SYNTAX DIRECTED TRANSLATION

INTRODUCTION

- It is not possible for CFG to represent certain properties such as
  - uniqueness in type declarations
  - Type compatibility in performing arithmetic operations …
- Some features are based on construct of the language.
- Simple syntax analysis is not sufficient.

INTRODUCTION[CONTD..]

- *Syntax-directed translation refers to a method of compiler implementation where the* source language translation is completely driven by the parser.
- Parsing process and parse trees are used to direct semantic analysis and the translation of the source program.
- SDT is done during parsing.

INTRODUCTION[CONTD..]

- A syntax directed definitions specifies the values of attributes by associating semantic rules with grammar productions.
- Ex:

Production	Semantic Rule
E → E1+T	E.Code = E1.code || T.code ||'+'

Above production has two NT, 'E' and 'T', 'E1' is used to distinguish occurrence of 'E' in the body of prodn.

- Both NT have string valued attribute code.
- The Semantic rule specifies that string E.code is formed by concatenating E1.code,T.code and '+'.

INTRODUCTION[CONTD..]

- Another Notation
- Syntax Directed Translation(SDT) scheme embeds program fragments called semantic actions within production bodies.
- Ex: E → E1 + T { print '+' }
- Commonly used is first notation.
- Order of evaluation of semantic rules are explicitly specified

SYNTAX DIRECTED DEFINATIONS(SDD)

- SDD is a CFG together with attributes and rules.
- Attributes → grammar symbols
- Rules → productions
- If ‘X’ is a symbol and ‘a’ is one of its attribute then X.a denote the value ‘a’ at a particular node labeled ‘X’ of parse tree.

SDD[CONTD…]

- Attribute can be any property of a programming language construct.
- An attribute has a name and an associated value.
  - string
  - number
  - Type
  - memory location
  - an assigned register
- whatever information we need.

SDD[CONTD…]

- With each production in a grammar, we give semantic rules or *actions*.
- *Describe* how to compute the attribute values associated with each grammar symbol in a production.

TYPES OF ATTRIBUTES

- There are two types of attributes we might encounter: synthesized or inherited
- *Synthesized attributes are those attributes that are passed up a parse tree, i.e.,* attribute of *the left -side* non terminal is computed from the right-side attributes.
- The lexical analyzer usually supplies the attributes of terminals and the synthesized ones are built up for the nonterminals and passed up the tree.

SYNTHESIZED ATTRIBUTE

- For a NT ‘A’ at a parse tree node N is defined by a semantic rule associated with the production at N.
- synthesized Attribute at node N is defined only in terms of attribute values at the children of N and at N itself.

INHERITED ATTRIBUTE

- *Inherited attributes are those that are passed down a parse tree, i.e., the right-side* attributes are derived from the left-side attributes (or other right-side attributes).
- These attributes are used for passing information about the context to nodes further down the tree.

INHERITED ATTRIBUTES

- An inherited attribute for a nonterminal B at a parse tree node N is defined by a semantic rule associated with the production at the parent of N.
- inherited Attribute at node N is defined only in terms of attribute values at the parent of N ,at N itself , and N's siblings.

SDD EXAMPLE

	PRODUCTION	SEMANTIC RULES
1)	$L \rightarrow E$ **n**	$L.val = E.val$
2)	$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \rightarrow T$	$E.val = T.val$
4)	$T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \rightarrow F$	$T.val = F.val$
6)	$F \rightarrow (E)$	$F.val = E.val$
7)	$F \rightarrow$ **digit**	$F.val =$ **digit**.lexval

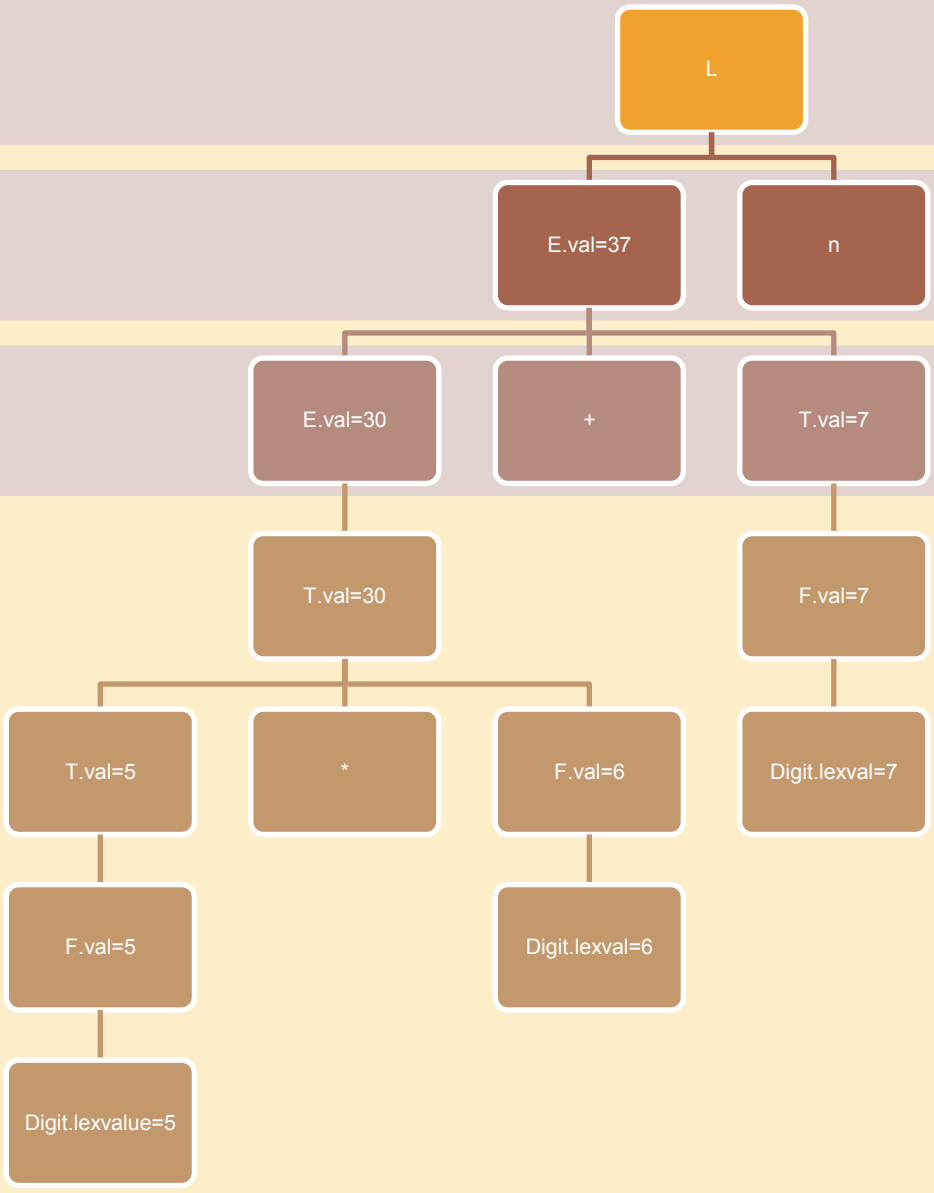
S- ATTRIBUTED SDD

- An SDD that involves only synthesized attributes is called S - attributed.

EXERCISE

- Draw annotated parse tree for input string 5*6+7

5*6+7

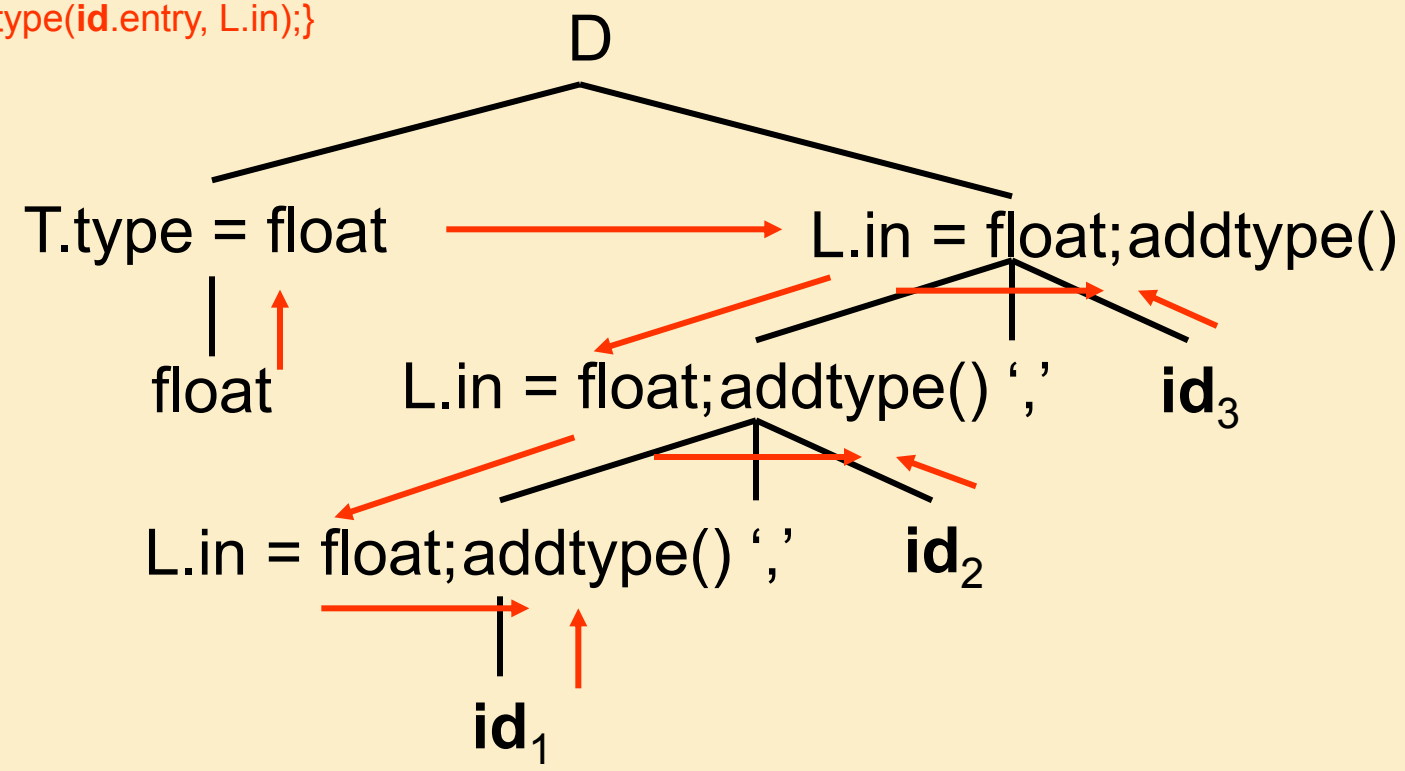

L
E.val=37
n
E.val=30
+
T.val=7
T.val=30
F.val=7
T.val=5
*
F.val=6
Digit.lexval=7
F.val=5
Digit.lexval=6
Digit.lexvalue=5

INHERITED ATTRIBUTES

D → T L	{L.in := T.type;}
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);}

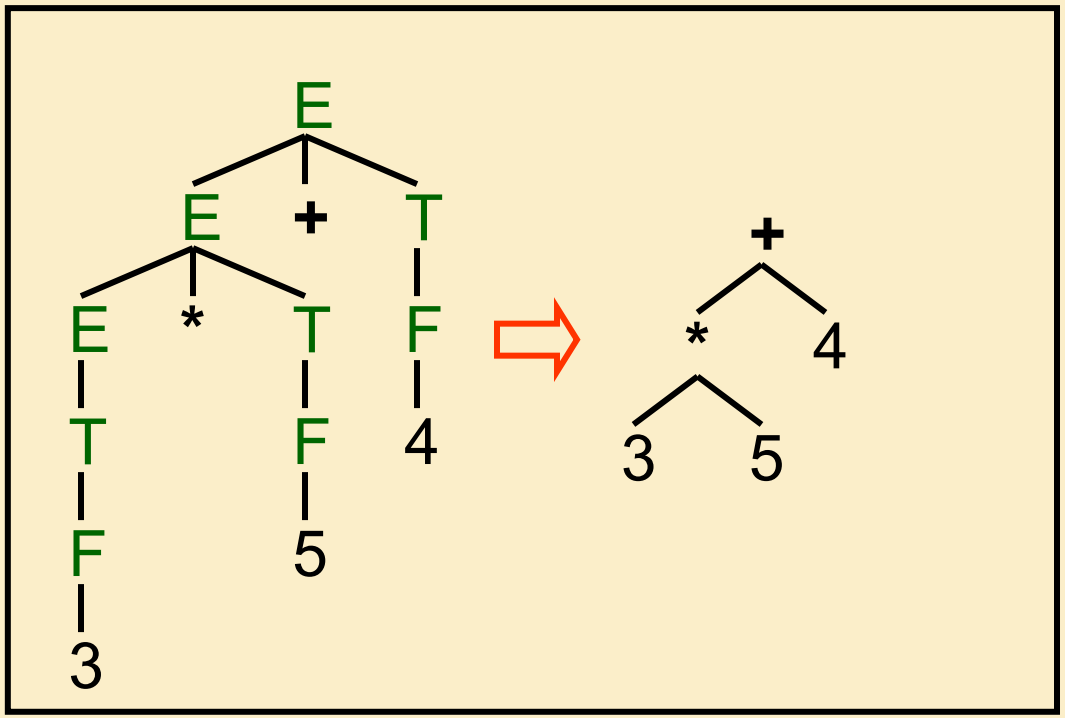
INHERITED ATTRIBUTES

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



CONSTRUCTION OF SYNTAX TREES

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



SYNTAX TREES FOR EXPRESSIONS

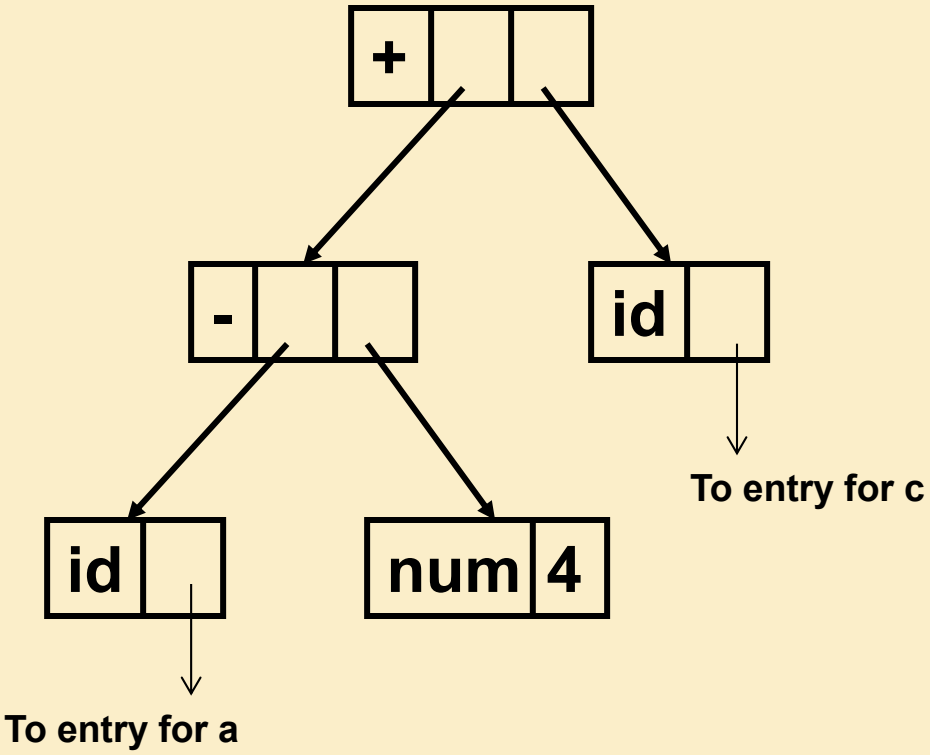
- Interior nodes are operators
- Leaves are identifiers or numbers
- Functions for constructing nodes
  - node(op, c1,c2, . . . , ck)
  - leaf(op, val)

AN EXAMPLE

	PRODUCTION	SEMANTIC RULES
1)	$E \rightarrow E_1 + T$	$E.node = \textbf{new}\ Node('+', E_1.node, T.node)$
2)	$E \rightarrow E_1 - T$	$E.node = \textbf{new}\ Node('-', E_1.node, T.node)$
3)	$E \rightarrow T$	$E.node = T.node$
4)	$T \rightarrow (E)$	$T.node = E.node$
5)	$T \rightarrow \textbf{id}$	$T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$
6)	$T \rightarrow \textbf{num}$	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$

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);

EXAMPLE

- Draw syntax tree for x*y-5+z