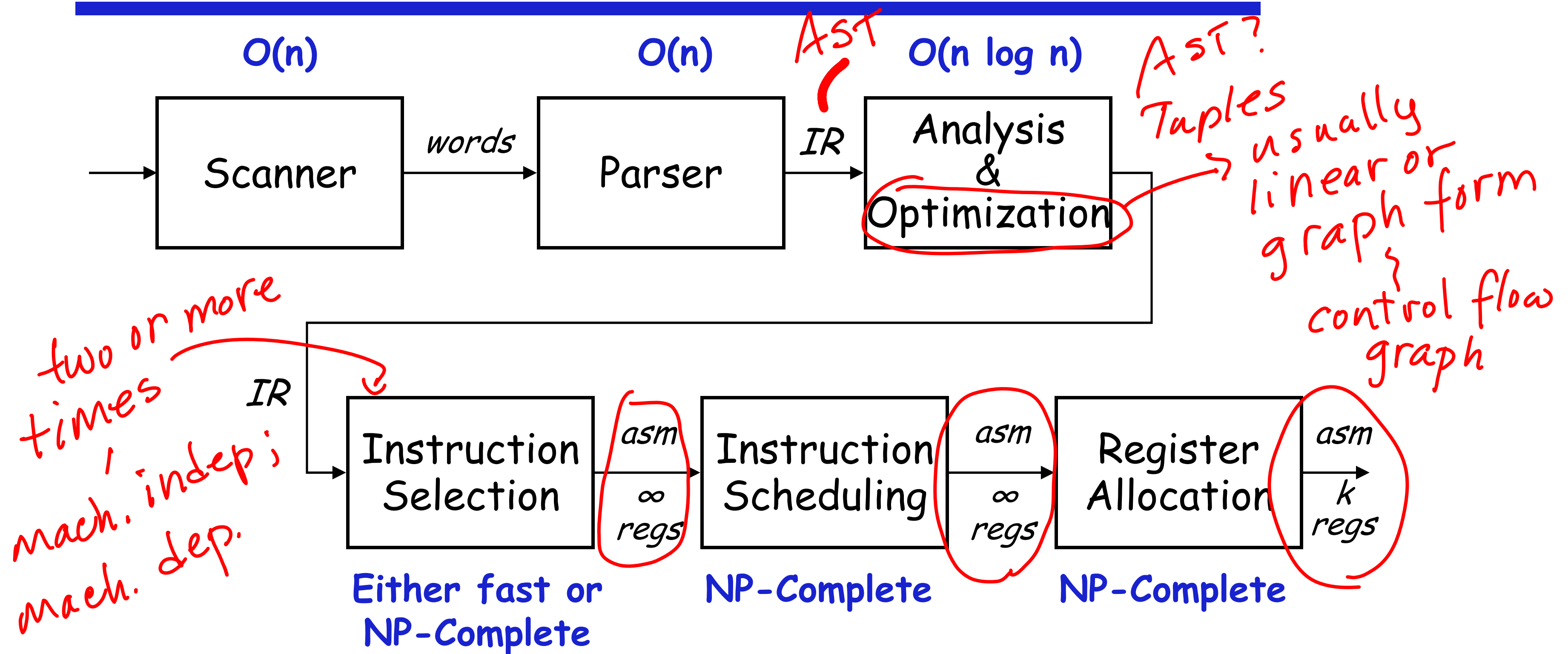
Introduction to Code Generation

Copyright 2003, Keith D. Cooper, Ken Kennedy & Linda Torczon, all rights reserved.
Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.

Structure of a Compiler

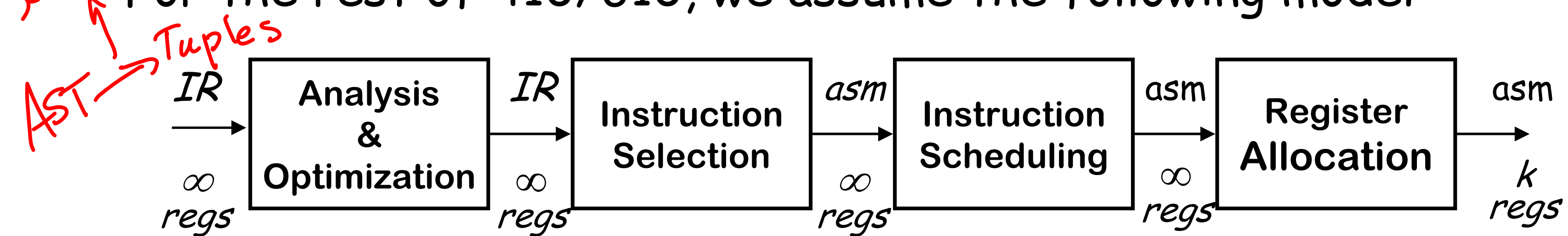


A compiler is a lot of fast stuff followed by some hard problems

- → The hard stuff is mostly in code generation and optimization
- → For superscalars, it's allocation & scheduling that count

Structure of a Compiler

For the rest of 410/610, we assume the following model



- Selection is fairly simple (problem of the 1980s)
- Allocation & scheduling are complex
- Operation placement is not yet critical *(unified register set)*

What about the IR ?

- Low-level, RISC-like IR called ILOC
- Has "enough" registers
- ILOC was designed for this stuff

{ Branches, compares, & labels
Memory tags
Hierarchy of loads & stores
Provision for multiple ops/cycle

Definitions

Instruction selection

- Mapping *IR* into assembly code (*tuples*)
- Assumes a fixed storage mapping & code shape
- Combining operations, using address modes

Instruction scheduling

- Reordering operations to hide latencies
- Assumes a fixed program *(set of operations)*
- Changes demand for registers

Register allocation

- Deciding which values will reside in registers
- Changes the storage mapping, may add false sharing
- Concerns about placement of data & memory operations

These 3 problems are tightly coupled.

AST

machine independent

The Big Picture

How hard are these problems?

Instruction selection

- Can make locally optimal choices, with automated tool
- Global optimality is (undoubtedly) NP-Complete

Instruction scheduling

- Single basic block $\Rightarrow$ heuristics work quickly
- General problem, with control flow $\Rightarrow$ NP-Complete

Register allocation

- Single basic block, no spilling, & 1 register size $\Rightarrow$ linear time
- Whole procedure is NP-Complete

The Big Picture

Conventional wisdom says that we lose little by solving these problems independently

This slide is full of "fuzzy" terms

Instruction selection
- Use some form of pattern matching
- Assume enough registers or target "important" values

Instruction scheduling
- Within a block, list scheduling is "close" to optimal
- Across blocks, build framework to apply list scheduling

Optimal for > 85% of blocks

Register allocation
- Start from virtual registers & map "enough" into k
- With targeting, focus on good priority heuristic

Code Shape

Definition

- All those nebulous properties of the code that impact performance & code "quality"
- Includes code, approach for different constructs, cost, storage requirements & mapping, & choice of operations
- Code shape is the end product of many decisions *(big & small)*

Impact

optimization

- Code shape influences algorithm choice & results
- Code shape can encode important facts, or hide them

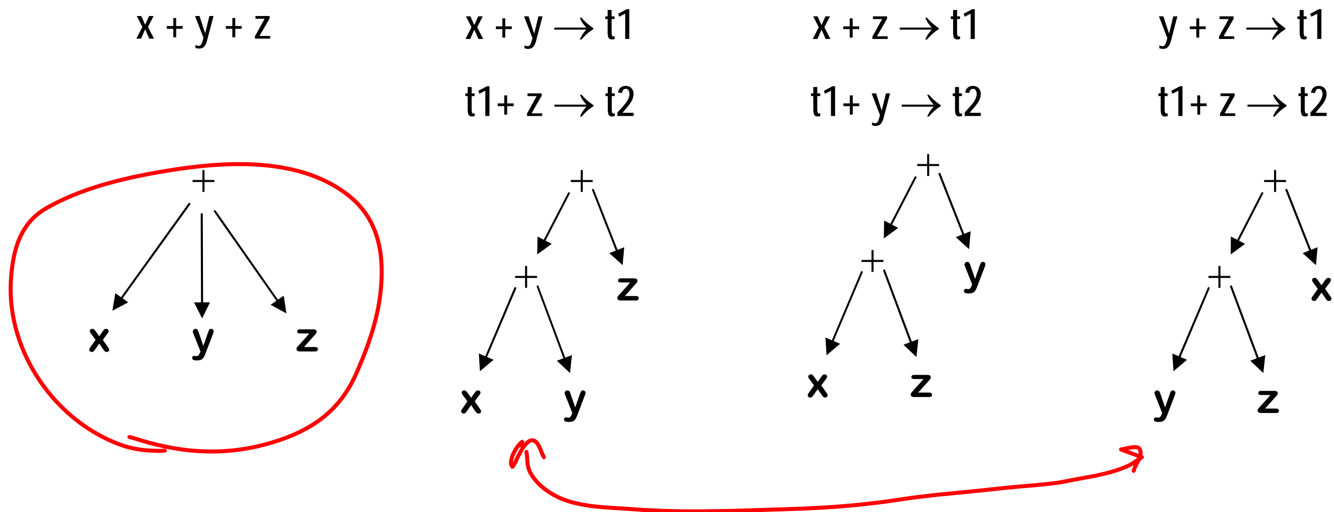
Rule of thumb: expose as much derived information as possible

- Example: explicit branch targets in ILOC simplify analysis
- Example: hierarchy of memory operations in ILOC *(in EaC)*

See Morgan's book for more ILOC examples

Code Shape

My favorite example



Addition is commutative & associative for integers

- What if x is 2 and z is 3?
- What if y+z is evaluated earlier?

The "best" shape for x+y+z depends on contextual knowledge

→ There may be several conflicting options

Code Shape

Another example -- the case statement

- Implement it as cascaded if-then-else statements
  - → Cost depends on where your case actually occurs
  - → O(number of cases)
- Implement it as a binary search
  - → ~~Need a dense set of conditions to search~~
  - → Uniform (log n) cost
- Implement it as a jump table
  - → Lookup address in a table & jump to it
  - → Uniform (constant) cost

Could use perfect hashing as well

Compiler must choose best implementation strategy

No amount of massaging or transforming will convert one into another