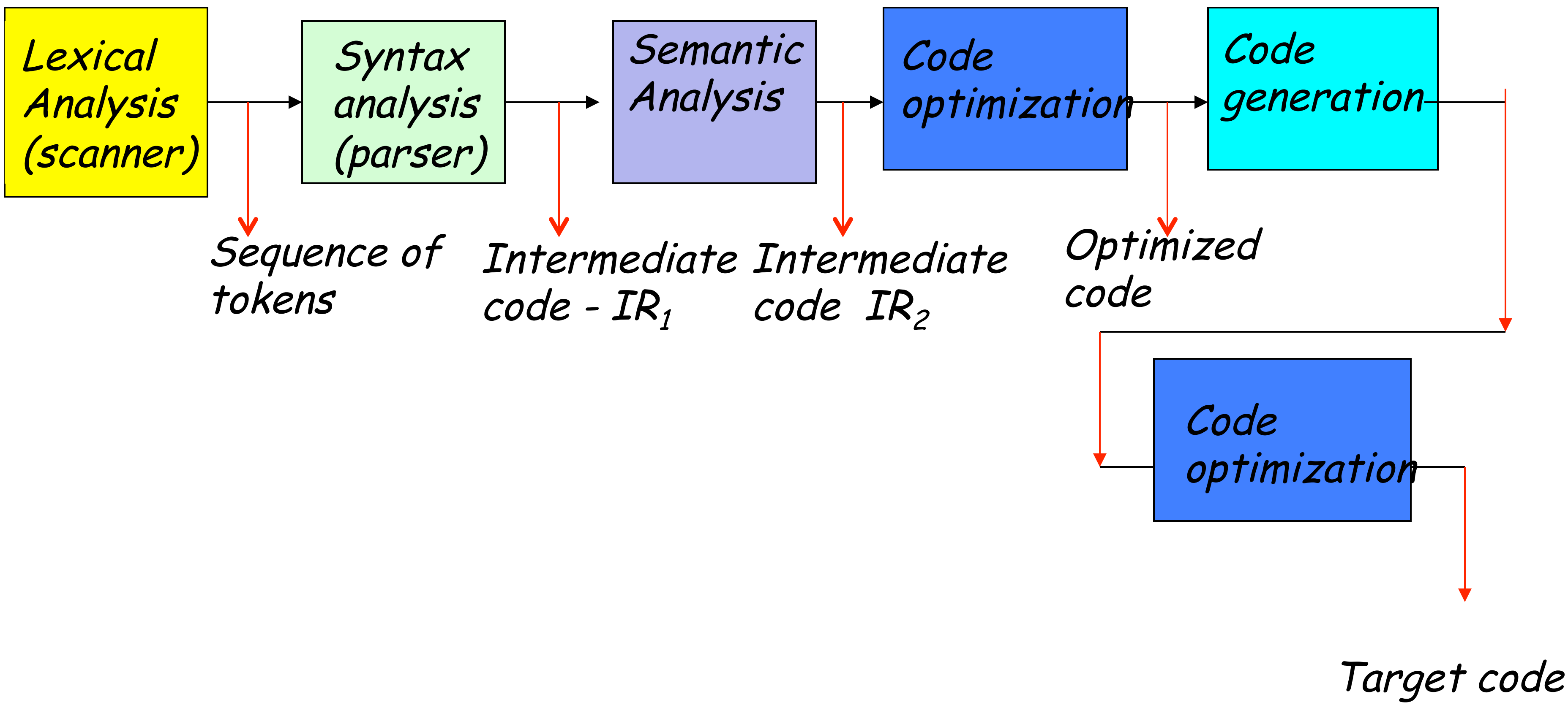
Overview of Compiler Optimization Phases



Not Really an Optimizer

- NP complete problem

- Instead, produces a “better” version:
  - memory
  - time
  - energy/power
  - network messages

- Better use of resources
- Reduce inefficiencies in generated code

3 Levels of Optimization

1. **Local**
   - Apply to a basic block in isolation
2. **Global**
   - Apply to a method/function in isolation
3. **Inter-procedural**
   - Apply across method boundaries

Most compilers do (1)

many do (2)

very few do (3)

Representing the Program for Optimization
Each Method : Control Flow Graph

The Control Flow Graph

```
BEGIN /* main routine of a nonsense program */
 x := 1;
WHILE (x = 1) DO
            x := 2;
            test ( x, 1 );
             x := 3;
OD;
WHILE (x = 1) DO
            x := 4;
            x := 5;
            test ( x, 2 );
OD;
WHILE (x = 1) DO
            x := 6;
            IF (x = 7)
               THEN x := 8;
                ELSE test ( x, 3 );
            FI;
OD;
END.
```

Node = a basic block
where Basic block =
maximal sequence of consecutive statements in which flow enters at the start and leaves at the end without halt or branching except at the end.

Edge = directed, to show the flow of control between basic blocks

Entry Node

Exit Node

CFG: Rooted, directed graph.

Construction of a CFG

A Leader = first statements of basic block

What constitutes a leader?

How can the CFG be built for a procedure?

Terminology:

- pred(b)
- succ(b)
- branch nodes
- join nodes

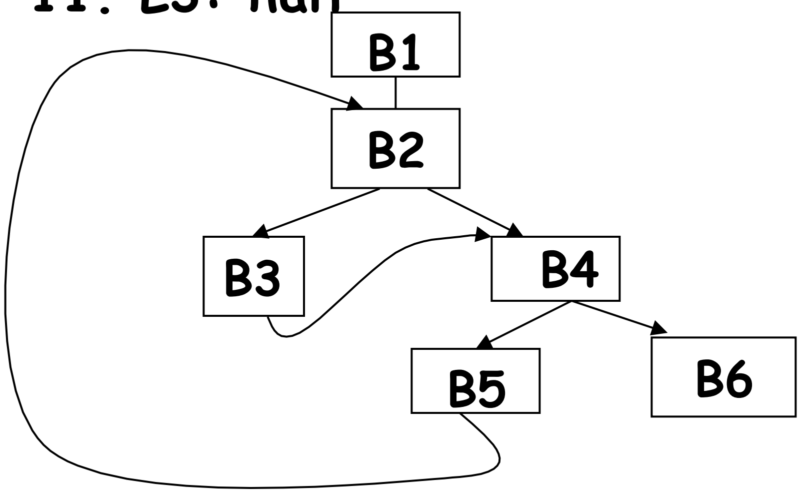
- program points
- join point
- split point

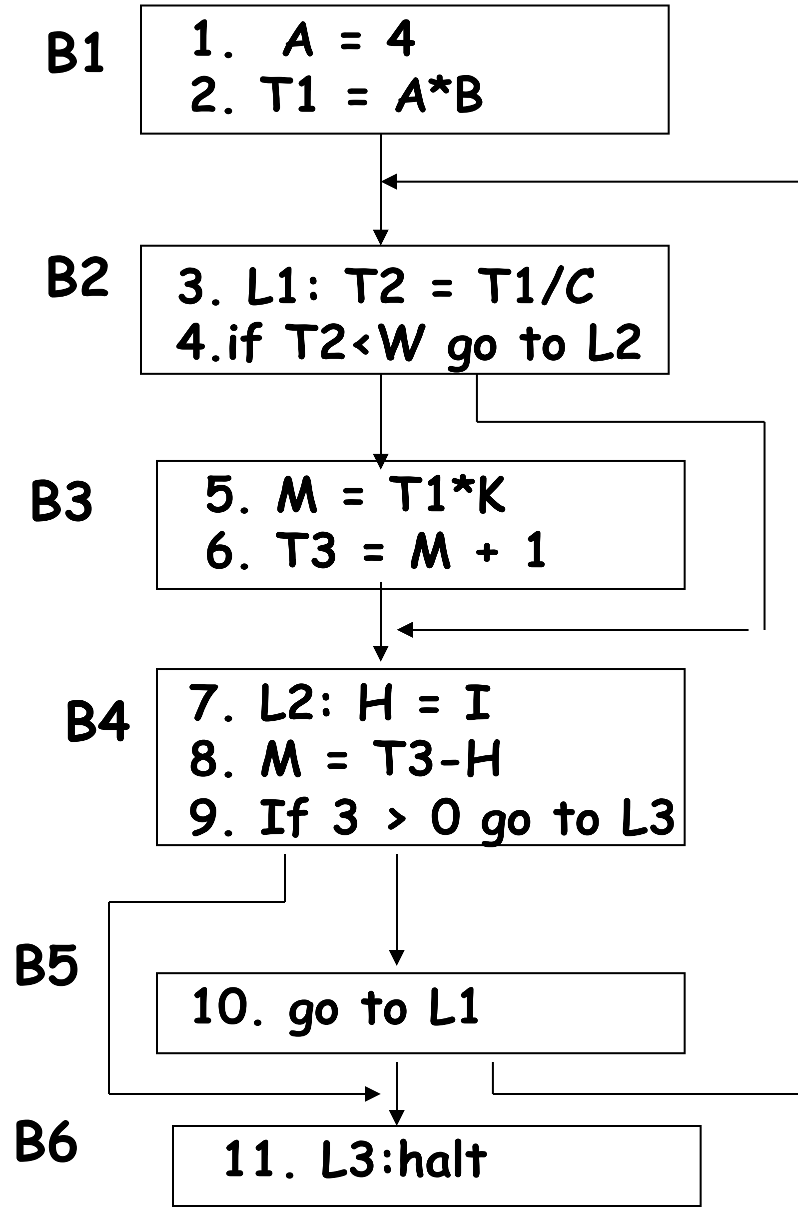
Let's Try It – Construct the CFG

```
      receive m (val)
      f0 ← 0
      f1 ← 1
      if m <= 1 goto L3
      i ← 2
L1:   if i <= m goto L2
      return f2
L2:   f2 ← f0 + f1
      f0 ← f1
      f1 ← f2
      i ← i + 1
      goto L1
L3:   return m
```

Another Example:

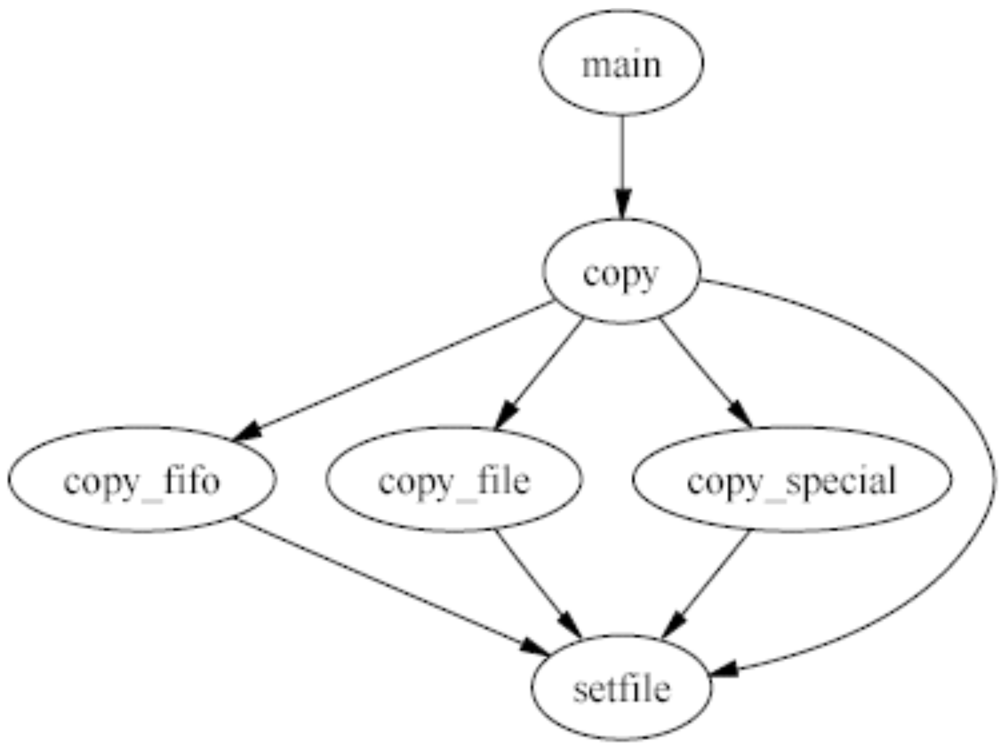
```
1.        A = 4
2.        T1 = A*B
3. L1:  T2 = T1/C
4.         If T2 < W go to L2
5.        M = T1 * K
6.        T3 = M + 1
7.  L2: H = I
8.        M = T3 - H
9.        If T3 > 0 go to L3
10.      Go to L1
11. L3: halt
```





Connect the Methods/Functions through Call Graph Representation

Node = function or method
Edge from A to B : A has a call site where B is potentially called



Let's Try It: Construct a call graph

```
procedure f( )
begin
   call g( )
   call g( )
   call h( )
end   || f
procedure g( )
begin
   call h( )
   call i( )
end   || g
procedure h( )
begin
end   || h
procedure i( )
   procedure j( )
   begin
   end   || j
begin
   call g( )
   call j( )
end   || i
```

Local Optimization

Algebraic Simplification

- Some statements can be deleted

```
x := x + 0
x := x * 1
```

- Some statements can be simplified

```
x := x * 0      ⇒   x := 0
y := y ** 2     ⇒   y := y * y
x := x * 8      ⇒   x := x << 3
```

(Use fastest operation:

e.g., On some machines `<<` is faster than `*`; but not on all!)

Copy Propagation

- If w := x appears in a block, all subsequent uses of w can be replaced with uses of x

- Example:

b := z + y		b := z + y
a := b	$\Rightarrow$	a := b
x := 2 * a		x := 2 * b

- This does not make the program smaller or faster but might enable other optimizations, e.g.,
  - Constant folding
  - Dead code elimination

Constant Folding

- Operations on constants can be computed at compile time

- In general, if there is a statement

  x = y op z

  - And y and z are constants
  - Then y op z can be computed at compile time

- Example: x = 2 + 2 $\Rightarrow$ x = 4

- Example: if 2 < 0 jump L can be deleted

Combining Copy Propagation and Constant Folding

- Example:

a := 5		a := 5
x := 2 * a	⇒	x := 10
y := x + 6		y := 16
t := x * y		t := x << 4

Common Subexpression Elimination

- **Assume**
  - Basic block is in single assignment form

  (Contiguous instructions with no jumps in or out; no more than 1 assignment per variable)
  - A definition x := is the first use of x in a block
- If any assignments have the same rhs, they compute the same value

- **Example:**

x := y + z		x := y + z
…	⇒	…
w := y + z		w := x

(the values of x, y, and z do not change in the code)

Dead Code Elimination

If w := rhs appears in a basic block

- And w does not appear anywhere else in the block (not live in block or rest of program)

Then

- the statement w := rhs is dead and can be eliminated
  - Dead = does not contribute to the program's result

Example: (a is not used anywhere else)

b := z + y		b := z + y		b := x + y
a := b	⇒	a := b	⇒	x := 2 * b
x := 2 * a		x := 2 * b		

Applying Local Optimizations

- Each local optimization does very little by itself
- Typically optimizations interact
  - Performing one optimization enables other opt.
- Typical optimizing compilers repeatedly perform optimizations until no more improvement
  - The optimizer can also be stopped at any time to limit the compilation time

Compiler Optimization Challenge

- **Given the following code segment in a basic block, optimize the code using algebraic simplification, copy propagation, constant folding, common subexpression elimination and dead code elimination.**
- **The goal is to produce the least number of instructions that will execute faster. Show each step of the optimization.**

1. a := x ** 2
2. b := 3
3. c := x
4. d := c * c
5. e := b * 2
6. f := a + d
7. g := e * f
8. Print (g)