I. Concrete Syntax Trees
   A. You’ve all seen simple parsing (CS31 or equivalent)
   B. Suppose we parse an expression
      1. What does is the parse representation
         b. Grammar 2:
            i. E := T E1
            ii. E1:= + E | -E | ϵ
            iii. T := F T1
            iv. T1:= * T | / T | ϵ
            v. F:= a | (E)
      c. Parse 1+2*(3+4)
      2. The representations are different
         a. But the languages recognized are the same
   C. What is an appropriate representation
      1. Do you care about T,F,T1,F1,E1,…?
      2. Do you care about parentheses?
      3. What about operators?
      4. How would you write an expression tree?

III. Abstract Syntax Trees
   A. Represent the underlying language
      1. Not the parse
      2. Not the quirks of the language or the grammar or the parse
   B. ASTs have multiple purposes
      1. Represent the program
         a. They are an intermediate representation
         b. Can be the basis for further computation
      2. Used for compilation / semantic analysis
         a. The tree should be designed for ease of use in compilation
            i. Why bother
One pass compilers rarely bother
But most languages these days aren’t one pass
  ▪ Need to go over the program multiple times to compute semantics
  ▪ ASTs give a compact, programmable representation that can be a basis for such computations
b. Can merge different ways of doing something into a single form
  i E.g. variable declarations
  ii E.g. function declarations in JavaScript
c. Generally need reference back to original source location (errors)
3. Used in IDEs for refactoring, editing, indentation, ...
  a. Might need to regenerate the original program
  i With comments, idiosyncratic syntax, etc.
C. Represent the syntax without the sugar
  1. They represent the actual meaning
  2. Thus they might be modified after the parse
     a. To handle ambiguous constructs
     b. To provide normalized representations
     c. To reflect the actual semantics
d. To show hidden operators
  i Where do these arise
e. To show implicit calls
  i Where do these arise
f. Can compress trees into dags at times
  i Common leaf nodes for common operations
D. Need to represent non-tree information
E. Annotated ASTs
  1. ASTs are annotated by associating properties with each node
     a. What information
        i Particular operators for a generic expression node
        ii Use-def linkages
        iii Type information for variables
     b. Values
        i Fields associated with the node to hold arbitrary values
           ▪ Some values are fixed (e.g. operator type)
           ▪ Some are complex references (variable being referenced)
        ii How to represent complex values
Two basic choices

2. Use separate semantic structures (attributed ASTs)
   a. Symbol table
   b. Type algebra (type table/structures)

3. Threaded ASTs
   a. Incorporate semantic structures in the AST
   b. AST contains links to other AST nodes
   c. Consider values (e.g. operators) as internal leafs

4. Pros and cons
   a. Symbols and types carry additional baggage
   b. Trees need to have real methods, not just store data
   c. Trees may be dags or graphs (harder to process, read/write)
   d. Single analysis mechanism suffices (print/analyze/code generation)
   e. Can be input-output easier
   f. Only one representation needs to be understood

F. Design principles for ASTs

1. Require insight into how the AST will be used
   a. Semantic analysis
   b. Intermediate code generation
   c. Program analysis
   d. Indentation, source code regeneration, ...

2. Hard to get right the first time
   a. Right = easiest to use for the purpose
   b. Least duplication of code

3. Objectives
   a. Reflect the underlying semantics (not the syntax)
   b. Provide a normalized representation
      i. Minimize different ways of doing things
   c. Provide hooks to store all the semantic information
   d. Allow semantic computation using tree walks
      i. Make this easy
   e. Regenerate (equivalent) source (JDT in Eclipse)

4. Generally an iterative process

G. Example: Decaf language

1. If then else – single statement
2. opExpression
3. new id(args) -> call(name(new(type),”|CTOR|”,expressionList)
4. super.id = name[name["super"],id]

IV. AST implementation
   A. General structure representing AST node
      1. Specialized for the various types of nodes
      2. Access to children
         a. Can be node-type specific (getLhs(), getRhs(), getOperator())
         b. Can be generic (getChild(i), getNumChildren())
         c. Can be both
      3. Access to parent (optional, but generally useful)
      4. Common properties
         a. Location in source (line/column, offset)
      5. Representing lists
         a. Can be implicit (variable number of children)
         b. Can be property-based (type of subtree is list of elements: eclipse)
         c. Can be explicit (require a separate tree node for the list)
   B. Specialize the nodes in a hierarchy
      1. Leaf nodes for actual constructs
      2. Intermediate type nodes for common features
         a. E.g. Expression
         b. E.g. Symbol, Scope, Type if threaded
         c. MemberDeclaration
      3. Common access methods can go in intermediate types
   C. Normalize the trees for compilation
      1. Can be done as part of generation
      2. Can be done as a post-pass (before or during semantic analysis)
      3. Can be a combination of the two
   D. Provide for computation over the ASTs
      1. Can be done by adding methods
      2. Easier to do using a visitor pattern
         a. accept(Visitor) on each AST node
         b. visit(NodeType) methods in the visitor
         c. visit children
            i. visit(Node) : preVisit(Node), visitChildren..., postVisit(Node)

V. Homework
   A. Read chapter 4