CS251: Introduction to Language Processing

Semantic Analysis

Vishwesh Jatala

Department of CSE

Indian Institute of Technology Bhilai

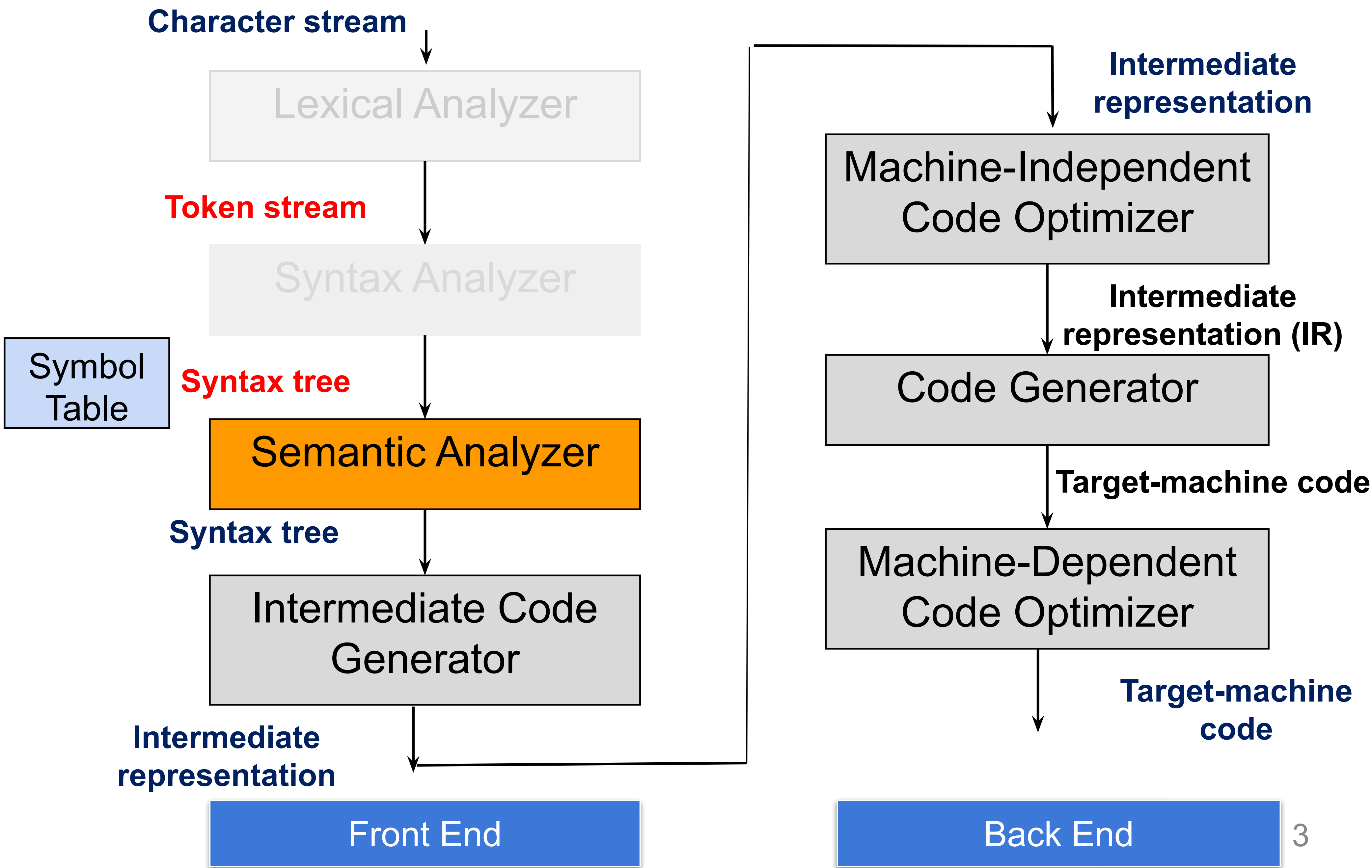
vishwesh@iitbhilai.ac.in

2023-24 Sem-1

Acknowledgement

- References for today's slides
  - *Lecture notes of Prof. Amey Karkare (IIT Kanpur) and Late Prof. Sanjeev K Aggarwal (IIT Kanpur)*
  - *IIT Madras (Prof. Rupesh Nasre)*
    - *http://www.cse.iitm.ac.in/~rupesh/teaching/compiler/aug15/schedule/4-sdt.pdf*
  - *Course textbook*
  - *Stanford University:*
    - *https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/*

Compiler Design



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

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 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

How to … ?

- Use attributes
- Do analysis along with parsing
- Use code for attribute value computation
- However, code is developed systematically
- Symbol Table

Attribute Grammar Framework

- Generalization of CFG where each grammar symbol has an associated set of attributes
- Values of attributes are computed by semantic rules
- Helps in doing computations
- Helps to express semantics

Attribute Grammar Framework

- Two notations for associating semantic rules with productions

- Syntax Directed Definition (SDD)
  - high level specifications
  - hides implementation details
  - explicit order of evaluation is not specified
- Syntax Directed Translation scheme (SDT)
  - Attaching rules or program fragments to productions
  - indicate order in which semantic rules are to be evaluated

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 evaluating arithmetic expression (*, +)

Sr. No.	Production
1	$E' \rightarrow E\ \$$
2	$E \rightarrow E_1 + T$
3	$E \rightarrow T$
4	$T \rightarrow T_1 * F$
5	$T \rightarrow F$
6	$F \rightarrow (E)$
7	$F \rightarrow \mathit{digit}$

Example

- Associate attributes with grammar symbols

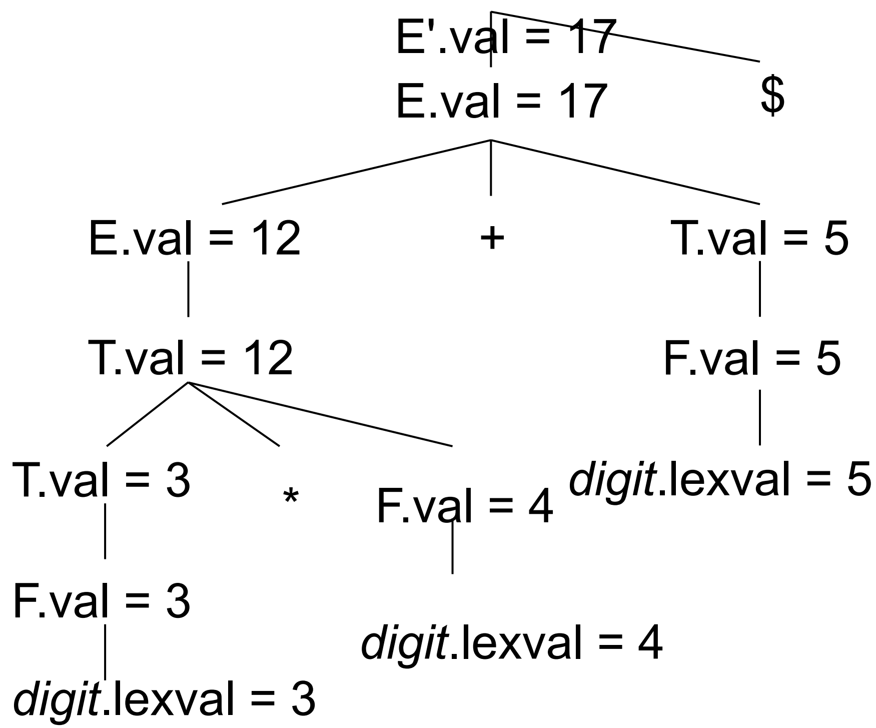
Sr. No.	Symbol	Attribute
1	E’	val
2	E	val
3	T	val
4	F	val
5	*digit*	lexval

Annotated Parse Tree

Input string

3 * 4 + 5 $

Annotated Parse Tree



Example

- Attributed grammar - Syntax Directed Definition

Sr. No.	Production	Semantic Rules
1	$E' \rightarrow E\ \$$	E'.val = E.val
2	$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3	$E \rightarrow T$	E.val = T.val
4	$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
5	$T \rightarrow F$	T.val = F.val
6	$F \rightarrow (E)$	F.val = E.val
7	F → *digit*	F.val = *digit*.lexval

Example-2

$D \rightarrow T\ L$

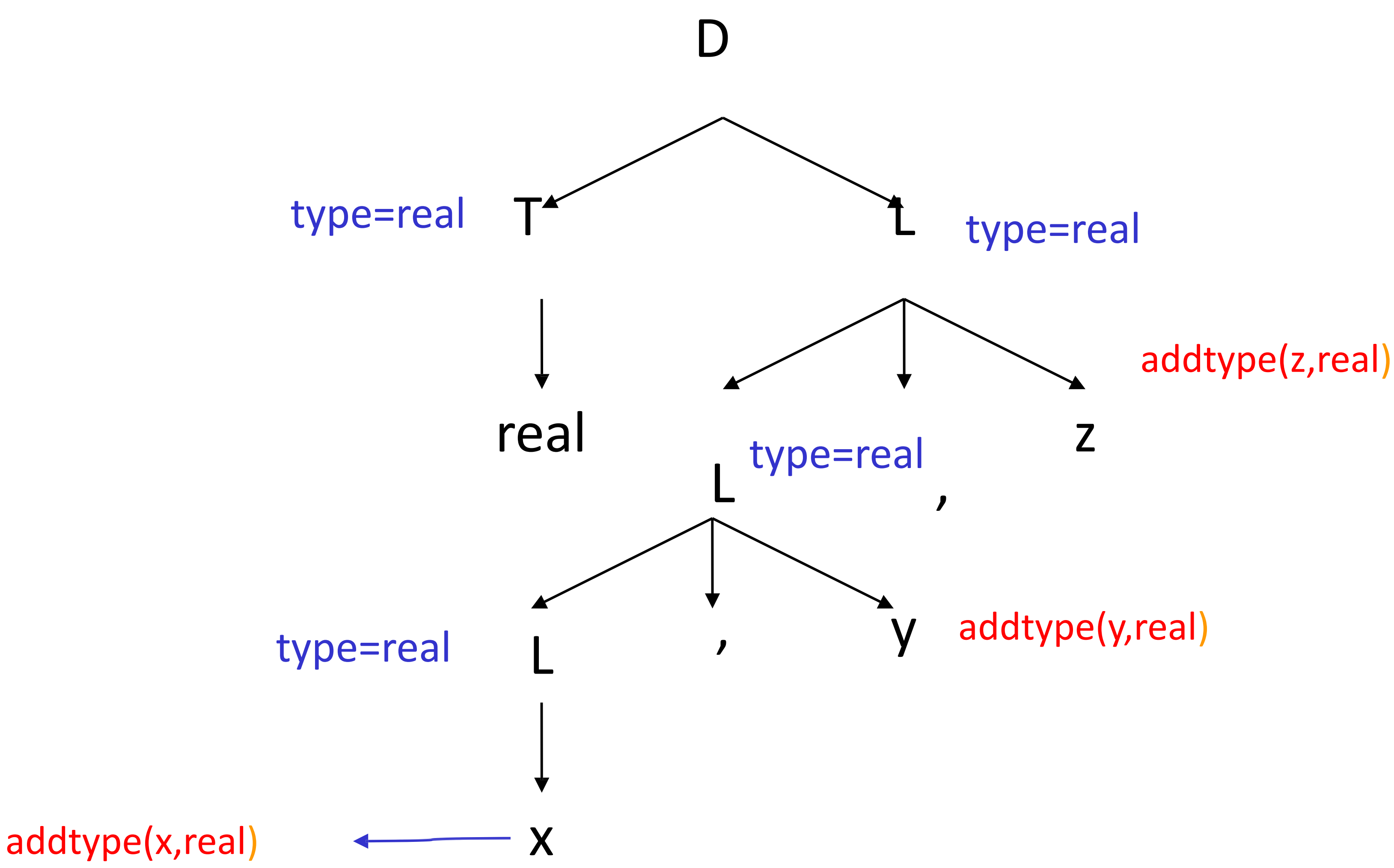
$T \rightarrow$ real

$T \rightarrow$ int

$L \rightarrow L,$ id

$L \rightarrow$ id

Annotated parse tree for real x, y, z

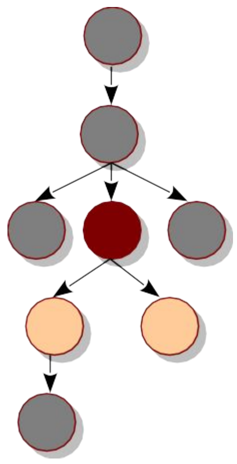


Inherited Attributes

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

Attributes …

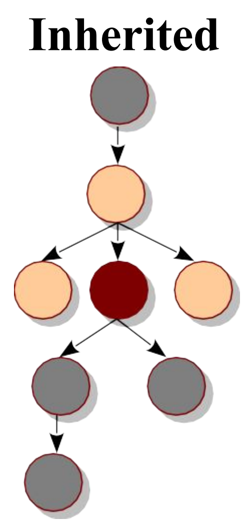
- Attributes fall into two classes: *Synthesized* and *Inherited*
- Value of a synthesized attribute is computed from the values of children nodes
  - Attribute value for LHS of a rule comes from attributes of RHS



Synthesized

Attributes …

- 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



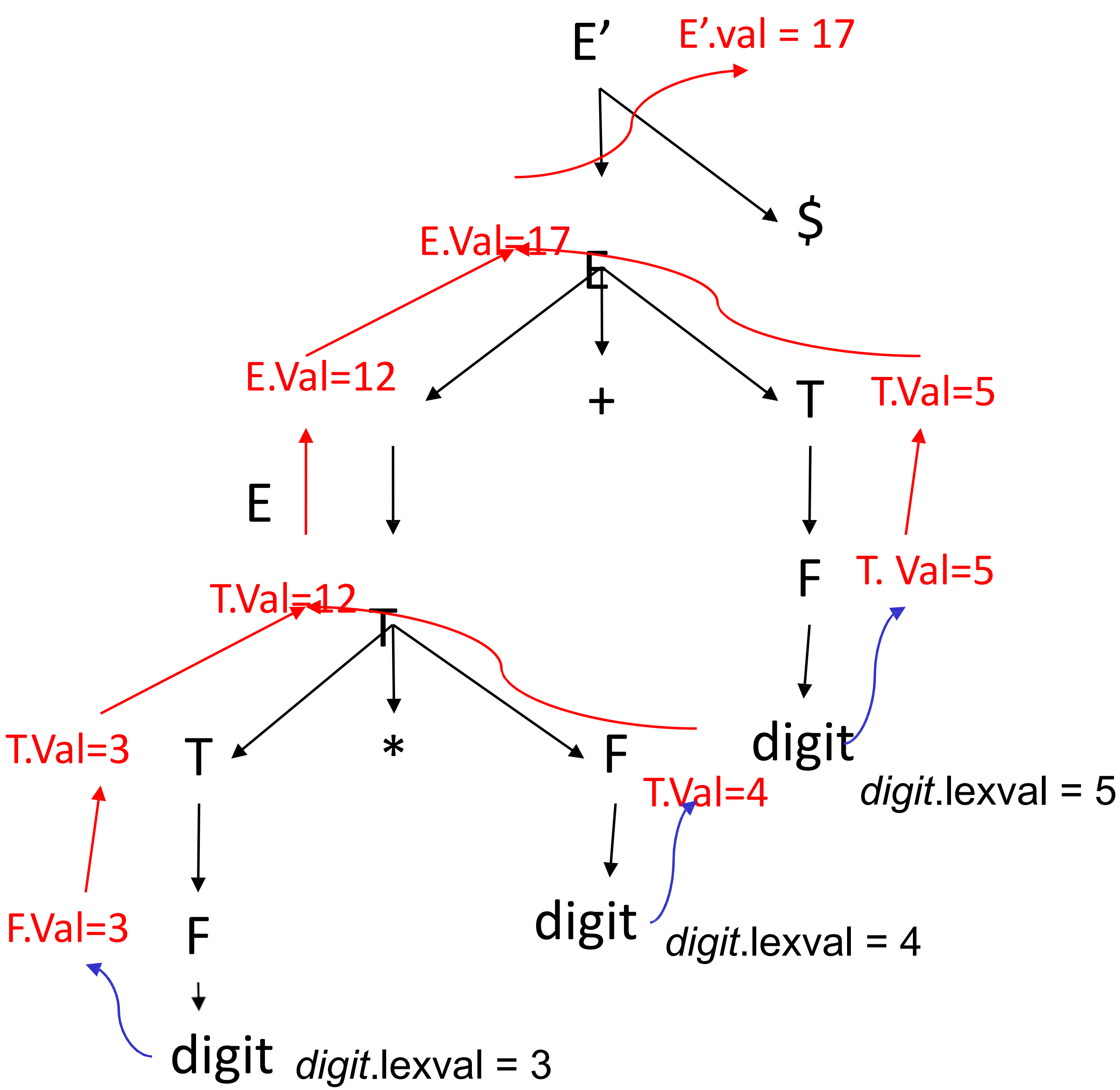
Semantics

- 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
  - 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 $c_1, c_2, \ldots, c_k$

Order of Evaluation

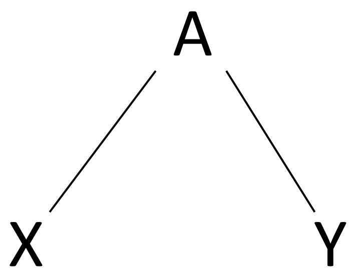


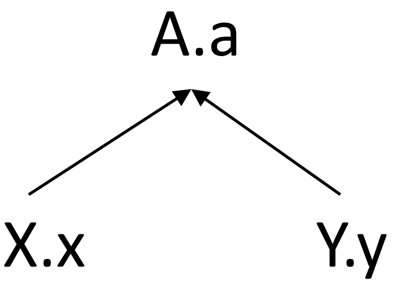
Dependence Graph

- 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

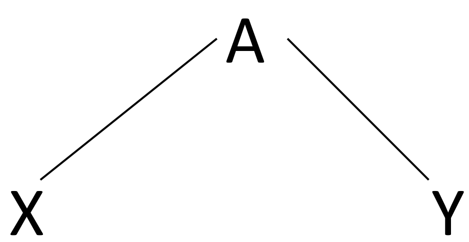
Example

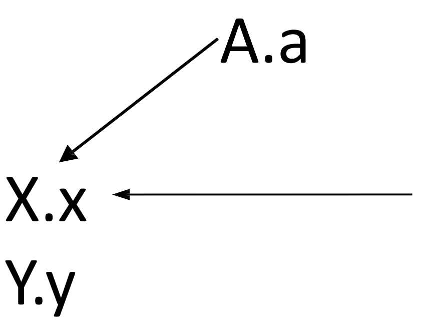
- Suppose $A.a = f(X.x , Y.y)$ is a semantic rule for $A \rightarrow X\ Y$





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





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

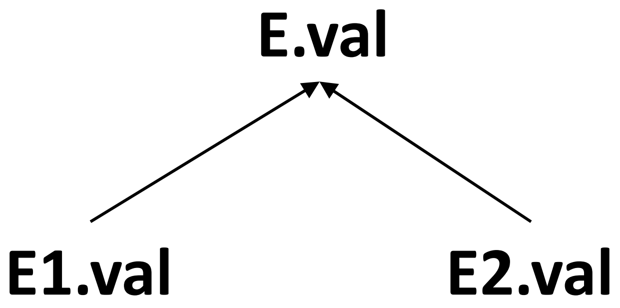
for i = 1 to k do

construct an edge from c_i to b

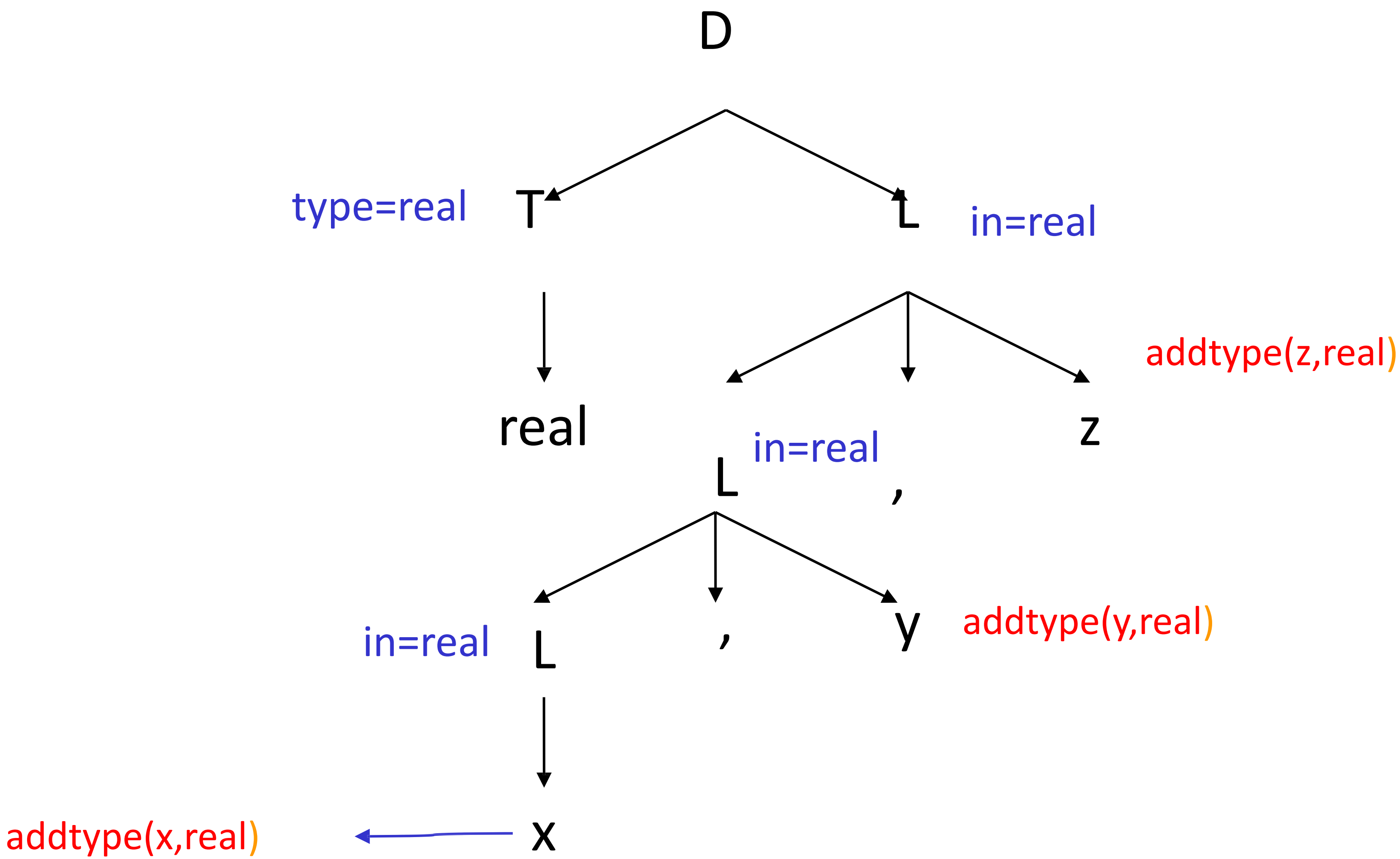
Example

- Consider the following production is used in a parse tree
  - $E \rightarrow E_1 + E_2$ $E.val = E_1.val + E_2.val$

we create a dependency graph

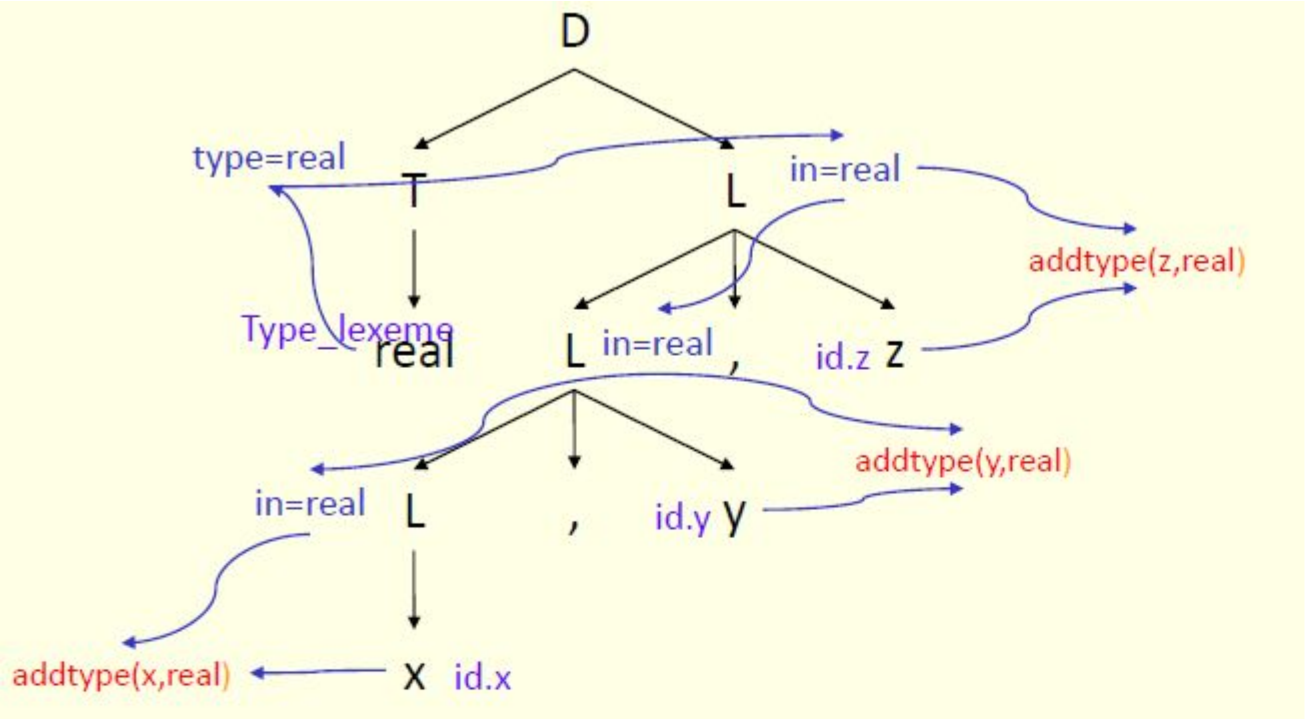


Example: real id1, id2, id3



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

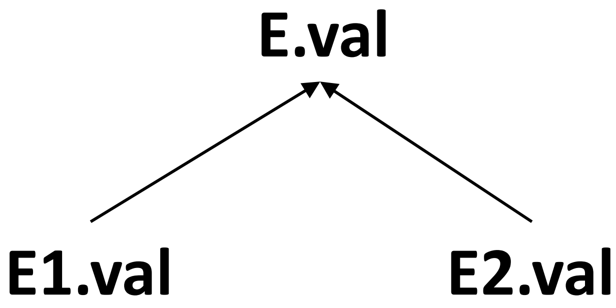
Synthesized Attributes

- a syntax directed definition that uses only synthesized attributes is said to be an S-attributed definition
  - A topological evaluation order is well-defined.
  - Any bottom-up order of the parse tree nodes.

Example

- Consider the following production is used in a parse tree
  - $E \rightarrow E_1 + E_2$ $\quad\quad$ $E.val = E_1.val + E_2.val$

  we create a dependency graph



Issues with S-attributed SDD

- It is too strict!
- There exist reasonable non-cyclic orders that it disallows.
  - If a non-terminal uses attributes of its parent only (no sibling attributes)
  - If a non-terminal uses attributes of its left-siblings only (and not of right siblings).
- The rules may use information "from above" and "from left".

L attributed definitions

- Each attribute must be either
  - synthesized, or
  - inherited, but with restriction. For production A ⟶ X1 X2 … Xn with inherited attributed Xi.a computed by an action; then the rule may use only
    - inherited attributes of A.
    - either inherited or synthesized attributes of X1, X2, …, Xi-1.
    - inherited or synthesized attributes of Xi with no cyclic dependence.
- L is for left-to-right.

L attributed definitions

$A \longrightarrow LM$	$L.i = f_1(A.i)$
	$M.i = f_2(L.s)$
	$A.s = f_3(M.s)$

$A \longrightarrow QR$	$R.i = f4(A.i)$
	$Q.i = f5(R.s)$
	$A.s = f6(Q.s)$

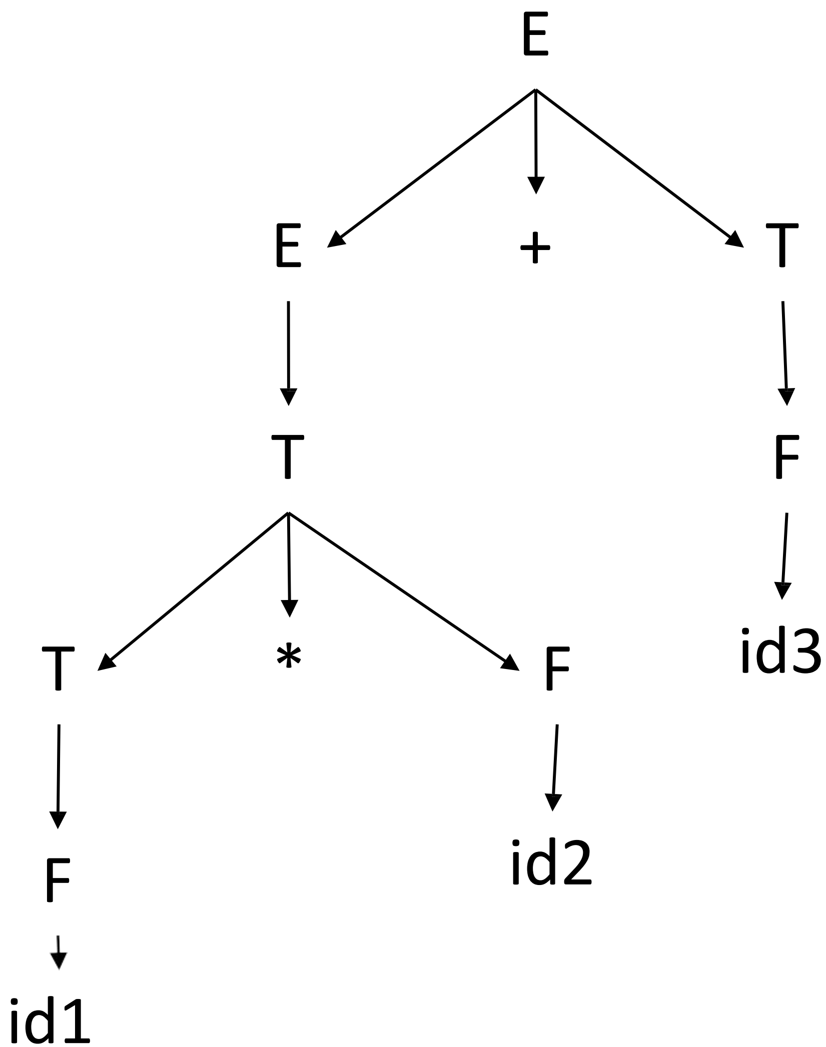
L attributed definitions

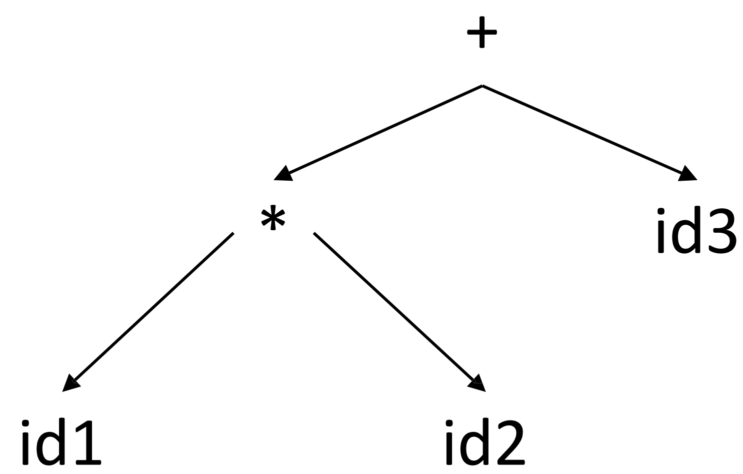
- We can adapt the grammar to compute the L-attributes during LR parsing.

  In the next class

Another Example for SDD

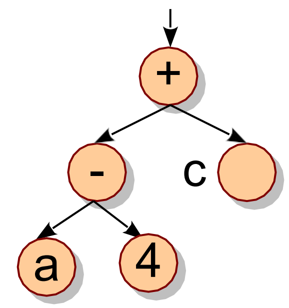
- Chain of single productions may be collapsed, and operators move to the parent nodes





Abstract Syntax Tree - Example

$a - 4 + c$



p1 = new Leaf(id_a);

p2 = new Leaf(num_4);

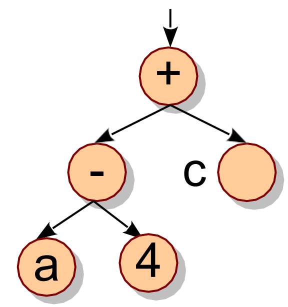
p3 = new Op(p1, '-', p2);

p4 = new Leaf(id_c);

p5 = new Op(p3, '+', p4);

Abstract Syntax Tree - Example

a – 4 + c



Production	Semantic Rules
E → E + T	$$.node = new Op($1.node, '+', $3.node)
E → E - T	$$.node = new Op($1.node, '-', $3.node)
E → T	$$.node = $1.node
T → (E)	$$.node = $2.node
T → *id*	$$.node = new Leaf($1)
T → *num*	$$.node = new Leaf($1)

Summary

- Express semantics:
  - Using attributed grammar
  - Synthesized attributes
  - Inherited attributes
  - Order of evaluation
    - Dependency graph
  - S-attributed and L-attributed grammar