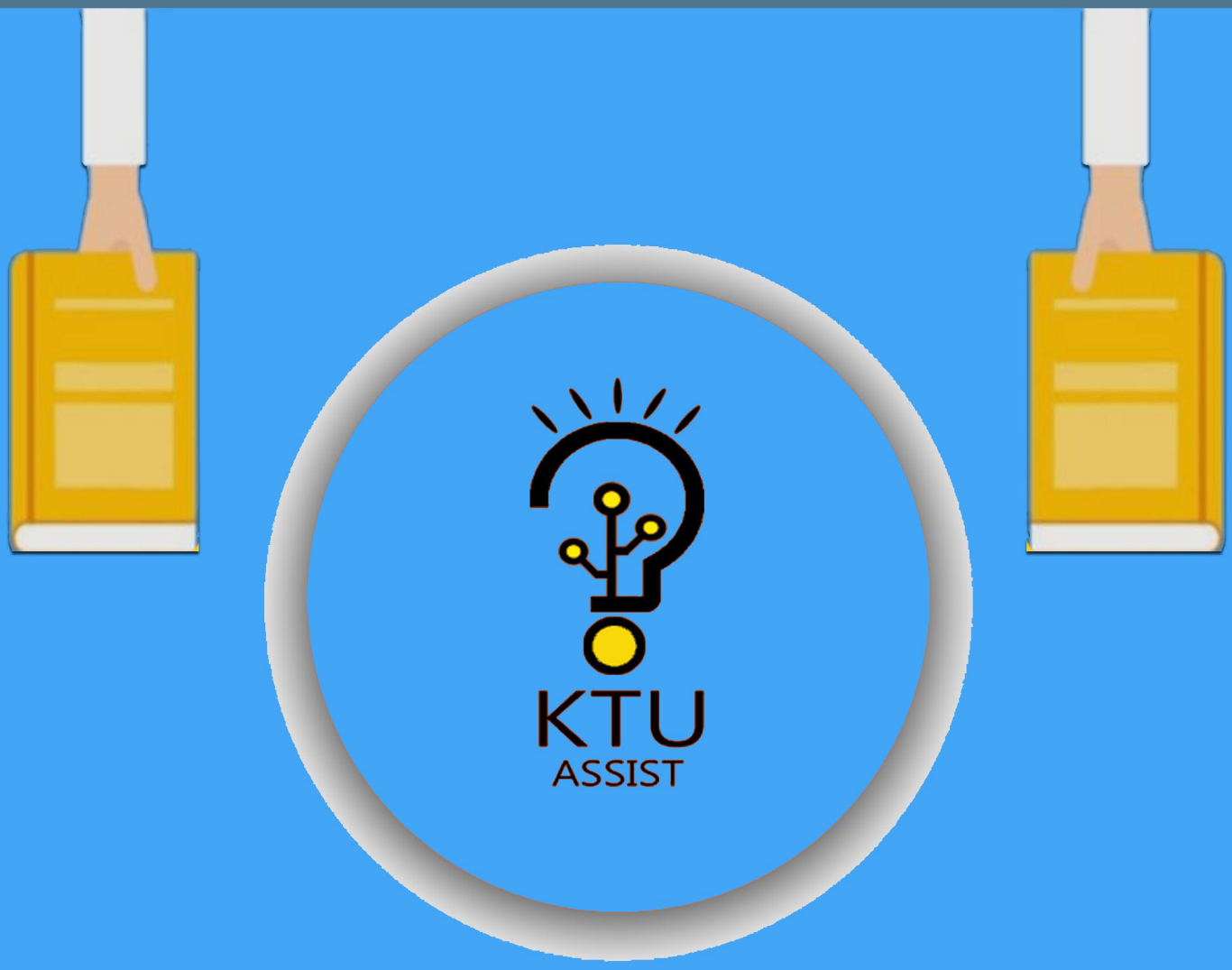
APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

STUDY MATERIALS



a complete app for ktu students

Get it on Google Play

www.ktuassist.in

Module 4

Syntax Directed Translation & Type Checking

SYNTAX DIRECTED TRANSLATION(SDT)

Ques 1) What do you understand by term Syntax directed translation?

Ans: Syntax-Directed Translation (SDT)
Syntax-directed translation (SDT) refers to a method of compiler implementation where the source language translation is completely driven by the parser, i.e., based on the syntax of the language The parsing process and parse trees are used to direct semantic analysis and the translation of the source program. Almost all modern compilers are syntax-directed.

SDT can be a separate phase of a compiler or one can augment conventional grammar with information to control the semantic analysis and translation. Such grammars are called **attribute grammars**.

We augment a grammar by associating attributes with each grammar symbol that describes its properties. With each production in a grammar, we give semantic rules/ actions, which describe how to compute the attribute values associated with each grammar symbol in a production.

The general approach to Syntax-Directed Translation is to construct a parse tree or syntax tree and compute the values of attributes at the nodes of the tree by visiting them in some order. In many cases, translation can be done during parsing without building an explicit tree.

A class of syntax-directed translations called "L-attributed translations" (L for left-to-right) includes almost all translations that can be performed during parsing. Similarly, "S-attributed translations" (S for synthesized) can be performed easily in connection with a bottom-up parse.

There are two ways to represent the semantic rules associated with grammar symbols:
1) Syntax-Directed Definitions (SDD)
2) Syntax-Directed Translation Schemes (SDT)

Conceptually, with both syntax-directed definitions and translation schemes, one can parse the input token stream, build the parse tree, and then traverse the tree as needed to evaluate the semantic rules at the parse-tree nodes (**figure 4.1**):

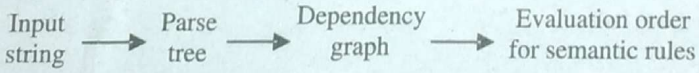


Figure 4.1: Conceptual View of Syntax-Directed Translation

Evaluation of the semantic rules may generate code, save information in a symbol table, issue error messages, or perform any other activities. The translation of the token stream is the result obtained by evaluating the semantic rules.

Ques 2) What is Syntax-Directed Definitions (SDD)?

Ans: Syntax-Directed Definitions (SDD)
Syntax directed definitions are high-level specifications for translations. They hide many implementation details and free the user from having to specify explicitly the order in which translation takes place.

A **syntax-directed definition (SDD)** associates a **semantic rule** with each grammar production; the rule states how attributes are calculated:
1) Conceptually, each node may have multiple attributes. Perhaps a struct/record/dictionary is used to group many attributes
2) Attributes may be concerned e.g. with data type, numeric value, symbol identification, code fragment, memory address, machine register choice

DOWNLOADED FROM WWW.KTUASSIST.IN
Scanned by CamScanner

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.

A SDD is a context-free grammar, plus attributes, and rules attached to the productions of the grammar stating how attributes are computed.

For example, consider the Desk calculator's Arithmetic Expression example:

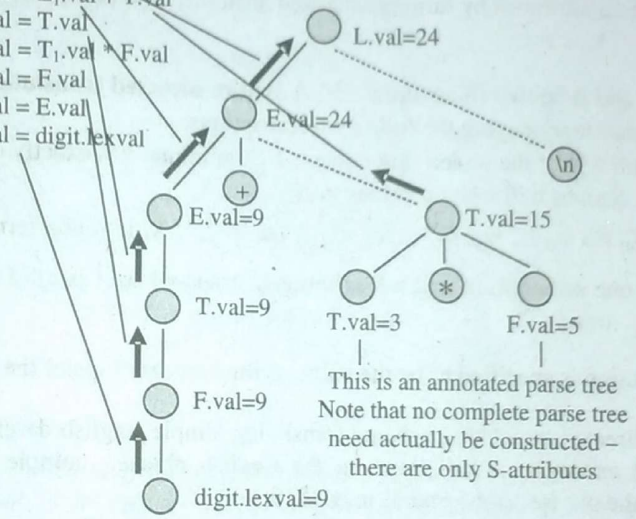
i) All attributes in this example are Synthesized,
ii) val and lexval attribute names are purposely different,
iii) Subscripts distinguish occurrences of the same grammar symbol, and
iv) Practical rules may also have side-effects (printing, manipulating a symbol table).

Desk calculator's Arithmetic Expression is shown below:

L::= E\n	L.val = E.val
E::= E_1 + T	E.val = E_1.val + T.val
E::= T	E.val = T.val
T::= T_1 * F	T.val = T_1.val * F.val
T::= F	T.val = F.val
F::= (E)	F.val = E.val
F::= digit	F.val = digit.lexval

SDD Applied to "9+3*5\n"

L::= E\n	L.val = E.val
E::= E_1 + T	E.val = E_1.val + T.val
E::= T	E.val = T.val
T::= T_1 * F	T.val = T_1.val * F.val
T::= F	T.val = F.val
F::= (E)	F.val = E.val
F::= digit	F.val = digit.lexval



In a syntax-directed definition, each grammar production A®a has associated with it a set of semantic rules of the form b = f(c_1, c_2,……., c_k), where f is a function, and either:

1) b is a synthesized attribute of A and c_1, c_2,……,c_k are attributes belonging to the grammar symbols of the production; or
2) b is an inherited attribute of one of the grammar symbols on the right side of the production, and c_1, c_2,……., c_k are attributes belonging to the grammar symbols of the production.

In either case, one say that the attribute b depends on attributes c_1, c_2,……,c_k. An attribute grammar is a syntax directed definition in which the functions in semantic rules cannot have side-effects.

Ques 3) Explain the Syntax Directed Translation Scheme (SDTS) with example.

Or

Write the guidelines to design translation schemes.

Ans: Syntax Directed Translation Scheme (SDTS)

In order to perform semantic analysis, we make use of a formalism called **syntax directed translation scheme (SDTS).**

Syntax directed translation scheme (SDTS) is a context free grammar in which there are attributes associated with the grammar symbols and semantic actions enclosed within braces are inserted with the right hand sides of productions. They are useful for specifying translations during parsing.

DOWNLOADED FROM WWW.KTUASSIST.IN

Scanned by CamScanner

Any SDT can be implemented by first building a parse tree and then performing the actions in a left-to-right depth-first order; that is, during a preorder traversal.

SDT implement two important classes of SDD's:

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.

Guidelines to Design Translation Schemes

1) If one have only synthesized attributes, create an action consisting of an assignment for each semantic rule and place this action at the end of the right side of the associated production.

   For example, in $E \rightarrow E_1 + E_2$ {E.val := E_1.val + E_2.val }

2) A translation scheme involving both synthesized and inherited attributes, the following rules have to be followed.
   i) An inherited attribute for a symbol on the right hand side of a production must be computed in an action before that symbol.
   ii) An action must not refer to a synthesized attribute of a symbol to the right of the action.
   iii) A synthesized attribute for the non-terminal on the left can only be computed after all attributes it references have been computed. The action for computing such attributes can usually be placed at the end of the right side of the production.

It is always possible to start with an L-attributed definition and construct a translation scheme that satisfies all the above requirements. Given an L-attributed grammar in which no inherited attributes depend on synthesized attributes, a recursive-descent parser can evaluate all attributes by turning inherited attributes into parameters and synthesize attributes into return values.

Formal Definition: Let $\sum$ and Δ be two finite alphabets. A **syntax-directed translation scheme** defining the translation T over $\sum \times \Delta$ is an attribute grammar obeying the following restrictions:

1) Each non-terminal symbol X of the underlying context-free grammar has exactly one synthesized attribute X.trans and no inherited attributes. X.trans will take on values in Δ^*.
2) Each production [p: $X_0 ::= \alpha_0 X_1 \alpha_1 X_2 \ldots X_{n_p} \alpha_{n_p}$] ($\alpha_i \in \sum^*$, X_i is a non-terminal of the underlying context-free grammar) has exactly one semantic function f defining X_0.trans where f is a token permuting function over Δ of the form f(X_1.trans,…, X_{n_p}.trans).
3) The value of the translation is specified to be the value of the trans attribute of the root (S.trans).

Table 4.1 gives a syntax-directed translation scheme translating simple English descriptions of mathematical expressions into post-fix notation. **For example,** it will translate the English phrase 'multiple 5.7 and 8' into the post-fix Polish expression '(5.7, 8,*)' and the phrase 'add 5 and 9' into '(5, 9, +)'.

Table 4.1: Syntax-Directed Translation Scheme

p_1:	S::= Op Number1 and Number2.
	S.trans = Concatenate('(', Number1.trans, ',', Number2.trans, ',',Op.trans, ')');
p_2:	Number::= Integer.
	Number.trans = Integer.trans;
p_3:	Number::= Decimal_num.
	Number.trans = Decimal_num.trans;
p_4:	Op::= add.
	Op.trans = '+';
p_5:	Op::= multiple.
	Op.trans = '*';
p_6:	Integer::= digits.
	Integer.trans = digits.trans;
p_7:	Decimal_num::= digits1.digits2.
	Decimal_num.trans = Concatenate(digits1.trans, '.',.digits2.trans);

DOWNLOADED FROM WWW.KTUASSIST.IN

Scanned by CamScanner

Example: Let us consider the grammar to generate simple expressions in infix form and a translator which converts it into postfix form.

```
expr → expr + term      {print ('+')}
expr → term
term → 0                {print ('0')}
⋮
term → 9                {print ('9')}
```

We can remove left recursion from this grammar. The transformed translation scheme is as follows:

```
expr → term R
R → + term      {print ('+')} R | ∈
term → 0        {print ('0')}
term → 1        { print ('1')}
⋮
term → 9        { print ('9')}.
```

We can show the parse tree for the input 5 + 4 + 6 with each semantic action attached as the appropriate child of the node corresponding to the left side of their production as in **figure 4.2**. When performed in depth-first order, the action of **figure 4.2** print the output 5 4 + 6 +.

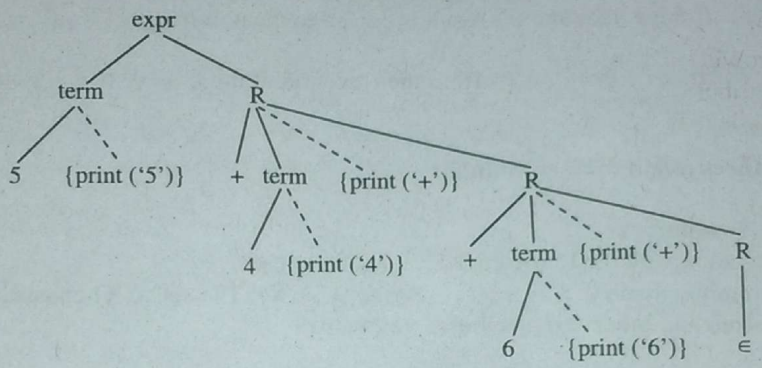


Figure 4.2: Parse Tree Showing Actions of a Translation Scheme to Translate 5 + 4 + 6 to 5 4 + 6 +.

Ques 4) What is attribute?

Ans: Attribute

An attribute is any property of a programming language construct. Attributes can vary widely in the information they contain, their complexity, and particularly the time during the translation/execution process when they can be determined.

Some typical **examples** of attributes are:

1) The data type of a variable,
2) The value of an expression,
3) The location of a variable in memory,
4) The object code of a procedure, and
5) The number of significant digits in a number.

Attributes may be fixed prior to the compilation process (or even the construction of a compiler). **For example,** the number of significant digits in a number may be fixed (or atleast given a minimum value) by the definition of a language.

Ques 5) What is attribute grammar?

Ans: Attribute Grammar

An **attribute grammar** is a **Syntax-Directed Definition (SDD)** in which the functions in the semantic rules cannot have side-effects (they can only evaluate values of attributes).

An attribute grammar is a device used to describe more of the structure of a programming language than can be described with a context-free grammar.

DOWNLOADED FROM WWW.KTUASSIST.IN

Scanned by CamScanner

An attribute grammar is an extension to a context-free grammar. The extension allows certain language rules to be conveniently described, such as type compatibility.

An attribute grammar is an SDD in which functions in the semantic rules cannot have side effects; they can only evaluate values of attributes. A semantic rule $b = f(C_1, C_2, \ldots, C_n)$ indicates that the attribute b depends on attributes $C_1, C_2, \ldots, C_n$. In an SDD, a semantic rule may just evaluate a value of an attribute or it may have some side effects such as printing values.

Attribute grammar is a special form of context-free grammar where some additional information (attributes) are appended to one or more of its non-terminals in order to provide context-sensitive information. Each attribute has well-defined domain of values, such as integer, float, character, string, and expressions.

Attribute grammar is a medium to provide semantics to the context-free grammar and it can help specify the syntax and semantics of a programming language. Attribute grammar (when viewed as a parse-tree) can pass values or information among the nodes of a tree.

For example,

$E \rightarrow E + T$ {E.value = E.value + T.value }

The right part of the CFG contains the semantic rules that specify how the grammar should be interpreted. Here, the values of non-terminals E and T are added together and the result is copied to the non-terminal E.

Semantic attributes may be assigned to their values from their domain at the time of parsing and evaluated at the time of assignment or conditions.

Ques 6) What are the features of attribute grammar?

Ans: Features of Attribute Grammars

An attribute grammar is a grammar with the following additional features:

1) Associated with each grammar symbol X is a set of attributes A(X). The set A(X) consists of two disjoint sets S(X) and I(X), called **synthesized** and **inherited attributes**, respectively.

   Synthesized attributes are used to pass semantic information up a parse tree, while inherited attributes pass semantic information down a tree.

2) Associated with each grammar rule is a set of semantic functions and a possibly empty set of predicate functions over the attributes of the symbols in the grammar rule.
   i) For a rule $X_0 \rightarrow X_1 \ldots X_n$, the **synthesized attributes** of X_0 are computed with semantic functions of the form $S(X_0) = f(A(X_1),\ldots,A(X_n))$. So the value of a synthesized attribute on a parse tree node depends only on the values of the attributes on that node's children nodes.
   ii) **Inherited attributes** of symbols X_i, $1 \leq j \leq n$ (in the rule above), are computed with a semantic function of the form $I(X_i) = f(A(X_0),\ldots, A(X_n))$. So the value of an inherited attribute on a parse tree node depends on the attribute values of that node's parent node and those of its sibling nodes.

Note: To avoid circularity, inherited attributes are often restricted to functions of the form I(Xj) = f(A(X0), …., A(X(j – 1))). This form prevents an inherited attribute from depending on itself or on attributes to the right in the parse tree.

Ques 7) What are the different types of attribute grammar?

Or

Explain the following:

1) S-Attribute

2) L-Attribute

Ans: Types of Attribute Grammar

Based on the way the attributes get their values, they can be broadly divided into two categories:

1) **S-Attributes:** S-attributes stand for synthesized attributes whose values are derived from children (leaves) of grammar symbols. An S-attributed grammar is one that uses only synthesized attributes. For such a grammar, the attributes can obviously be correctly evaluated using a bottom-up walk of the parse tree.

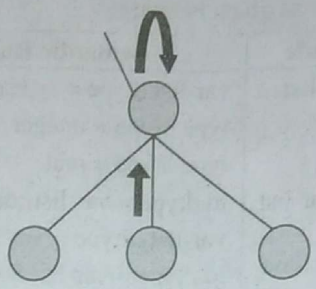
   Furthermore, such a grammar is easily handled by parsing algorithms (such as recursive descent) that do not explicitly build the parse trees.

DOWNLOADED FROM WWW.KTUASSIST.IN

Scanned by CamScanner

A Synthesized attribute is an attribute that depends on the attributes of symbols below it in the parse tree. Synthesized, meaning that its value is obtained from:

i) Attributes of child nodes in the parse tree, or
ii) The lexical analyser, or
iii) Other attributes of the same node in the parse tree.



For example,

$A \rightarrow XYZ$

$A.s = f(X.s \mid Y.s \mid Z.s)$

is synthesized, where s is the attribute of the variable.

2) **L-Attributes Grammar:** An L-attributed grammar is one whose value at a node in a parse tree is defined in terms of attributes at parent or sibling of that node, i.e. the value of an inherited attribute is computed from the values of attributes at the siblings and parents of that node. They are evaluated based on top-down parsing. The L-attributed grammar refers to the inherited attributes of a particular symbol in which any given production is restricted in certain ways. **For example,**

For HL decl → Pd: type, it gets its type from type

$A \rightarrow XYZ$

$Y.i = f(X.i \mid A.i \mid Z.i)$

In each production of the general form,

$A \rightarrow B_1, B_2, \ldots B_n$

The inherited attributes of B_k may depend only on the inherited attributes of A or synthesized attributes of $B_1, B_2, \ldots B_{k-1}$.

For such a grammar the attributes can be correctly evaluated using a left-to-right depth-first walk of the parse tree, and such grammars are usually easily handled by the recursive-descent parser, which implicitly walks the parse tree in this way.

Ques 8) Consider the following grammar:

exp→exp + term | exp – term | term

term →term*factor | factor

factor →(exp) | number

Write the attribute grammar.

Ans: The **attribute grammar** of this grammar is shown as below:

Grammar Rule	Semantic Rules
$exp_1 \rightarrow exp_2 + term$	$exp_1.val = exp_2.val + term.val$
$exp_1 \rightarrow exp_2 - term$	$exp_1.val = exp_2.val - term.val$
$exp \rightarrow term$	exp.val = term.val
$term_1 \rightarrow term_2 * factor$	$term_1.val = term_2.val * factor.val$
term→factor	term.val = factor.val
factor→(exp)	factor.val = exp.val
factor→number	factor.val = number.val

Ques 9) Let consider a grammar as shown below:

decl→type var-list

DOWNLOADED FROM WWW.KTUASSIST.IN

Scanned by CamScanner

type→int | float
var-list→id, var-list | id

Write the attribute grammar.

Ans: The **Attribute Grammar** of this grammar is as given below:

Grammar Rule	Semantic Rules
decl→type var-list	var-list.dtype = type.dtype
type→int	type.dtype = integer
type→ float	type.dtype = real
$var\text{-}list_1$→id, var-list	id.dtype = $var\text{-}list_1$.dtype
	$var\text{-}list_2$.dtype = $var\text{-}list_1$.dtype
var-list→id	id.dtype = var-list.dtype

Ques 10) Discuss about the bottom-up evaluation of S-attributed definitions.

Ans: Bottom-Up Evaluation of S-Attributed Definitions
Synthesised attributes can be evaluated by a bottom-up parser as the input is being parsed. The parser can keep the values of the synthesised attributes associated with the grammar symbols on its stack. Whenever a reduction is made, the values of the new synthesised attributes are computed from the attributes appearing on the stack for the grammar symbols on the right side of the reducing production.

Synthesised Attributes on the Parser Stack
1) A translator for S-attributed definition is implemented using LR parser generator.
2) A bottom up method is used to parse the input string.
3) A parser stack is used to hold the values of synthesised attribute.

The stack is implemented as a pair of state and value. Each state entry is the pointer to the LR (1) parsing table. There is no need to store the grammar symbol implicitly in the parser stack at the state entry. But for ease of understanding we will refer the state by unique grammar symbol that is been placed in the parser stack.

Hence parser stack can be denoted as stack[i]. And stack[i] is a combination of state[i] and value[i]. **For example,** for the production rule X → ABC the stack can be as shown in **figure. 4.3**.

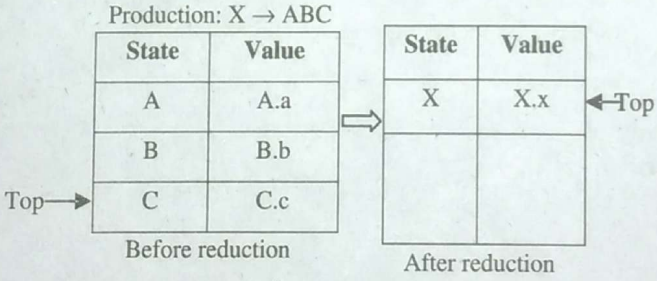


Figure 4.3: Parser Stack

The top symbol on the stack is pointed by pointer top.

Production Rule	Semantic Action
X → ABC	X. x = f (A. a, B. b, C. c)

Before reduction the states A, B and C can be inserted in the stack alongwith the values A.a, B.b, and C.c. The top pointer of value[top] will point the value C.c, similarly B.b is in value[top-1] and A.a is in value[top-2].

After reduction the left hand side symbol of the production, i.e., X will be placed in the stack alongwith the value X.x at the top. Hence after reduction value[top] = X.x.

4) After reduction top is decremented by 2 the state covering X is placed at the top of state[top] and value of synthesised attribute X.x is put in value[top].
5) If the symbol has no attribute then the corresponding entry in the value array will be kept undefined.

DOWNLOADED FROM WWW.KTUASSIST.IN
Scanned by CamScanner

Ques 11) For the following given grammar construct the syntax directed definition and generate the code fragment (translator) using S-attributed definition.

S → EN
E → E + T
E → E – T
E → T
T → T*F
T → T/F
T → F
F → (E)
F → digit
N → ;

Also evaluate the input string 2*3+4; with parser stack using LR parsing method.

Ans:

1) The syntax directed definition for the given grammar can be written as follows:

Production Rule	Semantic Actions
S → EN	Print (E.val)
E → E_1 + T	E.val=E_1.Val+T.val
E → E_1 – T	E.val:=E_1.Val–T.val
E → T	E.val:=T.val
T → T_1*F	T.val:=T_1.val X F.val
T → T_1/F	T.val:=T_1.val/F.val
T → F	T.val:=F.val
F → (E)	F.val:=E.val
F → digit	F.val=digit.lexval
N → ;	Can be ignored by lexical analyser As ; is terminating symbol.

2) The LR parser table can be generated.
3) To evaluate the attributes the code, fragment can be generated by using the parser stack. The appropriate reduction of each production and corresponding code fragment is as given below:

Production Rule	Code Fragment
S → EN	Print(value[top])
E → E_1 + T	value[top]:=value[top-2]+value[top]
E → E_1 – T	value[top]:=value[top-2]–value[top]
E → T	
T → T_1*F	value[top]:=value[top-2]*value[top]
T → T_1/F	value[top]:=value[top-2]/value[top]
T → F	
F → (E)	value[top]:=value[top-1]
F → digit	
N → ;	

4) The sequence of moves made by the parser for the input 2*3+4; are as given below:

Input String	State	Value	Production Rule Used
2*3+4;	–	–	
*3+4;	2	2	
*3+4;	F	2	F → digit

DOWNLOADED FROM WWW.KTUASSIST.IN

Scanned by CamScanner

*3+4;	T	2	T → F
3+4;	T*	2 –	
+4;	T * 3	2 – 3	
+4;	T * F	2 – 3	F → digit
+4;	T	6	T → T*F
+4;	E	6	E → T
4;	E +	6 –	
;	E + 4	6 – 4	
;	E + 4	6 – 4	
;	E + F	6 – 4	F → digit
;	E + T	6 – 4	T → F
;	E	10	E → E + T
	E;	10 –	
	EN	10	
	S	10	S → EN

On seeing the first input symbol 2, initially the symbol 2 is recognised as digit and the parser shifts F in the state stack F corresponding to digit and the semantic action F.val=digit.lexval will be implemented and the value[top] becomes = 2. In the next move parser reduces by T → F.

As no code fragment is associated with this production the value[top] and state[top] is left unchanged. Continuing in this fashion the evaluation of the input string is done and the parser halts successfully when it reaches to state[top]=S the start state.

In this way the bottom up evaluation of S-attributed definitions is done.

Ques 12) What is difference between synthesised attribute and inherited attribute?

Ans: Difference between Synthesised Attribute and Inherited Attribute
Table 4.2 shows the differences between synthesised and inherited attribute:

Table 4.2: Differences between Synthesised Attribute and Inherited Attribute

Synthesised Attribute	Inherited Attribute
The value of synthesised attribute at a node is computed from the values of attribute of children of the node in the parse tree.	The value of inherited attribute is computed from the values of attributes of siblings.
S-attribute can be evaluated during bottom-up parsing (traversal) of a parse tree	Inherited attributes can be evaluated during top-down traversal of a parse tree.
Synthesised attributes pass information up a parse tree.	Inherited attributes pass on information down the parse tree.
S-attributes are also called reference attributes (call by reference).	Inherited attributes are called value attributes (call by value).

Ques 13) What is L-Attributed Definitions?

Ans: L-Attributed Definitions
A syntax-directed definition is L-attributed if for every production $A \rightarrow X_1 X_2 \rightarrow X_n$, and each inherited attribute of X_j for i lies between 1 and n,

1) The attributes (both inherited as well as synthesised) of the symbols $X_1, X_2, ..., X_{j-1}$ (i.e., the symbols to the left of X_j in the production, and
2) The inherited attributes of A.

The syntax-directed definition above is an example of the L-attributed definition, because the inherited attribute. L.type depends on T.type, and T is to the left of L in the production D → TL. Similarly, the inherited attribute L_1.type depends on the inherited attribute L.type, and L is parent of L_1 in the production $L \rightarrow L_1$,id.

DOWNLOADED FROM WWW.KTUASSIST.IN

Scanned by CamScanner

When translation carried-out during parsing, the order in which the semantic rules are evaluated by the parser must be explicitly specified. Hence, instead of using the syntax-directed definitions, we use syntax-directed translation schemes to specify the translations.

Syntax-directed definitions are more abstract specifications for translations; therefore, they hide many implementations details, freeing the user from having to explicitly specify the order in which translation takes place.

Whereas the syntax-directed translation schemes indicate the order in which semantic rules are evaluated, allowing some implementation details to be specified.

Ques 14) Check whether the given SDD (Syntax Directed Definition) is L-attributed or not.

$A \rightarrow PQ$	P.in:=p(A.in)
	Q.in:=q(P.sy)
	A.sy:=f(Q.sy)
$A \rightarrow XY$	Y.in:=y(A.in)
	X.in:=x(Y.sy)
	A.sy:=f(X.sy)

Ans: The attributes 'in' and 'sy' represent the inherited and synthesised attributed respectively. The given syntax directed definition is not L-attributed definition.

Production	Semantic Action	Class of Attribute
$A \rightarrow PQ$	P.in := p(A.in)	L-attribute
	Q.in := q(P.sy)	L-attribute
	A.sy := f(Q.sy)	L-attribute
$A \rightarrow XY$	Y.in := y(A.in)	L-attribute
	X.in := x (Y.sy)	Not L-attribute
	A.sy := f(X.sy)	L-attribute

Because here value of left symbol (X) is dependent upon value of right symbol (i.e., Y)

This is because of the definition X.in:=x(Y.sy). This semantic action suggests that value of X.in depends upon value of Y.sy. That means value of left symbol is dependent on the value of the right symbol.

This violets the 1st rule of the L-attributed definition. [Logically also while parsing the scan is done from left to right and not from right to left] Thus X.in :x(Y.sy) is not L-attribute.

Ques 15) What are the main problems associated with attribute grammar?

Ans: Problems with Attribute Grammar

1) Handling non-local information using complex pattern of information flow:
   i) Requires access to non-local information, and
   ii) Uses global data, (e.g., symbol table) to create side-effects for some of the semantics actions.

   For example,
   a) Checking define/use of variable,
   b) Checking the type and storage address of a variable, and
   c) Checking whether a variable is used or not.
2) Storage management needed to handle large number of attributes.
3) Locating the evaluation result require traversing the tree.

Ques 16) Explain top down translation in detail.

Ans: Top-Down Translation

Let us now discuss about the processing of the translation scheme rather than discussing the syntax directed definitions. And thereby we will actually understand how the order of evaluation takes place with the help of semantic action and associated attributes. Let us take an example to understand the implementation of L-attributed definition using top-down translation scheme.

For example, the system directed definition for determining the value of arithmetic expression is as given below. We will give the translation scheme and draw the annotated tree.

DOWNLOADED FROM WWW.KTUASSIST.IN

Scanned by CamScanner

Production	Semantic Action
$E \rightarrow E_1 + T$	{E.val:=E_1.val + T.val}
$E \rightarrow E_1 - T$	{E.val:=E_1.val – T.val}
$E \rightarrow T$	{E.val:=T.val}
$T \rightarrow (E)$	{T.val:=E.val}
$T \rightarrow$ digit	{T.val:=digit.lexval}

First remove the left recursion and rewrite the translation scheme for non-left recursive grammar.

Production	Semantic Rule
E → T P	{P.in:=T.val} {E.val:=P.s}
P → +T P_1	{P_1.in:=P.in + T.val} {P.s:=P_1.s}
P → –T P_1	{P_1.in:=P.in -T.val} {P.s:=P_1.s}
P → ε	{P.s:=P.in}
T → (E)	{T.val:= E.val}
T → digit	{T.val:=digit.lexval}

Now the annotated parse tree can be drawn in **figure 4.4**.

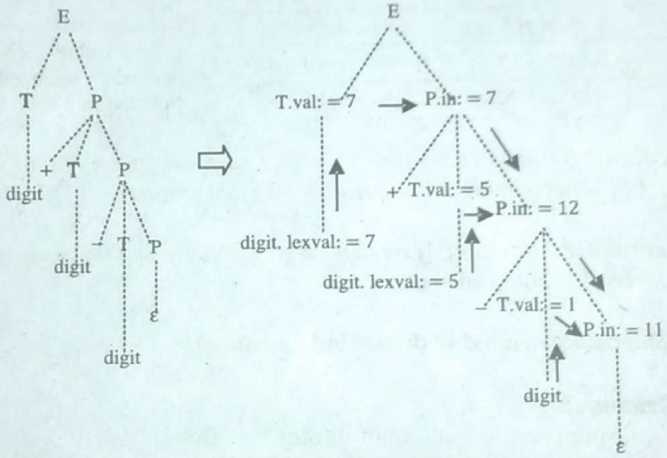


Figure. 4.4: Top Down Translation

Here 'val' and 's' represent the synthesised attributes and 'in' represents the inherited attribute. In translation scheme the computation of inherited attribute must be done by an action appearing before the symbol.

For example,

$$E \rightarrow T\{P.in: = T.val\}\ P\{E.val: = P.s\}$$

Just before P action appears

The inherited attribute for P is first computed then P symbol appears in the rule. Similarly synthesized attribute of non-terminal appearing on the left must be computed after computation of all the attributes on which it is dependent.

For example, consider a semantic rule,
E.val := P.s.

Thus to get the value of E we have computed synthesised attribute of P.

DOWNLOADED FROM WWW.KTUASSIST.IN
Scanned by CamScanner

In the parse tree the top-down translation takes place. The dotted lines show the parse tree whereas the solid line shows the way of computing values of expression. The final result is obtained at P.in:=11 at the bottom node of P.in the translation scheme the second parsing action of first production rule suggest E.val:=P.s. The P.s becomes 11 and that value is copied at the root being as value of E.

Ques 17) Discuss the bottom-up evaluation of inherited attributes.
Or
Discuss translation with inherited attributes.

Ans: Bottom-Up Evaluation of Inherited Attributes
Using a bottom-up translation scheme, one can implement any L-attributed definition based on LL (1) grammar. He/she can also implement some of L-attributed definitions based on LR (1) using bottom-up translations scheme:
1) The semantic actions are evaluated during the reductions.
2) During the bottom-up evaluation of S-attributed definitions, one has a parallel stack to hold synthesised attributes.

One will convert grammar to an equivalent grammar to guarantee the following:
1) All embedding semantic actions in our translation scheme will be moved to the end of the production rules.
2) All inherited attributes will be copied into the synthesised attributes (may be new non-terminals).

Thus one will evaluate all semantic actions during reductions, and he/she find a place to store an inherited attribute. The **steps** are as follows:

Step 1) Remove an embedding semantic action, S_i, put new non-terminal M_i instead of that semantic action.

Step 2) Put S_i into the end of a new production rule $M_i \rightarrow \in$.

Step 3) Semantic action S_i will be evaluated when this new production rule is reduced.

Step 4) Evaluation order of semantic rules is not changed. i.e., if
$A \rightarrow \{S_i\}\ X_1\ \{S_2\}\ X_2 \ldots\ldots \{S_n\}\ X_n$

After removing embedding semantic actions:
$A \rightarrow M_1\ X_1\ M_2\ X_2 \ldots M_n\ X_n$
$M_1 \rightarrow \in \{S_1\}$
$M_2 \rightarrow \in \{S_2\}$
.
.
.
$M_n \rightarrow \in \{S_n\}$

For example,
$E \rightarrow TR$
$R \rightarrow +T\ \{\text{print ('+')}\}\ R_1$
$R \rightarrow \in$
$T \rightarrow \text{id}\ \{\text{print (id.name)}\}$

⇓ Remove embedding semantic actions
$E \rightarrow TR$
$R \rightarrow +\ TMR_1$
$R \rightarrow \in$
$T \rightarrow \text{id}\ \{\text{print (id.name)}\}$
$M \rightarrow \in \{\text{print ('+')}\}$

Translation with Inherited Attributes
Let us assume that every non-terminal A has an inherited attribute A.i and every symbol X has a synthesised attribute X.s in our grammar.

For every production rule $A \rightarrow X_1, X_2 \ldots\ldots X_n$, introduce new market non-terminals,

$M_1, M_2 \ldots\ldots M_n$ and replace this production rule with $A \rightarrow M_1\ X_1\ M_2\ X_2 \ldots M_n\ X_n$

The synthesised attribute of X_i will not be changed.

DOWNLOADED FROM WWW.KTUASSIST.IN
Scanned by CamScanner

The inherited attribute of X_i will be copied into the synthesised attribute of M_1 by the new semantic action added at the end of the new production rule

$M_i \rightarrow \in$

Now, the inherited attribute of X_i can be found in the synthesised attribute of M_i.

$A \rightarrow \{B.i = f_1(..)\ B\ \{c.i = f_2(..)\}\ c\ \{A.s = f_3(..)\}$

$\Downarrow$

$A \rightarrow \{M_1.i = f_1(..)\}\ M_1\ \{B.i = M_1.s\}\ B\ \{M_2.i = f_2(..)\}\ M_2$
$\{c.i = M_2.S\}\ c\ \{A.s = f_3(..)\}$
$M_1 \rightarrow \in\ \{M_1.s = M_1.i\}$
$M_2 \rightarrow \in\ \{M_2.s = M_2.i\}$

TYPE CHECKING

Ques 18) What is type checking?

Ans: Type Checking

Type checking is one of the most important semantic aspects of compilation. Essentially, type checking:

1) Allows the programmer to limit what types may be used in certain circumstances,
2) Aligns types to values, and
3) Determines whether these values are used in an appropriate manner.

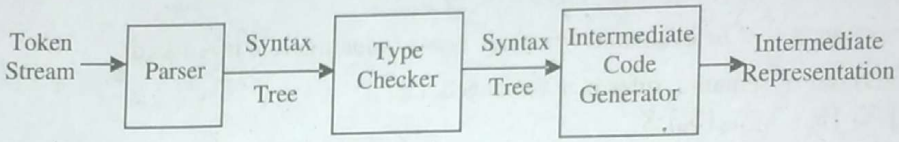


Figure 4.5: Position of Type Checker

Apart from verifying the code to be correct, like checking if function calls have the correct number and types of parameters, type checking also helps in deciding which code to be generated as in case of arithmetic expressions.

Type checking is involved in large parts of the annotated syntax tree. **For example,** the language rules usually specify which types can be combined with a certain operator, there are rules for formal and actual parameter types in a routine call; and assigning an expression value to a variable restricts the allowed types for the expression. Type information in a compiler has to be implemented in such a way that all these and many other checks can be performed conveniently.

Ques 19) What are the different approaches of type checking?

Or

Explain static and dynamic type checking.

Ans: Approaches of Type Checking

The process of verifying and enforcing the constraints of types – type checking – may occur either at compile-time (a static check) or run-time (a dynamic check). That is, there are two approaches to type checking:

1) **Static Type Checking:** It is carried-out at compile time. It must be possible to compute all the information required at compile time. That is, static type checking becomes a primary task of the semantic analysis carried-out by a compiler during compilation. If a language enforces type rules strongly (i.e., allowing only those automatic type conversions which do not lose information), one can refer to the process as strongly typed; if not, it is called weakly typed.

   Examples of static checks include:

   i) **Type Checks:** A compiler should report an error if an operator is applied to an incompatible operand; e.g., if an array variable and a function variable are added together.

   ii) **Flow-of-Control Checks:** Statements that cause flow of control to leave a construct must have some place to which to transfer the flow of control. **For example,** a break statement in C causes control to leave the smallest enclosing while, for, or switch statement; an error occurs if such an enclosing statement does not exist.

   iii) **Uniqueness Checks:** There are situations in which an object must be defined exactly once. **For example,** in Pascal, an identifier must be declared uniquely, labels in a case statement must be distinct, and elements in a scalar type may not be repeated.

DOWNLOADED FROM WWW.KTUASSIST.IN
Scanned by CamScanner

iv) **Name-Related Checks:** Sometimes, the same name must appear two or more times. **For example,** in Ada, a loop or block may have a name that appears at the beginning and end of the construct. The compiler must check that the same name is used at both places.

For example, C compilers apply static type checking during translation, but C is not really strongly typed since many type inconsistencies do not cause compilation errors but are automatically removed through the generation of conversion code, either with or without a warning message. Most modern compilers, however, have error level settings that do provide stronger typing if it is desired, C++ also adds stronger type checking to C, but also mainly in the form of compiler warnings rather than errors (for compatibility with C). Thus, in C++ (and to a certain extent also in C), many type errors appear only as warnings and do not prevent execution. Thus, ignoring warnings can be a "dangerous folly".

2) **Dynamic Type Checking:** Type checking is carried-out while the program is running. This is obviously less efficient, but if a language permits the type of variable to be determined at run time, then one must use dynamic type checking. (SNOBOL4 is an example of such a language).

In dynamic typing, type checking often takes place at run-time because variables can acquire different types depending on the execution path. Static type systems for dynamic types usually need to explicitly represent the concept of an execution path, and allow types to depend on it.

Dynamic typing often occurs in "scripting languages" and other rapid application development languages. Dynamic types appear more often in interpreted languages, whereas compiled languages favour static types.

Ques 20) What are the rules of type checking?

Ans: Rules of Type Checking

Type checking can take on two forms:

1) **Type Synthesis:** It builds-up the type of an expression from the types of its sub-expressions. It requires names to be declared before they are used.

For example, the type of $E_1 + E_2$ is defined in terms of the types of E_1 and E_2. A typical rule for type synthesis has the form:

if f has type s → t and x has type s,
then expression f(x) has type t .....(1)

Here, f and x denote expressions, and s → t denotes a function from s to t. This rule for functions, with one argument carries over to functions with several arguments. The rule (1) can be adapted for $E_1 + E_2$ by viewing it as a function application add (E_1, E_2).

2) **Type Inference:** It determines the type of a language construct from the way it is used. Let null be a function that tests whether a list is empty. Then, from the usage null(x), we can tell that x must be a list. The type of the elements of x is not known; all we know is that x must be a list of elements of some type that is presently unknown.

Variables representing type expressions allow us to talk about unknown types. We shall use Greek letters α, β,...for type variables in type expressions.

A typical rule for type inference has the form:
if f(x) is an expression, ...(2)
then for some α and β, f has type α → β and x has type α

Type inference is needed for languages like ML, which check types, but do not require names to be declared.

Ques 21) What is type system? What are the components of type system?

Ans: Type System

Most programming languages associate a collection of properties with each data value. This collection of properties is called **value's type**. It includes:

1) **Properties:** The type specifies a set of properties held in common by all values of that type. Types can be specified by membership; e.g., an integer might be any whole number i in the range $-2^{31} \le i < 2^{31}$, or red might be a value in an enumerated type colors, defined as the set {red, orange, yellow, green, blue, brown, black, white}.

DOWNLOADED FROM WWW.KTUASSIST.IN

Scanned by CamScanner

2) **Rules:** Types can be specified by rules; e.g., the declaration of a structure in C defines a type. In this case, the type includes any object with the declared fields in the declared order; the individual fields have types that specify the allowable ranges of values and their interpretation.

The set of types in a programming language, alongwith the rules that use types to specify program behaviour, are collectively called a **type system**.

Component of Type System

A type system for a typical modern language has four major components:

1) A set of base types, or built-in types
2) Rules for constructing new types from the existing types
3) A method for determining if two types are equivalent or compatible
4) Rules for inferring the type of each source-language expression.

Ques 22) What is the purpose of type system?

Ans: Purpose of Type System

1) **Ensuring Run-Time Safety:** A well-designed type system helps the compiler detect and avoid run-time errors. The type system should ensure that programs are well behaved – i.e., the compiler and run-time system can identify all ill-formed programs before they execute an operation that causes a run-time error.

   The compiler should eliminate as many run-time errors as it can using type-checking techniques. To accomplish this, the compiler must first infer a type for each expression. These inferred types expose situations in which a value is incorrectly interpreted, such as using a floating-point number in place of a Boolean value.

2) **Improving Expressiveness:** A well-constructed type system allows the language designer to specify behaviour more precisely than is possible with context-free rules. This capability lets the language designer include features that would be impossible to specify in a context-free grammar.

   An excellent example is operator overloading, which gives context-dependent meanings to an operator. Many programming languages use + to signify several kinds of addition.

   The interpretation of + depends on the types of its operands. In typed languages, many operators are overloaded. The alternative, in an untyped language, is to provide lexically different operators for each case.

3) **Generating Better Code:** A well-designed type system provides the compiler with detailed information about every expression in the program – information that can often be used to produce more efficient translations.

Ques 23) What is type conversion?

Ans: Type Conversions

Type conversion refers to the local modification of type for a variable or sub-expression. **For example,** it may be necessary to add an integer quantity to a real variable. However, the language may require both the operands of addition to be of the same type.

Modifying the integer variable to real will require more space, since the real are normally allocated more space than the integers.

Thus, the solution is to treat the integer operand as a real operand locally and perform the operation, whereas, the variable, otherwise, remains to be of type integer.

It may be done explicitly or implicitly:

1) The implicit conversion is called **type-coercion**.
2) In explicit conversion, the programmer writes code to instruct type conversion.

For example, in language C, a programmer can write either of the following two codes:

```
int x;                          int x;
float y;                        float y;
...                             ...
y = ((float)x)/14.0;            y = x/14.0;
```

DOWNLOADED FROM WWW.KTUASSIST.IN

Scanned by CamScanner

Ques 24) Discuss the specification of a simple type checker?

Or

Discuss the type checking of:

1) **A simple Language**
2) **Expressions**
3) **Statements**
4) **Functions**

Ans: Specification of a Simple Type Checker

A type checker for a simple language checks the type of each identifier. The type checker is a translation scheme that synthesizes the type of each expression from the types of its sub-expressions. The type checker can handle arrays, pointers, statements and functions.

Simple Language

Consider the grammar shown in **figure 4.6**.

$P \rightarrow D; E$
$D \rightarrow D; D \mid id : T$
$T \rightarrow char \mid integer \mid array\ [num]\ of\ T \mid \uparrow T$
$E \rightarrow literal \mid num \mid id \mid E\ mod\ E \mid E[E] \mid E\uparrow$

Figure 4.6: Grammar

Take a minute and discuss what can be in this language. Note that the declarations have to come before the usage of the variable.

The basic types are – character and integer. The constructed types are – array and pointer. The attribute type is added to each symbol. The translation scheme is shown in **figure 4.7**.

Production	Semantic Rules
$P \rightarrow D; E$	
$D \rightarrow D; D$	
$D \rightarrow id : T$	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(T_1.type)
$T \rightarrow array\ [num]\ of\ T_1$	T.type = array(num.val, T_1.type)

Figure 4.7 Type Checking Scheme

Type Checking of Expressions

Consider **figure 4.8** which performs the type checking for expressions.

Production	Semantic Rules
$E \rightarrow literal$	E.type = char
$E \rightarrow num$	E.type = integer
$E \rightarrow id$	E.type=lookup (id.entry)
$E \rightarrow E_1\ mod\ E_2$	E.type = if E_1.type = integer and E_2.type = integer then integer else type-error
$E \rightarrow E_1\ [E_2]$	E.type = if E_2.type = integer and E_1.type = array (s, t) then t else type _error
$E \rightarrow E_1 \uparrow$	E.type = if E_1.type = pointer (t) then t else type _error

Figure 4.8: Type System for Expressions

Note that the synthesised attribute type for E gives the type of the expression assigned by the type system for the expression generated by E. The function looking returns the type of id.

DOWNLOADED FROM WWW.KTUASSIST.IN

Scanned by CamScanner

Type Checking of Statements

Statements do not have values, therefore a special type, void, can be assigned to them. If an error occurs within a statement, the type assigned to the statement is type_error.

Consider the actions for statements shown in **figure 4.9**.

Production	Semantic Rules
$S \rightarrow id = E$	S.type = if id.type = E.type then void else type_error
$S \rightarrow$ if E then S_1	S.type = if E.type = boolean then void else type_error
$S \rightarrow$ while E do S_1	S.type = if E.type = boolean then void else type_error
$S \rightarrow S_1; S_2$	S.type = if S_1.type = void and S_2.type = void then void else type_error

Figure 4.9: Type System for Statements

There is more checking that must occur on assignment statement. A check must be made to tell whether the left hand side can be assigned to. **For example,** one is not able to assign to a constant.

Type Checking Functions

The application of functions can be seen in **figure 4.10**.

Production	Semantic Rules
$T \rightarrow T_1$ " $\rightarrow$ " T_2	$T.type = T_1.type \rightarrow T_2.type$
$E \rightarrow E_1 (E_2)$	E.type = if E_2.type = s and E_1.type = $s \rightarrow t$ then t else type_error

Figure 4.10: Type System for Functions

Notice that a function declaration would look like

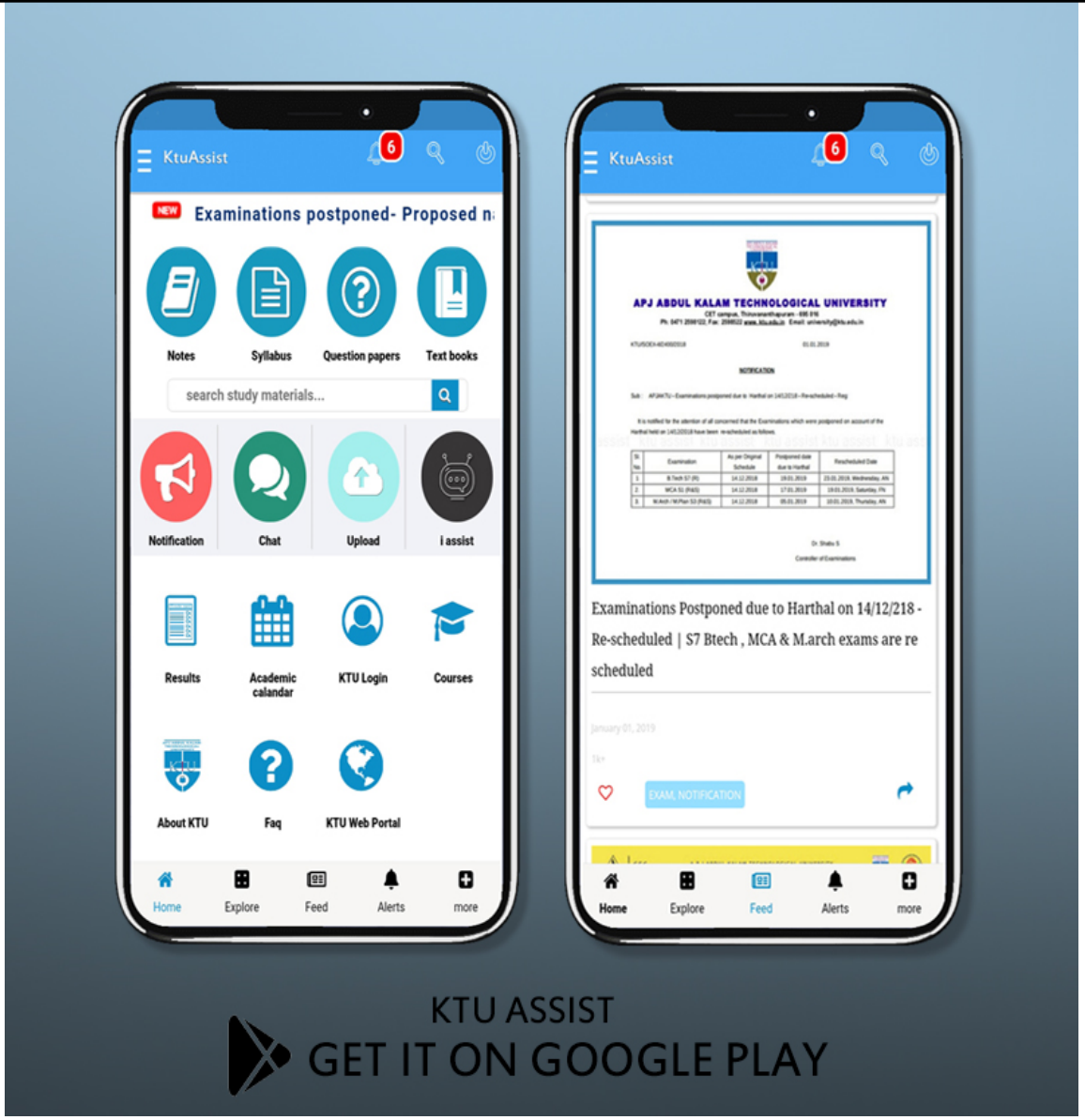
root: (real $\rightarrow$ real) $\times$ real $\rightarrow$ real

and would look like the following in Pascal.

function root (function f (real) : real; x : real) : real

DOWNLOADED FROM WWW.KTUASSIST.IN

Scanned by CamScanner



END

facebook.com/ktuassist

instagram.com/ktu_assist