Technische Universität Berlin

Compiler Design

Introduction to Compilers

Dr. Biagio Cosenza | TU Berlin | Wintersemester 2017-18

My Background

- PhD at the University of Salerno, Italy on Parallel Processing (2007-11)
  - HPC-Europa2 and HPC-Europa++ at HLRS Supercomputing, Stuttgart
  - ISCRA at CINECA Supercomputing, Bologna
  - DAAD Fellowship at VISUS, Universität Stuttgart
- Postdoctoral Researcher at the University of Innsbruck, Austria (2011-2015)
  - Insieme Compiler
  - Multi-disciplinary Research Platform (FWF DK-Plus Program)
- Senior Researcher at AES, TU Berlin, Germany (since April 2015)
  - Compilers and Software Optimization, High Performance Computing, Embedded Systems
  - International research projects

Lecture Organization

- Lecture
  - ➢Tuesday, 12:00 to 14:00 in room H 2032
- Lab
  - ➢Friday, 10:00 to 12:00 in rooms TEL 106 li and re (Telefunken-Hochhaus)
  - ➢Teaching assistant: Nikita Popov
  - ➢First lab is on October 27, 2017
- Final grade (Portfolio)
  - ➢50% final examination
  - ➢50% lab exercises
- Contact: (preferred) use the forum on ISIS
  - ➢Other students may have the same problem

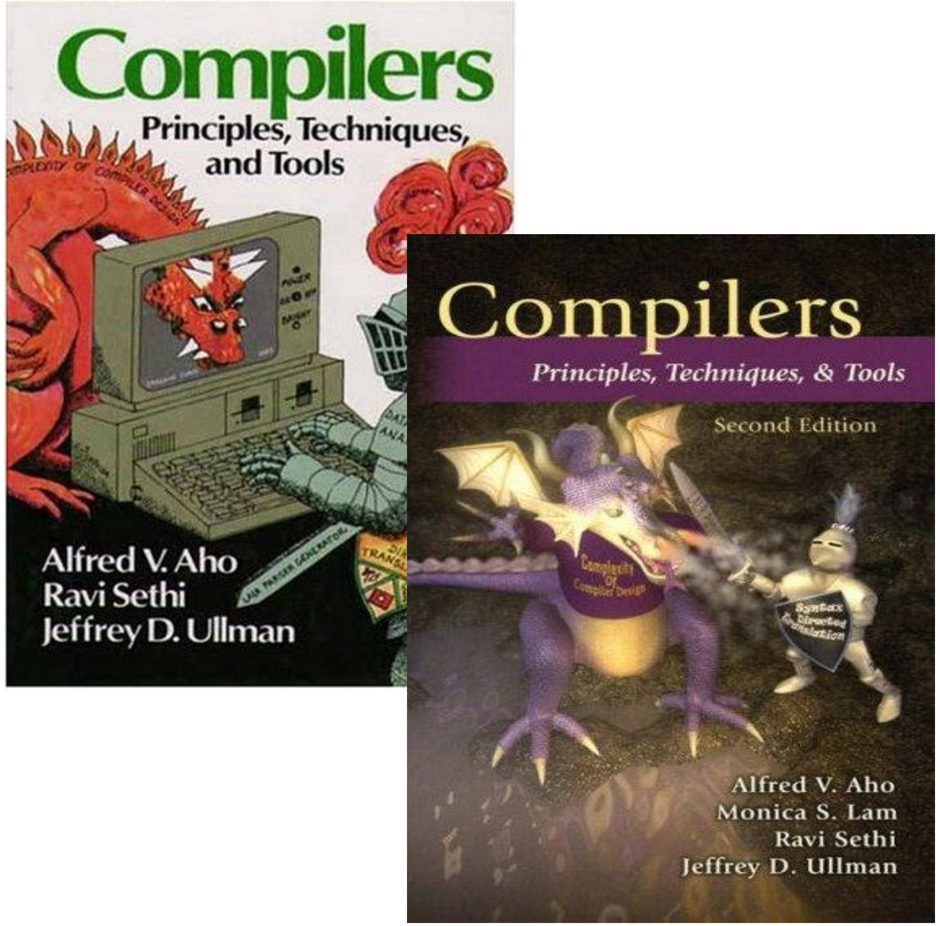
Tentative Schedule

Week		Lecture		Lab
1	17-Oct	Introduction		
2	24-Oct	Lexical Analysis	27-Oct	P1: Lex/Flex
3	31-Oct	Lexical Analysis	3-Nov	P1
4	7-Nov	Syntax Analysis	10-Nov	P1
5	14-Nov	Syntax Analysis	17-Nov	P1
6	21-Nov	Syntax Analysis	24-Nov	P2: Bison
7	28-Nov	Semantic Analysis	1-Dec	P2
8	5-Dec	Intermediate Representations	8-Dec	P2
9	12-Dec	Dataflow Analysis	15-Dec	P3: LLVM IR Analysis/Optimization
10	19-Dec	Dataflow Analysis	22-Dec	P3
	26-Dec	no lecture	29-Dec	no lab
	2-Jan	no lecture	5-Jan	no lab
11	9-Jan	SSA	12-Jan	P3
12	16-Jan	Runtime Env., Code Generation	19-Jan	P3
13	23-Jan	Registry Allocation	26-Jan	P4: LLVM IR Backend
14	30-Jan	Instruction Scheduling, Optimizations	2-Feb	P4
15	6-Feb	Compilation for Embedded Systems	9-Feb	P4
16	13-Feb	Exam exercise	16-Feb	P4

Updated calendar available on ISIS https://isis.tu-berlin.de/course/view.php?id=8583

Course Text Books

- The "Dragon Book", 2nd edition **[ALSU]**
  - Aho, Lam, Sethi, Ullman. "Compilers: Principles, Techniques and Tools", 2nd edition
  - international version has no dragon!
  - 1st edition by Aho, Sethi, Ullman **[AHO]**
  - Useful for the first part of the course
- Other books
  - Cooper & Torczon. "Engineering a Compiler"
  - Hunter et al. "The essence of Compilers" (Prentice-Hall)
  - Grune et al. "Modern Compiler Design" (Wiley)
- Additional materials: notes, papers, technical reports



Students' Information

- Are you enrolled on ISIS (isis.tu-berlin.de)?
  - ➢If not, do it!
- How many EIT students?
- How many Erasmus students?
- How many international exchange students?
- How many are enrolled to another German University (Humboldt U., Freie U., …)?
- How many of you cannot be listed on QISPOS?

Important: Plagiarism NOT Allowed

- Exercises must contain original solutions only
- TU Berlin has strong rules against plagiarism
  - Fak. IV rules (German) https://www.eecs.tu-berlin.de/fileadmin/f4/fkIVdokumente/plagiate.pdf
  - What is plagiarism (English) https://www.ox.ac.uk/students/academic/guidance/skills/plagiarism?wssl=1
  - Important: if you copy an exercise, you fail the whole course
- Also, you cannot
  - Upload your solution on the internet or share it on social networks
    - E.g., on GitHub

Syllabus

- Introduction
- Lexical analysis
  - Flex
- Syntax analysis
  - Bison
- Semantic analysis
- Intermediate representation
- Static Single Assignment (SSA)
- Dataflow analysis
  - LLVM analysis and optimization
- Code generation & runtime
- Register allocation
- Instruction scheduling
- Optimizations
- Compilation for embedded systems
  - LLVM code generation

Learning Outcomes

- A student successfully completing this course should be able to
  - understand the principles governing all phases of the compilation process
  - understand the role of each of the basic components of a standard compiler
  - show awareness of the problems of and methods and techniques applied to each phase of the compilation process
  - apply standard techniques to solve basic problems that arise in compiler construction
  - understating basic compiler optimizations and its implementation on real compiler (e.g., LLVM)
  - be a better programmer

What is a Compiler?

He always liked to open the Annual Programmer's Conference with a joke in binary.

What is a Compiler?

“a computer program that translates a program written in a high-level language into another language, usually machine language”

- source: http://dictionary.reference.com

- German translations (from http://dict.leo.org)
  - Der Kompilierer
  - Der Übersetzer
    - Very interesting translation
    - Literally: translator

More Generally

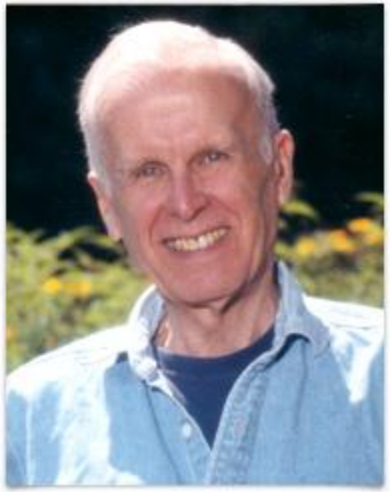
Human readable notation → Program → Something else

Historical Notes: Programming Languages

- Machine Languages
- 2nd generation: Assembly Languages – early 1950s
- 3rd generation: High-Level Languages – later 1950s
  - Fortran, ALGOL, COBOL
  - More recently C, C++, C#, Java, BASIC and Pascal,
- 4th generation higher level languages – 1970-1990
  - Also including domain specific languages (DSL)
  - SQL, Postscript, Python, Ruby, and Perl
- 5th generation languages: constraint-based, logic programming languages and some declarative languages
  - Prolog, OPS5, Mercury

Historical Notes: Fortran

- Fortran (Formula Translating System)
  - first, widely used high-level programming language
  - by John Warner Backus, IBM
  - in 1954 Backus assembled a team to define and develop Fortran for the IBM 704 computer
  - Backus also contributed to ALGOL58 and 60 and the BNF (Backus-Naur Form)
  - Turing Award, 1977



How many do you know?

javacc
Cetus
Polly/LLVM
NESL (CMU)

gcc/g++

Rose
Patus
Erlang
IBM xlc/xlcpp
Charm (Illinois)

PGI
Pluto
OpenMP compilers
ARM armcc
Sequoia (Stanford)

ispc
Insieme
HPCS Chapel (Cray)
Intel icc/icpc
HMPP

Borealis (Brown)
LLVM/Clang
HPCS Fortress (Sun)
NVIDIA cudacc

OpenCL compilers (Intel, AMD, NVIDIA, …)
Cilk (MIT/Intel)
OpenACC

Definitions

- What is a compiler?

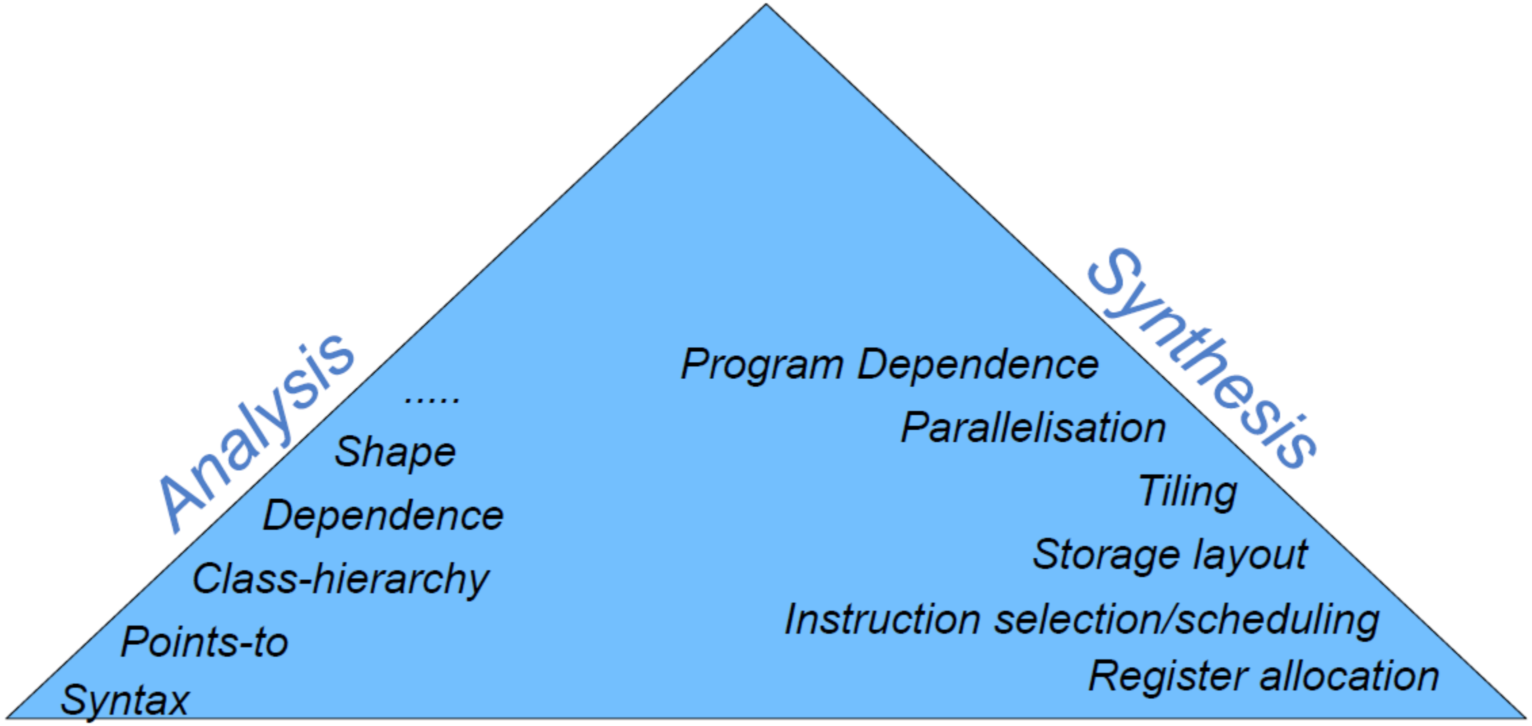
  a program that accepts as input a program text in a certain language and produces as output a program text in another language, while preserving the meaning of that text [Grune et al., 2000]

  a program that reads a program written in one language (source language) and translates it into an equivalent program in another language (target language) [AHO]

  - key: ability to extract properties of a source program (analysis) and transform it to construct a target program (synthesis)

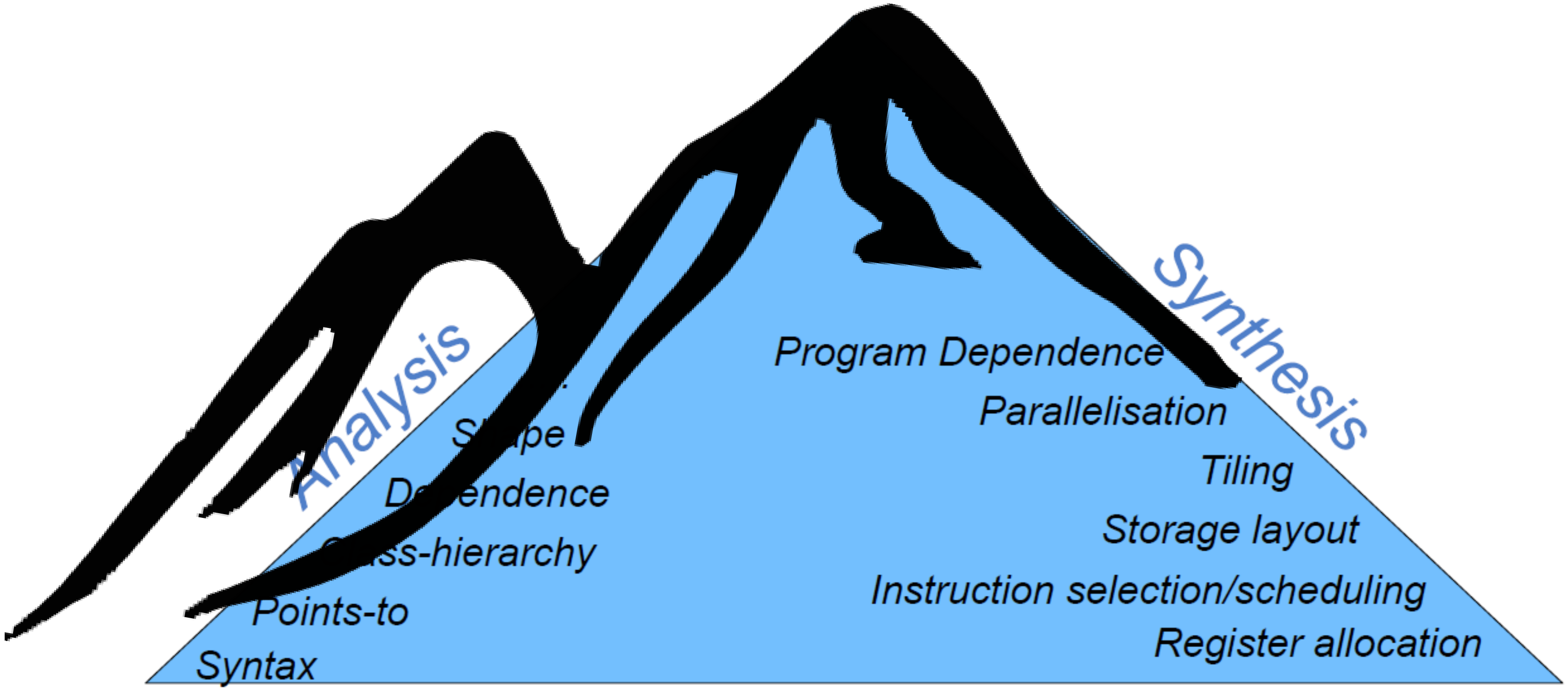
- What is an interpreter?
  - a program that reads a source program and produces the results of executing this source
  - an interpreter directly executes, i.e. performs, instructions written in a programming language, without previously compiling them into a machine language program

Analysis & Synthesis



Courtesy of Paul Kelly, Imperial College London

Analysis & Synthesis



Courtesy of Paul Kelly, Imperial College London

Questions

- Is it compiled or interpreted?
  - C
  - Lisp
  - Java
  - PHP
  - Latex
  - Ghostview
  - source-to-source C compiler
  - High Performance Fortran (HPF)

Example 1: Source-to-source Compilers

- Also called transpiler or transcompiler
- Typically C-to-C
- Code transformation at source level
  - ➢ examples: automatic parallelization, data layout transformations, …
- High-level intermediate representation
  - ➢ we will see this in the next lectures (Intermediate Representations)
- Examples
  - ➢ Rose, Insieme, Pluto, Cetus, …

Example 2: Java JIT Compiler

- Java compiler
  - the output is a class file (.class)
  - `javac` (Oracle), `gcj` (GNU Compiler for Java), `ECJ` (Eclipse for Java)
  - platform-neutral Java bytecode
  - there are also compilers that emit optimized native machine code for a particular hardware/operating system combination
- Most Java-to-bytecode compilers do little optimization, leaving this to the JRE (Java Runtime) at runtime
- Just-in-time (JIT) compilation
  - the Java virtual machine (JVM) loads the class files and either interprets the bytecode or just-in-time compiles it to machine code and then possibly optimizes it using dynamic compilation.
  - Interaction between JVM and Java compilers specified in JSR 199

Example 3: Domain-Specific Languages

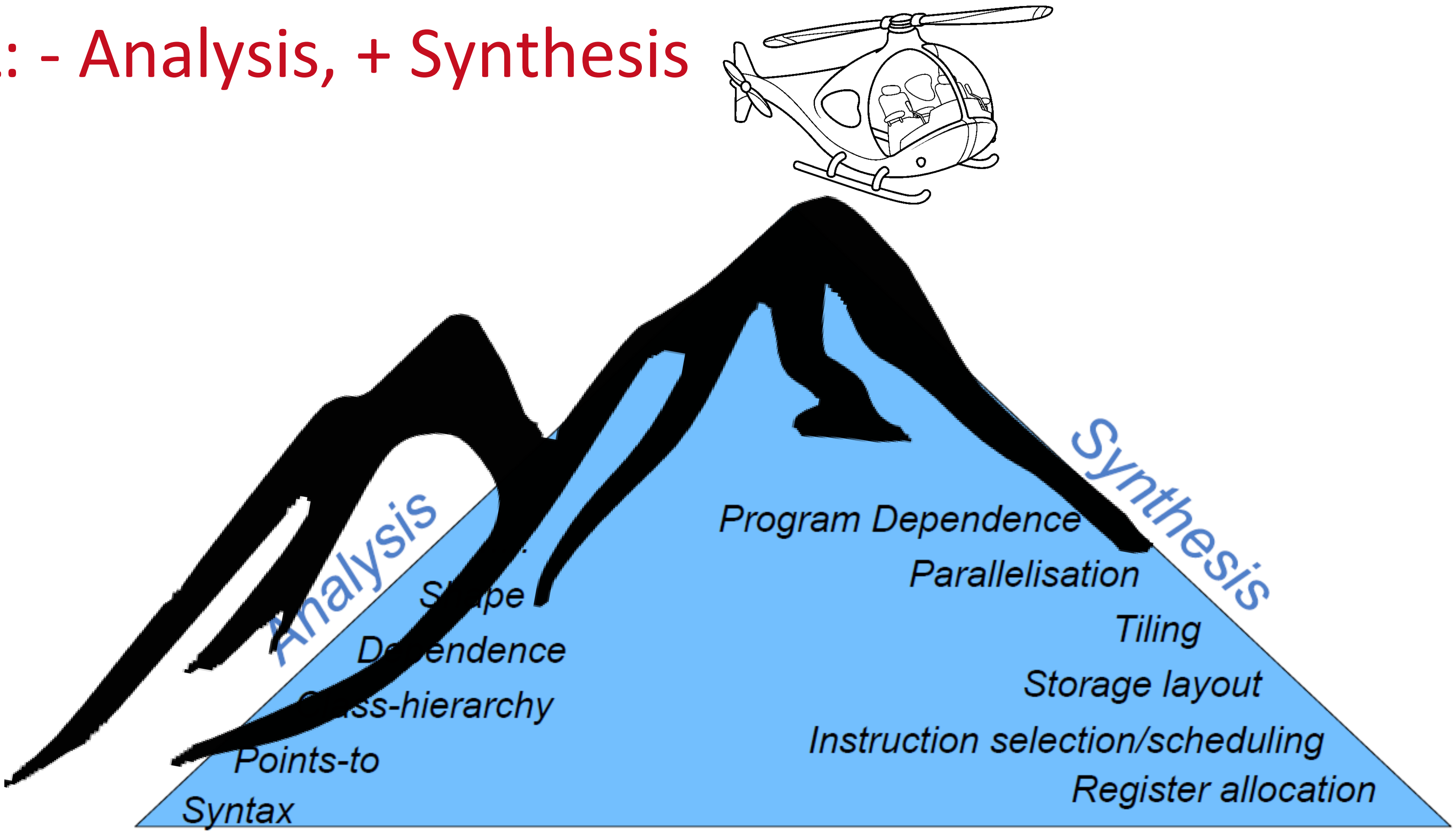
- Input is a domain-specific language (DSL)
  - ➢ a language with domain-specific construct and constraints
- Assumptions (and restriction) on the input
  - ➢ analysis simpler
  - ➢ optimization is relatively easier
    - Domain-specific optimization
- Examples
  - ➢ OpenGL Shading Language
  - ➢ Halide for image processing
  - ➢ SQL for database

Example: simple GLSL fragment shader

```
varying vec3 N;
varying vec3 v;

void main(void){
  vec3 L = normalize(gl_LightSource[0].position.xyz-v);
  vec4 Idiff = gl_FrontLightProduct[0].diffuse *
max(dot(N,L), 0.0);
  Idiff = clamp(Idiff, 0.0, 1.0);
  gl_FragColor = Idiff;
}
```

DSL: - Analysis, + Synthesis



Courtesy of Paul Kelly, Imperial College London

Example 4: Parallelizing Compilers

- Input is sequential code
- Output is parallel code, i.e., expose some kind of parallelism
- Typically, parallelism is extracted from loops
  - advanced analysis, e.g., using the polyhedral model
  - `#pragma` notations to help the compiler job
- Sometime parallelizing compilers enhance parallelization
  - E.g., from shared memory parallel code to distributed or heterogeneous systems
- Form of parallelism
  - automatic vectorization (SIMD instructions), e.g., by `gcc`, `llvm` and `icc`
  - multi-threading (pthread), e.g., by Rose, Pluto, Insieme, LLVM-Polly
  - distributed memory (typically MPI)

An Advanced Example: Automatic Parallelization

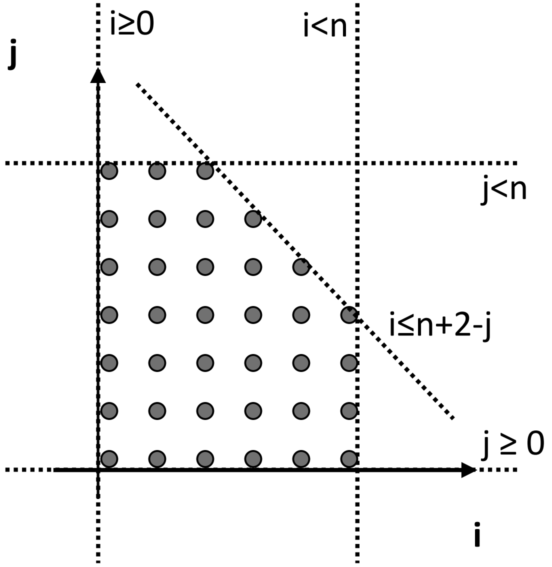
- How can you make this code parallel?

```
for(int i=0; i<n; i++)
  for(int j=0; j<n; j++)
    if(i <= n+2-j)
      b[j] = b[j] + a[i];
```

- Are iterations independent?
- Suppose you have four processors
  - ➢ How would you represent (and distribute) loop iterations between processors?

Automatic Parallelization & Polyhedral Model

```
for(int i=0; i<n; i++)
  for(int j=0; j<n; j++)
    if(i <= n+2-j)
(s)       b[j] = b[j] + a[i];
```



Polyhedron for n=6

$$\begin{array}{cccc} i & j & n & \text{constant} \end{array}$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 0 & 1 & -1 \\ 0 & 1 & 0 & 0 \\ 0 & -1 & 1 & -1 \\ -1 & -1 & -1 & -2 \end{bmatrix} \begin{pmatrix} i \\ j \\ n \\ 1 \end{pmatrix} \geq \vec{0}$$

Iteration domain with homogenous coord.

$$\begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ -1 & -1 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ n-1 \\ 0 \\ n-1 \\ -n-2 \end{pmatrix} \geq \vec{0}$$

Iteration domain of S

Qualities of a Good Compiler

- What qualities would you like in a compiler?
  - generates correct code (first and foremost!)
  - generates fast code
  - conforms to the specifications of the input language
  - copes with essentially arbitrary input size, variables, etc.
  - compilation time (linearly) proportional to size of source
  - good diagnostics
  - consistent optimizations
  - works well with the debugger

Principles of Compilation

- The compiler must
  - preserve the meaning of the program being compiled
  - “improve” the source code in some way
- Other issues (depending on the setting)
  - speed (of compiled code)
  - space (size of compiled code), energy consumption
  - feedback, latency (information provided to the user)
  - debugging (transformations obscure the relationship source code vs target)
  - compilation time efficiency (fast or slow compiler?)

Uses of Compilers

- Simply translation of high-level program to object code
  - ➢ program translation: binary translation, hardware synthesis, …
- Optimizations
  - ➢ improve program performance, take into account hardware
  - ➢ automatic parallelization
- Performance instrumentation
  - ➢ example: `-pg` option of gcc
- Interpreters
  - ➢ Perl, bash, …
- Software productivity tools
  - ➢ debugging aids, e.g. purify
- Security
  - ➢ Java VM uses compiler analysis to prove "safety" of Java code.
- Web-browsers (Javascript and HTML), text formatters, just-in-time compilation for Java, power management, global distributed computing, …

Ability to extract properties of a source program (analysis) and transform it to construct a target program (synthesis)

Questions

- Difference between compiler and interpreter
- What is a source-to-source compiler?
- What is a parallelizing compiler?
- What is a Domain Specific Language?

Programming Languages

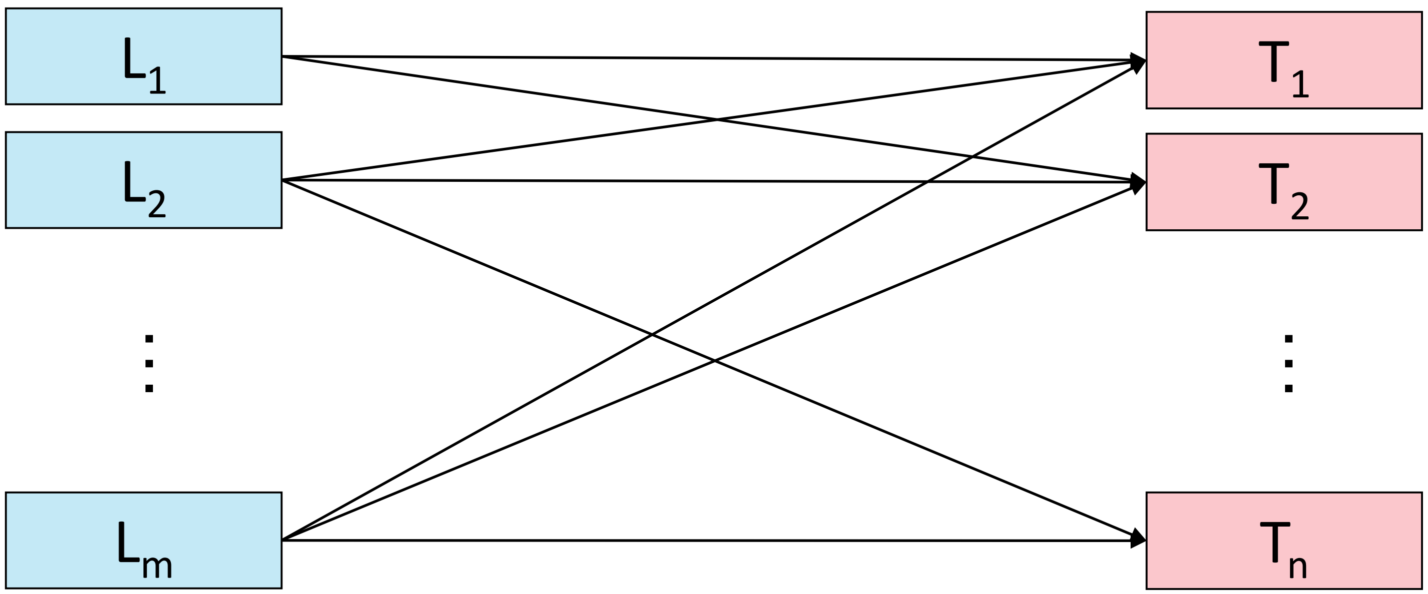
- L_1
- L_2
- $\vdots$
- L_m

Target Architecture

- T_1
- T_2
- $\vdots$
- T_n

Programming Languages

Target Architecture



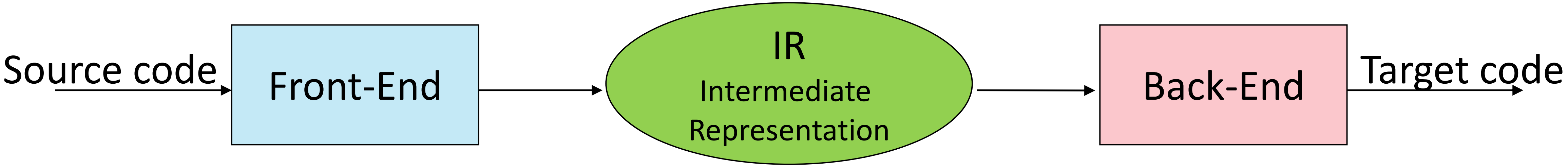
General Structure of a Compiler

Front-end performs the analysis of the source language

- recognizes legal and illegal programs and reports errors
- understands the input program and collects its semantics in an Intermediate Representation (IR)
- produces IR and shapes the code for the back-end

Back-end does the target language synthesis

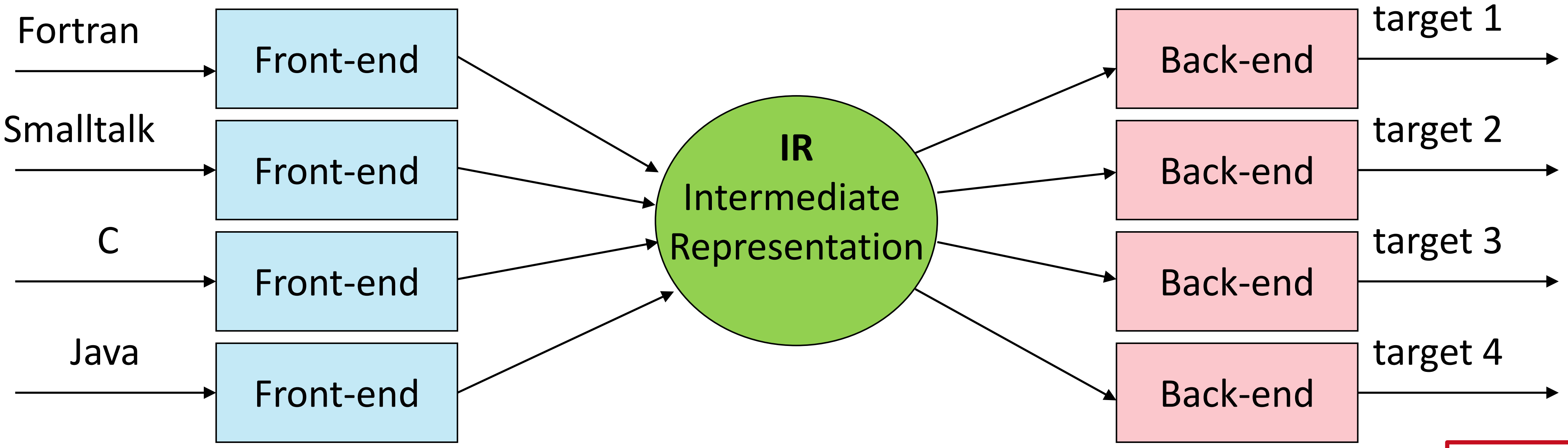
- chooses instructions to implement each IR operation
- translates IR into target code
- needs to conform with system interfaces
- automation has been less successful



Questions

- What is the implication of this separation (front-end: analysis; back-end: synthesis) in building a compiler for a new language?
- And for a new target architecture?

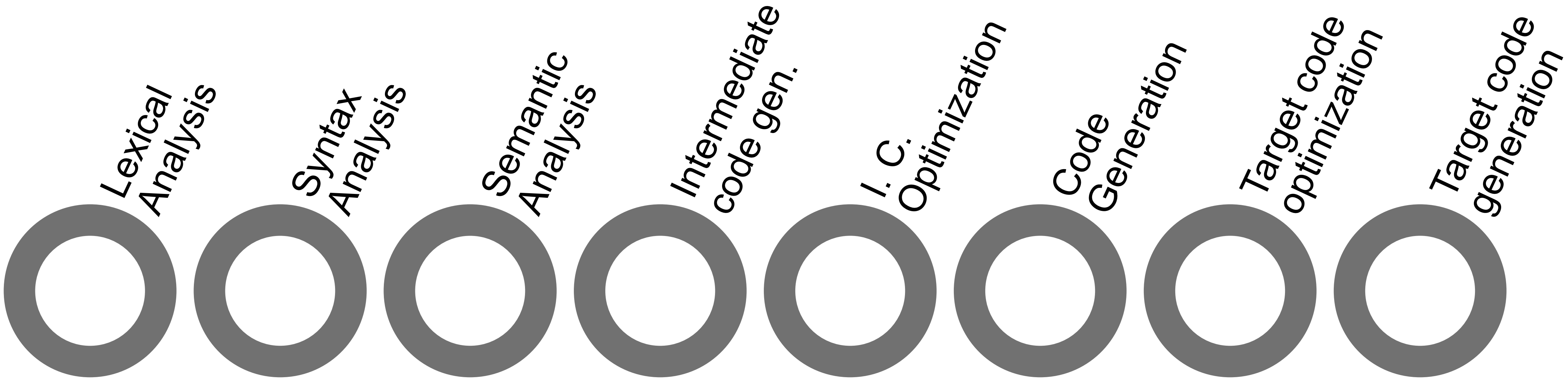
Answer: `mxn` compilers with `m+n` components!



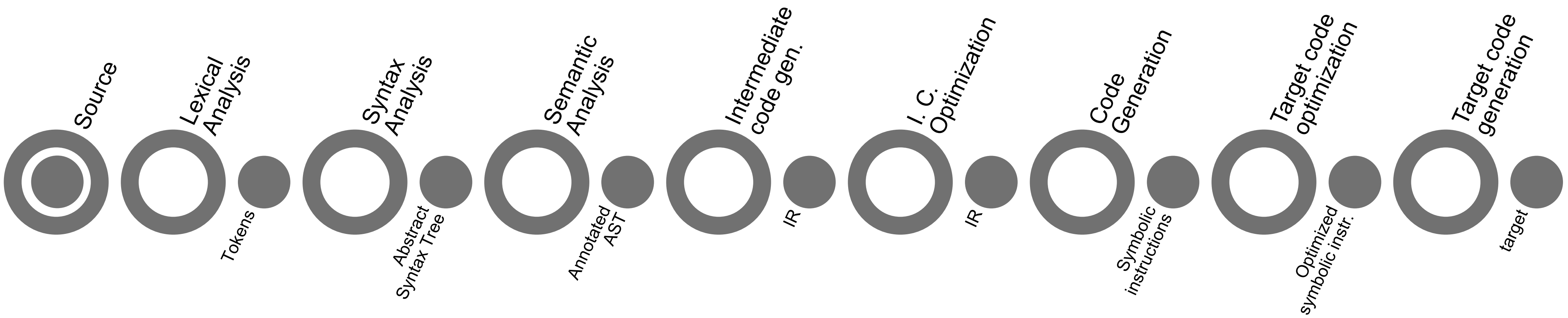
- All language-specific knowledge must be encoded in the front-end
- All target-specific knowledge must be encoded in the back-end

This strict separation is not free of charge!

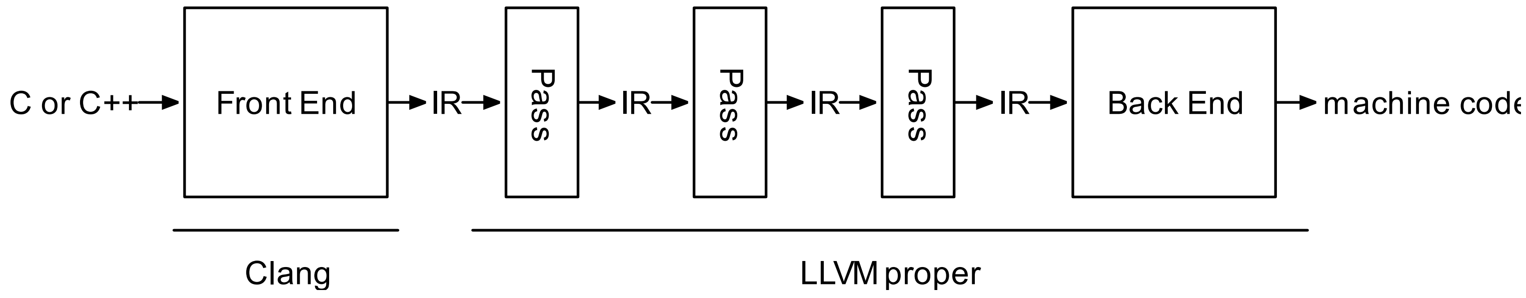
General Compiler Structure



General Compiler Structure



Example:
the LLVM compiler



Lexical Analysis

- Reads characters in the source program and groups them into words (basic unit of syntax)
- Produces words and recognizes what sort they are
- The output is called token and is a pair of the form <type,lexeme> or <token_class, attribute>
  - E.g.: **a=b+c** becomes <id,**a**> <=,> <id,**b**> <+,> <id,**c**>
- Needs to record each id attribute: keep a symbol table
  - Lexical analysis eliminates white space, etc
- Speed is important - use a specialized tool
  - Flex: a tool for generating scanners: programs which recognize lexical patterns in text

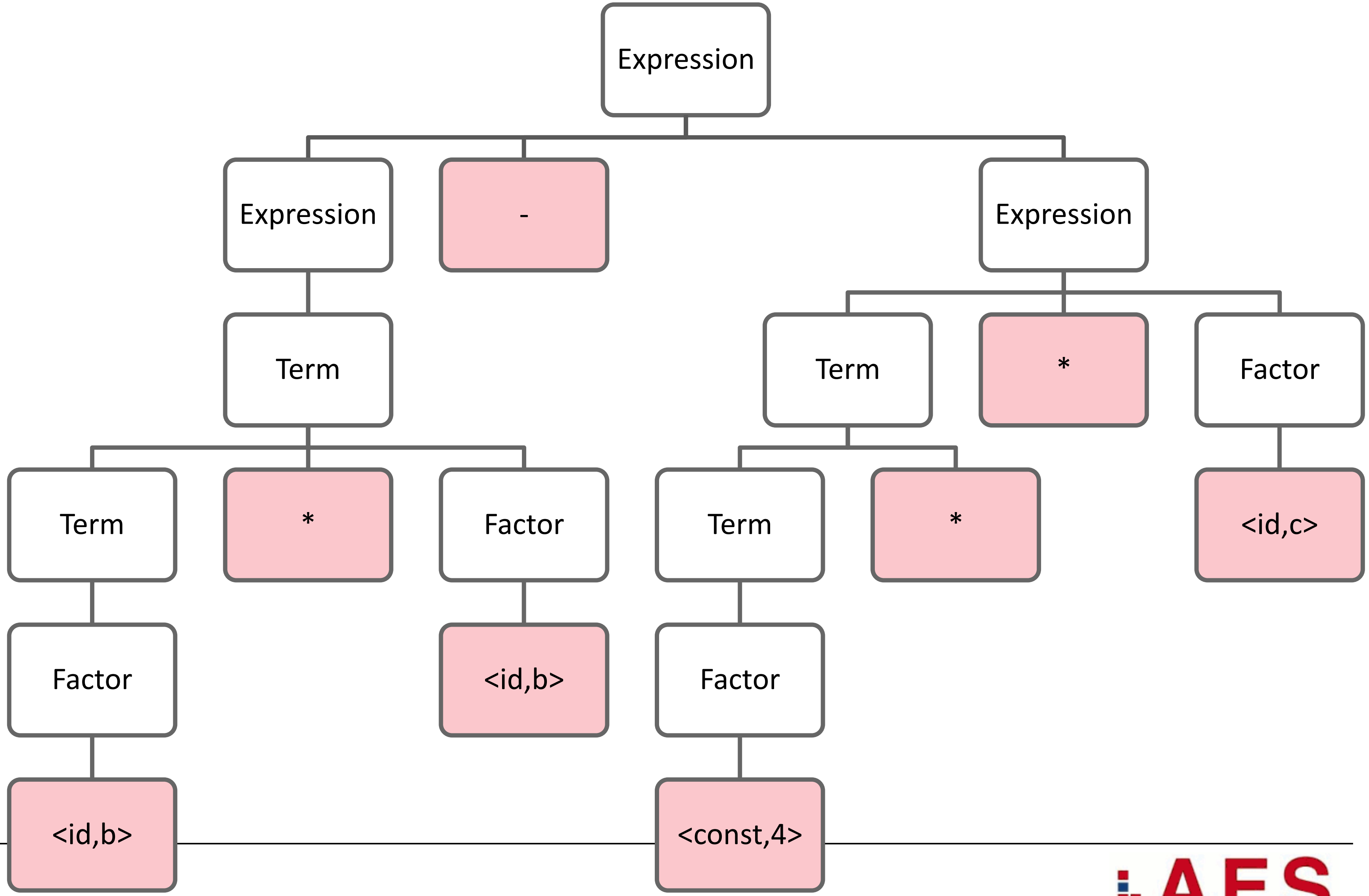
Syntax Analysis (Parsing)

- Imposes a hierarchical structure on the token stream
- This hierarchical structure is usually expressed by recursive rules
- Context-free grammars formalise these recursive rules and guide syntax analysis
- Example
  - A grammar defining simple algebraic expressions

```
expression  -> expression '+' term | expression '-' term | term
term -> term '*' factor | term '/' factor | factor
factor -> identifier | constant | '(' expression ')'
```

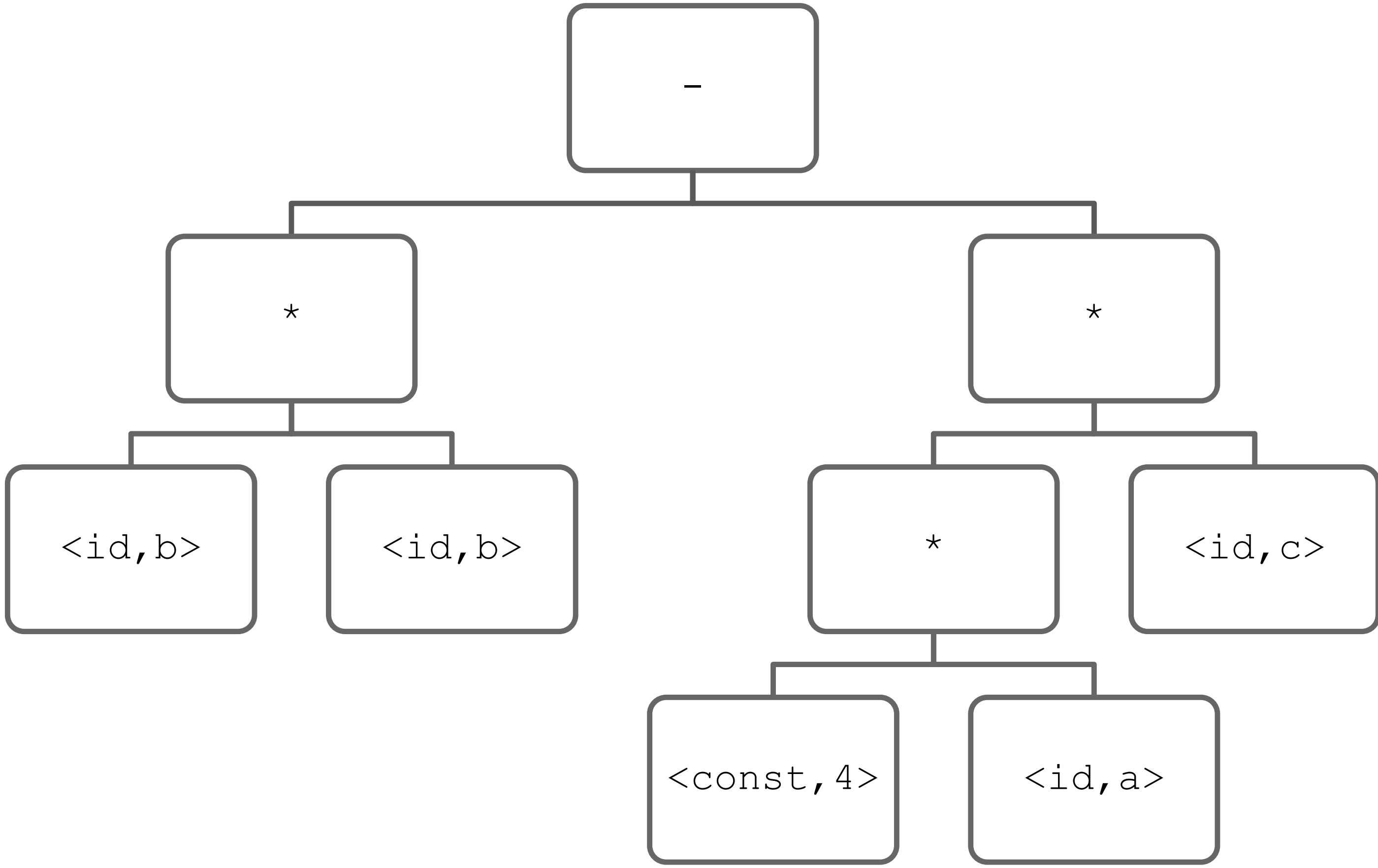
Parsing

- Parse tree for
  `b*b-4*a*c`



Abstract Syntax Tree (AST)

- AST for `b*b-4*a*c`
- An Abstract Syntax Tree (**AST**) is a more useful data structure for internal representation
- It is a compressed version of the parse tree (summary of grammatical structure without details about its derivation)
- ASTs are a form of IR



Semantic Analysis (Context Handling)

- Collects context (semantic) information, checks for semantic errors, and annotates nodes of the tree with the results
- Examples
  - ➢ type checking: report error if an operator is applied to an incompatible operand
  - ➢ check flow-of-controls
  - ➢ uniqueness or name-related checks

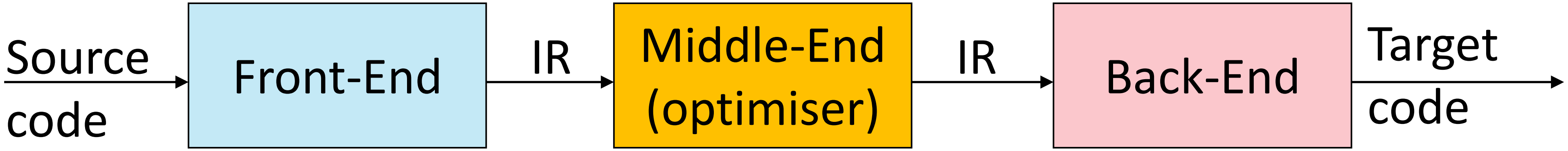
Intermediate Code Generation

- Translate language-specific constructs in the AST into more general constructs
- A criterion for the level of “generality”: it should be straightforward to generate the target code from the intermediate representation chosen
- Example of a form of IR (3-address code)

```
tmp1 = 4
tmp2 = tmp1*a
tmp3 = tmp2*c
tmp4 = b*b
tmp5 = tmp4-tmp3
```

Code Optimization

- The goal is to improve the intermediate code and, thus, the effectiveness of code generation and the performance of the target code
- Optimizations can range from trivial (e.g., constant folding) to highly sophisticated (e.g., in-lining)
  - Example: replace the first two statements in the example of the previous slide with: `tmp2=4*a`
- Modern compilers perform such a range of optimizations, that one could argue for:



Optimizations

- Example: Dead Code Elimination (**DCE**)

```
int global;
void f ()
{
  int i;
  i = 1;
  global = 1;
  global = 2;
  return;
  global = 3;
}
```

```
int global;
void f ()
{



  global = 2;
  return;

}
```

Code Generation

- Map the AST onto a linear list of target machine instructions in a symbolic form
  - instruction selection: a pattern matching problem
  - register allocation: each value should be in a register when it is used (but there is only a limited number): NP-Complete problem
  - instruction scheduling: take advantage of multiple functional units: NP-Complete problem
- Target, machine-specific properties may be used to optimize the code
- Finally, machine code and associated information required by the Operating System are generated

Questions

- List the phases of a compiler
- What is the front-end?
- What is the back-end?
- What is the IR?
- Why, in a compiler infrastructure, the front-end is typically separated from the back-end?

Summary

- Compiler vs interpreter
- Frontend, backend, IR
- Structure of a compiler and compilation phases
- Source-to-source, DSL

- Recommended reading:
  - [ALSU] 1.1, , 1.2, 1.3, 1.5
  - JSRs: Java Specification Requests - JSR 199: Java Compiler API
    - https://jcp.org/en/jsr/detail?id=199
  - J. W. Backus, *Can Programming be Liberated from the von Neumann Style?*
    - https://www.cs.cmu.edu/~crary/819-f09/Backus78.pdf
  - H. Massalin, *Superoptimizer: A look at the smallest program*
    - In ASPLOS II, pages 122-126, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.