Syntax Directed Translation

Mrs. Sunita M Dol

(sunitaaher@gmail.com)

Assistant Professor, Dept of Computer Science and Engineering

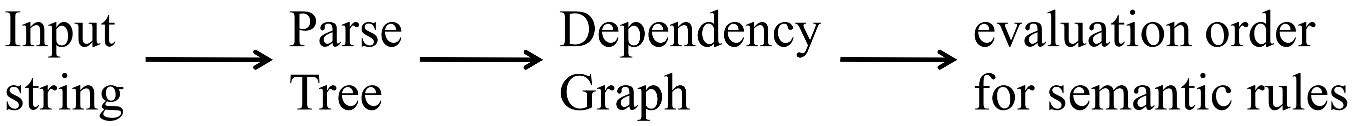
Walchand Institute of Technology, Solapur
(www.witsolapur.org)

Syntax Directed Translation

- Syntax directed definitions
- Construction of syntax tree
- Bottom-up evaluation of S-attributed definitions
- L-attributed definitions
- Top-down translation of inherited attributes
- Bottom-up evaluation of inherited attributes
- Analysis of syntax directed definitions.

Introduction

- Grammar symbols are associated with **attributes** to associate information with the programming language constructs that they represent.
- Values of these attributes are evaluated by the **semantic rules** associated with the production rules.
- An attribute may hold almost any thing.
  - a string, a number, a memory location, a complex record.
- Conceptually, we have the following flow:



Input string → Parse Tree → Dependency Graph → evaluation order for semantic rules

Introduction

- When we associate semantic rules with productions, we use two notations:
  - ✓ **Syntax-Directed Definitions**
  - ✓ **Translation Schemes**
- **Syntax-Directed Definitions:**
  - o Give high-level specifications for translations
  - o Hide many implementation details such as order of evaluation of semantic actions.
  - o We associate a production rule with a set of semantic actions, and we do not say when they will be evaluated.
- **Translation Schemes:**
  - o Indicate the order of evaluation of semantic actions associated with a production rule.
  - o In other words, translation schemes give a little bit information about implementation details.

Syntax-Directed Definitions

- 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 of that grammar symbol.
  - Each production rule is associated with a set of semantic rules.
- **Synthesized attributes** are computed from the values of the children of a node in the parse tree.
- **Inherited attributes** are computed from the attributes of the parents and/or siblings of a node in the parse tree.

Syntax-Directed Definitions

- **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.
- Evaluation of a semantic rule defines the value of an attribute. But a semantic rule may also have some side effects such as printing a value.

Syntax-Directed Definition

- 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.
- Of course, the order of these computations depends on the dependency graph induced by the semantic rules.

Syntax-Directed Definition

- In a syntax-directed definition, each production A→α 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 either

  ○ 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→α).

  ○ 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→α).

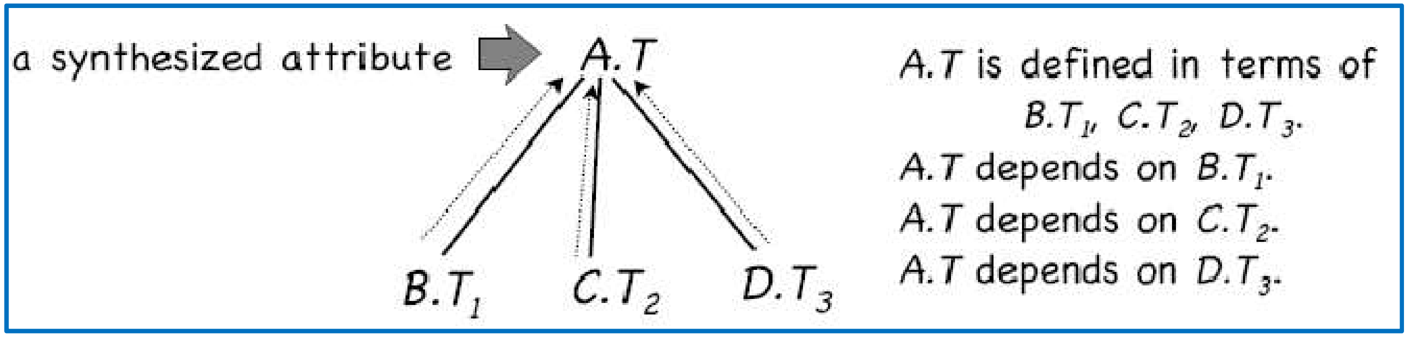
  in either case, we say b DEPENDS on c_1, c_2, …, c_k.

Syntax-Directed Definition

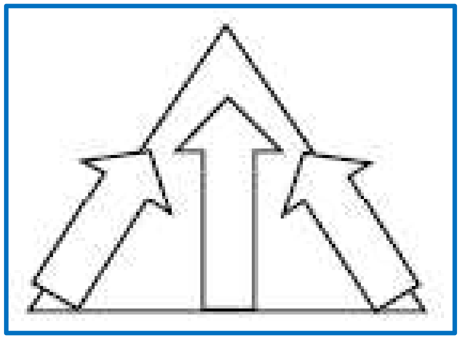
- So, a semantic rule $b=f(c_1,c_2,\ldots,c_n)$ indicates that the attribute b *depends* o*n* attributes $c_1,c_2,\ldots,c_n$.
- In a **syntax-directed definition**, a semantic rule may just evaluate a value of an attribute or it may have some side effects such as printing values or updating the symbol table then we write the rule as a procedure call.
- An **attribute grammar** is a syntax-directed definition in which the functions in the semantic rules cannot have side effects (they can only evaluate values of attributes).

Syntax-Directed Definition

- **Synthesized Attribute**
  - Synthesized attributes depend only on the attributes of children. They are the most common attribute type.



$A.T$ is defined in terms of $B.T_1$, $C.T_2$, $D.T_3$.
$A.T$ depends on $B.T_1$.
$A.T$ depends on $C.T_2$.
$A.T$ depends on $D.T_3$.



  - If a SDD has synthesized attributes only, it is called a s-attributed definition.
  - S-attributed definitions are convenient since the attributes can be calculated in a bottom-up traversal of the parse tree.

Syntax-Directed Definition

- **Synthesized Attribute**
  - Example Syntax-Directed Definition: desk calculator

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

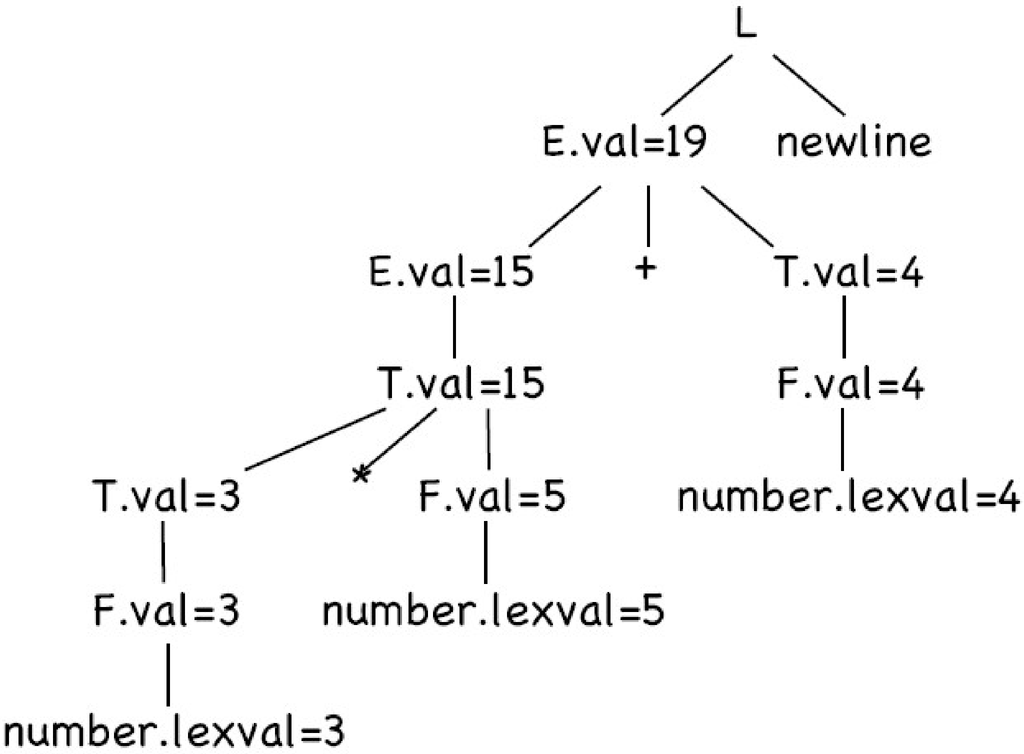
  - Symbols E, T, and F are associated with a synthesized attribute *val*.
  - The token **digit** has a synthesized attribute *lexval* (it is assumed that it is evaluated by the lexical analyzer).

Syntax-Directed Definition

- **Synthesized Attribute**

Production	**Semantic Rules**
$L \rightarrow E$ **n**	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

Input: 5+3*4



Syntax-Directed Definition

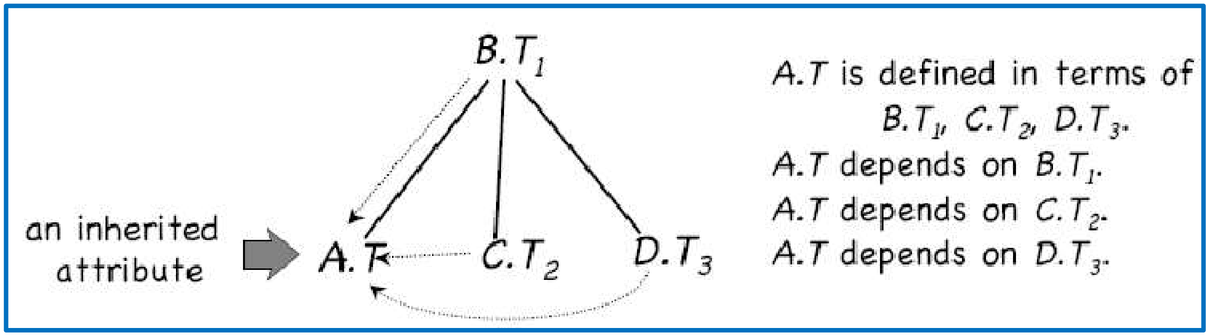
- **Synthesized Attribute**

Production	**Semantic Rules**
$E \rightarrow E_1 + T$	E.loc=newtemp(), E.code = E_1.code || T.code || add E_1.loc,T.loc,E.loc
$E \rightarrow T$	E.loc = T.loc, E.code=T.code
$T \rightarrow T_1 * F$	T.loc=newtemp(), T.code = T_1.code || F.code || mult T_1.loc,F.loc,T.loc
$T \rightarrow F$	T.loc = F.loc, T.code=F.code
$F \rightarrow (E)$	F.loc = E.loc, F.code=E.code
$F \rightarrow$ **id**	F.loc = **id**.name, F.code=“”

- Symbols E, T, and F are associated with synthesized attributes ***loc*** and ***code***.
- The token ***id*** has a synthesized attribute ***name*** (it is assumed that it is evaluated by the lexical analyzer).
- It is assumed that || is the string concatenation operator.

Syntax-Directed Definition

- **Inherited Attributes**
  - An inherited attribute is defined in terms of the attributes of the node's parents and/or siblings.



$A.T$ is defined in terms of $B.T_1$, $C.T_2$, $D.T_3$.
$A.T$ depends on $B.T_1$.
$A.T$ depends on $C.T_2$.
$A.T$ depends on $D.T_3$.

  - Inherited attributes are often used in compilers for passing contextual information forward, for example, the type keyword in a variable declaration statement.

Syntax-Directed Definition

- **Inherited Attributes**
  - **Example –** A declaration generated by the nonterminal D in the syntax directed definition consist of keywords int or real followed by a list of identifiers.

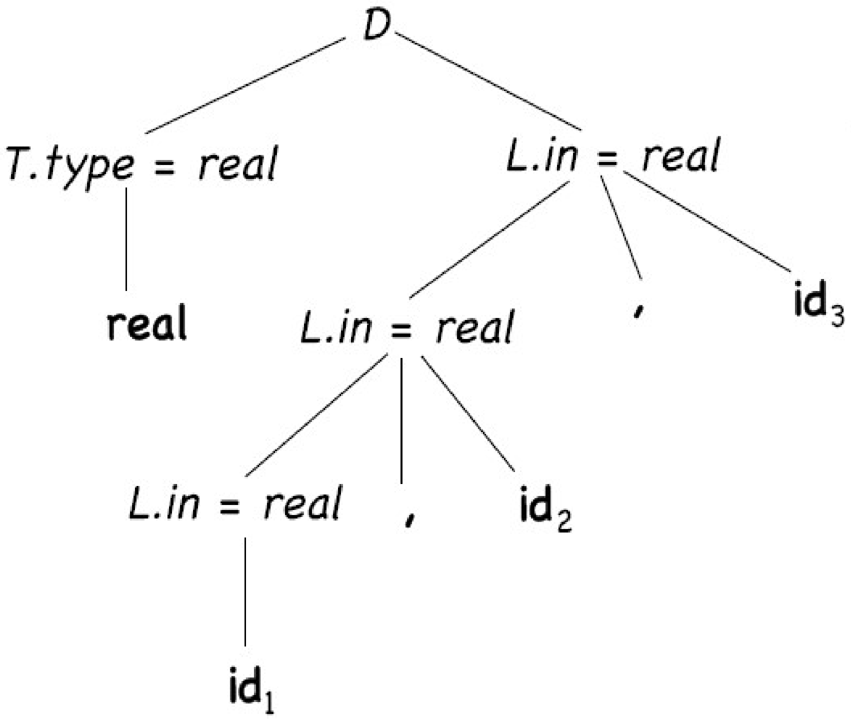
Production	**Semantic Rules**
D → T L	L.in = T.type
T → **int**	T.type = integer
T → **real**	T.type = real
L → L_1 **id**	L_1.in = L.in, addtype(**id**.entry,L.in)
L → **id**	addtype(**id**.entry,L.in)

addtype() is just a procedure that sets the type field in the symbol table.

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

Syntax-Directed Definition

Production	**Semantic Rules**
D → T L	L.in = T.type
T → **int**	T.type = integer
T → **real**	T.type = real
L → L_1 **id**	L_1.in = L.in, addtype(**id**.entry,L.in)
L → **id**	addtype(**id**.entry,L.in)



Syntax-Directed Definition

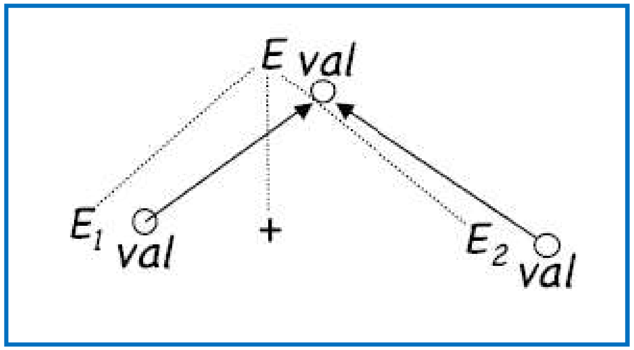
- **Dependency Graph**
  - If an attribute b depends on attribute c, then attribute b has to be evaluated AFTER c.
  - Dependency graph visualize these requirements.
    - ✓ Each attribute is a node
    - ✓ We add edges from the node for attribute c to the node for attribute b, if b depends on c.
    - ✓ For procedure calls, we introduce a dummy synthesized attribute that depends on the parameters of the procedure calls.

Syntax-Directed Definition

- **Dependency Graph**

Wherever this rule appears in the parse, tree we draw:

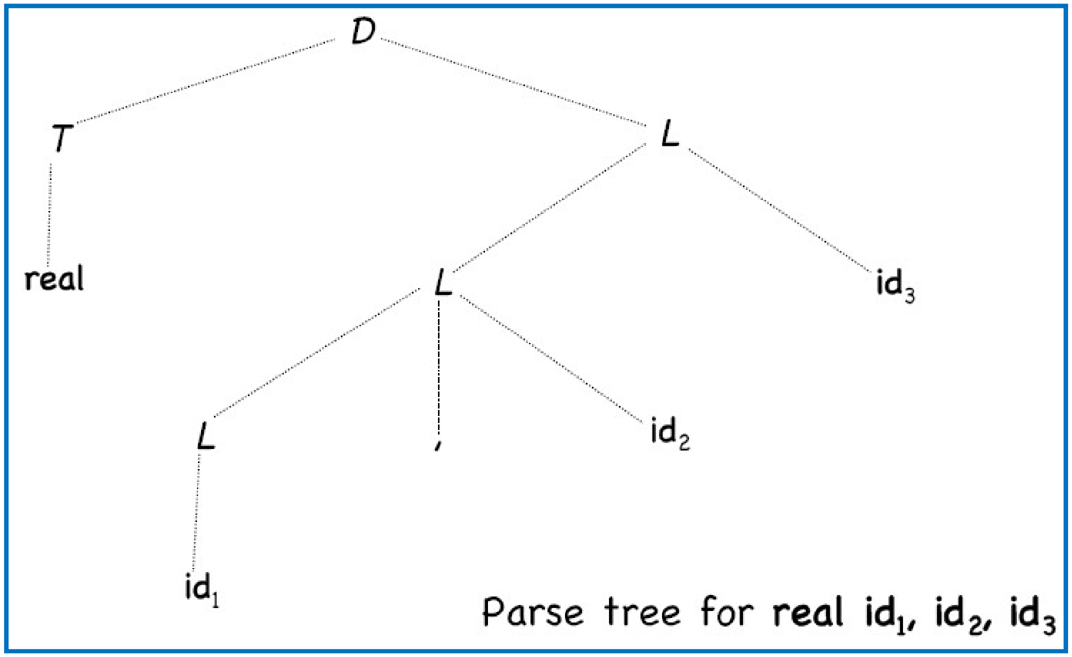
Production	**Semantic Rule**
E -> E1 + E2	E.val := E1.val + E2.val



Syntax-Directed Definition

- **Dependency Graph**

Production	**Semantic Rules**
D → T L	L.in = T.type
T → **int**	T.type = integer
T → **real**	T.type = real
L → L_1 **id**	L_1.in = L.in, addtype(**id**.entry,L.in)
L → **id**	addtype(**id**.entry,L.in)

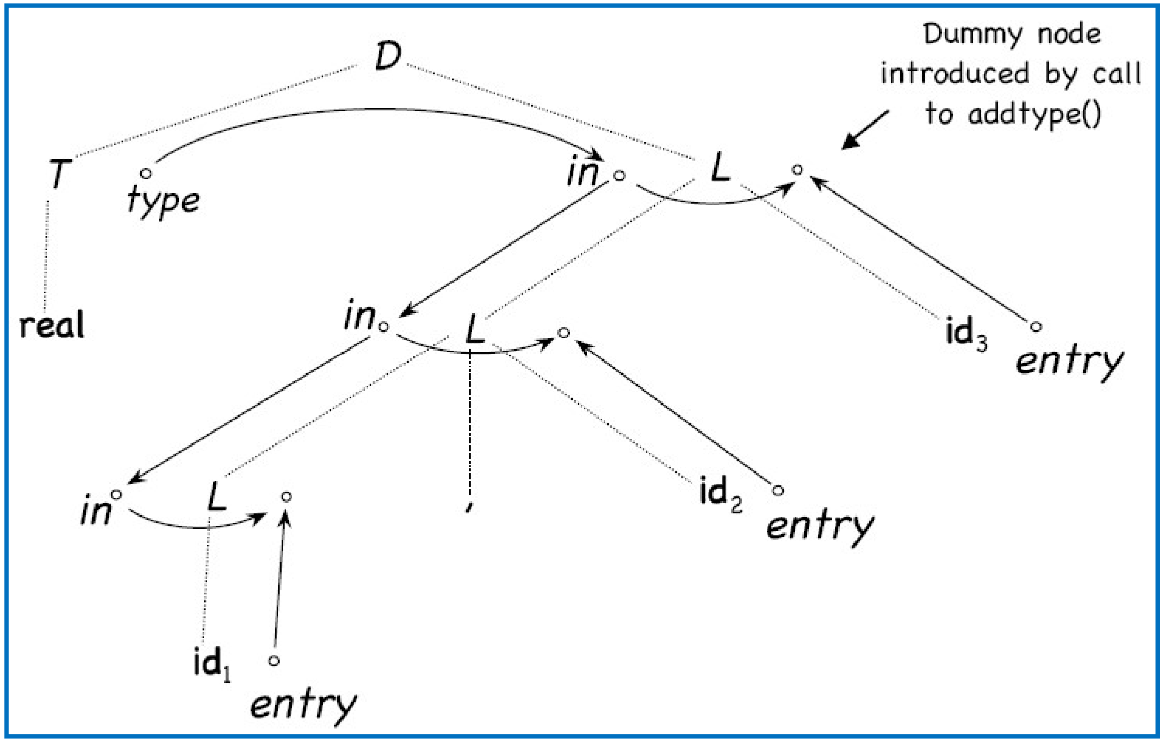


Parse tree for **real** id_1, id_2, id_3

Syntax-Directed Definition

- **Dependency Graph**

Production	**Semantic Rules**
D → T L	L.in = T.type
T → **int**	T.type = integer
T → **real**	T.type = real
L → L_1 **id**	L_1.in = L.in, addtype(**id**.entry,L.in)
L → **id**	addtype(**id**.entry,L.in)



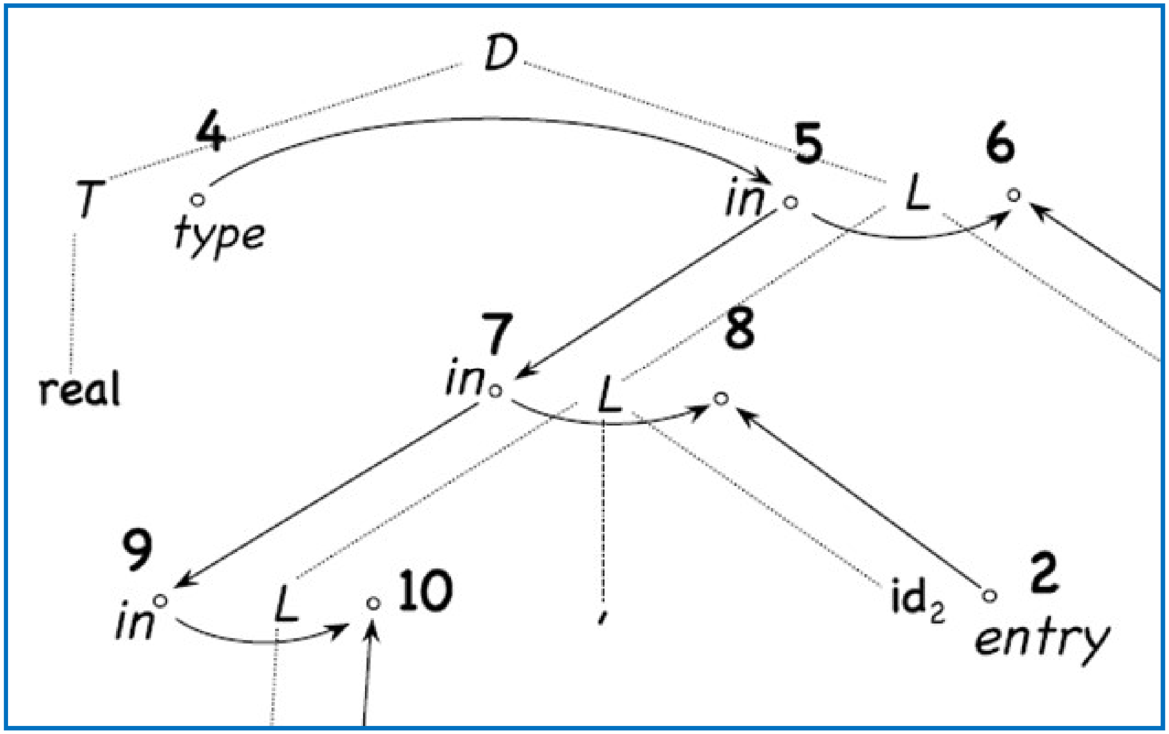
Syntax-Directed Definition

- **Evaluation Order**
  - A topological sort of a directed acyclic graph orders the nodes so that for any nodes a and b such that a -> b, a appears BEFORE b in the ordering.
  - There are many possible topological orderings for a DAG.
  - Each of the possible orderings gives a valid order for evaluation of the semantic rules.

Syntax-Directed Definition

- **Evaluation Order**

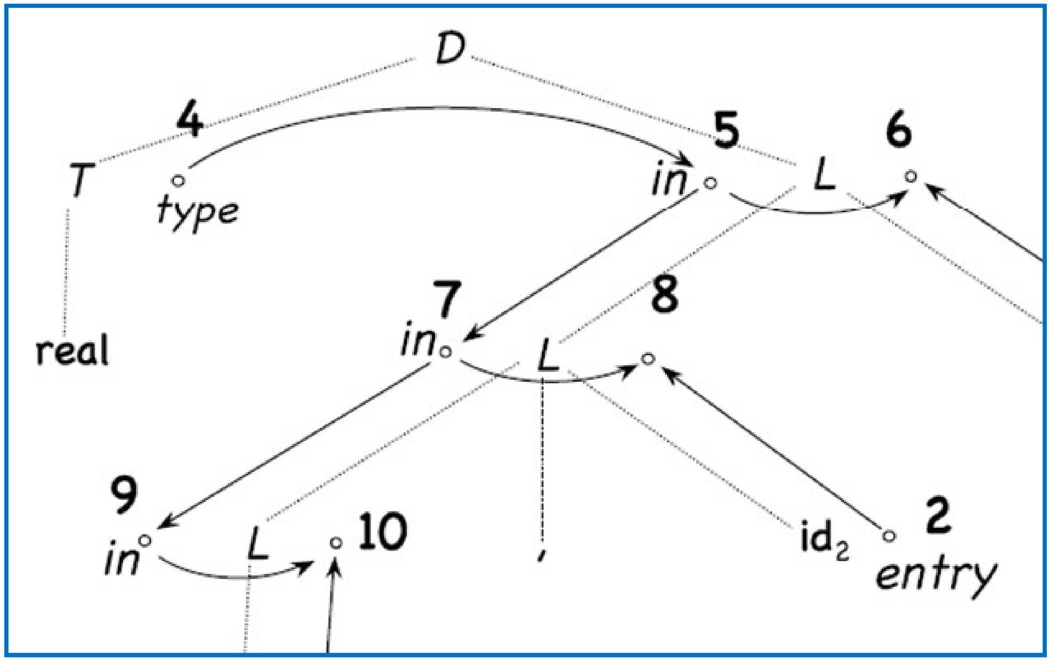
Production	Semantic Rules
D → T L	L.in = T.type
T → **int**	T.type = integer
T → **real**	T.type = real
L → L_1 **id**	L_1.in = L.in, addtype(**id**.entry,L.in)
L → **id**	addtype(**id**.entry,L.in)



Syntax-Directed Definition

- **Evaluation Order**
  - From topological sort, we obtain the following program

```
a4 := real;
a5 :=a4;
addtype(id3.entry, a5);
a7 :=a5;
addtype(id2.entry, a7);
a9 :=a7
addtype(id1.entry, a9);
```

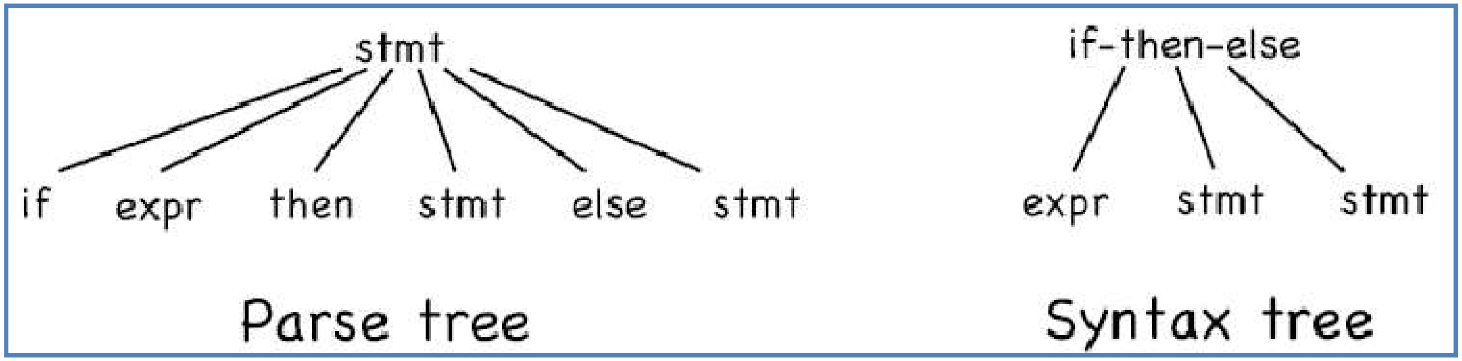


Syntax-Directed Definition

- **Evaluation Order**
  - In addition to topological sort, there are three methods for evaluating semantic rules:
  - **Parse tree method**: At compile time, these method obtain an evaluation order from topological sort of the dependency graph constructed from parse tree for each input.
  - **Rule based method**: At compiler-construction time, semantic rules associated with productions are analyzed either by hand or by a specialized tool.
  - **Oblivious method**: An evaluation order is chosen without considering semantic rules.

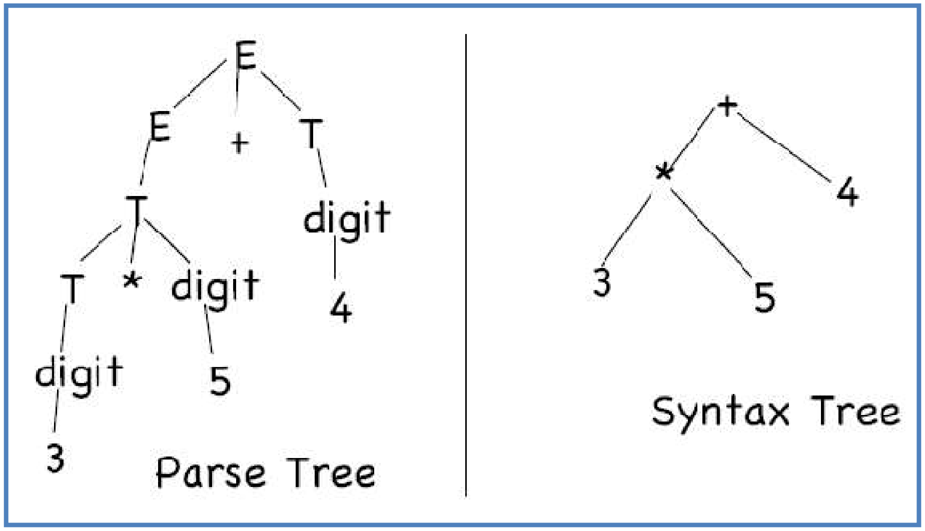
Construction of Syntax Tree

- **Syntax Trees**
  - One thing syntax directed definitions are useful for is construction of syntax trees.
  - A syntax tree is a condensed form of parse tree.
  - Syntax trees are useful for representing programming language constructs like expressions and statements.
  - They help compiler design by decoupling parsing from translation.



Construction of Syntax Tree

- **Syntax Tree**



  - Leaf nodes for operators and keywords are removed.
  - Internal nodes corresponding to uninformative non-terminals are replaced by the more meaningful operators.

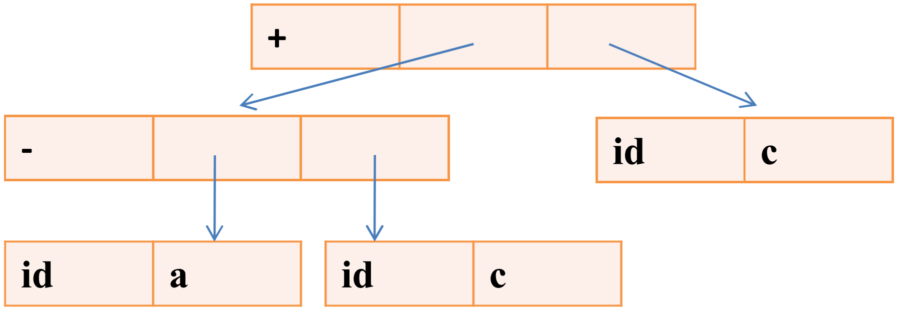
Construction of Syntax Tree

- **SDD for syntax tree construction**
  - We need some functions to help us build the syntax tree:
    - ✓ ***mknode(op,left,right)*** constructs an operator node with label ***op***, and two children, ***left*** and ***right***
    - ✓ ***mkleaf(id,entry)*** constructs a leaf node with label ***id*** and a pointer to a symbol table entry
    - ✓ ***mkleaf(num,val)*** constructs a leaf node with label ***num*** and the token's numeric value ***val***

Construction of Syntax Tree

- **SDD for syntax tree construction**
  - Use these functions to build a syntax tree for a-4+c:

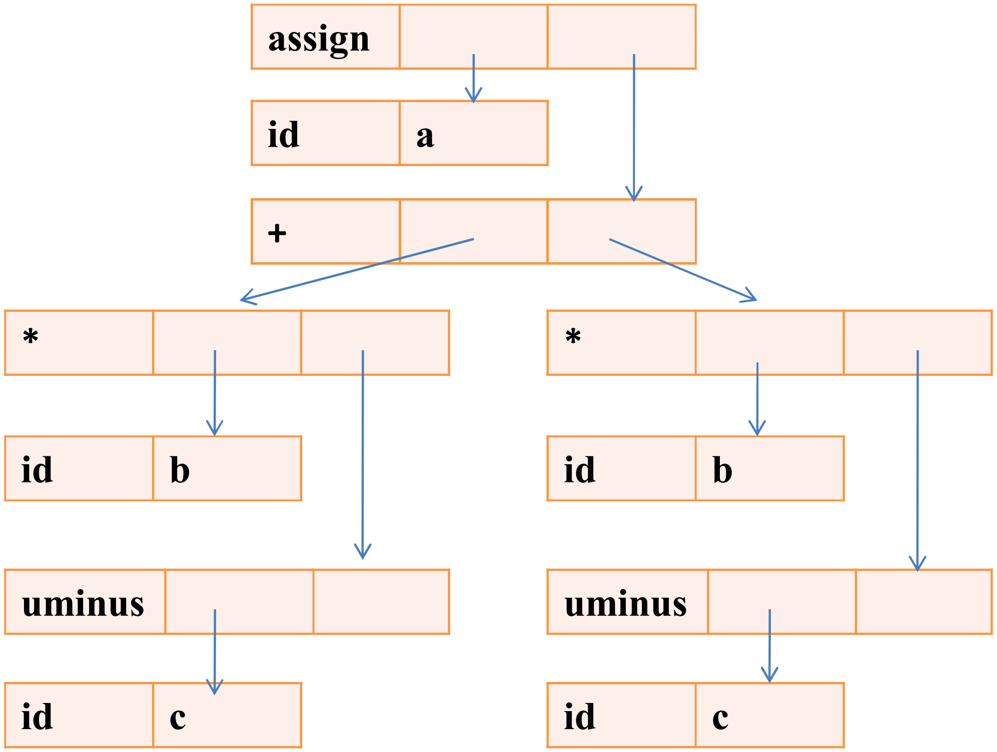
```
P1 := mkleaf( id, entrya );
P2 := mkleaf( num, 4);
P3 := mknode (‘-’ , P1, P2);
P4 := mkleaf (id, entryc);
P5 := mknode (‘+’, P3, P4)
```



Construction of Syntax Tree

- **SDD for syntax tree construction**
  - Use these functions to build a syntax tree for a : = b * - c + b* - c. :

```
P1 := mkleaf( id, entryc );
P2 := mknode (‘uminus’ , P1);
P3 := mkleaf (id, entryb);
P5 := mknode (‘*’, P3, P2)
P6 := mkleaf( id, entryc );
P7 := mknode (‘uminus’ , P6);
P8 := mkleaf (id, entryb);
P9 := mknode (‘*’, P8, P7)
P10 := mknode (‘+’, P5, P9)
P11 := mkleaf (id, entrya);
P12 := mknode (‘:=’, P11, P10);
```



Construction of Syntax Tree

- **SDD for syntax tree construction**

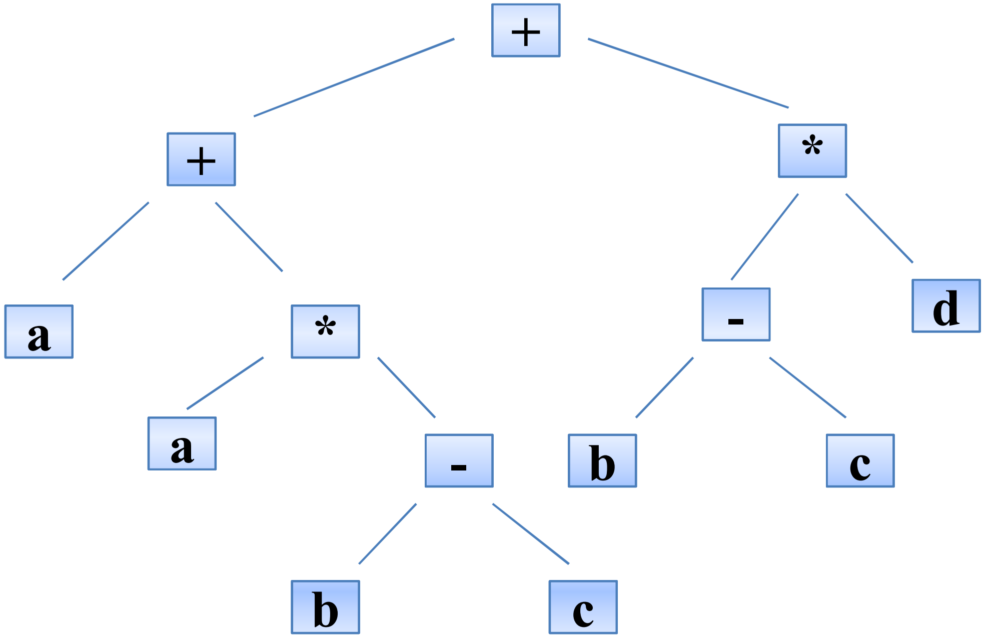
Production	Semantic Rules
E -> E_1 + T	E.nptr := mknode(‘+’, E_1.nptr,T.nptr)
E -> E_1 - T	E.nptr := mknode(‘-’, E_1.nptr,T.nptr)
E -> T	E.nptr := T.nptr
T -> (E)	T.nptr := E.nptr
T -> id	T.nptr := mkleaf(id, id.entry)
T -> num	T.nptr := mkleaf(num, num.val)

Construction of Syntax Tree

- **Directed Acyclic Graph for Expression**
  - A directed acyclic graph called a DAG for an expression identifies the common subexpression in the expressions.
  - Interior nodes represent an operator & its children represent its operand.
  - The difference between syntax tree & DAG is that a node in a DAG representing a common subexpression has more than one parent; in a syntax tree, the common subexpression would be represented as a duplicated subtree.
  - Example - a + a * a (b – c) + (b – c) * d

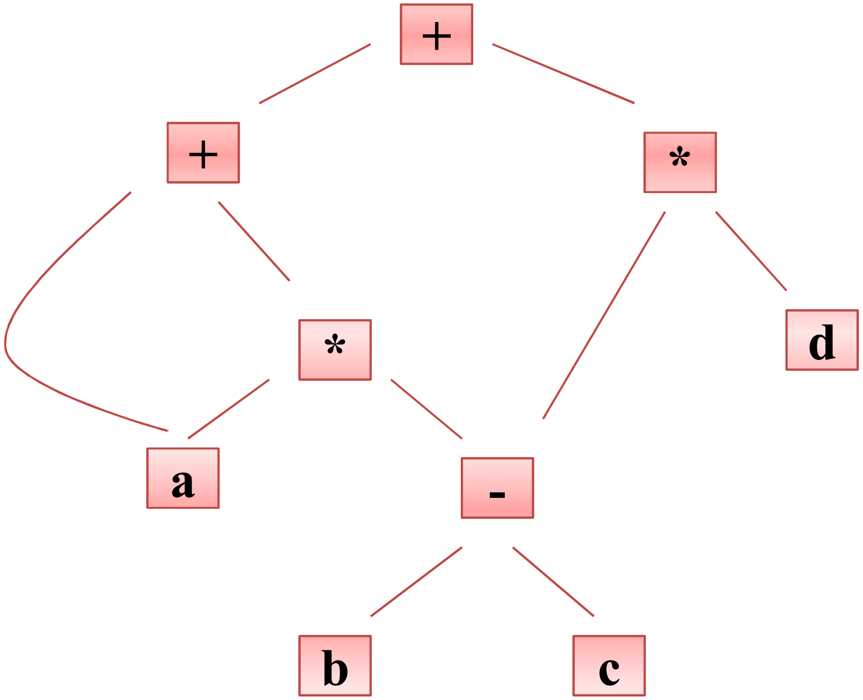
Construction of Syntax Tree

- **Directed Acyclic Graph for Expression**
  - Syntax Tree for a + a * (b – c) + (b – c) * d



Construction of Syntax Tree

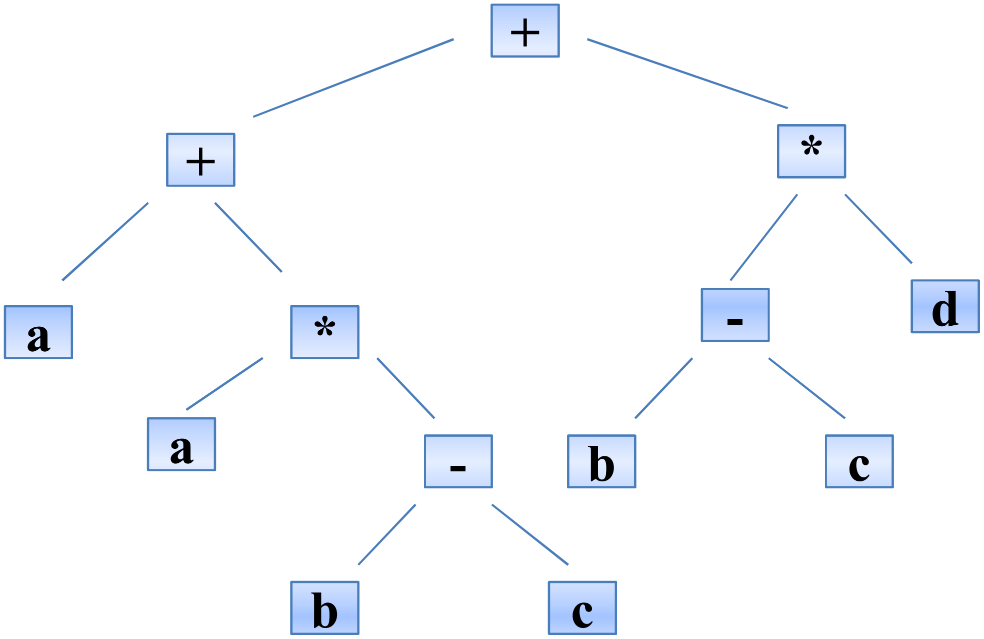
- **Directed Acyclic Graph for Expression**
  - DAG for a + a * (b – c) + (b – c) * d



Construction of Syntax Tree

- **Directed Acyclic Graph for Expression**
  - The sequence of instruction shown below constructs the DAG

```
P1 :=  mkleaf (id, a)
P2 := mkleaf(id, a)
P3 := mkleaf(id, b)
P4 := mkleaf(id, c)
P5 := mknode(‘-’, P3, P4)
P6 := mknode(‘*’, P2, P5)
P7 := mknode(‘+’, P1, P6)
P8 := mkleaf(id, b)
P9 := mkleaf(id, c)
P10 := mknode(‘-’, P8, P9)
P11 := mkleaf(id, d)
P12 := mknode(‘*’, P10, P11)
P13 := mknode(‘+’, P7, P12)
```

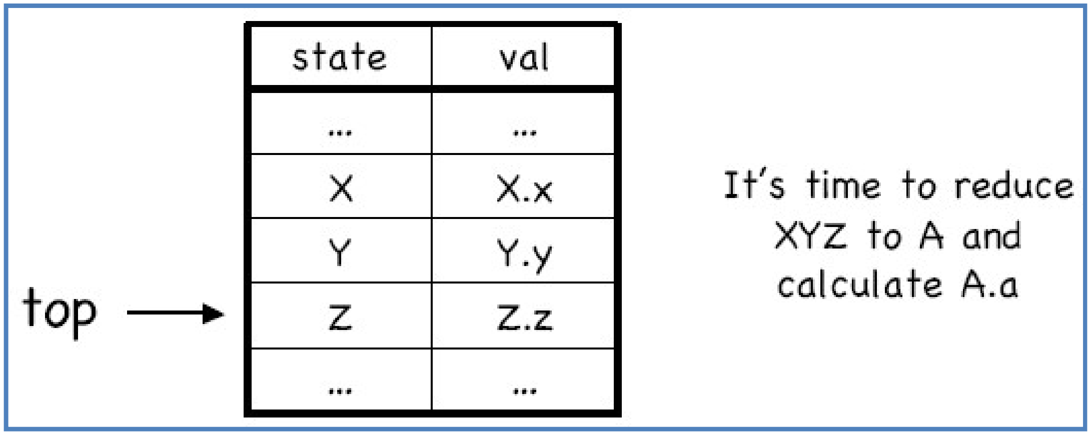


Bottom-up evaluation of S-attributed definitions

- **Synthesized Attributes on the Parser Stack**
  - A bottom-up shift-reduce parser can evaluate the (synthesized) attributes as the input is parsed.
  - We store the computed attributes with the grammar symbols and states on the stack.
  - When a reduction is made, we calculate the values of any synthesized attributes using the already-computed attributes from the stack.

Bottom-up evaluation of S-attributed definitions

- **Synthesized Attributes on the Parser Stack**
  - In the scheme, parser's stack stores grammar symbols and attribute values.
  - For every production A -> XYZ with semantic rule A.a := f(X.x, Y.y, Z.z), before XYZ is reduced to A, we should already have X.x Y.y and Z.z on the stack.



state	val
...	...
X	X.x
Y	Y.y
Z	Z.z
...	...

It's time to reduce XYZ to A and calculate A.a

Bottom-up evaluation of S-attributed definitions

- **Synthesized Attributes on the Parser Stack**
  - If attribute values are placed on the stack as described, it is now easy to implement the semantic rules for the desk calculator.
  - Example – Desk Calculator
    - At each shift of **digit**, we also push **digit.lexval** into *val-stack*.
    - At all other shifts, we do not put anything into *val-stack* because other terminals do not have attributes (but we increment the stack pointer for *val-stack*).

Production	Semantic Rule	Code
L -> E **newline**	print(E.val)	print val[top-1]
E -> E1 + T	E.val := E1.val + T.val	val[newtop] = val[top-2]+val[top]
E -> T	E.val := T.val	/*newtop==top, so nothing to do*/
T -> T1 * F	T.val := T1.val x F.val	val[newtop] = val[top-2]+val[top]
T -> F	T.val := F.val	/*newtop==top, so nothing to do*/
F -> (E)	F.val := E.val	val[newtop] = val[top-1]
F -> **number**	F.val := number.lexval	/*newtop==top, so nothing to do*/

Bottom-up evaluation of S-attributed definitions

- **Synthesized Attributes on the Parser Stack**
  - Example - For input 3 * 5 + 4 newline, assume when a terminal's attribute is shifted when it is.

Input	States	Values	Action
3*5+4n			shift
*5+4n	F	3	reduce F->number
*5+4n	T	3	reduce T->F
5+4n	T*	3 _	shift
+4n	T *5	3 _5	shift
+4n	T*F	3_5	reduce F->number
+4n	T	15	reduce T->T*F
+4n	E	15	reduce E->T
4n	E+	15_	shift
n	E+4	15_4	shift
n	E+F	15_4	reduce F->number
n	E+T	15_4	reduce T->F
n	E	19	reduce E->E+T
	En	19_	shift
	L	19	reduce E->En

L-Attributed Definition

- S-Attributed Definitions can be efficiently implemented.
- We are looking for a larger (larger than S-Attributed Definitions) subset of syntax-directed definitions which can be efficiently evaluated.
- L-Attributed Definitions can always be evaluated by the depth first visit of the parse tree.
- This means that they can also be evaluated during the parsing.

L-Attributed Definition

- **Depth First Traversal**

```
algorithm dfvisit( node n ) {
    for each child m of n, in left-to-right order, do {
        evaluate the inherited attributes of m
        dfvisit( m )
    }
    evaluate the synthesized attributes of n
}
```

L-Attributed Definition

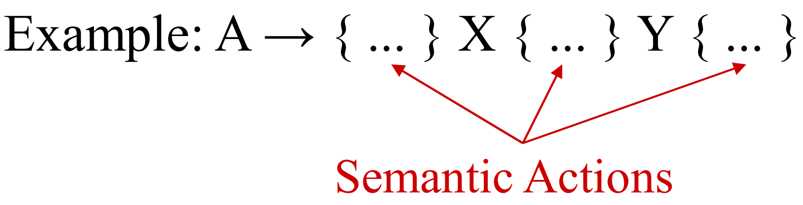
- **L-attributed Definition**
  - A syntax-directed definition is L-attributed if each inherited attribute of X_j, where $1 \leq j \leq n$, on the right side of $A \rightarrow X_1X_2 \ldots X_n$ depends only on:
    - ✓ The attributes of the symbols $X_1, \ldots, X_{j-1}$ to the left of X_j in the production and
    - ✓ the inherited attribute of A
  - Every S-attributed definition is L-attributed, the restrictions only apply to the inherited attributes (not to synthesized attributes).

L-Attributed Definition

Production	Semantic Rules
A -> L M	L.i := l(A.i)
	M.i := m(L.s)
	A.s := f(M.s)
A -> Q R	R.i := r(A.i)
	Q.i := q(R.s)
	A.s := f(Q.s)

- This syntax-directed definition is not L-attributed because the semantic rule Q.in=q(R.s) violates the restrictions of L-attributed definitions.
- When Q.in must be evaluated before we enter to Q because it is an inherited attribute.
- But the value of Q.in depends on R.s which will be available after we return from R. So, we are not be able to evaluate the value of Q.in before we enter to Q.

L-Attributed Definition

- **Translation Scheme**
  - In a syntax-directed definition, we do not say anything about the evaluation times of the semantic rules (when the semantic rules associated with a production should be evaluated?).
  - A **translation scheme** is a context-free grammar in which:
    - ✓attributes are associated with the grammar symbols and
    - ✓semantic actions enclosed between braces {} are inserted within the right sides of productions.
  - 

    Example: A → { ... } X { ... } Y { ... }

    Semantic Actions

L-Attributed Definition

- **Translation Scheme**
  - When designing a translation scheme, some restrictions should be observed to ensure that an attribute value is available when a semantic action refers to that attribute.
  - These restrictions (motivated by L-attributed definitions) ensure that a semantic action does not refer to an attribute that has not yet computed.
  - 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.

L-Attributed Definition

- **Translation Scheme**
  - If our syntax-directed definition is S-attributed, the construction of the corresponding translation scheme will be simple.
  - Each associated semantic rule in a S-attributed syntax-directed definition will be inserted as a semantic action into the end of the right side of the associated production.

Production	Semantic Rule
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$ ➔ a production of a syntax directed definition

$$\Downarrow$$

$E \rightarrow E_1 + T \; \{ E.val = E_1.val + T.val \}$ ➔ the production of the corresponding translation scheme

L-Attributed Definition

- **Translation Scheme**
  - Example - A simple translation scheme that converts infix expressions to the corresponding postfix expressions.

E → T R

R → + T { print("+") } R_1

R → ε

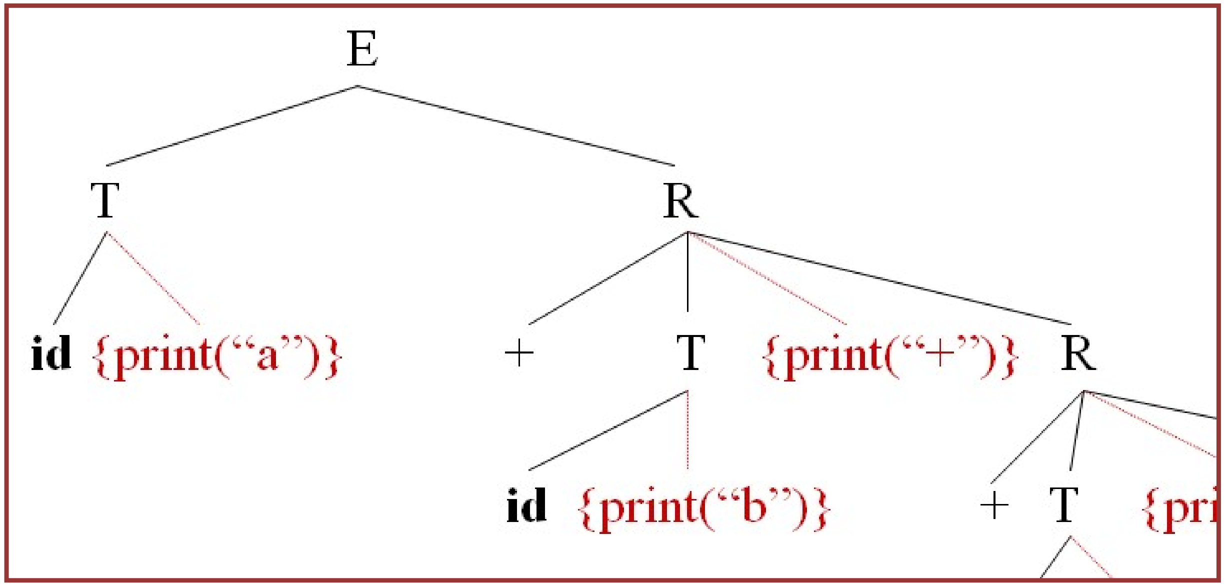
T → **id** { print(**id**.name) }

a+b+c ➔ ab+c+

infix expression postfix expression

L-Attributed Definition

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



L-Attributed Definition

- **Translation Scheme**
  - 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).

L-Attributed Definition

- **Translation Scheme**
  - With a L-attributed syntax-directed definition, it is always possible to construct a corresponding translation scheme which satisfies these three conditions (This may not be possible for a general syntax-directed translation).
  - **Example** –

    This translation scheme does NOT follow the rules:

    $S \text{ -> } A_1\ A_2 \qquad \{\ A_1.\text{in} = 1;\ A_2.\text{in} = 2\ \}$

    $A \text{ -> } a \qquad \{\ \text{print}(\ A.\text{in}\ )\ \}$

    If we traverse the parse tree depth first, A1.in has not been set when referred to in the action print(A.in)

    $S \text{ -> } \{\ A_1.\text{in} = 1\ \}\ A1\ \{\ A_2.\text{in} = 2\ \}\ A_2$

    $A \text{ -> } a\ \{\ \text{print}(\ A.\text{in}\ )\ \}$

Top-down Translation

- We will look at the implementation of L-attributed definitions during predictive parsing.
- Instead of the syntax-directed translations, we will work with translation schemes.
- We will see how to evaluate inherited attributes (in L-attributed definitions) during recursive predictive parsing.
- We will also look at what happens to attributes during the left-recursion elimination in the left-recursive grammars.

Top-down Translation

- This is a translation scheme for an L-attributed definitions.

D → T **id** { addtype(**id**.entry,T.type), L.in = T.type } L
T → **int** { T.type = integer }
T → **real** { T.type = real }
L → **id** { addtype(id.entry,L.in), L1.in = L.in } L_1
L → ε

Top-down Translation

- **Eliminating left recursion from a Translation Scheme**
  - A translation scheme with a left recursive grammar.

$E \rightarrow E_1 + T$	$\{ E.val = E_1.val + T.val \}$
$E \rightarrow E_1 - T$	$\{ E.val = E_1.val - T.val \}$
$E \rightarrow T$	$\{ E.val = T.val \}$
$T \rightarrow T_1 * F$	$\{ T.val = T_1.val * F.val \}$
$T \rightarrow F$	$\{ T.val = F.val \}$
$F \rightarrow (E)$	$\{ F.val = E.val \}$
$F \rightarrow$ **digit**	$\{ F.val =$ **digit**$.lexval \}$

  - 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 Translation

- **Eliminating left recursion from a Translation Scheme**

Inherited attribute **Synthesized attribute**

E → T { A.in=T.val } A { E.val=A.syn }

A → + T { A_1.in=A.in+T.val } A_1 { A.syn = A_1.syn}

A → - T { A_1.in=A.in-T.val } A_1 { A.syn = A_1.syn}

A → ε { A.syn = A.in }

T → F { B.in=F.val } B { T.val=B.syn }

B → * F { B_1.in=B.in*F.val } B_1 { B.syn = B_1.syn}

B → ε { B.syn = B.in }

F → (E) { F.val = E.val }

F → **digit** { F.val = **digit**.lexval }

Top-down Translation

- **Eliminating left recursion from a Translation Scheme**

$A \rightarrow A_1\ Y$ { $A.a = g(A_1.a, Y.y)$ } a left recursive grammar with

$A \rightarrow X$ { $A.a = f(X.x)$ } synthesized attributes (a,y,x).

⇓ eliminate left recursion

inherited attribute of the new non-terminal

synthesized attribute of the new non-terminal

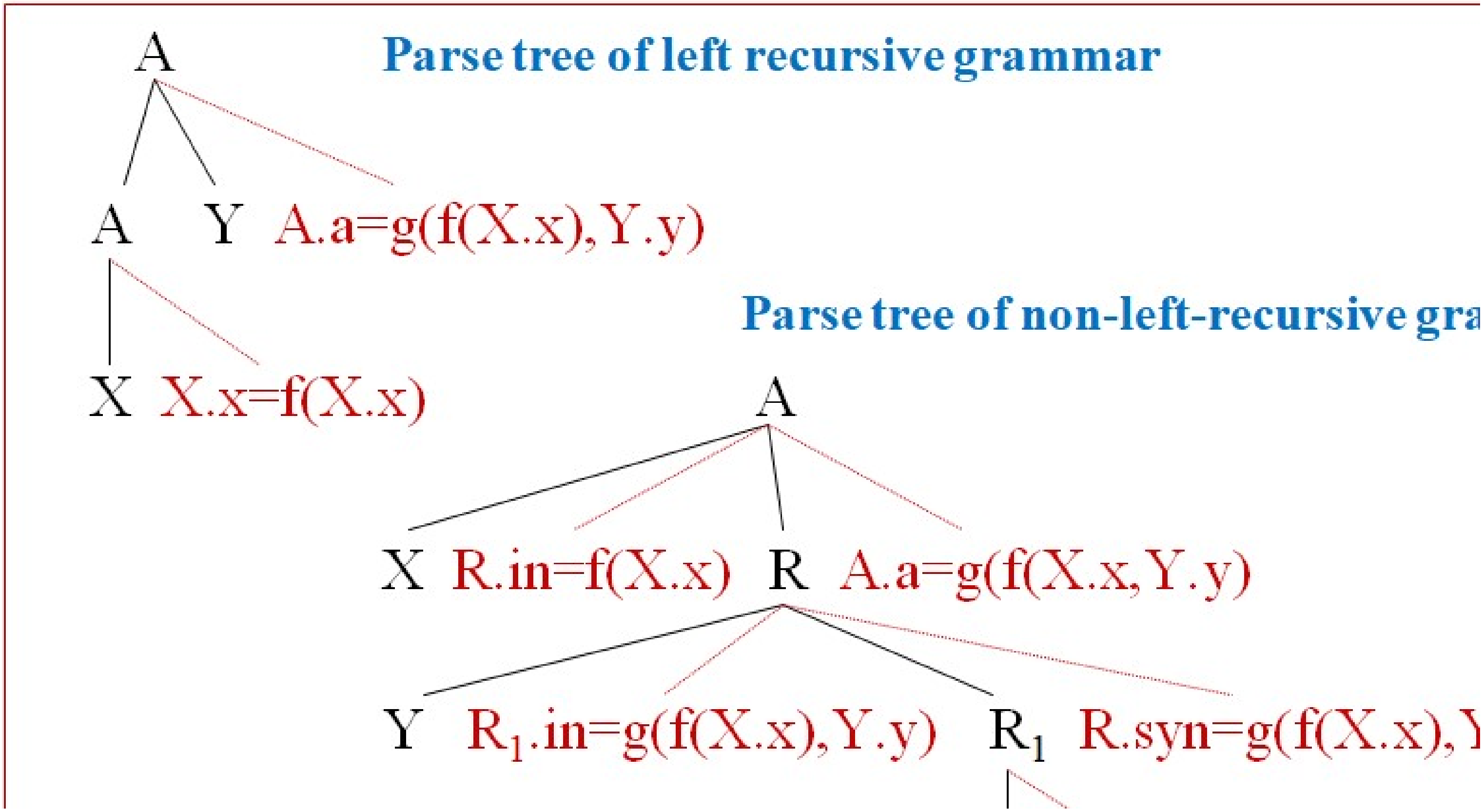
$A \rightarrow X$ { $R.in = f(X.x)$ } R { $A.a = R.syn$ }

$R \rightarrow Y$ { $R_1.in = g(R.in, Y.y)$ } R_1 { $R.syn = R_1.syn$ }

$R \rightarrow \varepsilon$ { $R.syn = R.in$ }

Top-down Translation

- **Eliminating left recursion from a Translation Scheme**



Bottom-up Evaluation of Inherited Attributes

- Using a top-down translation scheme, we can implement any L-attributed definition based on a LL(1) grammar.
- Using a bottom-up translation scheme, we can also implement any L-attributed definition based on a LL(1) grammar (each LL(1) grammar is also an LR(1) grammar).
- In addition to the L-attributed definitions based on LL(1) grammars, we can implement some of L-attributed definitions based on LR(1) grammars (not all of them) using the bottom-up translation scheme.

Bottom-up Evaluation of Inherited Attributes

- **Removing embedding actions from Translation Scheme**
  - In bottom-up evaluation scheme, the semantic actions are evaluated during the reductions.
  - During the bottom-up evaluation of S-attributed definitions, we have a parallel stack to hold synthesized attributes.

Bottom-up Evaluation of Inherited Attributes

- **Removing embedding actions from Translation Scheme**
  - **Problem**: where are we going to hold inherited attributes?
  - **A Solution**:
    - ✓ We will convert our grammar to an equivalent grammar to guarantee to the followings.
    - ✓ All embedding semantic actions in our translation scheme will be moved into the end of the production rules.
    - ✓ All inherited attributes will be copied into the synthesized attributes (most of the time synthesized attributes of new non-terminals).
    - ✓ Thus we will be evaluate all semantic actions during reductions, and we find a place to store an inherited attribute.

Bottom-up Evaluation of Inherited Attributes

- **Removing embedding actions from Translation Scheme**
  - To transform our translation scheme into an equivalent translation scheme:
    1. Remove an embedding semantic action S_i, put new a non-terminal M_i instead of that semantic action.
    2. Put that semantic action S_i into the end of a new production rule $M_i \rightarrow \varepsilon$ for that non-terminal M_i.
    3. That semantic action S_i will be evaluated when this new production rule is reduced.
    4. The evaluation order of the semantic rules are not changed by this transformation.

Bottom-up Evaluation of Inherited Attributes

- **Removing embedding actions from Translation Scheme**

$A\rightarrow \{S_1\}\ X_1\ \{S_2\}\ X_2 \ldots \{S_n\}\ X_n$

$\Downarrow$ remove embedding semantic actions

$A\rightarrow M_1\ X_1\ M_2\ X_2 \ldots M_n\ X_n$

$M_1\rightarrow\varepsilon\ \{S_1\}$

$M_2\rightarrow\varepsilon\ \{S_2\}$

.

.

$M_n\rightarrow\varepsilon\ \{S_n\}$

Bottom-up Evaluation of Inherited Attributes

- **Removing embedding actions from Translation Scheme**
  - Example

    $E \rightarrow T\ R$
    $R \rightarrow +\ T\ \{\ print(“+”)\ \}\ R_1$
    $R \rightarrow \varepsilon$
    $T \rightarrow \textbf{id}\ \{\ print(\textbf{id}.name)\ \}$

    $\Downarrow$ remove embedding semantic actions

    $E \rightarrow T\ R$
    $R \rightarrow +\ T\ M\ R_1$
    $R \rightarrow \varepsilon$
    $T \rightarrow \textbf{id}\ \{\ print(\textbf{id}.name)\ \}$
    $M \rightarrow \varepsilon\ \{\ print(“+”)\ \}$

Bottom-up Evaluation of Inherited Attributes

- **Inheriting attribute on parser stacks**
  - Bottom up parser reduces rhs of A → XY by removing XY from stack and putting A on the stack
  - Synthesized attributes of Xs can be inherited by Y by using the copy rule Y.i=X.s

D → T L L.in = T.type
T → real T.type = real
T → int T.type = int
L → L1 , id L1 .in = L.in; addtype(id.entry, L.in)
L → id addtype (id.entry,L.in)

⇨

D → T {L.in = T.type} L
T → int {T.type = integer}
T → real {T.type = real}
L → {L1 .in =L.in} L1 ,id {addtype(id.entry,Lin)}
L → id {addtype(id.entry,Lin)}

Bottom-up Evaluation of Inherited Attributes

- **Inheriting attribute on parser stacks**
  - Example: take string real p,q,r

State stack	INPUT	PRODUCTION
	real p,q,r	
real	p,q,r	
T	p,q,r	T → real
Tp	,q,r	
TL	,q,r	L → id
TL,	q,r	
TL,q	,r	
TL	,r	L → L,id
TL,	r	
TL,r	-	
TL	-	L → L,id
D	-	D →TL

Bottom-up Evaluation of Inherited Attributes

- **Inheriting attribute on parser stacks**
  - Every time a string is reduced to L, T.val is just below it on the stack
  - Every time a reduction to L is made value of T type is just below it
  - Use the fact that T.val (type information) is at a known place in the stack
  - When production L → id is applied, id.entry is at the top of the stack and T.type is just below it, therefore,

    addtype(id.entry,L.in) → addtype(val[top], val[top-1])
  - Similarly when production L → L1 , id is applied id.entry is at the top of the stack and T.type is three places below it, therefore,

    addtype(id.entry, L.in) → addtype(val[top],val[top-3])

Bottom-up Evaluation of Inherited Attributes

- **Inheriting attribute on parser stacks**
  - Therefore, the translation scheme becomes

D → T L	
T → int	val[top] =integer
T → real	val[top] =real
L → L,id	addtype(val[top], val[top-3])
L → id	addtype(val[top], val[top-1])

References

- Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullmann “Compilers- Principles, Techniques and Tools”, Pearson Education.

Thank You !!!