I. Introduction
   A. We assume that we have Abstract Syntax Trees
      1. Built by the parser or by some other tool
   B. We next want to do the appropriate semantic analysis
      1. This may mean annotating the ASTs with semantic information
         a. References from symbol to its definition
         b. Type for each expression
         c. May be threaded or external
      2. This may mean modifying the ASTs
         a. Normalization
            i. Single declaration per variable declaration
         b. Adding implicit operations, etc.
            i. Casts, calls to library functions (String +)
         c. Converting to low-level operations
            i. E.g. + => INT_ADD, FLOAT_ADD, DOUBLE_ADD, LONG_ADD
      3. This may mean generating a different representation
         a. One more appropriate for optimization or other program analysis
         b. One that represents the target language
      4. Note that semantic analysis is a form of program analysis
   C. We first want to consider how we might define such analyses
      1. Goal of the lecture – getting to think in terms of tree computations

II. History
   A. Goal:
      1. Allow a simple specification of the translation implemented by a
         compiler
         a. From source language
         b. To either target or intermediate language
         c. Including all the appropriate semantic information
      2. Should be non-procedural if possible (declarative)
      3. Should be a semantic specification for the language
a. Or a formal definition of the underlying analysis
b. Why?
c. What types of semantics are there?

B. Approaches
1. Syntax-directed translation with strings
   a. Example: Map expressions to polish form
   b. Example: derivatives
   c. How this works
2. Action routines
   a. Most general, least satisfying
   b. Structured by associated routine with a tree node
   c. Implemented using a tree walk of some sort (routine may call child routines)
3. 4GLs: special semantic languages
   a. Symbol table, type algebra, control flow, data flow
4. Generalized syntax-directed translations
   a. Multiple attributes
   b. Inherited attributes as well as synthesized ones
   c. Result is called an attribute grammar
   d. Example: Decl -> Type Elt; elt -> ID | [] elt; type-> INT | FLOAT

C. Attribute Grammars
1. Associate a set of attributes with each grammar symbol
   a. Nonterminal and terminal (in the AST)
   b. Attributes are typed (contain values)
   c. Attributes are either inherited or synthesized (not both)
      i. Other inherited examples
         ■ Compiler: want a Boolean expression for an IF
         ■ Languages where the return type matters
         ■ Compiler: current scope
2. Associate a set of rules with each production
   a. What is an AST production
   b. Rules must define all synthesized attributes of the LHS (root)
   c. Rules must define all inherited attributes of the RHS (children)
   d. Different production have different sets of rules
   e. What is the difficulty here
      i. What happens if the rhs sets depend on other rhs sets
3. What can go in the rules
a. String->string mappings (string operations)
b. General functions \(X.a = F(X.b,Y.a,Z.c)\)
c. Functional language expressions (no side effects)
d. Prolog-style denotational semantics

4. What are the trade-offs here

5. Example: what if we wanted to output the actual expression
   a. With parenthesis only where needed

III. Attribute Grammars in a Compiler

A. Example
   1. Grammar
      a. \(\text{program} ::= \text{PROGRAM ID ( id_list ) ; decls cmpdstmt}\)
      b. \(\text{id_list} ::= \text{ID \{ , ID }\)
      c. \(\text{decls} ::= \{ \text{VAR ID : type [ := expr ] ; }\)
      d. \(\text{type} ::= \text{INTEGER | REAL}\)
      e. \(\text{cmpdstmt} ::= \text{BEGIN \{ stmt \} END}\)
      f. \(\text{stmt} ::= \text{ID = expr}\)
      g. \(\text{expr} ::= \text{CONSTANT | ID}\)
   2. What we want to compute
      a. Scope/definition for each variable
   3. Define the scope as an inherited attribute
   4. Need to inherit from ID and type for declarations

B. Problems
   1. Need for performance in the compiler
   2. Need for scalability (large programs)
      a. All algorithms should be linear or close thereto
      b. This is true for most issues dealing with compilers
   3. Need to handle complex issues
      a. Symbol tables, lookup and definition rules
      b. Type algebras and conversion rules
      c. Operator overloading rules

C. Symbol tables
   1. The effective symbol table varies from point to point in a program
      a. Definitions are added one by one in most languages today
         i. Consider C++ with multiple declarations
         ii. Does each declaration start a new scope?
      b. Symbol availability changes over time
         i. Sometimes not monotonically
2. Thus an attribute grammar must pass around an environment
   a. One or more attributes
   b. Containing the symbol table
   c. Pass to each node (in and out)
3. This has to be treated functionally
   a. All instances available at all times
4. Doable for simple languages, much more difficult as the rules get more complex
5. Types and overloading can be worse

IV. Attribute Evaluation
A. Suppose we set up an attribute grammar, how can the attributes be evaluated
   1. Is left-to-right order or depth-first tree walk sufficient?
   2. Can the grammar have cycles?
      a. Is this an error, or do we do fix-point computations

B. Evaluation schemes
   1. Left-to-right, single pass
      a. Requires a restricted attribute grammar
      b. L-attributed (allows synthesized and inherited attributes)
      c. S-attributed (synthesized attributes only)
   2. Left-to-right multiple passes
      a. Do whatever can be done on each pass
      b. Problem: can require O(N) passes, n**2 algorithm
   3. Alternative left-to-right and right-to-left passes
      a. Handle most grammars in a constant number of passes
      b. Can still construct bad examples
      c. Requires checking what needs to be done (a flag for each attribute)
      d. But relatively simple to implement
   4. Dynamic evaluation ordering
      a. Build a dependency graph among the attributes
      b. Topologically sort the attributes based on this
      c. Evaluate in the resultant order
   5. Precompile an evaluation strategy
      a. For each production, generate a plan for evaluation
         i. Step of plan is to either evaluate a particular attribute
         ii. Or compute(rhs_element)
         iii. Or return
b. Dependent only the production, not its children
   i. Have to treat the children with same LHS similarly
   ii. Have to merge constraints form different rules

c. Algorithms to do this exist
   i. Rather messy, non-linear
   ii. But only done at generation time, not at compile time

d. Different algorithms restrict the grammar in various ways

C. Tools are needed
   1. You don’t want to do any of this manually
   2. Use whatever your tools support

V. Current Approaches

A. Attribute Grammars
   1. Not widely used in real compilers
   2. Too slow, complex, unsuitable for language complexity
   3. Not the cleanest framework for a whole compiler
   4. May be appropriate for simpler program analyses driven off the AST
   5. Useful in formally defining semantics

B. Attribute grammar variants
   1. What types of formulas to allow in the rules
   2. Some work with prolog-type rules (building a knowledge base)
   3. Logic is in a sense defined in the knowledge base
   4. Rule ordering may be defined there as well
   5. Closer to formal semantic specifications

C. Instead implement semantics by ordered tree walks
   1. Using outside data structures (implicitly as part of the AST or explicitly)
      a. As part of visitor
   2. Different orderings are possible, but generally
      a. Previsit node (visit)
      b. Visit each child in turn (left to right) [order may be changed]
      c. Postvisit node (endVisit)

D. Object-oriented languages make this clean
   1. Each AST node is a class
   2. Action routine differs by class
   3. Can do different types of walks to get different values
   4. Can think of this as attribute grammars
      a. Evaluation strategies are similar
      b. Specification is done in small steps, on a production basis
E. Typically a VISITOR pattern is used
   1. Don’t put the analysis routines in the AST per se
   2. Instead each analysis is handled by a visitor
   3. One method of the visitor for each type of AST node