Unit 4
Semantic Analysis

……………………………………………………………………………………………………….

Topics

Semantic Analysis: Static & Dynamic Checks, Typical Semantic errors, Scoping, Type Checking; Syntax directed definitions (SDD) & Translation (SDT), Attribute Types: Synthesized & Inherited, Annotated Parse Tree, S-attributed and L-attributed grammar, Applications of syntax directed translation, Type Systems, Type Checking and Conversion

………………………………………………………………………………………………………..

Semantic Analysis

Semantics of a language provide meaning to its constructs, like tokens and syntax structure. Semantics help interpret symbols, their types, and their relations with each other. Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not.

CFG + semantic rules = Syntax Directed Definitions

We have mentioned some of the semantics errors that the semantic analyzer is expected to recognize:

- Type mismatch
- Undeclared variable
- Reserved identifier misuse.
- Multiple declaration of variable in a scope.
- Accessing an out of scope variable.
- Actual and formal parameter mismatch.

Examples of Semantic Errors

Example 1: Use of a non-initialized variable

```
int i;
i++; // the variable i is not initialized
```

Example 2: Type incompatibility

```
int a = "hello"; // the types String and int are not compatible
```

Example 3: Errors in expressions

```
char s = 'A';
int a = 15 - s; // the - operator does not support arguments of type char
```

Example 4: Array index out of range (dynamic semantic error)

```
int a[10];
a[10] = 50; // 10 is not a legal index for an array of 10 elements
```

Example 5: Unknown references

```
int a;
```

```
printf("%d", a);
```

Type Checking

Compiler must check that the source program follows both the syntactic and semantic conventions of the source language. Type checking is the process of checking the data type of different variables. The design of a type checker for a language is based on information about the syntactic construct in the language, the notation of type, and the rules for assigning types to the language constructs.

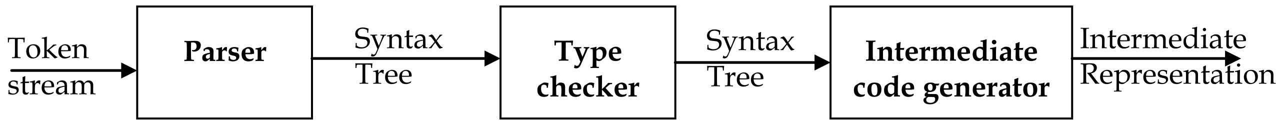


Figure: Position of Type Checker

Type Systems

The design of a type checker for a language is based on information about the syntactic constructs in the language, the notion of types, and the rules for assigning types to language constructs. **For example:** "if both operands of the arithmetic operators of +, - and * are of type integer, then the result is of type integer".

The collection of different data types and their associated rules to assign types to programming language constructs is known as type systems. It is an informal type system rules, for example "if both operands of addition are of type integer, then the result is of type integer". A type checker implements type system.

Type expressions

The type of a language construct will be denoted by a "type expression." A type expression is either a basic type or is formed by applying an operator called a type constructor to other type expressions. The sets of basic types and constructors depend on the language to be checked. The following are the definitions of type expressions:

1. Basic types such as boolean, char, integer, real are type expressions.
   A special basic type, type_error, will signal an error during type checking;
   void denoting "the absence of a value" allows statements to be checked.
2. Since type expressions may be named, a type name is a type expression.
3. A type constructor applied to type expressions is a type expression.

Constructors include:

- **Arrays:** If T is a type expression then array(I, T) is a type expression denoting the type of an array with elements of type T and index set I. Example: array(0..99, int)
- **Products:** If T1 and T2 are type expressions, then their Cartesian product T1 X T2 is a type expression. Example: int x int

- **Records:** The difference between a record and a product is that the names. The record type constructor will be applied to a tuple formed from field names and field types.
- **Pointers:** If T is a type expression, then **pointer(T)** is a type expression denoting the type "pointer to an object of type T". For example, var p: ↑ row declares variable p to have type pointer (row).
- **Functions:** A function in programming languages maps a domain type D to a range type R. The type of such function is denoted by the type expression D → R. Example: int → int represents the type of a function which takes an int value as parameter, and its return type is also int.

Example: Type Checking of Expressions

E → id	{E.type = lookup(id.entry)}
E → charliteral	{E.type = char}
E → intliteral	{E.type = int}
E → E1 + E2	{E.type = (E1.type == E2.type)? E1.type: type_error}
E → E1 [E2]	{E.type = (E2.type == int and E1.type == array(I, t))? t: type_error}
E → E1 ↑	{E.type = (E1.type == pointer(t)) ? t : type_error}
S → id = E	{S.type = (id.type == E.type)? E.type: type_error}

Note: the type of id is determined by: id.type = lookup(id.entry)

S → if E then S1	{S.type = (E.type == boolean)? S1.type: type_error}
S → if **E then S1 else** S_2	**{S.type = (E.type == boolean)? S1.type: S2.type}**
S → while E do S1	{S.type = (E.type == boolean)? S1.type: type_error}
S → S1; S2	{S.type = (S1.type == void and S2.type == void)? void: type_error}

Static versus Dynamic type Checking

Static checking

The type checking at the compilation time is known as static checking. Typically syntactical errors and misplacement of data type take place at this stage. A language is statically-typed if the type of a variable is known at compile time instead of at runtime. Common examples of statically-typed languages include Ada, C, C++, C#, JADE, Java, FORTRAN, Haskell, ML, Pascal, and Scala.

The big benefit of static type checking is that it allows many type errors to be caught early in the development cycle. Static typing usually results in compiled code that executes more quickly because when the compiler knows the exact data types that are in use, it can produce optimized machine code (i.e. faster and/or using less memory). Static type checkers evaluate only the type information that can be determined at compile time, but are able to verify that the checked conditions hold for all possible executions of the program, which eliminates the need to repeat type checks every time the program is executed.
Typical examples of static checking are:

- Type checks
- Flow-of-control checks
- Uniqueness checks
- Name-related checks

Static type checking is achieved typically through a type system. The main purpose of a type system is to reduce possibilities for bugs in programs by defining interfaces between different parts of a program, and then checking that the parts have been connected in a consistent way.

Dynamic type checking

The type checking at the run time is known as static checking. Compiler generates verification code to enforce programming language's dynamic semantics. A programming language is strongly-typed, if every program its compiler accepts will execute without type errors.
In practice, some of types checking operations are done at run-time (so, most of the programming languages are not strongly-typed).
Example: Some errors can only be detected at run-time; e.g., array out-of bounds as below,

```
int a[10];
for (int i=0; i<20; ++i)
        a[i] = i;
```

Dynamic type checking is the process of verifying the type safety of a program at runtime. Dynamically typed languages include Groovy, JavaScript, LISP, Objective-C, PHP, Prolog, Python, Ruby, Smalltalk etc.

Type Conversion and Coercion

Type conversion

The process of converting data from one type to another type is known as type conversion. In typing casting, a data type is converted into another data type by the programmer using the casting operator during the program design. In typing casting, the destination data type may be smaller than the source data type when converting the data type to another data type, so it is also called narrowing conversion. Often if different parts of an expression are of different types then type conversion is required.

Syntax/Declaration

```
destination_datatype = (target_datatype) variable;
(): is a casting operator
```

For example, in the expression: z = x + y what is the type of z if x is integer and y is real? Compiler have to convert one of them to ensure that both operand of same type!

Example	Result

int x;
x = (int) 10.50;
	x = 10

int code = (int) 'A';	code = 65
char Letter = (char) 66;	Letter = B

int a=20, b=8;
float Avg;
Avg = a/b;
	Avg = 2

int a=20, b=8;
float Avg;
Avg = (float) a/(float) b;
	Avg = 2.50

Coercion

The process of converting one type to another by compiler itself is known as coercion. Type conversion which happens implicitly is called coercion. In type conversion, the destination data type cannot be smaller than the source data type, that's why it is also called widening conversion. One more important thing is that it can only be applied to compatible data types.

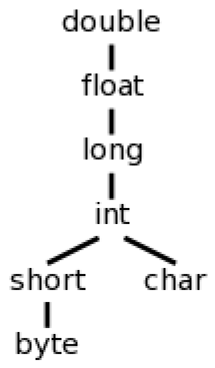
Example 1: In this example x is converted to their equivalent floating value

```
int x = 30;
float y;
y = x;              //y =30.000000
```

Example 2: Here the value of count is automatically converted to float before executing the divide operation.

```
float average, sum;
int count;
average = sum / count;
```

Mathematically the hierarchy on the right is a partially order set in which each pair of elements has a least upper bound. For many binary operators (all the arithmetic ones we are considering, but not exponentiation) the two operands are converted to the LUB. So adding a short to a char, requires both to be converted to an int. adding a byte to a float, requires the byte to be converted to a float (the float remains a float and is not converted).



Difference between Type casting and Type conversion (Coercion)

S.no	Type casting	Type conversion (Coercion)
1.	In type casting, a data type is converted into another data type by a programmer using casting operator.	Whereas in type conversion, a data type is converted into another data type by a compiler.
2.	Type casting can be applied to compatible data types as well as incompatible data types.	Whereas type conversion can only be applied to compatible data types.
3.	In type casting, casting operator is needed in order to cast the data type to another data type.	Whereas in type conversion, there is no need for a casting operator.
4.	In typing casting, the destination data type may be smaller than the source data type, when converting the data type to another data type.	Whereas in type conversion, the destination data type can't be smaller than source data type.
5.	Type casting takes place during the program design by programmer.	Whereas type conversion is done at the compile time.
6.	Type casting is also called narrowing conversion because in this, the destination data type may be smaller than the source data type.	Whereas type conversion is also called widening conversion because in this, the destination data type cannot be smaller than the source data type.
7.	Type casting is often used in coding and competitive programming works.	Whereas type conversion is less used in coding and competitive programming as it might cause incorrect answer.
8.	Type casting is more efficient and reliable.	Whereas type conversion is less efficient and less reliable.

Syntax-Directed Translation

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 we can augment our 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. There are two ways to represent the semantic rules associated with grammar symbols.

- Syntax-Directed Definitions (SDD)
- Syntax-Directed Translation Schemes (SDTS)

Syntax-Directed Definitions

A syntax-directed definition (SDD) is a context-free grammar together with attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions. Syntax-Directed Definitions are high level specifications for translations. They hide many implementation details and free the user from having to explicitly specify the order in which translation takes place. A syntax-directed definition is a generalization of a context-free grammar in which each grammar symbol is associated with a set of attributes. This set of attributes for a grammar symbol is partitioned into two subsets synthesized and inherited attributes of that grammar.

In brief, A syntax-directed definition is a grammar together with semantic rules associated with the productions. These rules are used to compute attribute values.

Example: The syntax directed definition for a simple desk calculator

Production	Semantic Rules
L → E return	print(E.val)
E → E1 + T	E.val = E1.val + T.val
E → T	E.val = T.val
T → T1 * F	T.val = T1.val * F.val
T → F	T.val = F.val
F → (E)	F.val = E.val
F → digit	F.val = digit.lexval

An attribute has a name and an associated value: a string, a number, a type, a memory location, an assigned register, strings. The strings may even be long sequences of code, say code in the intermediate language used by a compiler. If X is a symbol and 'a' is one of its attributes, then we write X.a to denote the value of 'a' at a particular parse-tree node labeled X. If we implement the nodes of the parse tree by records or objects, then the attributes of X can be implemented by data fields in the records that represent the nodes for X. The attributes are evaluated by the semantic rules attached to the productions.

Syntax-Directed Translation Schemes (SDT)

SDT embeds program fragments called semantic actions within production bodies. The position of semantic action in a production body determines the order in which the action is executed. It is used to evaluate the order of semantic rules.

Syntax,

$A \rightarrow$ {…..} X {……} Y {…….}

Where, within the curly brackets we define semantic actions.

Example: In the rule $E \rightarrow E_1 + T$ {print '+'}, the action is positioned after the body of the production.

SDTs are more efficient than SDDs as they indicate the order of evaluation of semantic actions associated with a production rule.

Example 1: A simple translation scheme that converts infix expressions to the corresponding postfix expressions.

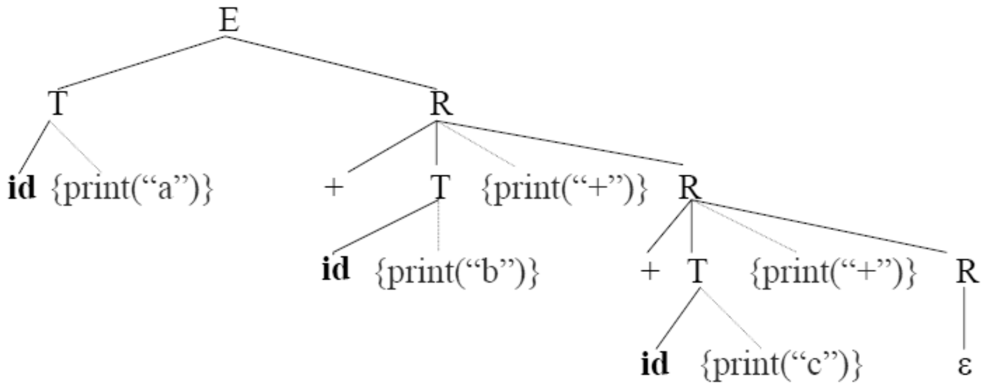
$E \rightarrow T\ R$

$R \rightarrow + T$ {print("+")} R_1

$R \rightarrow \varepsilon$

$T \rightarrow$ id {print(id.name)}

Infix expression (a + b + c) → postfix expression (a b + c +)



Attribute Types: Synthesized & Inherited

Terminals can have synthesized attributes, which are given to it by the lexer (not the parser). There are no rules in an SDD giving values to attributes for terminals. Terminals do not have inherited attributes. A non-terminal A can have both inherited and synthesized attributes. The difference is how they are computed by rules associated with a production at a node N of the parse tree.

Synthesized attribute (↑)

A synthesized attribute for a non-terminal A at a parse-tree node N is defined by a semantic rule associated with the production at N. Note that the production must have A as its head. A synthesized attribute at node N is defined only in terms of attribute values at the children of N and at N itself.

If the value of the attribute only depends upon its children then it is **synthesized attribute.** Simply the attributes of a node that are derived from its children nodes are called synthesized attributes. Terminals do not have inherited attributes. A non-terminal 'A' can have both inherited and synthesized attributes. The difference is how they are computed by rules associated with a production at a node N of the parse tree. To illustrate, assume the following production:

$S \rightarrow ABC$

If S is taking values from its child nodes (A, B, C), then it is said to be a synthesized attribute, as the values of ABC are synthesized to S.

Example for Synthesized Attributes

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

Production	Semantic Rules
$L \rightarrow E$ return	print(E.val)
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow$ digit	F.val = digit.lexval

Inherited attribute (→, ↓)

An inherited attribute for a non-terminal B at a parse-tree node N is defined by a semantic rule associated with the production at the parent of N. Note that the production must have B as a symbol in its body. An inherited attribute at node N is defined only in terms of attribute values at N's parent, N itself, and N's siblings. An inherited attribute at node N cannot be defined in

terms of attribute values at the children of node N. However, a synthesized attribute at node N can be defined in terms of inherited attribute values at node N itself.
Simply, a node in which attributes are derived from the parent or siblings of the node is called inherited attribute of that node. If the value of the attribute depends upon its parent or siblings then it is inherited attribute. As in the following production,

$S \rightarrow ABC$

'A' can get values from S, B and C. B can take values from S, A, and C. Likewise, C can take values from S, A, and B.

Example for Inherited Attributes

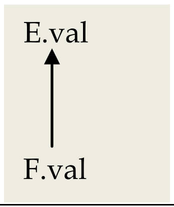
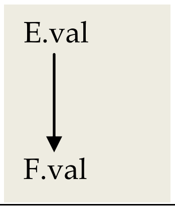
Let us consider the syntax directed definition with both inherited and synthesized attributes for the grammar for "type declarations":

Production	Semantic Rules
$D \rightarrow TL$	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 $D \rightarrow T\ L$ is associated with the semantic rule L.in = T.type which set the inherited attribute L.in.

Comparison between Synthesized and inherited attributes

S.no	Synthesized attributes	Inherited attributes
1.	An attribute is said to be Synthesized attribute if its parse tree node value is determined by the attribute value at child nodes.	An attribute is said to be Inherited attribute if its parse tree node value is determined by the attribute value at parent and/or siblings node.
2.	The production must have non-terminal as its head.	The production must have non-terminal as a symbol in its body.
3.	A synthesized attribute at node n is defined only in terms of attribute values at the children of n itself.	An Inherited attribute at node n is defined only in terms of attribute values of n's parent, n itself, and n's siblings.
4.	It can be evaluated during a single bottom-up traversal of parse tree.	It can be evaluated during a single top-down and sideways traversal of parse tree.
5.	Synthesized attributes can be contained by both the terminals and non-terminals.	Inherited attributes can't be contained by both; It is only contained by non-terminals.
6.	Synthesized attribute is used by both S-	Inherited attribute is used by only L-

	attributed SDT and L-attributed STD.	attributed SDT.
7.	E → F E.val = F.val 	E → F E.val = F.val 

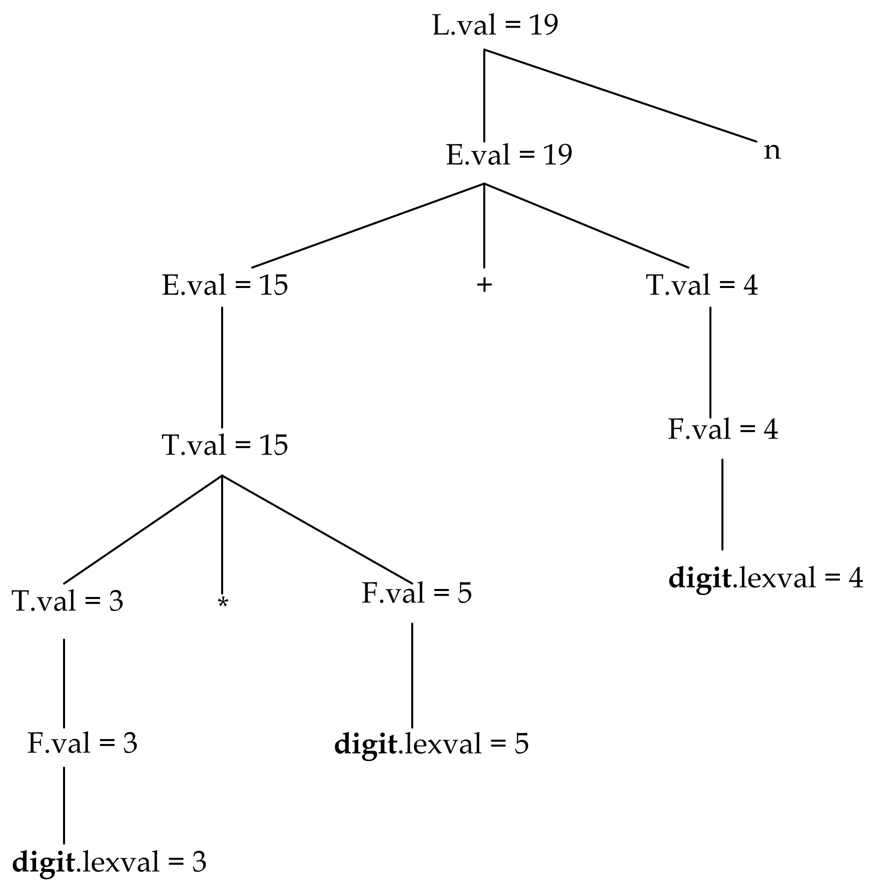
Annotated Parse Tree

A parse tree constructing for a given input string in which each node showing the values of attributes is called an annotated parse tree. It is a parse tree showing the values of the attributes at each node. The process of computing the attribute values at the nodes is called annotating or decorating the parse tree.

Example 1: Let's take a grammar,

L → E **n**
E → E_1 + T
E → T
T → T_1 * F
T → F
F → (E)
F → **digit**

Now the annotated parse tree for the input string 3*5+4 is,



For computation of attributes we start from leftmost bottom node. The rule F → digit is used to reduce digit to F and the value of digit is obtained from lexical analyzer which becomes value of F i.e. from semantic action F.val = digit.lexval. Hence, F.val = 3 and since T is parent node of F so, we get T.val = 3 from semantic action T.val = F.val. Then, for T → T_1 * F production, the corresponding semantic action is T.val = T_1.val * F.val. Hence, T.val = 3 * 5 = 15
Similarly, combination of E_1.val + T.val becomes E.val i.e. E.val = E_1.val + T.val = 19. Then, the production L → E is applied to reduce E.val = 19 and semantic action associated with it prints the result E.val. Hence, the output will be 19.

Example 2: For the SDD (Syntax-Directed Definitions) of following grammar, give annotated parse trees for the following expressions:

a. (3+4) ∗ (5+6) n
b. 1∗2∗3 ∗ (4+5) n
c. (9+8 ∗ (7+6)+5) ∗ 4 n

Solution: Parse tree and Annotated parse tree for: (3+4) ∗ (5+6) n

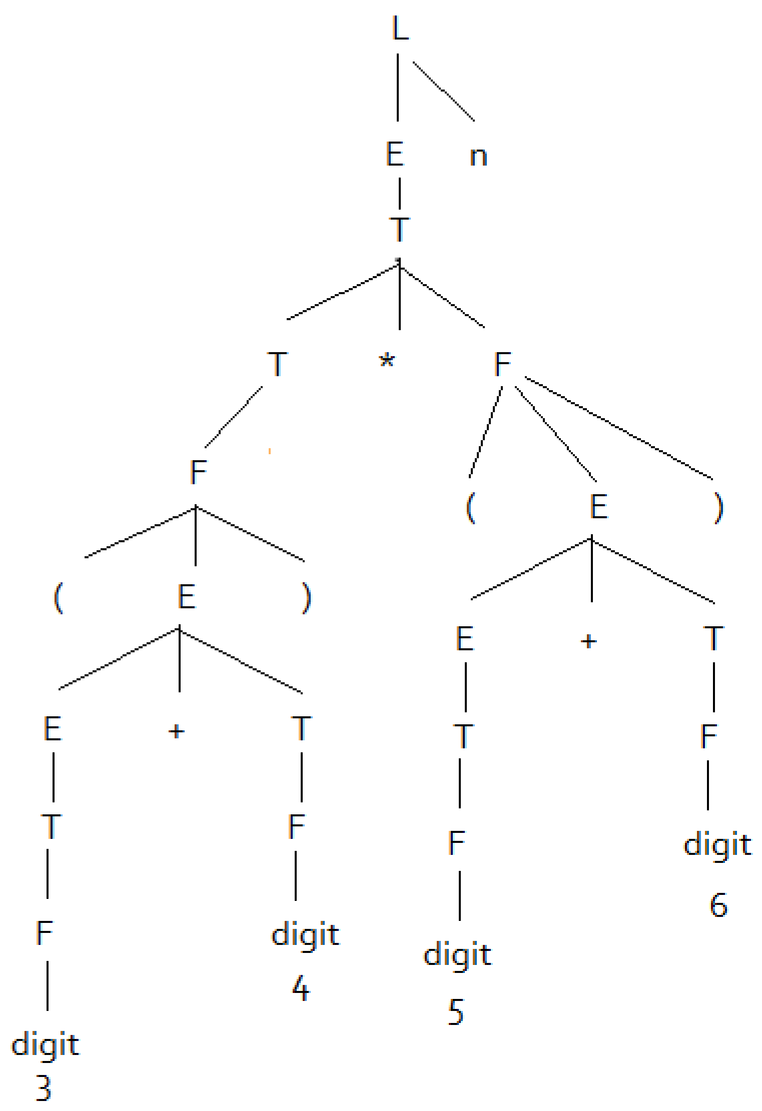


Figure: Parse tree for input string (3+4) $*$ (5+6) n

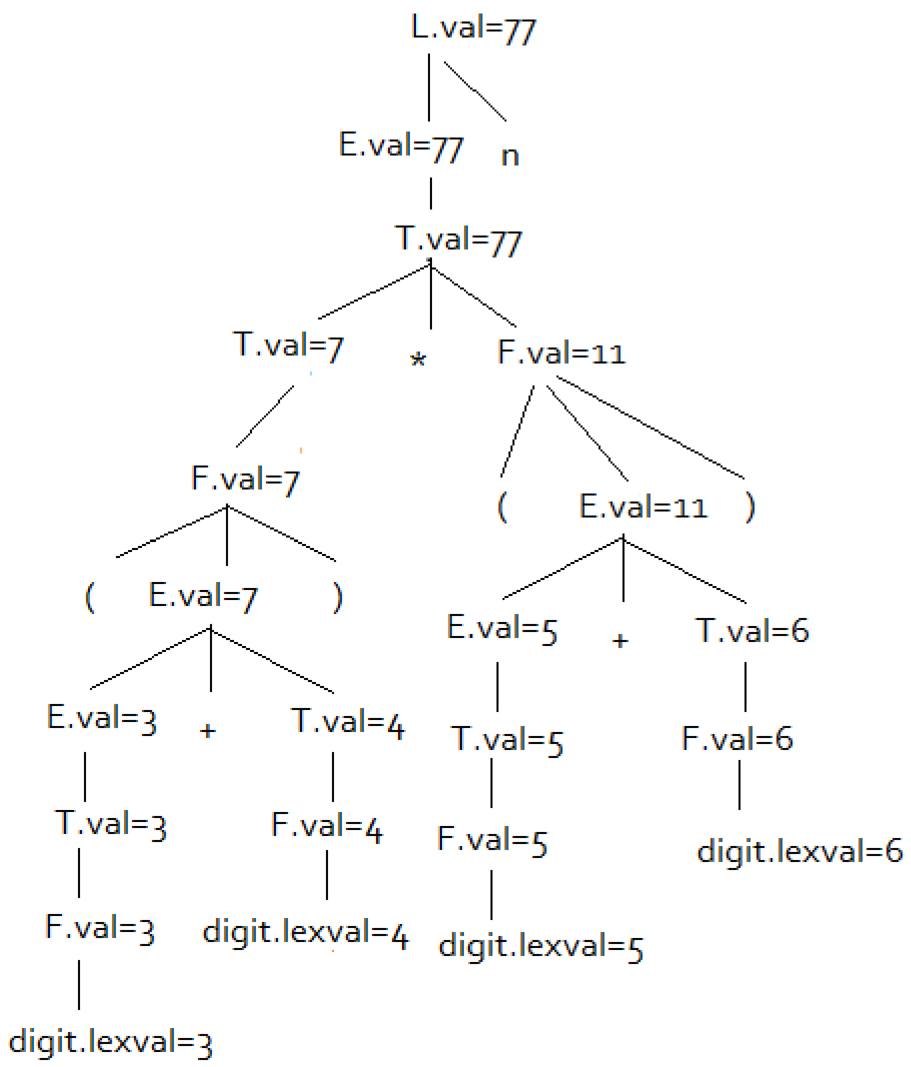


Figure: Annotated Parse tree for input string (3+4) ∗ (5+6) n

Solution: Annotated parse tree for: 1∗2∗3 ∗ (4+5) n

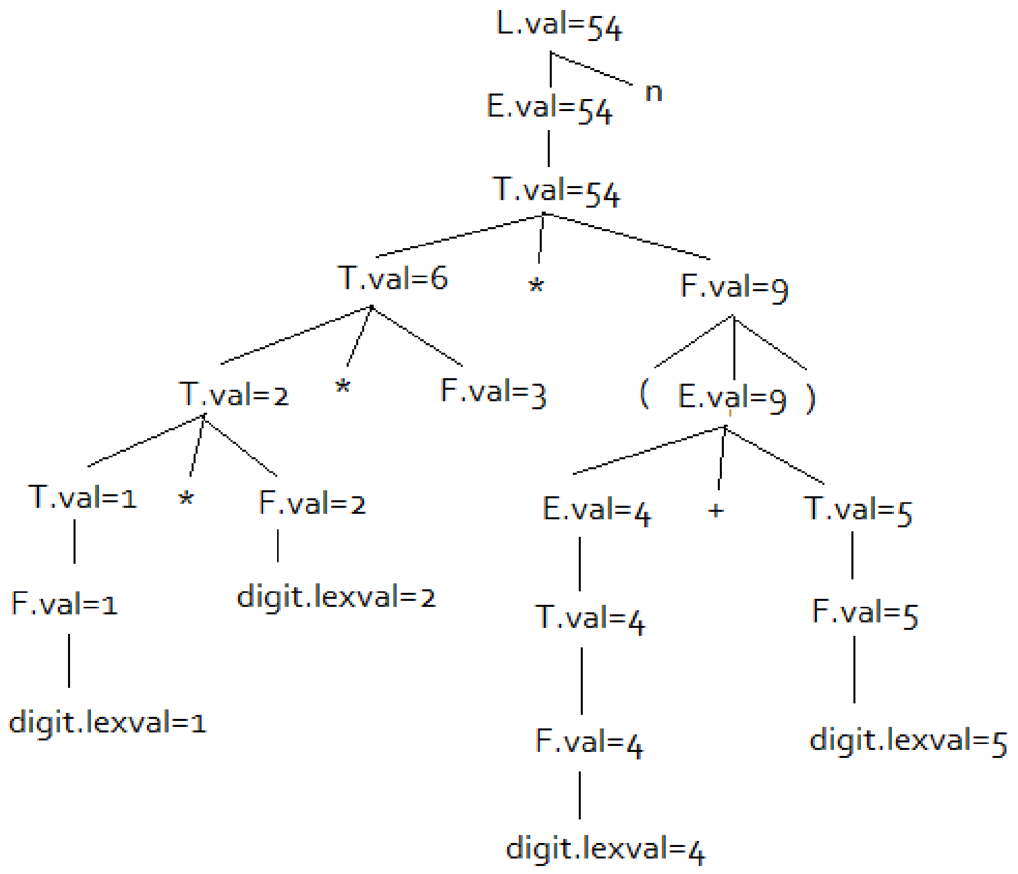


Figure: Annotated Parse tree for input string 1∗2∗3 ∗ (4+5) n

Solution: Annotated parse tree for: (9+8 ∗ (7+6)+5) ∗ 4 n

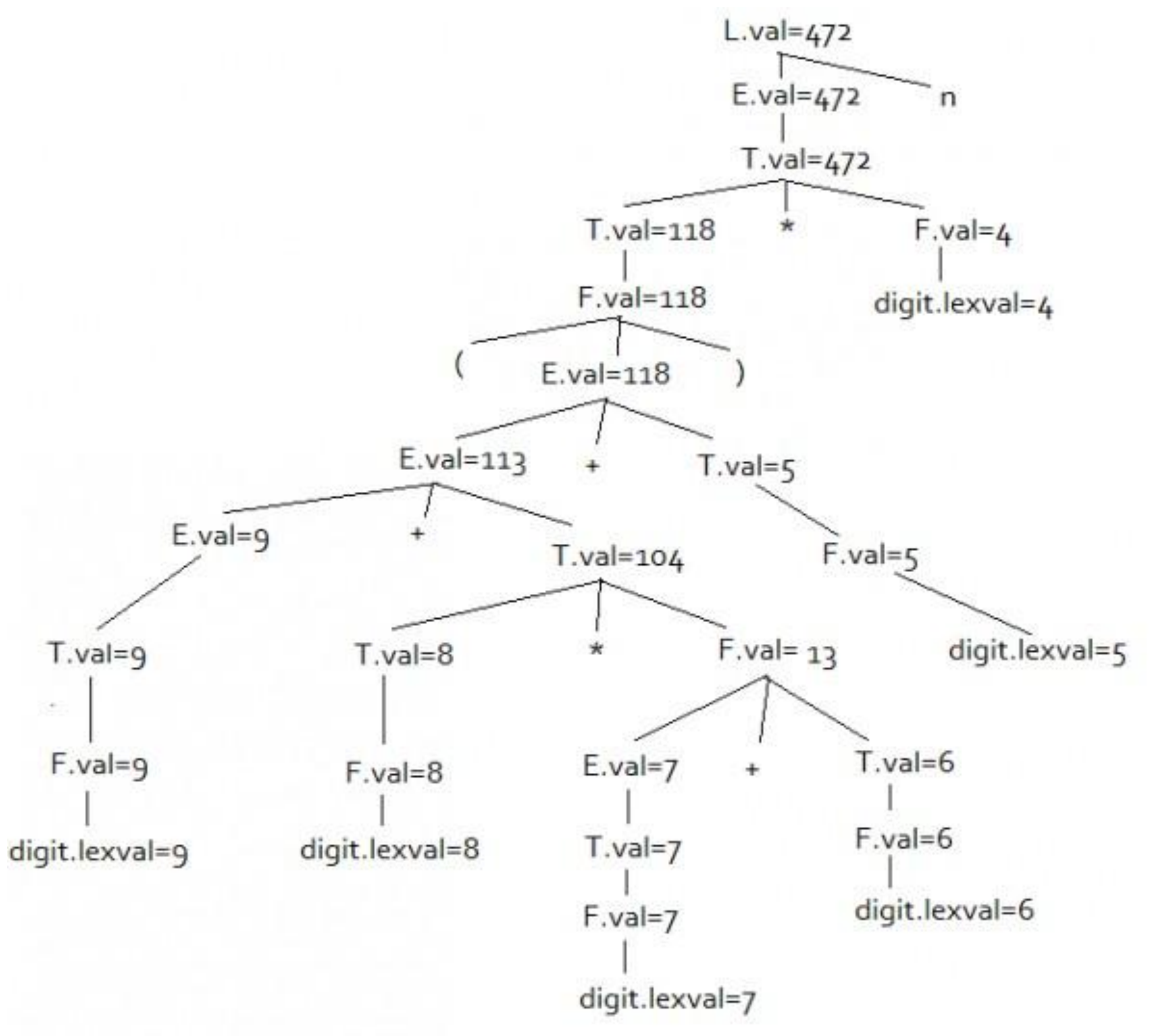


Figure: Annotated Parse tree for input string (9+8 ∗ (7+6)+5) ∗ 4 n

Dependency Graph

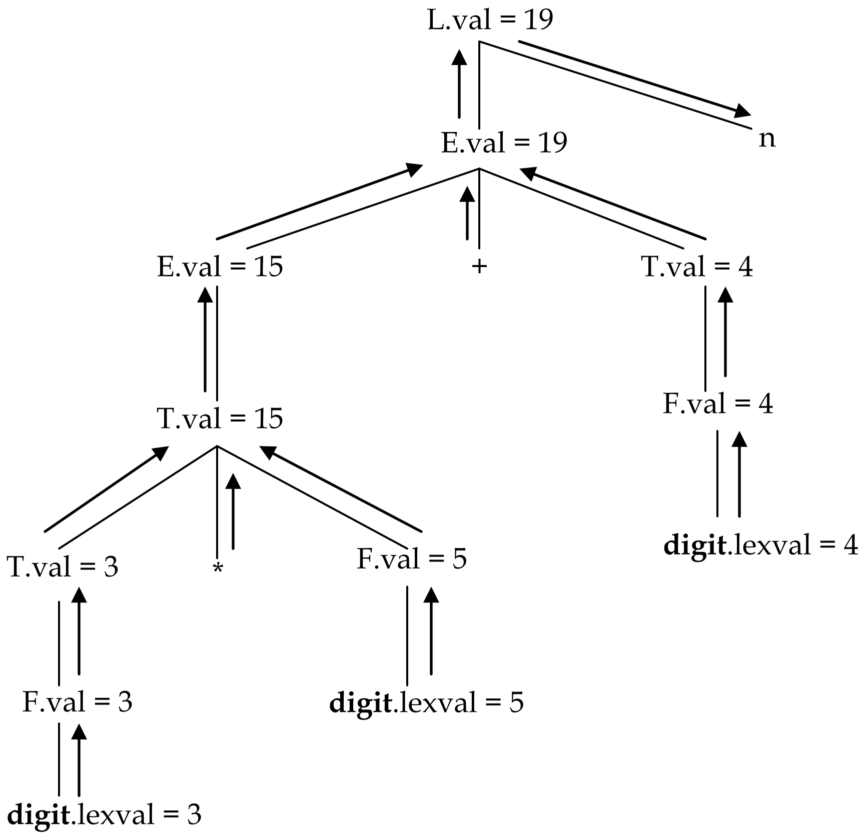
If interdependencies among the inherited and synthesized attributes in an annotated parse tree are specified by arrows then such a tree is called dependency graph. In order to correctly evaluate attributes of syntax tree nodes, a dependency graph is useful. A dependency graph is a directed graph that contains attributes as nodes and dependencies across attributes as edges.

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.

Example 1: Let's take a grammar,

L → E **print**
E → E_1 + T |T
T → T_1 * F |F
F → (E)| **digit**

Now the dependency graph for the input string 3*5+4 is,



Example 2: Draw dependency graph for following input string by using given grammar,

D → TL

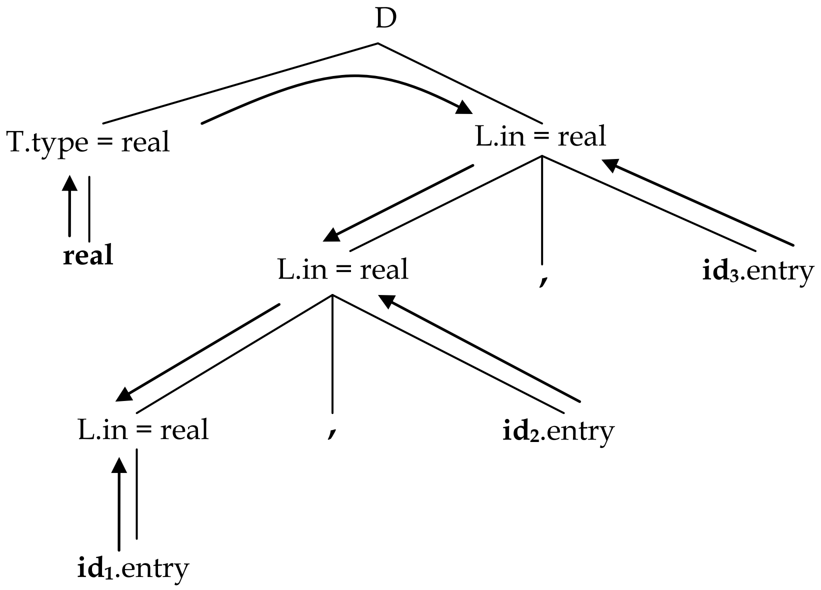
T → **int**

T → **real**

L → L_1 **id**

L → **id**

Input string: real id_1, id_2, id_3



Practice Example 1: For the following grammar

E → E + E | E * E | (E) | id

Annotate the grammar with syntax directed definitions using synthesized attributes. Remove left recursion from the grammar and rewrite the attributes correspondingly. [TU]

Solution: First part

Production	**Semantic Rule**
E → E + E	{E.val = E1.val + E2.val}
E → E * E	{E.val = E1.val * E2.val}
E → (E)	{E.val = E1.val}
E → id	{E.val=id.lexval}

A→Aα1 | Aα2 | Aα3… | Aα$_n$ |β1| β2|…| βn
A→ β1A′| β2A′|………| βnA′
A′→ α1A′ | α2A′ | α3A′|…………| α$_n$A′|ε

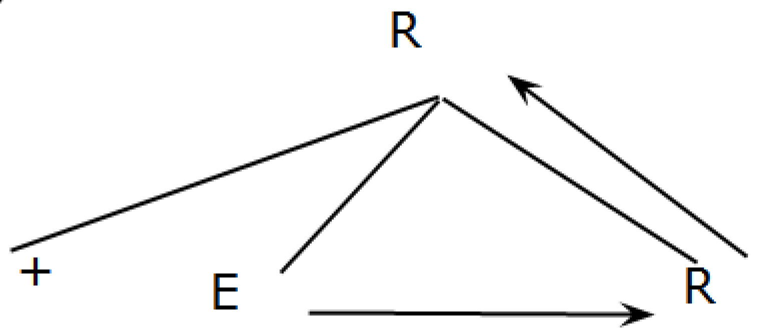
Second part

E → E + E |E * E|(E)| id

Removing left recursion as,

E → (E1) R | id R
R → +ER1 | *ER1 | ε

Now add the attributes within this non-left recursive grammar as,



E → (E1) {R.in=E$_1$.val} R {E.val=R.syn}
E → id {R.in=id.lexval} R$_1$ {E.val=R.syn }
R → +E {R$_1$.in=E.val+R.in} R$_1$ {R.syn=R$_1$.syn}
R → *E {R$_1$.in=E.val*R.in } R$_1$ { R.syn=R$_1$.syn }
R → ε {R.syn=R.in}

Practice Example 2: For the following grammar

E → E + E | E * E | (E) | id

At first remove the left recursion then construct an annotated parse tree for the expression 2*(3+5) using the modified grammar. Trace the path in which semantic attributes are evaluated. [TU]

Solution: At first add attributes to given grammar as,

E → E + E {E.val = E$_1$.val + E$_2$.val}
E → E * E {E.val = E$_1$.val * E$_2$.val}

E → (E) {E.val = E_1.val}

E → id {E.val=id.lexval}

Removing left recursion as,

E → (E) R | id R

R → +ER_1 | *ER_1 | ε

Now add the attributes within this non-left recursive grammar as,

E → (E) {R.in=E_1.val} R {E.val=R.syn}

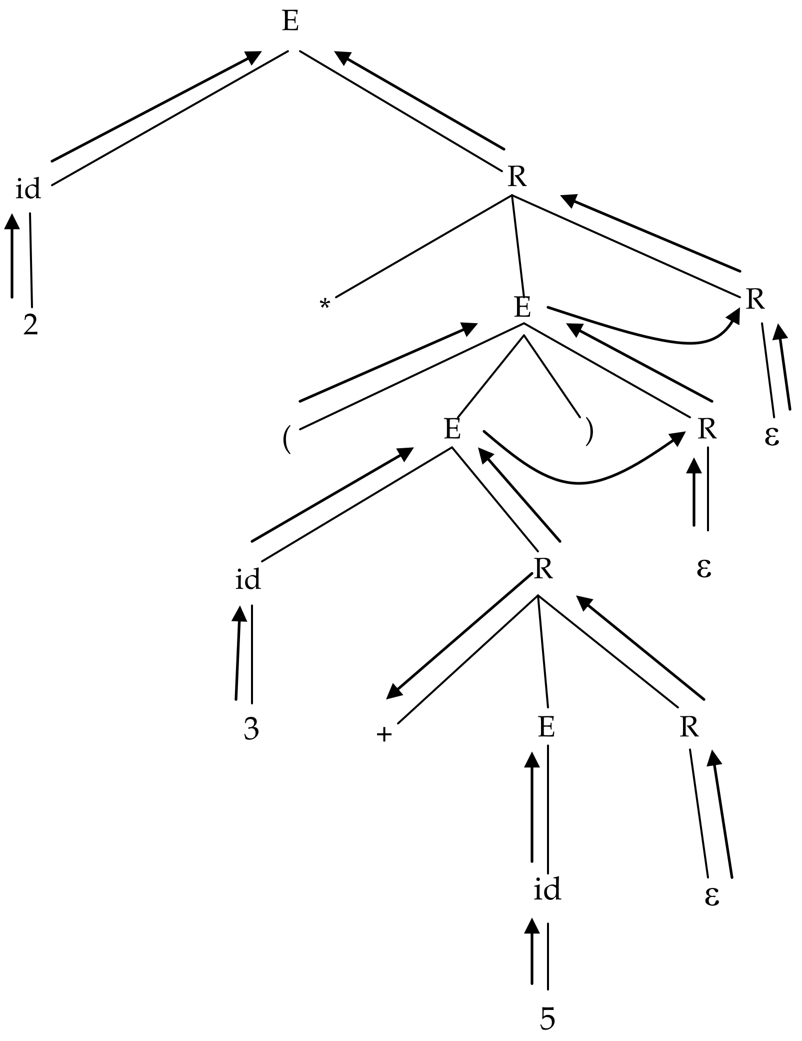
E → id {R.in=id.lexval} R_1 {E.val=R.syn}

R → +E {R1.in=E.val+R.in} R_1 {R.syn=R_1.syn}

R → *E {R_1.in=E.val*R.in} R_1 {R.syn=R_1.syn}

R → ε {R.syn=R.in}

Second part: An annotated parse tree for 2*(3+5) is,



S-Attributed Definitions

If an SDT uses only synthesized attributes, it is called as S-attributed SDT. Attribute values for the non-terminal at the head is computed from the attribute values of the symbols at the body of the production.

The attributes of an S-attributed SDD can be evaluated in bottom up order of nodes of the parse tree. i.e., by performing post order traversal of the parse tree and evaluating the attributes at a node when the traversal leaves that node for the last time.

Example for S-attributed definitions

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

Production	Semantic Rules
L → E return	print(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

As depicted above, attributes in S-attributed SDTs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.

L-Attributed Definitions

This form of SDT uses both synthesized and inherited attributes with restriction of not taking values from right siblings. In L-attributed SDTs, a non-terminal can get values from its parent, child, and sibling nodes. As in the following production

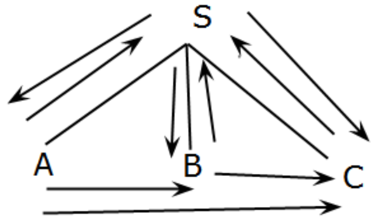
S → ABC

$$\begin{tikzcd}
 & S \arrow[dl, bend right] \arrow[d, bend right] \arrow[dr, bend left] & \\
A \arrow[ur] \arrow[r] \arrow[rr, bend right] & B \arrow[u] \arrow[r] & C \arrow[ul]
\end{tikzcd}$$

S can take values from A, B, and C (synthesized). ‘A’ can take values from S only. B can take values from S and A. C can get values from S, A, and B. No non-terminal can get values from the sibling to its right. Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner.

S-->ABC

S.val=A.val
S.val=B.val
S.val=C.val
A.val=S.val
B.val=S.val,A.val
C.val=S.val, A.val, B.val



The syntax directed definition in which the edges of dependency graph for the attributes in production body, can go from left to right and not from right to left is called L-attributed definitions. Attributes of L-attributed definitions may either be synthesized or inherited. If the attributes are inherited, it must be computed from:

- Inherited attribute associated with the production head.
- Either by inherited or synthesized attribute associated with the production located to the left of the attribute which is being computed.
- Either by inherited or synthesized attribute associated with the attribute under consideration in such a way that no cycles can be formed by it in dependency graph.

Example:

Production	**Semantic Rules**
T → FT'	T '.inh = F.val
T ' → *FT_1′	T'_1.inh = T'.inh * F.val

In production 1, the inherited attribute T' is computed from the value of F which is to its left. In production 2, the inherited attributed T_1' is computed from T'.inh associated with its head and the value of F which appears to its left in the production. i.e., for computing inherited attribute it must either use from the above or from the left information of SDD.

Exercise

Short questions

1. Define type checking. Differentiate between type checking and conversions.
2. Differentiate between static and dynamic type checking with example.
3. What is type system? Explain
4. What is parse tree? How it is differ from annotated parse tree?
5. Define syntax directed definition. How it is differ from syntax directed translation?
6. Define dependency graph with suitable example.
7. Mention applications of syntax directed translation.
8. What is type constructor? Explain with suitable example.
9. Define synthesized attribute. How it is differ from inherited attribute? Explain
10. Define S- attributed and L-attributed definitions.

Long questions

1. Define the syntax directed definitions with an example. How definitions are different from translation schemas?
2. Consider the following grammar and give the syntax directed definition to construct parse tree. For the input expression 5*3+4*9, construct an annotated parse tree along with dependency graph according to your syntax directed definition.

   $E \rightarrow TE'$

   $E' \rightarrow +TE'$

   $E' \rightarrow \varepsilon$

   $T \rightarrow FT'$

   $T' \rightarrow *FT'$

   $T' \rightarrow \varepsilon$

   $F \rightarrow \text{digit}$
3. What is type checking? Explain about static and dynamic type checking.
4. Define L-attributed definitions. For the following grammar

   $E \rightarrow E+E \mid E*E \mid (E) \mid id$

   At first remove the left recursion then construct an annotated parse tree for the expression 2*(3+5) using the modified grammar. Also trace the path in which semantic attributes are evaluated.
5. Define the syntax directed definitions with an example? How definitions are different from translation schemas?
6. Describe the inherited and synthesized attributes of grammar using an example.
7. Write the type expressions for the following types:
   a. An array of pointers to real's, where the array index range from 1 to 100
   b. Function whose domains are function from two characters and whose range is a pointer of integer

8. What do you mean by S-attributed definition and how they are evaluated? Explain with example.
9. Consider the following grammar for arithmetic expression using an operator 'op' to integer or real numbers.

   $E \rightarrow E_1$ op E_2 | num.num | num | id

   Give the syntax directed definitions to determine the type of expression as when two integers are used in expression, resulting type is integer otherwise real.
10. Construct a syntax directed translation scheme that translates arithmetic expressions from infix into postfix notation. Your solution should include the context free grammar; the semantic attributes for each the grammar symbols, and the semantic rules. Construct the annotated parse tree for the input 3 * 4 + 5 * 2.