Semantic Analysis

Introduction to Semantic Analysis

1. We know we have a correctly formatted sentence, but is it meaningful?
   - Need to perform type and other analysis to determine the soundness of the program
     - Not adding integers and characters
   - This phase and next phase are called the middle-end

Beyond Syntax

1. To generate code, the compiler needs to answer many questions
   - Is “x” a scalar, an array, or a function? Is “x” declared?
   - Are there names that are not declared?
   - Declared but not used?
   - Is the expression “x * y + z” type consistent?
   - In “a[i,j,k]”, does a have three dimensions?
   - Where can “z” be stored? (register, local, global, heap, static)
   - How many arguments does “fie()” take?
   - Do “p” & “q” refer to the same memory location?
   - Is “x” defined before it is used?
   - All of these questions go beyond a context-free grammar
     - Require context sensitive grammars
2. Context-sensitive analysis
   - Answers depend on values, not parts of speech
   - Questions & answers involve non-local information
   - Answers may involve computation

Syntax Problems

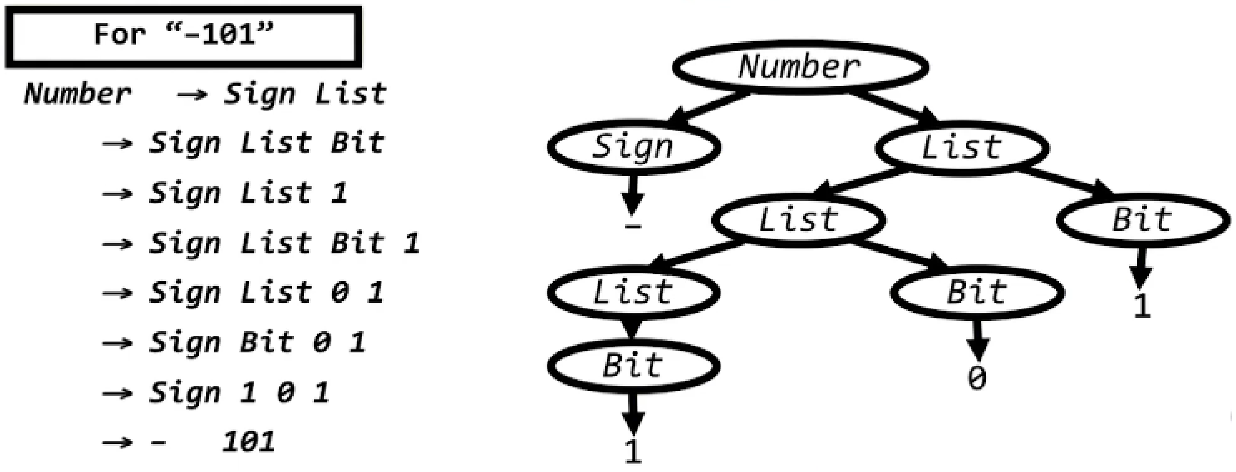
1. How can we answer these questions?
   - Use formal methods
     - Context-sensitive grammars?
       - Tells us the context of the derivation
       - Too expensive
     - Attribute grammars?
       - Syntactic structures with some additions of the attributes
   - Use ad-hoc techniques
     - Symbol tables
     - Ad-hoc code (action routines)
2. We will study the formalism - an attribute grammar
   - Clarify many issues in a succinct and immediate way
   - Separate analysis problems from their implementations
     - Expensive
     - Many compilers use symbol tables to avoid propagating information
   - Information is immunization
3. We will see that the problems with attribute grammars motivate actual, ad-hoc practice
   - Non-local computation
   - Need for centralized information
   - Some people still argue for attribute grammars

Attribute Grammar Example

1. What is an attribute grammar?
   - A context-free grammar augmented with a set of rules

- Each symbol in the derivation (or parse tree) has a set of named values, or attributes
- The rules specify how to compute a value for each attribute
- Attribution rules are functional; they uniquely define the value

2. Example grammar:
   - Number -> Sign List
   - Sign -> + | -
   - List -> List Bit | Bit
   - Bit -> 0 | 1



```
For "-101"
Number  → Sign List
        → Sign List Bit
        → Sign List 1
        → Sign List Bit 1
        → Sign List 0 1
        → Sign Bit 0 1
        → Sign 1 0 1
        → -   101
```

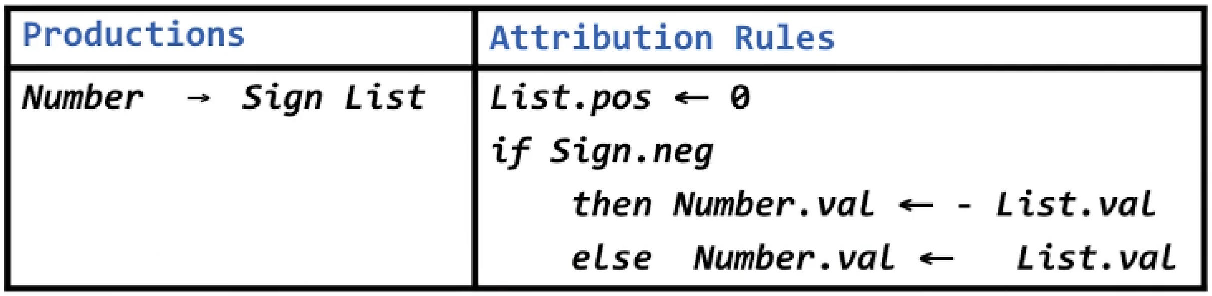
Attribute Grammar

Attribute Grammar Part 2

1. Add attributes to this grammar

Symbol	Attributes
Number	val
Sign	neg
List	pos, val
Bit	pos, val

2. Map productions to attribution rules

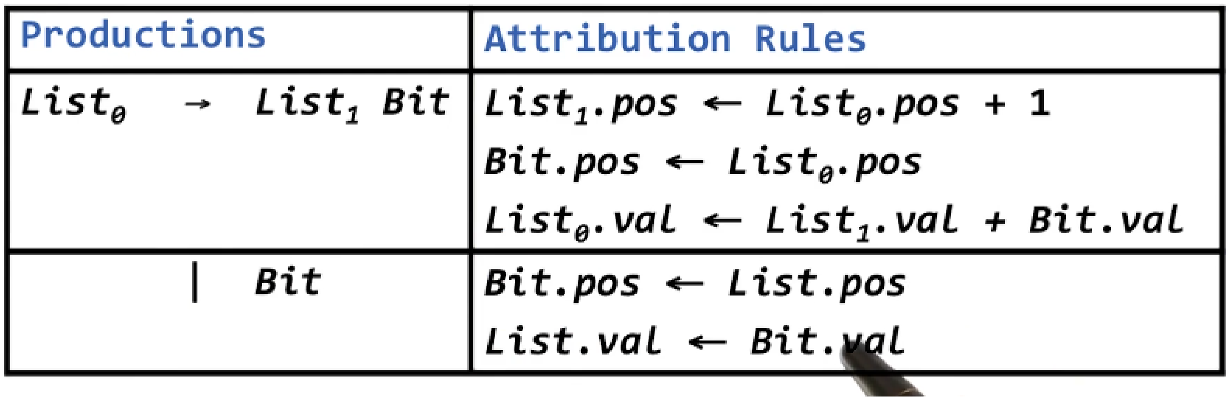


Productions	Attribution Rules

Number → Sign List	*List.pos ← 0*
	if Sign.neg
	then Number.val ← - List.val
	else Number.val ← List.val

Attribute Rules Example 1

Attribute Grammar Part 3

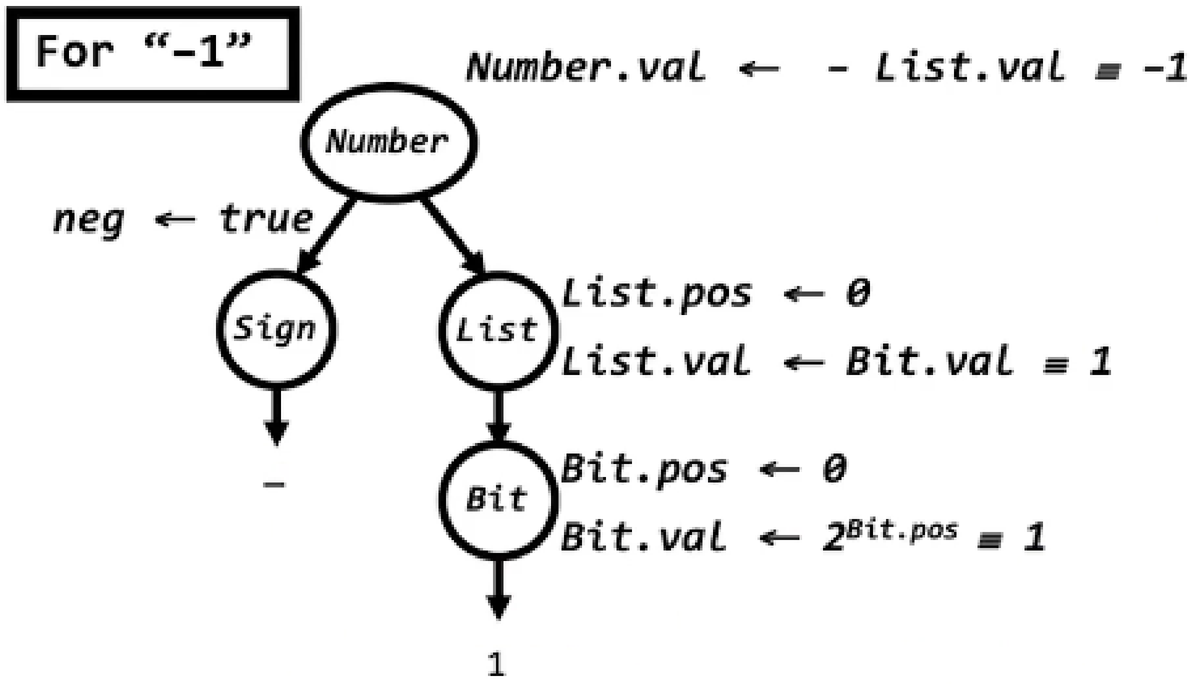


Productions	Attribution Rules
$List_0 \rightarrow List_1\ Bit$	$List_1.pos \leftarrow List_0.pos + 1$ $Bit.pos \leftarrow List_0.pos$ $List_0.val \leftarrow List_1.val + Bit.val$
$\mid\ Bit$	$Bit.pos \leftarrow List.pos$ $List.val \leftarrow Bit.val$

Attribute Rules Example 2

Attribute Grammar Example

1. Rules + parse tree imply an attribute dependence graph
2. One possible evaluation order:
   - List.pos
   - Sign.neg
   - Bit.pos
   - Bit.val
   - List.val
   - Number.val



Attribute Grammar Example

Attribute Grammar Other Methods

1. Other orders are possible!
    - Attribution rules are simply telling us the dependence
        - Not teling us a particular order
    - Evaluation order must be consistent with the attribute dependence graph
2. Knuth suggested a data-flow model for evaluation:
    - Independent attributes first
    - Others in order as input values become available
    - Guarantees correctness

Rules of Attribute Grammars

1. Types of attributes
    - Synthesized attribute: Depends on values from children
    - Inherited attribute: Depends on values from siblings and parents
2. Processing rules
    - Associate attributes with nodes in a parse tree
    - Rules are value assignments associated with productions
    - Attributes are defined once, using local information
    - Label identical terms in production for uniqueness
    - Rules and parse trees define an attribute dependence graph
        - Graphs must be non-circular
3. Rules of attribute grammars
    - This produces a high-level, functional specification
    - Attribute grammar is a specification for the computation, not an algorithm

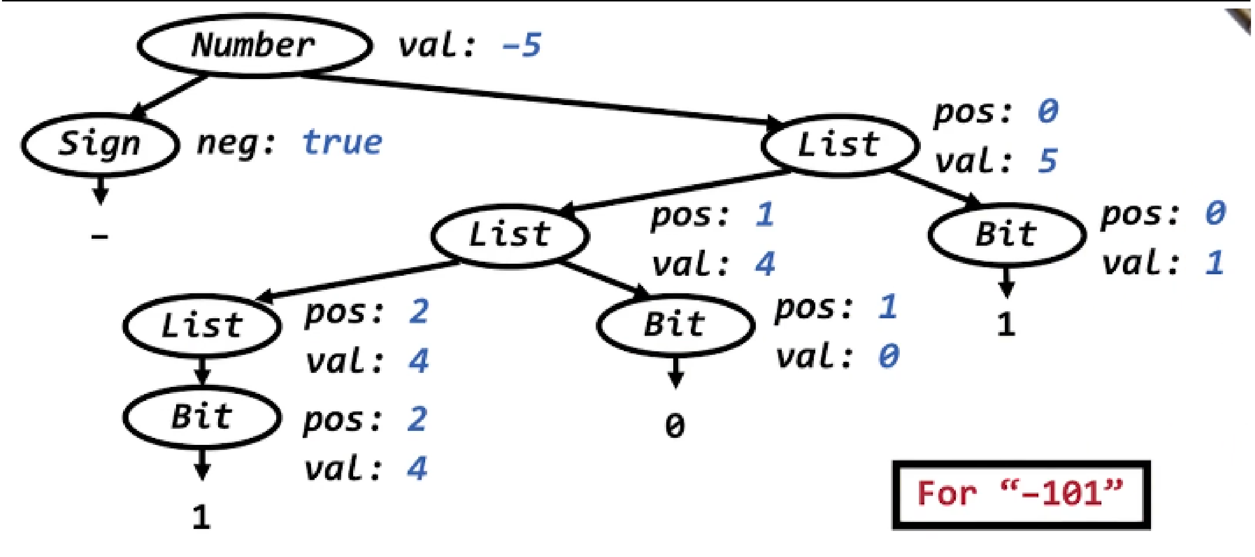
Using Attribute Grammars

1. Attribute grammars can specify context-sensitive actions:
   - Take values from syntax
   - Perform computations with values
   - Insert tests, logic, ...
2. Synthesized attributes
   - Use values from children & constants
   - S-attributed grammars
   - Evaluate in a single bottom-up pass
3. Inherited attributes
   - Use values from parent, constants, & siblings
   - Directly express context
   - Can rewrite to avoid them
   - Thought to be more natural
   - Not easily done at parse time

Evaluation Methods

1. Dynamic, dependence-based methods
   - Build the parse tree
   - Build the dependence graph
   - Topological sort the dependence graph
   - Define attributes in topological order
2. Rule-based methods (treewalk)
   - Analyze rules at compiler-generation time
   - Determine a fixed (static) ordering
   - Evaluate nodes in that order
3. Oblivious methods (passes, dataflow)
   - Ignore rules and parse tree
   - Pick a convenient order (at design time) and use it

Tree Example Inherited Attributes



Attributed Syntax Tree

Tree Example Synthesized Attributes

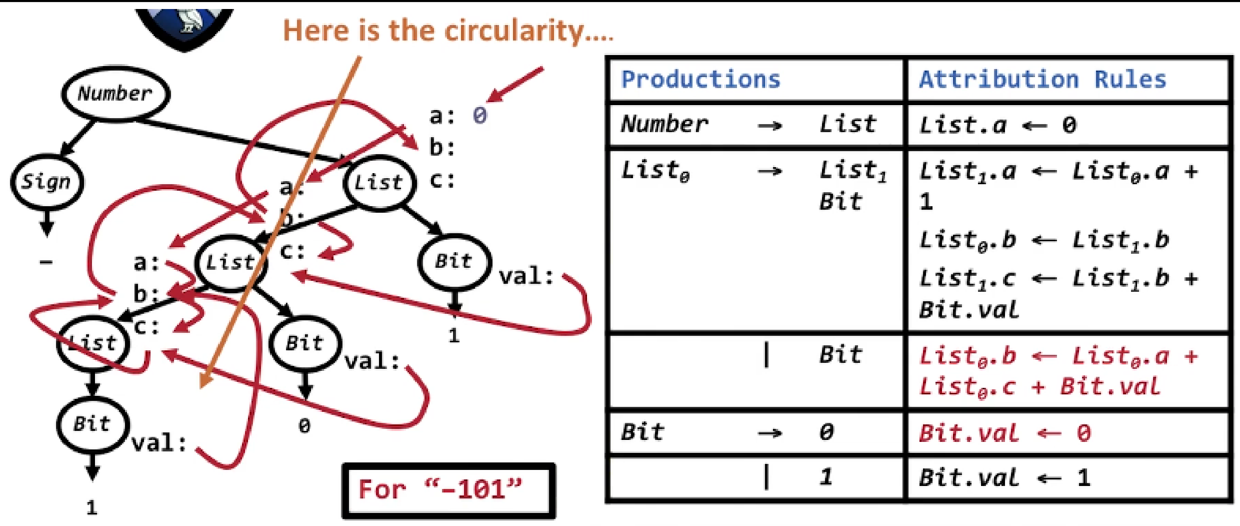
1. Synthesized attributes
   - Attribute of a particular node depends on the attribute of its children
     - Val draws from children and the same node
   - Flow from bottom to top
2. Inherited attributes
   - Flow from top to bottom
3. Attribute dependence graph
   - The dependence graph must be acyclic
   - Dynamic methods sort this graph to find independent values, then work along graph edges
   - Rule-based methods try to discover "good" orders by analyzing the rules
   - Oblivious methods ignore the structute of the graph

Circularity

1. We can only evaluate acyclic instances
   - General circularity testing problem is inherently exponential
2. We can prove that some grammars can only generate instances with acyclic dependence graphs
   - Largest such class is "strongly non-circular" grammars (SNC)
   - SNC grammars can be tested in polynomial time
   - Failing the SNC test is not conclusive
   - Many evaluation methods discover circularity dynamically
     - Bad property for a compiler to have

Circular Attribute Grammar

1. Adding these new rules generates a circular dependence



Productions	Attribution Rules
Number → List	List.a ← 0
$List_0$ → $List_1$ Bit	$List_1$.a ← $List_0$.a + 1 $List_0$.b ← $List_1$.b $List_1$.c ← $List_1$.b + Bit.val
| Bit	$List_0$.b ← $List_0$.a + $List_0$.c + Bit.val
Bit → 0	Bit.val ← 0
| 1	Bit.val ← 1

Circular Dependence in Grammar

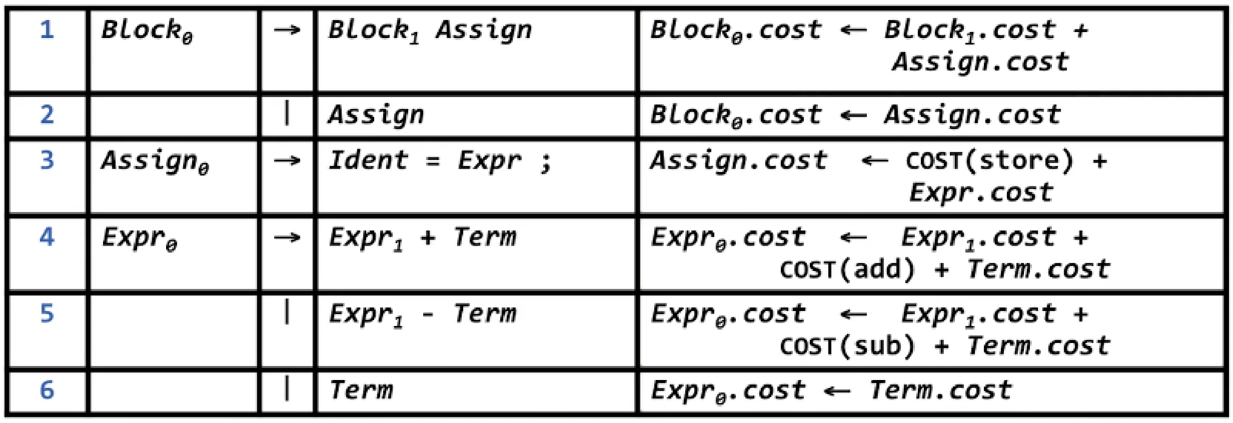
Circularity: The Point

1. Circular grammars have indeterminate values
   - Algorithmic evaluators will fail
   - Noncircular grammars evaluate to a unique set of values
   - Circular grammar might give rise to non-circular instance

    - Probably shouldn't bet the compiler on it...
2. Should (undoubtedly) use provably non-circular grammars
3. Remember, we are studying AGs to gain insight
    - We should avoid circular, indeterminate computations
    - If we stick to provably non-circular schemes, evaluation should be easier

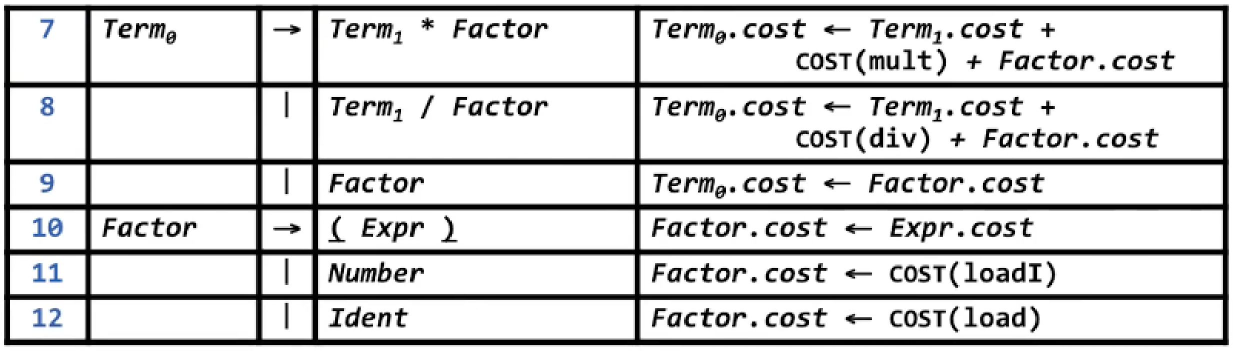
Attribute Grammar Example

1. Let's estimate cycle counts
    - Each operation has a COST
    - Add them, bottom up
    - Assume a load per value
    - Assume no reuse
    - Simple problem for an attribute grammar



1	Block$_0$	→	Block$_1$ Assign	Block$_0$.cost ← Block$_1$.cost + Assign.cost
2		|	Assign	Block$_0$.cost ← Assign.cost
3	Assign$_0$	→	Ident = Expr ;	Assign.cost ← COST(store) + Expr.cost
4	Expr$_0$	→	Expr$_1$ + Term	Expr$_0$.cost ← Expr$_1$.cost + COST(add) + Term.cost
5		|	Expr$_1$ - Term	Expr$_0$.cost ← Expr$_1$.cost + COST(sub) + Term.cost
6		|	Term	Expr$_0$.cost ← Term.cost

Cost for a Basic Block



7	Term$_0$	→	Term$_1$ * Factor	Term$_0$.cost ← Term$_1$.cost + COST(mult) + Factor.cost
8		|	Term$_1$ / Factor	Term$_0$.cost ← Term$_1$.cost + COST(div) + Factor.cost
9		|	Factor	Term$_0$.cost ← Factor.cost
10	Factor	→	(Expr)	Factor.cost ← Expr.cost
11		|	Number	Factor.cost ← COST(loadI)
12		|	Ident	Factor.cost ← COST(load)

Cost for a Basic Block

2. Properties of the example grammar
    - All attributes are synthesized S-attributed grammar
    - Rules can be evaluated bottom-up in a single pass
        - Good fit to bottom-up, shift/reduce parser
    - Easily understood solution
    - Seems to fit the problem well

Moral of the Story

1. Non-local computation needed lots of supporting rules
2. Complex local computation was relatively easy
3. The problems
   - Copy rules increase cognitive overhead
   - Copy rules increase space requirements
     - Need copies of attributes
     - Can use pointers, for even more cognitive overhead
4. A good rule of thumb is that the compiler touches all the space it allocates, usually multiple times
   - Result is an attributed tree (somewhat subtle points)
     - Must build the parse tree
     - Either search tree for answers or copy them to the root

Synthesized vs Inherited Attributes Quiz

1. Please fill in the boxes indicating whether the attributes in the following examples are synthesized (S) or inherited (I) or neither (N) or both (B):
   - A user defined type from a language's base type (such as float or int), the language follows structural type equivalence which says that two values are type equivalent provided it can be inferred that their base types and structures are equivalent
     - Inherited
   - Result of a typecast operation such as in : (int*) c where c is defined as : char* c; a pointer to a char
     - Synthesized
   - Type compatibility check as in <var> = <expr> using <var>'s type and <expr>'s type
     - Both

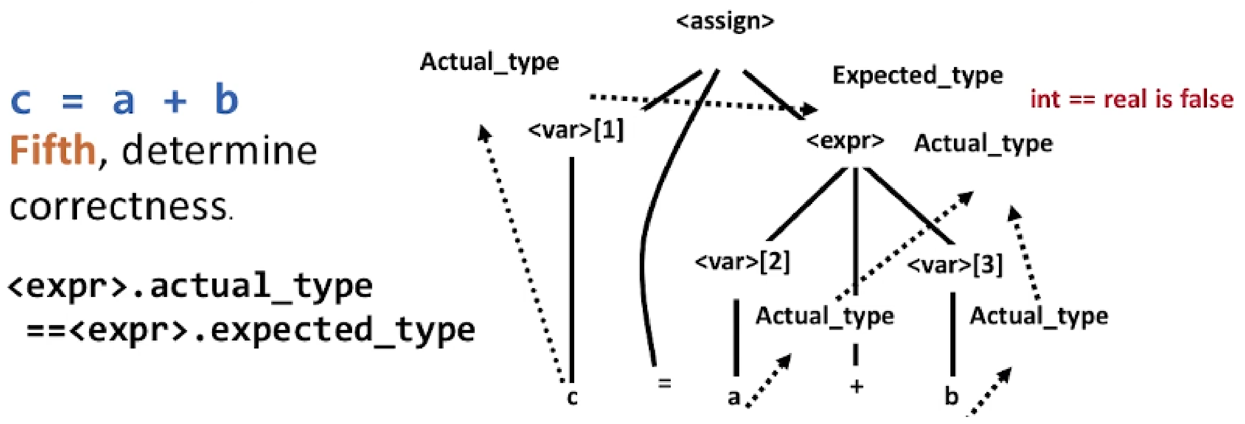
Attributes Grammars Quiz

1. Consider the following attribute grammar:
   - <assign> -> <var> [1] = <expr>
   - <expr> -> <var> [2] + <var> [3]
2. Semantic rule
   - <expr>.expected_type <- <var> [1].actual_type
   - <expr>.actual_type <-
     - if ([2].actual_type == int) and ([3].actual_type == int)
       - int
     - else
       - real
     - endif
   - <var>.actual_type <- lookup
     - (.string) // look-up returns 's type as per its declaration from the symbol table
   - Predicate:
     - .actual_type == <expr.expected_type
3. Consider the following declaration and assignment statements:
   - int a, c;
   - real b;
   - c = a + b;
4. Determine the following by filling in the boxes:
   - <var>[1].actual_type
     - Value: int
     - Synthesized or inherited: inherited
   - <var>[2].actual_type
     - Value: int

    - Synthesized or inherited: inherited
- <var>[3].actual_type
    - Value: real
    - Synthesized or inherited: inherited
- <expr>.actual_type
    - Value: real
    - Synthesized or inherited: synthesized
- <expr>.expected_type
    - Value: int
    - Synthesized or inherited: inherited
- Result of the predicate (true/false)
    - Value: false
    - Synthesized or inherited: both

Attributes Grammars Quiz Detailed Explanation

1. Create an attributed parse tree for: c = a + b where a and c is int and b is real
    - First, create the parse tree
    - Second, add intrinsic attributes
    - Third, obtain an inherited attribute
    - Fourth, obtain a synthesized attribute
    - Fifth, determine correctness



c = a + b

Fifth, determine correctness.

<expr>.actual_type ==<expr>.expected_type

Attribute Grammars Quiz