Compilers

Radan Ganchev

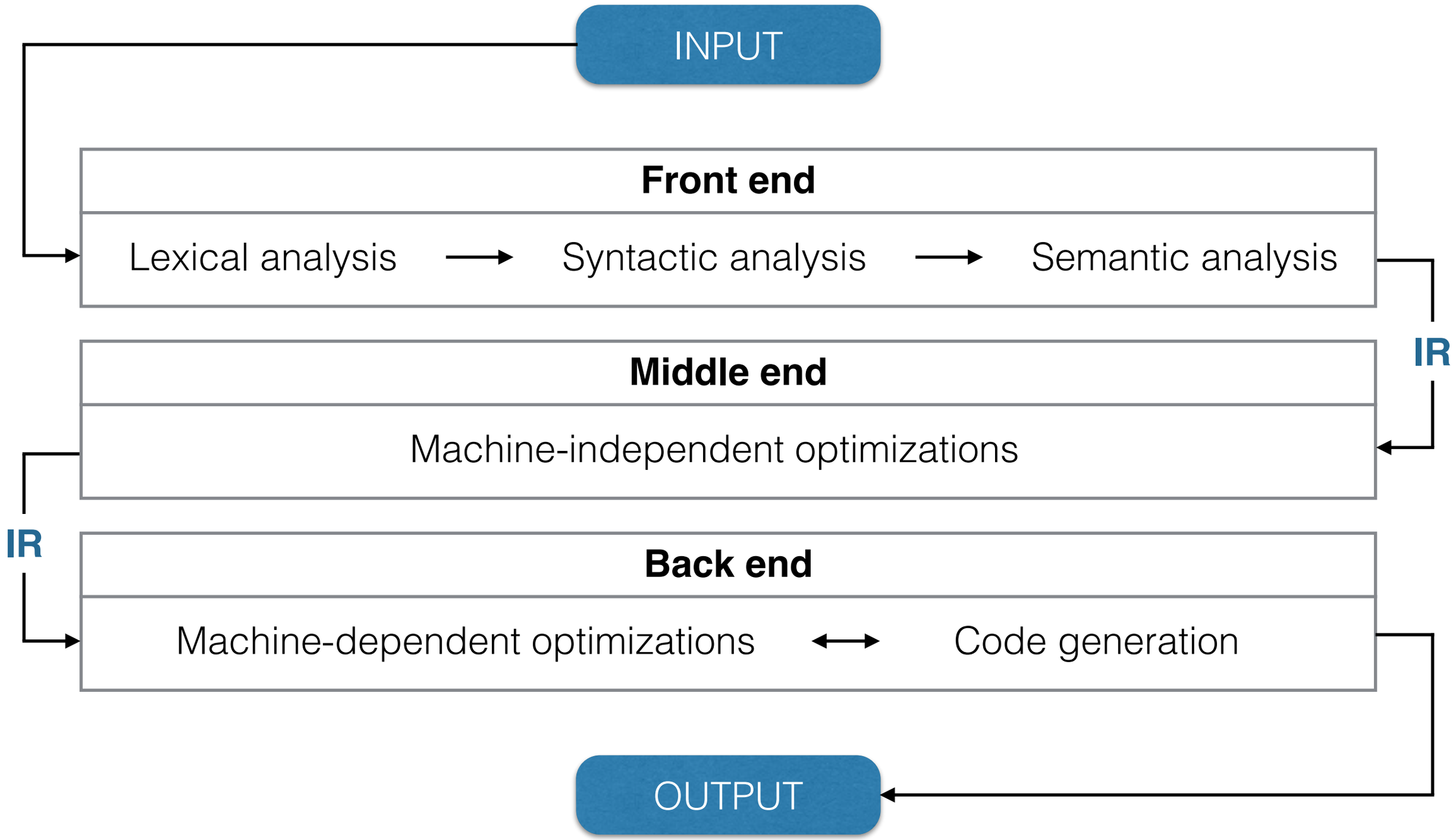
Compilers

- Purpose
- Structure
- How to write a compiler?
- Real life examples

Purpose of compilers

- Translation
  - high-level to low-level language translators
  - source-to-source compilers (transpilers)
  - language rewriters
  - AOT vs JIT
- Optimization

Structure of a compiler



Lexical analysis

- Tokens:
  <**token_class,** lexeme>

- Regular definition:
  token_class ⟶ *regular_expression*

Lexical analysis

Lexer

token classes:
if, then, else, =, -,
id, num, relop, ws

regular definitions:

id	⟶	[A-Za-z_]+
num	⟶	[0-9]+
relop	⟶	> | < | >= | <= | ==
ws	⟶	\s+

```
if x >= 0 then
   abs = x
else
   abs = -x
```

<if> <ws> <id, "x"> <ws>
<relop, ">="> <ws> <num, "0">
<ws> <then> <ws> <id, "abs">
<ws> <=> <ws> <id, "x"> <ws>
<else> <ws> <id, "abs"> <ws>
<=> <ws> <-> <id, "x">

Syntactic analysis

- Parsers construct the input's derivation in a formal grammar
- The Lexer's tokens are the Parser's terminals
- Derivations are described by *concrete syntax trees*
- Concrete syntax trees are usually transformed to *abstract syntax trees (AST)*

Syntactic analysis

Parser

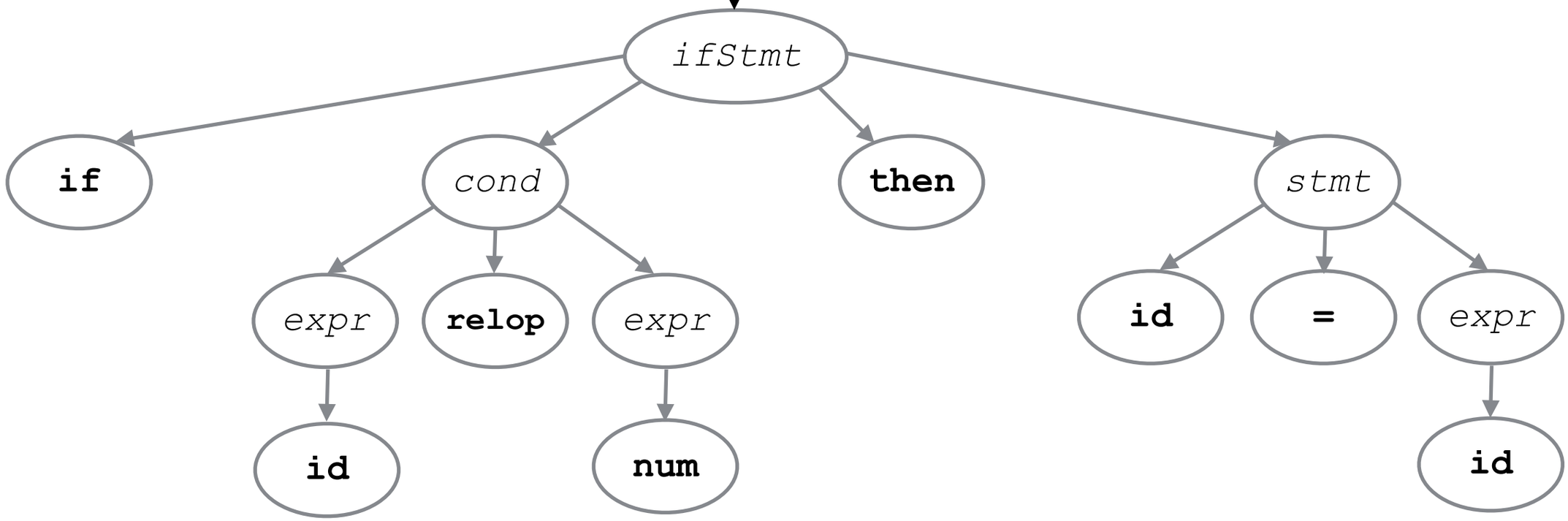
ifStmt ⟶ **if** *cond* **then** *stmt*

cond ⟶ *expr* **relop** *expr*

stmt ⟶ **id =** *expr*

expr ⟶ **id** | **num**

<if> <id, "x"> <relop, ">="> <num, "0"> <then> <id, "abs"> <=> <id, "x">



Syntactic analysis

Parser

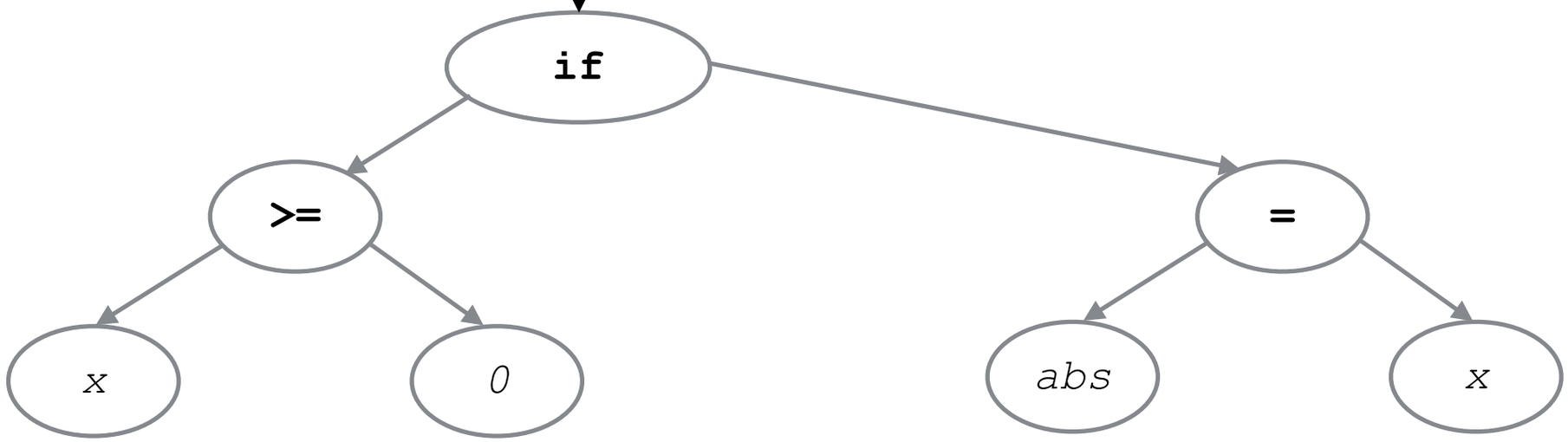
ifStmt ⟶ **if** *cond* **then** *stmt*

cond ⟶ *expr* **relop** *expr*

stmt ⟶ **id =** *expr*

expr ⟶ **id** | **num**

<if> <id, "x"> <relop, ">="> <num, "0"> <then> <id, "abs"> <=> <id, "x">



Semantic analysis

- Building/extending the symbol table(s)
- Type checking
- Definite assignment analysis

Optimizations

- Correctness and profitability
- Most optimizations run in two phases:
  - Analysis (data-flow, control-flow, etc.)
  - Transformation
- Optimizations usually require specific code representation:
  - Static Single Assignment (SSA)
  - Control-Flow Graph (CFG)

Machine-independent optimizations

- Redundancy elimination (CSE, GVN)
- Useless code elimination (DCE, DSE)
- Code motion (LICM, delayed allocation)
- Enabling optimizations (inlining, loop unrolling, loop peeling)

Machine-dependent optimizations

- Peephole optimizations
- Register allocation
- Instruction scheduling
- Trampolines

CFG Demo

How to write a compiler?

1. Learn some theory on lexical and syntactic analysis

   - The Dragon Book
   - Prof. Alex Aiken's Compilers course @ Coursera

2. Define a grammar

3. Use `lex` (`flex`) and `yacc` (`bison`) to generate a parser

4. Improvise!

Examples

- Jison
- CoffeeScript
- js-sequence-diagrams

Questions?