

Compiler Construction

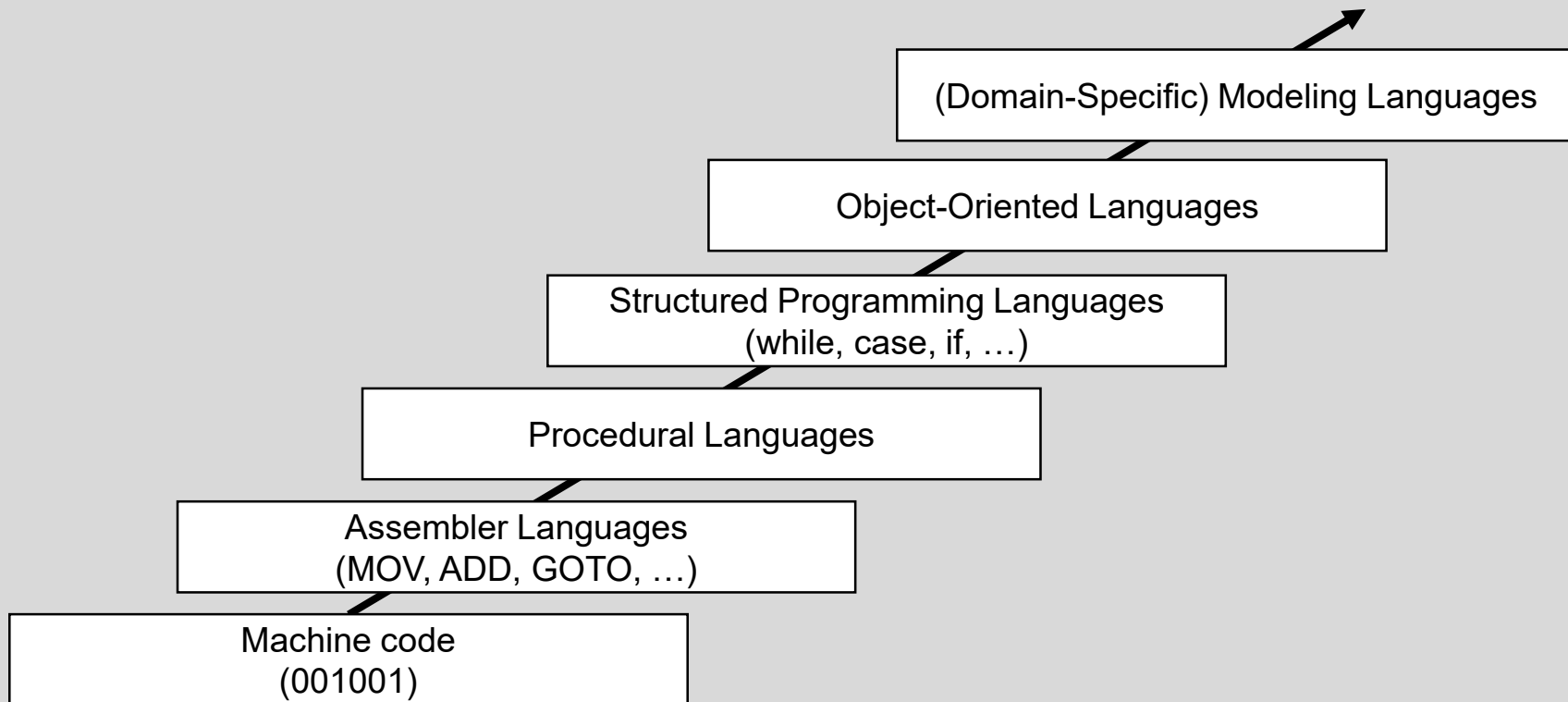
Introduction

Prof. Dr. Timo Kehrer, Master Course, FS2023

23rd February 2023, Hörraum 120, Hauptgebäude H4

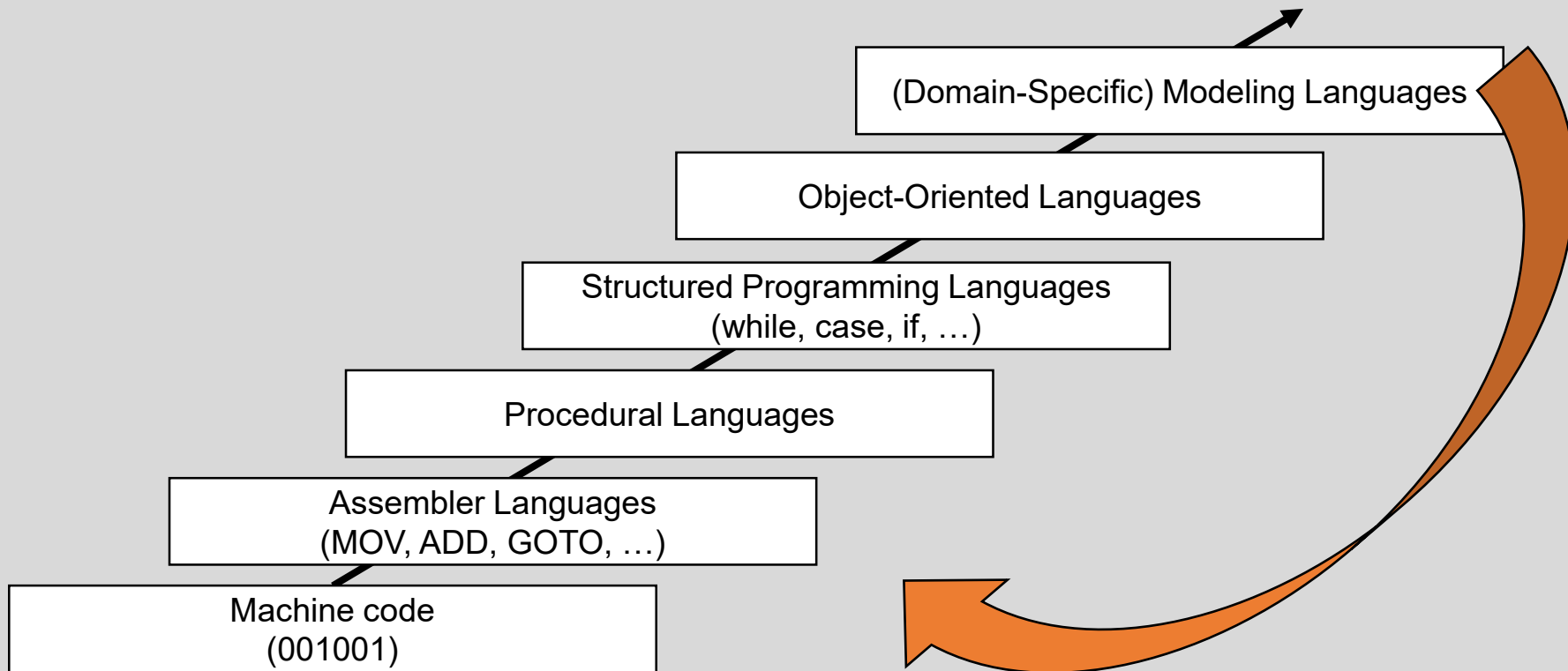
Translating computer languages

Hierarchy of abstraction



Translating computer languages

Hierarchy of abstraction

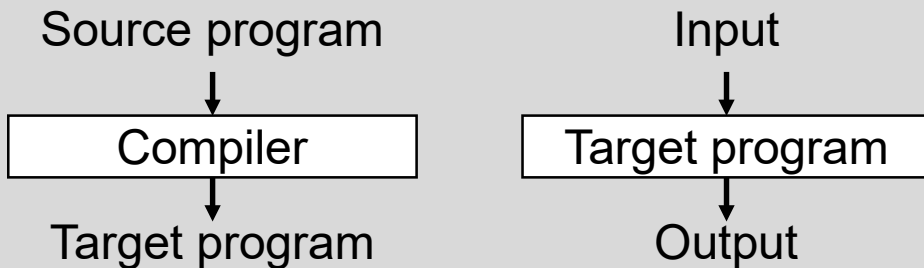


Translating computer languages

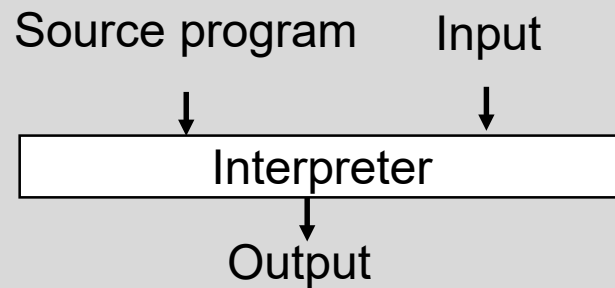
Compiler and interpreter

- A **compiler** is a program which takes an executable program of a certain language A as input and translates it into an executable program of a different language B.
- An **interpreter** is a program which reads an executable program and gives some result through its execution.

Compiler

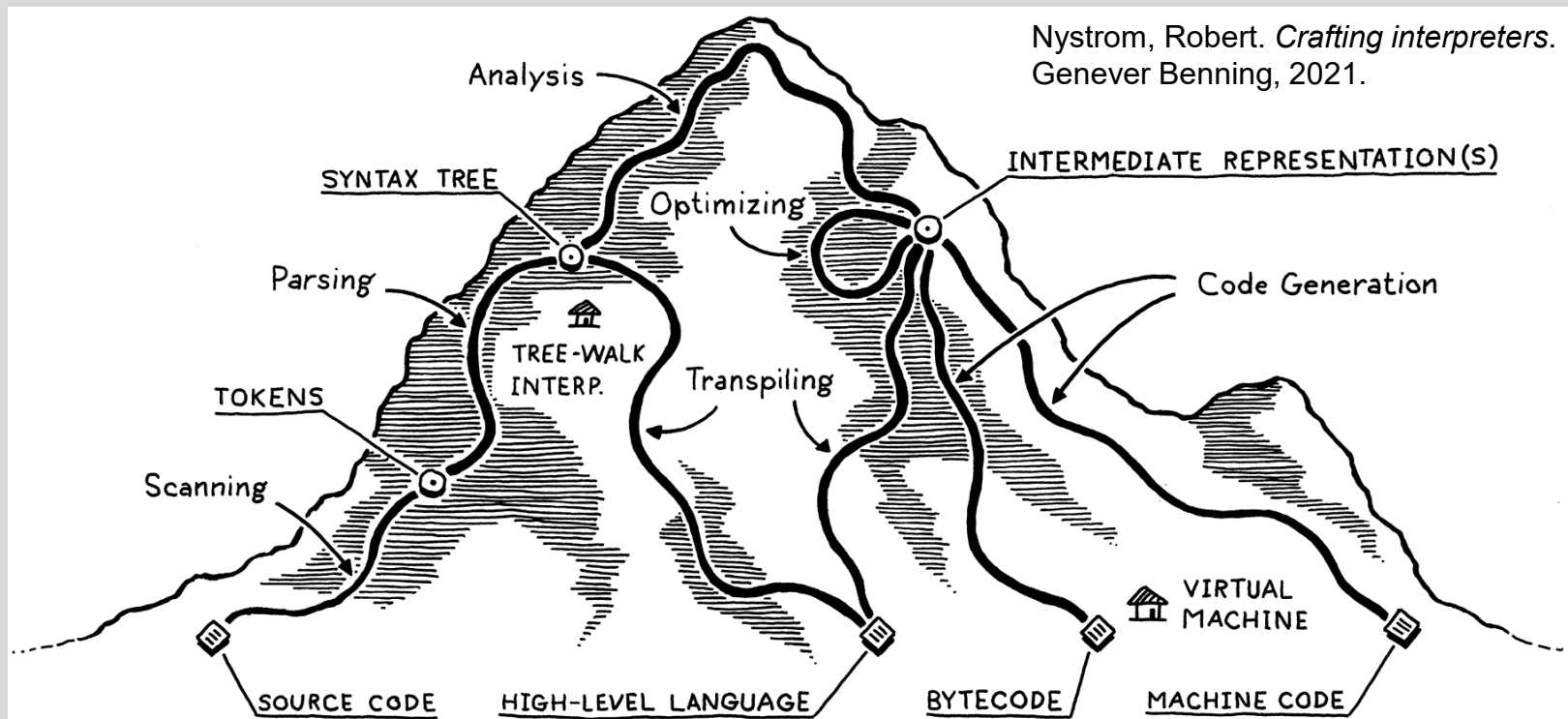


Interpreter



Translating computer languages

A Map of the Territory



Scanning (Lexical Analysis)

Tasks of a scanner

- Accepts **words** of a language.
- Technically: Translates a sequence of characters into a sequence of tokens.
 - token: atomic unit used in syntactic analysis
 - typical (types of) tokens: number, id, +, -, do, end
 - general structure: <token name, lexeme>
 - lexeme: string value of a token
- Removes white space (blanks, tabs, comments, etc.) and other symbols that have no meaning but are used for formatting.

Scanning (Lexical Analysis)

Example

position := initial + rate * 40



Lexical analysis



$\langle \text{id}, \text{position} \rangle := \langle \text{id}, \text{initial} \rangle + \langle \text{id}, \text{rate} \rangle * \langle \text{int}, 40 \rangle$

For brevity, we will often omit the explicit distinction between token type and lexeme and write something like this:

$$id_1 := id_2 + id_3 * 40$$

Parsing (Syntactic Analysis)

Tasks of a parser

- Recognizes **sentences** of a language.
- Technically: Takes the (flat) sequence of tokens and builds a tree structure that mirrors the nested nature of the language's syntax.
- The resulting trees have a couple of different names, usually **parse tree** or **syntax tree**.

Parsing (Syntactic Analysis)

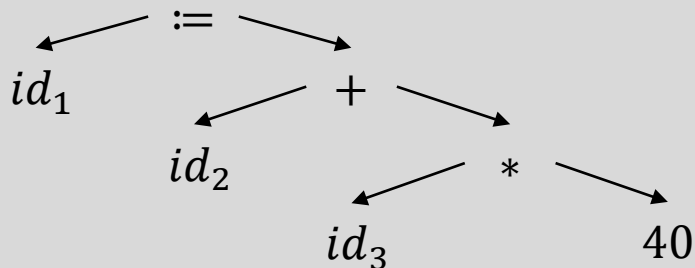
Example

position := initial + rate * 40

Lexical analysis

$id_1 := id_2 + id_3 * 40$

Syntactic analysis



Semantic Analysis (Static Analysis)

Motivation

Are our sentences semantically correct? What is their meaning? Answering questions like these includes, for example:

- Binding/resolution:
 - For each identifier, is it a declaration or a reference and if so, where is it defined?
 - Scope: The region of source code where a certain name can be used to refer to a certain declaration.
- Type checking (in case of a statically typed language):
 - Once we know where referenced elements are declared, we can also figure out their types.
 - If we detect incompatibilities in an expression, we report a type error.
 - This is where some implicit type casts are performed, if supported by our language.

Semantic Analysis (Static Analysis)

Motivation

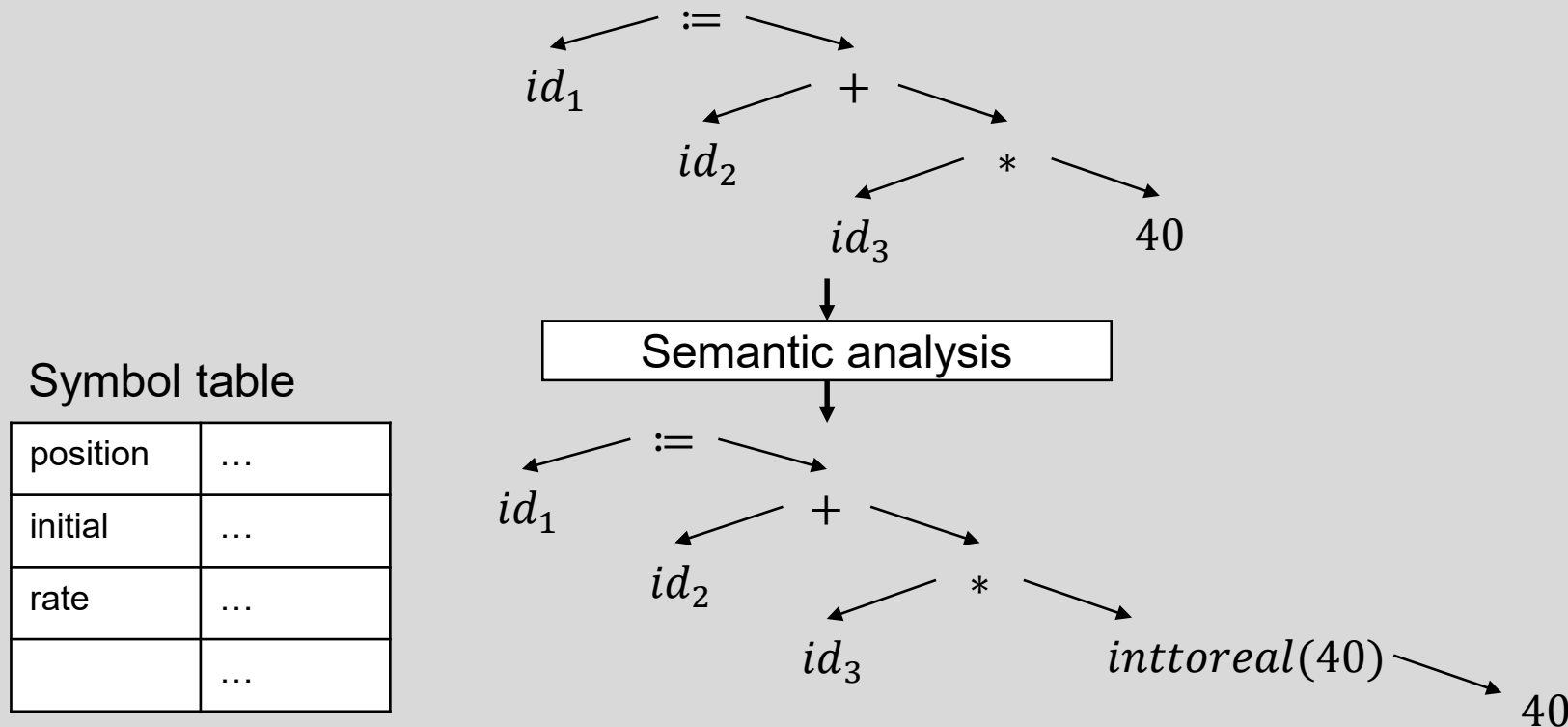
Semantic insights gained from analysis need to be stored somewhere.

Typical solutions are:

- In the syntax tree itself: Attributes of the nodes that are not initialized during parsing.
- Symbol table: A lookup table off to the side, where identifier names are used as keys.

Semantic Analysis (Static Analysis)

Example



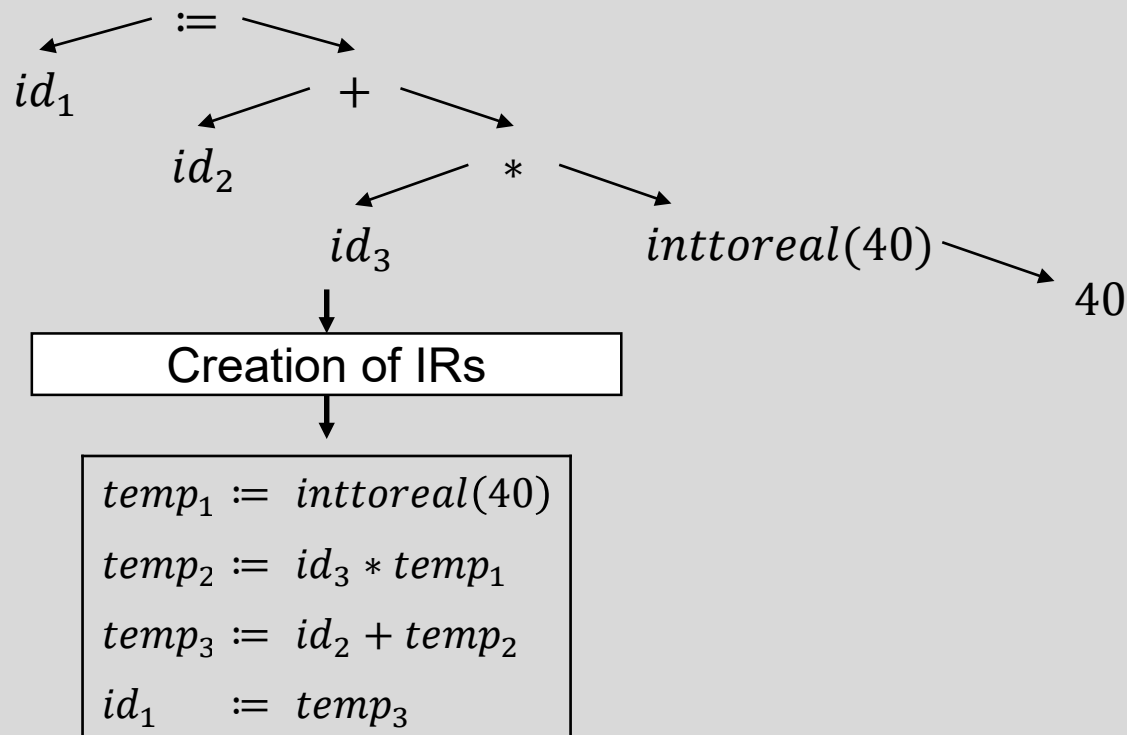
Intermediate Representation(s)

Introduction

- Often, the syntax tree is transformed into a new data structure that more directly expresses (parts of) the semantics of the code.
- This structure is typically called **Intermediate Representation** (IR, for short).
- A few well-established styles of IRs, e.g.
 - Control flow graph
 - Static single-assignment
 - Continuation-passing style
 - Three-address code
 - etc.

Intermediate Representation(s)

Example



Optimization

Introduction

- Once we understand what a program means, we may replace it with a semantically identical, but more efficient program.
- Goal: Code optimization to decrease runtime and reduce required resources
- A simple example is constant folding:
 - If some expression always evaluates to the exact same value, we can do the evaluation at compile time and replace the code for the expression with its result.

Optimization

Example

```
temp1 := inttoreal(40)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1    := temp3
```

Code optimization

```
temp1 := id3 * 40.0
id1    := id2 + temp1
```


Code generation

Introduction

The last step is to convert the IR into a form a machine can run.

- Machine code:

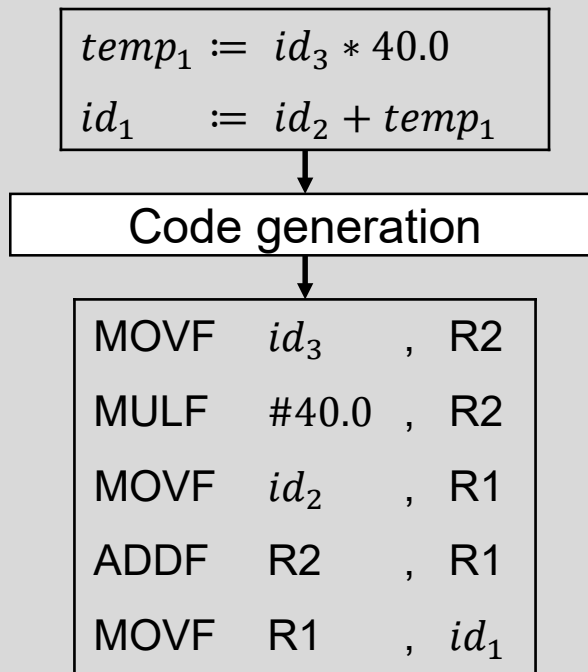
- Comprises instructions for a real CPU.
- We get an executable that the OS can load directly onto the chip.
- Fast, but ties the compiler to a specific architecture (e.g., x86 or ARM).

- Bytecode

- Comprises instructions for a Virtual Machine (VM).
- VM is a hypothetical, idealized machine which runs on the target OS.
- Originally called p-code (“p” like portable), but today, we generally call it bytecode because each instruction is often a single byte long.

Code generation

Example



Runtime

We usually need some services that our language provides while the program is running.

- Examples:
 - If the language automatically manages memory, we need a garbage collector.
 - If our language supports “instance of” tests, we need some representation to keep track of the type of each object during execution.

In a fully compiled language, the code implementing the runtime gets inserted directly into the resulting executable.

If compiled programs run inside an interpreter or VM, then the runtime lives there.

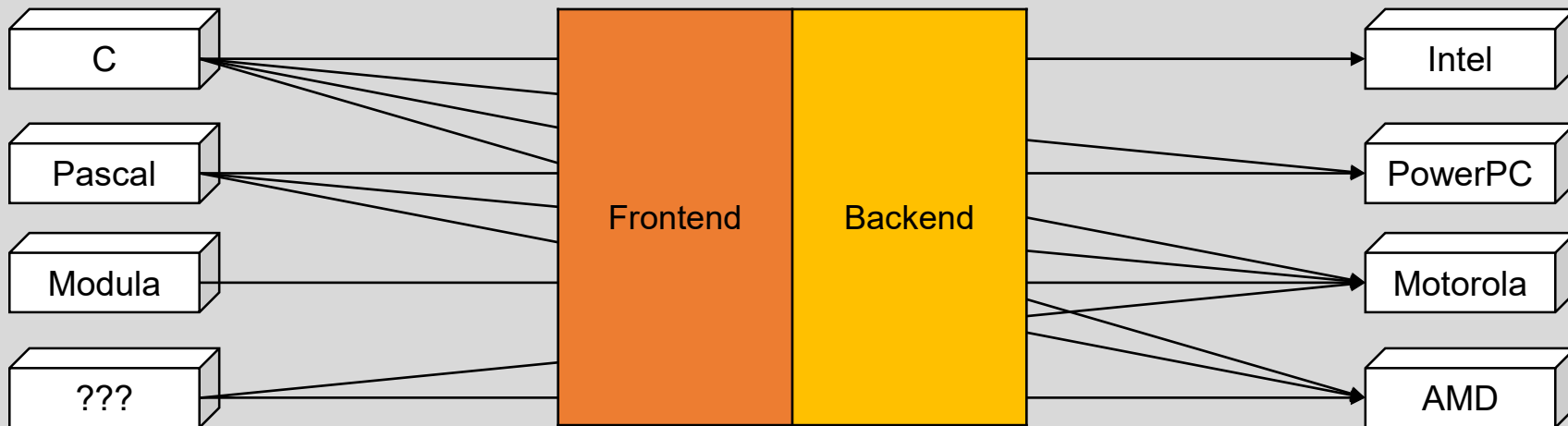
Compiler architecture

Desired qualities

- Creation of correct target code.
- The created program should be efficient.
- Consistent and predictable optimization.
- The compilation itself should be efficient.
- Translation time ~ program size
- Good error diagnostics.
- Good integration with the debugger.

Compiler architecture

Styles

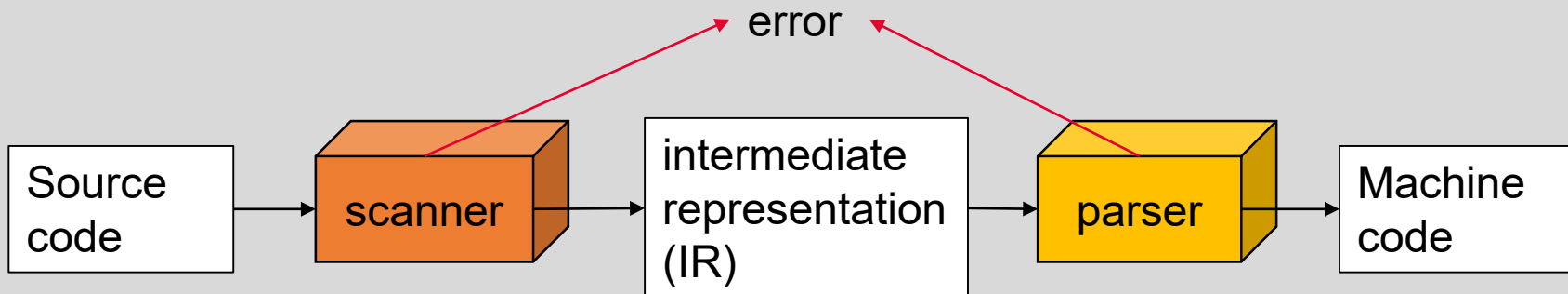


- Can we implement a $n \times m$ compiler with $n + m$ components?
 - All properties of the programming language must be encapsulated in the frontend.
 - All properties of the machine language must be encapsulated in the backend.
 - Intermediate representations must be able to cover various concepts of the programming languages and machine languages.

Compiler architecture

Traditionally: Two-stage compiler

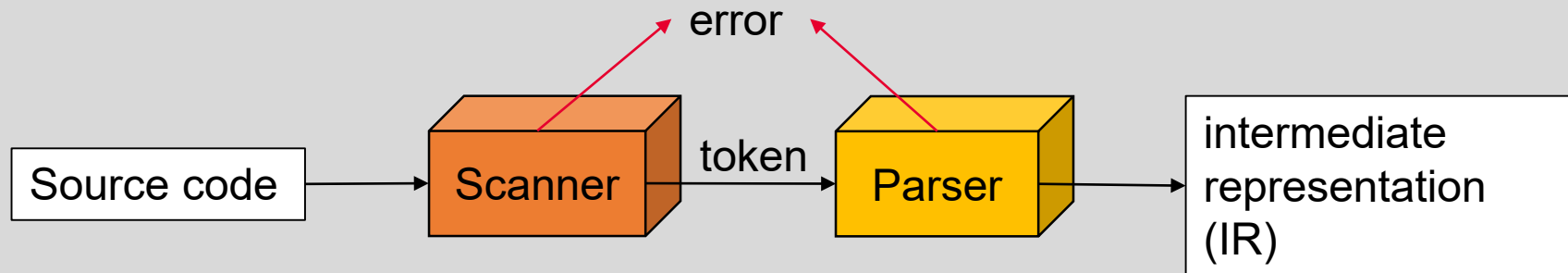
- Frontend creates a correct IR from source code.
- Backend creates correct machine code from the IR.
- Support of multiple source languages and machine languages.
- Multiple passes normally generate better code.



Compiler architecture

Tasks of the frontend

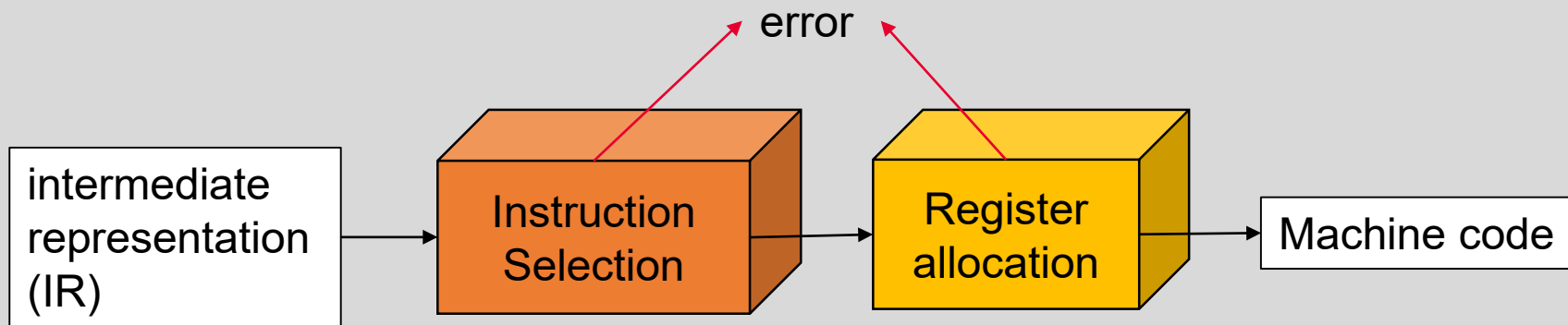
- Recognition of correct programs.
- Error detection and error handling.
- Creation of an IR.



Compiler architecture

Tasks of the backend

- Translation of the IR to machine code.
- Determination of the instructions for each operation in the IR.
- Determination which data is to be stored in the registers.

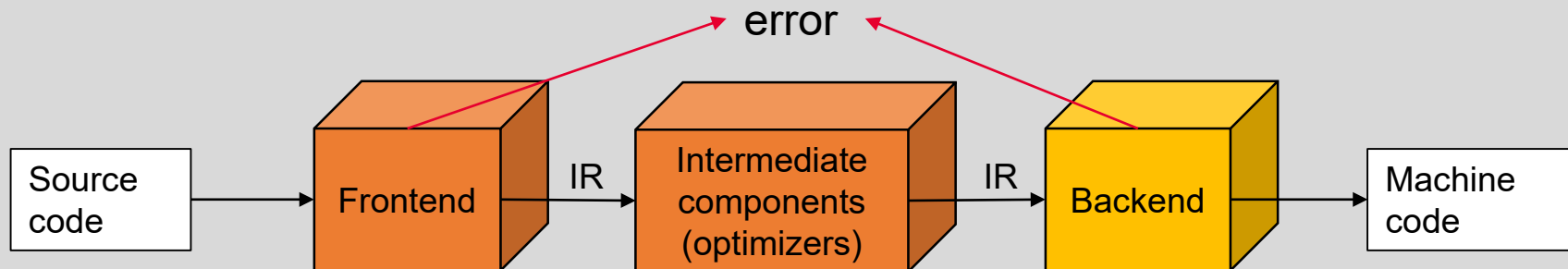


Compiler architecture

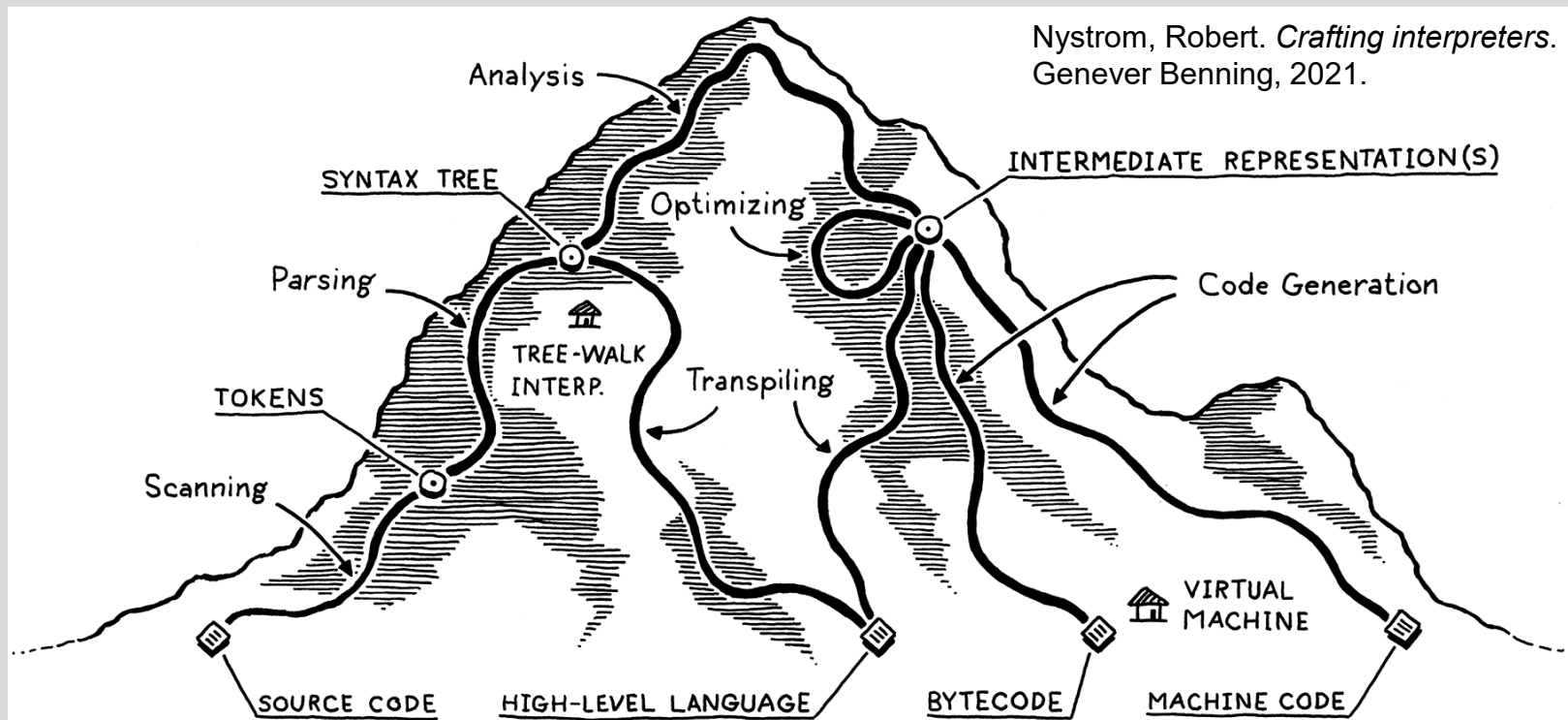
Today: Three-stage compiler

Multiple optimization phases:

- Declare and group constants.
- Reorganize code.
- Recognize partial expressions and eliminate them.
- Recognize redundancies and eliminate them.
- Recognize “dead code” and eliminate it.



Compiler architecture



Types of compilers

Single-Pass Compiler (Syntax-Directed Translation)

- Some simple compilers interleave parsing, analysis, and code generation.
- The code is produced directly while parsing, without ever creating any syntax trees or other IRs.
- Restricts the design of the language:
 - No intermediate data structures to store global information about the program,
 - No revisiting of any previously parsed part of the code.
 - => As soon as we see some expression, we need to know enough to correctly compile it.

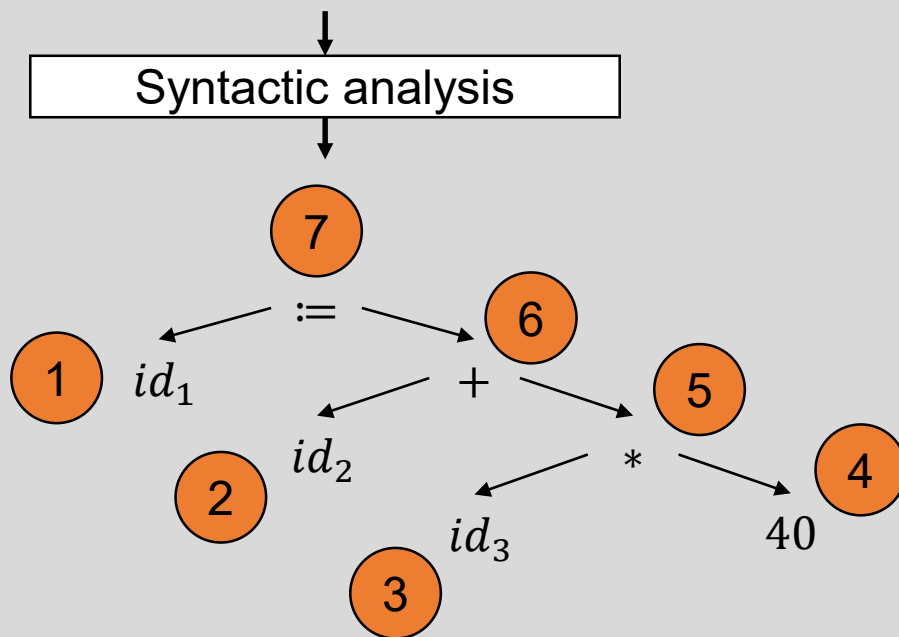
Types of compilers

Tree-Walk Interpreter

- Begin executing code right after parsing it to a syntax tree (with maybe a bit of static analysis applied).
- To run the program, the interpreter traverses the syntax tree, evaluating each node as it goes.
- Common for little languages, but not widely used for general-purpose languages since it tends to be slow.

Types of compilers

Tree-Walk Interpreter

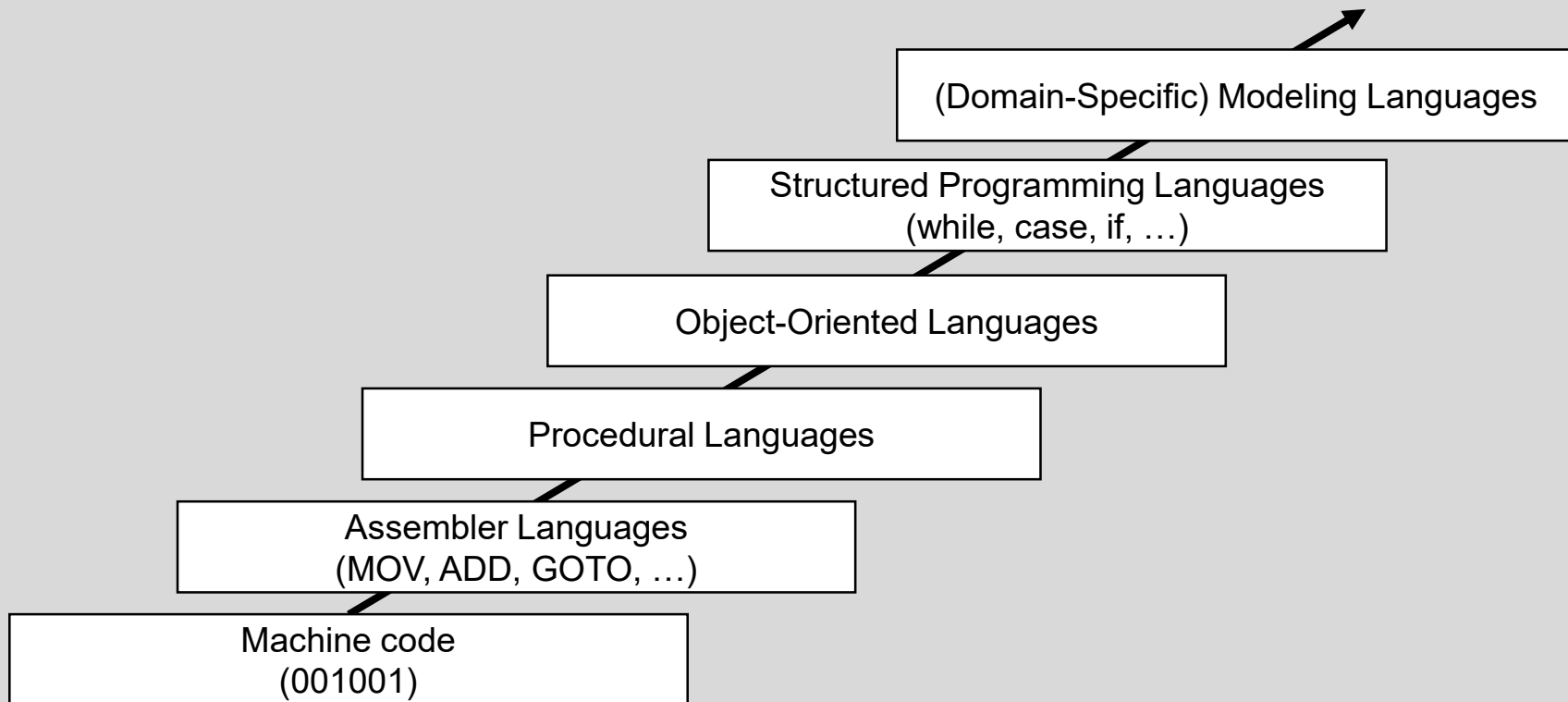


Types of compilers

Transpilers (Source-to-Source Compiler or Transcompiler)

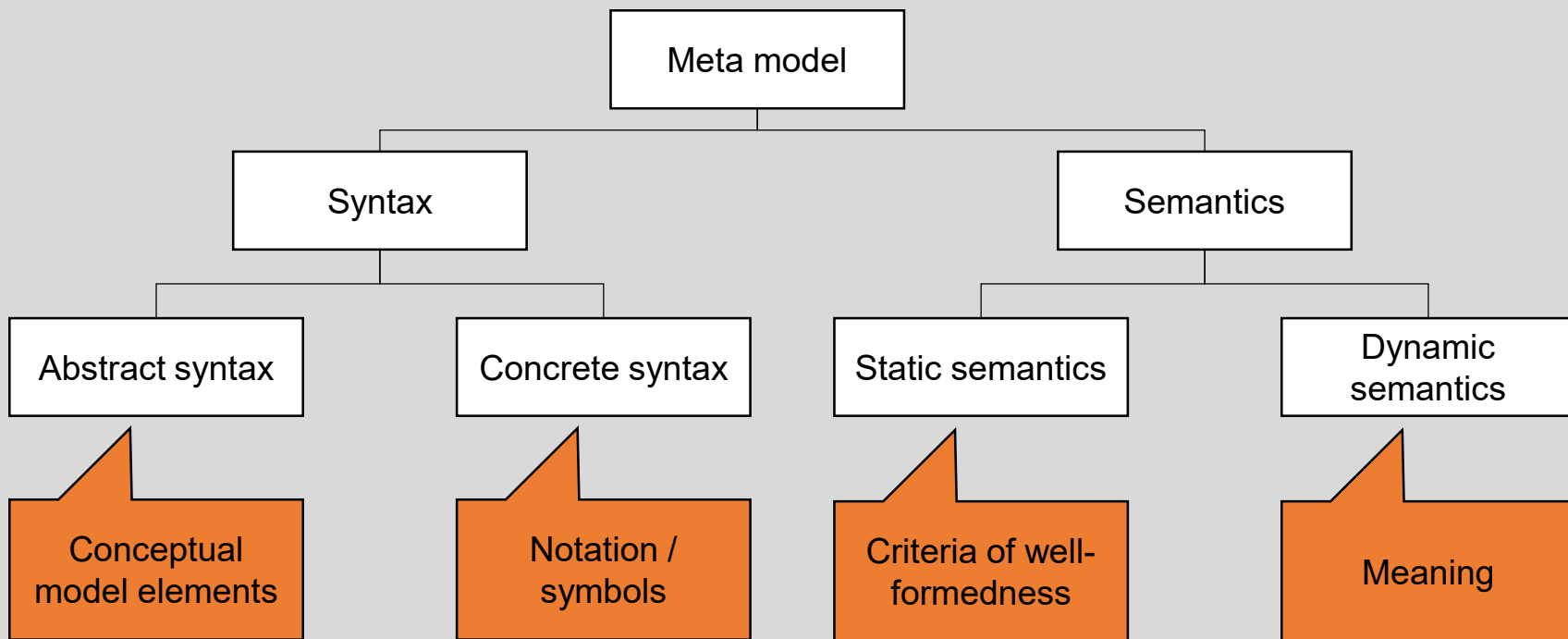
- In the backend, instead of generating code in some primitive target language, we use some target language that is about as high level as ours.
- Then, we use the existing compilation tools for that language as our escape route off the mountain and down to something we can execute.
- There is a long tradition of transpilers that produce C as their output language.
- C compilers are available on all UNIX platforms the produce efficient code.
- Today, many transpilers produce JavaScript as their output language.
- Web browsers are the "machines" of today, and their "machine code" is JavaScript.

Graphical modeling languages



Graphical modeling languages

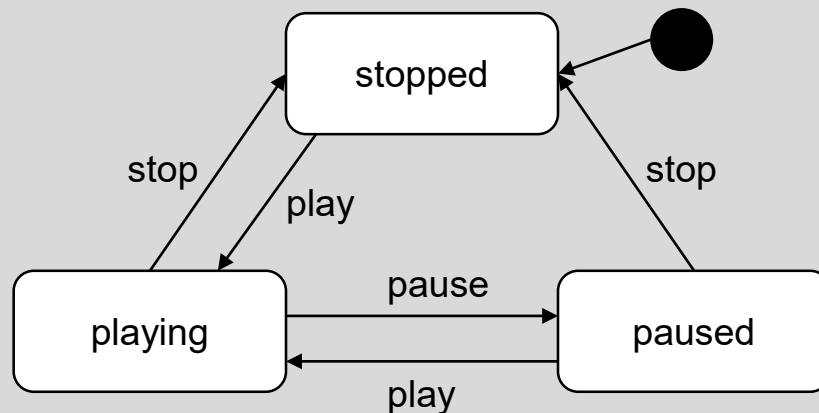
Definition



Graphical modeling languages

Example: Simple UML state machine

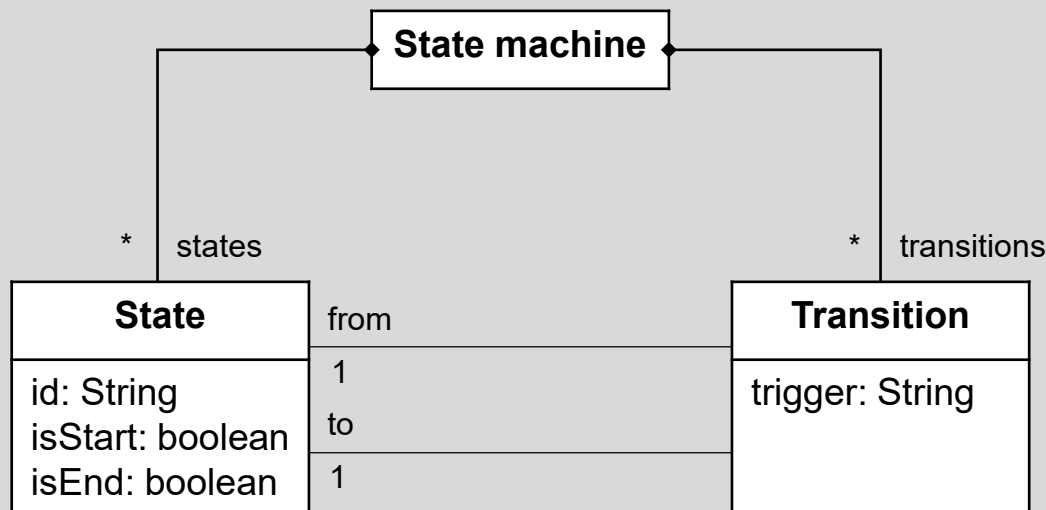
- We want to support the following basic concepts:
 - States (including a start and final state)
 - Transitions (including a trigger)
- Graphical example:



Graphical modeling languages





Abstract syntax

- We need the concepts used in the previous simple UML state machines (namely states and transitions).
- We use attributes to indicate start and final states.



Graphical modeling languages

Concrete syntax

Notation	Element
	State
	Start state
	Final state
	Transition

Graphical modeling languages

Static semantics

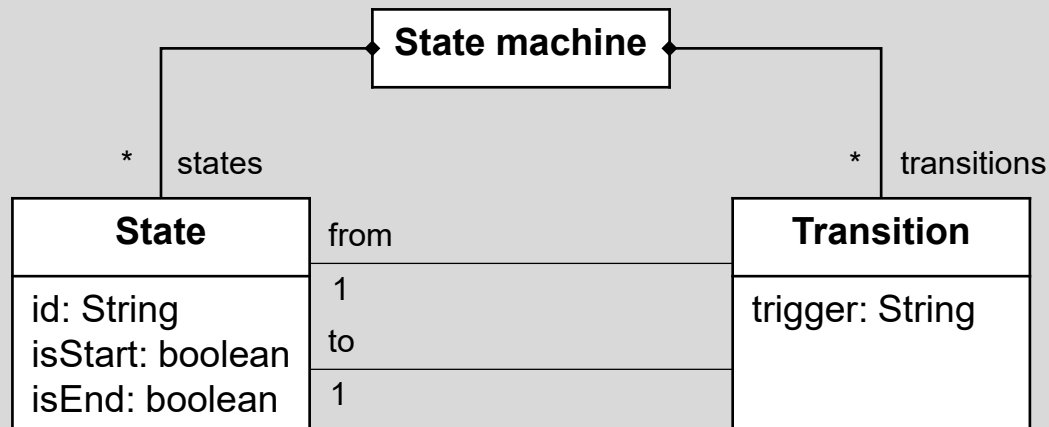
Criteria of well-formedness are conditions that may be imposed on a syntax, and which cannot be expressed by said syntax itself.

Examples:

- Every DFA needs at least one state.
- There must be exactly one state that fulfills `isStart == true`.
- Transitions must not be attributed with `NULL` or the empty string.
- ...

Graphical modeling languages

Static semantics

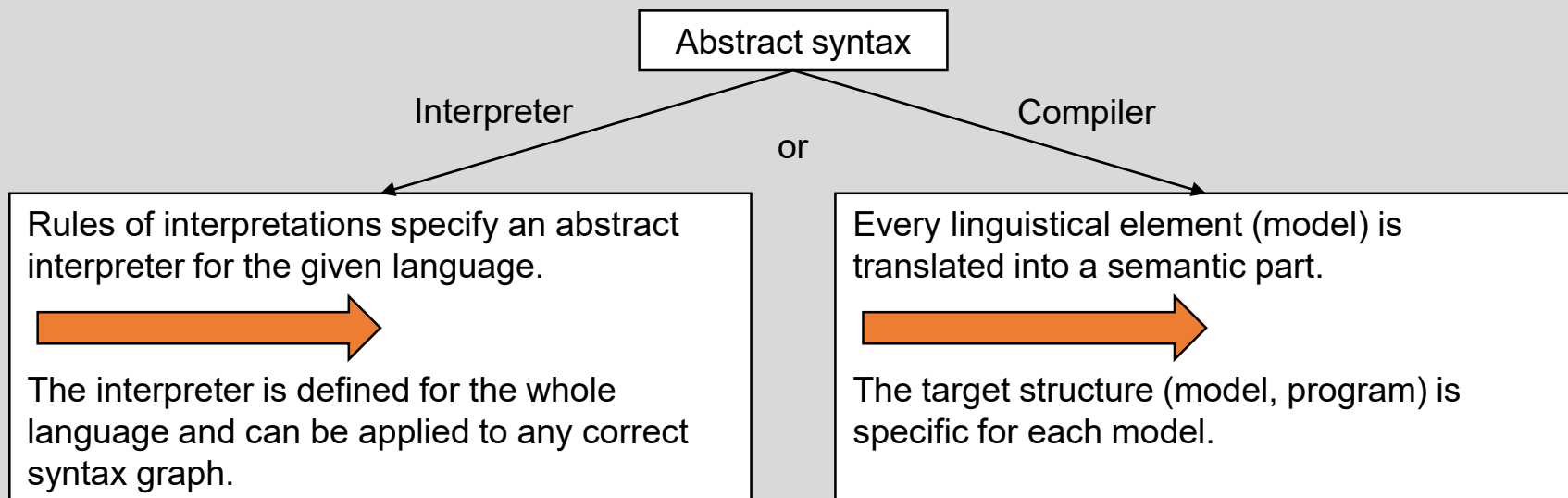


- Every state machine needs at least one state.
context state machine : `self.states->size() > 0`
- There must be exactly one state that fulfills `isStart == true`.
context state machine : `self.states->select(s | s.isStart)->size() = 1`
- Transitions must not be attributed with NULL or the empty string.
context state machine : `self.transitions->forAll(t | not t.trigger.isNull() and t.trigger <> "")`

Graphical modeling languages

Dynamic semantics

- In the context of model-driven engineering, this most often means either interpreter or compiler semantics.



Graphical modeling languages

Dynamic semantics

What is the “meaning” of a state machine?

- Behavioral state machine
 - States represent global system states
 - Transitions may be triggered by external events
- Protocol state machine
 - States represent object states.
 - Transitions may be triggered by method calls.

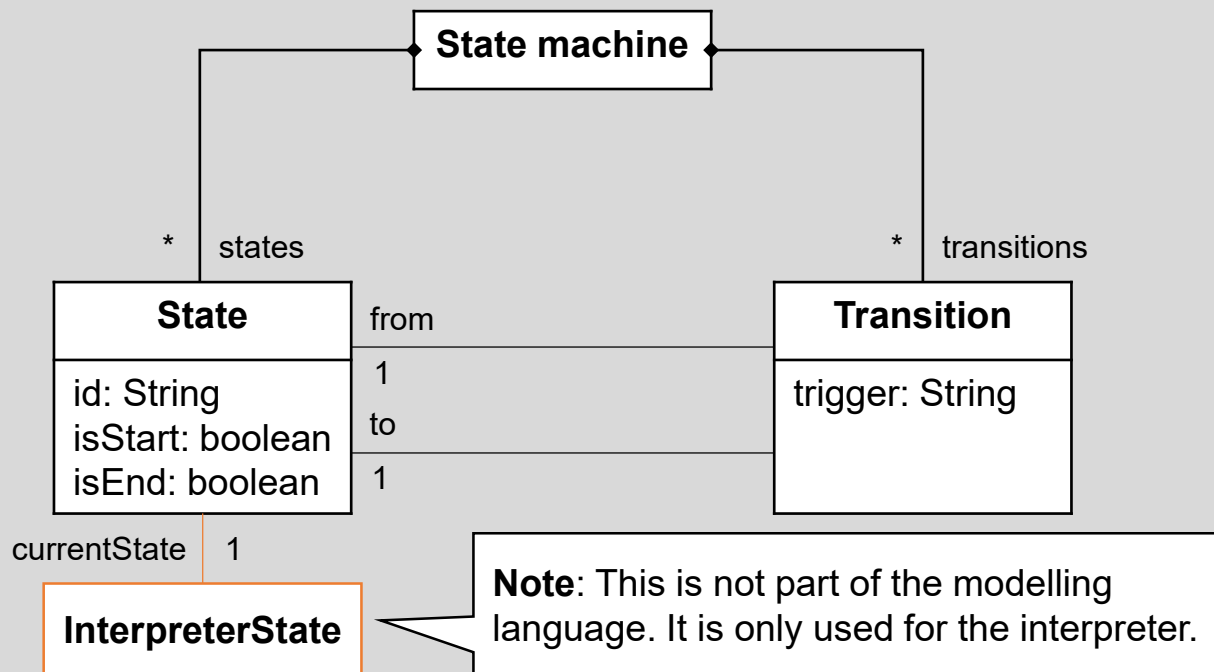
For this, we choose an
interpretative approach.

For this, we choose a
generative approach.

Graphical modeling languages

Dynamic semantics: Interpretative approach

- Trick: “Expansion” of the meta model to save the current state

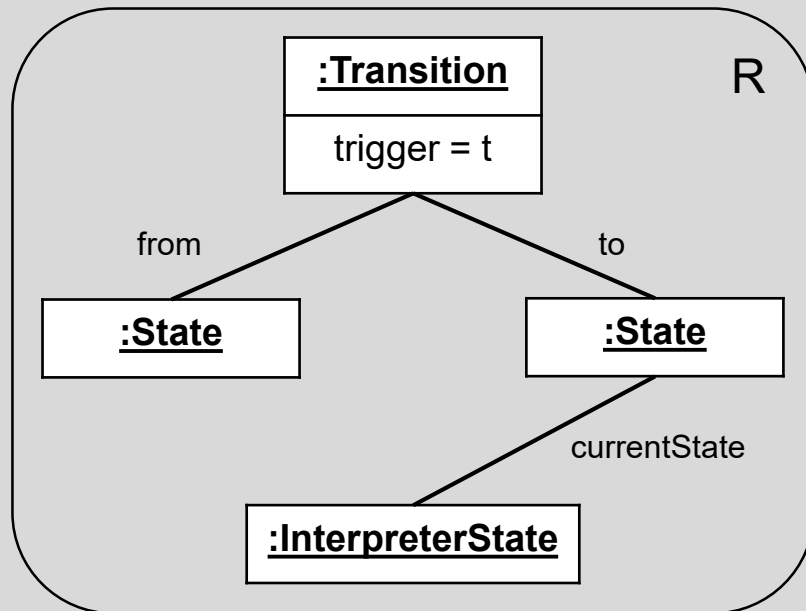
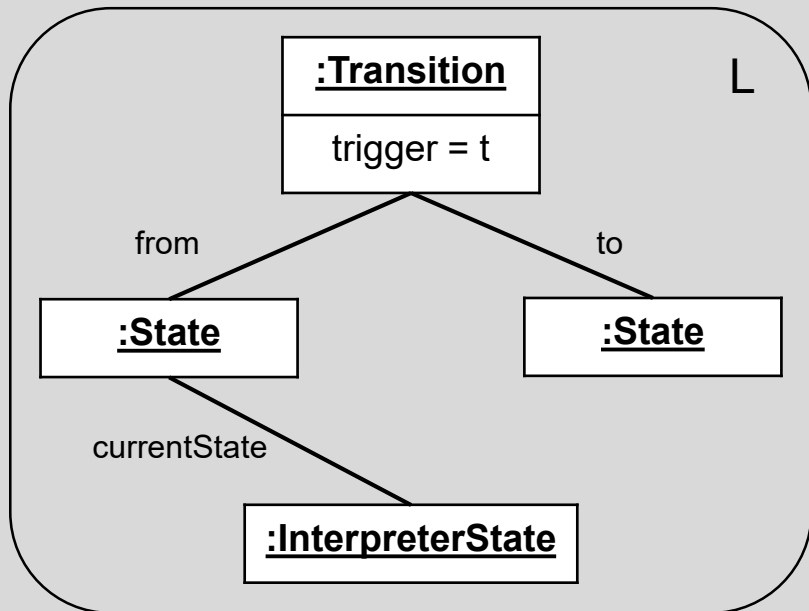


Graphical modeling languages

Dynamic semantics: Interpretative approach

- Treatment of trigger events as graph transformation rule

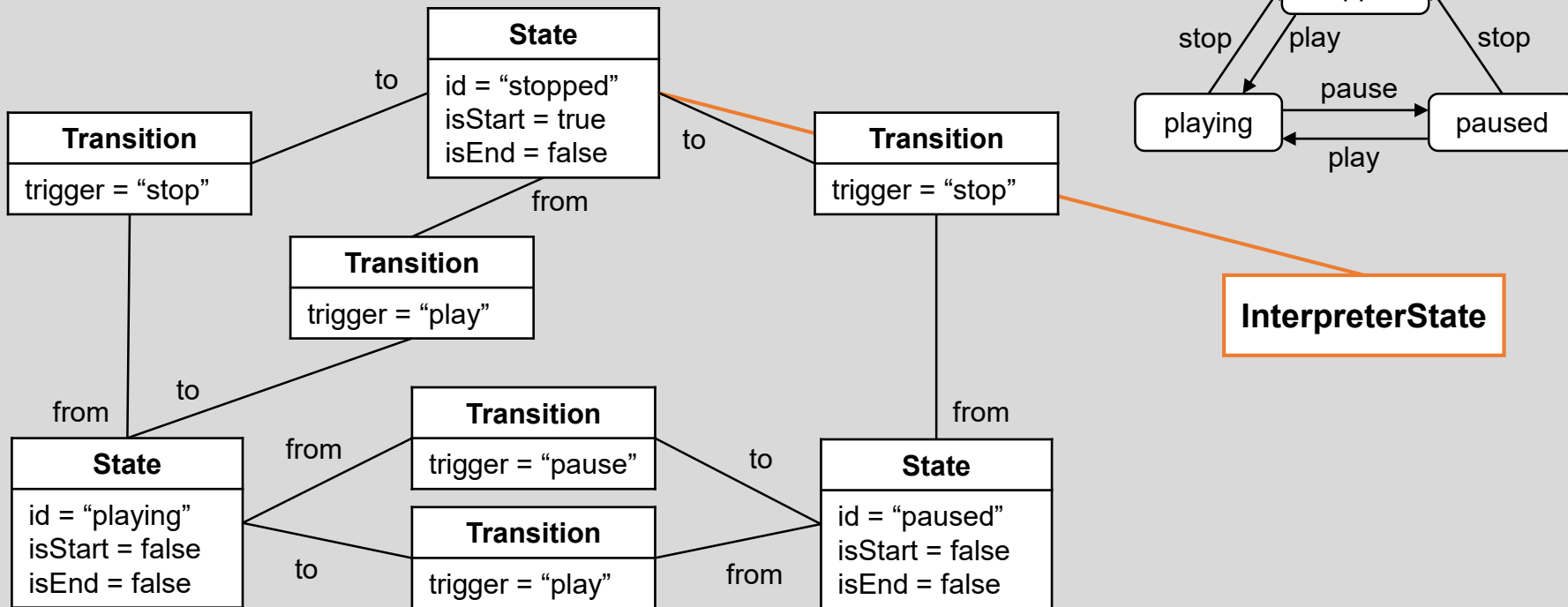
handleTriggerEvent (t:String)



Graphical modeling languages

Dynamic semantics: Interpretative approach

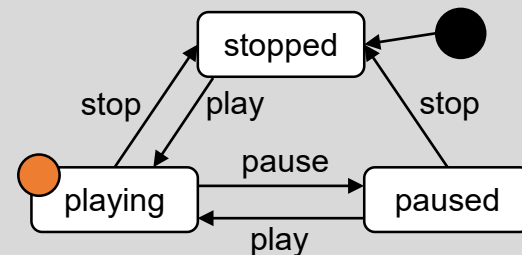
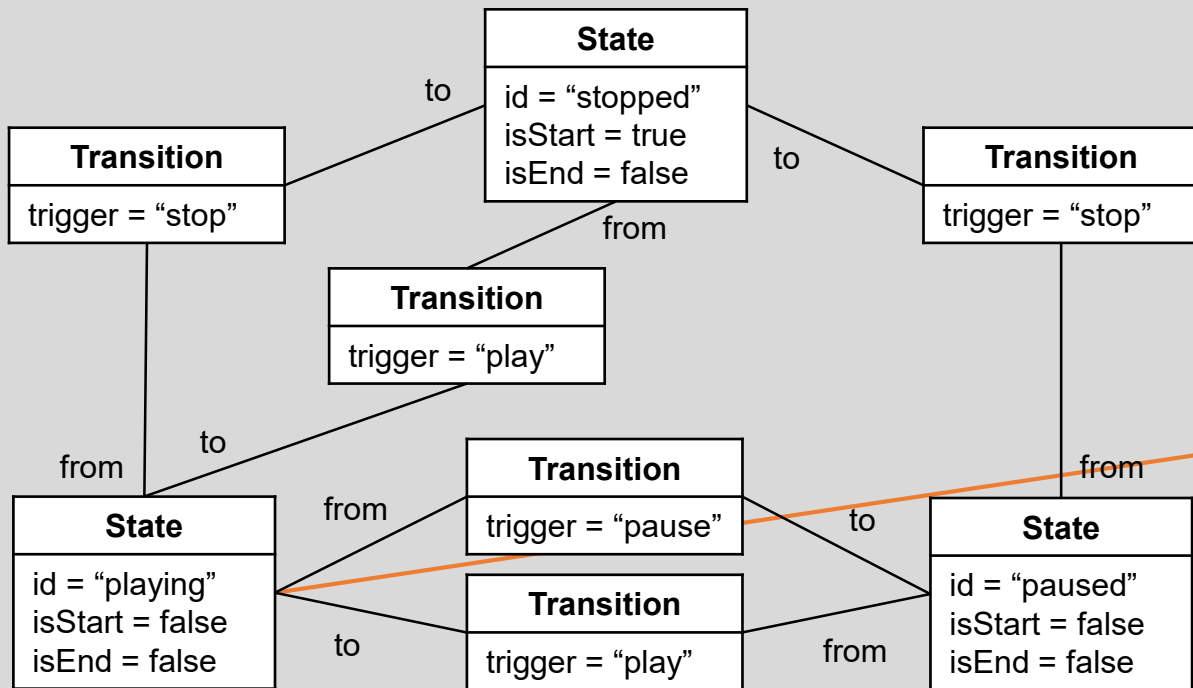
- Example: trigger sequence <play, pause, stop>



Graphical modeling languages

Dynamic semantics: Interpretative approach

- Example: trigger sequence <play, pause, stop>

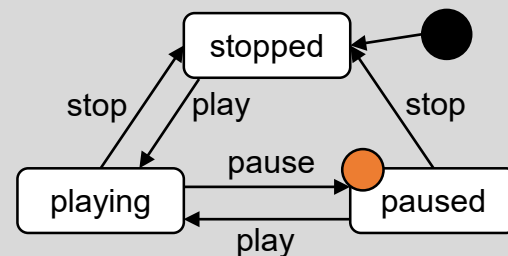
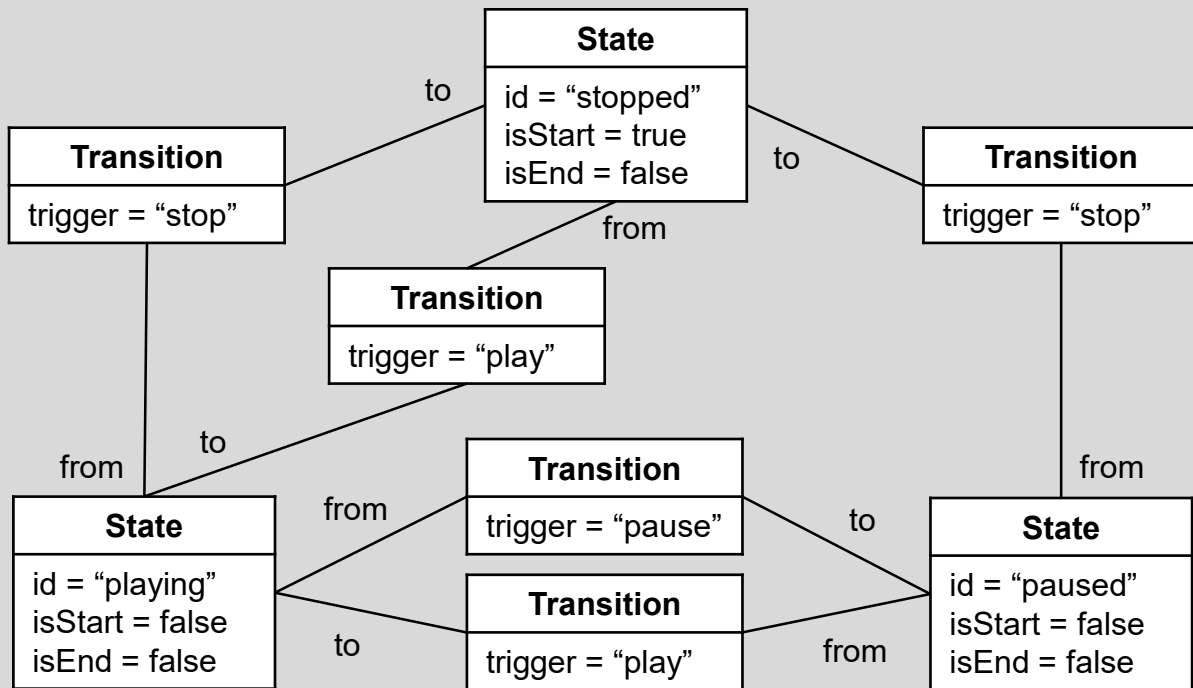


InterpreterState

Graphical modeling languages

Dynamic semantics: Interpretative approach

- Example: trigger sequence <play, pause, stop>

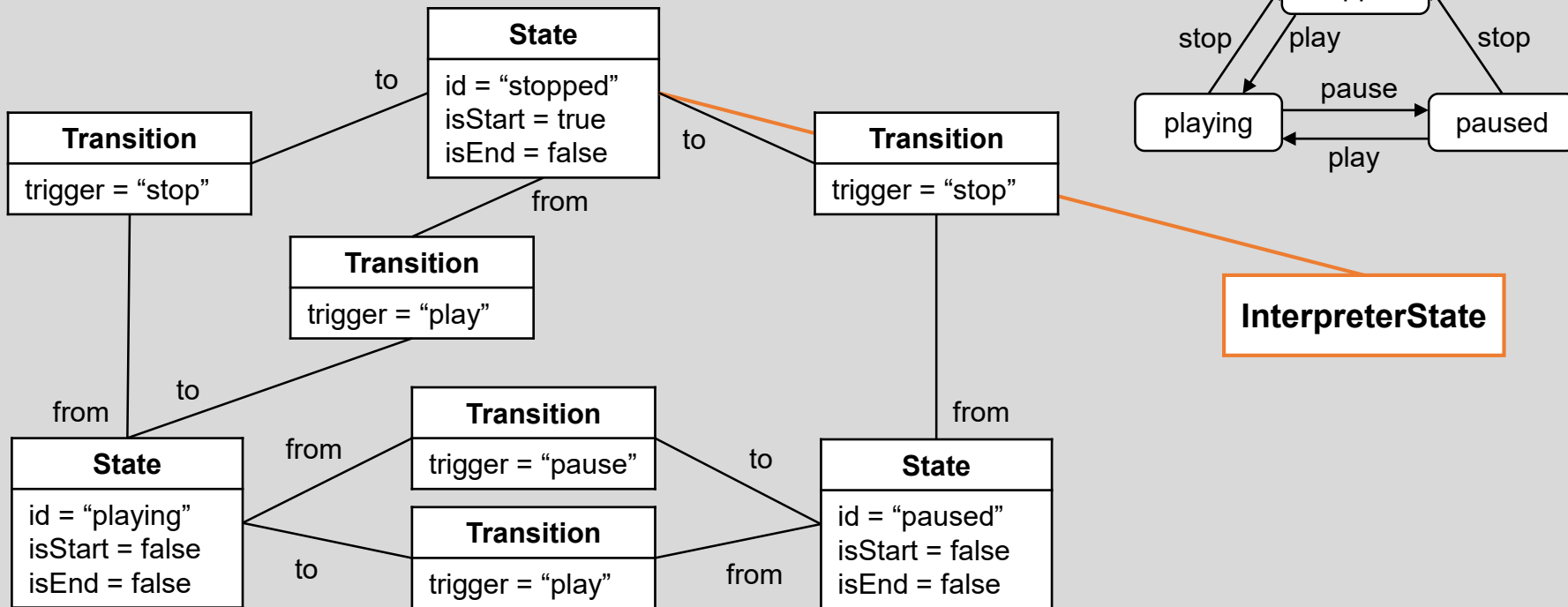


InterpreterState

Graphical modeling languages

Dynamic semantics: Interpretative approach

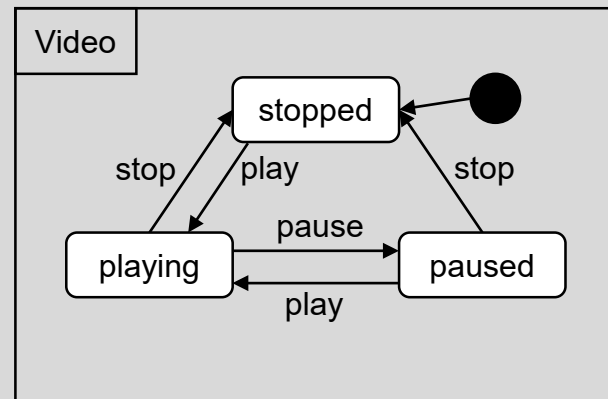
- Example: trigger sequence <play, pause, stop>



Graphical modeling languages

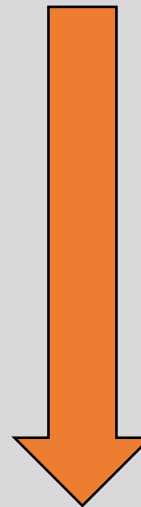
Dynamic semantics: Generative approach

```
public class Video {  
    public static final int PLAY = 0;  
    public static final int STOP = 1;  
    public static final int PAUSE = 2;  
    public static final int STOPPED = 0;  
    public static final int PLAYING = 1;  
    public static final int PAUSED = 2;  
    private int state = STOPPED;  
  
    public void play(){  
        switch(state){  
            case STOPPED:  
                state = PLAYING;  
                break;  
            case PAUSED:  
                state = PLAYING;  
                break;  
            default:  
                throw new RuntimeException("Unsupported Protocol!");  
        }  
    }  
  
    public void stop(){...}  
    public void pause(){...}  
}
```



Course overview

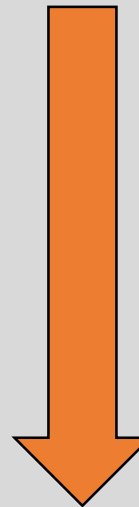
- Part I: Classical Compiler Construction
 - 02-Mar-23: Lexical Analysis: Handwritten Scanners
 - 09-Mar-23: Lexical Analysis: Scanner Generators
 - 16-Mar-23: Syntactic Analysis: Grammars and Syntax Trees
 - 23-Mar-23: Syntactic Analysis: Top-Down Parsing
 - 30-Mar-23: Syntactic Analysis: Parser Generators
 - 06-Apr-23: Semantic Analysis and Interpretation
 - 13-Apr-23: Spring Break
 - 20-Apr-23: Code Generation and Optimization
 - 27-Apr-23: PEGs, Packrats and Parser Combinators



In the exercises, we will develop a fully functional interpreter for our own programming language.

Course overview

- Part II: Modeling Language Engineering
 - 04-May-23: Metamodeling - Syntax and Static Semantics
 - 11-May-23: Dynamic Semantics: Rule-based Interpretation
 - 18-May-23: Holiday
 - 25-May-23: Dynamic Semantics: Model-2-Text Generation
 - 01-Jun-23: Q&A



In the exercises, we will develop a fully functional Petri net modeling environment.

Learning goals

- Architecture and tasks of a compiler
- Application of automata theory (finite automata, pushdown automata) to tackle challenges of compiler construction
- Concepts and techniques of parsing
- Semantic analysis (including type checking, dependence analysis)
- Basics of code generation (especially abstract machine code)
- Overview of optimization techniques
- Meta modelling and basic model transformation techniques
- Implementation of domain specific modelling languages and modelling environments

Why should I take this course?

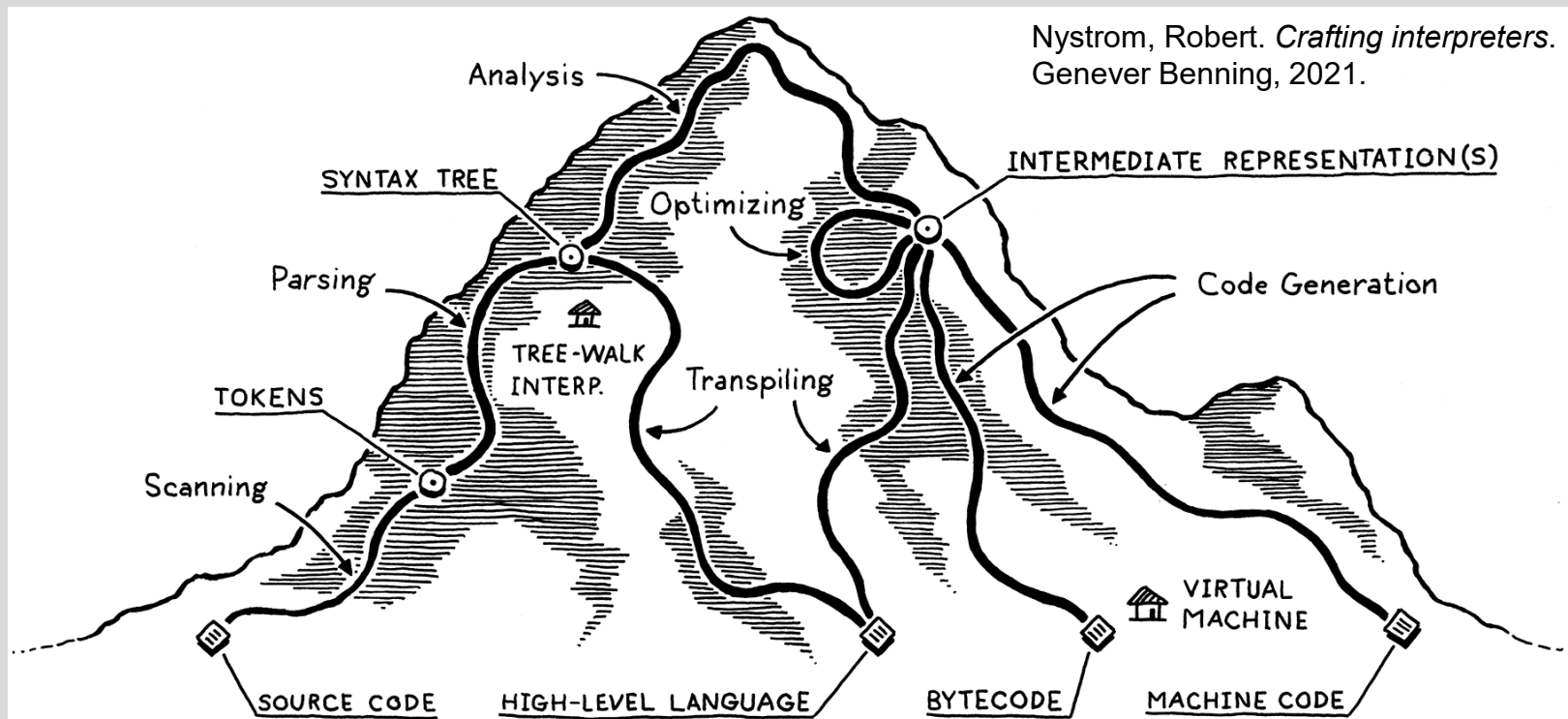
- The magic of computer languages
 - Ever wanted to make your own programming language or wondered how they are designed and built?
- Little languages are everywhere
 - There is a good chance you'll find yourself needing to whip up a parser or other tool when there is no existing library that fits your needs.
- Domain-specific languages + Model-driven development
 - Almost every large software project needs a handful of these.
 - Building sophisticated model-driven software engineering environments is the backbone of running these projects.

Languages are great exercise

- Implementing a language is a real test of programming skills.
- The code is complex and performance critical.
- You must master recursion, dynamic arrays, trees, graphs, and hash tables.
- Compiler construction uses many concepts and findings of various topics in computer science.

Data structures & algorithms	Graph algorithms
Theory	Finite automata / pushdown automata
Systems	Allocation / Synchronization
Architecture	Pipeline management / use of instructions sets
Artificial intelligence	Learning and search algorithms

Collect all the tools to climb the mountain

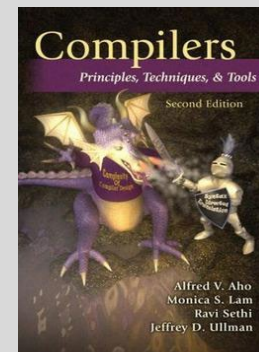


Course organization

- Lecturer: Timo Kehrer
- Assistants: Sandra Greiner, Manuel Ohrndorf
- Course materials: https://ilias.unibe.ch/goto_ilias3_unibe_crs_2469304.html
- Lectures: Thursday 13:15 - 15:00 (UniBE, Hörraum 120, Hauptgebäude H4)
- Exercise hour: Thursday 15:00 - 16:00 (UniBE, Hörraum 120, Hauptgebäude H4)
- Stream/Podcast: Available through ILIAS course
- Language: English
- Exam: Oral exam of 20 minutes, details will follow (registration: Academia)
- Repetition: Spring 2024

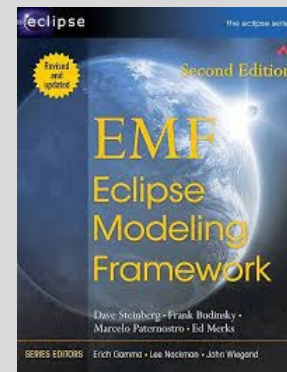
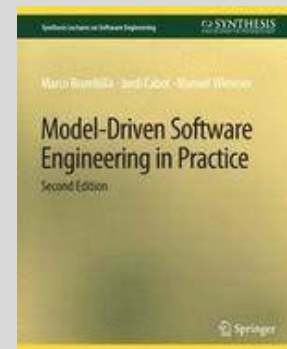
Recommended literature

- **Aho, Sethi, Ullman: Compilers: Principles, Techniques and Tools, Pearson Education, 2006**
 - Known as the Dragon Book to generations of computer scientists.
 - First published in 1986, it is widely regarded as the classic definitive compiler technology text.
- **Andrew, W. Appel, and Palsberg Jens. "Modern compiler implementation in Java." ISBN 0-521-58388-8. Cambridge University Press, 2002.**
 - First part describes classical compiler techniques in a way more amenable to Java programmers.
 - Second part includes the compilation of object-oriented and functional languages, which we will not cover in this course.



Recommended literature

- **Brambilla, M., Cabot, J., & Wimmer, M. (2017). Model-driven software engineering in practice. Morgan & Claypool Publishers.**
 - Second part deals with the technical aspects, spanning from the basics on when and how to build a domain-specific modeling language, to the description of Model-to-Text and Model-to-Model transformations.
- **Steinberg, D., Budinsky, F., Merks, E., & Paternostro, M. (2008). EMF: eclipse modeling framework. Pearson Education.**
 - Practical introduction into the very basics of working with the Eclipse Modeling Framework.



Next time

Lexical Analysis