Welcome to CSI4104-01 on Compiler Design!

Bernd Burgstaller
Yonsei University

Outline

- Administrative issues
- Subject overview

General information

- Course title: Compiler Design
- Course ID: CSI4104-01
- Credits: 3
- Time and Place
  - Tuesday 16:00 — 16:50, B-041
  - Thursday 13:00 — 14:50, A-528

Staff

- Dr. Bernd Burgstaller
  - Department of Computer Science
  - **Email:** bburg@yonsei.ac.kr
  - **Consultation time:** Monday, 15:00 — 17:00 or by email appointment
  - Office: Eng. Building D, room D-910
- Teaching Assistants
  - Mr. Seongho Jeong and Mr. Yeonsoo Kim
  - Office: ELC Lab, Engineering Building D, Room D-711
  - http://elc.yonsei.ac.kr

About the lecturer…

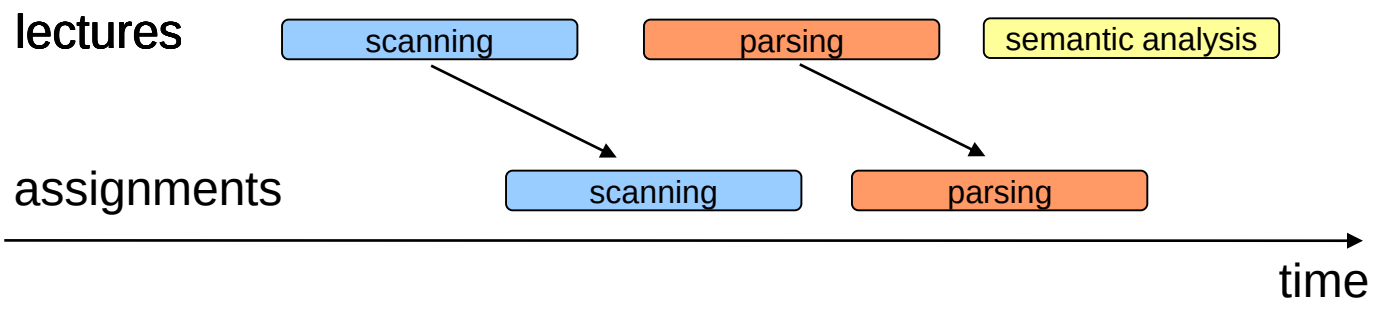
- Originally I am from Austria, Europe.
- MSc 1997, Vienna University of Technology
- 3 years industry experience with Philips Consumer Electronics (in embedded systems software development)
- 4 years as Predoc at the Vienna University of Technology
- PhD 2005, Vienna University of Technology
- 2005-2007: Postdoc at the University of Sydney, Australia
- 2007-2013: Assistant Professor at Yonsei University
- Since 2013/09: Associate Professor at Yonsei University

About the course language

- I am very far from being fluent in Korean yet. :(
- The course will therefore have to be in English.
- No safety-nets:
  - All lectures, homeworks, assignments, exams & questions will be in English.
  - Emails to the TAs have to be in English as well…
  - But I always try to read ``between the lines'' and consider language difficulties when grading.
- I will speak slowly
  - You should tell me **immediately** if I talk too fast.
  - You should ask **immediately** if something is unclear.
    I am happy to explain things in more detail.
- You can also ask questions via email, but then your colleagues won't benefit (your colleagues might have the same question!).

Teaching

- Theoretical parts of this course are presented in lectures.
- Assignments apply lecture materials to practical problems:



- We aim for a balanced approach between theory and practice.
- Homeworks repeat material needed for the exams:
  - After the homework due date, solutions will be provided.

Learning strategies

- Start early on assignments, and complete on time!
- Understand the theory presented in lectures and apply them when working on the assignments.
- Homeworks examples are worked out to ensure your understanding of the lecture materials and to prepare you for the exams.
- Consultations for this course:
  - My weekly consultation hours
  - YSCEC discussion board (ask/answer/learn from others)

Objectives

- Be able to build a compiler for a simplified programming language.
- Learn how to use compiler construction tools, such as generators for scanners and parsers.
- Become familiar with the Java virtual machine (JVM) and Java bytecode.
- Learn important compiler techniques, algorithms and tools.
- Improve your understanding of programming languages
  - syntax, semantics, type systems, scopes and variable bindings...
- Apply OO programming techniques on a medium-sized software project.
  - Software-engineering
  - Java programming skills

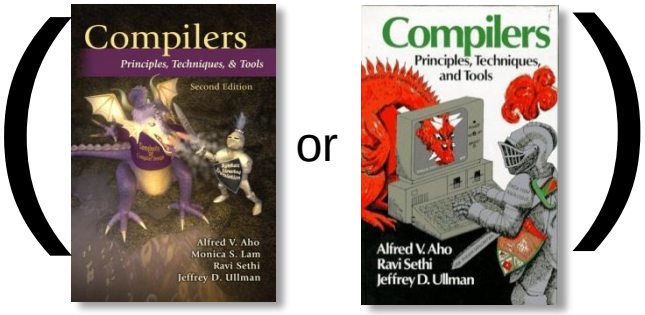
Acquired knowledge

- Regular expressions and finite state automata
  - Used with scripting languages (JavaScript, Python, Perl) but also C++
  - Used with editors, operating system shells, command line utilities
- Context-free grammars and parsing
  - Used with the definition of programming languages
  - Used with XML document type definitions
- Attribute grammars
- Type checking
- Java virtual machine (JVM) and Java bytecode
- OO programming techniques

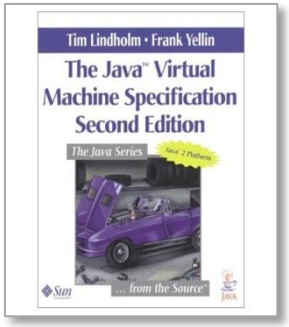
Acquired skills

- Be able to write scanners, parsers, semantic analysers and code generators.
- Know how to use compiler construction tools.
- Be able to specify syntax and semantics of a language.
- Be able to use algorithms and data structures used with compilers.
- Learn how to conduct an OO project in Java, using packages, inheritance, dynamic dispatching and design patterns.

Course materials



and



- Textbooks (reference)
  - Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison Wesley, 1st or 2nd edition
  - *The Java Virtual Machine Specification*, 2nd edition, downloadable from http://docs.oracle.com/javase/specs/
- YSCEC:
  - Lecture slides
  - Assignments, homeworks, solutions.
- Lecture slides may refer to textbooks (for background material and suggested readings).

Extensions

- Assignments and homeworks should be turned in before or at the due date before midnight.
- Late assignments:
  - 5% deducted from the assignment grade per day, until assignment is received.
  - Maximum extension of five days.
  - Example: assignment scores 100%, but 2 days late → 90%
- No extensions for homeworks
  - Reason: homework solutions are provided after the due date.

Marking Criteria for Assignments

- Evaluation based on test cases
  - A testcase is a file with a **few** lines of source code:

```
bool
```

```
BOOL
```

```
int a;
int b = 1.0;
void foo() { }
float foo, foo1;
```

  - Idea: understand aspects of the compiler and programming language, **not** about compiling large programs!
  - We will provide you with *some* test cases for each assignment, but you are invited to **invent further testcases**
    → facilitates your understanding
- Grade of assignment determined by the number of passed test cases.
  - Objective evaluation

Marking Criteria for Assignments (cont.)

- All programming assignments must be handed in on our server.
  - elc1.cs.yonsei.ac.kr
- Your code must work on the server!
  - Only solutions that can be compiled, do not crash nor enter an ``endless'' loop, and produce meaningful output can receive points.
  - If a solution works correctly for all test cases, it will receive maximum points.

Grading Policy

- 50% exams (25% midterm, 25% final)
- 35% assignments:
  - 5% Assignment 1
  - 5% Assignment 2
  - 6% Assignment 3
  - 10% Assignment 4
  - 9% Assignment 5
- 10% homeworks
- 5% participation

- Attendance of the lectures is recommended and expected.
- Missing a class does not influence your grade, except that...

> *University regulations set an upper bound on the number of missed classes. Exceeding the upper bound results in grade F.*

- Attendance is assessed **online**, at the beginning of class.
  - **No late attendance checks**.

Plagiarism

Yonsei University precautions against cheating and dishonesty:

"According to the University regulations, the student will get an "F" grade for the course that (s)he cheated on, and all the other courses for which exams have not yet been written at the time of the offence in the same semester will be granted a "W" (withdraw) grade."

Note: Assignments and homeworks are **individual**. You are **allowed to discuss** your solutions with colleagues, but you are **not allowed to share code**. Copying code (including "smarter" copying by changing variable names aso) counts as **plagiarism** and will be treated as outlined above.

Assignments Prepare You for the Exams

- You are expected to apply the techniques acquired with the assignments to exam problems.
- Be considerate about the assignments and come up with your own solutions.

Prerequisites

Required courses:

- Linear Algebra
- Automata and Formal Languages

Be familiar with:

- Algorithms and data structures
- Programming languages
- Java or C++ programming

Course outline (tentative)

- **Introduction to Programming Languages and Compilers**: high-level programming languages, principles and phases of a compiler.
- **Lexical analysis**: how to split program source code into tokens; discussion of regular expressions and automata.
- **Syntactic analysis:** analyzing the structure of a program; context-free grammars, recursive decent parsers, abstract syntax trees.
- **Semantic analysis:** checking the meaning of programs; attribute grammars, type checking.
- **Run-time organization:** data representation, storage allocation (static, stack, heap).
- **Code generation:** how a compiler generates code for a particular architecture; introduction to the Java virtual machine, Java bytecode, code optimizations.
- **Programming Language Implementation Techniques**: Just-in-time and Ahead-of-time compilation, virtual machine implementation techniques.

Expectations from students

- Work hard! :)
- Take initiative
  - Learn the details by yourself.
  - Do not wait till the last minute to get going.
- Complete your profile in YSCEC
  - Email address
  - Photo (to help me memorize your names)

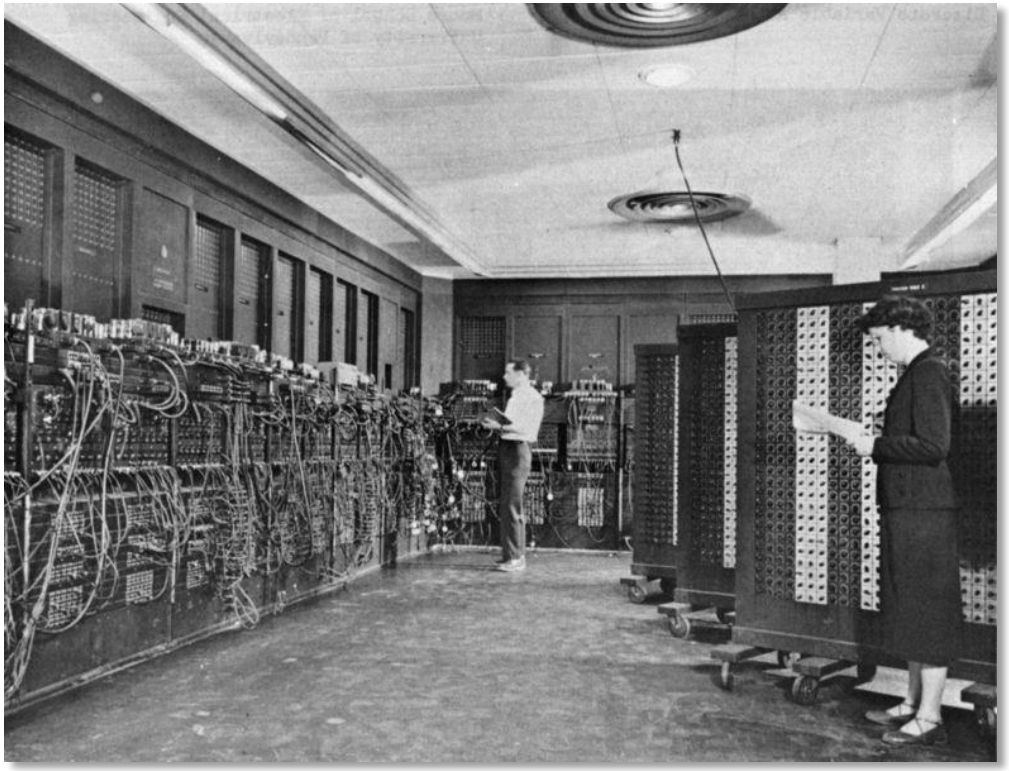
Good Luck!*

** Luck is where preparation meets opportunity.*

Outline

- Administrative issues ✓
- Subject overview

Back in the old days…



ENIAC (1946)

- 1940s: First electronic computers where monstrous beasts.
- Programmed in binary machine code via switches and later on via card readers.
- Code was not reusable.
- Computation and machine maintenance where difficult, vacuum tubes regularly burned out.
- The term “bug” originated from a bug that roamed around in a machine, causing short circuits.

Assembly Languages (1)

- Assembly languages where invented to allow machine operations to be expressed in mnemonic abbreviations.

```
    addiu    sp,sp,-32
    sw       ra,20(sp)
    jal      getint
    nop
    jal      getint
    sw       v0,28(sp)
    lw       a0,28(sp)
    move     v1,v0
    beq      a0,v0,D
    slt      at,v1,a0
A:  beq      at,zero,B
    nop
    b        C
    subu     a0,a0,v1
B:  subu     v1,v1,a0
C:  bne      a0,v1,A
    slt      at,v1,a0
D:  jal      putint
    nop
    lw       ra,20(sp)
    addiu    sp,sp,32
    jr       ra
    move     v0,zero
```

Example MIPS assembly program to compute GCD

- Example MIPS R4000 machine code of the assembly program
- There is (usually) a one-to-one correspondence between assembly language instructions and machine-code.
  - Straight-forward to translate.
- A program called **assembler** translates assembly language to machine code.

assemble

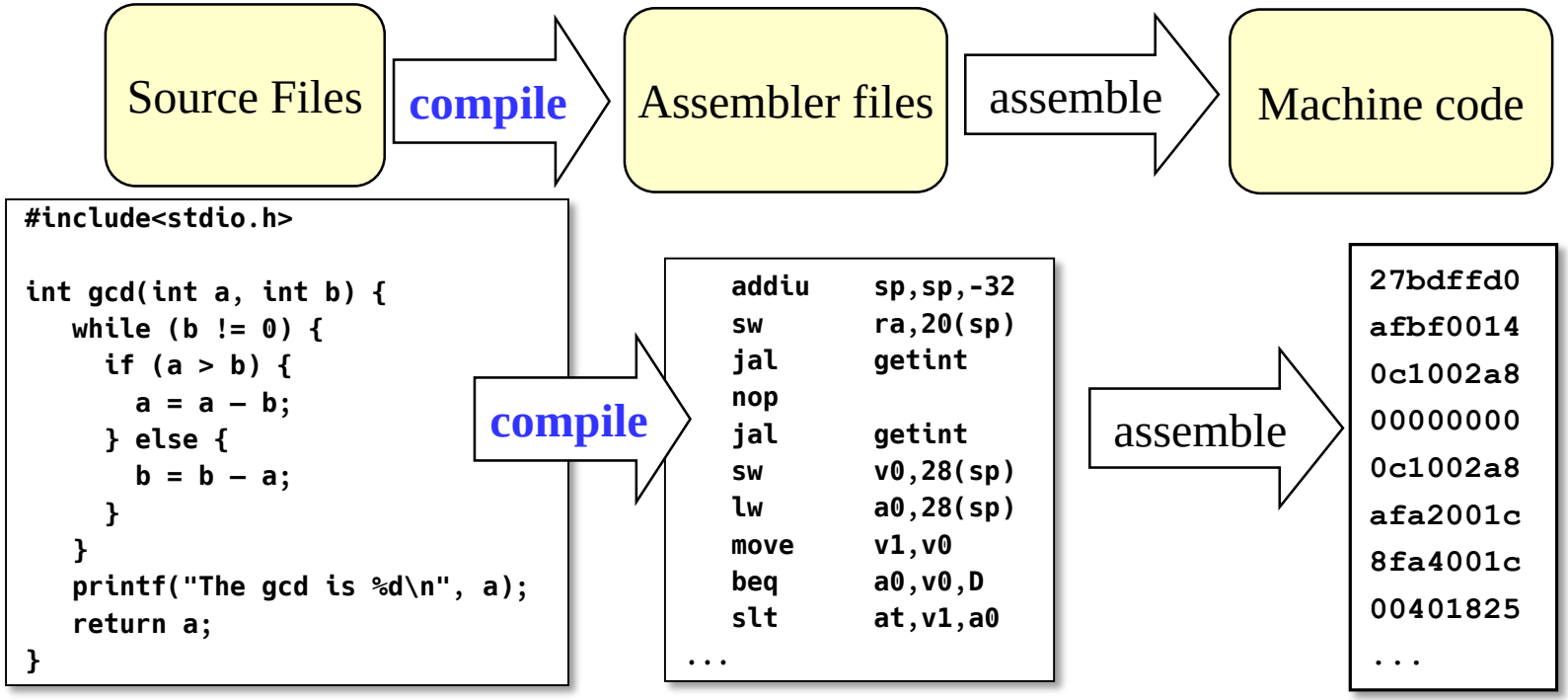
```
27bdffd0
afbf0014
0c1002a8
00000000
0c1002a8
afa2001c
8fa4001c
00401825
10820008
0064082a
10200003
00000000
10000002
00832023
00641823
1483fffa
0064082a
0c1002b2
00000000
8fbf0014
27bd0020
03e00008
00001025
```

Assembly Languages (2)

- Advantages of assembly language over machine code:
  - Mnemonics are easier to remember for humans
  - Allows reusable (on same architecture!) and relocatable programs
  - A program called “assembler” translates mnemonics to machine code (more productive than manual translation!)
  - Macros: expansion of a programmer-defined abbreviation into multiple machine instructions (a form of high-level programming or abstraction).

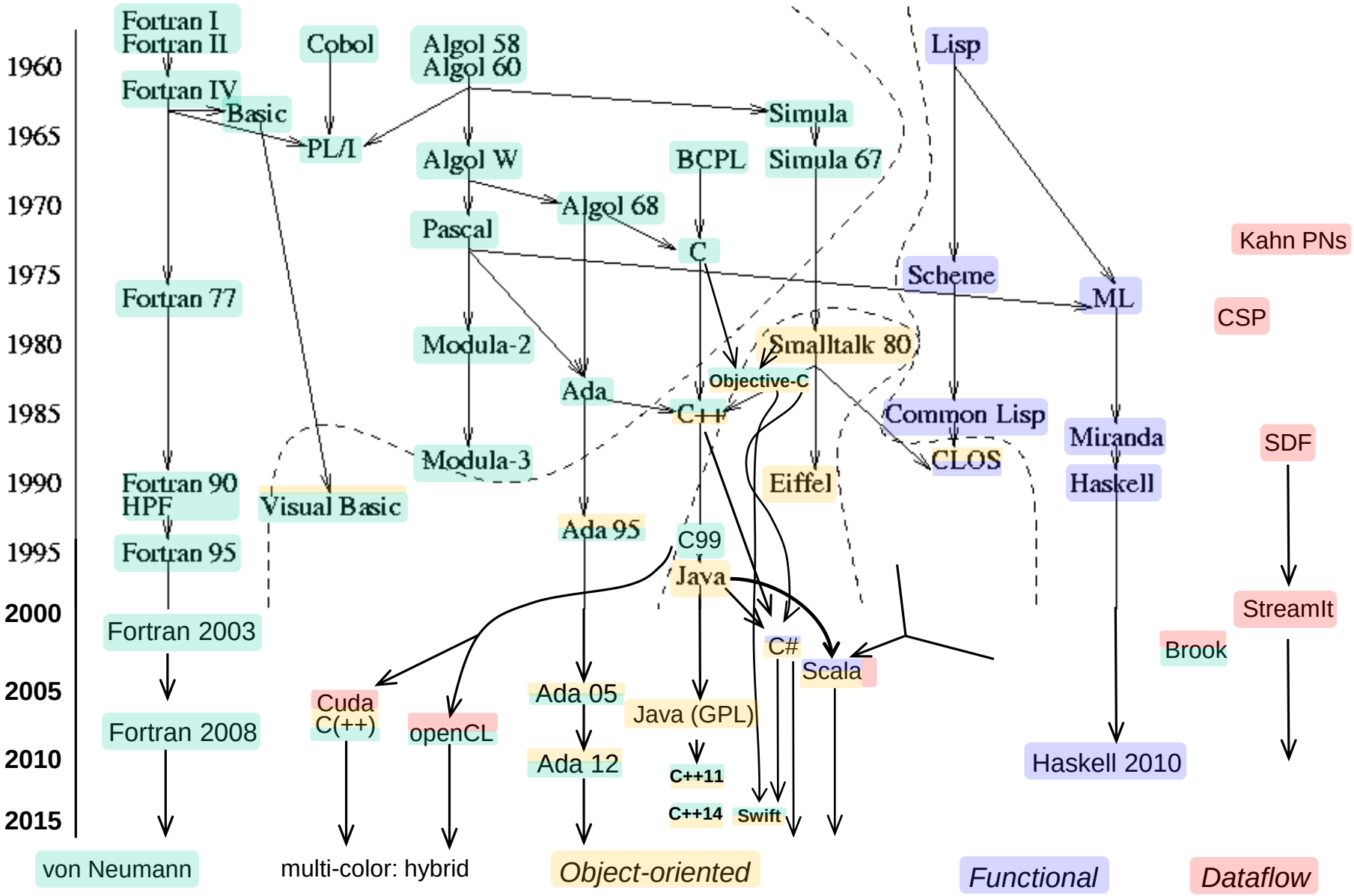
High-Level Programming Languages

- 1950s: development of FORTRAN (FORmula TRANslator), the first higher-level language.
- Finally, compilers enabled the development of programs that are machine-independent!



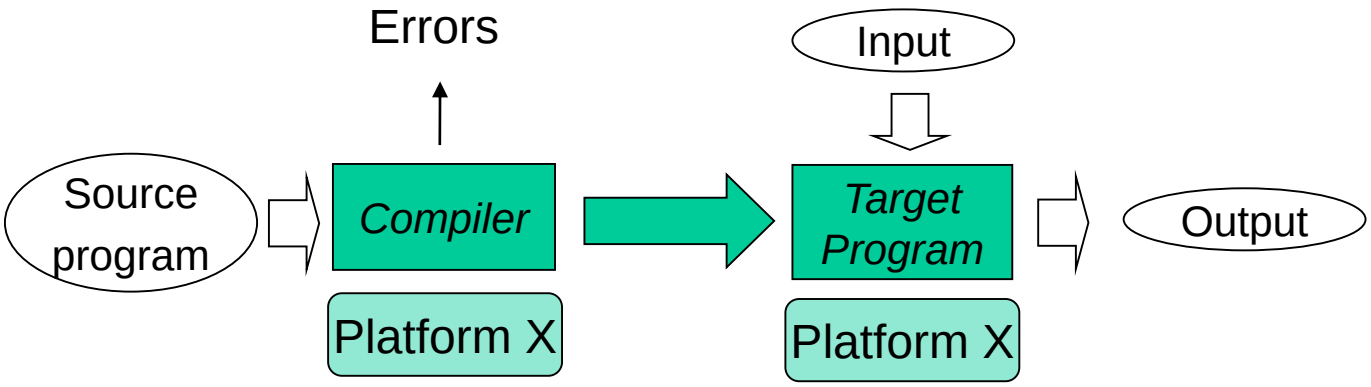
Example: C routine to compute GCD of two integers

Other High-level languages soon followed:



Compilation

- A compiler translates a high-level source program into a target program in machine language.



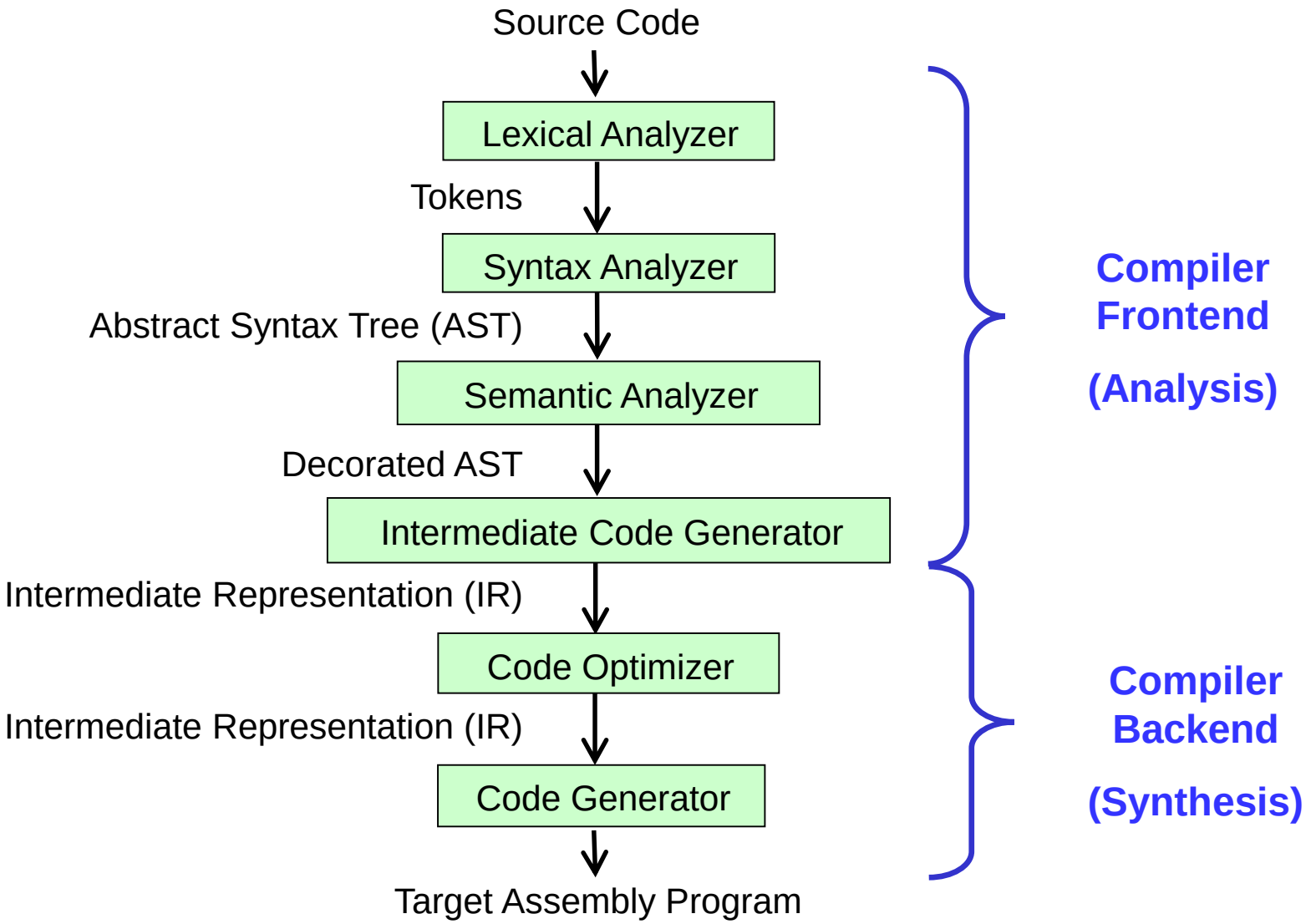
Platform X: any combination of operating system + target hardware

- Compiler recognizes legal and illegal programs.
- Generates correct and hopefully efficient code.
- At a later time, the user runs the target program.

The Analysis-Synthesis Model of Compilation

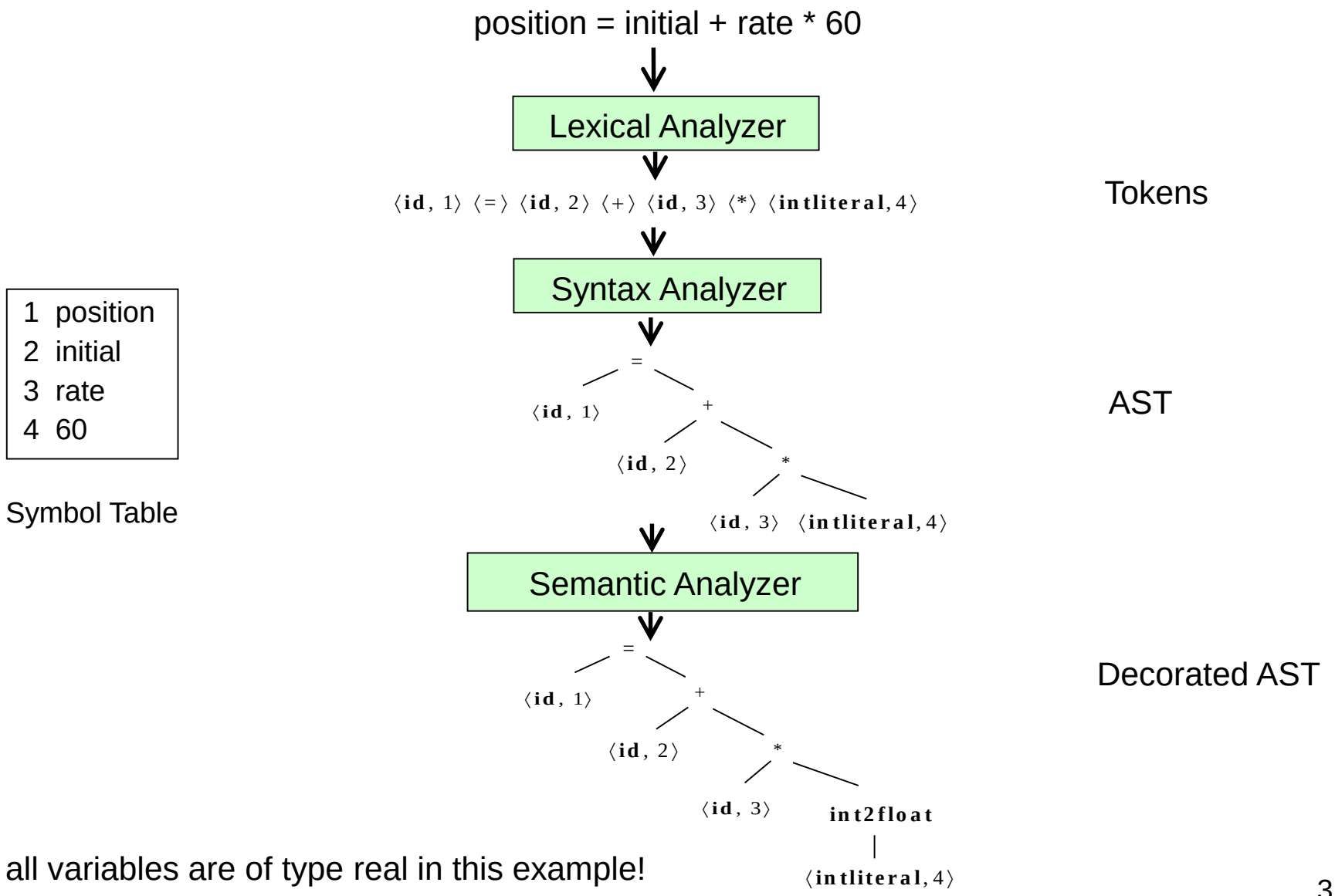
- Compilation consists of two parts:
  - Analysis determines the operations expressed in the source language
    - Think of ``*understanding the input program*''
    - Program's operations recorded in a tree structure
  - Synthesis takes the tree structure and translates the operations into the target program.
    - Think of ``*generating the target program corresponding to the input program*''

Structure of a Compiler



…each phase transforms the program from one representation to the next…

Example for Register-based Machines

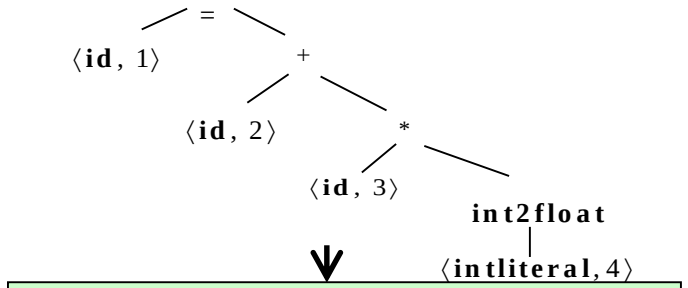


Note: all variables are of type real in this example!

Example (continued)

1	position
2	initial
3	rate
4	60

Symbol Table



Intermediate Code Generator

```
t1 = int2float (60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

IR

Code Optimizer

```
t1 = id3 * 60.0
id1 = id2 + t1
```

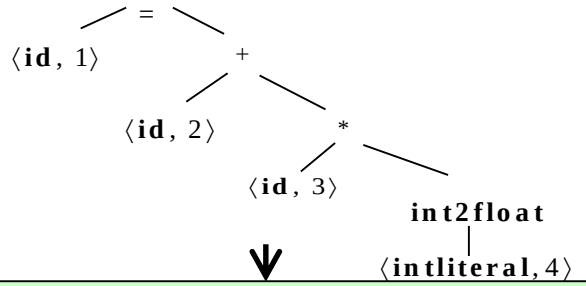
IR

Code Generator

ARM assembly code

```
LDF R2, id3          // load id3 into register R2
MULF R2, R2, #60.0   // R2 := R2 * 60.0
LDF R1, id2          // load id2 into register R1
ADDF R1, R1, R2      // R1 := R1 + R2
STF id1, R1          // store R1 in id1
```

Example (continued)



1	position
2	initial
3	rate
4	60

Symbol Table

Intermediate Code Generator

```
t1 = int2float (60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

Code Optimizer

```
t1 = id3 * 60.0
id1 = id2 + t1
```

Code Generator

```
LDF R2, id3          // load id3 into register R2
MULF R2, R2, #60.0   // R2 := R2 * 60
LDF R1, id2          // load id2 into register R1
ADDF R1, R1, R2      // R1 := R1 + R2
STF id1, R1          // store R1 in id1
```

Java bytecode (explanation on slides 44-45):

```
fload_2
fload_3
bipush 60
i2f
fmul
fadd
fstore_1
```

Lexical Analysis

- Done by the *Scanner.*
- Groups characters into tokens

  `position = initial + rate * 60`

  becomes
  1. The token ⟨**id**, 1⟩ for identifier "`position`"
  2. The token ⟨=⟩ for assignment operator "="
  3. The token ⟨**id**, 2⟩ for identifier "`initial`"
  4. The token ⟨+⟩ for the add operator "+"
  5. The token ⟨**id**, 3⟩ for identifier "`rate`"
  6. The token ⟨*⟩ for the multiplication operator "`*`"
  7. The token ⟨**intliteral**, 4⟩ for the integer literal "`60`"
- Character string making up a token is called *lexeme*.
  - "position" is the lexeme for token ⟨**id**, 1⟩ above.
- Scanner eliminates white space (blanks, tabs, newline) and comments.
- Scanner generates a **lexical error** in case of **malformed tokens**
  - un-terminated strings, illegal characters in tokens, ...

Syntax Analysis

- Done by the Parser.
  - Organizes tokens into a tree (AST). Interior nodes represent operations, children represent the arguments of that operation (see example on Slide 32).
- The **syntax** (or structure) of a language is defined by a **context-free grammar (CFG).** The CFG determines the **structure** of ASTs.
- Simple example CFG for expressions without operator precedence:

exp ::= **intliteral**
| **id**
| *exp* + e*xp*
| e*xp* – e*xp*
| e*xp* * e*xp*
| e*xp* / e*xp*

- The parser will generate a **syntax error** if it cannot create a parse tree (or AST) for the input program.
  - Such a faulty program does **not adhere to the CFG** of the programming language.

Semantic Analysis

- Semantic analysis discovers the meaning of a program by analyzing its AST. Checks for consistency with the language definition.
  - **Static** (at **compile time**) **semantic checks** ensure that
    - every identifier is defined before it is used,
    - no identifier is used in an inappropriate context (calling an integer as a subroutine, adding a string to an integer,…),
    - subroutine call arguments are ok,
    - type checking
      - real values cannot index an array, e.g., ~~a[2.6]~~
      - type conversions (e.g., int2float)
- Some semantic checks have to be deferred until **run-time** (→ **dynamic semantic checks**):
  - Array subscripts are within bounds, e.g., a[y]
  - Arithmetic errors (division by zero,…)
- A program that fails a semantic check has a **semantic error**!

Intermediate Code Generation

- Translates AST into IR
- Important IR properties:
  - ease of generation
  - ease of translation into machine instructions
- Subtle decisions in the IR design may have major impact on speed and effectiveness of the compiler.
- Popular IRs:
  - ASTs
    …used by GNU GCC
  - Directed acyclic graphs (DAGs)
  - Three address code (3AC or quadruples)
  - Static single assignment form (SSA)
    ... used by the LLVM compiler infrastructure project
    ... used by GNU GCC

Code Optimization

- Code optimizer
  - Analyzes and improves IR
  - Goals:
    - reduce execution-time, footprint, power-consumption
    - make program more robust
  - Must preserve the initial semantics of the program!

- Typical optimizations:
  - Discover and propagate a constant value.
  - Remove code that is useless or unreachable.
  - Move a computation to a less frequently executed place (e.g., out of a loop body).
  - Do things **in parallel**.
  - Examples: see next slides.

Code Optimization (Example 1)

Input for Pass 1

```
a = 1;
b = 2*x;
if (a+9 > 100) {
    foo (3*a);
    a++;
} else {
    foo (b);
}
foo(a+1);
```

→

Input for Pass 2

```
a = 1;
b = 2*x;
if (10 > 100) {
    foo (3);
    a++;
} else {
    foo (b);
}
foo(a+1);
```

→

Input for Pass 3

```
a = 1;
b = 2*x;
foo (b);
foo(a+1);
```

→

```
a = 1;
b = 2*x;
foo (b);
foo(2);
```

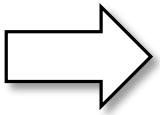
(We assume that “a” is a local variable, i.e., function foo does not modify “a”!)

- Pass 1: Constant propagation replaces “a+9” by “10” and “3*a” by “3”.
- Pass 2: Dead code elimination eliminates the “then” branch of the if statement.
- Pass 3: A second constant propagation pass can now replace expression “a+1” by the value 2.

Code Optimization (Example 2)

```
i = 0;
while (i < m) {
  j=0;

  while(j<n) {
    t1 = i*n + j;
    j = j + 1;
  }
  i = i + 1;
}
```



```
i = 0;
while (i < m) {
  j=0;
  t2 = i*n;
  while(j<n) {
    t1 = t2 + j;
    j = j + 1;
  }
  i = i + 1;
}
```

- Loop invariant: an expression that is always computed to the same value, each iteration of the loop. Example: "i*n" above.
- Optimization: move loop invariant outside of loop
  - Introduce t2=i*n and replace i*n by t2.
  - Move t2=i*n outside of loop.

Code Generation

- Code generator creates target code
  - either relocatable machine code or assembly code.
- Chooses target instructions for each IR instruction
- Decides what variables to keep in registers at each point of the program.
  - Number of registers is limited, usually much smaller than number of variables.
  - Assignment of variables to registers is therefore a crucial aspect of the code generator.

Compiler Phases, Theory and Tools

Compiler Phase	Theory	Tools
Lexical analysis	REs, NFA, DFA	scanner generators (Lex, JLex, SableCC, ANTLR)
Syntax analysis	Context-free grammars, parsers, e.g., LL(k), LR(k)	Parser generators (Yacc, JavaCC, SableCC, ANTLR, parser combinators)
Semantic analysis	Attribute grammars, type checking	Yacc, SableCC, ANTLR
Code optimizations	Theory on program analysis and optimizations	Data-flow engines, LLVM framework
Code generation	Syntax-directed translation	Automatic code generators, e.g., iburg.

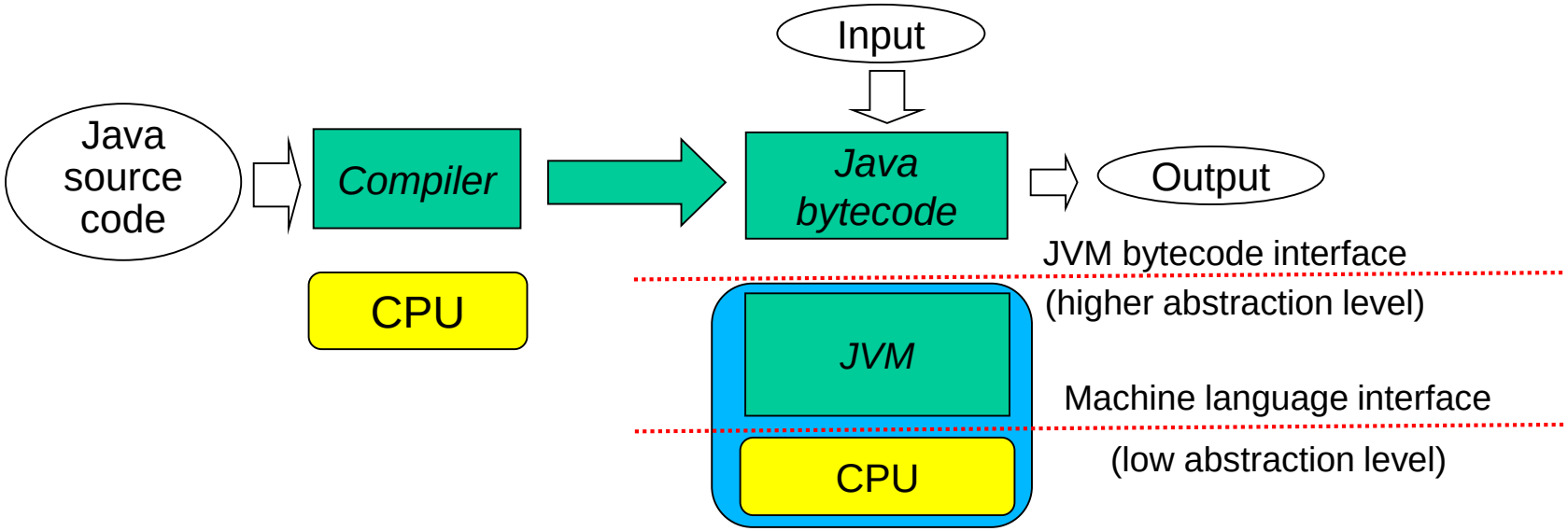
Error Detection, Error Reporting and Recovery

- The compiler has to **detect** illegal programs (programs that do not adhere to the specification of the programming language):
  - **Lexical errors**, e.g., “abc (un-terminated string)
  - **Syntax errors**, e.g., forgetting a closing parenthesis: a + (2*b
  - **Semantic errors**, e.g., variables that are used but not declared.
- The compiler should **report** the location of the error as accurately as possible.
- After detecting an error, the compiler may **recover** and proceed
  - to detect further errors in the source program.

Error recovery is *optional* with the compiler project of this course.

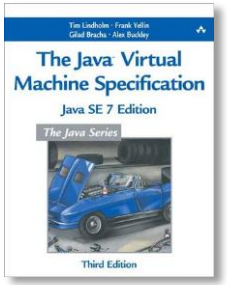
A Brief JVM & Java Bytecode Primer…

- The JVM is a stack-based virtual machine
  - uses a stack instead of registers as intermediate storage for values of a computation.
    - See example on next slides.
  - Other than that, you can think of the JVM as a physical processor (for now).
    - JVM internals will be discussed with our lecture on code generation.
  - Java bytecode is the ``machine code'' of the JVM.



JVM Stack and Instruction Set

- The JVM is a stack-based virtual machine
  - uses a stack instead of registers as intermediate storage for values of a computation
  - Arguments are pushed onto the stack
  - Operations take their operands from the stack and push the result back on the stack.



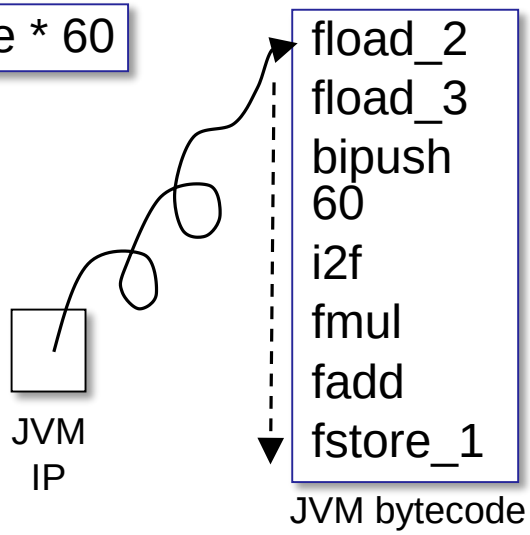
specifies the meaning of JVM bytecode instructions:

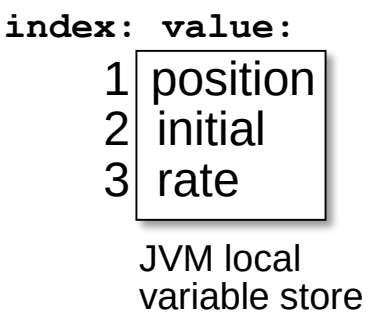
  - fload_<n> : push float value of local variable number <n> onto the stack
  - bipush <b> : push byte <b> onto the stack
  - i2f: convert the topmost stack-element from int to float
  - fmul: perform floating-point multiplication of the topmost stack elements, push the result onto the stack
  - fadd: compute the sum of the two topmost stack elements, push the result onto the stack.
  - fstore_<n>: pop the topmost stack element and store it in local variable number <n>.

JVM Bytecode Execution Example

position = initial + rate * 60

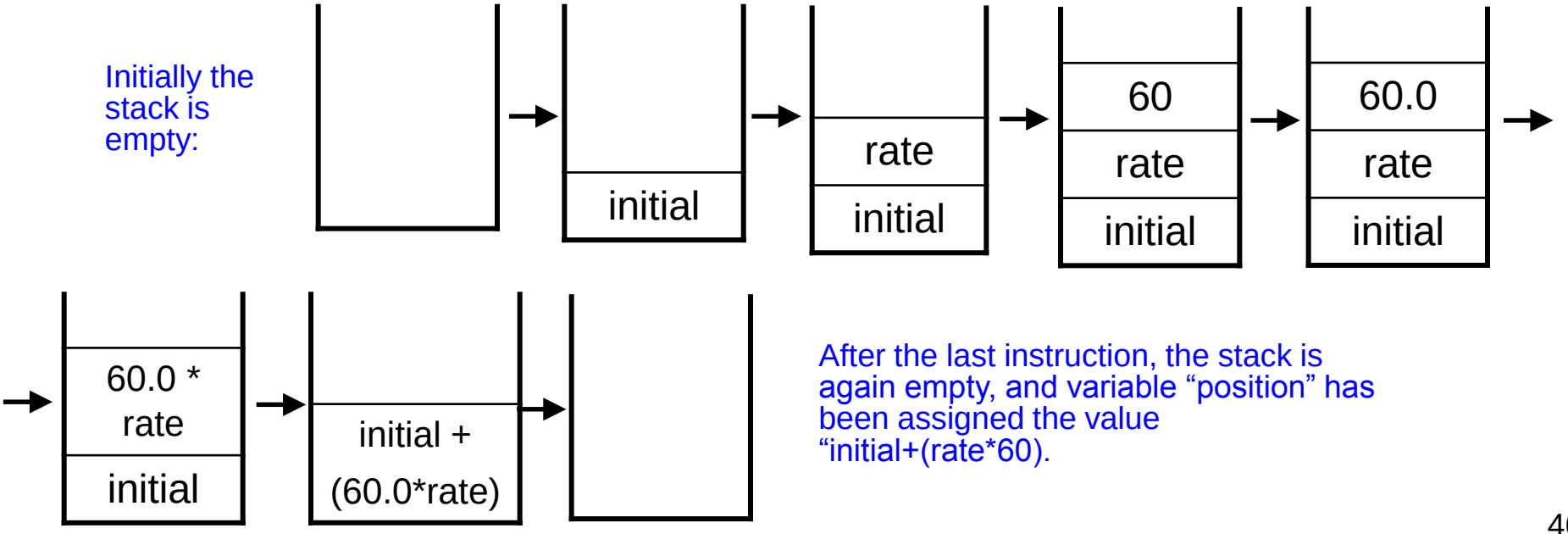
Java sourcecode





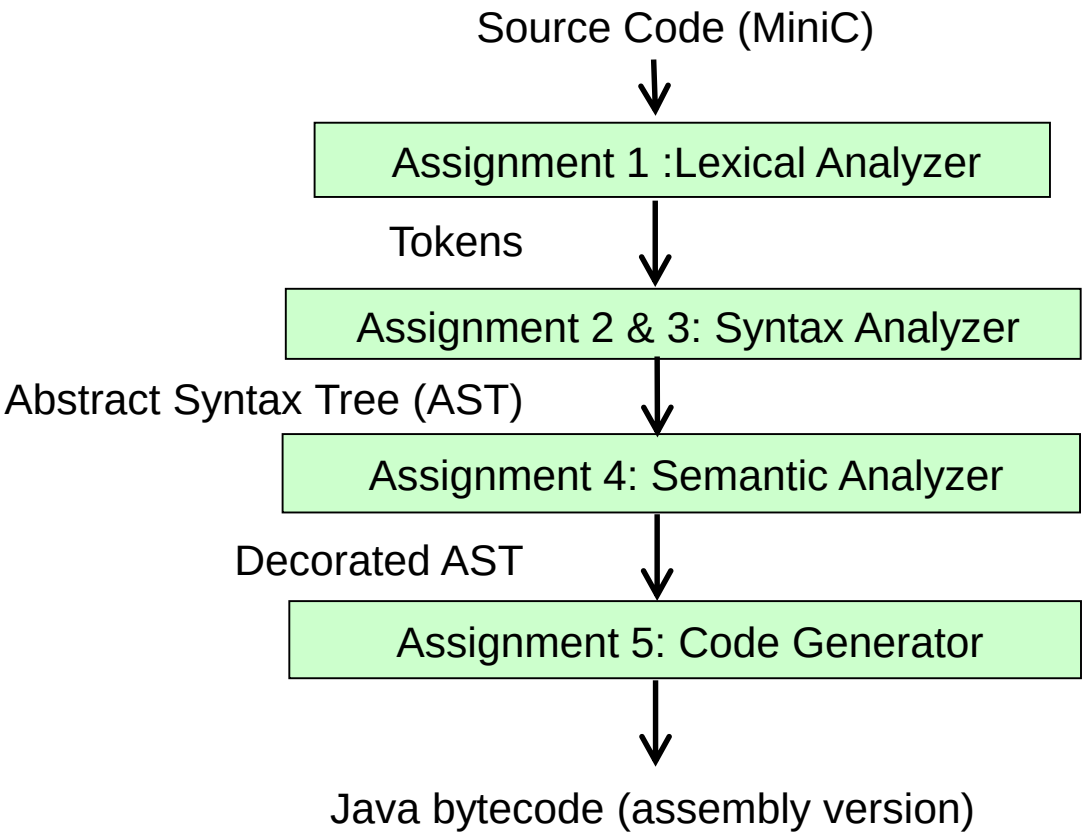
executing the bytecode sequence instruction by instruction results in the following JVM stack configurations:

Initially the stack is empty:



After the last instruction, the stack is again empty, and variable “position” has been assigned the value “initial+(rate*60).

Assignments



You can then execute your Java bytecode on the Java virtual machine…

MiniC

- comments: Java-like // and /* */
- Types:
  - primitive: void, int, float, bool, string
  - one-dimensional arrays: int[], float[], bool[]
- variables: global and local
- literals: integers, floats, bool, strings
- expressions: conditional, relational, arithmetic
- statements: if, for, while, assignment, return
- procedures and procedure calls

- The context-free grammar of MiniC will be provided on YSCEC.

Syllabus

- Lexical analysis
  - crafting a scanner by hand
  - regular expressions, NFAs, DFAs
  - scanner generators (Lex and JLex)
- Syntax analysis
  - Context-free grammars
  - recursive-descent parsing and LL(k)
  - bottom-up parsing and LR(k)
  - parser generators (Yacc, JavaCC)
  - abstract syntax trees
- Semantic analysis
  - symbol tables, variable bindings and scopes, type checking
- Run-time organization
  - data representation, storage allocation (static, stack, heap)
- Code generation
  - syntax-directed translation
  - Jasmin assembly language
  - Java Virtual Machine (JVM)

Lectures

- Introduce new material, often on the theoretical aspects of compilers.
  - REs and parsing, relation to scanner and parser generators
  - Lecture material practiced through homeworks, assessed in exams.

- Lectures provide guidelines for the assignments:
  - How to implement your compiler
  - Design issues
  - Explain code provided to you.

Outline

- Administrative issues ✓
- Subject overview ✓