CS536
SDT, Annotation, Dependency Graph and Evaluation

A Sahu

CSE, IIT Guwahati

http://jatinga.iitg.ac.in/~asahu/cs536/

Outline

- **Basic of Syntax Directed Translation**
- Symbol Table
- Intermediate Representation

http://jatinga.iitg.ac.in/~asahu/cs536/

Top Down Parsing

- This is example of Grammar to generate subset of statement in C/Java

stmt → *expr*

  | **if** (*expr*) stmt

  | **for**(*optexpr;optexpr; optexpr*) *stmt*

  | **others**

optexpr → *expr*| ϵ

Top Down Parsing

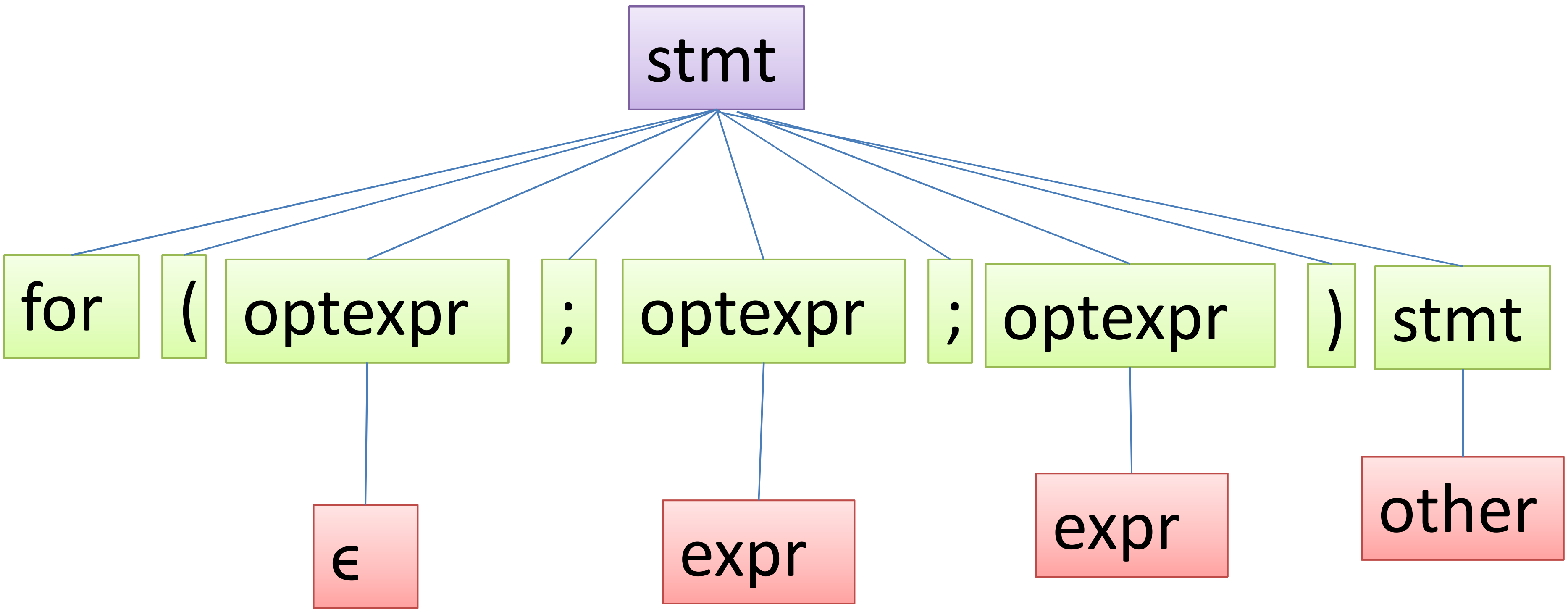
- Top Down Parsing can be done by starting with the root, labeled with NT stmt and repeatedly doing
  - At node N, labeled with non terminal A, select one production for A and children at N for the symbols
  - Find the next node at which a subtree is to constructed, typically the leftmost unexpanded NT of the tree
- The above steps during a single left-to-right scan of the input
- The current symbol being scan is ***lookahead.***

Top Down Parsing: example

- Suppose input string is

  `for ( ; expr; expr ) other`

- Tree for this is



Predictive parsing

- Recursive - descendent parsing (RDP) is a top down parsing of syntax analysis
  - Also called the predictive parsing
- One procedure associated with
  - each non-terminal of a grammar
- In RDP, lookahead symbol
  - unambiguously determined the flow of control through the procedure body for each non-terminals
- The sequence of procedure calls during the analysis of an input string
  - Implicitly define the parse tree for the input

Code for predictive parser

```
void stmt() {
  switch ( lookahead ) {
case expr        :  MC(expr); MC(' ; '); break;
case if          : MC(if); MC(' (I); MC (expr); MC(') '); stmt (); break;
case for         : MC (for); MC (' (') ; optexpr (); MC(' ; ‘);
                   optexpr(); MC(' ; '); optexpr(); MC(') '); stmt (); break;
case other       :  MC (other) ; break;
default          : report ("syntax error");
}
void optexpro { if ( lookahead == expr ) MC(expr); }
void MC(termina1 t) {
        if (Lookahead == t ) lookahead = nextTerminal;
        else report ("syntax error”);
}
```

Syntax-Directed Definition

In a syntax-directed definition, each production $A \rightarrow \alpha$ is associated with a set of semantic rules of the form:

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

where *f* is a function and *b* can be one of the followings:

➔ *b* is a synthesized attribute of A and $c_1, c_2, \ldots, c_n$ are attributes of the grammar symbols in the production ($A \rightarrow \alpha$).

OR

➔ *b* is an inherited attribute one of the grammar symbols in α (on the right side of the production), and $c_1, c_2, \ldots, c_n$ are attributes of the grammar symbols in the production ($A \rightarrow \alpha$).

Attribute Grammar

- So, a semantic rule $b=f(c_1,c_2,\ldots,c_n)$ indicates that the attribute b *depends on* attributes $c_1,c_2,\ldots,c_n$.
- In a **syntax-directed definition**, a semantic rule may just evaluate a value of an attribute
  - or it may have some side effects such as printing values ***; //suggestion to user, possible warning during compilation***
- An **attribute grammar** is a syntax-directed definition in which the functions in the semantic rules cannot have side effects
  - they can only evaluate values of attributes

Syntax-Directed Definition -- Example

Production	**Semantic Rules**
$L \rightarrow E$ **n**	print(E.val)
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow$ **digit**	F.val = **digit**.lexval

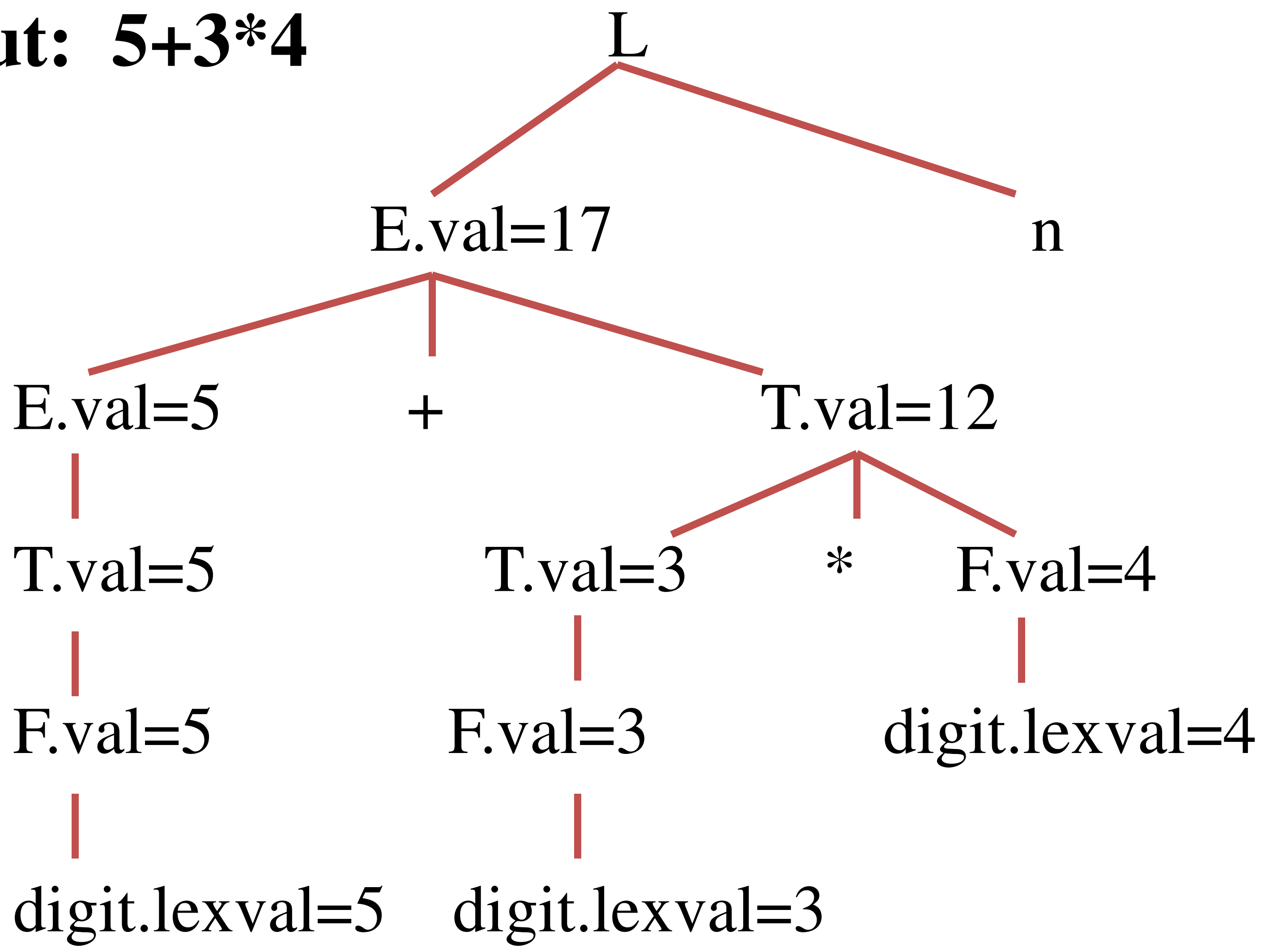
- Symbols E, T, and F are associated with a synthesized attribute *val*.
- The token **digit** has a synthesized attribute *lexval* (it is assumed that it is evaluated by the lexical analyzer).
- Terminals are assumed to have synthesized attributes only. Values for attributes of terminals are usually supplied by the lexical analyzer.
- The start symbol does not have any inherited attribute unless otherwise stated.

S-attributed definition

- S-attributed definition : A SD translation that uses synthesized attributes exclusively
- A parse tree for a S-attributed definition
  - Can be annotated by evaluating the semantic rules for the attributes at each node,
  - Bottom up from leaves to the root.

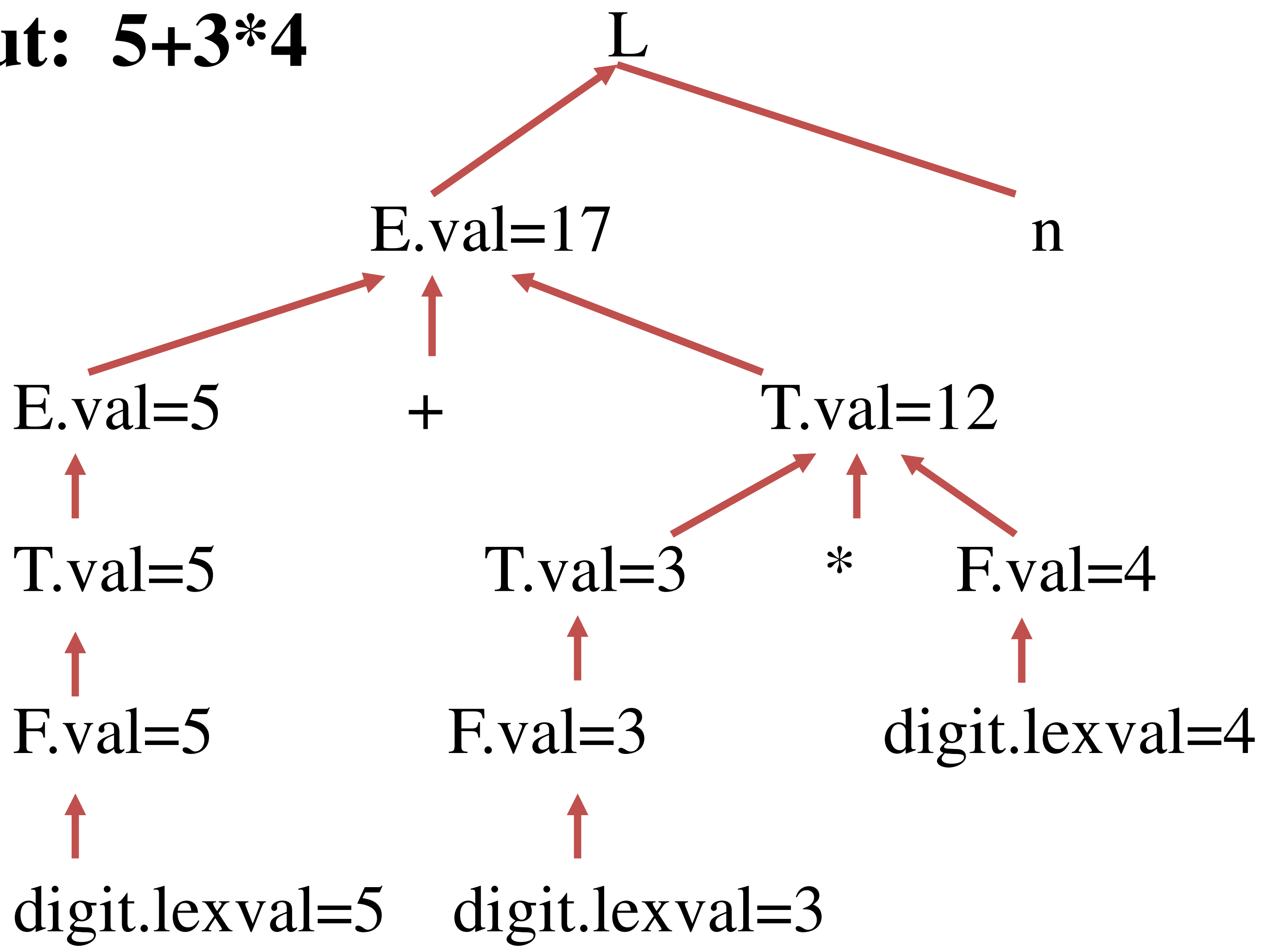
Annotated Parse Tree -- Example

Input: 5+3*4



Dependency Graph

Input: 5+3*4



Inherited attributes

- An inherited value at a node in a parse tree is
  - Defined in terms of attributes at the parent and/or siblings of the node.
- Convenient way for expressing the dependency of
  - Construct on the context in which it appears.
- We can use inherited attributes to
  - Keep track of whether an identifier appears on the left or right side of an assignment
  - To decide whether the address or value of the assignment is needed.
- Example: The inherited attribute distributes type
- information to the various identifiers in a declaration.

SDD– Inherited Attributes

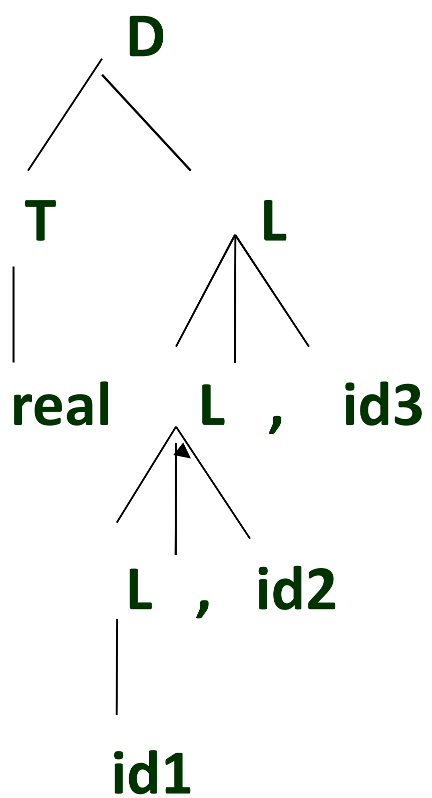
Production	Semantic Rules
D → T L	L.in = T.type
T → **int**	T.type = integer
T → **real**	T.type = real
L → L_1 **id**	L_1.in = L.in, addtype(**id**.entry,L.in)
L → **id**	addtype(**id**.entry,L.in)

- Symbol T is associated with a synthesized attribute *type*.
- Symbol L is associated with an inherited attribute *in.*

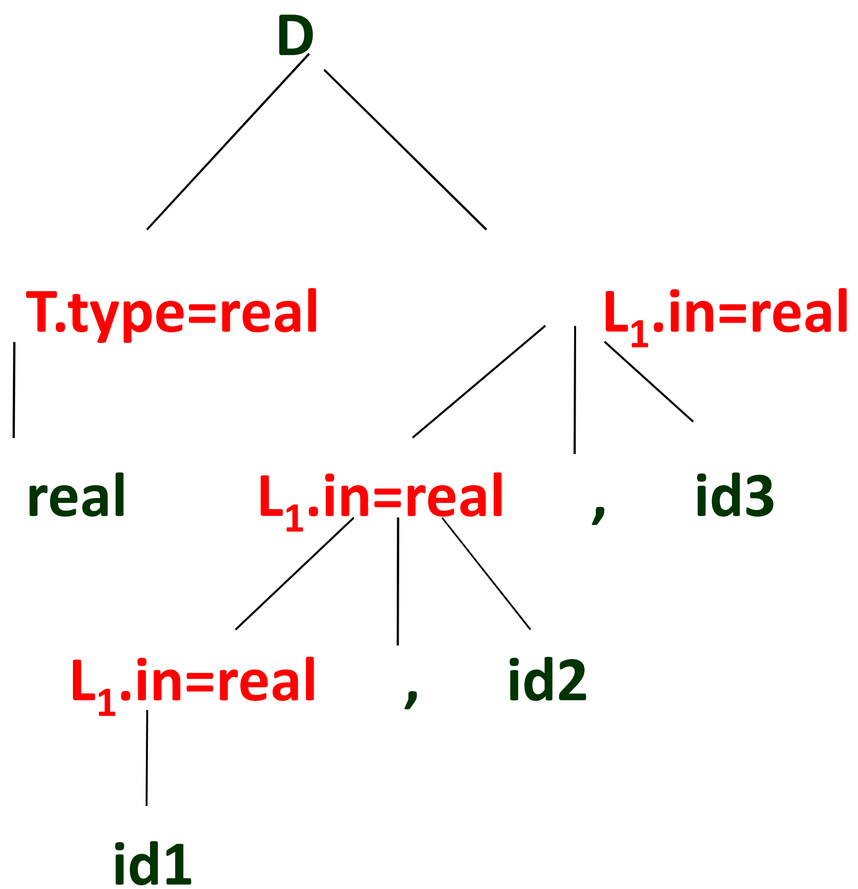
Annotated parse tree

Input: `real p,q,r`

parse tree



annotated parse tree



Dependency Graph

- Directed Graph shows interdependencies between attributes.
- If an attribute b at a node depends on an attribute c, then
  - The semantic rule for b at that node must be evaluated after the semantic rule that defines c.

Dependency Graph

- Construction:
  - Put each semantic rule into the form $b=f(c_1,\ldots,c_k)$ by introducing dummy synthesized attribute b for every semantic rule that consists of a procedure call.
  - E.g.,
    - **L $\rightarrow$ E n** ***print*(E.*val*)**
    - **Becomes: *dummy = print*(E.*val*)**
  - The graph has a node for each attribute and an edge to the node for b from the node for c if attribute b depends on attribute c.

Dependency Graph Construction

for each node ***n*** in the parse tree do

for each attribute **a** of the grammar symbol at ***n*** do

construct a node in the dependency graph for ***a***

for each node *n* in the parse tree do

for each semantic rule $b = f(c_1,\ldots,c_n)$

associated with the production used at *n* do

for i= 1 to n do

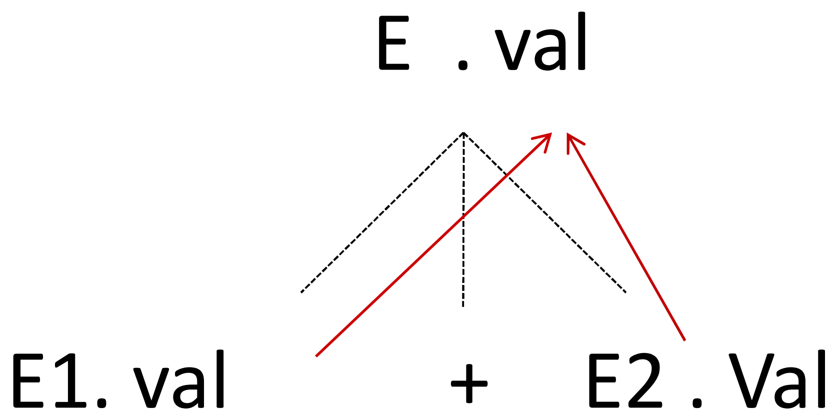
construct an edge from

the node for c_i to the node for ***b***

Dependency Graph Construction

- Example
- <u>Production</u> <u>Semantic Rule</u>

  E→E1 + E2 E.val = E1.val + E2.val



- E.val is synthesized from E1.val and E2.val
- The dotted lines represent the parse tree that is not part of the dependency graph.

Dependency Graph

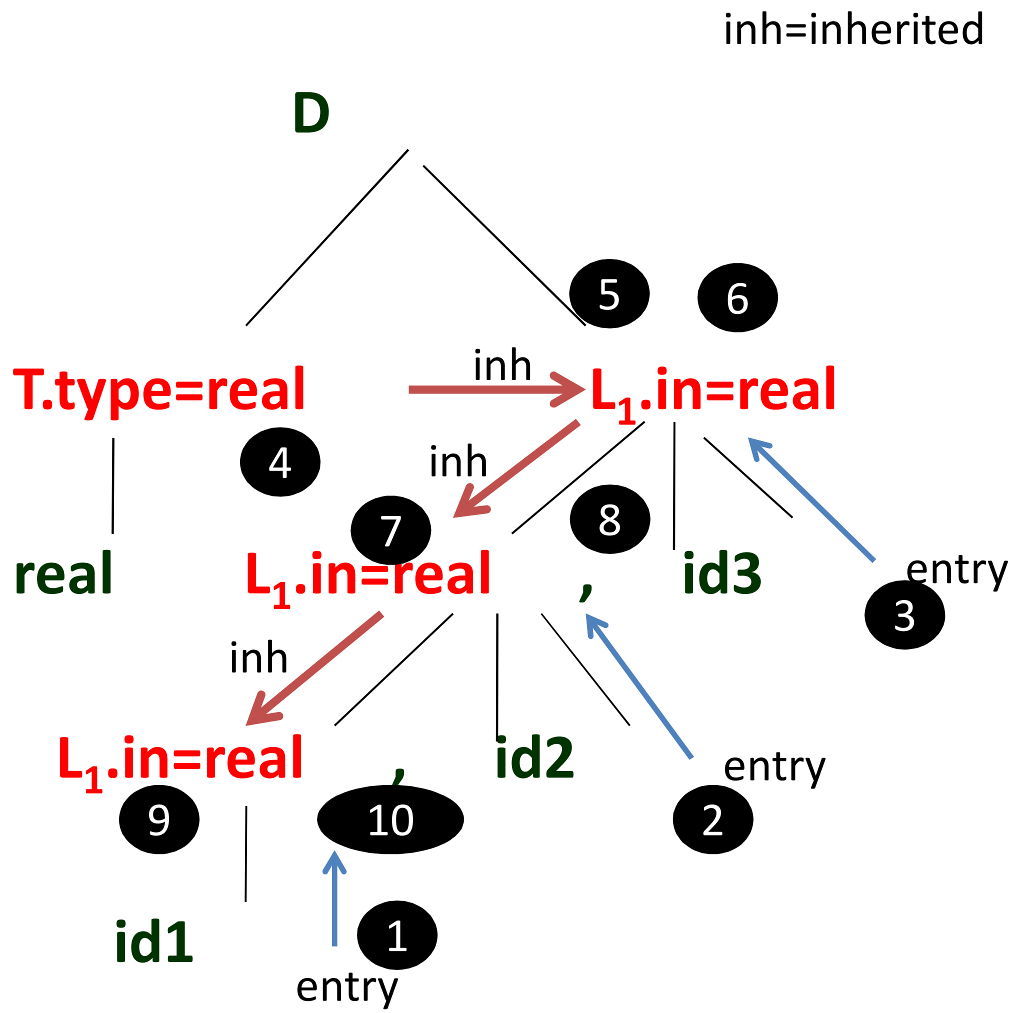
D → T L L.in = T.type

T → int T.type = integer

T → real T.type = real

L → L_1 id L_1.in = L.in,
addtype(id.entry,L.in)

L → id
addtype(id.entry,L.in)



Graph for a declaration: real id1, id2, id3;

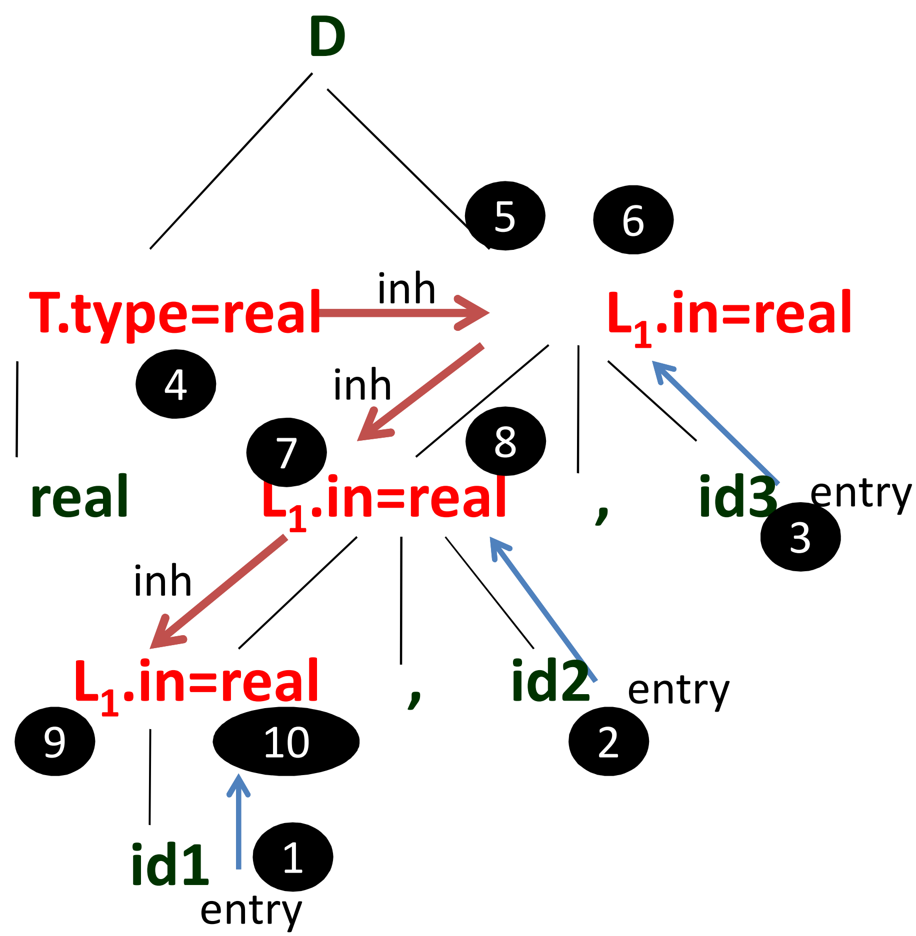
Evaluation Order

- A topological sort of a directed acyclic graph
  - is any ordering m1,m2…mk of the nodes of the graph such that edges go from nodes earlier in the ordering to later nodes.
  - i.e if there is an edge from m_i to m_j them m_i appears before m_j in the ordering
- **Any topological sort** of dependency graph gives a valid order for evaluation of semantic rules associated with the nodes of the parse tree.
  - The dependent attributes c1,c2….ck in b=f(c1,c2….ck) must be available before f is evaluated.

Evaluation Order

inh=inherited

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



Graph for a declaration: real id1, id2, id3;

Evaluating Semantic Rules

1. Parse Tree methods
   - At compile time evaluation order obtained from the topological sort of dependency graph.
   - Fails if dependency graph has a cycle
2. Rule Based Methods
   - Semantic rules analyzed by hand or specialized tools at compiler construction time
   - Order of evaluation of attributes associated with a production is pre-determined at compiler construction time

Evaluating Semantic Rules

3. Oblivious Methods
   - Evaluation order is chosen without considering the semantic rules.
   - Restricts the class of syntax directed definitions that can be implemented.
   - If translation takes place during parsing order of evaluation is forced by parsing method.

Syntax Trees

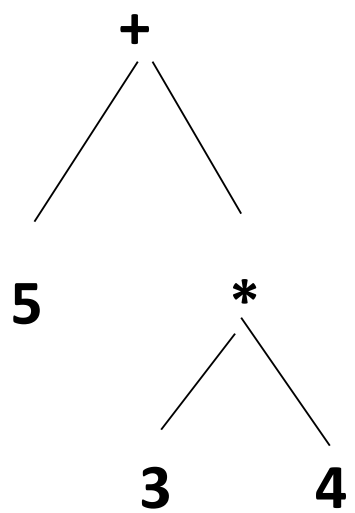
Syntax-Tree

- an intermediate representation of the compiler's input.
- A condensed form of the parse tree.
- Syntax tree shows the syntactic structure of the program while omitting irrelevant details.
- Operators and keywords are associated with the interior nodes.
- Chains of simple productions are collapsed.

Syntax directed translation can be based on syntax tree as well as parse tree.

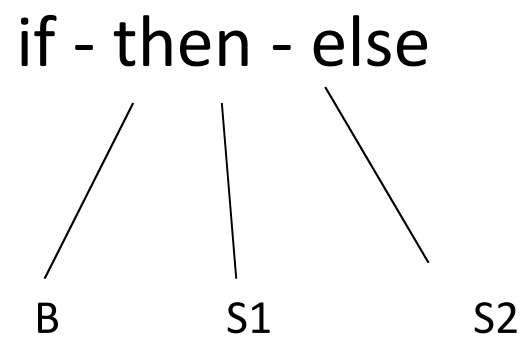
Syntax Tree-Examples

Expression:



- **Leaves: identifiers or constants**
- **Internal nodes: labelled with operations**
- **Children: of a node are its operands**

if B then S1 else S2



Statement:

- Node's label indicates what kind of a statement it is
- Children of a node correspond to the components of the statement

Constructing Syntax Tree for Expressions

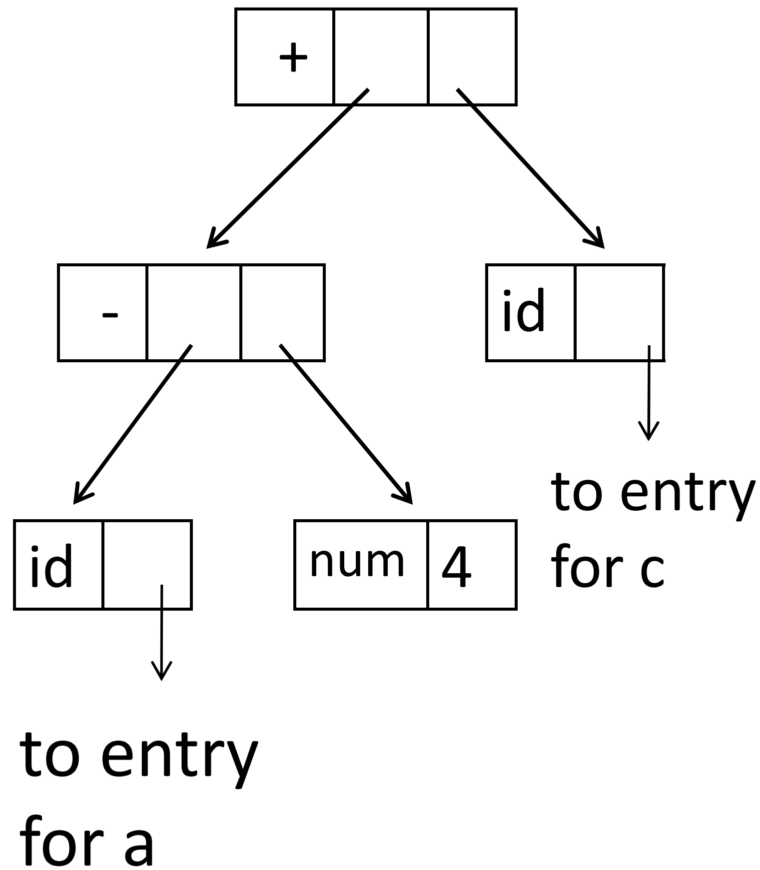
- Each node can be implemented as a record with several fields.
- Operator node: one field identifies the operator (called *label of the node*) and remaining fields contain pointers to operands.
- The nodes may also contain fields to hold the values (pointers to values) of attributes attached to the nodes.
- Functions used to create nodes of syntax tree for expressions with binary operator are given below.
  - mknode(op,left,right), mkleaf(id,entry)
  - mkleaf(num,val)

Each function returns a pointer to a newly created node

Constructing Syntax Tree for Expressions-

Example: a-4+c

1. p1:=mkleaf(id,entrya);
2. p2:=mkleaf(num,4);
3. p3:=mknode(-,p1,p2)
4. p4:=mkleaf(id,entryc);
5. p5:= mknode(+,p3,p4);



- **The tree is constructed bottom up.**

Thanks