Semantic analysis

- Semantic analysis is a phase by a compiler that adds **semantic information to the parse tree** (known as **annotated parse tree**) and performs certain checks based on this information.
- It logically follows the parsing phase, in which the parse tree is generated, and logically precedes the code generation phase, in which (intermediate/target) code is generated.
- In a compiler implementation, it may be possible to fold different phases into one pass.
- Semantic information that is added and checked is **type checking**
- Compiler usually maintains **symbol table** in which it stores what each symbol (variable names, function names, etc.) refers to.

Semantic Errors

- Type mismatch
- Undeclared variable
- Multiple declaration of variable in a scope
- Accessing an out of scope variable
- Actual and formal parameter mismatch

Syntax Directed Definition(SDD)

- Syntax Directed Definition is a generalization of context-free grammar in which:
  1. Grammar symbols have an associated **set of Attributes**
  2. Productions are associated with **Semantic Rules** for computing the values of attributes.
- Such formalism generates **Annotated Parse Trees** where each node of the tree is a record with a field for each attribute (Eg., X.a indicates the attribute a of the grammar symbol X).
- The value of an attribute of a grammar symbol at a given parse-tree node is defined by a semantic rule associated with the production used at that node.

Attributes

- Attributes are properties associated with grammar symbols. Attributes can be numbers, strings, memory locations, datatypes, etc.

  Eg: E → E + T   E.value = E.value + T.value

- Attributes can be broadly divided into two categories :

  synthesized attributes and **inherited attributes**

1. A *synthesized attribute* for a nonterminal A at a parse-tree node N is defined by a semantic rule associated with the production at N. Note that the production must have A as its head. A synthesized attribute at node N is defined only in terms of attribute values at the children of N and at N itself.

2. 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. Note that the production must have B as a symbol in its body. An inherited attribute at node N is defined only in terms of attribute values at N's parent, N itself, and N's siblings.

Evaluation of Synthesized Attributes

- Write the SDD using appropriate semantic rules for each production in given grammar.

<u>CFG</u>

$L \rightarrow E\ n$

$E \rightarrow E_1 + T$

$E \rightarrow T$

$T \rightarrow T_1 * F$

$T \rightarrow F$

$F \rightarrow (E)$

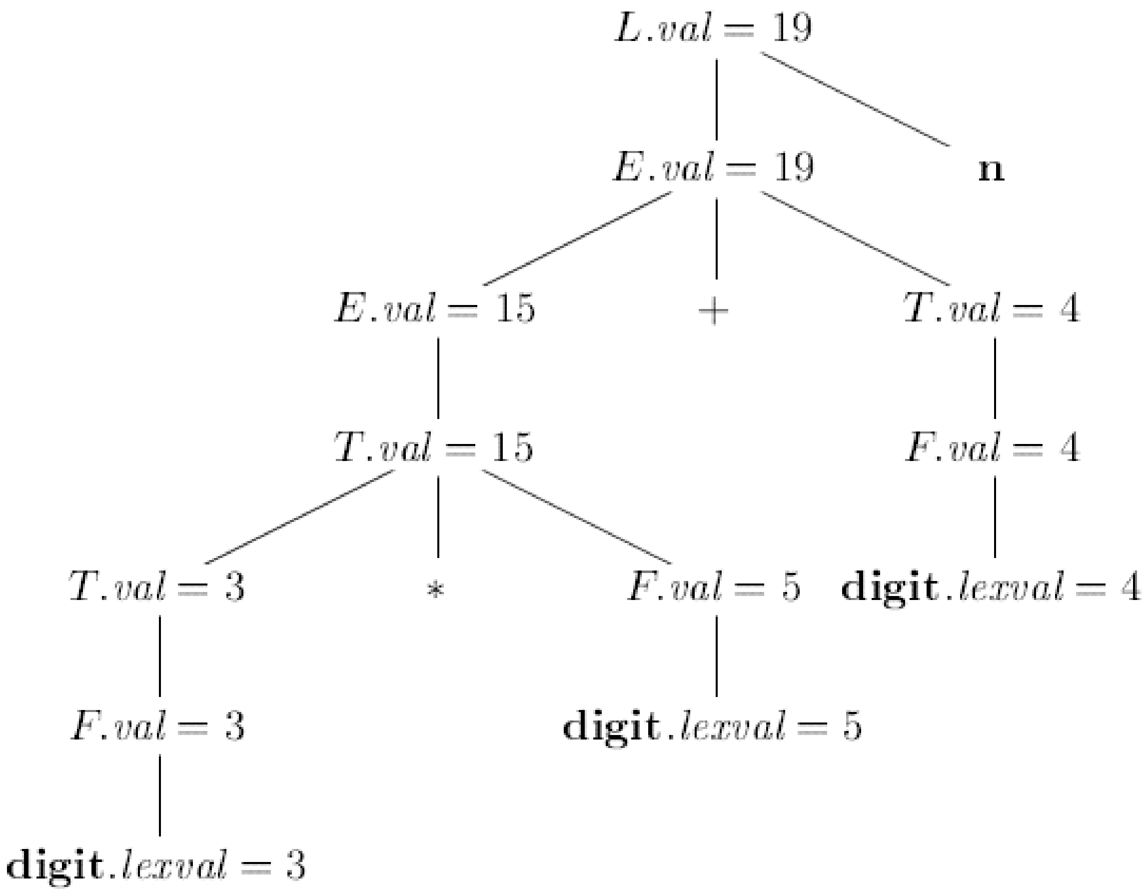
$F \rightarrow digit$

SDD

	Production	Semantic Rules
1)	$L \rightarrow E\ \mathbf{n}$	$L.val = E.val$
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)	$F \rightarrow \mathbf{digit}$	$F.val = \mathbf{digit}.\text{lexval}$

Annotated Parse Tree(Bottom up)

- The annotated parse tree is generated and attribute values are computed in bottom up manner for synthesized attributes.
- The value obtained at root node is the final output.



Annotated parse tree for $3 * 5 + 4\ \mathbf{n}$

Evaluation of Inherited Attributes

- Value of inherited attributes are computed by value of parent or sibling nodes.
- Using function addType the type of identifiers are inserted in symbol table at corresponding id.entry based on the declaration.

CFG

D → T L

T → int

T → float

T → double

L → L_1, id

L → id

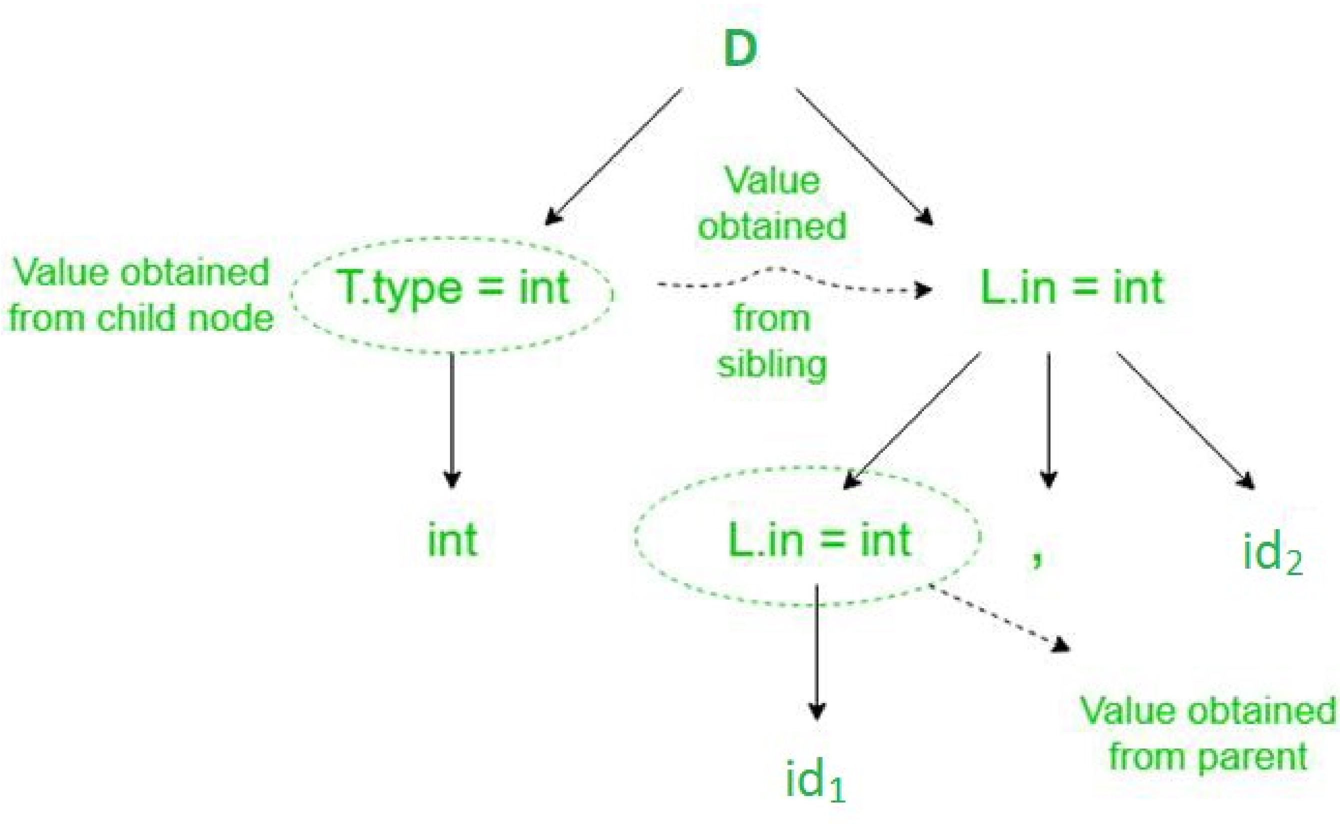
SDD

	PRODUCTION	SEMANTIC RULES
1)	$D \rightarrow T\ L$	$L.inh = T.type$
2)	$T \rightarrow \textbf{int}$	$T.type = \text{integer}$
3)	$T \rightarrow \textbf{float}$	$T.type = \text{float}$
4)	$L \rightarrow L_1\ ,\ \textbf{id}$	$L_1.inh = L.inh$ $addType(\textbf{id}.entry, L.inh)$
5)	$L \rightarrow \textbf{id}$	$addType(\textbf{id}.entry, L.inh)$

Annotated Parse Tree(Top down)

The annotated parse tree is generated and inherited attribute values are computed in top down manner.

- **type** is synthesized attribute and in is inherited attribute



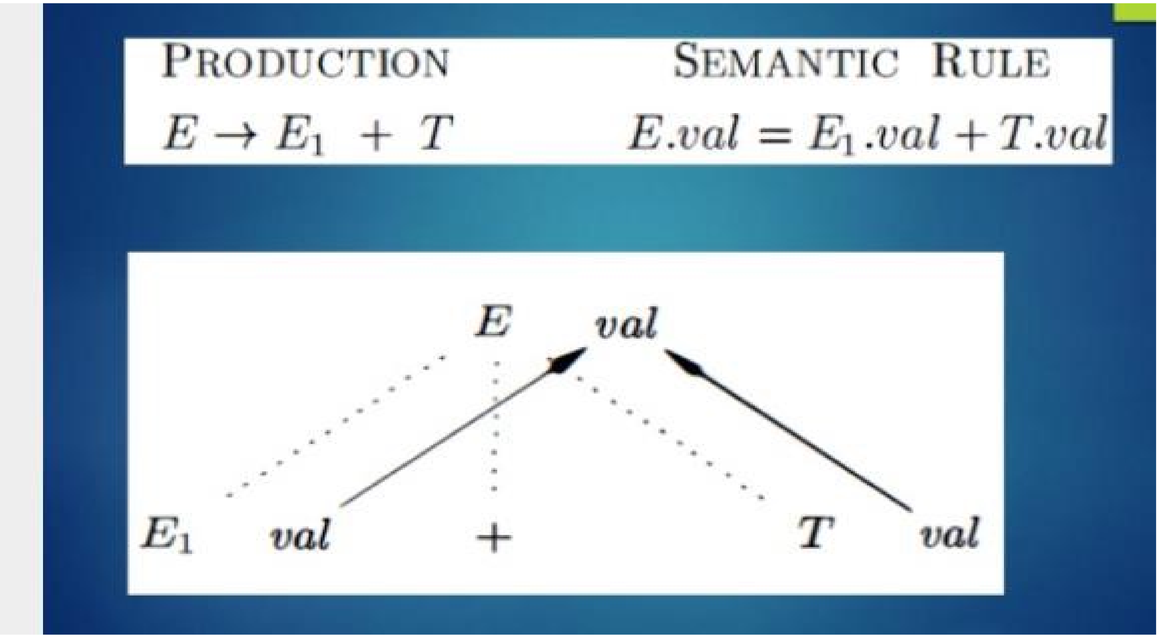
Input String: **int id_1 , id_2**

Dependency Graph

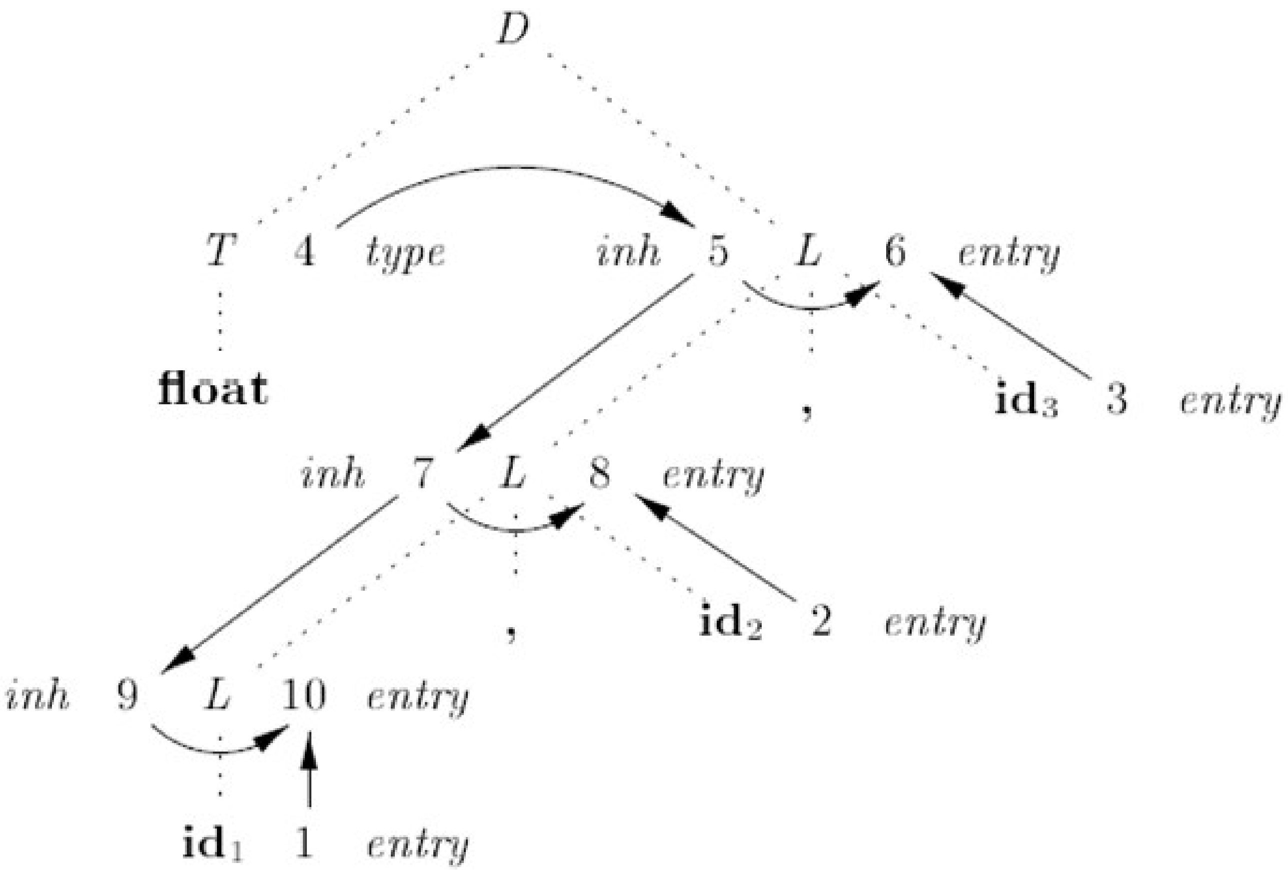
- Dependency Graph is constructed to find an **order for the evaluation of attributes** according to the sematic rules in SDD
- Each attribute value must be available when a computation is performed.
- Dependency Graphs are the most general technique used to evaluate syntax directed definitions **with both synthesized and inherited attributes.**
- Annotated parse tree shows the values of attributes, dependency graph helps to determine how(the order) those values are computed

Dependency Graph(contd..)

- If attribute **b depends on an attribute c** there is a link from the node for c to the node for b (**b ← c**) according to a **function or assignments**
- Dependency Rule: If an attribute b depends from an attribute c, then we **need to find the semantic rule for c first and then the semantic rule for b.**



Dependency Graph(contd..)



Dependency graph for a declaration **float id**$_1$, **id**$_2$, **id**$_3$

Evaluation Order

	Production	Semantic Rules
1)	$D \rightarrow T\ L$	$L.inh = T.type$
2)	$T \rightarrow \textbf{int}$	$T.type = \text{integer}$
3)	$T \rightarrow \textbf{float}$	$T.type = \text{float}$
4)	$L \rightarrow L_1\ ,\ \textbf{id}$	$L_1.inh = L.inh$ $addType(\textbf{id}.entry, L.inh)$
5)	$L \rightarrow \textbf{id}$	$addType(\textbf{id}.entry, L.inh)$

1) type_4 = float
2) $\text{Inh}_5 = \text{type}_4$
3) addType(id_3.entry, inh_5)
4) $\text{Inh}_7 = \text{inh}_5$
5) addType(id_2.entry, inh_7)
6) $\text{Inh}_9 = \text{inh}_7$
7) addType(id_1.entry, inh_9)

Syntax Directed Translation Scheme

- It is a CFG with **program fragments embedded with the production body**
- The program fragments are called semantic actions which is enclosed between braces

Eg.

$E \rightarrow T\ R$

$R \rightarrow$ **addop** T { print(addop.lexeme)} R1 | ϵ

$T \rightarrow$ **num** {print(num.val)}

Postfix Translation Scheme

Here all the actions are in the right end

$$
\begin{array}{lcll}
L & \rightarrow & E\ \mathbf{n} & \{\ \text{print}(E.val);\ \} \\
E & \rightarrow & E_1 + T & \{\ E.val = E_1.val + T.val;\ \} \\
E & \rightarrow & T & \{\ E.val = T.val;\ \} \\
T & \rightarrow & T_1 * F & \{\ T.val = T_1.val \times F.val;\ \} \\
T & \rightarrow & F & \{\ T.val = F.val;\ \} \\
F & \rightarrow & (\ E\ ) & \{\ F.val = E.val;\ \} \\
F & \rightarrow & \mathbf{digit} & \{\ F.val = \mathbf{digit}.lexval;\ \}
\end{array}
$$

S-attributed Definition

- S stands for synthesized
- If SDT uses only synthesized attributes, it is called as S-attributed SDT.
- **S-attributed SDTs** are evaluated in **bottom-up parsing**, as the values of the **parent nodes depend upon the values of the child nodes.**

L-attributed Definition

- Dependency graph edges can go from left to right
- Inherited attributes with a restriction that inherited attribute can inherit values from parent and left siblings only

Rules

1. Synthesized, or

2. Inherited, but with the rules limited as follows. Suppose that there is a production $A \to X_1 X_2 \cdots X_n$, and that there is an inherited attribute $X_i.a$ computed by a rule associated with this production. Then the rule may use only:

   (a) Inherited attributes associated with the head A.

   (b) Either inherited or synthesized attributes associated with the occurrences of symbols $X_1, X_2, \ldots, X_{i-1}$ located to the left of X_i.

   (c) Inherited or synthesized attributes associated with this occurrence of X_i itself, but only in such a way that there are no cycles in a dependency graph formed by the attributes of this X_i.

Example

1) A →BC {B.a=A.a, C.a=B.a} D → **L-Attributed SDT**

- Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner.
- Semantic actions are placed anywhere in RHS satisfying the conditions.
  - A→{ }BC
  - A→ B{ }C
  - A→ BC{ }
- If an attribute is S attributed , it is also L attributed.

Note: **A →B {B.a=A.a} C {C.a=B.a} D will do the same action**

Example(contd..)

2) A →B {B.i = A.s} C {C.i = D.s} D is

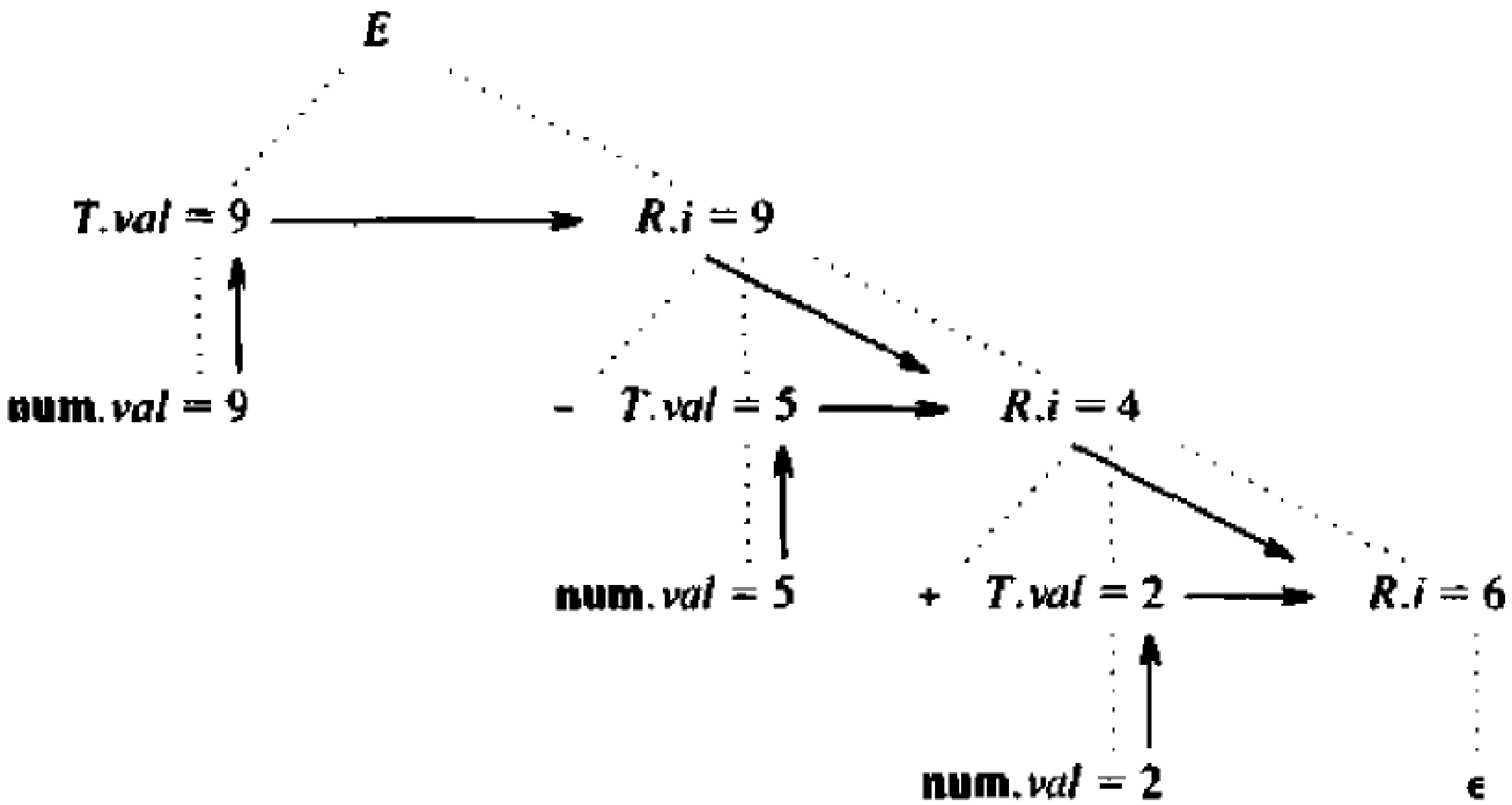
Since Rule 2b is violated in **{C.i = D.s}**

3**) A →B {B.i = A.s} C D {D.i = C.s} is**

Translation Scheme Example

$$
\begin{array}{lll}
E \rightarrow & T & \{\ R.i := T.val\ \} \\
 & R & \{\ E.val := R.s\ \} \\
 & & \\
R \rightarrow & + & \\
 & T & \{\ R_1.i := R.i + T.val\ \} \\
 & R_1 & \{\ R.s := R_1.s\ \} \\
 & & \\
R \rightarrow & - & \\
 & T & \{\ R_1.i := R.i - T.val\ \} \\
 & R_1 & \{\ R.s := R_1.s\ \} \\
 & & \\
R \rightarrow & \epsilon & \{\ R.s := R.i\ \} \\
 & & \\
T \rightarrow & (& \\
 & E & \\
 &) & \{\ T.val := E.val\ \} \\
 & & \\
T \rightarrow & \textbf{num} & \{\ T.val := \textbf{num}.val\ \}
\end{array}
$$

Evaluation of Expression 9-5+2



Bottom-up Evaluation of Inherited Attributes

INPUT	*stack*	PRODUCTION USED
real p,q,r	–	
p,q,r	real	
p,q,r	T	$T \rightarrow$ real
,q,r	T p	
,q,r	$T\ L$	$L \rightarrow$ id
q,r	$T\ L$,	
,r	$T\ L$, q	
,r	$T\ L$	$L \rightarrow L$, id
r	$T\ L$,	
	$T\ L$, r	
	$T\ L$	$L \rightarrow L$, id
	D	$D \rightarrow T\ L$

Bottom-up Evaluation Code

Production	Code Fragment
D → T L ;	
T → int	val [ntop] = integer
T → real	val [ntop] = real
L → L , id	addtype(val[top], val[top-3]
L → id	addtype(val[top], val[top-1]

- top and ntop of stack refers to top before and after reduction respectively
- addtype in code is equivalent to addtype(id.entry, T.type)

Type checking

- It is a check that the source program follows the syntax and semantics of the concerned language.
- The aim of type checking is to ensure that operations are used on the variable/expressions of the correct Types
- Some of the static check includes:
  1. **Type check**- Checks whether operators are applied to compatible operands
  2. **Flow of control check**-Statements that cause control flow must have some place to transfer the control flow.
  3. **Uniqueness check** – Object must be defined exactly once for some scenarios
  4. **Name related check**-Sometimes same name must appear two or more times
  5. **Function Check**-Function call with correct number arguments and type

Type System

- A type is a set of values and operations on those values
- A language's type system specifies which operations are valid for a type
- Languages can be divided into three categories with respect to the type

  1. Untyped
  - No type checking needs to be done
  - Assembly languages

  2. Statically typed
  - All type checking is done at compile time
  - Also called strongly typed

  3. Dynamically typed
  - Type checking is done at run time
  - Languages like Lisp

Type Expression

- Type of a language construct is denoted by a type expression
- It is either a basic type or it is formed by applying operator called ***type constructor*** to other type expressions

  1. A basic type is a type expression
     - boolean, char, integer, real
     - *type error*: error during type checking
     - *void*: no type value
  2. Since type expressions may be named, a type name is a type expression
  3. A type constructor applied to a type expression is a type expression

Type Constructors

Array: if T is a type expression then array(I, T) is a type expression denoting the type of an array with elements of type T and index set I

int A[10];

A can have type expression array(0 .. 9, integer)

Product: if T1 and T2 are type expressions then their Cartesian product T1 X T2 is a type expression

Type Constructors(contd..)

Records: it applies to a tuple formed from field names and field types. Consider the declaration

type row = record

addr : integer;

lexeme : array [1 .. 15] of char

end;

var table: array [1 .. 10] of row;

- The type row has type expression

record ((addr X integer) X (lexeme X array(1 .. 15,char)))

and type expression of table is array(1 .. 10, row)

Type Constructors(contd..)

Pointer: if T is a type expression then pointer(T) is a type expression denoting type “pointer to an object of type T”

Function: function maps domain set to range set. It is denoted by type expression

$D \rightarrow R$

– Function with two integer argument and the return type is also integer

$int \times int \rightarrow int$

– The type of function int* f(char a, char b) is denoted by

$char \times char \rightarrow pointer(int)$

Type Checking for Expressions

E → literal { E.type = char}

E → num {E.type = integer}

E → id {E.type = lookup(id.entry)}

E → E1 mod E2 { E.type = if E1.type == integer and E2.type==integer then integer else type_error}

E → E1[E2] {E.type = if E2.type==integer and E1.type==array(s,t) then t else type_error}

E → *E1 {E.type = if E1.type==pointer(t) then t else type_error}

Type Checking of Functions

E → E1 (E2) { E. type = E1.type == s → t and E2.type == s) then t
else type-error}

Type Checking for Statements

S → id := E {S.type = if id.type == E.type then void else type_error}

S → if E then S1 {S.type = if E.type == Boolean then S1.type else type_error}

S → while E do S1 {S.type = if E.type == Boolean then S1.type else type_error}

S → S1 ; S2 {S.type = if S1.type == void and S2.type == void
then void else type_error}

Coercion

- Coercion is the implicit type conversion that is performed in compile time
- The given SDD shows the coercion instead of the explicit type conversion function *inttoreal*

PRODUCTION	SEMANTIC RULE
$E \rightarrow$ **num**	$E.type$:= *integer*
$E \rightarrow$ **num . num**	$E.type$:= *real*
$E \rightarrow$ **id**	$E.type$:= *lookup* (**id**.*entry*)
$E \rightarrow E_1$ **op** E_2	$E.type$:= **if** $E_1.type$ = *integer* **and** $E_2.type$ = *integer* **then** *integer* **else if** $E_1.type$ = *integer* **and** $E_2.type$ = *real* **then** *real* **else if** $E_1.type$ = *real* **and** $E_2.type$ = *integer* **then** *real* **else if** $E_1.type$ = *real* **and** $E_2.type$ = *real* **then** *real* **else** *type_error*