# Syntax Directed Translation

Akshaya Arunan, Govt. Engg College, Trivandrum

# Syntax-directed translation



- Syntax-directed translation (SDT) refers to a method of compiler implementation where the source language translation is completely driven by the parser.
- The parsing process and parse trees are used to direct semantic analysis and the translation of the source program.
- We can augment grammar with information to control the semantic analysis and translation. Such grammars are called **attribute grammars.**

Akshaya Arunan, Govt. Engg College, Trivandrum

# Syntax-directed translation



- Associate attributes with each grammar symbol that describes its properties.
- An attribute has a name and an associated value.
- With each production in a grammar, give semantic rules or actions.
- The general approach to syntax-directed translation is to construct a parse tree or syntax tree and compute the values of attributes at the nodes of the tree by visiting them in some order.

# Syntax-directed translation



- There are two ways to represent the semantic rules associated with grammar symbols.
  - Syntax-Directed Definitions (SDD)
  - Syntax-Directed Translation Schemes (SDT)

Akshaya Arunan, Govt. Engg College, Trivandrum

# Syntax-Directed Definitions



- A syntax-directed definition (SDD) is a context-free grammar together with attributes and rules.
- Attributes are associated with grammar symbols and rules are associated with productions.

| PRODUCTION | SEMANTIC RULE |
|---|---|
| $E \rightarrow E1 + T$ | E.code = E1.code \|\| T.code \|\| ‘+’ |

Akshaya Arunan, Govt. Engg College, Trivandrum

# Syntax-Directed Definitions



- SDDs are highly readable and give high-level specifications for translations.
- But they hide many implementation details.
- For example, they do not specify order of evaluation of semantic actions.
- Syntax-Directed Translation Schemes (SDT) embeds program fragments called semantic actions within production bodies
- SDTs are more efficient than SDDs as they indicate the order of evaluation of semantic actions associated with a production rule.

Akshaya Arunan, Govt. Engg College, Trivandrum

# Inherited Attributes



- An INHERITED ATTRIBUTE for a non-terminal B at a parse-tree node N is defined by a semantic rule associated with the production at the parent of N. 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.

# Synthesized Attributes



| 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**.$lexval$ |

- Each of the non-terminals has a single synthesized attribute, called val.
- An SDD that involves only synthesized attributes is called **S-attributed**
- Each rule computes an attribute for the non-terminal at the head of a production from attributes taken from the body of the production.

Akshaya Arunan, Govt. Engg College Trivandrum

# Evaluating an SDD at the Nodes of a Parse Tree

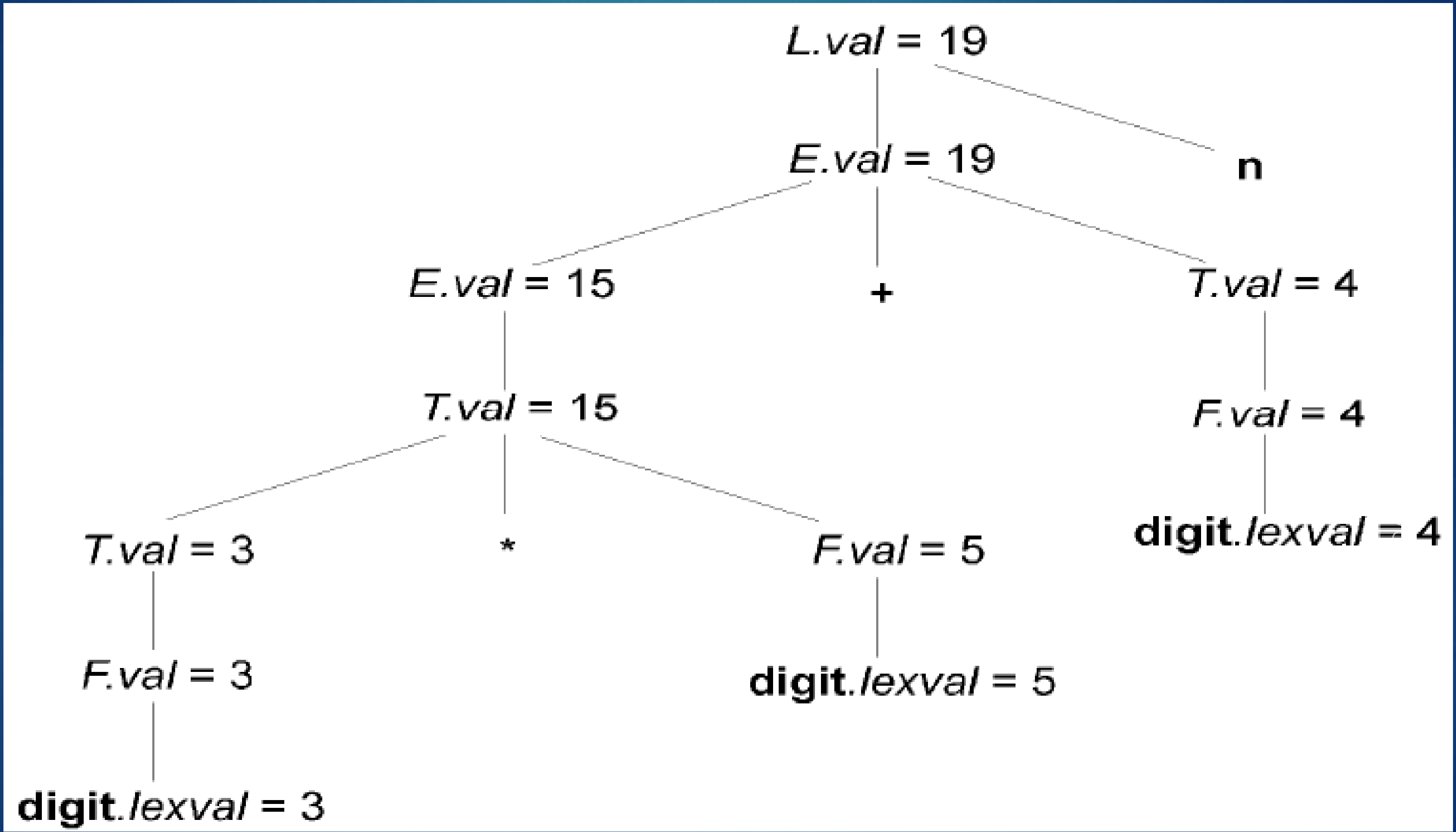


- A parse tree, showing the value(s) of its attribute(s) is called an **annotated parse tree.**
- With synthesized attributes, evaluate attributes in bottom-up order.

Akshaya Arunan, Govt. Engg College, Trivandrum

# Annotated Parse Tree for 3*5+4n





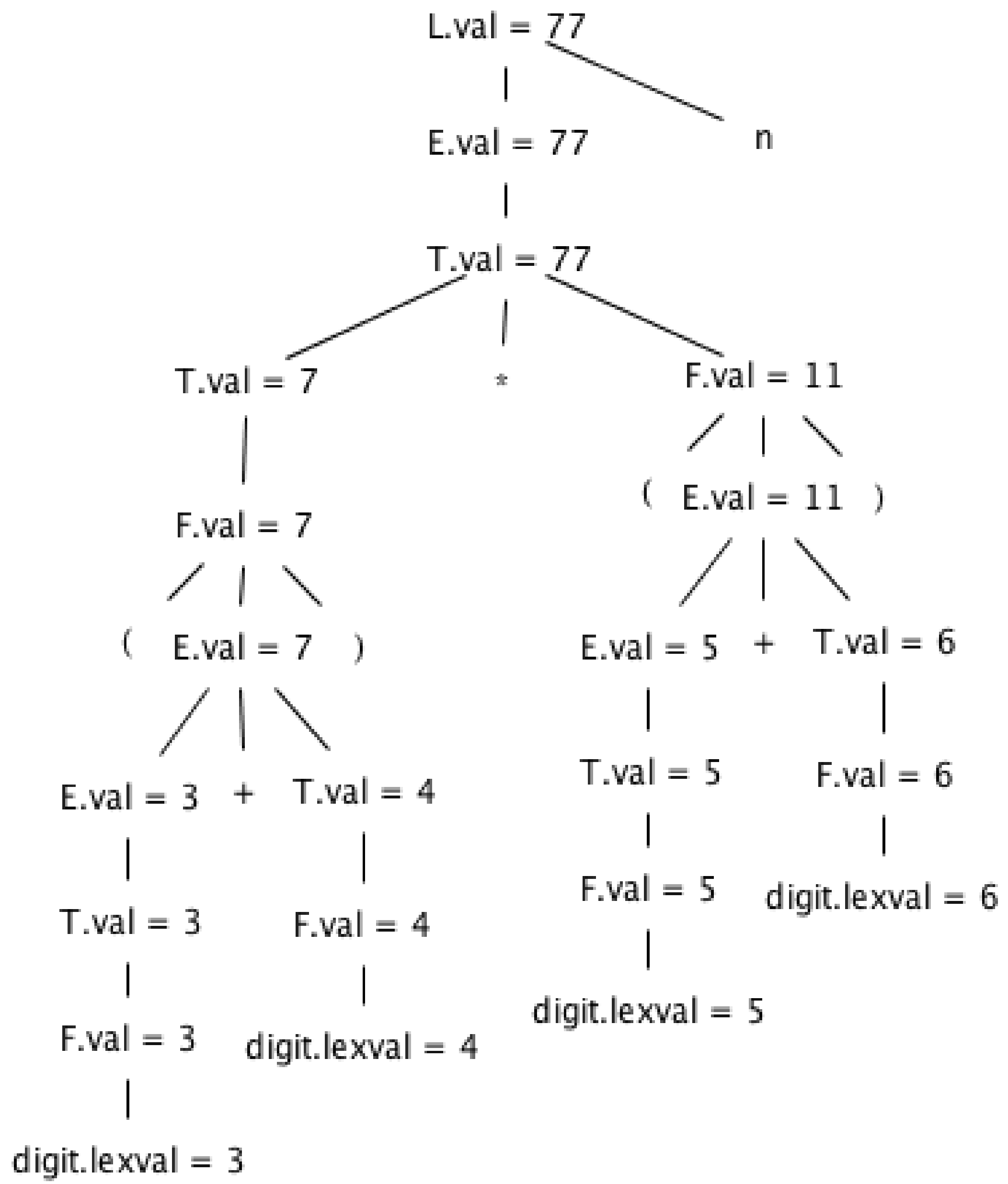
□give annotated parse trees for the following expressions:

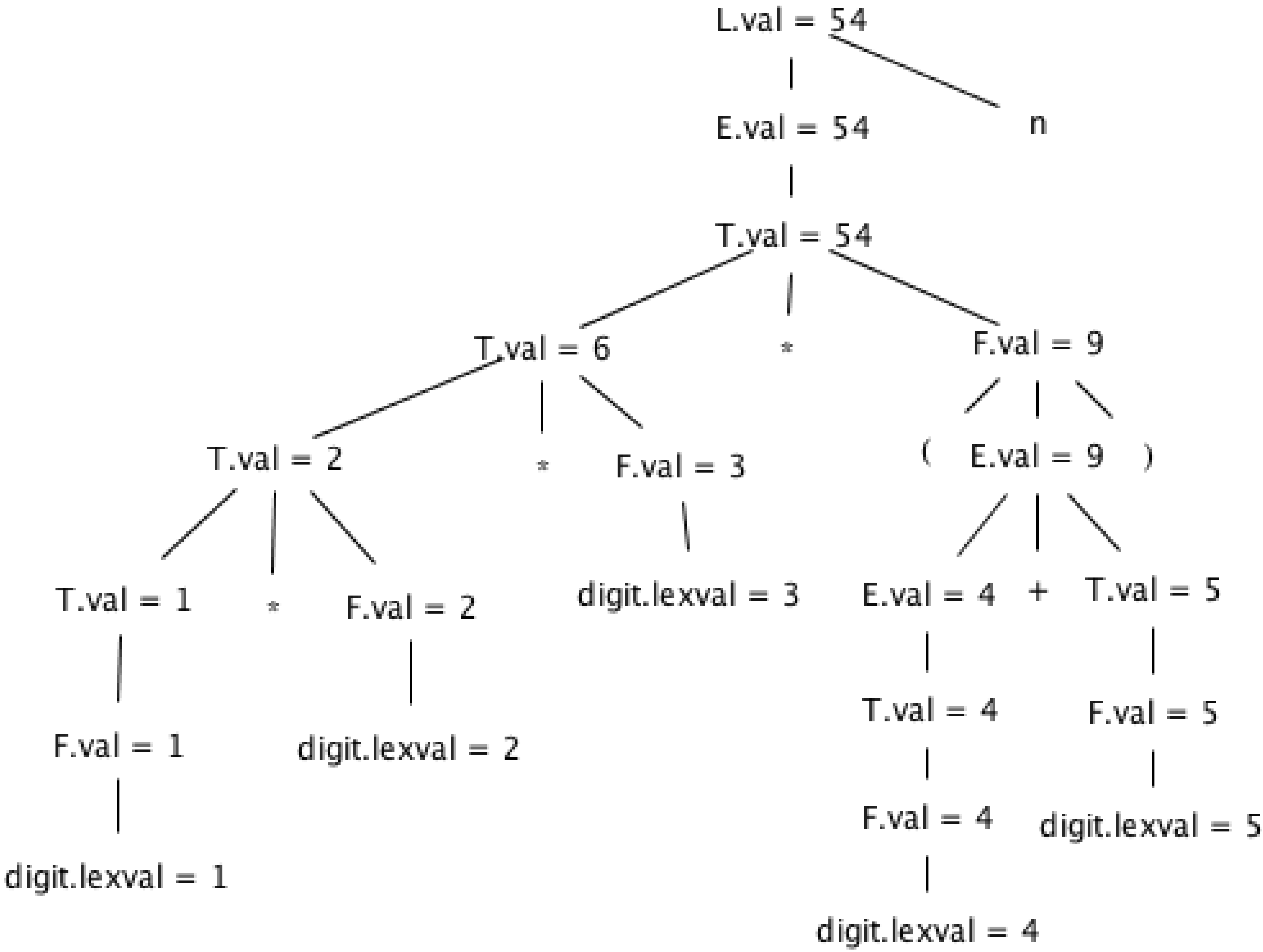
$(3 + 4) * (5 + 6)\ \mathbf{n}.$

$1 * 2 * 3 * (4 + 5)\ \mathbf{n}.$

$(9 + 8 * (7 + 6) + 5) * 4\ \mathbf{n}.$

Akshaya Arunan, Govt. Engg College, Trivandrum


L.val = 77
E.val = 77
n
T.val = 77
T.val = 7
*
F.val = 11
( E.val = 11 )
F.val = 7
( E.val = 7 )
E.val = 5
+
T.val = 6
E.val = 3
+
T.val = 4
T.val = 5
F.val = 6
T.val = 3
F.val = 4
F.val = 5
digit.lexval = 6
F.val = 3
digit.lexval = 4
digit.lexval = 5
digit.lexval = 3



L.val = 54
n
E.val = 54
T.val = 54
T.val = 6
*
F.val = 9
T.val = 2
*
F.val = 3
(
E.val = 9
)
T.val = 1
*
F.val = 2
digit.lexval = 3
E.val = 4
+
T.val = 5
F.val = 1
digit.lexval = 2
T.val = 4
F.val = 5
digit.lexval = 1
F.val = 4
digit.lexval = 5
digit.lexval = 4

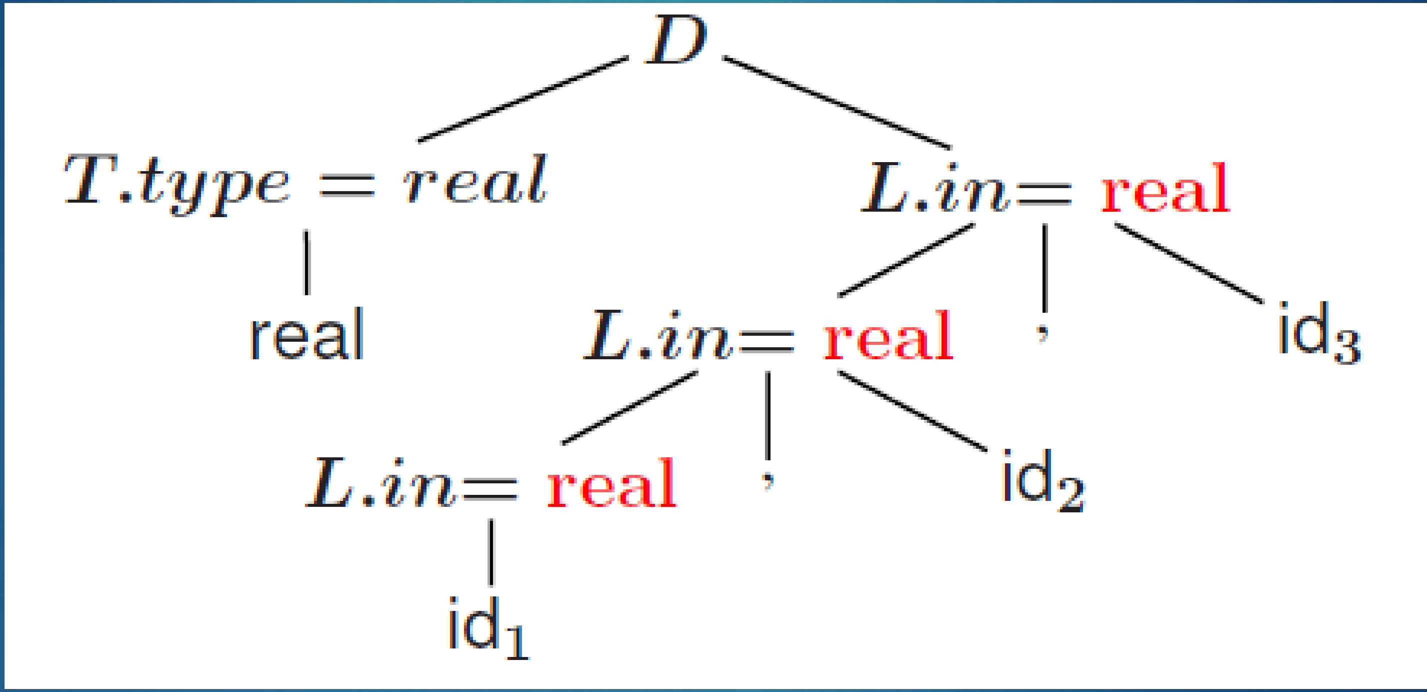
# SDD for expression grammar with inherited attributes



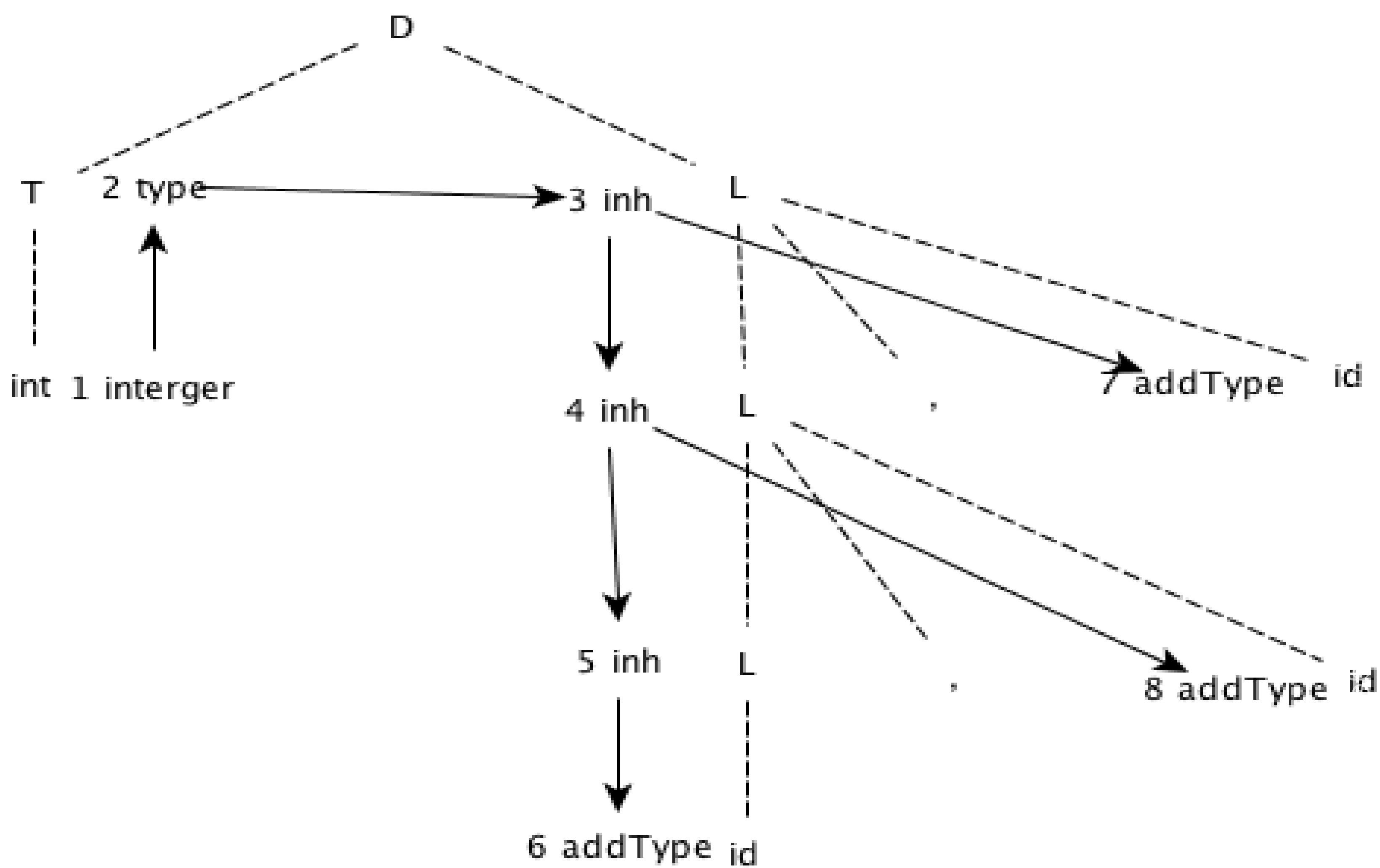
| PRODUCTION | SEMANTIC RULE |
|---|---|
| $D \rightarrow TL$ | $L.in := T.type$ |
| $T \rightarrow$ int | $T.type := integer$ |
| $T \rightarrow$ real | $T.type := real$ |
| $L \rightarrow L_1$, id | $L_1.in := L.in$; $addtype($id$.entry, L.in)$ |
| $L \rightarrow$ id | $addtype($id$.entry, L.in)$ |

# Annotated parse-tree for the input real id1, id2, id3






Akshaya Arunan, Govt. Engg College, Trivandrum


D
T
2 type
3 inh
L
int
1 interger
4 inh
L
,
7 addType
id
5 inh
L
,
8 addType
id
6 addType
id

# SDD for expression grammar with inherited attributes

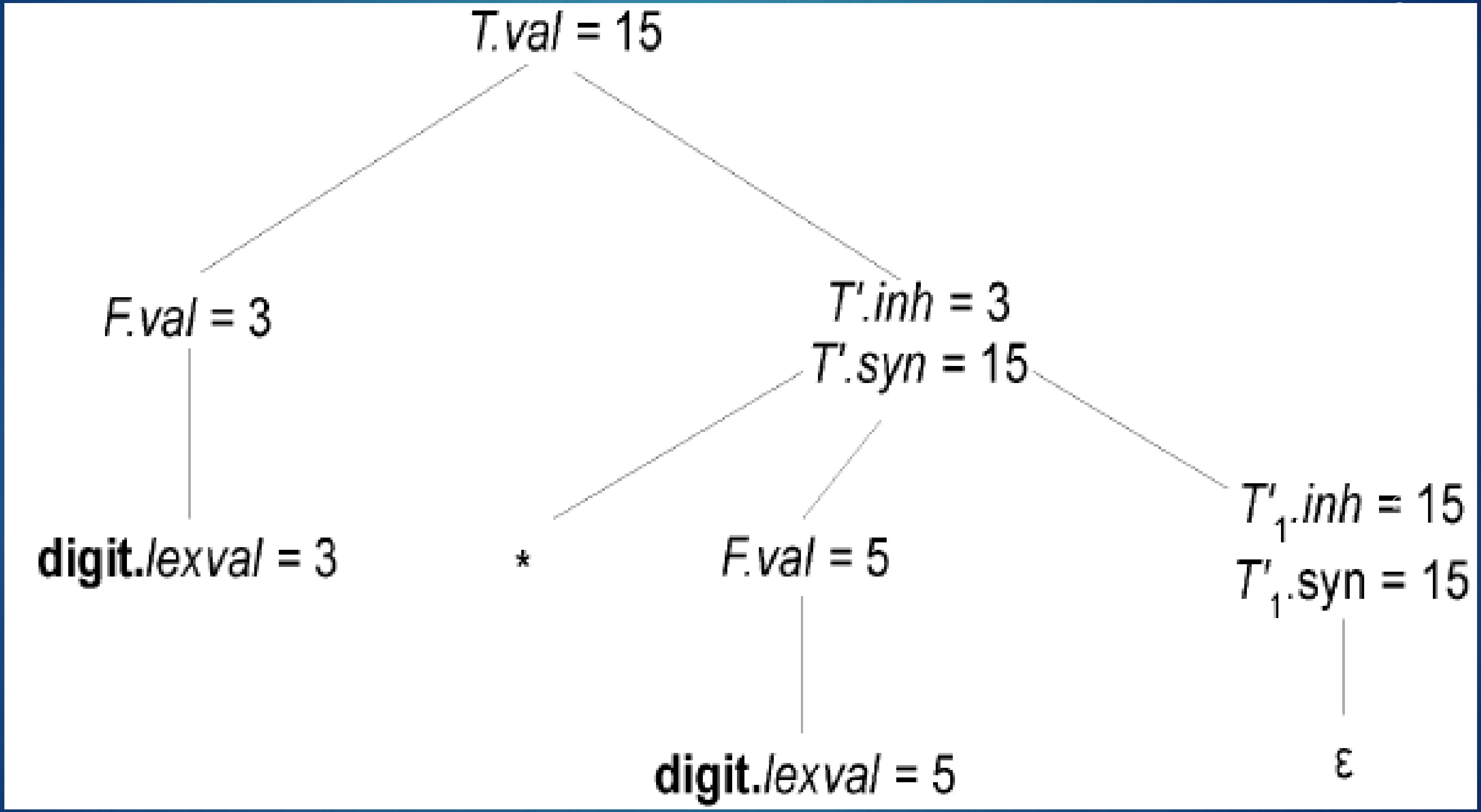




| Production | Semantic Rules |
|---|---|
| $T \rightarrow F\ T'$ | $T'.inh = F.val$<br>$T.val = T'.syn$ |
| $T' \rightarrow {}^*F\ T'_1$ | $T'_1.inh = T'.inh \times F.val$<br>$T'.syn = T'_1.syn$ |
| $T' \rightarrow \varepsilon$ | $T'.syn = T'.inh$ |
| $F \rightarrow \textbf{digit}$ | $F.val = \textbf{digit}.lexval$ |

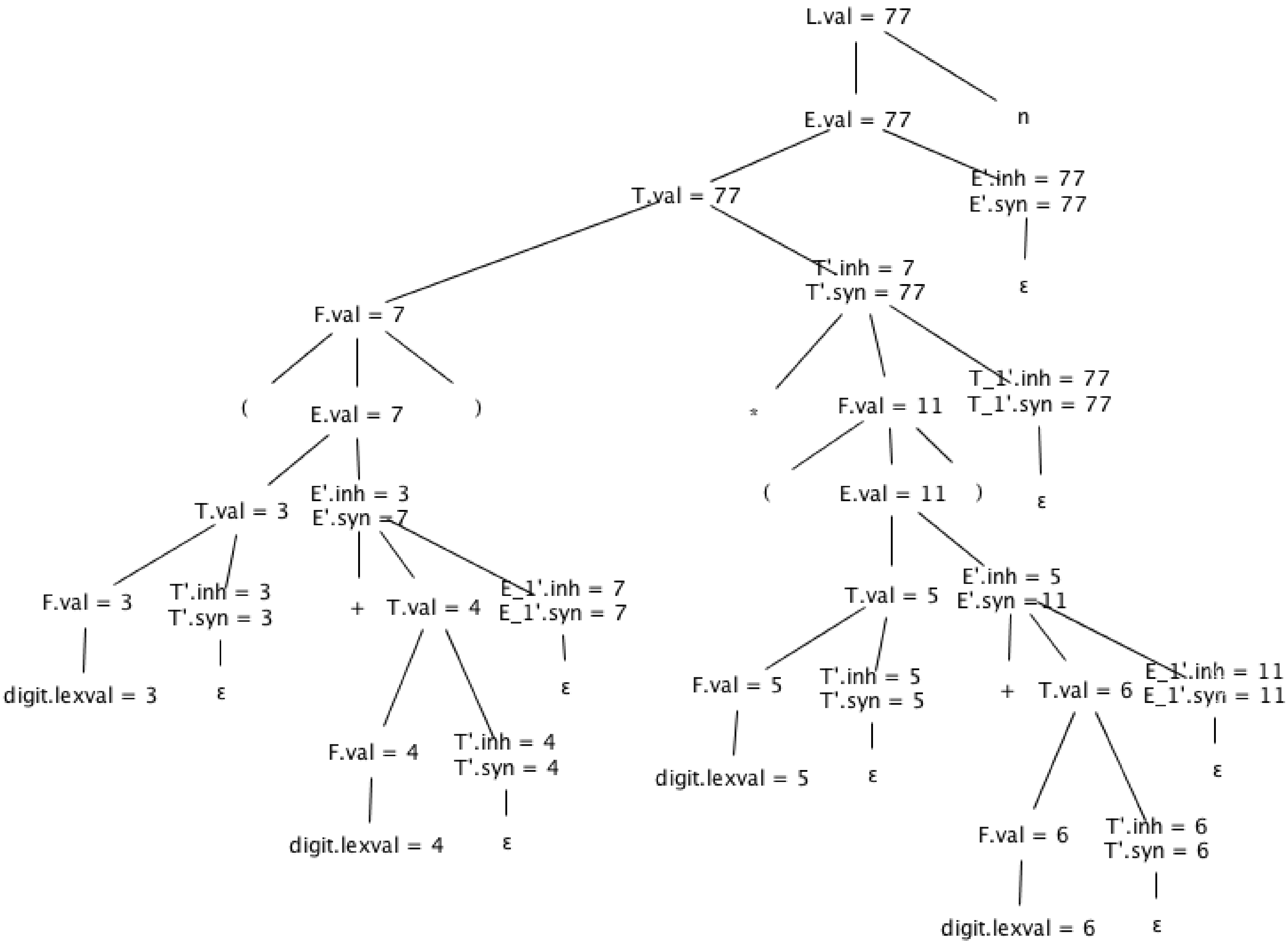
# Annotated Parse Tree for 3*5



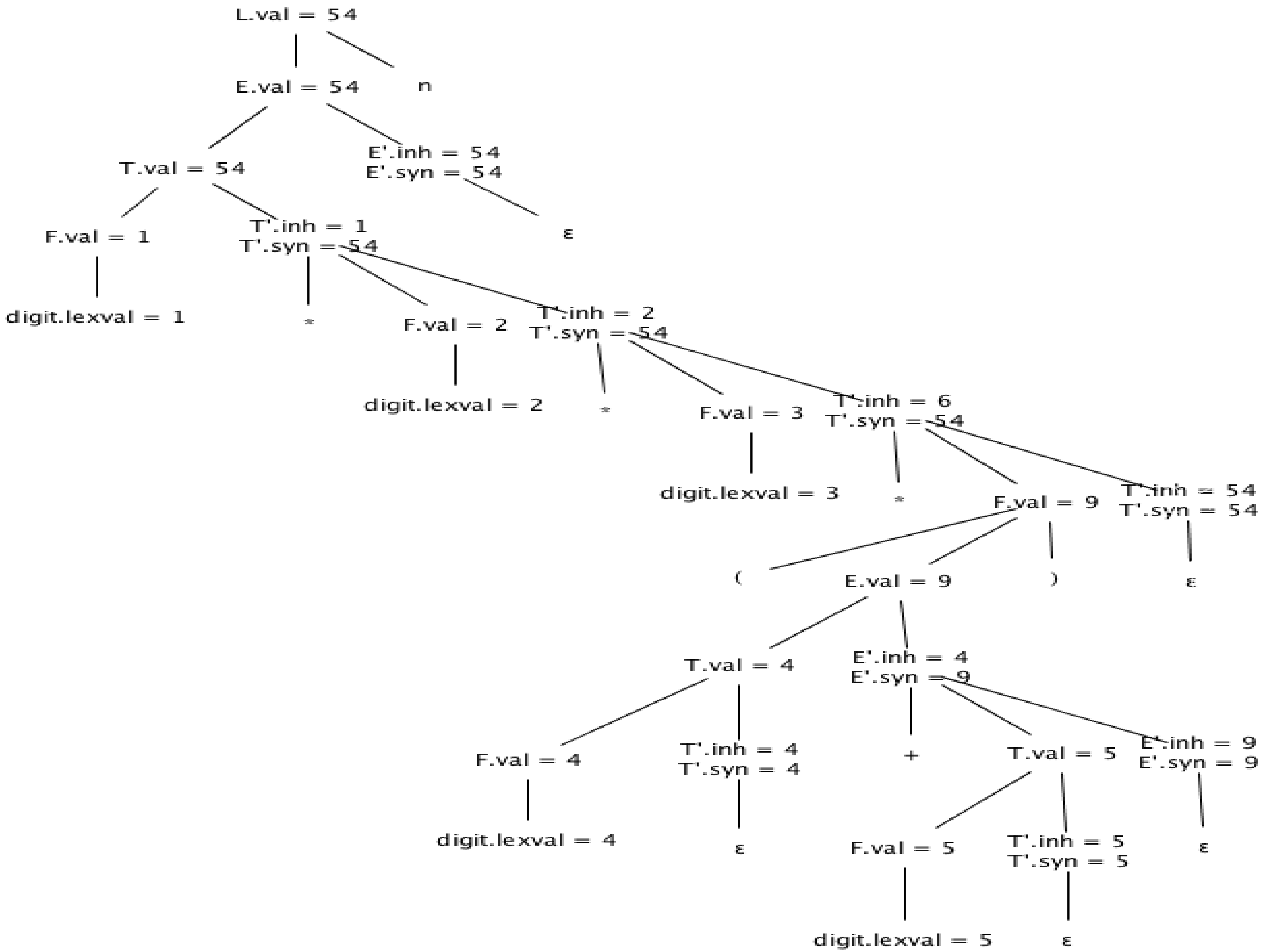


| | | |
|---|---|---|
| 1) | L -> En | L.val = E.val |
| 2) | E -> TE' | E'.inh = T.val<br>E.val = E'.syn |
| 3) | E' -> +TE_1' | E_1'.inh = E'.inh + T.val<br>E'.syn = E_1'.syn |
| 4) | E' -> ε | E'.syn = E'.inh |
| 5) | T -> FT' | T'.inh = F.val<br>T.val = T'.syn |
| 6) | T' -> *FT_1' | T_1'.inh = T'.inh * F.val<br>T'.syn = T_1'.syn |
| 7) | T' -> ε | T'.syn = T'.inh |
| 8) | F -> (E) | F.val = E.val |
| 9) | F -> digit | F.val = digit.lexval |

Akshaya Arunan, Govt. Engg College, Trivandrum



L.val = 77
E.val = 77
n
T.val = 77
E'.inh = 77
E'.syn = 77
ε
F.val = 7
T'.inh = 7
T'.syn = 77
(
E.val = 7
)
*
F.val = 11
T_1'.inh = 77
T_1'.syn = 77
T.val = 3
E'.inh = 3
E'.syn = 7
(
E.val = 11
)
ε
F.val = 3
T'.inh = 3
T'.syn = 3
+
T.val = 4
E_1'.inh = 7
E_1'.syn = 7
T.val = 5
E'.inh = 5
E'.syn = 11
digit.lexval = 3
ε
ε
F.val = 5
T'.inh = 5
T'.syn = 5
+
T.val = 6
E_1'.inh = 11
E_1'.syn = 11
F.val = 4
T'.inh = 4
T'.syn = 4
digit.lexval = 5
ε
ε
digit.lexval = 4
ε
F.val = 6
T'.inh = 6
T'.syn = 6
digit.lexval = 6
ε



L.val = 54
E.val = 54
n
T.val = 54
E'.inh = 54
E'.syn = 54
F.val = 1
T'.inh = 1
T'.syn = 54
ε
digit.lexval = 1
*
F.val = 2
T'.inh = 2
T'.syn = 54
digit.lexval = 2
*
F.val = 3
T'.inh = 6
T'.syn = 54
digit.lexval = 3
*
F.val = 9
T'.inh = 54
T'.syn = 54
(
E.val = 9
)
ε
T.val = 4
E'.inh = 4
E'.syn = 9
F.val = 4
T'.inh = 4
T'.syn = 4
+
T.val = 5
E'.inh = 9
E'.syn = 9
digit.lexval = 4
ε
F.val = 5
T'.inh = 5
T'.syn = 5
ε
digit.lexval = 5
ε

- SDD with circular dependencies no guarantee in the order of evaluation.

e.g.

| Production | Semantic Rules |
|---|---|
| $A \rightarrow B$ | $A.s = B.i$ |
| | $B.i = A.s + 1$ |

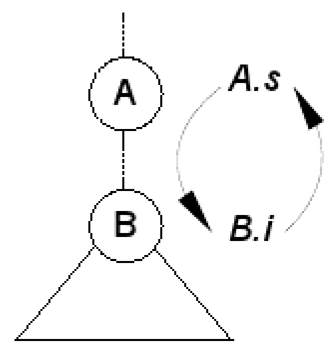



Fig 5.2: The circular dependency of *A.s* and *B.i* on one another

Akshaya Arunan, Govt. Engg College, Trivandrum

# Evaluation Orders for SDD's



□**"Dependency graphs" tool for determining an evaluation order for the** attribute instances in a given parse tree.

□annotated parse tree shows the values of attributes, a dependency graph helps to determine how those values can be computed.

Akshaya Arunan, Govt. Engg College, Trivandrum

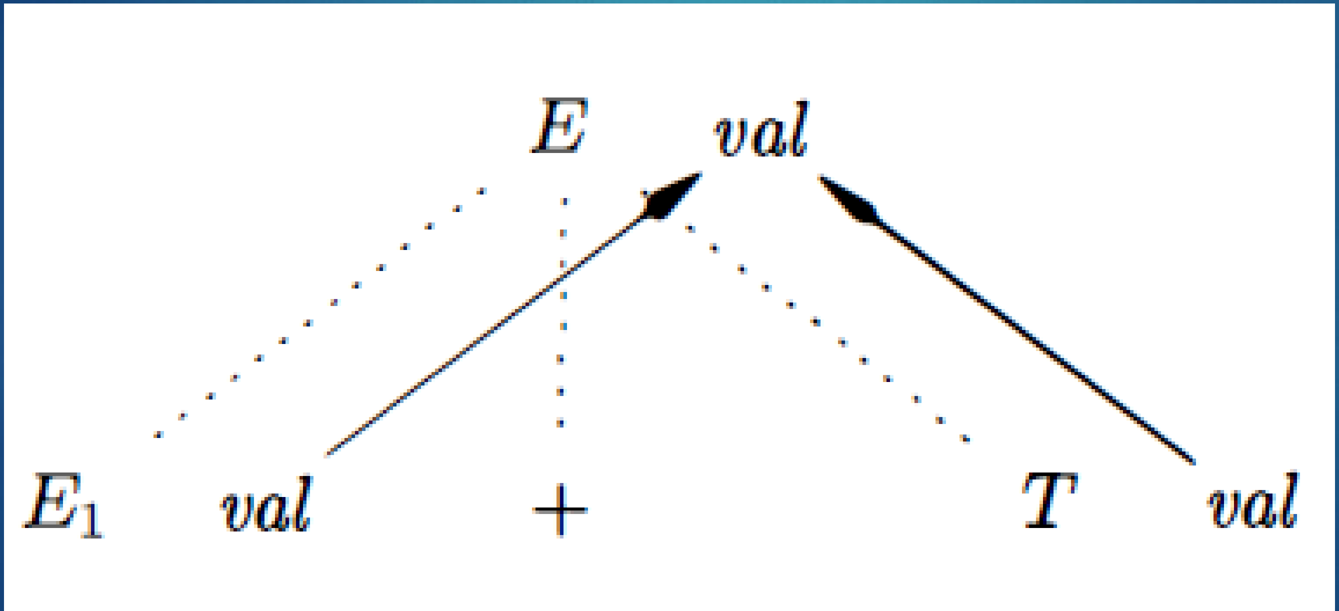
# Dependency graphs



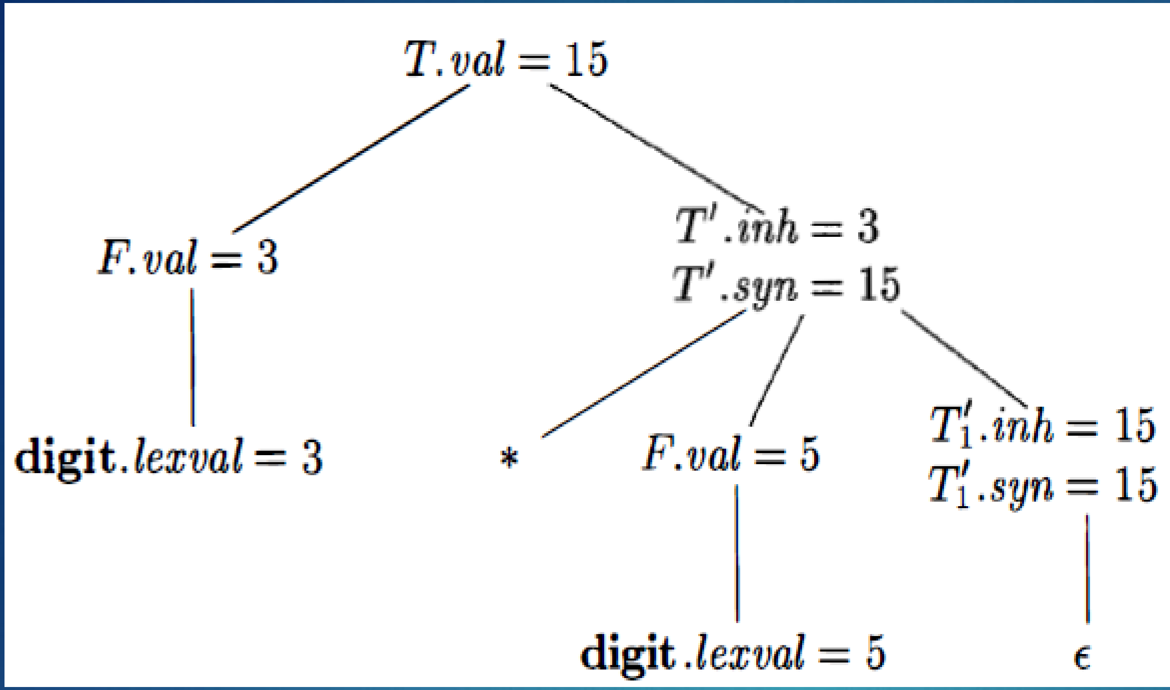
- Edges express constraints implied by the semantic rules.
- Each attribute is associated to a node
- If a semantic rule associated with a production *p defines the value of synthesized* attribute *A.b in terms of the value of X.c, then graph has an edge from X.c to A.b*
- If a semantic rule associated with a production *p defines the value of inherited* attribute *B.c in terms of value of X.a, then graph has an edge from X.a to B.c*

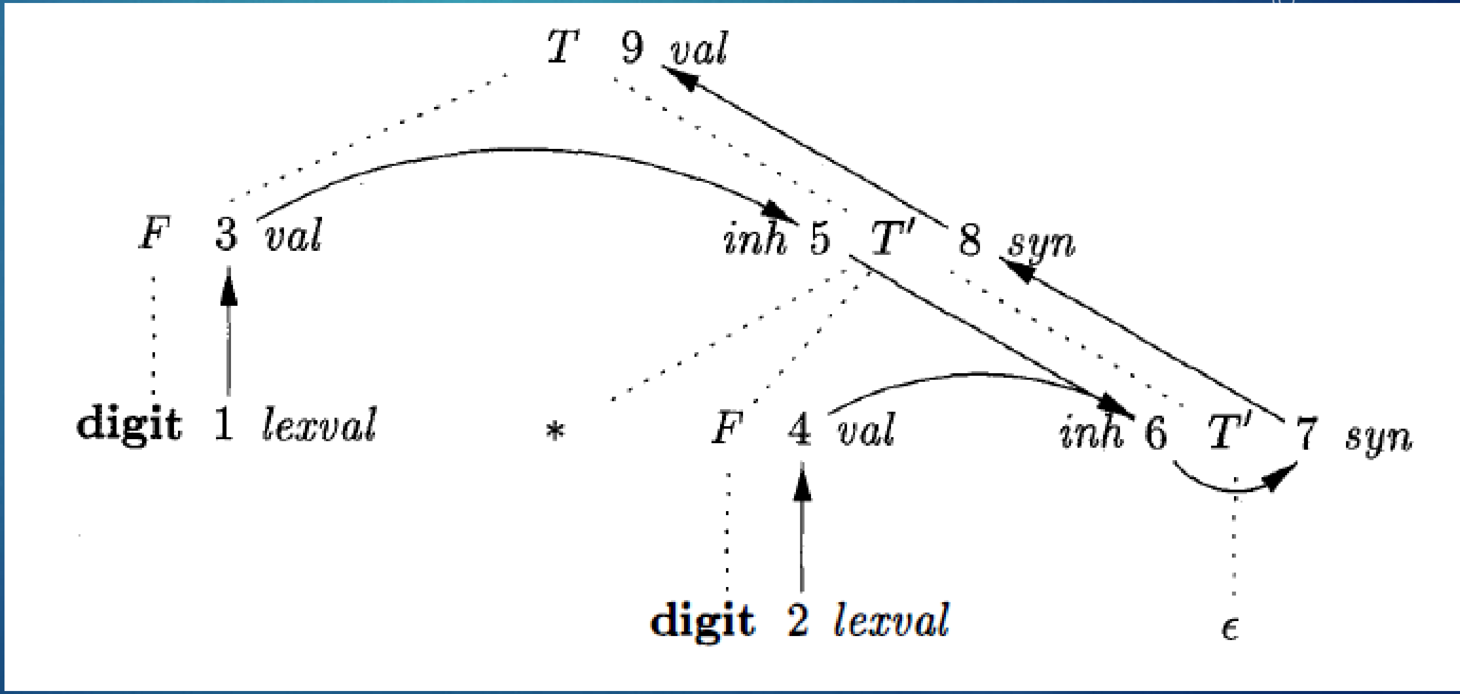
Akshaya Arunan, Govt. Engg College, Trivandrum

| PRODUCTION | SEMANTIC RULE |
|---|---|
| $E \to E_1 + T$ | $E.val = E_1.val + T.val$ |




Akshaya Arunan, Govt. Engg College, Trivandrum




Akshaya Arunan, Govt. Engg

# Topological Sort



- A dependency graph characterizes the possible order in which we can evaluate the attributes at various nodes of a parse tree.
- If there is an edge from node M to N, then attribute corresponding to M first be evaluated before evaluating N.
- Thus the allowable orders of evaluation are N1, N2, ..., Nk such that if there is an edge from Ni to Nj then i < j.
- Such an ordering embeds a directed graph into a linear order, and is called a ***topological sort*** *of the graph.*
- If there is any cycle in the graph, then there are no topological sorts.

# S-Attributed



- If every attribute is synthesized.
- S-attributed SDD can be evaluated in bottom up order of the nodes of the parse tree.

Akshaya Arunan, Govt. Engg College, Trivandrum

# L-attributed definitions



- Synthesized, or
- Inherited, but with the rules limited as follows . Suppose that there is a production A -> XIX2 · . . Xn , and that there is an inherited attribute Xi·a computed by a rule associated with this production. Then the rule may use only:

Akshaya Arunan, Govt. Engg College, Trivandrum

# L-attributed definitions



- Inherited attributes associated with the head A.
- Either inherited or synthesized attributes associated with the occurrences of symbols Xl , X2 , ••• ,Xi-l located to the left of Xi.
- Inherited or synthesized attributes associated with this occurrence of Xi itself, but only in such a way that there are no cycles in a dependency graph formed by the attributes of this Xi .

# Semantic Rules with Controlled Side Effects



- Permit incidental side effects that do not disturb attribute evaluation.
- Impose restriction on allowable evaluation orders, so that the same translation is produced for any allowable order.

Akshaya Arunan, Govt. Engg College, Trivandrum

# Applications of Syntax-Directed Translations



- Construction of Syntax Trees
- Syntax trees are useful for representing programming language constructs like expressions and statements.
- Each node of a syntax tree represents a construct; the children of the node represent the meaningful components of the construct.
  - e.g. a syntax-tree node representing an expression E1 + E2 has label + and two children representing the sub expressions E1 and E2

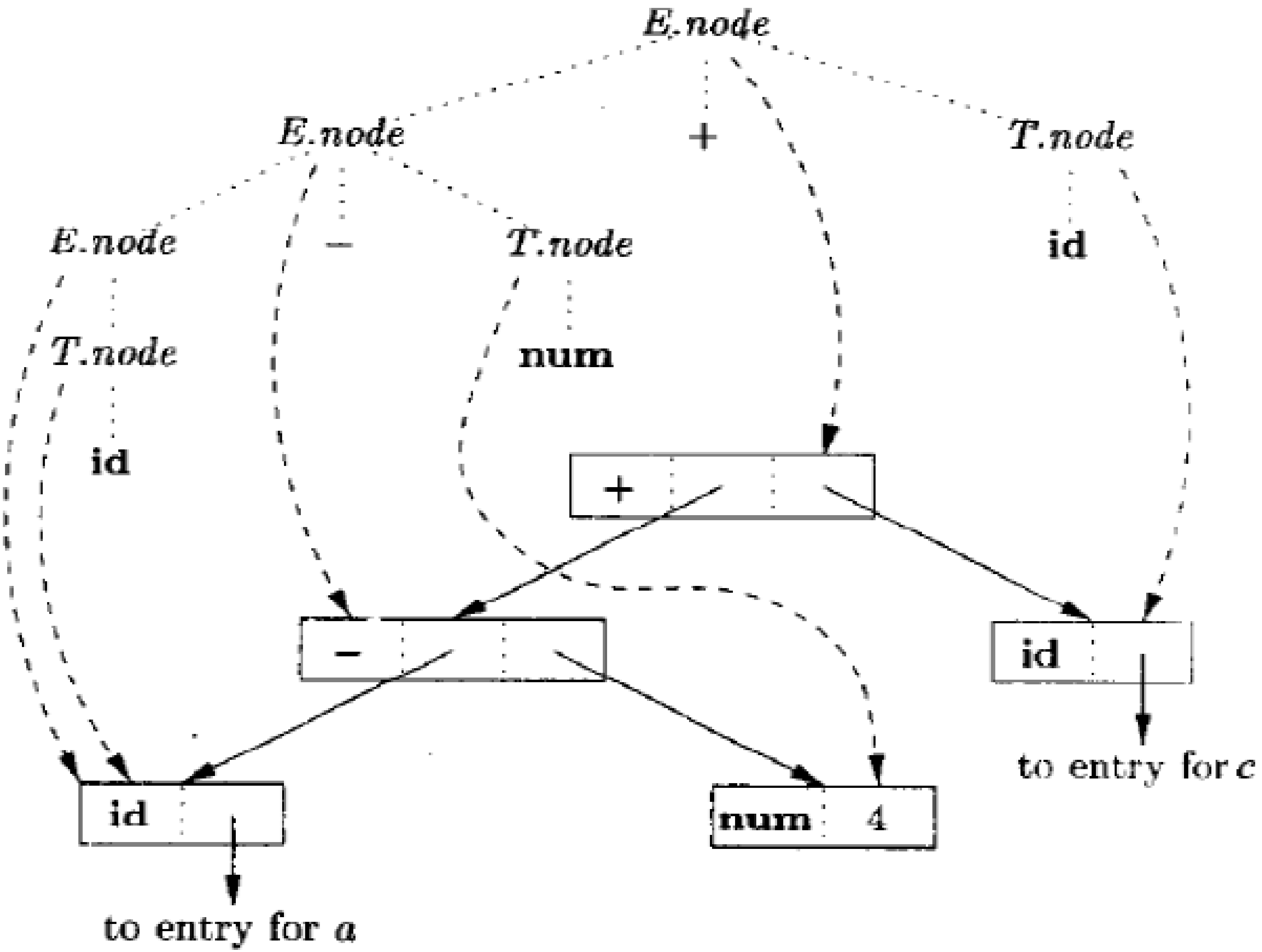
Akshaya Arunan, Govt. Engg College, Trivandrum

# Applications of Syntax-Directed Translations

- Each node is implemented by objects with suitable number of fields; each object will have an op field that is the label of the node with additional fields as follows:
  - If the node is a leaf, an additional field holds the lexical value for the leaf . This is created by function Leaf(op, val)
  - If the node is an interior node, there are as many fields as the node has children in the syntax tree. This is created by function Node(op, c1, 2,...,ck) .

Akshaya Arunan, Govt. Engg College, Trivandrum

| Production | Semantic Rules |
| --- | --- |
| 1) $E \rightarrow E_1 + T$ | $E.node$ = **new** $Node$ ( '+', $E_1.node$, $T.node$ ) |
| 2) $E \rightarrow E_1 - T$ | $E.node$ = **new** $Node$ ( '−', $E_1.node$, $T.node$ ) |
| 3) $E \rightarrow T$ | $E.node = T.node$ |
| 4) $T \rightarrow ( E )$ | $E.node = T.node$ |
| 5) $T \rightarrow$ **id** | $T.node$ = **new** $Leaf$ ( **id**, **id**.$entry$ ) |
| 6) $T \rightarrow$ **num** | $T.node$ = **new** $Leaf$ ( **num**, **num**.$val$ ) |



E.node
E.node
+
T.node
E.node
−
T.node
id
T.node
num
id
+
−
id
id
num 4
to entry for c
to entry for a

# Syntax Directed Translation Schemes



- SDT can be implemented by first building a parse tree and then performing the actions in a left-to-right depth-first order
- Postfix Translation Schemes:each semantic action can be placed at the end of production and executed along with the reduction of body to the head of the production.

Akshaya Arunan, Govt. Engg College, Trivandrum

| Production | Semantic Rules |
| --- | --- |
| $L \rightarrow E\ \mathbf{n}$ | { print ( $E.val$ ); } |
| $E \rightarrow E_1 + T$ | { $E.val = E_1.val + T.val$; } |
| $E \rightarrow T$ | { $E.val = T.val$; } |
| $T \rightarrow T_1 * F$ | { $T.val = T_1.val \times F.val$; } |
| $T \rightarrow F$ | { $T.val = F.val$; } |
| $F \rightarrow ( E )$ | { $F.val = E.val$; } |
| $F \rightarrow \mathbf{digit}$ | { $F.val = \mathbf{digit}.lexval$; } |

Akshaya Arunan, Govt. Engg College, Trivandrum

# Parser-Stack Implementation of Postfix SDTs

| | | | | |
|---|---|---|---|---|
| | $X$ | $Y$ | $Z$ | state/grammar symbol |
| | $X.a$ | $Y.b$ | $Z.c$ | synthesized attribute |

top

# Semantic Actions during Parsing



- when shifting
  - push the value of the terminal on the semantic stack
- when reducing
  - pop k values from the semantic stack, where k is the number of symbols on production's RHS
  - push the production's value on the semantic stack

# SDTs with Actions inside Productions

- Action can be placed at any position in the production body.
- Action is performed immediately after all symbols left to it are processed.
- Given B —> X { a } Y , an action a is done after
  - we have recognized X (if X is a terminal), or
  - all terminals derived from X (if X is a nonterminal).

Akshaya Arunan, Govt. Engg College, Trivandrum

# SDTs with Actions inside Productions



Akshaya Arunan, Govt. Engg College, Trivandrum

- If bottom-up parser is used, then action a is performed as soon as X appears on top of the stack.
- If top-down parser is used, then action a is performed
  - just before Y is expanded (if Y is nonterminal), or
  - check Y on input (if Y is a terminal).
- Any SDT can be implemented as follows:
  - Ignoring actions, parse input and produce parse tree.
  - Add additional children to node N for action in α, where A → α.
  - Perform preorder traversal of the tree, and as soon as a node labeled by an action is visited, perform that action.

# Semantic Actions during Parsing



| Production | Actions |
|---|---|
| $L \rightarrow E$ **n** | { print ( *stack* [*top* -1].*val* ); *top* = *top* - 1 ;} |
| $E \rightarrow E_1 + T$ | { *stack* [*top* - 2].*val* = *stack*[*top* - 2].*val* + *stack*[*top*].*val*; *top* = *top* - 2; } |
| $E \rightarrow T$ | |
| $T \rightarrow T_1 * F$ | { *stack* [*top* - 2].*val* = *stack*[*top* - 2].*val* x *stack*[*top*].*val*; *top* = *top* - 2; } |
| $T \rightarrow F$ | |
| $F \rightarrow ( E )$ | { *stack* [*top* - 2].*val* = *stack*[*top* - 1].*val*; *top* = *top* - 1; } |
| $F \rightarrow$ **digit** | |

# Type checking

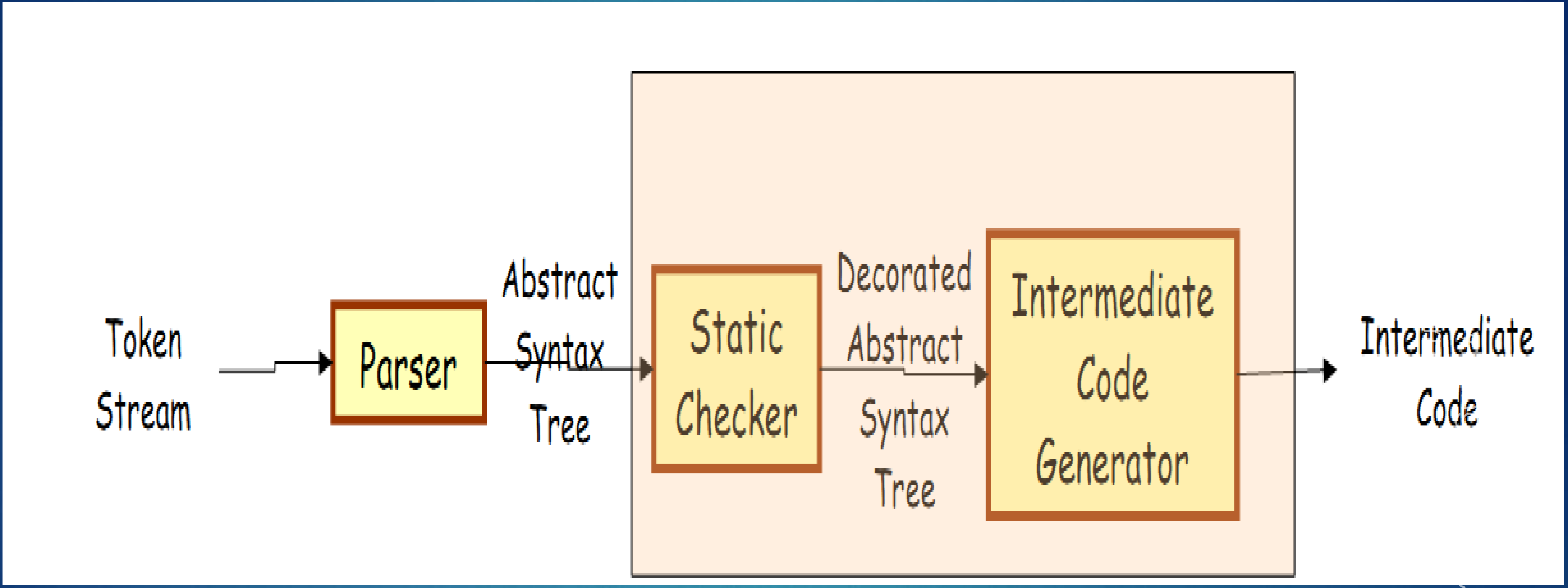
Akshaya Arunan, Govt. Engg College, Trivandrum

# Type Checking?



- Type checking is the process of verifying that each operation executed in a program respects the type system of the language.
- This generally means that all operands in any

  expression are of appropriate types and number.
- Much of what we do in the semantic analysis phase is type checking.


Token
Stream
Parser
Abstract
Syntax
Tree
Static
Checker
Decorated
Abstract
Syntax
Tree
Intermediate
Code
Generator
Intermediate
Code

# Static checks





- *Type checks.* A compiler should report an error if an operator is applied to an incompatible operand.
- *Flow-of-control checks.* Statements that cause flow of control to leave a construct must have some place to which to transfer flow of control. For example, branching to non-existent labels.
- *Uniqueness checks.* Objects should be defined only once. This is true in many languages.
- *Name-related checks.* Sometimes, the same name must appear two or more times.

# type system



- The compiler must then determine that these type expressions conform to a collection of logic al rules that is called the type system for the source language

Akshaya Arunan, Govt. Engg College, Trivandrum

# Type Expression

Informally, a type expression is either a basic type or is formed by applying an operator called a *type constructor* to other type expressions.

Akshaya Arun
College, Trivandrum

# Type Expression



1. A basic type is a type expression. A special basic type, *type_error*, will signal an error during type checking. Finally, a basic type *void* denoting the absence of a value allows statements to be checked.

2. Since type expressions may be named, a type name is a type expression.

3. A type constructor applied to type expressions is a type expression. Constructors include:

   (a) *Arrays.* 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 $I$.

   (b) *Products.* If $T_1$ and $T_2$ are type expressions, then their Cartesian product $T_1 \times T_2$ is a type expression.

# Type Expression



(c) *Records.* The type of a record is in a sense the product of the types of its fields. The difference between a record and a product is that the fields of a record have names. Type checking of records can be done using the type expression formed by applying the constructor *record* to a tuple formed from field names and their associated types.

(d) *Pointers.* If $T$ is a type expression, then $pointer(T)$ is a type expression denoting the type pointer to an object of type $T$.

(e) *Functions.* Functions take values in some domain and map them into value in some range. This is denoted *domain values* $\rightarrow$ *range values.*

4. Type expressions may contain variables whose values are type expressions.

# Type System



- A *type system* is a collection of rules for assigning type expressions to the various parts of a program.
- A type checker implements a type system.

# Static and Dynamic checking



- Checking done by the compiler is static, while if it done at run–time, it is dynamic.
- A *sound* type system eliminates the need for dynamic checking for type errors because it allows us to determine statically that these errors cannot occur when the target program runs.
- A language is *strongly typed* if its compiler can guarantee that the programs it accepts will execute without type errors.

# Error Recovery



Akshaya Arun

- It is important for a type checker to do something reasonable when an error is discovered.
- At the very least, the compiler must report the nature and location of the error.
- It is desirable for the type checker to recover from errors, so it can check the rest of the input.

# Coercions





- Conversion from one type to another is said to be *implicit* if it is to be done automatically by the compiler.

- Implicit type conversions are also called *coercions*.

- Conversion is said to be *explicit* if the programmer must write something to cause the conversion.