Welcome to CE414: Compilers

- Course Information
- Why Study Compilers?
- A Quick History of Compilers
- The Structure of a Compiler

Course Staff

- **Instructor**: Samane Hosseinmardi (s.hosseinmardi.sharif@gmail.com)

M.Sc: Computer Science, intelligent System

Bs: Computer Engineer, Software

Course Staff

Majid Taherkhani

[Head TA]

(majidtaherkhani55@gmail.com)

A Note on Communication

- We use Quera. @ Here
- Assignments and project must be uploaded there.
- Ask your questions! TAs will be happy to be able to help.
- Lectures will be on CW and course page.
- Telegram Channel for Notification: Link

Telegram

A Note on Communication

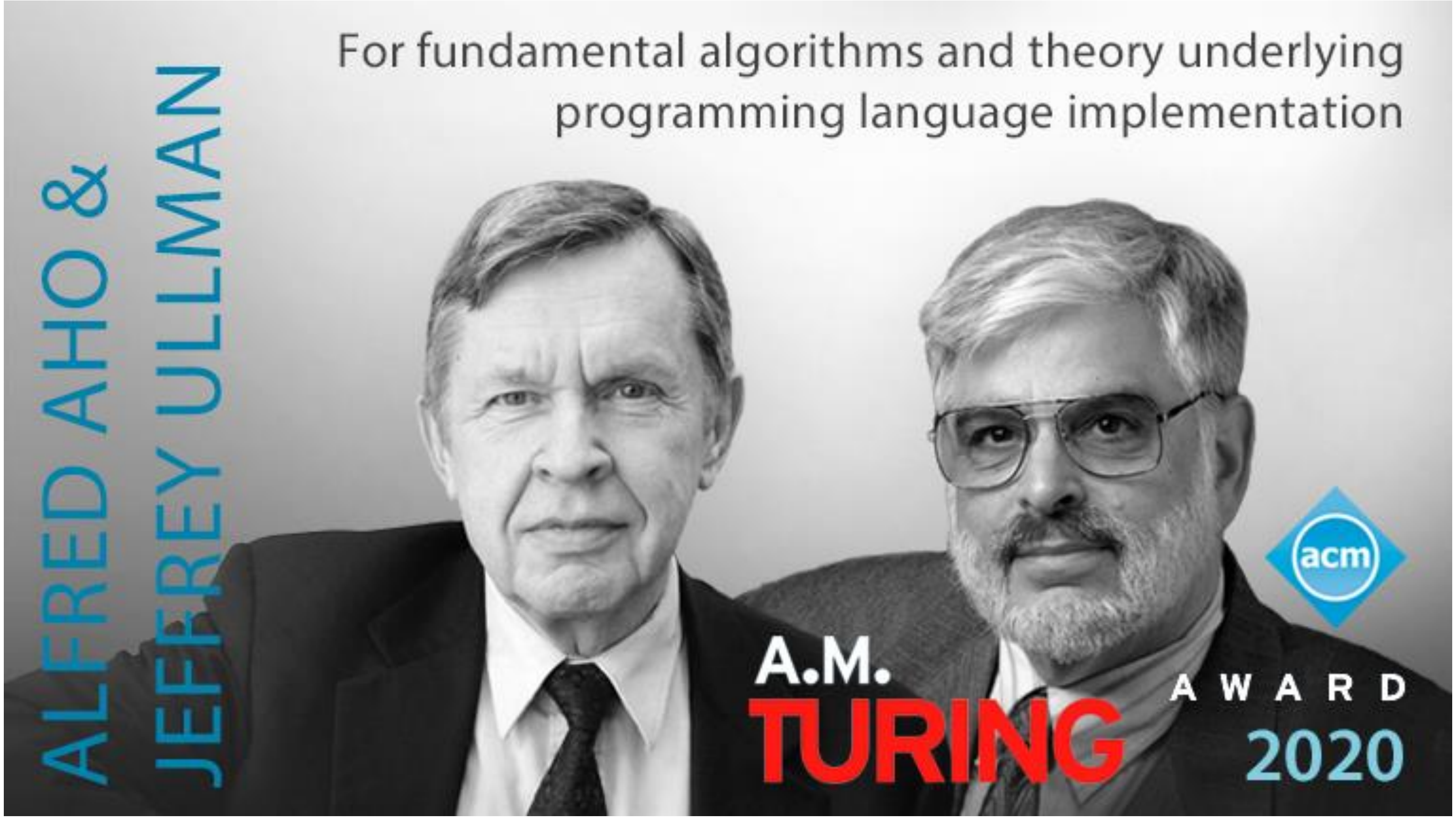
- We are all adults, no mandatory attendance.
- But you are responsible for all announcements (noclass, exam date, projects deadline, HWs and etc.).
- I can’t help you with your grade at the very end of the term.
- I’ll try to record.

A Note on Communication

- It is strongly recommended to Ask your question as soon as possible.
- Please use quera or Gmail.
- Don’t be ashamed. You can ask anonymously.
- No IM Please!

This Course Adapted form Stanford CS 143 And MIT CS 6.s081

(but with changes!)


For fundamental algorithms and theory underlying programming language implementation
ALFRED AHO &
JEFFREY ULLMAN
acm
A.M.
TURING
AWARD
2020

Course Theme

- Studying design and implementation a very complicated software.
- We break it apart!
- We are going to meet weird-useful language features.

Course Theme

- We use theory as much as possible to survive.

- Excellency at programming is needed.

Why Take Programming Languages and Compilers?

To appreciate the marriage of theory and practice

“Theory and practice are not mutually exclusive; they are intimately connected. They live together and support each other.” **[D.E. Knuth, 1989]**

Why Take Programming Languages and Compilers?

To appreciate the marriage of theory and practice

To explore the dimensions of computational thinking

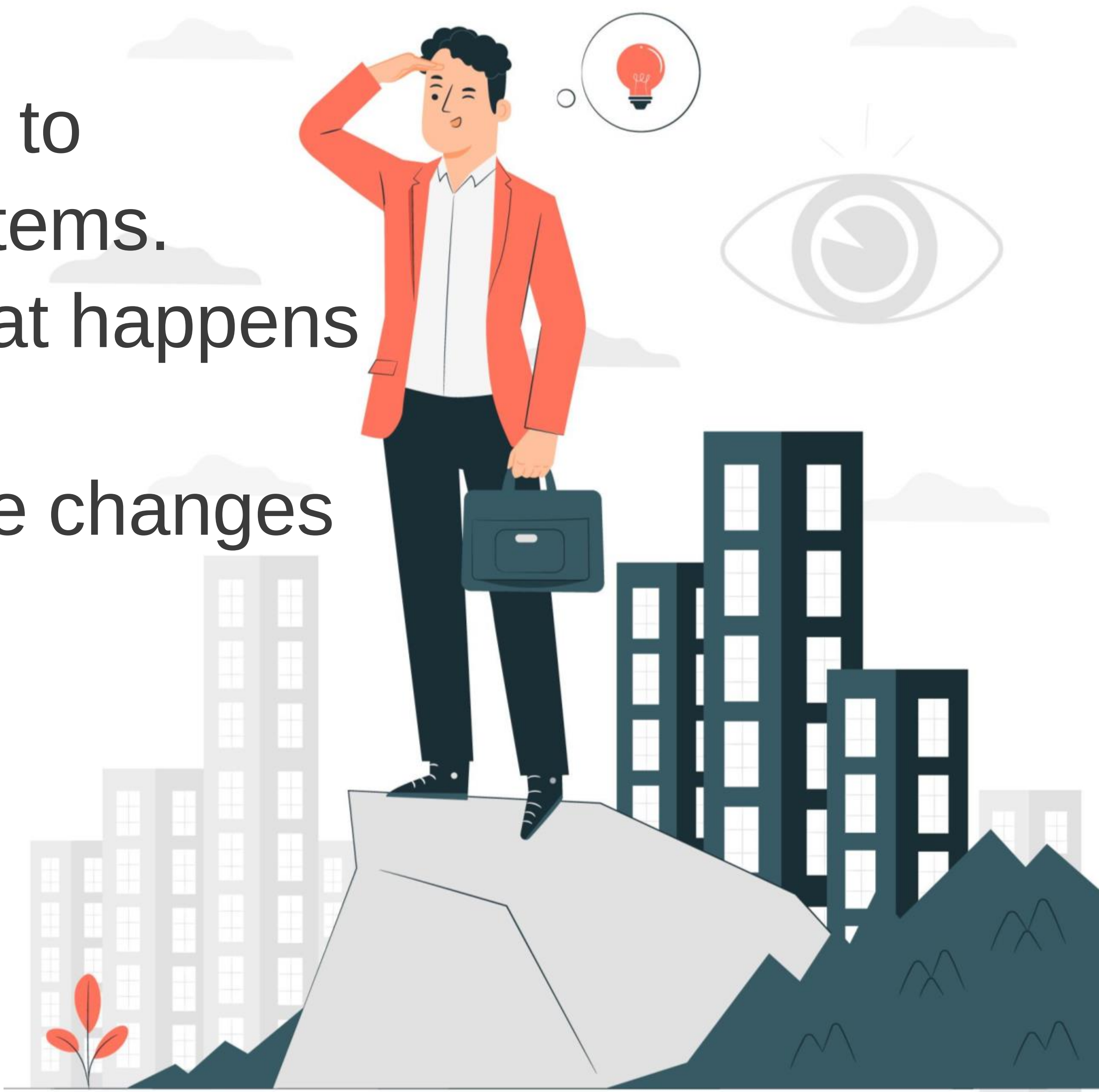
To exercise creativity

To learn robust software development practices

Why Study Compilers?

We Seek for a vision to

- build better systems.
- Understand what happens under the hood.
- Cope with future changes and era.



Why Study Compilers?

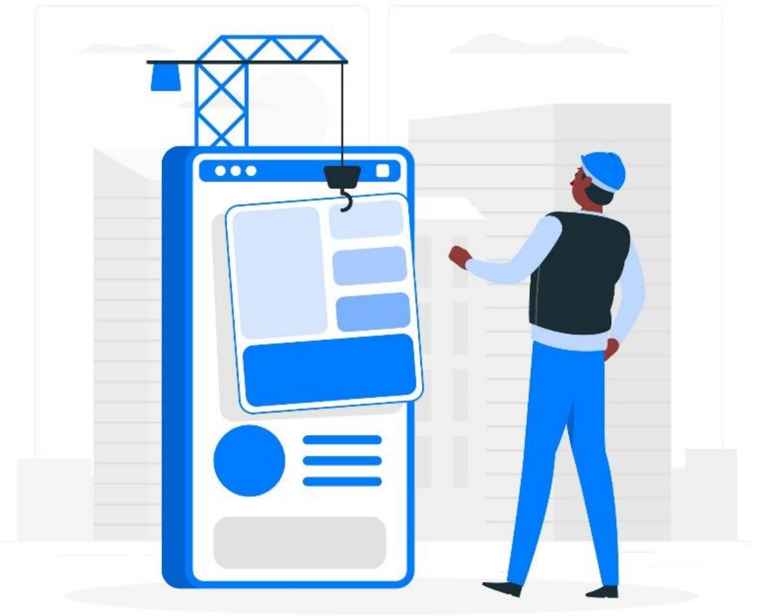
- Build a **large, ambitious software system**.

- Compiler Study Trains **Good Developers**.

- See theory **come to life**.

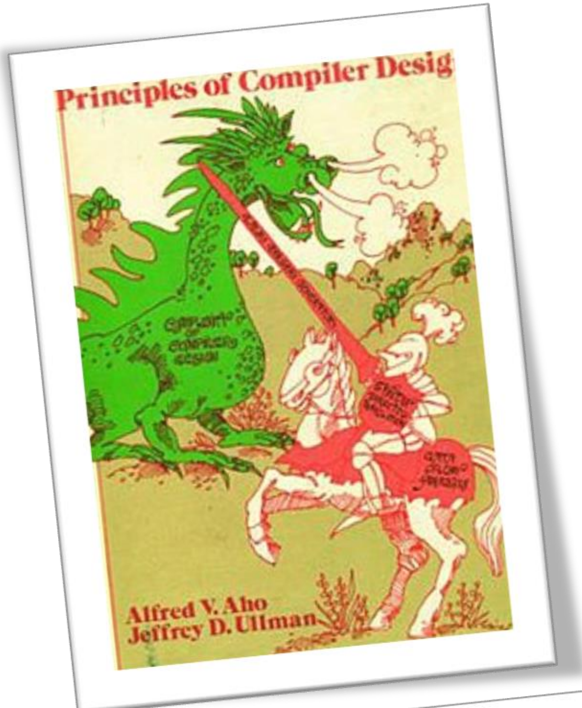
Why Study Compilers?

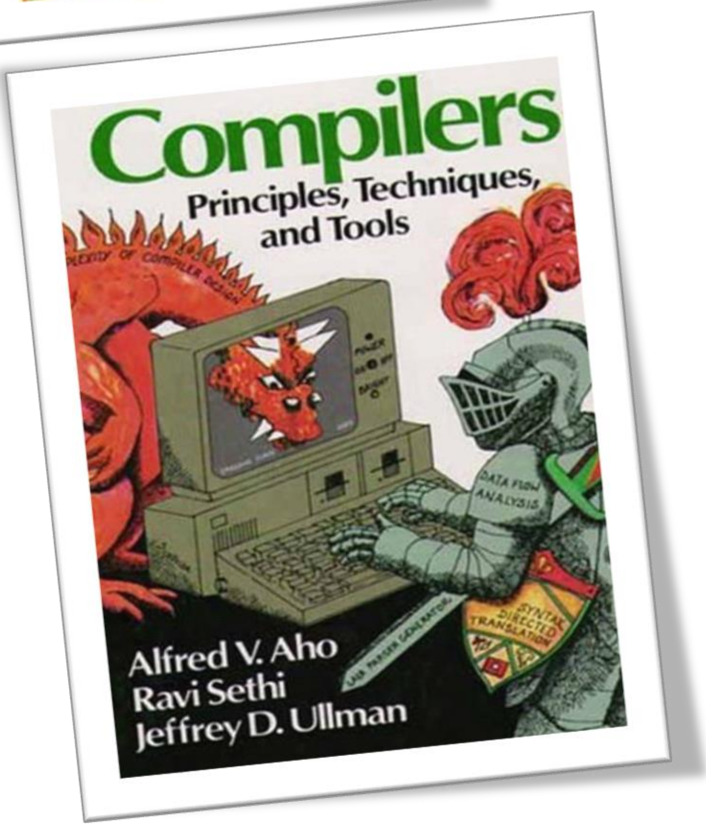
- Learn how to **build programming languages**.
- Learn **how programming languages work**.
- Learn **tradeoffs in language design**.

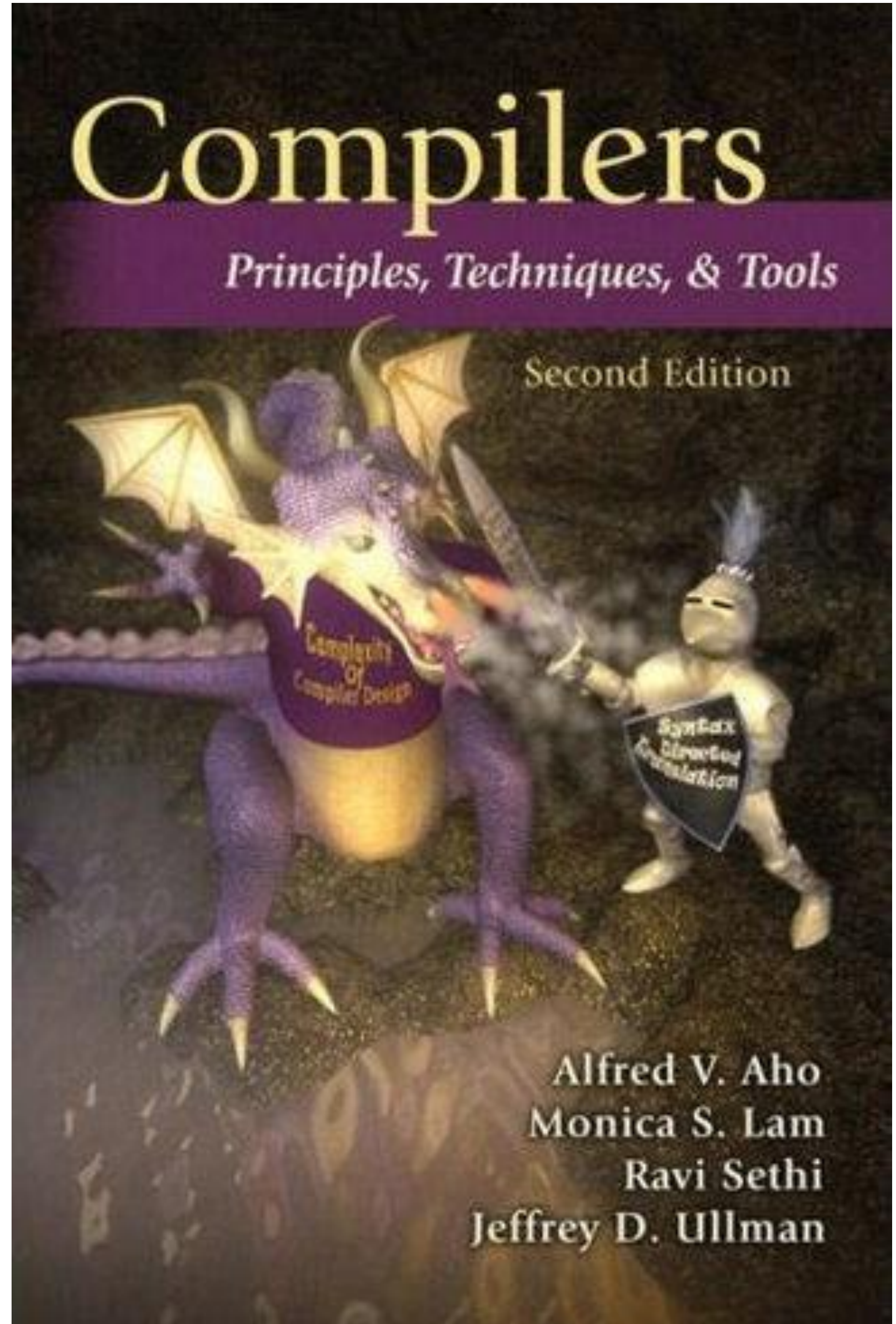


Why Study Compilers?

- Reasoning about programs makes better programmers.
- **Tool building**: there are programmers and there are tool builders…
- Transformable Skills; It is not all about programming: Javadoc comments to HTML, Server responds to net protocols and etc.


Principles of Compiler Desig
Alfred V. Aho
Jeffrey D. Ullman


Compilers
Principles, Techniques, and Tools
DATA FLOW ANALYSIS
Alfred V. Aho
Ravi Sethi
Jeffrey D. Ullman


Compilers
Principles, Techniques, & Tools
Second Edition
Complexity Of Compiler Design
Syntax Directed Translation
Alfred V. Aho
Monica S. Lam
Ravi Sethi
Jeffrey D. Ullman

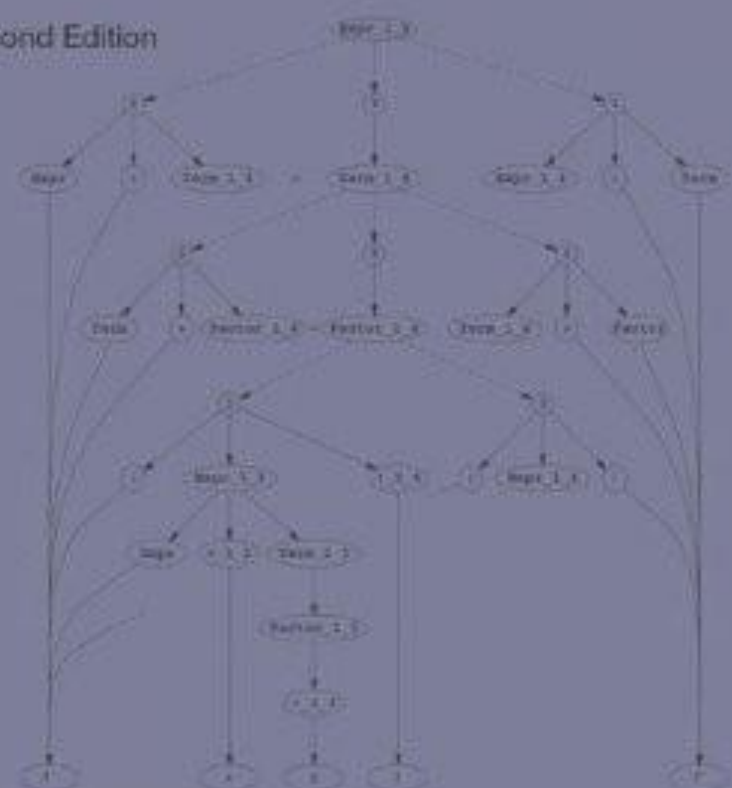
MONOGRAPHS IN COMPUTER SCIENCE

PARSING TECHNIQUES

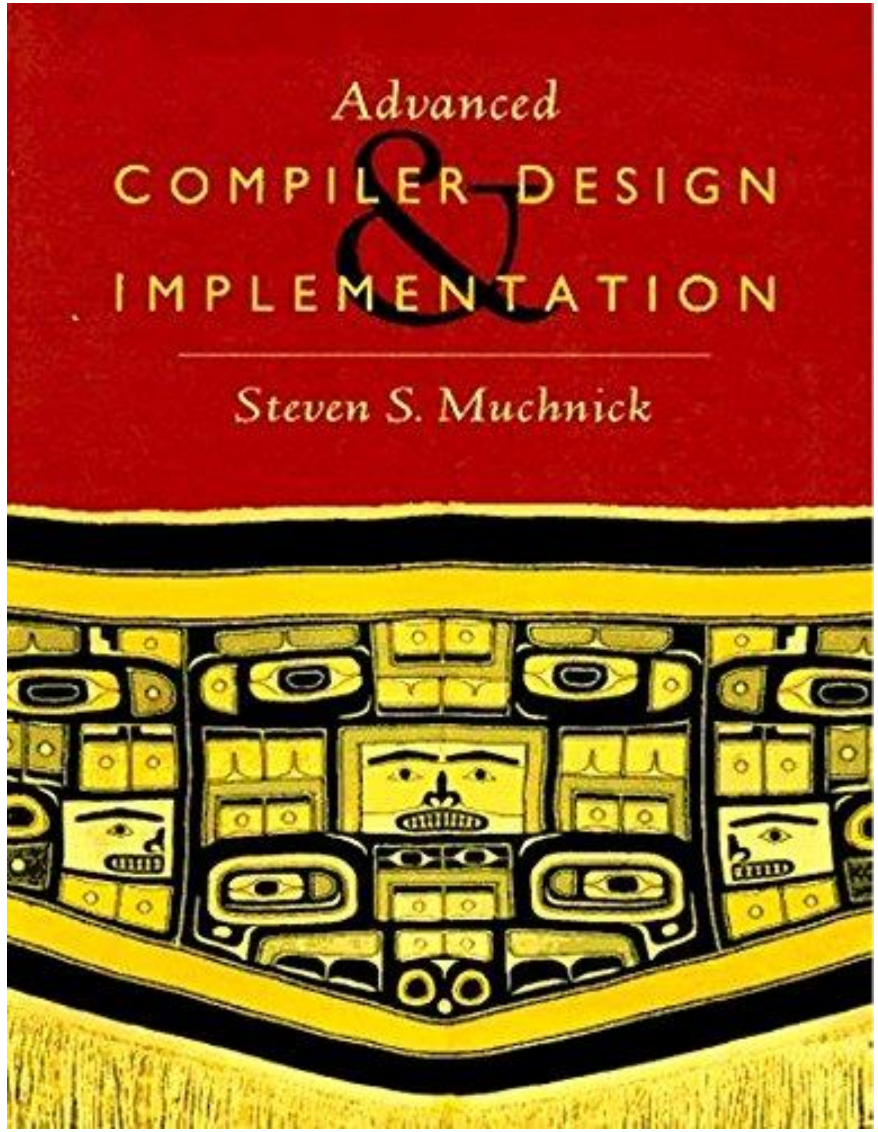
A Practical Guide

Dick Grune
Ceriel J.H. Jacobs

Second Edition



Springer


Advanced
COMPILER DESIGN
&
IMPLEMENTATION
Steven S. Muchnick

ENGINEERING A COMPILER

SECOND EDITION

MK
MORGAN KAUFMANN

Keith D. Cooper & Linda Torczon

Class Time:

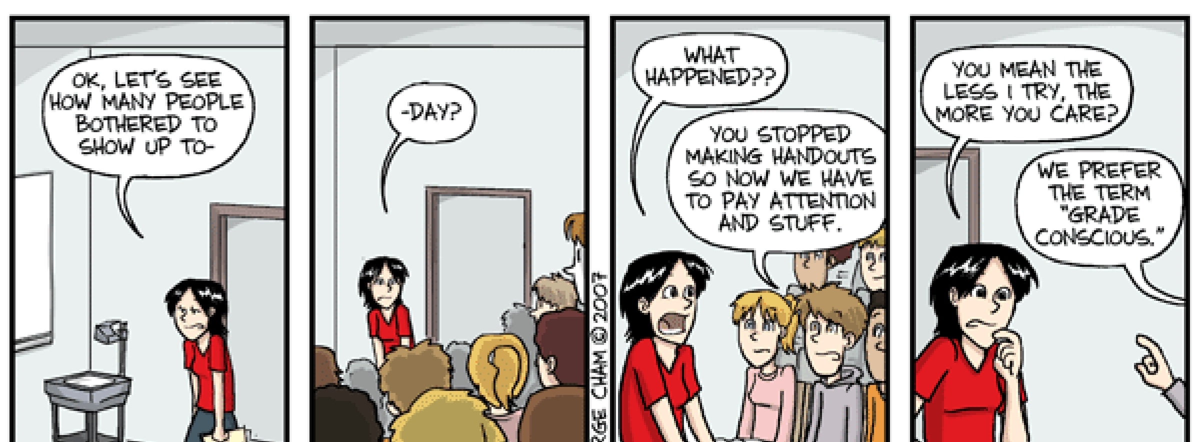
(Saturday)

And **Monday**

At: 16:30 - 18

Grading Policies ≈

- Midterm Exam: **5** (Last week of Aban or first week of Azar)
- Final Exam: **6** (24th Bahman 3:00 PM)
- Mini-Quiz: **1** (2 times, before & after midterm Exam with pre-announcement)
- Project: **4** (3 Phases)
- Homework: **4** (About 3-5 assignments)



Homework

- It should be written clearly, or no point is guaranteed.
- Soft late policy (10 days each 10%).

Homework

- If do it 4 days before deadline 10% extra point (10% of your Score)
- If do it 2 days before deadline 5% extra point (5% of your Score).

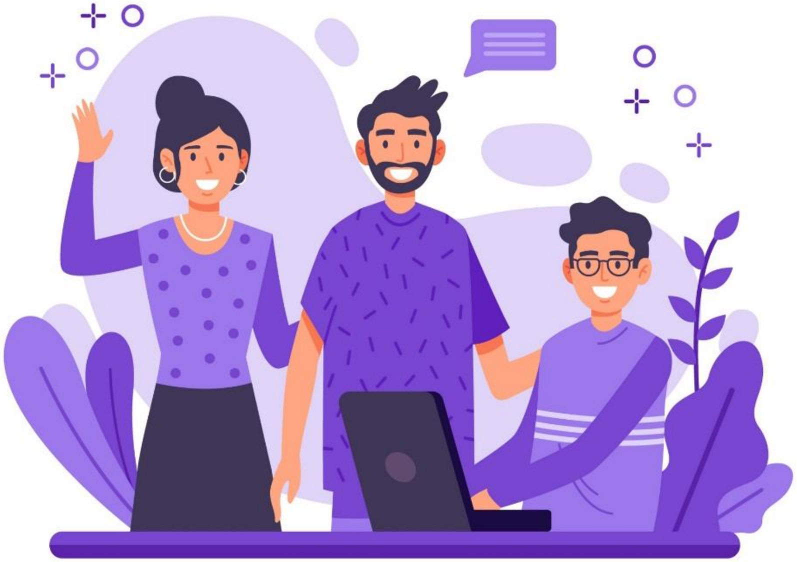
In the Exam We Have Personalized Questions From Homework

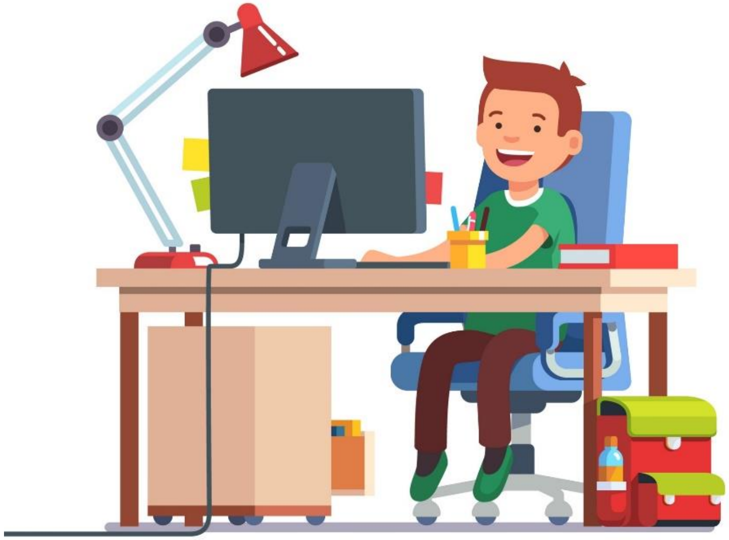
Exams

- We have two Comprehensive Exams.
- Reasonable exams, if you study study you can get a good mark.
- They are normal!
- You have samples.
- **No collaboration**!!!
- Possible random/nonrandom

oral exam.

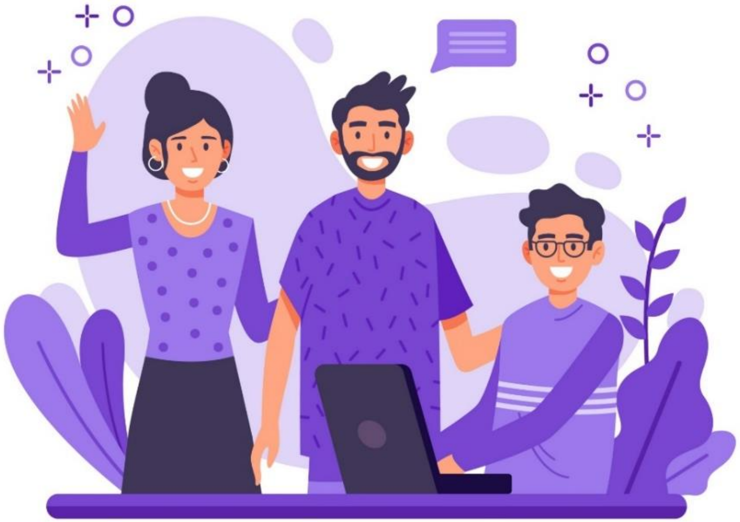
We work on team in project and homework.





Group Rules

- You work in the same group for the project and homework.
- Groups must consist of 2 or 3 people.
- Any change in the group’s members will change all former grades to 0.
- Communication among groups is not allowed.
- You work in a group but, you will have your own grade.



Projects

- It is a complete compiler in 3-4 phases.
  - Lexical Analyzer (Scanner)
  - Parser
  - Code Generation + Optimization

A Word on the Honor Code...

Prerequisites

Data Structure

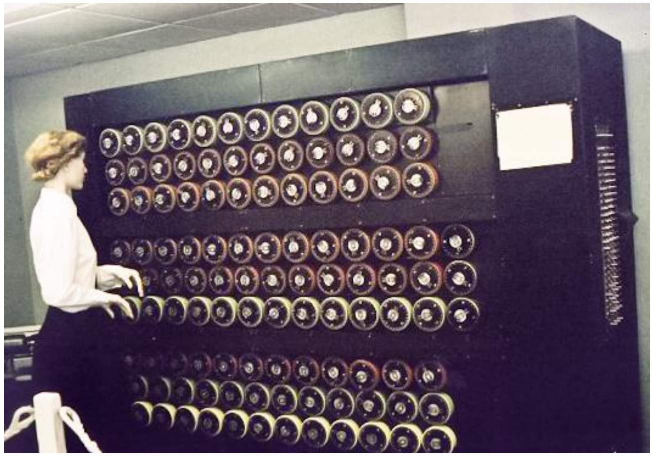
Automata

Maturity in programming and patience is also required.


HO
HO

What is a Compiler?

History of High-level Languages



A Short History of Compilers

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

First Practical Compiler

In his PhD dissertation 1951; published in 1954), Corrado Böhm describes for the first time a translation mechanism of a programming language, written in that same language.

https://en.wikipedia.org/wiki/Corrado_B%C3%B6hm
https://en.wikipedia.org/wiki/B%C3%B6hm%27s_language

High-Level Languages

Image: http://upload.wikimedia.org/wikipedia/commons/thumb/5/55/Grace_Hopper.jpg/300px-Grace_Hopper.jpg
http://www.nytimes.com/2007/03/20/business/20backus.html

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

High-Level Languages

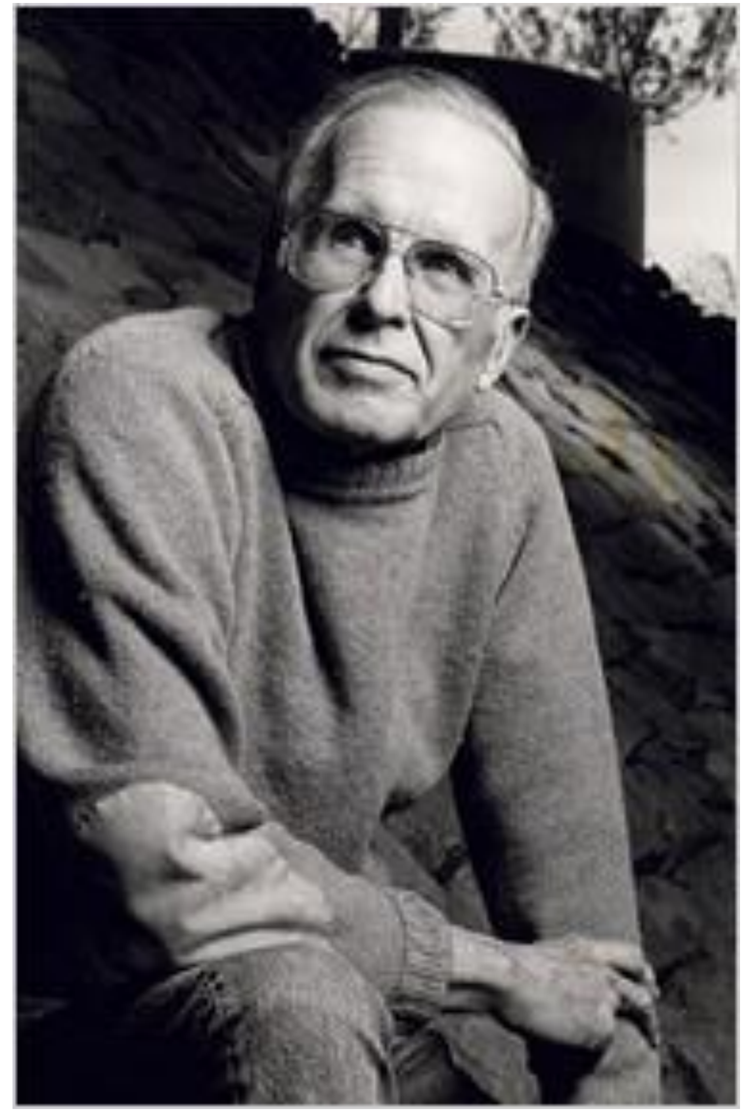
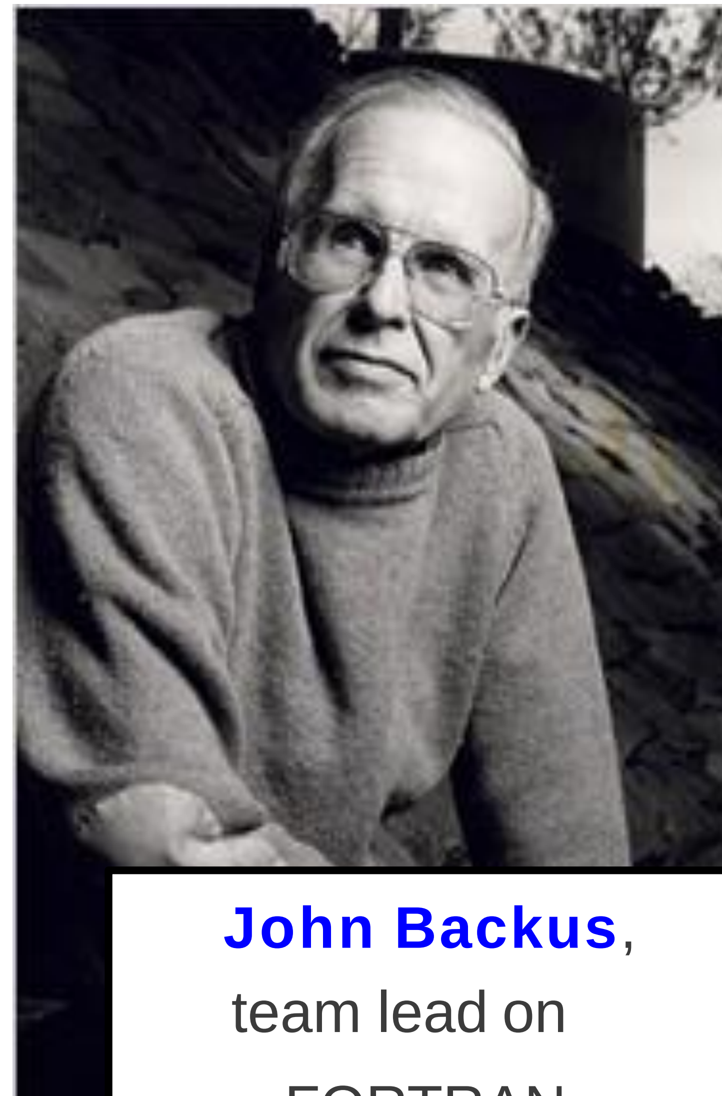


Image: http://upload.wikimedia.org/wikipedia/commons/thumb/5/55/Grace_Hopper.jpg/300px-Grace_Hopper.jpg
http://www.nytimes.com/2007/03/20/business/20backus.html

High-Level Languages



John Backus,
team lead on
FORTRAN.

Image: http://upload.wikimedia.org/wikipedia/commons/thumb/5/55/ ... er.jpg
http://www.nytimes.com/2007/03/20/business/20backus.html

FORTRAN I

- Translate high-level code to assembly.
- Many thought this impossible.
- Had already failed in other projects.
- Development time halved
- Performance is close to hand-written assembly!

Effect on Computer Science

- The first compiler
- Huge impact on computer science.
- Led to an enormous body of theoretical and practical work.
- Modern compilers preserve the outlines of FORTRAN I

```
INTEGER FUNCTION FCN20(NDIMS, X, NFCNS, FUNVLS)
    INTEGER NDIMS, NFCNS
    DOUBLE PRECISION X(*), FUNVLS(*)
    DOUBLE PRECISION Z
    Z = (X(1) + X(2) + X(3)) ** 2
    IF (Z .NE. 0.0) THEN
       FUNVLS(1) = 1.0 / Z
    ELSE
       FUNVLS(1) = 0.0
    ENDIF
    FCN20 = 1
    RETURN
END
```

What is a Compiler?

- Takes as input a program written in one language and **translates** it into a **functionally equivalent** program in another language.
- Source is usually high-level (e.g. Java), target is usually low-level (e.g. Assembly).

Computational Thinking in Programming Language Design

Underlying every programming language is a model of computation:

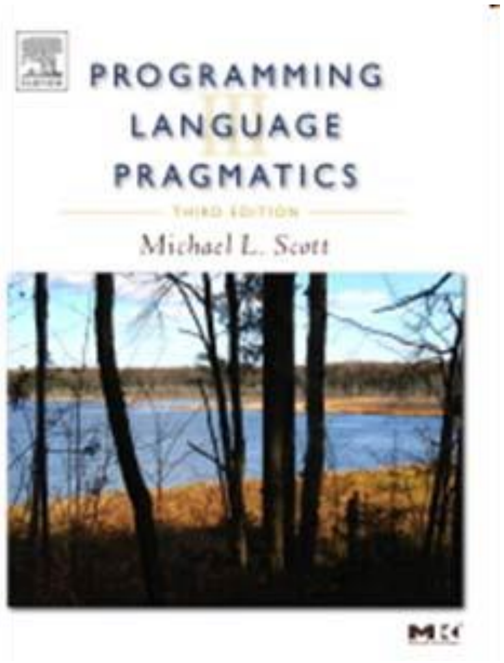
Procedural: C, C++, C#, Java

Declarative: SQL

Logic: Prolog

Functional: Haskell

Scripting: AWK, Perl, Python, Ruby



Alfred Aho, **SIGCSE 2010**

Evolutionary Forces on Languages and Compilers

More and different kinds of languages

Increasing diversity of applications

Stress on increasing productivity

Need to improve software reliability

Target machines more diverse

Parallel machine architectures

Massive compiler collections

1970	2010
Fortran	Java
Lisp	C
Cobol	PHP
Algol 60	C++
APL	Visual Basic
Snobol 4	C#
Simula 67	Python
Basic	Perl
PL/1	Delphi
Pascal	JavaScript

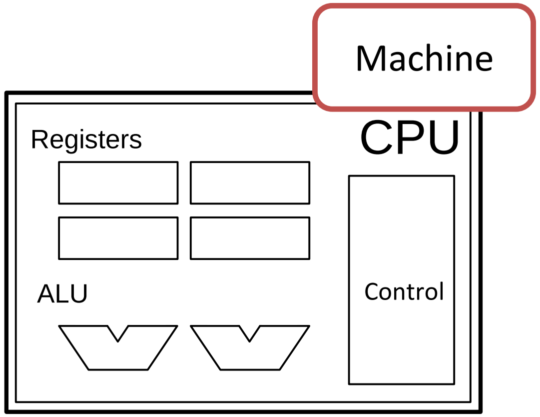
[http://www.tiobe.com]

Schema

Language

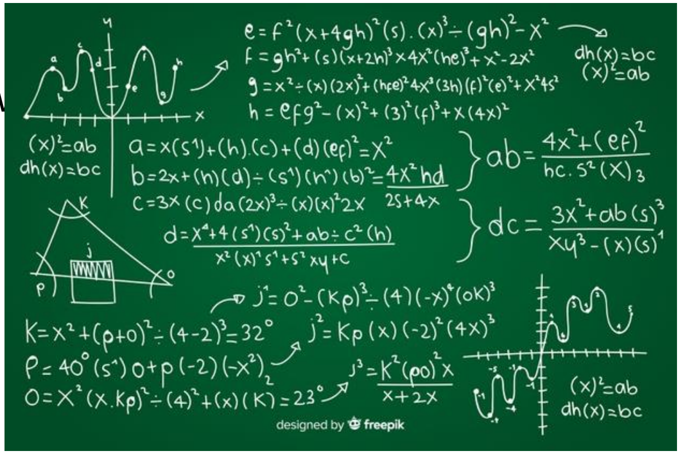
```
fun(y) {
  x = y - 2;
  return x;
};
```

Infinite resources
No performance specification



- Finite resources
- Extremely performance sensitive

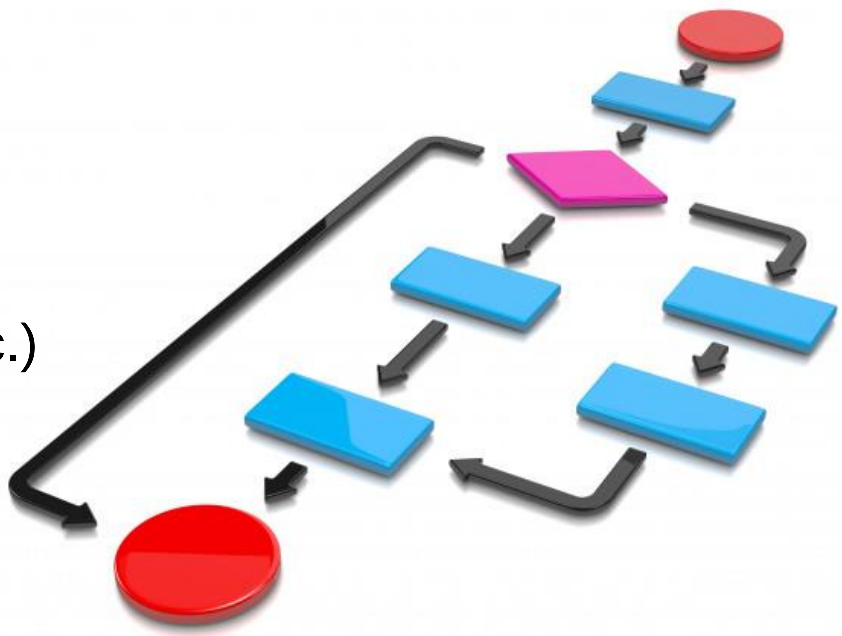
Language Environment Construction touches many topics in Computer Science



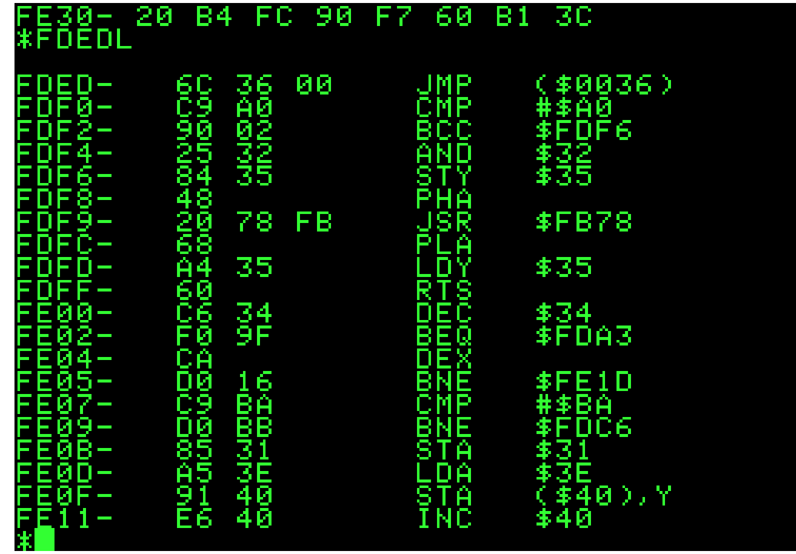
- Theory
  - Finite State Automata, Grammars and Parsing, data-flow
- Algorithms
  - Graph manipulation, dynamic programming
- Data structures
  - Symbol tables, abstract syntax trees
- Systems
  - Allocation and naming, multi-pass systems, compiler construction
- Computer Architecture
  - Memory hierarchy, instruction selection, interlocks and latencies, parallelism
- Security
  - Detection of and Protection against vulnerabilities
- Software Engineering
  - Software development environments, debugging
- Artificial Intelligence
  - Heuristic based search for best optimizations

Inputs

- Standard language
  - State
    - Variables,
    - Structures,
    - Arrays
  - Computation
    - Expressions (arithmetic, logical, etc.)
    - Assignment statements
    - Control flow (conditionals, loops)
    - Procedures



Outputs



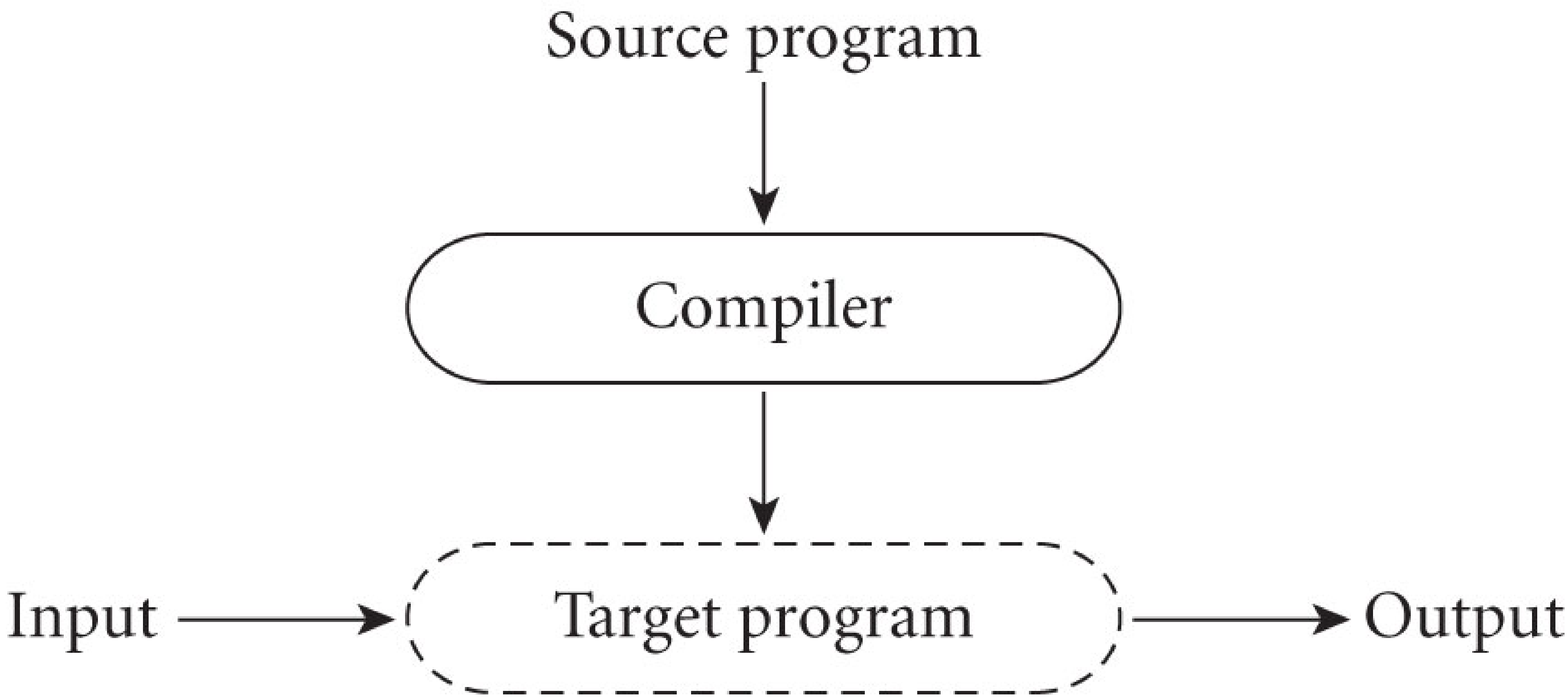
- State
  - Registers
  - Memory with Flat Address Space
- Machine code – load/store architecture
  - Load, store instructions
  - Arithmetic, logical operations on registers
  - Branch instructions

Translation Approaches

- Compiler Approach
- Interpreter Approach
- Dynamic Approach

Compiler Approach

- The target is not necessarily a machine code.
  - e.g. Assembly language e.g. MIPS or x86
  - e.g. VHDL: the output is C.
  - It might be intermediate code e.g. JBC
- By following **physical structure** of program we translate it.
- The generated code is much more faster.
- We decide before run the code (e.g. type)


Source program
Compiler
Input
Target program
Output

Target Languages

Another programming language

CISCs

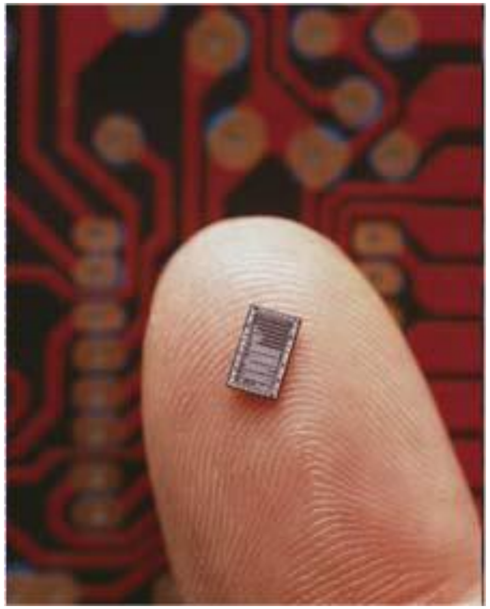
RISCs

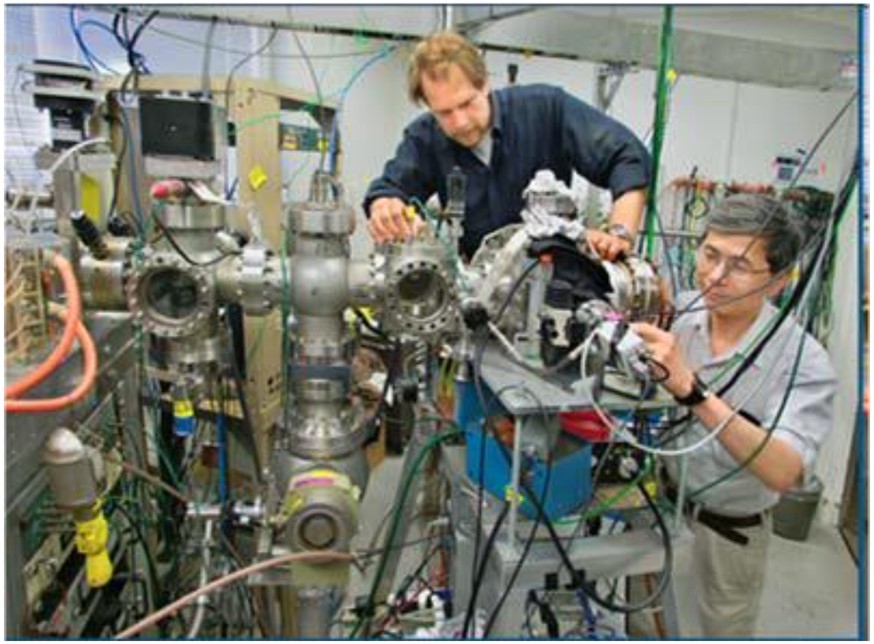
Vector machines

Multicores

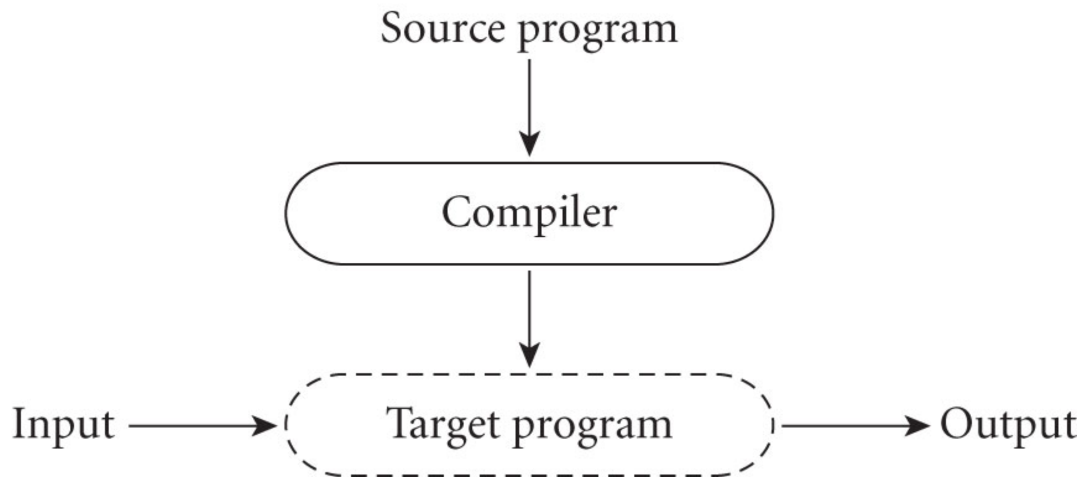
GPUs

Quantum computers

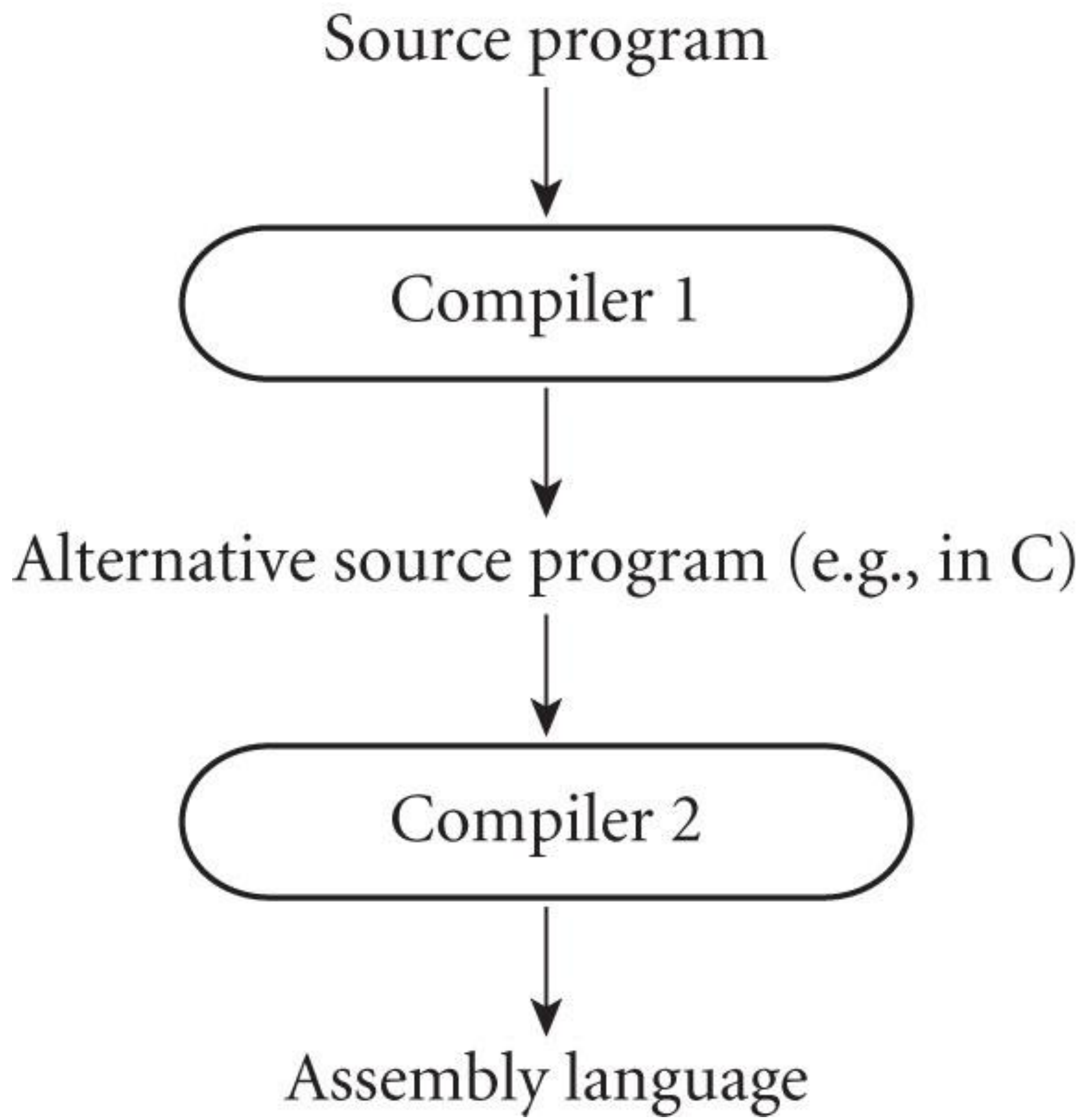




```c
#include <stdio.h>

int getint() {
    int i;
    scanf("%d", &i);
    return i;
}

int gcd() {
    int i = getint(), j = getint();

    while (i != j) {
        if (i > j) i = i - j;
        else j = j - i;
    }
     return I;
}
```

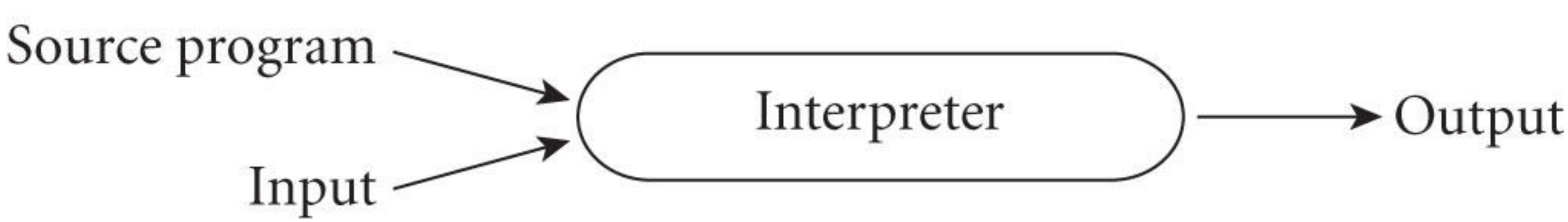



```asm
.LC0:
        .string "%d"
getint():
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        lea     rax, [rbp-4]
        mov     rsi, rax
        mov     edi, OFFSET FLAT:.LC0
        mov     eax, 0
        call    __isoc99_scanf
        mov     eax, DWORD PTR [rbp-4]
        leave
        ret
.LC1:
        .string "%d\n"
putint(int):
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR [rbp-4], edi
        mov     eax, DWORD PTR [rbp-4]
        mov     esi, eax
        mov     edi, OFFSET FLAT:.LC1
        mov     eax, 0
        call    printf
        leave
        ret
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        call    getint()
        mov     DWORD PTR [rbp-4], eax
        call    getint()
        mov     DWORD PTR [rbp-8], eax
        jmp     .L5
.L7:
        mov     eax, DWORD PTR [rbp-4]
        cmp     eax, DWORD PTR [rbp-8]
        jle     .L6
        mov     eax, DWORD PTR [rbp-8]
        sub     DWORD PTR [rbp-4], eax
        jmp     .L5
.L6:
        mov     eax, DWORD PTR [rbp-4]
        sub     DWORD PTR [rbp-8], eax
.L5:
        mov     eax, DWORD PTR [rbp-4]
        cmp     eax, DWORD PTR [rbp-8]
        jne     .L7
        mov     eax, DWORD PTR [rbp-4]
        mov     edi, eax
        call    putint(int)
        mov     eax, 0
        leave
        ret
```


Source program
Compiler 1
Alternative source program (e.g., in C)
Compiler 2
Assembly language

Interpreter Approach

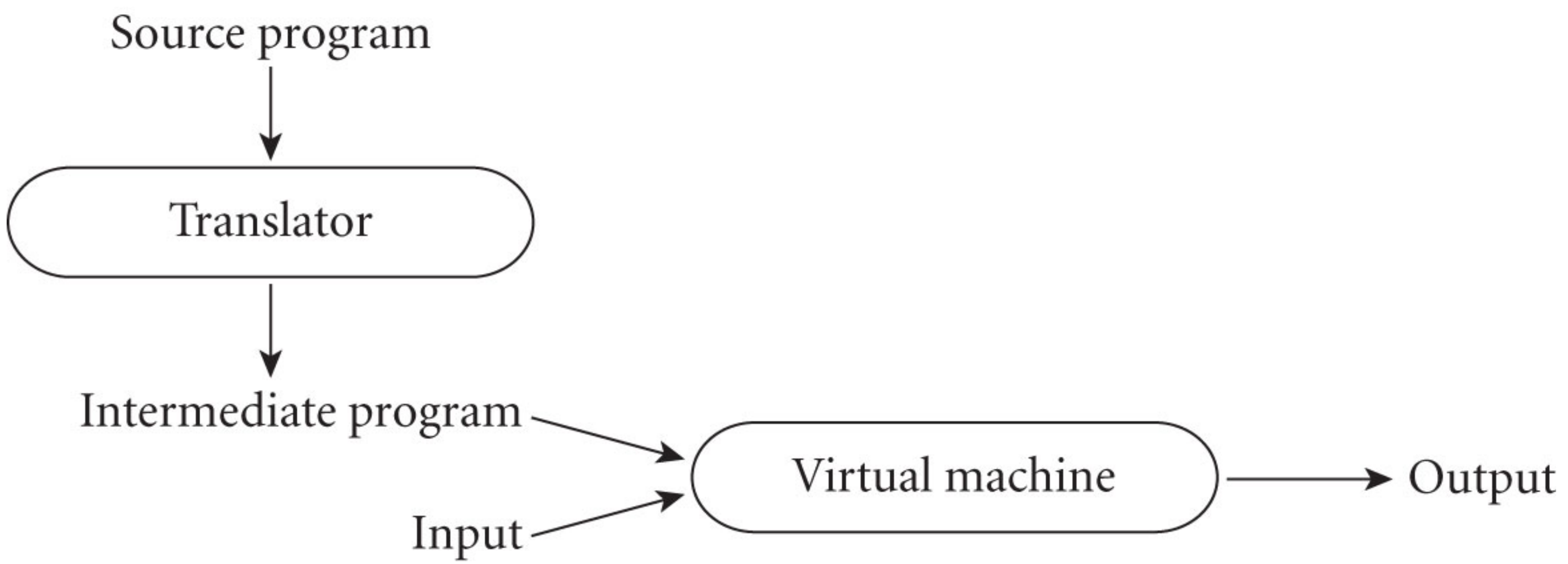
- Interpreter **translate** the source code to machine code **online**.
- The code is **translated** while it is **running.**



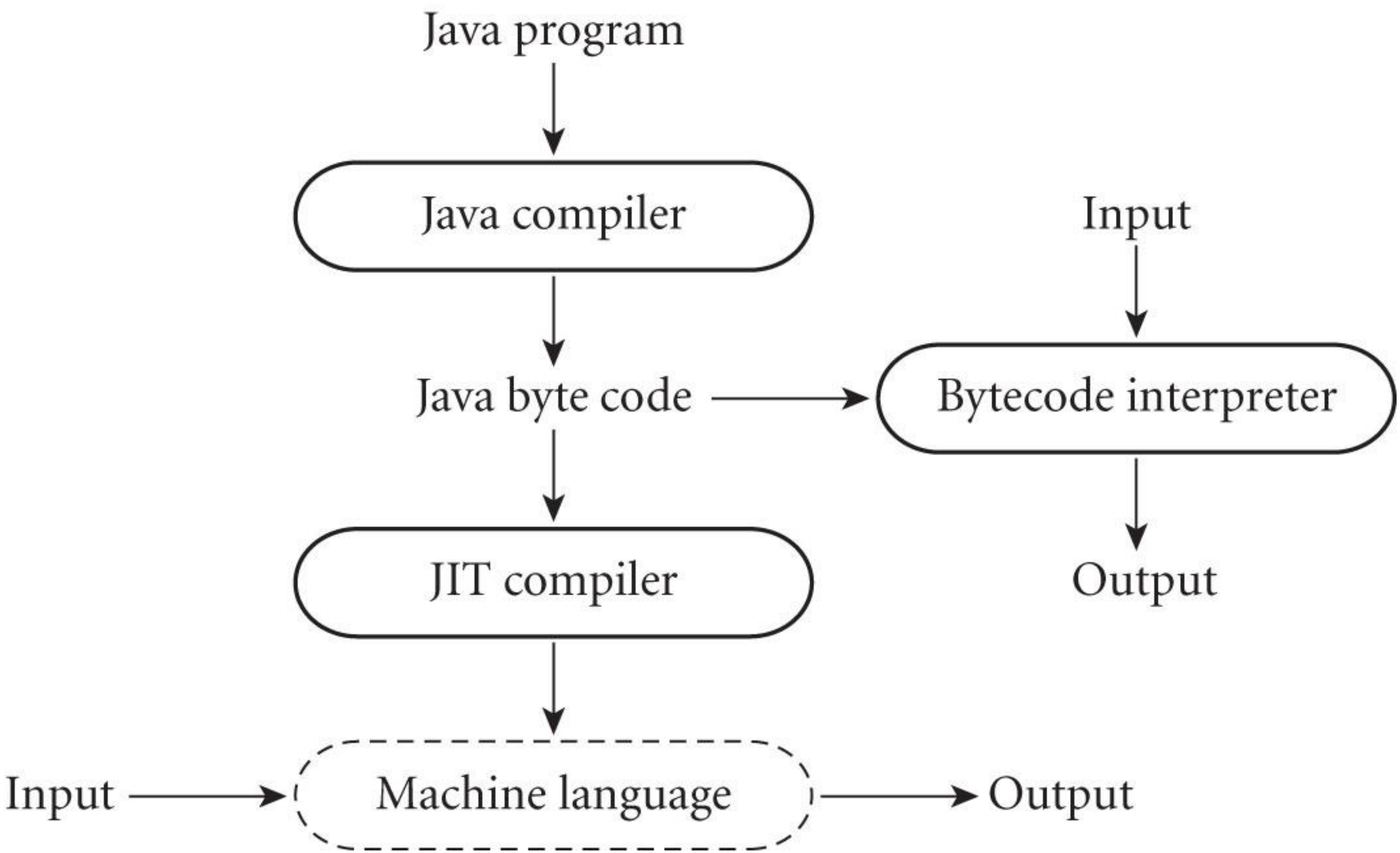
Interpreter Approach

- We follow the **execution path** to translate the code.
- leads to greater **flexibility** and better **diagnostics** (error messages) than does compilation.
- It can also cope with languages in which fundamental characteristics of the program, such as the sizes and types of variables, or even which names refer to which variables, can depend on the input data.

A Hybrid Approach



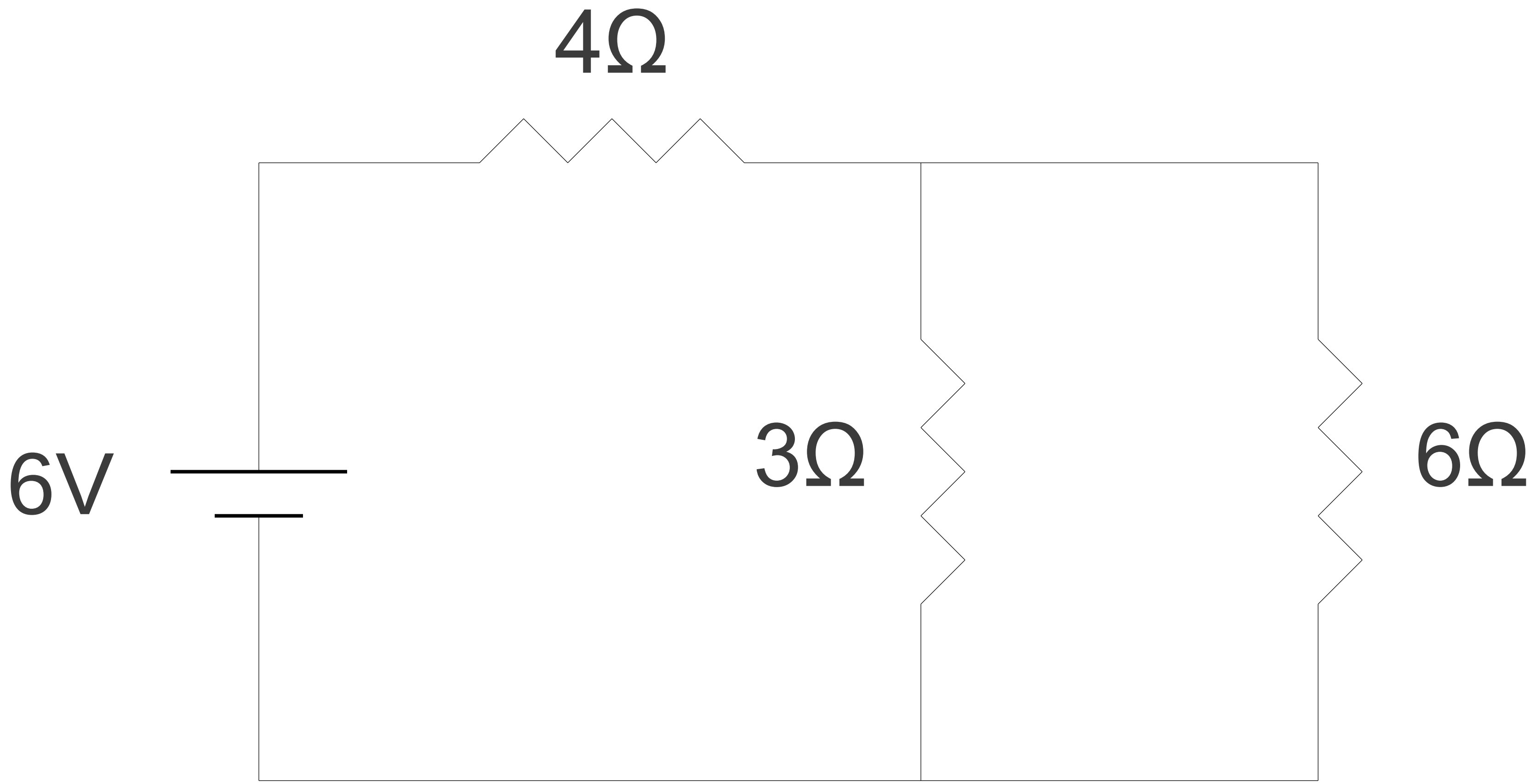
Hybrid Approach

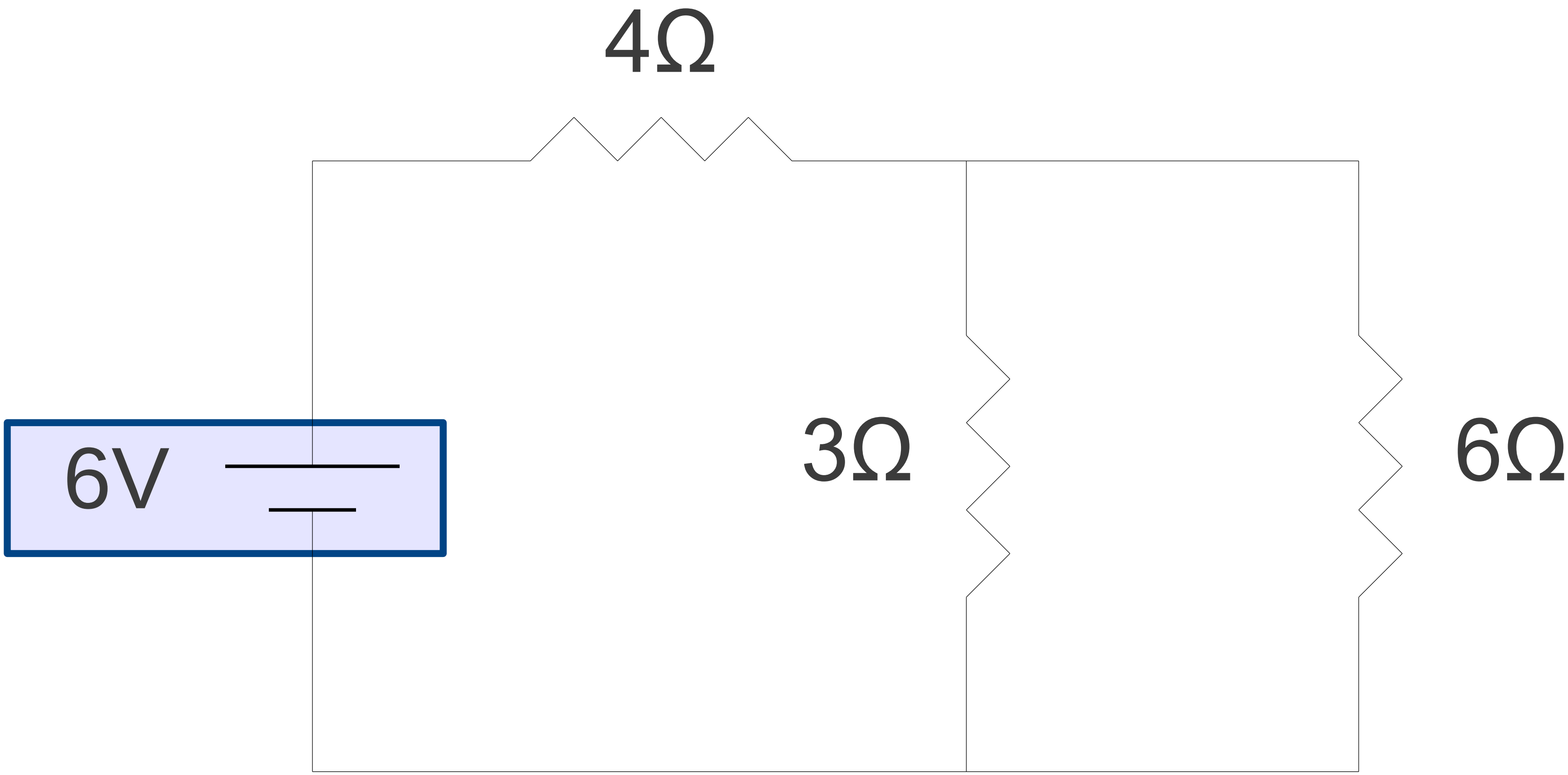


Compilers Can Have Many Other Forms

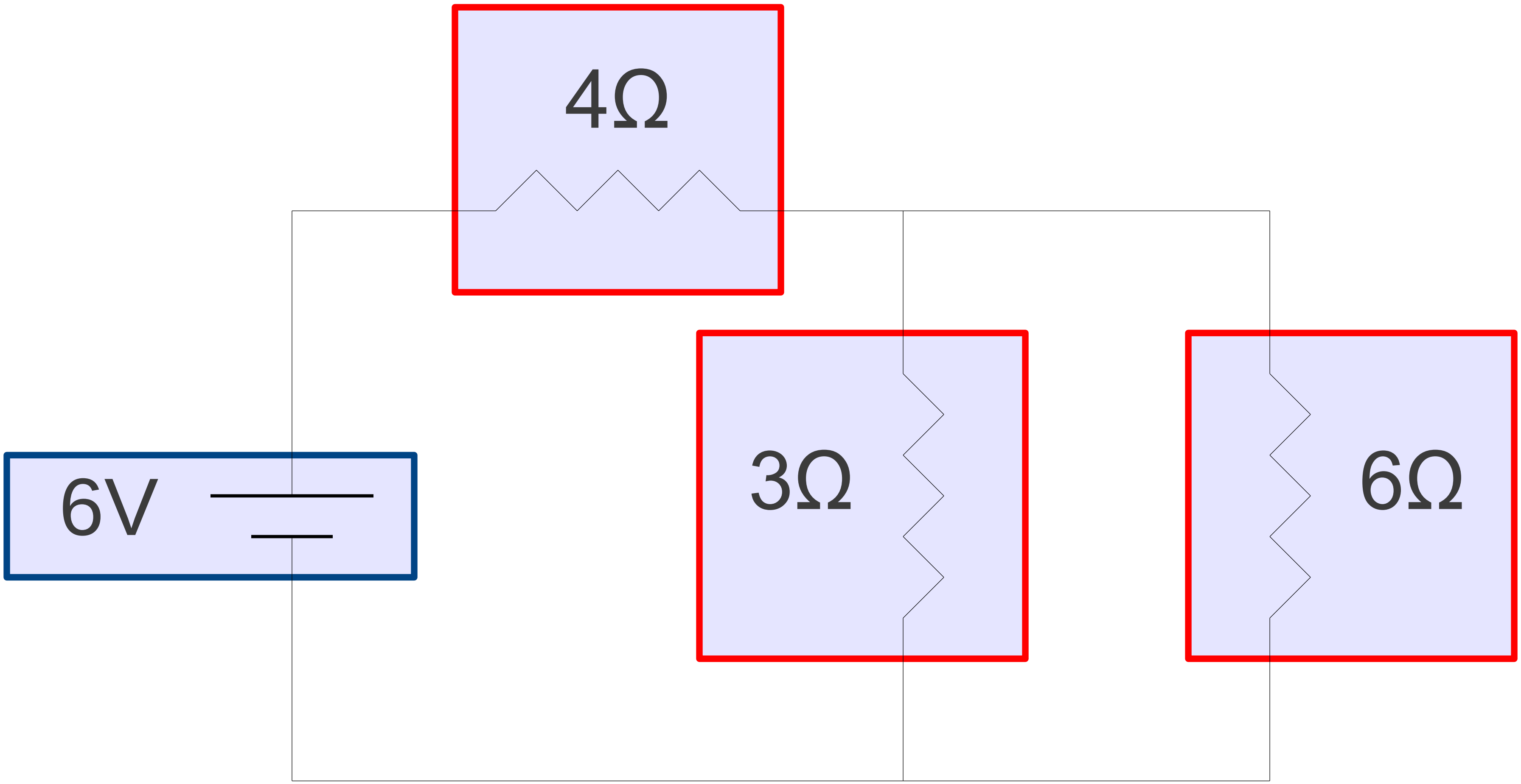
- **Cross compiler:** a compiler on one machine that generates target code for another machine
- **Incremental compiler:** one that can compile a source program in increments
- **Just-in-time compiler:** one that is invoked at runtime to compile each called method in the IR to the native code of the target machine
- **Ahead-of-time compiler:** one that translates IR to native code prior to program execution

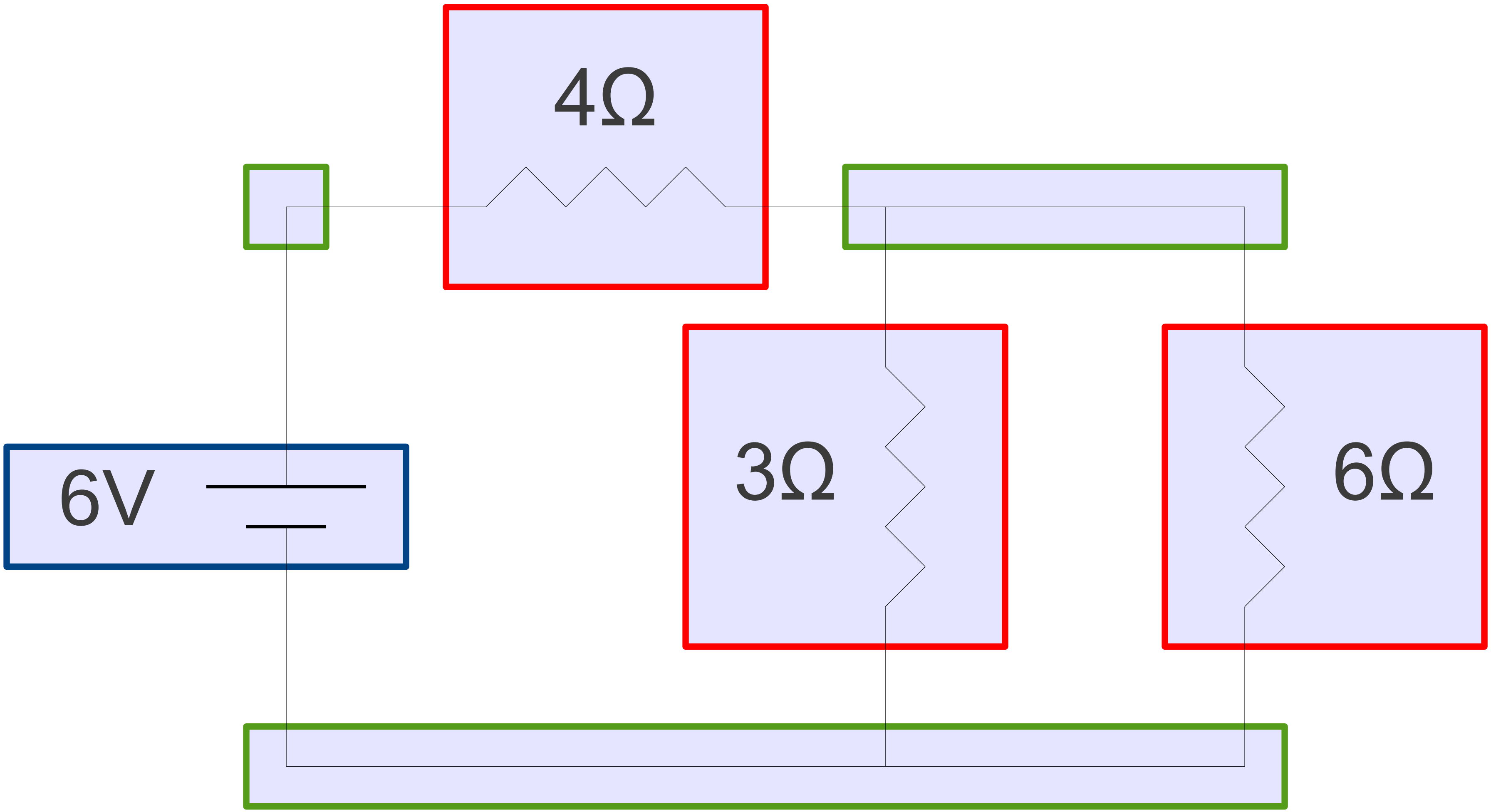
How does a compiler work?

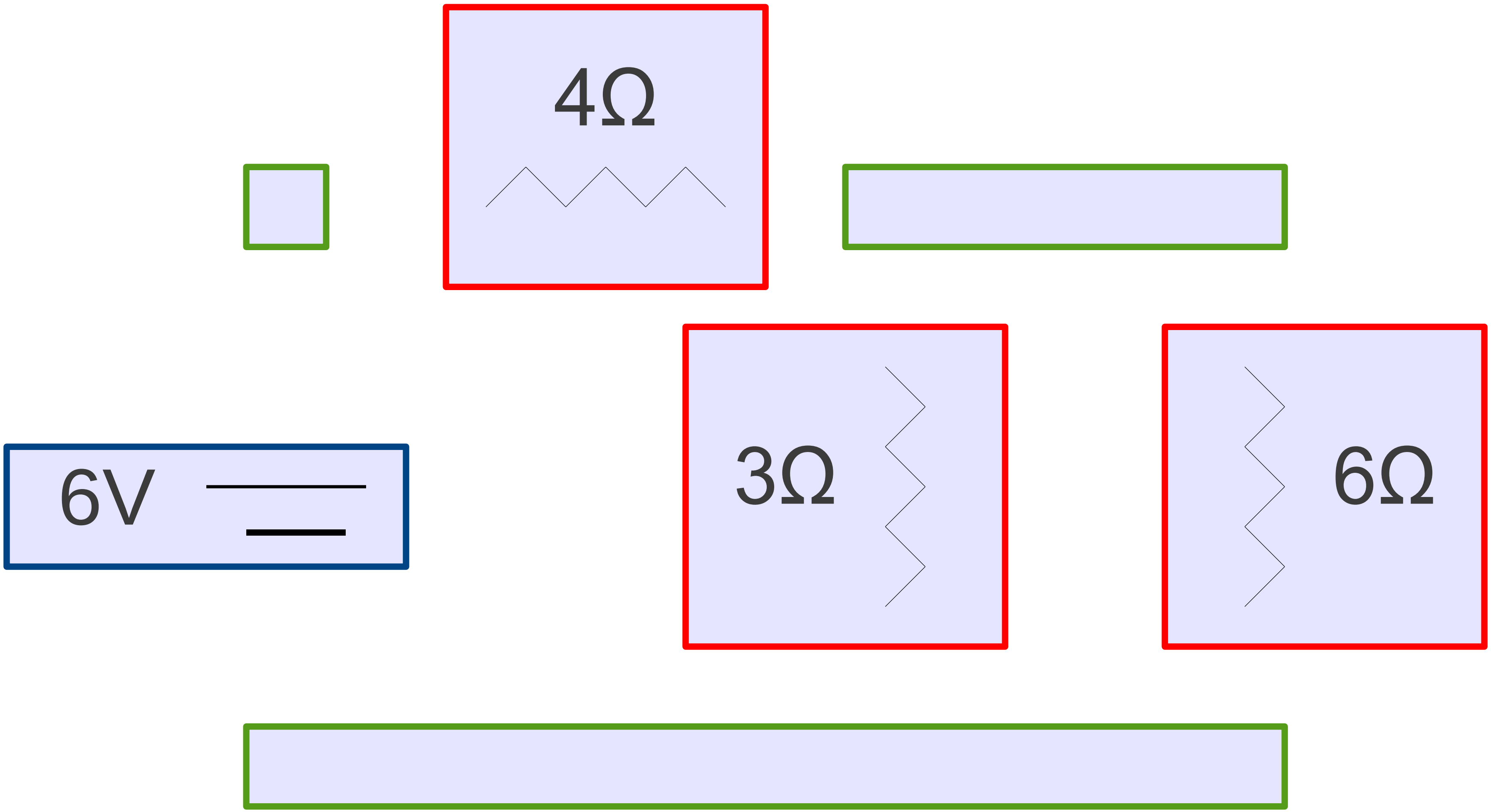

4Ω
6V
3Ω
6Ω

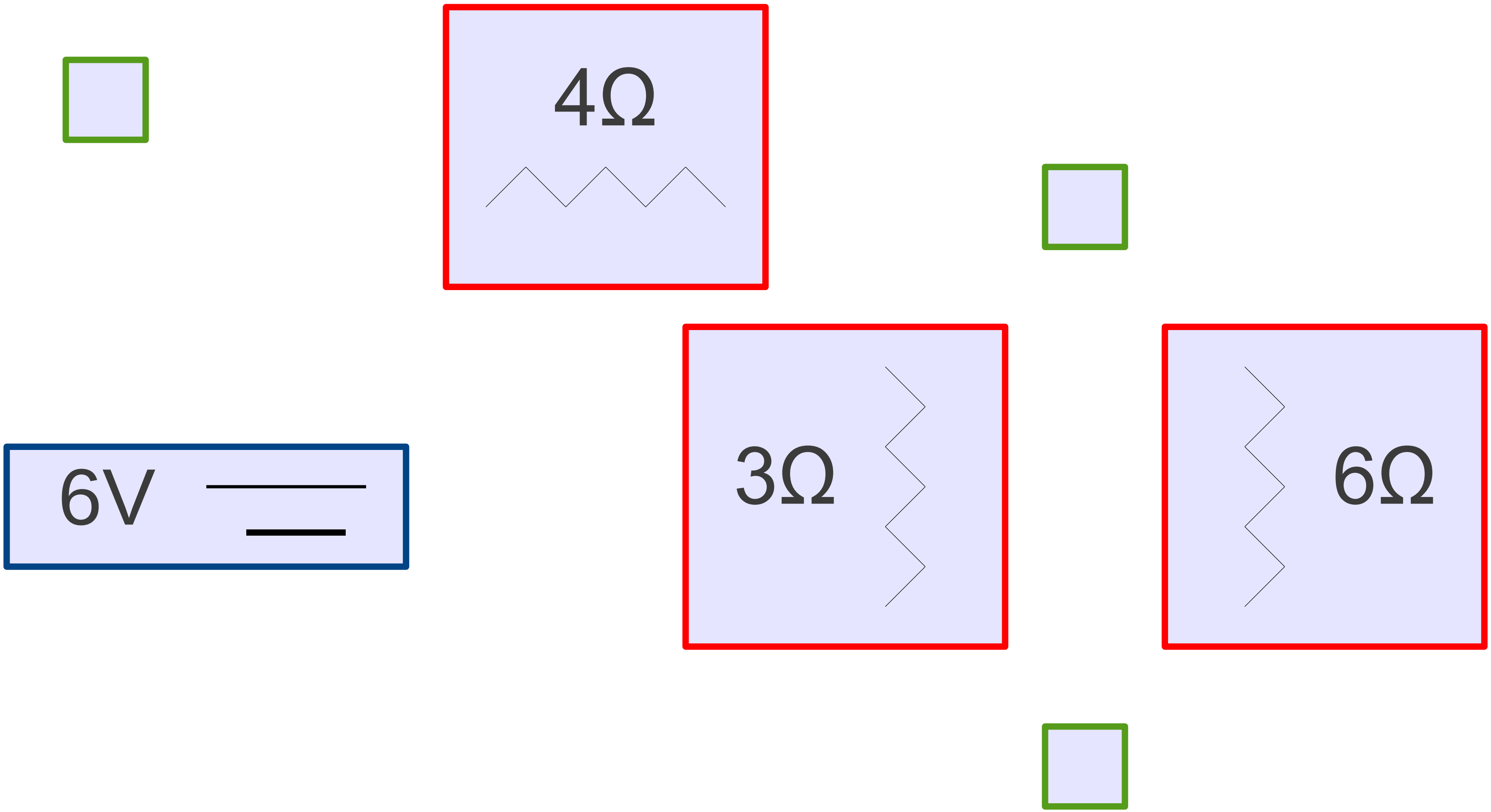


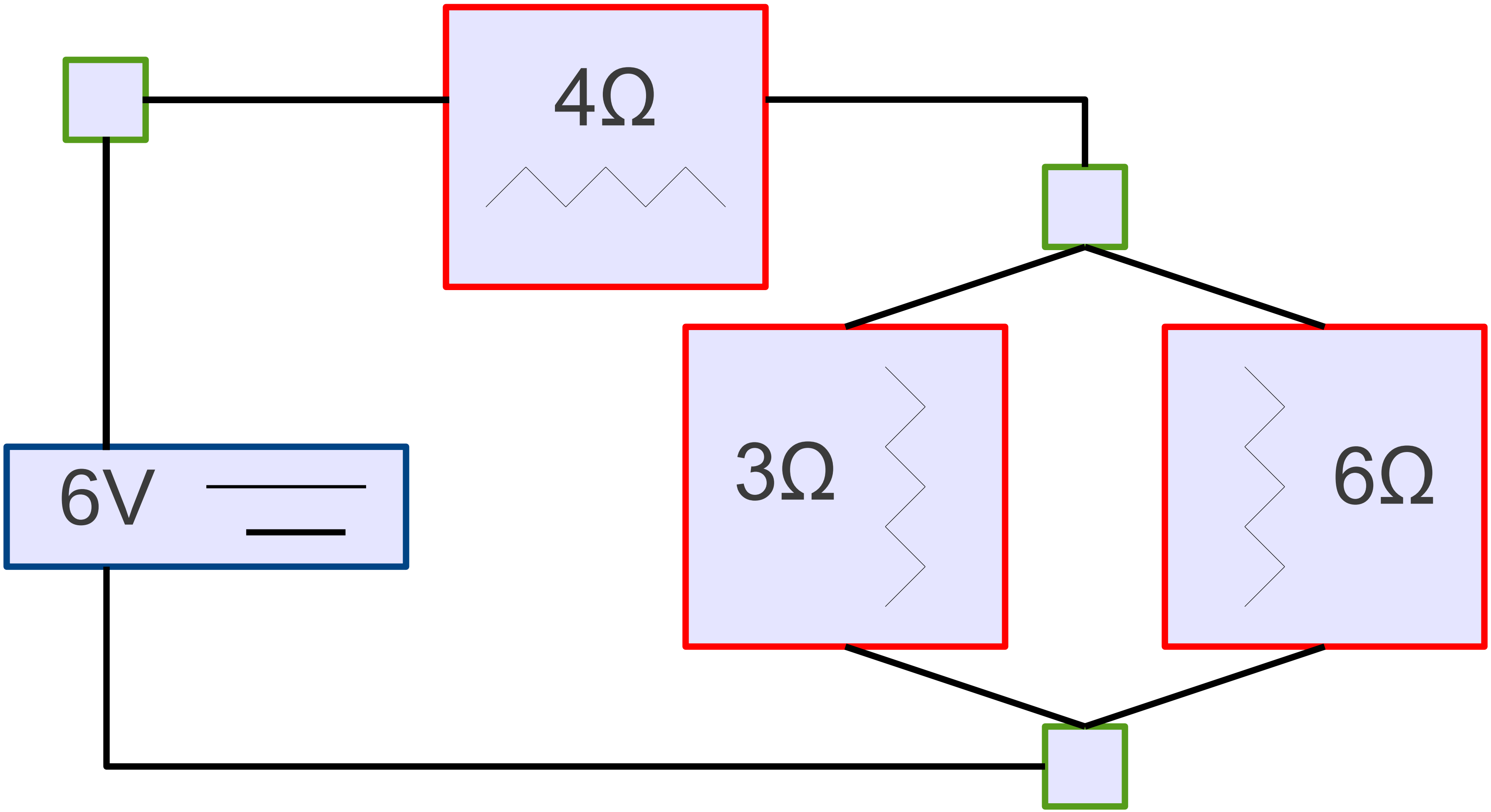
4Ω
6V
3Ω
6Ω

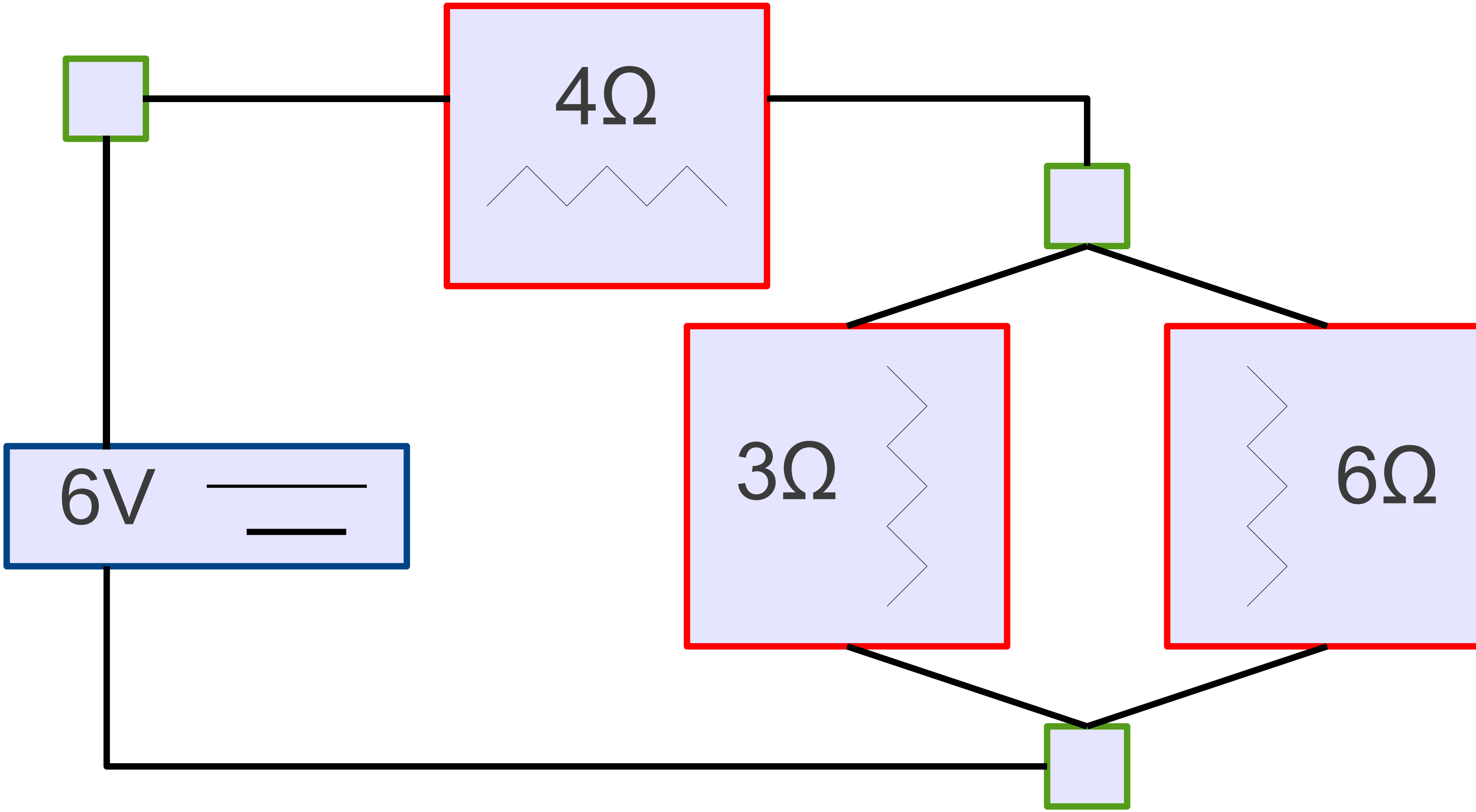

4Ω
6V
3Ω
6Ω


4Ω
6V
3Ω
6Ω

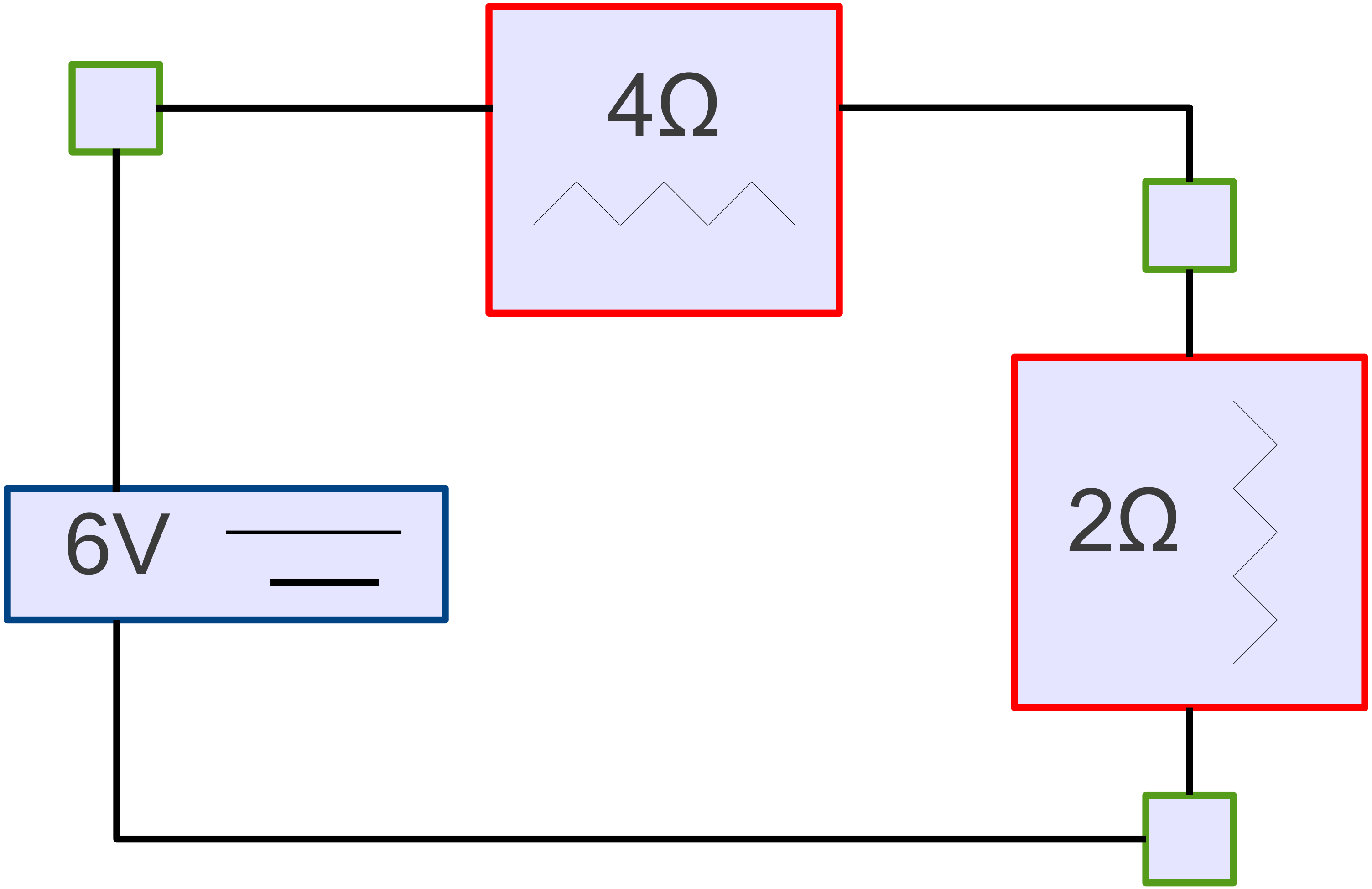

4Ω
6V
3Ω
6Ω


4Ω
6V
3Ω
6Ω

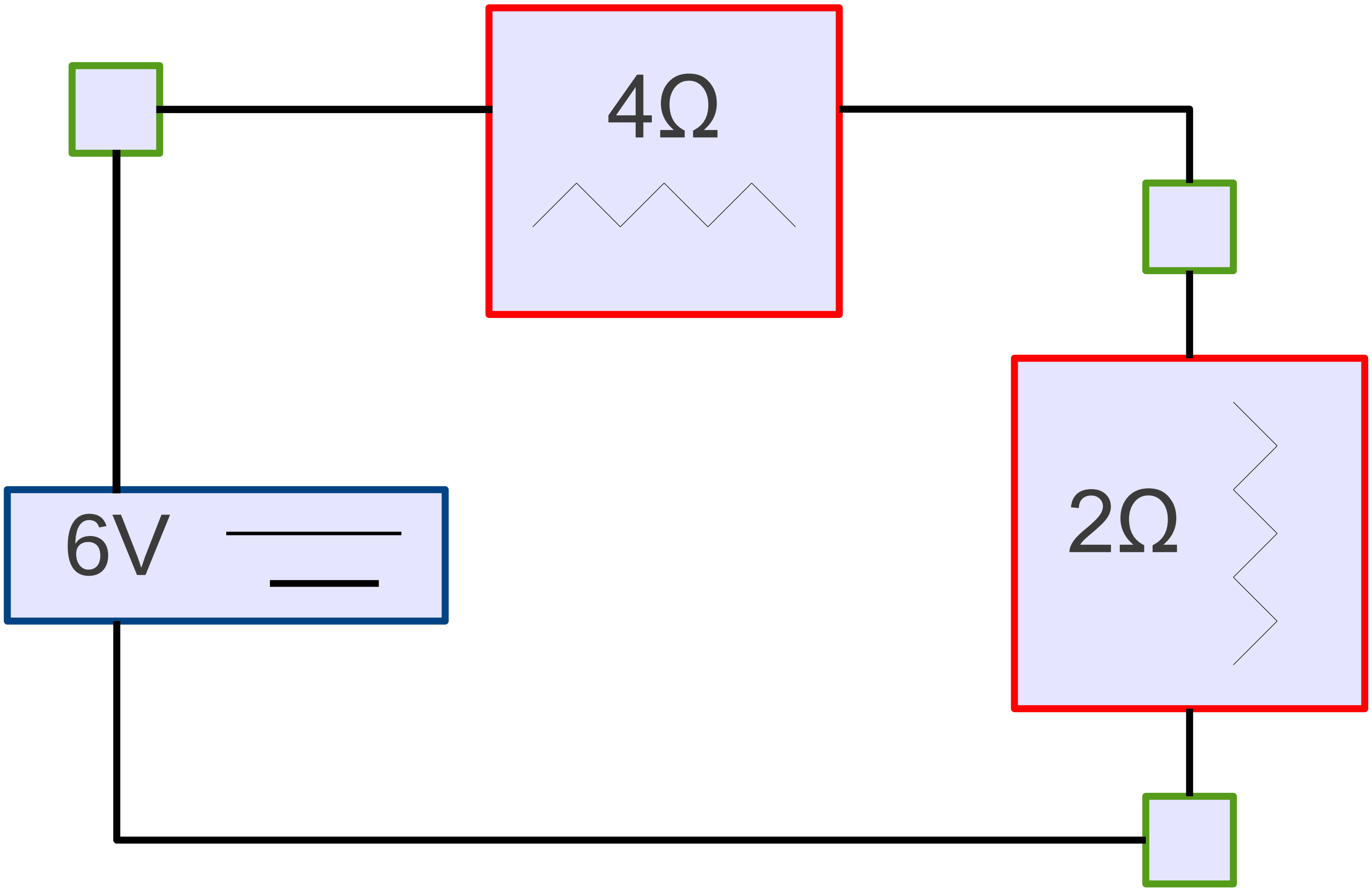

4Ω
6V
3Ω
6Ω

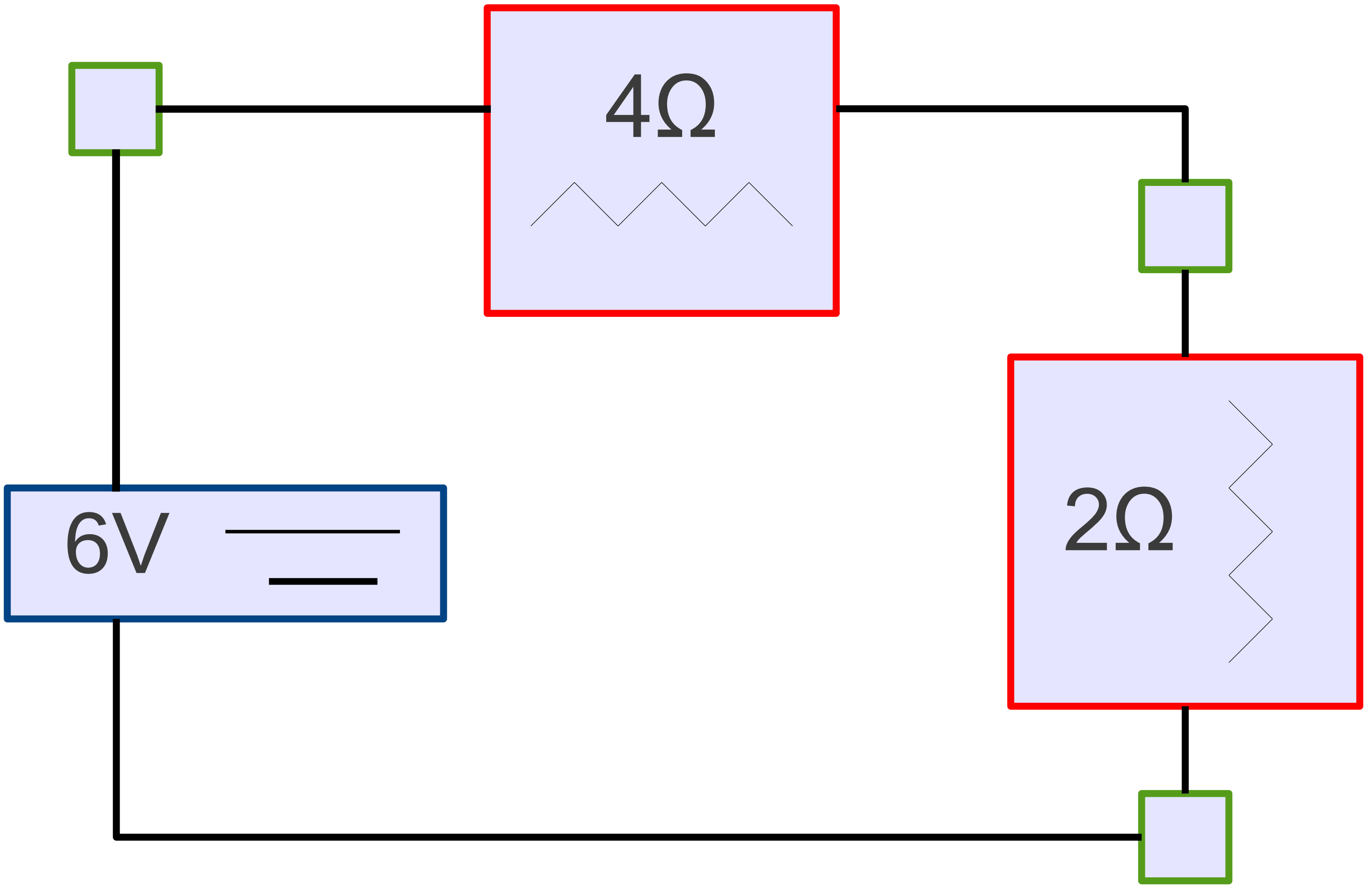


$$\frac{1}{\frac{1}{3\Omega}+\frac{1}{6\Omega}} = 2\ \Omega$$

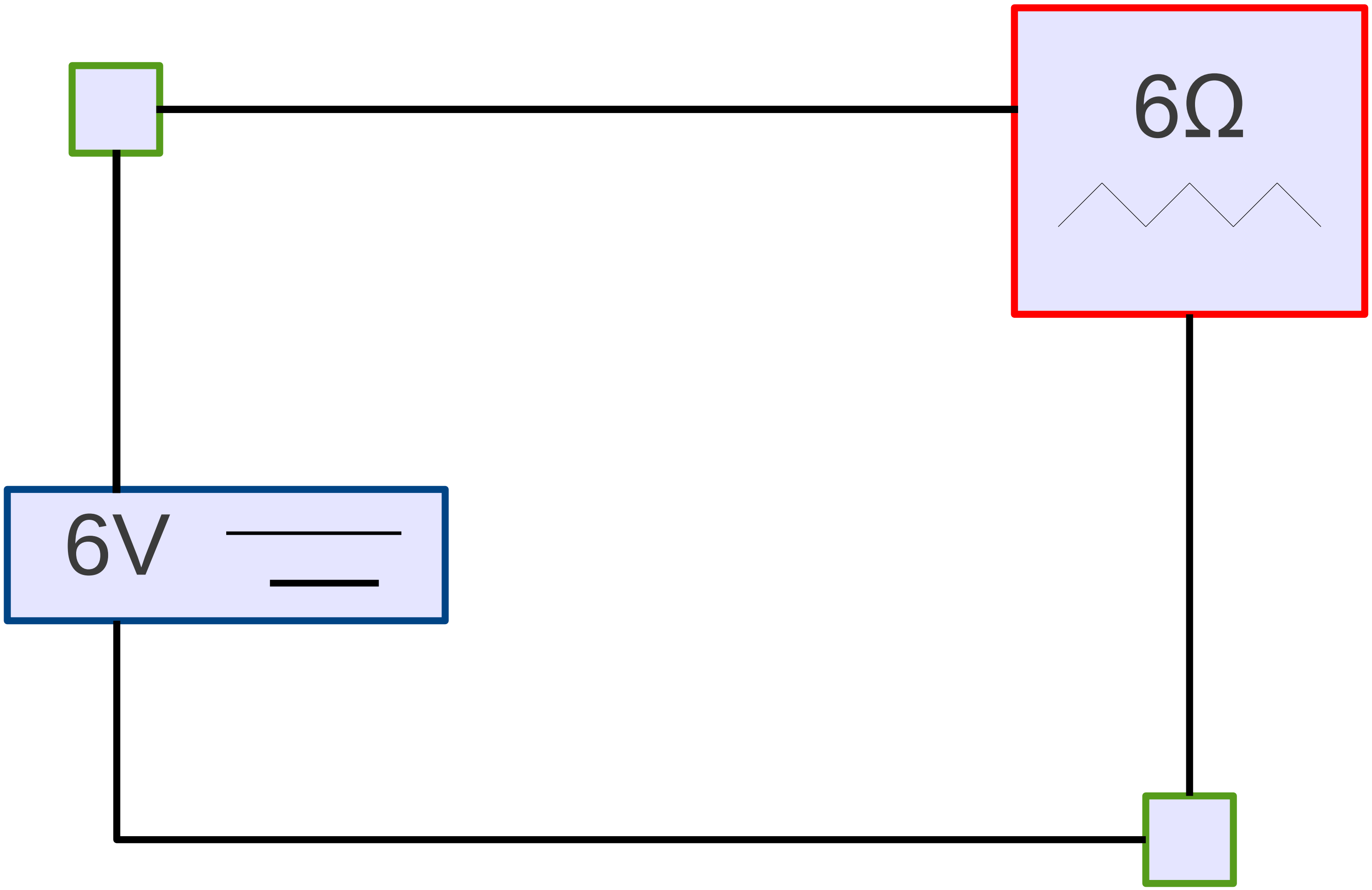


$$\frac{1}{\frac{1}{3\Omega}+\frac{1}{6\Omega}} = 2\ \Omega$$

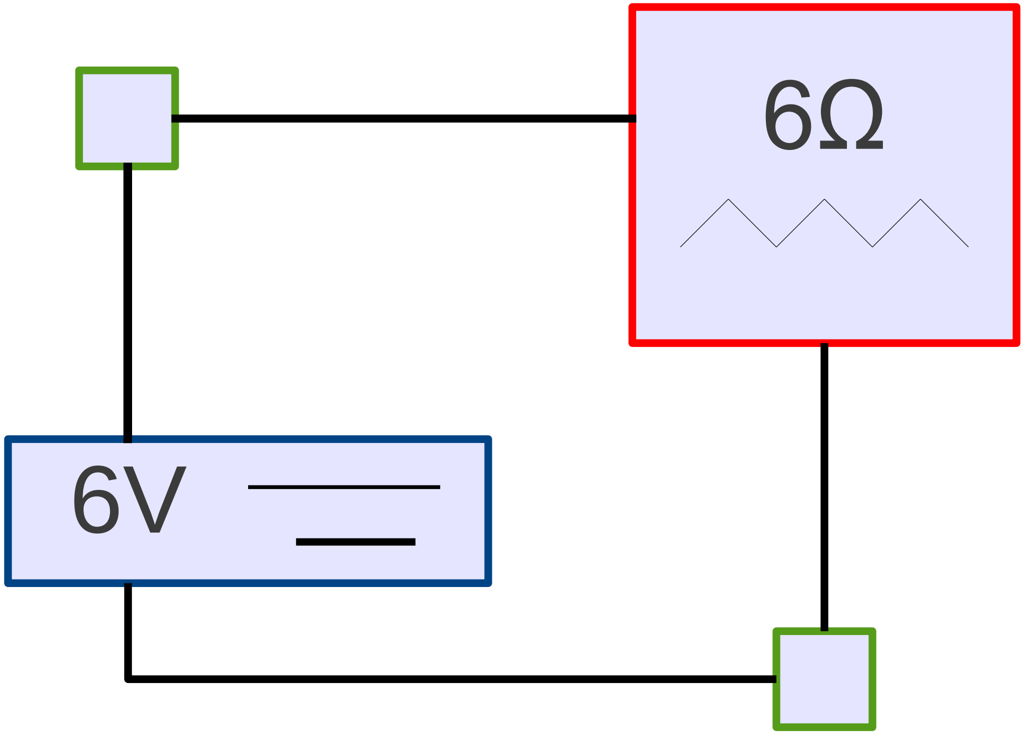

4Ω
6V
2Ω

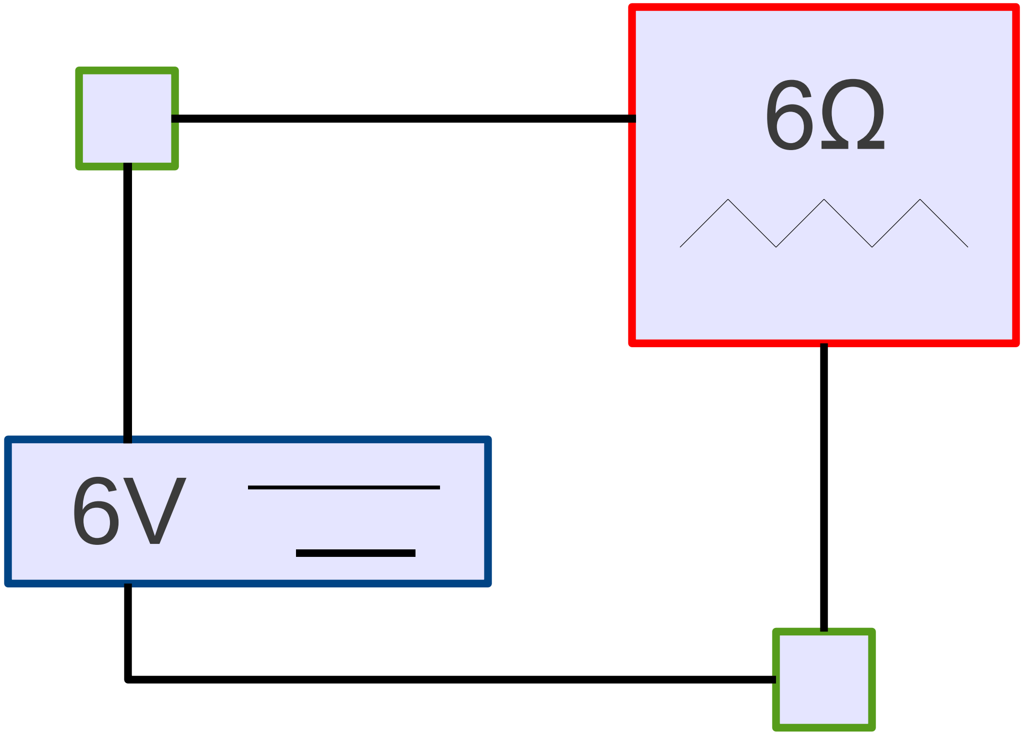


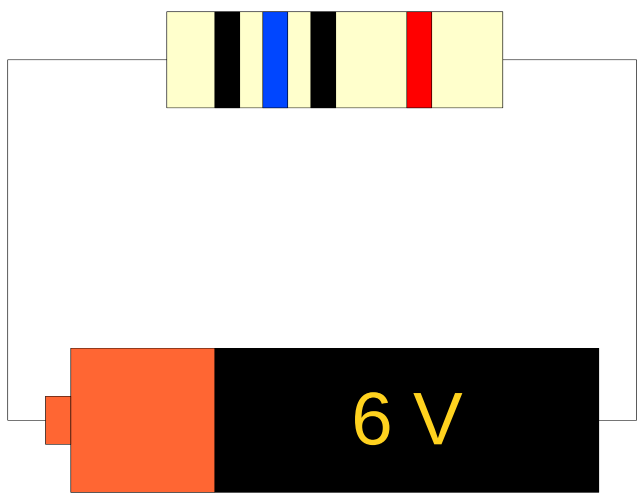
$$4\Omega + 2\Omega = 6\ \Omega$$

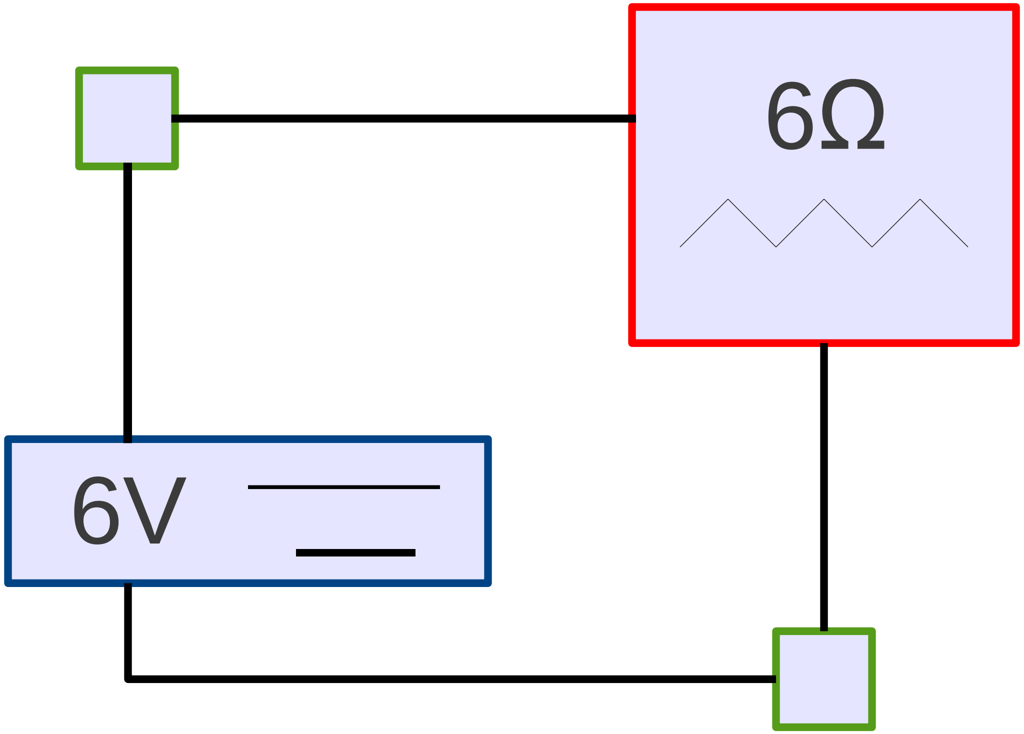


$$4\Omega + 2\Omega = 6\ \Omega$$

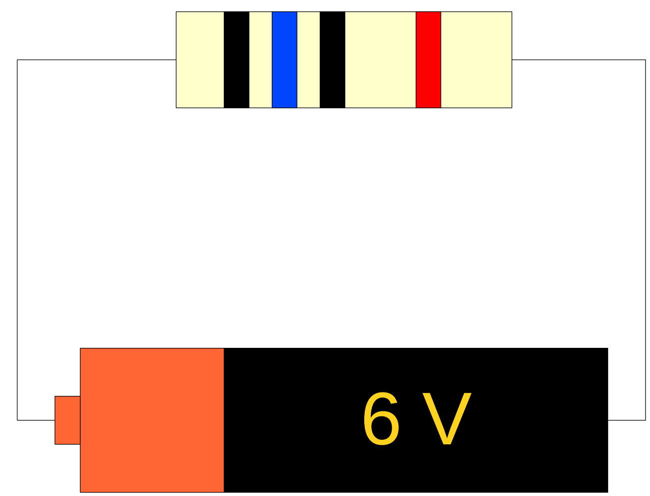

6Ω
6V

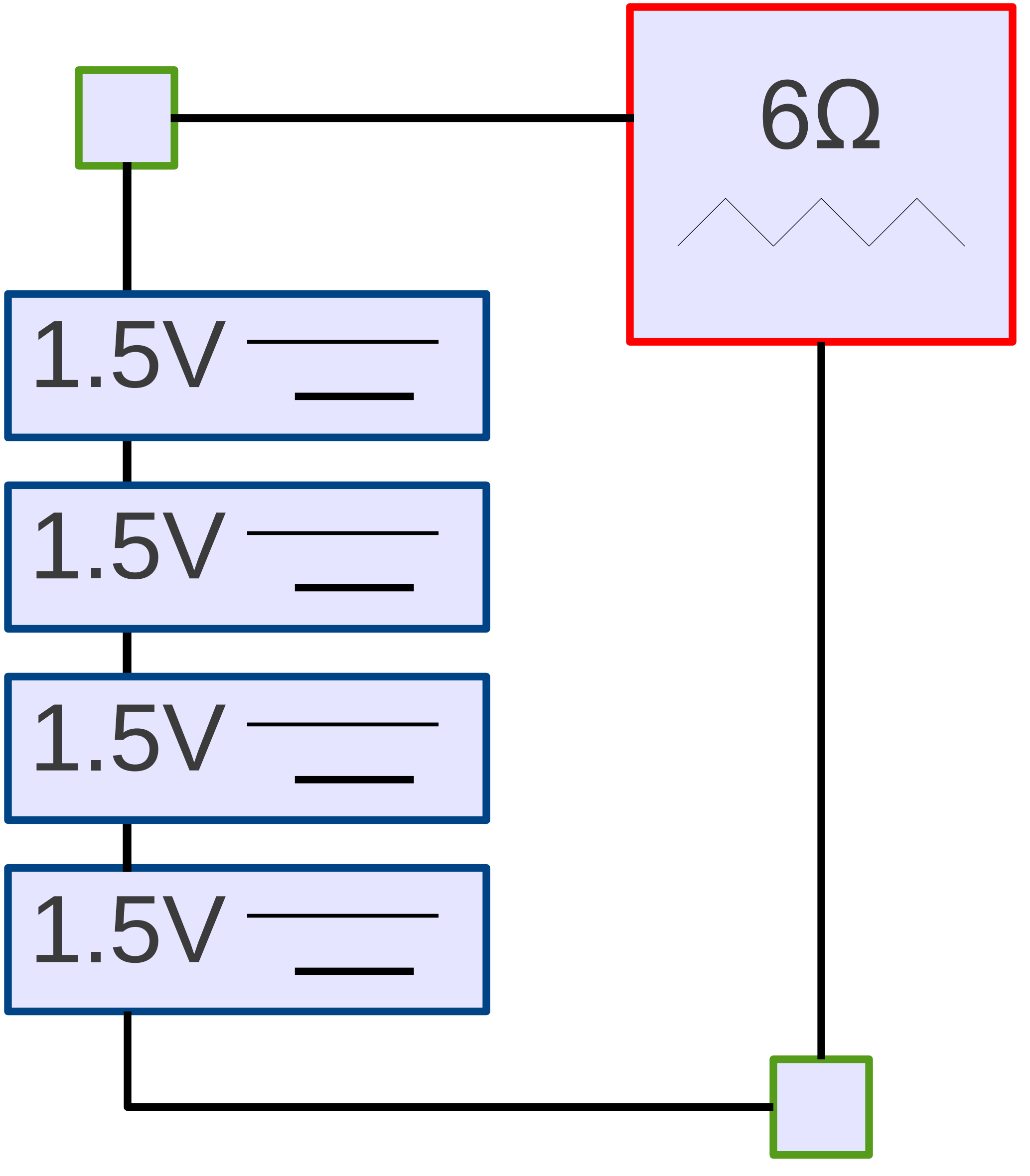

6Ω
6V

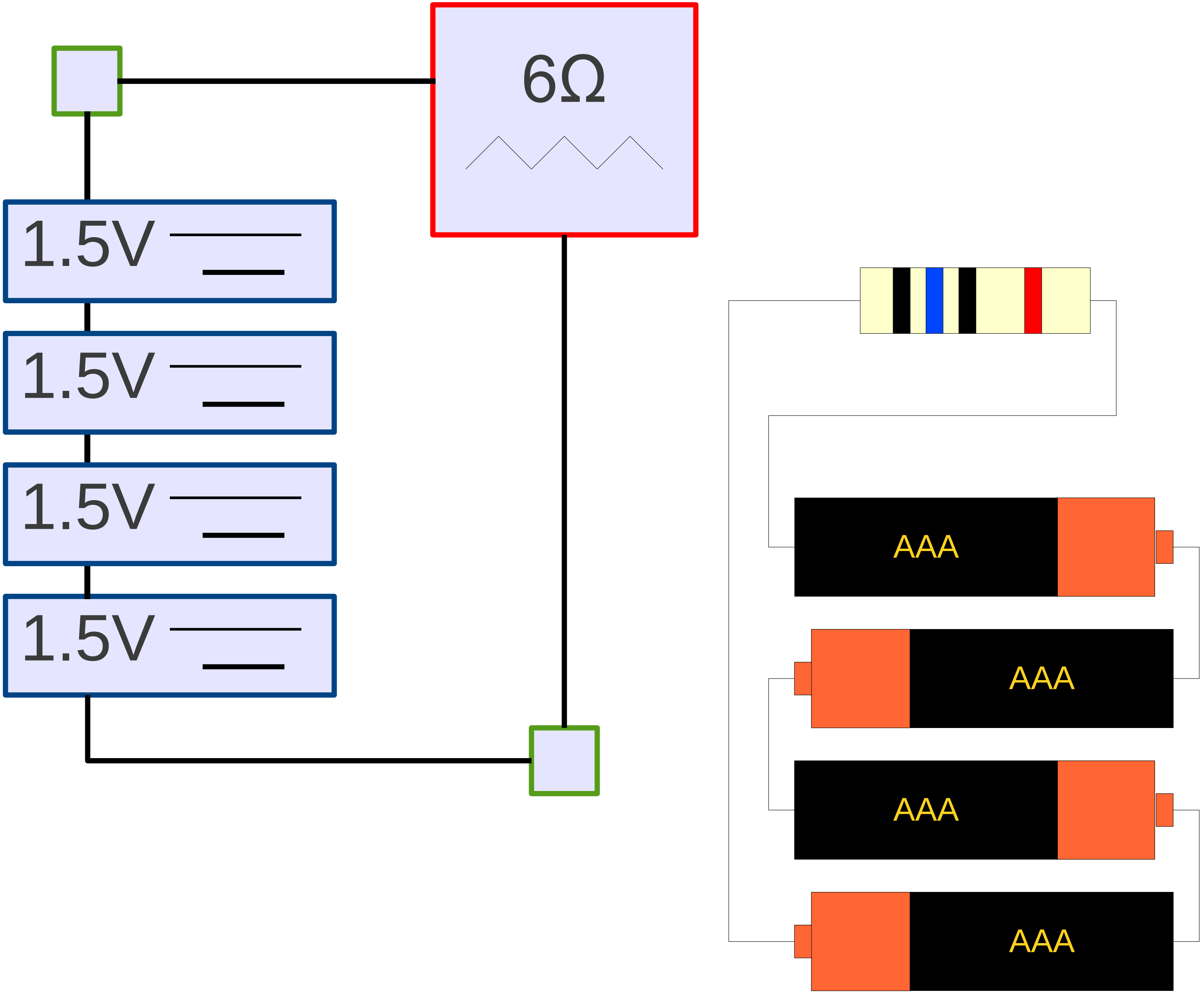

6 V

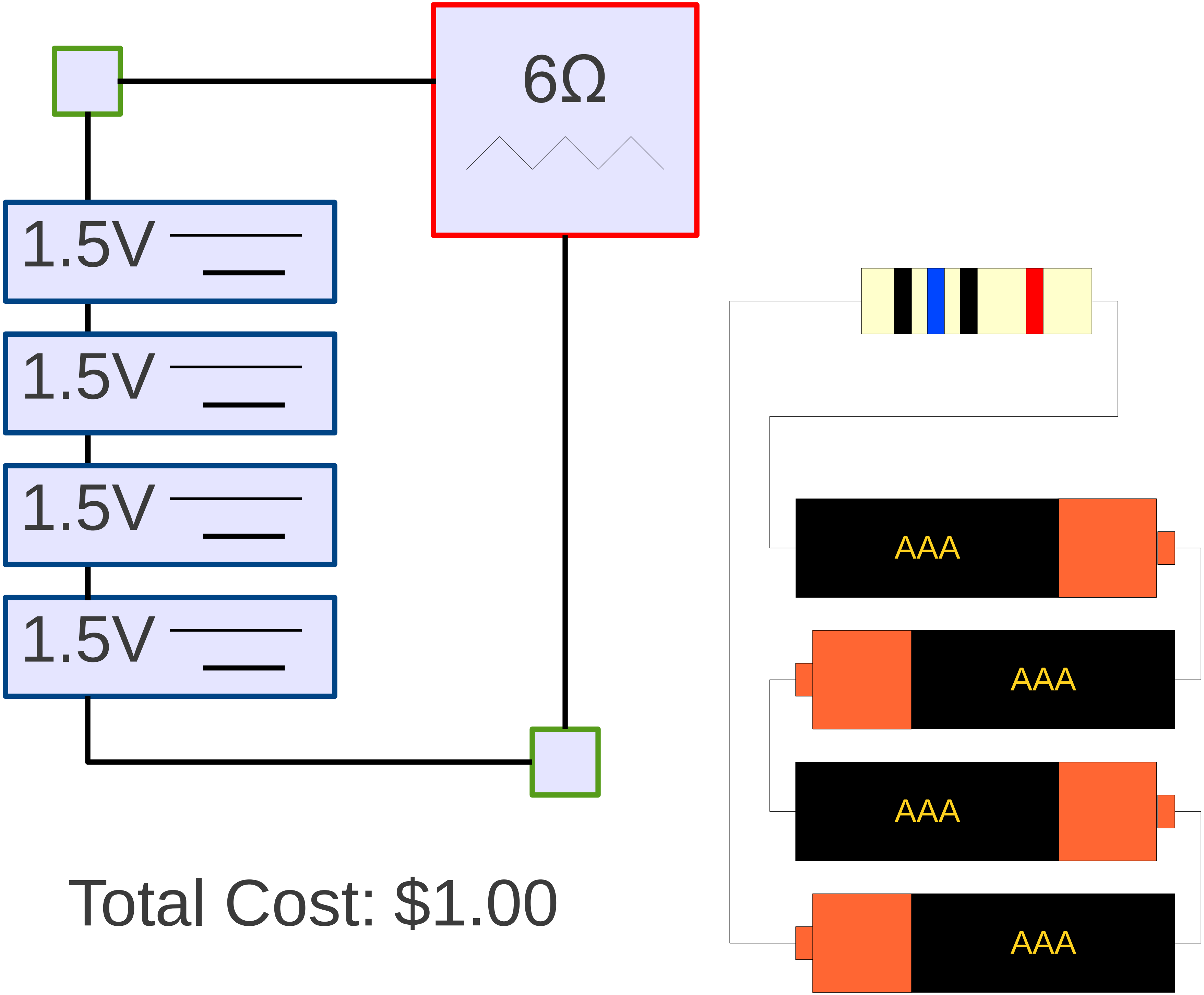


Total Cost: $4.75




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.

Lexical Analysis

- First step: recognize words.
  - Smallest unit above letters

This is a sentence.

More Lexical Analysis

- Lexical analysis is not trivial. Consider:

ist his ase nte nce

Real world Example: Watch it On YouTube!

And More Lexical Analysis

- Lexical analyzer divides program text into “words” or “tokens”

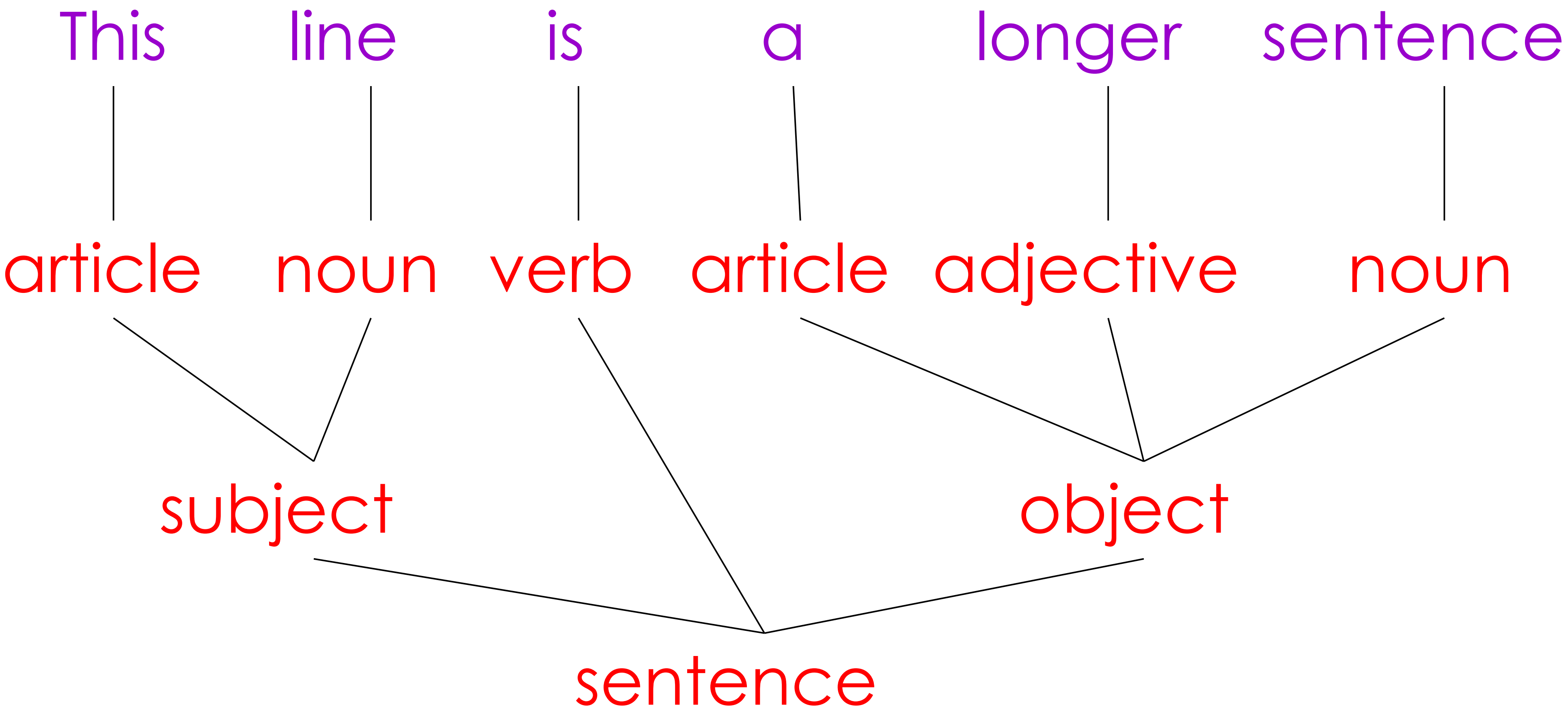
  If x == y then z = 1; else z = 2;

- Units:

Parsing

- Once words are understood, the next step is to understand sentence structure
- Parsing = Diagramming Sentences
  - The diagram is a tree

Diagramming a Sentence

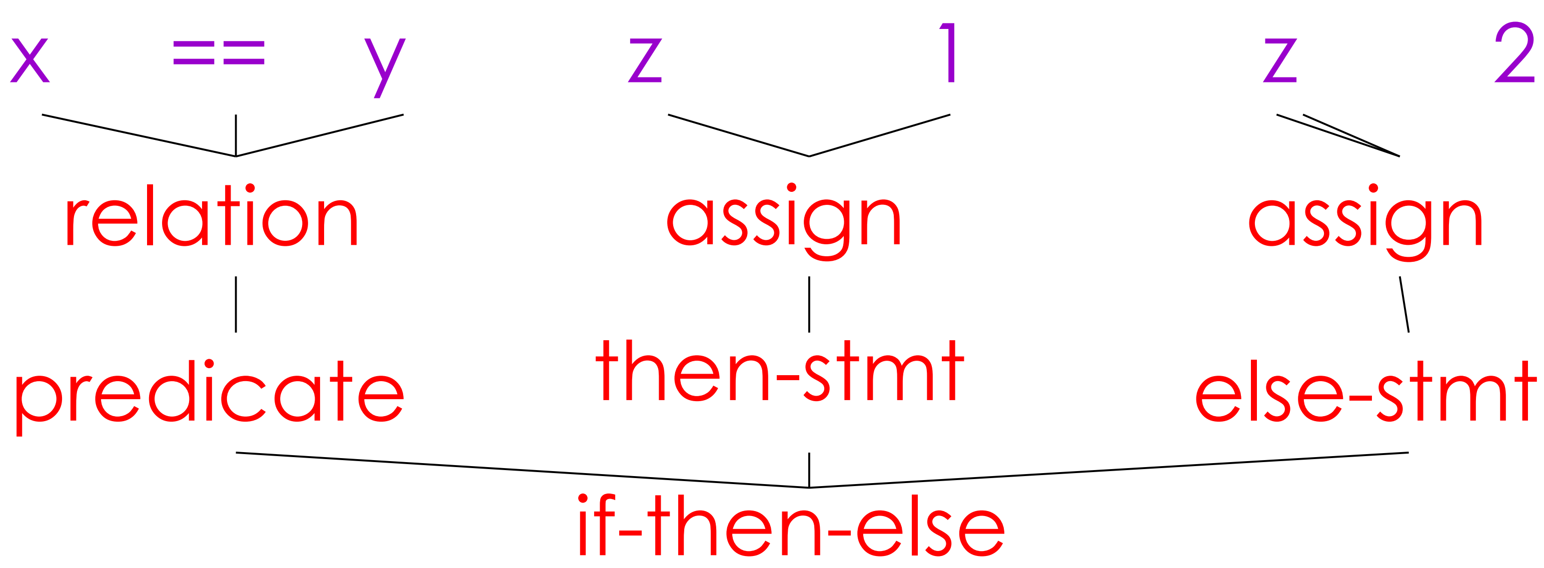


Parsing Programs

- Parsing program expressions is the same
- Consider:

If x == y then z = 1; else z = 2;

- Diagrammed:



Semantic Analysis

- Once sentence structure is understood, we can try to understand "meaning"
  - But meaning is too hard for compilers

- Compilers perform limited semantic analysis to catch inconsistencies

Semantic Analysis in English

- Example:

  Jack said Jerry left his assignment at home.

  What does "his" refer to? Jack or Jerry?

- Even worse:

  Jack said Jack left his assignment at home?

  How many Jacks are there?

  Which one left the assignment?

Semantic Analysis in Programming

- Programming languages define strict rules to avoid such ambiguities

- This C++ code prints "4"; the inner definition is used

```
{
   int Jack = 3;
   {
      int Jack = 4;
      cout << Jack;
   }
}
```

More Semantic Analysis

- Compilers perform many semantic checks besides variable bindings

- Example:

  Jack left her homework at home.

- Possible type mismatch between her and Jack
  - If Jack is male

Optimization

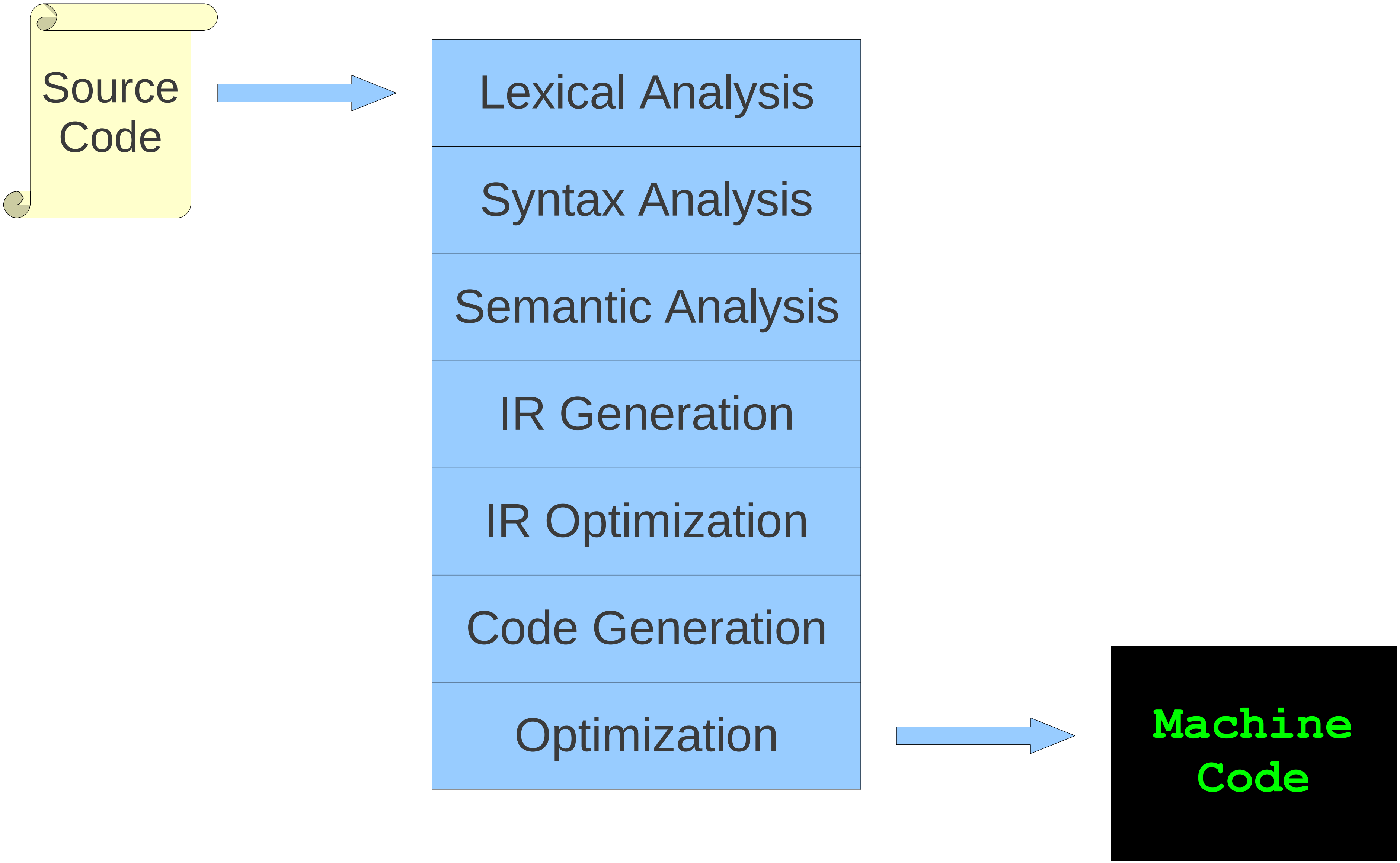
- No strong counterpart in English, but akin to editing
- Automatically modify programs so that they
  - Run faster
  - Use less memory
  - In general, to use or conserve some resource

Optimization Example

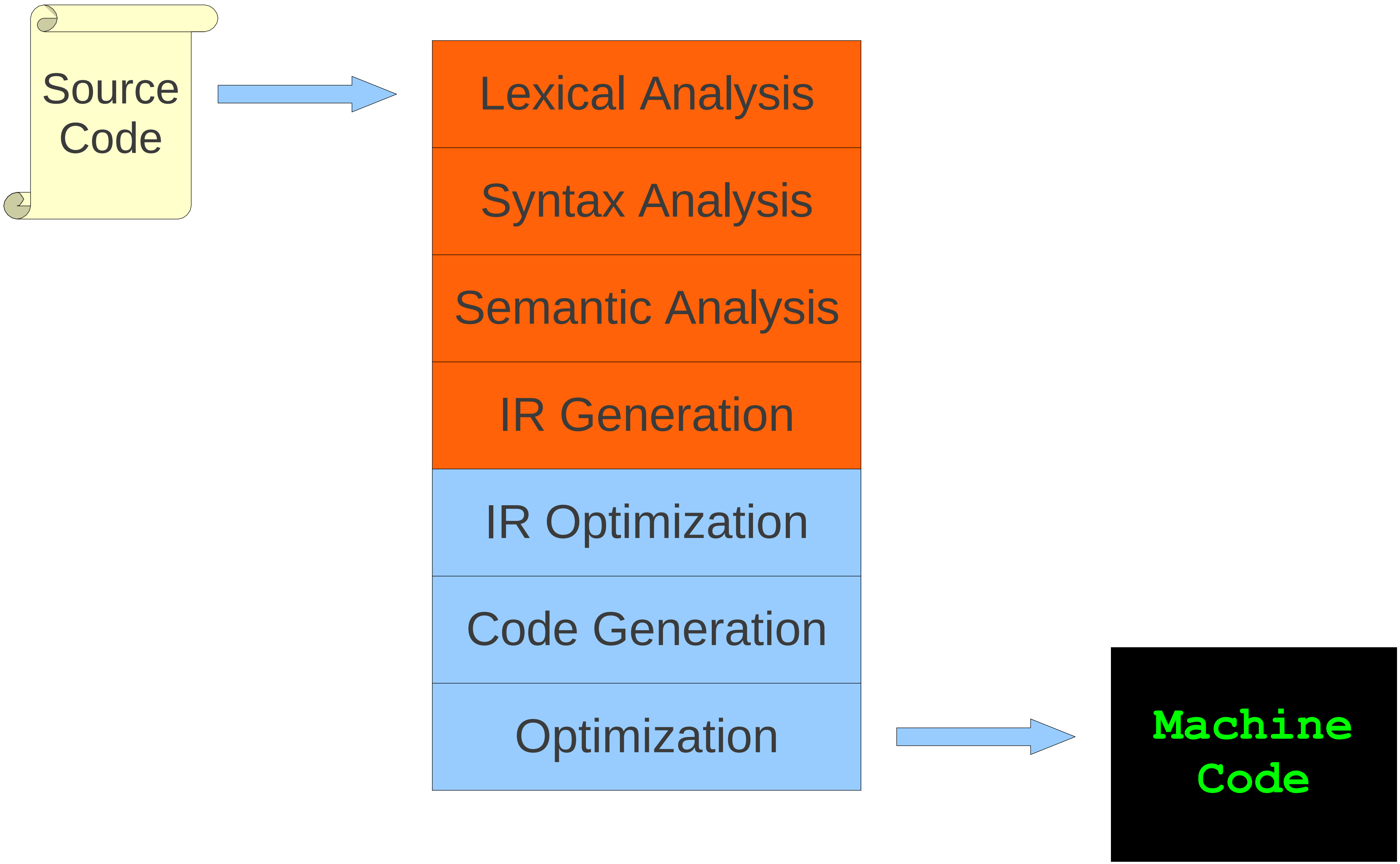
X = Y * 0   is the same as   X = 0

(the * operator is annihilated by zero)

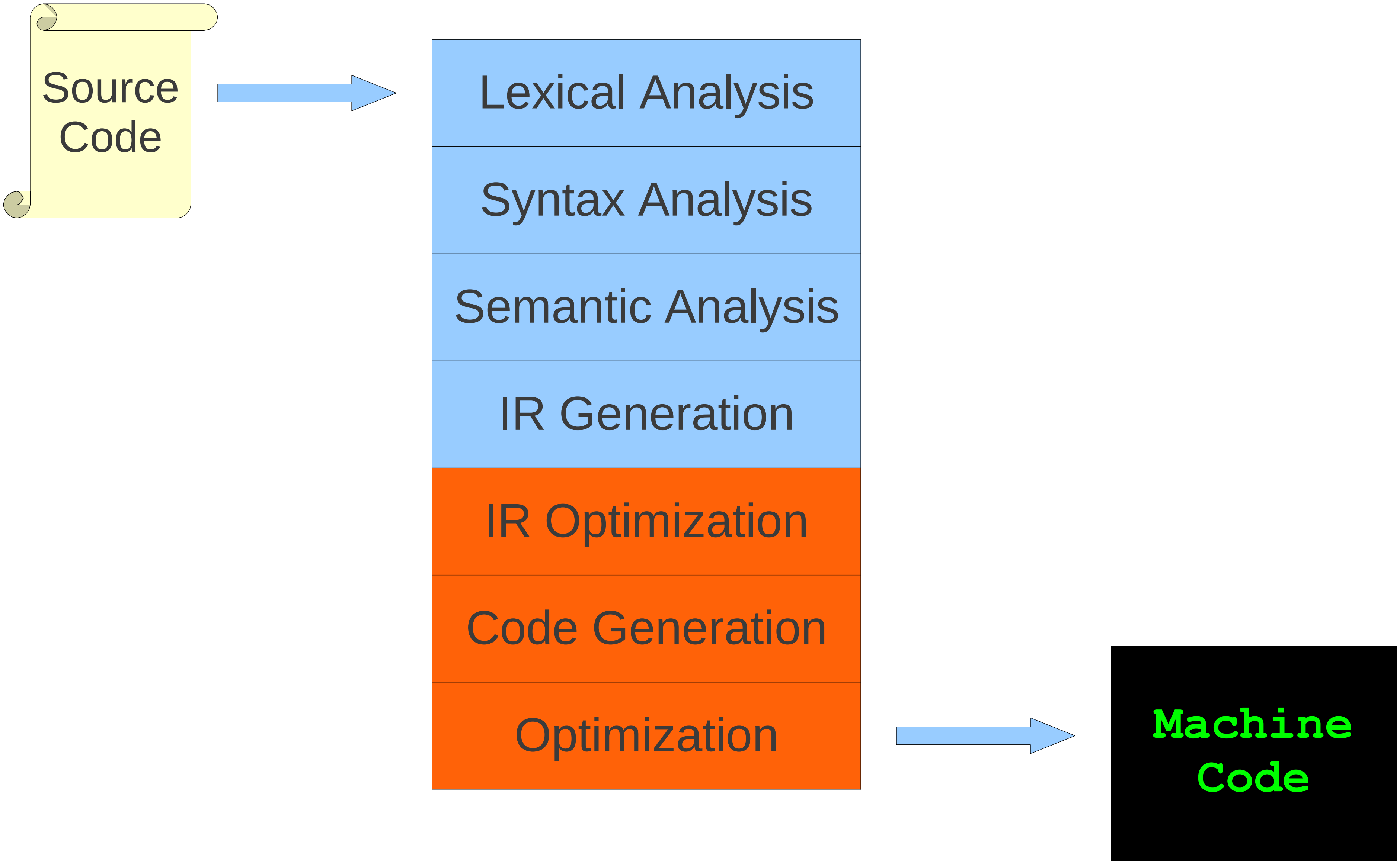
The Structure of a Modern Compiler



The Structure of a Modern Compiler



The Structure of a Modern Compiler



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

Lexical Analysis

Syntax Analysis

Semantic Analysis

IR Generation

IR Optimization

Code Generation

Optimization

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

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

```
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
```

Lexical Analysis

Syntax Analysis

Semantic Analysis

IR Generation

IR Optimization

Code Generation

Optimization

```
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
```

Lexical Analysis

Syntax Analysis

Semantic Analysis

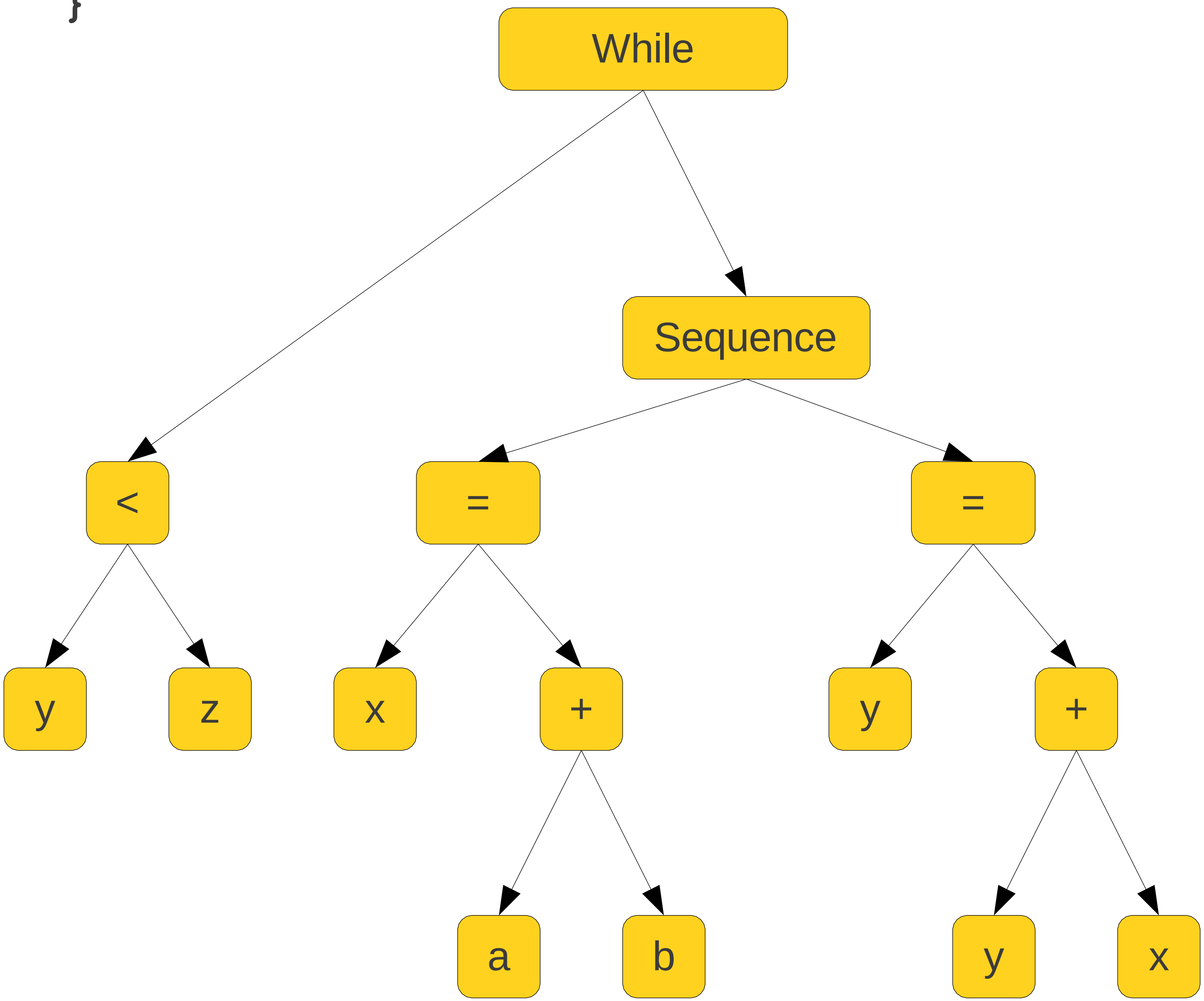
IR Generation

IR Optimization

Code Generation

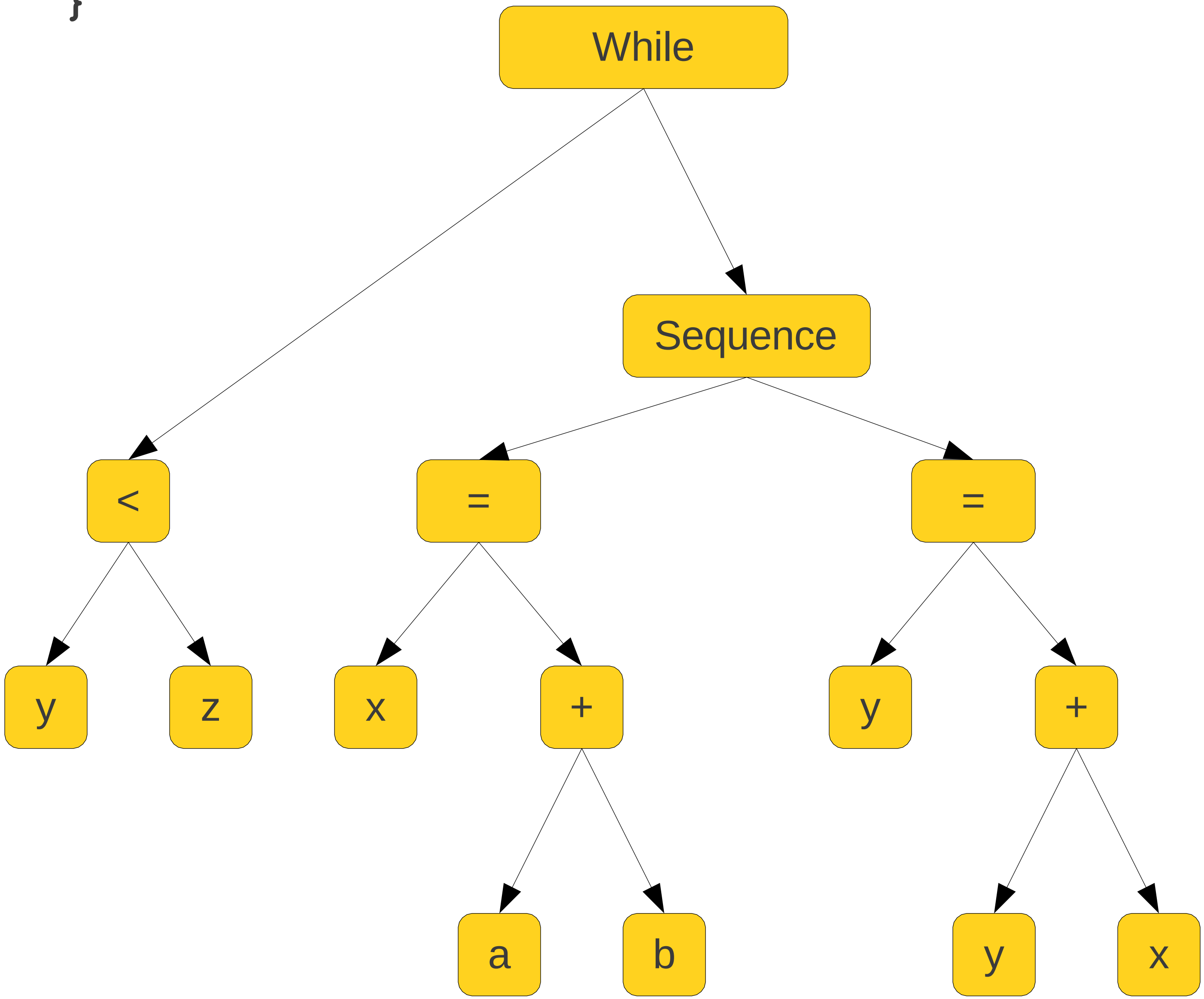
Optimization

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



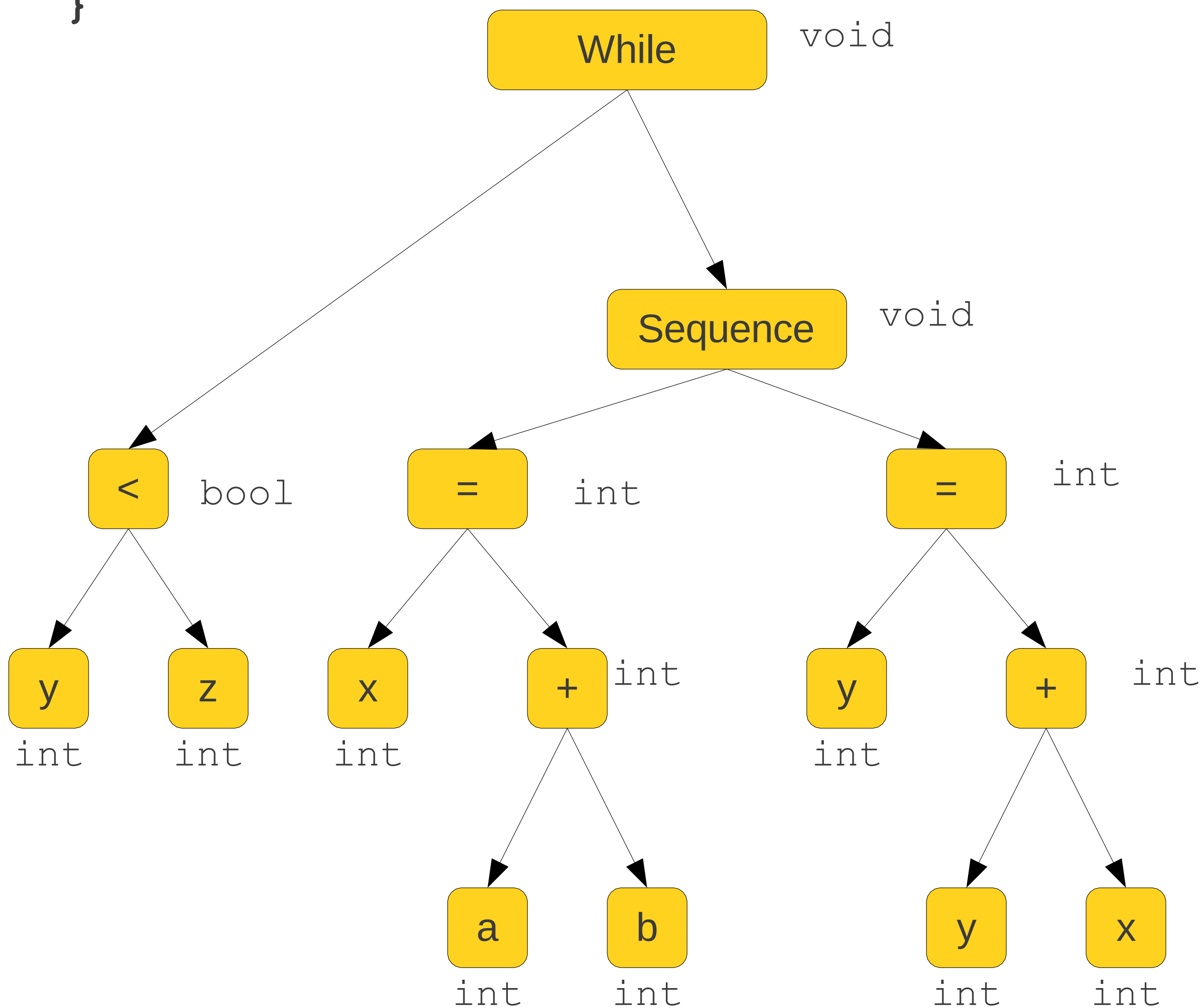
- Lexical Analysis
- **Syntax Analysis**
- Semantic Analysis
- IR Generation
- IR Optimization
- Code Generation
- Optimization

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



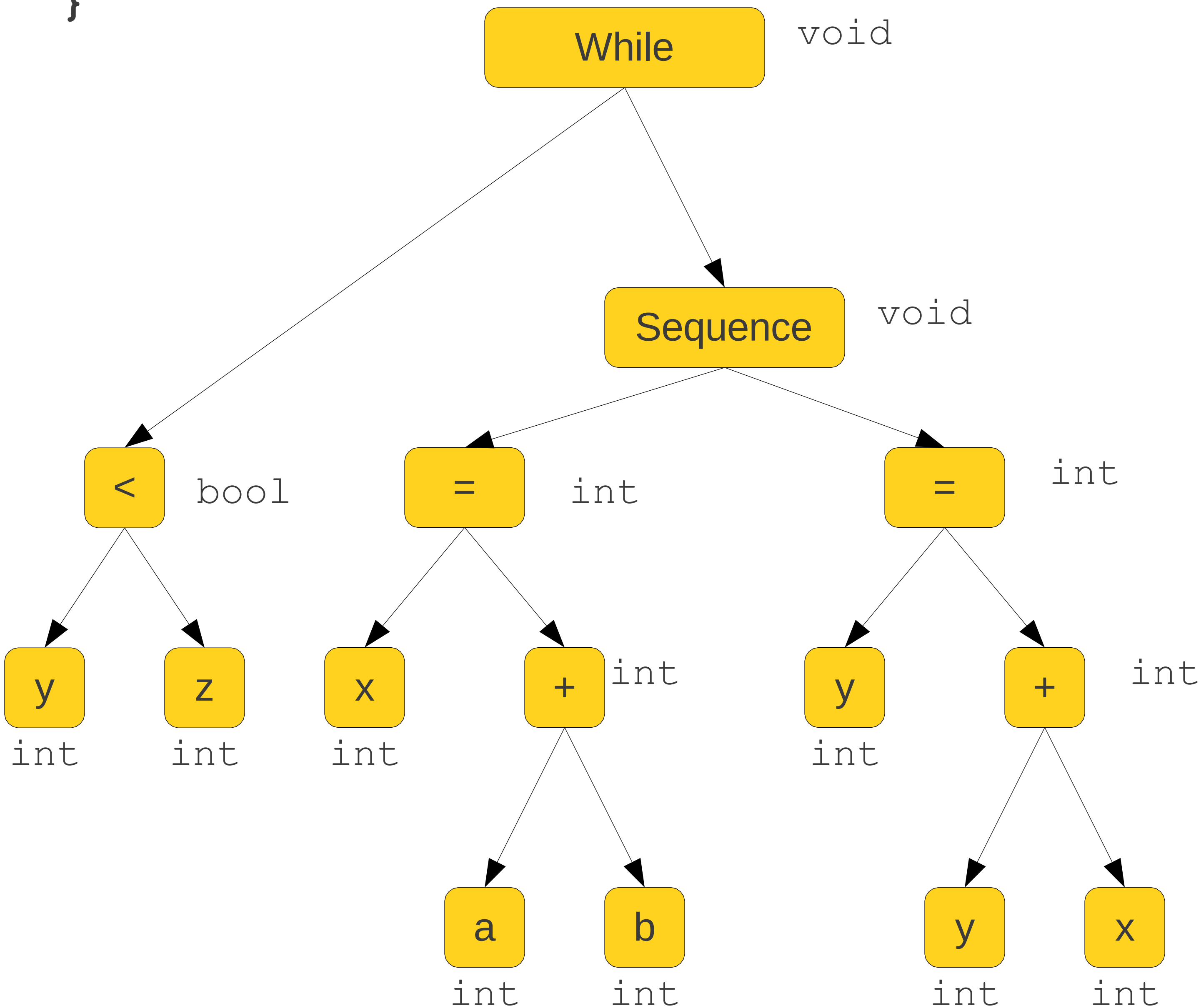
- Lexical Analysis
- Syntax Analysis
- **Semantic Analysis**
- IR Generation
- IR Optimization
- Code Generation
- Optimization

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



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

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



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
```

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
```

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

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

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

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

```
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
```

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

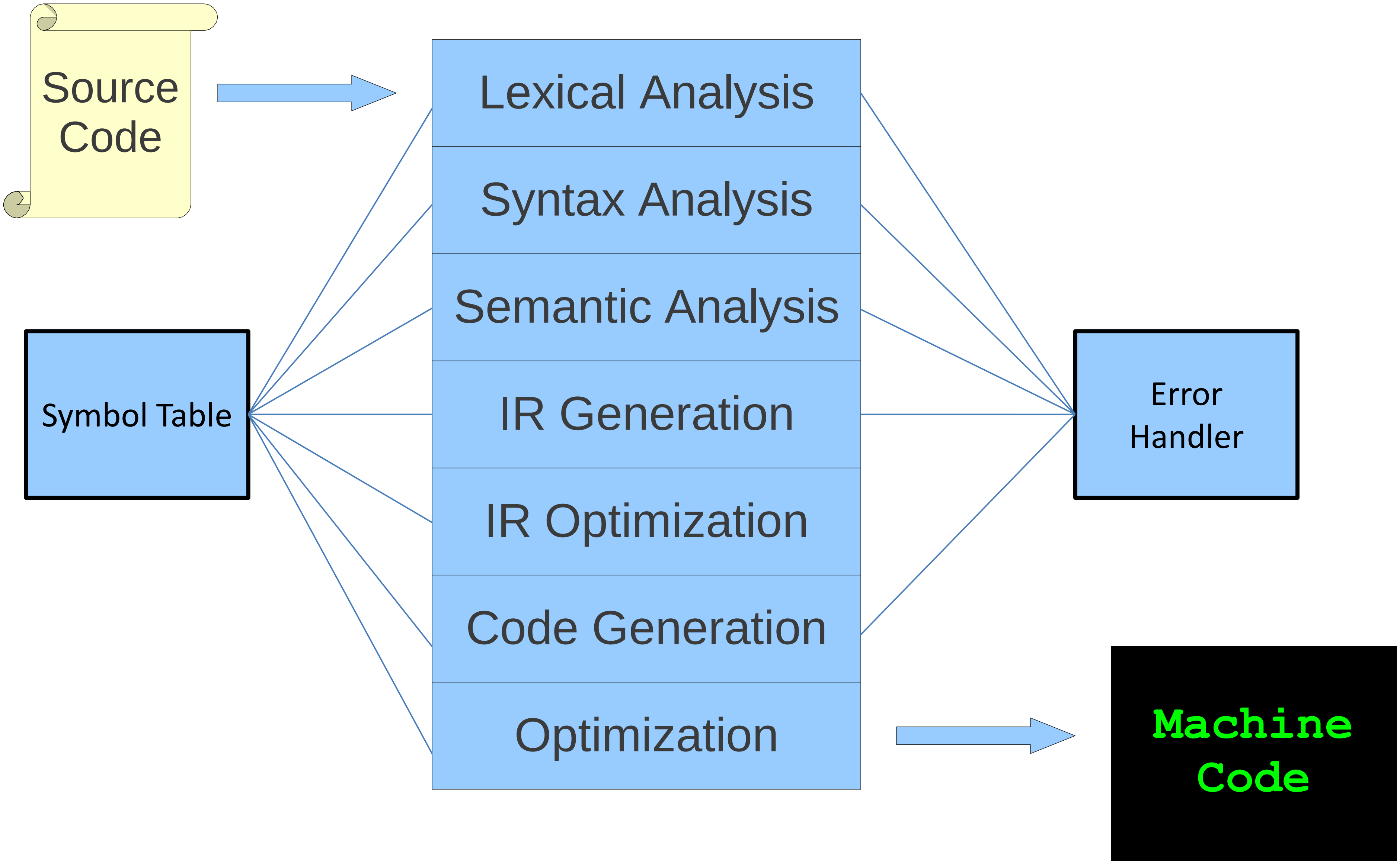
```
      add $1, $2, $3
Loop: add $4, $1, $4
      slt $6, $4, $5
      beq $6, loop
```

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

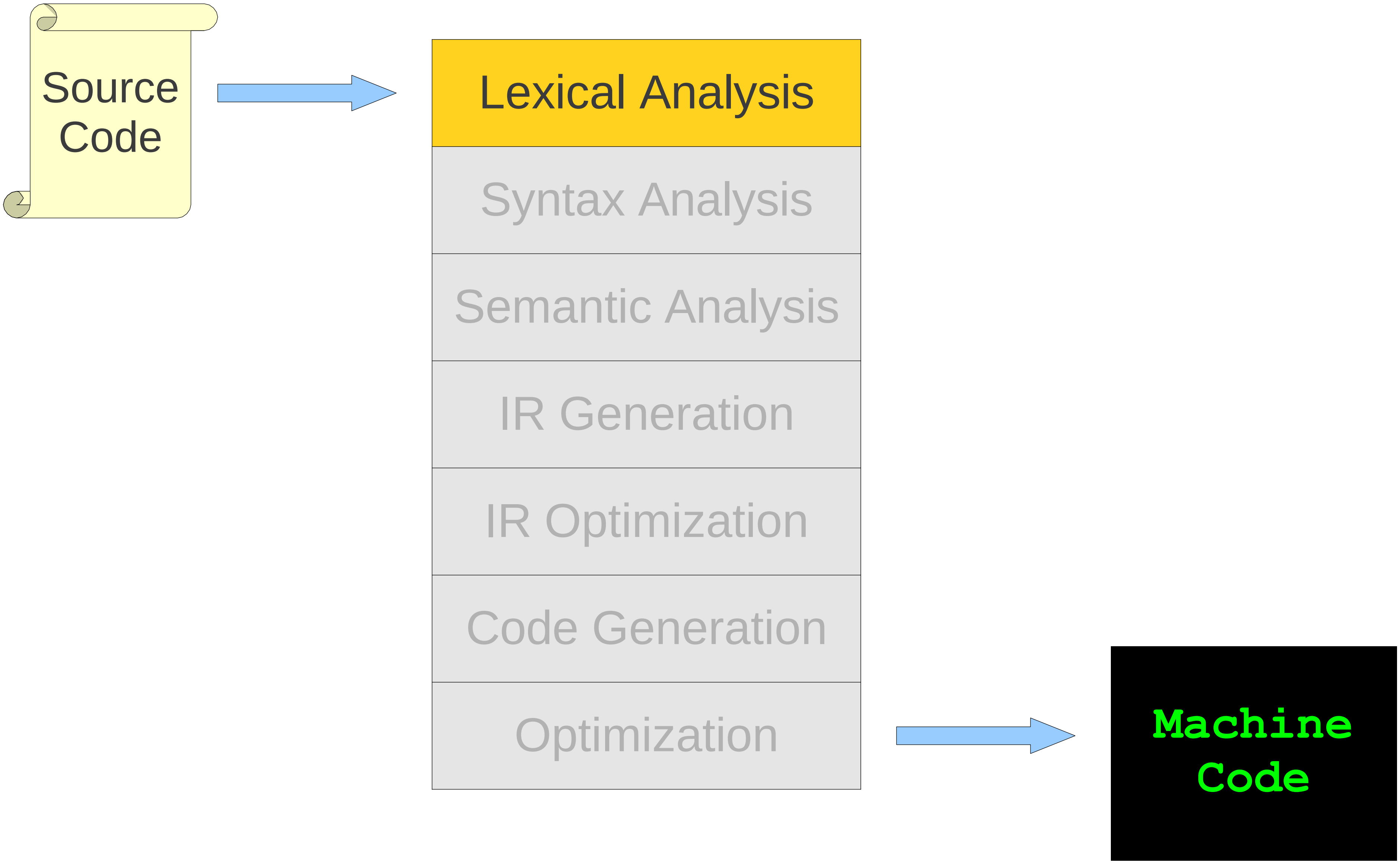
```
while (y < z) {
    int x = a + b;
    y += x;
}
```

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

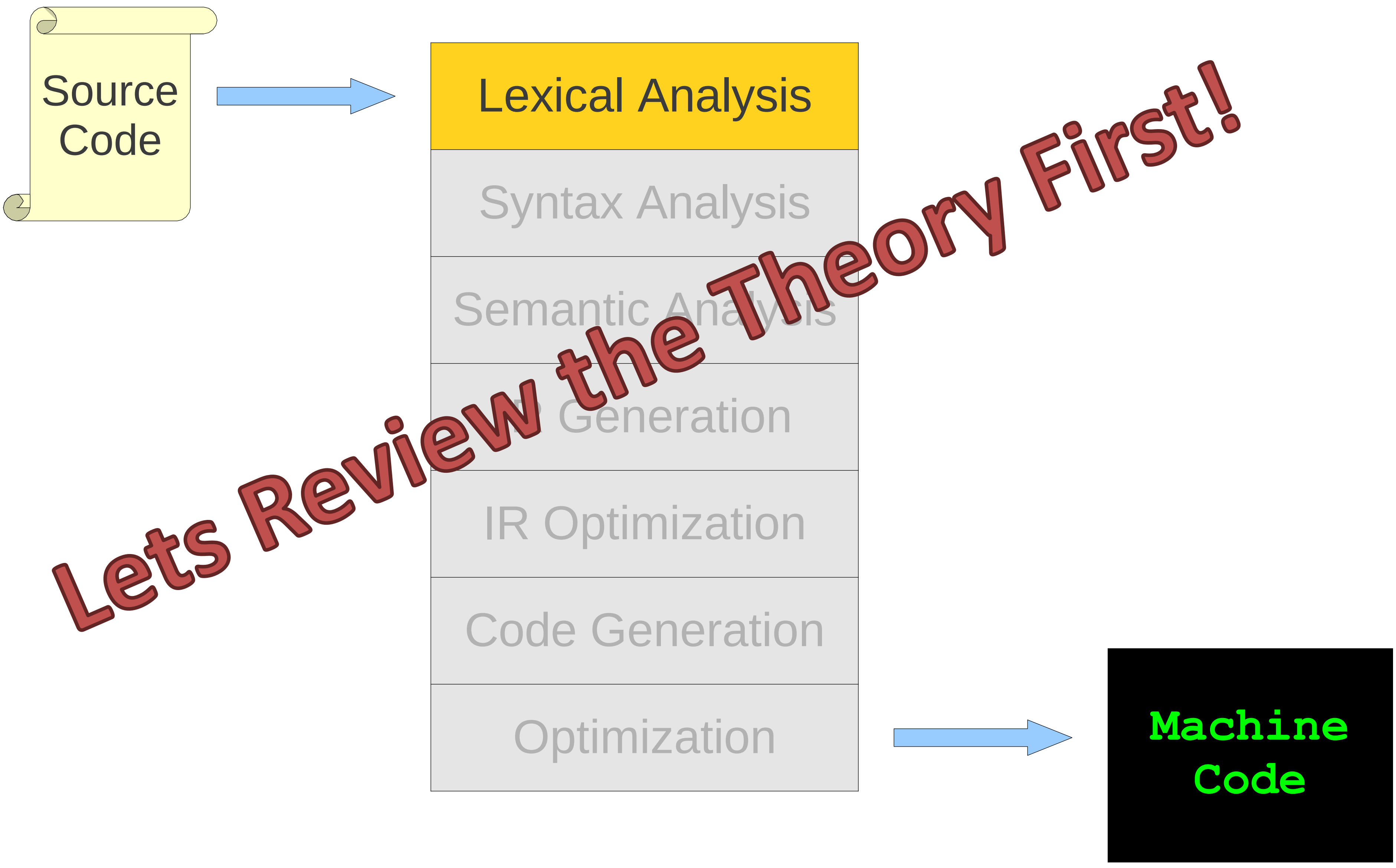
Structure of a Modern Compiler



Next Time...


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

Next Time...


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code
Lets Review the Theory First!