

EDAN65: Compilers

Introduction and Overview

Görel Hedin

Revised: 2020-08-30

Instructors

- Lectures
 - Prof. Görel Hedin
- Programming assignments and lab sessions
 - Researcher Jesper Öqvist
 - Ph. D. student Alfred Åkesson
 - Ph. D. student Idriss Riouak

Course registration

- Confirm that you will take the course by signing in to Moodle by **today**, Aug 31, 2020
- Register in LADOK, by **today**, Aug 31, 2020
- Sign up for lab session, by **Thursday, Sep 3**
- (PhD students: Sufficient to sign in to Moodle)

- Prerequisites
 - Object-oriented programming and 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:** <http://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, lab coordination, on-line quizzes.

Course structure

- **Lectures**, Mon 10-12, Tue 10-12
- **Assignment 0**, for freshening up on Unix, Java, and JUnit, and understanding the build system Gradle. Do on your own.
- **Assignment 1-6**. Mandatory.
 - Work in pairs. Use chat to form pairs. Or use the forum.
 - Heavy. Get approved and get help at Lab sessions.
 - Thu 8-10, 10-12, 15-17. **Sign up by Thursday Sep 3**
 - Lab sessions start next week (but start this week on your work)
 - Assignments prerequisite for doing exam
- **Lecture quizzes**
 - Do on your own. (In Moodle.)
- **Exercises**
 - Do on your own. (Exercises in the book + separate exercises with solutions)
 - Tick off in Moodle, using the "progress bar"
- **Exam** – sign up in advance through the LTH system
 - Exam: Wednesday, Oct 28, 2020, 8-13. Planned to be on campus.
 - Retake: Wednesday, April 7, 2021, 8-13.

Working with the assignments

- Work *together* with your partner on all parts
 - pair programming, switch *frequently* who is typing
 - you need hands-on experience from all parts
 - both of you should be able to run and explain all parts of your solution
- Recommendation: use a git repo for collaboration with your partner.
 - But make it private!
 - Free private plans available on, e.g., Github, Gitlab, Bitbucket
- If you get stuck
 - ask on the forum
 - you are encouraged to give answers to other students on the forum (for general advice, not solutions)
- Experimental "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	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).

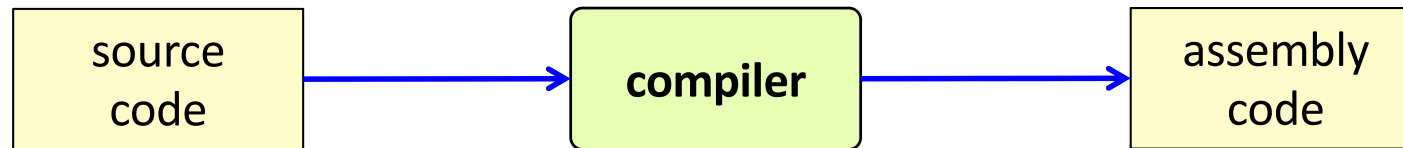
Q & A

Write questions in chat

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

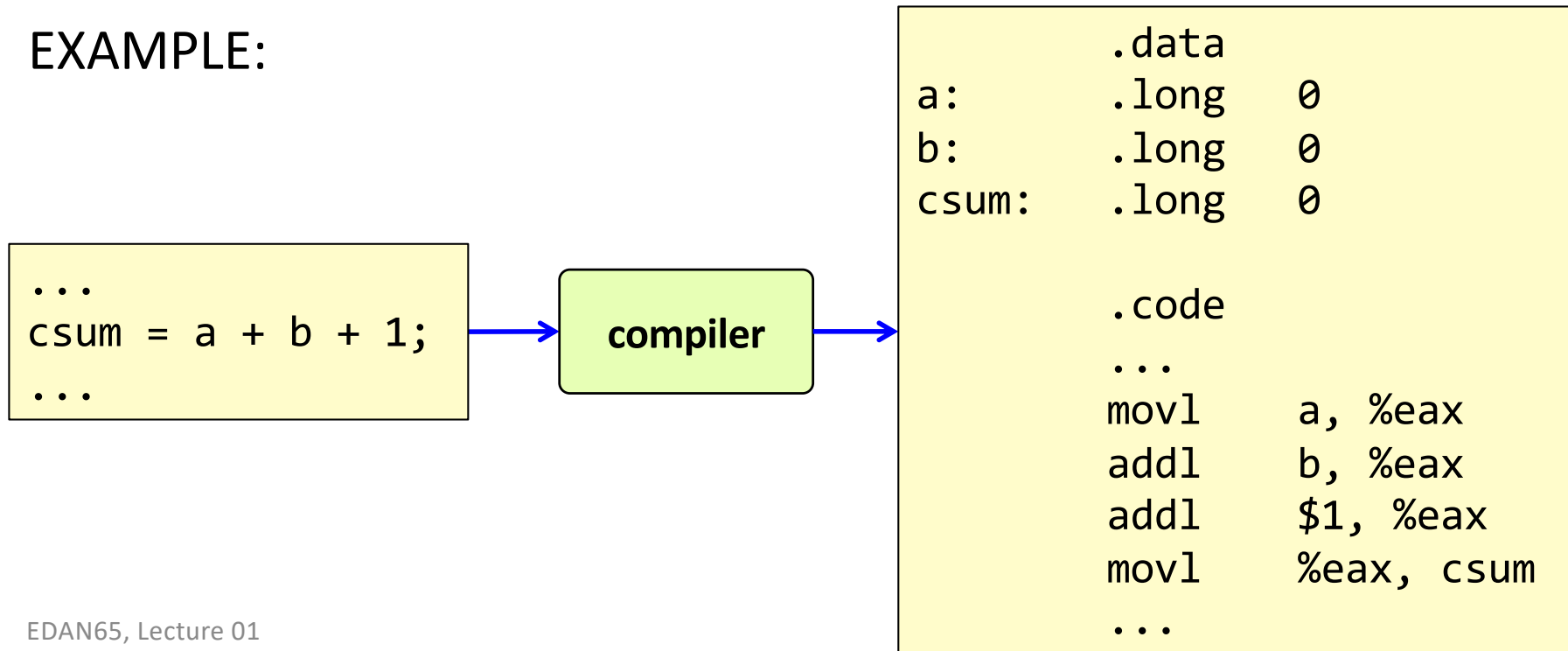
A traditional compiler



A traditional compiler



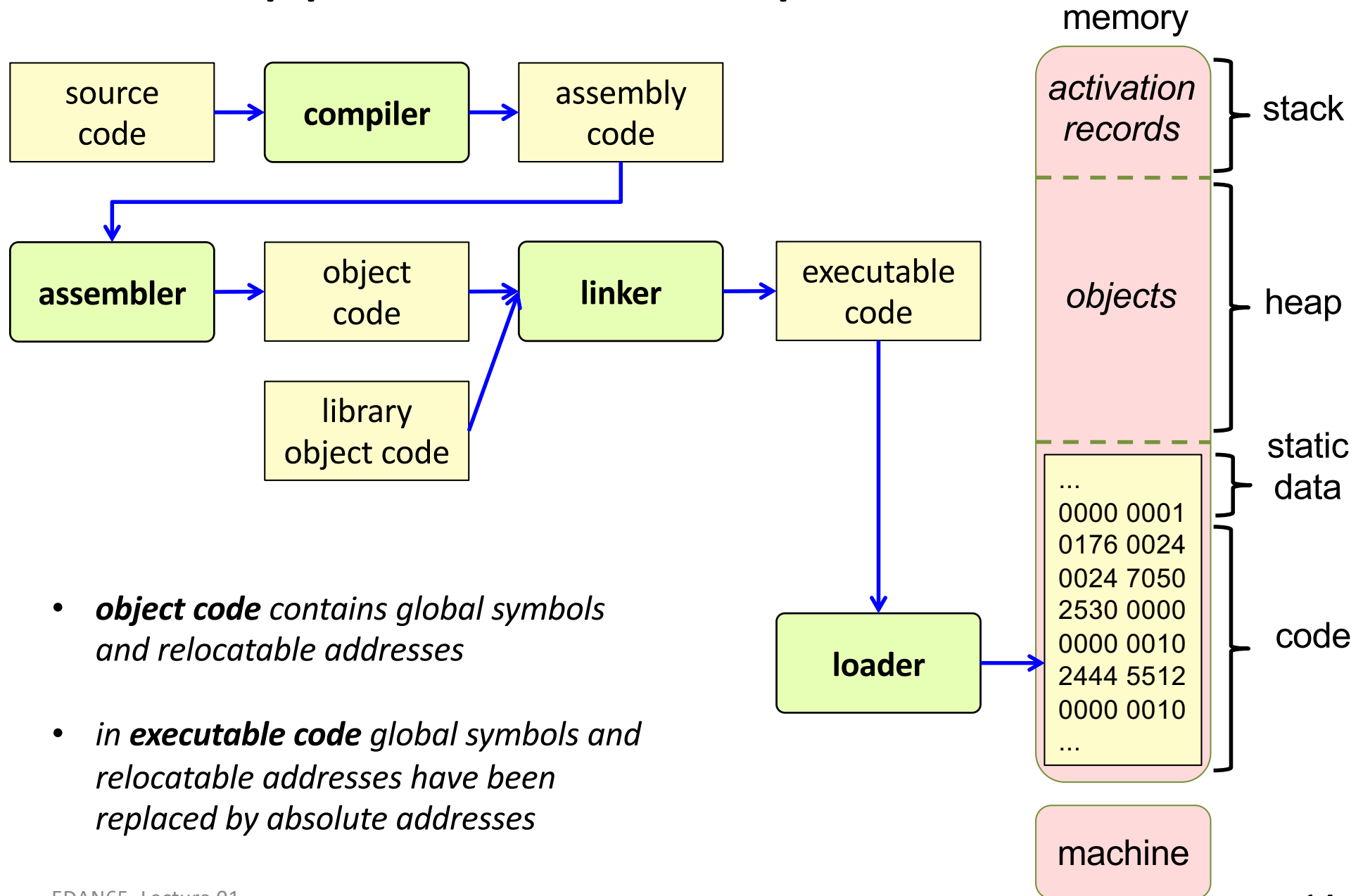
EXAMPLE:



What happens after compilation?



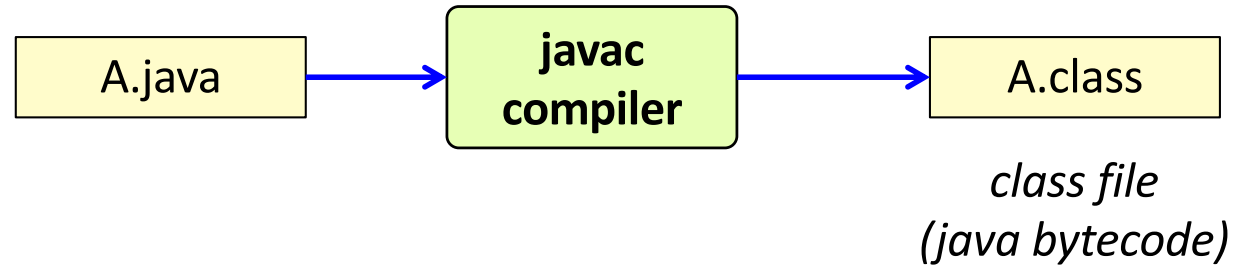
What happens after compilation?



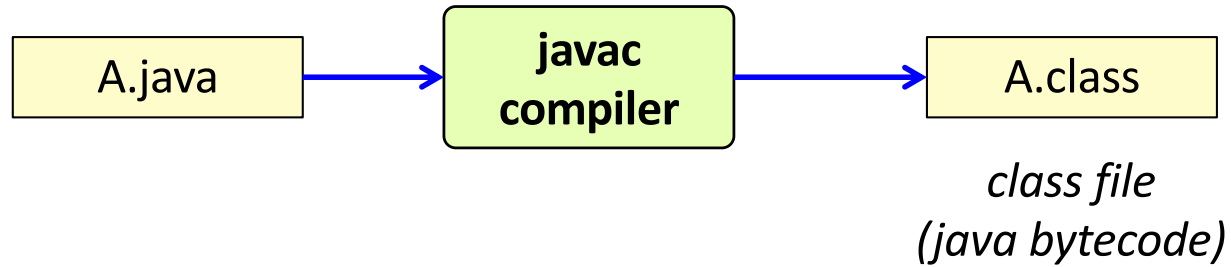
- **object code** contains global symbols and relocatable addresses
- in **executable code** global symbols and relocatable addresses have been replaced by absolute addresses

What about Java?

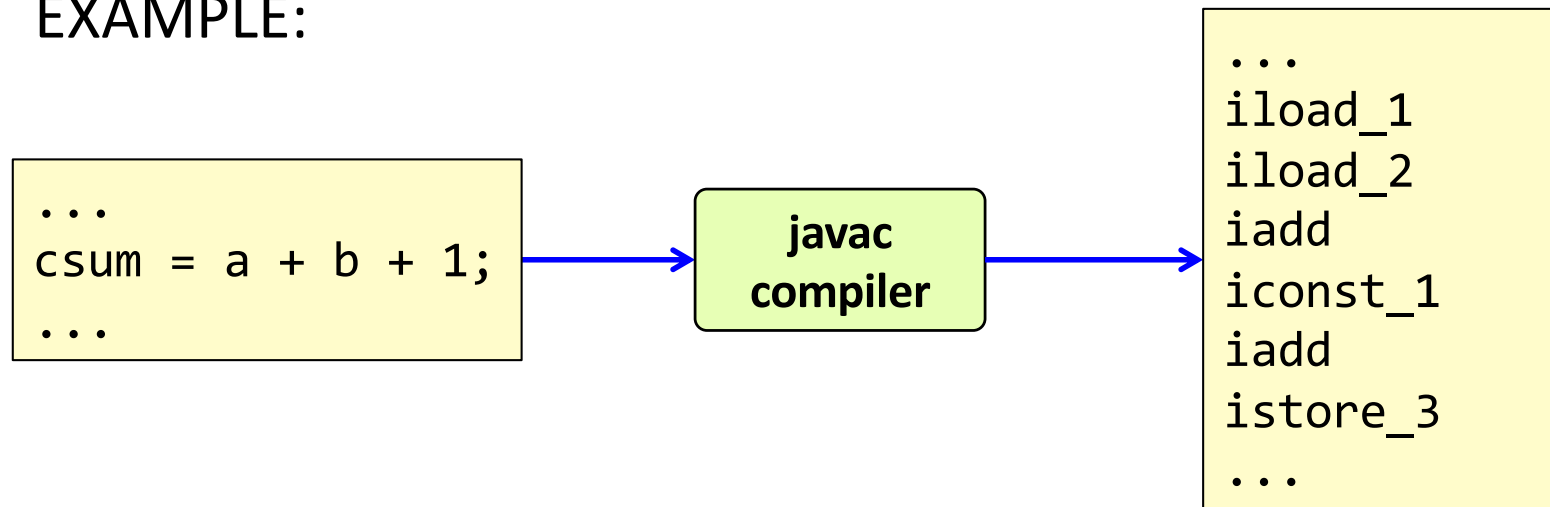
What about Java?



What about Java?



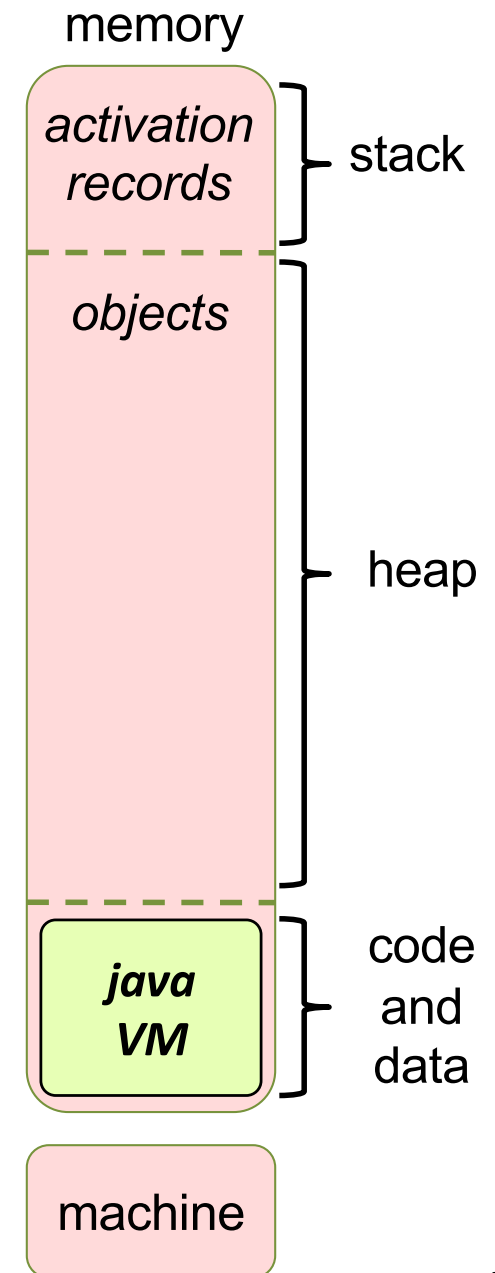
EXAMPLE:



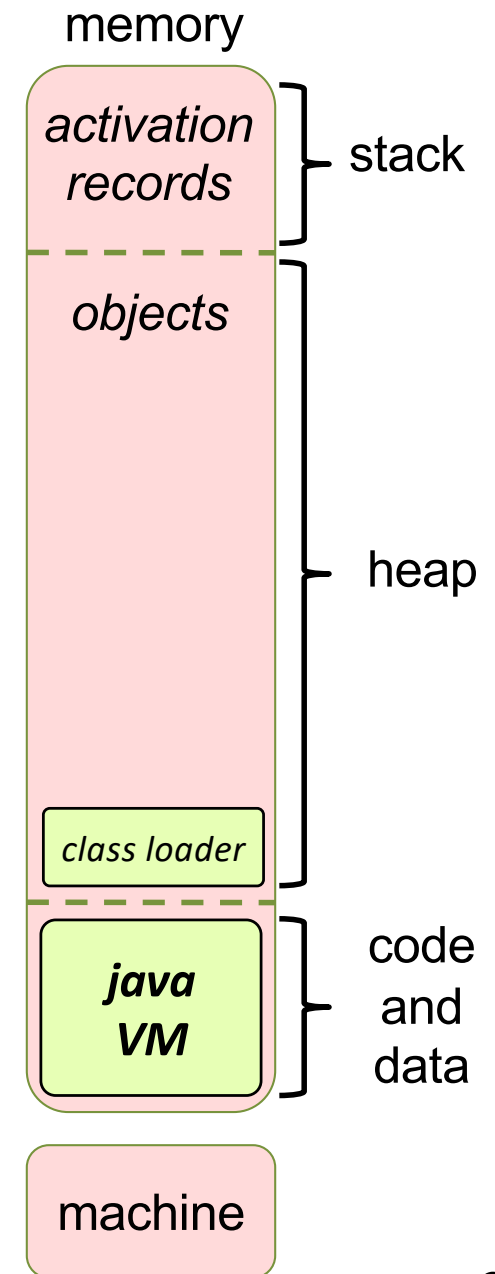
Running Java code?



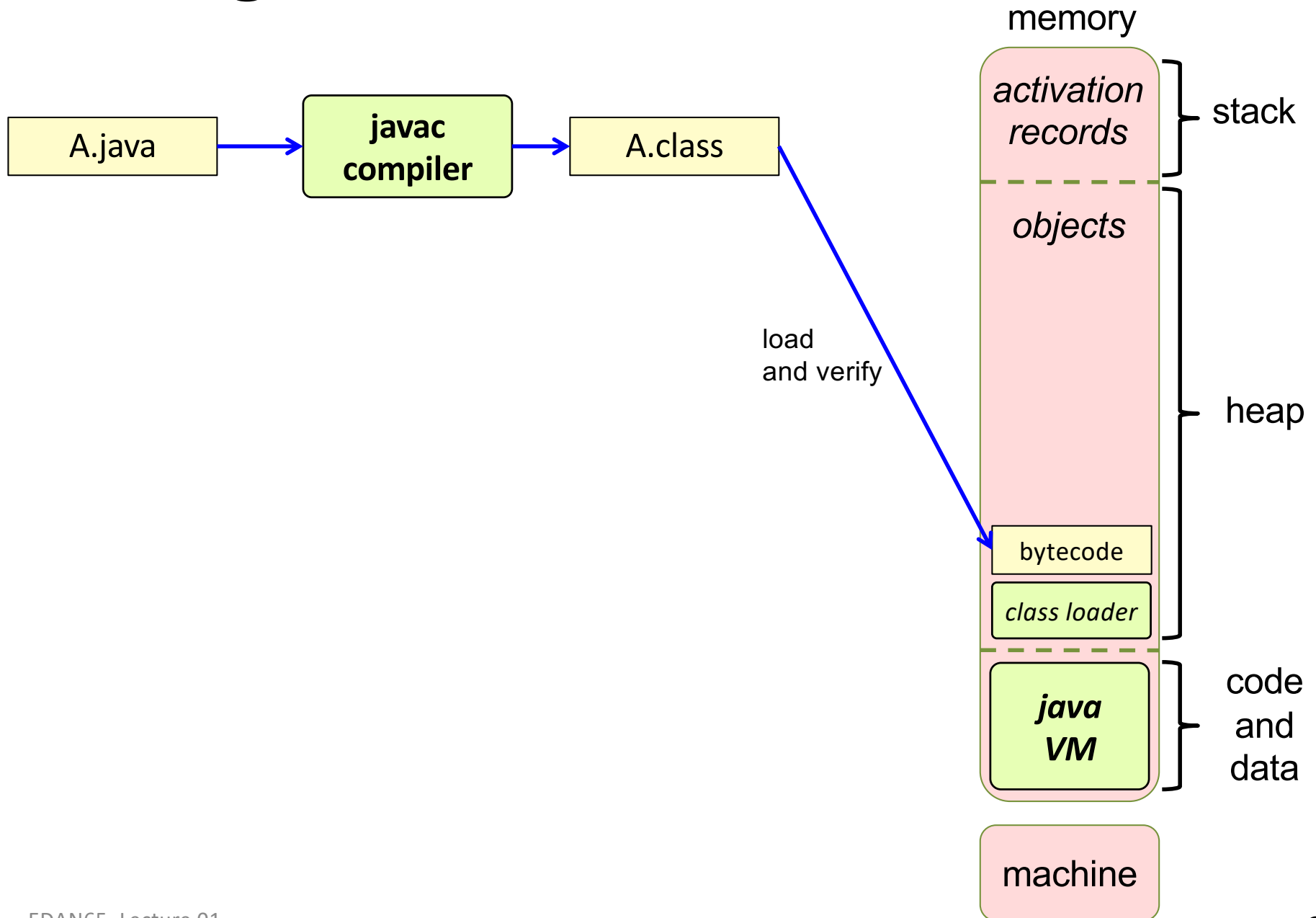
Running Java code?



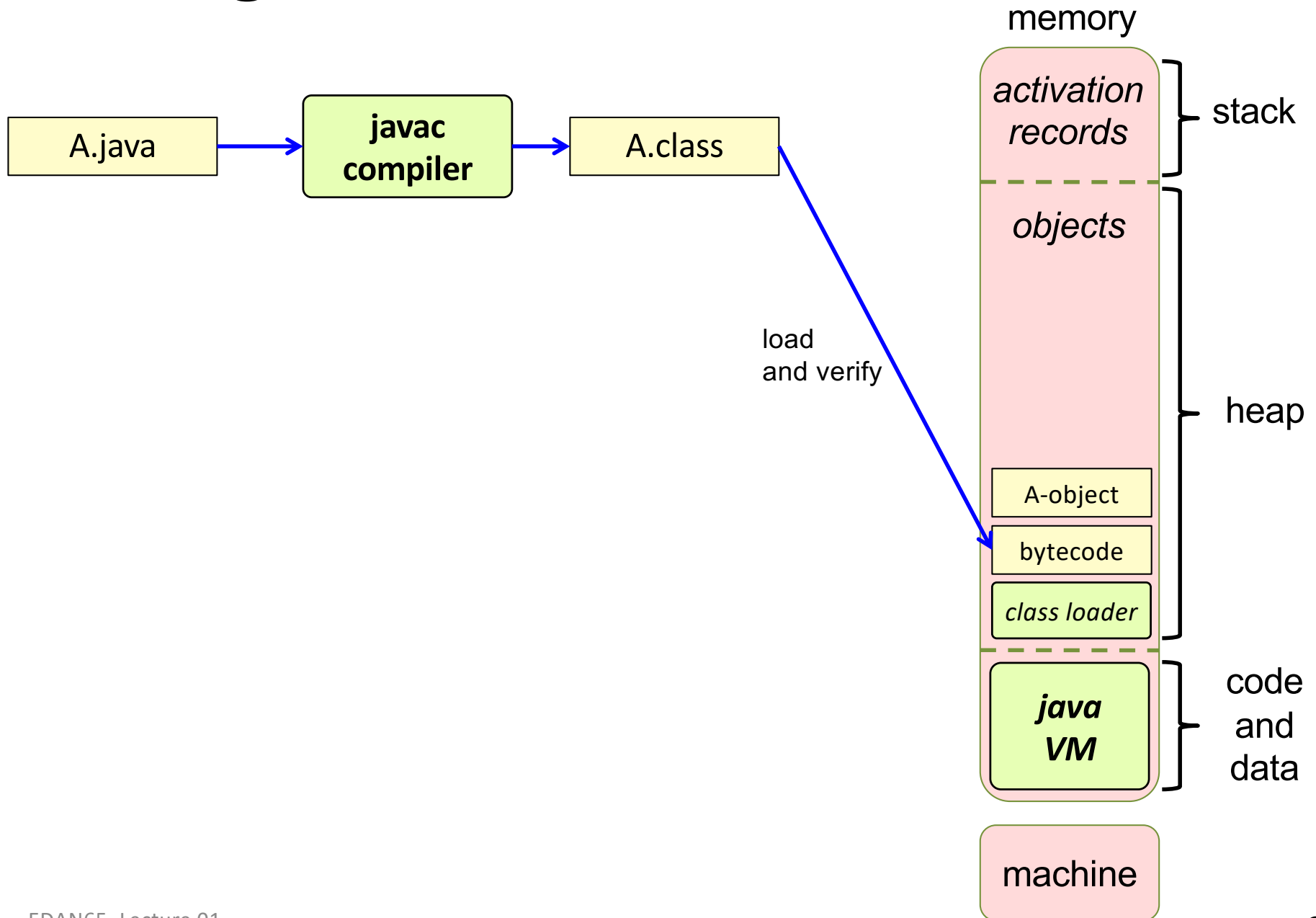
Running Java code?



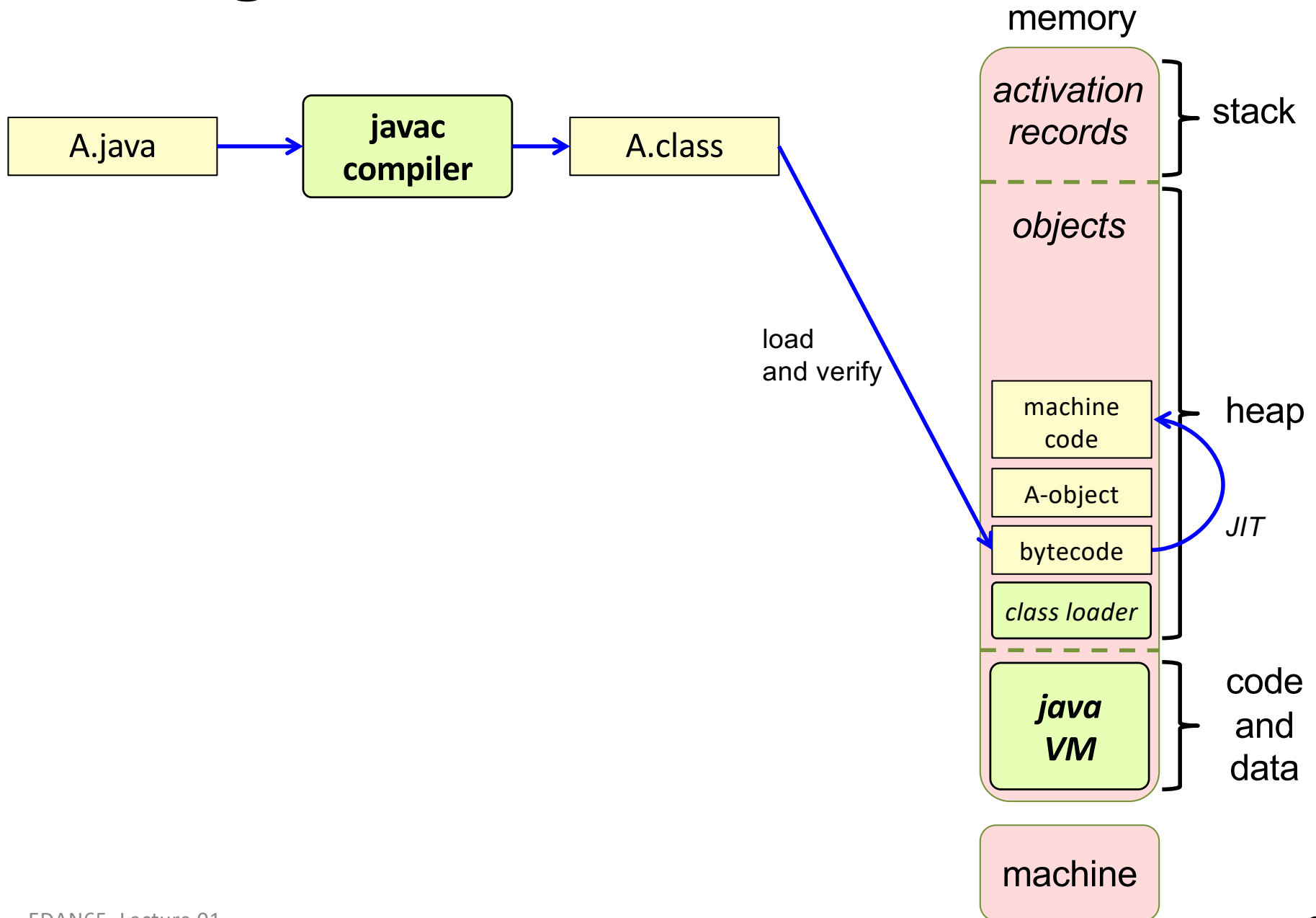
Running Java code?



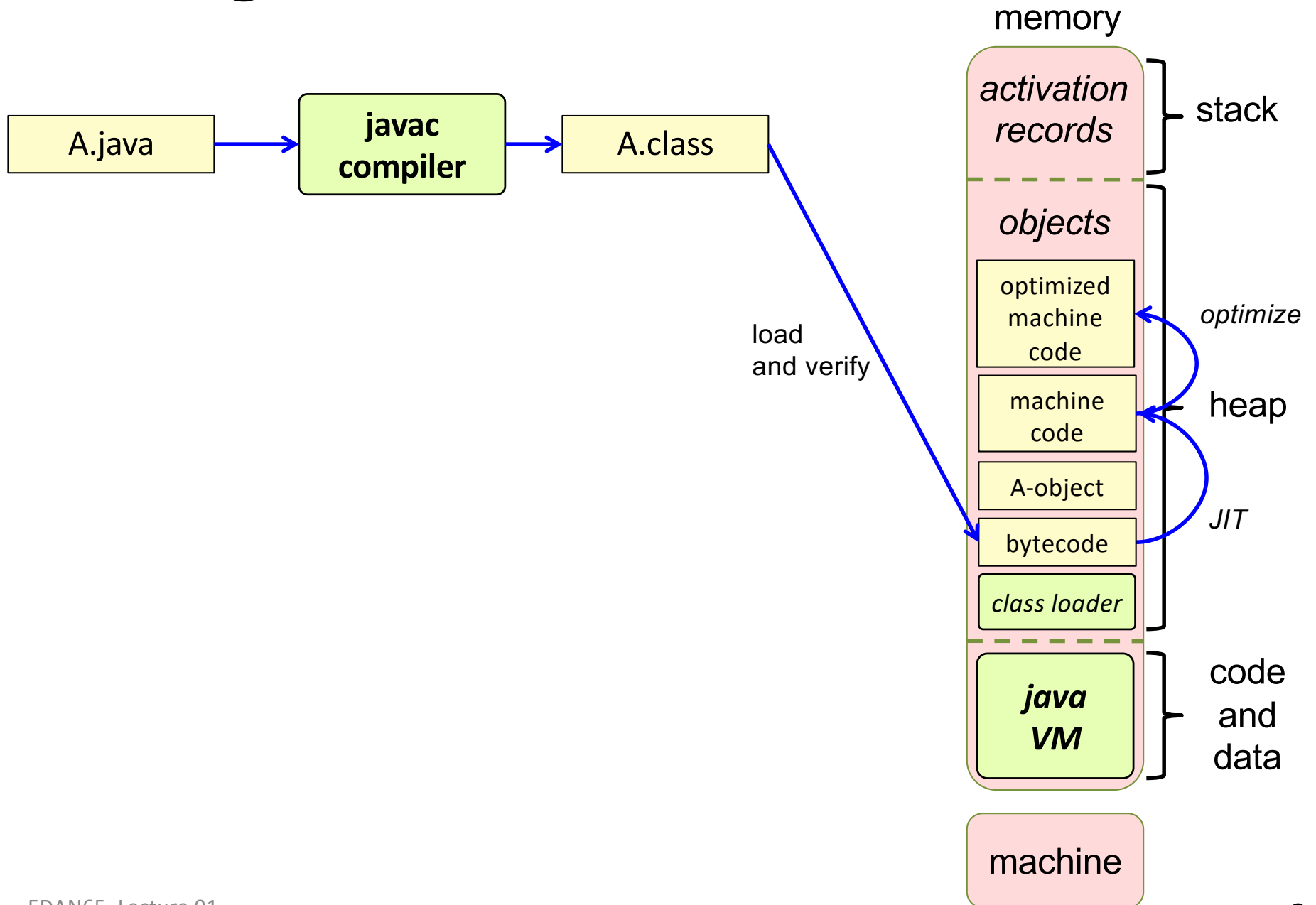
Running Java code?



Running Java code?



Running Java code?



Running Java code?

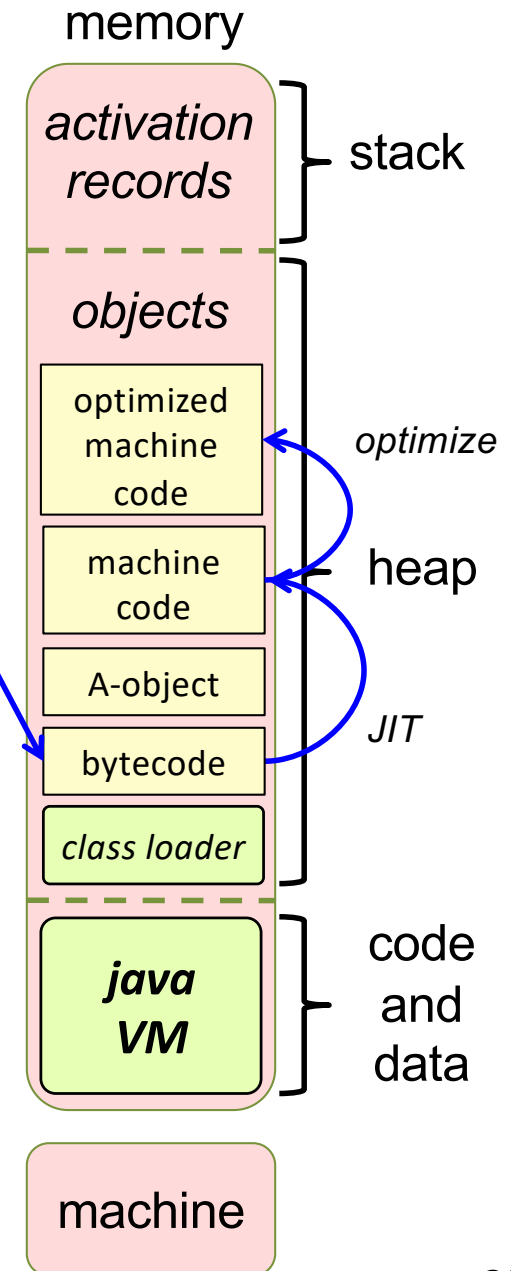


load and verify

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

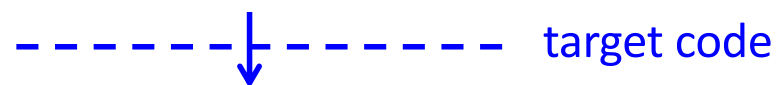
It can:

- 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



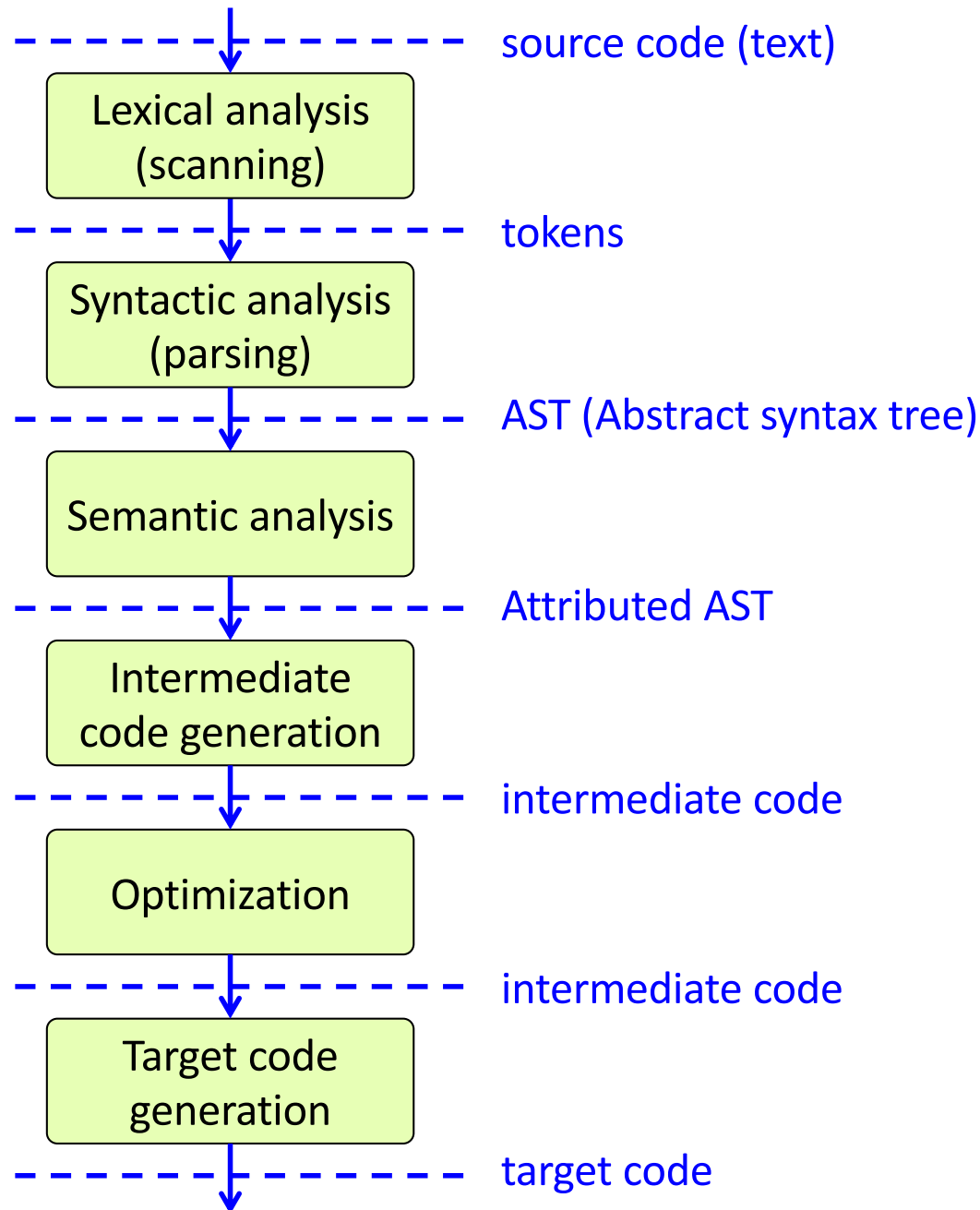
Inside the compiler:

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



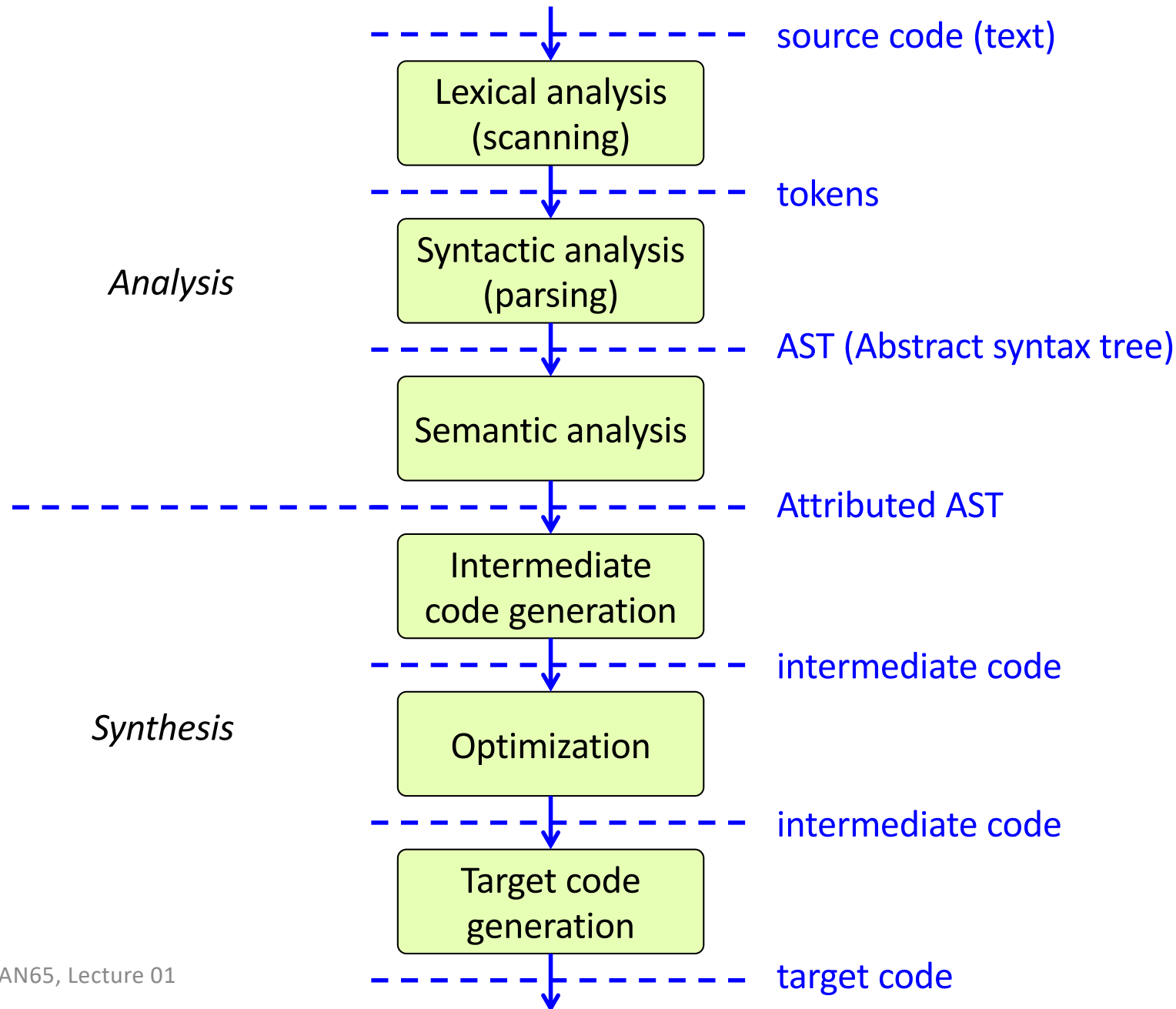
Inside the compiler:

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

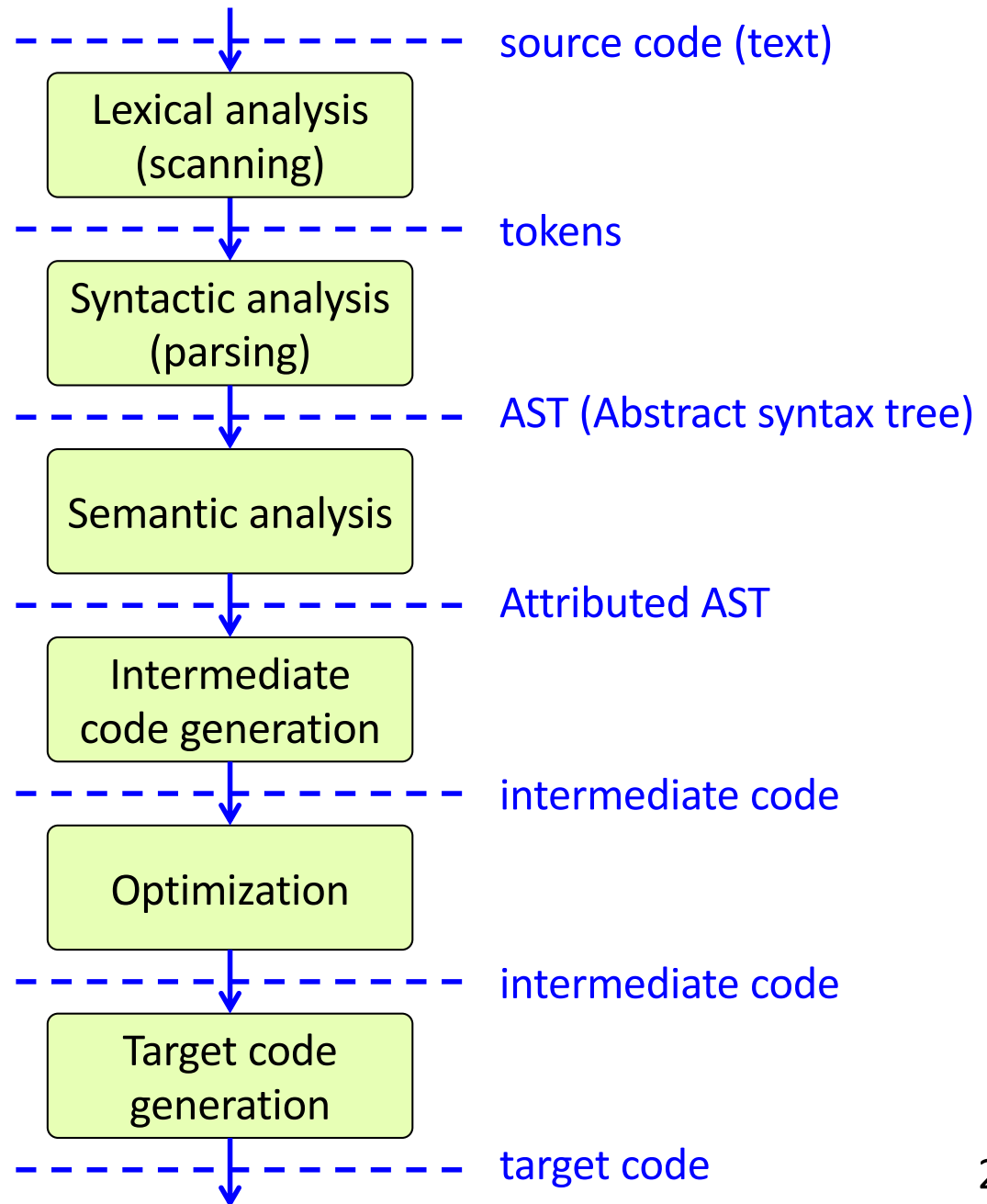


Inside the compiler:

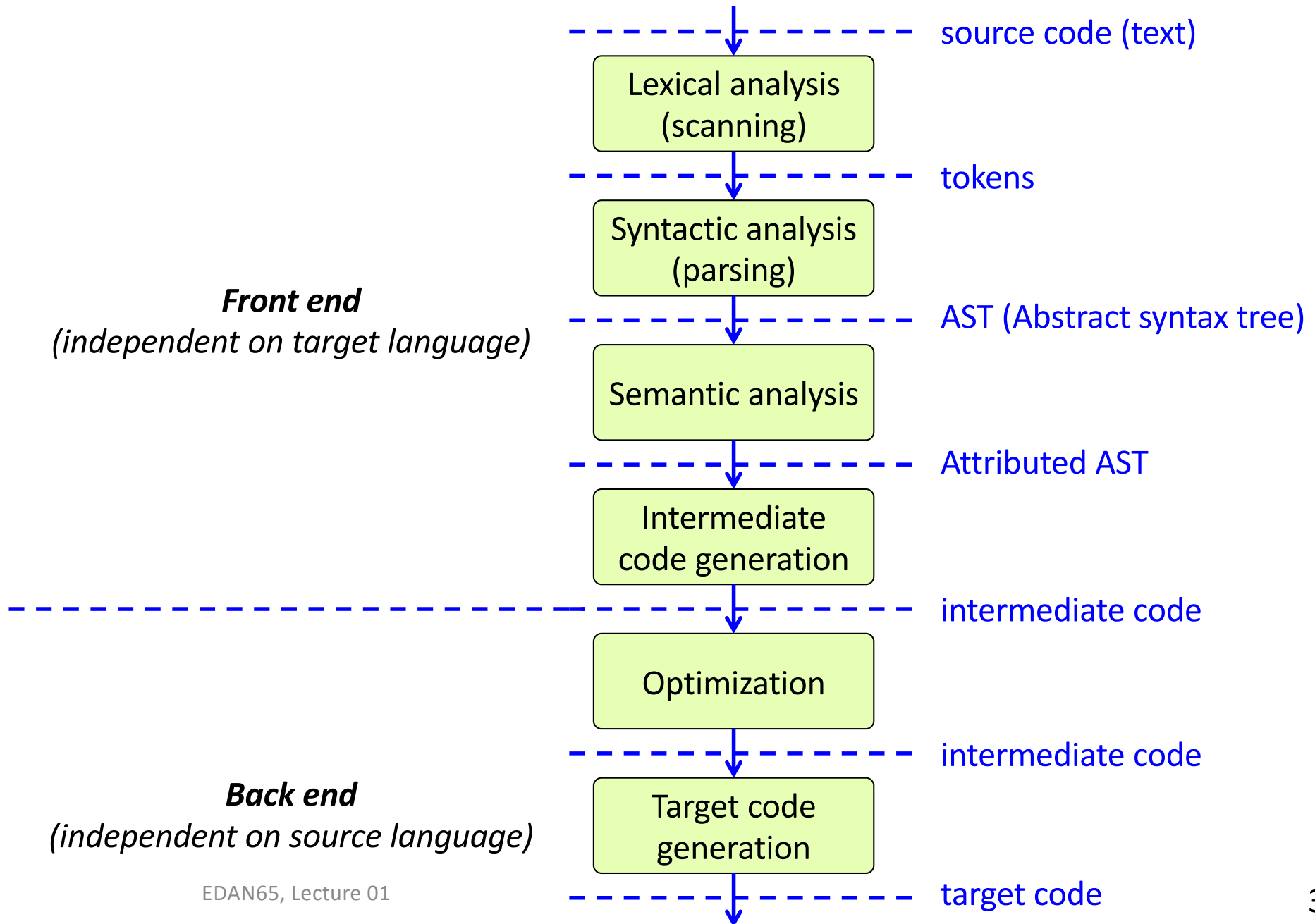
Each *phase* converts the program from one *representation* to another



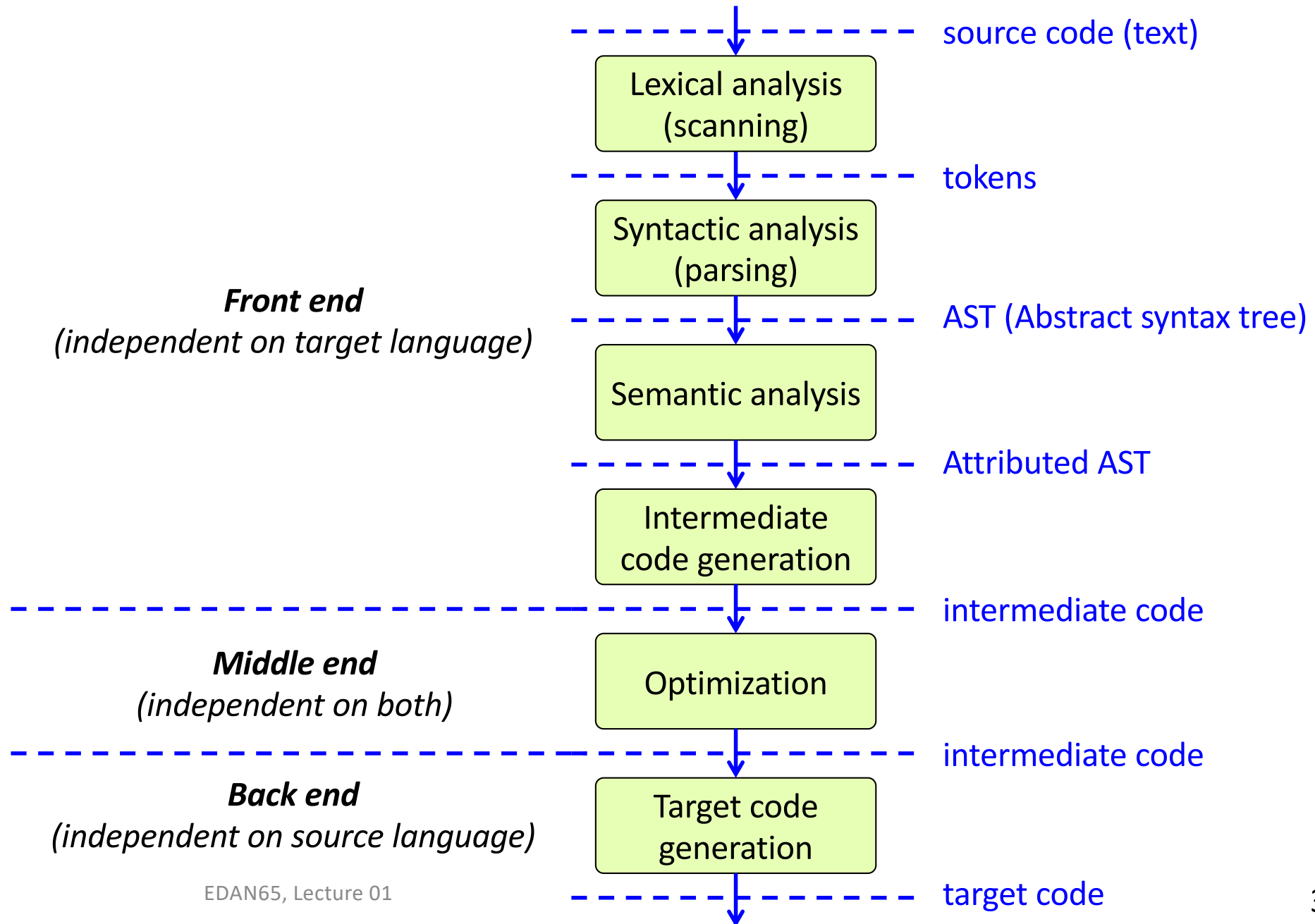
Front and back end:



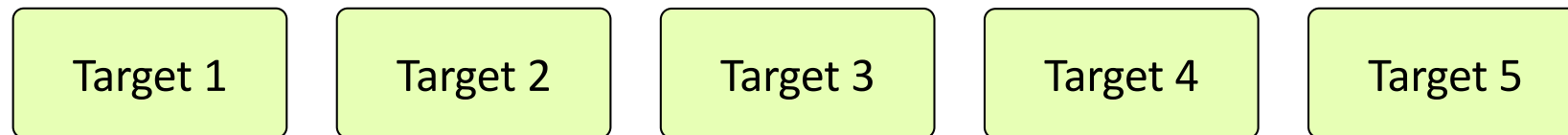
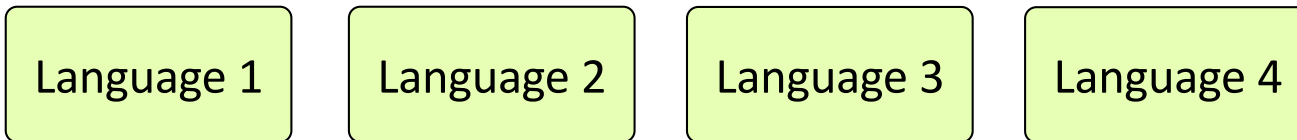
Front and back end:



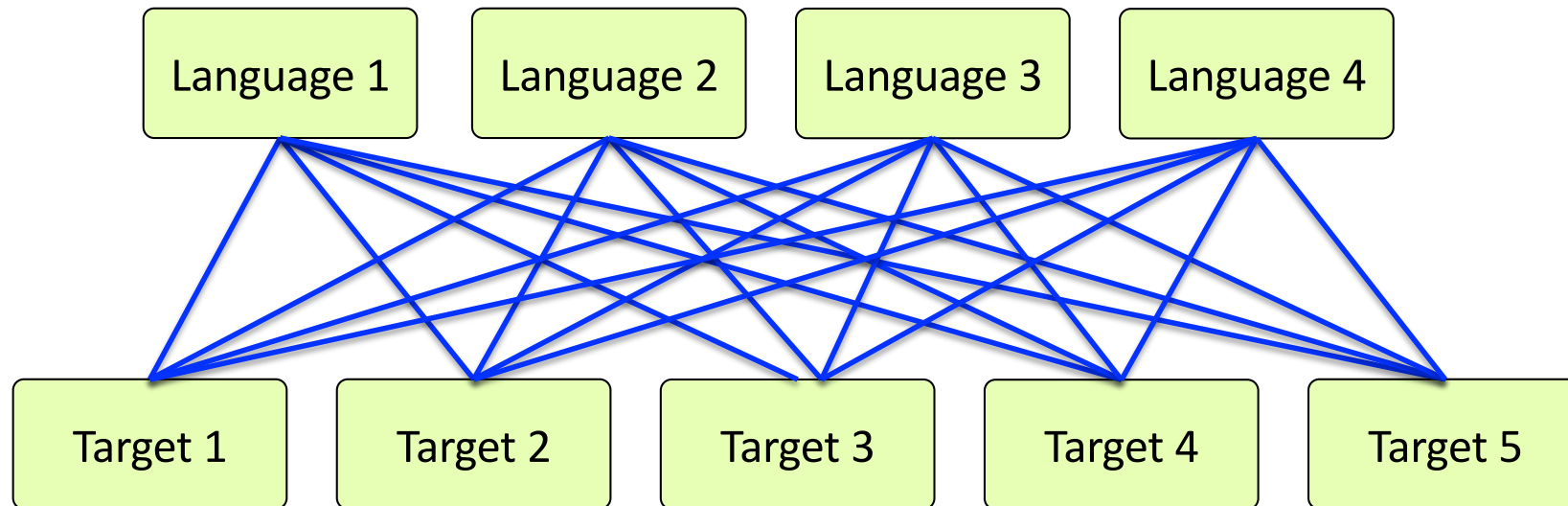
Front and back end:



Multiple languages and target machines



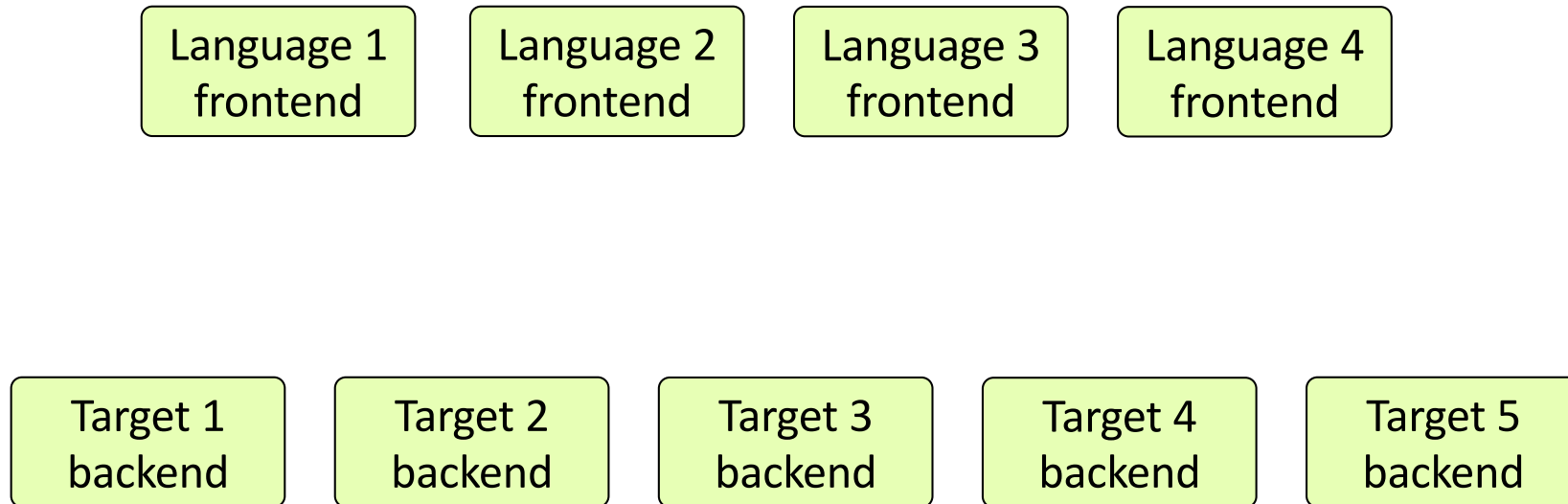
Multiple languages and target machines



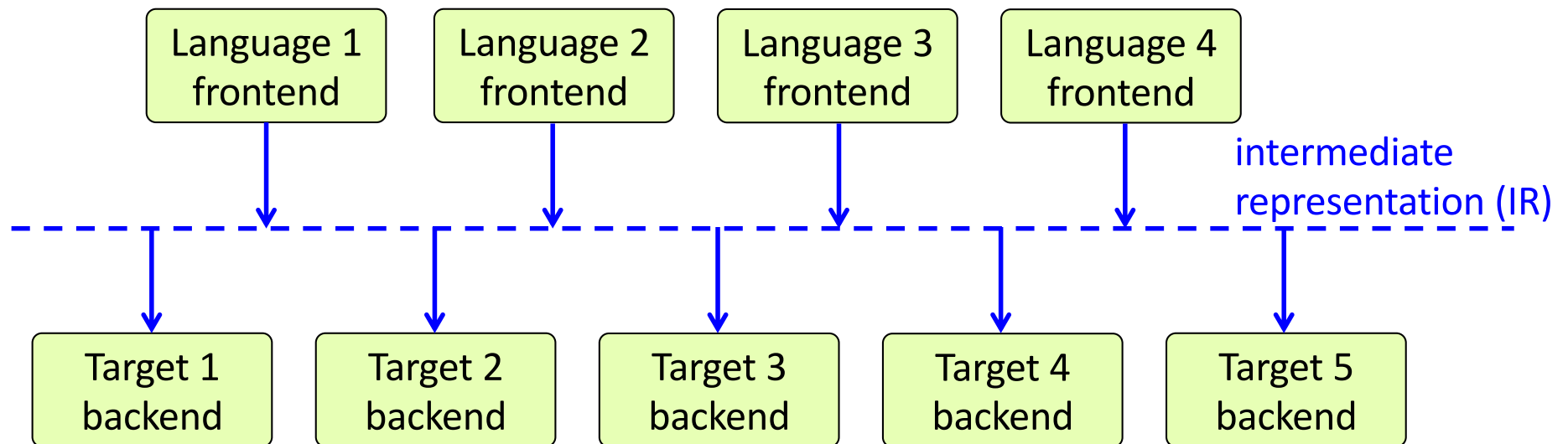
m languages
n target machines
 $m \cdot n$ compilers

Is there a smarter way?

Multiple languages and target machines

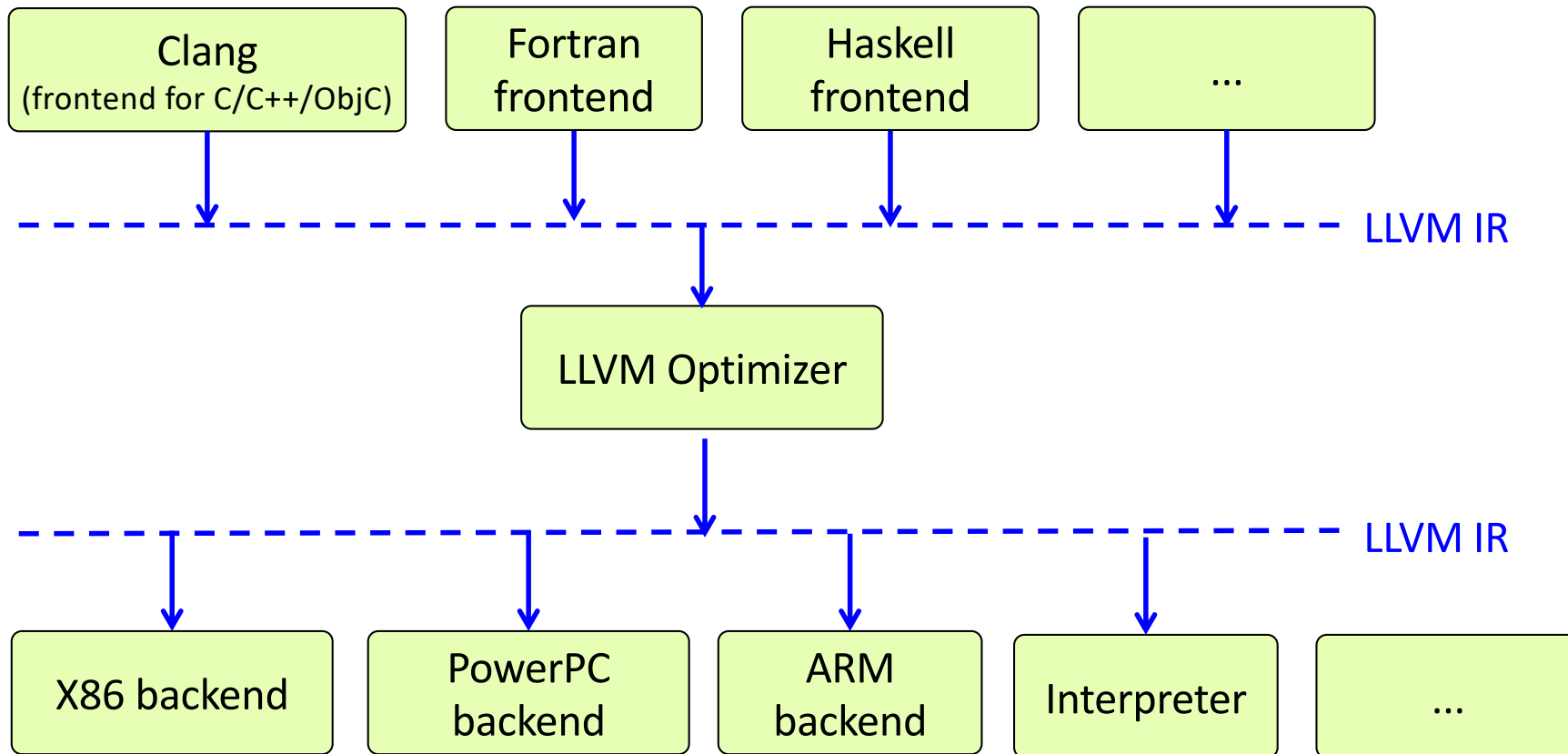


Multiple languages and target machines

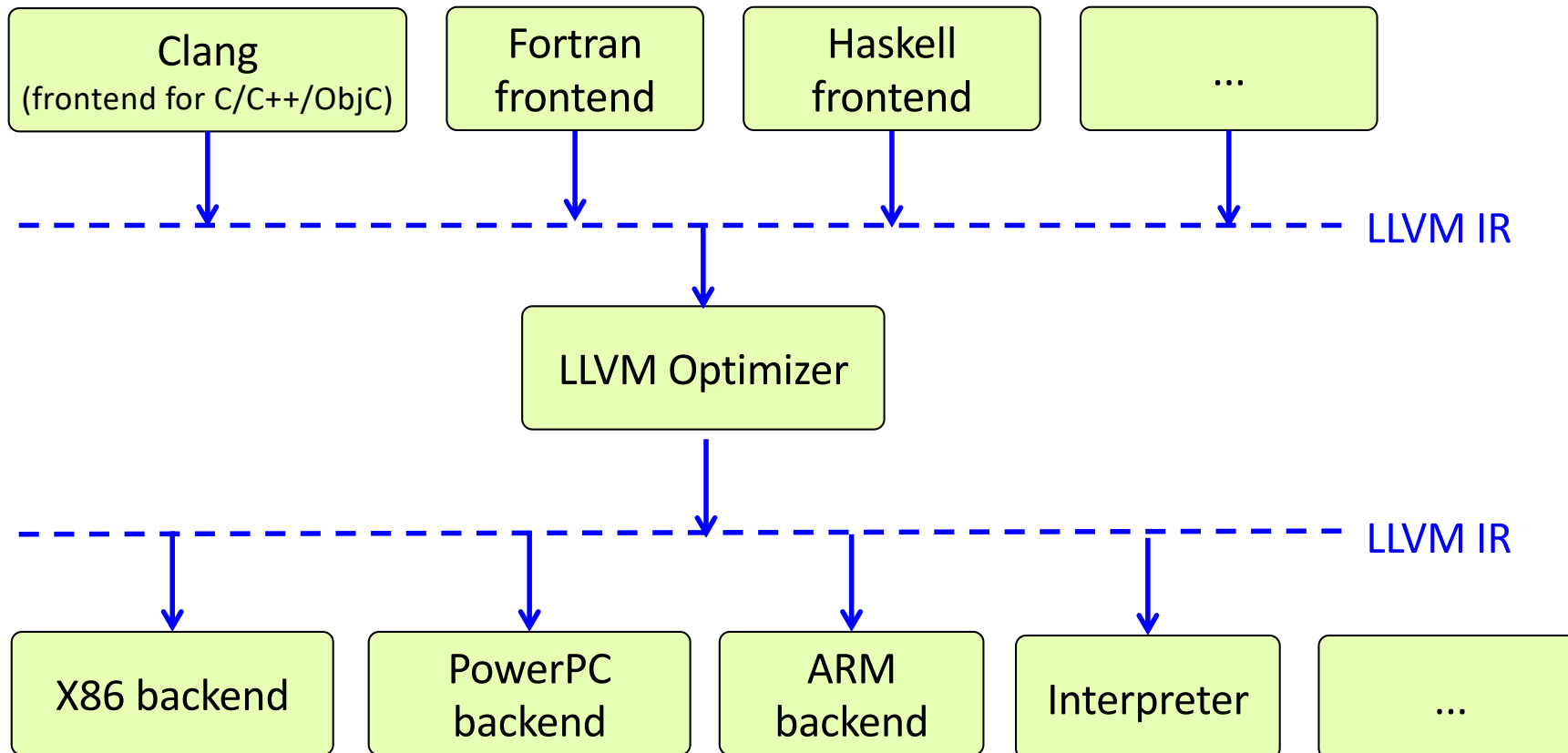


m language frontends
n target backends
m + n components

LLVM example



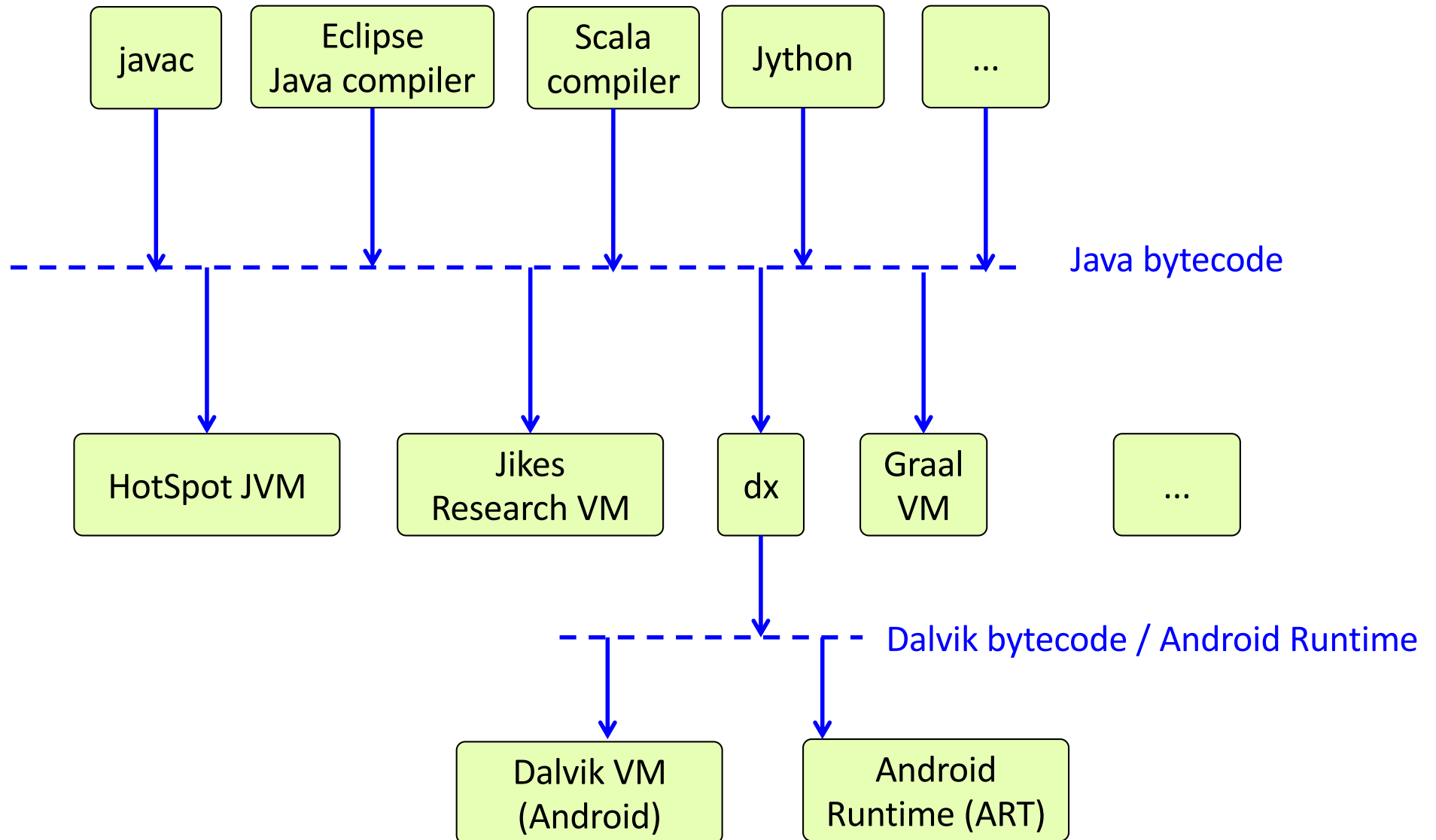
LLVM example



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:



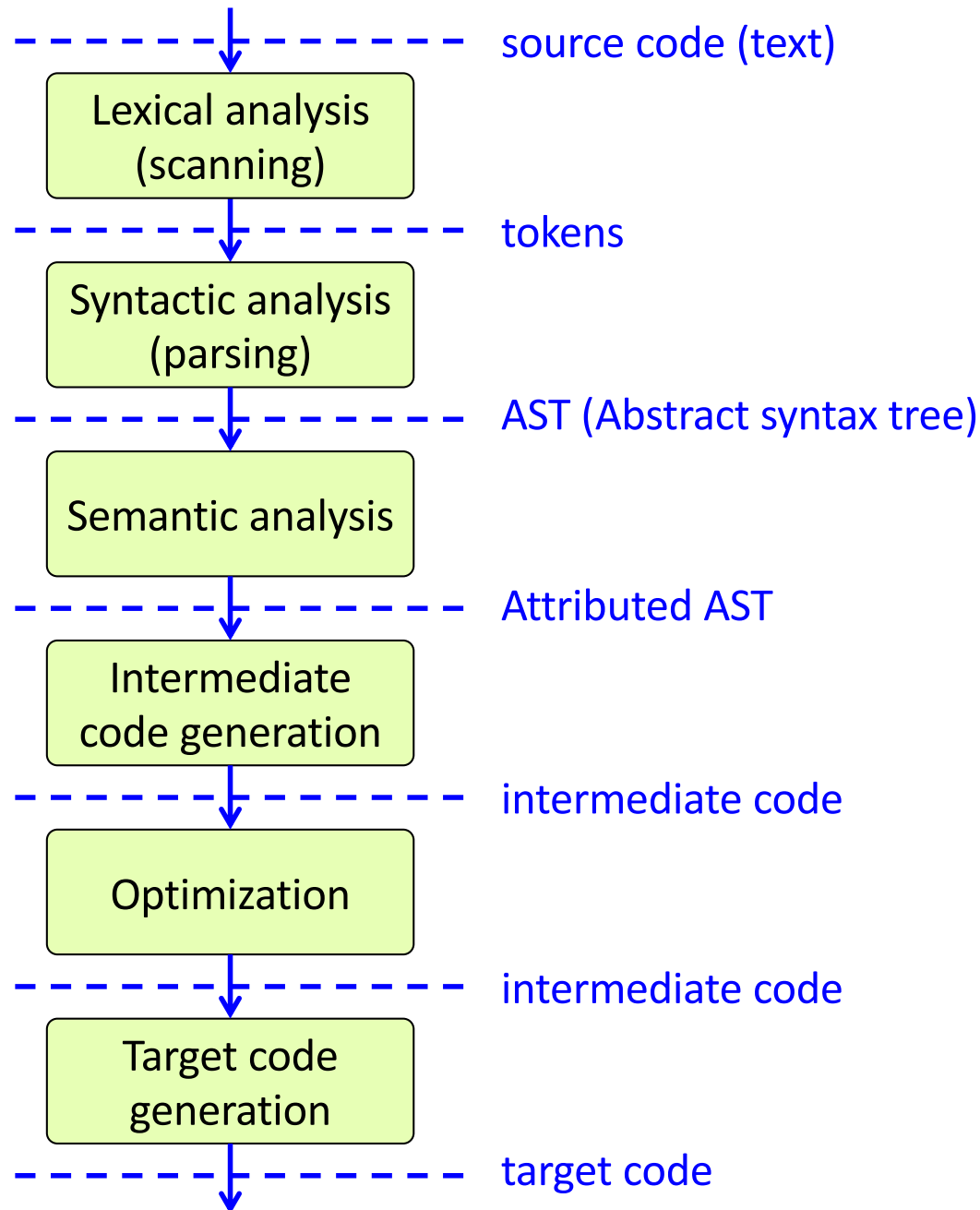
Some terminology

- 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.

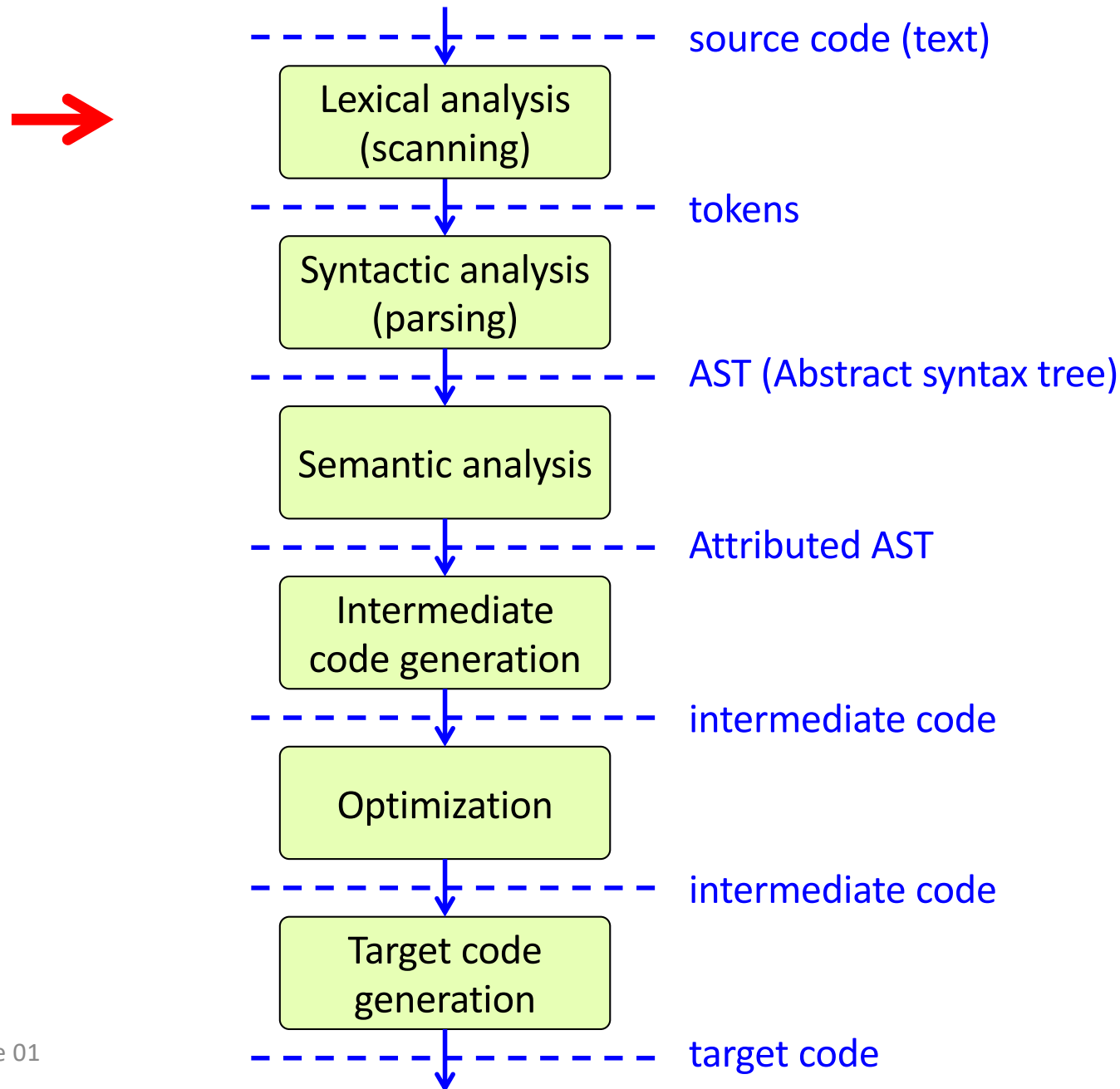
Some historical roots

- 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:



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:

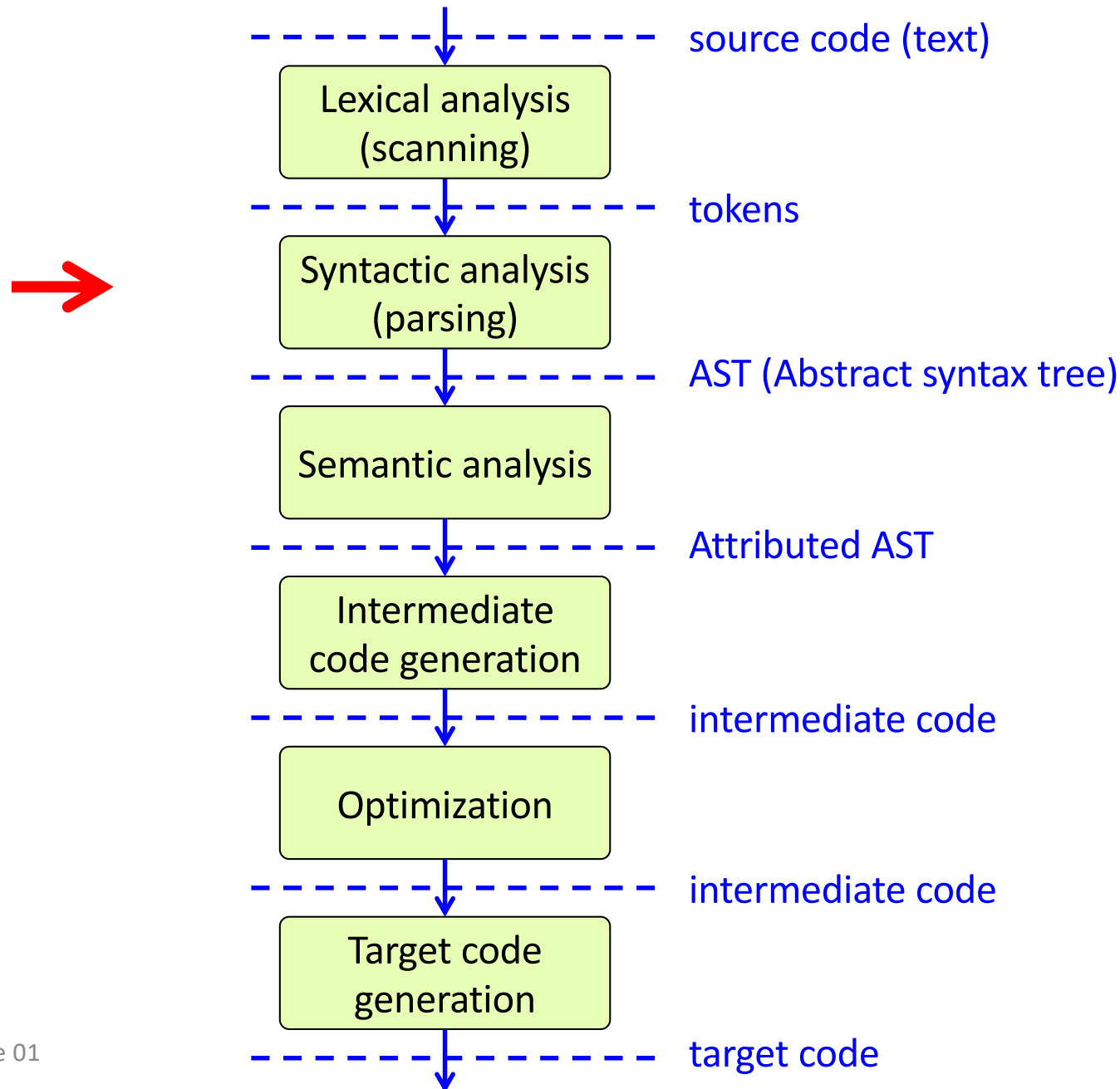
```
WHILE LPAR ID("k") LEQ ID("n") RPAR LBRA ID("sum") EQ ...
```

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)

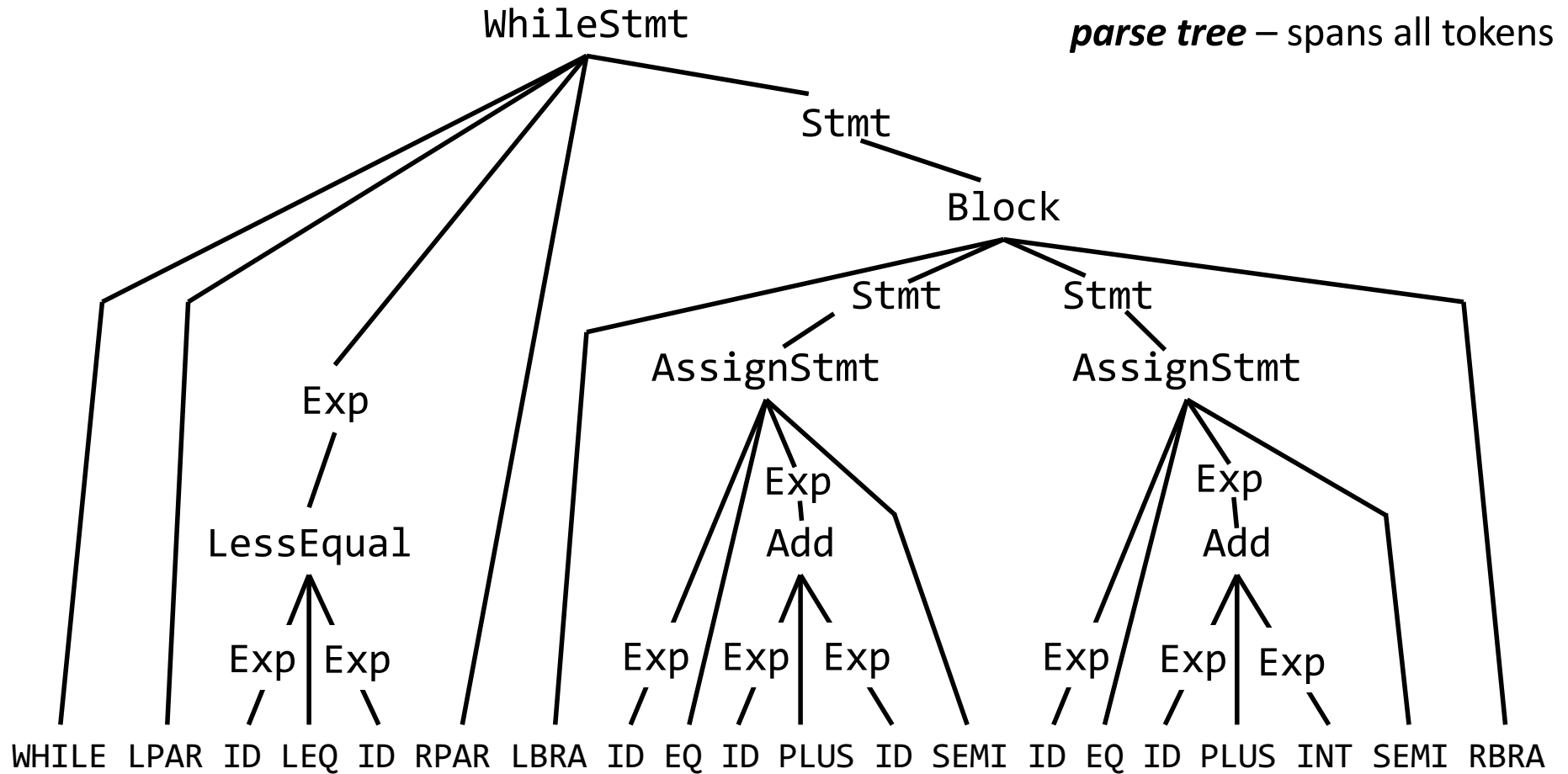
WhileStmt

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 ; }
```

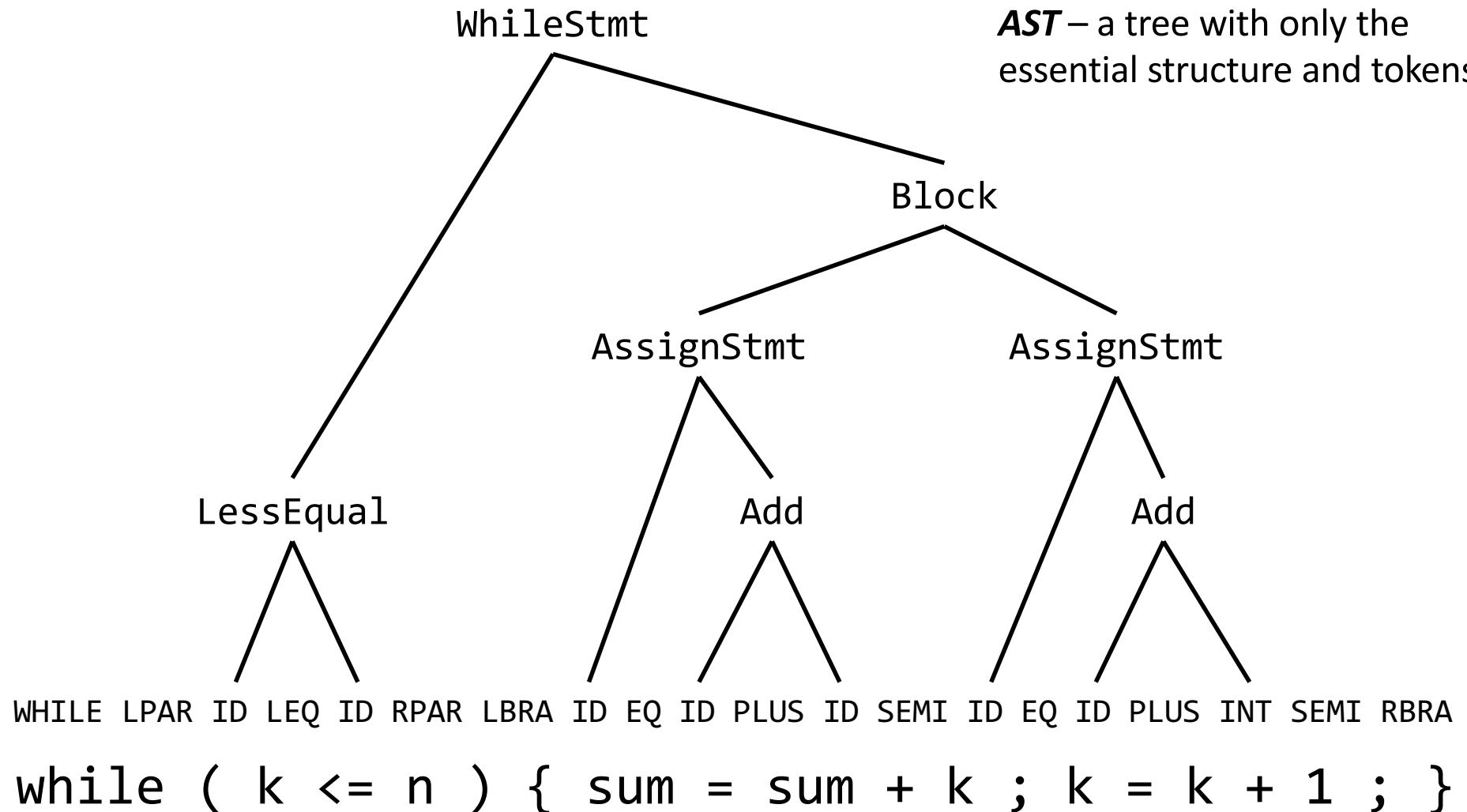
Syntactic analysis (parsing)



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

Abstract syntax tree (AST)

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, ...)

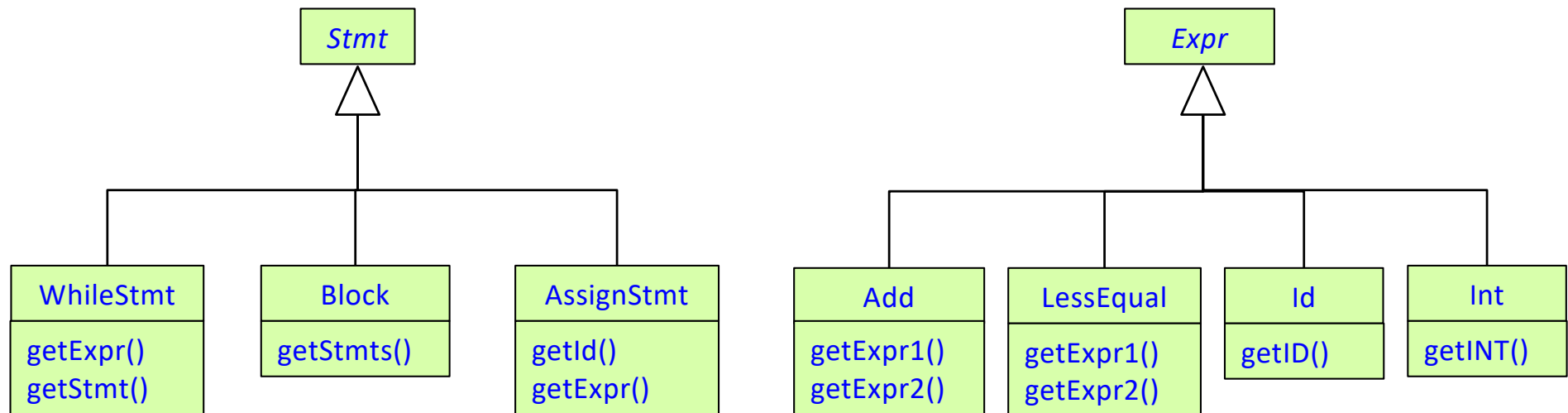
(or corresponding algebraic datatypes with constructors)

Designing an AST model

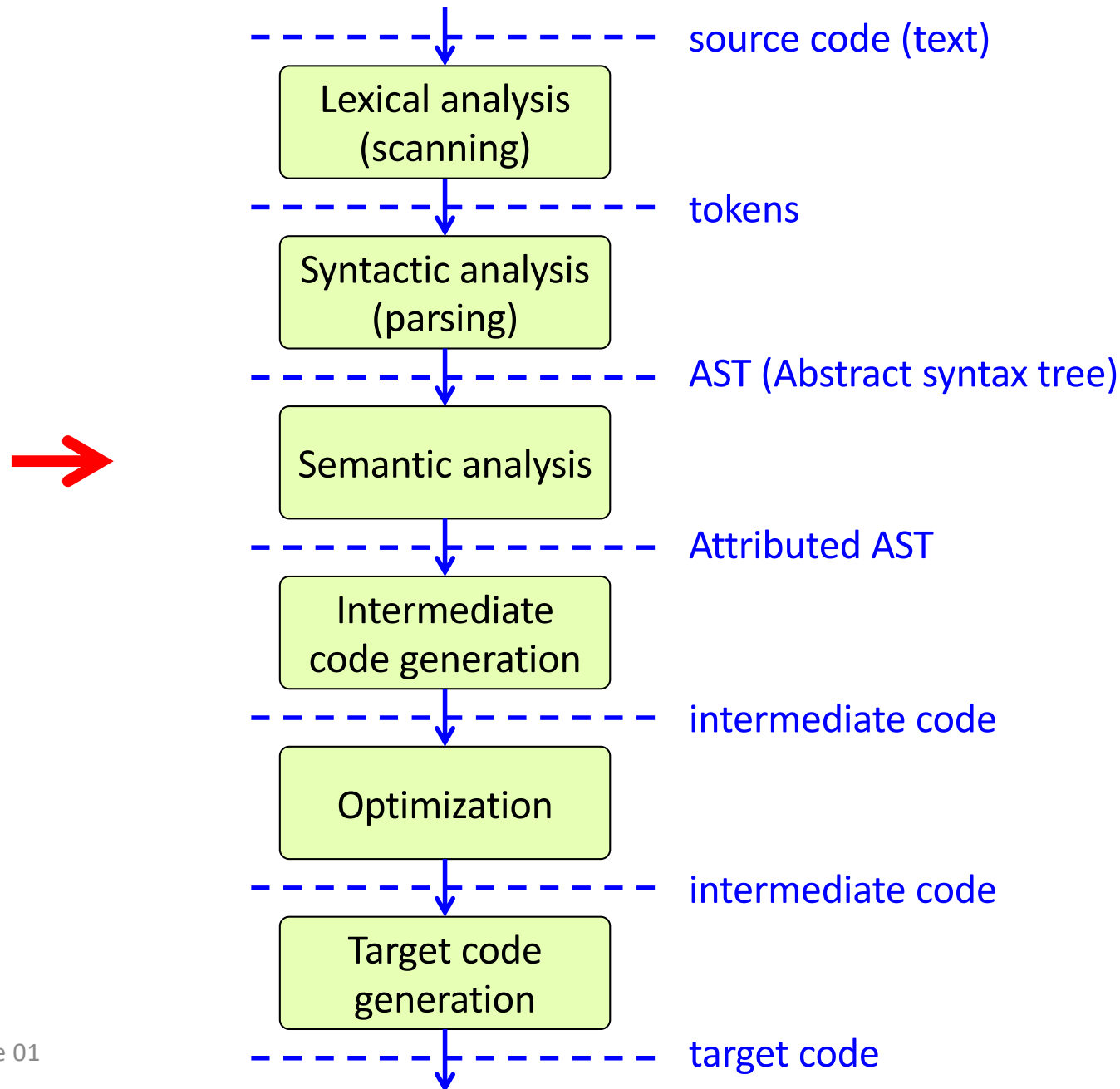
- What abstract constructs are there in the language
 - OOP: Make them abstract classes
 - FP: Make them algebraic data types
- What concrete constructs are there?
 - OOP: Make them subclasses of the abstract classes
 - FP: Make them constructors of the algebraic data types
- What parts do the concrete 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:



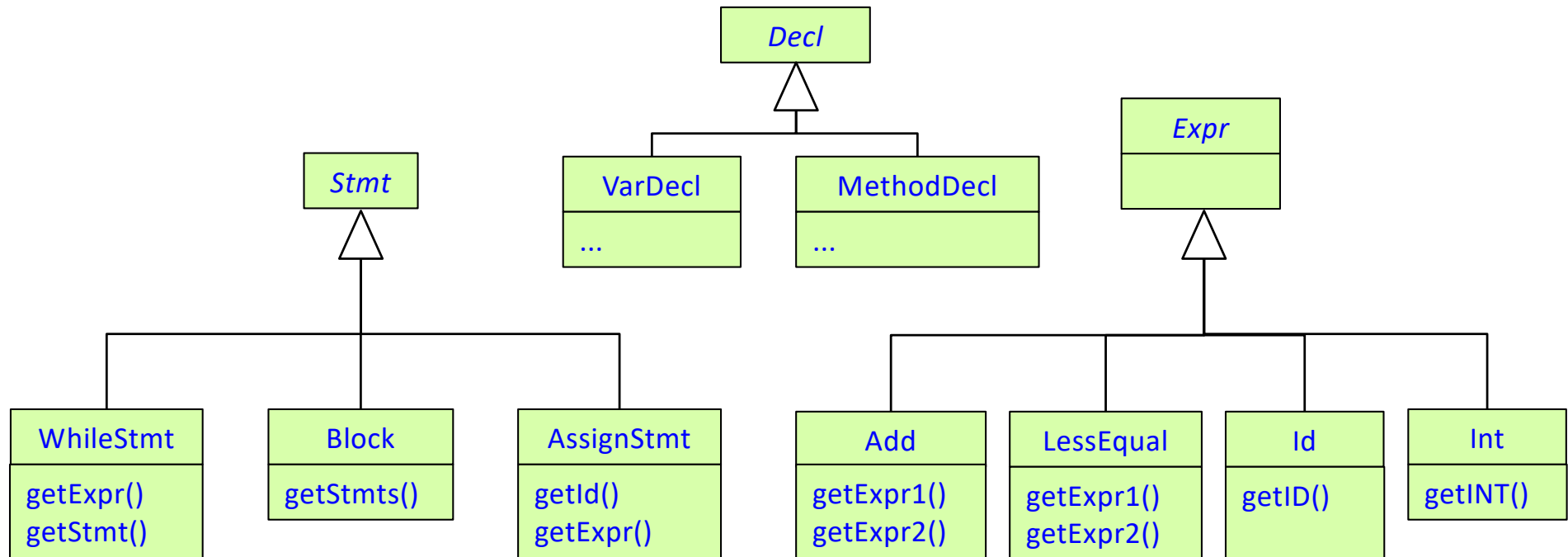
Semantic analysis

Analyze the AST, for example,

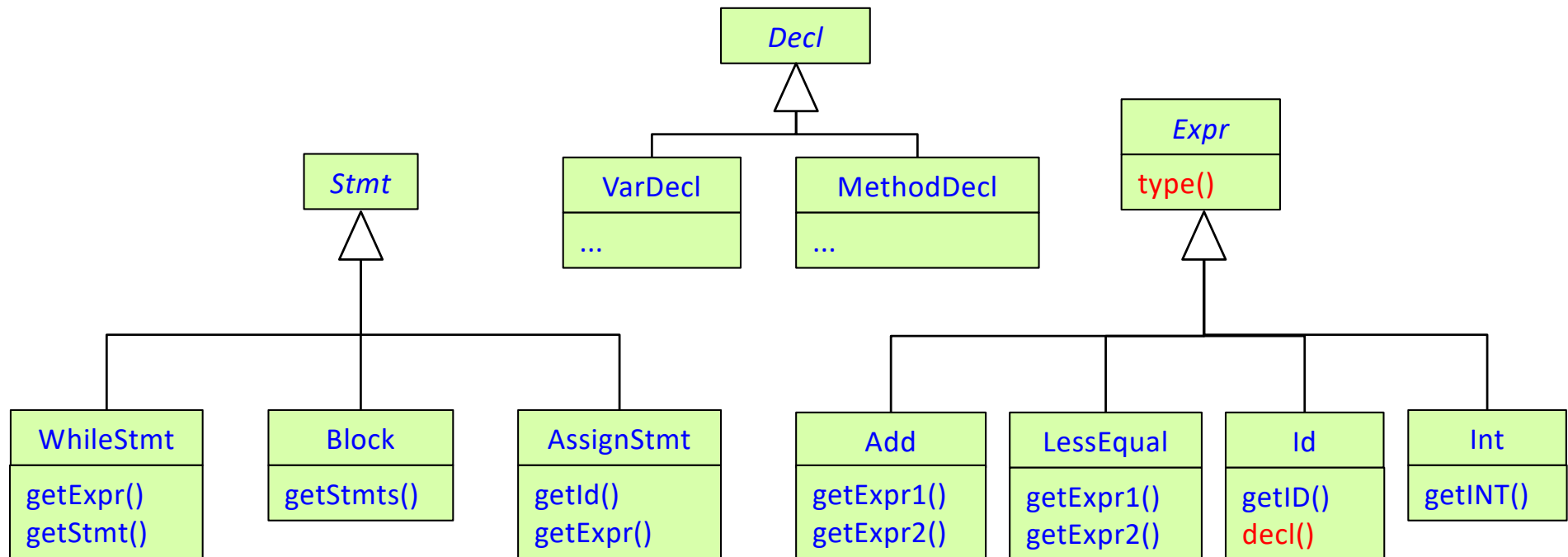
- Which declaration corresponds to a variable?
- What is the type of an expression?
- Are there compile time errors in the program?

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

Example attributes

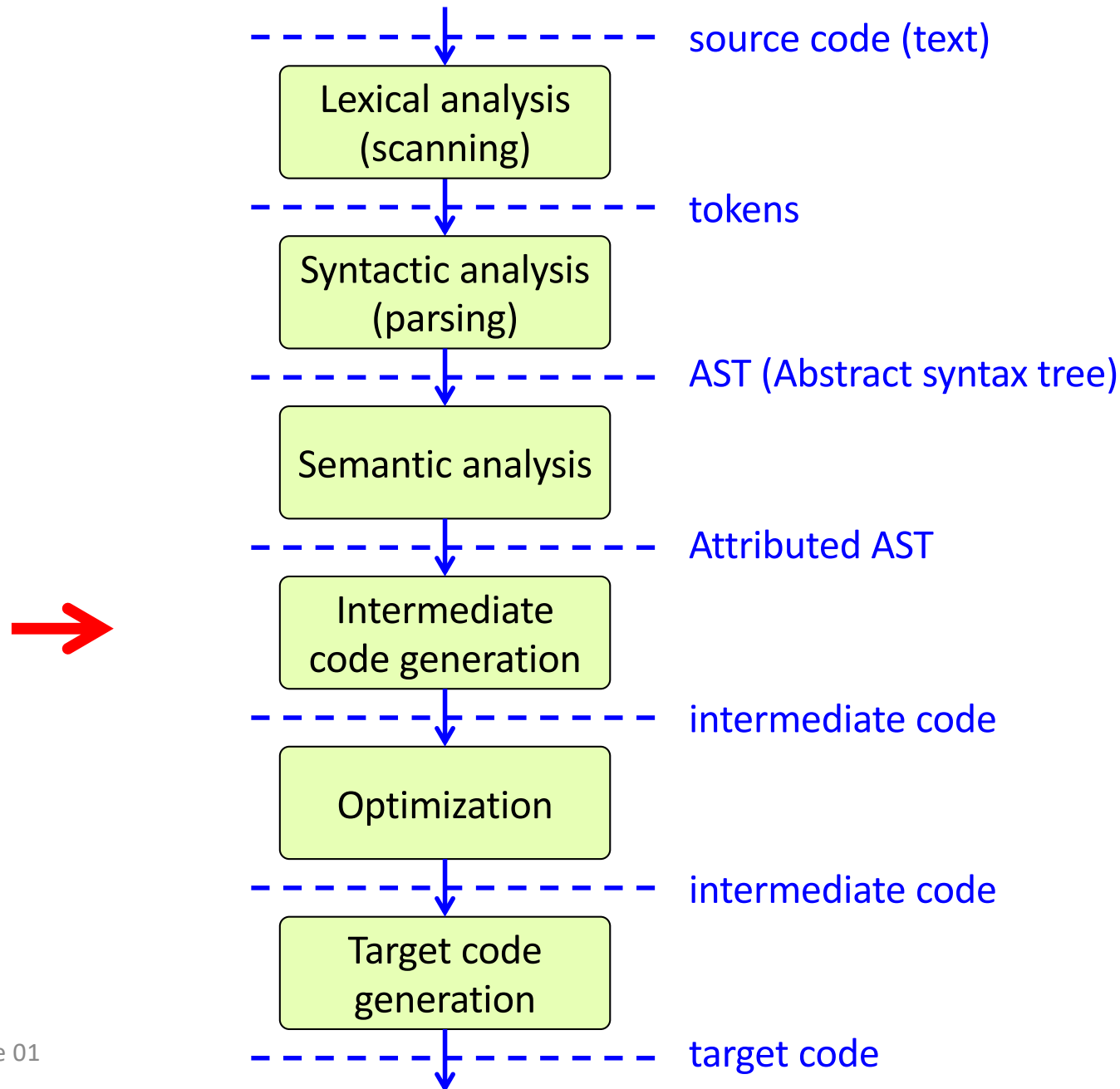


Example attributes



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:

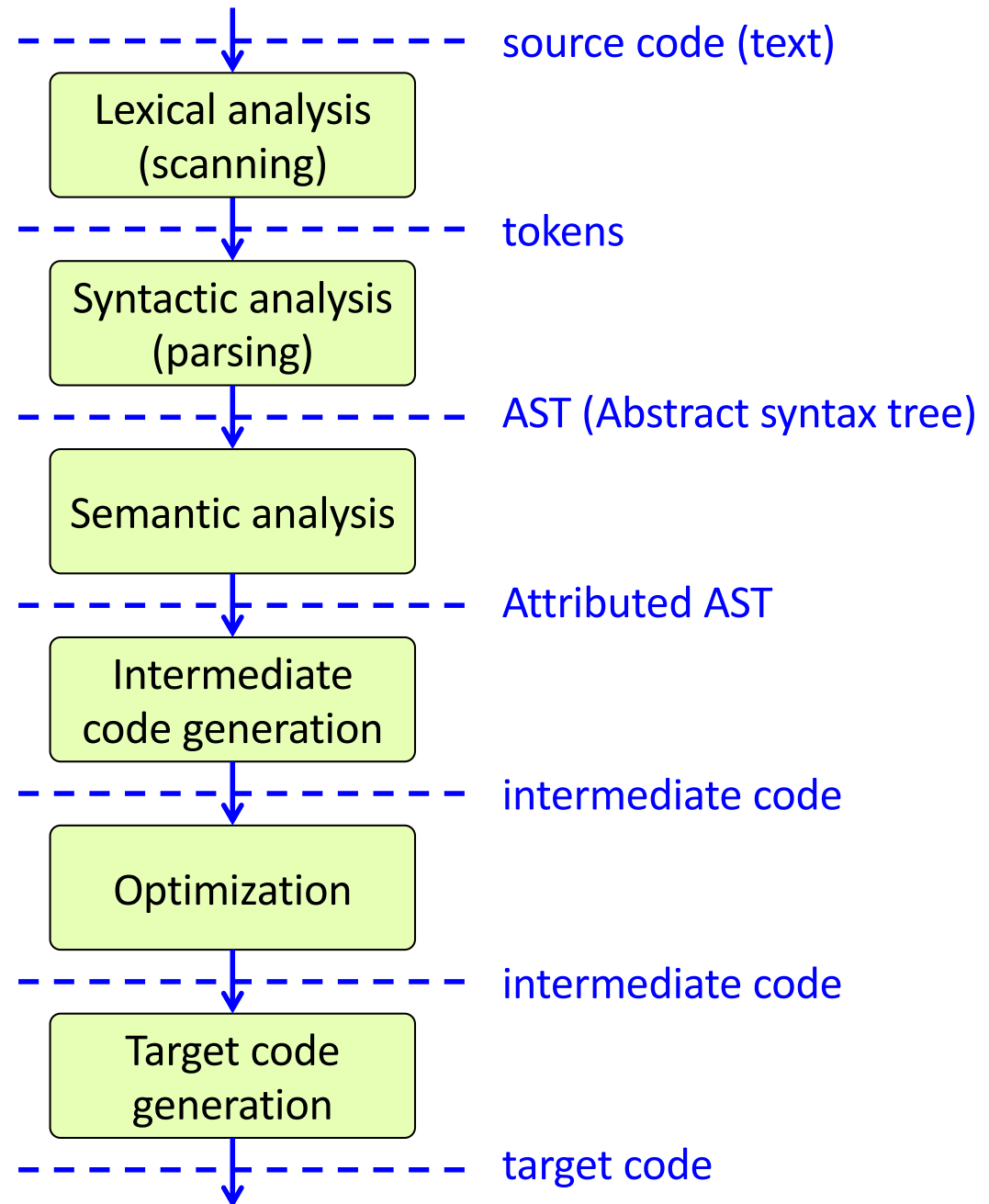


Intermediate code generation

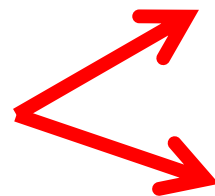
Intermediate code:

- 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)

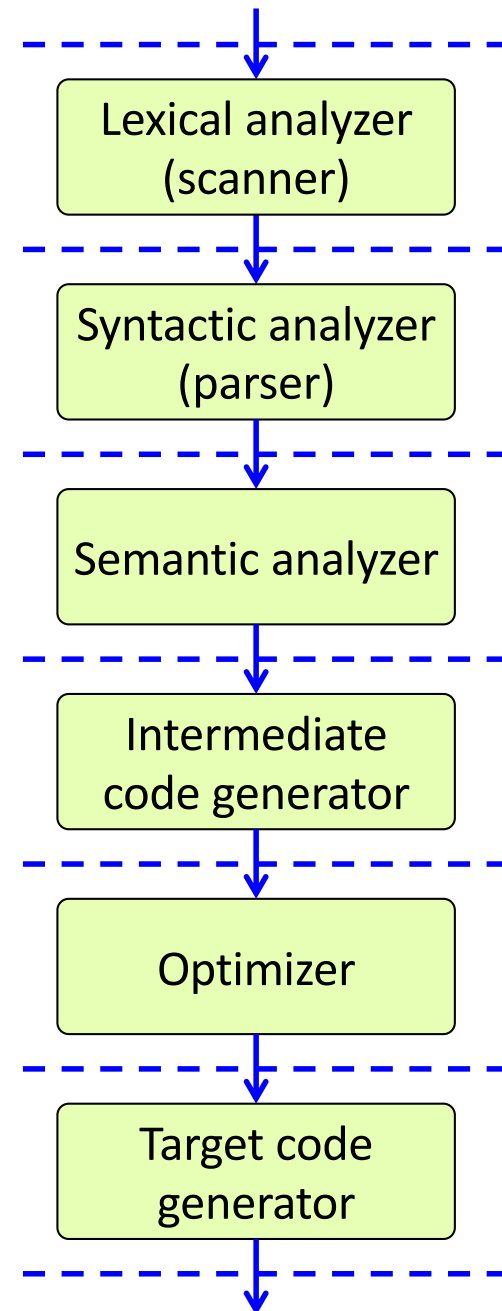
Compiler phases and program representations:



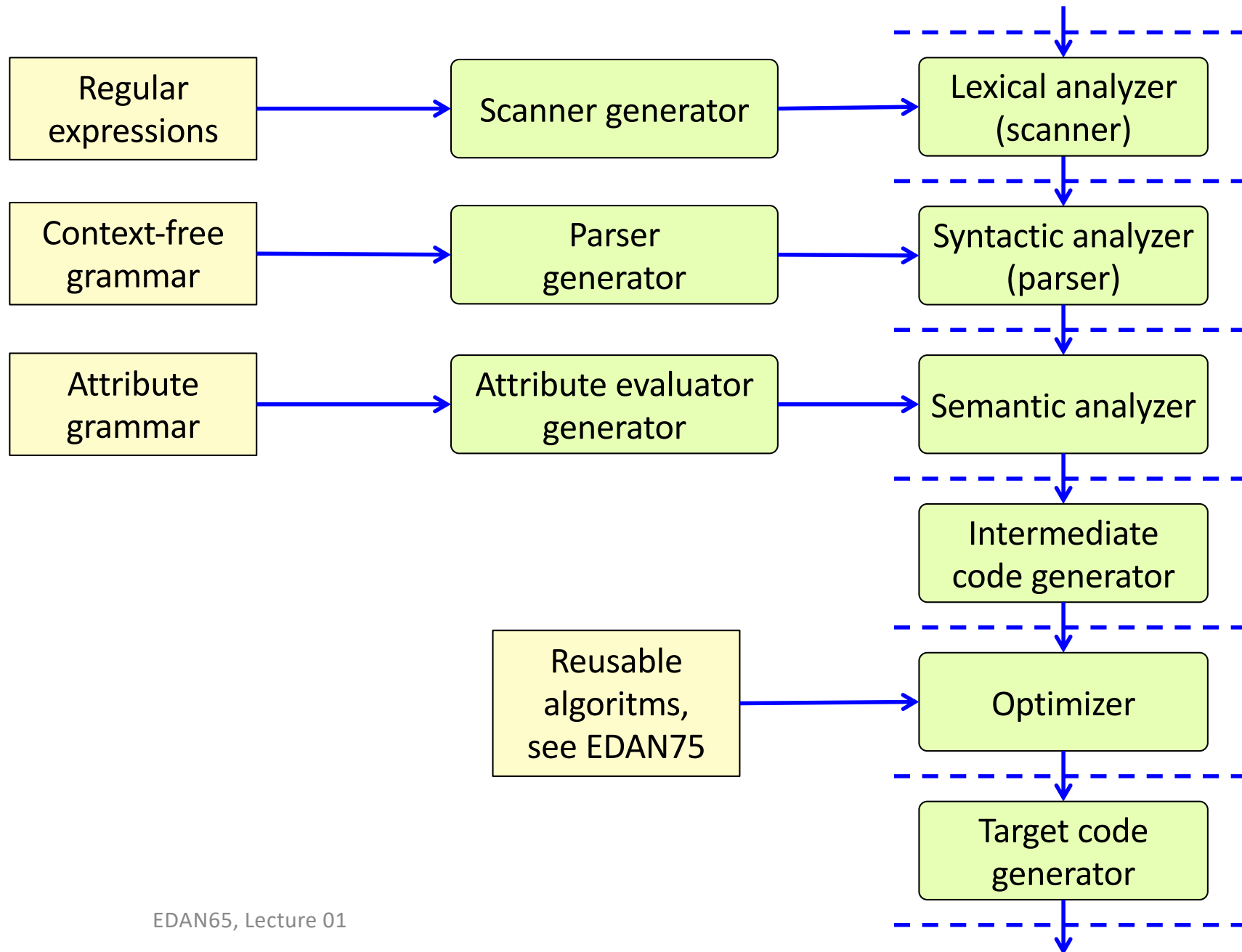
See the course Optimizing Compilers, EDAN75



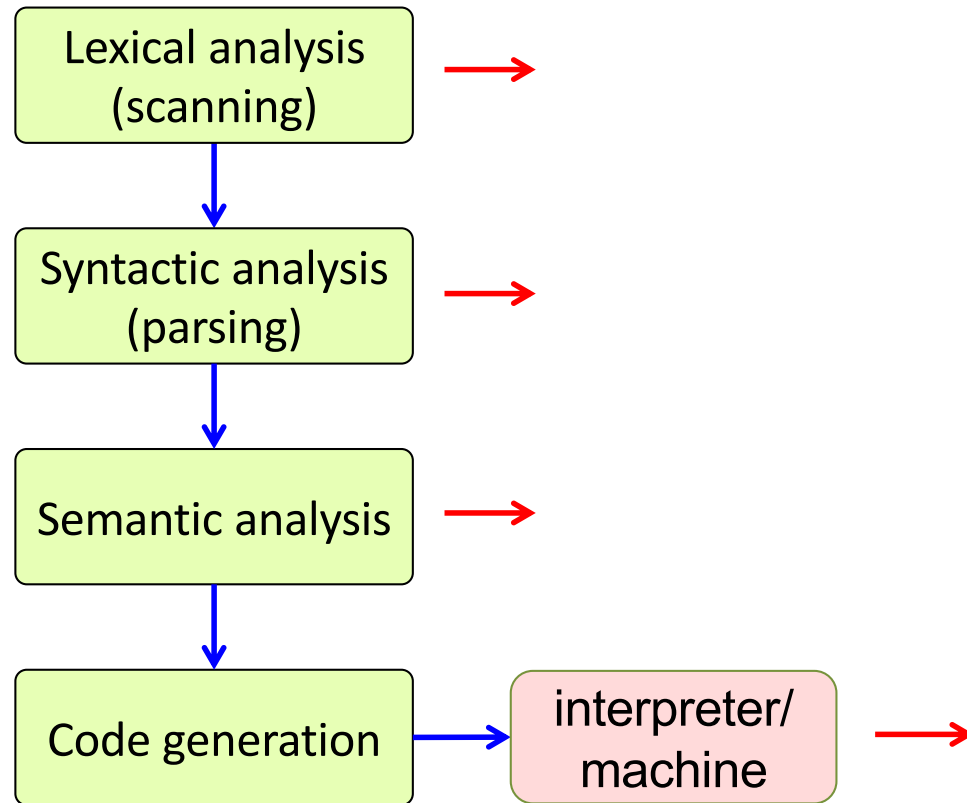
Generating the compiler:



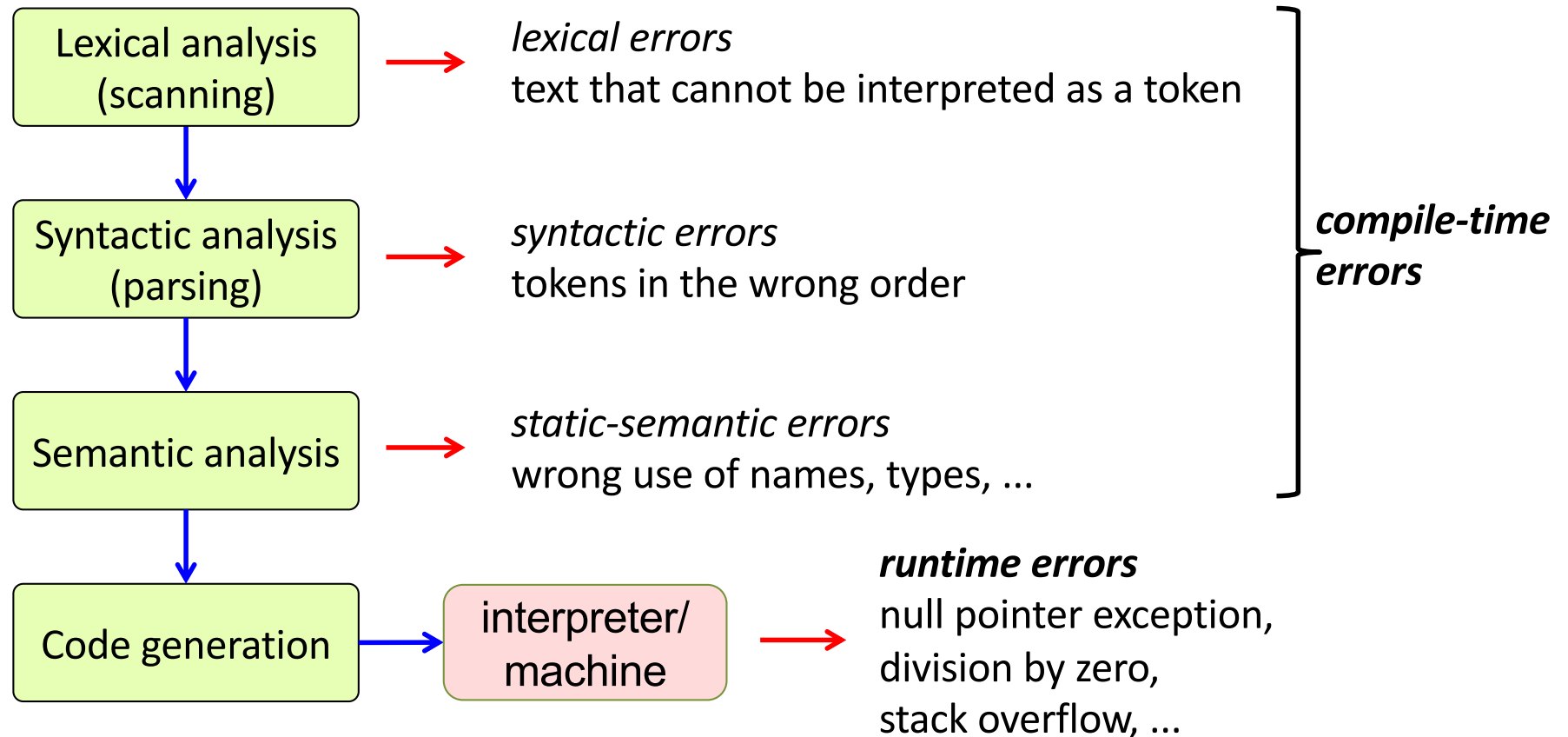
Generating the compiler:



Program errors



Program errors



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;  
}
```

4.

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

2.

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

5.

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

3.

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

Example errors

Lexical error:

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

Runtime error:

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

Syntactic error:

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

Logic error:

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

Static-semantic error:

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

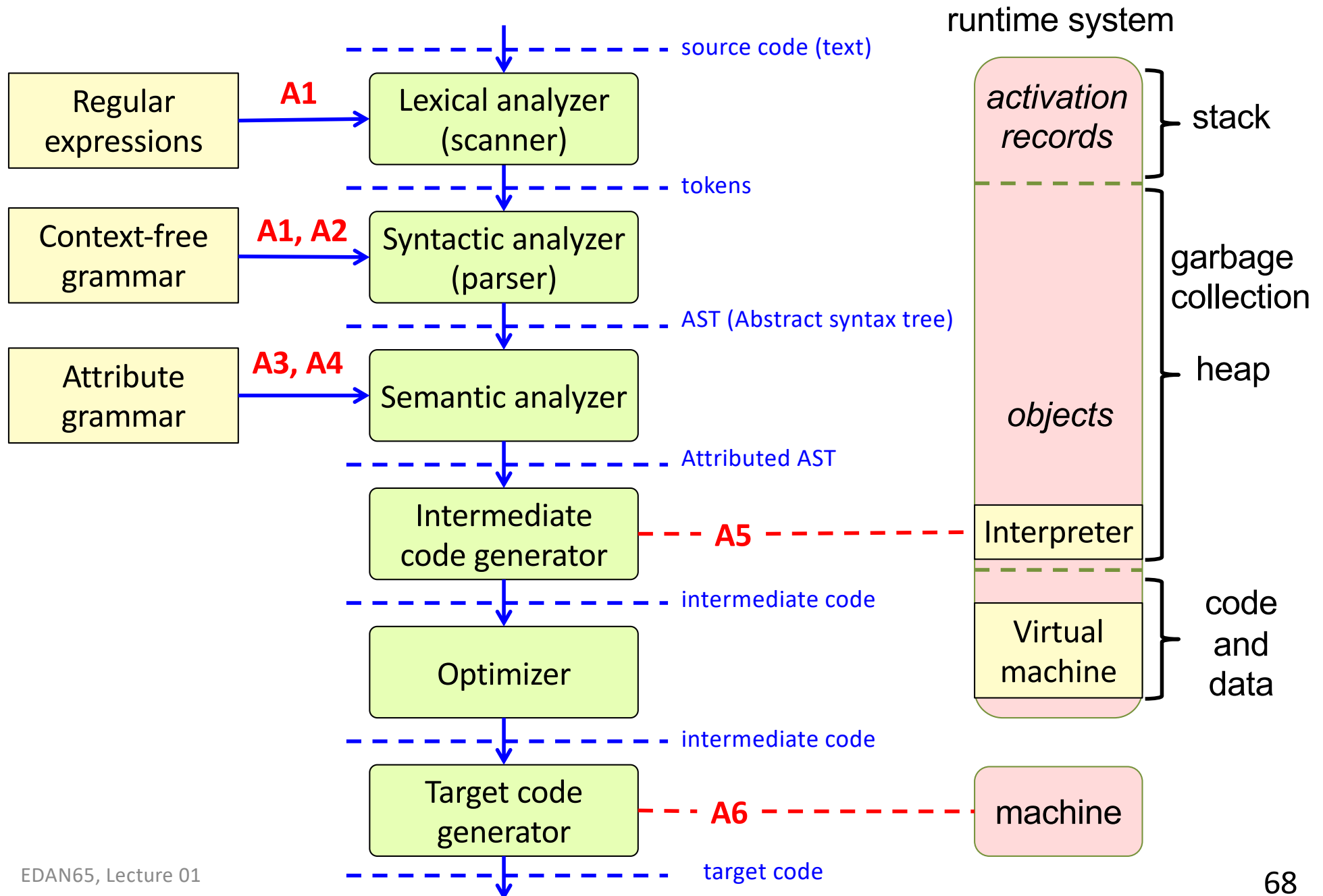
Examples: Java, C#, Smalltalk, Python, ...

- **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**, 1p2
 - Build a small tool, evaluate it, write a short paper
- EDAN90, **Advanced Project in Computer Science**, 1p2
 - Prerequisite: EDAN70
- EDAP05, **Concepts in Programming Languages**, 1p2
 - odd years (2021, 2023, ...)
- EDAP15, **Program Analysis**, 1p2
 - even years (2020, 2022, ...)
- EDAN75, **Optimizing compilers**, 1p1
 - even years (2020, 2022, ...)

- **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

<http://www.modelica.org>

The screenshot displays the Modelica software environment. On the left is a hierarchical block library with categories like Modelica, User's Guide, Blocks, Mechanics, Fluid, Electrical, and Analog. The main workspace shows a 3D cutaway model of a car with various electrical components overlaid. A text box in the upper left provides the definition for an 'Ideal linear electrical capacitor' component, including its parameters and governing equations. A plot in the lower left shows the current through the capacitor over time. The upper right shows a detailed schematic of the capacitor component with its internal structure and associated simulation parameters.

```
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;
```

plug_pin[1] plug

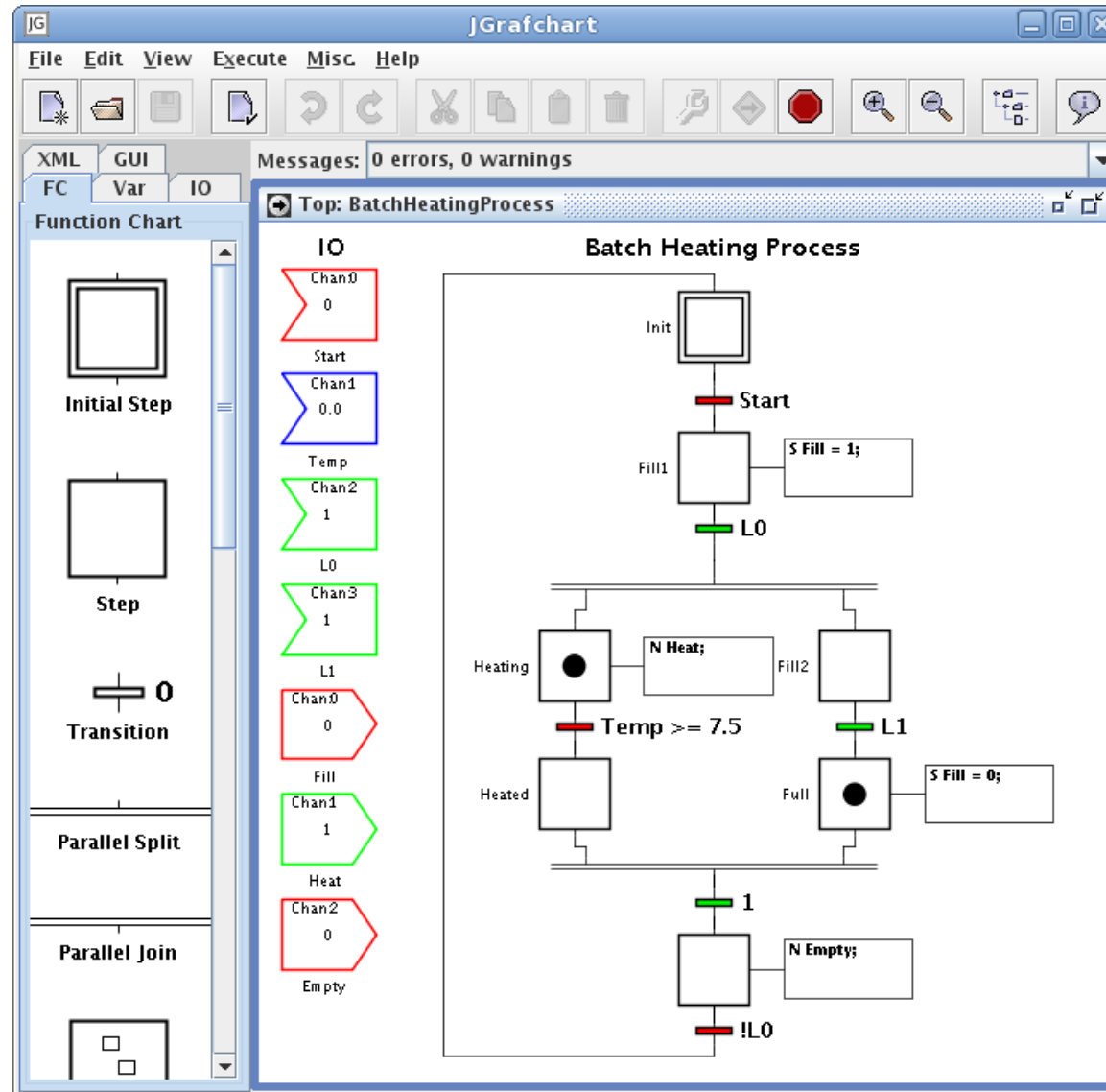
Time 20.0

Power [W]

time [s]

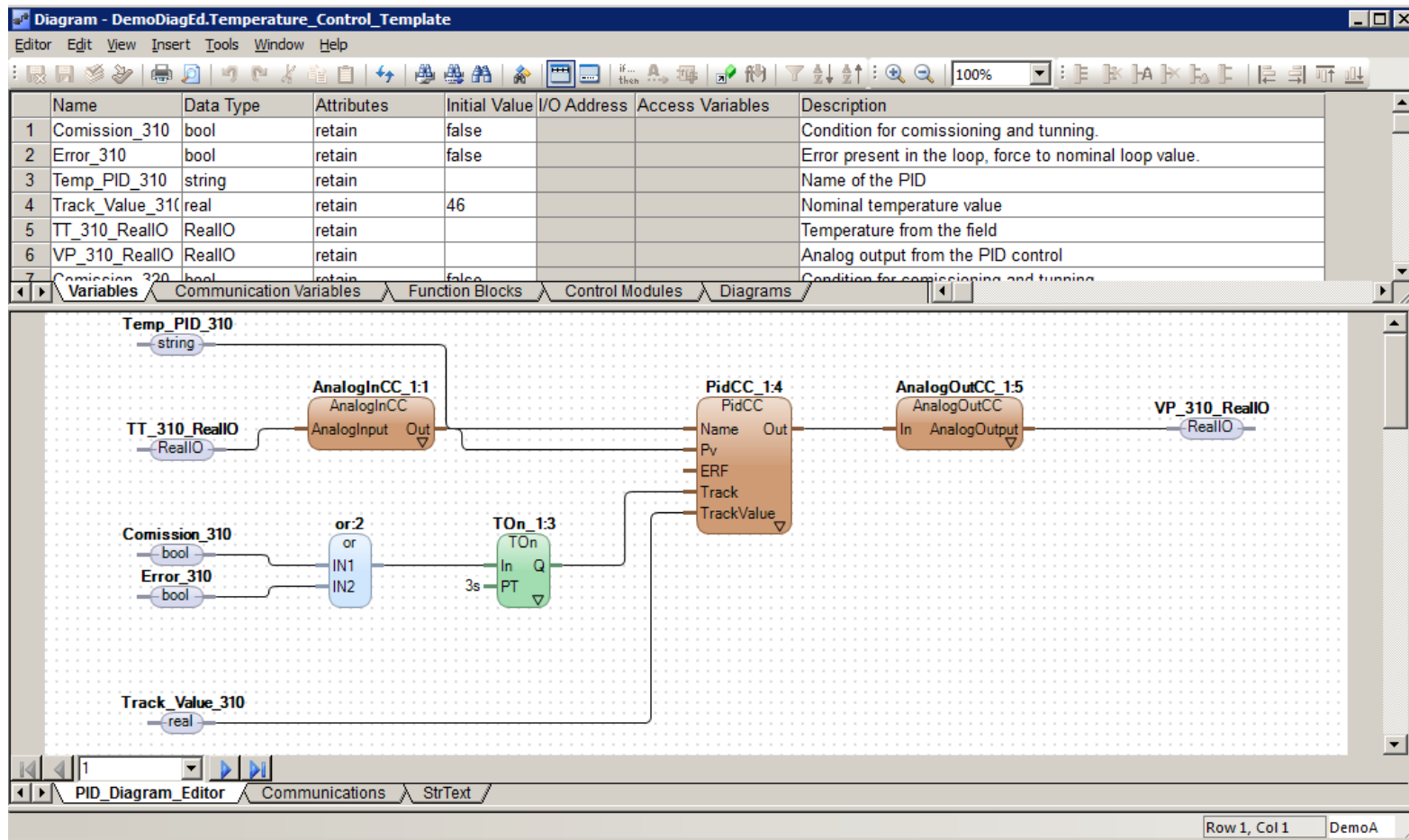
Grafchart

<http://www.control.lth.se/Research/tools/grafchart.html>



Control Builder Diagram

<https://global.abb>



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 <http://cs.lth.se/edan65> for what to do this week.