Semantic Analysis

- **Parser verifies that a program is syntactically correct and constructs a syntax tree (or other intermediate representation).**
- **Semantic analyzer checks that the program satisfies all other *static* language requirements (is “meaningful”) and collects and computes information needed for code generation.**

Semantic Analysis Tasks

- Have variables been declared before use?
- Have variables been declared twice in the same scope?
- Has every declared variable been used?
- Are the variable and expression in an assignment type-compatible?
- Do the operands of (arithmetic) operators have compatible types?
- Do the arguments in a function call match the parameters of the function definition in number and type?
- Are arguments passed by reference variables?

Important Semantic Information

- **Symbol table: collects declaration and scope information to satisfy “declaration before use” rule, and to establish data type and other properties of names in a program.**
- **Data types and type checking: compute data types for all typed language entities and check that language rules on types are satisfied.**

How to build the symbol table and check types:

- **Analyze the scope rules for the language and determine an appropriate table structure for maintaining this information.**
- **Analyze the type requirements and translate them into rules that can be applied recursively on a syntax tree.**

Theoretical framework for semantic analysis

- **Focus on *attributes*: computable properties of language constructs that are needed to satisfy language requirements and/or generate code**
- **Describe the computation of attributes using *equations* or algorithms.**
- **Associate these equations to grammar rules and/or kinds of nodes in a syntax tree.**

- **Analyze the structure of the equations to determine an order in which the attributes can be computed. (Tree traversals of syntax tree - preorder, postorder, inorder, or some combination of them.)**

- Such a set of equations, functions and conditions is called an **attribute grammar**.
- Formally describing the evaluation of attributes and the conditions that attributes must satisfy using an attribute grammar helps significantly, even if attribute grammar tools are not used for semantic analysis.
- Tools such as GAG and Eli generate semantic analysers from attribute grammar specifications.
- Tools such as Yacc implicitly use attribute grammars in their semantic actions.

Example of an attribute grammar

Grammar:

$exp \rightarrow exp$ **+** $term \mid exp$ **-** $term \mid term$

$term \rightarrow term$ ***** $factor \mid factor$

$factor \rightarrow$ **(** exp **)** $\mid$ ***number***

Attribute Grammar:

GRAMMAR RULE	SEMANTIC RULES
$exp_1 \rightarrow exp_2$ **+** $term$	$exp_1.val = exp_2.val + term.val$
$exp_1 \rightarrow exp_2$ **-** $term$	$exp_1.val = exp_2.val - term.val$
$exp \rightarrow term$	$exp.val = term.val$
$term_1 \rightarrow term_2$ ***** $factor$	$term_1.val = term_2.val * factor.val$
$term \rightarrow factor$	$term.val = factor.val$
$factor \rightarrow$ **(** exp **)**	$factor.val = exp.val$
$factor \rightarrow$ **number**	$factor.val =$ **number**$.val$

Notes:

- **Different instances of same nonterminal must be subscripted to distinguish them.**
- **Some attributes must have been precomputed (by scanner or parser), e.g. *number*.val.**
- **These particular attribute equations look a lot like a yacc specification, because they represent a *bottom-up* attribute computation.**

A Second Example

Grammar:

decl → *type var-list*
type → **`int`** | **`float`**
var-list → ***`id`*** **,** *var-list* | ***`id`***

Attribute Grammar:

GRAMMAR RULE	SEMANTIC RULES
decl → *type var-list*	*var-list.dtype* = *type.dtype*
type → **`int`**	*type.dtype* = *integer*
type → **`float`**	*type.dtype* = *real*
$var\text{-}list_1$ → ***`id`*** **,** $var\text{-}list_2$	***`id`*** *.dtype* = $var\text{-}list_1.dtype$ $var\text{-}list_2.dtype$ = $var\text{-}list_1.dtype$
var-list → ***`id`***	***`id`*** *.dtype* = *var-list.dtype*

Notes

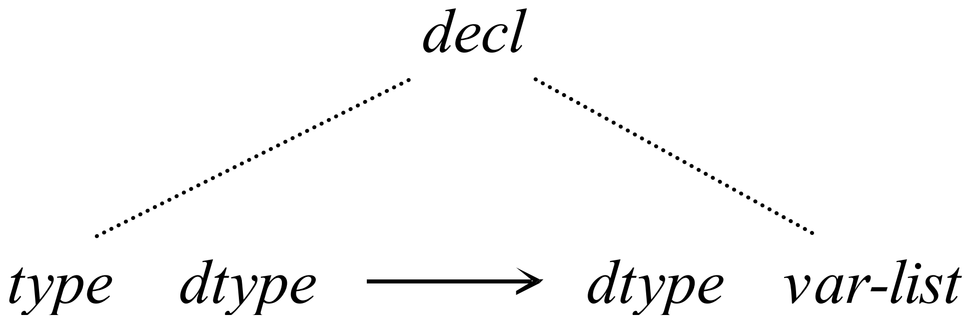
- **Data type typically propagates *down* a syntax tree via declarations.**
- **No longer something yacc can handle directly.**
- **Such an attribute is called *inherited*, while bottom-up calculation is called *synthesized.***
- **Syntax tree is a standard synthesized attribute computable by yacc; other attributes computed on the tree.**

Dependency graph

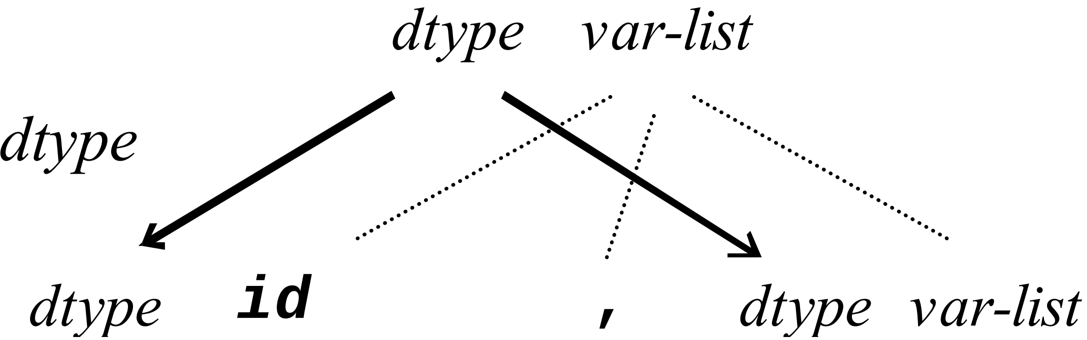
- **Indicates order in which attributes must be computed.**
- **Synthesized attributes always flow from children to parents, and can always be computed by a postorder traversal.**
- **Inherited attributes can flow any other way.**
- ***L-attributed*: a left-to-right traversal suffices to compute attributes. However, this may involve a combination of pre-order, inorder, and postorder traversal.**

Data type dependencies (by grammar rule):

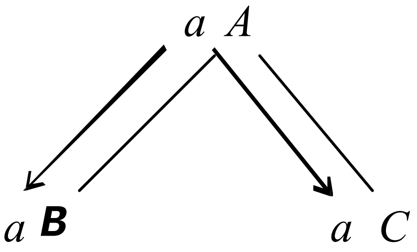
decl → *type var-list:*
var-list.dtype = type.dtype



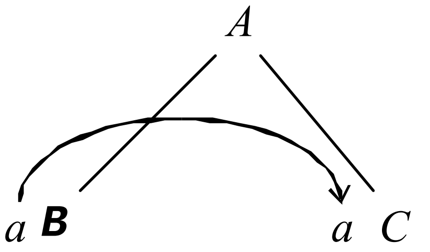
var-list → **id ,** *var-list:*
id *.dtype = var-list*$_1$*.dtype*
var-list$_2$*.dtype = var-list*$_1$*.dtype*



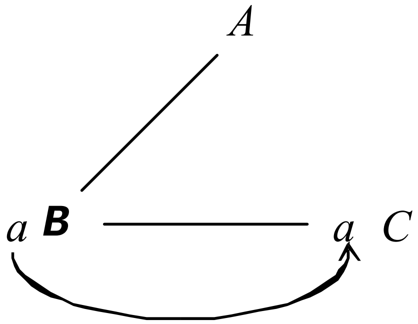
L-attributed dependencies have three basic mechanisms:



(a) Inheritance from parent to siblings



(b) Inheritance from sibling to sibling via the parent



(c) Sibling inheritance via sibling pointers

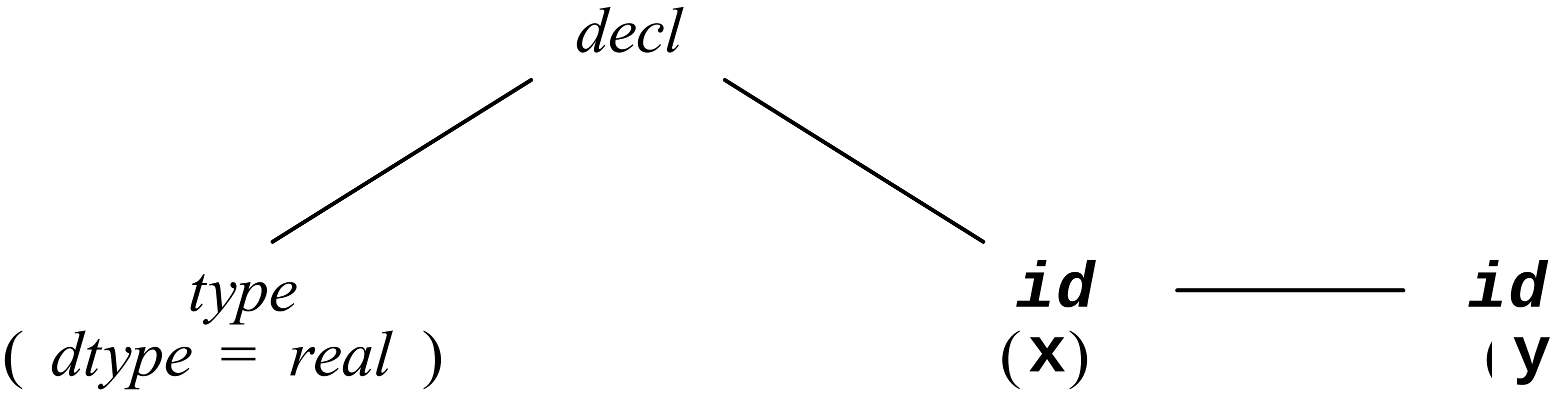
Sample tree structure:

```
typedef enum {decl,type,id} nodekind;
typedef enum {integer,real} typekind;
typedef struct treeNode
  { nodekind kind;
    struct treeNode
       * lchild, * rchild, * sibling;
    typekind dtype;
    /* for type and id nodes */
    char * name;
    /* for id nodes only */
  } * SyntaxTree;
```

Sample tree instance:

String: **`float x, y`**

Tree:



decl

type
(*dtype* = *real*)

`id`
(**`x`**)

`id`
(**`y`**

Traversal code:

```
void evalType (SyntaxTree t)
{ switch (t->kind)
  { case decl:
        t->rchild->dtype = t->lchild->dtype;
        evalType(t->rchild);
        break;
      case id:
        if (t->sibling != NULL)
        { t->sibling->dtype = t->dtype;
          evalType(t->sibling);
        }
        break;
  } /* end switch */
} /* end evalType */
```

Attributes need not be kept in the syntax tree:

GRAMMAR RULE	SEMANTIC RULES
decl → *type var-list*	
type → **int**	*dtype* = *integer*
type → **float**	*dtype* = *real*
var-list$_1$ → ***id*** **,** *var-list*$_2$	*insert*(***id*** .*name*, *dtype*)
var-list → ***id***	*insert*(***id*** .*name*, *dtype*)

dtype is global

Use a symbol table to store the type of each identifier

New traversal code:

```
typekind dtype; /* global */
void evalType (SyntaxTree t)
{ switch (t->kind)
  { case decl:
        dtype = t->lchild->dtype;
        evalType(t->rchild);
        break;
    case id:
        insert(t->name,dtype);
        if (t->sibling != NULL)
          evalType(t->sibling);
        break;
  } /* end switch */
} /* end evalType */
```

Even better, use a parameter instead of a global variable:

```
void evalDecl(SyntaxTree t)
{  evalType(t->rchild, t->lchild->dtype);
}
void evalType(SyntaxTree t, typekind dtype)
{ insert(t->name,dtype);
  if (t->sibling != NULL)
        evalType(t->sibling,dtype);
}
```

Note: inherited attributes can often be turned into parameters to recursive traversal functions, while synthesized attributes can be turned into returned values.

Alternative to a difficult inherited situation (not recommended):

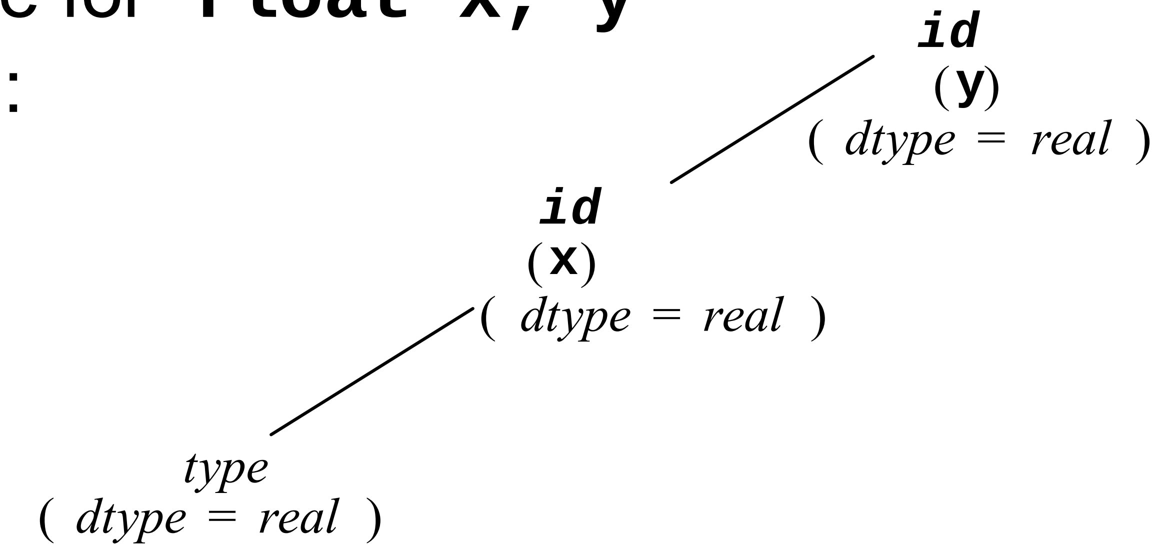
Theorem (Knuth [1968]). Given an attribute grammar, all inherited attributes can be changed into synthesized attributes by suitable modification of the grammar, without changing the language of the grammar.

Example:

New grammar for types:

decl → *var-list* ***id***
var-list → *var-list* ***id*** **,** | *type*
type → **int** | **float**

New Tree for **float x, y** might be:



id
(**y**)
(*dtype* = *real*)

id
(**x**)
(*dtype* = *real*)

type
(*dtype* = *real*)

Our approach:

- Compute inherited stuff first (symbol table) in a separate pass
- Then type inference and type checking turns into a purely synthesized attribute computation, since all uses of names have their types already computed.
- Next:
  - Symbol table structure
  - Synthesized type rules