CS 6410: Compilers

Fall 2023

Tamara Bonaci

t.bonaci@northeastern.edu

Thank you to UW faculty Hal Perkins. Today lecture notes are a modified version of his lecture notes.

Credits For Course Material

- **Big thank you to UW CSE faculty member, Hal Perkins**
- Some direct ancestors of this course:
  - UW CSE 401 (Chambers, Snyder, Notkin, Perkins, Ringenburg, Henry, …)
  - UW CSE PMP 582/501 (Perkins)
  - Cornell CS 412-3 (Teitelbaum, Perkins)
  - Rice CS 412 (Cooper, Kennedy, Torczon)
  - Many books (Appel; Cooper/Torczon; Aho, [[Lam,] Sethi,] Ullman [Dragon Book], Fischer, [Cytron ,] LeBlanc; Muchnick, …)

Agenda

- Review - Static semantics
- Attribute grammars
- Symbol tables
- Types & type checking
- Intermediate representations

Reading:

- Cooper & Torczon – chapter 3 and 5
- The Dragon book, chapters 4 and 6.1, 6.2

Hints about the Project – Part 2
Abstract Syntax Tree

Abstract Syntax Trees in Java

- Idea: capture the essential structure of a program; omit extraneous details

- In Java: use objects as simple tree nodes (basically structures with constructors, and maybe some convenience methods).
- Instance variables:
  - Subtree pointers
  - Source program coordinates (line numbers, …)
  - Later, links to semantic info (symbol tables, types)
  - But not much more!

- Use Java type system and inheritance to factor common AST node information and allow polymorphic treatment of related nodes

MiniJava Starter Code

- AST type hierarchy - root is ASTNode
- Some subclasses:
  - Exp subclasses: And, Plus, Times, True, Call, …
  - Statement subclasses: While, Assign, If, Print, …
  - Type subclasses: abstract representations of types, *not* source code type declarations – more about this when we get to semantics)
  - Declarations, Classes, others parts of abstract grammar, …
- Additional information in all AST nodes
  - Source code position info (hooks in starter JFlex and CUP rules to capture this, use in error messages, AST printout)
  - Accept methods for visitors
- Not required to use this AST, but it is *strongly* advised

AST Generation

- Idea: each time the parser recognizes a complete production, it produces as its result an AST node (with links to the subtrees that are the components of the production)
- When we finish parsing, the result is the complete AST for the program

Example: Recursive-Descent AST Generation

```
// parse while (exp) stmt
WhileNode whileStmt() {
    // skip "while ("
    skipToken(WHILE);
    skipToken(LPAREN);

    // parse exp
    ExpNode cond = exp();

(continued next col.)
```

```
    // skip ")"
    skipToken(RPAREN);

    // parse stmt
    StmtNode body = stmt();

    // return AST node for while
    return  new WhileNode (cond,
    body);
}
```

AST Generation in YACC/CUP

- A result type can be specified for each item in the grammar specification
- Each parser rule can be annotated with a semantic action, which is just a piece of Java code that returns a value of the result type
- The semantic action is executed when the rule is reduced

YACC/CUP Parser Specification

- CUP code

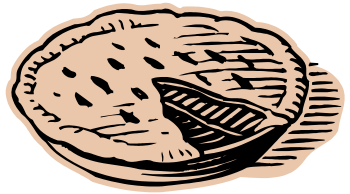
```
non terminal StmtNode stmt, whileStmt;
non terminal ExpNode exp;
…
stmt ::= …
        | WHILE LPAREN exp:e RPAREN stmt:s
          {:  RESULT = new WhileNode(e,s);  :}
        ;
```

  - See the starter code for version with line numbers

Operations on ASTs

- Once we have the AST, we may want to:
  - Print a readable dump of the tree (pretty printing)
  - Do static semantic analysis:
    - Type checking
    - Verify that things are declared and initialized properly
    - Etc. etc. etc. etc.
  - Perform optimizing transformations on the tree
  - Generate code from the tree
  - Generate another IR from the tree for further processing

Modularity



- Classic slogans:
  - Do one thing well
  - Minimize coupling, maximize cohesion
  - Isolate operations/abstractions in modules
  - Hide implementation details

- Okay, so where does the MiniJava typechecker module belong in the code?

Where do the Operations Go?

- Pure "object-oriented" style
  - Really, really, really smart AST nodes
  - Each node knows how to perform every operation on itself

```
public class WhileNode extends StmtNode {
  public WhileNode(…);
  public typeCheck(…);
  public StrengthReductionOptimize(…);
  public DeadCodeEliminationOptimize(…);
  public generateCode(…);
  public prettyPrint(…);
  …
}
```

Critique

- This is nicely encapsulated – all details about a WhileNode are hidden in that class
- But it is poor modularity
- What happens if we want to add a new optimization (or any other) operation?
  - Have to modify every node class ☹
- Worse: the details of any particular operation (optimization, type checking) are scattered across the node classes

Modularity Issues

- Smart nodes make sense if the set of operations is relatively fixed, but we expect to need flexibility to add new kinds of nodes
- Example: graphics system
  - Operations: draw, move, iconify, highlight
  - Objects: textbox, scrollbar, canvas, menu, dialog box, window, plus new objects defined as the system evolves

Modularity in a Compiler

- Abstract syntax does not change frequently over time

  ∴ Kinds of nodes are relatively fixed

- As a compiler evolves, it is common to modify or add operations on the AST nodes
  - Want to modularize each operation (type check, optimize, code gen) so its parts are together
  - Want to avoid having to change node classes when we modify or add an operation on the tree

Visitor Pattern

Visitor Pattern

- Idea: Package each operation (optimization, print, code gen, …) in a separate visitor class
- Create exactly one instance of each visitor class
  - Sometimes called a "function object"
  - Contains all of the methods for that particular operation, one for each kind of AST node
- Include a generic "accept visitor" method in every node class
- To perform an operation, pass the appropriate "visitor object" around the AST during a traversal

Avoiding instanceof

- We'd like to avoid huge if-elseif nests in the visitor to discover the node types

```
void checkTypes(ASTNode p) {
    if (p instanceof WhileNode) { … }
    else if (p instanceof IfNode) { … }
    else if (p instanceof BinExp) { … }
…
}
```

Visitor Double Dispatch

- Include a "visit" method for every AST node type in each Visitor

```
void visit(WhileNode);
void visit(ExpNode);
etc.
```

- Include an accept(Visitor v) method in each AST node class
- When Visitor v is passed to an AST node, the node's accept method calls v.visit(this)
  - Selects correct Visitor method for this node
  - "Double dispatch"

Visitor Interface

```
interface Visitor {
    // overload visit for each AST node type
    public void visit(WhileNode s);
    public void visit(IfNode s);
    public void visit(BinExp e);
    …
}
```

–Every separate Visitor implements this interface

–Aside: The result type can be whatever is convenient, doesn't have to be void, although that is common

Accept Method in Each AST Node Class

- Every AST class overrides accept(Visitor)
- Example

```
public class WhileNode extends StmtNode {
    …
    // accept a visit from a Visitor object v
    public void accept(Visitor v) {
      v.visit(this);   // dynamic dispatch on “this” (WhileNode)
    }
    …
}
```

- Key points
  - –Visitor object passed as a parameter to WhileNode
  - –WhileNode calls visitor’s visit method, which dispatches to visit(WhileNode) automatically – i.e., the correct method in the visitor object for this kind of node

Composite Objects (1)

- How do we handle composite objects?
- One possibility: the accept method passes the visitor down to subtrees before (or after) visiting itself

```
public class WhileNode extends StmtNode {
  Expr exp; Stmt stmt;   // children
  …
  // accept a visit from visitor v
  public void accept (Visitor v) {
     this.exp.accept(v);
     this.stmt.accept(v);
     v.visit(this);
  }
```

Composite Objects (2)

- Another possibility: the visitor can control the traversal

```
public void visit(WhileNode p) {
    p.expr.accept(this);
    p.stmt.accept(this);
}
```

Encapsulation

- A visitor object often needs to be able to access state in the AST nodes
  - ∴ May need to expose more node state than we might have done otherwise
    - i.e., lots of public fields in AST node objects
  - – Overall a good tradeoff – better modularity

    (plus, the nodes are relatively simple data objects anyway – not hiding much of anything)

Visitor Actions and State

- A visitor function has a reference to the node it is visiting (the parameter)

  ∴ can access and manipulate subtrees directly

- Visitor object can also include local data (state) shared by methods in the visitor
  - This data is effectively "global" to the methods that make up the visitor object, and can be used to store and pass around information as different kinds of nodes are visited

```
public class TypeCheckVisitor extends NodeVisitor {
    public void visit(WhileNode s) { … }
    public void visit(IfNode s) { … }
    …
    private <local state>;  // all methods can read/write this
}
```

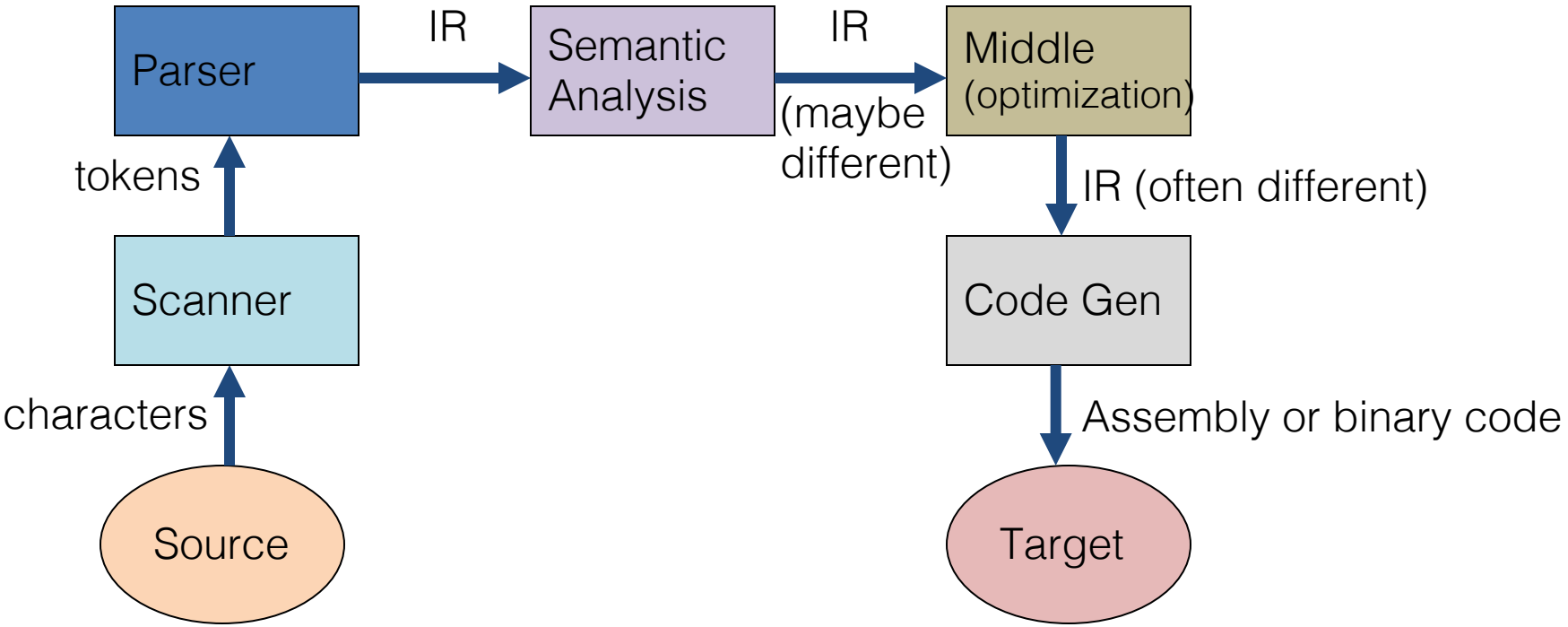
So which to choose?

- Possibilities:
  - Node objects drive the traversal and pass the visitors around the tree in standard ways
  - Visitor object drives the traversal (the visitor has access to the node, including references to child subtrees)
- In a compiler:
  - First choice handles many common cases
  - Big compilers often have multiple visitor schemes (e.g., several different traversals defined in Node interface plus custom traversals in some visitors)
  - For MiniJava: keep it simple and start with supplied examples, but if you really need to do something different, you can
    - (i.e., keep an open mind, but not so open that you create needless complexity)

References

- For Visitor pattern (and many others)
  - *Design Patterns: Elements of Reusable Object-Oriented Software*, Gamma, Helm, Johnson, and Vlissides, Addison-Wesley, 1995 (the classic; examples are in old C++ and Smalltalk)
  - James Maioriello, Survey of Common Design Patterns, online: https://www.developer.com/design/article.php/1502691/A-Survey-of-Common-Design-Patterns.htm
- Specific information for MiniJava AST and visitors in Appel textbook & online

Review: Compiler Structure



Intermediate Representations

- In most compilers:
  - The parser builds an intermediate representation (IR, typically an Abstract Syntax Tree)
  - Rest of the compiler transforms the IR to optimize it
  - Typically, will transform initial IR to one or more different IRs along the way
  - IR eventually translate to final target code

IR Design

- Decisions affect speed and efficiency of the rest of the compiler
  - General rule: compile time is important, but performance of generated code 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 compile time as long as they don't compromise the result

IR Design

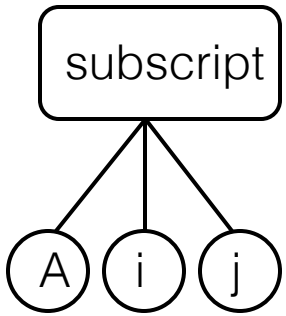
- Desirable properties
  - Easy to generate
  - Easy to manipulate
  - Expressive
  - Appropriate level of abstraction
- Different tradeoffs depending on compiler goals
- Different tradeoffs in different parts of the same compiler
  - So often different IRs in different parts

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

source: A[i,j]



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
  - Linear IRs are typically low-level

Graphical IRs

- IR represented as a graph (or a 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,
  - 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 syntax tree (parse tree) does not need to be used explicitly, but sometimes we want it (structured source code editors or transformations, …)

Example

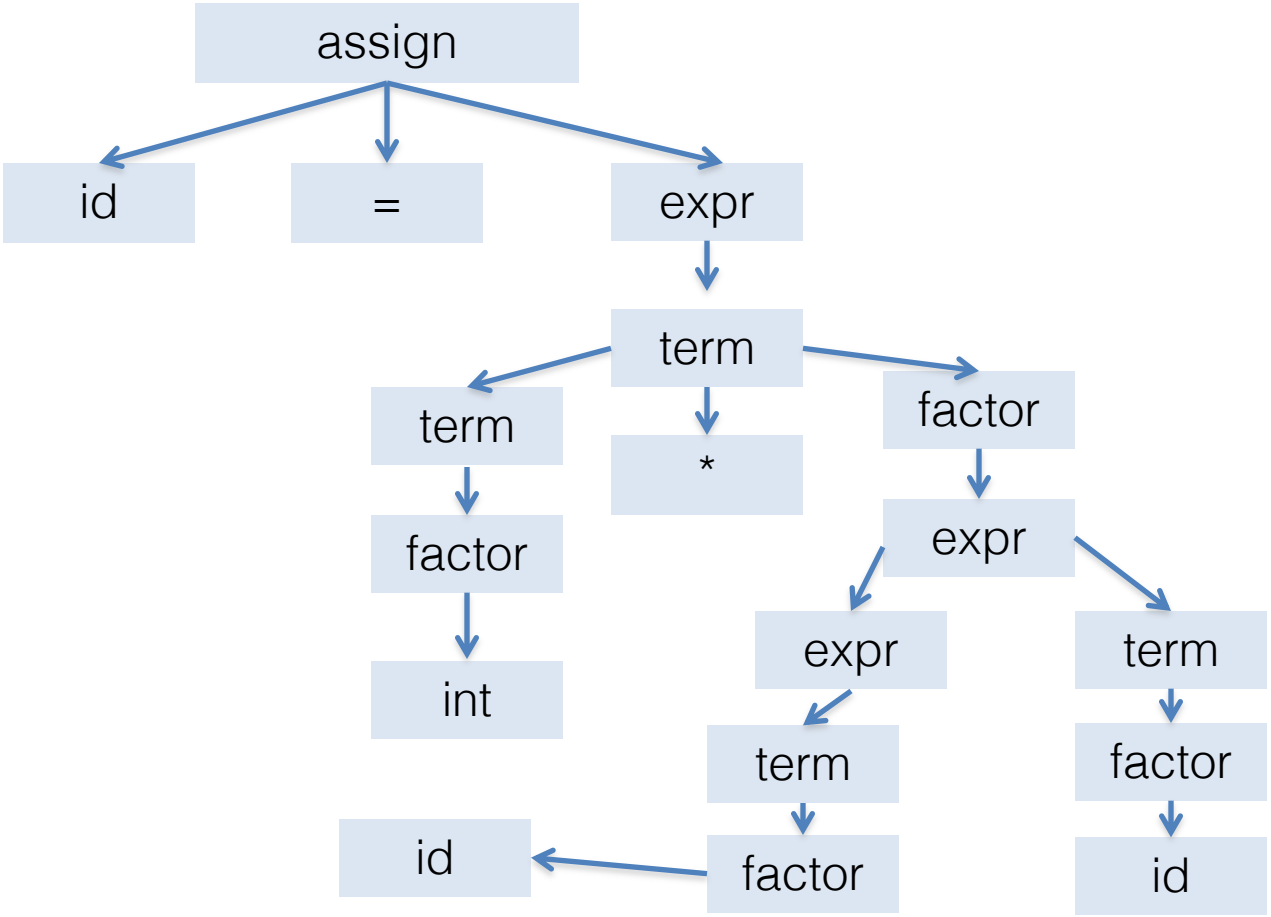
- Concrete syntax for x = 2*(n+m)

assign ::= *id* = *expr* ;
expr ::= *expr* + *term* | *expr* – *term* | *term*
term ::= *term* * *factor* | *term* / *factor* | *factor*
factor ::= *int* | *id* | (*expr*)

assign ::= *id* = *expr* ;
expr ::= *expr* + *term* | *expr* – *term* | *term*
term ::= *term* * *factor* | *term* / *factor* | *factor*
factor ::= *int* | *id* | (*expr*)

Example

- Concrete syntax for x = 2*(n+m)



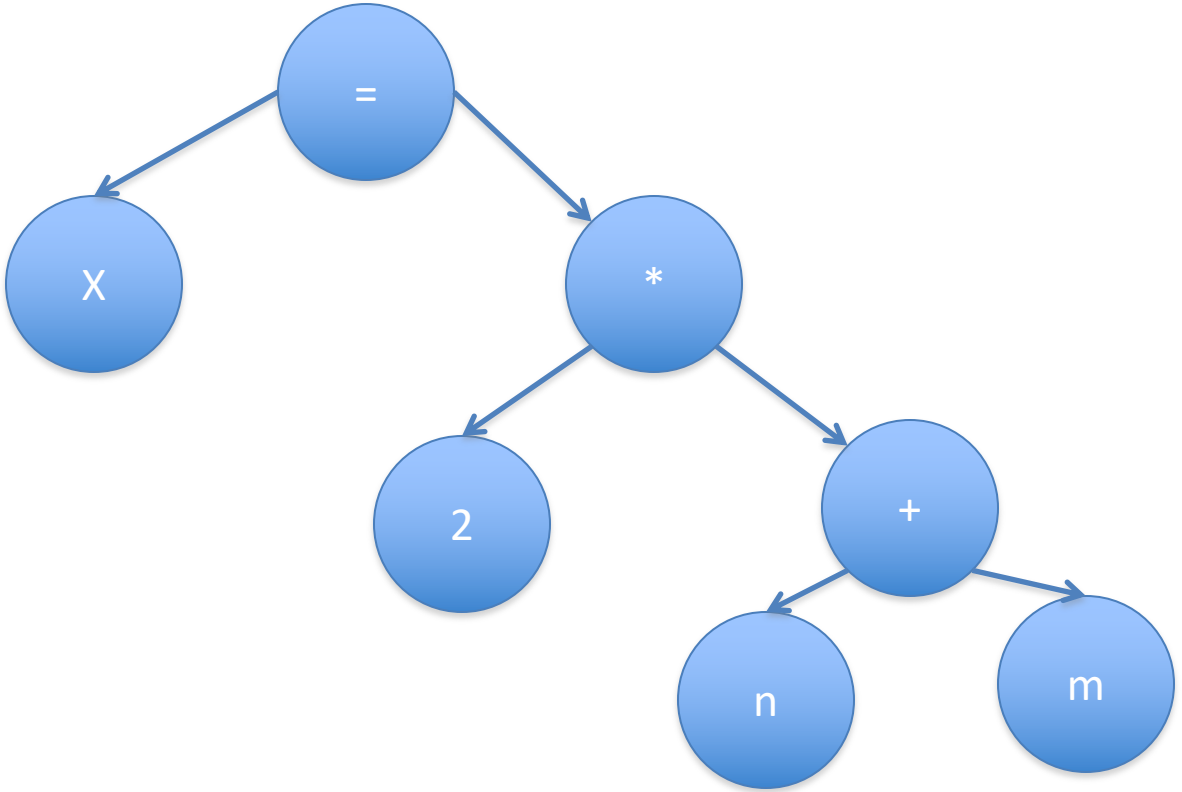
Abstract Syntax Trees

- Common output from parser:
  - Used for static semantics (type checking, etc)
  - Sometimes used high-level optimizations
- Focus on essential structural information
- Can be represented:
  - Explicitly as a tree or
  - In a linear form
- Example: LISP/Scheme S-expressions are essentially ASTs

assign ::= *id* = *expr* ;
expr ::= *expr* + *term* | *expr* – *term* | *term*
term ::= *term* * *factor* | *term* / *factor* | *factor*
factor ::= *int* | *id* | (*expr*)

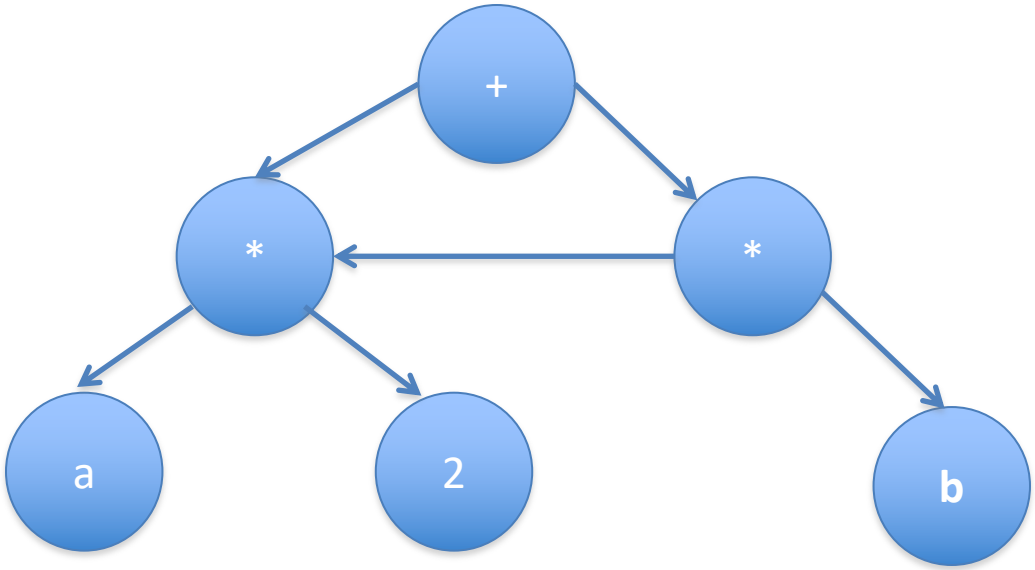
Example

- Abstract syntax for x = 2*(n+m)



DAGs (Directed Acyclic Graphs)

- Variation on ASTs with shared substructures
- Pro: saves space, exposes redundant sub-expressions
- Con: less flexibility if part needs to be changed



Linear IRs

- Pseudo-code for some abstract machine
- Level of abstraction varies
- Simple, compact data structures
  - Commonly used: arrays, linked lists
- 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 be:
  - High-level abstraction (close to the source language)
  - Medium-level abstraction
  - Very low-level abstraction
- Examples: Linear IRs for C array reference a[i][j+2]
  - High-level: t1 ← 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

- 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]

Abstraction Level Tradeoffs

- High-level abstraction:
  - Good for some source-level optimizations and semantic checking
  - Can't optimize things that are hidden – like address arithmetic for array subscripting
- Medium-level abstraction:
  - More details, but keeps more higher-level semantic information – great for machine-independent optimizations
  - Many (all?) optimizing compilers work at this level
- Low-level abstraction:
  - Need for good code generation and resource utilization in back end
  - Loses semantic knowledge (e.g., variables, data aggregates, source relationships are usually missing)
- 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
- "Expression temps":
  - Invent as many new temp names as needed
  - Don't correspond to any user variables; de-anonymize expressions
- Store in a quad(ruple)
  - <lhs, rhs1, op, rhs2>

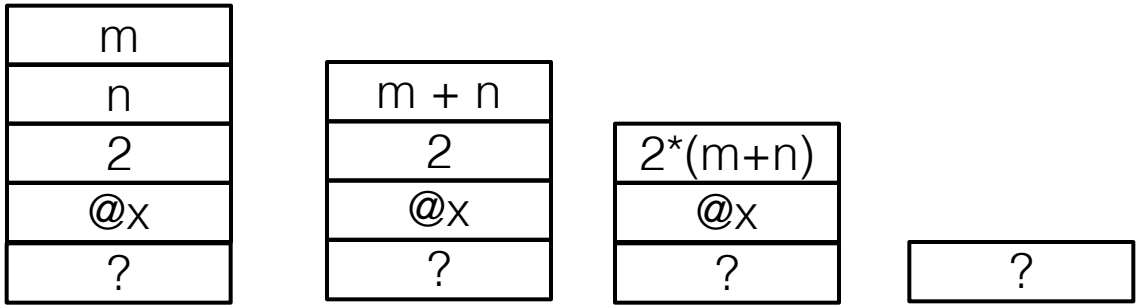
Three Address Code

- 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)
- Now also used for virtual machines:
  - UCSD 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 network)
  - Easy to generate; easy to write a FrontEnd compiler, leaving the 'heavy lifting' and optimizations to the JIT
  - Simple to interpret or compile to machine code
- Disadvantages
  - 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)

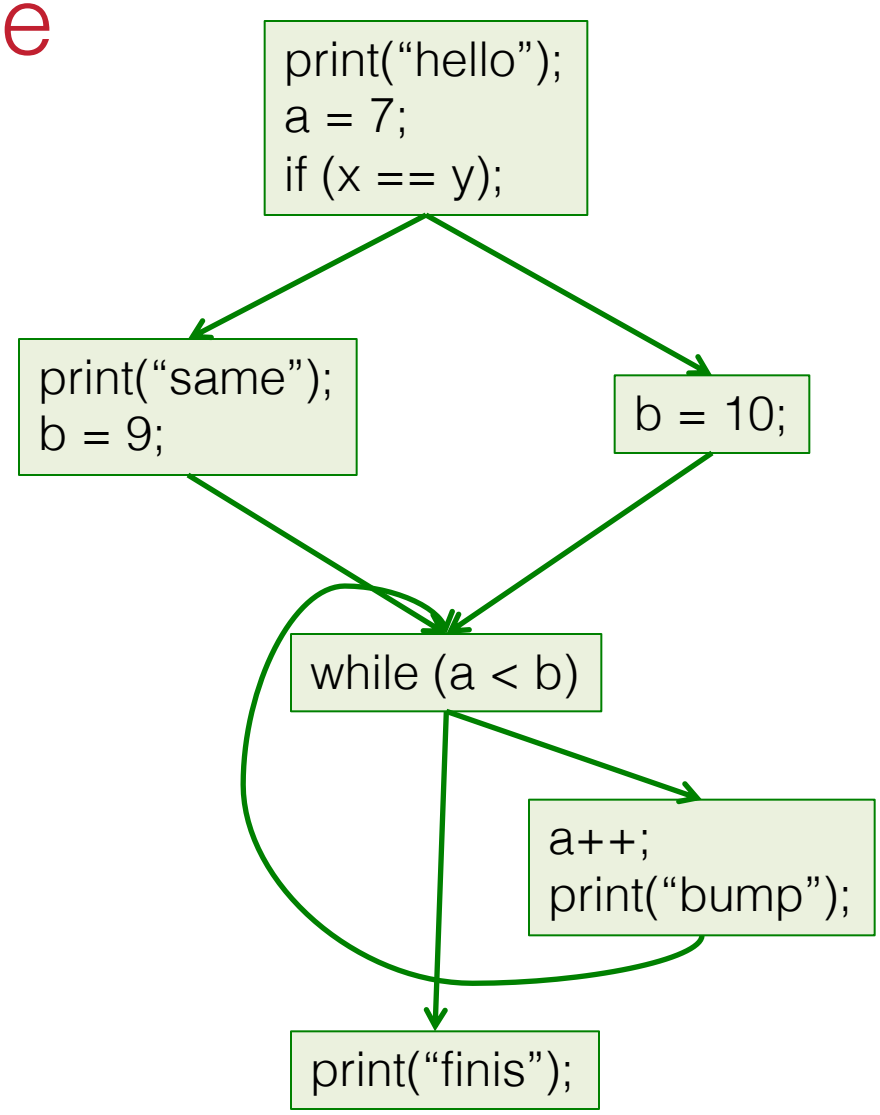
- Nodes: basic blocks
- Edges: possible flow of control from one block to another, (i.e., possible execution orderings)
  - Edge from A to B if B could execute immediately after A in some possible execution
- Required for much of the analysis done during optimization phases

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 ("straight-line code") – so it can be treated as an indivisible block
    - 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                          10 i = i + 1
2 j = 1                          11 if i <= 10 goto #2
3 t1 = 10 * i                    12 i = 1
4 t2 = t1 + j                    13 t5 = i - 1
5 t3 = 8 * t2                    14 t6 = 88 * t5
6 t4 = t3 - 88                   15 a[t6] = 1
7 a[t4] = 0                      16 i = i + 1
8 j = j + 1                      17 if i <= 10 goto #13
9 if j <= 10 goto #3
```

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

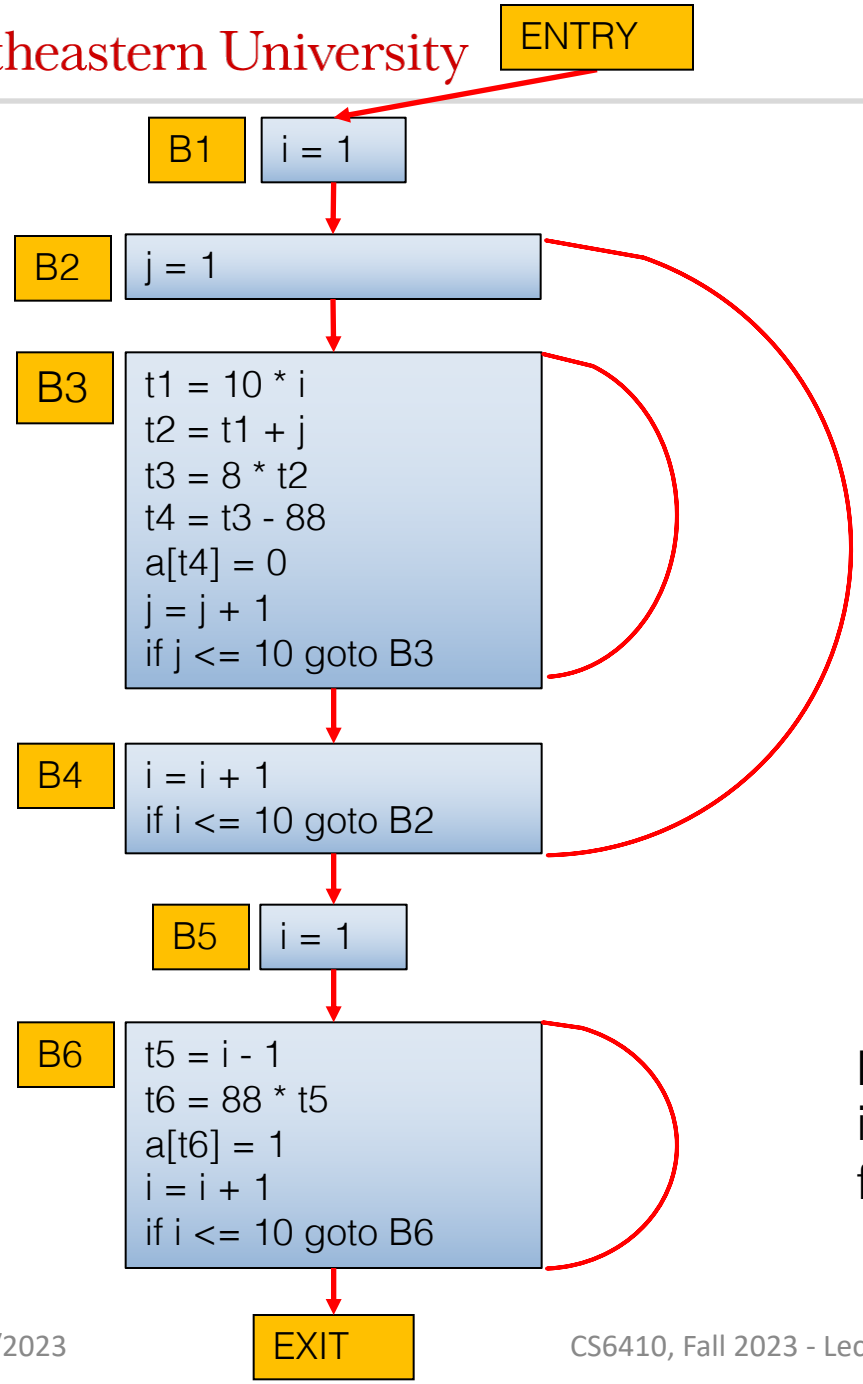
```
1 i = 1
2 j = 1
3 t1 = 10 * i
4 t2 = t1 + j
5 t3 = 8 * t2
6 t4 = t3 - 88
7 a[t4] = 0
8 j = j + 1
9 if j <= 10 goto #3
10 i = i + 1
11 if i <= 10 goto #2
12 i = 1
13 t5 = i - 1
14 t6 = 88 * t5
15 a[t6] = 1
16 i = i + 1
17 if i <= 10 goto #13
```

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 executions 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

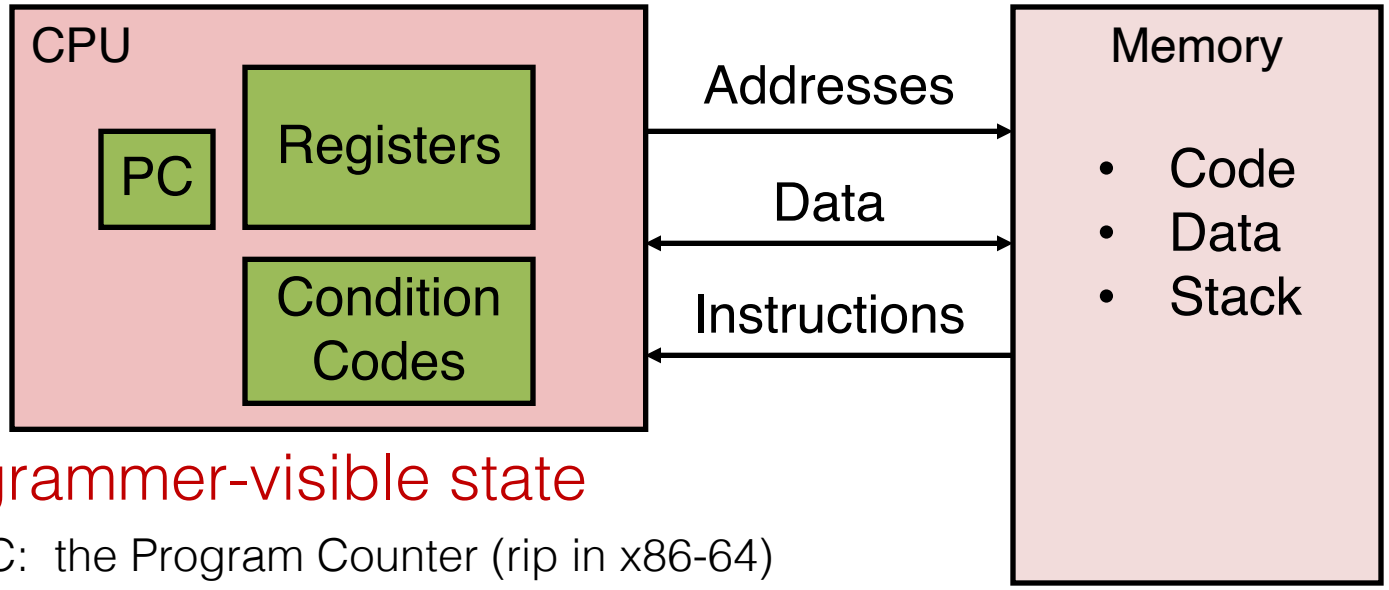
- Often used in conjunction with another IR
- Data dependency: edges between nodes that reference common data
- Examples
  - RAW – read after write: Block A defines x then B reads it
  - WAR – "anti-dependence": Block A reads x then B writes it
  - WAW: Blocks A and B both write x – order of blocks must 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 for optimization and codegen
    - Closer to machine code
    - Use to build control-flow graph
    - Exposes machine-related optimizations
  - Hybrid (graph + linear IR = CFG) for dataflow & opt

X86-64 Overview/Review

Assembly Programmer's View



- Programmer-visible state
  - PC: the Program Counter (rip in x86-64)
    - Address of next instruction
  - Named registers
    - Together in "register file"
    - Heavily used program data
  - Condition codes
    - Store status information about most recent arithmetic operation
    - Used for conditional branching
- Memory
  - Byte-addressable array
  - Code and user data
  - Includes the Stack (for supporting procedures)

What is a Register?

- A location in the CPU that stores a small amount of data, which can be accessed very quickly (once every clock cycle)
- Registers have names, not addresses
- Registers are at the heart of assembly programming
  - They are a precious commodity in all architectures, but especially x86

x86-64 Main features

- 16 64-bit general registers + 64-bit integers (but int is 32 bits usually; long is 64 bits)
- 64-bit address space; pointers are 8 bytes
- 16 SSE registers for floating point, SIMD
- Register-based function call conventions
- Additional addressing modes (pc relative)
- 32-bit legacy mode
- Some pruning of old features

Some x86-64 References

(Links on course web - * = most useful)

- **x86-64 Instructions and ABI
  - Handout for University of Chicago CMSC 22620, Spring 2009, by John Reppy
- *x86-64 Machine-Level Programming
  - Earlier version of sec. 3.13 of Computer Systems: A Programmer's Perspective, 2nd ed. by Bryant & O'Hallaron (CSE 351 textbook)
- Intel architecture processor manuals
- www.x86-64.org:
  - System V Application Binary Interface
  - Gentle Introduction to x86-64 Assembly

x86 Selected History

- Almost 40 Years of x86
  - 1978: 8086 16-bit, 5 MHz, 3μ, segmented
  - 1982: 80286 protected mode, floating point
  - 1985: 80386 32-bit, VM, 8 "general" registers
  - 1993: Pentium MMX
  - 1999: Pentium III SSE
  - 2000: Pentium IV SSE2, SSE3, HyperThreading
  - 2006: Core Duo, Core2 Multicore, SSE4, x86-64
  - 2013: Haswell 64-bit, 4-8 core, ~3 GHz, 22 nm, AVX2
- Many micro-architecture changes over the years:
  - pipelining, super-scalar, out-of-order, caching, multicore

And It's Backward-Compatible!

- Current processors can run 8086 code
  - You can get VisiCalc 1.0 on the web & run it ☺
- Intel architecture descriptions are engulfed with modes and flags, but the modern processor is fairly straightforward
- Modern processors have a RISC-like core
  - Load/Store from memory
  - Register-register operations
- We will focus on basic 64-bit instructions
  - Simple instructions preferred;
  - Complex ones exist for backward-compatibility but can be slow

x86-64 Assembler Language

- Target for our compiler project
- But, the nice thing about standards…
- Two main assembler languages for x86-64 exist
  - Intel/Microsoft version – what's in the Intel docs
  - AT&T/GNU assembler – what we're generating and what's in the linked handouts
    - Use gcc –S to generate asm code from C/C++ code
- Slides use gcc/AT&T/GNU syntax

x86-64 Assembly "Data Types"

- Integral data of 1, 2, 4, or 8 bytes
  - Data values
  - Addresses
- Floating point data of 4, 8, 10 or 2x8 or 4x4 or 8x2
  - Different registers for those (e.g. xmm1, ymm2)
  - Come from extensions to x86 (SSE, AVX, …)
- No aggregate types such as arrays or structures
  - Just contiguously allocated bytes in memory
- Two common syntaxes
  - "AT&T": used by our course, slides, textbook, gnu tools, …
  - "Intel": used by Intel documentation, Intel tools, …
  - Must know which you're reading

Intel vs. GNU Assembler

- Main differences between Intel docs and gcc assembler

	Intel/Microsoft	AT&T/GNU as
Operand order: op a,b	a = a op b (dst first)	b = b op a (dst last)
Memory address	[baseregister+offset]	offset(baseregister)
Instruction mnemonics	mov, add, push, …	movq, addq, pushq [operand size is added to end]
Register names	rax, rbx, rbp, rsp, …	%rax, %rbx, %rbp, %rsp, …
Constants	17, 42	$17, $42
Comments	; to end of line	# to end of line or /* … */

- Intel docs also include many complex, historical instructions and artifacts not commonly used by modern compilers – we won't use them either

x86-64 Memory Model

- 8-bit bytes, byte addressable
- 16-, 32-, 64-bit words, double words and quad words (Intel terminology)
  - That's why the 'q' in 64-bit instructions like movq, addq, etc.
- Data should normally be aligned on "natural" boundaries for performance, although unaligned accesses are generally supported – but with a big performance penalty on many machines
- Little-endian – address of a multi-byte integer is address of low-order byte

x86-64 registers

- 16 64-bit general registers
  - %rax, %rbx, %rcx, %rdx, %rsi, %rdi, %rbp, %rsp, %r8-%r15
- Registers can be used as 64-bit integers or pointers, or 32-bit ints
  - Also possible to reference low-order 16- and 8-bit chunks – we won't for the most part
- To simplify our project we'll use only 64-bit data (ints, pointers, even booleans!)

Processor Fetch-Execute Cycle

- Basic cycle (same for every processor you've ever seen)

```
while (running) {
    fetch instruction beginning at rip address
    rip ← rip + instruction length
    execute instruction
}
```

- Sequential execution unless a jump stores a new "next instruction" address in rip

Instruction Format

- Typical data manipulation instruction

```
label:    opcode   src,dst      # comment
```

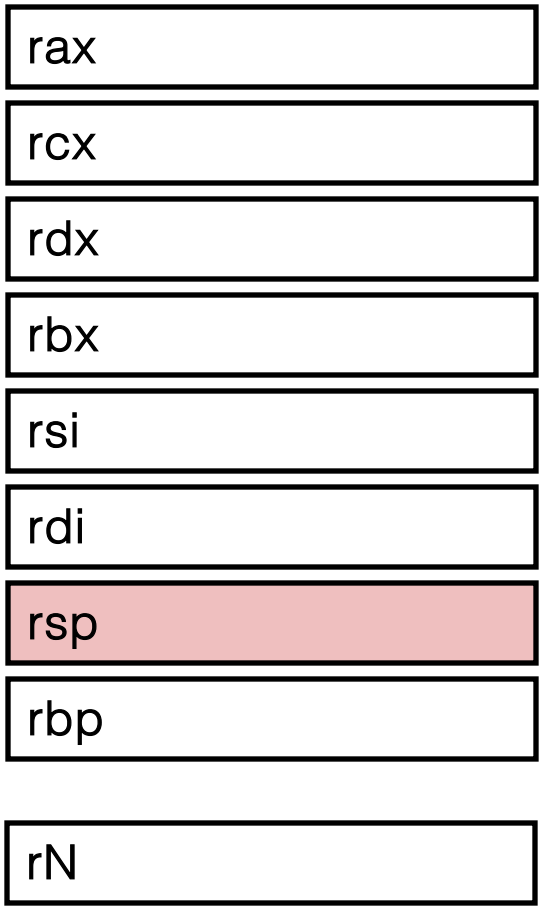
- Meaning is

  dst ← dst op src

- Normally, one operand is a register, the other is a register, memory location, or integer constant
  - Can't have both operands in memory – can't encode two memory addresses in a single instruction (e.g., cmp, mov)
- Language is free-form, comments and labels may appear on lines by themselves (and can have multiple labels per line of code)

Operand types

- Immediate: Constant integer data
  - Examples: `0x400, -533`
  - Encoded with 1, 2, 4, or 8 bytes depending on the instruction
- Register: 1 of 16 integer registers
  - Examples: `rax, r13`
  - But `rsp,rbp` reserved for special use
  - Others have special uses for particular instructions
- Memory: Consecutive bytes of memory at a computed address
  - Simplest example: `DWORD PTR[rax]`
  - Various other "address modes"



Three Basic Kinds of Instructions

1) Transfer data between memory and register
   - Load data from memory into register
     - reg = Mem[address]
   - Store register data into memory
     - Mem[address] = reg

Remember: Memory is indexed just like an array of bytes!

2) Perform arithmetic operation on register or memory data
   - c = a + b; z = x << y; i = h & g;

3) Control flow: what instruction to execute next
   - Unconditional jumps to/from procedures
   - Conditional branches

x86-64 - Instructions Overview

- Data transfer instruction (`mov`)
- Arithmetic operations
- Memory addressing modes
- Address computation instruction (`lea`)

x86-64 Memory Stack

- Register %rsp points to the "top" of stack
  - Dedicated for this use; don't use otherwise
  - Points to the last 64-bit quadword pushed onto the stack (not next "free" quadword)
  - Should always be quadword (8-byte) aligned
    - It will start out this way, and will stay aligned unless your code does something bad
    - Software generally requires 16-byte alignment when function is called
  - Stack grows down

Stack Instructions

pushq src

```
%rsp ⟵ %rsp - 8;
memory[%rsp] ⟵ src
```

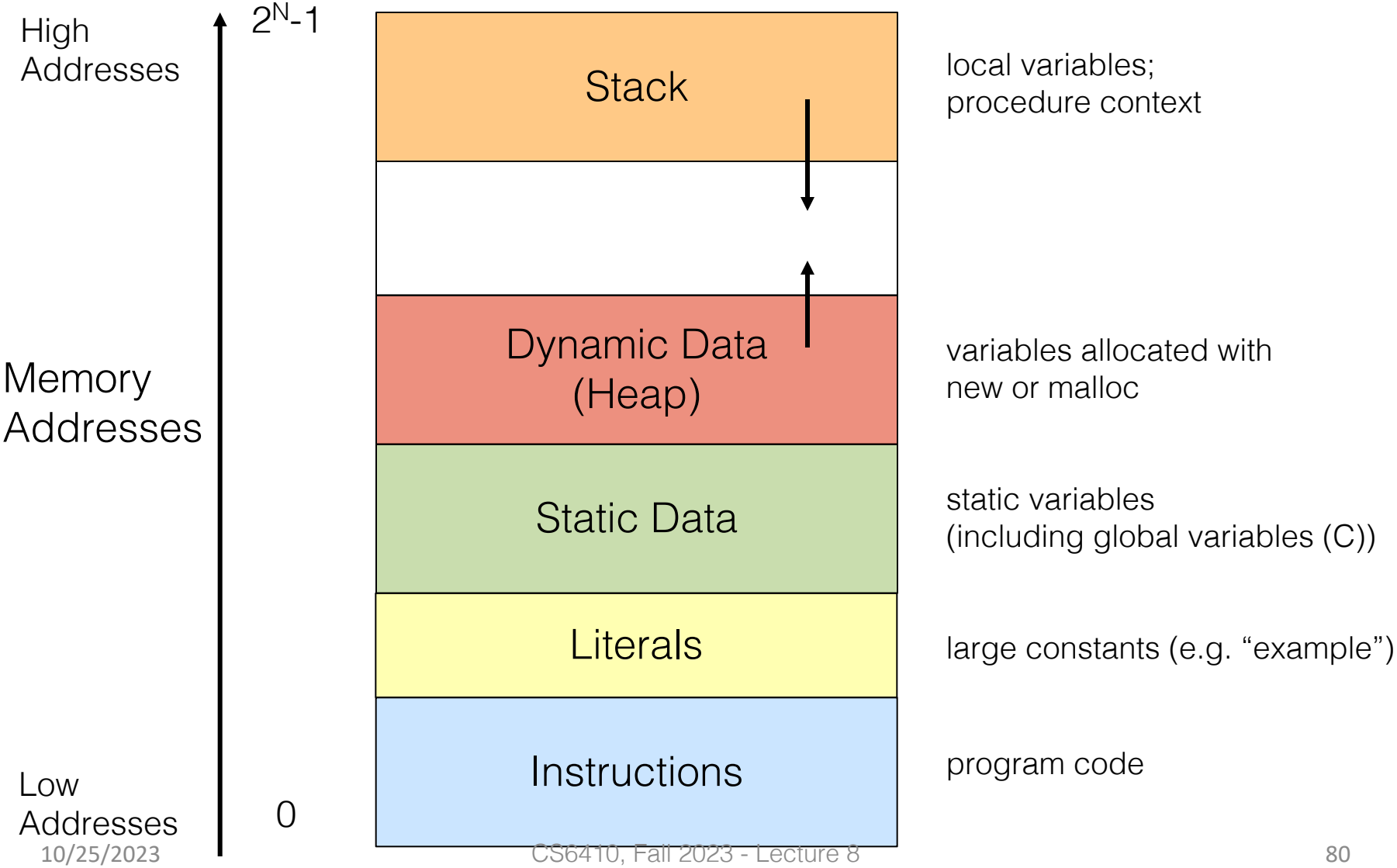
(e.g., push src onto the stack)

popq dst

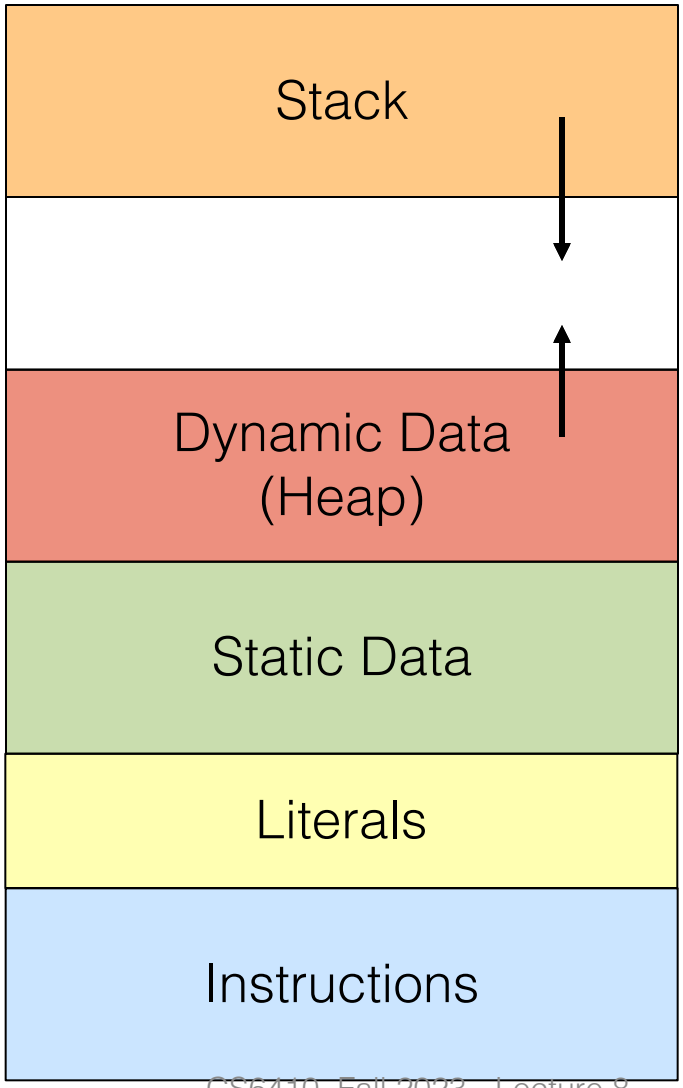
```
dst ⟵ memory[%rsp];
%rsp ⟵ %rsp + 8
```

(e.g., pop top of stack into dst and logically remove it from the stack)

Simplified Memory Layout



Memory Permissions



writable; not executable

Managed "automatically" (by compiler)

writable; not executable

Managed by programmer

writable; not executable

Initialized when process starts

read-only; not executable

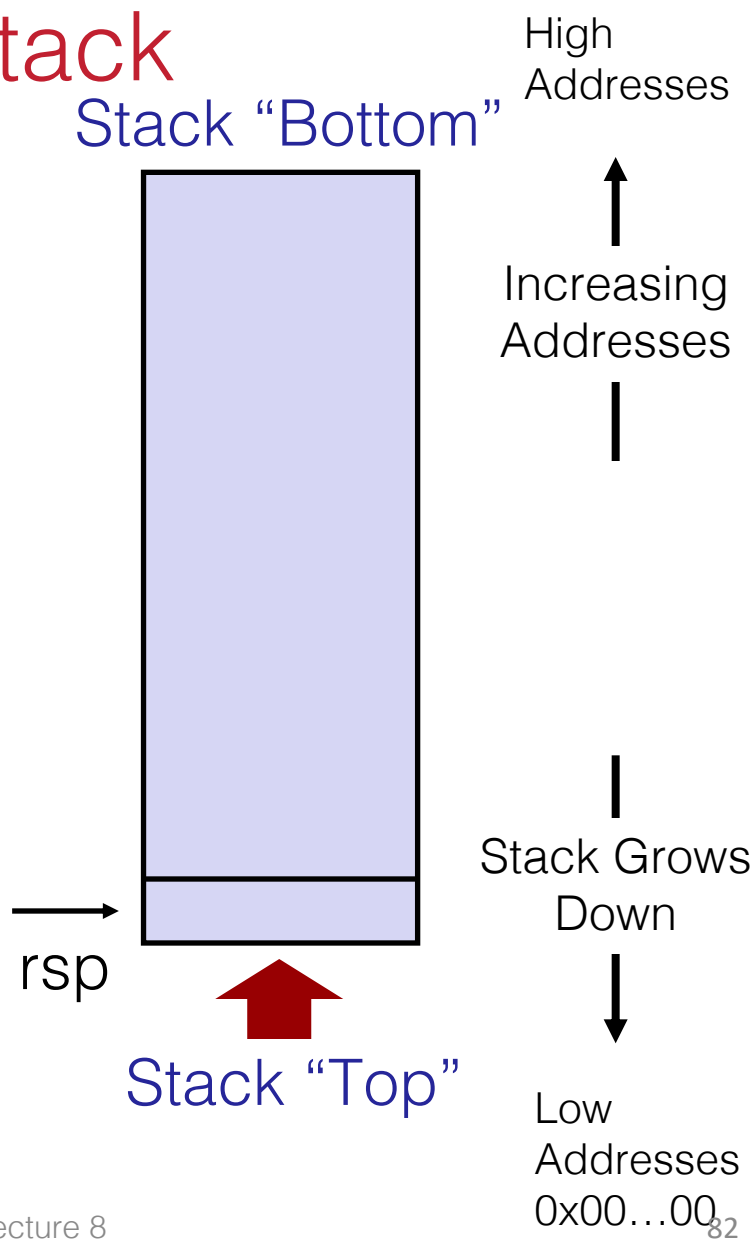
Initialized when process starts

read-only; executable

Initialized when process starts

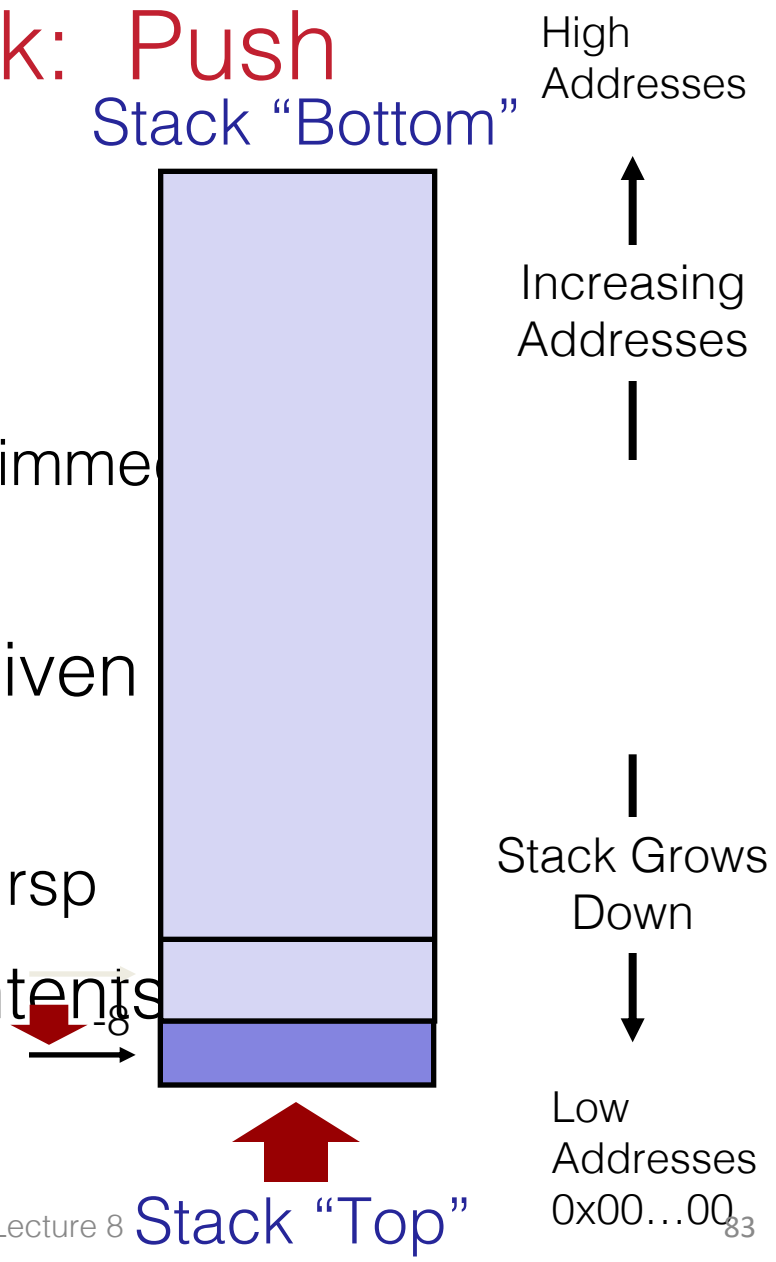
x86-64 Stack

- Region of memory managed with stack "discipline"
  - Grows toward lower addresses
  - Customarily shown "upside-down"
- Register `rsp` contains lowest stack address
  - `rsp` = address of top element, the most-recently-pushed item that is not-yet-popped



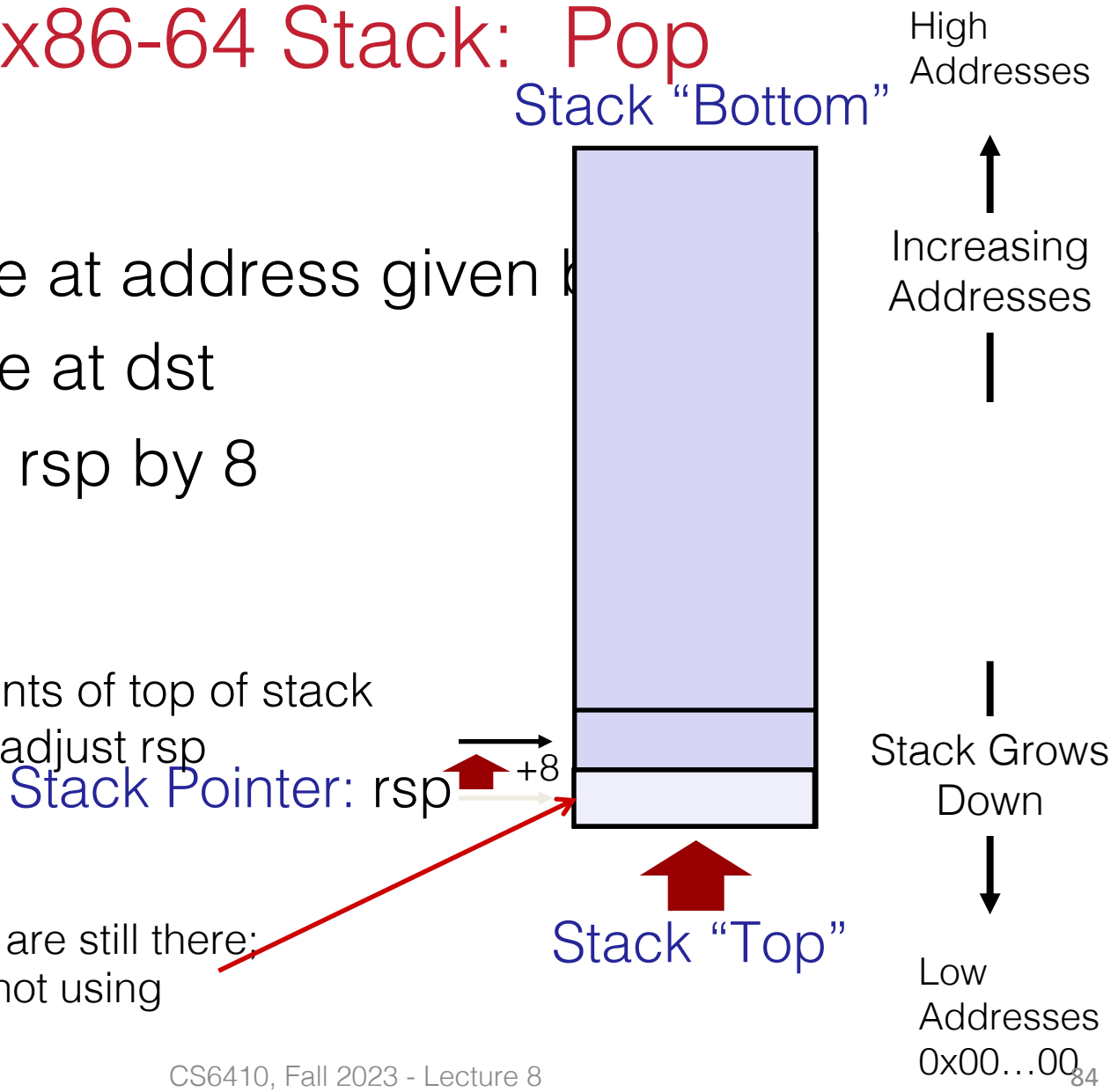
x86-64 Stack: Push

- push src
  - Fetch operand at src
    - Src can be reg, memory, imme
  - Decrement rsp by 8
  - Store value at address given
- Example:
  - push rcx
  - Adjust rsp and store contents of rcx on the stack



x86-64 Stack: Pop

- pop dst
  - Load value at address given b
  - Store value at dst
  - Increment rsp by 8
- Example:
  - pop rcx
  - Stores contents of top of stack into rcx and adjust rsp



Stack Frames

- When a method is called, a stack frame is traditionally allocated on logical "top" of the stack to hold its local variables
- Frame is popped on method return
- By convention, %rbp (base pointer) points to a known offset into the stack frame
  - Local variables referenced relative to %rbp
  - Base pointer common in 32-bit x86 code; less so in x86-64 code where push/pop used less & stack frame has a fixed size so locals can be referenced from %rsp easily
  - We will use %rbp in our project – simplifies addressing of local variables and compiler bookkeeping

Operand Address Modes (1)

- These should cover most of what we'll need

```
movq   $17,%rax          # store 17 in %rax
movq   %rcx,%rax         # copy %rcx to %rax
movq   -16(%rbp),%rax      # copy memory to %rax
movq   %rax,-24(%rbp)    # copy %rax to memory
```

- References to object fields work similarly – put the object's memory address in a register and use that address plus an offset
- Remember: can't have two memory addresses in a single instruction

Operand Address Modes (2)

- A memory address can combine the contents of two registers (with one optionally multiplied by 2, 4, or 8) `plus a constant:`

```
basereg + indexreg*scale + constant
```

- Main use of general form is for array subscripting or small computations - if the compiler is clever
- Example: suppose we have an array of 8-byte ints with address of the array A in %rcx and subscript i in %rax. Code to store %rbx in A[i]

```
movq   %rbx,(%rcx,%rax,8)
```

qword ptr – Intel assembler

- Obscure, but sometimes necessary…
- If the assembler can't figure out the size of the operands to move, you can explicitly tell it to move 64bits with the qualifier "`qword ptr`"

```
mov   qword ptr  [rax+16],[rbp-8]
```

  - Similarly for dword ptr, etc
  - Use this if the assembler complains; otherwise ignore
  - Not an issue in GNU asembler – operand sizes encoded in opcode mnemonics

Basic Data Movement and Arithmetic Instructions

`movq src,dst`

dst ← src

`addq src,dst`

dst ← dst + src

`subq src,dst`

dst ← dst – src

`incq dst`

dst ← dst + 1

`decq dst`

dst ← dst - 1

`negq dst`

dst ← - dst
(2's complement arithmetic negation)

Integer Multiply and Divide

`imulq src,dst`

dst ← dst * src

dst must be a register

`cqto`

`%rdx:%rax` ← 128-bit sign extended copy of %rax

(why??? To prep numerator for idivq!)

`idivq src`

Divide %rdx:%rax by src (%rdx:%rax holds sign-extended 128-bit value; cannot use other registers for division)

%rax ← quotient

%rdx ← remainder

(no division in MiniJava!)

Bitwise Operations

`andq src,dst`

dst ← dst & src

`orq src,dst`

dst ← dst | src

`xorq src,dst`

dst ← dst ^ src

`notq dst`

dst ← ~ dst
(logical or 1's complement)

Shifts and Rotates

`shlq  count,dst`

dst shifted left count bits

`shrq  count,dst`

dst ← dst shifted right count bits (0 fill)

`sarq  count,dst`

dst ← dst shifted right count bits (sign bit fill)

`rolq  count,dst`

dst ← dst rotated left count bits

`rorq  count,dst`

dst ← dst rotated right count bits

Uses for Shifts and Rotates

- Can often be used to optimize multiplication and division by small constants (mul/div by powers of 2)
  - If you're interested, look at "Hacker's Delight" by Henry Warren, A-W, 2nd ed, 2012
    - Lots of very cool bit fiddling and other algorithms
  - But be careful – be sure semantics are OK
    - Example: right shift is not the same as integer divide for negative numbers – shift truncates towards -∞
- There are additional instructions that shift and rotate double words, use a calculated shift amount instead of a constant, etc.

Address Computation Instruction

- `lea dst, src`
  - `lea` stands for load effective address
  - `src` is address expression (in brackets)
  - `dst` is a register
  - Sets `dst` to the address computed by the `src` expression (does not go to memory! – it just does math)
  - Example: lea rax, QWORD PTR[rdx+rcx*4+1]
- Uses:
  - Computing addresses without a memory reference
    - e.g. translation of p = &x[i];
  - Computing arithmetic expressions of the form x+k*i+d
    - Though k can only be 1, 2, 4, or 8

Load Effective Address

- The unary & operator in C/C++

```
leaq  src,dst        # dst ← address of src
```

  - `dst` must be a register
  - Address of `src` includes any address arithmetic or indexing
  - Useful to capture addresses for pointers, reference parameters, etc.
  - Also useful for computing arithmetic expressions that match r1 + scale*r2 + const

Control Flow - GOTO

- At this level, all we have is `goto` and conditional `goto`
- Loops and conditional statements are synthesized from these
- Note: random jumps play havoc with pipeline efficiency; much work is done in modern compilers and processors to minimize this impact

Unconditional Jumps

```
jmp   dst
   %rip ⟵ address of dst
```

- dst is usually a label in the code (which can be on a line by itself)
- dst address can also be indirect using the address in a register or memory location (*reg or *(reg)) – use for method calls, switch

Conditional Jumps

- Most arithmetic instructions set "condition code" bits to record information about the result (zero, non-zero, >0, etc.)
  - True of addq, subq, andq, orq; but not imulq, idivq, leaq
- Other instructions that set condition codes

```
cmpq src,dst # compare dst to src (e.g., dst-src)
testq src,dst      # calculate dst & src (logical and)
```

  - These do not alter src or dst

Conditional Jumps Following Arithmetic Operations

```
jz   label   # jump if result == 0
jnz  label   # jump if result != 0
jg   label   # jump if result > 0
jng  label   # jump if result <= 0
jge  label   # jump if result >= 0
jnge label   # jump if result < 0
jl   label   # jump if result < 0
jnl  label   # jump if result >= 0
jle  label   # jump if result <= 0
jnle label   # jump if result > 0
```

- Obviously, the assembler is providing multiple opcode mnemonics for several actual instructions

Compare and Jump Conditionally

- Want: compare two operands and jump if a relationship holds between them
- Would like to do this

  `jmp`$_{\texttt{cond}}$ `op1,op2,label`

  but can't, because 3-operand instructions can't be encoded in x86-64

  (also true of most other machines for that matter)

cmp and jcc

- Instead, we use a 2-instruction sequence

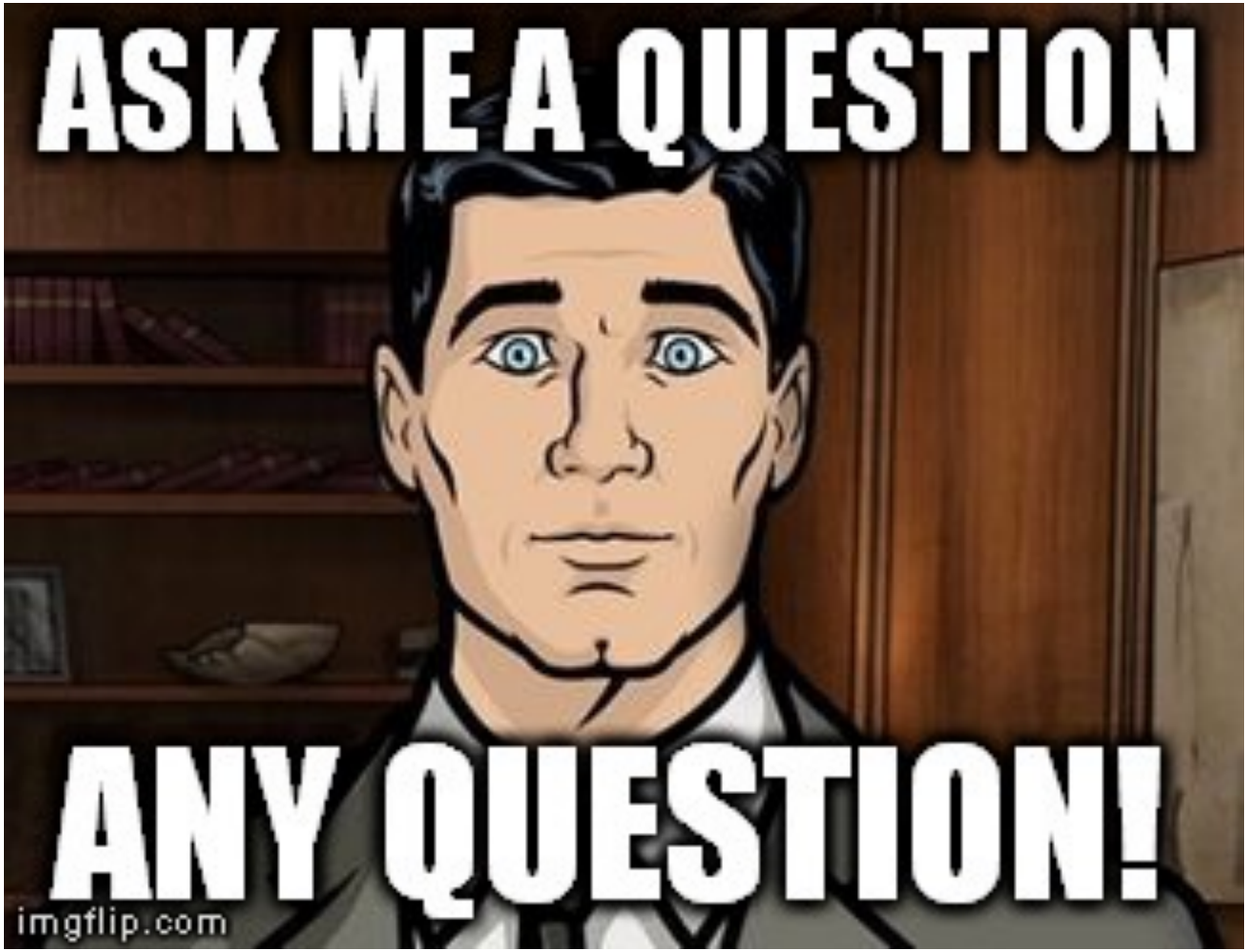
```
cmpq op1,op2
j_cc  label
```

where `j_cc` is a conditional jump that is taken if the result of the comparison matches the condition cc

Conditional Jumps Following Arithmetic Operations

```
je    label    # jump if op1 == op2
jne   label    # jump if op1 != op2
jg    label    # jump if op1 > op2
jng   label    # jump if op1 <= op2
jge   label    # jump if op1 >= op2
jnge  label    # jump if op1 < op2
jl    label    # jump if op1 < op2
jnl   label    # jump if op1 >= op2
jle   label    # jump if op1 <= op2
jnle  label    # jump if op1 > op2
```

- Again, the assembler is mapping more than one mnemonic to some machine instructions



[Meme credit: imgflip.com]