

EDAN65: Compilers

Introduction and Overview

Görel Hedin

Revised: 2017-08-27

Course registration

- Confirm by signing the Registration Form
- Prerequisites
 - Object-oriented programming and Java
 - Algorithms and data structures
(recursion, trees, lists, hash tables, ...)

Student representatives

- 2 students
- Participate in post course CEQ discussion (Course Experience Questionnaire)

Course information

- Web site: <http://cs.lth.se/edan65>
 - read the Week by Week page to find out what to do each week.
 - Lecture notes, articles to read, assignments, exercises
 - 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 an on-line e-book through Lund University
 - Only part of the book is used. Covers only part of the course.
- Forum (Questions&Answers) using the Piazza system.
- Quizzes using the Moodle system.

Course structure

- 14 lectures, Mon 13-15, Tue 10-12 (hopefully 1-2 guest lectures)
- Assignment 0, for freshening up on Java and Unix, and understanding build system Gradle. Do on your own.
- Assignment 1-6. Mandatory.
 - Work in pairs. Use the lecture break or the forum to form pairs.
 - Heavy. Get approved and get help at Lab sessions.
 - Thu 13-15, Thu 15-17, or Fri 10-12. **Sign up by Thursday Aug 31**
 - Lab sessions start next week (but start this week on your work)
 - Assignments prerequisite for doing exam
- Lecture quizzes
 - Do on your own. Not mandatory, but highly recommended.
- Exercises
 - Do on your own. Not mandatory, but highly recommended.
- Exam – sign up in advance through LTH system
 - Exam: Wednesday, Oct 25, 2017
 - Re-exam: January 2018, exact time and place to be announced.

More on 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
- If you get stuck
 - ask on the Piazza forum
 - you are encouraged to give answers to other students on the forum (for general advice, not solutions)
- Both of you should be able to explain all parts of your solution.
- Want to use a git repo? Make it private! Free on BitBucket.

Estimated typical effort for assignments

A0: Unix, Java, Gradle	0-4 hours	highly recommended
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

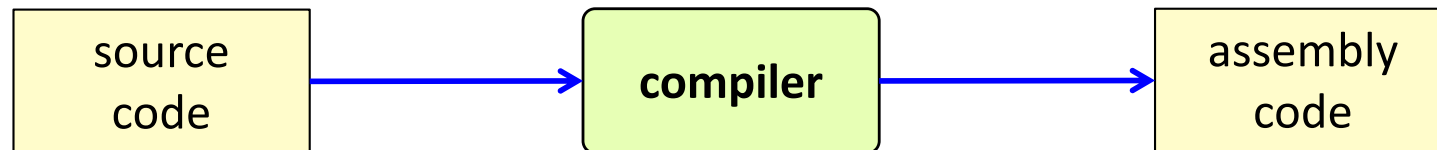
Instructors

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

Why learn compiler construction?

- Very useful in practice
 - Languages are everywhere
 - Your next project might need a small language
 - A compiler exemplifies the major techniques used for any software language processing
- Interesting
 - Compiler theory: fundamental to computer science
 - Essential for understanding programming languages

A traditional compiler



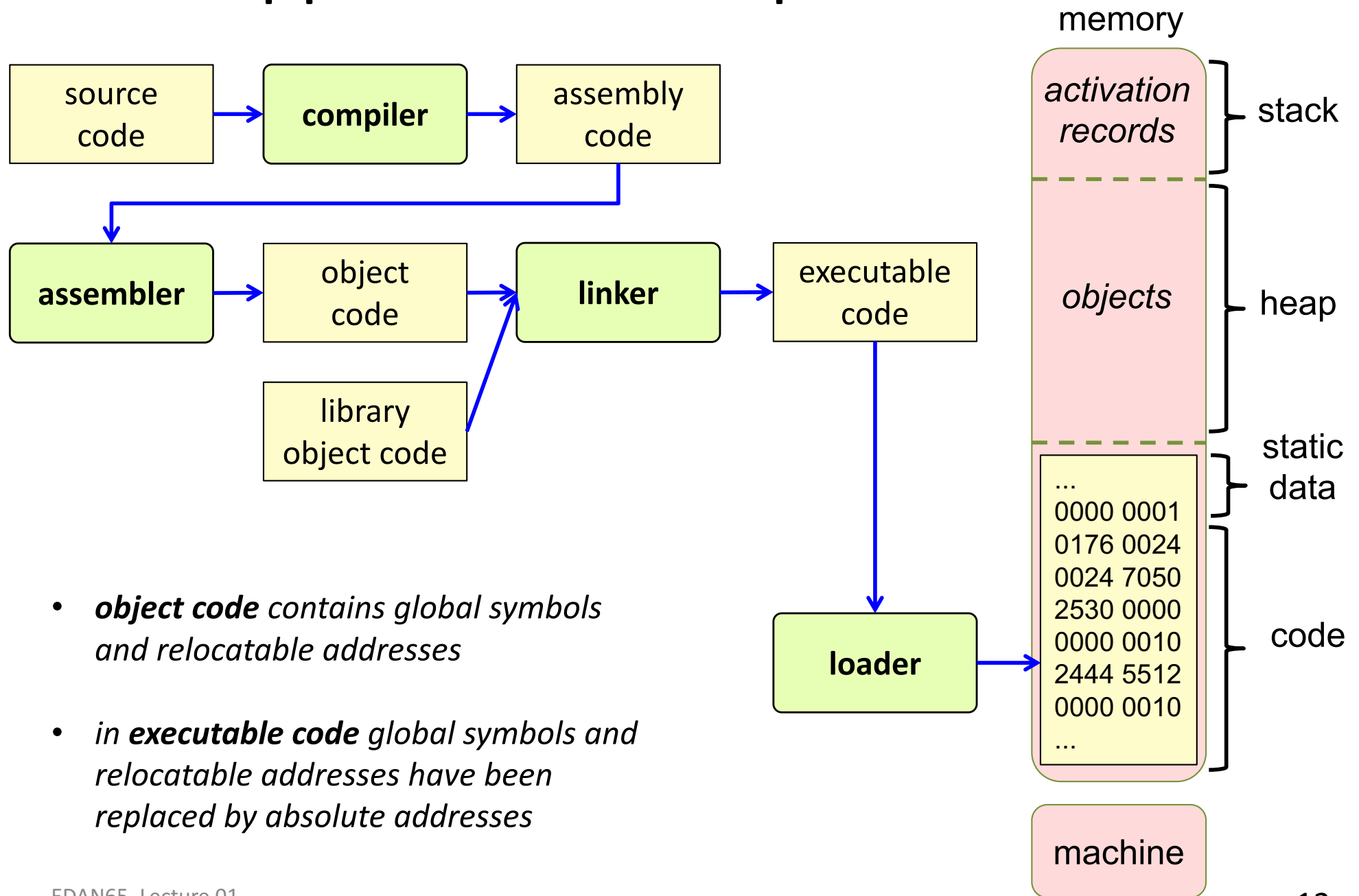
EXAMPLE:

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

compiler

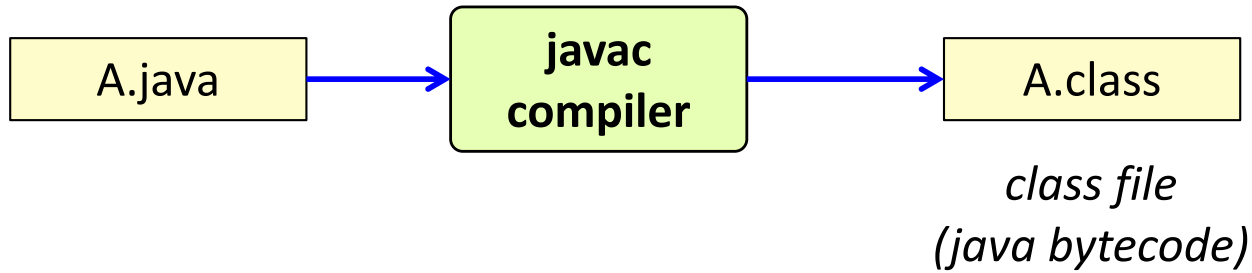
```
.data  
a:      .long  0  
b:      .long  0  
csum:   .long  0  
  
.code  
...  
movl    a, %eax  
addl    b, %eax  
addl    $1, %eax  
movl    %eax, csum  
...
```

What happens after compilation?

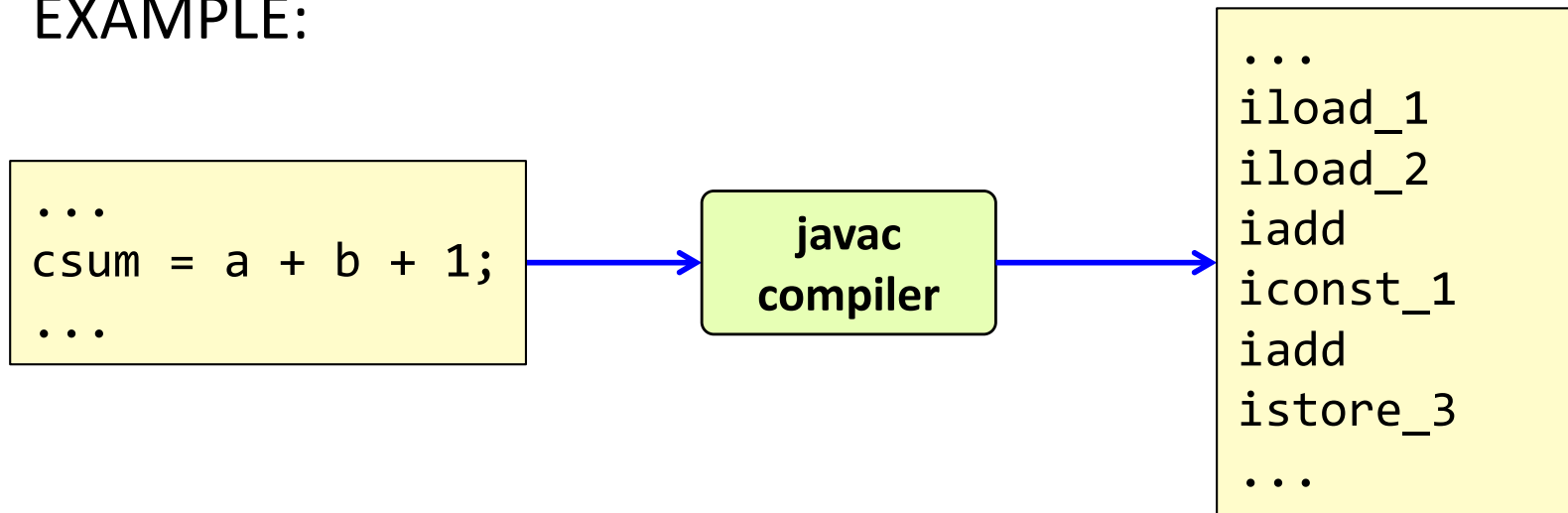


- **object code** contains global symbols and relocatable addresses
- in **executable code** global symbols and relocatable addresses have been replaced by absolute addresses

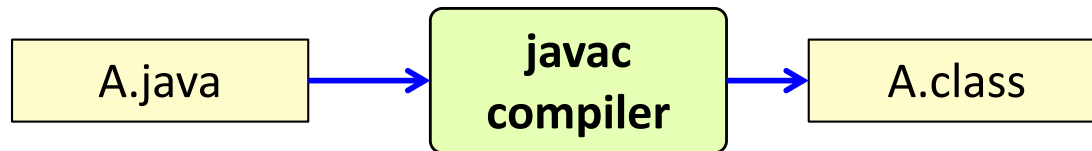
What about Java?



EXAMPLE:



Running Java code?

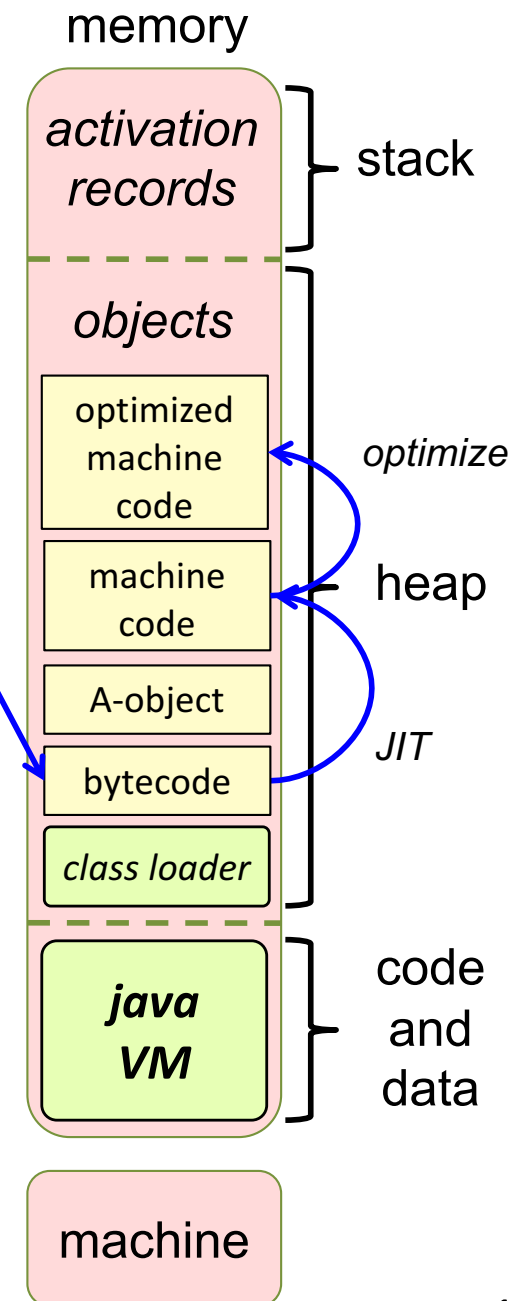


load
and verify

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

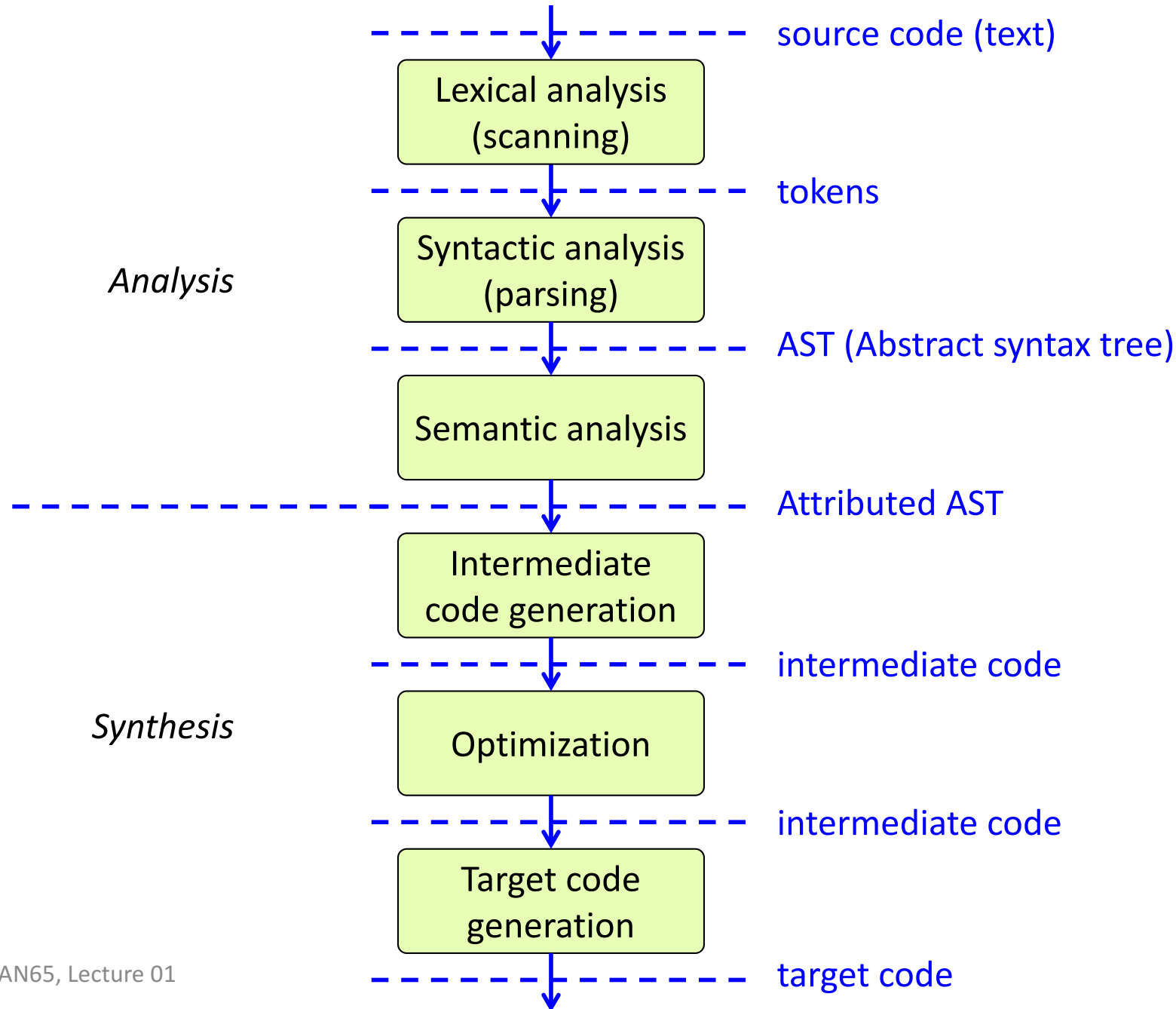
It can:

- 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

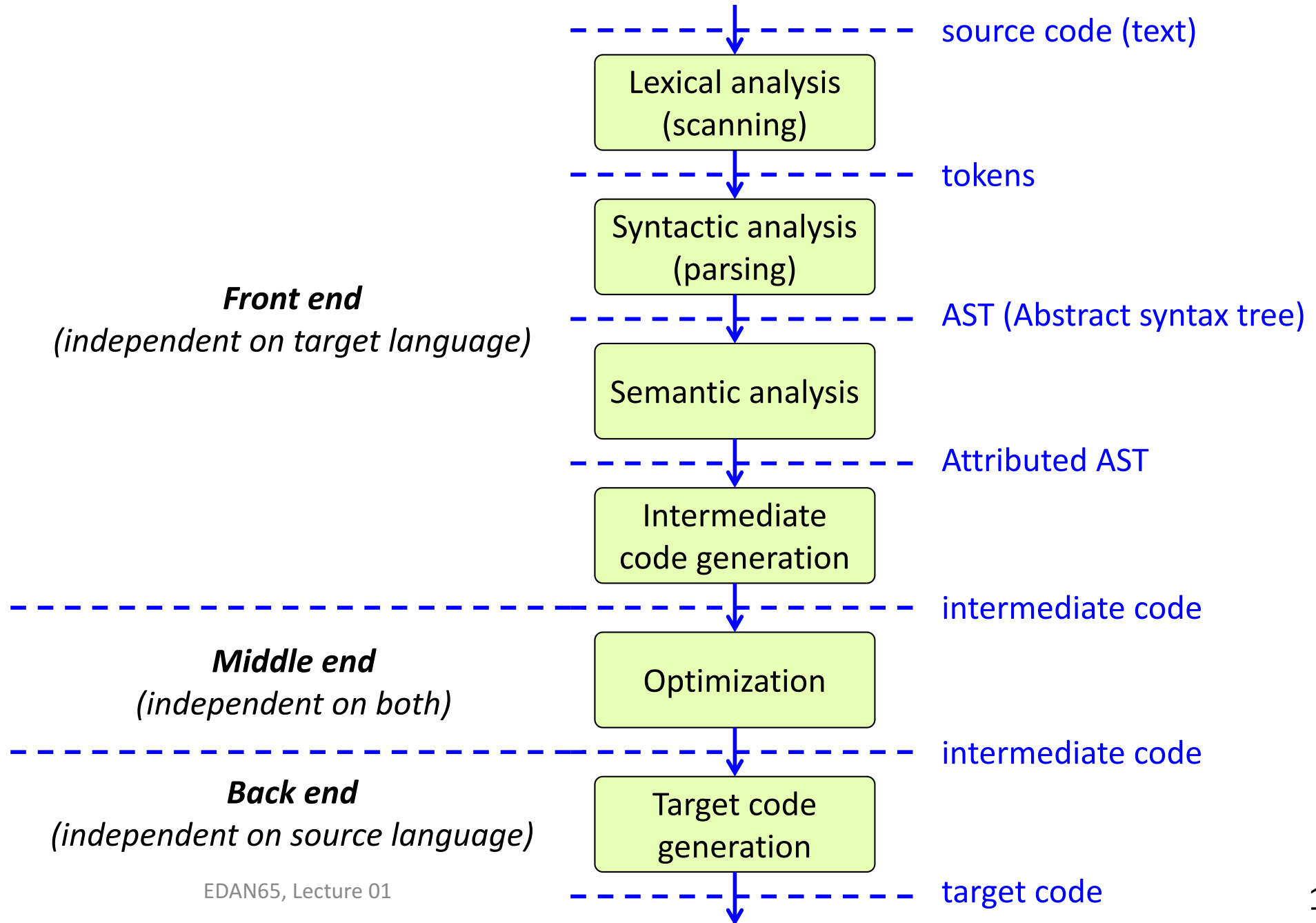


Inside the compiler:

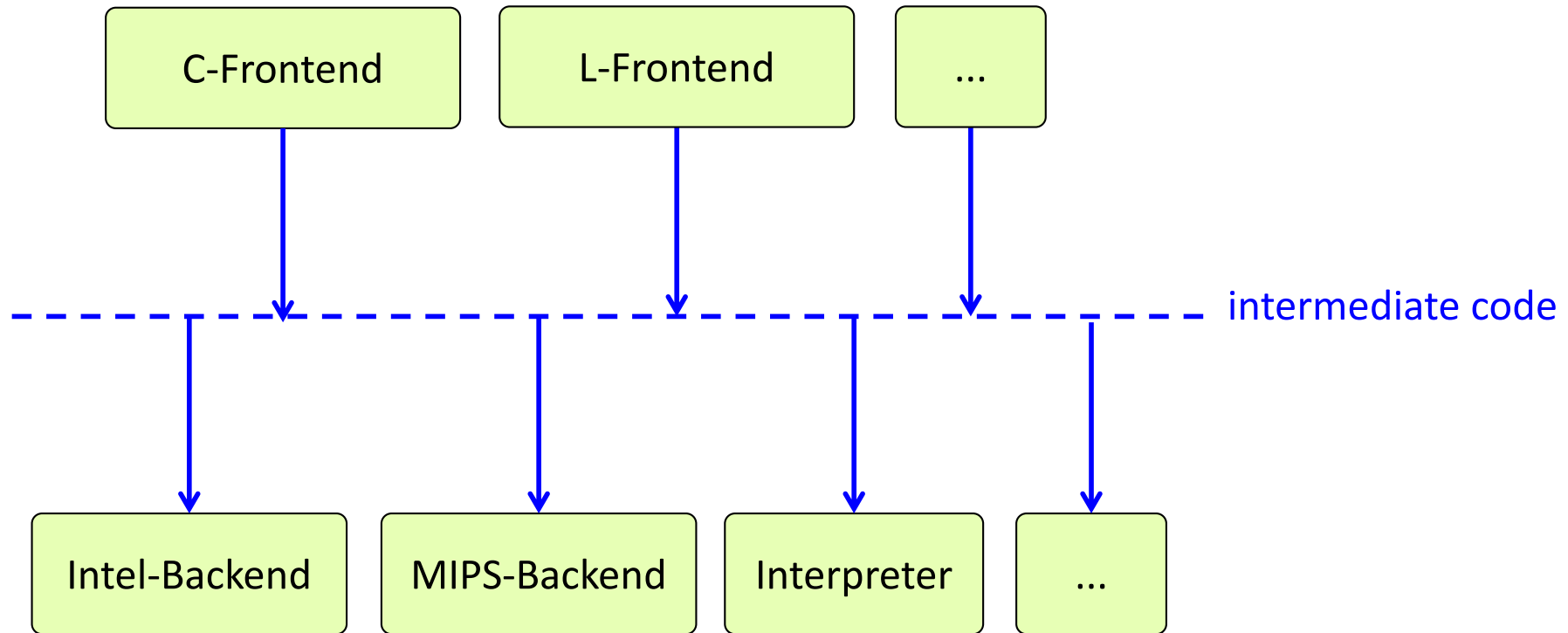
Each *phase* converts the program from one *representation* to another



Front and back end:



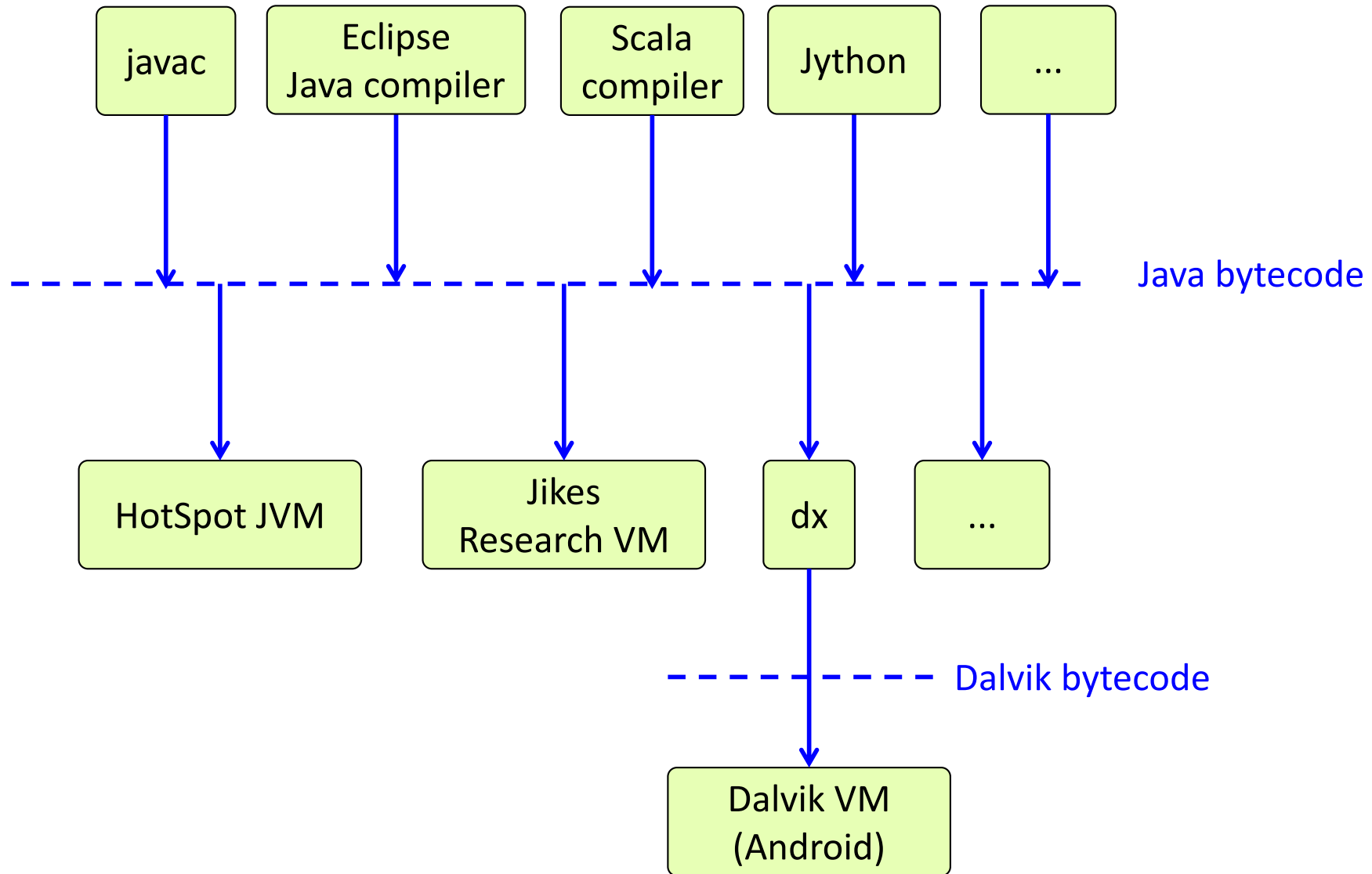
Several front and back ends:



Why?

- It is more rational to implement m front ends + n back ends than $m * n$ compilers.
- Many optimizations are best performed on intermediate code.
- It may be easier to debug the front end using an interpreter than a target machine

Example:



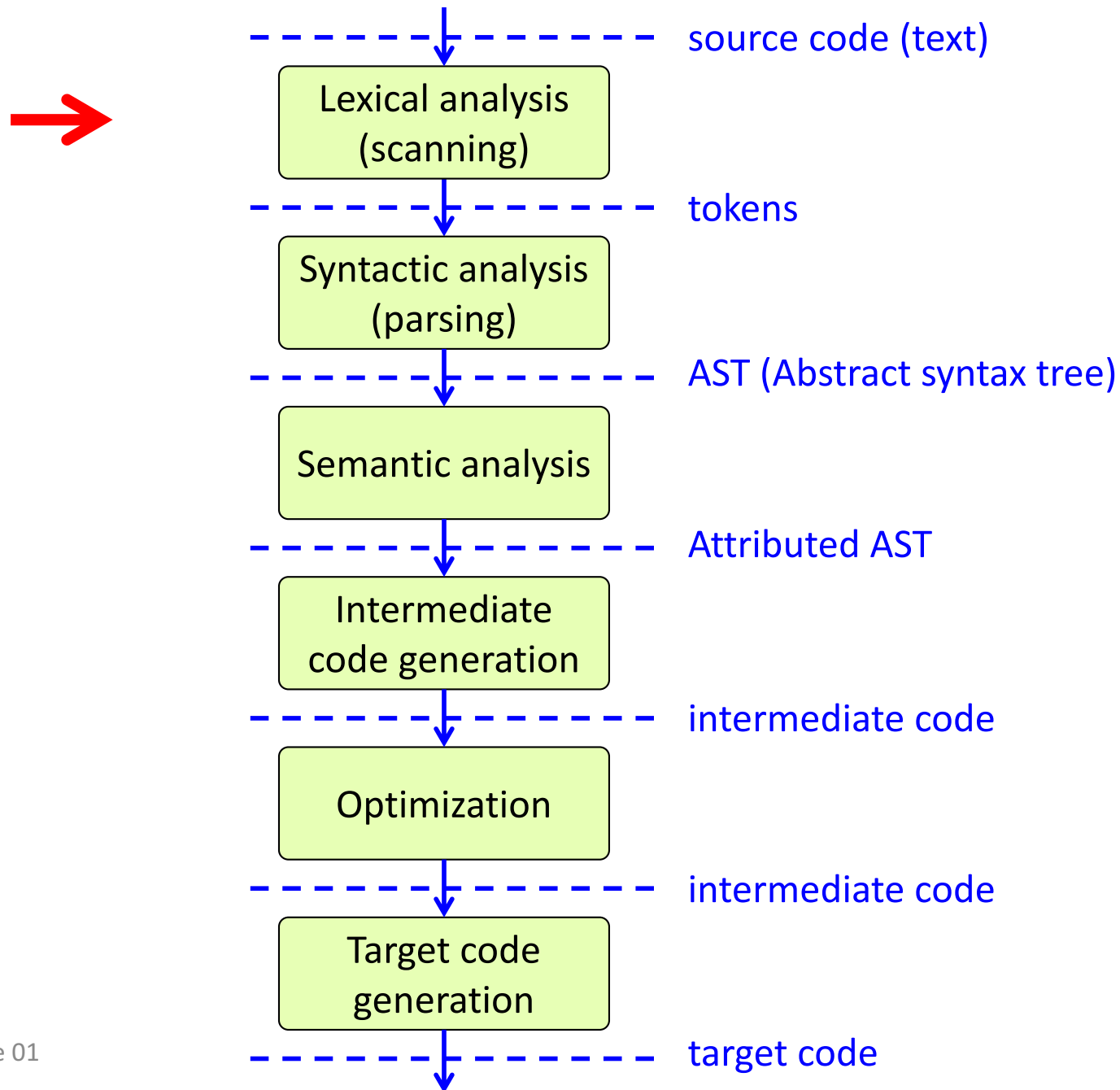
Some terminology

- A **compiler** translates a high-level program to low-level code.
- An **interpreter** is software that executes a high/low level program, often by calling one procedure for each program construct.
- In the context of compiler construction, a **virtual machine (VM)** is an interpreter that executes low-level, usually platform-independent code. (In other contexts, virtual machine can mean system virtualization.)
- 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.

Some historical anecdotes

- 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 in 1972, used portable code called "p-code". The interpreter was easy to port to different machines. The language spread quickly, and became a popular language taught at many universities.
- Smalltalk-80 used bytecode, and pioneered several runtime compilation and optimization techniques for object-oriented languages.

Compiler phases and program representations:



Lexical analysis (scanning)

Source text

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

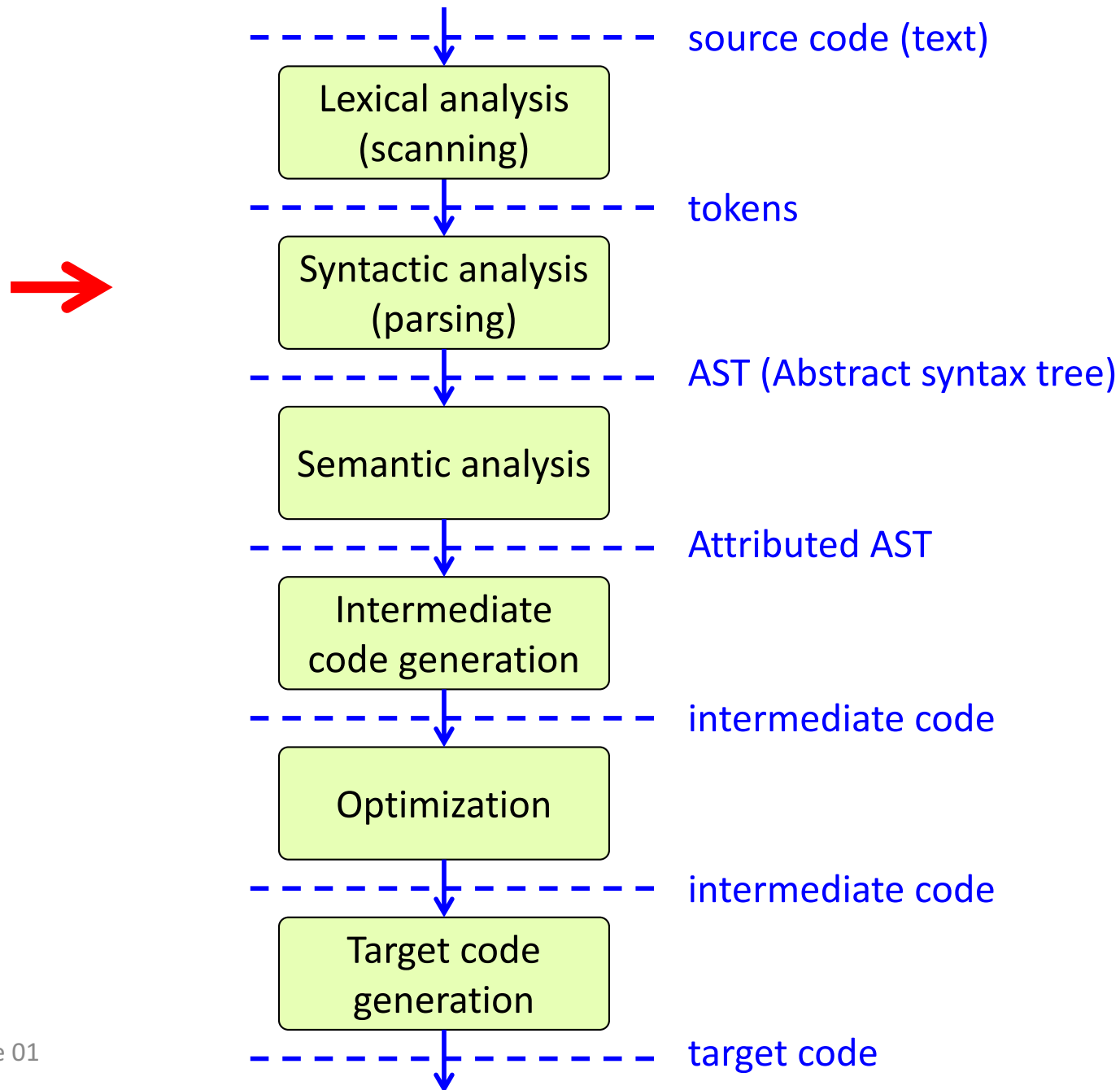
Tokens

```
WHILE LPAR ID(k) LEQ ID(n) RPAR LBRA  
ID(sum) EQ ID(sum) PLUS ID(k) SEMI  
ID(k) EQ ID(k) PLUS INT(1) SEMI  
RBRA
```

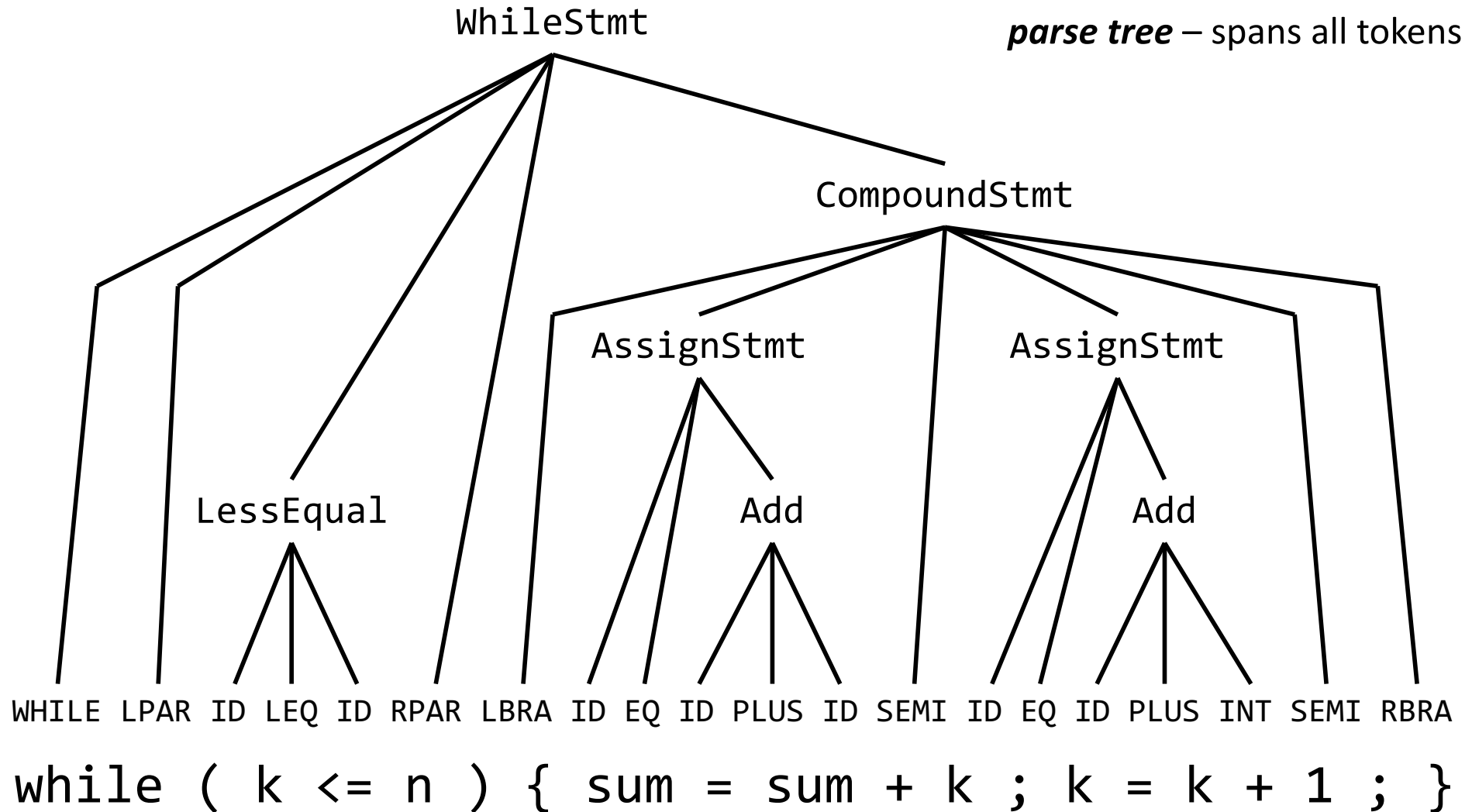
A *token* is a symbolic name, sometimes with an attribute.

A *lexeme* is a string corresponding to a token.

Compiler phases and program representations:

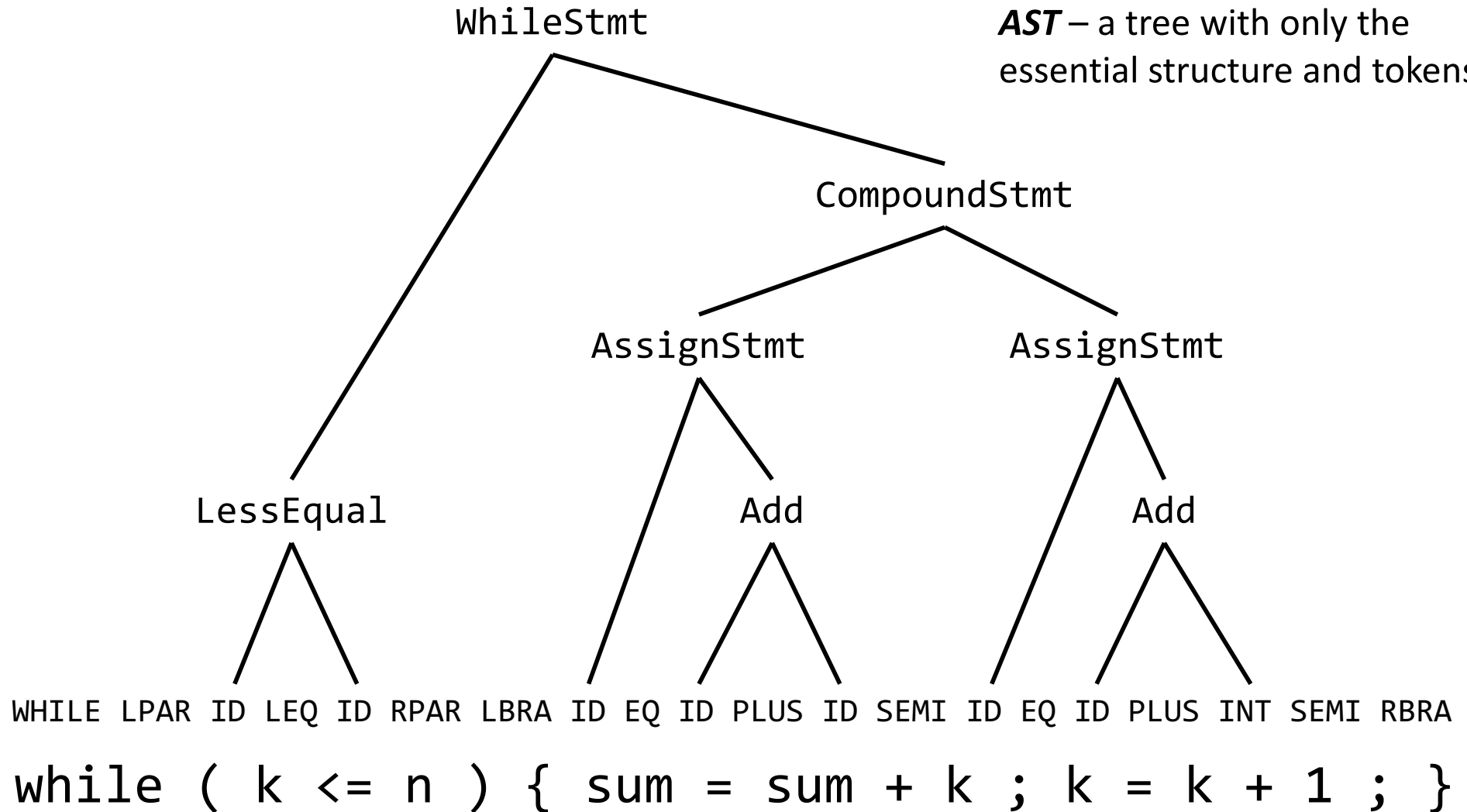


Syntactic analysis (parsing)



Abstract syntax tree (AST)

AST – a tree with only the essential structure and tokens



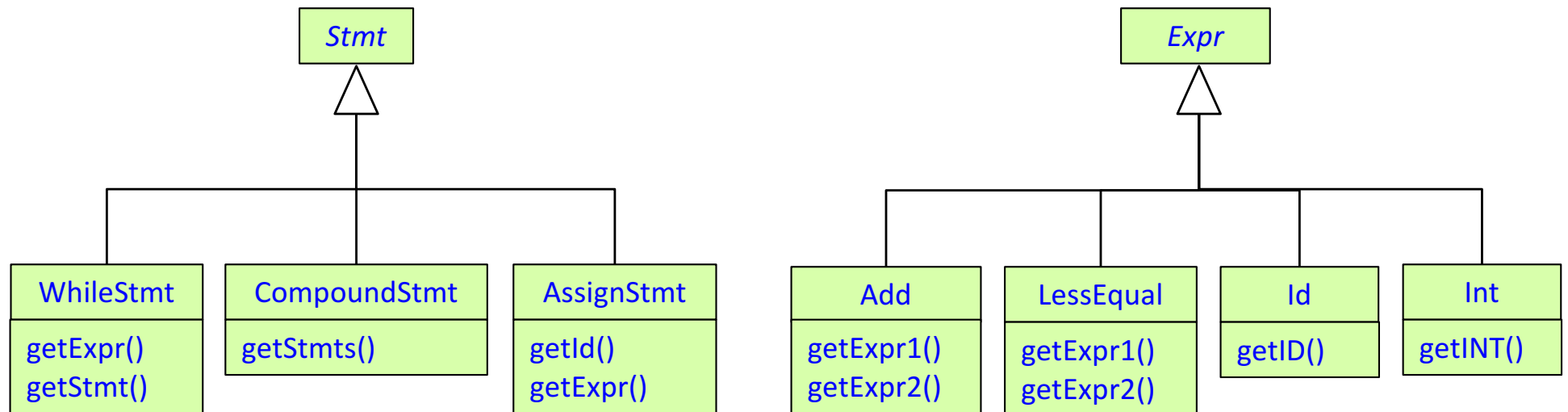
Abstract syntax trees

- 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, ...)

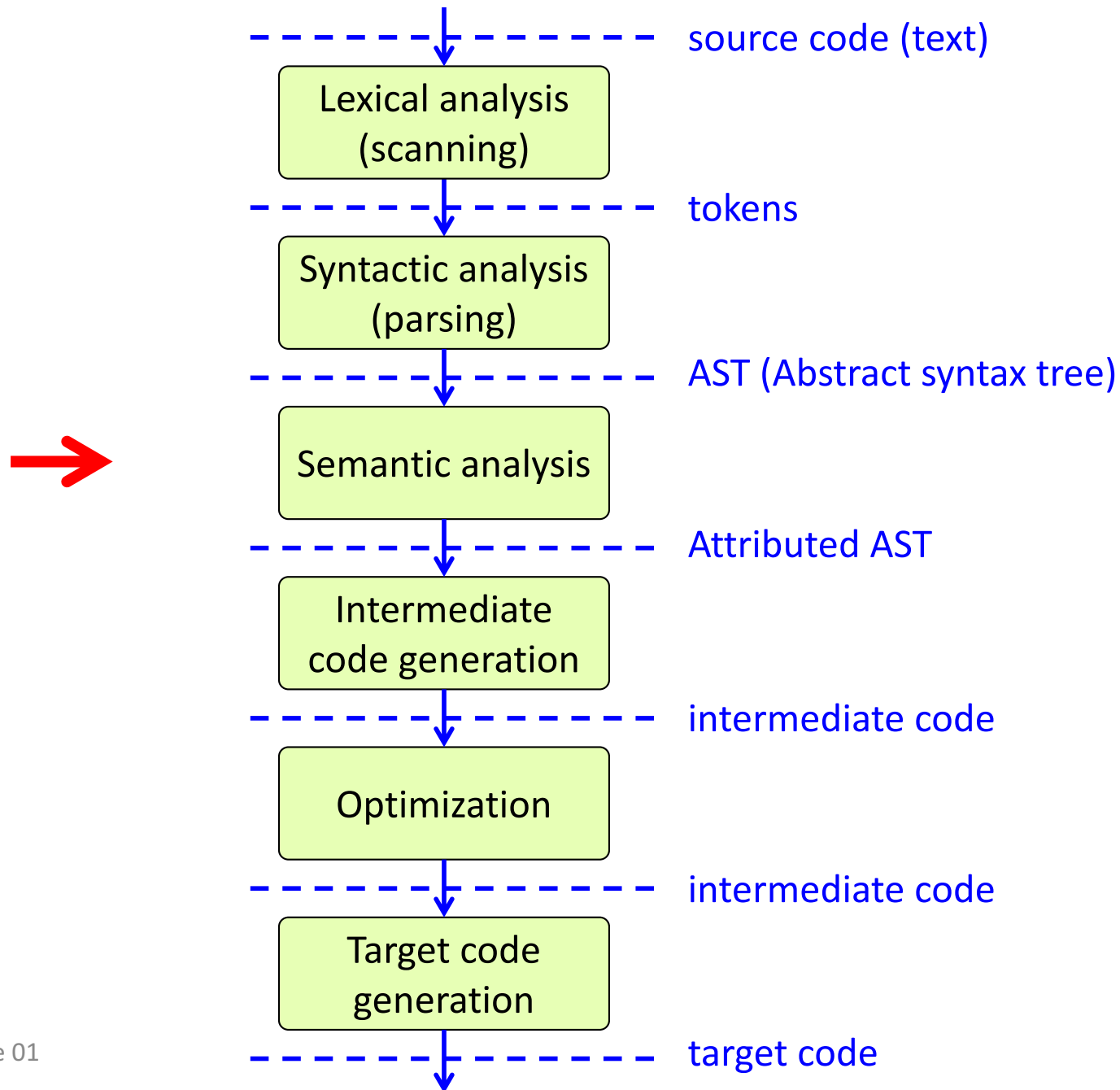
Designing an AST model (class hierarchy)

- What abstract constructs are there in the language
 - Make them abstract types
- What concrete constructs are there?
 - Make them subtypes
- What parts do the concrete constructs have?
 - Add getters for them, so the AST can be traversed

Example AST class hierarchy



Compiler phases and program representations:



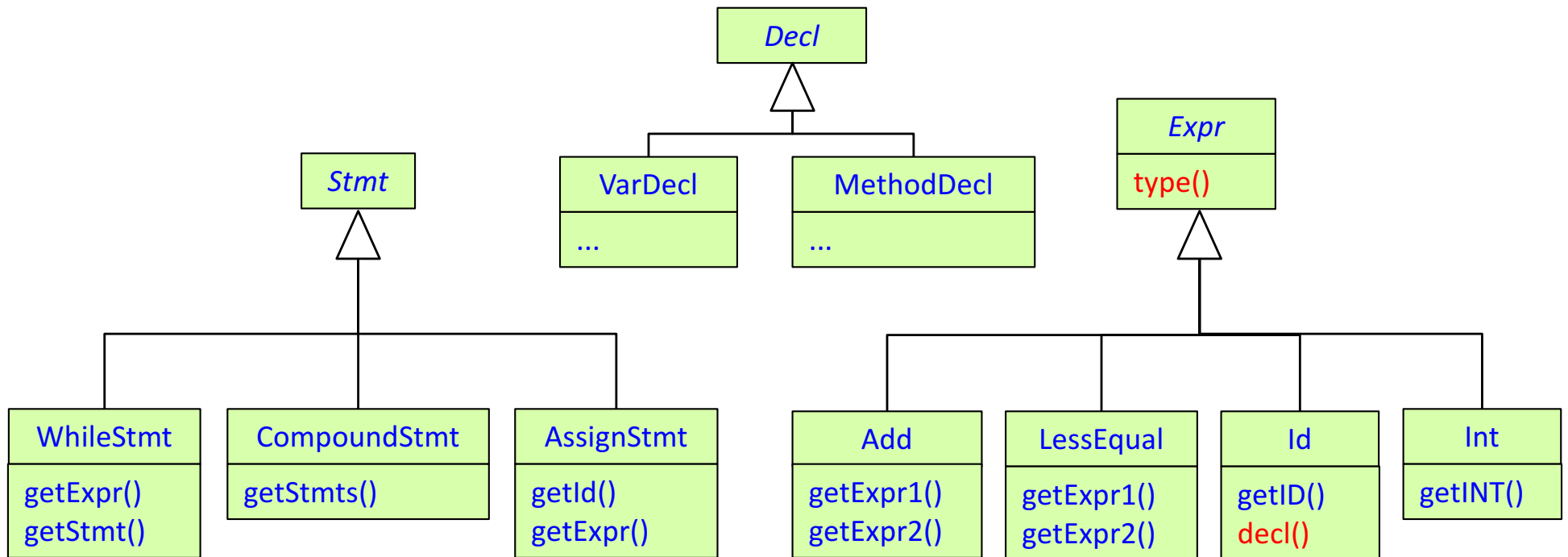
Semantic analysis

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?

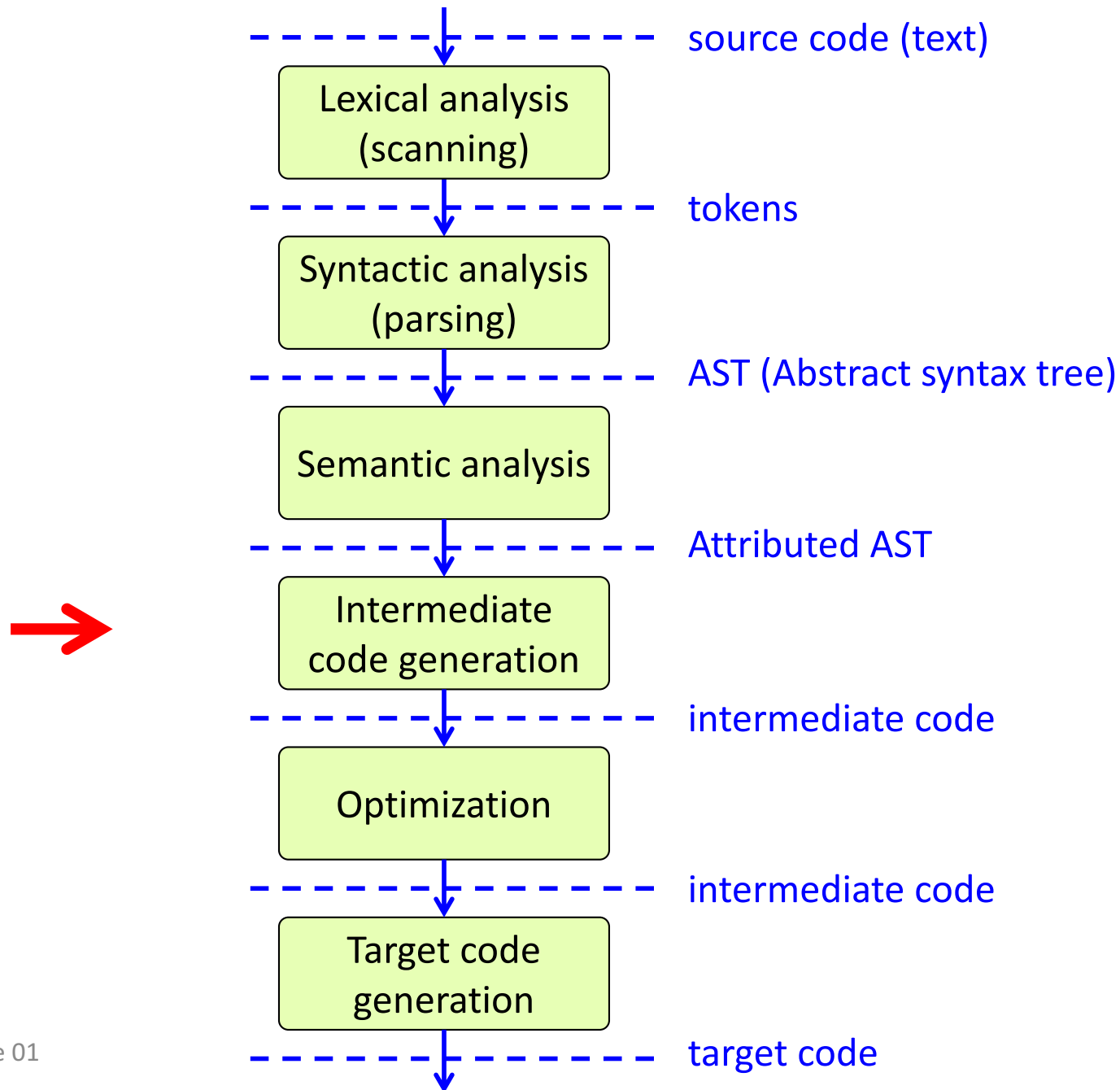
Analysis aided by adding *attributes* to the AST
(properties of AST nodes)

Example attributes



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:

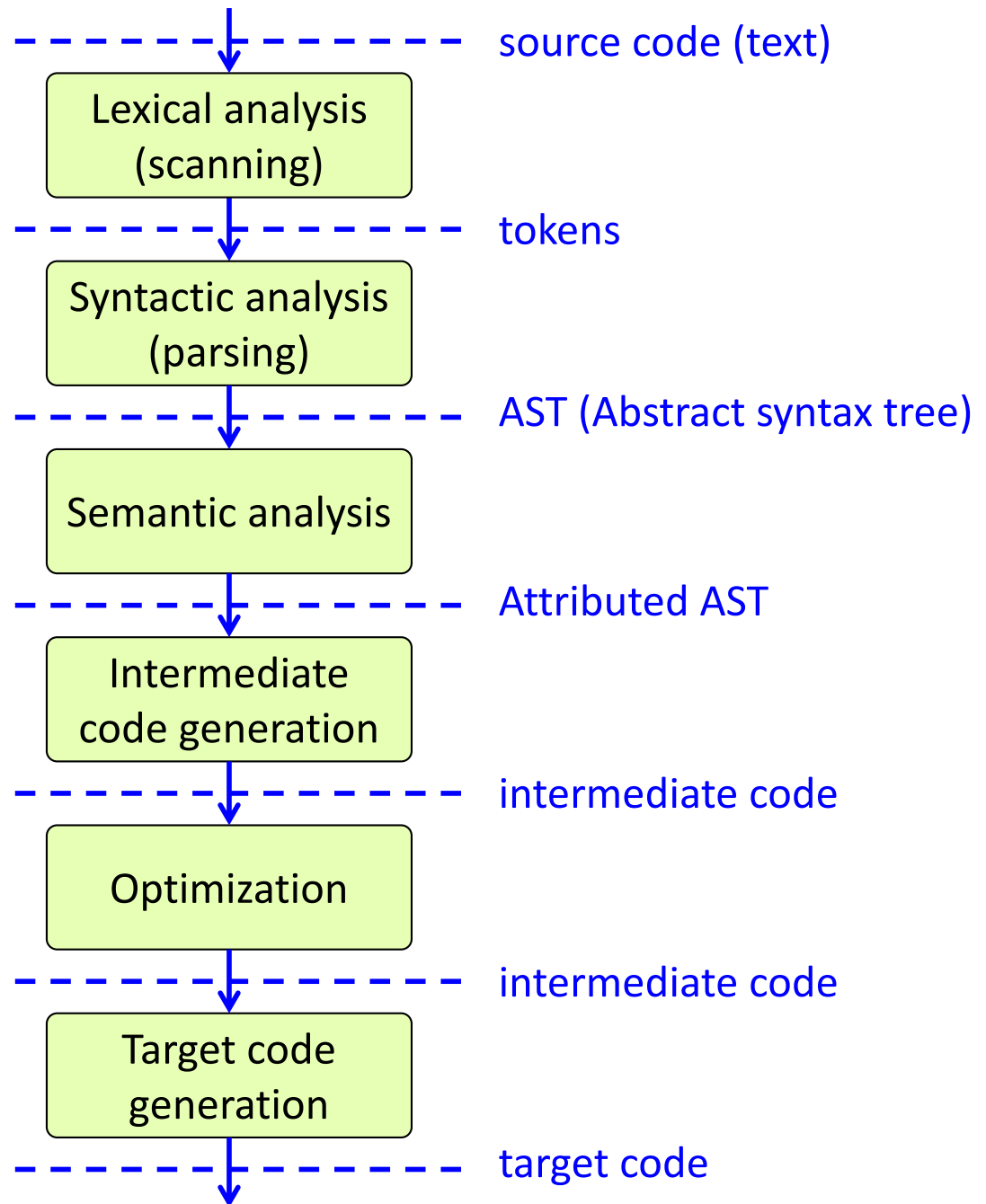


Intermediate code generation

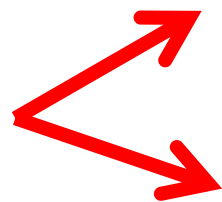
Intermediate code:

- 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)

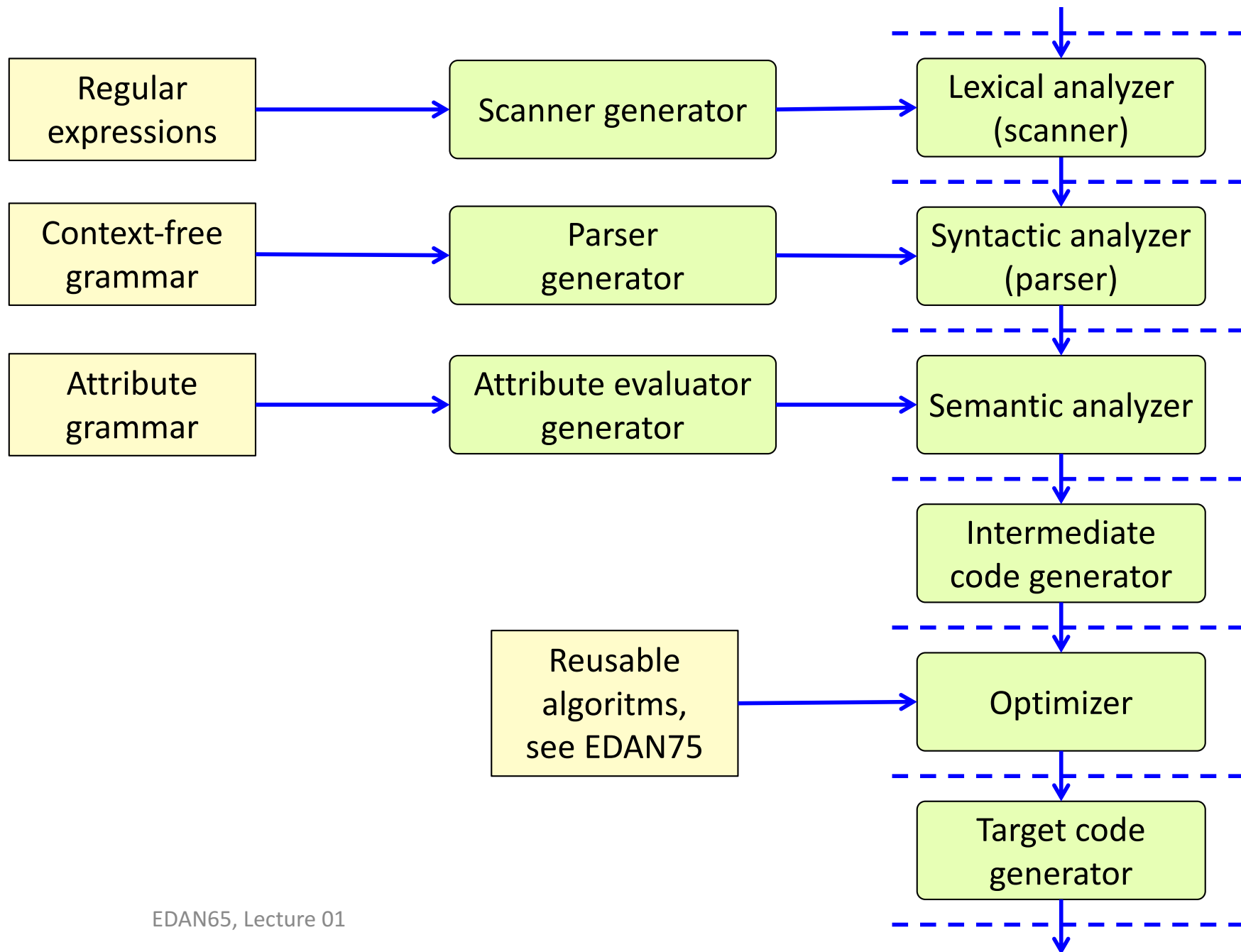
Compiler phases and program representations:



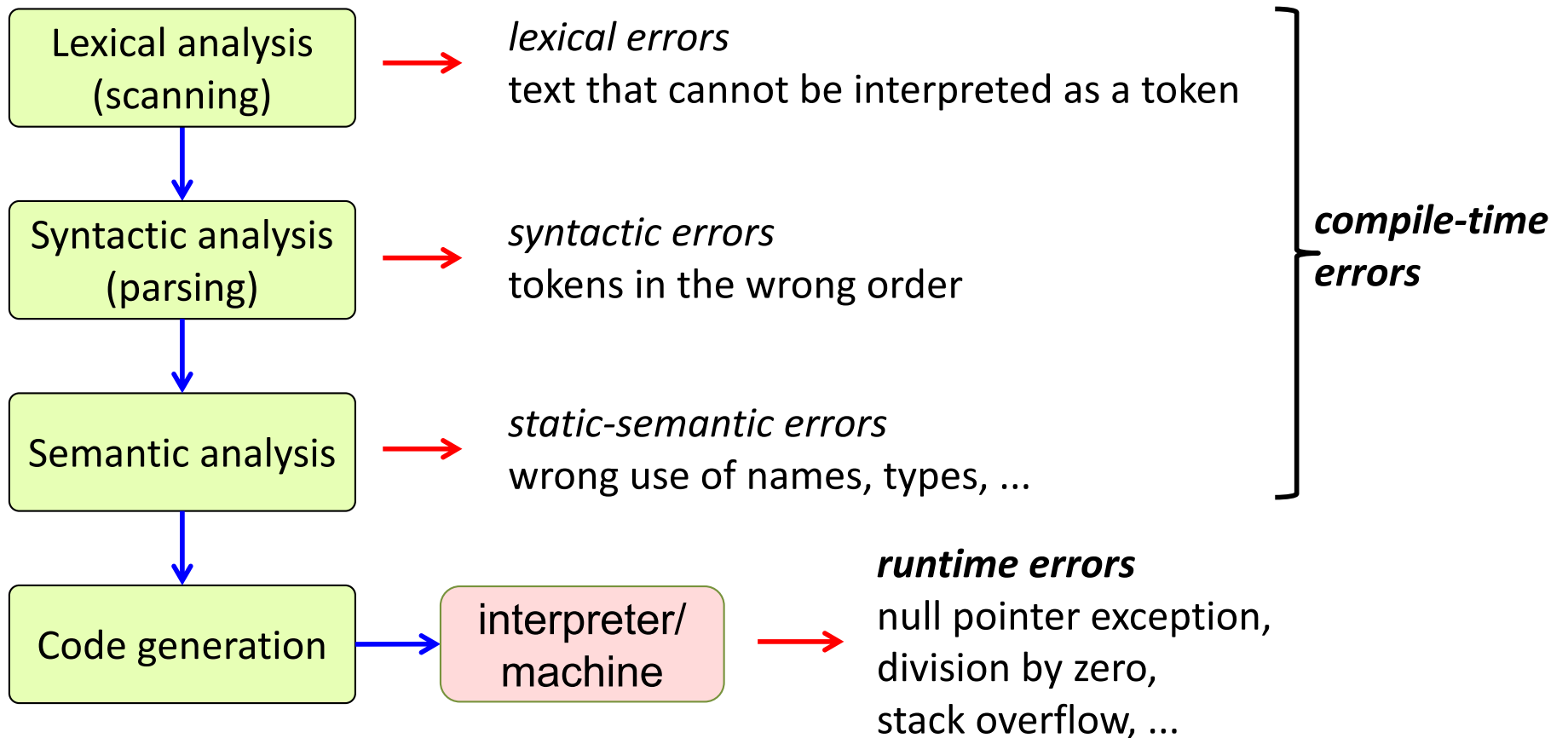
See the course Optimizing Compilers, EDAN75



Generating the compiler:



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.

Example errors

Lexical error:

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

Runtime error:

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

Syntactic error:

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

Logic error:

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

Static-semantic error:

```
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.

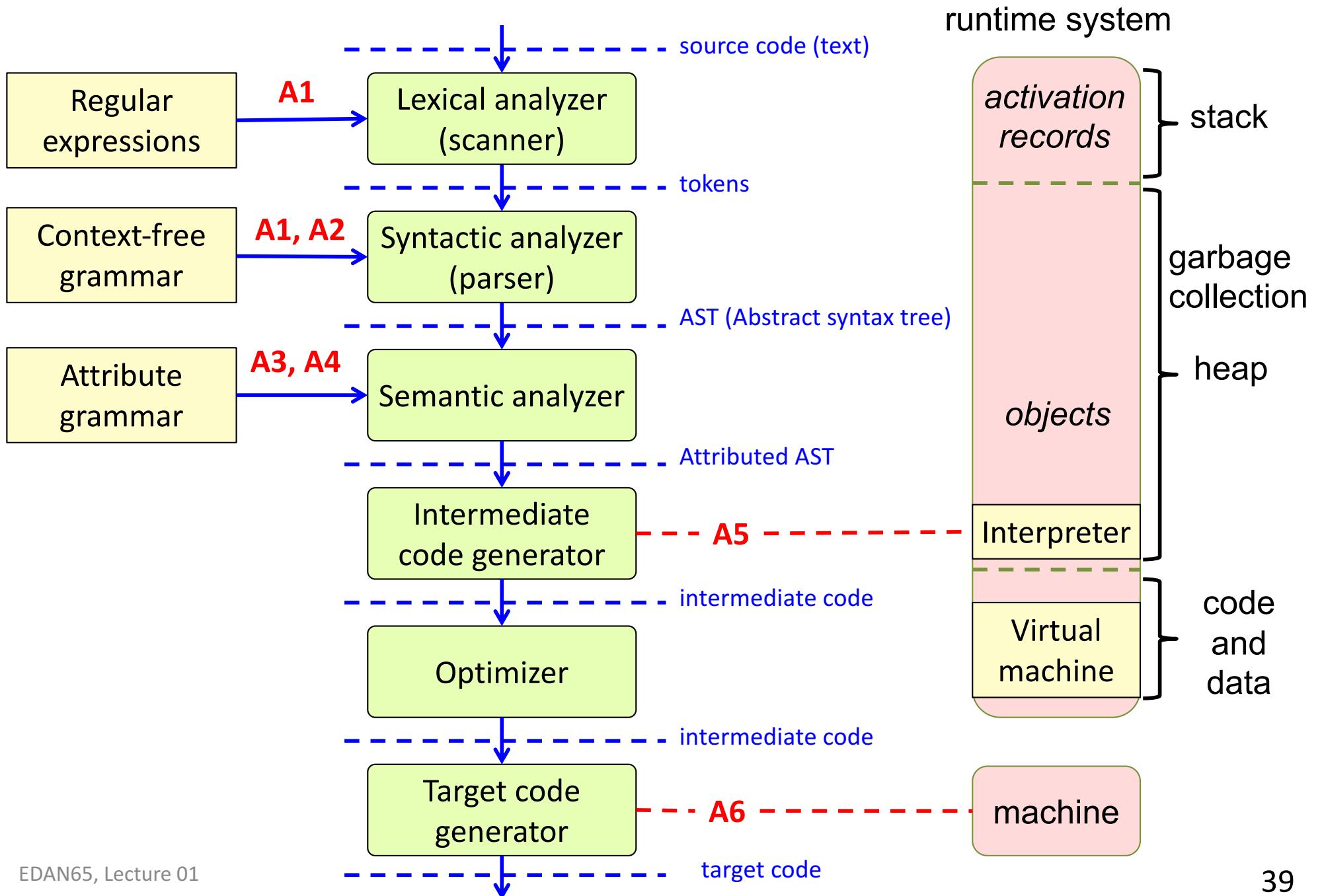
Examples: Java, C#, Smalltalk, Python, ...

- **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
- You might be continue with a compiler project in the EDAN70 course (Project in Computer Science)
- You might continue with a 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, preprocessors
- Interpreters and virtual machines
- Integrated programming environments
- Analysis tools
- Refactoring 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>
...
```

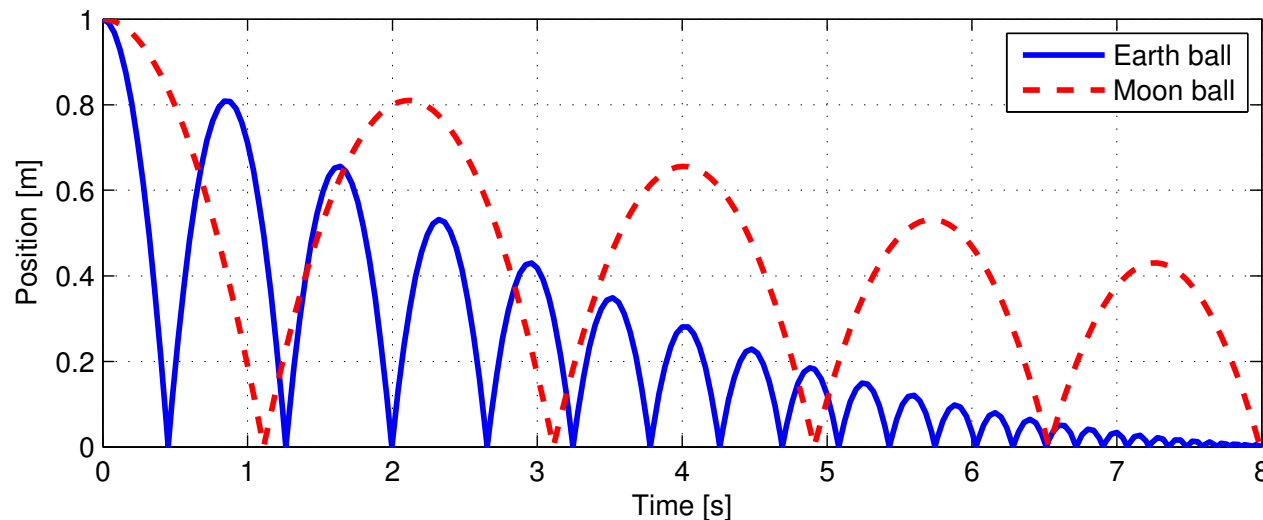
.gitconfig

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

Modelica

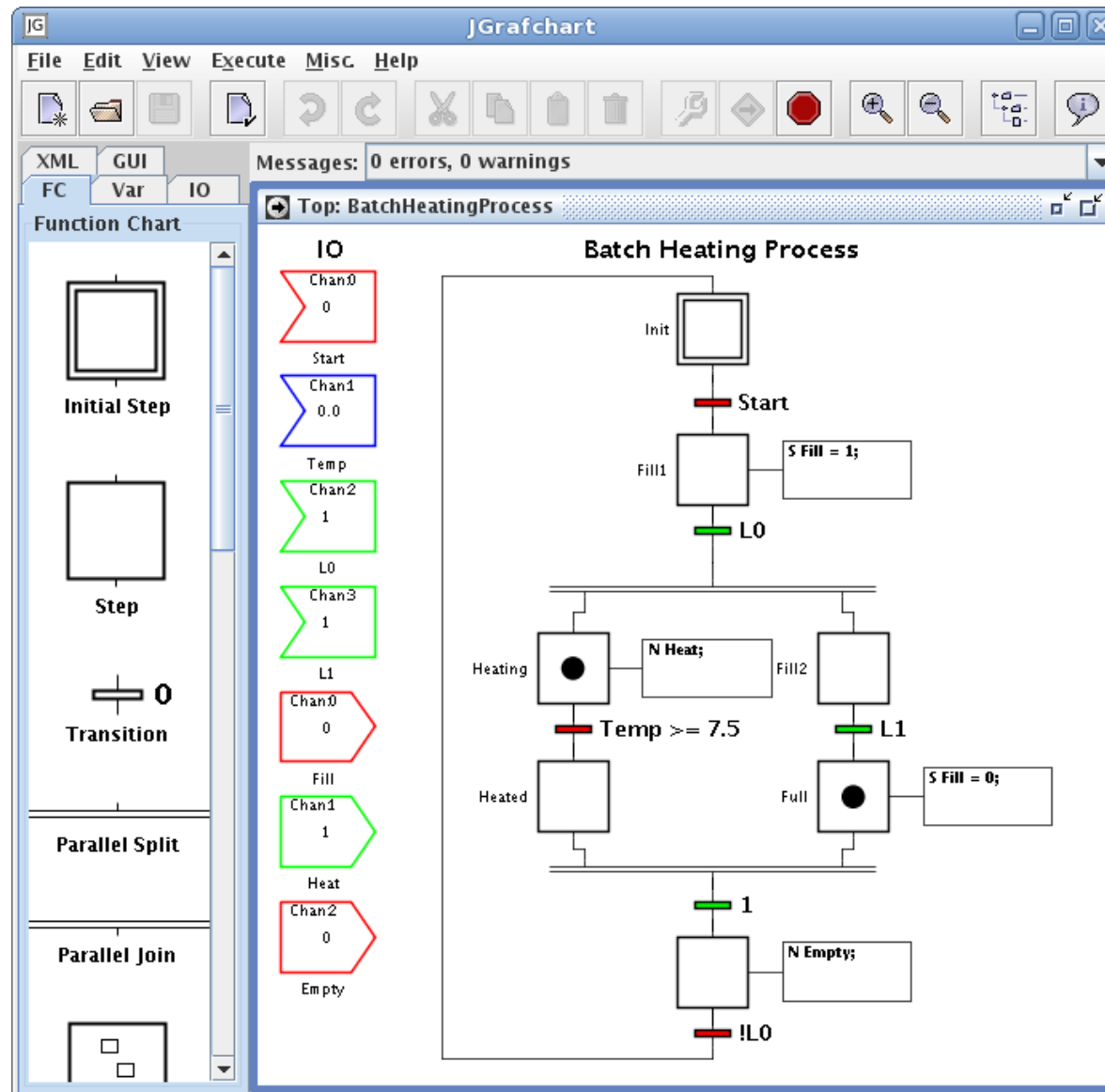
<http://www.modelica.org>

```
model BouncingBall //A model of a bouncing ball
  parameter Real g = 9.81; // Acceleration due to gravity
  parameter Real e = 0.9; // Elasticity coefficient
  Real pos(start=1); // Position of the ball
  Real vel(start=0); // Velocity of the ball
equation
  der(pos) = vel; // Newtons second law
  der(vel) = -g;
  when pos <= 0 then
    reinit(vel,-e*pre(vel)); // set velocity after bounce end when;
end BouncingBall;
```



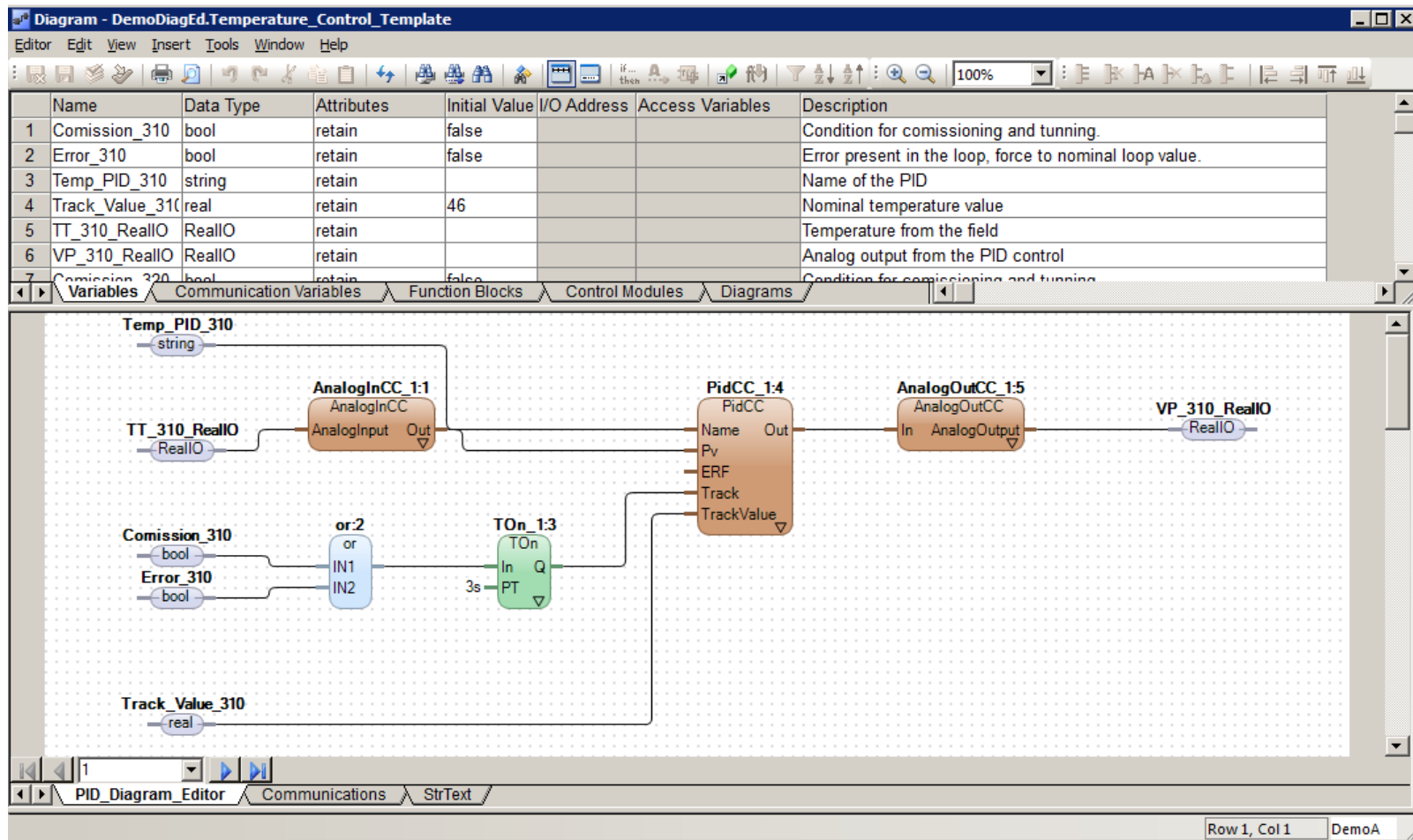
Grafchart

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



Control Builder Diagram

<http://new.abb.com>



Related research at LTH

- Extensible compiler tools (Görel Hedin)
- Real-time garbage collection (Roger Henriksson)
- Code optimization for multiprocessors (Jonas Skeppstedt)
- Natural language processing (Pierre Nugues)
- Constraint solver languages (Krzysztof Kuchcinski)
- Data-flow languages (Jörn Janneck)
- Languages for pervasive systems (Boris Magnusson)
- 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.