Lecture 1
Introduction to Compilers

Outline

Preface

- “Principle of Compiler” is designed to introduce the student to the principles and practices of programming language implementation.

- We cover **lexical analysis**, **parsing theory** (syntax analysis), **semantic analysis**, runtime environments, and code generation.

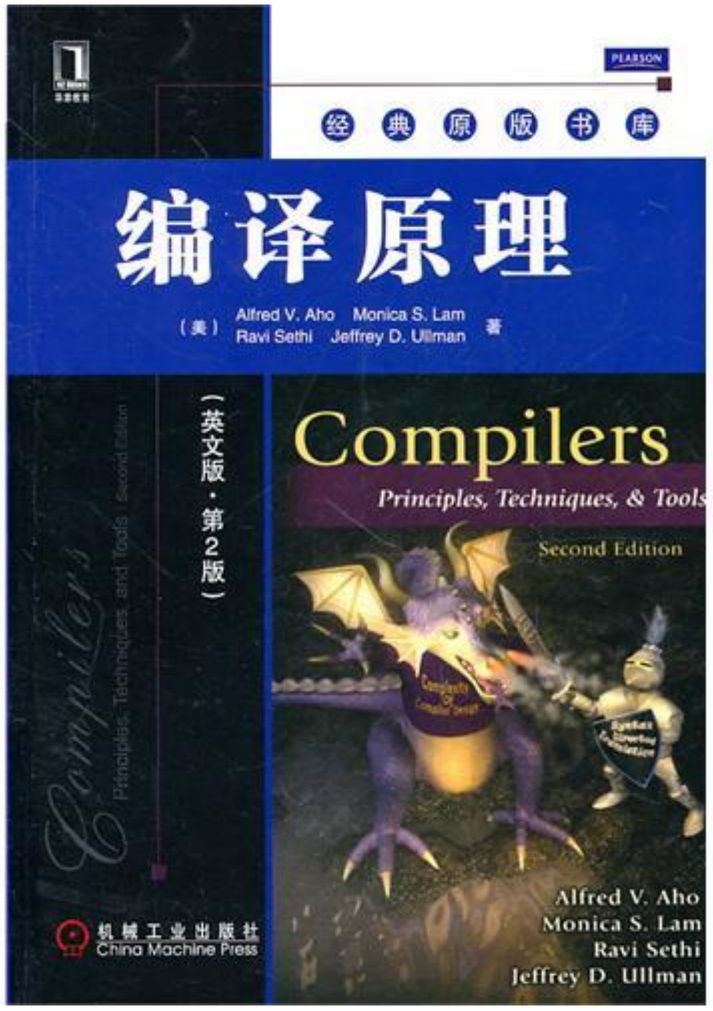
Grading

- The coursework will consist of programming projects, pencil-and-paper homework, and a final exam.
  - Assignments and attendance: 20%
  - Experiments: 20%
  - Final exam: 60%

Reference Material

- Textbook
  - *Compilers: Principles, Techniques, and Tools*
    - by Aho, Sethi, and Ullman—also known as "The Red Dragon Book"

- CS164 in Berkeley and CS143 in Stanford



Why Study Compilers?

- Learn how to **build programming languages**.
- Learn **how programming languages work**.
- Improve understanding of **program behavior**
- Learn **tradeoffs in language design**.
- Increase **capacity of expression**
- Learn to **build a large and reliable system**
- See many **basic CS concepts** at work
- See theory **come to life**.

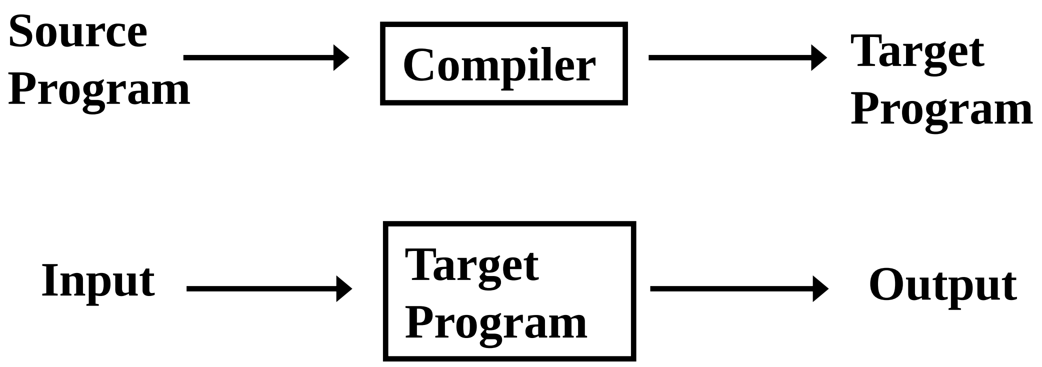
Outline

Why Compiler

- The development of programming language
  - Machine language (C7 06 0000 0002)
  - Assembly language (MOV x,2)
  - High-level language (x=2)
- Computer can only execute the code written in the machine instructions of the computer. For high-level programs to be used, there must have a program to translate high-level programs to machine code

What is Compiler

- Compilers are computer programs that translate one language to another
- Source language: the input of a compiler, usually high-level language(such as C,C++)
- Target language: the output of a compiler, usually object code for the target machine(such as machine code, assembly language)

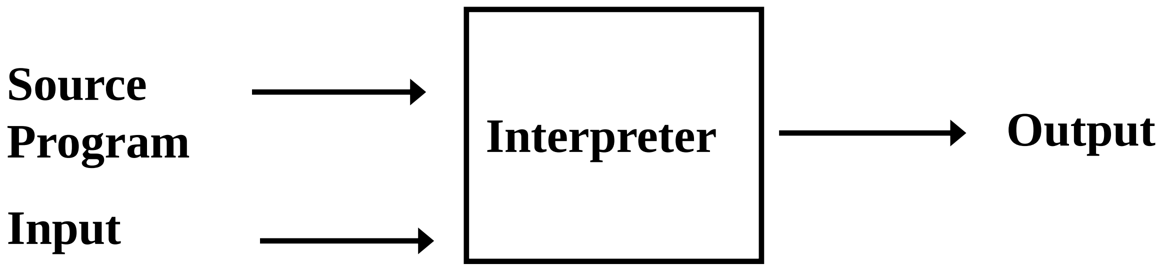


Problems to address

- How to *describe* language clearly for programmers, precisely for implementers?
- How to implement description, and know you’re right?
  - *Automatic conversion* of description to implementation
  - Testing
- How to save implementation effort?
  - Multiple languages to multiple targets: can we *re-use* effort?
- How to make languages *usable*?
  - Handle errors reasonably
  - Detect questionable constructs
  - Compile quickly

Interpreter and Compiler

- The same point: They are all language implementing system
- Differences:
  - Interpreter executes the source program during translation
  - Compiler generates object code that is executed after translation completes



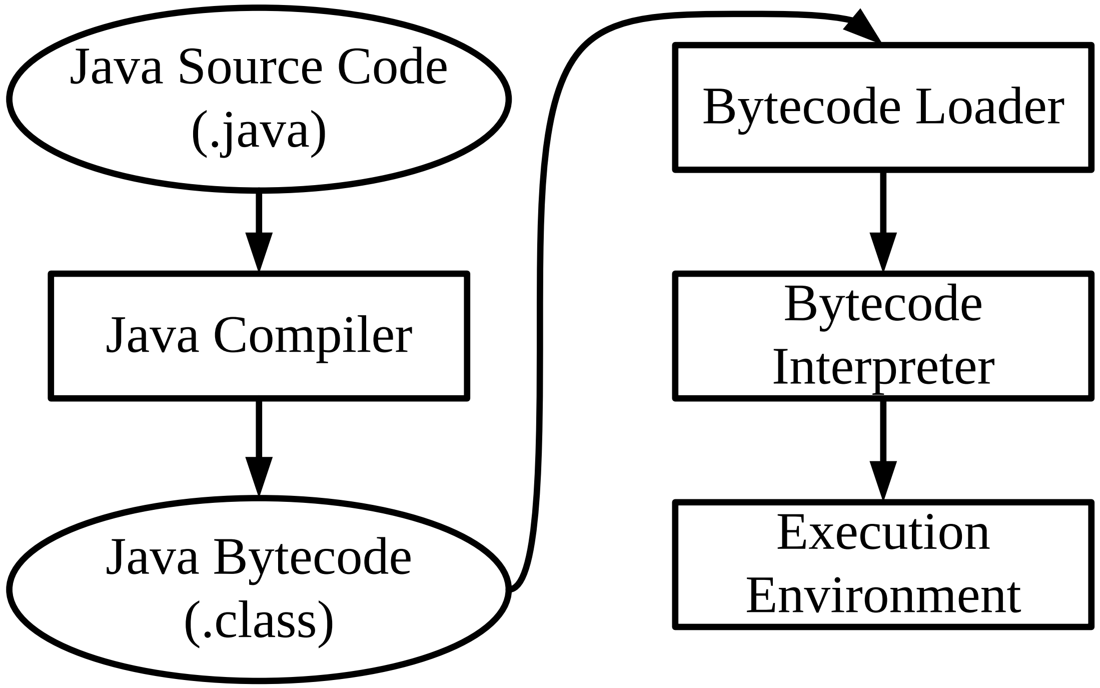
Interpreter and Compiler

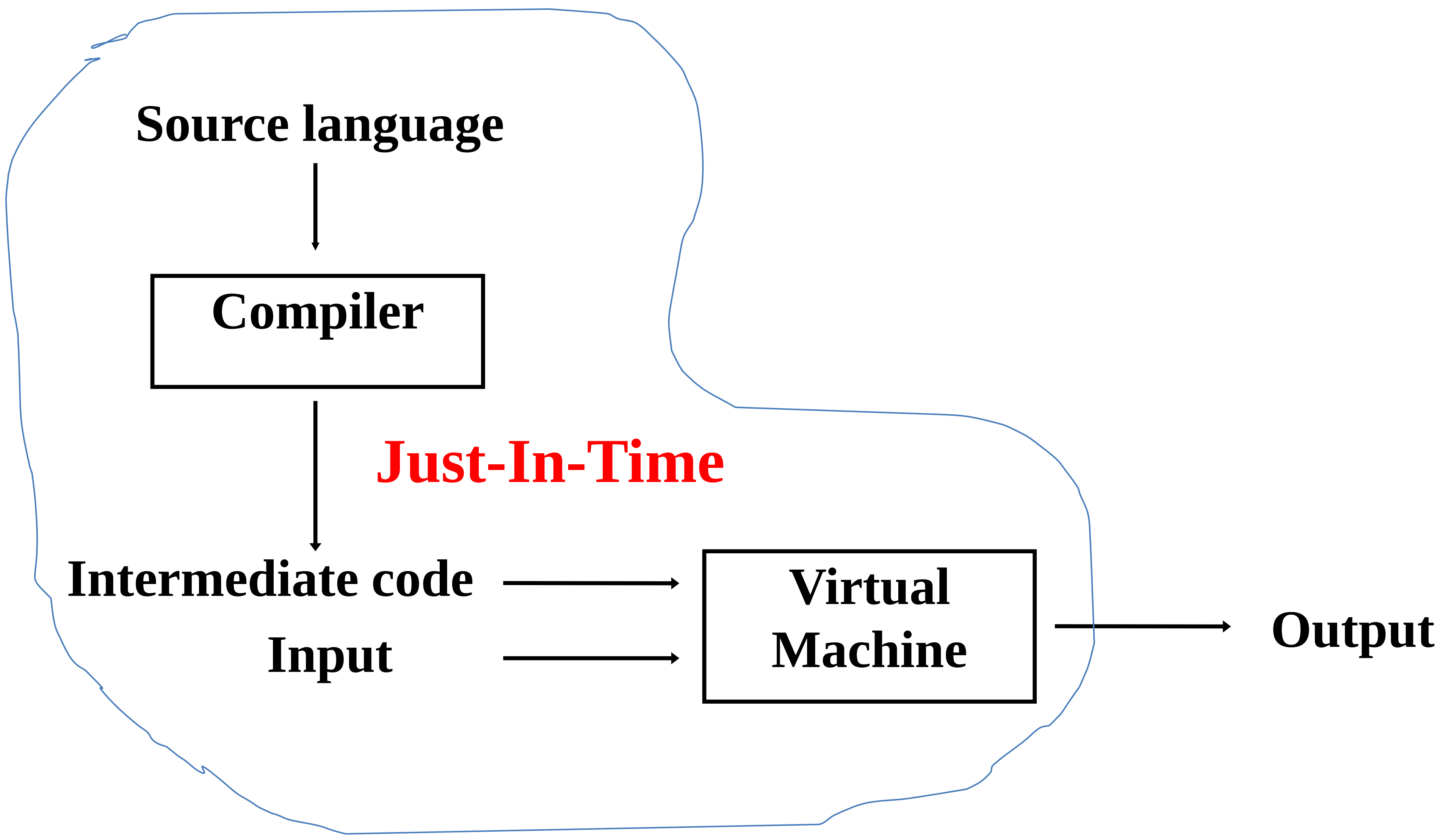
- Interpreters run programs "as is"
  - Little or no preprocessing
- Compilers do extensive preprocessing
  - Most implementations use compilers
- New trend is hybrid: "Just-In-Time" compilation, interpretation + compilation

How does JAVA work?

Interpreter and Compiler

- **JAVA work**

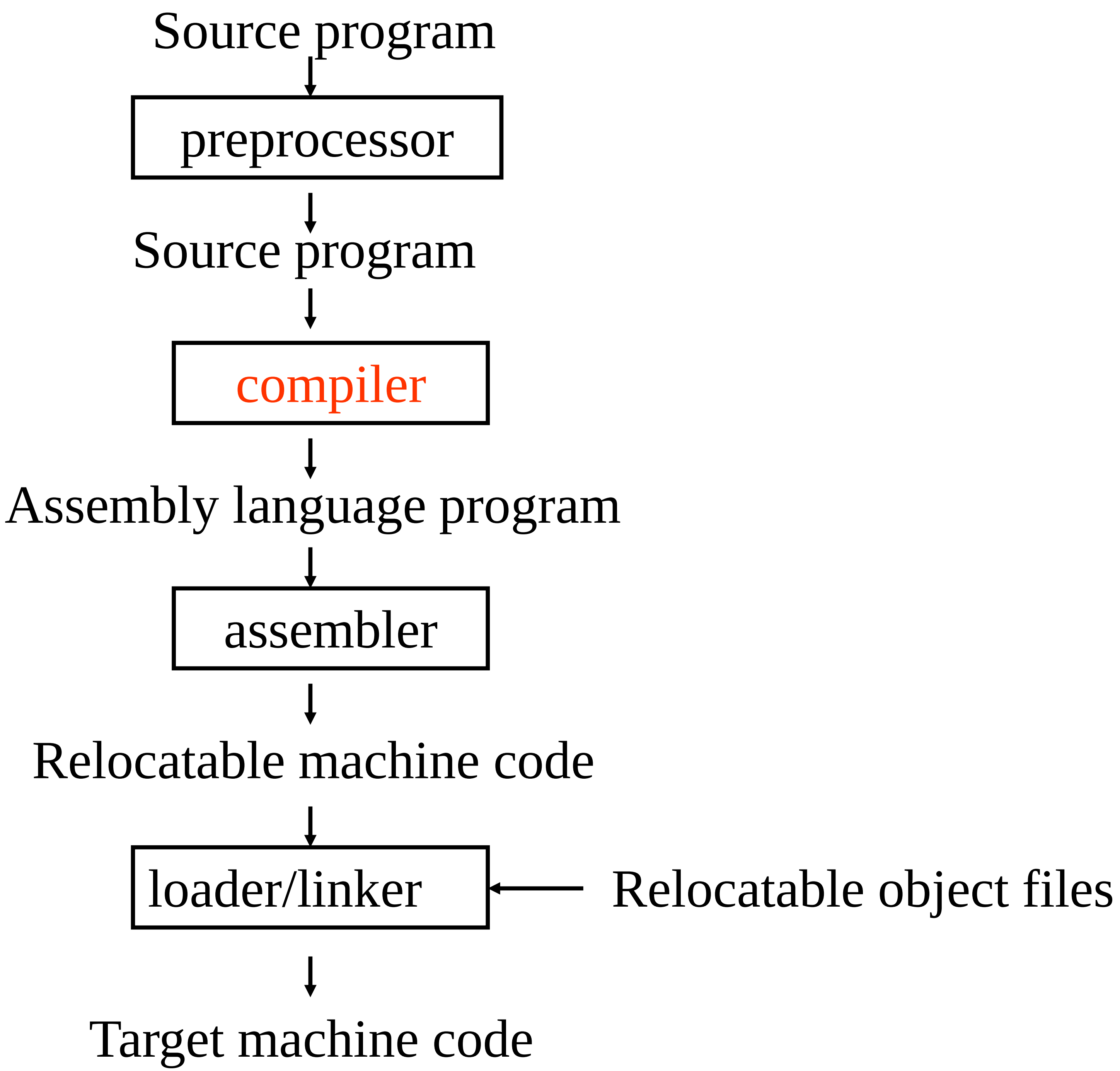




Mixed Compiler

Programs Related to Compilers

- The process of high-level programming language


Source program
preprocessor
Source program
compiler
Assembly language program
assembler
Relocatable machine code
loader/linker
Relocatable object files
Target machine code

Outline

A Short History of Compilers and Programming Languages

- First, there was nothing.
- Then, there was machine code.
- Then, there were assembly languages.
- Programming expensive; 50% of costs for machines went into programming.

History of High-Level Languages

- Initially, programs “hard-wired” or entered electro-mechanically: Analytical Engine, Jacquard Loom, ENIAC, punched-card-handling machines
- Programs encoded as numbers (machine language) stored as data: Manchester Mark I, EDSAC.
- In 1952 **Grace Hopper** had an operational compiler. "Nobody believed that," she said. "I had a running compiler and nobody would touch it. They told me computers could only do arithmetic."
- All programming done in assembly
- Problem: Software costs exceeded hardware costs!
- **John Backus**: “Speedcoding”
  - An interpreter
  - Ran 10-20 times slower than hand-written assembly

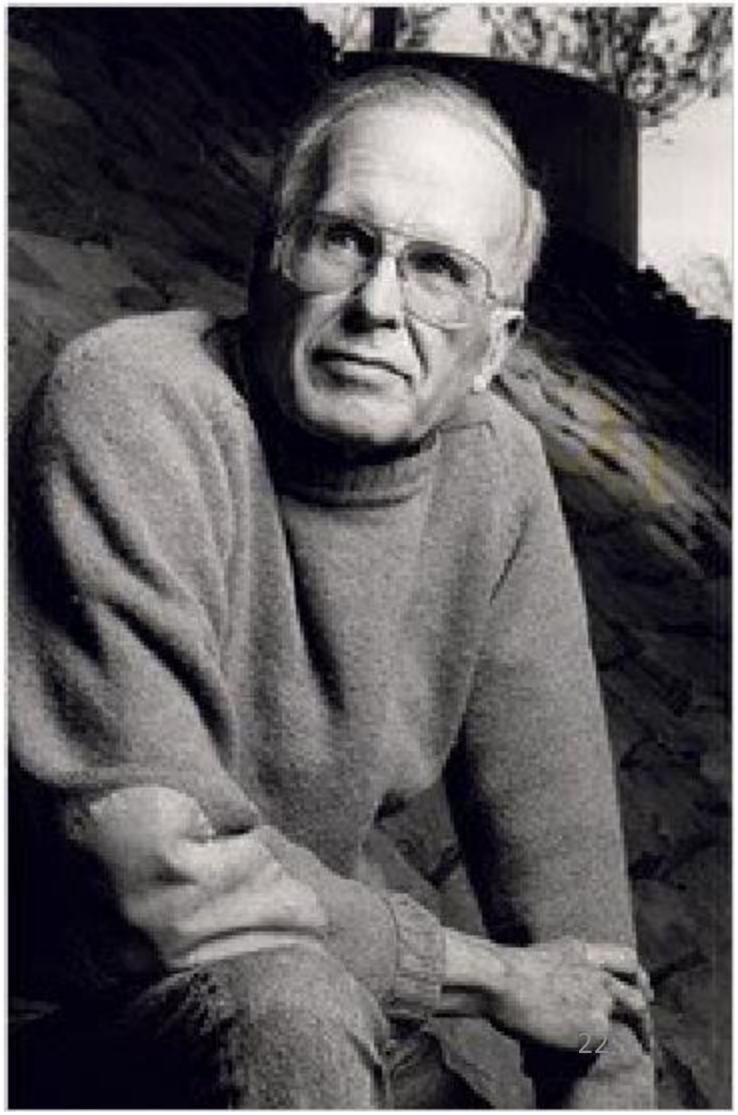
High-Level Languages

Rear Admiral **Grace Hopper**, inventor of A-0, COBOL, and the term "compiler."

Image: http://upload.wikimedia.org/wikipedia/commons/thumb/5/55/Grace_Hopper.jpg/300px-Grace_Hopper.jpg

High-Level Languages

John Backus, team leader on FORTRAN.



http://www.nytimes.com/2007/03/20/business/20backus.html

FORTRAN I

- 1953 IBM develops the 701 , and 1954 the 704
- John Backus
  - Idea: translate high-level code to assembly
  - Many thought this impossible

- 1954-7 FORTRAN I project
- By 1958, >50% of all software is in FORTRAN
- Cut development time dramatically
  - (2 wks to 2 hrs)

FORTRAN I

- The first compiler
  - Produced code almost as good as hand-written
  - Huge impact on computer science

- Led to an enormous body of theoretical work

- Modern compilers preserve the outlines of FORTRAN I

After FORTRAN

- LISP, late 1950s: dynamic, symbolic data structures.
- Algol 60: Europe’s answer to FORTRAN: modern syntax, block structure, explicit declaration. Set standard for language description.
- COBOL: late 1950’s. Business-oriented. Records.
  - Hopper's belief that programs should be written in a language that was close to English rather than in machine code or languages close to machine code (such as assembly language) was captured in the new business language, and COBOL would go on to be the most ubiquitous business language to date

The 60s Language Explosion

- APL (arrays), SNOBOL (strings), FORMAC (formulae), and many more.
- 1967-68: Simula 67, first “object-oriented” language.
- Algol 68: Combines FORTRANish numerical constructs, COBOLish records, pointers, all described in rigorous formalism. Remnants remain in C, but Algol68 deemed too complex.
- 1968: “Software Crisis” announced. Trend towards simpler languages: Algol W, Pascal, C

The 1970s

- Emphasis on “methodology”: modular programming,, Modula family.
- Mid 1970’s: Prolog. Declarative logic programming.
- Mid 1970’s: ML (Metalanguage) type inference, pattern-driven programming.
- Late 1970’s: DoD starts to develop Ada to consolidate >500 languages.

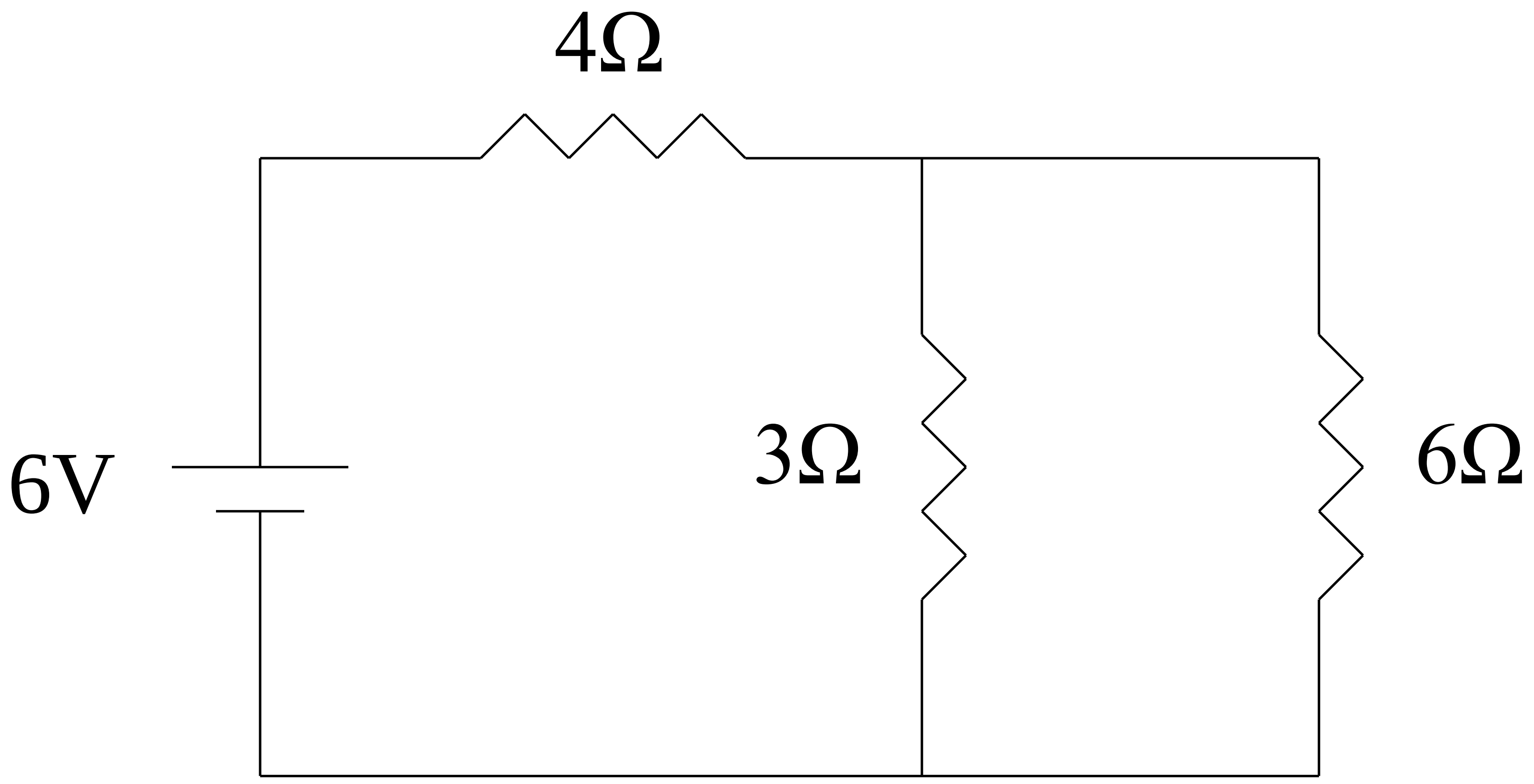
And on into the present

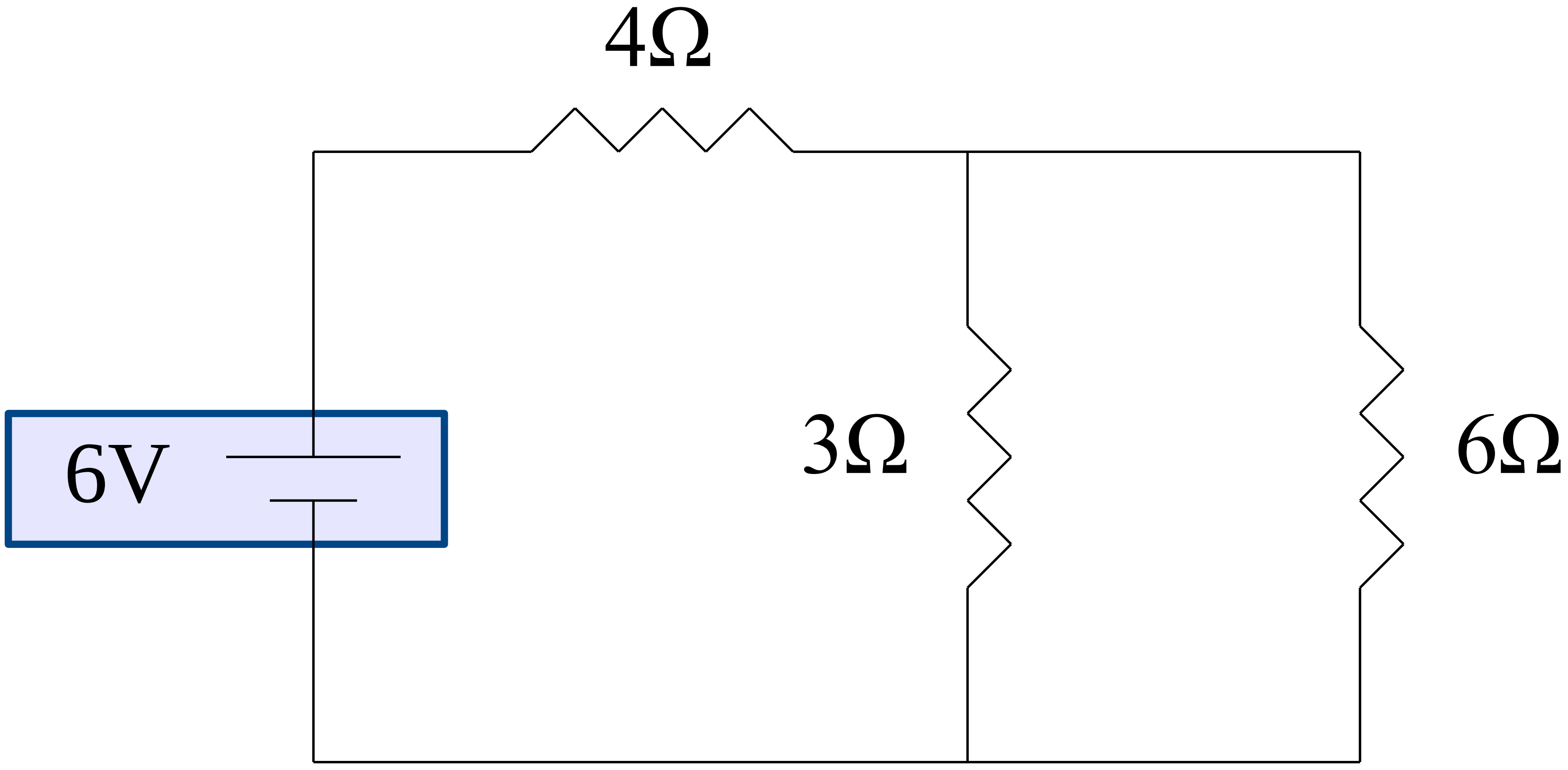
- Complexity increases with C++.
- Then decreases with Java.
- Then increases again (C#).
- Proliferation of little or specialized languages and scripting languages: HTML, PHP, Perl, Python, Ruby, …

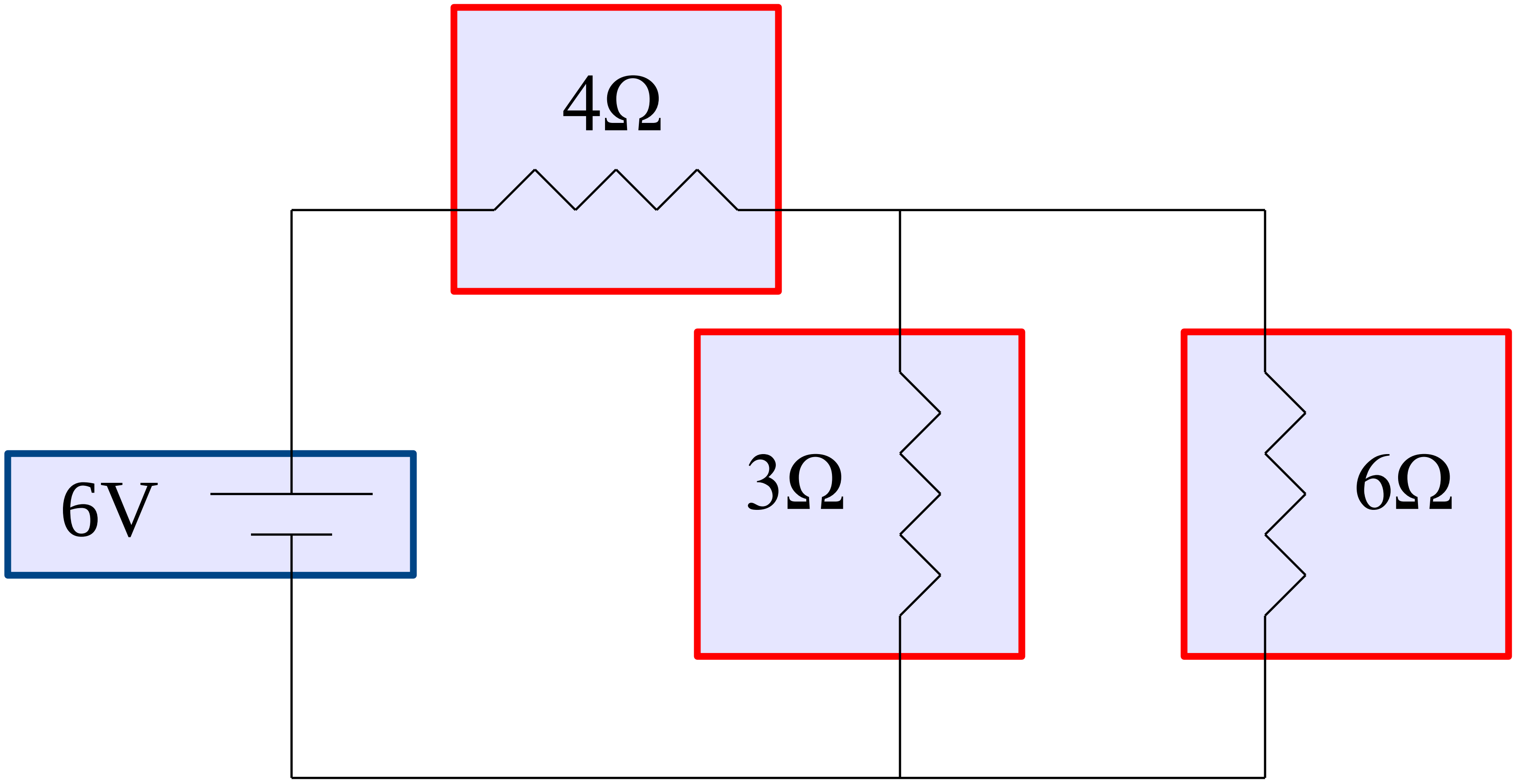
Time Line

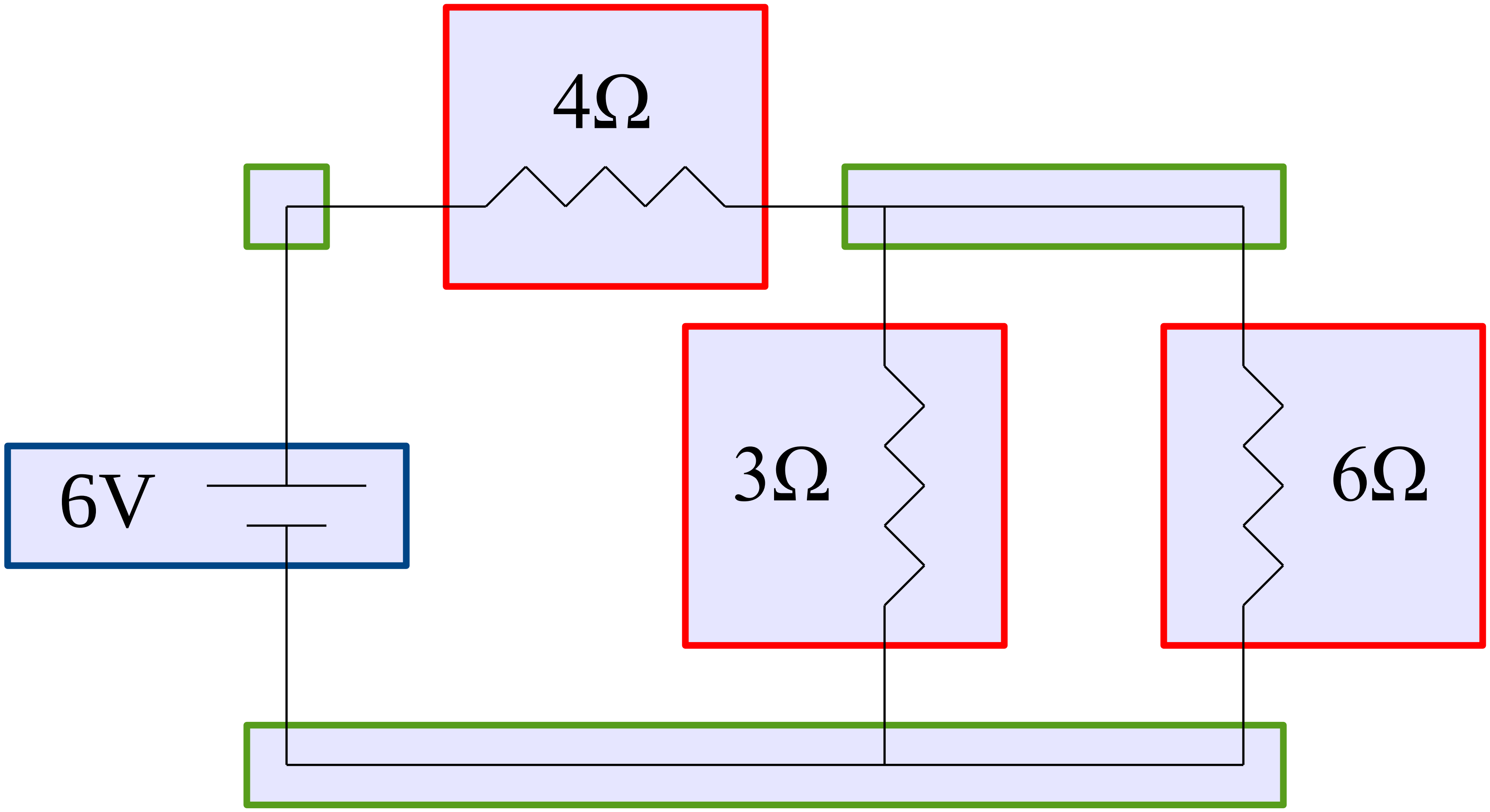
Outline

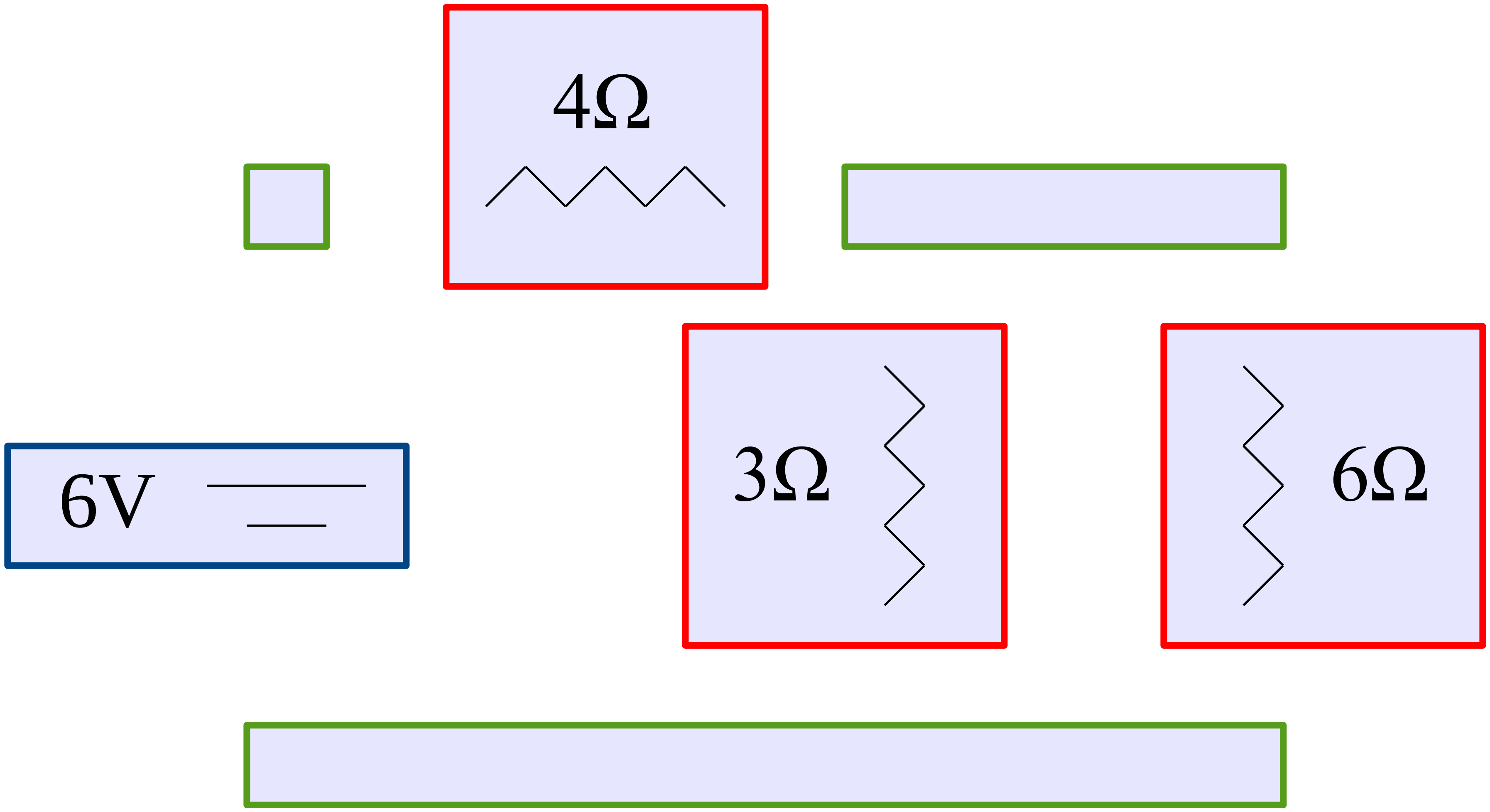
How does a compiler work?

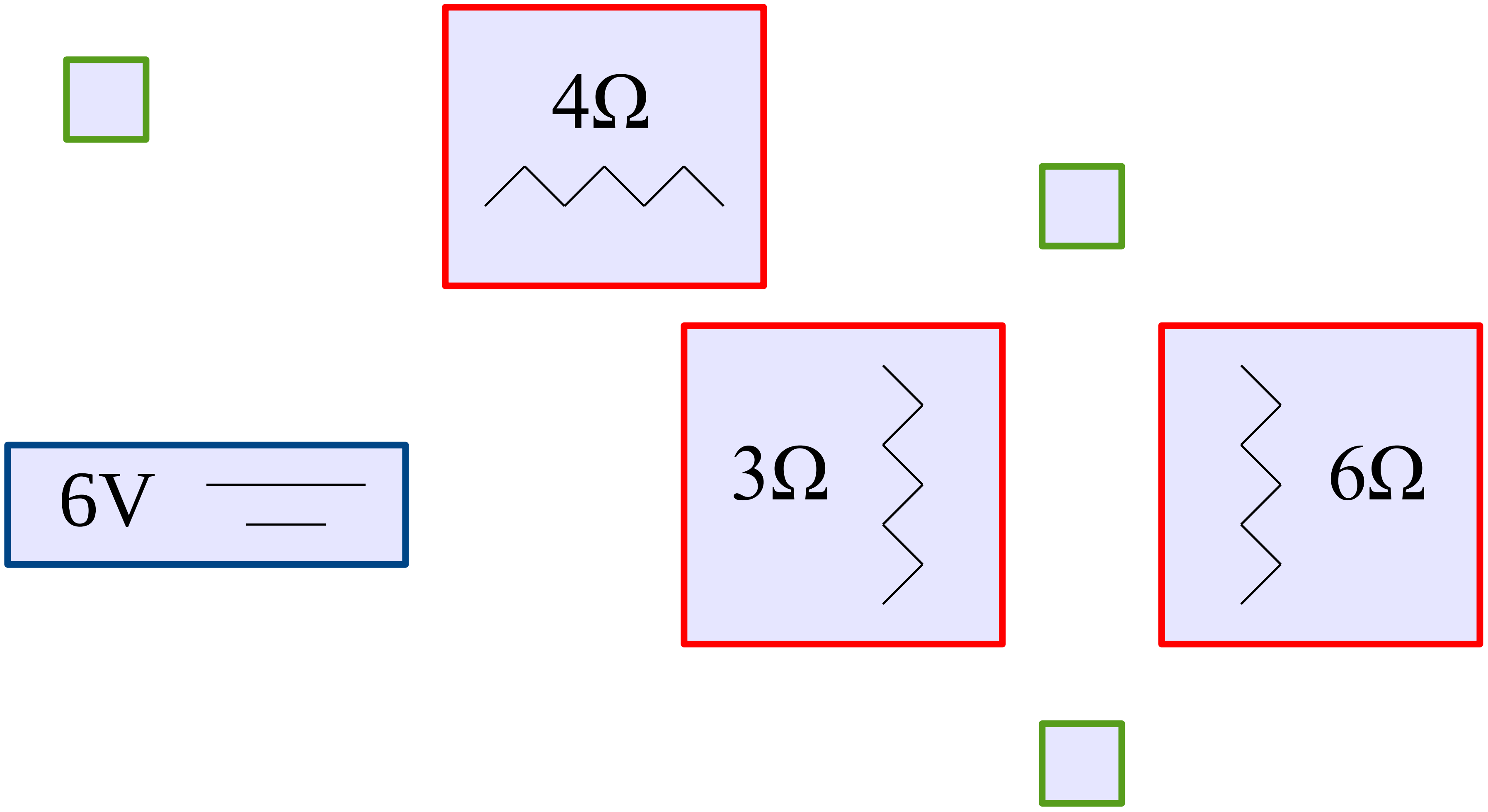

4Ω
6V
3Ω
6Ω

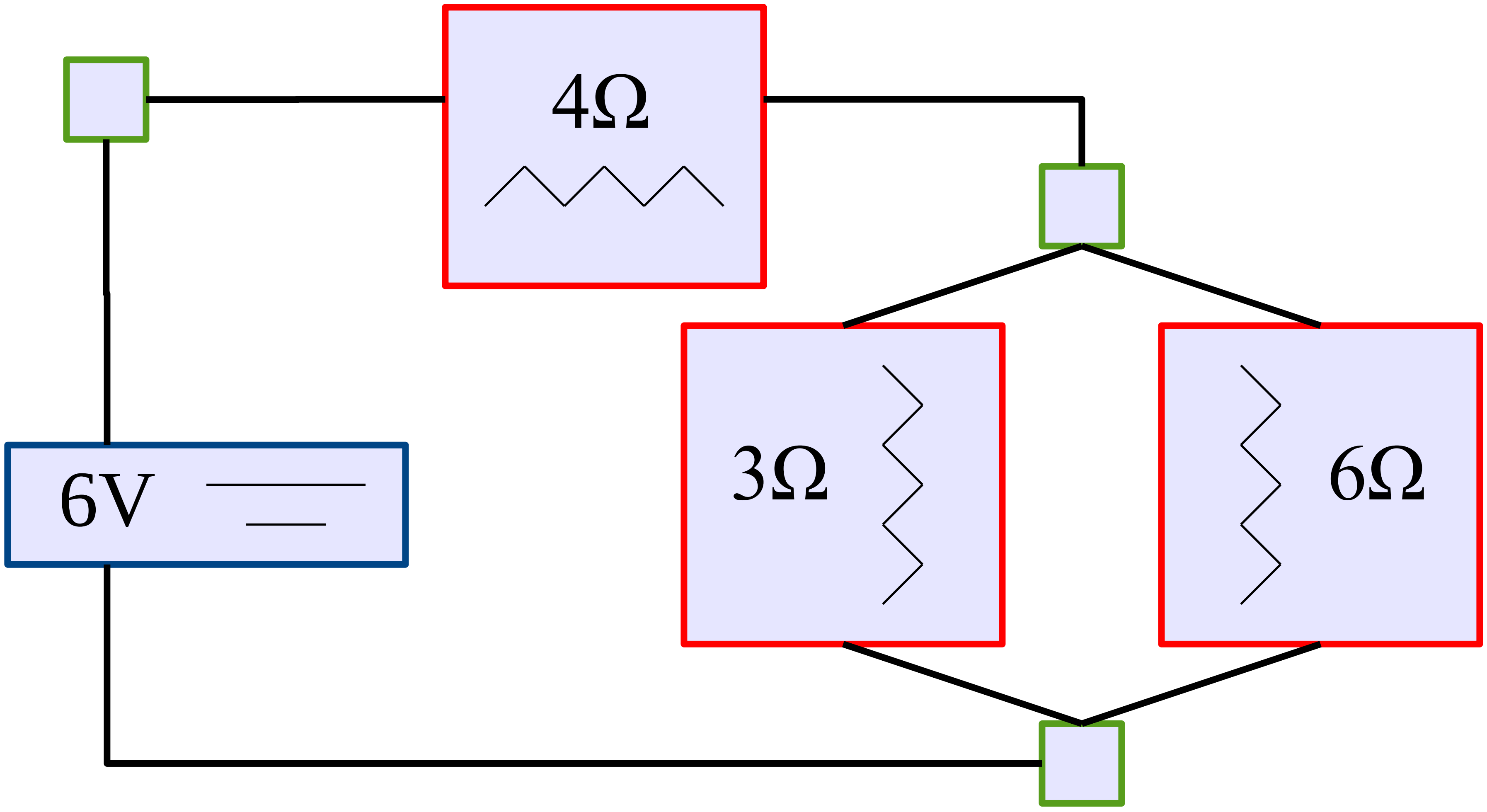

4Ω
6V
3Ω
6Ω

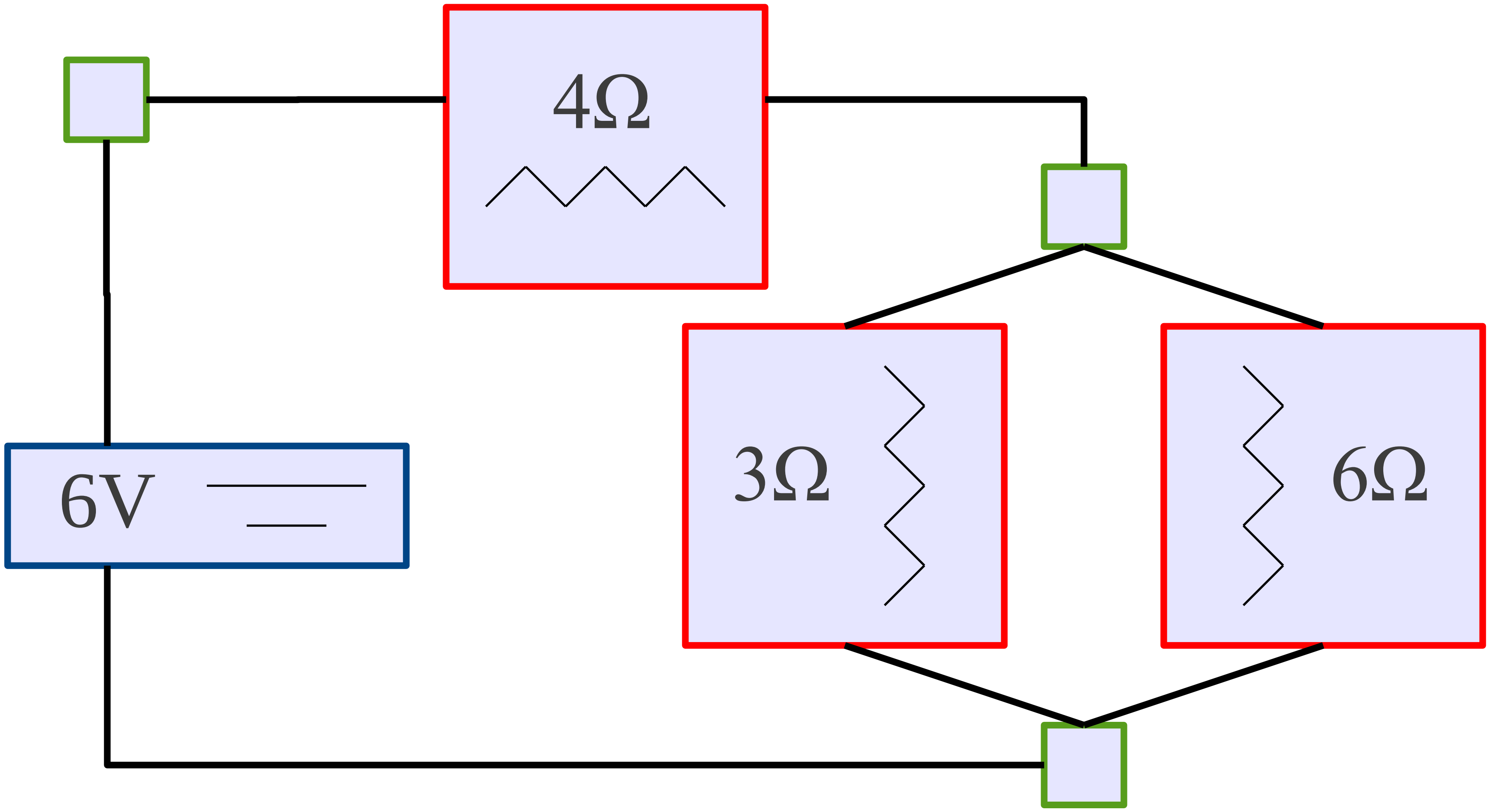

4Ω
6V
3Ω
6Ω


4Ω
6V
3Ω
6Ω

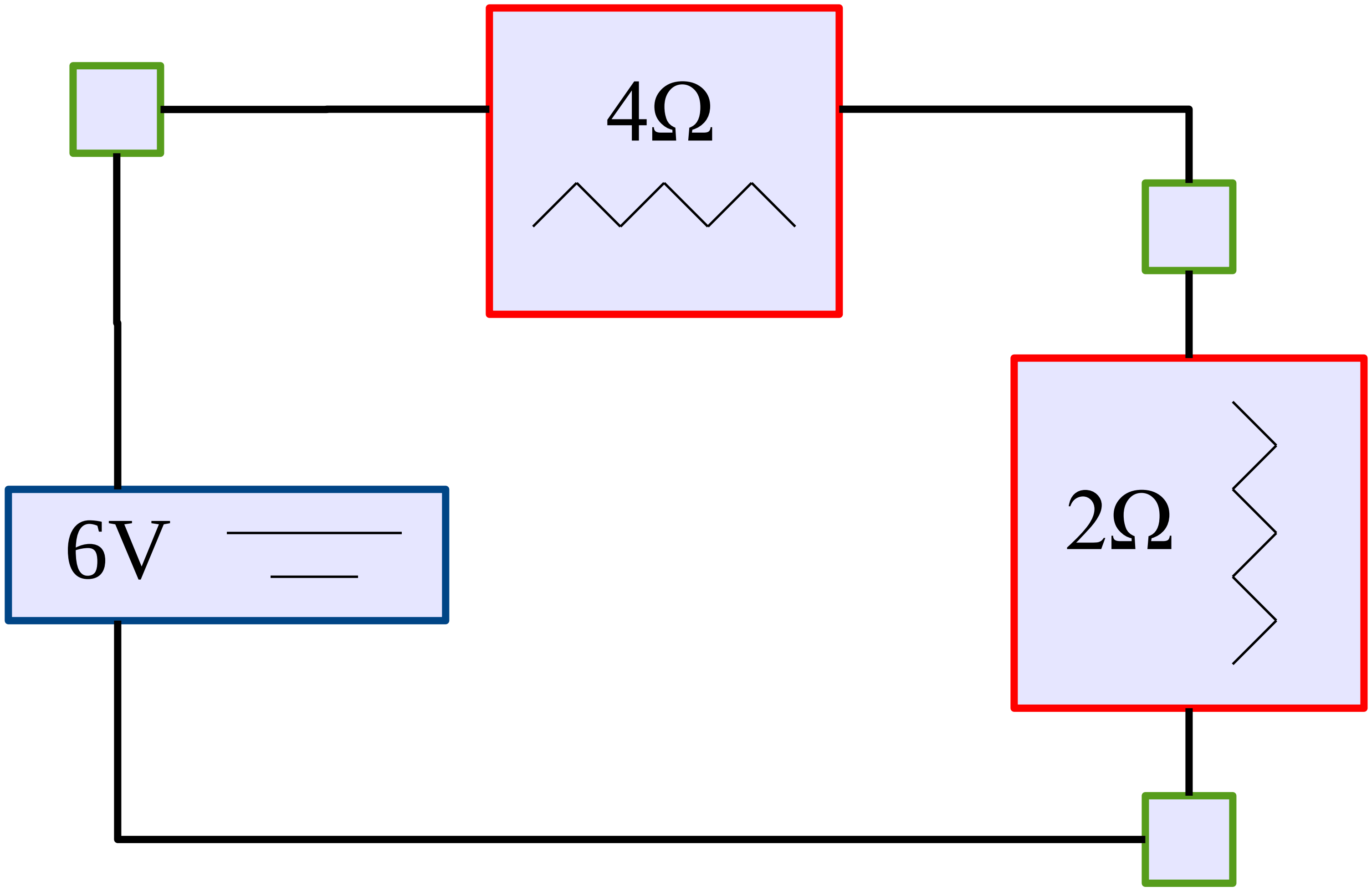

4Ω
3Ω
6Ω
6V


4Ω
6V
3Ω
6Ω

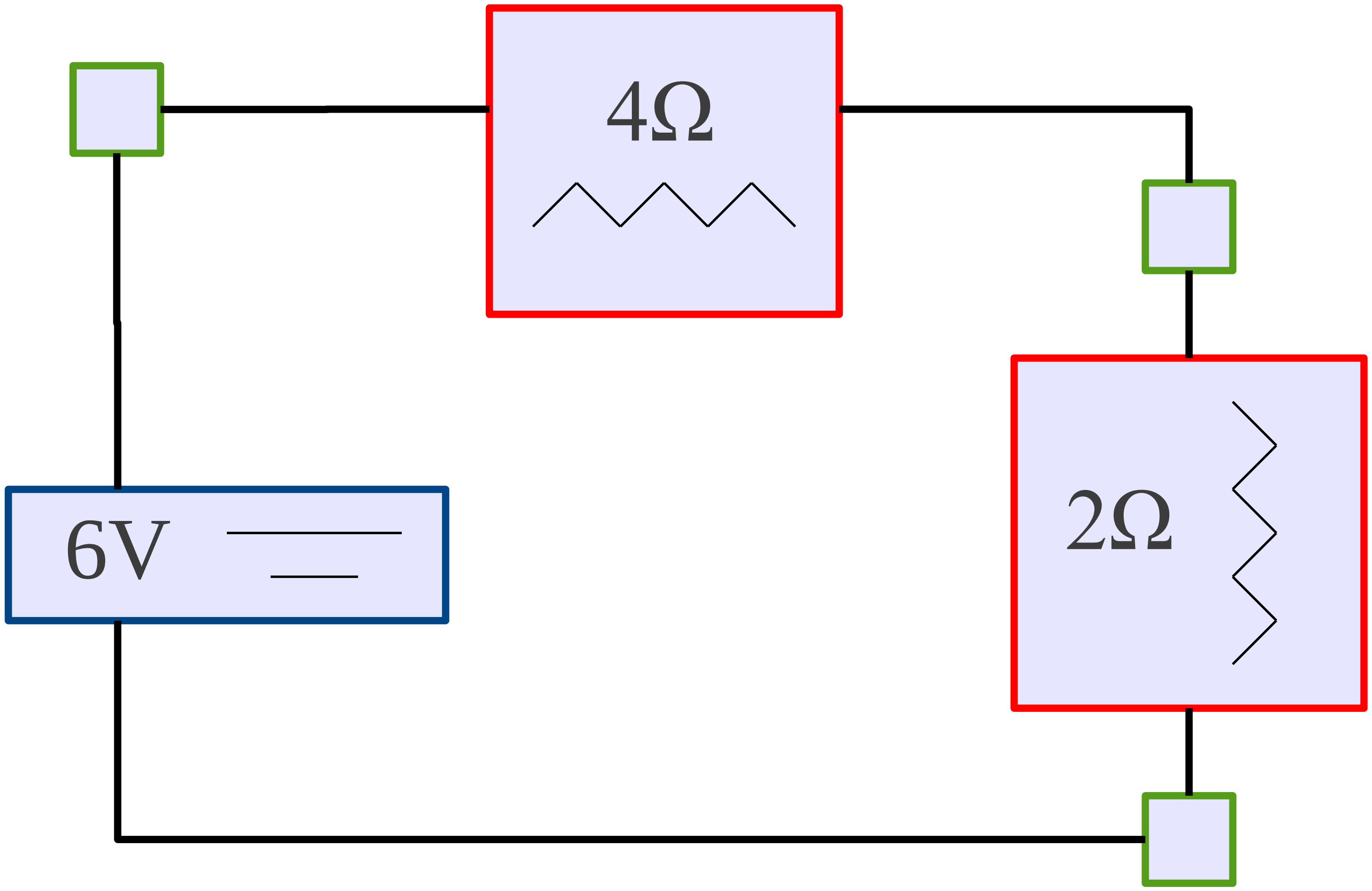

4Ω
6V
3Ω
6Ω

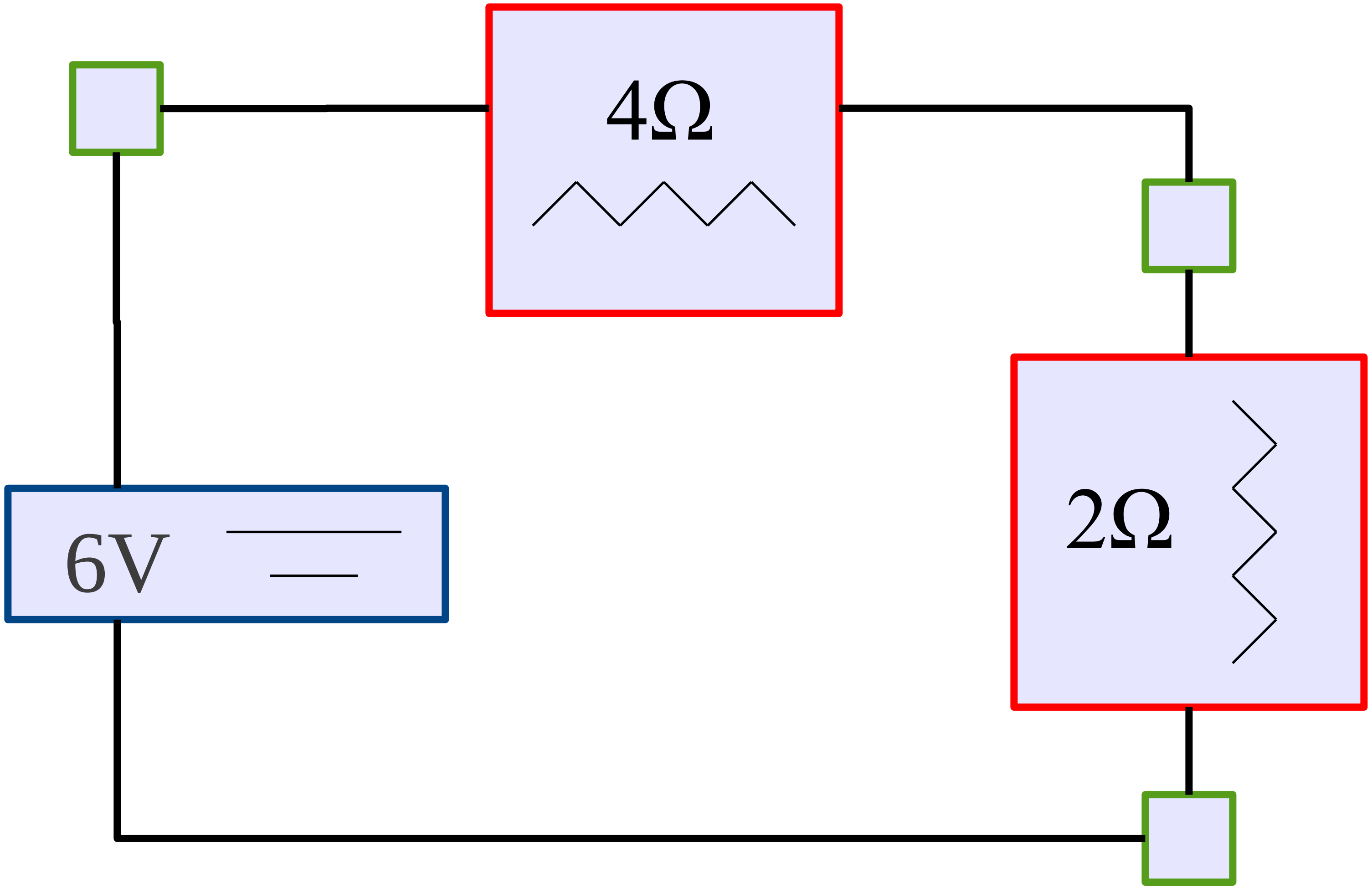


$$\frac{1}{\frac{1}{3\Omega}+\frac{1}{6\,\Omega}}=2\,\Omega$$

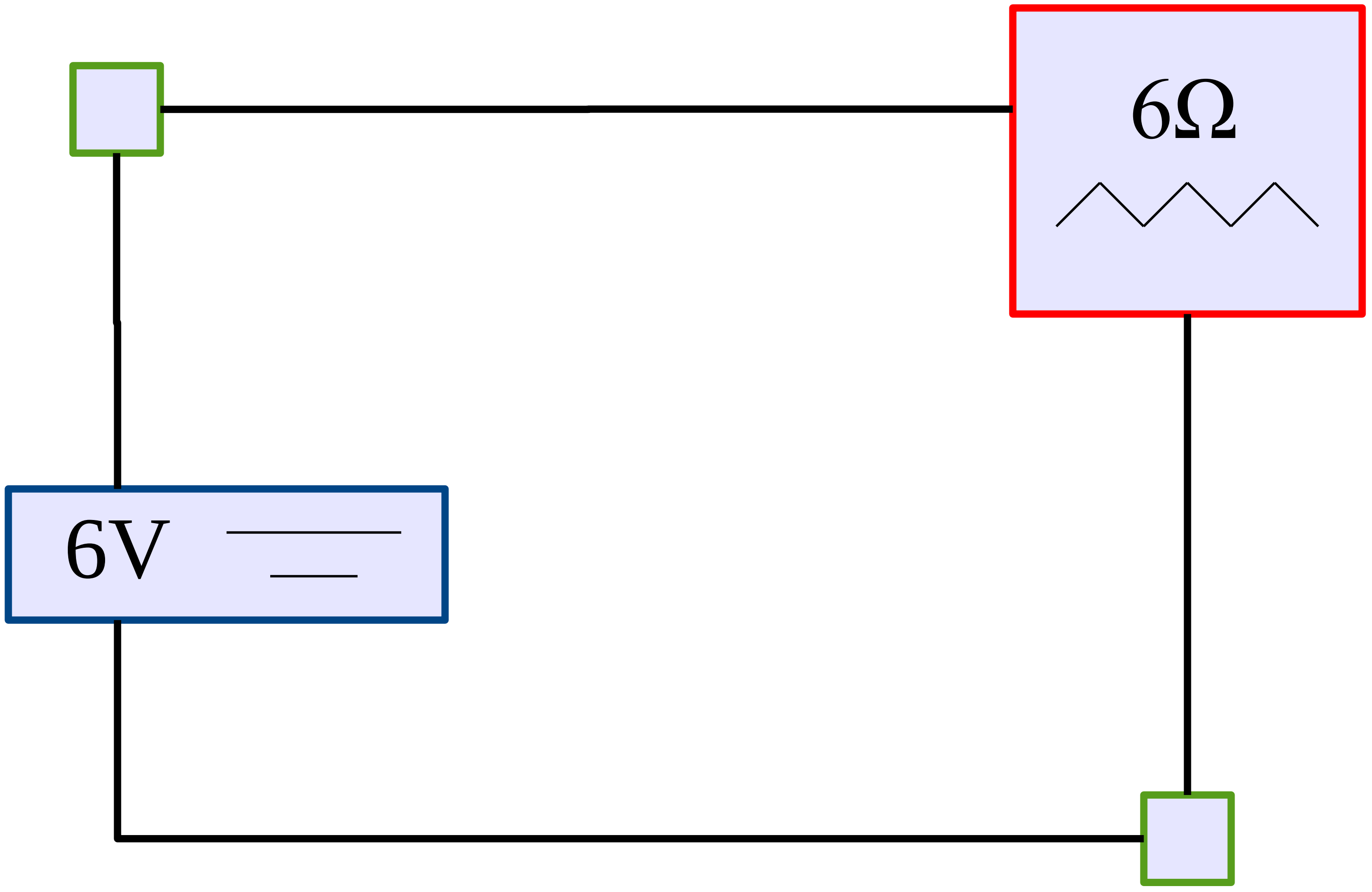


$$\frac{1}{\frac{1}{3\,\Omega}+\frac{1}{6\,\Omega}}=2\,\Omega$$

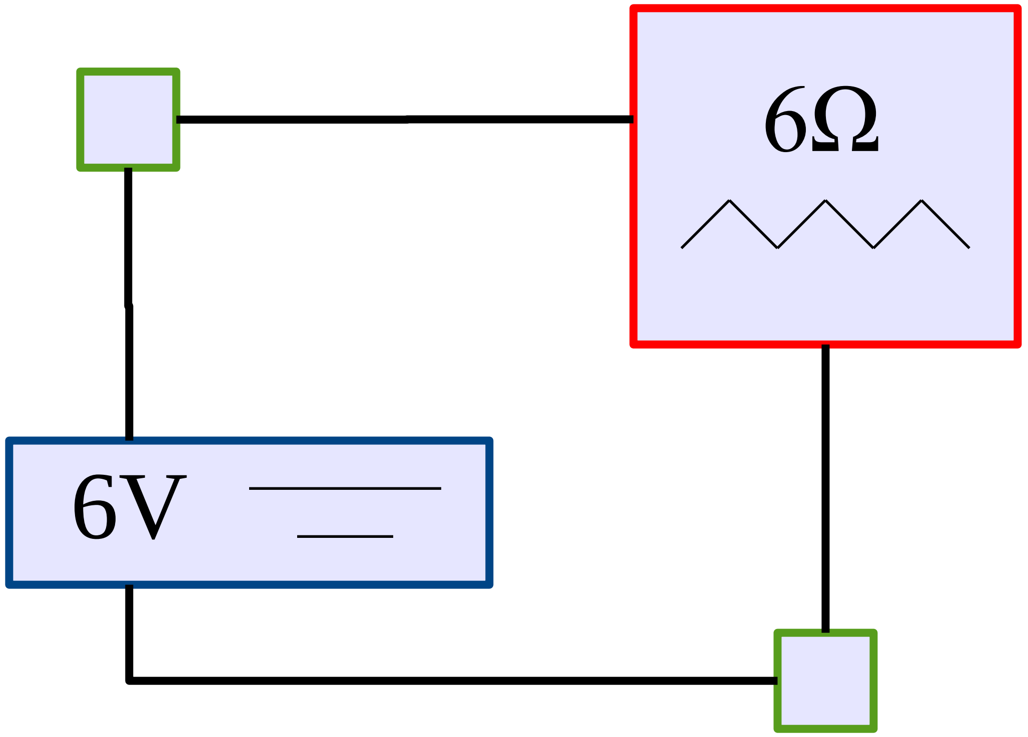

4Ω
2Ω
6V

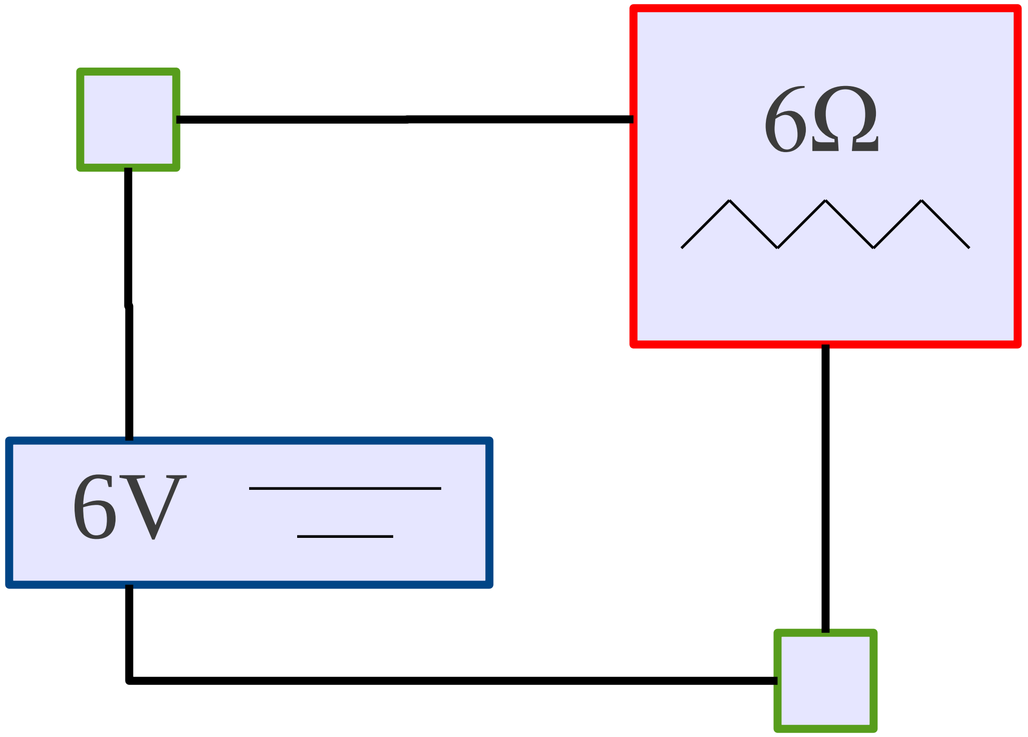
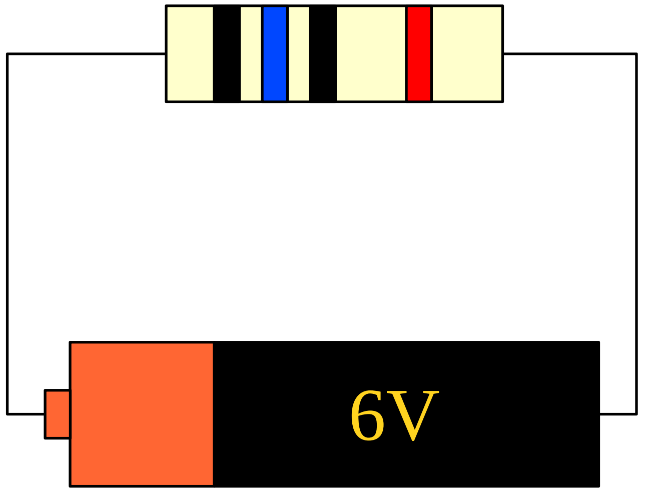


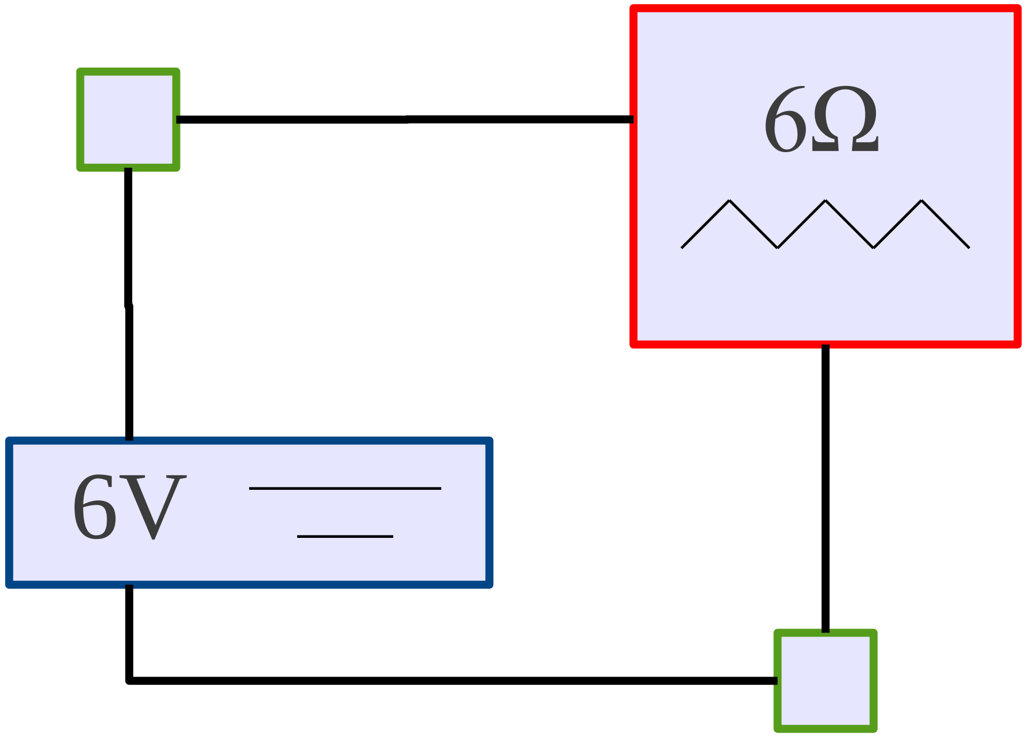
$4\,\Omega + 2\,\Omega = 6\,\Omega$

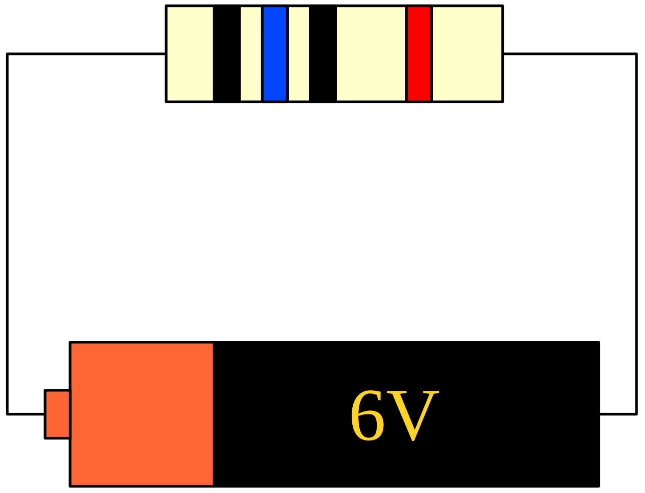


4 Ω + 2 Ω =6 Ω

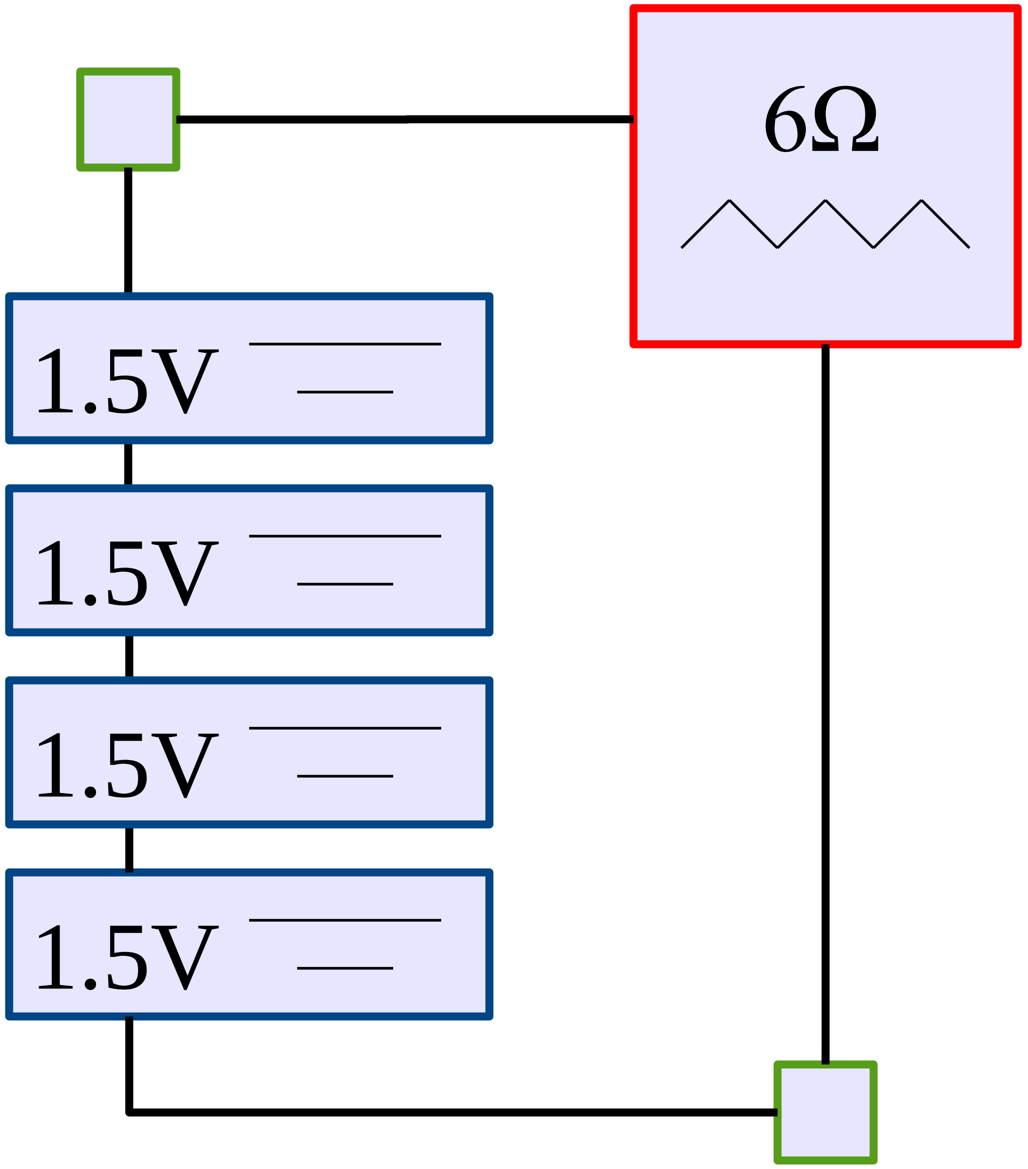

6Ω
6V

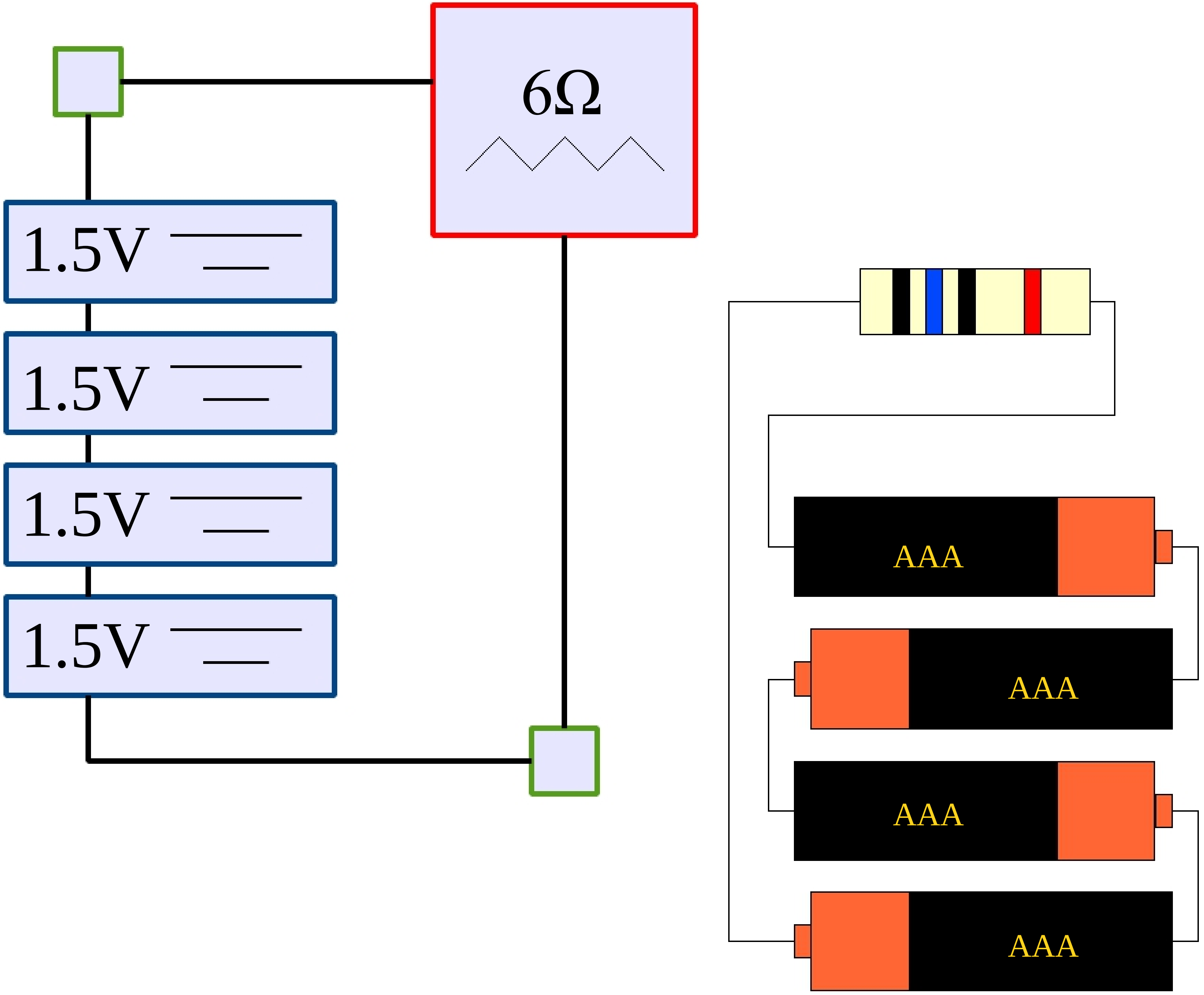

6Ω
6V

6V

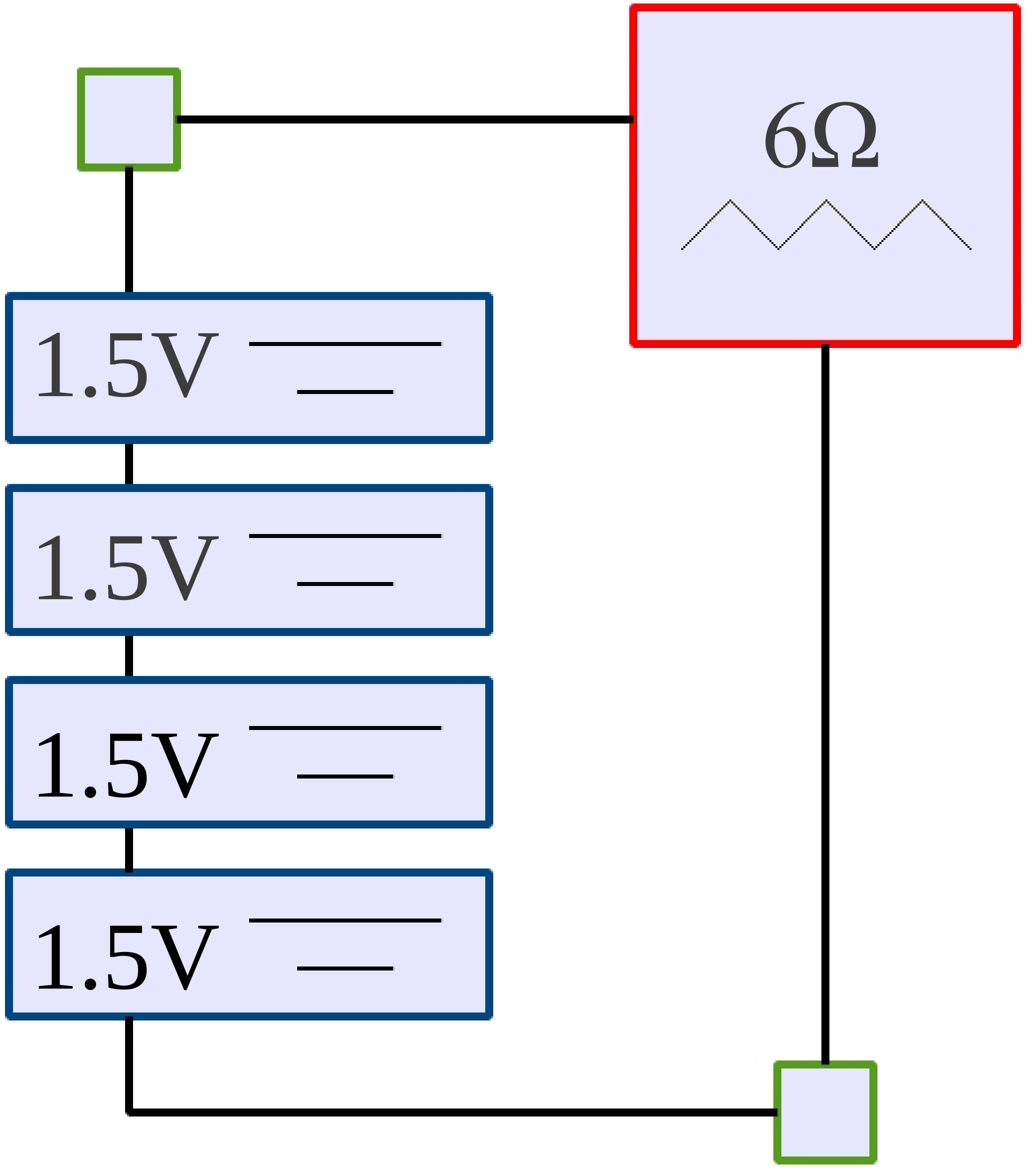


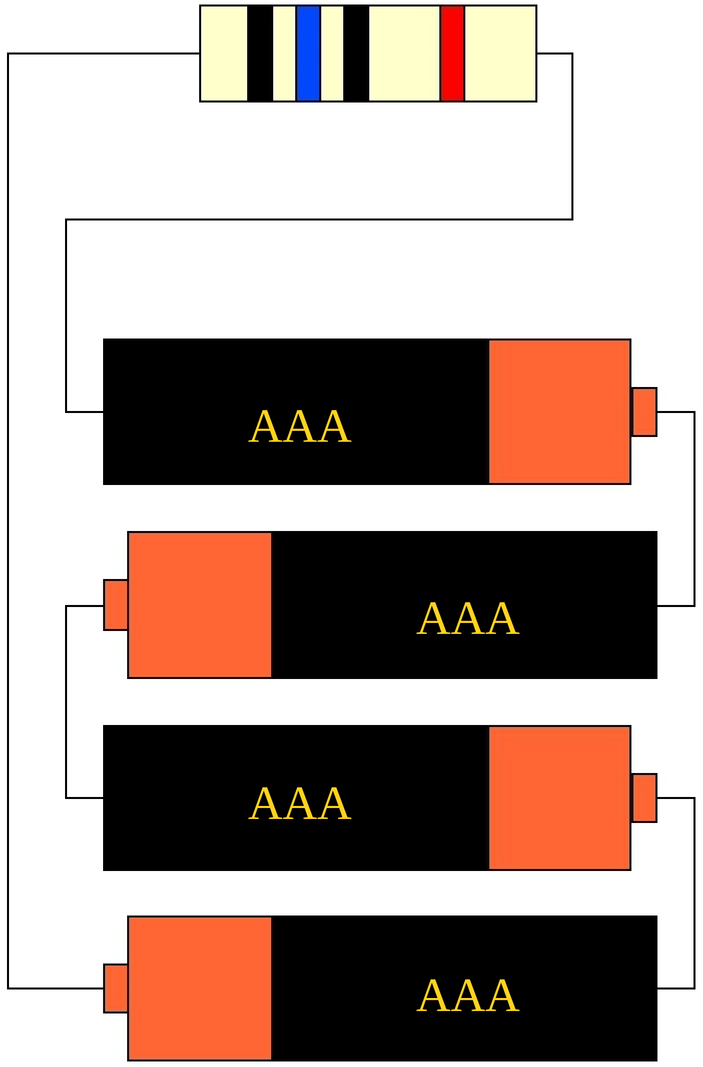


Total Cost: $4.75


6Ω
1.5V
1.5V
1.5V
1.5V


6Ω
1.5V
1.5V
1.5V
1.5V
AAA
AAA
AAA
AAA



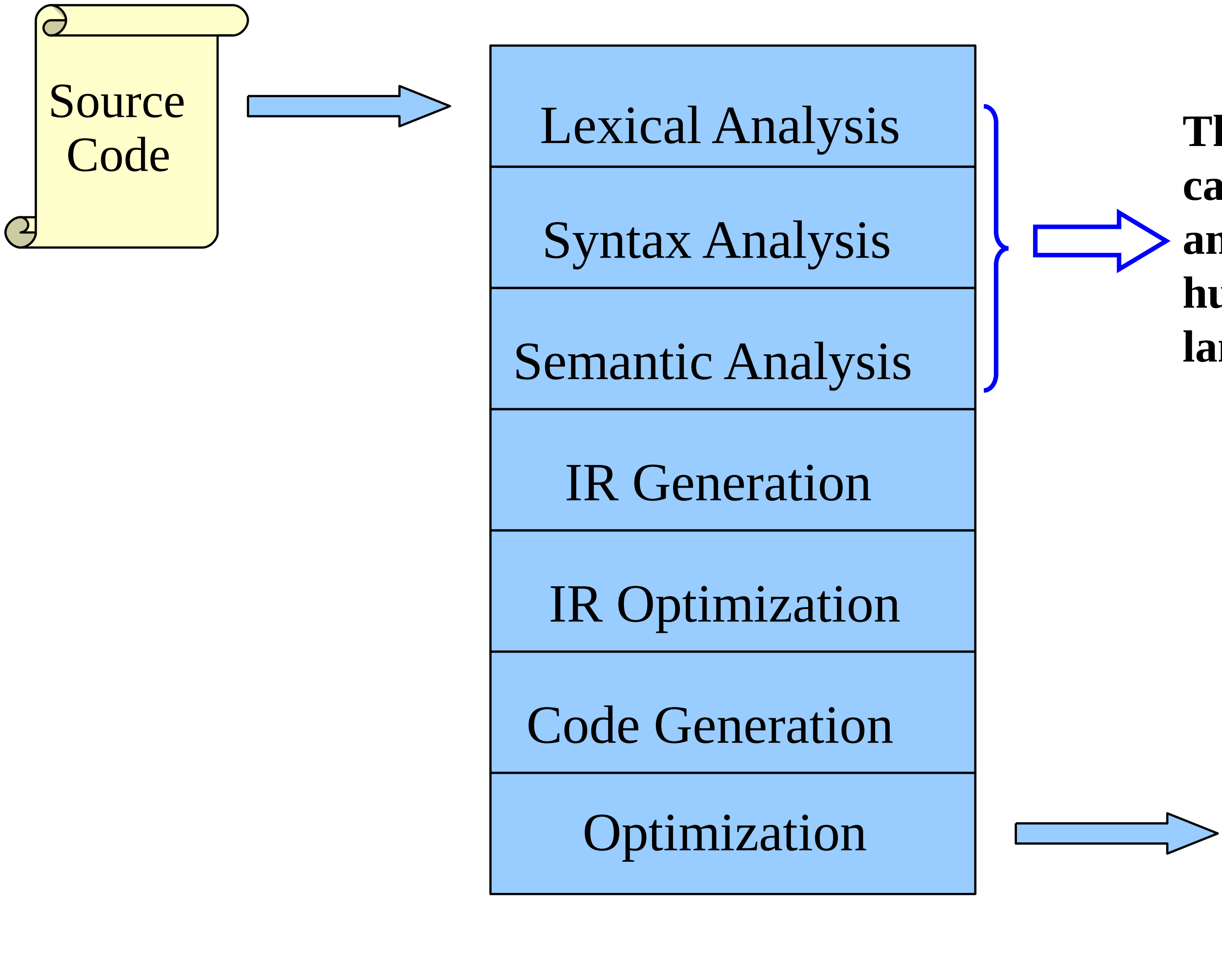


Total Cost: $1.00

From Description to Implementation

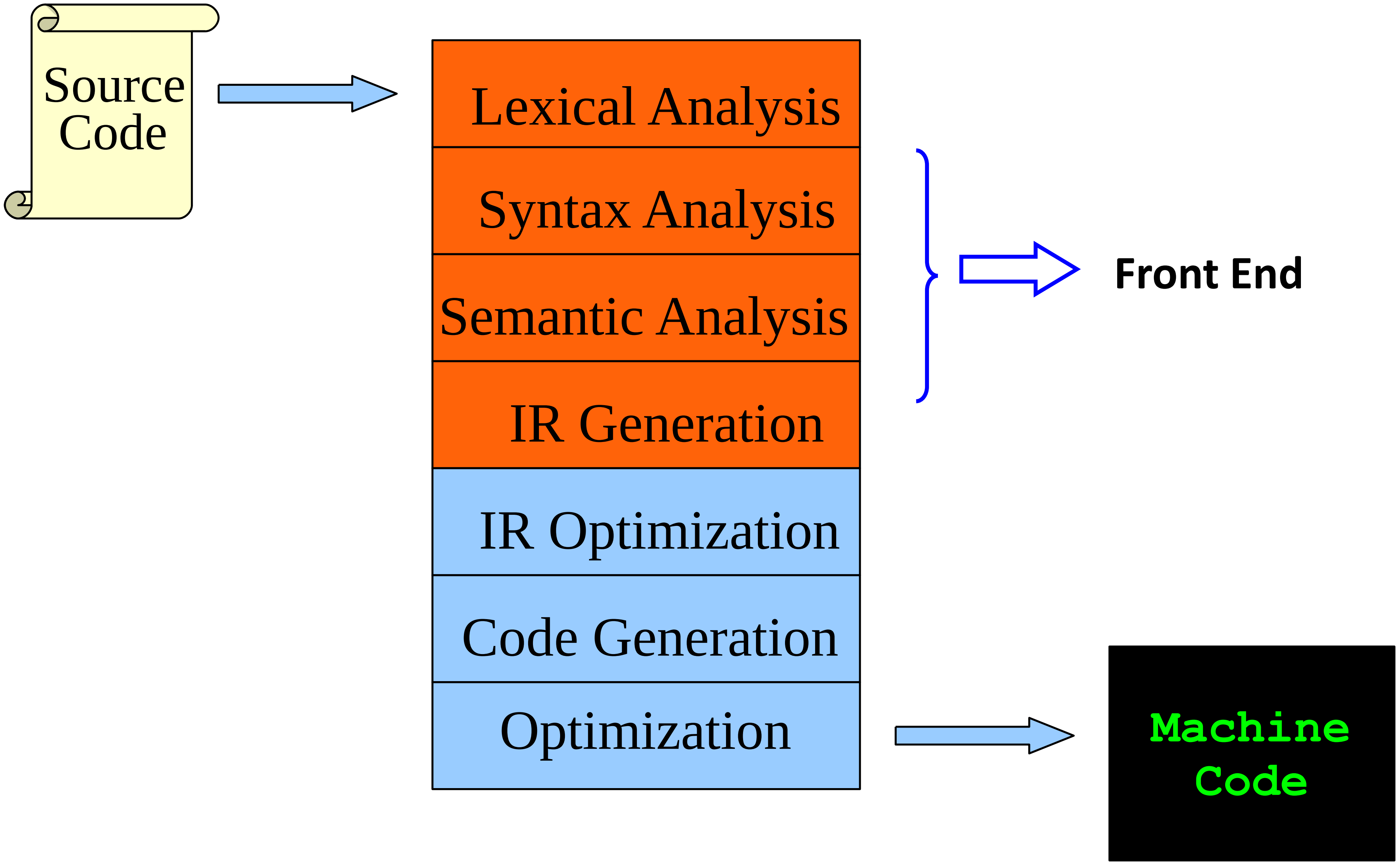
- **Lexical analysis (Scanning):** Identify logical pieces of the description.
- **Syntax analysis (Parsing):** Identify how those pieces relate to each other.
- **Semantic analysis:** Identify the meaning of the overall structure.
- **IR Generation:** Design one possible structure.
- **IR Optimization:** Simplify the intended structure.
- **Generation:** Fabricate the structure.
- **Optimization:** Improve the resulting structure.

The Structure of a Modern Compiler

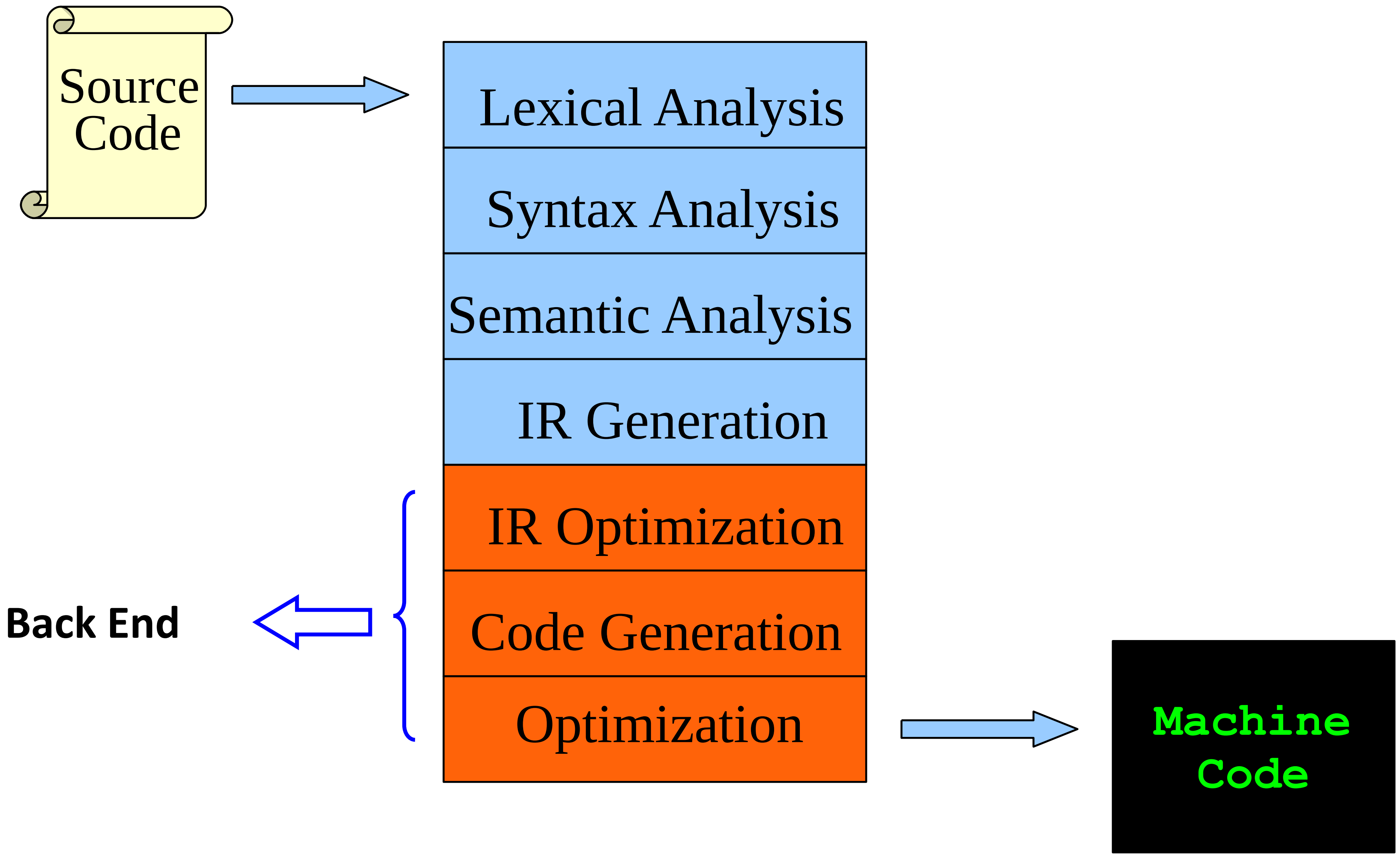


The first 3, at least, can be understood by analogy to how humans comprehend language.

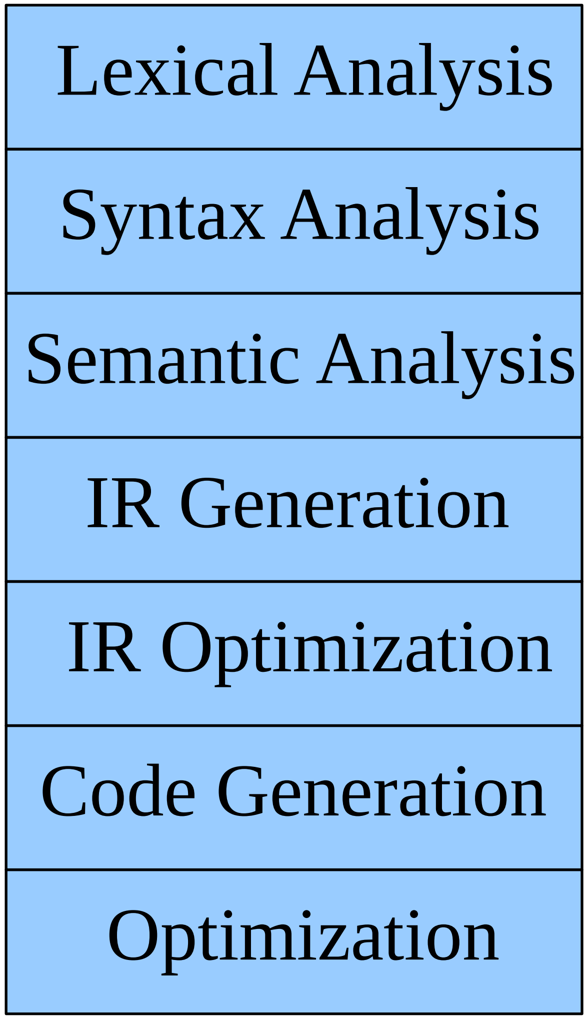
The Structure of a Modern Compiler



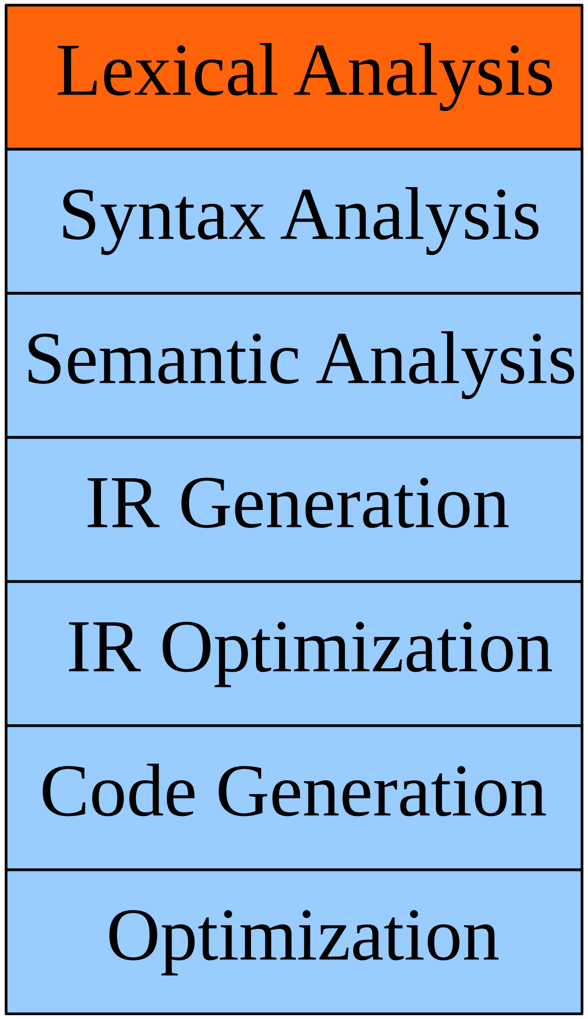
The Structure of a Modern Compiler



```
while (y < z) {
    int x = a + b;
    y += x;
}
```



```
while (y < z) {
    int x = a + b;
    y += x;
}
```



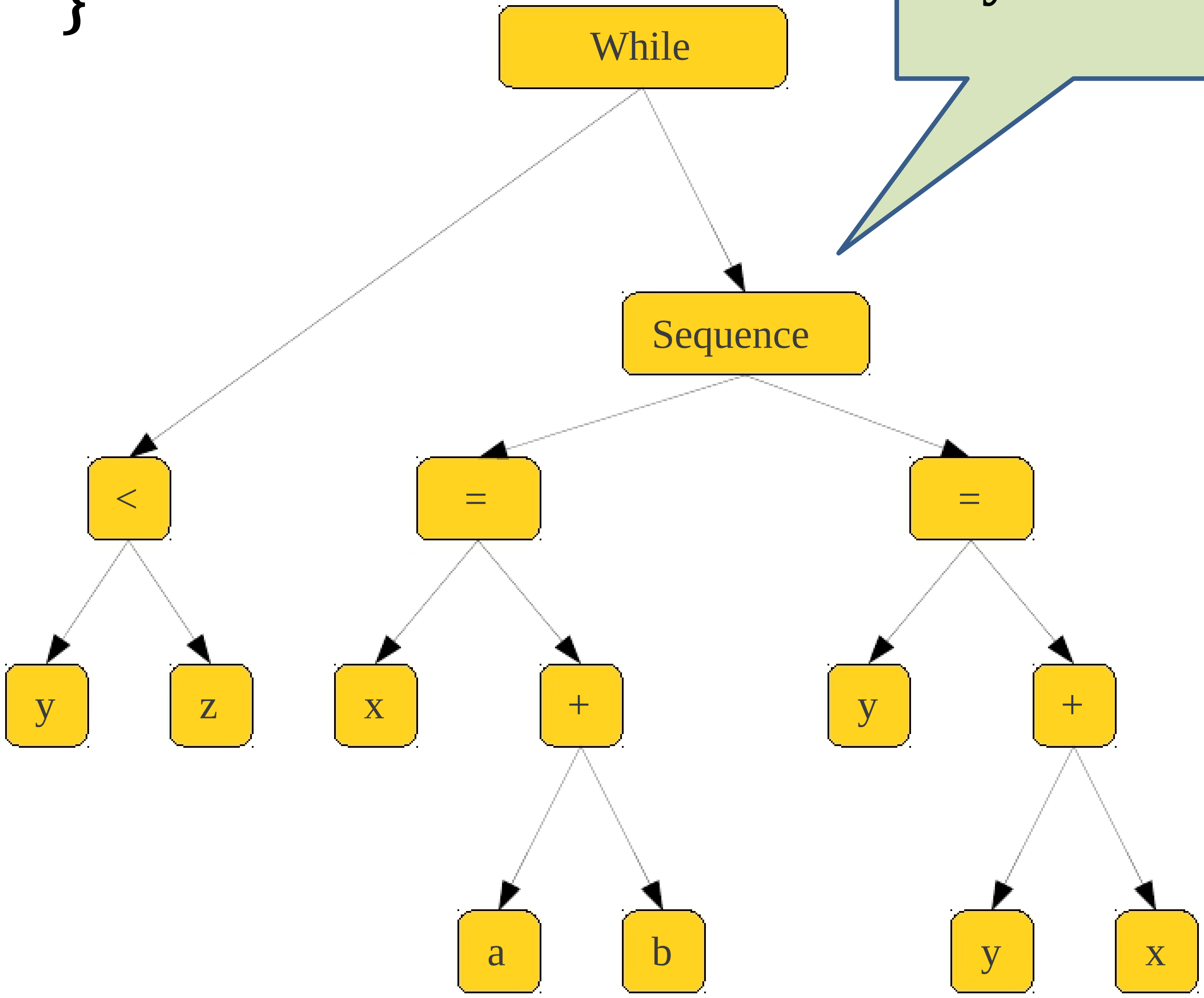
```
while (y < z) {
    int x = a + b;
    y += x;
}
```

```
T_While
T_LeftParen
T_Identifier y
T_Less
T_Identifier z
T_RightParen
T_OpenBrace
T_Int
T_Identifier x
T_Assign
T_Identifier a
T_Plus
T_Identifier b
T_Semicolon
T_Identifier y
T_PlusAssign
T_Identifier x
T_Semicolon
T_CloseBrace
```

Tokens

- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- IR Generation
- IR Optimization
- Code Generation
- Optimization

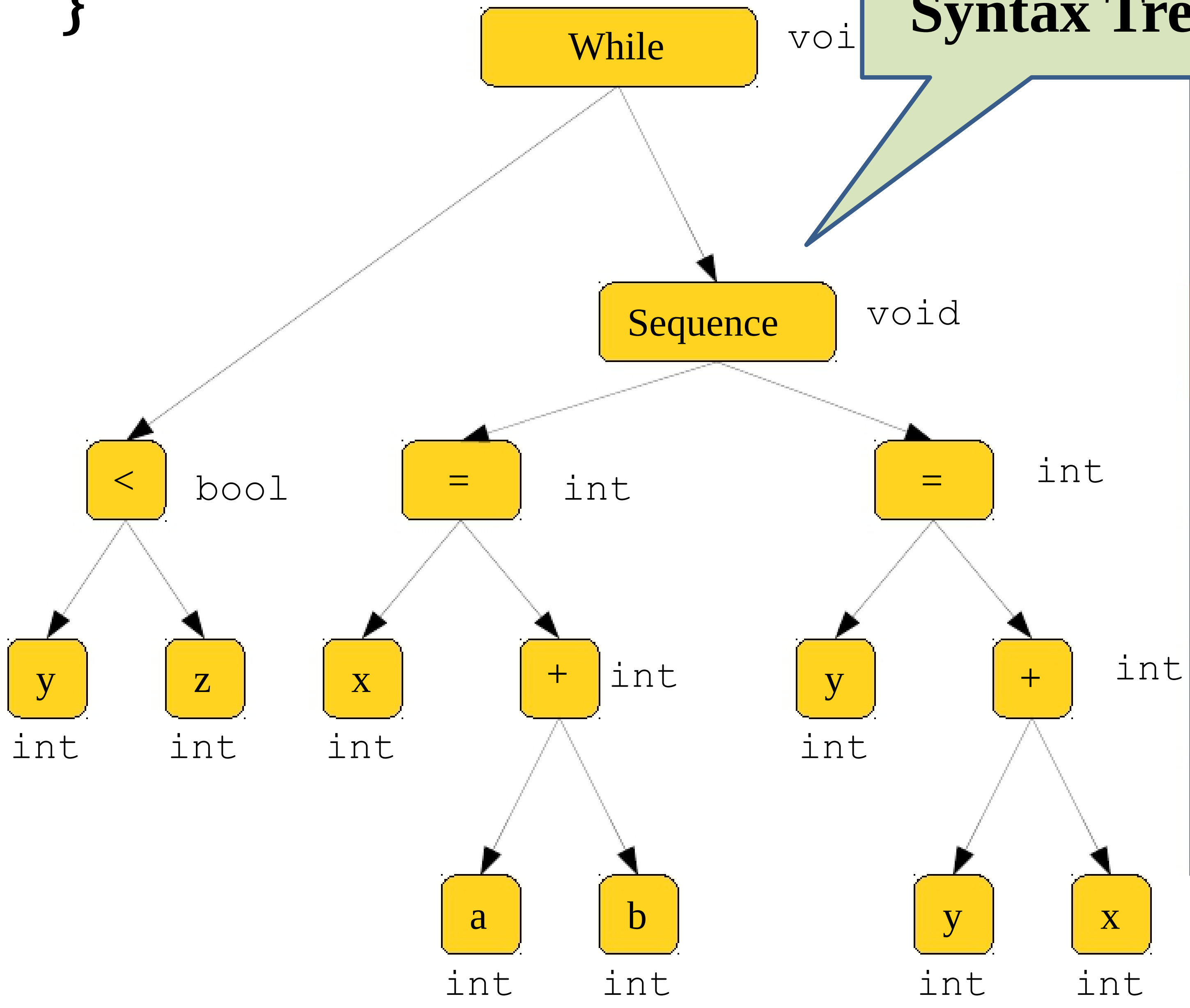
```
while (y < z) {
    int x = a + b;
    y += x;
}
```



Syntax Tree

Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization

```
while (y < z) {
    int x = a + b;
    y += x;
}
```



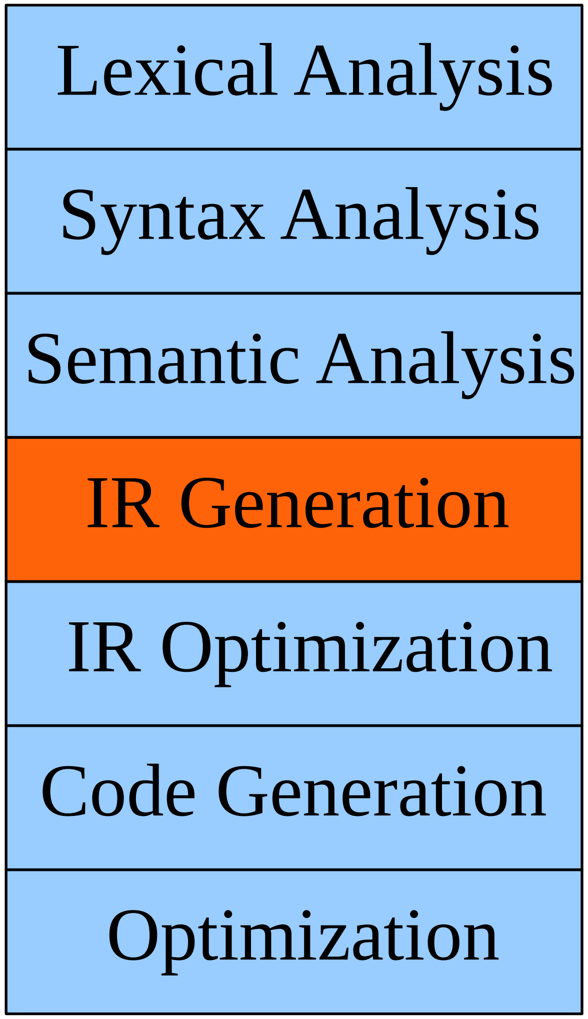
Annotated Syntax Tree

- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- IR Generation
- IR Optimization
- Code Generation
- Optimization

```
while (y < z) {
    int x = a + b;
    y += x;
}
```

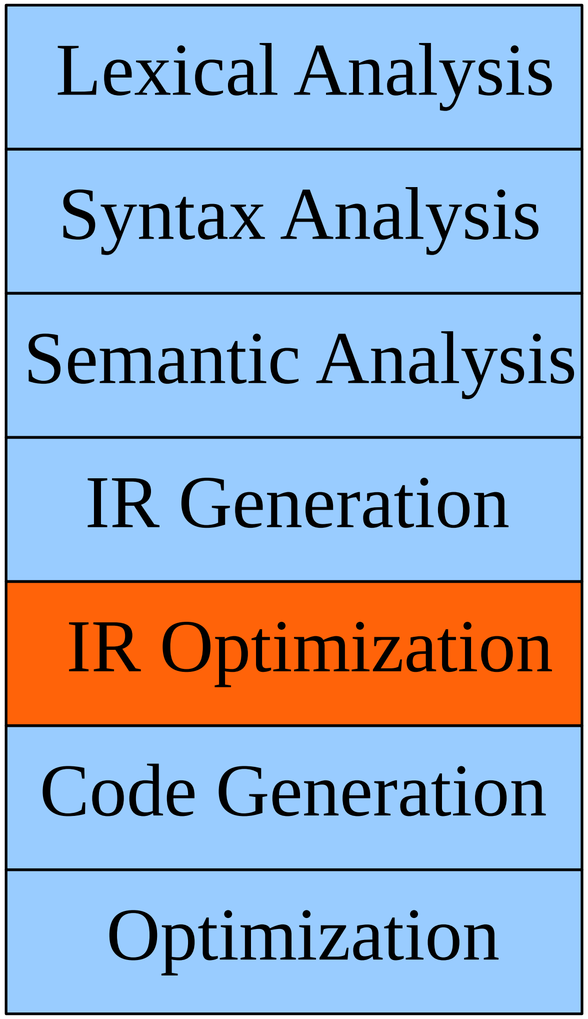
```
Loop:   x       = a + b
        y       = x + y
        _t1     = y < z
        if _t1  goto  Loop
```

Intermediate code



```
while (y < z) {
    int x = a + b;
    y += x;
}
```

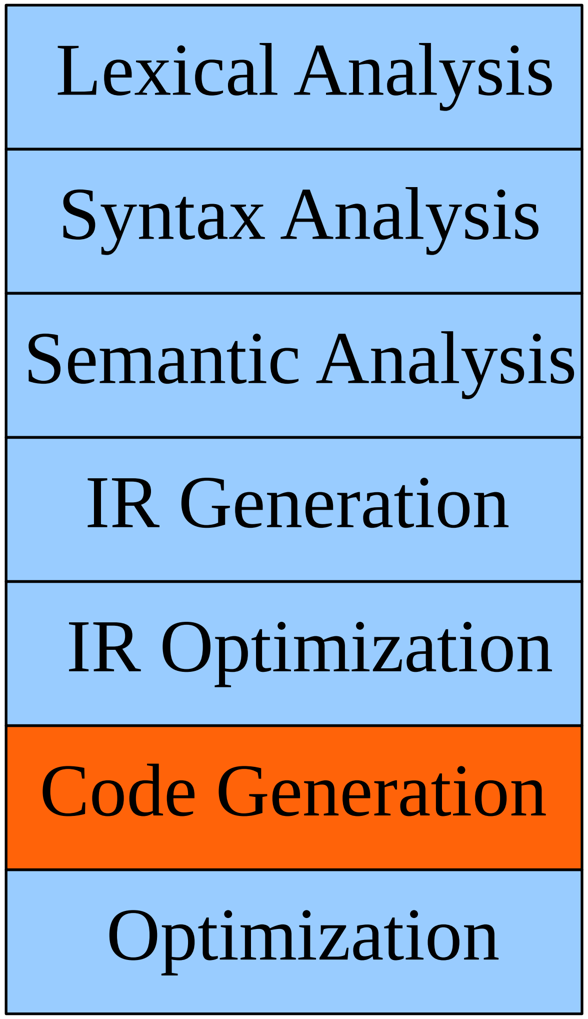
```
           x      = a + b
Loop:      y      = x + y
           _t1    = y < z
           if _t1  goto  Loop
```



```
while (y < z) {
    int x = a + b;
    y += x;
}
```

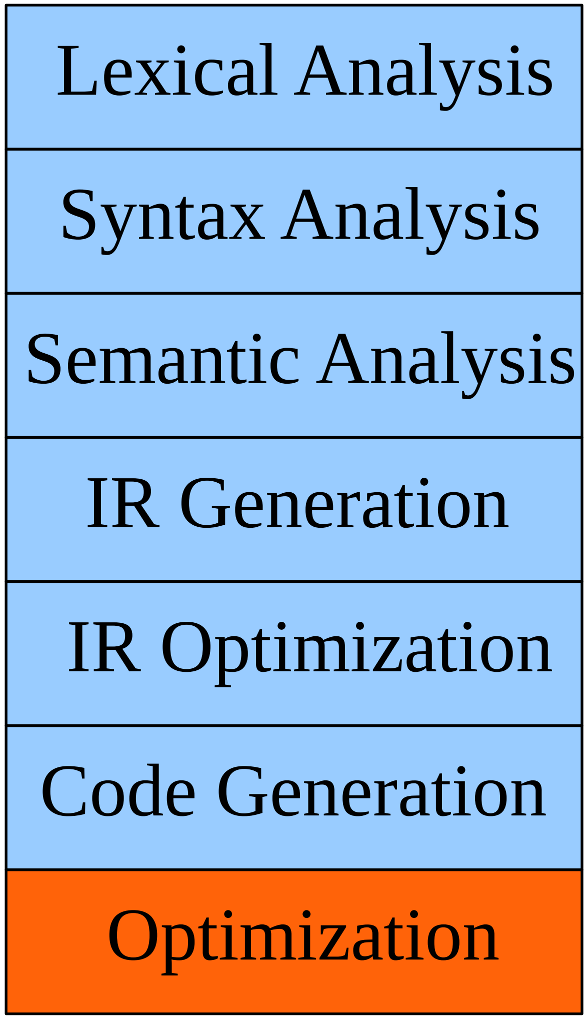
```
         add   $1,   $2,  $3
Loop:    add   $4,   $1,  $4
         slt   $6,   $4,  $5
         beq   $6,   loop
```

Target code



```
while (y < z) {
    int x = a + b;
    y += x;
}
```

```
        add  $1, $2, $3
Loop:   add  $4, $1, $4
        blt  $4, $5, loop
```



Outline

1 Preface

2 Language Process

3 The Evolution of Programming Languages

4 The Structure of a Compiler

5 Other Issues in Compiler Structure

About Cross Compiler

Appendix for lecture1

什么是交叉编译？

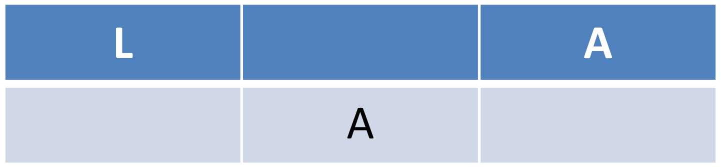
- 在一个平台上生成另一个平台上的可执行代码。
  - 这里需要注意的是所谓平台，实际上包含两个概念：体系结构（Architecture）、操作系统（Operating System）。同一个体系结构可以运行不同的操作系统；同样，同一个操作系统也可以在不同的体系结构上运行。
  - 举例来说，我们常说的x86 Linux平台实际上是Intel x86体系结构和Linux for x86操作系统的统称；而x86 WinNT平台实际上是Intel x86体系结构和Windows NT for x86操作系统的简称。

- “既然我们已经有了主机编译器，那为什么还要交叉编译呢？”
  - 有时是因为目的平台上不允许或不能够安装我们所需要的编译器，而我们又需要这个编译器的某些特征；有时是因为目的平台上的资源贫乏，无法运行我们所需要编译器；有时又是因为目的平台还没 有建立，连操作系统都没有，根本谈不上运行什么编译器。

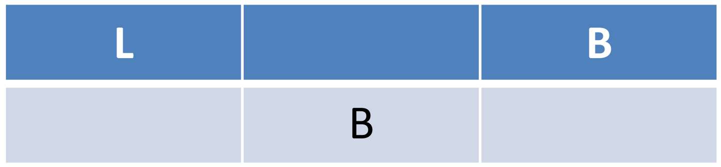
- “既然可以交叉编译，那还要主机编译干吗？”
  - 交叉编译是不得已而为之！与主机编译相比，交叉编译受的限制更多，虽然在理论上我们可以做任何形式的交叉编 译，但事实上，由于受到专利、版权、技术的限制，并不总是能够进行交叉编译，尤其是在业余条件下！
  - 举 例来说，我们至今无法生成惠普公司专有的som格式的可执行文件，因此我们根本无法做目的平台为 HPPA-HPUX的交叉编译。

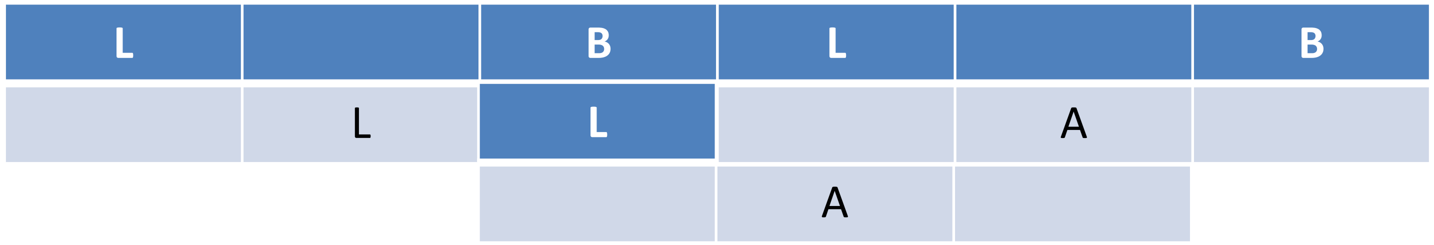
- iPhone的SDK只能在OSX上运行，这也是没办法的事情，因为Xcode从来没有被移植到OSX之外的地方去，就好像Visual C++从来都没有过*nix版本，如果有人要写windows CE的应用，多半也得装个windows
- 但是在Linux/windows上有人做了基于GNU Objective-C的交叉编译环境，有不少在iPhone SDK发布之前的软件就是这么做出来的。
  - 1、OS/X并不是只能用object-c开发程序，gcc编译器中支持的语言在OS/x下都可以，就不用说C了。
  - 2、Xcode是OS/X下的集成开发环境，和Windows下的VS类似。
  - 3、OS/X操作系统的底层内核是开源的（ Darwin是苹果机的操作系统OS X的基础“核心”，其中结合了两大著名的程序，Mach 内核和Berkeley BSD4.4 ）

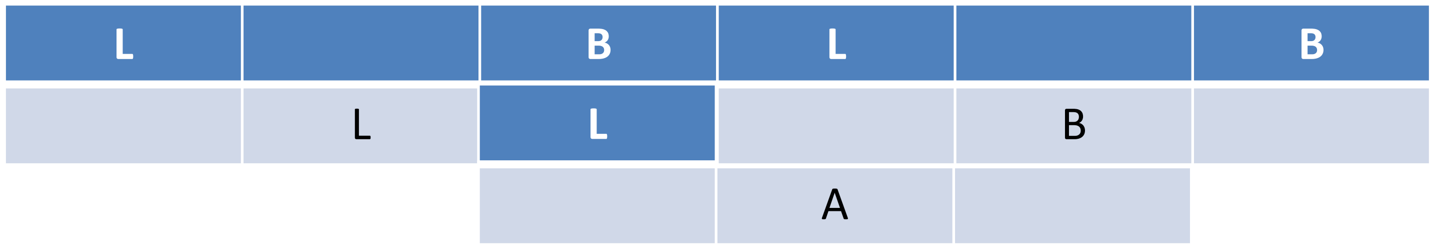
已存在A平台上的L语言编译器，要求生产一个B平台上的L语言编译器。如何做？



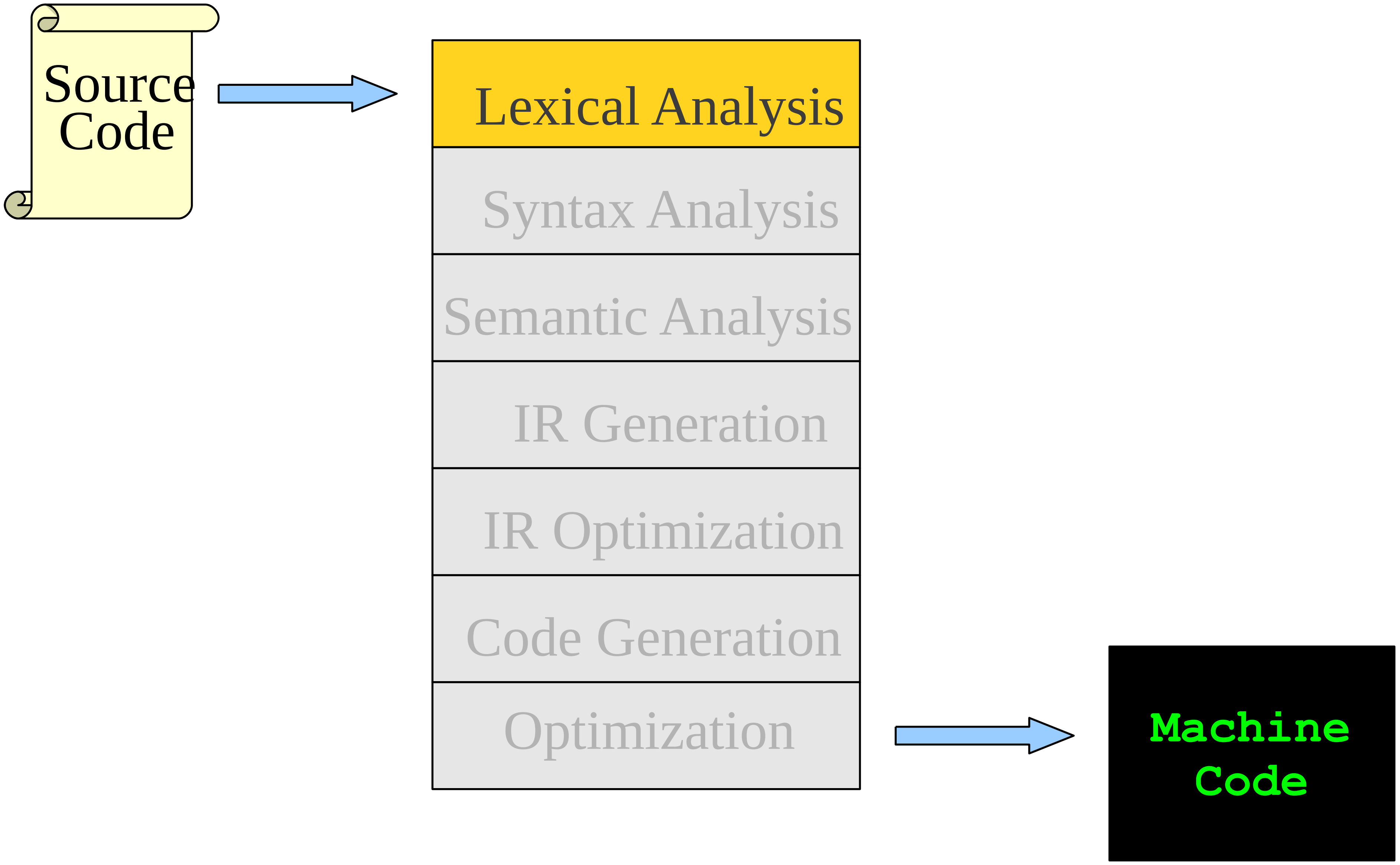
to?







Next Time...


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code