Syntax Directed Translation

Syntax-Directed Definitions(SDD)

- context-free grammar together with attributes and rule
- If X is a symbol and a is one of its attributes write X.a
- Attributes may be of any kind: numbers, types, table references, or strings

Attribute Grammars

- An attribute grammar is a context free grammar with associated semantic attributes and semantic rules
- Each grammar symbol is associated with a set of semantic attributes
- Each production is associated with a set of semantic rules for computing semantic attributes

An Example - Interpretation

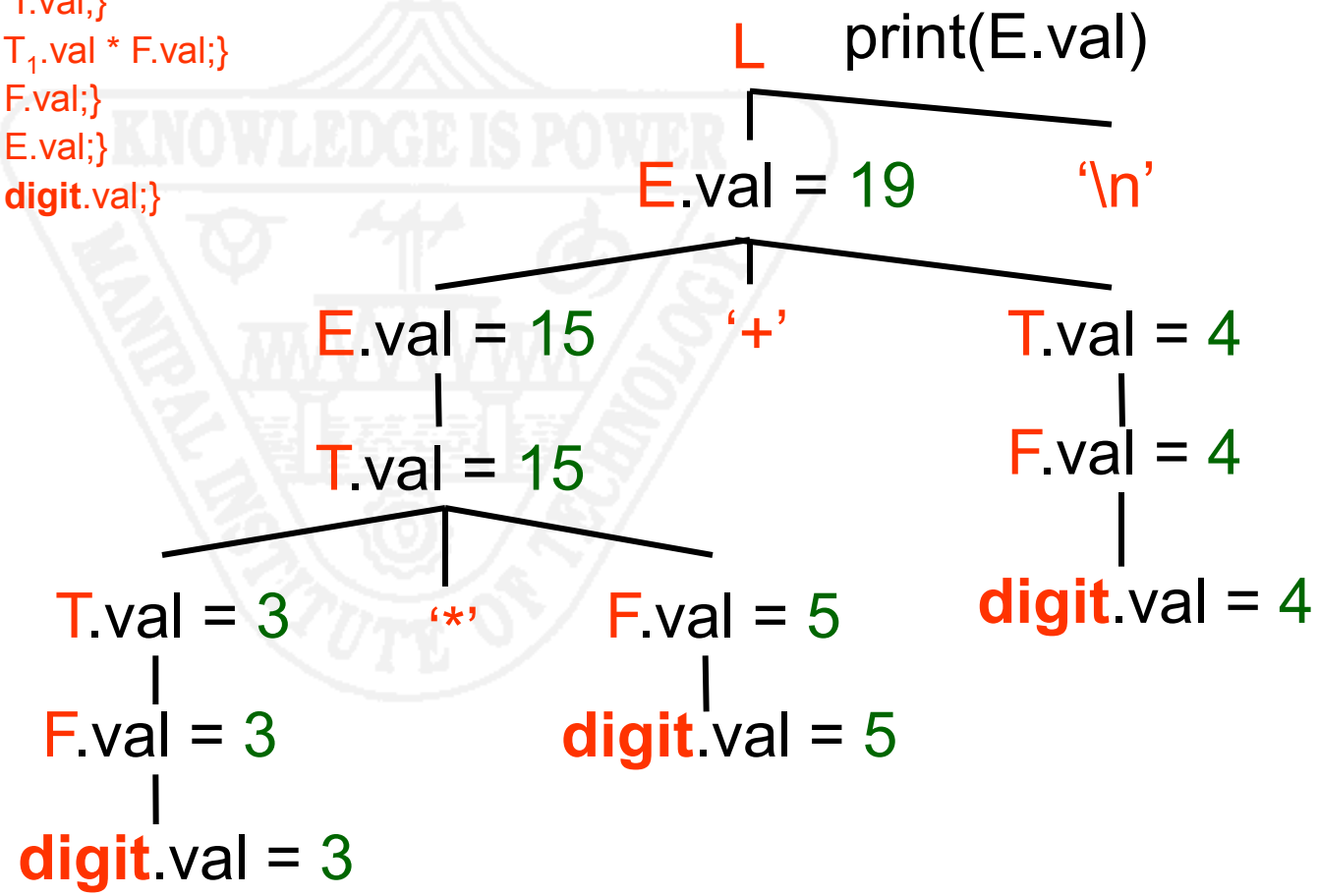
L → E '\n'	{print(E.val);}
E → E_1 '+' T	{E.val := E_1.val + T.val;}
E → T	{E.val := T.val;}
T → T_1 '*' F	{T.val := T_1.val * F.val;}
T → F	{T.val := F.val;}
F → '(' E ')'	{F.val := E.val;}
F → **digit**	{F.val := **digit**.val;}

Attribute val represents the value of a construct

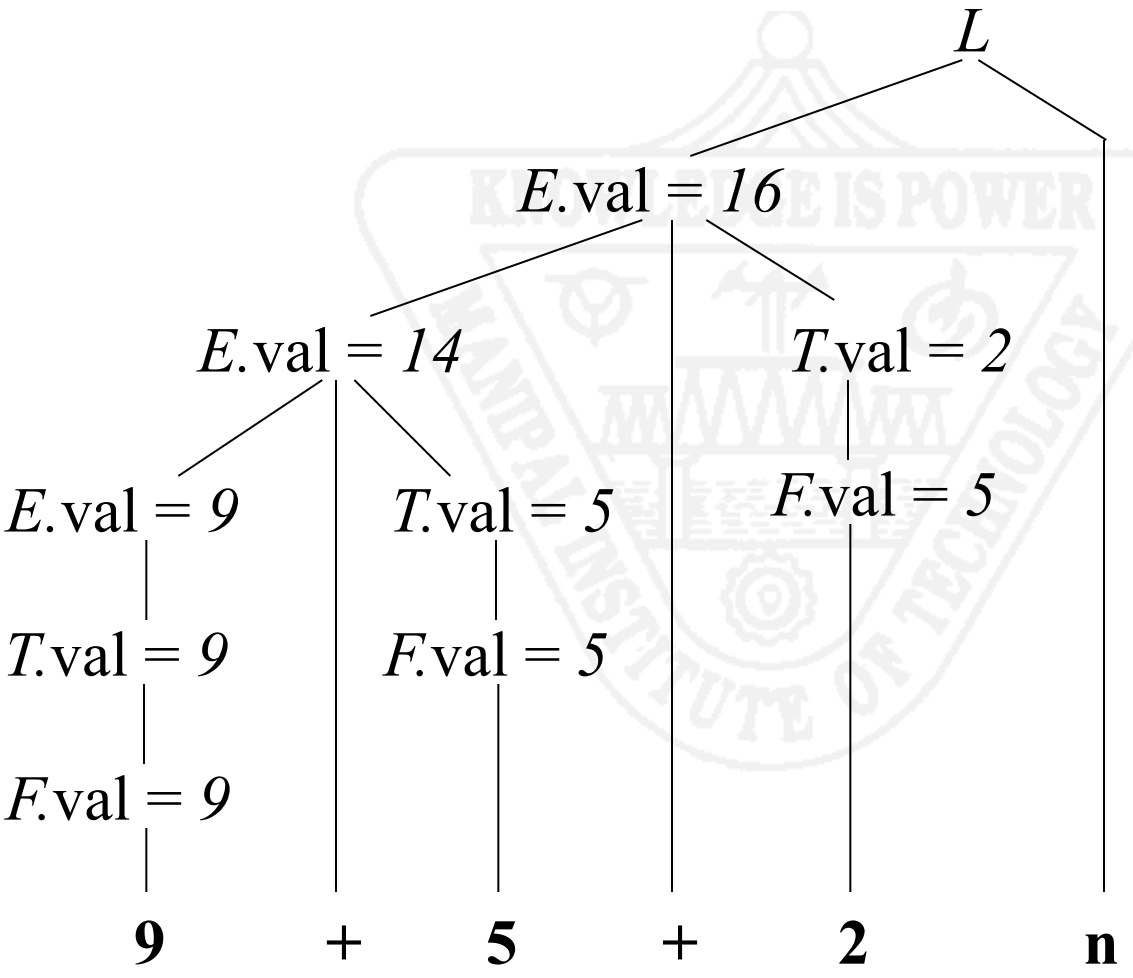
Annotated Parse Trees

3 * 5 + 4

L → E '\n'	{print(E.val);}
E → E_1 '+' T	{E.val := E_1.val + T.val;}
E → T	{E.val := T.val;}
T → T_1 '*' F	{T.val := T_1.val * F.val;}
T → F	{T.val := F.val;}
F → '(' E ')'	{F.val := E.val;}
F → **digit**	{F.val := **digit**.val;}



Example Annotated Parse Tree



Note: all attributes in this example are of the synthesized type

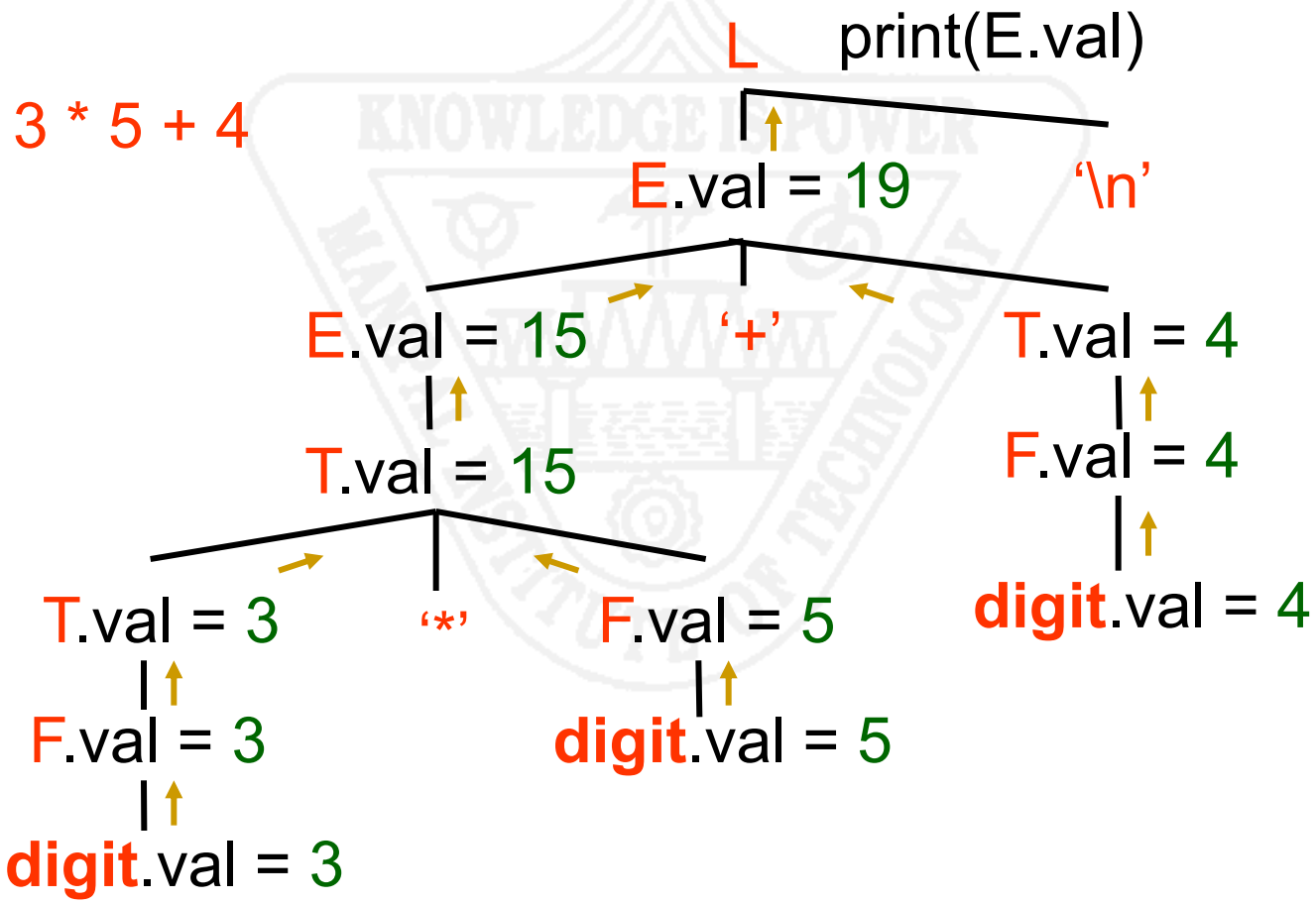
Semantic Attributes

- A (semantic) attribute of a node (grammar symbol) in the parse tree is synthesized if its value is computed from that of its children
- An attribute of a node in the parse tree is inherited if its value is computed from that of its parent and siblings

Synthesized Attributes

$L \rightarrow E$ '\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**.val;}

Synthesized Attributes

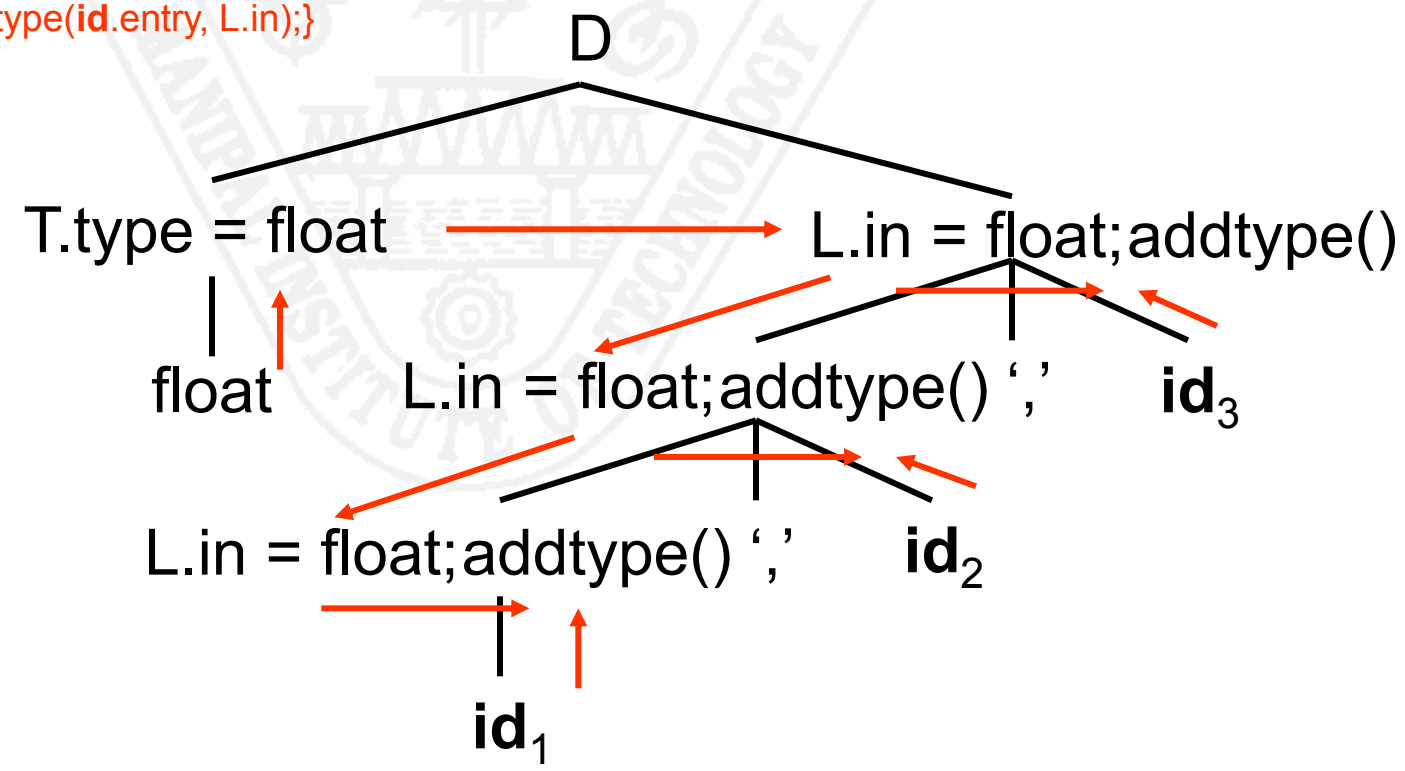


Inherited Attributes

D → T	{L.in := T.type;} L
T → **int**	{T.type := integer;}
T → **float**	{T.type := float;}
L → L_1 ‘,’ **id**	{L_1.in := L.in;}
	{addtype(**id**.entry, L.in);}
L → **id**	{addtype(**id**.entry, L.in);}

Inherited Attributes

D → T	{L.in := T.type;} L
T → **int**	{T.type := integer;}
T → **float**	{T.type := float;}
L → L_1 ‘,’ **id**	{L_1.in := L.in;} {addtype(**id**.entry, L.in);}
L → **id**	{addtype(**id**.entry, L.in);}



Dependencies of Attributes

- In the semantic rule

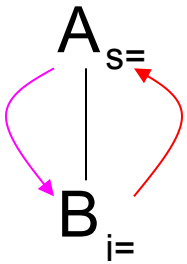
  $b := f(c_1, c_2, \ldots, c_k)$

  we say b depends on $c_1, c_2, \ldots, c_k$
- The semantic rule for b must be evaluated after the semantic rules for $c_1, c_2, \ldots, c_k$
- The dependencies of attributes can be represented by a directed graph called dependency graph

Attribute Dependencies

Circular dependences are a problem

Productions	Semantic Actions
A → B	A.s = B.i B.i = A.s + 1



S-Attributed Attribute Grammars

- An attribute grammar is S-attributed if it uses synthesized attributes exclusively

An Example

$L \rightarrow E$ '\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**.val;}

L-Attributed Attribute Grammars

- An attribute grammar is L-attributed if each attribute computed in each semantic rule for each production

  $$A \rightarrow X_1 X_2 \ldots X_n$$

  is a synthesized attribute, or an inherited attribute of X_j, $1 \leq j \leq n$, depending only on

  1. the attributes of $X_1, X_2, \ldots, X_{j-1}$
  2. the inherited attributes of A

An Example

$D \rightarrow T\ L$ {L.in := T.type;}

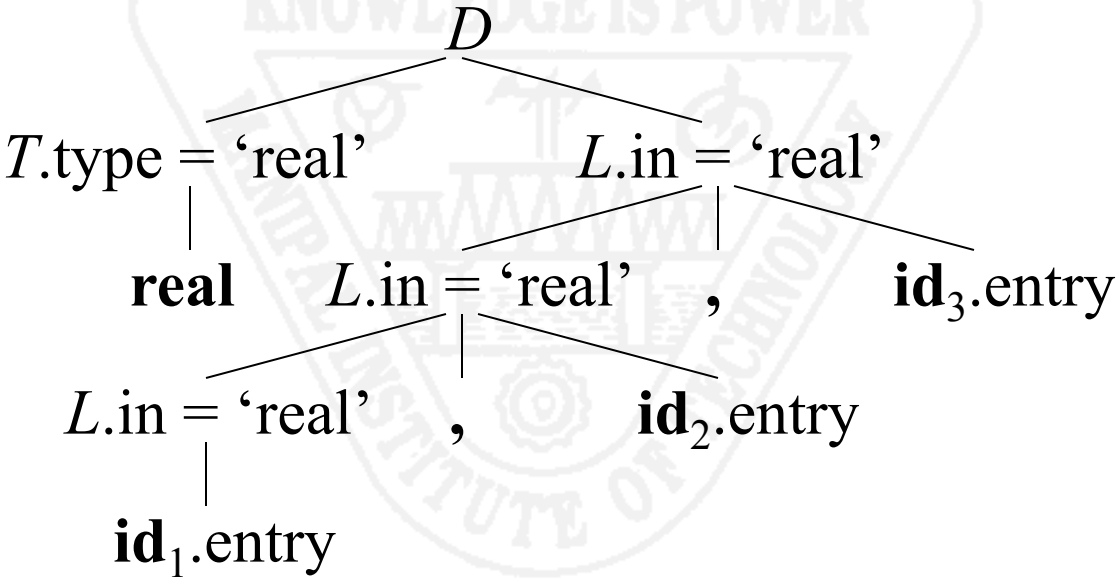
$T \rightarrow$ **int** {T.type := integer;}

$T \rightarrow$ **real** {T.type := real;}

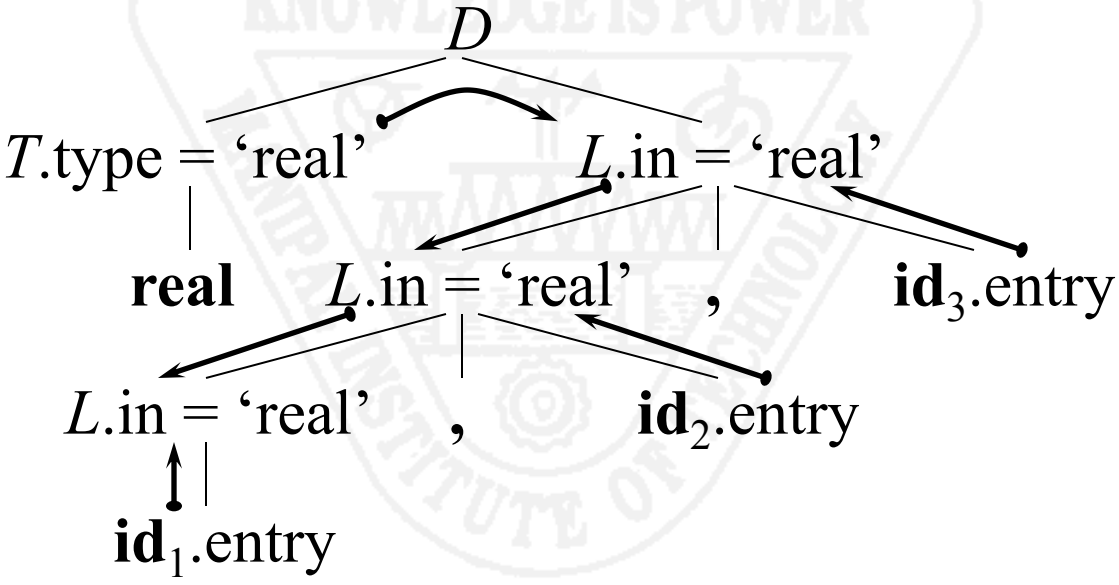
$L \rightarrow L_1$ ',' **id** {L_1.in := L.in;
addtype(**id**.entry, L.in);}

$L \rightarrow$ **id** {addtype(**id**.entry, L.in);}

Example Annotated Parse Tree

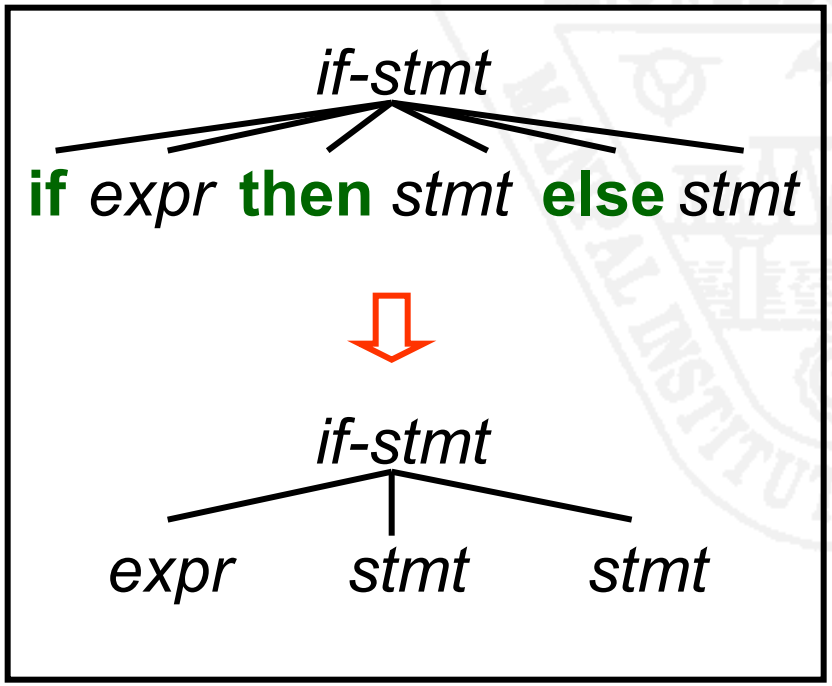


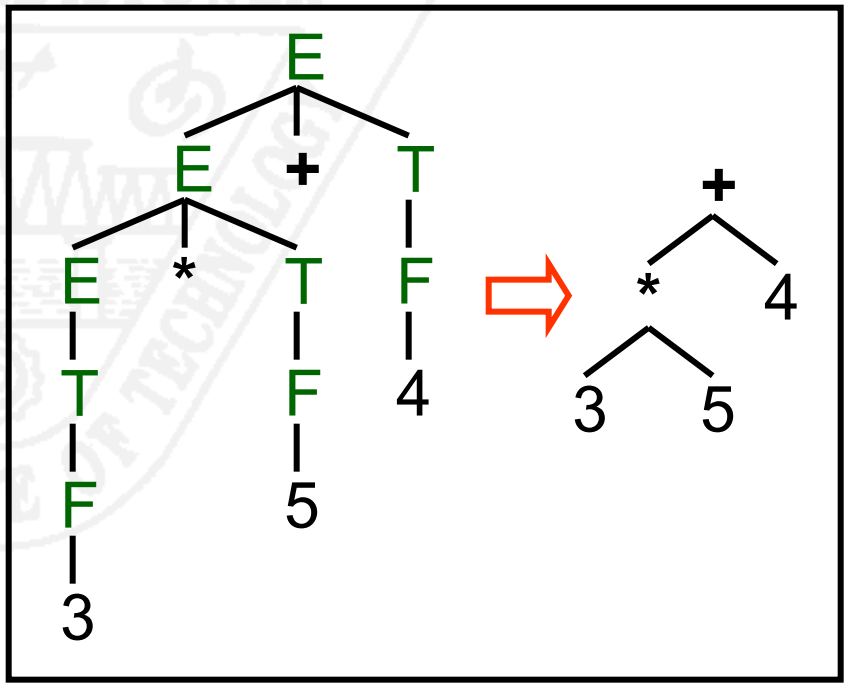
Example Annotated Parse Tree with Dependency Graph



Construction of Syntax Trees

- An abstract syntax tree is a condensed form of parse tree





Syntax Trees for Expressions

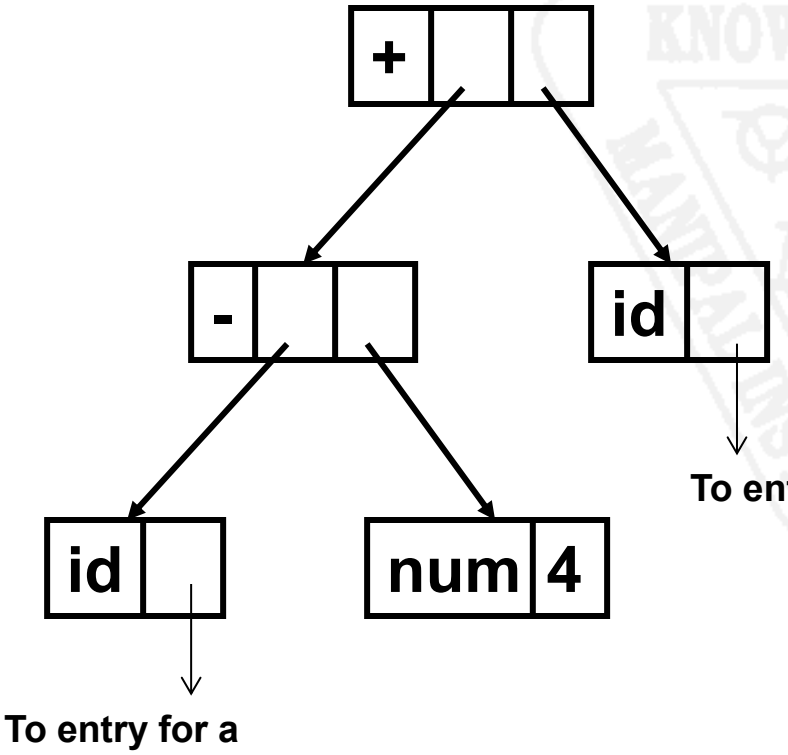
- Interior nodes are operators
- Leaves are identifiers or numbers
- Functions for constructing nodes
  - node(op, c1,c2, . . . , ck)
  - leaf(op, val)

An Example

$E \rightarrow E_1$ '+' T	{E.ptr := mknode('+', E_1.ptr, T.ptr);}
$E \rightarrow E_1$ '-' T	{E.ptr := mknode('-', E_1.ptr, T.ptr);}
$E \rightarrow T$	{E.ptr := T.ptr;}
T → '(' E ')'	{T.ptr := E.ptr;}
T → **id**	{T.ptr := mkleaf(id, **id**.entry);}
T → **num**	{T.ptr := mkleaf(num, **num**.value);}

An Example

a - 4 + c



p1 := new leaf(id, entry$_a$);
p2 := new leaf(num, 4);
p3 := new node('-', p1, p2);
p4 := new leaf(id, entry$_c$);
p5 := new node('+', p3, p4);