SYNTAX-DIRECTED TRANSLATION

Based on Chapter 5 of Aho, Lam, Sethi, Ullman:

Compilers: Principles, Techniques, & Tools

2nd Ed, Addison Wesley, 2007

Table of Contents

Introduction

- Syntax-directed translation
  - CFG guides the translation of programs by attaching rules or program fragments to productions in a grammar

  - Example : $E \rightarrow E_1 + T$
    1. Translate E1
    2. Translate T
    3. Handle +

- Two notations
  - Syntax-directed Definitions
  - Syntax-directed Translation Schemes

Introduction

- Syntax-directed definitions
  - A notation for specifying translation for PL constructs in terms of attributes and semantics rules
  - To associate information with a programming language construct by attaching attributes to the grammar symbols
  - Values for attributes are computed by semantic rules or associated with the grammar productions

  - Example : infix-to-postfix translator

Production	Semantic Rule
$E \rightarrow E_1 + T$	$E.code = E_1.code \;\|\|\; T.code \;\|\|\; '+'$

  - High-level specification
  - Hiding implementation details

Introduction

- Translation schemes
  - A notation for specifying a translation by attaching program fragments (semantic actions) to productions in a grammar
  - The order of evaluation of semantics rules is explicitly specified

  - Example : infix-to-postfix translator

    <u>Production Rule with Semantic Action</u>

    $E \rightarrow E_1 + T$ { print '+' }

  - Exposing implementation details

Introduction

- Conceptual view of syntax-directed translation

$$\text{input string} \longrightarrow \text{parse tree} \longrightarrow \text{dependency graph} \longrightarrow \text{evaluation order for semantic rules}$$

  - syntax-directed translations can be implemented without explicitly constructing a parse tree or a dependency graph

    e.g. L-attributed definitions

Syntax-Directed Definitions

- A syntax-directed definition is a generalization of a CFG using
  - attributes (attached to grammar symbols)
  - semantic rules (attached to production rules)
- Attributes
  - represents anything helpful for translation such as type, string, memory location, syntax tree, or whatever.
  - the value is defined by semantic rules
  - synthesized attributes : depends on the attributes of the child nodes in the parse tree
  - inherited attributes : depends on the attributes of parent, itself, and siblings in the parse tree
  - Terminals can have synthesized attributes, not inherited attributes
- Semantic rules
  - set up dependencies between attributes (dependency graph)
  - dependency graph derives the evaluation order of semantic rules
  - evaluation of semantic rules defines the values of the attributes
- Annotated parse tree
  - a parse tree to show the values of attributes at each node

Syntax-Directed Definitions

- Example 5.1 : desk calculator

Production	**Semantic Rules**
L -> E n	L.val = 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**.lexval

  - each of L, E, T, F has a synthesized attribute *val*
  - synthesized attribute *lexval* of **digit** is supplied by the lexical analyzer

Syntax-Directed Definitions

- S-attributed definition
  - Syntax-directed definition that uses only synthesized attributes
  - Example 5.1 is an S-attributed definition
  - An S-attributed SDD can be implemented naturally in conjunction with an LR parser
    - The SDD in Example 5.1 mirrors the Yacc program below, which illustrate translation during parsing

```
line : expr '\n'          { printf("%d\n", $1); }
     ;
expr : expr '+' term   { $$ = $1 + $3; }
     | term
     ;
term : term '*' factor  { $$ = $1 * $3; }
     | factor
     ;
factor : '(' expr ')'     { $$ = $2; }
     | DIGIT
     ;
```

Syntax-Directed Definitions

- Evaluating an SDD
  - In what order do we evaluate attributes
    - using dependency of attributes
  - S-attributed SDD
    - in bottom-up order
  - SDD with both synthesized and inherited attributes
    - more difficult
    - circular dependency – impossible to evaluate

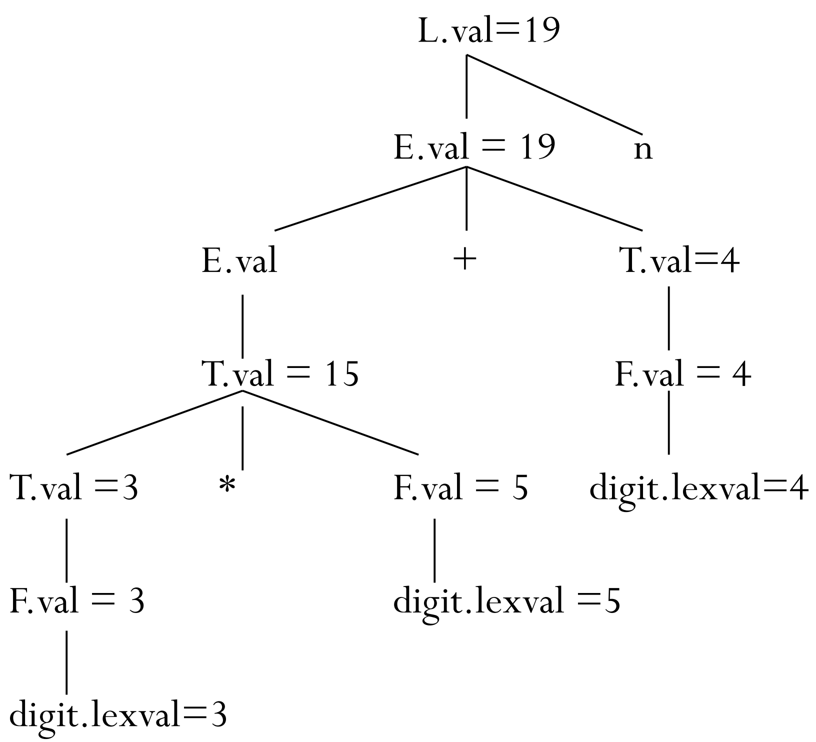
Production	Semantic Rules
A -> B	A.s = B.i
	B.i = A.s + 1

    - fortunately, there are useful subclasses of SDD's

  - Annotated parser tree is used to visualize the translation specified by SDD

Syntax-Directed Definitions

- Annotated parser tree (Example 5.1)



Inherited Attributes

- Inherited attributes
  - are defined in terms on attributes in the parent and/or siblings
  - useful for expressing the context information
  - e.g.
    - type information in a declaration
    - whether a variable appears on left or right side

Inherited Attributes

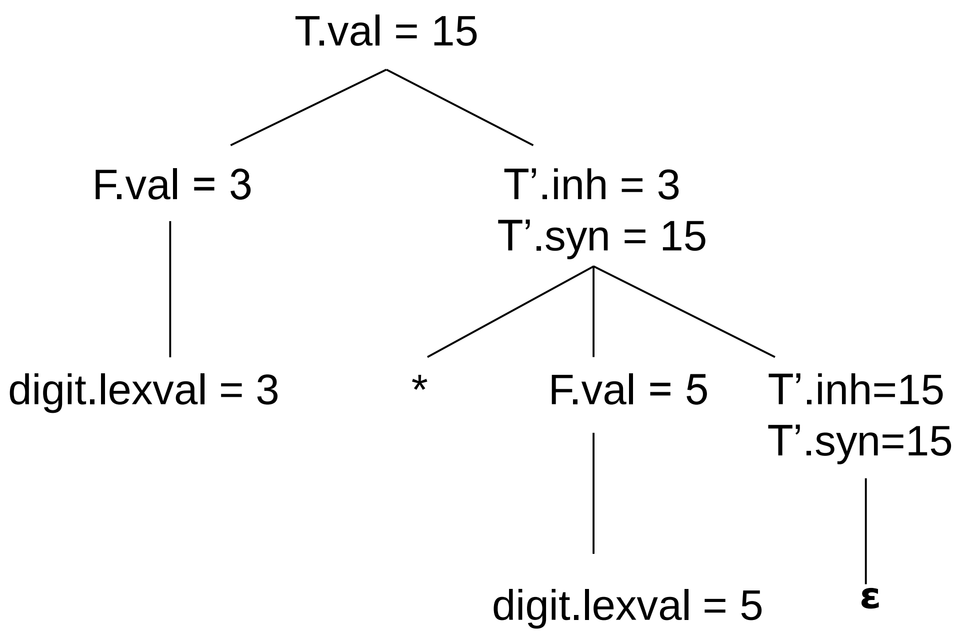
- Example 5.3 : Grammar for Expression (part)

Production	**Semantic Rules**
T -> F T’	T’.inh = F.val
	T.val = T’.syn
$T' \to * \; F \; T'_1$	T'_1.inh = T’.inh * F.val
	T’.syn = T'_1.syn
$T' \to \varepsilon$	T’.syn = T’.inh
F-> digit	F.val = digit.lexval

- **inh** is an inherited attribute of T’
- **val** and **syn** are synthesized attributes
- synthesized attribute *lexval* of **digit** is supplied by the lexical analyzer

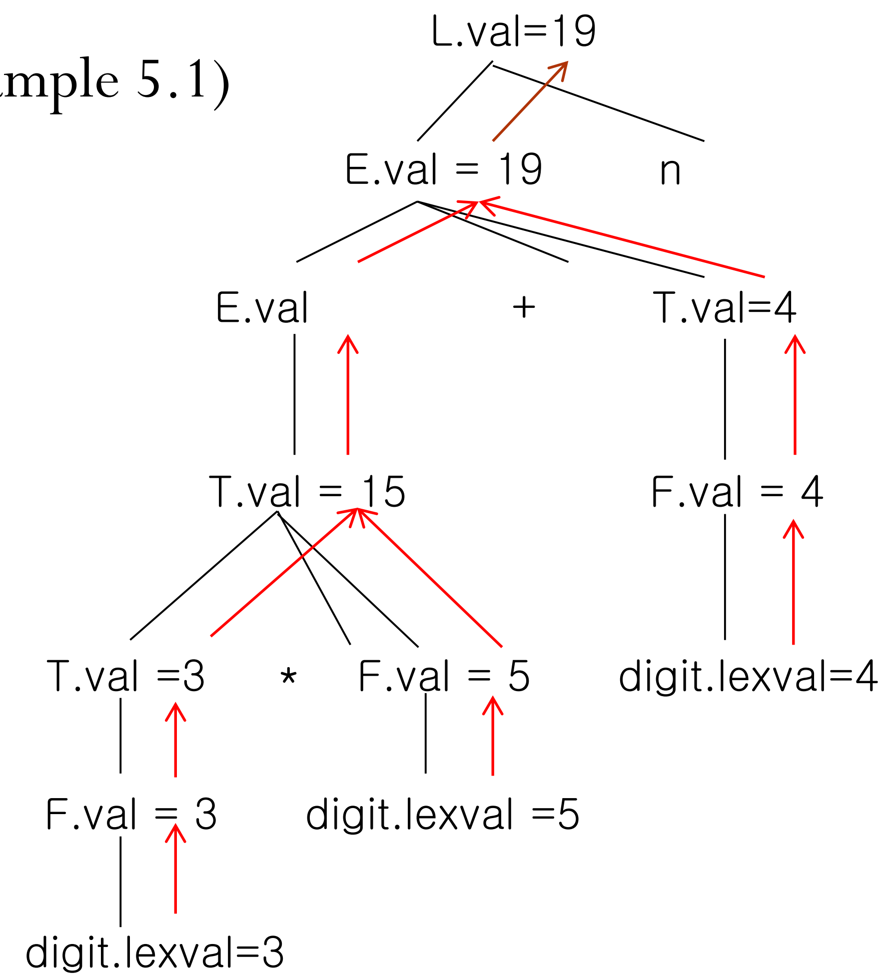
Inherited Attributes

- Annotated parser tree for 3*5 (Example 5.3)



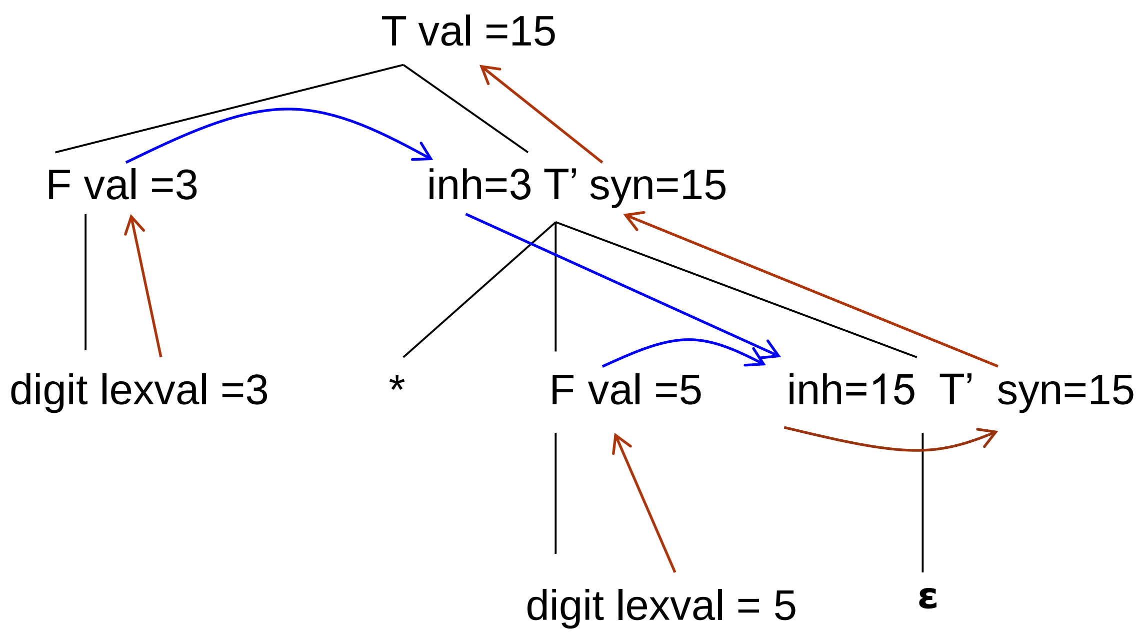
Dependency Graph

- Dependency graph and evaluation order
  - topological sort
- S-attributed SDD(Example 5.1)
  - post-order traversal



Dependency Graph

- SDD with Inherited Attributes (Example 5.3)



L-Attributed Definitions

- A syntax-directed definition is L-attributed if it uses
  - synthesized attributes, or
  - inherited attributes of X_j ($1<=j<=n$) on $A\to X_1,X_2,\ldots,X_n$ depends only on
    1. the attributes of the symbols $X1,X2,\ldots,Xj-1$
    2. the inherited attributes associated with A
    3. the attributes associated with Xj itself (if no cycle)
- every S-attributed definition is an L-attributed definition
  - Example 5.1 (Revisited)

Production	**Semantic Rules**
L -> E n	L.val = 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**.lexval

L-Attributed Definitions

- Example 5.3 (Revisited)

Production	**Semantic Rules**
T -> F T’	T’.inh = F.val
	T.val = T’.syn
$T' \to * F T'_1$	T'_1.inh = T’.inh * F.val
	T’.syn = T'_1.syn
…	

- Example 5.9 : not an L-attributed definition

Production	**Semantic Rules**
A -> B C	A.s = B.b; B.i = f(C.c, A.s)

L-Attributed Definitions

- L-attributed definitions can evaluated in depth-first order in the parse tree

```
procedure dfvisit(n:node)
begin
    evaluate inherited attributes of  n
    for each child m of n , from left to right do begin
      dfvisit(m)
   end
   evaluate synthesized attributes of n
end
```

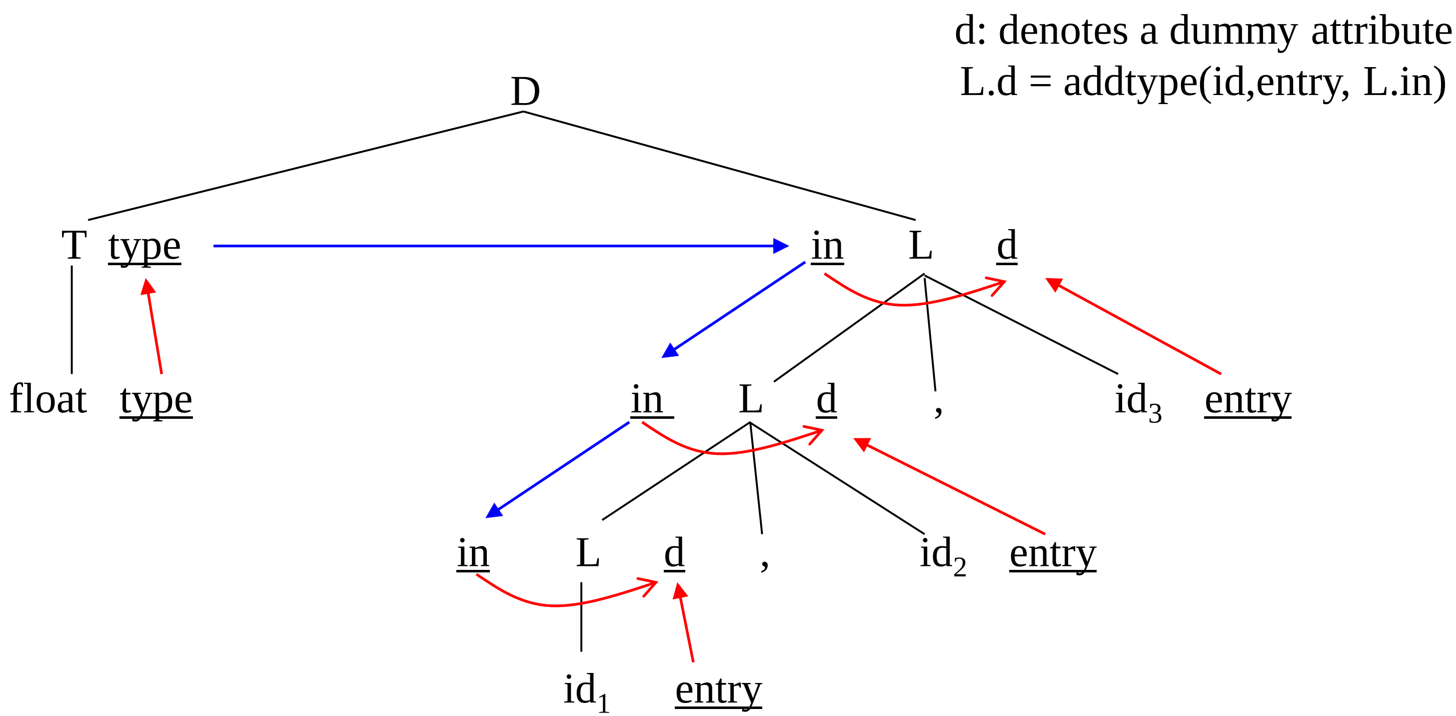
L-Attributed Definitions

- Example 5.10: Type declaration

Production	**Semantic Rules**
D-> T L	L.in = T.type
T-> int	T.type = integer
T-> float	T.type = real
L->L_1 , id	L_1.in = L.in; addtype(id.entry, L.in)
L-> id	addtype(id.entry, L.in)

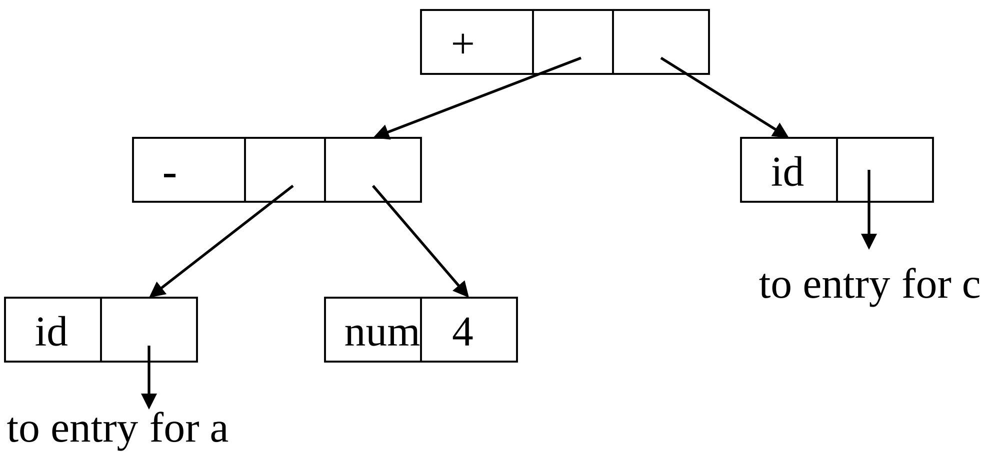
L-Attributed Definitions

- Dependency graph and evaluation order (**float id1, id2, id3**)



Application: Construction of Syntax Trees

- Syntax Tree : a – 4 + c



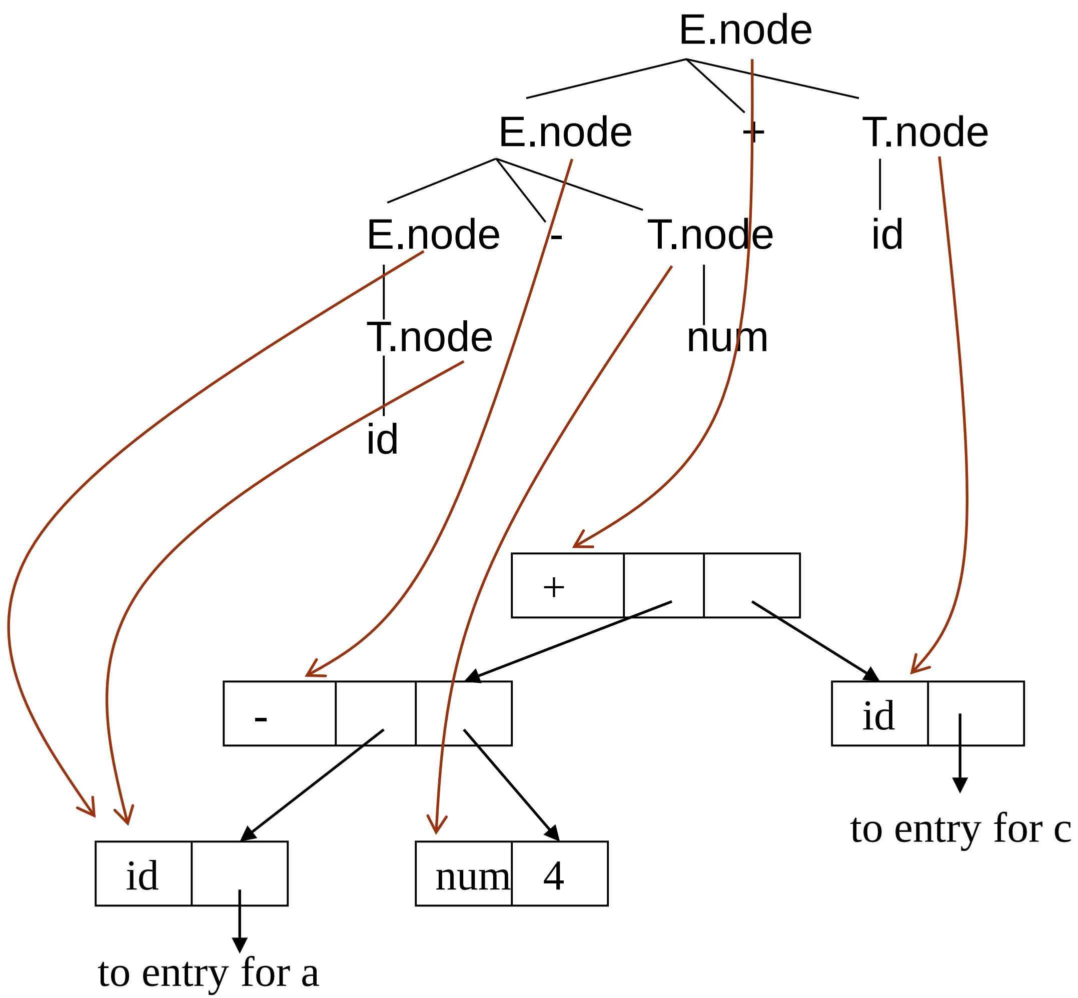
Construction of Syntax Trees

- Example 5.11 : S-attributed definition (Expression)

Production	**Semantic Rules**
$E \rightarrow E_1 + T$	E.node=mkNode('+', E_1.node, T.node)
$E \rightarrow E_1 - T$	E.node=mkNode('-', E_1.node, T.node)
E->T	E.node=T.node
T->(E)	T.node=E.node
T->id	T.node= mkLeaf(id, id.entry)
T-> num	T.node=mkLeaf(num, num.val)

Construction of Syntax Trees

- Annotated Parse Tree : a – 4 + c



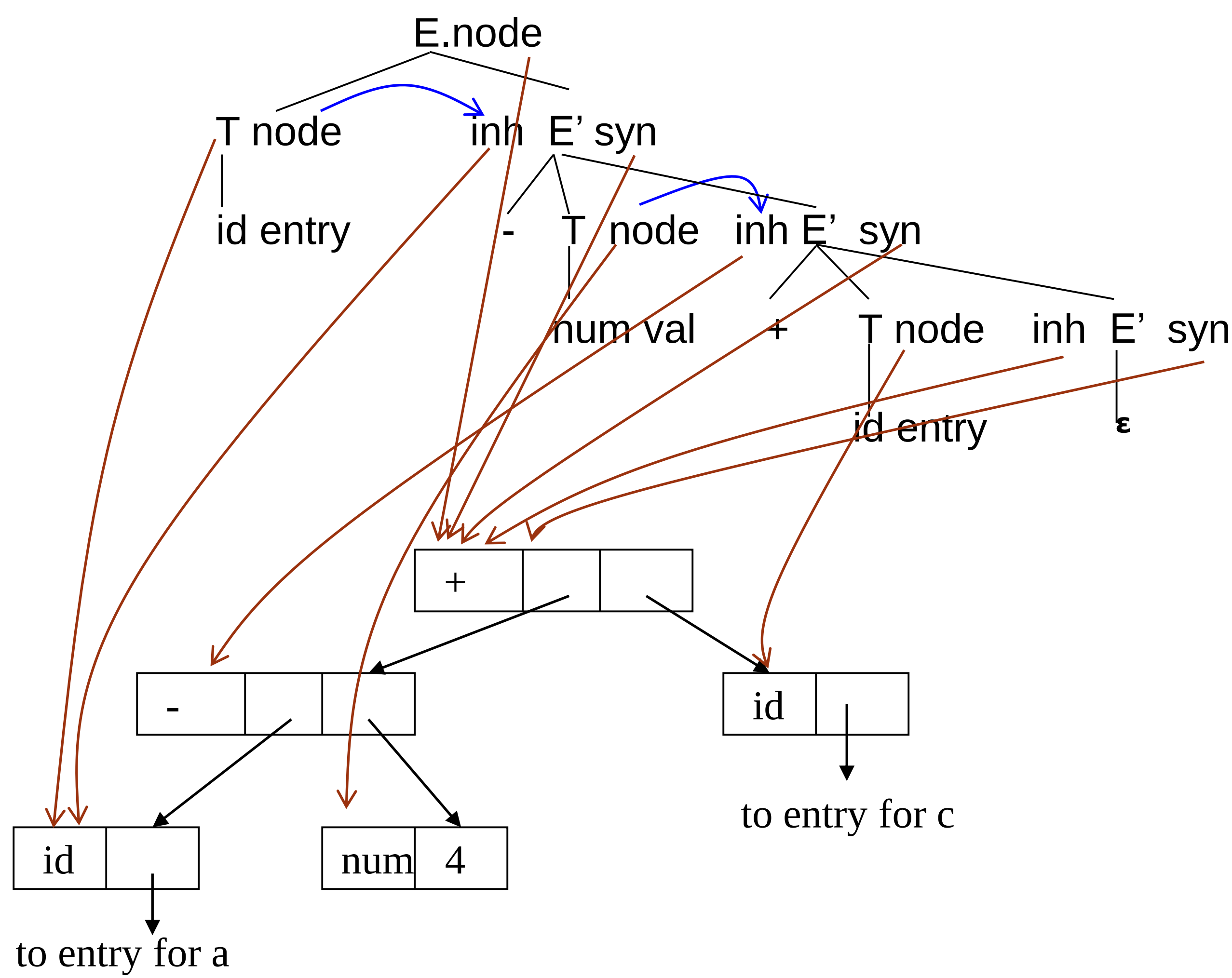
Construction of Syntax Trees

- Example 5.12 : L-attributed definition (Expression)

Production	**Semantic Rules**
E->T E’	E.node = E’.syn
	E’.inh = T.node
E’->+ T E'_1	E'_1.inh = mkNode(‘+’, E’.inh, T.node)
	E’.syn = E'_1.syn
E’->- T E'_1	E'_1.inh = mkNode(‘-’, E’.inh, T.node)
	E’.syn = E'_1.syn
E’ -> ε	E’.syn = E’.inh
T-> (E)	T.node = E.node
T->id	T.node = mkLeaf(id, id.entry)
T-> num	T.node=mkLeaf(num, num.val)

Construction of Syntax Trees

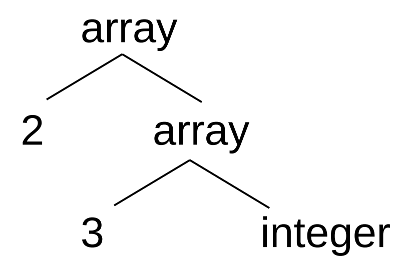
- Annotated Parse Tree : a – 4 + c



Application: Structure of a Type

- Example 5.13 : Array Type

  int [2][3] → array(2, array(3, integer))



Production	**Semantic Rules**
T -> B C	T.t = C.t
	C.b = B.t
B -> int	B.t = integer
B -> float	B.t = float
C -> [num]C_1	C.t = array(num.val, C_1.t)
	C_1.b = C.b
C -> ε	C.t = C.b

Application: Structure of a Type

- Annotated Parse Tree : int [2][3]