UNIT-3.2
SYNTAX-DIRECTED TRANSLATION

SIVA KUMAR RONANKI

Syntax-directed Definition(SDD):

SDD= CGF+SEMANTIC RULES

- A SDD is a context free grammar together with semantic rules.
- Attributes are associated with grammar symbols and semantic rules are associated with productions.
- If ‘x’ is a symbol and ‘a’ is one of its attribute then x.a denotes value at node ‘x’.
- Attributes may be of many kinds: numbers, types, table references, strings, etc.

Production	Semantic Rule
E->E+T	E.val=E.val+T.val
E->T	E.val=T.val

Syntax-directed Definition(SDD):

TYPES OF ATTRIBUTES:

- Synthesized attributes:

A synthesized attribute at node N is defined only in terms of attribute values of children of N.

EX: A->BCD, A be a parents node B,C,D are children nodes.

A.S=B.S

A.S=C.S

A.S=D.S Parent node A taking value from its children B,C,D.

- Inherited attributes:

An inherited attribute at node N is defined only in terms of attribute values at N's parent, N itself and N's siblings.

EX: A->BCD

C.i=A.i----→parent node

C.i=B.i----→sibling

C.i=D.i----→sibling

SDD of Simple Desk Calculator:

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

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

Syntax-directed Definition(SDD):

TYPES OF SDD:

1.S-Attributed SDD (or) S-Attributed Definitions (or)S-Attributed grammar.
2.L-Attributed SDD (or) L-Attributed Definitions (or)L-Attributed grammar.

S-Attributed SDD :

1.A SDD that uses only synthesized Attributes is called as S-Attributed SDD.

EX: A->BCD

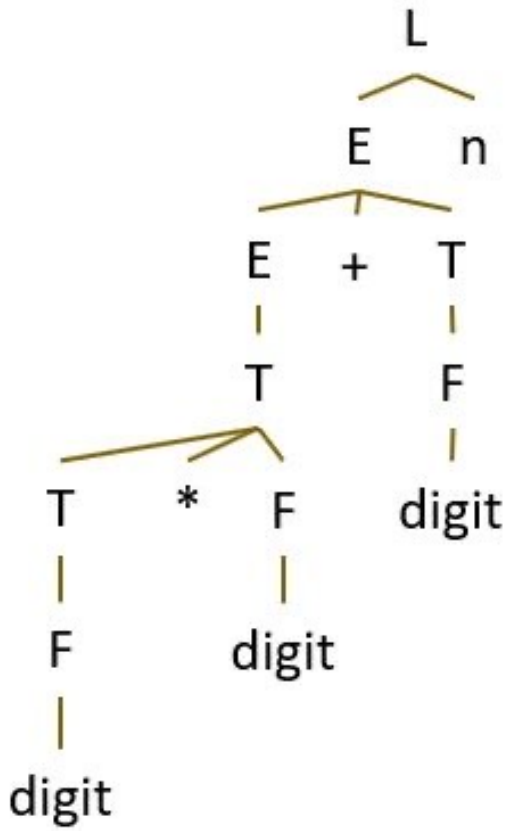
A.S=B.S

A.S=C.S

2.Semantic actions are always place at right end of the production. It is also called as “postfix SDD”.

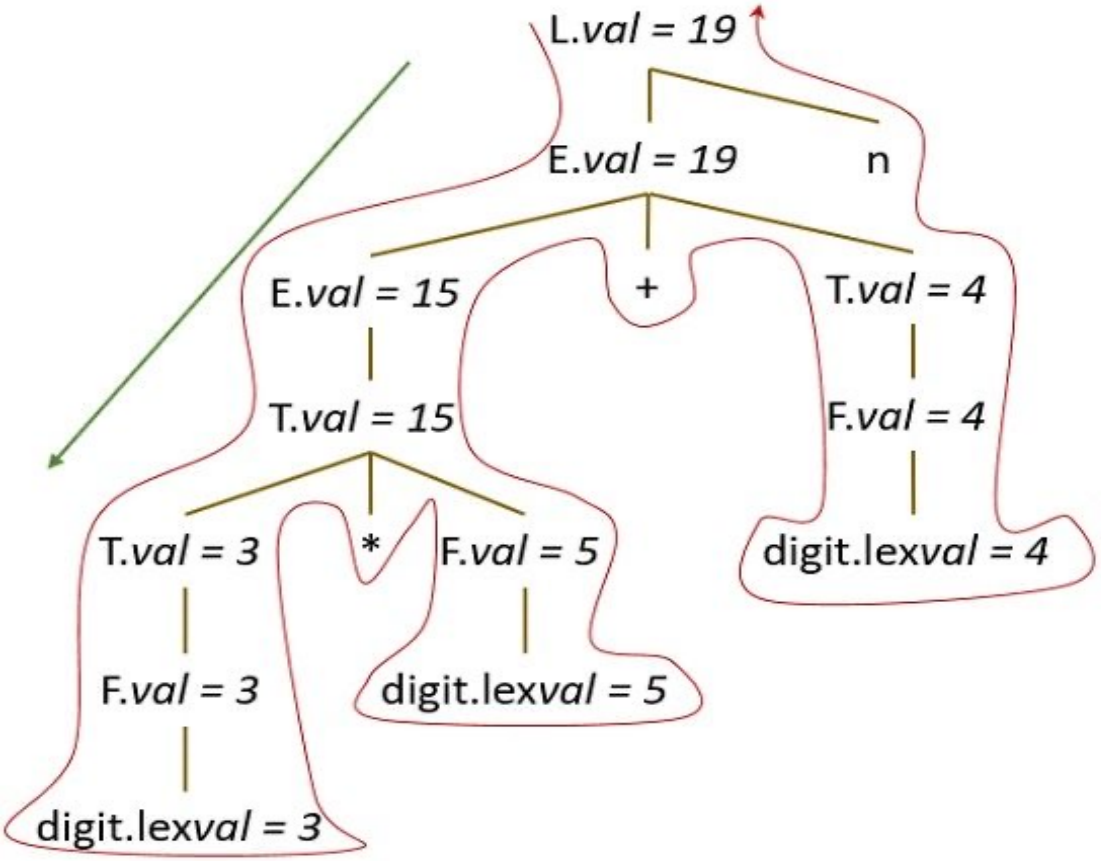
3.Attributes are evaluated with bottom-up parsing.

Parse Tree -- Example



Parse Tree for Input String 3 * 5 + 4 n

Annotated Parse Tree -- Example



Annotated Parse Tree for Input String 3 * 5 + 4 n

Syntax-directed Definition(SDD):

L-Attributed SDD :

1. A SDD that uses both synthesized & inherited attributed is called as L-Attributed is restricted to inherits from parent or left sibling only.

   Ex: A->XYZ {Y.S=A.S, Y.S=X.S, Y.S=Z.S}

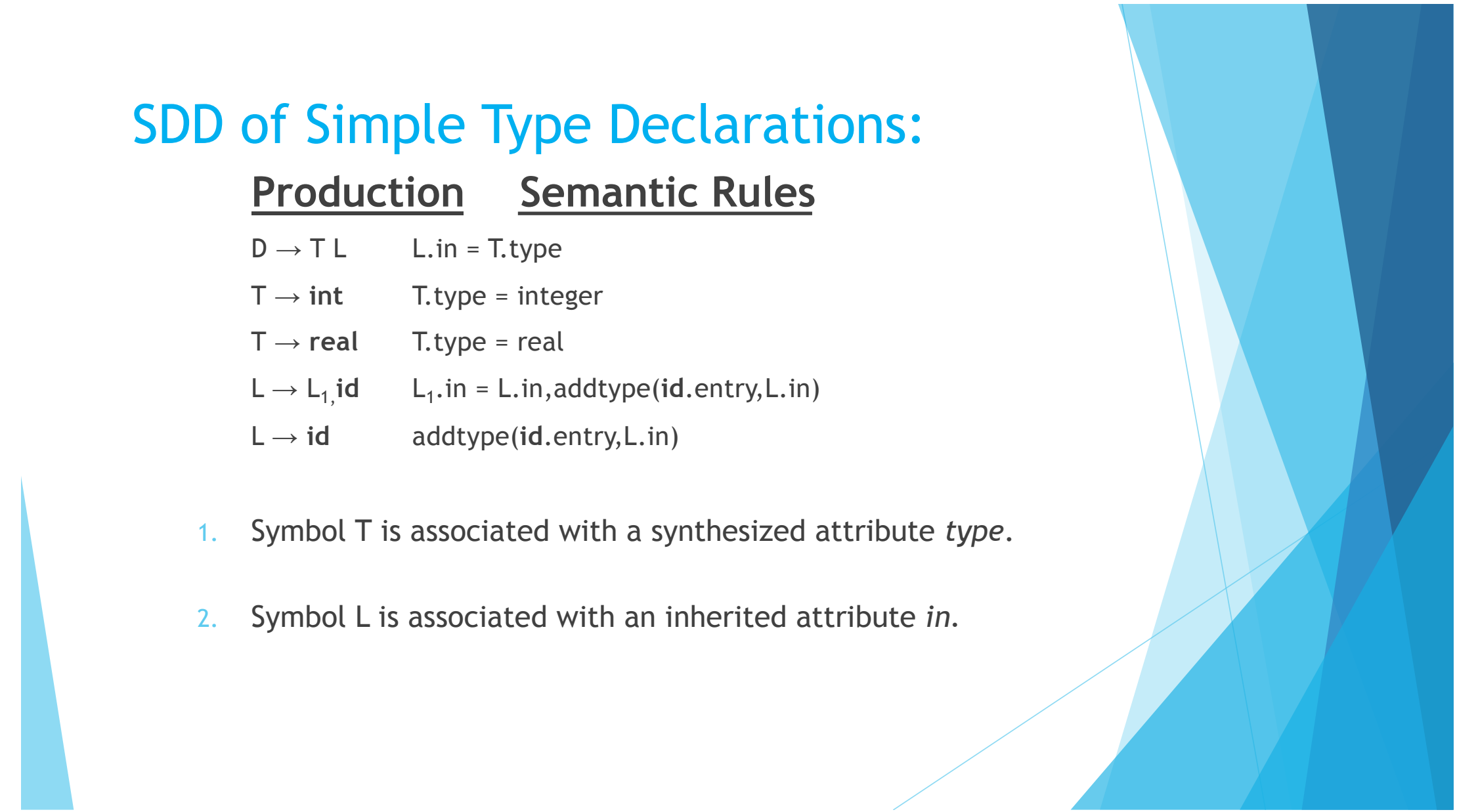
2. Semantic Action are placed anywhere on R.H.S

3. Attributes are evaluated by traversing parse tree depth first, left to right order.

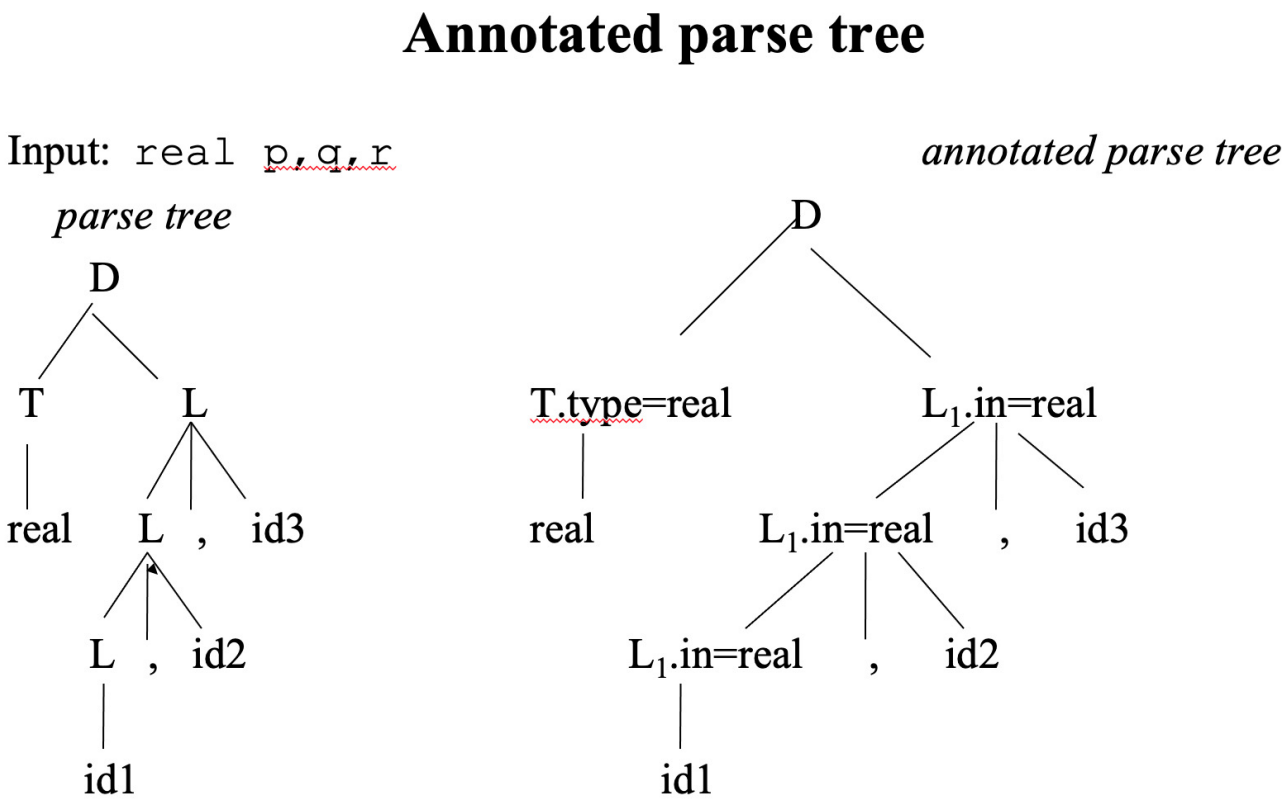
SDD of Simple Type Declarations:

Production	Semantic Rules
$D \rightarrow T\ L$	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)

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



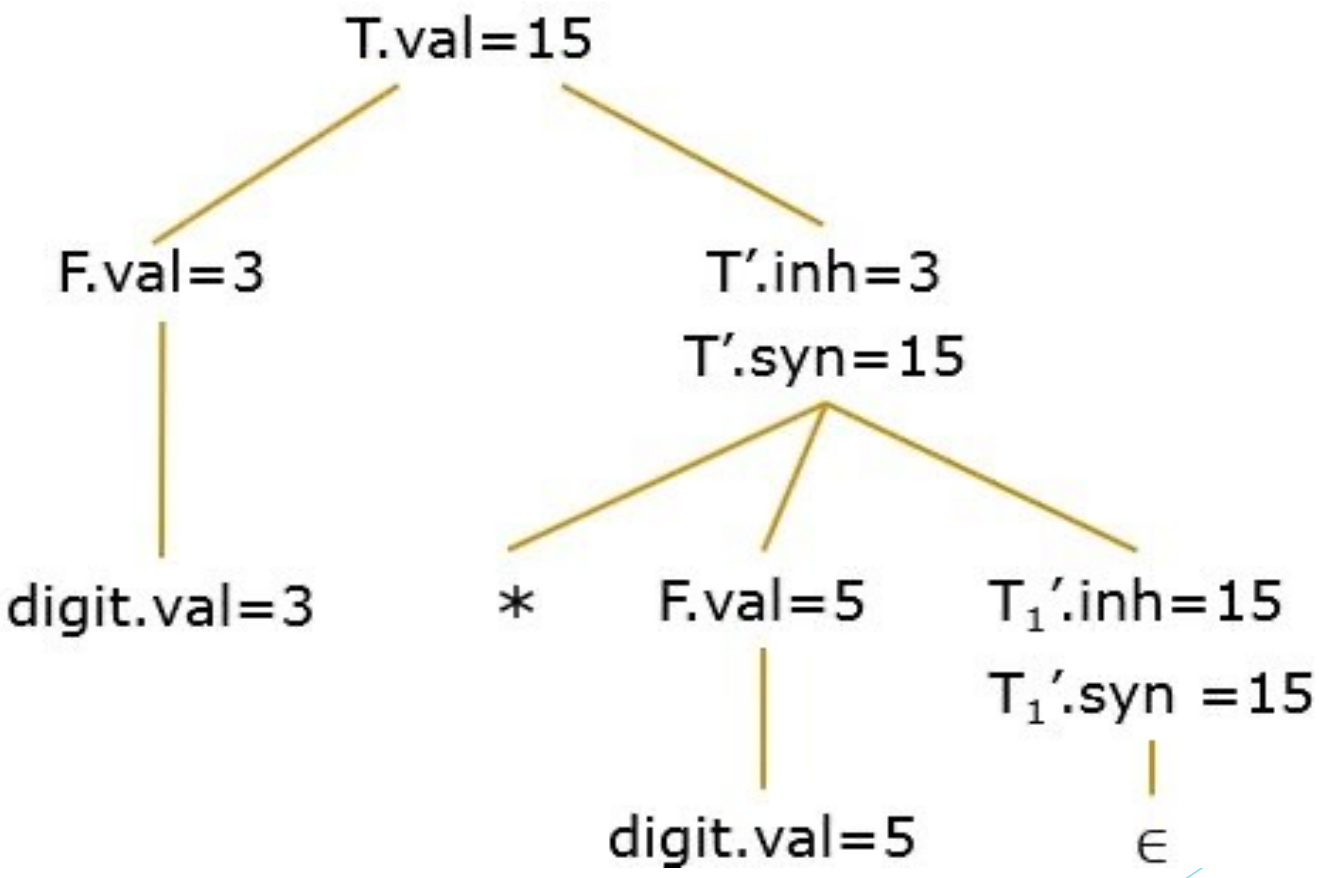
Annotated parse tree



SDD for a grammar:

Production	Semantic Rules
1) T -> FT’	T’.inh = F.val T.val = T’.syn
2) T’ -> *FT’$_1$	T’$_1$.inh = T’.inh*F.val T’.syn = T’$_1$.syn
3) T’$_1$ -> ε	T’$_1$.syn= T’$_1$.inh
1) F -> digit	F.val = F.val = digit.lexval

Annotated parse tree



Dependency Graph:

- Dependency Graph represents the flow of information among the attributes in a parse tree
- Dependency Graphs are useful for determining evaluation order for attributes in a parse tree.
- While an annotated parse tree shows the values of attributes, a dependency graph determines how those values can be computed.

Production	Semantic Rules
$D \rightarrow T\ L$	L.in = T.type
$T \rightarrow$ **int**	T.type = integer
$T \rightarrow$ **real**	.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)

Evaluation Order

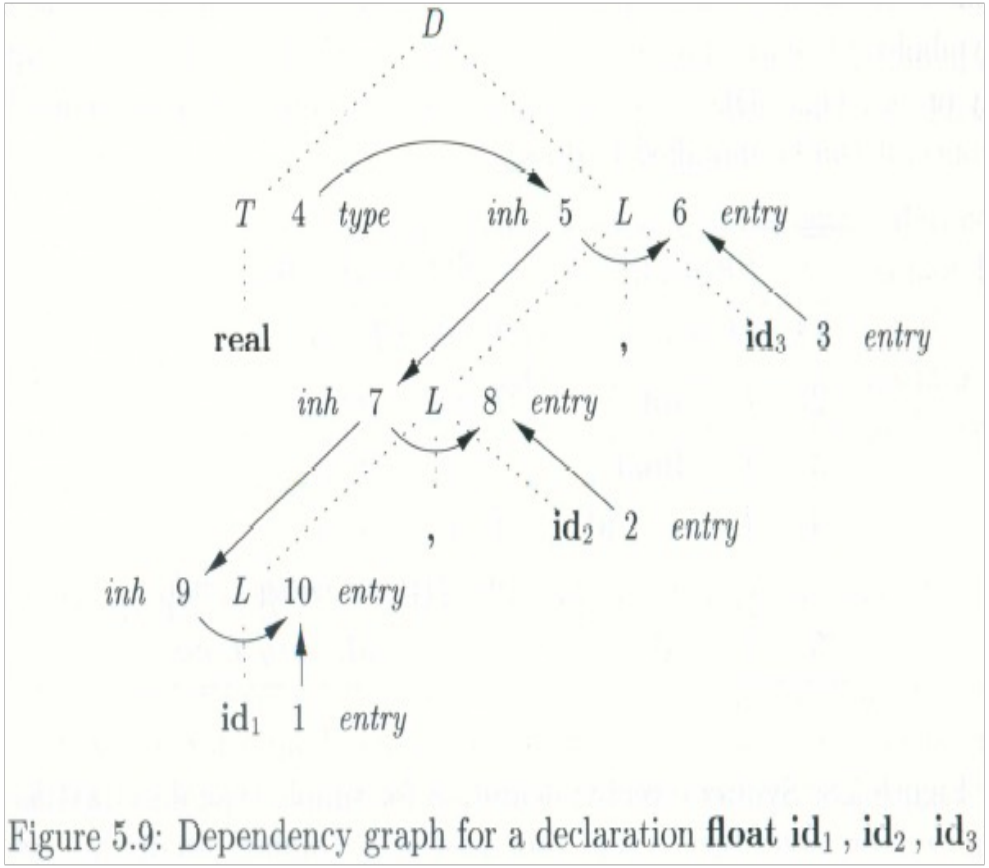


Figure 5.9: Dependency graph for a declaration **float** id_1 , id_2 , id_3

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

Translation Scheme:

- In a syntax-directed definition, we do not say anything about the evaluation times of the semantic rules (when the semantic rules associated with a production should be evaluated).
- Translation schemes describe the order and timing of attribute computation.
- A **translation scheme** is a context-free grammar in which:

  – attributes are associated with the grammar symbols and

  – semantic actions enclosed between braces {} are inserted within the right sides of productions.

  Each semantic rule can only use the information compute by already executed semantic rules.

- *Ex:* $A \rightarrow \{ \ldots \} X \{ \ldots \} Y \{ \ldots \}$

  Semantic Actions

A Translation Scheme Example

• A simple translation scheme that converts infix expressions to the corresponding postfix expressions.

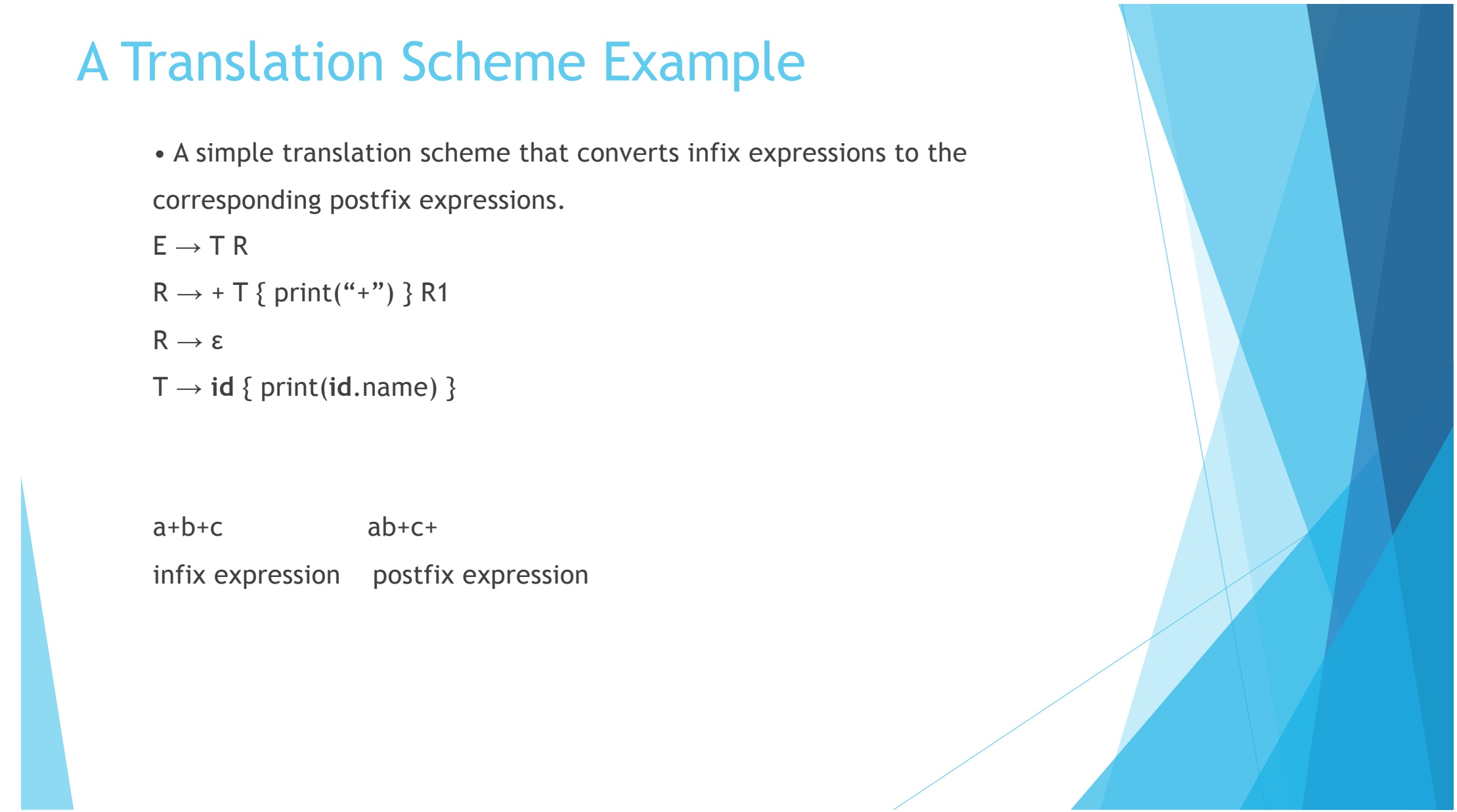
E → T R

R → + T { print(“+”) } R1

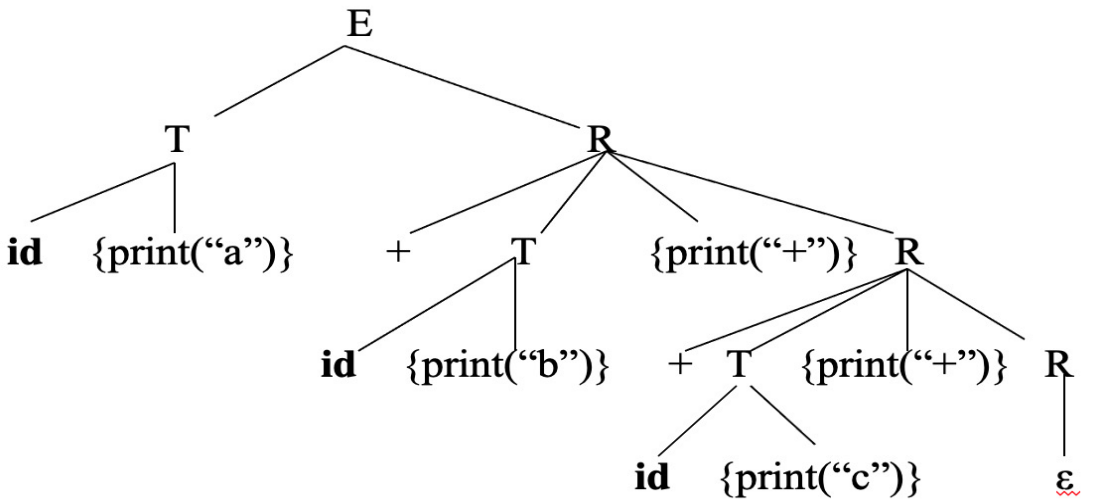
R → ε

T → **id** { print(**id**.name) }

a+b+c	ab+c+
infix expression	postfix expression



A Translation Scheme Example (cont.)



The depth first traversal of the parse tree (executing the semantic actions in that order) will produce the postfix representation of the infix expression.

Translation Schemes: Example

$S \rightarrow A_1 A_2 \quad \{A_1.in=1; \quad A_2.in=2\}$

$A \rightarrow a \quad \{ print (A.in)\}$