WWW.KTUNOTES.IN

MODULE-4.

SYNTAX DIRECTED TRANSLATION

It is possible to associate the CFG with extra information using programming language construct. There are 2 notations for associating semantic rules with production.

(i) Syntax directed definition (SDD)

(ii) Translation schemes

SDD has high level specification for translation. They hide many implementation details and free the user from having to specify explicitly the order in which translation takes place.

Translation schemes indicate order in which order in which semantic rules are to be evaluated, so they allow some implementation details to be shown.

Conceptual view of SD translation

input string → parse tree → dependency graph → evaluation order for semantic rules

7.2.18

SDD

A SDD is a generalisation of a CFG in which each grammar symbol has an associated set of attributes partitioned into two subsets called synthesized and inherited attribu

of that grammar symbol.

FORMS OF SDD

In a SDD, each grammar production

$$A \rightarrow \alpha$$

has associated with it a set of semantic rule of the form

$$b := f(c_1, c_2, \ldots, c_k)$$

where f is a function and either

(i) b is a synthesized attribute of A and $c_1, c_2, \ldots, c_k$ are attributes belonging to gramm. or symbol of production or.

(ii) b is an inherited attribute of grammar symbols on RHS of production and $c_1, c_2, \ldots, c_k$ are attributes belonging to grammar symbol of the production.

Eg: The SDD for a simple calculator is given below:

Production	Semantic rules
$L \rightarrow En$.	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 x F.val
$T \rightarrow F$	T.val = F.val
$F \rightarrow (E)$	F.val = E.val
$F \rightarrow$ digit	F.val = digit.lexval

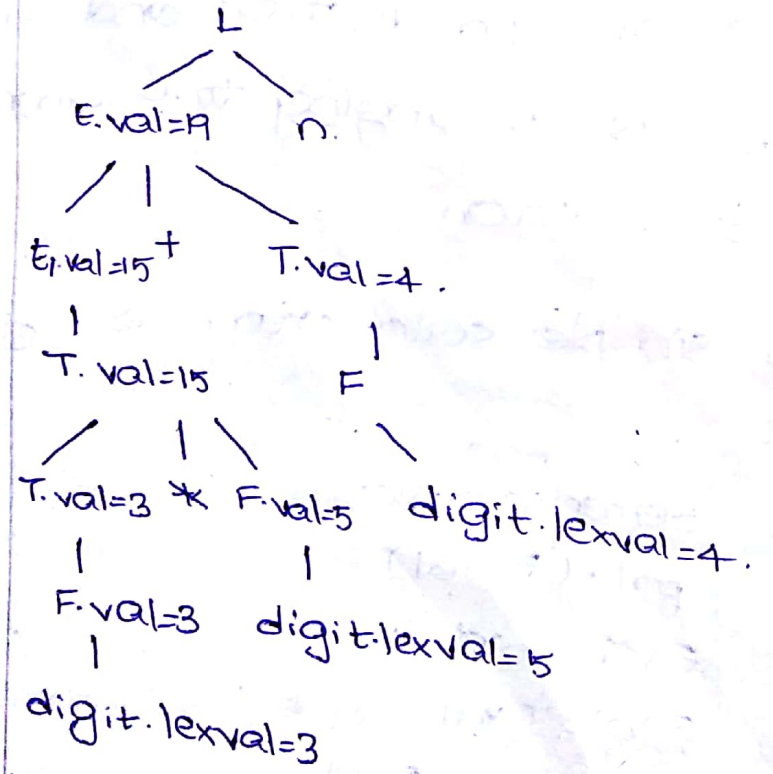
8.2.18 Synthesized attribute.

A SDD that uses synthesized attributes exclusively is said to be an S-attributed. A parse tree for an S-attributed definition can always be annotated (decorated) by evaluating the semantic rules for the attributes of each node bottom-up from the leaves to the root.

Annotated parse tree.

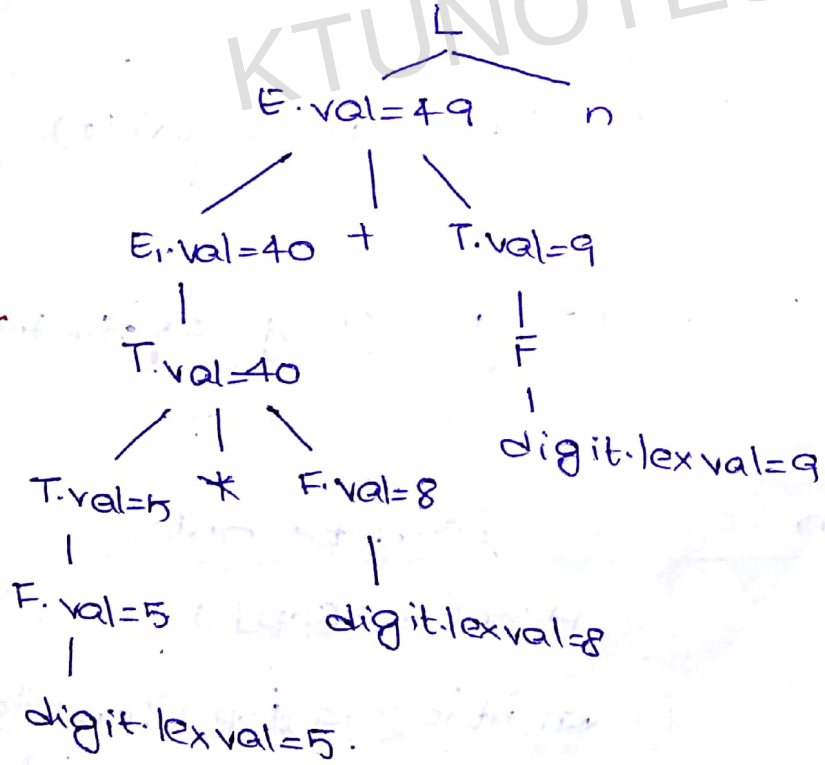
Draw an annotated parse tree for expression 3*5+4n using above given SDD of simple calculaton.

Solution.



A parse tree showing values of attributes at each node is called annotated parse tree. The process of computing the attribute values is called annotating or decorating parse tree. The value of a synthesized attribute at a node is computed from values of attribute at the children of the node in the parse tree. ~~The~~ An inherited attribute is one whose value at a node in a parse tree is defined in terms of attributes at parent and/or sibling of that node.

Q Draw an annotated parse tree for the expression 5 * 8 + 9n



3.18

Bottom-up evaluation of S attributed definitions

The syntax directed definition (SDD) with

only synthesized attributed called S-attributed definition. The Synthesized attribute can be evaluated by a bottom-up parser as the input is being parsed. Parser can keep the values of synthesized attribute associated with the values of a grammar symbol on its state. Whenever a reduction is made, the values of the new synthesized attributes are computed from the attributes appearing on the stack for the grammar on the right side of the reducing production.

The stack is implemented by a pair of arrays

1) State. (pointer to LR(1) Parser table)

2) val

Q Do LR parsing on input string 3* 5+4n using SDD

Production	Semantic rule.
L → En	Print (val [top])
E → E1 + T	val [ntop] = val [top-2] + val [top]
E → T	
T → T1 * F	val [ntop] = val [top-2] * val [top]
T → F	
F → (E)	val [ntop] = val [top]
F → digit	

stack.

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
-	En	19	-
	L	19	L → E

L-attributed Definition

A syntax directed definition is L attributed if each inherited attribute of X_j, $1 \le j \le n$ on the right side of production $A \rightarrow X_1 X_2 \dots X_n$

depends only on

1) The attributes of the symbols $X_1, X_2, \ldots X_{j-1}$ to the left X_j in the production and

2) The inherited attributes of A.

5.3.18 TRANSLATION SCHEMES

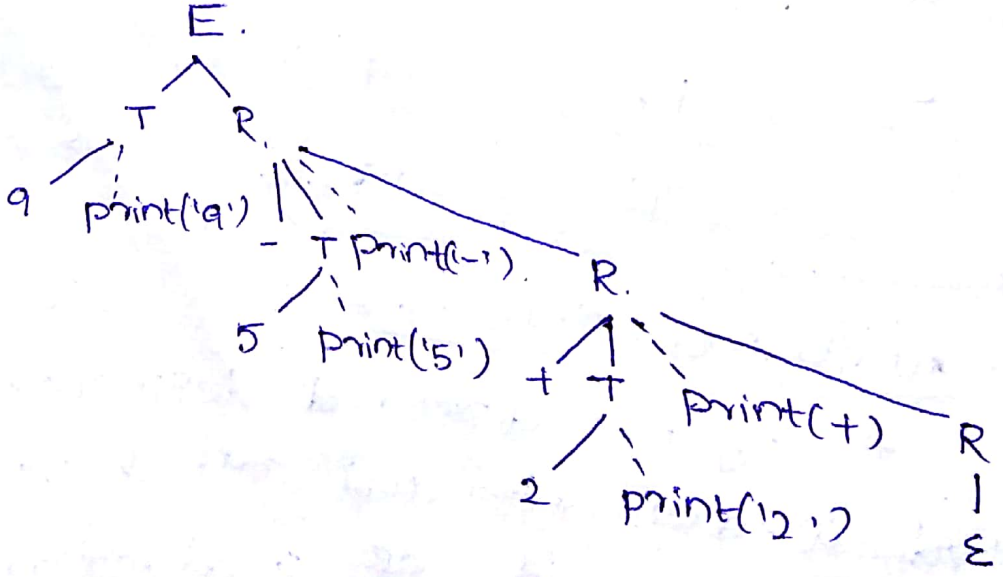
A translation scheme is a CFG in which attributes are associated with grammer symbol and semantic actions are enclosed within braces { } are inserted with the right side of the production.

Q. Convert the expression 9-5+2 into postfix expression using translation scheme as given below.

$E \rightarrow TR$.

$R \rightarrow addop\ T$ {print (addop.lexeme)} $R_1 \mid \varepsilon$

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



When designing a translation scheme, we must observe some restriction to ensure that an attribute value is available when a action refers to it. These restrictions motivated by L attributed definition ensure that an action does not refer to an attribute that has not yet bean computed

Q consider a SDD for size and height of boxes

Production	Semantic rule.
$S \to B$	$B.ps = 10$ $S.ht = B.ht$.
$B \to B_1 B_2$	$B_1.ps = B.ps$ $B_2.ps = B.ps$ $B.ht = max(B_1.ht, B_2.ht)$
$B \to B_1 sub B_2$	$B_1.ps = B.ps$ $B_2.ps = Shrink(B.ps)$ $B.ht = disp(B_1.ht, B_2.ht)$
$B \to text$	$B.ht = text.h \times B.ps$.

$S \to \{B.ps = 10\}$

$B\{S.ht = B.ht\}$

$B \to \{B_1.ps = B.ps\}$

B_1 { $B_2.ps = B.ps$ }

B_2 { $B.ht = \max(B_1.ht, B_2.ht)$ }

$B \rightarrow$ { $B_1.ps = B.ps$ }

B_1

sub { $B_2.ps = shrink(B.ps)$ }

B_2 { $B_2.ht = disp(B_1.ht, B_2.ht)$ }

$B \rightarrow$ text { $B.ht = text.h \times B.ps$ }

6.3.18

TOP DOWN TRANSLATION

Eliminating left recursion from translation scheme.

Since most arithmetic operators associate to the left, it is natural to use left recursive grammars for expressions.

Consider production of form

$A \rightarrow A\alpha \mid \beta$

that generate strings consisting of a β and any no. of α and replace them by productions that generate new same strings using new non-terminal R (for the 'remainder') of the first production.

$A \rightarrow \beta R$

$R \rightarrow \alpha R \mid \varepsilon$

Consider syntax directed translation

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

$E \rightarrow T$

$A - E$

$\alpha - + T$ { print ('+') }

$B - T$

$E \rightarrow T R$

$R \rightarrow + T$ { print ('+') } $R \mid \varepsilon$

BOTTOM UP EVALUATION OF INHERITED ATTRIBUTES

In bottom up evaluation of inherited attributes, we have ~~two~~ make to transform all embedded actions in translation schemes to be attached at the right end of their productions. The transformation inserts a new marker non-terminal generating ε into base grammar. We replace each embedded action by a distinct marker non-terminal and attach action to end of production $M \rightarrow \varepsilon$.

Example: Consider translation scheme

$E \rightarrow TR$

$R \rightarrow + T$ { print ('+') } ~~R | ε~~

$R \rightarrow - T$ { print ('-') } $R \mid \varepsilon$

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

The above given translation scheme can be transformed using marker non-terminals M, N into

$E \rightarrow TR$

$R \rightarrow +TMR \mid -TNR \mid \varepsilon$

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

$M \rightarrow \varepsilon \{ print('+') \}$

$N \rightarrow \varepsilon \{ print('-') \}$

4.3.18

A bottom up parser reduces right side of production $A \rightarrow xy$ by removing x and y from top of parser stack and replacing them by A

Consider the translation scheme for declaration of identifiers of type integer and real.

$D \rightarrow T \{ L.in = T.type \}$

L

$T \rightarrow int \{ T.type = integer \}$

$T \rightarrow real \{ T.type = real \}$

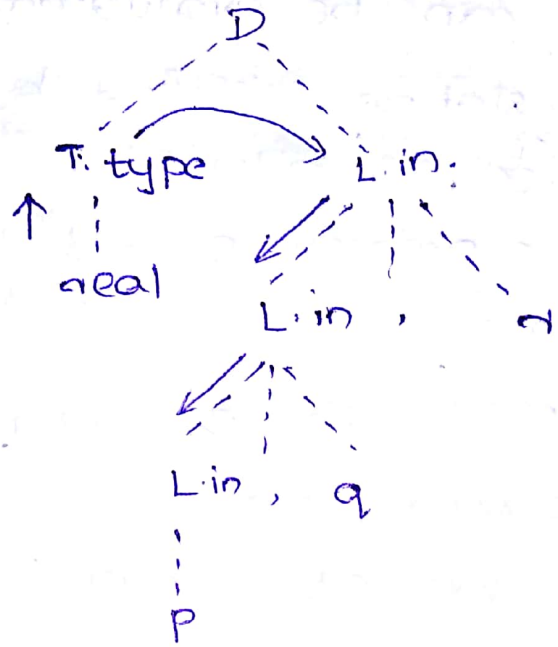
$L \rightarrow \{ L.in = L.in \}$

$Li, id \{ addtype(id.entry, L.in) \}$

$L \rightarrow id \{ addtype(id.entry, L.in) \}$

Construct a parse tree for the expression

real p, q, r



The dependency graph corresponding to the expression can be drawn using translation scheme given. The interdependency among the inherited and synthesized attributes at the nodes in a parse tree can be depicted by a directed graph called dependency graph.

The topological sort of a directed acyclic graph is any ordering $m_1, m_2, \ldots\ldots, m_k$ of the nodes of the graph such that edges go from nodes earlier in the ordering to the later nodes i.e., if $m_i \rightarrow m_j$ is an edge from m_i to m_j, then m_i appears before m_j in the ordering. Any topological sort of a dependency graph gives a valid order in which the semantic rules associated with nodes

in parse tree can be evaluated.

9.3.18. The translation specified by a SDD can be made precise as follows:

(i) The underline grammar is used to construct a parse tree for i/p.

(ii) The dependency graph is constructed as discussed

(iii) From a topological sort of dependency graph we obtain a evaluation order for semantic rules

(iv) evaluation of semantic rules in this order yields translation of i/p string.

THE BOTTOM up expansing of expression real p,q,r can be done as follows

Input	State	Production used
real p,q,r	-	-
p,q,r	real	T→real
p,q,r	T	T→real
,q,r	Tp	-
,q,r	TL	L→id.
q,r	TL,	-
,r	TL q	-
,r	TL	L→L,id
r	TL,	-
	TL,r	

	D	D→TL

Construction of syntax tree.

SDD can be used to specify construction of syntax tree and other graphical representation of language constructs

Definition.

An (abstract) syntax tree is a condensed form of parse tree useful for representing the language constructs

Constructing a syntax tree for expressions.

The construction of a syntax tree for an expression is similar to translation of exp. into postfix form. Each node in a syntax tree can be implemented as a record with several fields. In the node for an operator, one field identifies the operator and the remaining fields contain pointer to the nodes for the operands. The operator is often called label of node,

The following functions are used to create nodes of syntax tree for exp. with binary operators. Each function returns a pointer to a newly created nodes.

1) mknode (op, left, right) creates an operator node with label op and two fields containing pointers to left and right

2) mkleaf (id, entry) creates an identifier node with label id and field containing entry a pointer to the symbol table entry for the identifier id.

3) mkleaf (num, val) creates a number node with a label num and a field containing val, the value of the number.

Q. Construct a syntax tree for expression a-4+c

The following sequence of function calls create the syntax tree for given exp. In this sequence $P_1, P_2, \ldots, P_5$ are pointers to nodes. In entry a and entry c are pointers to symbol table entries for identifiers a and c, as follows

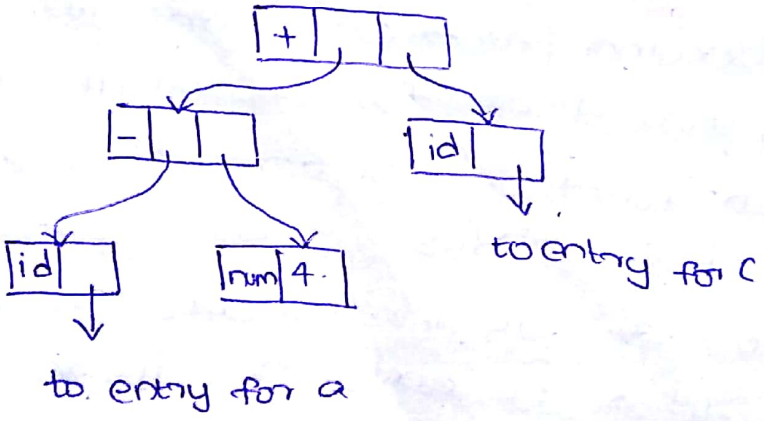
(1) P_1 = mkleaf (id, entry a)

(2) P_2 = mkleaf (num, 4)

(3) P_3 = mknode (-, P_1, P_2)

(4) P_4 = mkleaf (id, entry c)

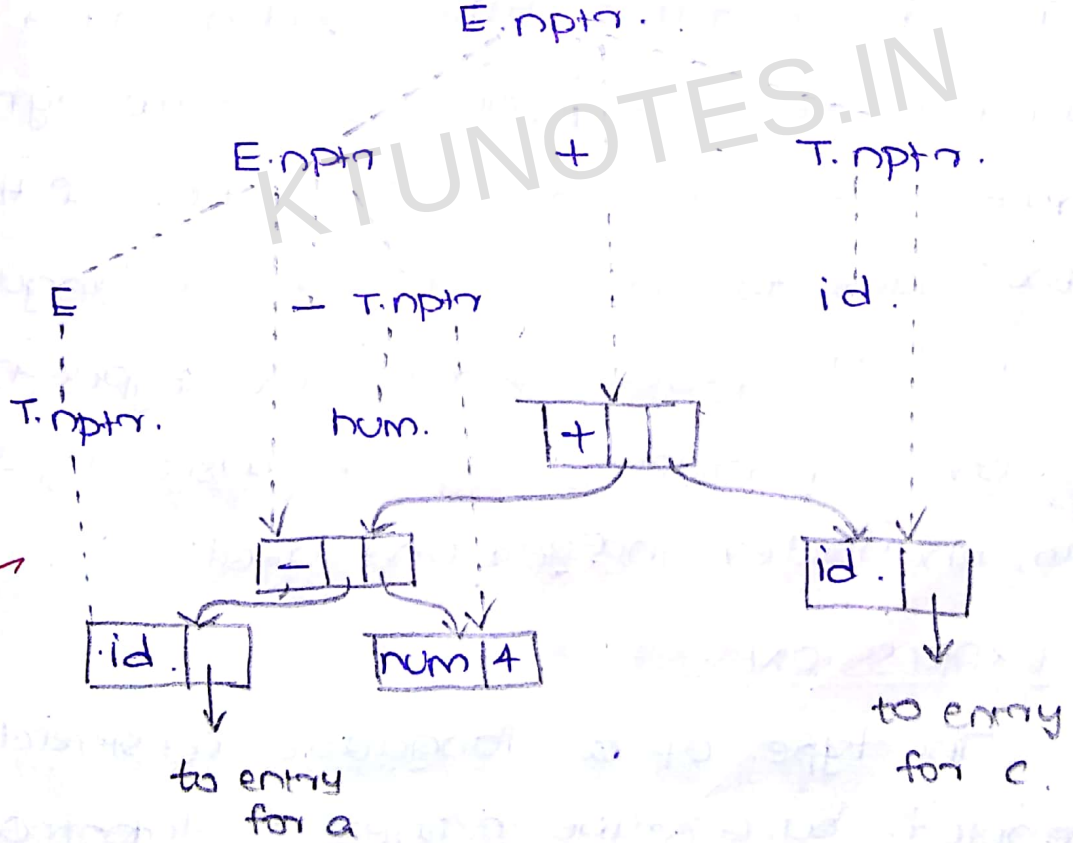
(5) P_5 = mknode (+, P_3, P_4)



SYNTAX DIRECTED DEFINITION FOR CONSTRUCTING SYNTAX TREES

Consider a SDD given below

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.entry)
$T \rightarrow num$	T.nptr = mkleaf (num, num.val).



TYPE CHECKING

A compiler must check that the source program follow both the synthatic and semantic conventions of source program language. Thus checking is called static checking. Eg: static checking are:

1. Type checks.
2. Flow of control check.
3. Uniqueness checks.
4. Name-related check.

TYPE SYSTEM

The design for a type checker for a language as based on the information about the synthactic construct in the language, the notion of the language and the rules for assigning types to language construct. In both pascal and C, the types are either basic or constructed Eg of basic types are boolean, character, integer and real.

12.3.18

TYPE EXPRESSION

The type of a language construction will be denoted by a type expression. Informally, a type expression is either a basic type or is formed by applying an operator called type constructor to other type expressions. Some of the type expressions are listed below:

(i) a basic type is type expression
Eg: Boolean, char, integer and real.

(ii) Since type expression may be named, a type name is a type expression.

(iii) A type constructor applied to type expressions is a type expression. Constructors include

(a) Arrays : If T is a type expression, then array(I,T) is a type exp. denoting the type of an array with elements of type T and index set I.

Eg : var A. array[1...10] of integer;

Associate type expression array [1...10 integer

(b) Products : If T_1 and T_2 are type exp., then their cartesian product $T_1 \times T_2$ is type exp.

(c) Records : The record type constructor will be applied to a tuple formed from field names and field type

(d) Pointer:

If T is a type exp. then pointer(T) is a type exp denoting type 'pointer to an object of type T. Eg: In pascal, declaration variable var p : ↑ row. Variable p to have type pointer of row.

(e) Functions : We may treat functions in programming language as a domain type D to a range type, R.

function f(a,b: char): ↑ integer;

SPECIFICATION OF SIMPLE TYPE CHECKER.

In this section, we specify a type checker for a simple language in which type of each identifier must be declared before identifier is used.

Eg: Grammar given below generates programs represented by the non-terminal P, consisting of declarations D followed by a single expression E.

$P \rightarrow D;E$

$D \rightarrow D;D \mid id:T.$

$T \rightarrow char \mid integer \mid array[num]\ of\ T \mid \uparrow T$

$E \rightarrow literal \mid num \mid id \mid E\ mod\ E \mid E[E] \mid E\uparrow$

The SDD corresponding to above given CFG is as follows:

$P \rightarrow D;E$	
$D \rightarrow D;D.$	
$D \rightarrow id;T$	{addtype(id.entry, T.type)}
$T \rightarrow char$	{T.type := char}
$T \rightarrow integer$	{T.type := integer}
$T \rightarrow \uparrow T_1$	{T.type := pointer(T1.type)}
$T \rightarrow array[num]\ of\ T_1$	{T.type := array(1...num.val, T1.type)}

13.3.18

TYPE CHECKING FOR EXPRESSIONS

In the following rules, the synthesized attribute type for E gives the type expression

assigned by type system to the expression generated by E.

The following semantic rules say that constants represented by tokens literal and num have type char and integer, respectively.

E → literal {E.type = char}

E → num {E.type = integer}

We use a function lookup(e) to fetch the type saved in symbol table entry pointer to by e, i.e.,

E → id {E.type = lookup(id.entry)}