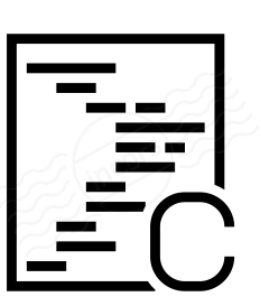
Basic Program Analysis

Suman Jana

*some slides are borrowed from Baishakhi Ray and Ras Bodik

Our Goal



Source code

Program Analyzer

Security bugs

Program analyzer must be able to understand program properties (e.g., can a variable be NULL at a particular program point?)

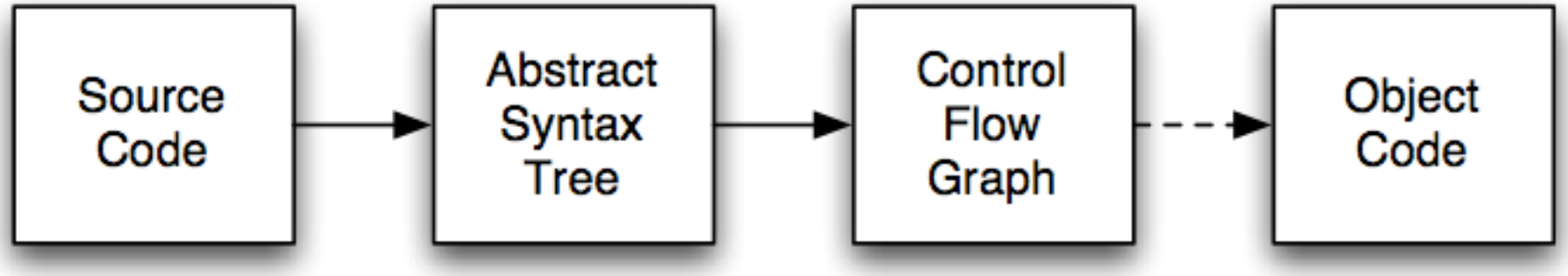
Must perform control and data flow analysis

Do we need to implement control and data flow analysis from scratch?

- Most modern compilers already perform several types of such analysis for code optimization
  - We can hook into different layers of analysis and customize them
  - We still need to understand the details

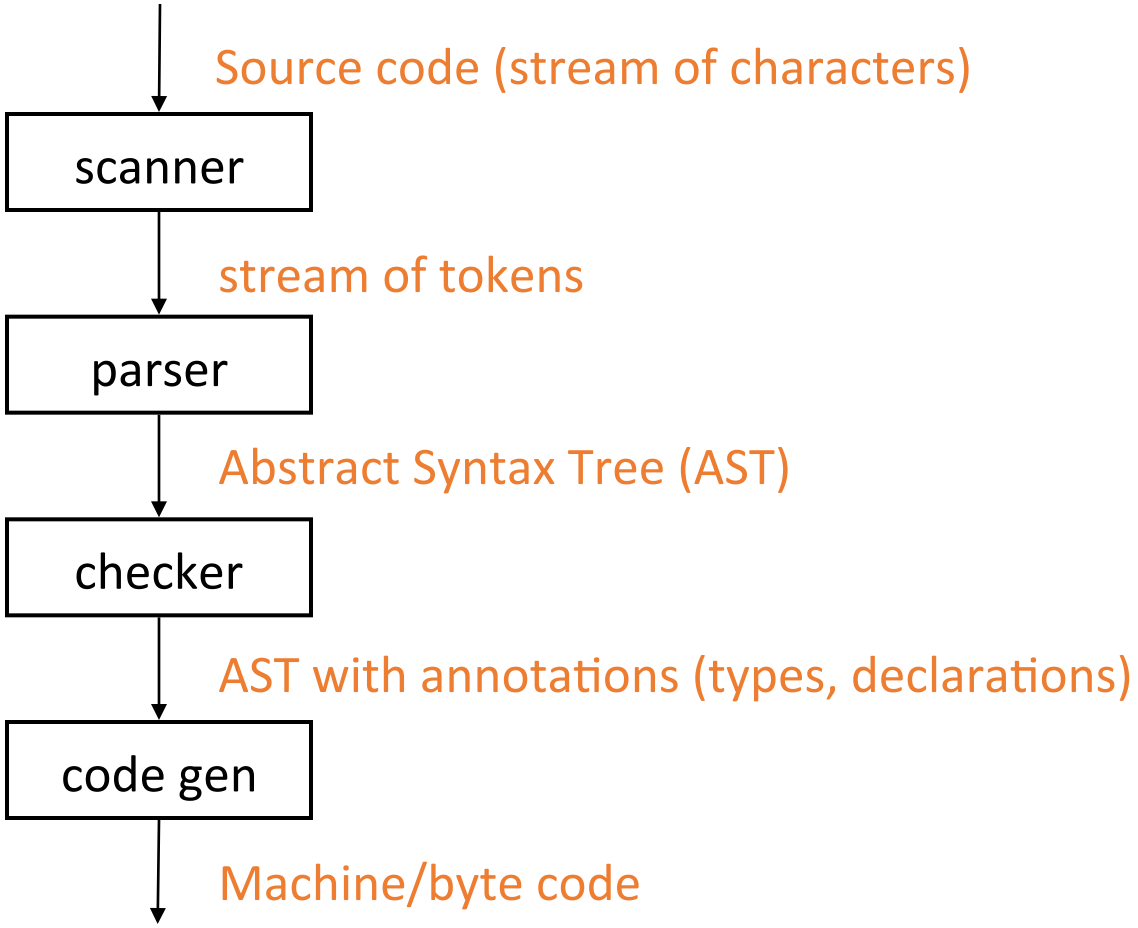
- LLVM (http://llvm.org/) is a highly customizable and modular compiler framework
  - Users can write LLVM passes to perform different types of analysis
  - Clang static analyzer can find several types of bugs
  - Can instrument code for dynamic analysis

Compiler Overview



- Abstract Syntax Tree : Source code parsed to produce AST
- Control Flow Graph: AST is transformed to CFG
- Data Flow Analysis: operates on CFG

The Structure of a Compiler



Syntactic Analysis

- **Input:** sequence of tokens from scanner
- **Output:** abstract syntax tree
- Actually,
  - parser first builds a parse tree
  - AST is then built by translating the parse tree
  - parse tree rarely built explicitly; only determined by, say, how parser pushes stuff to stack

Example

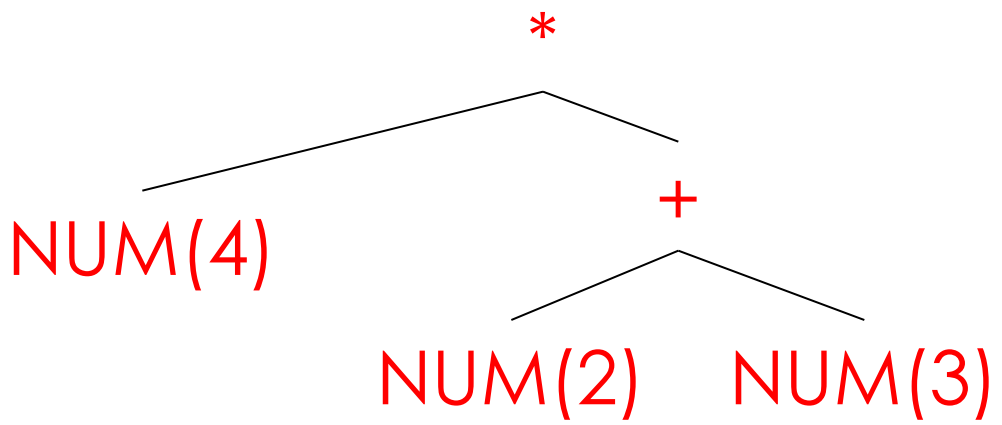
- Source Code

  4*(2+3)

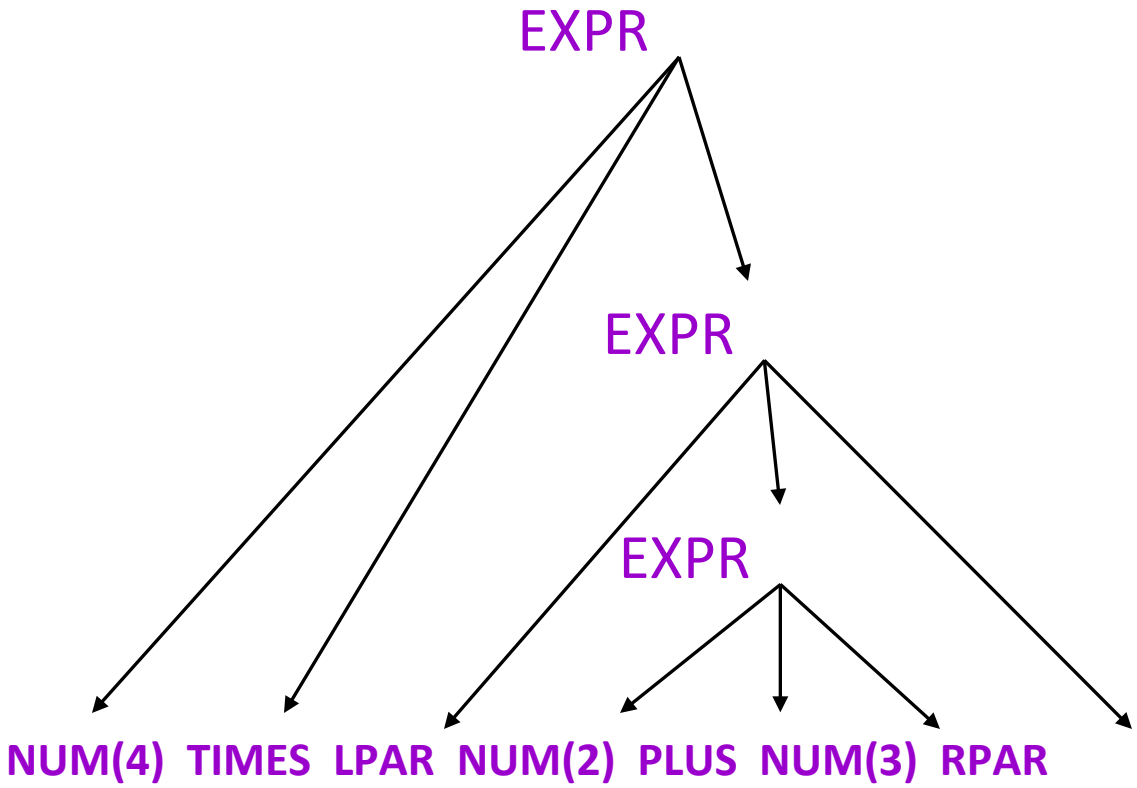
- Parser input

  NUM(4) TIMES LPAR NUM(2) PLUS NUM(3) RPAR

- Parser output (AST):



Parse tree for the example: 4*(2+3)



leaves are tokens

Another example

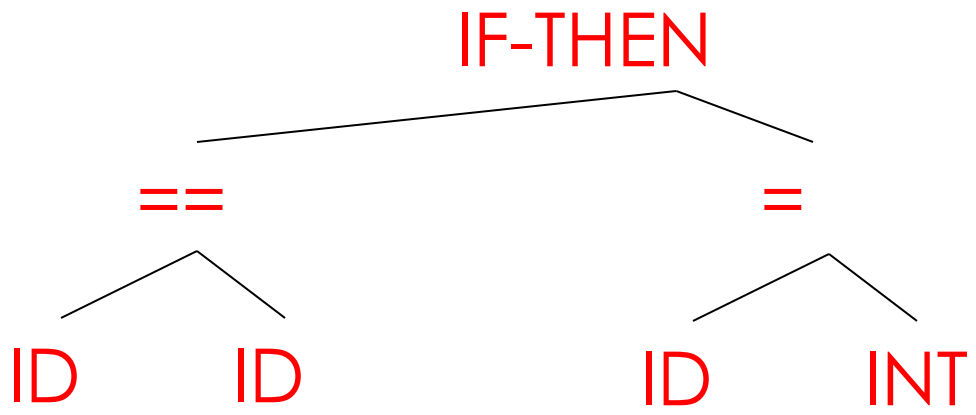
- Source Code

  if (x == y) { a=1; }

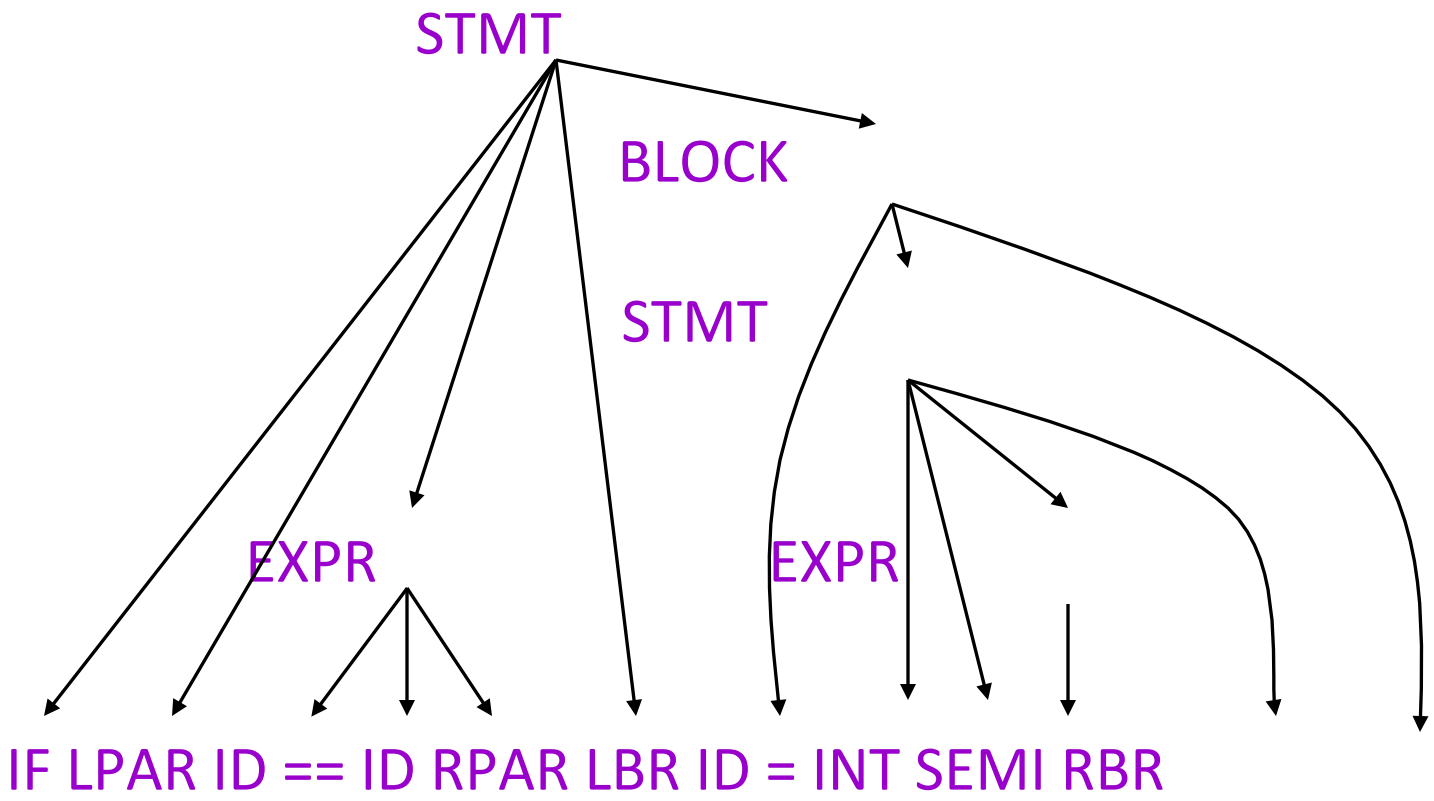
- Parser input

  IF LPAR ID EQ ID RPAR LBR ID AS INT SEMI RBR

- Parser output (AST):



Parse tree for example: if (x==y) {a=1;}



leaves are tokens

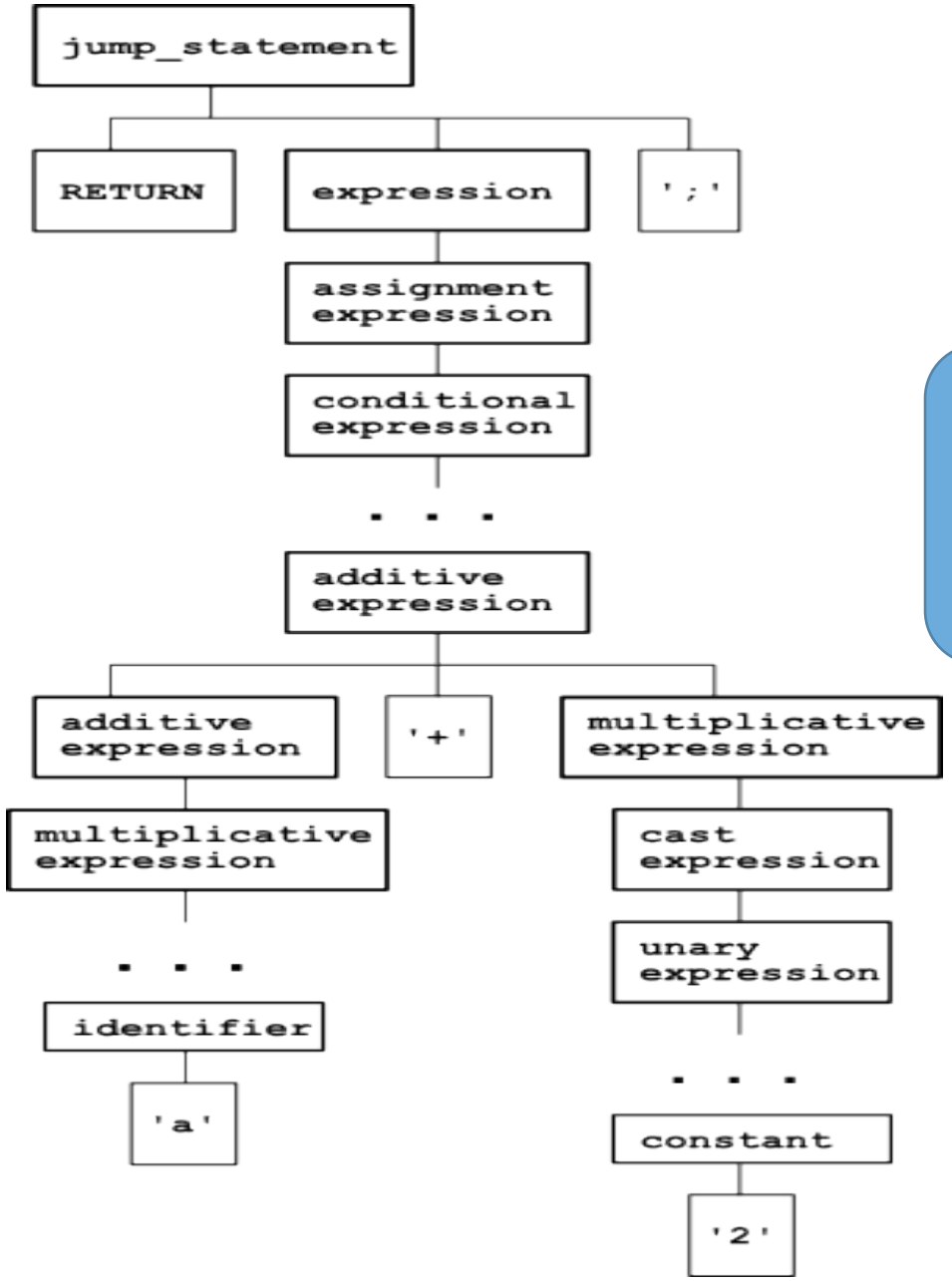
Parse Tree

- Representation of grammars in a tree-like form.

> A parse tree pictorially shows how the start symbol of a grammar derives a string in the language. … Dragon Book

- Is a one-to-one mapping from the grammar to a tree-form.

C Statement: return a + 2



a very formal representation that strictly shows how the parser understands the statement return a + 2;

Abstract Syntax Tree (AST)

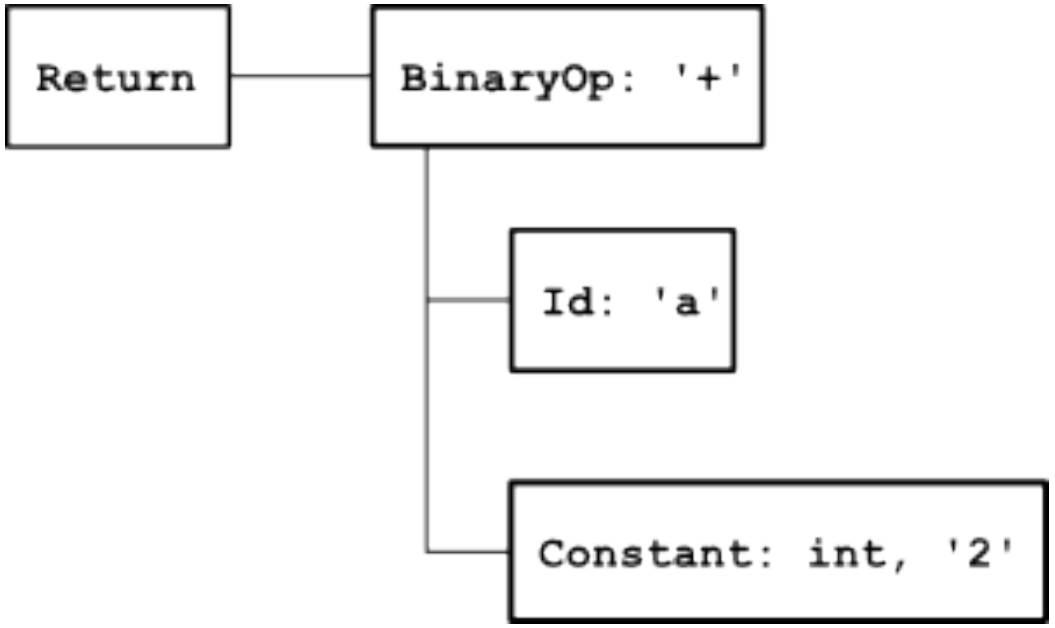
- Simplified syntactic representations of the source code, and they're most often expressed by the data structures of the language used for implementation

> ASTs differ from parse trees because superficial distinctions of form, unimportant for translation, do not appear in syntax trees.. … Dragon Book

- Without showing the whole syntactic clutter, represents the parsed string in a structured way, discarding all information that may be important for parsing the string, but isn't needed for analyzing it.

AST

C Statement: return a + 2



Disadvantages of ASTs

- AST has many similar forms
  - E.g., for, while, repeat...until
  - E.g., if, ?:, switch

```
int x = 1 // what's the value of x ?
          // AST traversal can give the answer, right?

What about int x; x = 1; or int x= 0; x += 1;  ?
```

- Expressions in AST may be complex, nested
  - (x * y) + (z > 5 ? 12 * z : z + 20)

- Want simpler representation for analysis
  - ...at least, for dataflow analysis

Control Flow Graph & Analysis

Representing Control Flow

High-level representation

–Control flow is implicit in an AST

Low-level representation:

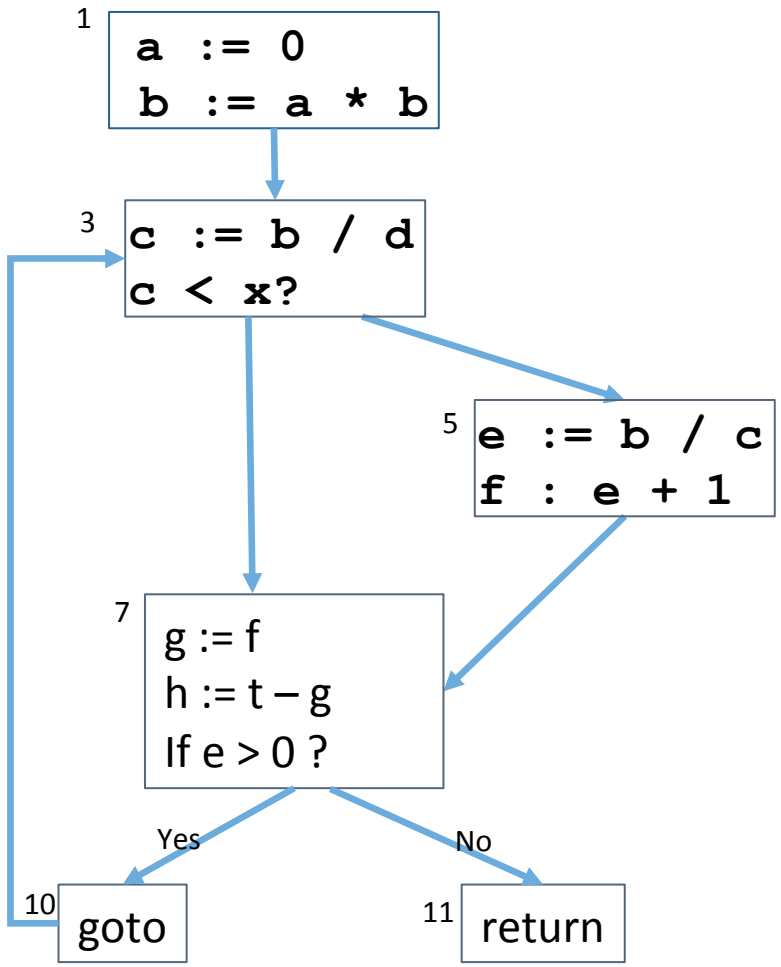
–Use a **Control-flow graph (CFG)**

–Nodes represent statements (low-level linear IR)

–Edges represent explicit flow of control

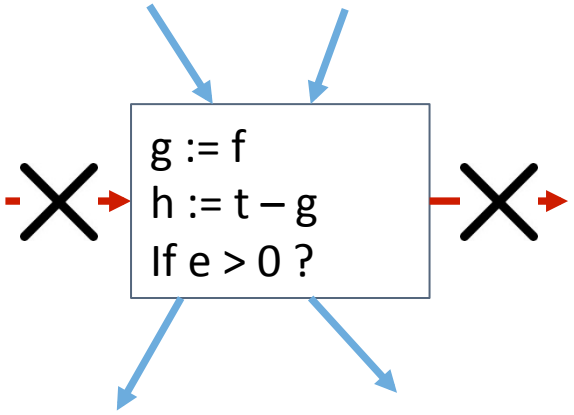
What Is Control-Flow Analysis?

```
1         a := 0
2         b := a * b
3   L1:   c := b/d
4         if c < x goto L2
5         e := b / c
6         f := e + 1
7   L2:   g := f
8         h := t - g
9         if e > 0 goto L3
10  goto L1
11  L3:   return
```



Basic Blocks

- A **basic block** is a sequence of straight line code that can be entered only at the beginning and exited only at the end



Building basic blocks

– Identify **leaders**

–The first instruction in a procedure, or

–The target of any branch, or

–An instruction immediately following a branch (implicit target)

– Gobble all subsequent instructions until the next leader

Basic Block Example

```
1        a := 0
2        b := a * b
3   L1:  c := b/d
4        if c < x goto L2
5        e := b / c
6        f := e + 1
7   L2:  g := f
8        h := t - g
9        if e > 0 goto L3
10  goto L1
11  L3:  return
```

Leaders?

- {1, 3, 5, 7, 10, 11}

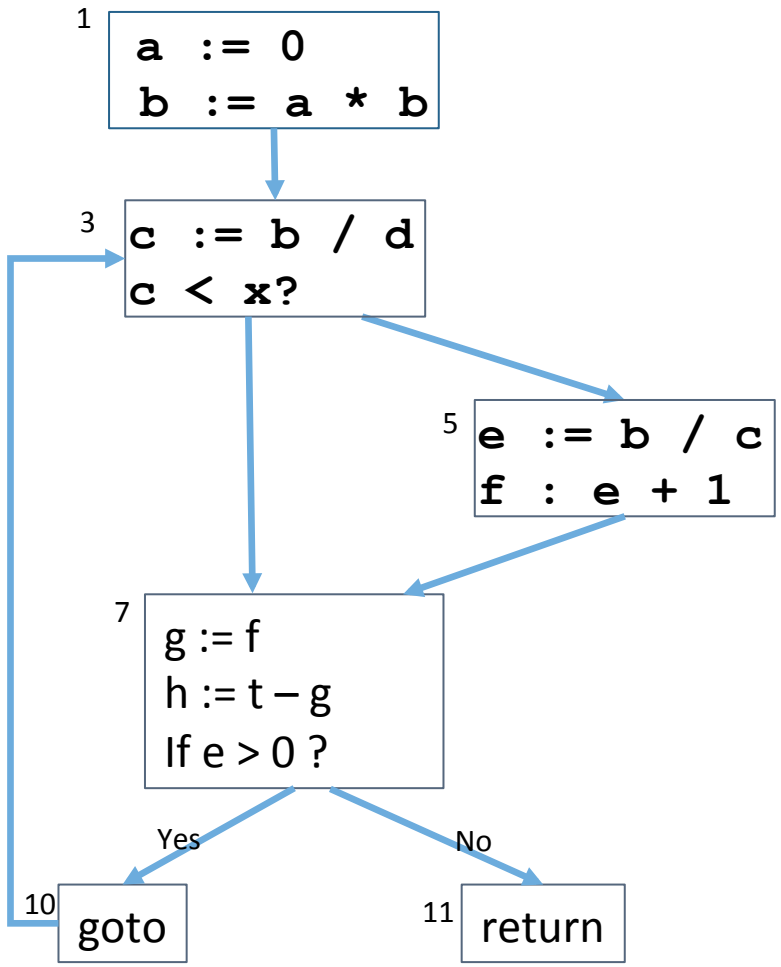
Blocks?

- {1, 2}
- {3, 4}
- {5, 6}
- {7, 8, 9}
- {10}
- {11}

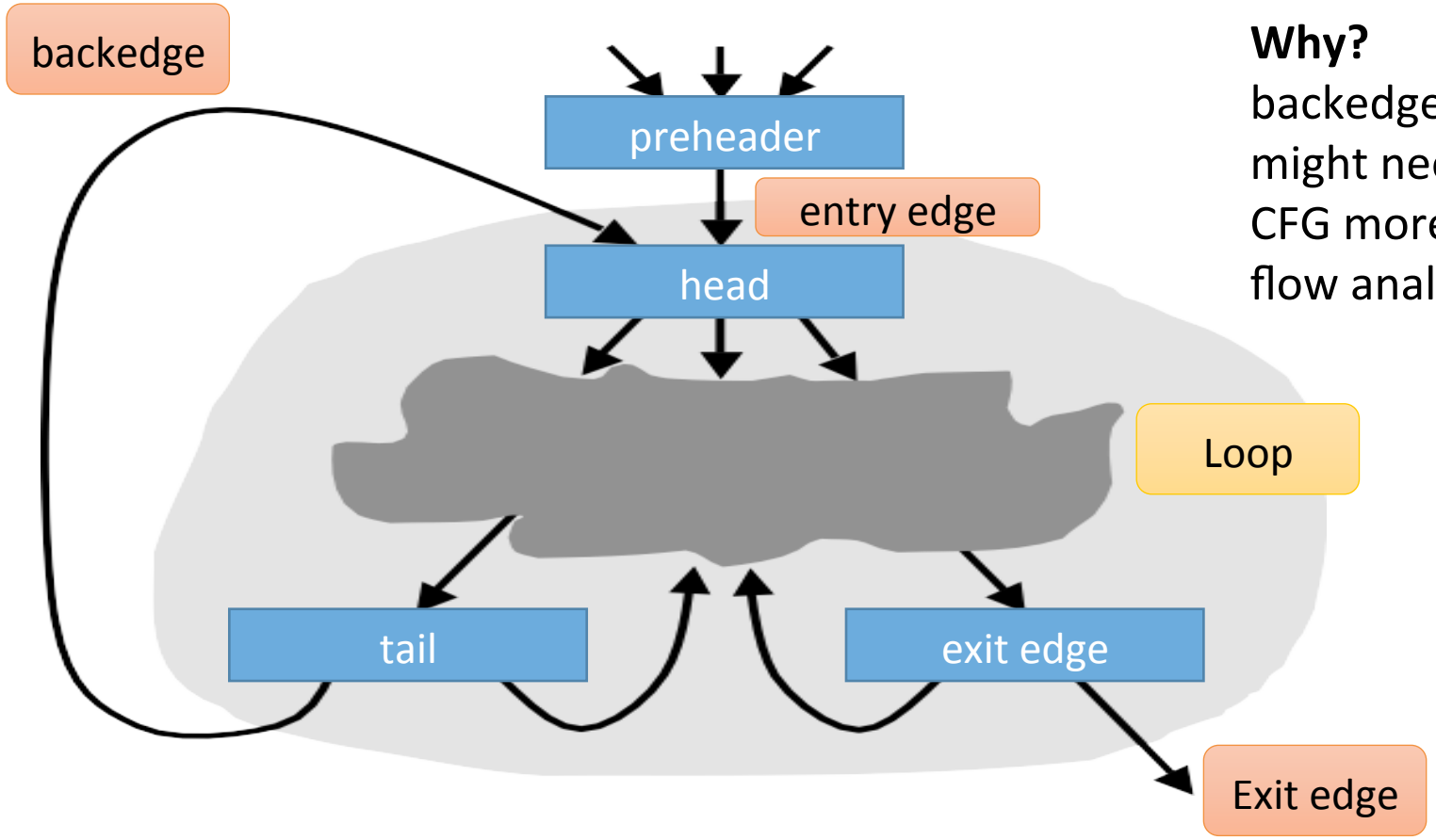
Building a CFG From Basic Block

Construction

- Each CFG node represents a basic block
- There is an edge from node i to j if
  - Last statement of block i branches to the first statement of j, or
  - Block i does **not** end with an unconditional branch and is immediately followed in program order by block j (fall through)

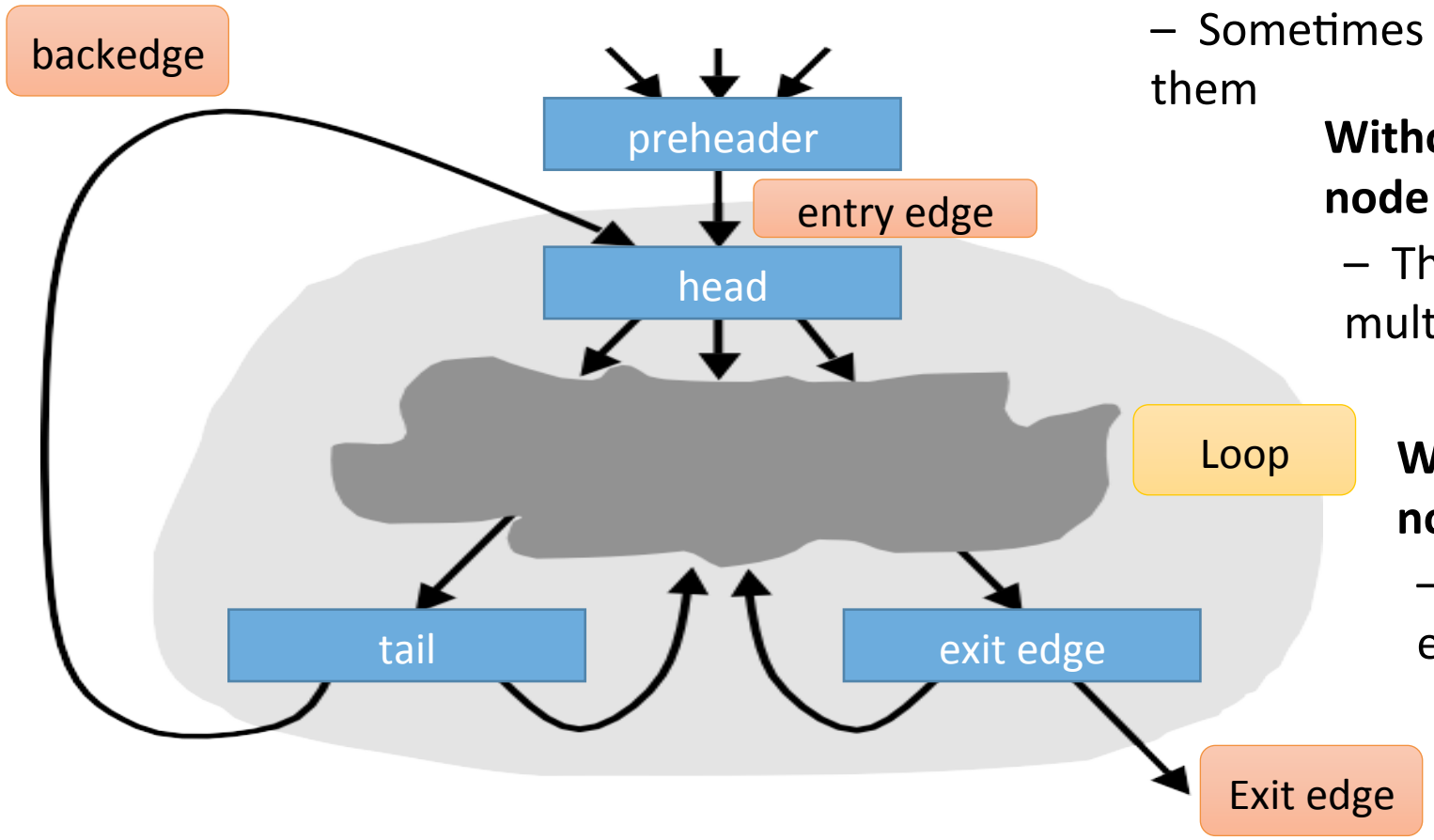


Looping



Why?
backedges indicate that we might need to traverse the CFG more than once for data flow analysis

Looping



Not all loops have preheaders

- Sometimes it is useful to create them

Without preheader node

- There can be multiple entry edges

With single preheader node

- There is only one entry edge

Dominators

- d **dom** i if all paths from entry to node i include d
- Strict Dominator (d sdom i)
  - If d dom i, but d != i
- Immediate dominator (a idom b)
  - a sdom b and there does not exist any node c such that a != c, c != b, a dom c, c dom b
- Post dominator (p pdom i)
  - If every possible path from i to exit includes p

Identifying Natural Loops and Dominators

- Back Edge
  - A **back edge** of a natural loop is one whose target dominates its source

- Natural Loop
  - The **natural loop** of a back edge (m→n), where n dominates m, is the set of nodes x such that n dominates x and there is a path from x to m not containing n

Reducibility

- A CFG is **reducible** (well-structured) if we can partition its edges into two disjoint sets, the **forward edges** and the **back** edges, such that
  - –The forward edges form an acyclic graph in which every node can be reached from the entry node
  - –The back edges consist only of edges whose targets dominate their sources
- **Structured control-flow constructs give rise to reducible CFGs**
  Value of reducibility:
  - –Dominance useful in identifying loops
  - –Simplifies code transformations (every loop has a single header)
  - –Permits interval analysis

Handling Irreducible CFG's

- Node splitting
  - Can turn irreducible CFGs into reducible CFGs

General idea

–Reduce graph (iteratively remove self edges, merge nodes with single pred)

–More than one node => irreducible

– Split any multi-parent node and start over

Why go through all this trouble?

- **Modern languages provide structured control flow**
  - –Shouldn't the compiler remember this information rather than throw it away and then re-compute it?
- **Answers?**
  - –We may want to work on the binary code
  - –Most modern languages still provide a **goto** statement
  - –Languages typically provide multiple types of loops. This analysis lets us treat them all uniformly
  - –We may want a compiler with multiple front ends for multiple languages; rather than translating each language to a CFG, translate each language to a canonical IR and then to a CFG