

# **Methodologies for Software Processes**

## **Lecture 1**

# Course website

**<http://www.cs.ubbcluj.ro/~craciunf/MethodologiesSoftwareProcesses>**

# Course Overview

1. Program Optimizations based on Dataflow analyses (about 5 lectures + 1 programming project)
2. Program Verification: Hoare Logic and Separation Logic (about 8 lectures + 1 project)

# Grading

Seminar activity (2 assignments) :--**50%**

1. DataFlow Programming Project: -- **25%**
2. Program Verification Assignment: --**15%**
3. Program Verification Paper Presentation: - **10%**

Final exam: -- **50%**

- Final Written Exam (about 2 hours, open books): --**50%**

# Rules

- seminar activity will be done at the group level
- groups are fixed at the first seminar and consist of max 2 students
- final exam is individual and is an open book exam (you can have access at the lecture notes and the seminar notes)

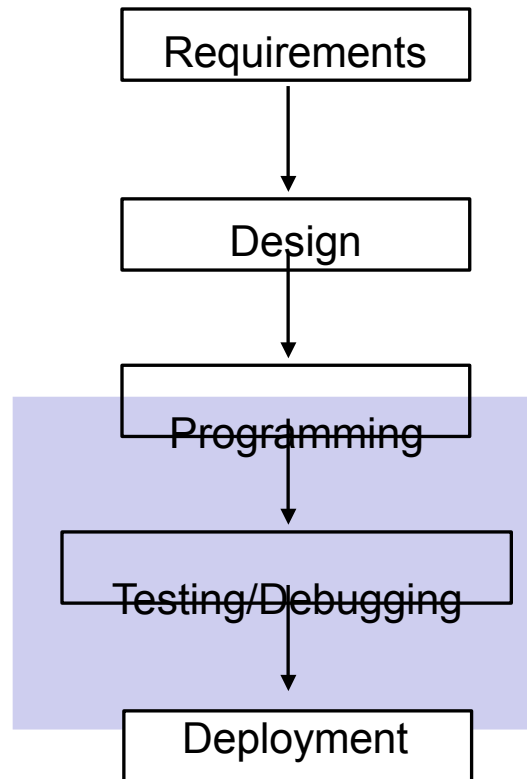
# References

- F.Nielson, H.R.Nielson, C. Hankin:  
Principles of Program Analysis, Springer  
Verlag, 2004
- Other research papers that will be made  
available with the lecture notes

# Static Program Analysis

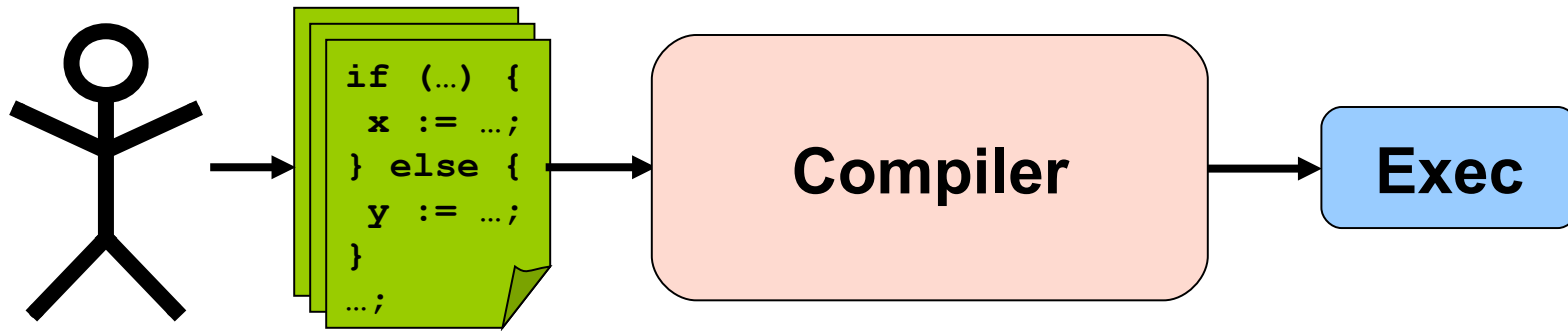
- **Testing:** manually checking a property for some execution paths
- **Model checking:** automatically checking a property for all execution paths
- **Static analysis** consists of automatically discovering properties of a program that hold for all possible execution paths of the program

# What is this course about?

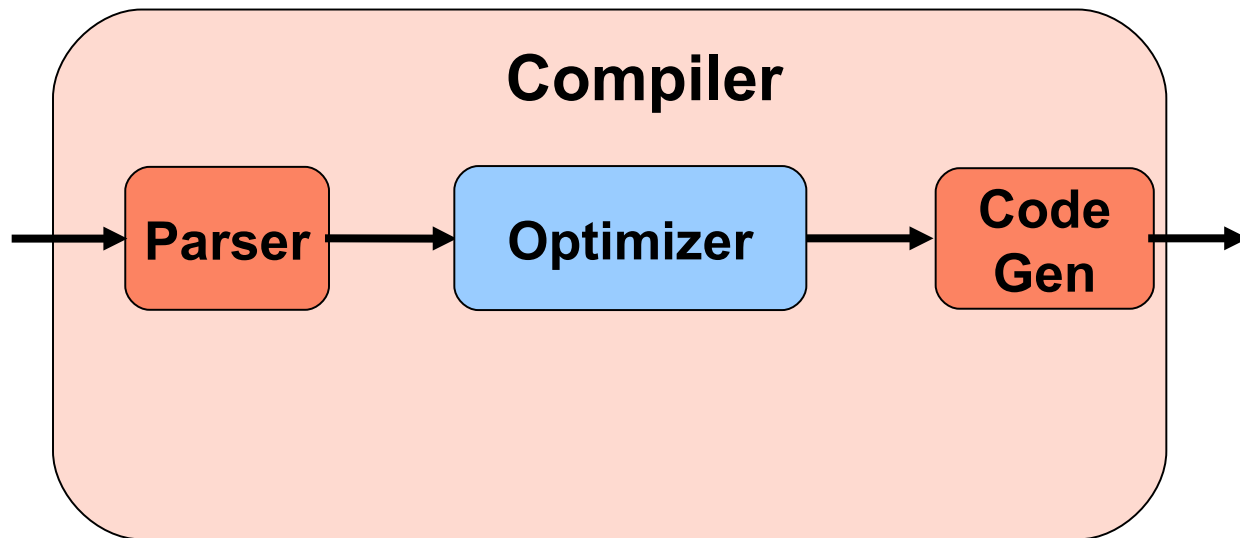




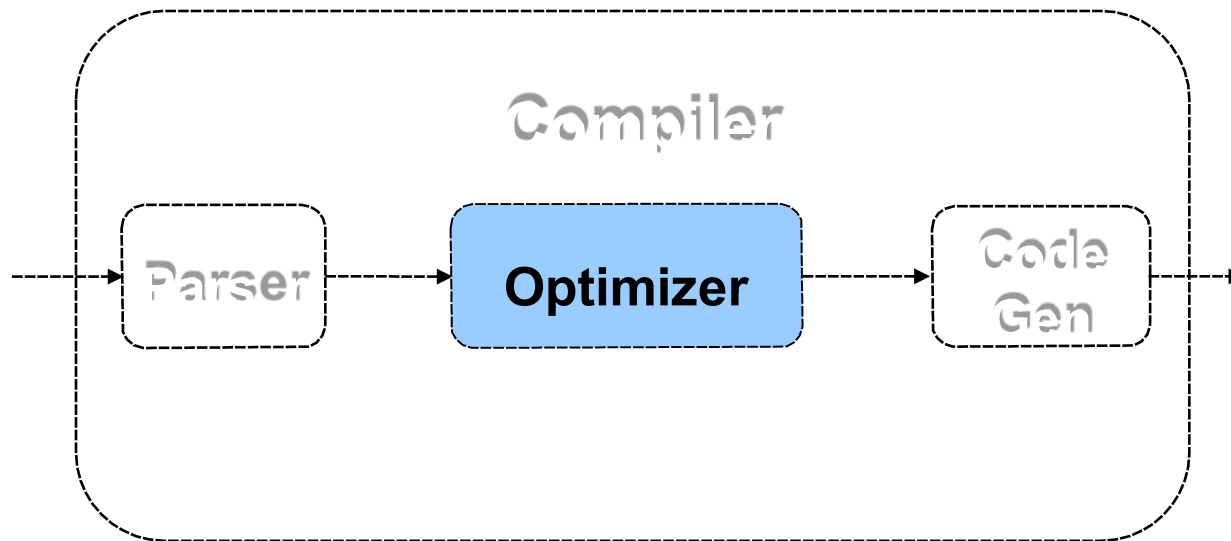
# Let's look at a compiler



# Let's look at a compiler

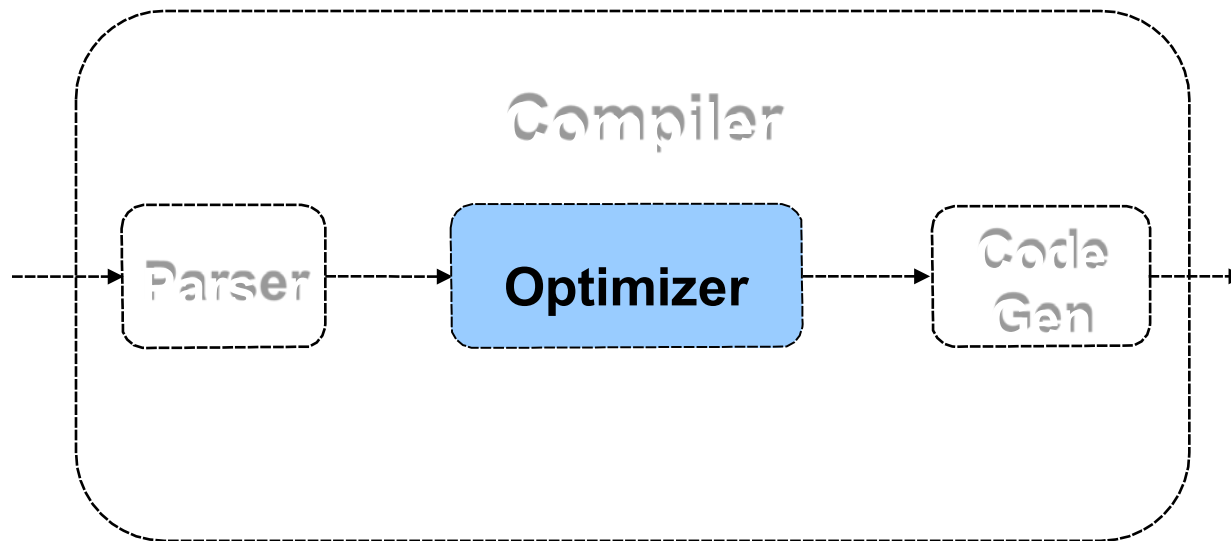


# What does an optimizer do?



1. Analysis
2. Transformations

# What does an optimizer do?



1. Compute information about a program
2. Use that information to perform  
program transformations  
(with the goal of improving some metric, e.g. performance)

# Example: Escape Analysis

- Consider these two C procedures:

```
typedef int A[10000];
```

```
typedef A* B;
```

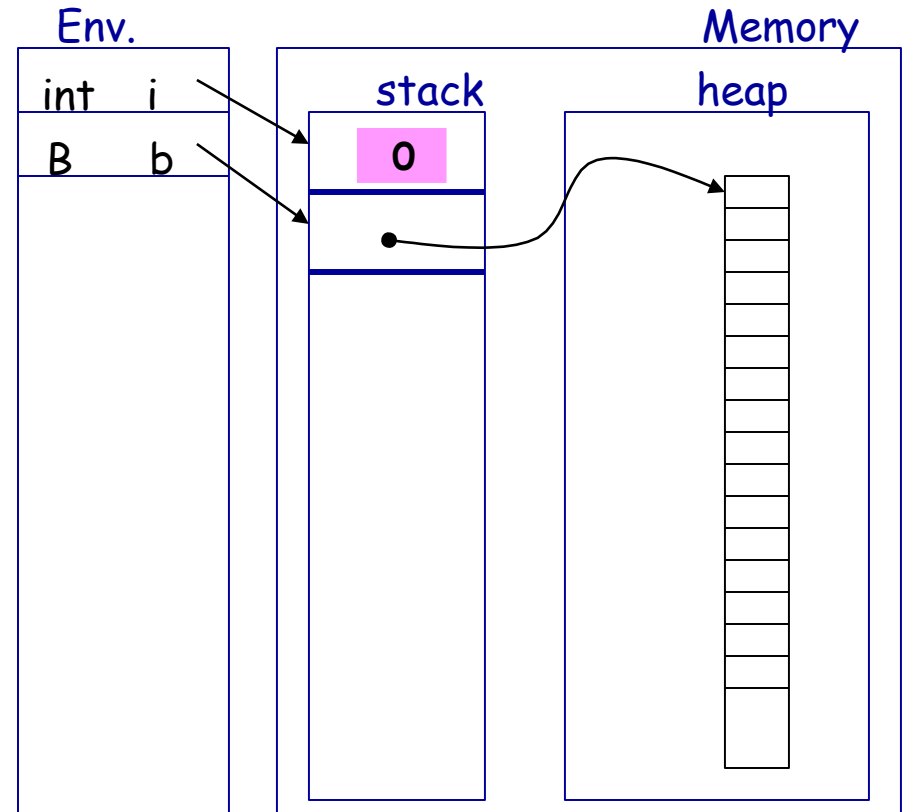
```
void execute1() {  
    int i;  
    B b;  
    b = (A*) malloc(sizeof(A));  
    for (i=0; i<10000; i++)  
        (*b)[i]=0;  
    free(b);  
}
```

```
void execute2() {  
    int i;  
    A a;  
    for (i=0; i<10000; i++)  
        a[i]=0;  
}
```

```

typedef int A[10000];
typedef A* B;
void execute1(){
    int i;
    B b;
    b = (A*) malloc(sizeof(A));
    for (i=0; i<10000; i++)
        (*b)[i]=0;
    free(b);
}

```



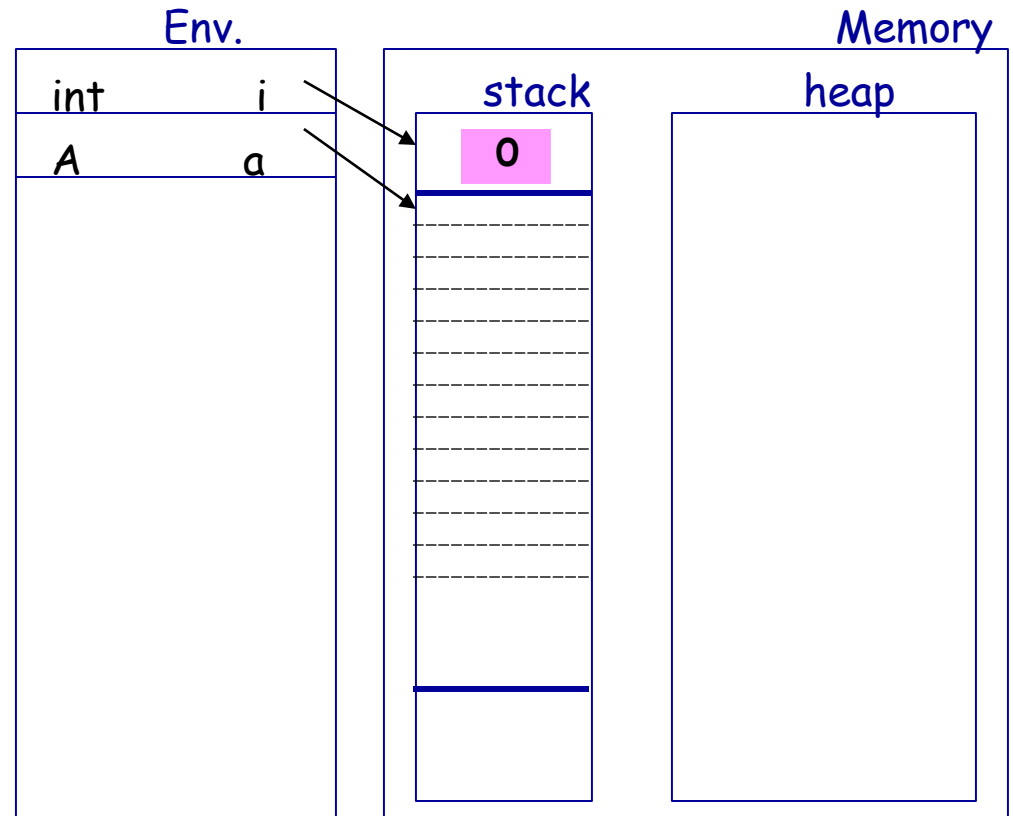
- The array `b` is allocated in the heap, and then de-allocated

```

typedef int A[10000];

void execute2() {
    int i;
    A a;
    for (i=0; i<10000; i++)
        a[i]=0;
}

```



- The array `a` is in the stack

# Memory allocation has a cost...

```
for (i=0; i<1.000.000; i++)  
    execute1();                // about 90 sec
```

```
for (i=0; i<1.000.000; i++)  
    execute2();                // about 35 sec
```

- The second program is three times faster than the first one
- If we can predict that a dynamic variable does not escape out of the procedure in which it is allocated, we can use a static variable instead, getting a more efficient program



# Program Verification

- Check that every operation of a program will never cause an error (division by zero, buffer overrun, deadlock, etc.)
- Example:

```
int a[1000];  
for (i = 0; i < 1000; i++) {  
    safe operation → a[i] = ... ;    // 0 ≤ i ≤ 999  
}  
buffer overrun → a[i] = ... ;    // i = 1000;
```

# Why Is Software Verification Important?

- One of the most prominent challenges for IT.
  - Software bugs cost the U.S. economy about \$59.5 billion each year (0.6% of the GDP) [NIST 02].
- Security is becoming a necessity.
  - The worldwide economic loss caused by all forms of overt attacks is \$226 billion.
- Software defects make programming so painful.
- Stories
  - The Role of Software in Spacecraft Accidents(<http://sunnyday.mit.edu/papers/jsr.pdf>)
  - History's Worst Software Bugs(<http://www.wired.com/software/coolapps/news/2005/11/69355>)

# Software bugs may cause big troubles...



On 4 June 1996, the maiden flight of the Ariane 5 launcher ended in a failure. Only about 40 seconds after initiation of the flight sequence, at an altitude of about 3700 m, the launcher veered off its flight path, broke up and exploded. »

The failure of the Ariane 5.01 was caused by the complete loss of guidance and attitude information 37 seconds after start of the main engine ignition sequence, 30 seconds after lift-off.

This loss of information was due to specification and design errors in the software of the inertial reference system.

# Software bugs may cause big troubles...



A conversion from 64-bit floating point to 16 bit integer with a value larger than possible. The overflow caused a hardware trap



# Remarks by Bill Gates

17th Annual ACM Conference on Object-Oriented  
Programming, Seattle, Washington, November 8, 2002

“... When you look at a big commercial software company like Microsoft, there's actually as much testing that goes in as development.

We have as many testers as we have developers. Testers basically test all the time, and developers basically are involved in the testing process about half the time...



# Remarks by Bill Gates

17th Annual ACM Conference on Object-Oriented  
Programming, Seattle, Washington, November 8, 2002

“... We've probably changed the industry we're in. We're not in the software industry; we're in the testing industry, and writing the software is the thing that keeps us busy doing all that testing.”

“...The test cases are unbelievably expensive; in fact, there's more lines of code in the test harness than there is in the program itself. Often that's a ratio of about three to one.”

# The goal – reliable software systems

- Quality dilemma: quality / features / time
- More efficient methods for test and verification are always needed
  - No ‘silver-bullet’ when it comes to testing.
  - **Formal verification** is on the rise...

# Testing / Formal Verification

A very crude dichotomy:

Testing
Correct with respect to the <i>set of test inputs</i> , and reference system
Easy to perform
Dynamic

In practice:

Many types of testing,



# Testing / Formal Verification

A very crude dichotomy:

Testing	Formal Verification
Correct with respect to the <i>set of test inputs</i> , and reference system	Correct with respect to <i>all inputs</i> , with respect to a <i>formal specification</i>
Easy to perform	Decidability problems, Computational problems,
Dynamic	Static

In practice:

Many types of testing,  
Many types of formal verification.

# Our approach

**Formal** = based on rigorous **mathematical logic** concepts.

**Semantics-based** = based on a formal specification of the “**meaning**” of the program

# Fundamental Limit: Undecidability

## Rice Theorem

Any non-trivial semantic property of programs is undecidable.

## Classical Example: Termination

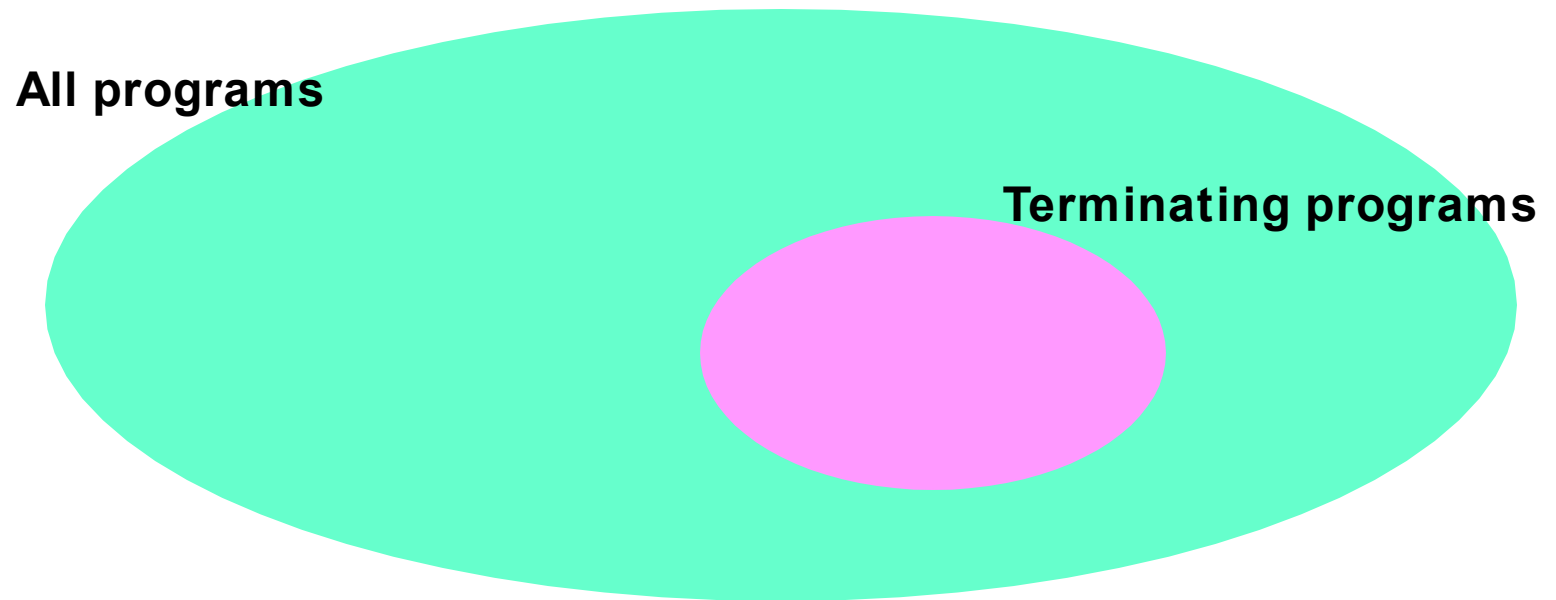
There exists no algorithm which can solve the halting problem: given a description of a program as input, decide whether the program terminates or loops forever.

# Incompleteness of Program Analysis

- Discovering a sufficient set of properties for checking every operation of a program is an undecidable problem!
- **False positives:**  
operations that are safe in reality but which cannot be decided safe or unsafe from the properties inferred by static analysis.

# Example

- We all know that the halting problem is not decidable



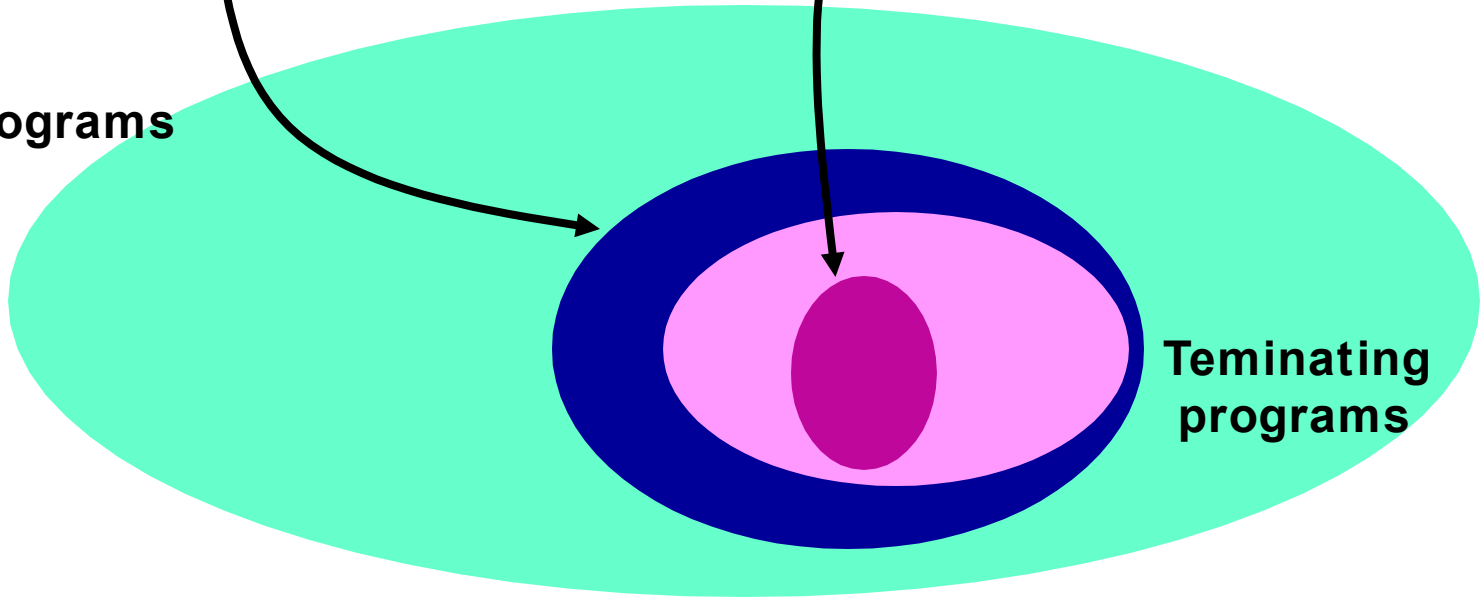
# Under- and Over-approximation

Programs that *may* terminate

Program that *surely* terminate

All programs

Terminating programs



# Soundness and Completeness

- A **sound** static analysis **overapproximates** the behaviors of the program.

A sound static analyzer is guaranteed to identify all violations of our property, but may also report some “false alarms”, or violations of the property that cannot actually occur.

- A **complete** static analysis **underapproximates** the behaviors of the program.

Any violation of our property reported by a complete static analyzer corresponds to an actual violation of the property, but there is no guarantee that all actual violations of will be reported

- Note that when a sound static analyzer reports no errors, our program is guaranteed not to violate the property. This is a powerful guarantee. As a result, most static analysis tools choose to be sound rather than complete.

# Precision versus Efficiency

Precision: number of program operations that can be decided safe or unsafe by the analyzer.

- Precision and computational complexity are strongly related
- Tradeoff precision/efficiency: limit in the average precision and scalability of a given analyzer
- Greater precision and scalability is achieved through **specialization**



# Specialization

- Tailoring the program analyzer algorithms for a specific class of programs (flight control commands, digital signal processing, etc.)
- Precision and scalability is guaranteed for this class of programs only
- Requires a lot of try-and-test to fine-tune the algorithms

# What do these tools have in common?

- Bug finders
- Program verifiers
- Code refactoring tools
- Garbage collectors
- Runtime monitoring system
- And... optimizers

■ They all analyze and transform programs

We will learn about the techniques underlying all these tools

# Course goals

- Understand basic techniques for doing program analyses and verification
  - these techniques are the cornerstone of a variety of program analysis tools
- Get a feeling for what research is like in the area by reading research papers, and getting your feet wet in small implementation projects

# Course topics

- Techniques for representing programs
- 
- 
- Techniques for analyzing and verifying programs
- 
- 
- Applications of these techniques

# Overview of Optimizations

# Simple example

```
foo(z) {  
    x := 3 + 6;  
    y := x - 5  
    return z * y  
}
```

# Simple example

```
foo(z) {
```

```
  x := 3 + 6;
```



```
  x := 9;
```

Applying Constant Folding

```
  y := x - 5;
```

```
  return z * y
```

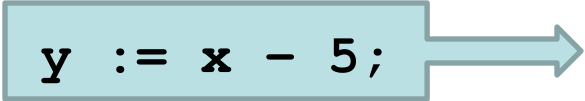
```
}
```

# Simple example

```
foo(z) {
```

```
    x := 9;
```

```
    y := x - 5;
```

 `y := 9 - 5;` By Constant Propagation

```
    return z * y
```

```
}
```



# Simple example

```
foo(z) {
```

```
    x := 9;
```

```
    y := 9 - 5;
```



```
    y := 4;
```

Applying Constant Folding

```
    return z * y
```

```
}
```

# Simple example

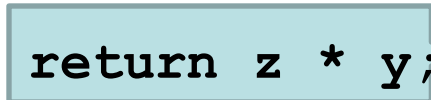
```
foo(z) {
```

```
    x := 9;
```

```
    y := 4;
```

```
    return z * y;
```

```
}
```

  `return z*4;` By Constant Propagation

# Simple example

```
foo(z) {
```

```
    x := 9;
```

```
    y := 4;
```

```
    return z*4;
```



```
    return z << 2;    By Strenght Reduction
```

```
}
```

# Simple example

```
foo(z) {
```

```
x := 9;
```

```
y := 4;
```

```
    return z << 2;
```

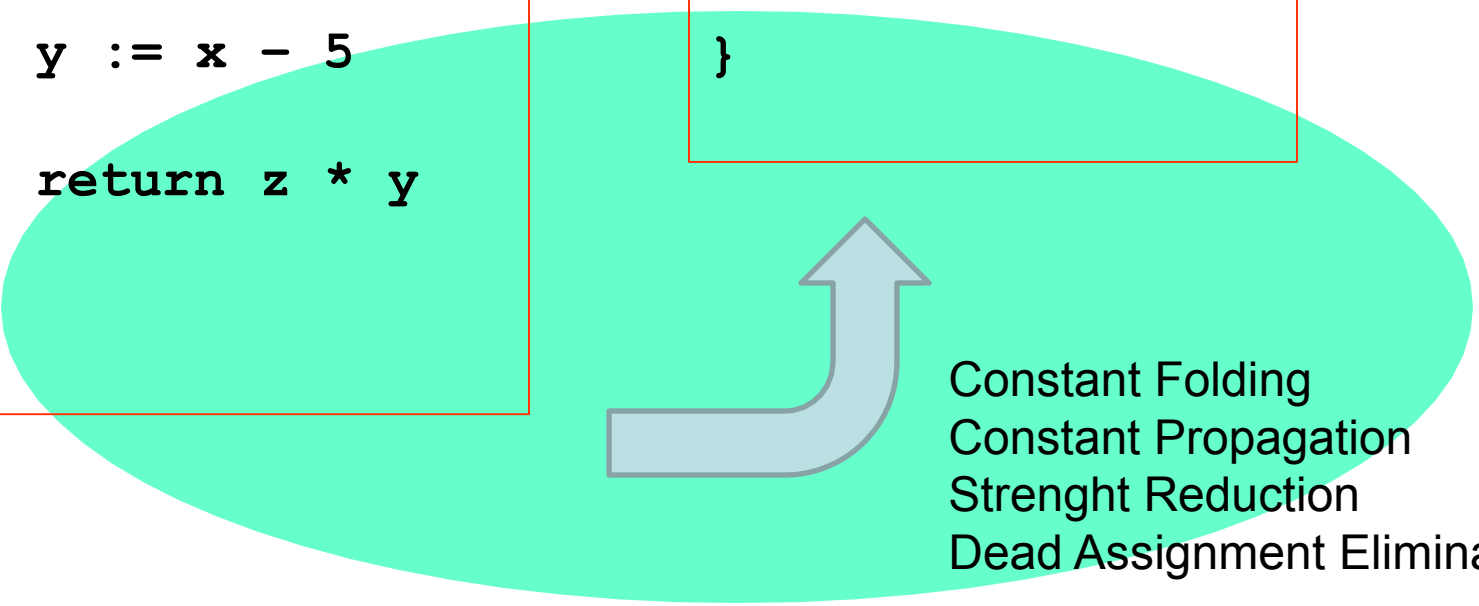
```
}
```

By Dead Assignment Elimination

# Simple example

```
foo(z) {  
  
    x := 3 + 6;  
  
    y := x - 5  
  
    return z * y  
  
}
```

```
foo(z) {  
  
    return z << 2;  
  
}
```



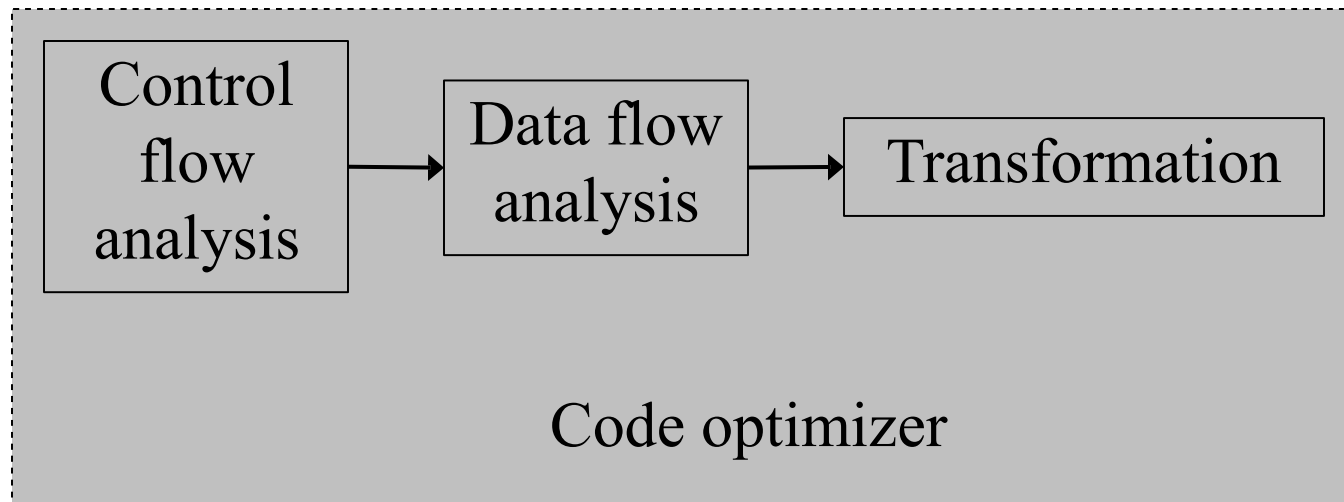
Constant Folding  
Constant Propagation  
Streight Reduction  
Dead Assignment Elimination

# Peephole optimizations

- Concerns with machine-independent code optimization
  - 90-10 rule: execution spends 90% time in 10% of the code.
  - Identification of the 10% of the code is not possible for a compiler – it is the job of a **profiler**.
- In general, loops are the hot-spots

- Criterion of code optimization
  - Must preserve the semantic equivalence of the programs
  - The algorithms should not be modified
    - Transformation, on average should speed up the execution of the program
    - Transformations should be simple enough to have a good effect

- Organization of an optimizing compiler





# Themes behind Optimization

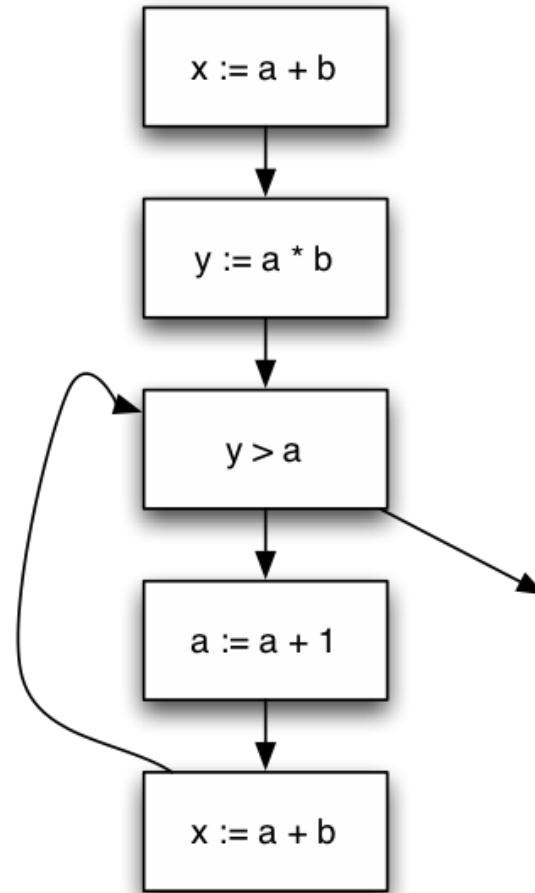
- Avoid redundancy: something already computed need not be computed again
- Smaller code: less work for CPU, cache, and memory!
- Less jumps: jumps interfere with code pre-fetch
- Code locality: codes executed close together in time is generated close together in memory – increase locality of reference
  - Extract more information about code: More info – better code generation

# Control-Flow Graph (CFG)

- A directed graph where
  - Each node represents a statement
  - Edges represent control flow
- Three address code: Statements may be
  - Assignments  $x = y \text{ op } z$  or  $x = \text{op } z$
  - Copy statements  $x = y$
  - Branches  $\text{goto } L$  or  $\text{if relop } y \text{ goto } L$
  - etc

# Control-flow Graph Example

```
x := a + b;  
y := a * b  
while (y > a)  
{  
  a := a + 1;  
  x := a + b  
}
```



# Variations on CFGs

- Usually don't include declarations (e.g. `int x;`).
- May want a unique entry and exit point.
- May group statements into *basic blocks*.
  - A *basic block* is a sequence of instructions with no branches into or out of the block.

# Basic Blocks

A **basic block** is a sequence of consecutive intermediate language statements in which flow of control can only enter at the beginning and leave at the end.

Only the last statement of a basic block can be a branch statement and only the first statement of a basic block can be a target of a branch.

In some frameworks, procedure calls may occur within a basic block.

# Basic Block Partitioning Algorithm

1. Identify leader statements (i.e. the first statements of basic blocks) by using the following rules:
  - (i) The **first statement** in the program is a leader
  - (ii) Any statement that is the **target of a branch** statement is a leader (for most intermediate languages these are statements with an associated label)
  - (iii) Any statement that **immediately follows a branch** or **return statement** is a leader

# Example: Finding Leaders

The following code computes the inner product of two vectors.

```
begin
  prod := 0;
  i := 1;
  do begin
    prod := prod + a[i] * b[i];
    i = i+ 1;
  end
  while i <= 20
end
```

**Source code**

```
(1)  prod := 0
(2)  i := 1
(3)  t1 := 4 * i
(4)  t2 := a[t1]
(5)  t3 := 4 * i
(6)  t4 := b[t3]
(7)  t5 := t2 * t4
(8)  t6 := prod + t5
(9)  prod := t6
(10) t7 := i + 1
(11) i := t7
(12) if i <= 20 goto (3)
```

**Three-address code**

# Example: Finding Leaders

The following code computes the inner product of two vectors.

```
begin
    prod := 0;
    i := 1;
    do begin
        prod := prod + a[i] * b[i];
        i = i + 1;
    end
    while i <= 20
end
```

**Source code**

**Rule (i)**

```
(1)  prod := 0
(2)  i := 1
(3)  t1 := 4 * i
(4)      t2 :=
a[t1] (5)  t3 :=
4 * i (6)  t4 :=
b[t3]
(7)  t5 := t2 * t4
(8)  t6 := prod + t5
(9)  prod := t6
(10) t7 := i + 1
(11) i := t7
(12) if i <= 20 goto
(3)
(13) ...
```

**Three-address code** <sup>20</sup>



# Example: Finding Leaders

The following code computes the inner product of two vectors.

```
begin
  prod := 0;
  i := 1;
  do begin
    prod := prod + a[i] * b[i];
    i = i+ 1;
  end
  while i <= 20
end
```

**Source code**

**Rule (i)**

(1) prod := 0

(2) i := 1

**Rule (ii)**

(3) t1 := 4 \* i

(4) t2 := a[t1]

(5) t3 := 4 \* i

(6) t4 := b[t3]

(7) t5 := t2 \* t4

(8) t6 := prod + t5

(9) prod := t6

(10) t7 := i + 1

(11) i := t7

(12) if i <= 20 goto (3)

(13) ...

**Three-address code**

# Example: Finding Leaders

The following code computes the inner product of two vectors.

```
begin
  prod := 0;
  i := 1;
  do begin
    prod := prod + a[i] * b[i];
    i = i+ 1;
  end
  while i <= 20
end
```

Source code

Rule (i)

(1) prod := 0

(2) i := 1

Rule (ii)

(3) t1 := 4 \* i

(4) t2 :=

a[t1] (5) t3 :=

4 \* i (6) t4 :=

b[t3]

(7) t5 := t2 \* t4

(8) t6 := prod + t5

(9) prod := t6

(10) t7 := i + 1

(11) i := t7

(12) if i <= 20 goto (3)

Rule (iii)

(13) ...

Three-address code

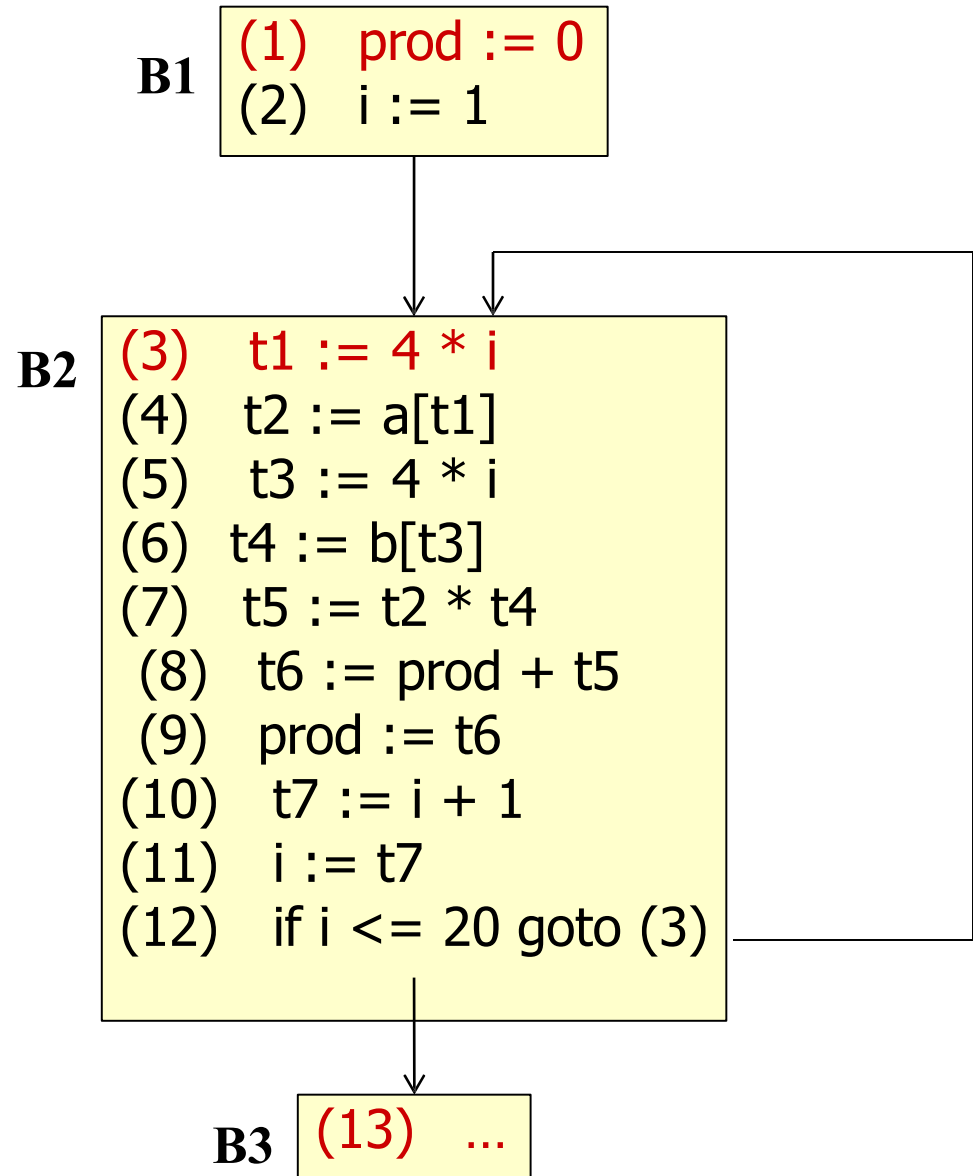
# Forming the Basic Blocks

Now that we know the leaders, how do we form the basic blocks associated with each leader?

2. The basic block corresponding to a leader consists of the leader, plus all statements up to **but not including** the next leader or up to the end of the program.

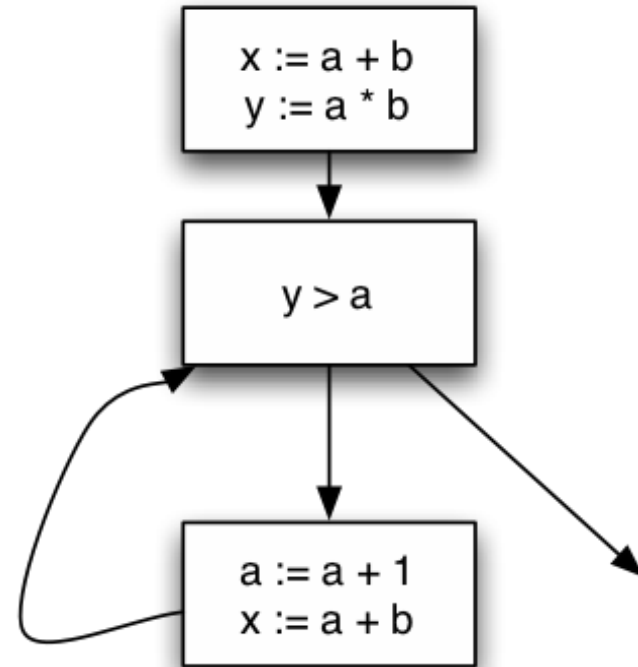
# Example: Forming the Basic Blocks

Control flow  
diagram



# Control-Flow Graph with Basic Blocks

```
x := a + b;  
y := a * b  
while (y > a)  
{  
  a := a + 1;  
  x := a + b  
}
```



Can lead to more efficient implementations

# Classical Optimization

- Types of classical optimizations
  - *Operation level*: one operation in isolation
  - *Local*: optimize pairs of operations in same basic block (with or without dataflow analysis)
  - *Global*: optimize pairs of operations spanning multiple basic blocks and must use dataflow analysis in this case, e.g. *reaching definitions*, *UD/DU chains*, or *SSA forms*
  - *Loop*: optimize loop body and nested loops

# Redundancy elimination

- **Redundancy elimination** = determining that two computations are equivalent and eliminating one.
- There are several types of redundancy elimination:
  - **Common subexpression elimination**
    - Identifies expressions that have operands with the same name
  - **Constant Folding and Constant/Copy propagation**
    - Identifies variables that have constant/copy values and uses the constants/copies in place of the variables.

# Compile-Time Evaluation

- **Constant folding:** Evaluation of an expression with constant operands to replace the expression with single value
- Example:

```
area := (22.0/7.0) * r ** 2
```



```
area := 3.14286 * r ** 2
```



# Compile-Time Evaluation

- **Constant Folding:** Replace a variable with constant which has been assigned to it earlier.

- Example:

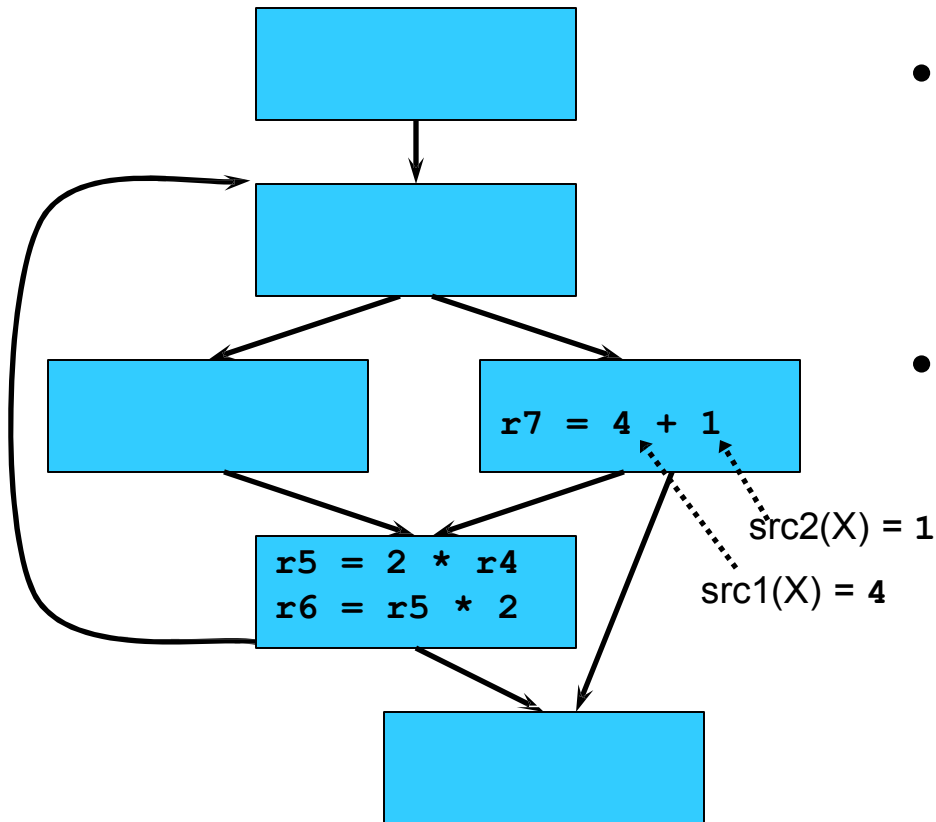
```
pi := 3.14286
```

```
area = pi * r ** 2
```



```
area = 3.14286 * r ** 2
```

# Local Constant Folding



- Goal: eliminate unnecessary operations
- Rules:
  1.  $X$  is an arithmetic operation
  2. If  $\text{src1}(X)$  and  $\text{src2}(X)$  are constant, then change  $X$  by applying the operation

# Constant Propagation

- What does it mean?
  - Given an assignment  $x = c$ , where  $c$  is a constant, replace later uses of  $x$  with uses of  $c$ , provided there are no intervening assignments to  $x$ .
    - Similar to copy propagation
    - Extra feature: It can analyze constant-value conditionals to determine whether a branch should be executed or not.
- When is it performed?
  - Early in the optimization process.
- What is the result?
  - Fewer registers
  - Smaller code

# Common Sub-expression Evaluation

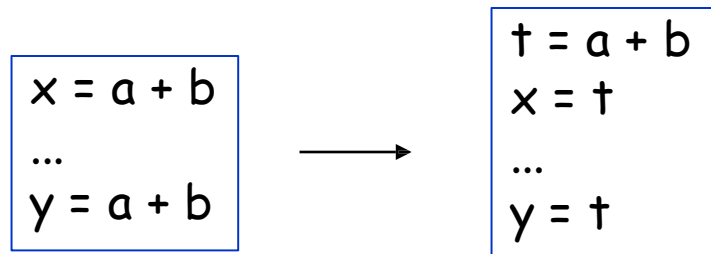
- Identify common sub-expression present in different expression, compute once, and use the result in all the places.
  - The *definition* of the variables involved should not change

Example:

a := b * c		temp := b * c
...		a := temp
...	→	...
x := b * c + 5		x := temp + 5

# Common Subexpression Elimination

- Local common subexpression elimination
  - Performed within basic blocks
  - Algorithm sketch:
    - Traverse BB from top to bottom
    - Maintain table of expressions evaluated so far
      - if any operand of the expression is redefined, remove it from the table
    - Modify applicable instructions as you go
      - generate temporary variable, store the expression in it and use the variable next time the expression is encountered.



# Common Subexpression Elimination

## Example

```
r2:= r1 * 5  
r2:= r2 + r3  
r3:= r1 * 5
```

may be transformed in

```
r4:= r1*5  
r2:= r4 + r3  
r3:= r4
```

# Common Subexpression Elimination

```
c = a + b
d = m * n
e = b + d
f = a + b
g = - b
h = b + a
a = j + a
k = m * n
j = b + d
a = - b
if m * n go to L
```



```
t1 = a + b
c = t1
t2 = m * n
d = t2
t3 = b + d
e = t3
f = t1
g = -b
h = t1 /* commutative */
a = j + a
k = t2
j = t3
a = -b
if t2 go to L
```

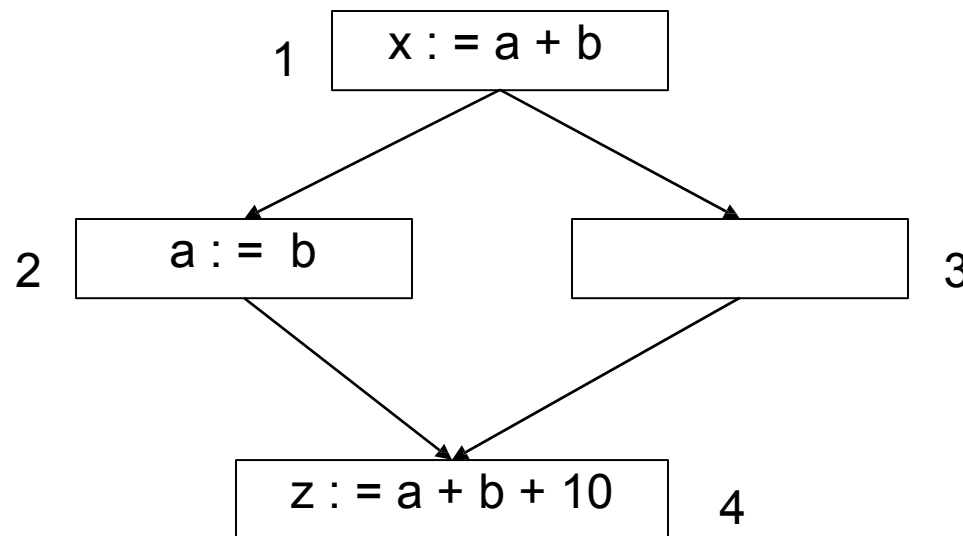
the table contains quintuples:  
(pos, opd1, opr, opd2, tmp)

# Common Subexpression Elimination

- Global common subexpression elimination
  - Performed on flow graph
  - Requires **available expression** information
    - In addition to finding what expressions are available at the endpoints of basic blocks, we need to know where each of those expressions was most recently evaluated (which block and which position within that block).



# Common Sub-expression Evaluation



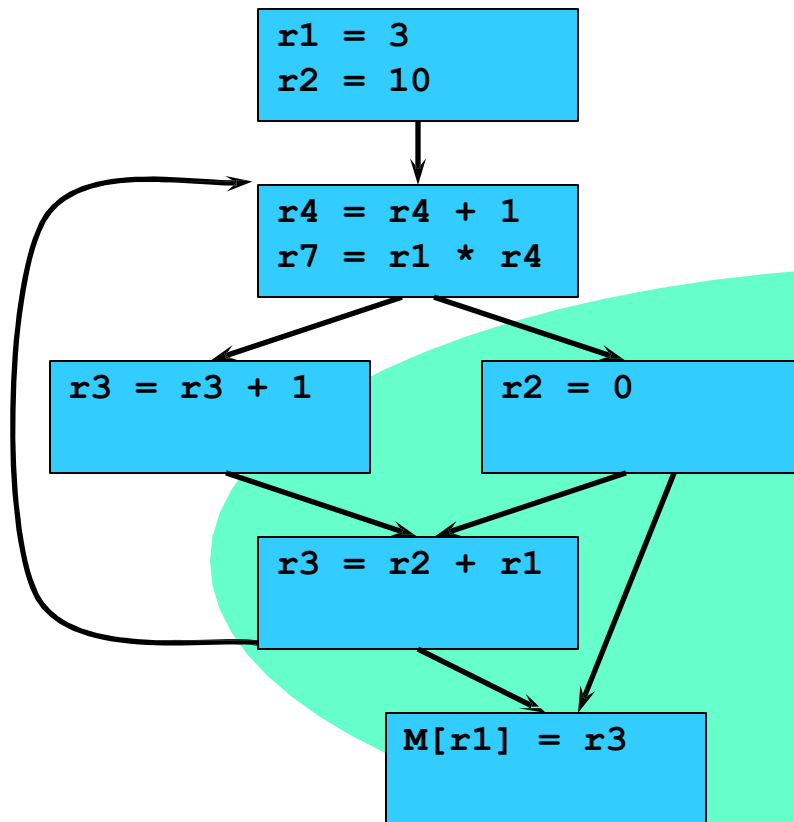
“a + b” is not a  
common sub-  
expression in 1 and 4

None of the variable involved should be modified in any  
path

# Dead Code Elimination

- Dead Code are portion of the program which will not be executed in any path of the program.
  - Can be removed
- Examples:
  - No control flows into a basic block
    - A variable is dead at a point -> its value is not used anywhere in the program
    - An assignment is dead -> assignment assigns a value to a dead variable

# Dead Code Elimination



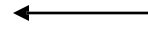
- Goal: eliminate any operation whose result is never used
- Rules (dataflow required)
  1. X is an operation with no use in DU chain, i.e. `dest(X)` is not live
  2. Delete X if removable (not a mem store or branch)
- Rules too simple!
  - Misses deletion of `r4`, even after deleting `r7`, since `r4` is live in loop
  - Better is to trace UD chains backwards from “critical” operations

# Dead Code Elimination

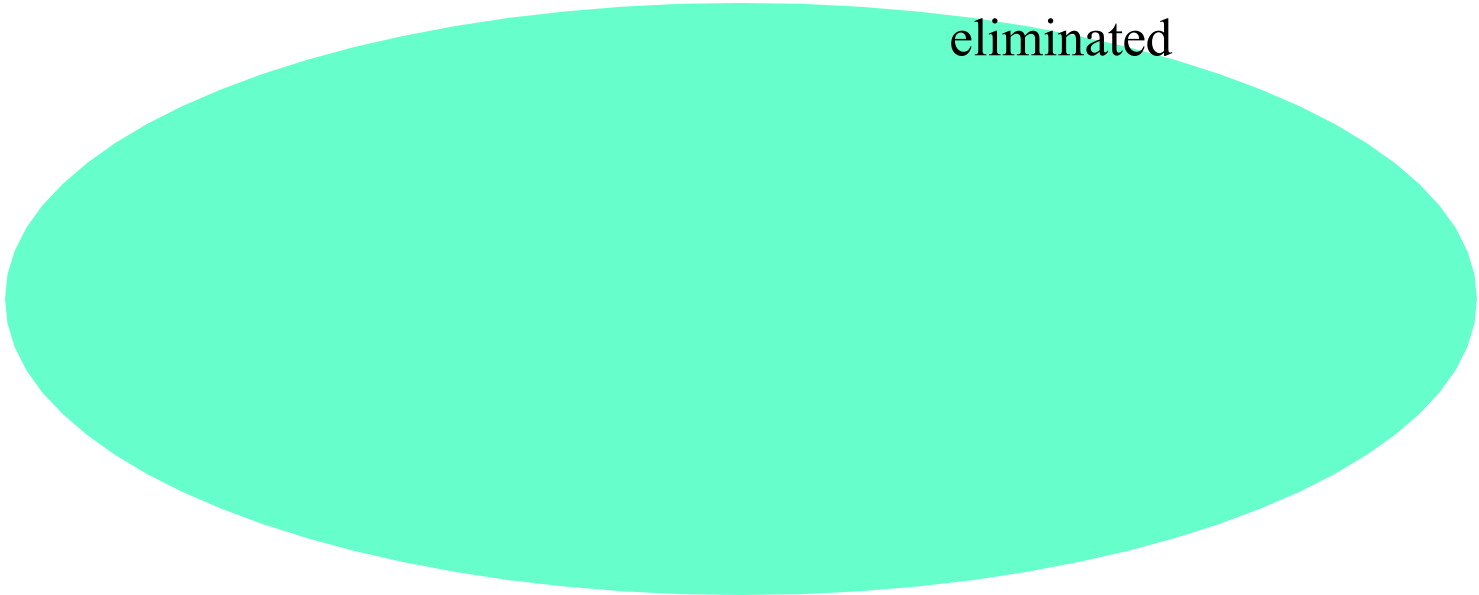
- Examples:

```
DEBUG:=0
```

```
if (DEBUG) print
```



Can be  
eliminated



# Copy Propagation

- What does it mean?
  - Given an assignment  $x = y$ , replace later uses of  $x$  with uses of  $y$ , provided there are no intervening assignments to  $x$  or  $y$ .
- When is it performed?
  - At any level, but usually early in the optimization process.
- What is the result?
  - Smaller code

# Copy Propagation

- $f := g$  are called copy statements or copies
- Use of  $g$  for  $f$ , whenever possible after copy statement

Example:

$x[i] = a;$

$sum = x[i] + a;$

$x[i] = a;$

$sum = a + a;$

- May not appear to be code improvement, but opens up scope for other optimizations.

# Local Copy Propagation

- Local copy propagation
  - Performed within basic blocks
  - Algorithm sketch:
    - traverse BB from top to bottom
    - maintain table of copies encountered so far
    - modify applicable instructions as you go

# Copy Propagation

$r2 := r1$

$r3 := r1 + r2$

$r2 := 5$

gets

$r2 := r1$

$r3 := r1 + r1$

$r2 := 5$

gets

$r3 := r1 + r1$

$r2 := 5$

By Copy Propagation

By Dead Assignment Elimination



# Loop Optimization

- Decrease the number of instructions in the inner loop
- Even if we increase no of instructions in the outer loop
- Techniques:
  - Code motion
  - Induction variable elimination
  - Strength reduction

# Optimization themes

- Don't compute if you don't have to
  - unused assignment elimination
- Compute at compile-time if possible
  - constant folding, loop unrolling, inlining
- Compute it as few times as possible
  - loop invariant code motion
- Compute it as cheaply as possible
  - strength reduction
- Enable other optimizations
  - constant and copy prop, pointer analysis
- Compute it with as little code space as possible
  - unreachable code elimination