I. Handout 3 as an example for this lecture
II. We want to do optimization and program analysis
   A. Generally want to do this at a lower level than the source
      1. You want to optimize what is actually run
      2. A lot of details are hidden at the source level
         a. Array references, hidden computations, registers, what the CPU is
            actually doing, memory hierarchies, ...
   B. We therefore want to have a different representation to work with
      1. This could be tree-based
      2. Abstract instruction based is generally easier
         a. Closer to the actual hardware you want to analyze for
         b. Programs are linear and directed graphs, not trees
   C. We first need to understand the target environment
      1. For compilation
      2. For program analysis – what level and type of analysis is appropriate

II. Target Environments
   A. Need to understand the use of the language
      1. The environment in which the program will be run
      2. What executing the program really means
   B. Basic components
      1. Type and object representations
      2. Memory layout in general
      3. How various language features are managed
   C. Type and object representations
      1. Primitive types: int, char, Boolean, pointer
      2. Object types
         a. Storage, virtual function tables
      3. Array types
         a. One dimensional arrays
         b. Multi-dimensional arrays
D. Memory Layout
   1. Stack versus dynamic memory
   2. Storage management
      a. Garbage collection
      b. Reference counting
      c. Programmer-dependent
E. Virtual function calls
   1. Virtual function table (vtable)
   2. Each method is an index into that table
   3. New methods are added at the end
      a. Show how this handles inheritance
      b. How might you handle multiple inheritance
F. Arrays and array access
   1. Store an array as a length + data as an example
   2. Access directly
      a. A[i] = A + 4 + i*sizeof(base)
      b. Why might you do this? What optimizations does it allow?
   3. Different ways of handling multidimensional arrays
      a. Direct access
      b. How to take slices of a direct multiple dimensional array
      c. Arrays of arrays
G. Constructor calls
   1. Allocate memory and pass it to the constructor
   2. Set up vtable here or in the constructor
      a. Depends on language (C++ versus Java)
   3. Constructors
      a. Already added call to super and initialization
H. Strings
   1. Need to make objects from these
   2. Assume this is done by a run time routine
I. Built in routines
   1. Done with the external modifier
   2. Assume a library that implements the right names
III. Intermediate Representations
   A. Art not a science
      1. If one exists, you probably should use it
      2. Difficult to create a good one
3. Machine independence is even harder

B. Reasons (purpose) for an IR
   1. High-level Optimization (not transformations)
      a. Different IRs will facilitate different optimizations
      b. IR should at least enable optimization, if not simplify it
      c. GIVE EXAMPLES – common subexpression elimination
   2. Low-level optimization
      a. RISC machines with pipelines
      b. Register utilization
      c. GIVE EXAMPLES – instruction reordering
   3. Program transformations
      a. Inlining routines
      b. Templates
      c. Refactorings
      d. GIVE EXAMPLES – C++ templates, adding constructors, refactorings

C. Principles
   1. Enable and facilitate the appropriate optimizations
      a. High-level transformations
      b. Middle level code manipulations
      c. Low level instruction manipulations
   2. Reflect the target architecture
      a. IR for a stack architecture
      b. IR for a register architecture
      c. IR for generating C code

D. Result
   1. Multiple IRs
   2. Reflecting different levels

IV. High-level Intermediate Representations
   A. Uses
      1. Program transformations
      2. Large scale code manipulations (refactorings)
      3. Type analysis
   B. Requirements
      1. Needs to be easy to do substitutions
      2. Needs to reflect source at times
   C. ASTs provide this functionality
      1. How might you do inlining
2. How might you implement templates

V. Low-level Intermediate Representations

A. Uses
   1. Register utilization
   2. Machine-specific transformations
      a. Specific instruction register manipulations
      b. Choosing the right instruction and addressing mode
      c. Instruction ordering for pipelining

B. Requirements
   1. Needs to reflect machine details
   2. Can still be manipulated
   3. Without unnecessary detail (e.g. exact offsets)

C. Assembler provides this functionality
   1. So do other languages we will get to later

VI. Middle-level Intermediate Representations

A. Uses
   1. Optimization
      a. Removing unnecessary computations
      b. Rearranging computations
      c. Constant propagation
   2. Program analysis
      a. Flow analysis
      b. Symbolic execution
   3. Machine independence
      a. Separate front and back ends for a compiler

B. Requirements
   1. Reflect the target architecture (not the machine code)
   2. Avoid machine-specific details where possible
   3. Easy to manipulate
   4. Contains all the information about the program
   5. Usable as both input and output for each optimization
   6. Usable for program analysis

C. Components
   1. Instructions
      a. Tuples (Op, Operand, ..., Result)
      b. Operator reflects high-level machine operations
         i. Can reflect machine operators as well (for low-level work)
c. May be labeled

2. Temporaries
   a. Hold results
   b. Think of these as registers
      i. Actually can be registers, stack space, static storage, etc.
      ii. Not reused (given out freely)
      iii. Infinite number of these
   c. May be typed
      i. Run time type set (NONE, BOOL, CHAR, INT, LABEL, COND_xx)
   d. Actually want to reuse temporaries
      i. When they represent the same computation
      ii. This means that constants will reuse temporaries
      iii. As will duplicate computations

3. Constants
   a. Of the types supported by the language
   b. Operands are either temporaries, constants, or labels
   c. Are unique representations (all instances of 1 are the same object)

4. Instruction sequences
   a. Need to represent code as a sequence of instructions
   b. Generally done per routine
   c. Actually we will break this up later into logical units

5. Variables
   a. Use temporaries exclusively for local variables
   b. Use temporaries exclusively for parameters
   c. Use temporaries for global variables, but store these as well

VII. Instructions
A. Opcode, operands, and result
   1. Generally typed
      a. Operand types are known
      b. Result types are known

B. Opcodes
   1. Calling sequences
      a. ENTER #param
      b. PARAM #i,T
      c. RETVAL T
      d. EXIT
      e. CALL m,a1,...,an
f. CALLR m,a1,...,an,dT  
g. VCALL m,a1,...,an  
h. VCALLR m,a1,...an,dT  

2. Arithmetic, etc  
a. MOVE sT,dT  
b. LDC c,dT  
c. C2I sT,dT  
d. I2C sT,dT  
e. X2B sT,dT  
f. iABS, bNEG, iNEG  
g. iADD, iSUB, iMUL, iDIV, iMOD  
h. iASH  

3. Control flow  
a. CMP s1,s2,dT  
b. BCOND sT,dA1,dA2  
c. BRANCH dA  

4. Memory access  
a. ST vT,aT  
b. LD aT,vT  
c. aADD s1,s2,dT for field offsets  
d. vADD s1,s2,dT for vtable offsets  

5. Other  
a. ALLOC size,dT for new  
b. STRING const,dT for creating strings  

C. Temporaries and instructions  
1. Two instructions with the same operator and args  
a. Compute the same thing  
b. Should have the same resultant temporary  
c. With exceptions for instructions that are unique (i.e. ALLOC, CALL)  

VIII. Basic Blocks  
A. What is a basic block and why is it useful  
1. Straight-line code without any branches  
2. If you enter it, all instructions will be executed  
3. Instructions can be rearranged safely  
4. Intermediate results can be discarded  
5. Omnipresent in optimizations  
B. We arrange instructions in blocks directly
1. IR is a graph of basic blocks
2. Each block is linked to its predecessors and successors
3. Successors are labeled T/F where appropriate
4. Could use other labeling (e.g. for a computed goto)
   a. Switch statements should be mapped to corresponding code
C. Blocks are numbered (1...n)
D. Edges between blocks
   1. Labeled by type
      a. ALWAYS
      b. TRUE
      c. FALSE
      d. NEVER
   2. Bi-directional
IX. Routines
A. Top level holder for the IR
   1. Optimization, analysis is done one routine at a time
   2. Temporaries, constants, blocks are managed by the routine
B. Properties
   1. Each routine has a unique start and exit block
   2. All blocks are reachable from the start block
   3. All blocks can reach the exit block
X. Example from handout
XI. HOMEWORK
A. Read Chapter 6
B. Meet with me to discuss project status