CVR COLLEGE OF ENGINEERING
In Pursuit of Excellence
(An Autonomous Institution, NAAC 'A' Grade)

Subject Name : AUTOMATA AND COMPILER DESIGN

Subject Code : 67302

Topic : Unit III - Semantic Analysis

Class : III YEAR - I Sem

Faculty Name : Dr. R RAJA

Designation : Associate Professor

Department : CSIT

Unit III - Semantic Analysis

Syntax directed definition and translation, s-attributed and l-attributed grammars, type checking, type conversion, equivalence of type expressions, Overloading of functions and operators, Chomsky hierarchy of languages and recognizers.

Semantic Analysis

- **Semantic Analysis** computes additional information related to the meaning of the program once the syntactic structure is known.
- In typed languages as C, semantic analysis involves adding information to the symbol table and performing type checking.
- The information to be computed is beyond the capabilities of standard parsing techniques, therefore it is not regarded as syntax.
- As for Lexical and Syntax analysis, also for Semantic Analysis we need both a *Representation Formalism* and an *Implementation Mechanism*.
- As representation formalism this lecture illustrates what are called *Syntax Directed Translations*.

Syntax Directed Translation: Intro

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

Syntax Directed Translation: Intro (Cont.)

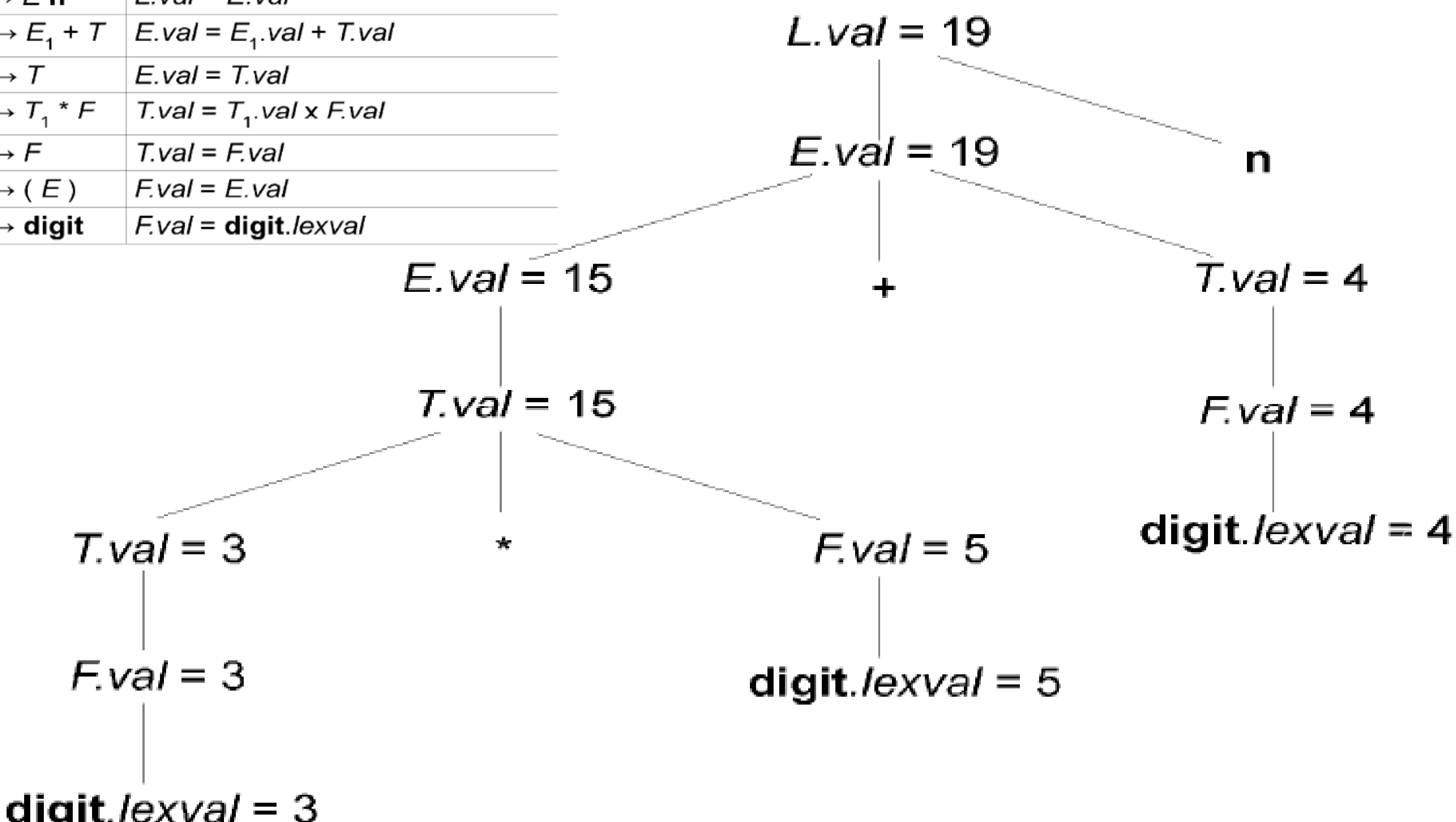
- Evaluation of Semantic Rules may:
  - Generate Code;
  - Insert information into the Symbol Table;
  - Perform Semantic Check;
  - Issue error messages;
  - etc.
- There are two notations for attaching semantic rules:
  1. **Syntax Directed Definitions.** High-level specification hiding many implementation 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

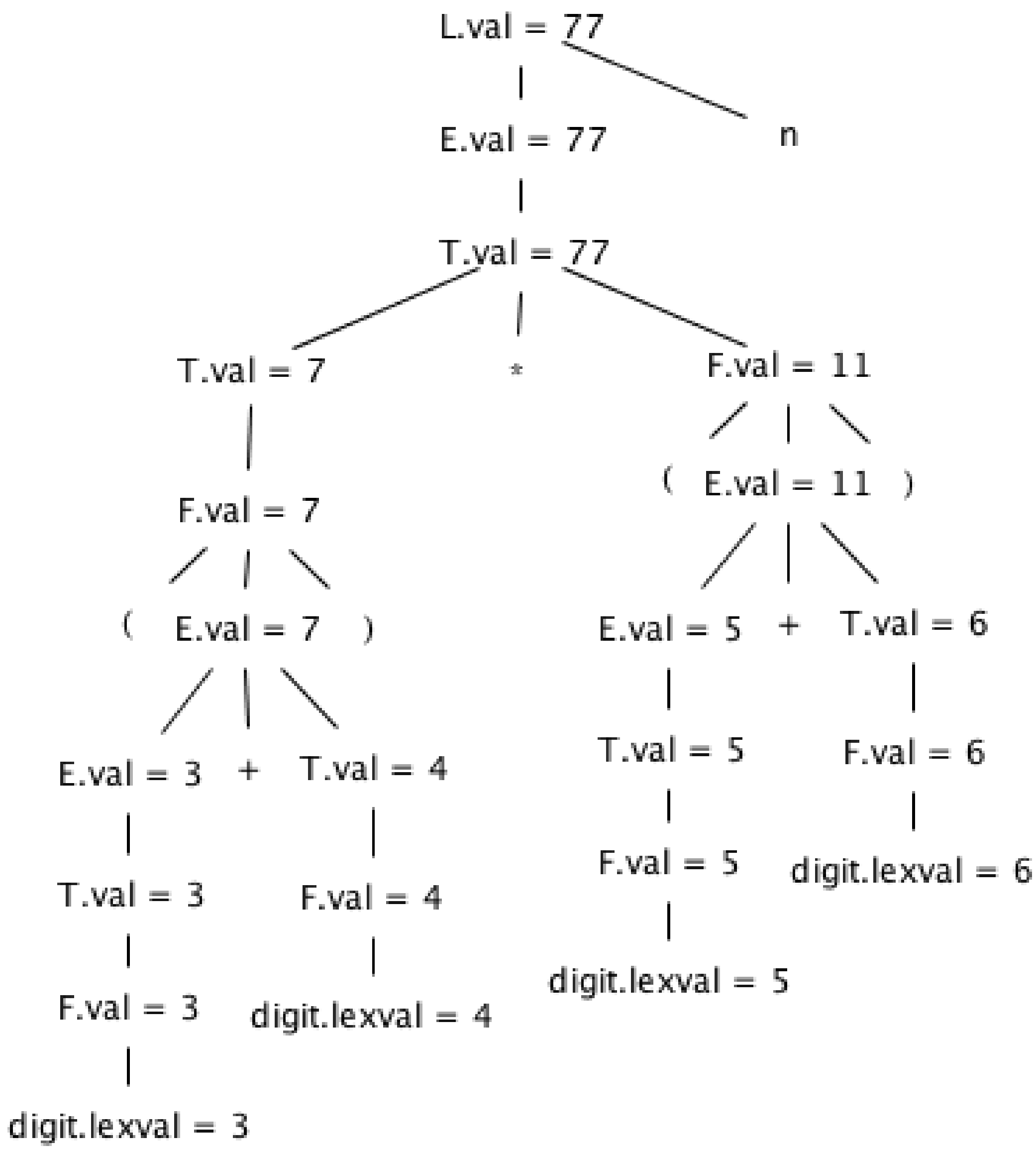
- **Syntax Directed Definitions** are a generalization of context-free grammars in which:
  1. Grammar symbols have an associated set of **Attributes**;
  2. Productions are associated with **Semantic Rules** for computing the values of attributes.
- Such formalism generates **Annotated Parse-Trees** 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).

Annotated Parse Tree for 3*5+4n

Production	Semantic Rules
1) $L \rightarrow E$ **n**	$L.val = E.val$
2) $E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3) $E \rightarrow T$	$E.val = T.val$
4) $T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
5) $T \rightarrow F$	$T.val = F.val$
6) $F \rightarrow (E)$	$F.val = E.val$
7) $F \rightarrow$ **digit**	$F.val =$ **digit**.*lexval*

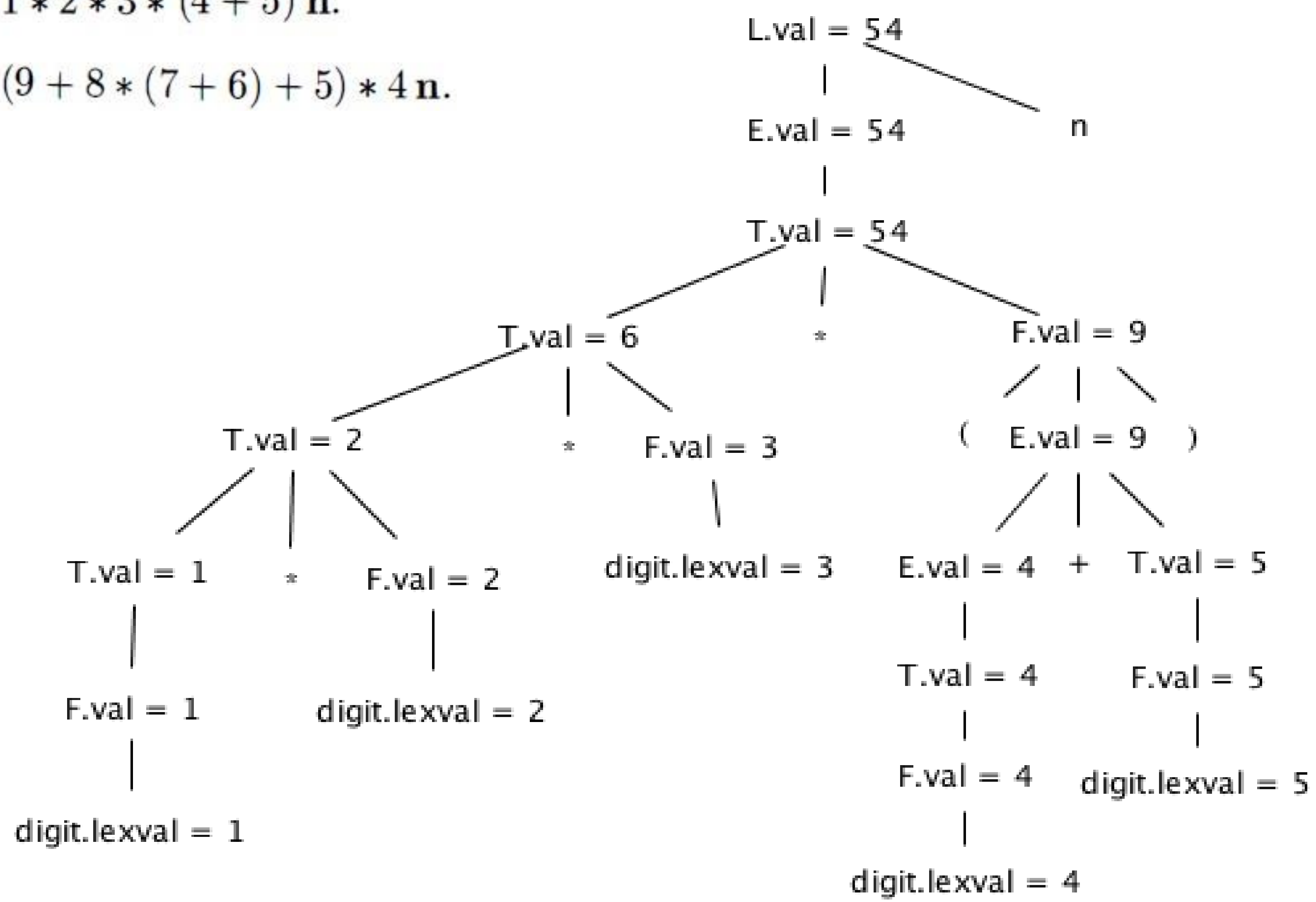


$(3+4)*(5+6)\ \mathbf{n}.$



$1 * 2 * 3 * (4 + 5)\ \mathbf{n}.$

$(9 + 8 * (7 + 6) + 5) * 4\ \mathbf{n}.$



Syntax Directed Definitions (Cont.)

- 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.

Form of Syntax Directed Definitions

- Each production, $A \rightarrow \alpha$, is associated with a set of semantic rules: $b := f(c_1, c_2, \ldots, c_k)$, where f is a function and either

  1. b is a **synthesized** attribute of A, and $c_1, c_2, \ldots, c_k$ are attributes of the grammar symbols of the production, or
  2. b is an **inherited** attribute of a grammar symbol in α, and $c_1, c_2, \ldots, c_k$ are attributes of grammar symbols in α or attributes of A.

- **Note.** Terminal symbols are assumed to have synthesized attributes supplied by the lexical analyzer.

- Procedure calls (e.g. *print* in the next slide) define values of *Dummy* synthesized attributes of the non terminal on the left-hand side of the production.

Syntax Directed Definitions: An Example

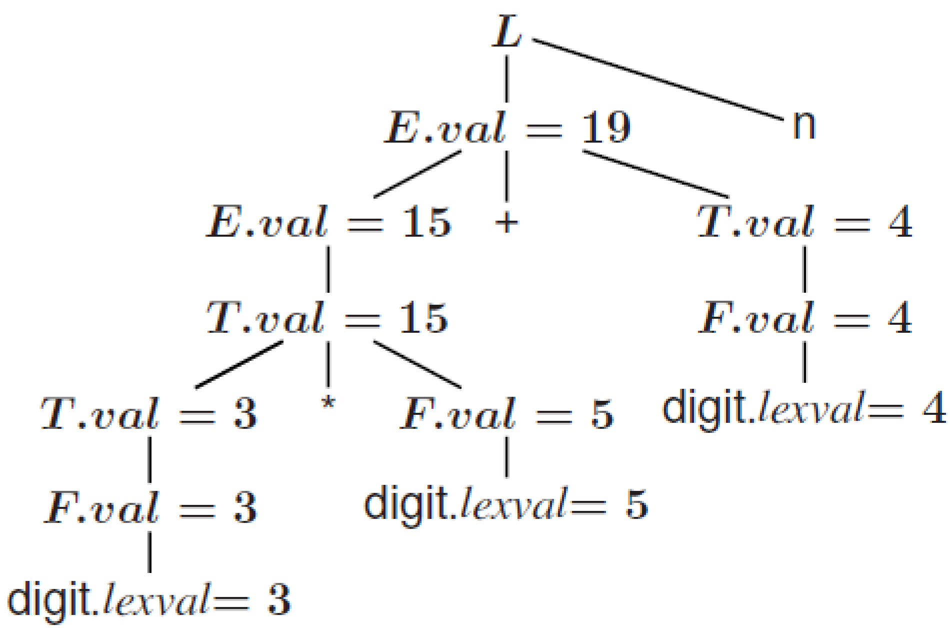
- **Example.** Let us consider the Grammar for arithmetic expressions. The Syntax Directed Definition associates to each non terminal a synthesized attribute called *val*.

PRODUCTION	SEMANTIC RULE
$L \to E\mathsf{n}$	$print(E.val)$
$E \to E_1 + T$	$E.val := E_1.val + T.val$
$E \to T$	$E.val := T.val$
$T \to T_1 * F$	$T.val := T_1.val * F.val$
$T \to F$	$T.val := F.val$
$F \to (E)$	$F.val := E.val$
$F \to \mathsf{digit}$	$F.val := \mathsf{digit}.lexval$

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:



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

Inherited Attributes

- **Inherited Attributes** are useful for expressing the dependence of a construct on the context in which it appears.
- It is always possible to rewrite a syntax directed definition to use only synthesized attributes, but it is often more natural to use both synthesized and inherited attributes.
- **Evaluation Order.** Inherited attributes cannot be evaluated by a simple PreOrder traversal of the parse-tree:
  - Unlike synthesized attributes, the order in which the inherited attributes of the children are computed is important!!! Indeed:
    - Inherited attributes of the children can depend from both left and right siblings!

Inherited Attributes: An Example

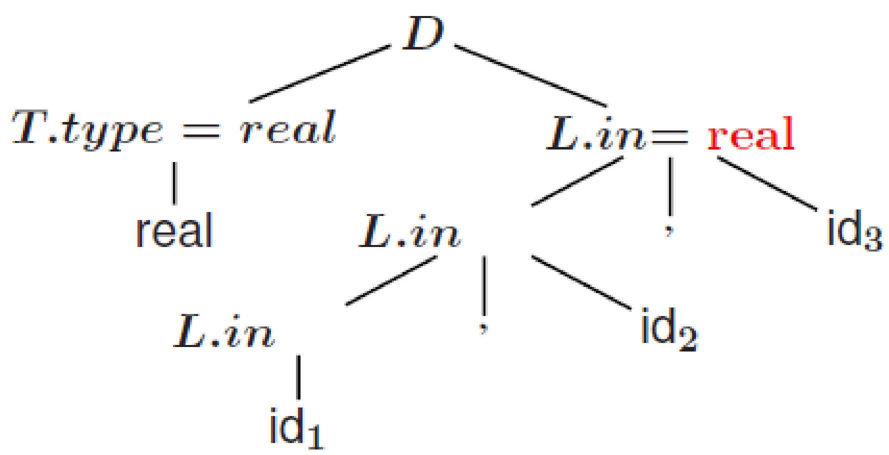
- **Example.** Let us consider the syntax directed definition with both inherited and synthesized attributes for the grammar for "type declarations":

PRODUCTION	SEMANTIC RULE
$D \rightarrow TL$	$L.in := T.type$
$T \rightarrow$ int	$T.type :=$ *integer*
$T \rightarrow$ real	$T.type :=$ *real*
$L \rightarrow L_1$, id	$L_1.in := L.in$; *addtype*(id.*entry*, *L.in*)
$L \rightarrow$ id	*addtype*(id.*entry*, *L.in*)

- The non terminal T has a synthesized attribute, *type*, determined by the keyword in the declaration.
- The production $D \rightarrow TL$ is associated with the semantic rule $L.in := T.type$ which set the *inherited* attribute $L.in$.
- Note: The production $L \rightarrow L_1$, id distinguishes the two occurrences of L.

Inherited Attributes: An Example (Cont.)

- Synthesized attributes can be evaluated by a PostOrder traversal.
- Inherited attributes that *do not depend from right children* can be evaluated by a classical PreOrder traversal.
- The annotated parse-tree for the input real id_1, id_2, id_3 is:



PRODUCTION	SEMANTIC RULE
$D \to TL$	$L.in := T.type$
$T \to$ int	$T.type :=$ *integer*
$T \to$ real	$T.type :=$ *real*
$L \to L_1$, id	$L_1.in := L.in$; *addtype*(id.*entry*, $L.in$)
$L \to$ id	*addtype*(id.*entry*, $L.in$)

- $L.in$ is then inherited top-down the tree by the other L-nodes.
- At each L-node the procedure *addtype* inserts into the symbol table the type of the identifier.

Dependency Graphs

- Implementing a Syntax Directed Definition consists primarily in finding an order for the evaluation of attributes
  - Each attribute value must be available when a computation is performed.
- **Dependency Graphs** are the most general technique used to evaluate syntax directed definitions with both synthesized and inherited attributes.
- A Dependency Graph shows the interdependencies among the attributes of the various nodes of a parse-tree.
  - There is a node for each attribute;
  - If attribute b depends on an attribute c there is a link from the node for c to the node for b ($b \leftarrow c$).
- **Dependency Rule:** If an attribute b depends from an attribute c, then we need to fire the semantic rule for c first and then the semantic rule for b.

Evaluation Order

- The evaluation order of semantic rules depends from a *Topological Sort* derived from the dependency graph.
- **Topological Sort:** Any ordering $m_1, m_2, \ldots, m_k$ such that if $m_i \rightarrow m_j$ is a link in the dependency graph then $m_i < m_j$.
- Any topological sort of a dependency graph gives a valid order to evaluate the semantic rules.

Implementing Attribute Evaluation: General Remarks

- Attributes can be evaluated by building a dependency graph at compile-time and then finding a topological sort.
- **Disavantages**
  1. This method fails if the dependency graph has a cycle: We need a test for non-circularity;
  2. This method is time consuming due to the construction of the dependency graph.
- **Alternative Approach.** Design the syntax directed definition in such a way that attributes can be evaluated with a *fixed order* avoiding to build the dependency graph (method followed by many compilers).

Evaluation of S-Attributed Definitions

- **Synthesized Attributes can be evaluated by a bottom-up parser as the input is being analyzed avoiding the construction of a dependency graph.**
- The parser keeps the values of the synthesized attributes in its stack.
- Whenever a reduction $A \rightarrow \alpha$ is made, the attribute for A is computed from the attributes of α which appear on the stack.
- Thus, a translator for an S-Attributed Definition can be simply implemented by extending the stack of an LR-Parser.

Extending a Parser Stack

- Extra fields are added to the stack to hold the values of synthesized attributes.
- In the simple case of just one attribute per grammar symbol the stack has two fields: *state* and *val*

state	*val*
Z	$Z.x$
Y	$Y.x$
X	$X.x$
$\ldots$	$\ldots$

- The current top of the stack is indicated by the pointer *top*.
- Synthesized attributes are computed just before each reduction:
  - Before the reduction $A \rightarrow XYZ$ is made, the attribute for A is computed: $A.a := f(val[top], val[top-1], val[top-2])$.

Extending a Parser Stack: An Example

- **Example.** Consider the S-attributed definitions for the arithmetic expressions. To evaluate attributes the parser executes the following code

PRODUCTION	CODE
$L \to E\text{n}$	$print(val[top-1])$
$E \to E_1 + T$	$val[ntop] := val[top] + val[top-2]$
$E \to T$	
$T \to T_1 * F$	$val[ntop] := val[top] * val[top-2]$
$T \to F$	
$F \to (E)$	$val[ntop] := val[top-1]$
$F \to \text{digit}$	

- The variable $ntop$ is set to the *new top of the stack*. After a reduction is done *top* is set to *ntop*: When a reduction $A \to \alpha$ is done with $|\alpha| = r$, then $ntop = top - r + 1$.

- During a shift action both the token and its value are pushed into the stack.

Extending a Parser Stack: An Example (Cont.)

- The following Figure shows the moves made by the parser on input 3*5+4n.
  - Stack states are replaced by their corresponding grammar symbol;
  - Instead of the token digit the actual value is shown.

INPUT	*state*	*val*	PRODUCTION USED
3*5+4 n	–	–	
*5+4 n	3	3	
*5+4 n	F	3	$F \to$ **digit**
*5+4 n	T	3	$T \to F$
5+4 n	T *	3 _	
+4 n	T * 5	3 _ 5	
+4 n	T * F	3 _ 5	$F \to$ **digit**
+4 n	T	15	$T \to T * F$
+4 n	E	15	$E \to T$
4 n	E +	15 _	
n	E + 4	15 _ 4	
n	E + F	15 _ 4	$F \to$ **digit**
n	E + T	15 _ 4	$T \to F$
n	E	19	$E \to E + T$
	E n	19 _	
	L	19	$L \to E$ **n**

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

L-Attributed Definitions

- **L-Attributed Definitions** contain both synthesized and inherited attributes but do not need to build a dependency graph to evaluate them.
- **Definition.** A syntax directed definition is *L-Attributed* if each *inherited attribute* of X_j in a production $A \rightarrow X_1 \ldots X_j \ldots X_n$, depends only on:
  1. The attributes of the symbols to the **left** (this is what L in *L-Attributed* stands for) of X_j, i.e., $X_1 X_2 \ldots X_{j-1}$, and
  2. The *inherited* attributes of A.
- **Theorem.** Inherited attributes in L-Attributed Definitions can be computed by a PreOrder traversal of the parse-tree.

Evaluating L-Attributed Definitions

- **L-Attributed Definitions are a class of syntax directed definitions whose attributes can always be evaluated by single traversal of the parse-tree.**
- The following procedure evaluate L-Attributed Definitions by mixing PostOrder (synthesized) and PreOrder (inherited) traversal.

Algorithm: L-Eval(n: Node)

Input: Node of an annotated parse-tree.

Output: Attribute evaluation.

```
Begin
    For each child m of n, from left-to-right Do
    Begin
        Evaluate inherited attributes of m;
        L-Eval(m)
    End;
    Evaluate synthesized attributes of n
End.
```

Translation Schemes

- **Translation Schemes** are more implementation oriented than syntax directed definitions since they **indicate the order** in which semantic rules and attributes are to be evaluated.
- **Definition.** A Translation Scheme is a context-free grammar in which
  1. Attributes are associated with grammar symbols;
  2. Semantic Actions are enclosed between braces {} and **are inserted within the right-hand side of productions.**
- Yacc uses Translation Schemes.

Translation Schemes (Cont.)

- Translation Schemes deal with both synthesized and inherited attributes.
- Semantic Actions are treated as terminal symbols: Annotated parse-trees contain semantic actions as children of the node standing for the corresponding production.
- Translation Schemes are useful to evaluate L-Attributed definitions at parsing time (even if they are a general mechanism).
  - **An L-Attributed Syntax-Directed Definition can be turned into a Translation Scheme.**

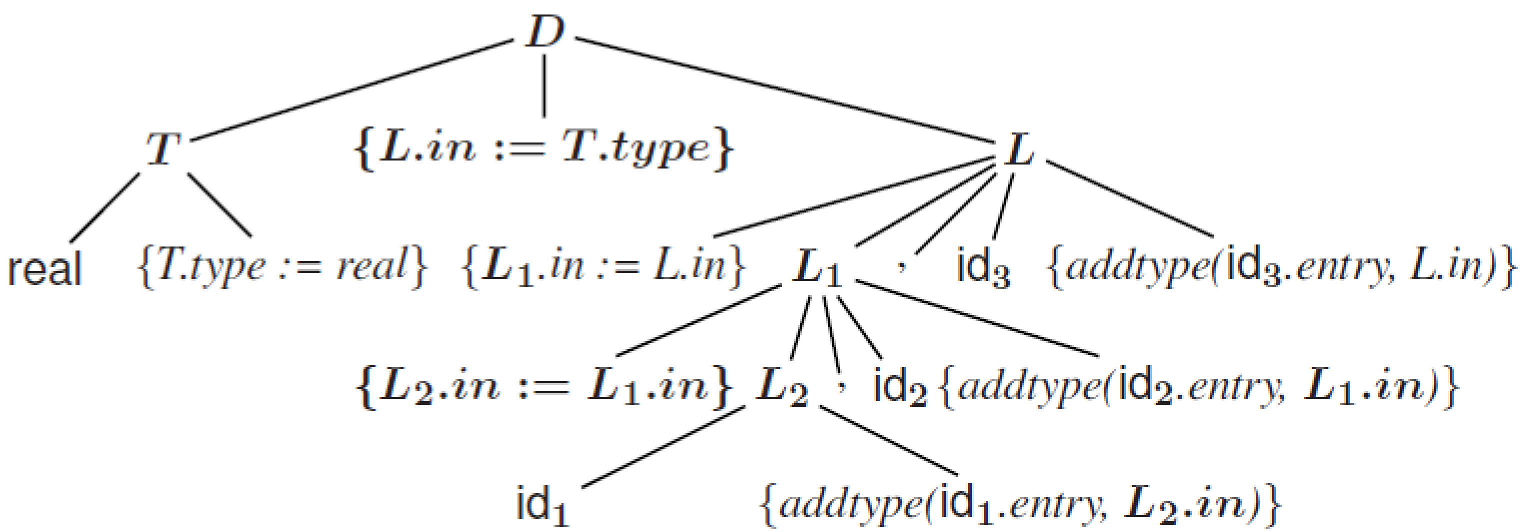
Translation Schemes: An Example

- Consider the Translation Scheme for the L-Attributed Definition for "type declarations":

$$
\begin{aligned}
D &\rightarrow \quad T\ \{L.in := T.type\}\ L \\
T &\rightarrow \quad \text{int}\ \ \{T.type := integer\} \\
T &\rightarrow \quad \text{real}\ \ \{T.type := real\} \\
L &\rightarrow \quad \{L_1.in := L.in\}\ L_1, \text{id}\ \ \{addtype(\text{id}.entry, L.in)\} \\
L &\rightarrow \quad \text{id}\ \ \{addtype(\text{id}.entry, L.in)\}
\end{aligned}
$$

Translation Schemes: An Example (Cont.)

- **Example (Cont).** The parse-tree with semantic actions for the input real id_1, id_2, id_3 is:



- **Traversing the Parse-Tree in depth-first order (PostOrder) we can evaluate the attributes.**

Design of Translation Schemes

- When designing a Translation Scheme we must be sure that an attribute value is available when a semantic action is executed.
- **When the semantic action involves only synthesized attributes: The action can be put at the end of the production.**
  - **Example.** The following Production and Semantic Rule:

    $$T \rightarrow T_1 * F \quad T.val := T_1.val * F.val$$

    yield the translation scheme:

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

Design of Translation Schemes (Cont.)

- **Rules for Implementing L-Attributed SDD's.** If we have an L-Attibuted Syntax-Directed Definition we must enforce the following restrictions:
  1. An inherited attribute for a symbol in the right-hand side of a production must be computed in an action before the symbol;
  2. A synthesized attribute for the non terminal on the left-hand side can only be computed when all the attributes it references have been computed: The action is usually put at the end of the production.

Compile-Time Evaluation of Translation Schemes

- Attributes in a Translation Scheme following the above rules can be computed at compile time similarly to the evaluation of S-Attributed Definitions.
- **Main Idea.** Starting from a Translation Scheme (with embedded actions) we introduce a transformation that makes all the actions occur at the right ends of their productions.
  - For each embedded semantic action we introduce a new *Marker* (i.e., a non terminal, say M) with an empty production ($M \rightarrow \epsilon$);
  - The semantic action is attached at the end of the production $M \rightarrow \epsilon$.

Compile-Time Evaluation of Translation Schemes (Cont.)

- **Example.** Consider the following translation scheme:

  $S \rightarrow aA\{\mathrm{C.i} = \mathrm{f}(\mathbf{A}.\mathrm{s})\}C$

  $S \rightarrow bAB\{\mathrm{C.i} = \mathrm{f}(\mathbf{A}.\mathrm{s})\}C$

  $C \rightarrow c\{C.s = g(C.i)\}$

  Then, we add new markers M_1, M_2 with:

  $S \rightarrow aA\mathrm{M}_1C$

  $S \rightarrow bAB\mathrm{M}_2C$

  $\mathrm{M}_1 \rightarrow \epsilon \qquad \{M_1.s := f(val[top])\}$

  $\mathrm{M}_2 \rightarrow \epsilon \qquad \{M_2.s := f(val[top-1])\}$

  $C \rightarrow c \qquad \{C.s := g(val[top-1])\}$

  The inherited attribute of C is the synthesized attribute of either M_1 or M_2: The value of $C.i$ is *always* in *val*[*top -1*] when $C \rightarrow c$ is applied.

Compile-Time Evaluation of Translation Schemes (Cont.)

General rules to compute translations schemes during bottom-up parsing assuming an L-attributed grammar.

- For every production $A \rightarrow X_1 \ldots X_n$ introduce n new markers $M_1, \ldots, M_n$ and replace the production by $A \rightarrow M_1 X_1 \ldots M_n X_n$.
- Thus, we know the position of every synthesized and inherited attribute of X_j and A:
  1. $X_j.s$ is stored in the *val* entry in the parser stack associated with X_j;
  2. $X_j.i$ is stored in the *val* entry in the parser stack associated with M_j;
  3. $A.i$ is stored in the *val* entry in the parser stack immediately before the position storing M_1.
- **Remark 1.** Since there is only one production for each marker a grammar remains LL(1) with addition of markers.
- **Remark 2.** Adding markers to an LR(1) Grammar can introduce conflicts for not L-Attributed SDD's!!!

Compile-Time Evaluation of Translation Schemes (Cont.)

Example. Computing the inherited attribute $X_j.i$ after reducing with $M_j \to \epsilon$.

	M_j	$X_j.i$
$top \rightarrow$	X_{j-1}	$X_{j-1}.s$
	M_{j-1}	$X_{j-1}.i$
	$\ldots$	$\ldots$
	X_1	$X_1.s$
	M_1	$X_1.i$
$(top\text{-}2j\text{+}2) \rightarrow$	M_A	$A.i$
$(top\text{-}2j) \rightarrow$		

- $A.i$ is in $val[top - 2j + 2]$;
- $X_1.i$ is in $val[top - 2j + 3]$;
- $X_1.s$ is in $val[top - 2j + 4]$;
- $X_2.i$ is in $val[top - 2j + 5]$;
- and so on.

Type Checking

- Parsing cannot detect some errors. Some errors are captured during compile time called static checking.
- Static checking is even called early binding. During static checking programming errors are caught early. This causes program execution to be efficient.
- Static checking not only increases the efficiency and reliability of the compiled program, but also makes execution faster.
- Type checking is not only limited to compile time, it is even performed at execution time. This is done with the help of information gathered by a compiler.
- Languages like C, C++, C#, Java, and Haskell uses static checking.
- Languages like Perl, python, and Lisp use dynamic checking.
- Dynamic checking is also called late binding. Dynamic checking allows some constructs that are rejected during static checking.

Type Checking

- A semantic analyzer mainly performs static checking. Static checks can be any one of the following **type of checks:**
- **Uniqueness checks:** This ensures uniqueness of variables/objects in situations where it is required.
- **Flow of control checks:** Statements that cause flow of control to leave a construct should have a place to transfer flow of control. If this place is missing, it is confusion.
- **Type checks:** A compiler should report an error if an operator is applied to incompatible operands.
- **Name-related checks:** Sometimes, the same name must appear two or more times.

Type Checking

- To do type checking a compiler needs to assign a type expression to each component of the source program.
- The compiler must then determine that these type expressions conform to a collection of logical rules that is called the type system for the source language.
- In principle, any check can be done dynamically, if the target code carries the type of an element along with the value of the element.
- A sound type system eliminates the need for dynamic checking for type errors, because it allows us to determine statically that these errors cannot occur when the target program runs.
- An implementation of a language is strongly typed if a compiler guarantees that the programs it accepts will run without type errors.

Type Checking

- What does semantic analysis do? It performs checks of many kinds which may include
  - All identifiers are declared before being used.
  - Type compatibility.
  - Inheritance relationships.
  - Classes defined only once.
  - Methods in a class defined only once.
  - Reserved words are not misused.
- The above examples indicate that most of the other static checks are routine and can be implemented using the techniques of SDT.
- Some of them can be combined with other activities. For example, for uniqueness check, while entering the identifier into the symbol table, we can ensure that it is entered only once.

Rules for Type Checking

- Type checking can take on two forms: synthesis and inference.
- Type synthesis builds up the type of an expression from the types of its sub expressions. It requires names to be declared before they are used.
- The type of E1 + E2 is defined in terms of the types of E1 and E2. A typical rule for type synthesis has the form

if f has type $s \rightarrow t$ **and** x has type s,
then expression $f(x)$ has type t

- 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 can be adapted for E1+E2 by viewing it as a function application

  add(E1; E2).

Rules for Type Checking

- Type inference determines the type of a language construct from the way it is used.
- Variables representing type expressions allow us to talk about unknown types.

if $f(x)$ is an expression,
then for some α and β, f has type $\alpha \rightarrow \beta$ **and** x has type α

- Type inference is needed for languages like ML, which check types, but do not require names to be declared.
- The rules for checking statements are similar to those for expressions. For example, we treat the conditional statement **"if (E) S;"** as if it were the application of a function if to E and S.
- Let the special type void denote the absence of a value. Then function if expects to be applied to a boolean and a void; the result of the application is a void.

Type Systems

- Consider the assembly language program fragment.

  Add R1, R2, and R3. What are the types of operands R1, R2, R3?

- Based on the possible type of operands and its values, operations are legal.
- It doesn't make sense to add a character and a function pointer in C language. It does make sense to add two float or int values.
- A language's type system specifies which operations are valid for which types. A type system is a collection of rules for assigning types to the various parts of a program.
- A type checker implements a type system. Types are represented by type expressions.
- Type system has a set of rules defined that take care of extracting the data types of each variables and check for the compatibility during the operation.

Type Expressions

- The type expressions are used to represent the type of a programming language construct.
- Type expression can be a basic type or formed by recursively applying an operator called a type constructor to other type expressions.
- The basic types and constructors depend on the source language to be verified. **Let us define type expression as follows:**
  - A basic type is a type expression
  - Boolean, char, integer, real, void, type_error
  - A type constructor applied to type expressions is a type expression

Type Expressions

Array: array(I, T)

- Array (I,T) is a type expression denoting the type of an array with elements of type T and index set I, where T is a type expression. Index set I often represents a range of integers. For example, the Pascal declaration

var C: array[1..20] of integer;

associates the type expression array(1..20, integer) with C.

Product: T1 × T2

- If T1 and T2 are two type expressions, then their Cartesian product T1 × T2 is a type expression. We assume that × associates to the left.

Record: record((N1 × T1) × (N2 × T2))

- A record differs from a product. The fields of a record have names. The record type constructor will be applied to a tuple formed from field types and field names.

Design Of Simple Type Checker

- Let us consider a simple language that has declaration statements followed by statements, where these statements are simple arithmetic statements, conditional statements, iterative statements, and functional statements.
- The program block of code can be generated by defining the rules as follows:

Type Declarations

	P → D ";" E	
	D → D ";" D	
	| id ":" T	{add_type(id.entry, T.type) }
	T → char	{T.type := char }
	T → integer	{T.type := int }
	:…..	:…..
	T → "*" T_1	{T.type := pointer(T_1.type) }
	T → array "["num "]" of T_1	{T.type := array(num.value, T_1.type) }

So the above SDT collects type information and stores in symbol table.

TYPE CHECKING OF EXPRESSIONS

- The expressions like 3 mod 5, A[10], *p can be generated by the following rules.
- The semantic rules are defined as follows to extract the type information and to check for compatibility.

- write a statement as **i mod 10,** then
- while parsing it uses the rule as E → id and performs the action.
- When it parses the lexeme 10, it uses the rule E → num
- While parsing the complete statement i mod 10**,** it uses the rule E → E1 mod E2.

	E → literal	{E.type := char}
	E → num	{E.type := int}
	E → id	{E.type := lookup(id.entry)}
	E → E1 mod E2	{E.type := if E1.type = int and E2.type = int
		then int
		else type_error}
	E → E1 "[" E2 "]"	{E.type := if E1.type = array(s, t) and E2.type = int
		Th en t
		else type_error}
	E → "*" E1	{E.type := if E1.type = pointer(t)
		then t
		else type_error}

Table 3.2: Type Checking Of Expressions

TYPE CHECKING OF STATEMENTS

- The statements are simple of the form "a = b + c" or "a = b." It can be a combination of statements followed by another statement or a conditional statement or iterative
- To generate either a simple or a complex group of statements, the rules can be framed as follows:
  - To validate the statement a special data type void is defined, which is assigned to a statement only when it is valid at expression level,
  - otherwise type_error is assigned to indicate that it is invalid.
  - If there is an error at expression level, then it is propagated to the statement, from the statement it is propagated to a set of statements and then to the entire block of program.
- The semantic rules are defined as follows to extract the type information and to check for compatibility.

TYPE CHECKING OF STATEMENTS

- The semantic rules are defined as follows to extract the type information and to check for compatibility.

P → D ";" S	
S → id ":=" E	{S.type := if lookup(id.entry)= E.type
S → S_1 ";" S_2	then void
	else type_error}
	{S.type := if S_1.type = void and S_2.type
	= void
	then void
	else type_error}
S → if E then S_1	{S.type := if E.type = boolean
	then S_1.type
	else type_error}
S → while E do S_1	{S.type := if E.type = boolean
	then S_1.type
	else type_error}

Table 3.3: Type Checking Of Statements

Type Conversion

- In an expression, if there are two operands of different type, then it may be required to convert one type to another in order to perform the operation.
- For example, the expression “a + b,” if a is of integer and b is real, then to perform the addition a may be converted to real.
- The type checker can be designed to do this conversion. The conversion done automatically by the compiler is implicit conversion and this process is known as coercion.
- If the compiler insists the programmer to specify this conversion, then it is said to be explicit. Explicit conversions are also called casts.
- For instance, all conversions in Ada are said to be explicit.

Type Conversion

- The semantic rules for type conversion are listed below.

	E → num	{E.type := int}
	E → num.num	{E.type := real}
	E → id	{E.type := lookup(id.entry)}

	E → E1 op E2	{E.type := if E1.type = int and E2.type = int
		then int
		else if E1.type = int and E2.type = real
		then real
		else if E1.type = real and E2.type = int
		then real
		else if E1.type = real and E2.type = real
		then real
		else type_error}

Table 3.4:Type Conversion

Overloading Of Functions And Operators

- An operator is overloaded if the same operator performs different operations.
- For example, in arithmetic expression a + b, the addition operator "+" is overloaded because it performs different operations, when a and b are of different types like integer, real, complex, and so on.
- Another example of operator overloading is overloaded parenthesis in ada, that i, the expression A(i) has different meanings. It can be the ith element of an array, or a call to function A with argument i, and so on.
- Operator overloading is resolved when the unique definition for an overloaded operator is determined.
- The process of resolving overloading is called operator identification because it specifies what operation an operator performs.
- The overloading of arithmetic operators can be easily resolved by processing only the operands of an operator.

Overloading Of Functions And Operators

- Like operator overloading, the function can also be overloaded.
- In function overloading, the functions have the same name but different numbers and arguments of different types.
- In Ada, the operator "*" has the standard meaning that it takes a pair of integers and returns an integer.
- The function of "*" can be overloaded by adding the following declarations:
  - ✓ Function "*"(a,b: integer) return integer.
  - ✓ Function "*"(a,b: complex) return integer.
  - ✓ Function "*"(a,b: complex) return complex.
- By addition of the above declarations, now the operator "*" can take the following possible types:
  - ✓ It takes a pair of integers and returns an integer
  - ✓ It takes a pair of integers and returns a complex number
  - ✓ It takes a pair of complex numbers and returns a complex number

Overloading Of Functions And Operators

- Function overloading can be resolved by the type checker based on the number and types of arguments.
- The type checking rule for function by assuming that each expression has a unique type is given as

E → E1(E2)
{
E.type : = t
E2.type : = t → u then
E.type : = u
else E.type : = type_error
}

	$E' \rightarrow E$	{E′.type := E.type}
	$E \rightarrow id$	{E.type := lookup(id.entry)}
	$E \rightarrow E_1(E_2)$	{E.type := { u | there exists an s in E_2.type Such that $s \rightarrow u$ is in E_1.type }

Table 3.5: Overloading Of Functions and Operators

Polymorphic Functions

- A piece of code is said to be polymorphic if the statements in the body can be executed with different types.
- A function that takes the arguments of different types and executes the same code is a polymorphic function.
- The type checker designed for a language like Ada that supports polymorphic functions, the type expressions are extended to include the expressions that vary with type variables.
- The same operation performed on different types is called overloading and are often found in object-oriented programming.
- For example, let us consider the function that takes two arguments and returns the result.

int add(int, int)
int add(real, real)
real add(real, real)

The type expression for the function add is given as

$$\text{int} \times \text{int} \rightarrow \text{int}$$
$$\text{real} \times \text{real} \rightarrow \text{int}$$
$$\text{real} \times \text{real} \rightarrow \text{real}$$

Chomsky hierarchy of languages and recognizers.

- Chomsky Hierarchy represents the class of languages that are accepted by the different machine. The category of language in Chomsky's Hierarchy is as given below:
  - ✓ Type 0 known as Unrestricted Grammar.
  - ✓ Type 1 known as Context Sensitive Grammar.
  - ✓ Type 2 known as Context Free Grammar.
  - ✓ Type 3 Regular Grammar.

Therefore every language of type 3 is also of type 2, 1 and 0. Similarly, every language of type 2 is also of type 1 and type 0, etc.

Recursively enumerable grammars –recognizable by a Turing machine

Context-sensitive grammars –recognizable by the linear bounded automaton

Context-free grammars - recognizable by the pushdown automaton

Regular grammars –recognizable by the finite state automaton

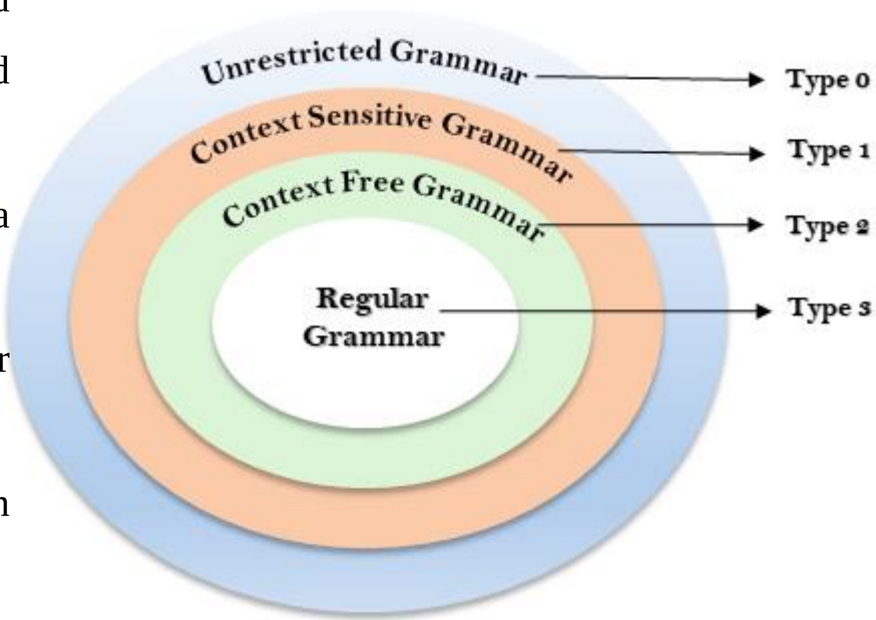


Fig: Chomsky Hierarchy

Chomsky hierarchy of languages and recognizers.

Type 0 –Recursively enumerable grammar (unrestricted grammars)

- Type-0 grammars (unrestricted grammars) include all formal grammars. They generate exactly all languages that can be recognized by a Turing machine.
- These languages are also known as the recursively enumerable languages. Note that this is different from the recursive languages which can be decided by an always-halting Turing machine.
- Class 0 grammars are too general to describe the syntax of programming languages and natural languages.

Ex:

bAa → aa

S → s

Chomsky hierarchy of languages and recognizers.

Type 1 –Context-sensitive grammars

- Type-1 grammars generate the context-sensitive languages.
- These grammars have rules of the form $\alpha A \beta \rightarrow \alpha \gamma \beta$ with A a nonterminal and α,β and γ strings of terminals and nonterminals.
- The strings α and β may be empty, but γ must be nonempty.
- The languages described by these grammars are exactly all languages that can be recognized by a linear bounded automaton.

Example:

$AB \rightarrow CDB$
$AB \rightarrow CdEB$
$ABcd \rightarrow abCDBcd$
$B \rightarrow b$

Chomsky hierarchy of languages and recognizers.

Type- 2 – Context-free grammars

- Type-2 grammars generate the context-free languages.
- These are defined by rules of the form $A \longrightarrow \gamma$ with A a nonterminal and γ a string of terminals and nonterminals.
- These languages are exactly all languages that can be recognized by a non-deterministic pushdown automaton.
- Context-free languages are the theoretical basis for the syntax of most programming languages.

Example:

$A \longrightarrow aBc$

Chomsky hierarchy of languages and recognizers.

Type 3 –Regular grammars

- Type-3 grammars generate the regular languages.
- Such a grammar restricts its rules to a single nonterminal on the left-hand side and a right-hand side consisting of a single terminal, possibly followed (or preceded, but not both in the same grammar) by a single nonterminal.
- The rule S → ε is also allowed here if S does not appear on the right side of any rule.
- These languages are exactly all languages that can be decided by a finite state automaton.
- Additionally, this family of formal languages can be obtained by regular expressions.
- Regular languages are commonly used to define search patterns and the lexical structure of programming languages.

Example:

A → ε   A → a   A → abc   A → B   A → abcB