

# 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

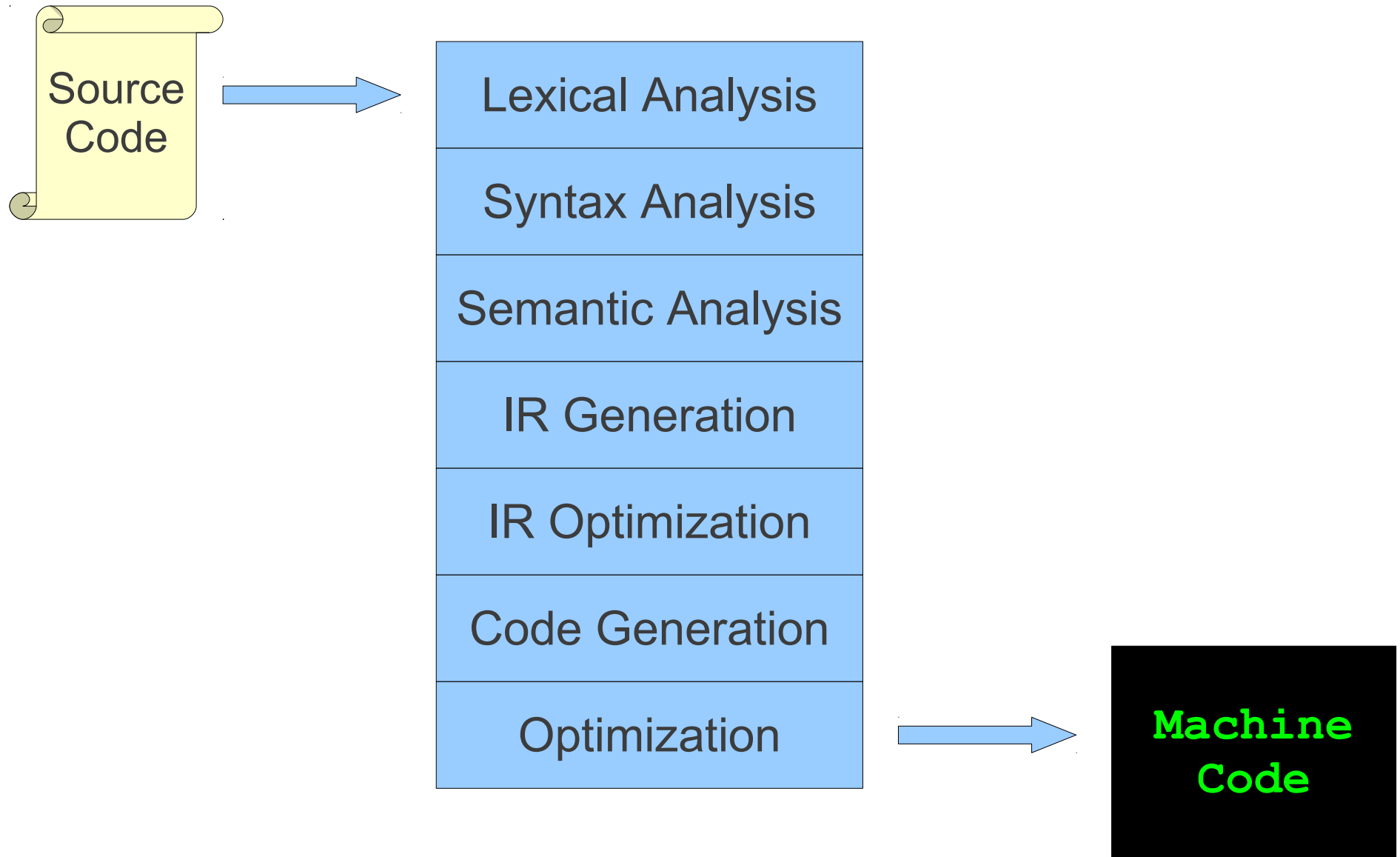
- First, there was nothing.
- Then, there was machine code.
- Then, there were assembly languages.
- Programming expensive; 50% of costs for machines went into programming.

How does a compiler work?

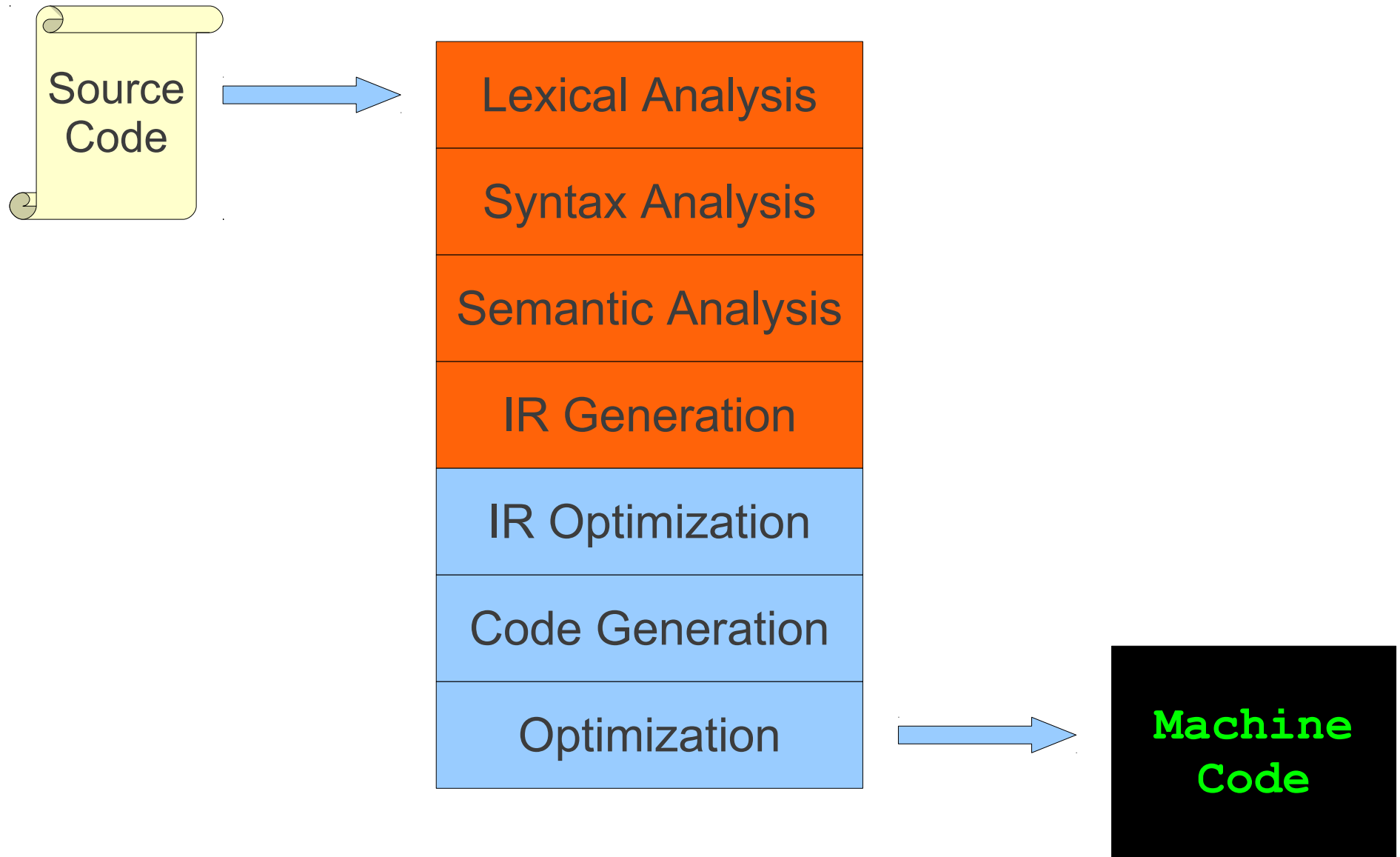
# From Description to Implementation

- **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

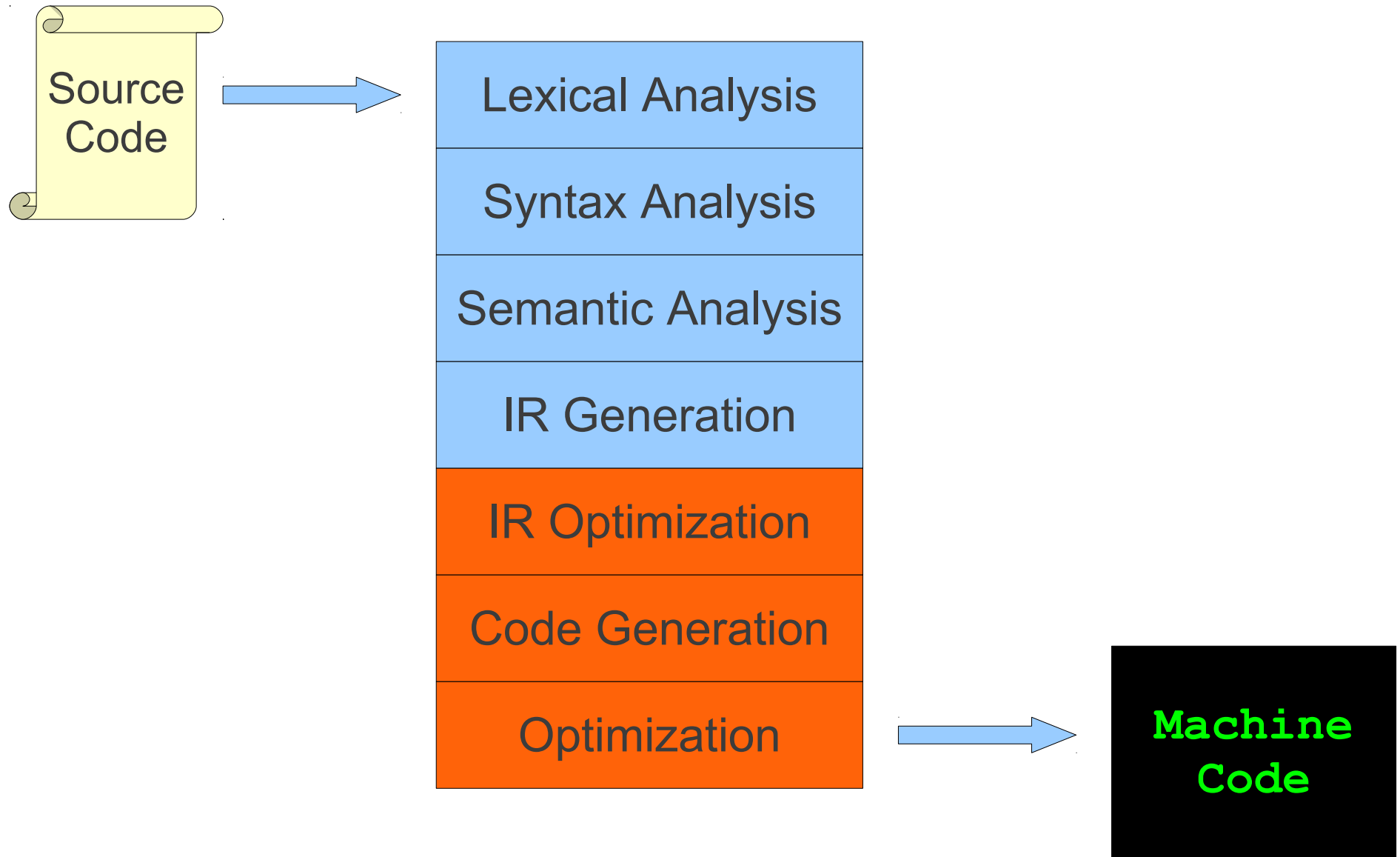


# The Structure of a Modern Compiler





# 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;  
}
```

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

Lexical Analysis

Syntax Analysis

Semantic Analysis

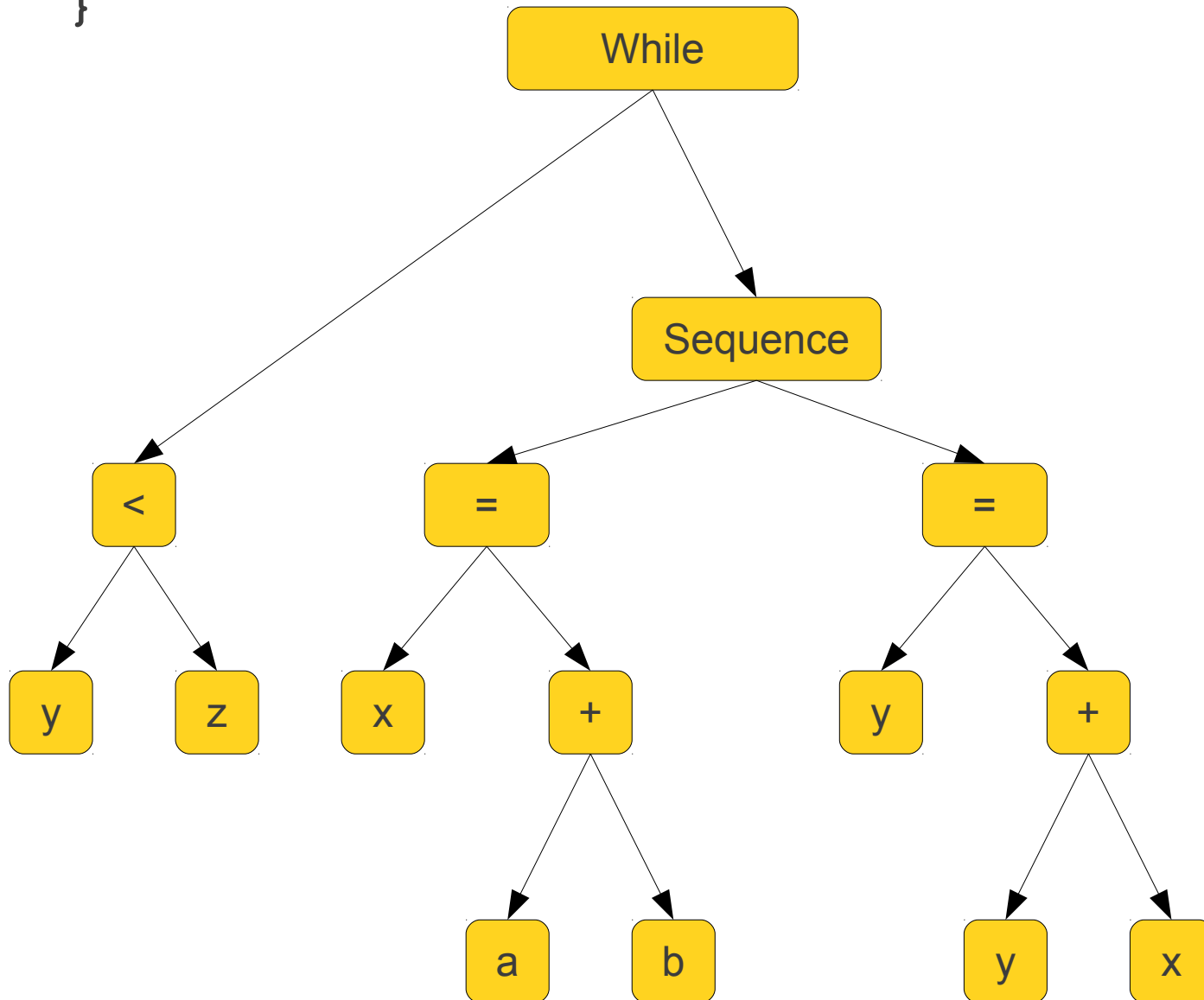
IR Generation

IR Optimization

Code Generation

Optimization

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



Lexical Analysis

Syntax Analysis

Semantic Analysis

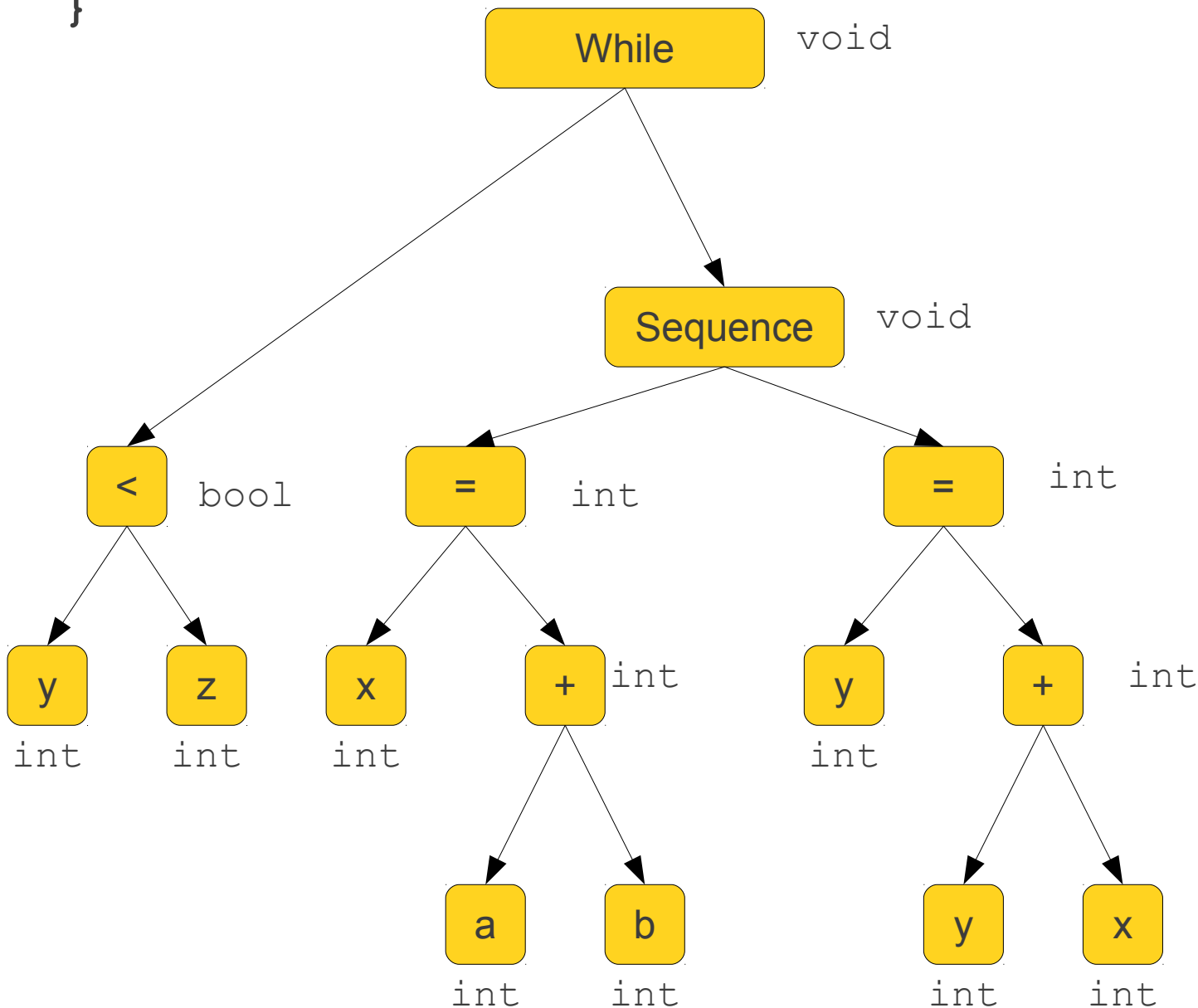
IR Generation

IR Optimization

Code Generation

Optimization

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



Lexical Analysis

Syntax Analysis

Semantic Analysis

IR Generation

IR Optimization

Code Generation

Optimization

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

```
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

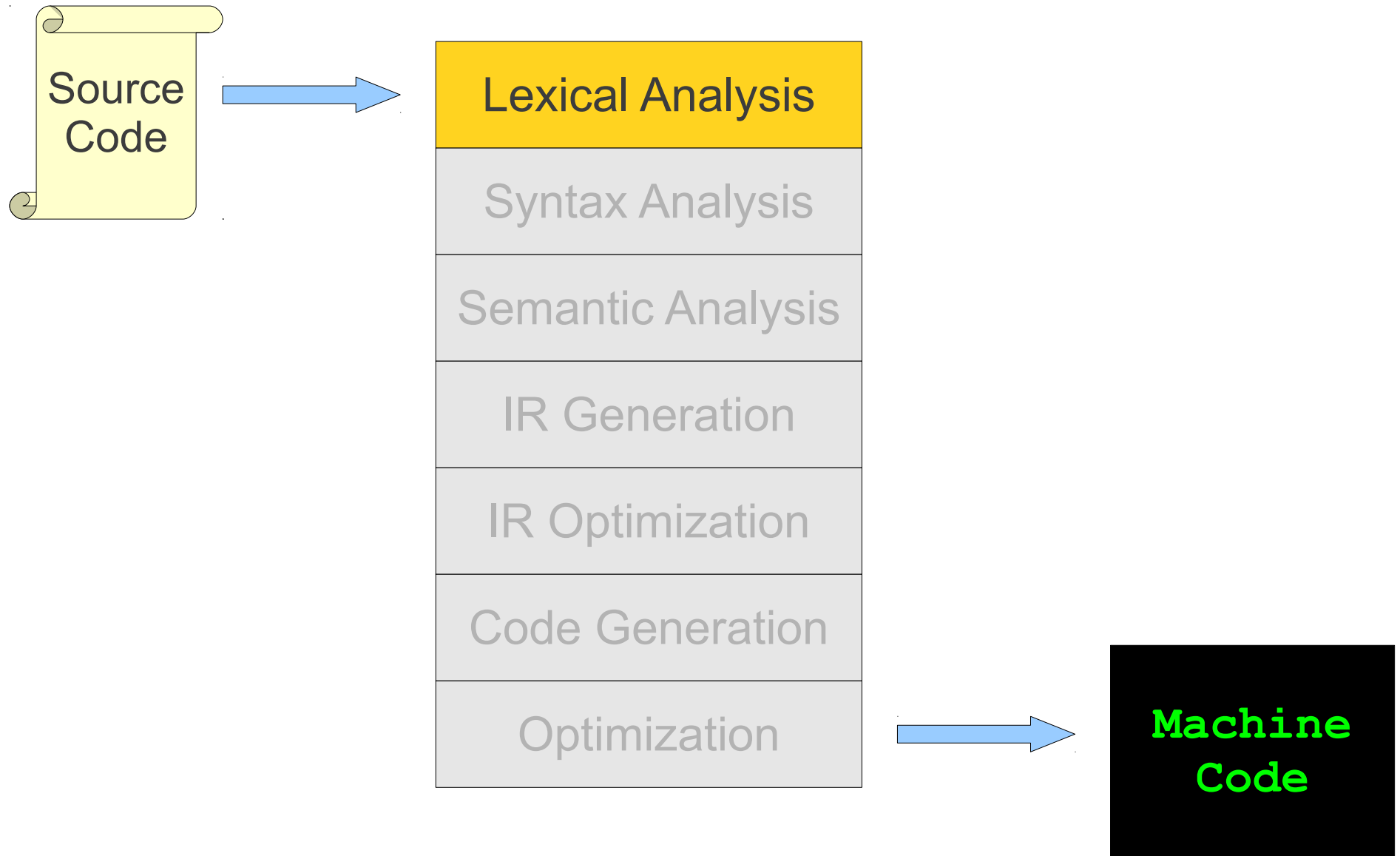
IR Optimization

Code Generation

Optimization

# Lexical Analysis

# Where We Are



# 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

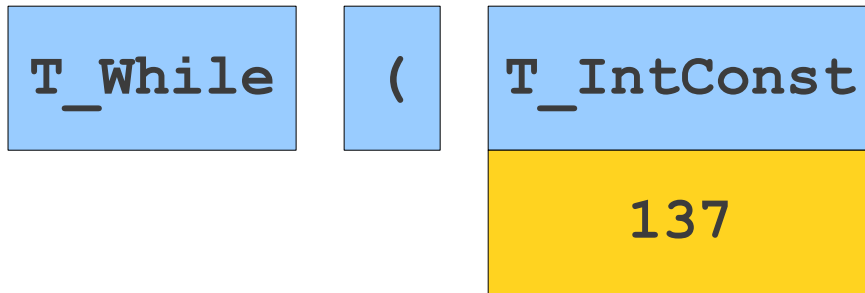
w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;
---	---	---	---	---	--	---	---	---	---	--	---	--	---	---	----	----	---	---	---	---

**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

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



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 **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)

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

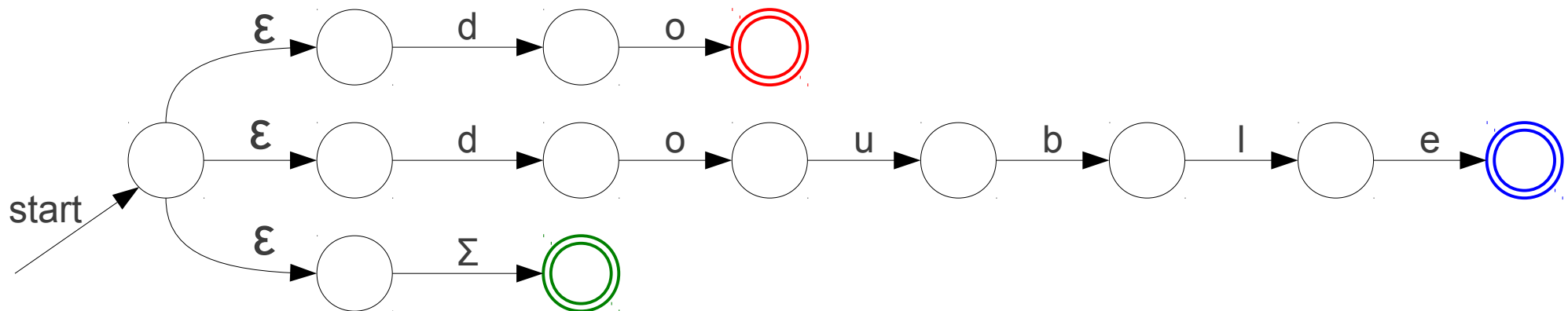
- **Regular expressions** are a family of descriptions that can be used to capture certain languages (the *regular languages*).
- Often provide a compact and human-readable description of the language.
- Used as the basis for numerous software systems.

# Matching Regular Expressions

# Implementing Regular Expressions

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

# A Minor Simplification



Annotate each accepting state with which automaton it came from.

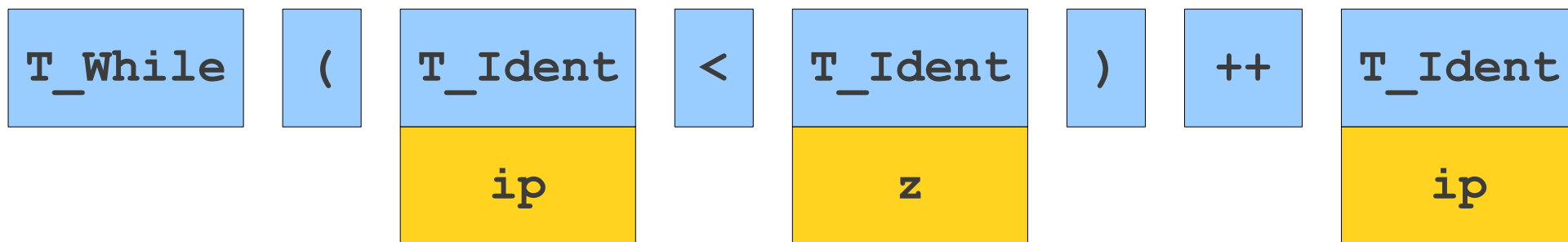
# Syntax Analysis





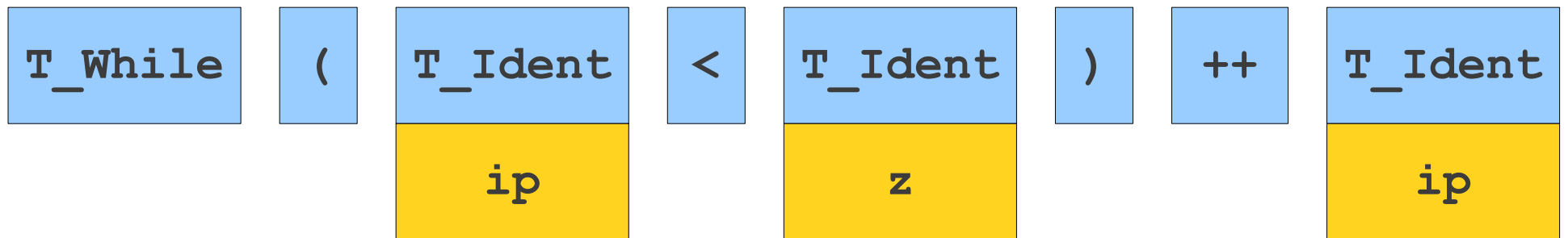
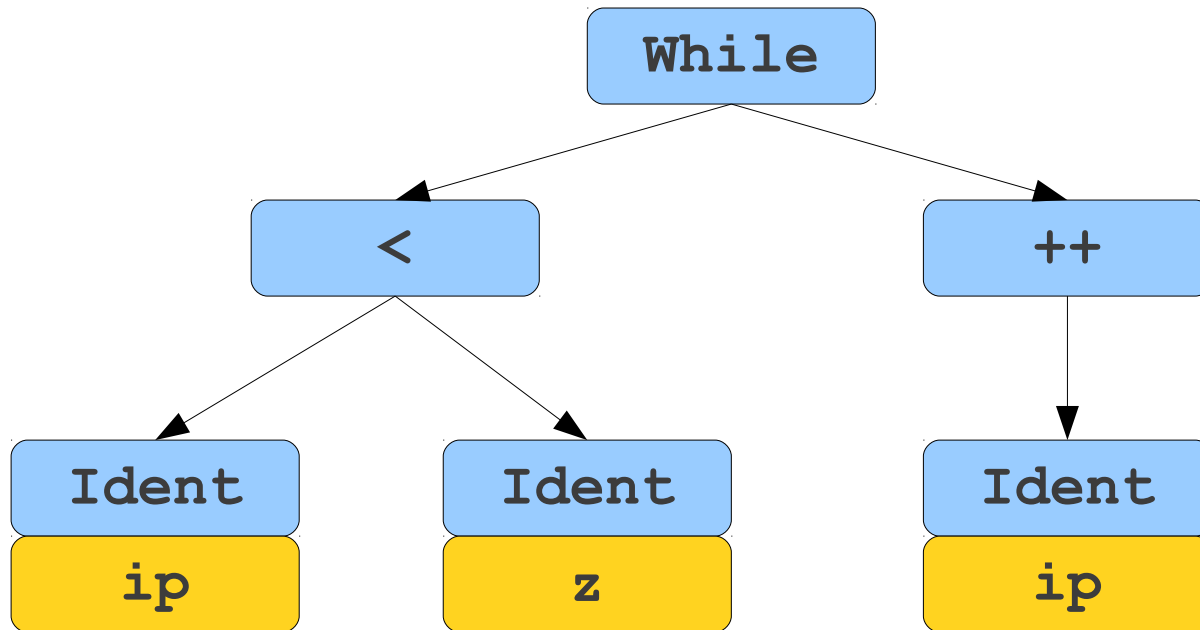
w	h	i	l	e		(	i	p		<		z	)	\n	\t	+	+	i	p	;
---	---	---	---	---	--	---	---	---	--	---	--	---	---	----	----	---	---	---	---	---

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



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

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

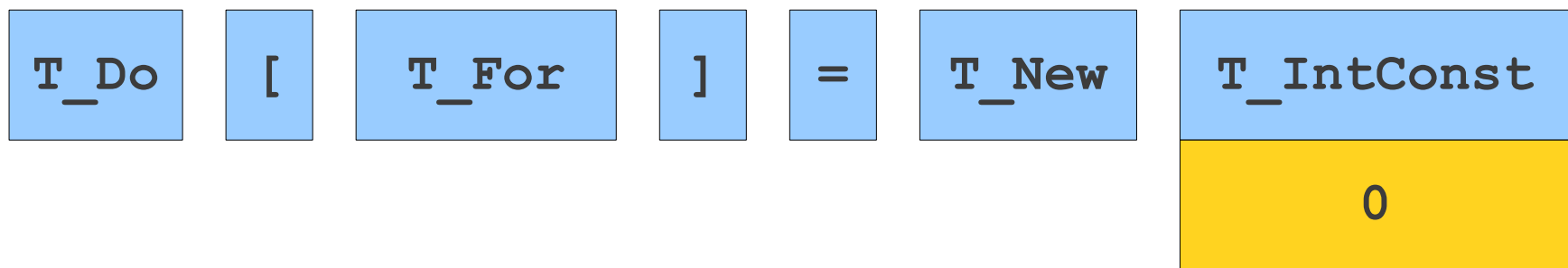


w	h	i	l	e		(	i	p		<		z	)	\n	\t	+	+	i	p	;
---	---	---	---	---	--	---	---	---	--	---	--	---	---	----	----	---	---	---	---	---

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

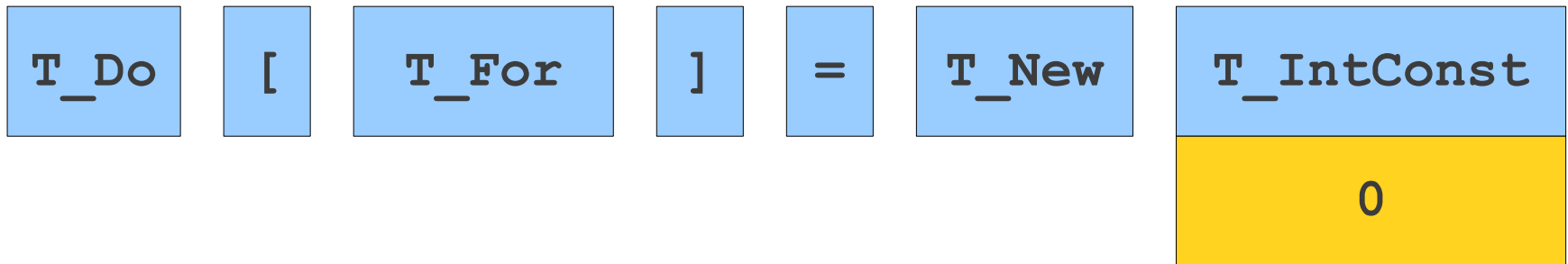
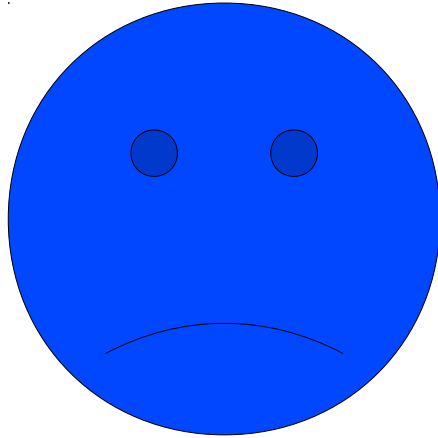
d	o	[	f	o	r	]		=		n	e	w		0	;
---	---	---	---	---	---	---	--	---	--	---	---	---	--	---	---

do[for] = new 0;



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

`do[for] = new 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.

# 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 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)**

# 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** → **/**

# Derivations

$E$   
 $\Rightarrow E \text{ Op } E$   
 $\Rightarrow E \text{ Op } (E)$   
 $\Rightarrow E \text{ Op } (E \text{ Op } E)$   
 $\Rightarrow E * (E \text{ Op } E)$   
 $\Rightarrow \text{int} * (E \text{ Op } E)$   
 $\Rightarrow \text{int} * (\text{int} \text{ Op } E)$   
 $\Rightarrow \text{int} * (\text{int} \text{ Op } \text{int})$   
 $\Rightarrow \text{int} * (\text{int} + \text{int})$

- This sequence of steps is called a **derivation**.
- A string  $\alpha A \omega$  **yields** string  $\alpha \gamma \omega$  iff  $A \rightarrow \gamma$  is a production.
- If  $\alpha$  yields  $\beta$ , we write  $\alpha \Rightarrow \beta$ .
- We say that  $\alpha$  **derives**  $\beta$  iff there is a sequence of strings where
$$\alpha \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \beta$$
- If  $\alpha$  derives  $\beta$ , we write  $\alpha \Rightarrow^* \beta$ .

# 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.

# 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.

# Top-Down Parsing

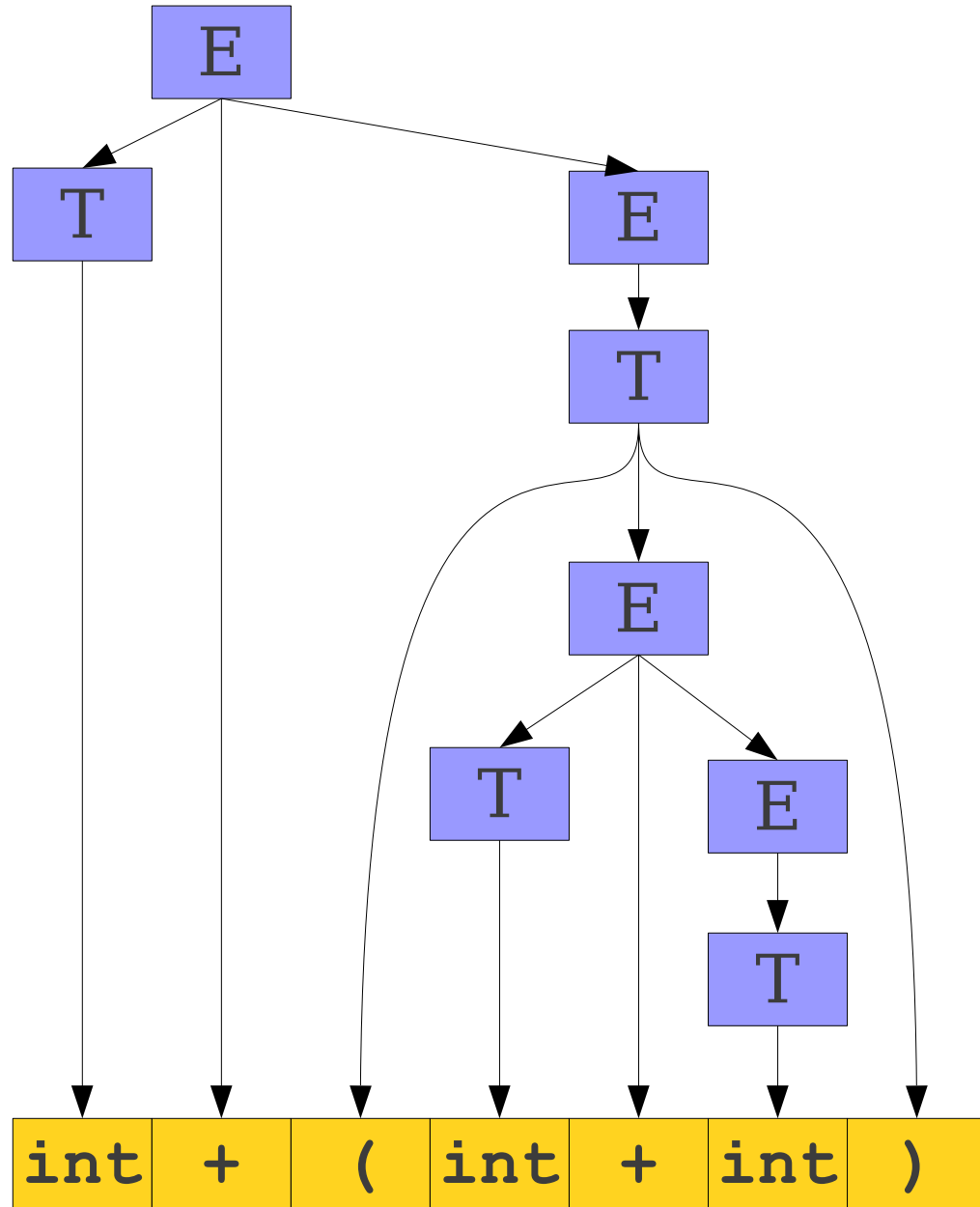


# Different Types of Parsing

- **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

$E \rightarrow T$   
 $E \rightarrow T + E$   
 $T \rightarrow \text{int}$   
 $T \rightarrow (E)$



# Predictive Parsing

# 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.

# 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

**E**  $\rightarrow$  **int**

**E**  $\rightarrow$  **(E Op E)**

**Op**  $\rightarrow$  **+**

**Op**  $\rightarrow$  **\***

# LL(1) Parse Tables

**E**  $\rightarrow$  **int**

**E**  $\rightarrow$  **(E Op E)**

**Op**  $\rightarrow$  **+**

**Op**  $\rightarrow$  **\***

	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

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**  $\rightarrow$  int

(2) **E**  $\rightarrow$  (**E Op E**)

(3) **Op**  $\rightarrow$  +

(4) **Op**  $\rightarrow$  \*

	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

(1) **E** → **int**

(2) **E** → (**E Op E**)

(3) **Op** → **+**

(4) **Op** → **\***

E\$	(int + (int * int))\$
(E Op E)\$	(int + (int * int))\$

	int	(	)	+	*
E	1	2			
Op				3	4

# LL(1) Parsing

- (1) **E**  $\rightarrow$  **int**
- (2) **E**  $\rightarrow$  (**E Op E**)
- (3) **Op**  $\rightarrow$  **+**
- (4) **Op**  $\rightarrow$  **\***

E\$	(int + (int * int))\$
(E Op E)\$	(int + (int * int))\$

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.

	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	(	)	+	*
E	1	2			
Op				3	4

# LL(1) Parsing

(1) **E**  $\rightarrow$  **int**

(2) **E**  $\rightarrow$  (**E Op E**)

(3) **Op**  $\rightarrow$  **+**

(4) **Op**  $\rightarrow$  **\***

E\$	(int + (int * int))\$
(E Op E)\$	(int + (int * int))\$
E Op E)\$	int + (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))\$

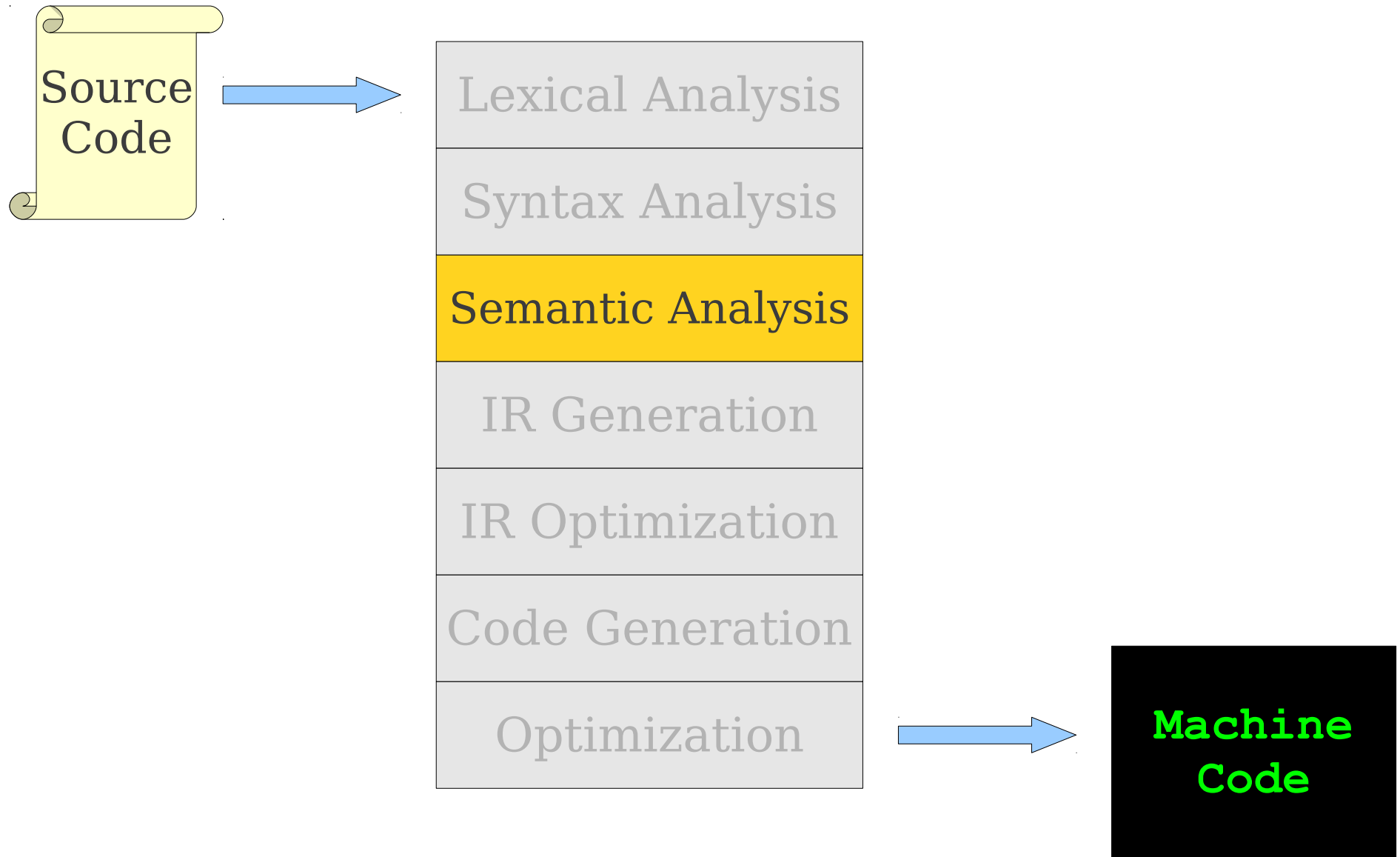
	int	(	)	+	*
E	1	2			
Op				3	4

# 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.

# Semantic Analysis

# Where We Are



# Where We Are

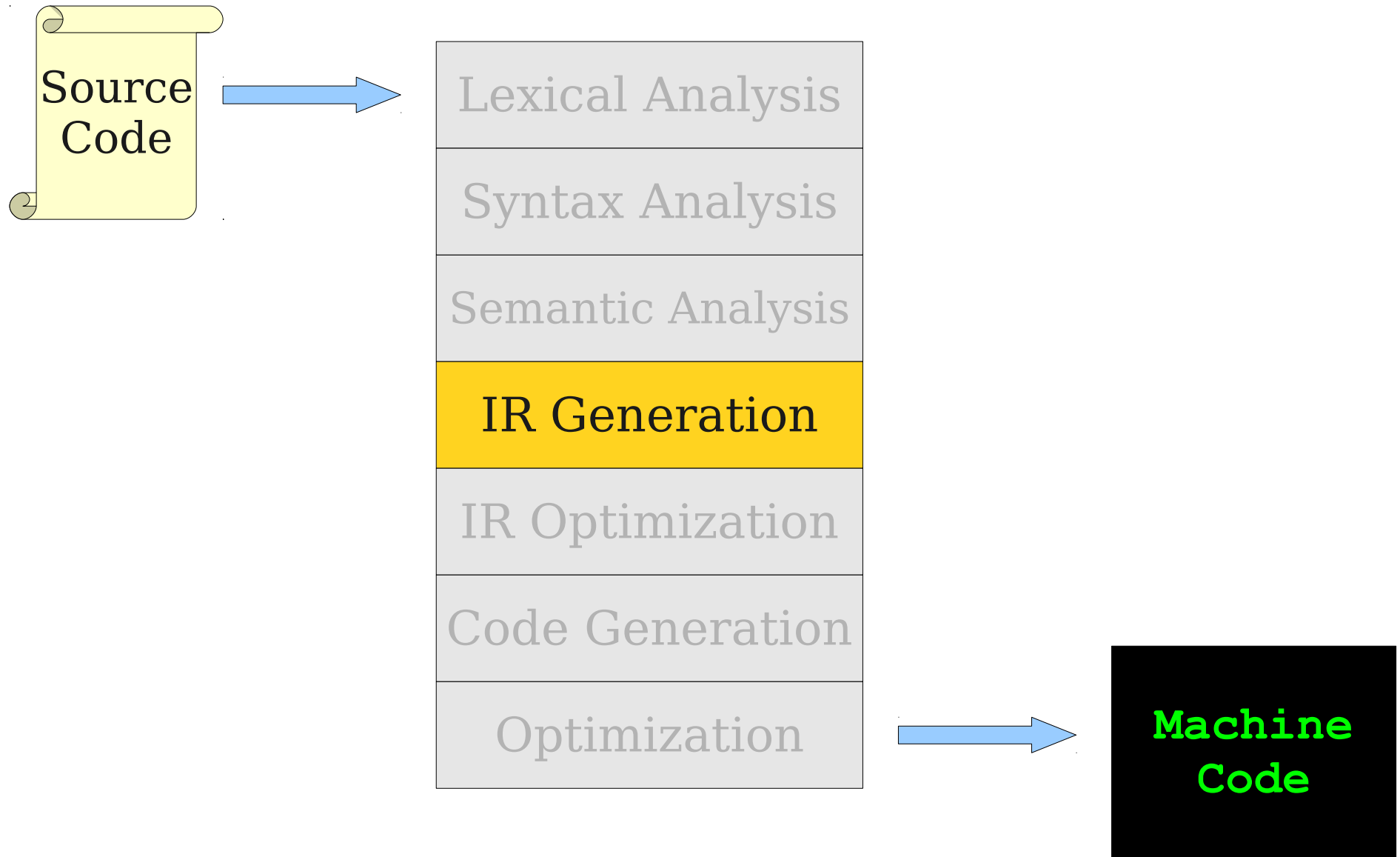
- Program is *lexically* well-formed:
  - Identifiers have valid names.
  - Strings are properly terminated.
  - No stray characters.
- Program is *syntactically* well-formed:
  - Class declarations have the correct structure.
  - Expressions are syntactically valid.
- Does this mean that the program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

# Three-Address Code IR

# Where We Are





# 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;
```

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

# Simple TAC Instructions

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

# 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;
```

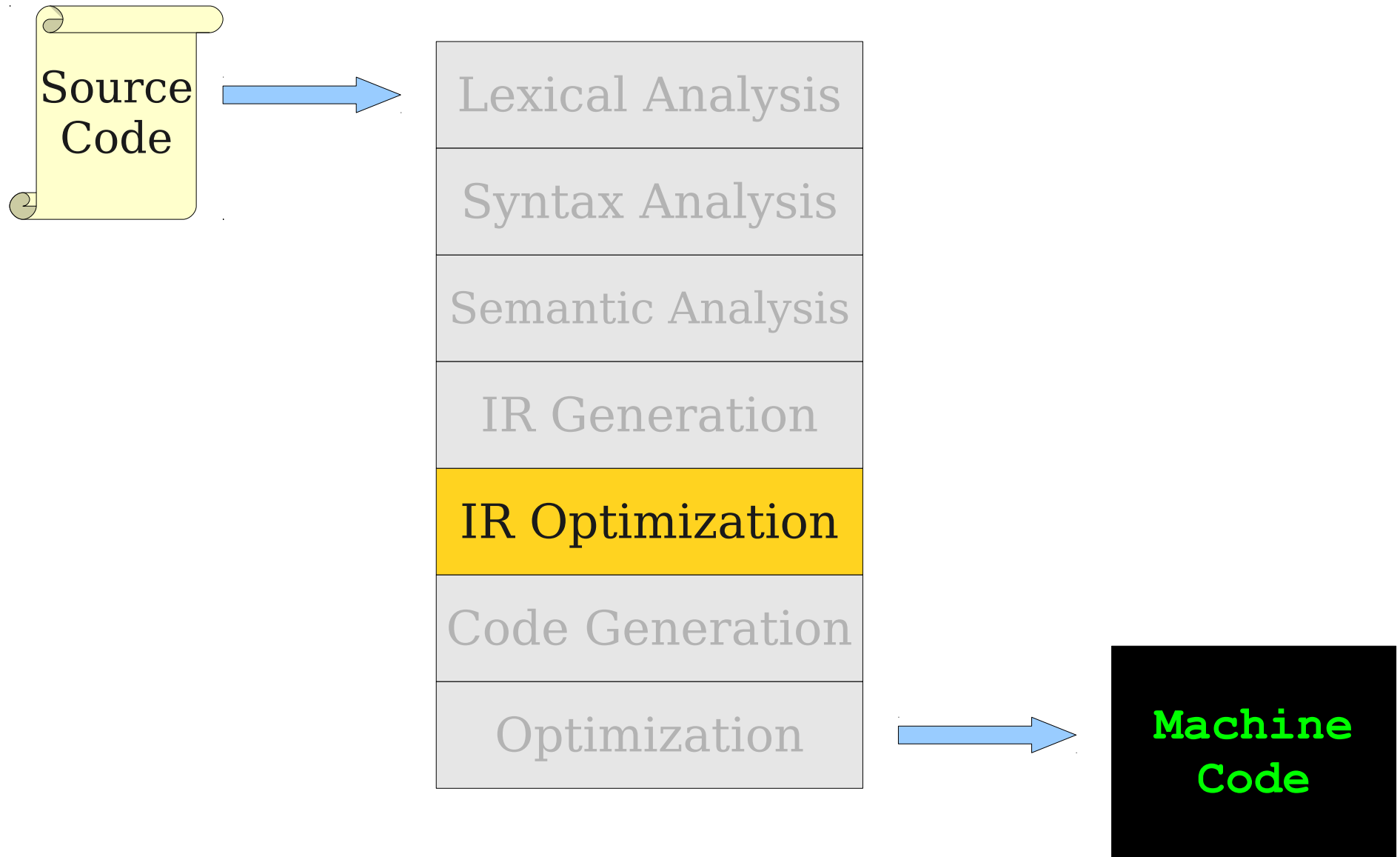
```
    Goto _L0;
```

```
_L1:
```

```
    y = x;
```

# IR Optimization

# Where We Are



# 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.

# 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.

# 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.

# 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;

```
_tmp0 = LCall _ReadInteger;  
a = _tmp0 ;  
_tmp1 = LCall _ReadInteger;  
b = _tmp1 ;
```

```
_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 ;
```

```
PushParam a ;  
LCall _PrintInt ;  
PopParams 4 ;
```



# 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;**

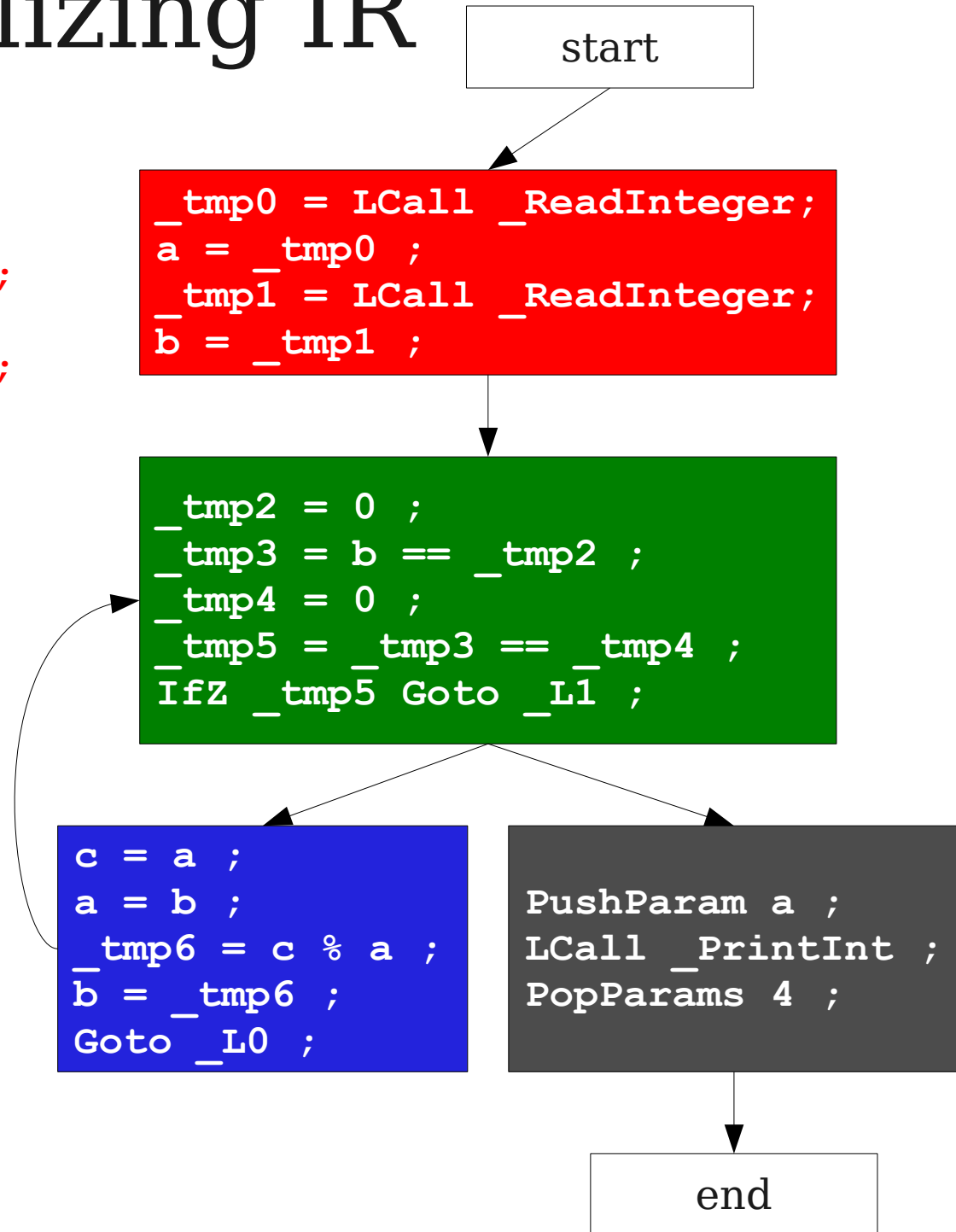
\_L1:

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

# 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

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

...

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

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

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

...

$\mathbf{v_2 = v_1}$

- 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  $v_1$  and  $v_2$  are not reassigned,  
we can rewrite expressions of the form

$$a = \dots v_1 \dots$$

as

$$a = \dots v_2 \dots$$

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;`

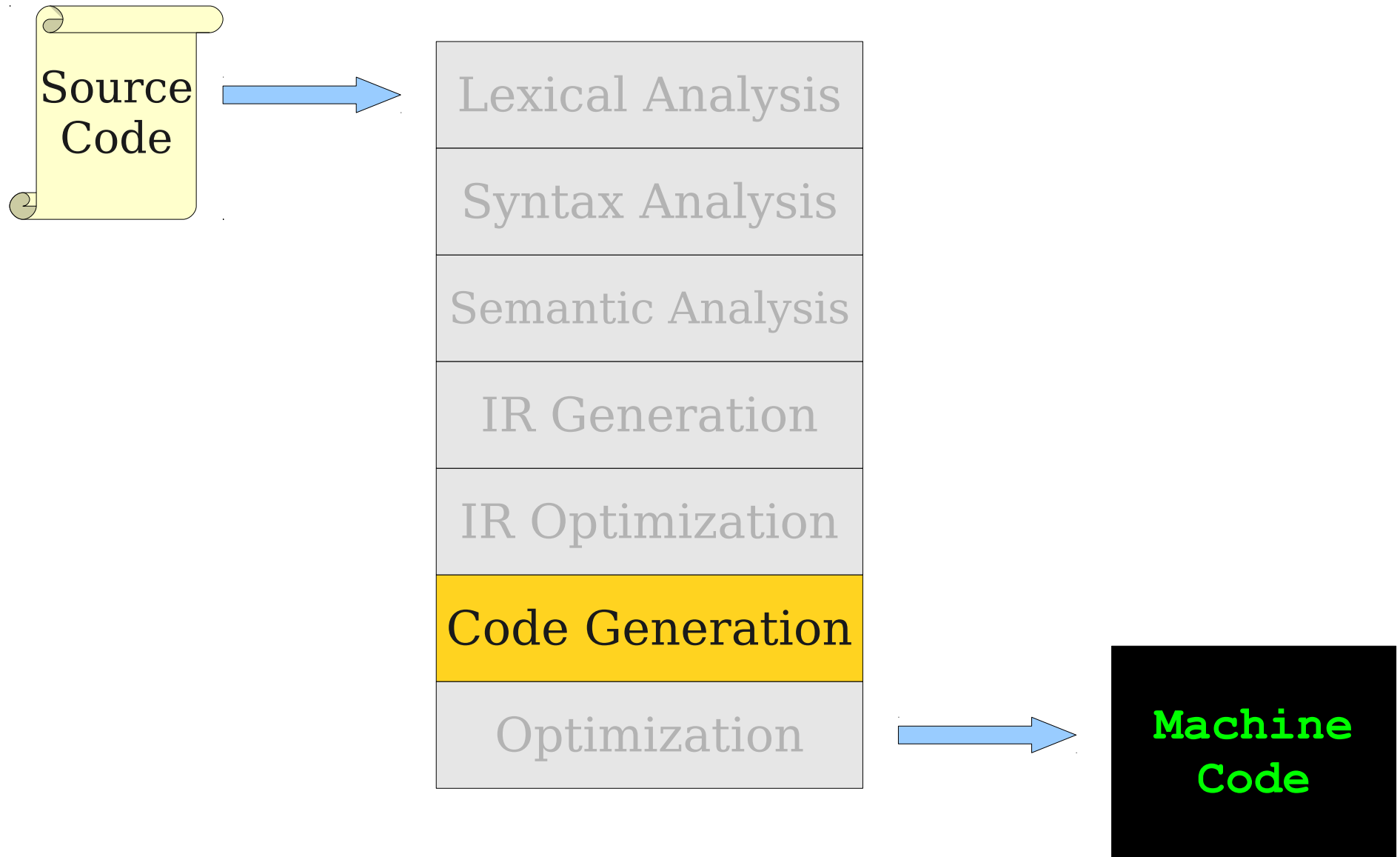
- **Constant Folding**

- Evaluate expressions at compile-time if they have a constant value.
- e.g. rewrite `x = 4 * 5;` as `x = 20;`.

# Register Allocation



# Where We Are



# 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.

# 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.

# 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
```

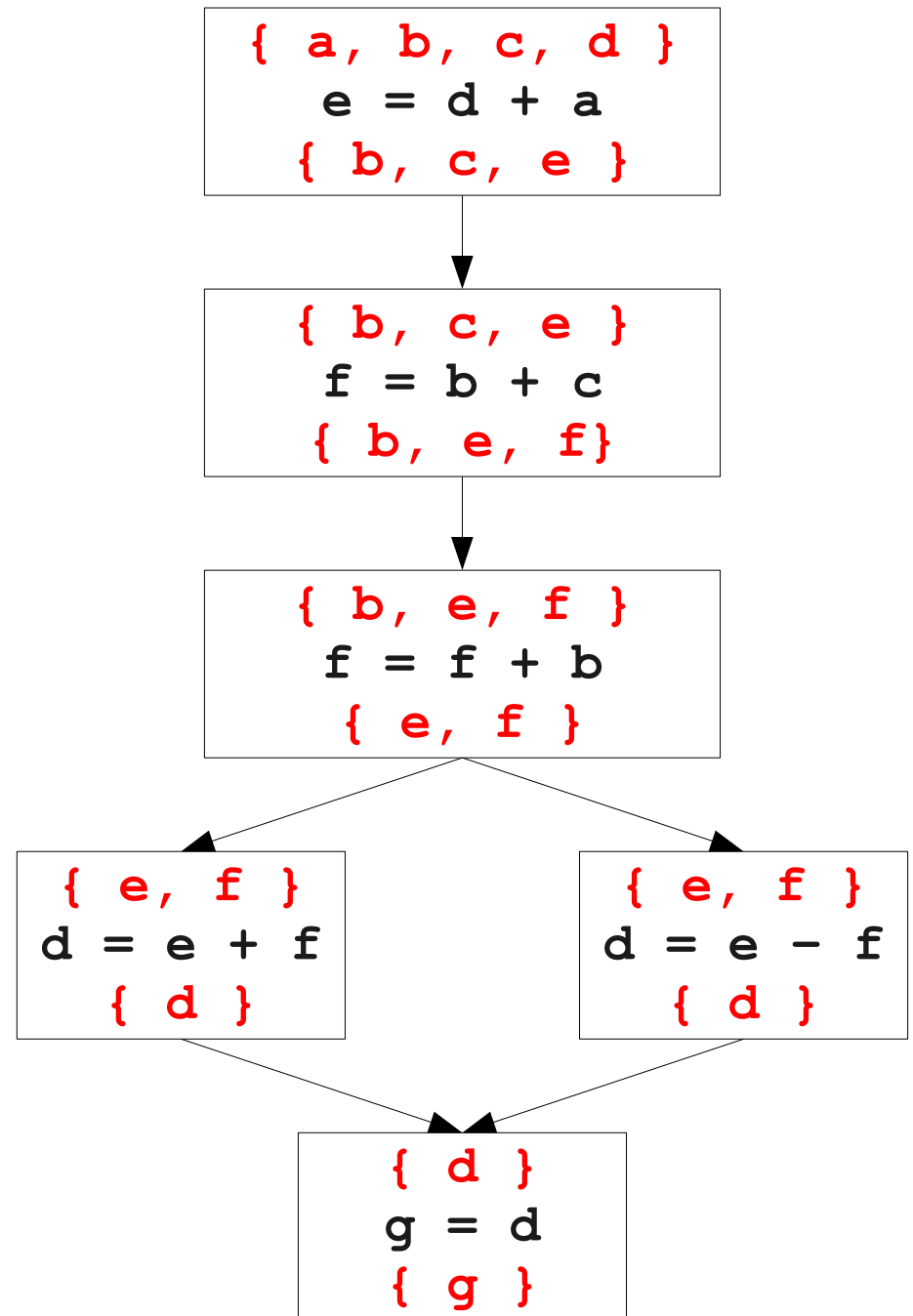
```
_L1:
```

```
g = d
```



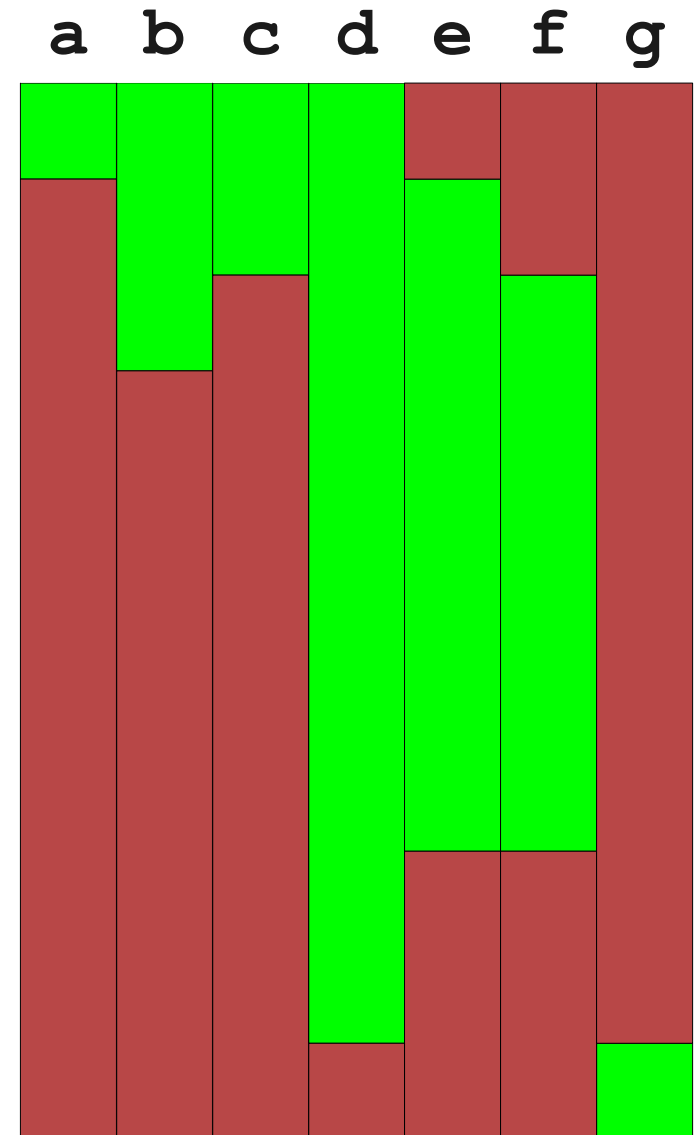
# Live Ranges and Live Intervals

	a	b	c	d	e	f	g
<code>e = d + a</code>							
<code>f = b + c</code>							
<code>f = f + b</code>							
<code>IfZ e Goto _L0</code>							
<code>d = e + f</code>							
<code>Goto _L1;</code>							
<code>_L0:</code>							
<code>d = e - f</code>							
<code>_L1:</code>							
<code>g = d</code>							

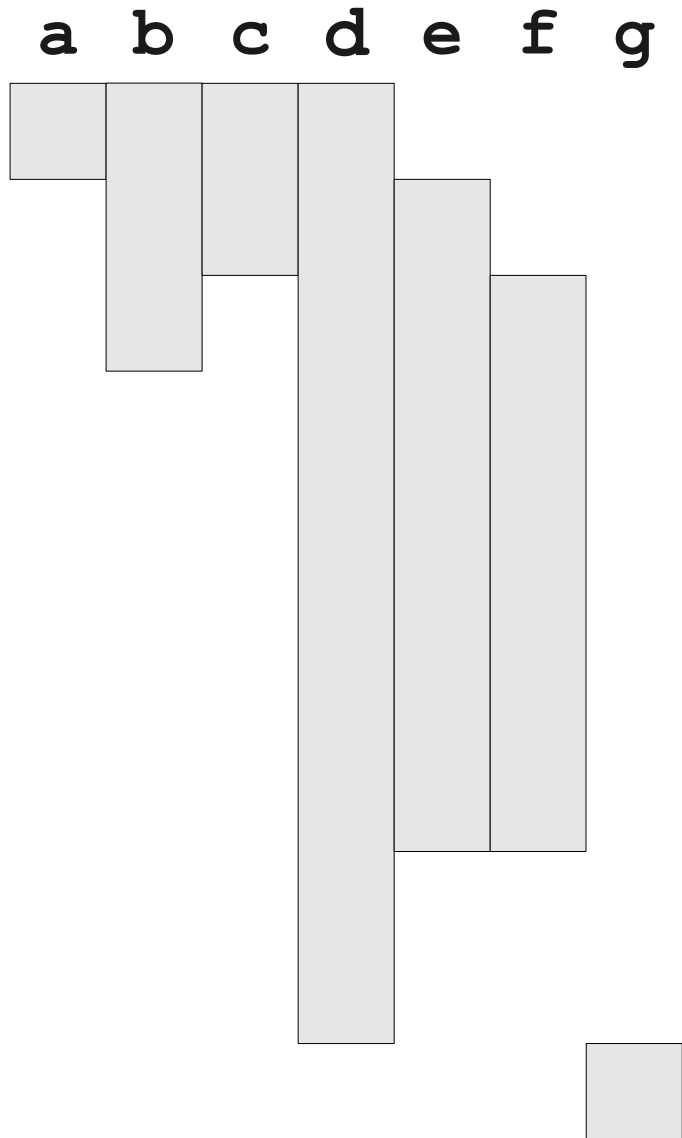


# Register Allocation with Live Intervals

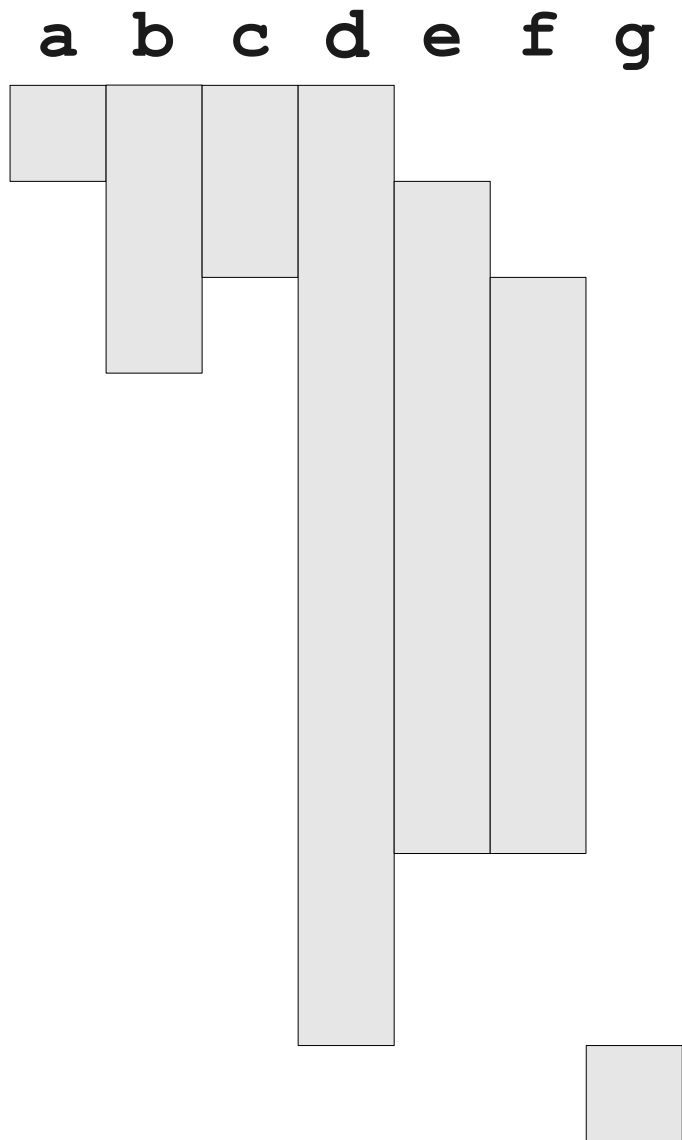
- Given the live intervals for all the variables in the program, we can allocate registers using a simple greedy algorithm.
- Idea: Track which registers are free at each point.
- When a live interval begins, give that variable a free register.
- When a live interval ends, the register is once again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



# Register Allocation with Live Intervals



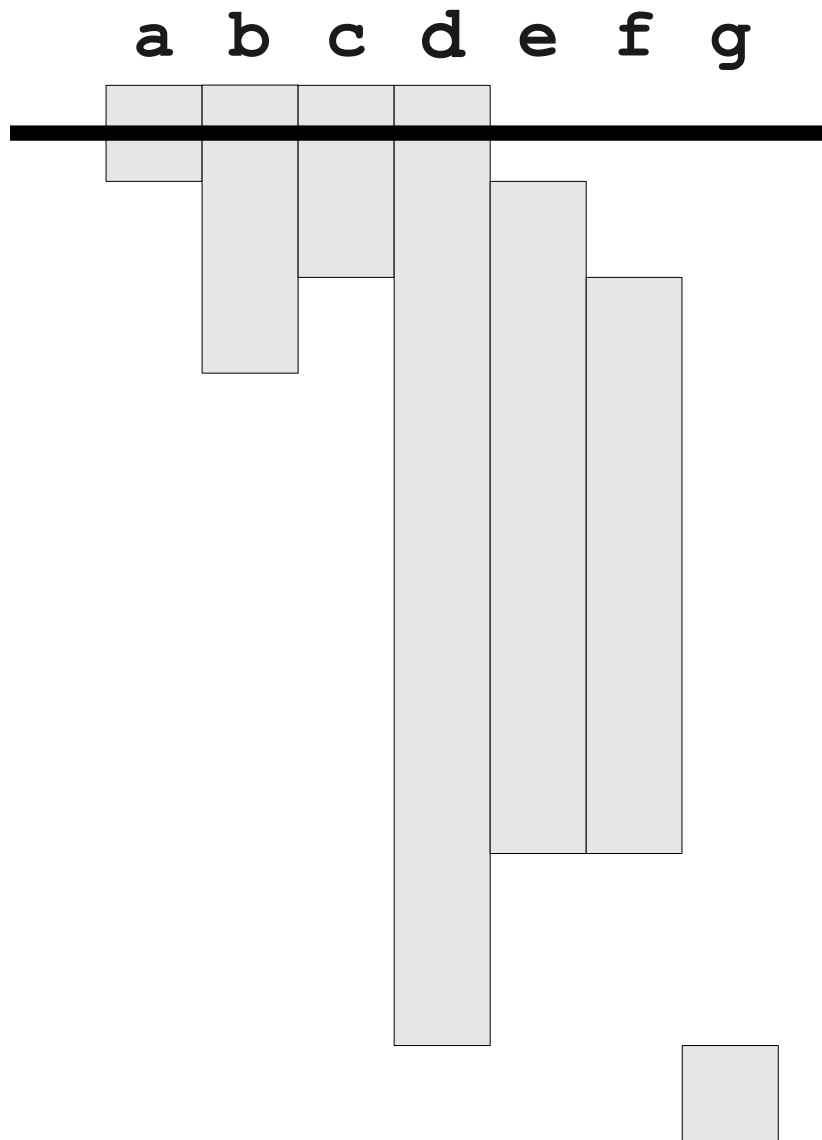
# Register Allocation with Live Intervals



**Free Registers**



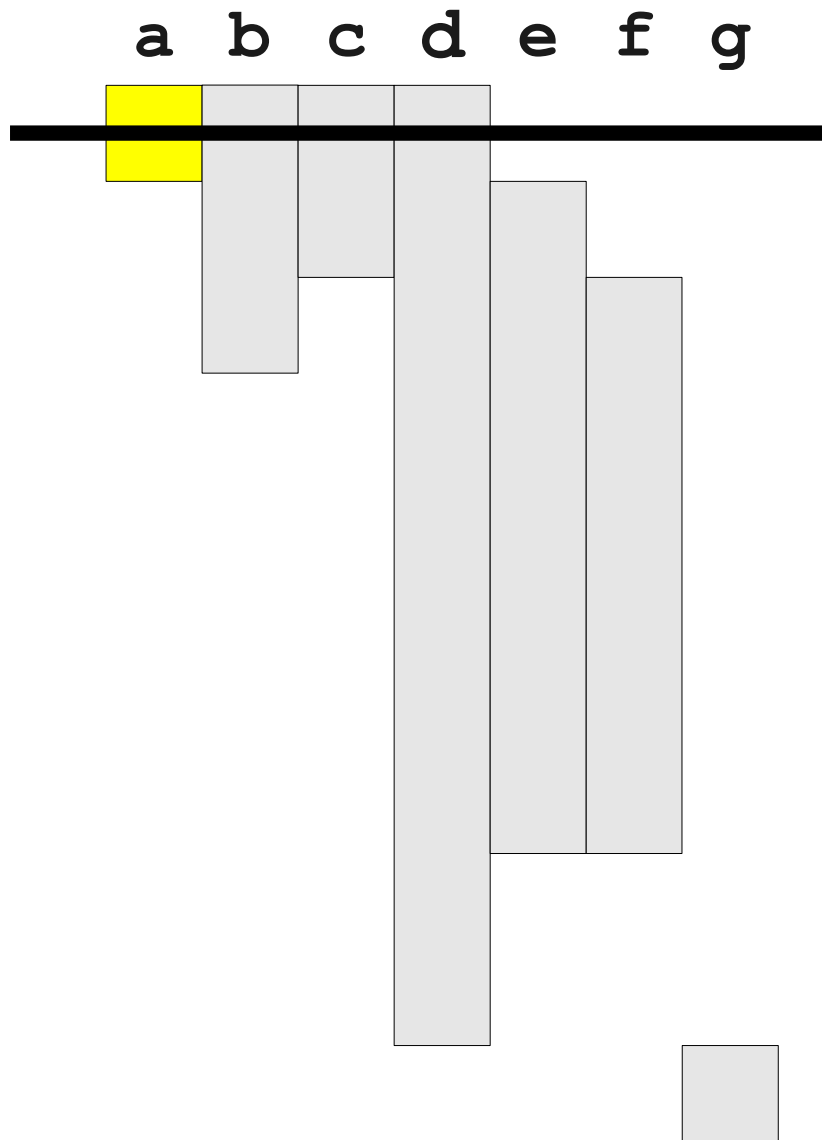
# Register Allocation with Live Intervals



**Free Registers**



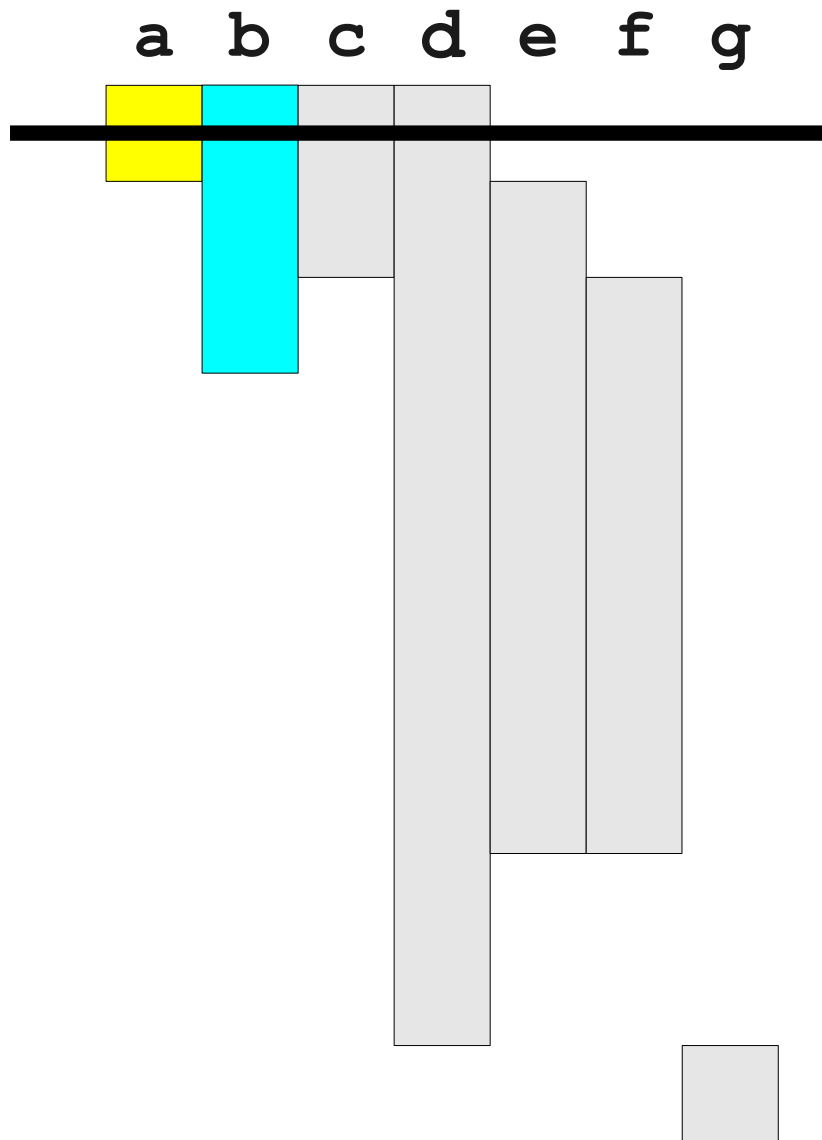
# Register Allocation with Live Intervals



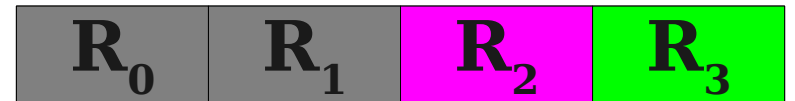
## Free Registers



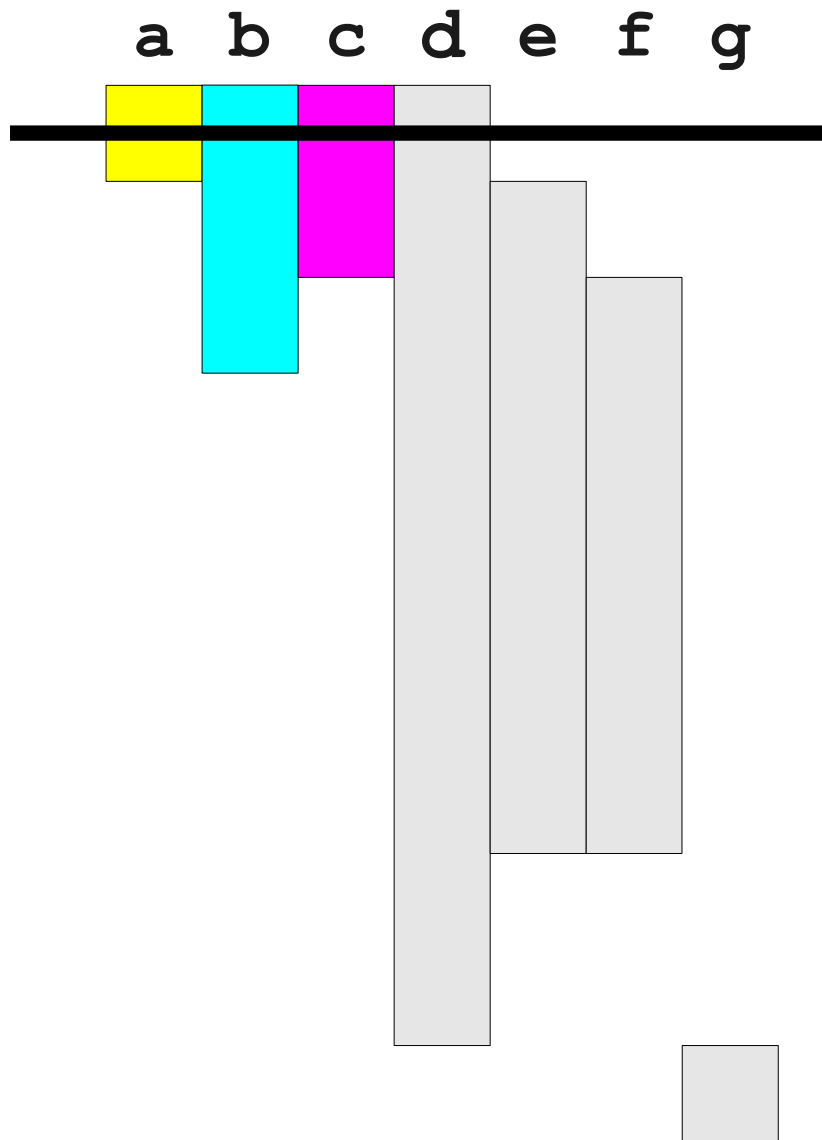
# Register Allocation with Live Intervals



## Free Registers



# Register Allocation with Live Intervals

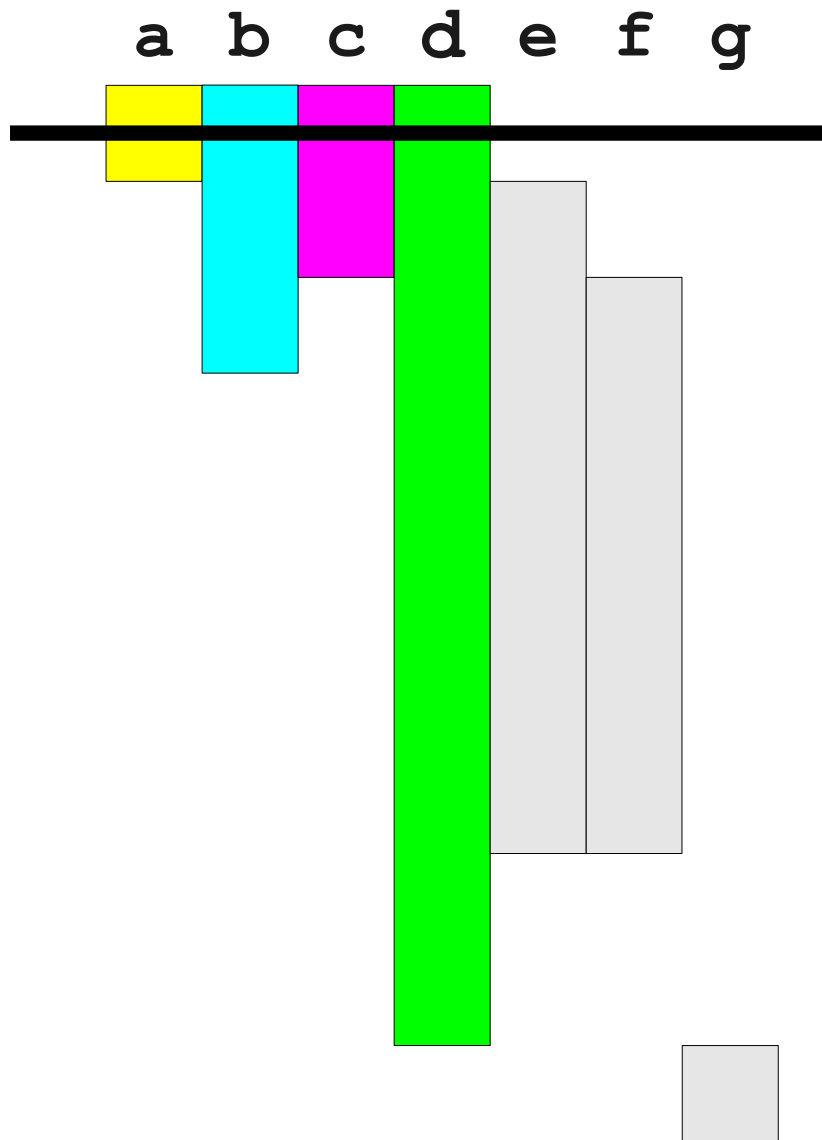


## Free Registers





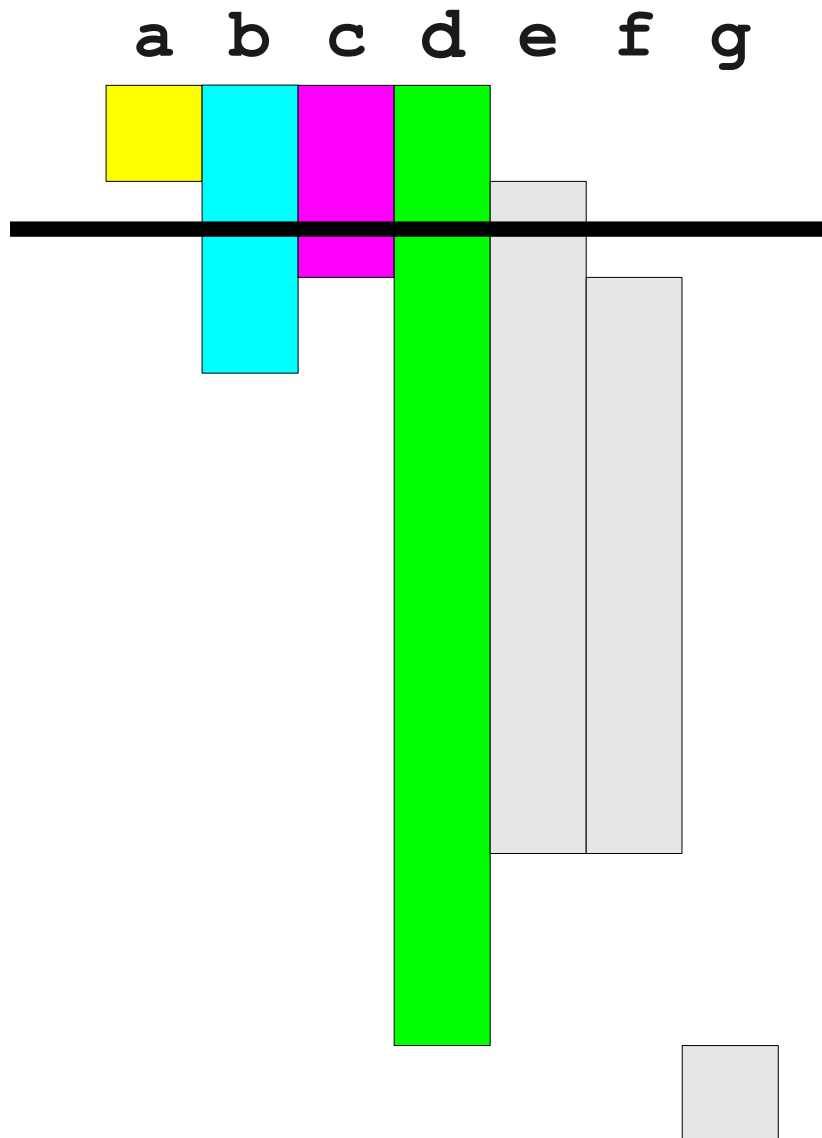
# Register Allocation with Live Intervals



## Free Registers

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

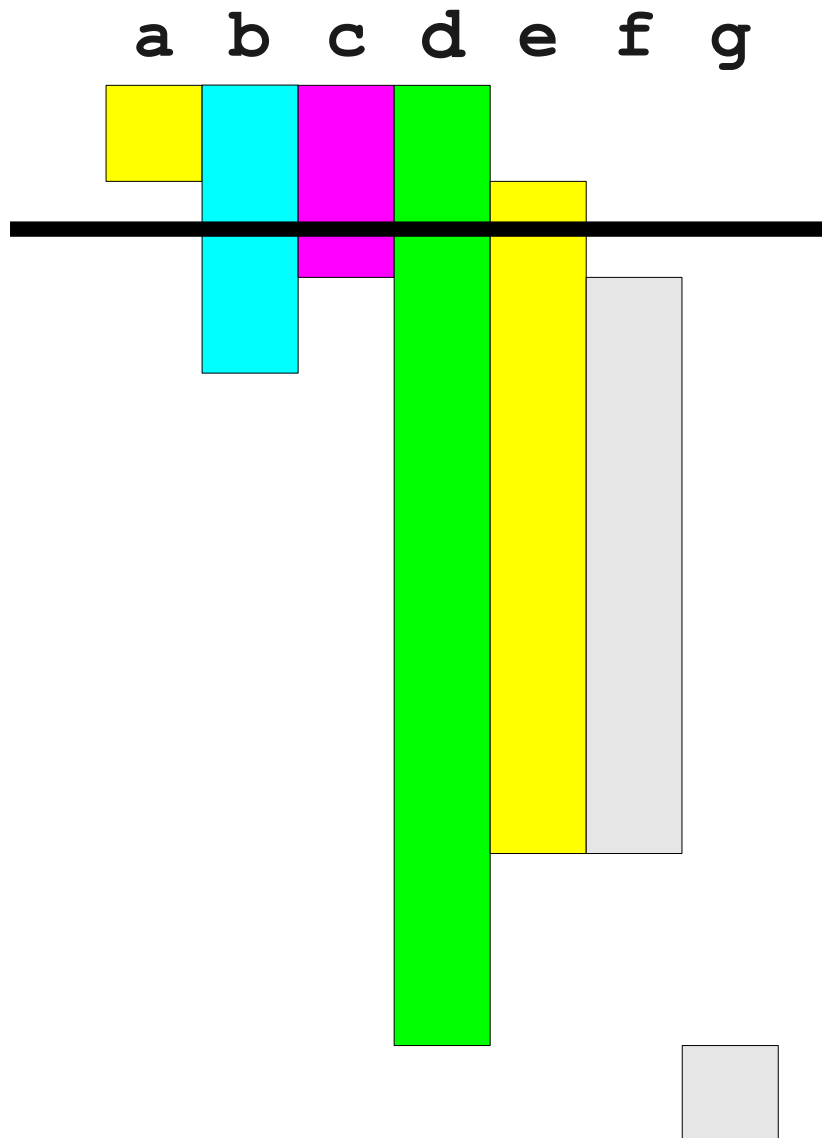
# Register Allocation with Live Intervals



## Free Registers



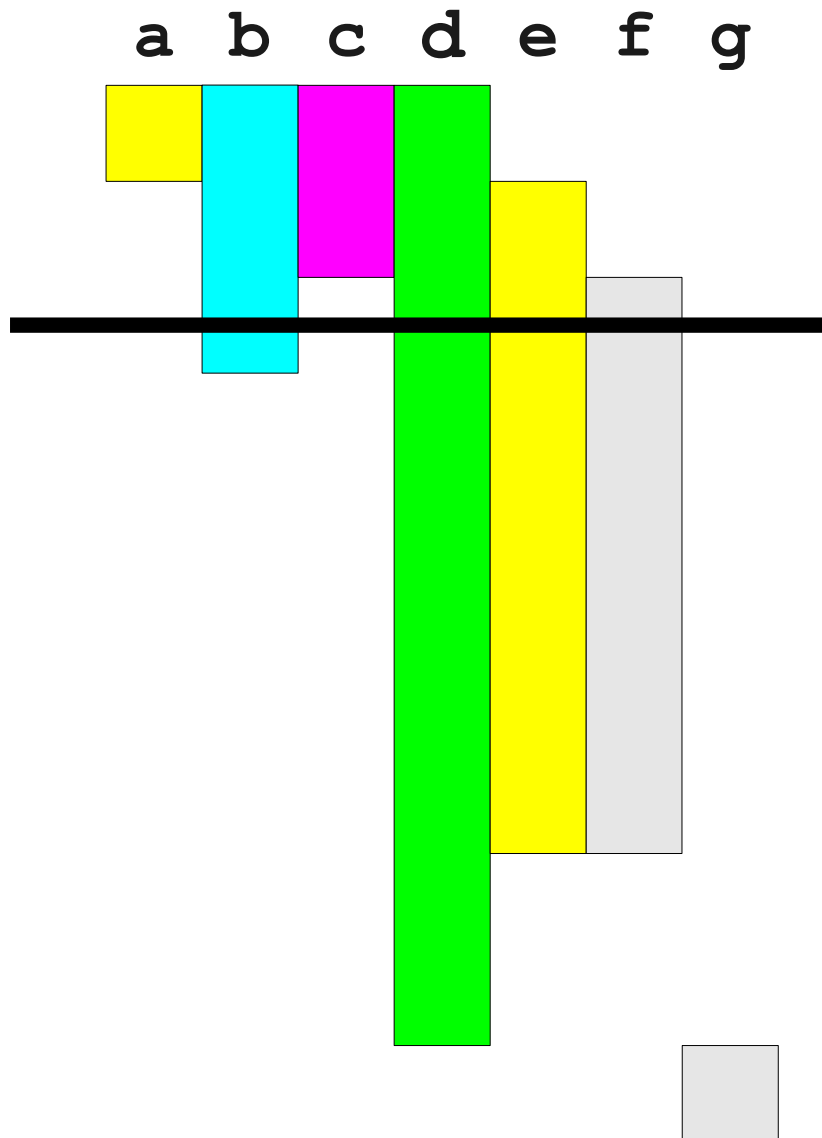
# Register Allocation with Live Intervals



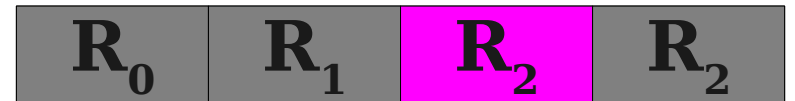
## Free Registers

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

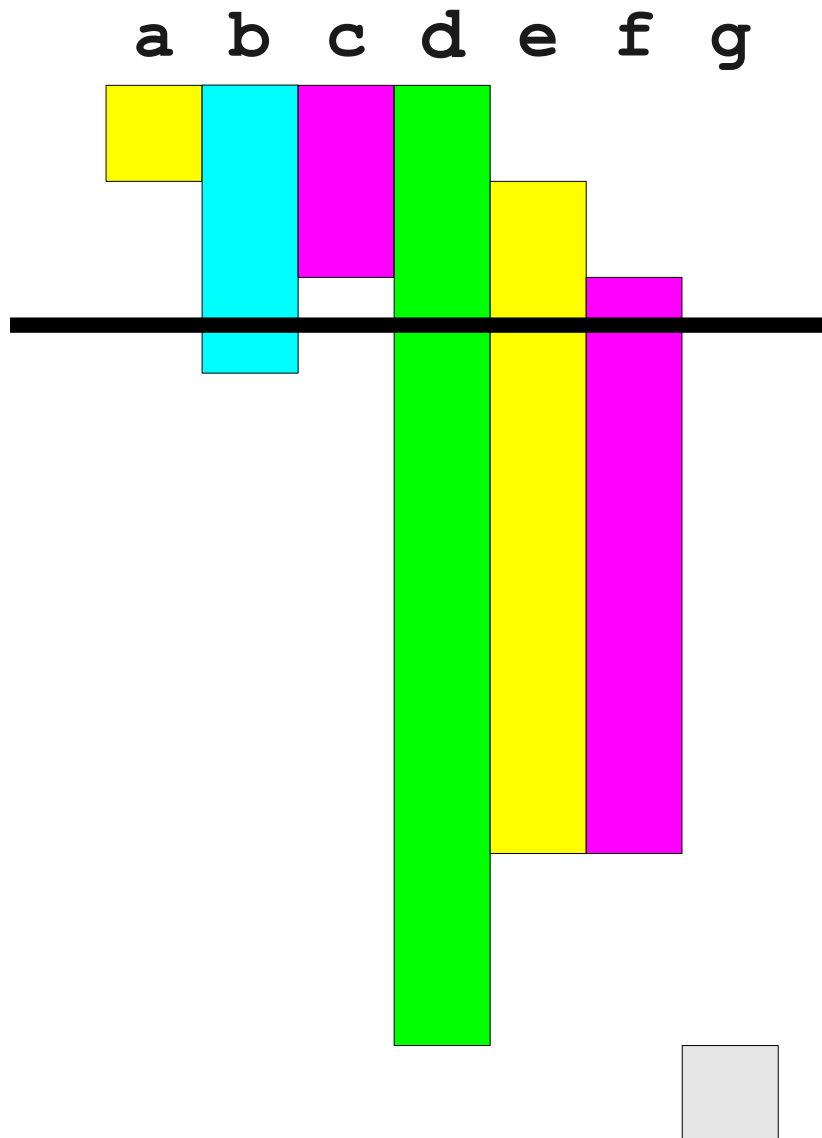
# Register Allocation with Live Intervals



## Free Registers



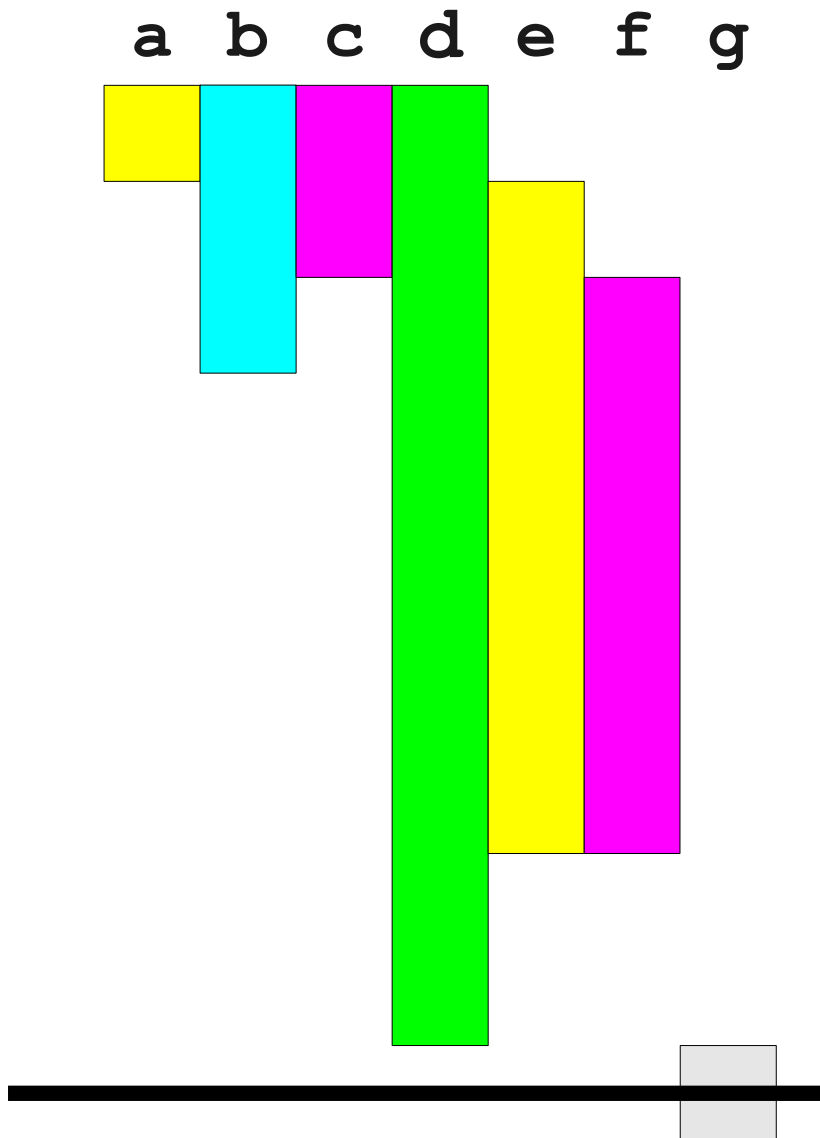
# Register Allocation with Live Intervals



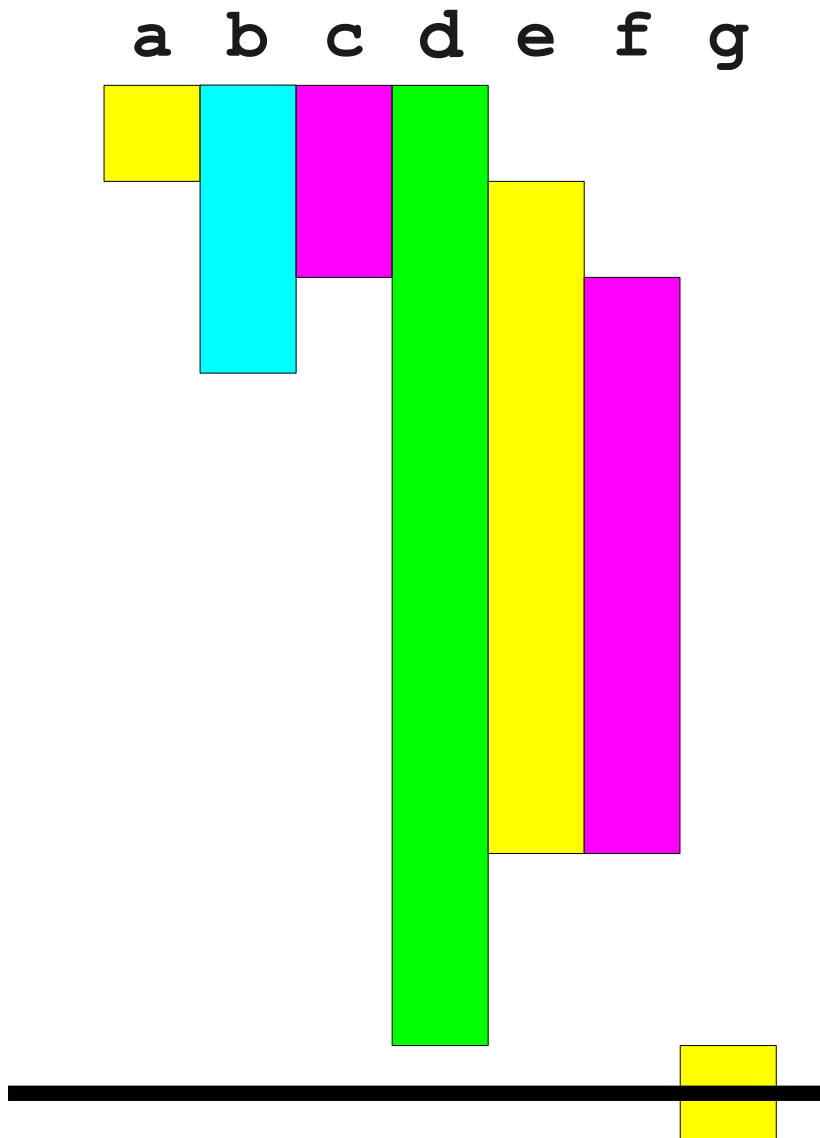
## Free Registers

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

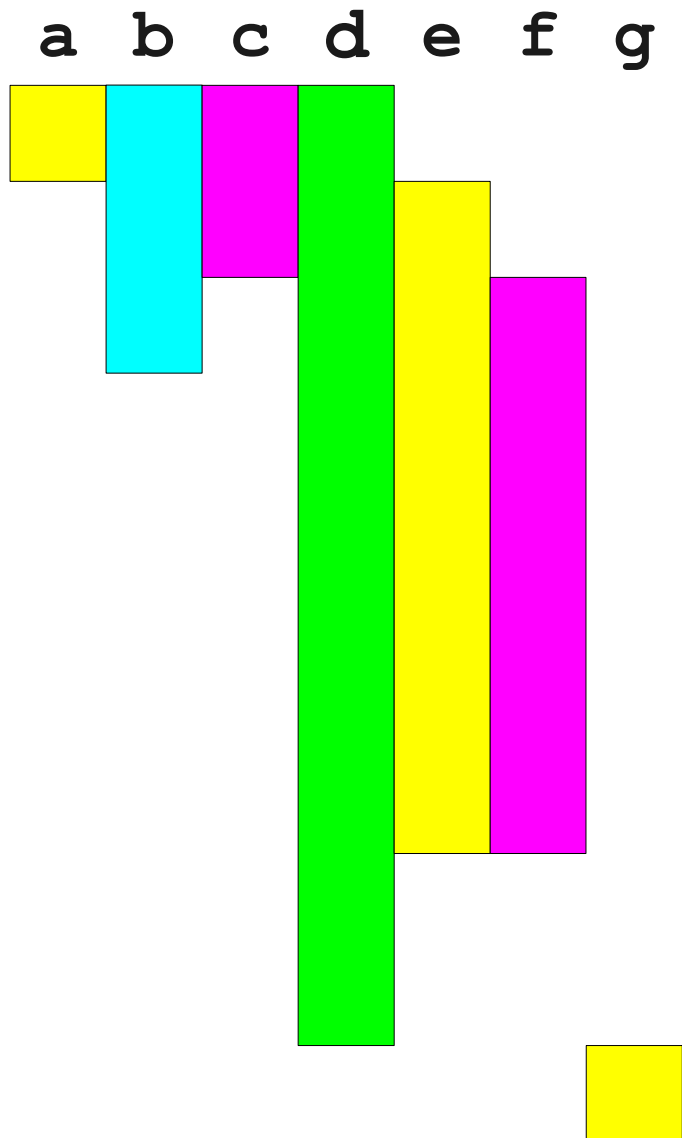
# Register Allocation with Live Intervals



# Register Allocation with Live Intervals



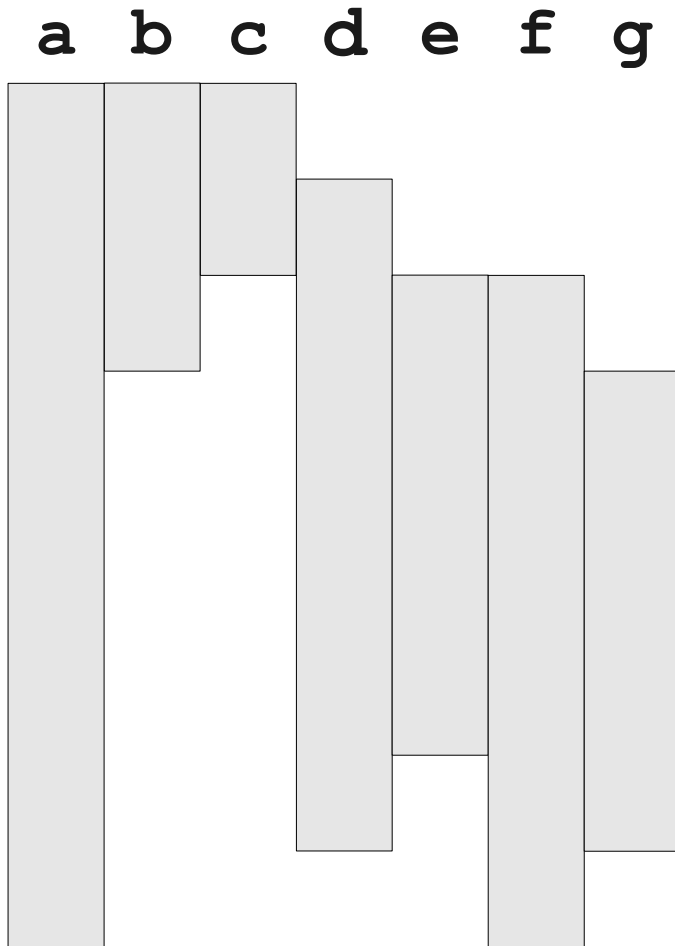
# Register Allocation with Live Intervals



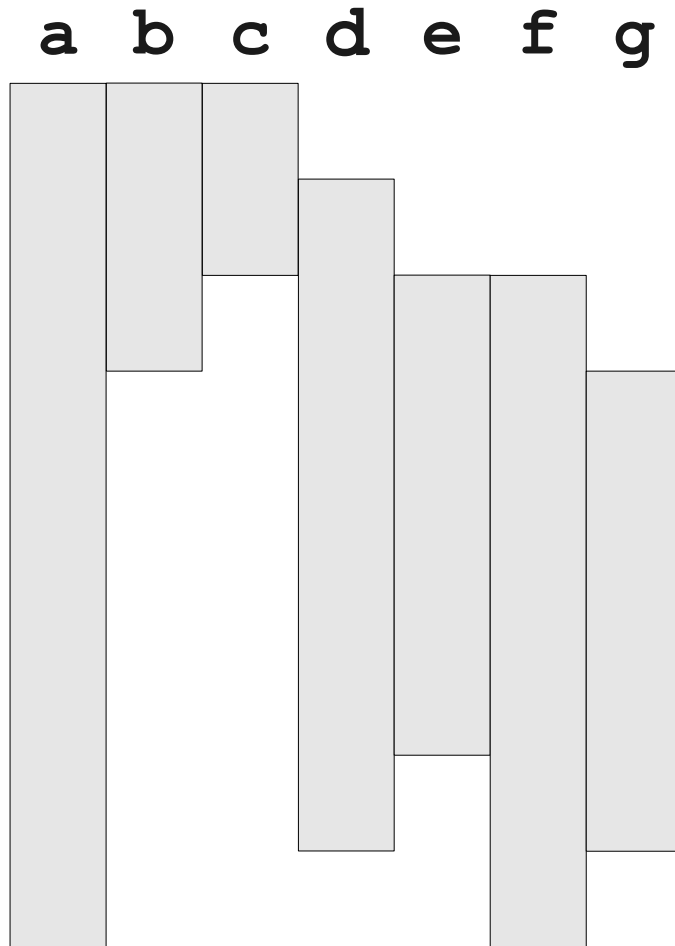


# Another Example

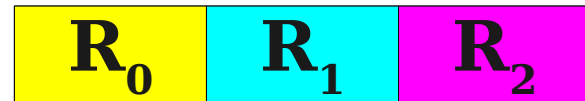
# Another Example



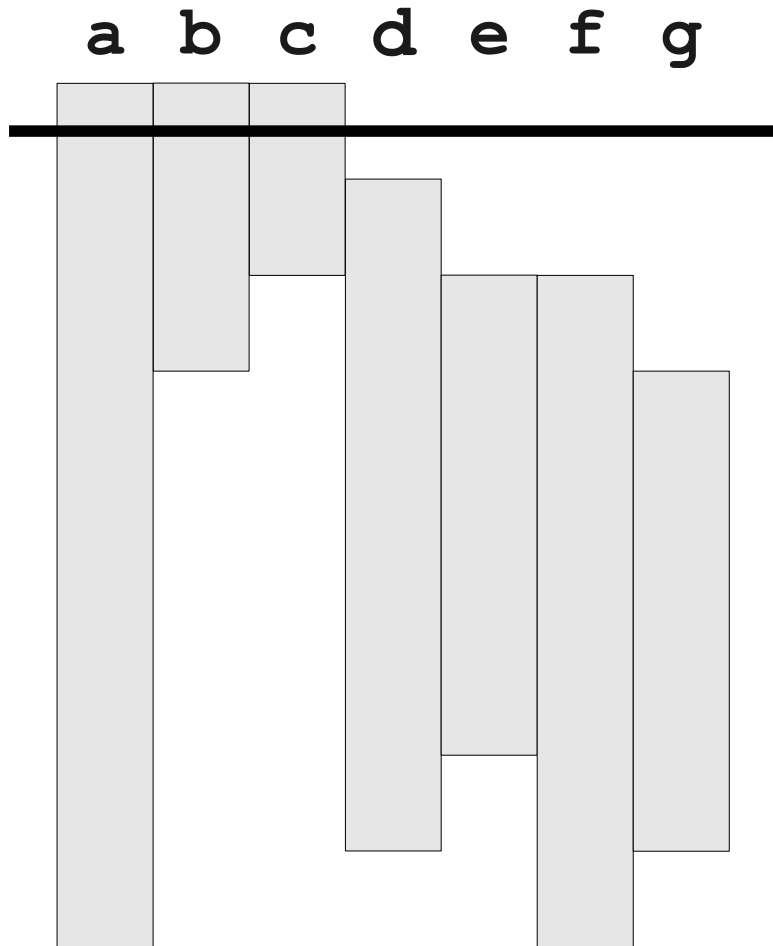
# Another Example



**Free Registers**



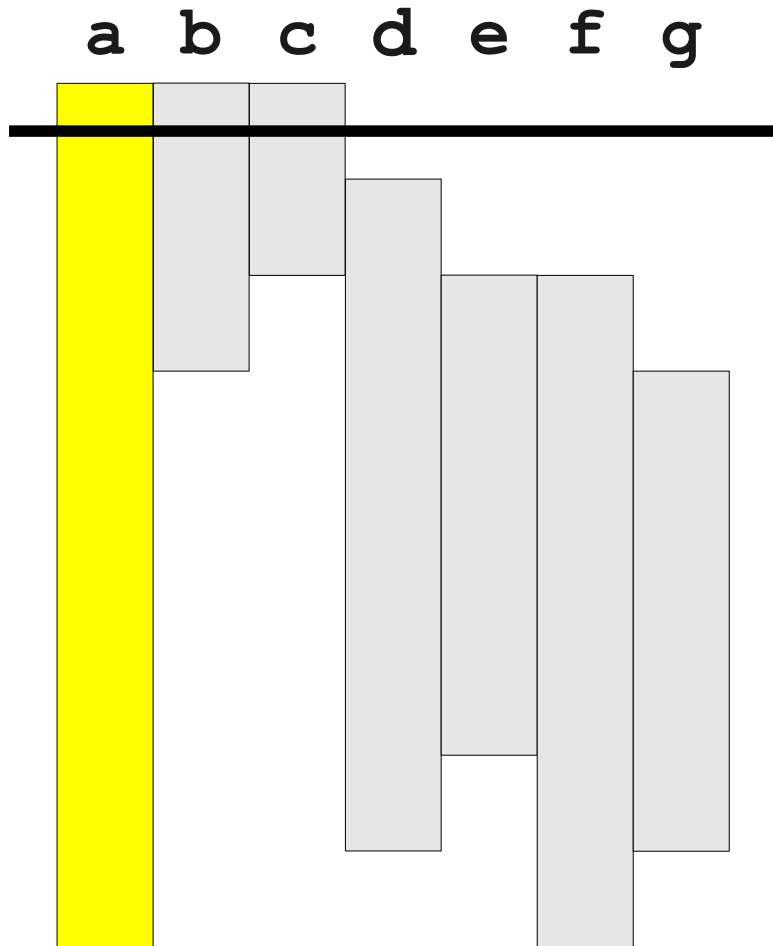
# Another Example



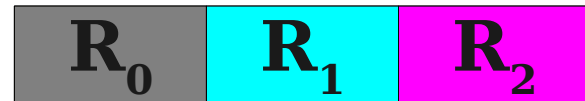
**Free Registers**



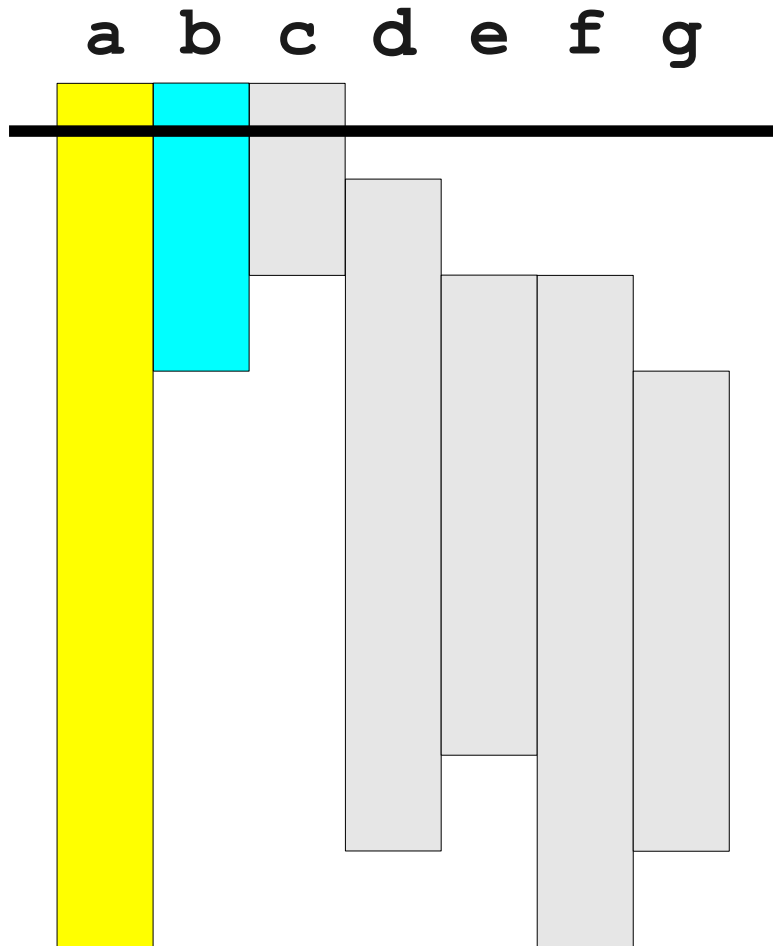
# Another Example



**Free Registers**



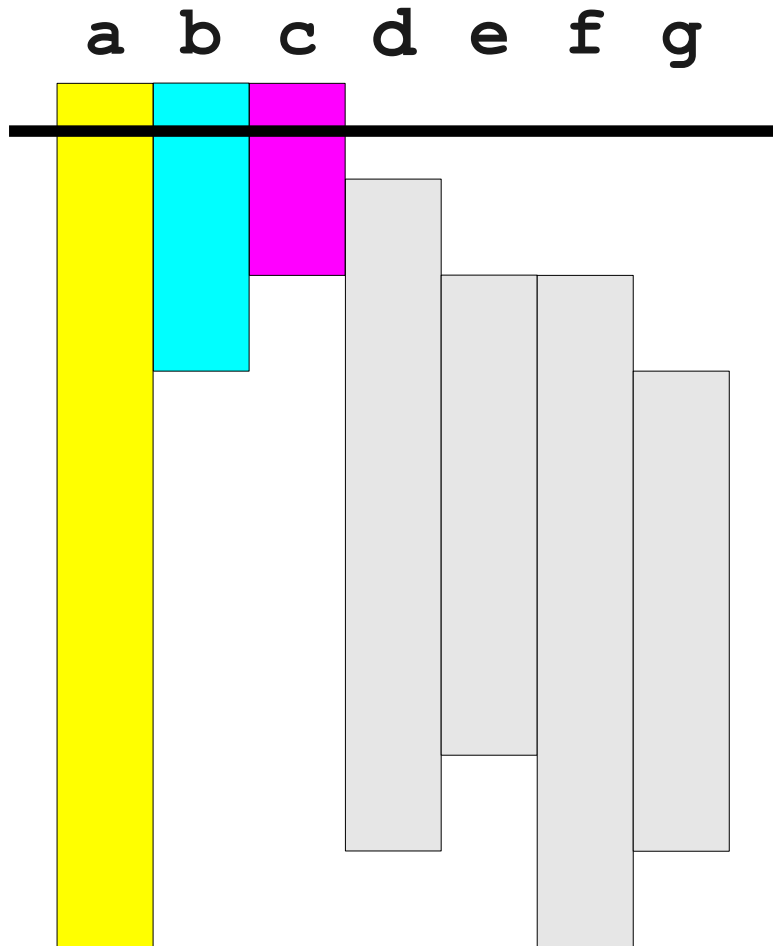
# Another Example



**Free Registers**



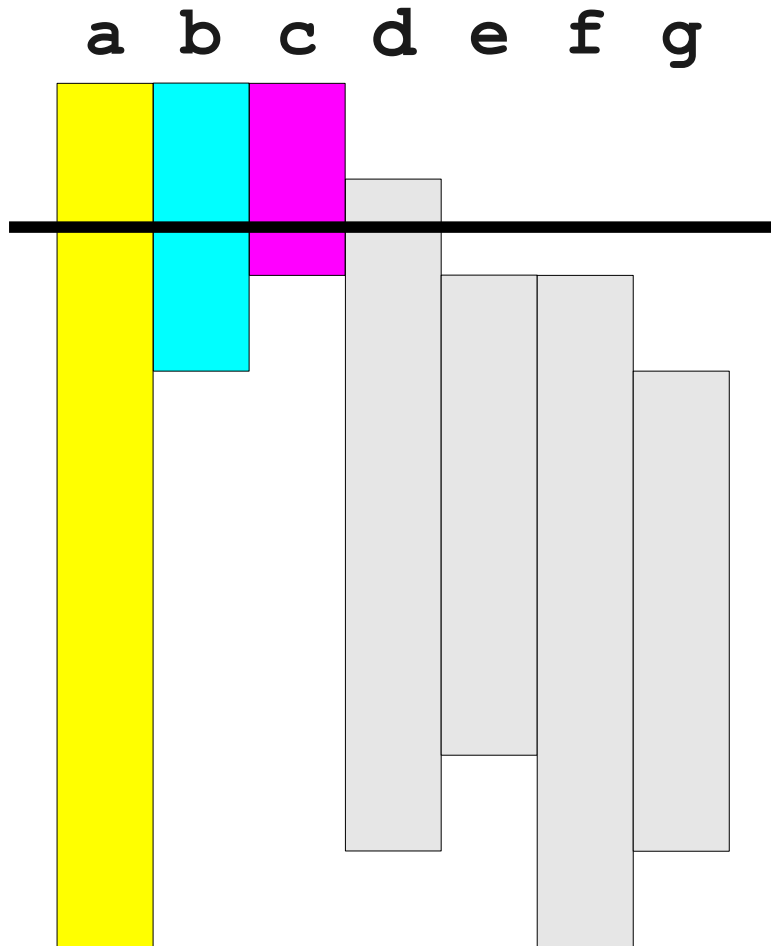
# Another Example



**Free Registers**



# Another Example



**Free Registers**



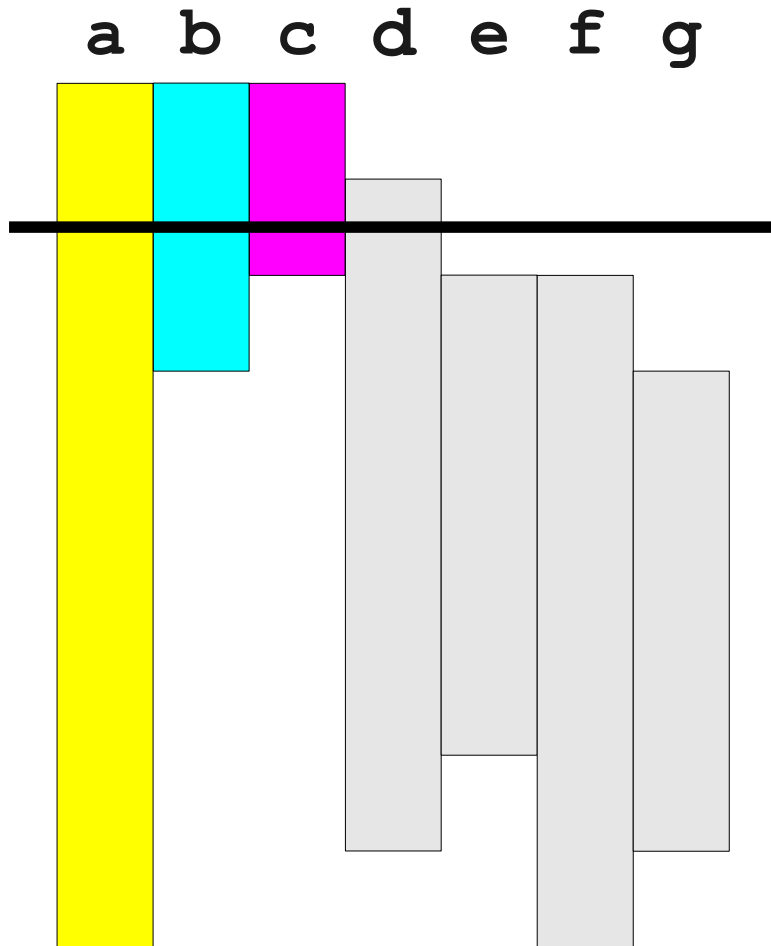
What do we do  
now?



# 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

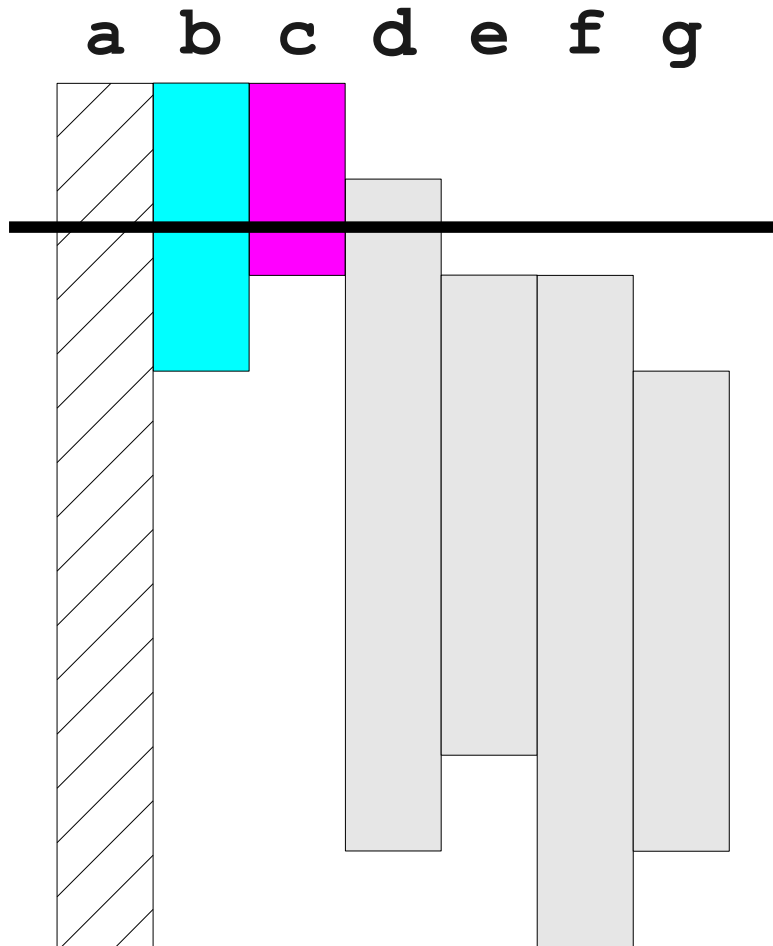


## Free Registers



What do we do  
now?

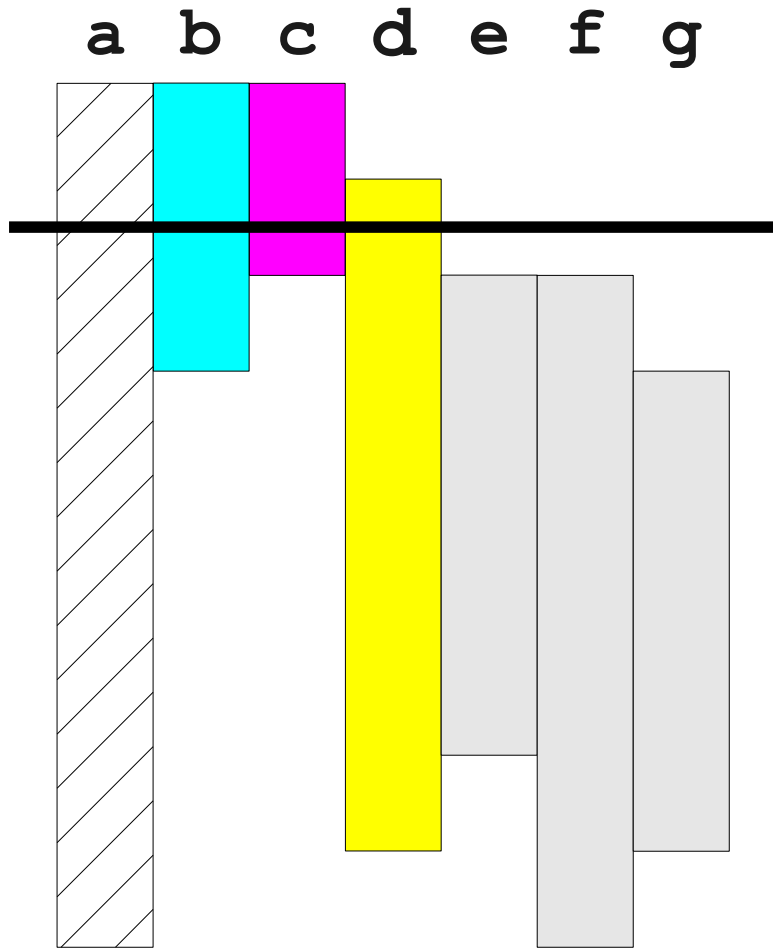
# Another Example



**Free Registers**



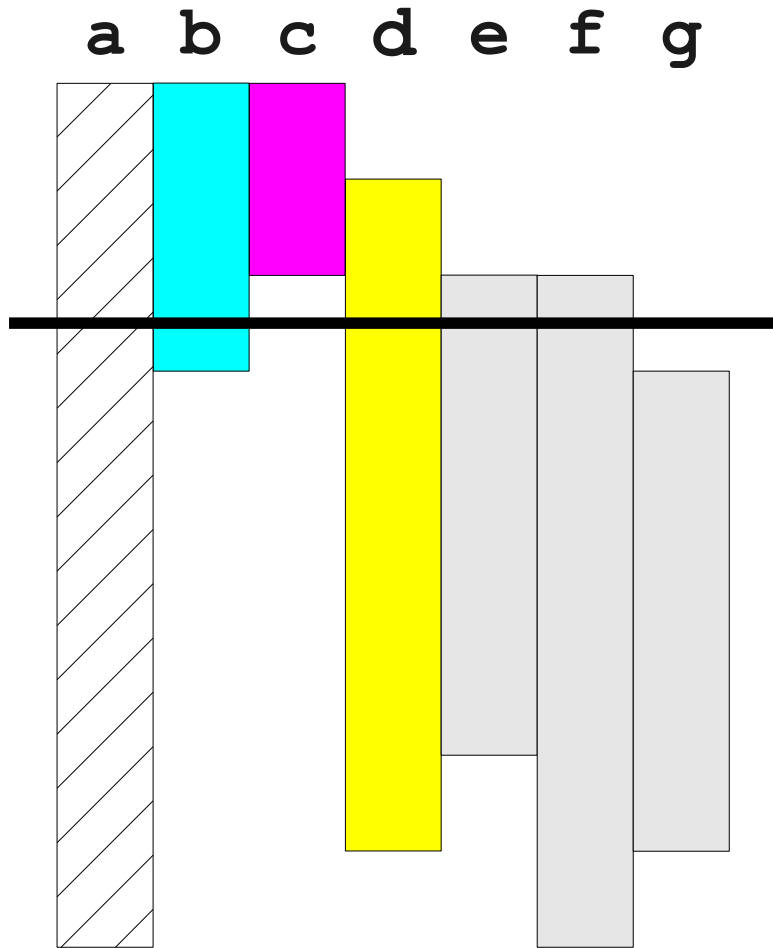
# Another Example



**Free Registers**



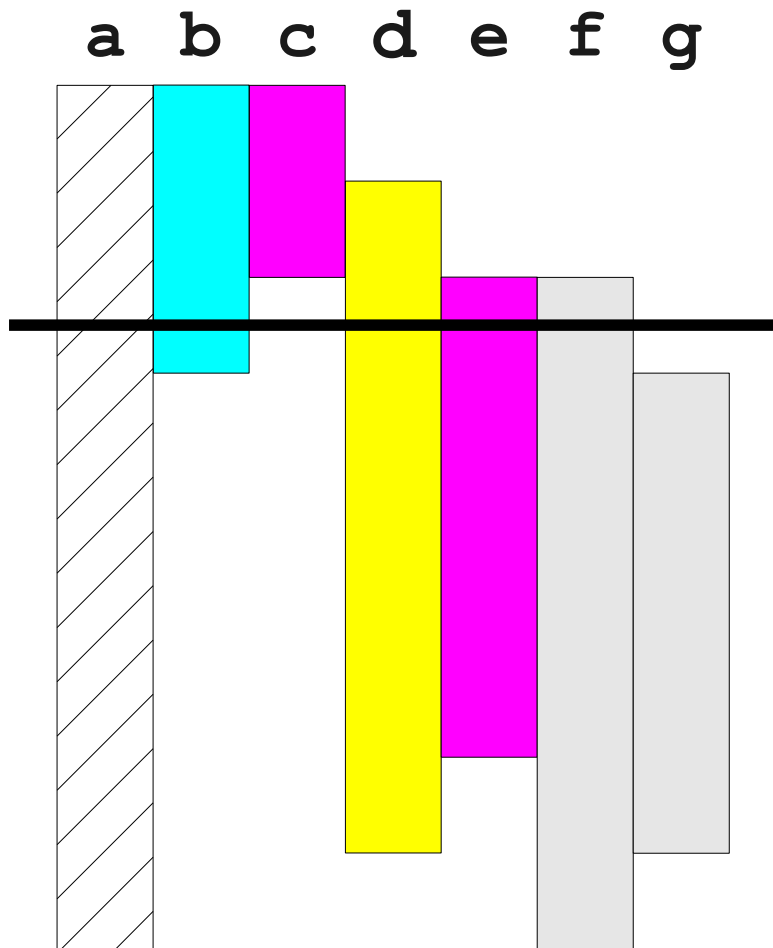
# Another Example



**Free Registers**



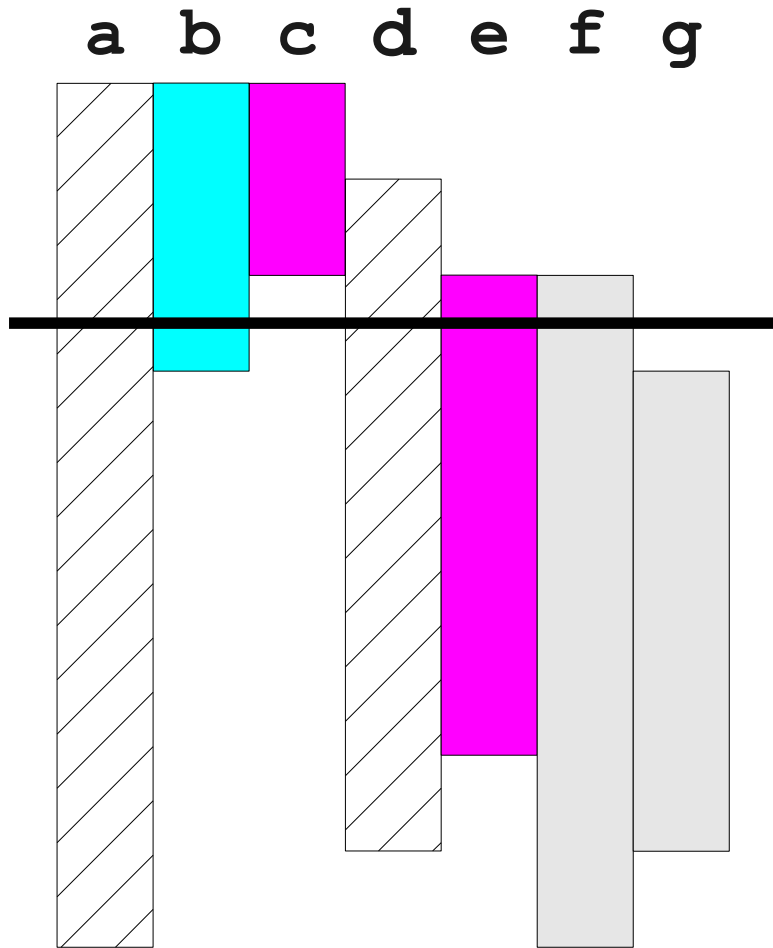
# Another Example



**Free Registers**



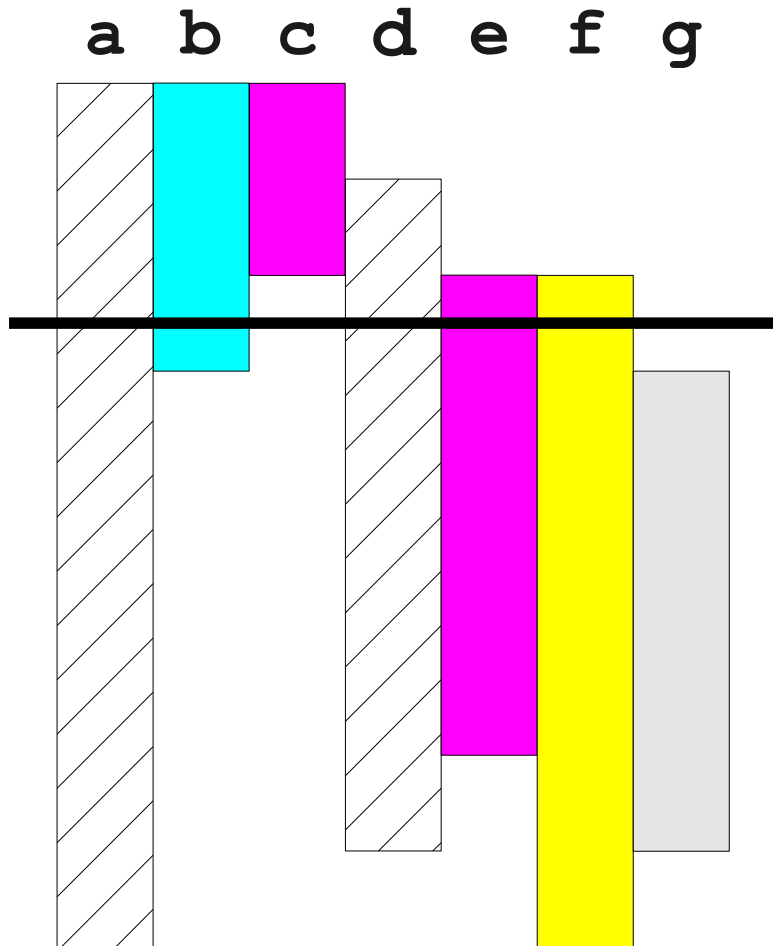
# Another Example



**Free Registers**



# Another Example

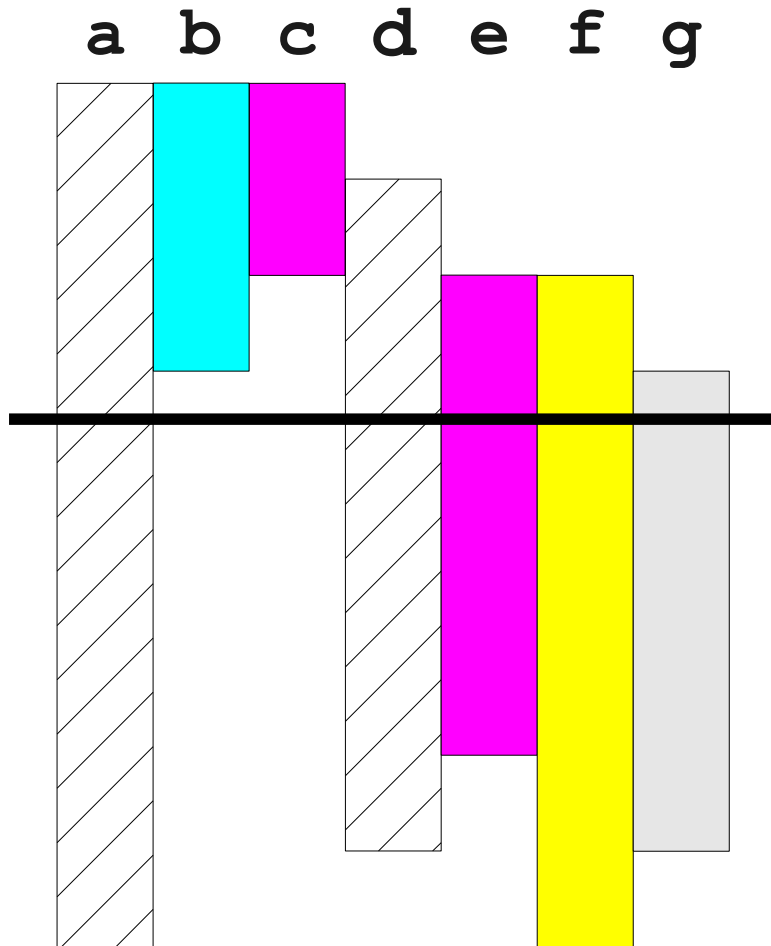


**Free Registers**

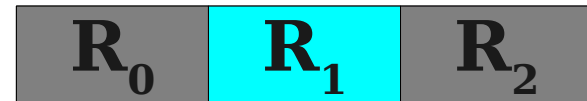




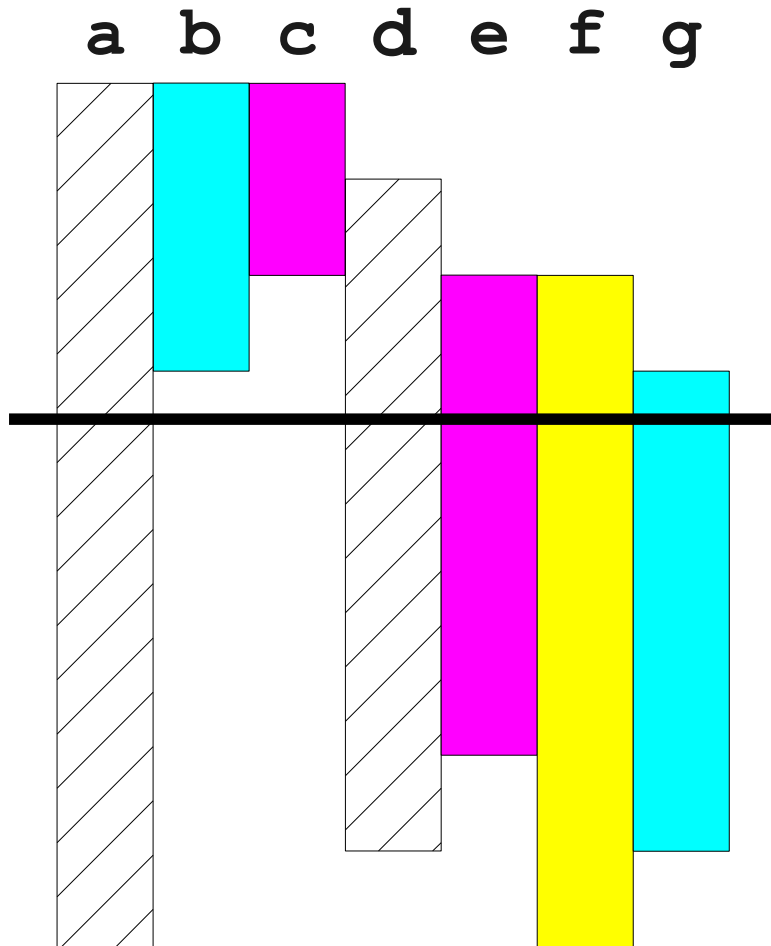
# Another Example



**Free Registers**



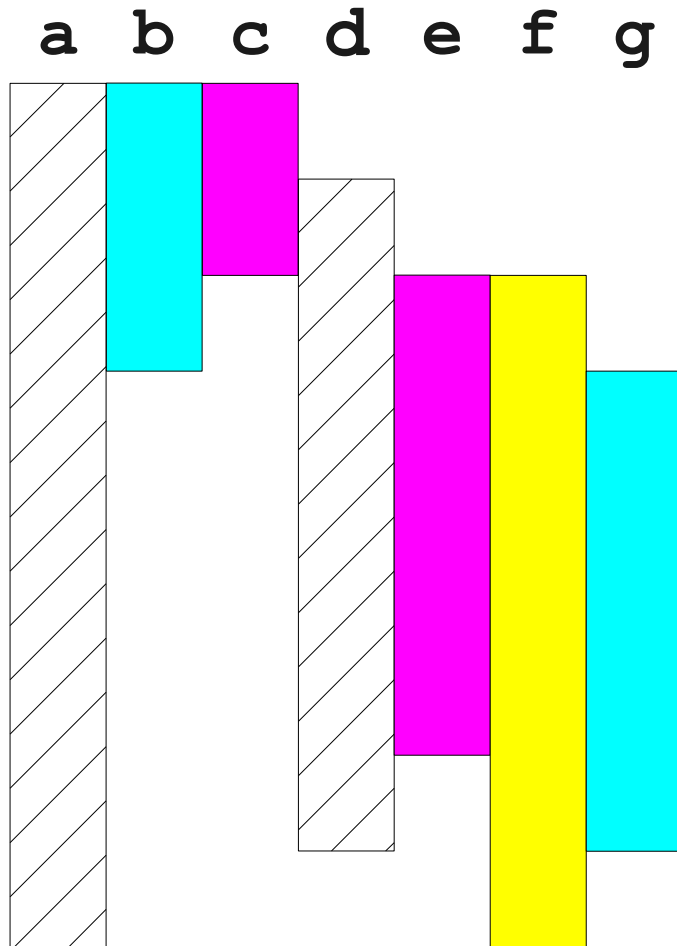
# Another Example



**Free Registers**



# Another Example



**Free Registers**

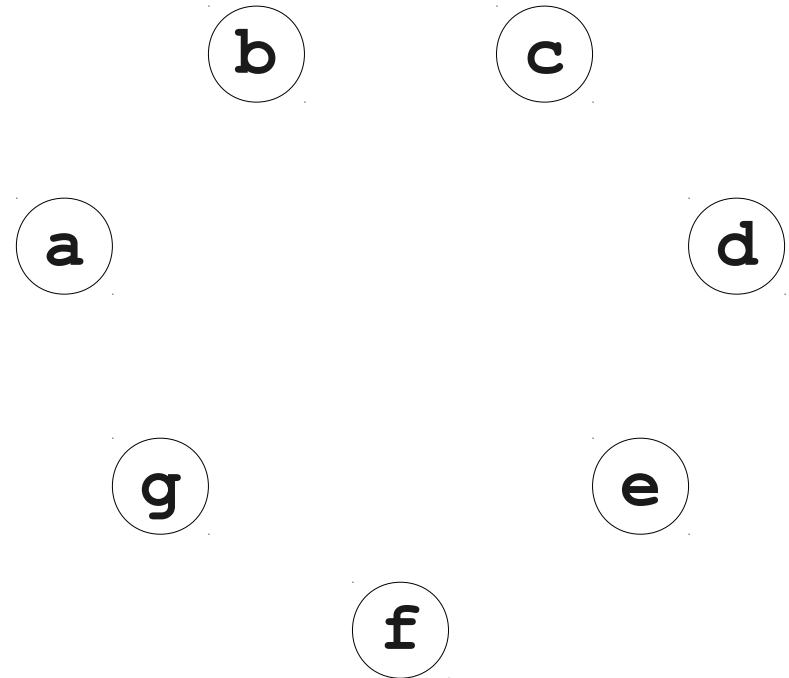
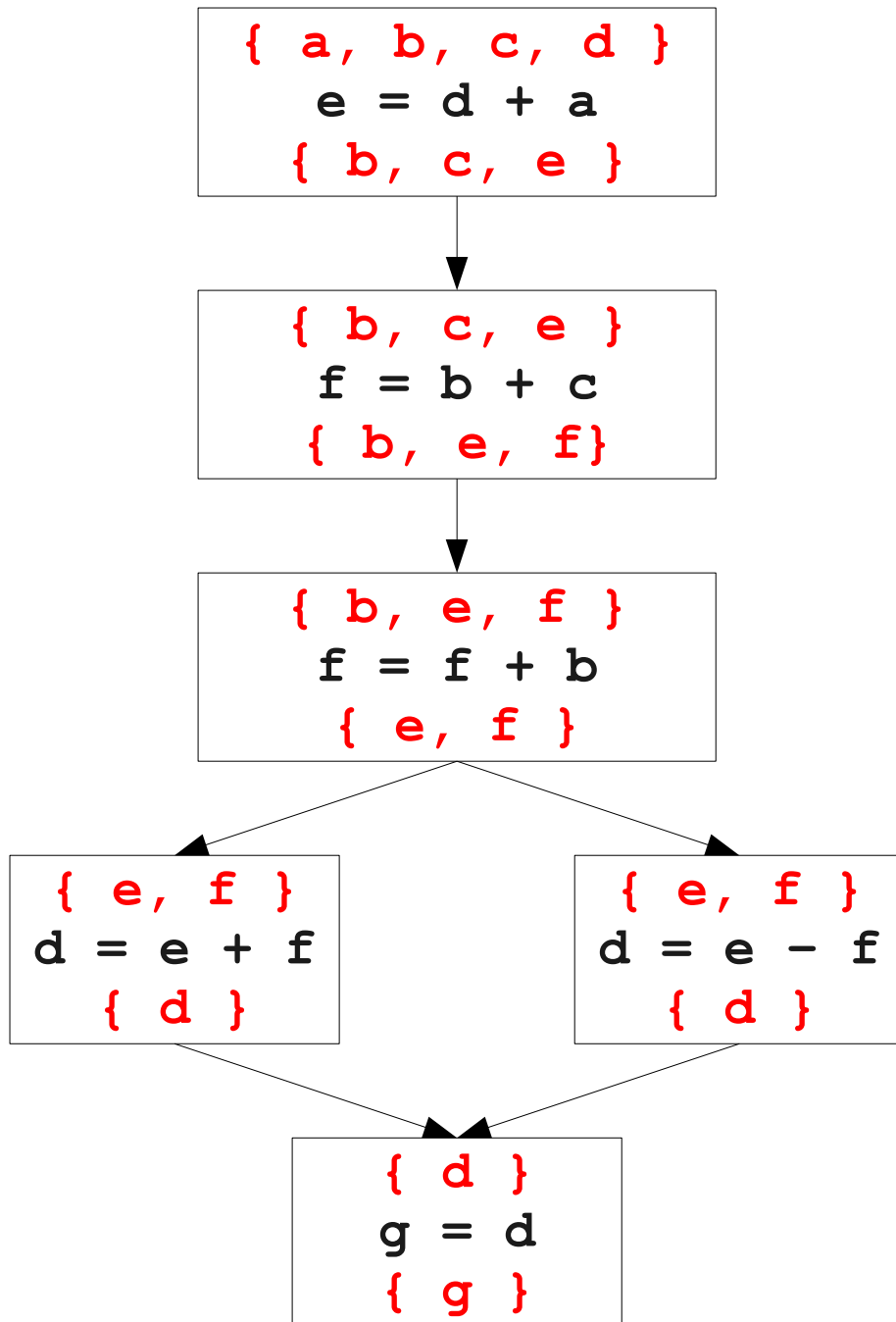


# Linear Scan Register Allocation

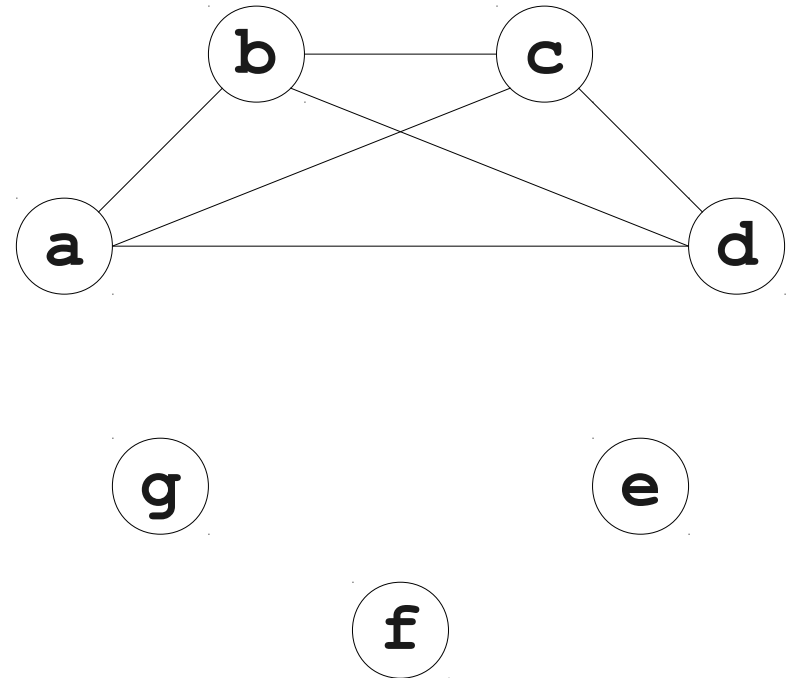
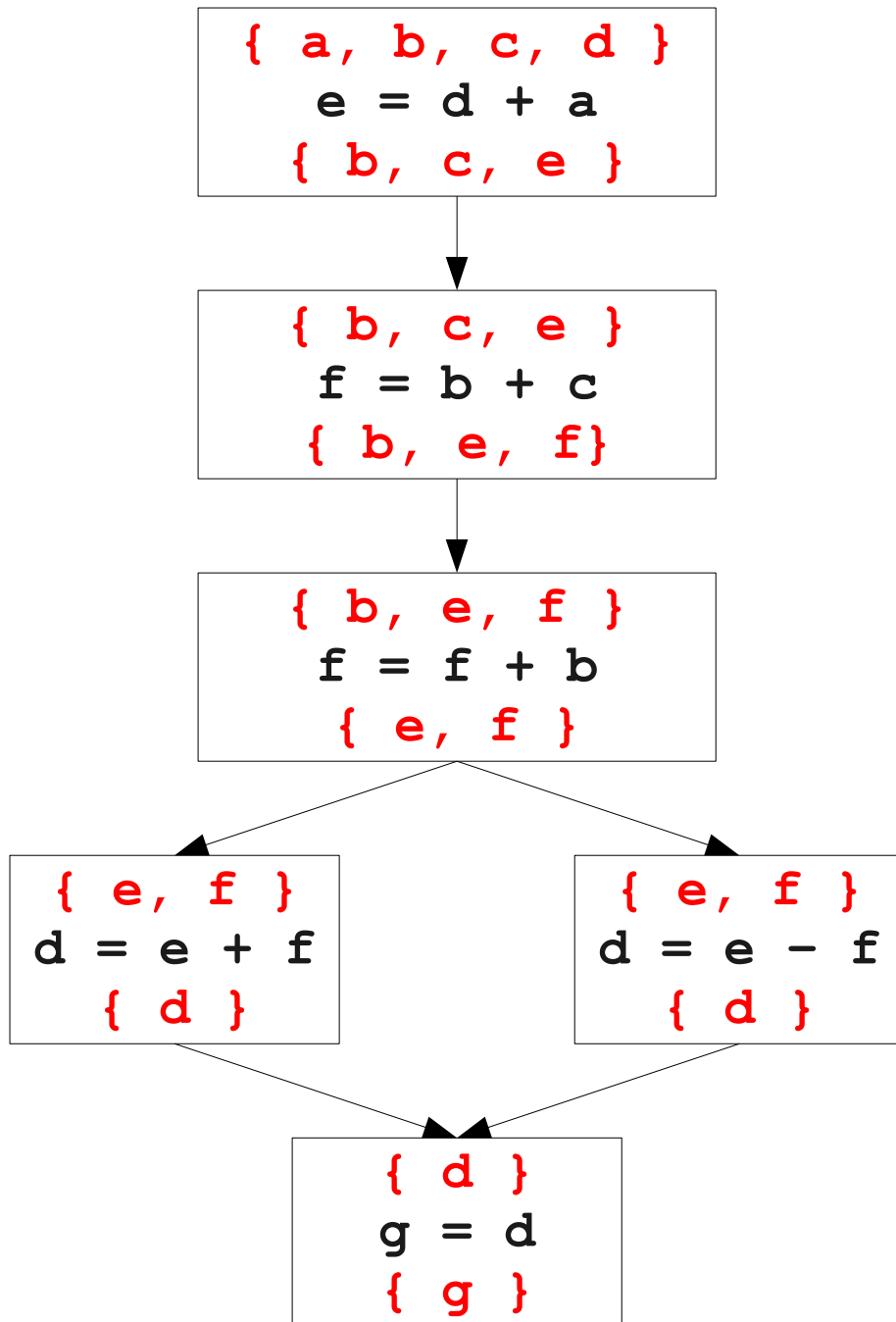
- This algorithm is called **linear scan register allocation** and is a comparatively new algorithm.
- Advantages:
  - Very efficient (after computing live intervals, runs in linear time)
  - Produces good code in many instances.
  - Allocation step works in one pass; can generate code during iteration.
  - 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

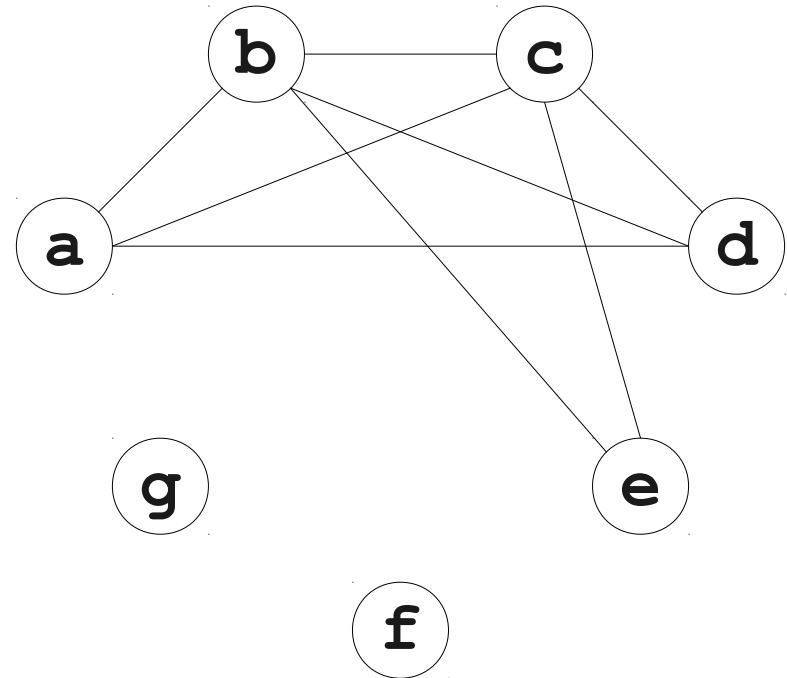
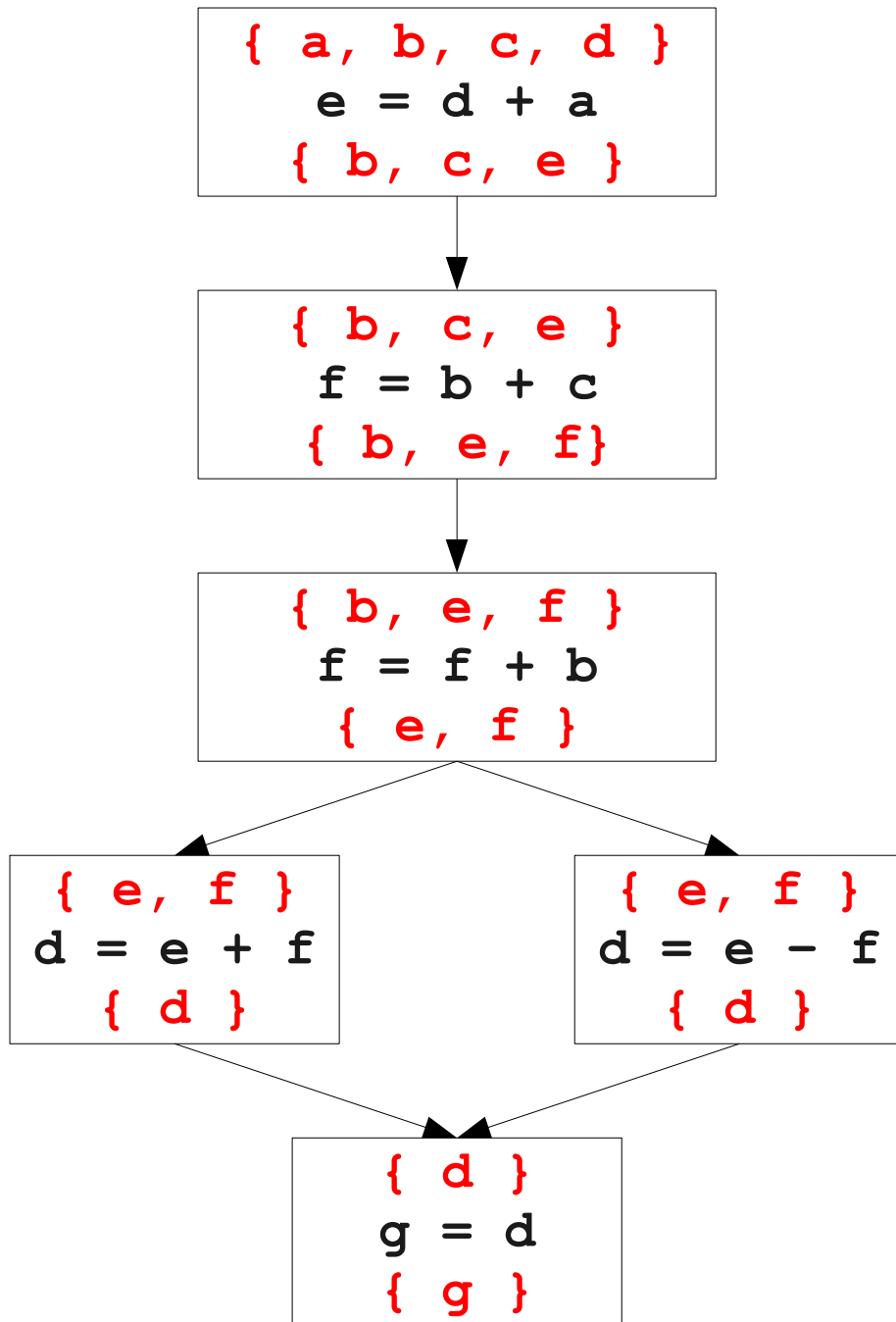
# An Entirely Different Approach



# An Entirely Different Approach

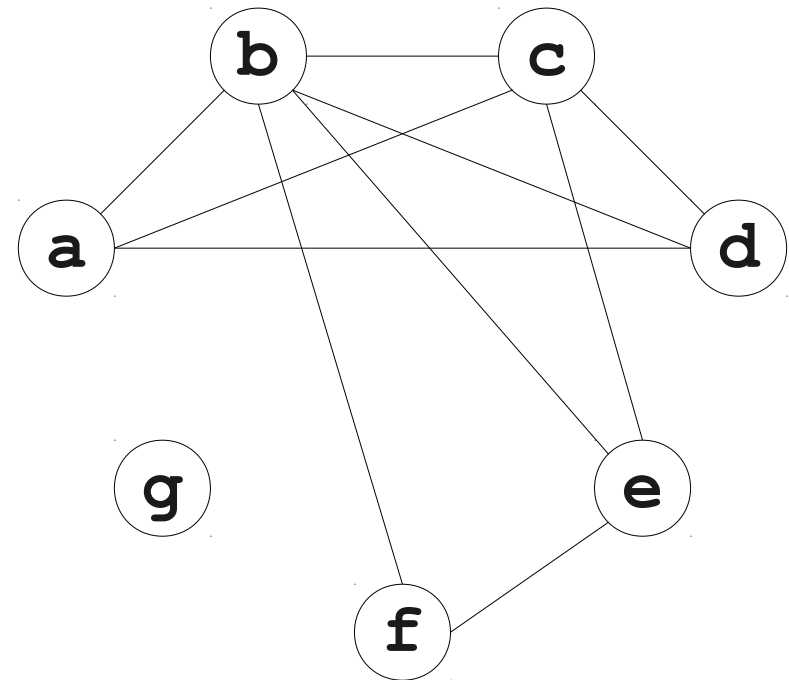
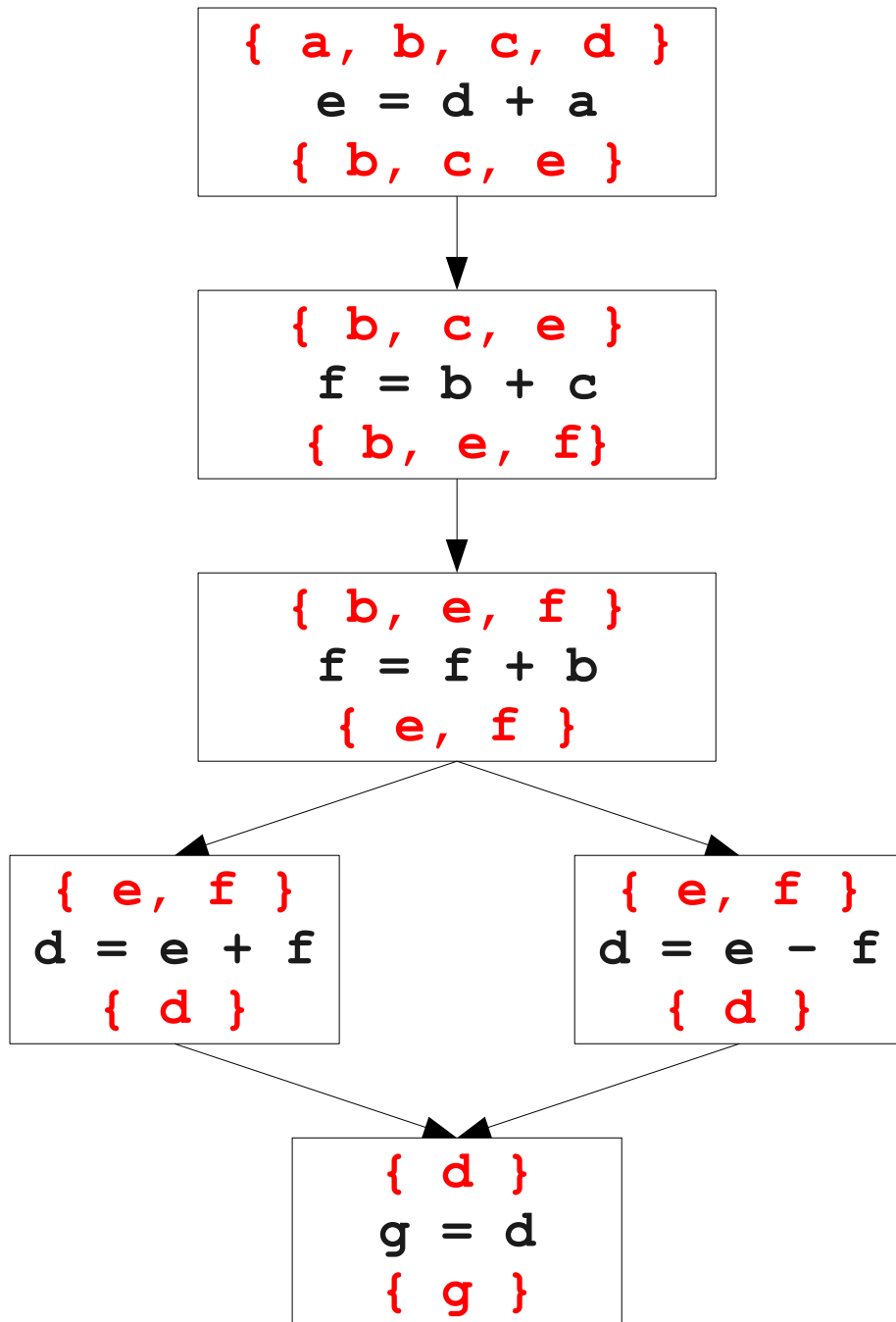


# An Entirely Different Approach

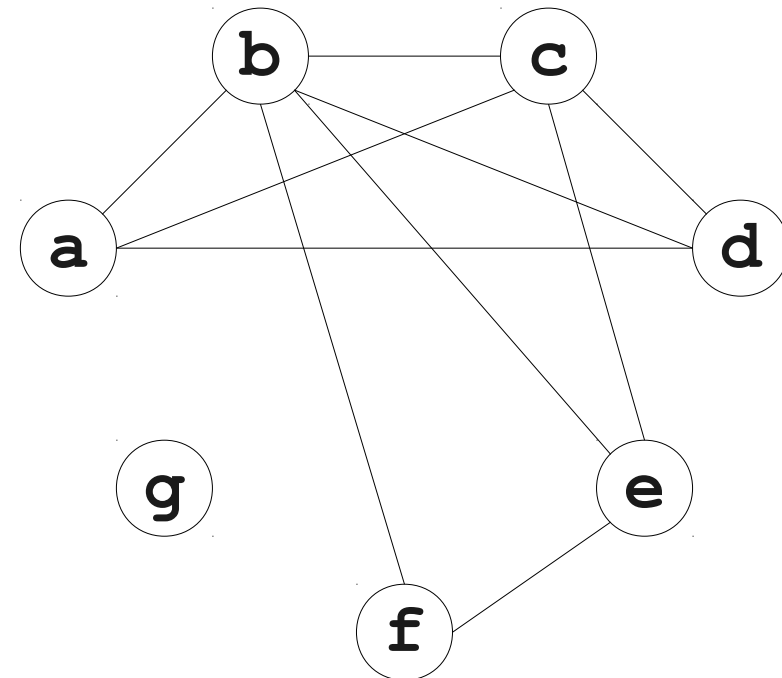
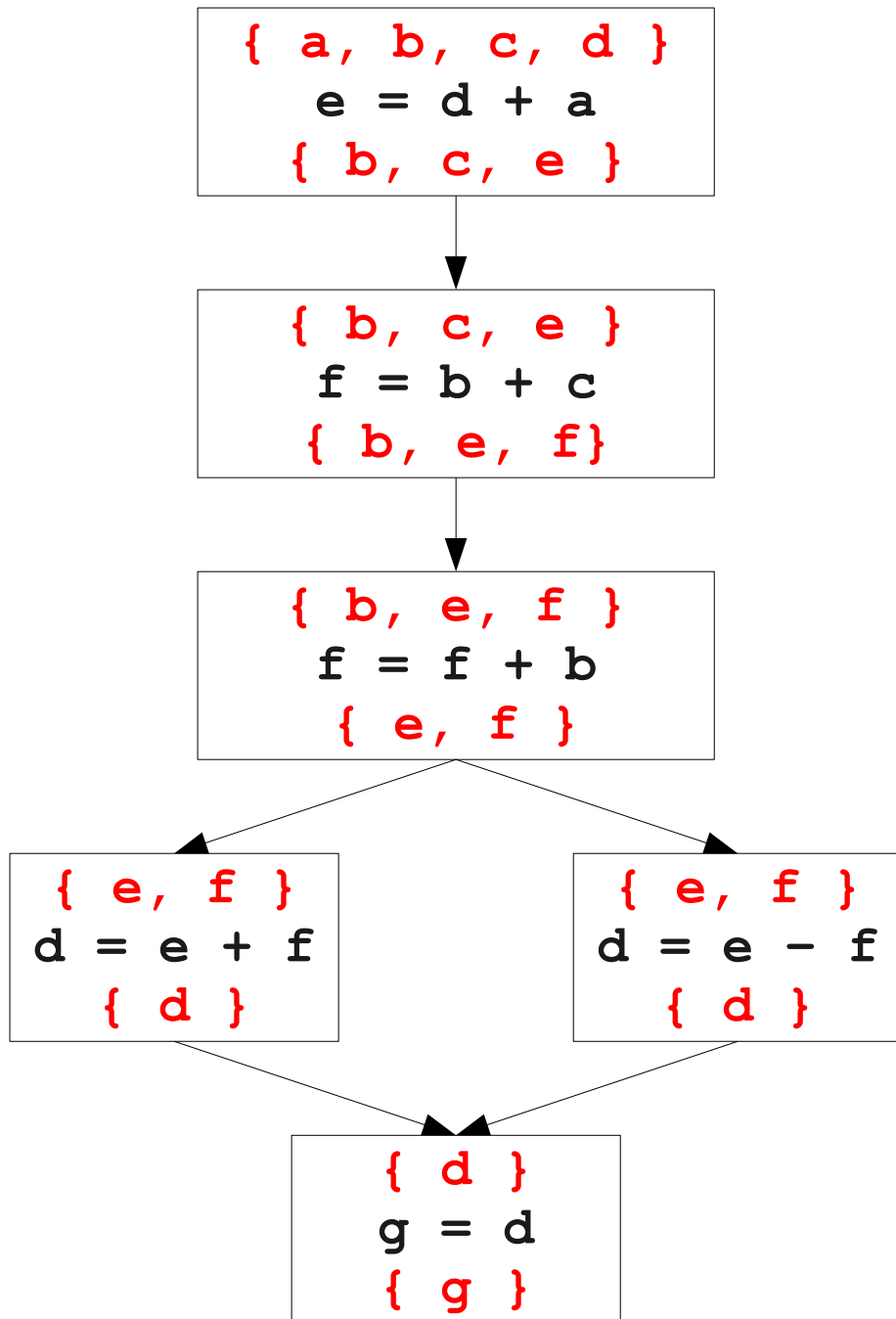




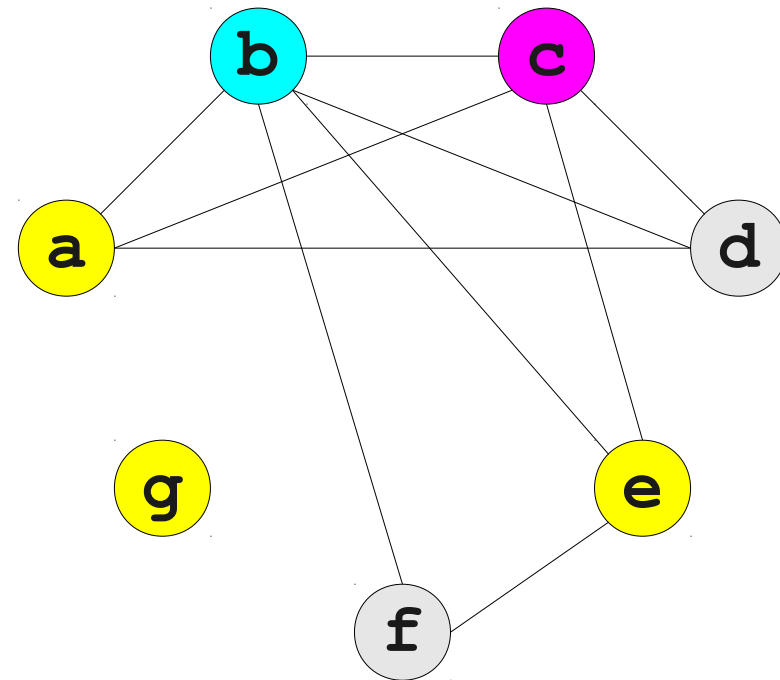
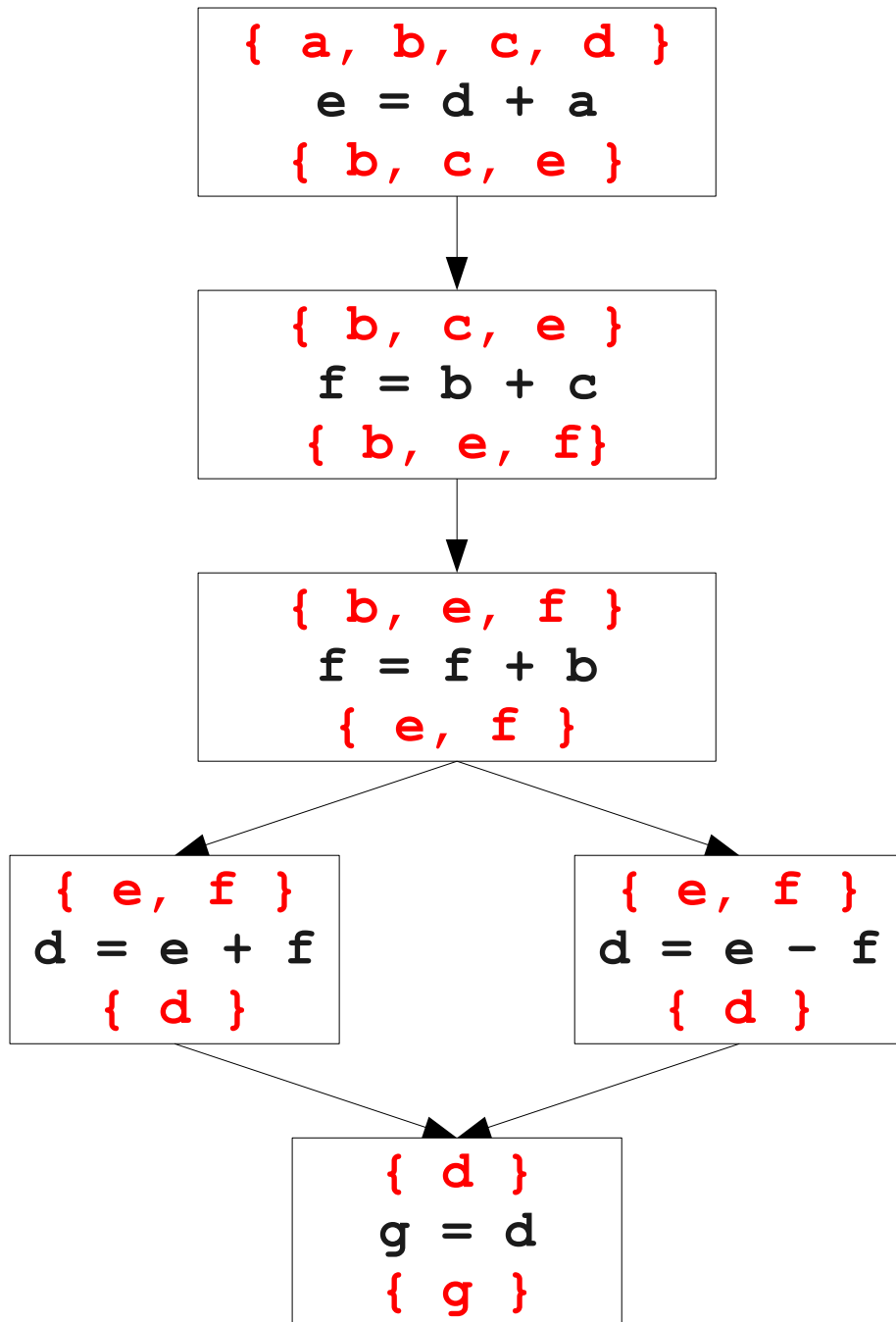
# An Entirely Different Approach



# An Entirely Different Approach



# An Entirely Different Approach



# The Register Interference Graph

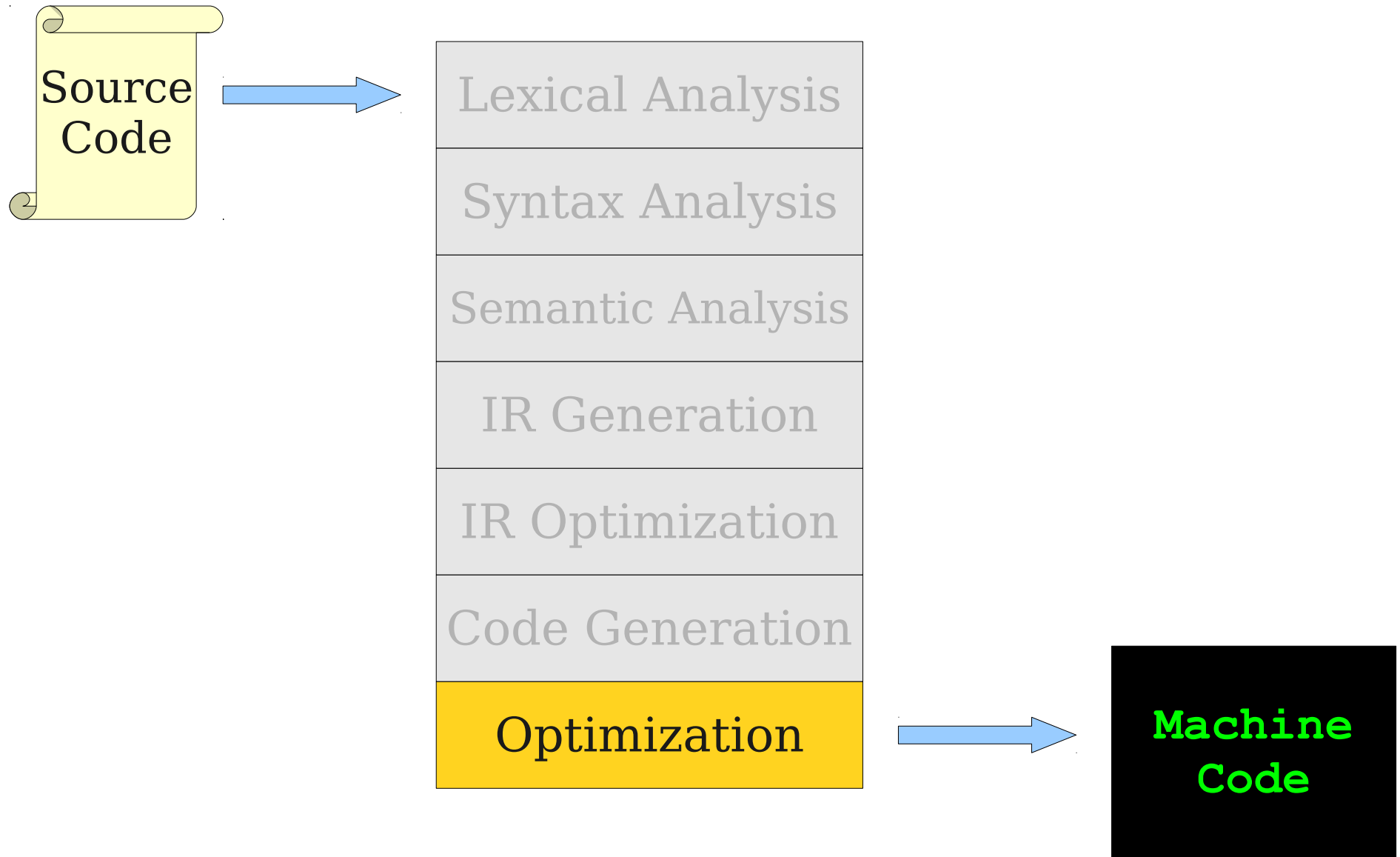
- The **register interference graph** (RIG) of a control-flow graph is an undirected graph where
  - Each node is a variable.
  - There is an edge between two variables that are live at the same program point.
- Perform register allocation by assigning each variable a different register from all of its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

# Code Optimization

# Where We Are



# 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
```

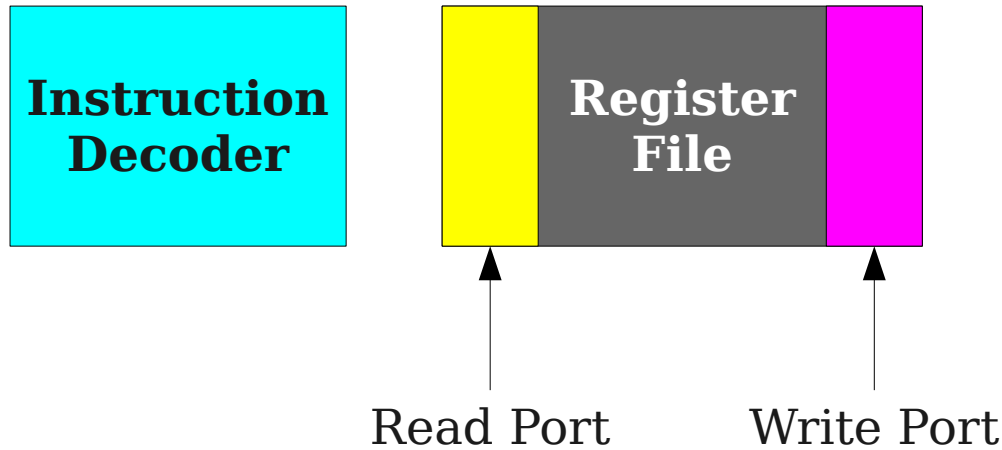
# Processor Pipelines

**Instruction  
Decoder**

**Register  
File**

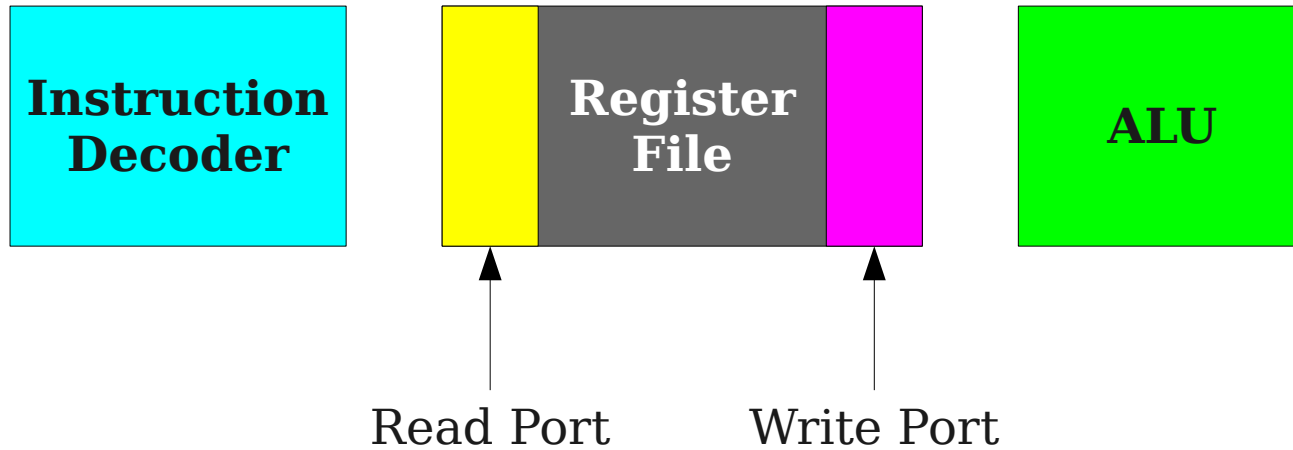
```
add $t2, $t0, $t1    # $t2 = $t0 + $t1
add $t5, $t3, $t4    # $t5 = $t3 + $t4
add $t8, $t6, $t7    # $t8 = $t6 + $t7
```

# Processor Pipelines



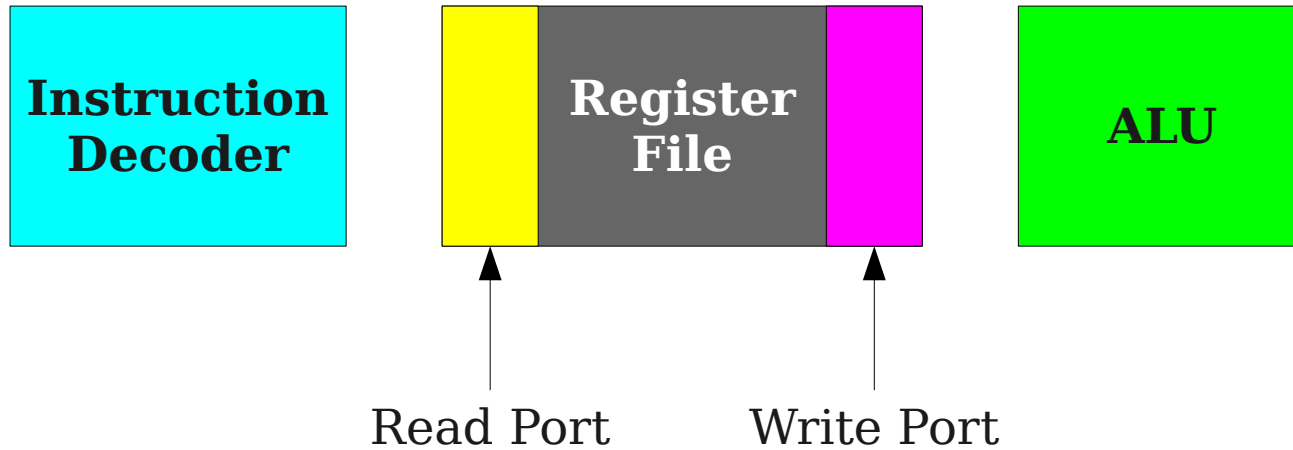
```
add $t2, $t0, $t1    # $t2 = $t0 + $t1
add $t5, $t3, $t4    # $t5 = $t3 + $t4
add $t8, $t6, $t7    # $t8 = $t6 + $t7
```

# 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

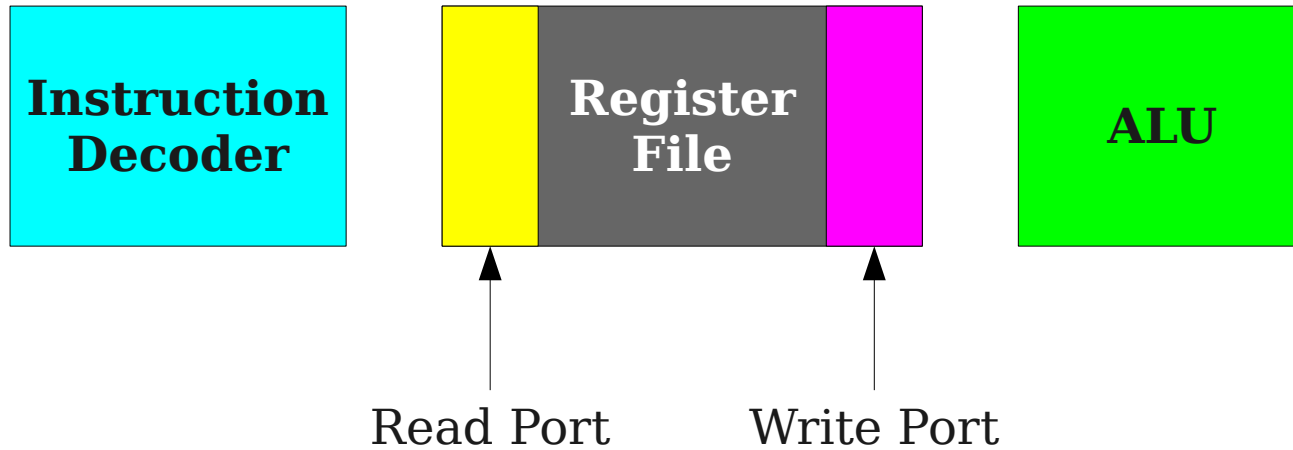


```
add $t2, $t0, $t1    # $t2 = $t0 + $t1
add $t5, $t3, $t4    # $t5 = $t3 + $t4
add $t8, $t6, $t7    # $t8 = $t6 + $t7
```

[illegible]



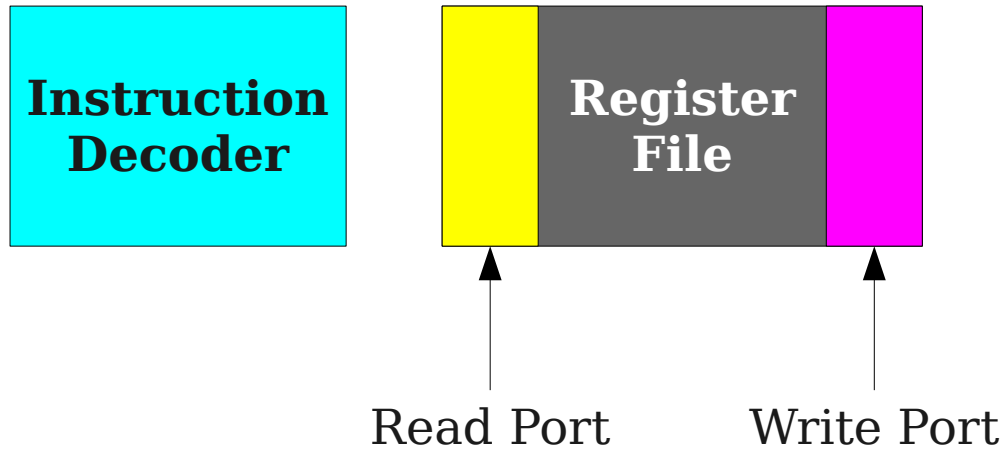
# Processor Pipelines



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

[illegible]

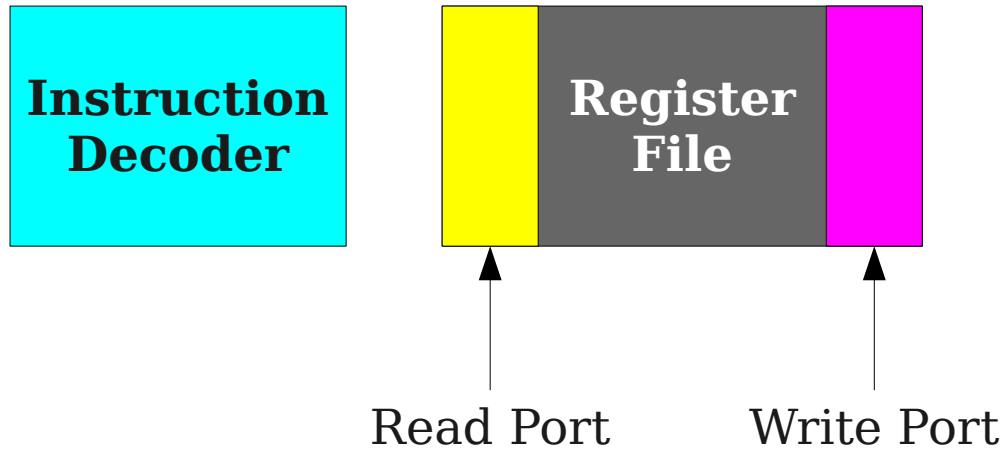
# Processor Pipelines



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

[illegible]

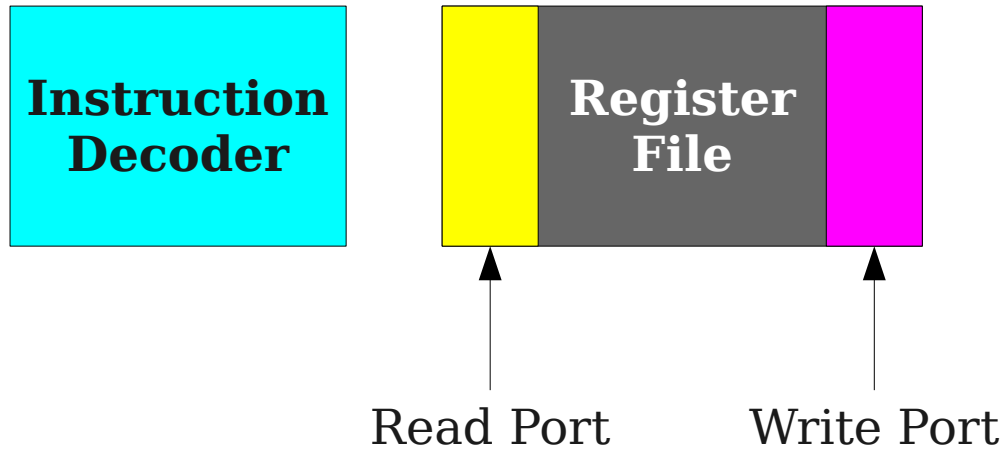
# Processor Pipelines



```
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

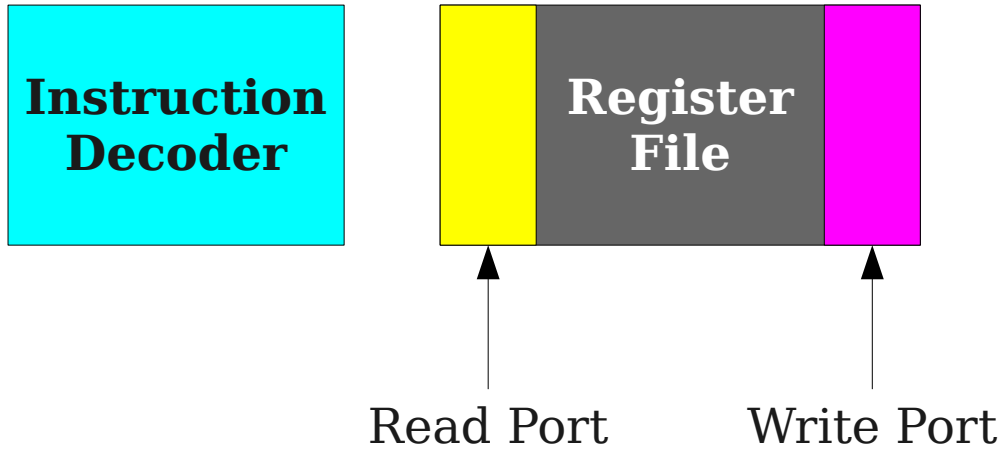
# Processor Pipelines



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

ID	RR	ALU	RW

# Processor Pipelines

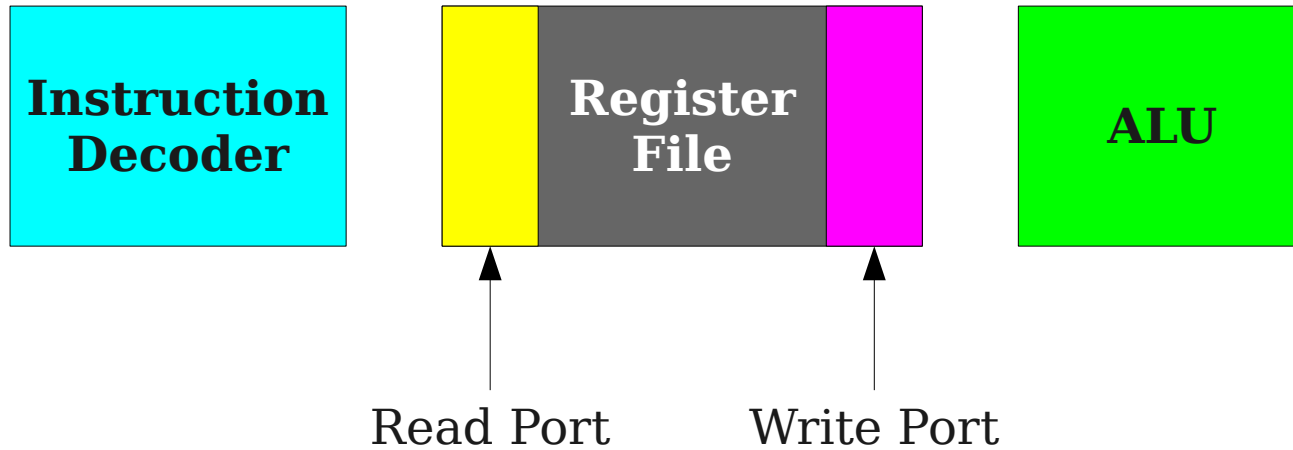


```
add $t2, $t0, $t1    # $t2 = $t0 + $t1
add $t5, $t3, $t4    # $t5 = $t3 + $t4
add $t8, $t6, $t7    # $t8 = $t6 + $t7
```

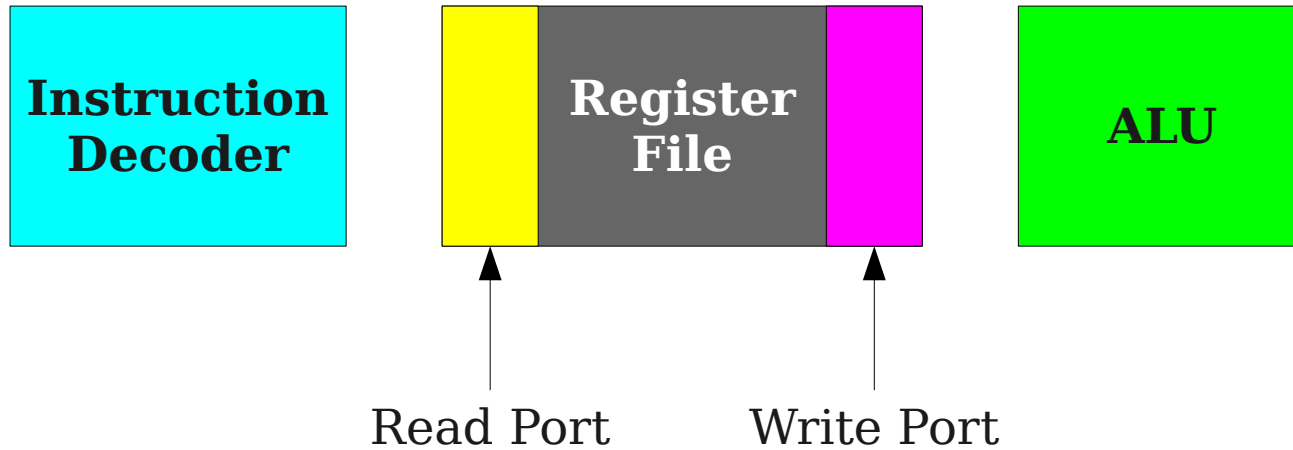
[illegible]

# Pipeline Hazards

# Pipeline Hazards



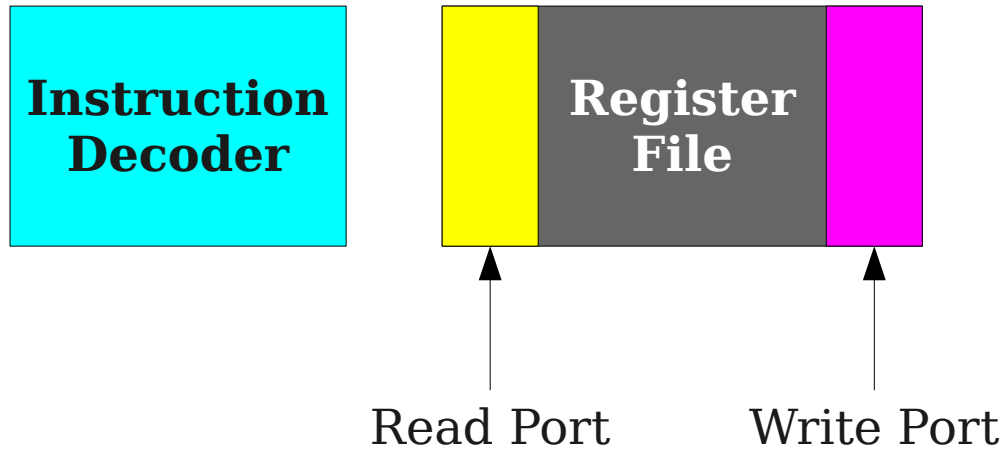
# Pipeline Hazards



```
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
```



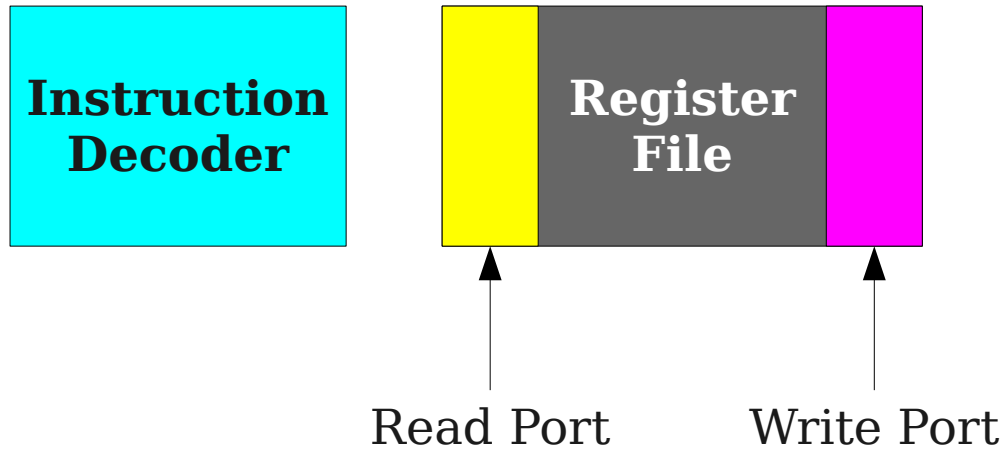
# Pipeline Hazards



```
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
```

[illegible]

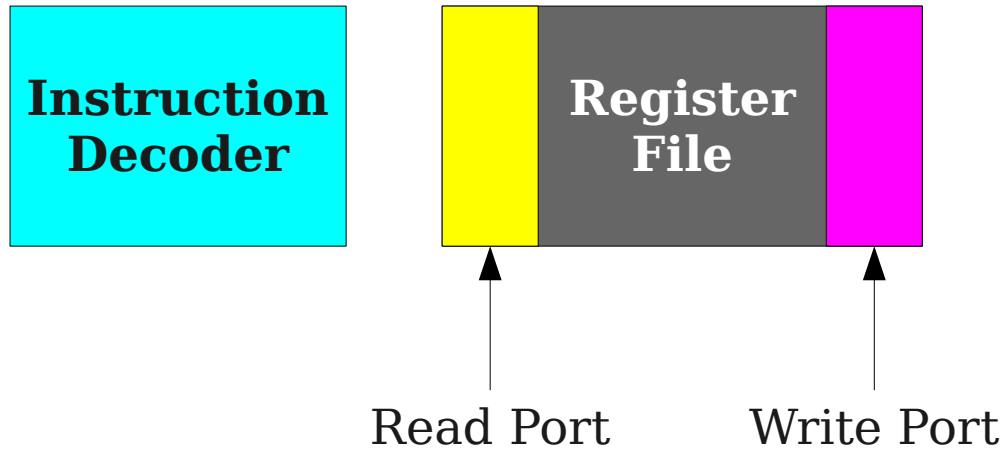
# Pipeline Hazards



```
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
```

[illegible]

# Pipeline Hazards

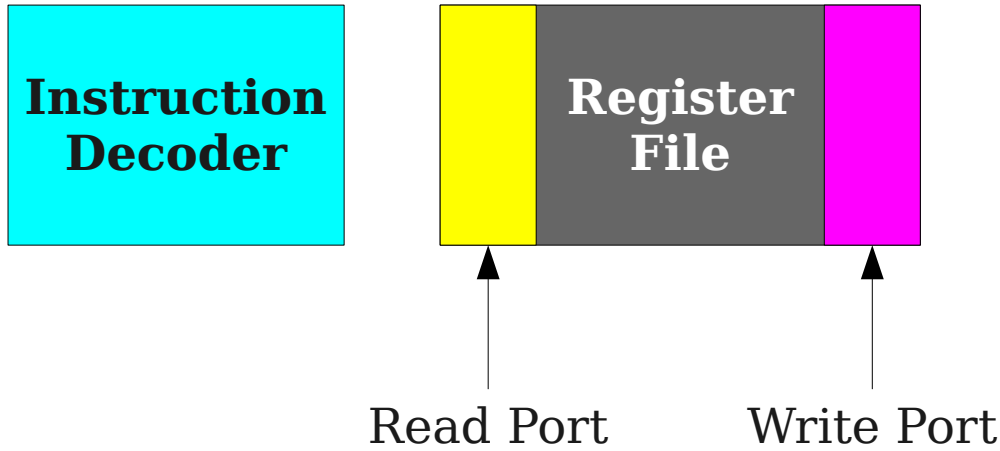


```
add $t2, $t0, $t1    # $t2 = $t0 + $t1
add $t4, $t3, $t2    # $t5 = $t3 + $t2
add $t7, $t5, $t6    # $t7 = $t5 + $t6
add $t0, $t0, $t7    # $t0 = $t0 + $t7
```

[illegible]

# Pipeline Hazards

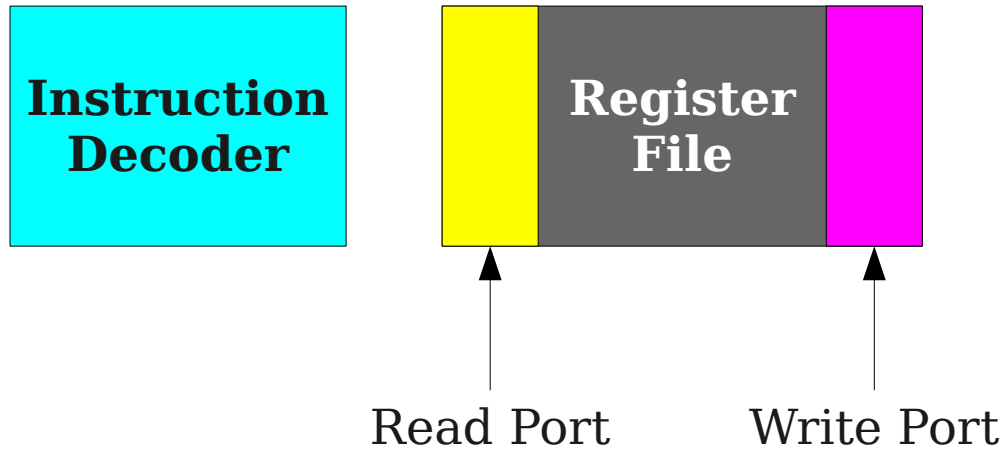
This value isn't ready yet!



```
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
```

[illegible]

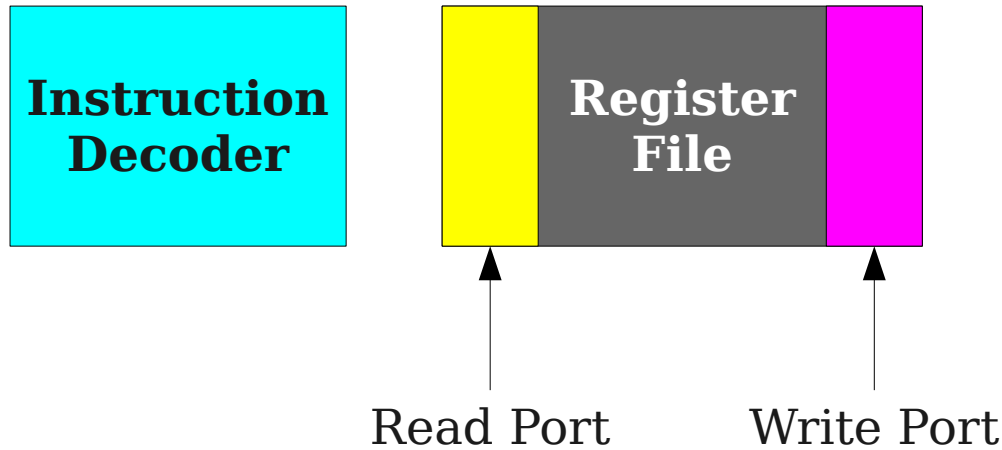
# Pipeline Hazards



```
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
```

[illegible]

# Pipeline Hazards

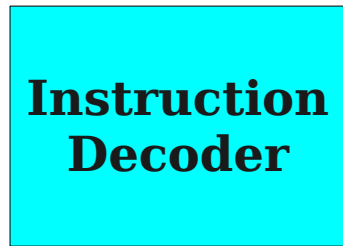


```
add $t2, $t0, $t1    # $t2 = $t0 + $t1
add $t4, $t3, $t2    # $t5 = $t3 + $t2
add $t7, $t5, $t6    # $t7 = $t5 + $t6
add $t0, $t0, $t7    # $t0 = $t0 + $t7
```

ID	RR	ALU	RW

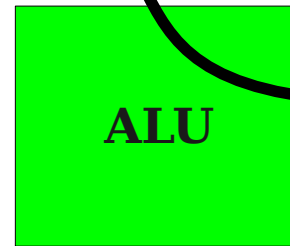
# Pipeline Hazards

stall pipeline until  
value is ready



## Read Port

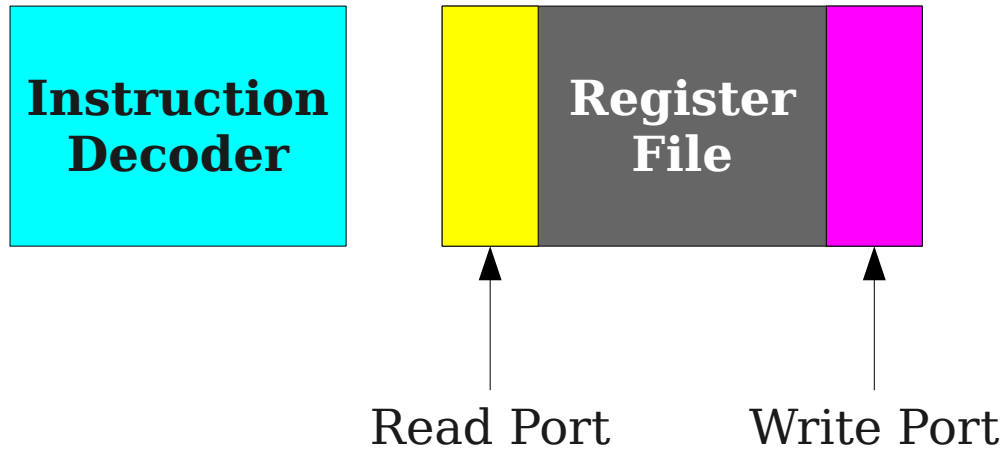
## Write Port



	ID	RR	ALU	RW
	Red			
	Green	Red		
▶	Green		Red	
	Green			Red
		Green		
			Green	
				Green

```
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

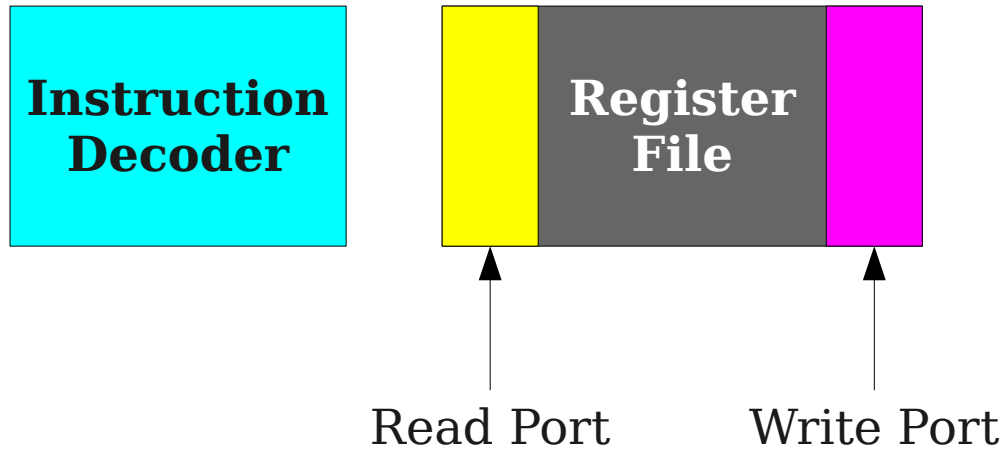


```
add $t2, $t0, $t1    # $t2 = $t0 + $t1
add $t4, $t3, $t2    # $t5 = $t3 + $t2
add $t7, $t5, $t6    # $t7 = $t5 + $t6
add $t0, $t0, $t7    # $t0 = $t0 + $t7
```

ID	RR	ALU	RW



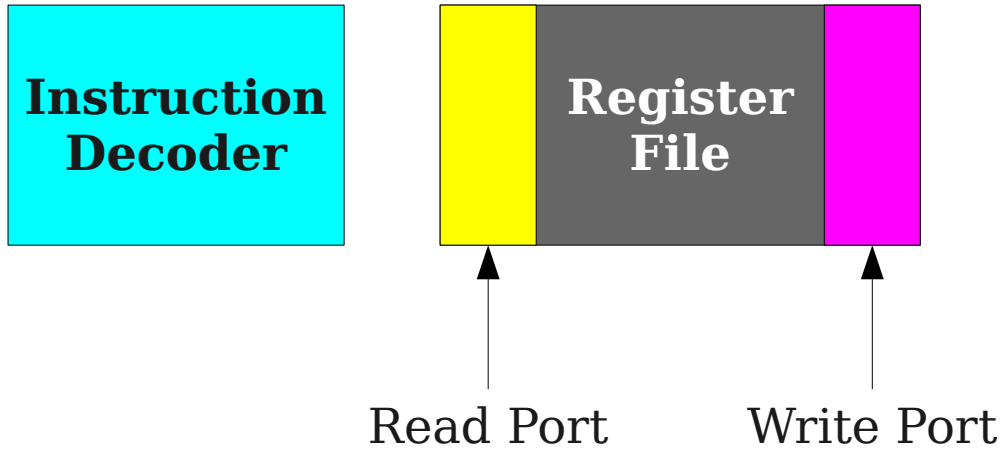
# Pipeline Hazards



```
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

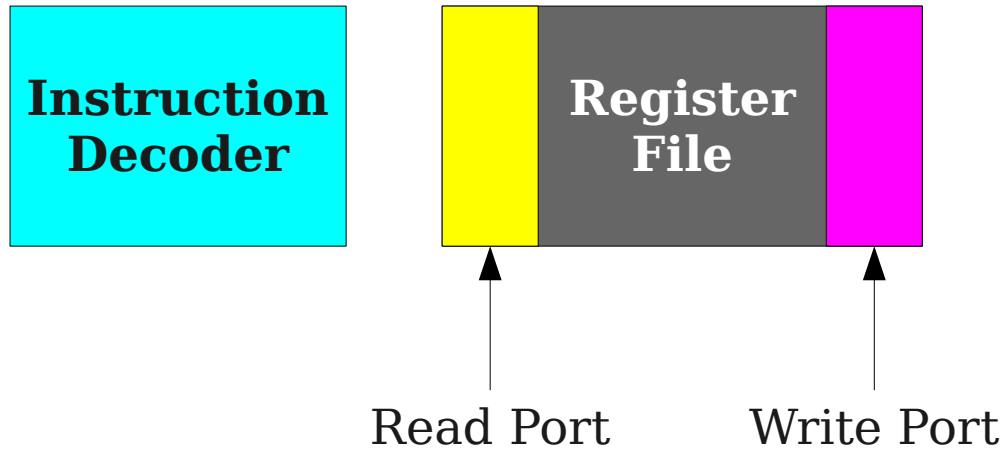
# Pipeline Hazards



```
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

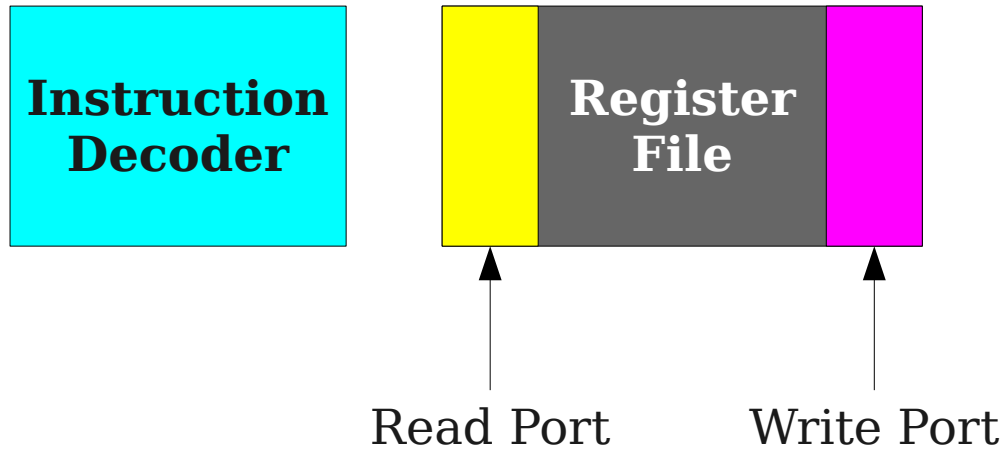
# Pipeline Hazards



```
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
```

[illegible]

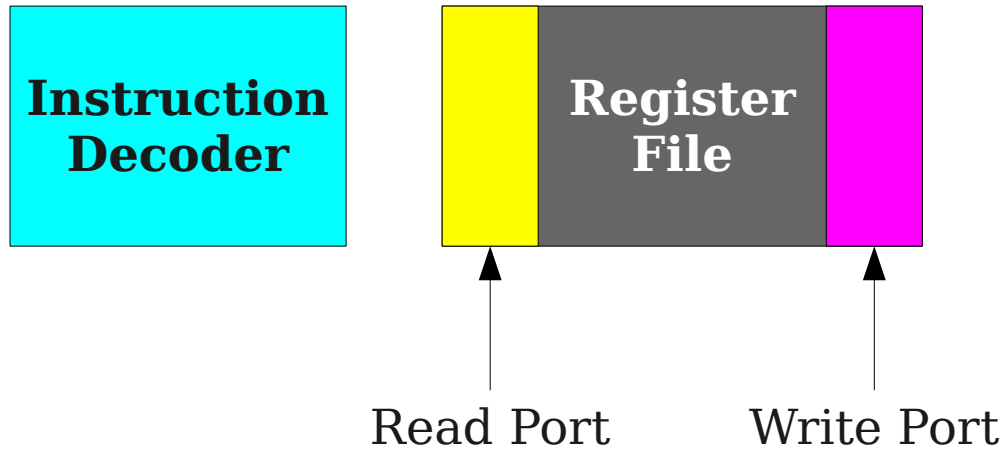
# Pipeline Hazards



```
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
```

[illegible]

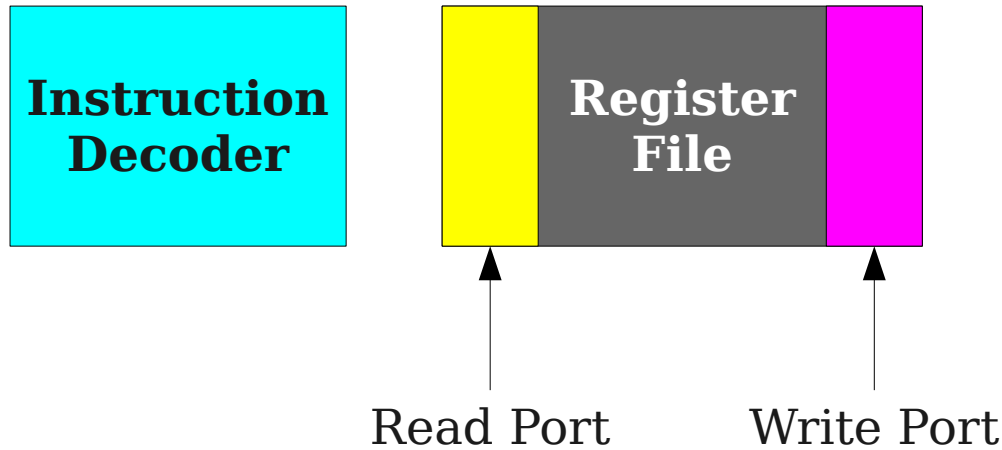
# Pipeline Hazards



```
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
```

[illegible]

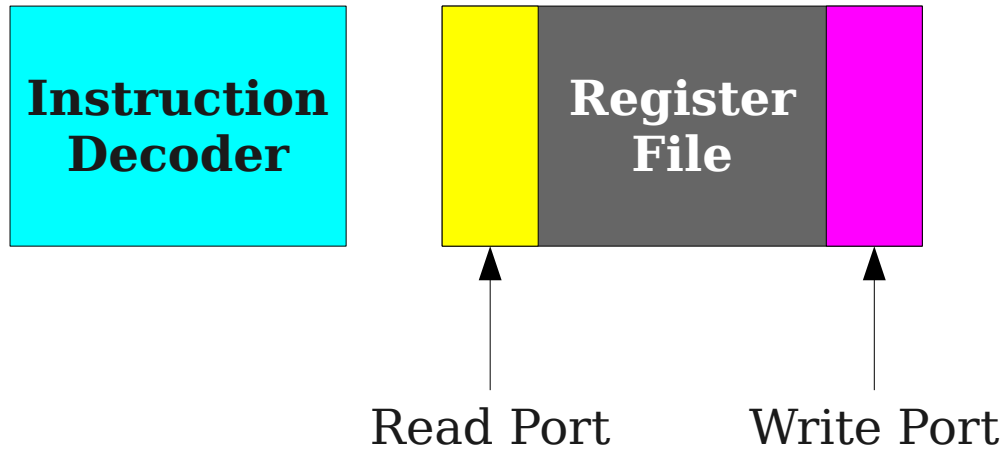
# Pipeline Hazards



```
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
```

[illegible]

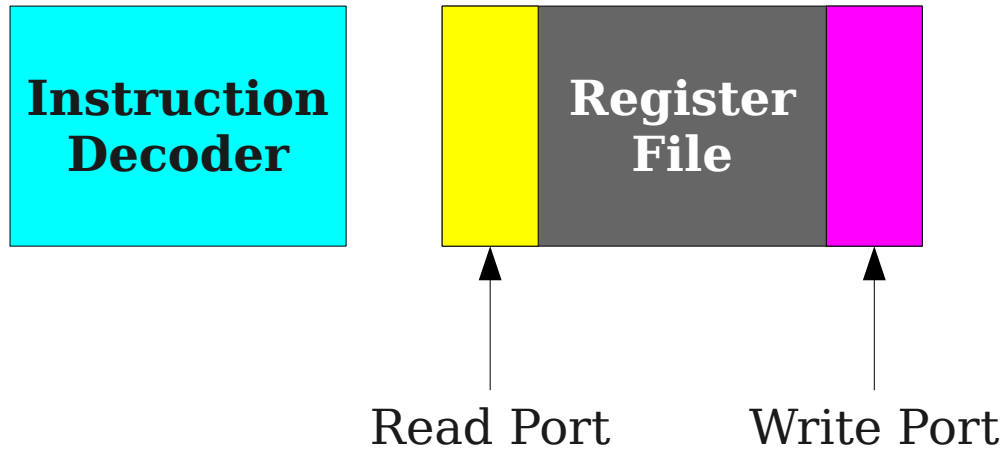
# Pipeline Hazards



```
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
```

ID	RR	ALU	RW

# Pipeline Hazards

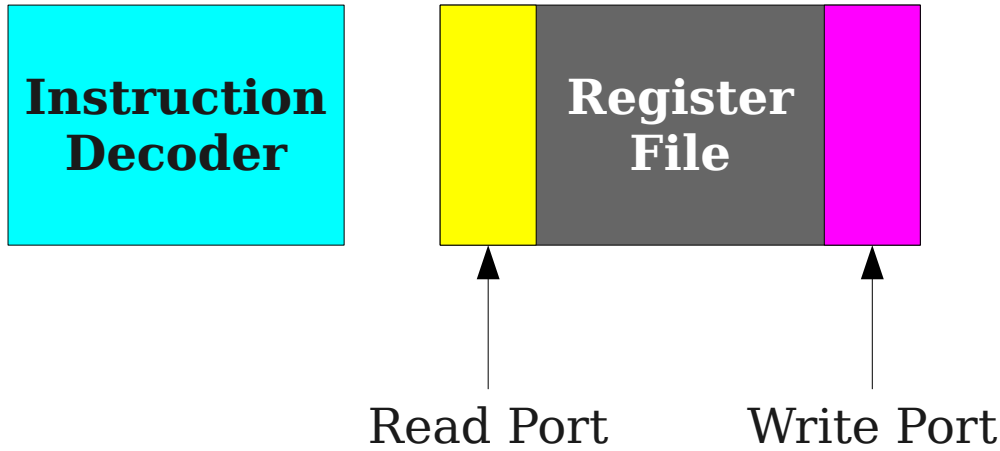


```
add $t2, $t0, $t1    # $t2 = $t0 + $t1
add $t7, $t5, $t6     # $t7 = $t5 + $t6
add $t4, $t3, $t2     # $t5 = $t3 + $t2
add $t0, $t0, $t7     # $t0 = $t0 + $t7
```

ID	RR	ALU	RW



# 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!

ID	RR	ALU	RW

# 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)**

$x = \dots$   
 $\dots = x$

**Write-after-Read  
(WAR)**

$\dots = x$   
 $x = \dots$

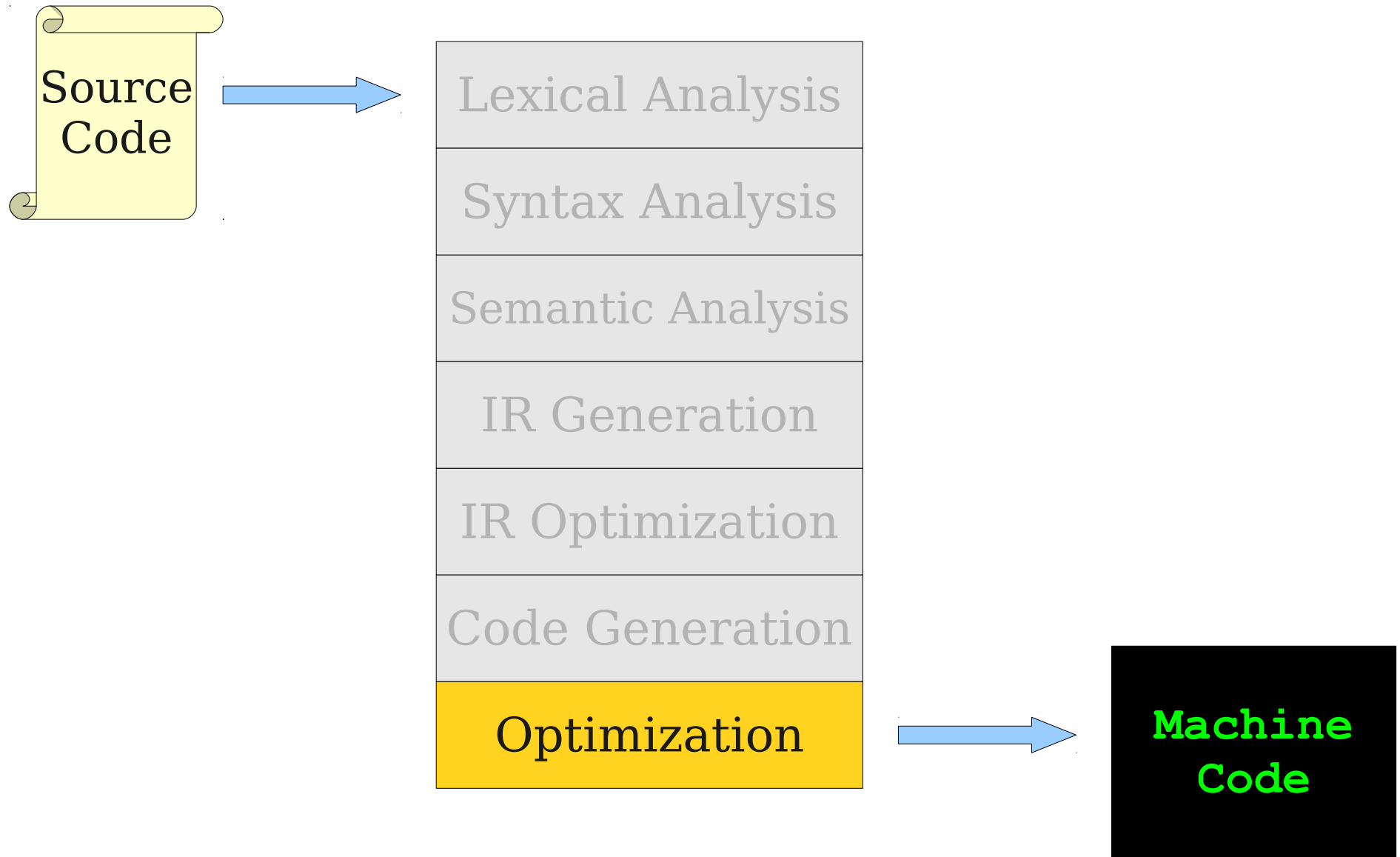
**Write-after-Write  
(WAW)**

$x = \dots$   
 $x = \dots$

# 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



# Where We Are

