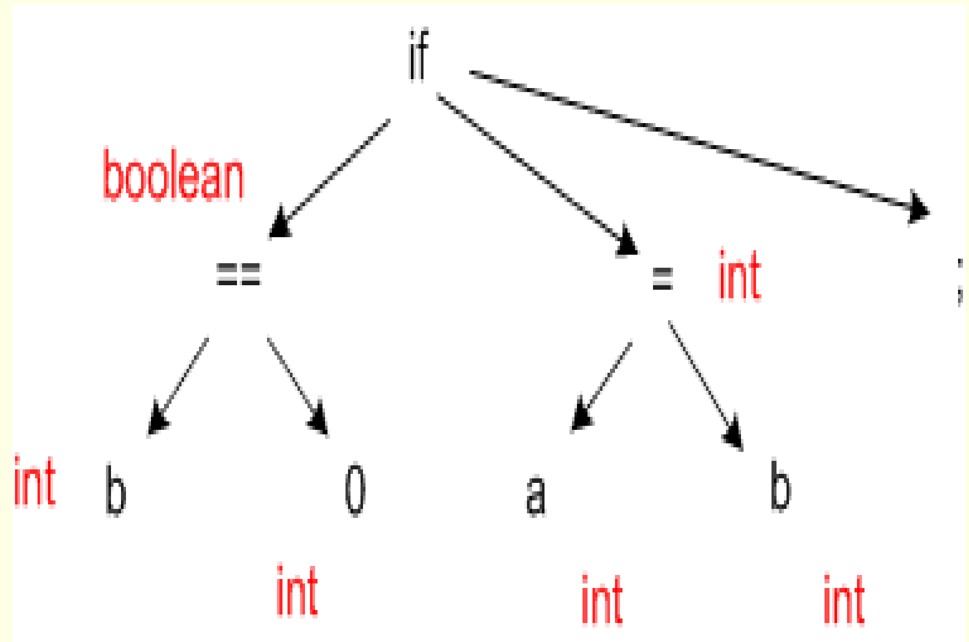
Semantic Analysis

- Static checking
  - Type checking
  - Control flow checking
  - Uniqueness checking
  - Name checks
- Disambiguate overloaded operators
- Type coercion
- Error reporting



Beyond syntax analysis

- Parser cannot catch all the program errors
- There is a level of correctness that is deeper than syntax analysis
- Some language features cannot be modeled using context free grammar formalism
  - Whether an identifier has been declared before use
  - This problem is of identifying a language $\{w\alpha w \mid w \in \Sigma^*\}$
  - This language is not context free

Beyond syntax …

- Examples
```
string x; int y;
y = x + 3
```
  the use of x could be a type error
```
int a, b;
a = b + c
```
  c is not declared

- An identifier may refer to different variables in different parts of the program
- An identifier may be usable in one part of the program but not another

Compiler needs to know?

- Whether a variable has been declared?
- Are there variables which have not been declared?
- What is the type of the variable?
- Whether a variable is a scalar, an array, or a function?
- What declaration of the variable does each reference use?
- If an expression is type consistent?
- If an array use like A[i,j,k] is consistent with the declaration? Does it have three dimensions?

- How many arguments does a function take?
- Are all invocations of a function consistent with the declaration?
- If an operator/function is overloaded, which function is being invoked?
- Inheritance relationship
- Classes not multiply defined
- Methods in a class are not multiply defined

- The exact requirements depend upon the language

How to answer these questions?

- These issues are part of semantic analysis phase
- Answers to these questions depend upon values like type information, number of parameters etc.
- Compiler will have to do some computation to arrive at answers
- The information required by computations may be non local in some cases

How to … ?

- Use formal methods
  - Context sensitive grammars
  - Extended attribute grammars
- Use ad-hoc techniques
  - Symbol table
  - Ad-hoc code
- Something in between !!!
  - Use attributes
  - Do analysis along with parsing
  - Use code for attribute value computation
  - However, code is developed systematically

Why attributes ?

- For lexical analysis and syntax analysis formal techniques were used.
- However, we still had code in form of actions along with regular expressions and context free grammar
- The attribute grammar formalism is important
  - However, it is very difficult to implement
  - But makes many points clear
  - Makes “ad-hoc” code more organized
  - Helps in doing non local computations

Attribute Grammar Framework

- Generalization of CFG where each grammar symbol has an associated set of attributes
- Values of attributes are computed by semantic rules

Attribute Grammar Framework

- Two notations for associating semantic rules with productions
- Syntax directed definition
  - high level specifications
  - hides implementation details
  - explicit order of evaluation is not specified
- Translation scheme
  - indicate order in which semantic rules are to be evaluated
  - allow some implementation details to be shown

Attribute Grammar Framework

- Conceptually both:
  - parse input token stream
  - build parse tree
  - traverse the parse tree to evaluate the semantic rules at the parse tree nodes
- Evaluation may:
  - save information in the symbol table
  - issue error messages
  - generate code
  - perform any other activity

Example

- Consider a grammar for signed binary numbers

```
number → sign list
sign   → + | -
list   → list bit | bit
bit    → 0 | 1
```

- Build attribute grammar that annotates number with the value it represents

Example

- Associate attributes with grammar symbols

symbol	**attributes**
number	value
sign	negative
list	position, value
bit	position, value

symbol	attributes
number	value
sign	negative
list	position, value
bit	position, value

production	**Attribute rule**

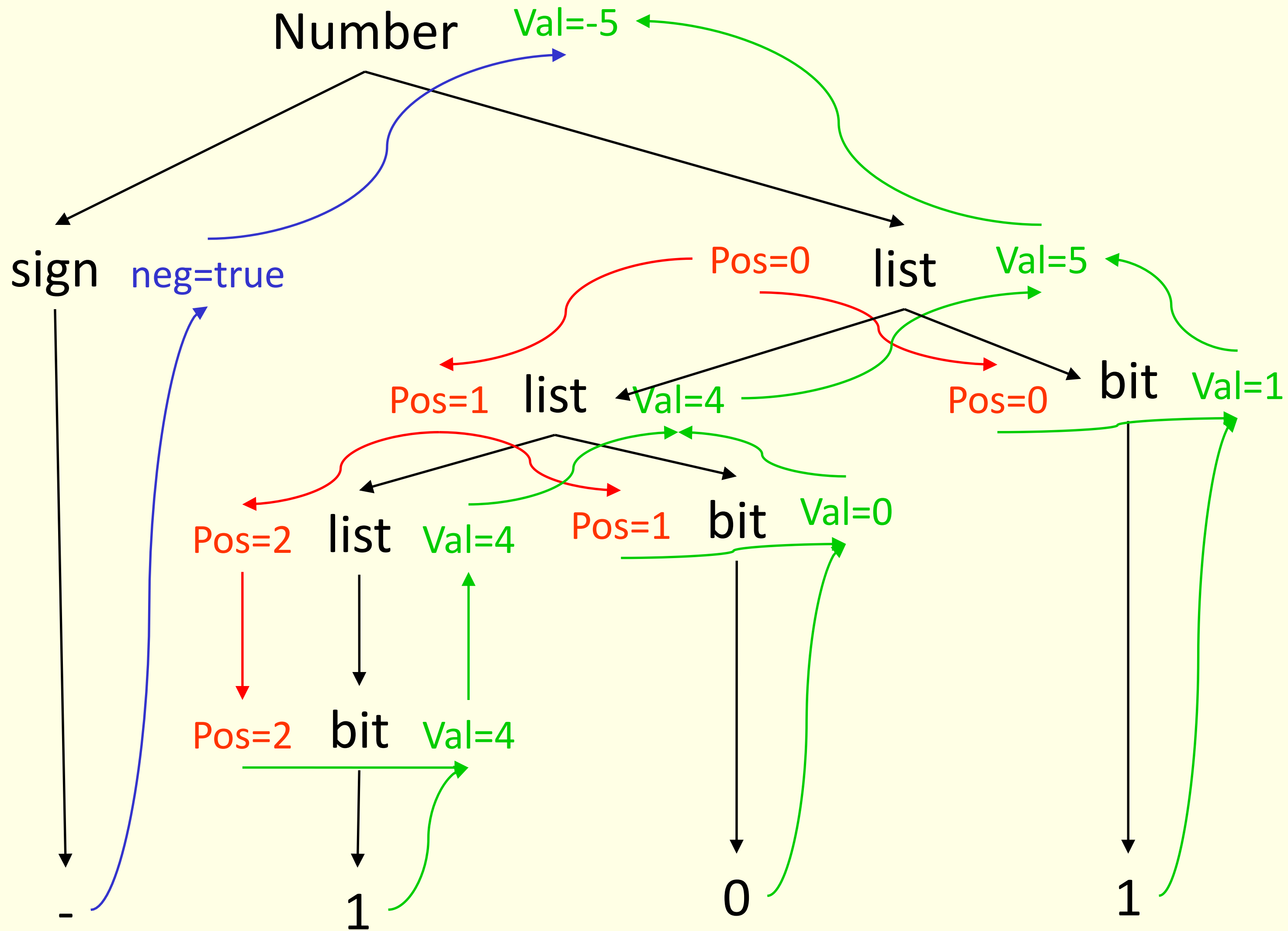
number → sign list	list.position ← 0
	if sign.negative
	    number.value ← -list.value
	else
	    number.value ← list.value

sign → +	sign.negative ← false
sign → -	sign.negative ← true

symbol	attributes
number	value
sign	negative
list	position, value
bit	position, value

production	Attribute rule
list → bit	bit.position ← list.position
	list.value ← bit.value
$list_0$ → $list_1$ bit	$list_1$.position ← $list_0$.position + 1
	bit.position ← $list_0$.position
	$list_0$.value ← $list_1$.value + bit.value
bit → 0	bit.value ← 0
bit → 1	bit.value ← $2^{bit.position}$

Parse tree and the dependence graph



Attributes …

- Attributes fall into two classes: *Synthesized* and *Inherited*
- Value of a synthesized attribute is computed from the values of children nodes
  - Attribute value for LHS of a rule comes from attributes of RHS
- Value of an inherited attribute is computed from the sibling and parent nodes
  - Attribute value for a symbol on RHS of a rule comes from attributes of LHS and RHS symbols

Attributes …

- Each grammar production $A \rightarrow \alpha$ has associated with it a set of semantic rules of the form

$$b = f\,(c_1, c_2, \ldots, c_k)$$

where f is a function, and x
  - Either b is a synthesized attribute of A
  - OR b is an inherited attribute of one of the grammar symbols on the right
- Attribute b depends on attributes $c_1, c_2, \ldots, c_k$

Synthesized Attributes

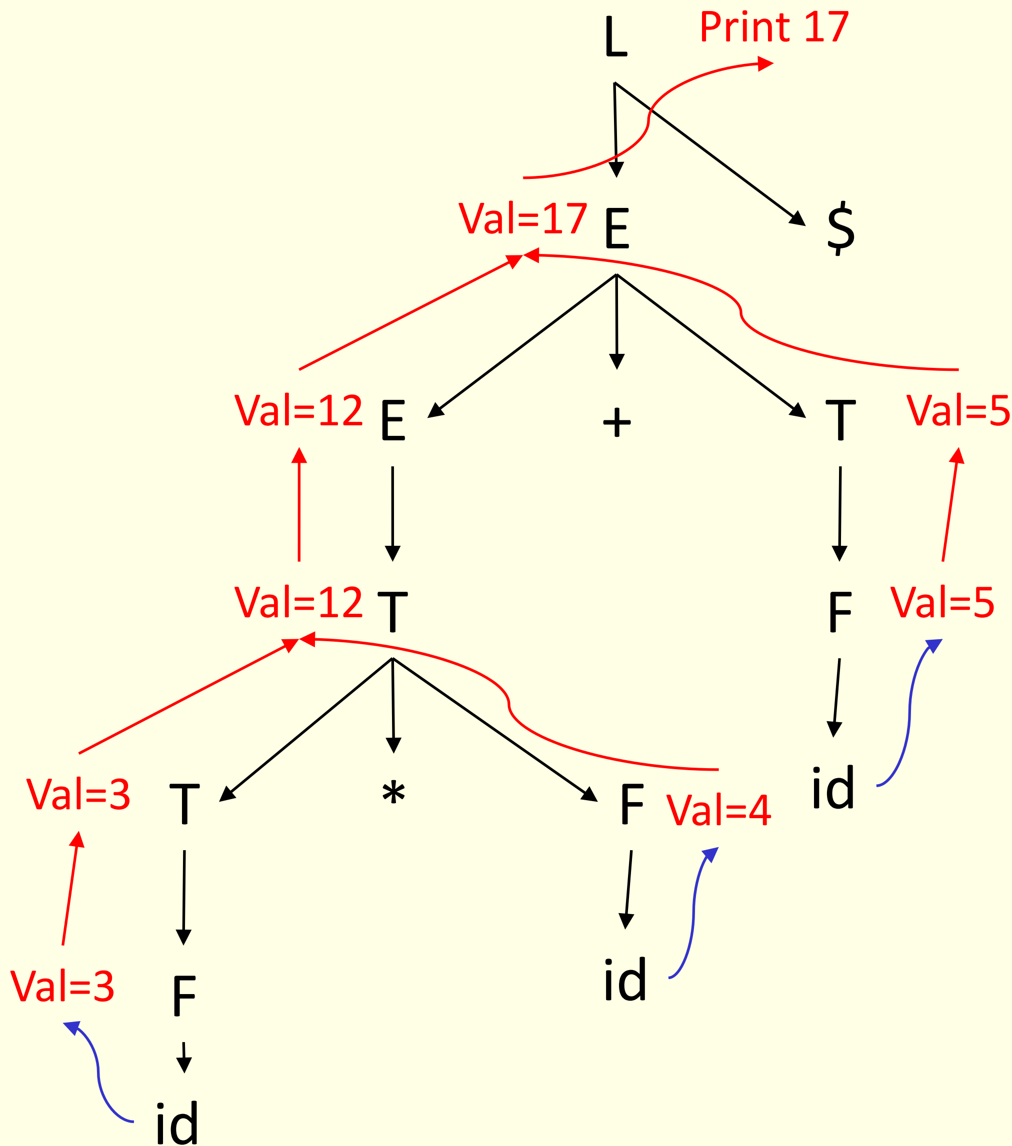
- a syntax directed definition that uses only synthesized attributes is said to be an S-attributed definition
- A parse tree for an S-attributed definition can be annotated by evaluating semantic rules for attributes

Syntax Directed Definitions for a desk calculator program

$L \rightarrow E\ \$$	Print (E.val)
$E \rightarrow E + T$	E.val = E.val + T.val
$E \rightarrow T$	E.val = T.val
$T \rightarrow T * F$	T.val = T.val * F.val
$T \rightarrow F$	T.val = F.val
$F \rightarrow (E)$	F.val = E.val
$F \rightarrow$ digit	F.val = digit.lexval

- terminals are assumed to have only synthesized attribute values of which are supplied by lexical analyzer
- start symbol does not have any inherited attribute

Parse tree for 3 * 4 + 5 n



Inherited Attributes

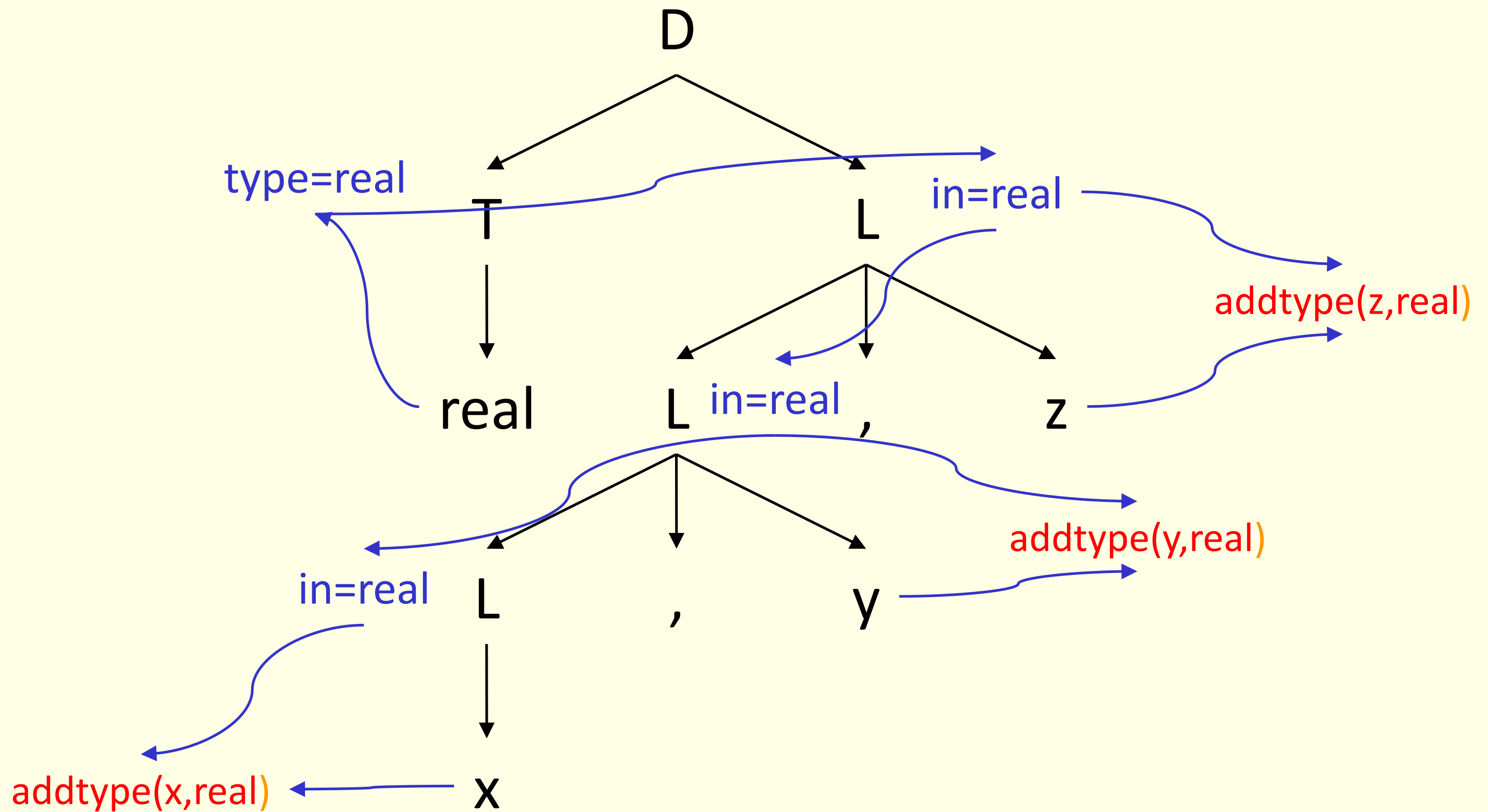
- an inherited attribute is one whose value is defined in terms of attributes at the parent and/or siblings
- Used for finding out the context in which it appears
- possible to use only S-attributes but more natural to use inherited attributes

Inherited Attributes

$D \to T\ L$	L.in = T.type
$T \to$ real	T.type = real
T int	T.type = int
$L \to L_1$, id	L_1.in = L.in; addtype(id.entry, L.in)
$L \to$ id	addtype (id.entry,L.in)

Parse tree for

real x, y, z



Dependence Graph

- If an attribute b depends on an attribute c then the semantic rule for b must be evaluated after the semantic rule for c
- The dependencies among the nodes can be depicted by a directed graph called dependency graph

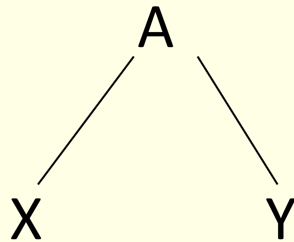
Algorithm to construct dependency graph

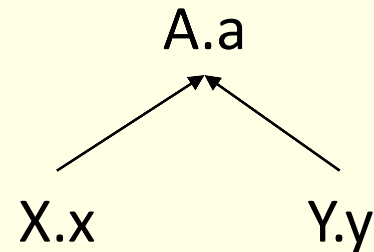
for each node **n** in the parse tree do
  for each attribute **a** of the grammar symbol do
    construct a node in the dependency graph
      for **a**

for each node **n** in the parse tree do
  for each semantic rule $b = f(c_1, c_2, \ldots, c_k)$
  { associated with production at **n** } do
    for i = 1 to k do
      construct an edge from c_i to **b**

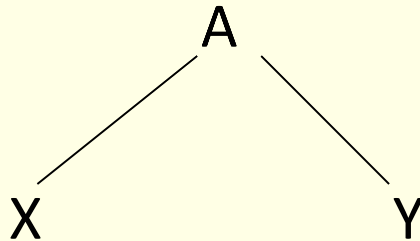
Example

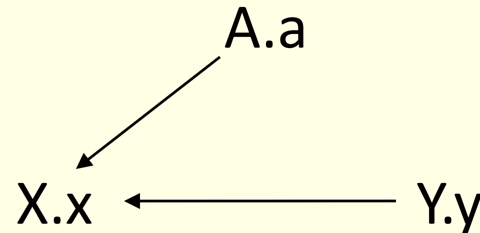
- Suppose A.a = f(X.x , Y.y) is a semantic rule for $A \rightarrow X\ Y$





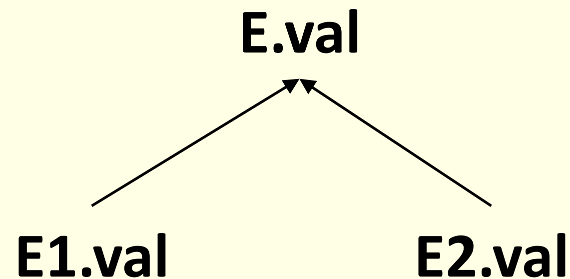
- If production $A \rightarrow X\ Y$ has the semantic rule X.x = g(A.a, Y.y)





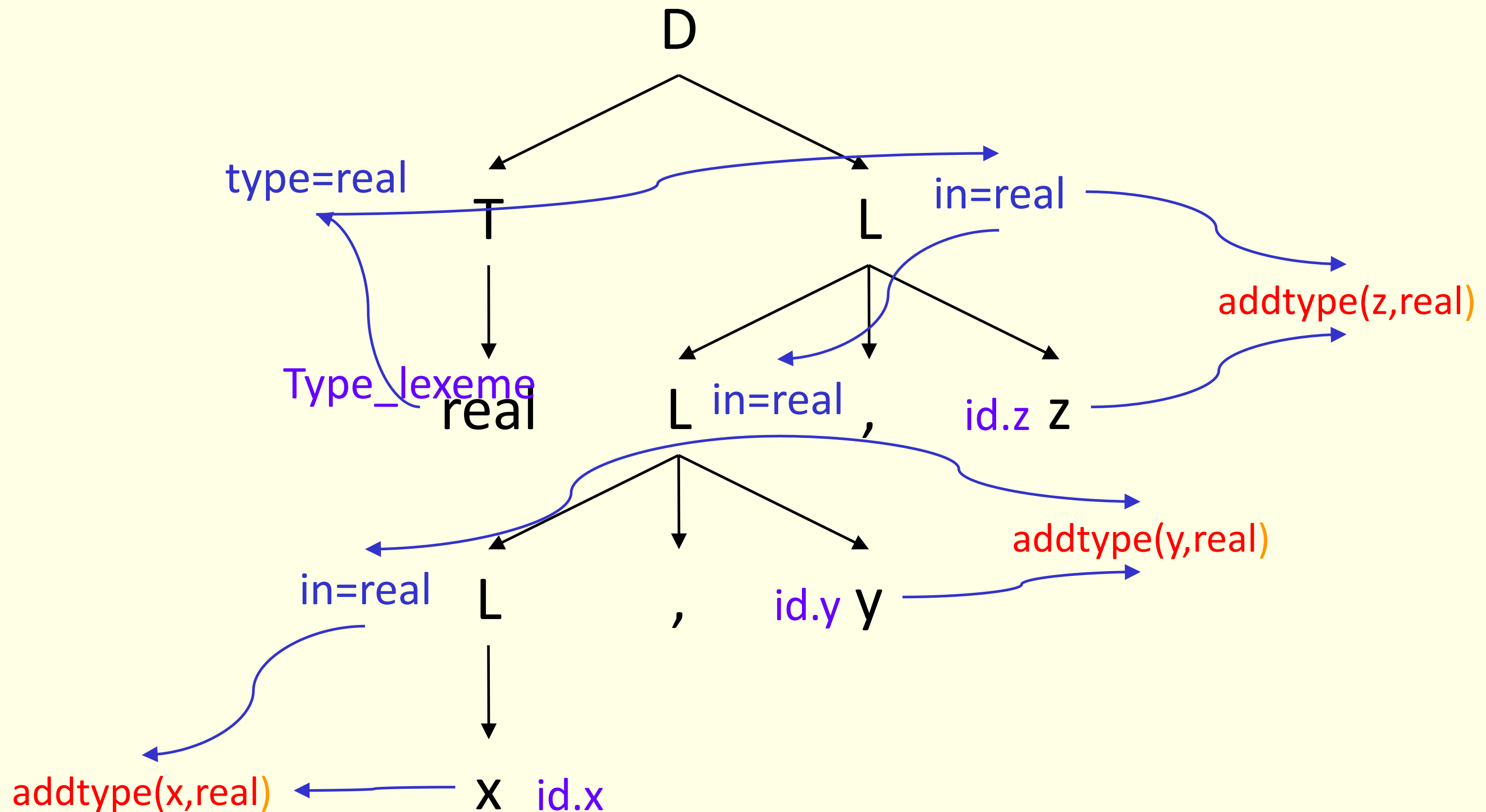
Example

- Whenever following production is used in a parse tree
  $E \rightarrow E_1 + E_2$ $\quad\quad\quad$ $E.val = E_1.val + E_2.val$
  we create a dependency graph



Example

- dependency graph for real id1, id2, id3
- put a dummy node for a semantic rule that consists of a procedure call



Evaluation Order

- Any topological sort of dependency graph gives a valid order in which semantic rules must be evaluated

a4 = real
a5 = a4
addtype(id3.entry, a5)
a7 = a5
addtype(id2.entry, a7)
a9 := a7
addtype(id1.entry, a9)