Compiler Design

LECTURE-20

Syntax Directed Translation

Topics Covered

- Syntax-directed translation
- Inherited attributes
- Annotated Parse Tree
- Dependency Graph

Phases of a Compiler

- 1. Lexical Analyzer (Scanner)
- Takes source Program and Converts into tokens
- 2. Syntax Analyzer (Parser)
- Takes tokens and constructs a parse tree.
- 3. Semantic Analyzer
- Takes a parse tree and constructs an abstract syntax tree with attributes.

Phases of a Compiler- Contd

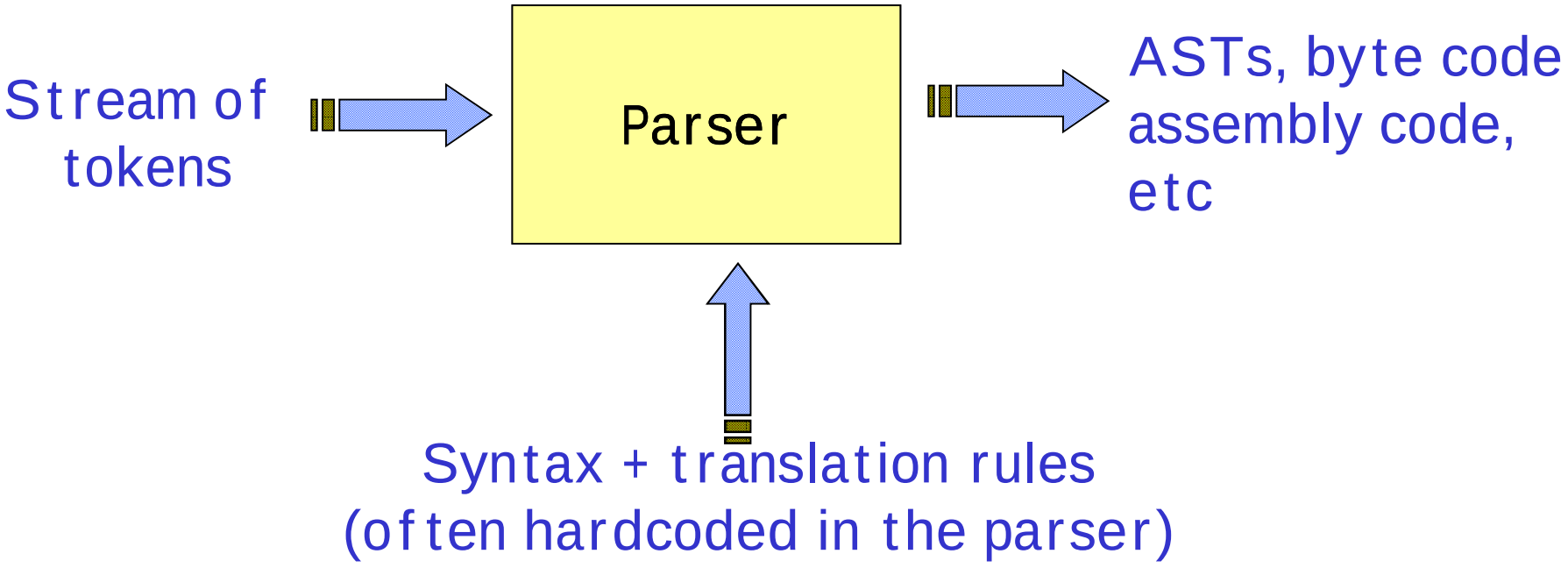
- 4. **Syntax Directed Translation**

Takes an abstract syntax tree and produces an Interpreter code (Translation output)

- 5. Intermediate-code Generator
- Takes an abstract syntax tree and produces un- optimized Intermediate code.

Motivation: Parser as Translator

Syntax-directed translation



Important

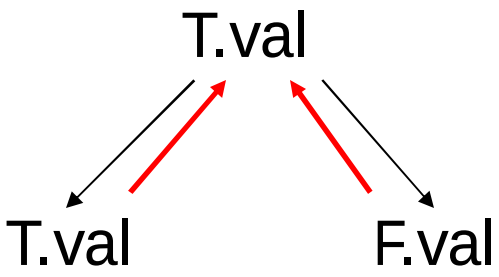
- Syntax directed translation: attaching actions to the grammar rules (productions).
- The actions are executed during the compilation (not during the generation of the compiler, not during run time of the program!). Either when replacing a nonterminal with its rhs (LL, top-down)
  or a handle with a nonterminal (LR, bottom-up).
- The compiler-compiler generates a parser which knows how to parse the program (LR,LL). The actions are “implanted” in the parser and are executed according to the parsing mechanism.

Example :Expressions

- E → E + T
- E → T
- T → T * F
- T → F
- F → (E)
- F → num

Synthesized Attributes

- The attribute value of the terminal at the left hand side of a grammar rule depends on the values of the attributes on the right hand side.
- Typical for LR (bottom up) parsing.
- Example: T→T*F {$$.val=$1.val×$3.val}.



Example :Expressions In LEX

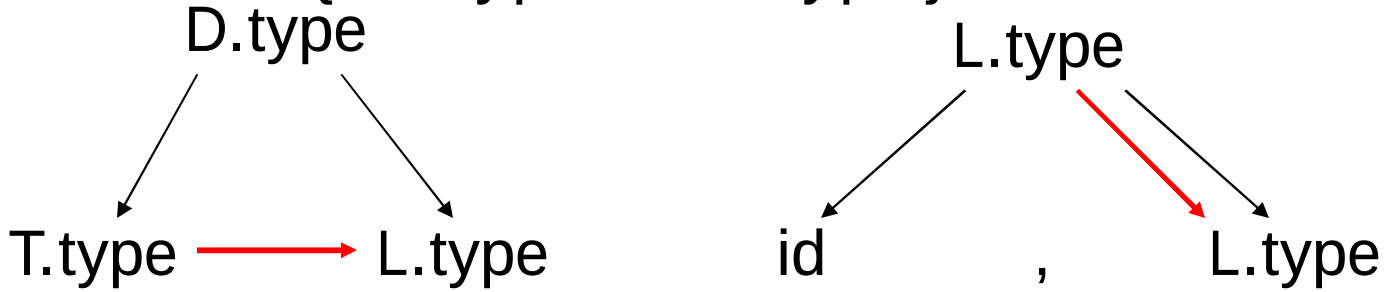
- E → E + T
  {$$.val:=$1.val+$3.val;}
- E → T {$$.val:=$1.val;}
- T → T * F {$$.val:=$1.val*$3.val;}
- T → F {$$.val:=$1.val;}
- F → (E) {$$.val:=$2.val;}
- F → num {$1.val:=$1.val;}

Example 2:Type definitions

- D → T L
- T → int
- T → real
- L → id , L
- L → id

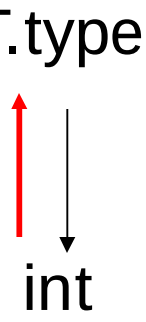
Inherited attributes

- The value of the attributes of one of the symbols to the right of the grammar rule depends on the attributes of the other symbols (left or right).
- Typical for LL parsing (top down).
- D → T {$2.type:=$1.type} L
- L → id , {$3.type:=$1.type} L



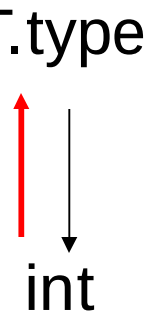
Type definitions

- D → T {$2.type:=$1.type} L
- T → int {$$.type:=int;}
- T → real {$$:=real;}
- L → id , L {gen(id.name,$$.type);
- $3.type:=$$.type;}
- L → id {gen(id.name,$$.type); }



Type definitions: LL(1)

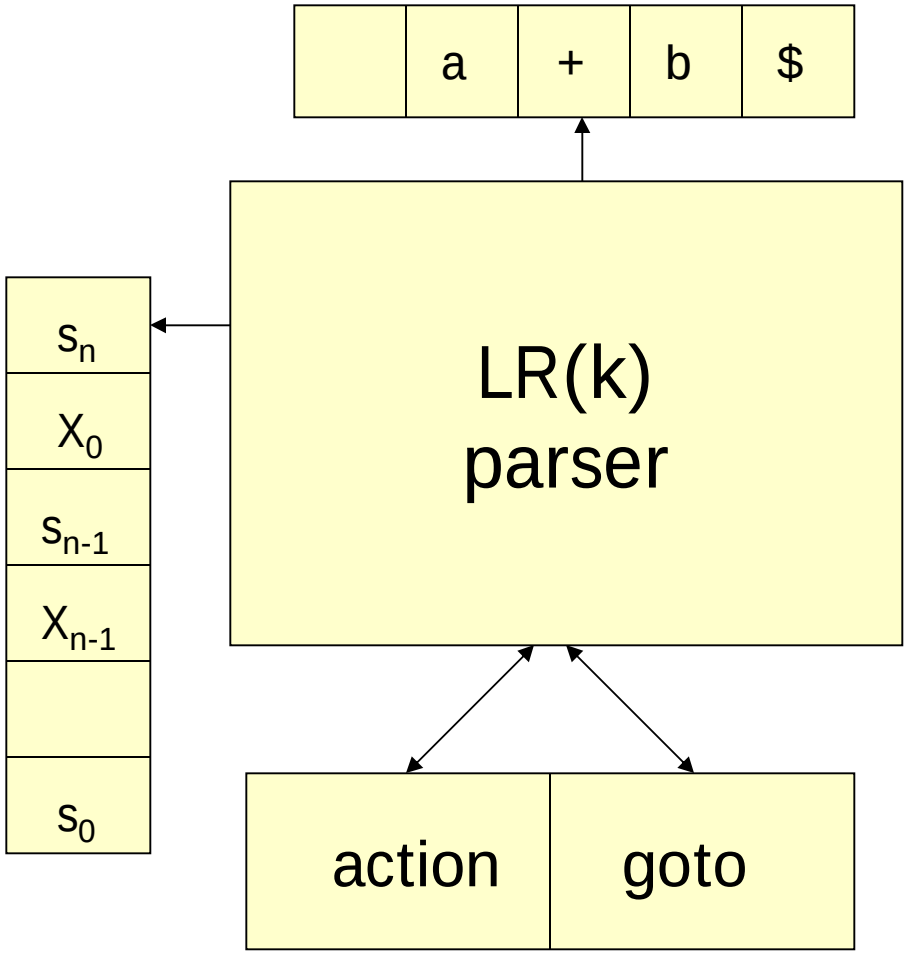
- D → T {$2.type:=$1.type} L
- T → int {$$.type:=int;}
- T → real {$$:=real;}
- L → id {gen(id.name,$$.type);
- $2.type:=$$.type;} R
- R → , id {gen(id.name,$$.type); }
- R → ε



How to arrange things for LL(1) on stack?

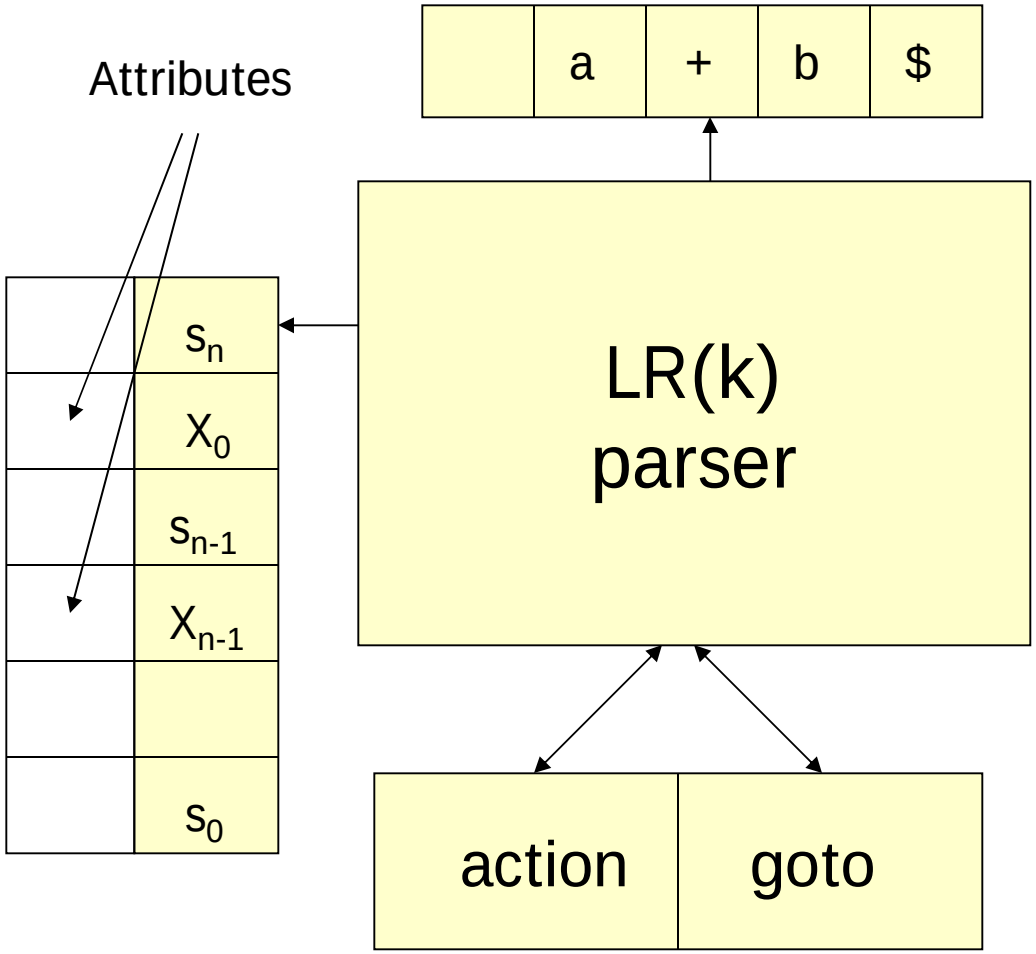
- Include on the stack, except for the grammar symbol also the actions, and a shadow copy for each nonterminal.
- Each time one sees an action on the stack, execute it.
- Shadow copies are used to get synthesized values and pass them further to the right of the rule.

LR parser



- Given the current state on top and current token, consult the action table.
- Either, shift, i.e., read a new token, put in stack, and push new state, or
- or Reduce, i.e., remove some elements from stack, and given the newly exposed top of stack and current token, to be put on top of stack, consult the goto table about new state on top of stack.

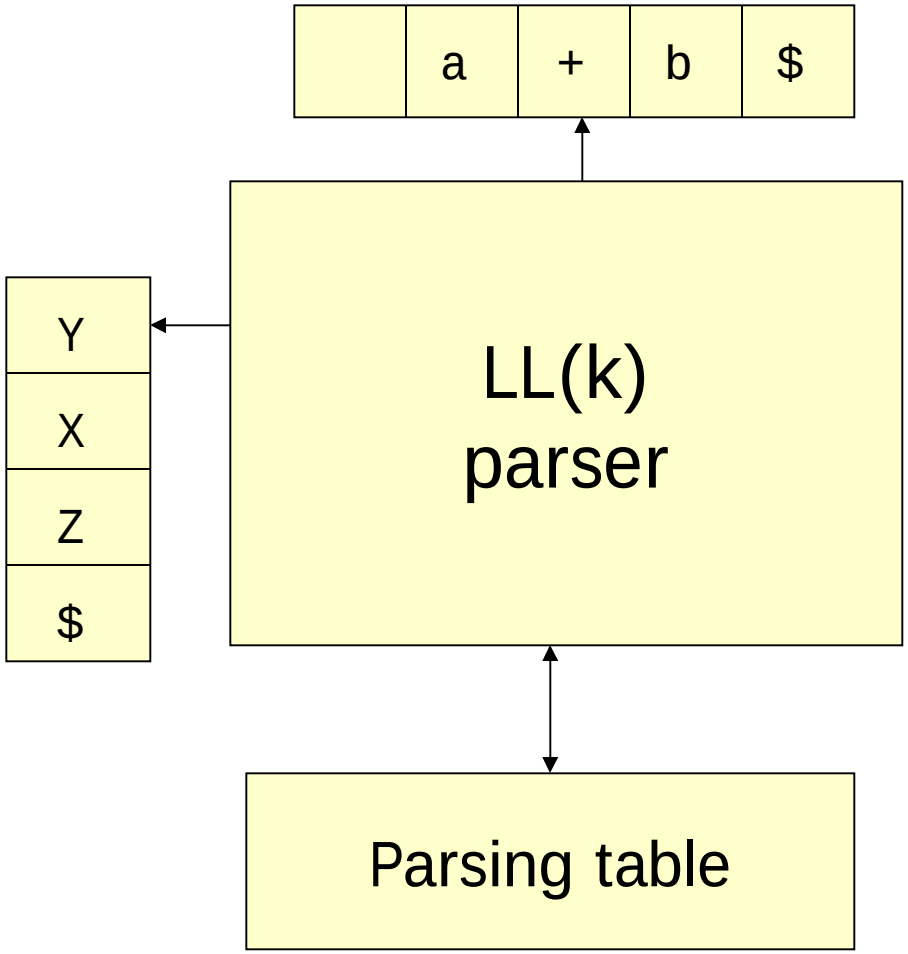
LR parser adapted.



Same as before, plus:

- Whenever reduce step, execute the action associated with grammar rule.
  If left-to right inherited attributes exist, can also execute actions in middle of rule.
- Can put record of attributes, associated with a grammar symbol, on stack.

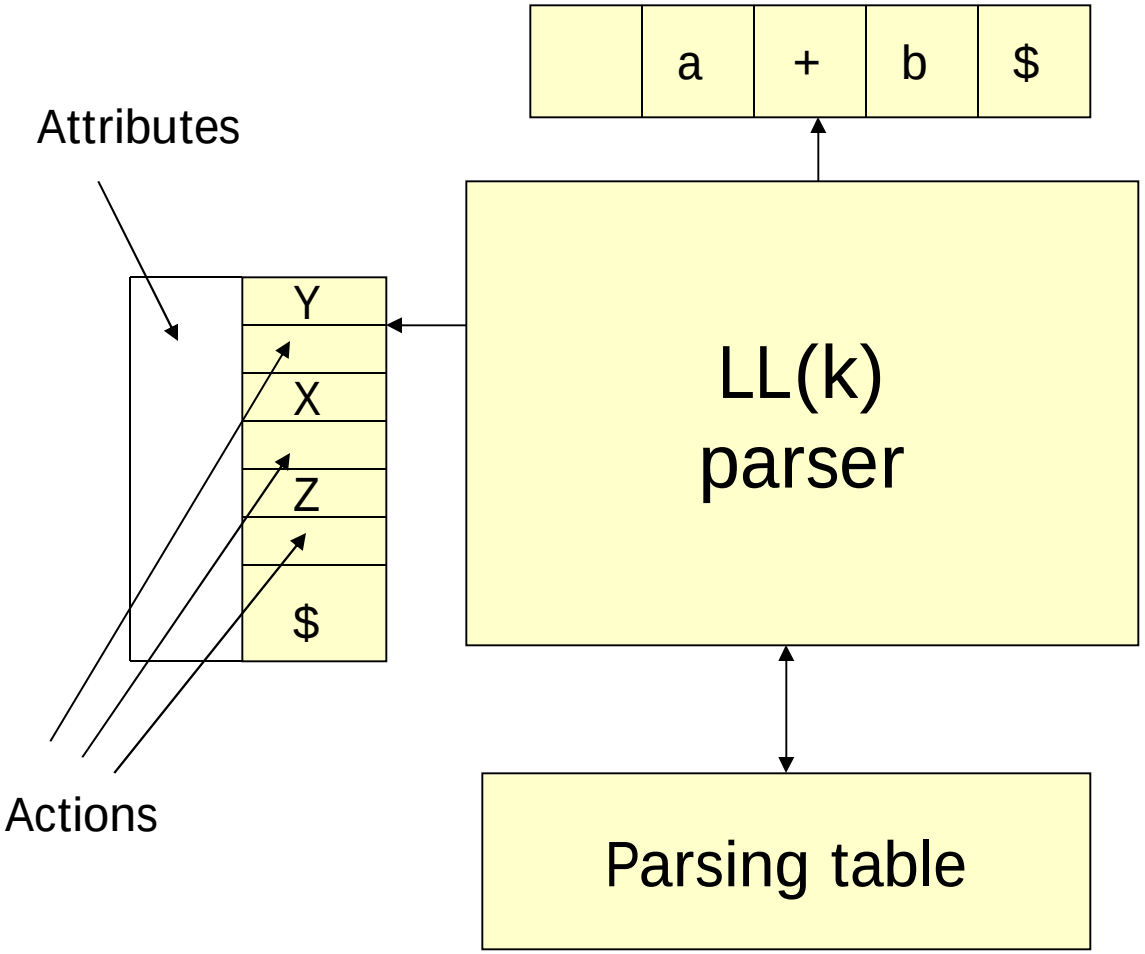
LL parser



- If top symbol *X* a terminal, must match current token *m*.
- If not, pop top of stack. Then look at table *T[X, m]* and push grammar rule there in reverse order.

$	2+3*4$		num.type:=2
$num	+3*4$	F→num	F.type:=2
$F	+3*4$	T→F	T.type:=2
$T	+3*4$	E→T	E.type:=2
$E	+3*4$	shift	
$E+	3*4$	shift	num.type:=3
$E+num	*4$	F→num	F.type:=3
$E+F	*4$	T→F	F.type:=3
$E+T	*4$	shift	
$E+T*	4$	shift	num.type:=4
$E+T*num	$	F→num	F.type:=4
$E+T*F	$	T→T*F	T.type:=12
$E+T	$	E→E*T	E.type:=14

LL parser Adapted



- If top symbol *X* a terminal, must match current token *m*.
- Put actions into stack as part of rules. Hold for each nonterminal a record with attributes.
- If nonterminal, replace top of stack with *shadow copy*. Then look at table *T[X, m]* and push grammar rule there in reverse order.
- If shadow copy, remove. This way nonterminal can deliver values down and up.

On stack	to be read	rule	action
$D	int a,b$		
$(D)L{}T	int a,b$	D→T{}L	
$(D)L{}(T)int{}	int a,b$	T→int{}	
$(D)L{}(T)	a,b$		T.type:=int
$(D)L	a,b$		L.type:=int
$(D)(L)R{}id	a,b$	L→id{}R	
$(D)(L)R	,b$		Gen(a,int), R.type:=int
$(D)(L)(R){}	.b$	R→. id	

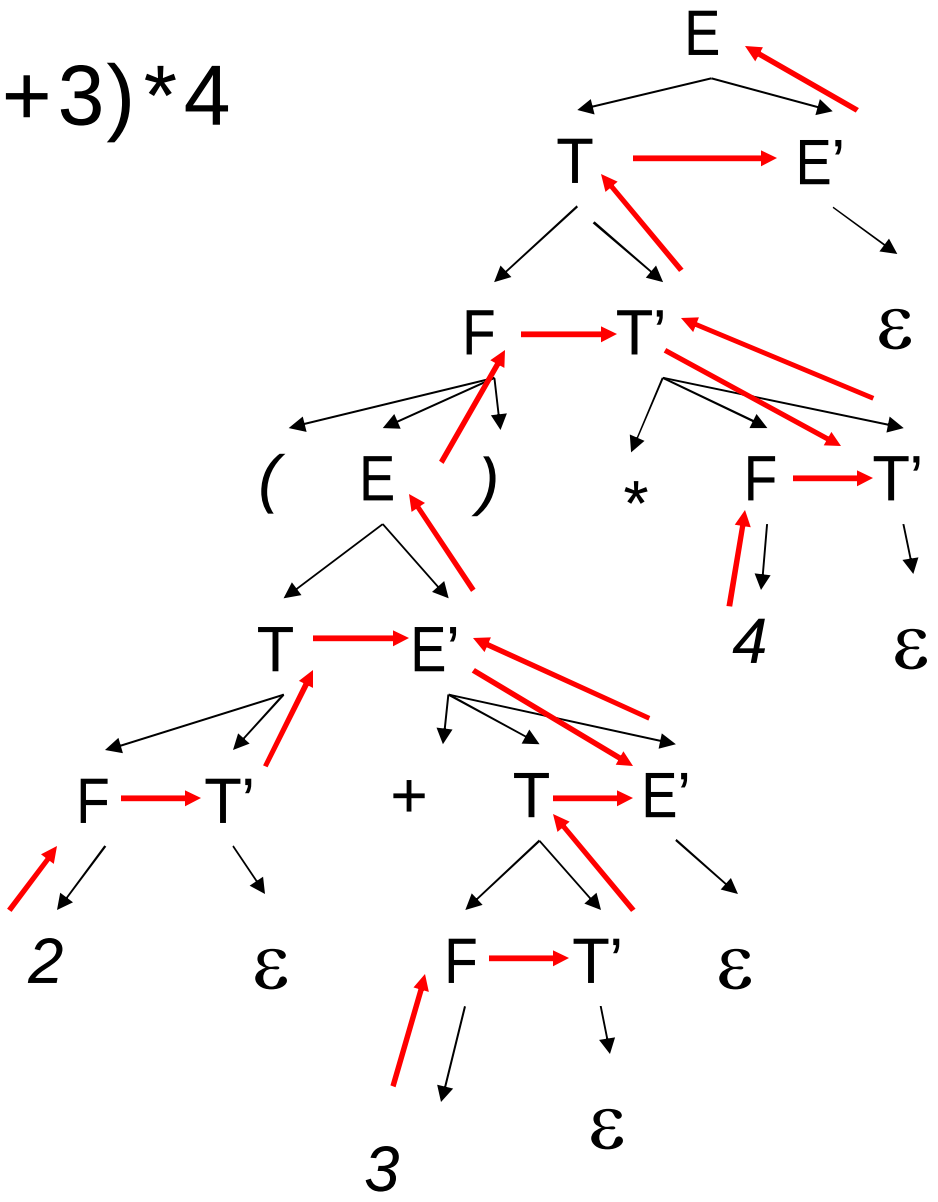
Expressions in LL: Eliminating left recursion

- E → E + T
- E → T
- T → T * F
- T → F
- F → (E)
- F → num

- E → T E'
- E' → + T E'
- E' → ε
- T → F T'
- T' → * F T'
- T' → ε
- F → (E)
- F → num

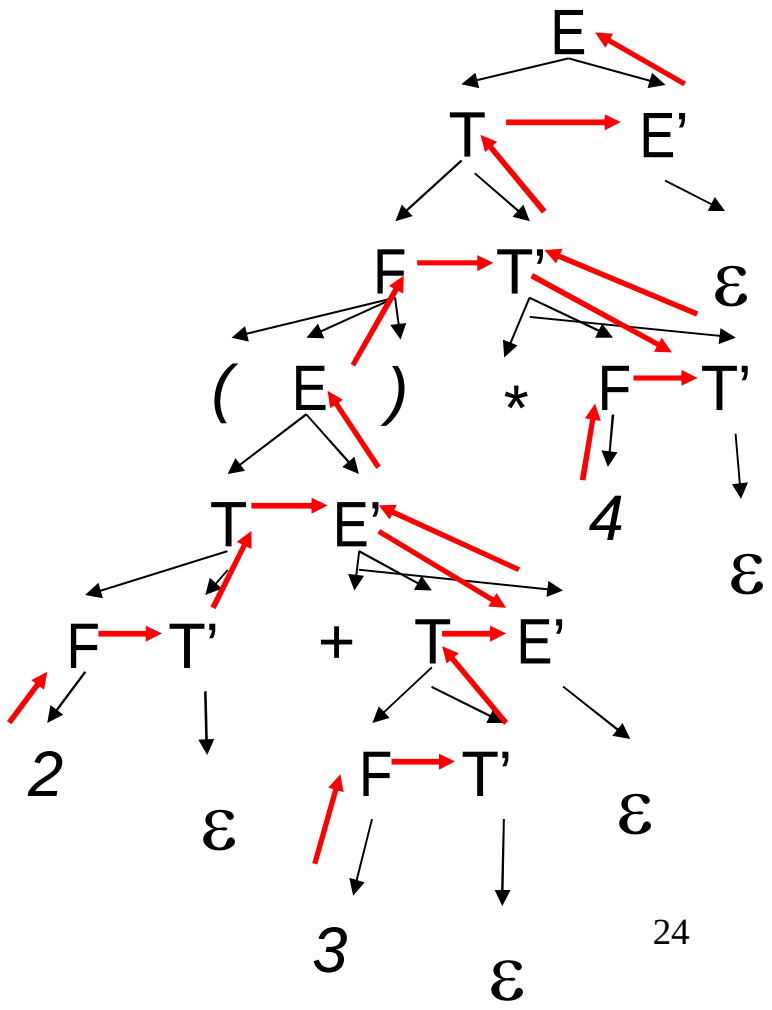
(2+3)*4

- E → T E’
- E’ → + T E’
- E’ → ε
- T → F T’
- T’ → * F T’
- T’ → ε
- F → (E)
- F → num



Actions in LL

- E → T {$2.down:=$1.up;}
  E' {$$.up:=$2.up;}
- E' → + T
  {$3.down:=$$.down+$2.up;}
  E' {$$.up:=$3.up;}
- E' → ε {$$.up:=$$.down;}
- T → F {$2.down:=$1.up;}
  T' {$$.up:=$2.up;}
- T' → * F
  {$3.down:=$$.down+$2.up;}
  T' {$$.up:=$3.down;}
- T' → ε {$$.up:=$$.down;}
- F → (E) {$$.up:=$2.up;}
- F → num {$$.up:=$1.up;}



Syntax Directed Translation Scheme

- A syntax directed translation scheme is a syntax directed definition in which the net effect of semantic actions is to print out a translation of the input to a desired output form.
- This is accomplished by including “emit” statements in semantic actions that write out text fragments of the output, as well as string-valued attributes that compute text fragments to be fed into emit statements.

Syntax-Directed Translation

1. Values of these attributes are evaluated by the **semantic rules** associated with the production rules.
2. Evaluation of these semantic rules:
   - may generate intermediate codes
   - may put information into the symbol table
   - may perform type checking
   - may issue error messages
   - may perform some other activities
   - in fact, they may perform almost any activities.
3. An attribute may hold almost any thing.
   - a string, a number, a memory location, a complex record.
4. Grammar symbols are associated with **attributes** to associate information with the programming language constructs that they represent.

Syntax-Directed Definitions and Translation Schemes

1. When we associate semantic rules with productions, we use two notations:
   - **Syntax-Directed Definitions**
   - **Translation Schemes**

Schemes

A. Syntax-Directed Definitions:

- give high-level specifications for translations
- hide many implementation details such as order of evaluation of semantic actions.
- We associate a production rule with a set of semantic actions, and we do not say when they will be evaluated.

B. Translation Schemes:

- indicate the order of evaluation of semantic actions associated with a production rule.
- In other words, translation schemes give a little bit information about implementation details.

Syntax-Directed Definitions

1. A syntax-directed definition is a generalization of a context-free grammar in which:
   - Each grammar symbol is associated with a set of attributes.
   - This set of attributes for a grammar symbol is partitioned into two subsets called
     - **synthesized** and
     - **inherited** attributes of that grammar symbol.
   - Each production rule is associated with a set of semantic rules.

2. *Semantic rules* set up dependencies between attributes which can be represented by a *dependency graph*.

3. This *dependency graph* determines the evaluation order of these semantic rules.

4. Evaluation of a semantic rule defines the value of an attribute. But a semantic rule may also have some side effects such as printing a value.

Annotated Parse Tree

1. A parse tree showing the values of attributes at each node is called an **annotated parse tree**.

2. The process of computing the attributes values at the nodes is called **annotating** (or **decorating**) of the parse tree.

3. Of course, the order of these computations depends on the dependency graph induced by the semantic rules.

Syntax-Directed Definition

In a syntax-directed definition, each production $A \rightarrow \alpha$ is associated with a set of semantic rules of the form:

$$b = f(c_1, c_2, \ldots, c_n)$$

where f is a function and b can be one of the followings:

➔ b is a synthesized attribute of A and $c_1, c_2, \ldots, c_n$ are attributes of the grammar symbols in the production ($A \rightarrow \alpha$).

OR

➔ b is an inherited attribute one of the grammar symbols in α (on the right side of the production), and $c_1, c_2, \ldots, c_n$ are attributes of the grammar symbols in the production ($A \rightarrow \alpha$).

Attribute Grammar

- So, a semantic rule $b=f(c_1,c_2,\ldots,c_n)$ indicates that the attribute b *depends* o*n* attributes $c_1,c_2,\ldots,c_n$.

- In a **syntax-directed definition**, a semantic rule may just evaluate a value of an attribute or it may have some side effects such as printing values.

- An **attribute grammar** is a syntax-directed definition in which the functions in the semantic rules cannot have side effects (they can only evaluate values of attributes).

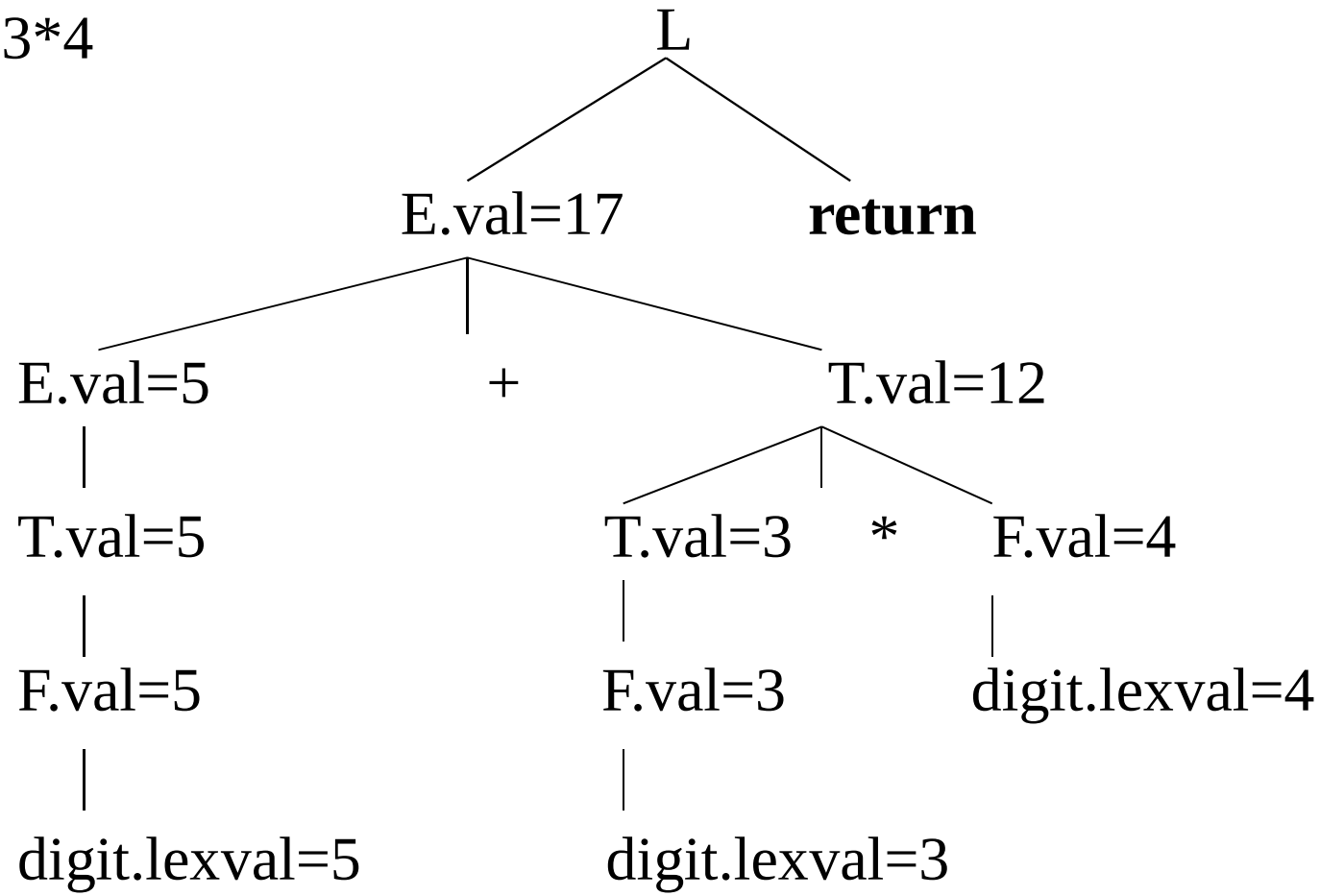
Syntax-Directed Definition -- Example

Production	Semantic Rules
$L \rightarrow E$ **return**	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).

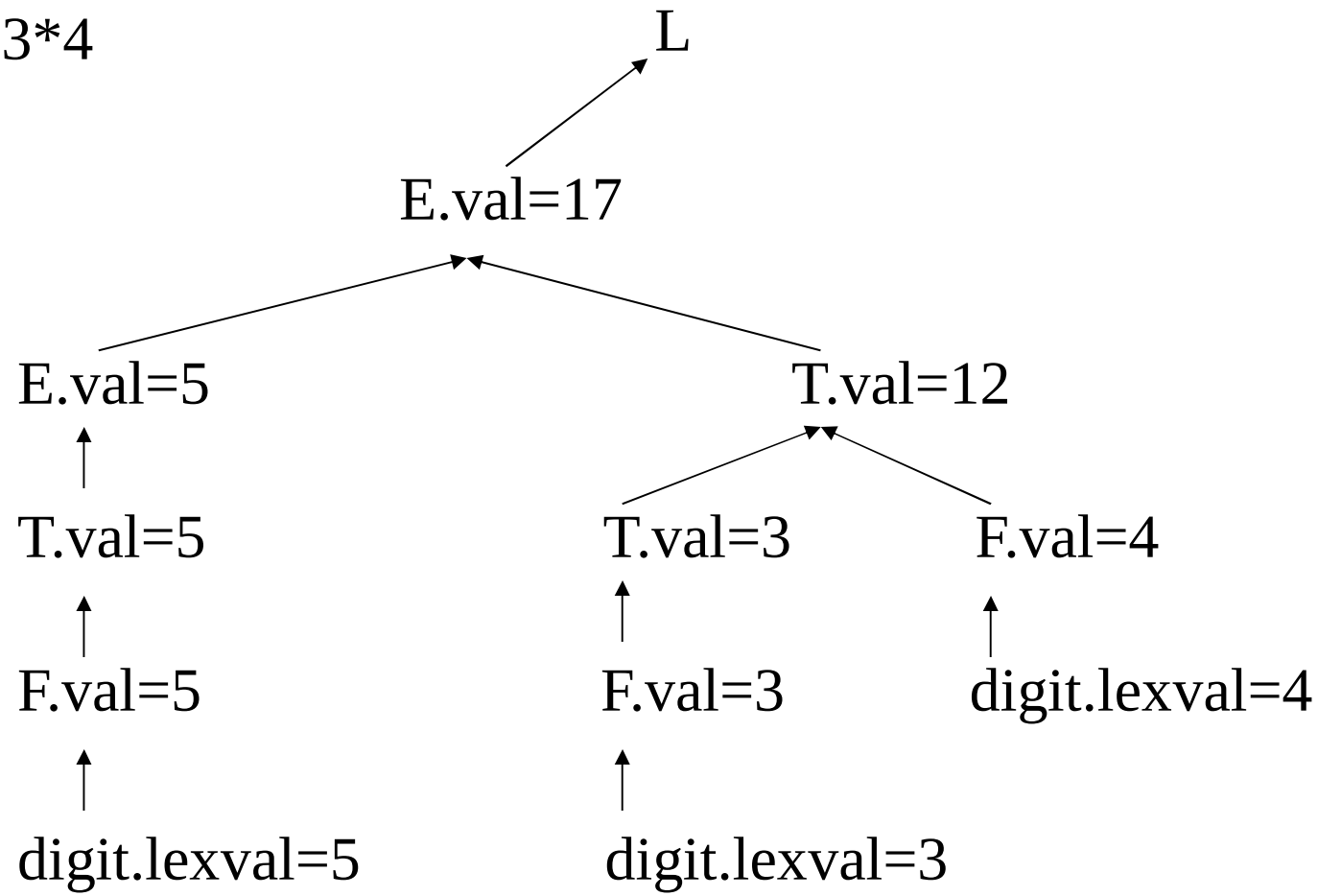
Annotated Parse Tree -- Example

Input: 5+3*4



Dependency Graph

Input: 5+3*4



Syntax-Directed Definition – Example2

Production	**Semantic Rules**
$E \rightarrow E_1 + T$	E.loc=newtemp(), E.code = E_1.code || T.code || add E_1.loc,T.loc,E.loc
$E \rightarrow T$	E.loc = T.loc, E.code=T.code
$T \rightarrow T_1 * F$	T.loc=newtemp(), T.code = T_1.code || F.code || mult T_1.loc,F.loc,T.loc
$T \rightarrow F$	T.loc = F.loc, T.code=F.code
$F \rightarrow (E)$	F.loc = E.loc, F.code=E.code
F → **id**	F.loc = **id**.name, F.code=""

1. Symbols E, T, and F are associated with synthesized attributes *loc* and *code*.
2. The token **id** has a synthesized attribute *name* (it is assumed that it is evaluated by the lexical analyzer).
3. It is assumed that || is the string concatenation operator.

Syntax-Directed Definition – Inherited Attributes

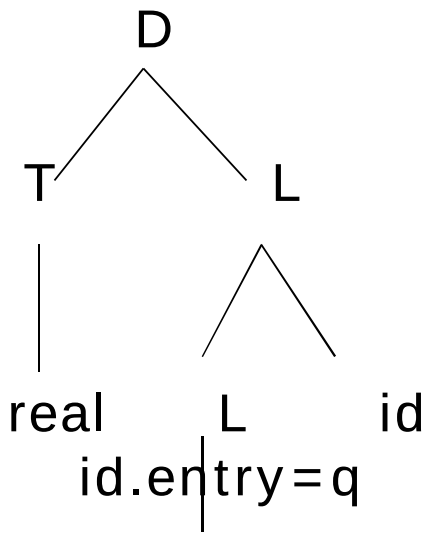
Production	Semantic Rules
$D \rightarrow T\ L$	L.in = T.type
$T \rightarrow$ **int**	T.type = integer
$T \rightarrow$ **real**	T.type = real
$L \rightarrow L_1$ **id**	L_1.in = L.in, addtype(**id**.entry,L.in)
$L \rightarrow$ **id**	addtype(**id**.entry,L.in)

1. Symbol T is associated with a synthesized attribute *type*.

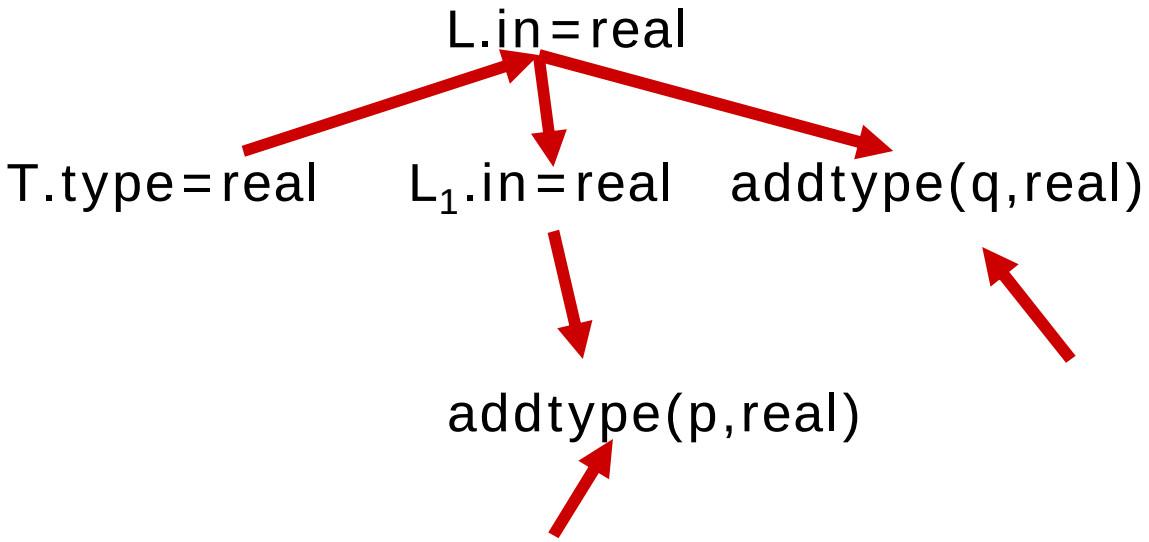
2. Symbol L is associated with an inherited attribute *in.*

A Dependency Graph – Inherited Attributes

Input: real p q



parse tree



dependency graph

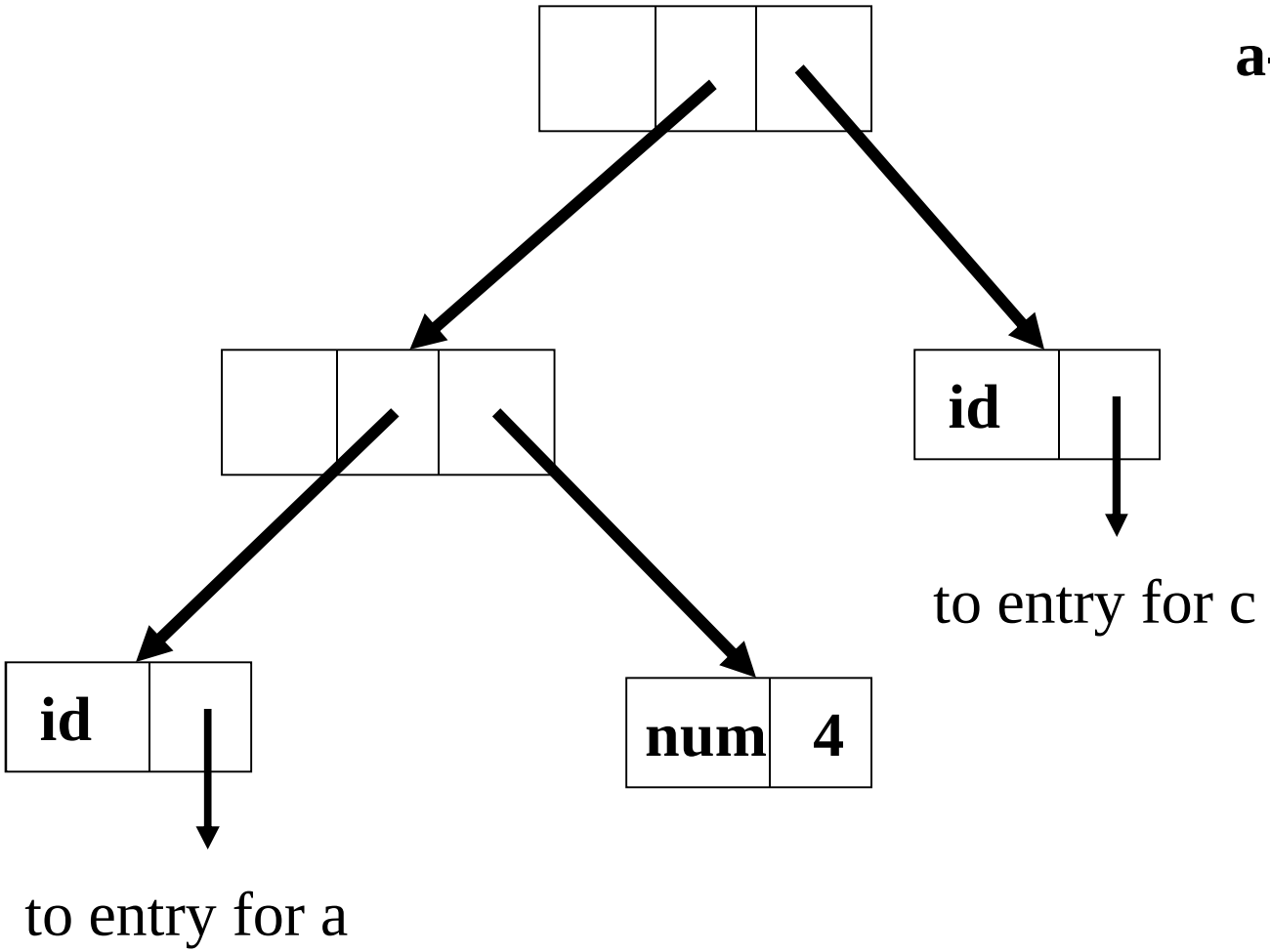
Syntax Trees

1. Decoupling Translation from Parsing-Trees.
2. Syntax-Tree: an intermediate representation of the compiler's input.
3. Example Procedures:
   mknode, *mkleaf*
4. Employment of the synthesized attribute *nptr* (pointer)

PRODUCTION	**SEMANTIC RULE**
$E \rightarrow E_1$ + T	E.*nptr* = *mknode*("+",E_1.*nptr* ,T.*nptr*)
$E \rightarrow E_1$ - T	E.*nptr* = *mknode*("-",E_1.*nptr* ,T.*nptr*)
$E \rightarrow T$	E.*nptr* = T.*nptr*
$T \rightarrow$ (E)	T.*nptr* = E.*nptr*
$T \rightarrow$ id	T.*nptr* = *mkleaf*(id, id.*lexval*)
$T \rightarrow$ num	T.*nptr* = *mkleaf*(num, num.*val*)

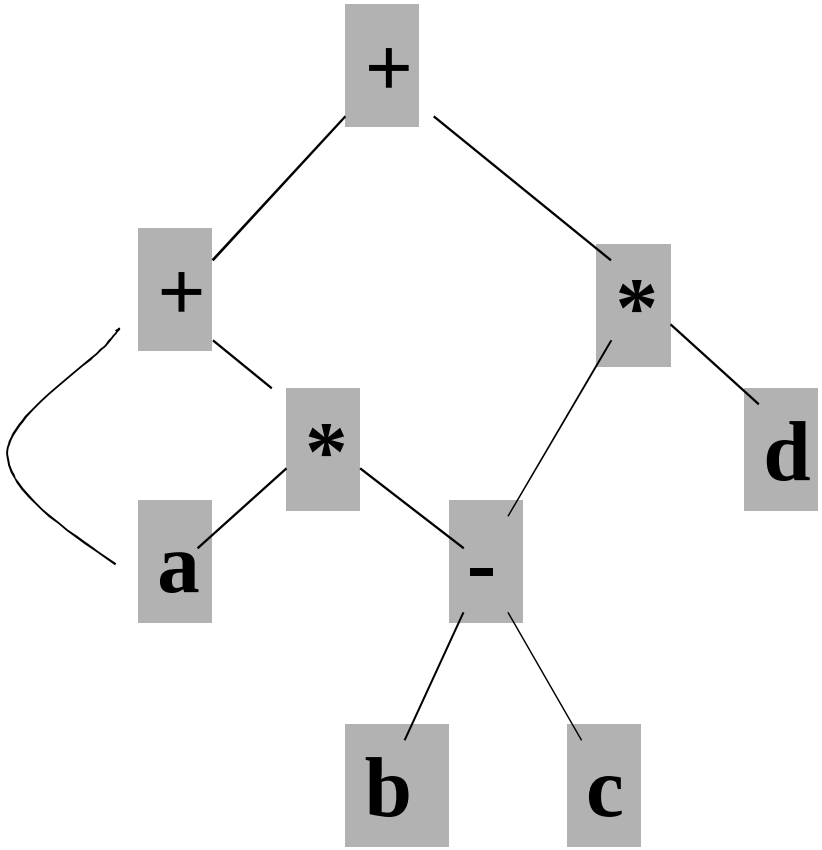
Draw the Syntax Tree

a-4+c



Directed Acyclic Graphs for Expressions

a + a * (b – c) + (b – c) * d



– Example an S-attributed definition:
  - A syntax directed definition that uses synthesized attributes exclusively is said to be an S-attributed definition.

Production	semantic rules
L ->E n	print(E.val)
E->E1 + T	E.val = E1.val + T.val
E->T	E.val = T.val
T->T1 * F	T.val = T1.val * F.val
T->F	T.val = F.val
F->(E)	F.val = E.val
F->digits	F.val = digits.lexval

3*5+4n

– L-attributed definitions:
  - A syntax directed definition is L-attributed if each inherited attribute of Xj, 1<=j<=n, on the right side of A->X1X2…Xn depends only on
    – attributes of the symbols X1, X2, …, Xj-1.
    – the inherited attributes of A.
  - L stands for Left since information appears to flow from left to right in the compilation process.

  - Example:

A->LM	{L.i=A.i; M.i=L.s; A.s = M.s}
A->QR	{R.i = A.i; Q.i = R.s; A.s = Q.s}

– Relation between S-attributed definitions and L-attributed definitions?
– Why L-attributed definitions are important?