Introduction to

# Compiler construction

**http://cs143.stanford.edu**

# Why Study Compilers?

- Build a **large, ambitious software system**.
- See theory **come to life**.
- Learn how to **build programming languages**.
- Learn **how programming languages work**.
- Learn **tradeoffs in language design**.

# A Short History of Compilers

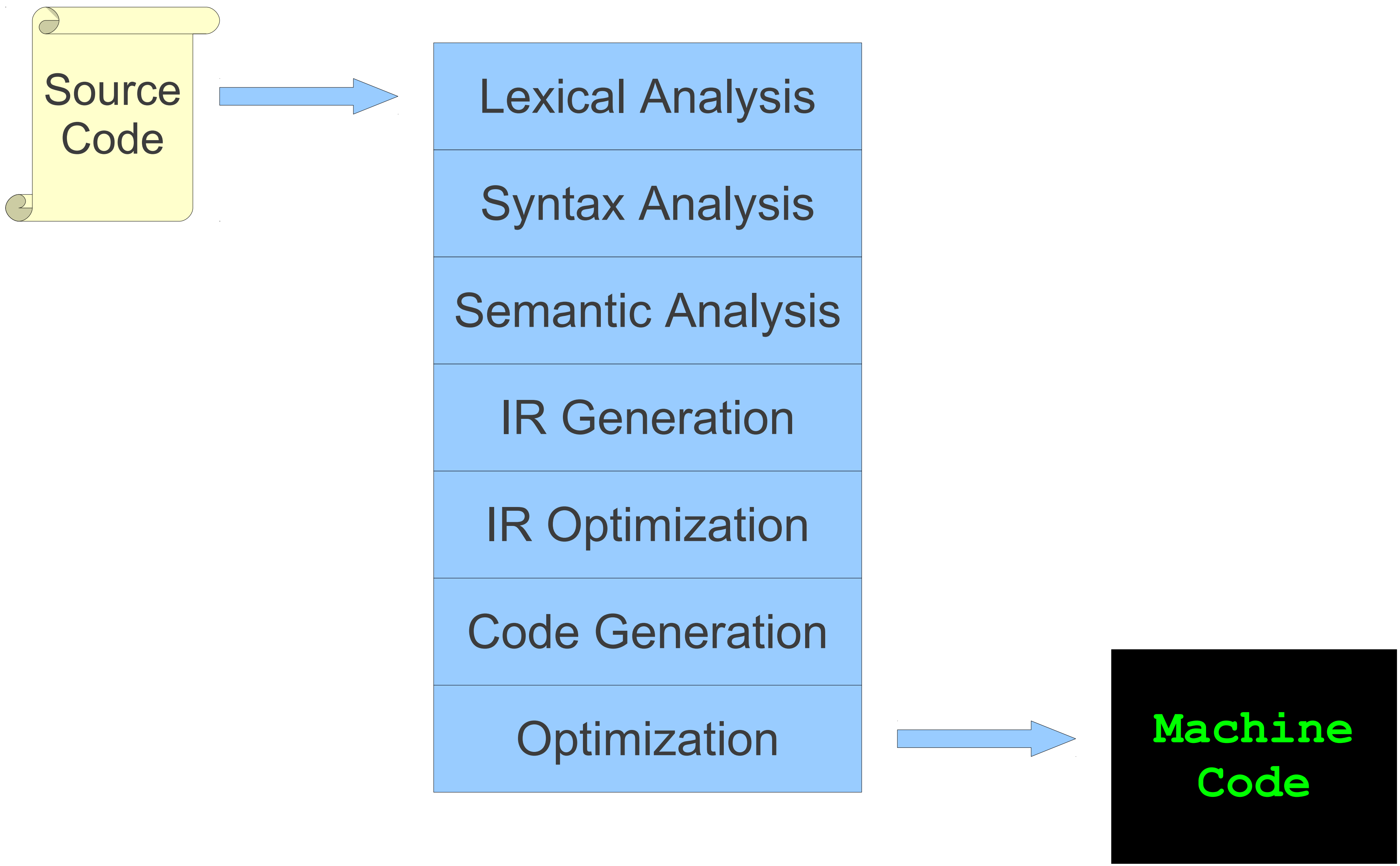
- First, there was nothing.
- Then, there was machine code.
- Then, there were assembly languages.
- Programming expensive; 50% of costs for machines went into programming.

# How does a compiler work?

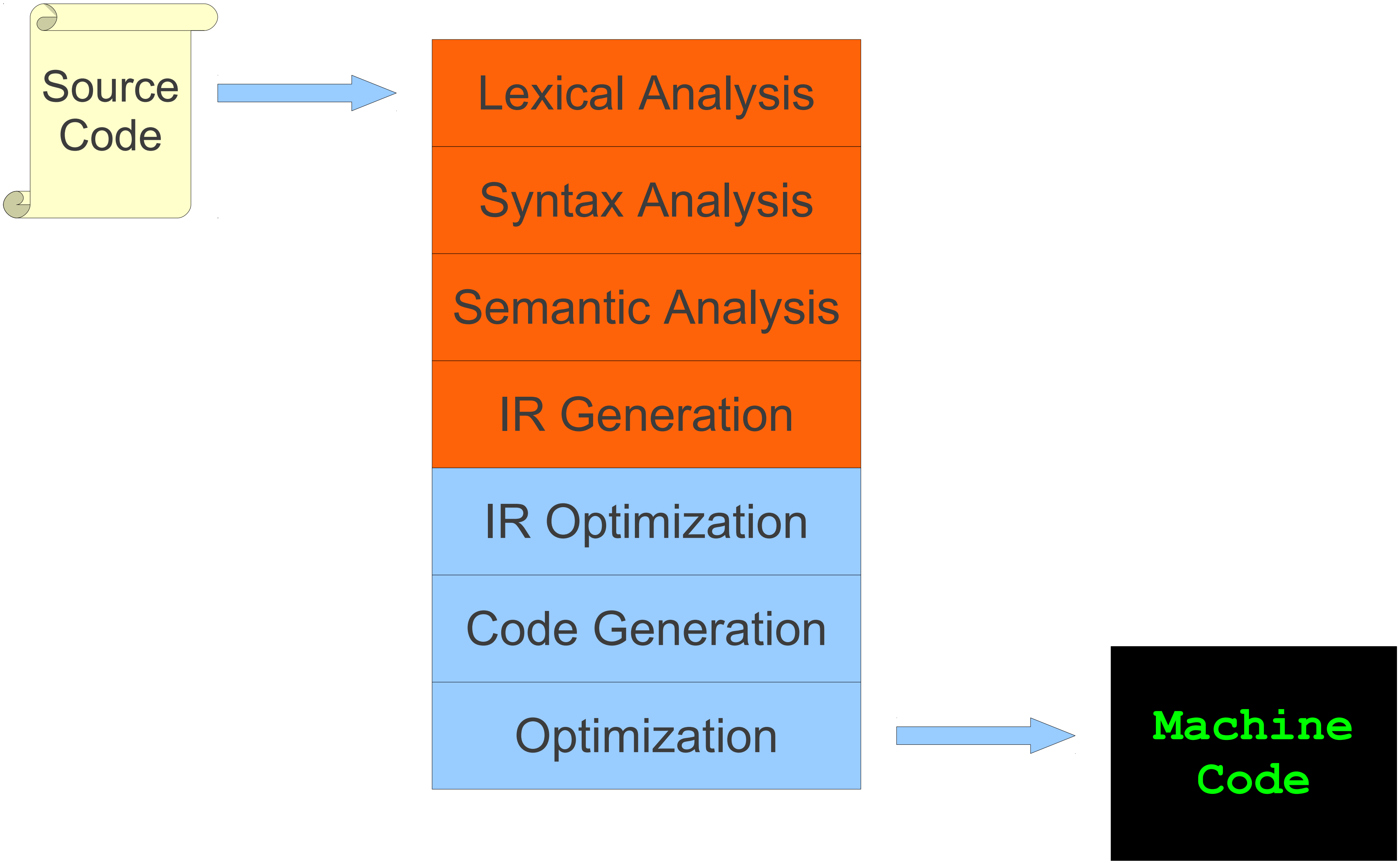
# From Description to Implementation

- **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.

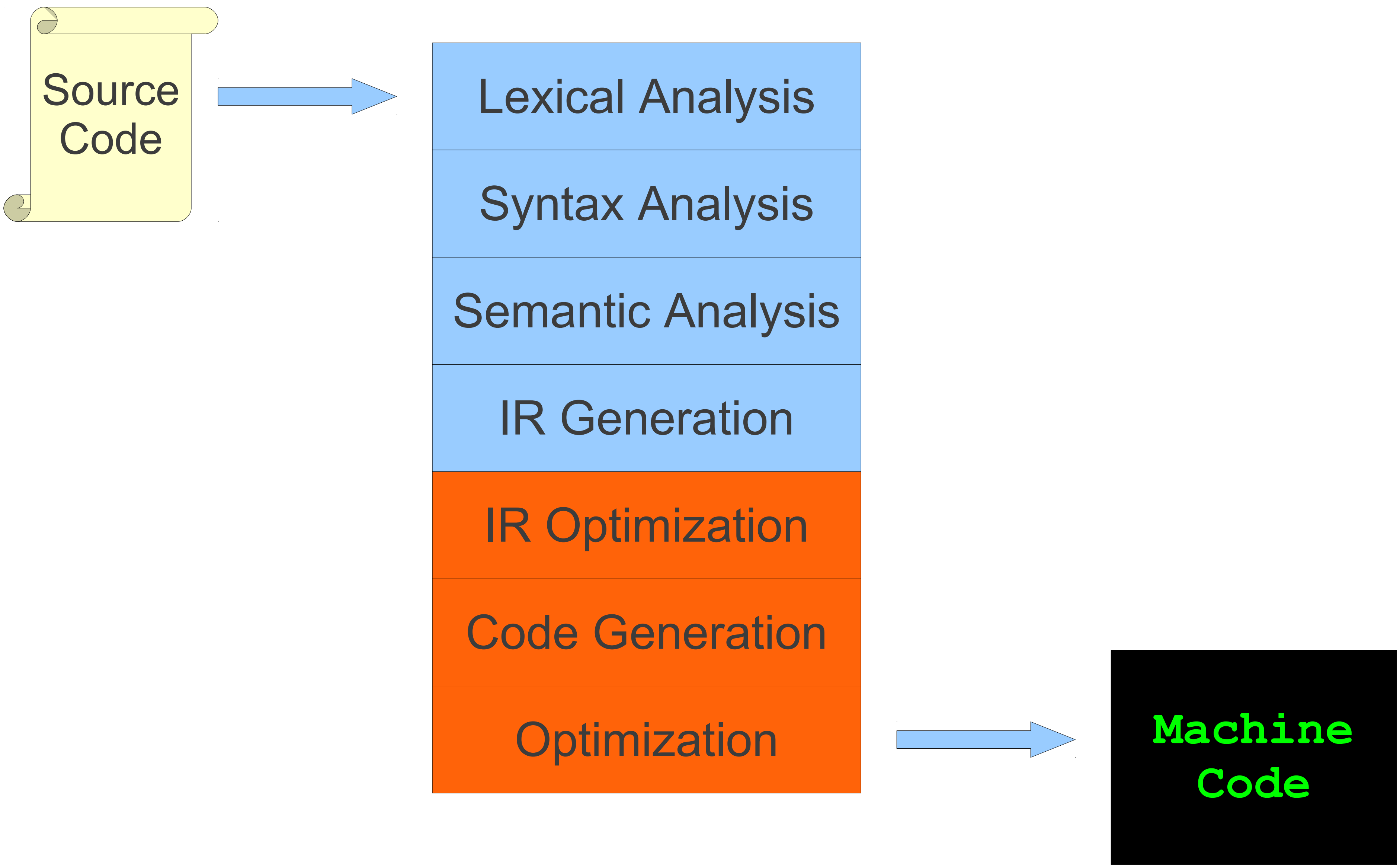
# The Structure of a Modern Compiler



# The 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

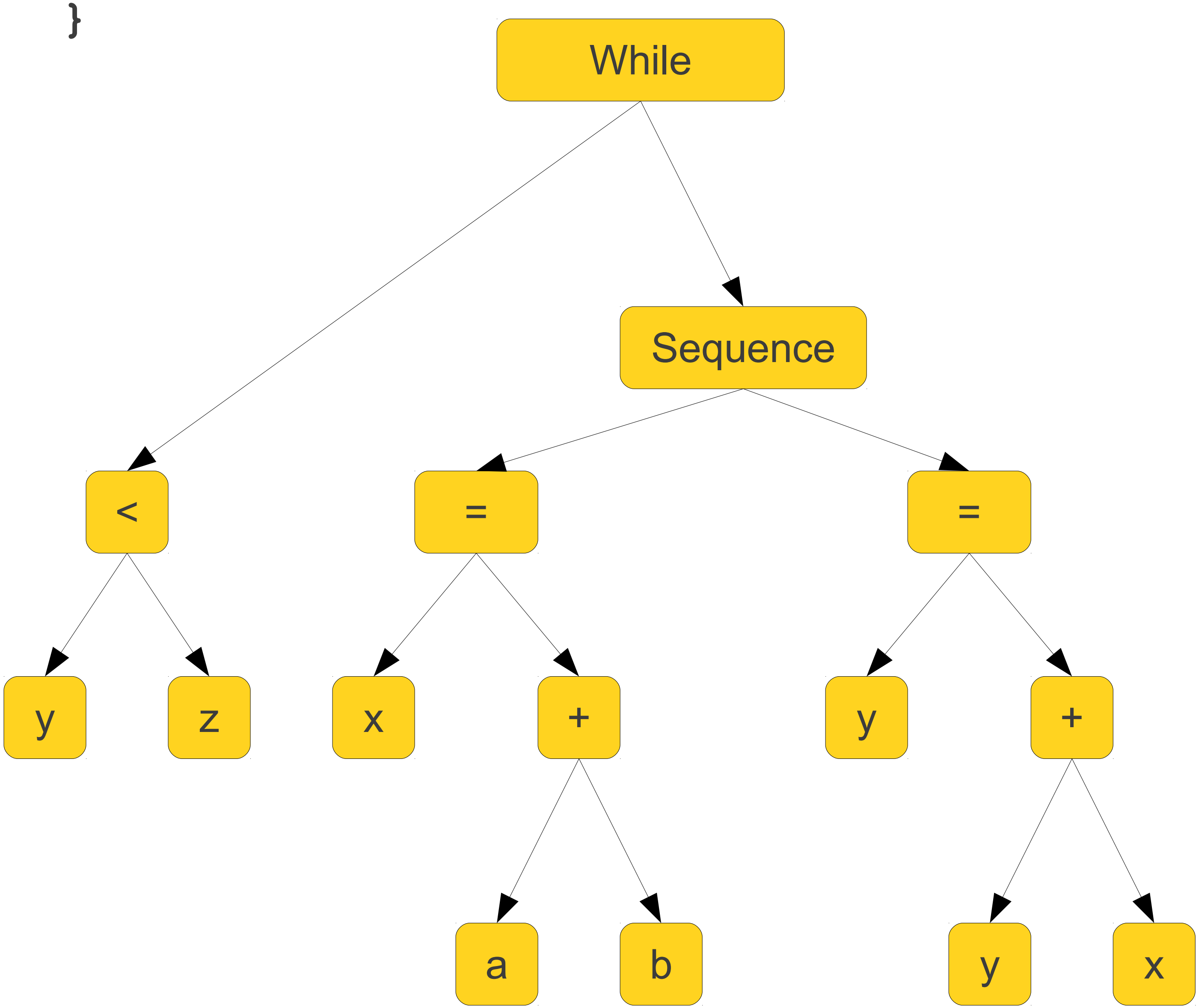
IR Generation

IR Optimization

Code Generation

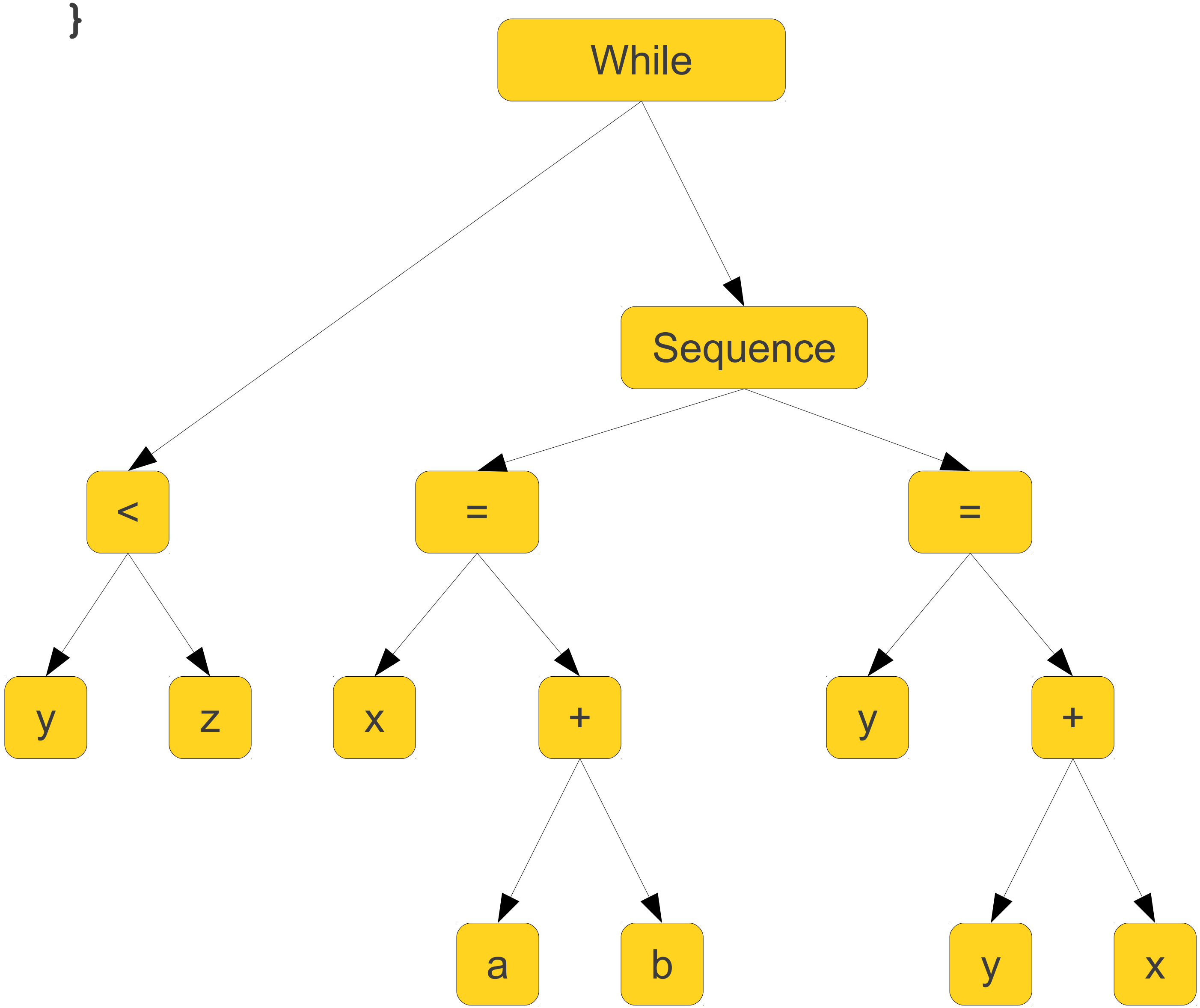
Optimization

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

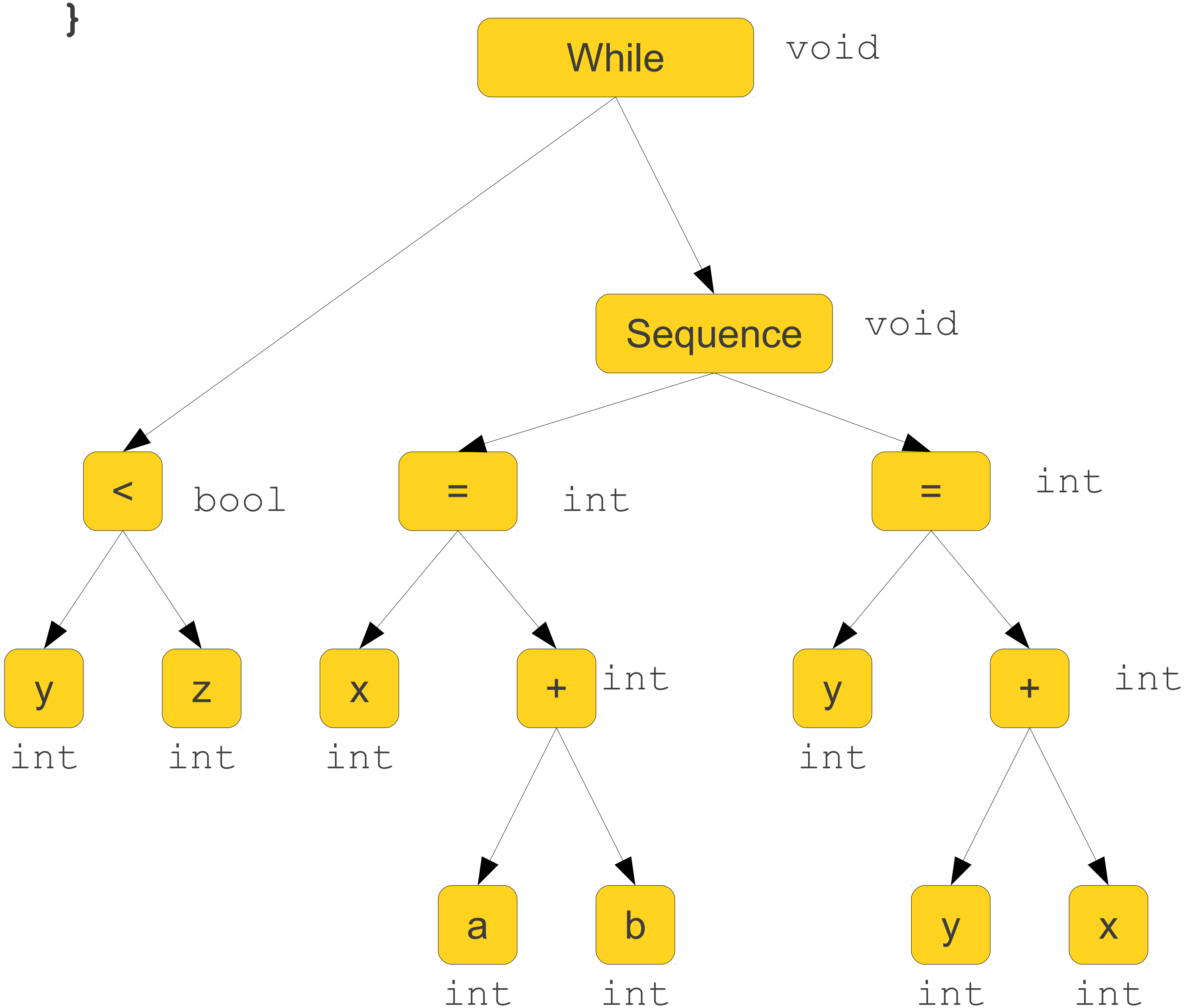



- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- IR Generation
- IR Optimization
- Code Generation
- Optimization

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

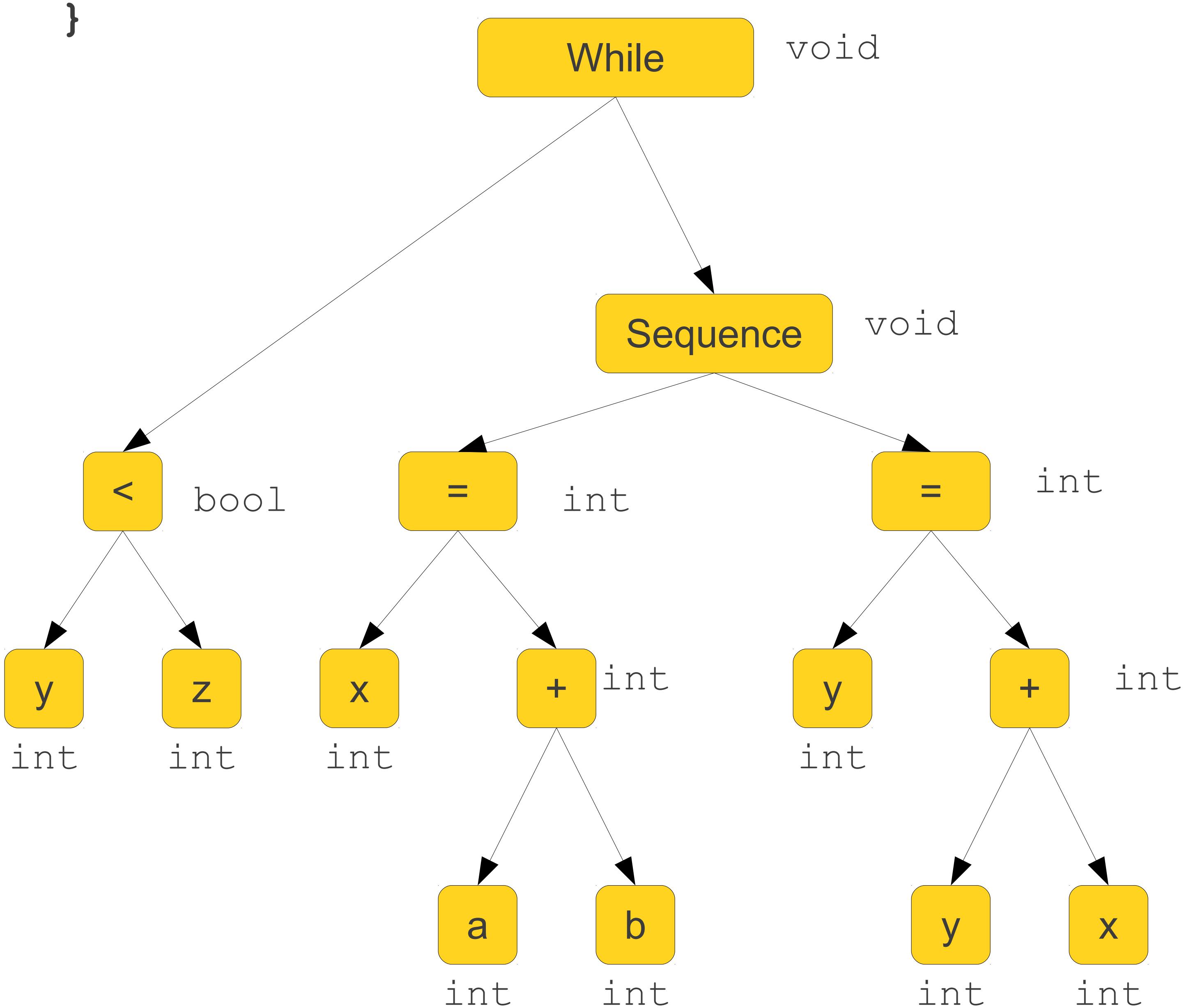



```
while (y < z) {
    int x = a + b;
    y += x;
}
```




- 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
```

| |
|---|
| 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
       slt $6, $1, $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
      slt $6, $1, $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 $1, $5, loop
```

Lexical Analysis

Syntax Analysis

Semantic Analysis

IR Generation

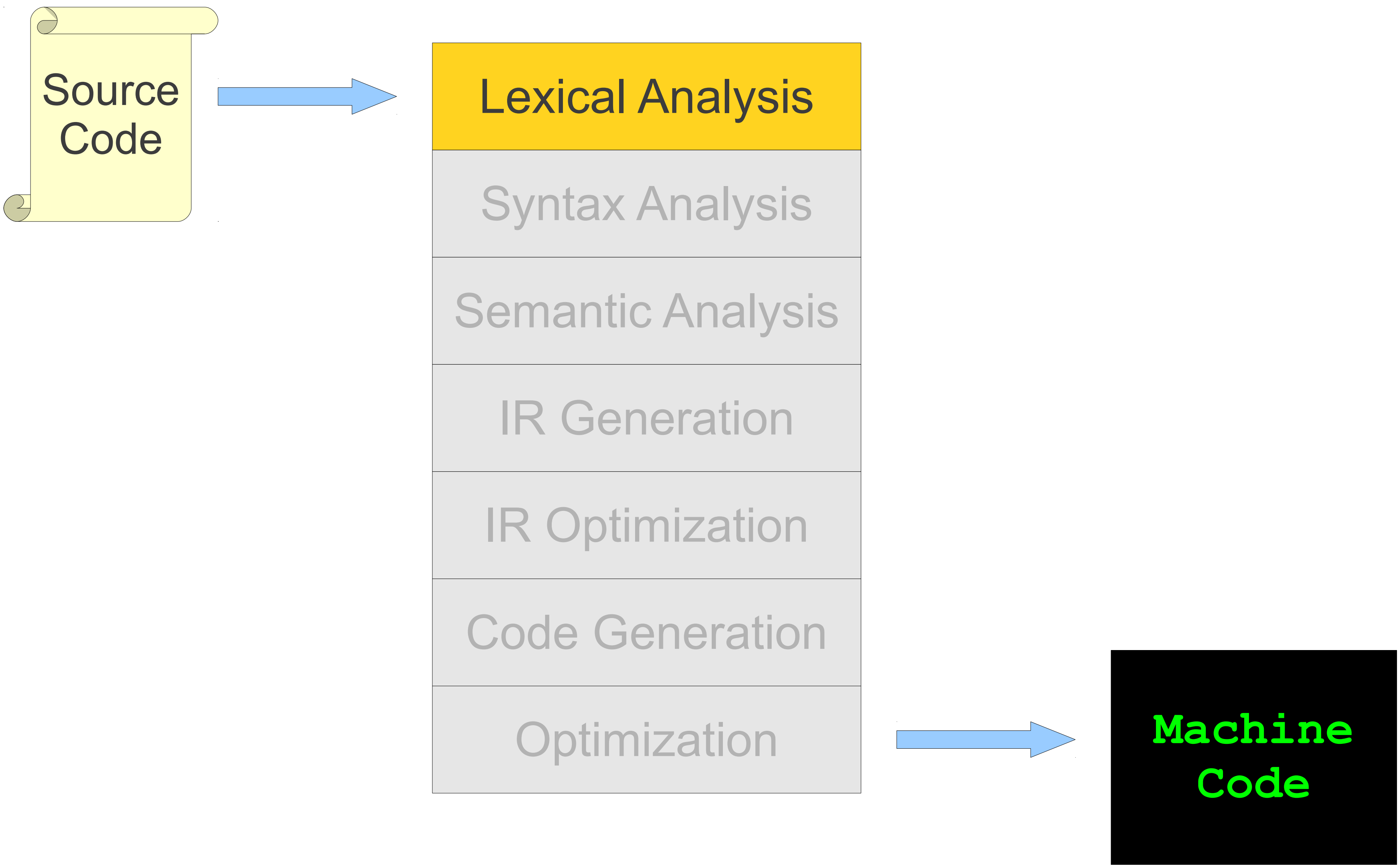
IR Optimization

Code Generation

**Optimization**

# Lexical Analysis

# Where We Are


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

# Scanning a Source File

| | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| w | h | i | l | e | | ( | 1 | 3 | 7 | | < | | i | ) | \n | \t | + | + | i | ; |

# Scanning a Source File

| | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| w | h | i | l | e | | ( | 1 | 3 | 7 | | < | | i | ) | \n | \t | + | + | i | ; |

# Scanning a Source File

| | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| w | h | i | l | e | | ( | 1 | 3 | 7 | | < | | i | ) | \n | \t | + | + | i | ; |

# Scanning a Source File

| | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| w | h | i | l | e | | ( | 1 | 3 | 7 | | < | | i | ) | \n | \t | + | + | i | ; |

# Scanning a Source File

| | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **w** | **h** | **i** | **l** | **e** | | **(** | **1** | **3** | **7** | | **<** | | **i** | **)** | **\n** | **\t** | **+** | **+** | **i** | **;** |

# Scanning a Source File

| | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| w | h | i | l | e | | ( | 1 | 3 | 7 | | < | | i | ) | \n | \t | + | + | i | ; |

# Scanning a Source File

| | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| w | h | i | l | e | | ( | 1 | 3 | 7 | | < | | i | ) | \n | \t | + | + | i | ; |

# Scanning a Source File

| | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| w | h | i | l | e | | ( | 1 | 3 | 7 | | < | | i | ) | \n | \t | + | + | i | ; |

`T_While`

# Scanning a Source File

| | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| w | h | i | l | e | | ( | 1 | 3 | 7 | | < | | i | ) | \n | \t | + | + | i | ; |

The piece of the original program from which we made the token is called a **lexeme**.

`T_While`

This is called a **token**. You can think of it as an enumerated type representing what logical entity we read out of the source code.

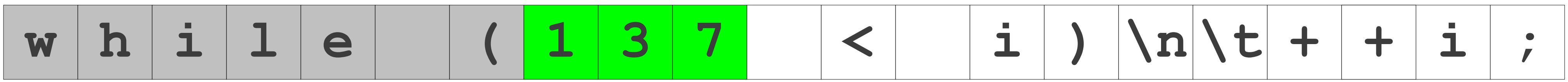
# Scanning a Source File

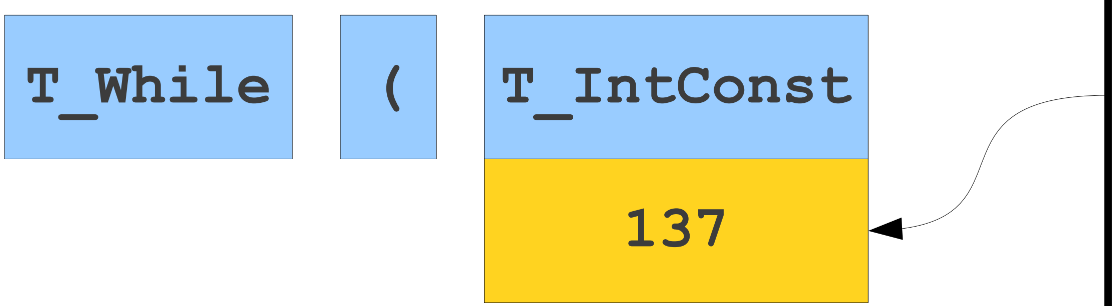
| | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| w | h | i | l | e | | ( | 1 | 3 | 7 | | < | | i | ) | \n | \t | + | + | i | ; |

`T_While`

Sometimes we will discard a lexeme rather than storing it for later use. Here, we ignore whitespace, since it has no bearing on the meaning of the program.

# Scanning a Source File




Some tokens can have **attributes** that store extra information about the token. Here we store which integer is represented.

# Goals of Lexical Analysis

- Convert from physical description of a program into sequence of of **tokens**.
  - Each token represents one logical piece of the source file – a keyword, the name of a variable, etc.
- Each token is associated with a **lexeme**.
  - The actual text of the token: “137,” “int,” etc.
- Each token may have optional **attributes**.
  - Extra information derived from the text – perhaps a numeric value.
- The token sequence will be used in the parser to recover the program structure.

# Choosing Tokens

# What Tokens are Useful Here?

```
for (int k = 0; k < myArray[5]; ++k) {
    cout << k << endl;
}
```

# What Tokens are Useful Here?

```
for (int k = 0; k < myArray[5]; ++k) {
    cout << k << endl;
}
```

```
for             {
int             }
<<              ;
=               <
(               [
)               ]
++

Identifier
IntegerConstant
```

# Choosing Good Tokens

- Very much dependent on the language.
- Typically:
  - Give keywords their own tokens.
  - Give different punctuation symbols their own tokens.
  - Group lexemes representing identifiers, numeric constants, strings, etc. into their own groups.
  - Discard irrelevant information (whitespace, comments)

# Scanning is Hard

- FORTRAN: Whitespace is irrelevant

```
DO 5 I = 1,25
 DO5I  = 1.25
```

Thanks to Prof. Alex Aiken

# Scanning is Hard

- FORTRAN: Whitespace is irrelevant

```
DO 5 I = 1,25
 DO5I  = 1.25
```

- Can be difficult to tell when to partition input.

Thanks to Prof. Alex Aiken

# Challenges in Scanning

- How do we determine which lexemes are associated with each token?
- When there are multiple ways we could scan the input, how do we know which one to pick?
- How do we address these concerns efficiently?

# Associating Lexemes with Tokens

# Lexemes and Tokens

- Tokens give a way to categorize lexemes by what information they provide.
- Some tokens might be associated with only a single lexeme:
  - Tokens for keywords like `if` and `while` probably only match those lexemes exactly.
- Some tokens might be associated with lots of different lexemes:
  - All variable names, all possible numbers, all possible strings, etc.

# Sets of Lexemes

- Idea: Associate a set of lexemes with each token.
- We might associate the "number" token with the set { **`0`**, **`1`**, **`2`**, …, **`10`**, **`11`**, **`12`**, … }
- We might associate the "string" token with the set { **`""`**, **`"a"`**, **`"b"`**, **`"c"`**, … }
- We might associate the token for the keyword **`while`** with the set { **`while`** }.

How do we describe which (potentially infinite) set of lexemes is associated with each token type?

# Formal Languages

- A **formal language** is a set of strings.
- Many infinite languages have finite descriptions:
  - Define the language using an automaton.
  - Define the language using a grammar.
  - Define the language using a regular expression.
- We can use these compact descriptions of the language to define sets of strings.

# Regular Expressions

- **Regular expressions** are a family of descriptions that can be used to capture certain languages (the *regular languages*).
- Often provide a compact and human-readable description of the language.
- Used as the basis for numerous software systems.

# Atomic Regular Expressions

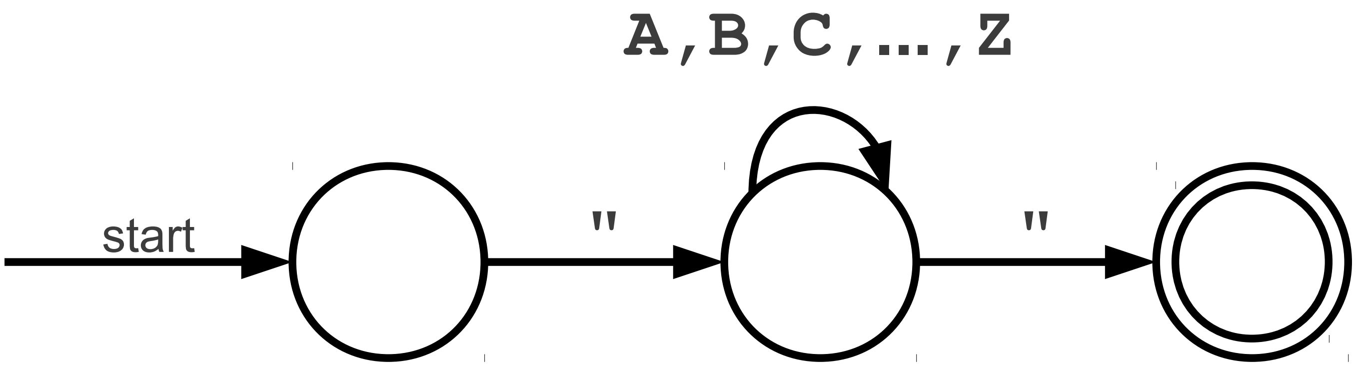
- The regular expressions we will use in this course begin with two simple building blocks.
- The symbol **ε** is a regular expression matches the empty string.
- For any symbol **`a`**, the symbol **`a`** is a regular expression that just matches **`a`**.

# Matching Regular Expressions

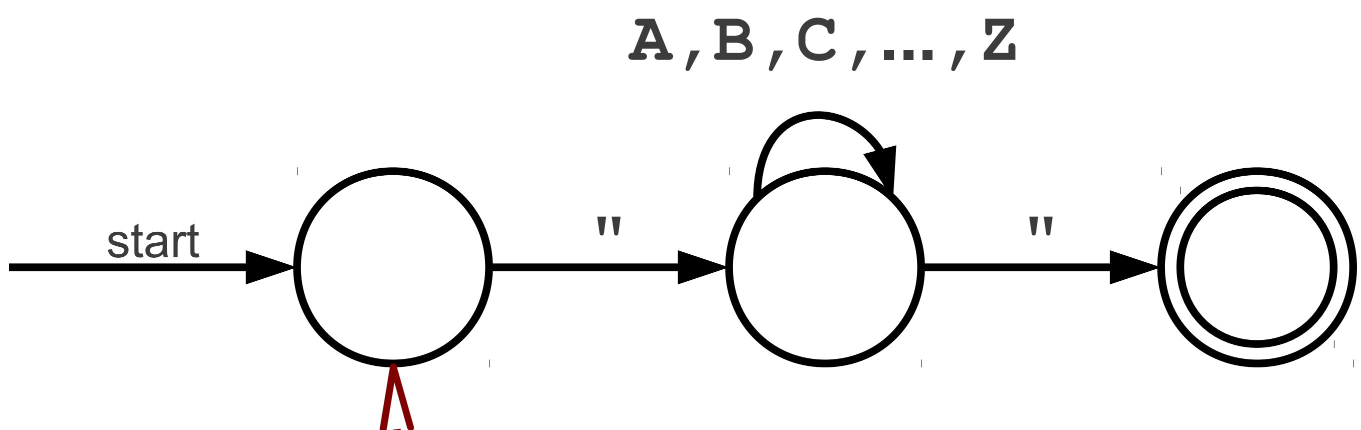
# Implementing Regular Expressions

- Regular expressions can be implemented using **finite automata**.
- There are two main kinds of finite automata:
  - **NFA**s (**nondeterministic** finite automata), which we'll see in a second, and
  - **DFA**s (**deterministic** finite automata), which we'll see later.
- Automata are best explained by example...

# A Simple Automaton

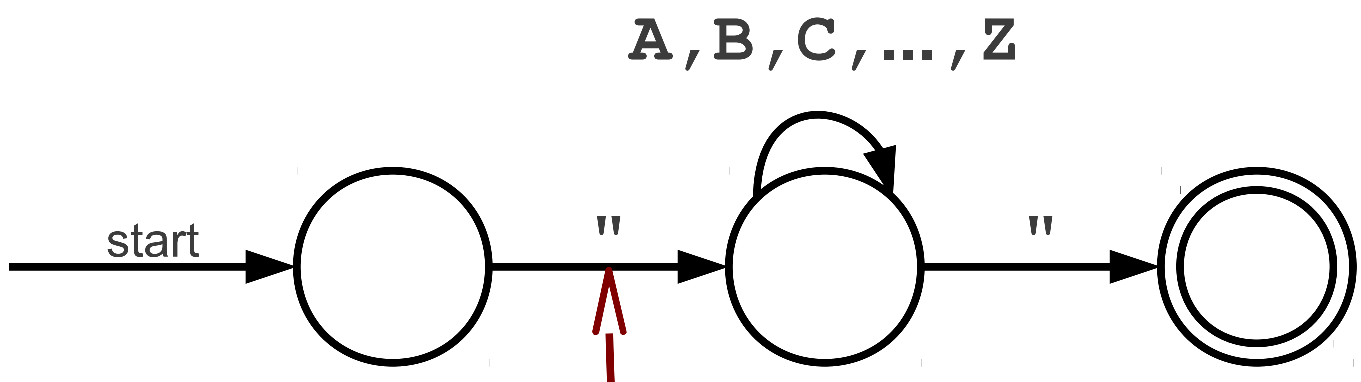


# A Simple Automaton



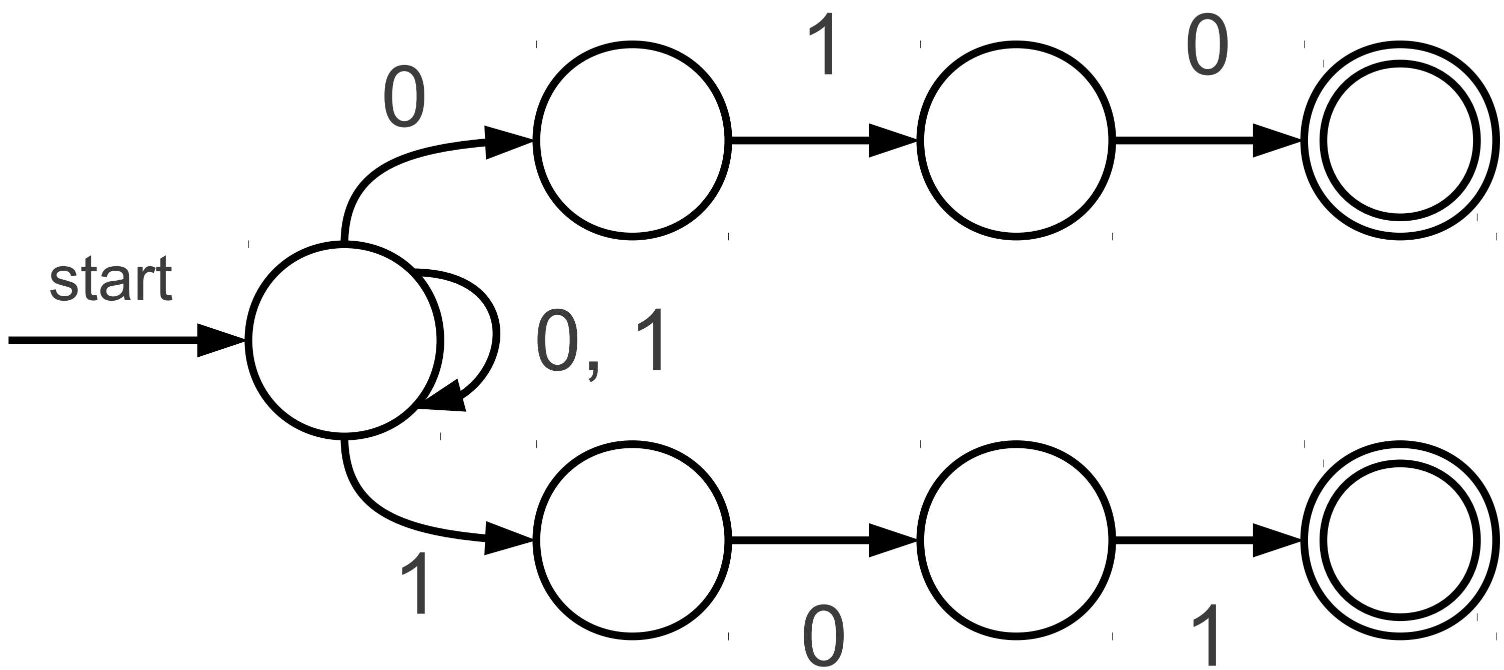

Each circle is a state of the automaton. The automaton's configuration is determined by what state(s) it is in.

# A Simple Automaton

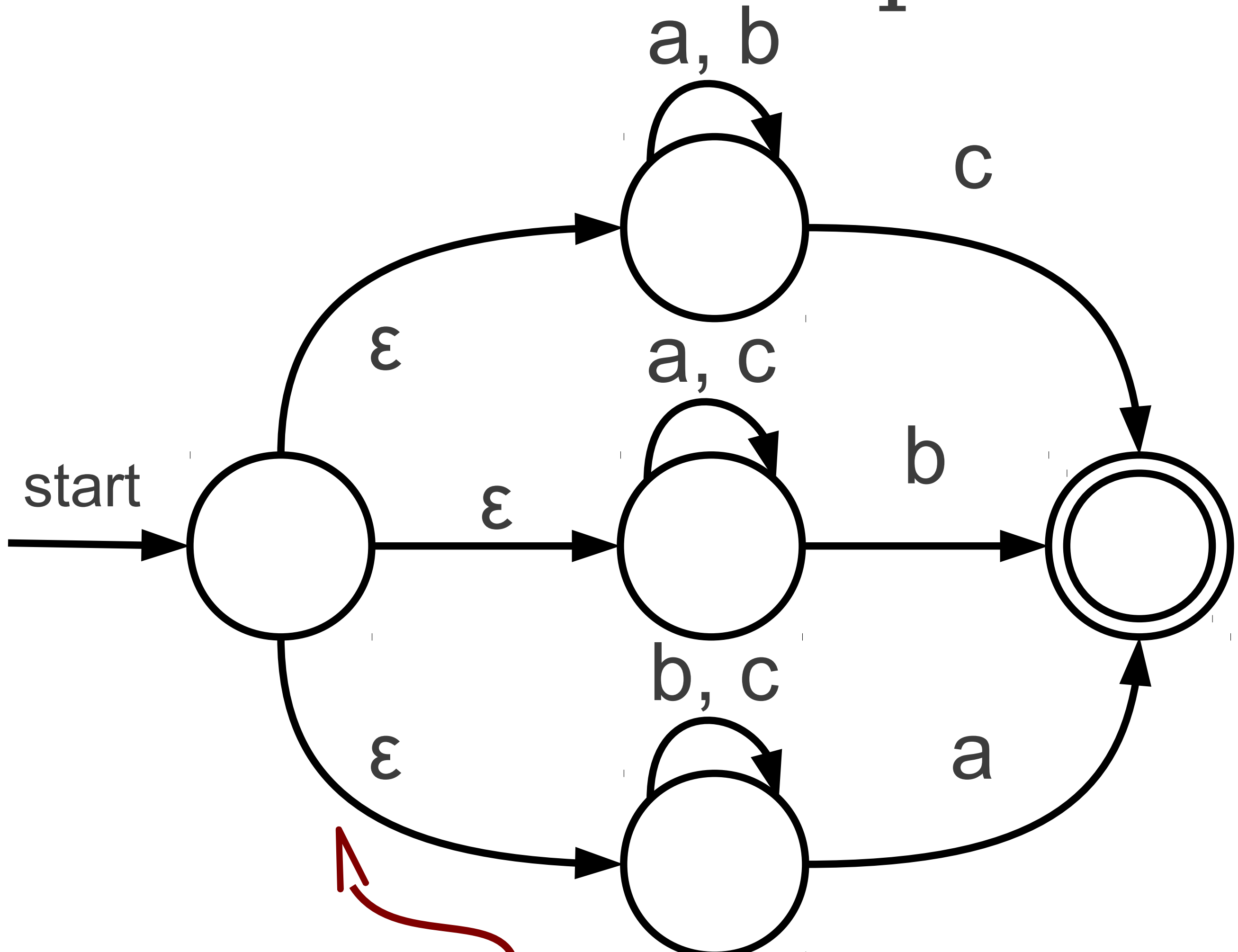



These arrows are called **transitions**. The automaton changes which state(s) it is in by following transitions.

# A More Complex Automaton



# An Even More Complex Automaton



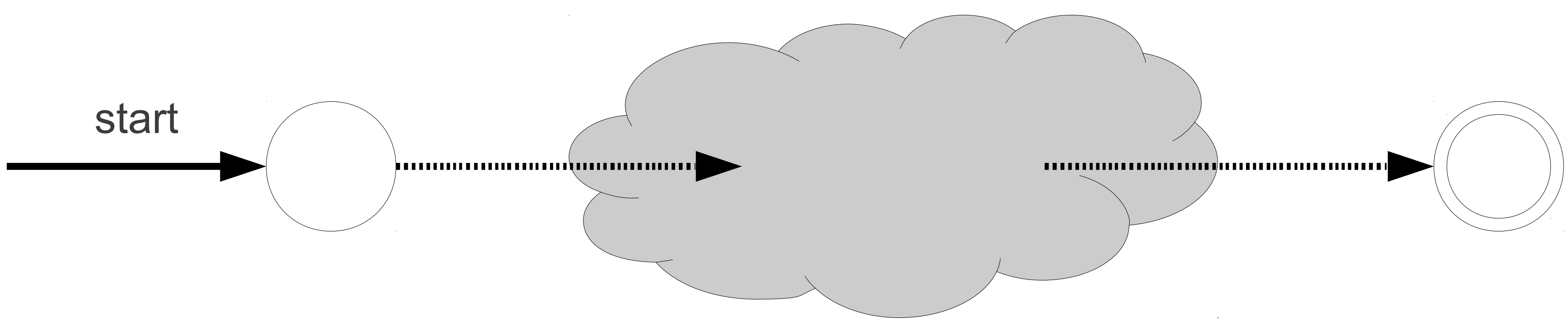

These are called **ε-transitions**. These transitions are followed automatically and without consuming any input.

# Simulating an NFA

- Keep track of a set of states, initially the start state and everything reachable by ε-moves.
- For each character in the input:
  - Maintain a set of next states, initially empty.
  - For each current state:
    - Follow all transitions labeled with the current letter.
    - Add these states to the set of new states.
  - Add every state reachable by an ε-move to the set of next states.
- Complexity: $O(mn^2)$ for strings of length $m$ and automata with $n$ states.

# From Regular Expressions to NFAs

- There is a (beautiful!) procedure from converting a regular expression to an NFA.
- Associate each regular expression with an NFA with the following properties:
  - There is exactly one accepting state.
  - There are no transitions out of the accepting state.
  - There are no transitions into the starting state.
- These restrictions are stronger than necessary, but make the construction easier.



# Overall Result

- Any regular expression of length *n* can be converted into an NFA with O(*n*) states.
- Can determine whether a string of length *m* matches a regular expression of length *n* in time $O(mn^2)$.
- We'll see how to make this O(*m*) later (this is independent of the complexity of the regular expression!)

# Challenges in Scanning

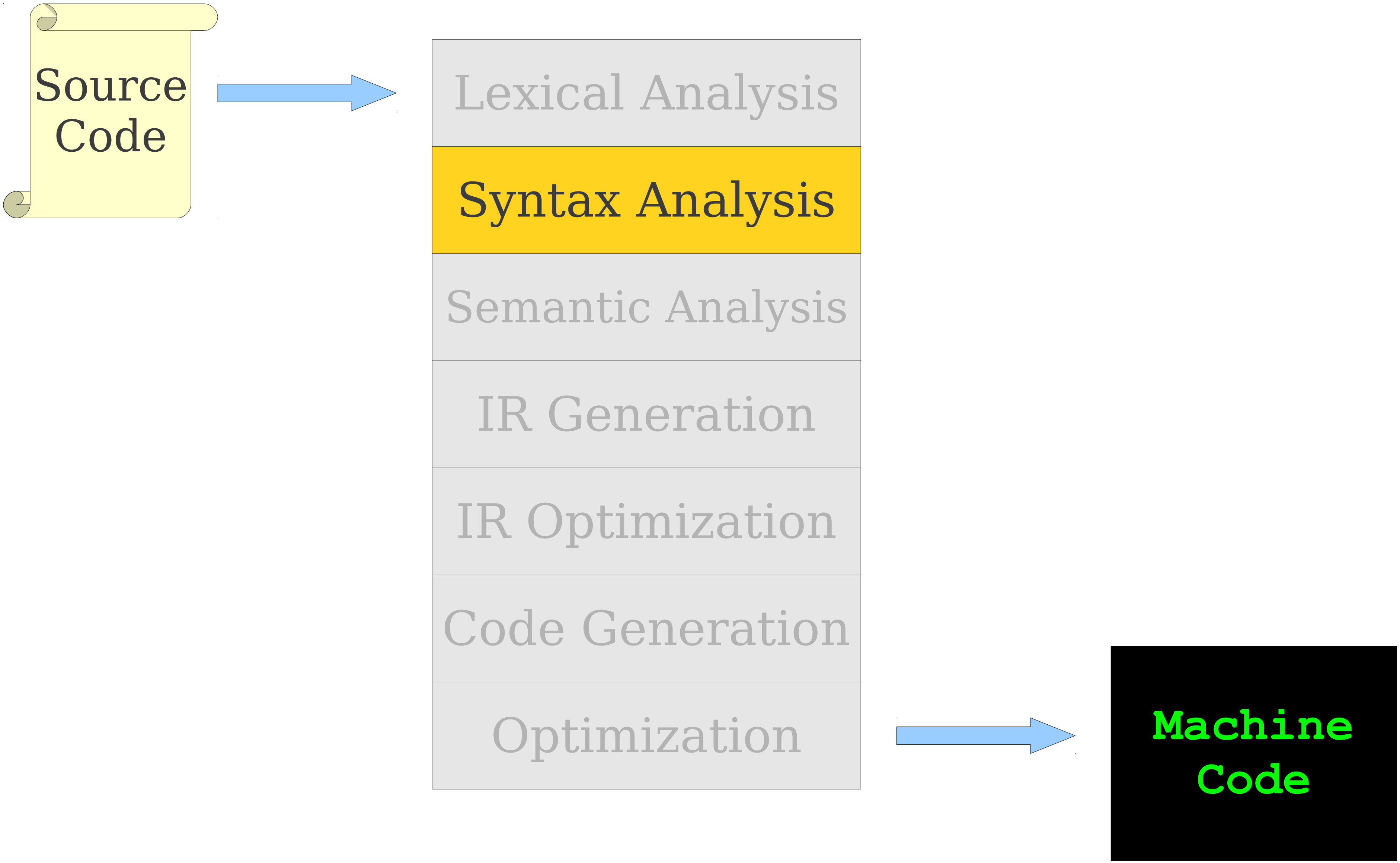
- How do we determine which lexemes are associated with each token?
- When there are multiple ways we could scan the input, how do we know which one to pick?
- How do we address these concerns efficiently?

# Summary

- Lexical analysis splits input text into **tokens** holding a **lexeme** and an **attribute**.
- Lexemes are sets of strings often defined with **regular expressions**.
- Regular expressions can be converted to **NFAs** and from there to **DFAs**.

# Syntax Analysis

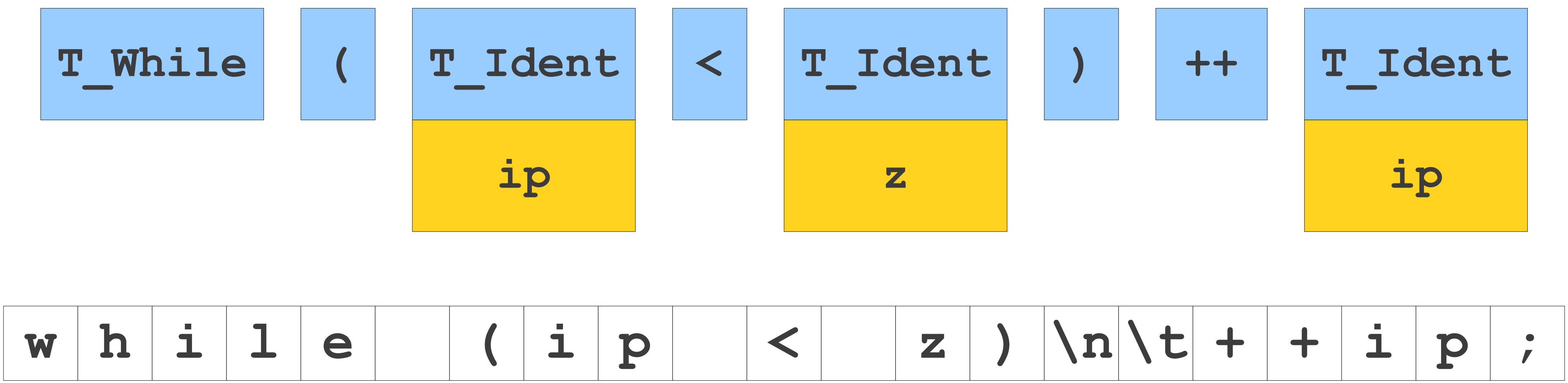
# Where We Are



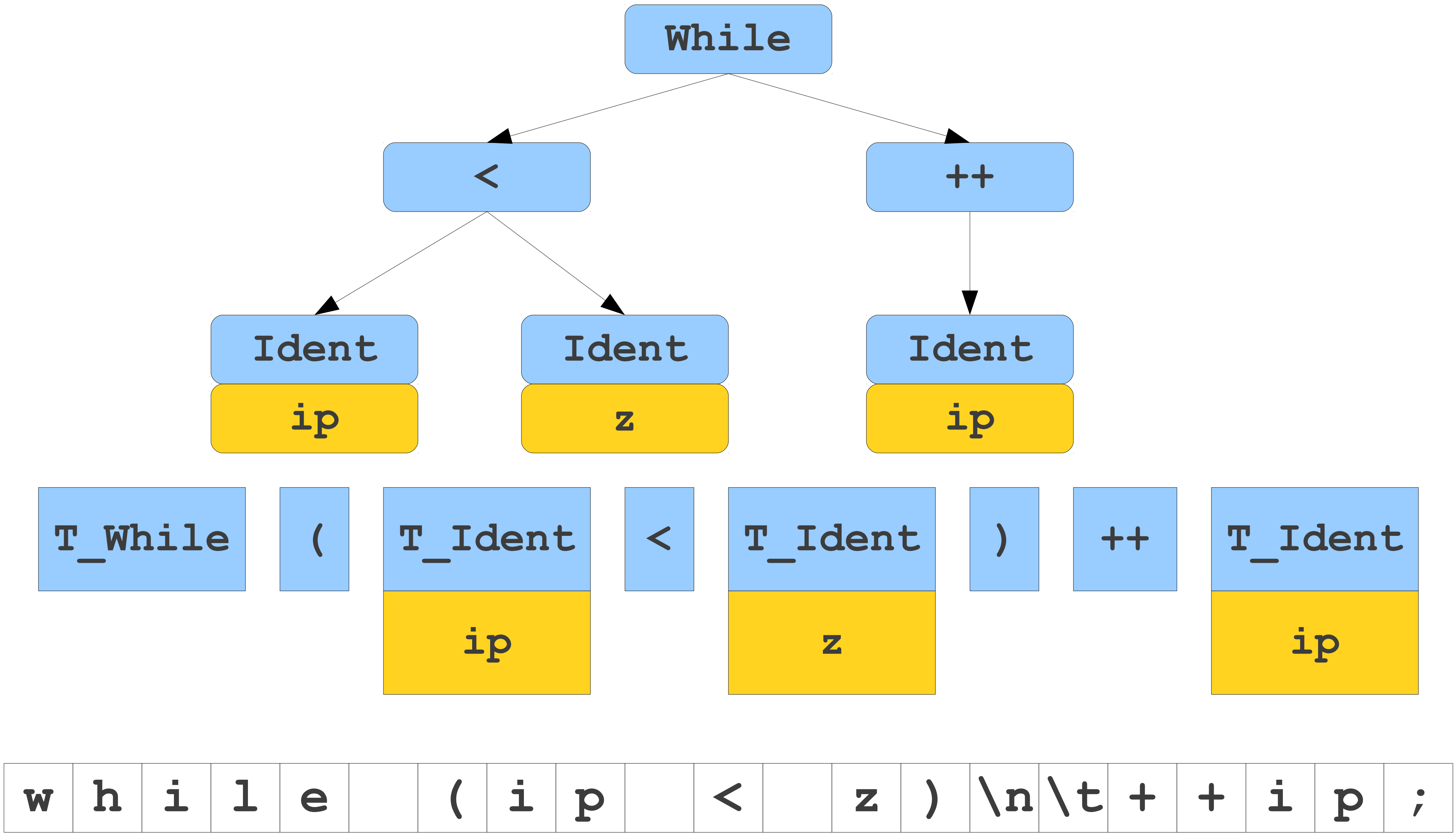
Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

| | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| w | h | i | l | e | | ( | i | p | | < | | z | ) | \n | \t | + | + | i | p | ; |

```
while (ip < z)
    ++ip;
```




```
while (ip < z)
++ip;
```

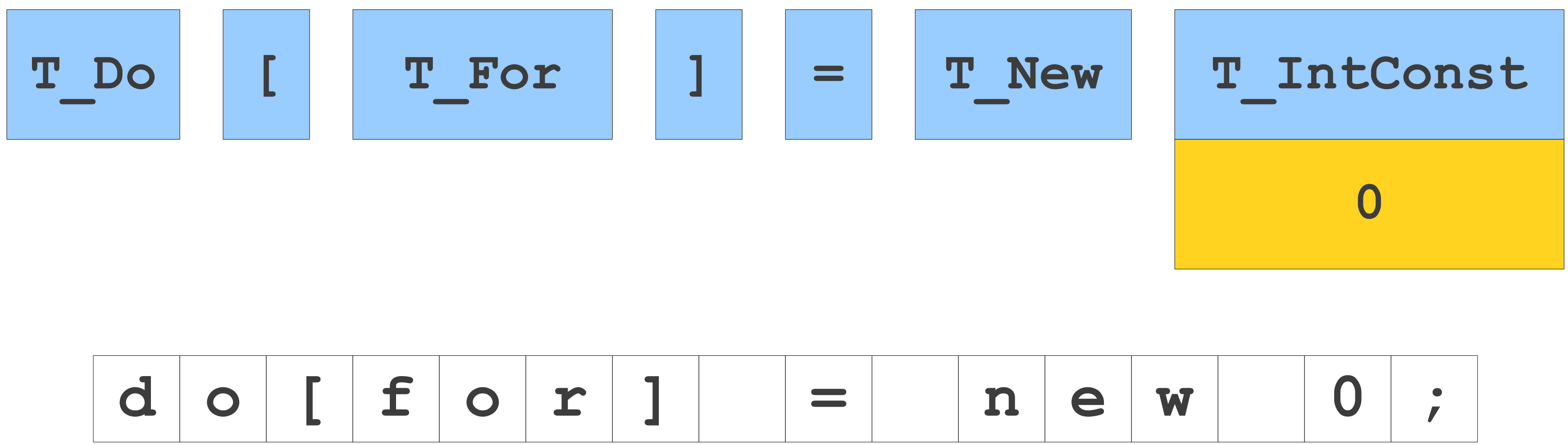


While
<
++
Ident
ip
Ident
z
Ident
ip
T_While
(
T_Ident
ip
<
T_Ident
z
)
++
T_Ident
ip
w h i l e ( i p < z ) \n \t + + i p ;

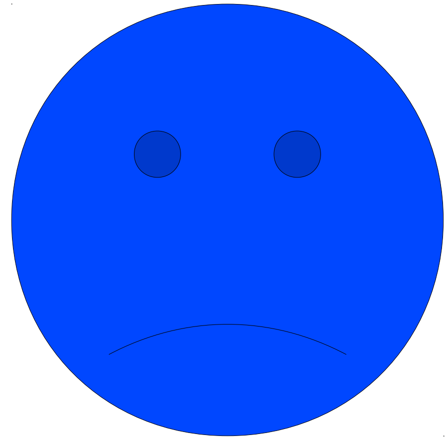

```
while (ip < z)
  ++ip;
```

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| d | o | [ | f | o | r | ] | | = | | n | e | w | | 0 | ; |

```
do[for] = new 0;
```




```
do[for] = new 0;
```



| | | | | | | |
|---|---|---|---|---|---|---|
| T_Do | [ | T_For | ] | = | T_New | T_IntConst |
| | | | | | | 0 |

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| d | o | [ | f | o | r | ] | | = | | n | e | w | | 0 | ; |

```
do[for] = new 0;
```

# What is Syntax Analysis?

- After lexical analysis (scanning), we have a series of tokens.
- In **syntax analysis** (or **parsing**), we want to interpret what those tokens mean.
- Goal: Recover the *structure* described by that series of tokens.
- Goal: Report *errors* if those tokens do not properly encode a structure.

# Formal Languages

- An **alphabet** is a set Σ of symbols that act as letters.
- A **language** over Σ is a set of strings made from symbols in Σ.
- When scanning, our alphabet was ASCII or Unicode characters. We produced tokens.
- When parsing, our alphabet is the set of tokens produced by the scanner.

# The Limits of Regular Languages

- When scanning, we used regular expressions to define each token.
- Unfortunately, regular expressions are (usually) too weak to define programming languages.
  - Cannot define a regular expression matching all expressions with properly balanced parentheses.
  - Cannot define a regular expression matching all functions with properly nested block structure.
- We need a more powerful formalism.

# Context-Free Grammars

- A **context-free grammar** (or **CFG**) is a formalism for defining languages.
- Can define the **context-free languages**, a strict superset of the the regular languages.
- CFGs are best explained by example...

# Arithmetic Expressions

- Suppose we want to describe all legal arithmetic expressions using addition, subtraction, multiplication, and division.
- Here is one possible CFG:

**E** → `int`

**E** → **E Op E**

**E** → `(`**E**`)`

**Op** → `+`

**Op** → `-`

**Op** → `*`

**Op** → `/`

**E**

⇒ **E Op E**

⇒ **E Op** `(`**E**`)`

⇒ **E Op** `(`**E Op E**`)`

⇒ **E** `*` `(`**E Op E**`)`

⇒ `int * (`**E Op E**`)`

⇒ `int * (int` **Op E**`)`

⇒ `int * (int` **Op** `int)`

⇒ `int * (int + int)`

# Arithmetic Expressions

- Suppose we want to describe all legal arithmetic expressions using addition, subtraction, multiplication, and division.
- Here is one possible CFG:

**E** → `int`

**E** → **E Op E**

**E** → `(`**E**`)`

**Op** → `+`

**Op** → `-`

**Op** → `*`

**Op** → `/`

  **E**
⇒ **E Op E**
⇒ **E Op** `int`
⇒ `int` **Op** `int`
⇒ `int / int`

# Context-Free Grammars

- Formally, a context-free grammar is a collection of four objects:
  - A set of **nonterminal symbols** (or **variables**),
  - A set of **terminal symbols**,
  - A set of **production rules** saying how each nonterminal can be converted by a string of terminals and nonterminals, and
  - A **start symbol** that begins the derivation.

**E** → `int`

**E** → **E Op E**

**E** → `(`**E**`)`

**Op** → `+`

**Op** → `-`

**Op** → `*`

**Op** → `/`

# A Notational Shorthand

**E** → `int` | **E Op E** | `(`**E**`)`

**Op** → `+` | `-` | `*` | `/`

# Some CFG Notation

- Capital letters at the beginning of the alphabet will represent nonterminals.
  - i.e. **A**, **B**, **C**, **D**
- Lowercase letters at the end of the alphabet will represent terminals.
  - i.e. **`t`**, **`u`**, **`v`**, **`w`**
- Lowercase Greek letters will represent arbitrary strings of terminals and nonterminals.
  - i.e. $\boldsymbol{\alpha}$, $\boldsymbol{\gamma}$, $\boldsymbol{\omega}$

# Examples

- We might write an arbitrary production as

$$\mathbf{A} \to \boldsymbol{\omega}$$

- We might write a string of a nonterminal followed by a terminal as

$$\mathbf{A}\mathtt{t}$$

- We might write an arbitrary production containing a nonterminal followed by a terminal as

$$\mathbf{B} \to \boldsymbol{\alpha}\mathbf{A}\mathtt{t}\boldsymbol{\omega}$$

# Derivations

**E**
⇒ **E Op E**
⇒ **E Op (E)**
⇒ **E Op (E Op E)**
⇒ **E * (E Op E)**
⇒ **int * (E Op E)**
⇒ **int * (int Op E)**
⇒ **int * (int Op int)**
⇒ **int * (int + int)**

- This sequence of steps is called a **derivation**.
- A string $\boldsymbol{\alpha}\mathbf{A}\boldsymbol{\omega}$ **yields** string $\boldsymbol{\alpha\gamma\omega}$ iff $\mathbf{A} \to \boldsymbol{\gamma}$ is a production.
- If $\boldsymbol{\alpha}$ yields $\boldsymbol{\beta}$, we write $\boldsymbol{\alpha} \Rightarrow \boldsymbol{\beta}$.
- We say that $\boldsymbol{\alpha}$ **derives** $\boldsymbol{\beta}$ iff there is a sequence of strings where

$$\boldsymbol{\alpha} \Rightarrow \boldsymbol{\alpha}_1 \Rightarrow \boldsymbol{\alpha}_2 \Rightarrow \ldots \Rightarrow \boldsymbol{\beta}$$

- If $\boldsymbol{\alpha}$ derives $\boldsymbol{\beta}$, we write $\boldsymbol{\alpha} \Rightarrow^* \boldsymbol{\beta}$.

# Leftmost Derivations

```
BLOCK  →  STMT
       |  { STMTS }

STMTS  →  ε
       |  STMT STMTS

STMT   →  EXPR;
       |  if (EXPR) BLOCK
       |  while (EXPR) BLOCK
       |  do BLOCK while (EXPR);
       |  BLOCK
       |  …

EXPR   →  identifier
       |  constant
       |  EXPR + EXPR
       |  EXPR – EXPR
       |  EXPR * EXPR
       |  EXPR = EXPR
       |  …
```

```
   STMTS
⇒  STMT STMTS
⇒  EXPR; STMTS
⇒  EXPR = EXPR; STMTS
⇒  id = EXPR; STMTS
⇒  id = EXPR + EXPR; STMTS
⇒  id = id + EXPR; STMTS
⇒  id = id + constant; STMTS
⇒  id = id + constant;
```

# Leftmost Derivations

- A **leftmost derivation** is a derivation in which each step expands the leftmost nonterminal.
- A **rightmost derivation** is a derivation in which each step expands the rightmost nonterminal.
- These will be of great importance when we talk about parsing.

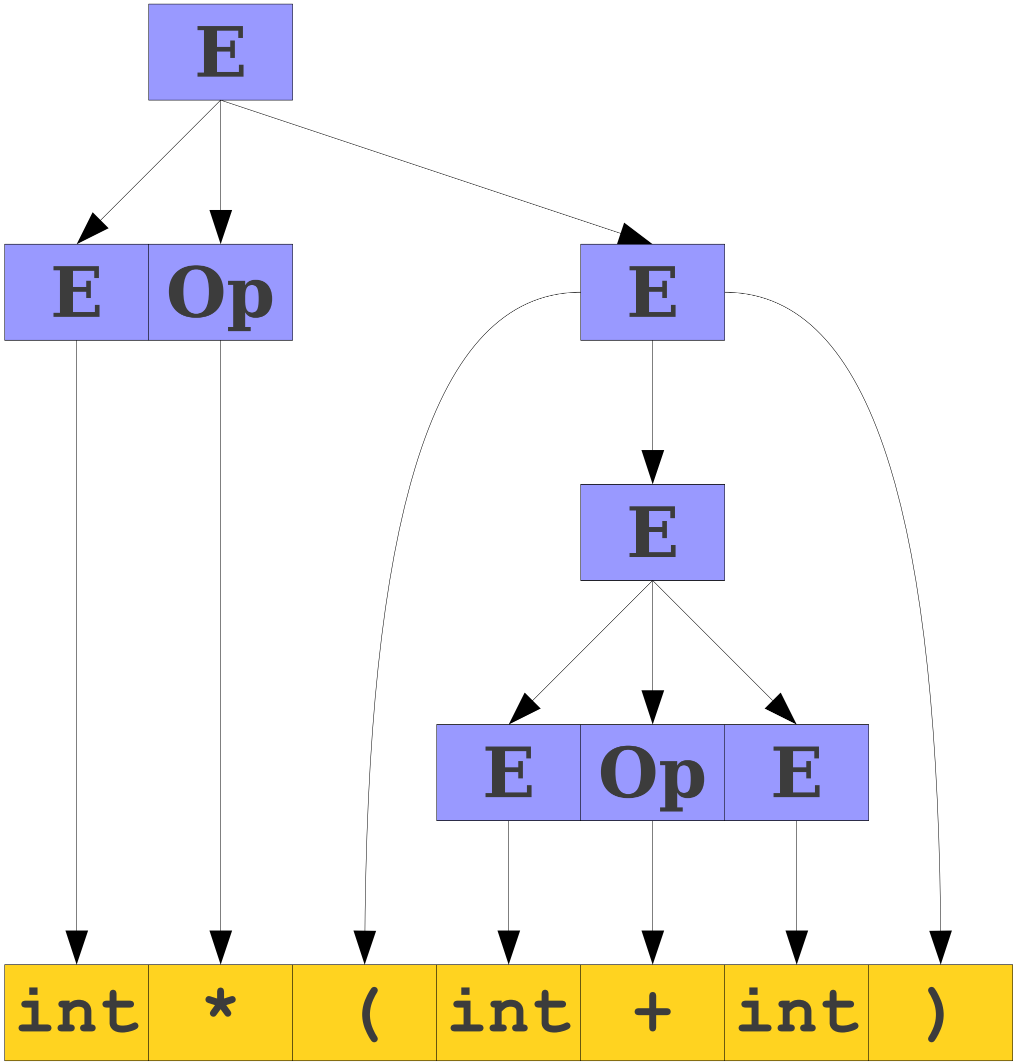
# Related Derivations

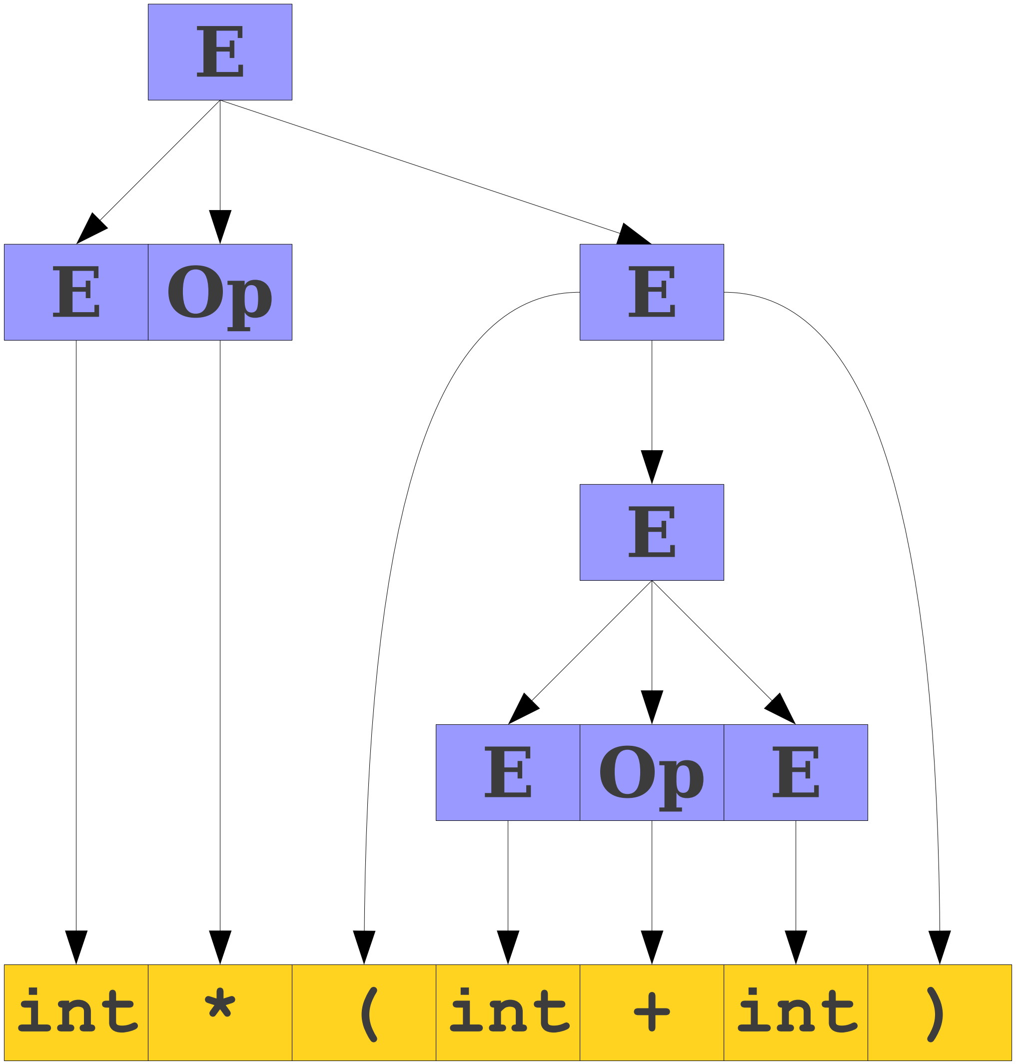
| | |
|---|---|
| **E** | **E** |
| ⇒ **E Op E** | ⇒ **E Op E** |
| ⇒ `int` **Op E** | ⇒ **E Op** `(`**E**`)` |
| ⇒ `int *` **E** | ⇒ **E Op** `(`**E Op E**`)` |
| ⇒ `int * (`**E**`)` | ⇒ **E Op** `(`**E Op** `int)` |
| ⇒ `int * (`**E Op E**`)` | ⇒ **E Op** `(`**E** `+ int)` |
| ⇒ `int * (int` **Op E**`)` | ⇒ **E Op** `(int + int)` |
| ⇒ `int * (int +` **E**`)` | ⇒ **E** `* (int + int)` |
| ⇒ `int * (int + int)` | ⇒ `int * (int + int)` |

# Derivations Revisited

- A derivation encodes two pieces of information:
  - What productions were applied produce the resulting string from the start symbol?
  - In what order were they applied?
- Multiple derivations might use the same productions, but apply them in a different order.

# For Comparison



# Parse Trees

- A **parse tree** is a tree encoding the steps in a derivation.
- Internal nodes represent nonterminal symbols used in the production.
- Inorder walk of the leaves contains the generated string.
- Encodes what productions are used, not the order in which those productions are applied.

# The Goal of Parsing

- Goal of syntax analysis: Recover the **structure** described by a series of tokens.
- If language is described as a CFG, goal is to recover a parse tree for the the input string.
  - Usually we do some simplifications on the tree; more on that later.

# Challenges in Parsing

# Context-Free Grammars

- A regular expression can be
  - Any letter
  - ε
  - The concatenation of regular expressions.
  - The union of regular expressions.
  - The Kleene closure of a regular expression.
  - A parenthesized regular expression.

# Context-Free Grammars

- This gives us the following CFG:

  **R** → `a` | `b` | `c` | …

  **R** → "`ε`"

  **R** → **RR**

  **R** → **R** "`|`" **R**

  **R** → **R**`*`

  **R** → `(`**R**`)`

# Abstract Syntax Trees (ASTs)

- A parse tree is a **concrete syntax tree**; it shows exactly how the text was derived.
- A more useful structure is an **abstract syntax tree**, which retains only the essential structure of the input.

# Summary

- Syntax analysis (**parsing**) extracts the structure from the tokens produced by the scanner.
- Languages are usually specified by **context-free grammars** (**CFG**s).
- A **parse tree** shows how a string can be **derived** from a grammar.
- A grammar is **ambiguous** if it can derive the same string multiple ways.
- There is no algorithm for eliminating ambiguity; it must be done by hand.
- **Abstract syntax trees** (**AST**s) contain an abstract representation of a program's syntax.

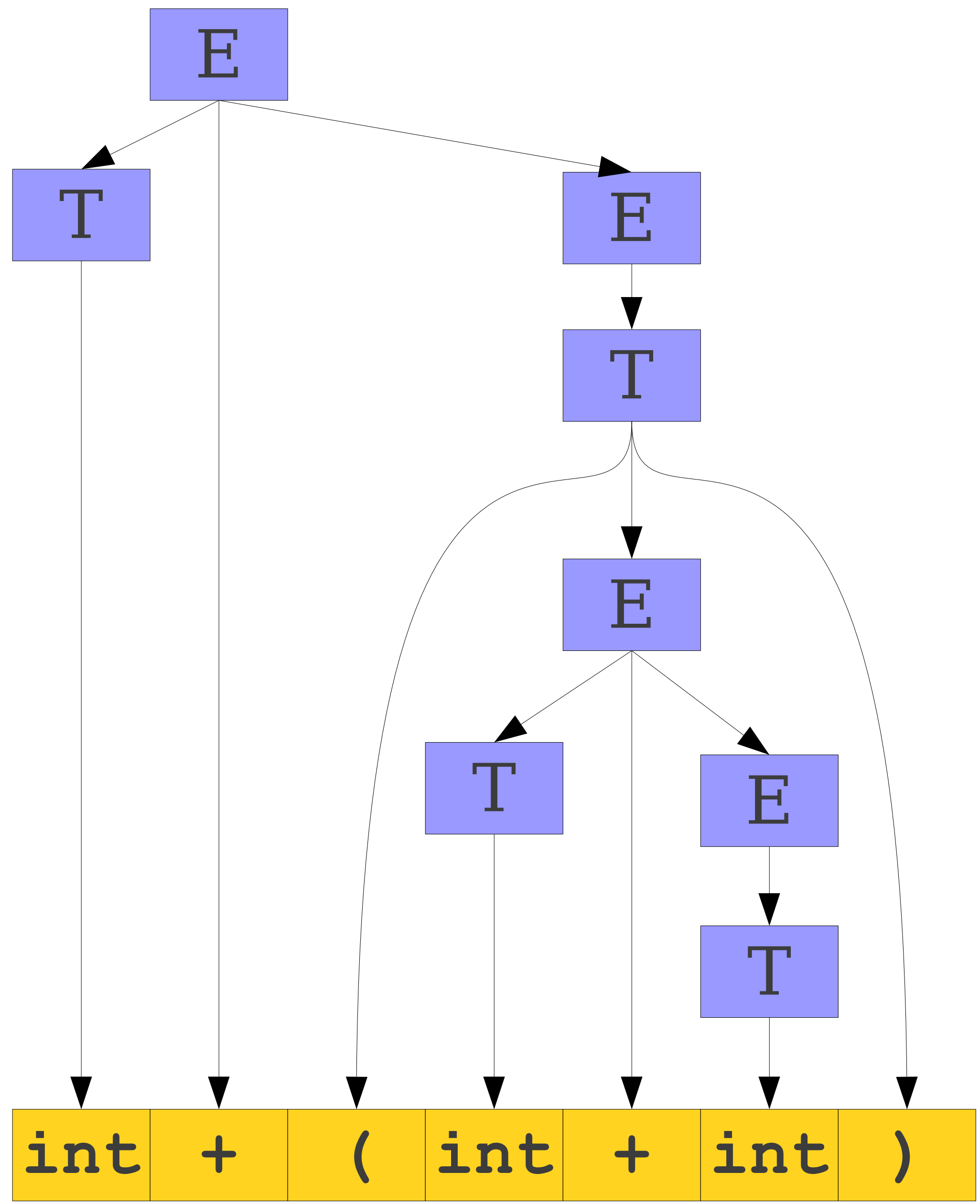
# Top-Down Parsing

# Different Types of Parsing

- **Top-Down Parsing** (Today)
  - Beginning with the start symbol, try to guess the productions to apply to end up at the user's program.
- **Bottom-Up Parsing**
  - Beginning with the user's program, try to apply productions in reverse to convert the program back into the start symbol.

# Top-Down Parsing

$\mathbf{E} \rightarrow \mathbf{T}$
$\mathbf{E} \rightarrow \mathbf{T} + \mathbf{E}$
$\mathbf{T} \rightarrow$ `int`
$\mathbf{T} \rightarrow (\mathbf{E})$



# Challenges in Top-Down Parsing

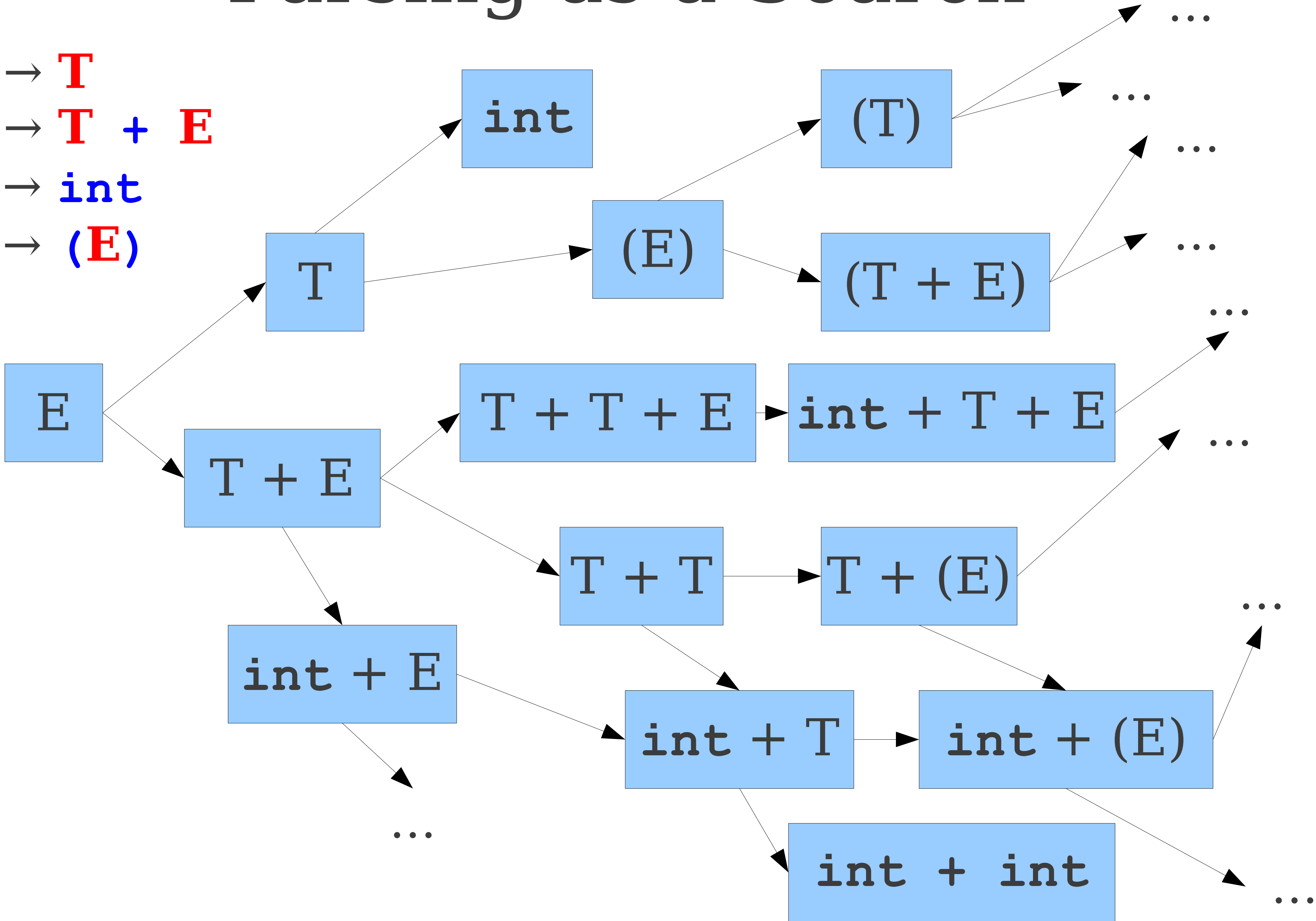
- Top-down parsing begins with virtually no information.
  - Begins with just the start symbol, which matches *every* program.
- How can we know which productions to apply?
- In general, we can't.
  - There are some grammars for which the best we can do is guess and backtrack if we're wrong.
  - If we have to guess, how do we do it?

# Parsing as a Search

- An idea: **treat parsing as a graph search**.
- Each node is a **sentential form** (a string of terminals and nonterminals derivable from the start symbol).
- There is an edge from node $\boldsymbol{\alpha}$ to node $\boldsymbol{\beta}$ iff $\boldsymbol{\alpha} \Rightarrow \boldsymbol{\beta}$.

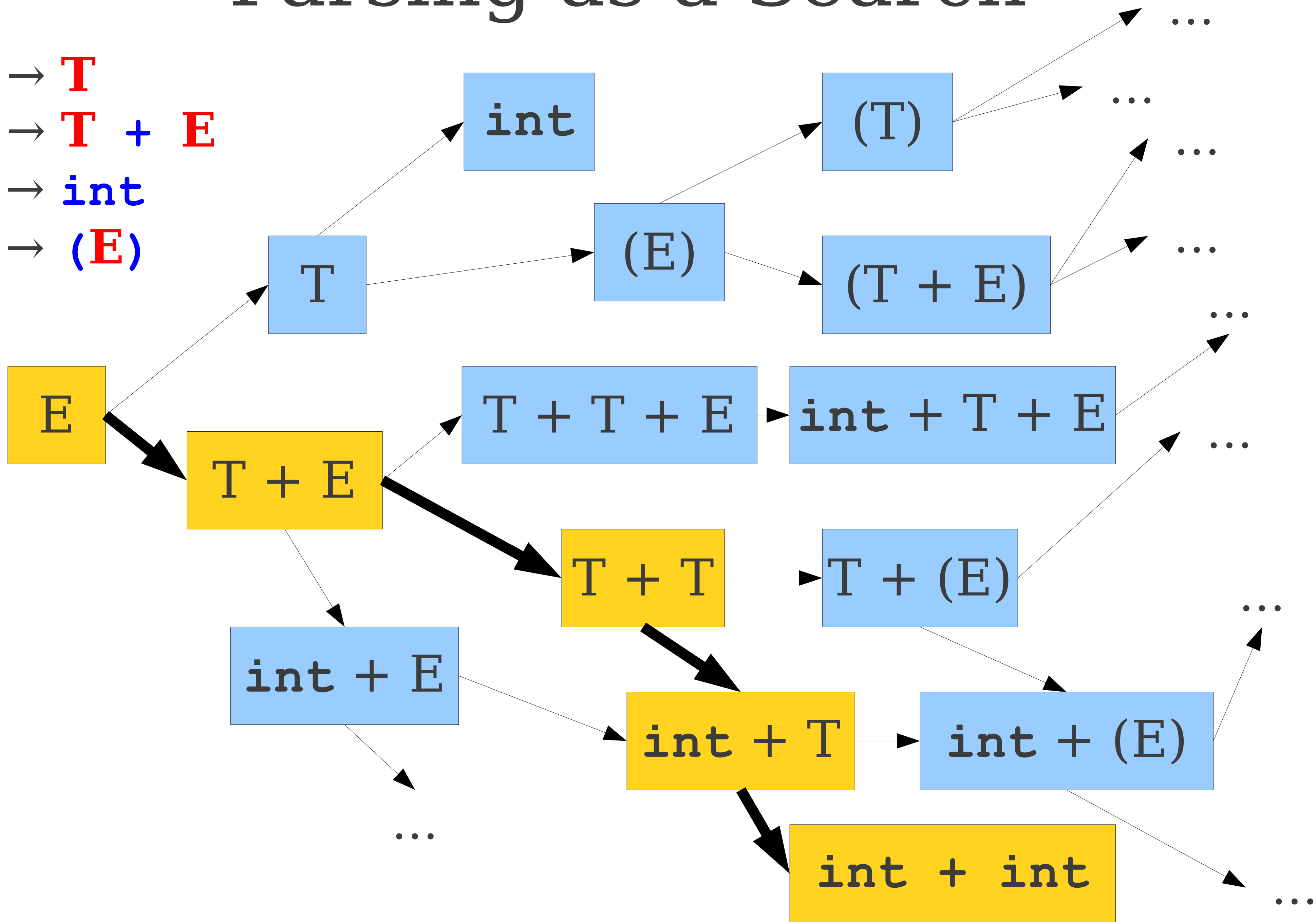
# Parsing as a Search

**E** → **T**
**E** → **T** + **E**
**T** → `int`
**T** → (**E**)



# Parsing as a Search

E → T
E → T + E
T → int
T → (E)



# BFS is Slow

- Enormous time and memory usage:
  - Lots of **wasted effort**:
    - Generates a lot of sentential forms that couldn't possibly match.
    - But in general, extremely hard to tell whether a sentential form can match – that's the job of parsing!
  - High **branching factor**:
    - Each sentential form can expand in (potentially) many ways for each nonterminal it contains.

# Leftmost Derivations

- Recall: A **leftmost derivation** is one where we always expand the leftmost symbol first.
- Updated algorithm:
  - Do a breadth-first search, **only considering leftmost derivations**.
    - Dramatically drops branching factor.
    - Increases likelihood that we get a prefix of nonterminals.
  - Prune sentential forms that can't possibly match.
    - Avoids wasted effort.

# Leftmost BFS

- Substantial improvement over naïve algorithm.
- Will always find a valid parse of a program if one exists.
- Can easily be modified to find if a program can't be parsed.
- But, there are still problems.

# Leftmost DFS

- Idea: Use **depth-first** search.
- Advantages:
  - Lower memory usage: Only considers one branch at a time.
  - High performance: On many grammars, runs very quickly.
  - Easy to implement: Can be written as a set of mutually recursive functions.

# Left Recursion

- A nonterminal **A** is said to be **left-recursive** iff

$$\mathbf{A} \Rightarrow^* \mathbf{A}\boldsymbol{\omega}$$

  for some string $\boldsymbol{\omega}$.

- Leftmost DFS may fail on left-recursive grammars.
- Fortunately, in many cases it is possible to eliminate left recursion.

# Summary of Leftmost BFS/DFS

- Leftmost BFS works on all grammars.
- Worst-case runtime is exponential.
- Worst-case memory usage is exponential.
- Rarely used in practice.

- Leftmost DFS works on grammars without left recursion.
- Worst-case runtime is exponential.
- Worst-case memory usage is linear.
- Often used in a limited form as **recursive descent**.

# Predictive Parsing

# Predictive Parsing

- The leftmost DFS/BFS algorithms are **backtracking** algorithms.
  - Guess which production to use, then back up if it doesn't work.
  - Try to match a prefix by sheer dumb luck.
- There is another class of parsing algorithms called **predictive** algorithms.
  - Based on remaining input, predict (*without backtracking*) which production to use.

# Tradeoffs in Prediction

- Predictive parsers are *fast*.
  - Many predictive algorithms can be made to run in linear time.
  - Often can be table-driven for extra performance.
- Predictive parsers are *weak*.
  - Not all grammars can be accepted by predictive parsers.
- Trade *expressiveness* for *speed*.

# Exploiting Lookahead

- Given just the start symbol, how do you know which productions to use to get to the input program?
- Idea: Use **lookahead tokens**.
- When trying to decide which production to use, look at some number of tokens of the input to help make the decision.

# Implementing Predictive Parsing

- Predictive parsing is only possible if we can predict which production to use given some number of lookahead tokens.
- Increasing the number of lookahead tokens increases the number of grammars we can parse, but complicates the parser.
- Decreasing the number of lookahead tokens decreases the number of grammars we can parse, but simplifies the parser.

# A Simple Predictive Parser: **LL(1)**

- Top-down, predictive parsing:
  - **L**: Left-to-right scan of the tokens
  - **L**: Leftmost derivation.
  - **(1)**: One token of lookahead
- Construct a leftmost derivation for the sequence of tokens.
- When expanding a nonterminal, we predict the production to use by looking at the next token of the input. **The decision is forced.**

# LL(1) Parse Tables

# LL(1) Parse Tables

**E** → `int`

**E** → `(`**E Op E**`)`

**Op** → `+`

**Op** → `*`

# LL(1) Parse Tables

**E** → `int`
**E** → `(`**E Op E**`)`
**Op** → `+`
**Op** → `*`

| | `int` | `(` | `)` | `+` | `*` |
|---|---|---|---|---|---|
| E | `int` | (E Op E) | | | |
| Op | | | | `+` | `*` |

# LL(1) Parsing

`(int + (int * int))`

(1) **E** → `int`
(2) **E** → `(`**E Op E**`)`
(3) **Op** → `+`
(4) **Op** → `*`

# LL(1) Parsing

| E | (int + (int * int)) |
|---|---|

**(1) E → `int`**
**(2) E → `(`E Op E`)`**
**(3) Op → `+`**
**(4) Op → `*`**

# LL(1) Parsing

| | |
|---|---|
| E | (int + (int * int)) |

(1) E → int
(2) E → (E Op E)
(3) Op → +
(4) Op → *

| | int | ( | ) | + | * |
|---|---|---|---|---|---|
| E | 1 | 2 | | | |
| Op | | | | 3 | 4 |

# LL(1) Parsing

| | |
|---|---|
| E$ | (int + (int * int))$ |

(1) E → int
(2) E → (E Op E)
(3) Op → +
(4) Op → *

| | int | ( | ) | + | * |
|---|---|---|---|---|---|
| E | 1 | 2 | | | |
| Op | | | | 3 | 4 |

# LL(1) Parsing

| E$ | (int + (int * int))$ |
|---|---|

(1) E → int
(2) E → (E Op E)
(3) Op → +
(4) Op → *

| | int | ( | ) | + | * |
|---|---|---|---|---|---|
| E | 1 | 2 | | | |
| Op | | | | 3 | 4 |

The $ symbol is the end-of-input marker and is used by the parser to detect when we have reached the end of the input. It is not a part of the grammar.

# LL(1) Parsing

| | |
|---|---|
| E$ | (int + (int * int))$ |

(1) E → int
(2) E → (E Op E)
(3) Op → +
(4) Op → *

| | int | ( | ) | + | * |
|---|---|---|---|---|---|
| E | 1 | 2 | | | |
| Op | | | | 3 | 4 |

# LL(1) Parsing

| E$ | (int + (int * int))$ |
|---|---|

(1) E → int
(2) E → (E Op E)
(3) Op → +
(4) Op → *

| | int | ( | ) | + | * |
|---|---|---|---|---|---|
| E | 1 | 2 | | | |
| Op | | | | 3 | 4 |

The first symbol of our guess is a nonterminal. We then look at our parsing table to see what production to use.

This is called a **predict** step.

# LL(1) Parsing

| | |
|---|---|
| E$ | (int + (int * int))$ |

(1) **E** → `int`
(2) **E** → `(`**E Op E**`)`
(3) **Op** → `+`
(4) **Op** → `*`

| | `int` | `(` | `)` | `+` | `*` |
|---|---|---|---|---|---|
| E | 1 | 2 | | | |
| Op | | | | 3 | 4 |

# LL(1) Parsing

| | |
|---|---|
| E$ | (int + (int * int))$ |
| (E Op E)$ | (int + (int * int))$ |

(1) E → int
(2) E → (E Op E)
(3) Op → +
(4) Op → *

| | int | ( | ) | + | * |
|---|---|---|---|---|---|
| E | 1 | 2 | | | |
| Op | | | | 3 | 4 |

# LL(1) Parsing

| | |
|---|---|
| E$ | (int + (int * int))$ |
| (E Op E)$ | (int + (int * int))$ |

(1) E → int
(2) E → (E Op E)
(3) Op → +
(4) Op → *

| | int | ( | ) | + | * |
|---|---|---|---|---|---|
| E | 1 | 2 | | | |
| Op | | | | 3 | 4 |

The first symbol of our guess is now a terminal symbol. We thus match it against the first symbol of the string to parse.

This is called a match step.

# LL(1) Parsing

| | |
|---|---|
| E$ | (int + (int * int))$ |
| (E Op E)$ | (int + (int * int))$ |
| E Op E)$ | int + (int * int))$ |

**(1)** **E** → `int`
**(2)** **E** → `(`**E Op E**`)`
**(3)** **Op** → `+`
**(4)** **Op** → `*`

| | int | ( | ) | + | * |
|---|---|---|---|---|---|
| E | 1 | 2 | | | |
| Op | | | | 3 | 4 |

# LL(1) Parsing

| | |
|---|---|
| E$ | (int + (int * int))$ |
| (E Op E)$ | (int + (int * int))$ |
| E Op E)$ | int + (int * int))$ |

(1) E → int
(2) E → (E Op E)
(3) Op → +
(4) Op → *

| | int | ( | ) | + | * |
|---|---|---|---|---|---|
| E | 1 | 2 | | | |
| Op | | | | 3 | 4 |

# LL(1) Parsing

| | |
|---|---|
| E$ | (int + (int * int))$ |
| (E Op E)$ | (int + (int * int))$ |
| E Op E)$ | int + (int * int))$ |
| int Op E)$ | int + (int * int))$ |

(1) E → int
(2) E → (E Op E)
(3) Op → +
(4) Op → *

| | int | ( | ) | + | * |
|---|---|---|---|---|---|
| E | 1 | 2 | | | |
| Op | | | | 3 | 4 |

# LL(1) Parsing

(1) E → `int`
(2) E → `(`E Op E`)`
(3) Op → `+`
(4) Op → `*`

| | |
|---|---|
| E$ | (int + (int * int))$ |
| (E Op E)$ | (int + (int * int))$ |
| E Op E)$ | int + (int * int))$ |
| int Op E)$ | int + (int * int))$ |
| Op E)$ | + (int * int))$ |

| | int | ( | ) | + | * |
|---|---|---|---|---|---|
| E | 1 | 2 | | | |
| Op | | | | 3 | 4 |

# LL(1) Parsing

| | |
|---|---|
| E$ | (int + (int * int))$ |
| (E Op E)$ | (int + (int * int))$ |
| E Op E)$ | int + (int * int))$ |
| int Op E)$ | int + (int * int))$ |
| Op E)$ | + (int * int))$ |

(1) **E** → **int**
(2) **E** → **(E Op E)**
(3) **Op** → **+**
(4) **Op** → *****

| | int | ( | ) | + | * |
|---|---|---|---|---|---|
| E | 1 | 2 | | | |
| Op | | | | 3 | 4 |

# LL(1) Parsing

(1) E → `int`
(2) E → `(`E Op E`)`
(3) Op → `+`
(4) Op → `*`

| | |
|---|---|
| E$ | (int + (int * int))$ |
| (E Op E)$ | (int + (int * int))$ |
| E Op E)$ | int + (int * int))$ |
| int Op E)$ | int + (int * int))$ |
| Op E)$ | + (int * int))$ |
| + E)$ | + (int * int))$ |

| | int | ( | ) | + | * |
|---|---|---|---|---|---|
| E | 1 | 2 | | | |
| Op | | | | 3 | 4 |

# LL(1) Parsing

(1) **E** → `int`
(2) **E** → `(`**E Op E**`)`
(3) **Op** → `+`
(4) **Op** → `*`

| | |
|---|---|
| E$ | (int + (int * int))$ |
| (E Op E)$ | (int + (int * int))$ |
| E Op E)$ | int + (int * int))$ |
| int Op E)$ | int + (int * int))$ |
| Op E)$ | + (int * int))$ |
| + E)$ | + (int * int))$ |
| E)$ | (int * int))$ |

| | int | ( | ) | + | * |
|---|---|---|---|---|---|
| E | 1 | 2 | | | |
| Op | | | | 3 | 4 |

# LL(1) Parsing

(1) **E** → `int`
(2) **E** → `(`**E Op E**`)`
(3) **Op** → `+`
(4) **Op** → `*`

| | |
|---|---|
| E$ | (int + (int * int))$ |
| (E Op E)$ | (int + (int * int))$ |
| E Op E)$ | int + (int * int))$ |
| int Op E)$ | int + (int * int))$ |
| Op E)$ | + (int * int))$ |
| + E)$ | + (int * int))$ |
| E)$ | (int * int))$ |

| | int | ( | ) | + | * |
|---|---|---|---|---|---|
| E | 1 | 2 | | | |
| Op | | | | 3 | 4 |

# LL(1) Parsing

(1) E → int
(2) E → (E Op E)
(3) Op → +
(4) Op → *

| | int | ( | ) | + | * |
|---|---|---|---|---|---|
| E | 1 | 2 | | | |
| Op | | | | 3 | 4 |

| | |
|---|---|
| E$ | (int + (int * int))$ |
| (E Op E)$ | (int + (int * int))$ |
| E Op E)$ | int + (int * int))$ |
| int Op E)$ | int + (int * int))$ |
| Op E)$ | + (int * int))$ |
| + E)$ | + (int * int))$ |
| E)$ | (int * int))$ |
| (E Op E))$ | (int * int))$ |

# LL(1) Parsing

(1) E → `int`
(2) E → `(`E Op E`)`
(3) Op → `+`
(4) Op → `*`

| | int | ( | ) | + | * |
|---|---|---|---|---|---|
| E | 1 | 2 | | | |
| Op | | | | 3 | 4 |

| | |
|---|---|
| E$ | (int + (int * int))$ |
| (E Op E)$ | (int + (int * int))$ |
| E Op E)$ | int + (int * int))$ |
| int Op E)$ | int + (int * int))$ |
| Op E)$ | + (int * int))$ |
| + E)$ | + (int * int))$ |
| E)$ | (int * int))$ |
| (E Op E))$ | (int * int))$ |
| E Op E))$ | int * int))$ |

# LL(1) Parsing

(1) E → `int`
(2) E → `(`E Op E`)`
(3) Op → `+`
(4) Op → `*`

| | int | ( | ) | + | * |
|---|---|---|---|---|---|
| E | 1 | 2 | | | |
| Op | | | | 3 | 4 |

| | |
|---|---|
| E$ | (int + (int * int))$ |
| (E Op E)$ | (int + (int * int))$ |
| E Op E)$ | int + (int * int))$ |
| int Op E)$ | int + (int * int))$ |
| Op E)$ | + (int * int))$ |
| + E)$ | + (int * int))$ |
| E)$ | (int * int))$ |
| (E Op E))$ | (int * int))$ |
| E Op E))$ | int * int))$ |

# LL(1) Parsing

(1) **E** → `int`
(2) **E** → `(`**E Op E**`)`
(3) **Op** → `+`
(4) **Op** → `*`

| | int | ( | ) | + | * |
|---|---|---|---|---|---|
| E | 1 | 2 | | | |
| Op | | | | 3 | 4 |

| | |
|---|---|
| E$ | (int + (int * int))$ |
| (E Op E)$ | (int + (int * int))$ |
| E Op E)$ | int + (int * int))$ |
| int Op E)$ | int + (int * int))$ |
| Op E)$ | + (int * int))$ |
| + E)$ | + (int * int))$ |
| E)$ | (int * int))$ |
| (E Op E))$ | (int * int))$ |
| E Op E))$ | int * int))$ |
| int Op E))$ | int * int))$ |

# LL(1) Parsing

(1) **E** → `int`
(2) **E** → `(`**E Op E**`)`
(3) **Op** → `+`
(4) **Op** → `*`

| | int | ( | ) | + | * |
|---|---|---|---|---|---|
| E | 1 | 2 | | | |
| Op | | | | 3 | 4 |

| | |
|---|---|
| E$ | (int + (int * int))$ |
| (E Op E)$ | (int + (int * int))$ |
| E Op E)$ | int + (int * int))$ |
| int Op E)$ | int + (int * int))$ |
| Op E)$ | + (int * int))$ |
| + E)$ | + (int * int))$ |
| E)$ | (int * int))$ |
| (E Op E))$ | (int * int))$ |
| E Op E))$ | int * int))$ |
| int Op E))$ | int * int))$ |
| Op E))$ | * int))$ |

# LL(1) Parsing

(1) E → `int`
(2) E → `(`E Op E`)`
(3) Op → `+`
(4) Op → `*`

| | int | ( | ) | + | * |
|---|---|---|---|---|---|
| E | 1 | 2 | | | |
| Op | | | | 3 | 4 |

| | |
|---|---|
| E$ | (int + (int * int))$ |
| (E Op E)$ | (int + (int * int))$ |
| E Op E)$ | int + (int * int))$ |
| int Op E)$ | int + (int * int))$ |
| Op E)$ | + (int * int))$ |
| + E)$ | + (int * int))$ |
| E)$ | (int * int))$ |
| (E Op E))$ | (int * int))$ |
| E Op E))$ | int * int))$ |
| int Op E))$ | int * int))$ |
| Op E))$ | * int))$ |

# LL(1) Parsing

(1) **E** → `int`
(2) **E** → `(`**E Op E**`)`
(3) **Op** → `+`
(4) **Op** → `*`

| | int | ( | ) | + | * |
|---|---|---|---|---|---|
| E | 1 | 2 | | | |
| Op | | | | 3 | 4 |

| | |
|---|---|
| E$ | (int + (int * int))$ |
| (E Op E)$ | (int + (int * int))$ |
| E Op E)$ | int + (int * int))$ |
| int Op E)$ | int + (int * int))$ |
| Op E)$ | + (int * int))$ |
| + E)$ | + (int * int))$ |
| E)$ | (int * int))$ |
| (E Op E))$ | (int * int))$ |
| E Op E))$ | int * int))$ |
| int Op E))$ | int * int))$ |
| Op E))$ | * int))$ |
| * E))$ | * int))$ |

# LL(1) Parsing

(1) **E** → `int`
(2) **E** → `(`**E Op E**`)`
(3) **Op** → `+`
(4) **Op** → `*`

| | int | ( | ) | + | * |
|---|---|---|---|---|---|
| E | 1 | 2 | | | |
| Op | | | | 3 | 4 |

| | |
|---|---|
| E$ | (int + (int * int))$ |
| (E Op E)$ | (int + (int * int))$ |
| E Op E)$ | int + (int * int))$ |
| int Op E)$ | int + (int * int))$ |
| Op E)$ | + (int * int))$ |
| + E)$ | + (int * int))$ |
| E)$ | (int * int))$ |
| (E Op E))$ | (int * int))$ |
| E Op E))$ | int * int))$ |
| int Op E))$ | int * int))$ |
| Op E))$ | * int))$ |
| * E))$ | * int))$ |
| E))$ | int))$ |

# LL(1) Parsing

(1) **E** → `int`
(2) **E** → **(E Op E)**
(3) **Op** → `+`
(4) **Op** → `*`

| | int | ( | ) | + | * |
|---|---|---|---|---|---|
| E | 1 | 2 | | | |
| Op | | | | 3 | 4 |

| | |
|---|---|
| E$ | (int + (int * int))$ |
| (E Op E)$ | (int + (int * int))$ |
| E Op E)$ | int + (int * int))$ |
| int Op E)$ | int + (int * int))$ |
| Op E)$ | + (int * int))$ |
| + E)$ | + (int * int))$ |
| E)$ | (int * int))$ |
| (E Op E))$ | (int * int))$ |
| E Op E))$ | int * int))$ |
| int Op E))$ | int * int))$ |
| Op E))$ | * int))$ |
| * E))$ | * int))$ |
| E))$ | int))$ |

# LL(1) Parsing

(1) **E** → `int`
(2) **E** → `(`**E Op E**`)`
(3) **Op** → `+`
(4) **Op** → `*`

| | int | ( | ) | + | * |
|---|---|---|---|---|---|
| E | 1 | 2 | | | |
| Op | | | | 3 | 4 |

| | |
|---|---|
| E$ | (int + (int * int))$ |
| (E Op E)$ | (int + (int * int))$ |
| E Op E)$ | int + (int * int))$ |
| int Op E)$ | int + (int * int))$ |
| Op E)$ | + (int * int))$ |
| + E)$ | + (int * int))$ |
| E)$ | (int * int))$ |
| (E Op E))$ | (int * int))$ |
| E Op E))$ | int * int))$ |
| int Op E))$ | int * int))$ |
| Op E))$ | * int))$ |
| * E))$ | * int))$ |
| E))$ | int))$ |
| int))$ | int))$ |

# LL(1) Parsing

(1) **E** → `int`
(2) **E** → `(`**E Op E**`)`
(3) **Op** → `+`
(4) **Op** → `*`

| | int | ( | ) | + | * |
|---|---|---|---|---|---|
| E | 1 | 2 | | | |
| Op | | | | 3 | 4 |

| | |
|---|---|
| E$ | (int + (int * int))$ |
| (E Op E)$ | (int + (int * int))$ |
| E Op E)$ | int + (int * int))$ |
| int Op E)$ | int + (int * int))$ |
| Op E)$ | + (int * int))$ |
| + E)$ | + (int * int))$ |
| E)$ | (int * int))$ |
| (E Op E))$ | (int * int))$ |
| E Op E))$ | int * int))$ |
| int Op E))$ | int * int))$ |
| Op E))$ | * int))$ |
| * E))$ | * int))$ |
| E))$ | int))$ |
| int))$ | int))$ |
| ))$ | ))$ |

# LL(1) Parsing

(1) E → int
(2) E → (E Op E)
(3) Op → +
(4) Op → *

| | int | ( | ) | + | * |
|---|---|---|---|---|---|
| E | 1 | 2 | | | |
| Op | | | | 3 | 4 |

| | |
|---|---|
| E$ | (int + (int * int))$ |
| (E Op E)$ | (int + (int * int))$ |
| E Op E)$ | int + (int * int))$ |
| int Op E)$ | int + (int * int))$ |
| Op E)$ | + (int * int))$ |
| + E)$ | + (int * int))$ |
| E)$ | (int * int))$ |
| (E Op E))$ | (int * int))$ |
| E Op E))$ | int * int))$ |
| int Op E))$ | int * int))$ |
| Op E))$ | * int))$ |
| * E))$ | * int))$ |
| E))$ | int))$ |
| int))$ | int))$ |
| ))$ | ))$ |
| )$ | )$ |

# LL(1) Parsing

(1) E → `int`
(2) E → `(`E Op E`)`
(3) Op → `+`
(4) Op → `*`

| | int | ( | ) | + | * |
|---|---|---|---|---|---|
| E | 1 | 2 | | | |
| Op | | | | 3 | 4 |

| | |
|---|---|
| E$ | (int + (int * int))$ |
| (E Op E)$ | (int + (int * int))$ |
| E Op E)$ | int + (int * int))$ |
| int Op E)$ | int + (int * int))$ |
| Op E)$ | + (int * int))$ |
| + E)$ | + (int * int))$ |
| E)$ | (int * int))$ |
| (E Op E))$ | (int * int))$ |
| E Op E))$ | int * int))$ |
| int Op E))$ | int * int))$ |
| Op E))$ | * int))$ |
| * E))$ | * int))$ |
| E))$ | int))$ |
| int))$ | int))$ |
| ))$ | ))$ |
| )$ | )$ |
| $ | $ |

# LL(1) Parsing

(1) E → int
(2) E → (E Op E)
(3) Op → +
(4) Op → *

| | int | ( | ) |
|---|---|---|---|
| E | 1 | 2 | |
| Op | | | |


| | |
|---|---|
| E$ | (int + (int * int))$ |
| (E Op E)$ | (int + (int * int))$ |
| E Op E)$ | int + (int * int))$ |
| int Op E)$ | int + (int * int))$ |
| Op E)$ | + (int * int))$ |
| | + (int * int))$ |
| | (int * int))$ |
| | (int * int))$ |
| | int * int))$ |
| | int * int))$ |
| | * int))$ |
| | * int))$ |
| | int))$ |
| int))$ | int))$ |
| ))$ | ))$ |
| )$ | )$ |
| $ | $ |

# The Limits of LL(1)

# A Grammar that is Not LL(1)

- Consider the following (left-recursive) grammar:

  **A** → **A**`b` | `c`

- FIRST(**A**) = {`c`}
- However, we cannot build an LL(1) parse table.
- **Why?**

# A Grammar that is Not LL(1)

- Consider the following (left-recursive) grammar:

  A → A`b` | `c`

- FIRST(A) = {`c`}
- However, we cannot build an LL(1) parse table.
- **Why?**

| | b | c |
|---|---|---|
| A | | A → A`b`<br>A → `c` |

# A Grammar that is Not LL(1)

- Consider the following (left-recursive) grammar:

  **A** → **A**`b` | `c`

- FIRST(**A**) = {`c`}
- However, we cannot build an LL(1) parse table.
- **Why?**

| | b | c |
|---|---|---|
| A | | **A** → **A**`b`<br>**A** → `c` |

- **Cannot uniquely predict production!**
- This is called a **FIRST/FIRST conflict**.

# The Strengths of LL(1)

# LL(1) is Straightforward

- Can be implemented quickly with a table-driven design.
- Can be implemented by **recursive descent**:
  - Define a function for each nonterminal.
  - Have these functions call each other based on the lookahead token.

# LL(1) is Fast

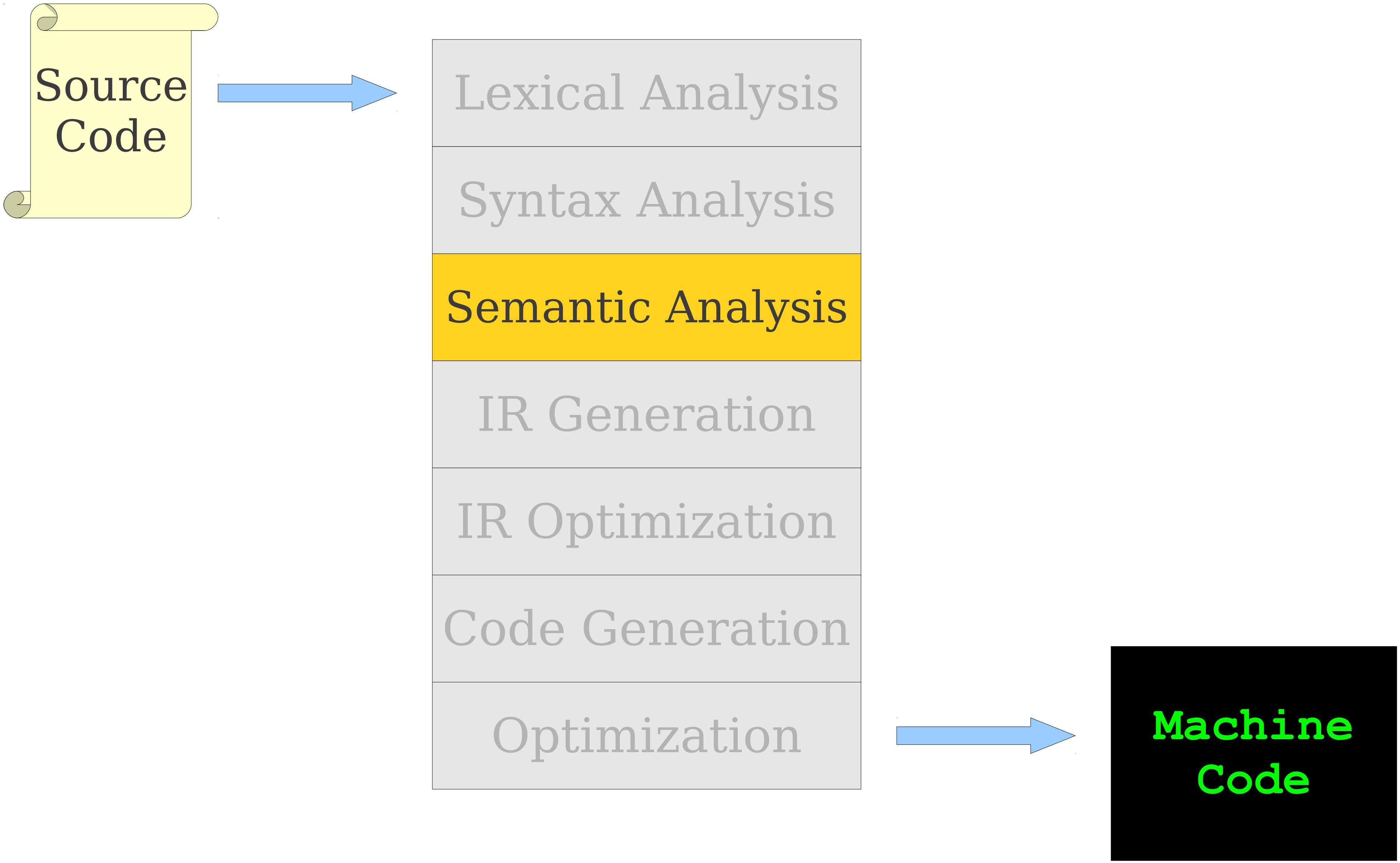
- Both table-driven LL(1) and recursive-descent-powered LL(1) are fast.
- Can parse in O($n$ $|G|$) time, where $n$ is the length of the string and $|G|$ is the size of the grammar.

# Summary

- **Top-down parsing** tries to derive the user's program from the start symbol.
- **Leftmost BFS** is one approach to top-down parsing; it is mostly of theoretical interest.
- **Leftmost DFS** is another approach to top-down parsing that is uncommon in practice.
- **LL(1)** parsing scans from left-to-right, using one token of lookahead to find a leftmost derivation.
- **Left recursion** and **left factorability** cause LL(1) to fail and can be mechanically eliminated in some cases.

# Semantic Analysis

# Where We Are



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

# Where We Are

- Program is *lexically* well-formed:
  - Identifiers have valid names.
  - Strings are properly terminated.
  - No stray characters.
- Program is *syntactically* well-formed:
  - Class declarations have the correct structure.
  - Expressions are syntactically valid.
- Does this mean that the program is legal?

# Semantic Analysis

- Ensure that the program has a well-defined **meaning**.
- Verify properties of the program that aren't caught during the earlier phases:
  - Variables are declared before they're used.
  - Expressions have the right types.
  - Classes don't inherit from nonexistent base classes
  - ...
- Once we finish semantic analysis, we know that the user's input program is legal.

# Challenges in Semantic Analysis

- Reject the largest number of incorrect programs.
- Accept the largest number of correct programs.

# Validity versus Correctness

```
int Fibonacci(int n) {
    if (n <= 1) return 0;

    return Fibonacci(n - 1) + Fibonacci(n - 2);
}

int main() {
    Print(Fibonacci(40));
}
```

# Validity versus Correctness

```
int Fibonacci(int n) {
    if (n <= 1) return 0;

    return Fibonacci(n - 1) + Fibonacci(n - 2);
}

int main() {
    Print(Fibonacci(40));
}
```

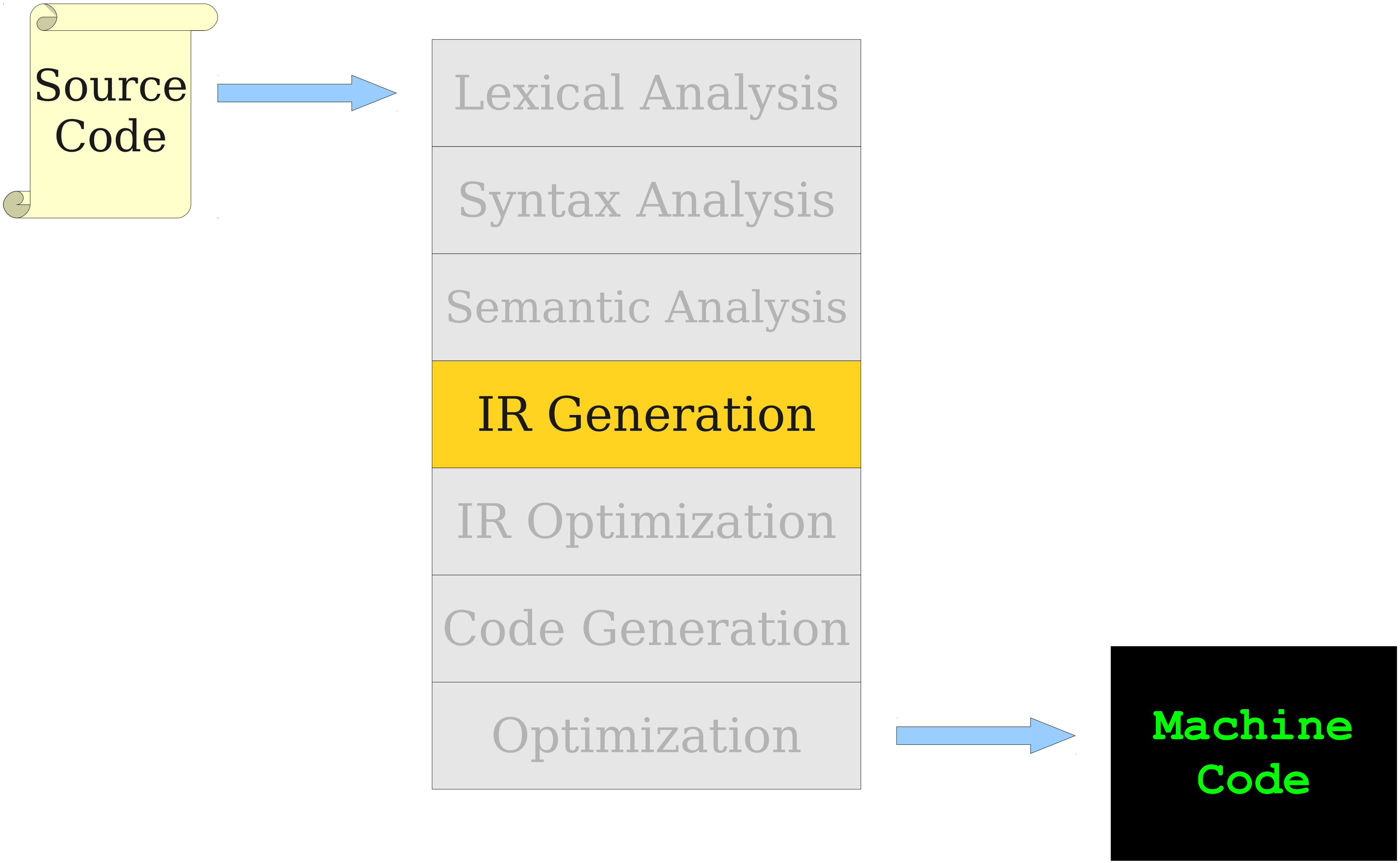
Incorrect, should be "return n;"

# Challenges in Semantic Analysis

- Reject the largest number of incorrect programs.
- Accept the largest number of correct programs.
- Do so quickly.

# Three-Address Code IR

# Where We Are


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

# An Important Detail

- When generating IR at this level, you do **not** need to worry about optimizing it.
- It's okay to generate IR that has lots of unnecessary assignments, redundant computations, etc.
- We'll see how to optimize IR code later.
  - It's tricky, but extremely cool!

# Temporary Variables

- The “three” in “three-address code” refers to the number of operands in any instruction.
- Evaluating an expression with more than three subexpressions requires the introduction of temporary variables.
- This is actually a lot easier than you might think; we'll see how to do it later on.

# Sample TAC Code

```
int a;
int b;

a = 5 + 2 * b;
```

# Sample TAC Code

```
int a;
int b;

a = 5 + 2 * b;
```

```
_t0 = 5;
_t1 = 2 * b;
a = _t0 + _t1;
```

# Sample TAC Code

```
int a;
int b;

a = 5 + 2 * b;
```

TAC allows for instructions with two operands.

```
_t0 = 5;
_t1 = 2 * b;
a = _t0 + _t1;
```

# Simple TAC Instructions

- **Variable assignment** allows assignments of the form
  - `var = constant;`
  - $\texttt{var}_1$ `=` $\texttt{var}_2$`;`
  - $\texttt{var}_1$ `=` $\texttt{var}_2$ **`op`** $\texttt{var}_3$`;`
  - $\texttt{var}_1$ `= constant` **`op`** $\texttt{var}_2$`;`
  - $\texttt{var}_1$ `=` $\texttt{var}_2$ **`op`** `constant;`
  - `var =` $\texttt{constant}_1$ **`op`** $\texttt{constant}_2$`;`
- Permitted operators are **`+`**, **`-`**, **`*`**, **`/`**, **`%`**.
- How would you compile **`y = -x`**`;` ?

# Control Flow Statements

```
int x;
int y;
int z;

if (x < y)
   z = x;
else
   z = y;

z = z * z;
```

# Control Flow Statements

```
int x;
int y;
int z;

if (x < y)
    z = x;
else
    z = y;

z = z * z;
```

```
    _t0 = x < y;
    IfZ _t0 Goto _L0;
    z = x;
    Goto _L1;
_L0:
    z = y;
_L1:
    z = z * z;
```

# Control Flow Statements

```
int x;
int y;
int z;

if (x < y)
    z = x;
else
    z = y;

z = z * z;
```

```
    _t0 = x < y;
    IfZ _t0 Goto _L0;
    z = x;
    Goto _L1;
_L0:
    z = y;
_L1:
    z = z * z;
```

# Control Flow Statements

```
int x;
int y;
int z;

if (x < y)
    z = x;
else
    z = y;

z = z * z;
```

```
    _t0 = x < y;
    IfZ _t0 Goto _L0;
    z = x;
    Goto _L1;
_L0:
    z = y;
_L1:
    z = z * z;
```

# Labels

- TAC allows for **named labels** indicating particular points in the code that can be jumped to.
- There are two control flow instructions:
  - **`Goto`** ***`label;`***
  - **`IfZ`** ***`value`*** **`Goto`** ***`label;`***
- Note that **`IfZ`** is always paired with **`Goto`**.

# Control Flow Statements

```
int x;
int y;

while (x < y) {
    x = x * 2;
}

y = x;
```

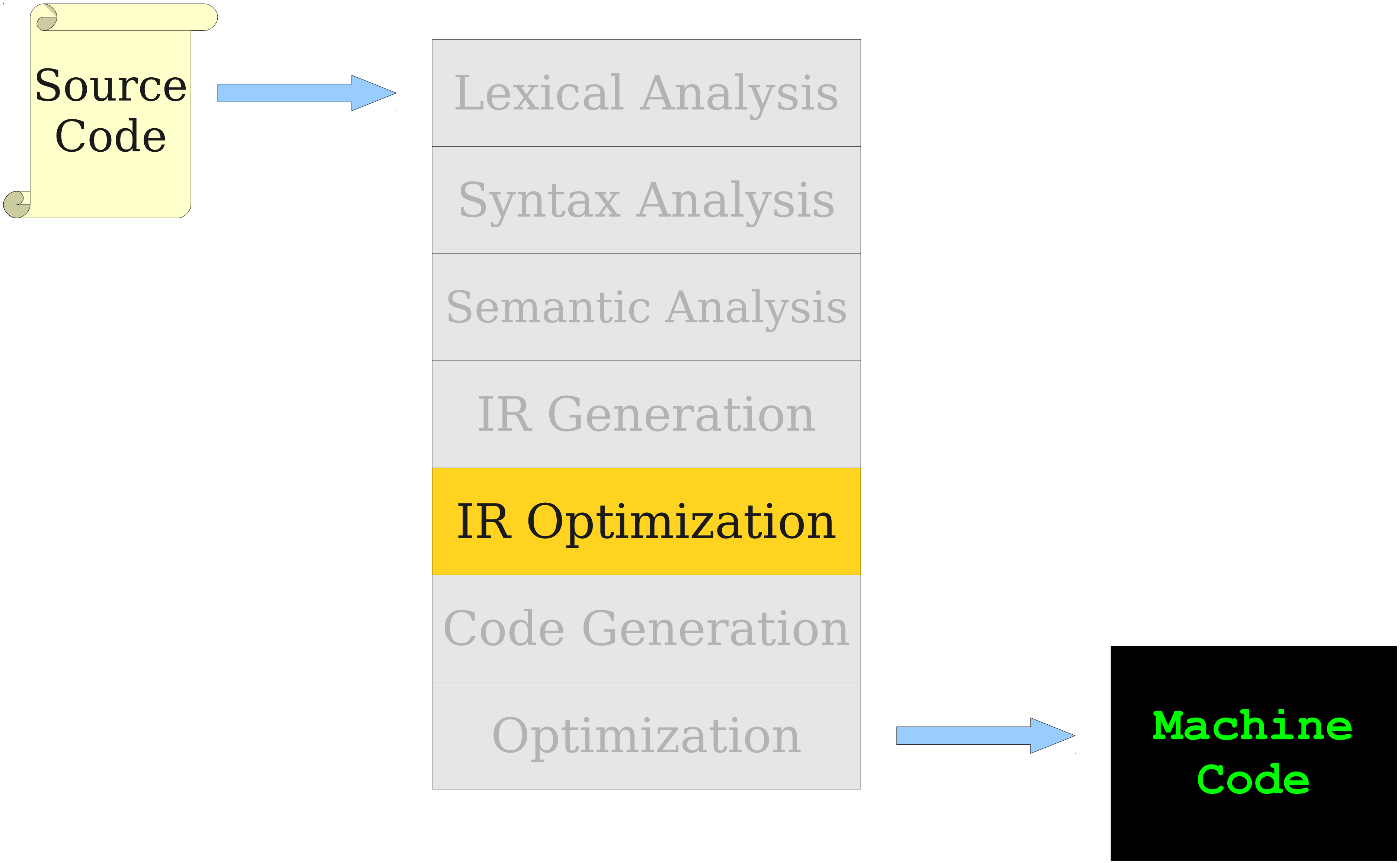
# Control Flow Statements

```
int x;
int y;

while (x < y) {
    x = x * 2;
}

y = x;
```

```
_L0:
    _t0 = x < y;
    IfZ _t0 Goto _L1;
    x = x * 2;
    Goto _L0;
_L1:
    y = x;
```

# IR Optimization

# Where We Are


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

# IR Optimization

- **Goal**: Improve the IR generated by the previous step to take better advantage of resources.
- One of the most important and complex parts of any modern compiler.
- A very active area of research.
- There is a whole class (CS243) dedicated to this material.

# Sources of Optimization

- In order to optimize our IR, we need to understand why it can be improved in the first place.
- **Reason one:** IR generation introduces redundancy.
  - A naïve translation of high-level language features into IR often introduces subcomputations.
  - Those subcomputations can often be sped up, shared, or eliminated.
- **Reason two:** Programmers are lazy.
  - Code executed inside of a loop can often be factored out of the loop.
  - Language features with side effects often used for purposes other than those side effects.

# Optimizations from IR Generation

```
int x;
int y;
bool b1;
bool b2;
bool b3;

b1 = x + x < y
b2 = x + x == y
b3 = x + x > y
```

# Optimizations from IR Generation

```
int x;
int y;
bool b1;
bool b2;
bool b3;

b1 = x + x < y
b2 = x + x == y
b3 = x + x > y
```

```
_t0 = x + x;
_t1 = y;
b1 = _t0 < _t1;

_t2 = x + x;
_t3 = y;
b2 = _t2 == _t3;

_t4 = x + x;
_t5 = y;
b3 = _t5 < _t4;
```

# Optimizations from IR Generation

```
int x;
int y;
bool b1;
bool b2;
bool b3;

b1 = x + x < y
b2 = x + x == y
b3 = x + x > y
```

```
_t0 = x + x;
_t1 = y;
b1 = _t0 < _t1;

_t2 = x + x;
_t3 = y;
b2 = _t2 == _t3;

_t4 = x + x;
_t5 = y;
b3 = _t5 < _t4;
```

# Optimizations from IR Generation

```
int x;
int y;
bool b1;
bool b2;
bool b3;

b1 = x + x < y
b2 = x + x == y
b3 = x + x > y
```

```
_t0 = x + x;
_t1 = y;
b1 = _t0 < _t1;



b2 = _t0 == _t1;



b3 = _t0 < _t1;
```

```
while (x < y + z) {
    x = x - y;
}
```

# Optimizations from Lazy Coders

```
while (x < y + z) {
    x = x - y;
}
```

```
_L0:
    _t0 = y + z;
    _t1 = x < _t0;
    IfZ _t1 Goto _L1;
    x = x - y;
    Goto _L0;
_L1:
```

# Optimizations from Lazy Coders

```
while (x < y + z) {
    x = x - y;
}
```

```
_L0:
    _t0 = y + z;
    _t1 = x < _t0;
    IfZ _t1 Goto _L1;
    x = x - y;
    Goto _L0;
_L1:
```

# Optimizations from Lazy Coders

```
while (x < y + z) {
    x = x - y;
}
```

```
    _t0 = y + z;
_L0:
    _t1 = x < _t0;
    IfZ _t1 Goto _L1;
    x = x - y;
    Goto _L0;
_L1:
```

# Optimizations from Lazy Coders

```
while (x < y + z) {
    x = x - y;
}
```

```
    _t0 = y + z;
_L0:
    _t1 = x < _t0;
    IfZ _t1 Goto _L1;
    x = x - y;
    Goto _L0;
_L1:
```

# A Note on Terminology

- The term “optimization” implies looking for an “optimal” piece of code for a program.
- This is, in general, undecidable.
  - e.g. create a program that can be simplified iff some other program halts.
- Our goal will be IR *improvement* rather than IR *optimization*.

# The Challenge of Optimization

- A good optimizer
  - Should never change the observable behavior of a program.
  - Should produce IR that is as efficient as possible.
  - Should not take too long to process inputs.
- Unfortunately:
  - Even good optimizers sometimes introduce bugs into code.
  - Optimizers often miss “easy” optimizations due to limitations of their algorithms.
  - Almost all interesting optimizations are **NP**-hard or undecidable.

# What are we Optimizing?

- Optimizers can try to improve code usage with respect to many observable properties.
- What are some quantities we might want to optimize?

# What are we Optimizing?

- Optimizers can try to improve code usage with respect to many observable properties.
- What are some quantities we might want to optimize?
- **Runtime** (make the program as fast as possible at the expense of time and power)
- **Memory usage** (generate the smallest possible executable at the expense of time and power)
- **Power consumption** (choose simple instructions at the expense of speed and memory usage)
- Plus a lot more (minimize function calls, reduce use of floating-point hardware, etc.)

# Overview of IR Optimization

- **Formalisms and Terminology**
  - Control-flow graphs.
  - Basic blocks.
- **Local optimizations**
  - Speeding up small pieces of a function.
- **Global optimizations**
  - Speeding up functions as a whole.

# Formalisms and Terminology

# Analyzing a Program

- In order to optimize a program, the compiler has to be able to reason about the properties of that program.
- An analysis is called **sound** if it never asserts an incorrect fact about a program.
- All the analyses we will discuss in this class are sound.
  - *(Why?)*

# Soundness

```
int x;
int y;

if (y < 5)
    x = 137;
else
    x = 42;

Print(x);
```

# Soundness

```
int x;
int y;

if (y < 5)
    x = 137;
else
    x = 42;

Print(x);
```

# Soundness

```
int x;
int y;

if (y < 5)
    x = 137;
else
    x = 42;

Print(x);
```

"At this point in the program, **x** holds some integer value."

# Soundness

```
int x;
int y;

if (y < 5)
    x = 137;
else
    x = 42;

Print(x);
```

"At this point in the program, **x** is either 137 or 42"

# Semantics-Preserving Optimizations

- An optimization is **semantics-preserving** if it does not alter the semantics of the original program.
- Examples:
  - Eliminating unnecessary temporary variables.
  - Computing values that are known statically at compile-time instead of runtime.
  - Evaluating constant expressions outside of a loop instead of inside.
- Non-examples:
  - Replacing bubble sort with quicksort.
- The optimizations we will consider in this class are all semantics-preserving.

# A Formalism for IR Optimization

- Every phase of the compiler uses some new abstraction:
  - Scanning uses regular expressions.
  - Parsing uses CFGs.
  - Semantic analysis uses proof systems and symbol tables.
  - IR generation uses ASTs.
- In optimization, we need a formalism that captures the structure of a program in a way amenable to optimization.

# Visualizing IR

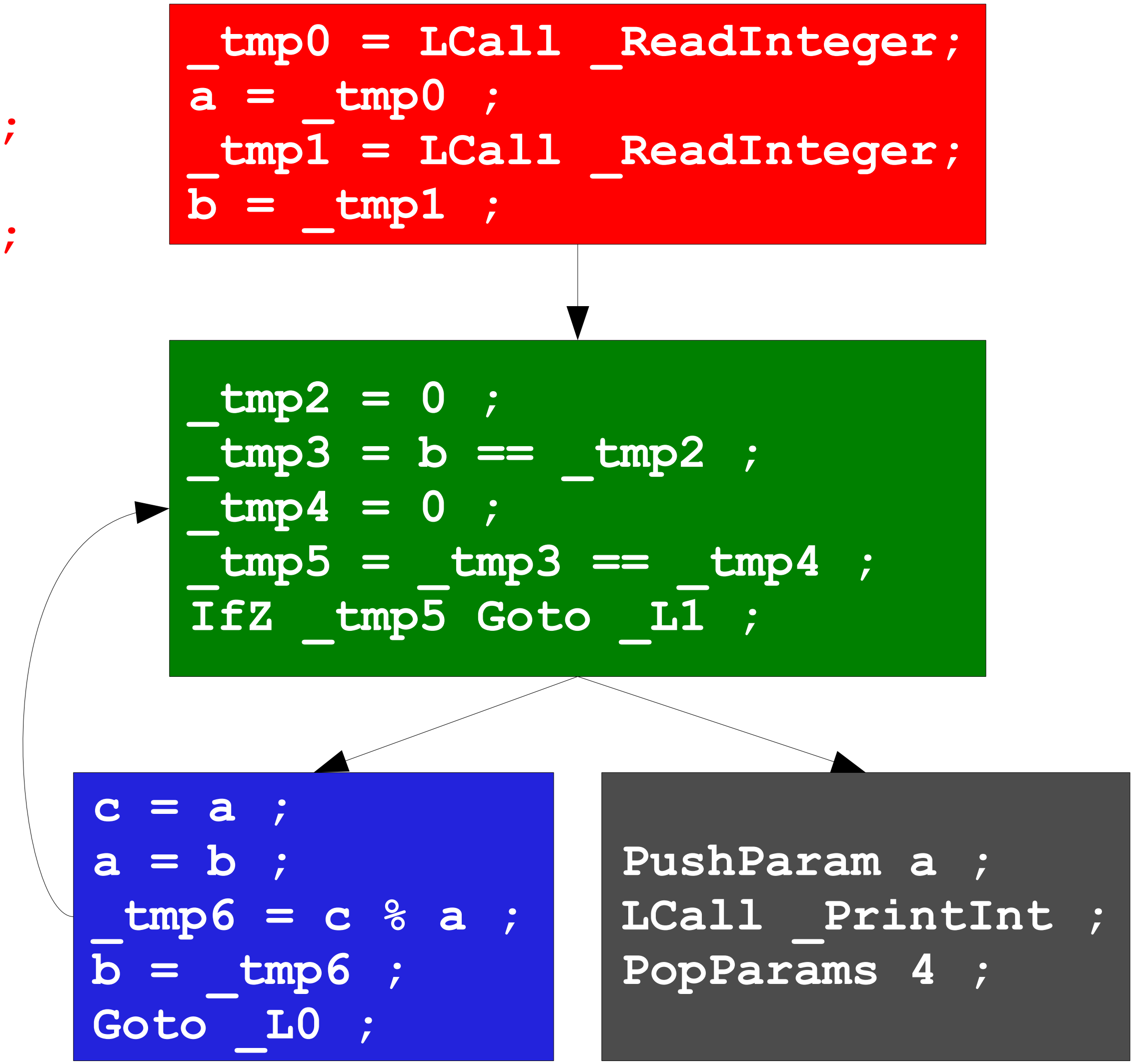
```
main:
    BeginFunc 40;
    _tmp0 = LCall _ReadInteger;
    a = _tmp0;
    _tmp1 = LCall _ReadInteger;
    b = _tmp1;
_L0:
    _tmp2 = 0;
    _tmp3 = b == _tmp2;
    _tmp4 = 0;
    _tmp5 = _tmp3 == _tmp4;
    IfZ _tmp5 Goto _L1;
    c = a;
    a = b;
    _tmp6 = c % a;
    b = _tmp6;
    Goto _L0;
_L1:
    PushParam a;
    LCall _PrintInt;
    PopParams 4;
    EndFunc;
```

# Visualizing IR

```
main:
    BeginFunc 40;
    _tmp0 = LCall _ReadInteger;
    a = _tmp0;
    _tmp1 = LCall _ReadInteger;
    b = _tmp1;
_L0:
    _tmp2 = 0;
    _tmp3 = b == _tmp2;
    _tmp4 = 0;
    _tmp5 = _tmp3 == _tmp4;
    IfZ _tmp5 Goto _L1;
    c = a;
    a = b;
    _tmp6 = c % a;
    b = _tmp6;
    Goto _L0;
_L1:
    PushParam a;
    LCall _PrintInt;
    PopParams 4;
    EndFunc;
```

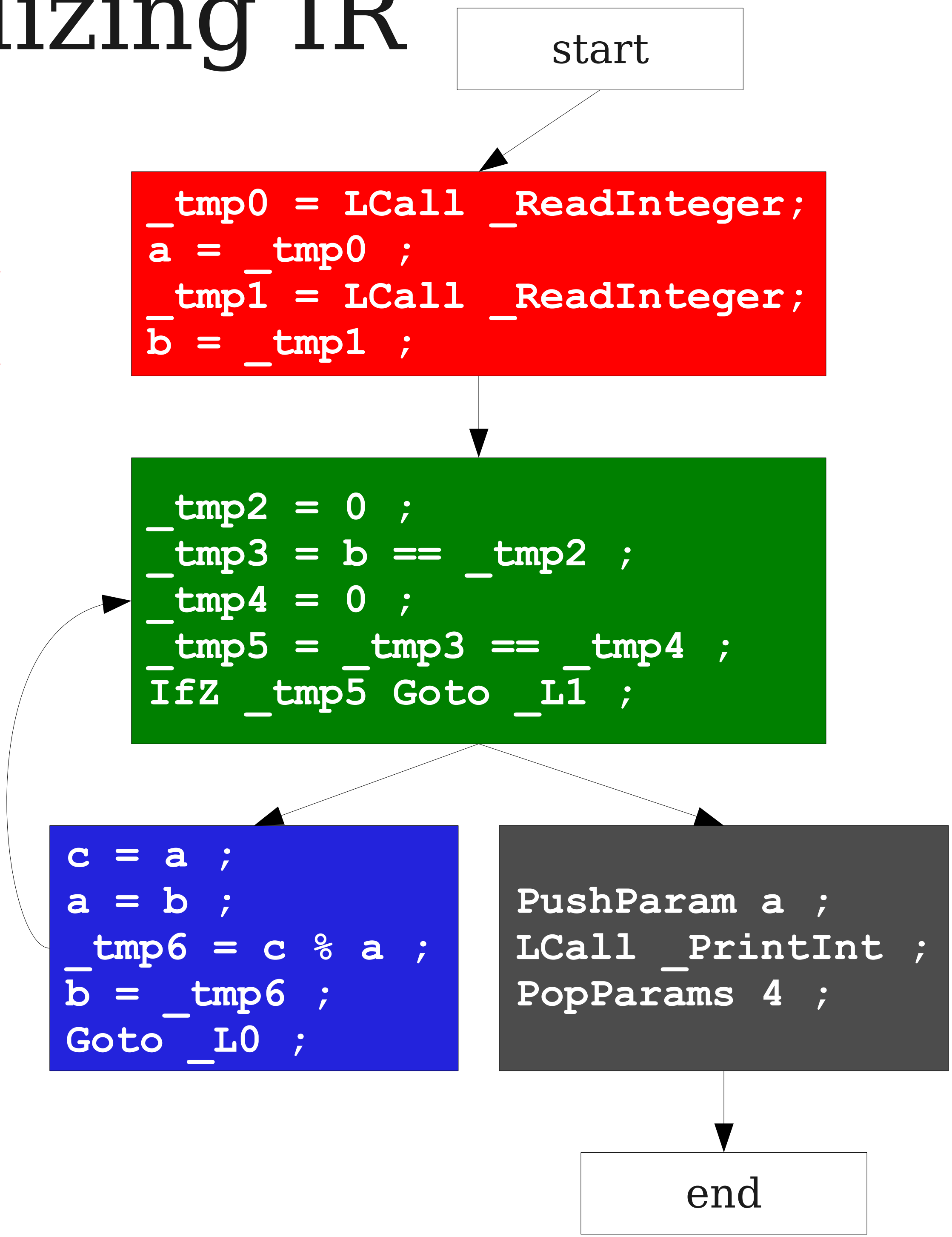
# Visualizing IR

```
main:
    BeginFunc 40;
    _tmp0 = LCall _ReadInteger;
    a = _tmp0;
    _tmp1 = LCall _ReadInteger;
    b = _tmp1;
_L0:
    _tmp2 = 0;
    _tmp3 = b == _tmp2;
    _tmp4 = 0;
    _tmp5 = _tmp3 == _tmp4;
    IfZ _tmp5 Goto _L1;
    c = a;
    a = b;
    _tmp6 = c % a;
    b = _tmp6;
    Goto _L0;
_L1:
    PushParam a;
    LCall _PrintInt;
    PopParams 4;
    EndFunc;
```



# Visualizing IR

```
main:
    BeginFunc 40;
    _tmp0 = LCall _ReadInteger;
    a = _tmp0;
    _tmp1 = LCall _ReadInteger;
    b = _tmp1;
_L0:
    _tmp2 = 0;
    _tmp3 = b == _tmp2;
    _tmp4 = 0;
    _tmp5 = _tmp3 == _tmp4;
    IfZ _tmp5 Goto _L1;
    c = a;
    a = b;
    _tmp6 = c % a;
    b = _tmp6;
    Goto _L0;
_L1:
    PushParam a;
    LCall _PrintInt;
    PopParams 4;
    EndFunc;
```



# Basic Blocks

- A **basic block** is a sequence of IR instructions where
  - There is exactly one spot where control enters the sequence, which must be at the start of the sequence.
  - There is exactly one spot where control leaves the sequence, which must be at the end of the sequence.
- Informally, a sequence of instructions that always execute as a group.

# Control-Flow Graphs

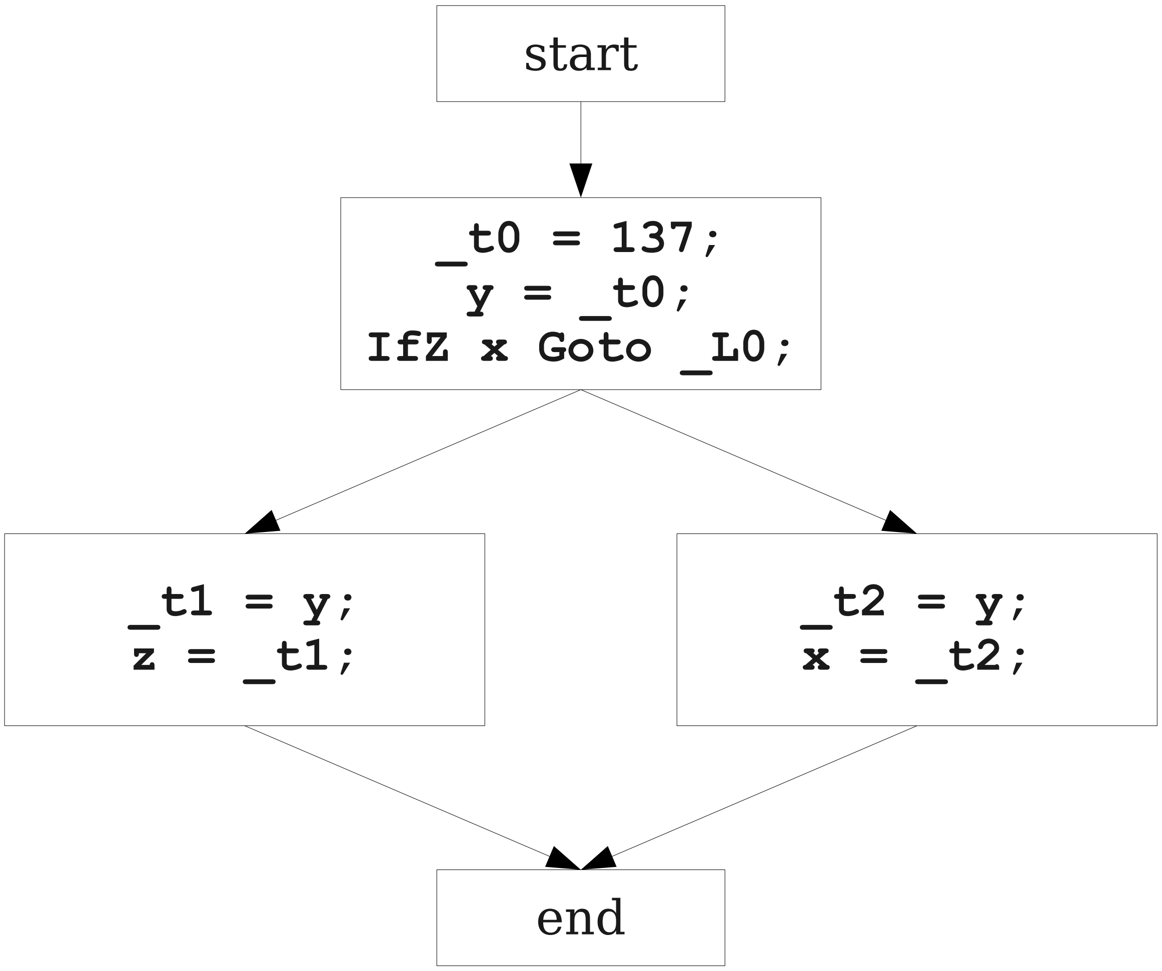
- A **control-flow graph** (CFG) is a graph of the basic blocks in a function.
  - The term CFG is overloaded – from here on out, we'll mean "control-flow graph" and not "context-free grammar."
- Each edge from one basic block to another indicates that control can flow from the end of the first block to the start of the second block.
- There is a dedicated node for the start and end of a function.

# Types of Optimizations

- An optimization is **local** if it works on just a single basic block.
- An optimization is **global** if it works on an entire control-flow graph.
- An optimization is **interprocedural** if it works across the control-flow graphs of multiple functions.
  - We won't talk about this in this course.

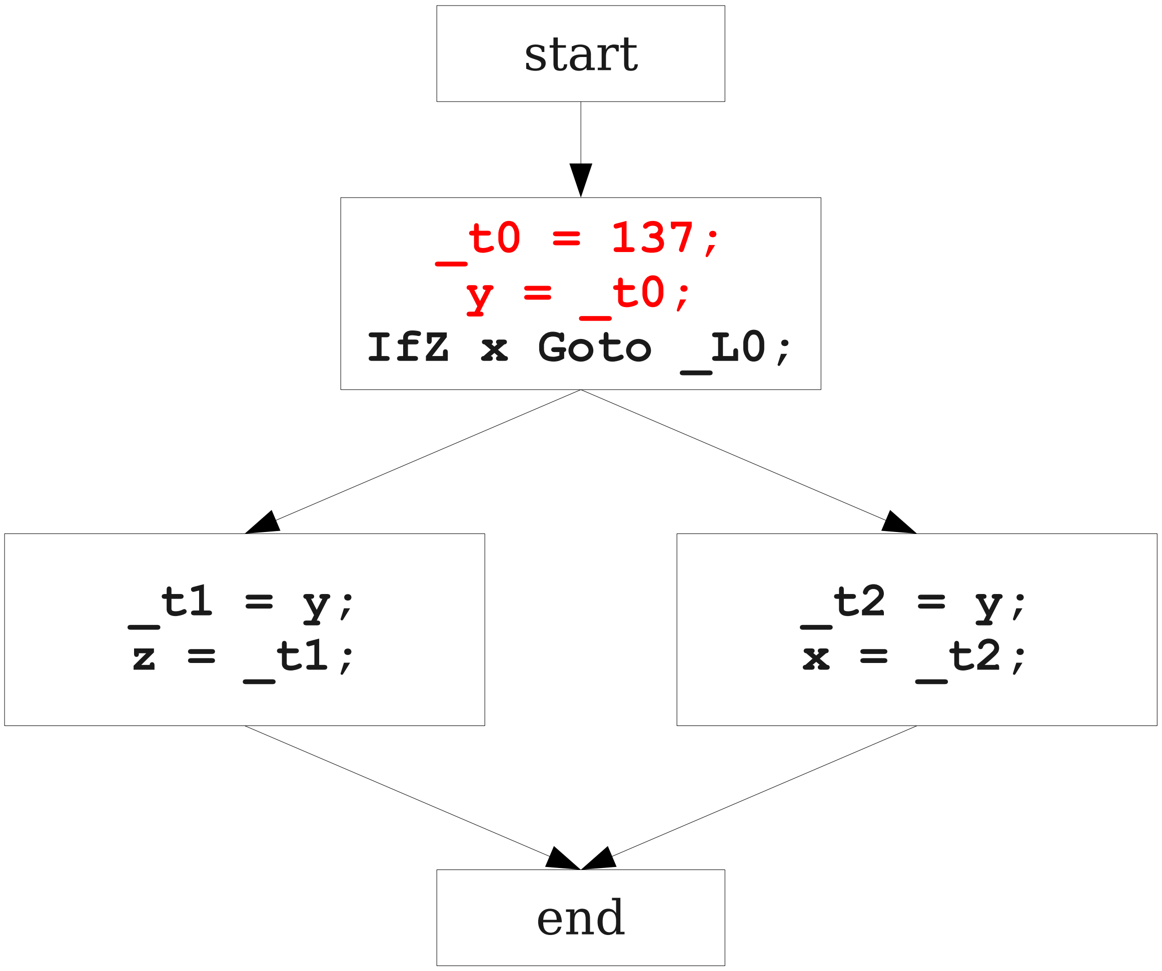
# Local Optimizations

```
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```



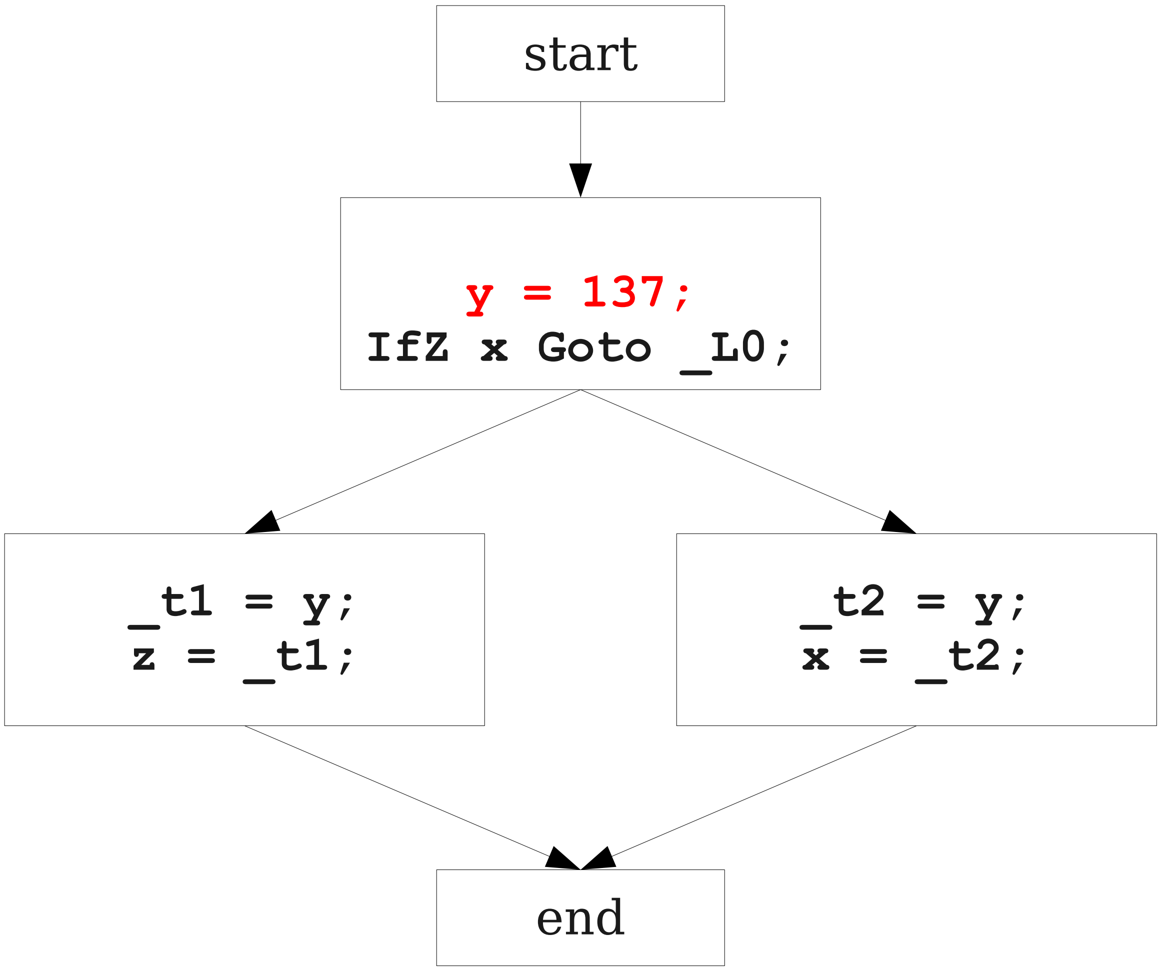
# Local Optimizations

```
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```



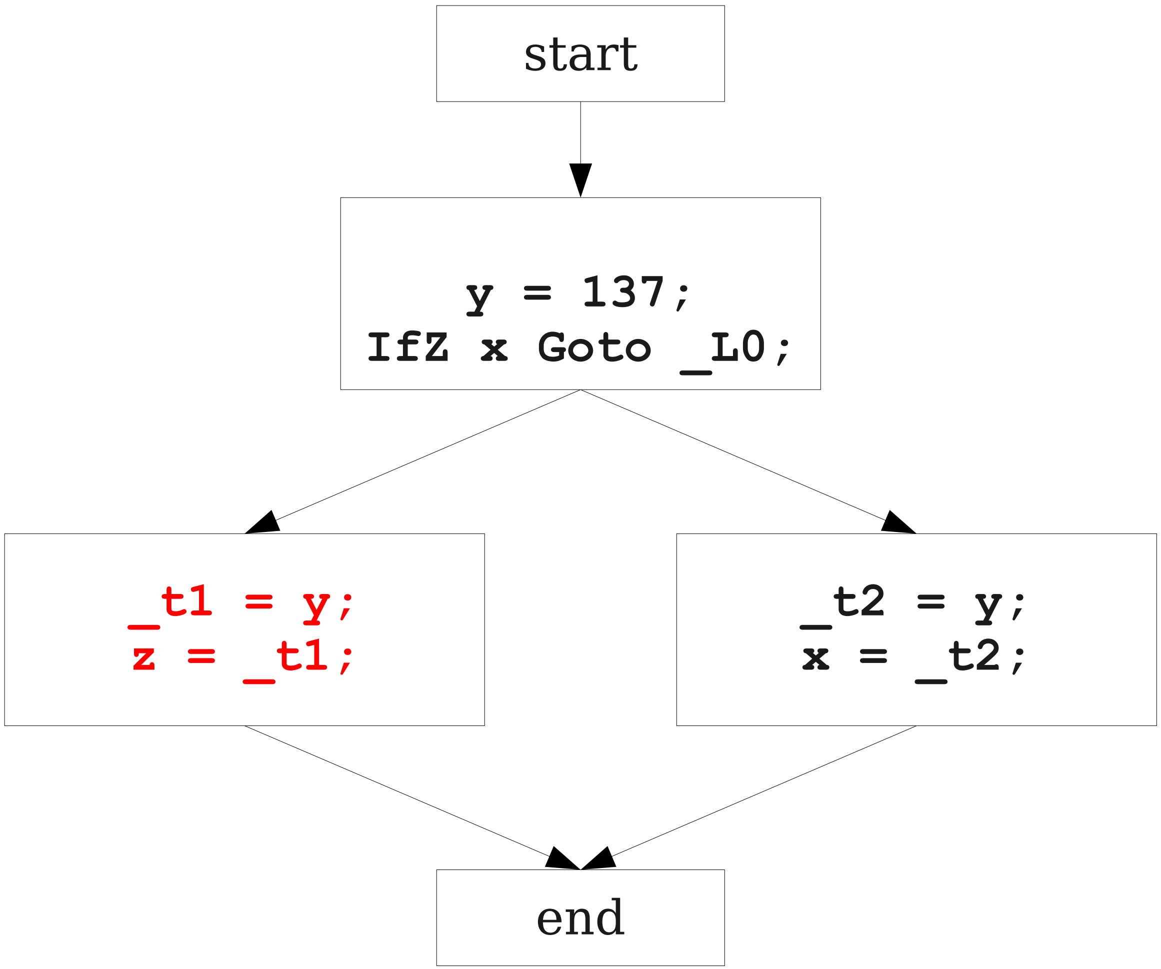
# Local Optimizations

```
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```



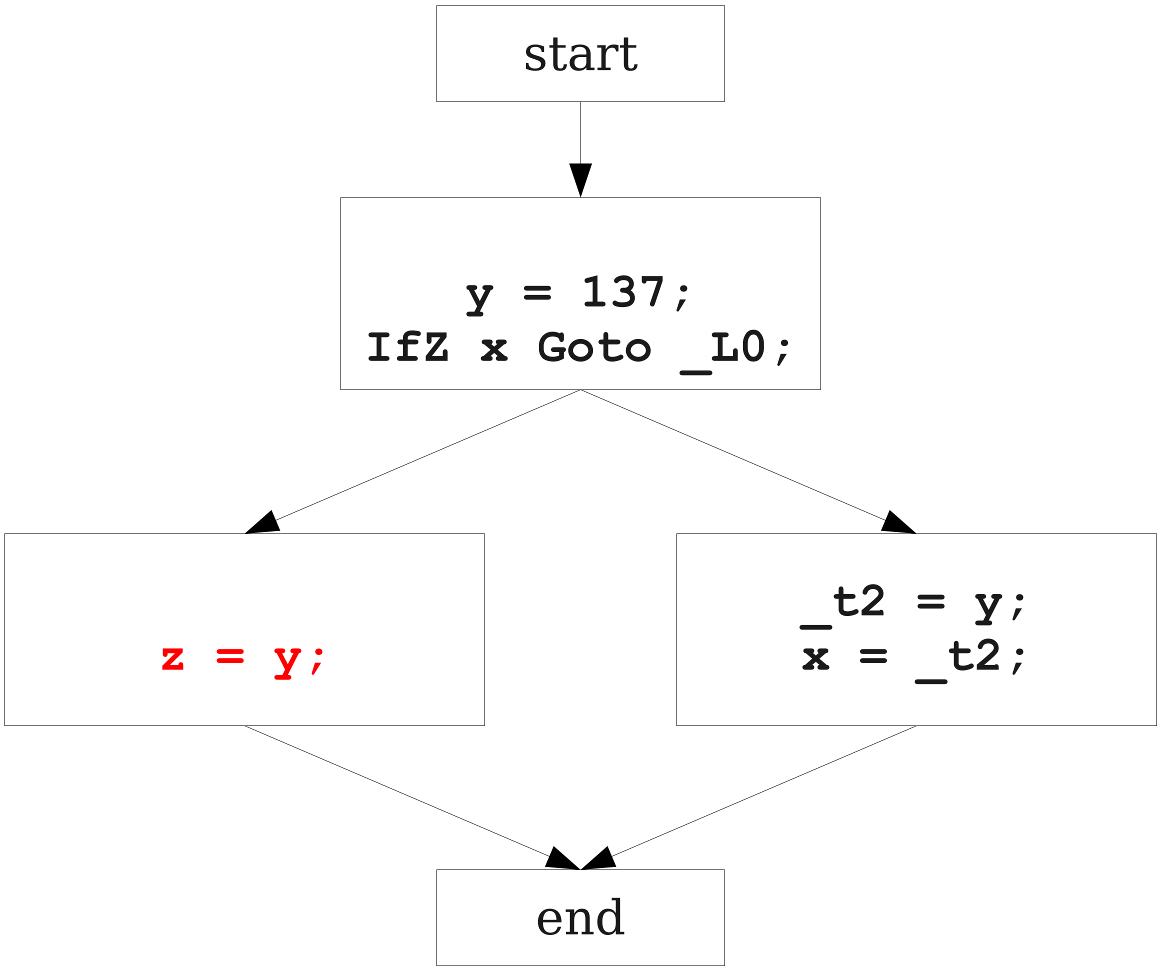
# Local Optimizations

```
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```



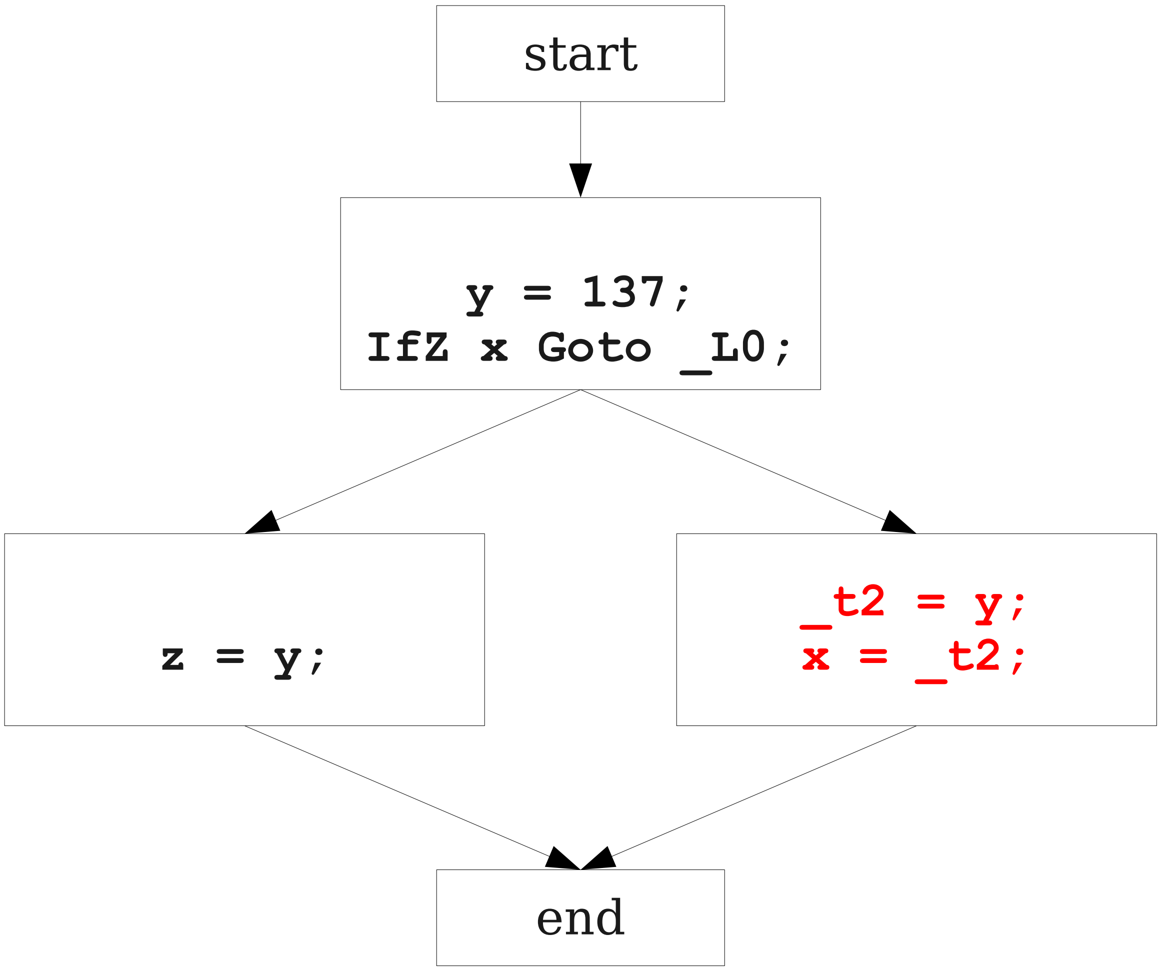
# Local Optimizations

```
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```



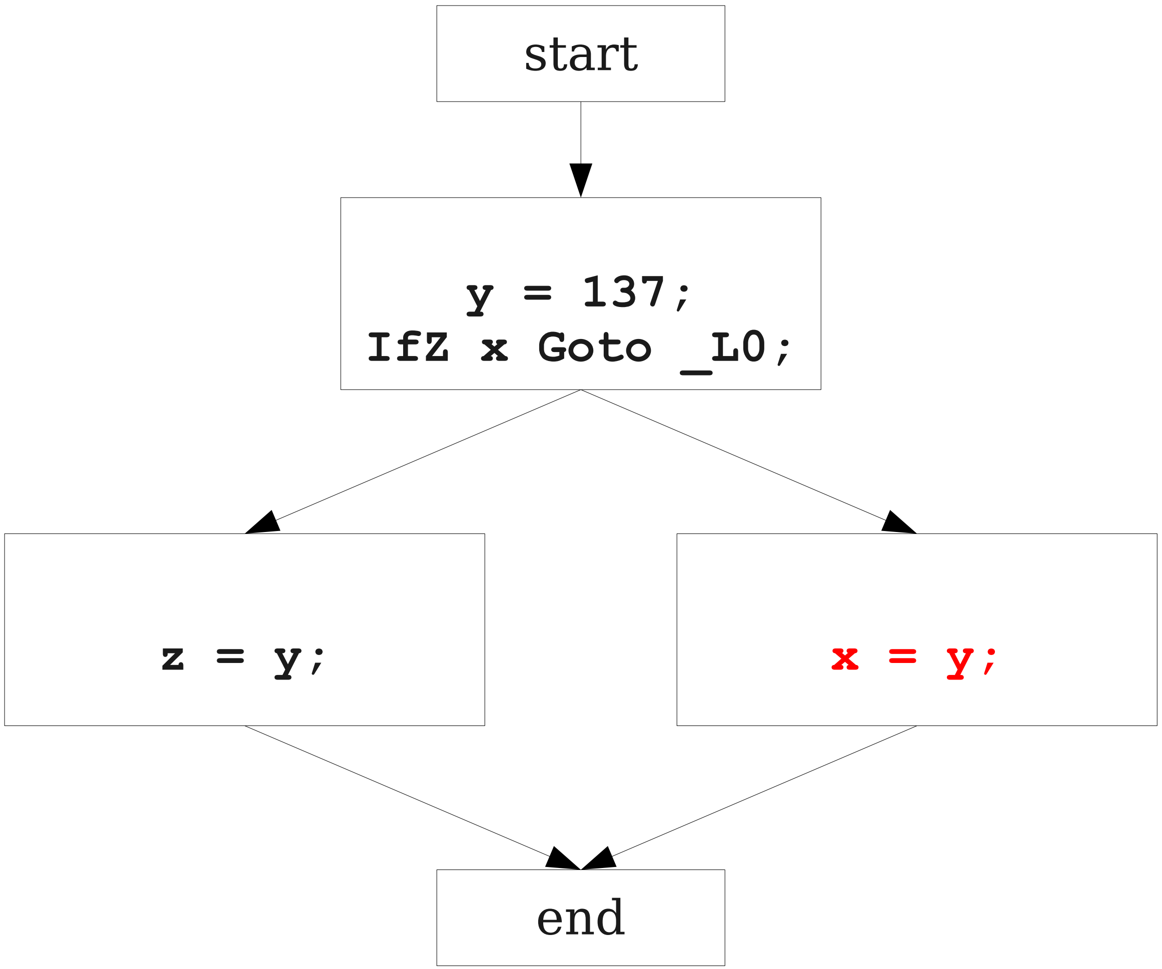
# Local Optimizations

```
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```



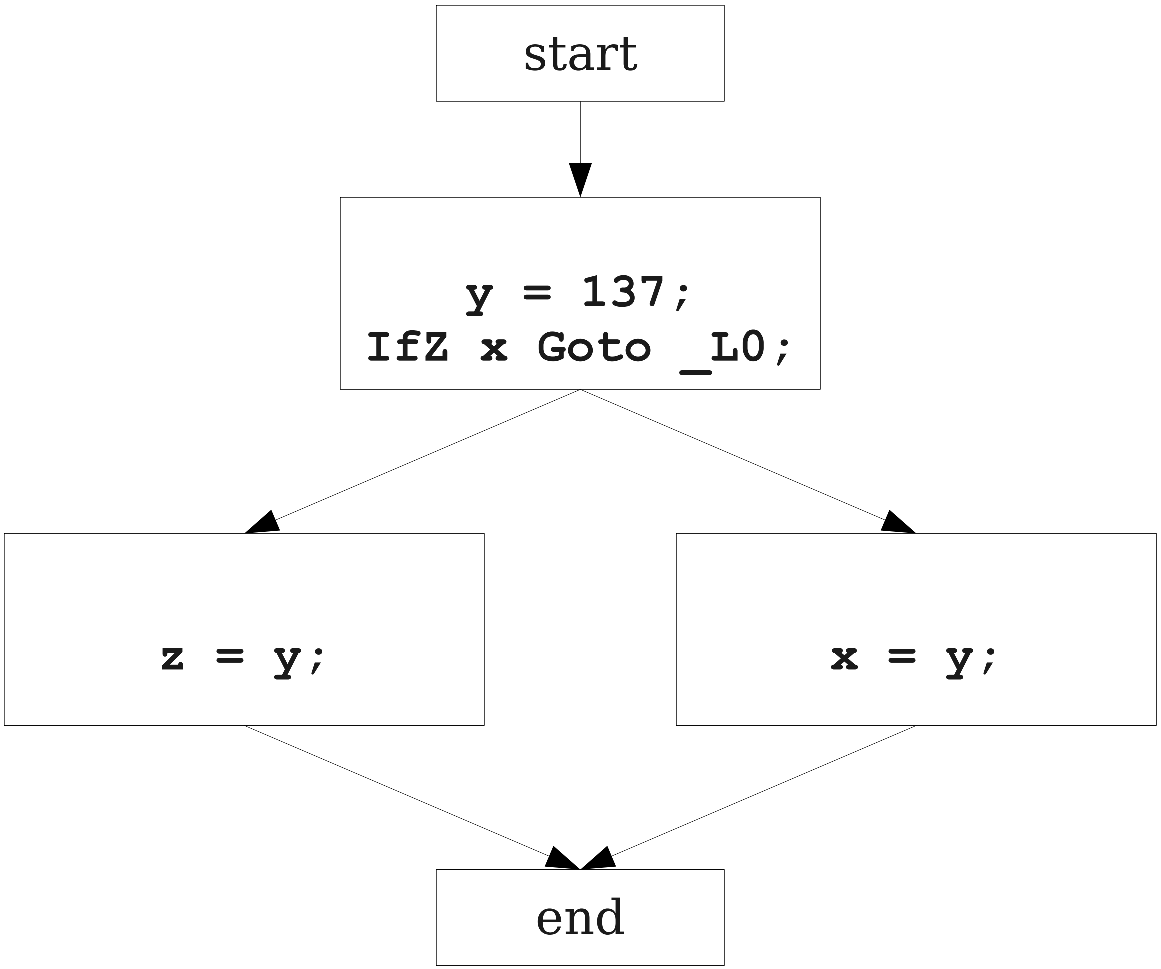
# Local Optimizations

```
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```



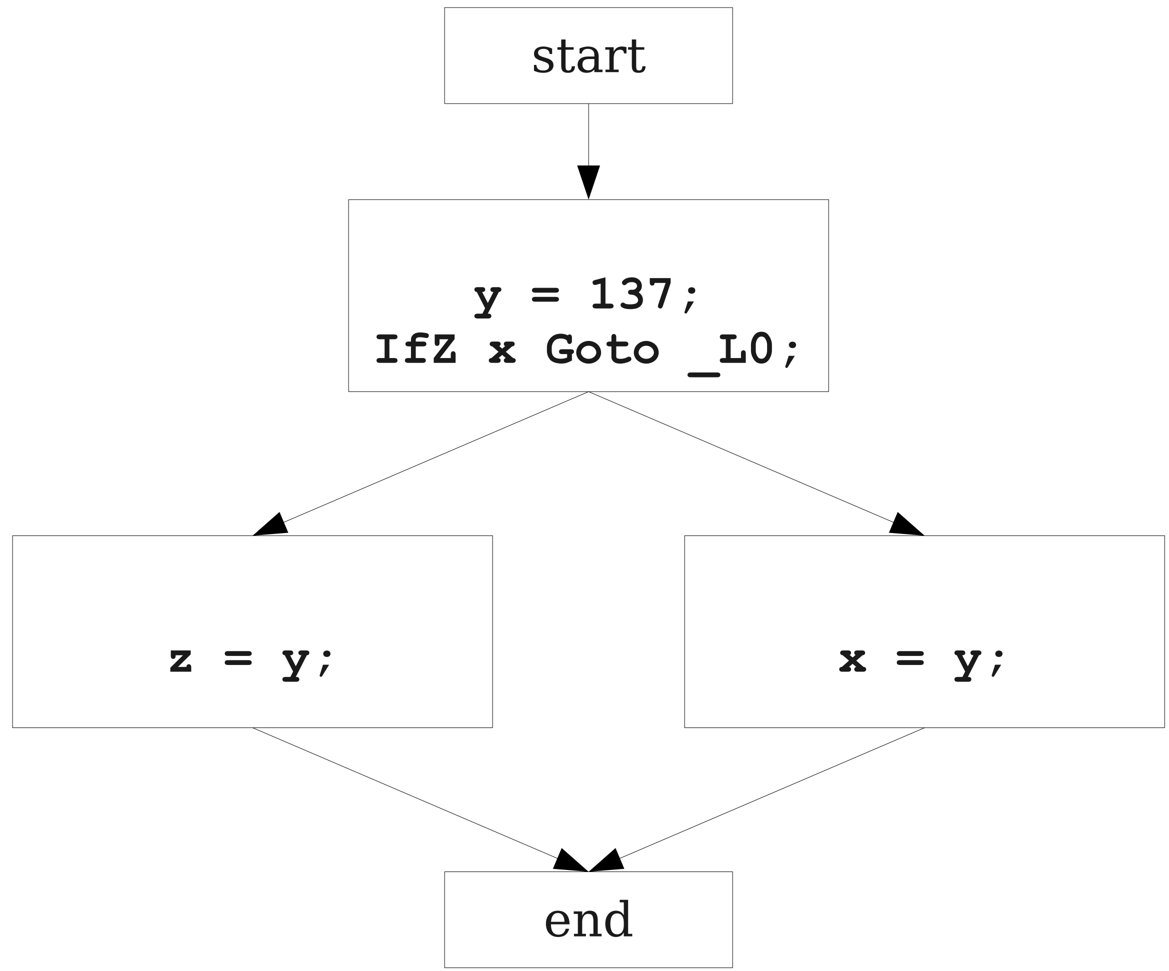
# Local Optimizations

```
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```



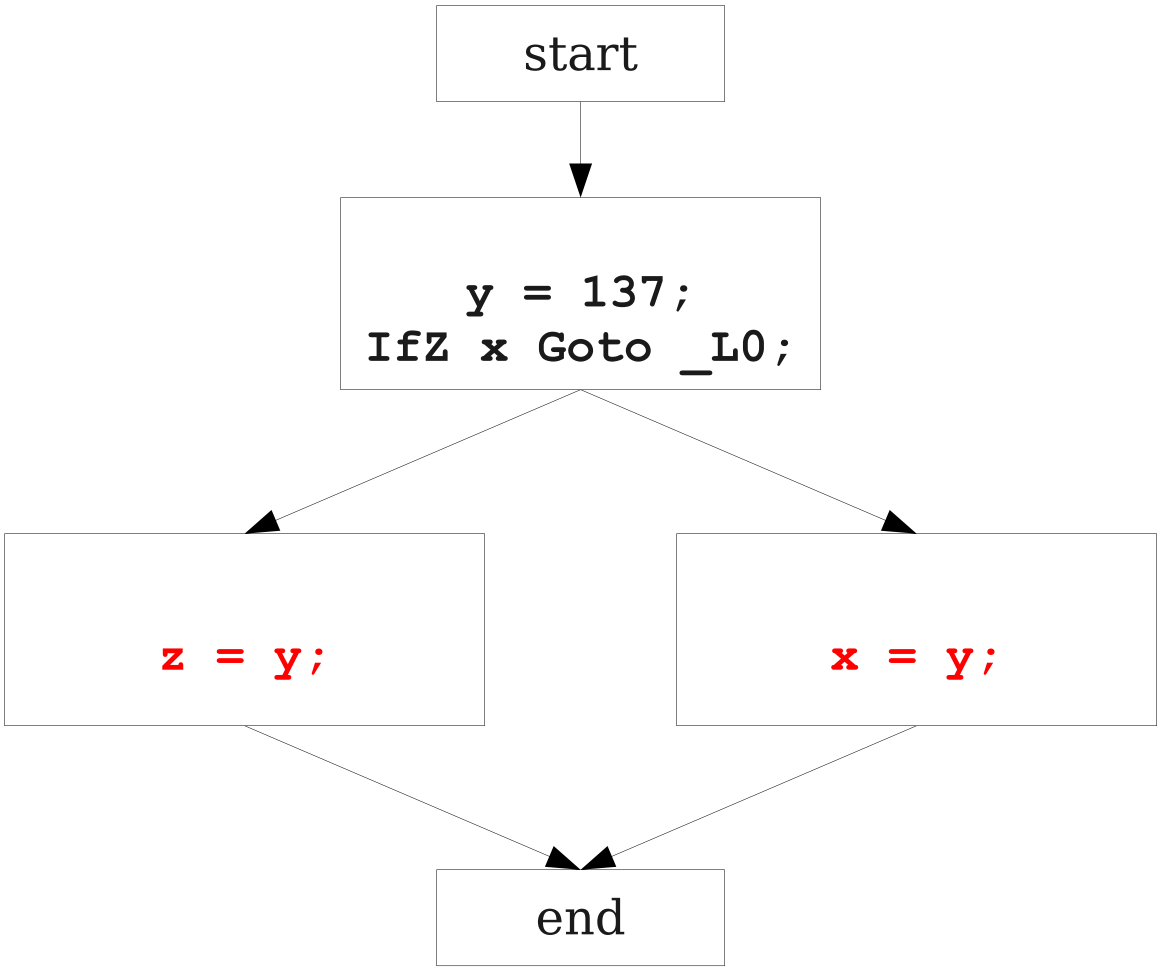
# Global Optimizations

```
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```



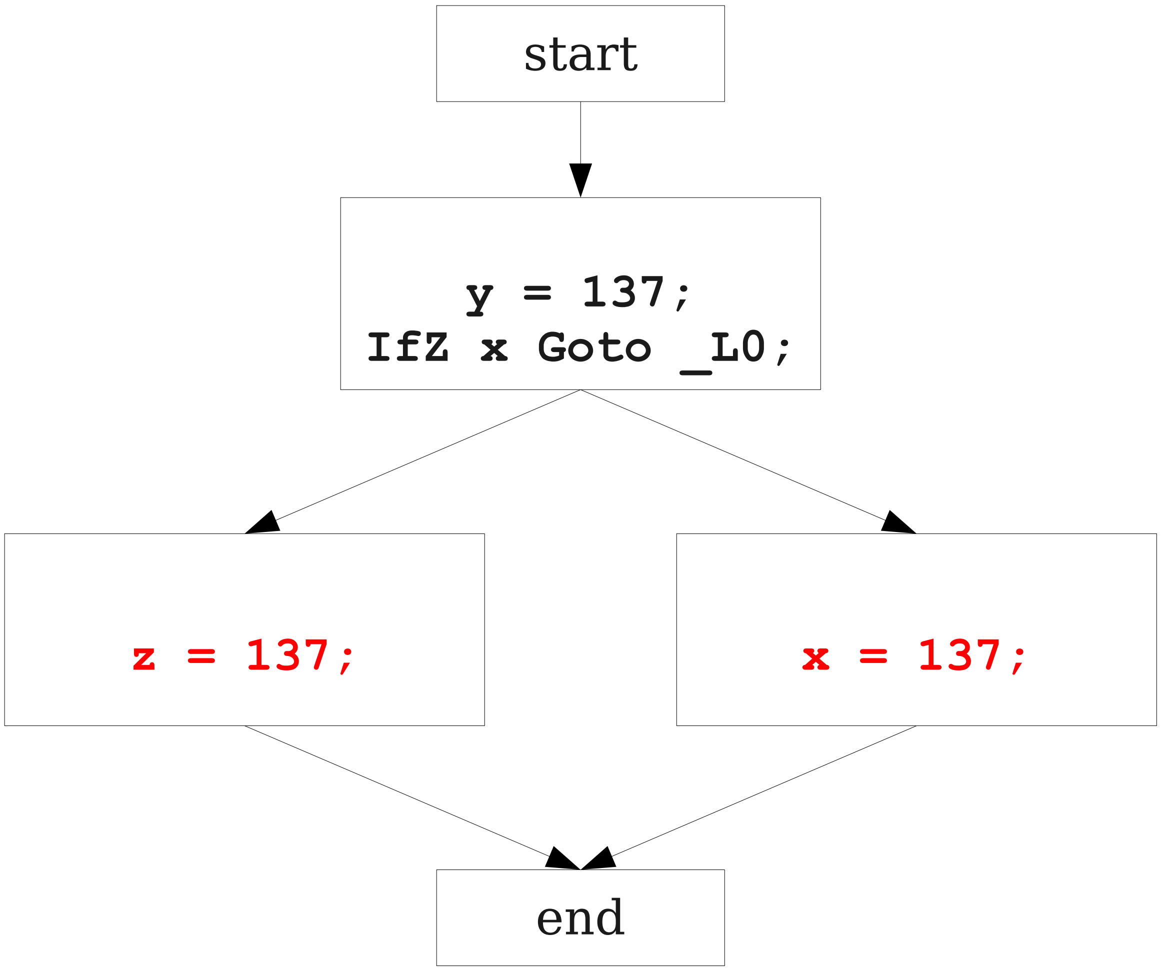
# Global Optimizations

```
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```



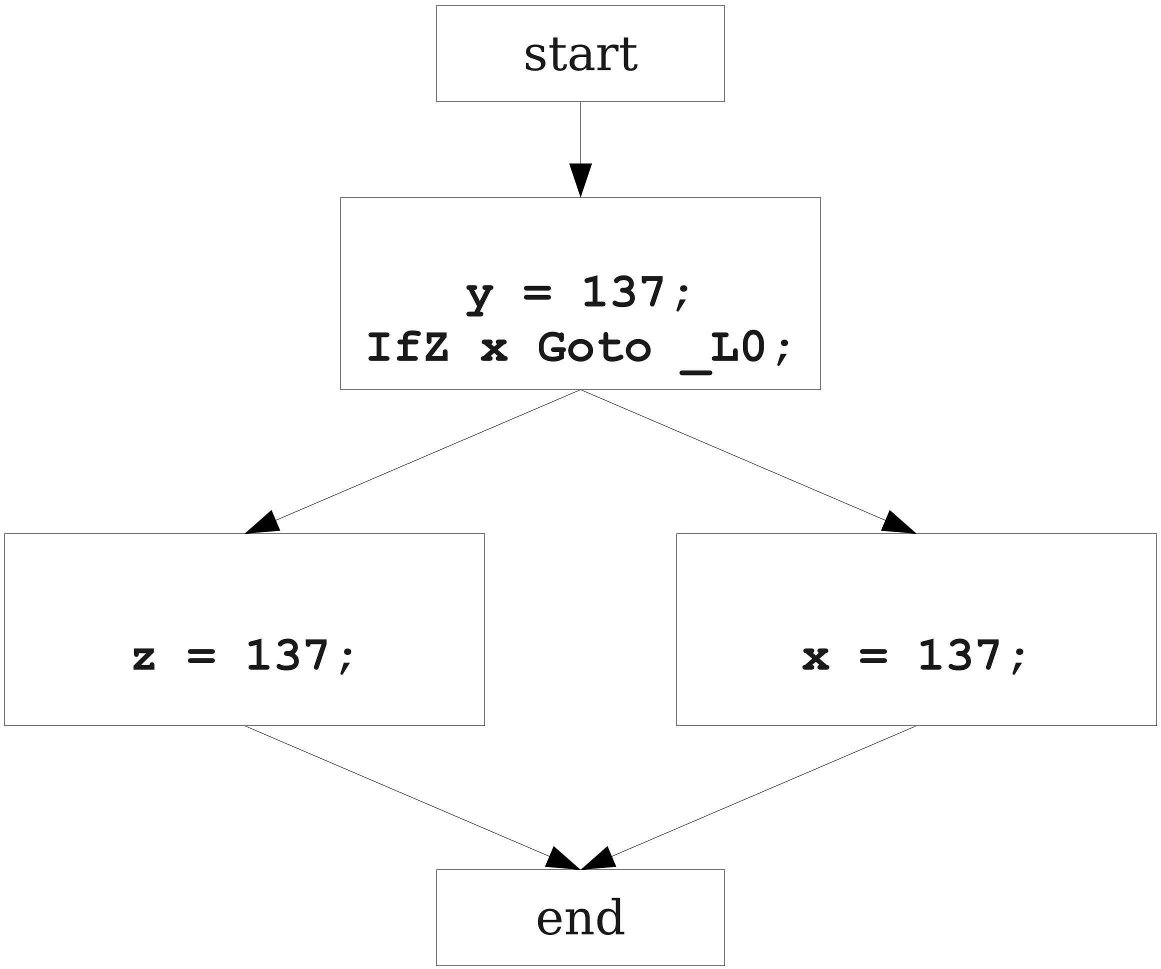
# Global Optimizations

```
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```



# Global Optimizations

```
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```



# Local Optimizations

# Common Subexpression Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

# Common Subexpression Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = a + b ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

# Common Subexpression Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = a + b ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

# Common Subexpression Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

# Common Subexpression Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

# Common Subexpression Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

# Common Subexpression Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

# Common Subexpression Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

# Common Subexpression Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

# Common Subexpression Elimination

- If we have two variable assignments

$$\mathbf{v_1 = a\ op\ b}$$

$$\mathbf{\ldots}$$

$$\mathbf{v_2 = a\ op\ b}$$

  and the values of $\mathbf{v_1}$, **a**, and **b** have not changed between the assignments, rewrite the code as

$$\mathbf{v_1 = a\ op\ b}$$

$$\mathbf{\ldots}$$

$$\mathbf{v_2 = v_1}$$

- Eliminates useless recalculation.
- Paves the way for later optimizations.

# Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

# Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

# Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = *(_tmp1) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

# Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = *(_tmp1) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

# Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = _tmp3 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = *(_tmp1) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

# Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = _tmp3 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = *(_tmp1) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

# Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = _tmp3 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = *(_tmp1) ;
_tmp7 = *(_tmp6) ;
PushParam c ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

# Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = _tmp3 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = *(_tmp1) ;
_tmp7 = *(_tmp6) ;
PushParam c ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

# Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = _tmp3 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp6) ;
PushParam c ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

# Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = _tmp3 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp6) ;
PushParam c ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

# Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = _tmp3 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam c ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

# Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = _tmp3 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam c ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

# Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp0 ;
_tmp4 = _tmp0 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam c ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

# Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp0 ;
_tmp4 = _tmp0 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam c ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

# Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = 4 ;
_tmp4 = _tmp0 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam c ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

# Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = 4 ;
_tmp4 = _tmp0 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam c ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

# Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = 4 ;
_tmp4 = _tmp0 + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

# Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = 4 ;
_tmp4 = _tmp0 + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

# Copy Propagation

- If we have a variable assignment

  $v_1 = v_2$

  then as long as $\mathbf{v_1}$ and $\mathbf{v}_2$ are not reassigned, we can rewrite expressions of the form

  $a = \ldots v_1 \ldots$

  as

  $a = \ldots v_2 \ldots$

  provided that such a rewrite is legal.
- This will help immensely later on, as you'll see.

# Dead Code Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = 4 ;
_tmp4 = _tmp0 + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

# Dead Code Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = 4 ;
_tmp4 = _tmp0 + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

# Dead Code Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;

_tmp3 = 4 ;
a = 4 ;
_tmp4 = _tmp0 + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

# Dead Code Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;

_tmp3 = 4 ;
a = 4 ;
_tmp4 = _tmp0 + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

# Dead Code Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;

a = 4 ;
_tmp4 = _tmp0 + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

# Dead Code Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;

a = 4 ;
_tmp4 = _tmp0 + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

# Dead Code Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;



_tmp4 = _tmp0 + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

# Dead Code Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;



_tmp4 = _tmp0 + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

# Dead Code Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;



_tmp4 = _tmp0 + b ;

_tmp5 = _tmp4 ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

# Dead Code Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;



_tmp4 = _tmp0 + b ;

_tmp5 = _tmp4 ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

# Dead Code Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;



_tmp4 = _tmp0 + b ;


_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

# Dead Code Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;



_tmp4 = _tmp0 + b ;


_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

# Dead Code Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;



_tmp4 = _tmp0 + b ;



_tmp7 = *(_tmp2) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

# Dead Code Elimination

- An assignment to a variable **v** is called **dead** if the value of that assignment is never read anywhere.
- **Dead code elimination** removes dead assignments from IR.
- Determining whether an assignment is dead depends on what variable is being assigned to and when it's being assigned.

# Applying Local Optimizations

- The different optimizations we've seen so far all take care of just a small piece of the optimization.
  - Common subexpression elimination eliminates unnecessary statements.
  - Copy propagation helps identify dead code.
  - Dead code elimination removes statements that are no longer needed.
- To get maximum effect, we may have to apply these optimizations numerous times.

# Applying Local Optimizations

```
b = a * a;
c = a * a;
d = b + c;
e = b + b;
```

# Applying Local Optimizations

```
b = a * a;
c = a * a;
d = b + c;
e = b + b;
```

# Applying Local Optimizations

```
b = a * a;
c = a * a;
d = b + c;
e = b + b;
```

Common Subexpression Elimination

# Applying Local Optimizations

```
b = a * a;
c = b;
d = b + c;
e = b + b;
```

Common Subexpression Elimination

# Applying Local Optimizations

```
b = a * a;
c = b;
d = b + c;
e = b + b;
```

# Applying Local Optimizations

```
b = a * a;
c = b;
d = b + c;
e = b + b;
```

# Applying Local Optimizations

```
b = a * a;
c = b;
d = b + c;
e = b + b;
```

Copy Propagation

# Applying Local Optimizations

```
b = a * a;
c = b;
d = b + b;
e = b + b;
```

Copy Propagation

# Applying Local Optimizations

```
b = a * a;
c = b;
d = b + b;
e = b + b;
```

# Applying Local Optimizations

```
b = a * a;
c = b;
d = b + b;
e = b + b;
```

# Applying Local Optimizations

```
b = a * a;
c = b;
d = b + b;
e = b + b;
```

Common Subexpression Elimination (Again)

# Applying Local Optimizations

```
b = a * a;
c = b;
d = b + b;
e = d;
```

Common Subexpression Elimination (Again)

# Applying Local Optimizations

```
b = a * a;
c = b;
d = b + b;
e = d;
```

# Other Types of Local Optimization

- **Arithmetic Simplification**
  - Replace “hard” operations with easier ones.
  - e.g. rewrite `x = 4 * a;` as `x = a << 2;`
- **Constant Folding**
  - Evaluate expressions at compile-time if they have a constant value.
  - e.g. rewrite **x = 4 * 5;** as **x = 20;**.

# Optimizations and Analyses

- Most optimizations are only possible given some analysis of the program's behavior.
- In order to implement an optimization, we will talk about the corresponding program analyses.

# Available Expressions

- Both common subexpression elimination and copy propagation depend on an analysis of the **available expressions** in a program.
- An expression is called **available** if some variable in the program holds the value of that expression.
- In common subexpression elimination, we replace an available expression by the variable holding its value.
- In copy propagation, we replace the use of a variable by the available expression it holds.

# Finding Available Expressions

- Initially, no expressions are available.
- Whenever we execute a statement **a = b + c**:
  - Any expression holding **a** is invalidated.
  - The expression **a = b + c** becomes available.
- **Idea**: Iterate across the basic block, beginning with the empty set of expressions and updating available expressions at each variable.

# Live Variables

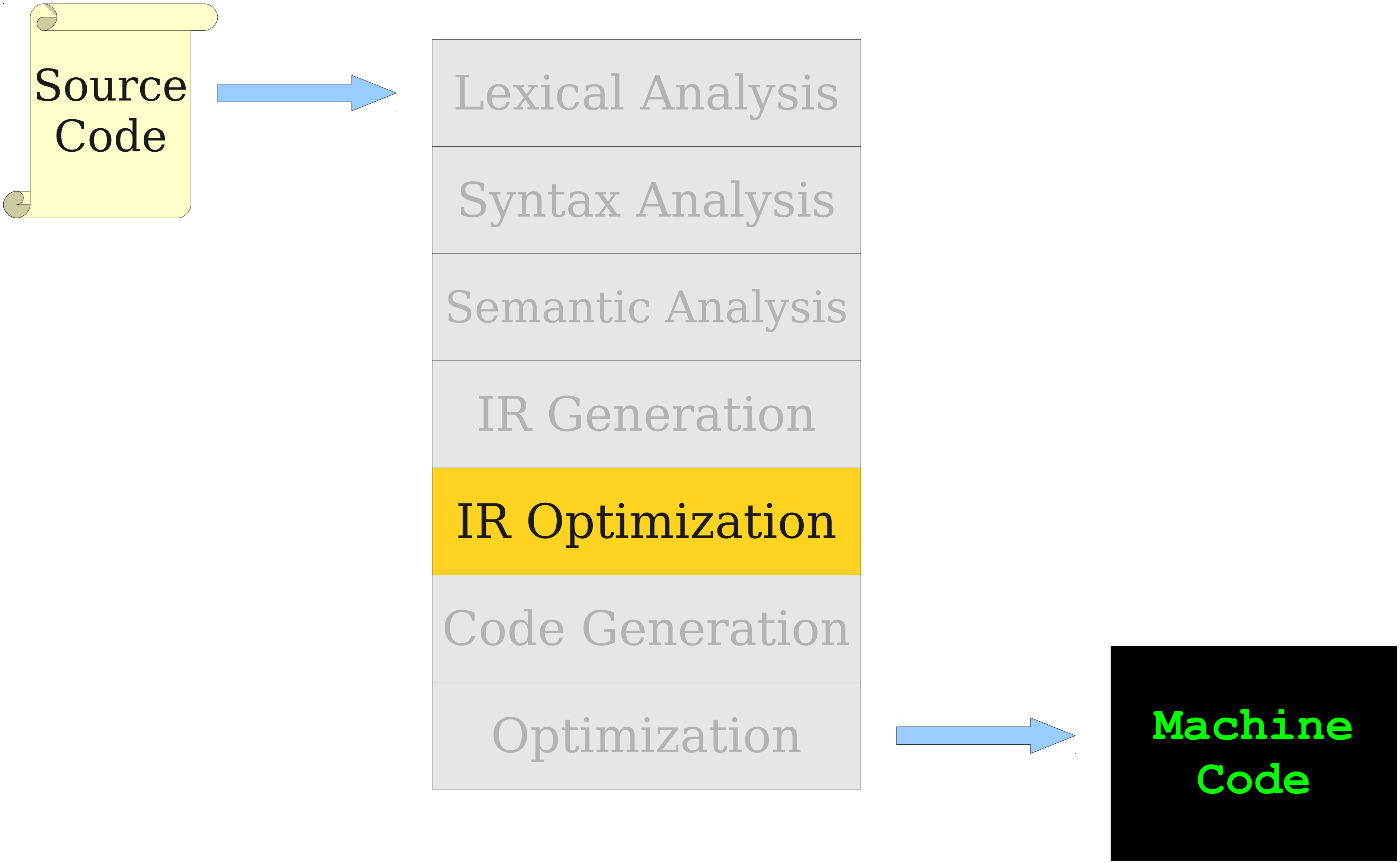
- The analysis corresponding to dead code elimination is called **liveness analysis**.
- A variable is **live** at a point in a program if later in the program its value will be read before it is written to again.
- Dead code elimination works by computing liveness for each variable, then eliminating assignments to dead variables.

# Computing Live Variables

- To know if a variable will be used at some point, we iterate across the statements in a basic block in reverse order.
- Initially, some small set of values are known to be live (which ones depends on the particular program).
- When we see the statement **a = b + c**:
  - Just before the statement, **a** is not alive, since its value is about to be overwritten.
  - Just before the statement, both **b** and **c** are alive, since we're about to read their values.
  - *(what if we have* **a = a + b***?)*

# Global Optimization

# Where We Are


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

# Global Analysis

- A **global analysis** is an analysis that works on a control-flow graph as a whole.
- Substantially more powerful than a local analysis.
  - (Why?)
- Substantially more complicated than a local analysis.
  - (Why?)

# Local vs. Global Analysis

- Many of the optimizations from local analysis can still be applied globally.
  - We'll see how to do this later today.
- Certain optimizations are possible in global analysis that aren't possible locally:
  - e.g. **code motion:** Moving code from one basic block into another to avoid computing values unnecessarily.

# Major Changes, Part One

- In a local analysis, each statement has exactly one predecessor.
- In a global analysis, each statement may have **multiple** predecessors.
- A global analysis must have some means of combining information from all predecessors of a basic block.

# Major Changes, Part II

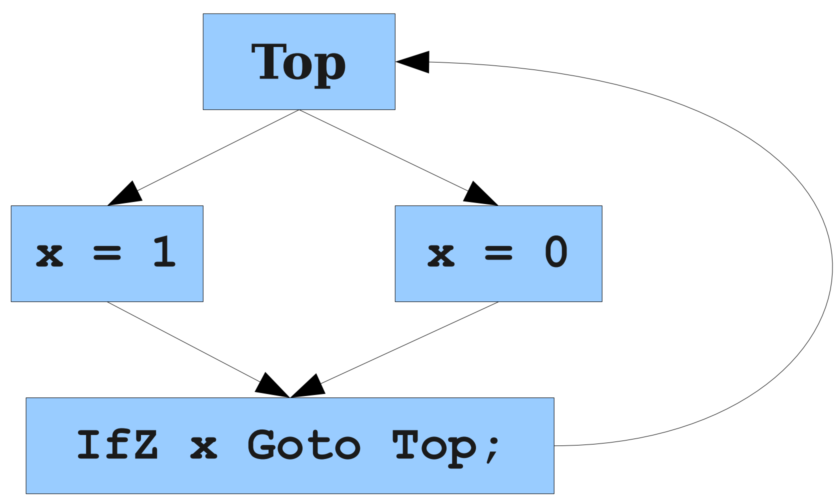
- In a local analysis, there is only one possible path through a basic block.
- In a global analysis, there may be **many** paths through a CFG.
- May need to recompute values multiple times as more information becomes available.
- Need to be careful when doing this not to loop infinitely!
  - (More on that later)

# CFGs with Loops

- Up to this point, we've considered loop-free CFGs, which have only finitely many possible paths.
- When we add loops into the picture, this is no longer true.
- Not all possible loops in a CFG can be realized in the actual program.

# CFGs with Loops

- Up to this point, we've considered loop-free CFGs, which have only finitely many possible paths.
- When we add loops into the picture, this is no longer true.
- Not all possible loops in a CFG can be realized in the actual program.

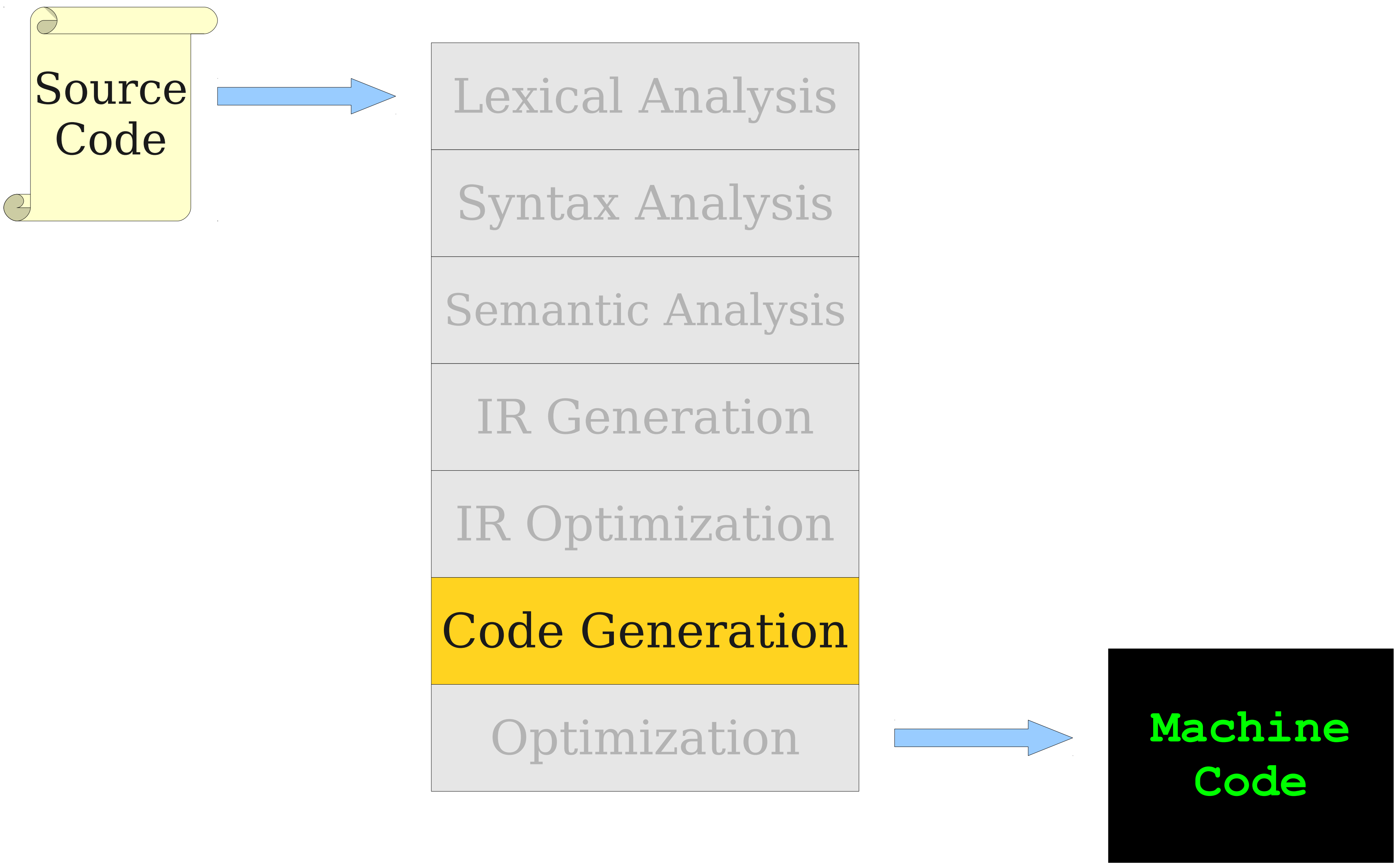


# Major Changes, Part III

- In a local analysis, there is always a well-defined “first” statement to begin processing.
- In a global analysis with loops, every basic block might depend on every other basic block.
- To fix this, we need to assign initial values to all of the blocks in the CFG.

# Register Allocation

# Where We Are


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

# Code Generation at a Glance

- At this point, we have optimized IR code that needs to be converted into the target language (e.g. assembly, machine code).
- Goal of this stage:
  - Choose the appropriate machine instructions for each IR instruction.
  - Divvy up finite machine resources (registers, caches, etc.)
  - Implement low-level details of the runtime environment.
- Machine-specific optimizations are often done here, though some are treated as part of a final optimization phase.

# Memory Tradeoffs

- There is an enormous tradeoff between *speed* and *size* in memory.
- SRAM is fast but very expensive:
  - Can keep up with processor speeds in the GHz.
  - As of 2007, cost is $10/MB
  - Good luck buying 1TB of the stuff!
- Hard disks are cheap but very slow:
  - As of 2012, you can buy a 2TB hard drive for about $100
  - As of 2012, good disk seek times are measured in ms (about two to four million times slower than a processor cycle!)

# Registers

- Most machines have a set of **registers**, dedicated memory locations that
  - can be accessed quickly,
  - can have computations performed on them, and
  - exist in small quantity.
- Using registers intelligently is a critical step in any compiler.
  - A good register allocator can generate code orders of magnitude better than a bad register allocator.

# Register Allocation

- In TAC, there are an unlimited number of variables.
- On a physical machine there are a small number of registers:
  - x86 has four general-purpose registers and a number of specialized registers.
  - MIPS has twenty-four general-purpose registers and eight special-purpose registers.
- **Register allocation** is the process of assigning variables to registers and managing data transfer in and out of registers.

# Challenges in Register Allocation

- **Registers are scarce**.
  - Often substantially more IR variables than registers.
  - Need to find a way to reuse registers whenever possible.
- **Registers are complicated**.
  - x86: Each register made of several smaller registers; can't use a register and its constituent registers at the same time.
  - x86: Certain instructions must store their results in specific registers; can't store values there if you want to use those instructions.
  - MIPS: Some registers reserved for the assembler or operating system.
  - Most architectures: Some registers must be preserved across function calls.

# An Initial Register Allocator

- **Idea**: Store every value in main memory, loading values only when they're needed.
- To generate a code that performs a computation:
  - Generate **load** instructions to pull the values from main memory into registers.
  - Generate code to perform the computation on the registers.
  - Generate **store** instructions to store the result back into main memory.

# Analysis of our Allocator

- Disadvantage: **Gross inefficiency**.
  - Issues unnecessary loads and stores by the dozen.
  - Wastes space on values that could be stored purely in registers.
  - Easily an order of magnitude or two slower than necessary.
  - Unacceptable in any production compiler.
- Advantage: **Simplicity**.
  - Can translate each piece of IR directly to assembly as we go.
  - Never need to worry about running out of registers.
  - Never need to worry about function calls or special-purpose registers.
  - Good if you just needed to get a prototype compiler up and running.

# Building a Better Allocator

- **Goal**: Try to hold as many variables in registers as possible.
  - Reduces memory reads/writes.
  - Reduces total memory usage.
- We will need to address these questions:
  - Which registers do we put variables in?
  - What do we do when we run out of registers?

# Live Ranges and Live Intervals

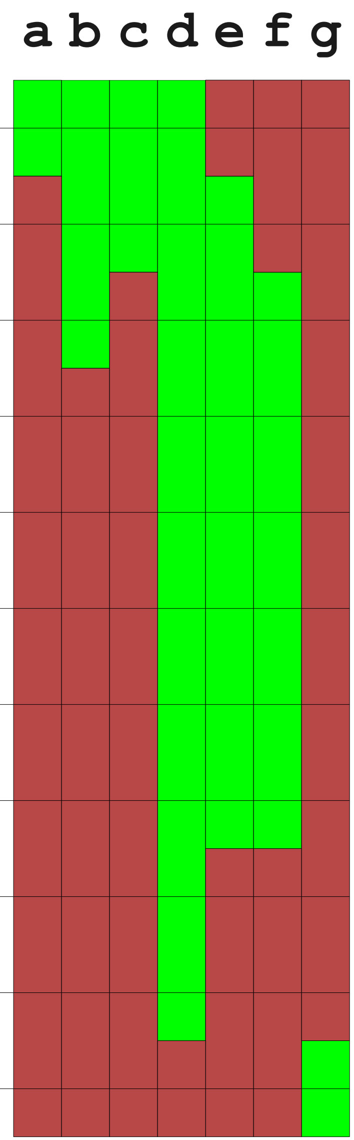
- Recall: A variable is **live** at a particular program point if its value may be read later before it is written.
  - Can find this using global liveness analysis.
- The **live range** for a variable is the set of program points at which that variable is live.
- The **live interval** for a variable is the smallest subrange of the IR code containing all a variable's live ranges.
  - A property of the IR code, **not** the CFG.
  - Less precise than live ranges, but simpler to work with.

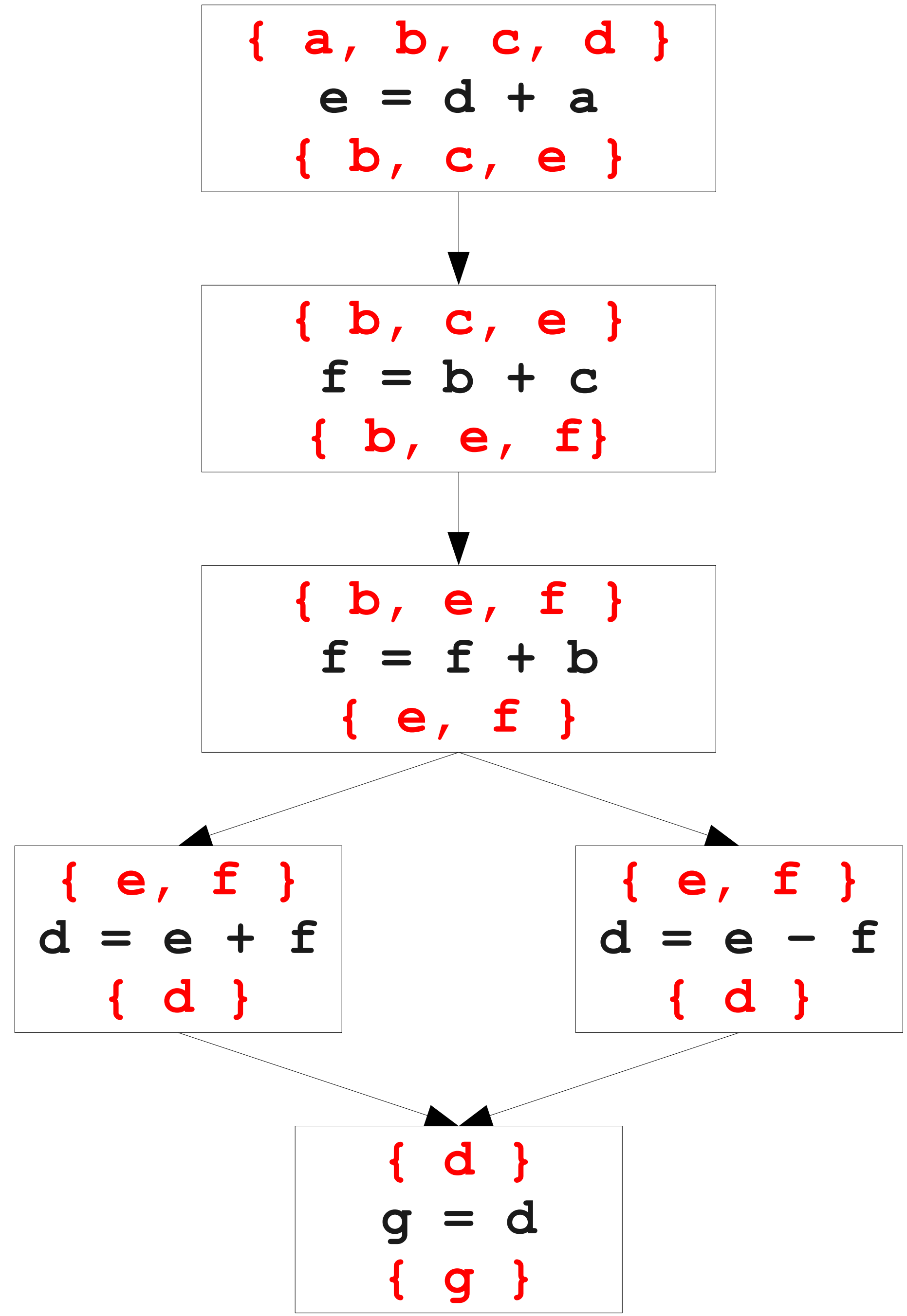
# Live Ranges and Live Intervals

```
    e = d + a
    f = b + c
    f = f + b
    IfZ e Goto _L0
    d = e + f
    Goto _L1;
_L0:
    d = e - f
_L1:
    g = d
```

# Live Ranges and Live Intervals

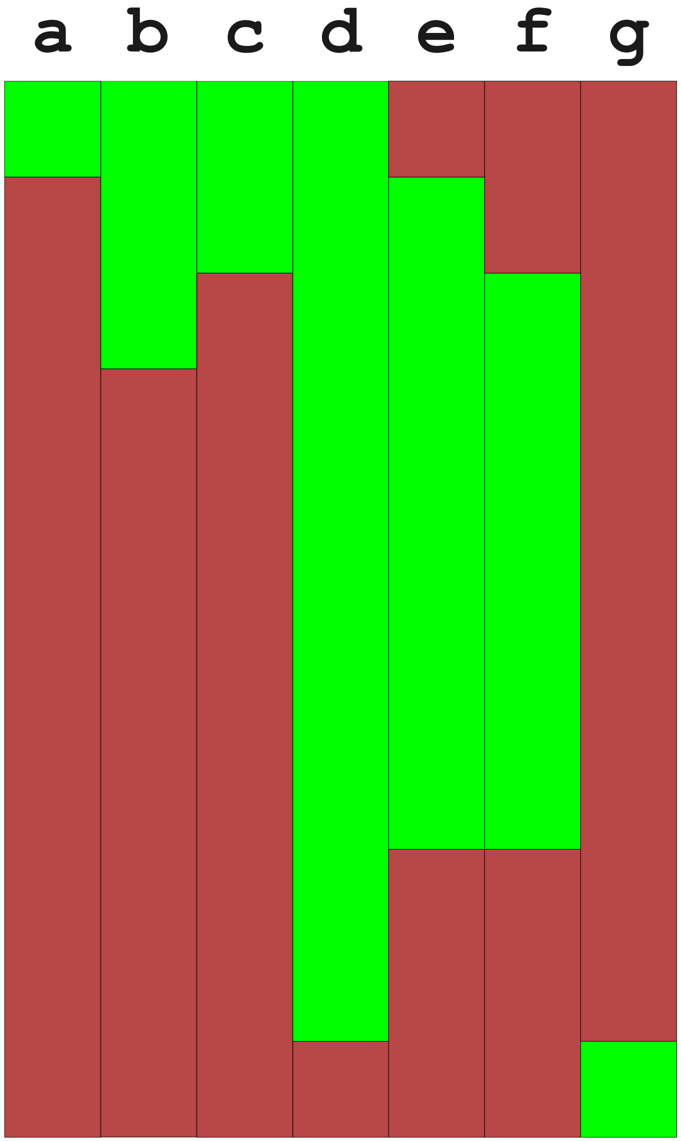
```
    e = d + a
    f = b + c
    f = f + b
    IfZ e Goto _L0
    d = e + f
    Goto _L1;
_L0:
    d = e - f
_L1:
    g = d
```



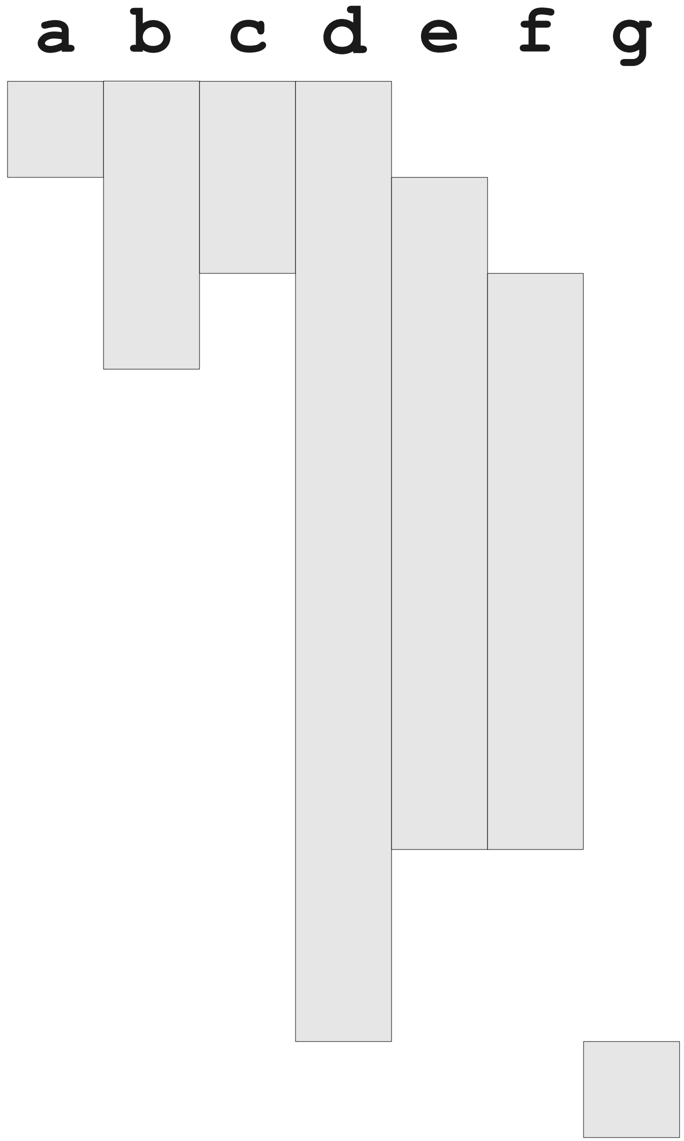



# Register Allocation with Live Intervals

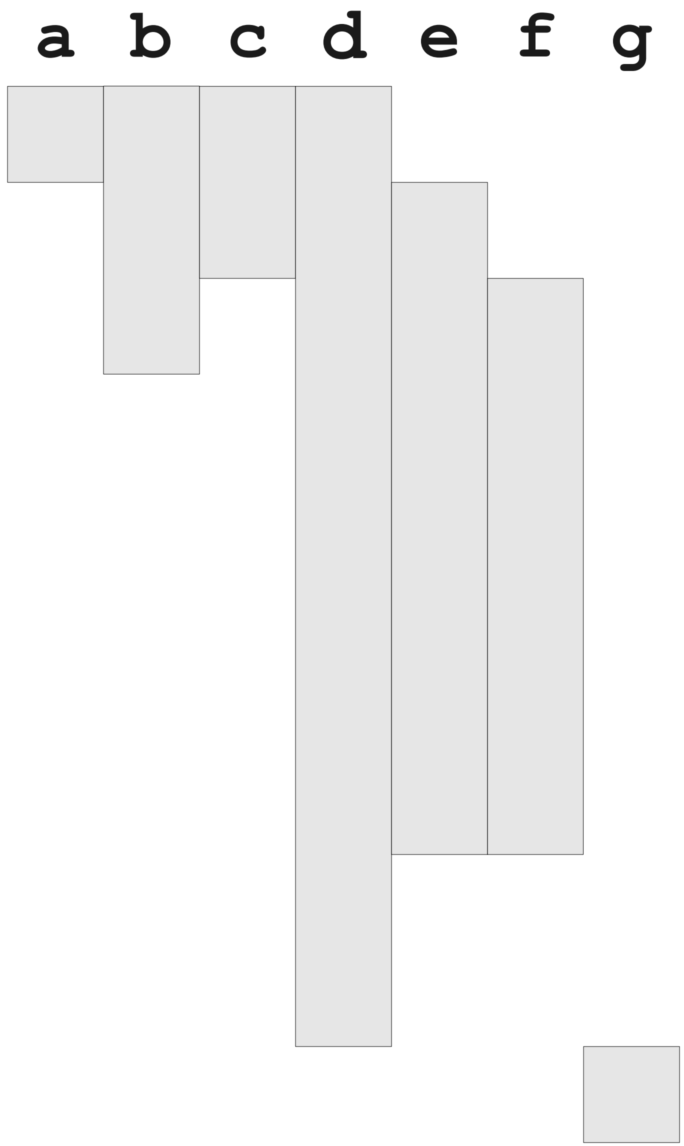
- Given the live intervals for all the variables in the program, we can allocate registers using a simple greedy algorithm.
- Idea: Track which registers are free at each point.
- When a live interval begins, give that variable a free register.
- When a live interval ends, the register is once again free.
- We can't always fit everything into a register; we'll see what do to in a minute.



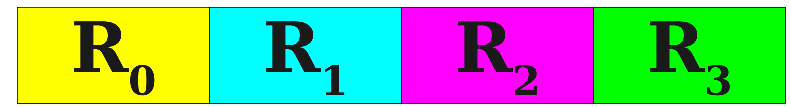
# Register Allocation with Live Intervals



# Register Allocation with Live Intervals

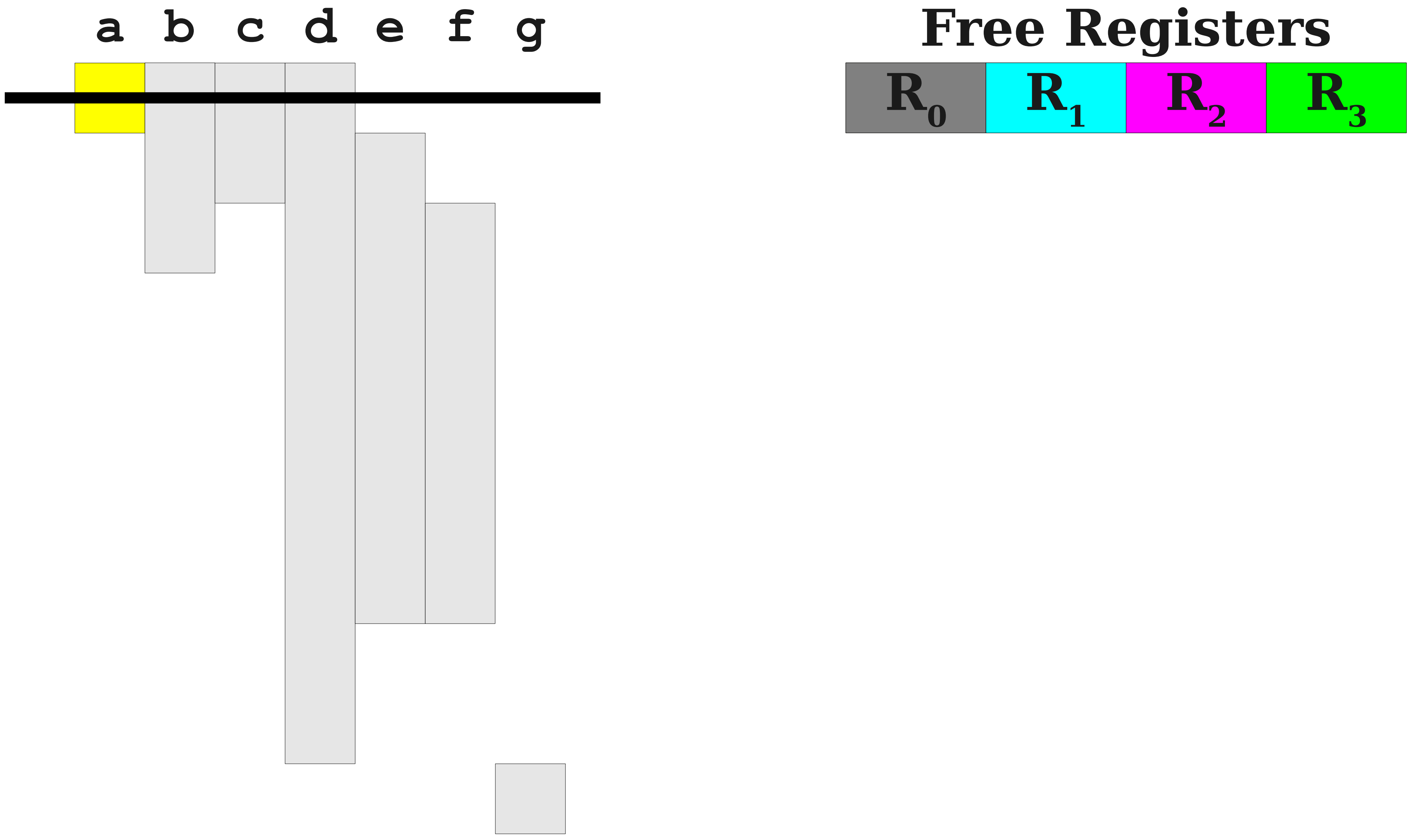



Free Registers

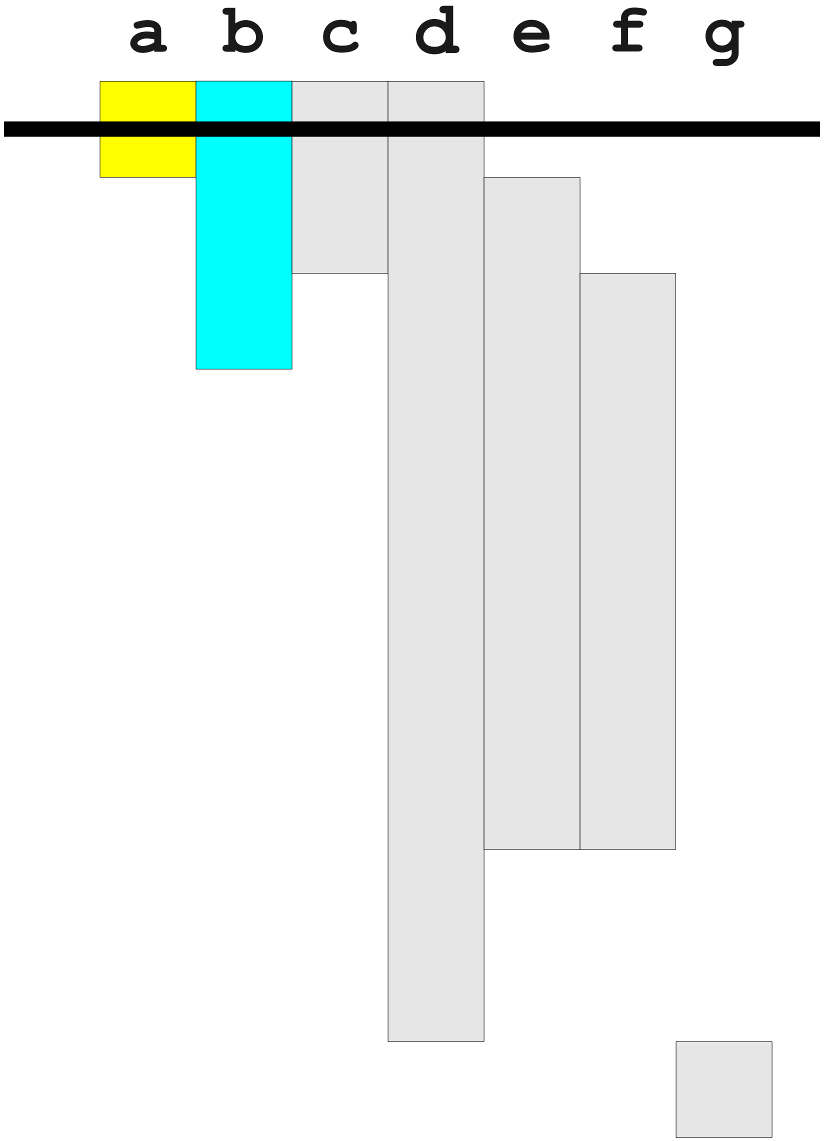


# Register Allocation with Live Intervals

# Register Allocation with Live Intervals



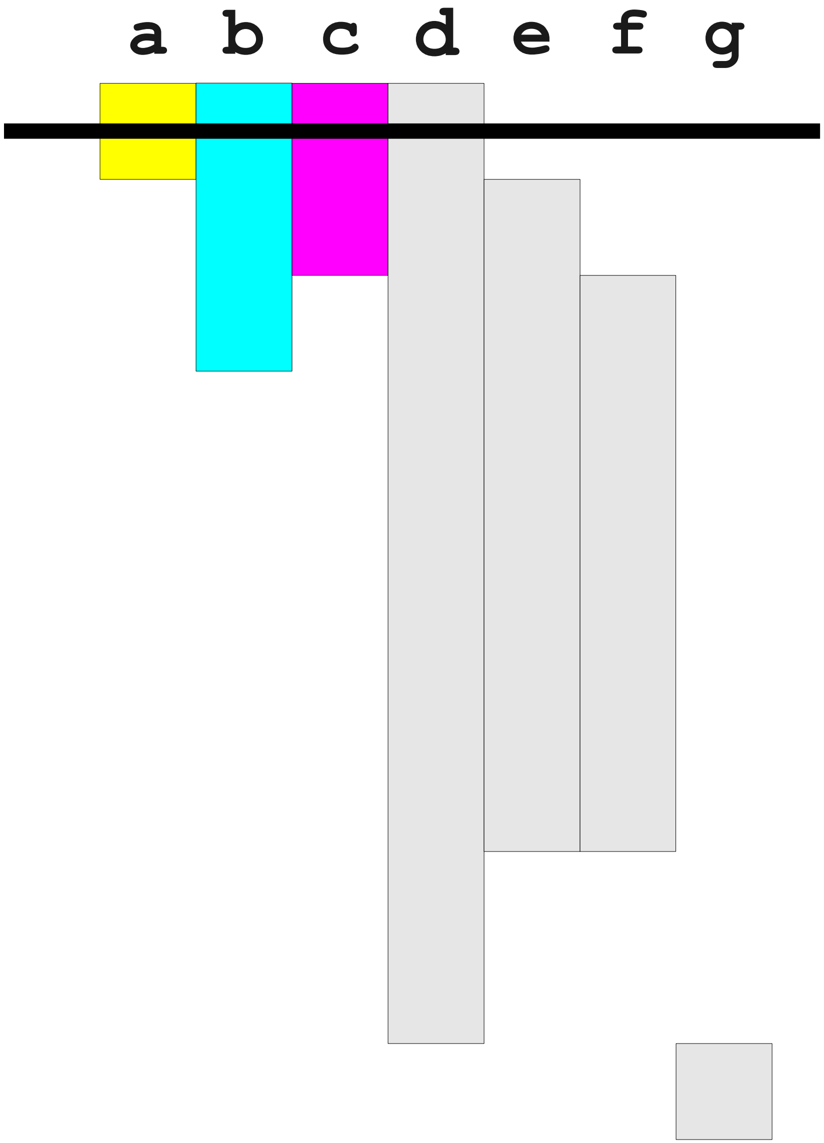
# Register Allocation with Live Intervals



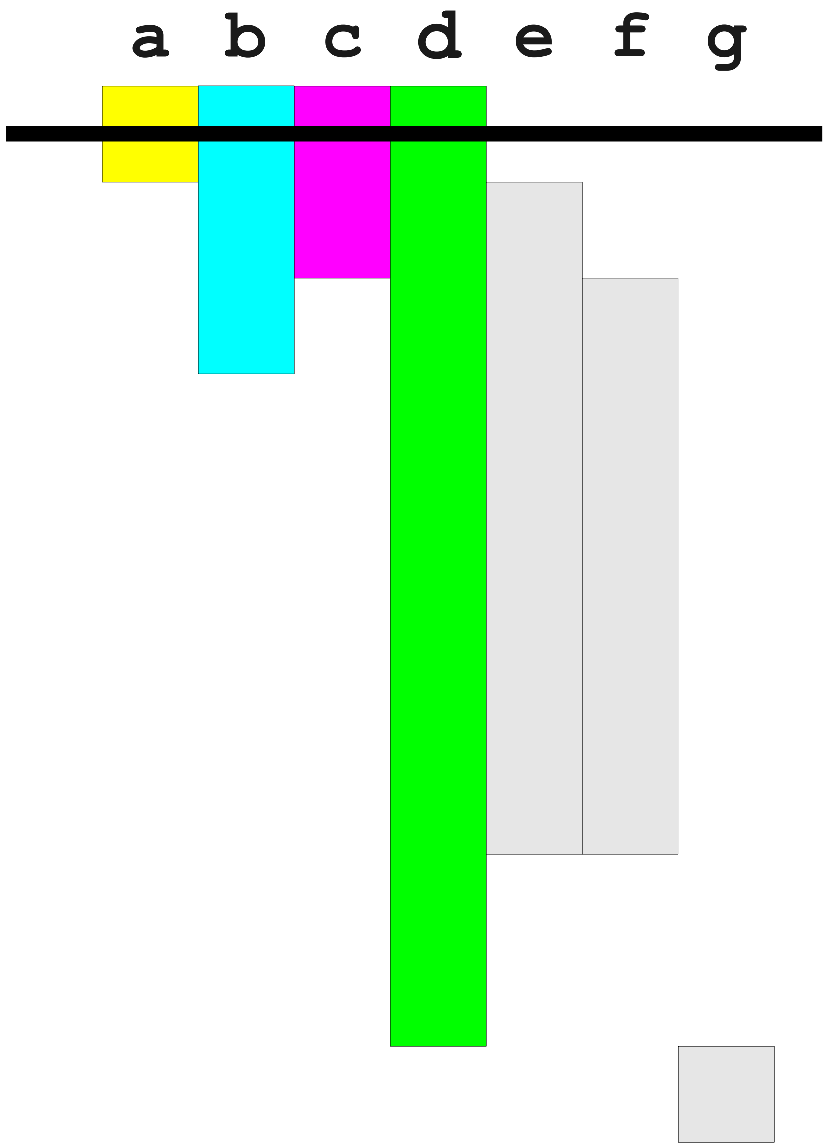

**Free Registers**

| $R_0$ | $R_1$ | $R_2$ | $R_3$ |
|---|---|---|---|

# Register Allocation with Live Intervals



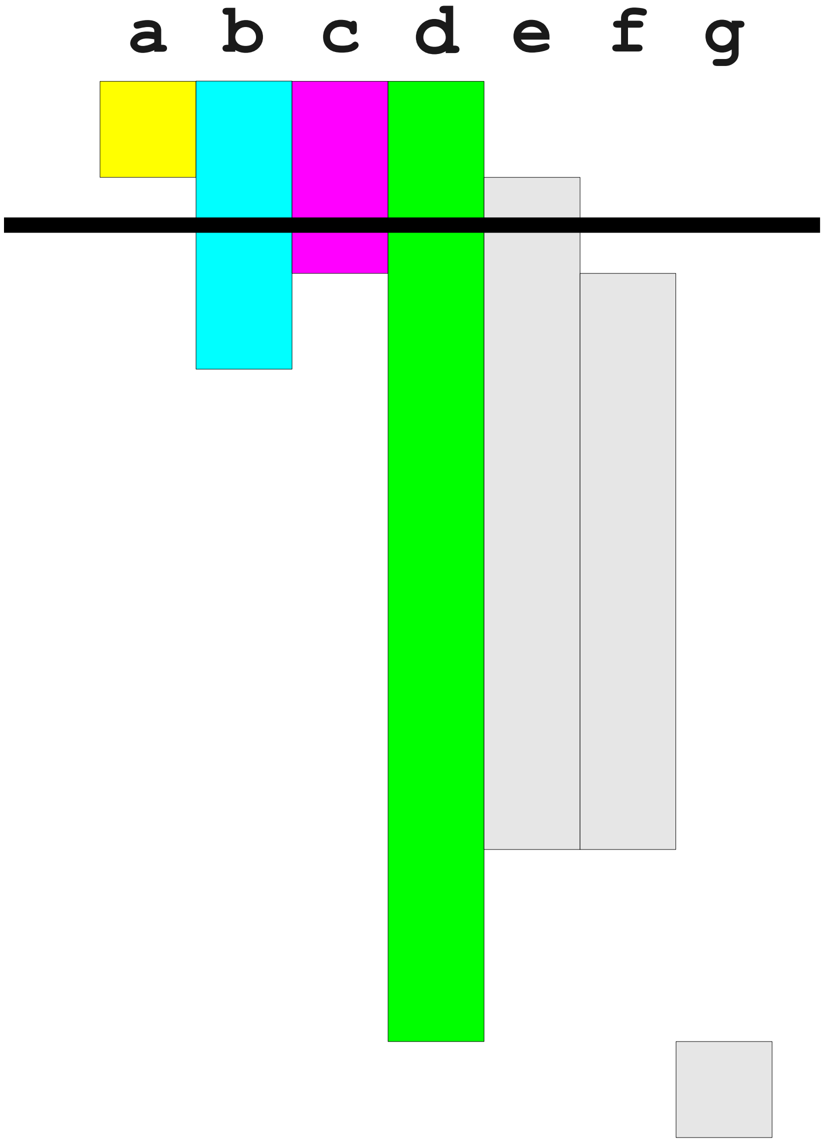

# Register Allocation with Live Intervals




**Free Registers**

| $R_0$ | $R_1$ | $R_2$ | $R_2$ |
|---|---|---|---|

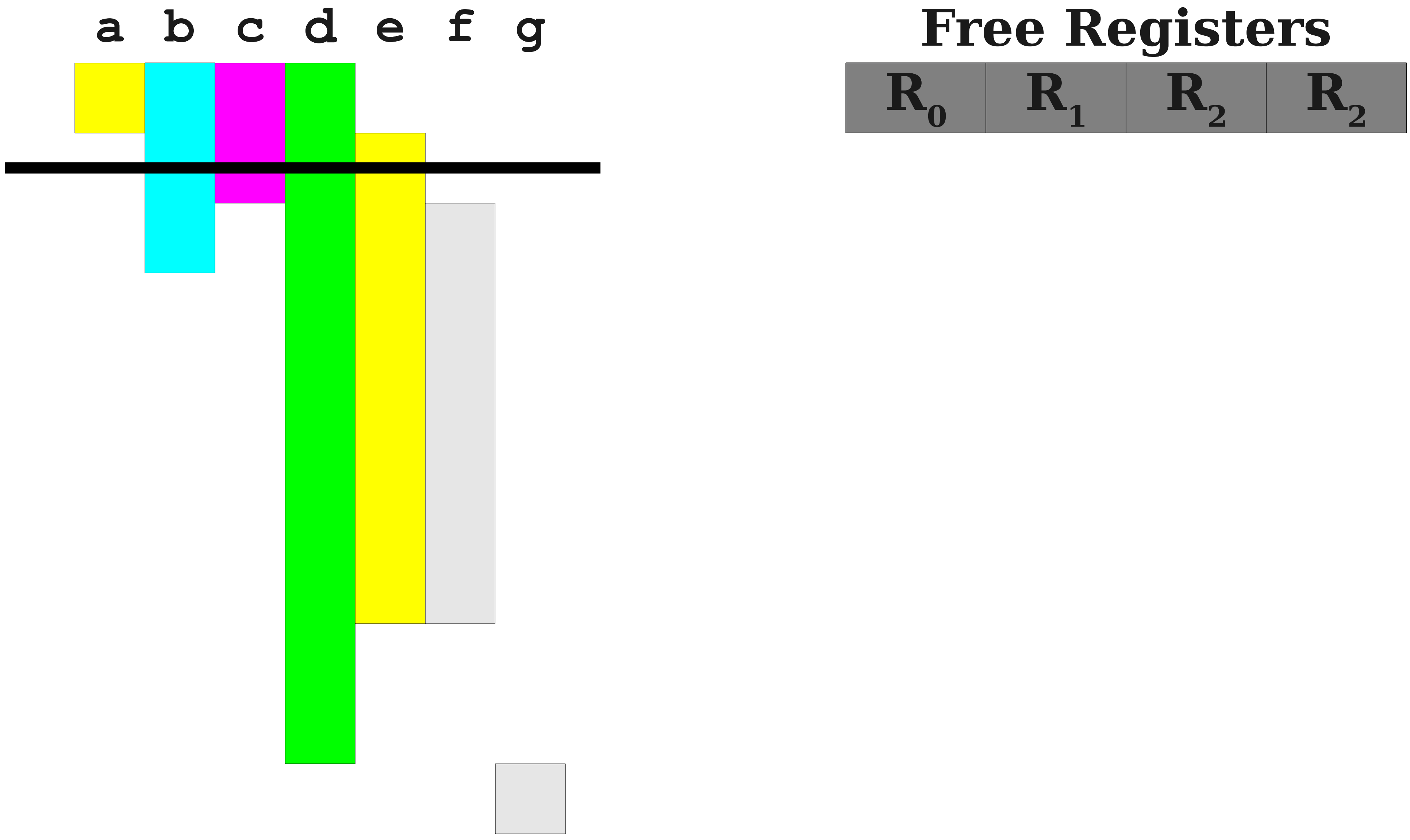
# Register Allocation with Live Intervals



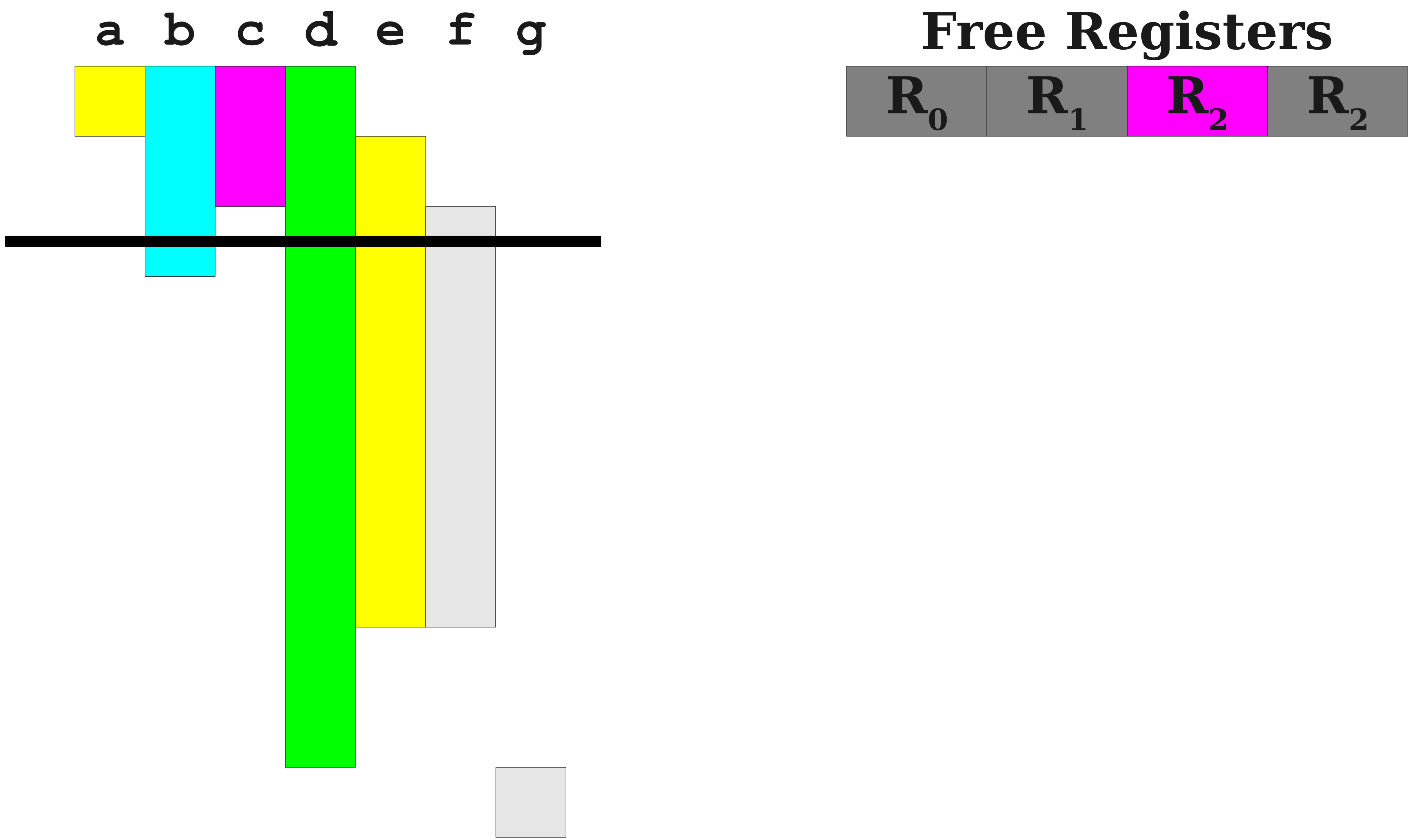

**Free Registers**

| | | | |
|---|---|---|---|
| $R_0$ | $R_1$ | $R_2$ | $R_2$ |

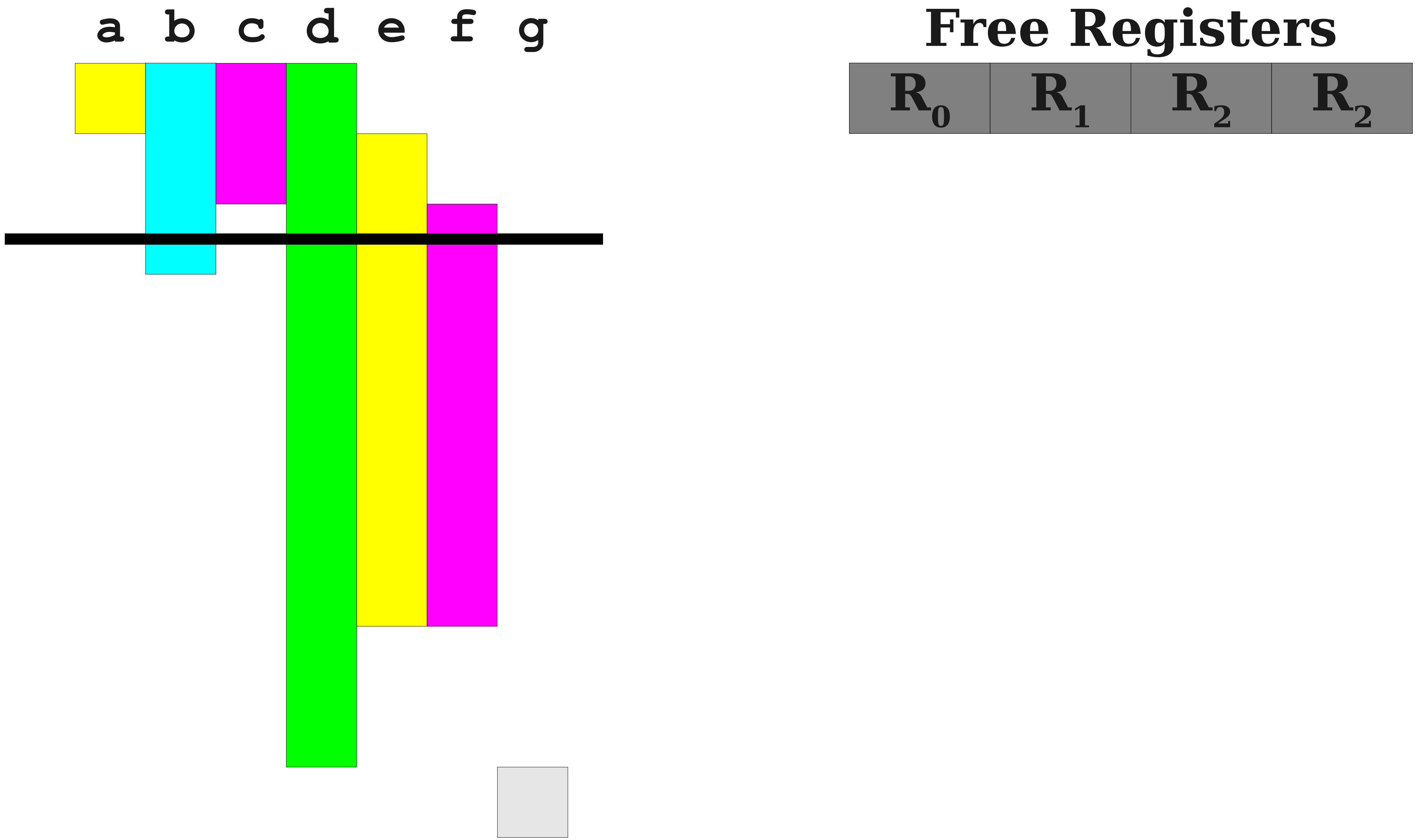
# Register Allocation with Live Intervals



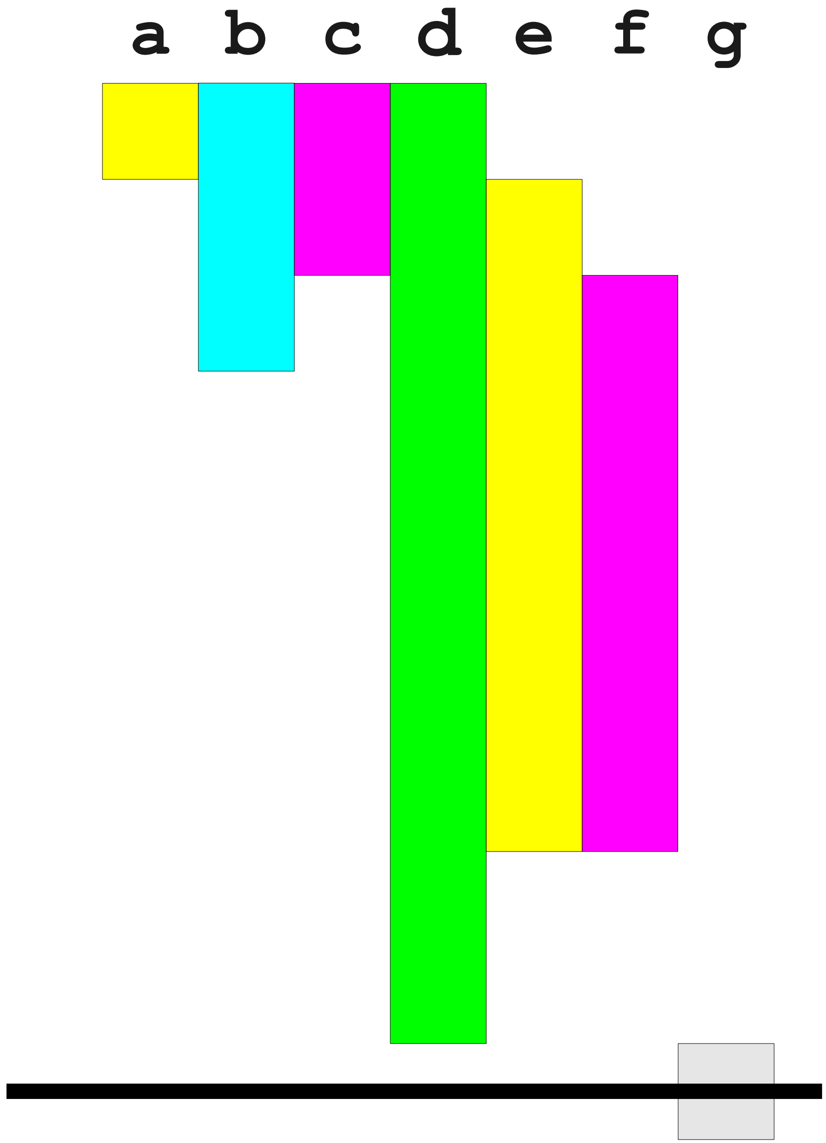
# Register Allocation with Live Intervals



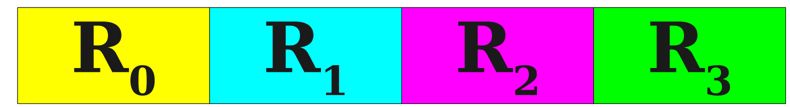
# Register Allocation with Live Intervals



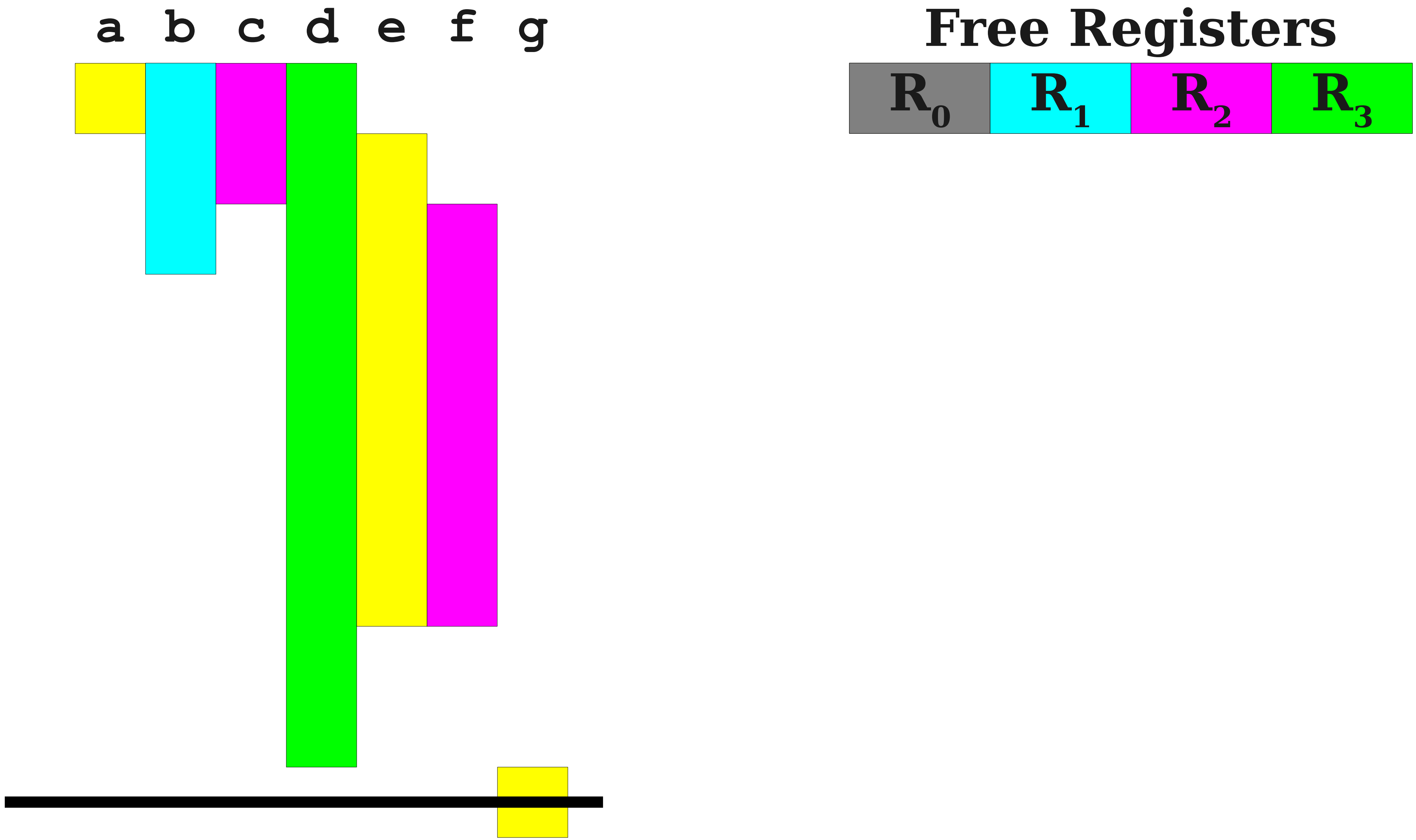
# Register Allocation with Live Intervals



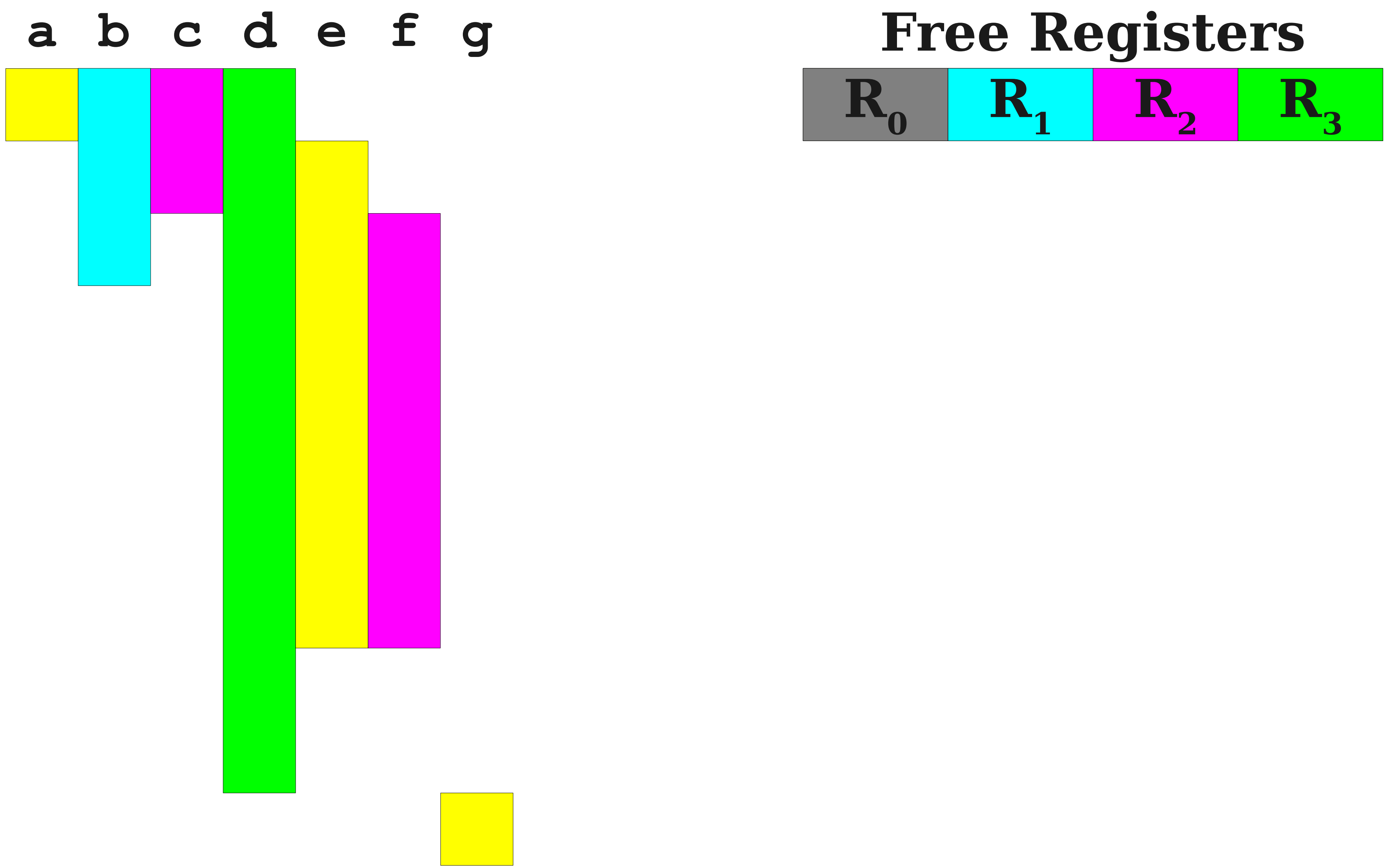

**Free Registers**



# Register Allocation with Live Intervals

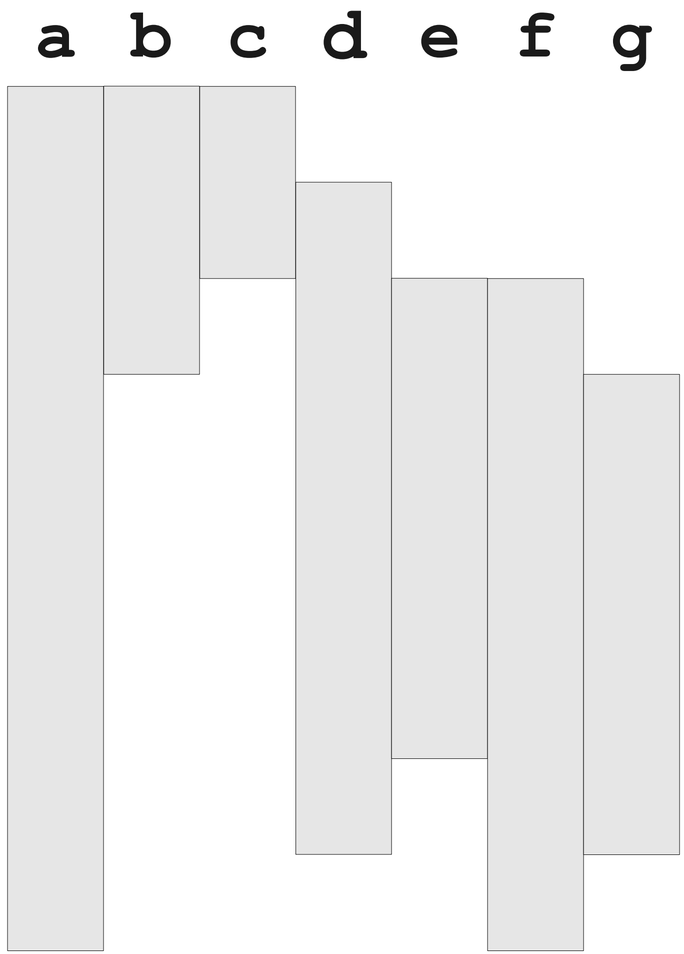


# Register Allocation with Live Intervals



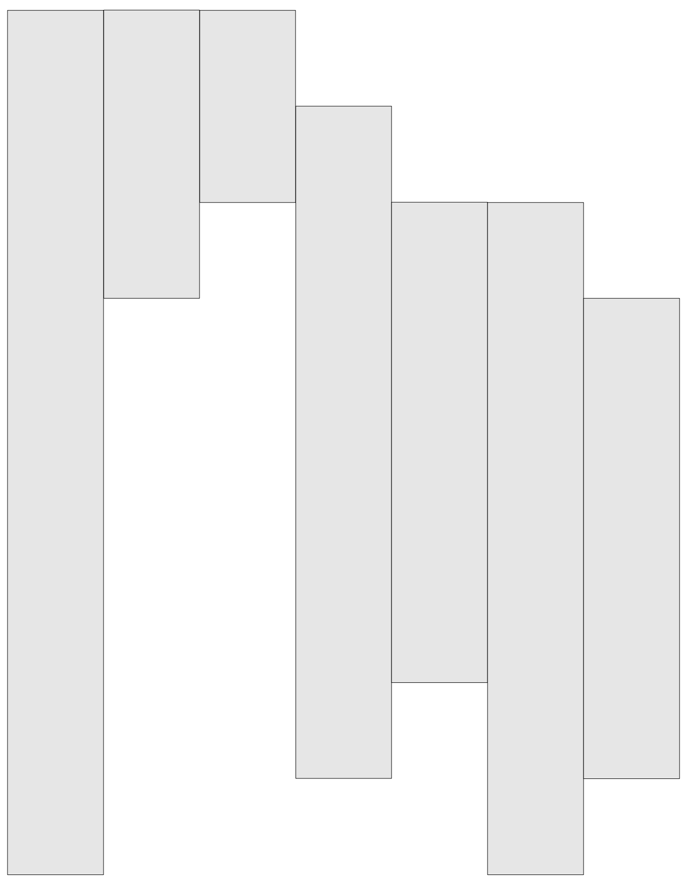
# Another Example

# Another Example



# Another Example

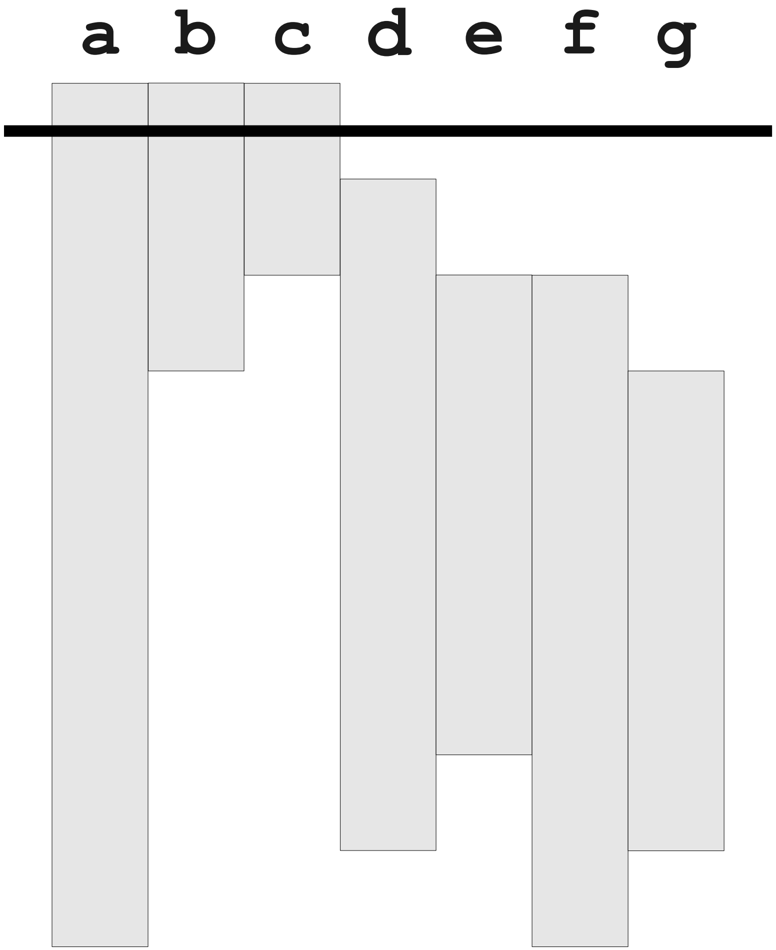
a b c d e f g



**Free Registers**

| | | |
|---|---|---|
| $R_0$ | $R_1$ | $R_2$ |

# Another Example

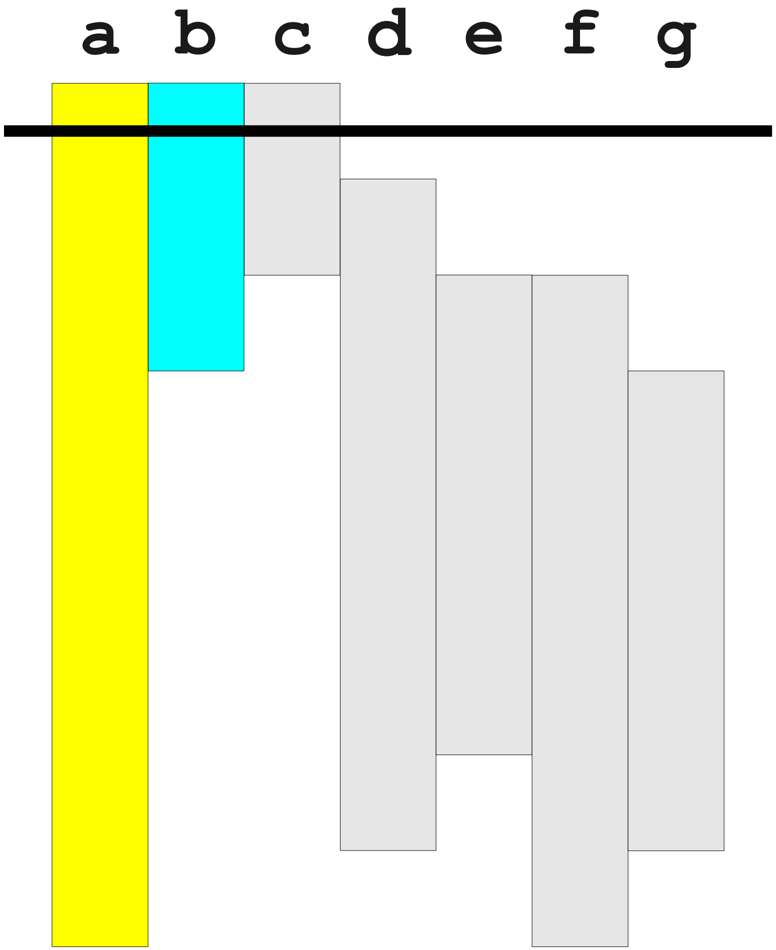



**Free Registers**

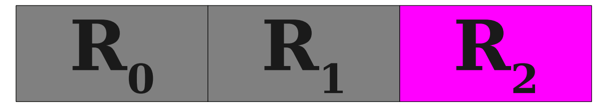
| $R_0$ | $R_1$ | $R_2$ |
|---|---|---|

# Another Example

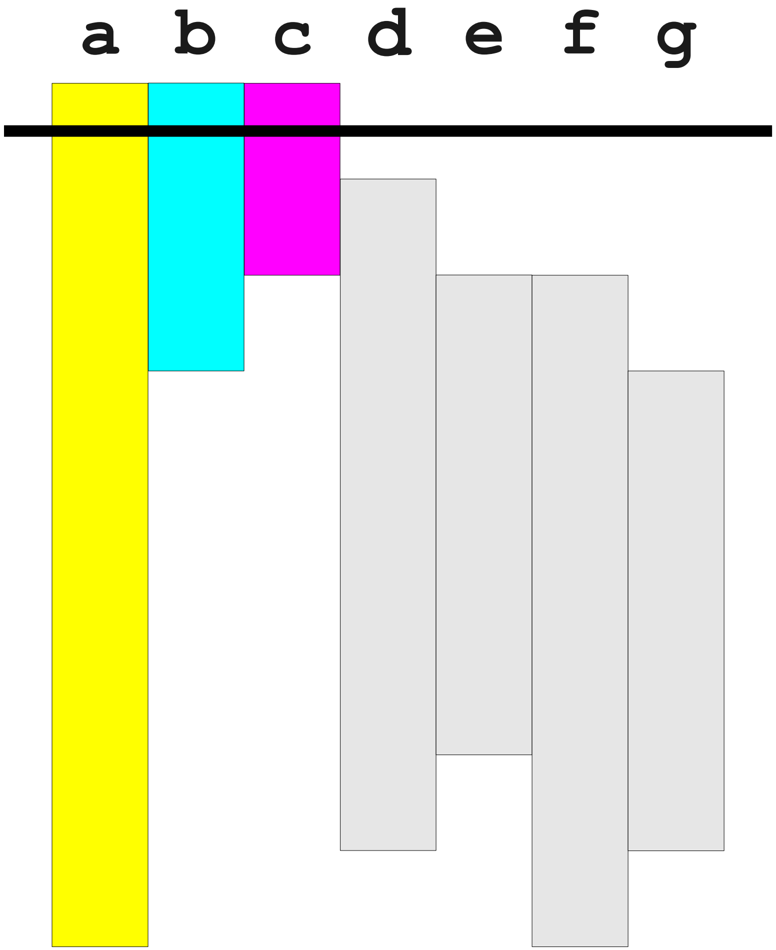
# Another Example




**Free Registers**



# Another Example




**Free Registers**

| $R_0$ | $R_1$ | $R_2$ |
|---|---|---|

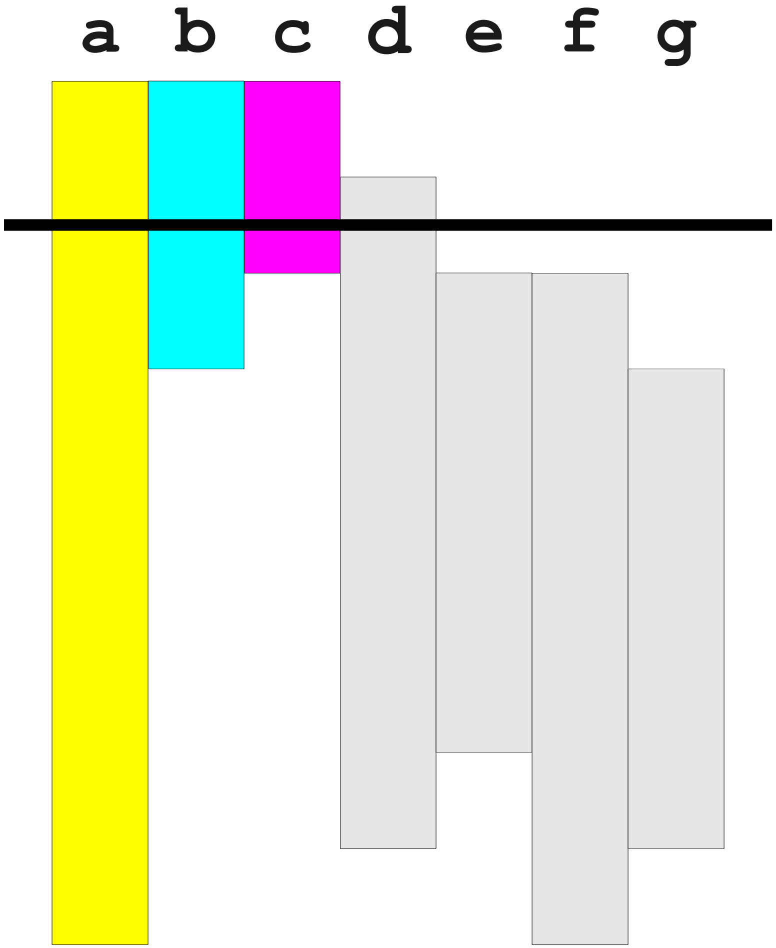
# Another Example


What do we do now?

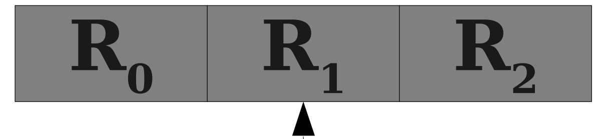
# Register Spilling

- If a register cannot be found for a variable *v*, we may need to **spill** a variable.
- When a variable is spilled, it is stored in memory rather than a register.
- When we need a register for the spilled variable:
  - Evict some existing register to memory.
  - Load the variable into the register.
  - When done, write the register back to memory and reload the register with its original value.
- Spilling is slow, but sometimes necessary.

# Another Example

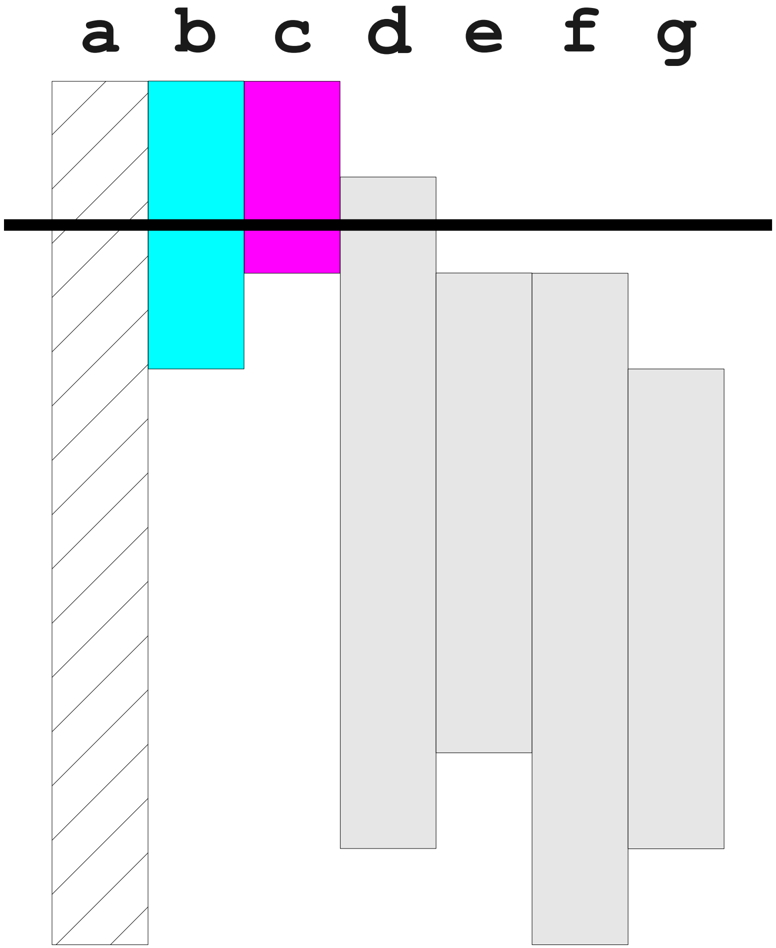



**Free Registers**

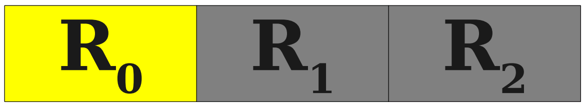



What do we do now?

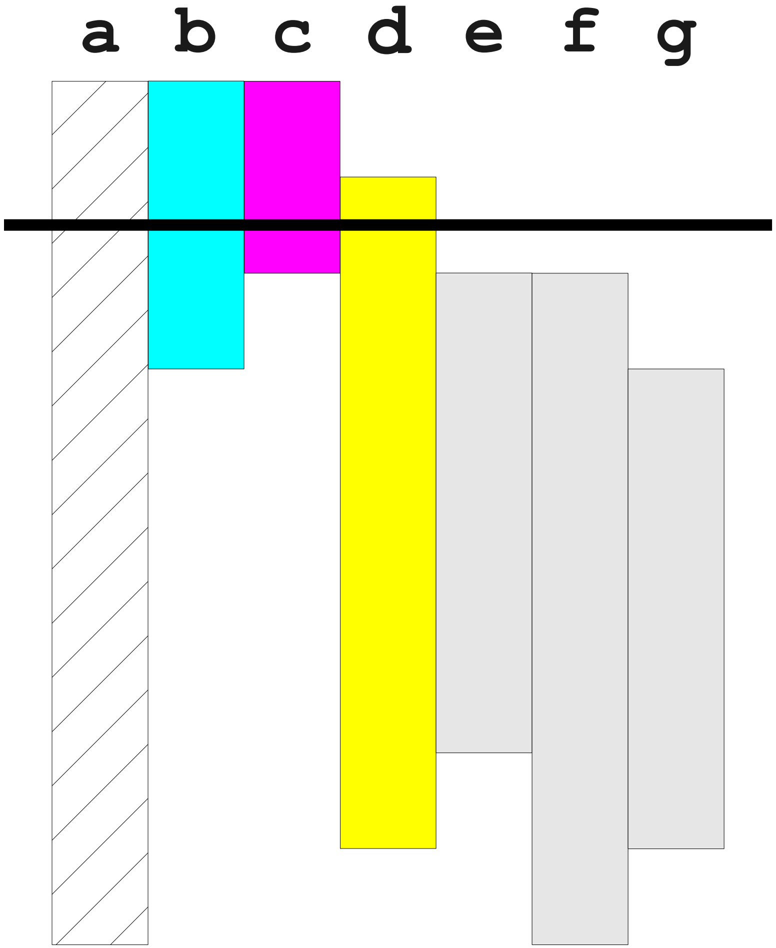
# Another Example




**Free Registers**



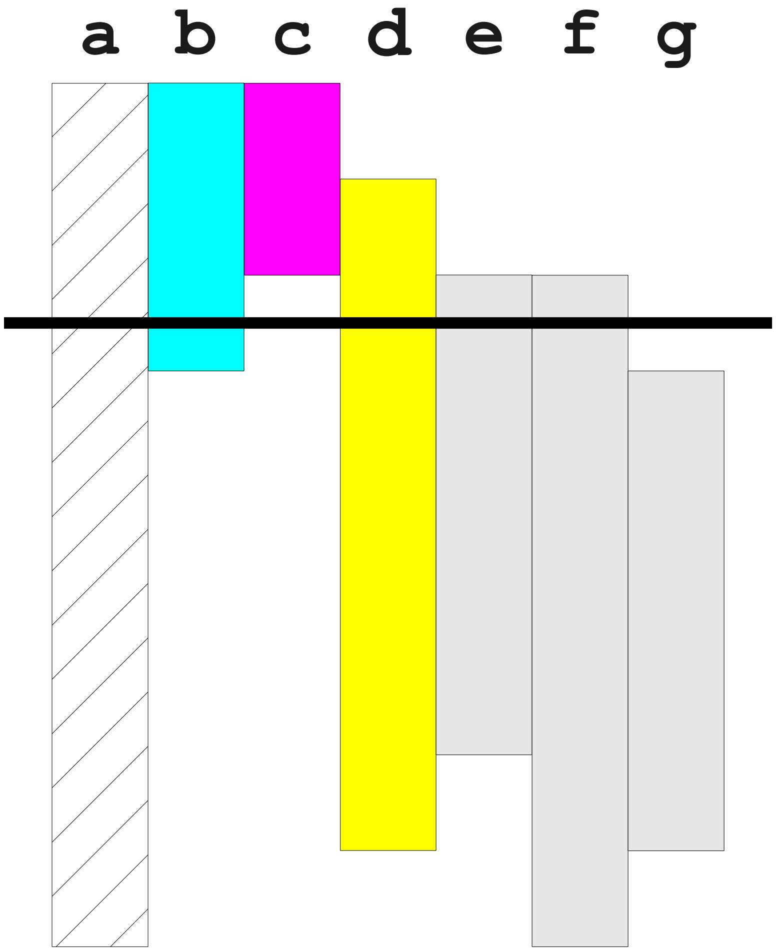
# Another Example




**Free Registers**

| $R_0$ | $R_1$ | $R_2$ |
|---|---|---|

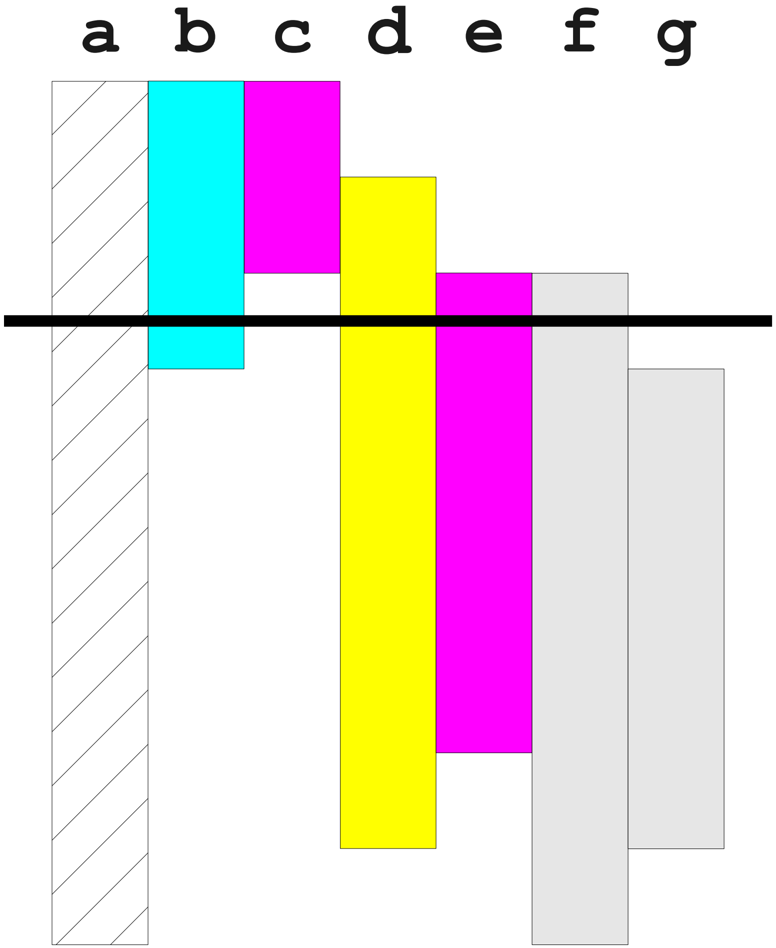
# Another Example




**Free Registers**

| | | |
|---|---|---|
| $R_0$ | $R_1$ | $R_2$ |

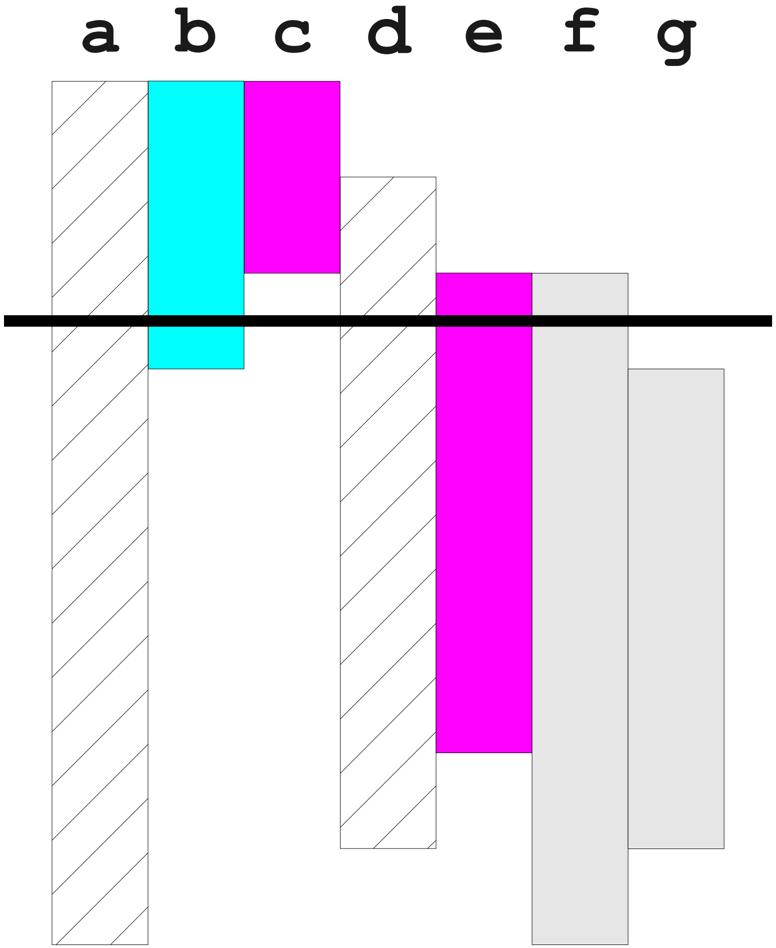
# Another Example



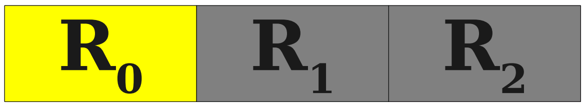

**Free Registers**

| | | |
|---|---|---|
| $R_0$ | $R_1$ | $R_2$ |

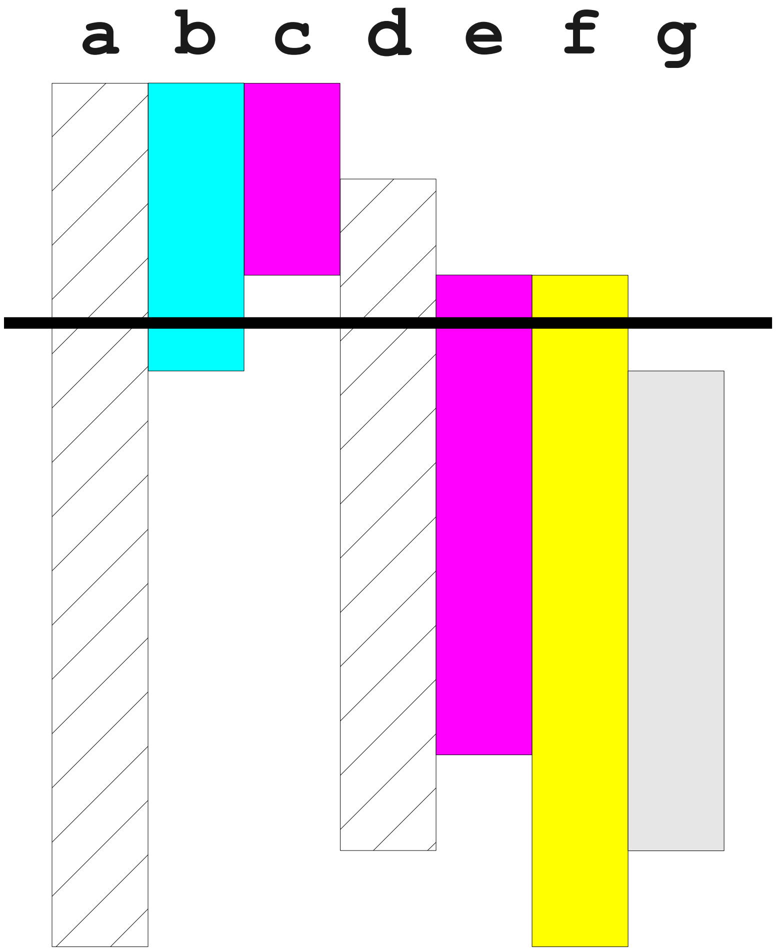
# Another Example




**Free Registers**



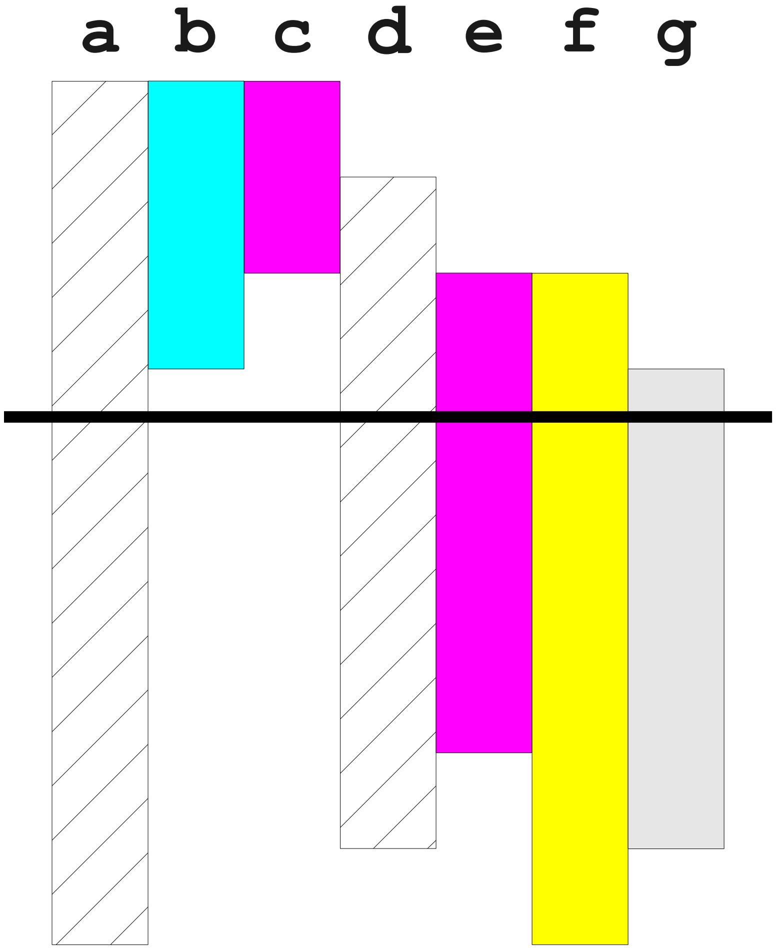
# Another Example



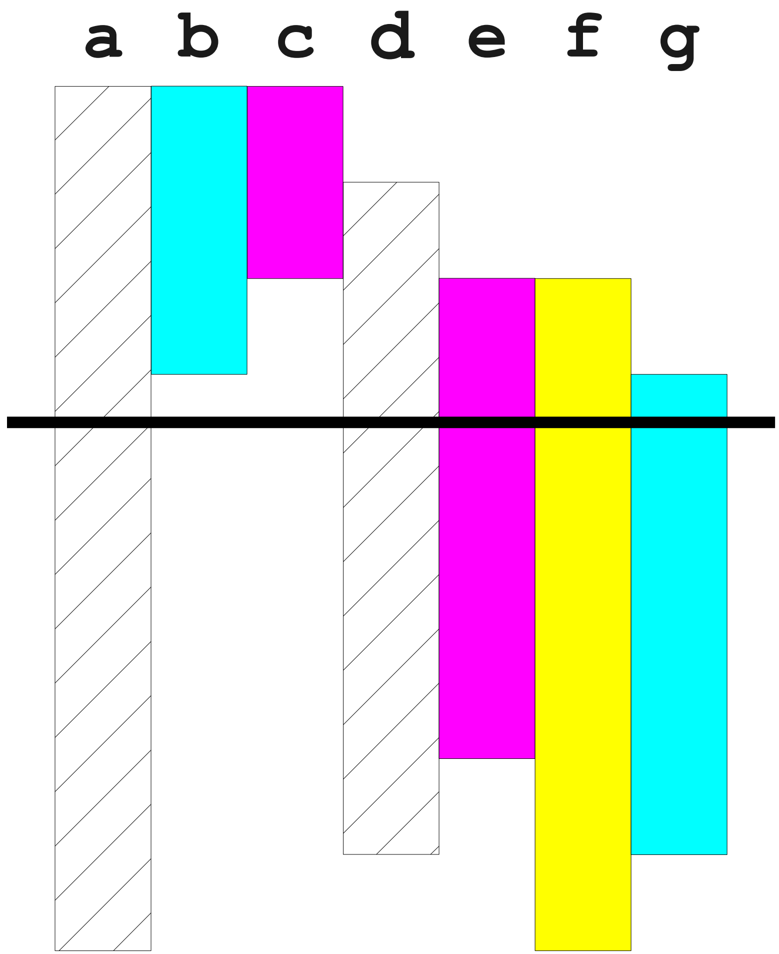

**Free Registers**

| | | |
|---|---|---|
| $R_0$ | $R_1$ | $R_2$ |

# Another Example



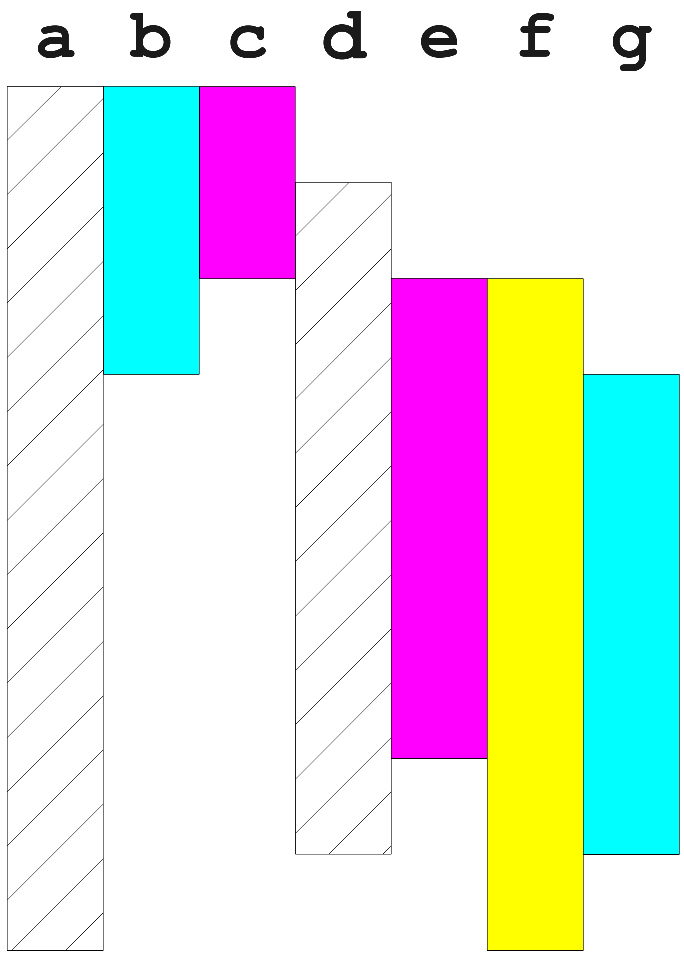

# Another Example




**Free Registers**

| | | |
|---|---|---|
| $R_0$ | $R_1$ | $R_2$ |

# Another Example




**Free Registers**

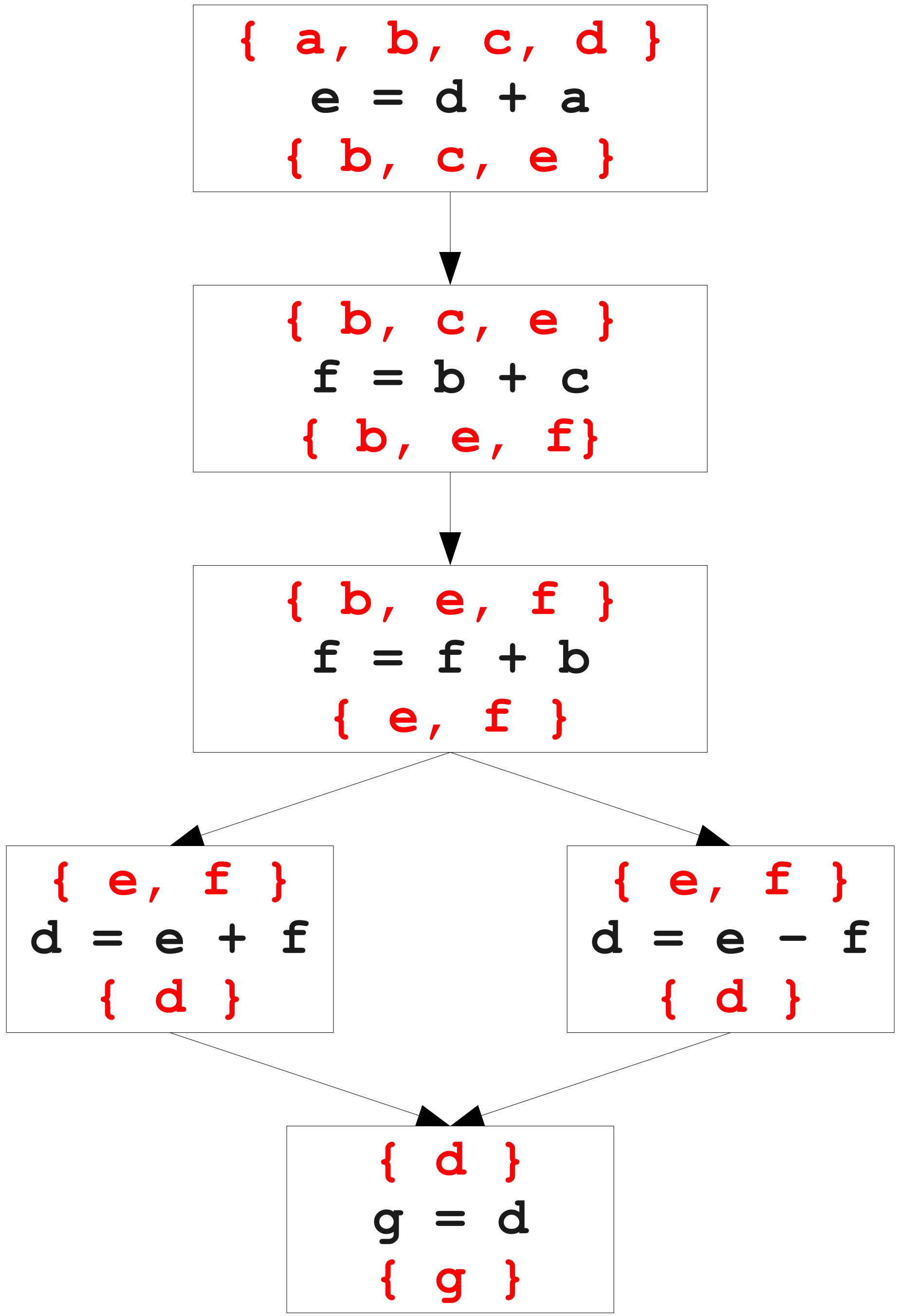
| | | |
|---|---|---|
| $R_0$ | $R_1$ | $R_2$ |

# Linear Scan Register Allocation

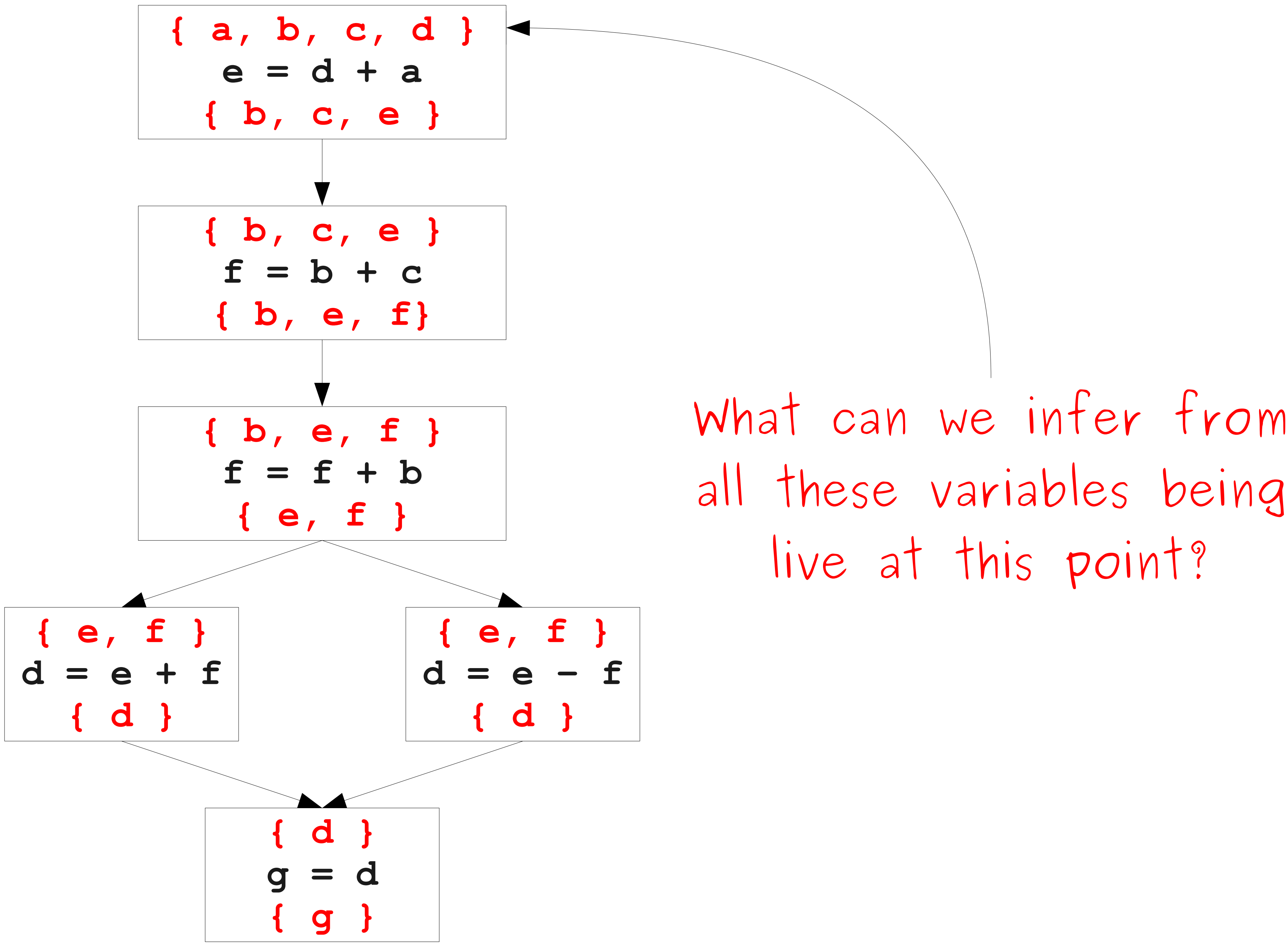
- This algorithm is called **linear scan register allocation** and is a comparatively new algorithm.
- Advantages:
  - Very efficient (after computing live intervals, runs in linear time)
  - Produces good code in many instances.
  - Allocation step works in one pass; can generate code during iteration.
  - Often used in JIT compilers like Java HotSpot.
- Disadvantages:
  - Imprecise due to use of live **intervals** rather than live **ranges**.
  - Other techniques known to be superior in many cases.

# An Entirely Different Approach

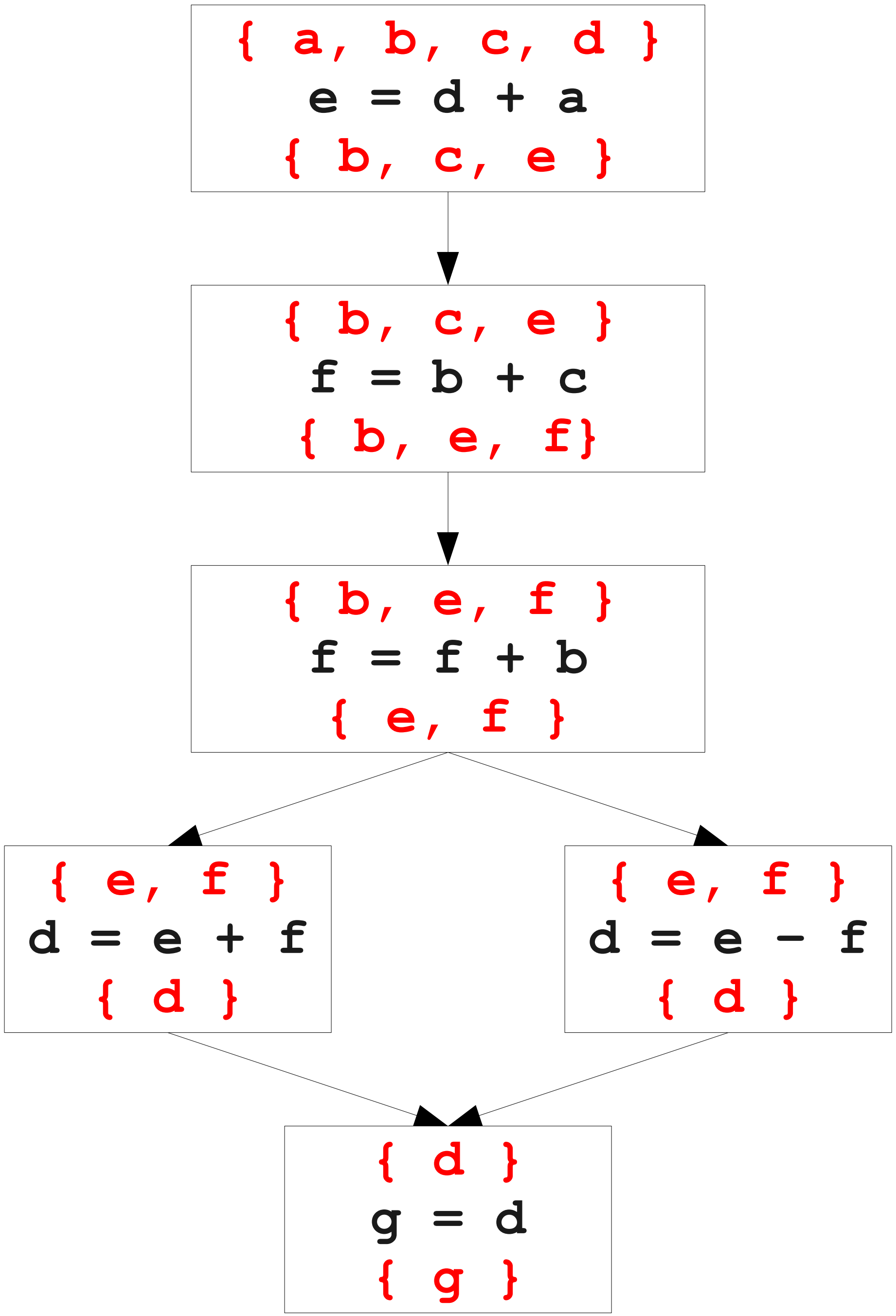
# An Entirely Different Approach



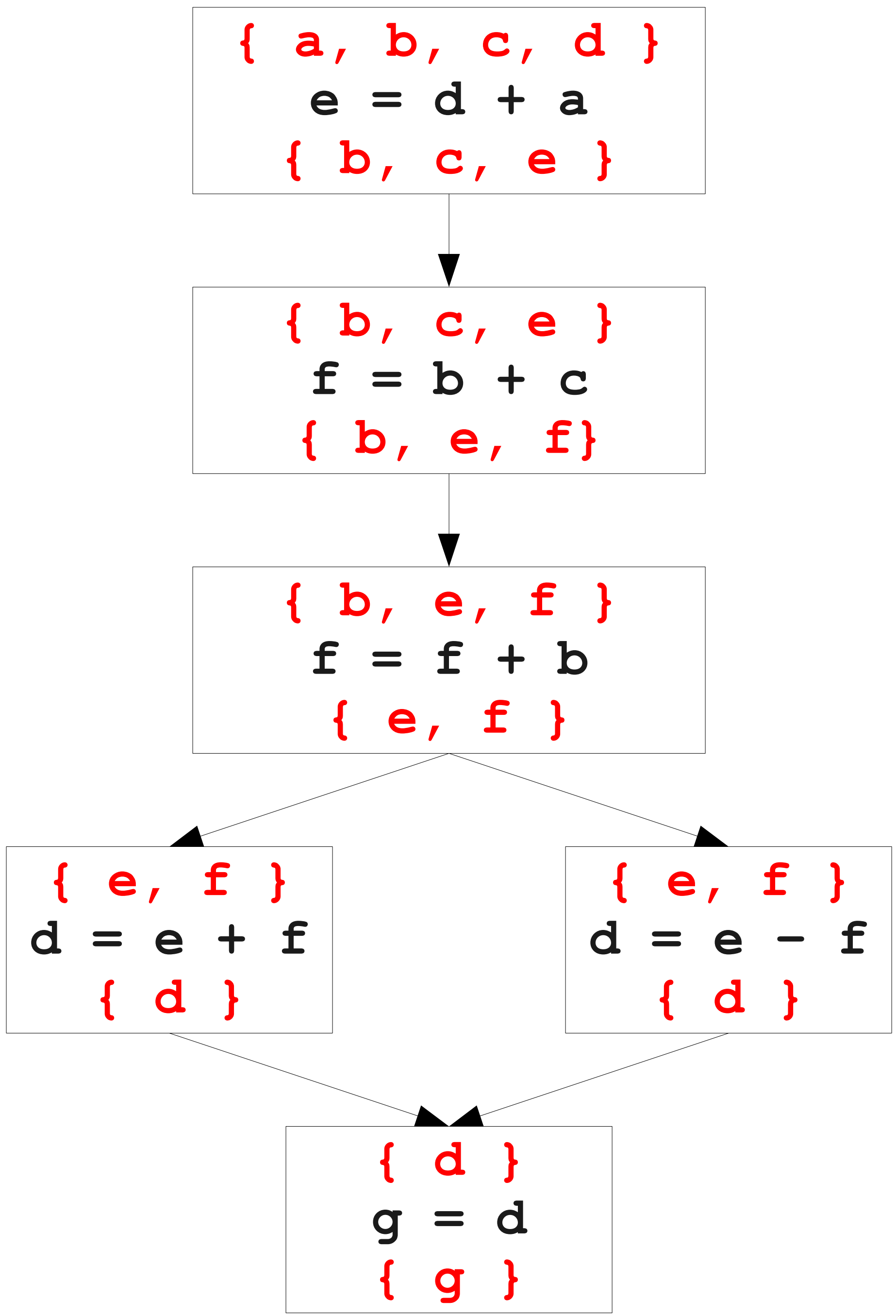
# An Entirely Different Approach

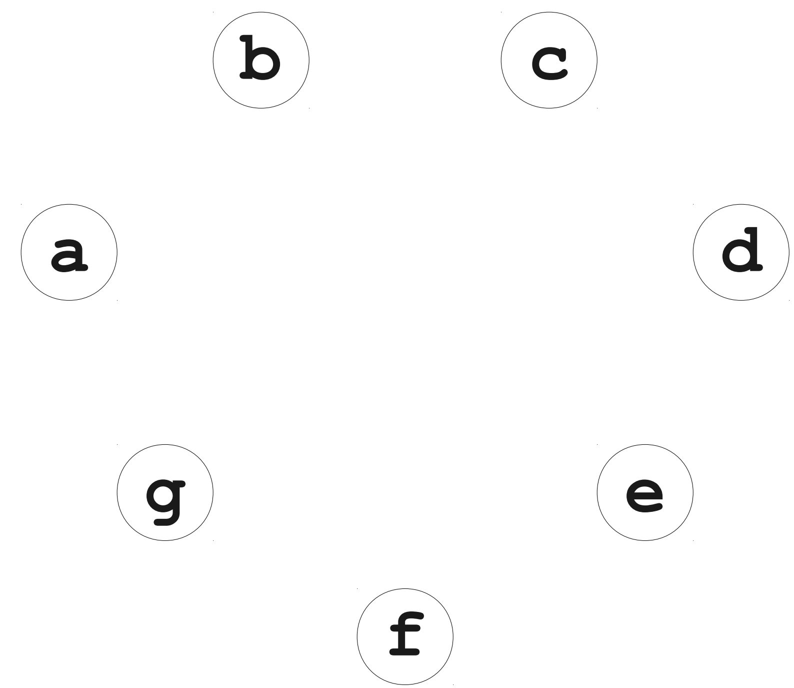


# An Entirely Different Approach

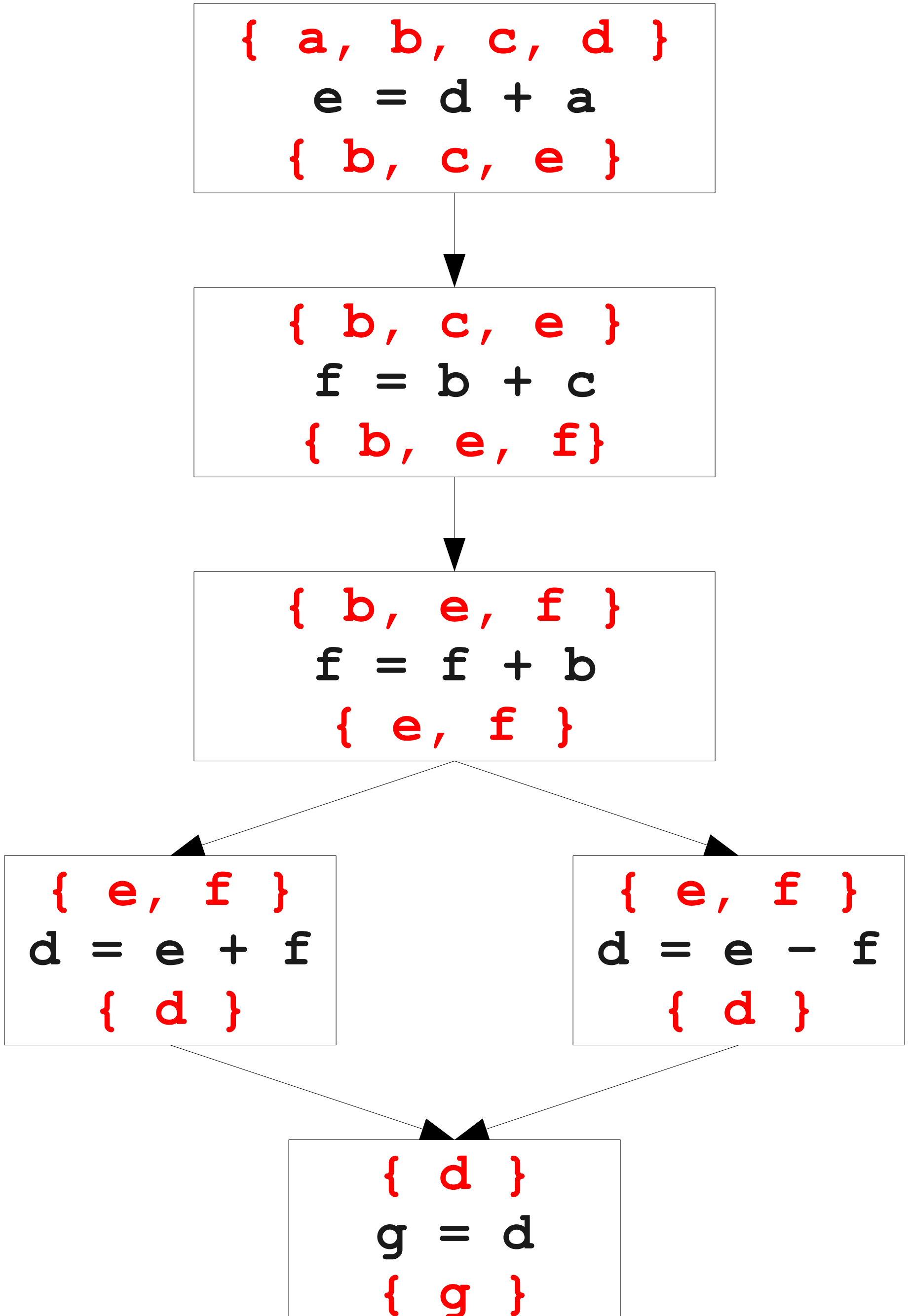


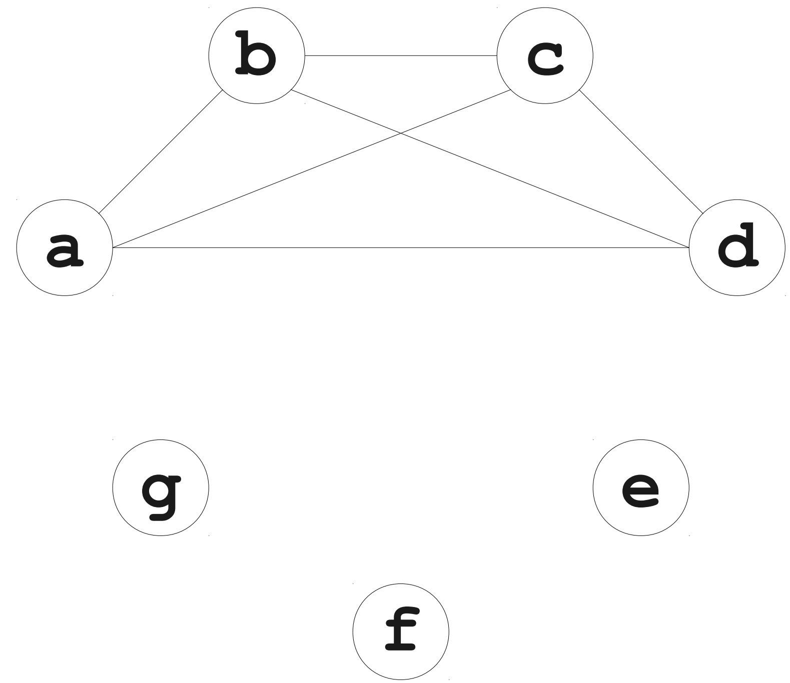
# An Entirely Different Approach



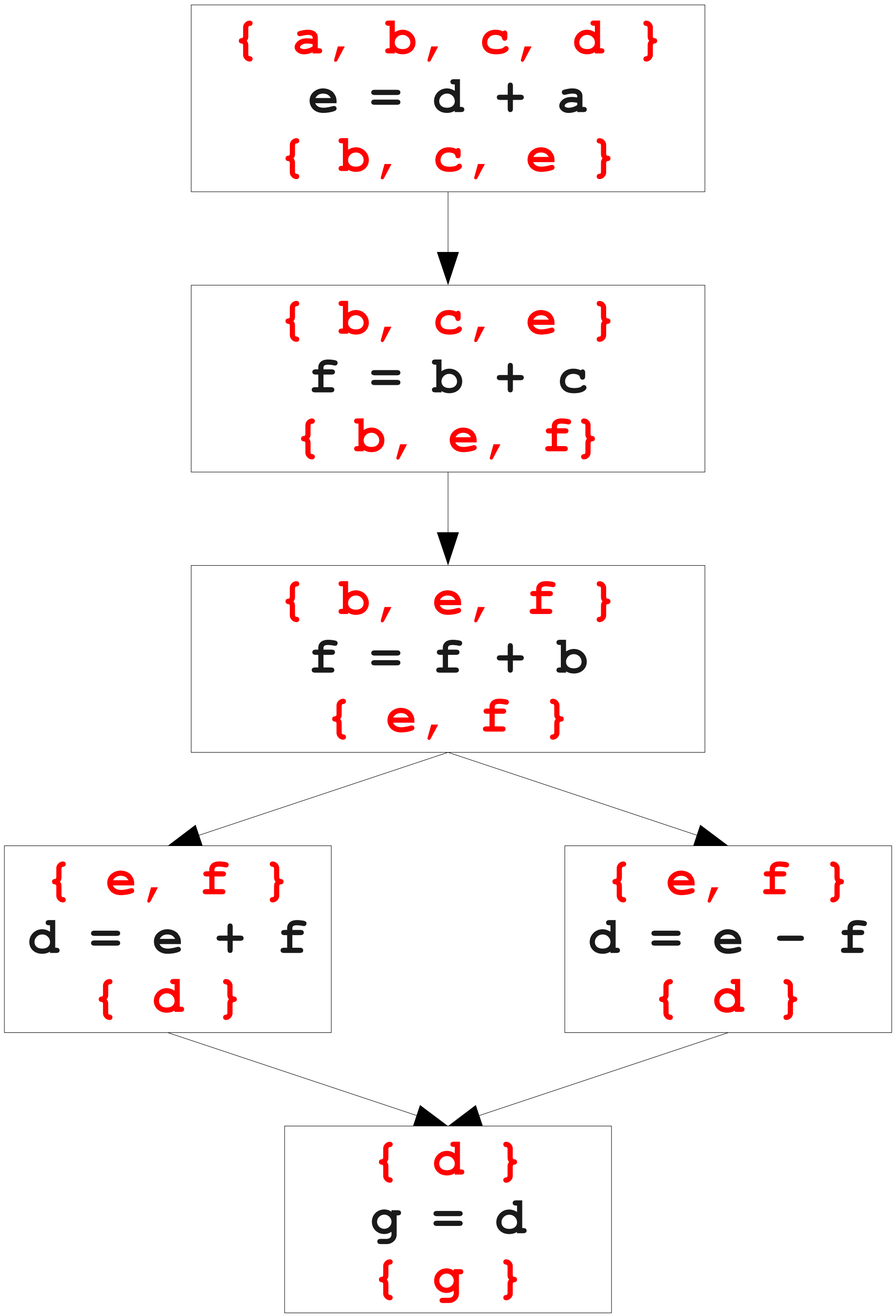



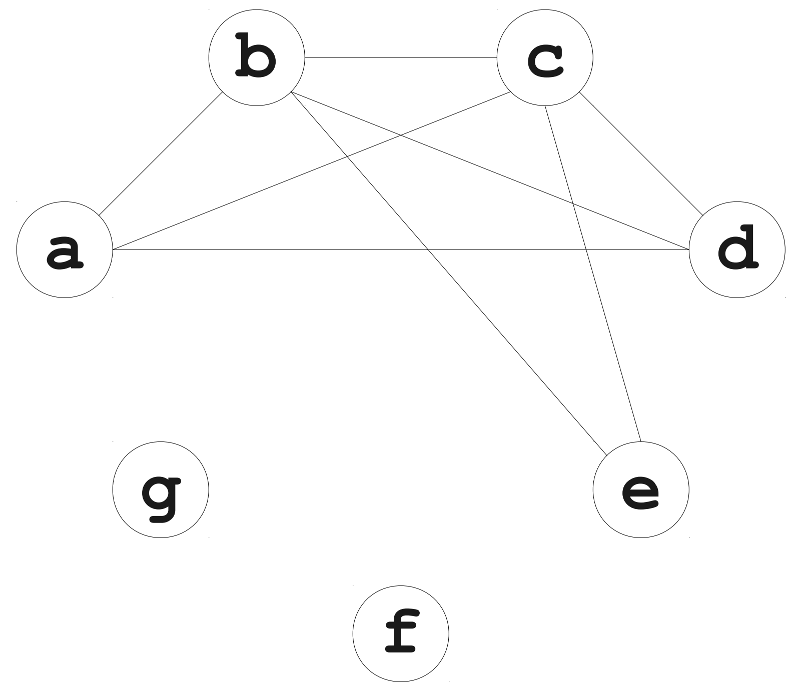
# An Entirely Different Approach



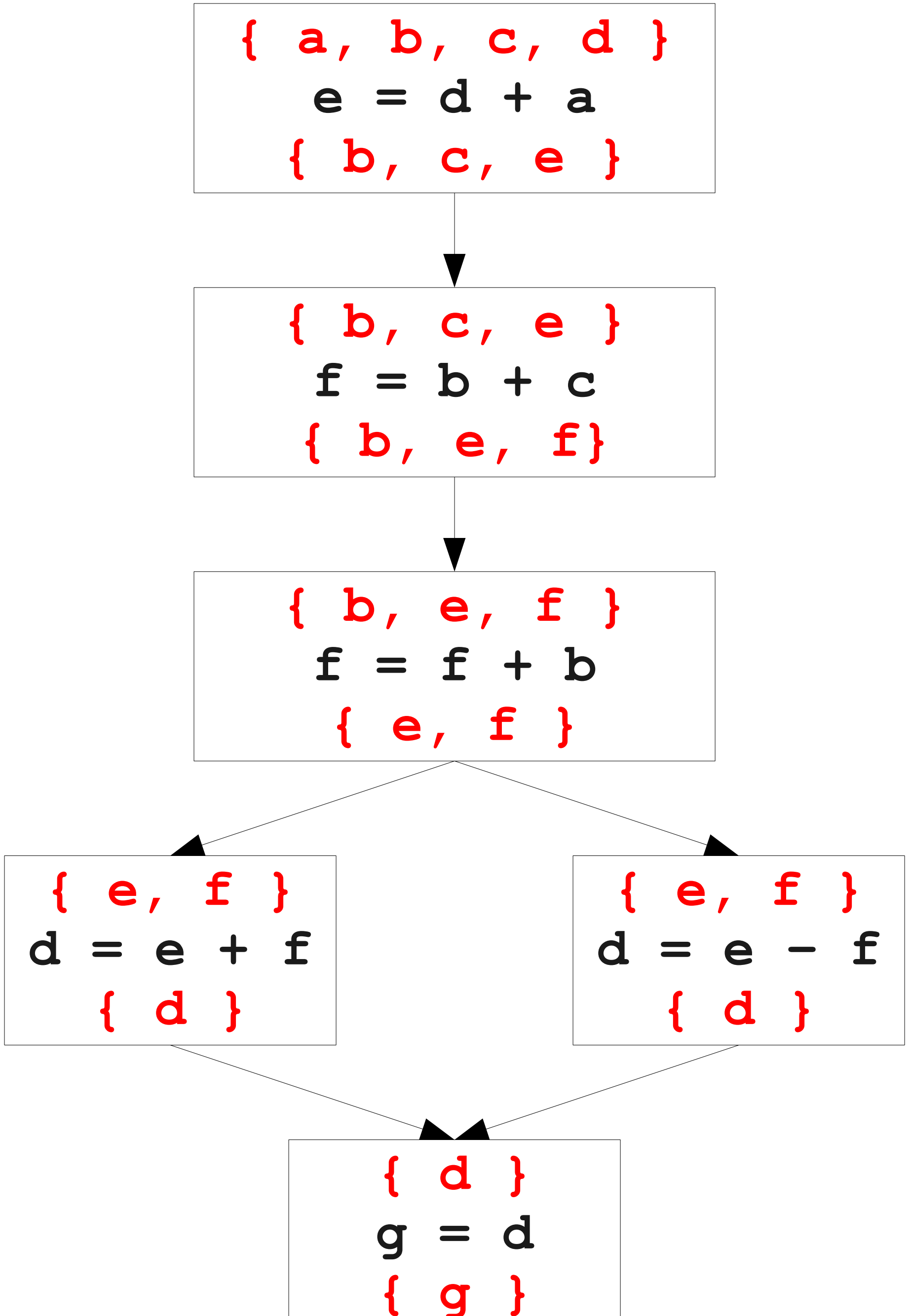



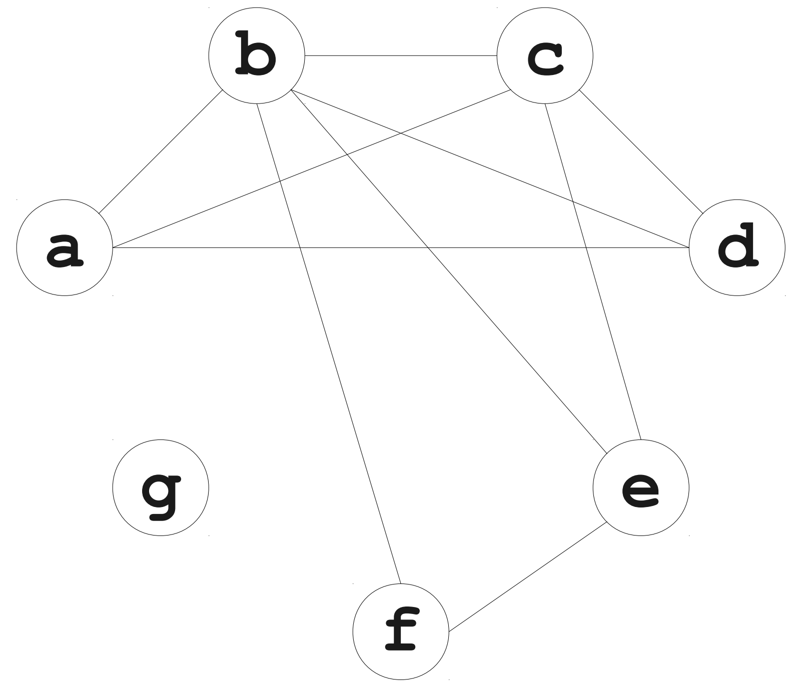
# An Entirely Different Approach



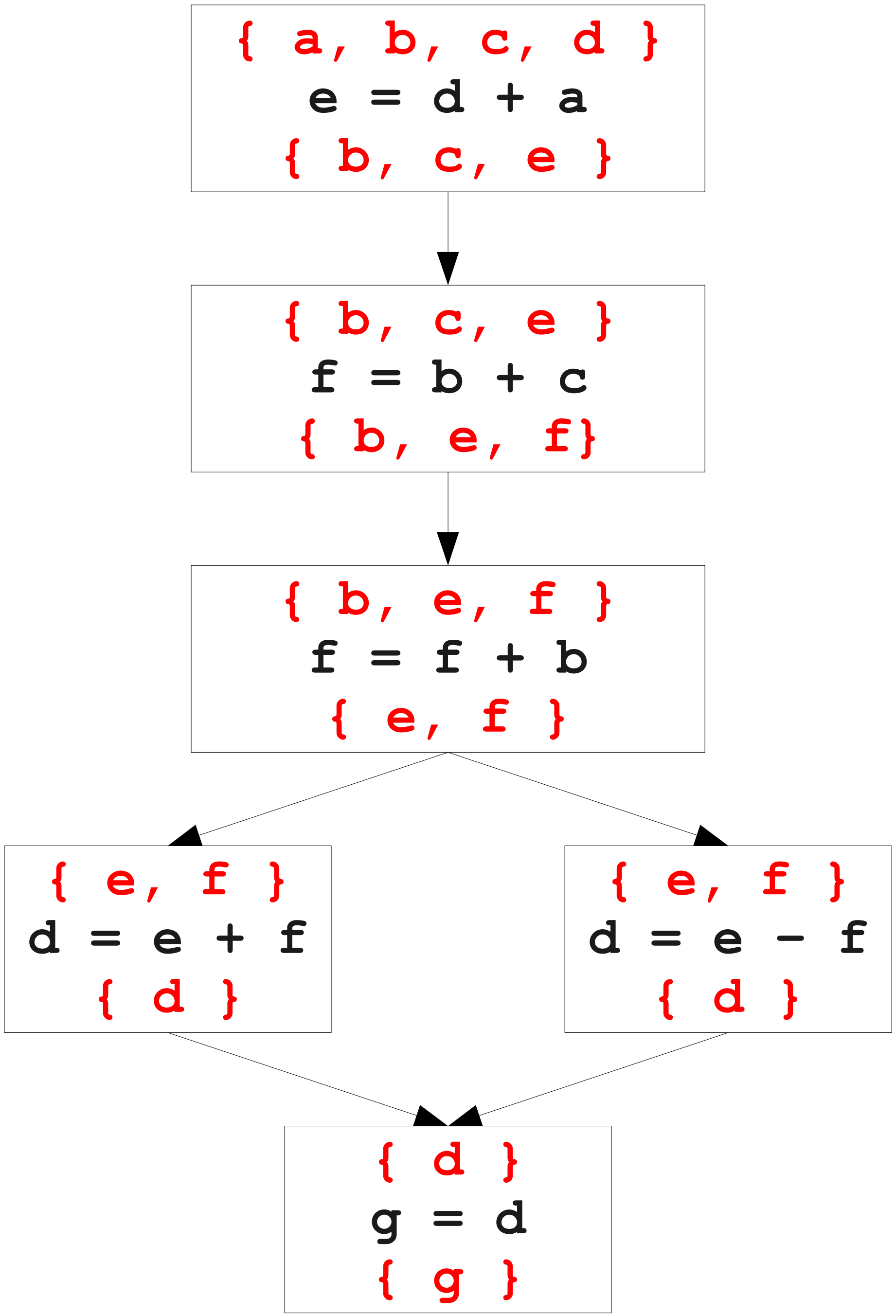



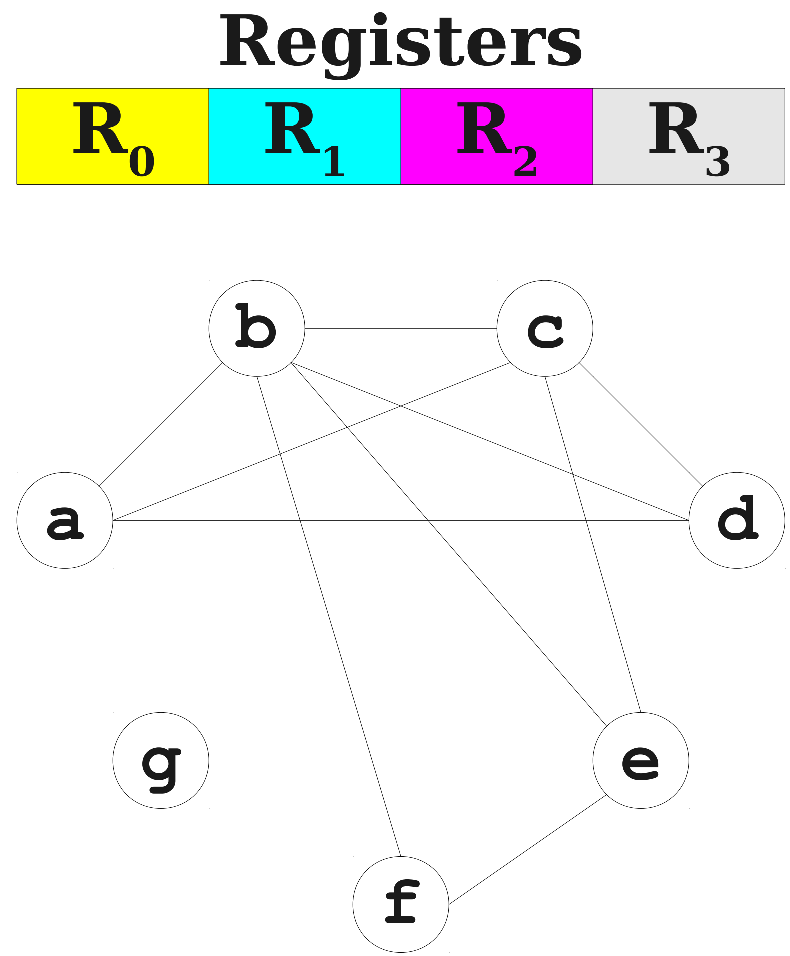
# An Entirely Different Approach



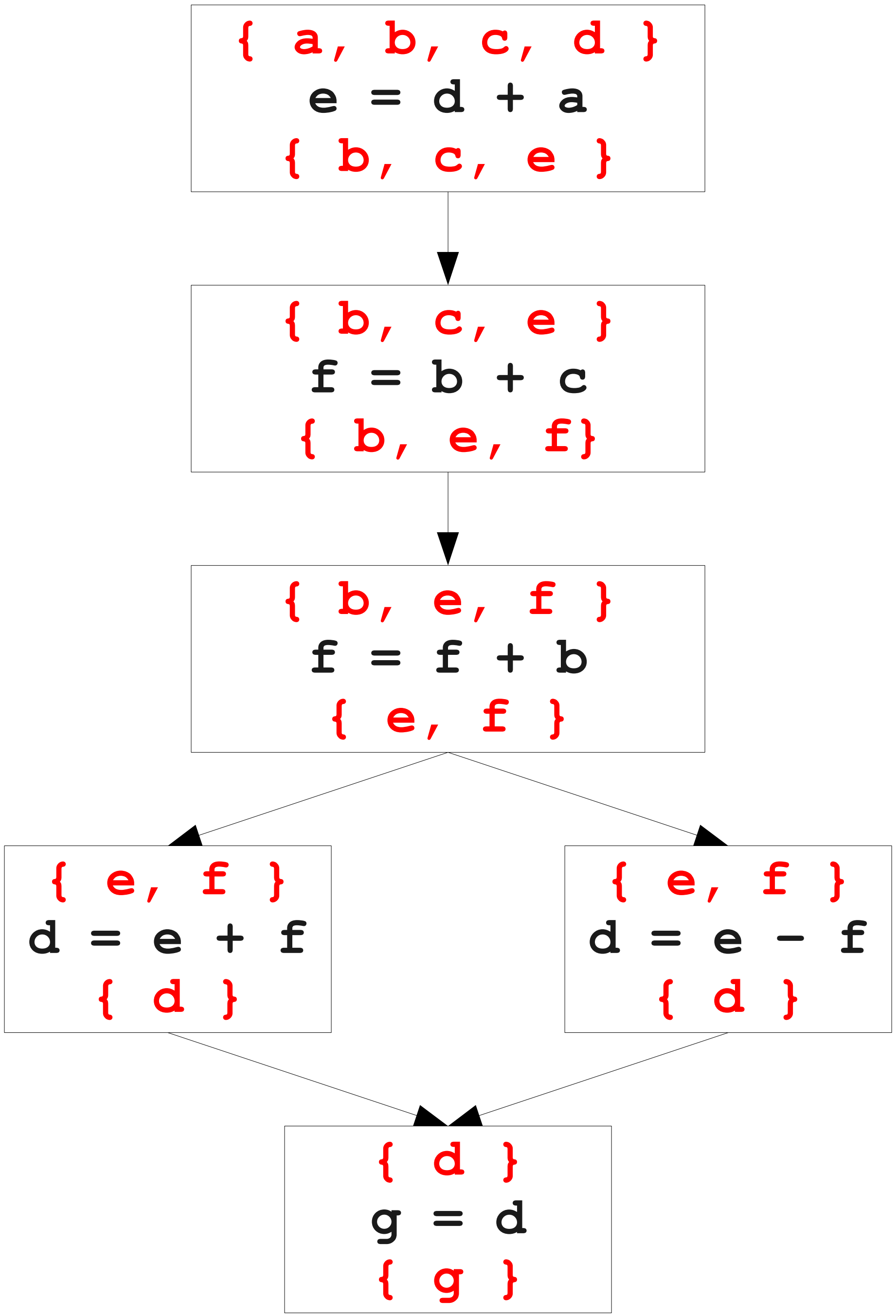



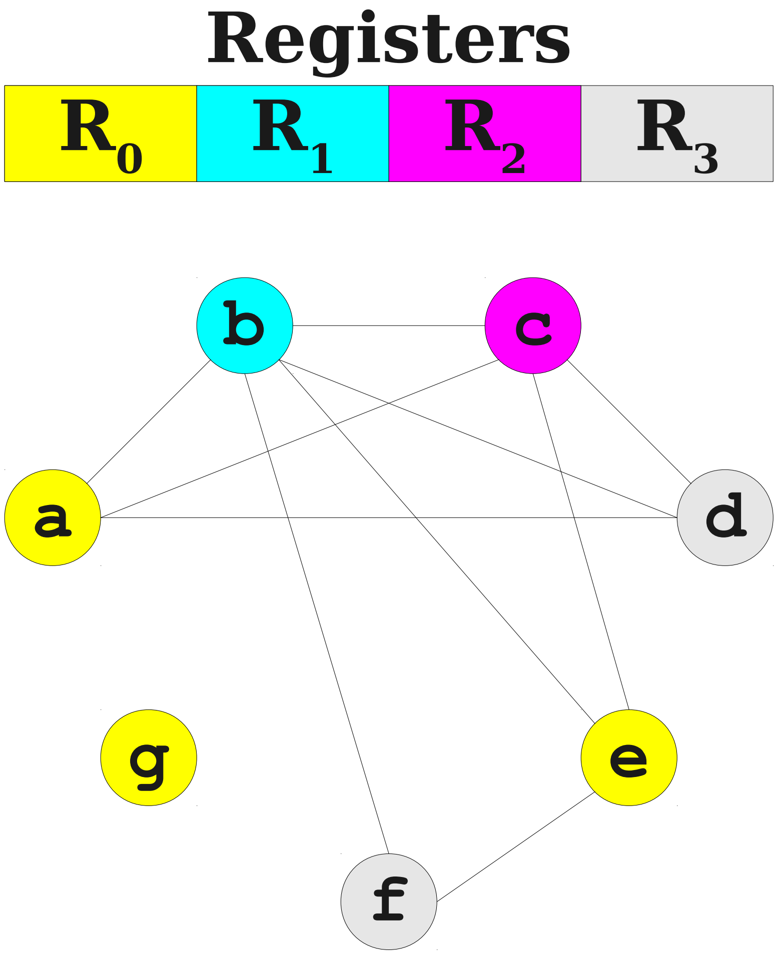
# An Entirely Different Approach



# An Entirely Different Approach



# The Register Interference Graph

- The **register interference graph** (RIG) of a control-flow graph is an undirected graph where
  - Each node is a variable.
  - There is an edge between two variables that are live at the same program point.
- Perform register allocation by assigning each variable a different register from all of its neighbors.
- There's just one catch...

# The One Catch

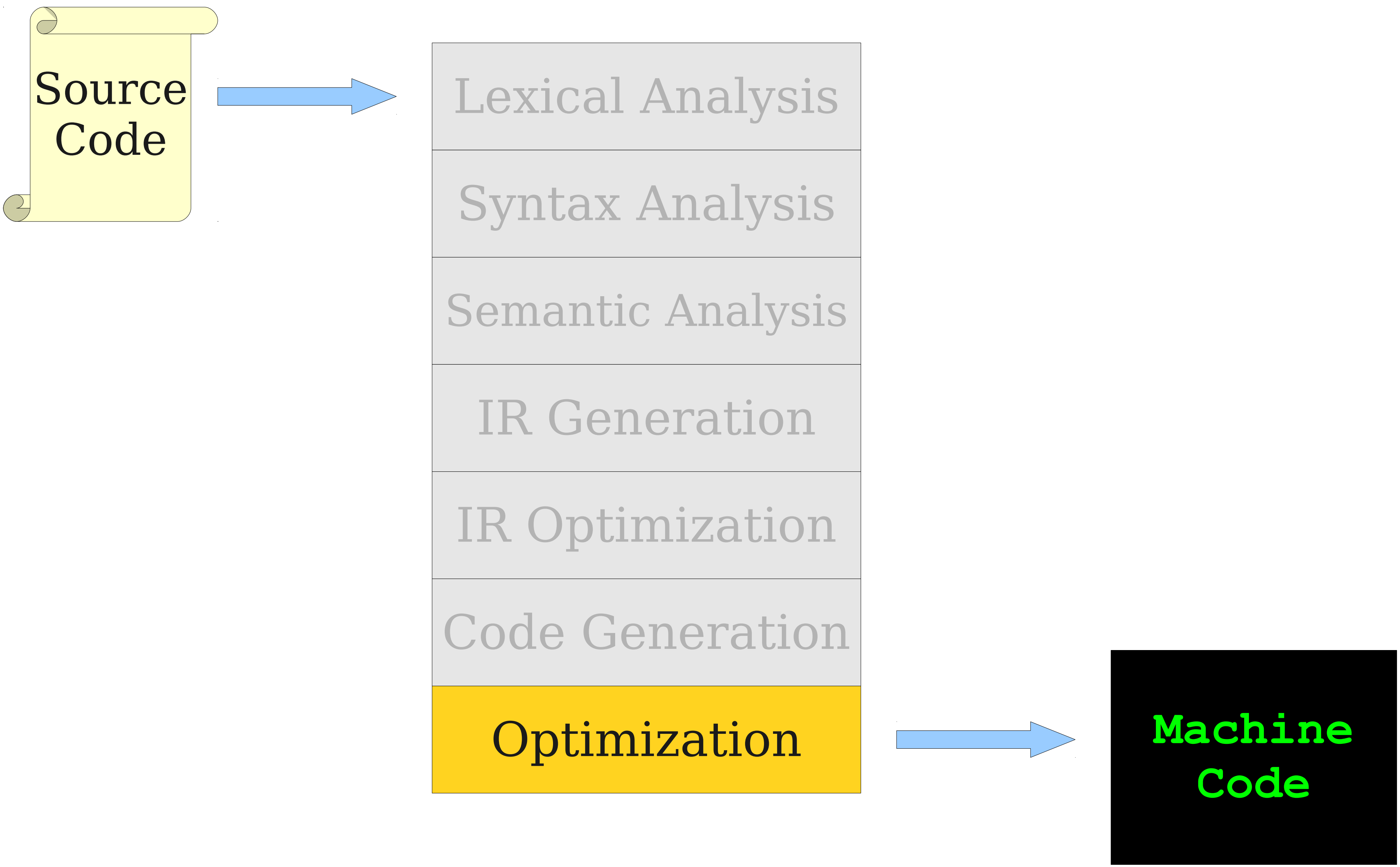
- This problem is equivalent to **graph-coloring**, which is **NP-hard** if there are at least three registers.
- No good polynomial-time algorithms (or even good approximations!) are known for this problem.
- We have to be content with a heuristic that is good enough for RIGs that arise in practice.

# Summary of Register Allocation

- Critical step in all optimizing compilers.
- The **linear scan** algorithm uses **live intervals** to greedily assign variables to registers.
  - Often used in JIT compilers due to efficiency.
- **Chaitin's algorithm** uses the **register interference graph** (based on **live ranges**) and **graph coloring** to assign registers.
  - The basis for the technique used in GCC.

# Code Optimization

# Where We Are


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

# Final Code Optimization

- **Goal**: Optimize generated code by exploiting machine-dependent properties not visible at the IR level.
- Critical step in most compilers, but often very messy:
  - Techniques developed for one machine may be completely useless on another.
  - Techniques developed for one language may be completely useless with another.

# Optimizations for Pipelining

# Processor Pipelines

# Processor Pipelines

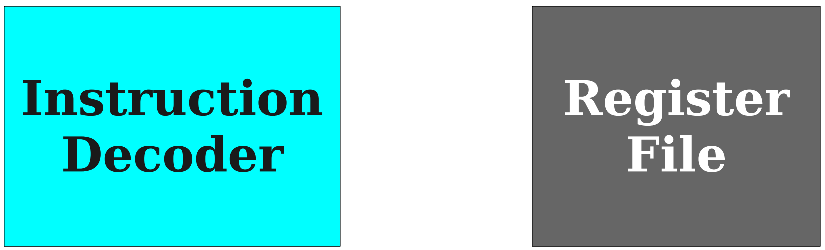
```
add $t2, $t0, $t1  # $t2 = $t0 + $t1
add $t5, $t3, $t4  # $t5 = $t3 + $t4
add $t8, $t6, $t7  # $t8 = $t6 + $t7
```

# Processor Pipelines

**Instruction Decoder**

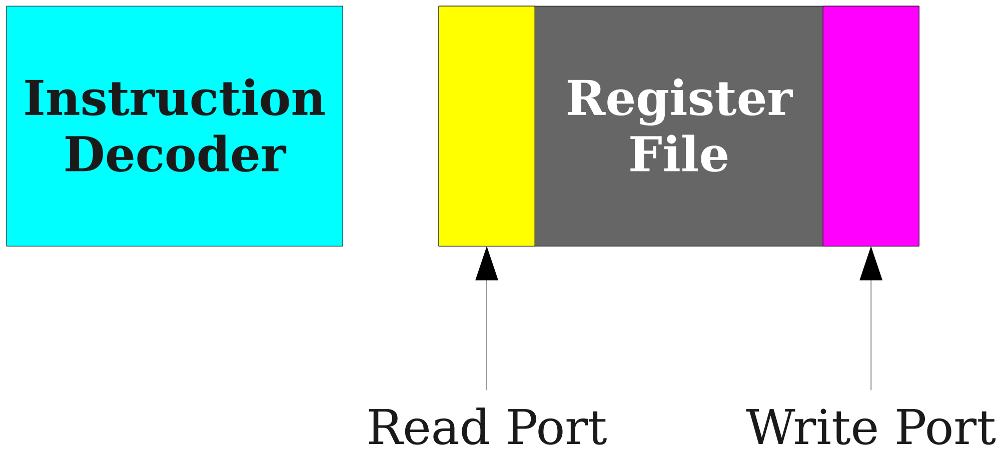
```
add $t2, $t0, $t1   # $t2 = $t0 + $t1
add $t5, $t3, $t4   # $t5 = $t3 + $t4
add $t8, $t6, $t7   # $t8 = $t6 + $t7
```

# Processor Pipelines



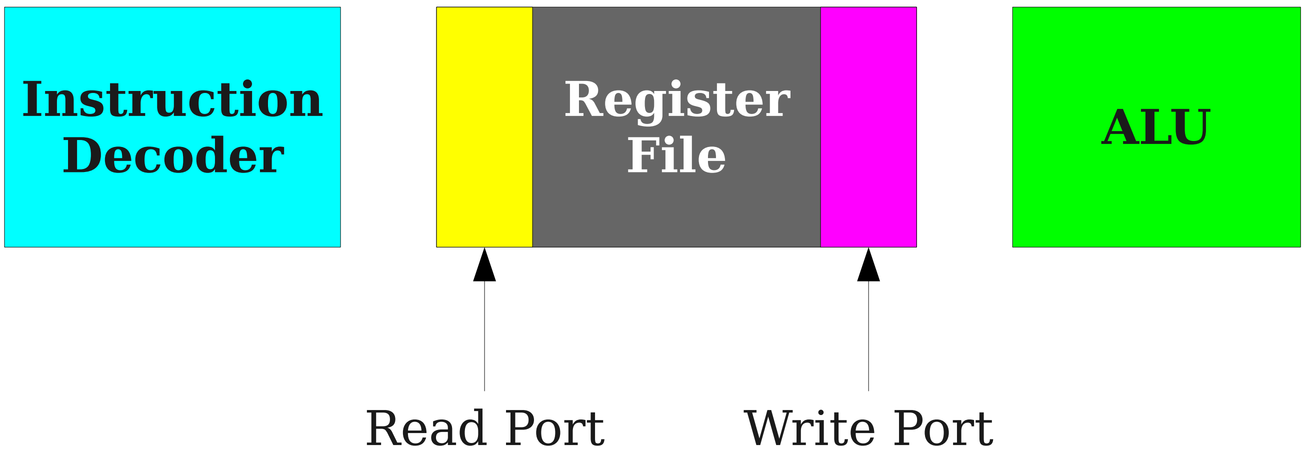

```
add $t2, $t0, $t1   # $t2 = $t0 + $t1
add $t5, $t3, $t4   # $t5 = $t3 + $t4
add $t8, $t6, $t7   # $t8 = $t6 + $t7
```

# Processor Pipelines



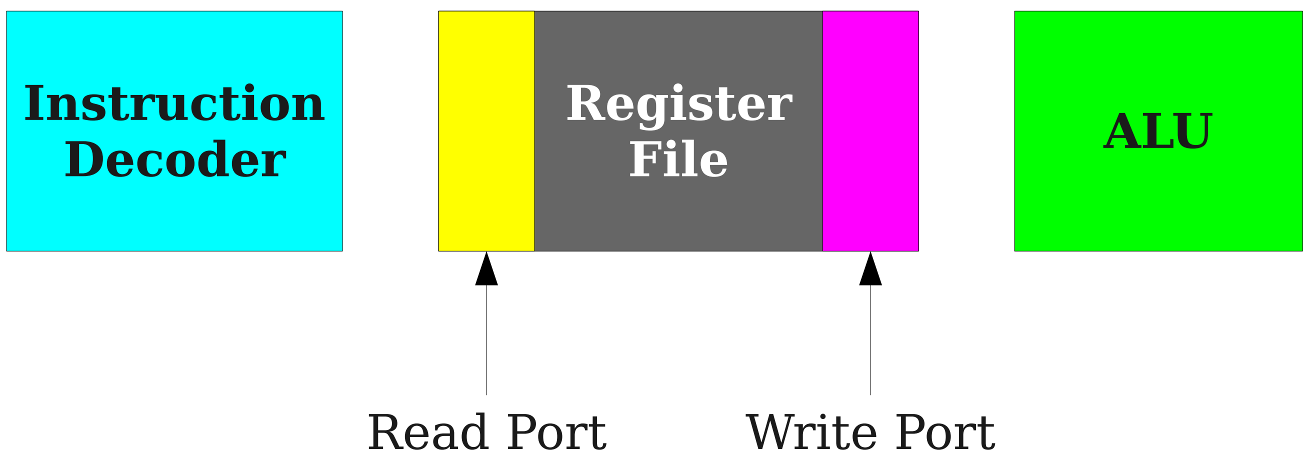

```
add $t2, $t0, $t1   # $t2 = $t0 + $t1
add $t5, $t3, $t4   # $t5 = $t3 + $t4
add $t8, $t6, $t7   # $t8 = $t6 + $t7
```

# Processor Pipelines




```
add $t2, $t0, $t1   # $t2 = $t0 + $t1
add $t5, $t3, $t4   # $t5 = $t3 + $t4
add $t8, $t6, $t7   # $t8 = $t6 + $t7
```

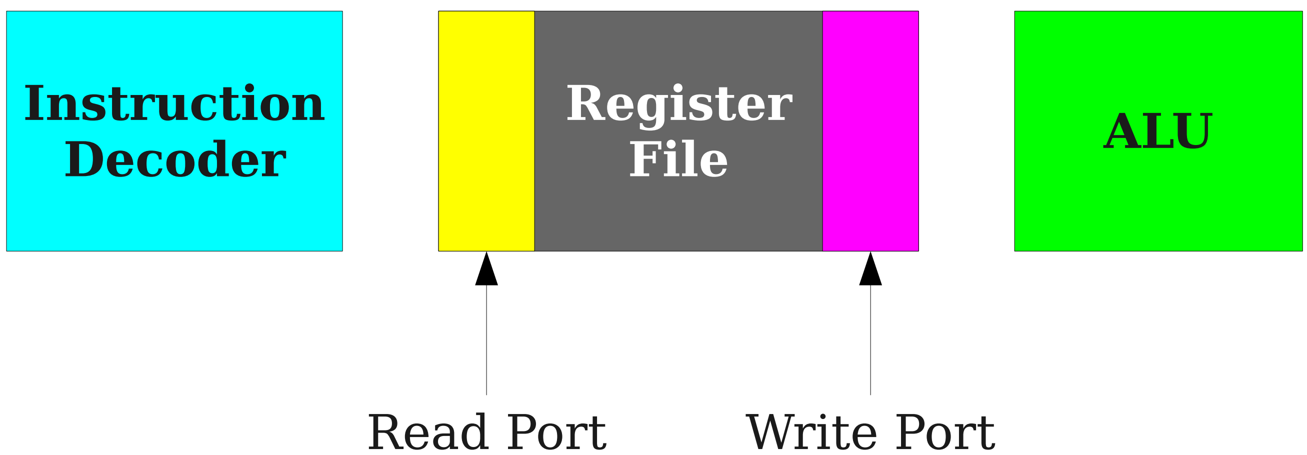
# Processor Pipelines




| ID | RR | ALU | RW |
|---|---|---|---|
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

```
add $t2, $t0, $t1   # $t2 = $t0 + $t1
add $t5, $t3, $t4   # $t5 = $t3 + $t4
add $t8, $t6, $t7   # $t8 = $t6 + $t7
```

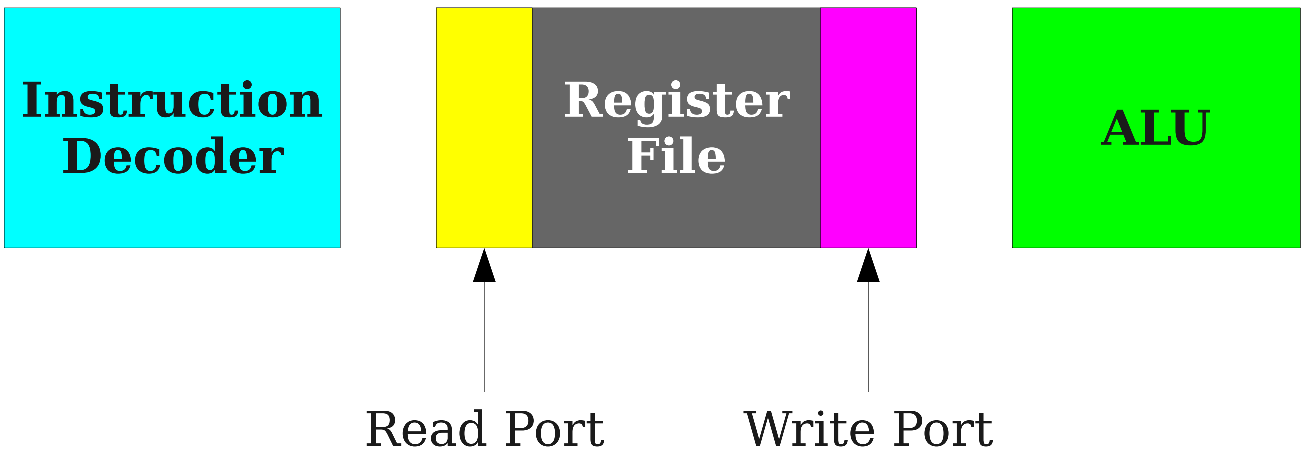
# Processor Pipelines

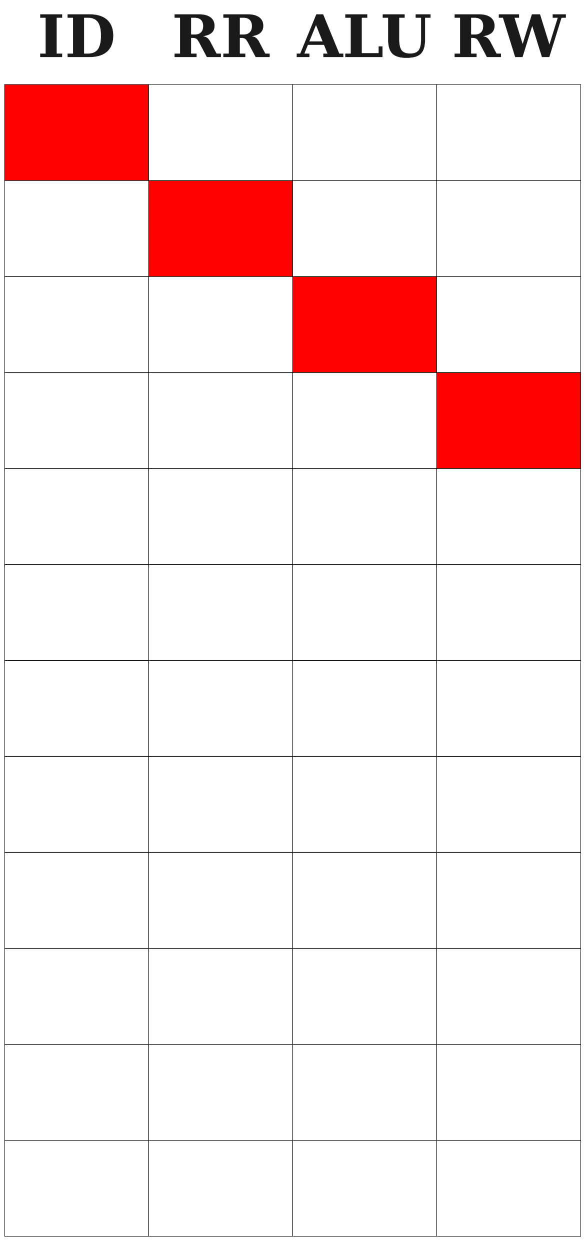



```
add $t2, $t0, $t1   # $t2 = $t0 + $t1
add $t5, $t3, $t4   # $t5 = $t3 + $t4
add $t8, $t6, $t7   # $t8 = $t6 + $t7
```

| ID | RR | ALU | RW |
|---|---|---|---|
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

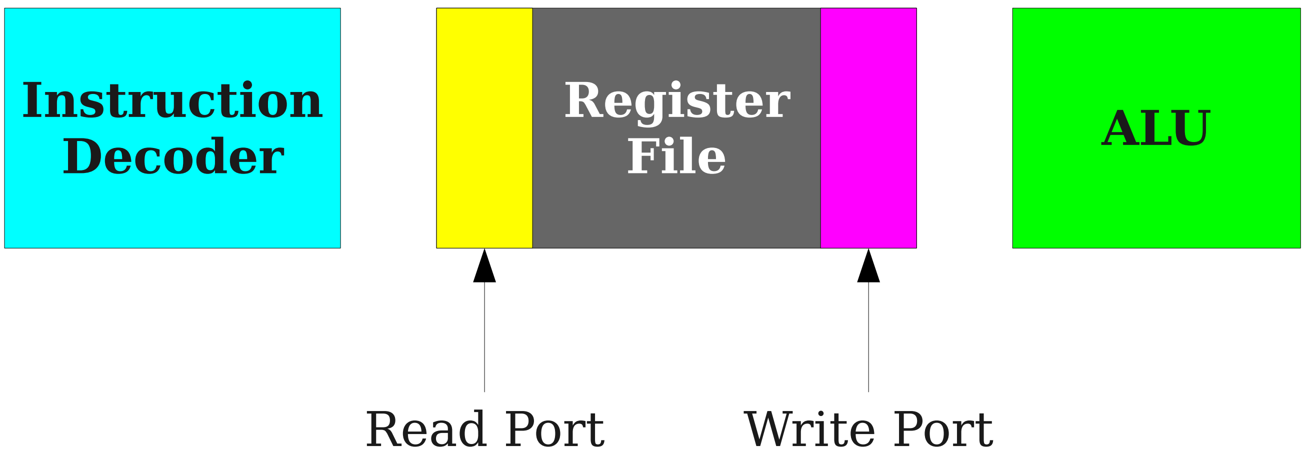
# Processor Pipelines

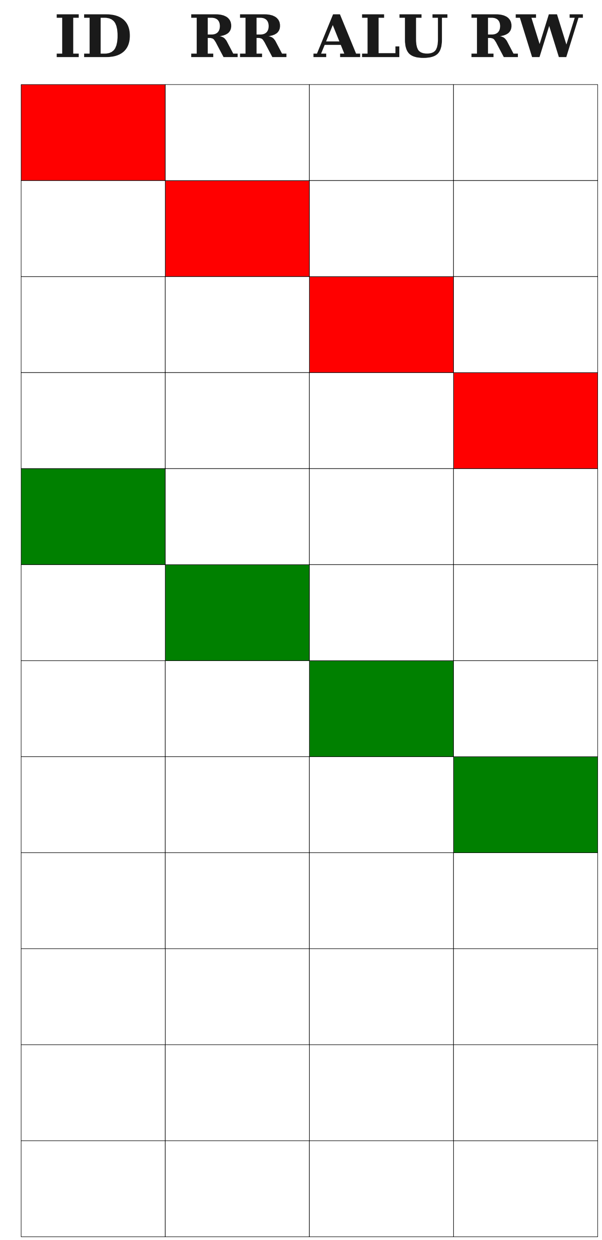



```
add $t2, $t0, $t1    # $t2 = $t0 + $t1
add $t5, $t3, $t4    # $t5 = $t3 + $t4
add $t8, $t6, $t7    # $t8 = $t6 + $t7
```

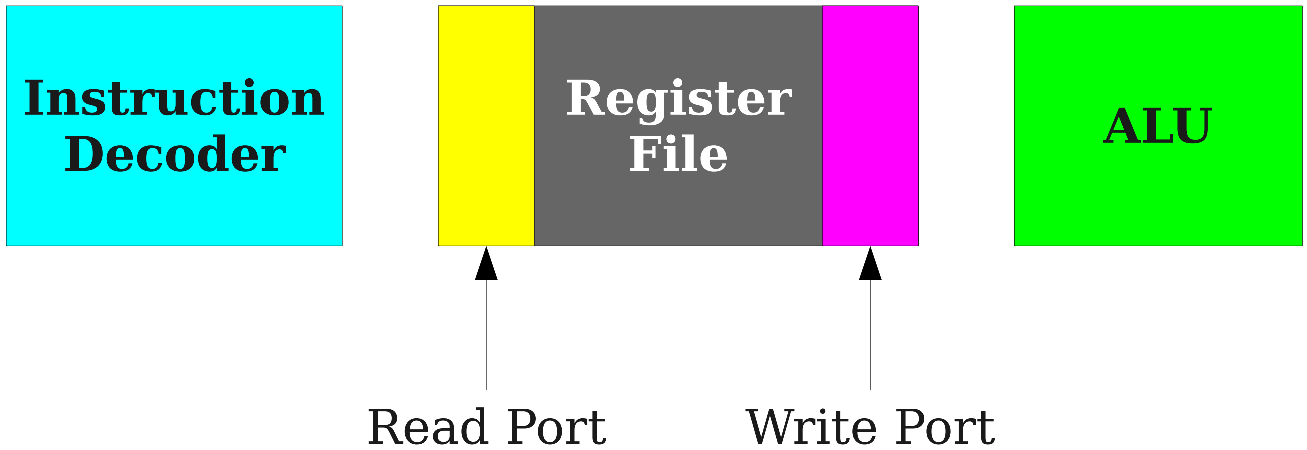
# Processor Pipelines



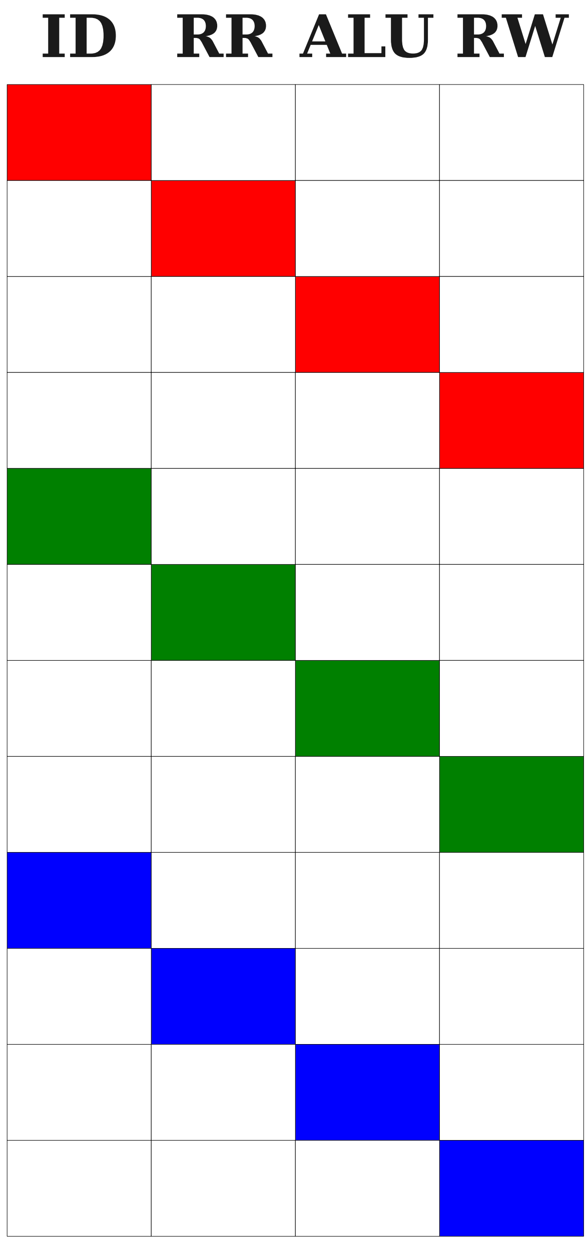




```
add $t2, $t0, $t1     # $t2 = $t0 + $t1
add $t5, $t3, $t4     # $t5 = $t3 + $t4
add $t8, $t6, $t7     # $t8 = $t6 + $t7
```

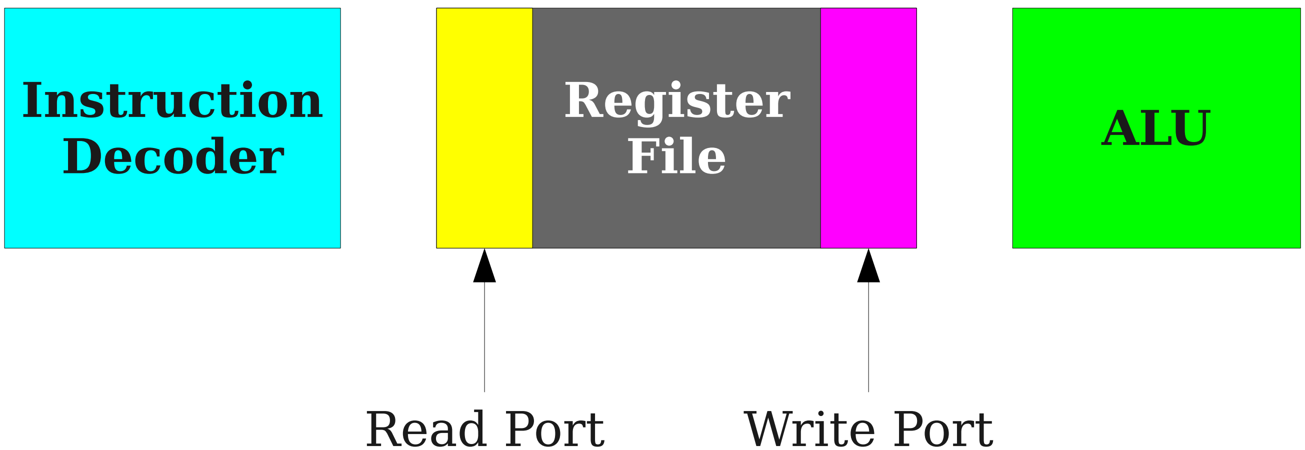
# Processor Pipelines

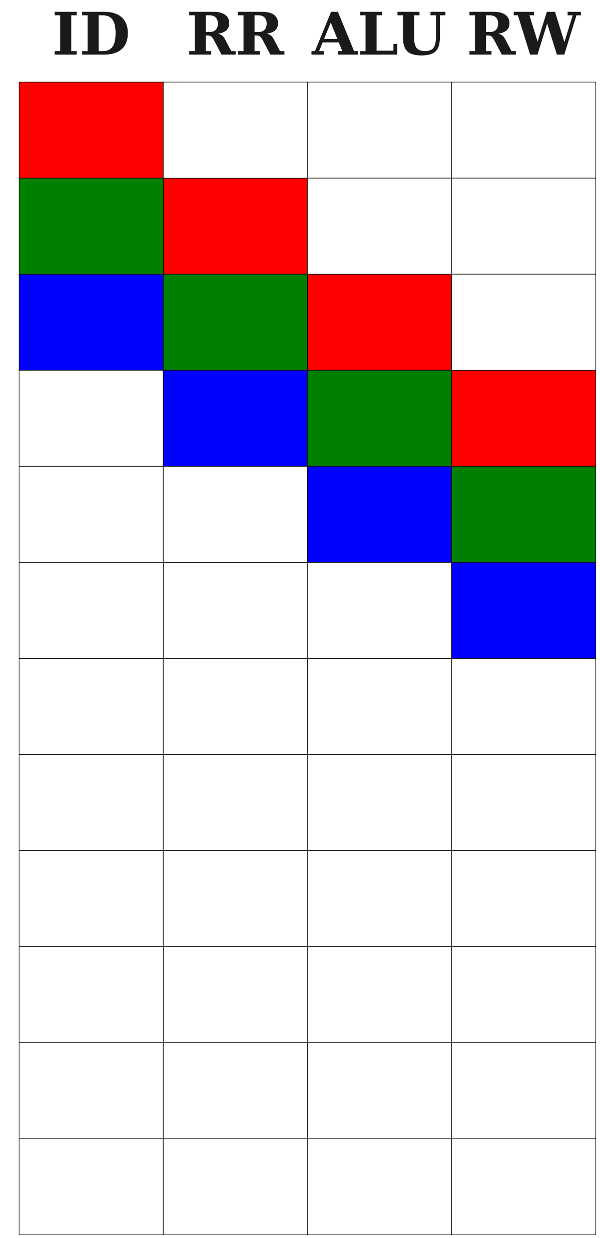



```
add $t2, $t0, $t1    # $t2 = $t0 + $t1
add $t5, $t3, $t4    # $t5 = $t3 + $t4
add $t8, $t6, $t7    # $t8 = $t6 + $t7
```



# Processor Pipelines

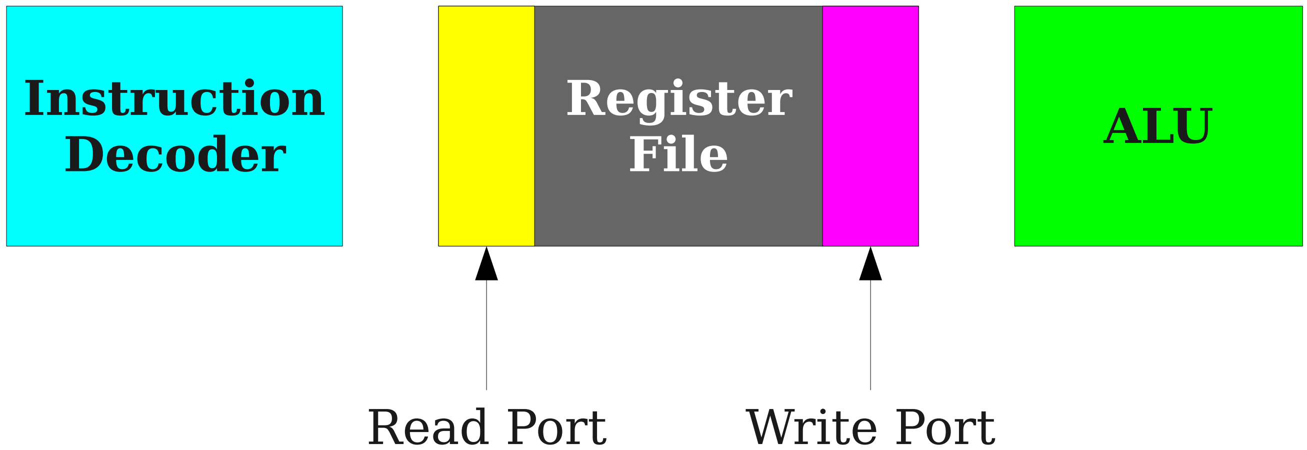


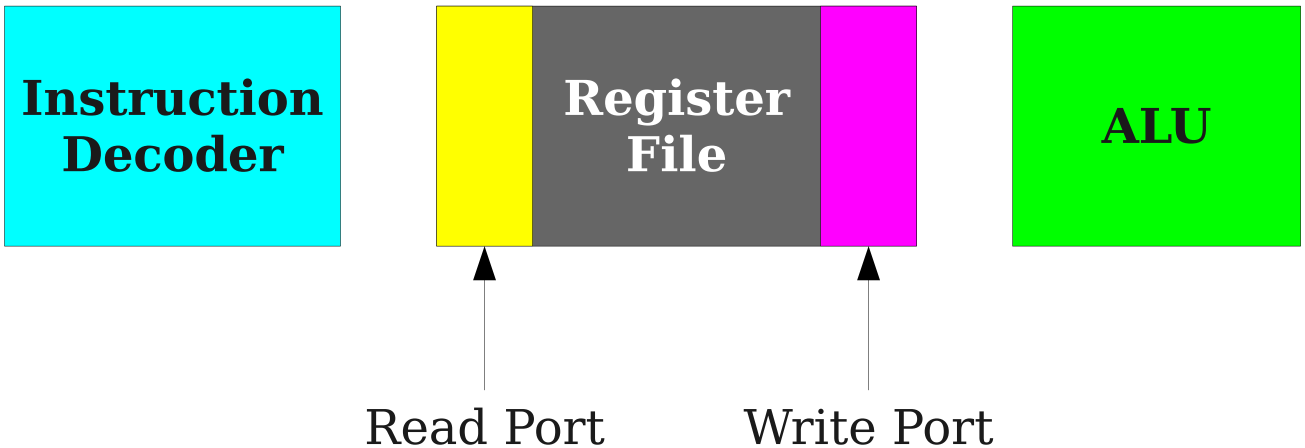

```
add $t2, $t0, $t1    # $t2 = $t0 + $t1
add $t5, $t3, $t4    # $t5 = $t3 + $t4
add $t8, $t6, $t7    # $t8 = $t6 + $t7
```

# Pipeline Hazards

# Pipeline Hazards

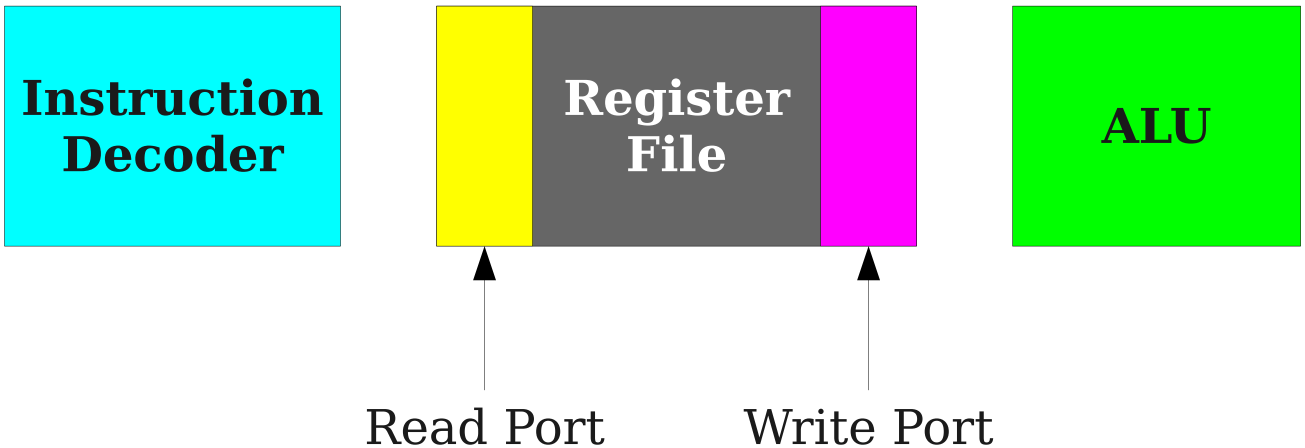


# Pipeline Hazards




```
add $t2, $t0, $t1   # $t2 = $t0 + $t1
add $t4, $t3, $t2   # $t5 = $t3 + $t2
add $t7, $t5, $t6   # $t7 = $t5 + $t6
add $t0, $t0, $t7   # $t0 = $t0 + $t7
```

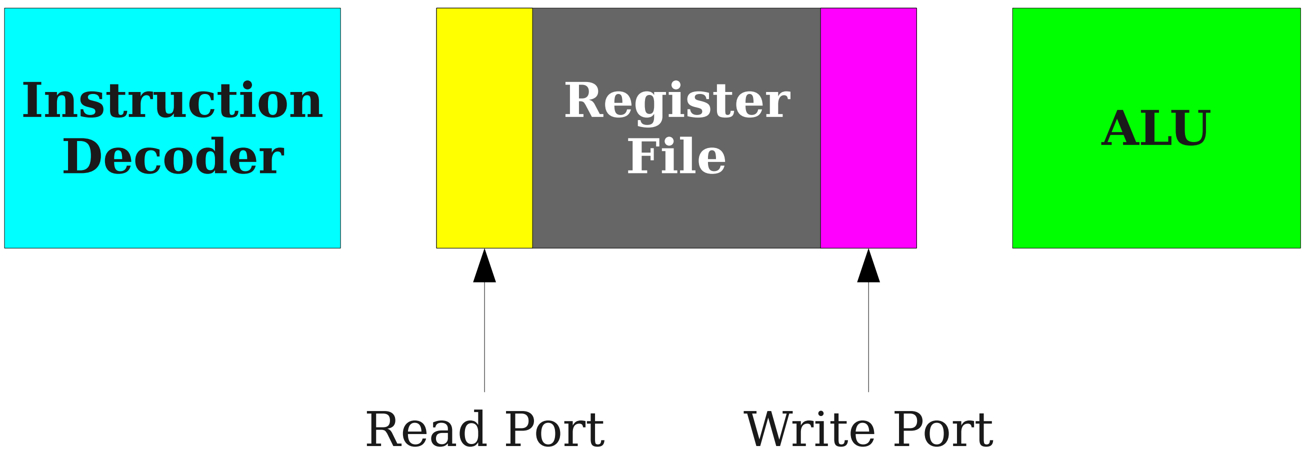
# Pipeline Hazards

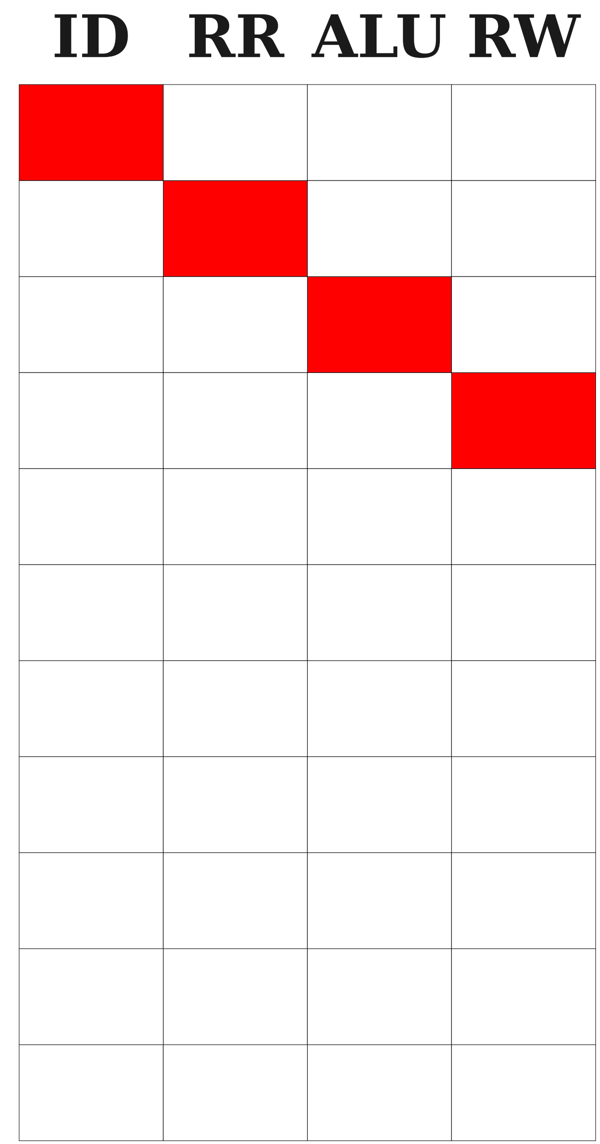



| ID | RR | ALU | RW |
|---|---|---|---|
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

```
add $t2, $t0, $t1    # $t2 = $t0 + $t1
add $t4, $t3, $t2    # $t5 = $t3 + $t2
add $t7, $t5, $t6    # $t7 = $t5 + $t6
add $t0, $t0, $t7    # $t0 = $t0 + $t7
```

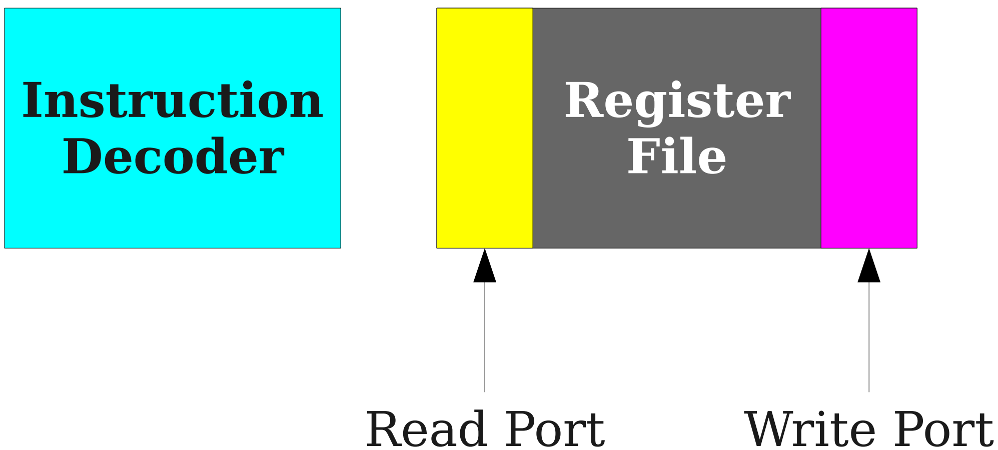
# Pipeline Hazards

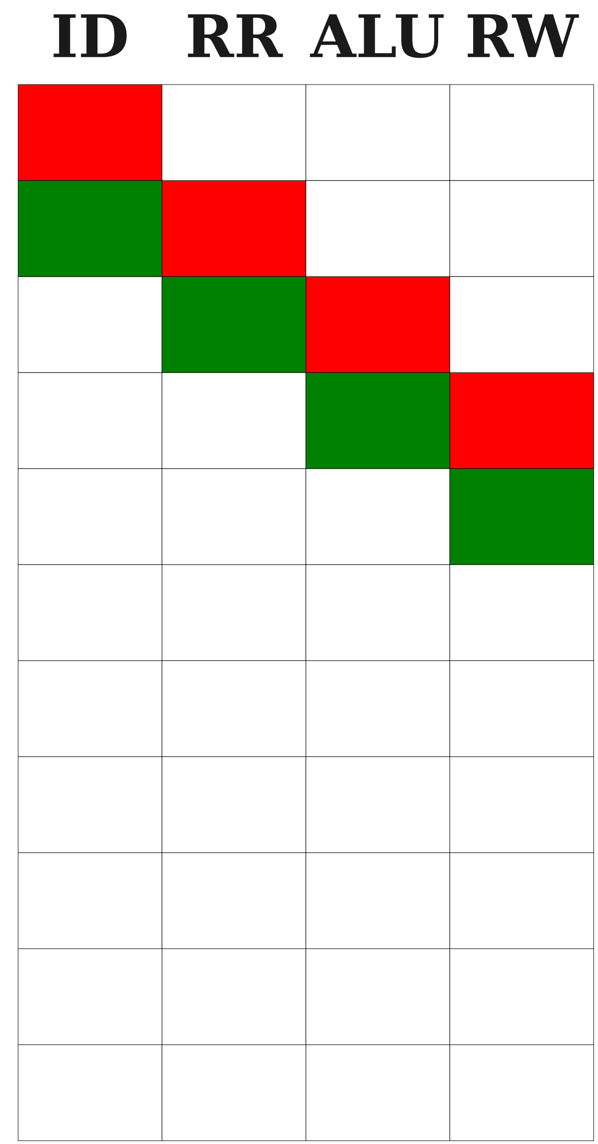



```
add $t2, $t0, $t1    # $t2 = $t0 + $t1
add $t4, $t3, $t2    # $t5 = $t3 + $t2
add $t7, $t5, $t6    # $t7 = $t5 + $t6
add $t0, $t0, $t7    # $t0 = $t0 + $t7
```

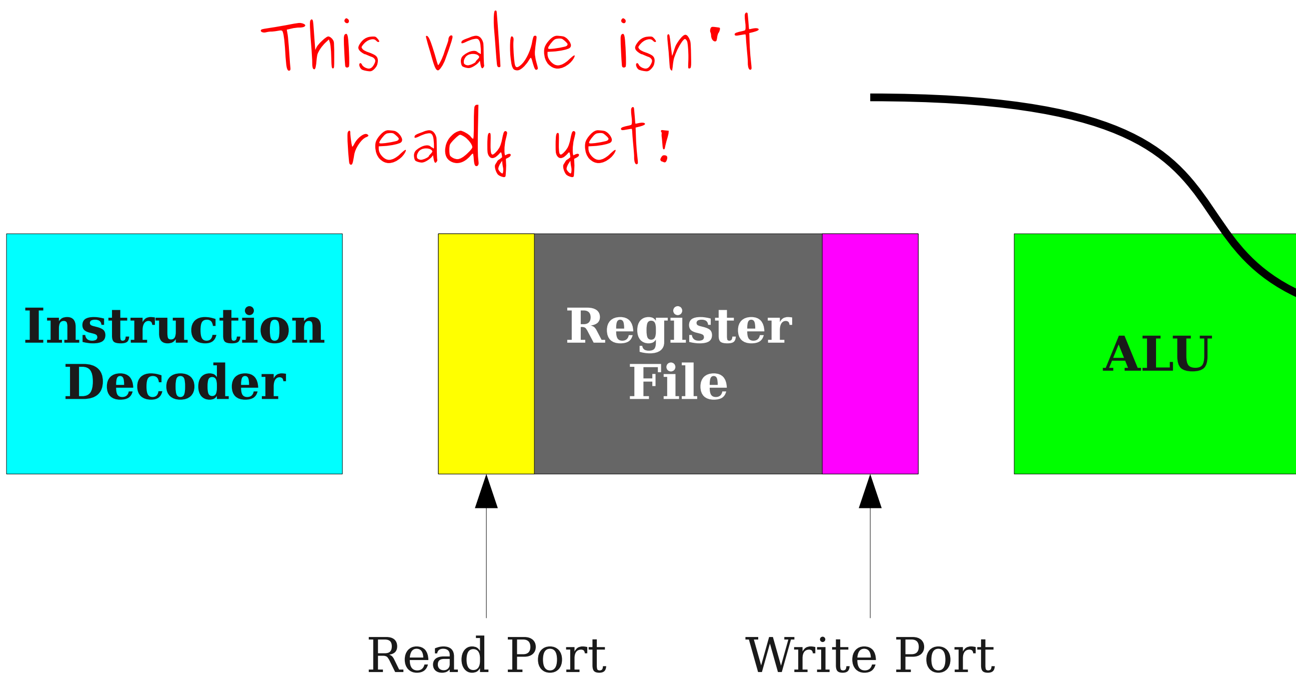
# Pipeline Hazards

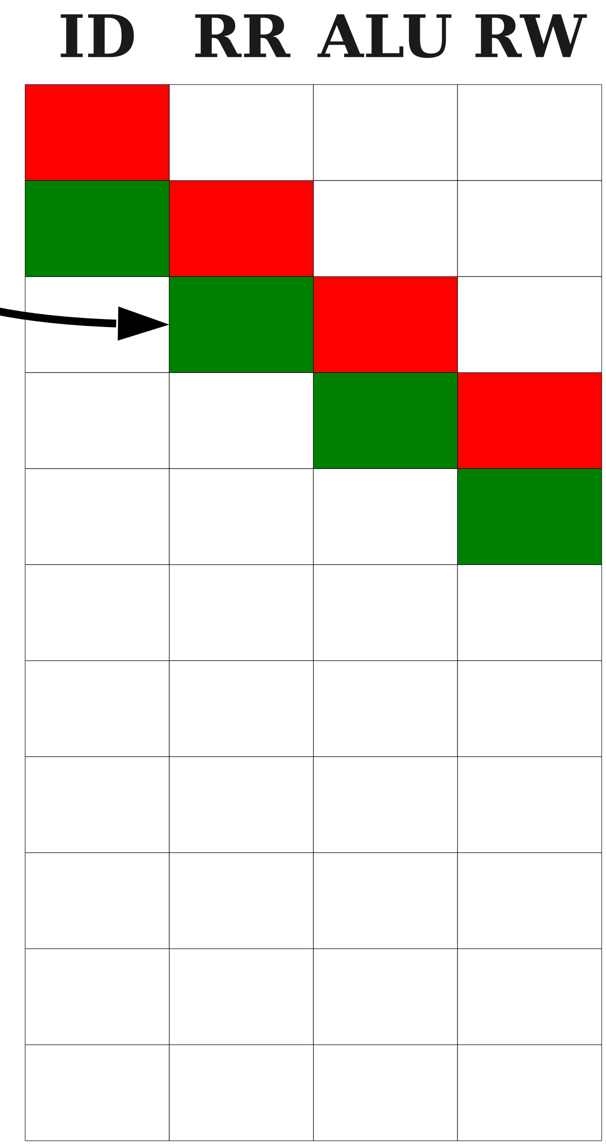



```
add $t2, $t0, $t1    # $t2 = $t0 + $t1
add $t4, $t3, $t2    # $t5 = $t3 + $t2
add $t7, $t5, $t6    # $t7 = $t5 + $t6
add $t0, $t0, $t7    # $t0 = $t0 + $t7
```

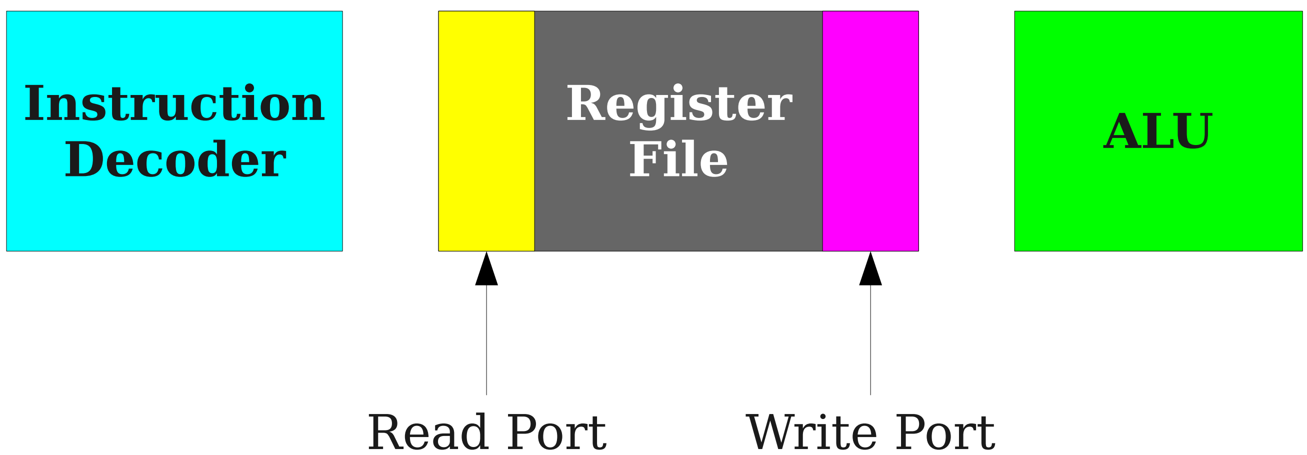
# Pipeline Hazards

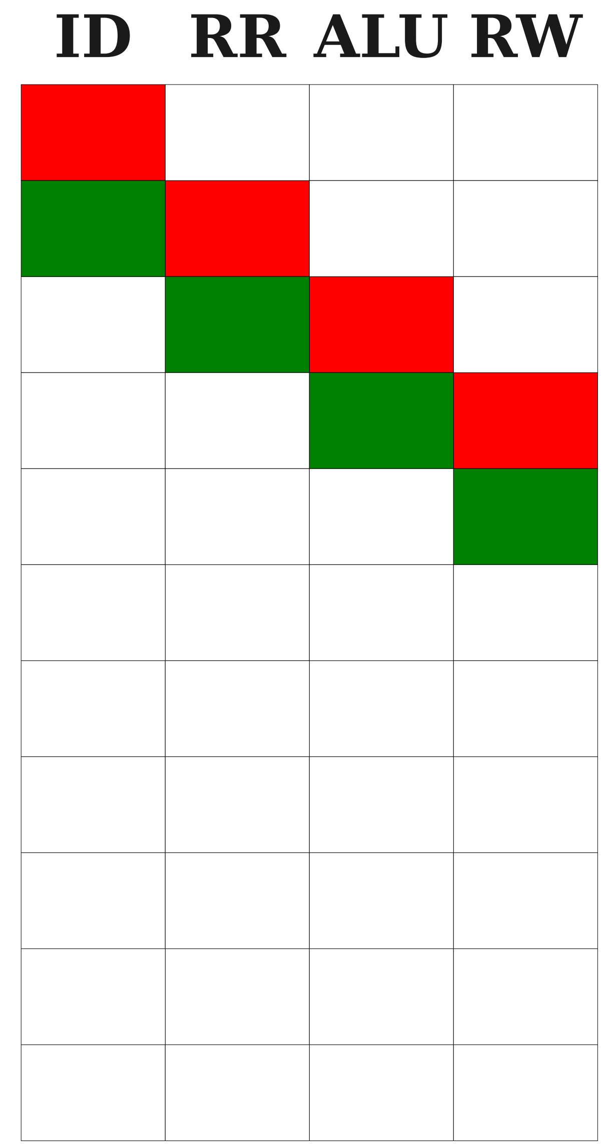



```
add $t2, $t0, $t1    # $t2 = $t0 + $t1
add $t4, $t3, $t2    # $t5 = $t3 + $t2
add $t7, $t5, $t6    # $t7 = $t5 + $t6
add $t0, $t0, $t7    # $t0 = $t0 + $t7
```

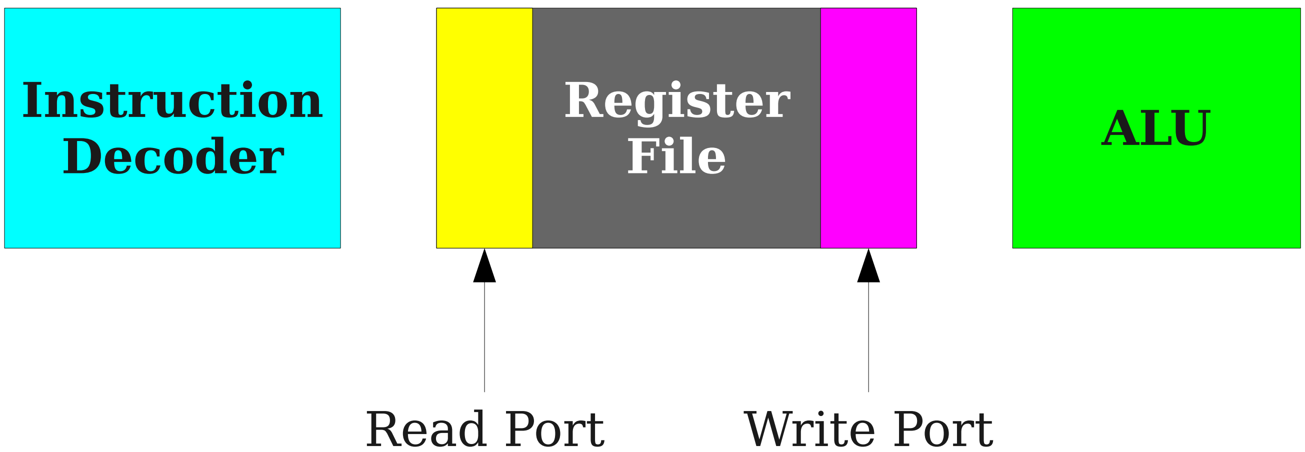
# Pipeline Hazards

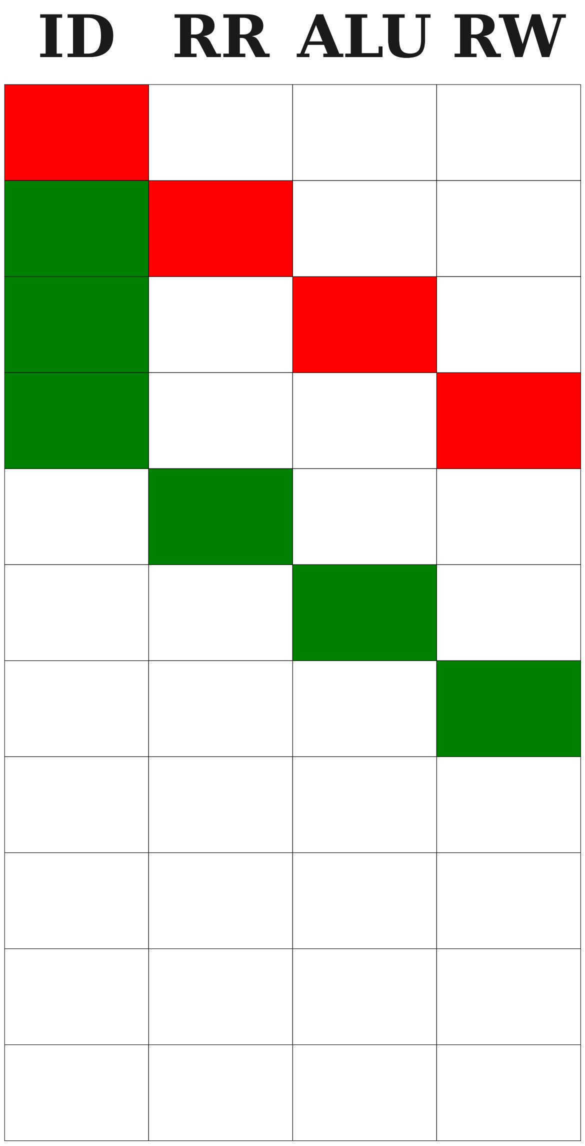



```
add $t2, $t0, $t1    # $t2 = $t0 + $t1
add $t4, $t3, $t2    # $t5 = $t3 + $t2
add $t7, $t5, $t6    # $t7 = $t5 + $t6
add $t0, $t0, $t7    # $t0 = $t0 + $t7
```

# Pipeline Hazards

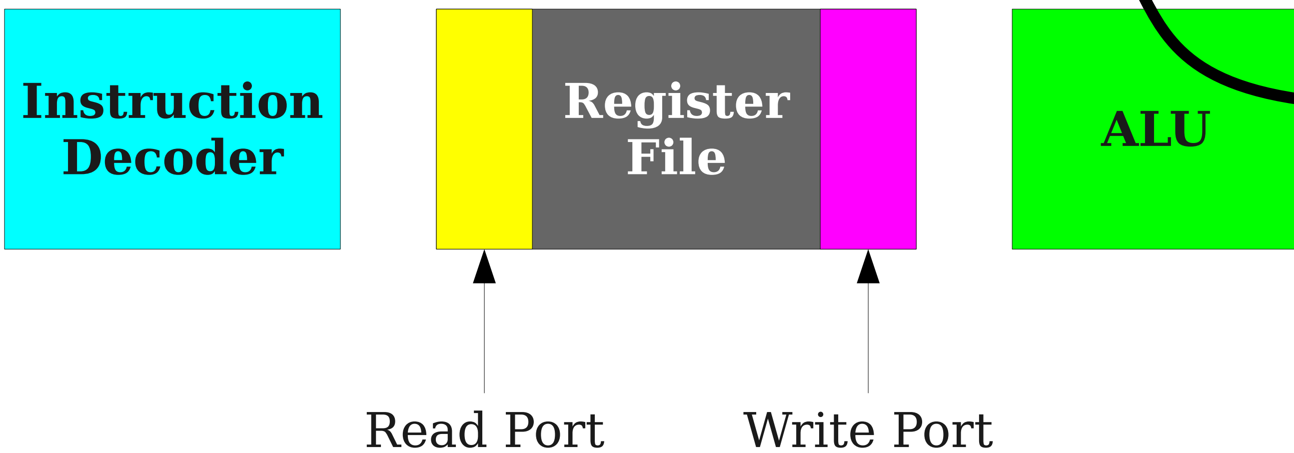


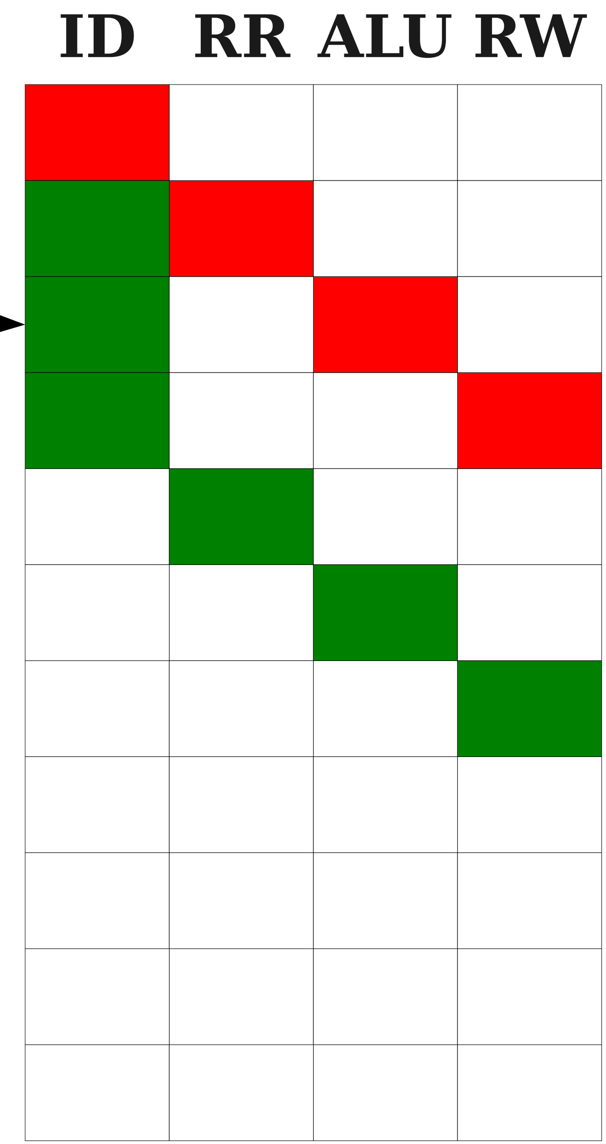




```
add $t2, $t0, $t1    # $t2 = $t0 + $t1
add $t4, $t3, $t2    # $t5 = $t3 + $t2
add $t7, $t5, $t6    # $t7 = $t5 + $t6
add $t0, $t0, $t7    # $t0 = $t0 + $t7
```

# Pipeline Hazards

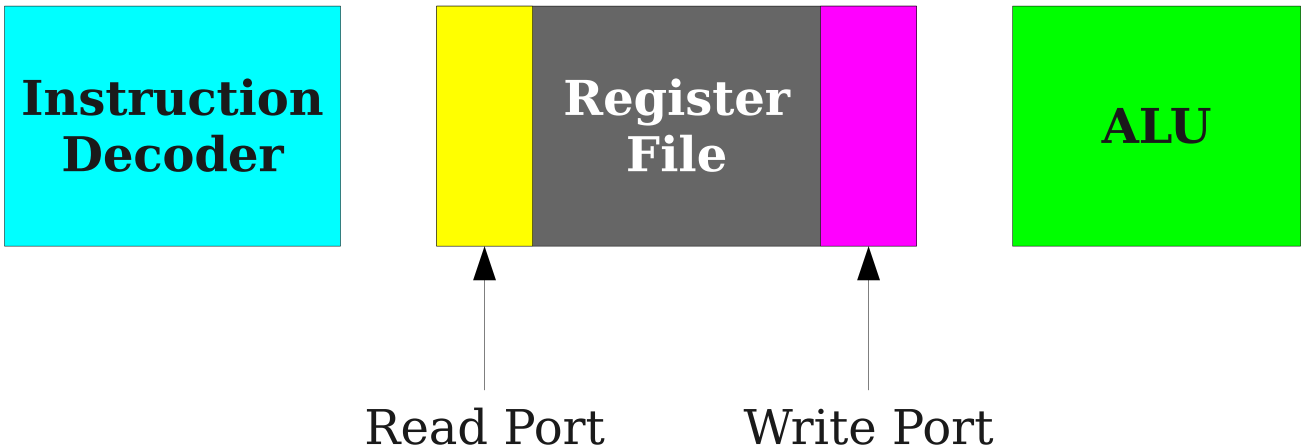
Stall pipeline until
value is ready

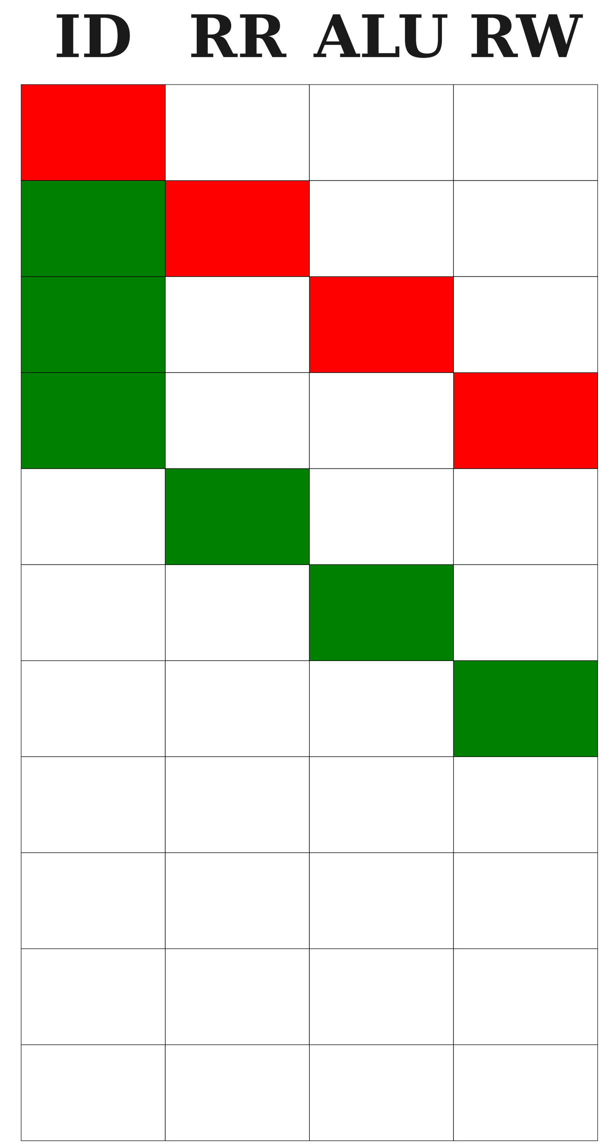



```
add $t2, $t0, $t1    # $t2 = $t0 + $t1
add $t4, $t3, $t2    # $t5 = $t3 + $t2
add $t7, $t5, $t6    # $t7 = $t5 + $t6
add $t0, $t0, $t7    # $t0 = $t0 + $t7
```

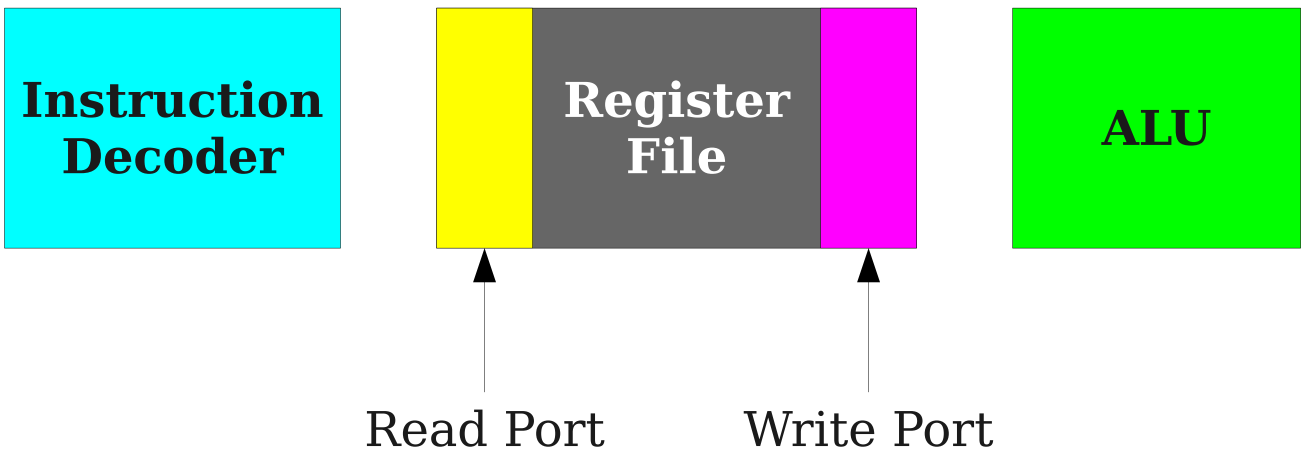
# Pipeline Hazards

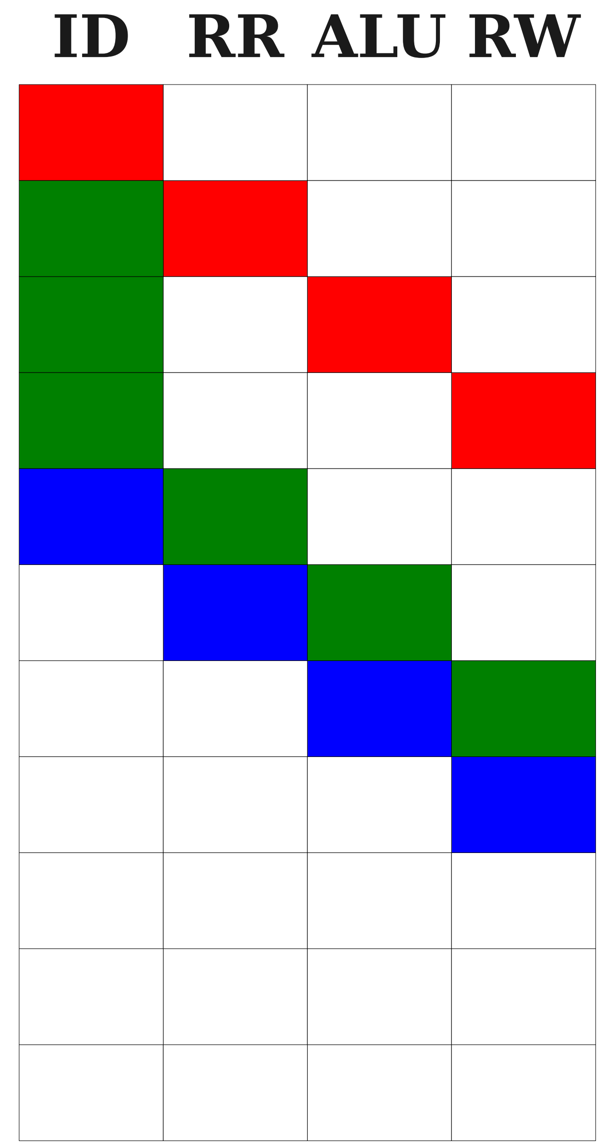



```
add $t2, $t0, $t1     # $t2 = $t0 + $t1
add $t4, $t3, $t2     # $t5 = $t3 + $t2
add $t7, $t5, $t6     # $t7 = $t5 + $t6
add $t0, $t0, $t7     # $t0 = $t0 + $t7
```

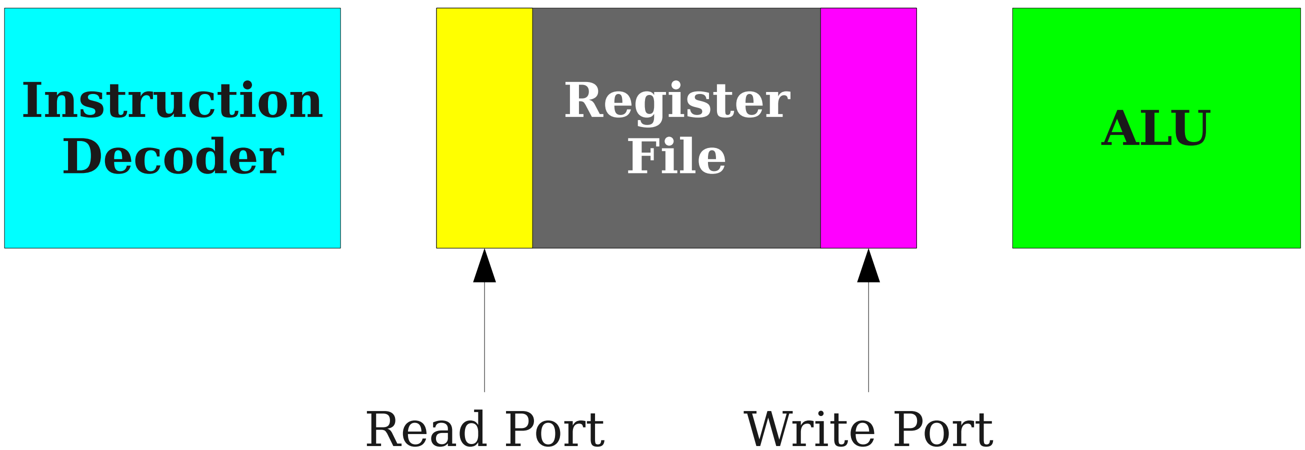
# Pipeline Hazards

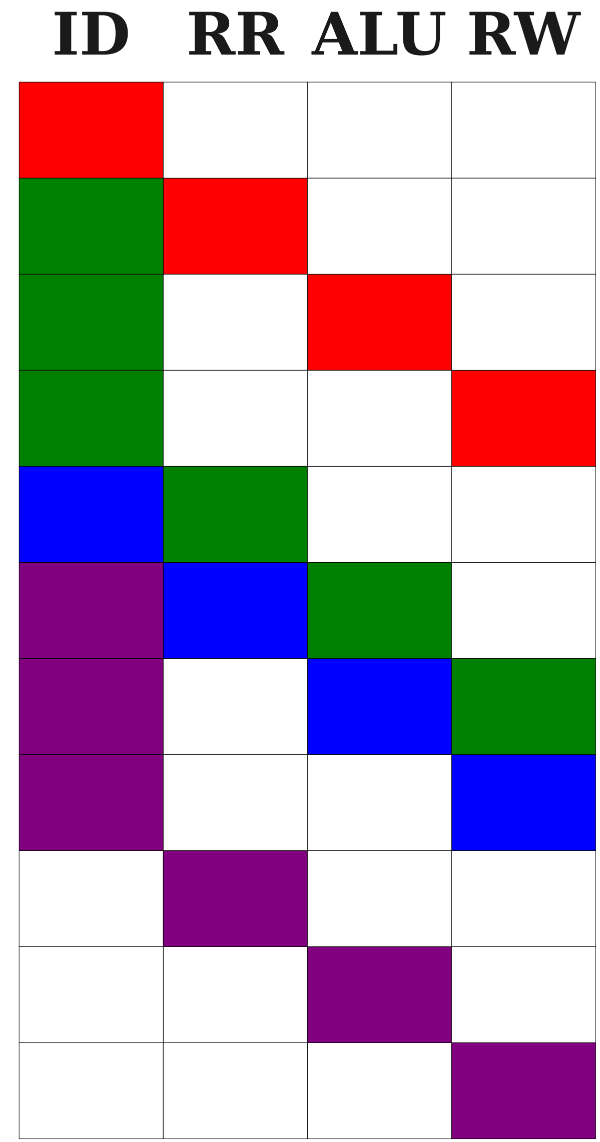



```
add $t2, $t0, $t1     # $t2 = $t0 + $t1
add $t4, $t3, $t2     # $t5 = $t3 + $t2
add $t7, $t5, $t6     # $t7 = $t5 + $t6
add $t0, $t0, $t7     # $t0 = $t0 + $t7
```

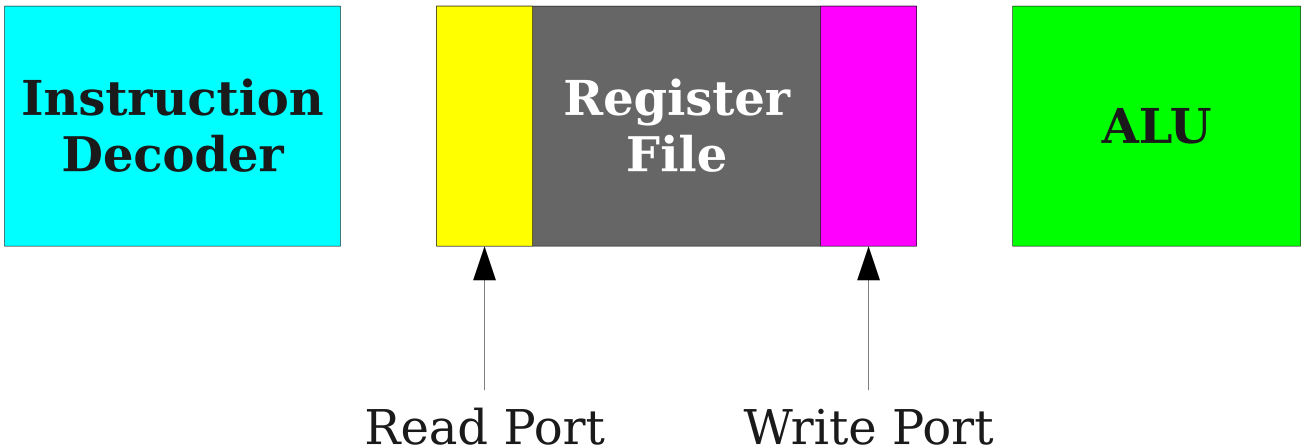
# Pipeline Hazards




```
add $t2, $t0, $t1    # $t2 = $t0 + $t1
add $t4, $t3, $t2    # $t5 = $t3 + $t2
add $t7, $t5, $t6    # $t7 = $t5 + $t6
add $t0, $t0, $t7    # $t0 = $t0 + $t7
```

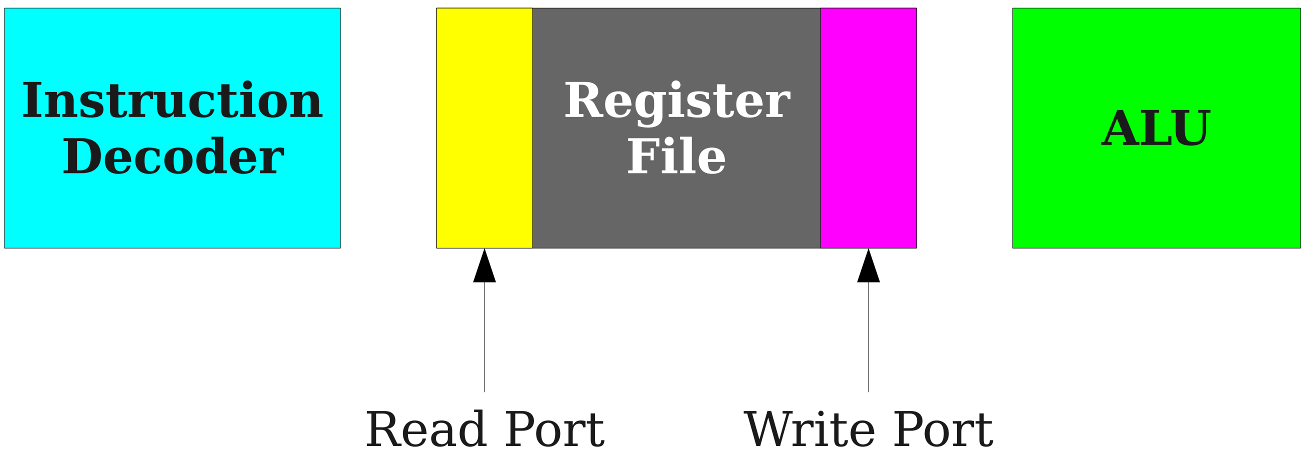
# Pipeline Hazards




```
add $t2, $t0, $t1    # $t2 = $t0 + $t1
add $t4, $t3, $t2    # $t5 = $t3 + $t2
add $t7, $t5, $t6    # $t7 = $t5 + $t6
add $t0, $t0, $t7    # $t0 = $t0 + $t7
```

| ID | RR | ALU | RW |
|---|---|---|---|
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

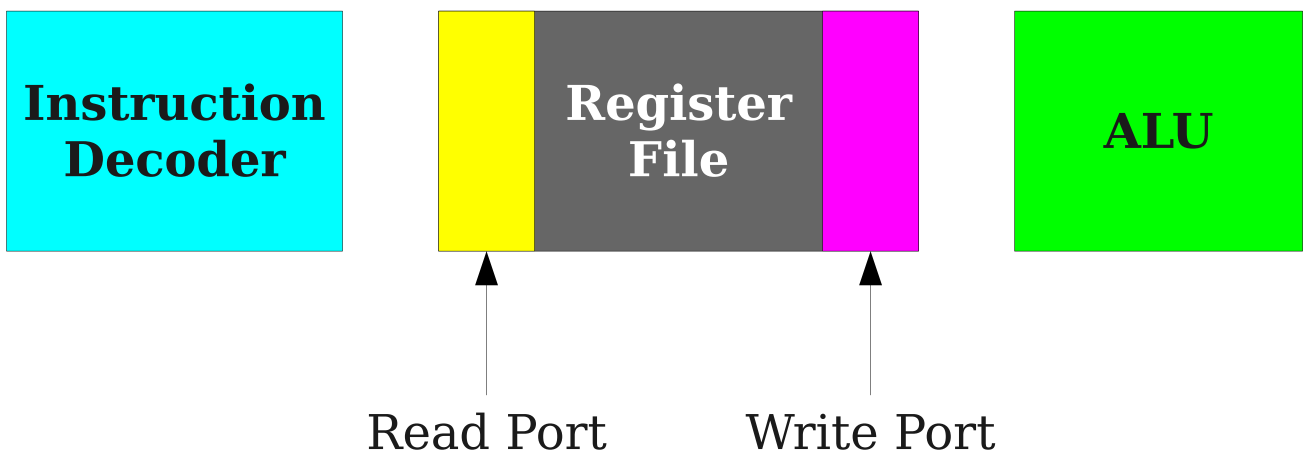
# Pipeline Hazards

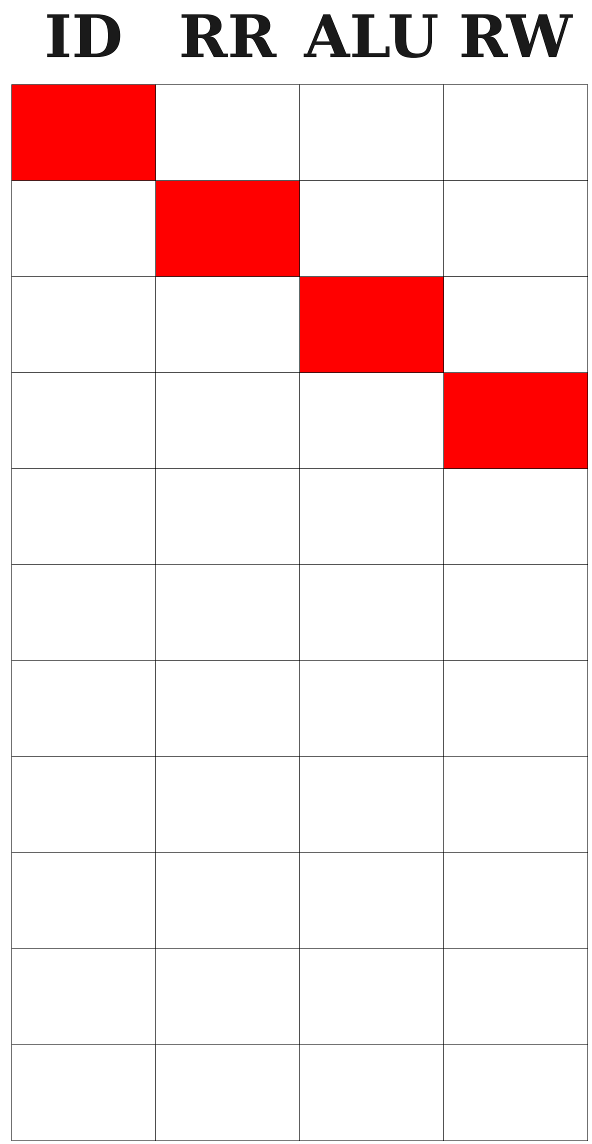



| ID | RR | ALU | RW |
|---|---|---|---|
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

```
add $t2, $t0, $t1    # $t2 = $t0 + $t1
add $t7, $t5, $t6    # $t7 = $t5 + $t6
add $t4, $t3, $t2    # $t5 = $t3 + $t2
add $t0, $t0, $t7    # $t0 = $t0 + $t7
```

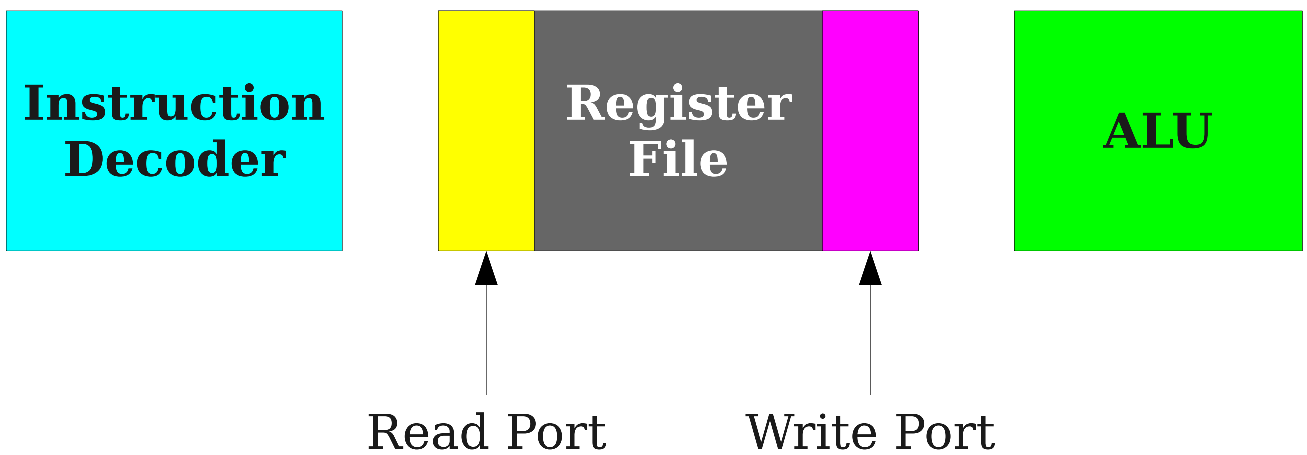
# Pipeline Hazards

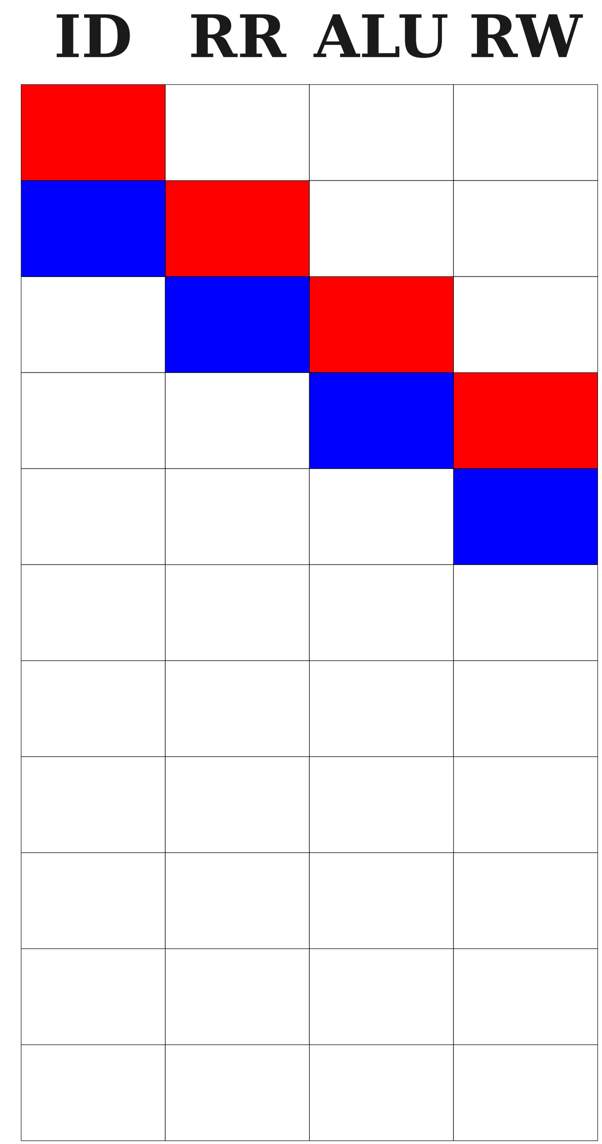



```
add $t2, $t0, $t1    # $t2 = $t0 + $t1
add $t7, $t5, $t6    # $t7 = $t5 + $t6
add $t4, $t3, $t2    # $t5 = $t3 + $t2
add $t0, $t0, $t7    # $t0 = $t0 + $t7
```

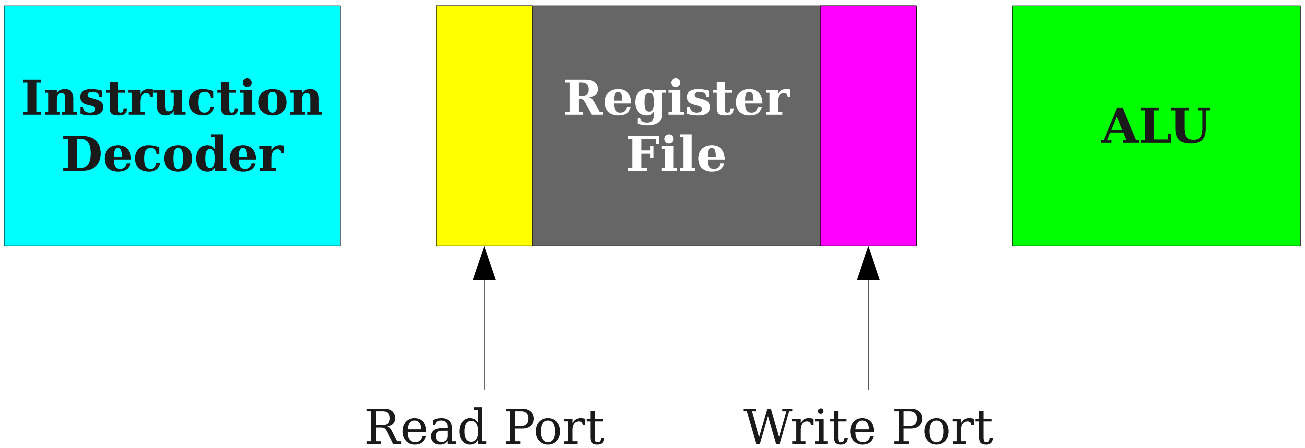
# Pipeline Hazards

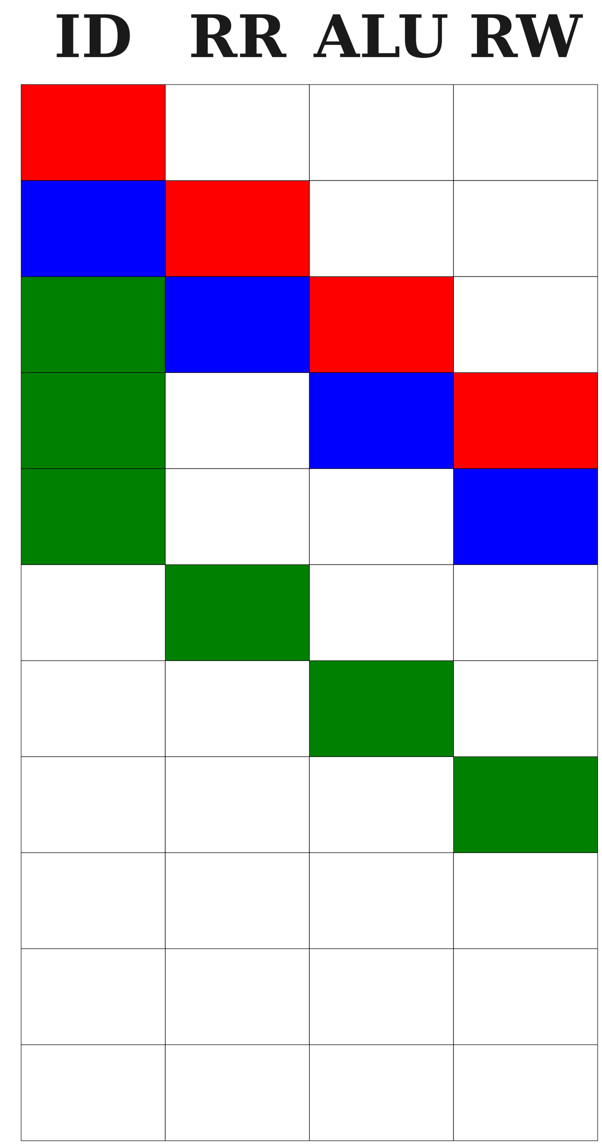



```
add $t2, $t0, $t1    # $t2 = $t0 + $t1
add $t7, $t5, $t6    # $t7 = $t5 + $t6
add $t4, $t3, $t2    # $t5 = $t3 + $t2
add $t0, $t0, $t7    # $t0 = $t0 + $t7
```

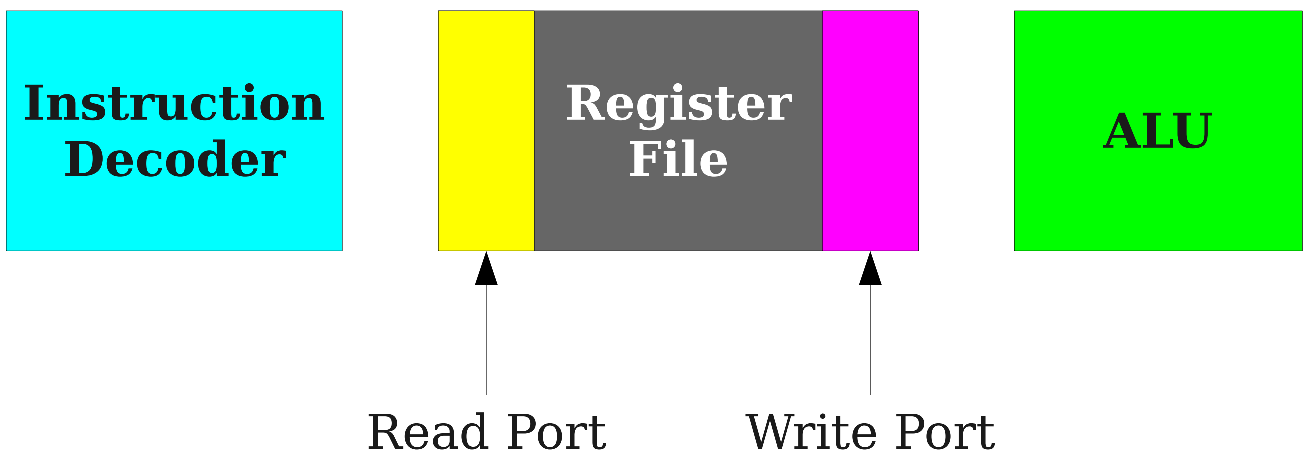
# Pipeline Hazards

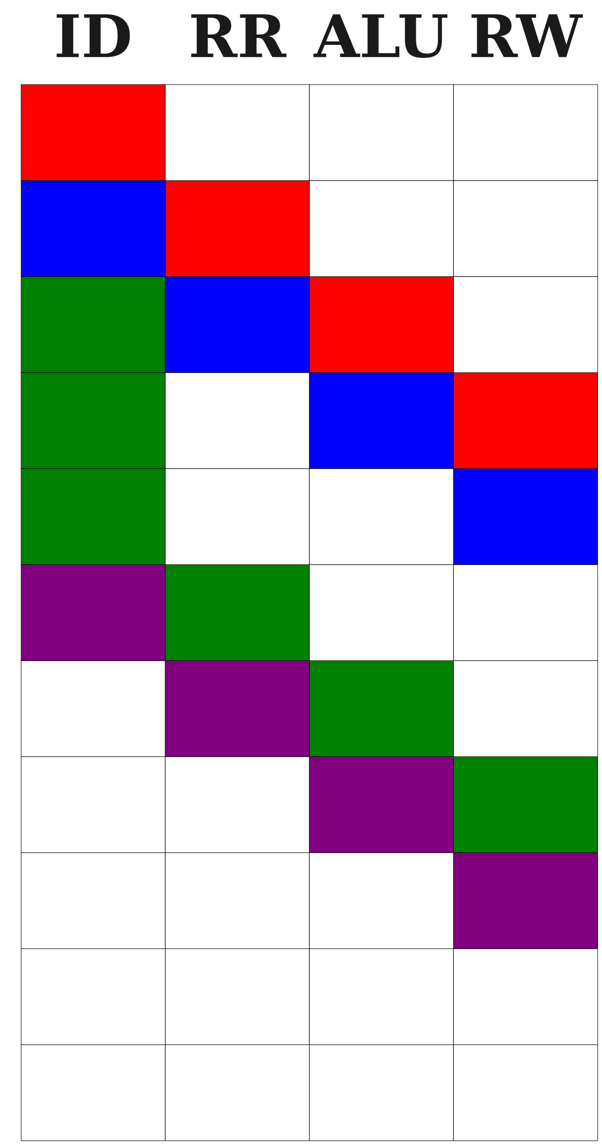



```
add $t2, $t0, $t1    # $t2 = $t0 + $t1
add $t7, $t5, $t6    # $t7 = $t5 + $t6
add $t4, $t3, $t2    # $t5 = $t3 + $t2
add $t0, $t0, $t7    # $t0 = $t0 + $t7
```

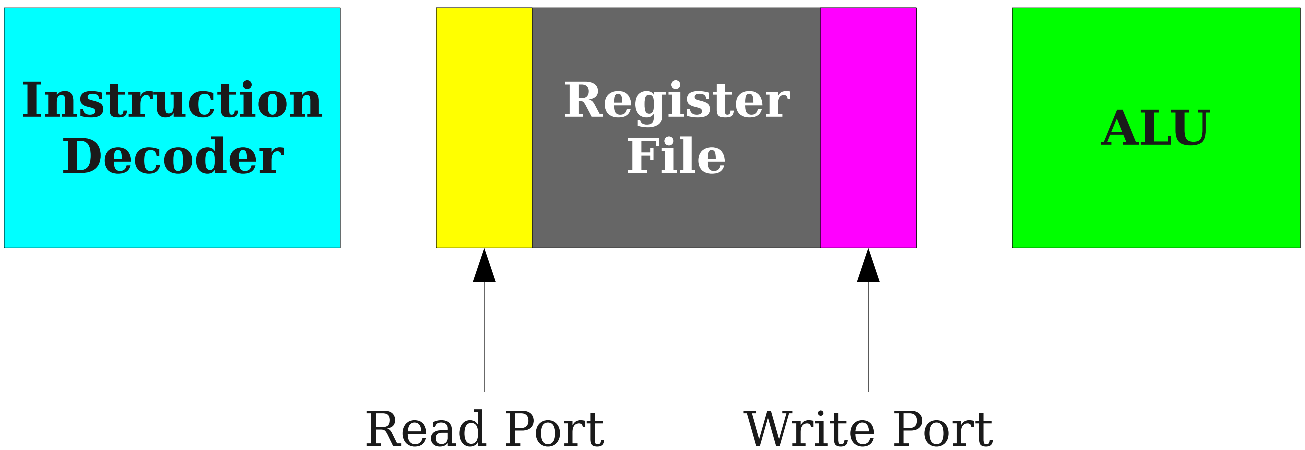
# Pipeline Hazards

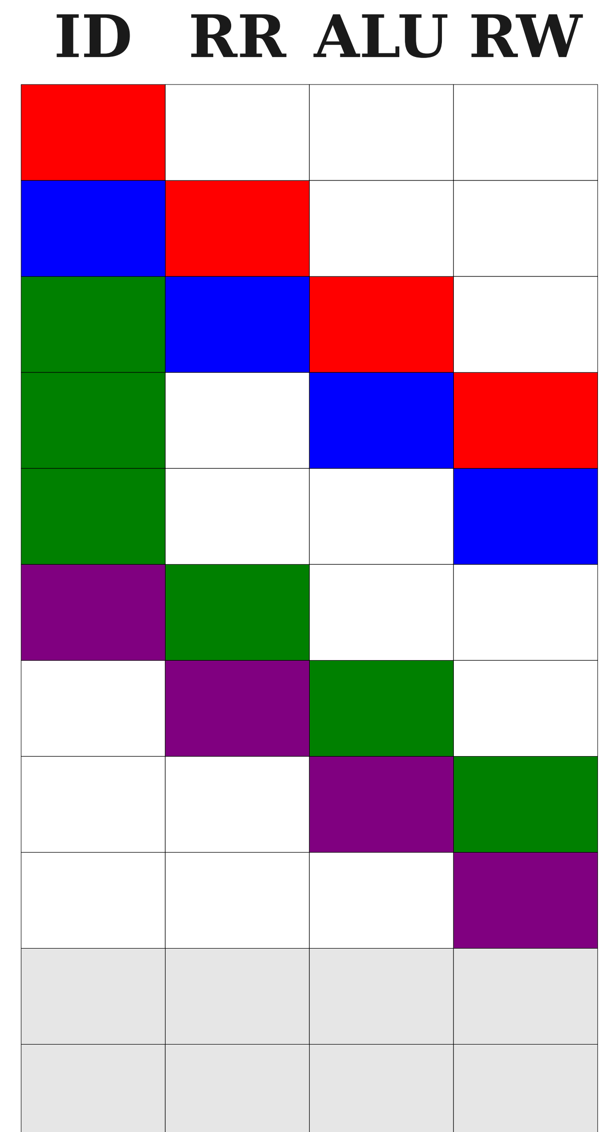



```
add $t2, $t0, $t1   # $t2 = $t0 + $t1
add $t7, $t5, $t6   # $t7 = $t5 + $t6
add $t4, $t3, $t2   # $t5 = $t3 + $t2
add $t0, $t0, $t7   # $t0 = $t0 + $t7
```

# Pipeline Hazards




```
add $t2, $t0, $t1    # $t2 = $t0 + $t1
add $t7, $t5, $t6    # $t7 = $t5 + $t6
add $t4, $t3, $t2    # $t5 = $t3 + $t2
add $t0, $t0, $t7    # $t0 = $t0 + $t7
```

Two clock cycles faster!

# Instruction Scheduling

- Because of processor pipelining, the order in which instructions are executed can impact performance.
- **Instruction scheduling** is the reordering or insertion of machine instructions to increase performance.
- All good optimizing compilers have some sort of instruction scheduling support.

# Data Dependencies

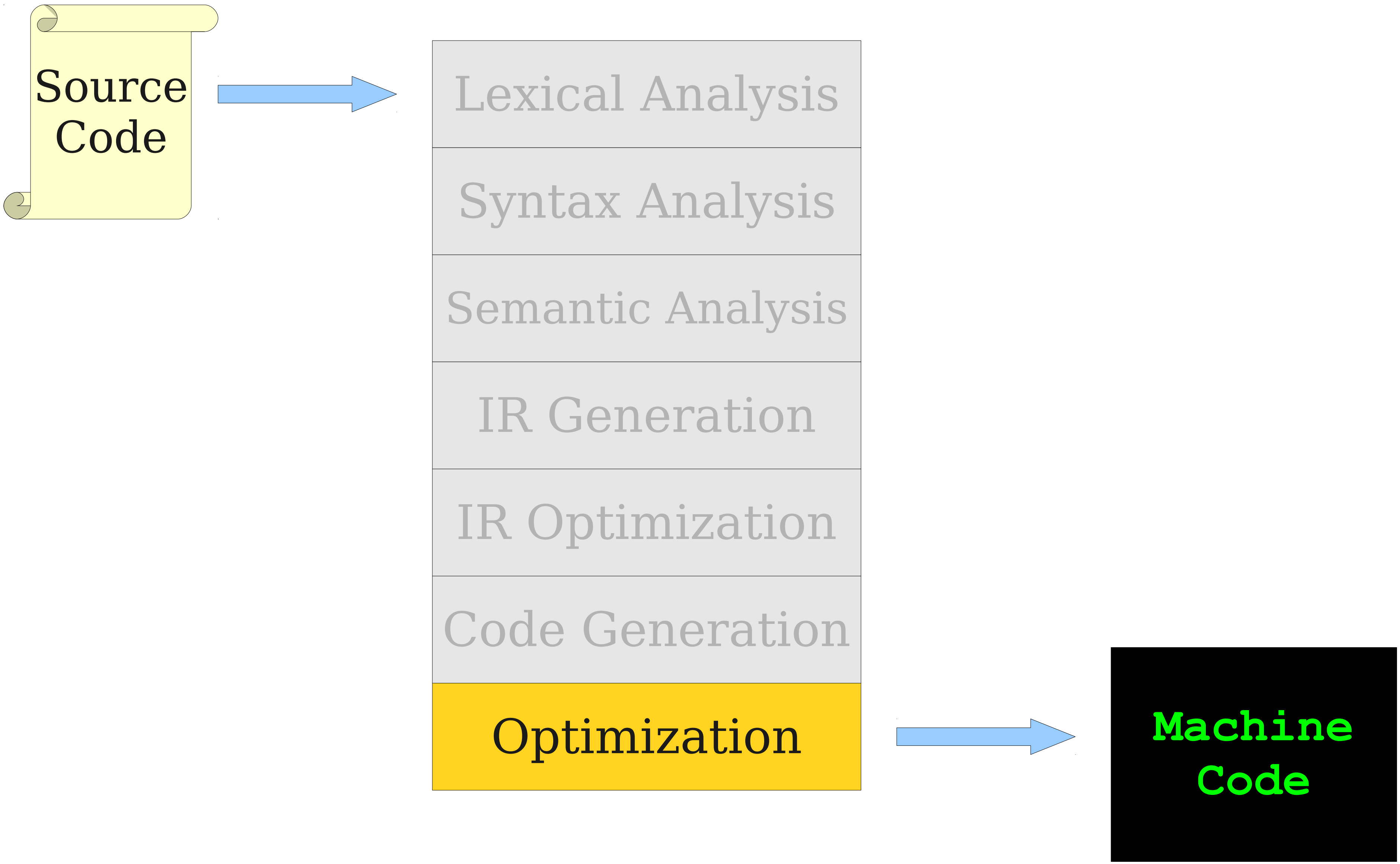
- A **data dependency** in machine code is a set of instructions whose behavior depends on one another.
- Intuitively, a set of instructions that cannot be reordered around each other.
- Three types of data dependencies:

| **Read-after-Write (RAW)** | **Write-after-Read (WAR)** | **Write-after-Write (WAW)** |
|---|---|---|
| x = ...<br>... = x | ... = x<br>x = ... | x = ...<br>x = ... |

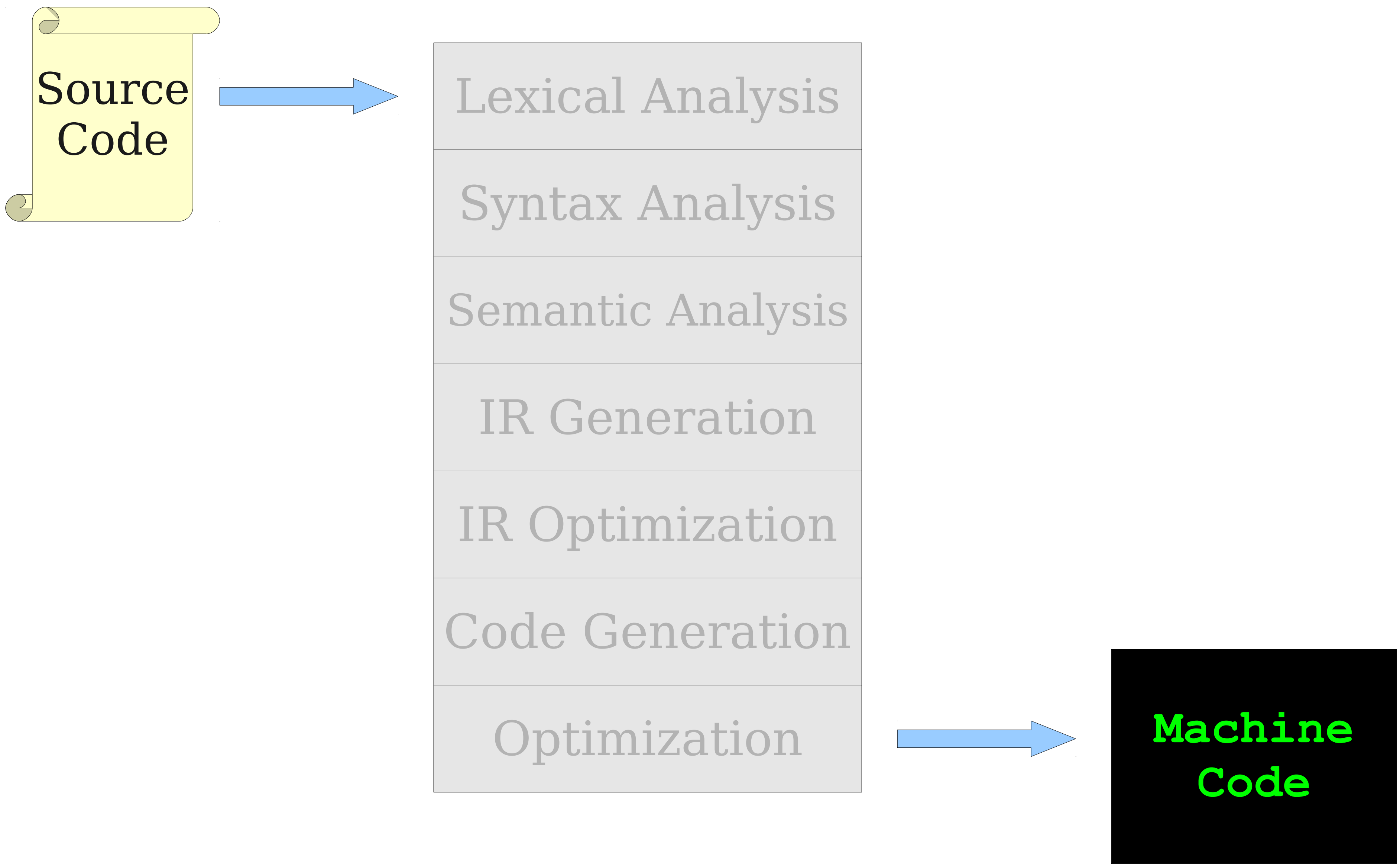
# Summary

- **Instruction scheduling** optimizations try to take advantage of the processor pipeline.
- **Locality** optimizations try to take advantage of cache behavior.
- **Parallelism** optimizations try to take advantage of multicore machines.
- There are *many more* optimizations out there!

# Where We Are


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

# Where We Are



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