CSE 401/M501 – Compilers

Intermediate Representations

Hal Perkins

Spring 2023

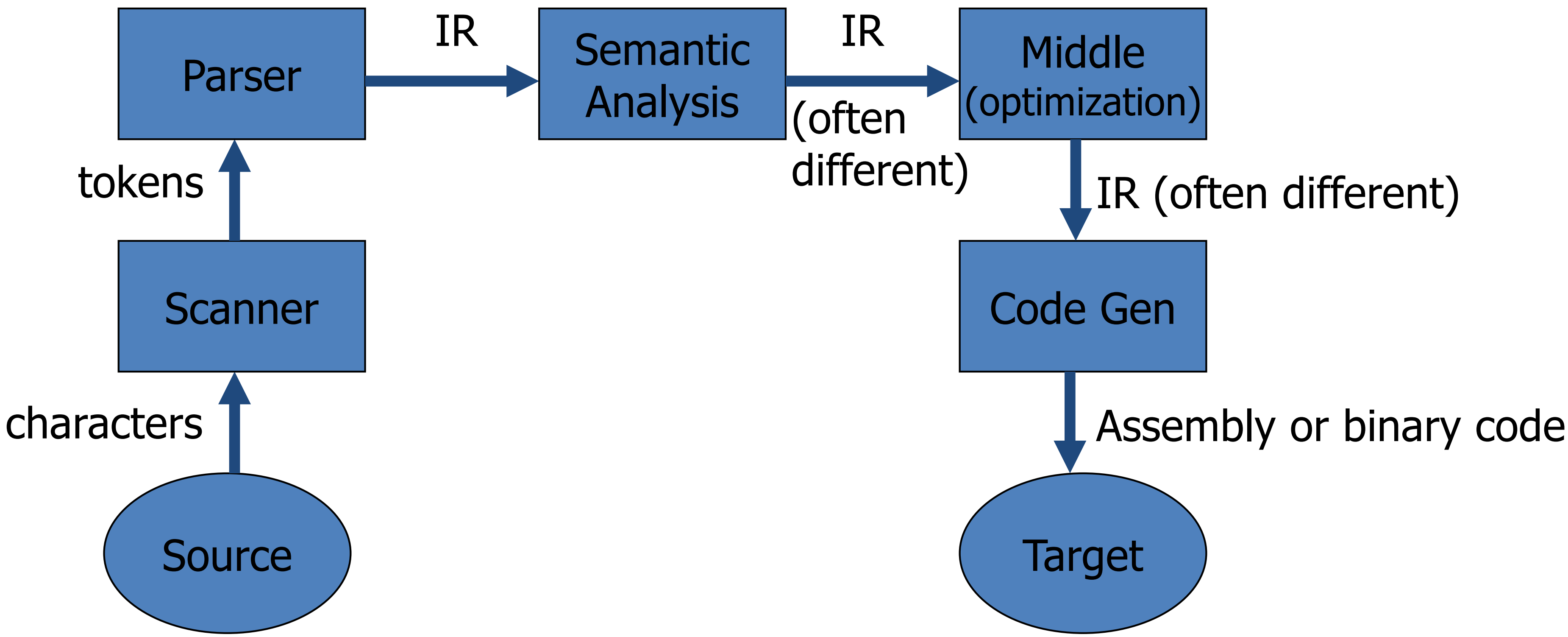
Administrivia

- Short hw3 due Monday – 1 late day max
- Midterm next Friday – topics + old exams online; blank 5x8 cards available at the end of class
  - Review in sections next week
- Semantics/typechecking project assignment posted now; due Thursday, Nov. 16, 2 weeks after the midterm
  - Fair amount to do, so get started and work steadily; don't ignore completely until after midterm…
    - And ***definitely*** plan to get a lot done next weekend after the midterm, starting with symbol tables, Type ADT and methods, and other data structures
      - Required check-in showing APIs for symbol table and type ADTs during Nov. 9 sections - will award a point or something ☺

Agenda

- Survey of Intermediate Representations
  - Graphical
    - Concrete/Abstract Syntax Trees (ASTs)
    - Control Flow Graph
    - Dependence Graph
  - Linear Representations
    - Stack Based
    - 3-Address
- Several of these will show up as we explore program analysis and optimization

Compiler Structure (review)



Intermediate Representations

- In most compilers, the parser builds an intermediate representation of the program
  - Typically an AST, as in the MiniJava project
- Rest of the compiler transforms the IR to improve ("optimize") it and eventually translate to final target code
  - Typically will transform initial IR to one or more different IRs along the way
- Some general examples now; more specifics later as needed

IR Design

- Decisions affect speed and efficiency of the rest of the compiler
  - General rule: compile time is important, but performance/quality of generated code is often more important
  - Typical case for production code: compile a few times, run many times
    - Although the reverse is true during development
  - So make choices that improve compiler speed as long as they don't compromise the desired result

IR Design

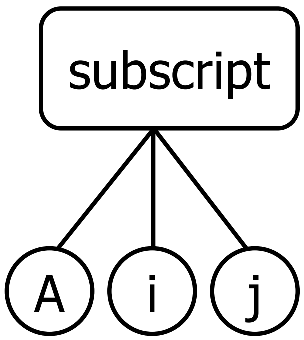
- Desirable properties
  - Easy to generate
  - Easy to manipulate
  - Expressive
  - Appropriate level of abstraction
- Different tradeoffs depending on compiler goals
- Different tradeoffs in different parts of the same compiler
  - So often different IRs in different parts

IR Design Taxonomy

- Structure
  - Graphical (trees, graphs, etc.)
  - Linear (code for some abstract machine)
  - Hybrids are common (e.g., control-flow graphs whose nodes are basic blocks of linear code)
- Abstraction Level
  - High-level, near to source language
  - Low-level, closer to machine (exposes more details to compiler)

Examples: Array Reference

A[i,j]



or

t1 ← A[i,j]

```
loadI   1   => r1
sub  rj,r1  => r2
loadI  10  => r3
mult r2,r3 => r4
sub  ri,r1  => r5
add  r4,r5 => r6
loadI @A  => r7
add  r7,r6 => r8
load r8     => r9
```

Levels of Abstraction

- Key design decision: how much detail to expose
  - Affects possibility and profitability of various optimizations
    - Depends on compiler phase: some semantic analysis & optimizations are easier with high-level IRs close to the source code. Low-level usually preferred for other optimizations, register allocation, code generation, etc.
  - Structural (graphical) IRs are typically fairly high-level – but are also used for low-level
  - Linear IRs are typically low-level
  - But these generalizations don't always hold

Graphical IRs

- IRs represented as a graph (or tree)
- Nodes and edges typically reflect some structure of the program
  - E.g., source code, control flow, data dependence
- May be large (especially syntax trees)
- High-level examples: syntax trees, DAGs
  - Generally used in early phases of compilers
- Other examples: control flow graphs and data dependency graphs
  - Often used in optimization and code generation

Concrete Syntax Trees

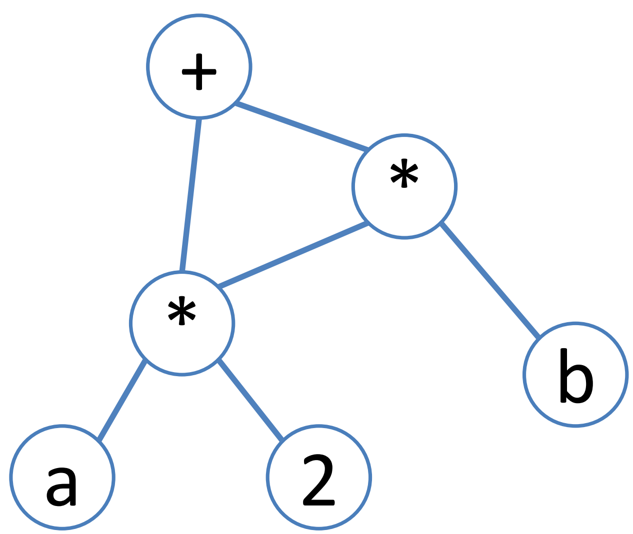
- The full grammar is needed to guide the parser, but contains many extraneous details
  - Chain productions
  - Rules that control precedence and associativity
- Typically the full concrete syntax tree (parse tree) is not used explicitly, but sometimes we want it (structured source code editors or for transformations, …)

Abstract Syntax Trees

- Want only essential structural information
  - Omit extra junk
- Can be represented explicitly as a tree or in a linear form
  - Example: LISP/Scheme/Racket S-expressions are essentially ASTs (e.g., **`(* 2 (+ 3 4)`**)
- Common output from parser; used for static semantics (type checking, etc.) and sometimes high-level optimizations

DAGs (Directed Acyclic Graphs)

- Variation on ASTs to capture shared substructures
- Pro: saves space, exposes redundant sub-expressions
- Con: less flexibility if part of tree should be changed
- Example: (a*2) + ((a*2) * b)



Linear IRs

- Pseudo-code for some abstract machine
- Level of abstraction varies
- Simple, compact data structures
  - Commonly used: arrays, linked structures
- Examples: 3-address code, stack machine code

```
t1 ← 2
t2 ← b
t3 ← t1 * t2
t4 ← a
t5 ← t4 – t3
```

- Fairly compact
- Compiler can control reuse of names – clever choice can reveal optimizations
- ILOC & similar code

```
push 2
push b
multiply
push a
subtract
```

- Each instruction consumes top of stack & pushes result
- Very compact
- Easy to create and interpret
- Java bytecode, MSIL

Abstraction Levels in Linear IR

- Linear IRs can also be close to the source language, very low-level, or somewhere in between.
- Example: Linear IRs for C array reference a[i][j+2]

- High-level: $t1 \leftarrow a[i,j+2]$

More IRs for a[i][j+2]

- Medium-level

  t1 ← j + 2

  t2 ← i * 20

  t3 ← t1 + t2

  t4 ← 4 * t3

  t5 ← addr a

  t6 ← t5 + t4

  t7 ← *t6

retains basic symbolic info about variables

- Low-level

  r1 ← [fp-4]

  r2 ← r1 + 2

  r3 ← [fp-8]

  r4 ← r3 * 20

  r5 ← r4 + r2

  r6 ← 4 * r5

  r7 ← fp – 216

  f1 ← [r7+r6]

expose all details of the low-level layout; explicit memory refs and calcs

Abstraction Level Tradeoffs

- High-level: good for some high-level optimizations, semantic checking; but can't optimize things that are hidden – like address arithmetic for array subscripting
- Low-level: need for good code generation and resource utilization in back end but loses some semantic knowledge (e.g., variables, data aggregates, source relationships are usually missing)
- Medium-level: more detail but keeps more higher-level semantic information – great for machine-independent optimizations. Many (all?) optimizing compilers work at this level
- Many compilers use all 3 in different phases

Three-Address Code (TAC)

- Usual form: x ← y op z
  - One operator
  - Maximum of 3 names
  - (Copes with: nullary x ← y and unary x ← op y)

- Eg: x = 2 * (m + n) becomes

  t1 ← m + n;   t2 ← 2 * t1;   x ← t2
  - You may prefer: add t1, m, n;   mul t2, 2, t1;   mov x, t2
  - Invent as many new temp names as needed. "expression temps" – don't correspond to any user variables; de-anonymize expressions

- Store in a quad(ruple)
  - <lhs, rhs1, op, rhs2>

Three Address Code

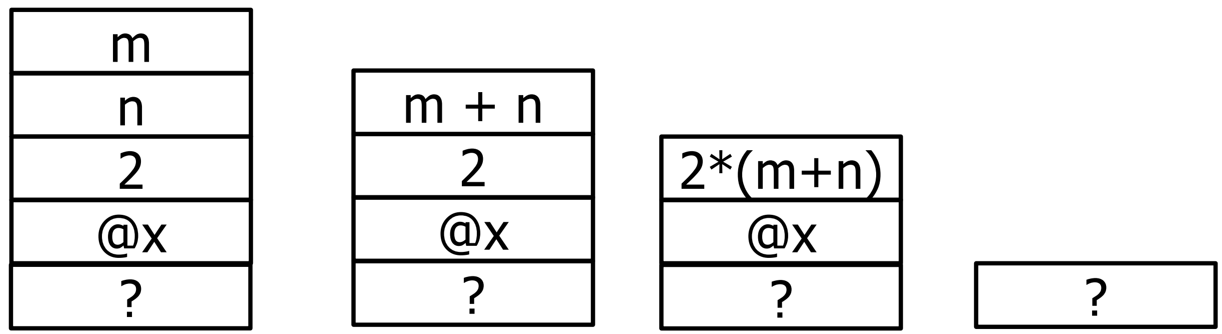
- Advantages
  - Resembles code for actual machines
  - Explicitly names intermediate results
  - Compact
  - Often easy to rearrange

- Various representations
  - Quadruples, triples, SSA (Static Single Assignment)
  - We will see much more of this…

Stack Machine Code Example

Hypothetical code for x = 2 * (m + n)

```
pushaddr   x
pushconst  2
pushval    n
pushval    m
add
mult
store
```



Compact: common opcodes just 1 byte wide; instructions have 0 or 1 operand

Stack Machine Code

- Originally used for stack-based computers (famous example: B5000, ~1961)
- Often used for virtual machines. Classic examples:
  - Pascal – pcode
  - Forth
  - Java bytecode in a .class files (generated by Java compiler)
  - MSIL in a .dll or .exe assembly (generated by C#/F#/VB compiler)
- Advantages
  - Compact; mostly 0-address opcodes (fast download over slow network)
  - Easy to generate; easy to write a front-end compiler, leaving the "heavy lifting" and optimizations to the JIT
  - Simple to interpret or compile to machine code
- Disadvantages
  - Somewhat inconvenient/difficult to optimize directly
  - Does not match up with modern chip architectures

Hybrid IRs

- Combination of structural and linear
- Level of abstraction varies
- Most common example: control-flow graph (CFG)

Control Flow Graph (CFG)

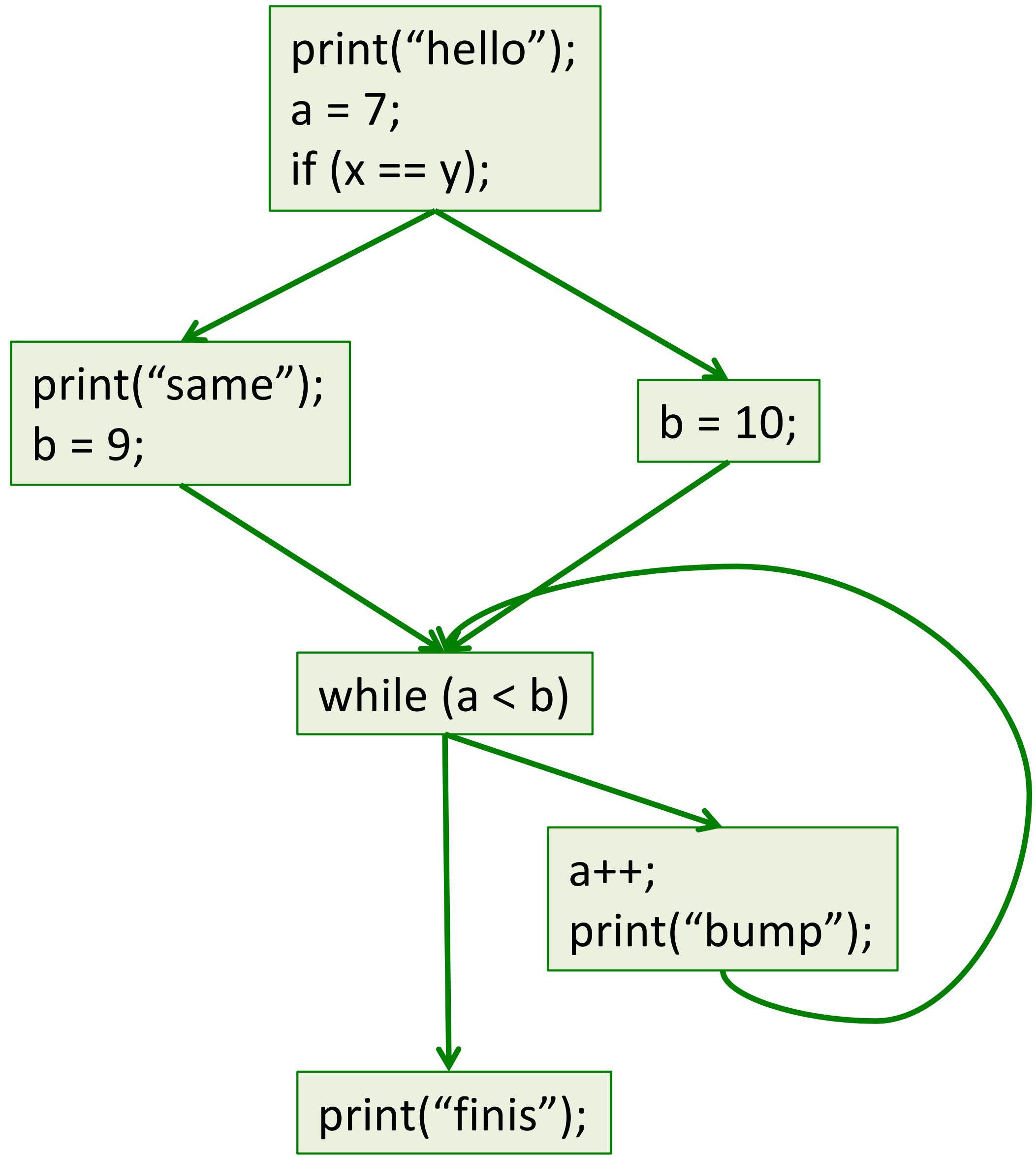
- Nodes: *basic blocks*
- Edges: represent possible flow of control from one block to another, i.e., possible execution orderings
  - Edge from A to B if B could execute immediately after A in some possible execution
- Required for much of the analysis done during optimization phases

Basic Blocks

- Fundamental concept in analysis/optimization
- A *basic block* is:
  - A sequence of code
  - One entry, one exit
  - Always executes as a single unit (“straightline code”) – so it can be treated as an indivisible unit
    - We’ll ignore exceptions, at least for now
- Usually represented as some sort of a list although Trees/DAGs are possible

CFG Example

```
print(“hello”);
a=7;
if (x == y) {
  print(“same”);
  b = 9;
} else {
  b = 10;
}
while (a < b) {
  a++;
  print(“bump”);
}
print(“finis”);
```



Basic Blocks: Start with Tuples

```
1 i = 1
2 j = 1
3 t1 = 10 * i
4 t2 = t1 + j
5 t3 = 8 * t2
6 t4 = t3 - 88
7 a[t4] = 0
8 j = j + 1
9 if j <= 10 goto #3
10 i = i + 1
11 if i <= 10 goto #2
12 i = 1
13 t5 = i - 1
14 t6 = 88 * t5
15 a[t6] = 1
16 i = i + 1
17 if i <= 10 goto #13
```

Typical "tuple stew" - IR generated by traversing an AST

Partition into Basic Blocks:

- Sequence of consecutive instructions
- No jumps into the middle of a BB
- No jumps out of the middles of a BB
- "I've started, so I'll finish"
- (Ignore exceptions)

Basic Blocks: Leaders

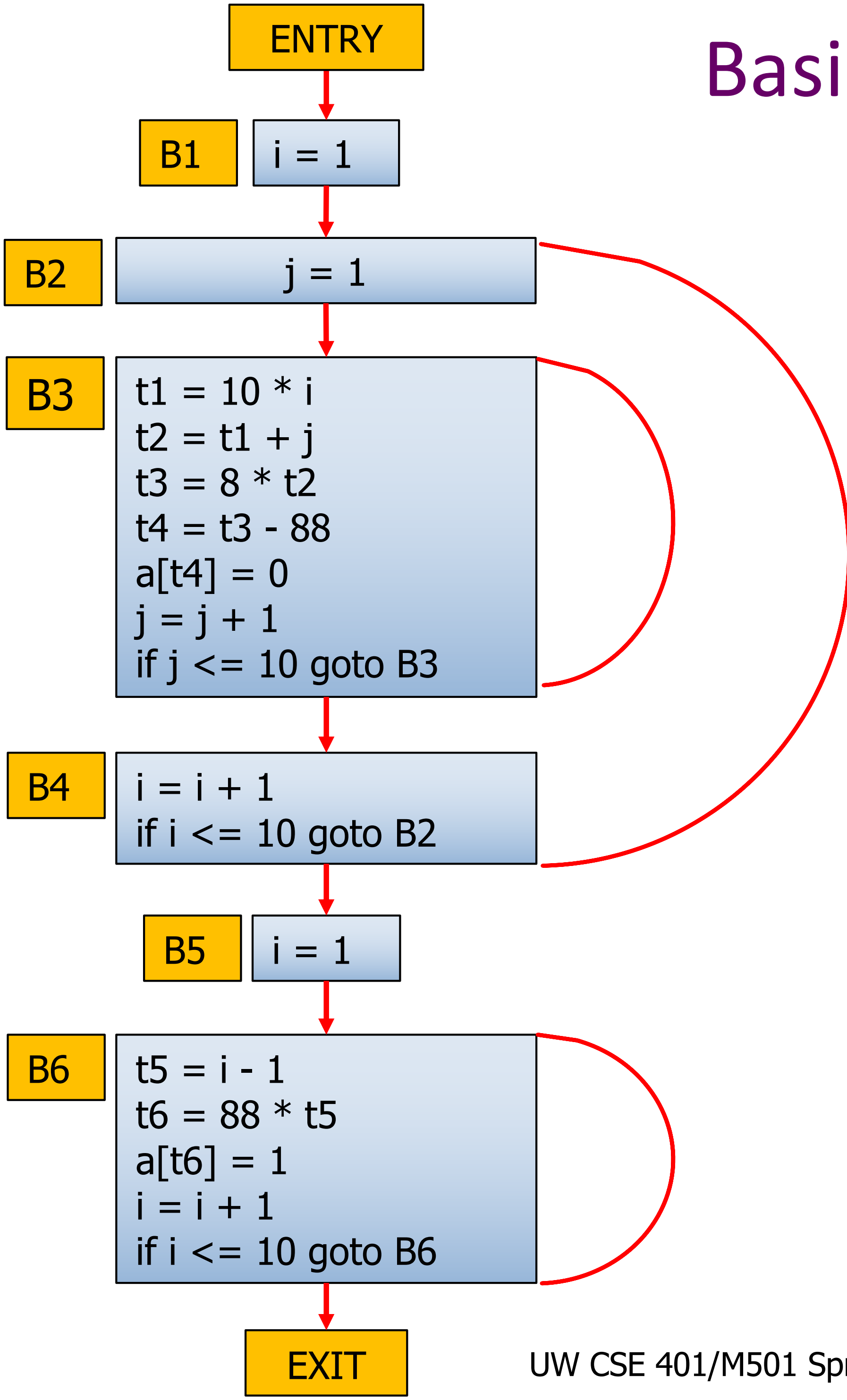
```
1 i = 1
2 j = 1
3 t1 = 10 * i
4 t2 = t1 + j
5 t3 = 8 * t2
6 t4 = t3 - 88
7 a[t4] = 0
8 j = j + 1
9 if j <= 10 goto #3
10 i = i + 1
11 if i <= 10 goto #2
12 i = 1
13 t5 = i - 1
14 t6 = 88 * t5
15 a[t6] = 1
16 i = i + 1
17 if i <= 10 goto #13
```

Identify Leaders (first instruction in a basic block):

- First instruction is a leader
- Any target of a branch/jump/goto
- Any instruction immediately after a branch/jump/goto

Leaders in red. Why is each leader a leader?

Basic Blocks: Flowgraph



Control Flow Graph ("CFG", again!)

- 3 loops total
- 2 of the loops are nested

Most of the execution likely spent in loop bodies; that's where to focus efforts at optimization

Identifying Basic Blocks: Recap

- Perform linear scan of instruction stream

- A basic blocks begins at each instruction that is:
  - The beginning of a method
  - The target of a branch
  - Immediately follows a branch or return

Dependency Graphs

- Often used in conjunction with another IR
- Data dependency: edges between nodes that reference common data
- Examples
  - Block A defines x then B reads it (RAW – read after write)
  - Block A reads x then B writes it (WAR – “anti-dependence”)
  - Blocks A and B both write x (WAW) – order of blocks must reflect original program semantics
- These restrict reorderings the compiler can do

What IR to Use?

- Common choice: all(!)
  - AST used in early stages of the compiler
    - Closer to source code
    - Good for semantic analysis
    - Facilitates some higher-level optimizations
  - Lower to linear IR(s) for optimization and codegen
    - Closer to machine code
    - Exposes machine-related optimizations
    - Use to build control-flow graph
  - Hybrid (graph + linear IR = CFG) for dataflow & opt

Coming Attractions

- "Code shape" – target code for language constructs
- Survey of compiler "optimizations"
- Analysis and transformation algorithms for optimizations (including SSA IR)
- Back-end organization in production compilers
  - Instruction selection and scheduling, register allocation
- Other topics depending on time

- And we'll also slip in project-specific codegen