Compilers

Radan Ganchev

Compilers

Compilers

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

Purpose of compilers

Purpose of compilers

- Translation

Purpose of compilers

- Translation
    - high-level to low-level language translators

Purpose of compilers

- Translation
    - high-level to low-level language translators
    - source-to-source compilers (transpilers)

Purpose of compilers

- Translation
  - high-level to low-level language translators
  - source-to-source compilers (transpilers)
  - language rewriters

Purpose of compilers

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

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

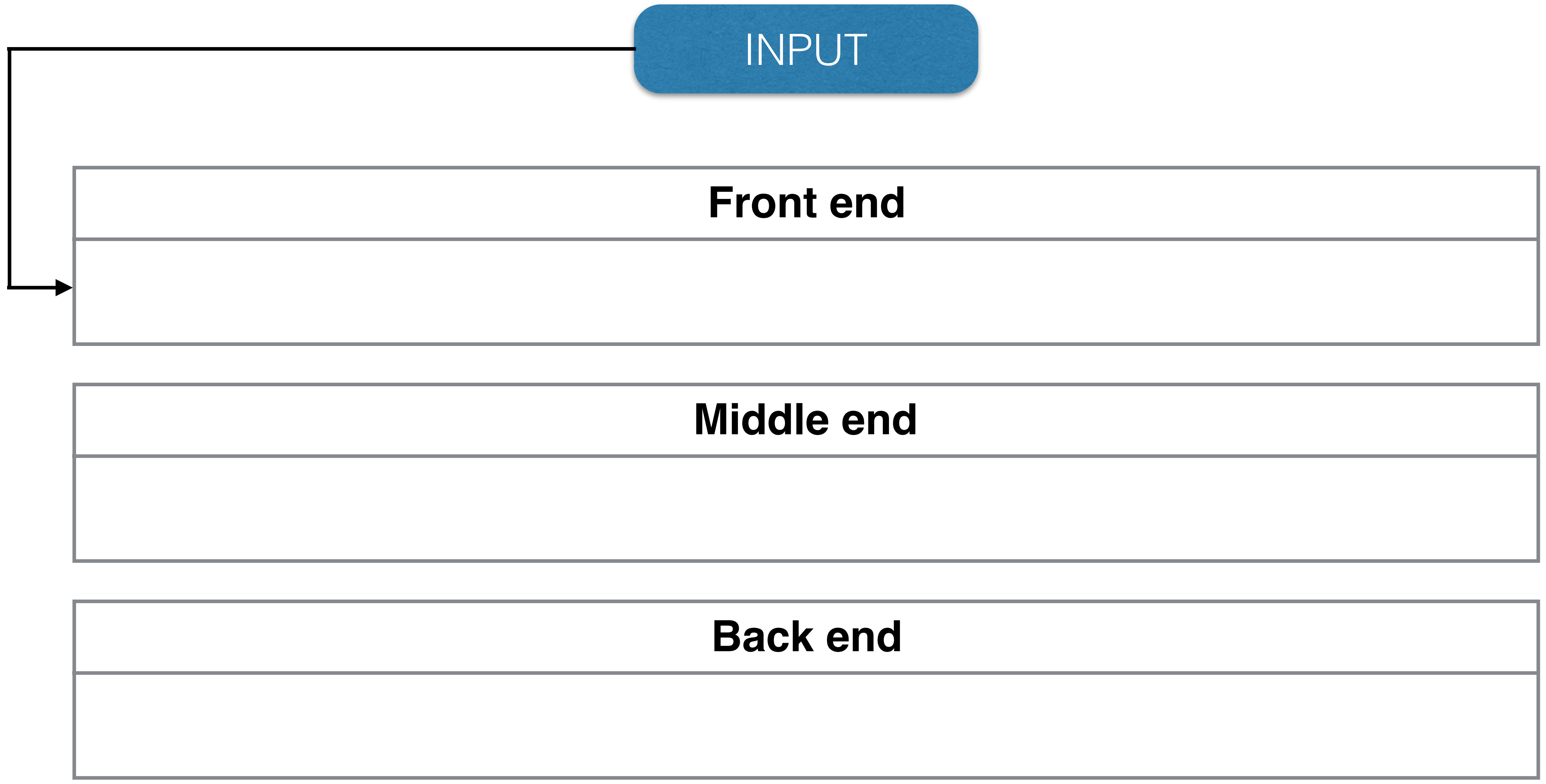
Structure of a compiler

Front end

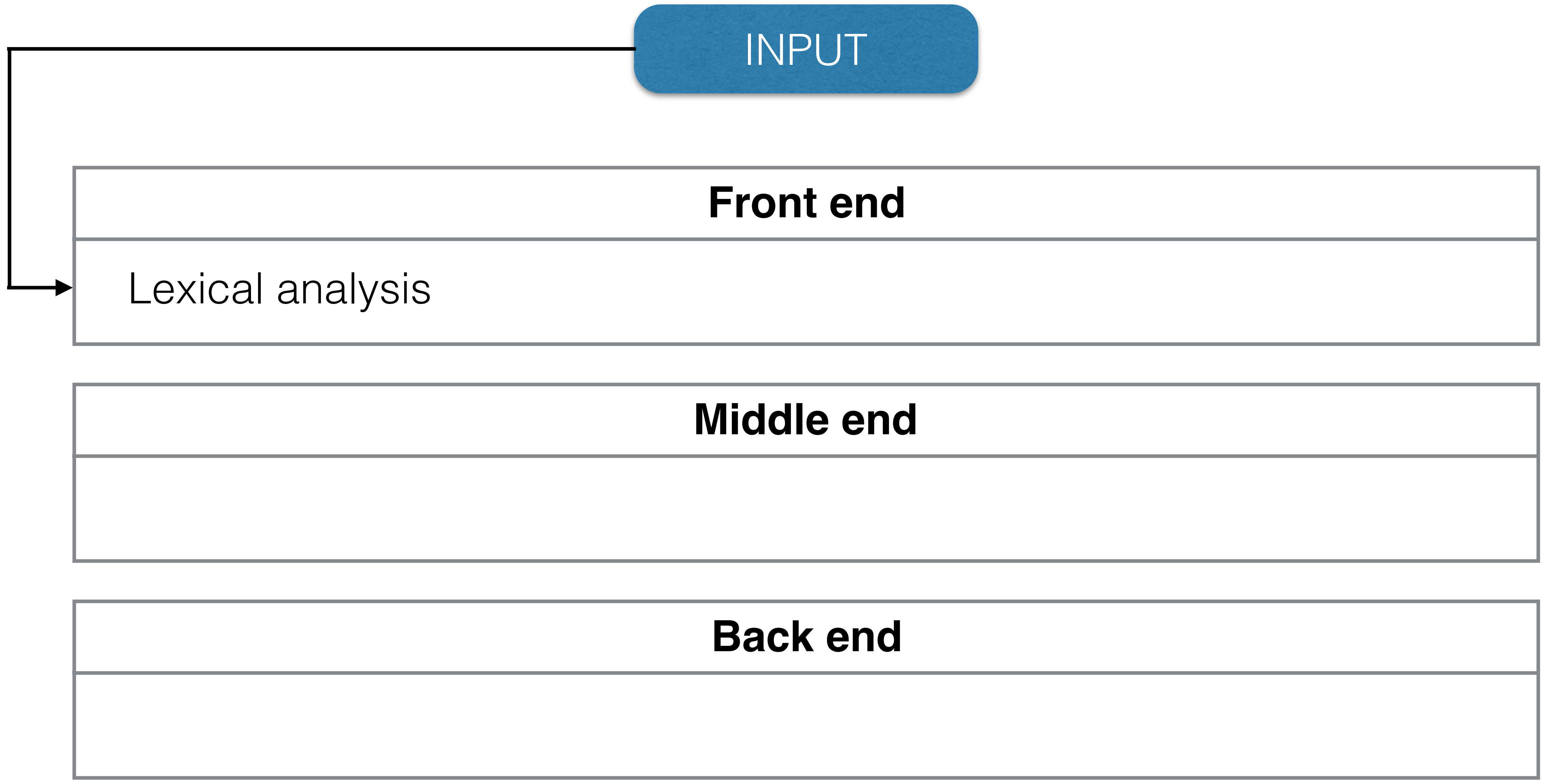
Middle end

Back end

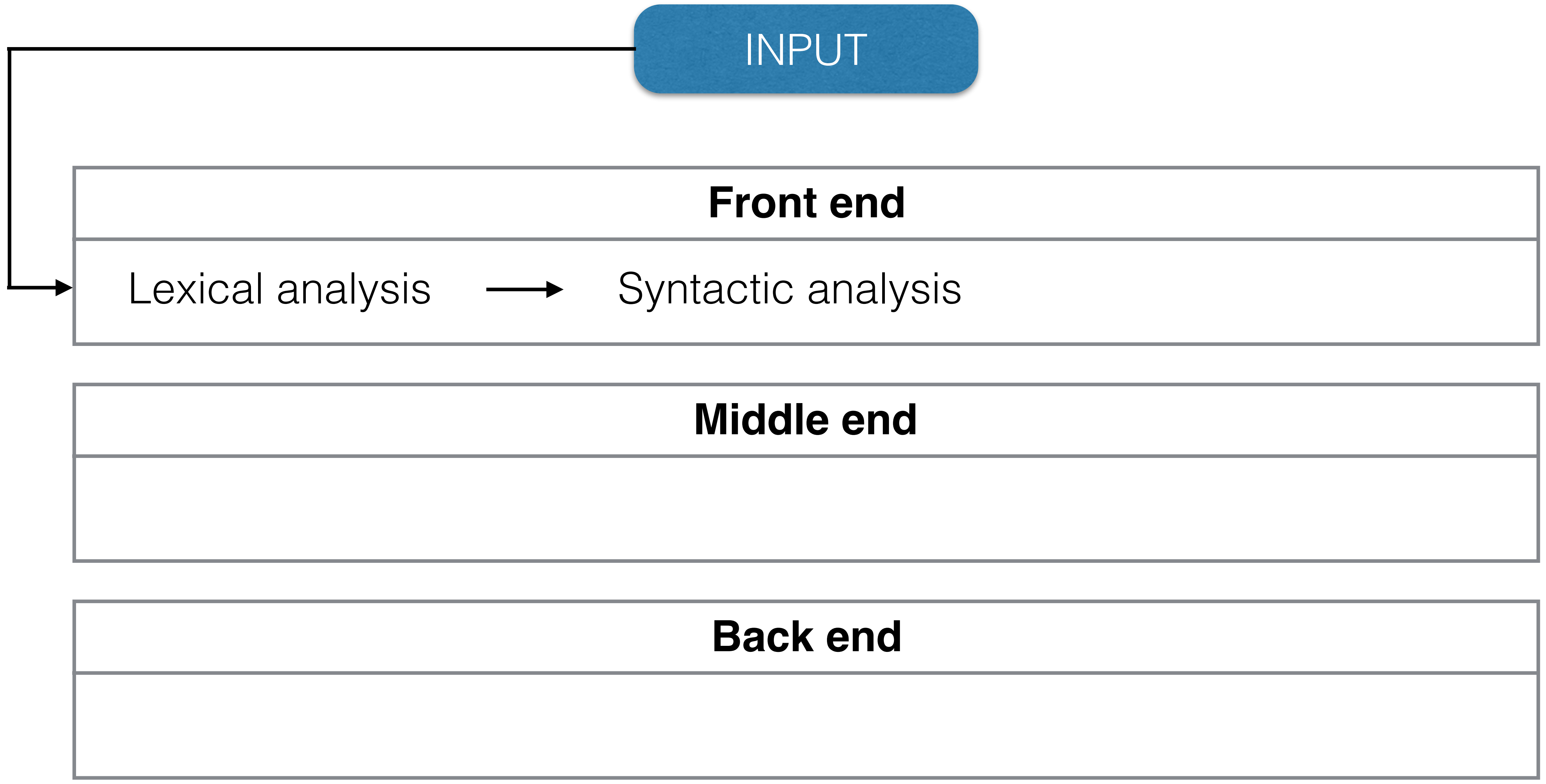
Structure of a compiler



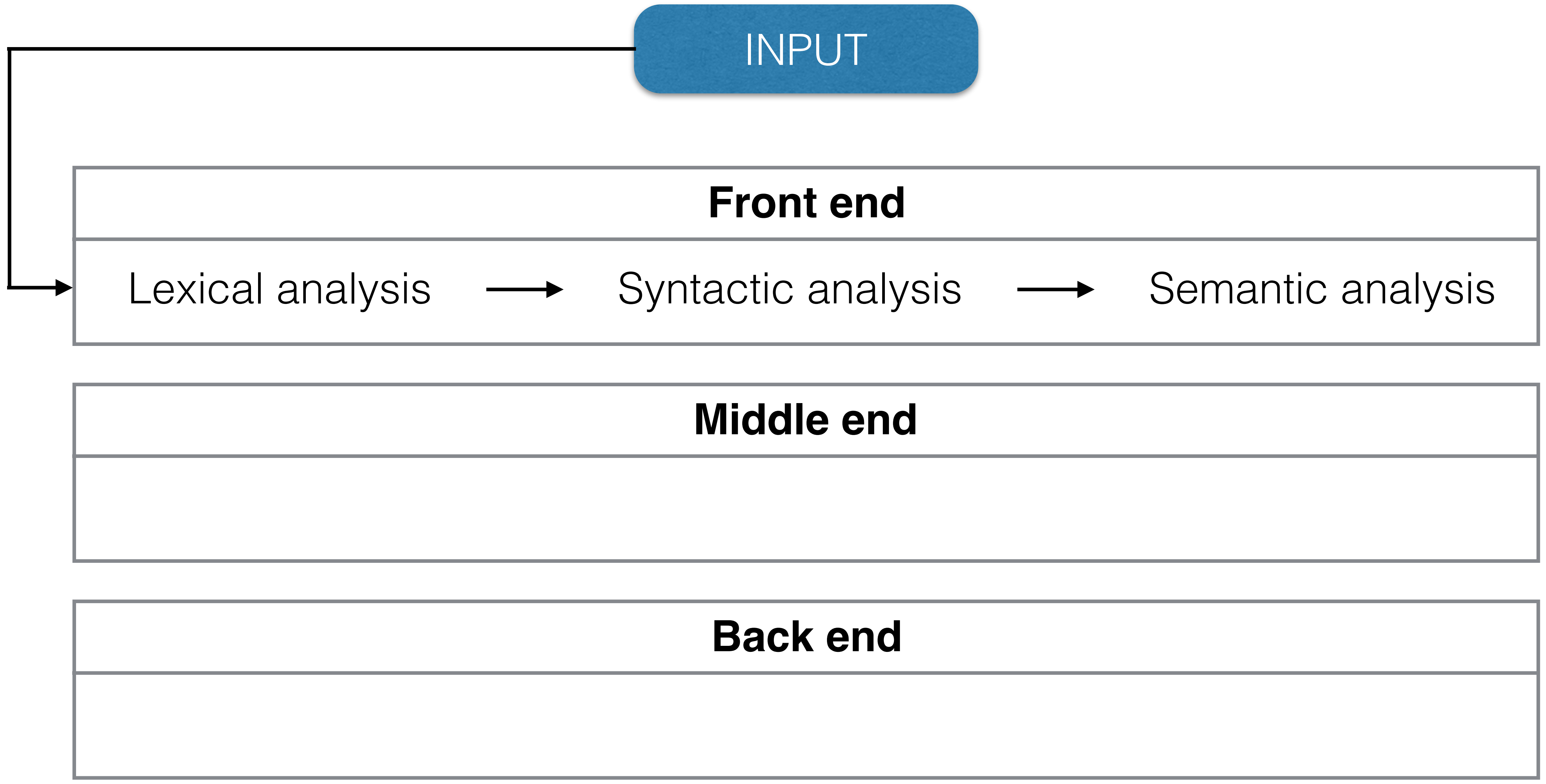
Structure of a compiler



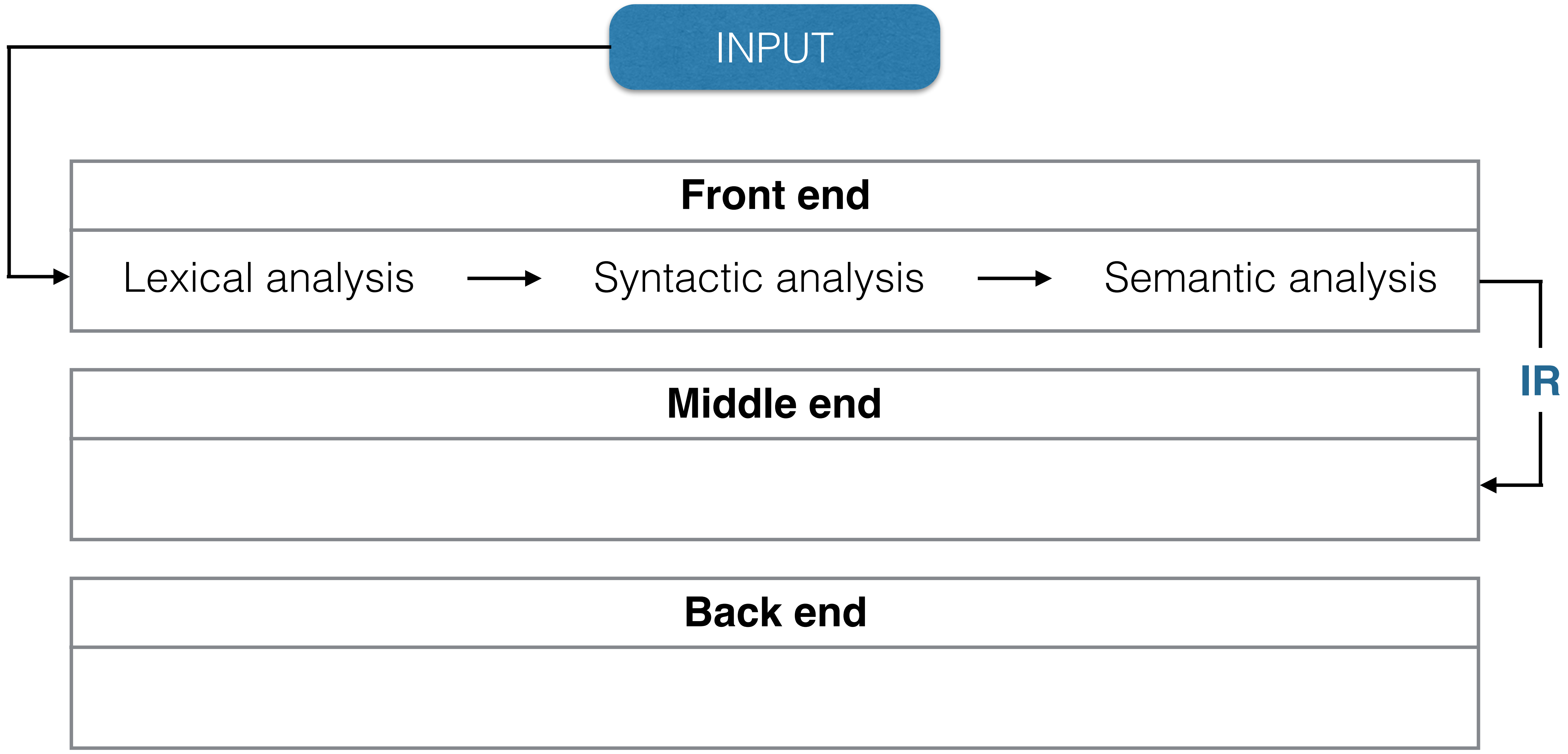
Structure of a compiler



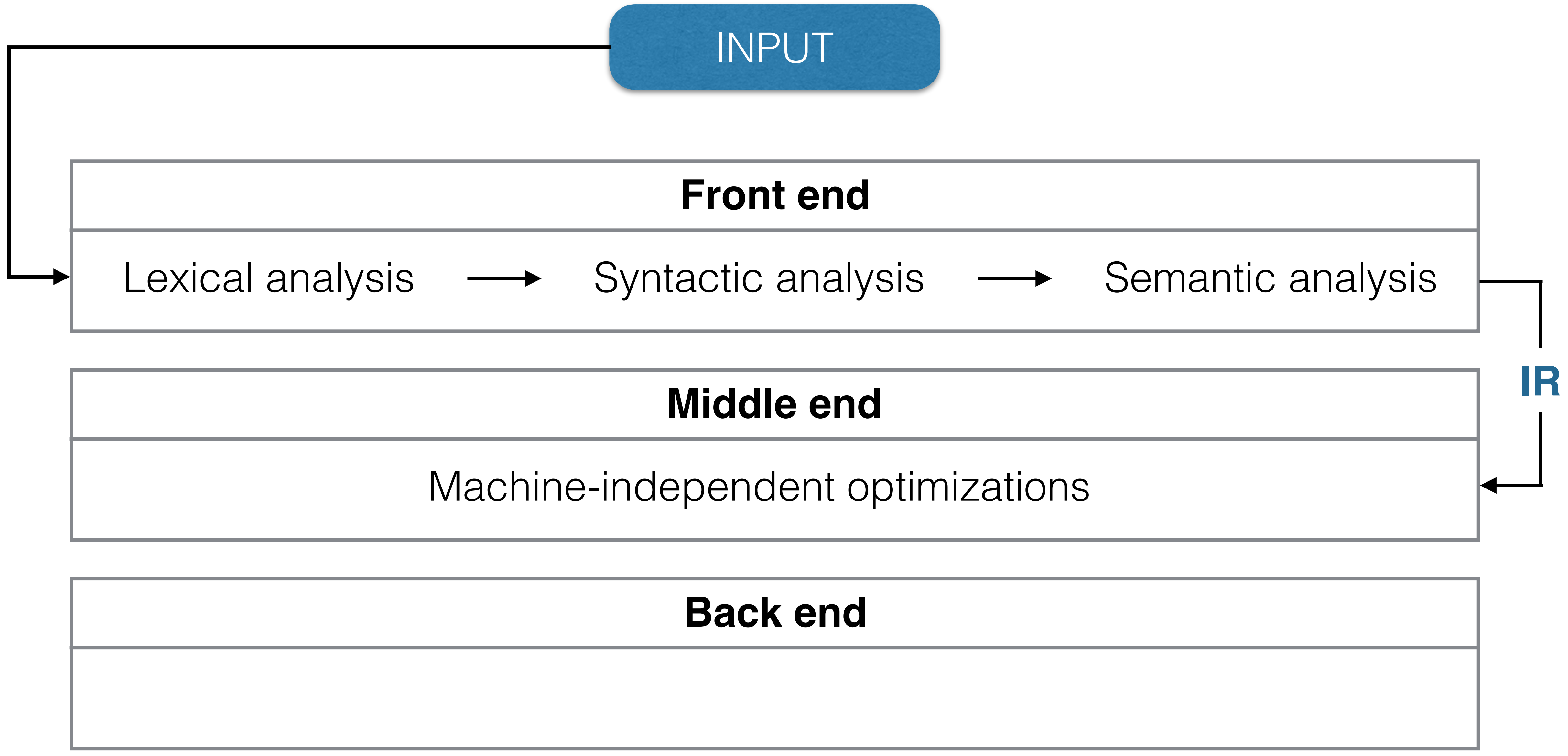
Structure of a compiler



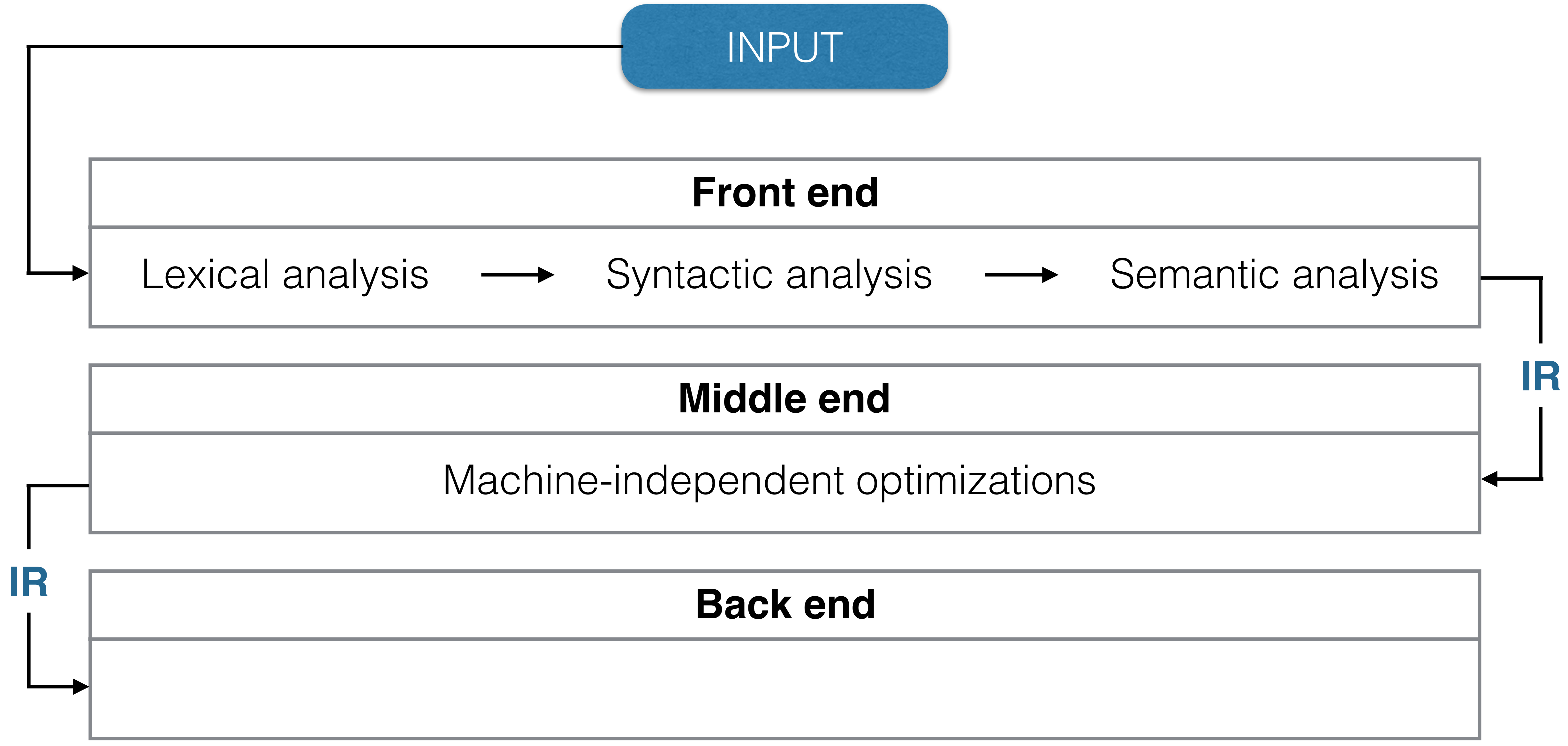
Structure of a compiler



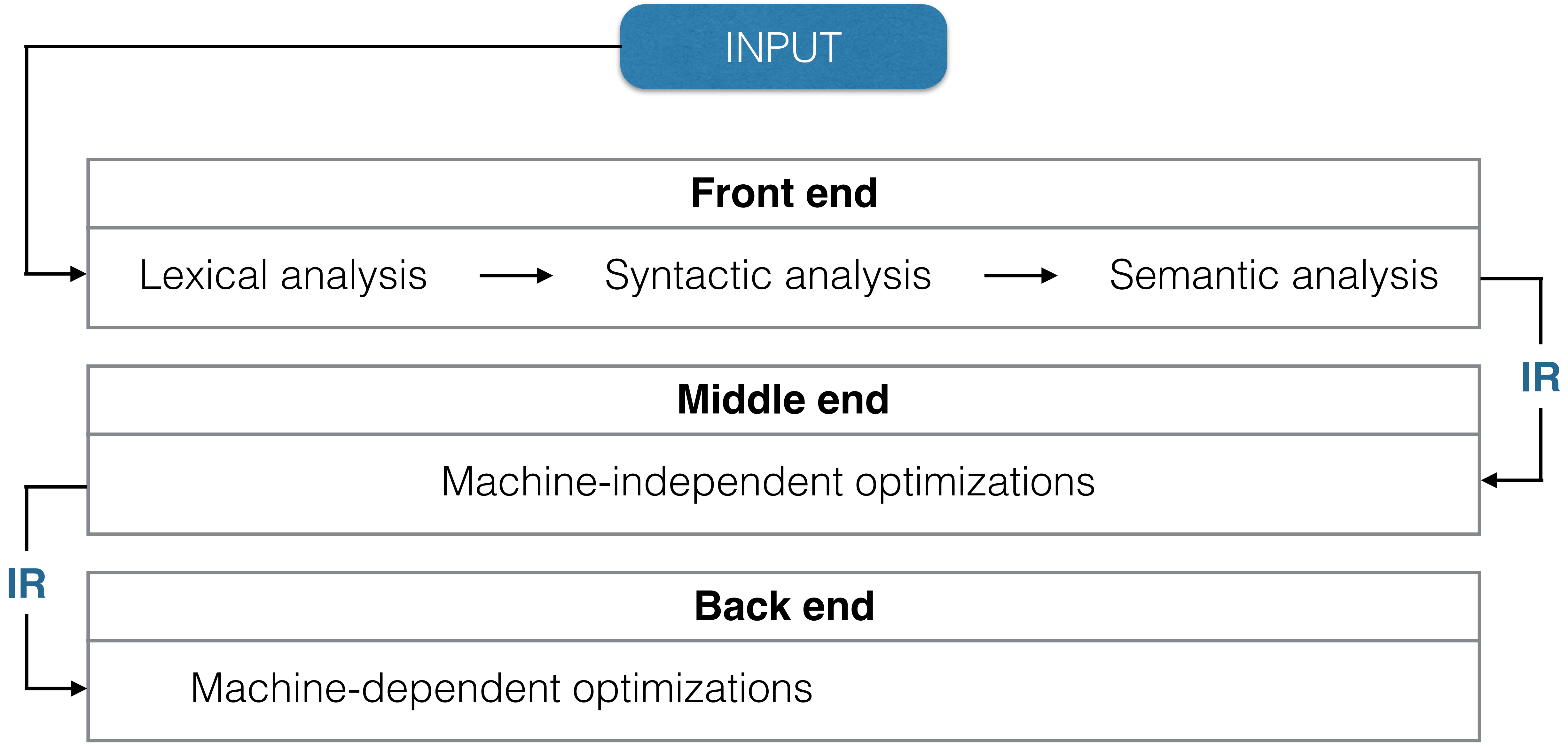
Structure of a compiler



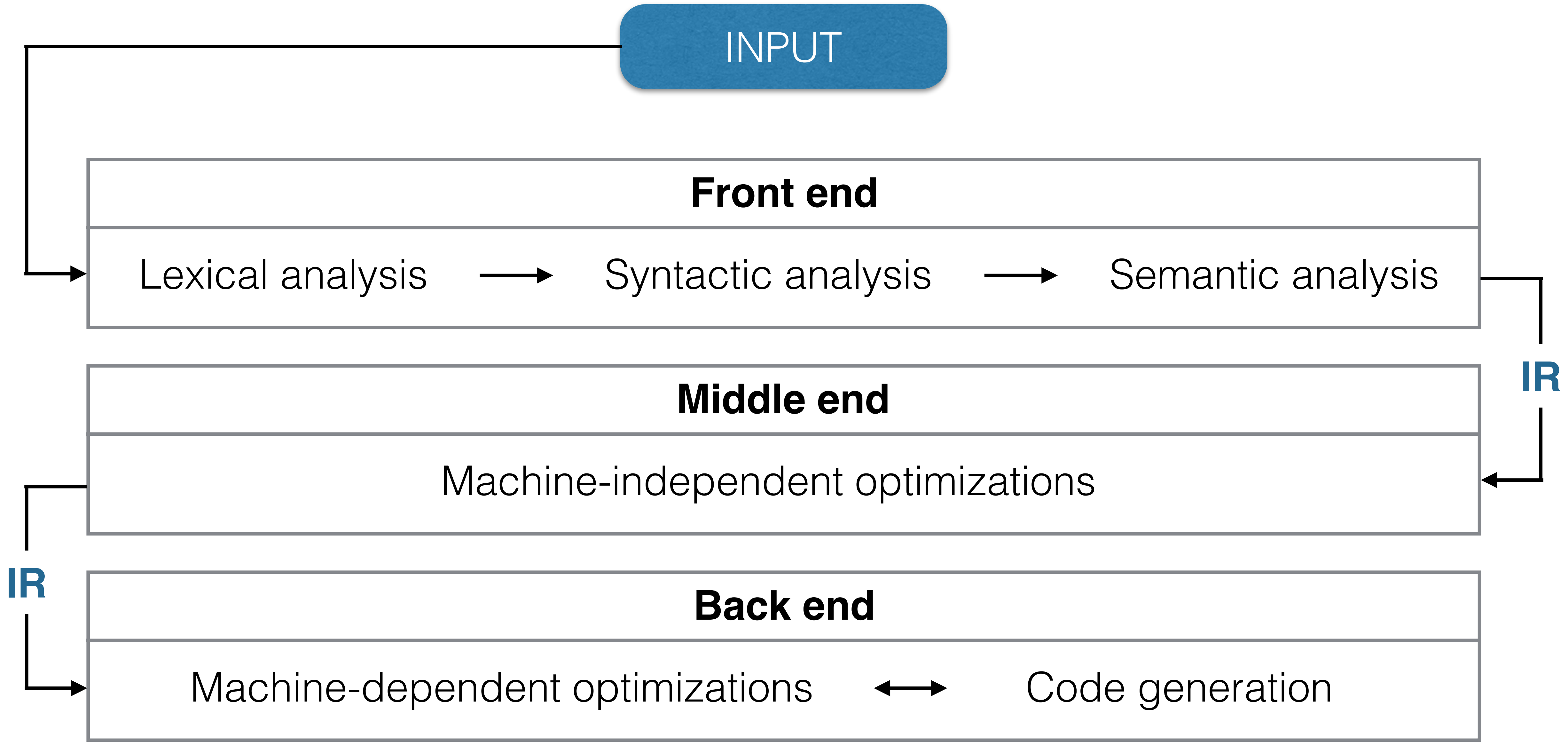
Structure of a compiler



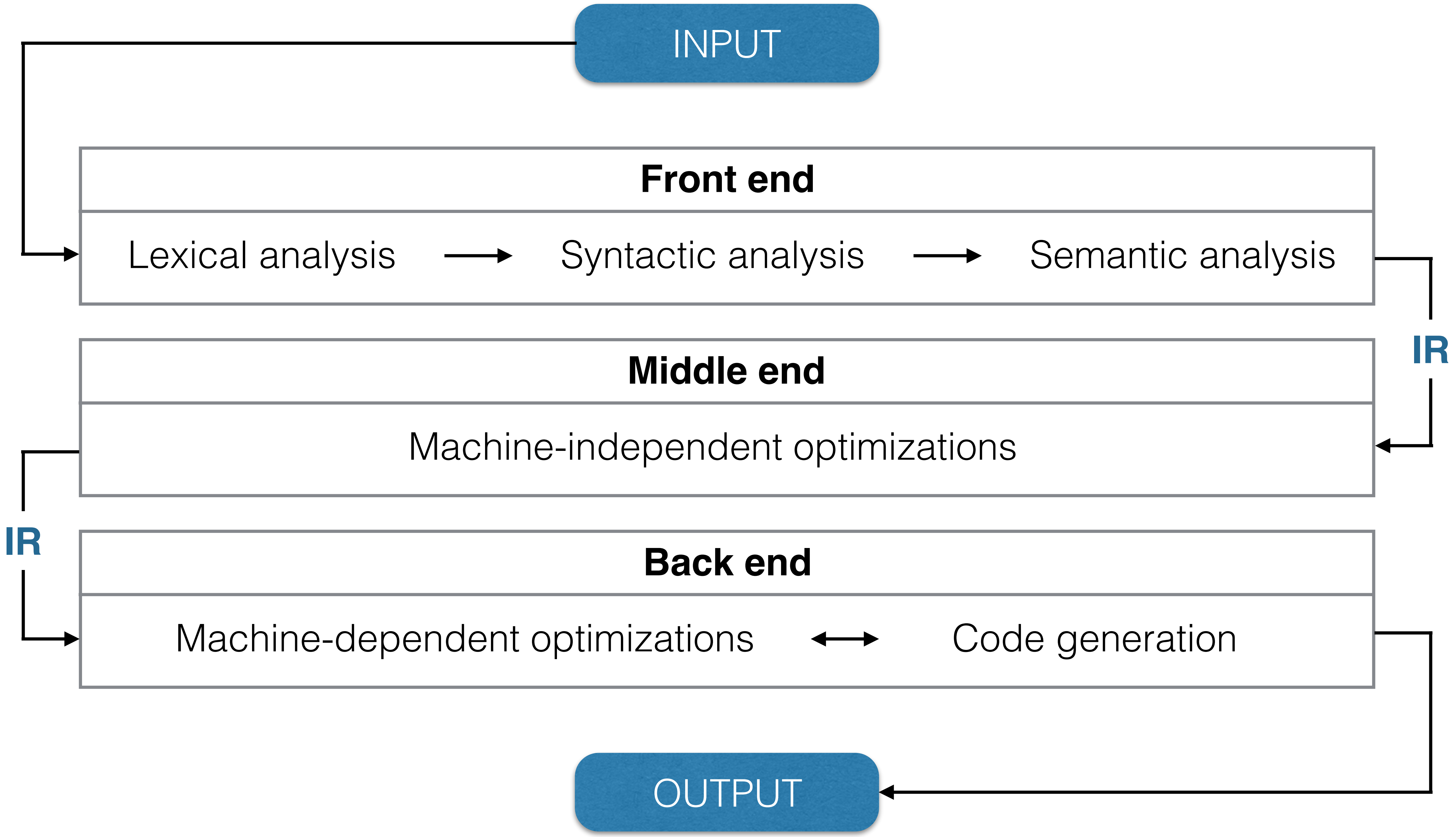
Structure of a compiler



Structure of a compiler



Structure of a compiler



Lexical analysis

Lexical analysis

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

Lexical analysis

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

- Regular definition:
  `token_class` $\longrightarrow$ *`regular_expression`*

Lexical analysis

Lexical analysis

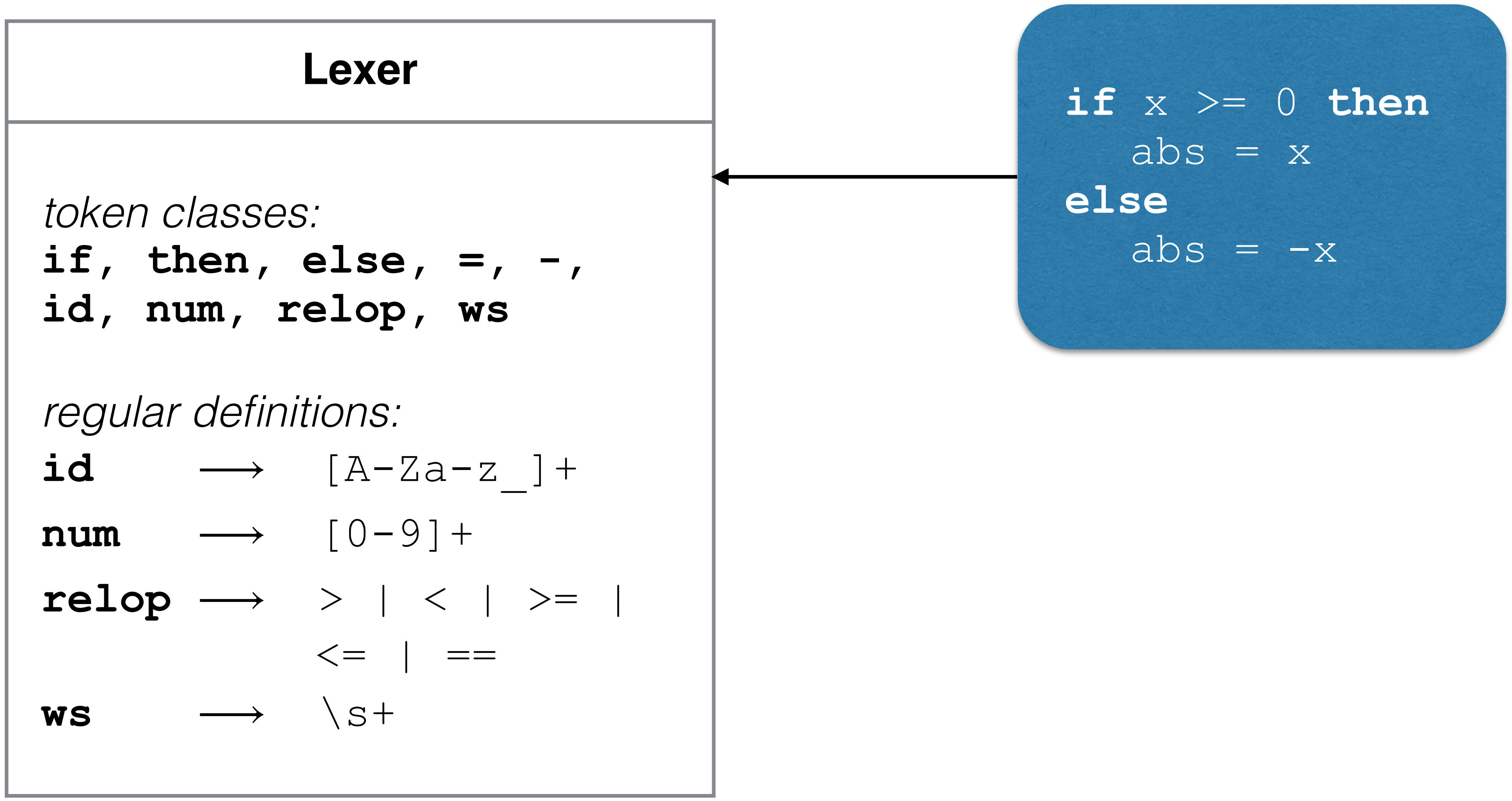
Lexer

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

regular definitions:

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

Lexical analysis



Lexer

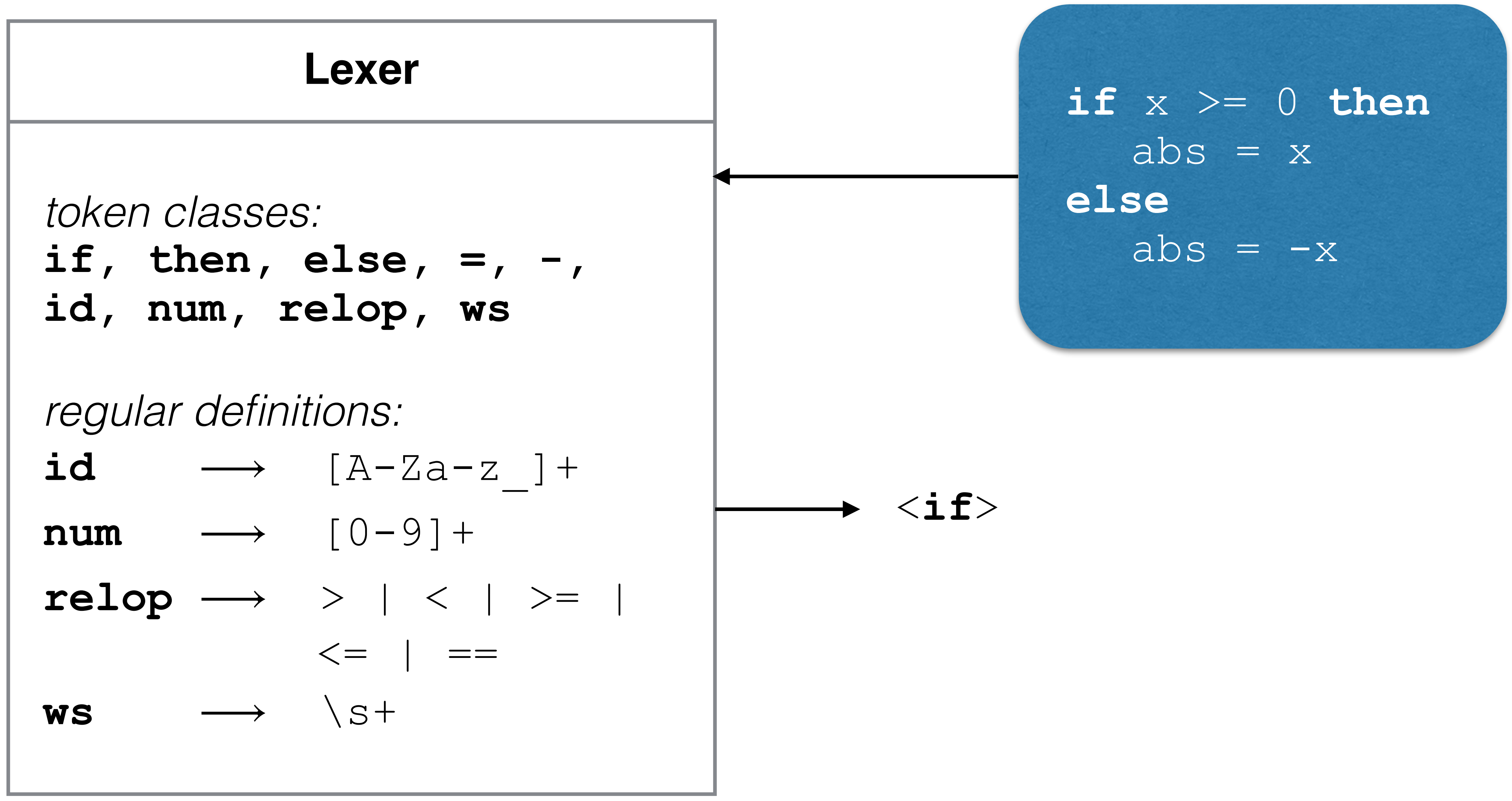
token classes:
`if, then, else, =, -,`
`id, num, relop, ws`

regular definitions:

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

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

Lexical analysis



Lexer

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

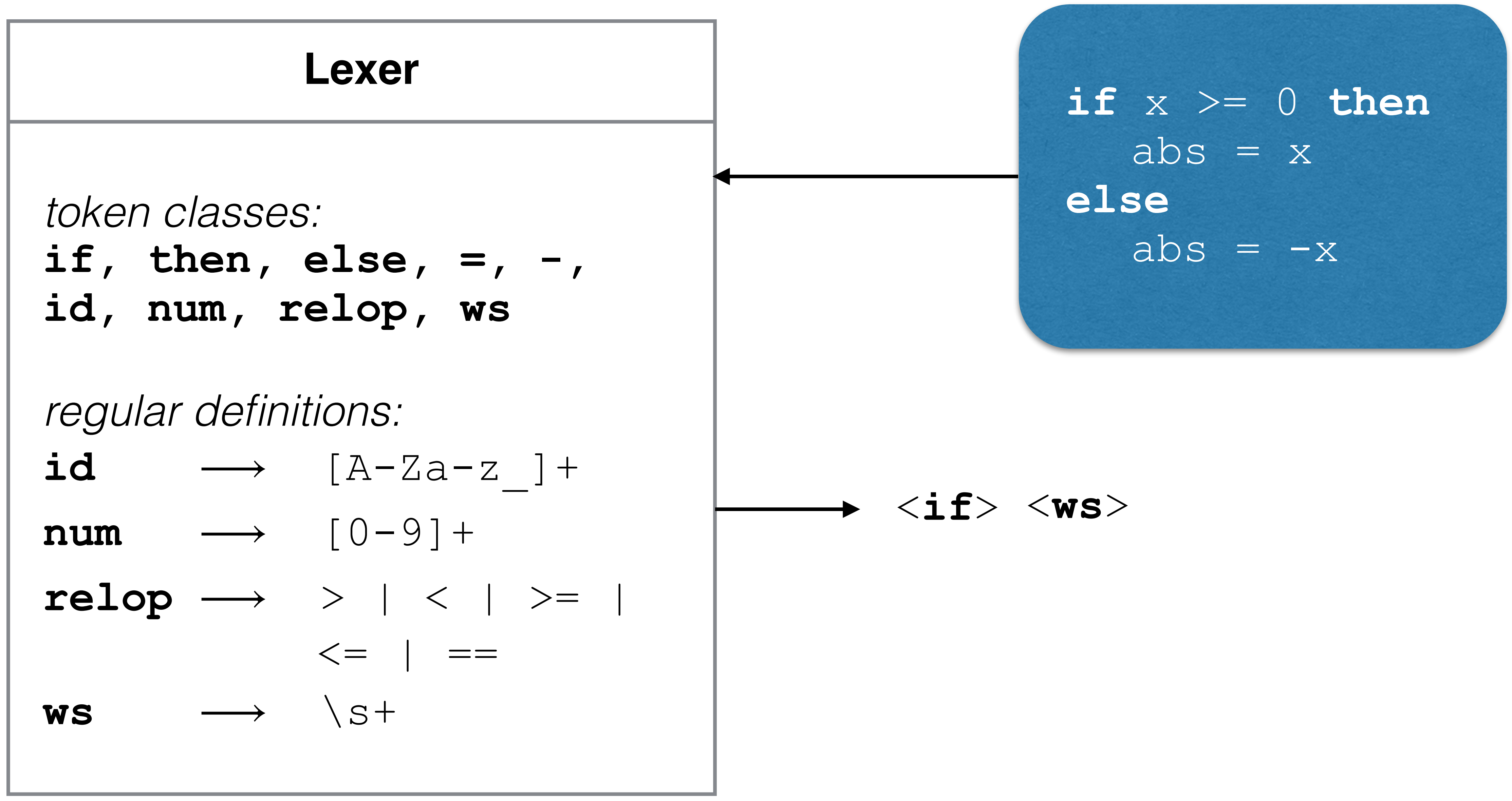
regular definitions:

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

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

<**if**>

Lexical analysis



Lexer

token classes:

if, then, else, =, -,
id, num, relop, ws

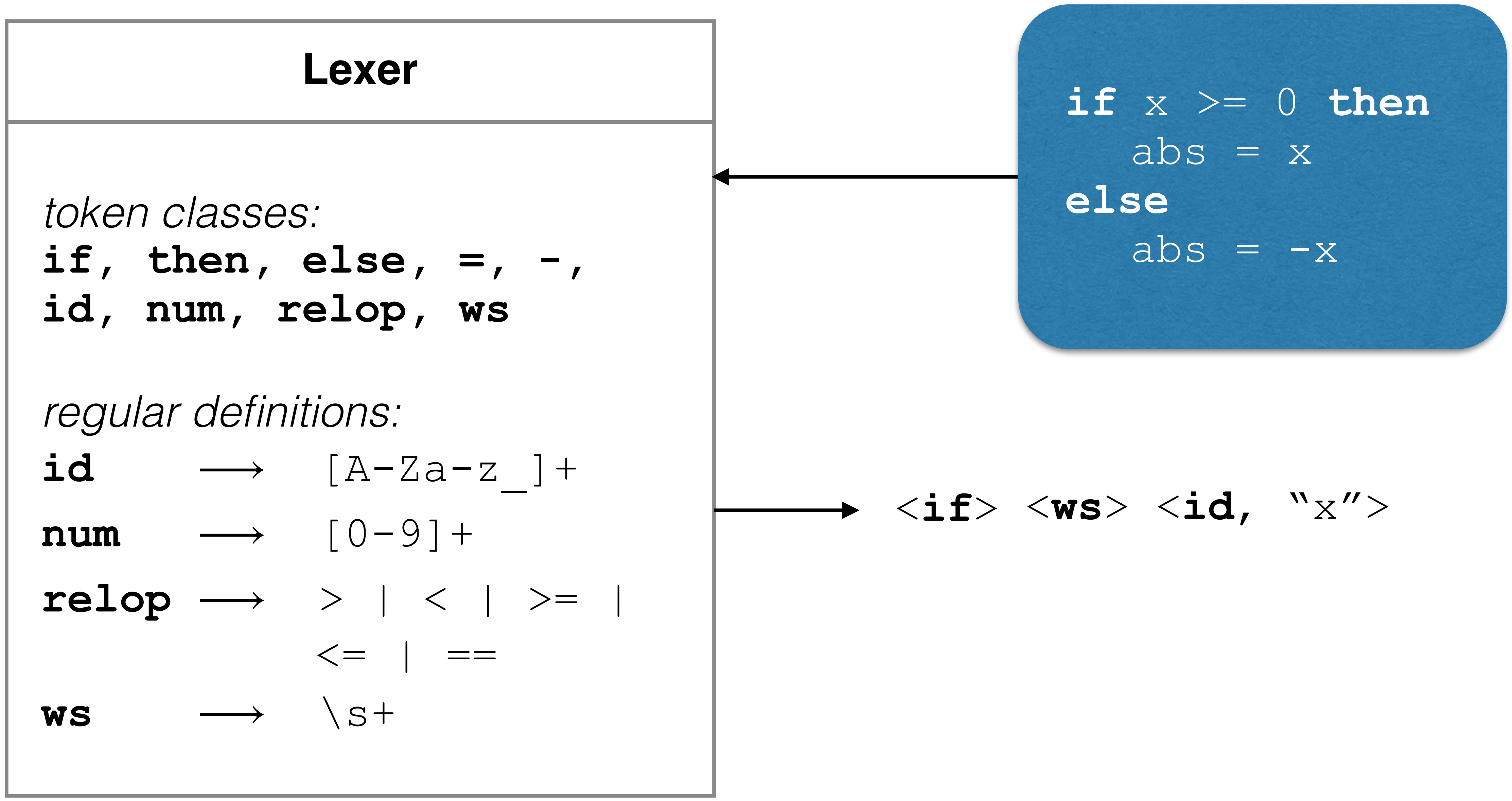
regular definitions:

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

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

<**if**> <**ws**>

Lexical analysis



Lexer

token classes:

```
if, then, else, =, -,
id, num, relop, ws
```

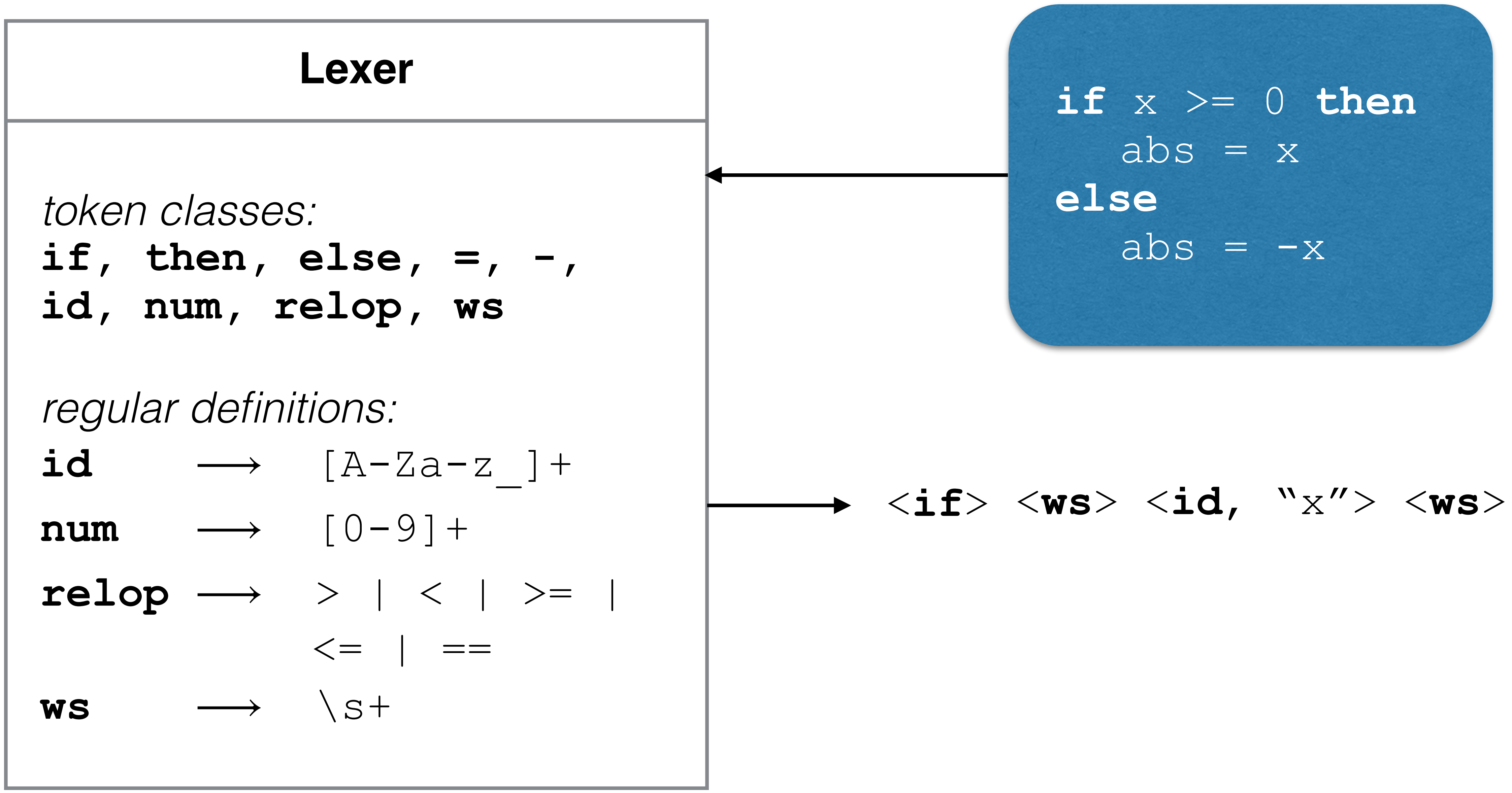
regular definitions:

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

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

```
<if> <ws> <id, “x”>
```

Lexical analysis



Lexer

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

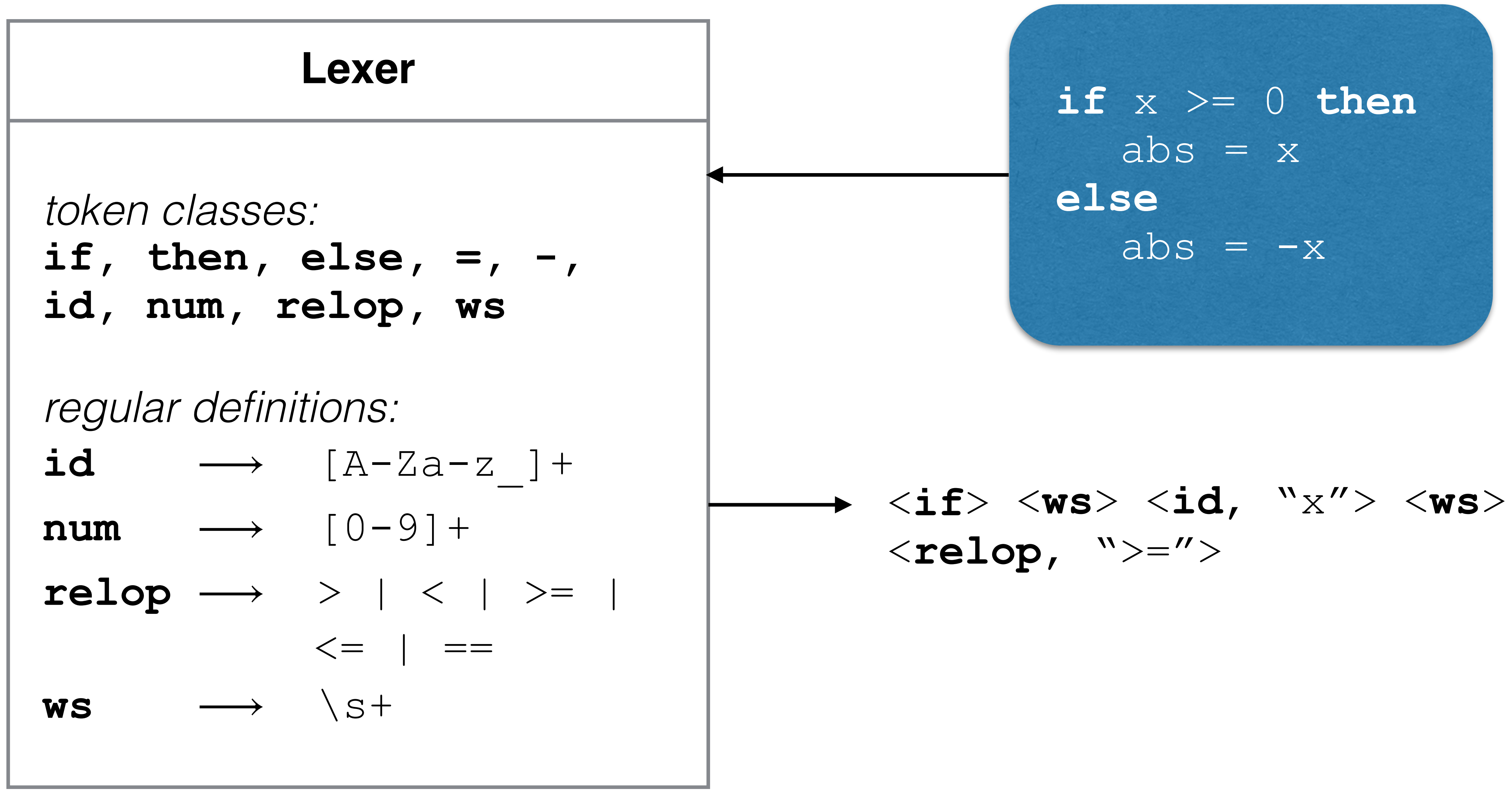
regular definitions:

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

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

<**if**> <**ws**> <**id**, “x”> <**ws**>

Lexical analysis



Lexer

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

regular definitions:

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

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

<**if**> <**ws**> <**id**, "x"> <**ws**>
<**relop**, ">=">

Lexical analysis

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

Lexer

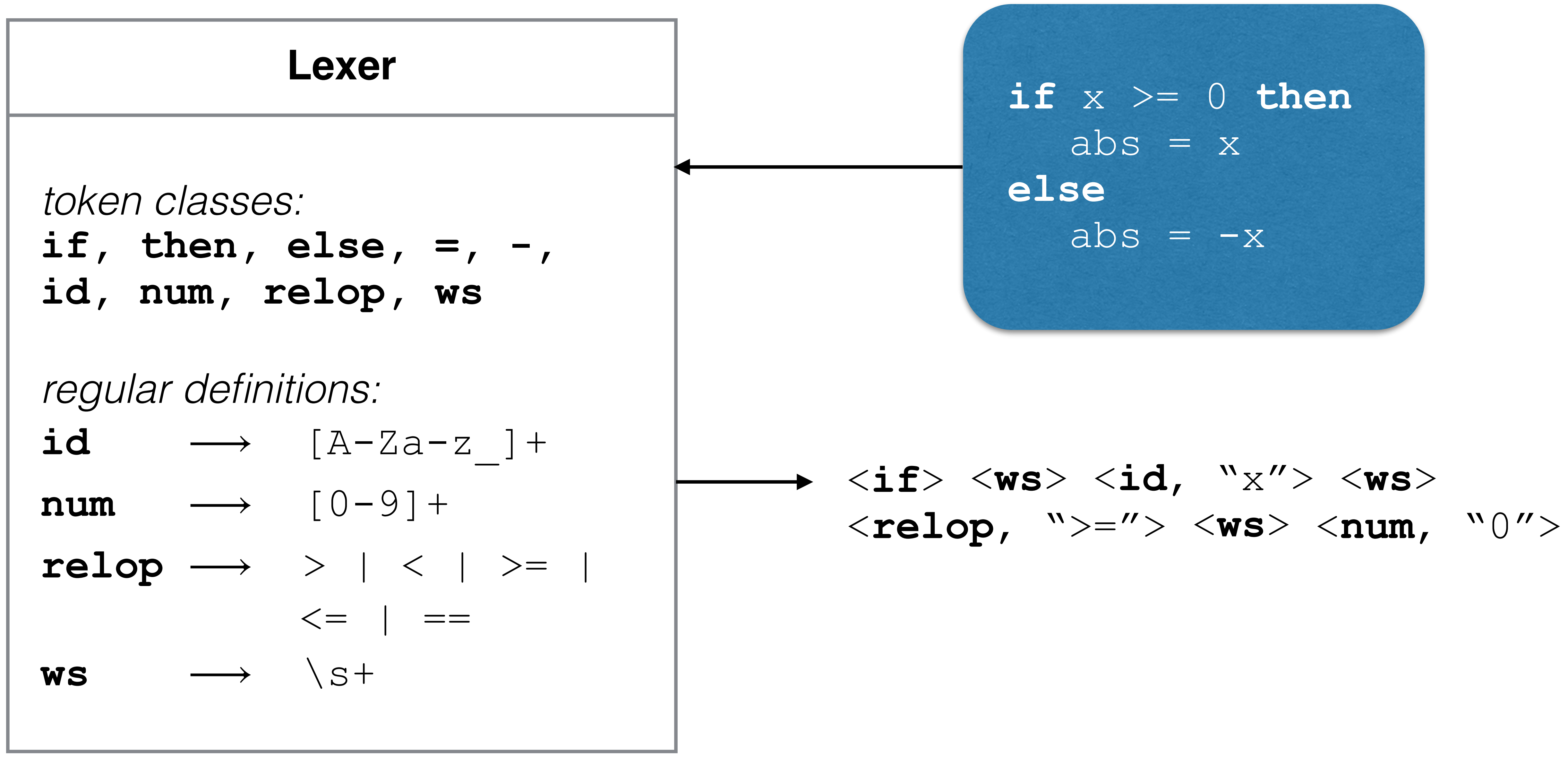
token classes:
if, then, else, =, -, id, num, relop, ws

regular definitions:

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

<**if**> <**ws**> <**id**, "x"> <**ws**>
<**relop**, ">="> <**ws**>

Lexical analysis



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

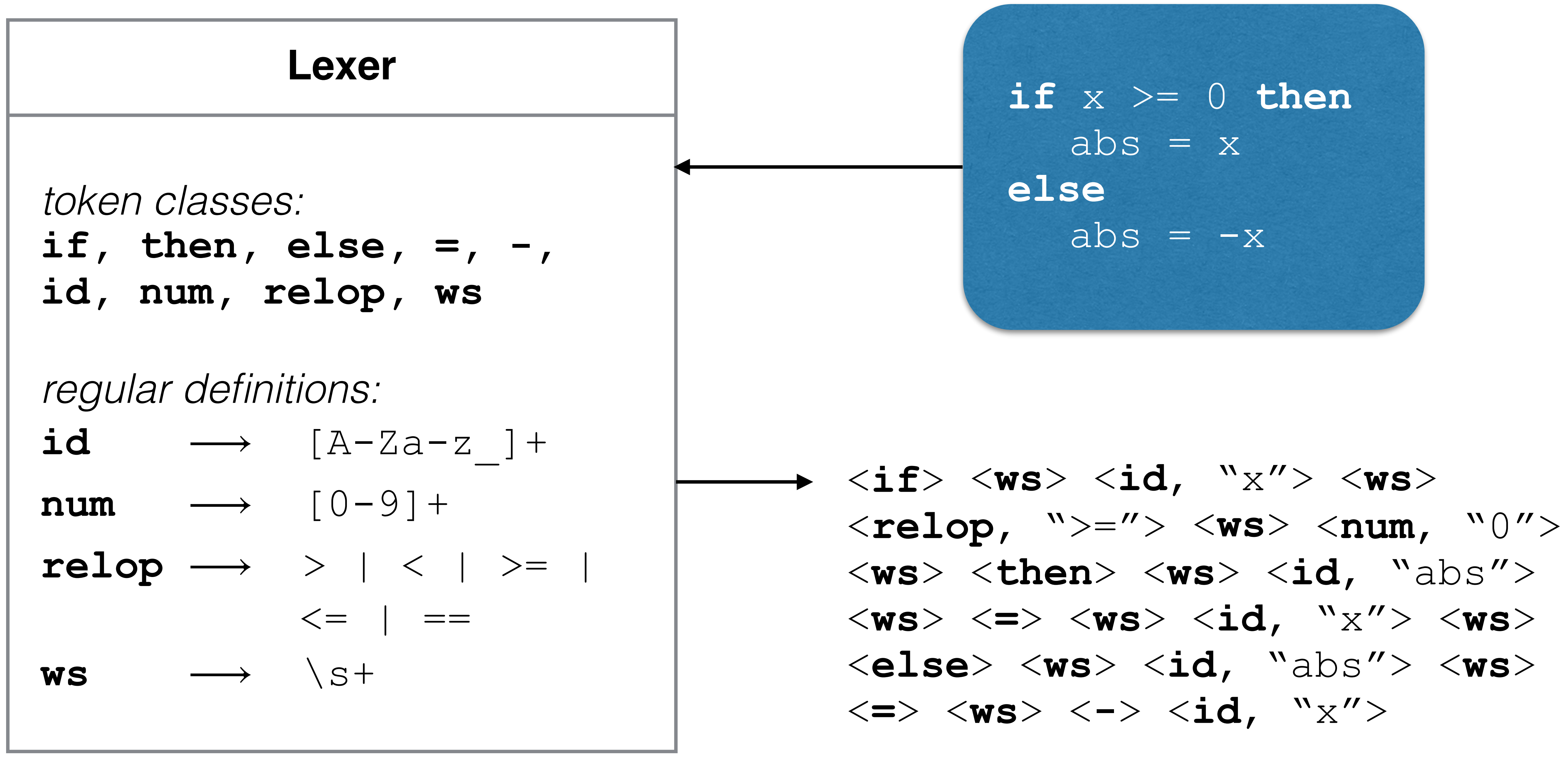
token classes:
if, then, else, =, -,
id, num, relop, ws

regular definitions:

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

```
<if> <ws> <id, "x"> <ws>
<relop, ">="> <ws> <num, "0">
```

Lexical analysis



Lexer

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

regular definitions:

id	$\longrightarrow$	`[A-Za-z_]+`
num	$\longrightarrow$	`[0-9]+`
relop	$\longrightarrow$	`> | < | >= | <= | ==`
ws	$\longrightarrow$	`\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

Syntactic analysis

- Parsers construct the input’s derivation in a formal grammar

Syntactic analysis

- Parsers construct the input’s derivation in a formal grammar
- The Lexer’s tokens are the Parser’s terminals

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*

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)*

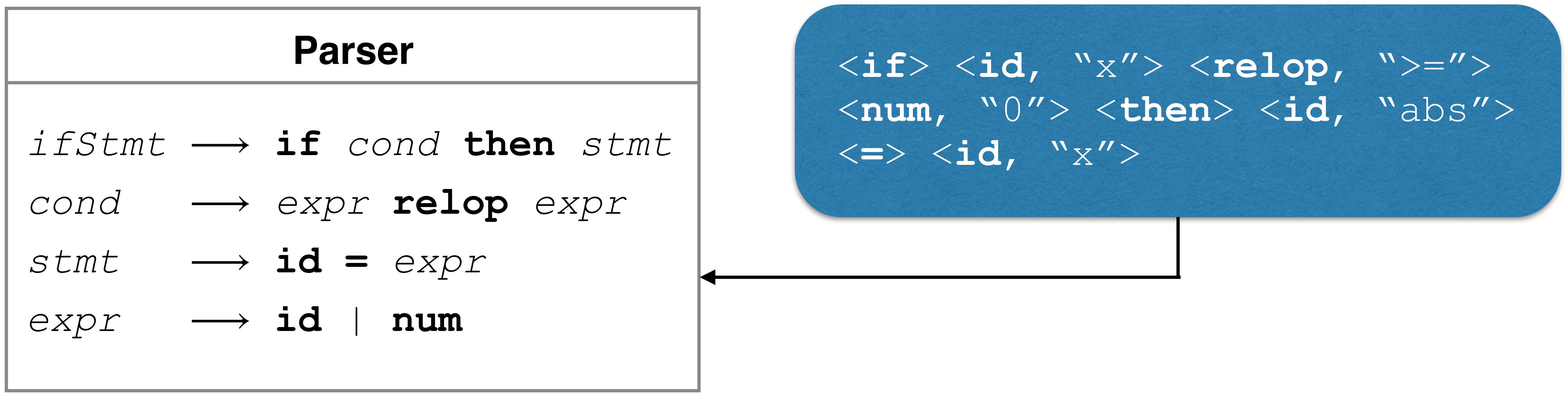
Syntactic analysis

Syntactic analysis

Parser

ifStmt	$\longrightarrow$	**if** *cond* **then** *stmt*
cond	$\longrightarrow$	*expr* **relop** *expr*
stmt	$\longrightarrow$	**id** **=** *expr*
expr	$\longrightarrow$	**id** | **num**

Syntactic analysis



Parser

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

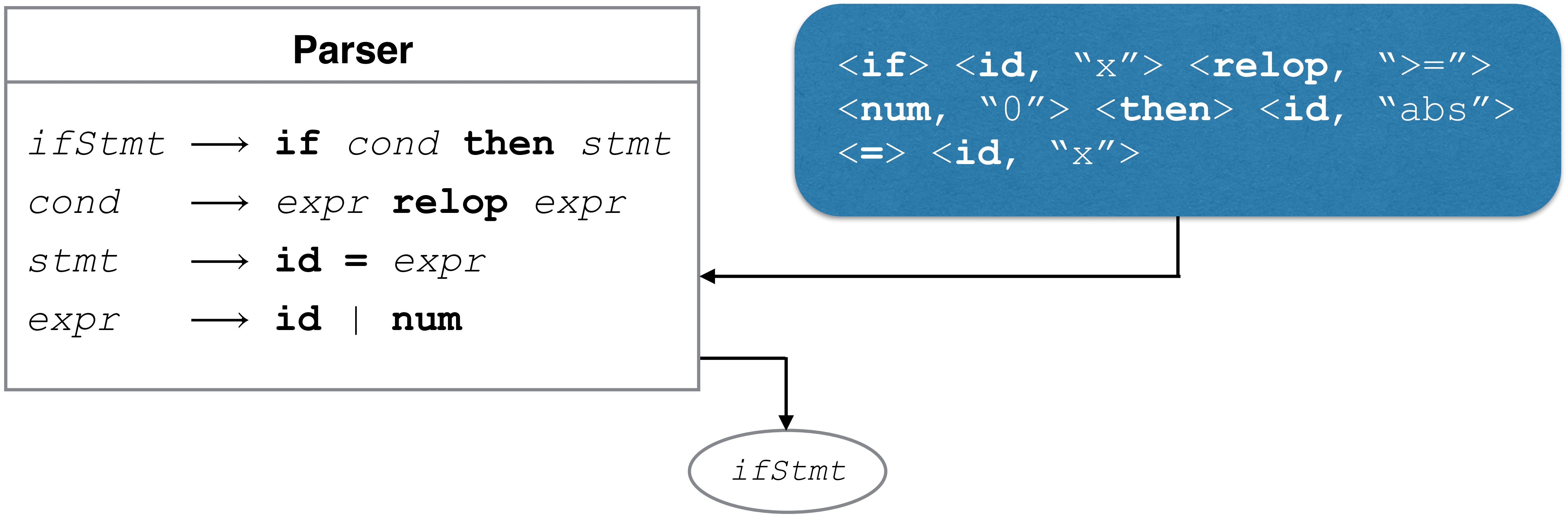
cond ⟶ *expr* **relop** *expr*

stmt ⟶ **id = ** *expr*

expr ⟶ **id** | **num**

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

Syntactic analysis

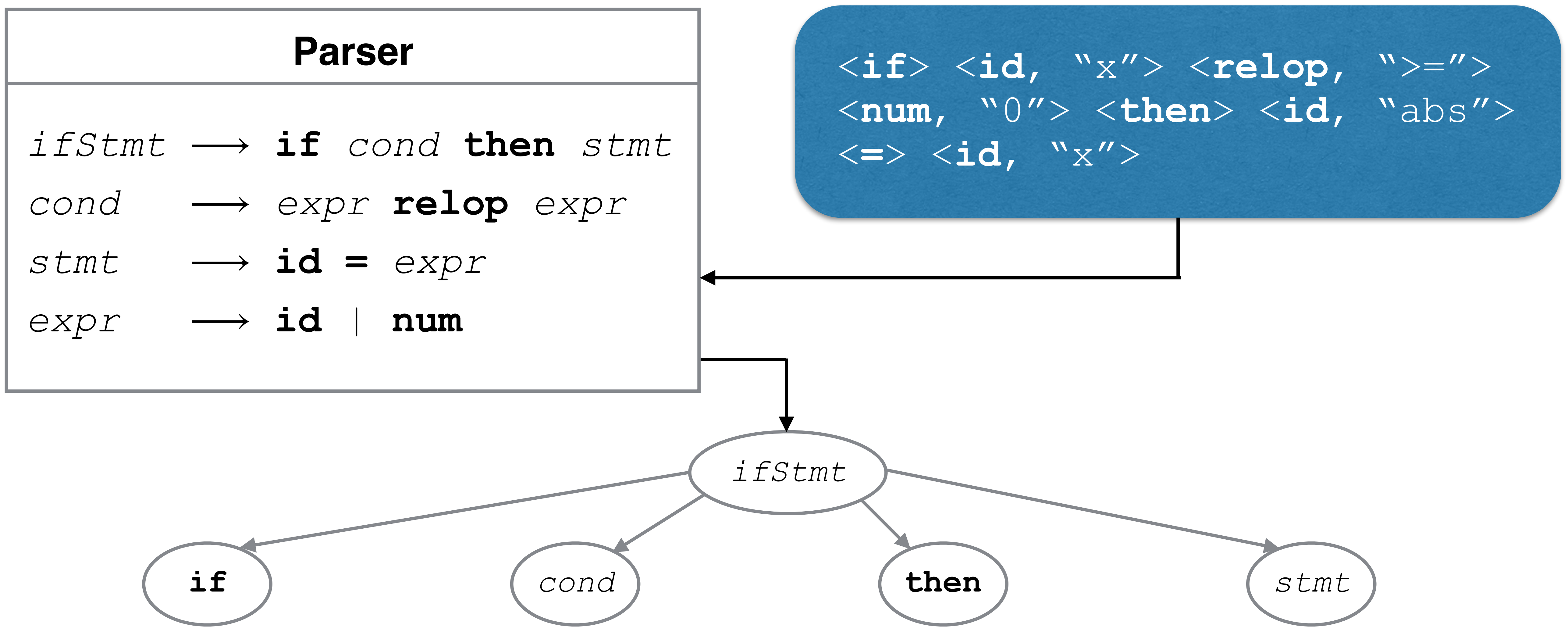


Parser

$$
\begin{array}{lll}
\mathit{ifStmt} & \longrightarrow & \textbf{if}\ \mathit{cond}\ \textbf{then}\ \mathit{stmt} \\
\mathit{cond} & \longrightarrow & \mathit{expr}\ \textbf{relop}\ \mathit{expr} \\
\mathit{stmt} & \longrightarrow & \textbf{id}\ \textbf{=}\ \mathit{expr} \\
\mathit{expr} & \longrightarrow & \textbf{id}\ |\ \textbf{num}
\end{array}
$$

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

Syntactic analysis

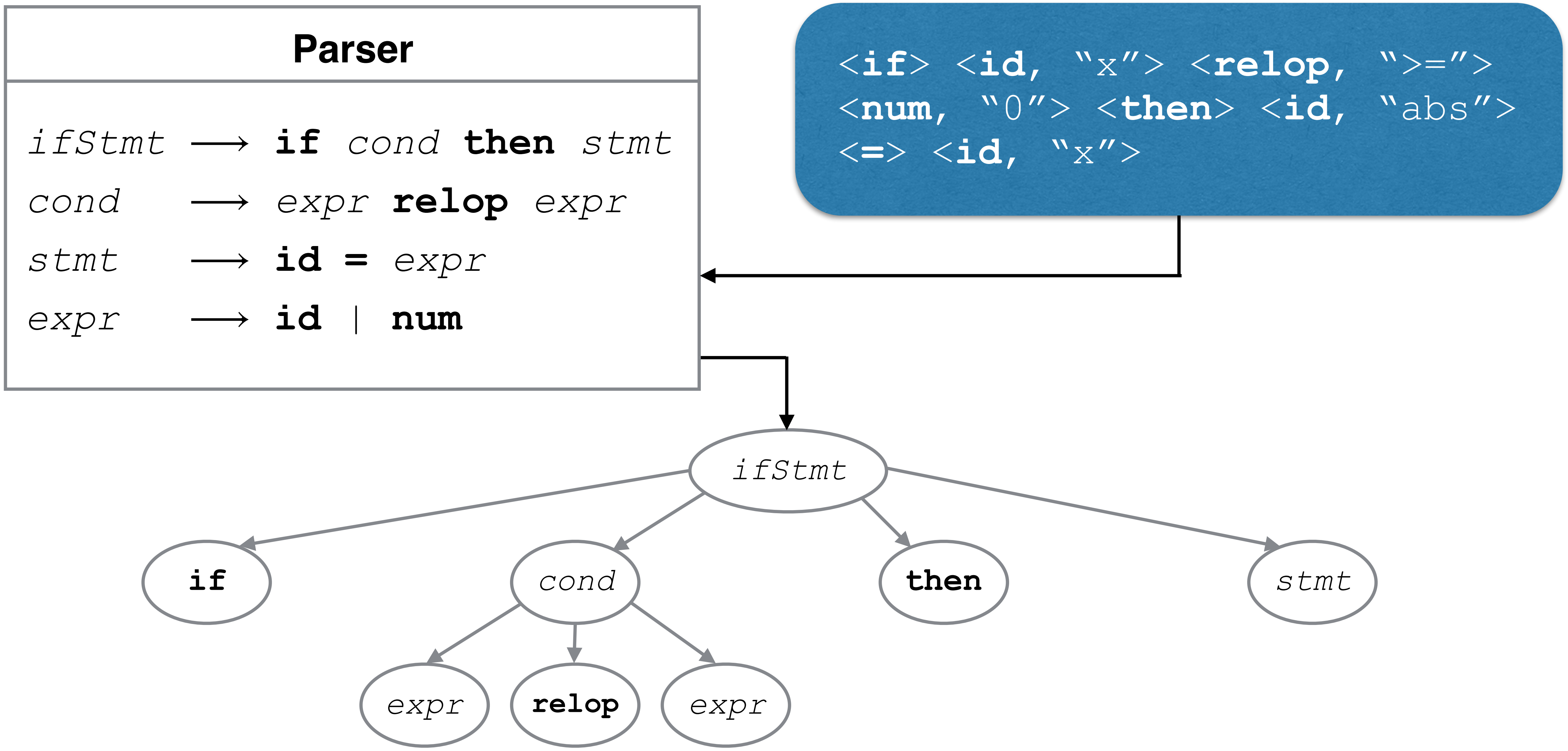


Parser

$$
\begin{array}{lcl}
\mathit{ifStmt} & \longrightarrow & \mathbf{if}\ \mathit{cond}\ \mathbf{then}\ \mathit{stmt} \\
\mathit{cond} & \longrightarrow & \mathit{expr}\ \mathbf{relop}\ \mathit{expr} \\
\mathit{stmt} & \longrightarrow & \mathbf{id}\ \mathbf{=}\ \mathit{expr} \\
\mathit{expr} & \longrightarrow & \mathbf{id}\ \mid\ \mathbf{num}
\end{array}
$$

<**if**> <**id**, “x”> <**relop**, “>=”> <**num**, “0”> <**then**> <**id**, “abs”> <**=**> <**id**, “x”>

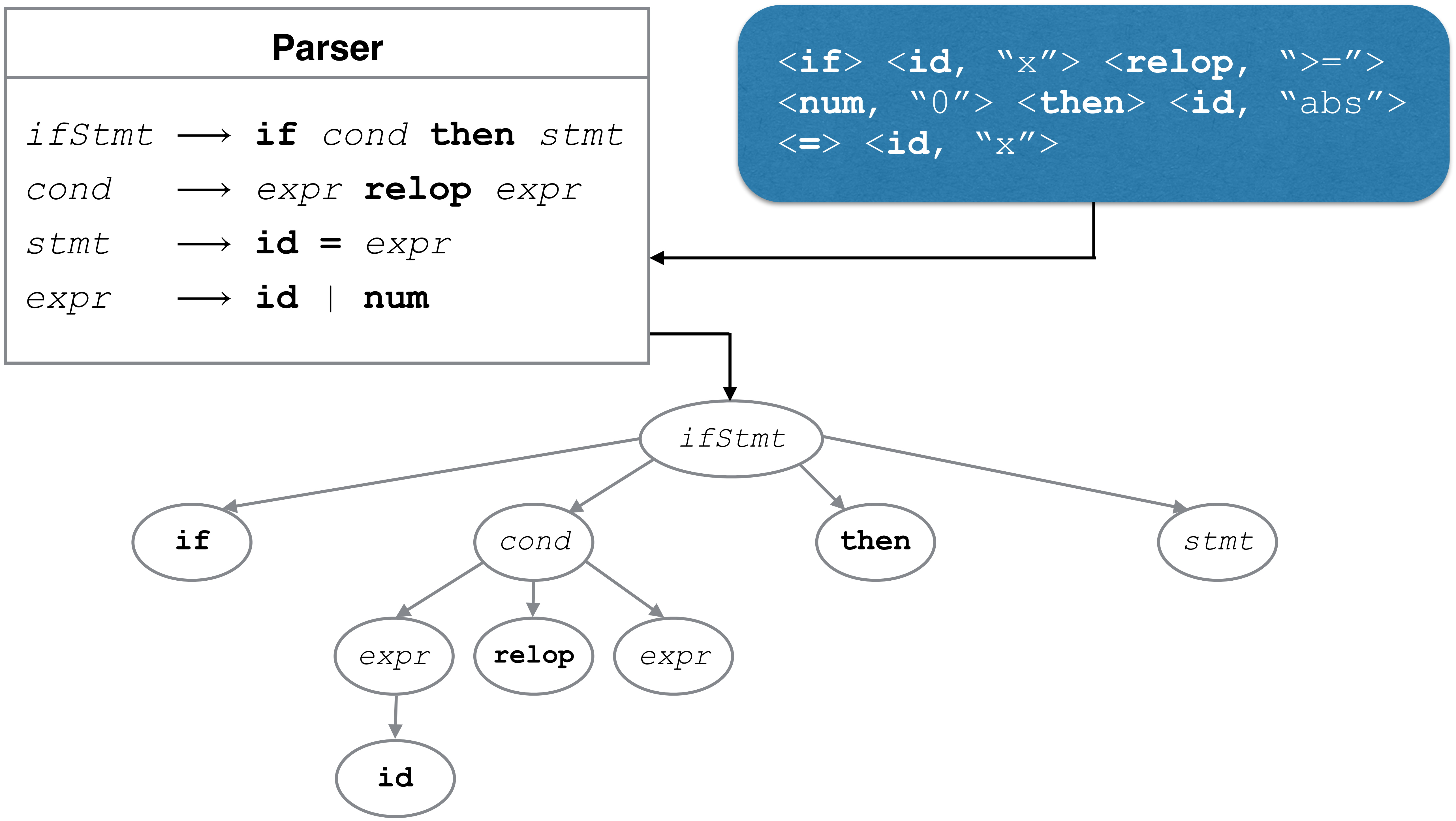
Syntactic analysis



```
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

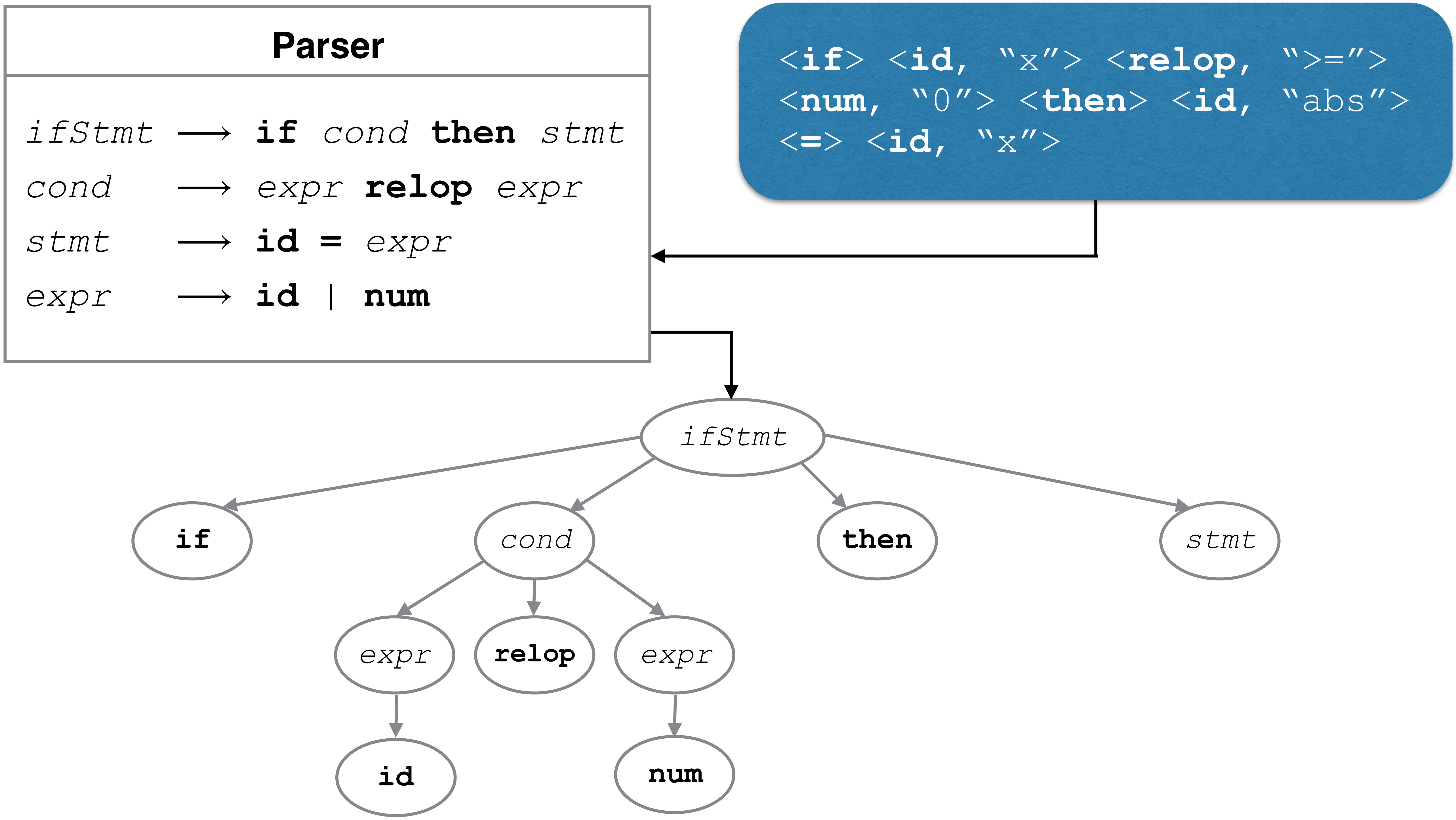


Parser

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">

Syntactic analysis

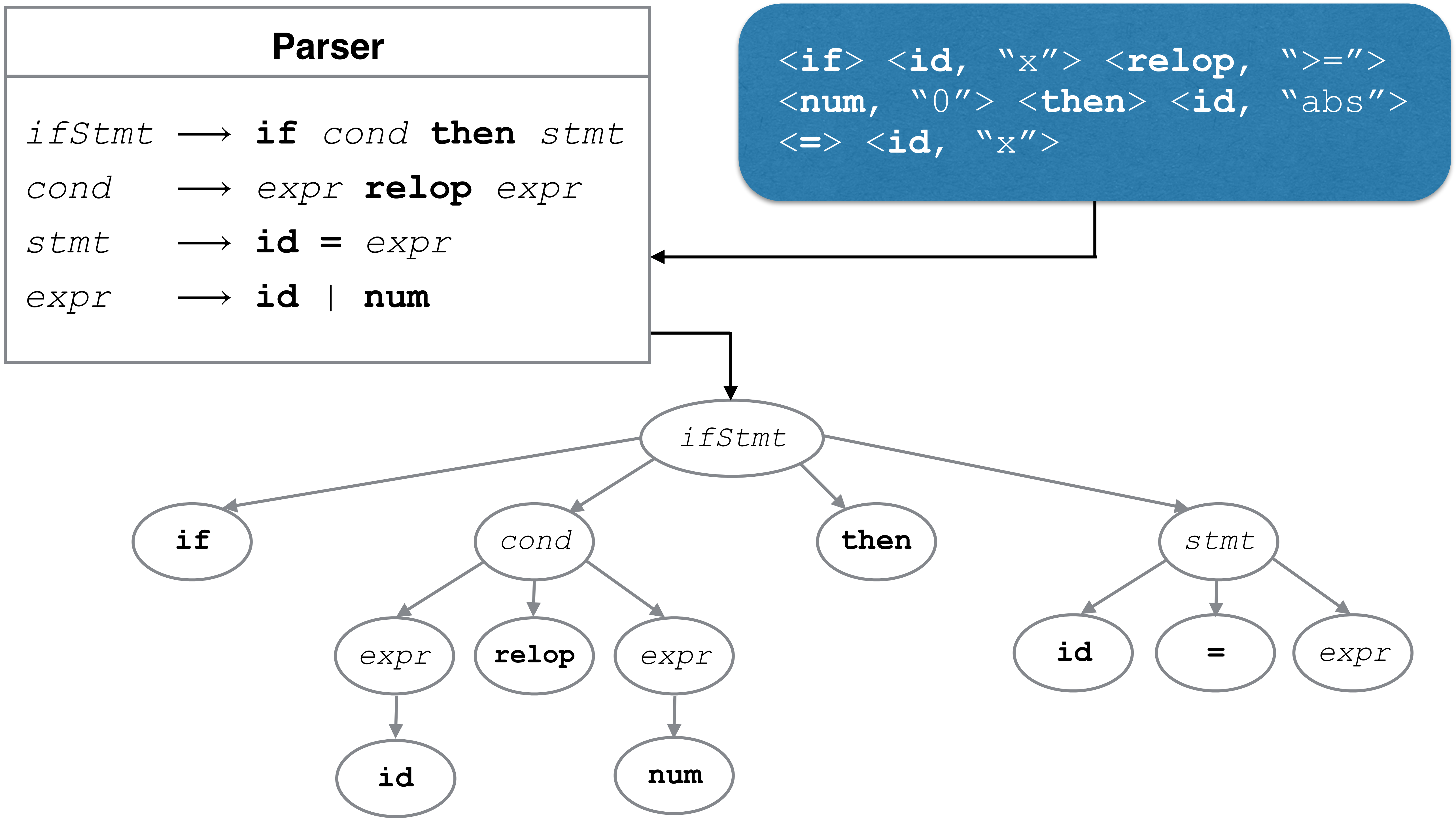


Parser

$$
\begin{array}{lcl}
\mathit{ifStmt} & \longrightarrow & \textbf{if}\ \mathit{cond}\ \textbf{then}\ \mathit{stmt} \\
\mathit{cond} & \longrightarrow & \mathit{expr}\ \textbf{relop}\ \mathit{expr} \\
\mathit{stmt} & \longrightarrow & \textbf{id}\ \textbf{=}\ \mathit{expr} \\
\mathit{expr} & \longrightarrow & \textbf{id}\ \mid\ \textbf{num}
\end{array}
$$

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

Syntactic analysis

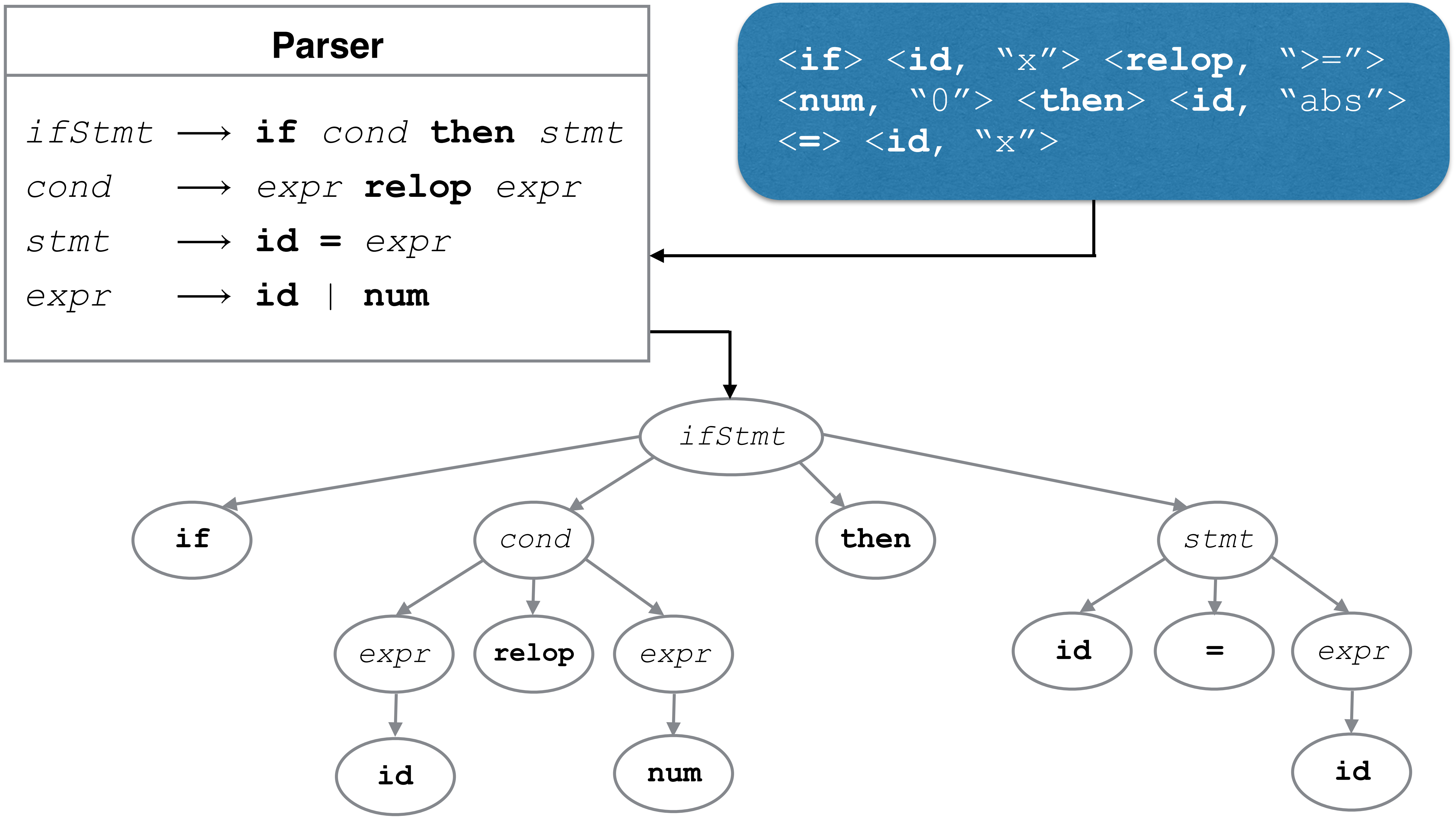


Parser

```
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”>
```

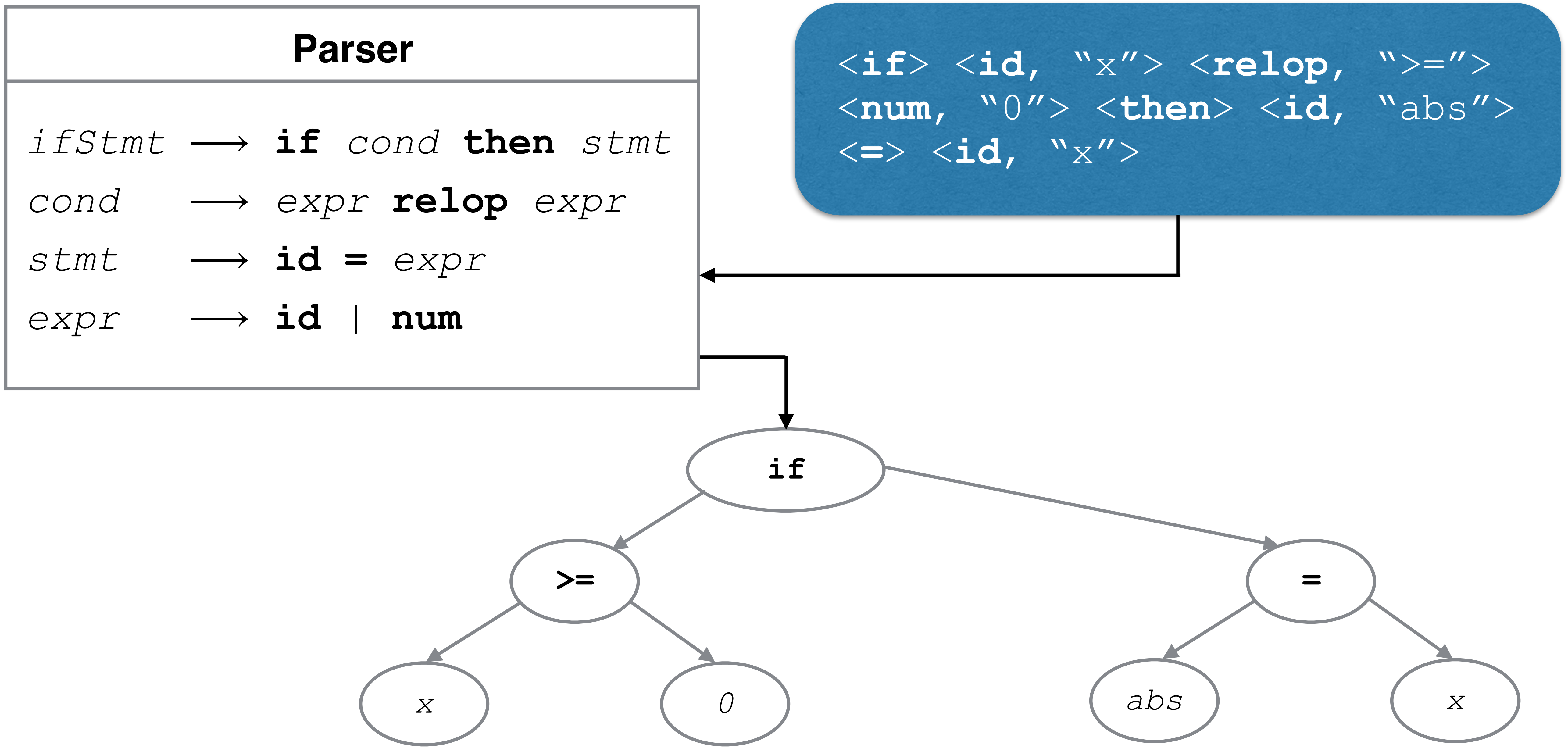
Syntactic analysis



$$
\begin{aligned}
\textit{ifStmt} &\longrightarrow \textbf{if}\ \textit{cond}\ \textbf{then}\ \textit{stmt}\\
\textit{cond} &\longrightarrow \textit{expr}\ \textbf{relop}\ \textit{expr}\\
\textit{stmt} &\longrightarrow \textbf{id}\ \textbf{=}\ \textit{expr}\\
\textit{expr} &\longrightarrow \textbf{id}\ \mid\ \textbf{num}
\end{aligned}
$$

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

Syntactic analysis



Parser

ifStmt	$\longrightarrow$	**if** *cond* **then** *stmt*
cond	$\longrightarrow$	*expr* **relop** *expr*
stmt	$\longrightarrow$	**id** **=** *expr*
expr	$\longrightarrow$	**id** | **num**

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

Semantic analysis

Semantic analysis

- Building/extending the symbol table(s)

Semantic analysis

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

Semantic analysis

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

Optimizations

Optimizations

- Correctness and profitability

Optimizations

- Correctness and profitability
- Most optimizations run in two phases:
    - Analysis (data-flow, control-flow, etc.)
    - Transformation

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

Machine-independent optimizations

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

Machine-dependent optimizations

Machine-dependent optimizations

- Peephole optimizations
- Register allocation
- Instruction scheduling
- Trampolines

How to write a compiler?

How to write a compiler?

1. Learn some theory on lexical and syntactic analysis

    - <u>The Dragon Book</u>
    - <u>Prof. Alex Aiken's Compilers course @ Coursera</u>

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

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

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!

Demo

Questions?