Program Analysis (CSCI-4450/CSCI-6450) Spring 2024

www.cs.rpi.edu/~milanova/csci4450/

Ana Milanova
Office: Lally 314
Email: milanova@cs.rpi.edu

Office hours: Wednesdays Noon-2pm, Mondays after class or by appointment

Introductions

- Ana Milanova
- You
  - Tell us
    - Your name
    - Graduate or undergraduate student
    - Concentration, interests, research area

Outline

- Logistics

  www.cs.rpi.edu/~milanova/csci4450/

- Program analysis, introduction
- Course topics, tools and homework
- Introduction to Dataflow analysis

Logistics

- Course webpage

  http://www.cs.rpi.edu/~milanova/csci4450

  - Schedule, Notes, Reading
    - Schedule, lecture slides and assigned reading
  - Submitty
    - All homework submission and grades, **forum**
    - Check forum regularly for announcements

Logistics

- Recommended reading
  - **Compilers: Principles, Techniques and Tools,** by Alfred Aho, Monica Lam, Ravi Sethi, and Jeffrey Ullman (the Dragon Book)
  - **Types and Programming Languages**, by Benjamin C. Pierce
  - MIT's Open Courseware Program Analysis
  - **Principles of Program Analysis** by Flemming Nielson, Hanne Riis Nielson, and Chris Hankin
- Papers and lecture notes

Logistics

- Syllabus

  www.cs.rpi.edu/~milanova/csci4450/syllabus.htm

  Topics, outcomes, policies and grading
- In-class quizzes (6-8): 20%
- Homework assignments: 35%
- Paper presentation and critique: 12%
- Take-home final: 25%
- Attendance and participation: 8%

Logistics

- Assignments and take-home exam are to be completed individually unless otherwise specified

- Quizzes are in-class, open-notes, and may be completed individually or in small groups
- We will drop the lowest quiz

Logistics

- Graduate students enrolled in CSCI-6450
- Grade breakdown:
  - In-class quizzes (6-8): 15%
  - Homework assignments: 38%
  - Paper presentation: 12%
  - Take-home final: 27%
  - Attendance and participation: 8%
- Some assignments will have additional problems

Late Homework

- Homework assignments must be submitted in Submitty by 12pm on the due date
- You have 10 late days for the semester, with a max of 5 late days per assignment
- Exceptions to policy may be granted in rare cases

New This Term!

- Communication Intensive (CI) Designation (Pending)

- Paper presentation
- Two written assignments
- Participation

Academic Integrity

- Trust

- Discussion is allowed, even encouraged

- Taking written notes out of discussion is not allowed. Actual work should be your own
- Posting solutions on public forums (e.g., Discord, Github) is not allowed

Program Analysis

- Tools and techniques that help us reason about the run-time behavior of the program
  - Dynamic analysis – <u>during</u> program execution
    - Static instrumentation
    - Dynamic (binary) instrumentation (DBI)
  - Static analysis – <u>before</u> program execution
    - E.g., Java compiler's definite-assignment-check
    - E.g., Type checking and type inference are forms of static analysis
    - E.g., Dafny-style verification
    - And many, many more!

Program Analysis in Security

- E.g., is there uninitialized memory?
  - `char buf[64]; -> definition-free path -> use of buf`
  - or `char * buf = malloc(64); -> definition-free path -> use of buf`
- E.g., is there an information leak?

```
void * fp = &exit // sensitive source
…
x->f = fp;
y = x;
fp1 = y->f;
printf(“libc exit function @ %p\n” fp1) // sink
```

Program Analysis in Security

- E.g., is there an information leak?

```
gold += fight(&map[row*3+col]); // source
…
print_highscore(gold); // printf(param) leak, sink
```

- E.g., is there a TICTOU bug?

```
char * buf = malloc(bar->name_len);
...
modifies_bar(bar); // we can detect side-effects!
...
memcpy(buf, bar->name, bar->name_len);
```

- E.g., is there a buffer overflow? Many analyses

Our focus will be Static Analysis

- Many techniques
  - Decades of research and Turing Awards!

  - Dataflow analysis and abstract interpretation
    - Kildall ’73, Kam and Ullman ’77, Cousot & Cousot ’77
  - Types and type-based analysis
    - Following John Backus’ “Can Programming…” ‘78
  - Axiomatic semantics (i.e., Hoare Logic)
    - C.A.R. Hoare’s “An Axiomatic Basis for Computer Programming”, ’69

Static Analysis

- What this course is mostly about
  - How can we define the meaning of programs
  - How can we model behavior of programs, and prove theorems about programs
  - How can we use tools and build tools that reason about programs
- Many applications

Applications

- Compiler optimization, traditional application
  - We'll start with dataflow analysis

- Finding bugs, verifying the absence of bugs
- Improving security and privacy
  - Android, smart contracts, binary analysis
- Refactoring and testing
- Improving energy efficiency
- Education. Submitty uses static analysis!

Examples of Properties Deducible by Static Analysis

- Can `x` ever be null at program point `i: x.m()`
- Can `y` be different than `1` at program point `i: x = y*10`?
- Can `n` at `x[n]` cause out-of-bounds access?
- Does an app leak private data (e.g., phone number, phone identifier, location) to ad networks?
  - Answer: Yes!

Examples of Properties Deducible by Static Analysis

- What inputs avoid divide-by-zero at `x/y`?

```
{x!=1 && x!=-2}
y = x + 4;
if (x > 0) {
  y = x*x - 1;
}
else {
  y = y + x;
}
{y!=0}
x = x/y;
```

- Formalism of Axiomatic Semantics (Hoare logic)
  - Different from dataflow and types
- Allows us to specify program behavior with preconditions and postconditions that form logical assertions
  - Support complex logics
  - Enabiles reasoning about correctness

Nature of Static Analysis

- To remain computable, static analysis must approximate. It is undecidable to find exactly what happens at runtime
  - Typically, analysis errs on the safe (sound) side --- that is, it over-approximates
  - Sometimes, analysis is unsafe (unsound) --- that is, it under-approximates

Nature of Static Analysis, cont.

- A static analysis is said to be safe (also, sound, correct) if it over-approximates in the sense that it accounts for every possible execution path
  - E.g., suppose ground truth is **`x`** in **`{1,2}`**
  - A value-flow analysis that reports **`x`** in **`{1,2,21}`** is safe
  - An analysis that reports **`x`** in **`{0,2}`** is unsafe (unsound, incorrect)

Analysis Safety

- Safety is crucial when analysis enables compiler optimizations. Why?
  - E.g., an unsafe analysis may report that **`y`** is **`1`** at **`z = y*10`** while there is an execution path that sets **`y`** to **`10`**
    If the optimizing compiler changes **`z = y*10`** to **`z = 10`**, the program produces incorrect result along the path when **`y`** is **`10`**!

Analysis Safety

- Safety is often relinquished when analysis is used in static debugging tools. Why?
- E.g., suppose we have code that contains 10 “ground truth” null-pointer dereferences
  - Safe analysis A reports 100 potential null-pointer dereferences (all 10 “true” bugs and 90 “false-positives”).
  - Unsafe analysis B reports 10 potential null-pointer dereferences (8 “true” and 2 false-positives). Which one would you take?

Analysis Precision

- Analysis precision refers to how “close” results are to actual runtime
  - E.g., in our running example, an analysis that reports **`x`** in **`{1,2,3}`** is more precise than the one that reports **`x`** in **`{1,2,3,21}`**
  - Typically, we use the term precision with safe analysis (safe analysis has 100% recall)
- Wide spectrum of static analyses and tradeoff between cost and precision

Outline

- Logistics

  www.cs.rpi.edu/~milanova/csci4450/

- Static analysis, introduction

- Course topics, tools, and homework

- Introduction to Dataflow analysis

Course Topics

- Dataflow analysis
  - Lattices, transfer functions, dataflow frameworks
  - Classical analyses: constant propagation and points-to analysis
  - Binary analysis
- Abstract interpretation (a more powerful formalism)
  - Abstract vs. concrete semantics
  - Galois connections

Course Topics

- Types and type-based analysis
  - Simply typed Lambda calculus
  - Type systems and type soundness
  - Simple type inference
  - Hindley Milner type inference

Course Topics

- Axiomatic semantics
  - You know already: Hoare logic!
  - Logics to specify assertions (as you know them, **P** and **Q** in **{ P }** code **{ Q }**)
  - SMT solvers and proving Hoare triples

Historical Perspective

- “An axiomatic basis for computer programming” by C.A.R. Hoare 1969
  - Great enthusiasm about verification 1970-ties
- “Social processes and proofs of theorems and programs” by De Millo, Lipton, and Perlis 1979
  - Credited with setting back work on formal verification
- “Can programming be liberated… A functional style and its algebra of programs” by John Backus 1977
  - Research on functional programming, type theory
- Z3 theorem prover from Microsoft about 2005
  - Lots of new enthusiasm about verification and symbolic execution

Tools and Programming Languages

- Soot
- Z3
- Ghidra (optional)

- Java
- Haskell
- OCaml

Homework Assignments

- There will be 6-7 homework assignments
  - Each makes about 5-6% of your grade
  - Larger assignments are broken into 2-3 parts
  - Some are individual, some are team assignments

  - Submitty!

Homework Assignments

- HW1
  - Problem set to practice dataflow analysis
- HW2-HW4
  - Classical OO analyses in Soot: the CHA, RTA, and XTA family of analyses
- HW5-HW6
  - Problem set to practice abstract interpretation/type inference concepts
  - Implement simple type inference (and maybe Hindley Milner) in Haskell

Homework Assignments

- HW7
  - Implement a simple verifier for C using Z3

- If you are interested in Binary analysis, we'll replace <u>parts</u> of HW2-HW4 and HW7 with Ghidra projects
  - E.g., a taint analysis, a buffer overflow analysis, or other

Dataflow Analysis

Outline

- Motivation and origin of dataflow analysis: compiler optimization

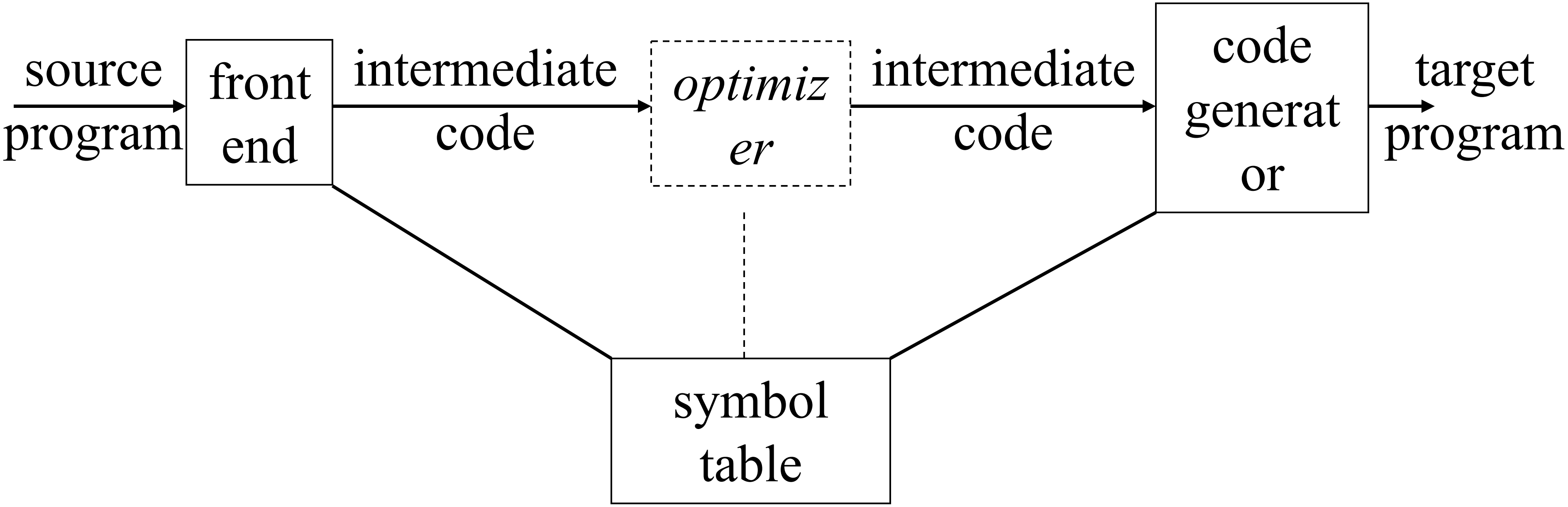
- Overview of the compiler
- Classical compiler optimizations
- Control flow graphs

- Reading:
  - Dragon Book, Chapter 9.1

Overview of the Compiler

- Phases of the compiler
  - Lexical Analyzer (scanner)
  - Syntax Analyzer (parser)
  - Semantic Analyzer and Intermediate Code Generator
  - Machine-Independent Code Optimizer
  - Code Generator
  - Machine-Dependent Code Optimizer

Overview of the Compiler



An optimization is a semantics-preserving transformation

Classical Compiler Optimizations

- We will show the classical optimizations using an example Fortran loop
- Opportunities for optimization due to automatic generation of intermediate code

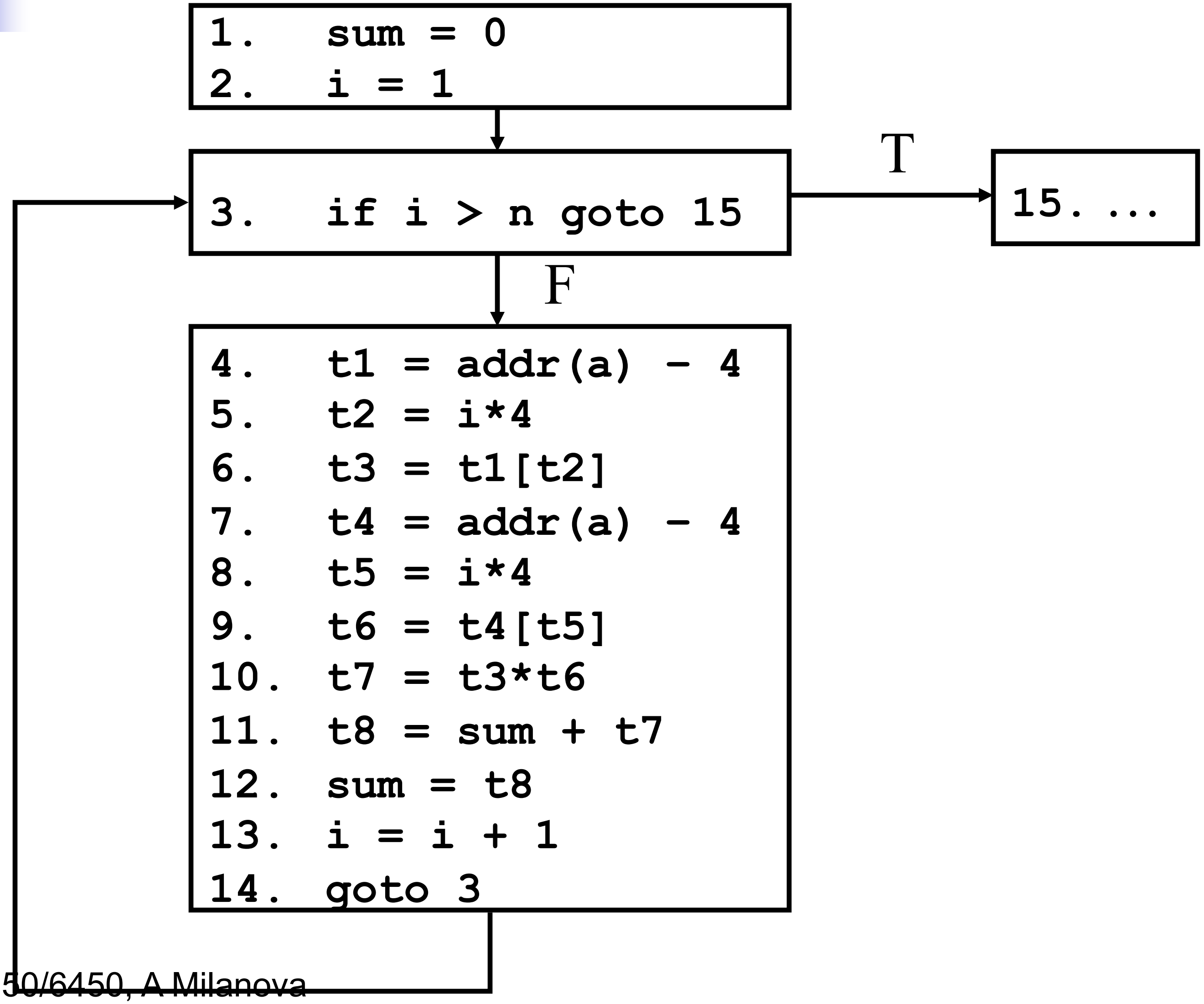
```
  …
sum = 0
do 10 i = 1, n
 10   sum = sum + a[i]*a[i]
  …
```

Three Address Code Intermediate Representation (IR)

```
1.   sum = 0                   initialize sum
2.   i = 1                     initialize loop counter
3.   if i > n goto 15          loop test, check for limit
4.   t1 = addr(a) – 4
5.   t2 = i * 4                a[i]
6.   t3 = t1[t2]
7.   t4 = addr(a) – 4
8.   t5 = i * 4                a[i]
9.   t6 = t4[t5]
10.  t7 = t3 * t6              a[i]*a[i]
11.  t8 = sum + t7
12.  sum = t8                  increment sum
13.  i = i + 1                        increment loop counter
14.  goto 3


15.  …
```

Control Flow Graph (CFG)



```
1.   sum = 0
2.   i = 1
```

```
3.   if i > n goto 15
```

T

```
15. ...
```

F

```
4.   t1 = addr(a) - 4
5.   t2 = i*4
6.   t3 = t1[t2]
7.   t4 = addr(a) - 4
8.   t5 = i*4
9.   t6 = t4[t5]
10.  t7 = t3*t6
11.  t8 = sum + t7
12.  sum = t8
13.  i = i + 1
14.  goto 3
```

Common Subexpression Elimination

```
1.   sum = 0
2.   i = 1
3.   if i > n goto 15
4.   t1 = addr(a) – 4
5.   t2 = i*4
6.   t3 = t1[t2]
7.   t4 = addr(a) – 4
8.   t5 = i*4
9.   t6 = t4[t5]
10.  t7 = t3*t6
11.  t8 = sum + t7
12.  sum = t8
13.  i = i + 1
14.  goto 3
15.  …
```

```
1.   sum = 0
2.   i = 1
3.   if i > n goto 15
4.   t1 = addr(a) – 4
5.   t2 = i*4
6.   t3 = t1[t2]
7.   t4 = addr(a) – 4
8.   t5 = i*4
9.   t6 = t4[t5]
10.  t7 = t3*t6
10a  t7 = t3*t3
11.  t8 = sum + t7
        12.  sum = t8
13.  i = i + 1
14.  goto 3
```

After Common Subexpression Elimination

```
1.  sum = 0
2.  i = 1
3.  if i > n goto 15
4.  t1 = addr(a) - 4
5.  t2 = i*4
6.  t3 = t1[t2]
10a t7 = t3*t3
11.  t8 = sum + t7
12.  sum = t8
13.  i = i + 1
14.  goto 3
```

Copy Propagation

```
1.   sum = 0
2.   i = 1
3.   if i > n goto 15
4.   t1 = addr(a) – 4
5.   t2 = i * 4
6.   t3 = t1[t2]
10a t7 = t3 * t3
11   t8 = sum + t7
12.  sum = t8
13.  i = i + 1
14.  goto 3
15.  …
```

```
1.   sum = 0
2.   i = 1
3.   if i > n goto 15
4.   t1 = addr(a) – 4
5.   t2 = i * 4
6.   t3 = t1[t2]
10a t7 = t3 * t3
11.  t8 = sum + t7
11a sum = sum + t7
12.  sum = t8
13.  i = i + 1
14.  goto 3
15.  …
```

After Copy Propagation

```
1.  sum = 0
2.   i = 1
3.   if i > n goto 15
4.   t1 = addr(a) - 4
5.   t2 = i * 4
6.   t3 = t1[t2]
10a t7 = t3 * t3
11a sum = sum + t7
13. i = i + 1
14. goto 3
15. …
```

Invariant Code Motion

```
1.   sum = 0
2.   i = 1
3.   if i > n goto 15
4.   t1 = addr(a) - 4
5.   t2 = i * 4
6.   t3 = t1[t2]
10a t7 = t3 * t3
11a sum = sum + t7
13. i = i + 1
14. goto 3
15. …
```

```
1.   sum = 0
2.   i = 1
2a   t1 = addr(a) - 4
3.   if i > n goto 15
4.   t1 = addr(a) - 4
5.   t2 = i * 4
6.   t3 = t1[t2]
10a t7 = t3 * t3
11a sum = sum + t7
13. i = i + 1
14. goto 3
15. …
```

After Invariant Code Motion

```
1.   sum = 0
2.   i = 1
2a   t1 = addr(a) - 4
3.   if i > n goto 15
5.   t2 = i * 4
6.   t3 = t1[t2]
10a t7 = t3 * t3
11a sum = sum + t7
13. i = i + 1
14. goto 3
15. …
```

Strength Reduction

```
1.   sum = 0
2.   i = 1
2a   t1 = addr(a) - 4
3.   if i > n goto 15
5.   t2 = i * 4
6.   t3 = t1[t2]
10a  t7 = t3 * t3
11a  sum = sum + t7
13.  i = i + 1
14.  goto 3
15.  …
```

```
1.   sum = 0
2.   i = 1
2a   t1 = addr(a) - 4
2b   t2 = i * 4
3.   if i > n goto 15
5.   t2 = i * 4
6.   t3 = t1[t2]
10a  t7 = t3 * t3
11a  sum = sum + t7
11b  t2 = t2 + 4
13.  i = i + 1
14.  goto 3
15.  …
```

After Strength Reduction

```
1.  sum = 0
2.  i = 1
2a  t1 = addr(a) - 4
2b  t2 = i * 4
3.  if i > n goto 15
6.  t3 = t1[t2]
10a t7 = t3 * t3
11a sum = sum + t7
11b t2 = t2 + 4
13. i = i + 1
14. goto 3
15. …
```

Test Elision and Induction Variable Elimination

```
1.   sum = 0
2.   i = 1
2a   t1 = addr(a) - 4
2b   t2 = i * 4
3.   if i > n goto 15
6.   t3 = t1[t2]
10a  t7 = t3 * t3
11a  sum = sum + t7
11b  t2 = t2 + 4
13.  i = i + 1
14.  goto 3
15.  …
```

```
1.   sum = 0
2.   i = 1
2a   t1 = addr(a) - 4
2b   t2 = i * 4
2c   t9 = n * 4
3.   if i > n goto 15
3a   if t2 > t9 goto 15
6.   t3 = t1[t2]
10a  t7 = t3 * t3
11a  sum = sum + t7
11b  t2 = t2 + 4
13.  i = i + 1
14.  goto 3a
15.  …
```

After Test Elision and Induction Variable Elimination

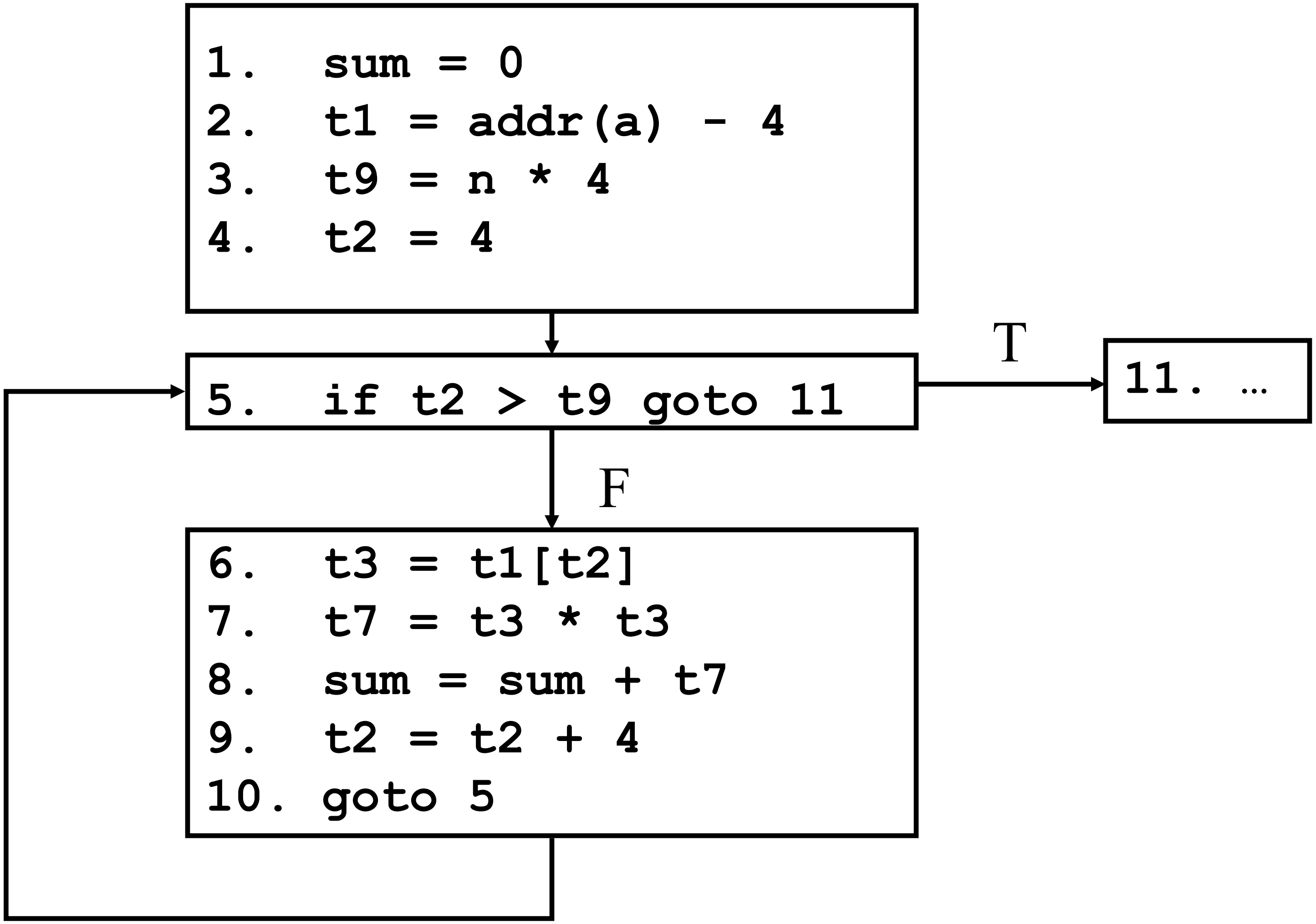
```
1.   sum = 0
2.   i = 1
2a   t1 = addr(a) - 4
2b   t2 = i * 4
2c   t9 = n * 4
3a   if t2 > t9 goto 15
6.   t3 = t1[t2]
10a t7 = t3 * t3
11a sum = sum + t7
11b t2 = t2 + 4
14. goto 3a
15. …
```

Constant Propagation and Dead Code Elimination

```
1.   sum = 0
2.   i = 1
2a   t1 = addr(a) - 4
2b   t2 = i * 4
2c   t9 = n * 4
3a   if t2 > t9 goto 15
6.   t3 = t1[t2]
10a  t7 = t3 * t3
11a  sum = sum + t7
11b  t2 = t2 + 4
14.  goto 3a
15.  …
```

```
1.   sum = 0
2.   i = 1
2a   t1 = addr(a) - 4
2b   t2 = i * 4
2c   t9 = n * 4
2d   t2 = 4
3a   if t2 > t9 goto 15
6.   t3 = t1[t2]
10a  t7 = t3 * t3
11a  sum = sum + t7
11b  t2 = t2 + 4
14.  goto 3a
15.  …
```

New Control Flow Graph



```
1.   sum = 0
2.   t1 = addr(a) - 4
3.   t9 = n * 4
4.   t2 = 4
```

```
5.   if t2 > t9 goto 11
```

```
11.  …
```

```
6.   t3 = t1[t2]
7.   t7 = t3 * t3
8.   sum = sum + t7
9.   t2 = t2 + 4
10.  goto 5
```

Classical Compiler Optimizations

- To summarize
  - Common subexpression elimination
  - Copy propagation
  - Strength reduction
  - Test elision and induction variable elimination
  - Constant propagation
  - Dead code elimination
- Dataflow analysis <u>enables</u> these optimizations

Building the Control Flow Graph

Build the CFG from linear 3-address code:

- Step 1: partition code into basic blocks
  - Basic blocks are the nodes in the CFG
- Step 2: add control flow edges

- Aside: in Principles of Software, we built a CFG from structured (AST) IR:
  - S ::= x = y op z | S;S | if (b) then S else S |
    while (b) S

Step 1. Partition Code Into Basic Blocks

1. Determine the *leader* statements:

    (i) First program statement

    (ii) Targets of gotos, conditional or unconditional

    (iii) Any statement following a goto

2. For each leader, its basic block consists of the leader and all statements up to, but not including, the next leader or the end of the program

Question. Find the Leader Statements

```
1.  sum = 0
2.  i = 1
3.  if i > n goto 15
4.  t1 = addr(a) - 4
5.  t2 = i*4
6.  t3 = t1[t2]
7.  t4 = addr(a) - 4
8.  t5 = i*4
9.  t6 = t5[t5]
10. t7 = t3*t6
11. t8 = sum + t7
12. sum = t8
13. i = i + 1
14. goto 3
15. …
```

Step 2. Add Control Flow Edges

- There is a directed edge from basic block B_1 to block B_2 if B_2 can immediately follow B_1 in some execution sequence
- Determine edges as follows:
  (i) There is an edge from B_1 to B_2 if B_2 follows B_1 in three-address code, and B_1 does not end in an unconditional goto

  (ii) There is an edge from B_1 to B_2 if there is a goto from the last statement in B_1 to the first statement in B_2

Question. Add Control Flow Edges

```
1.    sum = 0
2.    i = 1
3.    if i > n goto 15
4.    t1 = addr(a) - 4
5.    t2 = i*4
6.    t3 = t1[t2]
7.    t4 = addr(a) - 4
8.    t5 = i*4
9.    t6 = t5[t5]
10.   t7 = t3*t6
11.   t8 = sum + t7
12.   sum = t8
13.   i = i + 1
14.   goto 3
15.   …
```

Next Class

- Dataflow analysis
- Four classical dataflow analysis problems