[Syntax Directed Translation]

Compiler Design and Construction (CSc 352)

Compiled By

Bikash Balami

Central Department of Computer Science and Information Technology (CDCSIT)

Tribhuvan University, Kirtipur

Kathmandu, Nepal

Syntax – Directed Translation and Definitions

We associate information with a programming language construct by attaching attributes to the grammar symbols representing the grammar symbols representing the construct. Values for attributes are computed by "*semantic rules*" associated with the grammar productions. The way of associating attributes to each of the symbols in the grammar that provides us the order of translation (using dependency graph) allowing some level of implementation details is called syntax directed translation.

Evaluation of these semantic rules may:

- generate intermediate codes
- put information into the symbol table
- perform type checking
- issue error messages

An attribute may hold a string, a number, a memory location, a complex record. There are two notations for associating semantic rules with productions.

- **Syntax – Directed Definitions**
  - give high-level specifications for translations
  - hide many implementation details such as order of evaluation of semantic actions
  - associate a production rule with a set of semantic actions, and we do not say when they will be evaluated
- **Translation Schemes**
  - indicate the order of evaluation of semantic actions associated with a production rule
  - translation schemes give a little bit information about implementation details

Syntax – Directed Definitions

A syntax-directed definitions (or attribute grammar) bind a set of semantic rules to productions. They are the high level specification for the translation schemes i.e. they hide the implementation details and do not necessitate the consideration of translations order. It is the generalization of parse tree where each terminal and nonterminals has attributes associated to it whose values are determined by the semantic rules. In other words, 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 called *synthesized* and *inherited* attributes.
- Each production rule is associated with a set of semantic rules..

Semantic rules set up dependencies between attributes which can be represented by a dependency graph. This dependency graph determines the evaluation order of these semantic rules. If the value of the attribute only depends upon its children then it is **synthesized attribute**. The Syntax directed definition with only synthesized attribute is called S-attributed definition or S- attributed grammar. Where annotation is done using bottom up traversal. If the value of the attribute depends upon its parent or siblings then it is **inherited attribute**. Useful in the context like identifying the data type of the variables. A parse tree showing the values of attributes at each node is called an **annotated parse tree.** The process of computing the attributes values at the nodes is called **annotating (**or **decorating)** of the parse tree.

In a syntax-directed definition, each production $A \rightarrow \alpha$ is associated with a set of semantic rules of the form:

$b=f(c_1,c_2,\ldots,c_n)$ where f is a function and b can be one of the followings

- b is a synthesized attribute of A and $c_1,c_2,\ldots,c_n$ are attributes of the grammar symbols in the production ($A \rightarrow \alpha$). OR
- b is an inherited attribute one of the grammar symbols in α (on the right side of the production), and $c_1,c_2,\ldots,c_n$ are attributes of the grammar symbols in the production ($A \rightarrow \alpha$).

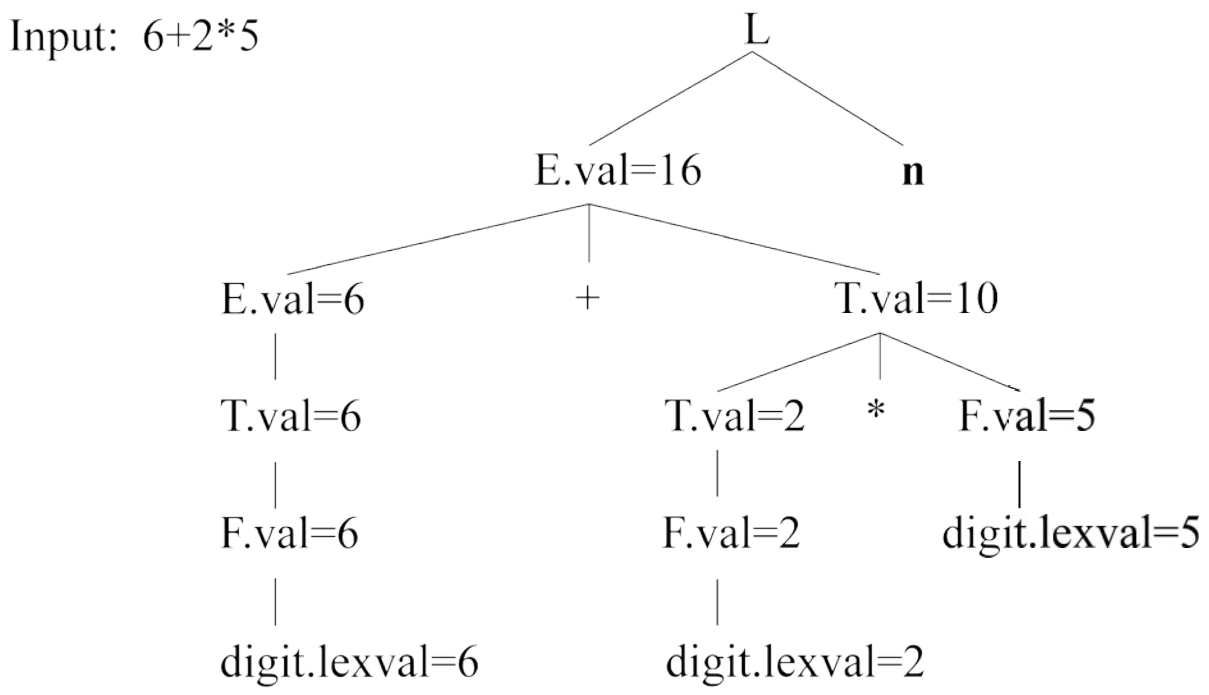
So, a semantic rule $b=f(c_1,c_2,\ldots,c_n)$ indicates that the attribute b *depends* o*n* attributes $c_1,c_2,\ldots,c_n$.

Example (Syntax Directed Definition)

<u>Production</u>	<u>Semantic Rules</u>
L → E **n**	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

Note :- All attributes in this example here are of synthesized types. Symbols E, T, and F are associated with a synthesized attribute *val*. The token **digit** has a synthesized attribute *lexval* (it is assumed that it is evaluated by the lexical analyzer).

Annotated Parse Tree -- Example



Inherited Attributes

An inherited attribute at any node is defined based on the attributes at the parent and/or siblings of the node. Useful for describing context sensitive behavior of grammar symbols. For example, an inherited attribute can be used to keep track of whether an identifier appears at the left or right side of an assignment operator. This can be used to decide whether the address or the value of the identifier needed.

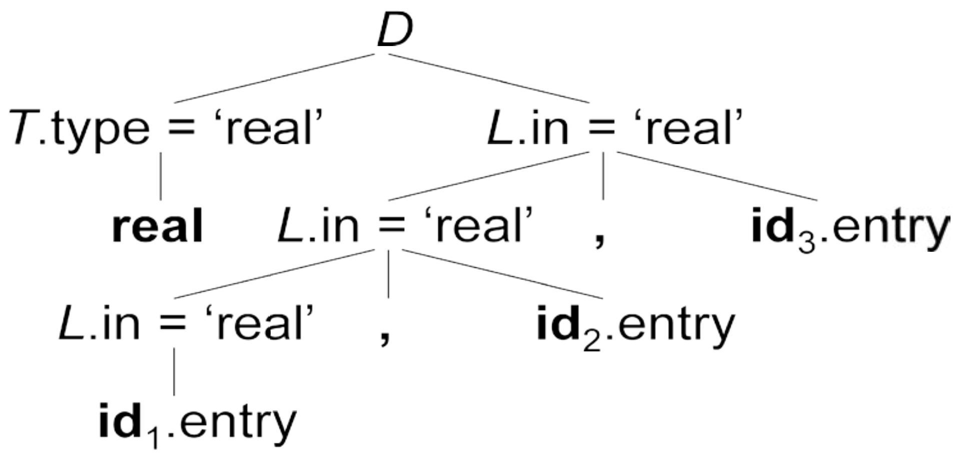
Example

Production	Semantic Rules
$D \rightarrow T\ L$	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)

Symbol T is associated with a synthesized attribute *type*. Symbol L is associated with an inherited attribute *in*.

Example (Annotated Parse Tree)

Input :- real id1, id2, id3



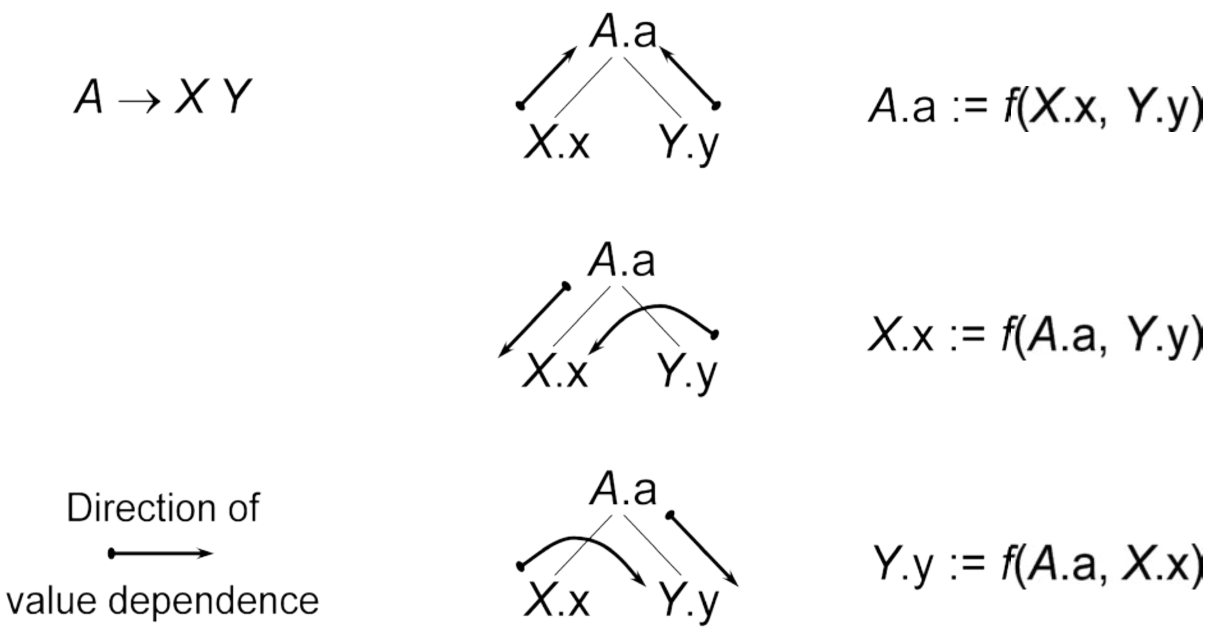
Dependency Graph

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.

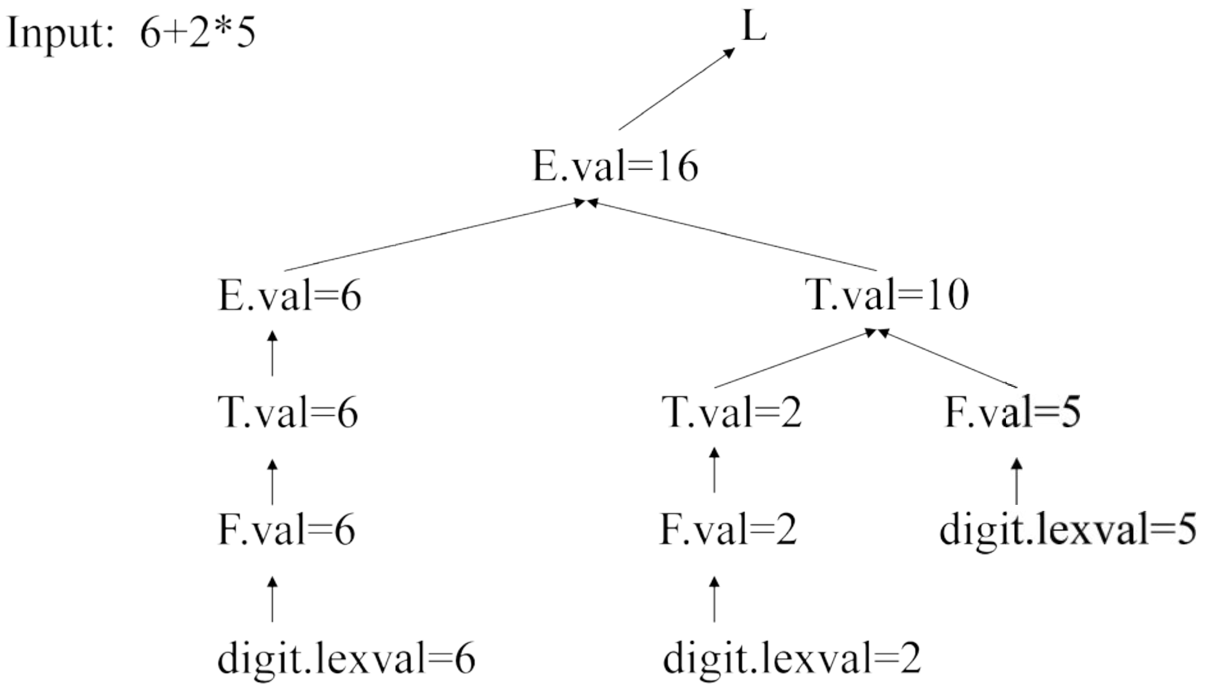
Algorithm: Dependency graph construction

```
for each node n in the parse tree do
    for each attribute a of the grammar symbol n do
        Construct a node in the dependency graph for a
    done
done
for each node n in the parse tree do
    for each semantic rule of the form $b = f(c_1,c_2, ...,c_n)$ associated with the production used at n do
        for i = 1 to k do
            construct an edge from $c_i$ to b
        done
    done
done
```

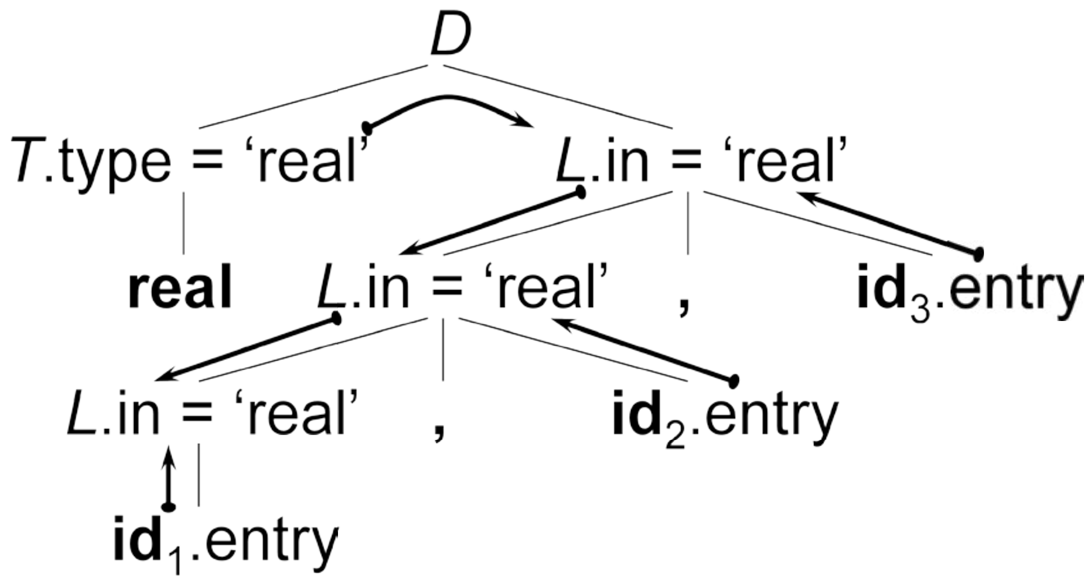
Acyclic Dependency Graphs for Parse Trees



Example (Dependency Graph for Annotated Parse Tree for *6 + 2 * 5*)



Example (Annotated Parse Tree with Dependency Graph for *real id1, id2, id3*)



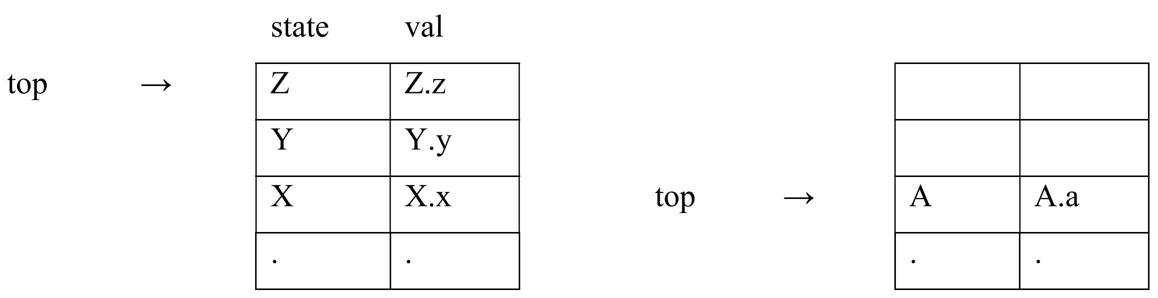
S – Attributed Definitions

Syntax-directed definitions are used to specify syntax-directed translations. Syntax-directed definitions are used to specify syntax-directed translation. An S-attributed grammar can be translated bottom-up as and when the grammar is being parsed using an LR parser.

Bottom-Up Evaluation of S-Attributed Definitions

The bottom up parser uses a stack to hold information about sub tress that have been parsed. Let us suppose that the stack is implemented by a pair of arrays *state* and *val*. If the ith *state* symbol is A, then *val[i]* will hold the value of the attribute associate with the parse tree node corresponding to A. The current top of the stack is indicated by the pointer *top*. We assume that synthesized attributes are evaluated just before each reduction. Let us see the following example.

A → XYZ A.a=f(X.x,Y.y,Z.z) where all attributes are synthesized. Before XYZ is reduced to A, the value of the attribute Z.z is in *val[top]*, that of Y.y in *val[top – 1]*, and that of X.x in *val[top – 2]*. If a symbol has no attribute, then the corresponding entry in the *val* array is undefined. After the reduction, *top* is decremented by 2, the state covering A is put in *state[top],* i.e. where X was, and the value of the synthesized attribute A.a is put in *val[top]*.



Example

At each shift of **digit**, we also push **digit.lexval** into *val-stack*

stack	val-stack	input	action	semantic rule
0		6+2*5n	s6	d.lexval(6) into val-stack
0d6	6	+2*5n	$F \rightarrow d$	F.val=d.lexval – do nothing
0F4	6	+2*5n	$T \rightarrow F$	T.val=F.val – do nothing
0T3	6	+2*5n	$E \rightarrow T$	E.val=T.val – do nothing
0E2	6	+2*5n	s8	push empty slot into val-stack
0E2+8	6-	2*5n	s6	d.lexval(2) into val-stack
0E2+8d6	6-2	*5n	$F \rightarrow d$	F.val=d.lexval – do nothing
0E2+8F4	6-2	*5n	$T \rightarrow F$	T.val=F.val – do nothing
0E2+8T11	6-2	*5n	s9	push empty slot into val-stack
0E2+8T11*9	6-2-	5n	s6	d.lexval(5) into val-stack
0E2+8T11*9d6	6-2-5	n	$F \rightarrow d$	F.val=d.lexval – do nothing
0E2+8T11*9F12	6-2-5	n	$T \rightarrow T*F$	T.val=T_1.val*F.val
0E2+8T11	6-10	n	$E \rightarrow E+T$	E.val=E_1.val*T.val
0E2	16	n	s7	push empty slot into val-stack
0E2n7	16-	$	$L \rightarrow Er$	print(16), pop empty slot from val-stack
0L1	16	$	acc	

L – Attributed Definition

A syntax-directed definition is L-attributed if each inherited attribute of X_j on the right side of $A \rightarrow X_1 X_2 \ldots X_n$ depends only on

- The attributes of the symbols $X_1, X_2, \ldots, X_{j-1}$
- The inherited attributes of A

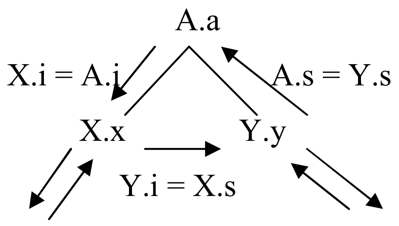
L – attributed definitions allow for a natural order of evaluating attributes: depth-first and left to right. Every S-attributed syntax-directed definition is also L-attributed and the above rule applied only for inherited attributes.

Depth First Evaluation of L – Attributed Definitions

```
Procedure dfvisit(n: node); //depth-first evaluation
        for each child m of n, from left to right do
                evaluate inherited attributes of m;
                dfvisit(m);
        evaluate synthesized attributes of n
```

Example

$A \rightarrow XY$



X.i = A.i

Y.i = X.s

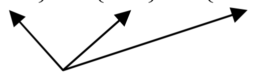
A.s = Y.s

Translation Scheme

A translation scheme is a context-free grammar in which:

- attributes are associated with the grammar symbols
- semantic actions enclosed between braces {} are inserted within the right sides of productions

Ex: $A \rightarrow$ { ... } X { ... } Y { ... }



Semantic Actions

In translation schemes, we use semantic action terminology instead of semantic rule terminology used in syntax-directed definitions. The position of the semantic action on the right side indicates when that semantic action will be evaluated

Example

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

E → T R

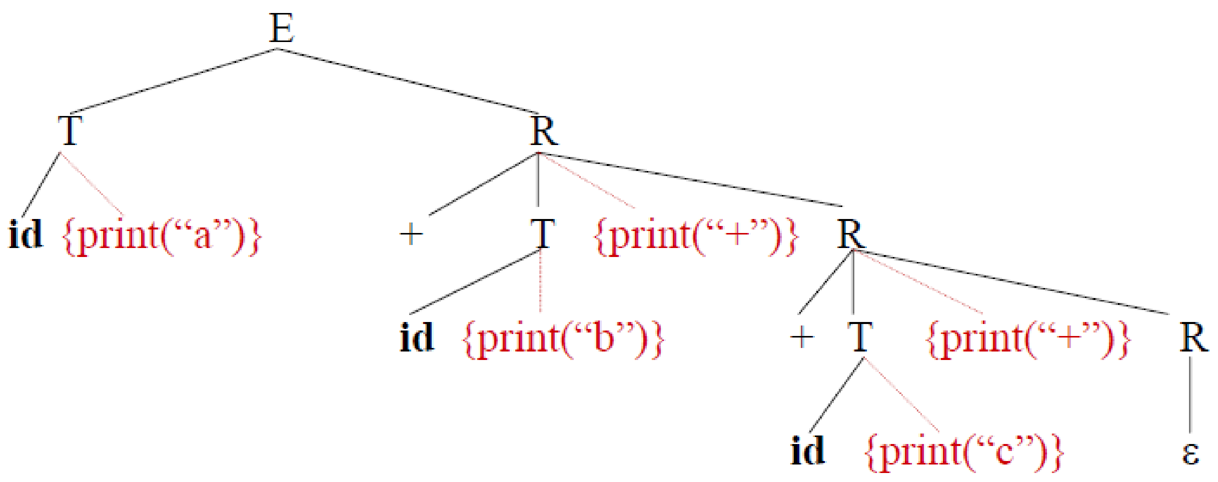
R → + T { print("+") } R1

R → ε

T → id { print(id.name) }

a+b+c → ab+c+

infix expression postfix expression



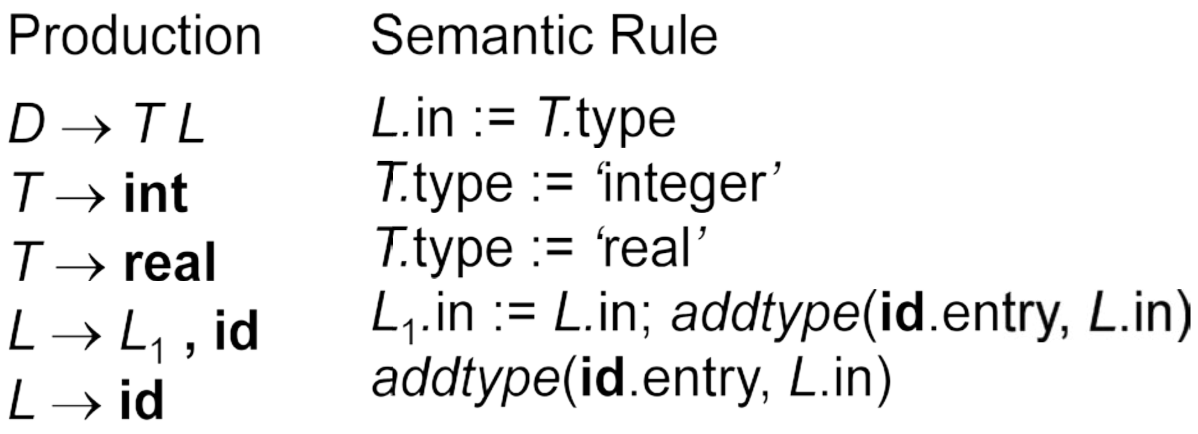
The depth first traversal of the parse tree (executing the semantic actions in that order) will produce the postfix representation of the infix expression.

Translation Schemes Requirements

If a translation scheme has to contain both synthesized and inherited attributes, we have to observe the following rules:

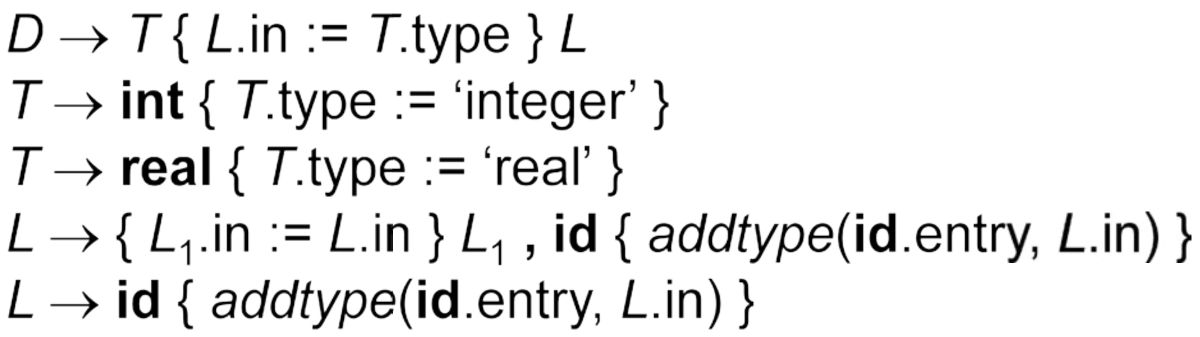
1. An inherited attribute of a symbol on the right side of a production must be computed in a semantic action before that symbol.
2. A semantic action must not refer to a synthesized attribute of a symbol to the right of that semantic action.
3. A synthesized attribute for the non-terminal on the left can only be computed after all attributes it references have been computed (we normally put this semantic action at the end of the right side of the production).

Using Translation Schemes for L-Attributed Definitions



Production	Semantic Rule
$D \rightarrow T\ L$	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)

Translation Scheme



$D \rightarrow T$ { L.in := T.type } L
$T \rightarrow$ **int** { T.type := 'integer' }
$T \rightarrow$ **real** { T.type := 'real' }
$L \rightarrow$ { L_1.in := L.in } L_1 , **id** { *addtype*(**id**.entry, L.in) }
$L \rightarrow$ **id** { *addtype*(**id**.entry, L.in) }

Top-Down Translation

L-attributed definitions can be evaluated in a top-down fashion (predictive parsing) with translation schemes. The algorithm for elimination of left recursion is extended to evaluate action and attribute. Attributes in L-attributed definitions implemented in translation schemes are passed as arguments to procedures (synthesized) or returned (inherited).

Eliminating Left Recursion from a Translation Scheme

We can implement translation of L-attributed definition for top down parser by using the method that is similar to the removal of left recursion from the grammar with the following rule:

$$\begin{array}{ll} A \to A_1\ Y & \{\ A.\mathrm{a} := g(A_1.\mathrm{a},\ Y.\mathrm{y})\ \} \\ A \to X & \{\ A.\mathrm{a} := f(X.\mathrm{x})\ \} \end{array}$$

Figure :- a left recursive grammar with synthesized attributes (a,y,x).

Eliminate Left Recursion

$$\begin{array}{l} A \to X\ \{\ R.\mathrm{i} := f(X.\mathrm{x})\ \}\ \ R\ \ \{\ A.\mathrm{a} := R.\mathrm{s}\ \} \\ R \to Y\ \{\ R_1.\mathrm{i} := g(R.\mathrm{i},\ Y.\mathrm{y})\ \}\ \ R_1\ \ \{\ R.\mathrm{s} := R_1.\mathrm{s}\ \} \\ R \to \varepsilon\ \ \{\ R.\mathrm{s} := R.\mathrm{i}\ \} \end{array}$$

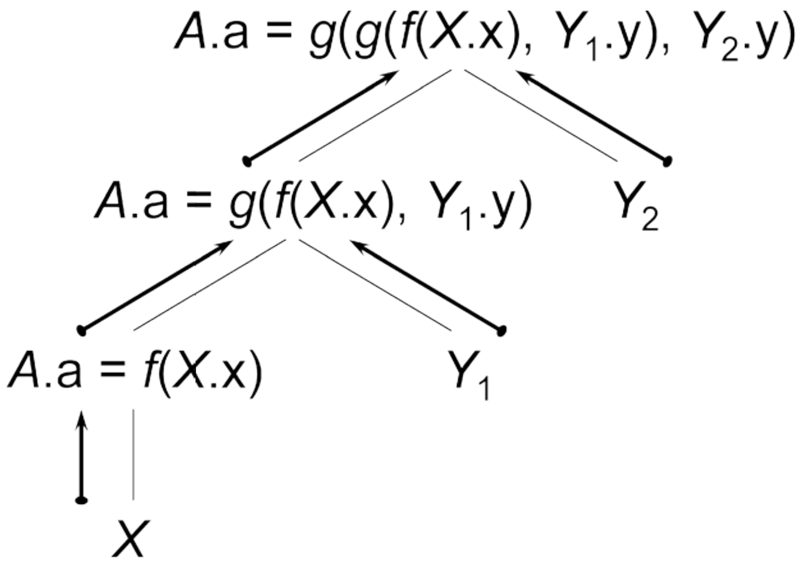
R.i → inherited attribute of the new non terminal

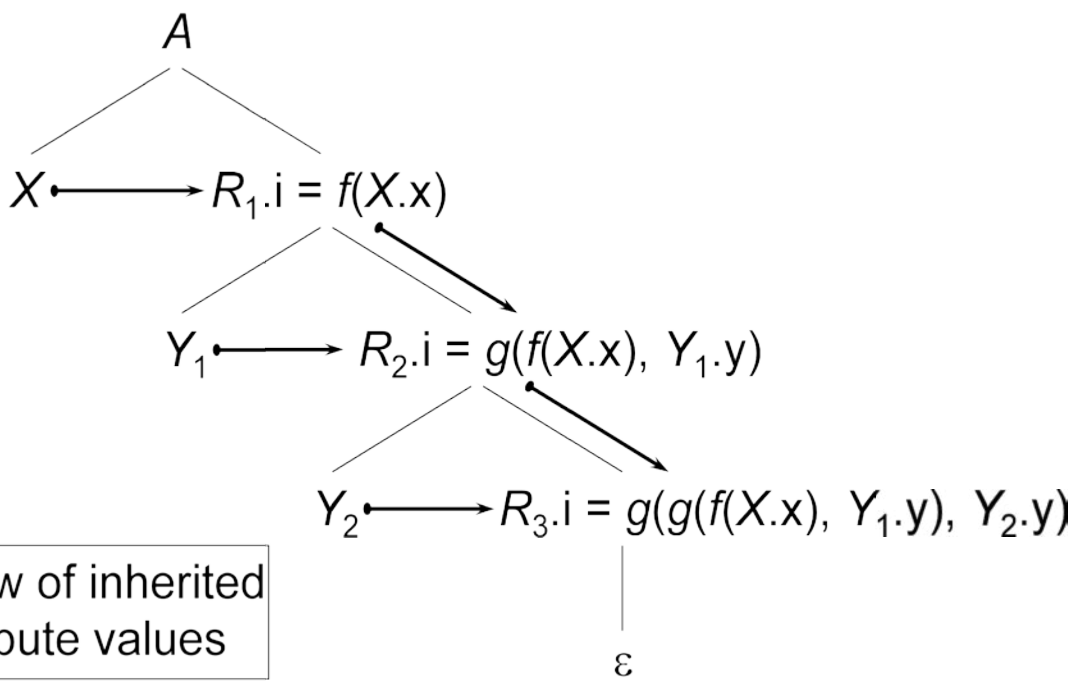
R.s → synthesized attribute of the new non terminal

When we eliminate the left recursion from the grammar (to get a suitable grammar for the top-down parsing) we also have to change semantic actions.

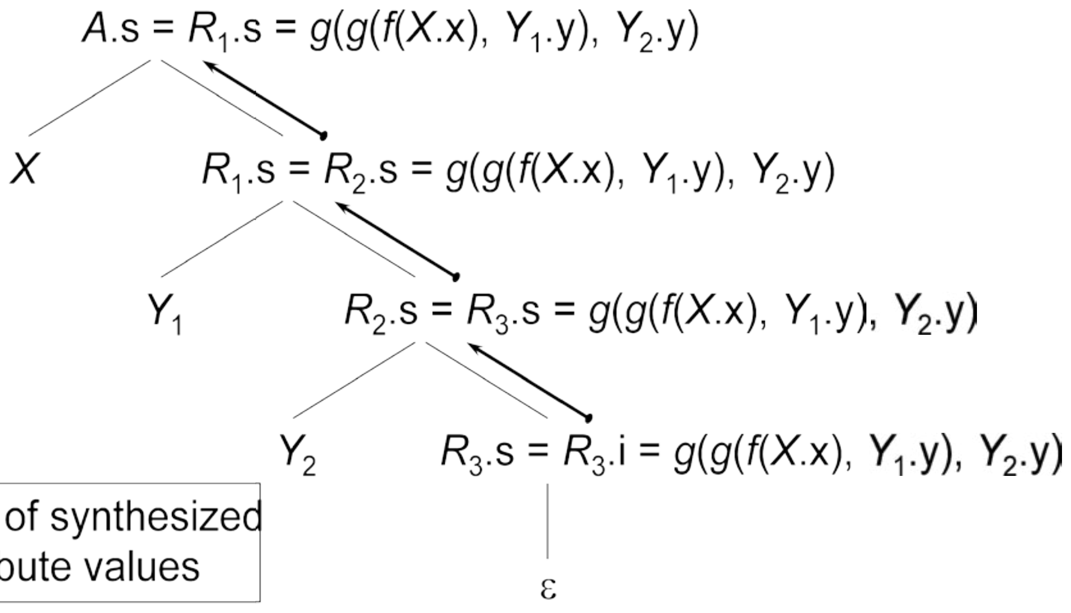
Top - Down Evaluation Example

Evaluation of string XYY


$A.\mathrm{a} = g(g(f(X.\mathrm{x}),\ Y_1.\mathrm{y}),\ Y_2.\mathrm{y})$
$A.\mathrm{a} = g(f(X.\mathrm{x}),\ Y_1.\mathrm{y})$
Y_2
$A.\mathrm{a} = f(X.\mathrm{x})$
Y_1
X


A
X
$R_1.i = f(X.x)$
Y_1
$R_2.i = g(f(X.x), Y_1.y)$
Y_2
$R_3.i = g(g(f(X.x), Y_1.y), Y_2.y)$
ε

1. Flow of inherited attribute values


$A.s = R_1.s = g(g(f(X.x), Y_1.y), Y_2.y)$
X
$R_1.s = R_2.s = g(g(f(X.x), Y_1.y), Y_2.y)$
Y_1
$R_2.s = R_3.s = g(g(f(X.x), Y_1.y), Y_2.y)$
Y_2
$R_3.s = R_3.i = g(g(f(X.x), Y_1.y), Y_2.y)$
ε

2. Flow of synthesized attribute values

Bottom-Up Evaluation of Inherited Attributes

If we are able to identify the position of the inherited attribute in the stack, then the simple assignment of the value stored in the stack may be used to implement the L-attributed definition in bottom up manner. The method used in Bottom-up Evaluation of S-Attributed Definitions is extended with conditions:

1. All embedding semantic actions in translation scheme should be in the end of the production rules.
2. All inherited attributes should be copied into the synthesized attributes (most of the time synthesized attributes of new non-terminals).
3. Evaluate all semantic actions during reductions.

Insert marker nonterminals to remove embedded actions from translation schemes, that is

$A \rightarrow X$ { actions } Y, is rewritten with marker nonterminal N into

$A \rightarrow X\ N\ Y$

$N \rightarrow \varepsilon$ { actions}

Problem: inserting a marker nonterminal may introduce a conflict in the parse table.

Example

E → T R

R → + T { print("+") } R1

R → ε

T → id { print(id.name) }

$\Longrightarrow$

E → T R

R → + T M R1

R → ε

T → id { print(id.name) }

M → ε { print("+") }

Example

Consider the following grammar

D → T {L.*in* = T.*type*} L

T → **int** {T.*type* = *integer*}

T → **real** {T.*type* = *real*}

L → L_1, id {*addtype*(id.*entry*, L.*in*)

L → id {*addtype*(id.*entry*, L.*in*)

Let us examine the moves made by a bottom – up parser on the input real id_1, id_2, id_3

Input	**State**	**Production Used**
real id_1, id_2, id_3	-	
id_1, id_2, id_3	real	
id_1, id_2, id_3	T	T → real
, id_2, id_3	T id1	
,id_2, id_3	T L	L → id
id_2, id_3	T L ,	
, id_3	T L , id2	
, id_3	T L	L → L_1, id
id_3	T L ,	
	T L , id3	
	T L	L → L_1, id
	D	D → TL

Exercise

1. Consider the grammar

   L → E **n**

   E → E_1 + T

   E → T

   T → T_1 * F

   T → F

   F → (E)

   F → **digit**

Write the semantic rule for above grammar and construct the annotated parse tree and dependency graph for the expression 2 + 4 * 7.

2. Eliminate the left recursion from syntax – directed definitions from the following grammar.

   E → E + T | T

   T → num.num | num

3. In Pascal, a programmer can declare two integer variables a and b with the syntax var a, b : int. Write a grammar for such declaration.
4. Construct a syntax directed translation scheme that translate 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.
5. For the grammar below, design an L – attributed syntax directed definition to compute S.val, the decimal value of an input string in binary. For example the translation of the string 101.101 should be the decimal number 5.625. Hint, use an inherited attribute L.pos that tells which side of the decimal point a given bit is on. For all symbols, specify their attributes, if they are inherited or synthesized, and the various semantic rules. Construct the annotated parse tree for the binary string 101.101.

   S → L.L | L

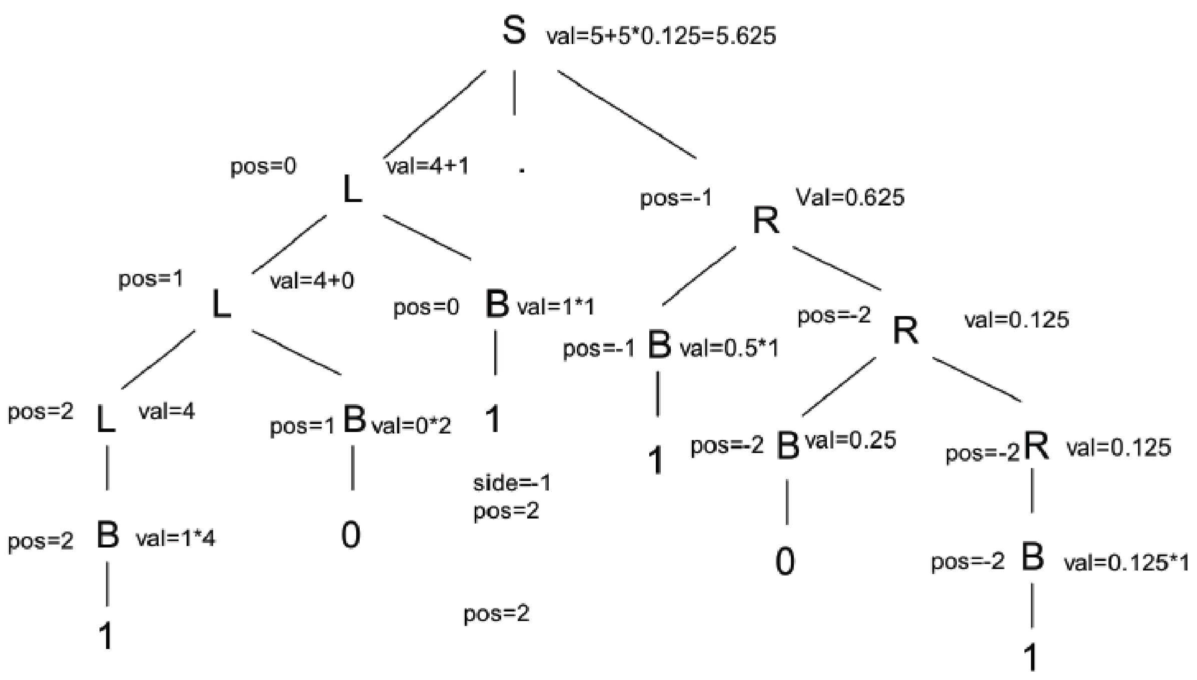
   L → L B | B

   B → 0 | 1
6. For the input expression (4 * 7 + 1) * 2, construct an annotated parse tree according to the syntax directed definition of exercise 1.

Solution to Q. No. 5

One possible solution is to define 2 attributes, 1 inherited and 1 synthesized, with *pos* inherited and *val* as synthesized attribute. Below we present the revised set of semantic actions and the example of the attributes for the input string "101.101".

Productions	**Semantic Rules**
S → L . R	L.pos = 0; R.pos = -1; S.val = L.val + R.val
S → L	L.pos = 0; S.val = L.val
L → L1 B	L1.pos = L.pos + 1; B.pos = L.pos; L.val = L1.val + B.val
L → B	B.pos = L.pos; L.val = B.val
R → R1 B	R1.pos = R.pos - 1; B.pos = R.pos; L.val = L1.val + B.val
R → B	B.pos = R.pos; L.val = B.val
B → 0	B.val = 0
B → 1	B.val = $1*2^{B.pos}$



Bibliography

- Aho, R. Sethi, J. Ullman, Compilers: Principles, Techniques, and Tools. 2008
- Notes from CDCSIT, TU of Samujjwal Bhandari and Bishnu Gautam
- Manuals of Srinath Srinivasa, Indian Institute of Information Technology, Bangalore