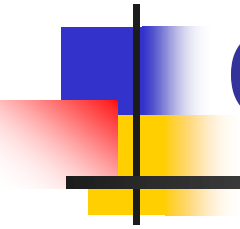
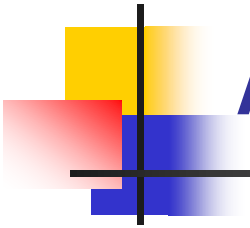


CSE P 501 – Compilers

Instruction Selection

Hal Perkins

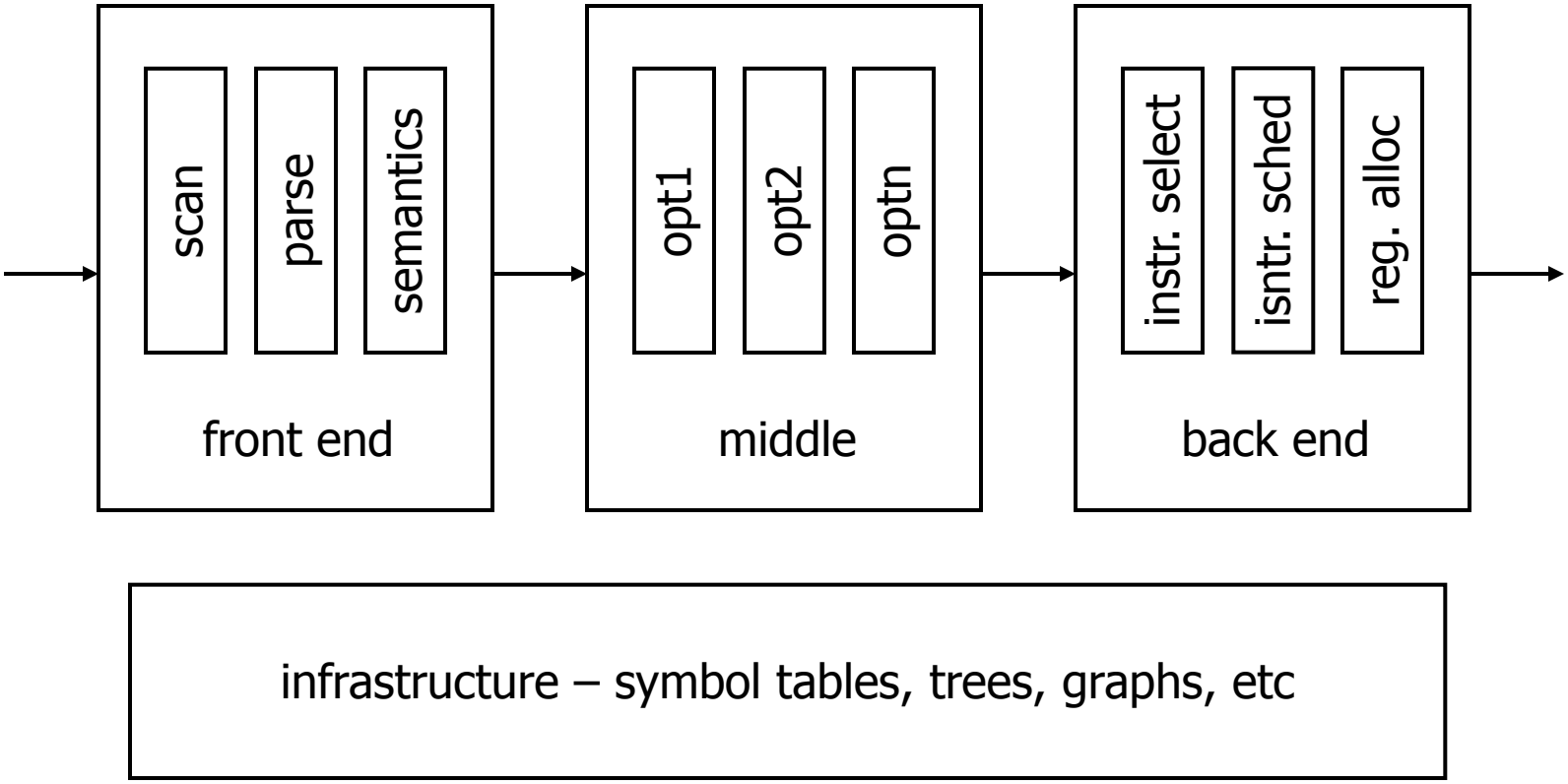
Winter 2008

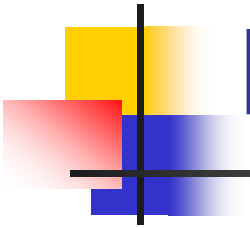


Agenda

- Compiler back-end organization
- Low-level intermediate representations
  - Trees
  - Linear
- Instruction selection algorithms
  - Tree pattern matching
  - Peephole matching
- Credits: Much of this material is adapted from slides by Keith Cooper (Rice) and material in Appel's *Modern Compiler Implementation in Java*

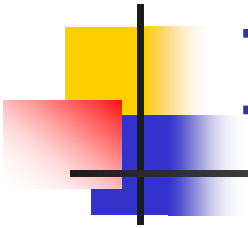
Compiler Organization





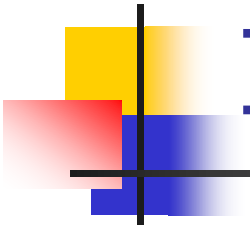
Big Picture

- Compiler consists of lots of fast stuff followed by hard problems
  - Scanner: O(n)
  - Parser: O(n)
  - Analysis & Optimization: ~ O(n log n)
  - Instruction selection: fast or NP-Complete
  - Instruction scheduling: NP-Complete
  - Register allocation: NP-Complete



Intermediate Representations

- Tree or linear?
- Closer to source language or machine?
  - Source language: more context for high-level optimizations
  - Machine: exposes opportunities for low-level optimizations and easier to map to actual code
- Common strategy
  - Initial IR is AST, close to source
  - After some optimizations, transform to lower-level IR, either tree or linear; use this to optimize further and generate code

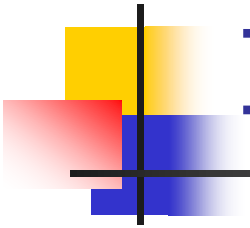


IR for Code Generation

- Assume a low-level RISC-like IR
  - 3 address, register-register instructions + load/store

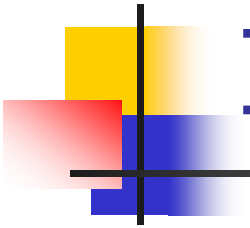
    r1 <- r2 op r3
  - Could be tree structure or linear
  - Expose as much detail as possible
- Assume "enough" registers
  - Invent new temporaries for intermediate results
  - Map to actual registers later

Overview
Instruction Selection

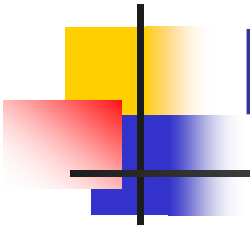


- Map IR into assembly code
- Assume known storage layout and code shape
  - i.e., the optimization phases have already done their thing
- Combine low-level IR operations into machine instructions (take advantage of addressing modes, etc.)

Overview
Instruction Scheduling

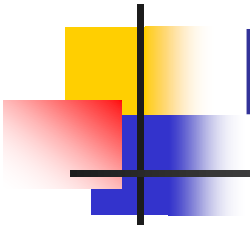


- Reorder operations to hide latencies – processor function units; memory/cache
  - Originally invented for supercomputers (1960s)
  - Now important everywhere
    - Even non-RISC machines, i.e., x86
    - Even if processor reorders on the fly
- Assume fixed program



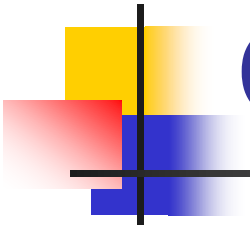
Overview
Register Allocation

- Map values to actual registers
  - Previous phases change need for registers
- Add code to spill values to temporaries as needed, etc.



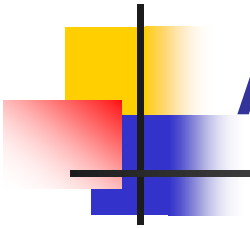
How Hard?

- Instruction selection
  - Can make locally optimal choices
  - Global is undoubtedly NP-Complete
- Instruction scheduling
  - Single basic block – quick heuristics
  - General problem – NP Complete
- Register allocation
  - Single basic block, no spilling, interchangeable registers – linear
  - General – NP Complete



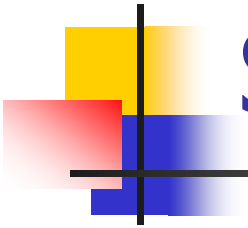
Conventional Wisdom

- We probably lose little by solving these independently
- Instruction selection
  - Use some form of pattern matching
  - Assume "enough" registers
- Instruction scheduling
  - Within a block, list scheduling is close to optimal
  - Across blocks: build framework to apply list scheduling
- Register allocation
  - Start with virtual registers and map "enough" to K
  - Targeting, use good priority heuristic



An Simple Low-Level IR (1)

- Details not important for our purposes; point is to get a feeling for the level of detail involved
  - This example is from Appel
- Expressions
  - CONST(i) – integer constant i
  - TEMP(t) – temporary t (i.e., register)
  - BINOP(op,e1,e2) – application of op to e1,e2
  - MEM(e) – contents of memory at address e
    - Means value when used in an expression
    - Means address when used on left side of assignment
  - CALL(f,args) – application of function f to argument list args



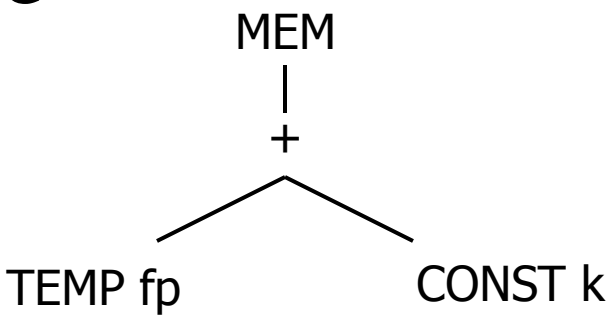
Simple Low-Level IR (2)

- Statements
  - MOVE(TEMP t, e) – evaluate e and store in temporary t
  - MOVE(MEM(e1), e2) – evaluate e1 to yield address a; evaluate e2 and store at a
  - EXP(e) – evaluate expressions e and discard result
  - SEQ(s1,s2) – execute s1 followed by s2
  - NAME(n) – assembly language label n
  - JUMP(e) – jump to e, which can be a NAME label, or more compex (e.g., switch)
  - CJUMP(op,e1,e2,t,f) – evaluate e1 op e2; if true jump to label t, otherwise jump to f
  - LABEL(n) – defines location of label n in the code

Low-Level IR Example (1)

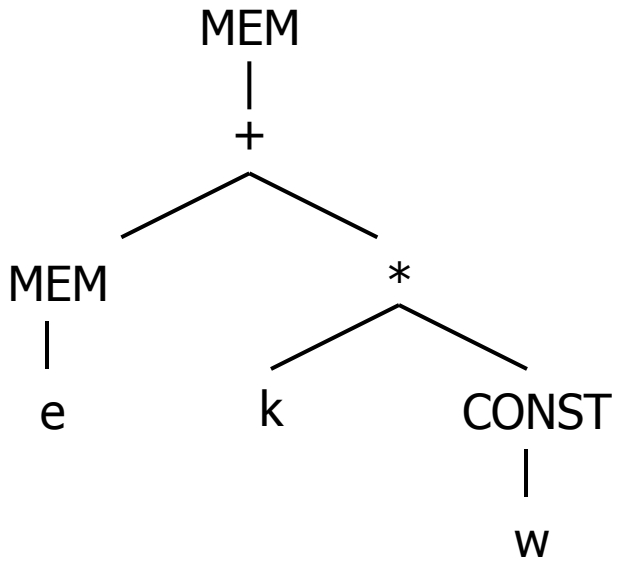
- For a local variable at a known offset k from the frame pointer fp
  - Linear

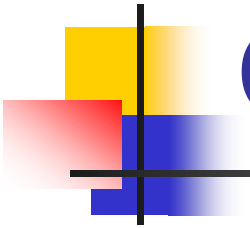
    MEM(BINOP(PLUS, TEMP fp, CONST k))
  - Tree



Low-Level IR Example (2)

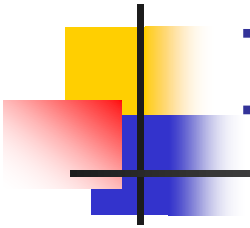
- For an array element e(k), where each element takes up w storage locations





Generating Low-Level IR

- Assuming initial IR is an AST, a simple treewalk can be used to generate the low-level IR
  - Can be done before, during, or after optimizations in the middle part of the compiler
    - Typically AST is lowered to some lower-level IR, but maybe not final lowest-level one used in instruction selection
- Create registers (temporaries) for values and intermediate results
  - Value can be safely allocated to a register when only 1 name can reference it
    - Trouble: pointers, arrays, reference parameters
  - Assign a virtual register to anything that can go into one
  - Generate loads/stores for other values

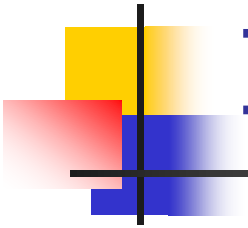


Instruction Selection Issues

- Given the low-level IR, there are many possible code sequences that implement it correctly
  - e.g. to set eax to 0 on x86

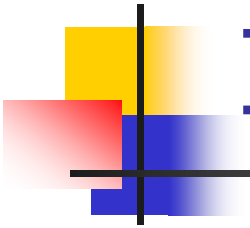
```
mov  eax,0        xor  eax,eax
sub  eax,eax      imul  eax,0
```

  - Many machine instructions do several things at once – e.g., register arithmetic and effective address calculation



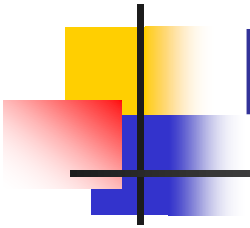
Instruction Selection Criteria

- Several possibilities
  - Fastest
  - Smallest
  - Minimize power consumption (ex: don't use a function unit if leaving it powered-down is a win)
- Sometimes not obvious
  - e.g., if one of the function units in the processor is idle and we can select an instruction that uses that unit, it effectively executes for free, even if that instruction wouldn't be chosen normally
    - (Some interaction with scheduling here…)



Implementation

- Problem: We need some representation of the target machine instruction set that facilitates code generation
- Idea: Describe machine instructions using same low-level IR used for program
- Use pattern matching techniques to pick machine instructions that match fragments of the program IR tree
  - Want this to run quickly
  - Would like to automate as much as possible

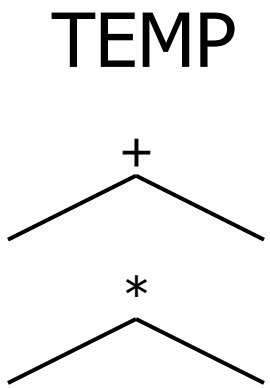


Matching: How?

- Tree IR – pattern match on trees
  - Tree patterns as input
  - Each pattern maps to target machine instruction (or sequence)
  - Use dynamic programming or bottom-up rewrite system (BURS)
- Linear IR – some sort of string matching
  - Strings as input
  - Each string maps to target machine instruction sequence
  - Use text matching or peephole matching
- Both work well in practice; actual algorithms are quite different

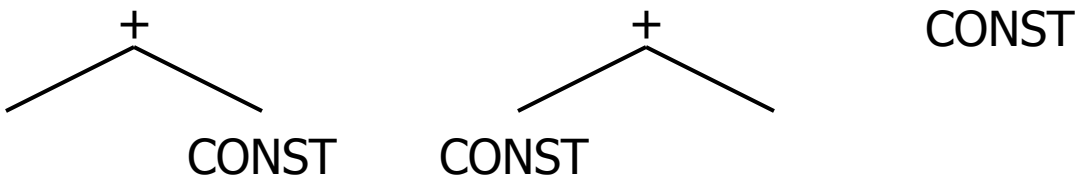
An Example Target Machine (1)

- Arithmetic Instructions
  - (unnamed) ri
  - ADD ri <- rj + rk
  - MUL ri <- rj * rk
  - SUB and DIV are similar

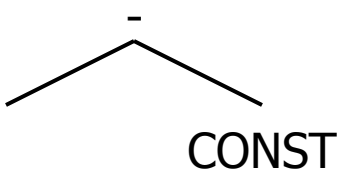


An Example Target Machine (2)

- Immediate Instructons
  - ADDI ri <- rj + c

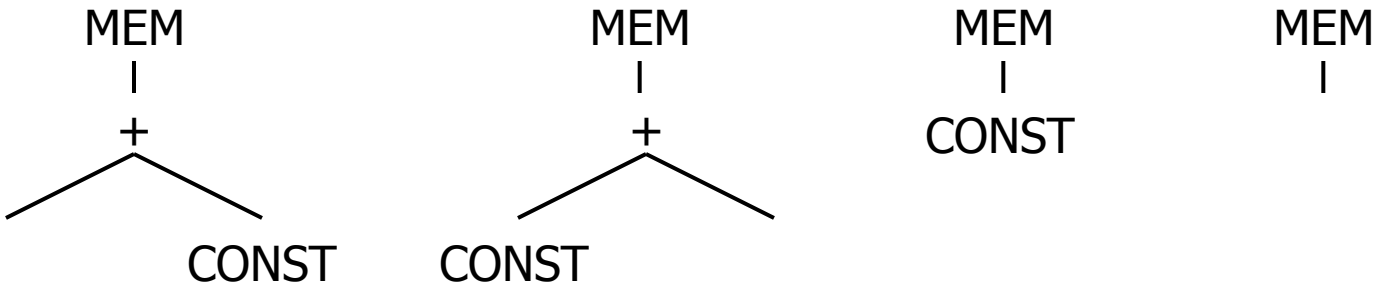


  - SUBI ri <- rj - c



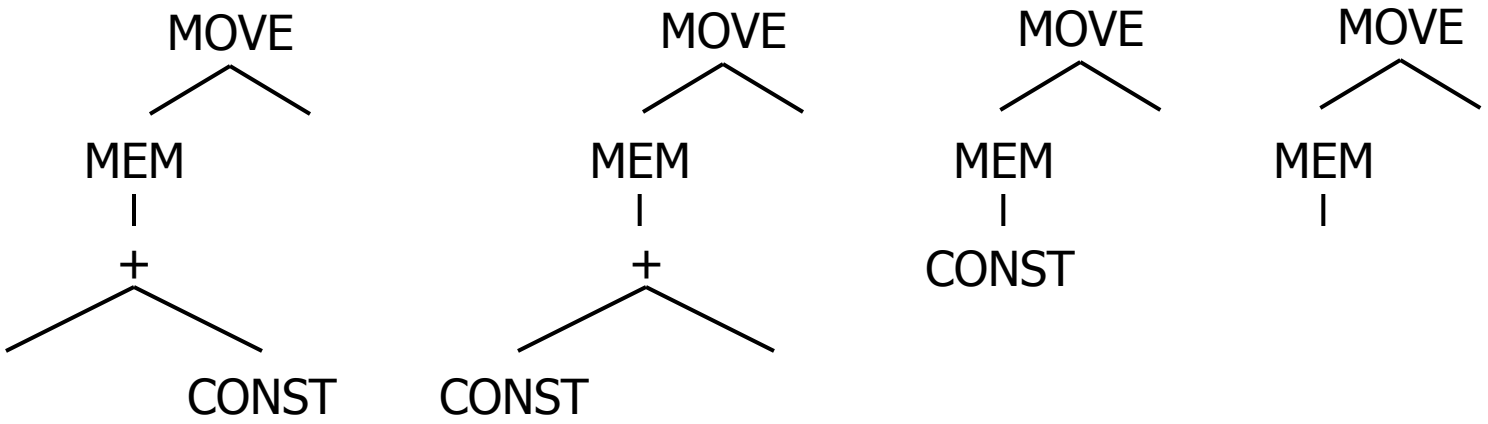
An Example Target Machine (3)

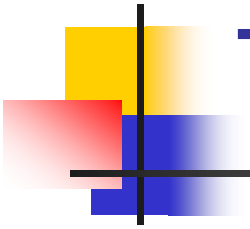
- Load
  - LOAD ri <- M[rj + c]



An Example Target Machine (4)

- Store
  - STORE M[rj + c] <- ri



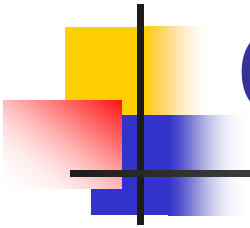


Tree Pattern Matching (1)

- Goal: Tile the low-level tree with operation (instruction) trees
- A *tiling* is a collection of <node,op> pairs
  - node is a node in the tree
  - op is an operation tree
  - <node,op> means that op could implement the subtree at node

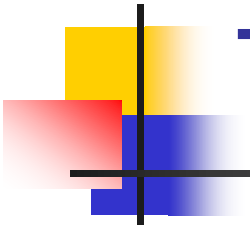
Tree Pattern Matching (2)

- A tiling "implements" a tree if it covers every node in the tree and the overlap between any two tiles (trees) is limited to a single node
  - If <node,op> is in the tiling, then node is also covered by a leaf in another operation tree in the tiling – unless it is the root
  - Where two operation trees meet, they must be compatible (i.e., expect the same value in the same location)



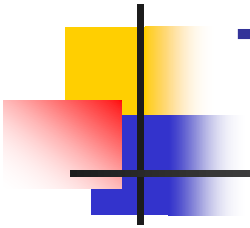
Generating Code

- Given a tiled tree, to generate code
  - Postorder treewalk; node-dependant order for children
  - Emit code sequences corresponding to tiles in order
  - Connect tiles by using same register name to tie boundaries together



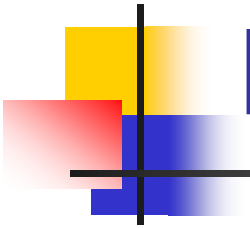
Tiling Algorithm

- There may be many tiles that could match at a particular node
- Idea: Walk the tree and accumulate the set of all possible tiles that could match at that point – Tiles(n)
  - Later: can keep lowest cost match at each point
  - Generates local optimality – lowest cost match at each point



Tile(Node n)

```
Tiles(n) <- empty;
if n has two children then
    Tile(left child of n)
    Tile(right child of n)
    for each rule r that implements n
      if (left(r) is in Tiles(left(n)) and right(r) is in Tiles(right(n)))
          Tiles(n) <- Tiles(n) + r
else if n has one child then
    Tile(child of n)
    for each rule r that implements n
       if(left(r) is in Tiles(child(n)))
          Tiles(n) <- Tiles(n) + r
else /* n is a leaf */
    Tiles(n) <- { all rules that implement n }
```



Peephole Matching

- A code generaton/improvement strategy for linear representations
- Basic idea
  - Look at small sequences of adjacent operations
  - Compiler moves a sliding window ("peephole") over the code and looks for improvements

Peephole Optimizations (1)

- Classic example: store followed by a load, or push followed by a pop

original	improved
mov [ebp-8],eax	mov [ebp-8],eax
mov eax,[ebp-8]	
push eax	---
pop eax	

Peephole Optimizations (2)

- Simple algebraic identies

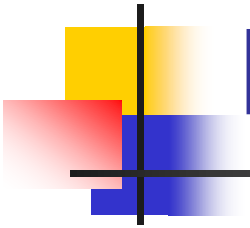
original	improved
add eax,0	---
add eax,1	inc eax
mul eax,2	add eax,eax
mul eax,4	shl eax,2

Peephole Optimizations (3)

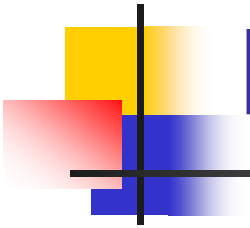
- Jump to a Jump

original	improved
jmp here	jmp there
here: jmp there	

Implementing Peephole Matching

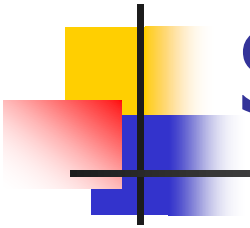


- Early versions
  - Limited set of hand-coded patterns
  - Modest window size to ensure speed
- Modern
  - Break problem in to expander, simplifier, matcher
  - Apply symbolic interpretation and simplification systematically



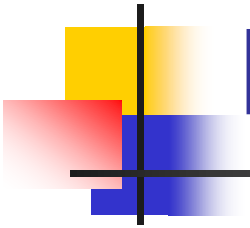
Expander

- Turn IR code into very low-level IR (LLIR)
- Template-driven rewriting
- LLIR includes all direct effects of instructions, e.g., setting condition codes
- Big, although constant size expansion



Simplifier

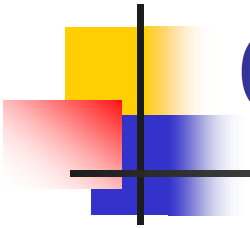
- Look at LLIR through window and rewrite using
  - Forward substitution
  - Algebraic simplification
  - Local constant propagation
  - Eliminate dead code
- This is the heart of the processing



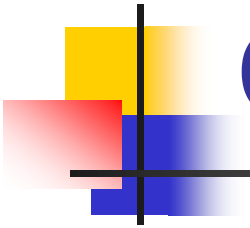
Matcher

- Compare simplified LLIR against library of patterns
- Pick low-cost pattern that captures effects
- Must preserve LLIR effects; can add new ones (condition codes, etc.)
- Generates assembly code output

Peephole Optimization Considered



- LLIR is largely machine independent (RTL)
- Target machine description is LLIR -> ASM patterns
- Pattern matching
  - Use hand-coded matcher (classical gcc)
  - Turn patterns into grammar and use LR parser
- Used in several important compilers
- Seems to produce good portable instruction selectors



Coming Attractions

- Instruction Scheduling
- Register Allocation
- Optimization
- Supporting technologies (if time)
  - Memory management & garbage collection
  - Virtual machines, portability, and security