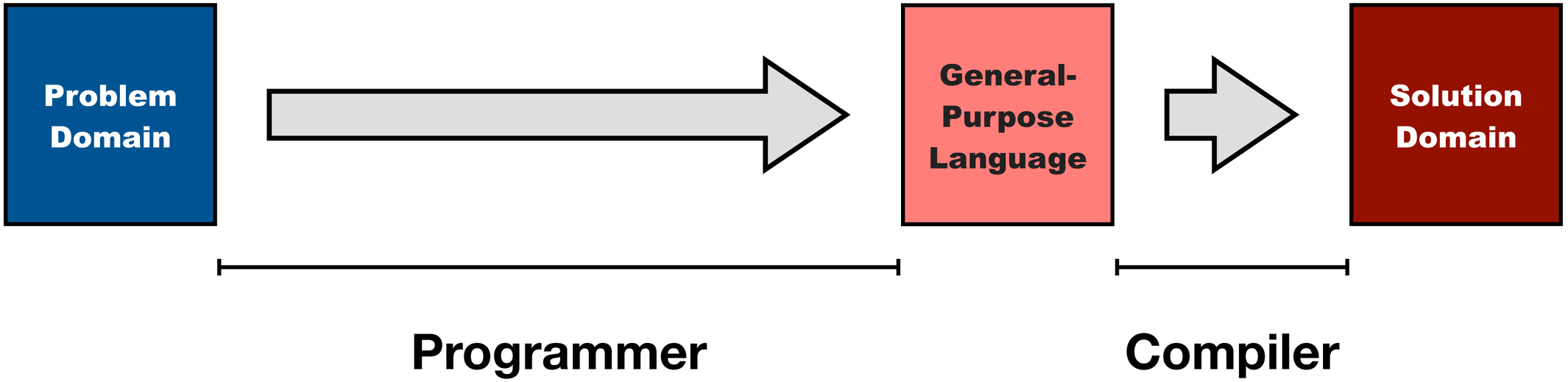
http://en.wikipedia.org/wiki/Malloc

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

Linguistic Abstraction



Compiler Automates Work of Programmer

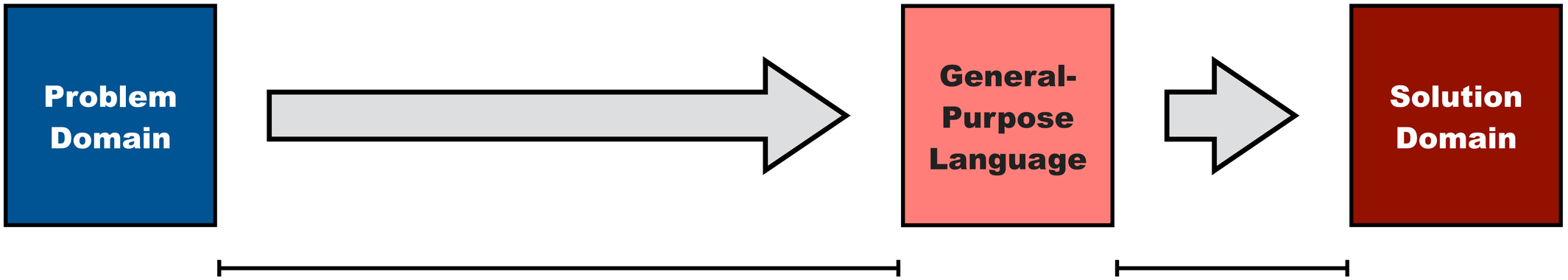


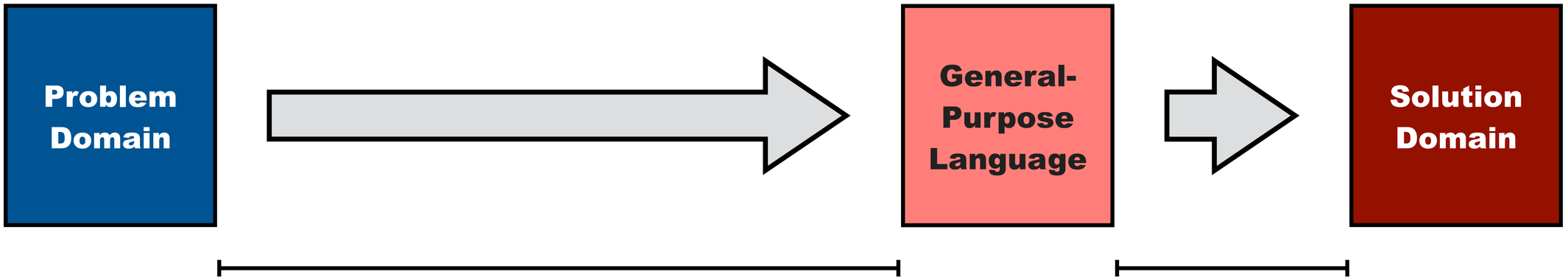
Compilers for modern high-level languages

- 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 (Meta-) Languages

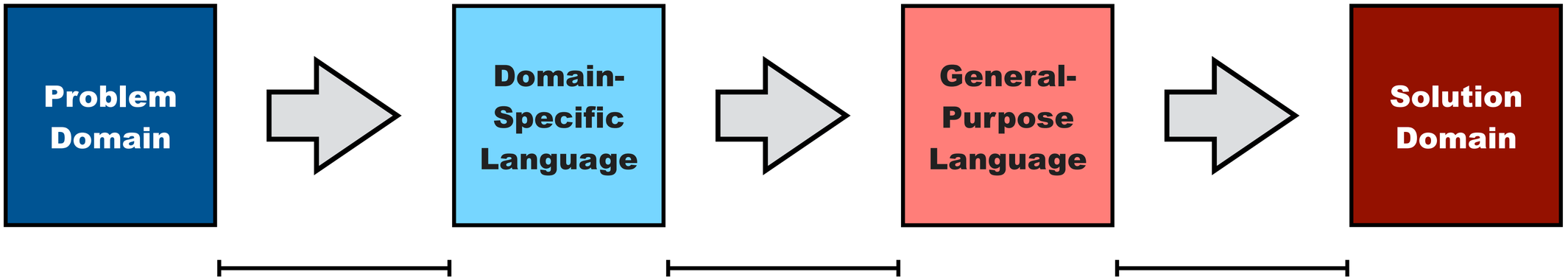
Domains of Computation





"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

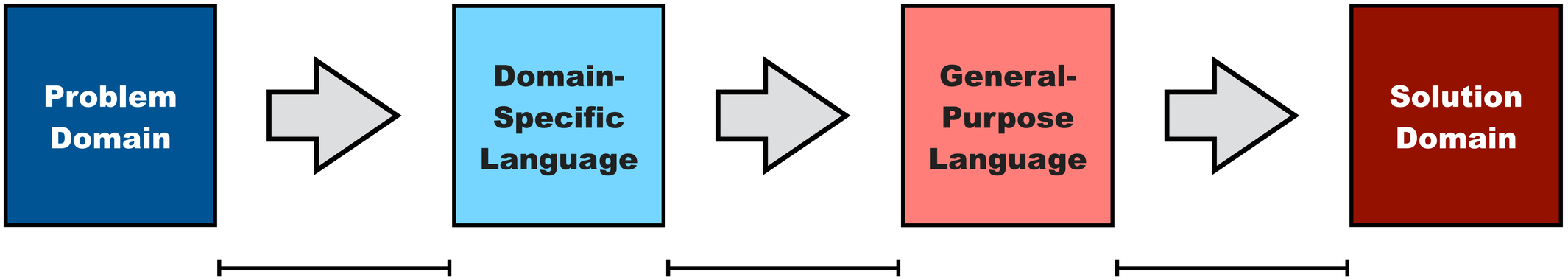
- What is the semantics of such definitions?
- How can we verify the correctness / consistency of language definitions?

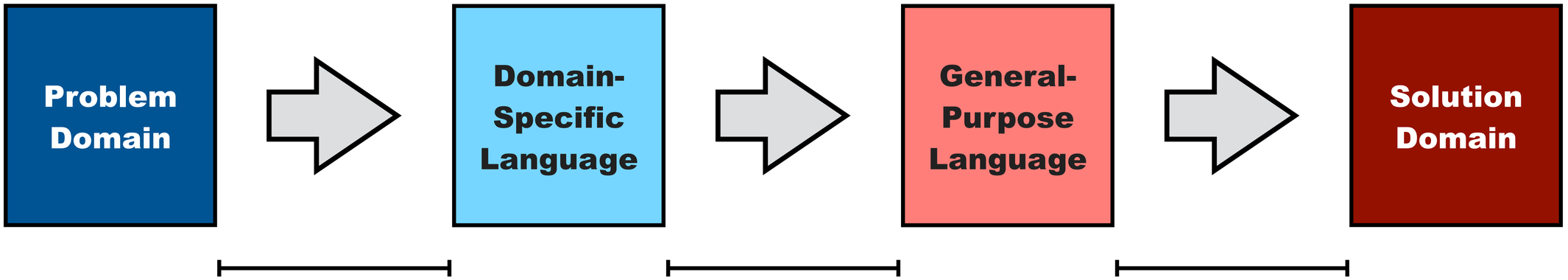
Implementation

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

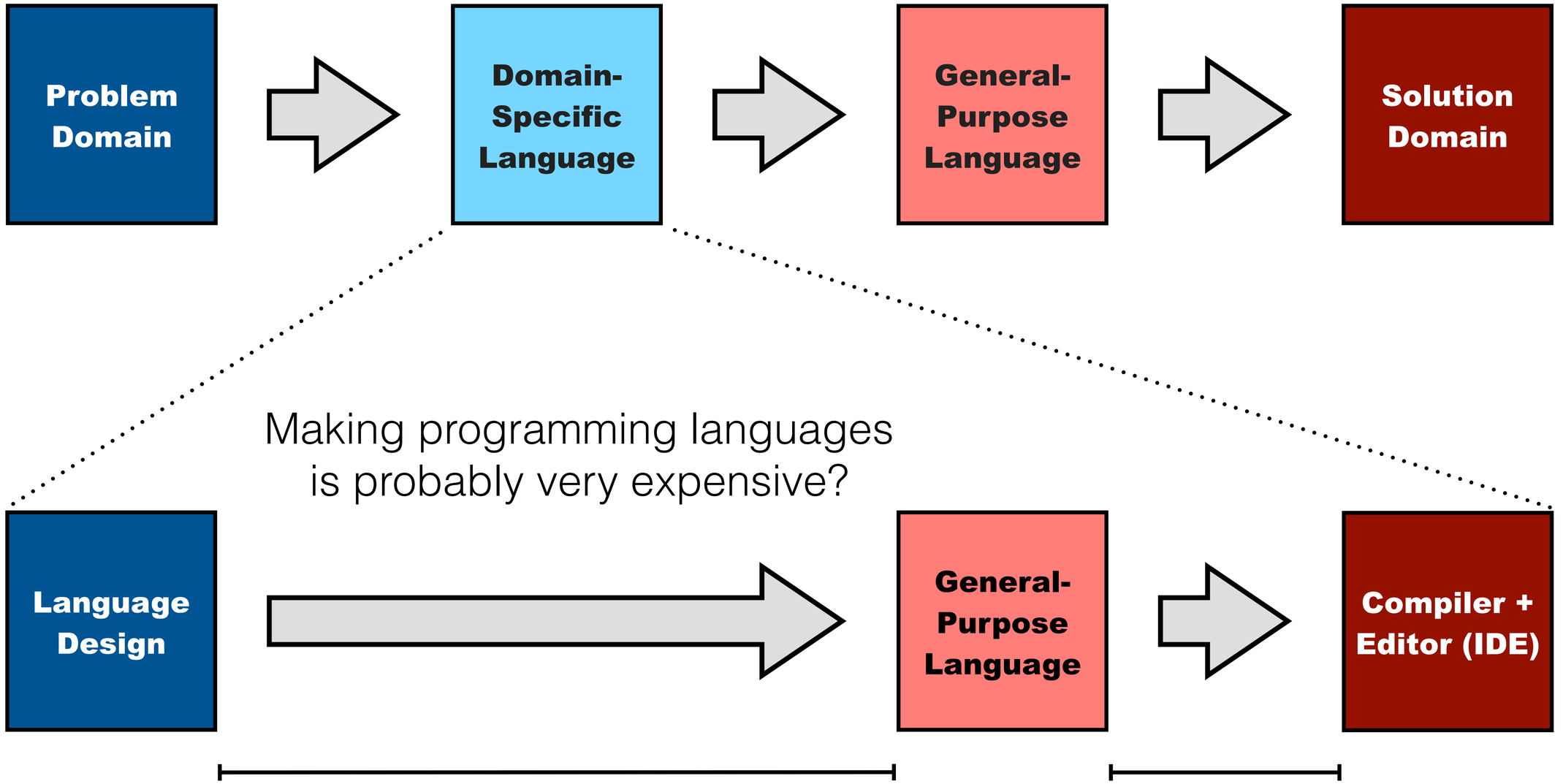
Evaluation

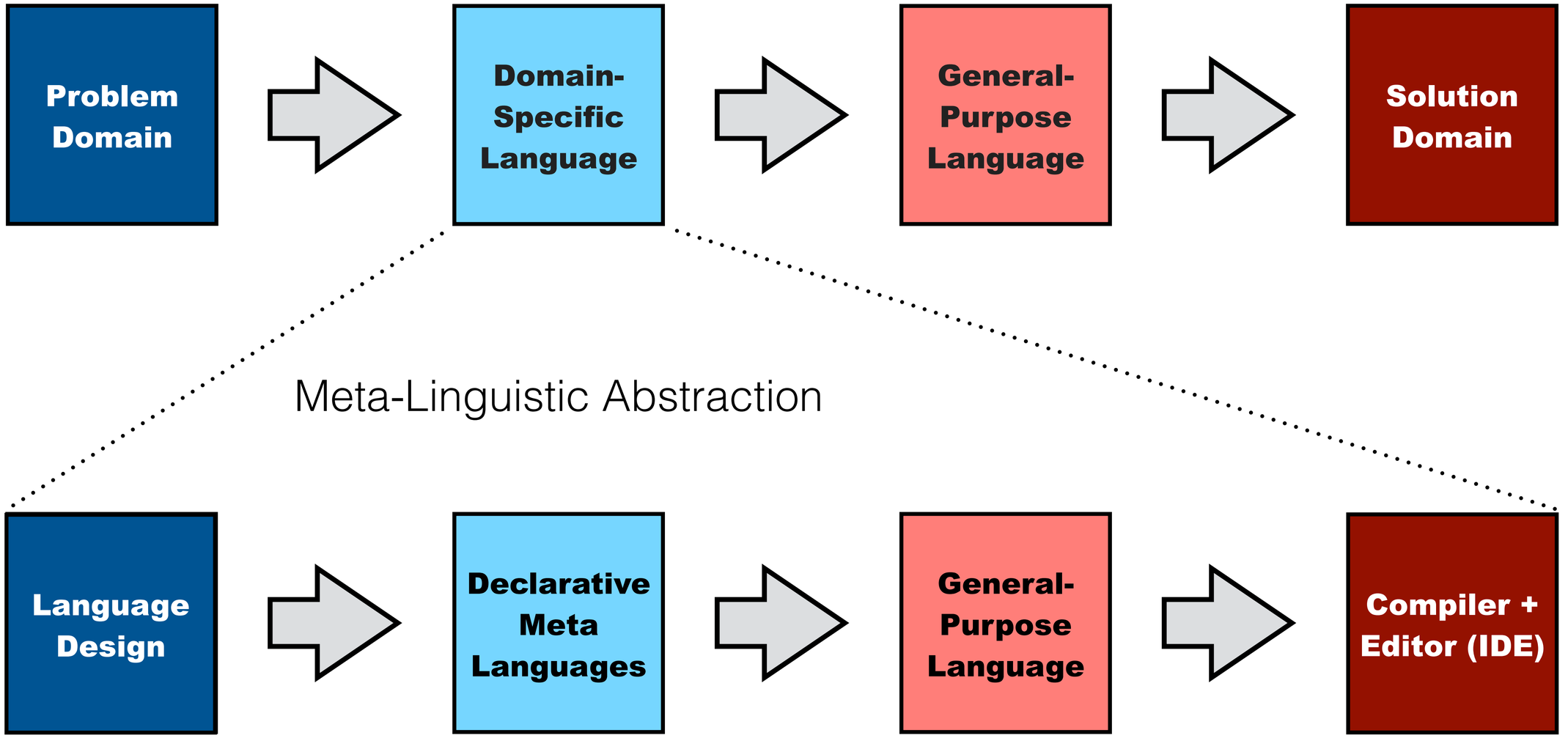
- Apply to new and existing languages to determine adequacy


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

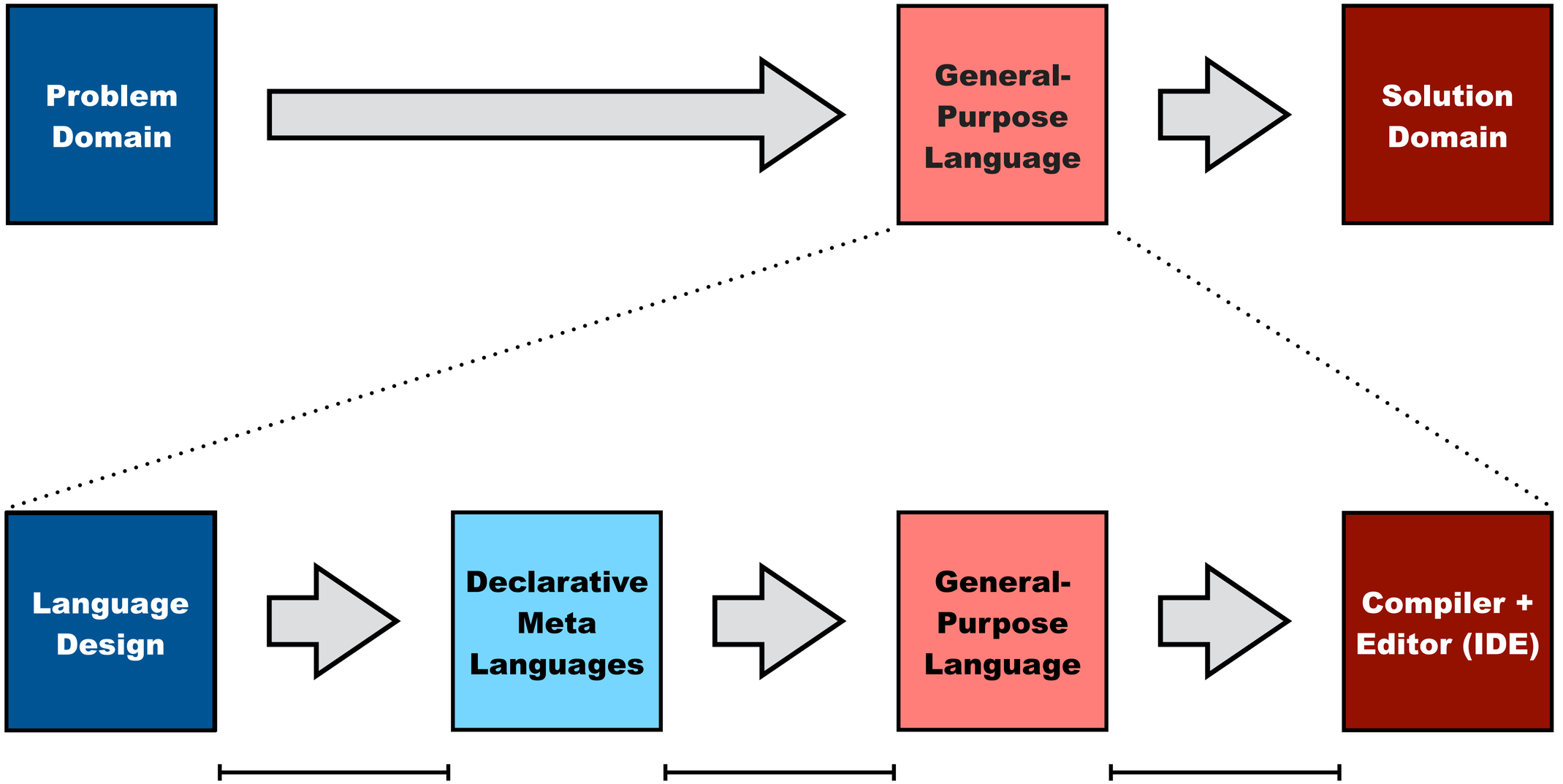


Making programming languages
is probably very expensive?


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)

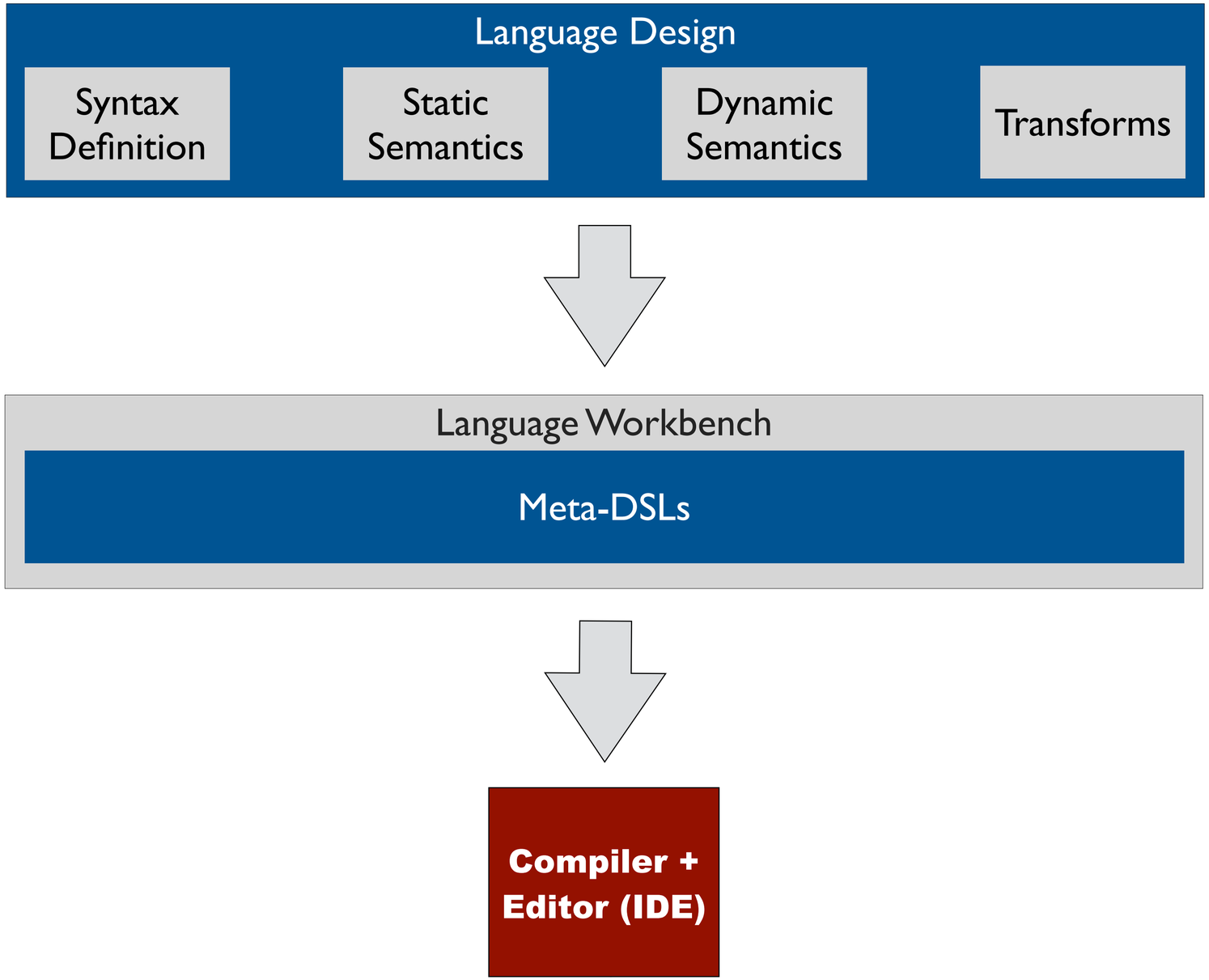


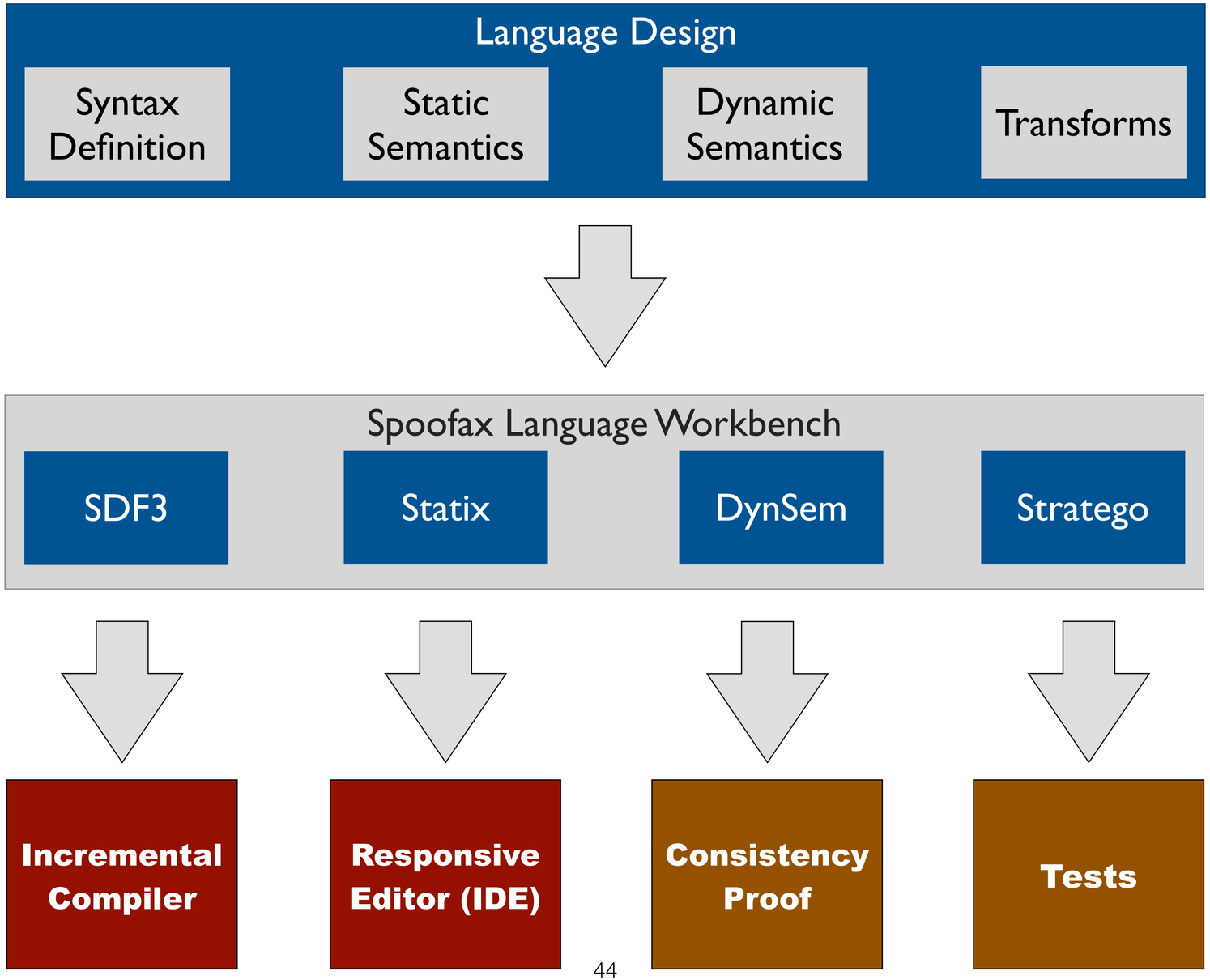
Applying compiler construction to the domain of compiler construction



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
Spoofax Language Workbench
SDF3
Statix
DynSem
Stratego
Incremental Compiler
Responsive Editor (IDE)
Consistency Proof
Tests

Meta-Languages in Spoofax Language Workbench

SDF3: Syntax definition

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

Statix: Names & Types

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

Stratego: Program Transformation

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

FlowSpec: Data-Flow Analysis

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

DynSem: Dynamic Semantics

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

Spoofax in Action

Education

- Compiler Construction (MiniJava, ChocoPy)
- Language Engineering Project (2020: Ada, C, ChocoPy, FlowSpec)

Research

- Language Engineering, Language Prototyping

Academic Workflow Engineering

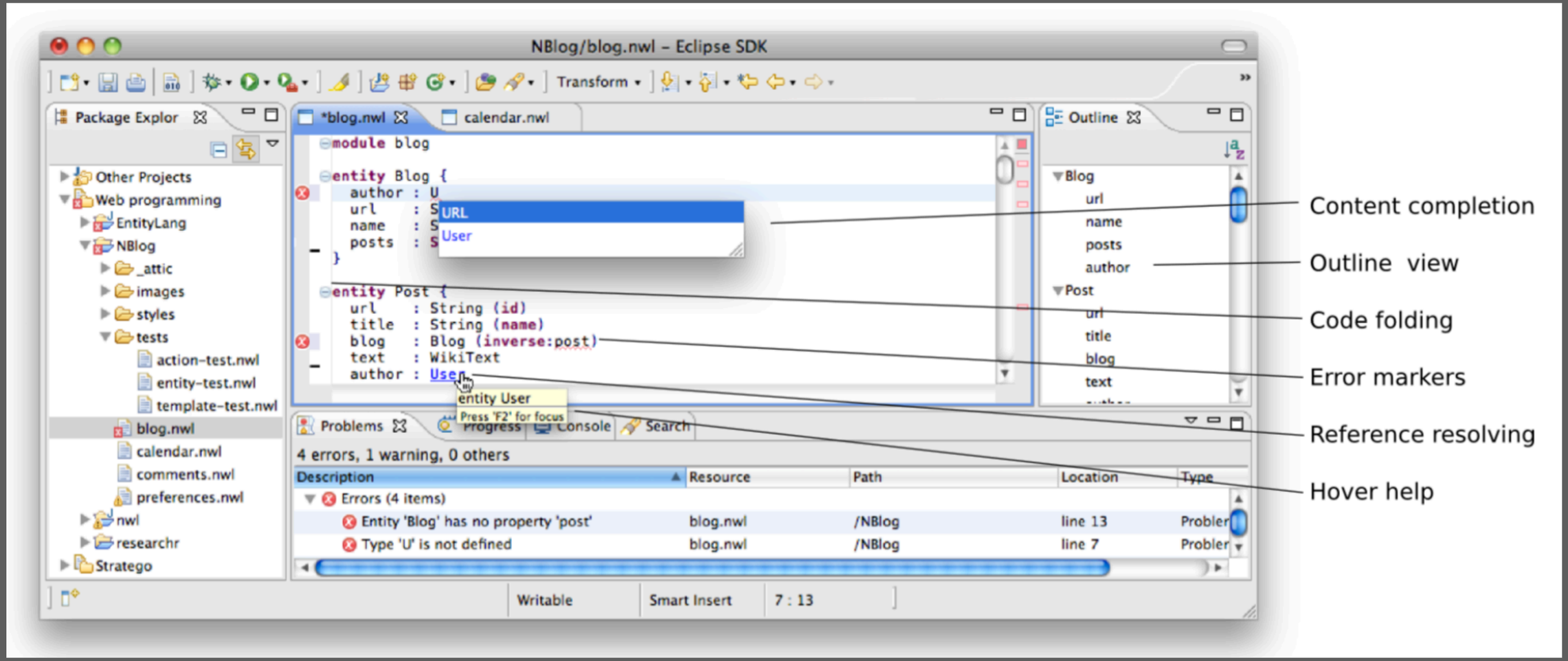
- WebDSL (conf.researchr.org, WebLab, MyStudyPlanning, EvaTool)

Industry

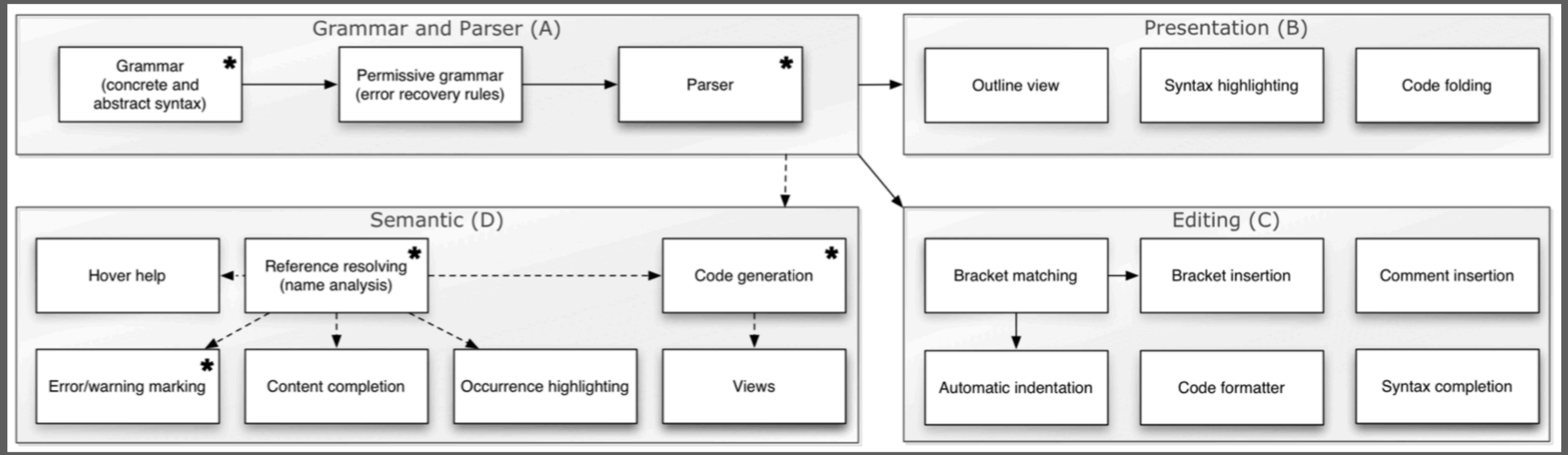
- Oracle Labs: Graph Analytics
- Canon: Several DSLs
- Philips: Software Restructuring

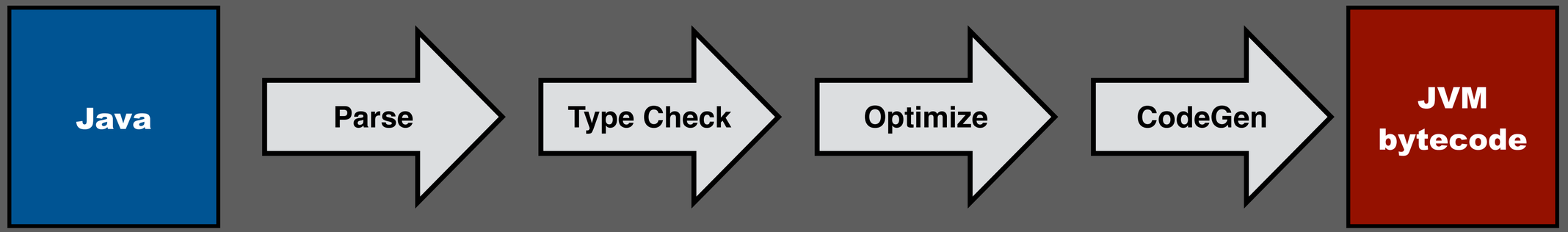
Language Development with a Language Workbench

Programming Environment (IDE)

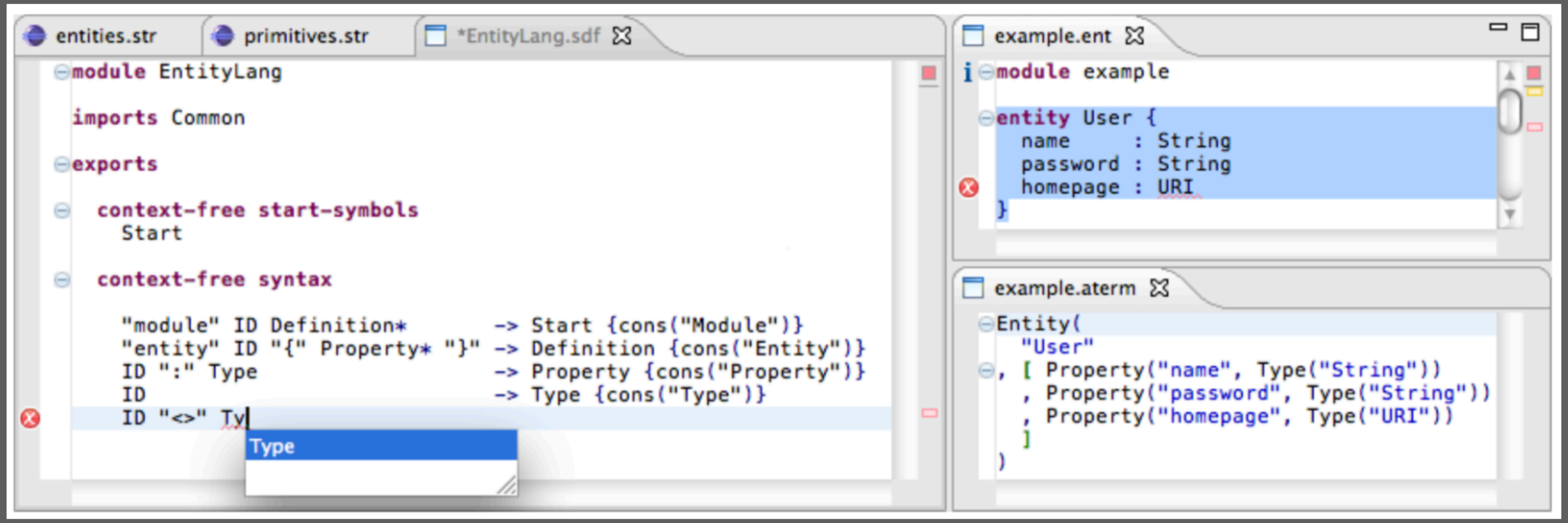


Architecture: IDE vs Compiler





Language Workbench: Live Language Development



ChocoPy: Project Language

ChocoPy: A Typed Restricted Subset of Python 3

```
# Binary-search trees
class TreeNode(object):
   value:int = 0
   left:"TreeNode" = None
   right:"TreeNode" = None

   def insert(self:"TreeNode", x:int) → bool:
      if x < self.value:
         if self.left is None:
            self.left = makeNode(x)
            return True
         else:
            return self.left.insert(x)
      elif x > self.value:
         if self.right is None:
            self.right = makeNode(x)
            return True
         else:
            return self.right.insert(x)
      return False

   def contains(self:"TreeNode", x:int) → bool:
      if x < self.value:
         if self.left is None:
            return False
         else:
            return self.left.contains(x)
      elif x > self.value:
         if self.right is None:
            return False
         else:
            return self.right.contains(x)
      else:
         return True
```

ChocoPy is a programming language designed for classroom use in undergraduate compilers courses. ChocoPy is a restricted subset of Python 3, which can easily be compiled to a target such as RISC-V. The language is fully specified using formal grammar, typing rules, and operational semantics. ChocoPy is used to teach CS 164 at UC Berkeley. ChocoPy has been designed by Rohan Padhye and Koushik Sen, with substantial contributions from Paul Hilfinger.

At a glance, ChocoPy is:

- **Familiar**: ChocoPy programs can be executed directly in a Python (3.6+) interpreter. ChocoPy programs can also be edited using standard Python syntax highlighting.
- **Safe**: ChocoPy uses Python 3.6 type annotations to enforce static type checking. The type system supports nominal subtyping.
- **Concise**: A full compiler for ChocoPy be implemented in about 12 weeks by undergraduate students of computer science. This can be a hugely rewarding exercise for students.
- **Expressive**: One can write non-trivial ChocoPy programs using lists, classes, and nested functions. Such language features also lead to interesting implications for compiler design.

Bonus: Due to static type safety and ahead-of-time compilation, most student implementations outperform the reference Python implementation on non-trivial benchmarks.

Source: https://chocopy.org/

A Compiler and IDE for ChocoPy

ChocoPy IDE with syntax checking, syntax coloring, type checking (CS4200-A)

Compiler from ChocoPy to RISC-V (CS4200-B)

Executing RISC-V with simulator

binary_tree.cpy

```
51          else:
52              return self.root.contains(x)
53
54  def makeNode(x: int) -> TreeNode:
55      b:TreeNode = None
56      b = TreeNode()
57      b.value = x
58      return b
59
60
65  # Data
66  t:Tree = None
67  i:int = 0
68  k:int = 37813
69
70  # Crunch
71  t = Tree()
72  while i < n:
73      t.insert(k)
74      k = (k * 37813) % 37831
75      if i % c != 0:
76          t.insert(i)
77      i = i + 1
78
79  print(t.size)
80
81  for i in [4, 8, 15, 16, 23, 42]:
82      if t.contains(i):
83          print(i)
84
```

binary_tree.rv32im

```
1  .equiv @sbrk, 9
2  .equiv @print_string, 4
3  .equiv @print_char, 11
4  .equiv @print_int, 1
5  .equiv @exit2, 17
6  .equiv @read_string, 8
7  .equiv @fill_line_buffer, 18
8  .equiv @.__obj_size__, 4
9  .equiv @.__len__, 12
10 .equiv @.__int__, 12
11 .equiv @
12 .equiv @
13 .equiv @
14 .equiv @error_div_zero, 2
15 .equiv @error_arg, 1
16 .equiv @error_oob, 3
17 .equiv @error_none, 4
18 .equiv @error_oom, 5
19 .equiv @error_nyi, 6
20 .equiv @listHeaderWords, 4
21 .equiv @bool.True, const_39
22 .equiv @bool.False, const_38
23
24 .data
25
26 .globl $object$prototype
27 $object$prototype:
28 .word 0
```

binary_tree.result.txt

```
1 175
2 15
3 23
4 42
5
```

ChocoPy: Language Manual and Reference

Start Reading!

ChocoPy v2.2: Language Manual and Reference

Designed by Rohan Padhye and Koushik Sen; v2 changes by Paul Hilfnger

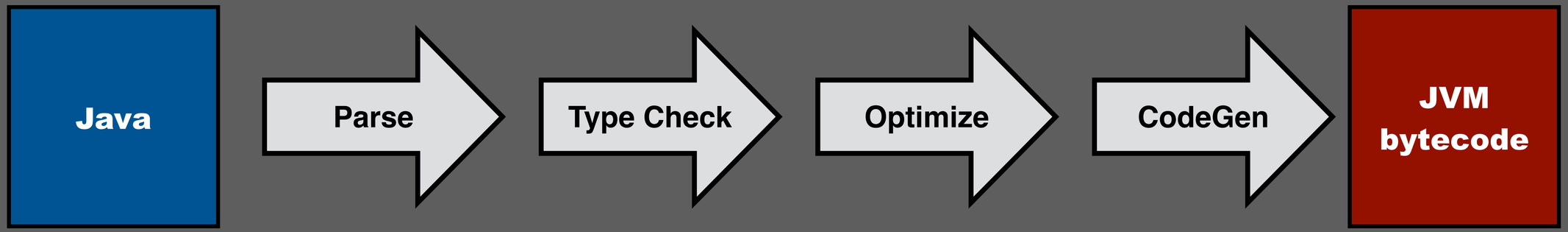
University of California, Berkeley

November 23, 2019

Contents

Studying Compiler Construction

The Basis



Levels of Understanding Compilers

Specific

- Understanding a specific compiler
- Understanding a programming language (ChocoPy)
- Understanding a target machine (RISC-V)
- Understanding a compilation scheme (ChocoPy to RISC-V)

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

CS4200: Two Courses

CS4200-A: Front-End (5 ECTS)

- Syntax and Type Checking
- Project: Build front-end of compiler for ChocoPy in Spoofax
- Exam in October

CS4200-B: Back-End (5 ECTS)

- Analysis and Code Generation
- Project: Build back-end of compiler for ChocoPy in Spoofax
- Exam in January

Lectures Topics CS4200-A (Tentative)

- What is a compiler?
- Syntax Definition
- Disambiguation and Layout-Sensitive Syntax
- Syntactic Editor Services
- Static Semantics & Name Resolution
- Type Checking
- Specification with Statix
- Constraint Resolution
- Parsing

Lectures: Thursday, ~~10:45~~ 11:00

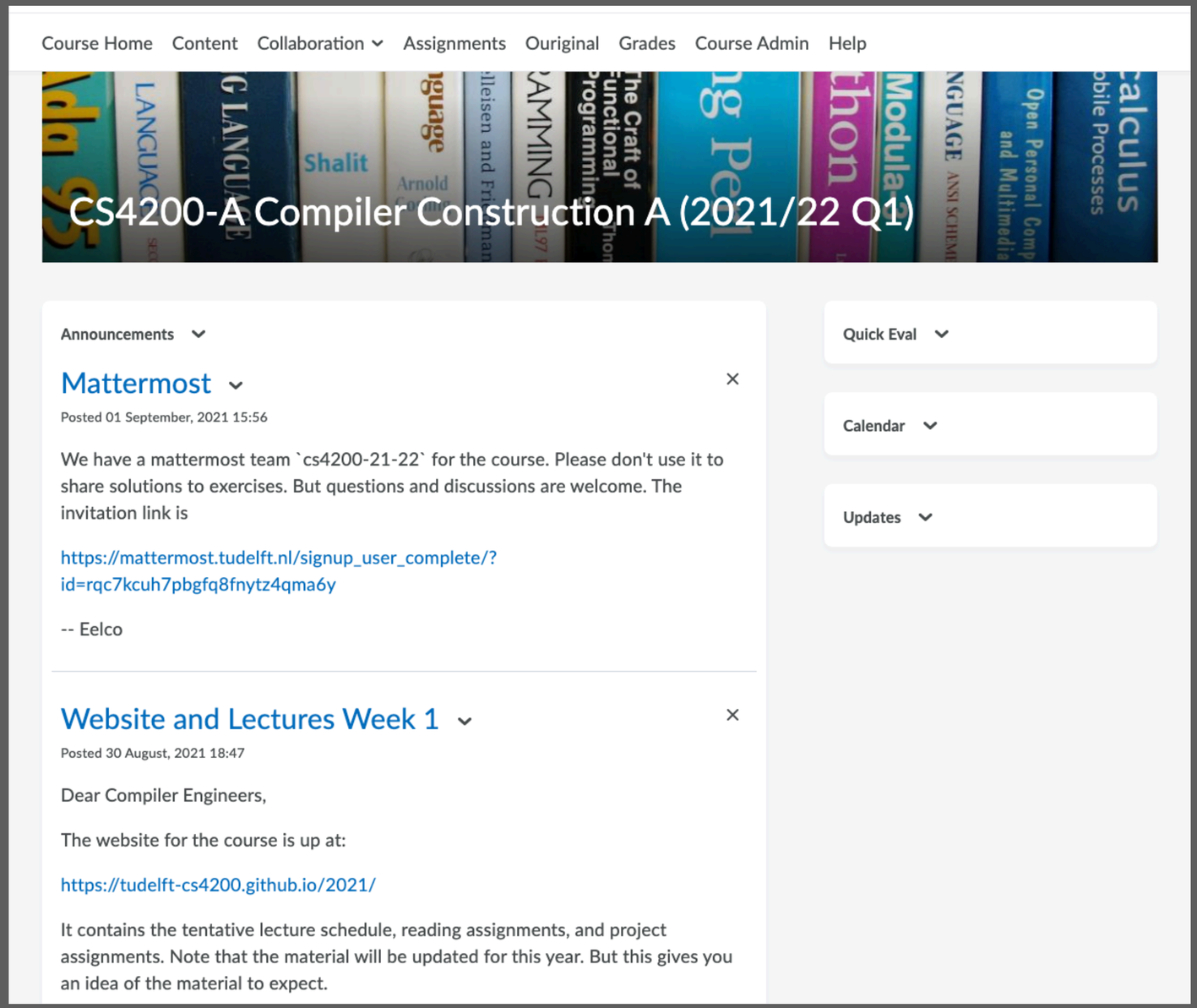
Extra Lecture: Friday, Sept 3, 13:45

Lab: Tuesday, 10:45, 13:45

Lecture Topics CS4200-B (Q2) (Tentative)

- Virtual Machines
- Transformation
- Code Generation
- Data-Flow Analysis
- Monotone Frameworks
- Register Allocation
- Memory Management

Brightspace: Announcements



Course Website

TU Delft | CS4200 Lectures Homework Project News Blog

TU Delft | CS4200 | 2021-2022
Compiler Construction

Master Computer Science

Common Core Software Technology

Specialization Programming Languages

Credits: 2 x 5 ECTS

Schedule

Lectures: Thursday 11:00 - 12:45

Lab Q1: Tuesday 10:45 - 12:45, 13:45 - 15:45

Lab Q2: Friday 13:45 - 17:45

Shared Study Lab on Wednesday

Organization

Programming Languages Group

Department of Software Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

Delft University of Technology

Delft, The Netherlands (CEST/CET)

Course Staff

Lecturer: Eelco Visser (e.visser@tudelft.nl)

TA: Ruben van Baarle

TA: Thijs Molendijk

TA: Aron Zwaan

Compiler Construction

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 compiler construction program consists of two courses of 5 ECTS each.

CS4200-A: Front-End (Q1)

In the first course we study the front-end of the compiler that parses and type checks programs. We study meta-languages for the definition of the syntax and static semantics of programming languages. [Study guide]

CS4200-B: Back-End (Q2)

In the second course we study the back-end of the compiler that performs static analysis, optimization, and code generation. [Study guide]

Spoofax Language Workbench

The Spoofax Language Workbench is an environment for the development of programming languages using high-level declarative meta-languages for syntax, static semantics, dynamic semantics, program analysis, and program transformation. We use Spoofax to implement a compiler and IDE for the project language and in homework assignments to experiment with various aspects of compilers in isolation.

ChocoPy: Project Language

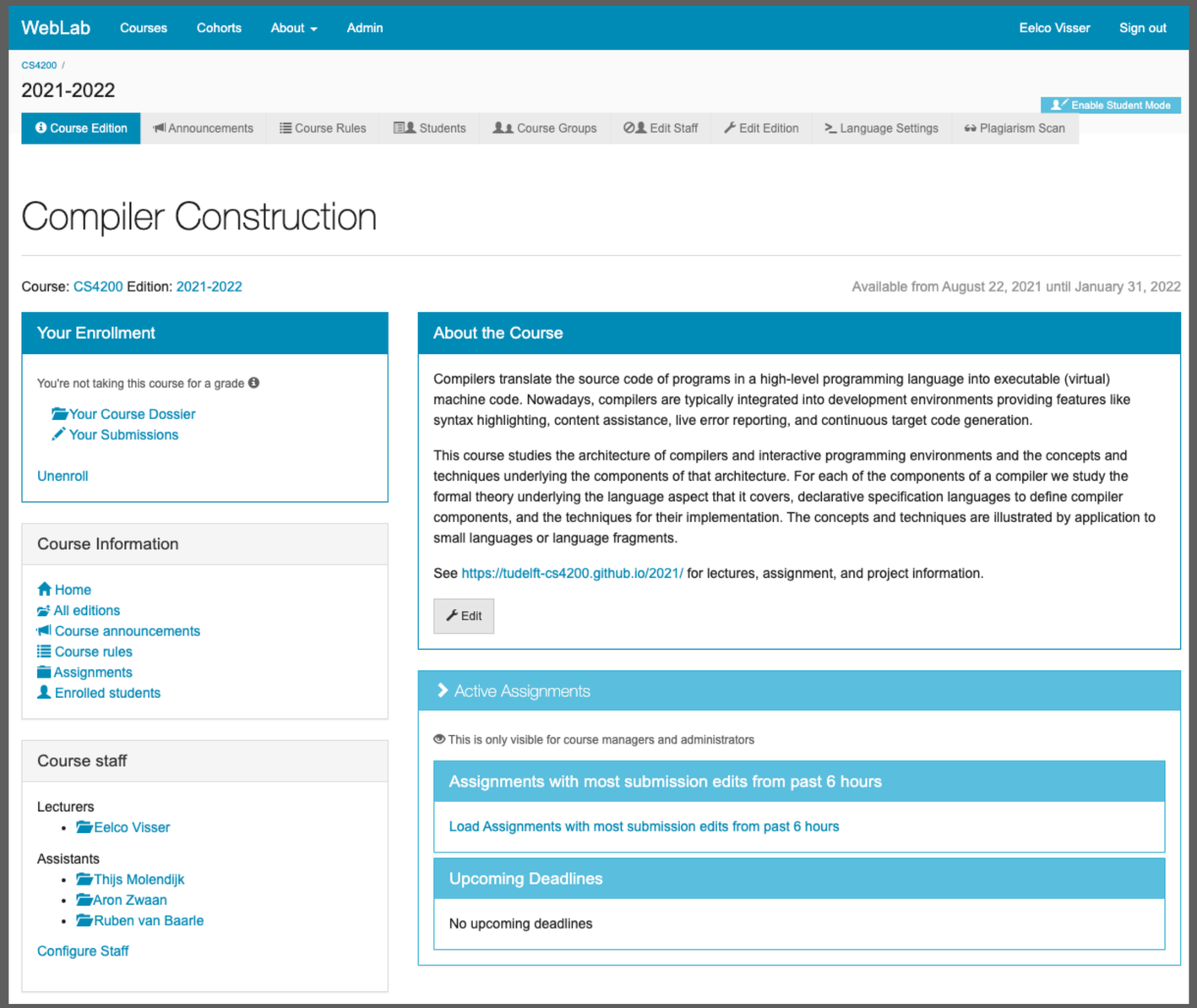
In the course project students implement a compiler for ChocoPy, a subset of Python3 with static types. The back-end of the compiler generates RISC-V code, which can be run with a simulator running on the JVM. The language comes with a fully specified using formal grammar, typing rules, and operational semantics. ChocoPy was developed at UC Berkeley for their compiler course.

WebLab: Homework Assignments and Exams

We use the WebLab learning management system for homework assignments. In WebLab you can submit answers to exercises

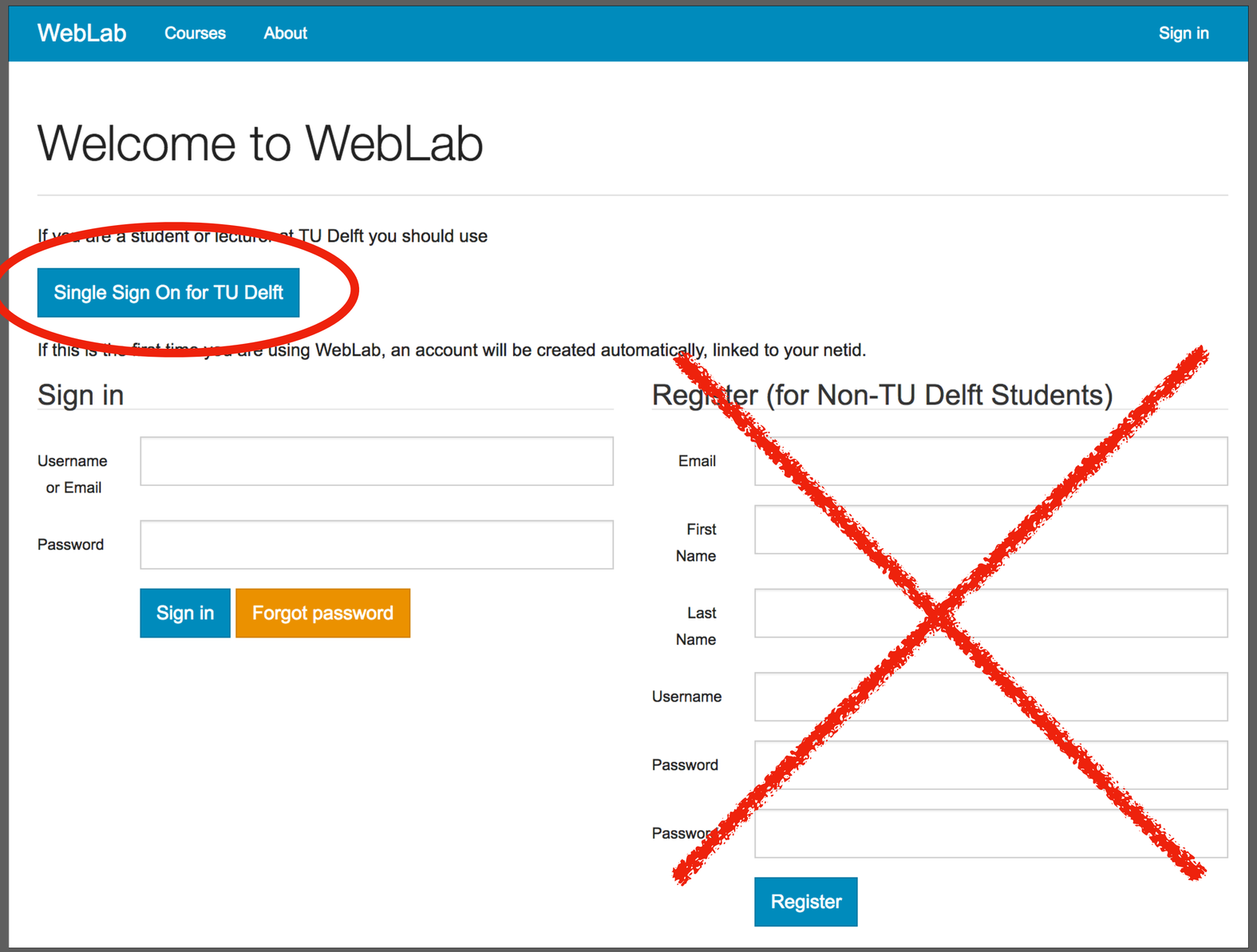
https://tudelft-cs4200.github.io/2021

WebLab for Homework, Exams, Grade Registration

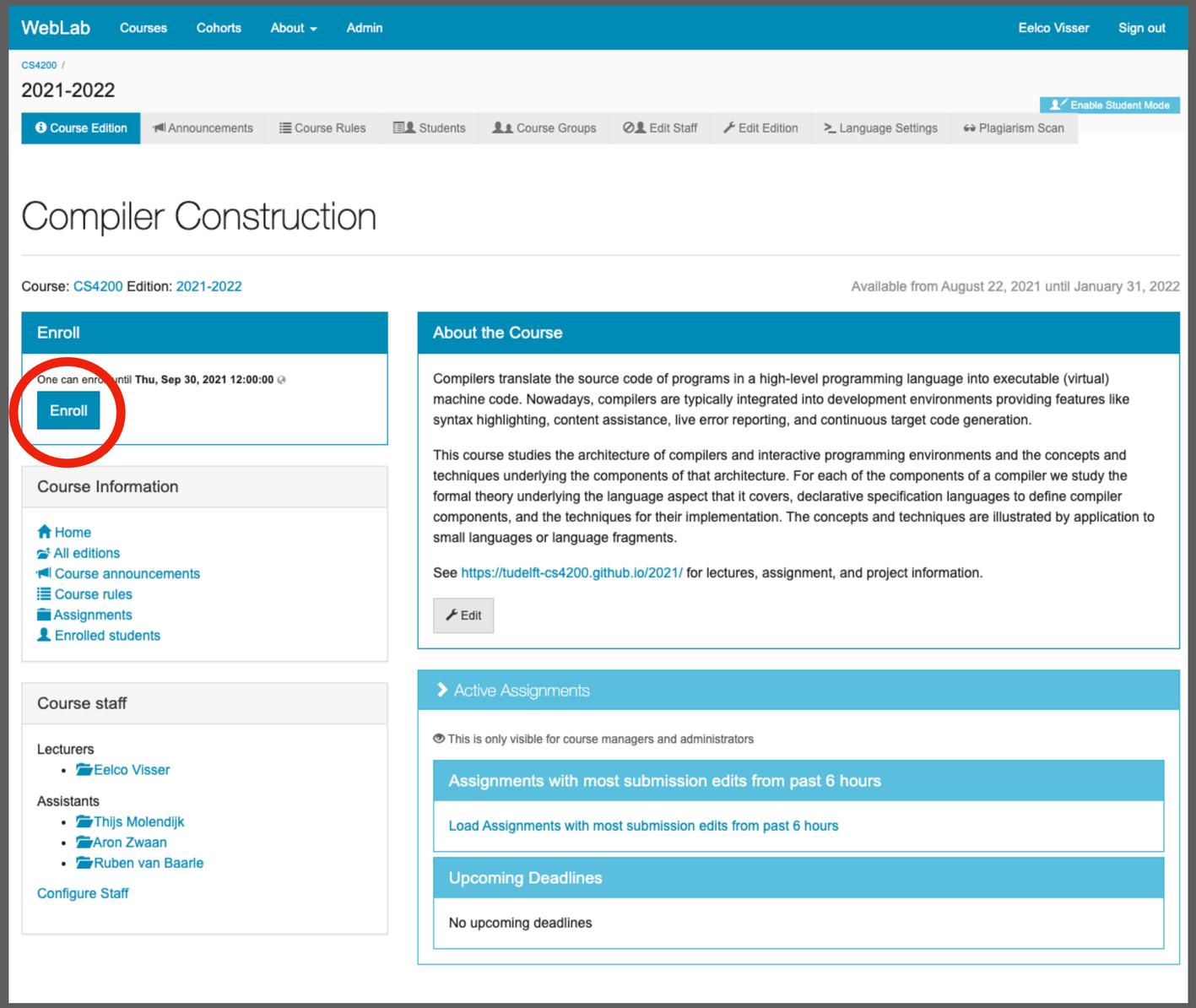


https://weblab.tudelft.nl/cs4200/2021-2022/

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



Enroll for Course CS4200 in WebLab



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!

Reading Material

ChocoPy: Language Manual and Reference

The reference manual of the course project language. The language is a subset of Python 3.

Start Reading!

https://chocopy.org

ChocoPy v2.2: Language Manual and Reference

Designed by Rohan Padhye and Koushik Sen; v2 changes by Paul Hilfnger

University of California, Berkeley

November 23, 2019

Contents

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.

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

This paper gives an overview of the Syntax Definition Formalism SDF3, the language for syntax definition in Spoofax and in this course.

It provides a summary of research on syntax definition that we did in the last 20 years and provides a good introduction to the features of SDF3 that we will study in the next couple of weeks.

https://doi.org/10.1007/978-3-030-58768-0_1

Multi-Purpose Syntax Definition with SDF3

Luís Eduardo Amorim de Souza[1] and Eelco Visser[2]

[1] Australian National University, Australia
[2] Delft University of Technology, The Netherlands

Abstract. SDF3 is a syntax definition formalism that extends plain context-free grammars with features such as constructor declarations, declarative disambiguation rules, character-level grammars, permissive syntax, layout constraints, formatting templates, placeholder syntax, and modular composition. These features support the multi-purpose interpretation of syntax definitions, including derivation of type schemas for abstract syntax tree representations, scannerless generalized parsing of the full class of context-free grammars, error recovery, layout-sensitive parsing, parenthesization and formatting, and syntactic completion. This paper gives a high level overview of SDF3 by means of examples and provides a guide to the literature for further details.

Keywords: Syntax definition · programming language · parsing.

1 Introduction

A syntax definition formalism is a formal language to describe the syntax of formal languages. At the core of a syntax definition formalism is a *grammar formalism* in the tradition of Chomsky's context-free grammars [14] and the Backus-Naur Form [4]. But syntax definition is concerned with more than just phrase structure, and encompasses all aspects of the syntax of languages.

In this paper, we give an overview of the syntax definition formalism SDF3 and its tool ecosystem that supports the multi-purpose interpretation of syntax definitions. The paper does not present any new technical contributions, but it is the first paper to give a (high-level) overview of all aspects of SDF3 and serves as a guide to the literature. SDF3 is the third generation in the SDF family of syntax definition formalisms, which were developed in the context of the ASF+SDF [5], Stratego/XT [10], and Spoofax [38] language workbenches.

The first SDF [23] supported modular composition of syntax definition, a direct correspondence between concrete and abstract syntax, and parsing with the full class of context-free grammars enabled by the Generalized-LR (GLR) parsing algorithm [56,44]. Its programming environment, as part of the ASF+SDF MetaEnvironment [40], focused on live development of syntax definitions through

To appear in: F. S. de Boer and A. Cerone (Eds.). *Software Engineering and Formal Methods (SEFM 2020)*, LNCS, Springer, 2020.

Next: Declarative Syntax Definition
Friday, Sept 3 at 13:45!