Made by Qingkai Shi
qingkaishi@nju.edu.cn
南京大学编译技术
及软件安全研究组

Chapter 1
Introduction to Compilers

授课老师及助教

- 冯洋，南京大学 计算机学院 313室 （一班）
- 主页：https://fengyang-nju.github.io/ 邮箱: fengyang@nju.edu.cn

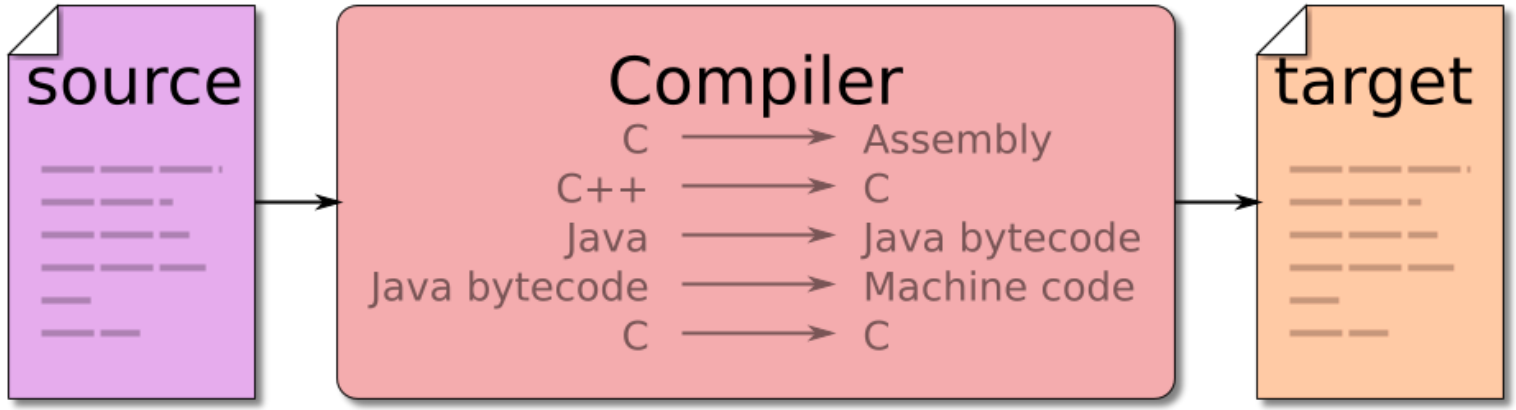
- 时清凯，南京大学 计算机学院 518室 （二班）
- 主页：https://qingkaishi.github.io/ 邮箱: qingkaishi@nju.edu.cn

- 吴杰伦（jielunwu@smail.nju.edu.cn）储　备（beichu@smail.nju.edu.cn）
- 张城铨（chengquanzhang@smail.nju.edu.cn）
- 傅小龙（191220029@smail.nju.edu.cn）
- 宣宇豪（221830019@smail.nju.edu.cn）
- 王朝晖（221850037@smail.nju.edu.cn）

PART I: What is a Compiler?

What is a Compiler?

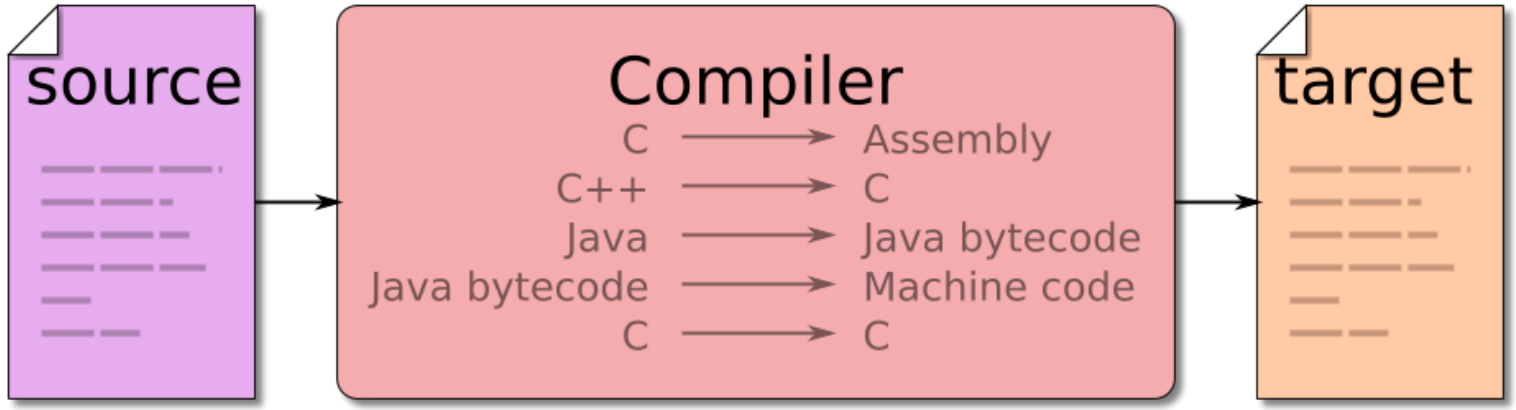
- A compiler translates source language into target language
  - C/C++/Java/… to x86/Java bytecode/…
- Some compiler may also translate a source language to another
  - C to Rust translation, a hot research topic



What is a Compiler?

- A compiler translates source language into target language
  - C/C++/Java/… to x86/Java bytecode/…
- Some compiler may also translate a source language to another
  - C to Rust translation, a hot research topic

Transpiler



Compiler vs. Interpreter

- What is an interpreter? Any differences?

Compiler vs. Interpreter

- Interpreter does not translate the source to another language, but directly interprets statements in the source language
  - Shell, Javascripts, Python, …

Compiler vs. Interpreter

- Interpreter does not translate the source to another language, but directly interprets statements in the source language
  - Shell, Javascripts, Python, …
- **Java is special: compiler + interpreter + just-in-time compiler**

Compiler vs. Linker

- What is a linker?

Compiler vs. Linker

- Linking multiple compilation units into a single file, which could be an executable, a library, …

How does a Compiler/Linker Work?

- Using gcc as the compiler to compile programs in C

How does a Compiler/Linker Work?

- Using gcc as the compiler to compile programs in C

```
gcc   file1.c   file2.c   file3.c   -o   executable.exe
```

How does a Compiler/Linker Work?

- Using gcc as the compiler to compile programs in C

```
gcc  file1.c  file2.c  file3.c  -o  executable.exe
```

Compile

```
gcc  -c  file1.c  -o  file1.o
```

```
gcc  -c  file2.c  -o  file2.o
```

```
gcc  -c  file3.c  -o  file3.o
```

How does a Compiler/Linker Work?

- Using gcc as the compiler to compile programs in C

```
gcc  file1.c  file2.c  file3.c  -o  executable.exe
```

Compile

```
gcc  -c  file1.c  -o  file1.o
```

```
gcc  -c  file2.c  -o  file2.o
```

```
gcc  -c  file3.c  -o  file3.o
```

Link

```
gcc  *.o  -o  executable.exe
```

For example:

- file1.c declares and uses foo()
- file2.c defines foo()

How does a Compiler/Linker Work?

- Using gcc as the compiler to compile programs in C

```
gcc   file1.c   file2.c   file3.c   -o   executable.exe
```

Compile

```
gcc -c -O1 file1.c -o file1.o
```

```
gcc -c -O1 file2.c -o file2.o
```

```
gcc -c -O1 file3.c -o file3.o
```

Link

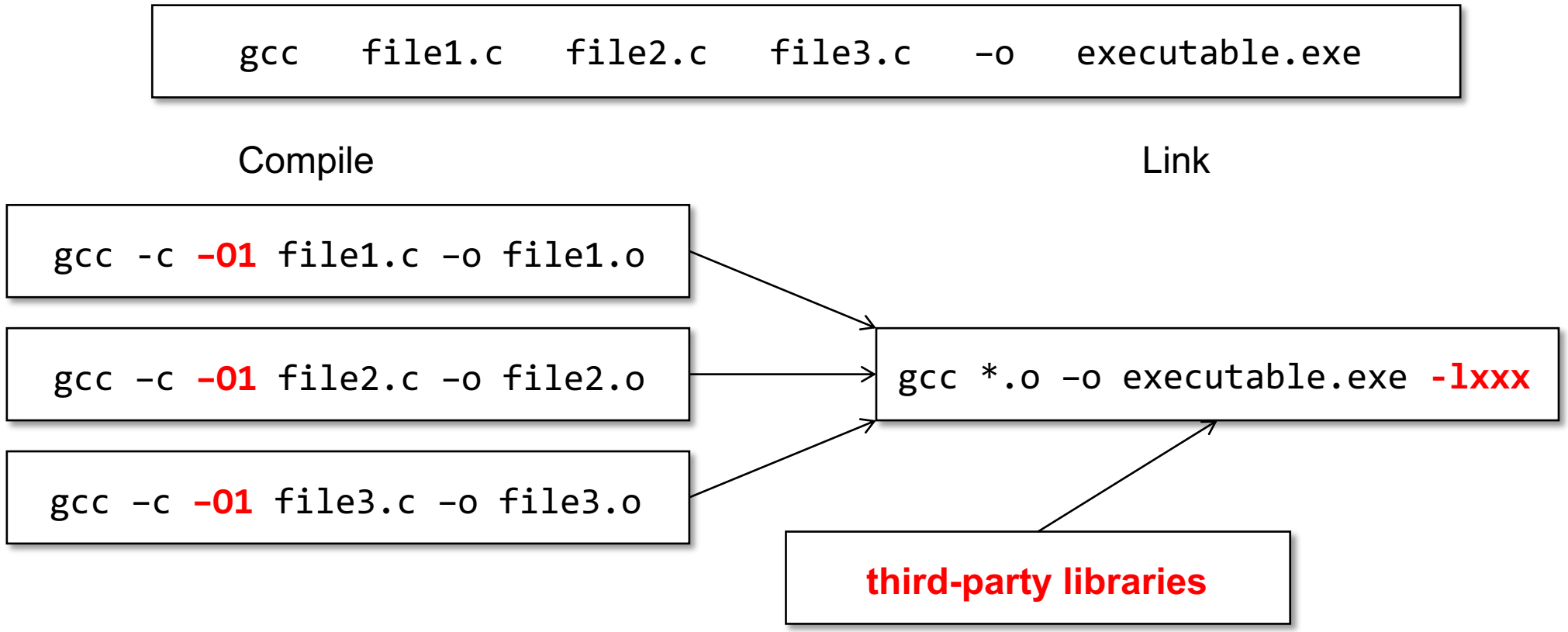
```
gcc  *.o  -o  executable.exe
```

For example:

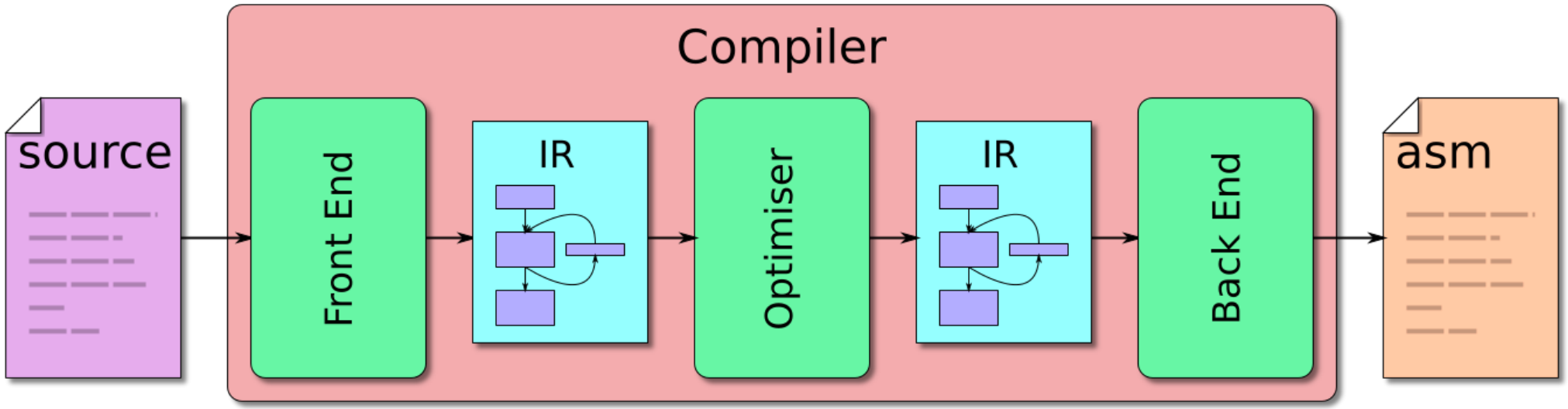
- file1.c declares and uses foo()
- file2.c defines foo()

How does a Compiler/Linker Work?

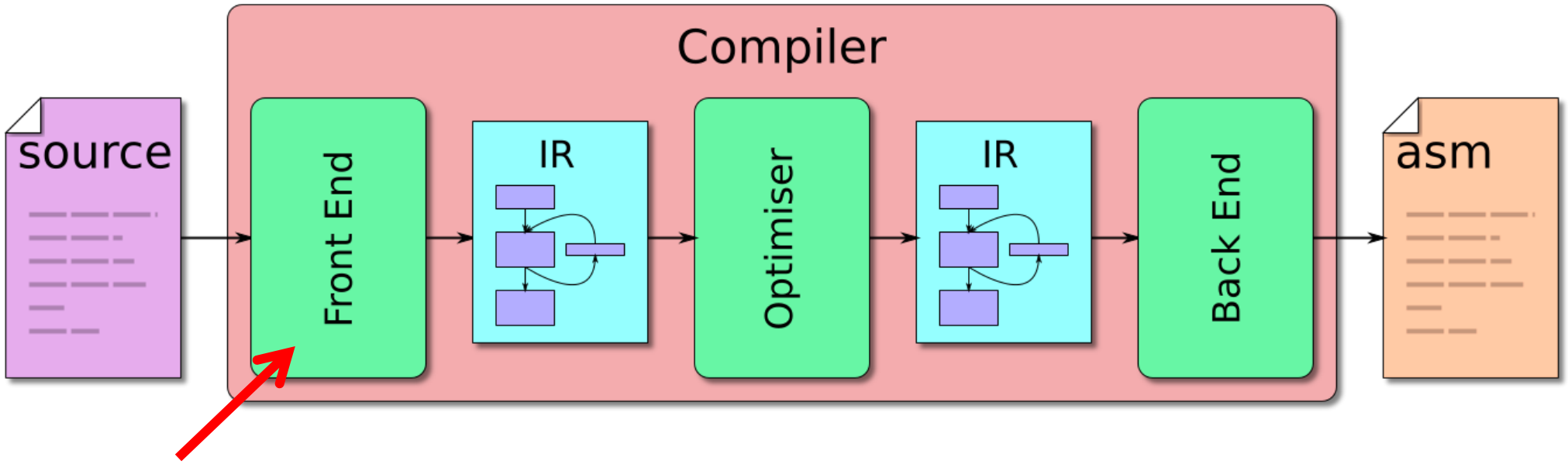
- Using gcc as the compiler to compile programs in C



How does a Compiler/Linker Work?

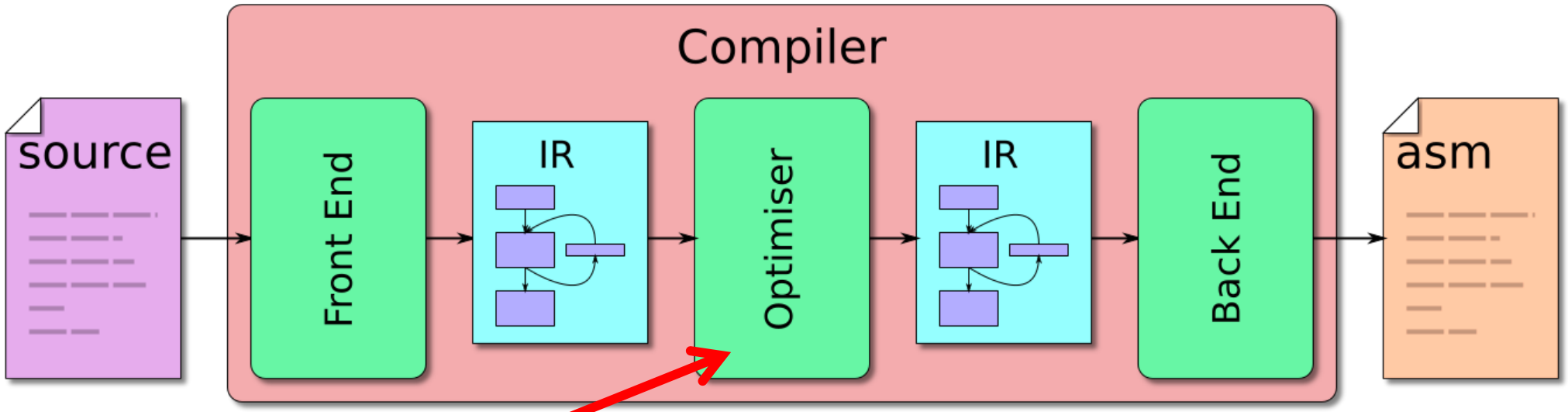


How does a Compiler/Linker Work?



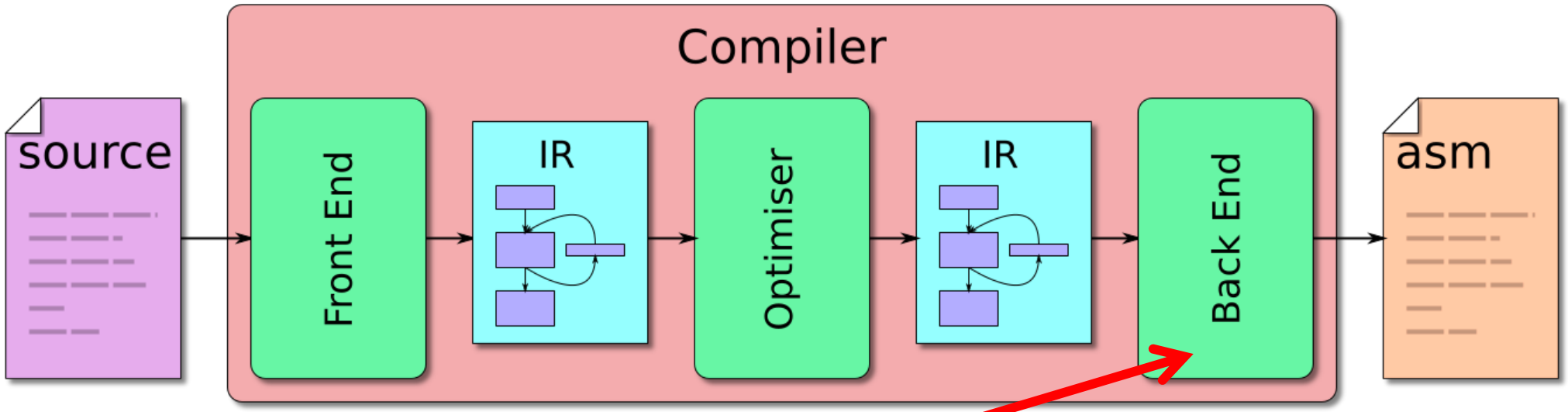
- **Front End**: read text in the source file, translate into IR

How does a Compiler/Linker Work?



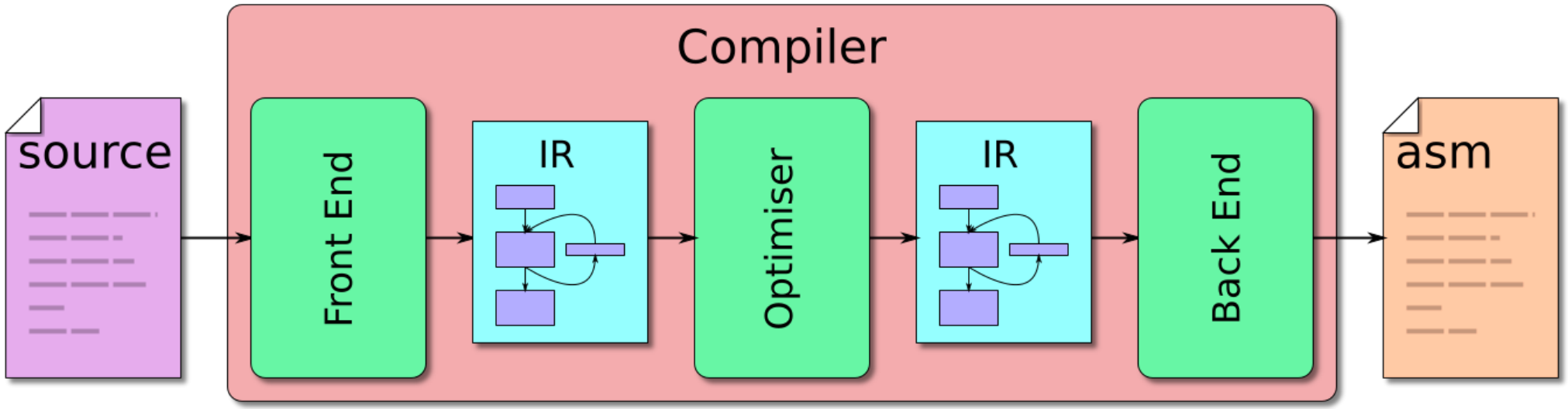
- **Front End**: read text in the source file, translate into IR
- **Middle End**: machine-independent optimization, IR → IR

How does a Compiler/Linker Work?

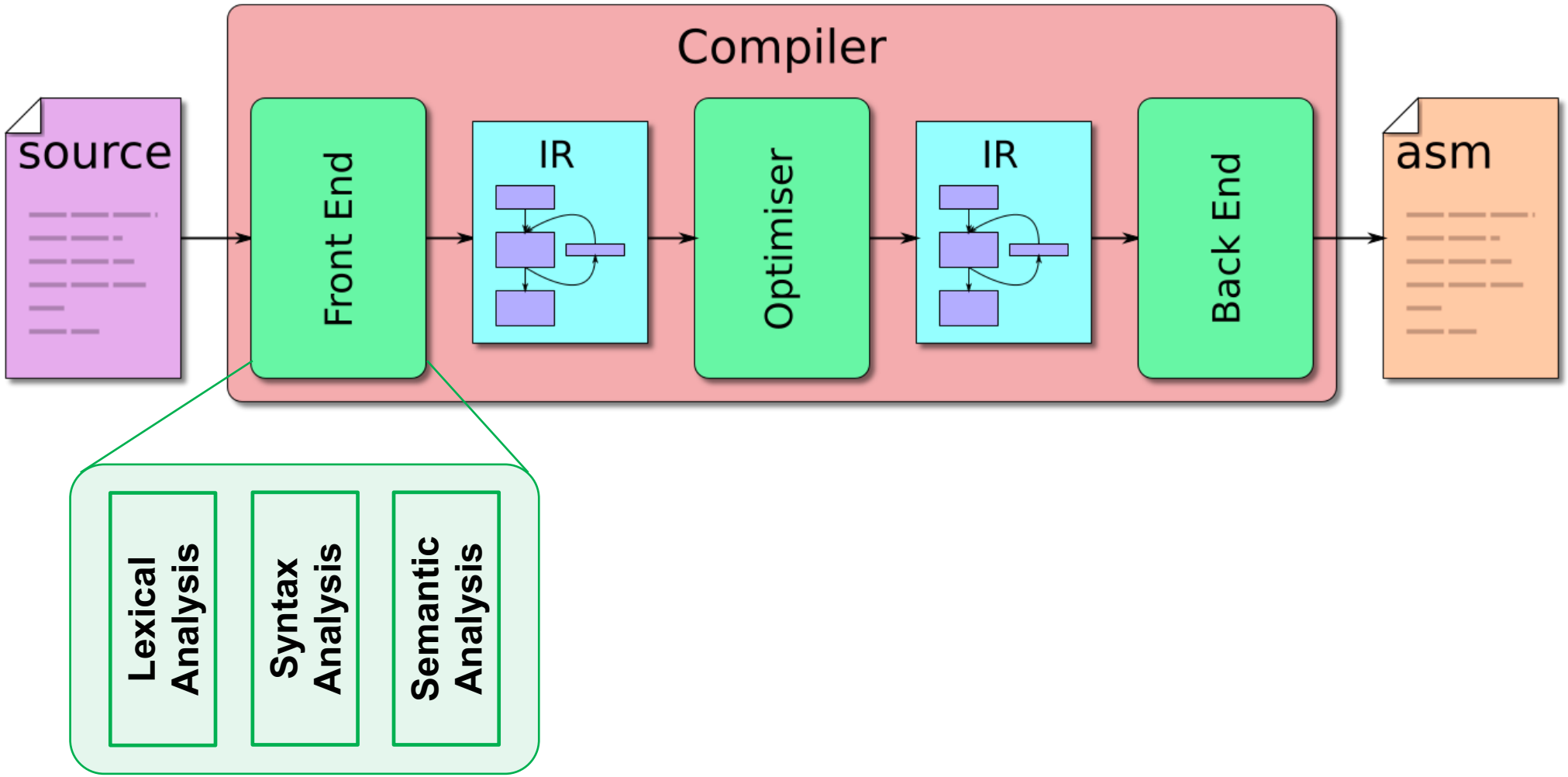


- **Front End**: read text in the source file, translate into IR
- **Middle End**: machine-independent optimization, IR → IR
- **Back End**: machine-dependent optimization and target generation

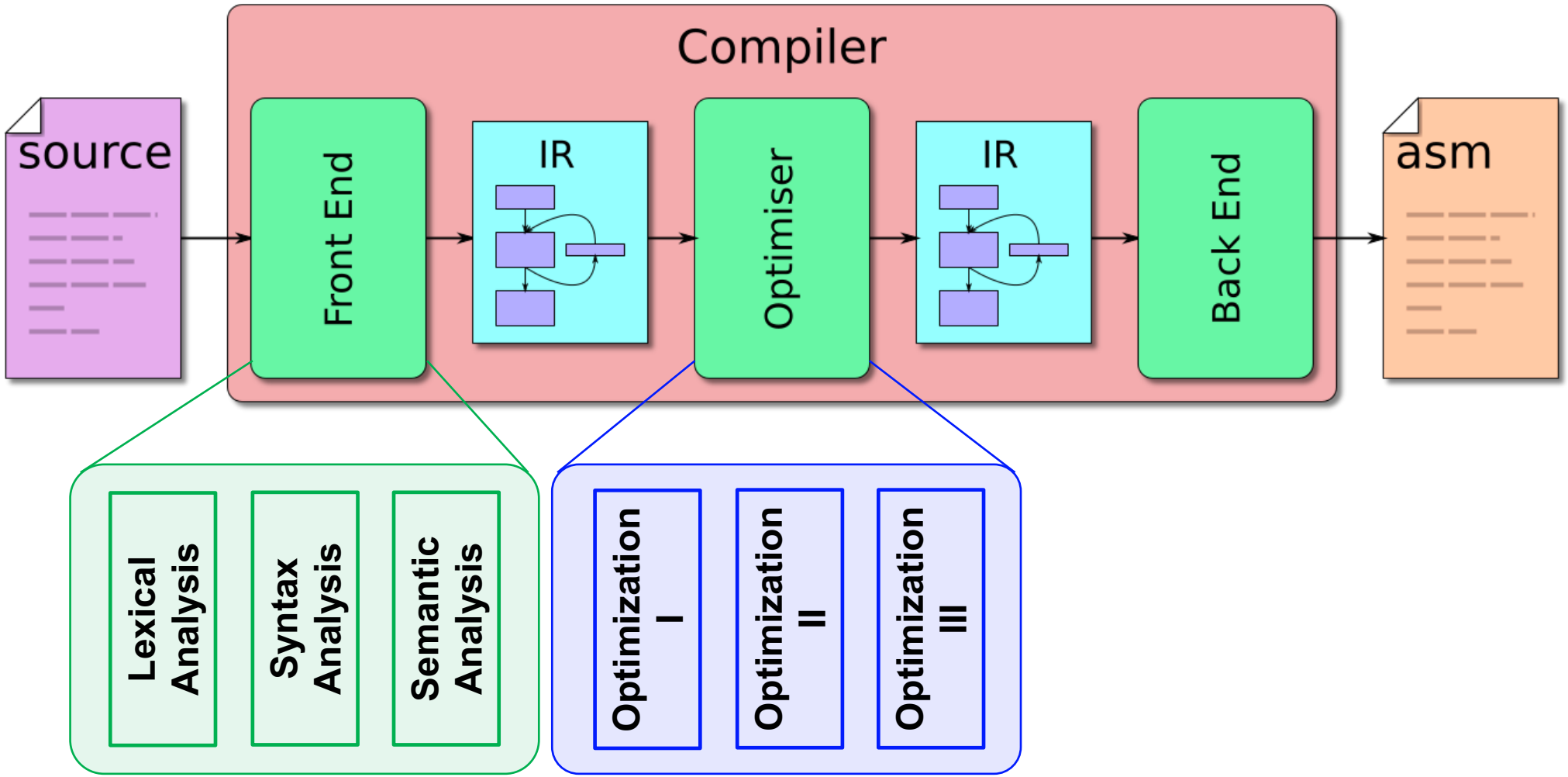
Breaking into Small Steps



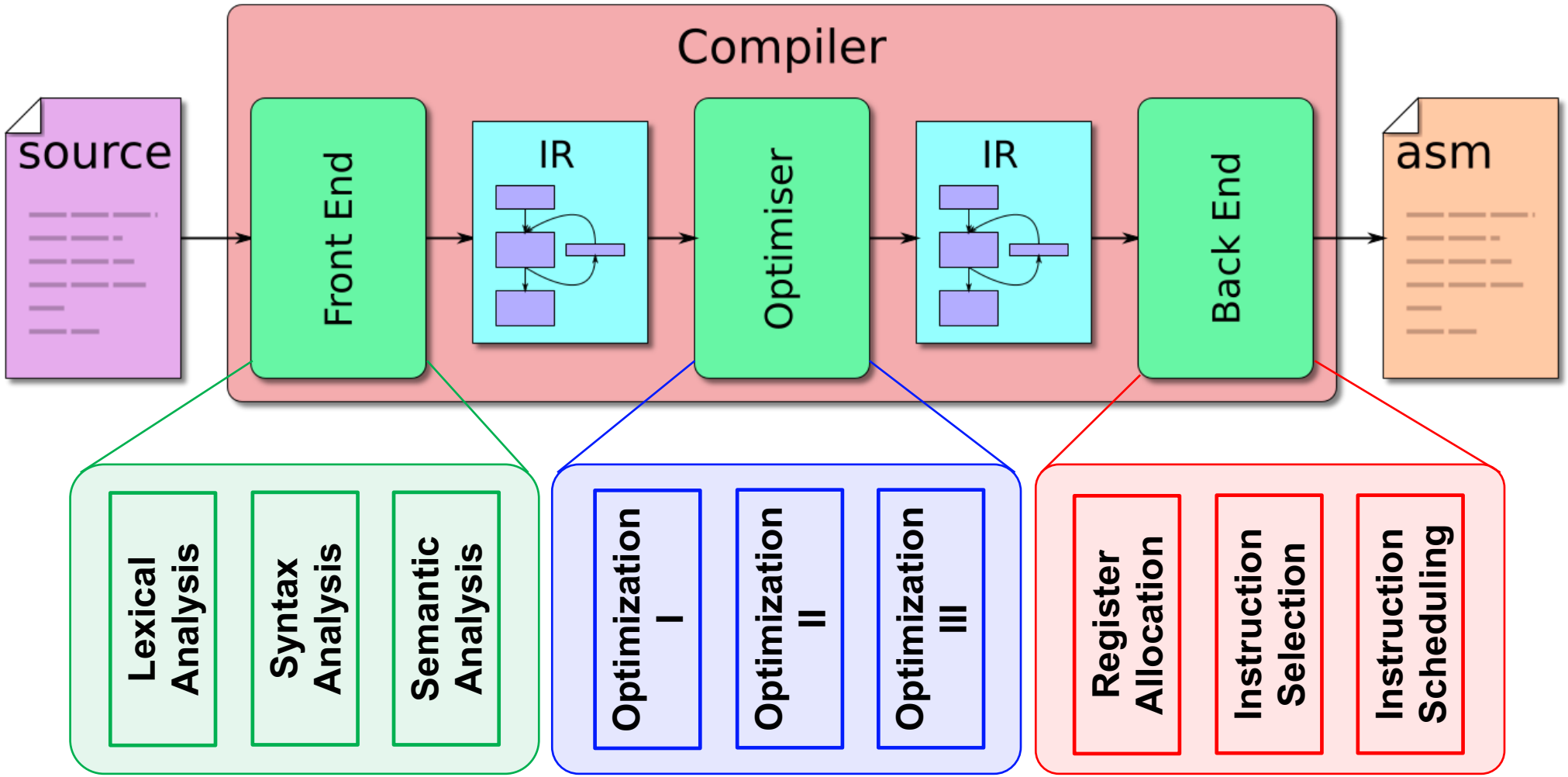
Breaking into Small Steps



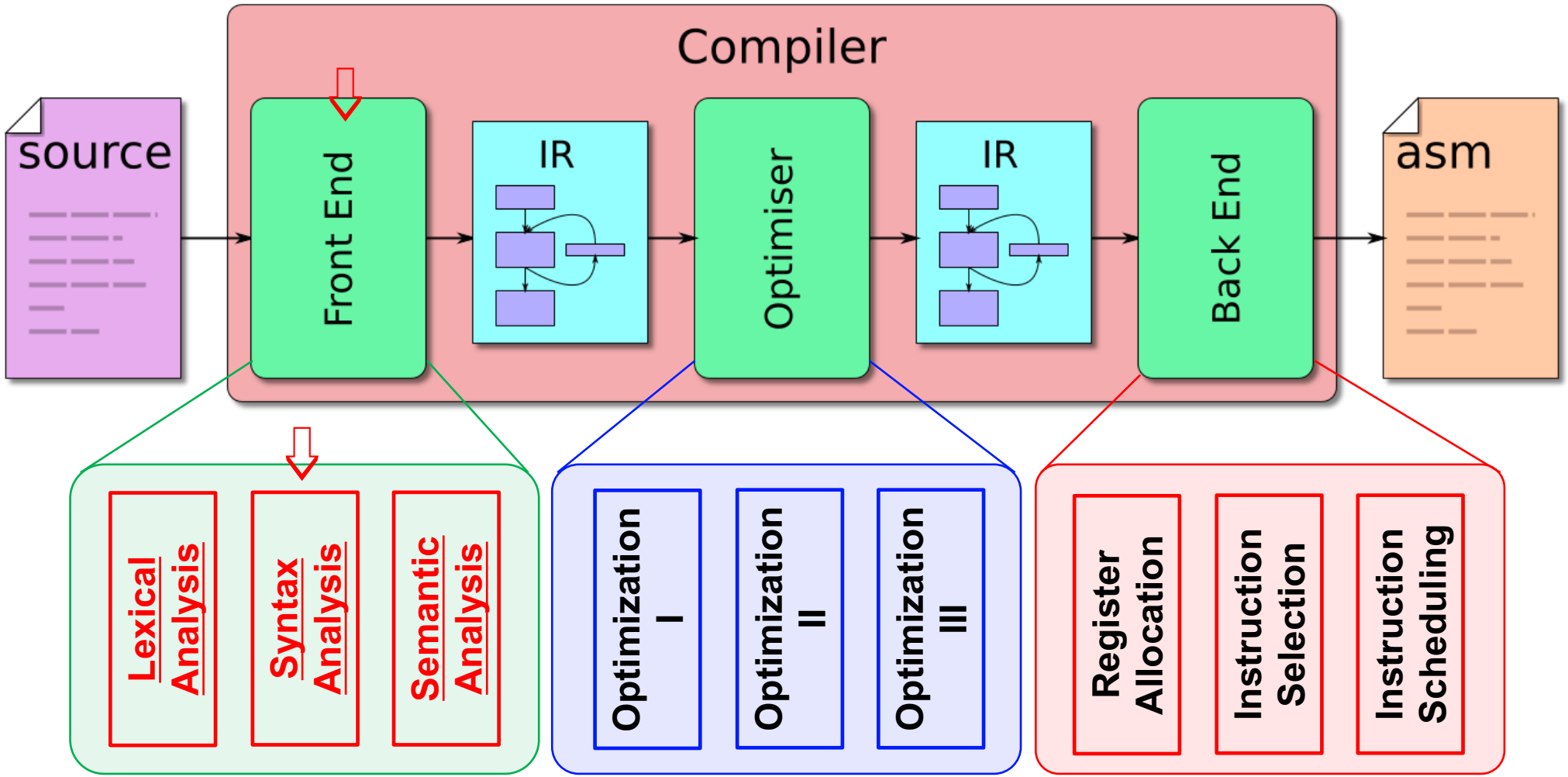
Breaking into Small Steps



Breaking into Small Steps



Front End


Compiler
source
Front End
IR
Optimiser
IR
Back End
asm
Lexical Analysis
Syntax Analysis
Semantic Analysis
Optimization I
Optimization II
Optimization III
Register Allocation
Instruction Selection
Instruction Scheduling

Front End

- Lexical Analysis → Syntax Analysis → Semantic Analysis

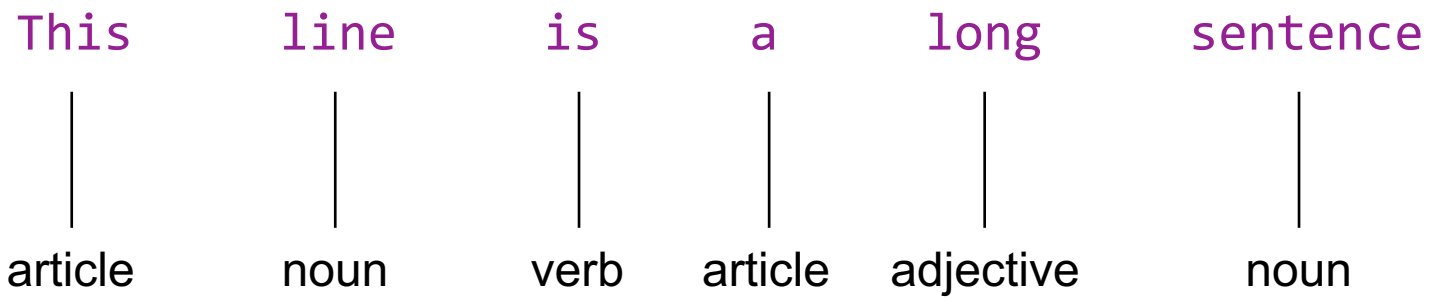
Front End

- Lexical Analysis → Syntax Analysis → Semantic Analysis

T h i s l i n e i s a l o n g s e n t e n c e

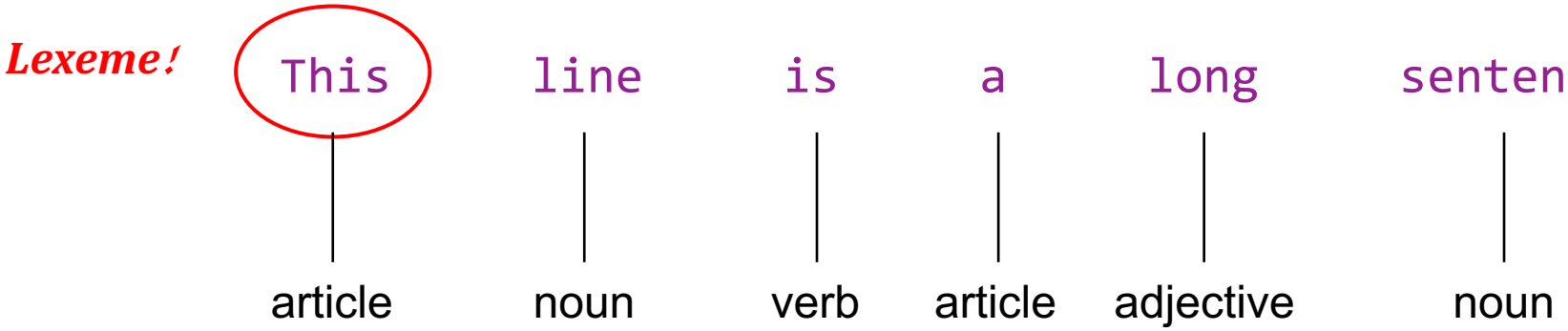
Front End

- Lexical Analysis → Syntax Analysis → Semantic Analysis



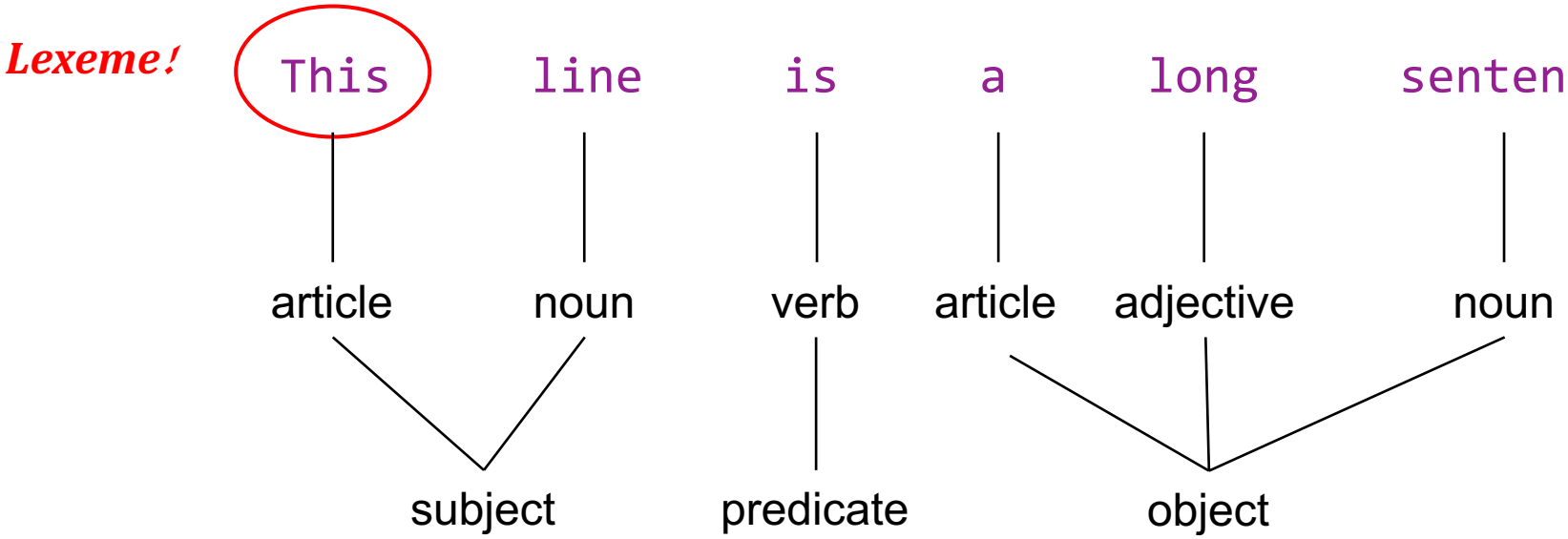
Front End

- Lexical Analysis → Syntax Analysis → Semantic Analysis



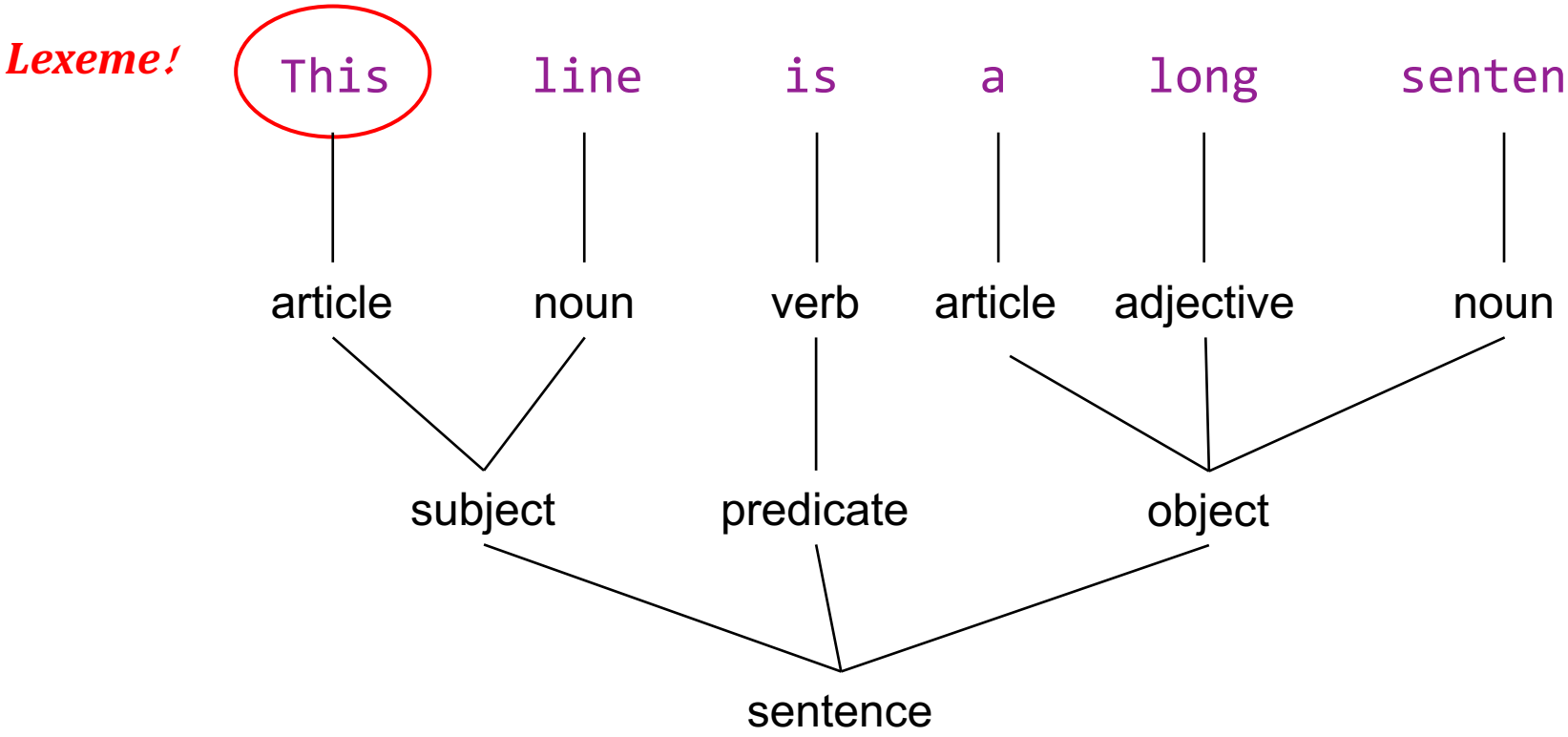
Front End

- Lexical Analysis → Syntax Analysis → Semantic Analysis



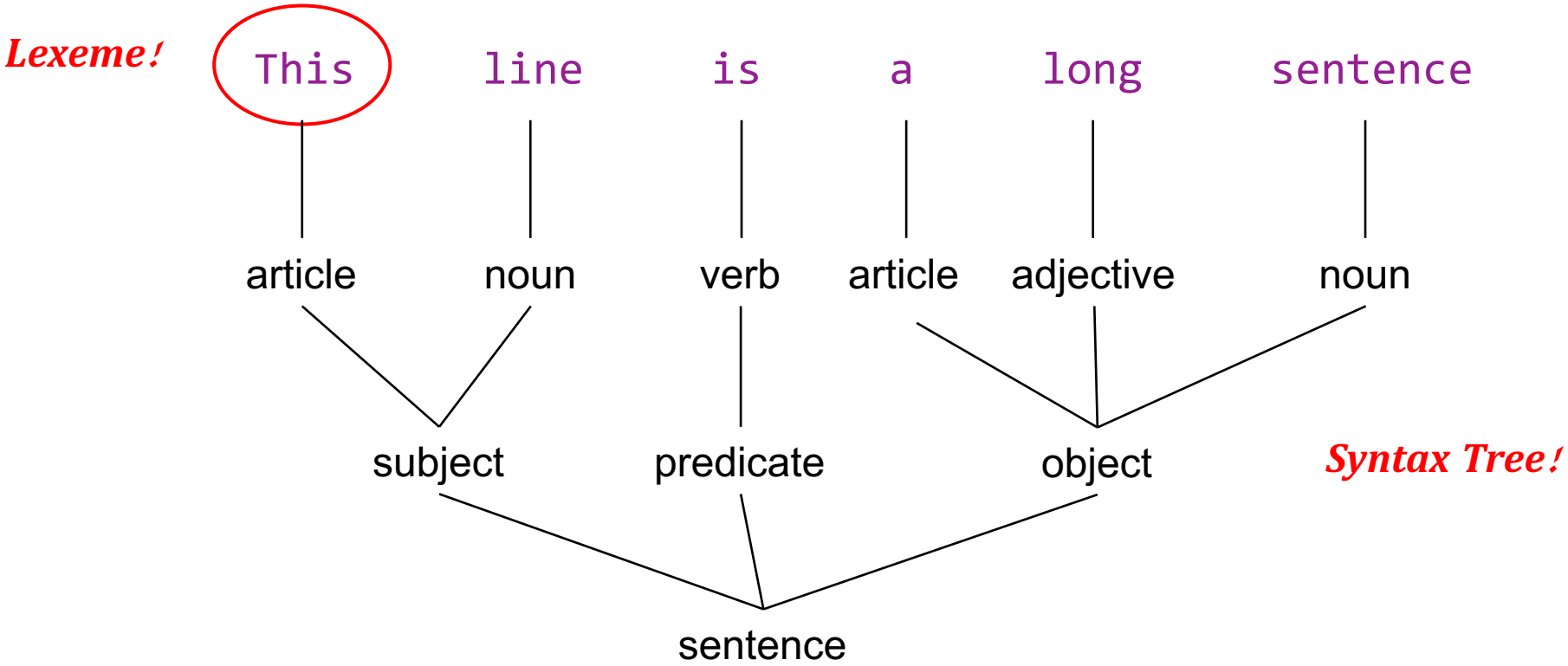
Front End

- Lexical Analysis → Syntax Analysis → Semantic Analysis



Front End

- Lexical Analysis → Syntax Analysis → Semantic Analysis



Front End: Lexical Analysis

- Find **lexemes** according to **patterns**, and create **tokens**
  - Lexeme – a character string
  - Pattern – regular expression (lexical errors if no patterns matched)
  - Token – <token-class, attribute>

Front End: Lexical Analysis

- Find **lexemes** according to **patterns**, and create **tokens**
  - Lexeme – a character string
  - Pattern – regular expression (lexical errors if no patterns matched)
  - Token – <token-class, attribute> **attribute distinguishes tokens in the same class**

Front End: Lexical Analysis

- Find **lexemes** according to **patterns**, and create **tokens**
  - Lexeme – a character string
  - Pattern – regular expression (lexical errors if no patterns matched)
  - Token – <token-class, attribute> **attribute distinguishes tokens in the same class**

```
p o s i t i o n  =  i n i t i a l  +  r a t e  *  6 0
```

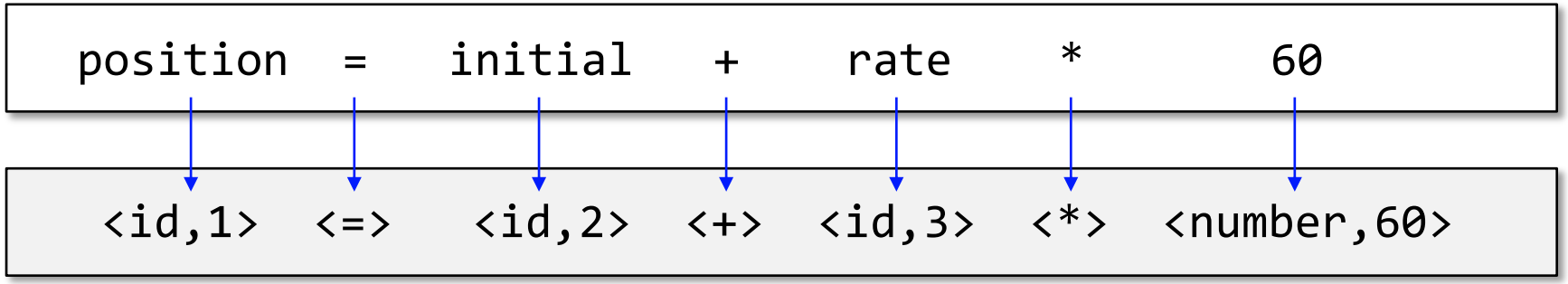
Front End: Lexical Analysis

- Find **lexemes** according to **patterns**, and create **tokens**
  - Lexeme – a character string
  - Pattern – regular expression (lexical errors if no patterns matched)
  - Token – <token-class, attribute> **attribute distinguishes tokens in the same class**

```
position  =  initial  +  rate  *  60
```

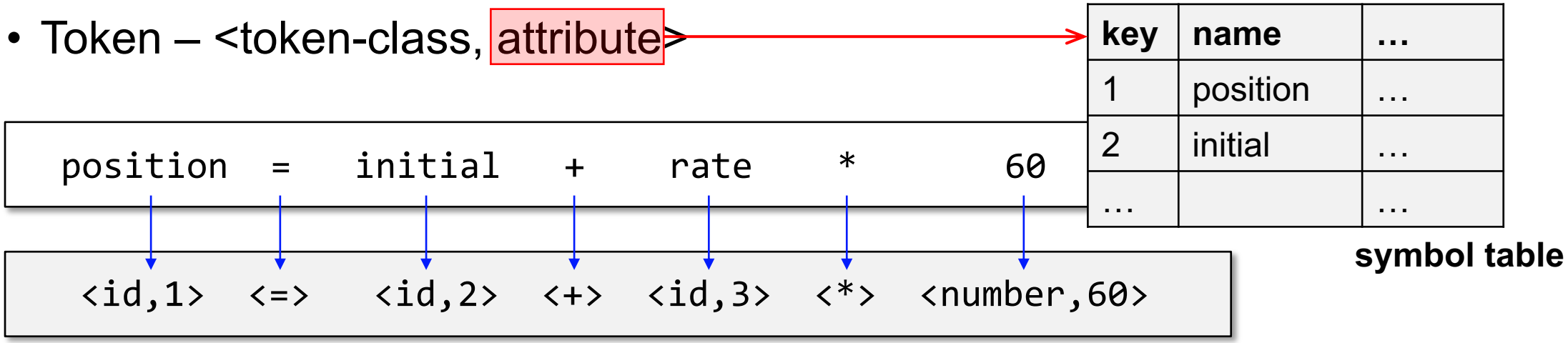
Front End: Lexical Analysis

- Find **lexemes** according to **patterns**, and create **tokens**
  - Lexeme – a character string
  - Pattern – regular expression (lexical errors if no patterns matched)
  - Token – <token-class, attribute> **attribute distinguishes tokens in the same class**



Front End: Lexical Analysis

- Find **lexemes** according to **patterns**, and create **tokens**
  - Lexeme – a character string
  - Pattern – regular expression (lexical errors if no patterns matched)
  - Token – <token-class, attribute>

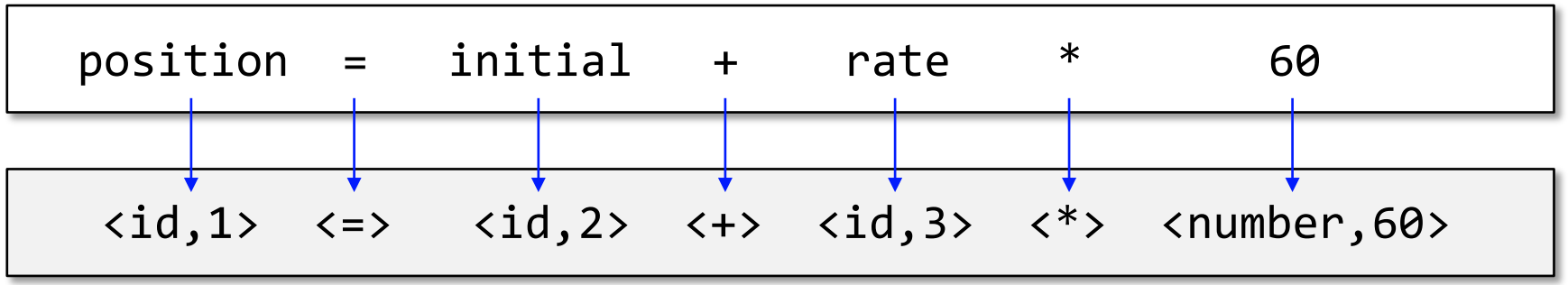


Front End: Syntax Analysis

- Create the (abstract) syntax tree (AST)

Front End: Syntax Analysis

- Create the (abstract) syntax tree (AST)

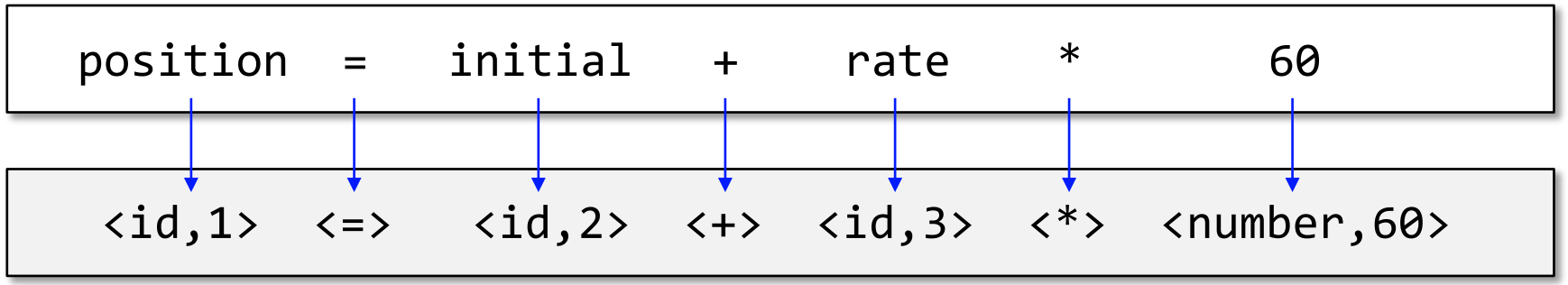


symbol table

key	name	…
1	position	…
2	initial	…
3	rate	…
…	…	…

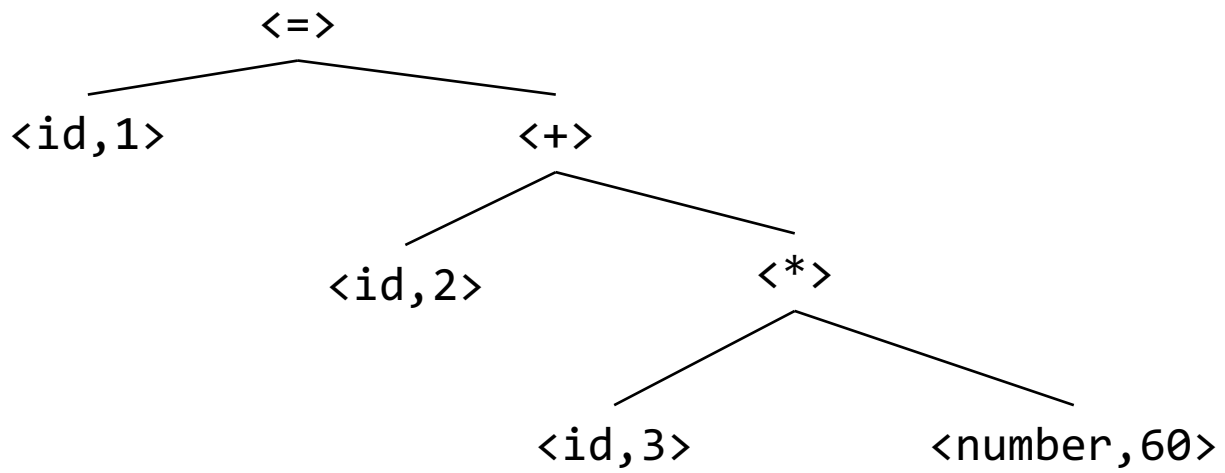
Front End: Syntax Analysis

- Create the (abstract) syntax tree (AST)



symbol table

key	name	…
1	position	…
2	initial	…
3	rate	…
…	…	…

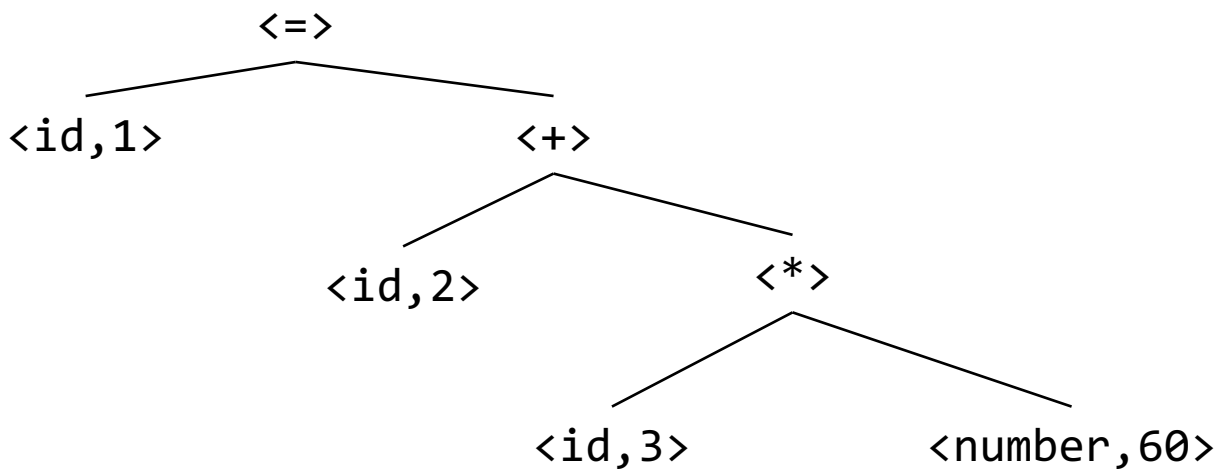


Front End: Semantic Analysis

- Passing syntax analysis does not mean the program is valid
- Semantic analysis checks correct meaning and decorates AST
  - types, scopes, …

Front End: Semantic Analysis

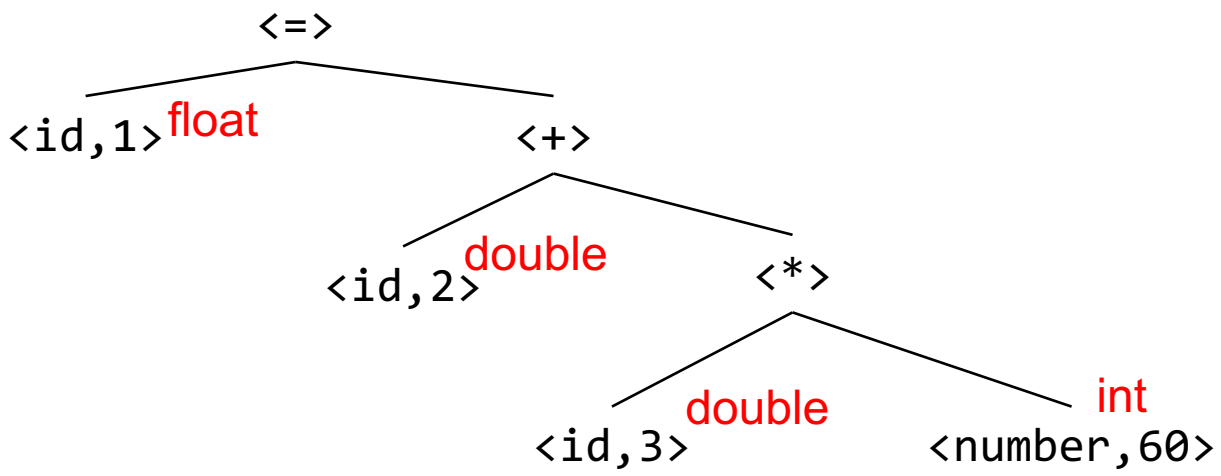
- Passing syntax analysis does not mean the program is valid
- Semantic analysis checks correct meaning and decorates AST
  - types, scopes, …



key	name	type
1	position	float
2	initial	double
3	rate	double
…	…	…

Front End: Semantic Analysis

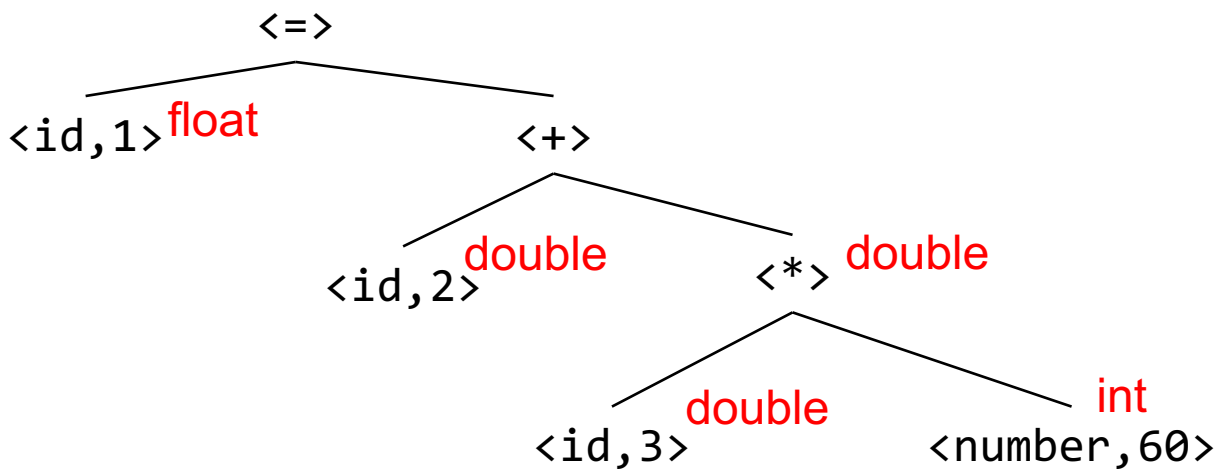
- Passing syntax analysis does not mean the program is valid
- Semantic analysis checks correct meaning and decorates AST
  - types, scopes, …



key	name	type
1	position	float
2	initial	double
3	rate	double
…	…	…

Front End: Semantic Analysis

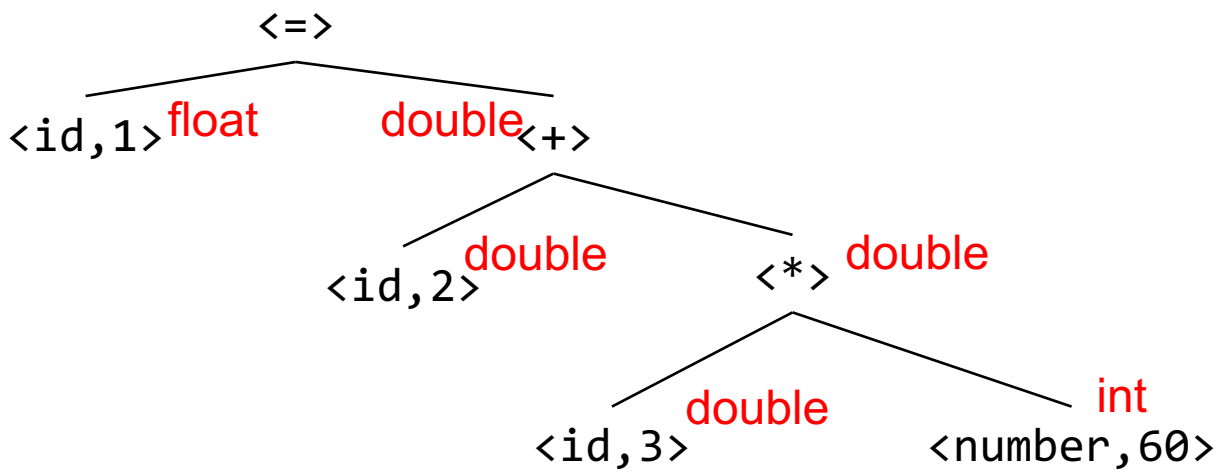
- Passing syntax analysis does not mean the program is valid
- Semantic analysis checks correct meaning and decorates AST
  - types, scopes, …



key	name	type
1	position	float
2	initial	double
3	rate	double
…	…	…

Front End: Semantic Analysis

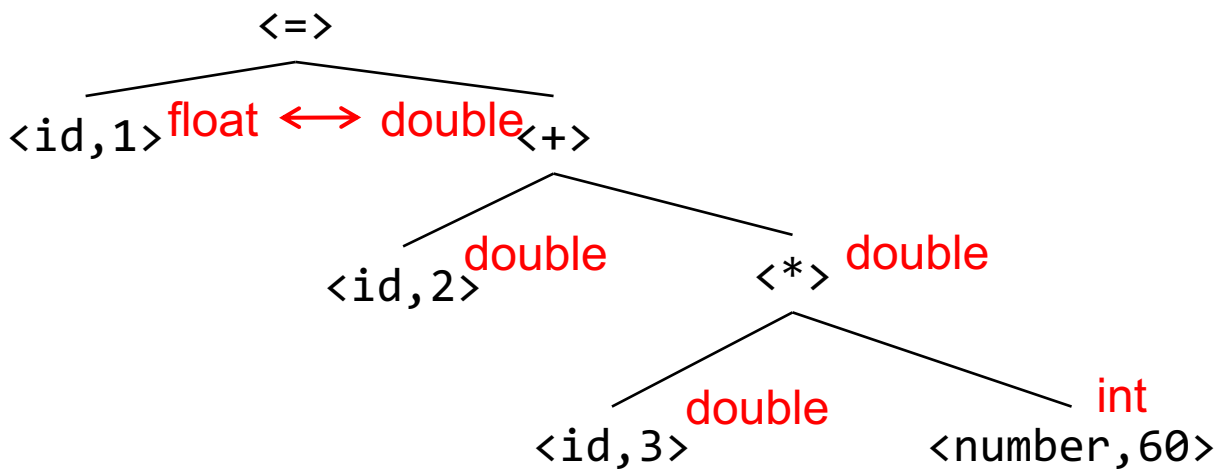
- Passing syntax analysis does not mean the program is valid
- Semantic analysis checks correct meaning and decorates AST
  - types, scopes, …



key	name	type
1	position	float
2	initial	double
3	rate	double
…	…	…

Front End: Semantic Analysis

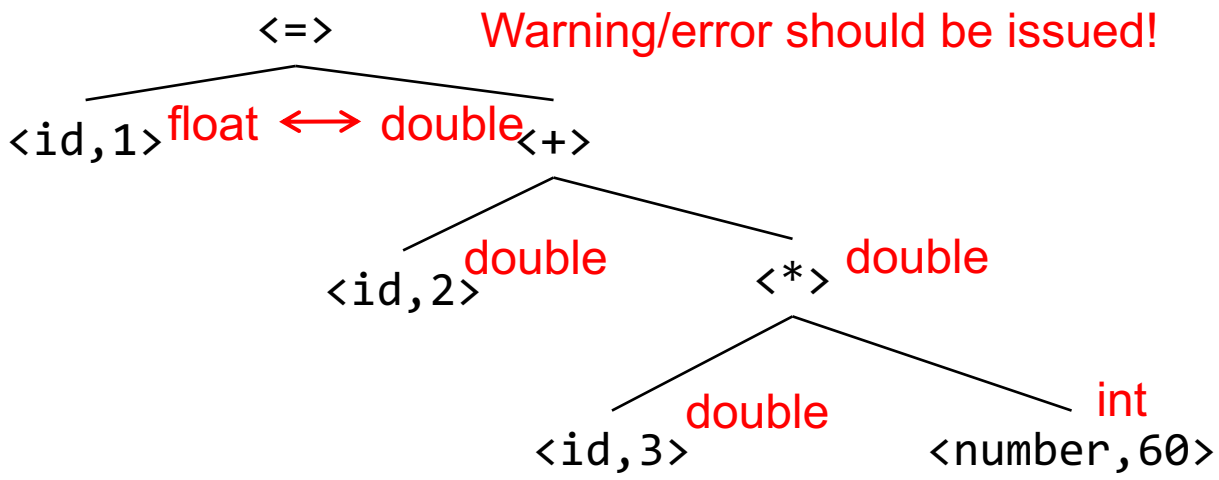
- Passing syntax analysis does not mean the program is valid
- Semantic analysis checks correct meaning and decorates AST
  - types, scopes, …



key	name	type
1	position	float
2	initial	double
3	rate	double
…	…	…

Front End: Semantic Analysis

- Passing syntax analysis does not mean the program is valid
- Semantic analysis checks correct meaning and decorates AST
  - types, scopes, …



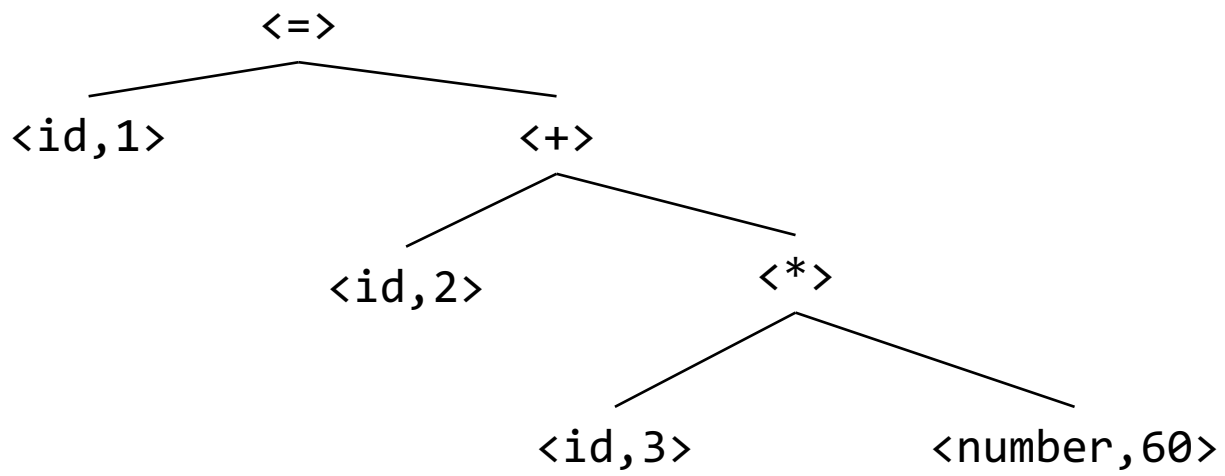
key	name	type
1	position	float
2	initial	double
3	rate	double
…	…	…

Front End: IR Generation

- Generate machine-independent intermediate representation (IR) based on the syntax tree

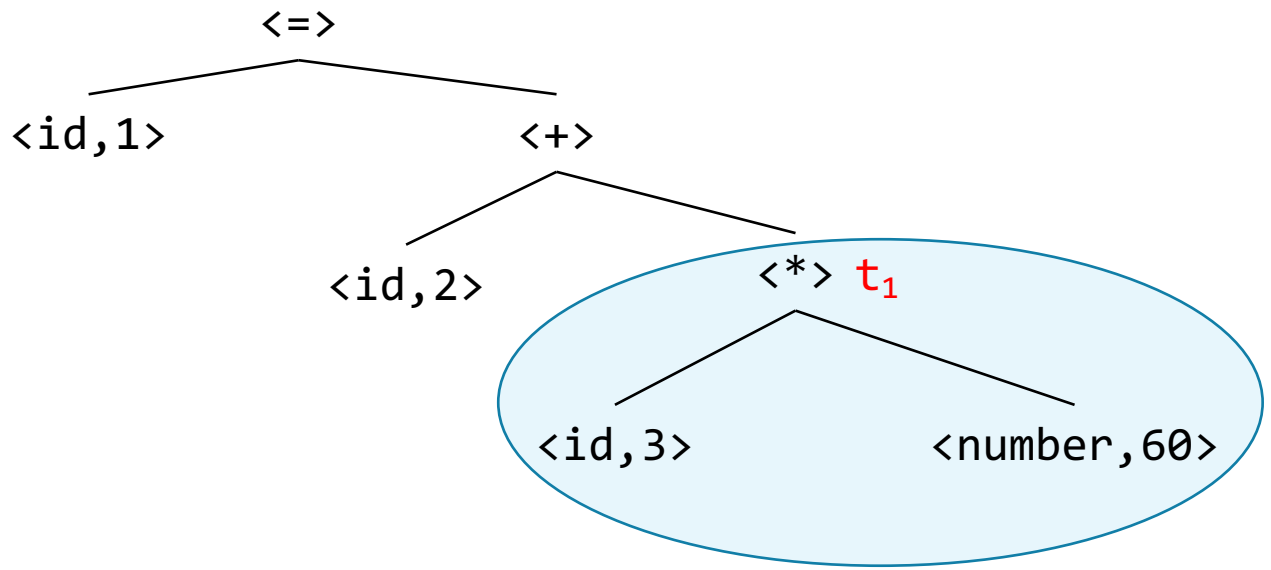
Front End: IR Generation

- Generate machine-independent intermediate representation (IR) based on the syntax tree



Front End: IR Generation

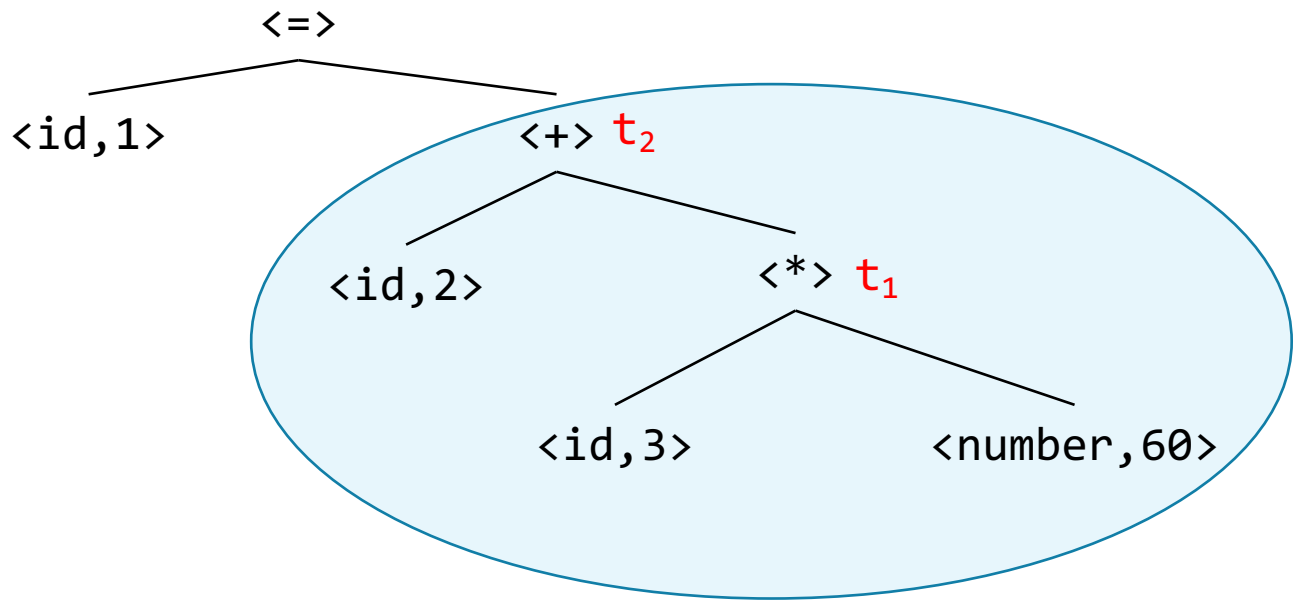
- Generate machine-independent intermediate representation (IR) based on the syntax tree



$t_1 = id_3 * 60$

Front End: IR Generation

- Generate machine-independent intermediate representation (IR) based on the syntax tree

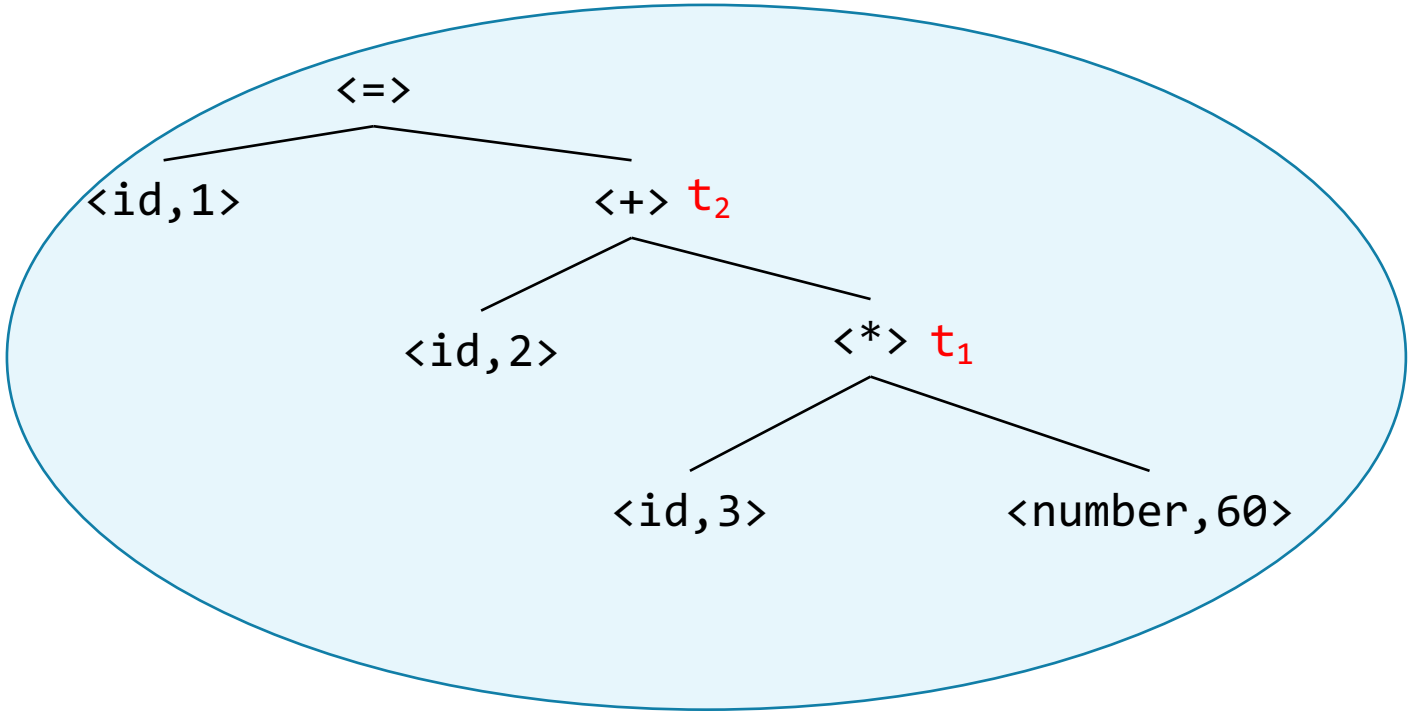


$t_1 = id_3 * 60$

$t_2 = id_2 + t_1$

Front End: IR Generation

- Generate machine-independent intermediate representation (IR) based on the syntax tree



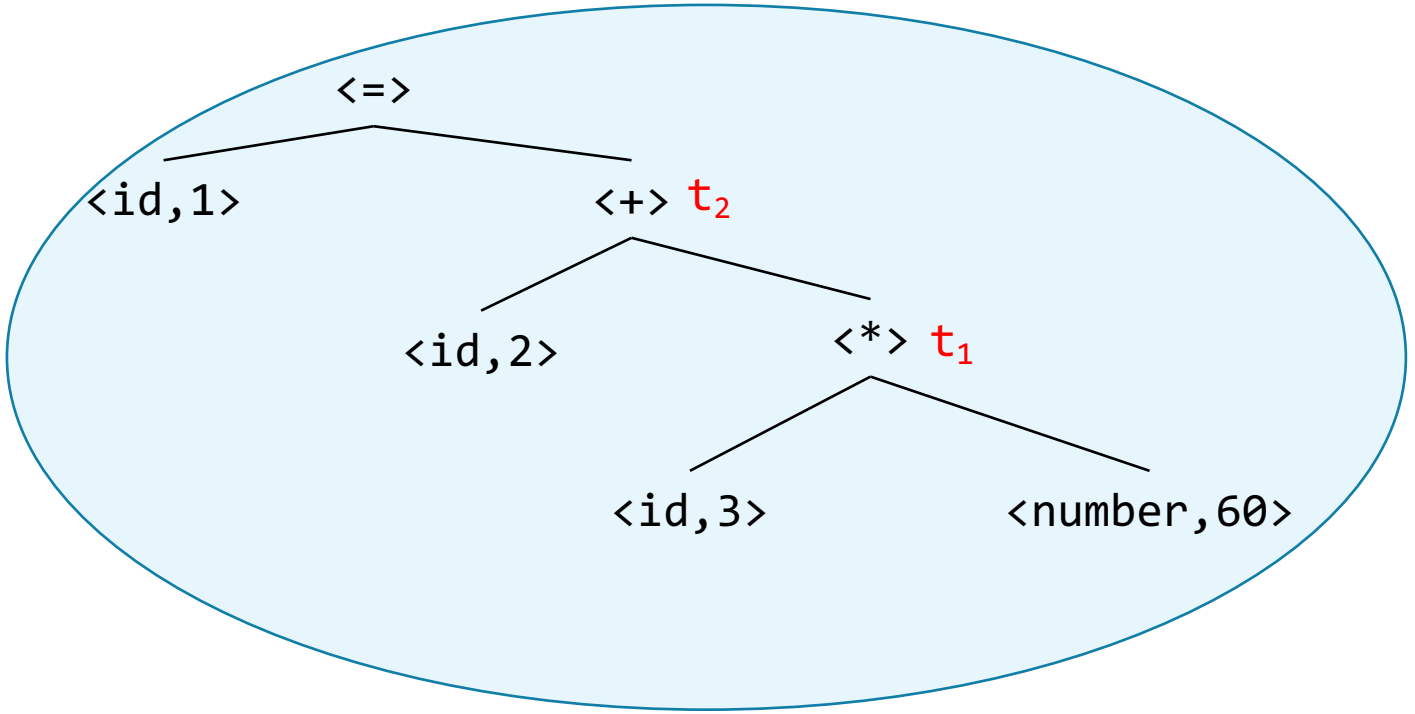
$t_1 = id_3 * 60$

$t_2 = id_2 + t_1$

$id_1 = t_2$

Front End: IR Generation

- Generate machine-independent intermediate representation (IR) based on the syntax tree



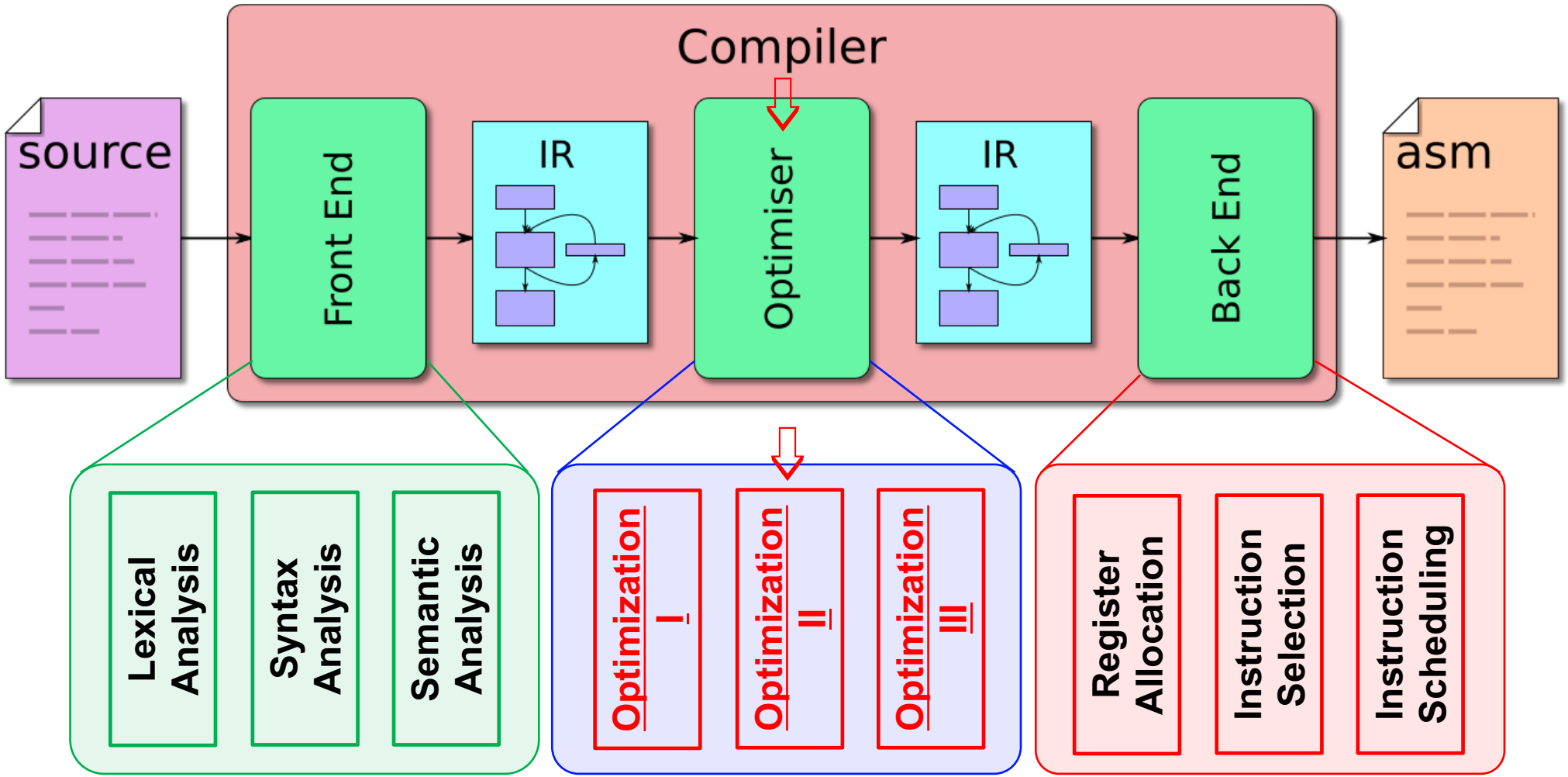
$t_1 = id_3 * 60$

$t_2 = id_2 + t_1$

$id_1 = t_2$

Three-address code

Middle End


Compiler
source
Front End
IR
Optimiser
IR
Back End
asm
Lexical Analysis
Syntax Analysis
Semantic Analysis
Optimization I
Optimization II
Optimization III
Register Allocation
Instruction Selection
Instruction Scheduling

Middle End: Optimizations

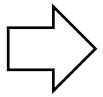
- Machine-independent optimizations, working on IR
  - **vs. source code**: IR provides standard form, easier to process
  - **vs. machine code**: IR provides machine-independent abstraction with source information (e.g., types via the symbol table)

Middle End: Optimizations

- Machine-independent optimizations, working on IR
  - **vs. source code**: IR provides standard form, easier to process
  - **vs. machine code**: IR provides machine-independent abstraction with source information (e.g., types via the symbol table)

```
t1  = id3  *  60
t2  = id2  +  t1
id1 = t2
```

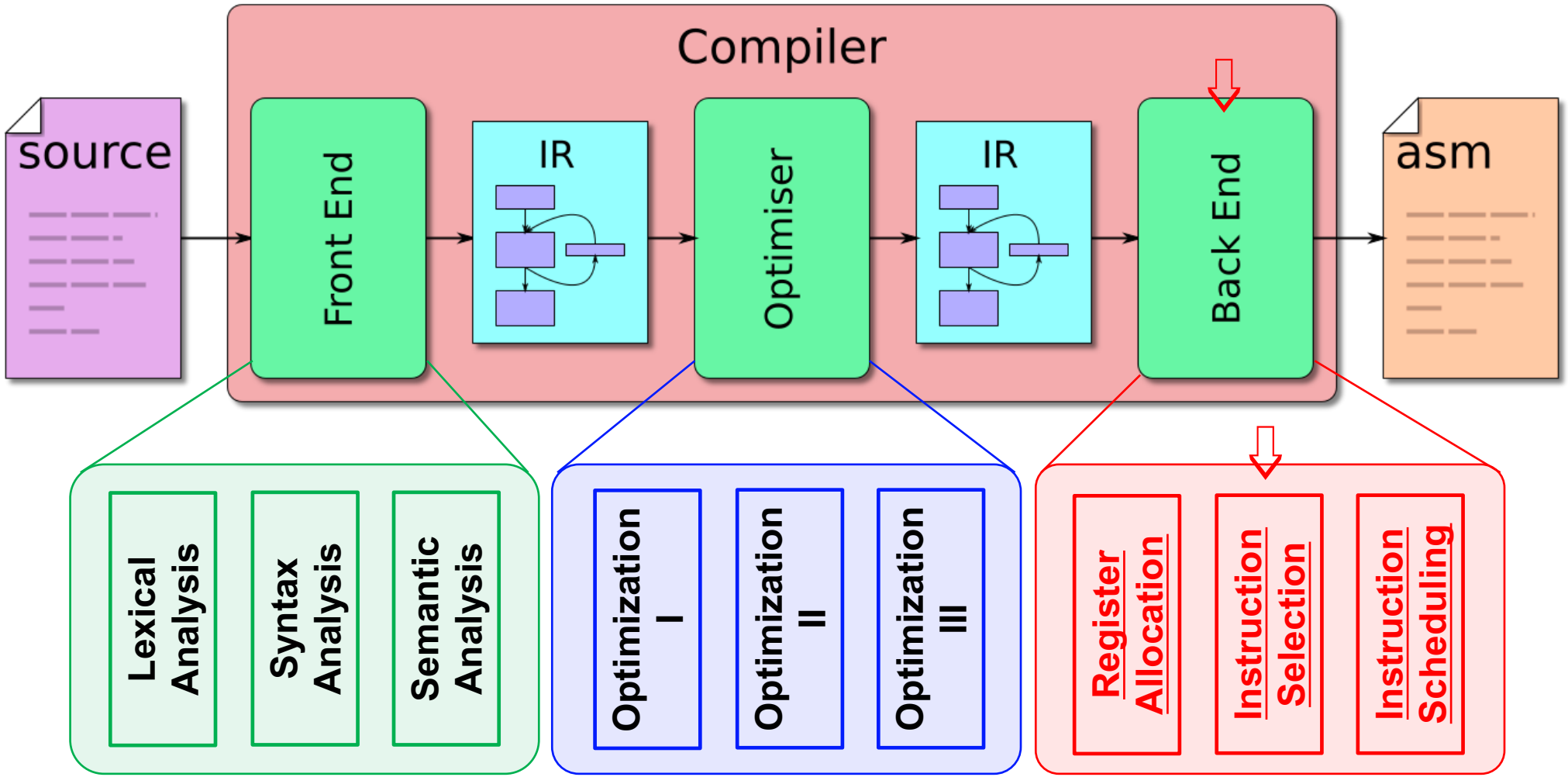
Three-address code



```
t1  = id3  *  60
id1 = id2  +  t1
```

Optimized code

Breaking into Small Steps



Back End: Instruction Selection

- Translate IR to machine code
- Perform machine-dependent optimization

Back End: Instruction Selection

- Translate IR to machine code
- Perform machine-dependent optimization

```
LD   R0, a              // R0 = a
Add  R0, R0, #1         // R0 = R0 + 1
ST   a,  R0             // a = R0
```

Possible machine code for a = a + 1

Back End: Instruction Selection

- Translate IR to machine code
- Perform machine-dependent optimization

```
LD   R0, a              // R0 = a
Add  R0, R0, #1         // R0 = R0 + 1
ST   a,  R0             // a = R0
```

Possible machine code for a = a + 1

```
INC a          // a = a + 1
```

More compact machine code

Back End: Register Allocation

- Accessing registers is faster than accessing memory

Back End: Register Allocation

- Accessing registers is faster than accessing memory
- **The number of registers is limited**

Back End: Register Allocation

- Accessing registers is faster than accessing memory
- The number of registers is limited
- **Allocating reasonable # registers for machine code generation**

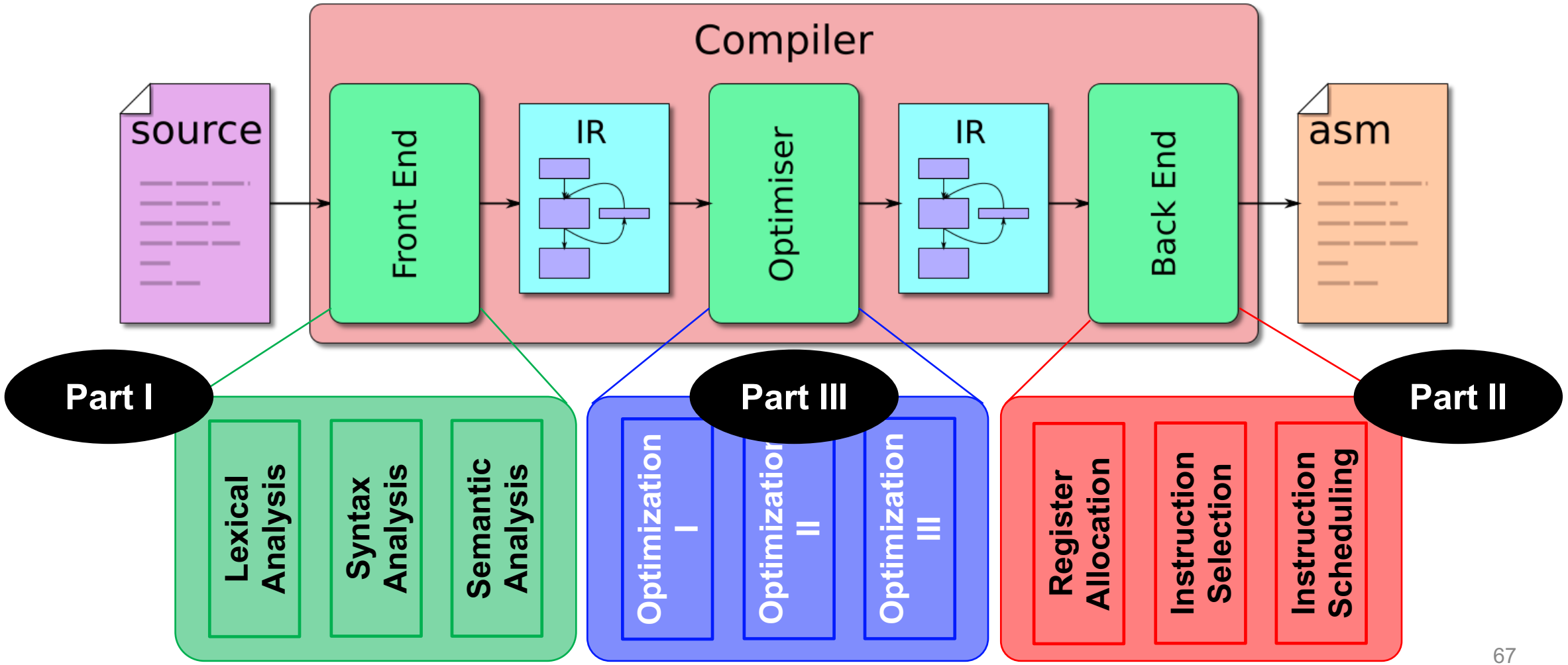
Back End: Instruction Scheduling

- Target machines often provide hardware resources for instruction-level parallelism

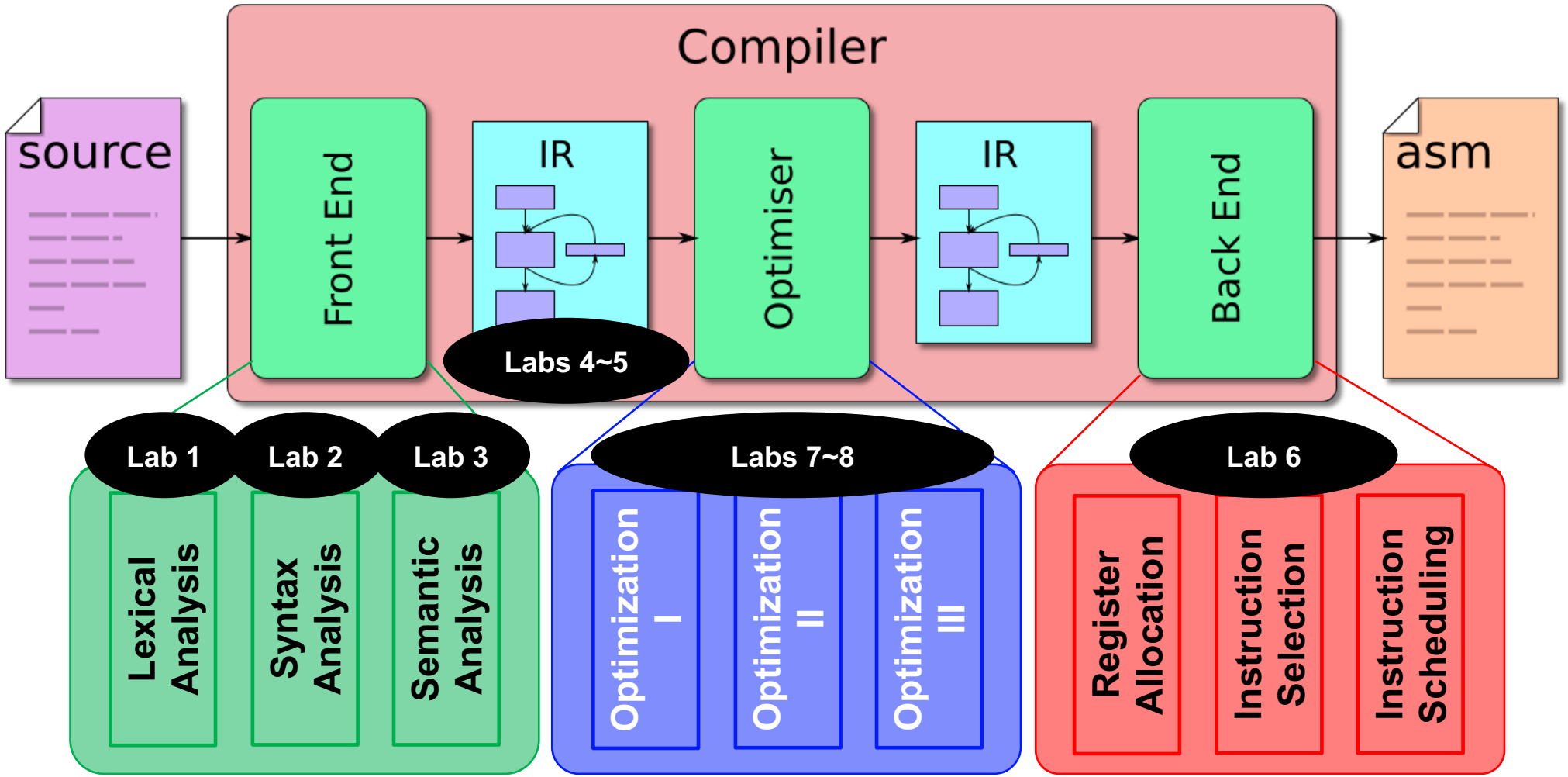
Back End: Instruction Scheduling

- Target machines often provide hardware resources for instruction-level parallelism
- Generating machine code to take advantage of such parallelism

Course Structure



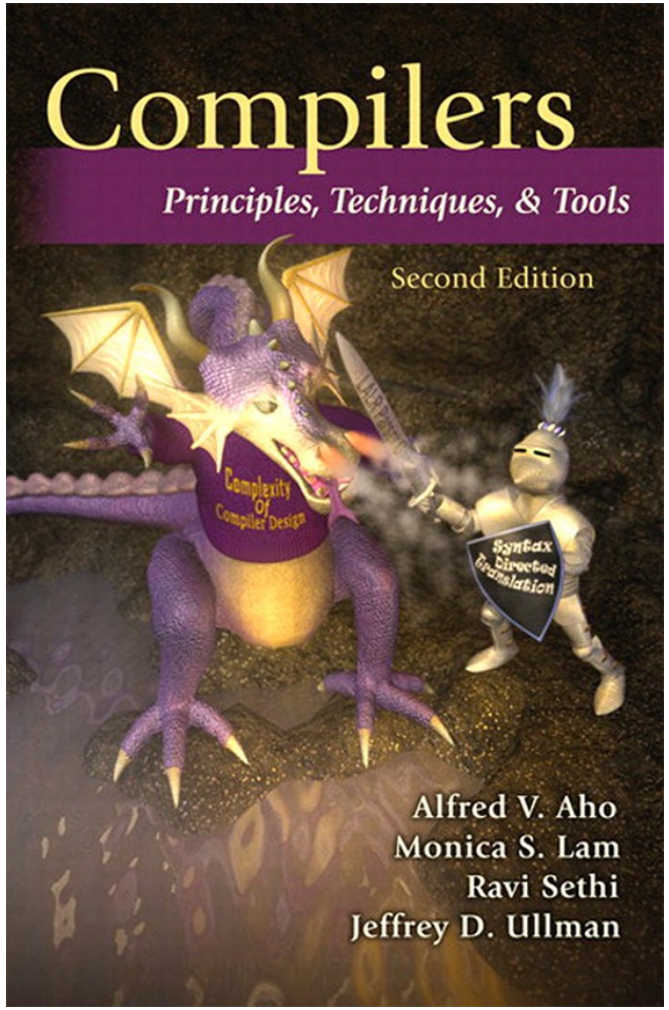
Labs


Compiler
source
Front End
IR
Optimiser
IR
Back End
asm
Labs 4~5
Lab 1
Lab 2
Lab 3
Lexical Analysis
Syntax Analysis
Semantic Analysis
Labs 7~8
Optimization I
Optimization II
Optimization III
Lab 6
Register Allocation
Instruction Selection
Instruction Scheduling

Course Homepage

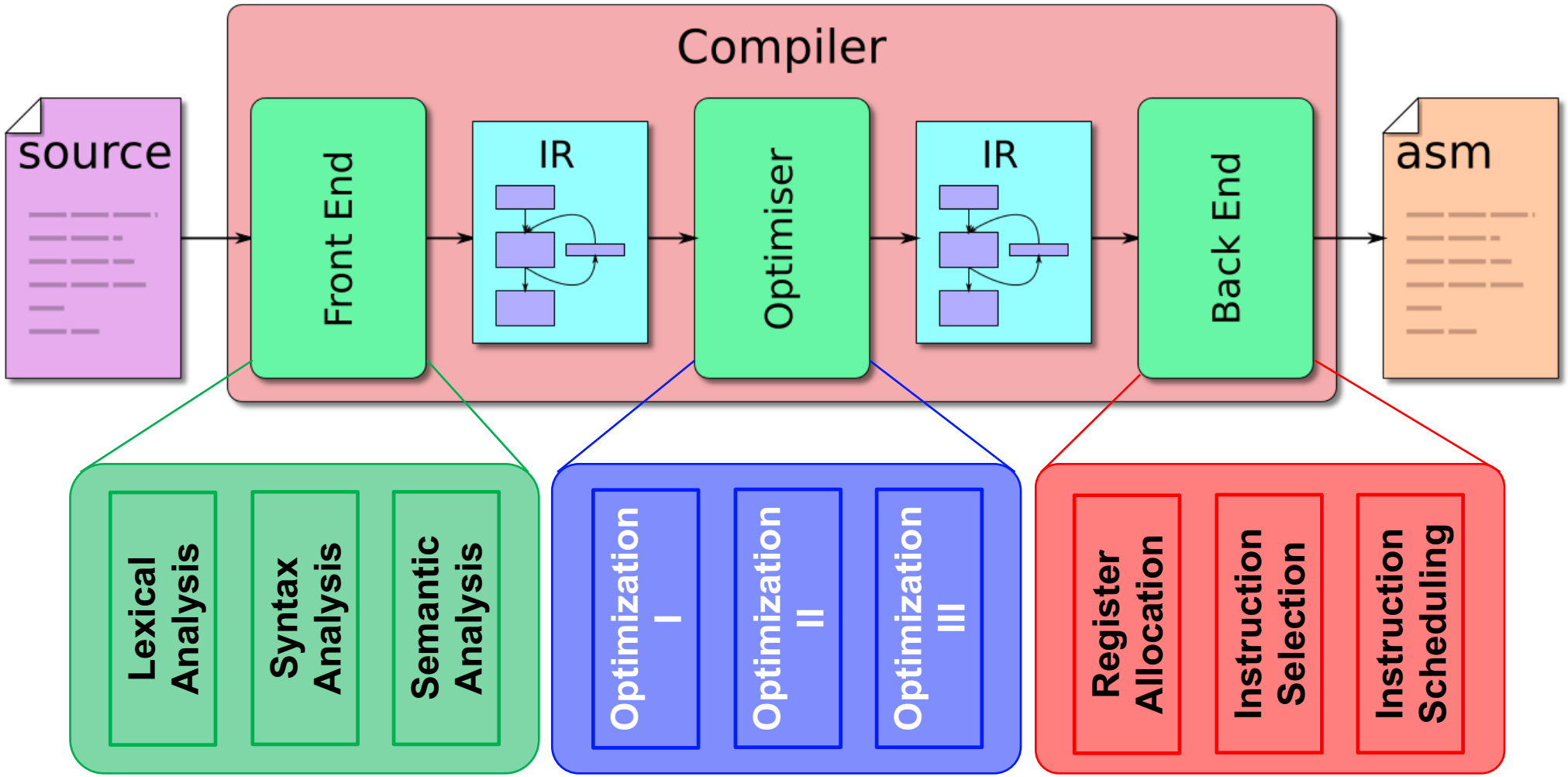
https://qingkaishi.github.io/Compilers.html

- Find additional info about the course in the course homepage
  - QQ Group
    - 1033582067
  - Grading Scheme
    - Assignments (0%)
    - Labs (70%)
    - Final Exam (30%)
  - Slides …



PART II: Tools

Tools


Compiler
source
Front End
IR
Optimiser
IR
Back End
asm
Lexical Analysis
Syntax Analysis
Semantic Analysis
Optimization I
Optimization II
Optimization III
Register Allocation
Instruction Selection
Instruction Scheduling

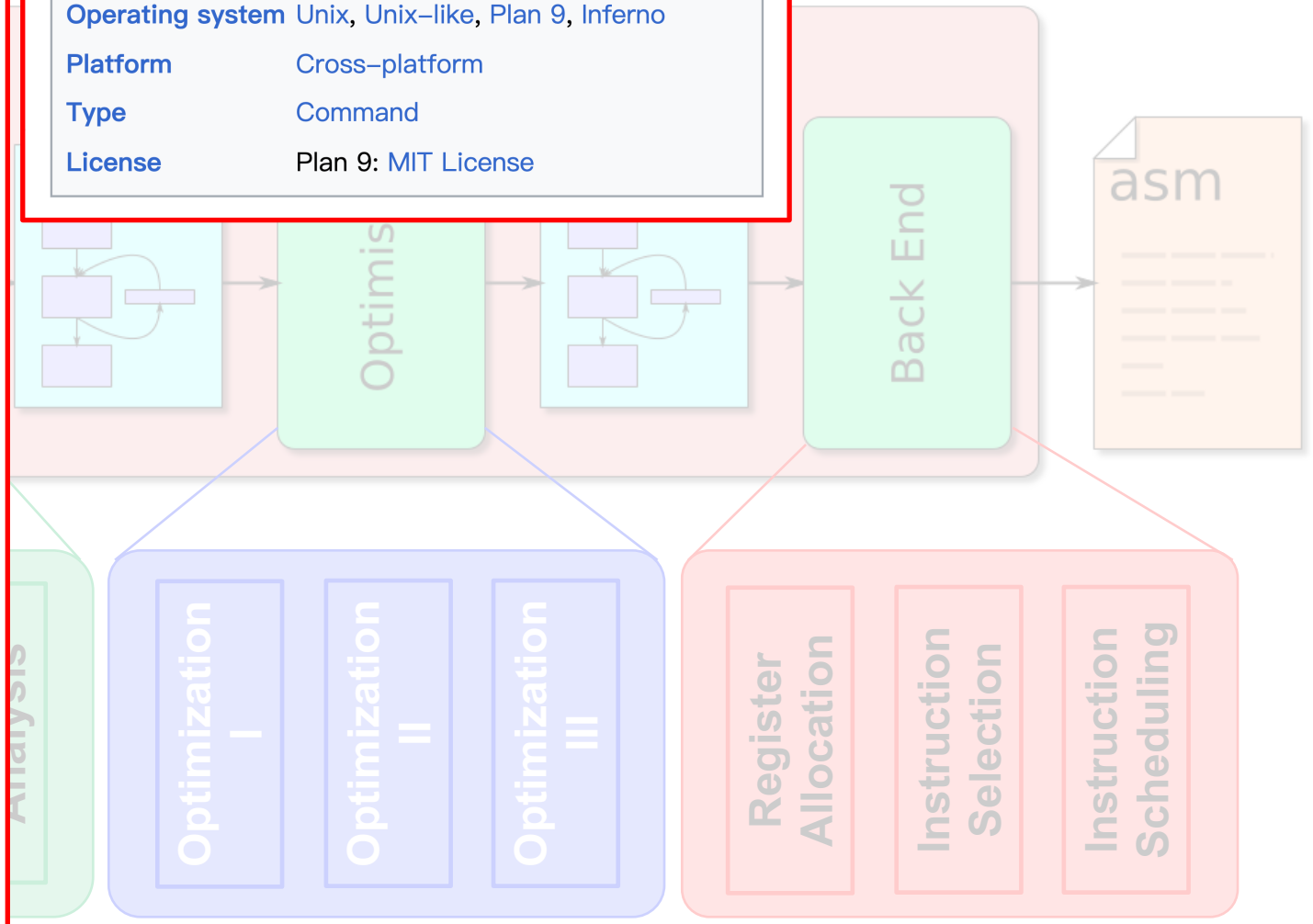
Tools

Yacc

Original author(s)	Stephen C. Johnson
Repository	www.tuhs.org/cgi-bin/utree.pl?file=V6%2Fusr%2Fsource%2Fyacc
Written in	C
Operating system	Unix, Unix-like, Plan 9, Inferno
Platform	Cross-platform
Type	Command
License	Plan 9: MIT License

GNU Bison

Original author(s)	Robert Corbett
Developer(s)	The GNU Project
Initial release	June 1985; 38 years ago[1]
Stable release	3.8.2[2] / 25 September 2021
Repository	git.savannah.gnu.org/cgit/bison.git
Written in	C and m4
Operating system	Unix-like
Type	Parser generator
License	GPL
Website	www.gnu.org/software/bison/



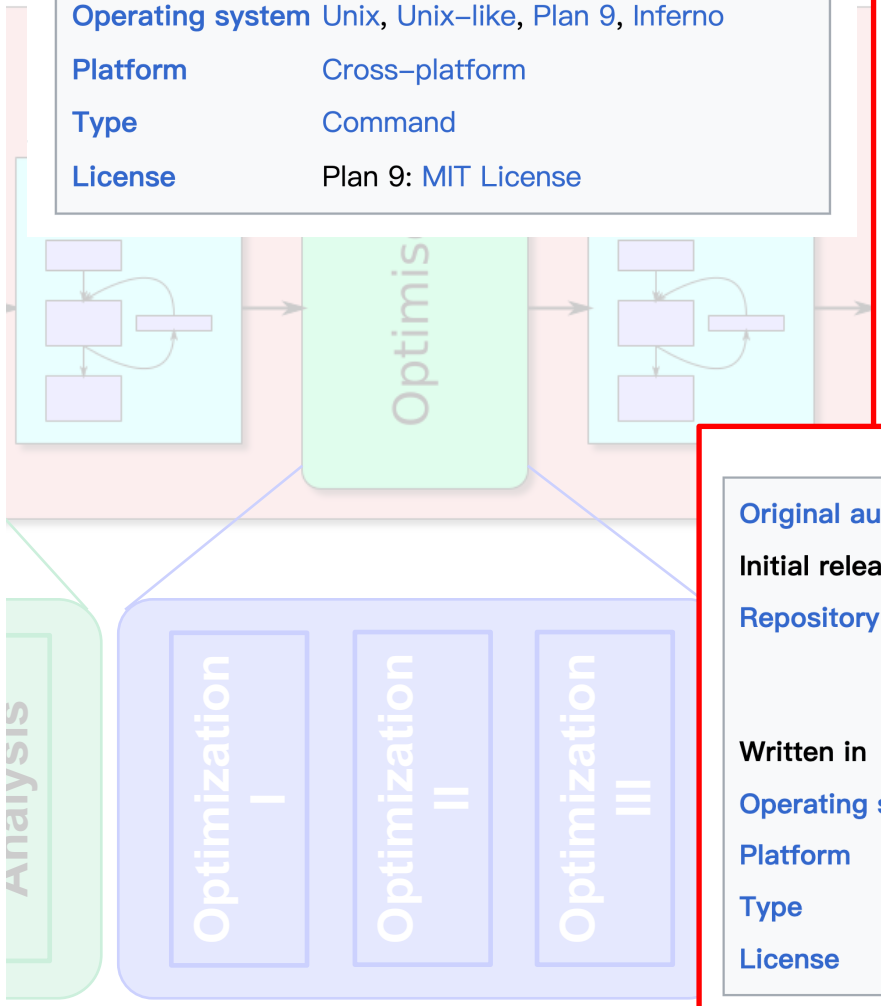
Tools

GNU Bison

Original author(s)	Robert Corbett
Developer(s)	The GNU Project
Initial release	June 1985; 38 years ago[1]
Stable release	3.8.2[2] / 25 September 2021
Repository	git.savannah.gnu.org/cgit/bison.git
Written in	C and m4
Operating system	Unix-like
Type	Parser generator
License	GPL
Website	www.gnu.org/software/bison/

Yacc

Original author(s)	Stephen C. Johnson
Repository	www.tuhs.org/cgi-bin/utree.pl?file=V6%2Fusr%2Fsource%2Fyacc
Written in	C
Operating system	Unix, Unix-like, Plan 9, Inferno
Platform	Cross-platform
Type	Command
License	Plan 9: MIT License



flex

Developer(s)	Vern Paxson
Initial release	around 1987; 37 years ago[1]
Stable release	2.6.4 / May 6, 2017; 6 years ago
Repository	github.com/westes/flex.git
Operating system	Unix-like
Type	Lexical analyzer generator
License	BSD license
Website	github.com/westes/flex

Lex

Original author(s)	Mike Lesk, Eric Schmidt
Initial release	1975; 49 years ago
Repository	minnie.tuhs.org/cgi-bin/utree.pl?file=4BSD%2Fusr%2Fsrc%2Fcmd%2Flex
Written in	C
Operating system	Unix, Unix-like, Plan 9
Platform	Cross-platform
Type	Command
License	Plan 9: MIT License

Tools

GNU Bison

Original author(s)	Robert Corbett
Developer(s)	The GNU Project
Initial release	June 1985; 38 years ago[1]
Stable release	3.8.2[2] / 25 September 2021
Repository	git.savannah.gnu.org/cgit/bison.git
Written in	C and m4
Operating system	Unix–like
Type	Parser generator
License	GPL
Website	www.gnu.org/software/bison/

Yacc

Original author(s)	Stephen C. Johnson
Repository	www.tuhs.org/cgi-bin/utree.pl?file=V6%2Fusr%2Fsource%2Fyacc
Written in	C
Operating system	Unix, Unix–like, Plan 9, Inferno
Platform	Cross–platform
Type	Command
License	Plan 9: MIT License

ANTLR

Original author(s)	Terence Parr and others
Initial release	April 10, 1992; 32 years ago
Stable release	4.13.1 / 4 September 2023; 7 months ago
Repository	github.com/antlr/antlr4
Written in	Java
Platform	Cross–platform
License	BSD License
Website	www.antlr.org

flex

Developer(s)	Vern Paxson
Initial release	around 1987; 37 years ago[1]
Stable release	2.6.4 / May 6, 2017; 6 years ago
Repository	github.com/westes/flex.git
Operating system	Unix–like
Type	Lexical analyzer generator
License	BSD license
Website	github.com/westes/flex

Lex

Original author(s)	Mike Lesk, Eric Schmidt
Initial release	1975; 49 years ago
Repository	minnie.tuhs.org/cgi-bin/utree.pl?file=4BSD%2Fusr%2Fsrc%2Fcmd%2Flex
Written in	C
Operating system	Unix, Unix–like, Plan 9
Platform	Cross–platform
Type	Command
License	Plan 9: MIT License

LLVM

The LLVM logo, a stylized wyvern[1]

Original author(s)	Chris Lattner, Vikram Adve
Developer(s)	LLVM Developer Group
Initial release	2003; 21 years ago
Stable release	18.1.5[2] / 2 May 2024
Repository	github.com/llvm/llvm-project
Written in	C++
Operating system	Cross-platform
Type	Compiler
License	UIUC (BSD-style) Apache License 2.0 with LLVM Exceptions (v9.0.0 or later)[3]
Website	www.llvm.org

Yacc

Original author(s)	Stephen C. Johnson
Repository	www.tuhs.org/cgi-bin/utree.pl?file=V6%2Fusr%2Fsource%2Fyacc
Written in	C
Operating system	Unix, Unix-like, Plan 9, Inferno
Platform	Cross-platform
Type	Command
License	Plan 9: MIT License

ANTLR

Original author(s)	Terence Parr and others
Initial release	April 10, 1992; 32 years ago
Stable release	4.13.1 / 4 September 2023; 7 months ago
Repository	github.com/antlr/antlr4
Written in	Java
Platform	Cross-platform
License	BSD License
Website	www.antlr.org

flex

Developer(s)	Vern Paxson
Initial release	around 1987; 37 years ago[1]
Stable release	2.6.4 / May 6, 2017; 6 years ago
Repository	github.com/westes/flex.git
Operating system	Unix-like
Type	Lexical analyzer generator
License	BSD license
Website	github.com/westes/flex

Lex

Original author(s)	Mike Lesk, Eric Schmidt
Initial release	1975; 49 years ago
Repository	minnie.tuhs.org/cgi-bin/utree.pl?file=4BSD%2Fusr%2Fsrc%2Fcmd%2Flex
Written in	C
Operating system	Unix, Unix-like, Plan 9
Platform	Cross-platform
Type	Command
License	Plan 9: MIT License

Website	www.gnu.org/software/bison/

LLVM

The LLVM logo, a stylized wyvern[1]

Original author(s)	Chris Lattner, Vikram Adve
Developer(s)	LLVM Developer Group
Initial release	2003; 21 years ago
Stable release	18.1.5[2] / 2 May 2024
Repository	github.com/llvm/llvm-project
Written in	C++
Operating system	Cross-platform
Type	Compiler
License	UIUC (BSD-style) Apache License 2.0 with LLVM Exceptions (v9.0.0 or later)[3]
Website	www.llvm.org

Yacc

Original author(s)	Stephen C. Johnson
Repository	www.tuhs.org/cgi-bin/utree.pl?file=V6%2Fusr%2Fsource%2Fyacc
Written in	C
Operating system	Unix, Unix-like, Plan 9, Inferno
Platform	Cross-platform
Type	Command
License	Plan 9: MIT License

ANTLR

Original author(s)	Terence Parr and others
Initial release	April 10, 1992; 32 years ago
Stable release	4.13.1 / 4 September 2023; 7 months ago
Repository	github.com/antlr/antlr4
Written in	Java
Platform	Cross-platform
License	BSD License
Website	www.antlr.org

flex

Developer(s)	Vern Paxson
Initial release	around 1987; 37 years ago[1]
Stable release	2.6.4 / May 6, 2017; 6 years ago
Repository	github.com/westes/flex.git
Operating system	Unix-like
Type	Lexical analyzer generator
License	BSD license
Website	github.com/westes/flex

Lex

Original author(s)	Mike Lesk, Eric Schmidt
Initial release	1975; 49 years ago
Repository	minnie.tuhs.org/cgi-bin/utree.pl?file=4BSD%2Fusr%2Fsrc%2Fcmd%2Flex
Written in	C
Operating system	Unix, Unix-like, Plan 9
Platform	Cross-platform
Type	Command
License	Plan 9: MIT License

Website	www.gnu.org/software/bison/

编译技术
全研究组

PART III: Applications

Applications

- Implementation of High-Level Programming Languages

Applications

- Implementation of High-Level Programming Languages

VERILOG

VHDL

P4

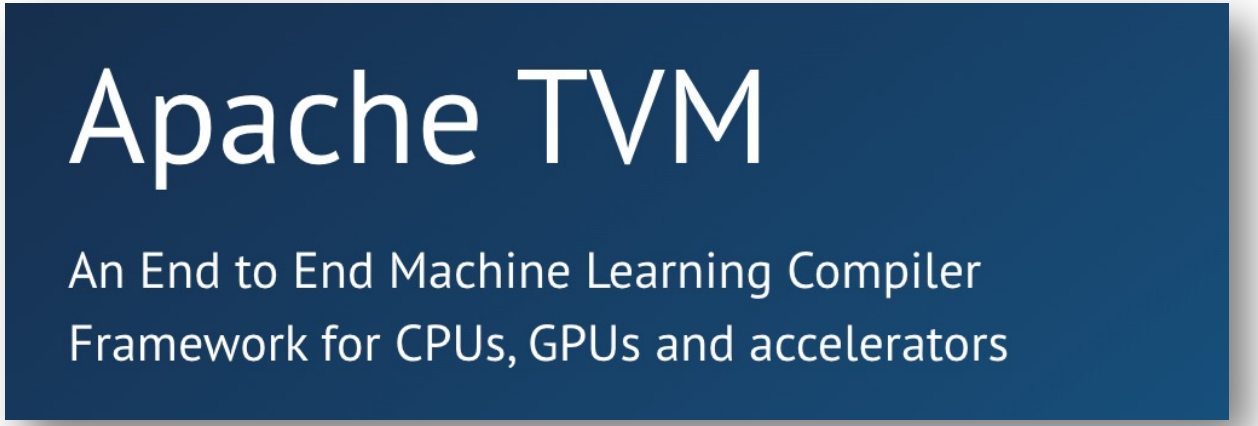
Domain Specific Language

Applications

- Implementation of High-Level Programming Languages
- **Design & Optimization of Computer Architectures**
  - AI, Gaming, Embedded Systems, High Performance Computing, …

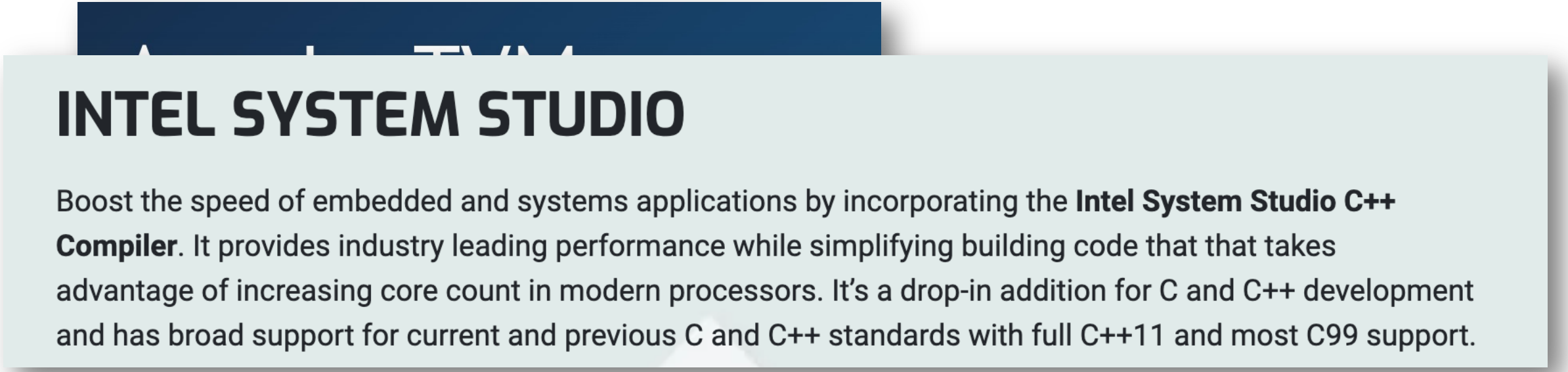
Applications

- Implementation of High-Level Programming Languages
- **Design & Optimization of Computer Architectures**
  - AI, Gaming, Embedded Systems, High Performance Computing, …



Applications

- Implementation of High-Level Programming Languages
- **Design & Optimization of Computer Architectures**
  - AI, Gaming, Embedded Systems, High Performance Computing, …



Applications

- Implementation of High-Level Programming Languages
- **Design & Optimization of Computer Architectures**
  - AI, Gaming, Embedded Systems, High Performance Computing, …

NVIDIA HPC Fortran, C++ and C Compilers with OpenACC

Using NVIDIA HPC compilers for NVIDIA data center GPUs and X86-64, OpenPOWER and Arm Server multi-core CPUs, programmers can accelerate science and engineering applications using Standard C++ and Fortran parallel constructs, OpenACC directives and CUDA Fortran.

Applications

- Implementation of High-Level Programming Languages
- Design & Optimization of Computer Architectures
- **Program Translation (e.g., C to Rust)**

Applications

- Implementation of High-Level Programming Languages
- Design & Optimization of Computer Architectures
- **Program Translation (e.g., C to Rust)**

Biden: “All non-Rust projects are illegal”

”Programmers do not write code without consequence, and the way they do it is critical to the national interest.”

Aaron 0928 · Follow

8 min read · Feb 29, 2024

Applications

- Implementation of High-Level Programming Languages
- Design & Optimization of Computer Architectures
- **Program Translation (e.g., C to Rust)**

Biden: "All non-Rust proj
illegal"

"Programmers do not write code without conse
they do it is critical to the national interest."

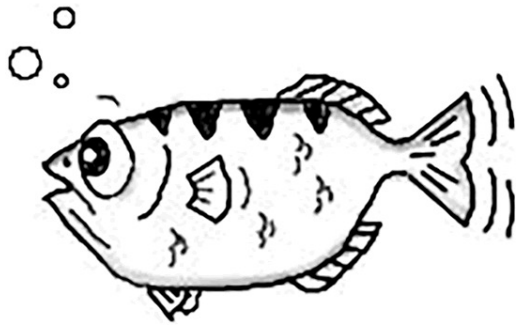
Aaron 0928 · Follow
8 min read · Feb 29, 2024

Applications

- Implementation of High-Level Programming Languages
- Design & Optimization of Computer Architectures
- Program Translation (e.g., C to Rust)
- **Software Productivity Tools**

Applications

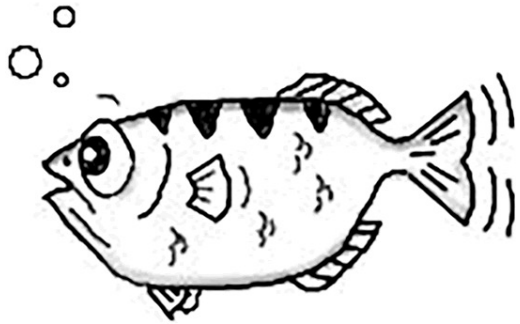
- Implementation of High-Level Programming Languages
- Design & Optimization of Computer Architectures
- Program Translation (e.g., C to Rust)
- **Software Productivity Tools**



GDB
The GNU Project Debugger

Applications

- Implementation of High-Level Programming Languages
- Design & Optimization of Computer Architectures
- Program Translation (e.g., C to Rust)
- **Software Productivity Tools**

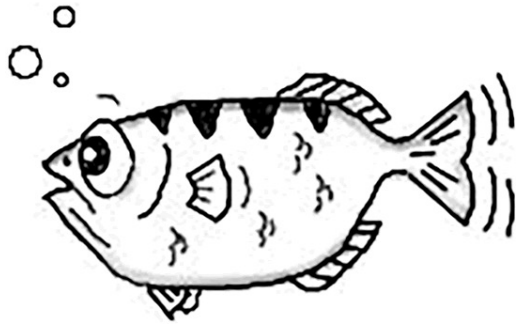


GDB
The GNU Project Debugger

JProfiler

Applications

- Implementation of High-Level Programming Languages
- Design & Optimization of Computer Architectures
- Program Translation (e.g., C to Rust)
- **Software Productivity Tools**



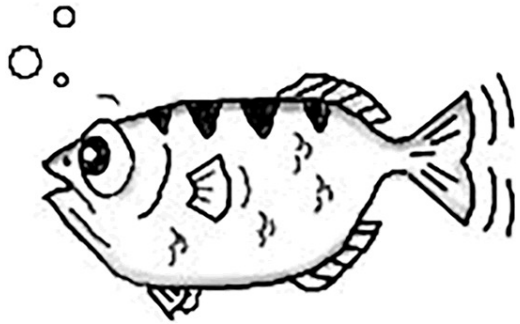
GDB
The GNU Project
Debugger

JProfiler

JACOCO
Java Code Coverage

Applications

- Implementation of High-Level Programming Languages
- Design & Optimization of Computer Architectures
- Program Translation (e.g., C to Rust)
- **Software Productivity Tools**



GDB
The GNU Project Debugger

JProfiler

checkstyle

JACOCO
Java Code Coverage

Applications

- Implementation of High-Level Programming Languages
- Design & Optimization of Computer Architectures
- Program Translation (e.g., C to Rust)
- Software Productivity Tools
- **Security Assurance Tools**
  - Bug finding, Instrumentation, Defense, …

Applications

- Implementation of High-Level Programming Languages
- Design & Optimization of Computer Architectures
- Program Translation (e.g., C to Rust)
- Software Productivity Tools
- **Security Assurance Tools**
  - Bug finding, Instrumentation, Defense, …

Clang Power Tools

Applications

- Implementation of High-Level Programming Languages
- Design & Optimization of Computer Architectures
- Program Translation (e.g., C to Rust)
- Software Productivity Tools
- **Security Assurance Tools**
  - Bug finding, Instrumentation, Defense, …

Applications

- Implementation of High-Level Programming Languages
- Design & Optimization of Computer Architectures
- Program Translation (e.g., C to Rust)
- Software Productivity Tools
- **Security Assurance T**
  - Bug finding, Instrumenta

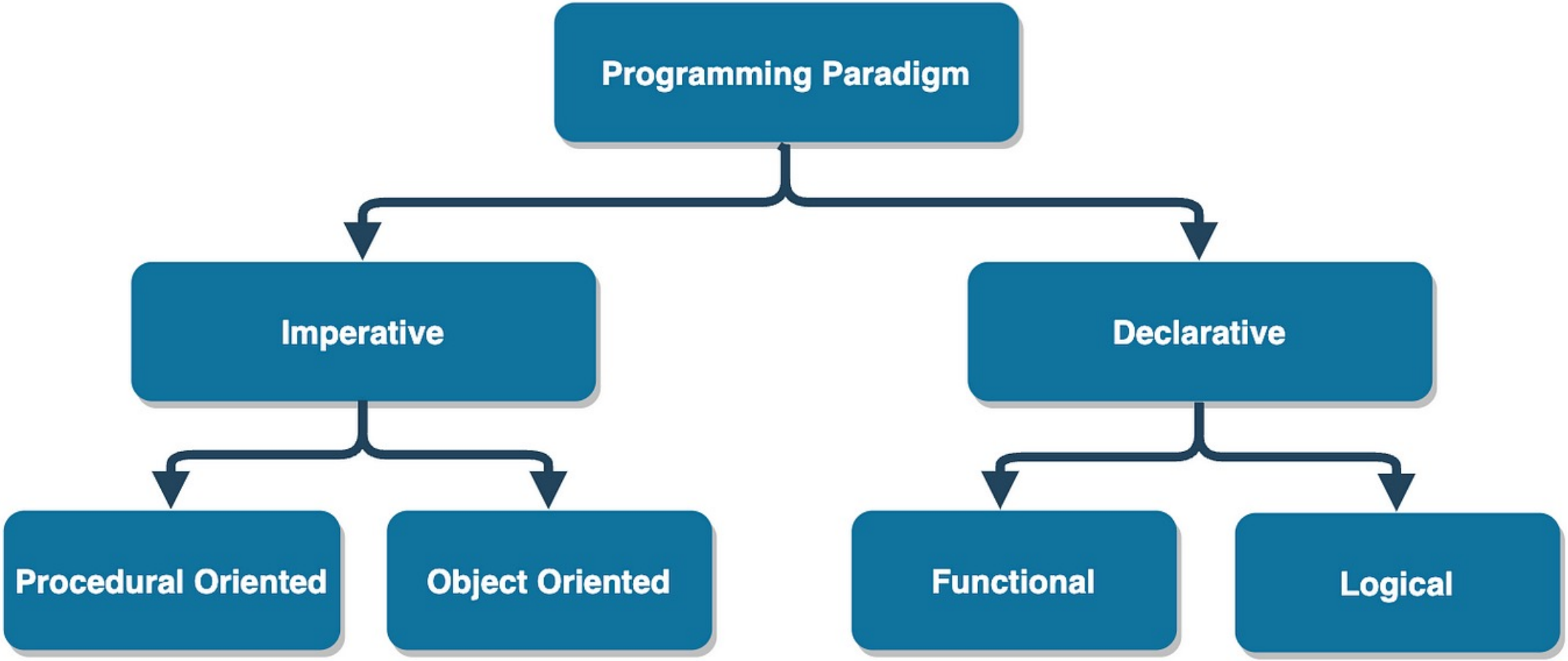
Clang
Power Tools

Applications

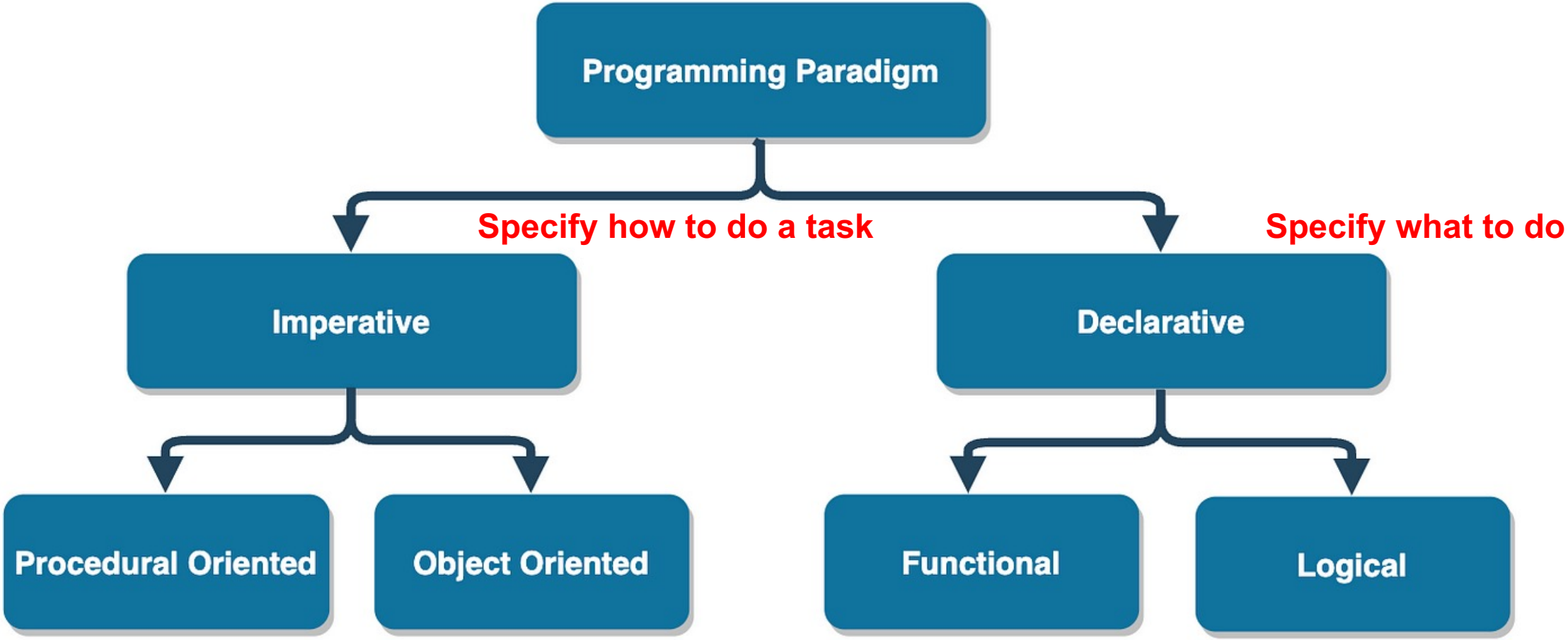
- **Security Assurance T**
  - Bug finding, Instrumenta

PART IV: Terminologies Review

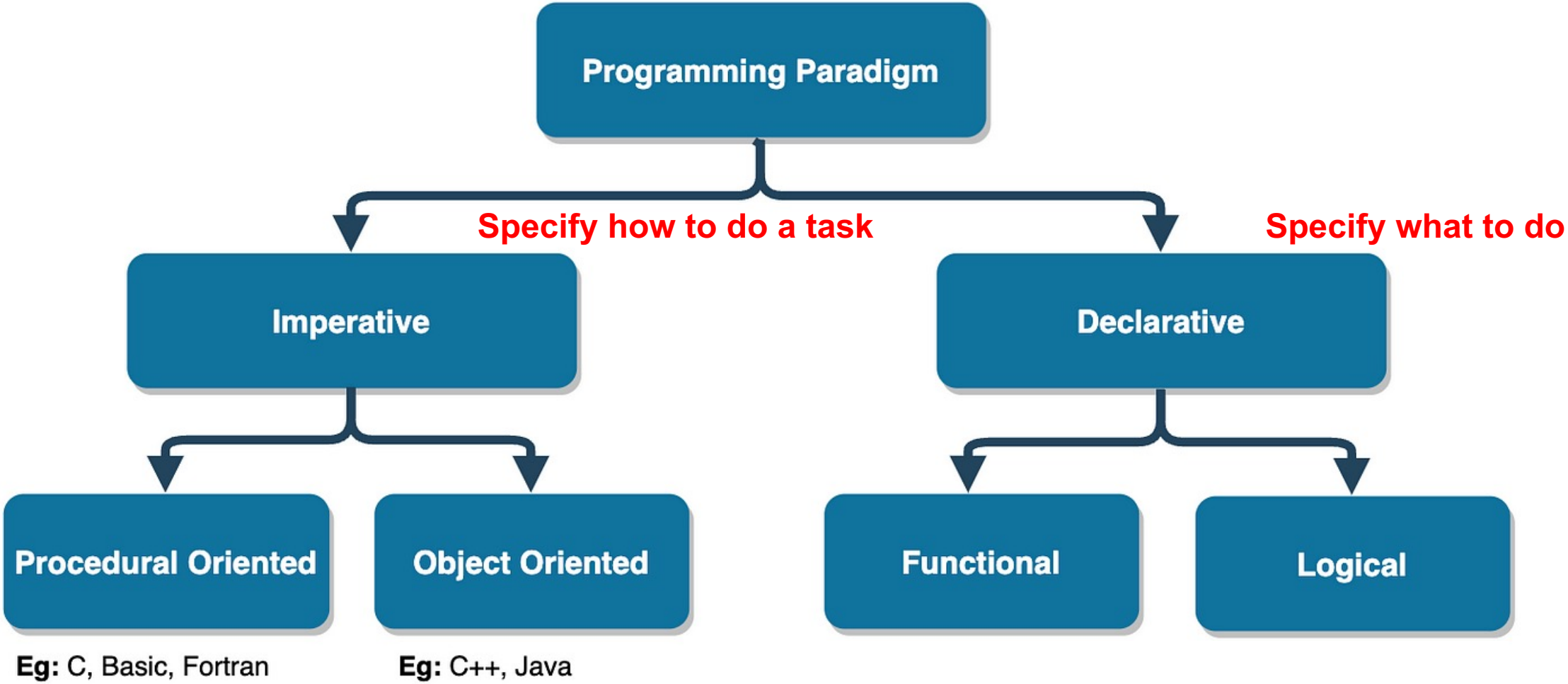
Programming Languages



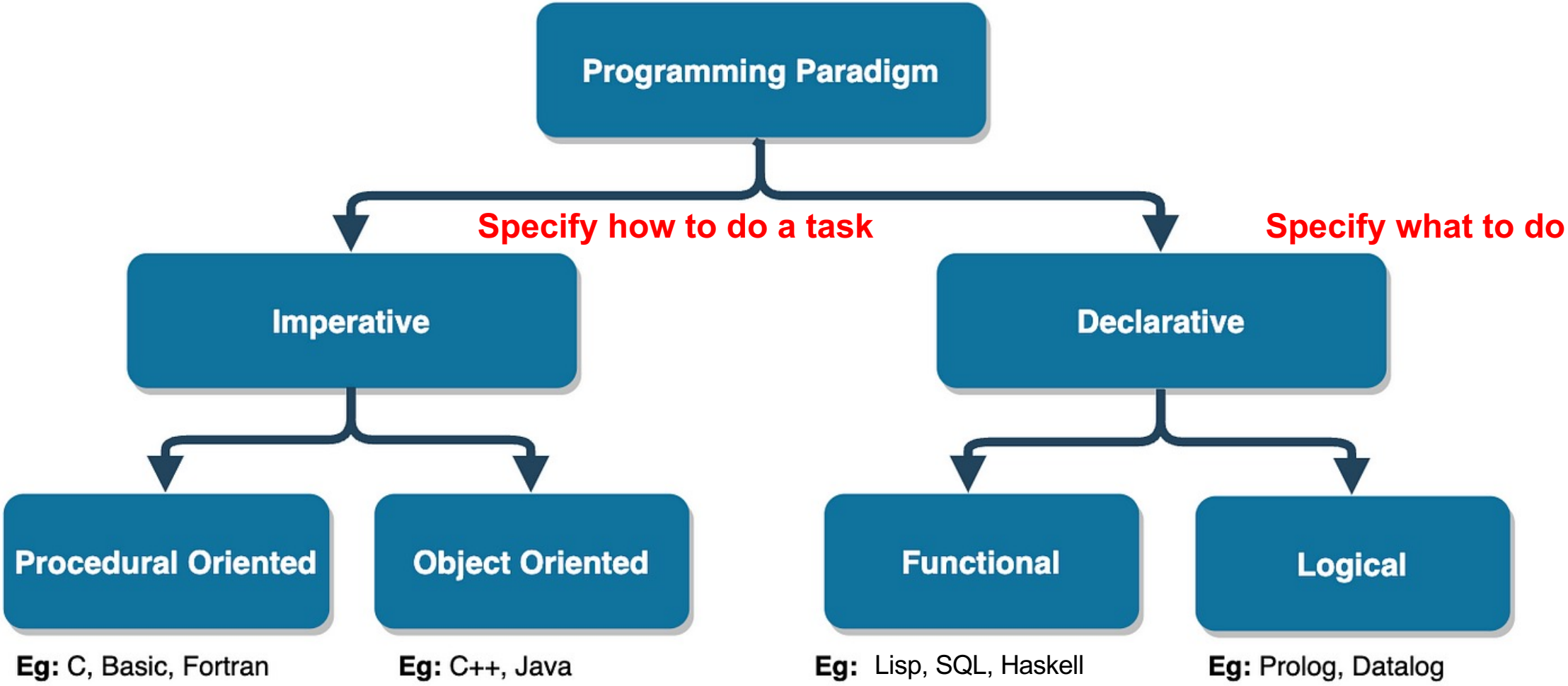
Programming Languages



Programming Languages



Programming Languages



Functional Programming

Functional Programming

- Functional programming contains **only functions** without mutating the state or data.
  - It means that there are **no global data/variables**.

Functional Programming

- Functional programming contains only functions without mutating the state or data.
- Composing and applying functions is the main driving force in this paradigm.

Functional Programming

- Functional programming contains only functions without mutating the state or data.
- Composing and applying functions is the main driving force in this paradigm.

```
select
    sum(case when some_flag = 'X' then some_column else other_column end)
from ...
```

Functional Programming

- Functional programming contains only functions without mutating the state or data.
- Composing and applying functions is the main driving force in this paradigm.
- A function can take another as an argument and return a new function.

Logic Programming

Logic Programming

- The Logical paradigm has its foundations in mathematical logic in which we declare **facts** and **rules**

Logic Programming

- The Logical paradigm has its foundations in mathematical logic in which we declare **facts** and **rules**, e.g.,
  - **Fact:** Nanjing is rainy
  - **Fact:** Beijing is rainy; Beijing is cold
  - **Rule:** If a city is both rainy and cold, then it is snowy

Logic Programming

- The Logical paradigm has its foundations in mathematical logic in which we declare **facts** and **rules**, e.g.,
  - **Fact:** Nanjing is rainy
  - **Fact:** Beijing is rainy; Beijing is cold
  - **Rule:** If a city is both rainy and cold, then it is snowy

```
1.  rainy(“Nanjing”)
2.  rainy(“Beijing”);
3.  cold(“Beijing”)
4.  snowy(c) :- rainy(c),  cold(c)
5.  .output snowy
```

Type Systems

- Static vs. Dynamic Typing
- Strong vs. Weak Typing

Type Systems

- **Static vs. Dynamic Typing**
  - is about when type information is acquired (at compile time or runtime?)

Type Systems

- **Static vs. Dynamic Typing**
  - is about when type information is acquired (at compile time or runtime?)

For example in Java:

```
String str = "Hello";  //statically typed as string
str = 5;               //would throw an error since java is statically typed
```

Type Systems

- **Static vs. Dynamic Typing**
  - is about when type information is acquired (at compile time or runtime?)

For example in Java:

```
String str = "Hello";  //statically typed as string
str = 5;               //would throw an error since java is statically typed
```

For example in Python:

```
str = "Hello" # it is a string
str = 5       # now it is an integer; perfectly OK
```

Type Systems

- **Strong vs. Weak Typing**
  - is about how strictly types are distinguished (e.g. whether the language can do an **implicit conversion** from **strings** to **numbers**).

Type Systems

- **Strong vs. Weak Typing**
  - is about how strictly types are distinguished (e.g. whether the language can do an **implicit conversion** from strings to numbers).

For example in Python:

```
str = 5 + "hello"
# would throw an error since it does not want to cast one type to the other
```

whereas in PHP:

```
$str = 5 + "hello"; // equals 5 because "hello" is implicitly casted to 0
// PHP is weakly typed, thus is a very forgiving language.
```

Scoping

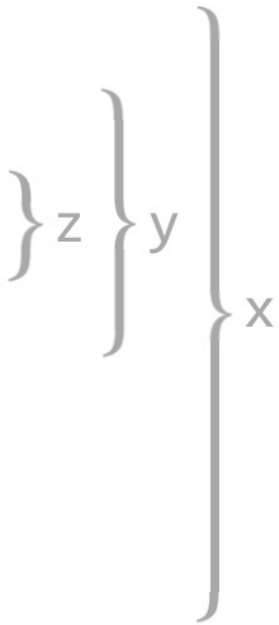
- Static vs. Dynamic scoping

Scoping

- Static vs. Dynamic scoping

```
/* C++ Example Code */
int x = 5;
if(x < 10) {
    int y = 20;
    if(x < 30) {
        int z = y*2;
    }
}

/* Print Contents of Variables */
cout << x << endl;  /* Ok, prints 5 */
cout << y << endl;  /* Error! */
cout << z << endl;  /* Error! */
```



Scoping

- Static vs. Dynamic scoping

```
/* C++ Example Code */
int x = 5;
if(x < 10) {
    int y = 20;
    if(x < 30) {
        int z = y*2;
    }
}

/* Print Contents of Variables */
cout << x << endl;  /* Ok, prints 5 */
cout << y << endl;  /* Error! */
cout << z << endl;  /* Error! */
```

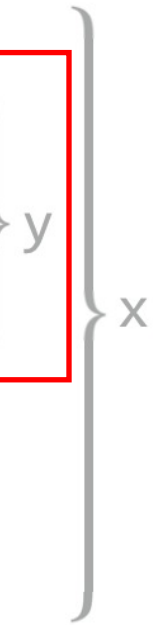
z, y, x

Scoping

- Static vs. Dynamic scoping

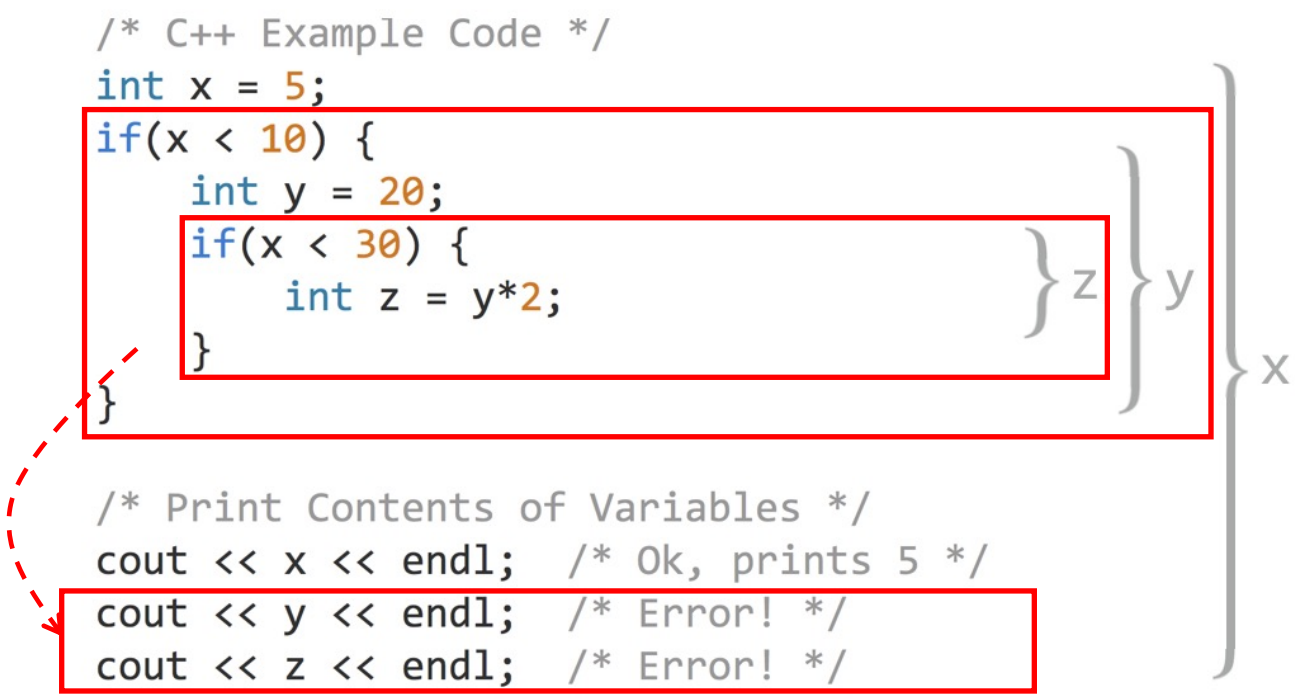
```
/* C++ Example Code */
int x = 5;
if(x < 10) {
    int y = 20;
    if(x < 30) {
        int z = y*2;
    }
}

/* Print Contents of Variables */
cout << x << endl;  /* Ok, prints 5 */
cout << y << endl;  /* Error! */
cout << z << endl;  /* Error! */
```



Scoping

- Static vs. Dynamic scoping

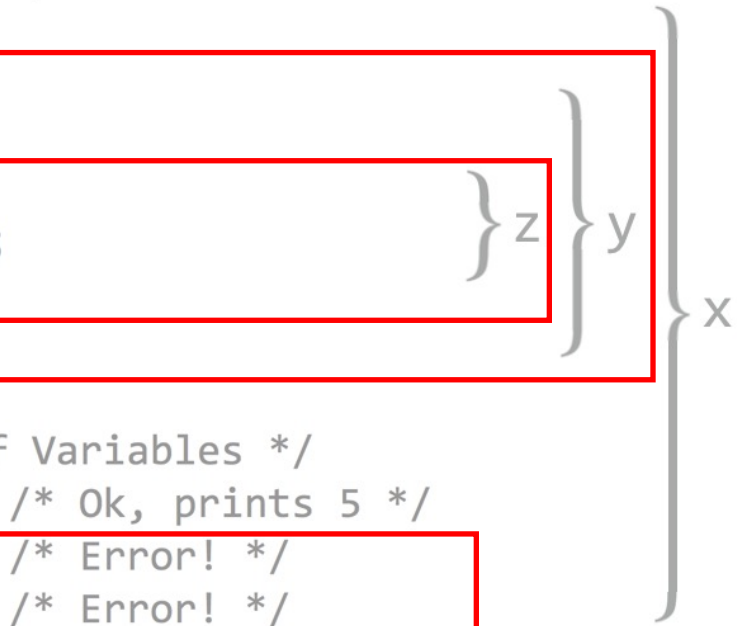


Scoping

- Static vs. Dynamic scoping

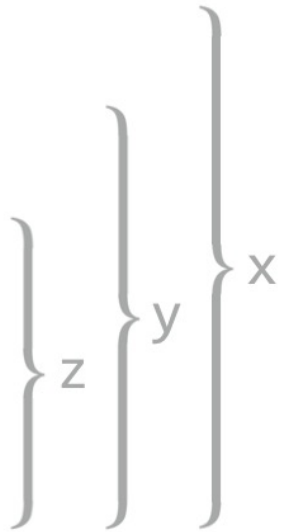
```cpp
/* C++ Example Code */
int x = 5;
if(x < 10) {
    int y = 20;
    if(x < 30) {
        int z = y*2;
    }
}

/* Print Contents of Variables */
cout << x << endl;  /* Ok, prints 5 */
cout << y << endl;  /* Error! */
cout << z << endl;  /* Error! */
```




```python
# Python Example Code
x = 5
if x < 10:
    y = 20
    if x < 30:
        z = y * 2

# Print Contents of Variables
print(x)  # Ok, prints 5
print(y)  # Ok, prints 20
print(z)  # Ok, prints 40
```



Function Invocation

- Call by value vs. Call by reference

Function Invocation

- Call by value vs. Call by reference

```
void swap(int x, int y)
{
    int t;

    t = x;
    x = y;
    y = t;
}
```

vs.

```
void swap(int &x, int &y)
{
    int t;

    t = x;
    x = y;
    y = t;
}
```

Function Invocation

- Call by value vs. Call by reference

```
void swap(int x, int y)
{
    int t;

    t = x;
    x = y;
    y = t;
}
```

vs.

```
void swap(int &x, int &y)
{
    int t;

    t = x;
    x = y;
    y = t;
}
```

```
int a = 5, b = 10; swap(a, b); printf("a = %d; b = %d", a, b);
```

Virtual Functions

Virtual Functions

- A virtual function is a member function declared within a base class and can be re-defined (overridden) by a derived class.

Virtual Functions

- A virtual function is a member function declared within a base class and can be re-defined (overridden) by a derived class.
- Any non-private, non-static, and non-final method in Java is a virtual function.

Virtual Functions

- A virtual function is a member function declared within a base class and can be re-defined (overridden) by a derived class.
- Any non-private, non-static, and non-final method in Java is a virtual function.

```
1.    void add(List<int> list, int y)
2.    {
3.        list.add(y); // ArrayList.add() or LinkedList.add()?
4.    }
```

Many many terminologies / features in a programming language!

Many many terminologies / features in a programming language!