I. Handout 3 as an example for the lecture
II. Last time we went over the Intermediate Representation
   A. Structures
      1. Routines containing blocks
      2. Blocks containing code
      3. Links between the blocks
   B. Next we consider building it
      1. Note that this is equivalent to basic code generation
         a. Use assembler/target code as the IR
         b. You can then generate actual code directly
II. Representing the IR
   A. Trivial – list of instructions
      1. But this isn’t that useful
      2. More useful to reflect the logical structure in the physical
      3. Look at this for Decaf, but it should be similar for other languages, IRs
   B. Icode
      1. Top-level factory
      2. Provides iterators over routines and classes
      3. Provides calls to print the code
      4. You will use
         a. createRoutine(Ast.Method, AsmGenerator): Routine
            i  AsmGenerator: information about allocation
            ii Builds a new routine structure
         b. addClass(Ast.Class)
            i  Used to that you can generate VTables later on
   C. Routine
      1. Holds information about a routine
         a. Block getStartBlock()
         b. Block getExitBlock()
         c. Iterator getBlocks()
d. Iterator getTemporaries()
e. String getName()
f. String getOutputName()

2. Factory for blocks
a. Block createBlock()
b. Start and exit blocks created automatically

3. Factory for temporaries
a. Temporary createTemporary(CodeType ct)
b. Temporary createTemporary(CodeType,int op,Argument [])
   i. Why is this call needed?
   ii. Temporaries are unique for a given op & argument set

4. Factory for constants
a. Constant createConstant(CodeType ct,int v)
b. Constant createConstant(String s)
c. Constant createConstant(Block b)

5. Factory for instructions
a. Instruction createInstruction(int opc, Argument [] args)

6. Factory for CodeValue information
a. CodeValue is a typed value we will use in code generation
b. Consists of an argument (Temporary or Constant)
c. And a flag (CodeType)
   i. CONDITION: block ending w/test and branch
   ii. TEMPORARY: temporary result
   iii. ADDRESS: temporary holding address
   iv. METHOD: reference to a method
   v. LOCAL_ADDRESS: address is the temporary

7. Other functions
a. Allocating memory for locals
b. Managing jumps at the end of blocks

D. Block
1. Maintain graph of blocks
   a. Block addBlock(EdgeType)
   b. Block addEdge(EdgeType,Block)

2. Factory for instructions
   a. addInstruction(Instruction i)
   b. addInstruction(int opc,Argument[] args)
   c. addInstruction(int opc, Argument args...)
d. addConditionalBranch(Argument a)
e. Temporary addResultInstruction(CodeType, into pc, Argument ...)
   i. Builds a new temporary for instruction
   ii. Adds the instruction
   iii. Returns the temporary

3. Access methods
   a. Block getNextBlock(EdgeType)
   b. Iterator<Edge> getSuccessors()
   c. Iterator<Edge> getPrecessors()
   d. ListIterator<Instruction> getInstructions()
   e. ListIterator<Instruction> getInstructionsReverse()

E. Edge
   1. Managed by blocks
   2. Access methods
      a. EdgeType getEdgeType()
      b. Block getFromBlock()
      c. Block getToBlock()

F. Instruction
   1. Generally created and just accessed
      a. getOpCode()
      b. int getNumArgument()
      c. Argument getArgument(int)
      d. Int getNumOperand()
      e. Argument getOperand(int)
      f. Temporary getTemporaryOperand(int)
      g. Temporary getTarget()
   2. Can set arguments, operands, target

G. Argument (Temporary, Constant)
   1. Access methods
      a. CodeType getCodeType()

H. Constant
   1. Access methods
      a. Int getIntValue()
      b. String getStringValue()
      c. Block getCodeBlock()

I. Temporary
   1. Access methods
a. Int getId()
b. Instruction getDefiningInstruction()
c. Boolean isVariable()
d. get/set Register
e. Boolean hasOffset, get/set Offset

III. Creating the IR from the AST
A. Creating the IR from the AST
   1. Done via a visitor over the AST
   2. Output involves invoking routines to create the IR
B. Planning
   1. For each AST node, determine what code will be generated
      a. Determine what information is needed from other nodes
      b. Determine how to generate that information
      c. Determine how to generate the code
   2. Can be done at the IR level or at the assembler level
      a. Requires determining what the code will look like
      b. Requires determining how to represent language constructs
      c. Requires understanding target environment and architecture
C. Most of the code generation can be done in one pass
   1. A prepass to generate the offsets is helpful however
      a. Could be done as part of resolution pass
   2. Main pass does the actual code generation

IV. Allocation Pass
A. Purpose
   1. For each field and method, determine its offset
   2. For each class, determine size to allocate
   3. For each local, determine stack offset
      a. Could be optional – only if you need to store locals
      b. Not necessary
   4. For each parameter, determine stack offset
B. AsmGeneartor models the target architecture
   1. Formals (parameters)
      a. Int getFormalBase()
      b. Int getFormalSize(DecafType)
      c. Int getFormalAlignment(DecafType)
   2. Locals (variables)
      a. int getLocalBase()
b. int getLocalSize(DecafType)
c. int getLocalSize(CodeType)
d. int getLocalAlignment(DecafType)
e. int getLocalAlignment(CodeType)
f. int getLocalAlignment(int off, CodeType)

3. Fields
   a. int getFieldBase()
   b. int getFieldSize(DecafType)
   c. int getFieldAlignment(DecafType)

4. VTable
   a. int getVtableBase()
   b. int getVtableIncrement()

C. Visitation methods
1. Class node
   a. PRE:
      i. field_offset = sizeof(superclass offset)
      ii. method_offset = sizeof(superclass vtable)
      iii. Handle case with no superclass: getVtableBase(), getFieldBase()
      iv. Track current super type in visitor
   b. POST:
      i. type->setLengths(field, method) offsets
      ii. clear values (push-pop if nesting allowed)

2. Field node
   a. Get type; get size and alignment for type
   b. Align offset based on type
   c. Store offset
   d. Add length to offset

3. Method node
   a. Check if this is a redefinition of a supertype method
      i. If so, set offset to that of the parent
      ii. Otherwise, set to method_offset and increment the offset
   b. PRE: Set formal offset to getFormalBase
   c. POST: clear formal offset

4. Formal node
   a. Allocate space using formal_offset and type
   b. Set using node->setOffset()

5. Locals
a. What would you have to do here to allocation locals
   i. Maintain local offset for each routine
   ii. Update it by allocating each local
   iii. But you also have to handle scopes
b. Easier: don’t allocate locals, leave them in temporaries
   i. Temporaries will be assigned to memory where needed
   ii. Note this would be incompatible with debugging

V. Expression Generation Framework

A. Associate a value with each expression node
1. What does the value represent
   a. Internalization of what is happening in the code
   b. Where the result is stored
   c. What type of result it is (run time data type)
   d. What kind of result it is (address, value, special)

2. What kinds or results are there
   a. Resultant temporary
   b. For use in an If/While: result is a branch
   c. &&, ||: result is a branch (but might be a temp)
   d. ==, <=, !=, >=: result is conditional temporary
   e. Method
   f. Variable: can be lhs (address) or rhs (value)

3. CodeValue (factory in routine)
   a. CodeGenType getCodeGenType()
      i. CONDITION: block ending w/test and branch
      ii. TEMPORARY: temporary result
      iii. ADDRESS: temporary holding address
      iv. METHOD: reference to a method
      v. LOCAL_ADDRESS: address is the temporary
      vi. ERROR
   b. CodeType getCodeType()
      i. NONE, BOOL, CHAR, INT, REF, COND_EQ, COND_NE, COND_LT,
         COND_LE, COND_GT, COND_GE, LABEL
   c. Block getFalseBlock(), getTrueBlock()
   d. Argument getArgument()
   e. Temporary getVariableTemporary()
   f. Set methods
i. setArgument, setCondition, setTrueBlock, setFalseBlock, setVariableTemporary

g. Other possible information:
i. If value is a constant, the constant value

B. Converting values
1. Two approaches: top-down or bottom-up
2. Top-down
   a. Set a flag in visitor indicating what type of output is desired
      i. Visitor generates that type of output
   b. Alternately have visitors for different type
      i. Easier to do if not done with visitors
3. Bottom-up (easier?)
   a. Each item is free to generate whatever it wants within limits
      i. Must generate narrowest form
      ii. Lvalues must generate addresses, for example
   b. Provide a common routine for converting the given from to desired
      i. That routine can issue instructions as needed
      ii. Result is guaranteed to be a value of the appropriate type
      iii. Handling errors (exceptions/ERROR type)
   c. The set of values is language dependent

C. Associating values
1. Direct association: store the values in the AST
   a. These won’t be need later however, waste of space
2. Maintain a value stack in the visitor
   a. Each expression pushes a value onto the stack
   b. Expression can pop values of subexpression, convert as needed, then push result
   c. Implies visitor value_stack

D. Basic concepts
1. Current block:: where next instruction will go
   a. This has to be maintained in the visitor
2. Each AST node will add instructions and modify this
3. Basic idea is to generate the code for the node
   a. With holes for the code from its children
   b. Using visit calls to generate code for the children
4. Requires planning what the code for each node will look like
   a. Requires understanding how to tie children code to parent
VI. Expression Code Generation

A. Unary Operators
   1. Pop value off the stack
   2. Convert to temporary
   3. Call cur_block->addResultInstruction(type,op,args...)
      a. This find temporary for op...args
      b. If needed, it create a new temporary with the given type
      c. Then it adds the corresponding instruction

B. Binary operators
   1. Pop both operands off the stack
   2. Convert them to temporaries
   3. Generate a new instruction as appropriate

C. Short circuit operators
   1. Generate a condition from the left side
   2. True/False branch to new block
   3. Generate a condition from the right side
   4. True/false branch and link up
   5. Note that values here are branches

D. Assignment
   1. Pop both operands off stack; rhs to temporary, lhs to address
   2. If lhs is temporary (local address) generate move
   3. if lhs is address, generate store

E. Literals
   1. Use LDC or STRING to appropriate temporary

F. Names
   1. Check for ‘length’
      a. Pop off stack and load into T0
      b. LDC array_length_offset,Tx
      c. aADD T0,Tx,Ty
      d. LD [Ty],Tr
      e. Push Tr as value
   2. For var/arg
      a. Just push the temporary with type = LOCAL_ADDRESS
      b. Visitor symbol_map mapping symbol to temporary
   3. For field
      a. Pop lhs off stack and load into T0
      b. aADD T0,<offset>,Tr
c. Push Tr as ADDRESS
4. For method
   a. Push offset on stack as METHOD
G. New Object
1. ALLOC class_length, class_name, T0
2. Class name used for setting up the Vtable
3. Class length used for call to malloc
4. Result is used for subsequent call to constructor
5. Store result in a new temporary
H. New Array
1. LDC <type>, T0
2. LDC <#dims>, T1
3. Pop dims off value stack into temporaries
4. CALLR <_ArrayAllocat>, T0, T1, Tdim1, Tdim2, ...
I. Array reference
1. LDC <offset of length>, T0
2. LDC <length of element>, T1
3. aADD Tarr, T0, T2
4. iMUL Tidx, T1, T3
5. aADD T2, T3, T4
6. Push T4 with type ADDRESS
J. Static call
1. Pop values off stack into temporaries
2. CALLR <routine_name>, Ta1, ... Tan, Tresult
3. This includes constructor calls
K. Virtual call
1. What is on the stack at this point?
2. VCALLR <vtableoffset>, Tthis, Ta0, ..., Tan, Tresult
VII. Control Flow
A. Handling Routines
  1. What needs to be done
     a. Register classes
     b. Create ICode Routine
     c. Generate prefix code
  2. Class
     a. Call DecafCode.Icode addClass() to register the class
  3. Method (if not external)
a. Begin a new routine
b. Make its start block the current block
c. Add ENTER instruction
d. Create temp for return value (unless void)
e. At end, add edge from current block to Exit block
   i. Handle ctor return values
   ii. Add EXIT instruction

4. Formal (if not external (cur_block != null))
   a. Need a temporary to match the variable
      i. Where is the variable stored
      ii. Temporary-based on instruction, LD [offset]
   b. Then generate PARAM #i,Temp
      i. Note you need to track the argument number
   c. Save first formal for constructor exit

5. StatementList
   a. If first statement in a method, create a new block

B. Statements (Show/ask what the code looks like in each case)
   1. IfStatement
   2. WhileStatement
   3. ReturnStatement
   4. ContinueStatement
   5. BreakStatement
   6. OpExpression:: ANDAND, OROR
   7. Call: what do you want to do here

C. Required data in the Visitor
   1. Current routine
   2. Current block
   3. Block to link to for continue statements
   4. Block to link to for break statements
   5. Temporary to use for return value
   6. Flag indicating whether we have seen first statement yet
   7. Pointer to arg0 for constructors
   8. Argument number

VIII. Example
   A. Generate icode for WHILE block in handout3
   B. Go over what is generated elsewhere in handout

IX. Generating machine code or assembler
A. Can be done from the intermediate code
B. Can be done directly
   1. Register allocation
      a. Do this on the fly, storing registers as needed
      b. Generate code for a stack machine
   2. Forward references
      a. Generate assembler
      b. Maintain linked list of back references
C. Interpretation
   1. Directly from ASTs
   2. Through a IR that represents a virtual machine

X. Note for writing a simple fast compiler
A. Recursive descent grammars are nice for control flow and high level constructs
   1. Don’t work that well for expressions
   2. Bottom up grammars are easier for expressions
B. Write a compiler using a top-down grammar except for expressions
C. Use a two-stack (Dijkstra) scanner for expressions
D. The expression stack / unstack mechanism corresponds to the expression code generation method cited here
   1. Can generate code directly while parsing