Chapter 8 Treewalking Passes and Attribute Evaluation

Division into passes

Once we start developing a real language, we have to process declarations, and perform name lookup and type checking. We end up with multiple recursive treewalking passes through the abstract syntax tree.

Typical passes could include:

- A lexical analysis and parsing phase to process the input to build the abstract syntax tree.
- A treewalk to reprint the program. (Just to check it has been parsed correctly.)
- A treewalk to build a "compile-time environment" or "symbol table", containing information about the declarations in each block.
- A treewalk to map type constructs to data structures representing "types", and fill in the type information for declarations. This includes looking up class type names in the compile-time environment to map them to their declaration.
- A treewalk to determine the types of expressions. This includes looking up variable and method names in the compile-time environment to map them to their declaration.
- A treewalk to determine the run-time offset of each variable (and perhaps method) from an appropriate base pointer. (At run-time, variables are accessed as an offset from the base of a block of memory, such as the activation record for the current method.)
- If writing an interpreter, an evaluation treewalk, to interpret the program. This pass involves the creation of a run-time environment, containing the values of variables.

  The compile-time environment is used to determine how to access the run-time environment. Some constructs might be evaluated multiple times (due to the execution of a loop, or a recursive method), and some never evaluated. The run-time environment might be different for each evaluation.
- If writing a real compiler, a code generation treewalk, to generate assembly language. The assembly language can later be assembled, to generate machine code, and the machine code can be executed. The execution of the machine code will create a run-time environment, containing the values of variables.

We could combine the building of the compile-time environment, mapping of identifiers to declarations, type checking, and determination of offsets into a single pass. However, this would limit our ability to process recursively declared methods and data structures. By making sure we put information about a class or method declaration in the compile-time environment before we process the body, we can cope with simple recursion. However, we cannot cope with mutually recursive class types or methods. For that, we need more than one pass. It is also a good idea to separate different tasks into different passes, to simplify the structure of our compiler

Representation of the Abstract Syntax Tree

To implement any computer language, we first perform lexical analysis and parsing, to build a tree.

For each nonterminal in the grammar, we declare an abstract class. (However, nonterminals that correspond to the same kind of entity, and only differ in terms of precedence, will share the

same abstract class. For example, all nonterminals corresponding to different precedence expressions might correspond to “ExprNode”.) The fields of this abstract class represent attributes that we expect all constructs corresponding to the nonterminal to have. For example, all expressions have a type and precedence. The public methods of this abstract class return information about the construct or represent computations we expect to be able to perform on all constructs corresponding to the nonterminal.

For each grammar rule, we declare a concrete class that extends the abstract class corresponding to the nonterminal on the left hand side of the rule. The concrete class has additional fields such as references to the sub-constructs that make up the right hand side of the rule.

(Actually, sometimes users declare nonterminals with a single grammar rule. In this case, it is sensible to have a concrete class corrsponding to both the nonterminal and rule. There are other situations in which we might have a single concrete class. For example, a nonterminal corresponding to a list could be represented by a single concrete class containing the list of components.)

The nodes of the tree generated by the parser correspond to the constructs that occur in the program being compiled. Each node is an instance of the concrete class corresponding to the grammar rule used to match the construct. (However, we omit the nodes that correspond to nothing but “syntactic sugar”, such as parenthesizing of expressions, etc.)

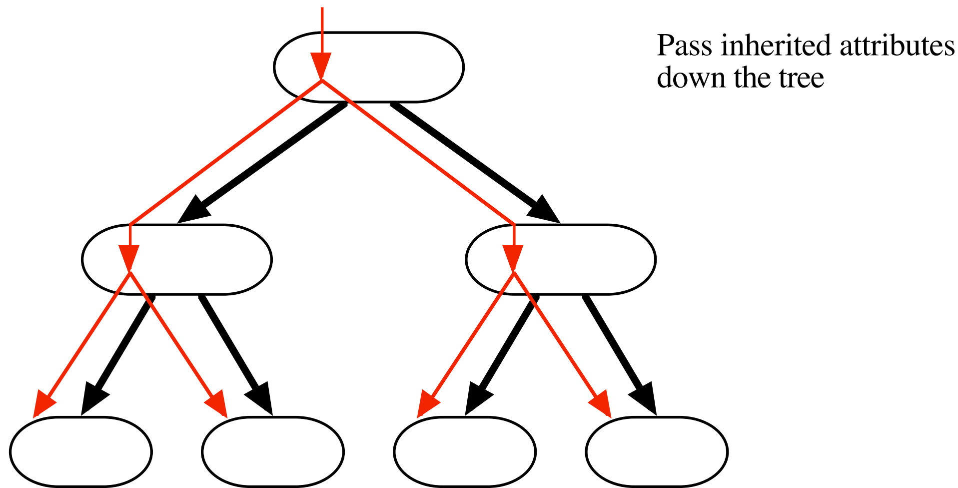
Treewalking

After creation of the tree, we perform a number of treewalks. Each pass through the tree computes attributes for the nodes of the tree, and builds tables. The final pass either interprets the program or generates code for later execution. The division into passes is primarily based on the interdependency of the attributes. For example, we have to first build compile-time environment information indicating the declarations in each block, then perform mapping of identifier applications to declarations, then perform type checking of expressions, and then finally either interpret the program or generate code. If we do not build the environment information before mapping identifiers to declarations, we will not cope with mutually recursive declarations. No matter which declaration is processed first, we will refer to the other declaration before it is declared. We might also split some unrelated analyses into separate passes to simplify the structure of the compiler, and decrease the amount of information we have to deal with at one time. Yet another reason for having multiple passes is that we can collect more information about the program before generating code, and hence generate better quality code.

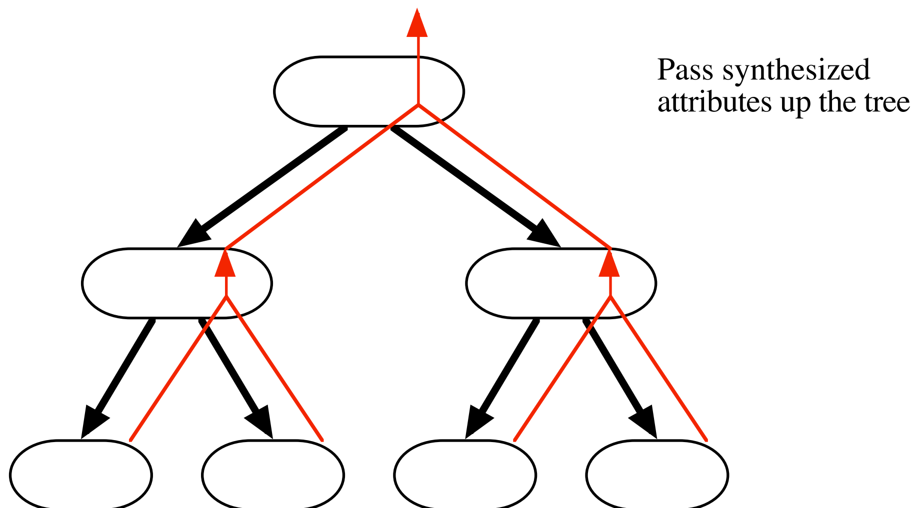
Evaluation of Attributes

Attributes can usually be classified as **inherited**, **synthesized**, or **threaded**.

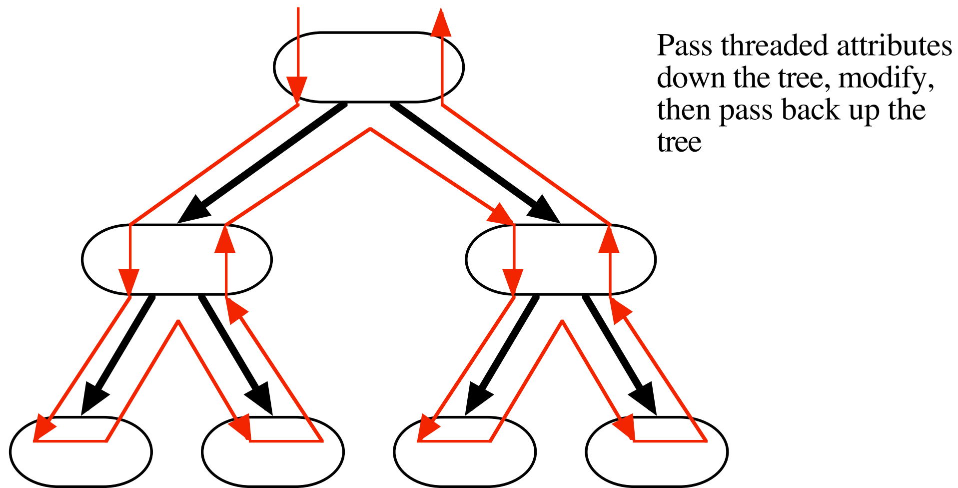
If we **pass information down the tree**, so that the attribute is computed from that of the parent, then the attribute is said to be **inherited**. For example, when processing statements, the compile-time environment (a table containing information on the declarations currently accessible), is to a large extent inherited from the parent node. Inherited attributes can be passed down to the children, by recursively invoking the treewalking method with the attribute as a value parameter.



If we **pass information up the tree**, so that the attribute is computed from that of the children, then the attribute is said to be **synthesized**. For example, the type of an expression is synthesized from the types of the children. The textual representation of a construct is synthesized from the textual representation of the children. The code generated for a construct is also largely synthesized from that of the children. Synthesized attributes can be returned to the parent as the return value of the treewalking method.



If we **pass information down the tree, modify the attribute, then pass a modified value back to the parent**, then the attribute is said to be **threaded**. For example, when processing a local block, the list of declarations for the block is threaded through the declarations, which add information about that declaration to the list. Threaded attributes can be passed down to the children, by recursively invoking the treewalking method with the attribute as a reference parameter. On return from the treewalking method, the reference attribute will be modified.



Another way of dealing with threaded attributes is to have a global variable that is modified by the treewalking method. For example, when generating labels for control statements, etc, it is convenient to have a class containing the next available label number as a static variable, and to increment this each time a new label number is needed.