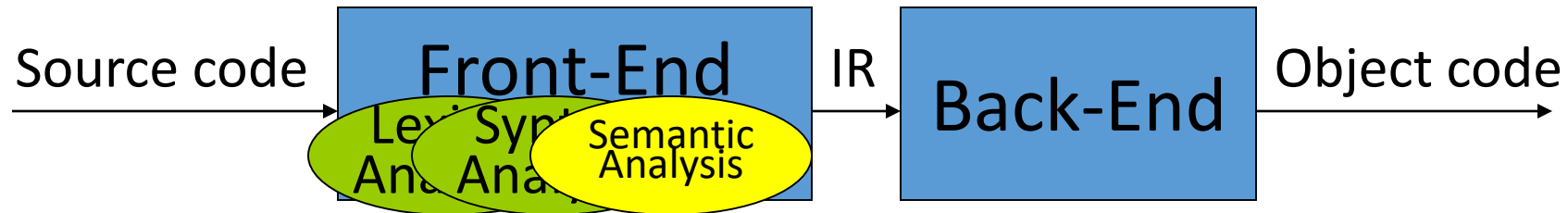


Semantic Analysis with Attribute Grammars

Semantic Analysis with Attribute Grammars



(from last lectures) :

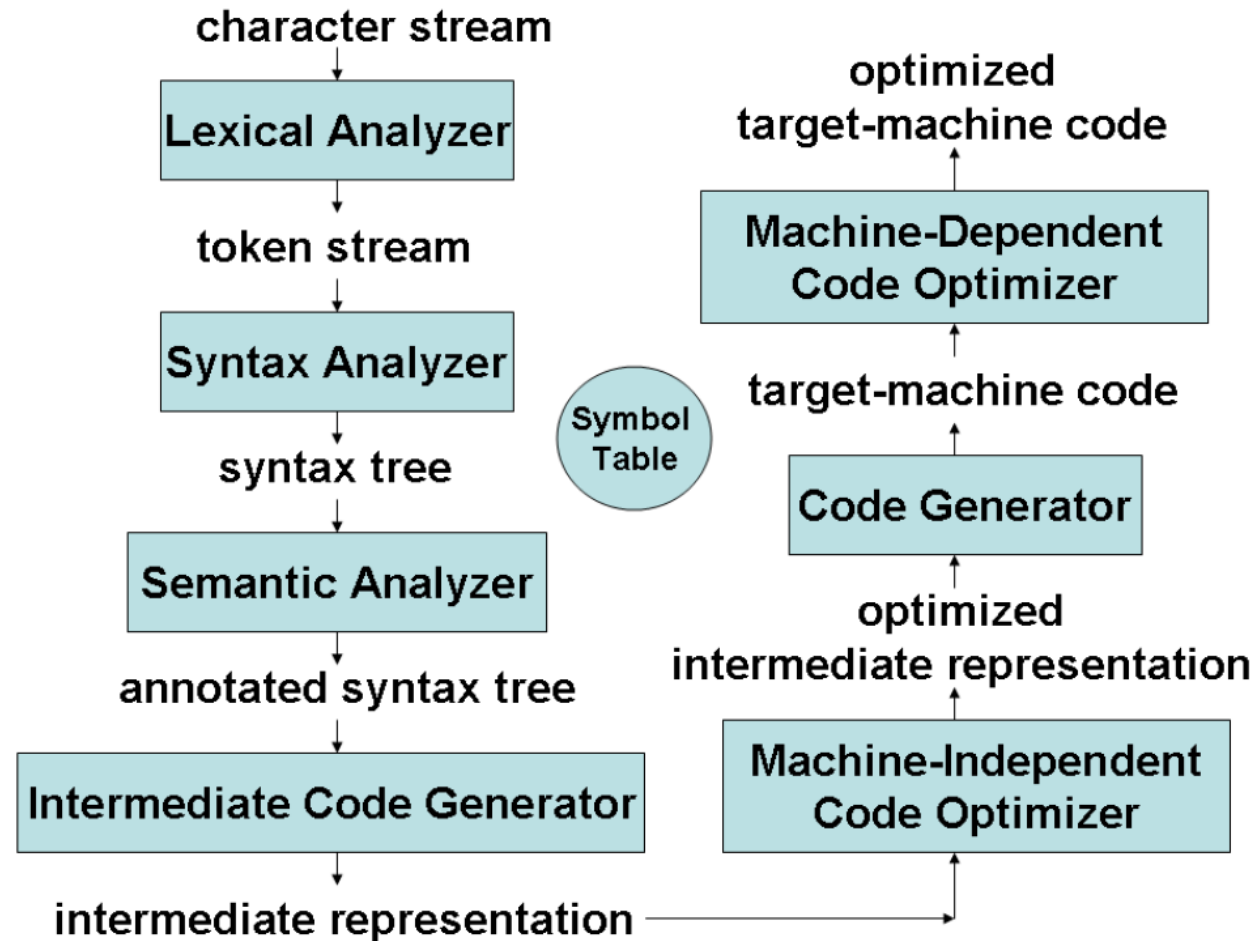
- After Lexical & Syntax analysis we have determined that the input program is a valid sentence in the source language.
- The outcome from **syntax analysis** is a **parse tree**: a graphical representation of how the start symbol of a grammar derives a string in the language.
- *But, have we finished with analysis & **detection of all possible errors?***

Semantic analysis with attribute grammars

Outline

- Introduction
- Attribute grammars
- Attributed translation grammars
- Semantic analysis with attributed translation grammars

Overview- Compiler



What is wrong with the following?

```
void foo(int a, int b, int c)
{ ... }

lala()
{
    int f[3], g[2], h, i, j, k, l;
    char *p;
    l=foo(h, i, "ab", j, k);
    k=f*i+j;
    h=g[7];
    printf("%s, %s", p, q);
    p=10;
}
```

Any errors!

All beyond
syntax!

These are not
issues for the
context-free
grammar!

To generate code, we need to understand its meaning!

Beyond syntax: context sensitive questions

To generate code, the compiler needs to answer questions such as:

- Is **x** a var, an array or a function? Is **x** declared?
- Are there names that are not declared? Declared but not used?
- Which declaration of **x** does each use reference?
- Is the expression **x-2*y** type-consistent? (**type checking**: the compiler needs to assign a type to each expression it calculates)
- In **a[i,j,k]**, is **a** declared to have 3 dimensions?
- How many arguments does **foo()** take?

These are beyond a context-free grammar!

Semantic Analysis

- Semantic consistency that cannot be handled at the parsing stage is handled here
- These are **static semantics** of programming languages and can be checked by the *semantic analyzer*
 - Variables are declared before use
 - Types match on both sides of assignments
 - Parameter types and number match in **declaration** and **use**
- Compilers can only generate code to check **dynamic semantics** of programming languages **at runtime**
 - whether an overflow will occur during an arithmetic operation
 - whether array limits will be crossed during execution
 - whether recursion will cross stack limits
 - whether heap memory will be insufficient

Static Semantics

```
int dot_prod(int x[], int y[]){
    int d, i; d = 0;
    for (i=0; i<10; i++) d += x[i]*y[i];
    return d;
}
main(){
    int p; int a[10], b[10];
    p = dot_prod(a,b);
}
```

Samples of static semantic checks in *main*

- Types of **p** and return type of **dot_prod** match
- *Number* and *type* of the parameters of **dot_prod** are the same in both its *declaration* and *use*
- **p** is declared before use, same for **a** and **b**

Static Semantics: Errors given by gcc Compiler ??

```
int dot_product(int a[], int b[]) {...}
```

```
1 main(){int a[10]={1,2,3,4,5,6,7,8,9,10};  
2 int b[10]={1,2,3,4,5,6,7,8,9,10};  
3 printf("%d", dot_product(b));  
4 printf("%d", dot_product(a,b,a));  
5 int p[10]; p=dotproduct(a,b); printf("%d",p);}
```

Static Semantics: Errors given by gcc Compiler

```
int dot_product(int a[], int b[]) {...}
```

```
1 main(){int a[10]={1,2,3,4,5,6,7,8,9,10};  
2 int b[10]={1,2,3,4,5,6,7,8,9,10};  
3 printf("%d", dot_product(b));  
4 printf("%d", dot_product(a,b,a));  
5 int p[10]; p=dotproduct(a,b); printf("%d",p);}
```

Errors given by gcc Compiler

In function 'main':

error in 3: too few arguments to fn 'dot_product'

error in 4: too many arguments to fn 'dot_product'

error in 5: incompatible types in assignment

warning in 5: format '%d' expects type 'int', but
argument 2 has type 'int *'

Static Semantics

```
int dot_prod(int x[], int y[]){
    int d, i; d = 0;
    for (i=0; i<10; i++) d += x[i]*y[i];
    return d;
}
main() {
    int p; int a[10], b[10];
    p = dot_prod(a,b);
}
```

Samples of static semantic checks in **dot_prod** ??

Static Semantics

```
int dot_prod(int x[], int y[]){
    int d, i; d = 0;
    for (i=0; i<10; i++) d += x[i]*y[i];
    return d;
}
main() {
    int p; int a[10], b[10];
    p = dot_prod(a,b);
}
```

Samples of static semantic checks in **dot_prod**

- **d** and **i** are declared before use
- Type of **d** matches the return type of **dot_prod**
- Type of **d** matches the result type of “*”
- Elements of arrays **x** and **y** are compatible with “*”

Dynamic Semantics

```
int dot_prod(int x[], int y[]){
    int d, i; d = 0;
    for (i=0; i<10; i++) d += x[i]*y[i];
    return d;
}
main(){
    int p; int a[10], b[10];
    p = dot_prod(a,b);
}
```

Samples of dynamic semantic checks in **dot_prod**

- Value of **i** does not exceed the declared range of arrays **x** and **y** (both lower and upper)
- There are no overflows during the operations of “*” and “+” in **d += x[i]* y[i]**

Dynamic Semantics

```
int fact(int n) {  
    if (n==0) return 1;  
    else return (n*fact(n-1));  
}  
main(){int p; p = fact(10); }
```

Samples of dynamic semantic checks in *fact* ??

Dynamic Semantics

```
int fact(int n) {  
    if (n==0) return 1;  
    else return (n*fact(n-1));  
}  
main(){int p; p = fact(10); }
```

Samples of dynamic semantic checks in *fact*

- Program stack does not overflow due to **recursion**
- There is no overflow due to “*” in **n*fact(n-1)**

Semantic Analysis

- If declarations need not appear before use (as in C++), semantic analysis needs more than one pass
- **Static semantics** of PL can be specified using **attribute grammars**
- Semantic analyzers can be generated *semi-automatically* from attribute grammars
- **Attribute grammars** are extensions of ***context-free grammars***

Attribute Grammars

- **Idea:**

- Annotate each grammar symbol with a set of **attributes**.
- Associate **semantic rules** with production rules that defines the value of each attribute in terms of other attributes.

- **Attribute Grammars:**

- A context-free grammar **augmented with a set of (semantic) rules**.
- Each symbol has a set of attributes.
- The rules specify how to compute a value for each attribute

Attribute Grammars

- Let $G = (N, T, P, S)$ be a CFG and let $V = N \cup T$.
- Every symbol X of V has associated with it a set of attributes (denoted by $X.a$, $X.b$, etc.)
- Each attribute takes values from a specified domain (finite or infinite), which is its type
 - Typical domains of attributes are, *integers, reals, characters, strings, booleans, structures*, etc.
- Two types of attributes: **inherited** (denoted by $AI(X)$) and **synthesized** (denoted by $AS(X)$)

Synthesized and inherited attributes

- An attribute cannot be both **synthesized** and **inherited**, but a symbol can have both types of attributes
- Attributes of symbols are evaluated over a parse tree by making passes over the parse tree
- **Synthesized attributes** are computed in a **bottom-up** fashion from the leaves upwards
 - Always synthesized from the attribute values of the children of the node
 - **Leaf nodes** (terminals) have **synthesized attributes initialized by the lexical analyzer** and cannot be modified
 - An AG with only synthesized attributes is an **S-attributed grammar (SAG)**
 - YACC permits only SAGs
- **Inherited attributes** flow down from the parent or siblings to the node in question

Attribute Computation Rules

- A production $p \in P$ has a set of attribute computation rules (functions)
- Rules are provided for the computation of
 - Synthesized attributes of the LHS non-terminal of p
 - Inherited attributes of the RHS non-terminals of p
- These rules can use attributes of symbols from the production p only
 - Rules are strictly local to the production p (no side effects)
- Restrictions on the rules define different types of attribute grammars
 - L-attribute grammars, S-attribute grammars, ordered attribute grammars, absolutely non-circular attribute grammars, circular attribute grammars, etc.

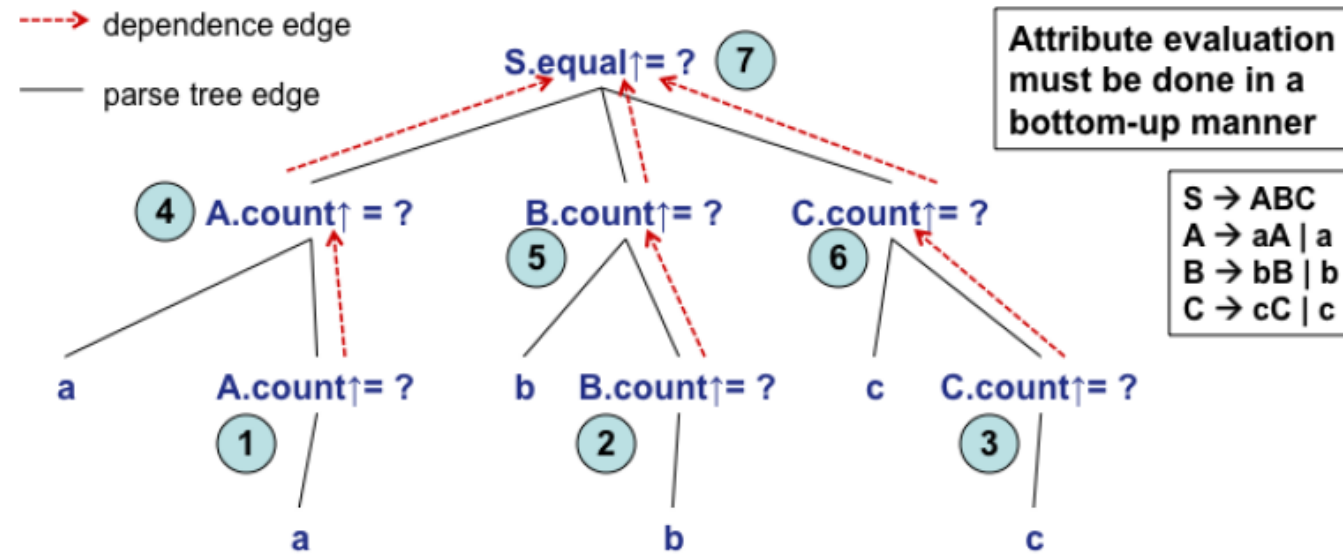
Example- Attribute Grammar

- Consider the following **CFG**

$S \rightarrow A B C, A \rightarrow aA \mid a, B \rightarrow bB \mid b, C \rightarrow cC \mid c$
generates: $L(G) = \{a^m b^n c^p \mid m, n, p \geq 1\}$

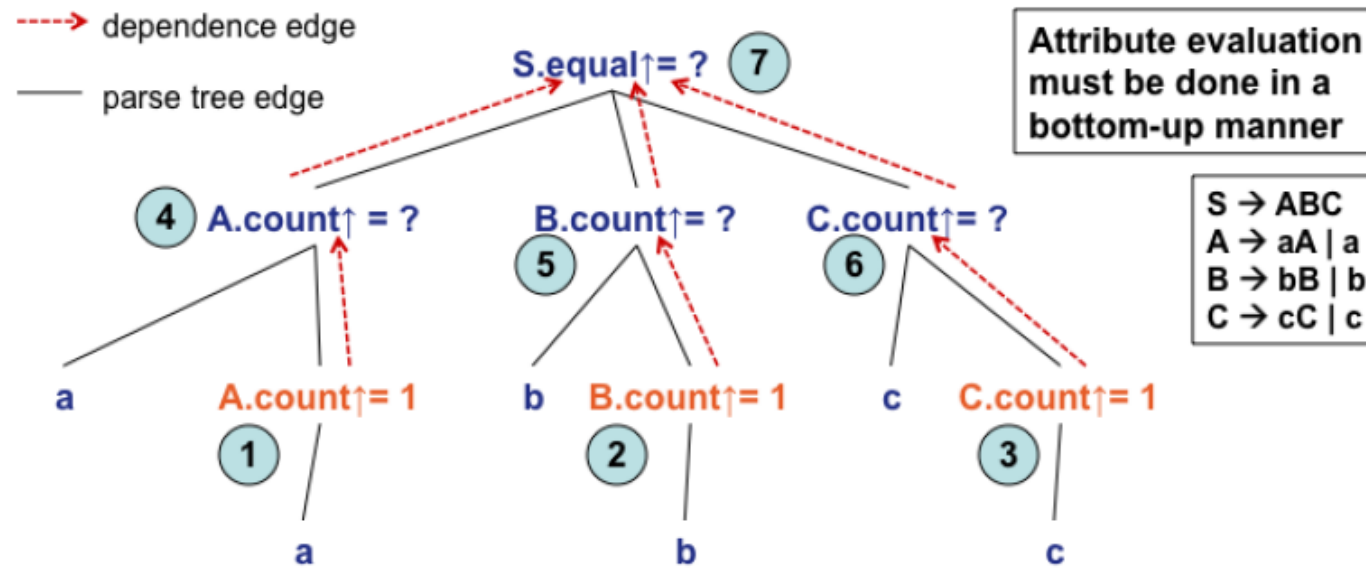
- We define an AG based on the CFG to generate $L = \{a^n b^n c^n \mid n \geq 1\}$
- All the non-terminals will have only synthesized attributes
 - $AS(S) = \{equal \uparrow: \{T, F\}\}$
 - $AS(A) = AS(B) = AS(C) = \{count \uparrow: integer\}$

Example- Attribute Grammar



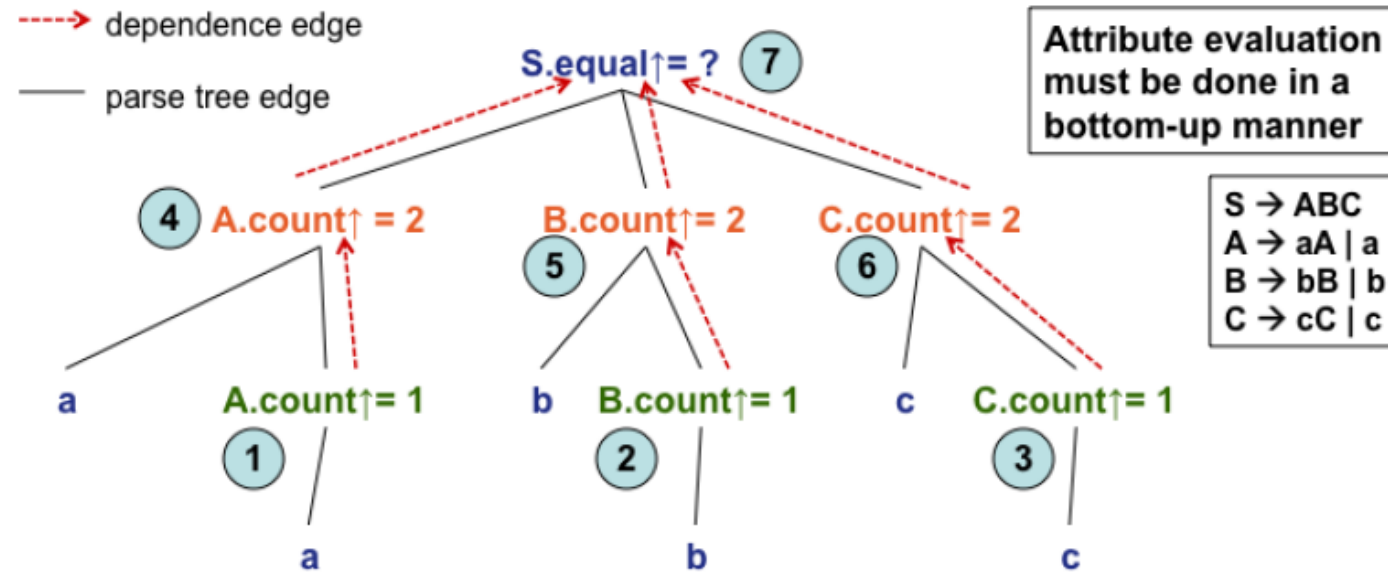
- ① $S \rightarrow ABC \{ S.equal \uparrow := \text{if } A.count \uparrow = B.count \uparrow \ \& \ B.count \uparrow = C.count \uparrow \text{ then } T \text{ else } F \}$
- ② $A_1 \rightarrow aA_2 \{ A_1.count \uparrow := A_2.count \uparrow + 1 \}$
- ③ $A \rightarrow a \{ A.count \uparrow := 1 \}$
- ④ $B_1 \rightarrow bB_2 \{ B_1.count \uparrow := B_2.count \uparrow + 1 \}$
- ⑤ $B \rightarrow b \{ B.count \uparrow := 1 \}$
- ⑥ $C_1 \rightarrow cC_2 \{ C_1.count \uparrow := C_2.count \uparrow + 1 \}$
- ⑦ $C \rightarrow c \{ C.count \uparrow := 1 \}$

Example- Attribute Grammar



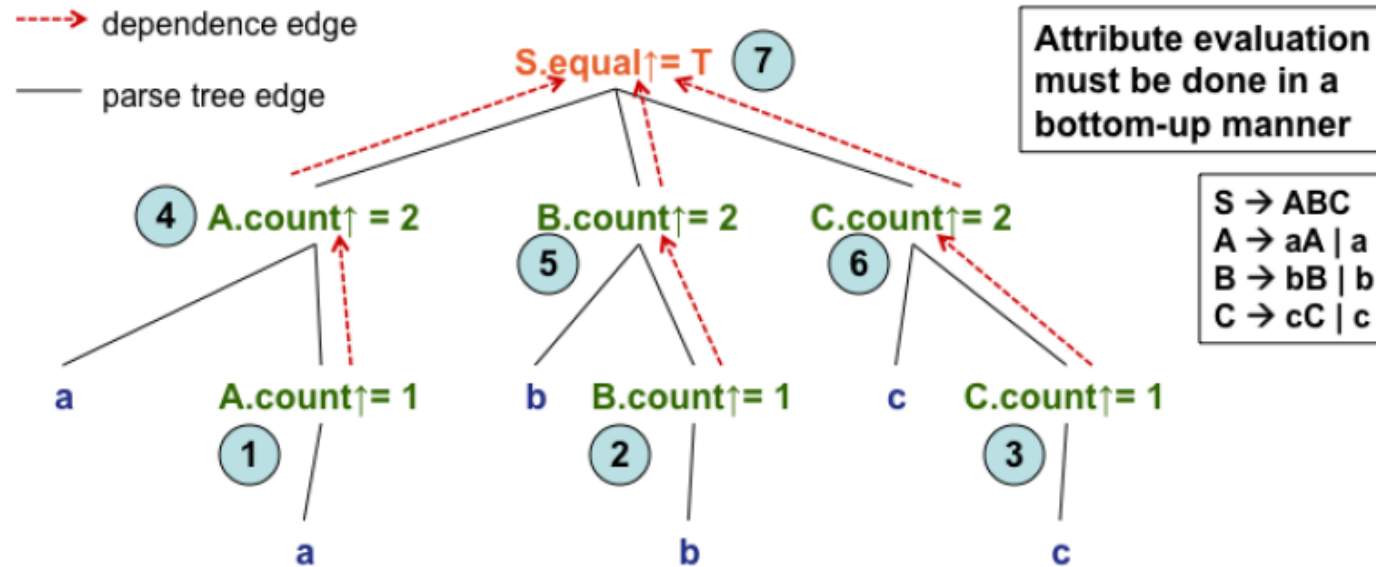
- ① $S \rightarrow ABC \{ S.equal \uparrow := \text{if } A.count \uparrow = B.count \uparrow \ \& \ B.count \uparrow = C.count \uparrow \text{ then } T \text{ else } F \}$
- ② $A_1 \rightarrow aA_2 \{ A_1.count \uparrow := A_2.count \uparrow + 1 \}$
- ③ $A \rightarrow a \{ A.count \uparrow := 1 \}$
- ④ $B_1 \rightarrow bB_2 \{ B_1.count \uparrow := B_2.count \uparrow + 1 \}$
- ⑤ $B \rightarrow b \{ B.count \uparrow := 1 \}$
- ⑥ $C_1 \rightarrow cC_2 \{ C_1.count \uparrow := C_2.count \uparrow + 1 \}$
- ⑦ $C \rightarrow c \{ C.count \uparrow := 1 \}$

Example- Attribute Grammar



- ① $S \rightarrow ABC \{ S.equal \uparrow := \text{if } A.count \uparrow = B.count \uparrow \ \& \ B.count \uparrow = C.count \uparrow \text{ then } T \text{ else } F \}$
- ② $A_1 \rightarrow aA_2 \{ A_1.count \uparrow := A_2.count \uparrow + 1 \}$
- ③ $A \rightarrow a \{ A.count \uparrow := 1 \}$
- ④ $B_1 \rightarrow bB_2 \{ B_1.count \uparrow := B_2.count \uparrow + 1 \}$
- ⑤ $B \rightarrow b \{ B.count \uparrow := 1 \}$
- ⑥ $C_1 \rightarrow cC_2 \{ C_1.count \uparrow := C_2.count \uparrow + 1 \}$
- ⑦ $C \rightarrow c \{ C.count \uparrow := 1 \}$

Example- Attribute Grammar



- ① $S \rightarrow ABC \{ S.equal \uparrow := \text{if } A.count \uparrow = B.count \uparrow \ \& \ B.count \uparrow = C.count \uparrow \text{ then } T \text{ else } F \}$
- ② $A_1 \rightarrow aA_2 \{ A_1.count \uparrow := A_2.count \uparrow + 1 \}$
- ③ $A \rightarrow a \{ A.count \uparrow := 1 \}$
- ④ $B_1 \rightarrow bB_2 \{ B_1.count \uparrow := B_2.count \uparrow + 1 \}$
- ⑤ $B \rightarrow b \{ B.count \uparrow := 1 \}$
- ⑥ $C_1 \rightarrow cC_2 \{ C_1.count \uparrow := C_2.count \uparrow + 1 \}$
- ⑦ $C \rightarrow c \{ C.count \uparrow := 1 \}$

Exercise:

Considering the same attribute grammar.

Give a parse-tree for “**aaabbc**”.

Show attribute instances, and dependencies, and evaluate them.

Attribute Dependency Graph

If an attribute at one node depends on an attribute of another node then the former must be evaluated before the latter.

Attribute Dependency Graph:

- Nodes represent attributes; edges represent the flow of values.
- Graph is specific to the parse tree (can be built alongside and its size is related to the size of the parse tree).
- **Evaluation order:**
 - **Parse tree methods:** use a **topological sort** (any ordering of the nodes such that edges go only from the earlier nodes to the later nodes): cyclic graph fails!
 - **Rule-based methods:** the order is statically predetermined.

Attribute Dependency Graph

- Let T be a parse tree generated by the CFG of an AG, G .
- The attribute dependence graph (dependence graph for short) for T is the directed graph, $DG(T) = (V, E)$, where
 - $V = \{b \mid b \text{ is an *attribute instance* of some tree node}\}$, and
 - $E = \{ (b, c) \mid b, c \in V, \text{ } b \text{ and } c \text{ are attributes of grammar symbols in the same production } p, \text{ and the value of } b \text{ is used for computing the value of } c \text{ in an attribute computation rule associated with production } p \}$

Strategy for Evaluating Attributes

- Construct the **parse tree**
- Construct the **dependence graph**
- Perform ***topological sort*** on the dependence graph and obtain an evaluation order
- Evaluate attributes according to this order using the corresponding attribute evaluation rules attached to the respective productions
- Multiple attributes at a node in the parse tree may result in that node to be visited multiple number of times
 - Each visit resulting in the evaluation of at least one attribute

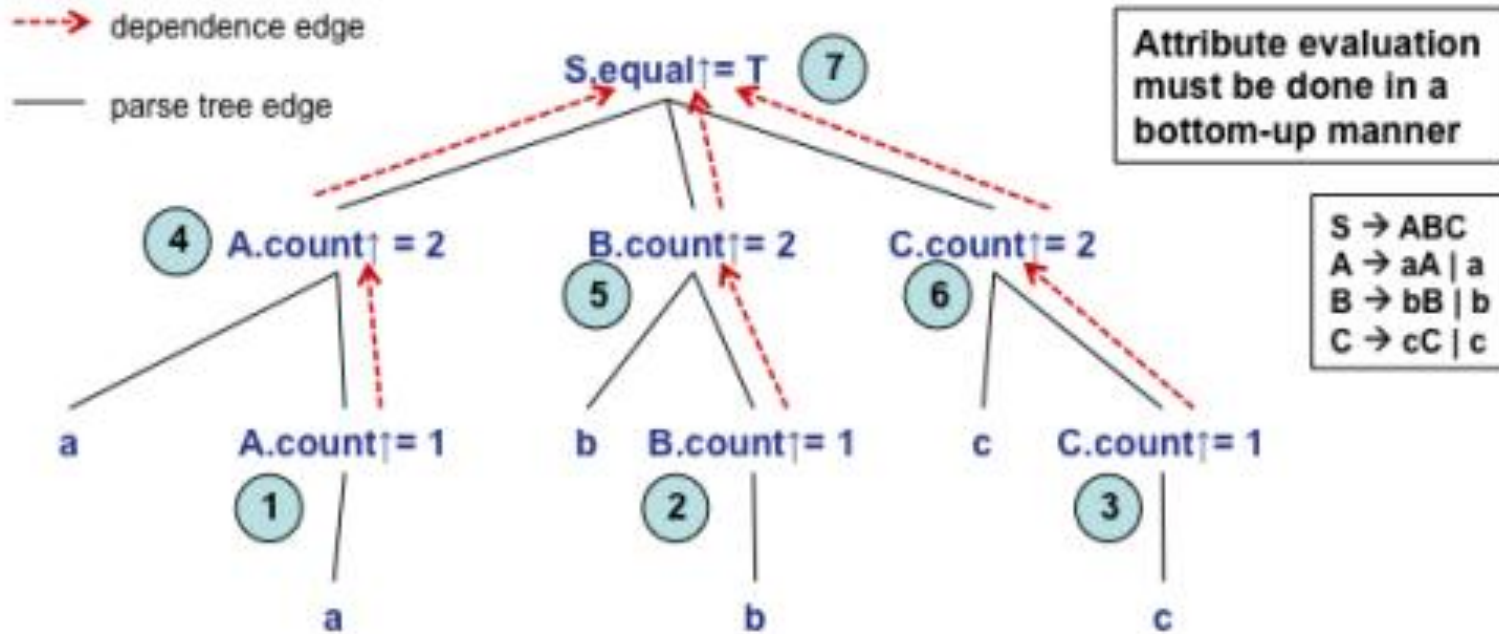
Attribute Evaluation Algorithm

Input: A parse tree T with unevaluated attribute instances

Output: T with consistent attribute values

```
{ Let  $(V, E) = DG(T)$ ;  
  Let  $W = \{b \mid b \in V \ \& \ indegree(b) = 0\}$ ;  
  while  $W \neq \phi$  do  
    { remove some  $b$  from  $W$ ;  
       $value(b) :=$  value defined by appropriate attribute  
        computation rule;  
      for all  $(b, c) \in E$  do  
        {  $indegree(c) := indegree(c) - 1$ ;  
          if  $indegree(c) = 0$  then  $W := W \cup \{c\}$ ;  
        }  
      }  
    }  
}
```

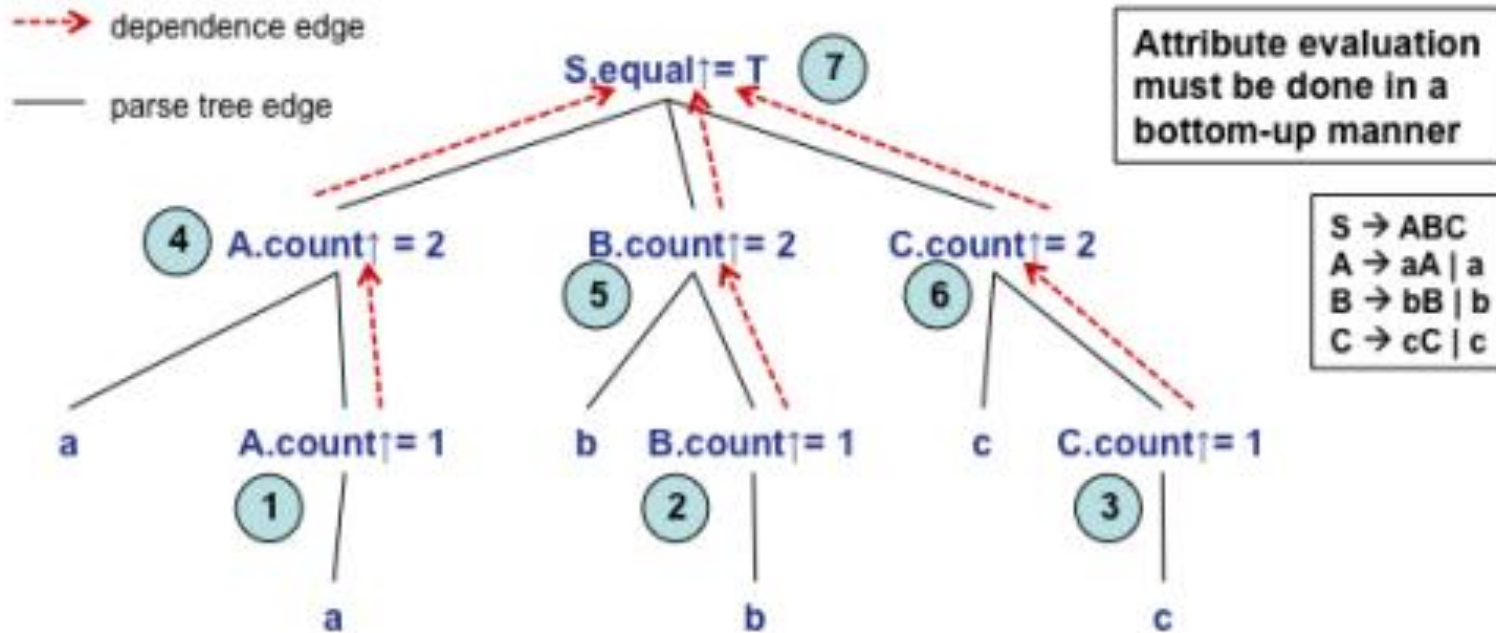
Dependency Graph –Example



Evaluation Orders?

- 1 $S \rightarrow ABC \{ S.equal \uparrow := \text{if } A.count \uparrow = B.count \uparrow \ \& \ B.count \uparrow = C.count \uparrow \text{ then } T \text{ else } F \}$
- 2 $A_1 \rightarrow aA_2 \{ A_1.count \uparrow := A_2.count \uparrow + 1 \}$
- 3 $A \rightarrow a \{ A.count \uparrow := 1 \}$
- 4 $B_1 \rightarrow bB_2 \{ B_1.count \uparrow := B_2.count \uparrow + 1 \}$
- 5 $B \rightarrow b \{ B.count \uparrow := 1 \}$
- 6 $C_1 \rightarrow cC_2 \{ C_1.count \uparrow := C_2.count \uparrow + 1 \}$
- 7 $C \rightarrow c \{ C.count \uparrow := 1 \}$

Dependency Graph –Example



Evaluation Orders?

1,2,3,4,5,6,7

2,3,6,5,1,4,7

1,4,2,5,3,6,7 (can be used with LR parsing)

Right-most derivation (reverse is LR parsing order)

1 $S \rightarrow ABC$ { $S.equal \uparrow :=$ if $A.count \uparrow = B.count \uparrow$ & $B.count \uparrow = C.count \uparrow$ then T else F }

2 $A_1 \rightarrow aA_2$ { $A_1.count \uparrow := A_2.count \uparrow + 1$ }

3 $A \rightarrow a$ { $A.count \uparrow := 1$ }

4 $B_1 \rightarrow bB_2$ { $B_1.count \uparrow := B_2.count \uparrow + 1$ }

5 $B \rightarrow b$ { $B.count \uparrow := 1$ }

6 $C_1 \rightarrow cC_2$ { $C_1.count \uparrow := C_2.count \uparrow + 1$ }

7 $C \rightarrow c$ { $C.count \uparrow := 1$ }

$S \Rightarrow ABC \Rightarrow ABcC \Rightarrow ABcc \Rightarrow AbBcc \Rightarrow Abbcc \Rightarrow aAbbcc \Rightarrow aabbcc$

1. $A.count = 1$ { $A \rightarrow a$, { $A.count := 1$ } }

4. $A.count = 2$ { $A_1 \rightarrow aA_2$, { $A_1.count := A_2.count + 1$ } }

2. $B.count = 1$ { $B \rightarrow b$, { $B.count := 1$ } }

5. $B.count = 2$ { $B_1 \rightarrow bB_2$, { $B_1.count := B_2.count + 1$ } }

3. $C.count = 1$ { $C \rightarrow c$, { $C.count := 1$ } }

6. $C.count = 2$ { $C_1 \rightarrow cC_2$, { $C_1.count := C_2.count + 1$ } }

7. $S.equal = 1$ { $S \rightarrow ABC$, { $S.equal :=$ if $A.count = B.count$ & $B.count = C.count$ then T else F } }

Example

(semantic rules to calculate the value of an expression)

$$\begin{array}{l} E \rightarrow E_1 + T \\ \quad | \quad T \\ T \rightarrow T_1 * F \\ \quad | \quad F \\ F \rightarrow \text{integer} \end{array}$$

Consider the above grammar for expressions.

(Q1) Give **attribute grammar** (set of attributes for each symbol, and semantic rules) **to be able to calculate the value of the given expression.**

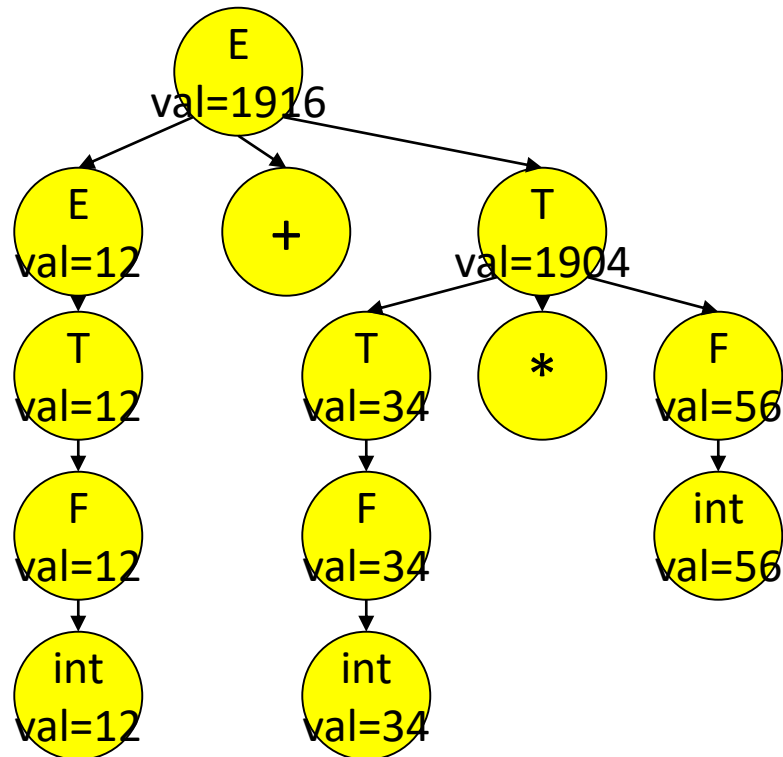
(Q2) Provide a **parse-tree** and a **dependency graph** for the word “**12+34*56**”

Example

(semantic rules to calculate the value of an expression)

$$\begin{aligned} E &\rightarrow E_1 + T \\ &\quad | \quad T \\ T &\rightarrow T_1 * F \\ &\quad | \quad F \\ F &\rightarrow \text{integer} \end{aligned}$$
$$\begin{aligned} E.\text{val} &= E_1.\text{val} + T.\text{val} \\ E.\text{val} &= T.\text{val} \\ T.\text{val} &= T_1.\text{val} \times F.\text{val} \\ T.\text{val} &= F.\text{val} \\ F.\text{val} &= \text{integer.lexval} \end{aligned}$$

12+34*56



Evaluation order:
start from the leaves
and proceed bottom-up!

Example- Attribute Grammar

- Grammar that generates binary numbers with a decimal point

$N \rightarrow L.R,$

$L \rightarrow BL \mid B,$

$R \rightarrow BR \mid B,$

$B \rightarrow 0 \mid 1$

- AG for the evaluation of a real number from its bit-string representation

Example: $110.101 = 6.625$

Example- Attribute Grammar

- Grammar that generates binary numbers with a decimal point

$$N \rightarrow L.R, \quad L \rightarrow BL \mid B, \quad R \rightarrow BR \mid B, \quad B \rightarrow 0 \mid 1$$

- AG for the evaluation of a real number from its bit-string representation

Example: 110.101 = 6.625

Attributes

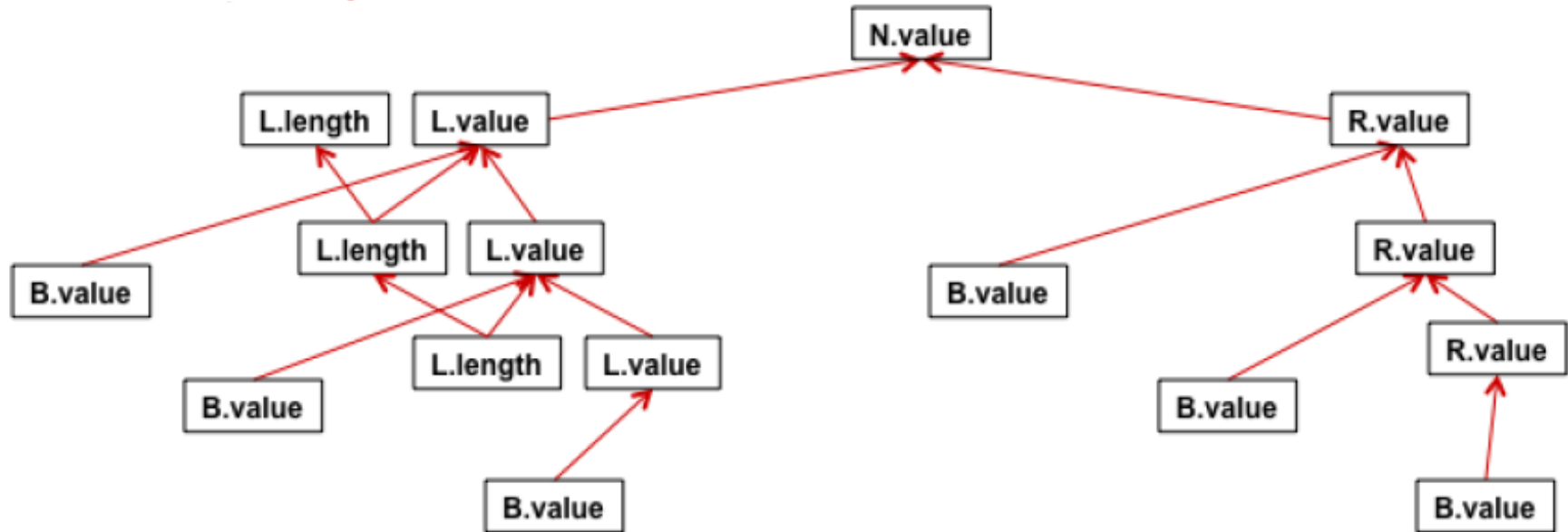
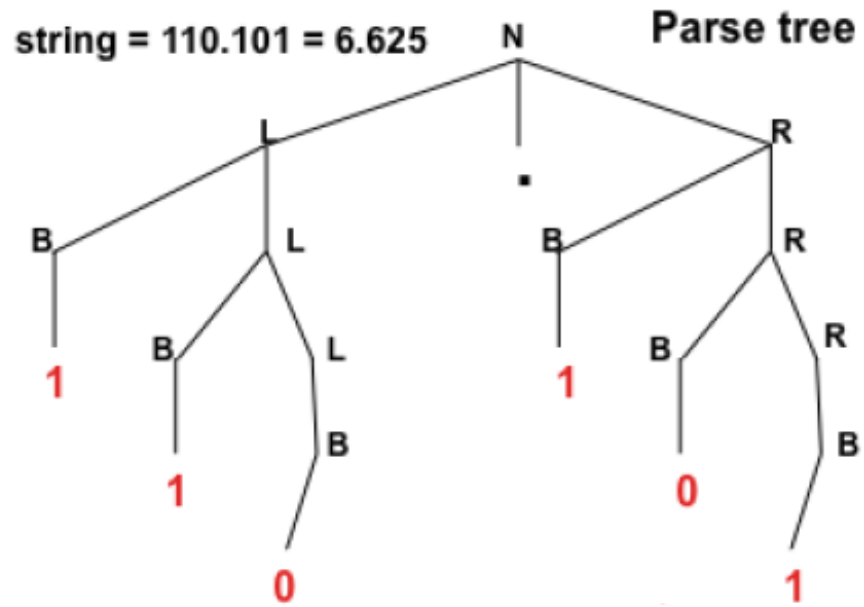
$AS(N) = AS(R) = AS(B) = \{\text{value} \uparrow: \text{real}\},$

$AS(L) = \{\text{length} \uparrow: \text{integer}, \text{value} \uparrow: \text{real}\}$

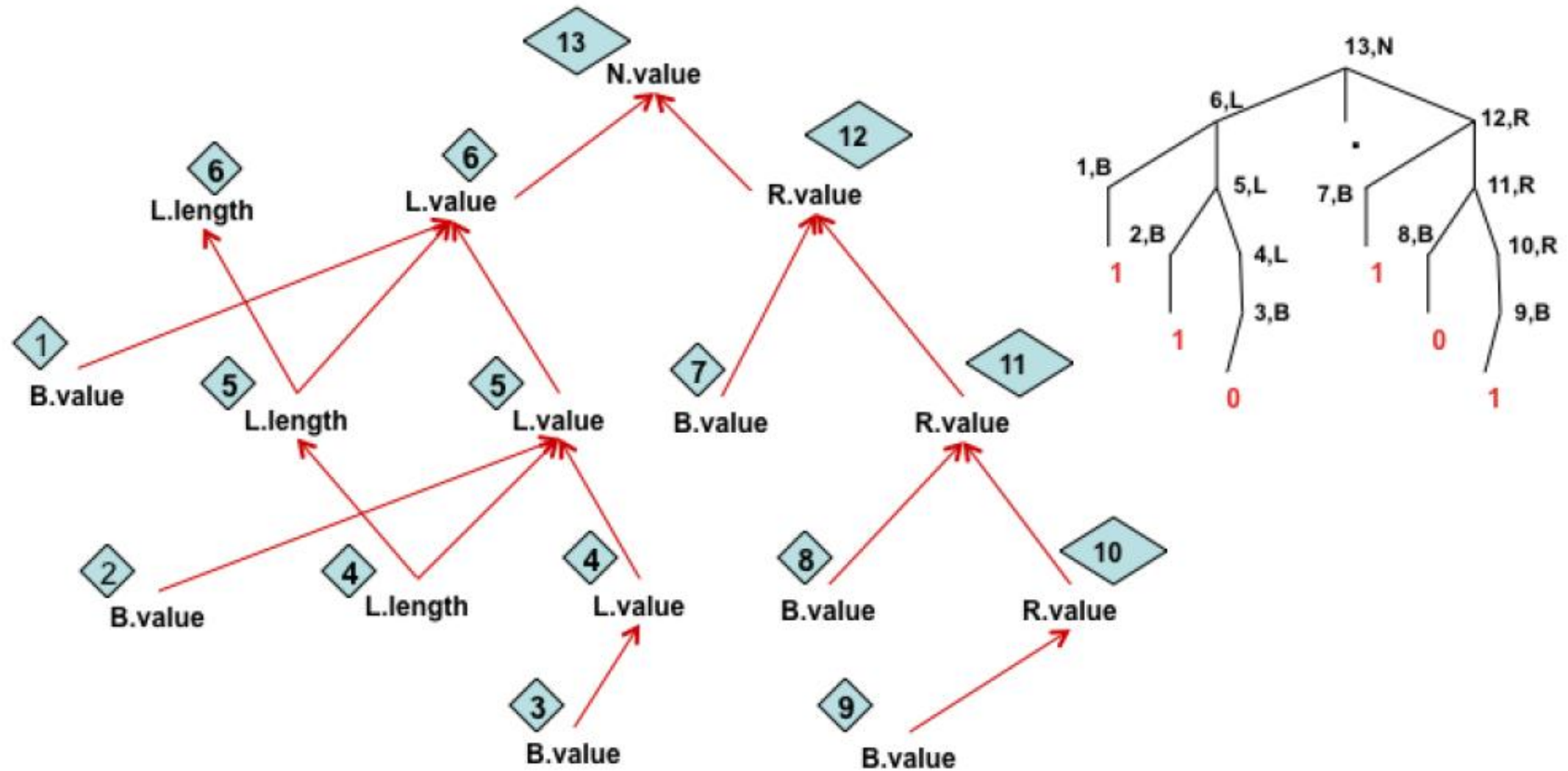
Rules

- 1 $N \rightarrow L.R \{N.value \uparrow := L.value \uparrow + R.value \uparrow\}$
- 2 $L \rightarrow B \{L.value \uparrow := B.value \uparrow; L.length \uparrow := 1\}$
- 3 $L_1 \rightarrow BL_2 \{L_1.length \uparrow := L_2.length \uparrow + 1;$
 $L_1.value \uparrow := B.value \uparrow * 2^{L_2.length \uparrow} + L_2.value \uparrow\}$
- 4 $R \rightarrow B \{R.value \uparrow := B.value \uparrow / 2\}$
- 5 $R_1 \rightarrow BR_2 \{R_1.value \uparrow := (B.value \uparrow + R_2.value \uparrow) / 2\}$
- 6 $B \rightarrow 0 \{B.value \uparrow := 0\}$
- 7 $B \rightarrow 1 \{B.value \uparrow := 1\}$

Dependency graph for prev. example



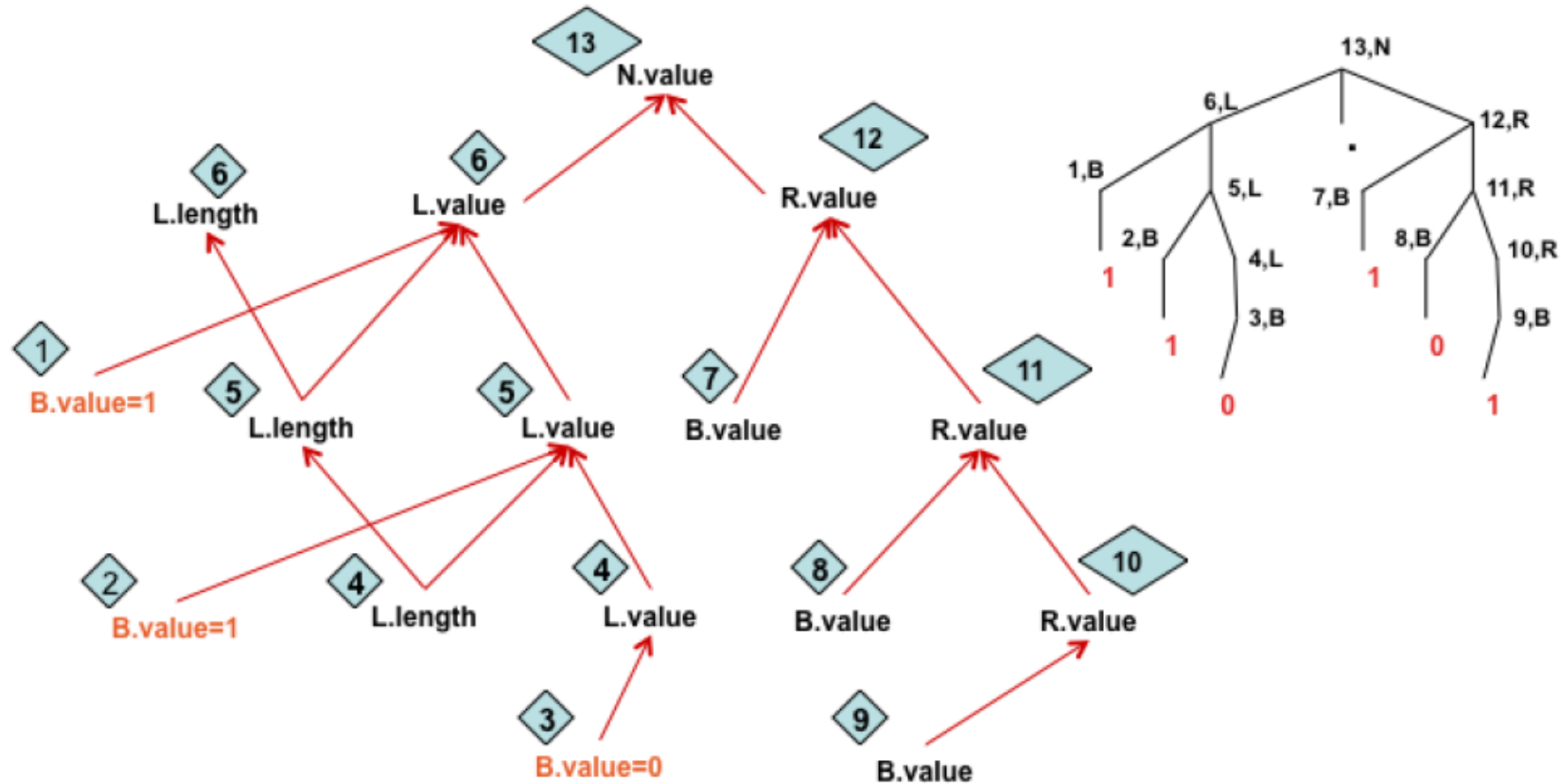
Attribute Evaluation -1



Attribute Evaluation - 2

Nodes 1,2: $B \rightarrow 1$ $\{B.value \uparrow := 1\}$

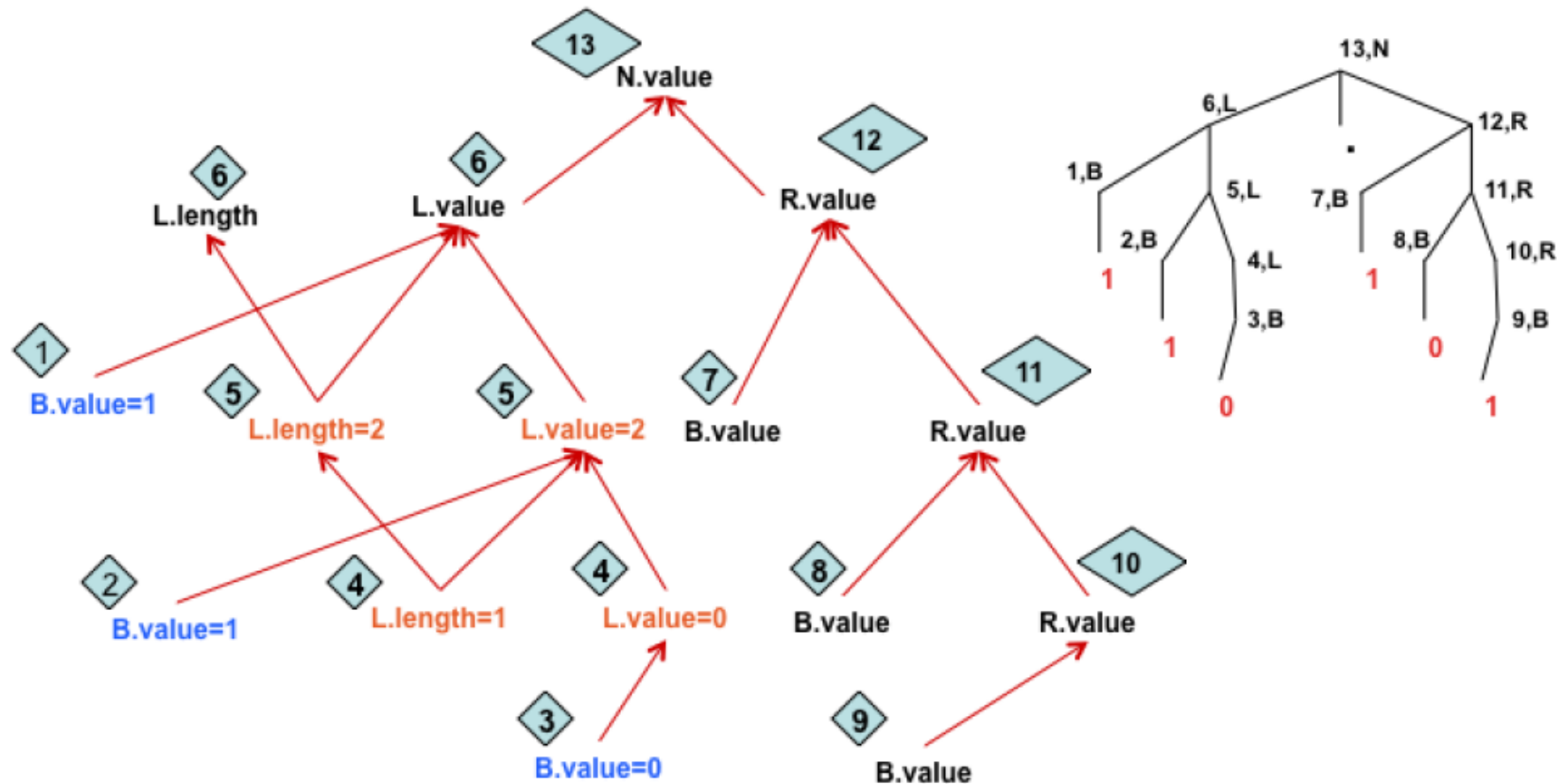
Node 3: $B \rightarrow 0$ $\{B.value \uparrow := 0\}$



Attribute Evaluation - 3

Node 4: $L \rightarrow B \{L.value \uparrow := B.value \uparrow; L.length \uparrow := 1\}$

Node 5: $L_1 \rightarrow BL_2 \{L_1.length \uparrow := L_2.length \uparrow + 1;$
 $L_1.value \uparrow := B.value \uparrow * 2^{L_2.length \uparrow} + L_2.value \uparrow\}$

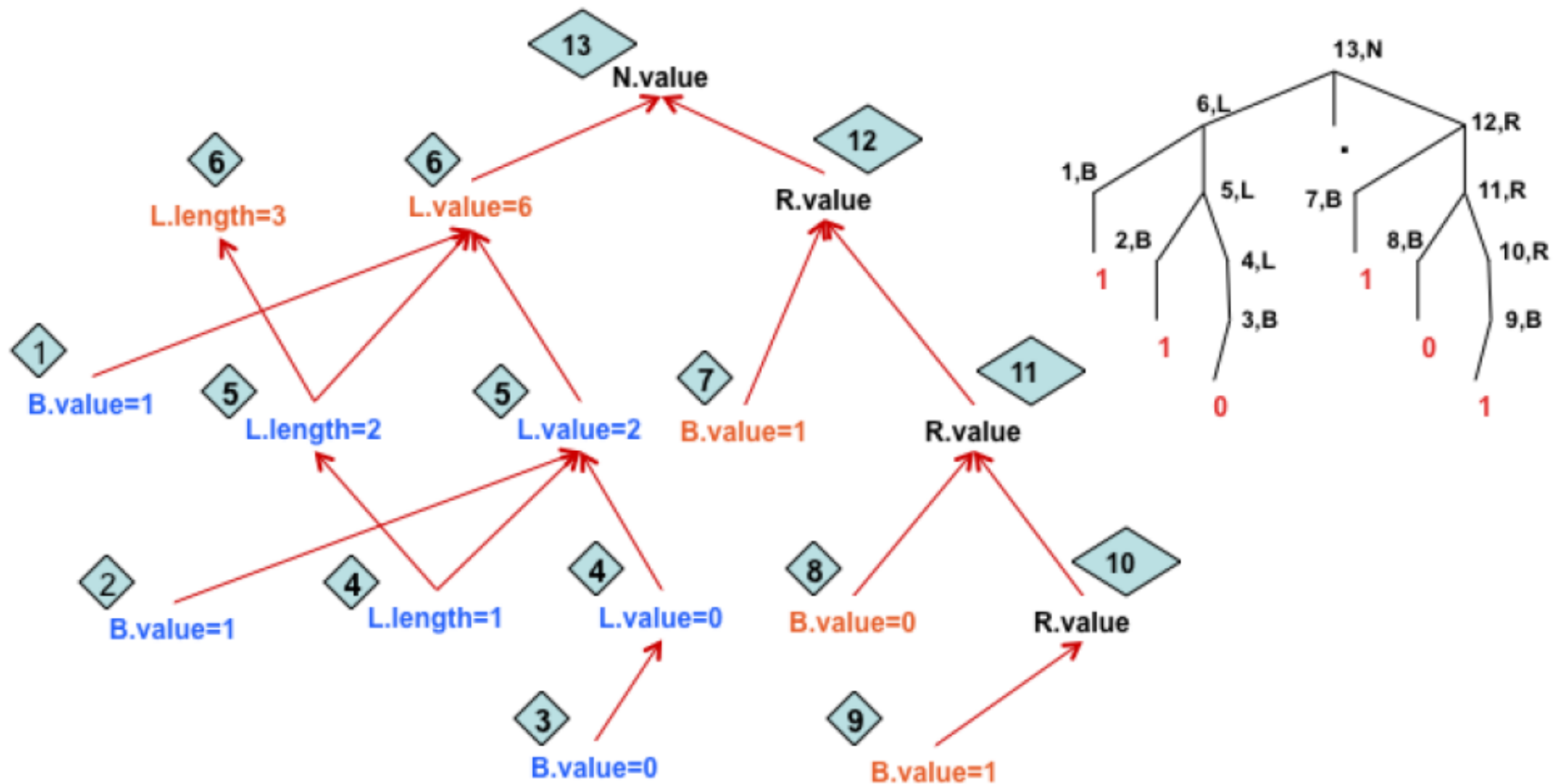


Attribute Evaluation - 4

Node 6: $L_1 \rightarrow BL_2 \{L_1.length \uparrow := L_2.length \uparrow + 1;$
 $L_1.value \uparrow := B.value \uparrow * 2^{L_2.length \uparrow} + L_2.value \uparrow\}$

Nodes 7,9: $B \rightarrow 1 \{B.value \uparrow := 1\}$

Node 8: $B \rightarrow 0 \{B.value \uparrow := 0\}$



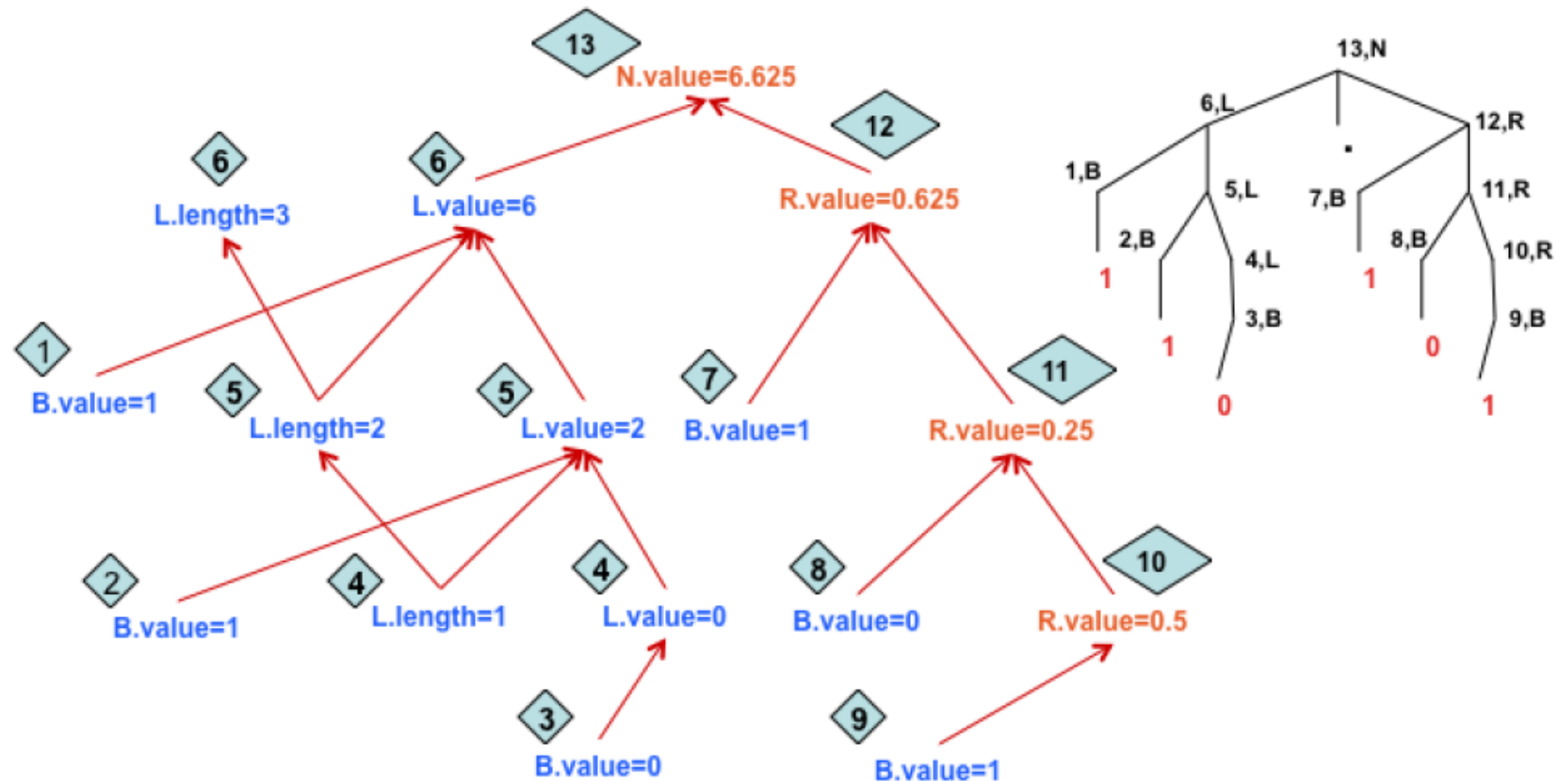
Attribute Evaluation - 5

Node 10: $R \rightarrow B \{R.value \uparrow := B.value \uparrow / 2\}$

Nodes 11,12:

$R_1 \rightarrow BR_2 \{R_1.value \uparrow := (B.value \uparrow + R_2.value \uparrow) / 2\}$

Node 13: $N \rightarrow L.R \{N.value \uparrow := L.value \uparrow + R.value \uparrow\}$



Another Example

- A simple grammar/AG for the evaluation of a real number from its bit-string representation
- **Example:** $110.1010 = 6 + 10/2^4 = 6 + 10/16 = 6 + 0.625 = 6.625$
- **CFG:** $N \rightarrow X.X, \quad X \rightarrow BX \mid B, \quad B \rightarrow 0 \mid 1$

Attributes

- $AS(N) = AS(B) = \{\text{value} \uparrow: \text{real}\},$
- $AS(X) = \{\text{length} \uparrow: \text{integer}, \text{value} \uparrow: \text{real}\}$

Rules

- 1 $N \rightarrow X_1.X_2 \{N.value \uparrow := X_1.value \uparrow + X_2.value \uparrow / 2^{X_2.length}\}$
- 2 $X \rightarrow B \{X.value \uparrow := B.value \uparrow; X.length \uparrow := 1\}$
- 3 $X_1 \rightarrow BX_2 \{X_1.length \uparrow := X_2.length \uparrow + 1;$
 $X_1.value \uparrow := B.value \uparrow * 2^{X_2.length \uparrow} + X_2.value \uparrow\}$
- 4 $B \rightarrow 0 \{B.value \uparrow := 0\}$
- 5 $B \rightarrow 1 \{B.value \uparrow := 1\}$

Example- AG for associating type information

- An AG for associating type information with names in variable declarations

CFG

$DList \rightarrow D \mid DList ; D$

$D \rightarrow T L$

$T \rightarrow int$

$T \rightarrow float$

$L \rightarrow ID$

$L \rightarrow L, ID$

$ID \rightarrow identifier$

Example: `int a,b,c; float x,y`

Example- AG for associating type information

- An AG for associating type information with names in variable declarations

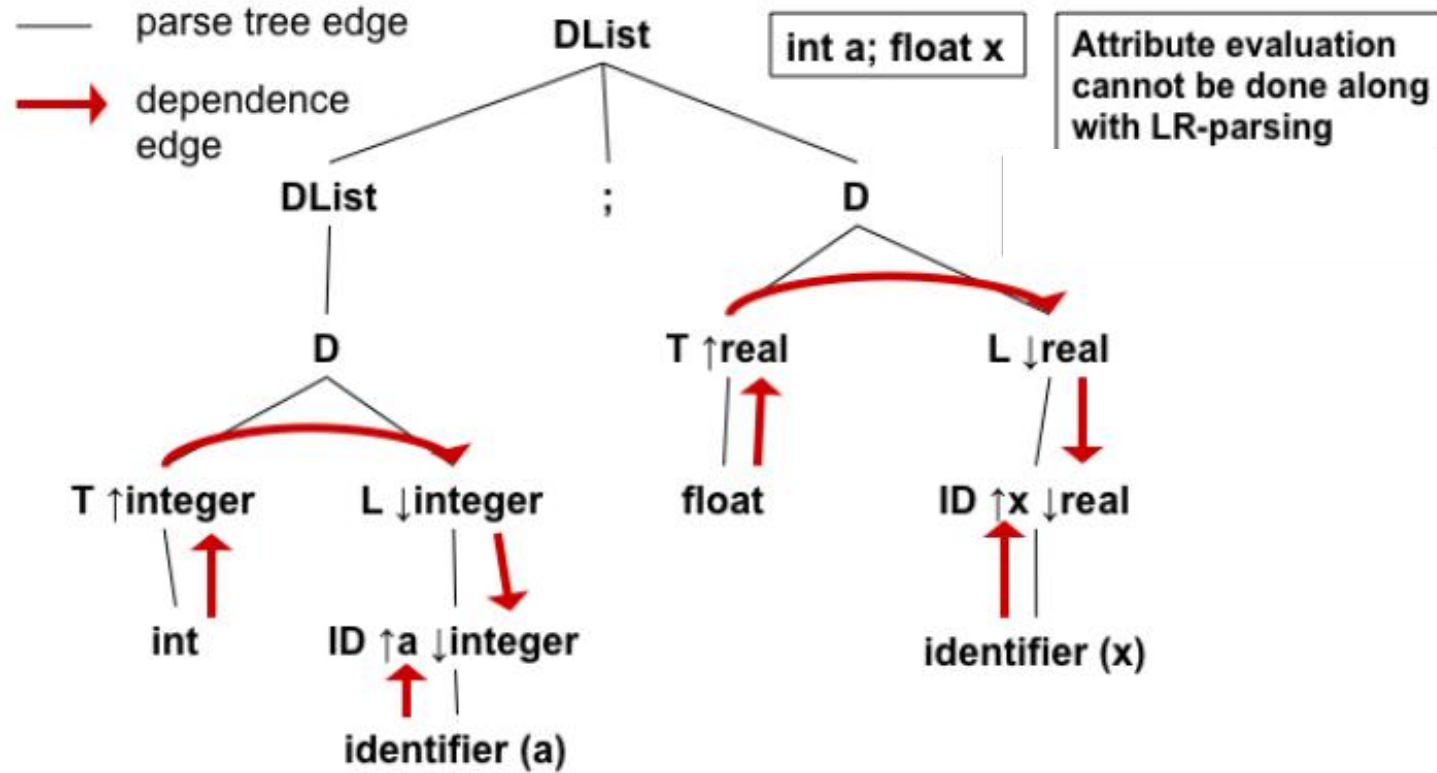
Attributes

- $AI(L) = \{\mathbf{type} \downarrow : \{\text{integer, real}\}\}$
- $AI(ID) = \{\mathbf{type} \downarrow : \{\text{integer, real}\}\}$
- $AS(T) = \{\mathbf{type} \uparrow : \{\text{integer, real}\}\}$
- $AS(ID) = AS(\text{identifier}) = \{\mathbf{name} \uparrow : \text{string}\}$

Rules

- 1) $DList \rightarrow D \mid DList ; D$
- 2) $D \rightarrow T L \{L.type \downarrow := T.type \uparrow\}$
- 3) $T \rightarrow \text{int} \{T.type \uparrow := \text{integer}\}$
- 4) $T \rightarrow \text{float} \{T.type \uparrow := \text{real}\}$
- 5) $L \rightarrow ID \{ID.type \downarrow := L.type \downarrow\}$
- 6) $L_1 \rightarrow L_2 , ID \{L_2.type \downarrow := L_1.type \downarrow; \quad ID.type \downarrow := L_1.type \downarrow\}$
- 7) $ID \rightarrow \text{identifier} \{ID.name \uparrow := \text{identifier.name} \uparrow\}$

Attribute evaluation for prev. example



1. $DList \rightarrow D \mid DList ; D$ 2. $D \rightarrow T \ L \ \{L.type \downarrow := T.type \uparrow\}$
3. $T \rightarrow int \ \{T.type \uparrow := integer\}$ 4. $T \rightarrow float \ \{T.type \uparrow := real\}$
5. $L \rightarrow ID \ \{ID.type \downarrow := L.type \downarrow\}$
6. $L_1 \rightarrow L_2 \ , \ ID \ \{L_2.type \downarrow := L_1.type \downarrow ; ID.type \downarrow := L_1.type \downarrow\}$
7. $ID \rightarrow identifier \ \{ID.name \uparrow := identifier.name \uparrow\}$

S-Attributed and L-Attributed Grammars

S-Attributed grammar

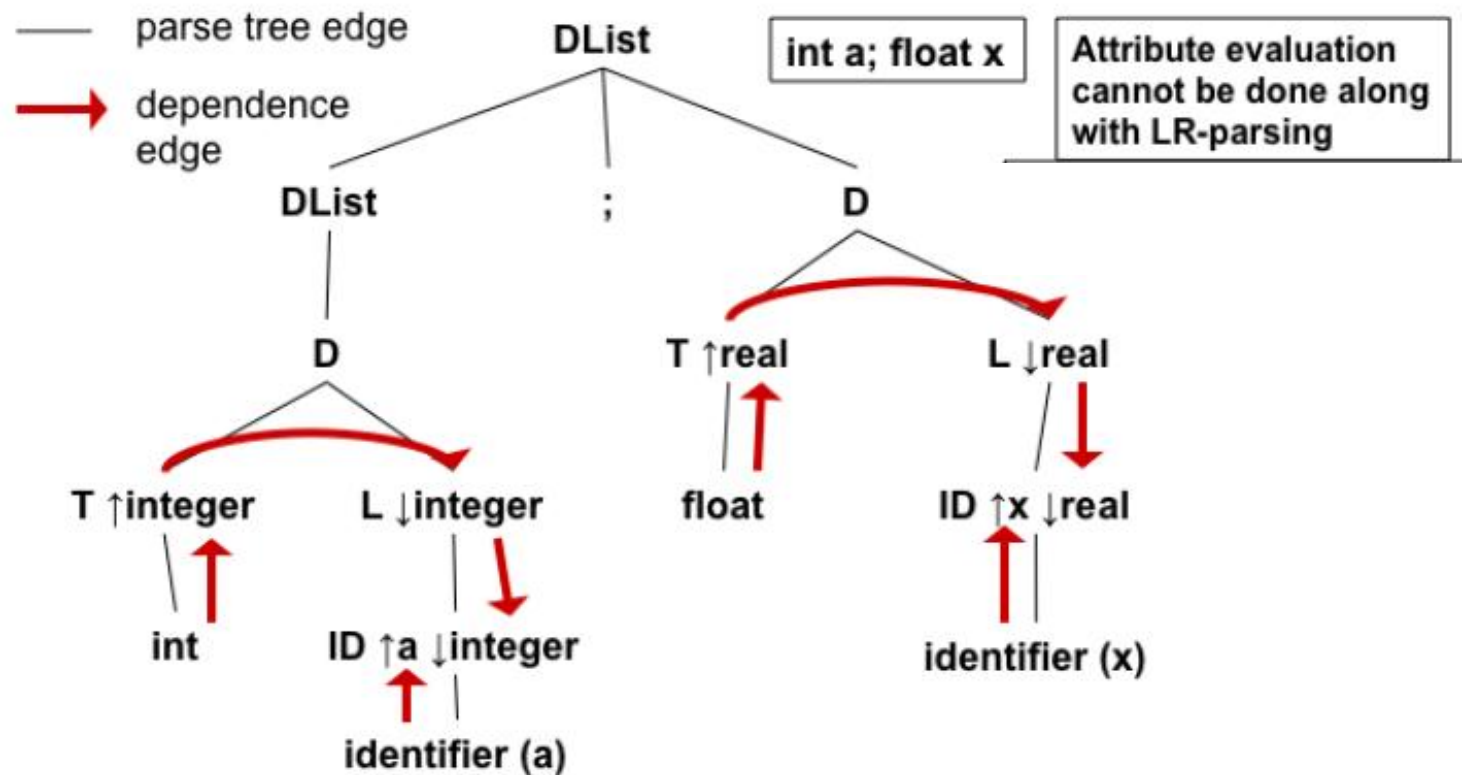
- An AG with **only synthesized attributes** is an **S-attributed grammar**
- Attributes of SAGs can be evaluated in any bottom-up order over a parse tree (**single pass**)
- **Attribute evaluation can be combined with LR-parsing** (YACC)

S-Attributed and L-Attributed Grammars

L-Attributed grammar

- In L-attributed grammars, each attribute must be
 - Synthesized, or
 - Inherited, but with the following limitations:
consider a production $p : A \rightarrow X_1X_2...X_n$. Let $X_i.a \in AI(X_i)$.
 $X_i.a$ may use only
 - elements of $AI(A)$
 - elements of $AI(X_k)$ or $AS(X_k)$, $k = 1, \dots, i-1$
(i.e., attributes of X_1, \dots, X_{i-1})
 - Attributes of X_i , provided no dependency cycle is formed
- In L-attributed grammars, attribute dependencies always go from left to right

Example: L-Attributed Grammar (LAG)



1. $DList \rightarrow D \mid DList ; D$
2. $D \rightarrow T L \{L.type \downarrow := T.type \uparrow\}$
3. $T \rightarrow int \{T.type \uparrow := integer\}$
4. $T \rightarrow float \{T.type \uparrow := real\}$
5. $L \rightarrow ID \{ID.type \downarrow := L.type \downarrow\}$
6. $L_1 \rightarrow L_2 , ID \{L_2.type \downarrow := L_1.type \downarrow ; ID.type \downarrow := L_1.type \downarrow\}$
7. $ID \rightarrow identifier \{ID.name \uparrow := identifier.name \uparrow\}$

Attribute Evaluation for LAGs

Input: A parse tree T with unevaluated attribute instances

Output: T with consistent attribute values

```
void dfvisit( $n$ : node)
```

```
{ for each child  $m$  of  $n$ , from left to right do
```

```
    { evaluate inherited attributes of  $m$ ;
```

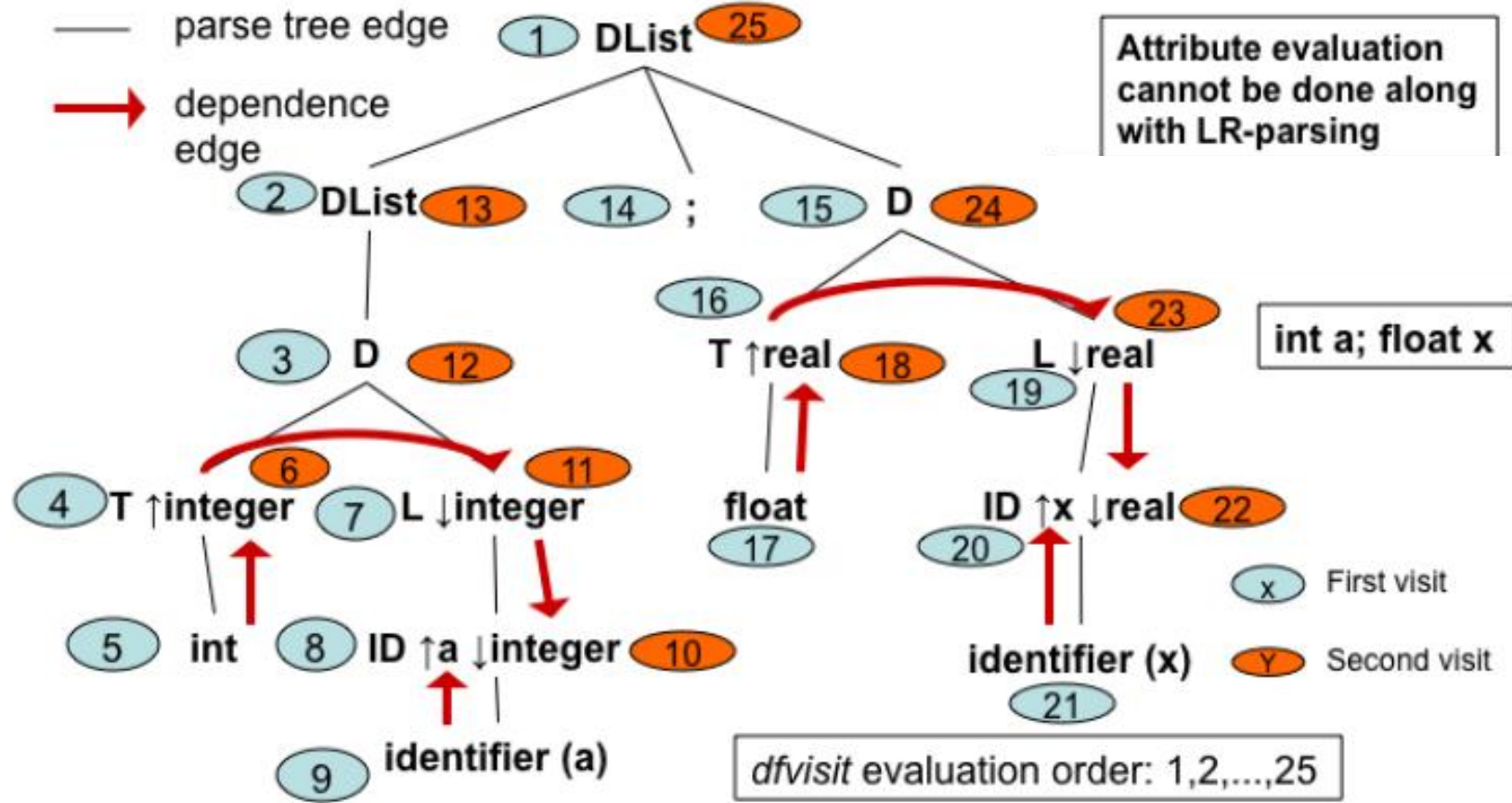
```
      dfvisit( $m$ )
```

```
    };
```

```
  evaluate synthesized attributes of  $n$ 
```

```
}
```

Example: Attribute Evaluation for LAG



1. $DList \rightarrow D \mid DList ; D$
2. $D \rightarrow T L \{L.type \downarrow := T.type \uparrow\}$
3. $T \rightarrow int \{T.type \uparrow := integer\}$
4. $T \rightarrow float \{T.type \uparrow := real\}$
5. $L \rightarrow ID \{ID.type \downarrow := L.type \downarrow\}$
6. $L_1 \rightarrow L_2 , ID \{L_2.type \downarrow := L_1.type \downarrow; ID.type \downarrow := L_1.type \downarrow\}$
7. $ID \rightarrow identifier \{ID.name \uparrow := identifier.name \uparrow\}$

Attributes

$AI(L) = \{type \downarrow : \{integer, real\}\}$

$AI(ID) = \{type \downarrow : \{integer, real\}\}$

$AS(T) = \{type \uparrow : \{integer, real\}\}$

$AS(ID) = AS(identifier) = \{name \uparrow : string\}$

Example of Non-LAG

- An AG for associating type information with names in variable declarations

Attributes

- $AI(L) = AI(ID) = \{\text{type } \downarrow: \{\text{integer, real}\}\}$
- $AS(T) = \{\text{type } \uparrow: \{\text{integer, real}\}\}$
- $AS(ID) = AS(\text{identifier}) = \{\text{name } \uparrow: \text{string}\}$

Rules

1. $DList \rightarrow D \mid DList ; D$
2. $D \rightarrow L : T \{L.type \downarrow := T.type \uparrow\}$
3. $T \rightarrow \text{int} \{T.type \uparrow := \text{integer}\}$
4. $T \rightarrow \text{float} \{T.type \uparrow := \text{real}\}$
5. $L \rightarrow ID \{ID.type \downarrow := L.type \downarrow\}$
6. $L1 \rightarrow L2 , ID \{L2.type \downarrow := L1.type \downarrow; ID.type \downarrow := L1.type \downarrow\}$
7. $ID \rightarrow \text{identifier} \{ID.name \uparrow := \text{identifier.name } \uparrow\}$

Example: $a,b,c: \text{int}; x,y: \text{float}$

$a, b,$ and c are tagged with type integer

$x, y,$ are tagged with type real

Syntax-Directed Definition

- A **syntax-directed definition** is a context-free grammar where
 - **attributes** are associated with **grammar symbols**.
 - Rules for computing the attributes are associated with the **production rules**.
 - There should be no circularity in the definition.
- These are called *attribute grammars* (when the definition do not have any side-effects).
- **Attribute grammars** have no side-effects and allow any evaluation order consistent with the dependency graph.

Evaluation: Syntax-Directed Definition

- SDD is a CFG along with attributes and rules.
- Attributes are associated with grammar symbols.
- Rules are associated with productions
- Attribute Grammars
 - Synthesized, inherited attributes
 - Dependency graphs
- Evaluation: Parse-tree methods (**cyclic graphs fail**)
 - build the parse tree
 - build the dependency graph
 - topological sort the graph
 - evaluate it
- Focus on S-attributed, L-attributed grammars
 - **S-attributed** (evaluation together with LR parsing)
 - **L-attributed** (evaluation together with LL parsing)

Syntax-directed translation/ Attribute translation grammar

Translation

- To perform **semantic actions** along with **parsing actions** (such as a **reduction**), we associate required computation with the production rules.
Computed values are **propagated** as **attributes** of **non-terminals**.
- Our main goal is **translation**
- Semantic actions to translate ***source language program*** to a ***target program*** often go hand-in-hand with parsing.
It is called **syntax-directed-translation**

Syntax-directed translation

- A *syntax-directed translation* is an executable specification of SDD.
 Fragments of programs are associated with different points in the **production rules**.
- The order of execution of the code is important in this case.

SDD and SDT Scheme

- **SDD**: Specifies the values of attributes by associating semantic rules with the productions.
- **SDT scheme**: embeds program fragments (also called semantic actions) within production bodies.
 - The position of the action defines the order in which the action is executed (in the middle of production or end).
- **SDD** is easier to read; easy for specification.
- **SDT** scheme – can be more efficient; easy for implementation.

Syntax-Directed Translation

- Attach rules or program fragments to productions in a grammar.

- **Syntax directed definition (SDD)**

- $E_1 \rightarrow E_2 + T$ $E_1.\text{code} = E_2.\text{code} || T.\text{code} || '+'$

- **Syntax directed translation Scheme (SDT)**

- $E \rightarrow E + T$ `{print '+'}` // semantic action
 - $F \rightarrow \text{id}$ `{print id.val}`

Example: SDD Vs. SDT Scheme (infix to postfix transformation)

<i>SDT Scheme</i>		<i>SDD</i>	
$E \rightarrow E + T$	$\{\text{print}'+' \}$	$E \rightarrow E + T$	$E.\text{code} = E.\text{code} T.\text{code} '+'$
$E \rightarrow E - T$	$\{\text{print}'- ' \}$	$E \rightarrow E - T$	$E.\text{code} = E.\text{code} T.\text{code} '-'$
$E \rightarrow T$		$E \rightarrow T$	$E.\text{code} = T.\text{code}$
$T \rightarrow 0$	$\{\text{print}'0' \}$	$T \rightarrow 0$	$T.\text{code} = '0'$
$T \rightarrow 1$	$\{\text{print}'1' \}$	$T \rightarrow 1$	$T.\text{code} = '1'$
...		...	
$T \rightarrow 9$	$\{\text{print}'9' \}$	$T \rightarrow 9$	$T.\text{code} = '9'$

Syntax-Directed Translation

$A \rightarrow \{Action_1\} B \{Action_2\} C \{Action_3\}$

$Action_1$: takes place before parsing of the input corresponding to the non-terminal B .

$Action_2$: takes place after consuming the input for B , but before consuming the input for C .

$Action_3$: takes place at the time of reduction of BC to A or after consuming the input corresponding to BC .

Attribute Translation Grammar

- Apart from attribute computation rules, we may require some actions (*program segments*) that performs operations such as
 - *printing output,*
 - *symbol table operations,*
 - *writing generated code to a file,* etc
- These actions can be added to both SAGs and LAGs (making them, **SATG** and **LATG** resp.)
- As a result of these action code segments, evaluation orders may be constrained

Example: LAG, LATG

LAG (notice the changed grammar)

1. $Decl \rightarrow DList\$$
2. $DList \rightarrow D D'$
3. $D' \rightarrow \epsilon \mid ; DList$
4. $D \rightarrow T L \{L.type \downarrow := T.type \uparrow\}$
5. $T \rightarrow int \{T.type \uparrow := integer\}$
6. $T \rightarrow float \{T.type \uparrow := real\}$
7. $L \rightarrow ID L' \{ID.type \downarrow := L.type \downarrow; L'.type \downarrow := L.type \downarrow; \}$
8. $L' \rightarrow \epsilon \mid , L \{L.type \downarrow := L'.type \downarrow; \}$
9. $ID \rightarrow identifier \{ID.name \uparrow := identifier.name \uparrow\}$

LATG (notice the changed grammar)

1. $Decl \rightarrow DList\$$
2. $DList \rightarrow D D'$
3. $D' \rightarrow \epsilon \mid ; DList$
4. $D \rightarrow T \{L.type \downarrow := T.type \uparrow\} L$
5. $T \rightarrow int \{T.type \uparrow := integer\}$
6. $T \rightarrow float \{T.type \uparrow := real\}$
7. $L \rightarrow id \{insert_symtab(id.name \uparrow, L.type \downarrow);$
 $L'.type \downarrow := L.type \downarrow; \} L'$
8. $L' \rightarrow \epsilon \mid , \{L.type \downarrow := L'.type \downarrow; \} L$

Example: SATG

1. $Decl \rightarrow DList\$$
2. $DList \rightarrow D \mid DList ; D$
3. $D \rightarrow T L \{patchtype(T.type \uparrow, L.namelist \uparrow); \}$
4. $T \rightarrow int \{T.type \uparrow := integer\}$
5. $T \rightarrow float \{T.type \uparrow := real\}$
6. $L \rightarrow id \{sp = insert_symtab(id.name \uparrow);$
 $L.namelist \uparrow = makelist(sp); \}$
7. $L_1 \rightarrow L_2 , id \{sp = insert_symtab(id.name \uparrow);$
 $L_1.namelist \uparrow = append(L_2.namelist \uparrow, sp); \}$

Syntax Directed Translations

1. Construct a parse tree
 2. Compute the values of the attributes at the nodes of the tree by visiting the tree
- **Translation can be done during parsing.**
 - class of SDTs called “L-attributed translations”.
 - class of SDTs called “S-attributed translations”.

LL Parser and Actions

How does an **LL** parser **handle** (aka - execute) **actions**?

Expand productions before scanning RHS symbols, so:

1. **push actions** onto parse stack like other grammar symbols
2. **pop** and perform action when it comes to top of parse stack

LL Parsers and Actions

```
push EOF
push Start Symbol
token  $\leftarrow$  next_token()
repeat
    pop X
    if X is a terminal or EOF then
        if X = token then
            token  $\leftarrow$  next_token()
        else error()
    else if X is an action
        perform X
    else /* X is a non-terminal */
        if  $M[X, \text{token}] = X \rightarrow Y_1 Y_2 \cdots Y_k$  then
            push  $Y_k, Y_{k-1}, \cdots, Y_1$ 
        else error()
until X = EOF
```

LR Parsers and Action Symbols

- What about LR parsers?
 - Scan entire RHS before applying production, so:
 - cannot perform actions until entire RHS scanned
 - **can only place actions at very end of RHS of production**
 - introduce new marker non-terminals and corresponding productions to get around this restriction

$$A \rightarrow w \text{ action } \beta$$

becomes

$$A \rightarrow M\beta$$

$$M \rightarrow w \text{ action}$$

****Yacc, Bison does this automatically**

Embedded Actions in Parser Generators

- Embedded actions may create some issues in parser-generators such as **Bison, YACC**
- Bison replaces embedded actions in a production rule by an **ϵ -production** and associates the embedded action with the new rule
- The above transformation may change the nature of the grammar

Example- Embedded Actions in Parser Generators

- Consider the following grammar which is **LALR**

$$S \rightarrow A \mid B, A \rightarrow aba, B \rightarrow abb$$

- Let an embedded action be introduced as shown below:

$$S \rightarrow A \mid B, A \rightarrow a \{\text{action}\} ba, B \rightarrow abb$$

- Bison modifies the grammar to:

$$S \rightarrow A \mid B, A \rightarrow aMba, B \rightarrow abb, \\ M \rightarrow \varepsilon \{\text{action}\} .$$

The modified grammar is no longer LALR

Building Abstract Syntax Trees

Building Abstract Syntax Trees

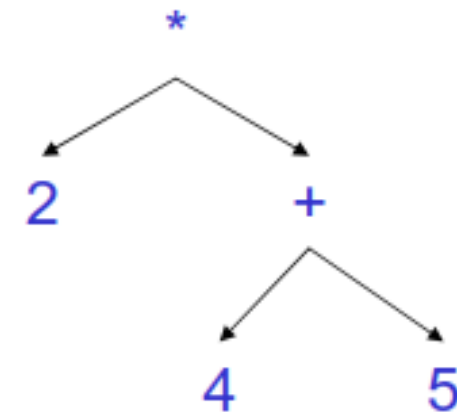
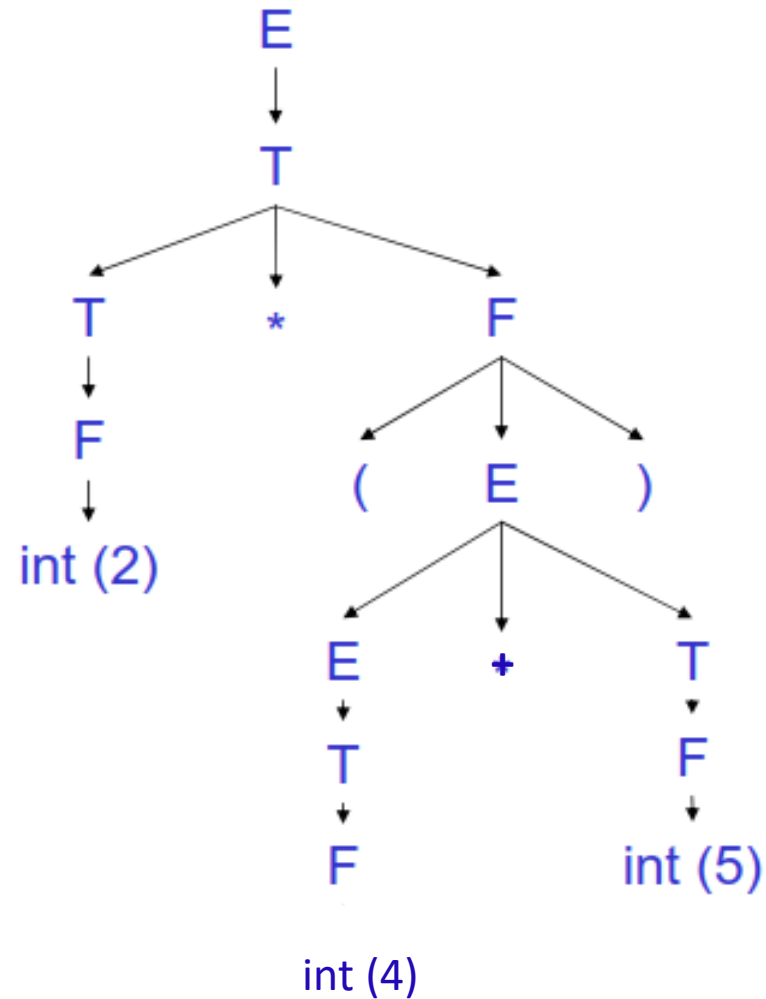
- Examples so far, streams of tokens translated into
 - integer values, or
 - types
- Translating into ASTs is not very different

AST Vs. Parse Tree

- AST is condensed form of a parse tree
 - operators appear at *internal nodes*, not at leaves.
 - "Chains" of single productions are collapsed.
 - Syntactic details are omitted
 - e.g., parentheses, commas, semi-colons
- AST is a better structure for later compiler stages
 - omits details having to do with the source language,
 - only contains information about the essential structure of the program.

Example: $2 * (4 + 5)$

parse tree vs AST



Rules: Building AST

$E_1 \rightarrow E_2 + T$ $E_1.trans = \text{new PlusNode}(E_2.trans, T.trans)$
 $E \rightarrow T$ $E.trans = T.trans$
 $T_1 \rightarrow T_2 * F$ $T_1.trans = \text{new TimesNode}(T_2.trans, F.trans)$
 $T \rightarrow F$ $T.trans = F.trans$
 $F \rightarrow \text{int}$ $F.trans = \text{new IntLitNode}(\text{int.value})$
 $F \rightarrow (E)$ $F.trans = E.trans$

