MODULE IV - Syntax directed translation and Intermediate code generation

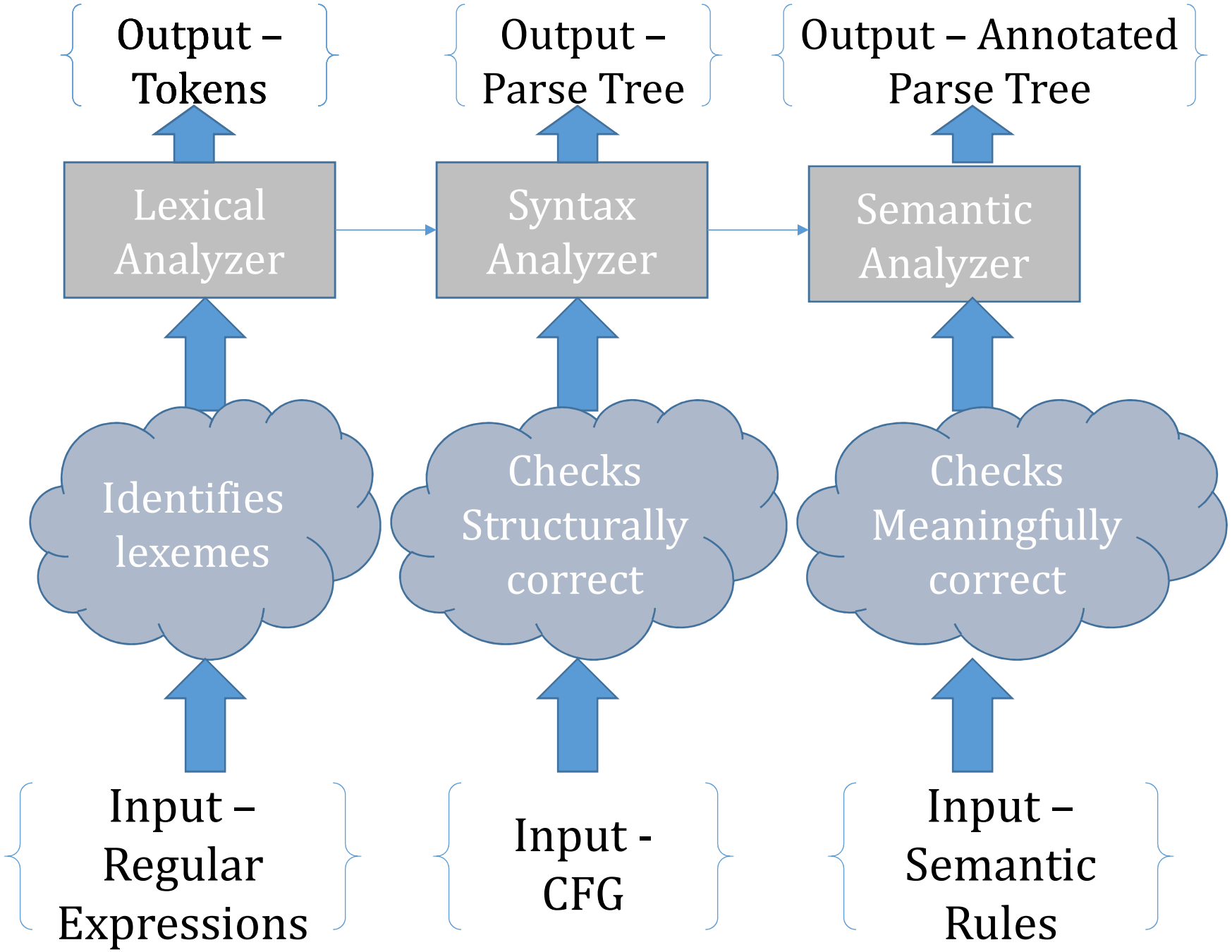
Syntax directed translation - Syntax directed definitions, S-attributed definitions, L-attributed definitions, Bottom-up evaluation of S-attributed definitions.

Run-Time Environments - Source Language issues, Storage organization, Storage-allocation strategies.

Intermediate Code Generation - Intermediate languages, Graphical representations, Three-Address code, Quadruples, Triples.

Need for Semantic Analysis

- A phase by a compiler that adds semantic information to the 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


Output – Tokens
Output – Parse Tree
Output – Annotated Parse Tree
Lexical Analyzer
Syntax Analyzer
Semantic Analyzer
Identifies lexemes
Checks Structurally correct
Checks Meaningfully correct
Input – Regular Expressions
Input - CFG
Input – Semantic Rules

- Typical examples of semantic information that is added and checked are
  - Typing information (type checking) and
  - The binding of variables and function names to their definitions (object binding).
- Sometimes also some early code optimization is done in this phase.
- For this phase the compiler usually maintains *symbol tables* in which it stores what each symbol (variable names, function names, etc.) refers to.

Following things are done in Semantic Analysis

1. **Disambiguate Overloaded operators**

- If an operator is overloaded, one would like to specify the meaning of that particular operator

2. **Type checking**

- The process of verifying and enforcing the constraints of types is called type checking
- This may occur either at compile-time (a static check) or run-time (a dynamic check).
- Static type checking is a primary task of the semantic analysis carried out by a compiler.
- If type rules are enforced strongly (that is, generally allowing only those automatic type conversions which do not lose information), the process is called strongly typed, if not, weakly typed.

3. **Uniqueness checking**

- Whether a variable name is unique or not, in the its scope.

4. **Type coercion**

- Converting from one type to another
- If some kind of mixing of types is allowed.
- Done in languages which are not strongly typed. This can be done dynamically as well as statically.

5. **Name Checks**

- Check whether any variable has a name which is not allowed
- Ex. Name is same as an identifier(Ex. int in java).

- Eg : int a ; binding of the type integer to the variable ‘a’
- Semantic Analysis also uses CFG
- Meaning of each production / a programming construct is added to the CFG in the form of certain rules – “Semantic Rules”
- Semantic Analyzer therefore have CFGs with Semantic Rules – Syntax Directed Translation (SDT)
- So, meanings of programming language constructs are explained with the help of SDTs

Semantic Rules

- Specifies what is to be done when a particular grammar production is used while parsing
- There should be some relation between CFG and Semantic Rules – Attributes of a grammar symbol
- Attributes
  - An additional entity or information about the grammar symbol
  - Like type of the grammar symbol, the memory location to be used for the grammar symbol
  - Two types : **Inherited & Synthesized**
  - Corresponding to a grammar symbol there could be inherited and/or synthesized attributes

- There are two kinds of attributes:

1. **Synthesized Attributes**

- They are computed from the values of the attributes of the children nodes.

2. **Inherited Attributes**

- They are computed from the values of the attributes of both the siblings and the parent nodes.

Syntax Directed Translation

- The Principle of Syntax Directed Translation states that the meaning of an input sentence is related to its syntactic structure, i.e., to its Parse-Tree.
- By SDT we indicate those formalisms for specifying translations for programming language constructs guided by context-free grammars.
- We associate “attributes” to the grammar symbols representing the language constructs.
- Values for attributes are computed by “Semantic Rules” associated with grammar productions.

- There are two types of SDTs
    - Syntax-directed definitions (SDDs)
    - Translation schemes (TS)
- These are two notations for associating semantic rules with the grammar productions

- Syntax-directed definitions (SDD)
  - High-level specifications for translations (conversion)
  - Hide the implementation details
  - Free the user from having to specify explicitly the order in which the translation takes place
- Translation schemes
  - Indicates the order in which semantic rules are to be evaluated
  - Allow some implementation details to be shown

Syntax Directed Definition

- Syntax Directed Definition is a generalization of context-free grammars in which each Grammar symbol has an associated set of attributes; partitioned into two subsets called the synthesized and inherited attributes, of that grammar symbol
- Example : Grammar for arithmetic expression
- Here we call it as a “Desk Calculator”

E -> E + T
E -> T
T -> T * F
T -> F
F -> (E)
F -> digit

CFG of arithmetic expression

- Grammar symbols are : E, T, F and num
- Associated with each grammar symbol there will be an attribute(an additional information)
- Here we require the value of each grammar symbol
- 1+2*3
- We will be assigning values to each and every grammar symbol
- Attributes : E, T, F => val, num => lexval
- Use the dot operator for relating attribute with grammar symbol
- E.val, T.val, F.val, digit.lexval
- How the value of the expression 1+2*3 is evaluated is explained with the help of SDDs

Form of a Syntax-Directed Definition

- Each grammar production A→α has associated with it a set of semantic rules of the form b:=f(c1,c2,……,ck) where f is a function and either

1. b is a synthesized attribute of A and c1,c2…ck are attributes belonging to the grammar symbols of the production or
2. b is an inherited attribute at one of the grammar symbols on the right side of the production and c1,c2..ck are attributes belonging to the grammar symbols of the production

In either case, we say that the attribute b depends on attributes c1,c2…ck

Functions in semantic rules will often be written as expressions

- Occasionally, the only purpose of semantic rule in a syntax-directed definition is to create a side-effect
- Such semantic rules are written as procedure calls or program fragments eg printing a value or updating a global variable
- They can be thought of as rules defining the values of dummy synthesized attributes of the non-terminal on the left side of the associated production; the dummy attribute and the sign = in the semantic rule are not shown

Example for SDD

PRODUCTION	SEMANTIC RULE
E→E1+T	E.val=E1.val+T.val

- Example 5.1 The syntax directed definition for a desk calculator program. This definition associates an integer-valued synthesized attribute called val with each non-terminal E,T and F
- For each E,T and F production, the semantic rule computes the value of attribute val for the non-terminal on the left side from the values of val for the non-terminals on the right side

PRODUCTION	**SEMANTIC RULE**
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 → num	F.val=digit.lexval

PRODUCTION	SEMANTIC RULES
$L \rightarrow E$ n	*print*(*E.val*)
$E \rightarrow E_1 + T$	$E.val := E_1.val + T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T_1 * F$	$T.val := T_1.val \times F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow (E)$	$F.val := E.val$
$F \rightarrow$ **digit**	$F.val :=$ **digit**.*lexval*

Fig. 5.2. Syntax-directed definition of a simple desk calculator.

- The token digit has a synthesized attribute lexval whose value is assumed to be supplied by the lexical analyzer
- The rule associated with the production L→E n for the starting nonterminal L is just a procedure that prints as output the value of the arithmetic expression generated by E, this rule can be thought of as defining a dummy attribute for the nonterminal L

S-Attributed definition

- A syntax-directed definition that uses synthesized attributes exclusively is said to be an S-attributed definition
- A parse tree for an S-attributed definition can always be annotated by evaluating the semantic rules for the attributes at each node bottom up, from the leaves to the root

- Example 5.2
- The S-attributed definition in example 5.1 specifies a desk calculator that read an input line containing an arithmetic expression involving digits , parentheses, the operators + and * followed by a new line character n and prints the value of an expression

- For eg, given the expression 3*5+4 followed by a newline, the program prints the value 19
- Fig shows the annotated parse tree for the input 3*5+4n
- The output printed at the root of the tree, is the value of E.val at the first child of the root

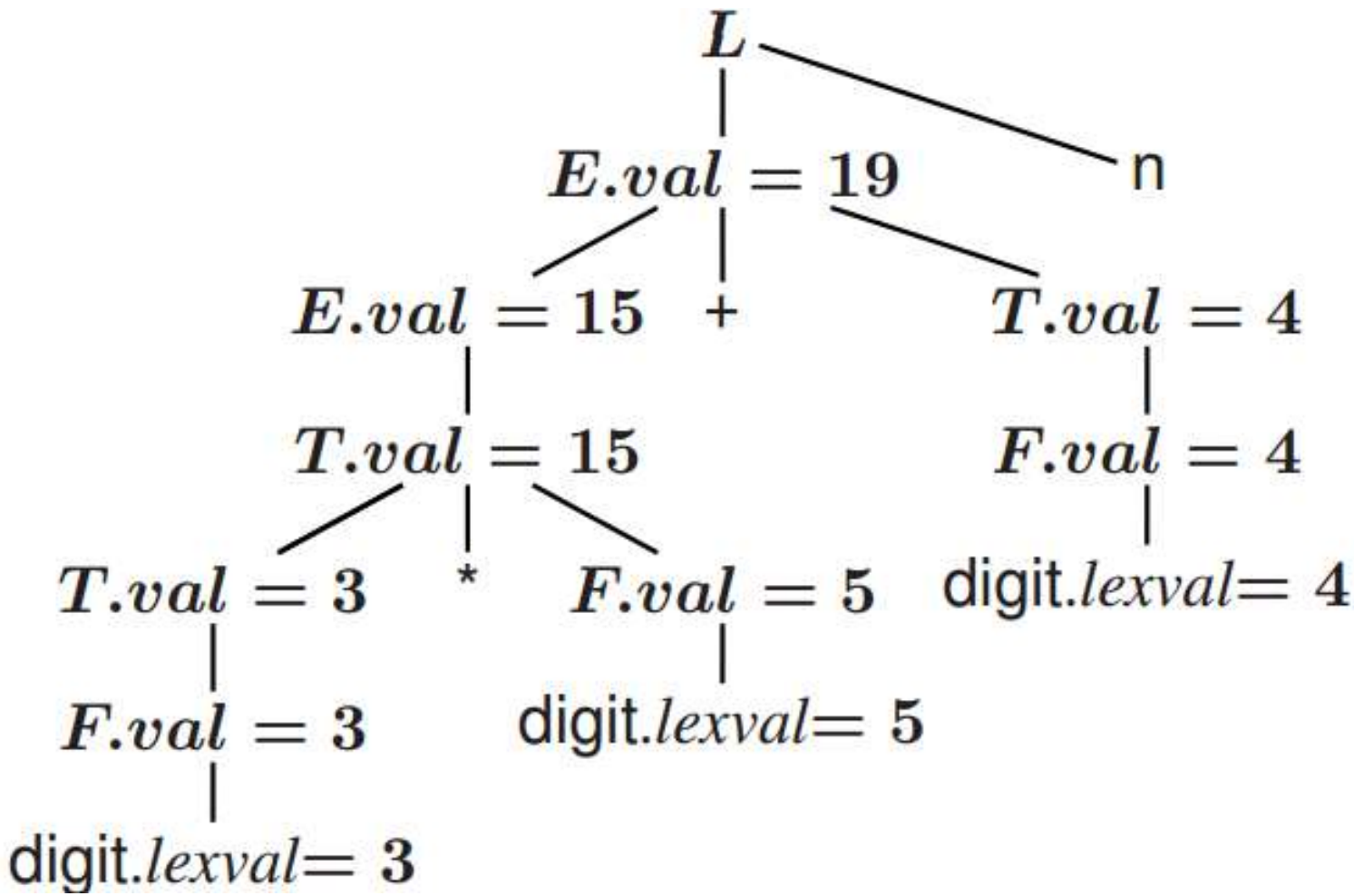


Fig 5.3 Annotated parse tree for 3*5+4n

How the attribute values are computed

- Consider the left most bottom most interior node,which corresponds to the use of the production F→digit
- The corresponding semantic rule

  F.val = digit.lexval

  defines the attribute F.val at that node to have the value 3 as the value of digit.lexval at the child of this node is 3
- Similarly, at the parent of this F-node, the attribute T.val has the value 3

- Consider the node for the production T → T *F
- The value of the attribute T.val at this node is defined by

Production	Semantic Rule
T → T1 * F	T.val = T1.val X F.val

- When we apply the semantic rule at this node, T1.val has the value 3 from the left child and F.val has the value 5 from the right child
- Thus T.val acquires the value 15 at this node
- The rule associated with the production for the starting nonterminal L → En prints the value of the expression generated by E

Inherited attributes

- An inherited attribute is one whose value at a node in a parse tree is defined in terms of attributes at the parent and/or siblings of that node
- Convenient for expressing the dependence of a programming language construct on the context in which it appears
- For eg, we can use an inherited attribute to keep track of whether an identifier appears on the left or right side of an assignment in order to decide whether the address or the value of the identifier is needed

- It is always possible to rewrite a syntax-directed definition to use only synthesized attributes, but we can use inherited attributes also
- The following example shows an inherited attribute that distributes type information to the various identifiers in a declaration

- Example of an inherited attribute : int a,b,c
- Type followed by list of variables
- Type declaration D can be of the form Type followed by a list of identifiers
- Type can be either int or real
- List of identifiers is a set of identifiers separated by commas
- So, the CFG for variable declaration

    D -> T L

    T -> int

    T -> real

    L -> L1, id

    L -> id

- Attributes
- T => type
- L => in
- id => entry
- T.type determines L.in

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*, $L.in$)

Fig. 5.4. Syntax-directed definition with inherited attribute $L.in$.

- A declaration generated by the nonterminal D in the syntax directed definition in fig 5.4 consists of the keyword int or real, followed by a list of identifiers
- The nonterminal T has a synthseized attribute type, whose value is determined by the keyword in the declaration
- The semantic rule L.in=T.type, associated with the production D → TL sets the inherited attribute L.in to the type of the declaration
- The rule then pass this type down the parse tree using the inherited attribute L.in

- Rules associated with the procedure for L, call procedure addtype to add the type of each identifier to its entry in the symbol table (pointed to by the attribute entry)
- Fig 5.5 shows the annotated parse tree for the sentence **real id1, id2, id3**
- The value of L.in at the three L-nodes gives the type of the identifiers id1, id2 and id3
- These values are determined by computing the value of the attribute T.type at the left child of the root and then evalutaing L.in top-down at the three L-nodes in the right subtree of the root
- At each L-node, we also call a procedure addtype to insert into the symbol table the fact that the identifier at the right child of this node has type real

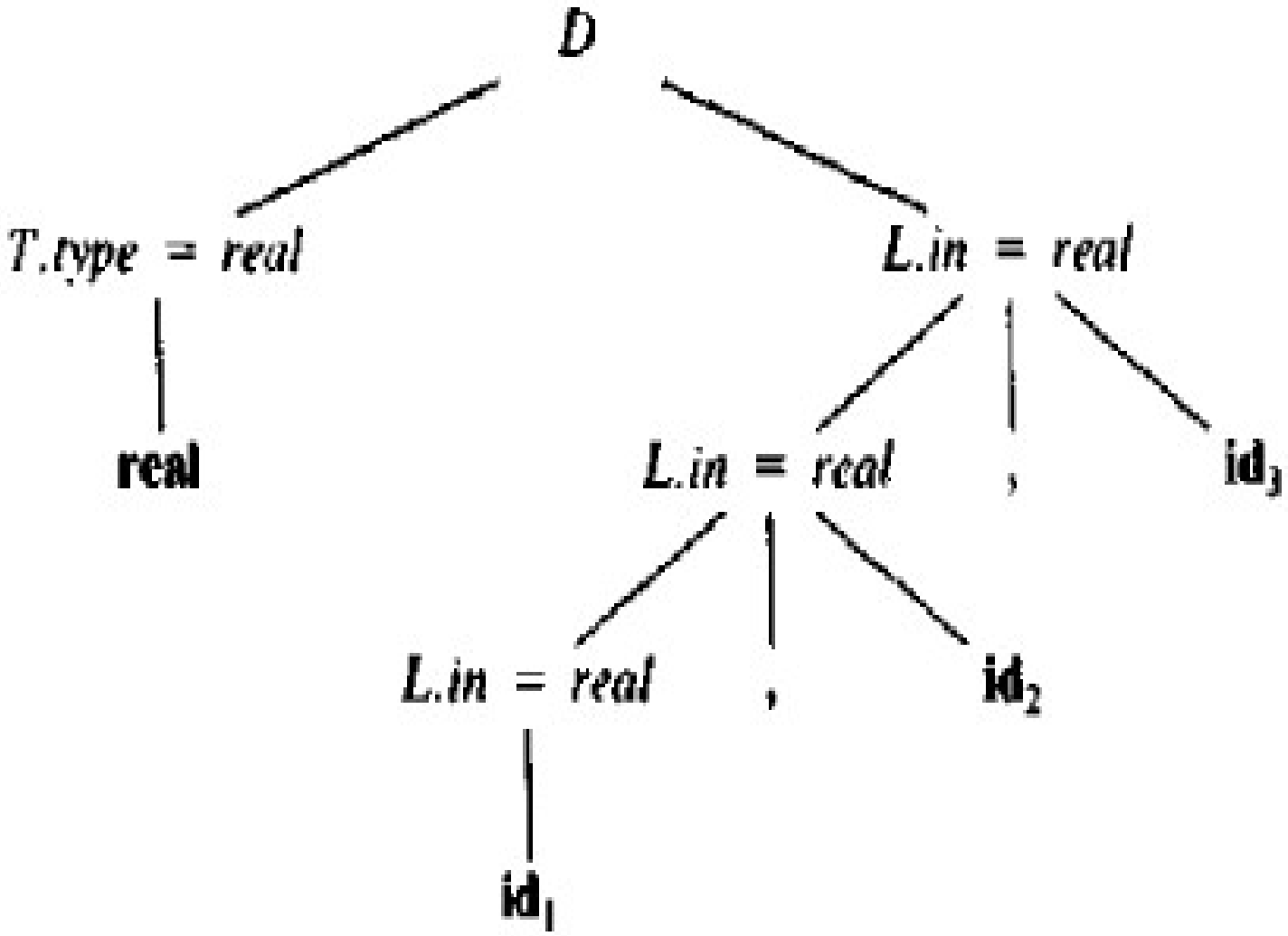


Fig. 5.5. Parse tree with inherited attribute *in* at each node labeled *L*.

- ***Semantic rules*** set up dependencies between attributes that will be represented by a graph – dependency graph
- From the dependency graph, we derive an evaluation order for the semantic rules
- Evaluation of the semantic rules defines the values of the attributes at the nodes in the parse tree for the input string

Fig. 5.1. Conceptual view of syntax-directed translation.

Dependency Graph

- If an attribute b at a node in a parse tree depends on an attribute c, then the semantic rule for b at that node must be evaluated after the semantic rule that defines c
- The interdependencies among the inherited and synthesized attributes at the nodes in a parse tree can be depicted by a directed graph called a dependency graph
- Before constructing a dependency graph for a parse tree, we put each semantic rule into the form b=f(c1,c2,…ck)

- For each semantic rule that consists of a procedure call, we introduce a dummy synthesized attribute b
- The graph has a node for each attribute and an edge to the node for b from the node for c if attribute b depends on attribute c

for each node n in the parse tree **do**

for each attribute ‘a’ of the grammar symbol at n **do**

construct a node in the dependency graph for ‘a’;

for each node n in the parse tree **do**

for each semantic rule b:=f(c1,c2,…..,ck)

associated with the production used at n **do**

for i:=1 to k **do**

construct an edge from the node for ci to the node for b;

- Example
- For the production A → XY associated with a semantic rule A.a = f(X.x, Y.y),
- The rule defines a synthesized attribute A.a that depends on the attributes X.x and Y.y
- If this production is used in the parse tree, then there will be three nodes A.a,X.x and Y.y in the dependency graph with an edge to A.a from X.x since A.a depends on X.x and an edge to A.a fromY.y since A.a also depends on Y.y

- If the production A → XY has the semantic rule X.i = g(A.a,Y.y) associated with it, then
- There will be an edge to X.i from A.a and fromY.y since X.i depends on both A.a and Y.y
- Whenever the following production is used in a parse tree,we add edges shown in fig 5.6 to the dependency graph

PRODUCTION	SEMANTIC RULE
E → E1 + E2	E.val = E1.val + E2.val

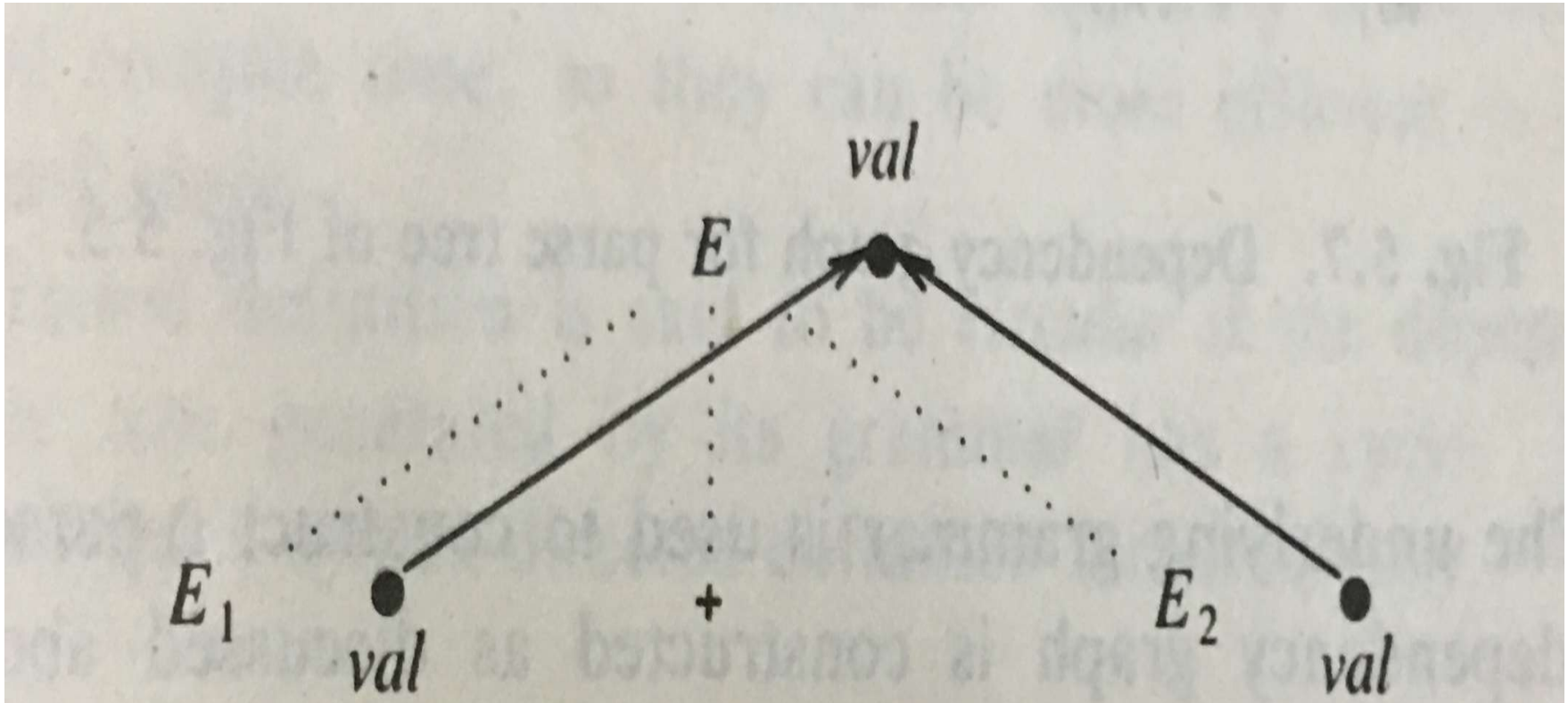


Fig. 5.6. *E.val* is synthesized from E_1.*val* and E_2.*val*.

- The three nodes of the dependency graph marked by black dot represent the synthesized attributes E.val, E1.val and E2.val at the corresponding nodes in the parse tree
- The edge to E.val from E1.val shows that E.val depends on E1.val and the edge to E.val fromE2.val shows that E.val also depends on E2.val
- The dotted lines represent the parse tree and are not part of the dependency graph

- Fig 5.7 shows the dependency graph for the parse tree in fig 5.5
- Nodes in the dependency graph are marked by numbers
- There is an edge to node 5 for L.in from node 4 for T.type as the inherited attribute L.in depends on the attribute T.type according to the semantic rule L.in=T.type for the production D→TL

- The two downward edges into nodes 7 and 9 arise because L1.in depends on L.in according to the semantic rule L1.in = L.in, for the production L →L1 , id
- Each of the semantic rules addtype(id.entry, L.in) associated with the L-productions leads to the creation of a dummy attribute
- Nodes 6,8 and 10 are constructed for these dummy attributes

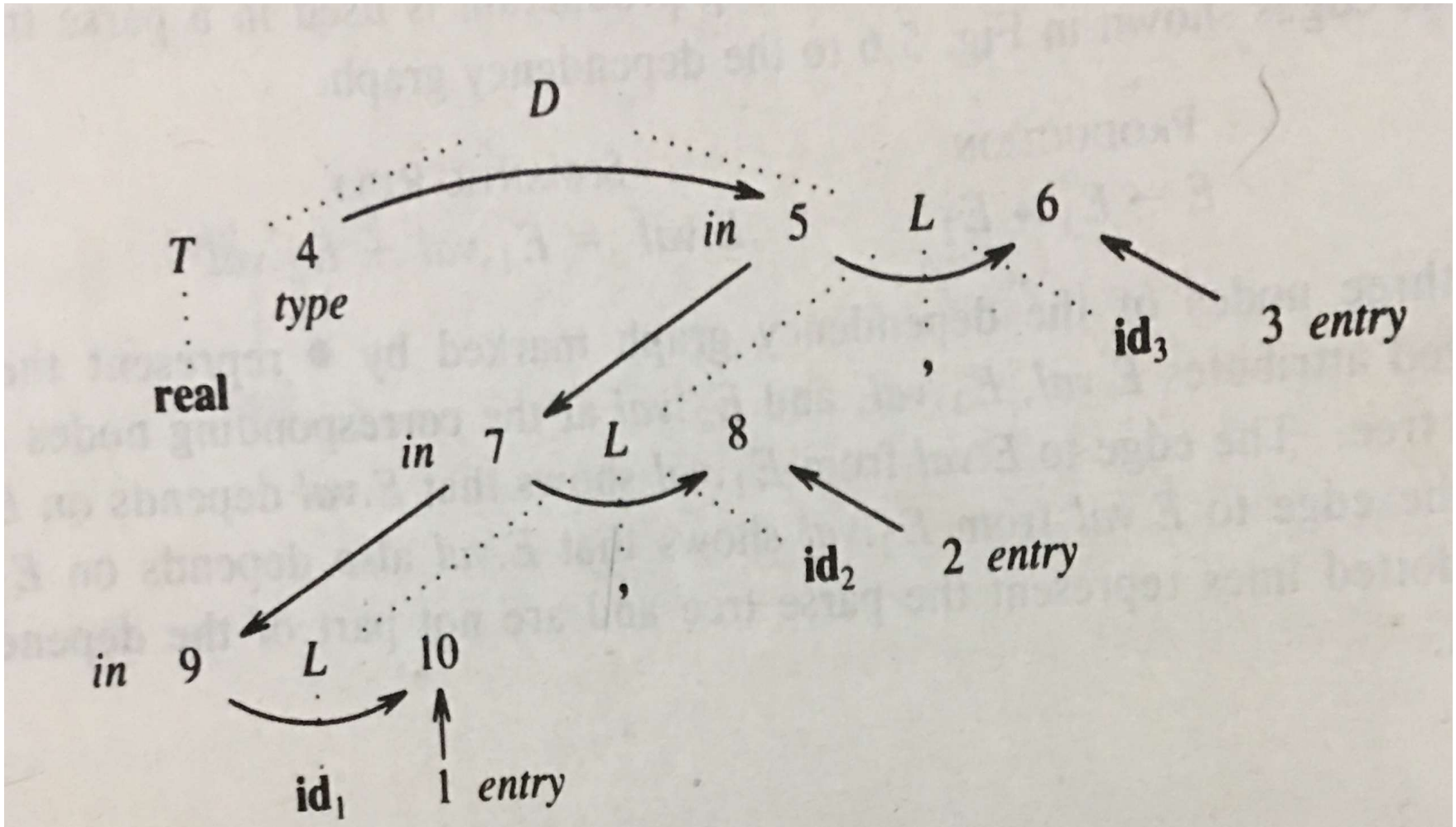


Fig. 5.7. Dependency graph for parse tree of Fig. 5.5.

Bottom-up evaluation of s-attributed definitions

- S-attributed definitions : Syntax-directed definitions with only synthesized attributes
- Synthesized attributes can be evaluated by a bottom-up parser as the input is being parsed
- The parser can keep the values of the synthesized attributes associated with the grammar symbols on its stack

- Whenever a reduction is made , the values of the new synthesized attributes are computed from the attributes appearing on the stack for the grammar symbols on the right side of the reducing production
- This section shows how the parser stack can be extended to hold the values of the synthesized attributes

Synthesized attributes on the Parser stack

- A bottom-up parser uses a stack to hold extra information about sub-trees that have been parsed
- We use extra fields in the parser stack to hold the values of the synthesized attributes
- Fig 5.15 shows an eg of a parser stack with space for one attribute value

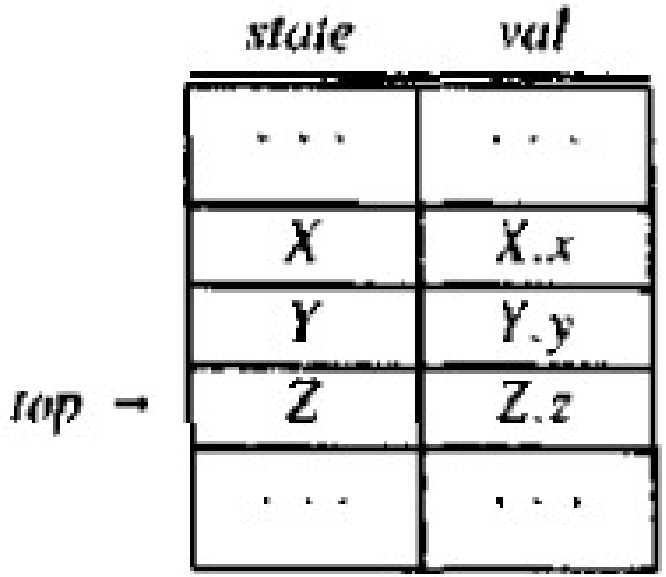


Fig. 5.15. Parser stack with a field for synthesized attributes.

- As in figure, the stack is implemented by a pair of arrays state and val
- Each state entry is a pointer to an LR(1) parsing table
- If the ith state symbol is A, then val[i] will hold the value of the attribute associated with the parse tree node corresponding to this A
- The current top of the stack is indicated by the pointer top
- We assume that synthesized attributes are evaluated just before each reduction

- Suppose the semantic rule A.a = f(X.x, Y.y, Z.z)is associated with the production A -> XYZ
- Before XYZ is reduced to A, the value of the attribute Z.z is in val[top], that of Y.y is in val[top-1] and that of X.x is in val[top-2]
- If a symbol has no attribute, then the corresponding entry in the val array is undefined
- After the reduction, top is decremented by 2, the state covering A is put in state[top]

Example 5.10

- Consider the syntax-directed definition of the desk calculator in fig 5.2. The synthesized attributes in the annotated parse tree of fig 5.3 can be evaluated by an LR parser during a bottom-up parse of the input line 3*5+4n
- To evaluate the attributes, we modify the parser to execute the code fragments shown in fig 5.16just before making the appropriate reduction

Fig 5.16

PRODUCTION	CODE FRAGMENT
$L \rightarrow E$ **n**	*print* (*val* [*top*])
$E \rightarrow E_1 + T$	*val* [*ntop*] := *val* [*top* − 2] + *val* [*top*]
$E \rightarrow T$	
$T \rightarrow T_1 * F$	*val* [*ntop*] := *val* [*top* − 2] × *val* [*top*]
$T \rightarrow F$	
$F \rightarrow (E)$	*val* [*ntop*] := *val* [*top* − 1]
$F \rightarrow$ **digit**	

Fig. 5.16. Implementation of a desk calculator with an LR parser.

- The code fragments do not show how the variables top and ntop are managed
- When a production with r symbols on the right side is reduced the value of ntop is set to top-r+1
- After each code fragment is executed, top is set to ntop
- Fig 5.17 shows the sequence of moves made by the parser on input 3*5+4n

INPUT	*state*	*val*	PRODUCTION USED
3*5+4n	_	_	
*5+4n	3	3	
*5+4n	*F*	3	*F* → **digit**
*5+4n	*T*	3	*T* → *F*
5+4n	*T* *	3 _	
+4n	*T* * 5	3 _ 5	
+4n	*T* * *F*	3 _ 5	*F* → **digit**
+4n	*T*	15	*T* → *T* * *F*
+4n	*E*	15	*E* → *T*
4n	*E* +	15 _	
n	*E* + 4	15 _ 4	
n	*E* + *F*	15 _ 4	*F* → **digit**
n	*E* + *T*	15 _ 4	*T* → *F*
n	*E*	19	*E* → *E* + *T*
	E n	19 _	
	L	19	*L* → *E* n

Fig. 5.17. Moves made by translator on input 3*5+4n.

L-ATTRIBUTED DEFINITION

- When translation takes place during parsing, the order of evaluation of attributes is linked to the order in which nodes of a parse tree are "created" by the parsing method
- A natural order that characterizes many top down and bottom-up translation methods is the one obtained by applying the procedure dfsvisit in fig 5.8 to the root of a parse tree
- We call this evaluation order the depth-first order

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

Fig. 5.18. Depth-first evaluation order for attributes in a parse tree.

- A class of syntax-directed definitions, called L-attributed definitions, whose attributes can always be evaluated in depth-first order.
- The L *is for "left," because attribute information appears to flow from left to* right

L-Attributed Definition

- A syntax-directed definition is *L-attributed if each inherited attribute of Xj,* l ≤j ≤n, on the right side of A→X l X2 . ..Xn depends only on

1. The attributes of the symbols X l, X 2 , . . . , Xj- l to the left of Xj, in the production and

2. The inherited attributes of A.

- Note that every S-attributed definition is L-attributed, because the restrictions (1) and (2) apply only to inherited attributes

Translation schemes

- 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

Example 5.12. Here is a simple translation scheme that maps infix expressions with addition and subtraction into corresponding postfix expressions

$E \rightarrow T\ R$

$R \rightarrow addop\ T\ \{print(addop.lexeme)\}R1 \mid \varepsilon$

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

Figure 5.20 shows the parse tree for the input 9-5+2 with each semantic action attached as the appropriate child at the node corresponding to the leftside of their production

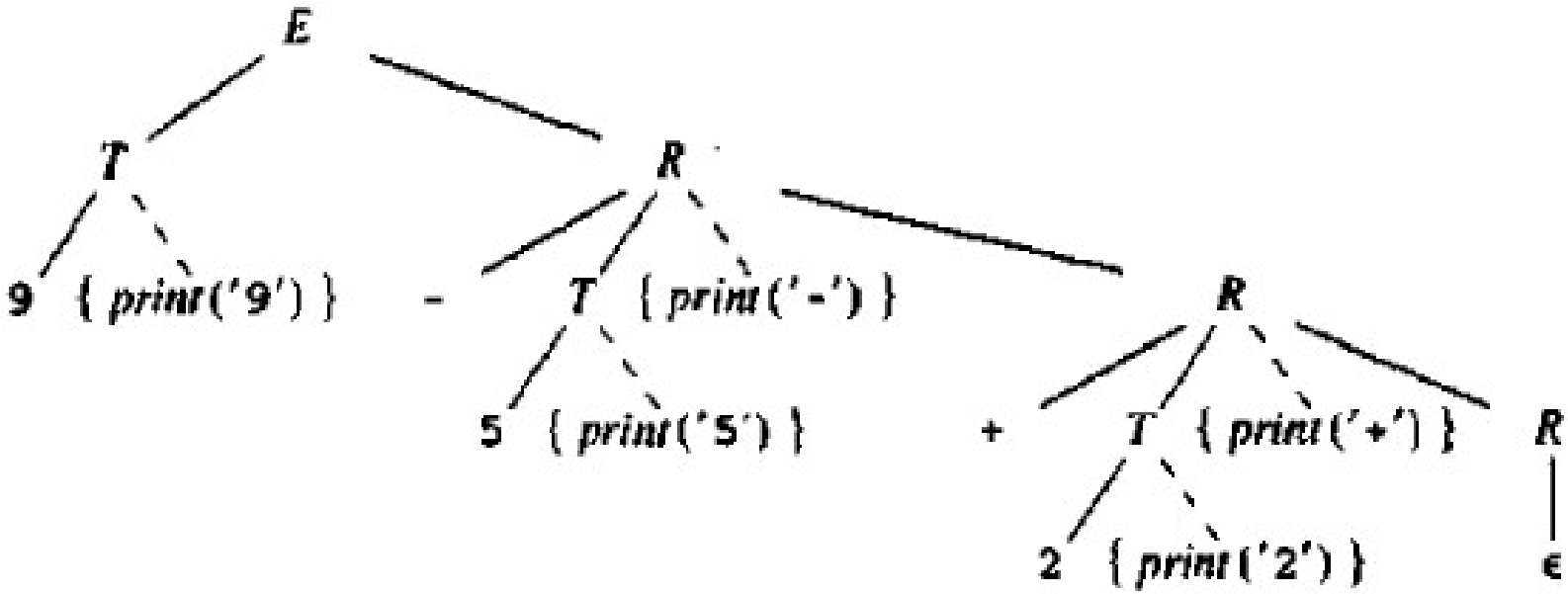


Fig. 5.20. Parse tree for 9-5+2 showing actions.

- With synthesized attributes, the translation rules are placed at the end of the productions
- **For example, the production and semantic rule**

PRODUCTION	SEMANTIC RULE
$T \rightarrow T_1 * F$	$T.val := T_1.val \times F.val$

yield the following production and semantic action

$T \rightarrow T_1 * F \ \{ T.val := T_1.val \times F.val \}$

- If we have both inherited and synthesized attributes, we must be more careful

1. An inherited attribute for a symbol on the right side of a production must be computed in an action before the symbol
2. An action must not refer to a synthesized attribute of a symbol to the right of the action
3. A synthesized attribute for the non-terminal on the left can only be computer after all attribute it references have been computed. The action computing such attributes can usually be placed at the end of the right side of the production

Module IV – Part II

Intermediate Code Generation - Intermediate languages, Graphical representations, Three-Address code, Quadruples, Triples

Source Language Issues

- Suppose that a program is made up of procedures, as in Pascal
- This section distinguishes between the source text of a procedure and its activations at run time

Source language issues

- Procedures
- Activation trees
- Control stacks
- The scope of a declaration
- Bindings of names

Storage organization

- Subdivision of run-time memory
- Activation records
- Compile-time layout of local data

Storage allocation strategies

- Static allocation
- Stack allocation
- Heap allocation

Intermediate Code Generation

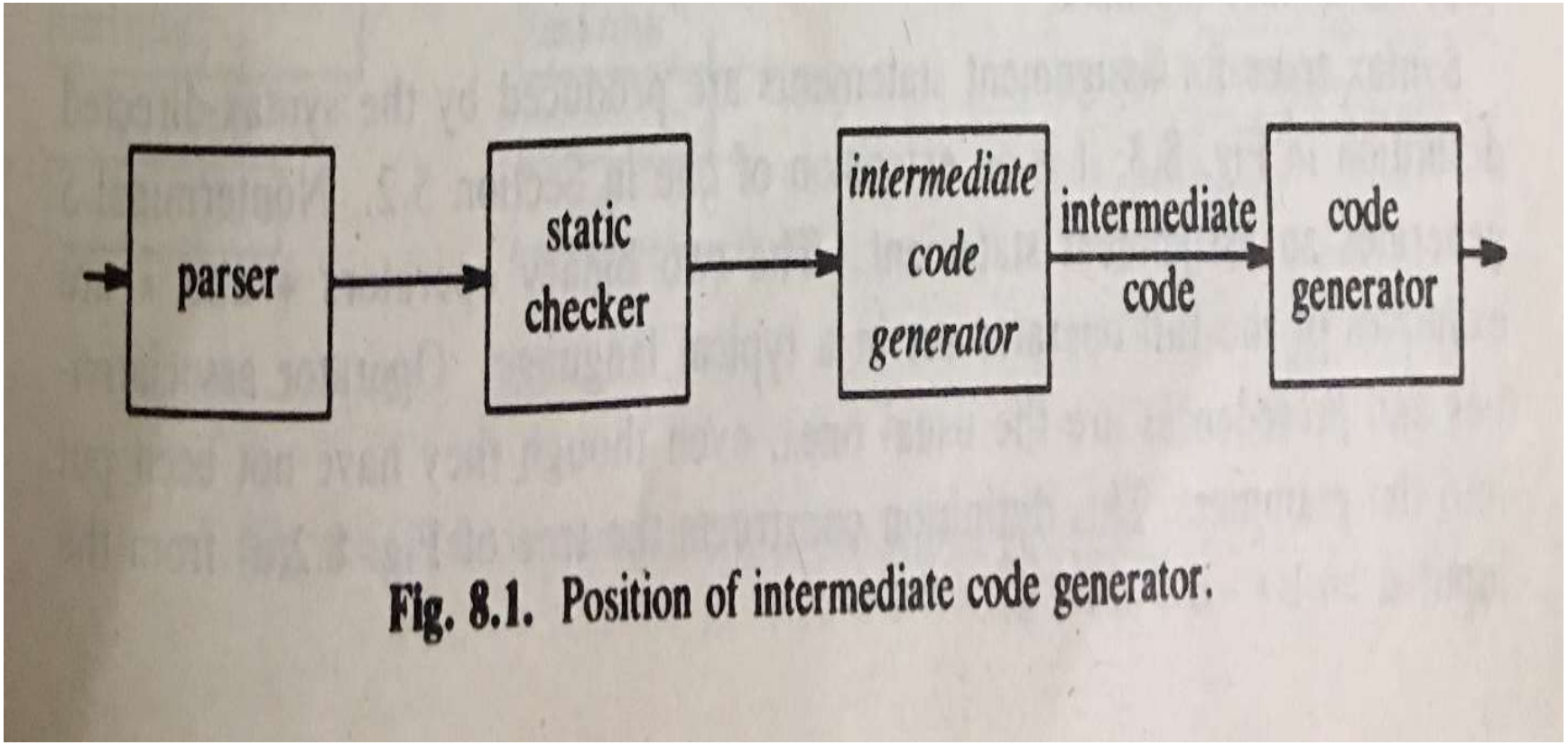


Fig. 8.1. Position of intermediate code generator.

Intermediate Languages

- Graphical representations
  - Syntax trees
  - DAGs
- Linear representations
  - Postfix notation
  - Three address code

Graphical Representations

- Two representations
- Syntax tree
  - Depicts the natural hierarchical structure of a source program
- Directed Acyclic Graphs
  - A dag gives the same information in a more compact way because common sub expressions are eliminated

Draw a syntax tree and dag for the assignment statement a:=b * -c + b * -c

- A syntax tree and dag for the assignment statement a:=b * -c + b * -c appear in fig 8.2

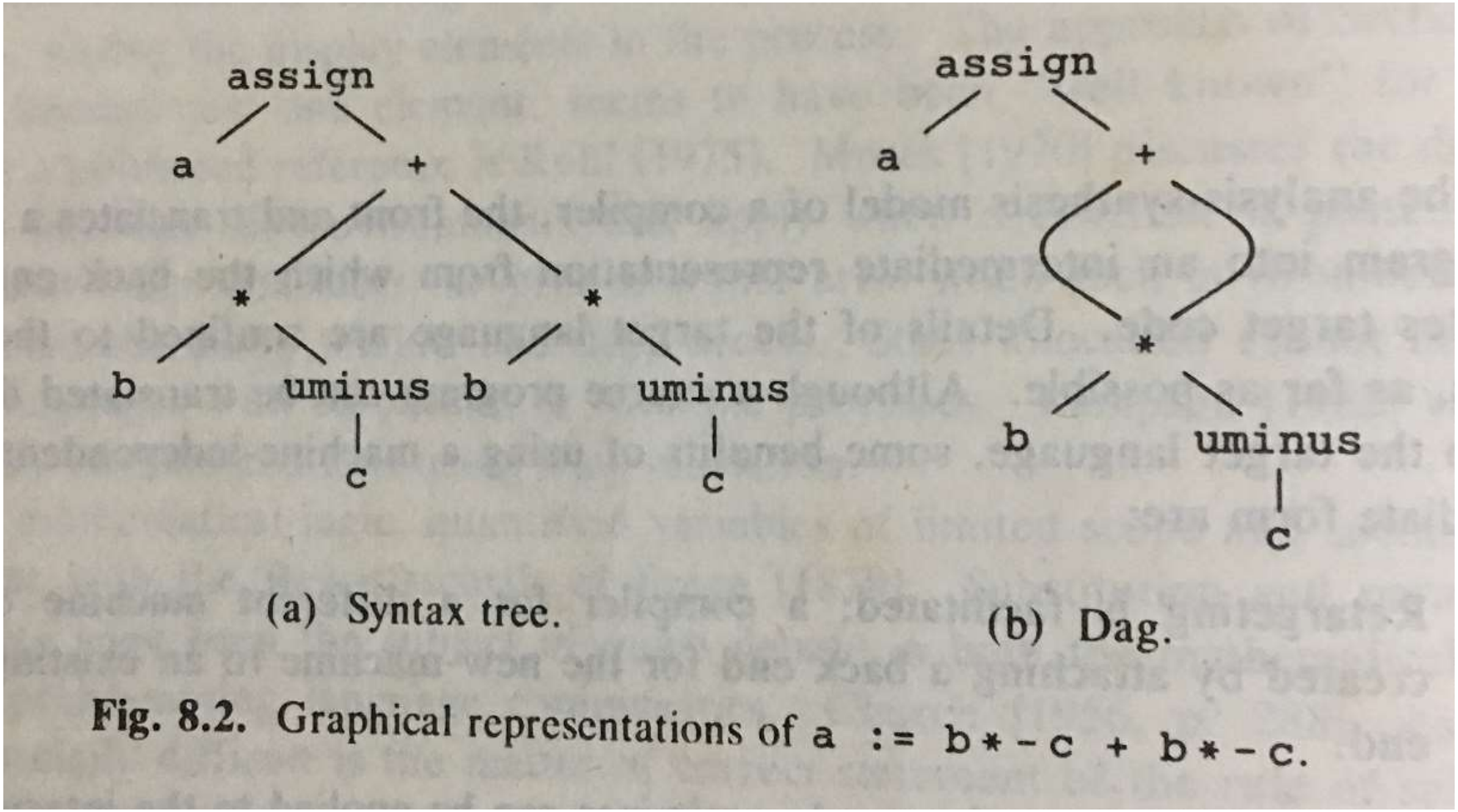


Fig. 8.2. Graphical representations of a := b * -c + b * -c.

- Postfix notation is a linearized representation of a syntax tree
- It is a list of the nodes of the tree in which a node appears immediately after its children (???)
- The postfix notation of the syntax tree in fig 8.2 a is

  a b c uminus * b c uminus * + assign

Three-Address Code

- A sequence of statements of the general form

  x := y op z

- Where x,y and z are names, constants or compiler-generated temporaries
- op stands for any operator, such as fixed- or floating-point arithmetic operator or a logical
- Thus a source language expression like x+y*z might be translated into a sequence

  t1 := y *z

  t2 := x + t1 where t1 and t2 are compiler generated temporaries
- The syntax tree and dag in fig 8.2 are represented by the following three address code

```
t1 := - c
t2 := b * t1
t3 := - c
t4 := b * t3
t5 := t2 + t4
a  := t5
```

(a) Code for the syntax tree.

```
t1 := - c
t2 := b * t1
t5 := t2 + t2
a  := t5
```

(b) Code for the dag.

- The reason for the term “three-address code”
  - Each statement usually contains three addresses, two for the operands and one for the result

Types of Three-Address Statements

1. Assignment statements of the form x:=y op z, where op is a binary arithmetic or logical operation
2. Assignment instructions of the form x := op y, where op is a unary operation
3. Copy statements of the form x:=y
4. The unconditional jump goto L
5. Conditional jumps such as if relop y goto L
6. param x and call p for procedure calls and return y
7. Indexed assignments of the form x:=y[i] and x[i]:=y
8. Address and pointer assignments of the form x:=&y, x:=*y and *x :=y

Implementations of Three-Address Statements

- A three-address statement is an abstract form of intermediate code
- In a compiler, these statements can be implemented as records with fields for the operator and the operands
- Three such representations are quadruples, triples and indirect triples

Quadruples

- A quadruple is a record structure with four fields, which we call op, arg1, arg2 and result
- The op field contains an internal code for the operator
- The three-address statement x:= y op z is represented by placing y in arg1, z in arg2 and x in result
- Statements like x:=-y or x:=y do not use arg2
- Operators like param use neither arg2 nor result
- Conditional and unconditional jumps put the target label in result
- Quadruple for a:=b*-c+b*-c

	op	*arg* 1	*arg* 2	*result*
(0)	uminus	c		t_1
(1)	*	b	t_1	t_2
(2)	uminus	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	:=	t_5		a

(a) Quadruples

Triples

- To avoid entering temporary names into the symbol table, we might refer to the position of the statement that computes it
- If we do so, three address statements can be represented by records with only three fields – op, arg1, arg2
- arg1 and arg2 are either pointers to the symbol table (for programmer defined names or constants) or pointers into the triple structure (for temporary values)
- Since only three fields are used, this format is called triples
- Parenthesized numbers represent pointers into the triple structure while symbol-table pointers are represented by the names themselves

	op	*arg* 1	*arg* 2
(0)	`uminus`	`c`	
(1)	`*`	`b`	(0)
(2)	`uminus`	`c`	
(3)	`*`	`b`	(2)
(4)	`+`	(1)	(3)
(5)	`assign`	`a`	(4)

(b) Triples

- A ternary operation like x[i]:=y requires two entries in the triple structure as shown in fig (a) while x:=y[i] is as shown in fig (b)

	op	*arg* 1	*arg* 2
(0)	[ ]=	x	i
(1)	assign	(0)	y

(a) x[i] := y

	op	*arg* 1	*arg* 2
(0)	`=[]`	`y`	`i`
(1)	`assign`	`x`	(0)

(b) `x := y[i]`

Indirect triples

- Listing of pointers to triples rather than listing the triples themselves
- For eg: we might use an array "statement" to list the pointers to the triples in the desired order

	statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

	op	*arg* 1	*arg* 2
(14)	uminus	c	
(15)	*	b	(14)
(16)	uminus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	assign	a	(18)