UNIT –III

More Powerful LR parser (LR1,LALR) Using Armigers Grammars Equal Recovery in Lr parser Syntax Directed Transactions Definition, Evolution order of SDTS Application of SDTS. Syntax Directed Translation Schemes.

UNIT -3

3.1 CANONICAL LR PARSING

CLR refers to canonical lookahead. CLR parsing use the canonical collection of LR (1) items to build the CLR (1) parsing table. CLR (1) parsing table produces the more number of states as compare to the SLR (1) parsing.

In the CLR (1), we place the reduce node only in the lookahead symbols.

Various steps involved in the CLR (1) Parsing:

1) For the given input string write a context free grammar
2) Check the ambiguity of the grammar
3) Add Augment production in the given grammar
4) Create Canonical collection of LR (0) items
5) Draw a data flow diagram (DFA)
6) Construct a CLR (1) parsing table

In the SLR method we were working with LR(0)) items. In CLR parsing we will be using LR(1) items. LR(k) item is defined to be an item using lookaheads of length k. So ,the LR(1) item is comprised of two parts : the LR(0) item and the lookahead associated with the item. The look ahead is used to determine that where we place the final item. The look ahead always add $ symbol for the argument production.

LR(1) parsers are more powerful parser.
for LR(1) items we modify the Closure and GOTO function.

Closure Operation

Closure(I)

repeat

for (each item [A -> ?.B?, a] in I)

for (each production B -> ? in G')

for (each terminal b in FIRST(?a))

add [B -> .? , b] to set I;

until no more items are added to I;

return I;

Goto Operation

Goto(I, X)

Initialise J to be the empty set;

for (each item A -> ?.X?, a] in I)

Add item A -> ?X.?, a] to se J; /* move the dot one step */

return Closure(J); /* apply closure to the set */

LR(1) items

Void items(G')

Initialise C to { closure ({[S' -> .S, $]})};

Repeat

For (each set of items I in C)

For (each grammar symbol X)

if(GOTO(I, X) is not empty and not in C)

Add GOTO(I, X) to C;

Until no new set of items are added to C;

3.2 ALGORITHM FOR CONSTRUCTION OF THE CANONICAL LR PARSING TABLE

Input: grammar G'
Output: canonical LR parsing table functions action and goto

1. Construct C = {I0, I1 , ..., In} the collection of sets of LR(1) items for G'.State i is constructed from Ii.
2. if [*A* -> a.ab, b>] is in Ii and goto(Ii, a) = Ij, then set action[i, a] to "shift j". Here a must be a terminal.
3. if [*A* -> a., a] is in Ii, then set action[i, a] to "reduce *A* -> a" for all a in FOLLOW(*A*). Here *A* may *not* be *S*'.
4. if [*S*' -> *S*.] is in Ii, then set action[i, $] to "accept"
5. If any conflicting actions are generated by these rules, the grammar is not LR(1) and the algorithm fails to produce a parser.
6. The goto transitions for state i are constructed for all *nonterminal*s *A* using the rule: If goto(Ii, A)= Ij, then goto[i, *A*] = j.
7. All entries not defined by rules 2 and 3 are made "error".
8. The inital state of the parser is the one constructed from the set of items containing [*S*' -> .S, $].

Example,

Consider the following grammar,

S‟->S
S->CC
C->cC
C->d

Sets of LR(1) items

I0: S‟->.S,$
S->.CC,$
C->.Cc,c/d
C->.d,c/d

I1: S‟->S.,$

I2: S->C.C,$
C->.Cc,$
C->.d,$

I3: C->c.C,c/d
C->.Cc,c/d
C->.d,c/d

I4: C->d.,c/d

I5: S->CC.,$

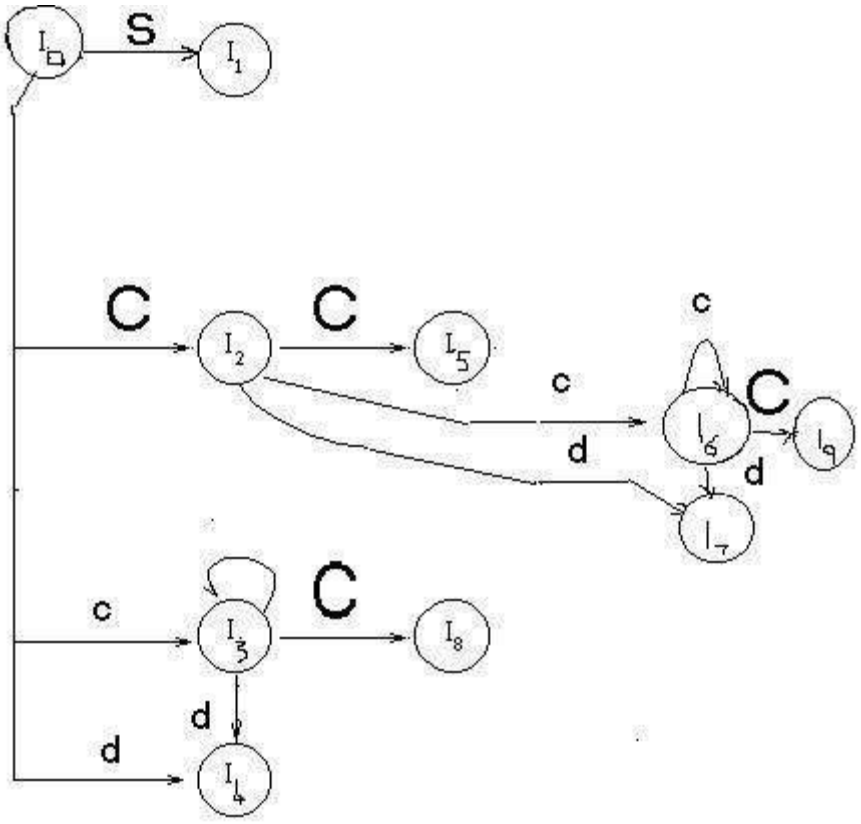
I6: C->c.C,$
C->.cC,$
C->.d,$

I7: C->d.,$

I8: C->cC.,c/d

I9: C->cC.,$

Here is what the corresponding DFA looks like



Parsing Table:state	c	d	$	S	C
0	S3	S4		1	2
1			acc		
2	S6	S7			5
3	S3	S4			8
4	R3	R3			
5			R1		
6	S6	S7			9
7			R3		
8	R2	R2			
9			R2		

3.3.LALR PARSER:

We begin with two observations. First, some of the states generated for LR(1) parsing have the same set of core (or first) components and differ only in their second component, the lookahead symbol. Our intuition is that we should be able to merge these states and reduce the number of states we have, getting close to the number of states that would be generated for LR(0) parsing. This observation suggests a hybrid approach: We can construct the canonical LR(1) sets of items and then look for sets of items having the same core. We merge these sets with common cores into one set of items. The merging of states with common cores can never produce a shift/reduce conflict that was not present in one of the original states because shift actions depend only on the core, not the lookahead. But it is possible for the merger to produce a reduce/reduce conflict.

Our second observation is that we are really only interested in the lookahead symbol in places where there is a problem. So our next thought is to take the LR(0) set of items and add lookaheads only where they are needed. This leads to a more efficient, but much more complicated method.

3.4 ALGORITHM FOR EASY CONSTRUCTION OF AN LALR TABLE

Input: G'
Output: LALR parsing table functions with action and goto for G'.

Method:

1. Construct C = {I0, I1 , ..., In} the collection of sets of LR(1) items for G'.
2. For each core present among the set of LR(1) items, find all sets having that core and replace these sets by the union.
3. Let C' = {J0, J1 , ..., Jm} be the resulting sets of LR(1) items. The parsing actions for state i are constructed from Ji in the same manner as in the construction of the canonical LR parsing table.
4. If there is a conflict, the grammar is not LALR(1) and the algorithm fails.
5. The goto table is constructed as follows: If J is the union of one or more sets of LR(1) items, that is, J = I0U I1 U ... U Ik, then the cores of goto(I0, X), goto(I1, X), ..., goto(Ik, X) are the same, since I0, I1 , ..., Ik all have the same core. Let K be the union of all sets of items having the same core asgoto(I1, X).

6. Then goto(J, X) = K.

Consider the above example,

I3 & I6 can be replaced by their union I36:C->c.C,c/d/$

C->.Cc,C/D/$

C->.d,c/d/$

I47:C->d.,c/d/$

I89:C->Cc.,c/d/$

Parsing Table

state	c	d	$	S	C
0	S36	S47		1	2
1			Accept		
2	S36	S47			5
36	S36	S47			89
47	R3	R3			
5			R1		
89	R2	R2	R2		

3.5HANDLING ERRORS

The LALR parser may continue to do reductions after the LR parser would have spotted an error, but the LALR parser will never do a shift after the point the LR parser would have discovered the error and will eventually find the error.

3.6 DANGLING ELSE

The dangling else is a problem in computer programming in which an optional else clause in an If–then(–else) statement results in nested conditionals being ambiguous. Formally, the context-free grammar of the language is ambiguous, meaning there is more than one correct parse tree.

In many programming languages one may write conditionally executed code in two forms: the if-then form, and the if-then-else form – the else clause is optional:

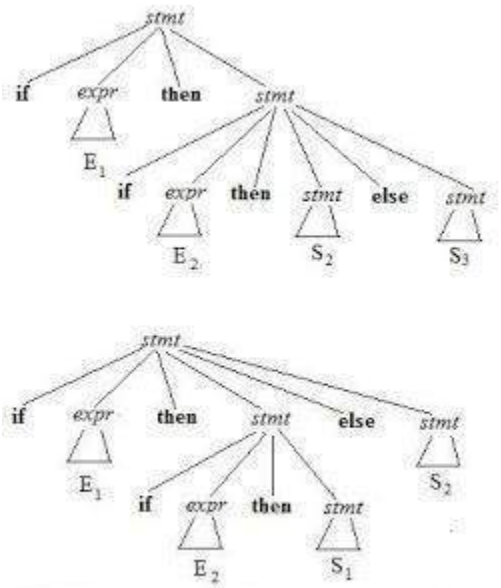


Fig 2.4 Two parse trees for an ambiguous sentence

Consider the grammar:

S ::= E $

E ::= E + E

| E * E

| (E)

| id

| num

and four of its LALR(1) states:

I0: S ::= . E $?

E ::= . E + E +*$

E ::= . E * E +*$

E ::= . (E) +*$

E ::= . id +*$

E ::= . num +*$

I1: S ::= E . $?

E ::= E . + E +*$

E ::= E . * E +*$

I2: E ::= E * . E +*$

E ::= . E + E +*$

E ::= . E * E +*$

E ::= . (E) +*$

E ::= . id +*$

E ::= . num +*$

I3: E ::= E * E . +*$

E ::= E . + E +*$

E ::= E . * E +*$

E ::= E . * E +*$

Here we have a shift-reduce error. Consider the first two items in I3. If we have a*b+c and we parsed a*b, do we reduce using E ::= E * E or do we shift more symbols? In the former case we get a parse tree (a*b)+c; in the latter case we get a*(b+c). To resolve this conflict, we can specify that * has higher precedence than +. The precedence of a grammar production is equal to the precedence of the rightmost token at the rhs of the production. For example, the precedence of the production E ::= E * E is equal to the precedence of the operator *, the precedence of the production E ::= (E) is equal to the precedence of the token), and the precedence of the production E ::= if E then E else E is equal to the precedence of the token else. The idea is that if the look ahead has higher precedence than the production currently used, we shift. For example, if we are parsing E + E using the production rule E ::= E + E and the look ahead is *, we shift *. If the look ahead has the same precedence as that of the current production and is left associative, we reduce, otherwise we shift. The above grammar is valid if we define the precedence and associativity of all the operators. Thus, it is very important when you write a parser using CUP or any other LALR(1) parser generator to specify associativities and precedence‟s for most tokens (especially for those used as operators). Note: you can explicitly define the precedence of a rule in CUP using the %prec directive:

E ::= MINUS E %prec UMINUS

where UMINUS is a pseudo-token that has higher precedence than TIMES, MINUS etc, so that -1*2 is equal to (-1)*2, not to -(1*2).

Another thing we can do when specifying an LALR(1) grammar for a parser generator is error recovery. All the entries in the ACTION and GOTO tables that have no content correspond to syntax errors. The simplest thing to do in case of error is to report it and stop the parsing. But we would like to continue parsing finding more errors. This is called *error recovery*. Consider the grammar:

```
S ::= L = E ;
   | { SL }
   ; | error ;
SL ::= S ; |
    SL S ;
```

The special token error indicates to the parser what to do in case of invalid syntax for S (an invalid statement). In this case, it reads all the tokens from the input stream until it finds the first semicolon. The way the parser handles this is to first push an error state in the stack. In case of an error, the parser pops out elements from the stack until it finds an error state where it can proceed. Then it discards tokens from the input until a restart is possible. Inserting error handling productions in the proper places in a grammar to do good error recovery is considered very hard.

3.7 LR ERROR RECOVERY

An LR parser will detect an error when it consults the parsing action table and find a blank or error entry. Errors are never detected by consulting the goto table. An LR parser will detect an error as soon as there is no valid continuation for the portion of the input thus far

scanned. A canonical LR parser will not make even a single reduction before announcing the error. SLR and LALR parsers may make several reductions before detecting an error, but they will never shift an erroneous input symbol onto the stack.

3.7.1 PANIC-MODE ERROR RECOVERY

We can implement panic-mode error recovery by scanning down the stack until a state s with a goto on a particular nonterminal A is found. Zero or more input symbols are then discarded until a symbol a is found that can legitimately follow A. The parser then stacks the state GOTO(s, A) and resumes normal parsing. The situation might exist where there is more than one choice for the nonterminal A. Normally these would be nonterminals representing major program pieces, e.g. an expression, a statement, or a block. For example, if A is the nonterminal stmt, a might be semicolon or }, which marks the end of a statement sequence. This method of error recovery attempts to eliminate the phrase containing the syntactic error. The parser determines that a string derivable from A contains an error. Part of that string has already been processed, and the result of this processing is a sequence of states on top of the stack. The remainder of the string is still in the input, and the parser attempts to skip over the remainder of this string by looking for a symbol on the input that can legitimately follow A. By removing states from the stack, skipping over the input, and pushing GOTO(s, A) on the stack, the parser pretends that if has found an instance of A and resumes normal parsing.

3.7.2 PHRASE-LEVEL RECOVERY

Phrase-level recovery is implemented by examining each error entry in the LR action table and deciding on the basis of language usage the most likely programmer error that would give rise to that error. An appropriate recovery procedure can then be constructed; presumably the top of the stack and/or first input symbol would be modified in a way deemed appropriate for each error entry. In designing specific error-handling routines for an LR parser, we can fill in each blank entry in the action field with a pointer to an error routine that will take the appropriate action selected by the compiler designer.

The actions may include insertion or deletion of symbols from the stack or the input or both, or alteration and transposition of input symbols. We must make our choices so that the LR parser will not get into an infinite loop. A safe strategy will assure that at least one input symbol will be removed or shifted eventually, or that the stack will eventually shrink if the end of the input has been reached. Popping a stack state that covers a non terminal should be avoided, because this modification eliminates from the stack a construct that has already been successfully parsed.

Syntax Directed Translations

We associate information with a language construct by attaching attributes to the grammar symbol(s) representing the construct, A syntax-directed definition specifies the values of attributes by associating semantic rules with the grammar productions. For example, an infix-to-postfix translator might have a production and rule

PRODUCTION	SEMANTIC RULE
$E \rightarrow Ei + T$	$E.code = Ei.code \parallel T.code \parallel '+'$

This production has two nonterminals, E and T; the subscript in E1 distinguishes the occurrence of E in the production body from the occurrence of E as the head. Both E and T have a string-valued attribute code. The semantic rule specifies that the string E.code is formed by concatenating Ei.code, T.code, and the character '+'. While the rule makes it explicit that the translation of E is built up from the translations of E1, T, and '+', it may be inefficient to implement the translation directly by manipulating strings.

a syntax-directed translation scheme embeds program fragments called semantic actions within production bodies
There are two notations for attaching semantic rules:

1. **Syntax Directed Definitions.** High-level specification hiding many implementation details (also called **Attribute Grammars**).

2. **Translation Schemes.** More implementation oriented: Indicate the order in which semantic rules are to be evaluated.

Syntax Directed Definitions

Syntax Directed Definitions are a generalization of context-free grammars 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 (e.g.,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.

We distinguish between 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 Definitions: An Example

Let us consider the Grammar for arithmetic expressions. The Syntax Directed Definition associates to each non terminal a synthesized attribute called *val*.

PRODUCTION	SEMANTIC RULE
PRODUCTION	SEMANTIC RULE
E -> Ei + T	E.code = Ei.code ‖ T.code ‖ '+'
$F \rightarrow (E)$	$F.val := E.val$
$F \rightarrow$ digit	$F.val :=$ digit.*lexval*

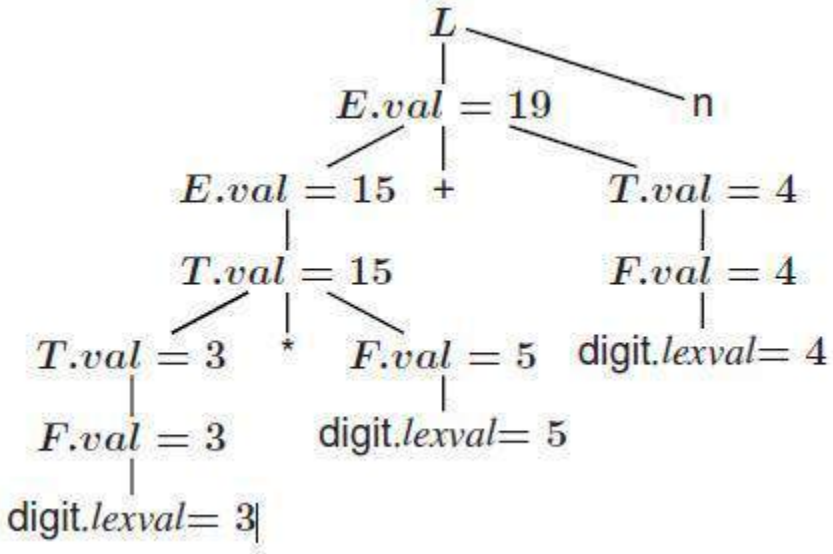
SDD of a simple desk calculator

S-ATTRIBUTED DEFINITIONS

Definition. An **S-Attributed Definition** is a Syntax Directed Definition that uses only synthesized attributes.

• **Evaluation Order.** Semantic rules in a S-Attributed Definition can be evaluated by a bottom-up, or PostOrder, traversal of the parse-tree.

• **Example.** The above arithmetic grammar is an example of an S-Attributed Definition. The annotated parse-tree for the input 3*5+4n is:



L-attributed definition

Definition: A SDD its *L-attributed* if each inherited attribute of Xi in the RHS of A ! X1 : :Xn depends only on

1. attributes of X1;X2; : : : ;Xi 1 (symbols to the left of Xi in the RHS)
2. inherited attributes of A.

Restrictions for translation schemes:

1. Inherited attribute of Xi must be computed by an action before Xi.
2. An action must not refer to synthesized attribute of any symbol to the right of that action.
3. Synthesized attribute for A can only be computed after all attributes it references have been completed (usually at end of RHS).

Evaluation order of SDTS

"Dependency graphs" are a useful tool for determining an evaluation order for the attribute instances in a given parse tree. While an annotated parse tree shows the values of attributes, a dependency graph helps us determine how those values can be computed.

1 Dependency Graphs

A *dependency graph* depicts the flow of information among the attribute in-stances in a particular parse tree; an edge from one attribute instance to an-other means that the value of the first is needed to compute the second. Edges express constraints implied by the semantic rules. In more detail:

Suppose that a semantic rule associated with a production *p* defines the value of inherited attribute *B.c* in terms of the value of *X.a.* Then, the dependency graph has an edge from *X.a* to *B.c.* For each node *N* labeled *B* that corresponds to an occurrence of this *B* in the body of production *p,* create an edge to attribute c at *N* from the attribute *a* at the node *M* that corresponds to this occurrence of *X.* Note that *M* could be either the parent or a sibling of *N.*

Since a node ***N*** can have several children labeled ***X,*** we again assume that subscripts distinguish among uses of the same symbol at different places in the production.

Example: Consider the following production and rule:

PRODUCTION	SEMANTIC RULE
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$

At every node *N* labeled *E*, with children corresponding to the body of this production, the synthesized attribute *val* at *N* is computed using the values of *val* at the two children, labeled *E* and *T*. Thus, a portion of the dependency graph for every parse tree in which this production is used looks like Fig. 5.6. As a convention, we shall show the parse tree edges as dotted lines, while the edges of the dependency graph are solid.

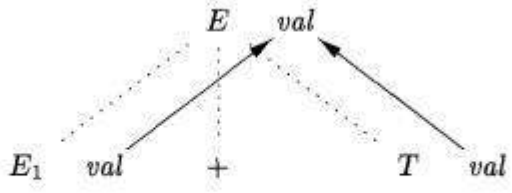


Figure 5.6: $E.val$ is synthesized from $E_1.val$ and $E_2.val$

2. Ordering the Evaluation of Attributes

The dependency graph characterizes the possible orders in which we can evalu-ate the attributes at the various nodes of a parse tree. If the dependency graph has an edge from node M to node N, then the attribute corresponding to M must be evaluated before the attribute of N. Thus, the only allowable orders of evaluation are those sequences of nodes N1, N2,... ,Nk such that if there is an edge of the dependency graph 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; that is, there is no way to evaluate the SDD on this parse tree. If there are no cycles, however, then there is always at least one topological sort

3. S-Attributed Definitions

An SDD is *S-attributed* if every attribute is synthesized. When an SDD is S-attributed, we can evaluate its attributes in any bottom-up order of the nodes of the parse tree. It is often especially simple to evaluate the attributes by performing a postorder traversal of the parse tree and evaluating the attributes at a node *N* when the traversal leaves *N* for the last time.

S-attributed definitions can be implemented during bottom-up parsing, since a bottom-up parse corresponds to a postorder traversal. Specifically, postorder corresponds exactly to the order in which an LR parser reduces a production body to its head.

4 L-Attributed Definitions

The idea behind this class is that, between the attributes associated with a production body, dependency-graph edges can go from left to right, but not from right to left (hence "L-attributed"). More precisely, each attribute must be either

1. Synthesized, or
2. Inherited, but with the rules limited as follows. Suppose that there is a production A -> X1 X2 ..... Xn, and that there is an inherited attribute Xi.a computed by a rule associated with this production.

Then the rule may use only:
Inherited attributes associated with the head A.
Either inherited or synthesized attributes associated with the occurrences of symbols X1, X2,... , X(i-1) 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 X i

Application of SDTS

1 Construction of Syntax Trees
2 The Structure of a Type

The main application is the construction of syntax trees. Since some compilers use syntax trees as an intermediate representation, a common form of SDD turns its input string into a tree. To complete the translation to intermediate code, the compiler may then walk the syntax tree, using another set of rules that are in effect an SDD on the syntax tree rather than the parse tree.

1 Construction of Syntax Trees

Each node in a syntax tree represents a construct; the children of the node represent the meaningful components of the construct. A syntax-tree node representing an expression *E1 + E2* has label + and two children representing the subexpressions *E1* and E_2

implement the nodes of a syntax tree by objects with a suitable number of fields. Each object will have an *op* field that is the label of the node.
The objects will have additional fields as follows:

• If the node is a leaf, an additional field holds the lexical value for the leaf. A constructor function Leaf (op, val) creates a leaf object. Alternatively, if nodes are viewed as records, then Leaf returns a pointer to a new record for a leaf.

• If the node is an interior node, there are as many additional fields as the node has children in the syntax tree. A constructor function Node takes two or more arguments: Node(op,ci,c2,... ,ck) creates an object with first field op and k additional fields for the k children c1,... , .

Example

	PRODUCTION	SEMANTIC RULES
1)	$E \to E_1 + T$	$E.node = \textbf{new}\ Node('+', E_1.node, T.node)$
2)	$E \to E_1 - T$	$E.node = \textbf{new}\ Node('-', E_1.node, T.node)$
3)	$E \to T$	$E.node = T.node$
4)	$T \to (\ E\ )$	$T.node = E.node$
5)	$T \to \textbf{id}$	$T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$
6)	$T \to \textbf{num}$	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$

Figure 5.10: Constructing syntax trees for simple expressions

Figure 5.1 1 shows the construction of a syntax tree for the input *a* — 4 + *c*. The nodes of the syntax tree are shown as records, with the *op* field first. Syntax-tree edges are now shown as solid lines. The underlying parse tree, which need not actually be constructed, is shown with dotted edges. The

third type of line, shown dashed, represents the values of *E.node* and *T-node;* each line points to the appropriate syntax-tree node.

.

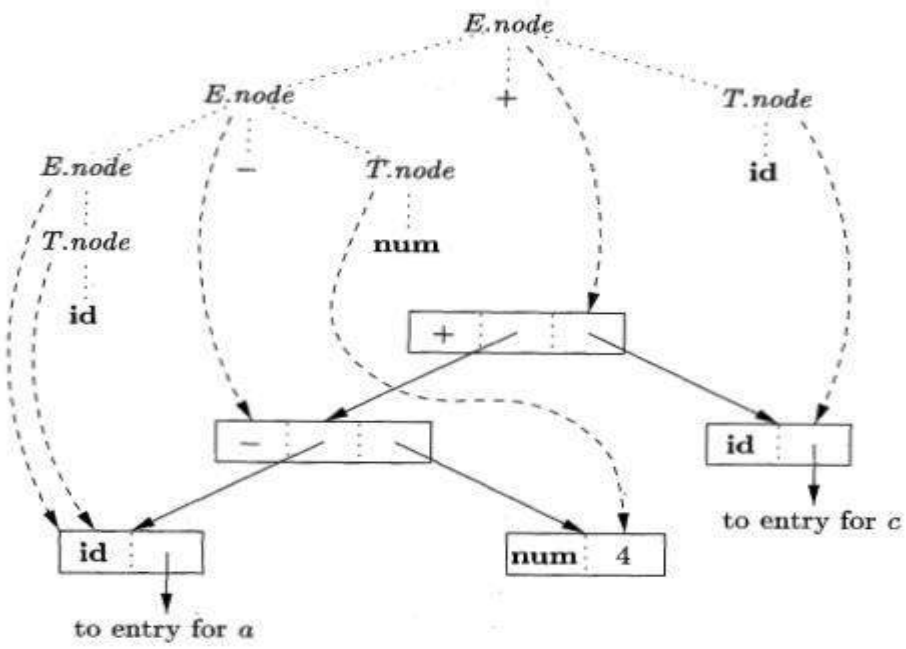


Figure 5.11: Syntax tree for $a - 4 + c$

1) $p_1 = \textbf{new}\ \mathit{Leaf}(\textbf{id}, \mathit{entry\text{-}a});$
2) $p_2 = \textbf{new}\ \mathit{Leaf}(\textbf{num}, 4);$
3) $p_3 = \textbf{new}\ \mathit{Node}('-', p_1, p_2);$
4) $p_4 = \textbf{new}\ \mathit{Leaf}(\textbf{id}, \mathit{entry\text{-}c});$
5) $p_5 = \textbf{new}\ \mathit{Node}('+', p_3, p_4);$

Figure 5.12: Steps in the construction of the syntax tree for $a - 4 + c$

2 The Structure of a Type

The type int [2][3] can be read as, "array of 2 arrays of 3 integers." The corresponding type expression array(2, array(3, integer)) is represented by the tree in Fig. 5.15. The operator array takes two parameters, a number and a type. If types are represented by trees, then this operator returns a tree node labeled array with two children for a number and a type.

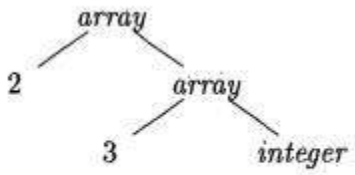


Figure 5.15: Type expression for **int**[2][3]

Nonterminal *B* generates one of the basic types **int** and **float.** *T* generates a basic type when *T* derives *B C* and *C* derives e. Otherwise, *C* generates array components consisting of a sequence of integers, each integer surrounded by brackets.

PRODUCTION	SEMANTIC RULES
$T \rightarrow B\ C$	$T.t = C.t$
	$C.b = B.t$
$B \rightarrow$ **int**	$B.t = integer$
$B \rightarrow$ **float**	$B.t = float$
$C \rightarrow$ [**num**] C_1	$C.t = array(\mathbf{num}.val,\ C_1.t)$
	$C_1.b = C.b$
$C \rightarrow \epsilon$	$C.t = C.b$

Figure 5.16: T generates either a basic type or an array type

An annotated parse tree for the input string **int [** 2] [3] is shown in Fig. 5.17. The corresponding type expression in Fig. 5.15 is constructed by passing the type *integer* from *B,* down the chain of C's through the inherited attributes *b.* The array type is synthesized up the chain of C's through the attributes *t.*

In more detail, at the root for *T* -» *B C,* nonterminal *C* inherits the type from *B,* using the inherited attribute *C.b.* At the rightmost node for C, the production is C e, so C.t equals C.6. The semantic rules for the production C [num] C1 form C.t by applying the operator array to the operands num.ua/ and C1.t.

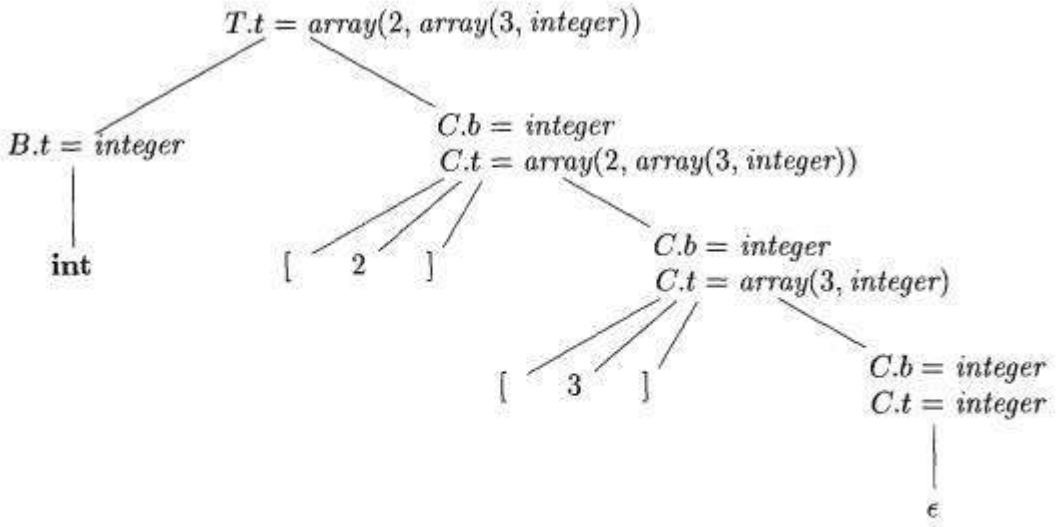


Figure 5.17: Syntax-directed translation of array types

Syntax Directed Translation Schemes.

syntax-directed translation scheme (SDT) is a context-free grammar with program fragments embedded within production bodies. The program fragments are called *semantic actions* and can appear at any position within a production body. By convention, we place curly braces around actions; if braces are needed as grammar symbols, then we quote them.SDT's are implemented during parsing, without building a parse tree.

Two important classes of SDD's are

1.The underlying grammar is LR-parsable, and the SDD is S-attributed.

2.The underlying grammar is LL-parsable, and the SDD is L-attributed.

1 Postfix Translation Schemes

simplest SDD implementation occurs when we can parse the grammar bottom-up and the SDD is S-attributed. In that case, we can construct an SDT in which each action is placed at the end of the production and is executed along with the reduction of the body to the head of that production. SDT's with all actions at the right ends of the production bodies are called postfix SDT's.

Example 5.14 : The postfix SDT in Fig. 5.18 implements the desk calculator SDD of Fig. 5.1, with one change: the action for the first production prints a value. The remaining actions are exact counterparts of the semantic rules. Since the underlying grammar is LR, and the SDD is S-attributed, these actions can be correctly performed along with the reduction steps of the parser.

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

Figure 5.18: Postfix SDT implementing the desk calculator

2 Parser-Stack Implementation of Postfix SDT's

The attribute(s) of each grammar symbol can be put on the stack in a place where they can be found during the reduction. The best plan is to place the attributes along with the grammar symbols (or the LR states that represent these symbols) in records on the stack itself.

In Fig. 5.19, the parser stack contains records with a field for a grammar symbol (or parser state) and, below it, a field for an attribute. The three grammar symbols *X YZ* are on top of the stack; perhaps they

are about to be reduced according to a production like *A —> X YZ.* Here, we show *X.x* as the one attribute of *X*, and so on. In general, we can allow for more attributes, either by making the records large enough or by putting pointers to records on the stack. With small attributes, it may be simpler to make the records large enough, even if some fields go unused some of the time. However, if one or more attributes are of unbounded size — say, they are character strings — then it would be better to put a pointer to the attribute's value in the stack record and store the actual value in some larger, shared storage area that is not part of the stack.

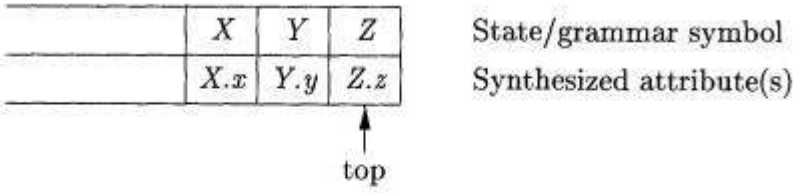


Figure 5.19: Parser stack with a field for synthesized attributes

3 SDT's With Actions Inside Productions

An action may be placed at any position within the body of a production.It is performed immediately after all symbols to its left are processed. Thus,if we have a production B -» X {a} Y, the action a is done after we have recognized X (if X is a terminal) or all the terminals derived from X (if X is a nonterminal).

More precisely,

• If the parse is bottom-up, then we perform action a as soon as this occurrence of X appears on the top of the parsing stack.

• If the parse is top-down, we perform a just before we attempt to expand this occurrence of Y (if Y a nonterminal) or check for Y on the input (if Y is a terminal).

4 Eliminating Left Recursion From SDT's

First, consider the simple case, in which the only thing we care about is the order in which the actions in an SDT are performed. For example, if each action simply prints a string, we care only about the order in which the strings are printed. In this case, the following principle can guide us:

When transforming the grammar, treat the actions as if they were terminal symbols.

This principle is based on the idea that the grammar transformation preserves the order of the terminals in the generated string. The actions are therefore executed in the same order in any left-to-right parse, top-down or bottom-up.

The "trick" for eliminating left recursion is to take two productions

$$A \rightarrow Aa \mid b$$

that generate strings consisting of a *j3* and any number of en's, and replace them by productions that generate the same strings using a new nonterminal *R* (for "remainder") of the first production:

$A \rightarrow bR$

R —»• $aR \mid e$

If *(3* does not begin with *A,* then *A* no longer has a left-recursive production. In regular-definition terms, with both sets of productions, *A* is defined by ***0(a)****.

Example **5** . 1 7 : Consider the following E-productions from an SDT for translating infix expressions into postfix notation:

E -> E i + T { print('+'); }

E -> T

If we apply the standard transformation to E, the remainder of the left-recursive production is

a = + T { print('-r'); }

and the body of the other production is T. If we introduce R for the remainder of E, we get the set of productions:

E --> T R

R --> + T { printC-h'); } R

R -> e

When the actions of an SDD compute attributes rather than merely printing output, we must be more careful about how we eliminate left recursion from a grammar. However, if the SDD is S-attributed, then we can always construct an SDT by placing attribute-computing actions at appropriate positions in the new productions.