EDAN65: Compilers

Introduction and Overview

Görel Hedin

Revised: 2022-08-29

Instructors

Lectures

- Prof. Görel Hedin

Programming assignments and lab sessions

- Jesper Öqvist
- Idriss Riouak
- Anton Risberg Alaküla
- Alfred Åkesson

Course registration

Confirm that you will take the course by signing list

Register in LADOK this week

Sign up for lab session, by **Thursday, Sep 1**

Prerequisites

- Object-oriented programming, Java
  - Assignment 0 includes some details on Java to fresh up
- Simple algorithms and data structures (recursion, trees, lists, hash tables, …)

Course information

Web site: https://cs.lth.se/edan65

- read the Week by Week page to find out what to do each week.
- Lecture slides, readings, assignments, exercises, quizzes
- Material added continuously during the course
- No handouts – print yourself if you want it on paper

Textbook

- A. W. Appel, Jens Palsberg: Modern Compiler Implementation in Java, 2nd Edition, Cambridge University Press, 2002, ISBN: 0-521-82060-X
- Available as on-line e-book through Lund University
- Only part of the book is used. Covers only part of the course.

Moodle: Forum, online quizzes, info on extra help sessions.

Course structure

- **Lectures**, Mon 15-17, Tue 13-15 (+ Thu 13-15 week 1)
- **Assignment 0**, for freshing up on Unix, Java, and JUnit, and understanding the build system Gradle. Working with Git. Do on your own.
- **Assignment 1-6**. Mandatory.
  - **Work in pairs**. Form pairs in the break. Or use the forum.
  - **Heavy**. Lab sessions for getting approved and getting help if you are stuck.
  - Thu 8-10, 10-12, 13-15, 15-17. **Sign up by Thursday Sep 1**
  - Lab sessions start next week (but start this week on your work)
  - Assignments **prerequisite for doing exam**
  - **Make sure the TAs note your presence at the lab session.** If you get stuck, you can get your assignment approved the next week (if you were present at the ordinary session). Email Görel for exemption due to illness.
- **Lecture quizzes**
  - Do on your own. (In Moodle.)
- **Exercises**
  - Do on your own or with your partner. (Book exercises without solutions + separate exercises with solutions)
- **Exam** – sign up in advance through the LTH system
  - Exam: Tuesday, Oct 25, 2022, 14-19. Eden (downtown).
  - Retake: April 2023.

Working with the assignments

- Collaborating with your partner, git repos on coursegit.cs.lth.se (see moodle)
  - in the **same** repository (recommended)
    - sit together, alternate who commits, **commit often**
    - git history should show progression during solving the assignment
    - if you commit from the same account, switch which account you use for the next assignment
  - in **separate** repositories
    - looser collaboration, help each other out if needed, compare solutions to get new insights, sync before lab session
    - **commit often**, git history should show progression during solving the assignment
  - Important: you need hands-on experience from all parts - you should be able to run and explain all parts of the solution

- If you (and your partner) get stuck before your lab session
  - ask on the forum
    - you are encouraged to give answers to other students on the forum
(for general advice, small examples, but not full solutions)
  - use the extra online help session

- "Self-grading" web service:
  Drop in your compiler jar file and run our tests.
  Available for labs 2-6.

Estimated typical effort for assignments

A0: Unix, Java, Gradle, Git	1-4 hours	do on your own
A1: Scanning	5 hours	mandatory
A2: Parsing	15 hours	mandatory
A3: Visitors, aspects	12 hours	mandatory
A4: Semantic analysis	18 hours	mandatory
A5: Interpreter	15 hours	mandatory
A6: Code generator	12 hours	mandatory

Student representatives

- **Who?** 2-3 students. The D programme should be represented.
- **During the course**: listen to your peers. Give feedback to instructor if relevant.
- **After the course**: participate in discussion of course evaluation results (with instructor).

Why learn compiler construction?

Why learn compiler construction?

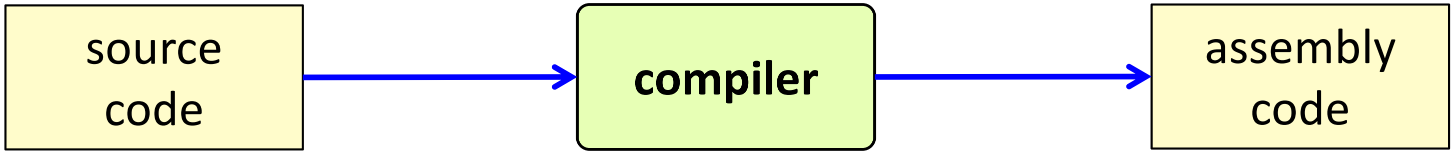
Languages are everywhere in software

- Understand how the language tools you use work (editors, compilers, transpilers, interpreters, ...)
- Many software projects use some kind of domain-specific language, e.g., for configuration or describing input.
- A compiler is just one example of a language tool. But it includes all the major techniques used for any software language processing.

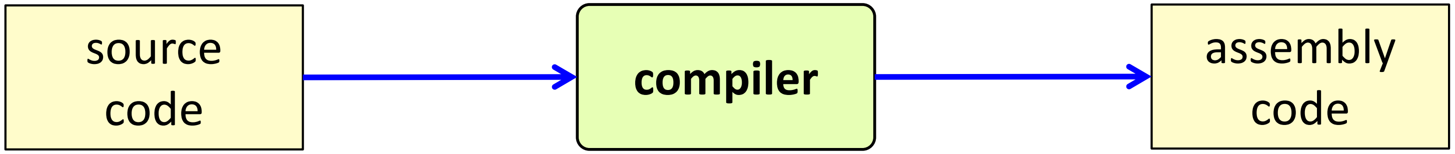
Fundamental theory and concepts

- Compiler theory: fundamental to computer science
- Essential for understanding programming languages

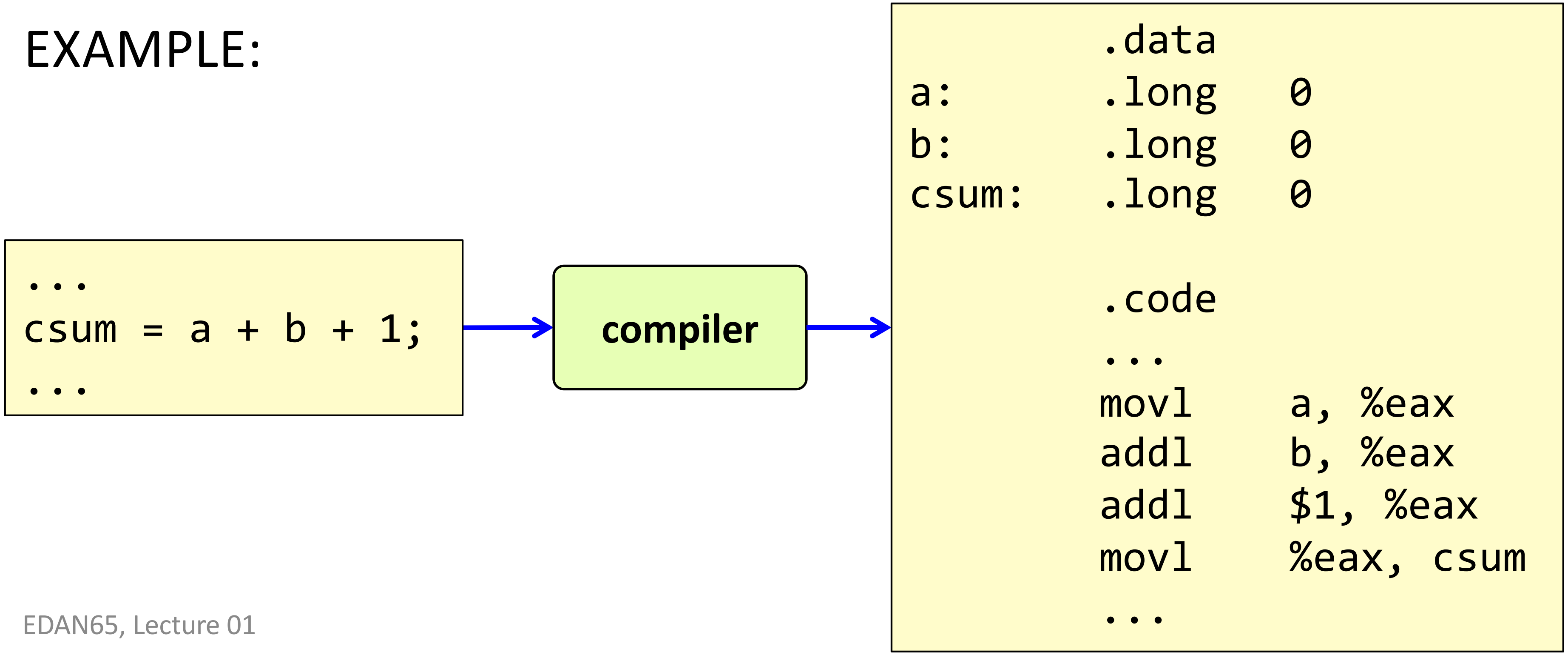
A traditional compiler



A traditional compiler



EXAMPLE:

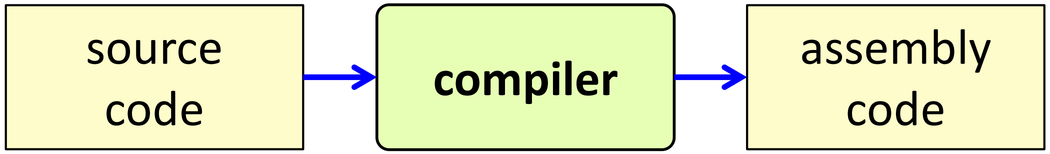


```
...
csum = a + b + 1;
...
```

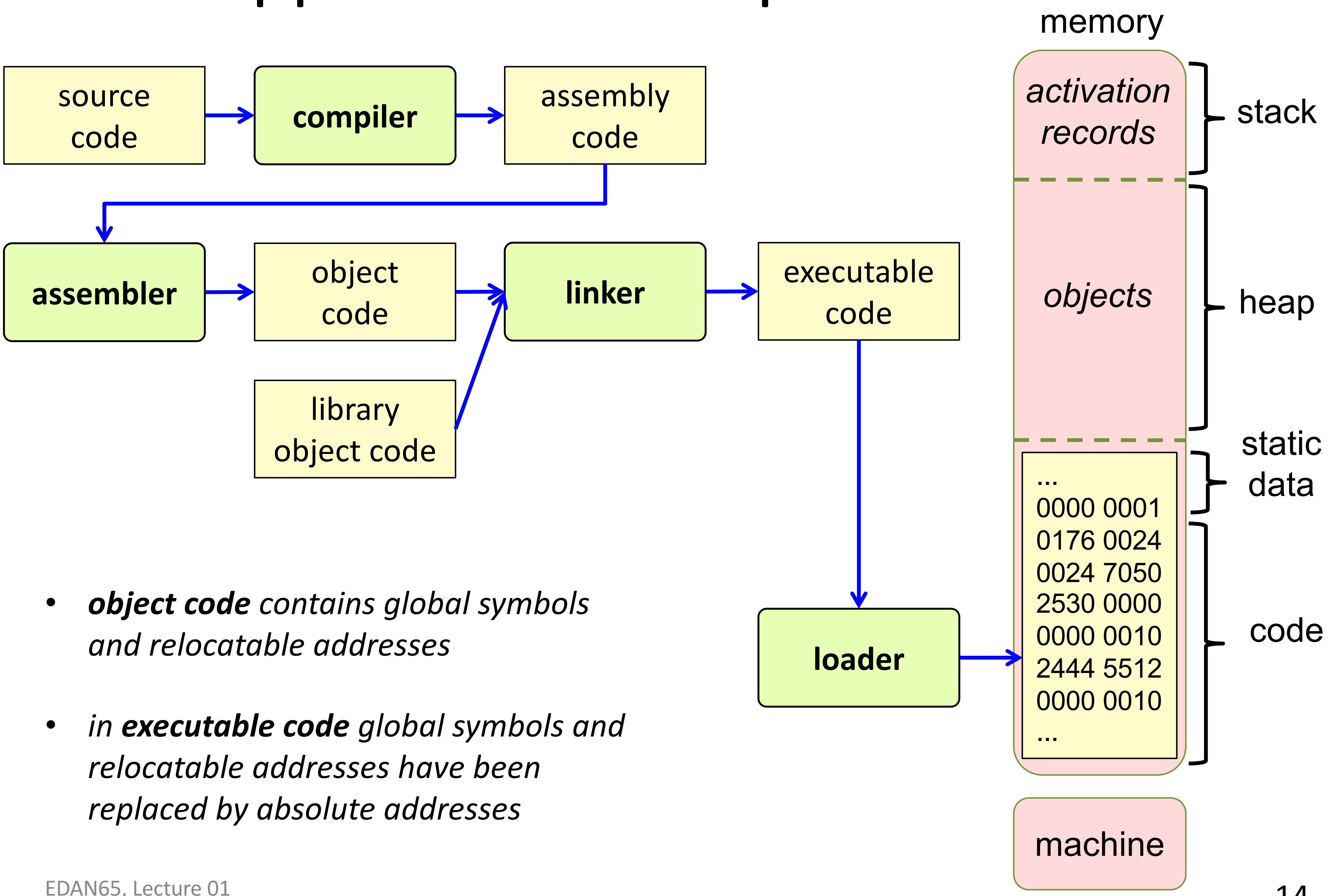
```
          .data
a:        .long   0
b:        .long   0
csum:     .long   0

          .code
          ...
          movl    a, %eax
          addl    b, %eax
          addl    $1, %eax
          movl    %eax, csum
          ...
```

What happens after compilation?



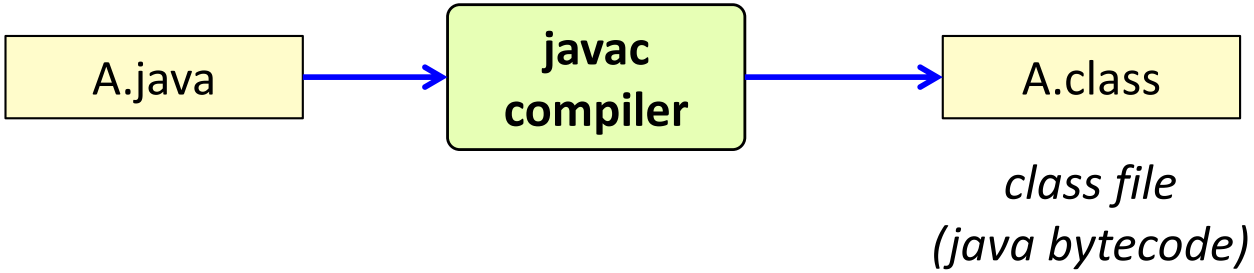
What happens after compilation?



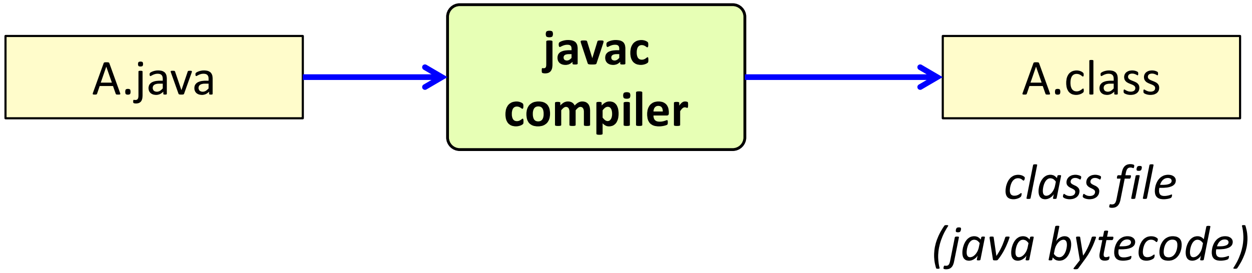
- ***object code*** *contains global symbols and relocatable addresses*
- *in* ***executable code*** *global symbols and relocatable addresses have been replaced by absolute addresses*

What about Java?

What about Java?



What about Java?



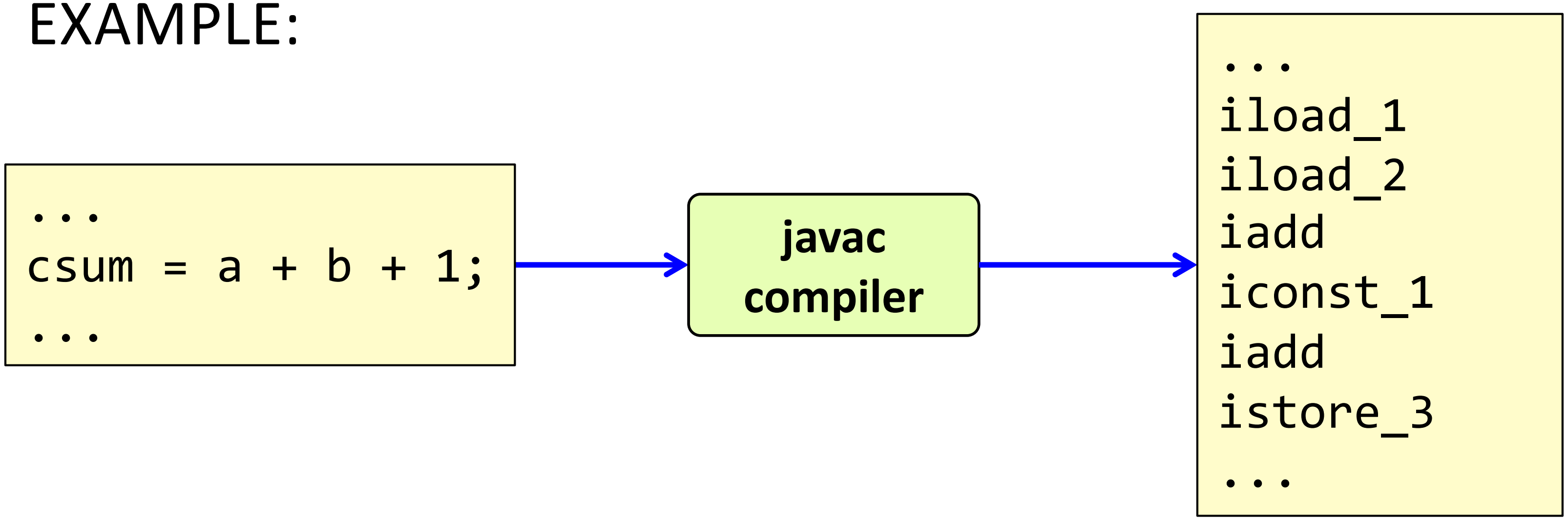
A.java

javac compiler

A.class

class file
(java bytecode)

EXAMPLE:

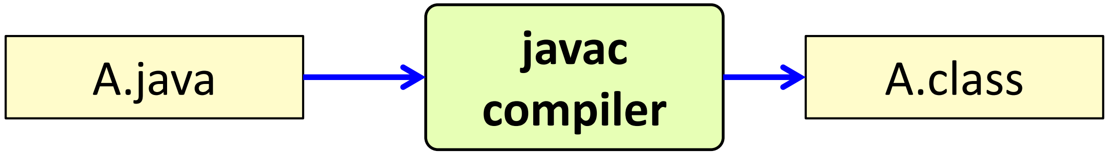


```
...
csum = a + b + 1;
...
```

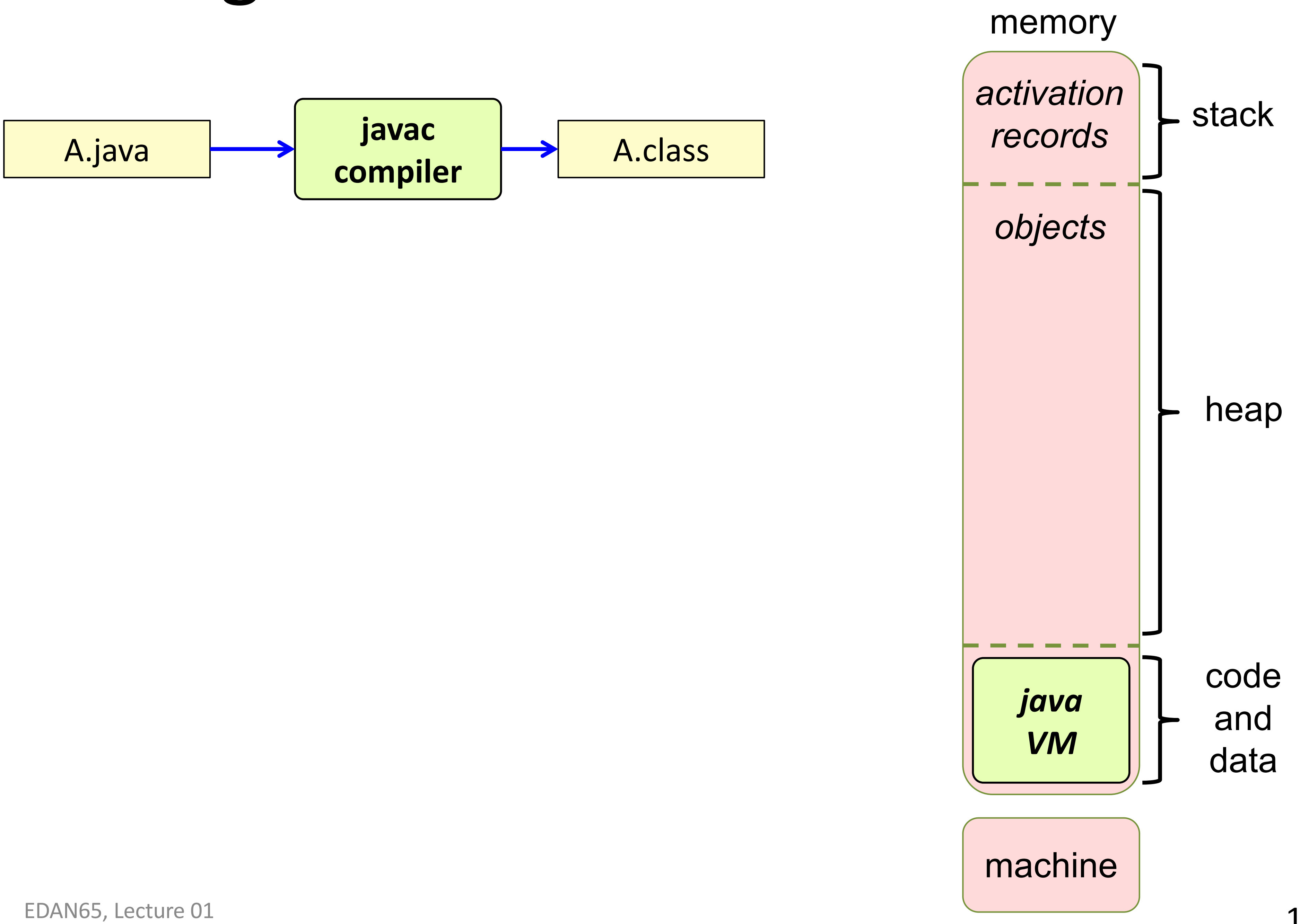
javac compiler

```
...
iload_1
iload_2
iadd
iconst_1
iadd
istore_3
...
```

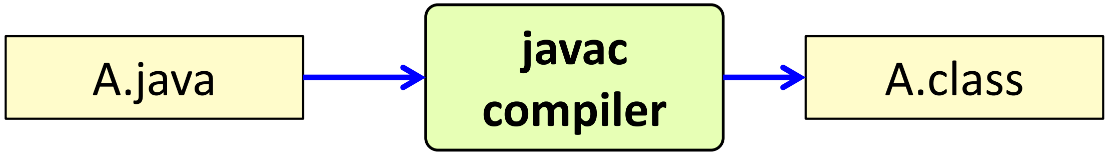
Running Java code?

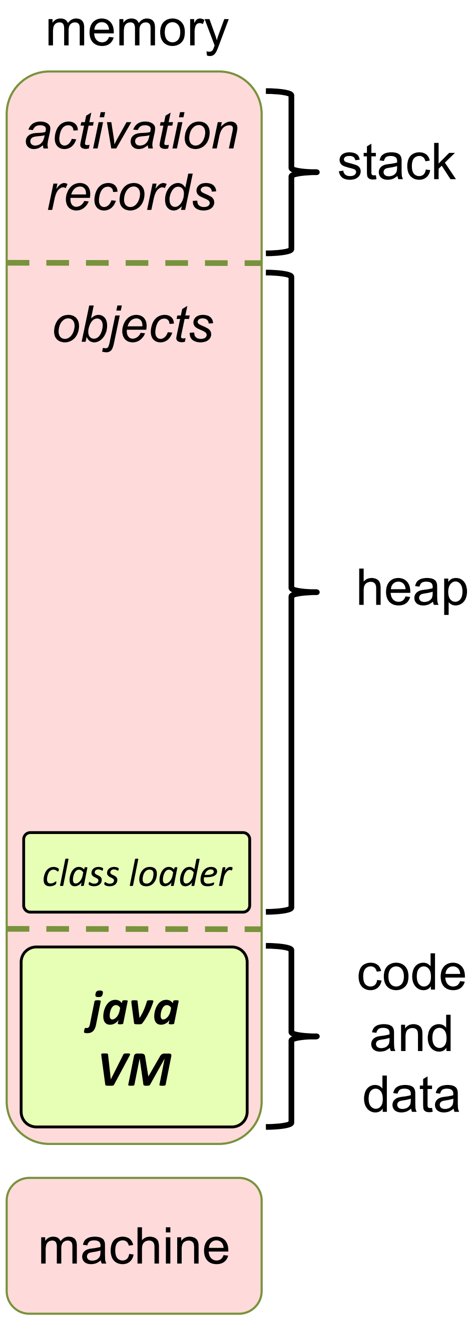


Running Java code?

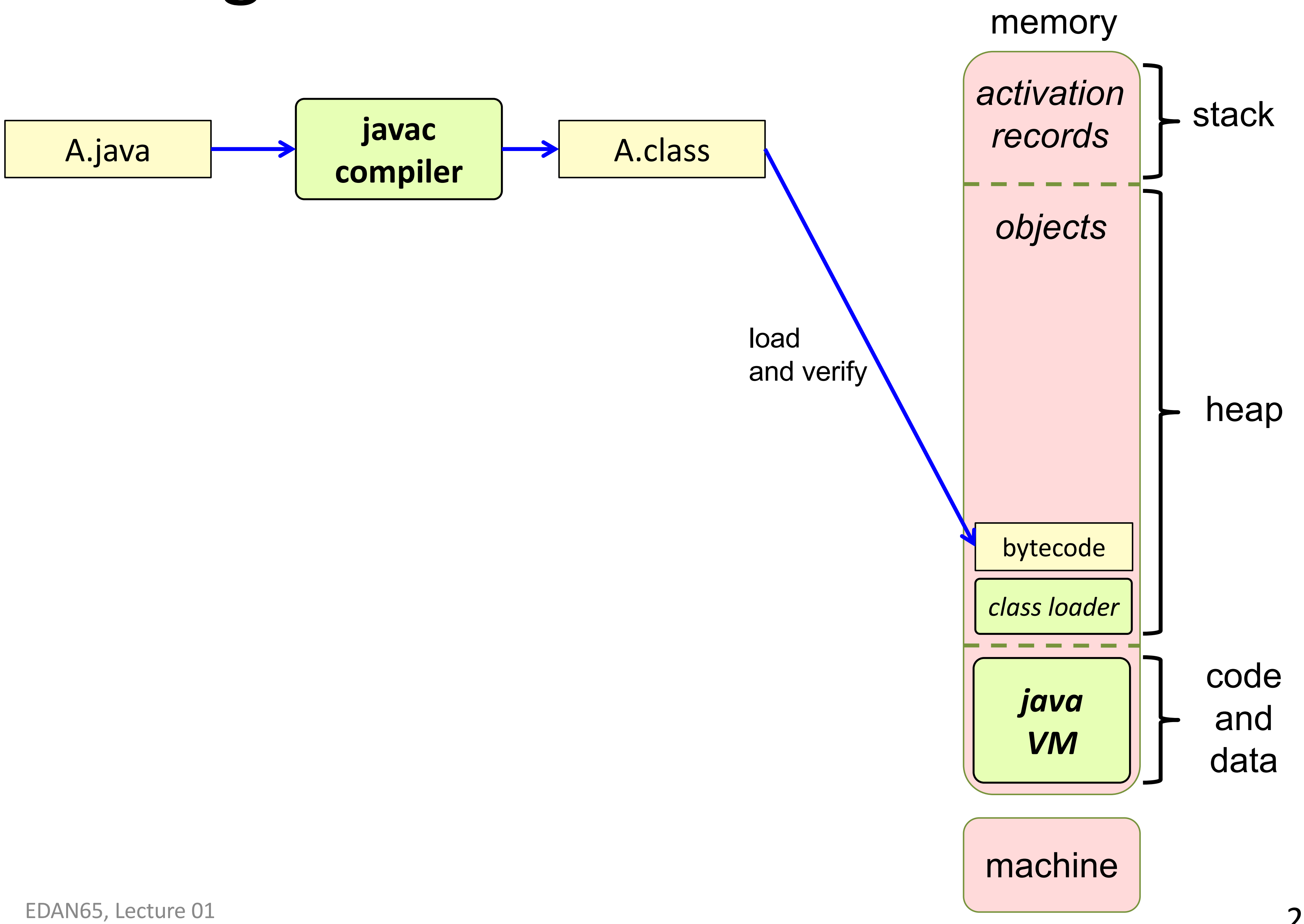


Running Java code?





Running Java code?



A.java

javac compiler

A.class

load and verify

memory

activation records

stack

objects

heap

bytecode

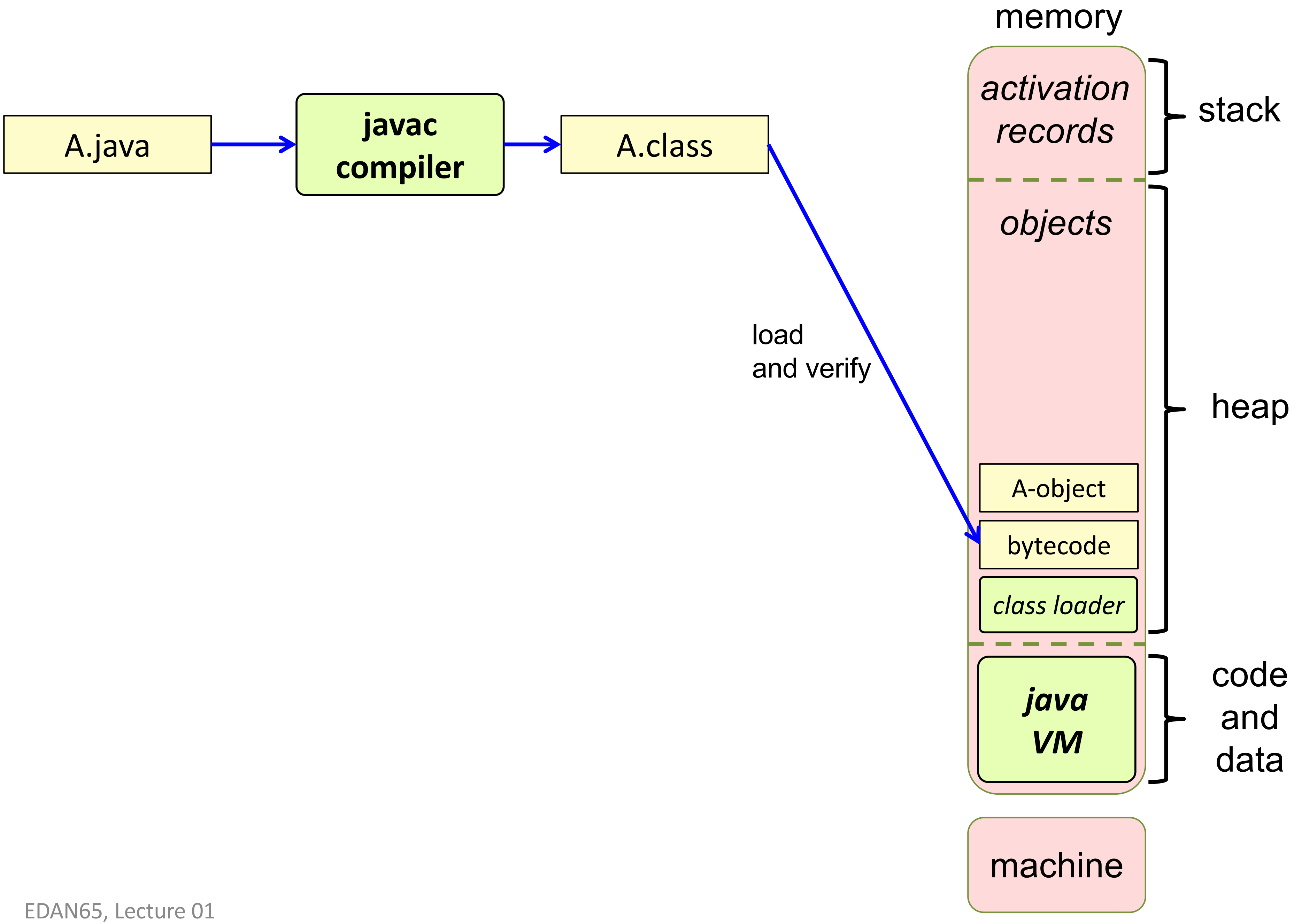
class loader

java VM

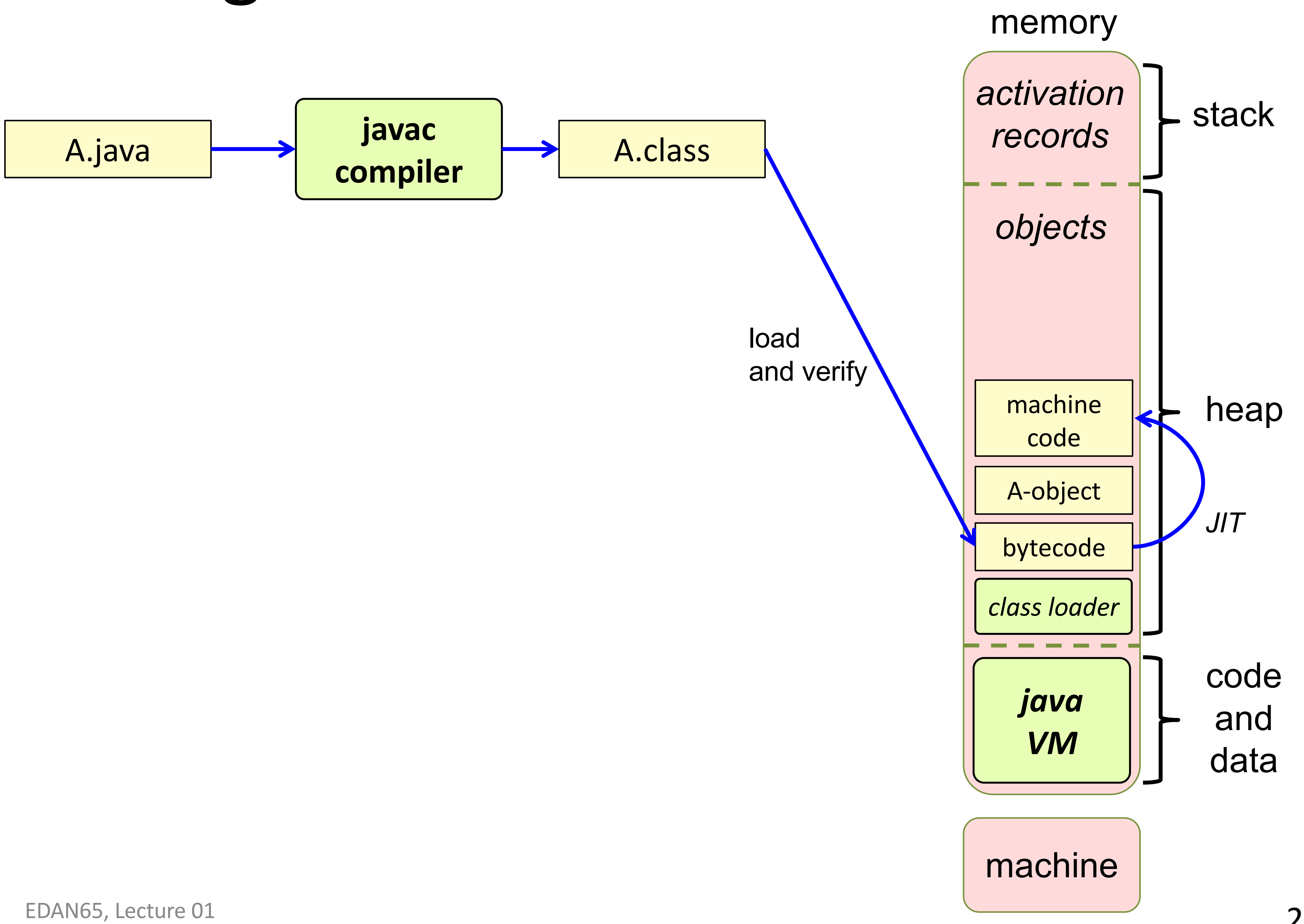
code and data

machine

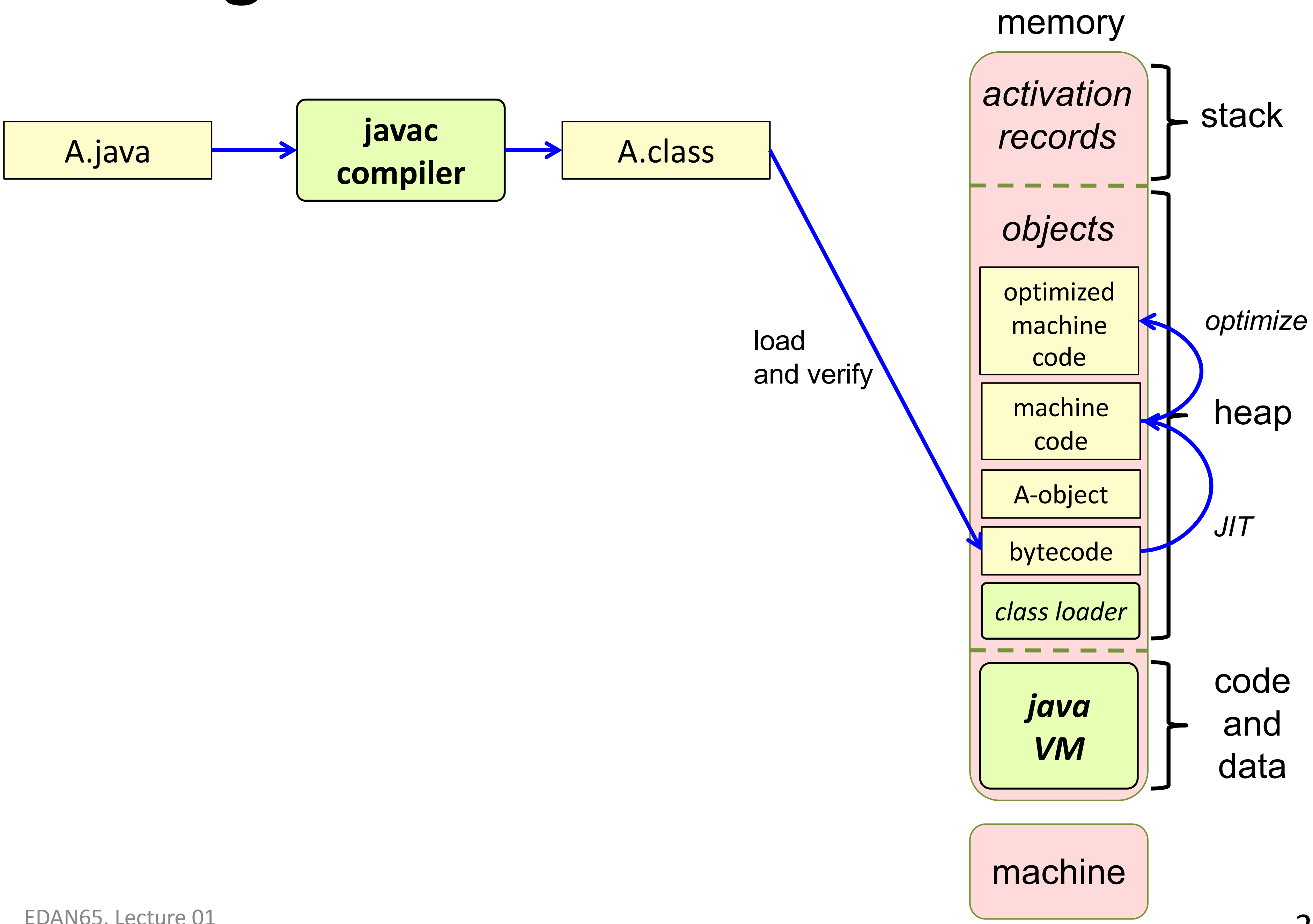
Running Java code?



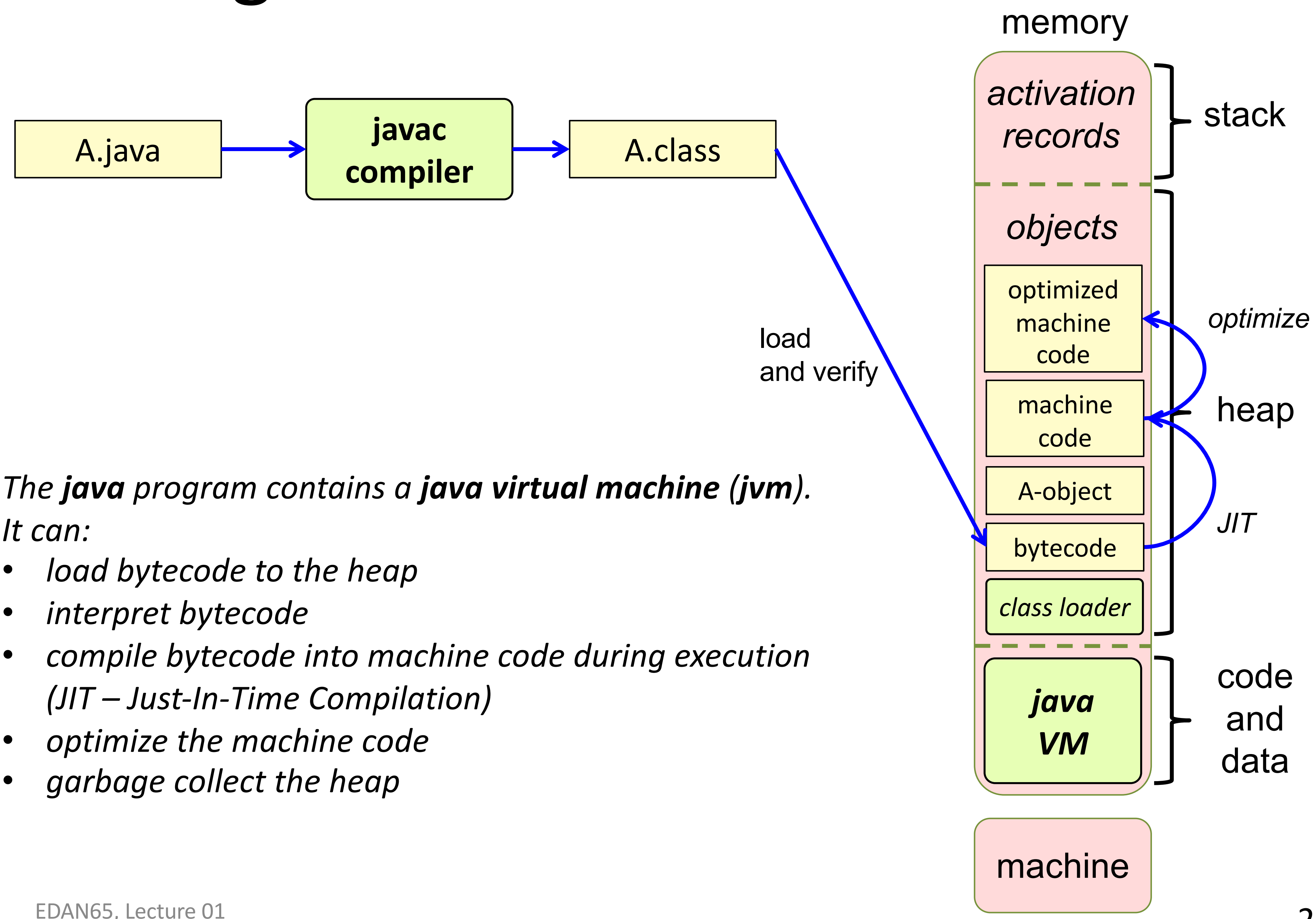
Running Java code?



Running Java code?



Running Java code?



*The **java** program contains a **java virtual machine** (**jvm**).*
It can:

- *load bytecode to the heap*
- *interpret bytecode*
- *compile bytecode into machine code during execution (JIT – Just-In-Time Compilation)*
- *optimize the machine code*
- *garbage collect the heap*

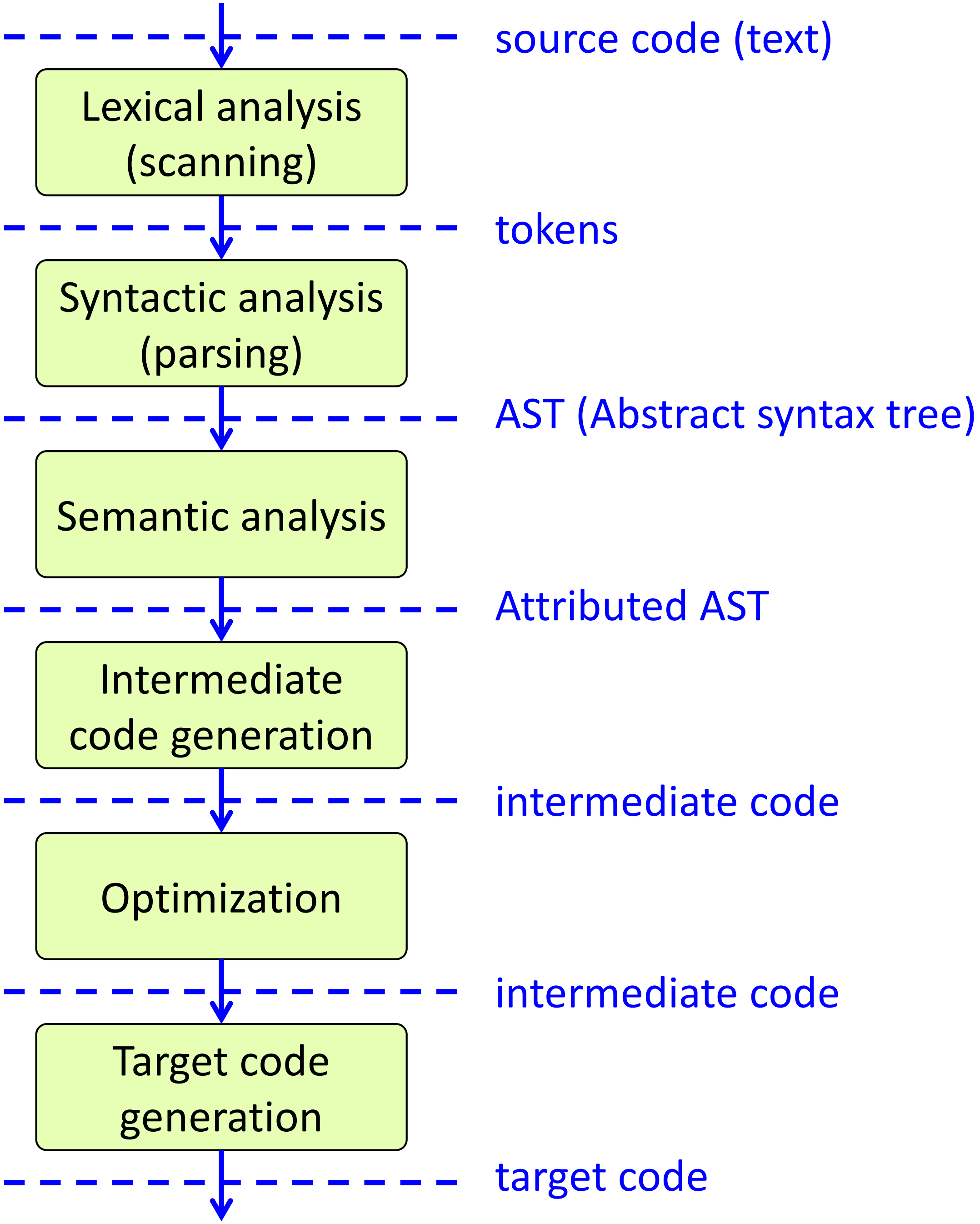
Inside the compiler:

source code (text)

target code (instructions for machine)

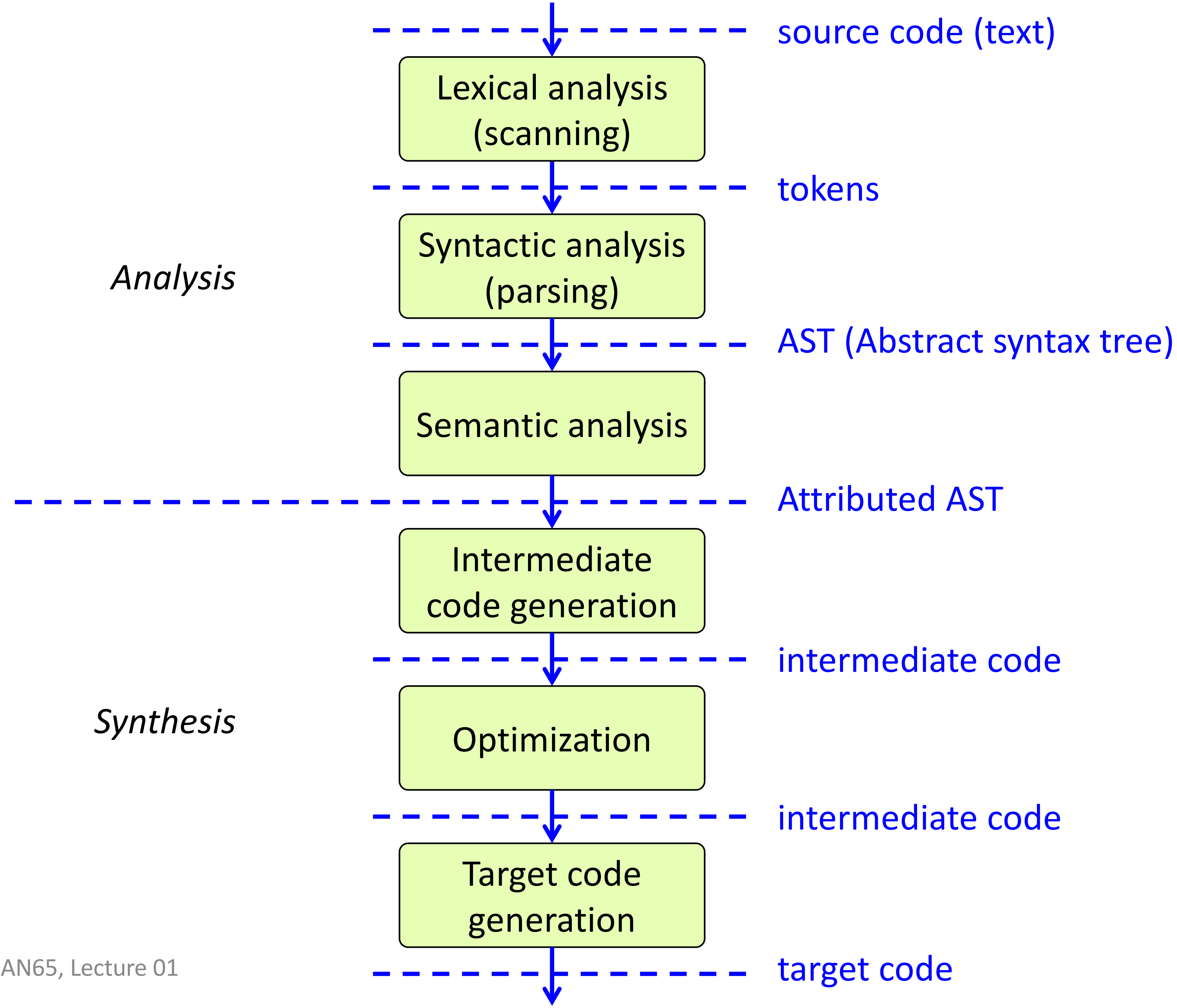
Inside the compiler:

Each ***phase*** converts the program from one ***representation*** to another

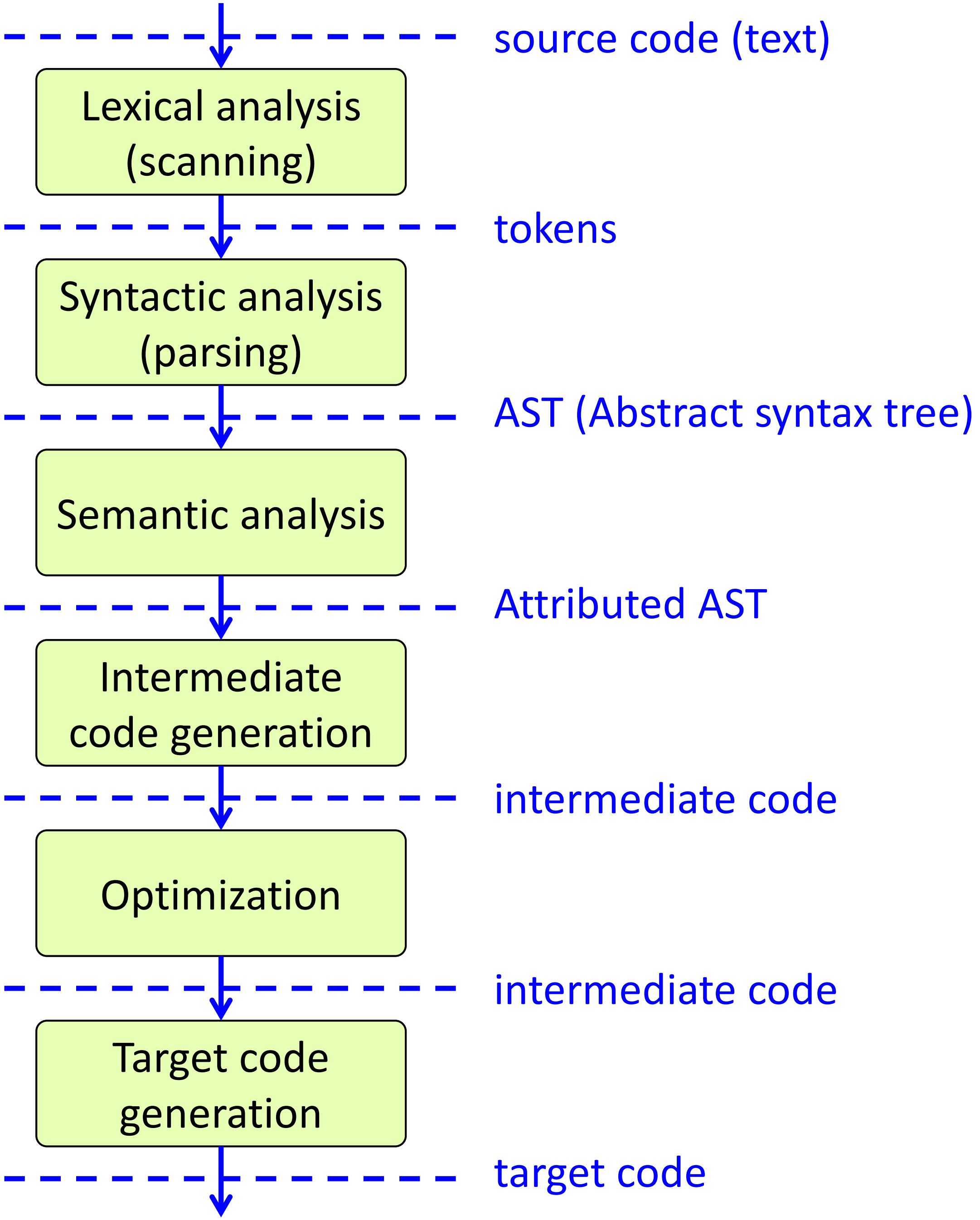


Inside the compiler:

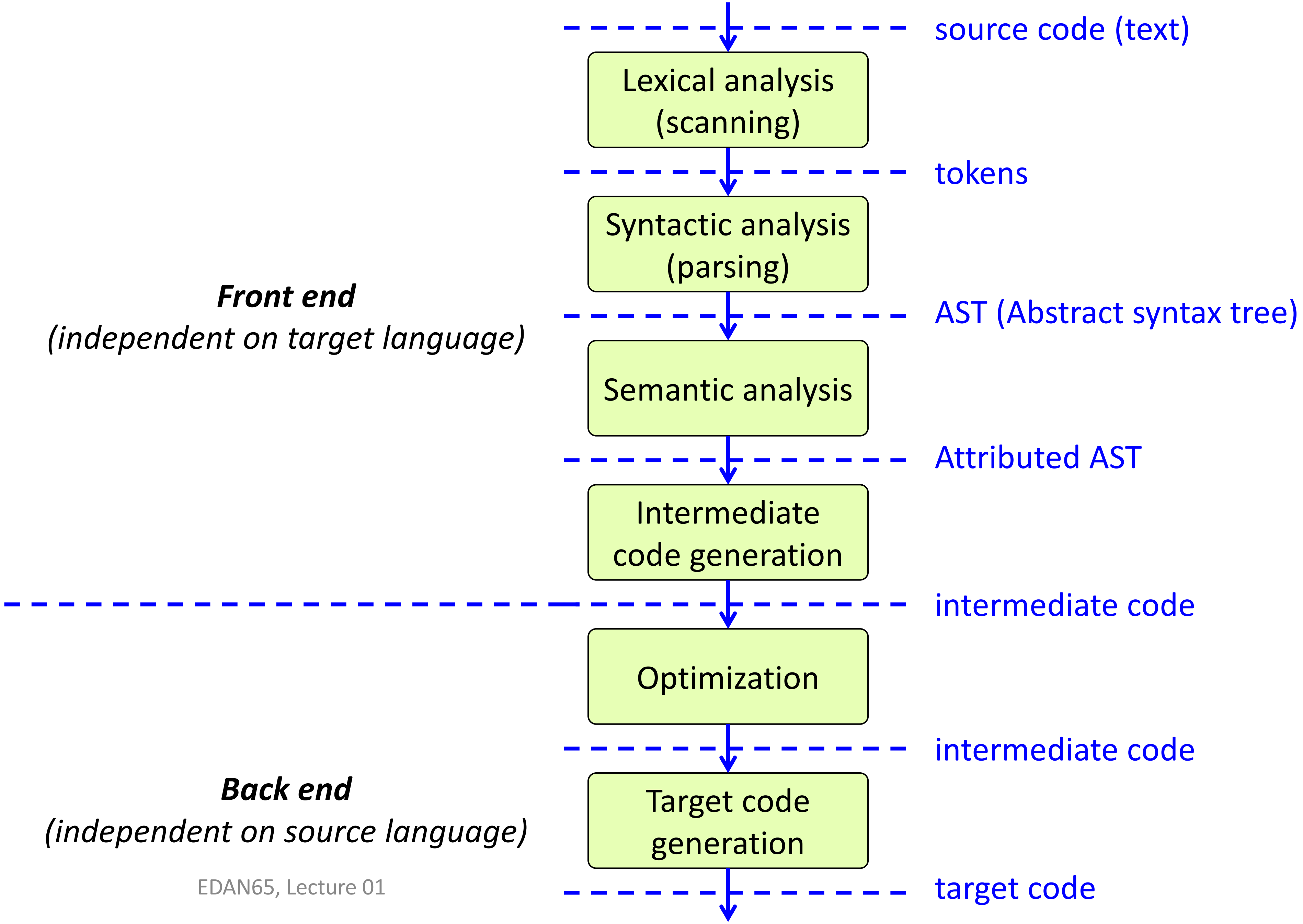
Each ***phase*** converts the program from one ***representation*** to another



Front and back end:



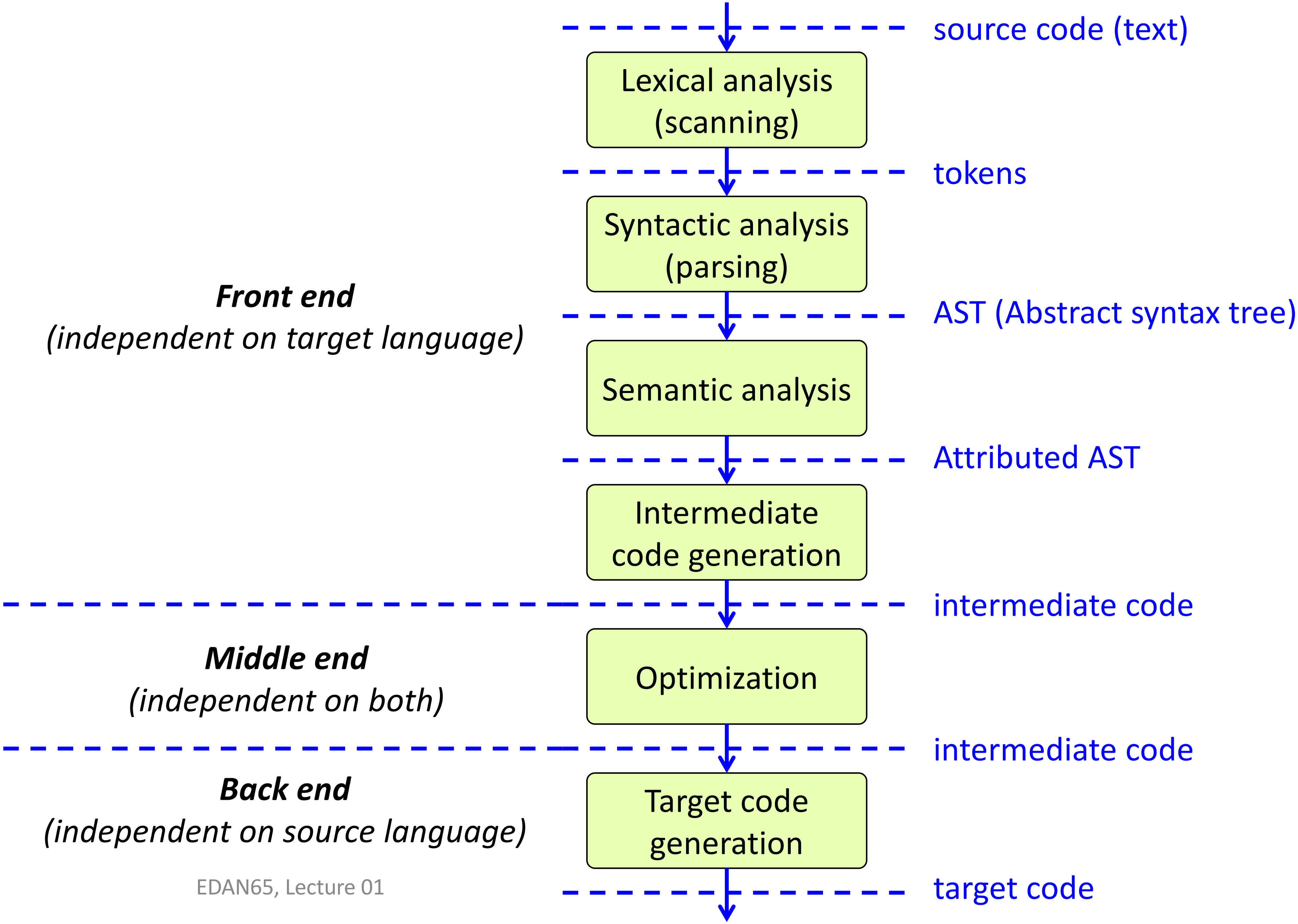
Front and back end:



Front end
(independent on target language)

Back end
(independent on source language)

Front and back end:

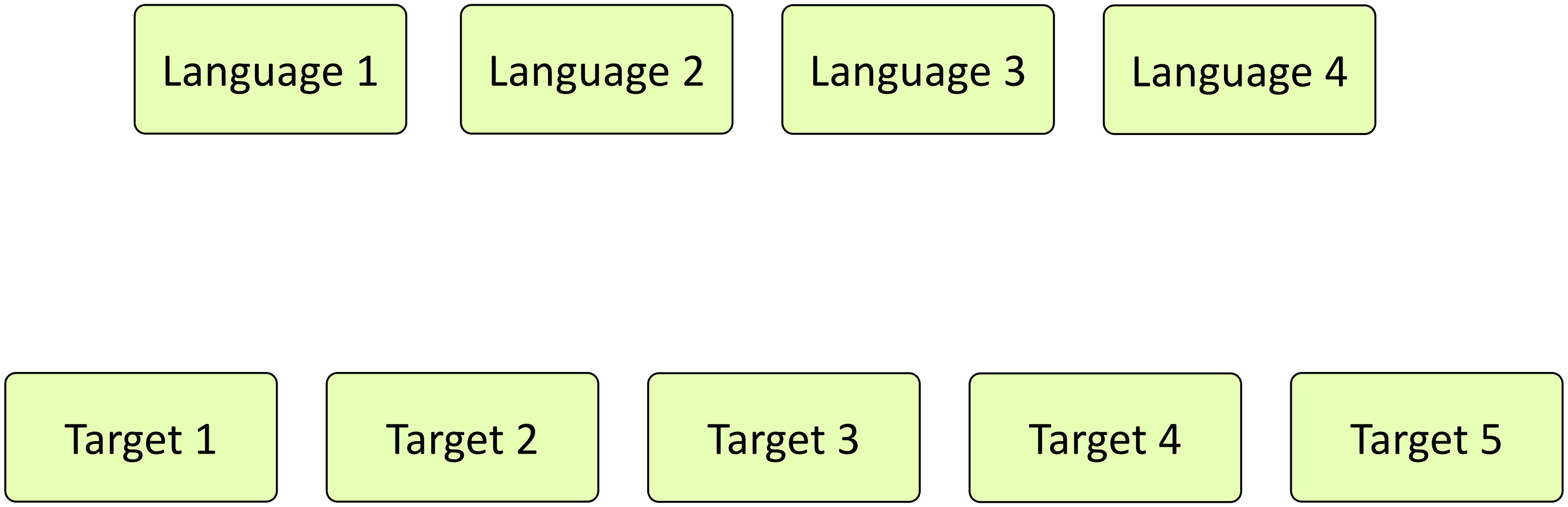


Front end
(independent on target language)

Middle end
(independent on both)

Back end
(independent on source language)

Multiple languages and target machines



Language 1

Language 2

Language 3

Language 4

Target 1

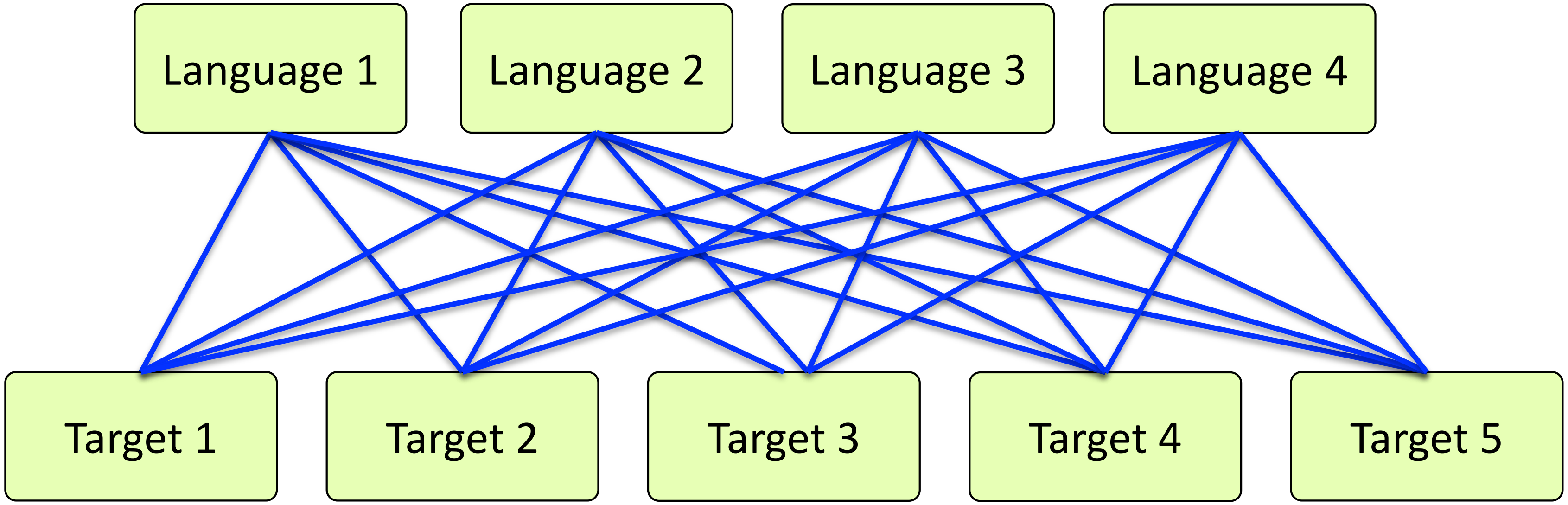
Target 2

Target 3

Target 4

Target 5

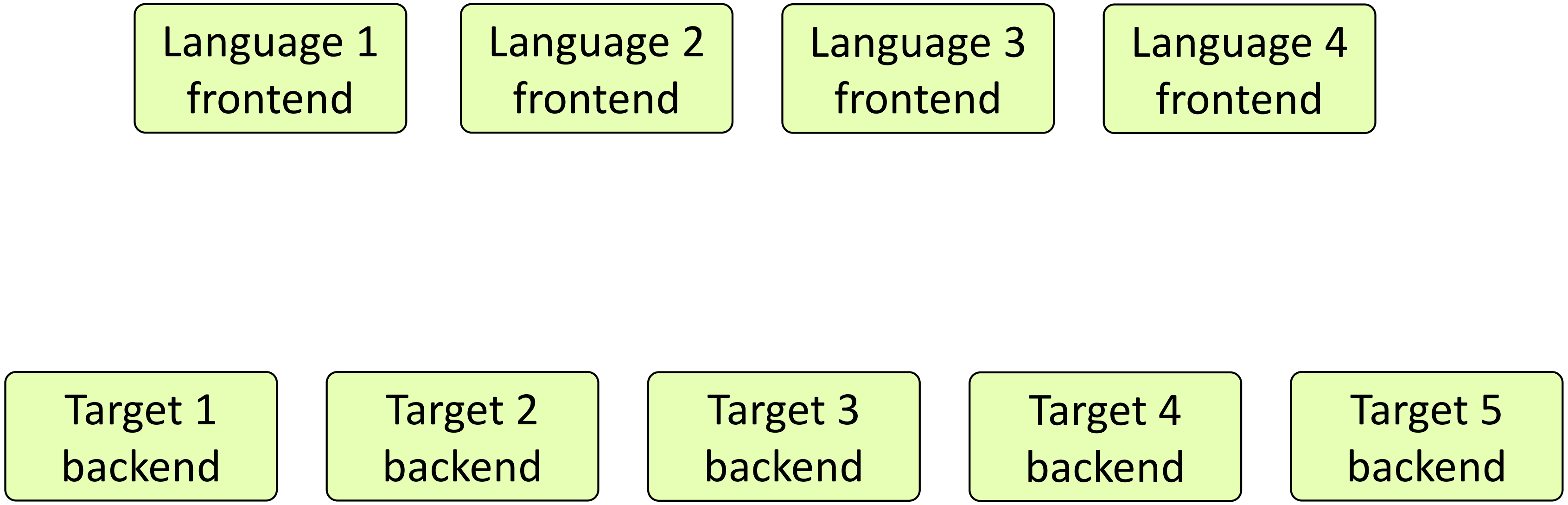
Multiple languages and target machines



m languages
n target machines
m*n compilers

Is there a smarter way?

Multiple languages and target machines



Language 1 frontend

Language 2 frontend

Language 3 frontend

Language 4 frontend

Target 1 backend

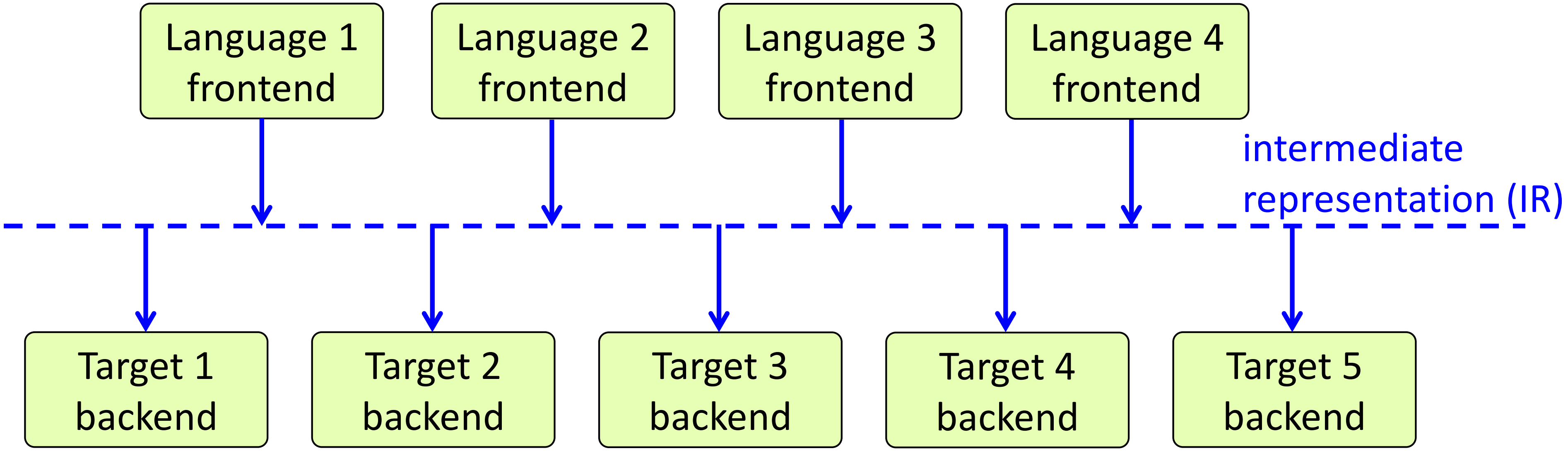
Target 2 backend

Target 3 backend

Target 4 backend

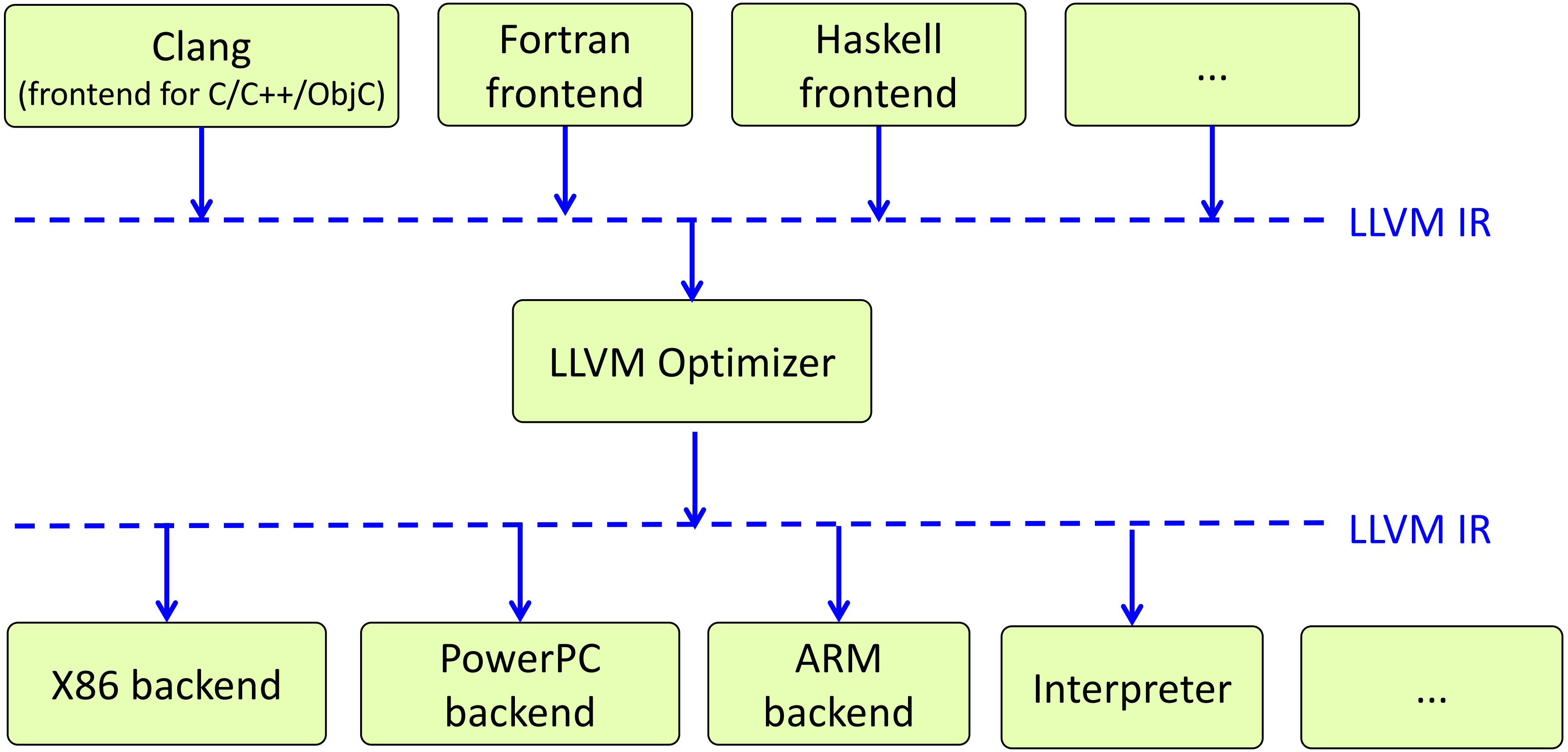
Target 5 backend

Multiple languages and target machines



m language frontends
n target backends
m + n components

LLVM example



Clang
(frontend for C/C++/ObjC)

Fortran frontend

Haskell frontend

...

LLVM IR

LLVM Optimizer

LLVM IR

X86 backend

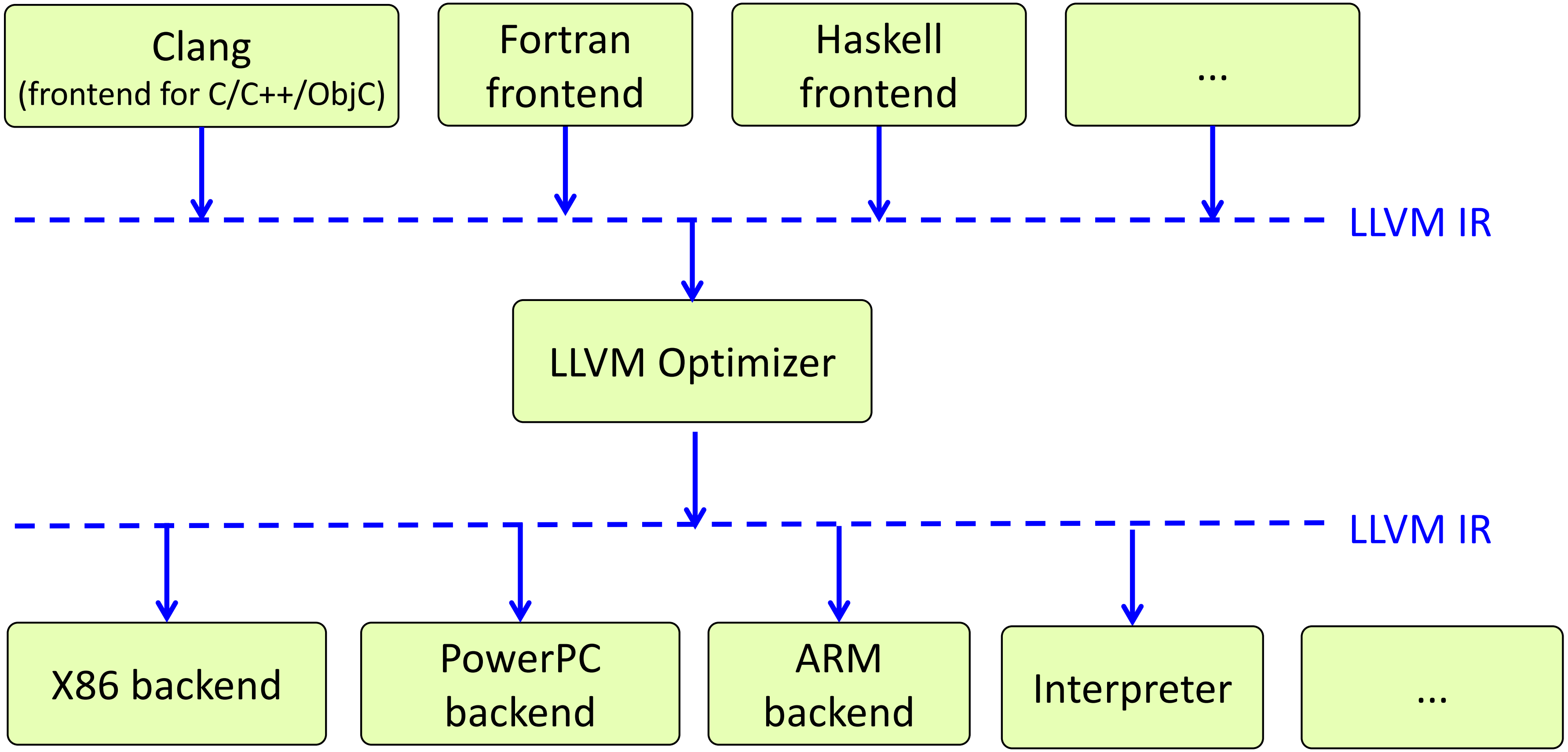
PowerPC backend

ARM backend

Interpreter

...

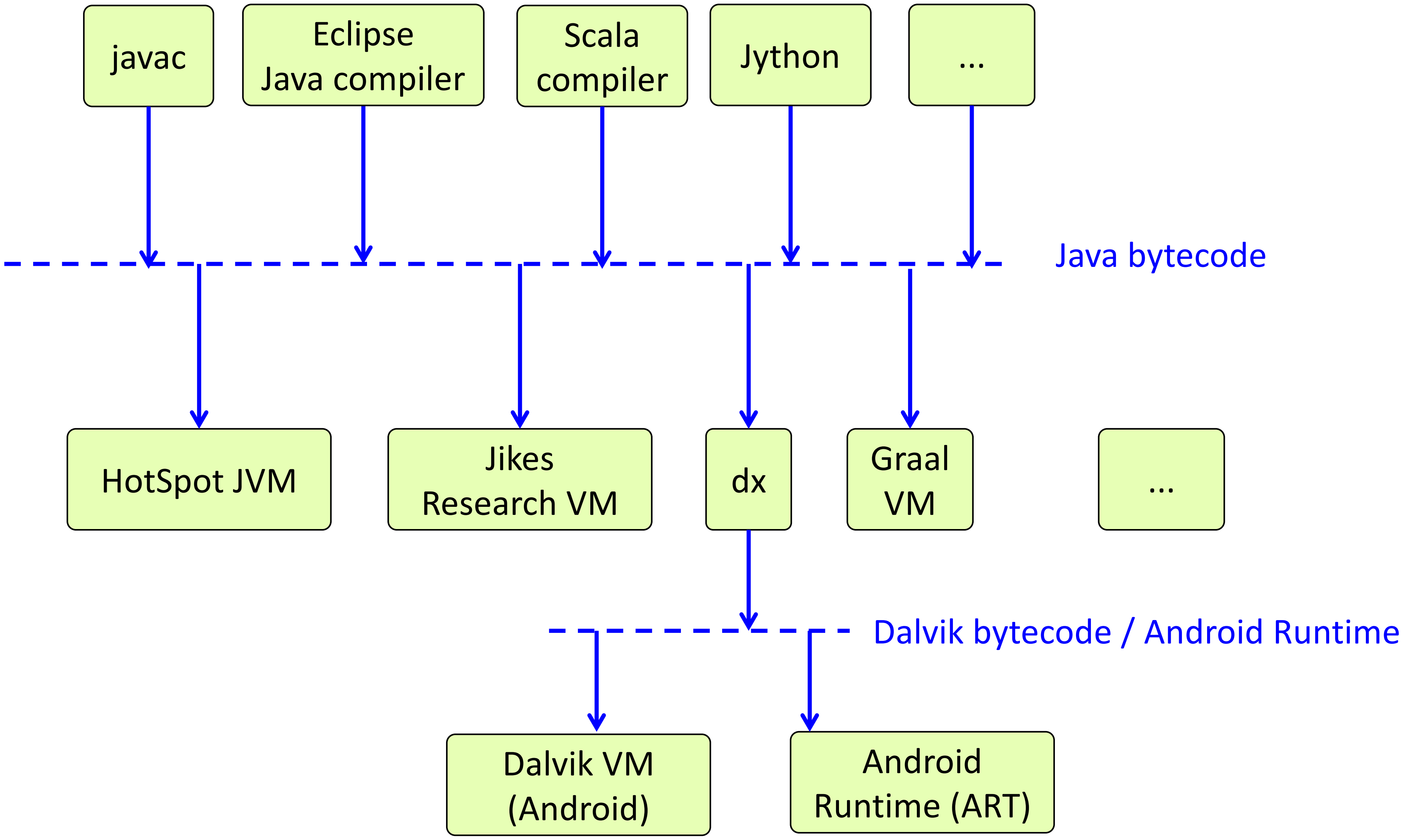
LLVM example



Why?

- Implement m front ends + n back ends instead of m * n compilers.
- Many optimizations are best performed on intermediate code.
- Easier to debug the front end using an interpreter than a target machine

Java example:



Some terminology

- A ***compiler*** translates code in a high-level language to a lower-level language.
  Examples: Compiling Java source code to Java bytecode. Compiling C source code to assembly code.

- An ***interpreter*** is software that executes a high/low level program, often by calling one procedure for each program construct. (This is in contrast to executing the program directly on hardware.)
  Example: A Python interpreter reads Python code and runs it.

- A ***transpiler*** (or source-to-source translator) translates code from one high-level language to another.
  Example: Transpiling Typescript source code to Javascript.

- A ***virtual machine (VM)*** is an interpreter that executes low-level, usually platform-independent code. (This is in the context of language implementation. In other contexts, the term "virtual machine" can mean operating system virtualization.)
  Example: The JVM (Java Virtual Machine) executes Java bytecode.

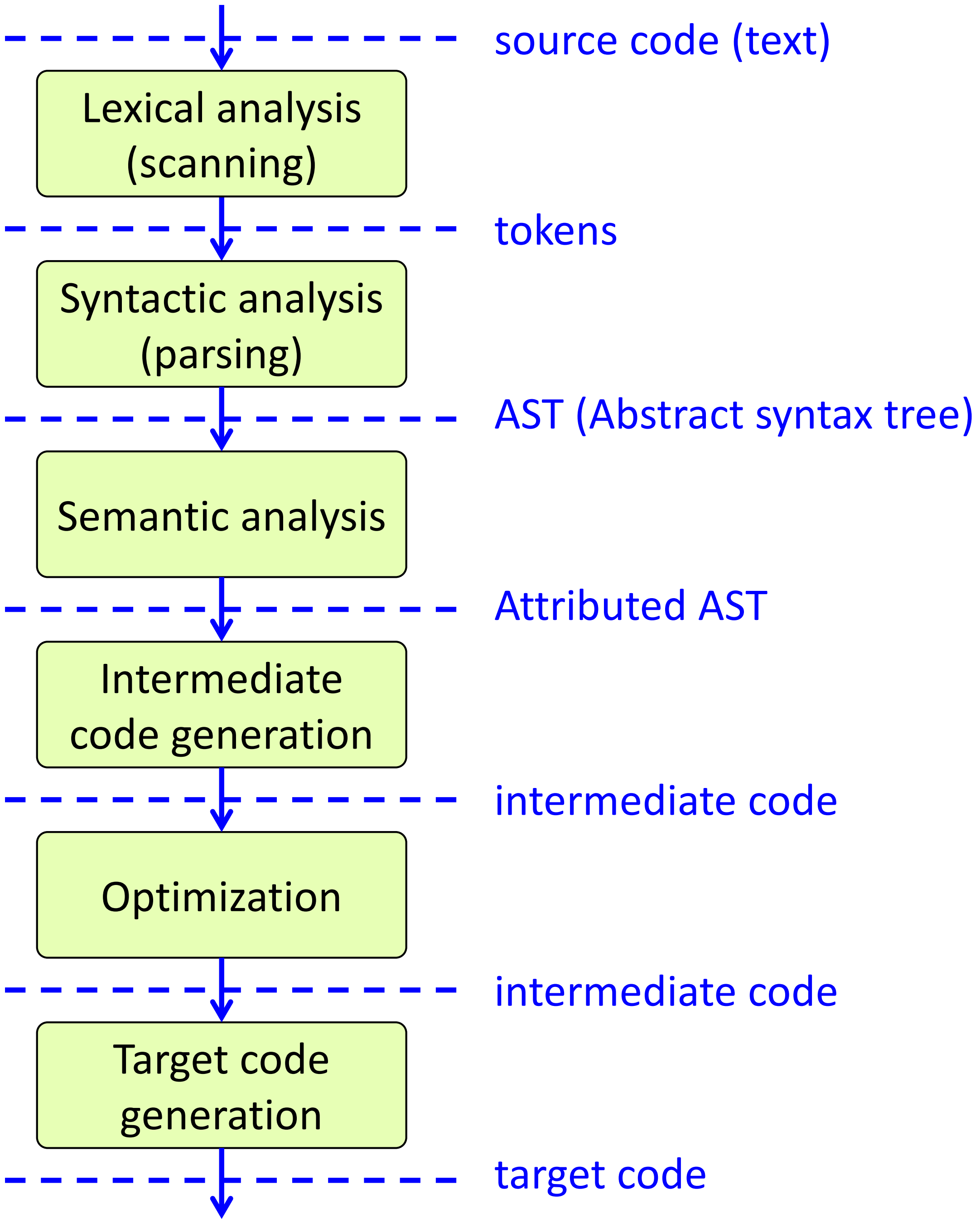
- Platform-independent low-level code, designed to be executed by a VM, was originally called ***p-code*** (portable code), but is now usually called ***bytecode***.

- An interpreter or VM may use a ***JIT*** (“Just In Time”) compiler to compile all or parts of the program into machine code during execution. In contrast, a traditional compiler compiles to machine code ***AOT*** ("Ahead Of Time"), i.e., before execution starts.

Some historical roots

- The first compiler was developed by Grace Hopper in 1952.
- John McCarthy used JIT compilation in his LISP interpreter in 1960. This was called "Compile and Go". The term JIT came later, and was popularized with Java.
- The Pascal-P system, developed by Niklaus Wirth's group at ETH in 1972, used portable code called "p-code". A successor, UCSD-Pascal, used a variant of p-code that was byte-oriented. The UCSD interpreter was easy to port to different machines and the language spread quickly, and became a popular language taught at many universities. Java, introduced in 1994, used easily portable bytecode, and spread quickly for the same reasons.
- Smalltalk-80 used bytecode in the 1980s, and pioneered several runtime compilation and optimization techniques for object-oriented languages.
- The research language Self refined Smalltalk's dynamic optimization techniques during the early 1990s. This work laid the foundation for the dynamically optimizing VMs we use today, like the Java Hotspot VM, the Javascript V8 VM, and many others.

Compiler phases and program representations:



source code (text)

Lexical analysis (scanning)

tokens

Syntactic analysis (parsing)

AST (Abstract syntax tree)

Semantic analysis

Attributed AST

Intermediate code generation

intermediate code

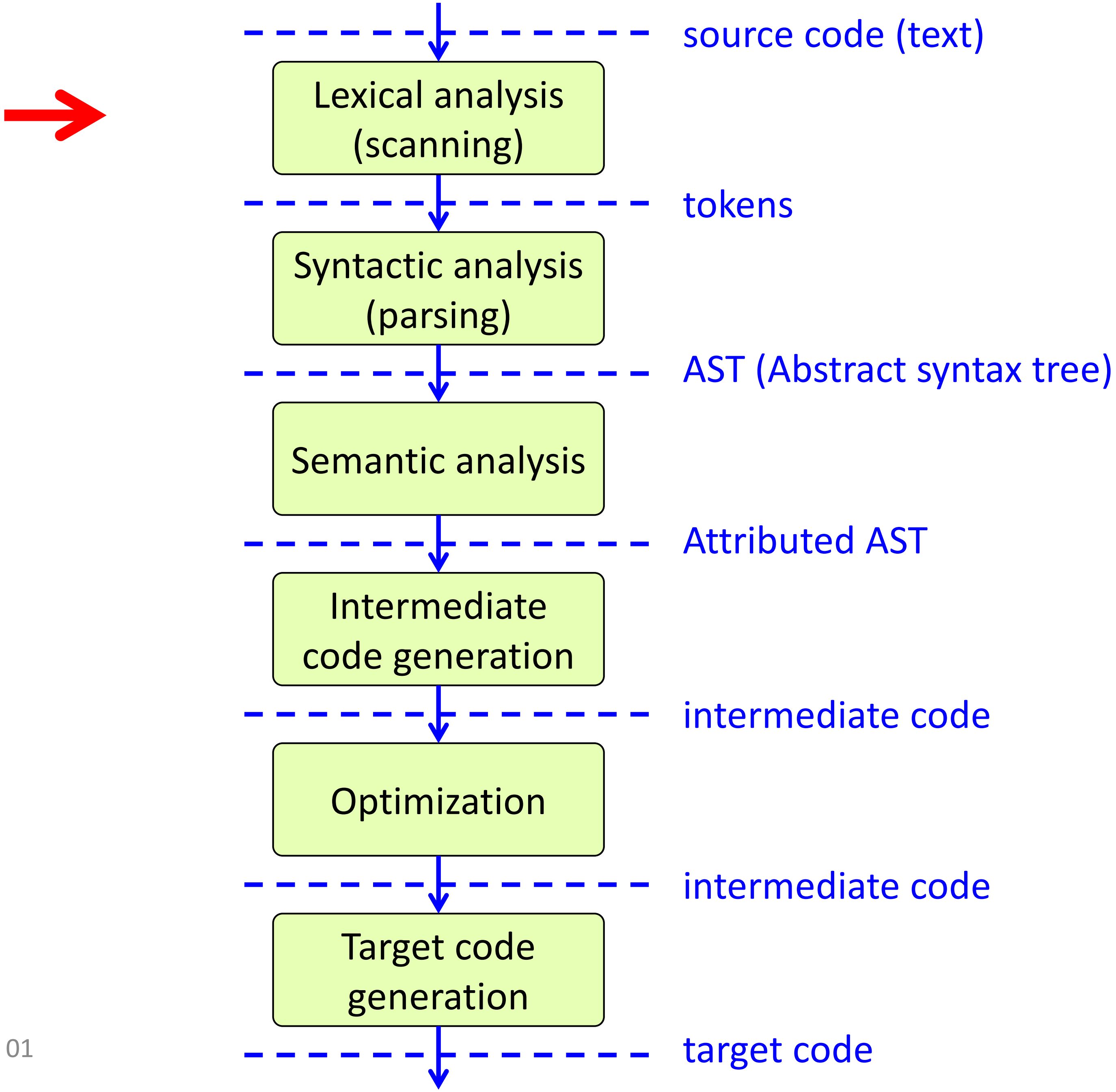
Optimization

intermediate code

Target code generation

target code

Compiler phases and program representations:



Lexical analysis (scanning)

```
Source text:
```

```
while (k<=n) {
  sum=sum+k;
  k=k+1;
}
```

Lexical analysis (scanning)

Source text:

```
while (k<=n) {
  sum=sum+k;
  k=k+1;
}
```

What the file looks like:

```
while (k<=n) {\n  sum=sum+k;\n  k=k+1;\n}
```

Lexical analysis (scanning)

Source text:

```
while (k<=n) {
  sum=sum+k;
  k=k+1;
}
```

What the file looks like:

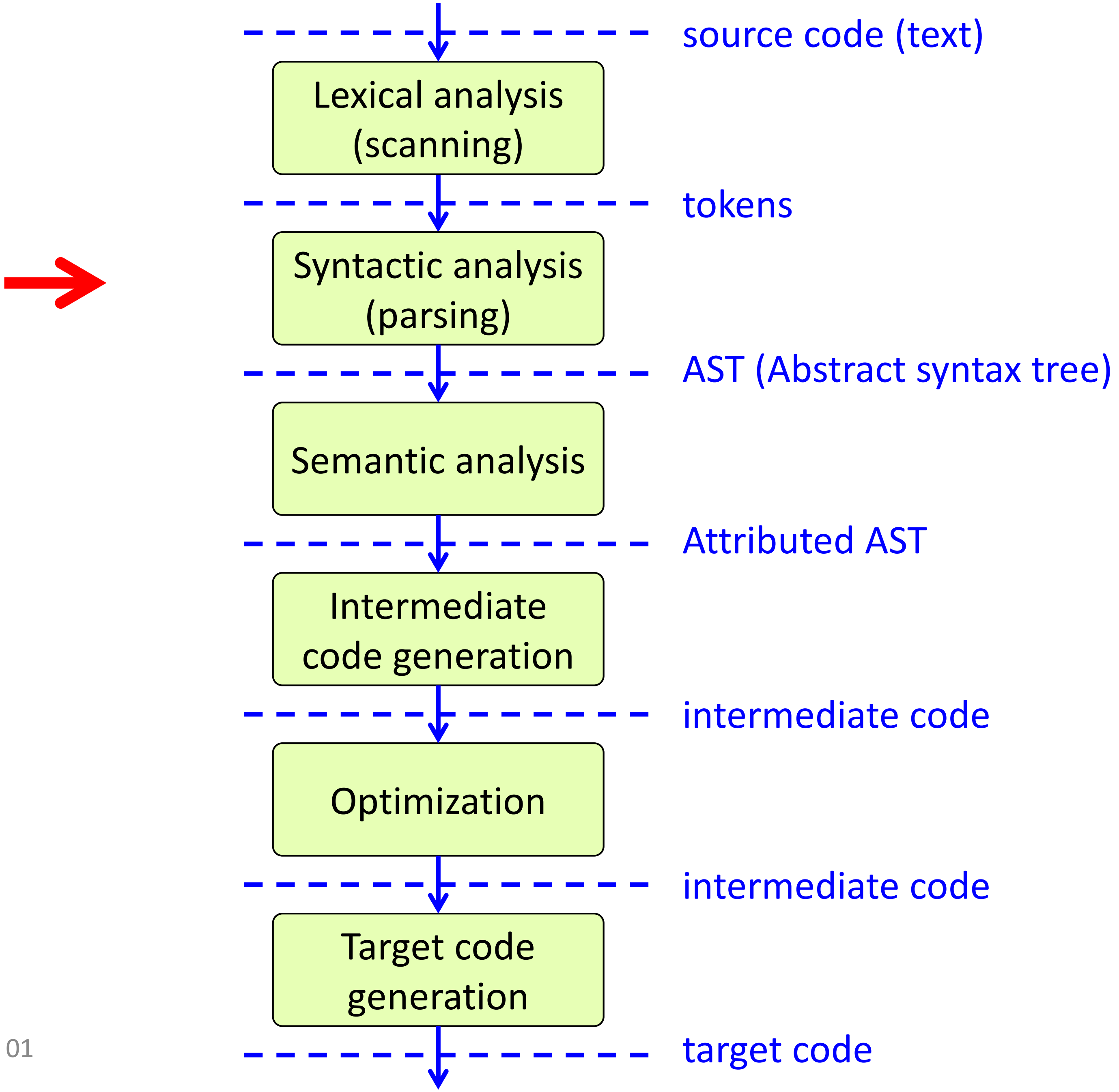
```
while (k<=n) {\n  sum=sum+k;\n  k=k+1;\n}
```

Tokens the scanner produces:

```
WHILE LPAR ID("k") LEQ ID("n") RPAR LBRA ID("sum") EQ ...
```

A *token* is a symbolic name, sometimes with an attribute.
A *lexeme* is a string corresponding to a token.
Whitespace (blanks, newlines, etc., are skipped)

Compiler phases and program representations:



Syntactic analysis (parsing)

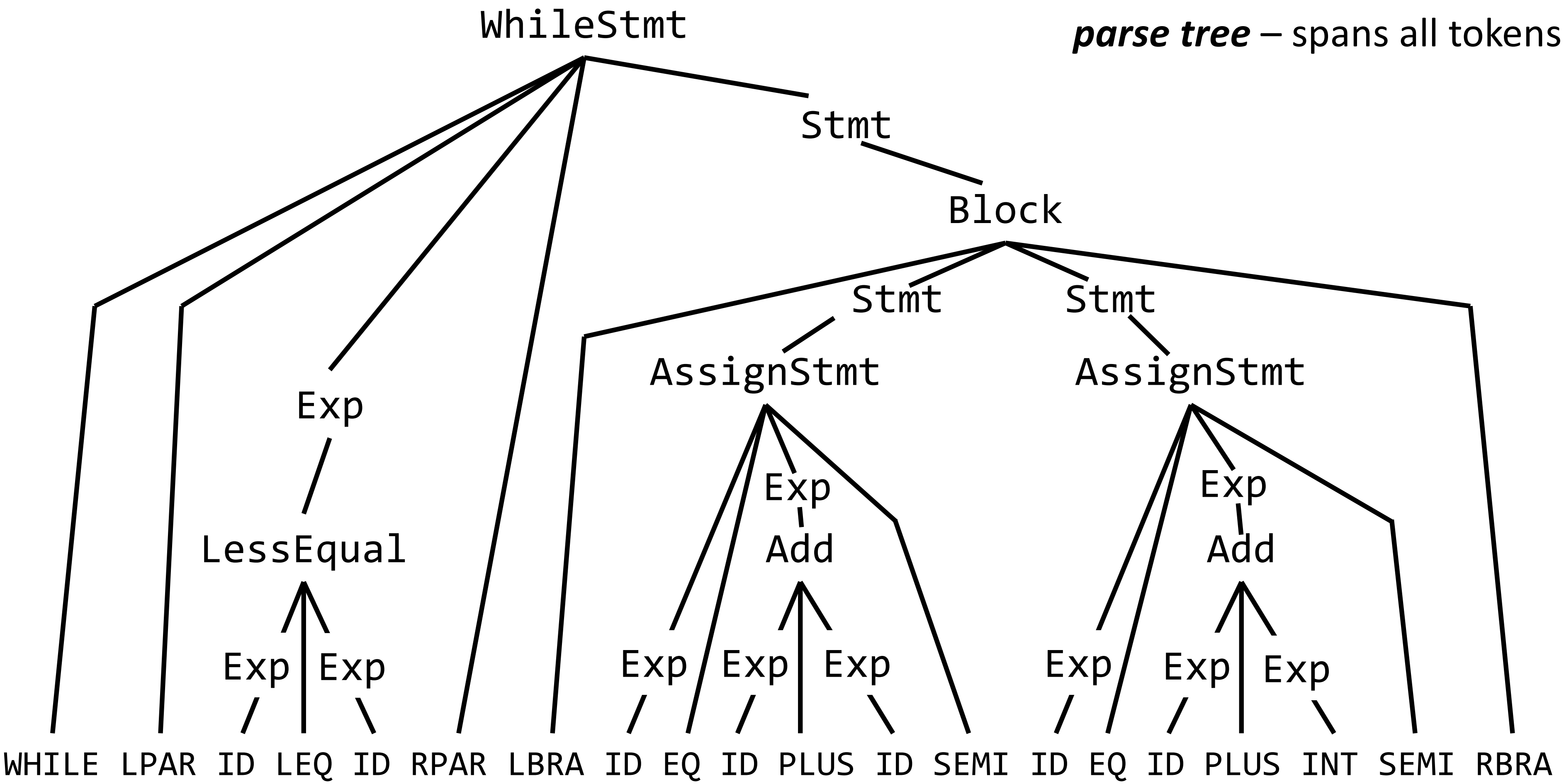
WhileStmt

parse tree – spans all tokens

```
WHILE LPAR ID LEQ ID RPAR LBRA ID EQ ID PLUS ID SEMI ID EQ ID PLUS INT SEMI RBRA
while ( k <= n ) { sum = sum + k ; k = k + 1 ; }
```

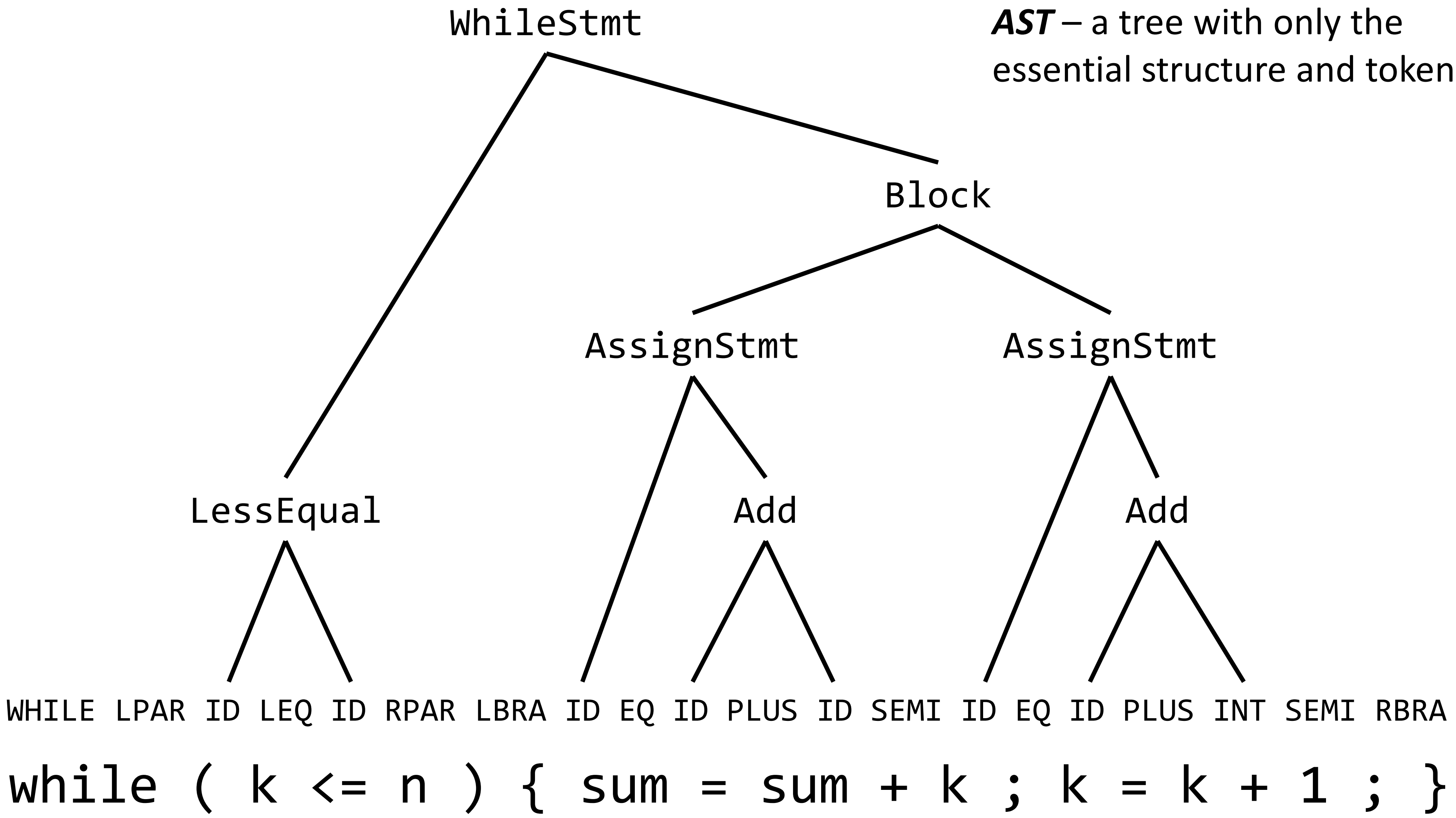
Syntactic analysis (parsing)

parse tree – spans all tokens



```
while ( k <= n ) { sum = sum + k ; k = k + 1 ; }
```

Abstract syntax tree (AST)


WhileStmt
Block
AssignStmt
AssignStmt
LessEqual
Add
Add
WHILE LPAR ID LEQ ID RPAR LBRA ID EQ ID PLUS ID SEMI ID EQ ID PLUS INT SEMI RBRA
while (k <= n) { sum = sum + k ; k = k + 1 ; }

AST – a tree with only the essential structure and tokens

Abstract syntax trees

- Used inside the compiler for representing the program
- Very similar to the parse tree, but
  - contains only essential tokens
  - has a simpler more natural structure
- Often represented by a typed object-oriented model
  - abstract classes (Stmt, Expr, Decl, ...)
  - concrete classes (WhileStmt, IfStmt, Add, Sub, ...)

(or corresponding algebraic datatypes with constructors)

Designing an AST model

What abstract constructs are there in the language

- OOP: Make them abstract classes
- FP: Make them algebraic data types

What concrete constructs are there?

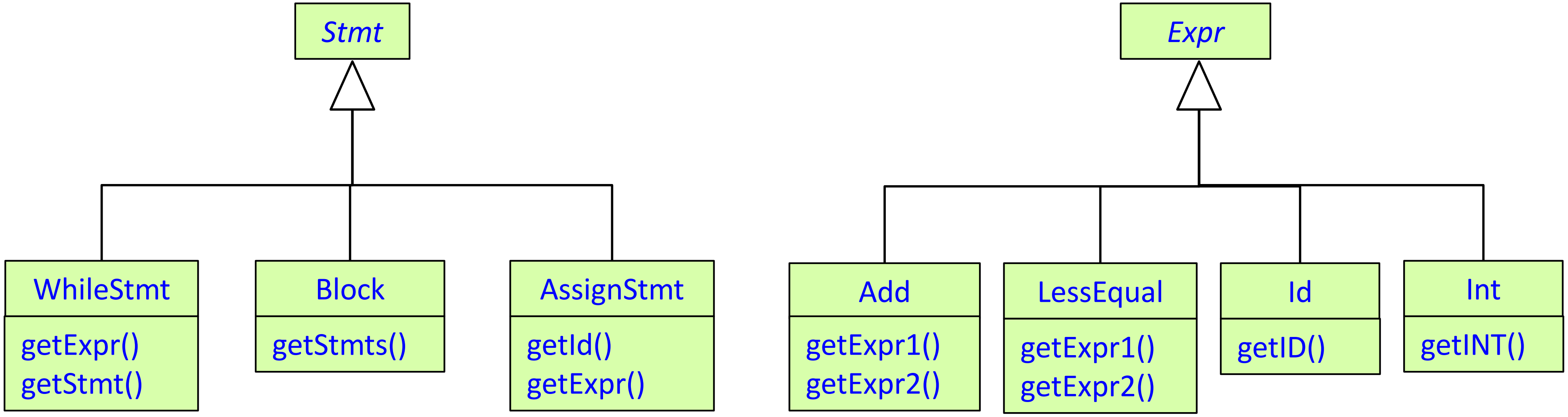
- OOP: Make them subclasses of the abstract classes
- FP: Make them constructors of the algebraic data types

What parts do the concrete constructs have?

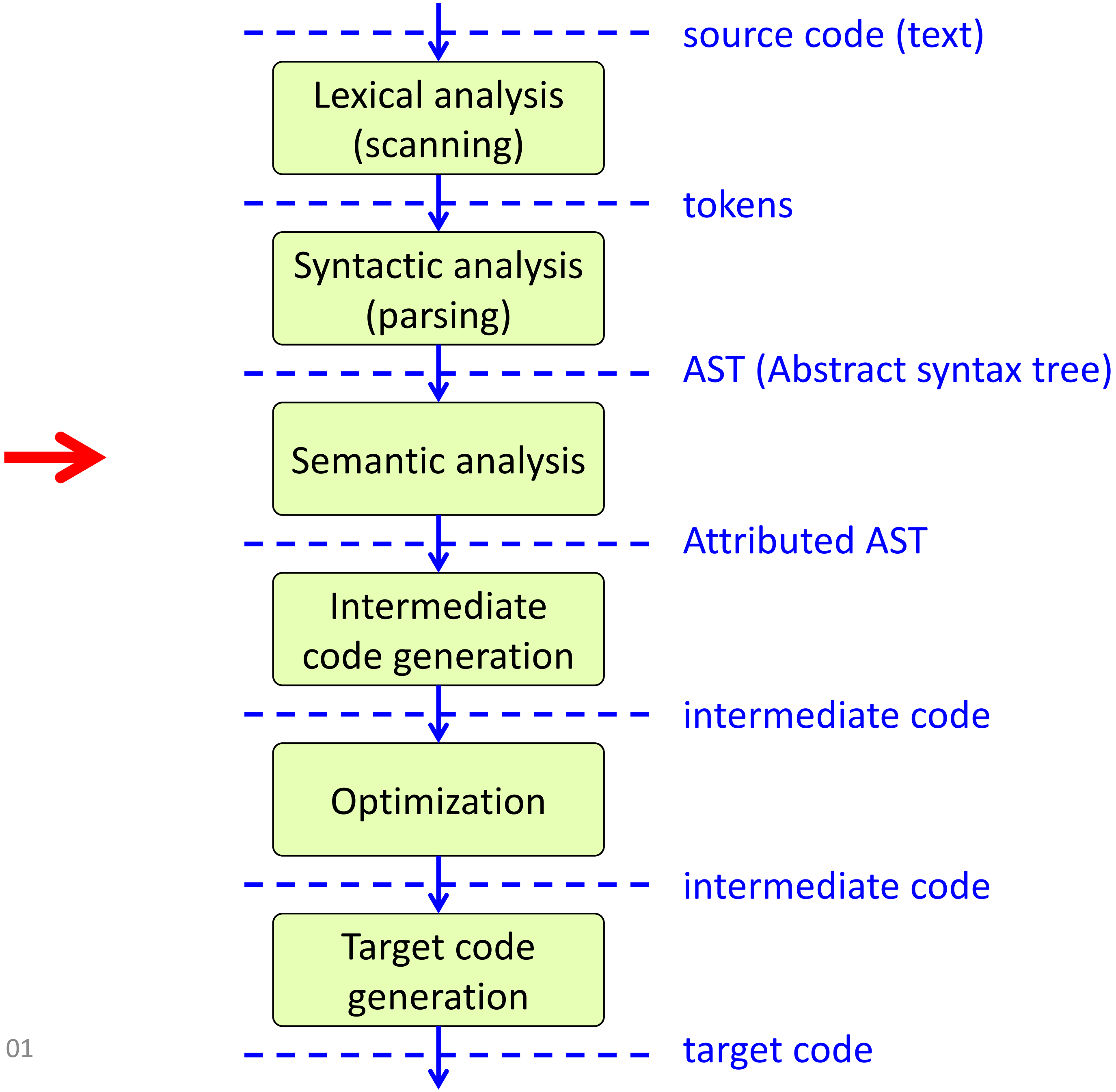
- OOP: Add getters for them, to access parts
- FP: Use pattern matching to access parts

Example AST class hierarchy

Example AST class hierarchy



Compiler phases and program representations:



source code (text)

Lexical analysis (scanning)

tokens

Syntactic analysis (parsing)

AST (Abstract syntax tree)

Semantic analysis

Attributed AST

Intermediate code generation

intermediate code

Optimization

intermediate code

Target code generation

target code

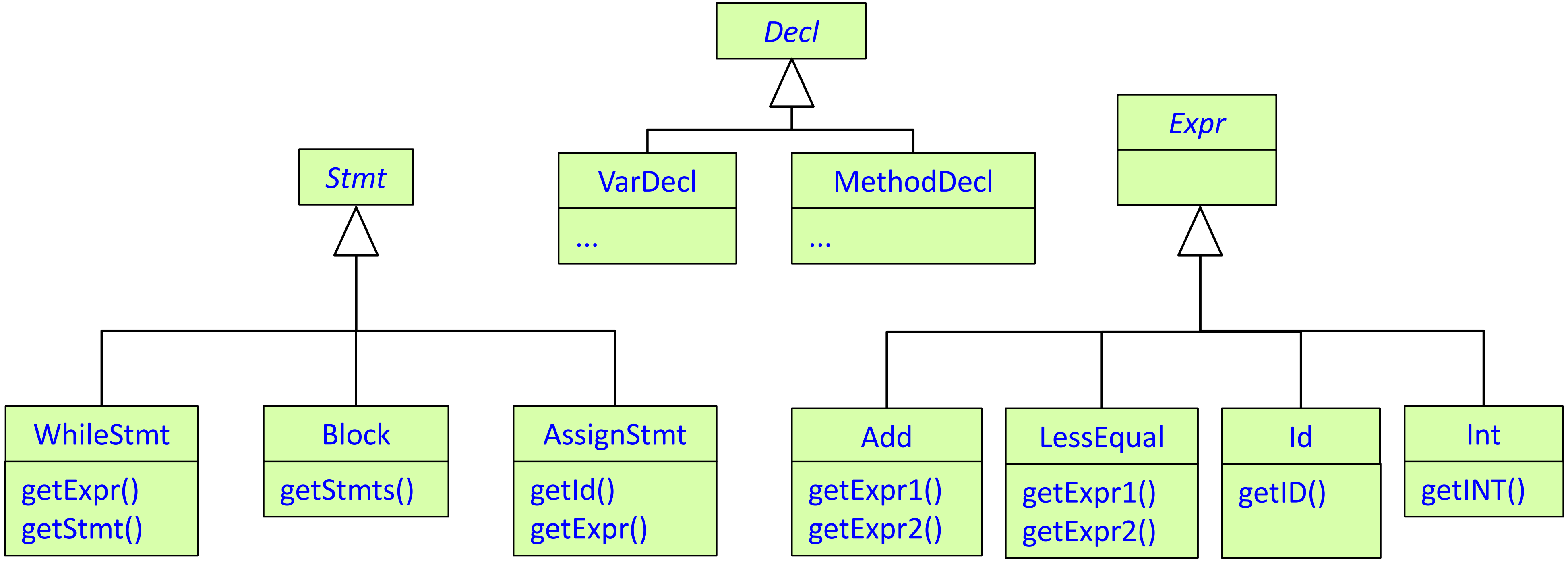
Semantic analysis

Analyze the AST, for example,

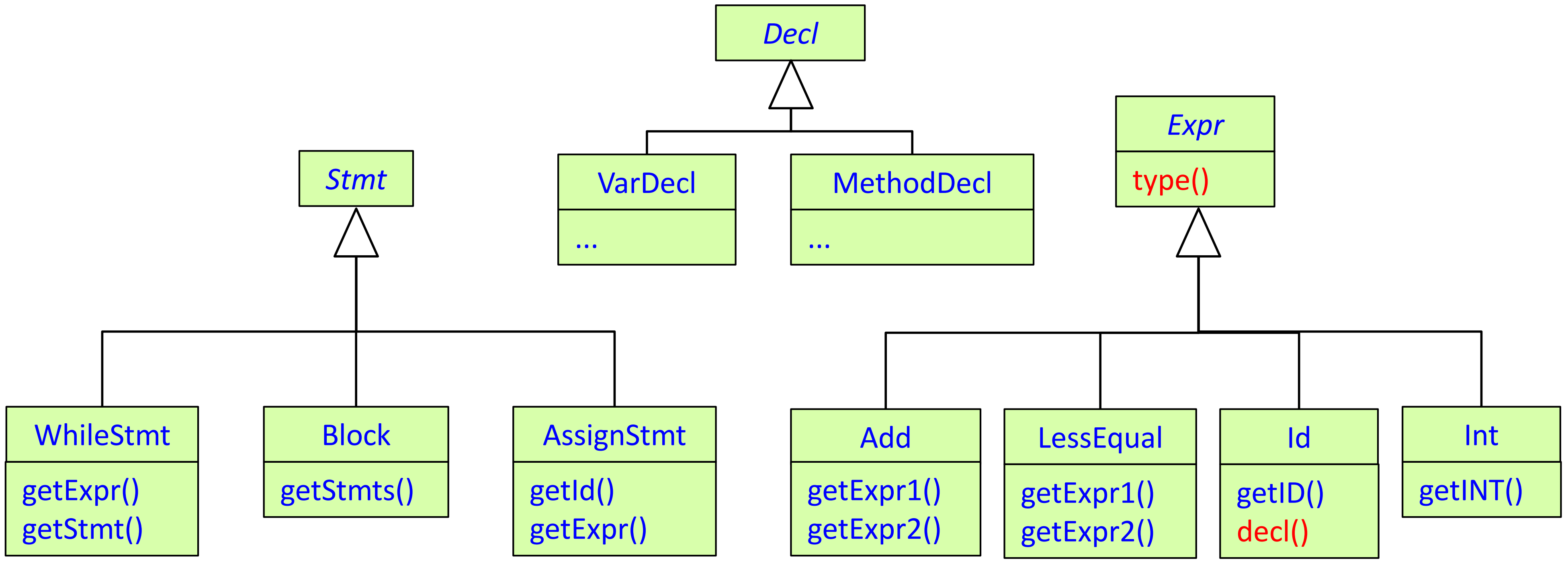
- Which declaration corresponds to a variable?
- What is the type of an expression?
- Are there compile time errors in the program?

Analysis aided by adding *attributes* to the AST (properties of AST nodes)

Example attributes

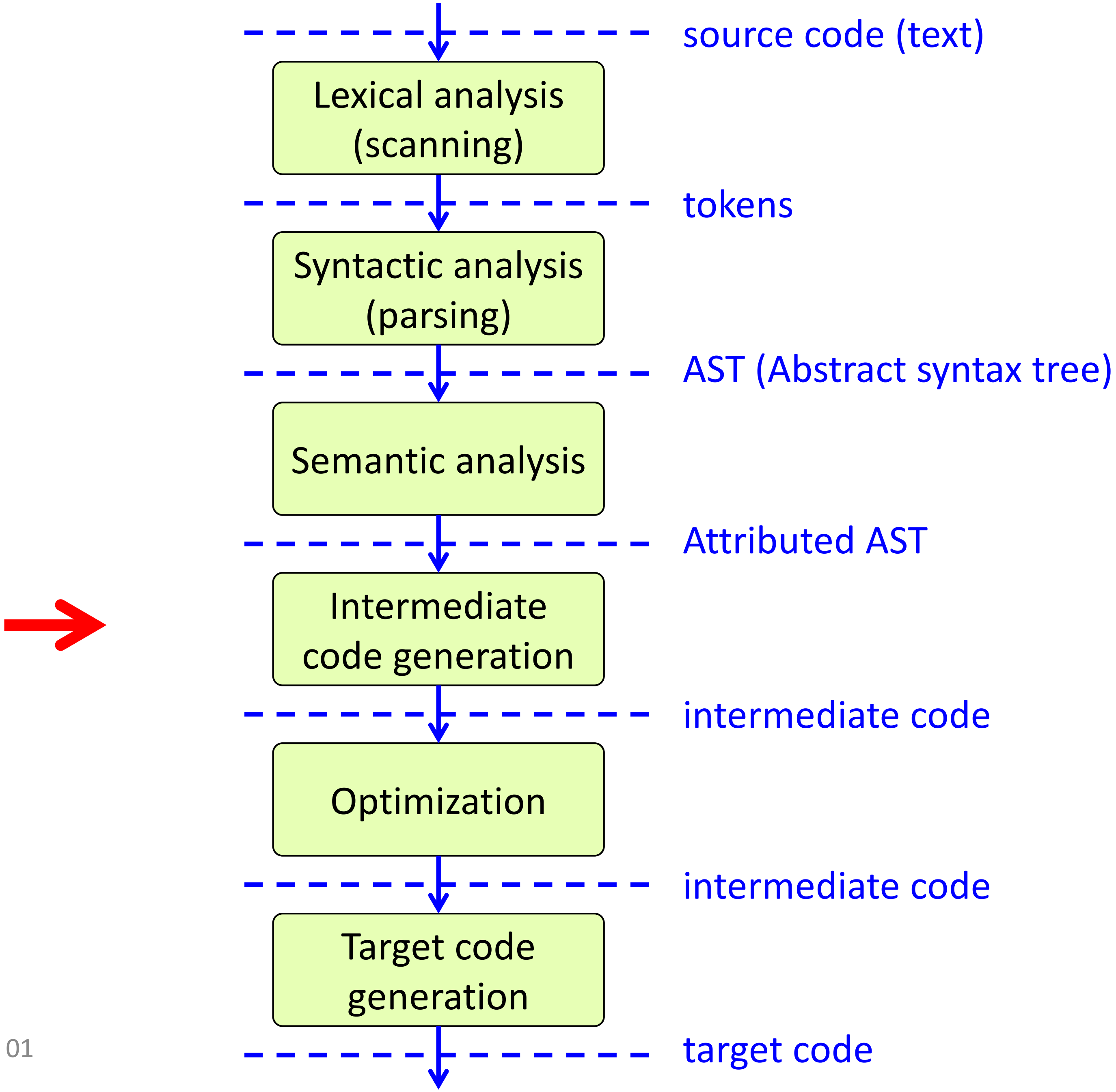


Example attributes



Each Expr has a type() attribute, indicating if the expression is integer, boolean, etc.
Each Id has a decl() attribute, referring to the appropriate declaration node.

Compiler phases and program representations:

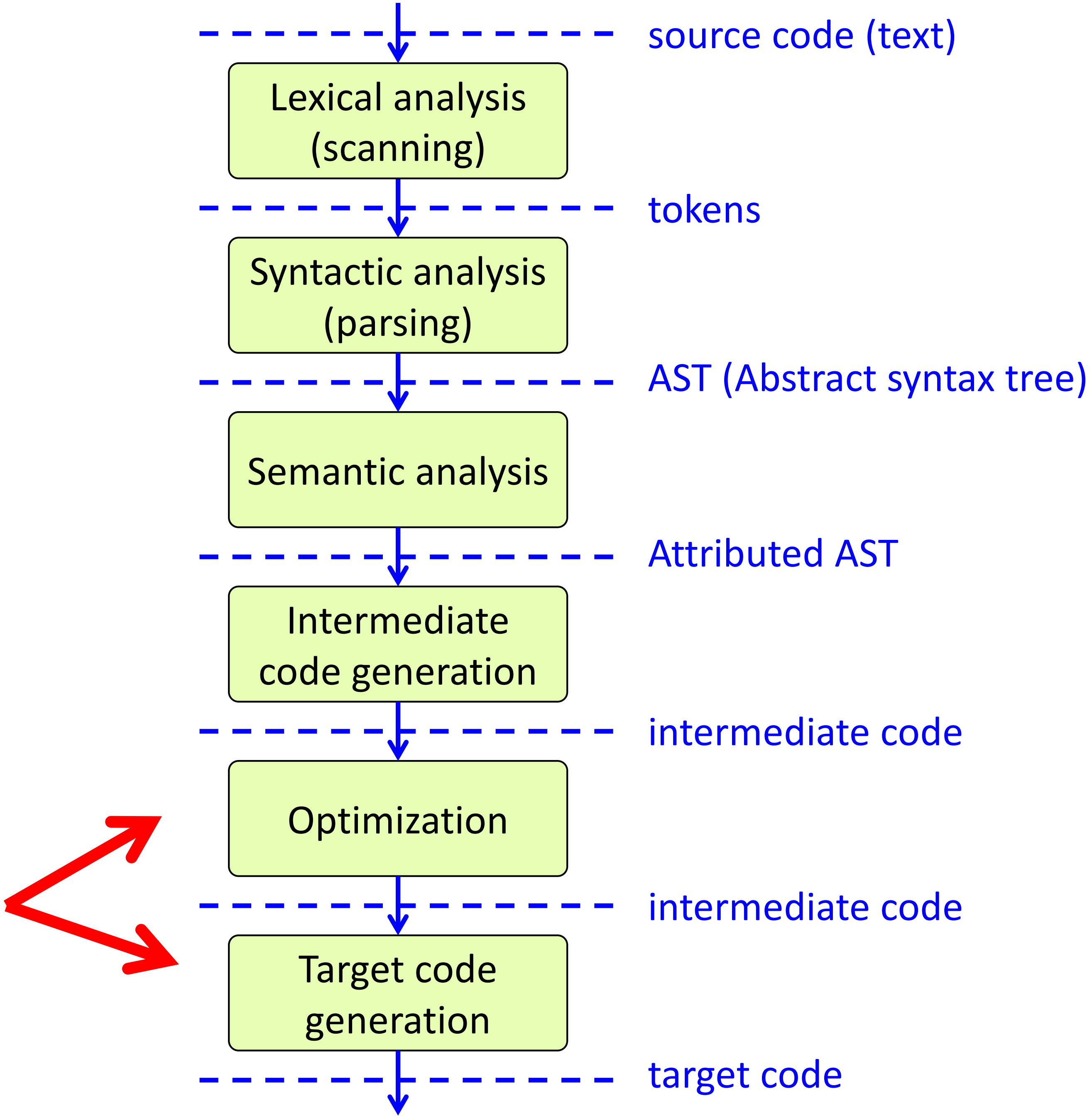


Intermediate code generation

Intermediate code:

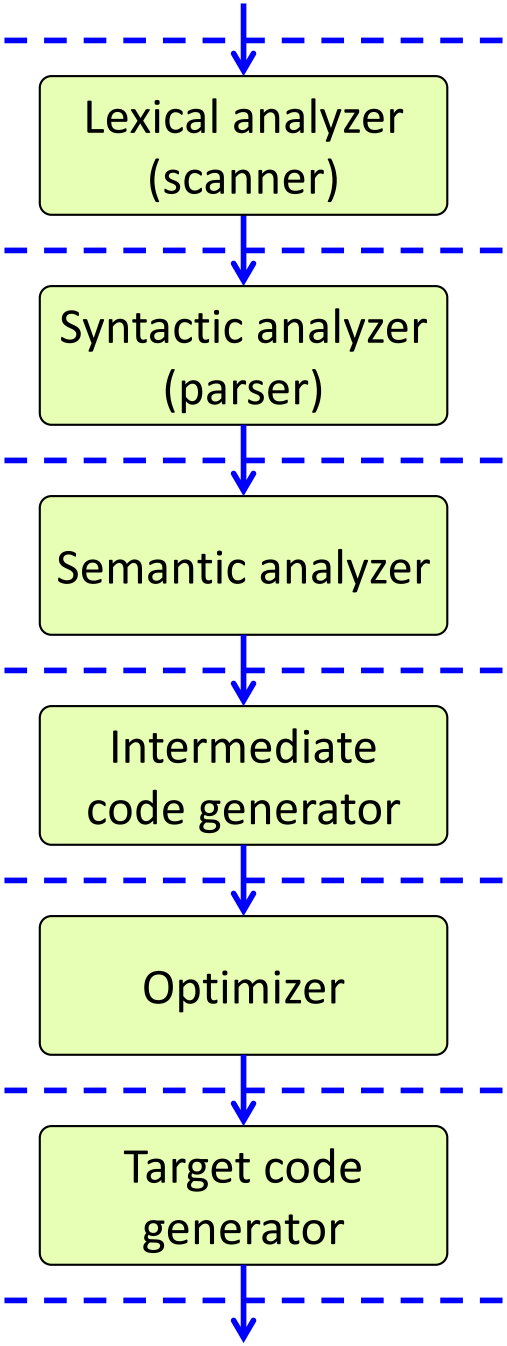
- also known as *intermediate representation* (IR)
- independent of source language
- independent of target machine
- usually assembly-like
  - but simpler, without many instruction variants
  - and with an unlimited number of registers
    (or uses a stack instead of registers)

Compiler phases and program representations:

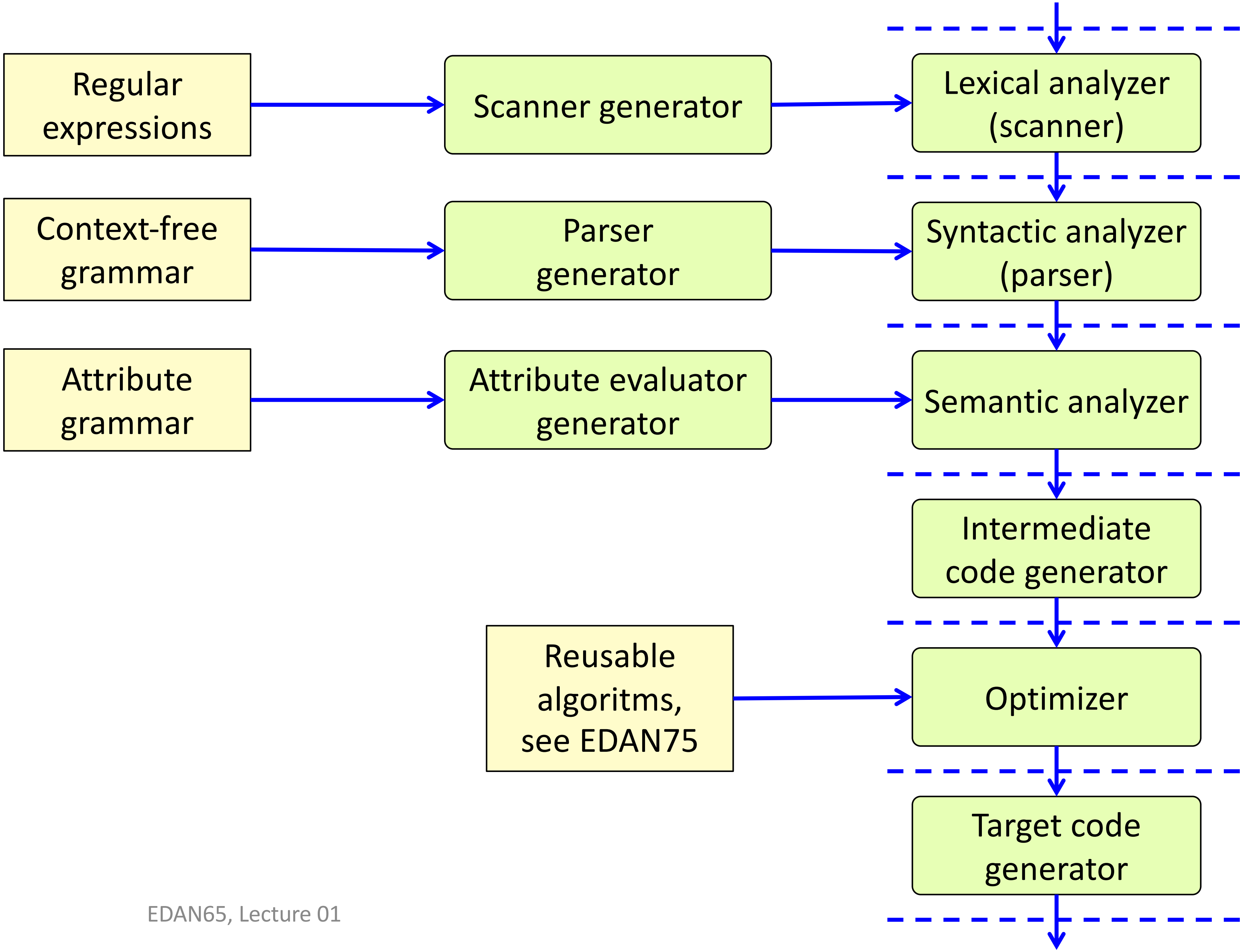


See the course Optimizing Compilers, EDAN75

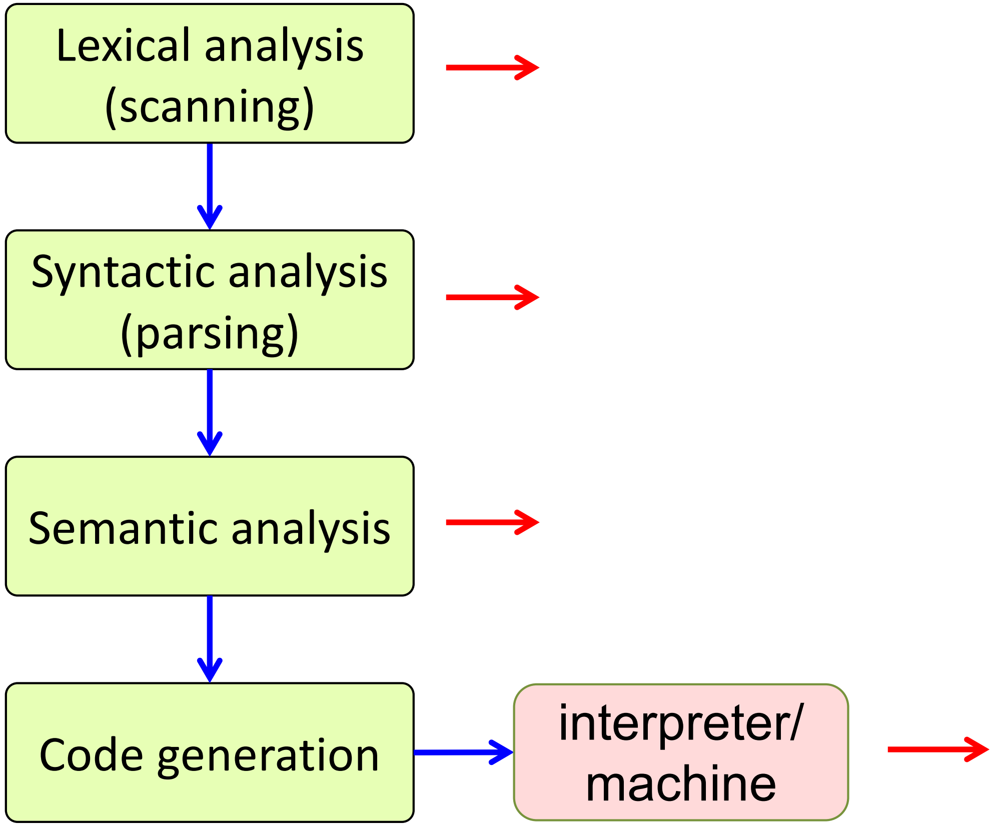
Generating the compiler:


Lexical analyzer (scanner)
Syntactic analyzer (parser)
Semantic analyzer
Intermediate code generator
Optimizer
Target code generator

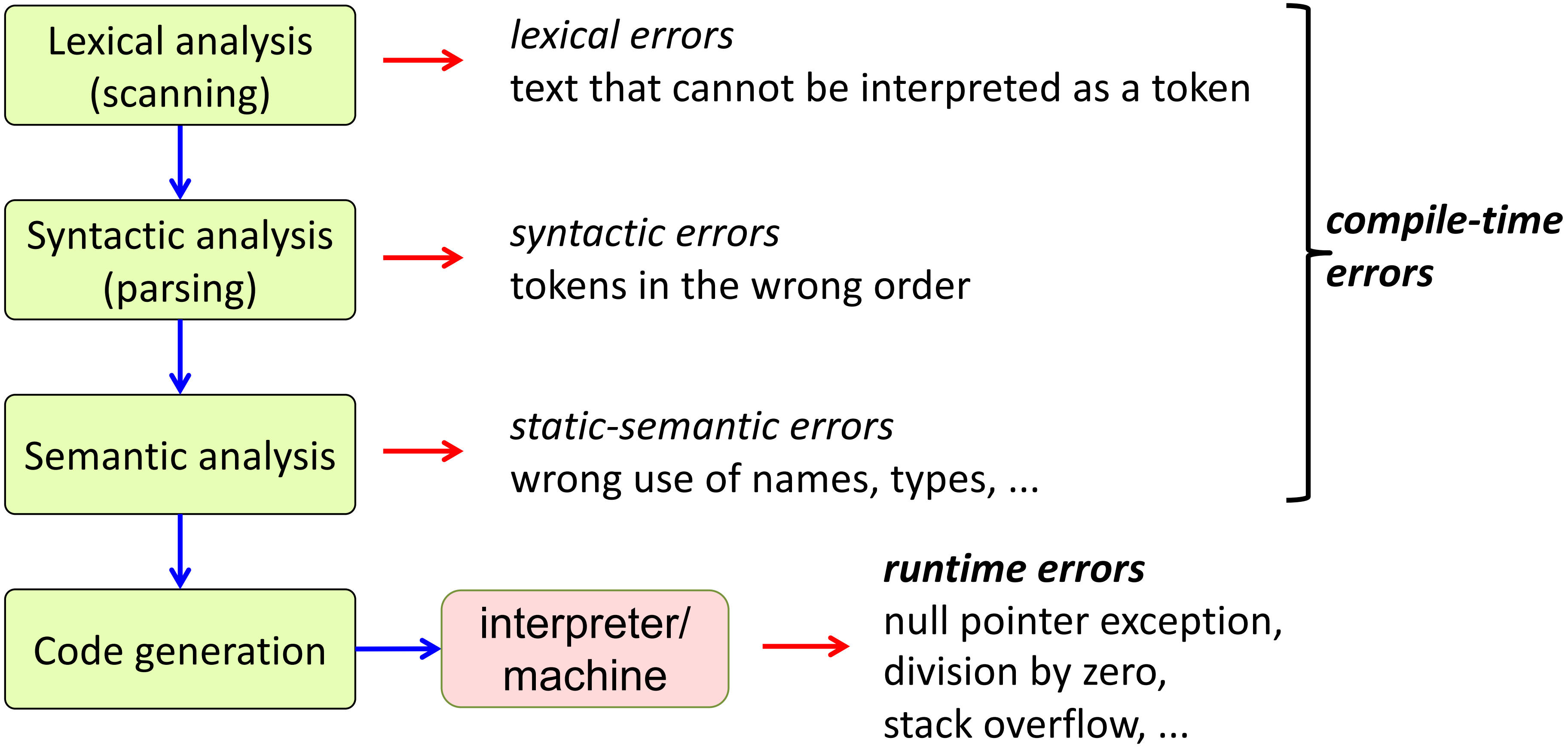
Generating the compiler:


Regular expressions
Scanner generator
Lexical analyzer (scanner)
Context-free grammar
Parser generator
Syntactic analyzer (parser)
Attribute grammar
Attribute evaluator generator
Semantic analyzer
Intermediate code generator
Reusable algoritms, see EDAN75
Optimizer
Target code generator

Program errors



Program errors



Lexical analysis (scanning)

lexical errors
text that cannot be interpreted as a token

Syntactic analysis (parsing)

syntactic errors
tokens in the wrong order

Semantic analysis

static-semantic errors
wrong use of names, types, ...

compile-time errors

Code generation

interpreter/ machine

runtime errors
null pointer exception,
division by zero,
stack overflow, ...

logic errors
Compute the wrong result.
Not caught by the compiler or the machine.
Normally try to catch using test cases.
Assertions and program verification can also help.

What kind is each error? Lexical, Syntactic, Static-semantic, Run-time, Logic?

Example errors

1.
```
int # square(int x) {
    return x * x;
}
```

2.
```
int double square(int x) {
    return x * x;
}
```

3.
```
boolean square(int x) {
    return x * x;
}
```

4.
```
int p(int x) {
    return x / 0;
}
```

5.
```
int square(int x) {
    return 2 * x;
}
```

Example errors

Lexical error:

1.
```
int # square(int x) {
    return x * x;
}
```

Syntactic error:

2.
```
int double square(int x) {
    return x * x;
}
```

Static-semantic error:

3.
```
boolean square(int x) {
    return x * x;
}
```

Runtime error:

4.
```
int p(int x) {
    return x / 0;
}
```

Logic error:

5.
```
int square(int x) {
    return 2 * x;
}
```

Safe versus unsafe languages

- **Safe language**

  All runtime errors are caught by the generated code and/or runtime system, and are reported in terms of the language.

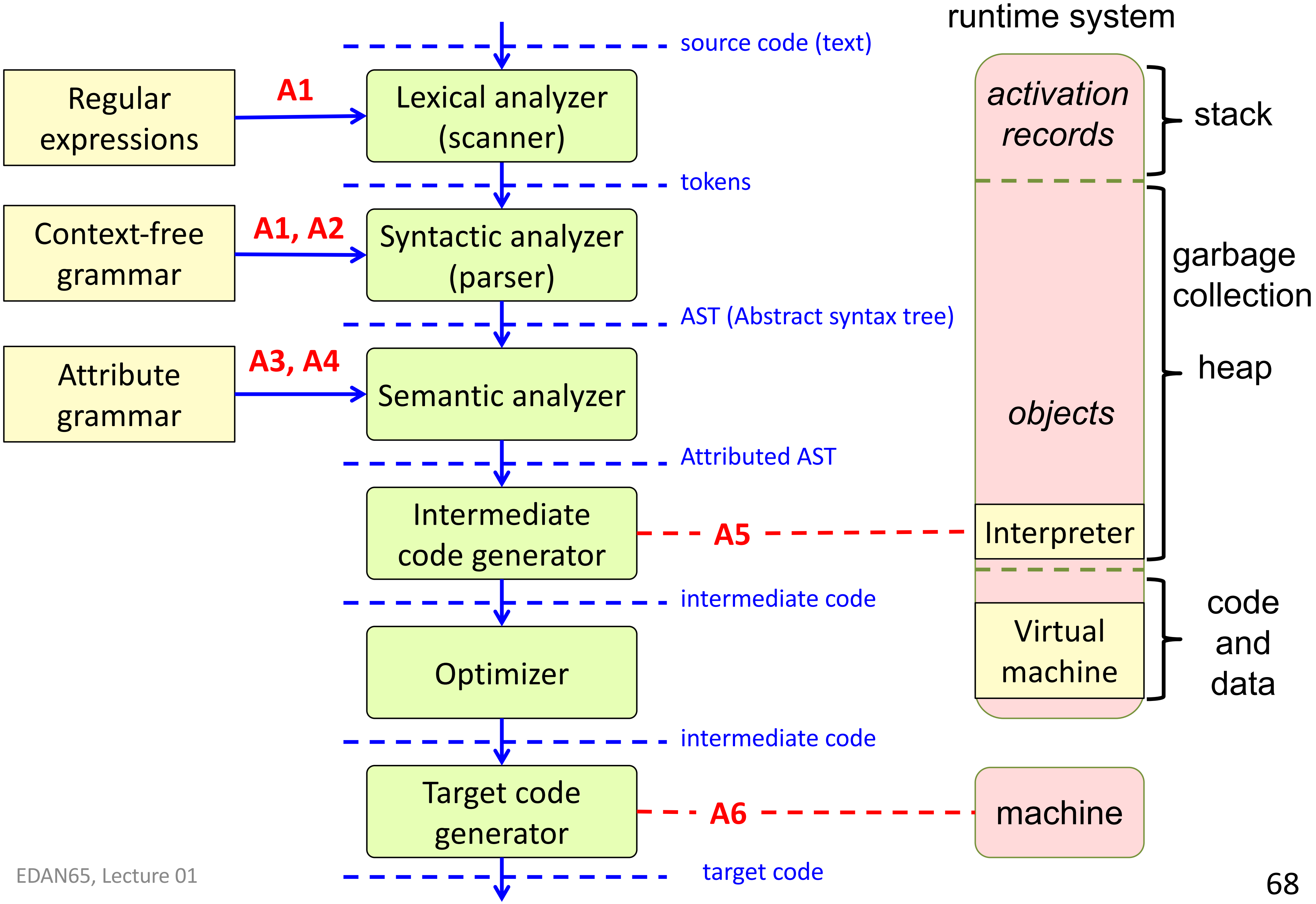
  Examples: Java, C#, Smalltalk, Python, ...

- **Unsafe language**

  Runtime errors in the generated code can lead to undefined behavior, for example an out of bounds array access. In the best case, this gives a hardware exception soon after the real error, stopping the program ("segmentation fault"). In the worst case, the execution continues, computing the wrong result or giving a segmentation fault much later, leading to bugs that can be extremely hard to find.

  Examples: C, Assembly

Course overview



After this course...

- You will have built a complete compiler
- You will have seen new declarative ways of programming
- You will have learnt some fundamental computer science theory
- You will have experience from using several practical tools

Some related courses

- EDAN70, **Project in Computer Science**, lp2
  - Build a small tool, evaluate it, write a short paper
- EDAN90, **Advanced Project in Computer Science**, lp2
  - Prerequisite: Project in CS (EDAN70)
- EDAP15, **Program Analysis**, lp2
  - Prerequisite: Compiler assignments (EDAN65)
- EDAN75, **Optimizing compilers**, lp3

- **Master's thesis project** in compilers
  (related to research or industry)

Applications of compiler construction

- Traditional compilers from source to assembly
- Source-to-source translators, transpilers, preprocessors
- Interpreters and virtual machines
- Integrated programming environments
- Program analysis tools
- Refactoring and other program transformation tools
- Domain-specific languages

Examples of Domain-Specific Languages

HTML

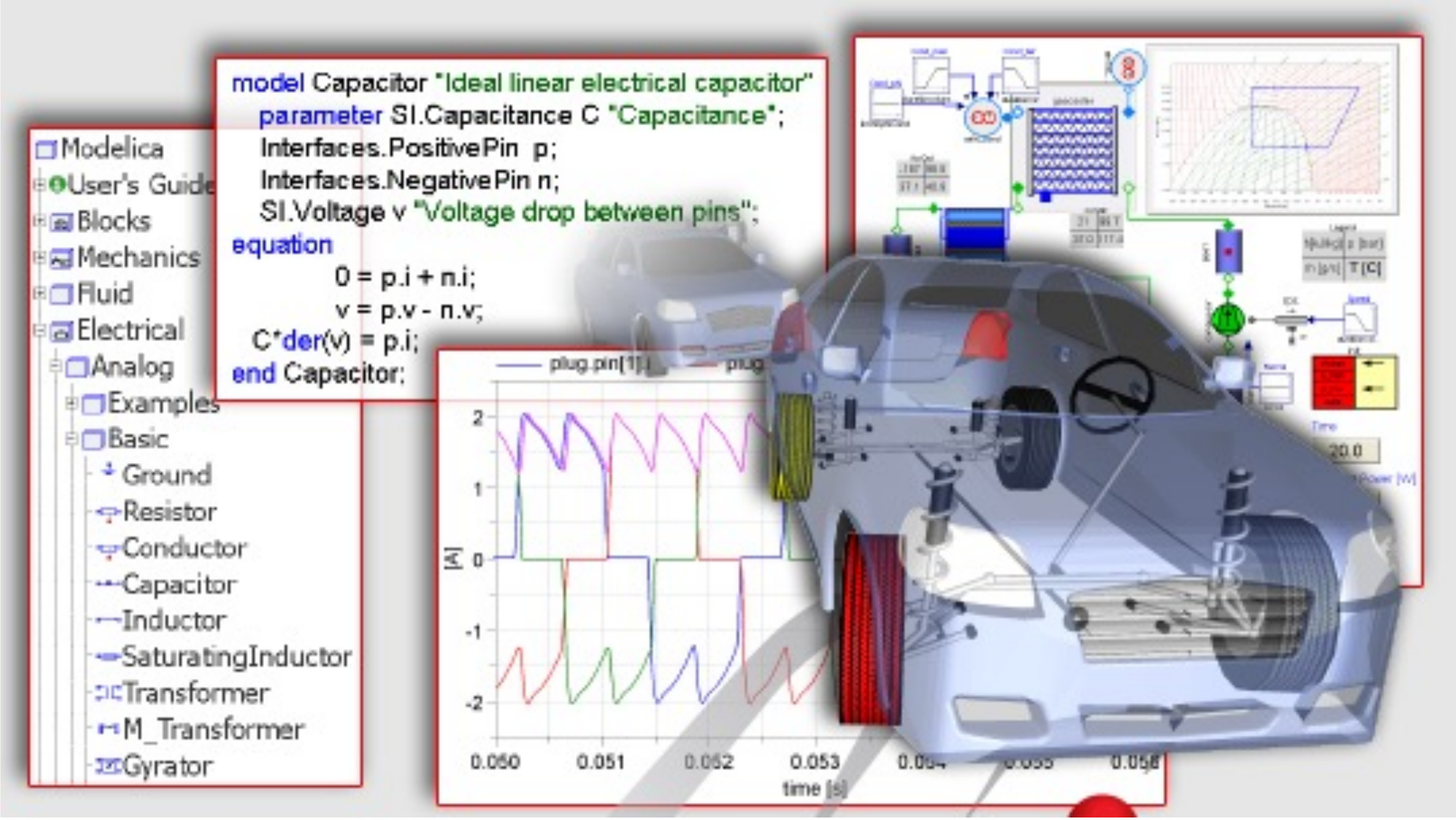
```
...
<h3>Lecture 1: Introduction. Mon 13-15. <a
href="http://fileadmin.cs.lth.se/cs/Education/EDAN65/2016/document
s/EDAN65-map.pdf">M:A</a></h3>
  <ul>
    <li><a
href="http://fileadmin.cs.lth.se/cs/Education/EDAN65/2016/lectures
/L01.pdf">Slides</a>
    <li>Appel Book: Ch 1-1.2
    <li><a href
="https://moodle2.cs.lth.se/moodle/mod/quiz/view.php?id=43">Moodle
Quiz</a>
  </ul>
...
```

.gitconfig

```
[user]
	name = Görel Hedin
	email = gorel.hedin@cs.lth.se
[push]
	default = simple
```

Modelica

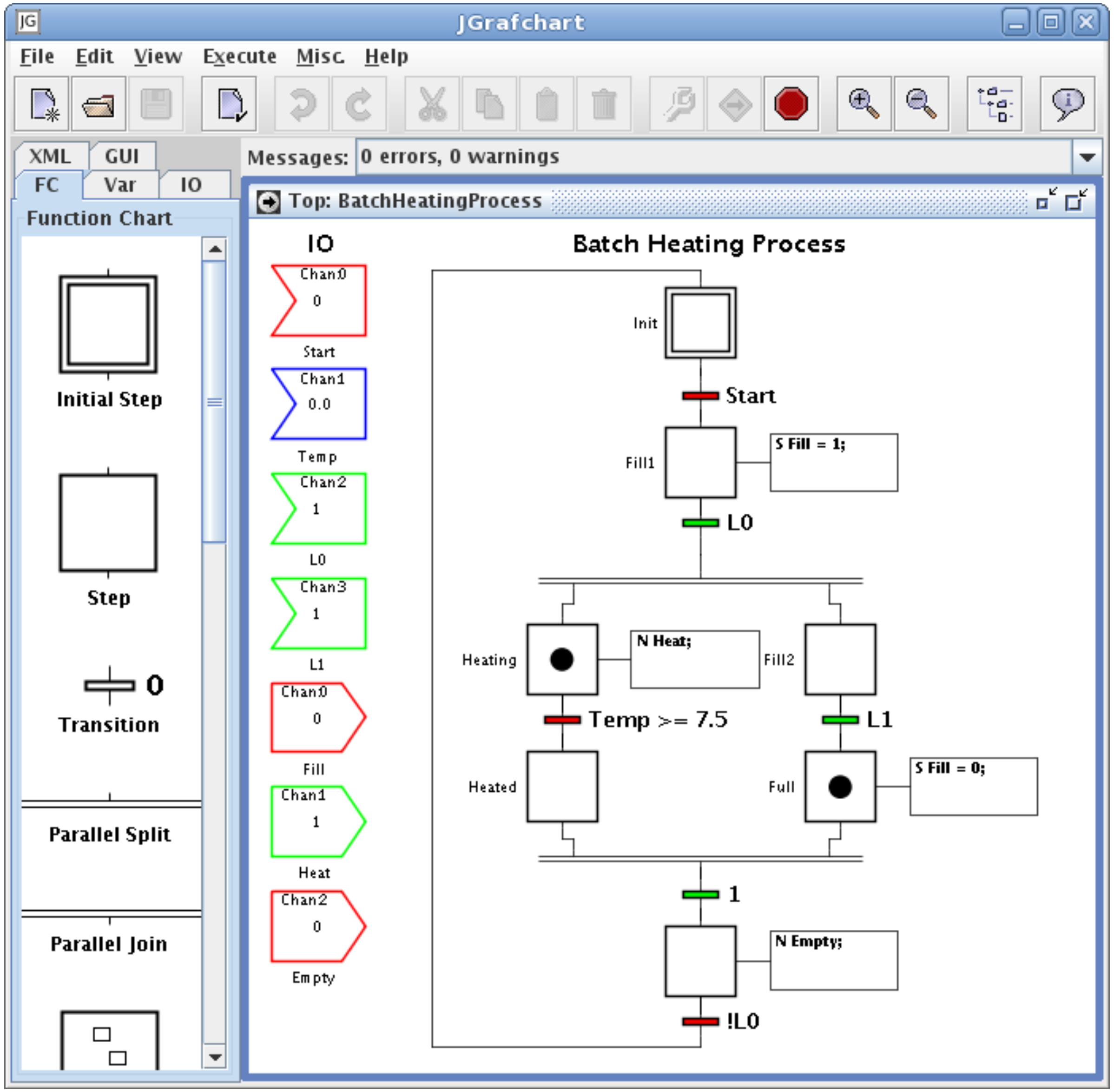
https://www.modelica.org/



```
model Capacitor "Ideal linear electrical capacitor"
   parameter SI.Capacitance C "Capacitance";
   Interfaces.PositivePin  p;
   Interfaces.NegativePin n;
   SI.Voltage v "Voltage drop between pins";
equation
           0 = p.i + n.i;
           v = p.v - n.v;
   C*der(v) = p.i;
end Capacitor;
```

Grafchart

https://www.control.lth.se/research/tools-and-software/grafchart.html



Messages: 0 errors, 0 warnings

Automation Builder

https://abb.com/automationbuilder

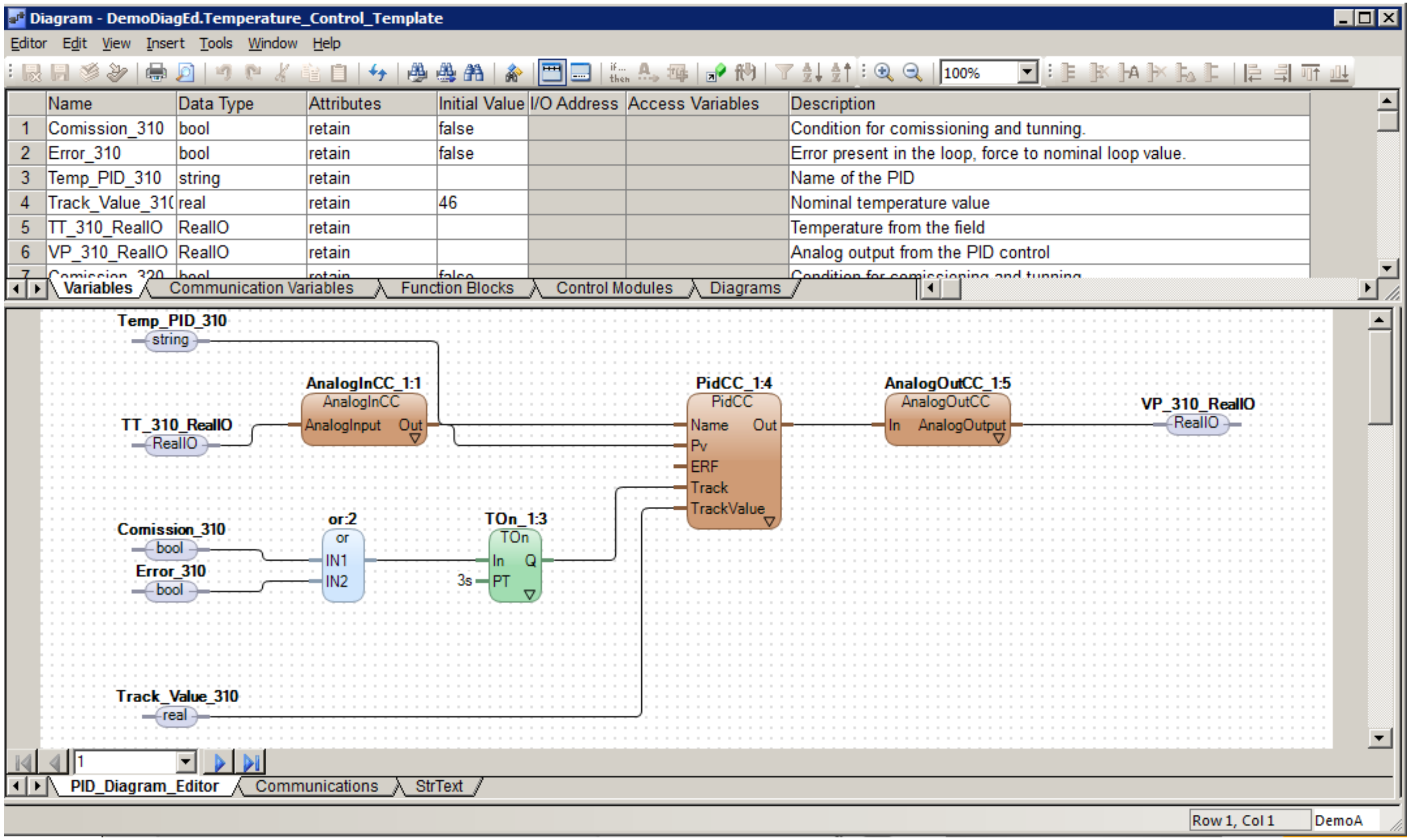


Diagram - DemoDiagEd.Temperature_Control_Template
Editor Edit View Insert Tools Window Help
100%

	Name	Data Type	Attributes	Initial Value	I/O Address	Access Variables	Description
1	Comission_310	bool	retain	false			Condition for comissioning and tunning.
2	Error_310	bool	retain	false			Error present in the loop, force to nominal loop value.
3	Temp_PID_310	string	retain				Name of the PID
4	Track_Value_31(	real	retain	46			Nominal temperature value
5	TT_310_RealIO	RealIO	retain				Temperature from the field
6	VP_310_RealIO	RealIO	retain				Analog output from the PID control
7	Comission_320	bool	retain	false			Condition for comissioning and tunning

Some related research at LTH

Language tools

- Extensible compiler tools (Görel Hedin)
- Program analysis, software tools (Christoph Reichenbach)
- Adaptive developer tools (Emma Söderberg)

Backend

- Real-time garbage collection (Roger Henriksson)
- Code optimization for multiprocessors (Jonas Skeppstedt)

NLP

- Natural language processing (Pierre Nugues)

Domain-specific languages

- Languages for stream computing (Jörn Janneck)
- Languages for robotics (Volker Krüger, Christoph Reichenbach)
- Languages for pervasive systems (Boris Magnusson, Görel Hedin)
- Languages for requirements modeling (Björn Regnell)
- Languages for simulation and control (The control department)

Summary questions

- What are the major compiler phases?
- What is the difference between the analysis and synthesis phases?
- Why do we use intermediate code?
- What is the advantage of separating the front and back ends?
- What is
    - a lexeme?
    - a token?
    - a parse tree?
    - an abstract syntax tree?
    - intermediate code?
- What is the difference between assembly code, object code, and executable code?
- What is bytecode, an interpreter, a virtual machine?
- What is a JIT compiler?
- What kind of errors can be caught by a compiler? A runtime system?

See course website `https://cs.lth.se/edan65` for what to do this week.