CC WEEK 11-12

Prepared for: 7th Sem, CE, DDU

Prepared by: Niyati J. Buch

Contents

Contents

What is Semantic Analysis?

- Source program → **Lexical Analyzer** → Token stream → **Syntax Analyzer** → Syntax tree → **Semantic Analyzer** → Annotated syntax tree → **Intermediate Code Generator**

- Semantic consistency that cannot be handled at the parsing stage is handled in this phase.

- Parsers cannot handle context-sensitive features of the programming languages.

Static vs. Dynamic Semantics

- There are some **static semantics** of the programming languages that are checked by the semantic analyzer:
  - If variables are declared before use.
  - If types match on both sides of the assignment.
  - If parameter types and number match in the declaration and use.
- Compilers can generate the code to check **dynamic semantics** of the programming languages only at runtime:
  - Whether an overflow will occur during an arithmetic operation
  - Whether the array limits will be crossed during the execution
  - Whether recursion will cross the stack limits
  - Whether the 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

```
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 “+”

Static Semantics: Errors given by gcc Compiler

```
#include<stdio.h>
int dot_product(int a[], int b[])
{return 1;}
int main(){
        int a[10]={1,2,3,4,5,6,7,8,9,10};
        int b[10]={1,2,3,4,5,6,7,8,9,10};
        printf("%d", dot_product(b));
        printf("%d", dot_product(a,b,a));
        int p[10];
        p=dot_product(a,b);
        printf("%d",p);
}
```

Static Semantics: Errors given by gcc Compiler

```
1.  #include<stdio.h>
2.  int dot_product(int a[], int b[])
    {return 1;}
3.  int main(){
4.          int a[10]={1,2,3,4,5,6,7,8,9,10};
5.          int b[10]={1,2,3,4,5,6,7,8,9,10};
6.          printf("%d", dot_product(b));
7.          printf("%d", dot_product(a,b,a));
8.          int p[10];
9.          p=dot_product(a,b);
10.         printf("%d",p);
11. }
```

6: error: too few arguments to function 'dot_product'

7: error: too many arguments to function 'dot_product'

9: error: assignment to expression with array type

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

Semantic Analysis

- Type information is stored in the symbol table or the syntax tree.
  - Types of variables, function parameters, array dimensions, etc.
  - Used not only for semantic validation but also for the subsequent phases of compilation.
- If the declarations need not appear before the use, then the 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

- Let **G = (N, T, P, S)** be a context-free grammar (**CFG**) consisting of a finite set of grammar rules where
  - **N** is a set of non-terminal symbols.
  - **T** is a set of terminals where **N ∩ T = NULL.**
  - **P** is a set of rules, **P: N → (N ∪ T)***
  - **S** is the start symbol.
- and let **V = N ∪ T**.
- Every symbol **X of V** has associated with it, a set of attributes (denoted by **X:a**; **X:b**, etc.)
- Hence, the name is **attribute grammar**.

Attribute Types

- **Inherited attributes**
  - denoted by **AI(X)**

- **Synthesized attributes**
  - denoted by **AS(X)**

- 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.

Attribute Grammar

- Each attribute takes the values from a specified domain (finite or infinite) [domain is its type]
  - Typical domains of attributes are, integers, reals, characters, strings, booleans, structures, etc.
  - New domains can be constructed from the given domains by mathematical operations like cross product, map, etc.
- **Example: array**
  - a map, N → D, where, N and D are domains of natural numbers and the given objects, respectively
- **Example :structure**
  - a cross-product, A1 X A2 X ... X An, where n is the number of fields in the structure, and Ai is the domain of the ith field

Attribute Computation Rules

- A production **p ∈ P** has a set of attribute computation rules.
- 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.
- 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.

Synthesized and Inherited Attributes

- **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 under consideration.

Attribute Grammar - Example 1

- The following CFG(context free grammar)

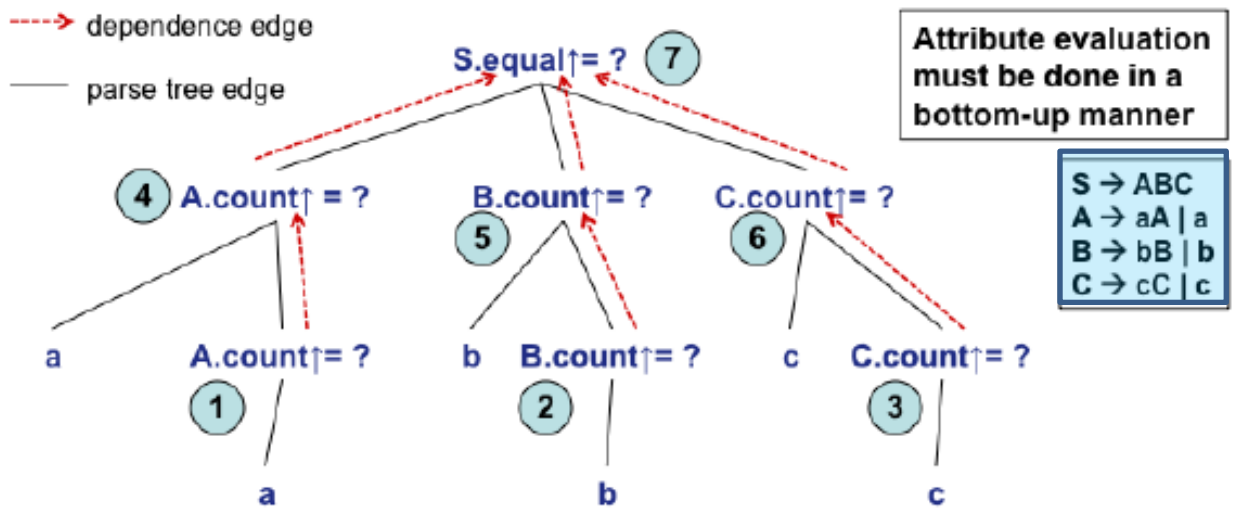
  S → ABC, A → aA|a, B→ bB|b, C →cC|c

  generates: **L(G) = {$a^m b^n c^p$ | m, n, p ≥ 1}**
- We define an AG (attribute grammar) based on this CFG to generate **L = {$a^n b^n c^n$ | n ≥ 1}**
- All the non-terminals will have only synthesized attributes

  AS(S) = {equal ↑: {T, F}}

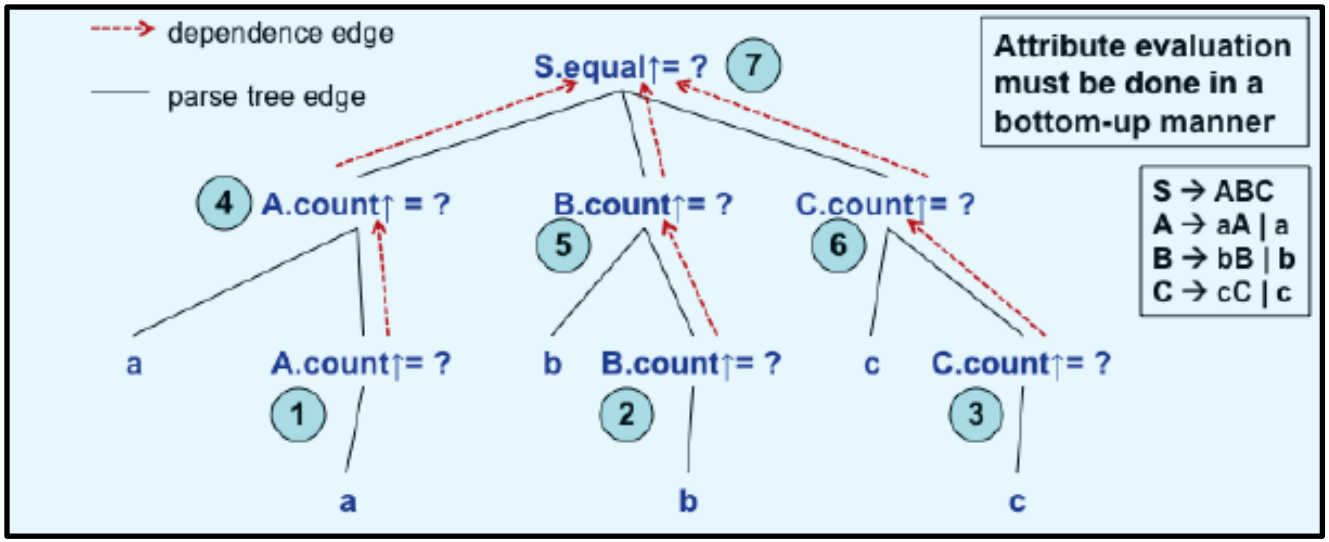
  AS(A) = AS(B) = AS(C) = {count ↑: integer}
- Up arrow means synthesized attribute
- Down arrow means inherited attribute

Attribute Grammar - Example 1



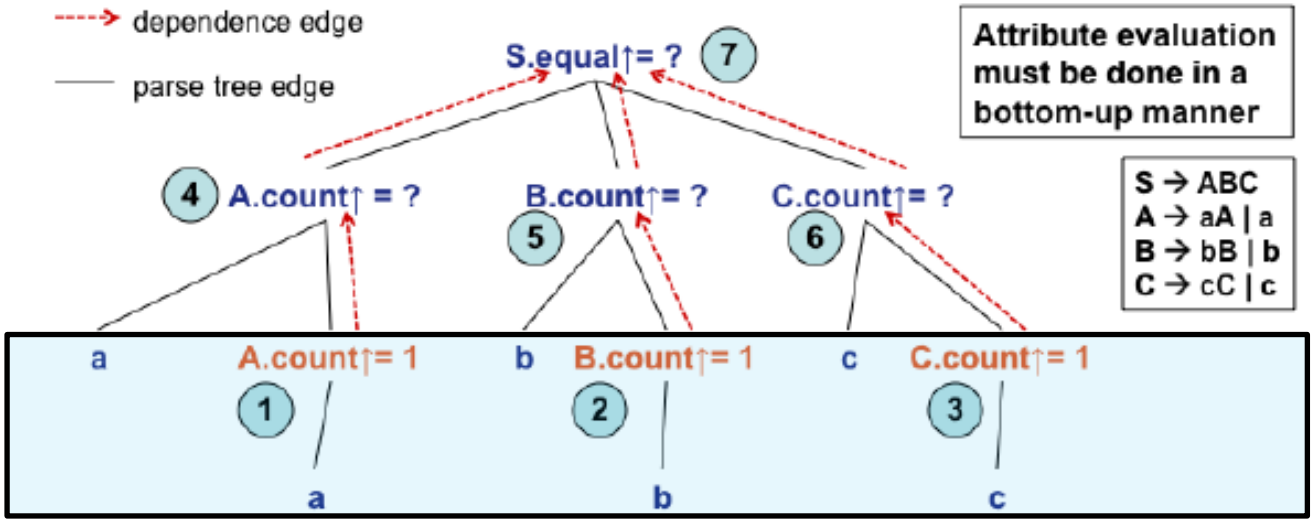
1. $S \rightarrow ABC$ $\{S.equal \uparrow := if\ A.count \uparrow = B.count \uparrow\ \&\ B.count \uparrow = C.count \uparrow\ then\ T\ else\ F\}$
2. $A_1 \rightarrow aA_2$ $\{A_1.count \uparrow := A_2.count \uparrow +1\}$
3. $A \rightarrow a$ $\{A.count \uparrow := 1\}$
4. $B_1 \rightarrow bB_2$ $\{B_1.count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

Attribute Grammar - Example 1



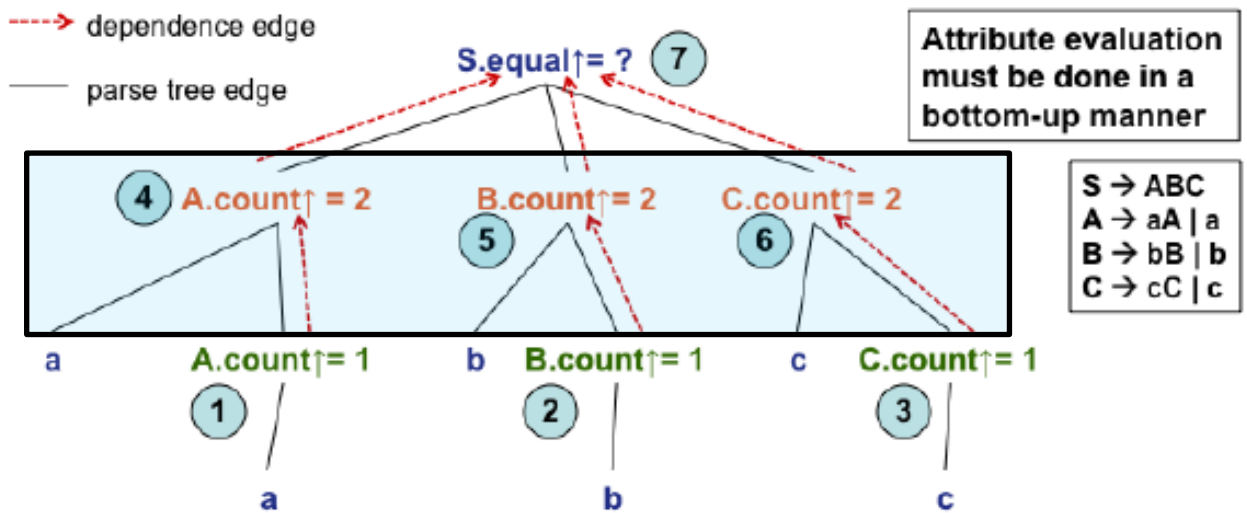
1. $S \rightarrow ABC$ $\{S.equal \uparrow := if\ A.count \uparrow = B.count \uparrow\ \&\ B.count \uparrow = C.count \uparrow\ then\ T\ else\ F\}$
2. $A_1 \rightarrow aA_2$ $\{A_1.count \uparrow := A_2.count \uparrow +1\}$
3. $A \rightarrow a$ $\{A.count \uparrow := 1\}$
4. $B_1 \rightarrow bB_2$ $\{B_1.count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

Attribute Grammar - Example 1



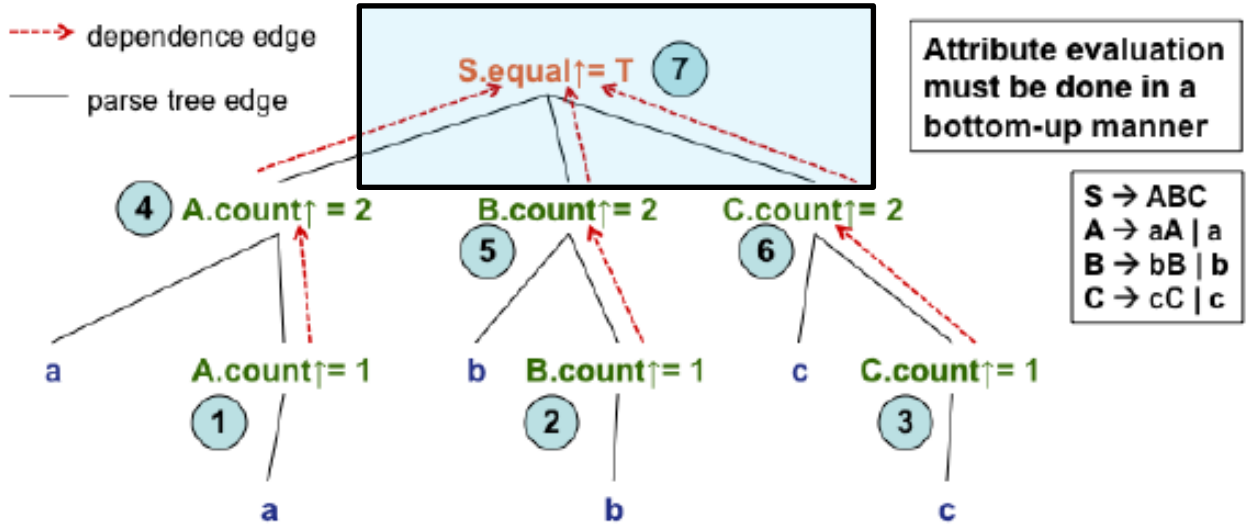
1. $S \rightarrow ABC$ $\{S.equal\uparrow := if\ A.count\uparrow = B.count\uparrow\ \&\ B.count\uparrow = C.count\uparrow\ then\ T\ else\ F\}$
2. $A_1 \rightarrow aA_2$ $\{A_1.count\uparrow := A_2.count\uparrow + 1\}$
3. $A \rightarrow a$ $\{A.count\uparrow := 1\}$
4. $B_1 \rightarrow bB_2$ $\{B_1.count\uparrow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

Attribute Grammar - Example 1



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

Attribute Grammar - Example 1



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

Attribute Dependence 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|b is an attribute instance of some tree node}

  E = {(b, c)|b, c Є V, b and c are attributes of grammar symbols in the same production p of B, and the value of b is used for computing the value of c in an attribute computation rule associated with production p}

Attribute Dependence Graph

- An AG(attribute grammar) G is **non-circular**, if and only if for all trees T derived from G, DG(T) is acyclic
  - Non-circularity is very expensive to determine (exponential in the size of the grammar)
  - Therefore, our interest will be in subclasses of AGs whose non-circularity can be determined efficiently

- Assigning consistent values to the attribute instances in DG(T) is attribute evaluation.

Attribute Evaluation Strategy

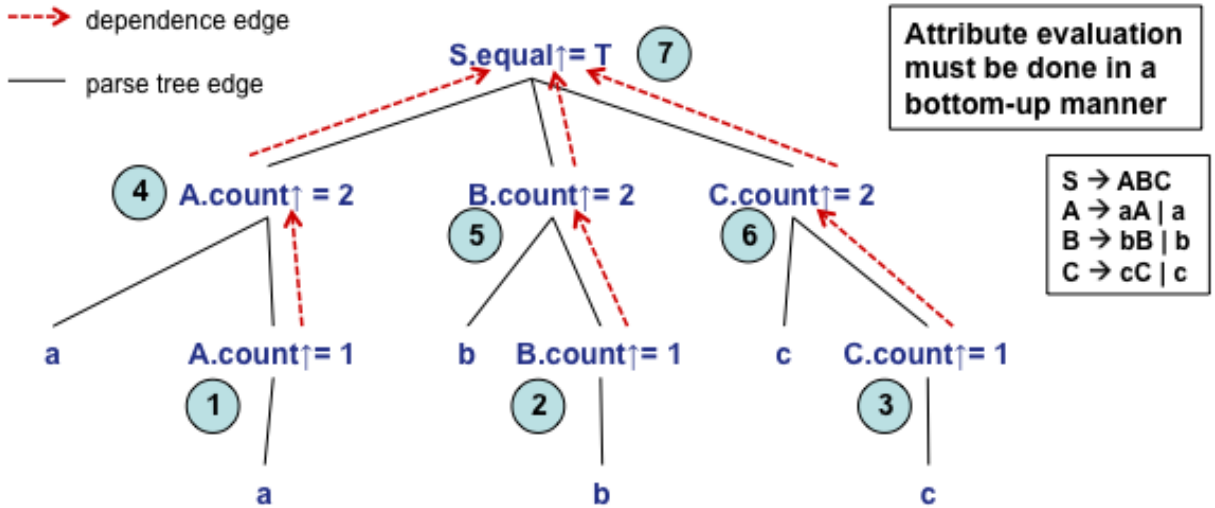
- Construct the parse tree
- Construct the dependence graph
- Perform topological sort on the dependence graph and obtain an evaluation order
- Evaluate attributes according to this order using the corresponding attribute evaluation rules attached to the respective productions
- Multiple attributes at 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);
(W is a queue) Let W = {b | b ∈ V & indegree(b) = 0};
  while W ≠ φ do
    { remove some b from W;
      value(b) := value defined by appropriate attribute
                  computation rule;
      for all (b, c) ∈ E do
        { indegree(c) := indegree(c) − 1;
          if indegree(c) = 0 then W := W ∪ {c};
        }
    }
}
```

Dependence Graph for Example 1



1,2,3,4,5,6,7 and 2,3,6,5,1,4,7 are two possible evaluation orders. 1,4,2,5,3,6,7 can be used with LR-parsing. The right-most derivation is below (its reverse is LR-parsing order)

S => ABC => ABcC => ABcc => AbBcc => Abbcc => aAbbcc => aabbcc

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

Syntax Directed Translation

=

Grammar + Semantic Rules

S → ABC
A → aA|a
B→ bB|b
C →cC|c

+

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

Example 2

- Write an attribute grammar for the evaluation of a real number from its bit-string representation.

- Example: $(\mathbf{110.101})_2 = (\mathbf{6.625})_{10}$

110	.	101
110 → **6**		101→5 (decimal value)/(2^ no. of bits) = 5 / 2^3 = 5 / 8 = **0.625**

Example 2

- Write an attribute grammar for the evaluation of a real number from its bit-string representation.

$N \rightarrow L \,.\, L$

$L \rightarrow BL \mid B$

$B \rightarrow 0 \mid 1$

Example 2

N→L.L

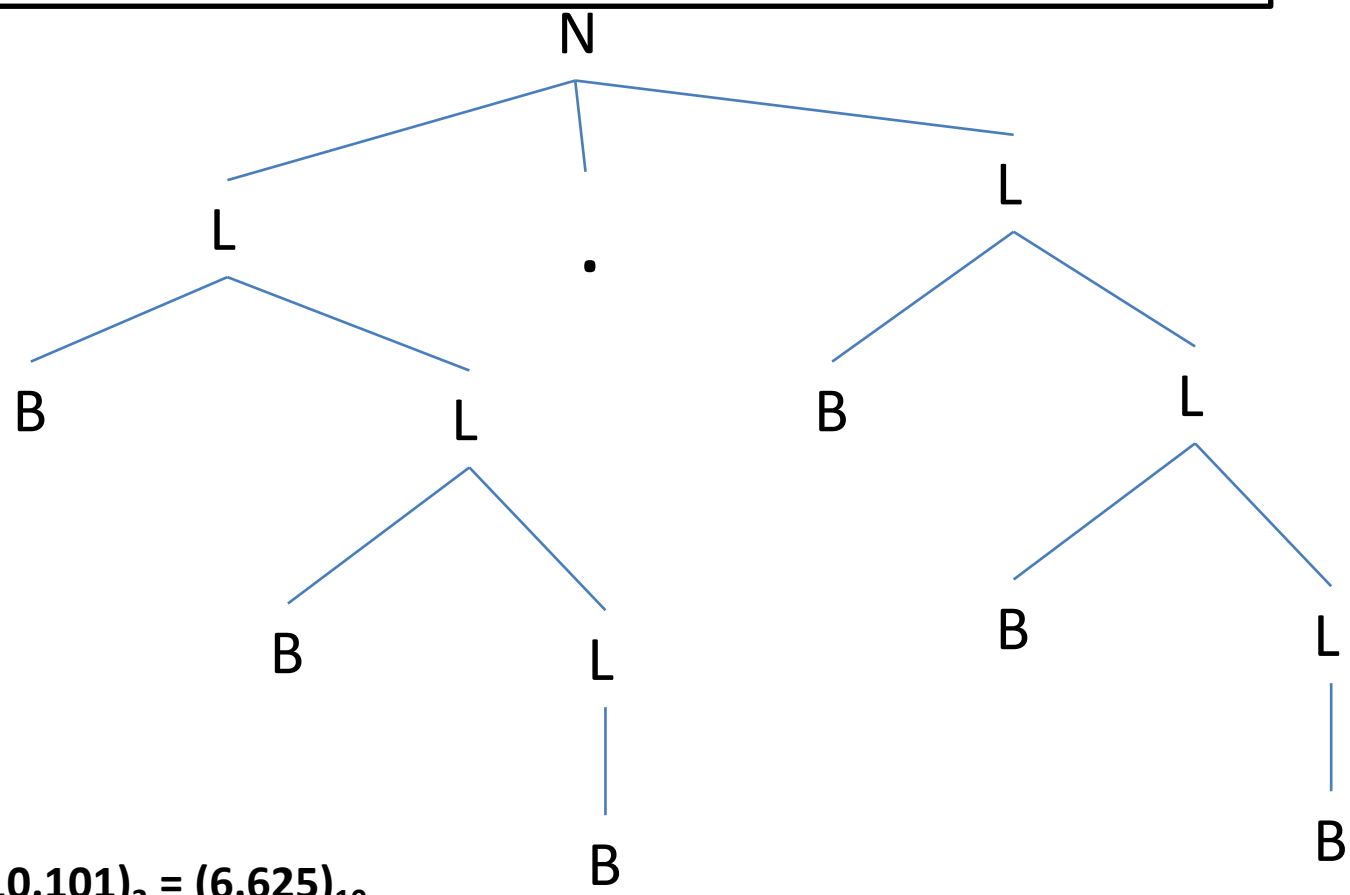
L→BL | B

B →0|1

- AS(N)= AS(B) ={val↑:real}
- AS(L) ={cnt↑:integer, val↑:real}

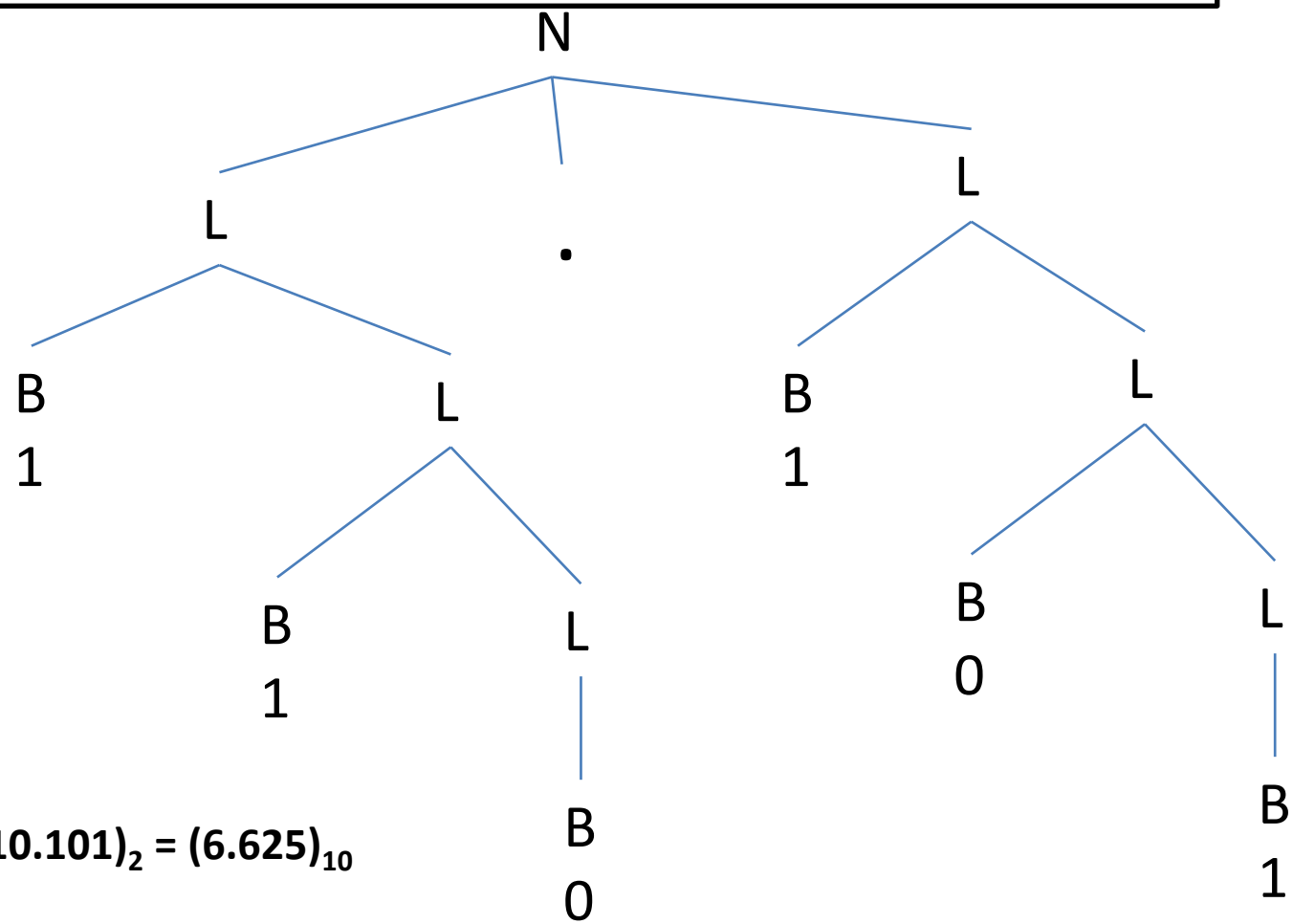
1. N → $L_1.L_2$ {N.val = L_1.val + (L_2.val / 2^L_2.cnt)}
2. L → BL_1 {L.cnt=L_1.cnt+1; L.val=L_1.val+(B.val* 2^L_1.cnt)}
3. L → B {L.cnt = 1 ; L.val = B.val}
4. B → 0 {B.val = 0}
5. B → 1 {B.val = 1}

1.	$N \rightarrow L_1.L_2$	$\{N.val = L_1.val + (L_2.val / 2\hat{}L_2.cnt)\}$
2.	$L \rightarrow BL_1$	$\{L.cnt=L_1.cnt+1;\ L.val=L_1.val+(B.val * 2\hat{}L_1.cnt)\}$
3.	$L \rightarrow B$	$\{L.cnt = 1\ ;\ L.val = B.val\}$
4.	$B \rightarrow 0$	$\{B.val = 0\}$
5.	$B \rightarrow 1$	$\{B.val = 1\}$



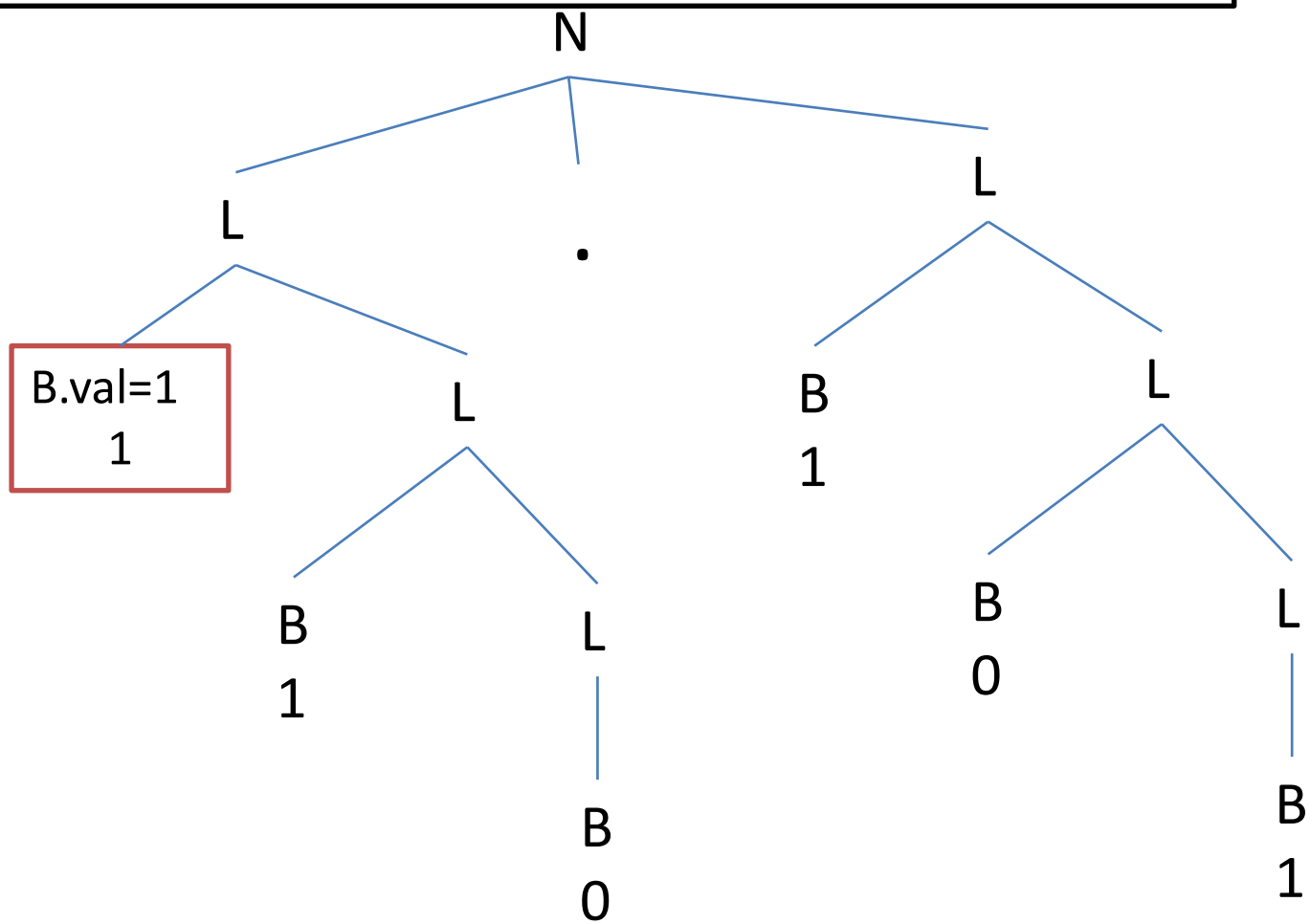
Example: $(\mathbf{110.101})_2 = (\mathbf{6.625})_{10}$

1.	$N \rightarrow L_1.L_2$	$\{N.val = L_1.val + (L_2.val / 2\hat{}L_2.cnt)\}$
2.	$L \rightarrow BL_1$	$\{L.cnt=L_1.cnt+1;\ L.val=L_1.val+(B.val * 2\hat{}L_1.cnt)\}$
3.	$L \rightarrow B$	$\{L.cnt = 1\ ;\ L.val = B.val\}$
4.	$B \rightarrow 0$	$\{B.val = 0\}$
5.	$B \rightarrow 1$	$\{B.val = 1\}$

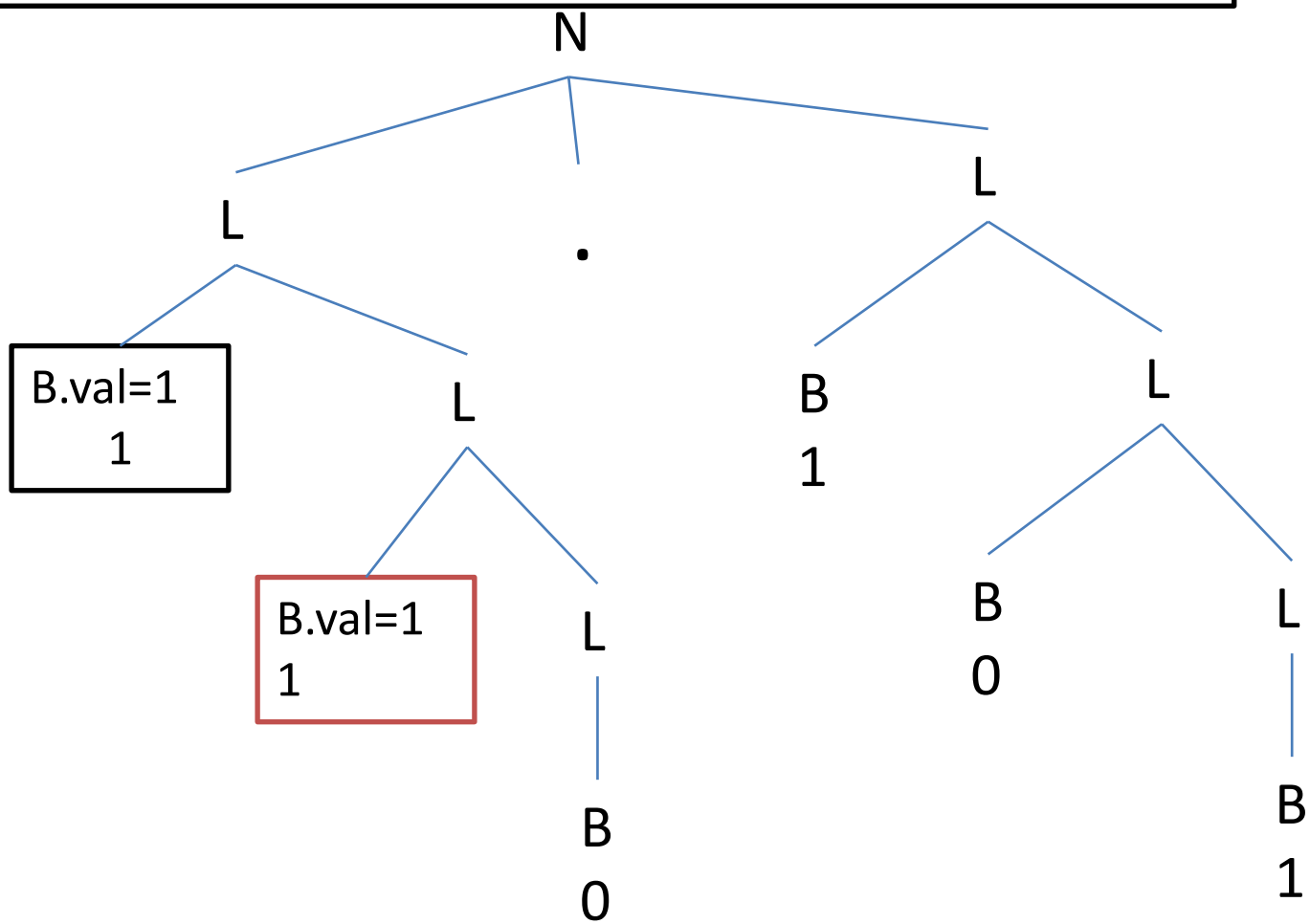


Example: $(\mathbf{110.101})_2 = (\mathbf{6.625})_{10}$

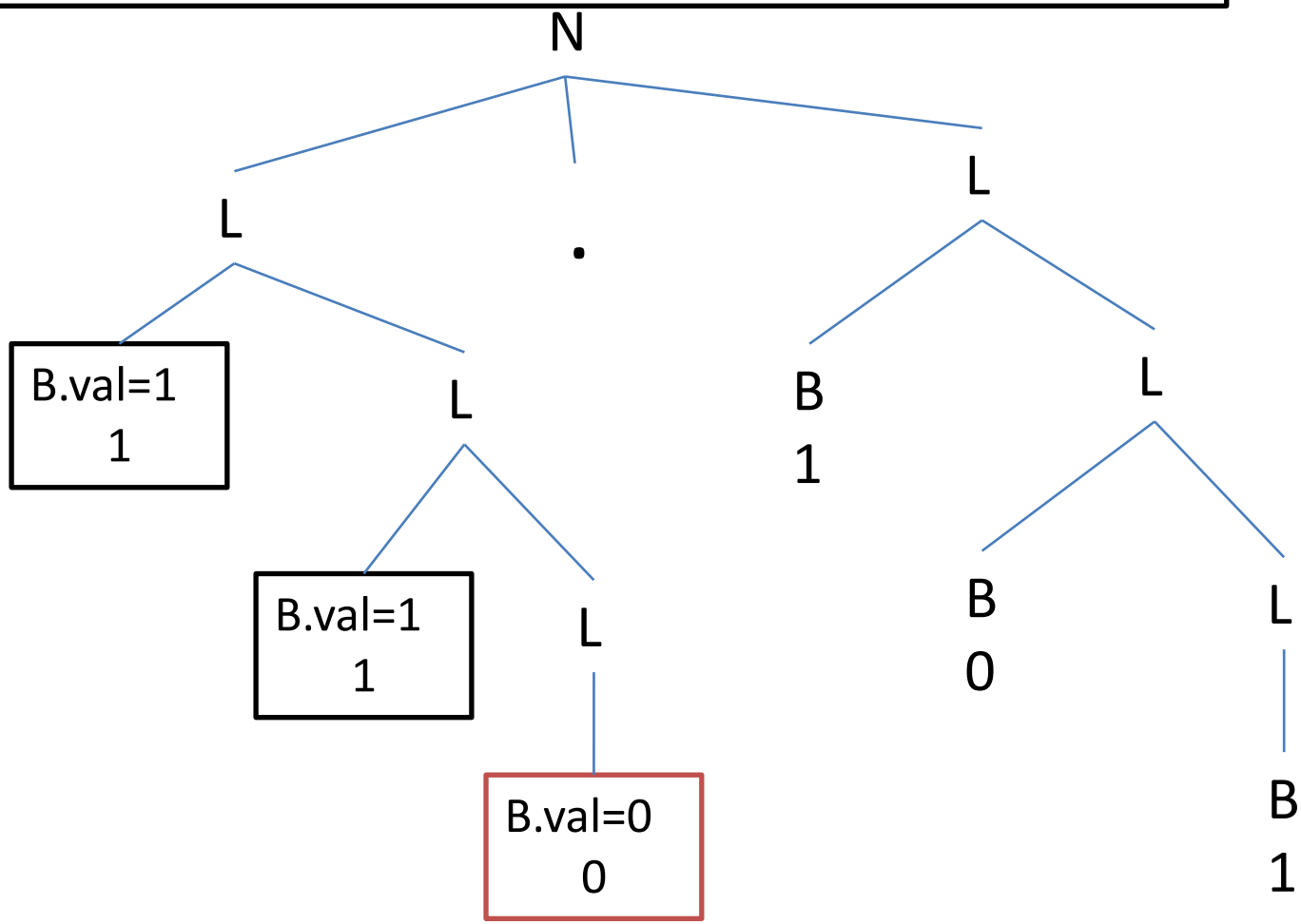
1.	$N \rightarrow L_1.L_2$	$\{N.val = L_1.val + (L_2.val / 2^{L_2.cnt})\}$
2.	$L \rightarrow BL_1$	$\{L.cnt=L_1.cnt+1;\ L.val=L_1.val+(B.val * 2^{L_1.cnt})\}$
3.	$L \rightarrow B$	$\{L.cnt = 1\ ;\ L.val = B.val\}$
4.	$B \rightarrow 0$	$\{B.val = 0\}$
5.	$B \rightarrow 1$	$\{B.val = 1\}$



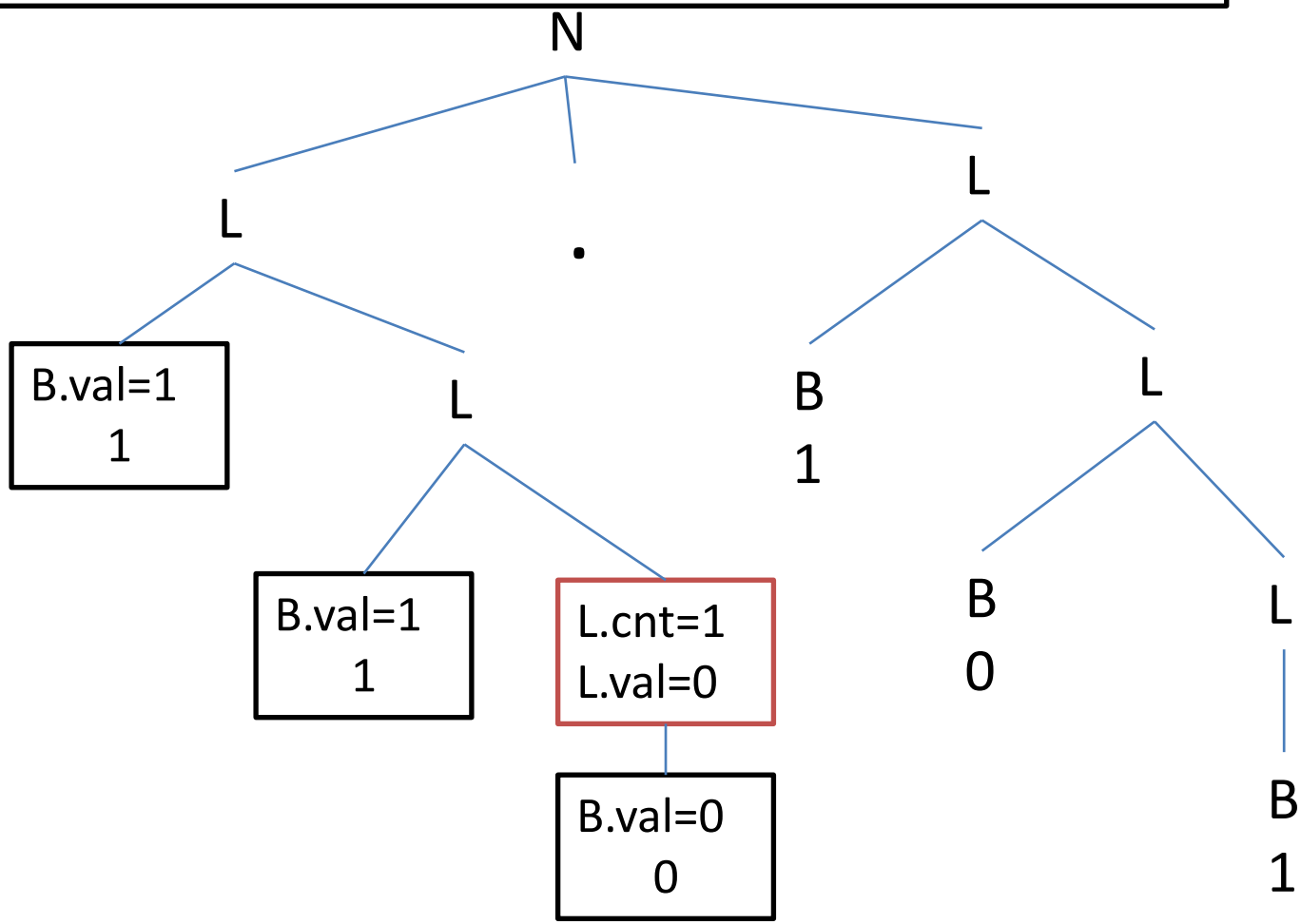
1.	$N \rightarrow L_1.L_2$	$\{N.val = L_1.val + (L_2.val / 2\hat{}L_2.cnt)\}$
2.	$L \rightarrow BL_1$	$\{L.cnt=L_1.cnt+1;\ L.val=L_1.val+(B.val * 2\hat{}L_1.cnt)\}$
3.	$L \rightarrow B$	$\{L.cnt = 1\ ;\ L.val = B.val\}$
4.	$B \rightarrow 0$	$\{B.val = 0\}$
5.	$B \rightarrow 1$	$\{B.val = 1\}$



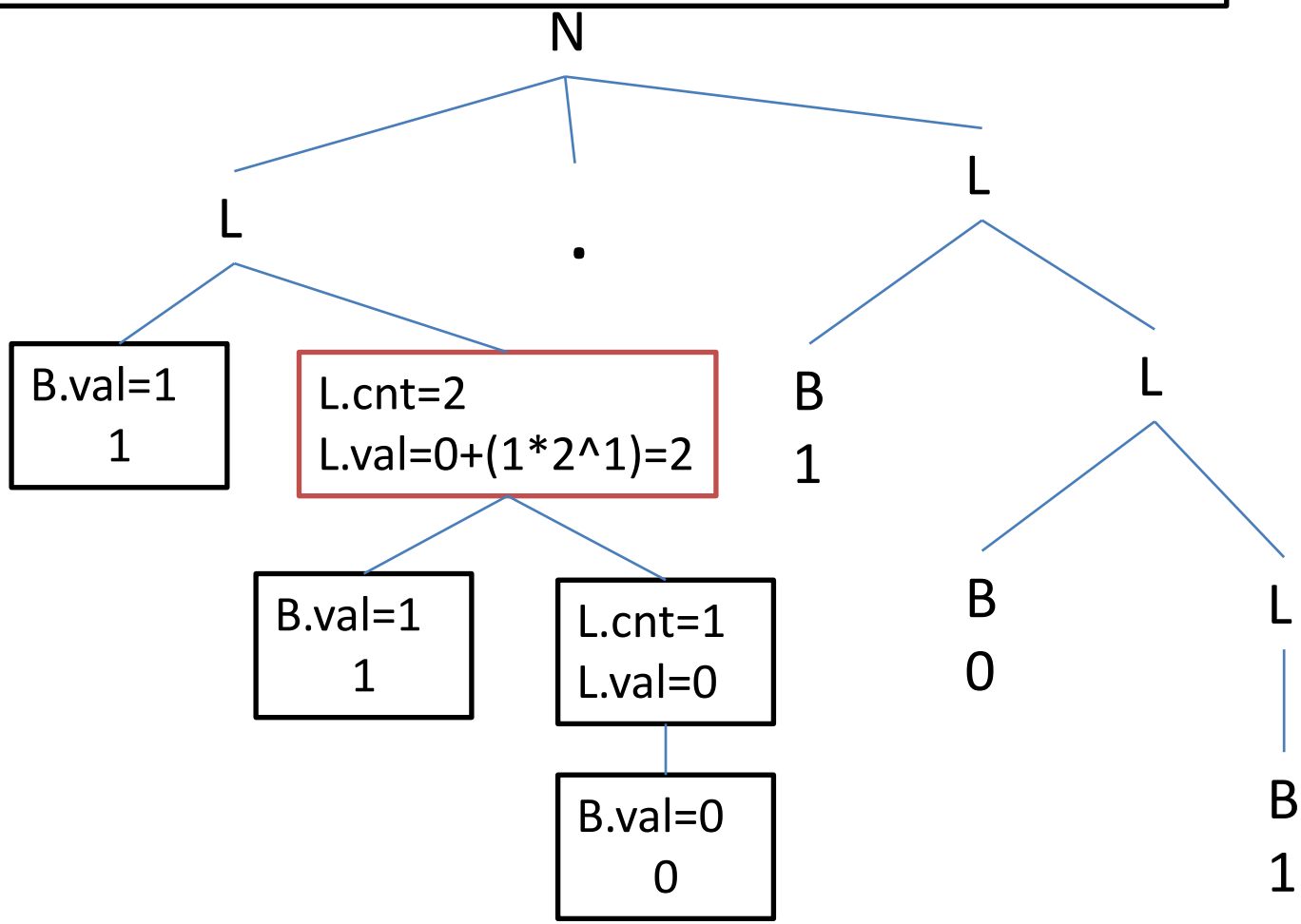
1.	$N \rightarrow L_1.L_2$	$\{N.val = L_1.val + (L_2.val / 2^{L_2.cnt})\}$
2.	$L \rightarrow BL_1$	$\{L.cnt=L_1.cnt+1;\ L.val=L_1.val+(B.val * 2^{L_1.cnt})\}$
3.	$L \rightarrow B$	$\{L.cnt = 1\ ;\ L.val = B.val\}$
4.	$B \rightarrow 0$	$\{B.val = 0\}$
5.	$B \rightarrow 1$	$\{B.val = 1\}$



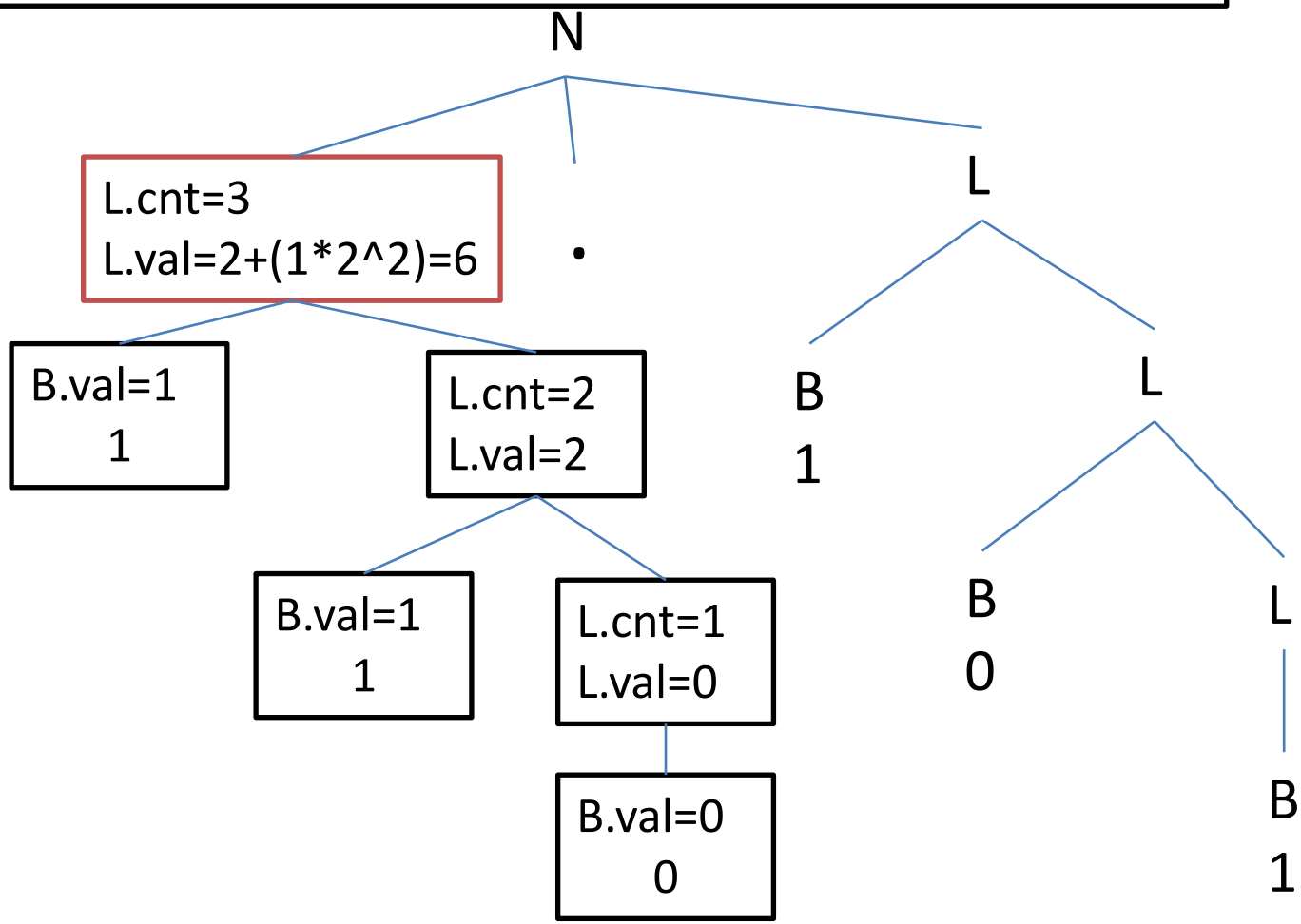
1.	$N \rightarrow L_1.L_2$	$\{N.val = L_1.val + (L_2.val / 2^{L_2.cnt})\}$
2.	$L \rightarrow BL_1$	$\{L.cnt=L_1.cnt+1;\ L.val=L_1.val+(B.val * 2^{L_1.cnt})\}$
3.	$L \rightarrow B$	$\{L.cnt = 1\ ;\ L.val = B.val\}$
4.	$B \rightarrow 0$	$\{B.val = 0\}$
5.	$B \rightarrow 1$	$\{B.val = 1\}$



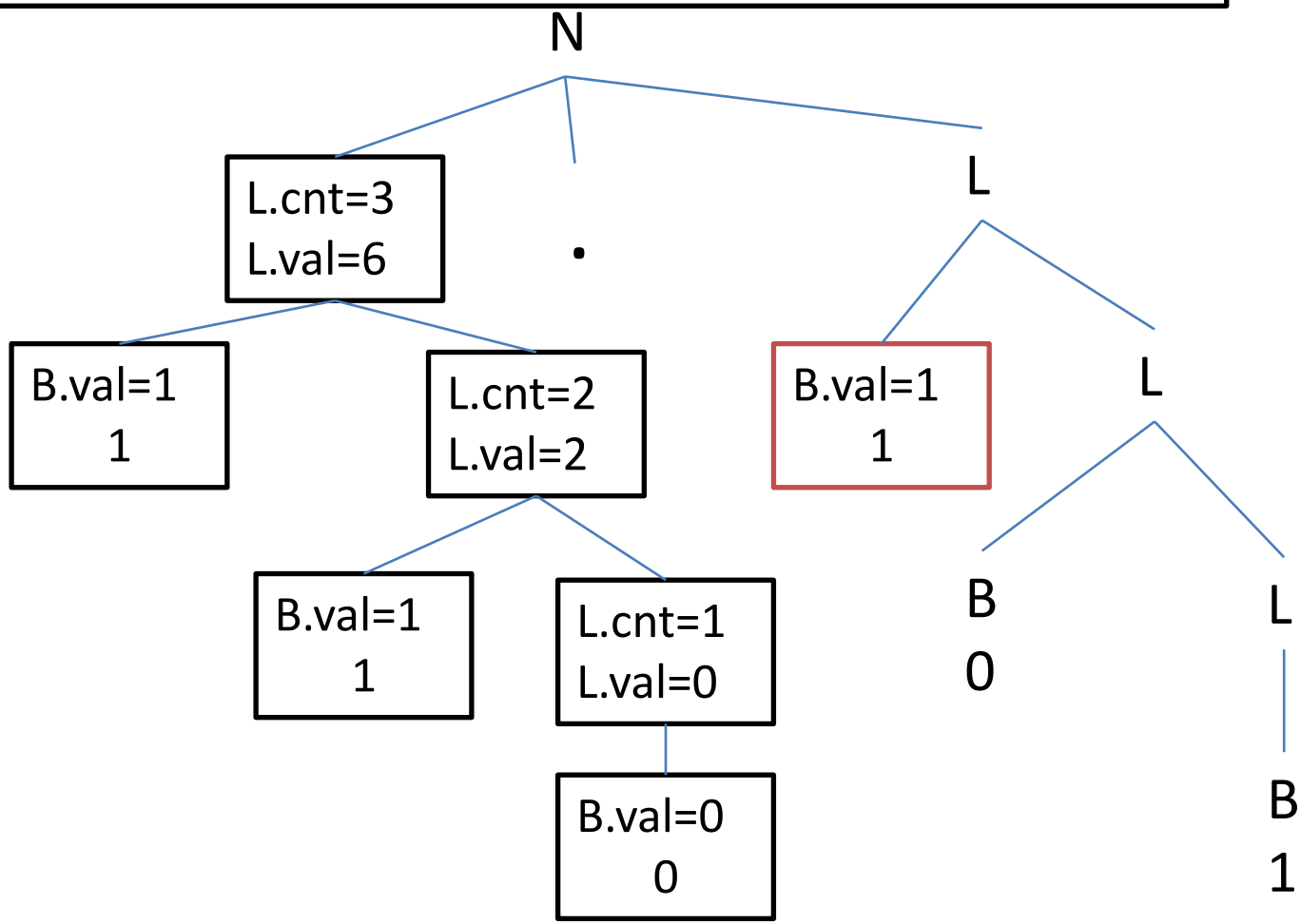
1.	$N \rightarrow L_1.L_2$	$\{N.val = L_1.val + (L_2.val / 2^{L_2.cnt})\}$
2.	$L \rightarrow BL_1$	$\{L.cnt=L_1.cnt+1;\ L.val=L_1.val+(B.val * 2^{L_1.cnt})\}$
3.	$L \rightarrow B$	$\{L.cnt = 1\ ;\ L.val = B.val\}$
4.	$B \rightarrow 0$	$\{B.val = 0\}$
5.	$B \rightarrow 1$	$\{B.val = 1\}$



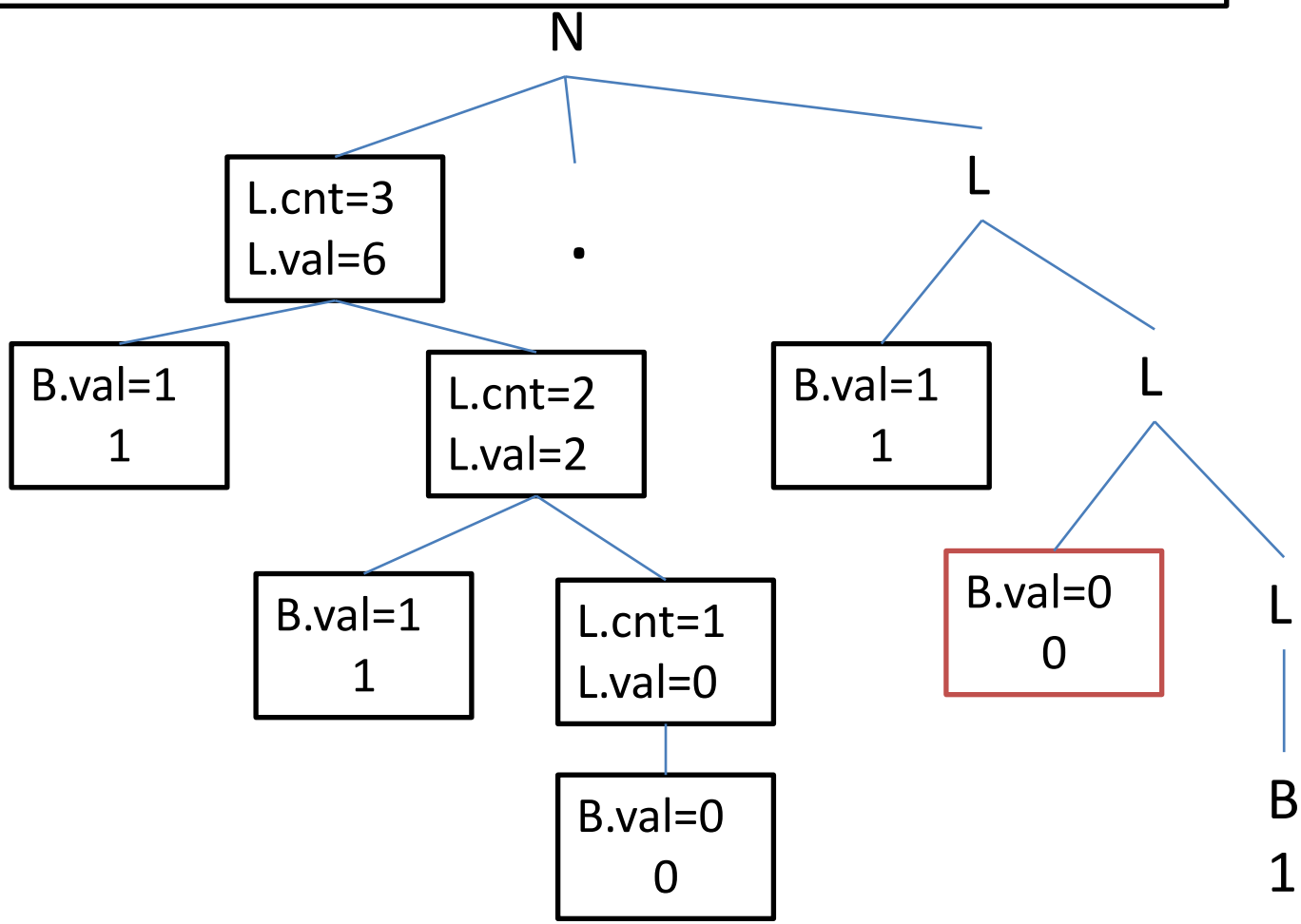
1.	$N \rightarrow L_1.L_2$	$\{N.val = L_1.val + (L_2.val / 2\hat{}L_2.cnt)\}$
2.	$L \rightarrow BL_1$	$\{L.cnt=L_1.cnt+1;\ L.val=L_1.val+(B.val * 2\hat{}L_1.cnt)\}$
3.	$L \rightarrow B$	$\{L.cnt = 1\ ;\ L.val = B.val\}$
4.	$B \rightarrow 0$	$\{B.val = 0\}$
5.	$B \rightarrow 1$	$\{B.val = 1\}$



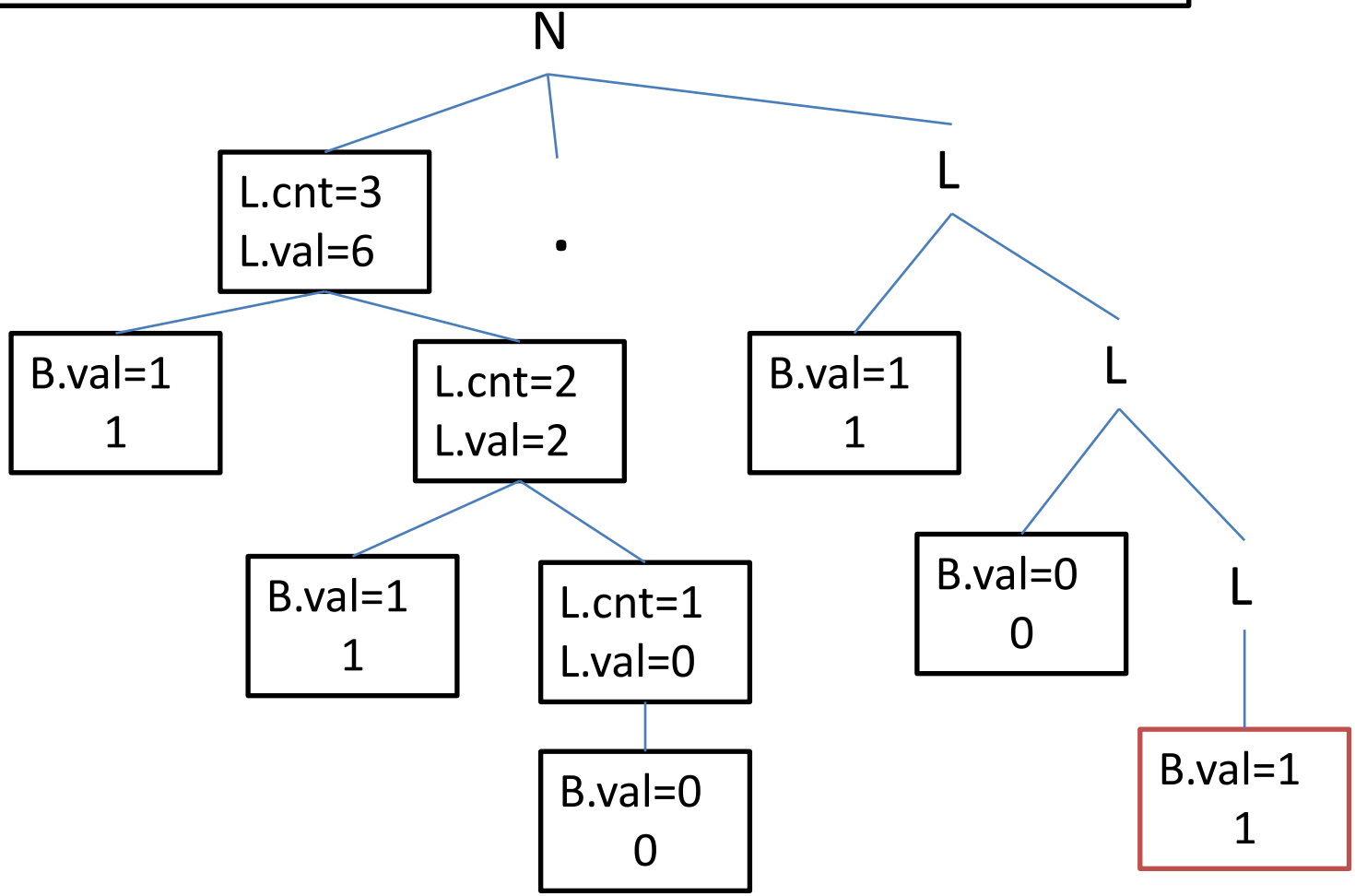
1. $N \rightarrow L_1.L_2$ {N.val = L_1.val + (L_2.val / 2^L_2.cnt)}
2. $L \rightarrow BL_1$ {L.cnt=L_1.cnt+1; L.val=L_1.val+(B.val* 2^L_1.cnt)}
3. $L \rightarrow B$ {L.cnt = 1 ; L.val = B.val}
4. $B \rightarrow 0$ {B.val = 0}
5. $B \rightarrow 1$ {B.val = 1}



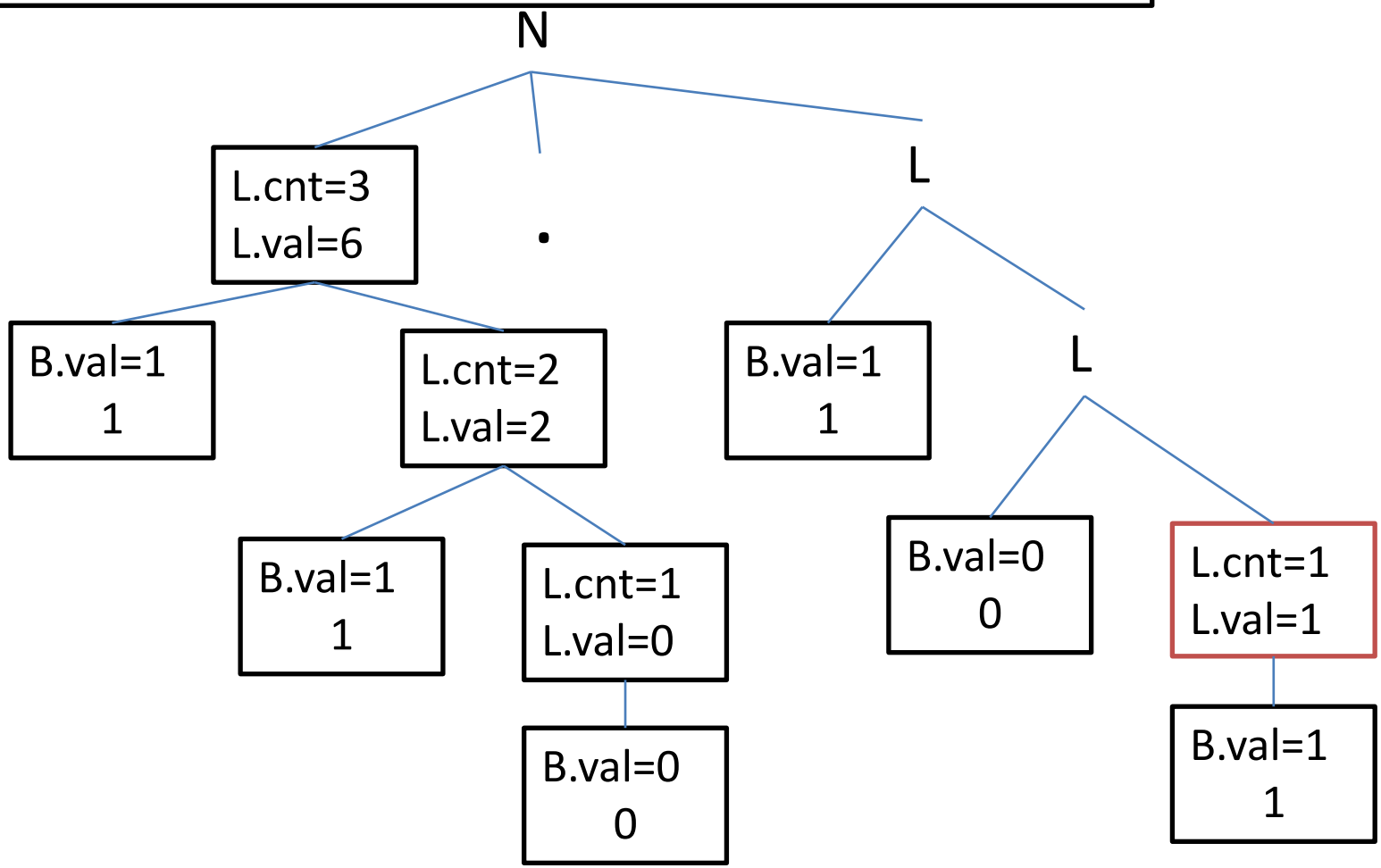
1.	$N \rightarrow L_1.L_2$	$\{N.val = L_1.val + (L_2.val / 2^{L_2.cnt})\}$
2.	$L \rightarrow BL_1$	$\{L.cnt=L_1.cnt+1;\ L.val=L_1.val+(B.val * 2^{L_1.cnt})\}$
3.	$L \rightarrow B$	$\{L.cnt = 1\ ;\ L.val = B.val\}$
4.	$B \rightarrow 0$	$\{B.val = 0\}$
5.	$B \rightarrow 1$	$\{B.val = 1\}$



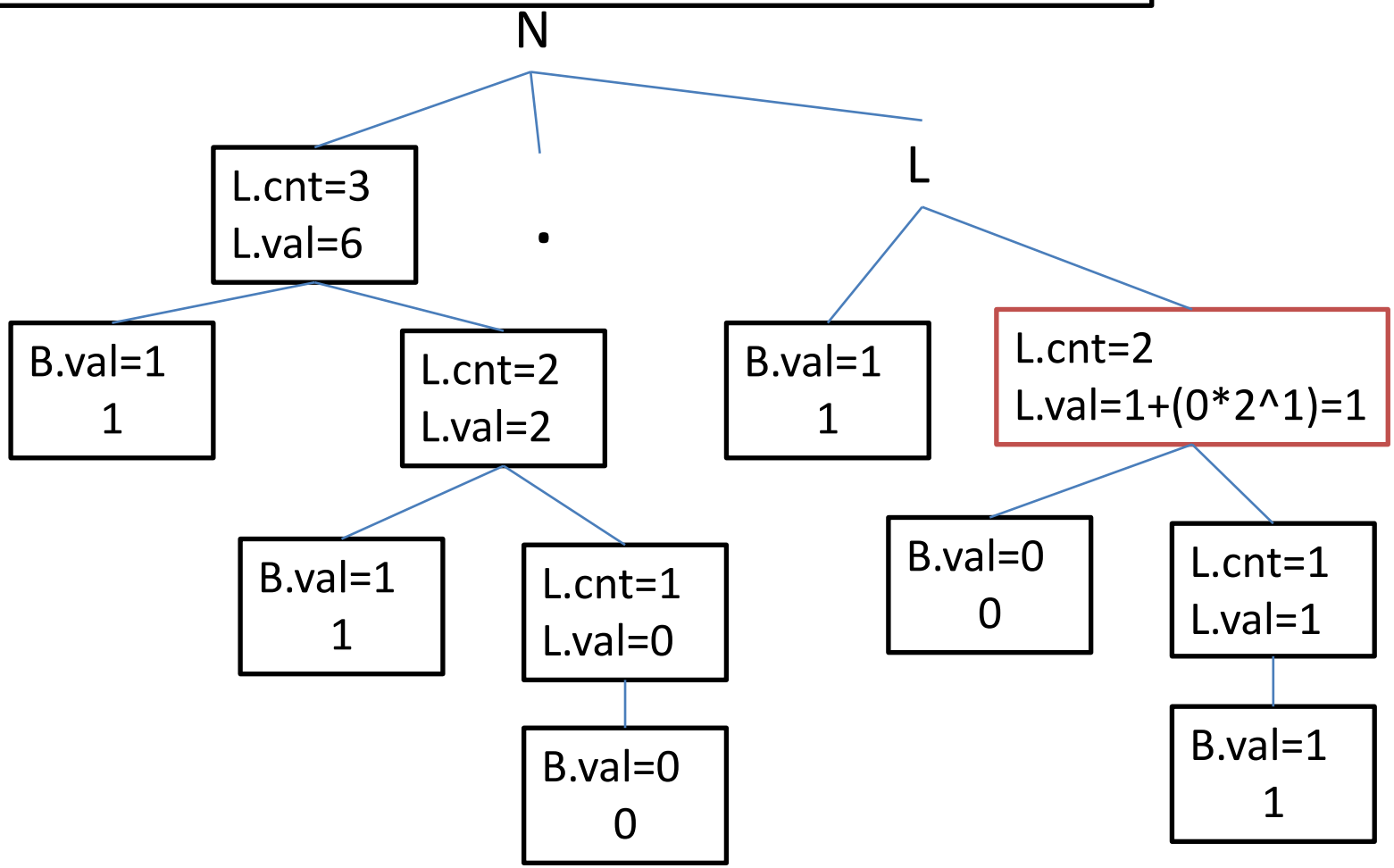
1. $N \rightarrow L_1.L_2$ {N.val = L_1.val + (L_2.val / 2^L_2.cnt)}
2. $L \rightarrow BL_1$ {L.cnt=L_1.cnt+1; L.val=L_1.val+(B.val* 2^L_1.cnt)}
3. $L \rightarrow B$ {L.cnt = 1 ; L.val = B.val}
4. $B \rightarrow 0$ {B.val = 0}
5. $B \rightarrow 1$ {B.val = 1}



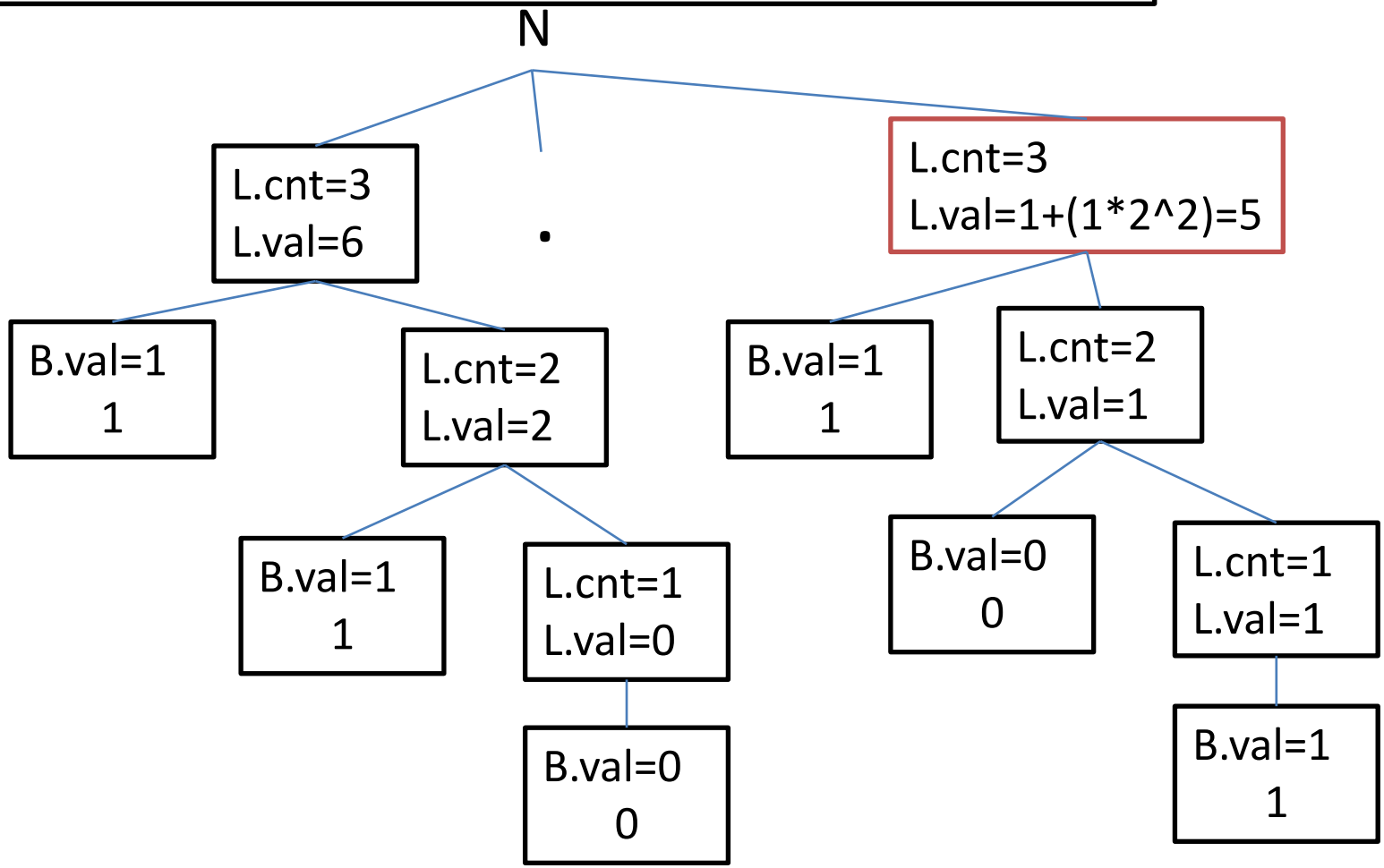
1.	$N \rightarrow L_1.L_2$	$\{N.val = L_1.val + (L_2.val / 2^{\wedge} L_2.cnt)\}$
2.	$L \rightarrow BL_1$	$\{L.cnt=L_1.cnt+1;\ L.val=L_1.val+(B.val * 2^{\wedge} L_1.cnt)\}$
3.	$L \rightarrow B$	$\{L.cnt = 1\ ;\ L.val = B.val\}$
4.	$B \rightarrow 0$	$\{B.val = 0\}$
5.	$B \rightarrow 1$	$\{B.val = 1\}$



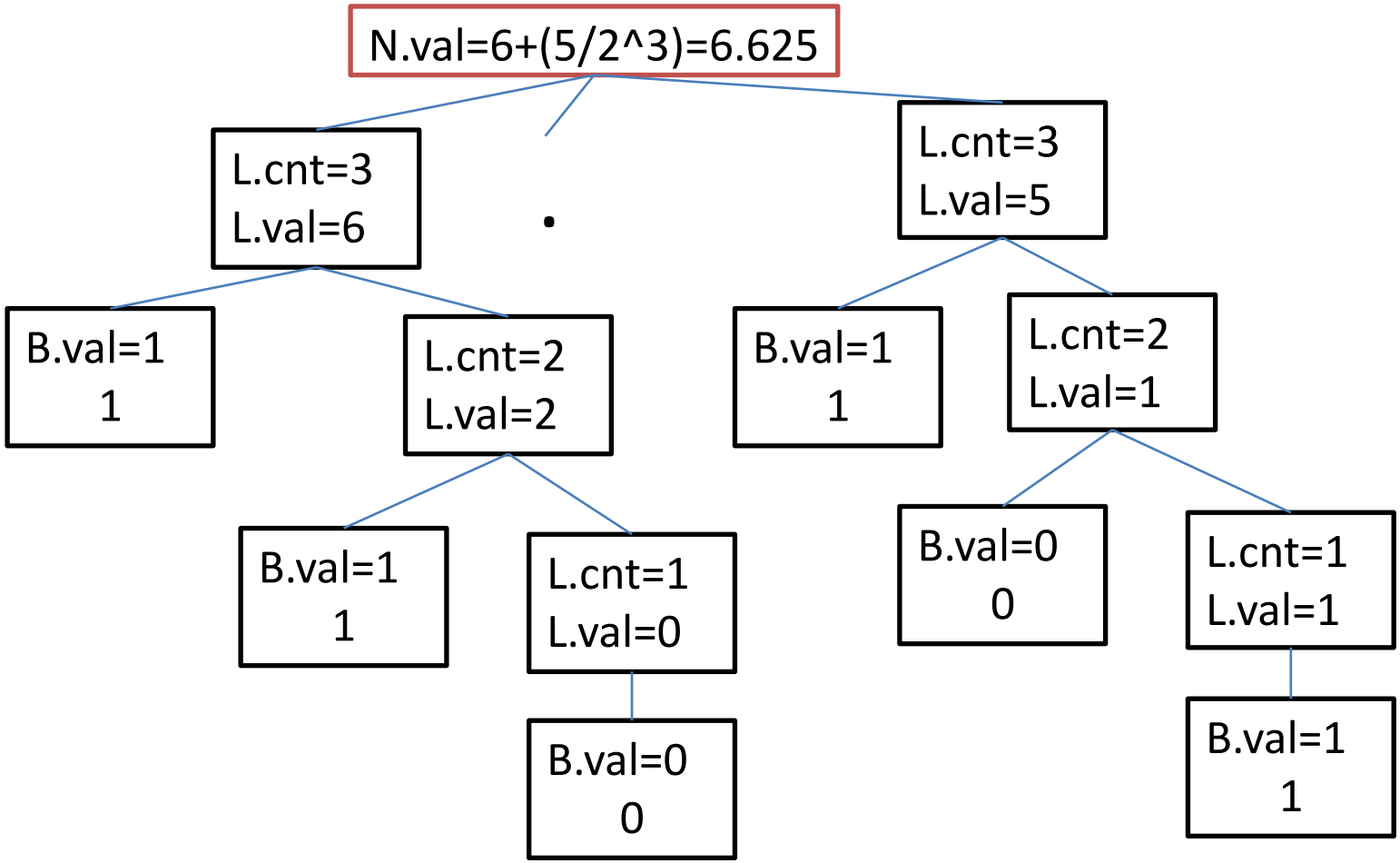
1.	$N \rightarrow L_1.L_2$	$\{N.val = L_1.val + (L_2.val / 2\hat{}L_2.cnt)\}$
2.	$L \rightarrow BL_1$	$\{L.cnt=L_1.cnt+1;\ L.val=L_1.val+(B.val * 2\hat{}L_1.cnt)\}$
3.	$L \rightarrow B$	$\{L.cnt = 1\ ;\ L.val = B.val\}$
4.	$B \rightarrow 0$	$\{B.val = 0\}$
5.	$B \rightarrow 1$	$\{B.val = 1\}$



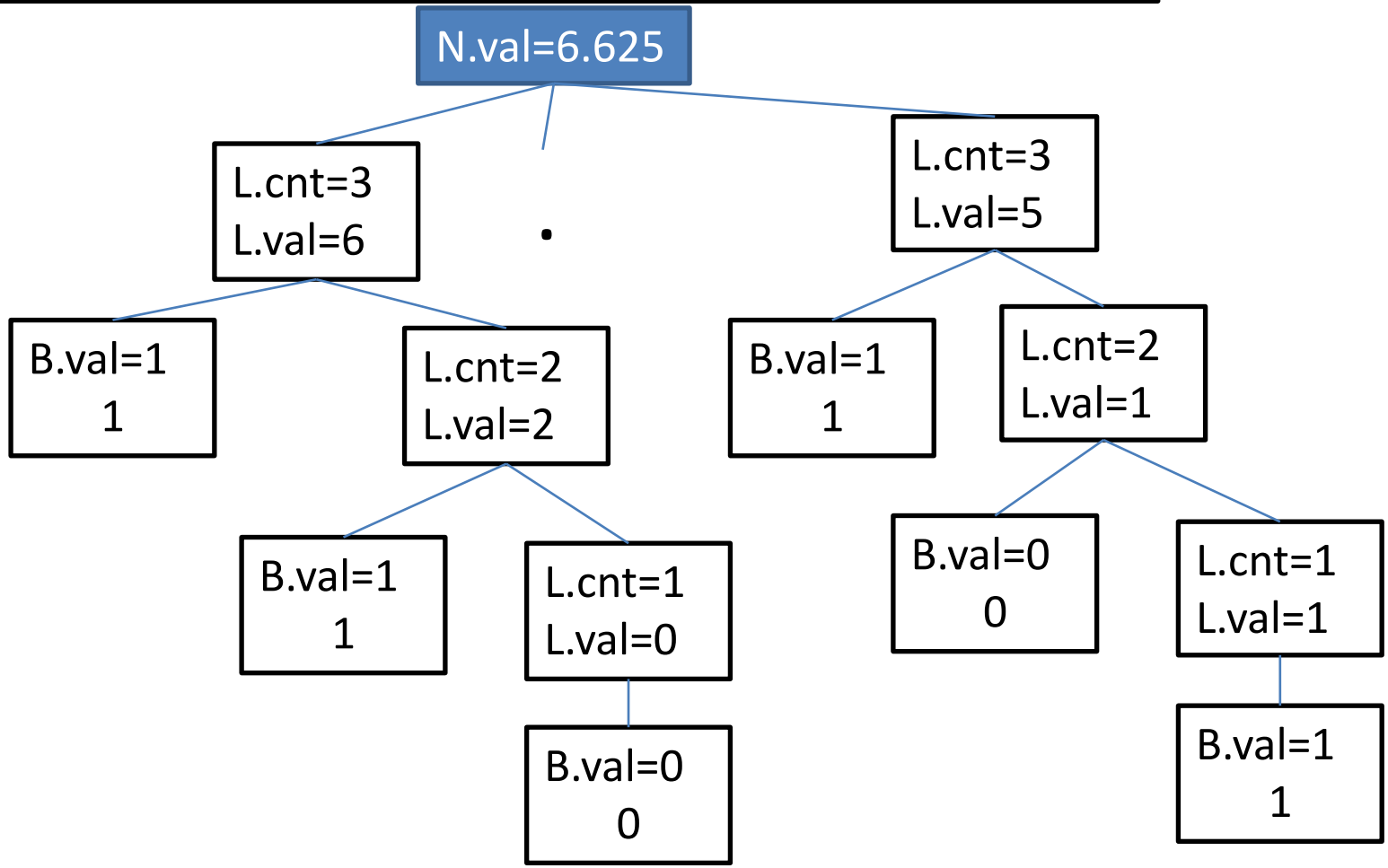
1.	$N \rightarrow L_1.L_2$	$\{N.val = L_1.val + (L_2.val / 2\hat{}L_2.cnt)\}$
2.	$L \rightarrow BL_1$	$\{L.cnt=L_1.cnt+1;\ L.val=L_1.val+(B.val * 2\hat{}L_1.cnt)\}$
3.	$L \rightarrow B$	$\{L.cnt = 1\ ;\ L.val = B.val\}$
4.	$B \rightarrow 0$	$\{B.val = 0\}$
5.	$B \rightarrow 1$	$\{B.val = 1\}$



1.	$N \rightarrow L_1.L_2$	$\{N.val = L_1.val + (L_2.val / 2^{\wedge} L_2.cnt)\}$
2.	$L \rightarrow BL_1$	$\{L.cnt=L_1.cnt+1;\ L.val=L_1.val+(B.val * 2^{\wedge} L_1.cnt)\}$
3.	$L \rightarrow B$	$\{L.cnt = 1\ ;\ L.val = B.val\}$
4.	$B \rightarrow 0$	$\{B.val = 0\}$
5.	$B \rightarrow 1$	$\{B.val = 1\}$



1. $N \rightarrow L_1.L_2$ {N.val = L_1.val + (L_2.val / 2^L_2.cnt)}
2. $L \rightarrow BL_1$ {L.cnt=L_1.cnt+1; L.val=L_1.val+(B.val* 2^L_1.cnt)}
3. $L \rightarrow B$ {L.cnt = 1 ; L.val = B.val}
4. $B \rightarrow 0$ {B.val = 0}
5. $B \rightarrow 1$ {B.val = 1}



Example 2 (second method)

- Write an attribute grammar for the evaluation of a real number from its bit-string representation.

- Example: $(\mathbf{110.101})_2 = (\mathbf{6.625})_{10}$

110	.	101
110 → **6**		(decimal value)/(2^ no. of bits) = 5 / 2^3 = 5 / 8 = **0.625**

Example 2 (second method)

- Write an attribute grammar for the evaluation of a real number from its bit-string representation.

$N \rightarrow L \,.\, L$

$L \rightarrow LB \mid B$

$B \rightarrow 0 \mid 1$

Example 2 (second method)

Binary to Decimal

N→L.L

L→LB | B

B →0|1

- AS(N)={val↑:real}
- AS(L) = AS(B) ={cnt↑:integer, val↑:real}

1. N → $L_1.L_2$ {N.val = L_1.val + (L_2.val / 2^L_2.cnt)}
2. L → L_1B {L.cnt=L_1.cnt+1; L.val=L_1.val*2 + B.val}
3. L → B {L.cnt = 1 ; L.val = B.val}
4. B → 0 {B.val = 0}
5. B → 1 {B.val = 1}

Example 2 (second method)

Octal to Decimal

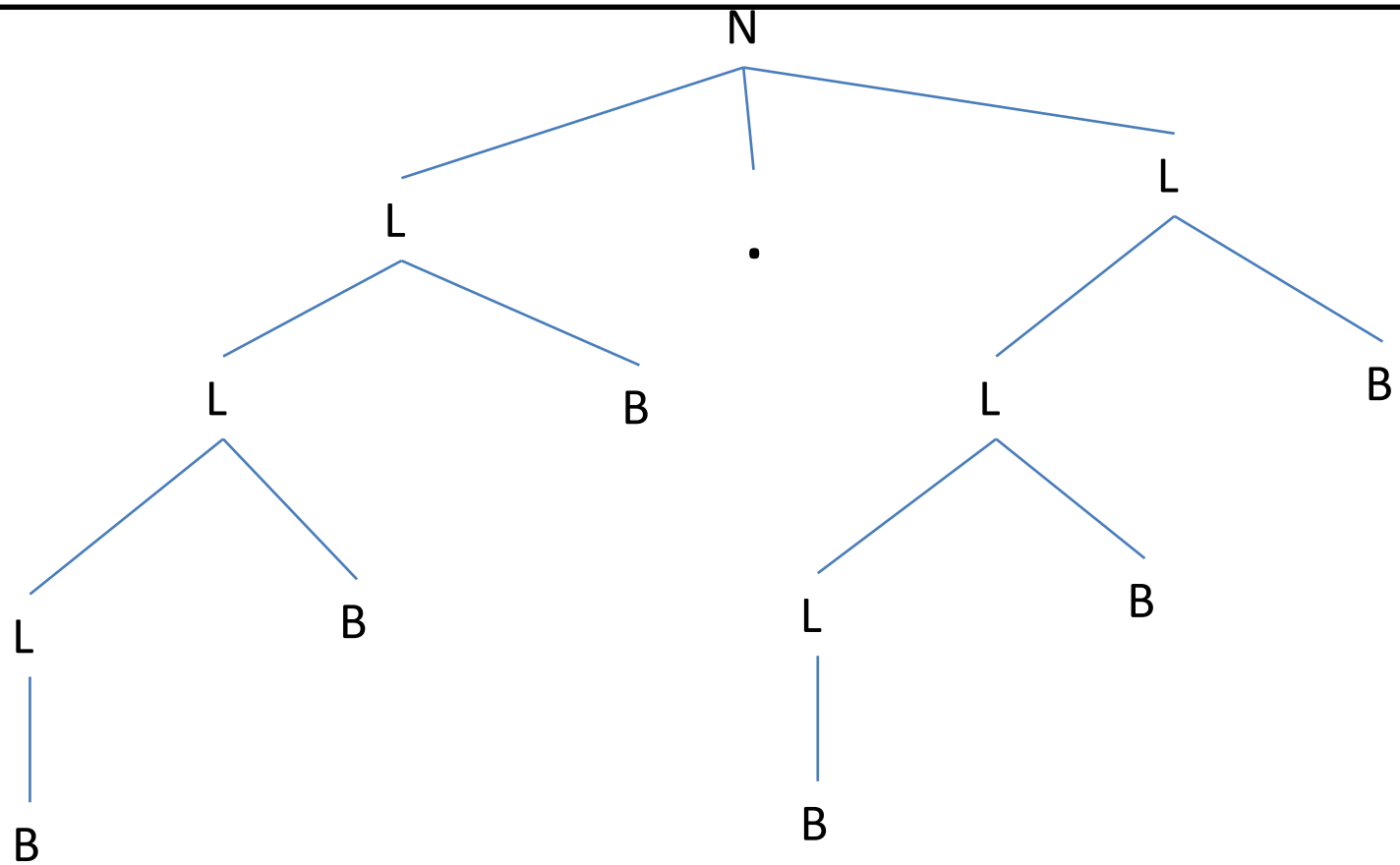
1. $N \rightarrow L_1.L_2$ $\{N.val = L_1.val + (L_2.val / 8\hat{}L_2.cnt)\}$
2. $L \rightarrow L_1B$ $\{L.cnt=L_1.cnt+1;\ L.val=L_1.val*8 + B.val\}$
3. $L \rightarrow B$ $\{L.cnt = 1\ ;\ L.val = B.val\}$
4. $B \rightarrow 0$ $\{B.val = 0\}$
5. $B \rightarrow 1$ $\{B.val = 1\}$
6. $B \rightarrow 2$ $\{B.val = 2\}$
7. $B \rightarrow 3$ $\{B.val = 3\}$
8. $B \rightarrow 4$ $\{B.val = 4\}$
9. $B \rightarrow 5$ $\{B.val = 5\}$
10. $B \rightarrow 6$ $\{B.val = 6\}$
11. $B \rightarrow 7$ $\{B.val = 7\}$

Example 2 (second method)

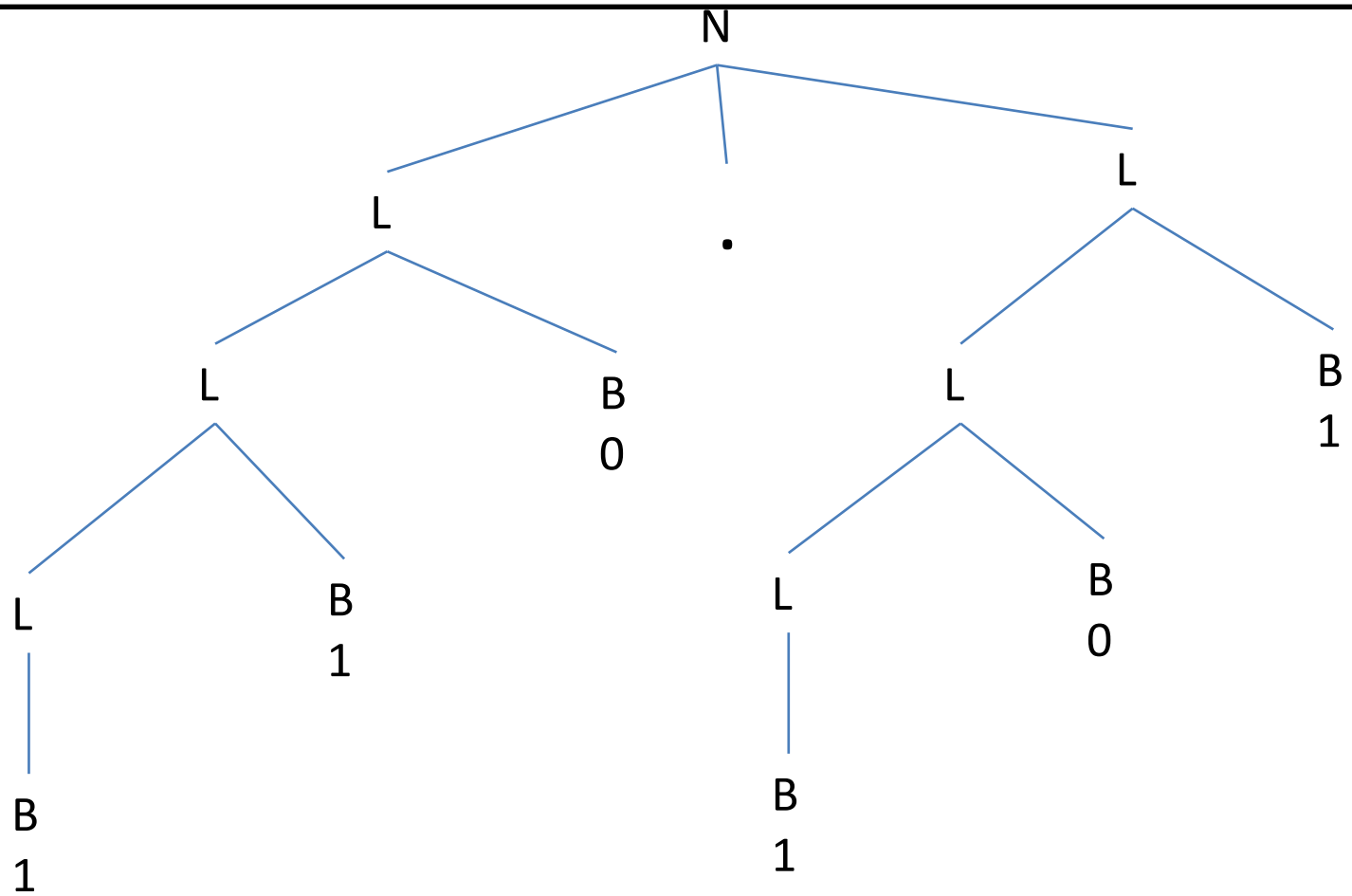
Hexadecimal to Decimal

1. $N \rightarrow L_1.L_2$ {$N.val = L_1.val + (L_2.val / 16\hat{}L_2.cnt)$}
2. $L \rightarrow L_1B$ {$L.cnt=L_1.cnt+1$; $L.val=L_1.val*16 + B.val$}
3. $L \rightarrow B$ {$L.cnt = 1$; $L.val = B.val$}
4. $B \rightarrow 0$ {$B.val = 0$}
5. $B \rightarrow 1$ {$B.val = 1$}
6. $B \rightarrow 2$ {$B.val = 2$}
7. $B \rightarrow 3$ {$B.val = 3$}
8. $B \rightarrow 4$ {$B.val = 4$}
9. $B \rightarrow 5$ {$B.val = 5$}
10. $B \rightarrow 6$ {$B.val = 6$}
11. $B \rightarrow 7$ {$B.val = 7$}
12. $B \rightarrow 8$ {$B.val = 8$}
13. $B \rightarrow 9$ {$B.val = 9$}
14. $B \rightarrow a$ {$B.val = 10$}
15. $B \rightarrow b$ {$B.val = 11$}
16. $B \rightarrow c$ {$B.val = 12$}
17. $B \rightarrow d$ {$B.val = 13$}
18. $B \rightarrow e$ {$B.val = 14$}
19. $B \rightarrow f$ {$B.val = 15$}

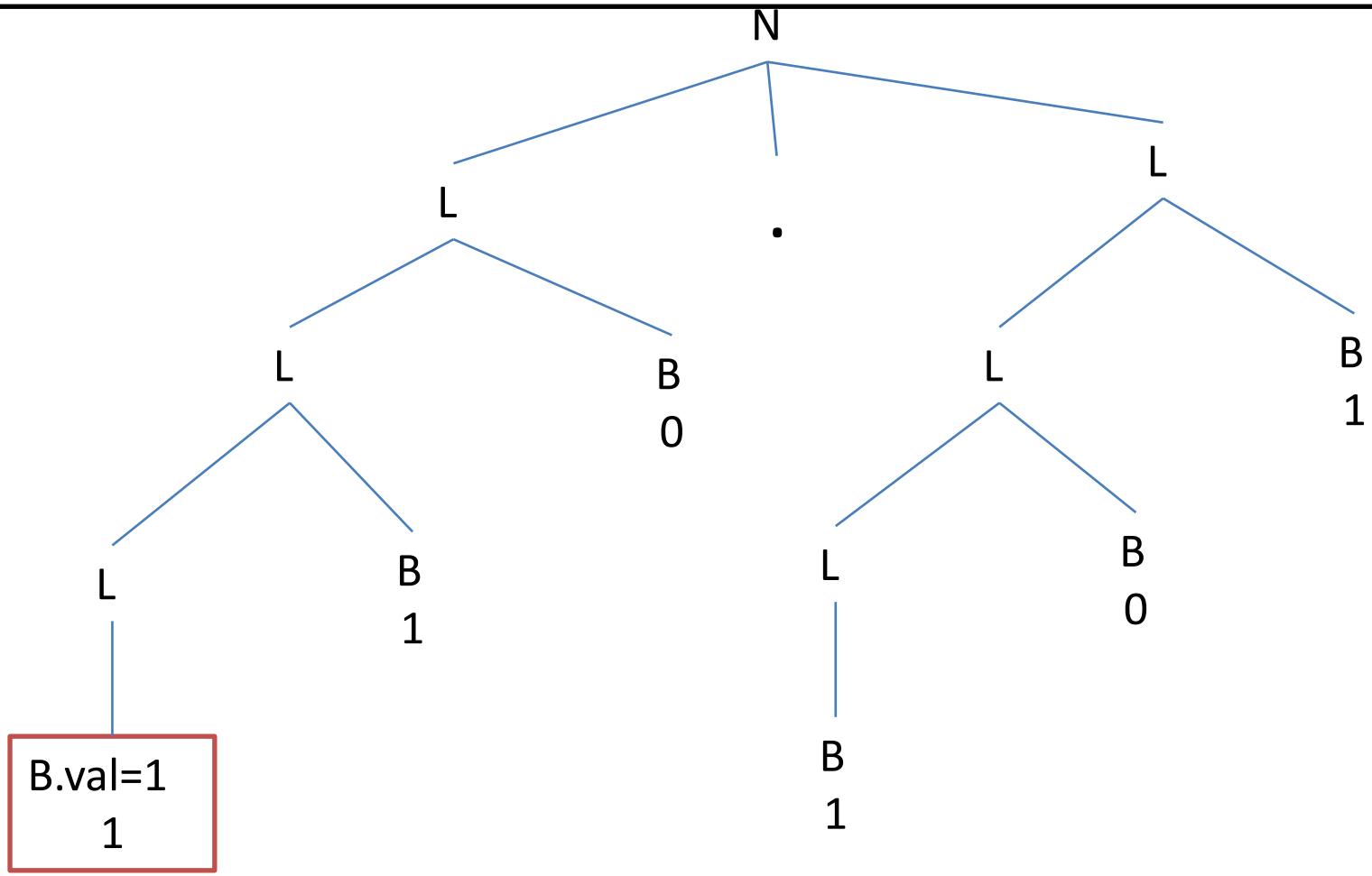
1.	$N \rightarrow L_1.L_2$	$\{N.val = L_1.val + (L_2.val / 2\hat{}L_2.cnt)\}$
2.	$L \rightarrow L_1B$	$\{L.cnt=L_1.cnt+1;\ L.val=L_1.val*2 + B.val\}$
3.	$L \rightarrow B$	$\{L.cnt = 1\ ;\ L.val = B.val\}$
4.	$B \rightarrow 0$	$\{B.val = 0\}$
5.	$B \rightarrow 1$	$\{B.val = 1\}$



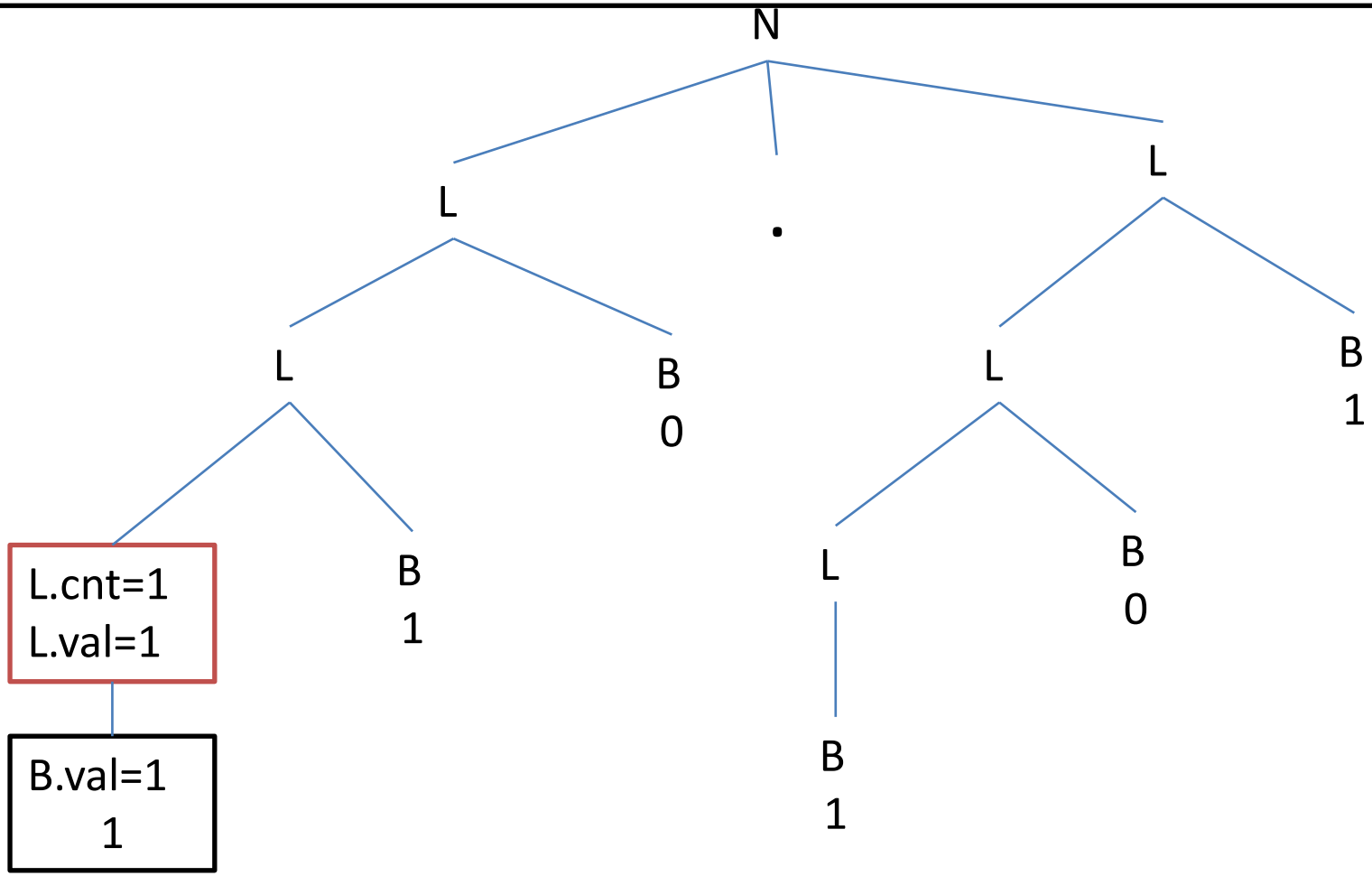
1.	$N \rightarrow L_1.L_2$	$\{N.val = L_1.val + (L_2.val / 2^{\wedge} L_2.cnt)\}$
2.	$L \rightarrow L_1 B$	$\{L.cnt=L_1.cnt+1;\ L.val=L_1.val*2 + B.val\}$
3.	$L \rightarrow B$	$\{L.cnt = 1\ ;\ L.val = B.val\}$
4.	$B \rightarrow 0$	$\{B.val = 0\}$
5.	$B \rightarrow 1$	$\{B.val = 1\}$



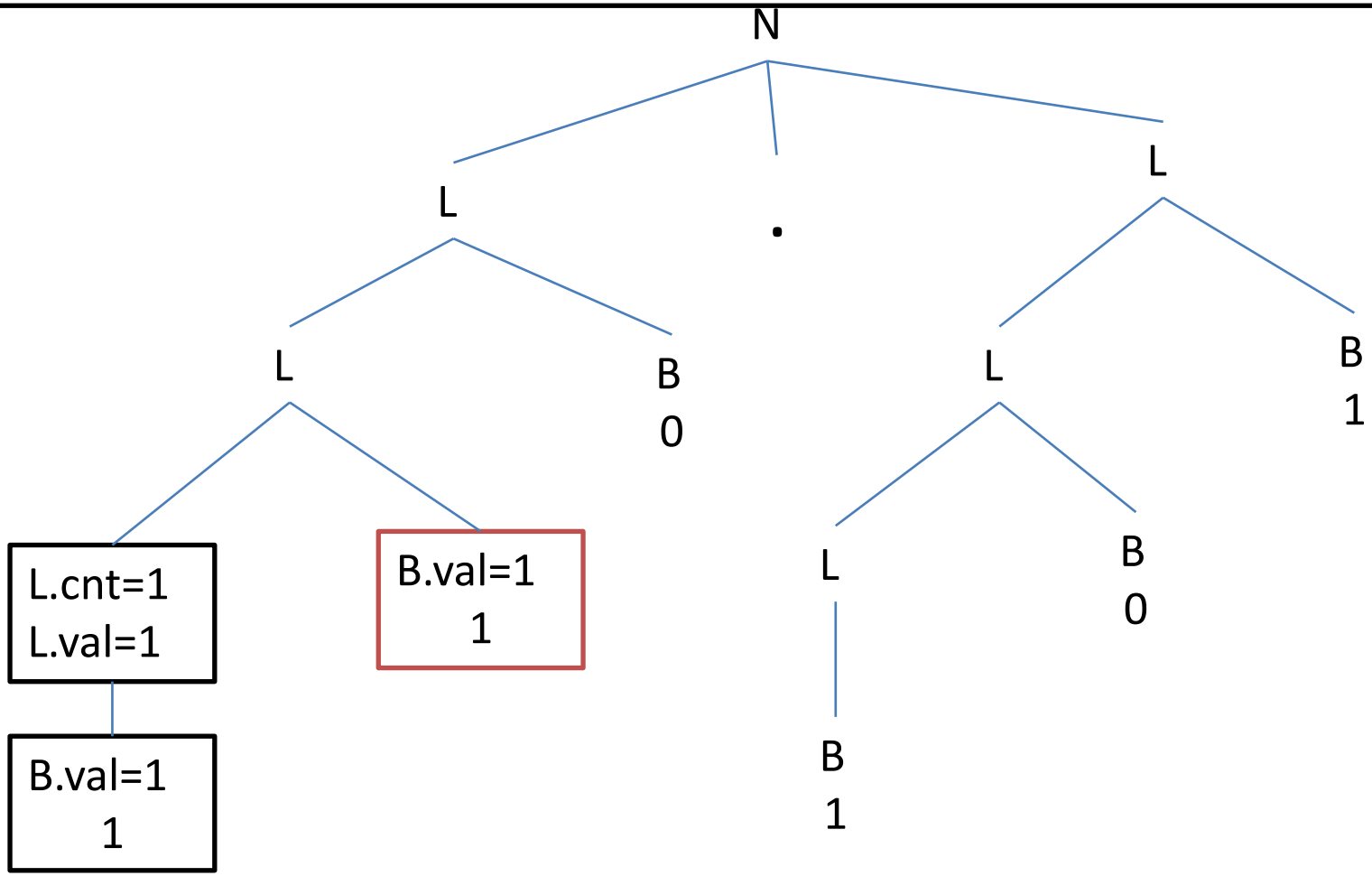
1.	$N \rightarrow L_1.L_2$	$\{N.val = L_1.val + (L_2.val / 2^{L_2.cnt})\}$
2.	$L \rightarrow L_1B$	$\{L.cnt=L_1.cnt+1;\ L.val=L_1.val*2 + B.val\}$
3.	$L \rightarrow B$	$\{L.cnt = 1\ ;\ L.val = B.val\}$
4.	$B \rightarrow 0$	$\{B.val = 0\}$
5.	$B \rightarrow 1$	$\{B.val = 1\}$



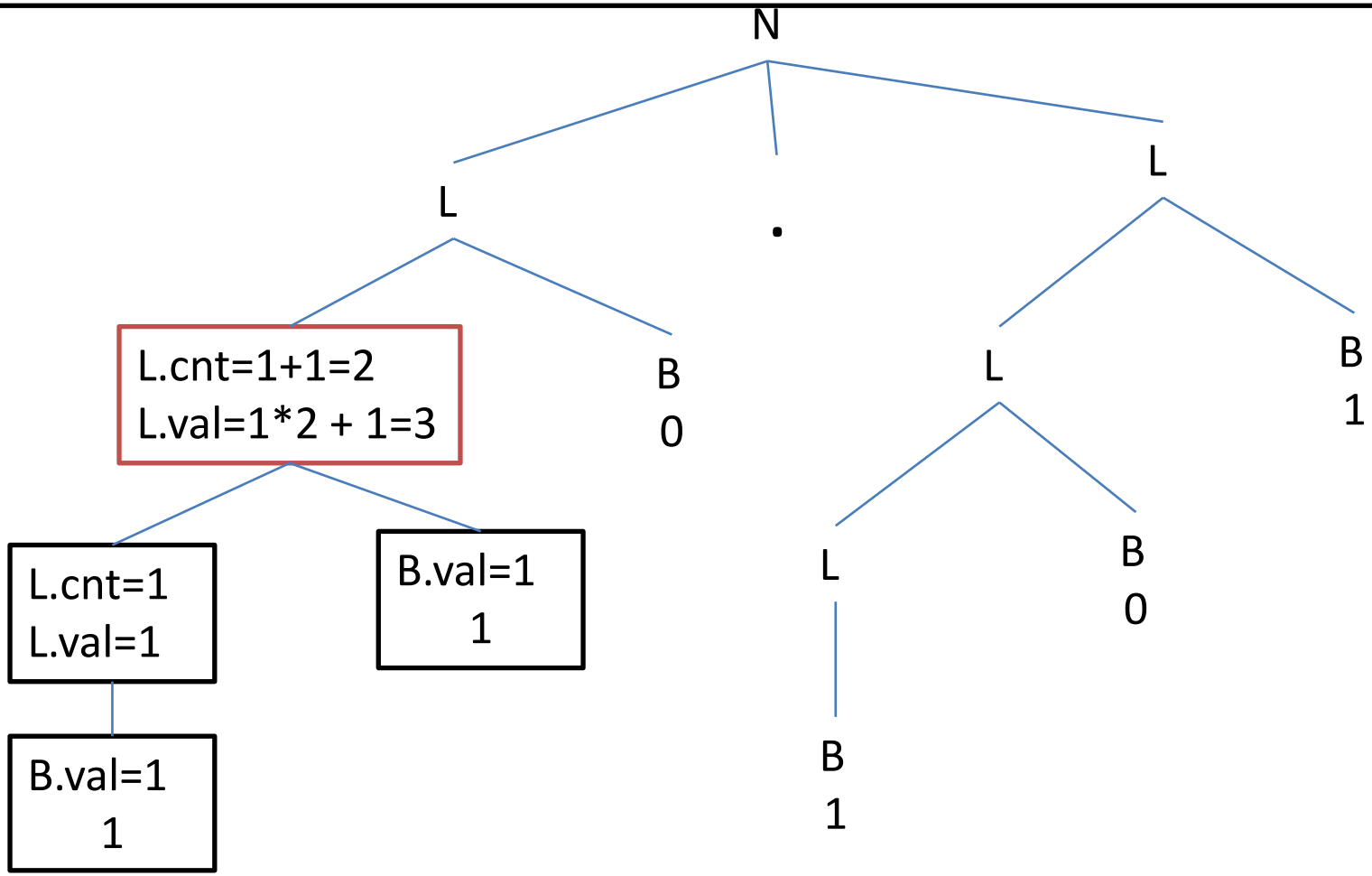
1.	$N \rightarrow L_1.L_2$	$\{N.val = L_1.val + (L_2.val / 2^{\wedge} L_2.cnt)\}$
2.	$L \rightarrow L_1 B$	$\{L.cnt=L_1.cnt+1;\ L.val=L_1.val*2 + B.val\}$
3.	$L \rightarrow B$	$\{L.cnt = 1\ ;\ L.val = B.val\}$
4.	$B \rightarrow 0$	$\{B.val = 0\}$
5.	$B \rightarrow 1$	$\{B.val = 1\}$



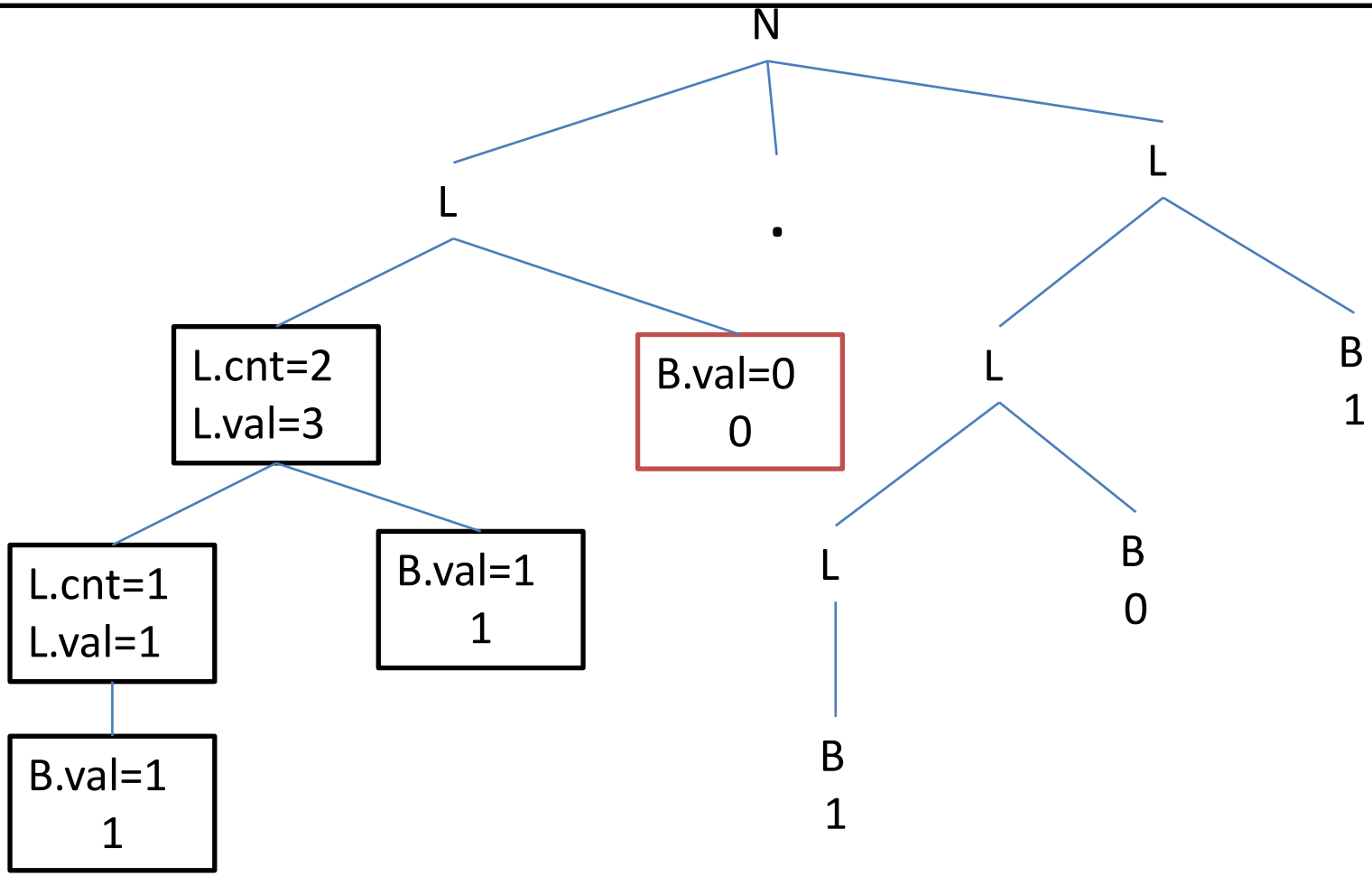
1. $N \rightarrow L_1.L_2$ {$N.val = L_1.val + (L_2.val / 2^{L_2.cnt})$}
2. $L \rightarrow L_1B$ {$L.cnt=L_1.cnt+1$; $L.val=L_1.val*2 + B.val$}
3. $L \rightarrow B$ {$L.cnt = 1$; $L.val = B.val$}
4. $B \rightarrow 0$ {$B.val = 0$}
5. $B \rightarrow 1$ {$B.val = 1$}



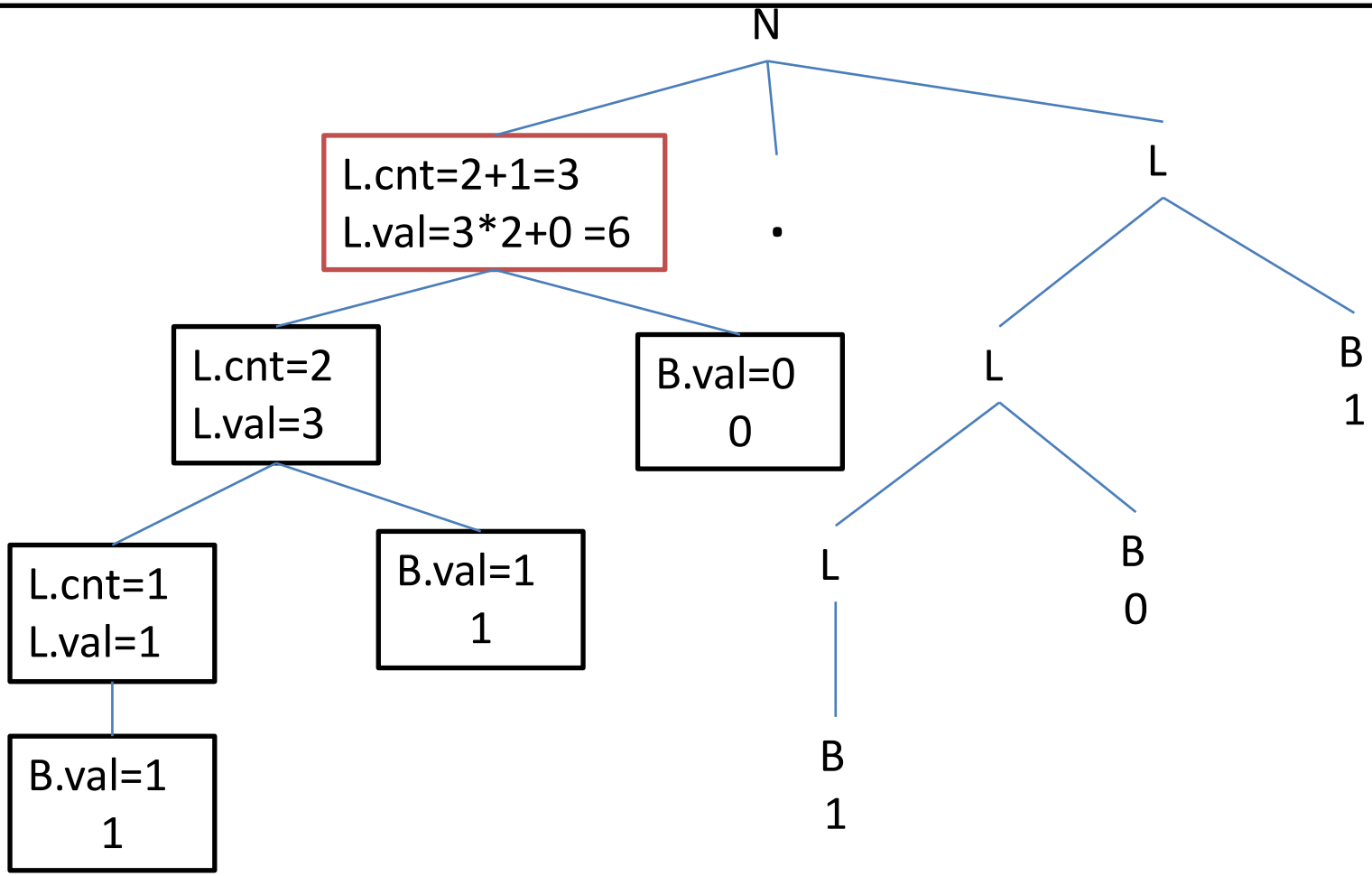
1. $N \rightarrow L_1.L_2$ {N.val = L_1.val + (L_2.val / 2^L_2.cnt)}
2. $L \rightarrow L_1B$ {L.cnt=L_1.cnt+1; L.val=L_1.val*2 + B.val}
3. $L \rightarrow B$ {L.cnt = 1 ; L.val = B.val}
4. $B \rightarrow 0$ {B.val = 0}
5. $B \rightarrow 1$ {B.val = 1}



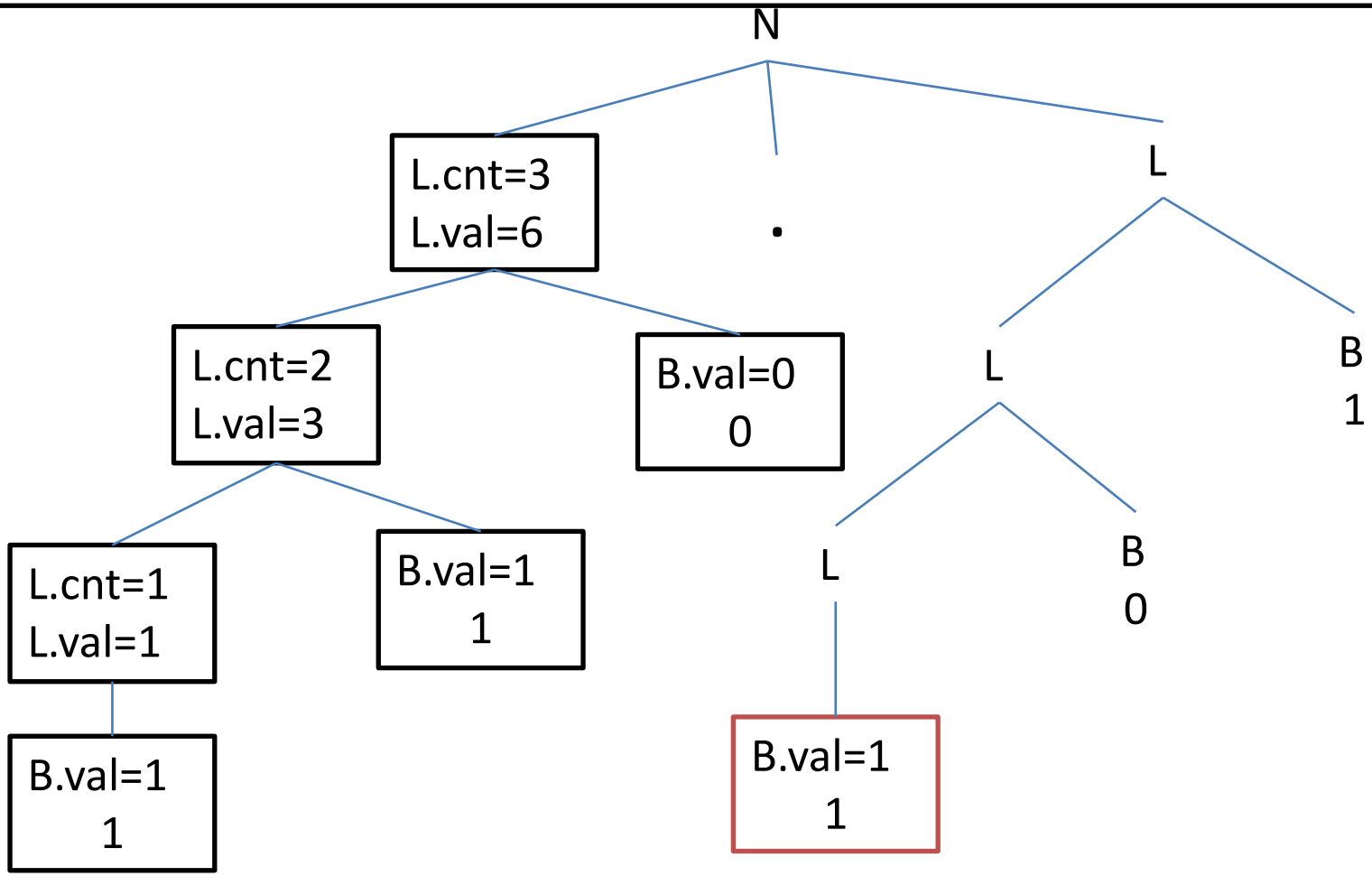
1.	$N \rightarrow L_1.L_2$	$\{N.val = L_1.val + (L_2.val / 2^{\wedge} L_2.cnt)\}$
2.	$L \rightarrow L_1 B$	$\{L.cnt=L_1.cnt+1;\ L.val=L_1.val*2 + B.val\}$
3.	$L \rightarrow B$	$\{L.cnt = 1\ ;\ L.val = B.val\}$
4.	$B \rightarrow 0$	$\{B.val = 0\}$
5.	$B \rightarrow 1$	$\{B.val = 1\}$



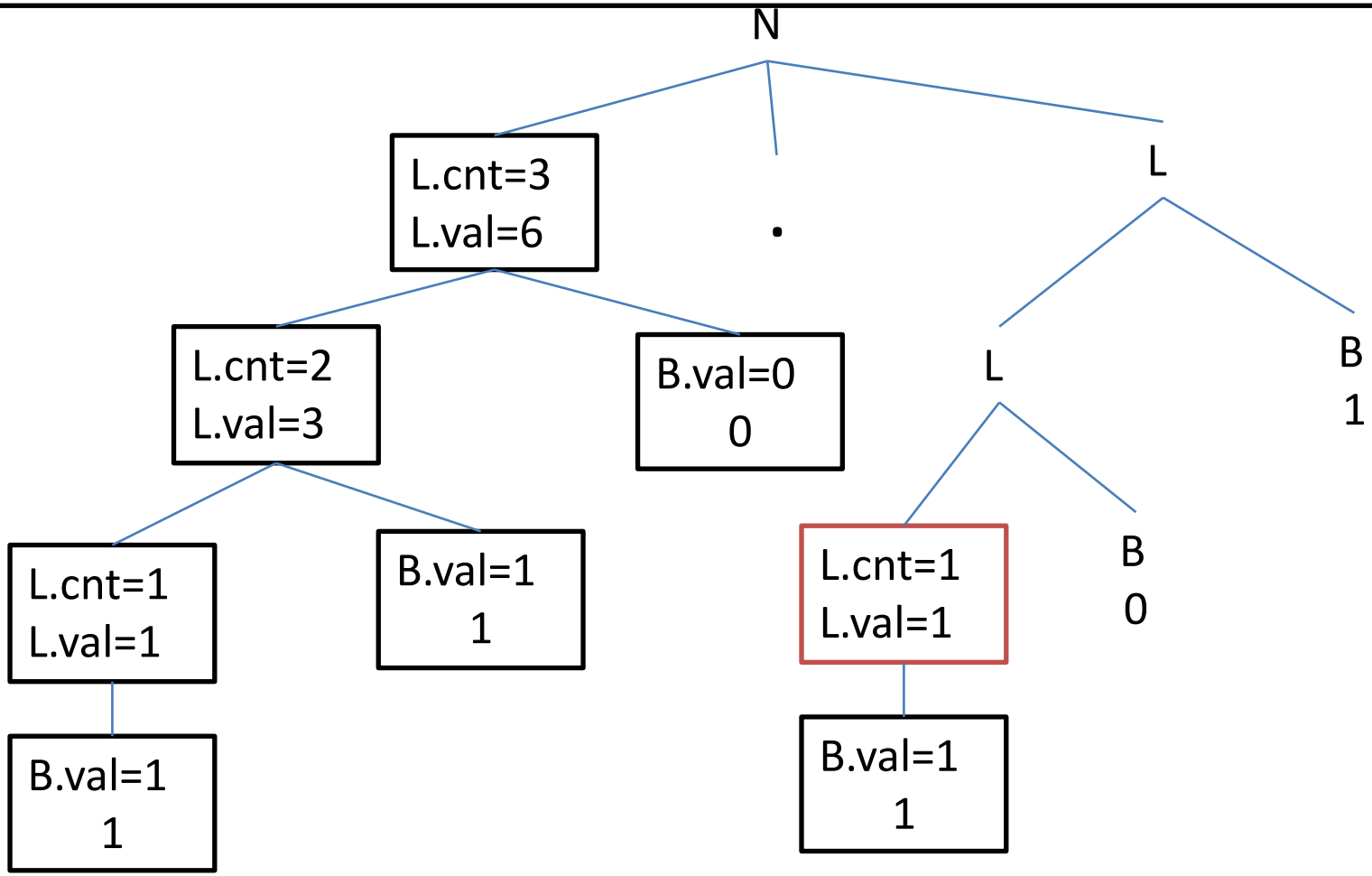
1. $N \rightarrow L_1.L_2$ {N.val = L_1.val + (L_2.val / 2^L_2.cnt)}
2. $L \rightarrow L_1B$ {L.cnt=L_1.cnt+1; L.val=L_1.val*2 + B.val}
3. $L \rightarrow B$ {L.cnt = 1 ; L.val = B.val}
4. $B \rightarrow 0$ {B.val = 0}
5. $B \rightarrow 1$ {B.val = 1}



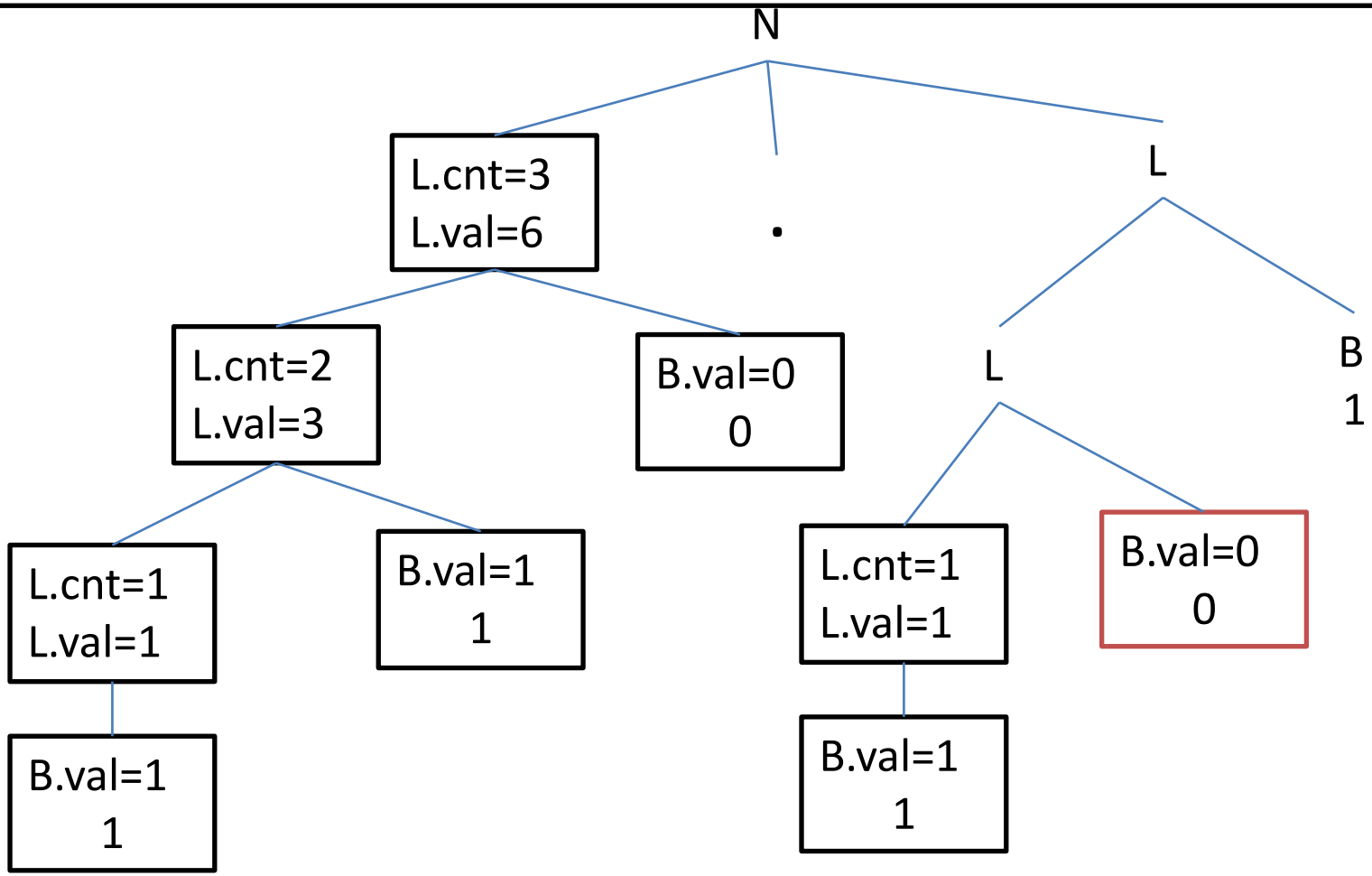
1. $N \rightarrow L_1.L_2$ {N.val = L_1.val + (L_2.val / 2^L_2.cnt)}
2. $L \rightarrow L_1B$ {L.cnt=L_1.cnt+1; L.val=L_1.val*2 + B.val}
3. $L \rightarrow B$ {L.cnt = 1 ; L.val = B.val}
4. $B \rightarrow 0$ {B.val = 0}
5. $B \rightarrow 1$ {B.val = 1}



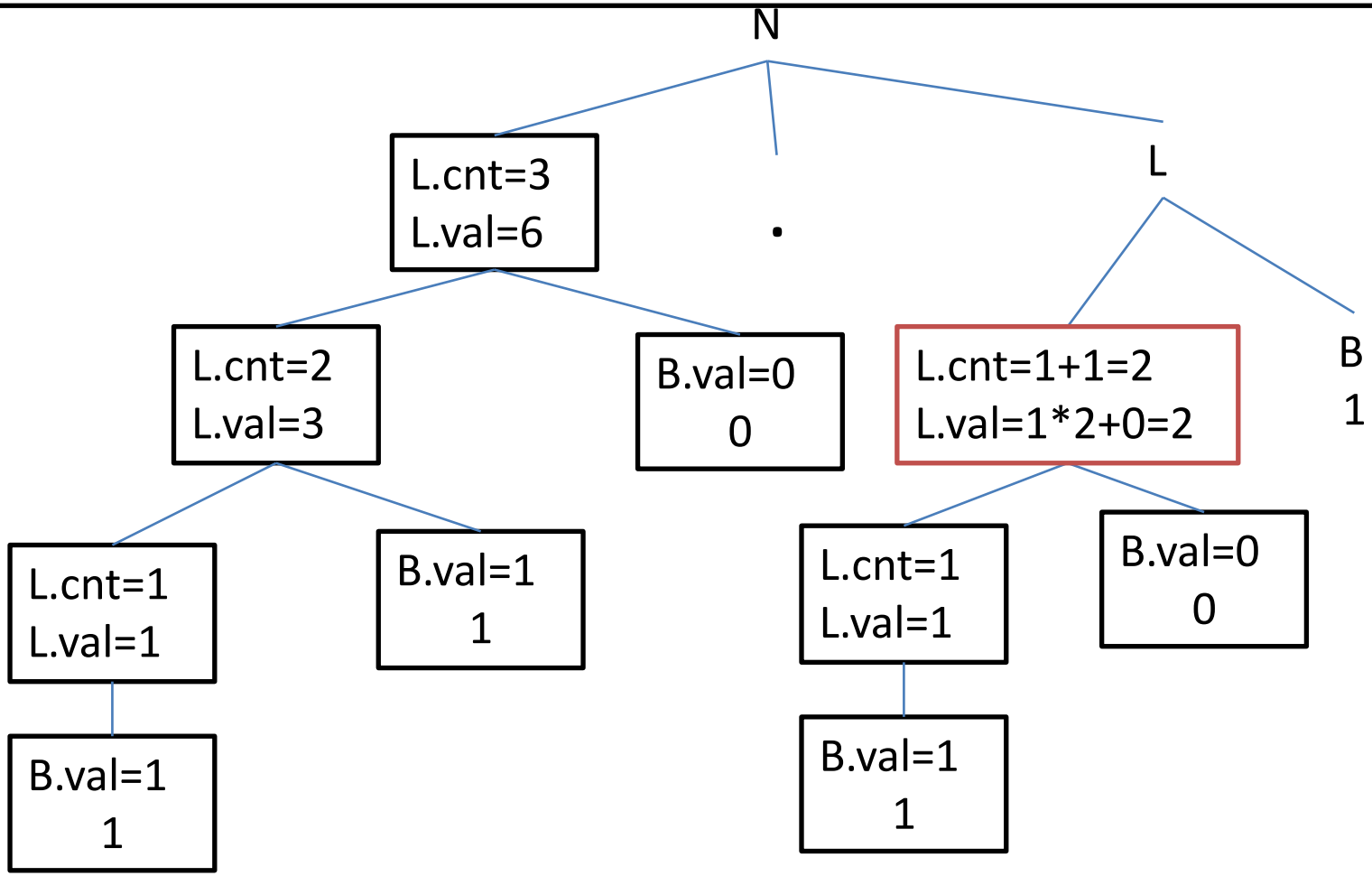
1. $N \rightarrow L_1.L_2$ {N.val = L_1.val + (L_2.val / 2^L_2.cnt)}
2. $L \rightarrow L_1B$ {L.cnt=L_1.cnt+1; L.val=L_1.val*2 + B.val}
3. $L \rightarrow B$ {L.cnt = 1 ; L.val = B.val}
4. $B \rightarrow 0$ {B.val = 0}
5. $B \rightarrow 1$ {B.val = 1}



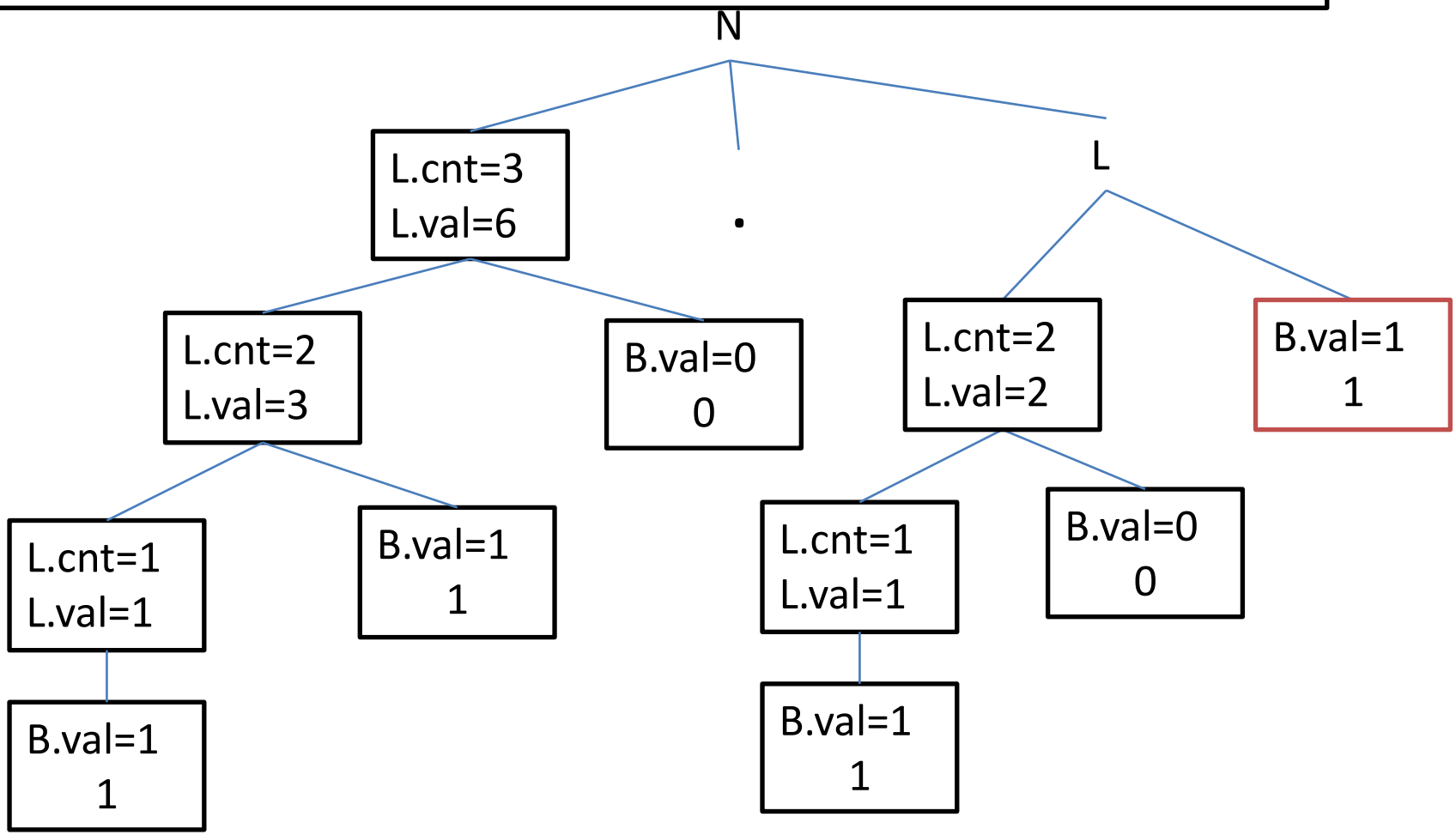
1. $N \rightarrow L_1.L_2$ {N.val = L_1.val + (L_2.val / 2^L_2.cnt)}
2. $L \rightarrow L_1B$ {L.cnt=L_1.cnt+1; L.val=L_1.val*2 + B.val}
3. $L \rightarrow B$ {L.cnt = 1 ; L.val = B.val}
4. $B \rightarrow 0$ {B.val = 0}
5. $B \rightarrow 1$ {B.val = 1}



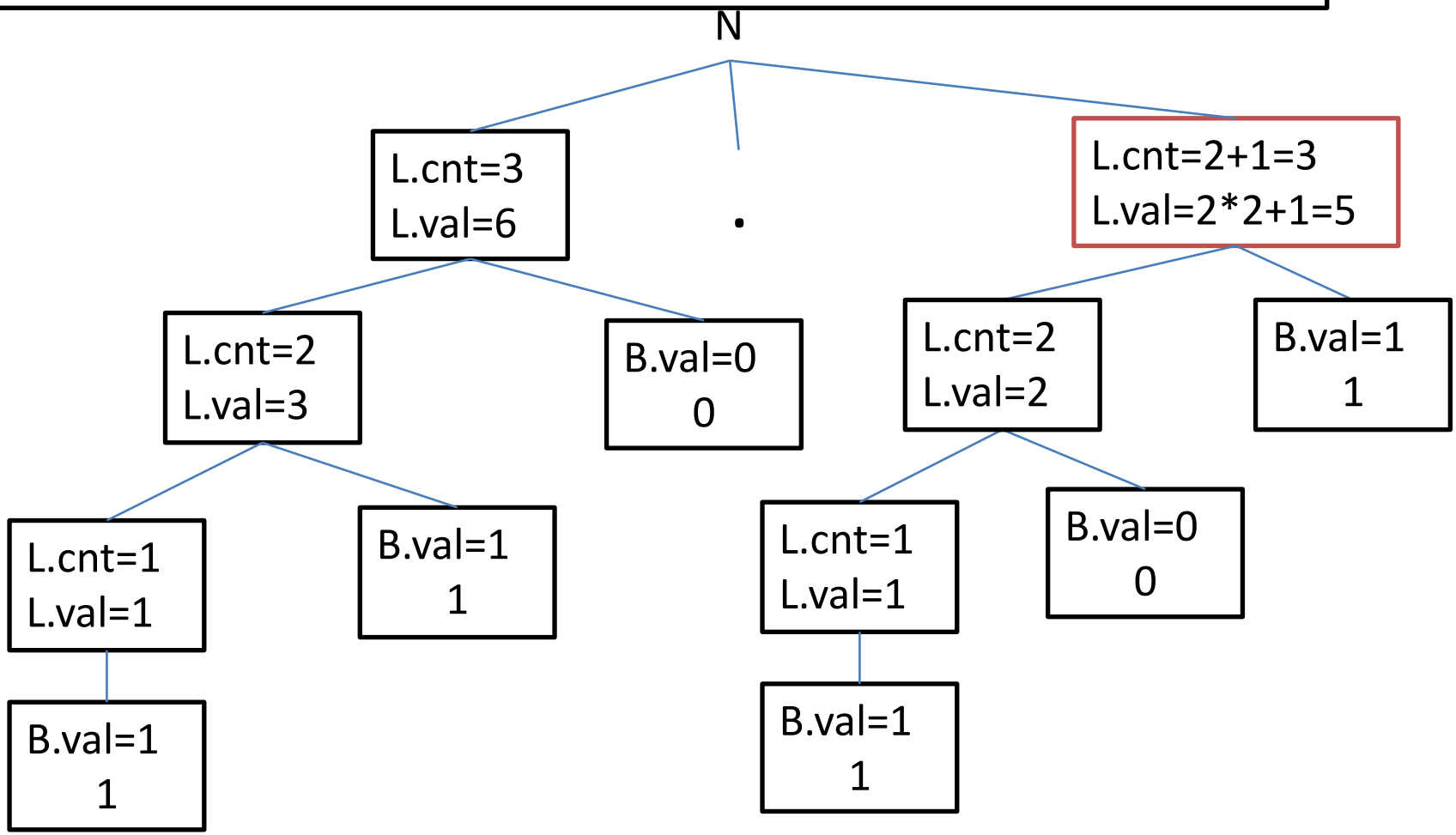
1.	$N \rightarrow L_1.L_2$	$\{N.val = L_1.val + (L_2.val / 2^{\wedge} L_2.cnt)\}$
2.	$L \rightarrow L_1 B$	$\{L.cnt=L_1.cnt+1;\ L.val=L_1.val*2 + B.val\}$
3.	$L \rightarrow B$	$\{L.cnt = 1\ ;\ L.val = B.val\}$
4.	$B \rightarrow 0$	$\{B.val = 0\}$
5.	$B \rightarrow 1$	$\{B.val = 1\}$



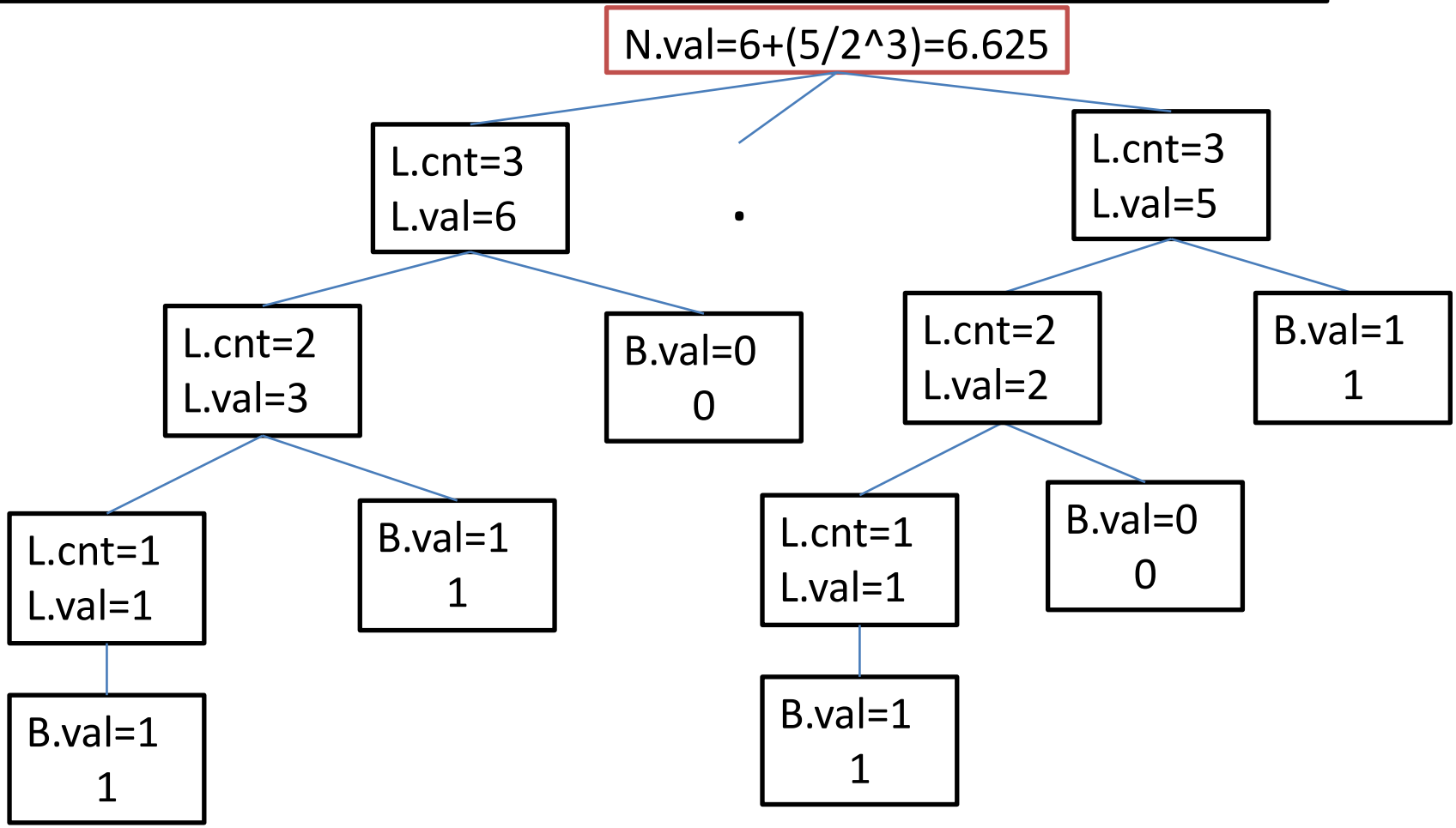
1. $N \rightarrow L_1.L_2$ {N.val = L_1.val + (L_2.val / 2^L_2.cnt)}
2. $L \rightarrow L_1B$ {L.cnt=L_1.cnt+1; L.val=L_1.val*2 + B.val}
3. $L \rightarrow B$ {L.cnt = 1 ; L.val = B.val}
4. $B \rightarrow 0$ {B.val = 0}
5. $B \rightarrow 1$ {B.val = 1}



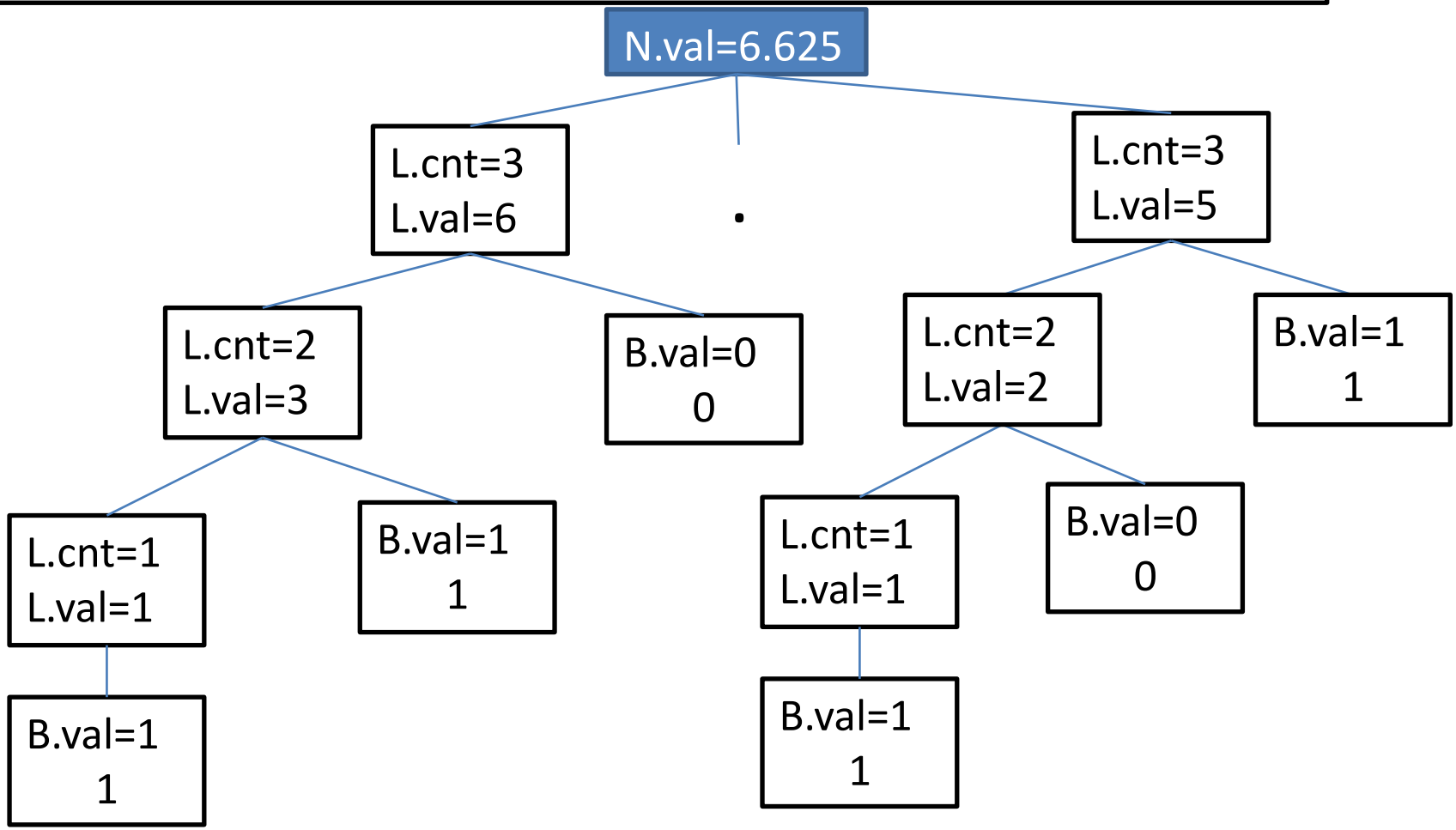
1. $N \rightarrow L_1.L_2$ {N.val = L_1.val + (L_2.val / 2^L_2.cnt)}
2. $L \rightarrow L_1B$ {L.cnt=L_1.cnt+1; L.val=L_1.val*2 + B.val}
3. $L \rightarrow B$ {L.cnt = 1 ; L.val = B.val}
4. $B \rightarrow 0$ {B.val = 0}
5. $B \rightarrow 1$ {B.val = 1}



1. $N \rightarrow L_1.L_2$ {N.val = L_1.val + (L_2.val / 2^L_2.cnt)}
2. $L \rightarrow L_1B$ {L.cnt=L_1.cnt+1; L.val=L_1.val*2 + B.val}
3. $L \rightarrow B$ {L.cnt = 1 ; L.val = B.val}
4. $B \rightarrow 0$ {B.val = 0}
5. $B \rightarrow 1$ {B.val = 1}



1. $N \rightarrow L_1.L_2$ {N.val = L_1.val + (L_2.val / 2^L_2.cnt)}
2. $L \rightarrow L_1B$ {L.cnt=L_1.cnt+1; L.val=L_1.val*2 + B.val}
3. $L \rightarrow B$ {L.cnt = 1 ; L.val = B.val}
4. $B \rightarrow 0$ {B.val = 0}
5. $B \rightarrow 1$ {B.val = 1}



Example 3

- Given a grammar:

  $E \rightarrow E + T \mid T$

  $T \rightarrow T * F \mid F$

  $F \rightarrow num$

- What are the semantic rules(informal notations) for this grammar?
  - There can be a ‘value’ attribute for E, T and F. (non-terminals)
  - There can be a ‘lexvalue’ attribute for num (terminal)

Example 3

- Given a grammar:

  E → E + T | T

  T → T * F | F

  F → num

1. E → E + T { E.value = E.value + T.value}
2. E → T {E.value = T.value}
3. T → T * F {T.value = T.value * F.value}
4. T → F {T.value = F.value}
5. F → num {F.value = num.lexvalue}

Example 3

1. E → E + T { E.value = E.value + T.value}
2. E → T {E.value = T.value}
3. T → T * F {T.value = T.value * F.value}
4. T → F {T.value = F.value}
5. F → num {F.value = num.lexvalue}

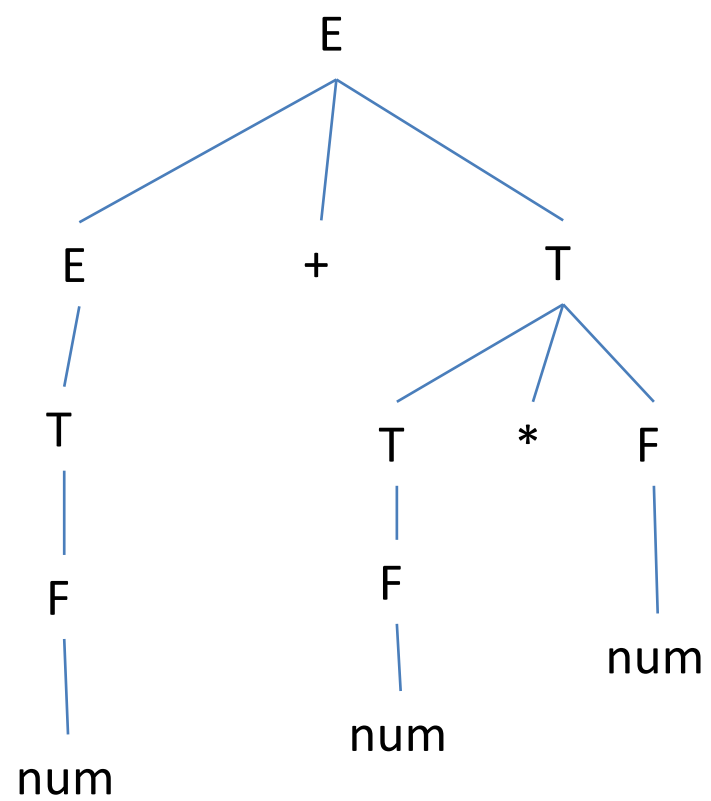
- Parse Tree??

For input, 2 + 3 * 4

Example 3

1. E → E + T { E.value = E.value + T.value}
2. E → T {E.value = T.value}
3. T → T * F {T.value = T.value * F.value}
4. T → F {T.value = F.value}
5. F → num {F.value = num.lexvalue}

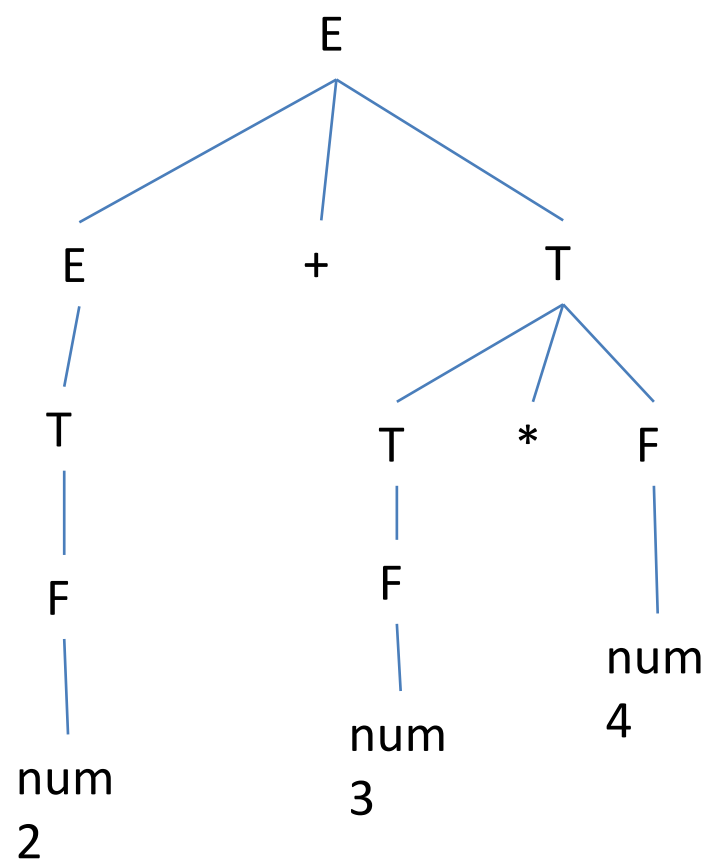
For input, 2 + 3 * 4



Example 3

1. E → E + T { E.value = E.value + T.value}
2. E → T {E.value = T.value}
3. T → T * F {T.value = T.value * F.value}
4. T → F {T.value = F.value}
5. F → num {F.value = num.lexvalue}

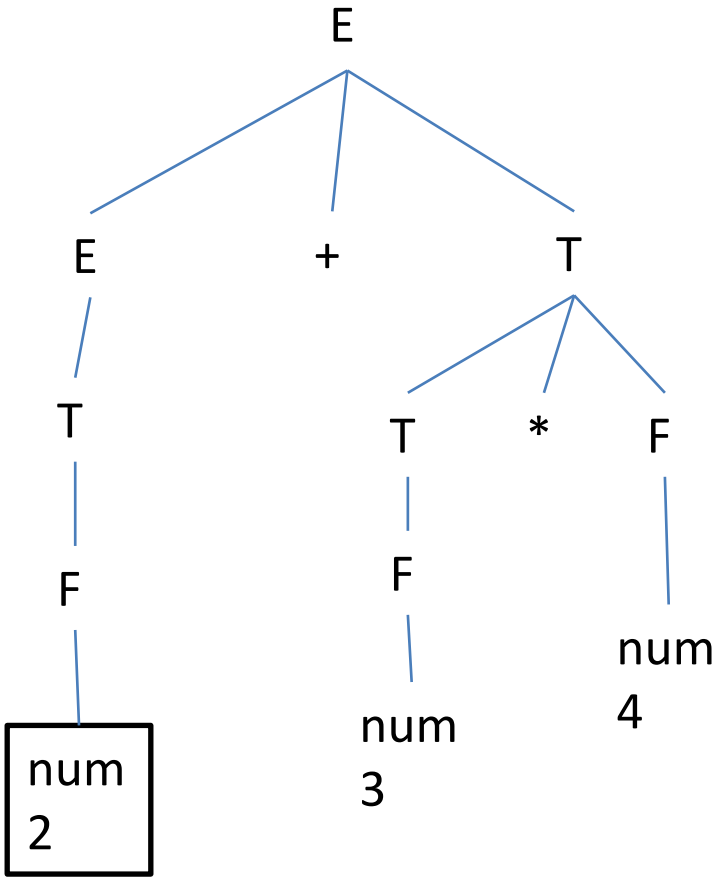
For input, 2 + 3 * 4



Example 3

1. E → E + T { E.value = E.value + T.value}
2. E → T {E.value = T.value}
3. T → T * F {T.value = T.value * F.value}
4. T → F {T.value = F.value}
5. F → num {F.value = num.lexvalue}

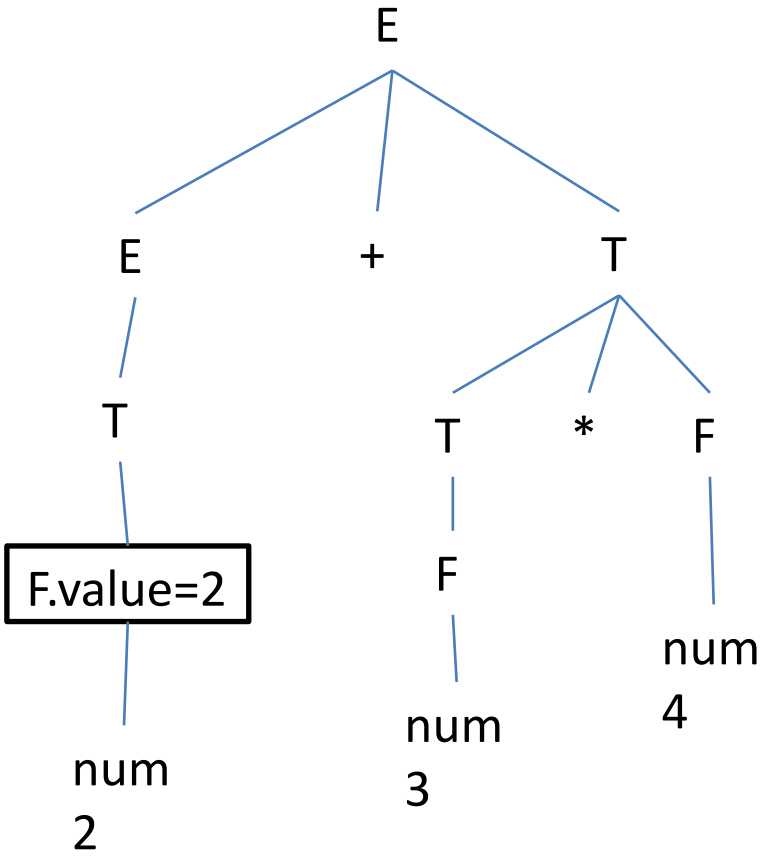
For input, 2 + 3 * 4



Example 3

1. E → E + T { E.value = E.value + T.value}
2. E → T {E.value = T.value}
3. T → T * F {T.value = T.value * F.value}
4. T → F {T.value = F.value}
5. F → num {F.value = num.lexvalue}

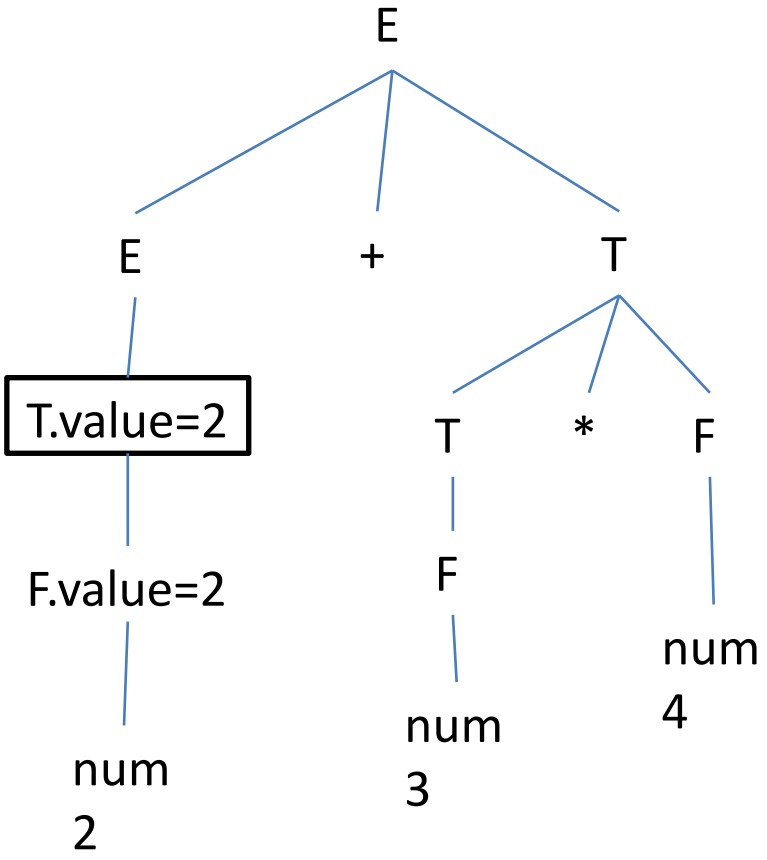
For input, 2 + 3 * 4



Example 3

1. E → E + T { E.value = E.value + T.value}
2. E → T {E.value = T.value}
3. T → T * F {T.value = T.value * F.value}
4. T → F {T.value = F.value}
5. F → num {F.value = num.lexvalue}

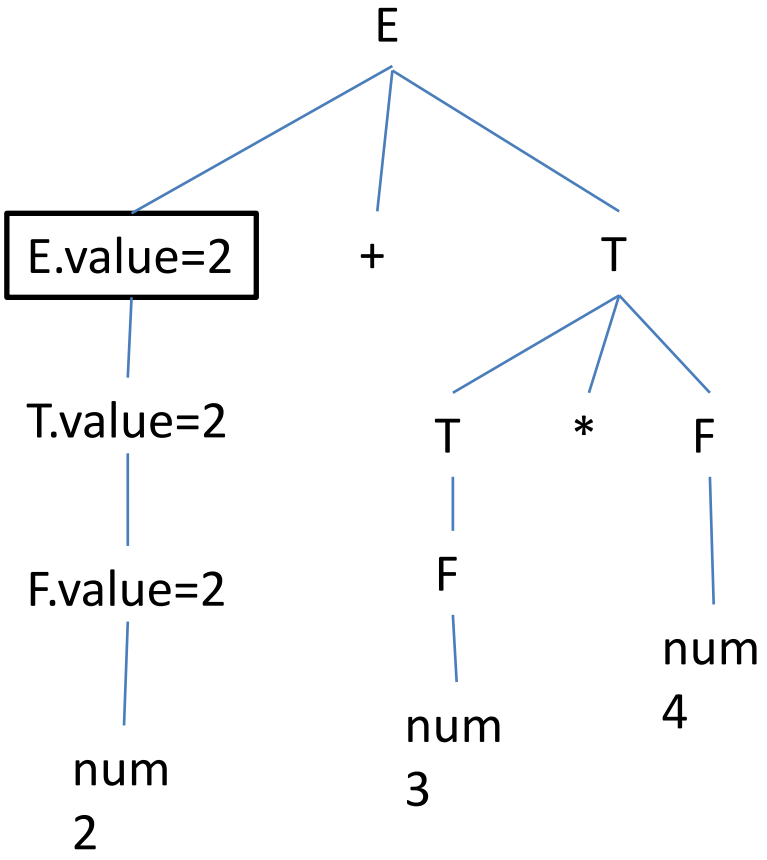
For input, 2 + 3 * 4



Example 3

1. E → E + T { E.value = E.value + T.value}
2. E → T {E.value = T.value}
3. T → T * F {T.value = T.value * F.value}
4. T → F {T.value = F.value}
5. F → num {F.value = num.lexvalue}

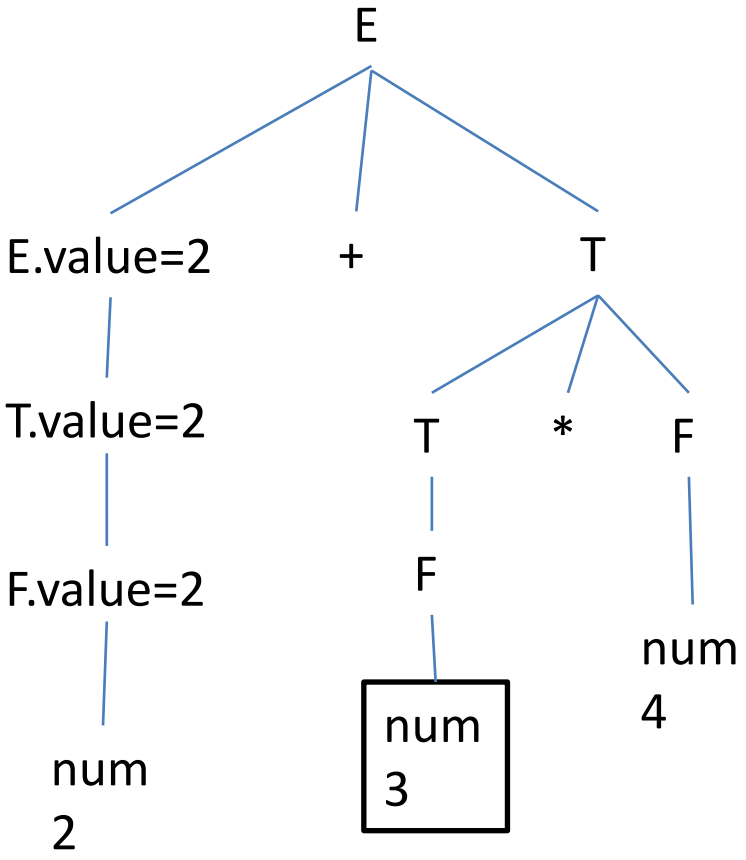
For input, 2 + 3 * 4



Example 3

1. E → E + T { E.value = E.value + T.value}
2. E → T {E.value = T.value}
3. T → T * F {T.value = T.value * F.value}
4. T → F {T.value = F.value}
5. F → num {F.value = num.lexvalue}

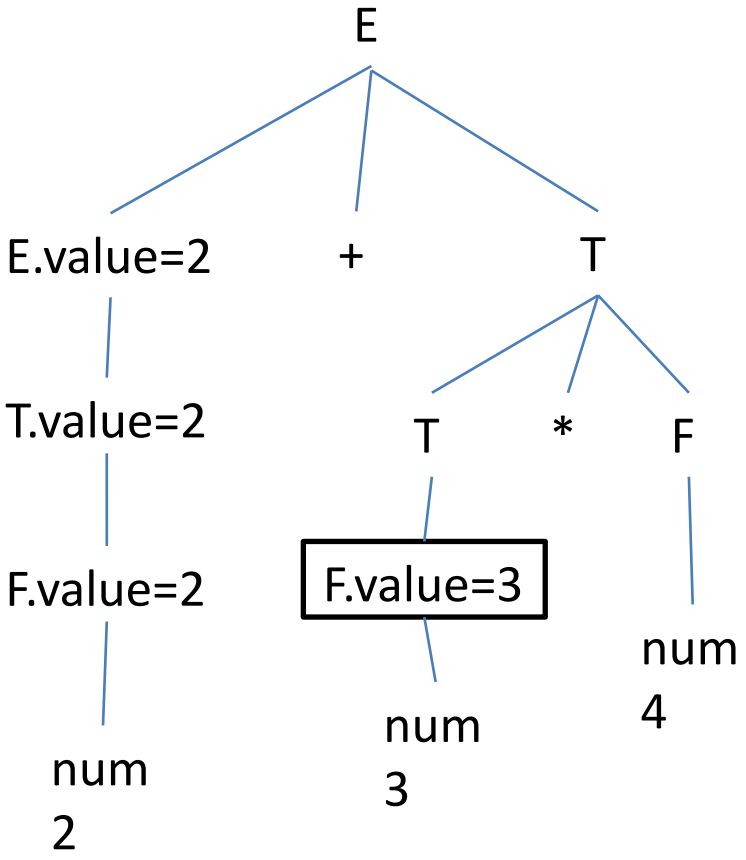
For input, 2 + 3 * 4



Example 3

1. E → E + T { E.value = E.value + T.value}
2. E → T {E.value = T.value}
3. T → T * F {T.value = T.value * F.value}
4. T → F {T.value = F.value}
5. F → num {F.value = num.lexvalue}

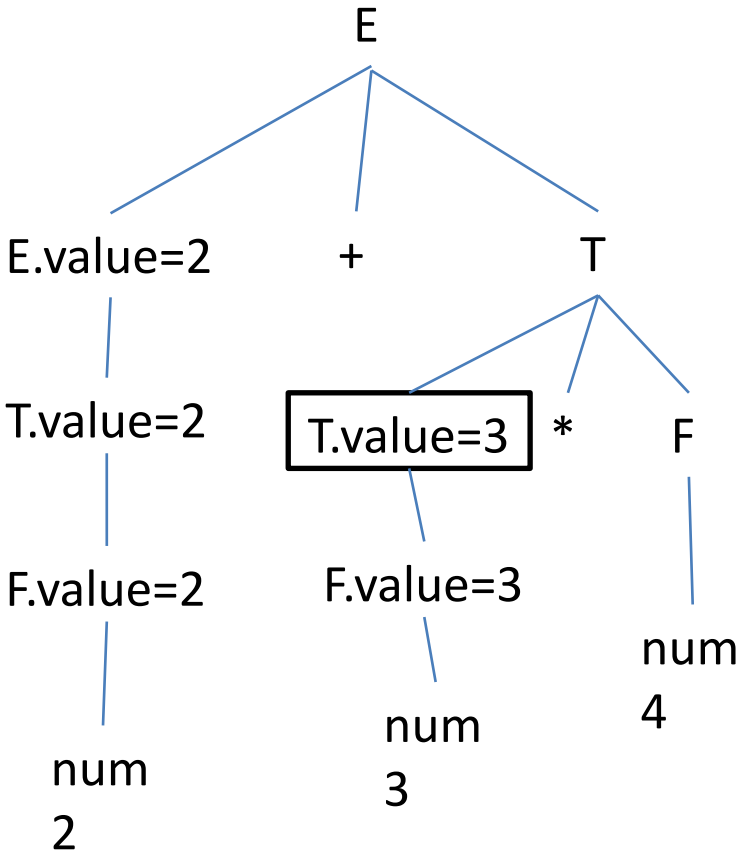
For input, 2 + 3 * 4



Example 3

1. E → E + T { E.value = E.value + T.value}
2. E → T {E.value = T.value}
3. T → T * F {T.value = T.value * F.value}
4. T → F {T.value = F.value}
5. F → num {F.value = num.lexvalue}

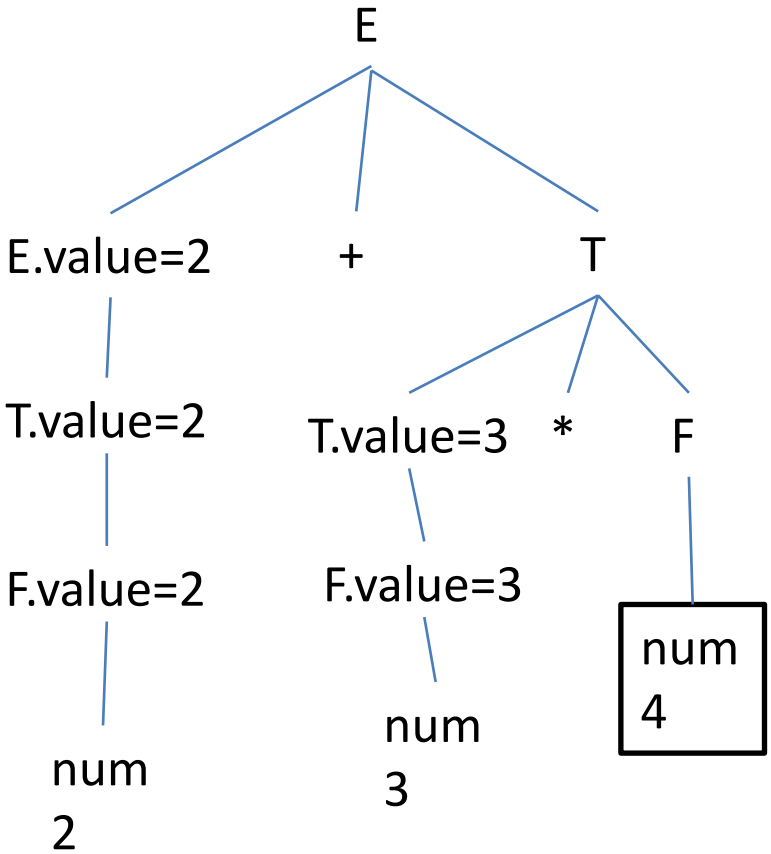
For input, 2 + 3 * 4



Example 3

1. E → E + T { E.value = E.value + T.value}
2. E → T {E.value = T.value}
3. T → T * F {T.value = T.value * F.value}
4. T → F {T.value = F.value}
5. F → num {F.value = num.lexvalue}

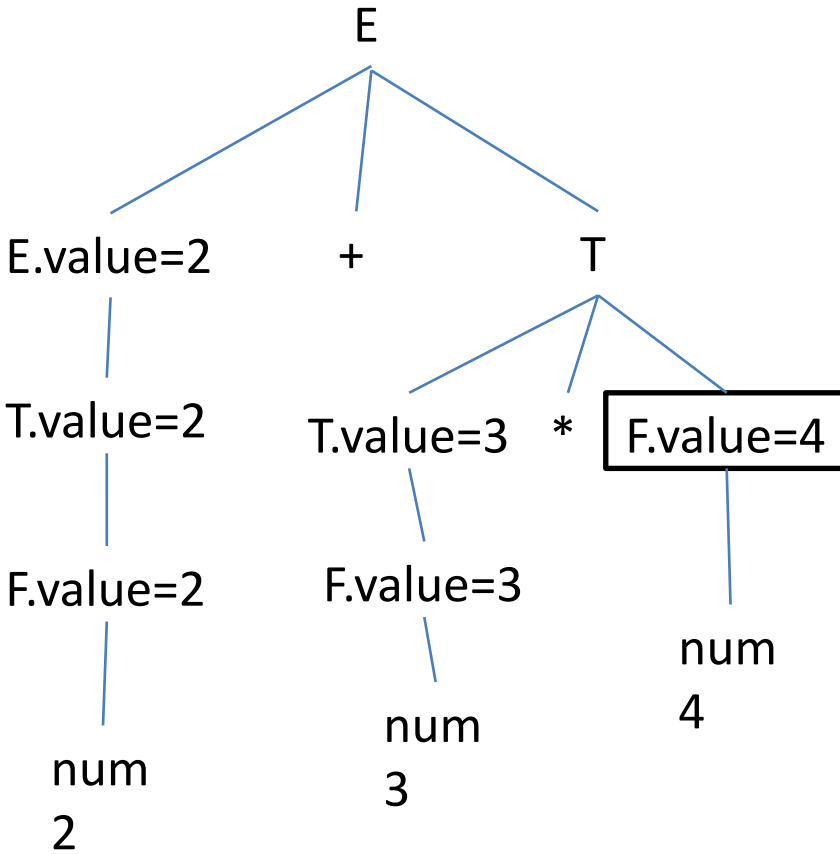
For input, 2 + 3 * 4



Example 3

1. E → E + T { E.value = E.value + T.value}
2. E → T {E.value = T.value}
3. T → T * F {T.value = T.value * F.value}
4. T → F {T.value = F.value}
5. F → num {F.value = num.lexvalue}

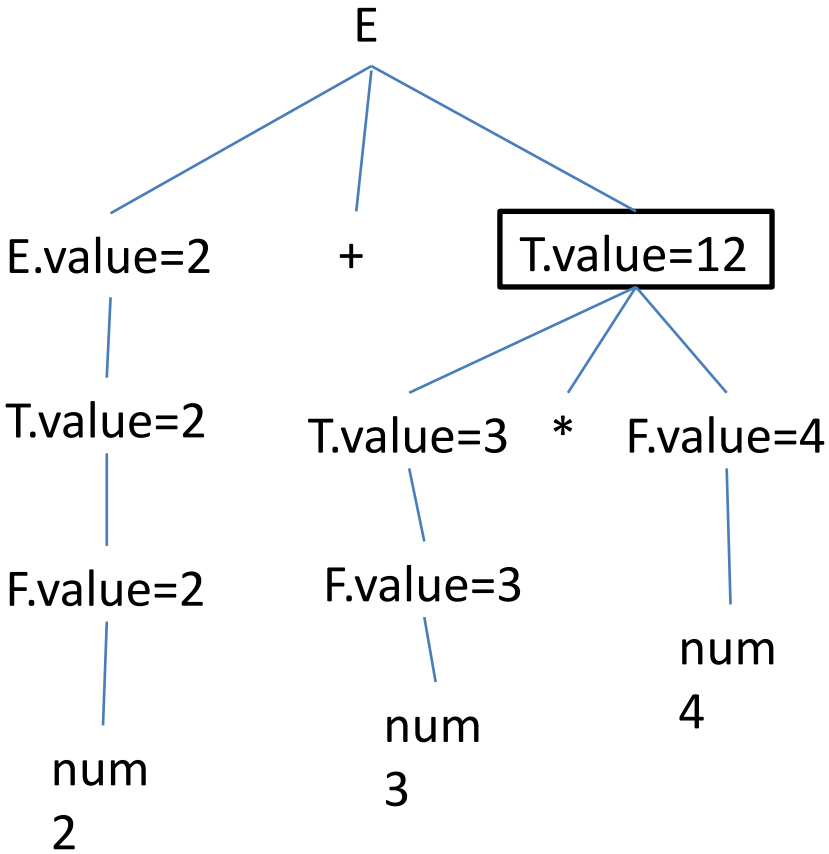
For input, 2 + 3 * 4



Example 3

1. E → E + T { E.value = E.value + T.value}
2. E → T {E.value = T.value}
3. T → T * F {T.value = T.value * F.value}
4. T → F {T.value = F.value}
5. F → num {F.value = num.lexvalue}

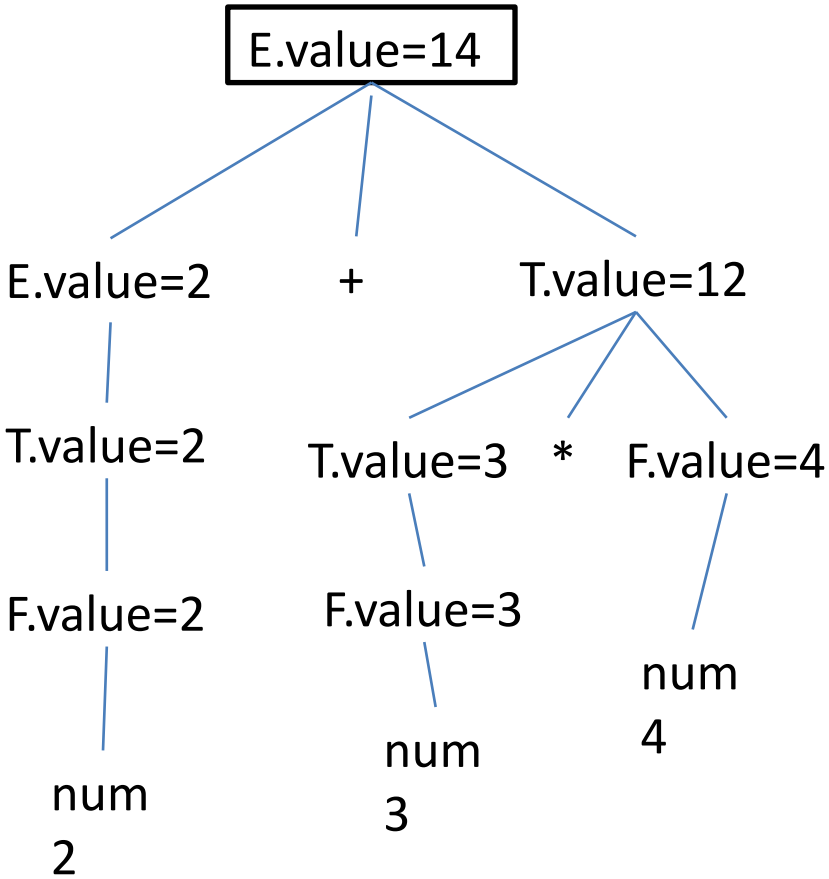
For input, 2 + 3 * 4



Example 3

1. E → E + T { E.value = E.value + T.value}
2. E → T {E.value = T.value}
3. T → T * F {T.value = T.value * F.value}
4. T → F {T.value = F.value}
5. F → num {F.value = num.lexvalue}

For input, 2 + 3 * 4



SDT for desktop calculator

E → E + T

E → T

T → T * F

T → F

F → X ^ F

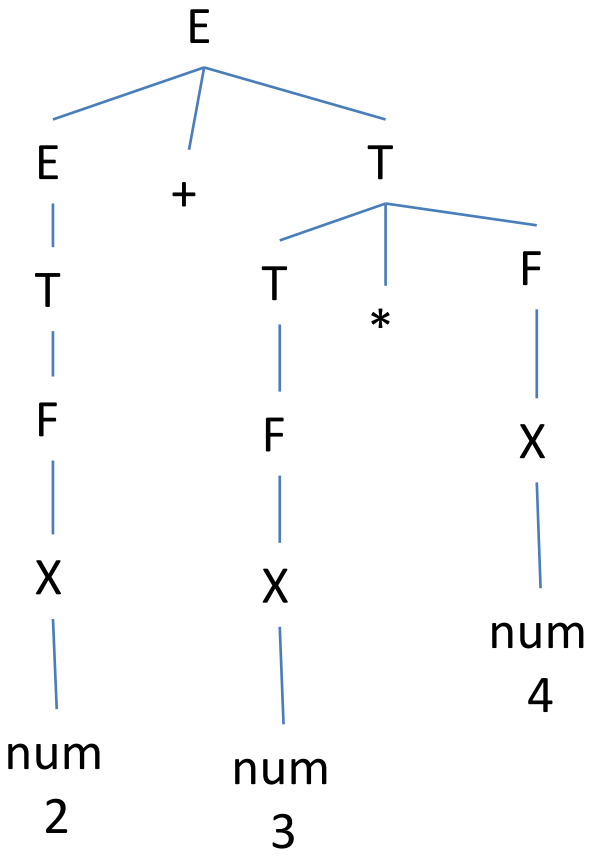
F → X

X → num

SDT for desktop calculator

$E \rightarrow E_1 + T$	$\{E.value = E_1.value + T.value\}$
$E \rightarrow T$	$\{E.value = T.value\}$
$T \rightarrow T_1 * F$	$\{T.value = T_1.value * F.value\}$
$T \rightarrow F$	$\{T.value = F.value\}$
$F \rightarrow X \wedge F_1$	$\{F.value = X.value \wedge F_1.value\}$
$F \rightarrow X$	$\{F.value = X.value\}$
$X \rightarrow num$	$\{X.value = num.lexvalue\}$

For input, 2 + 3 * 4



SDT for desktop calculator

$E \rightarrow E_1 + T$	$\{E.value = E_1.value + T.value\}$
$E \rightarrow T$	$\{E.value = T.value\}$
$T \rightarrow T_1 * F$	$\{T.value = T_1.value * F.value\}$
$T \rightarrow F$	$\{T.value = F.value\}$
$F \rightarrow X \wedge F_1$	$\{F.value = X.value \wedge F_1.value\}$
$F \rightarrow X$	$\{F.value = X.value\}$
$X \rightarrow num$	$\{X.value = num.lexvalue\}$

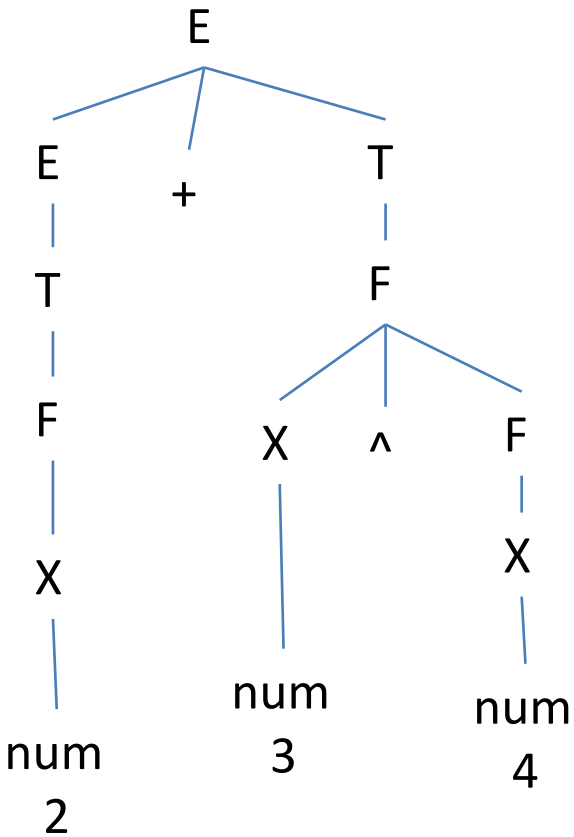
Parse tree???

For input, 2 + 3 ^ 4

SDT for desktop calculator

$E \rightarrow E_1 + T$	$\{E.value = E_1.value + T.value\}$
$E \rightarrow T$	$\{E.value = T.value\}$
$T \rightarrow T_1 * F$	$\{T.value = T_1.value * F.value\}$
$T \rightarrow F$	$\{T.value = F.value\}$
$F \rightarrow X \wedge F_1$	$\{F.value = X.value \wedge F_1.value\}$
$F \rightarrow X$	$\{F.value = X.value\}$
$X \rightarrow num$	$\{X.value = num.lexvalue\}$

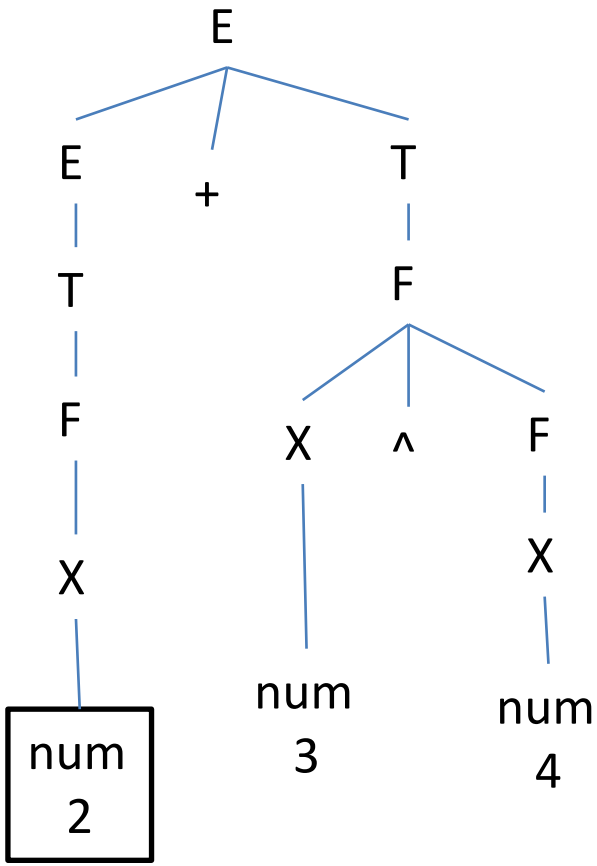
For input, 2 + 3 ^ 4



SDT for desktop calculator

$E \rightarrow E_1 + T$	$\{E.value = E_1.value + T.value\}$
$E \rightarrow T$	$\{E.value = T.value\}$
$T \rightarrow T_1 * F$	$\{T.value = T_1.value * F.value\}$
$T \rightarrow F$	$\{T.value = F.value\}$
$F \rightarrow X \wedge F_1$	$\{F.value = X.value \wedge F_1.value\}$
$F \rightarrow X$	$\{F.value = X.value\}$
$X \rightarrow num$	$\{X.value = num.lexvalue\}$

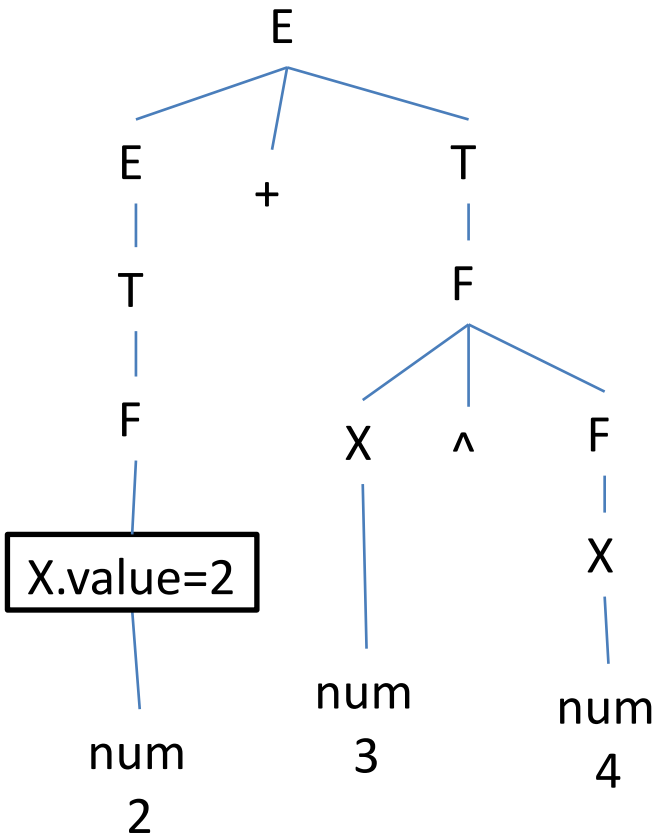
For input, 2 + 3 ^ 4



SDT for desktop calculator

$E \rightarrow E_1 + T$	$\{E.value = E_1.value + T.value\}$
$E \rightarrow T$	$\{E.value = T.value\}$
$T \rightarrow T_1 * F$	$\{T.value = T_1.value * F.value\}$
$T \rightarrow F$	$\{T.value = F.value\}$
$F \rightarrow X \wedge F_1$	$\{F.value = X.value \wedge F_1.value\}$
$F \rightarrow X$	$\{F.value = X.value\}$
$X \rightarrow num$	$\{X.value = num.lexvalue\}$

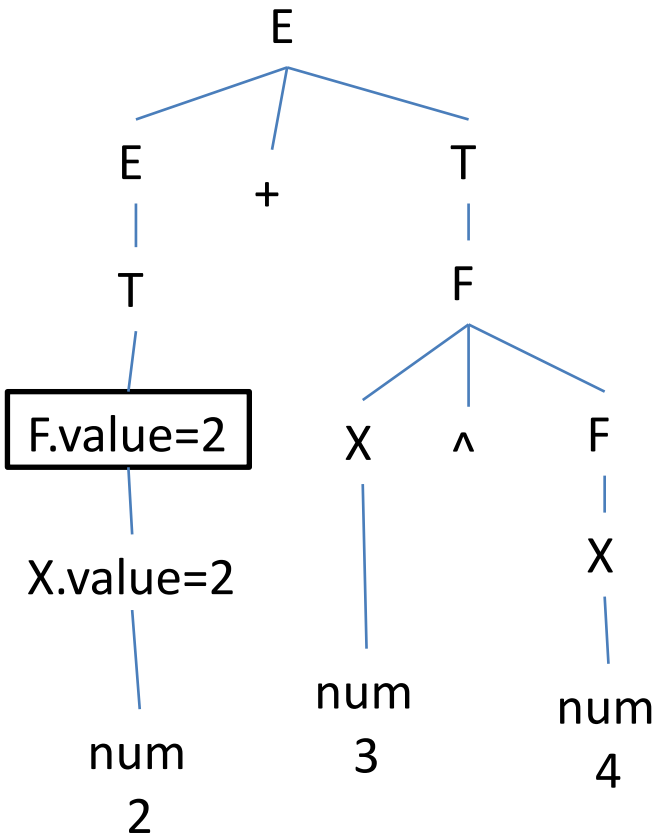
For input, 2 + 3 ^ 4



SDT for desktop calculator

$E \rightarrow E_1 + T$	$\{E.value = E_1.value + T.value\}$
$E \rightarrow T$	$\{E.value = T.value\}$
$T \rightarrow T_1 * F$	$\{T.value = T_1.value * F.value\}$
$T \rightarrow F$	$\{T.value = F.value\}$
$F \rightarrow X \wedge F_1$	$\{F.value = X.value \wedge F_1.value\}$
$F \rightarrow X$	$\{F.value = X.value\}$
$X \rightarrow num$	$\{X.value = num.lexvalue\}$

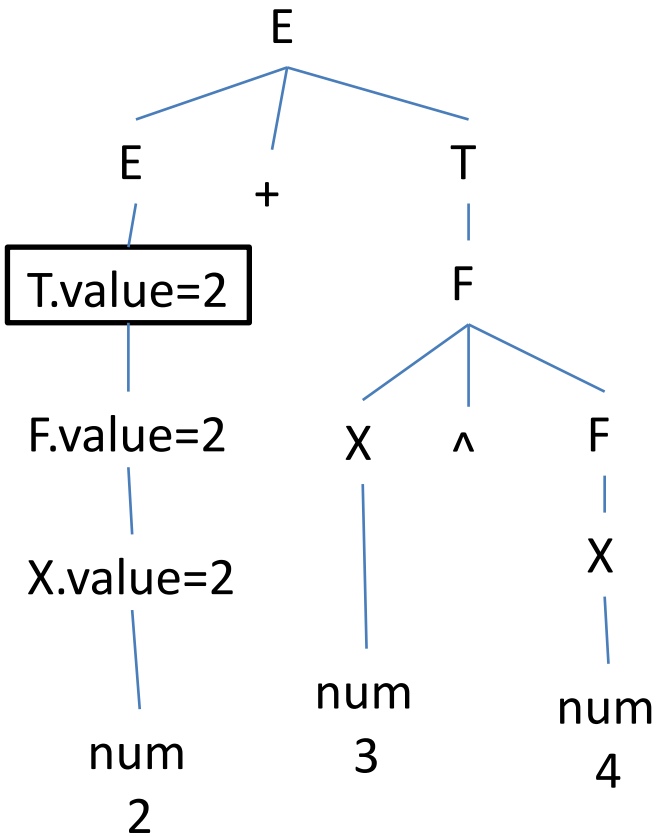
For input, 2 + 3 ^ 4



SDT for desktop calculator

$E \rightarrow E_1 + T$	$\{E.value = E_1.value + T.value\}$
$E \rightarrow T$	$\{E.value = T.value\}$
$T \rightarrow T_1 * F$	$\{T.value = T_1.value * F.value\}$
$T \rightarrow F$	$\{T.value = F.value\}$
$F \rightarrow X \wedge F_1$	$\{F.value = X.value \wedge F_1.value\}$
$F \rightarrow X$	$\{F.value = X.value\}$
$X \rightarrow num$	$\{X.value = num.lexvalue\}$

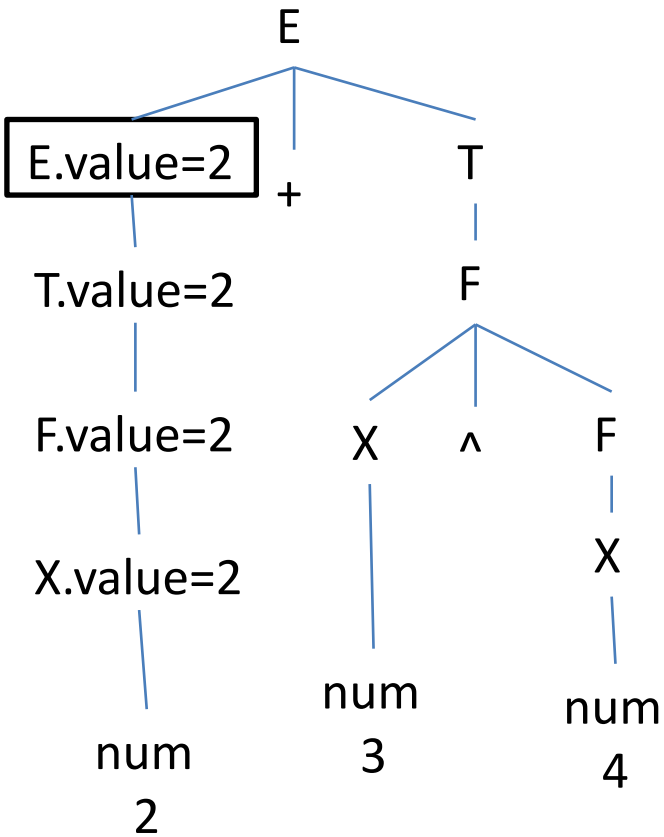
For input, 2 + 3 ^ 4



SDT for desktop calculator

$E \rightarrow E_1 + T$	$\{E.value = E_1.value + T.value\}$
$E \rightarrow T$	$\{E.value = T.value\}$
$T \rightarrow T_1 * F$	$\{T.value = T_1.value * F.value\}$
$T \rightarrow F$	$\{T.value = F.value\}$
$F \rightarrow X \wedge F_1$	$\{F.value = X.value \wedge F_1.value\}$
$F \rightarrow X$	$\{F.value = X.value\}$
$X \rightarrow num$	$\{X.value = num.lexvalue\}$

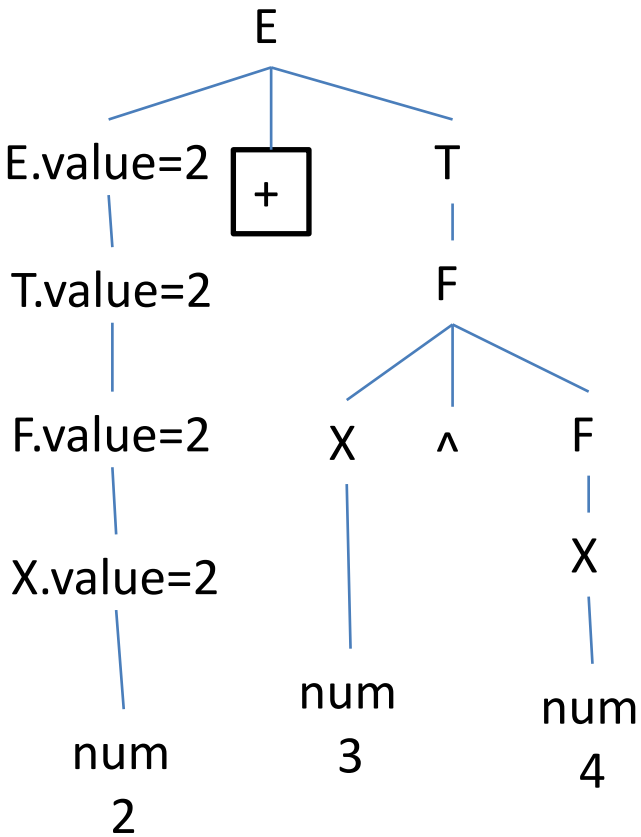
For input, 2 + 3 ^ 4



SDT for desktop calculator

$E \rightarrow E_1 + T$	$\{E.value = E_1.value + T.value\}$
$E \rightarrow T$	$\{E.value = T.value\}$
$T \rightarrow T_1 * F$	$\{T.value = T_1.value * F.value\}$
$T \rightarrow F$	$\{T.value = F.value\}$
$F \rightarrow X \wedge F_1$	$\{F.value = X.value \wedge F_1.value\}$
$F \rightarrow X$	$\{F.value = X.value\}$
$X \rightarrow num$	$\{X.value = num.lexvalue\}$

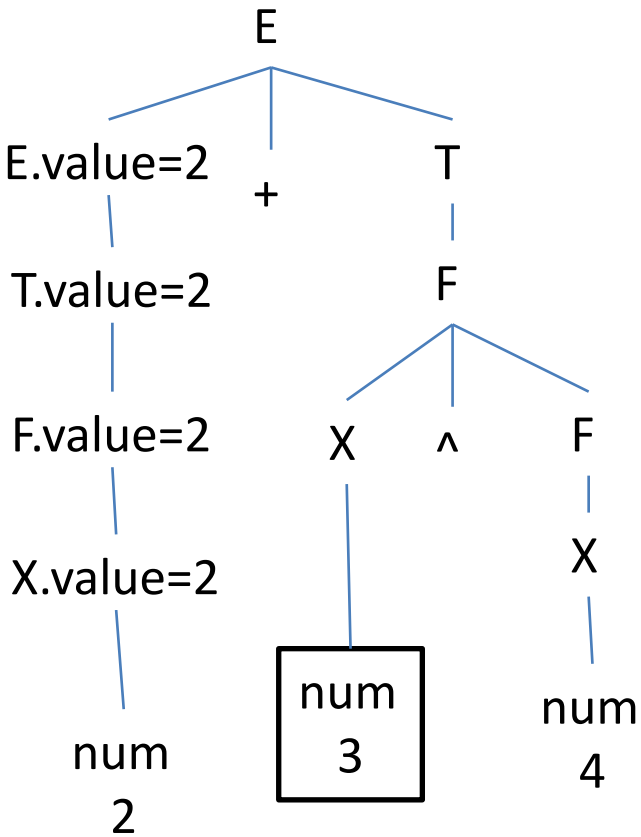
For input, 2 + 3 ^ 4



SDT for desktop calculator

$E \rightarrow E_1 + T$	$\{E.value = E_1.value + T.value\}$
$E \rightarrow T$	$\{E.value = T.value\}$
$T \rightarrow T_1 * F$	$\{T.value = T_1.value * F.value\}$
$T \rightarrow F$	$\{T.value = F.value\}$
$F \rightarrow X \wedge F_1$	$\{F.value = X.value \wedge F_1.value\}$
$F \rightarrow X$	$\{F.value = X.value\}$
$X \rightarrow num$	$\{X.value = num.lexvalue\}$

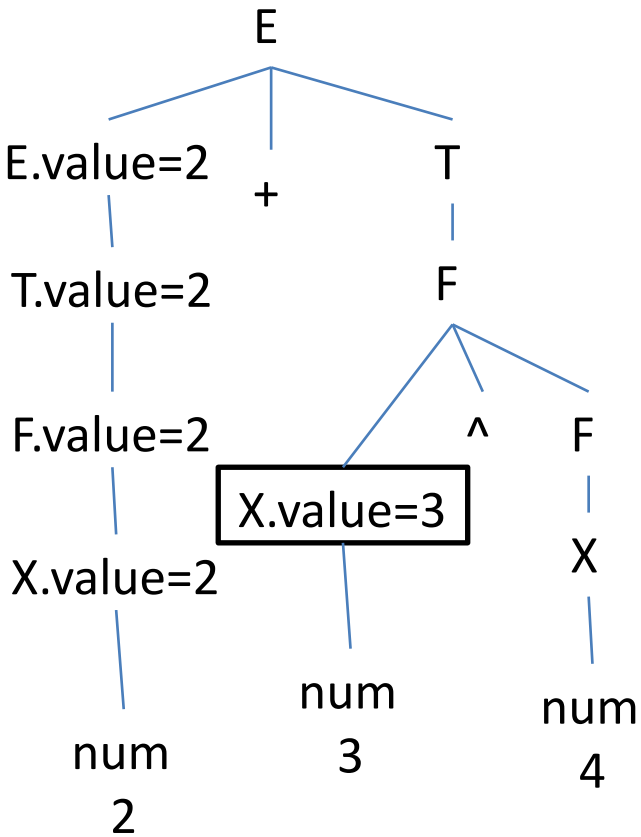
For input, 2 + 3 ^ 4



SDT for desktop calculator

$E \rightarrow E_1 + T$	$\{E.value = E_1.value + T.value\}$
$E \rightarrow T$	$\{E.value = T.value\}$
$T \rightarrow T_1 * F$	$\{T.value = T_1.value * F.value\}$
$T \rightarrow F$	$\{T.value = F.value\}$
$F \rightarrow X \wedge F_1$	$\{F.value = X.value \wedge F_1.value\}$
$F \rightarrow X$	$\{F.value = X.value\}$
$X \rightarrow num$	$\{X.value = num.lexvalue\}$

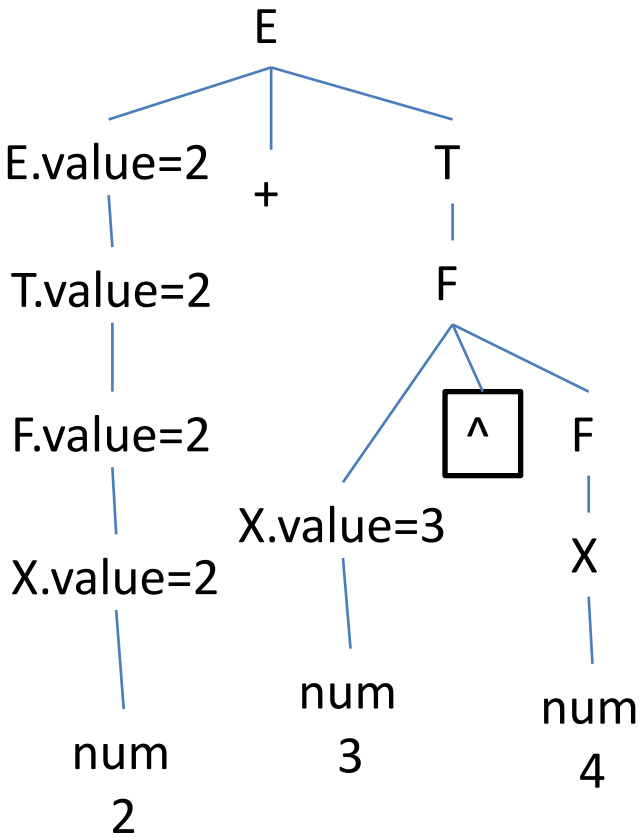
For input, 2 + 3 ^ 4



SDT for desktop calculator

$E \rightarrow E_1 + T$	$\{E.value = E_1.value + T.value\}$
$E \rightarrow T$	$\{E.value = T.value\}$
$T \rightarrow T_1 * F$	$\{T.value = T_1.value * F.value\}$
$T \rightarrow F$	$\{T.value = F.value\}$
$F \rightarrow X \wedge F_1$	$\{F.value = X.value \wedge F_1.value\}$
$F \rightarrow X$	$\{F.value = X.value\}$
$X \rightarrow num$	$\{X.value = num.lexvalue\}$

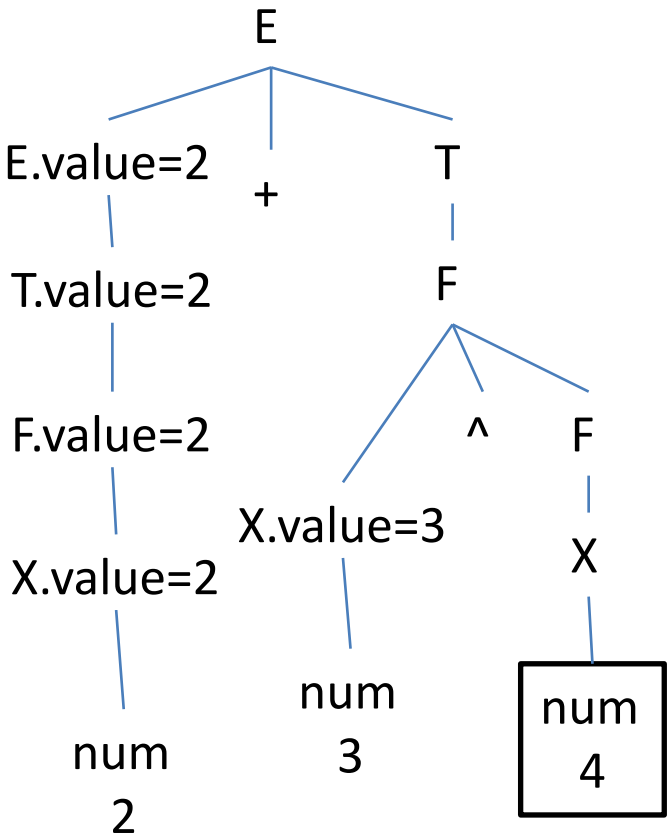
For input, 2 + 3 ^ 4



SDT for desktop calculator

$E \rightarrow E_1 + T$	$\{E.value = E_1.value + T.value\}$
$E \rightarrow T$	$\{E.value = T.value\}$
$T \rightarrow T_1 * F$	$\{T.value = T_1.value * F.value\}$
$T \rightarrow F$	$\{T.value = F.value\}$
$F \rightarrow X \wedge F_1$	$\{F.value = X.value \wedge F_1.value\}$
$F \rightarrow X$	$\{F.value = X.value\}$
$X \rightarrow num$	$\{X.value = num.lexvalue\}$

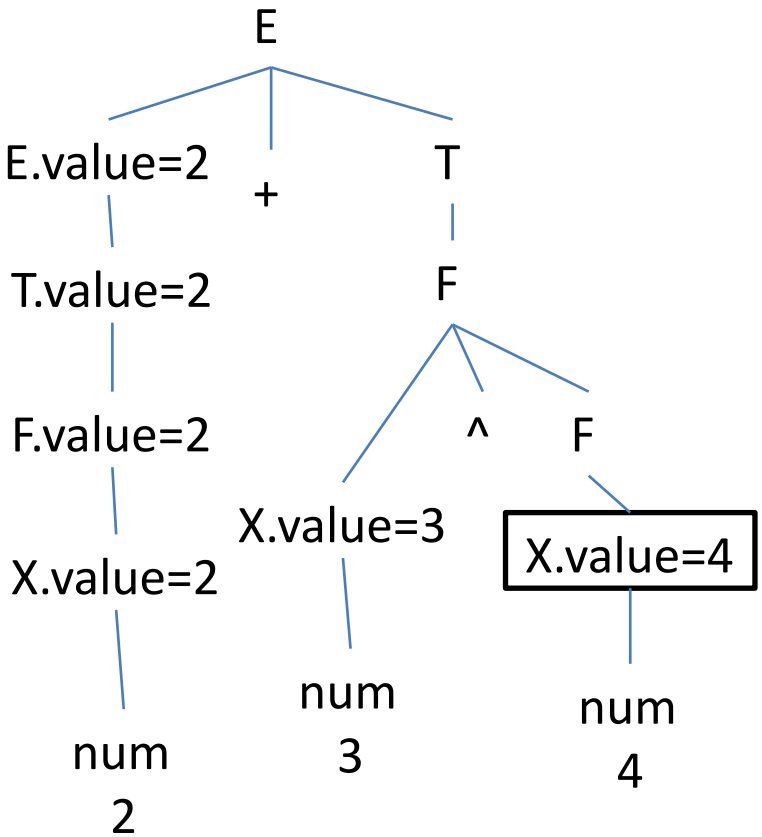
For input, 2 + 3 ^ 4



SDT for desktop calculator

$E \rightarrow E_1 + T$	$\{E.value = E_1.value + T.value\}$
$E \rightarrow T$	$\{E.value = T.value\}$
$T \rightarrow T_1 * F$	$\{T.value = T_1.value * F.value\}$
$T \rightarrow F$	$\{T.value = F.value\}$
$F \rightarrow X \wedge F_1$	$\{F.value = X.value \wedge F_1.value\}$
$F \rightarrow X$	$\{F.value = X.value\}$
$X \rightarrow num$	$\{X.value = num.lexvalue\}$

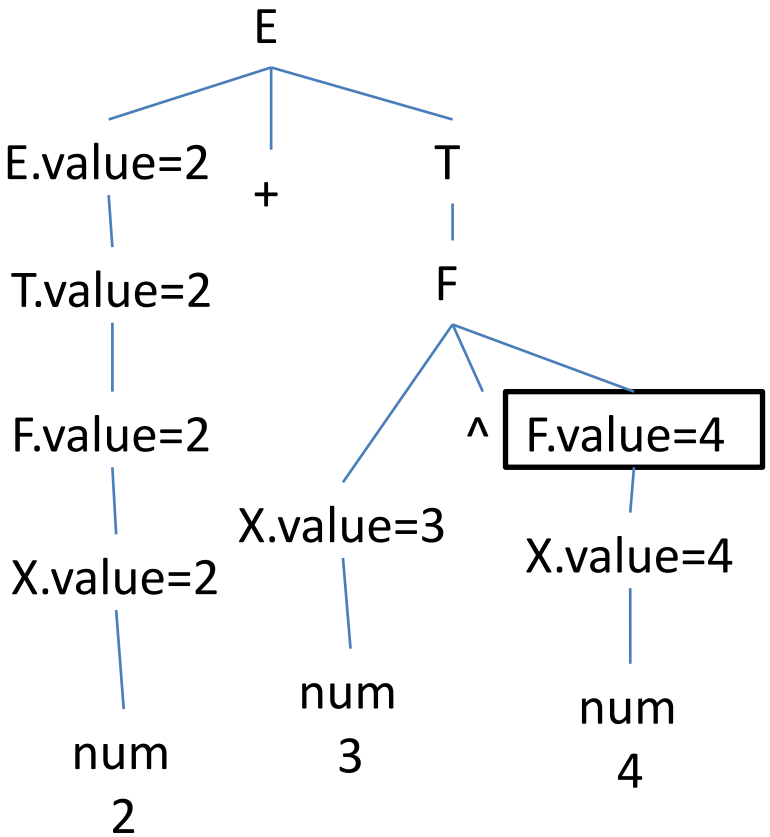
For input, 2 + 3 ^ 4



SDT for desktop calculator

$E \rightarrow E_1 + T$	$\{E.value = E_1.value + T.value\}$
$E \rightarrow T$	$\{E.value = T.value\}$
$T \rightarrow T_1 * F$	$\{T.value = T_1.value * F.value\}$
$T \rightarrow F$	$\{T.value = F.value\}$
$F \rightarrow X \wedge F_1$	$\{F.value = X.value \wedge F_1.value\}$
$F \rightarrow X$	$\{F.value = X.value\}$
$X \rightarrow num$	$\{X.value = num.lexvalue\}$

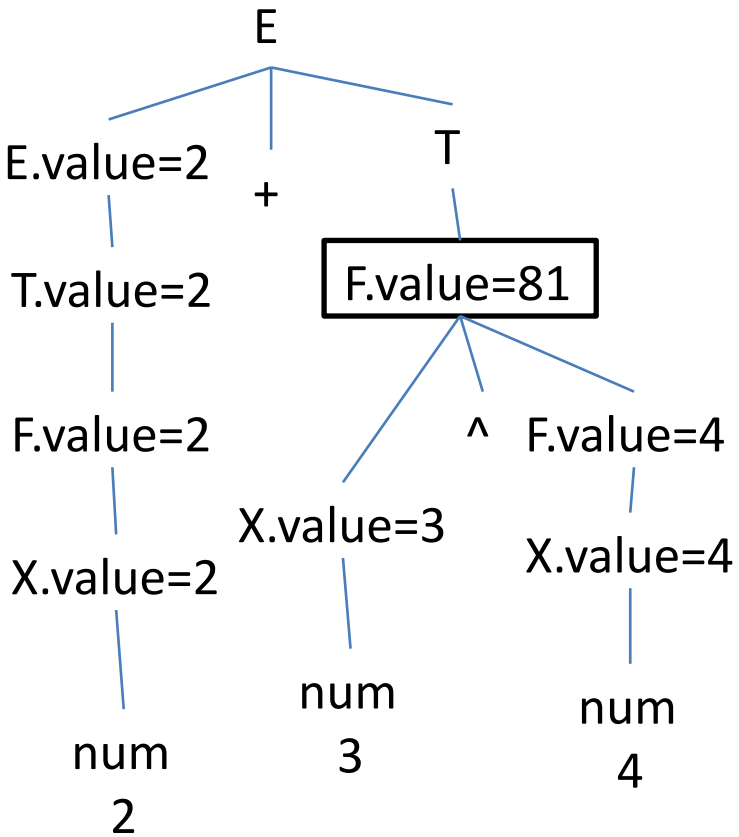
For input, 2 + 3 ^ 4



SDT for desktop calculator

$E \rightarrow E_1 + T$	$\{E.value = E_1.value + T.value\}$
$E \rightarrow T$	$\{E.value = T.value\}$
$T \rightarrow T_1 * F$	$\{T.value = T_1.value * F.value\}$
$T \rightarrow F$	$\{T.value = F.value\}$
$F \rightarrow X \wedge F_1$	$\{F.value = X.value \wedge F_1.value\}$
$F \rightarrow X$	$\{F.value = X.value\}$
$X \rightarrow num$	$\{X.value = num.lexvalue\}$

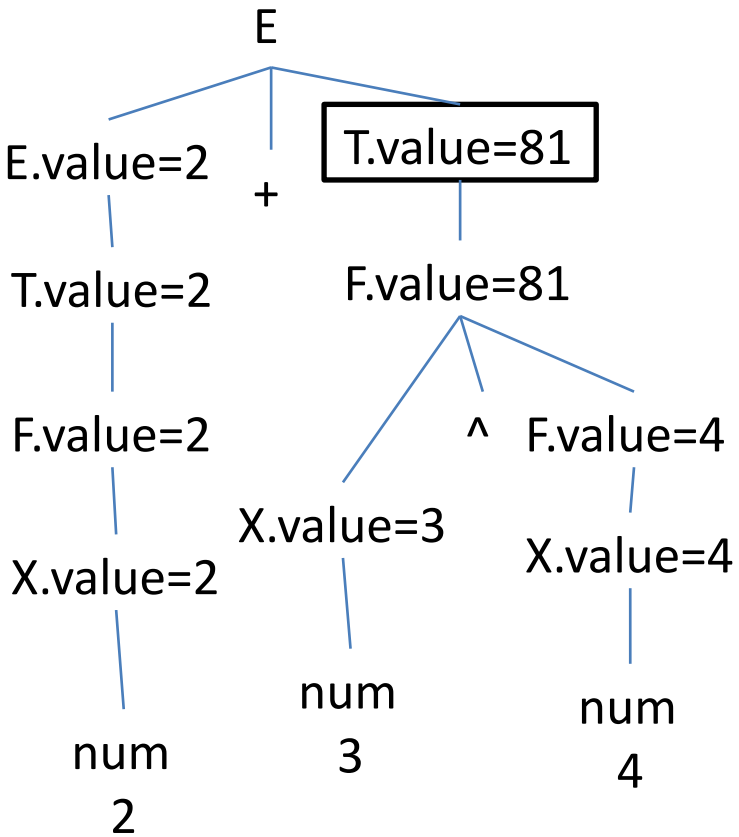
For input, 2 + 3 ^ 4



SDT for desktop calculator

$E \rightarrow E_1 + T$	$\{E.value = E_1.value + T.value\}$
$E \rightarrow T$	$\{E.value = T.value\}$
$T \rightarrow T_1 * F$	$\{T.value = T_1.value * F.value\}$
$T \rightarrow F$	$\{T.value = F.value\}$
$F \rightarrow X \wedge F_1$	$\{F.value = X.value \wedge F_1.value\}$
$F \rightarrow X$	$\{F.value = X.value\}$
$X \rightarrow num$	$\{X.value = num.lexvalue\}$

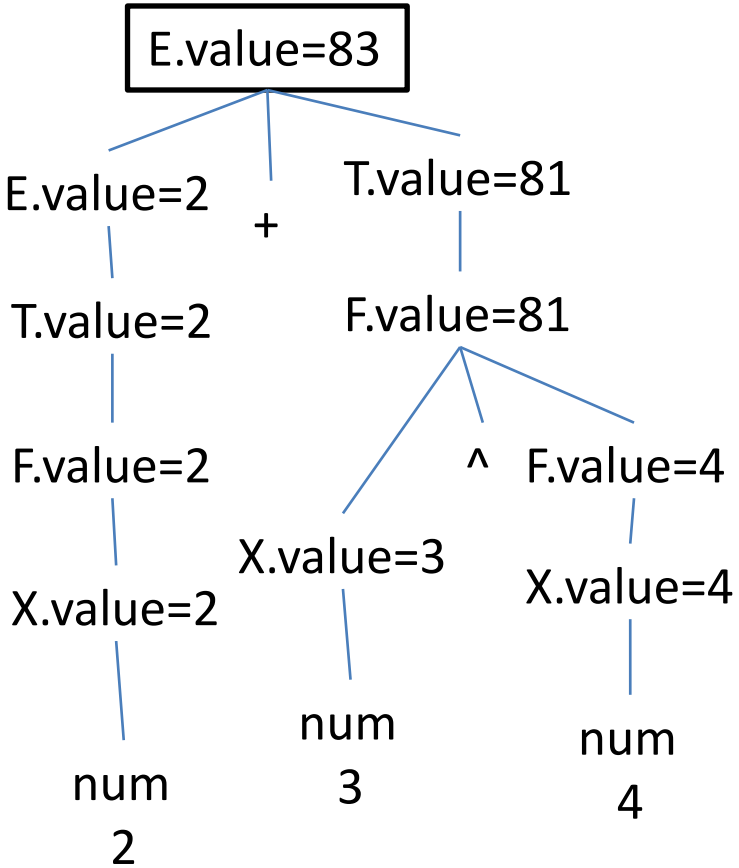
For input, 2 + 3 ^ 4



SDT for desktop calculator

$E \rightarrow E_1 + T$	$\{E.value = E_1.value + T.value\}$
$E \rightarrow T$	$\{E.value = T.value\}$
$T \rightarrow T_1 * F$	$\{T.value = T_1.value * F.value\}$
$T \rightarrow F$	$\{T.value = F.value\}$
$F \rightarrow X \wedge F_1$	$\{F.value = X.value \wedge F_1.value\}$
$F \rightarrow X$	$\{F.value = X.value\}$
$X \rightarrow num$	$\{X.value = num.lexvalue\}$

For input, 2 + 3 ^ 4
Output: 83



SDT for desktop calculator

$E \rightarrow E_1 + T$	$\{E.value = E_1.value + T.value\}$
$E \rightarrow T$	$\{E.value = T.value\}$
$T \rightarrow T_1 * F$	$\{T.value = T_1.value * F.value\}$
$T \rightarrow F$	$\{T.value = F.value\}$
$F \rightarrow X \wedge F_1$	$\{F.value = X.value \wedge F_1.value\}$
$F \rightarrow X$	$\{F.value = X.value\}$
$X \rightarrow num$	$\{X.value = num.lexvalue\}$

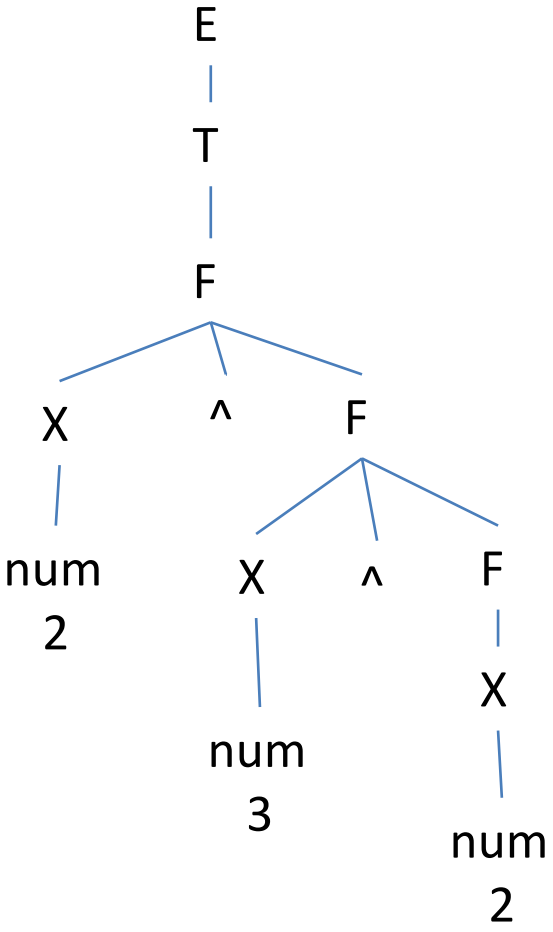
Parse tree???

For input, 2 ^ 3 ^ 2

SDT for desktop calculator

$E \rightarrow E_1 + T$	$\{E.value = E_1.value + T.value\}$
$E \rightarrow T$	$\{E.value = T.value\}$
$T \rightarrow T_1 * F$	$\{T.value = T_1.value * F.value\}$
$T \rightarrow F$	$\{T.value = F.value\}$
$F \rightarrow X \wedge F_1$	$\{F.value = X.value \wedge F_1.value\}$
$F \rightarrow X$	$\{F.value = X.value\}$
$X \rightarrow num$	$\{X.value = num.lexvalue\}$

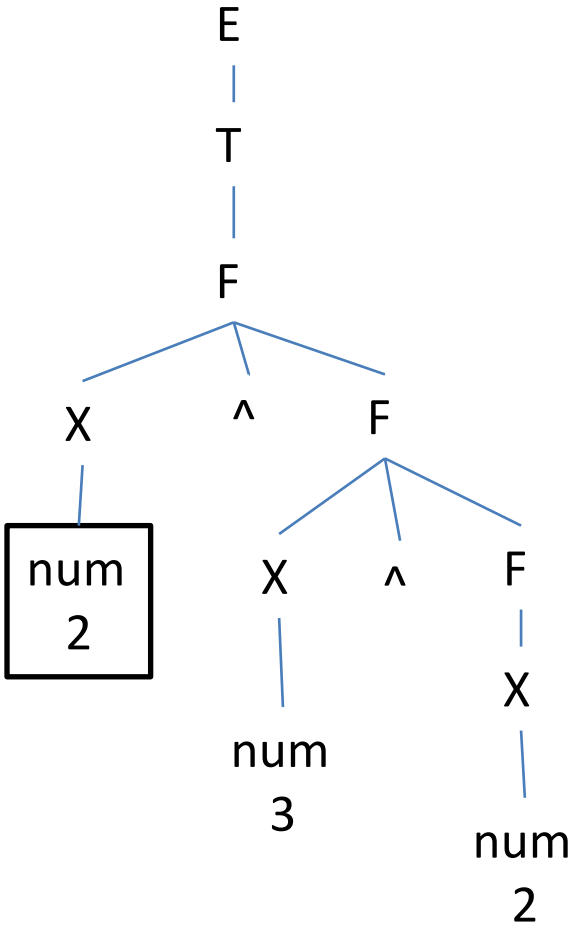
For input, 2 ^ 3 ^ 2



SDT for desktop calculator

$E \rightarrow E_1 + T$	$\{E.value = E_1.value + T.value\}$
$E \rightarrow T$	$\{E.value = T.value\}$
$T \rightarrow T_1 * F$	$\{T.value = T_1.value * F.value\}$
$T \rightarrow F$	$\{T.value = F.value\}$
$F \rightarrow X \wedge F_1$	$\{F.value = X.value \wedge F_1.value\}$
$F \rightarrow X$	$\{F.value = X.value\}$
$X \rightarrow num$	$\{X.value = num.lexvalue\}$

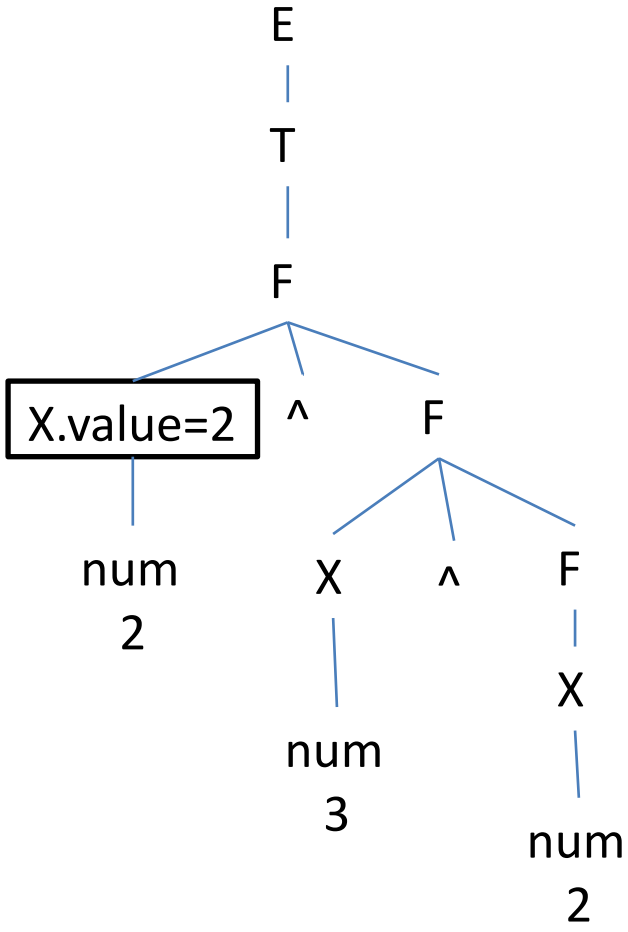
For input, 2 ^ 3 ^ 2



SDT for desktop calculator

$E \rightarrow E_1 + T$	$\{E.value = E_1.value + T.value\}$
$E \rightarrow T$	$\{E.value = T.value\}$
$T \rightarrow T_1 * F$	$\{T.value = T_1.value * F.value\}$
$T \rightarrow F$	$\{T.value = F.value\}$
$F \rightarrow X \wedge F_1$	$\{F.value = X.value \wedge F_1.value\}$
$F \rightarrow X$	$\{F.value = X.value\}$
$X \rightarrow num$	$\{X.value = num.lexvalue\}$

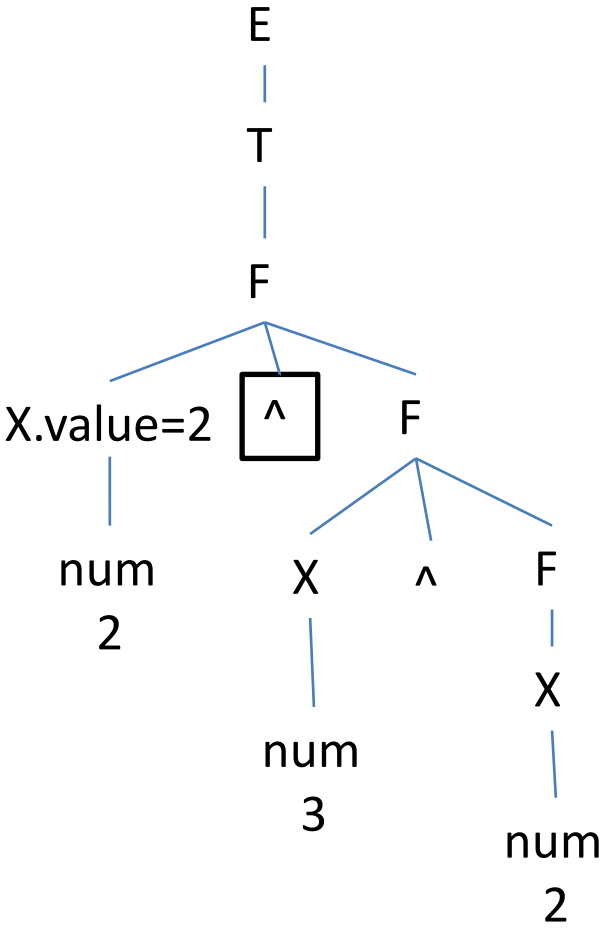
For input, 2 ^ 3 ^ 2



SDT for desktop calculator

$E \rightarrow E_1 + T$	$\{E.value = E_1.value + T.value\}$
$E \rightarrow T$	$\{E.value = T.value\}$
$T \rightarrow T_1 * F$	$\{T.value = T_1.value * F.value\}$
$T \rightarrow F$	$\{T.value = F.value\}$
$F \rightarrow X \wedge F_1$	$\{F.value = X.value \wedge F_1.value\}$
$F \rightarrow X$	$\{F.value = X.value\}$
$X \rightarrow num$	$\{X.value = num.lexvalue\}$

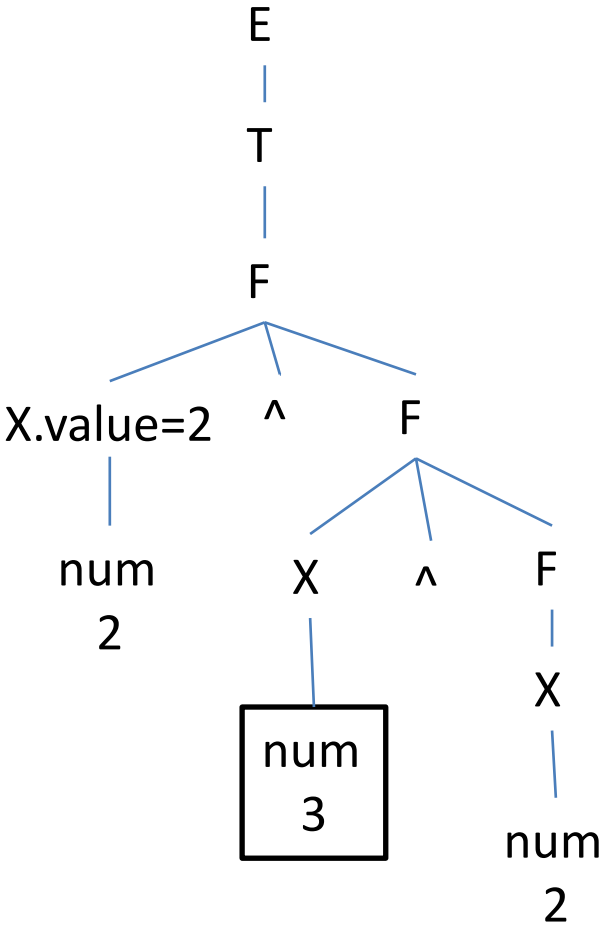
For input, 2 ^ 3 ^ 2



SDT for desktop calculator

$E \rightarrow E_1 + T$	$\{E.value = E_1.value + T.value\}$
$E \rightarrow T$	$\{E.value = T.value\}$
$T \rightarrow T_1 * F$	$\{T.value = T_1.value * F.value\}$
$T \rightarrow F$	$\{T.value = F.value\}$
$F \rightarrow X \wedge F_1$	$\{F.value = X.value \wedge F_1.value\}$
$F \rightarrow X$	$\{F.value = X.value\}$
$X \rightarrow num$	$\{X.value = num.lexvalue\}$

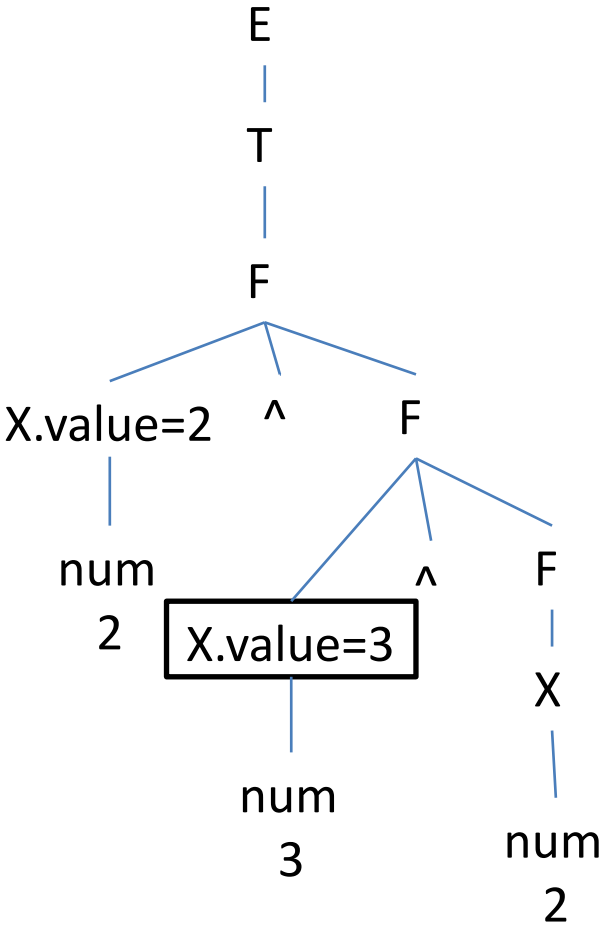
For input, 2 ^ 3 ^ 2



SDT for desktop calculator

$E \rightarrow E_1 + T$	$\{E.value = E_1.value + T.value\}$
$E \rightarrow T$	$\{E.value = T.value\}$
$T \rightarrow T_1 * F$	$\{T.value = T_1.value * F.value\}$
$T \rightarrow F$	$\{T.value = F.value\}$
$F \rightarrow X \wedge F_1$	$\{F.value = X.value \wedge F_1.value\}$
$F \rightarrow X$	$\{F.value = X.value\}$
$X \rightarrow num$	$\{X.value = num.lexvalue\}$

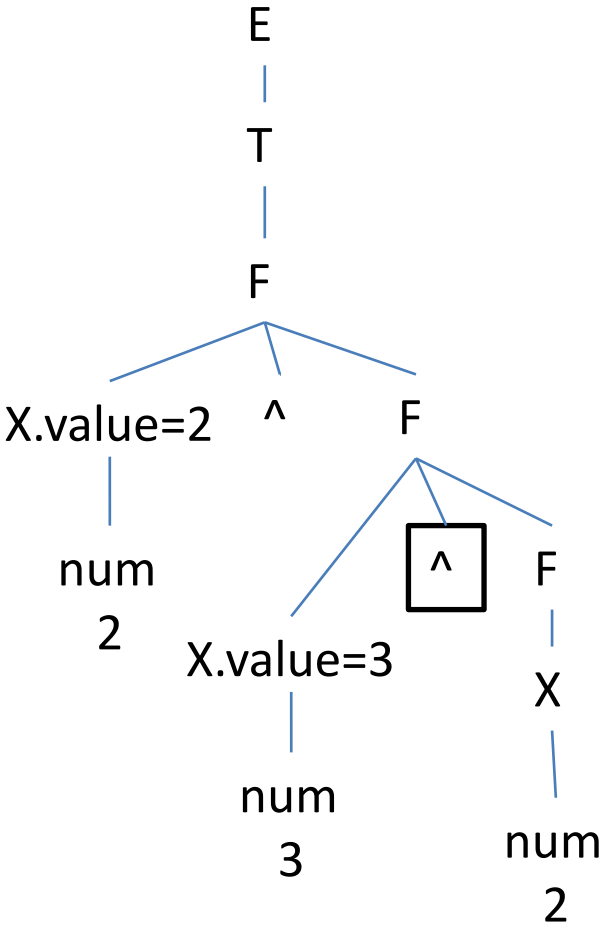
For input, 2 ^ 3 ^ 2



SDT for desktop calculator

$E \rightarrow E_1 + T$	$\{E.value = E_1.value + T.value\}$
$E \rightarrow T$	$\{E.value = T.value\}$
$T \rightarrow T_1 * F$	$\{T.value = T_1.value * F.value\}$
$T \rightarrow F$	$\{T.value = F.value\}$
$F \rightarrow X \wedge F_1$	$\{F.value = X.value \wedge F_1.value\}$
$F \rightarrow X$	$\{F.value = X.value\}$
$X \rightarrow num$	$\{X.value = num.lexvalue\}$

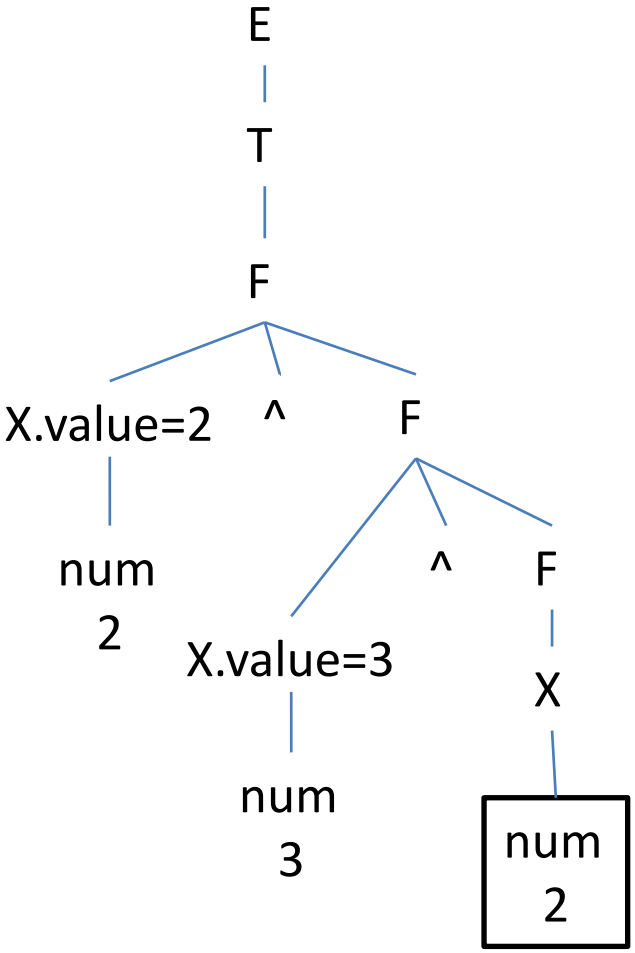
For input, 2 ^ 3 ^ 2



SDT for desktop calculator

$E \rightarrow E_1 + T$	$\{E.value = E_1.value + T.value\}$
$E \rightarrow T$	$\{E.value = T.value\}$
$T \rightarrow T_1 * F$	$\{T.value = T_1.value * F.value\}$
$T \rightarrow F$	$\{T.value = F.value\}$
$F \rightarrow X \wedge F_1$	$\{F.value = X.value \wedge F_1.value\}$
$F \rightarrow X$	$\{F.value = X.value\}$
$X \rightarrow num$	$\{X.value = num.lexvalue\}$

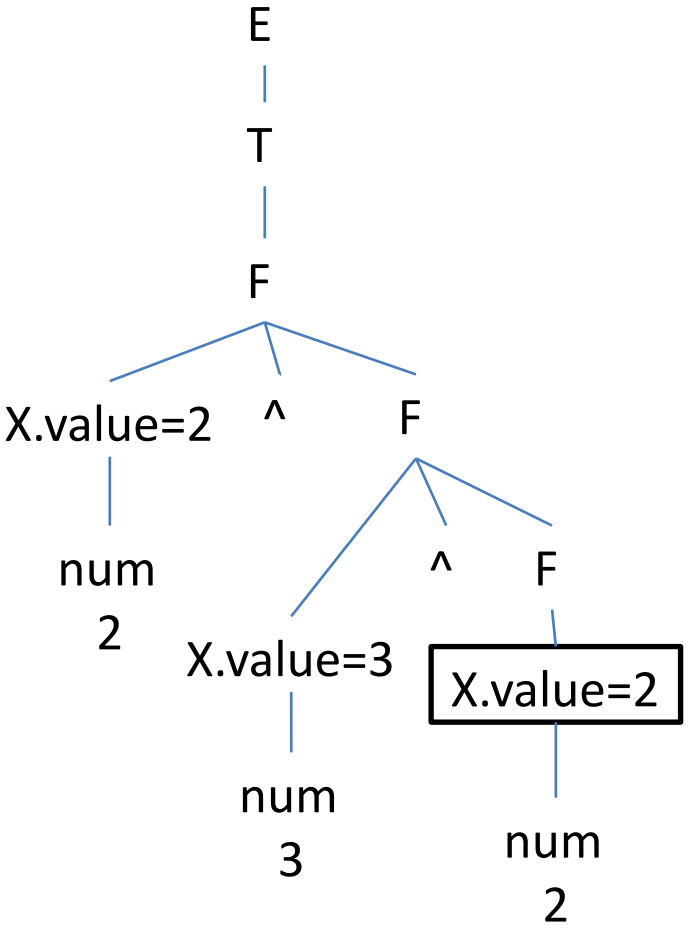
For input, 2 ^ 3 ^ 2



SDT for desktop calculator

$E \rightarrow E_1 + T$	$\{E.value = E_1.value + T.value\}$
$E \rightarrow T$	$\{E.value = T.value\}$
$T \rightarrow T_1 * F$	$\{T.value = T_1.value * F.value\}$
$T \rightarrow F$	$\{T.value = F.value\}$
$F \rightarrow X \wedge F_1$	$\{F.value = X.value \wedge F_1.value\}$
$F \rightarrow X$	$\{F.value = X.value\}$
$X \rightarrow num$	$\{X.value = num.lexvalue\}$

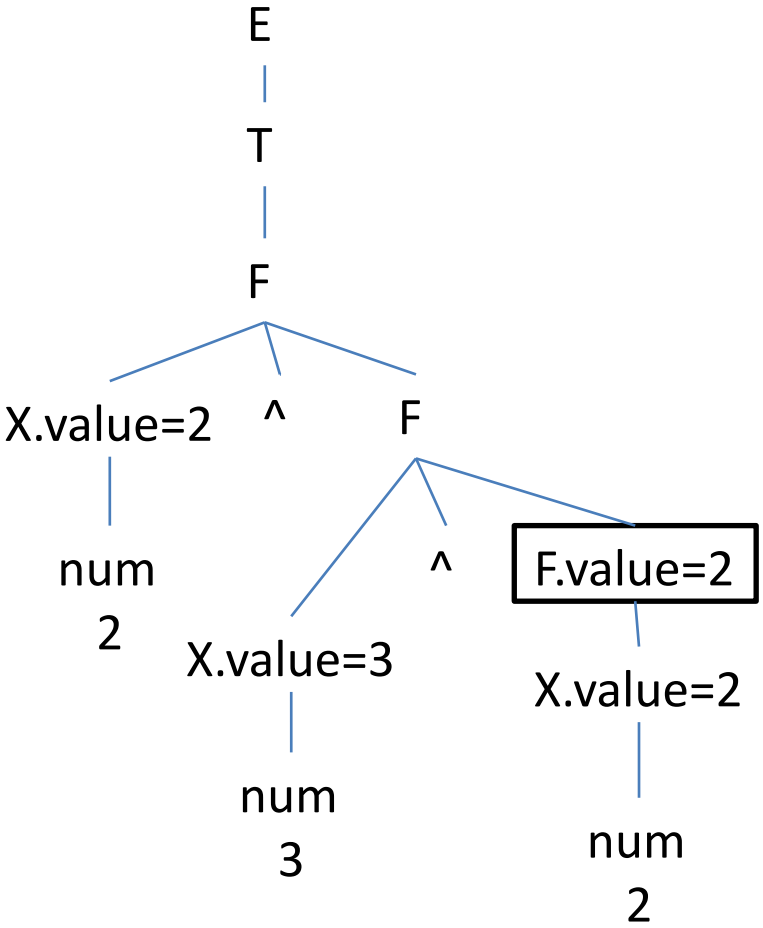
For input, 2 ^ 3 ^ 2



SDT for desktop calculator

$E \rightarrow E_1 + T$	$\{E.value = E_1.value + T.value\}$
$E \rightarrow T$	$\{E.value = T.value\}$
$T \rightarrow T_1 * F$	$\{T.value = T_1.value * F.value\}$
$T \rightarrow F$	$\{T.value = F.value\}$
$F \rightarrow X \wedge F_1$	$\{F.value = X.value \wedge F_1.value\}$
$F \rightarrow X$	$\{F.value = X.value\}$
$X \rightarrow num$	$\{X.value = num.lexvalue\}$

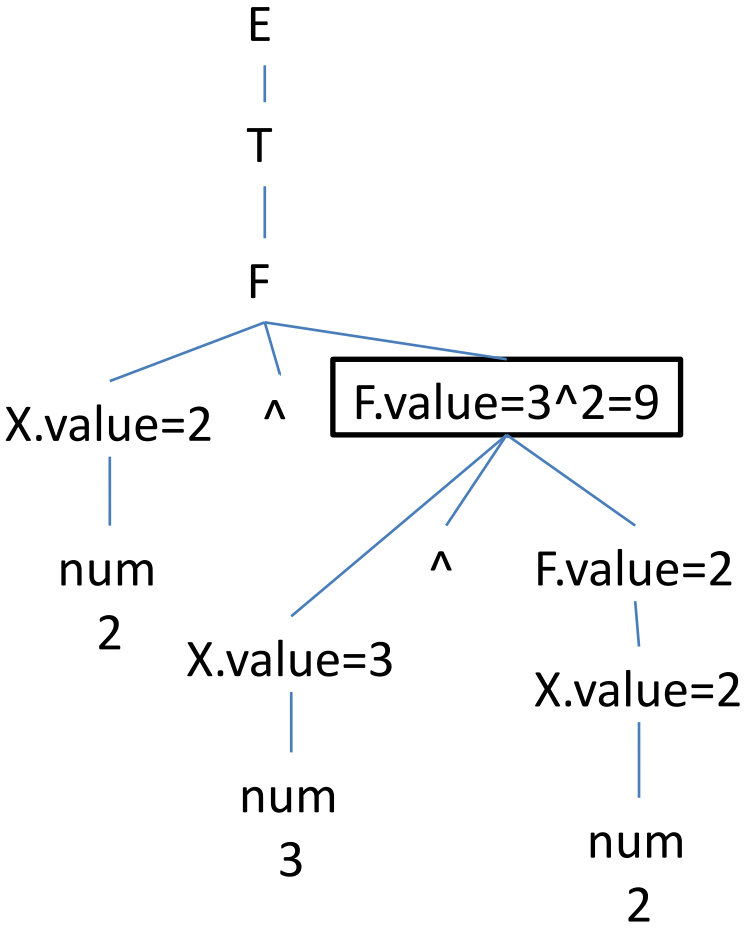
For input, 2 ^ 3 ^ 2



SDT for desktop calculator

$E \rightarrow E_1 + T$	$\{E.value = E_1.value + T.value\}$
$E \rightarrow T$	$\{E.value = T.value\}$
$T \rightarrow T_1 * F$	$\{T.value = T_1.value * F.value\}$
$T \rightarrow F$	$\{T.value = F.value\}$
$F \rightarrow X \wedge F_1$	$\{F.value = X.value \wedge F_1.value\}$
$F \rightarrow X$	$\{F.value = X.value\}$
$X \rightarrow num$	$\{X.value = num.lexvalue\}$

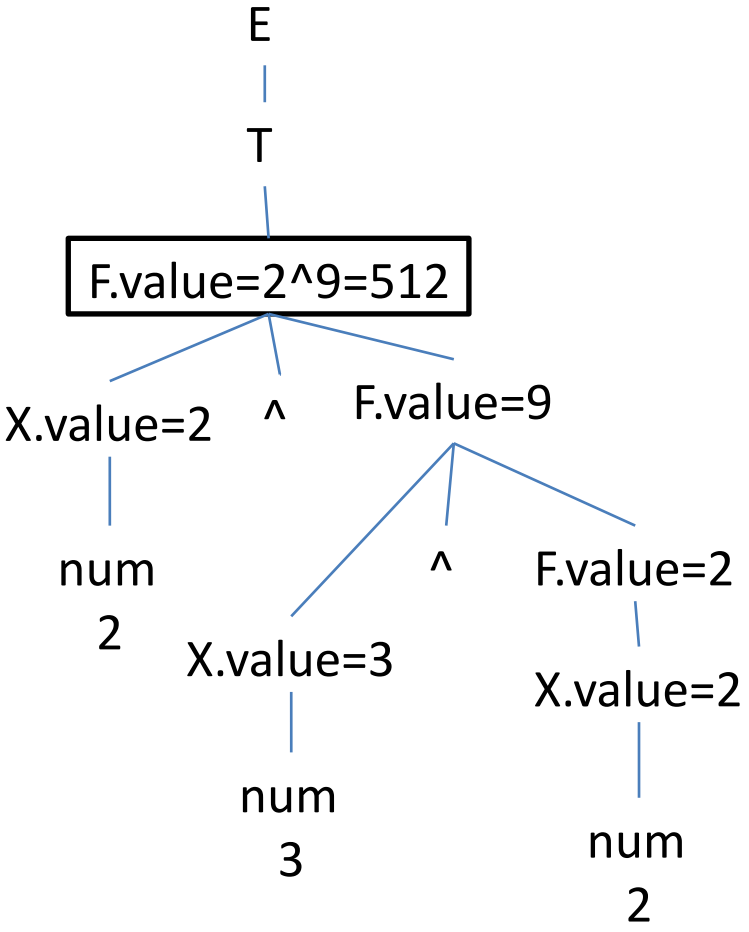
For input, 2 ^ 3 ^ 2



SDT for desktop calculator

$E \rightarrow E_1 + T$	$\{E.value = E_1.value + T.value\}$
$E \rightarrow T$	$\{E.value = T.value\}$
$T \rightarrow T_1 * F$	$\{T.value = T_1.value * F.value\}$
$T \rightarrow F$	$\{T.value = F.value\}$
$F \rightarrow X \wedge F_1$	$\{F.value = X.value \wedge F_1.value\}$
$F \rightarrow X$	$\{F.value = X.value\}$
$X \rightarrow num$	$\{X.value = num.lexvalue\}$

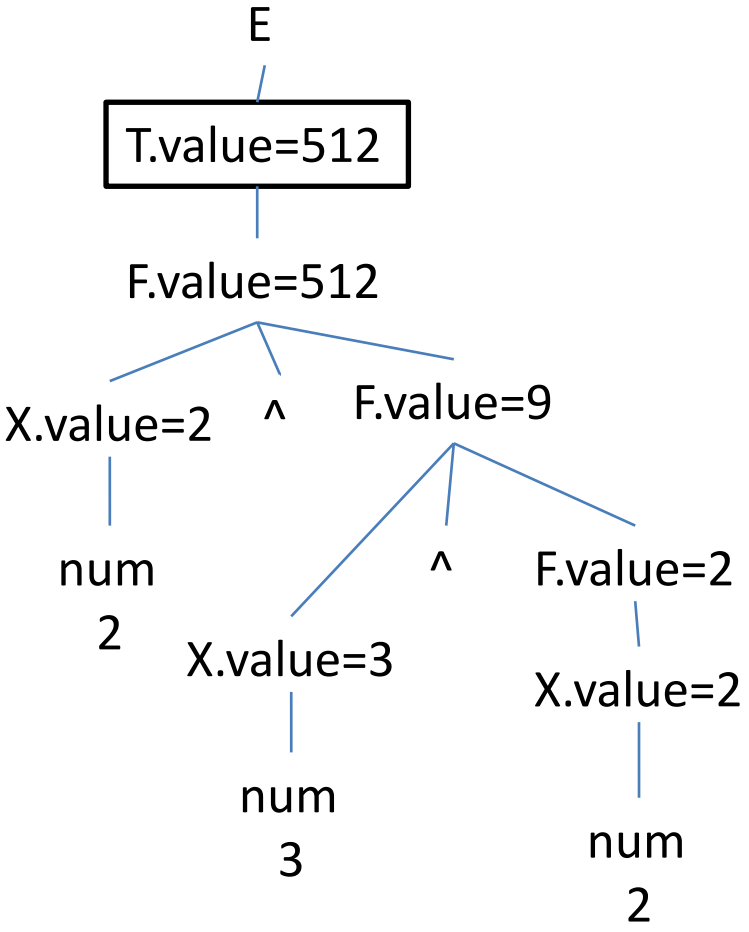
For input, 2 ^ 3 ^ 2



SDT for desktop calculator

$E \rightarrow E_1 + T$	$\{E.value = E_1.value + T.value\}$
$E \rightarrow T$	$\{E.value = T.value\}$
$T \rightarrow T_1 * F$	$\{T.value = T_1.value * F.value\}$
$T \rightarrow F$	$\{T.value = F.value\}$
$F \rightarrow X \wedge F_1$	$\{F.value = X.value \wedge F_1.value\}$
$F \rightarrow X$	$\{F.value = X.value\}$
$X \rightarrow num$	$\{X.value = num.lexvalue\}$

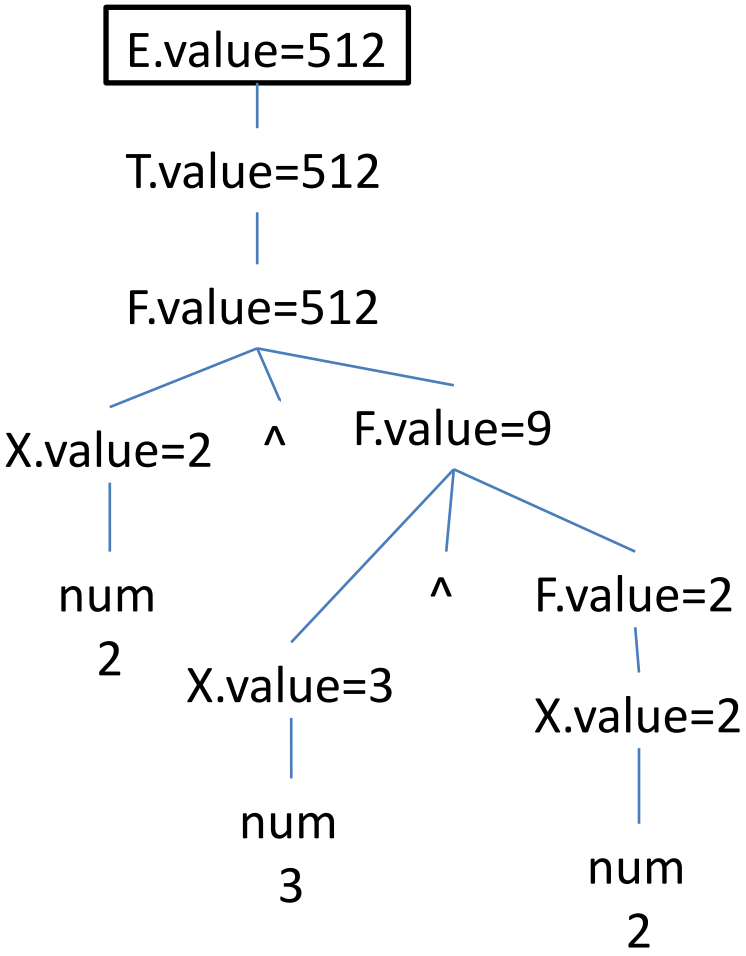
For input, 2 ^ 3 ^ 2



SDT for desktop calculator

$E \rightarrow E_1 + T$	$\{E.value = E_1.value + T.value\}$
$E \rightarrow T$	$\{E.value = T.value\}$
$T \rightarrow T_1 * F$	$\{T.value = T_1.value * F.value\}$
$T \rightarrow F$	$\{T.value = F.value\}$
$F \rightarrow X \wedge F_1$	$\{F.value = X.value \wedge F_1.value\}$
$F \rightarrow X$	$\{F.value = X.value\}$
$X \rightarrow num$	$\{X.value = num.lexvalue\}$

For input, 2 ^ 3 ^ 2
Output: 512



Example 4

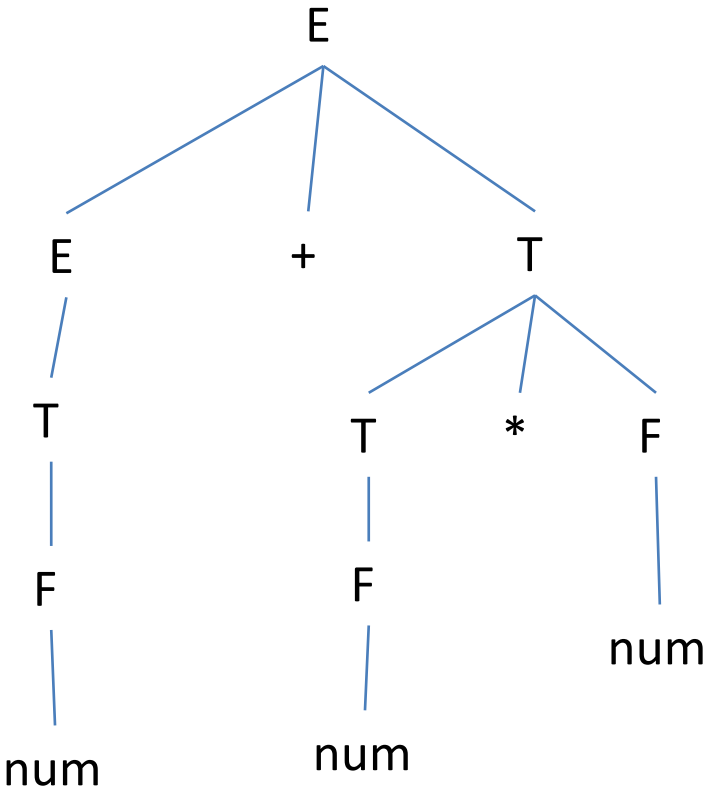
- Write SDT to convert infix to postfix

For input, 2 + 3 * 4
Output: 2 3 4 * +

Example 4

- SDT to convert infix to postfix

E→ E + T	{printf("+");}
E → T	{}
T → T * F	{printf("*");}
T→F	{}
F→ num	{printf(num.lval);}

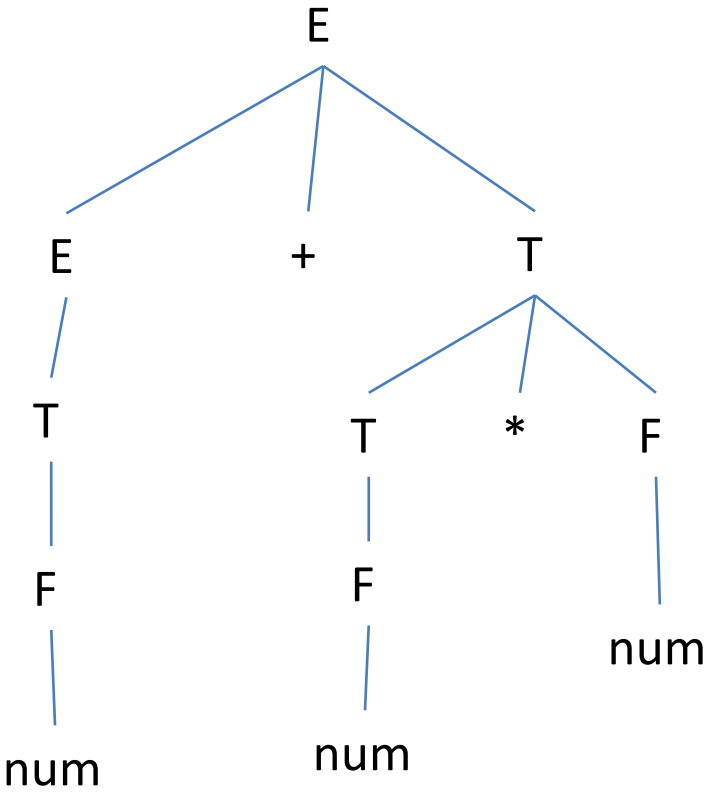


For input, 2 + 3 * 4

Example 4

- SDT to convert infix to postfix

E → E + T	{printf("+");}	1
E → T	{}	2
T → T * F	{printf("*");}	3
T → F	{}	4
F → num	{printf(num.lval);}	5



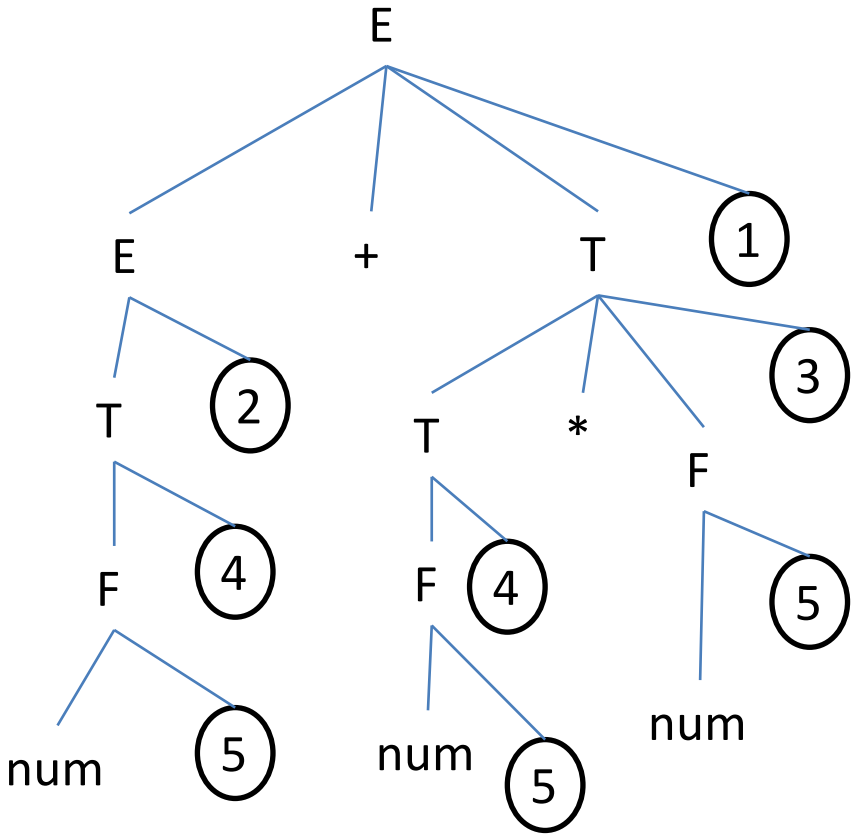
For input, 2 + 3 * 4

Example 4 (taking top-down approach)

- SDT to convert infix to postfix

E→ E + T	{printf("+");}	1
E → T	{}	2
T → T * F	{printf("*");}	3
T→F	{}	4
F→ num	{printf(num.lval);}	5

For input, 2 + 3 * 4

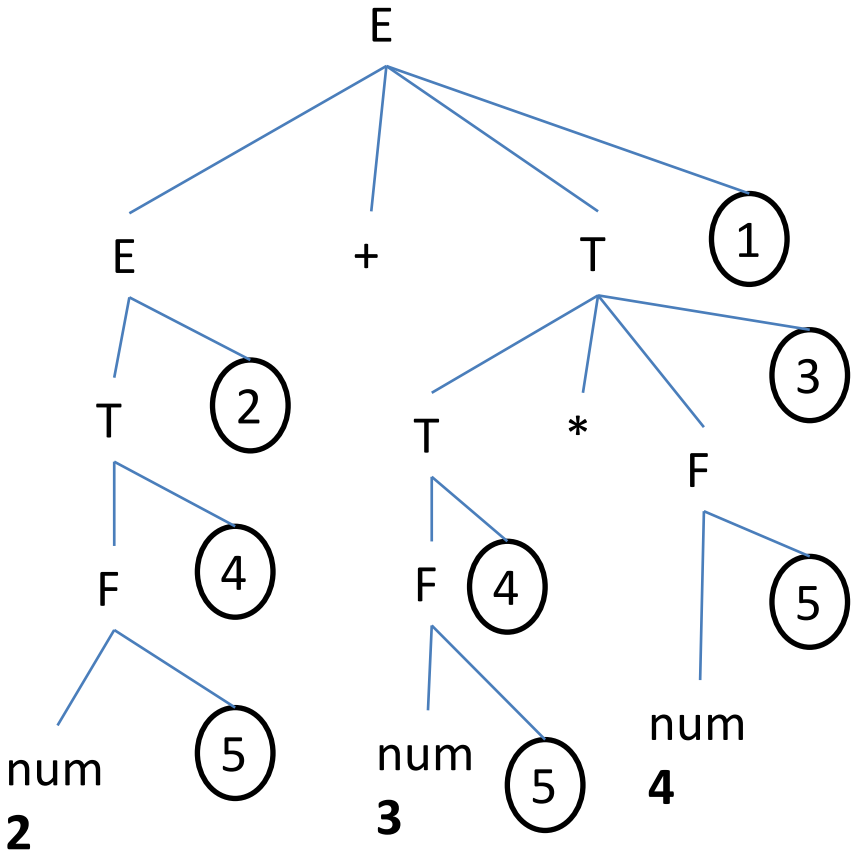


Example 4 (taking top-down approach)

- SDT to convert infix to postfix

E → E + T	{printf("+");}	1
E → T	{}	2
T → T * F	{printf("*");}	3
T → F	{}	4
F → num	{printf(num.lval);}	5

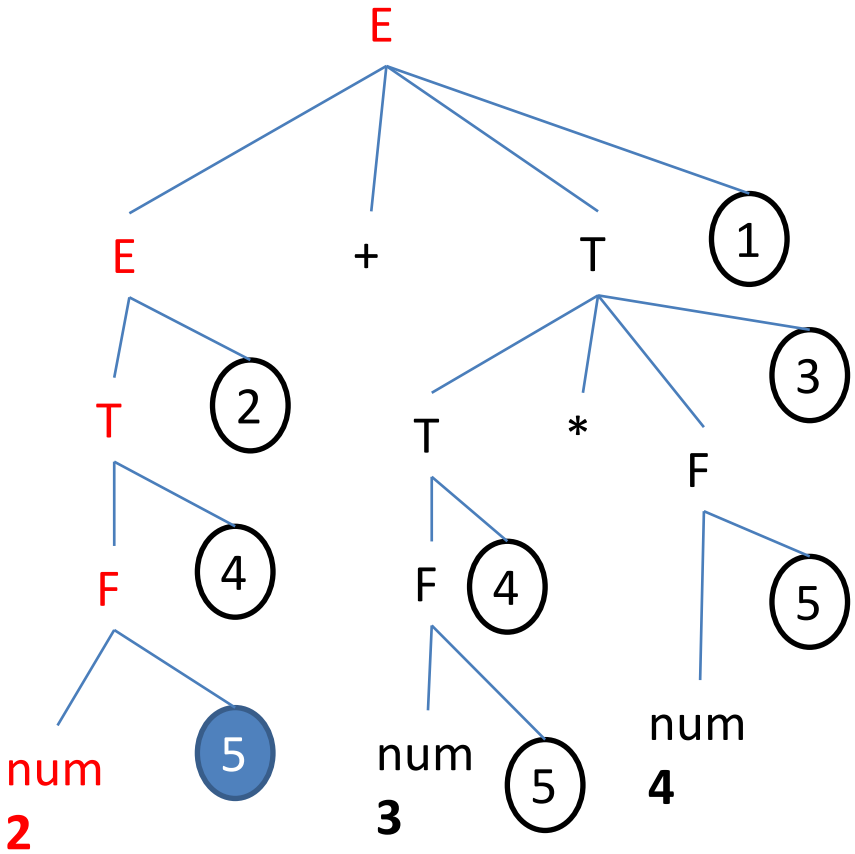
For input, 2 + 3 * 4



Example 4 (taking top-down approach)

- SDT to convert infix to postfix

E → E + T	{printf("+");}	1
E → T	{}	2
T → T * F	{printf("*");}	3
T → F	{}	4
F → num	{printf(num.lval);}	5



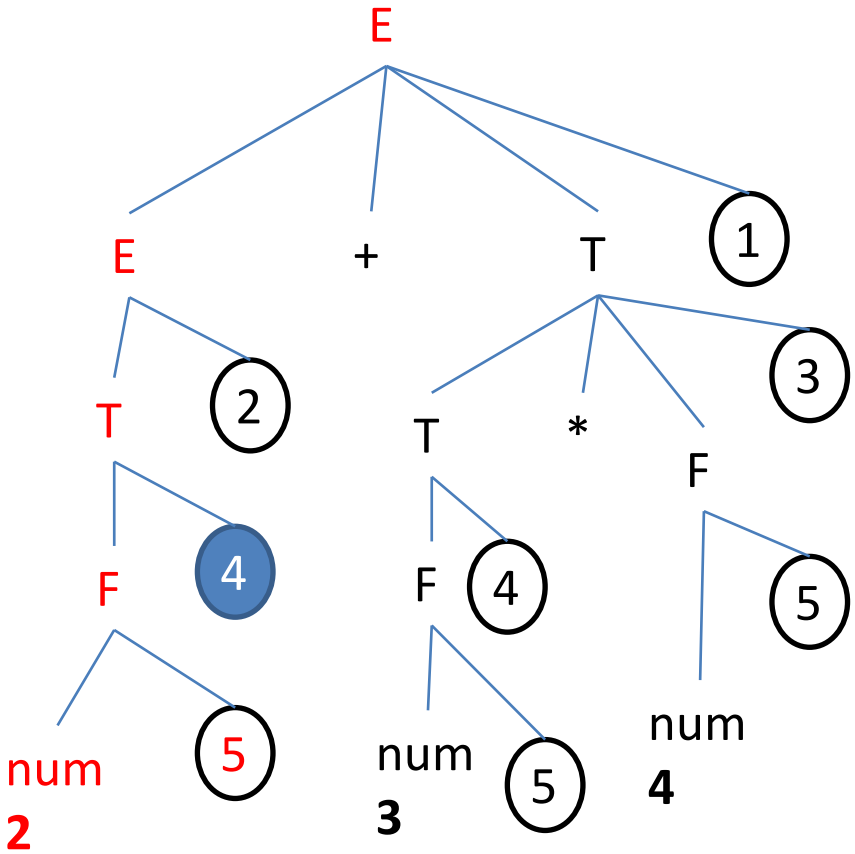
For input, 2 + 3 * 4

Output: 2

Example 4 (taking top-down approach)

- SDT to convert infix to postfix

E → E + T	{printf("+");}	1
E → T	{}	2
T → T * F	{printf("*");}	3
T → F	{}	4
F → num	{printf(num.lval);}	5



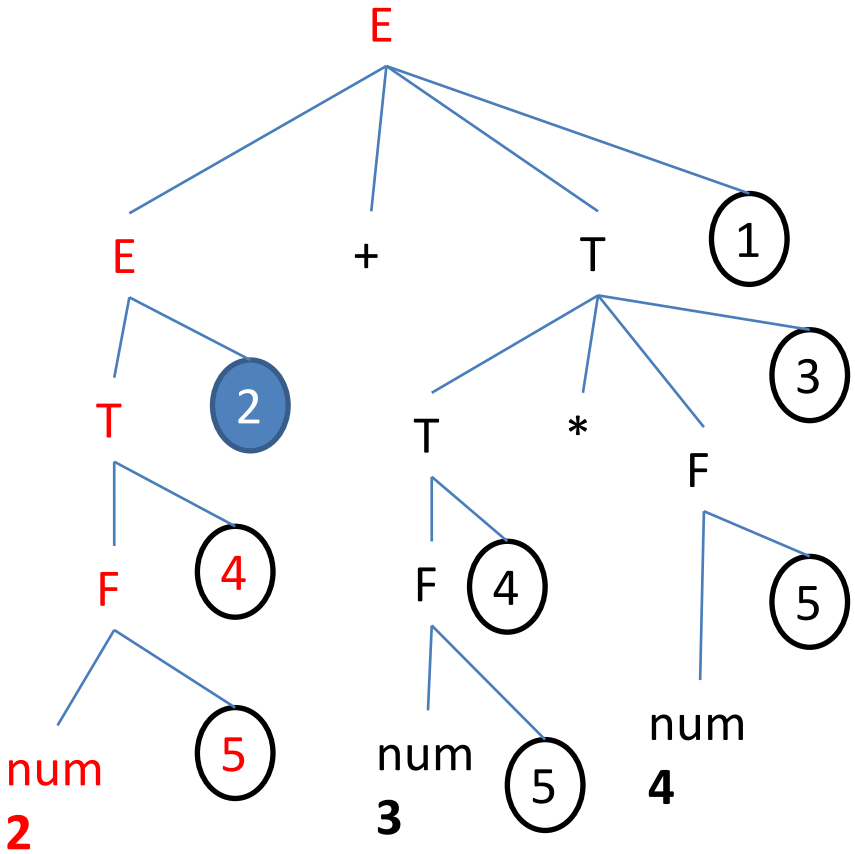
For input, 2 + 3 * 4

Output: 2

Example 4 (taking top-down approach)

- SDT to convert infix to postfix

E → E + T	{printf("+");}	1
E → T	{}	2
T → T * F	{printf("*");}	3
T → F	{}	4
F → num	{printf(num.lval);}	5



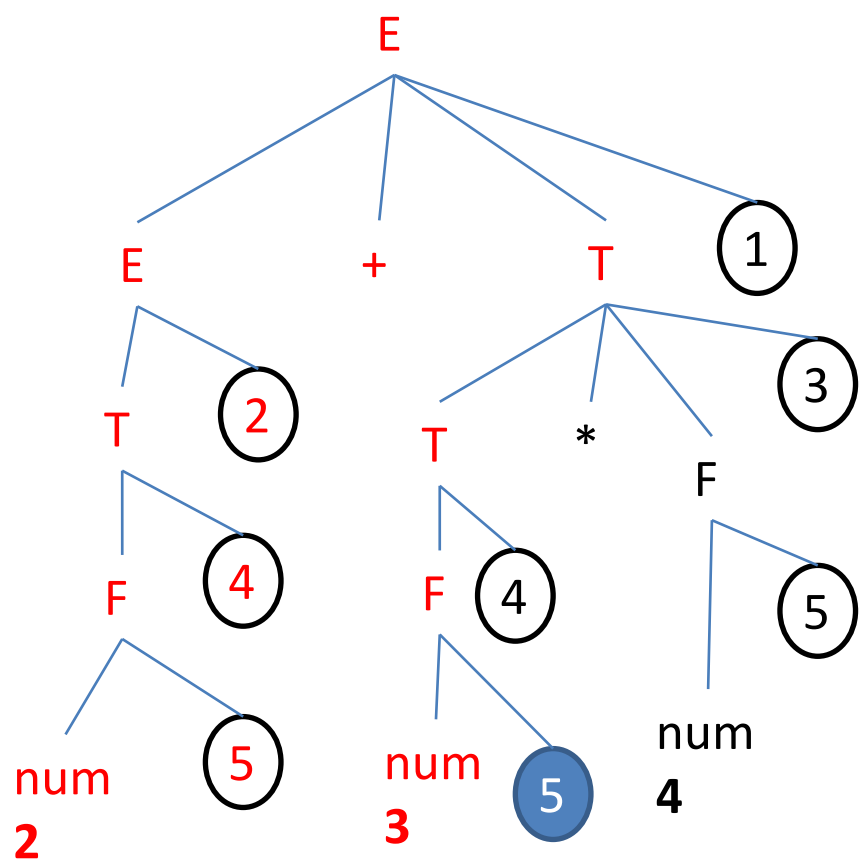
For input, 2 + 3 * 4

Output: 2

Example 4 (taking top-down approach)

- SDT to convert infix to postfix

E→ E + T	{printf("+");}	1
E → T	{}	2
T → T * F	{printf("*");}	3
T→F	{}	4
F→ num	{printf(num.lval);}	5



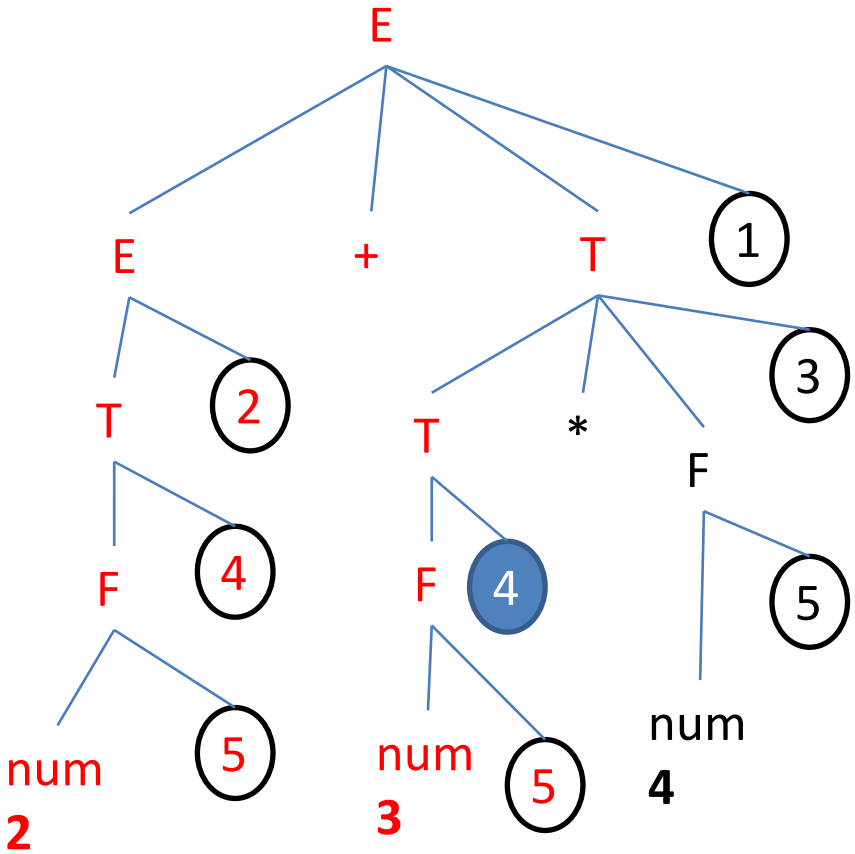
For input, 2 + 3 * 4

Output: 2 3

Example 4 (taking top-down approach)

- SDT to convert infix to postfix

E→ E + T	{printf("+");}	1
E → T	{}	2
T → T * F	{printf("*");}	3
T→F	{}	4
F→ num	{printf(num.lval);}	5



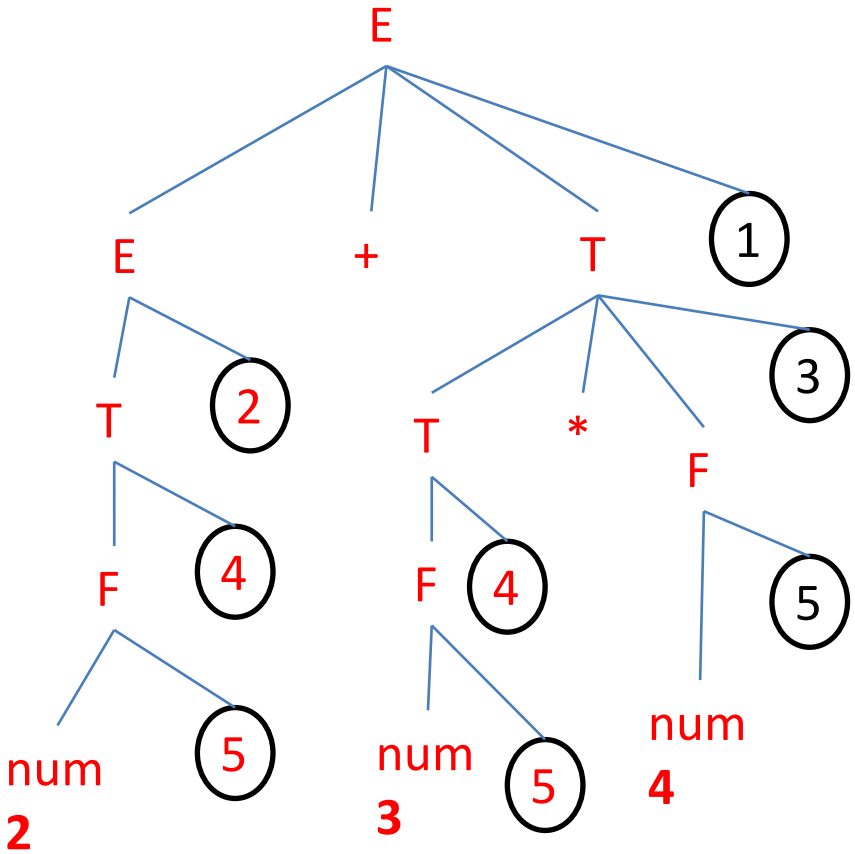
For input, 2 + 3 * 4

Output: 2 3

Example 4 (taking top-down approach)

- SDT to convert infix to postfix

E→ E + T	{printf("+");}	1
E → T	{}	2
T → T * F	{printf("*");}	3
T→F	{}	4
F→ num	{printf(num.lval);}	5



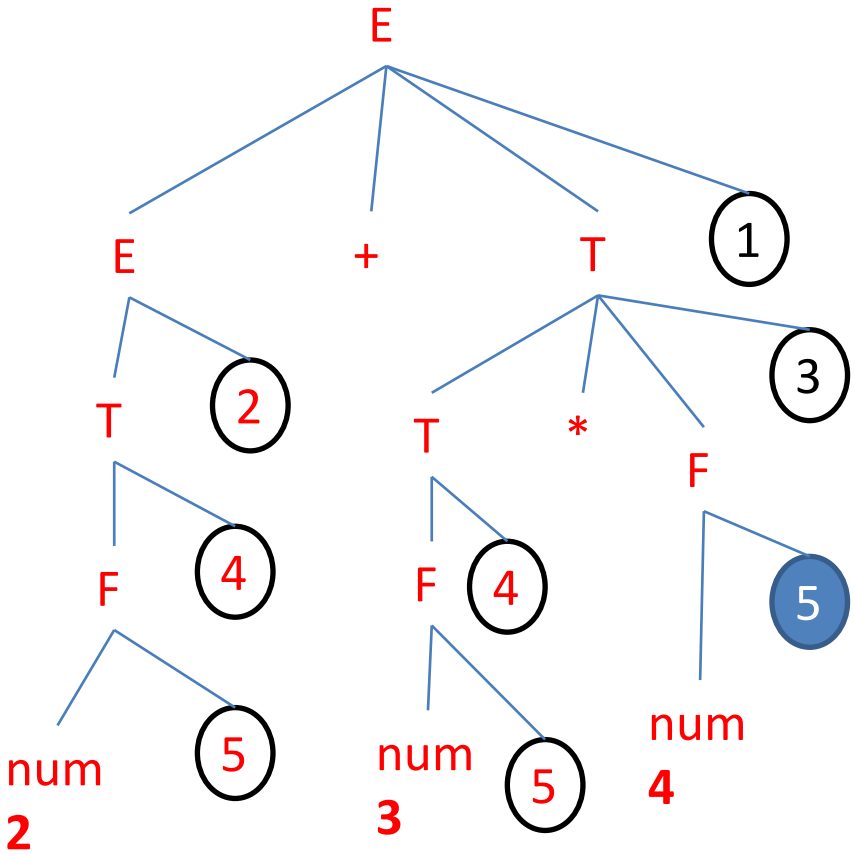
For input, 2 + 3 * 4

Output: 2 3

Example 4 (taking top-down approach)

- SDT to convert infix to postfix

E → E + T	{printf("+");}	1
E → T	{}	2
T → T * F	{printf("*");}	3
T → F	{}	4
F → num	{printf(num.lval);}	5



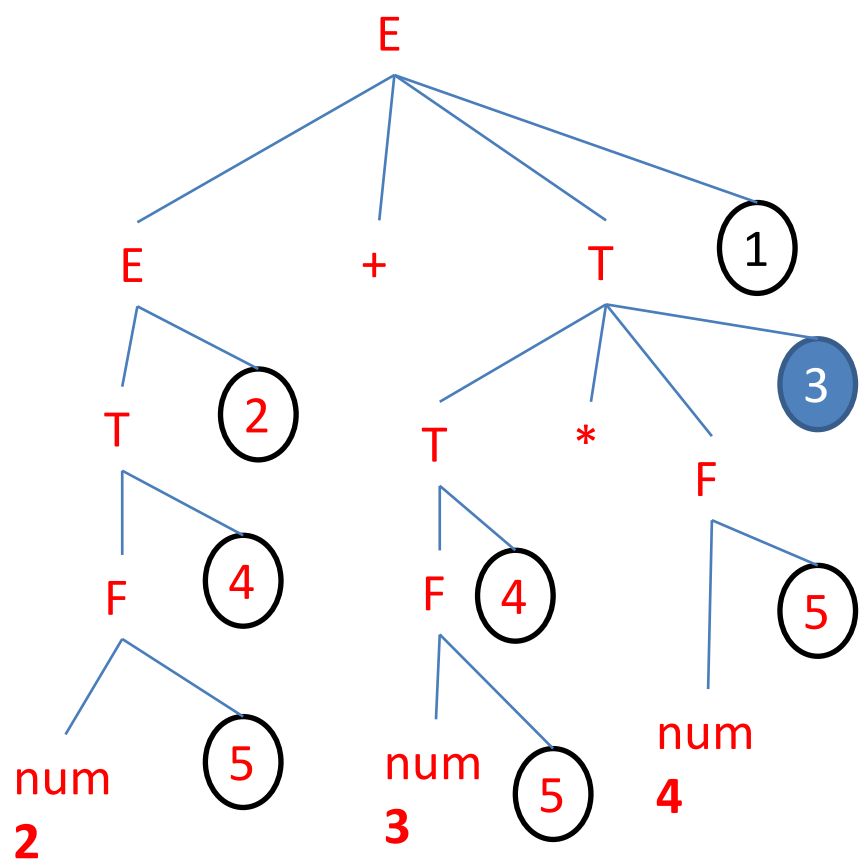
For input, 2 + 3 * 4

Output: 2 3 4

Example 4 (taking top-down approach)

- SDT to convert infix to postfix

E → E + T	{printf("+");}	1
E → T	{}	2
T → T * F	{printf("*");}	3
T → F	{}	4
F → num	{printf(num.lval);}	5



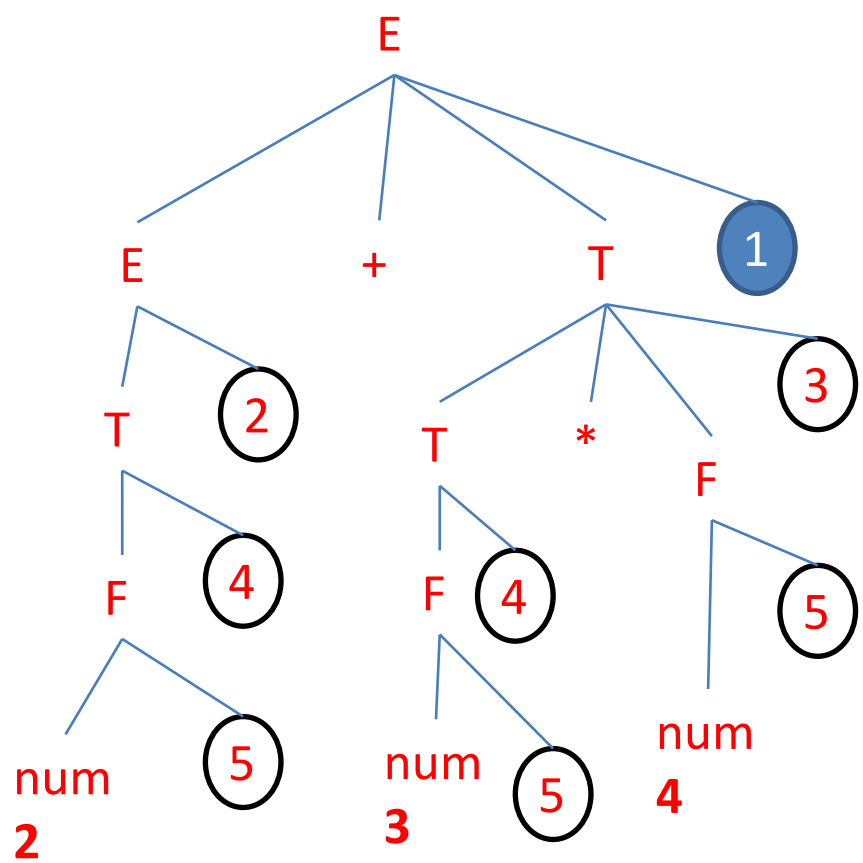
For input, 2 + 3 * 4

Output: 2 3 4 *

Example 4 (taking top-down approach)

- SDT to convert infix to postfix

E → E + T	{printf("+");}	1
E → T	{}	2
T → T * F	{printf("*");}	3
T → F	{}	4
F → num	{printf(num.lval);}	5



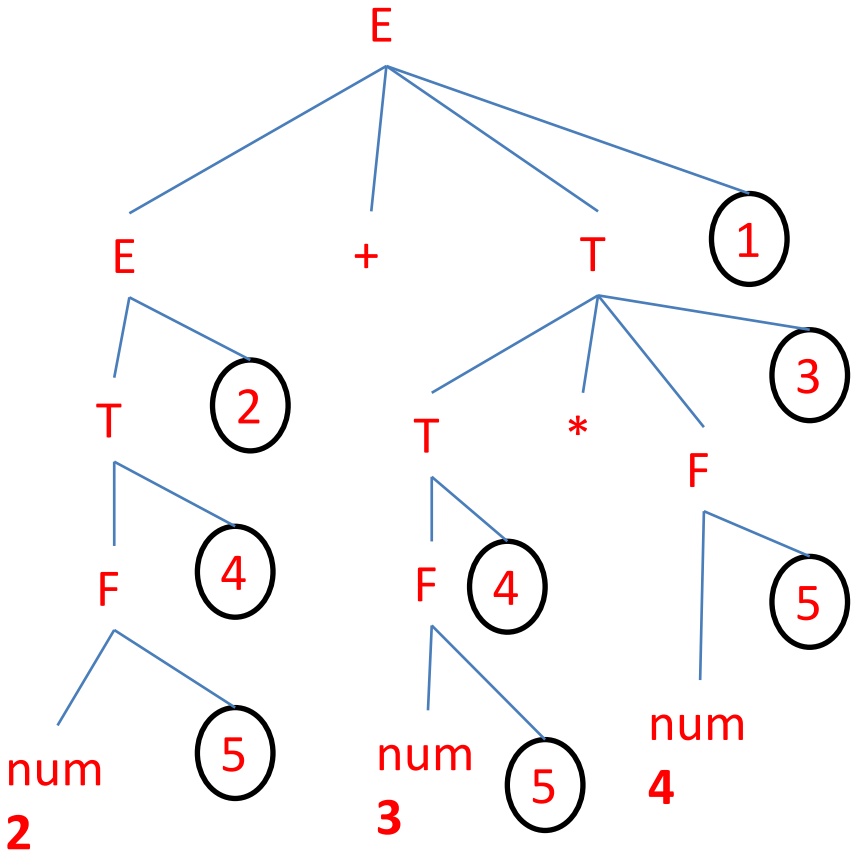
For input, 2 + 3 * 4

Output: 2 3 4 * +

Example 4 (taking top-down approach)

- SDT to convert infix to postfix

E → E + T	{printf("+");}	1
E → T	{}	2
T → T * F	{printf("*");}	3
T → F	{}	4
F → num	{printf(num.lval);}	5



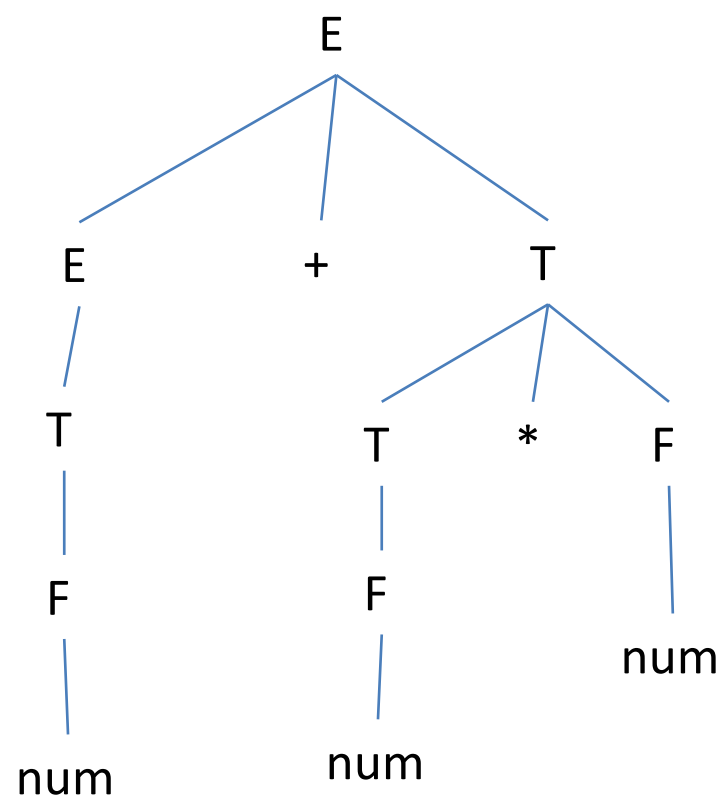
For input, 2 + 3 * 4

Output: 2 3 4 * +

Example 4 (bottom-up approach)

- SDT to convert infix to postfix

E→ E + T	{printf("+");}
E → T	{}
T → T * F	{printf("*");}
T→F	{}
F→ num	{printf(num.lval);}



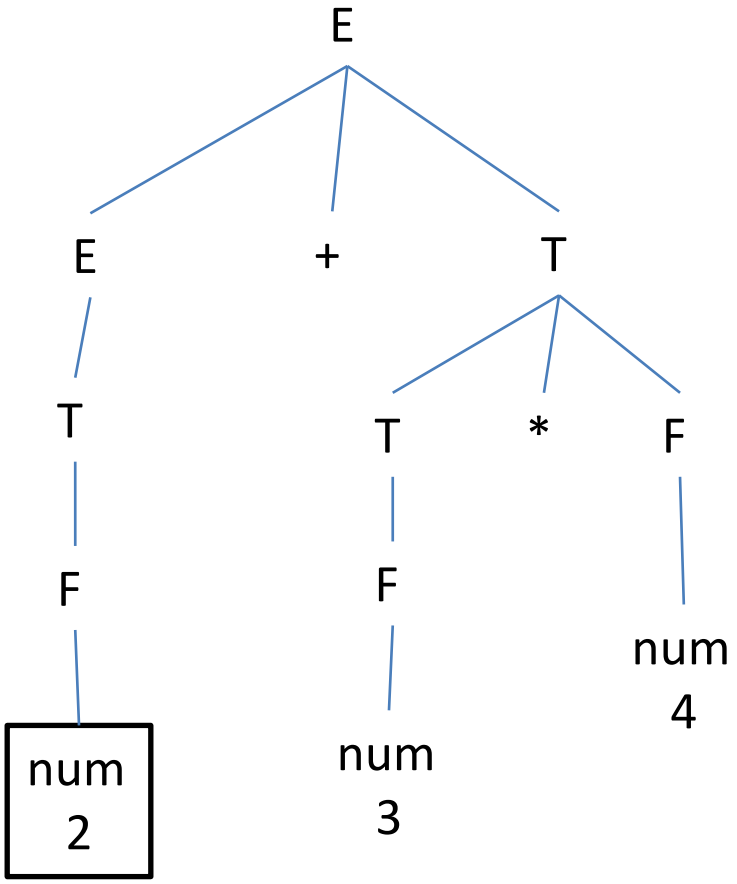
For input, 2 + 3 * 4

Example 4 (bottom-up approach)

- SDT to convert infix to postfix

E→ E + T	{printf("+");}
E → T	{}
T → T * F	{printf("*");}
T→F	{}
F→ num	{printf(num.lval);}

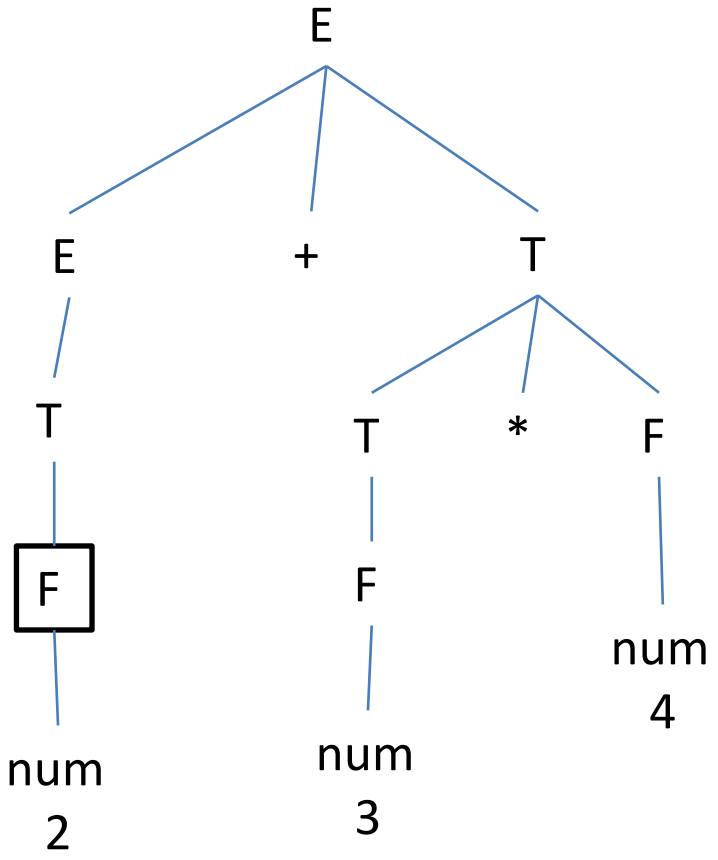
For input, 2 + 3 * 4



Example 4 (bottom-up approach)

- SDT to convert infix to postfix

E→ E + T	{printf("+");}
E → T	{}
T → T * F	{printf("*");}
T→F	{}
F→ num	{printf(num.lval);}



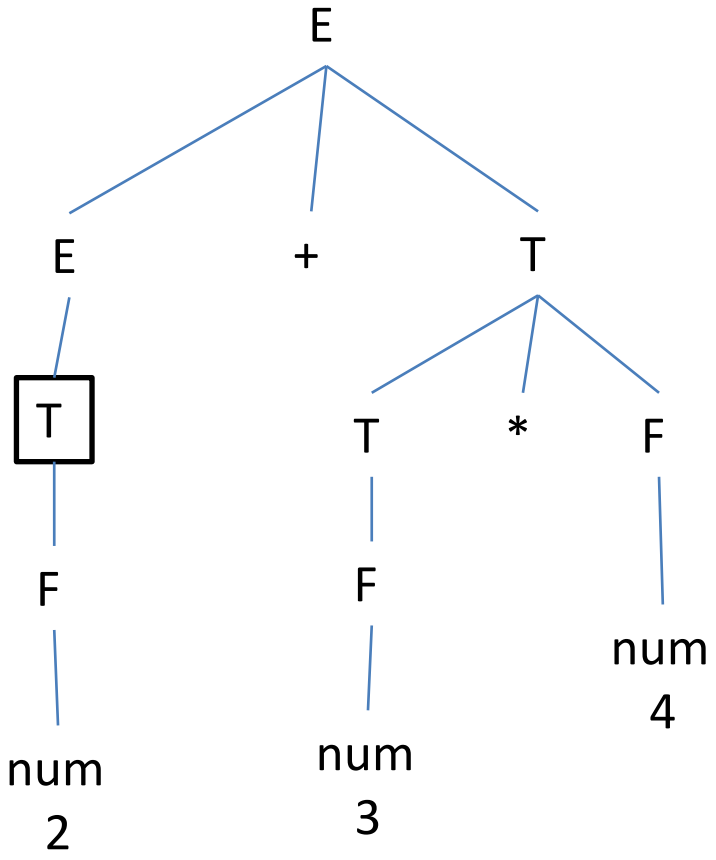
For input, 2 + 3 * 4

Output: 2

Example 4 (bottom-up approach)

- SDT to convert infix to postfix

E→ E + T	{printf("+");}
E → T	{}
T → T * F	{printf("*");}
T→F	{}
F→ num	{printf(num.lval);}



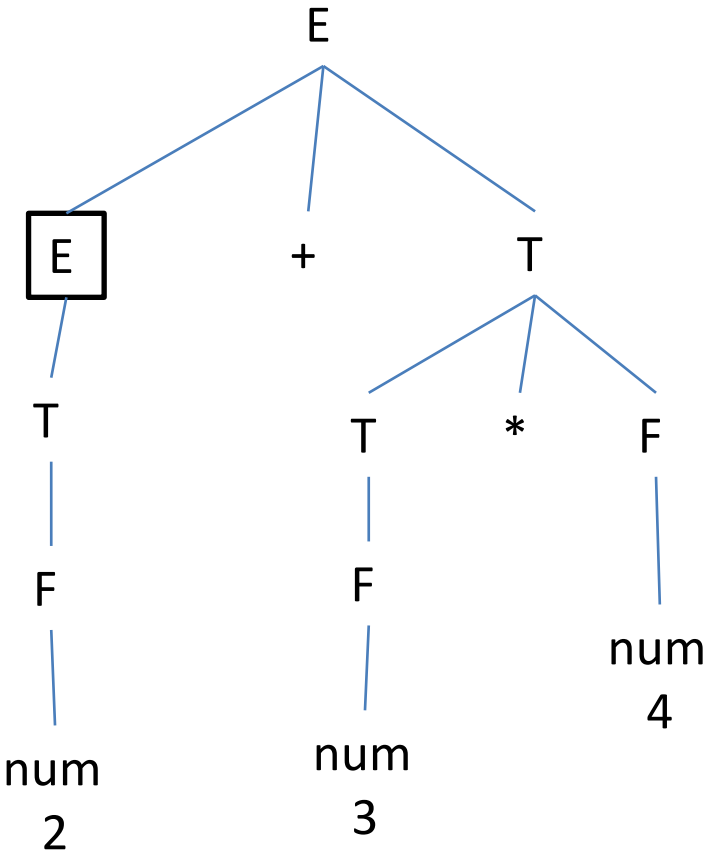
For input, 2 + 3 * 4

Output: 2

Example 4 (bottom-up approach)

- SDT to convert infix to postfix

E→ E + T	{printf("+");}
E → T	{}
T → T * F	{printf("*");}
T→F	{}
F→ num	{printf(num.lval);}



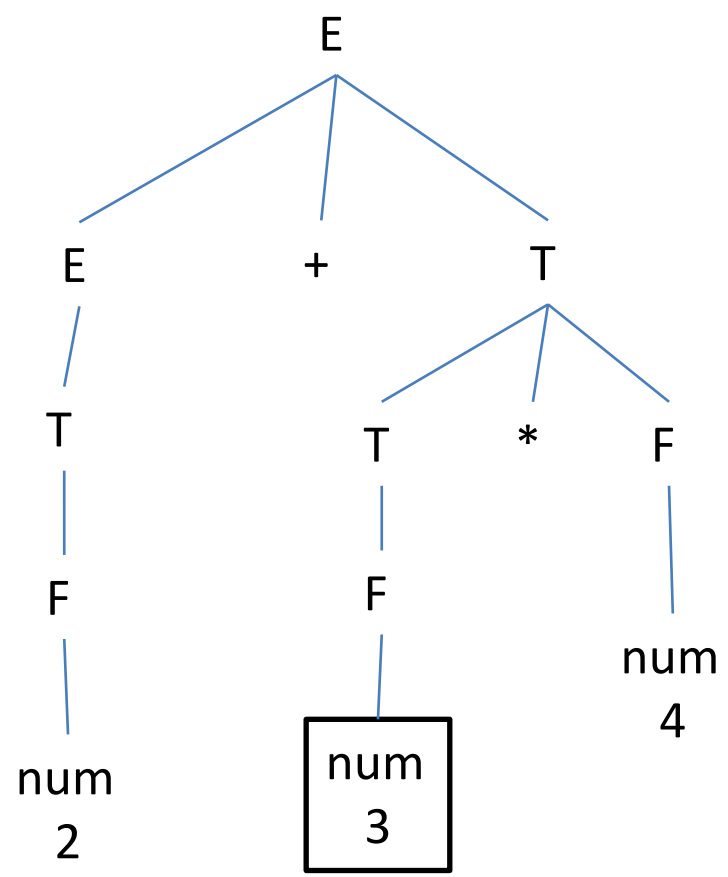
For input, 2 + 3 * 4

Output: 2

Example 4 (bottom-up approach)

- SDT to convert infix to postfix

E→ E + T	{printf("+");}
E → T	{}
T → T * F	{printf("*");}
T→F	{}
F→ num	{printf(num.lval);}



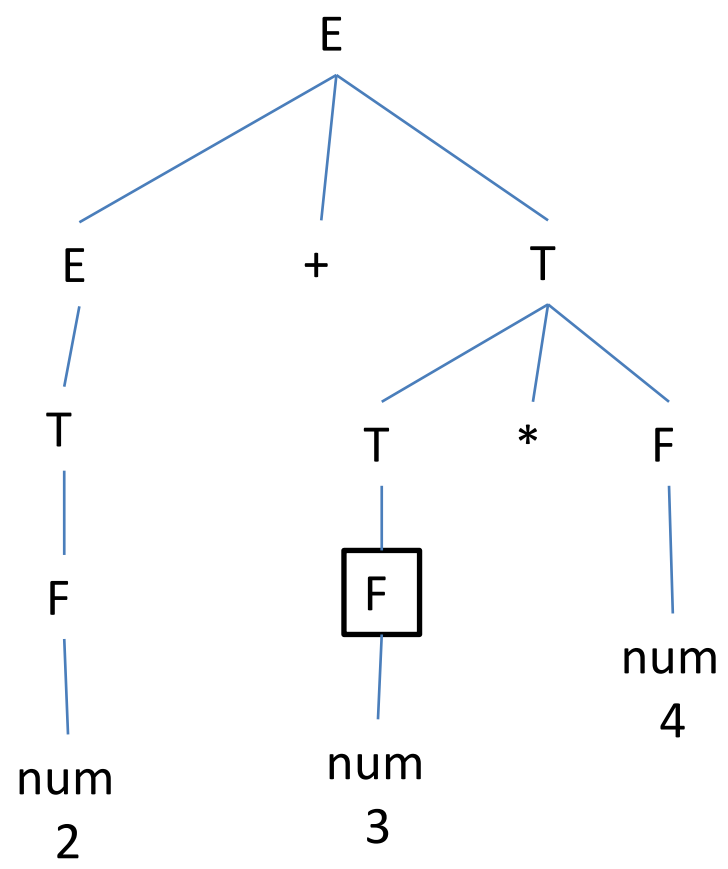
For input, 2 + 3 * 4

Output: 2

Example 4 (bottom-up approach)

- SDT to convert infix to postfix

E→ E + T	{printf("+");}
E → T	{}
T → T * F	{printf("*");}
T→F	{}
F→ num	{printf(num.lval);}



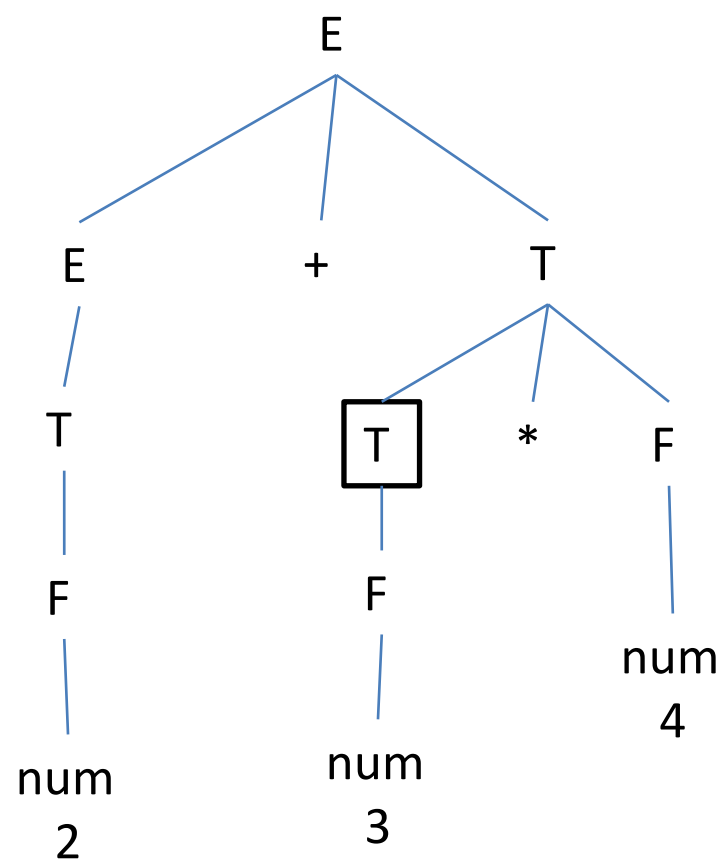
For input, 2 + 3 * 4

Output: 2 3

Example 4 (bottom-up approach)

- SDT to convert infix to postfix

E→ E + T	{printf("+");}
E → T	{}
T → T * F	{printf("*");}
T→F	{}
F→ num	{printf(num.lval);}



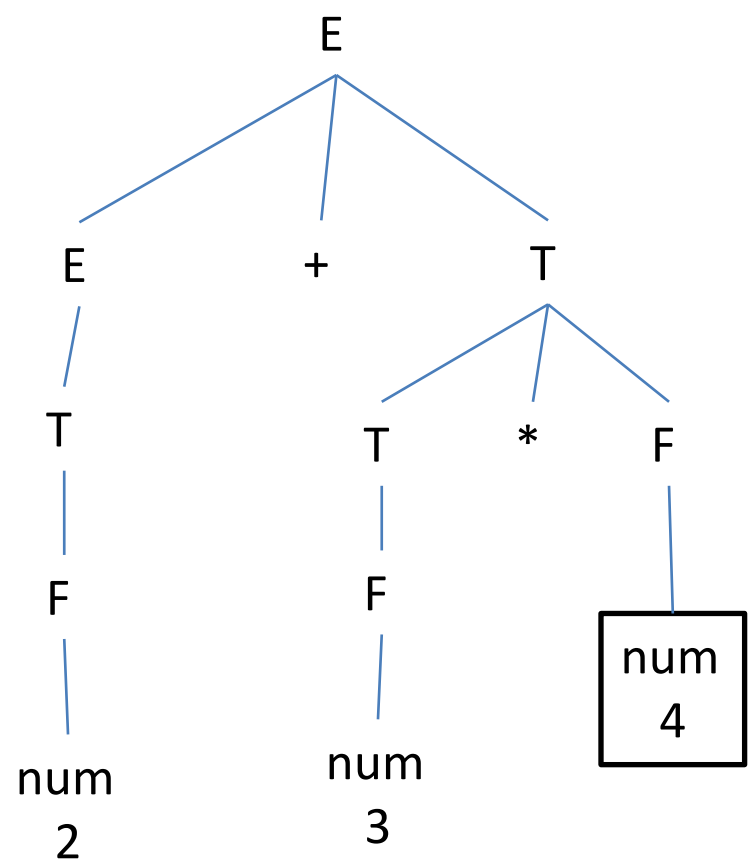
For input, 2 + 3 * 4

Output: 2 3

Example 4 (bottom-up approach)

- SDT to convert infix to postfix

E→ E + T	{printf("+");}
E → T	{}
T → T * F	{printf("*");}
T→F	{}
F→ num	{printf(num.lval);}



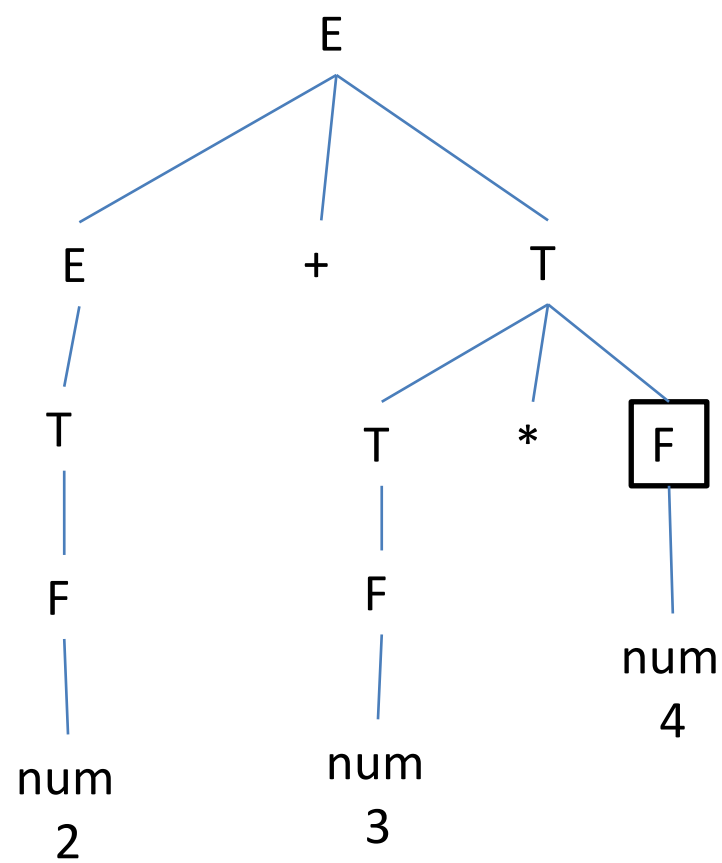
For input, 2 + 3 * 4

Output: 2 3

Example 4 (bottom-up approach)

- SDT to convert infix to postfix

E→ E + T	{printf("+");}
E → T	{}
T → T * F	{printf("*");}
T→F	{}
F→ num	{printf(num.lval);}



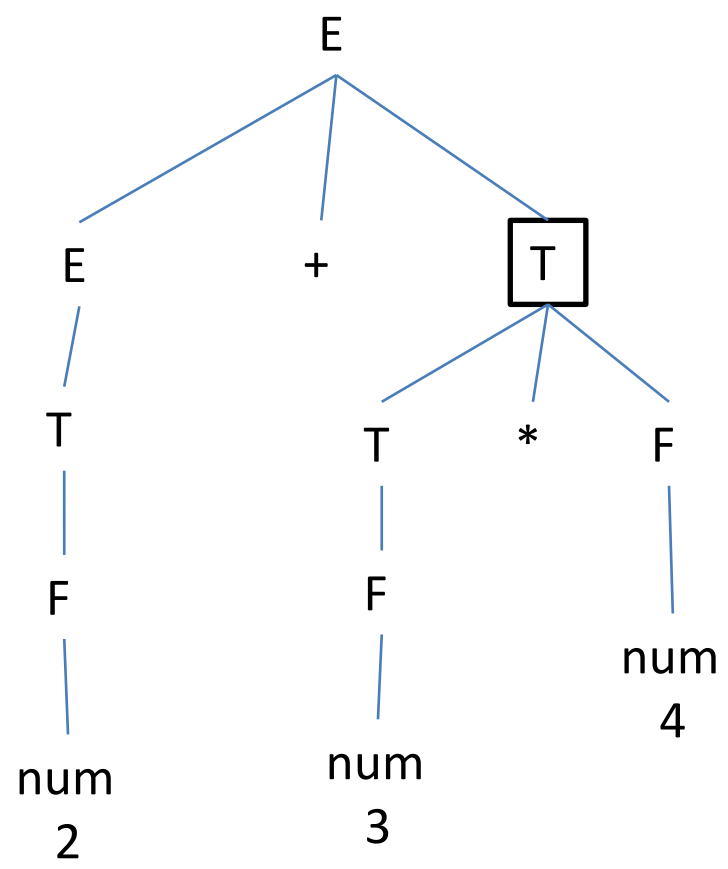
For input, 2 + 3 * 4

Output: 2 3 4

Example 4 (bottom-up approach)

- SDT to convert infix to postfix

E→ E + T	{printf("+");}
E → T	{}
T → T * F	{printf("*");}
T→F	{}
F→ num	{printf(num.lval);}



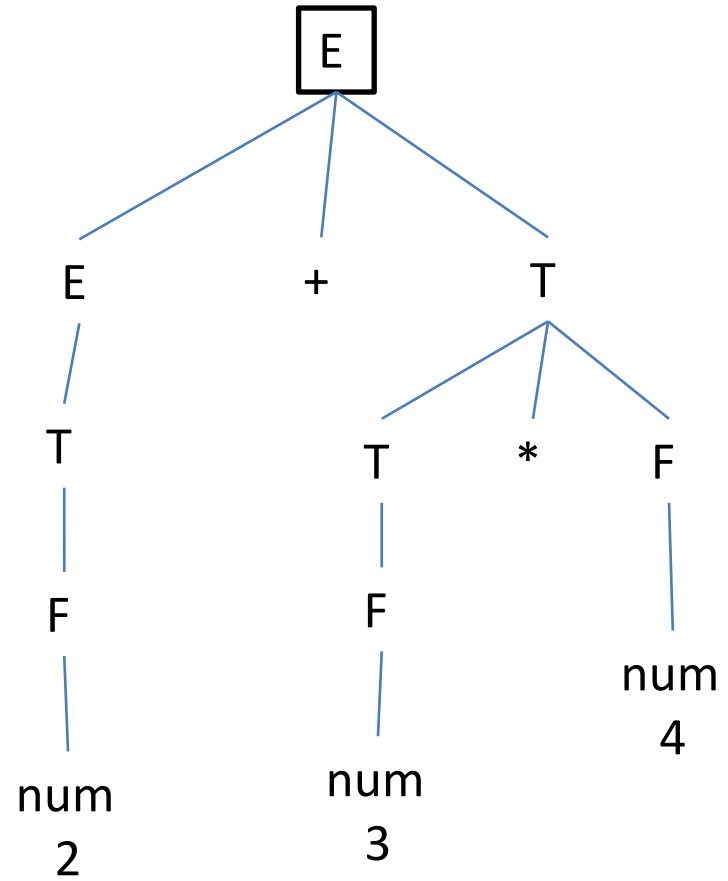
For input, 2 + 3 * 4

Output: 2 3 4 *

Example 4 (bottom-up approach)

- SDT to convert infix to postfix

E→ E + T	{printf("+");}
E → T	{}
T → T * F	{printf("*");}
T→F	{}
F→ num	{printf(num.lval);}



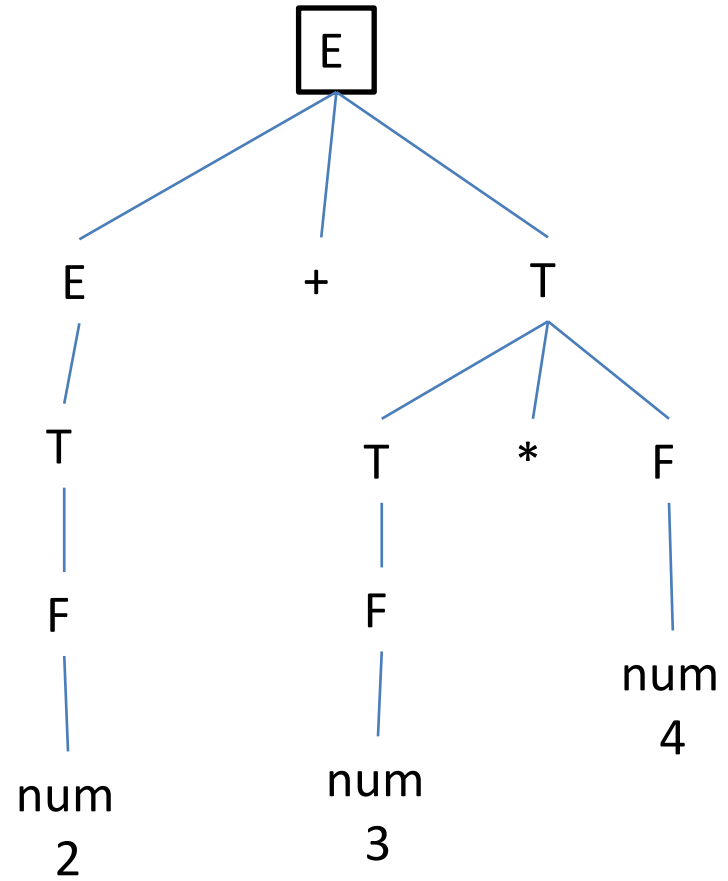
For input, 2 + 3 * 4

Output: 2 3 4 * +

Example 4 (bottom-up approach)

- SDT to convert infix to postfix

E→ E + T	{printf("+");}
E → T	{}
T → T * F	{printf("*");}
T→F	{}
F→ num	{printf(num.lval);}



For input, 2 + 3 * 4

Output: 2 3 4 * +

Example 5

- SDT to build a syntax tree

E→ E + T {E.nptr=mknode(E.nptr, '+', T.nptr);}

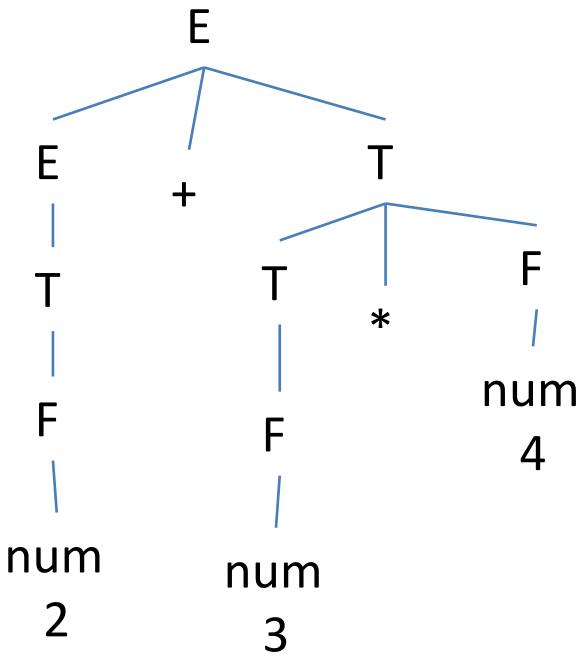
E → T {E.nptr=T.nptr;}

T → T * F {T.nptr=mknode(T.nptr, *, F.nptr);}

T→F {T.nptr=F.nptr}

F→ num {F.nptr=mknode(null, idname, null);}

For input, 2 + 3 * 4



Example 5

- SDT to build a syntax tree

E→ E + T {E.nptr=mknode(E.nptr, '+', T.nptr);}

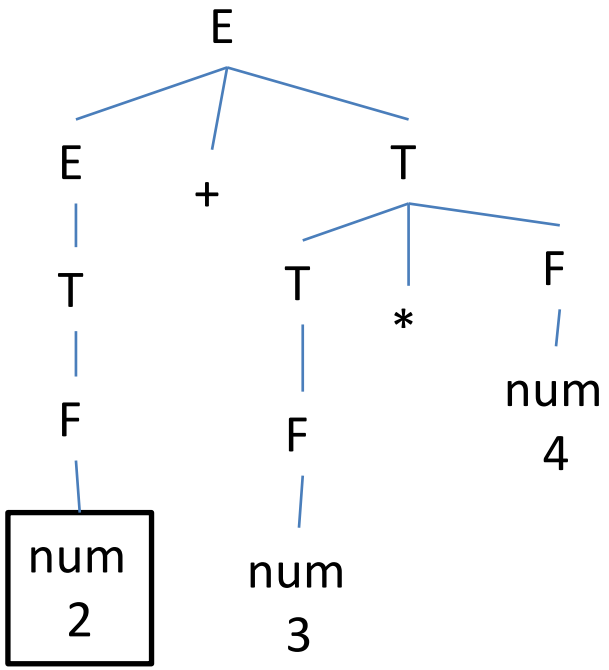
E → T {E.nptr=T.nptr;}

T → T * F {T.nptr=mknode(T.nptr, *, F.nptr);}

T→F {T.nptr=F.nptr}

F→ num {F.nptr=mknode(null, idname, null);}

For input, 2 + 3 * 4



Example 5

- SDT to build a syntax tree

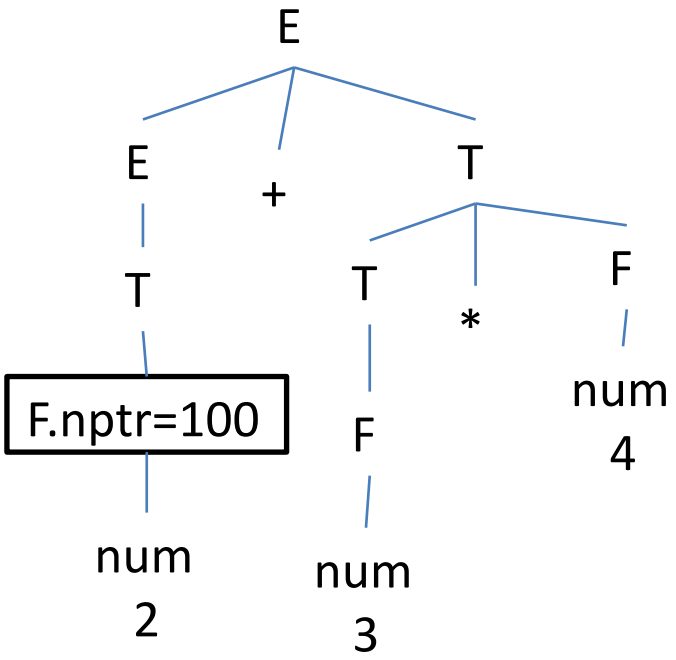
E→ E + T {E.nptr=mknode(E.nptr, '+', T.nptr);}

E → T {E.nptr=T.nptr;}

T → T * F {T.nptr=mknode(T.nptr, *, F.nptr);}

T→F {T.nptr=F.nptr}

F→ num {F.nptr=mknode(null, idname, null);}



For input, 2 + 3 * 4

100

null	2	null

Example 5

- SDT to build a syntax tree

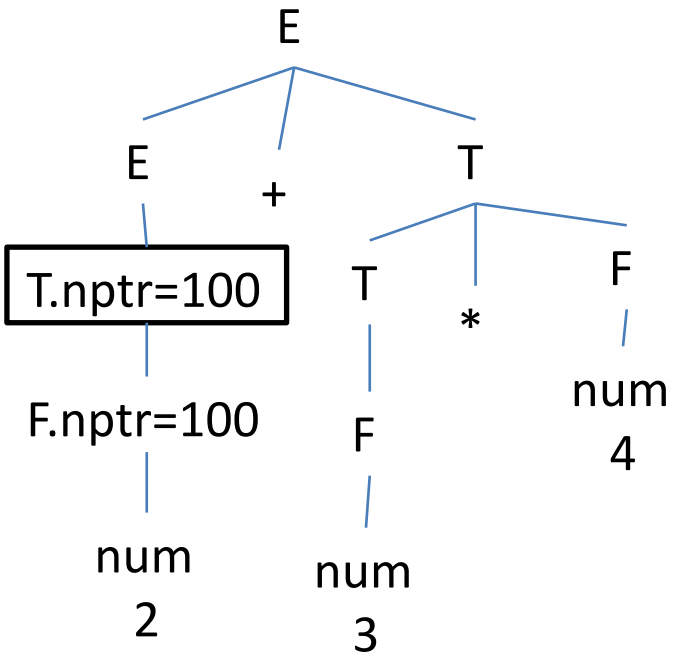
E→ E + T {E.nptr=mknode(E.nptr, '+', T.nptr);}

E → T {E.nptr=T.nptr;}

T → T * F {T.nptr=mknode(T.nptr, *, F.nptr);}

T→F {T.nptr=F.nptr}

F→ num {F.nptr=mknode(null, idname, null);}



For input, 2 + 3 * 4

100

null	2	null

Example 5

- SDT to build a syntax tree

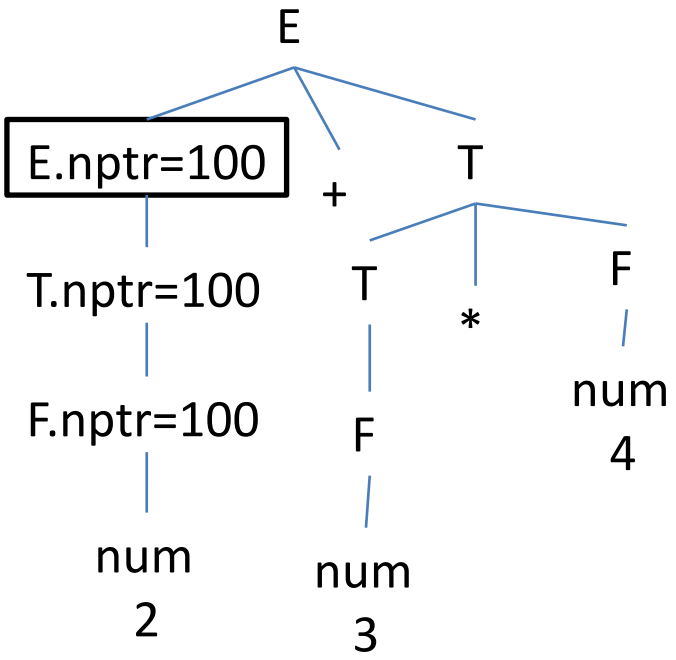
E→ E + T {E.nptr=mknode(E.nptr, '+', T.nptr);}

E → T {E.nptr=T.nptr;}

T → T * F {T.nptr=mknode(T.nptr, *, F.nptr);}

T→F {T.nptr=F.nptr}

F→ num {F.nptr=mknode(null, idname, null);}



For input, 2 + 3 * 4

100

null	2	null

Example 5

- SDT to build a syntax tree

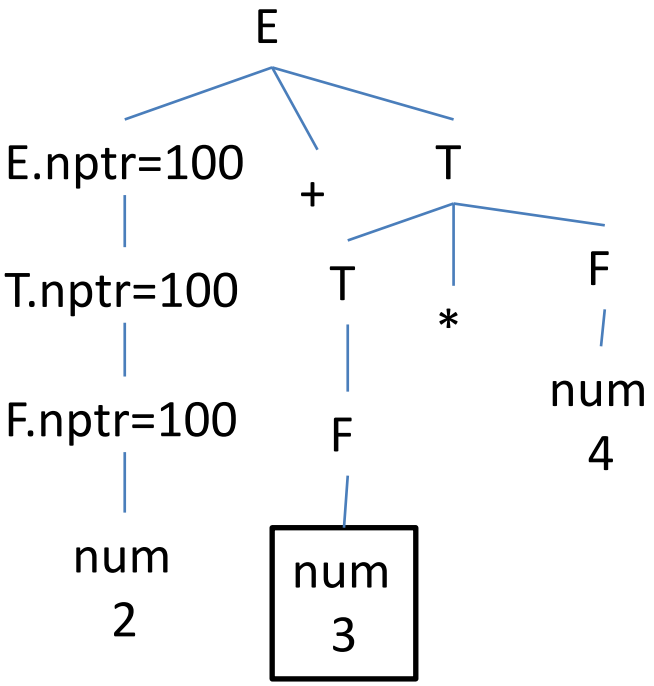
E→ E + T {E.nptr=mknode(E.nptr, '+', T.nptr);}

E → T {E.nptr=T.nptr;}

T → T * F {T.nptr=mknode(T.nptr, *, F.nptr);}

T→F {T.nptr=F.nptr}

F→ num {F.nptr=mknode(null, idname, null);}



For input, 2 + 3 * 4

100

null	2	null

Example 5

- SDT to build a syntax tree

E→ E + T {E.nptr=mknode(E.nptr, '+', T.nptr);}

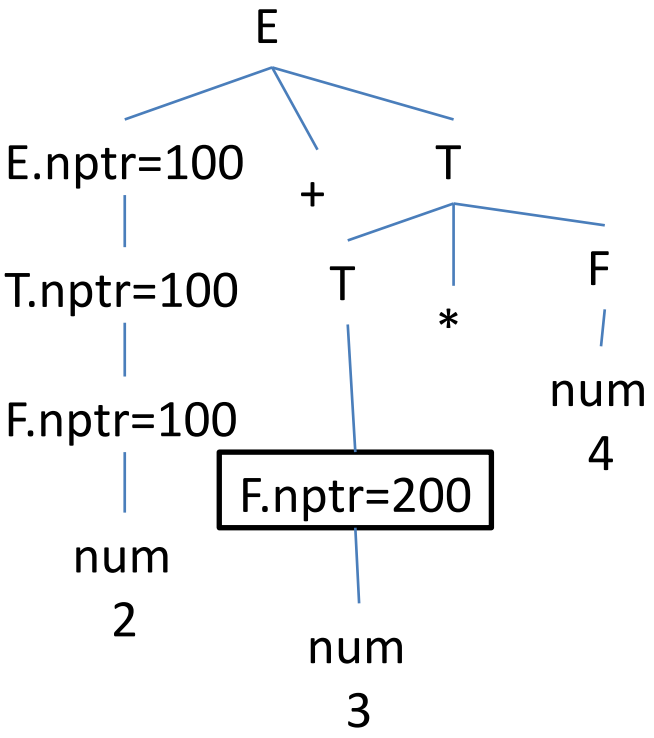
E → T {E.nptr=T.nptr;}

T → T * F {T.nptr=mknode(T.nptr, *, F.nptr);}

T→F {T.nptr=F.nptr}

F→ num {F.nptr=mknode(null, idname, null);}

For input, 2 + 3 * 4



100

null	2	null

200

null	3	null

Example 5

- SDT to build a syntax tree

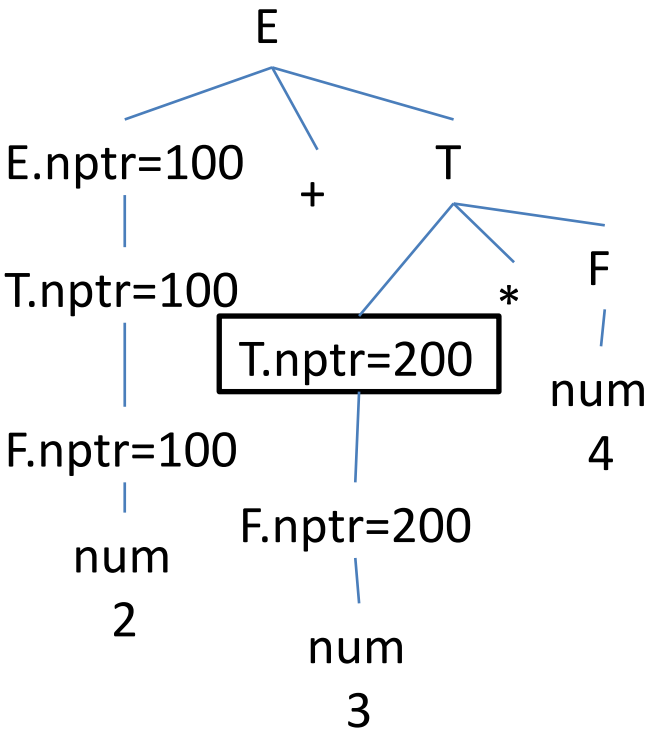
E→ E + T {E.nptr=mknode(E.nptr, '+', T.nptr);}

E → T {E.nptr=T.nptr;}

T → T * F {T.nptr=mknode(T.nptr, *, F.nptr);}

T→F {T.nptr=F.nptr}

F→ num {F.nptr=mknode(null, idname, null);}



For input, 2 + 3 * 4

100

null	2	null

200

null	3	null

Example 5

- SDT to build a syntax tree

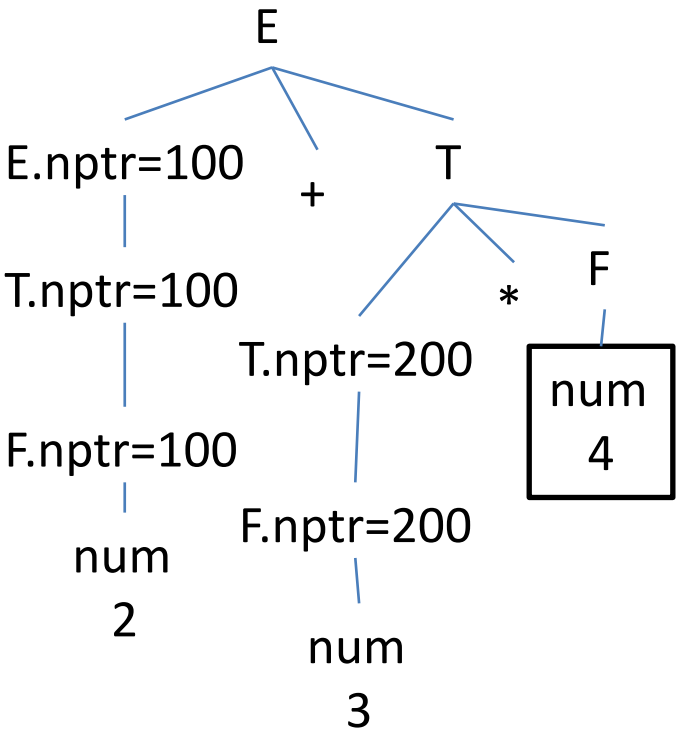
E→ E + T {E.nptr=mknode(E.nptr, '+', T.nptr);}

E → T {E.nptr=T.nptr;}

T → T * F {T.nptr=mknode(T.nptr, *, F.nptr);}

T→F {T.nptr=F.nptr}

F→ num {F.nptr=mknode(null, idname, null);}



For input, 2 + 3 * 4

100

null	2	null

200

null	3	null

Example 5

- SDT to build a syntax tree

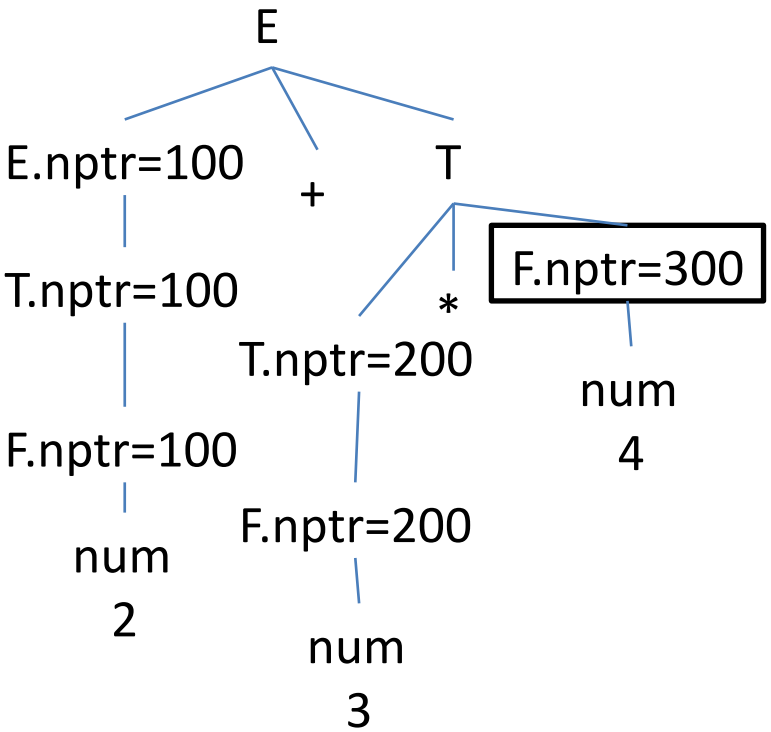
E→ E + T {E.nptr=mknode(E.nptr, '+', T.nptr);}

E → T {E.nptr=T.nptr;}

T → T * F {T.nptr=mknode(T.nptr, *, F.nptr);}

T→F {T.nptr=F.nptr}

F→ num {F.nptr=mknode(null, idname, null);}



For input, 2 + 3 * 4

100

null	2	null

200

null	3	null

300

null	4	null

Example 5

- SDT to build a syntax tree

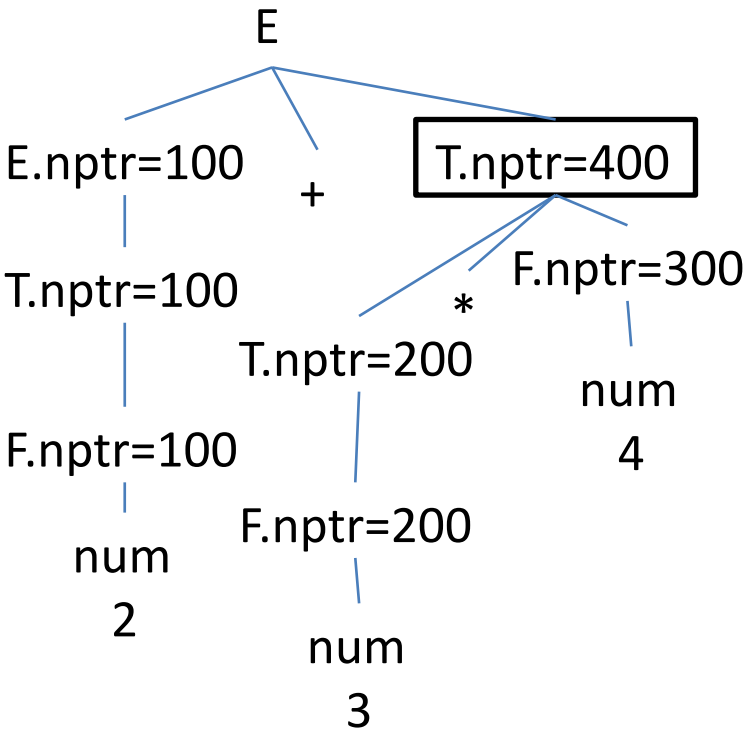
E→ E + T {E.nptr=mknode(E.nptr, '+', T.nptr);}

E → T {E.nptr=T.nptr;}

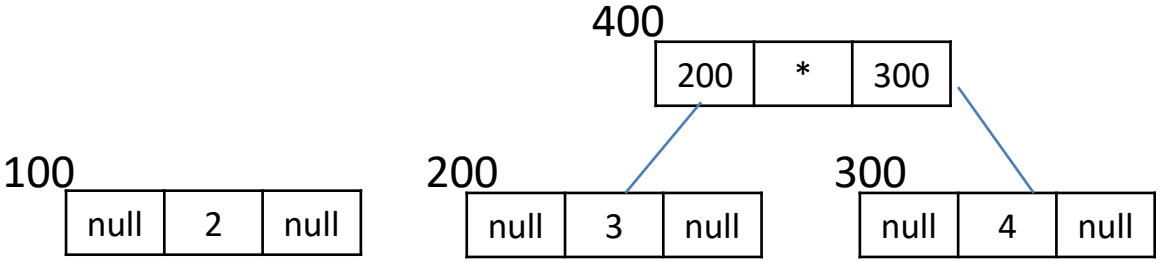
T → T * F {T.nptr=mknode(T.nptr, *, F.nptr);}

T→F {T.nptr=F.nptr}

F→ num {F.nptr=mknode(null, idname, null);}



For input, 2 + 3 * 4



Example 5

- SDT to build a syntax tree

E→ E + T {E.nptr=mknode(E.nptr, '+', T.nptr);}

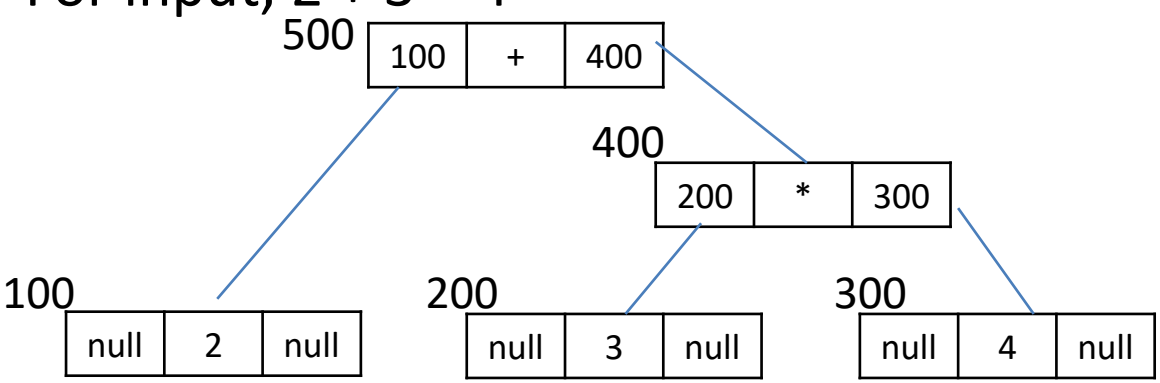
E → T {E.nptr=T.nptr;}

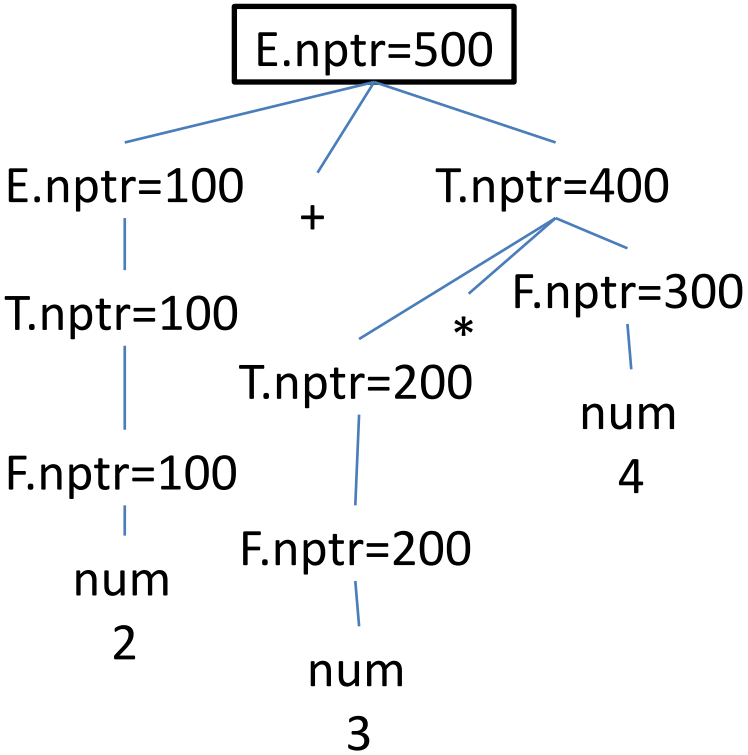
T → T * F {T.nptr=mknode(T.nptr, *, F.nptr);}

T→F {T.nptr=F.nptr}

F→ num {F.nptr=mknode(null, idname, null);}

For input, 2 + 3 * 4





Example 6

- SDT to generate three address code

$S \rightarrow id = E$	{gen(id.name = E.place}
$E \rightarrow E_1 + T$	{E.place = newTemp(); gen(E.place = E_1.place + T.place);}
$E \rightarrow T$	{E.place = T.place}
$T \rightarrow T_1 * F$	{T.place = newTemp(); gen(E.place = T_1.place * F.place);}
$T \rightarrow F$	{T.place = F.place}
$F \rightarrow id$	{F.place = id.name}

gen() :- generates a statement in three address code

Example 6

- SDT to generate three address code

$S \rightarrow id = E$	{gen(id.name = E.place}
$E \rightarrow E_1 + T$	{E.place = newTemp(); gen(E.place = E_1.place + T.place);}
$E \rightarrow T$	{E.place = T.place}
$T \rightarrow T_1 * F$	{T.place = newTemp(); gen(E.place = T_1.place * F.place);}
$T \rightarrow F$	{T.place = F.place}
$F \rightarrow id$	{F.place = id.name}

```
x = a + b * c
```

⟹

```
t1 = b * c
t2 = a + t1
x = t2
```

Example 6

- SDT to generate three address code

$S \rightarrow id = E$ {gen(id.name = E.place}

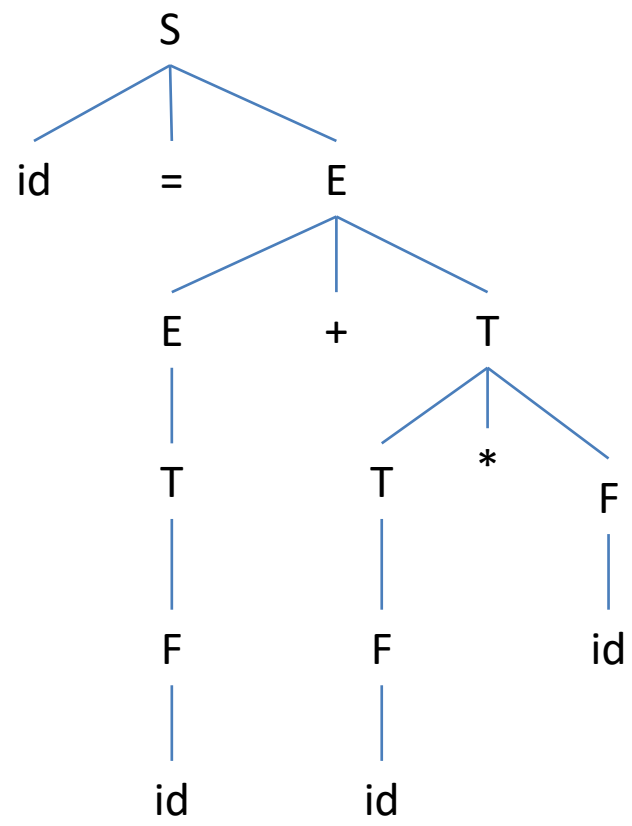
$E \rightarrow E_1 + T$ {E.place = newTemp();
gen(E.place = E_1.place + T.place);}

$E \rightarrow T$ {E.place = T.place}

$T \rightarrow T_1 * F$ {T.place = newTemp();
gen(E.place = T_1.place * F.place);}

$T \rightarrow F$ {T.place = F.place}

$F \rightarrow id$ {F.place = id.name}



x = a + b * c

Example 6

- SDT to generate three address code

$S \rightarrow id = E$ {gen(id.name = E.place}

$E \rightarrow E_1 + T$ {E.place = newTemp();
gen(E.place = E_1.place + T.place);}

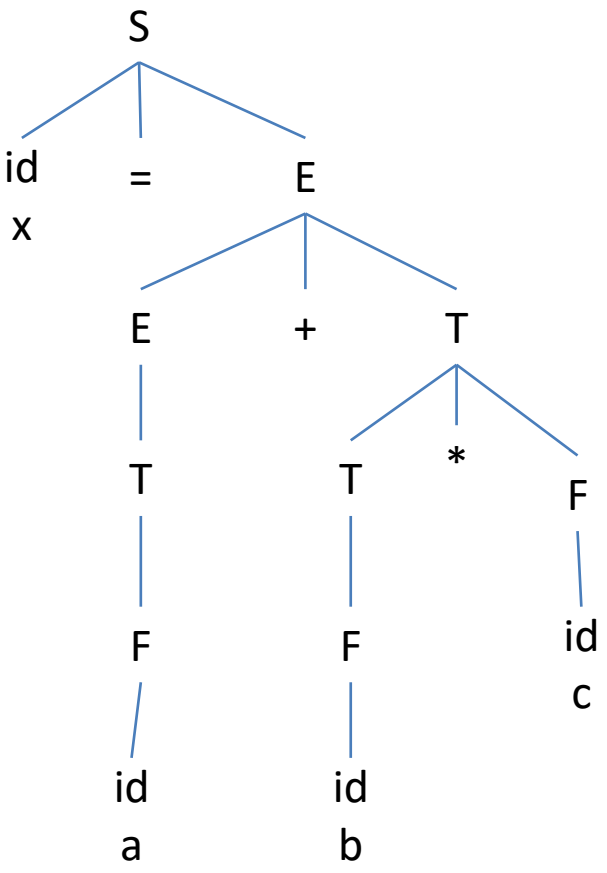
$E \rightarrow T$ {E.place = T.place}

$T \rightarrow T_1 * F$ {T.place = newTemp();
gen(E.place = T_1.place * F.place);}

$T \rightarrow F$ {T.place = F.place}

$F \rightarrow id$ {F.place = id.name}

x = a + b * c



Example 6

- SDT to generate three address code

$S \rightarrow id = E$ {gen(id.name = E.place}

$E \rightarrow E_1 + T$ {E.place = newTemp();
gen(E.place = E_1.place + T.place);}

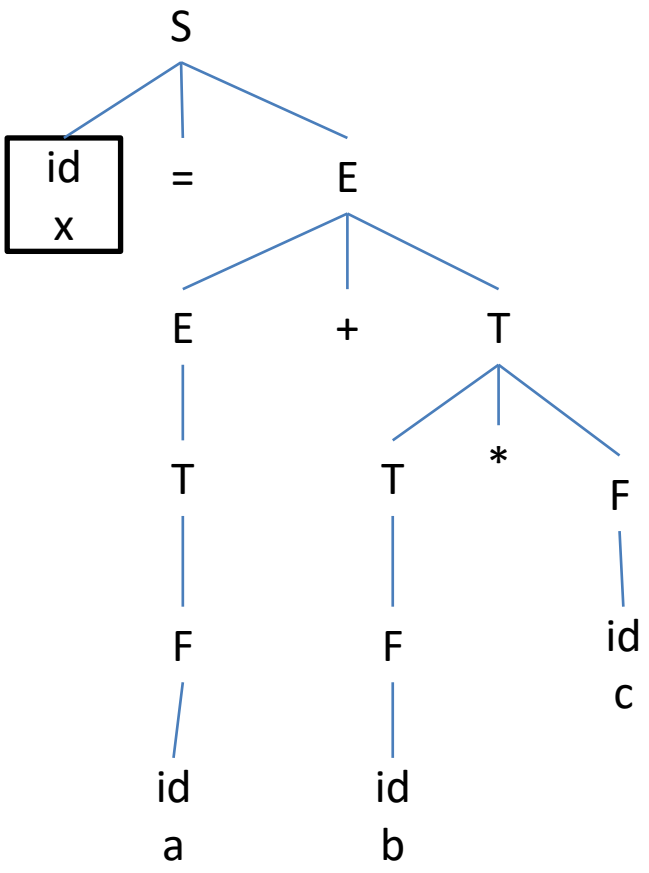
$E \rightarrow T$ {E.place = T.place}

$T \rightarrow T_1 * F$ {T.place = newTemp();
gen(E.place = T_1.place * F.place);}

$T \rightarrow F$ {T.place = F.place}

$F \rightarrow id$ {F.place = id.name}

x = a + b * c



Example 6

- SDT to generate three address code

$S \rightarrow id = E$ {gen(id.name = E.place}

$E \rightarrow E_1 + T$ {E.place = newTemp();
gen(E.place = E_1.place + T.place);}

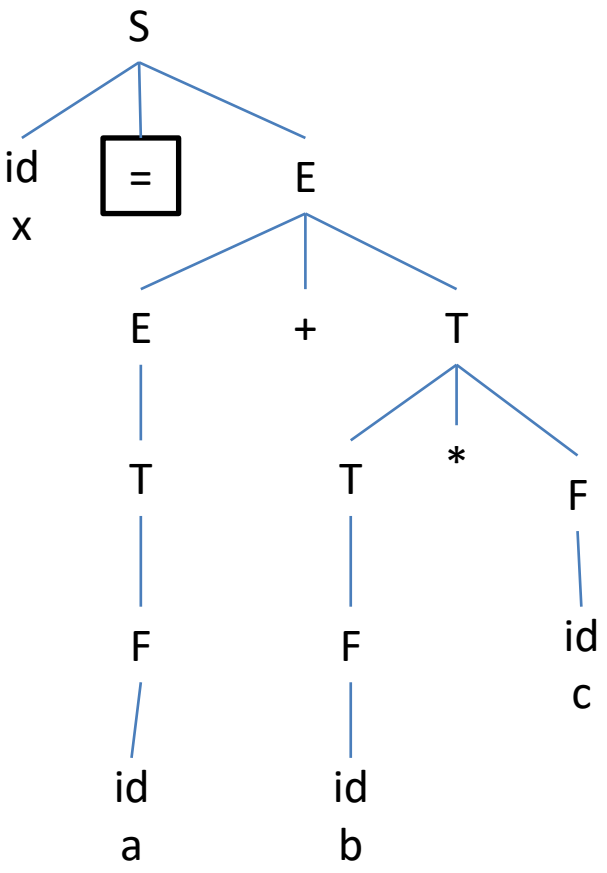
$E \rightarrow T$ {E.place = T.place}

$T \rightarrow T_1 * F$ {T.place = newTemp();
gen(E.place = T_1.place * F.place);}

$T \rightarrow F$ {T.place = F.place}

$F \rightarrow id$ {F.place = id.name}

x = a + b * c



Example 6

- SDT to generate three address code

$S \rightarrow id = E$ {gen(id.name = E.place}

$E \rightarrow E_1 + T$ {E.place = newTemp();
gen(E.place = E_1.place + T.place);}

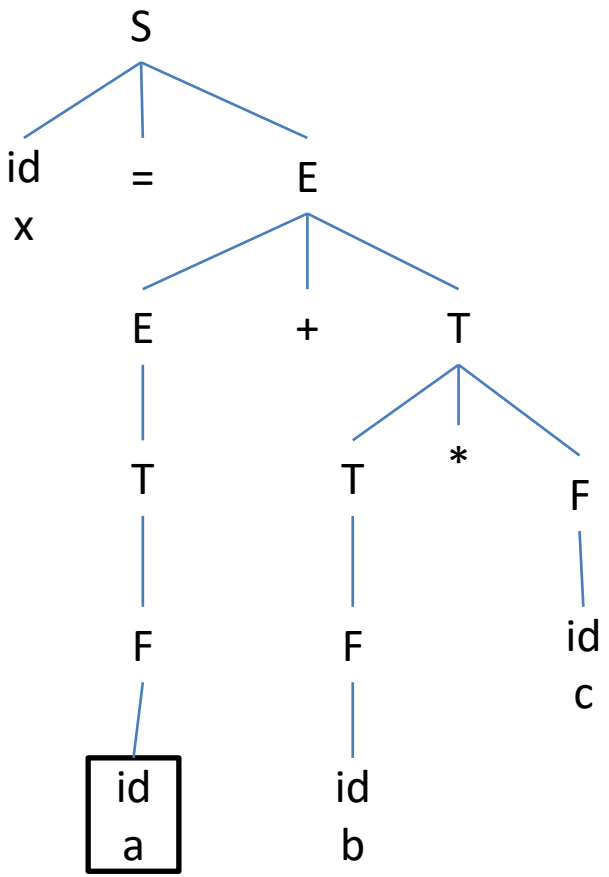
$E \rightarrow T$ {E.place = T.place}

$T \rightarrow T_1 * F$ {T.place = newTemp();
gen(E.place = T_1.place * F.place);}

$T \rightarrow F$ {T.place = F.place}

$F \rightarrow id$ {F.place = id.name}

x = a + b * c



Example 6

- SDT to generate three address code

$S \rightarrow id = E$ {gen(id.name = E.place}

$E \rightarrow E_1 + T$ {E.place = newTemp();
gen(E.place = E_1.place + T.place);}

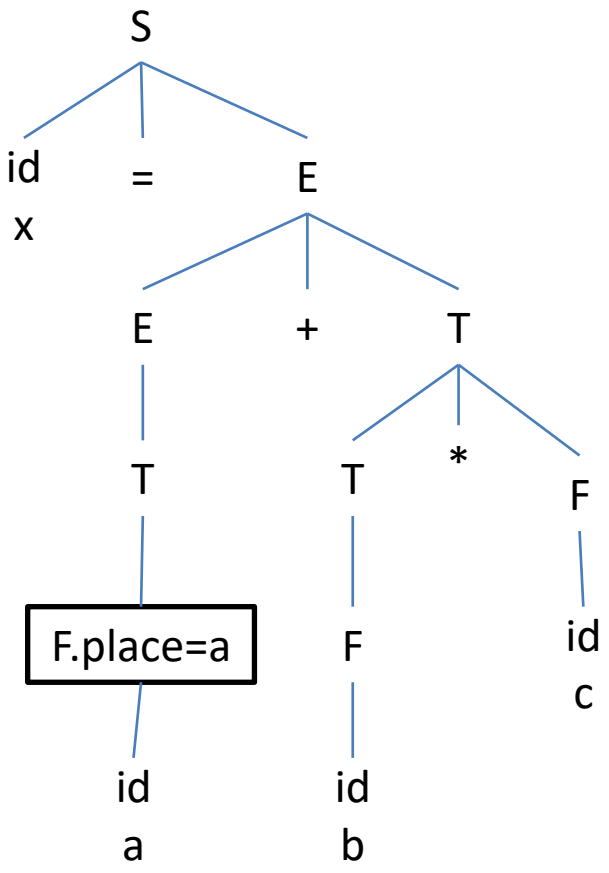
$E \rightarrow T$ {E.place = T.place}

$T \rightarrow T_1 * F$ {T.place = newTemp();
gen(E.place = T_1.place * F.place);}

$T \rightarrow F$ {T.place = F.place}

$F \rightarrow id$ {F.place = id.name}

x = a + b * c



Example 6

- SDT to generate three address code

$S \rightarrow id = E$ {gen(id.name = E.place}

$E \rightarrow E_1 + T$ {E.place = newTemp();
gen(E.place = E_1.place + T.place);}

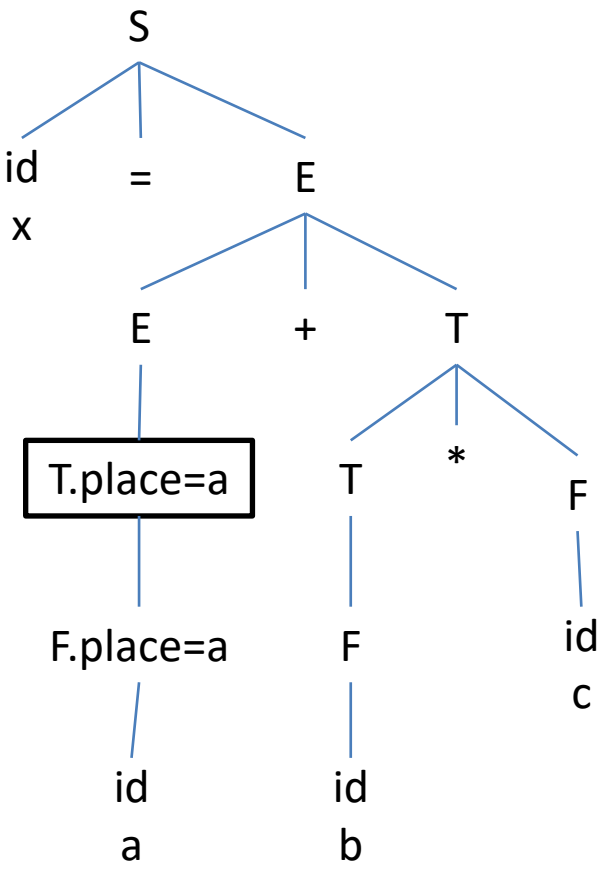
$E \rightarrow T$ {E.place = T.place}

$T \rightarrow T_1 * F$ {T.place = newTemp();
gen(E.place = T_1.place * F.place);}

$T \rightarrow F$ {T.place = F.place}

$F \rightarrow id$ {F.place = id.name}

x = a + b * c



Example 6

- SDT to generate three address code

$S \rightarrow id = E$ {gen(id.name = E.place}

$E \rightarrow E_1 + T$ {E.place = newTemp();
gen(E.place = E_1.place + T.place);}

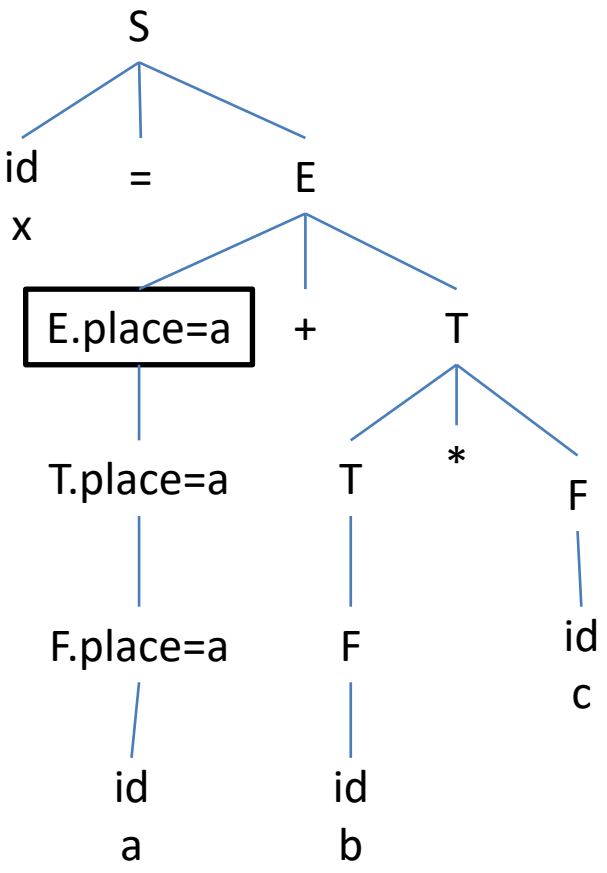
$E \rightarrow T$ {E.place = T.place}

$T \rightarrow T_1 * F$ {T.place = newTemp();
gen(E.place = T_1.place * F.place);}

$T \rightarrow F$ {T.place = F.place}

$F \rightarrow id$ {F.place = id.name}

x = a + b * c



Example 6

- SDT to generate three address code

$S \rightarrow id = E$ {gen(id.name = E.place}

$E \rightarrow E_1 + T$ {E.place = newTemp();
gen(E.place = E_1.place + T.place);}

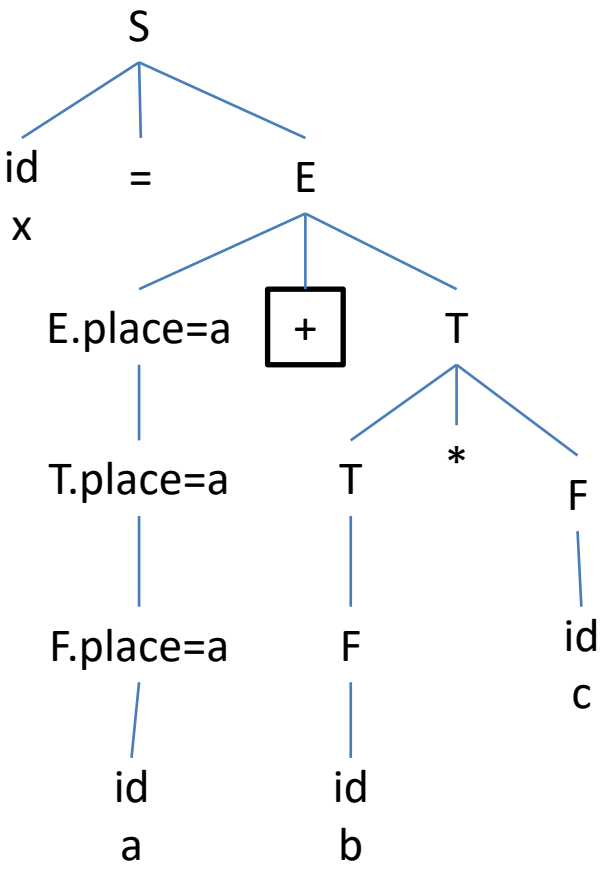
$E \rightarrow T$ {E.place = T.place}

$T \rightarrow T_1 * F$ {T.place = newTemp();
gen(E.place = T_1.place * F.place);}

$T \rightarrow F$ {T.place = F.place}

$F \rightarrow id$ {F.place = id.name}

x = a + b * c



Example 6

- SDT to generate three address code

$S \rightarrow id = E$ {gen(id.name = E.place}

$E \rightarrow E_1 + T$ {E.place = newTemp();
gen(E.place = E_1.place + T.place);}

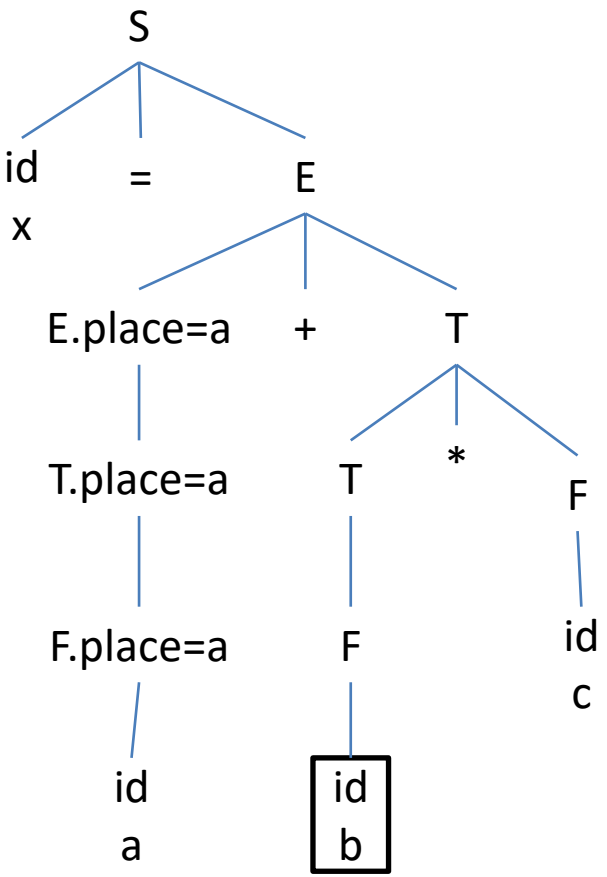
$E \rightarrow T$ {E.place = T.place}

$T \rightarrow T_1 * F$ {T.place = newTemp();
gen(E.place = T_1.place * F.place);}

$T \rightarrow F$ {T.place = F.place}

$F \rightarrow id$ {F.place = id.name}

x = a + b * c



Example 6

- SDT to generate three address code

$S \rightarrow id = E$ {gen(id.name = E.place}

$E \rightarrow E_1 + T$ {E.place = newTemp();
gen(E.place = E_1.place + T.place);}

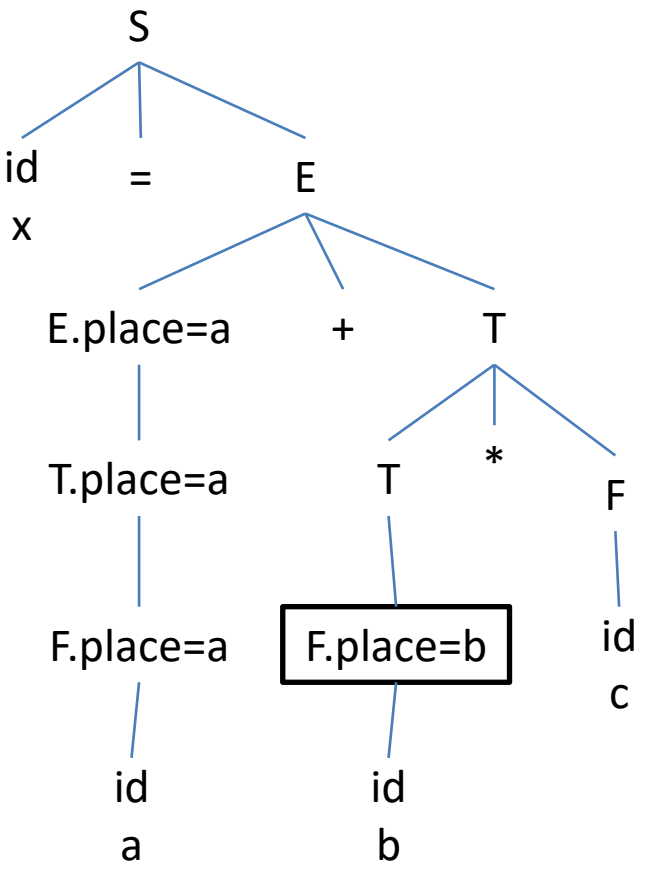
$E \rightarrow T$ {E.place = T.place}

$T \rightarrow T_1 * F$ {T.place = newTemp();
gen(E.place = T_1.place * F.place);}

$T \rightarrow F$ {T.place = F.place}

$F \rightarrow id$ {F.place = id.name}

x = a + b * c



Example 6

- SDT to generate three address code

$S \rightarrow id = E$ {gen(id.name = E.place}

$E \rightarrow E_1 + T$ {E.place = newTemp();
gen(E.place = E_1.place + T.place);}

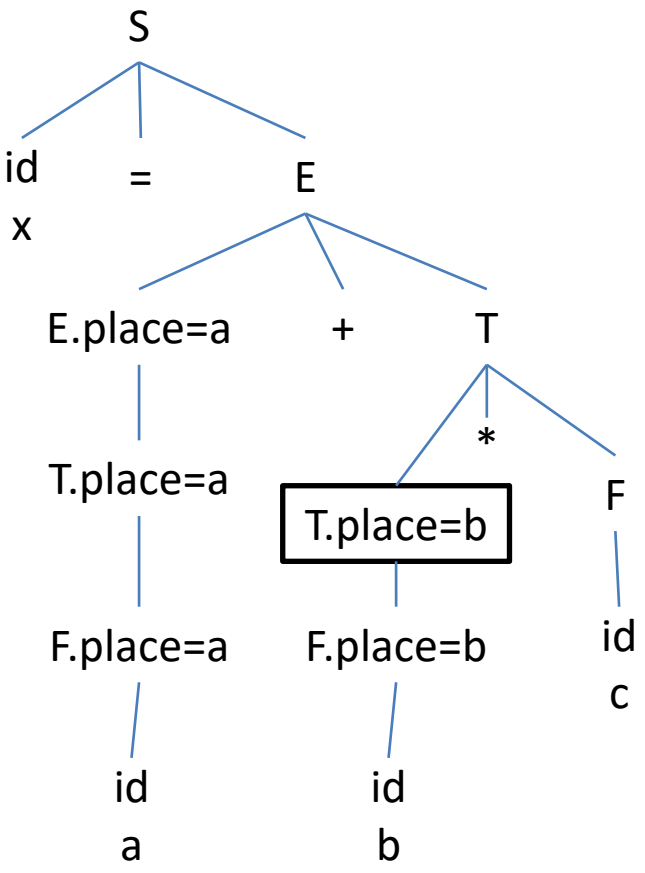
$E \rightarrow T$ {E.place = T.place}

$T \rightarrow T_1 * F$ {T.place = newTemp();
gen(E.place = T_1.place * F.place);}

$T \rightarrow F$ {T.place = F.place}

$F \rightarrow id$ {F.place = id.name}

x = a + b * c



Example 6

- SDT to generate three address code

$S \rightarrow id = E$ {gen(id.name = E.place}

$E \rightarrow E_1 + T$ {E.place = newTemp();
gen(E.place = E_1.place + T.place);}

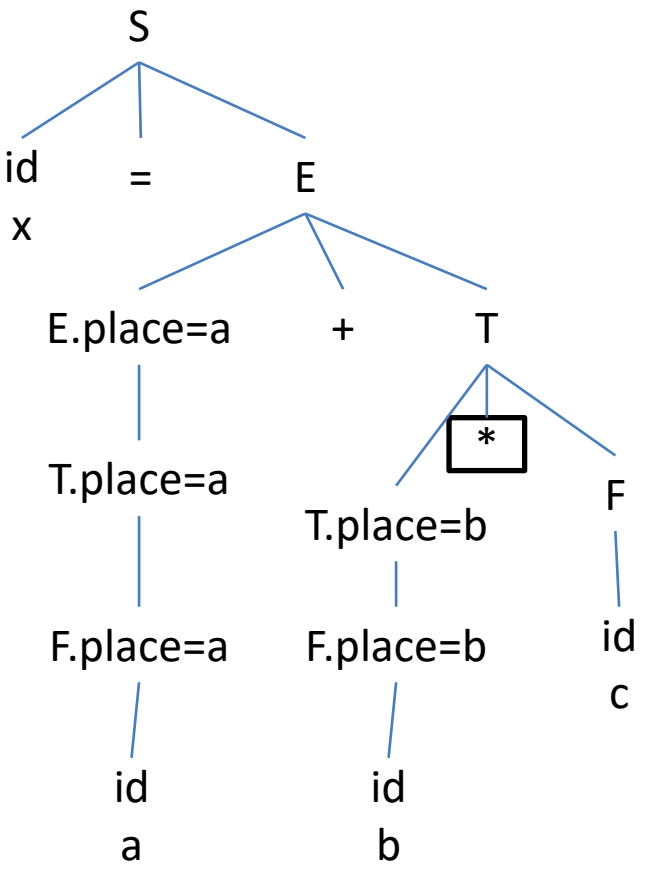
$E \rightarrow T$ {E.place = T.place}

$T \rightarrow T_1 * F$ {T.place = newTemp();
gen(E.place = T_1.place * F.place);}

$T \rightarrow F$ {T.place = F.place}

$F \rightarrow id$ {F.place = id.name}

x = a + b * c



Example 6

- SDT to generate three address code

$S \rightarrow id = E$ {gen(id.name = E.place}

$E \rightarrow E_1 + T$ {E.place = newTemp();
gen(E.place = E_1.place + T.place);}

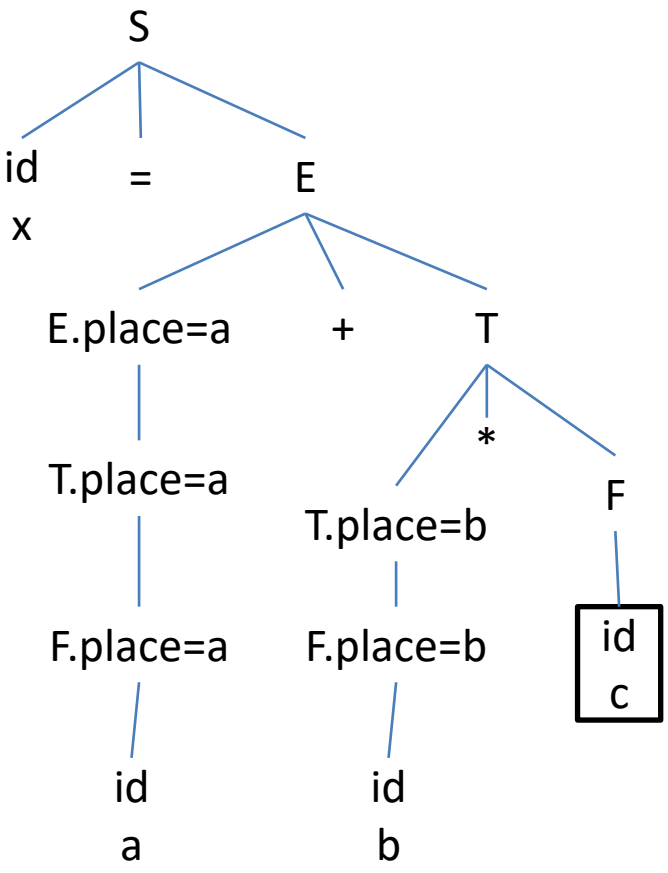
$E \rightarrow T$ {E.place = T.place}

$T \rightarrow T_1 * F$ {T.place = newTemp();
gen(E.place = T_1.place * F.place);}

$T \rightarrow F$ {T.place = F.place}

$F \rightarrow id$ {F.place = id.name}

x = a + b * c



Example 6

- SDT to generate three address code

$S \rightarrow id = E$ {gen(id.name = E.place}

$E \rightarrow E_1 + T$ {E.place = newTemp();
gen(E.place = E_1.place + T.place);}

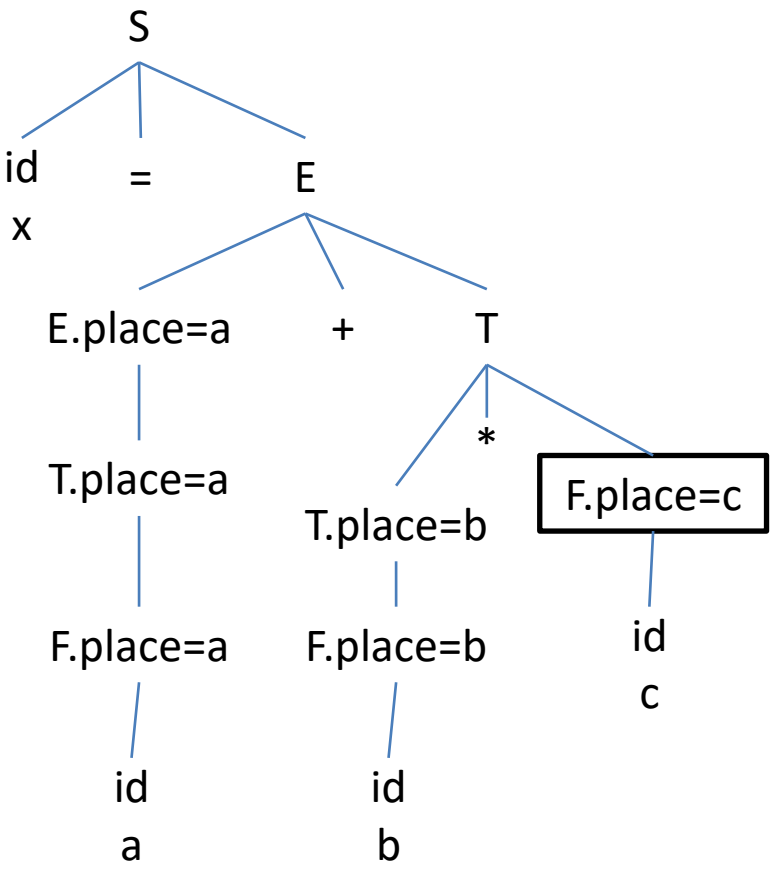
$E \rightarrow T$ {E.place = T.place}

$T \rightarrow T_1 * F$ {T.place = newTemp();
gen(E.place = T_1.place * F.place);}

$T \rightarrow F$ {T.place = F.place}

$F \rightarrow id$ {F.place = id.name}

x = a + b * c



Example 6

- SDT to generate three address code

$S \rightarrow id = E$ {gen(id.name = E.place}

$E \rightarrow E_1 + T$ {E.place = newTemp();
gen(E.place = E_1.place + T.place);}

$E \rightarrow T$ {E.place = T.place}

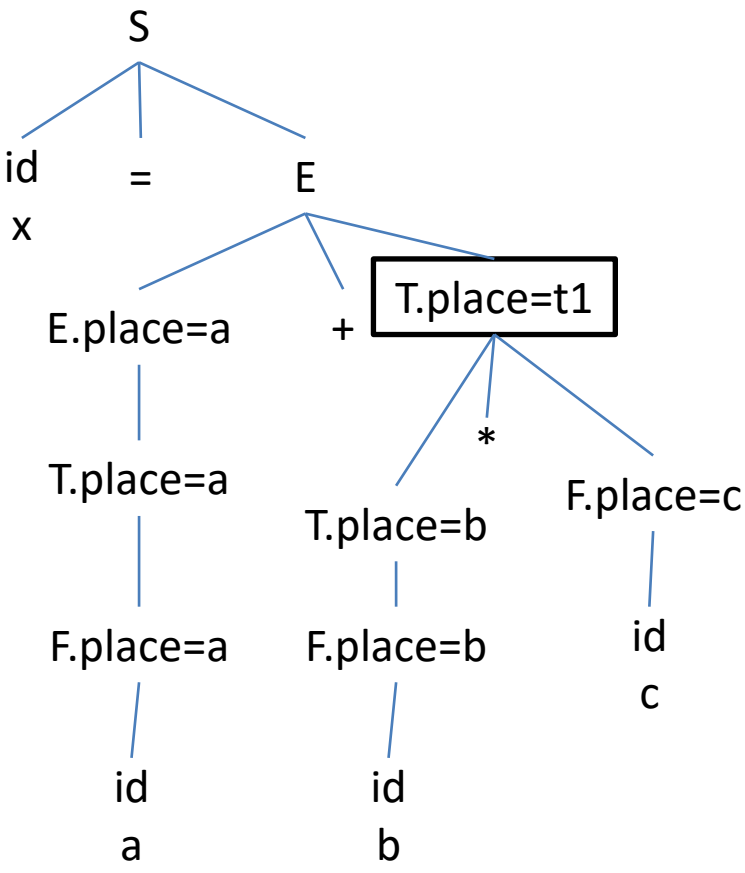
$T \rightarrow T_1 * F$ {T.place = newTemp();
gen(E.place = T_1.place * F.place);}

$T \rightarrow F$ {T.place = F.place}

$F \rightarrow id$ {F.place = id.name}

x = a + b * c

t1 = b * c



Example 6

- SDT to generate three address code

$S \rightarrow id = E$ {gen(id.name = E.place}

$E \rightarrow E_1 + T$ {E.place = newTemp();
gen(E.place = E_1.place + T.place);}

$E \rightarrow T$ {E.place = T.place}

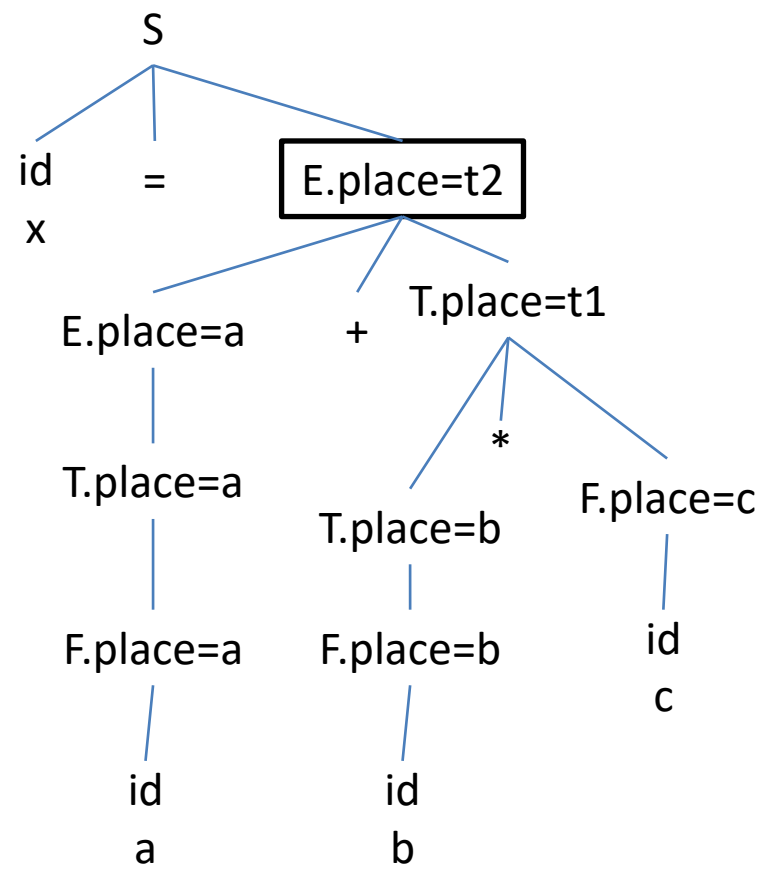
$T \rightarrow T_1 * F$ {T.place = newTemp();
gen(E.place = T_1.place * F.place);}

$T \rightarrow F$ {T.place = F.place}

$F \rightarrow id$ {F.place = id.name}

x = a + b * c

t1 = b * c
t2 = a + t1



Example 6

- SDT to generate three address code

S → id = E {gen(id.name = E.place}

E → E_1 + T {E.place = newTemp();
gen(E.place = E_1.place + T.place);}

E → T {E.place = T.place}

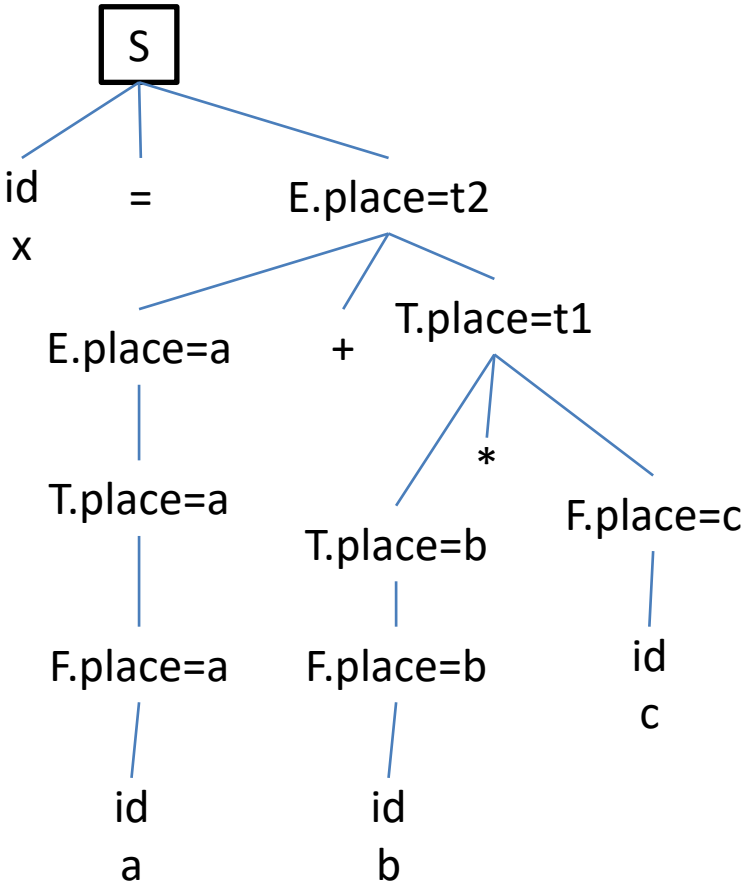
T → T_1 * F {T.place = newTemp();
gen(E.place = T_1.place * F.place);}

T → F {T.place = F.place}

F → id {F.place = id.name}

x = a + b * c

t1 = b * c
t2 = a + t1
x = t2



Example 7

- SDT to add type information in symbol table

- int x, y, z

x	int
y	int
z	int

Example 7

- SDT to add type information in symbol table

D → TL	{L.in=T.type}
T → int	{T.type = int}
T → char	{T.type=char}
L → L_1 ,id	{L_1.in = L.in; addtype(id.name , L_1.in)}
L → id	{addtype(id.name , L.in)}

The in attribute of L is an inherited attribute that takes the value of type attribute of T.

addtype() :- adds the type information in symbol table

Example 7

- SDT to add type information in symbol table

$D \rightarrow TL$	{L.in=T.type}	inherited attribute
$T \rightarrow int$	{T.type = int}	synthesized attribute
$T \rightarrow char$	{T.type=char}	synthesized attribute
$L \rightarrow L_1$,id	{L_1.in = L.in; addtype(id.name , L_1.in)}	inherited attribute
$L \rightarrow id$	{addtype(id.name , L.in)}	

addtype() :- adds the type information in symbol table

Example 7

- SDT to add type information in symbol table

L-attributed definition

$D \rightarrow TL$	{L.in=T.type}	inherited attribute
$T \rightarrow int$	{T.type = int}	synthesized attribute
$T \rightarrow char$	{T.type=char}	synthesized attribute
$L \rightarrow L_1$,id	{L_1.in = L.in; addtype(id.name , L_1.in)}	inherited attribute
$L \rightarrow id$	{addtype(id.name , L.in)}	

addtype() :- adds the type information in symbol table

Example 7
[L-attributed:- top-down left to right]

- SDT to add type information in symbol table

D → TL {L.in = T.type}

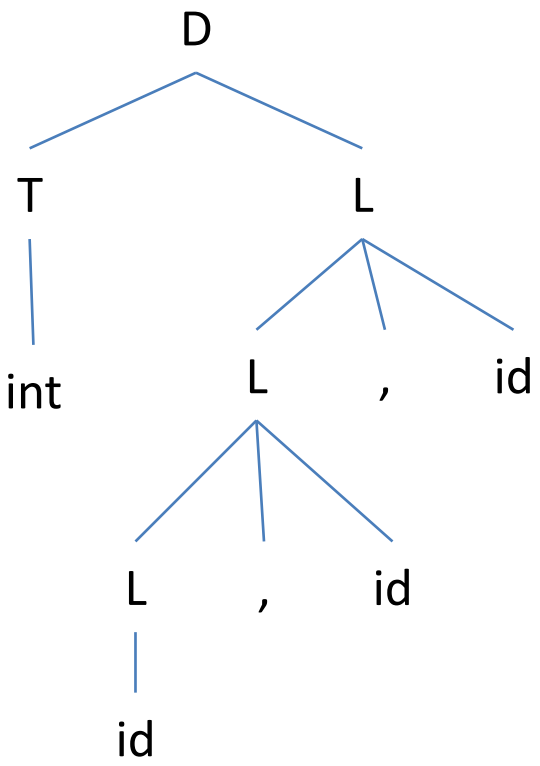
T → int {T.type = int}

T →char {T.type = char}

L → L_1 ,id {L_1.in = L.in; addtype(id.name , L_1.in)}

L→ id {addtype(id.name , L.in)}

int x, y, z



Example 7
[L-attributed:- top-down left to right]

- SDT to add type information in symbol table

$D \rightarrow TL$ {L.in = T.type}

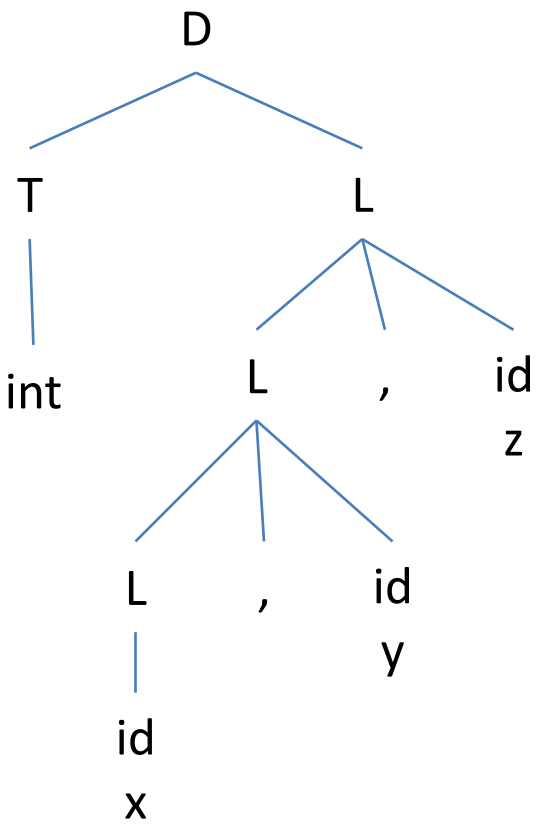
$T \rightarrow int$ {T.type = int}

$T \rightarrow char$ {T.type = char}

$L \rightarrow L_1$,id {L_1.in = L.in; addtype(id.name , L_1.in)}

$L \rightarrow id$ {addtype(id.name , L.in)}

int x, y, z



Example 7
[L-attributed:- top-down left to right]

- SDT to add type information in symbol table

D → TL {L.in = T.type}

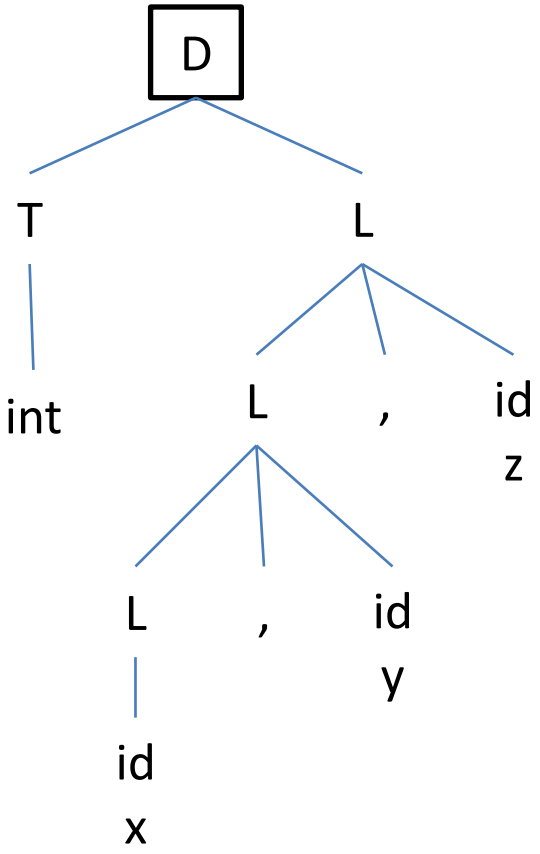
T → int {T.type = int}

T →char {T.type = char}

L → L_1 ,id {L_1.in = L.in; addtype(id.name , L_1.in)}

L→ id {addtype(id.name , L.in)}

int x, y, z



Example 7
[L-attributed:- top-down left to right]

- SDT to add type information in symbol table

D → TL {L.in = T.type}

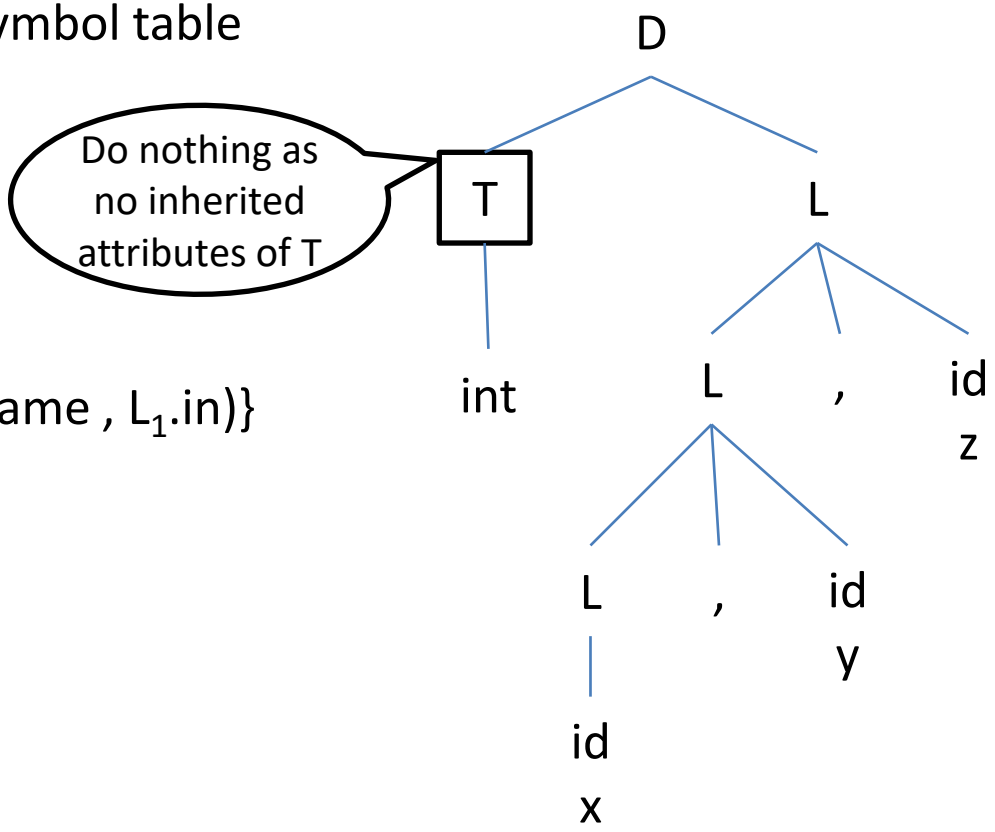
T → int {T.type = int}

T →char {T.type = char}

L → L_1 ,id {L_1.in = L.in; addtype(id.name , L_1.in)}

L→ id {addtype(id.name , L.in)}

int x, y, z



Example 7
[L-attributed:- top-down left to right]

- SDT to add type information in symbol table

$D \rightarrow TL$ {L.in = T.type}

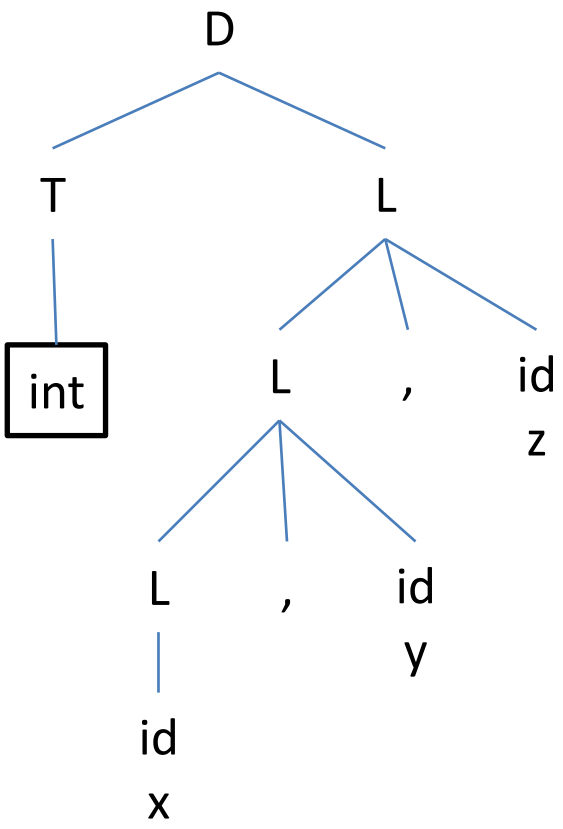
$T \rightarrow int$ {T.type = int}

$T \rightarrow char$ {T.type = char}

$L \rightarrow L_1$,id {L_1.in = L.in; addtype(id.name , L_1.in)}

$L \rightarrow id$ {addtype(id.name , L.in)}

int x, y, z



Example 7
[L-attributed:- top-down left to right]

- SDT to add type information in symbol table

D → TL {L.in = T.type}

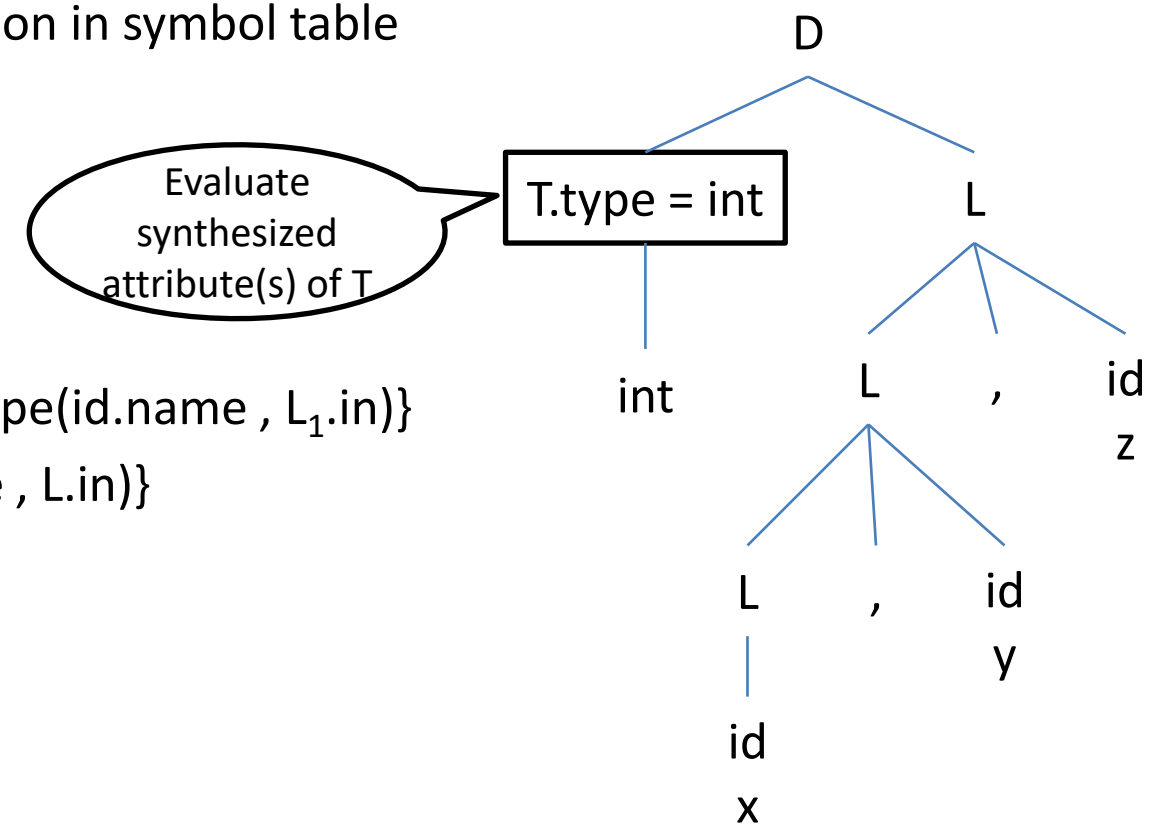
T → int {T.type = int}

T → char {T.type = char}

L → L_1 ,id {L_1.in = L.in; addtype(id.name , L_1.in)}

L → id {addtype(id.name , L.in)}

int x, y, z



Example 7
[L-attributed:- top-down left to right]

- SDT to add type information in symbol table

D → TL {L.in = T.type}

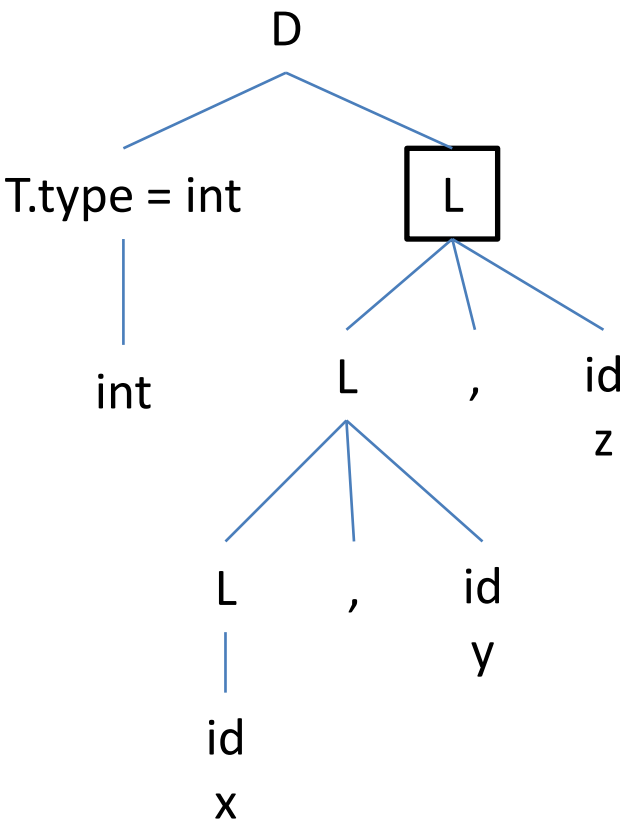
T → int {T.type = int}

T →char {T.type = char}

L → L_1 ,id {L_1.in = L.in; addtype(id.name , L_1.in)}

L→ id {addtype(id.name , L.in)}

int x, y, z



Example 7
[L-attributed:- top-down left to right]

- SDT to add type information in symbol table

$D \rightarrow TL$ {L.in = T.type}

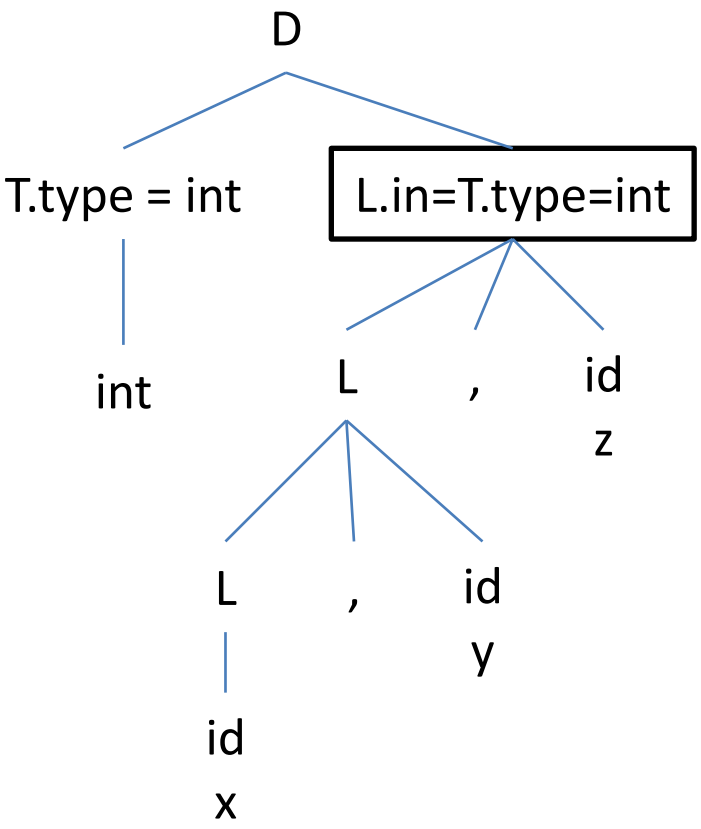
$T \rightarrow int$ {T.type = int}

$T \rightarrow char$ {T.type = char}

$L \rightarrow L_1$,id {L_1.in = L.in; addtype(id.name , L_1.in)}

$L \rightarrow id$ {addtype(id.name , L.in)}

int x, y, z



Example 7
[L-attributed:- top-down left to right]

- SDT to add type information in symbol table

D → TL {L.in = T.type}

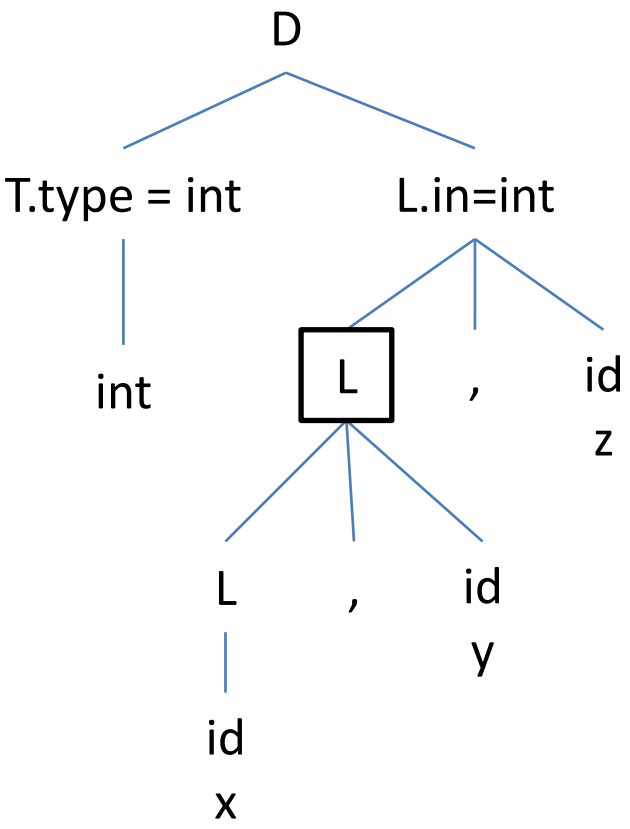
T → int {T.type = int}

T →char {T.type = char}

L → L_1 ,id {L_1.in = L.in; addtype(id.name , L_1.in)}

L→ id {addtype(id.name , L.in)}

int x, y, z



Example 7
[L-attributed:- top-down left to right]

- SDT to add type information in symbol table

D → TL {L.in = T.type}

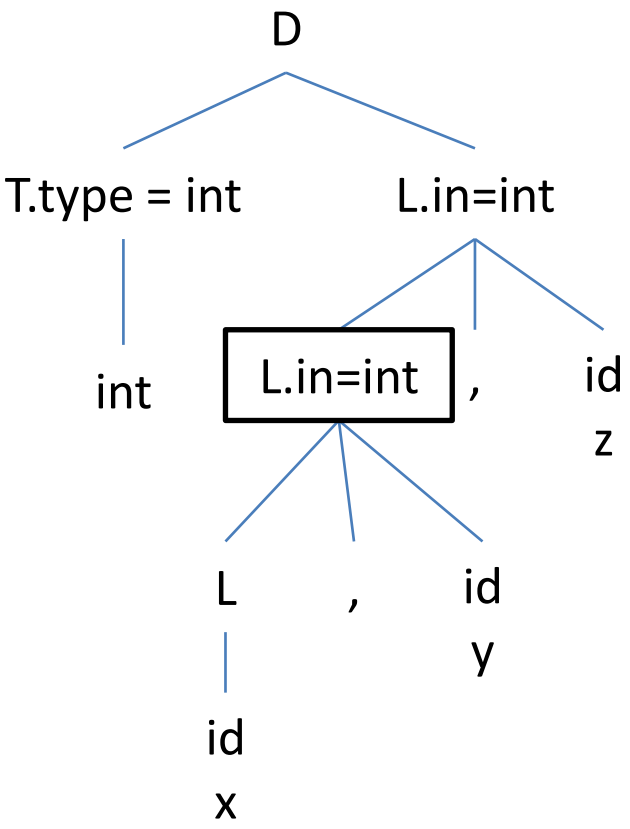
T → int {T.type = int}

T →char {T.type = char}

L → L_1 ,id {L_1.in = L.in; addtype(id.name , L_1.in)}

L→ id {addtype(id.name , L.in)}

int x, y, z



Example 7
[L-attributed:- top-down left to right]

- SDT to add type information in symbol table

D → TL {L.in = T.type}

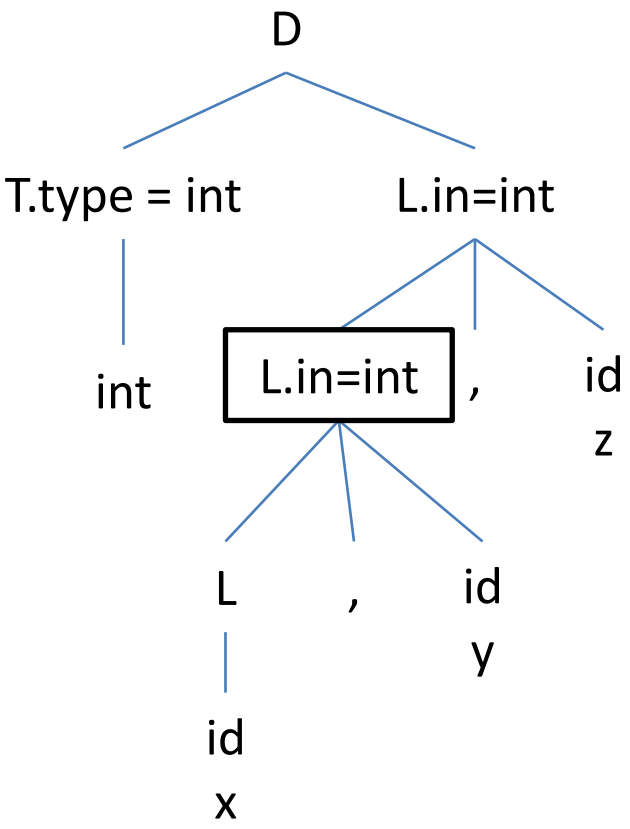
T → int {T.type = int}

T →char {T.type = char}

L → L_1 ,id {L_1.in = L.in; addtype(id.name , L_1.in)}

L→ id {addtype(id.name , L.in)}

int x, y, z



Example 7
[L-attributed:- top-down left to right]

- SDT to add type information in symbol table

D → TL {L.in = T.type}

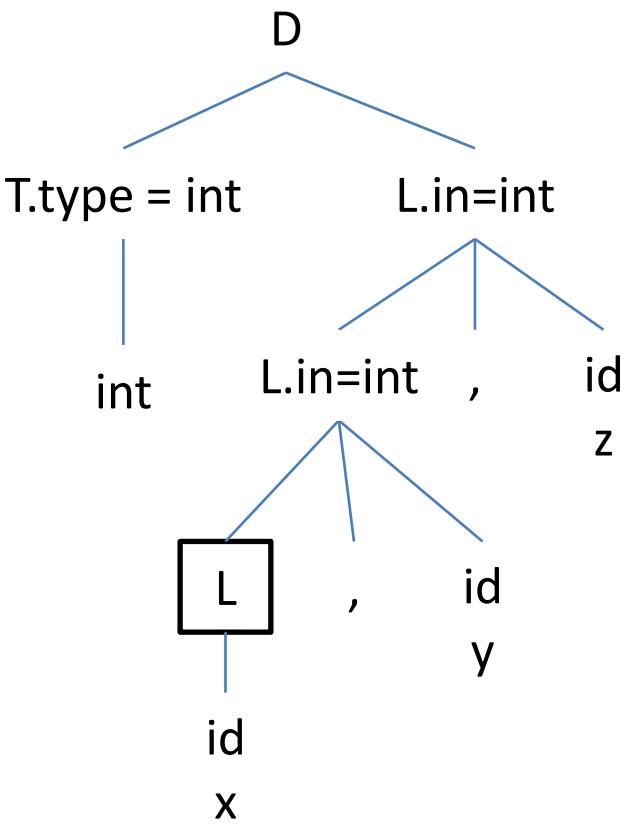
T → int {T.type = int}

T →char {T.type = char}

L → L_1 ,id {L_1.in = L.in; addtype(id.name , L_1.in)}

L→ id {addtype(id.name , L.in)}

int x, y, z



Example 7
[L-attributed:- top-down left to right]

- SDT to add type information in symbol table

D → TL {L.in = T.type}

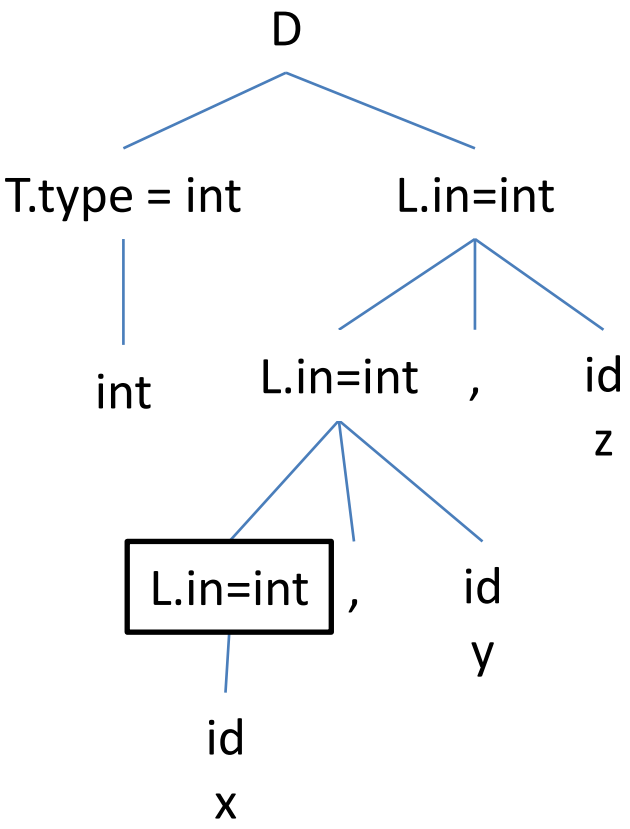
T → int {T.type = int}

T →char {T.type = char}

L → L_1 ,id {L_1.in = L.in; addtype(id.name , L_1.in)}

L→ id {addtype(id.name , L.in)}

int x, y, z



Example 7
[L-attributed:- top-down left to right]

- SDT to add type information in symbol table

D → TL {L.in = T.type}

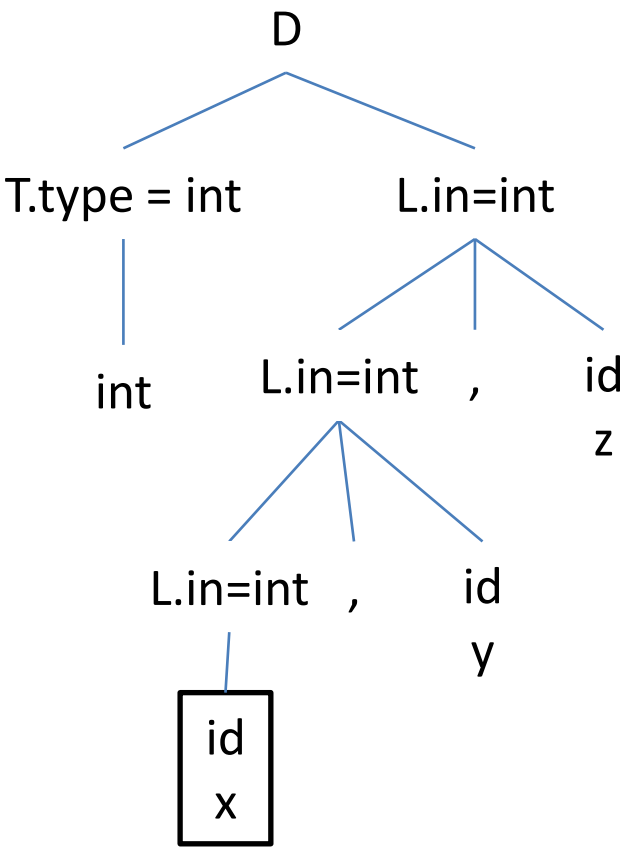
T → int {T.type = int}

T →char {T.type = char}

L → L_1 ,id {L_1.in = L.in; addtype(id.name , L_1.in)}

L→ id {addtype(id.name , L.in)}

int x, y, z



Example 7
[L-attributed:- top-down left to right]

- SDT to add type information in symbol table

D → TL {L.in = T.type}

T → int {T.type = int}

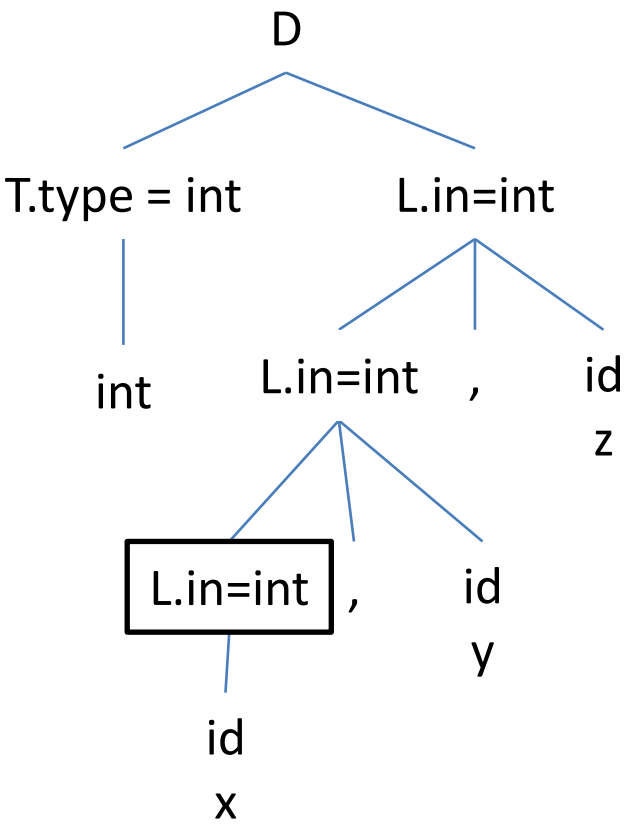
T →char {T.type = char}

L → L_1 ,id {L_1.in = L.in; addtype(id.name , L_1.in)}

L→ id {addtype(id.name , L.in)}

int x, y, z

x	int



Example 7
[L-attributed:- top-down left to right]

- SDT to add type information in symbol table

D → TL {L.in = T.type}

T → int {T.type = int}

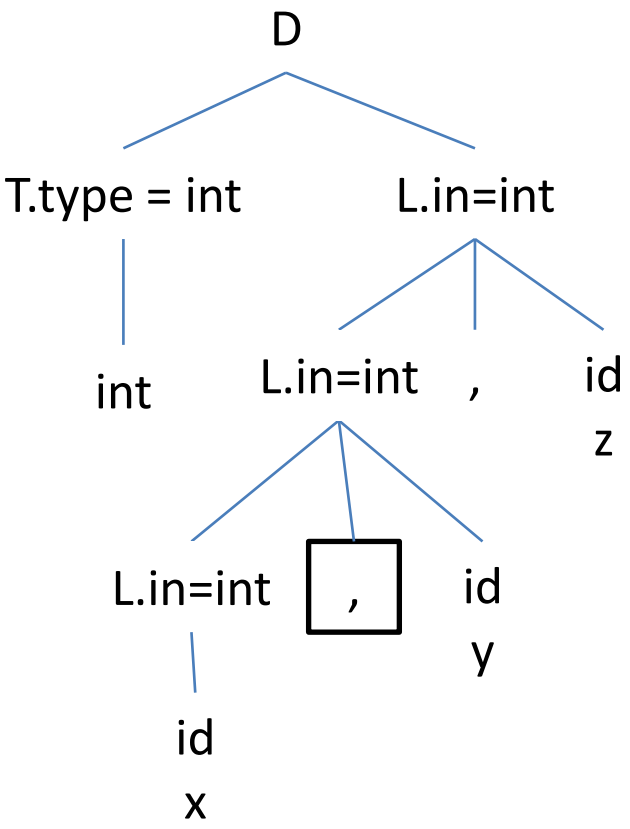
T →char {T.type = char}

L → L_1 ,id {L_1.in = L.in; addtype(id.name , L_1.in)}

L→ id {addtype(id.name , L.in)}

int x, y, z

x	int



Example 7
[L-attributed:- top-down left to right]

- SDT to add type information in symbol table

D → TL {L.in = T.type}

T → int {T.type = int}

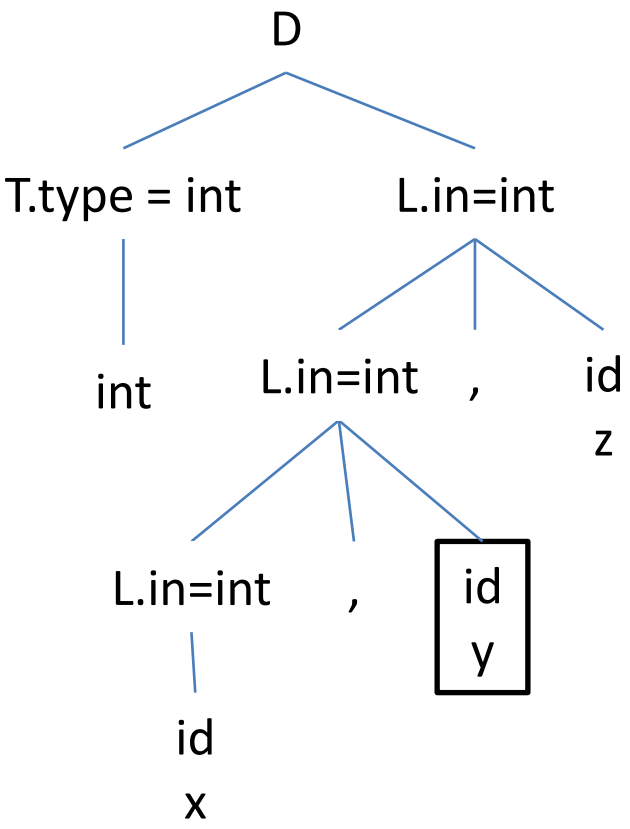
T →char {T.type = char}

L → L_1 ,id {L_1.in = L.in; addtype(id.name , L_1.in)}

L→ id {addtype(id.name , L.in)}

int x, y, z

x	int



Example 7
[L-attributed:- top-down left to right]

- SDT to add type information in symbol table

D → TL {L.in = T.type}

T → int {T.type = int}

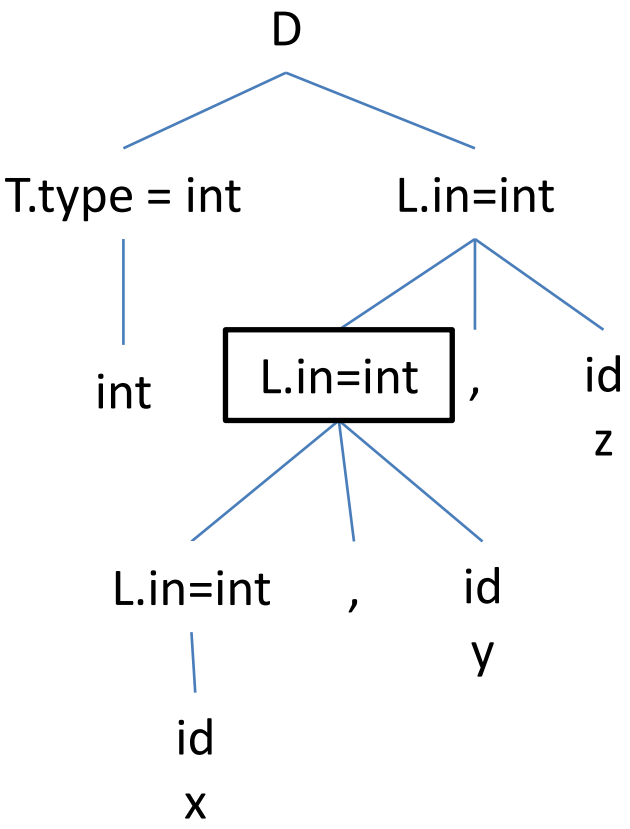
T →char {T.type = char}

L → L_1 ,id {L_1.in = L.in; addtype(id.name , L_1.in)}

L→ id {addtype(id.name , L.in)}

int x, y, z

x	int
y	int



Example 7
[L-attributed:- top-down left to right]

- SDT to add type information in symbol table

D → TL {L.in = T.type}

T → int {T.type = int}

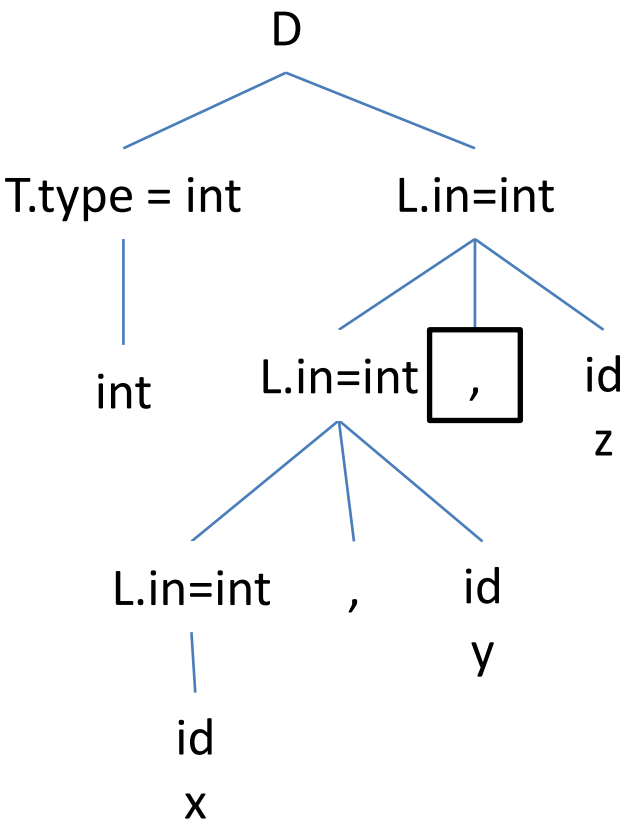
T →char {T.type = char}

L → L_1 ,id {L_1.in = L.in; addtype(id.name , L_1.in)}

L→ id {addtype(id.name , L.in)}

int x, y, z

x	int
y	int



Example 7
[L-attributed:- top-down left to right]

- SDT to add type information in symbol table

D → TL {L.in = T.type}

T → int {T.type = int}

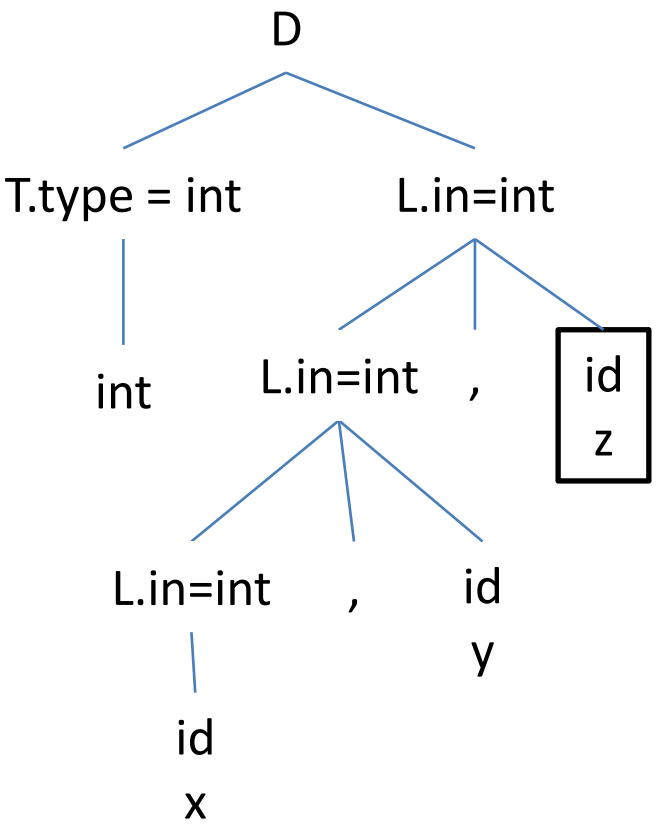
T →char {T.type = char}

L → L_1 ,id {L_1.in = L.in; addtype(id.name , L_1.in)}

L→ id {addtype(id.name , L.in)}

int x, y, z

x	int
y	int



Example 7
[L-attributed:- top-down left to right]

- SDT to add type information in symbol table

D → TL {L.in = T.type}

T → int {T.type = int}

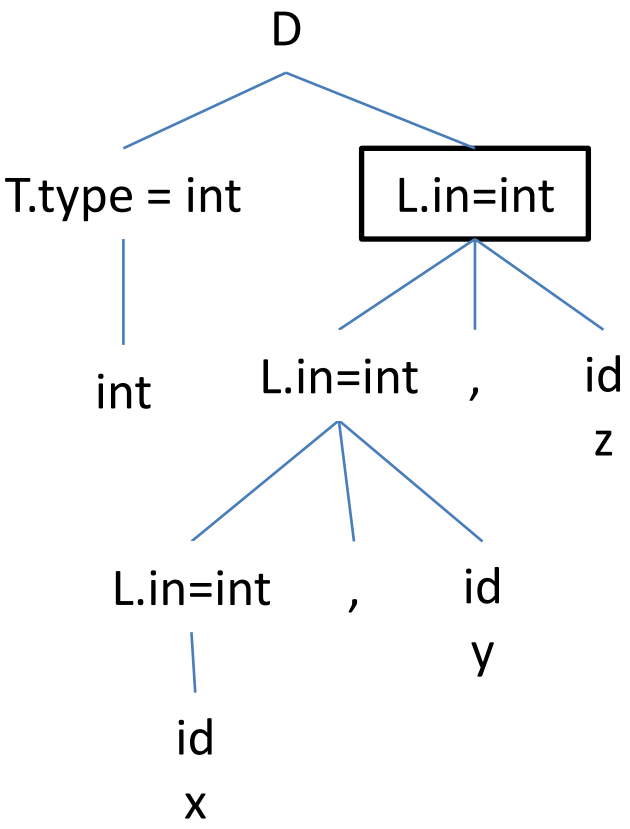
T →char {T.type = char}

L → L_1 ,id {L_1.in = L.in; addtype(id.name , L_1.in)}

L→ id {addtype(id.name , L.in)}

int x, y, z

x	int
y	int
z	int



Example 7
[L-attributed:- top-down left to right]

- SDT to add type information in symbol table

D → TL {L.in = T.type}

T → int {T.type = int}

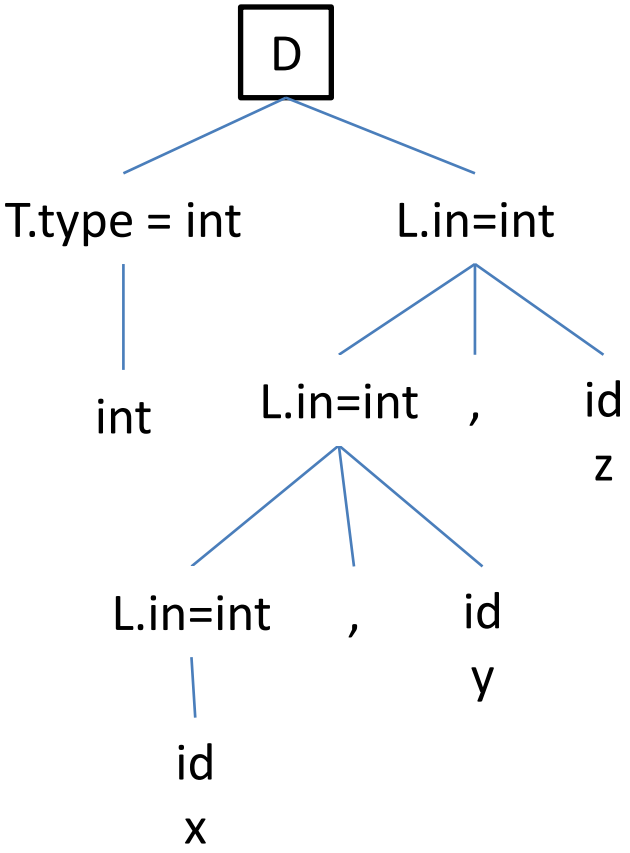
T →char {T.type = char}

L → L_1 ,id {L_1.in = L.in; addtype(id.name , L_1.in)}

L→ id {addtype(id.name , L.in)}

int x, y, z

x	int
y	int
z	int



Example 7 (second method)

- SDT to add type information in symbol table

$D \rightarrow D_1$, id {add_type(id.name, D_1.type), D.type= D_1.type}

$D \rightarrow$ T id {add_type(id.name, T.type), D.type=T.type}

$T \rightarrow$ int {T.type = int}

$T \rightarrow$ char {T.type=char}

add_type() :- adds the type information in symbol table

Example 7 (second method)

- SDT to add type information in symbol table

S-attributed definition

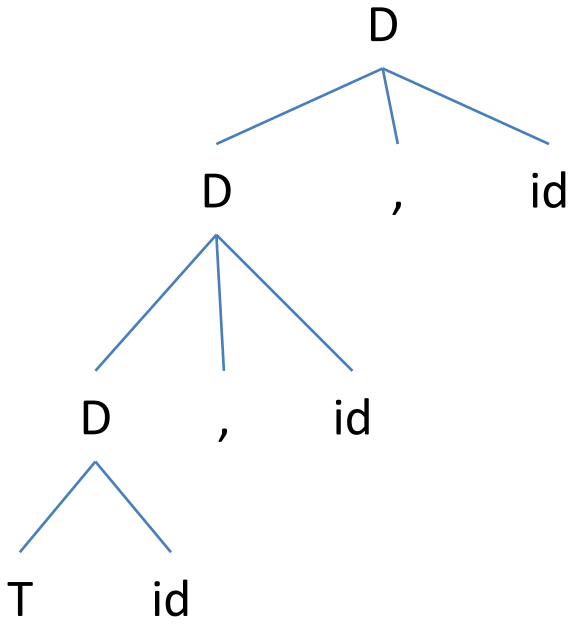
$D \rightarrow D_1$, id	{add_type(id.name, D_1.type), D.type=D_1.type}
$D \rightarrow$ T id	{add_type(id.name, T.type), D.type=T.type}
$T \rightarrow$ int	{T.type = int}
$T \rightarrow$ char	{T.type=char}

add_type() :- adds the type information in symbol table

Example 7 (second method) [S-attributed:- bottom-up left to right]

- SDT to add type information in symbol table

$D \rightarrow D_1$, id	{add_type(id.name, D_1.type) , D.type= D_1.type}
$D \rightarrow T$ id	{add_type(id.name, T.type), D.type=T.type}
$T \rightarrow$ int	{T.type = int}
$T \rightarrow$ char	{T.type=char}



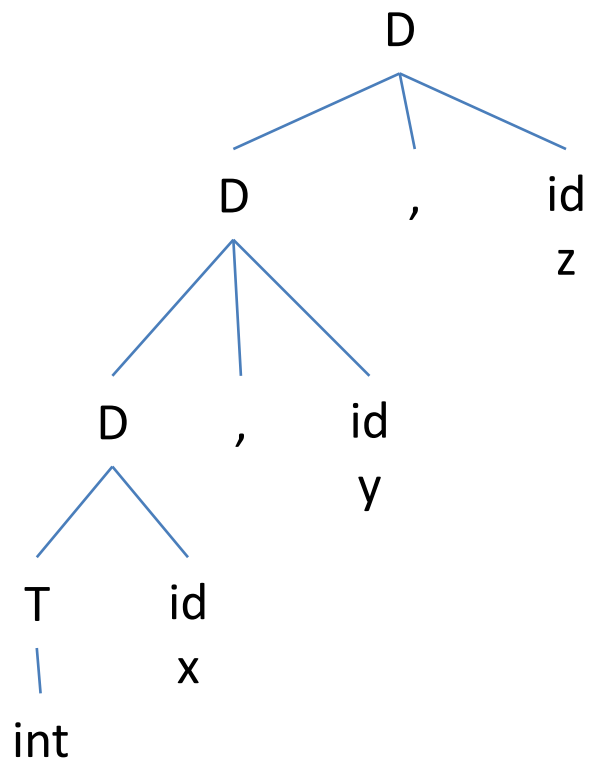
int x, y, z

Example 7 (second method) [S-attributed:- bottom-up left to right]

- SDT to add type information in symbol table

$D \rightarrow D_1$, id	{add_type(id.name, D_1.type) , D.type= D_1.type}
$D \rightarrow T$ id	{add_type(id.name, T.type), D.type=T.type}
$T \rightarrow$ int	{T.type = int}
$T \rightarrow$ char	{T.type=char}

int x, y, z

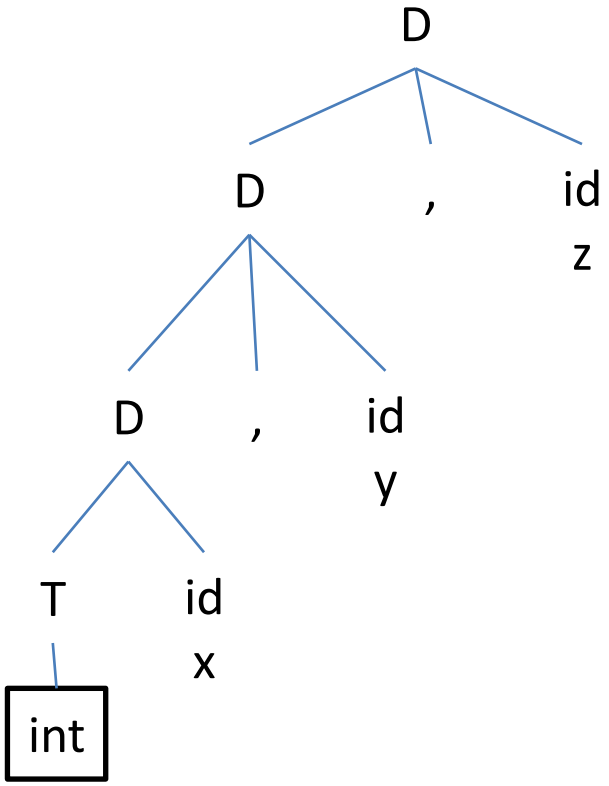


Example 7 (second method) [S-attributed:- bottom-up left to right]

- SDT to add type information in symbol table

$D \rightarrow D_1$, id	{add_type(id.name, D_1.type) , D.type= D_1.type}
$D \rightarrow T$ id	{add_type(id.name, T.type), D.type=T.type}
$T \rightarrow$ int	{T.type = int}
$T \rightarrow$ char	{T.type=char}

int x, y, z

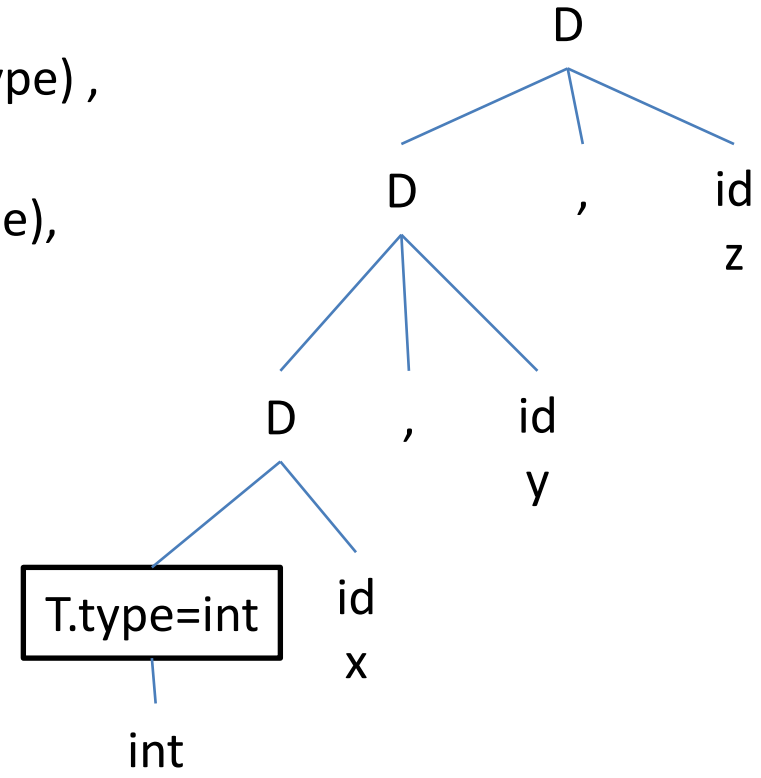


Example 7 (second method) [S-attributed:- bottom-up left to right]

- SDT to add type information in symbol table

$D \rightarrow D_1$, id	{add_type(id.name, D_1.type) , D.type= D_1.type}
$D \rightarrow T$ id	{add_type(id.name, T.type), D.type=T.type}
$T \rightarrow$ int	{T.type = int}
$T \rightarrow$ char	{T.type=char}

int x, y, z

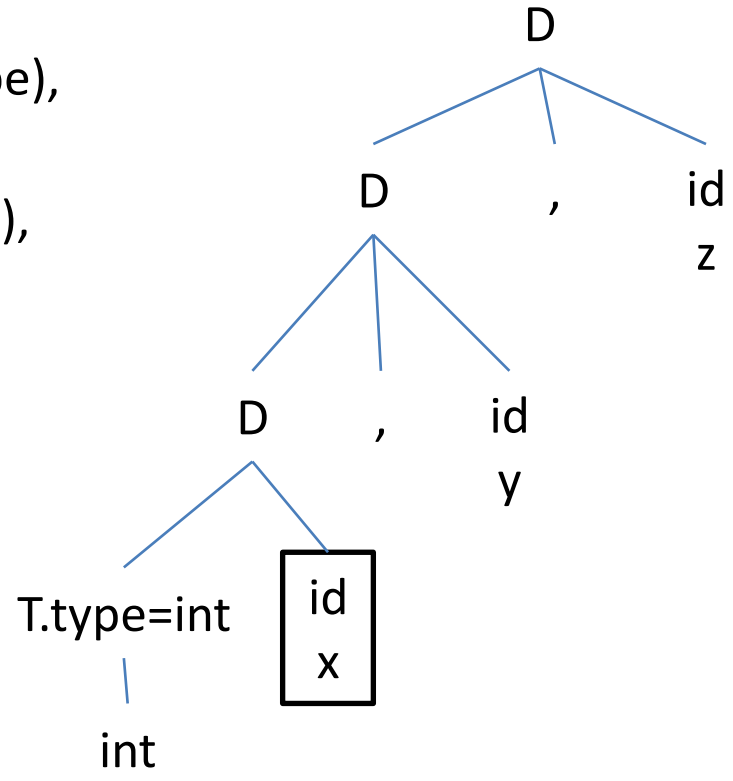


Example 7 (second method) [S-attributed:- bottom-up left to right]

- SDT to add type information in symbol table

$D \rightarrow D_1$, id	{add_type(id.name, D_1.type), D.type= D_1.type}
$D \rightarrow$ T id	{add_type(id.name, T.type), D.type=T.type}
$T \rightarrow$ int	{T.type = int}
$T \rightarrow$ char	{T.type=char}

int x, y, z



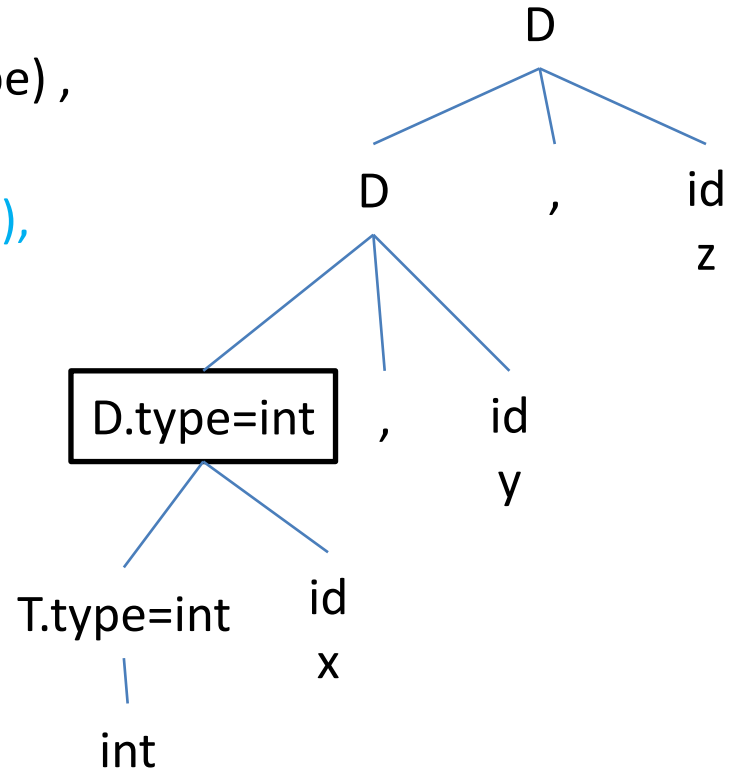
Example 7 (second method) [S-attributed:- bottom-up left to right]

- SDT to add type information in symbol table

$D \rightarrow D_1$, id	{add_type(id.name, D_1.type) , D.type= D_1.type}
$D \rightarrow T$ id	{add_type(id.name, T.type), D.type=T.type}
$T \rightarrow$ int	{T.type = int}
$T \rightarrow$ char	{T.type=char}

int x, y, z

x	int



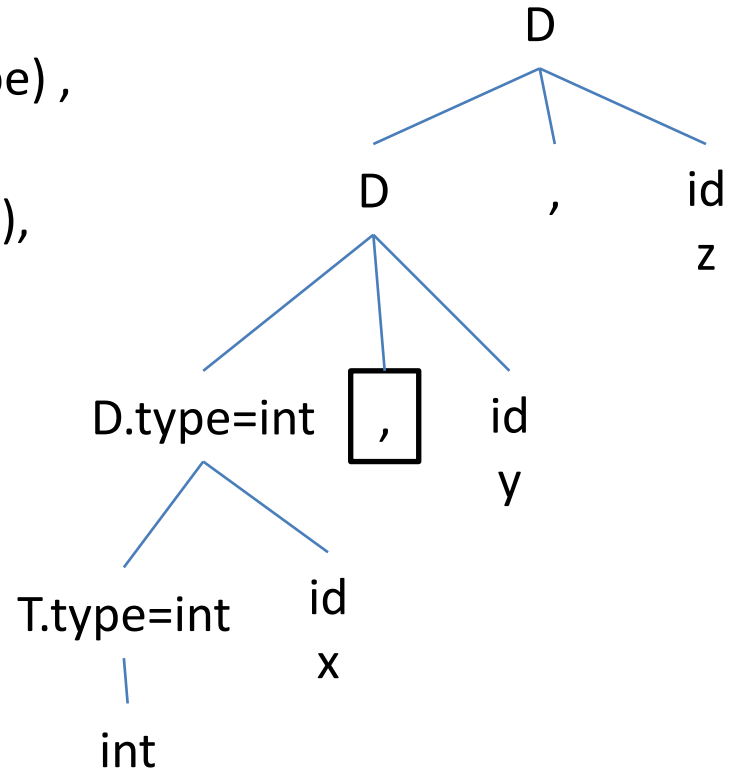
Example 7 (second method) [S-attributed:- bottom-up left to right]

- SDT to add type information in symbol table

$D \rightarrow D_1$, id	{add_type(id.name, D_1.type) , D.type= D_1.type}
$D \rightarrow T$ id	{add_type(id.name, T.type), D.type=T.type}
$T \rightarrow$ int	{T.type = int}
$T \rightarrow$ char	{T.type=char}

int x, y, z

x	int



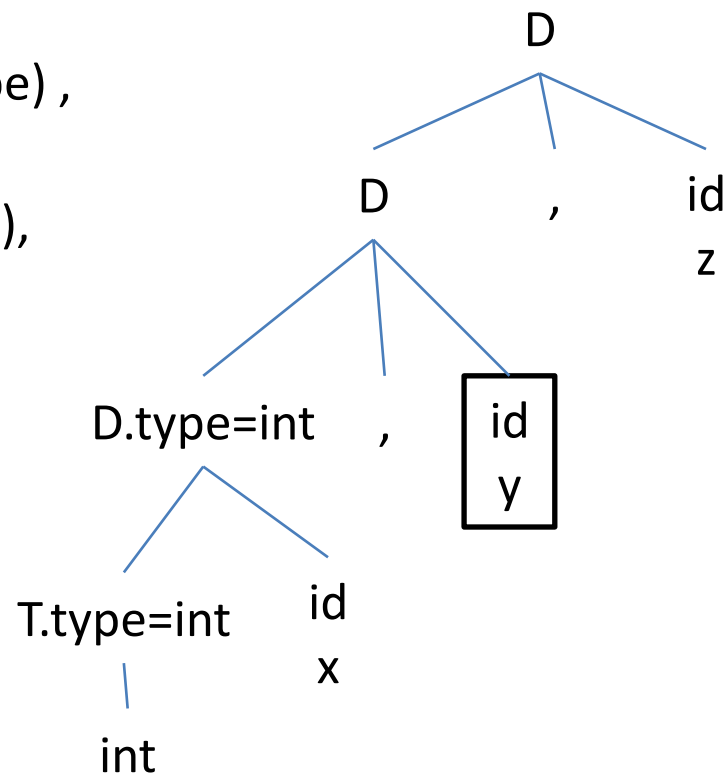
Example 7 (second method) [S-attributed:- bottom-up left to right]

- SDT to add type information in symbol table

$D \rightarrow D_1$, id	{add_type(id.name, D_1.type) , D.type= D_1.type}
$D \rightarrow T$ id	{add_type(id.name, T.type), D.type=T.type}
$T \rightarrow$ int	{T.type = int}
$T \rightarrow$ char	{T.type=char}

int x, y, z

x	int



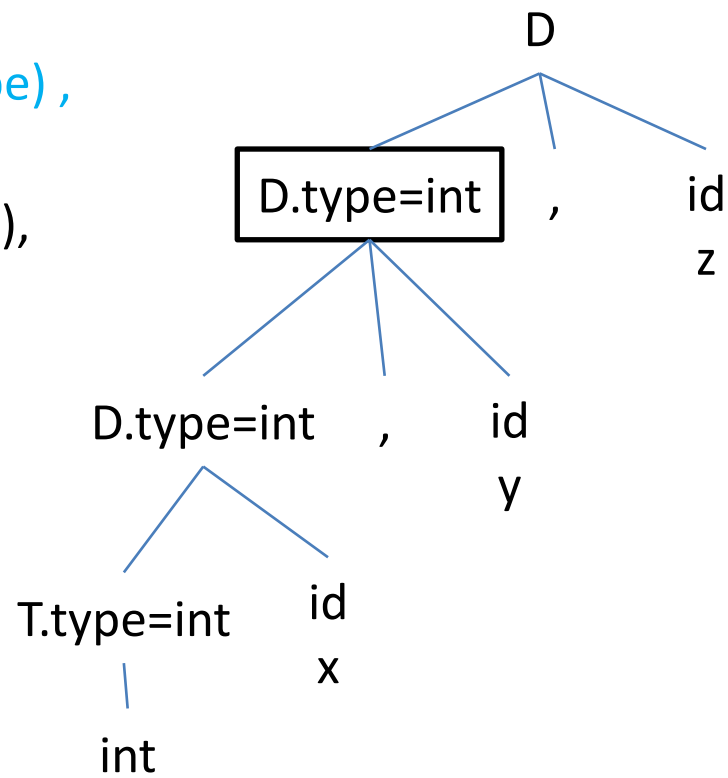
Example 7 (second method) [S-attributed:- bottom-up left to right]

- SDT to add type information in symbol table

$D \rightarrow D_1$, id	{add_type(id.name, D_1.type) , D.type= D_1.type}
D → T id	{add_type(id.name, T.type), D.type=T.type}
T → int	{T.type = int}
T → char	{T.type=char}

int x, y, z

x	int
y	int



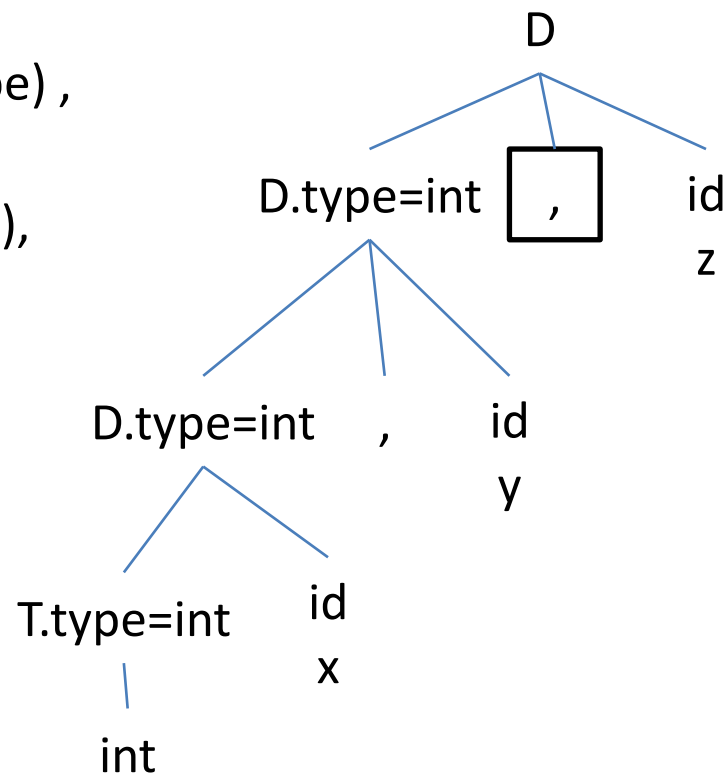
Example 7 (second method) [S-attributed:- bottom-up left to right]

- SDT to add type information in symbol table

$D \rightarrow D_1$, id	{add_type(id.name, D_1.type) , D.type= D_1.type}
$D \rightarrow T$ id	{add_type(id.name, T.type), D.type=T.type}
$T \rightarrow$ int	{T.type = int}
$T \rightarrow$ char	{T.type=char}

int x, y, z

x	int
y	int



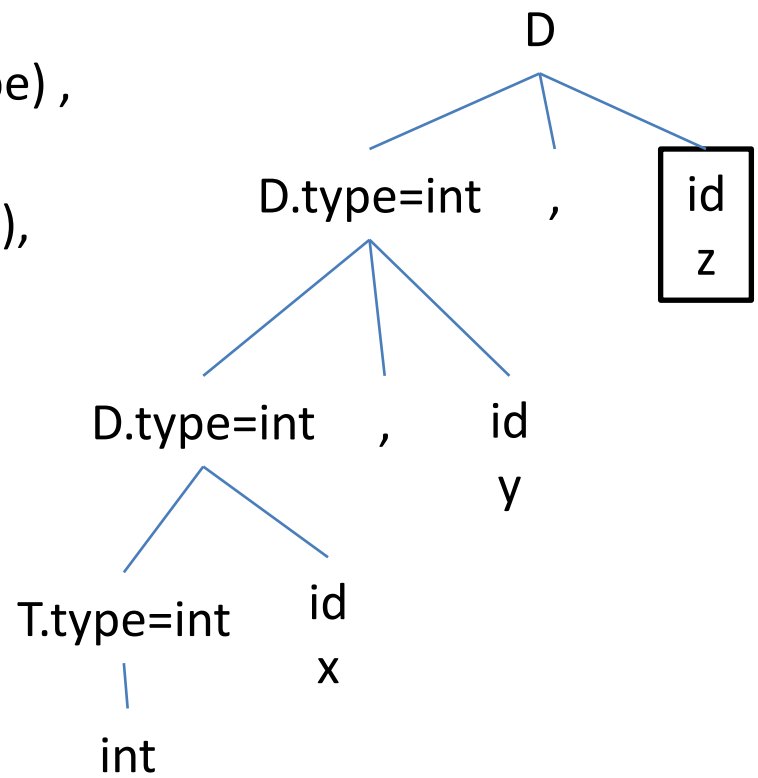
Example 7 (second method) [S-attributed:- bottom-up left to right]

- SDT to add type information in symbol table

$D \rightarrow D_1$, id	{add_type(id.name, D_1.type) , D.type= D_1.type}
$D \rightarrow T$ id	{add_type(id.name, T.type), D.type=T.type}
$T \rightarrow$ int	{T.type = int}
$T \rightarrow$ char	{T.type=char}

int x, y, z

x	int
y	int



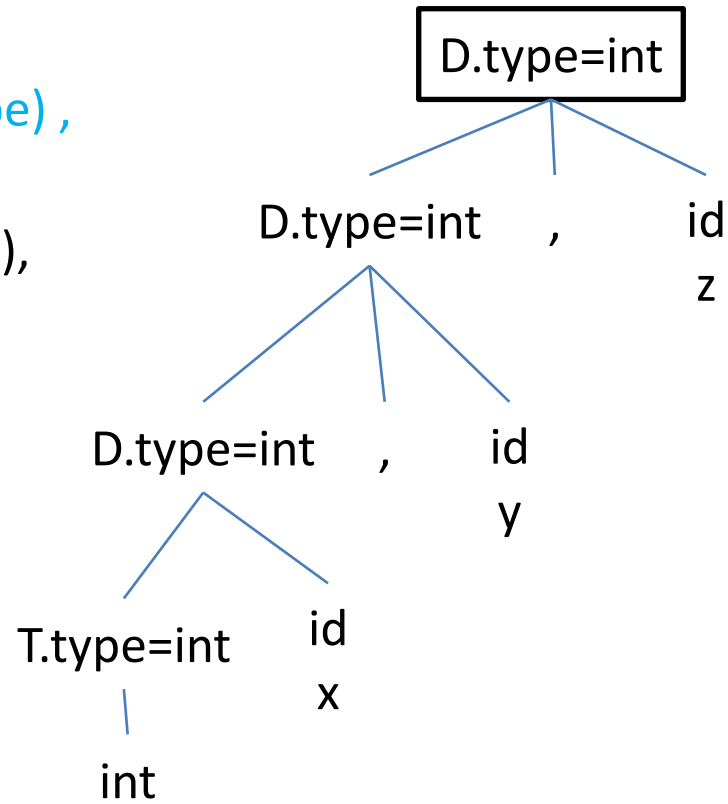
Example 7 (second method) [S-attributed:- bottom-up left to right]

- SDT to add type information in symbol table

$D \rightarrow D_1$, id	{add_type(id.name, D_1.type) , D.type= D_1.type}
D → T id	{add_type(id.name, T.type), D.type=T.type}
T → int	{T.type = int}
T → char	{T.type=char}

int x, y, z

x	int
y	int
z	int



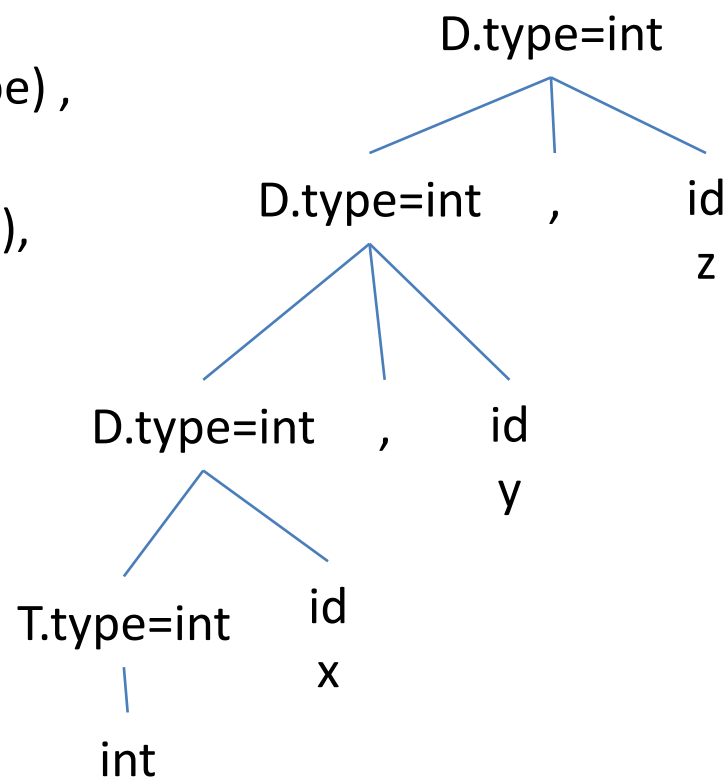
Example 7 (second method) [S-attributed:- bottom-up left to right]

- SDT to add type information in symbol table

$D \rightarrow D_1$, id	{add_type(id.name, D_1.type) , D.type= D_1.type}
$D \rightarrow T$ id	{add_type(id.name, T.type), D.type=T.type}
$T \rightarrow$ int	{T.type = int}
$T \rightarrow$ char	{T.type=char}

int x, y, z

x	int
y	int
z	int



Example 8

- SDT to check the type of an expression

$E \rightarrow E_1 + E_2$ {if((E_1.type == E_2.type) && (E_1.type == int))
then E.type = int else error;}

$E \rightarrow E_1 == E_2$ {if((E_1.type == E_2.type) && (E_1.type == int|boolean))
then E.type = boolean else error;}

$E \rightarrow (E_1)$ {E.type = E_1.type}

$E \rightarrow$ num {E.type = int}

$E \rightarrow$ true {E.type = boolean}

$E \rightarrow$ false {E.type = boolean}

Example 8

- SDT to check the type of an expression

$E \rightarrow E_1 + E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int))
then E.type = int else error;}

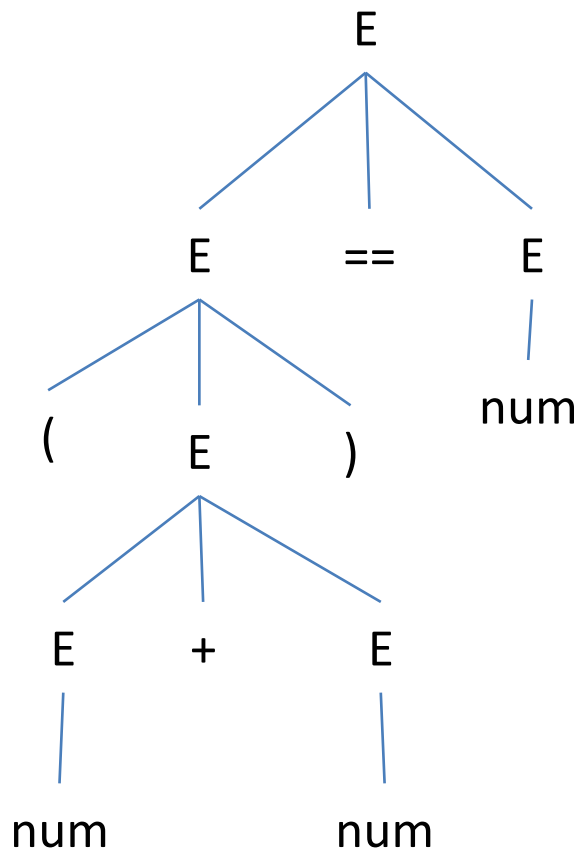
$E \rightarrow E_1 == E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int|boolean))
then E.type = boolean else error;}

$E \rightarrow (E_1)$ {E.type = E_1.type}

$E \rightarrow$ num {E.type = int}

$E \rightarrow$ true {E.type = boolean}

$E \rightarrow$ false {E.type = boolean}



(2 + 3) == 8

Example 8
[S-attributed:- bottom-up left to right]

- SDT to check the type of an expression

$E \rightarrow E_1 + E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int))
then E.type = int else error;}

$E \rightarrow E_1 == E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int|boolean))
then E.type = boolean else error;}

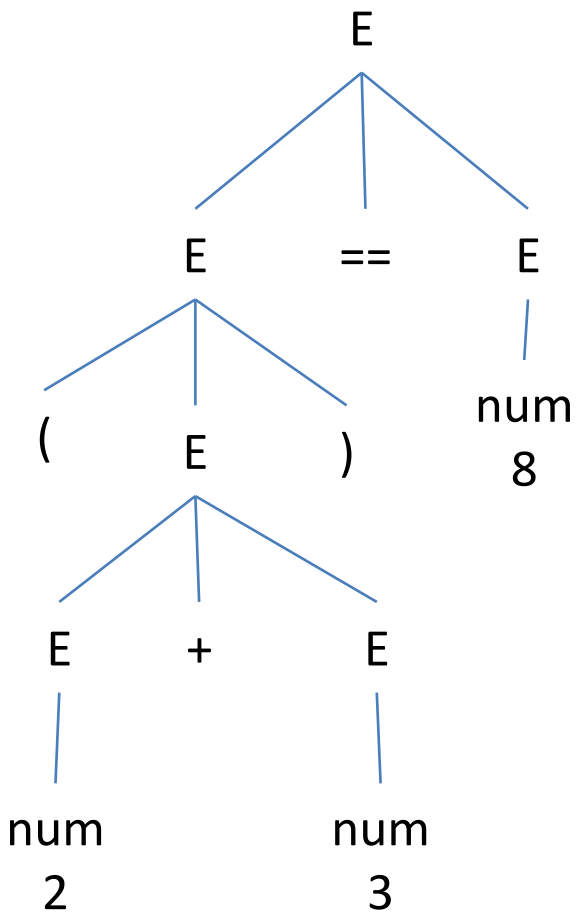
$E \rightarrow (E_1)$ {E.type = E_1.type}

$E \rightarrow$ num {E.type = int}

$E \rightarrow$ true {E.type = boolean}

$E \rightarrow$ false {E.type = boolean}

(2 + 3) == 8



Example 8
[S-attributed:- bottom-up left to right]

- SDT to check the type of an expression

$E \rightarrow E_1 + E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int))
then E.type = int else error;}

$E \rightarrow E_1 == E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int|boolean))
then E.type = boolean else error;}

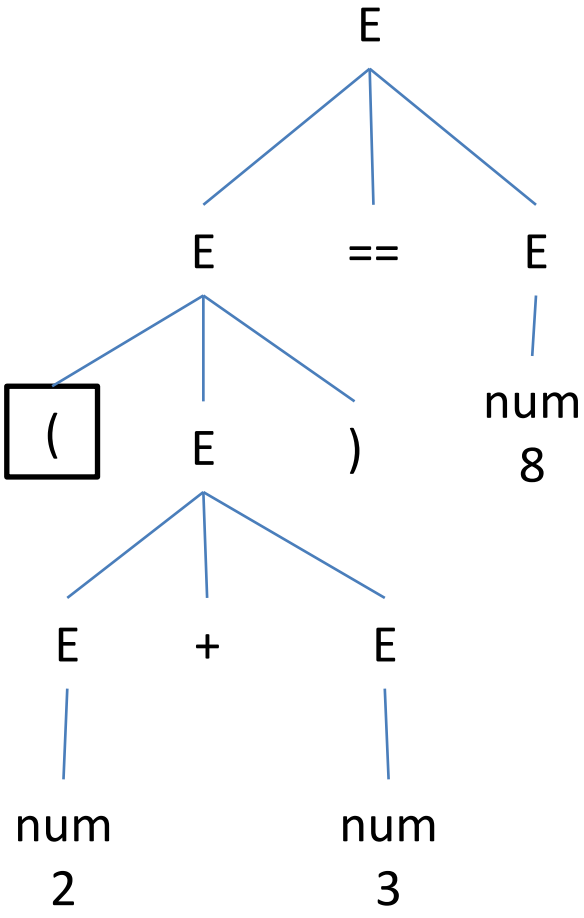
$E \rightarrow (E_1)$ {E.type = E_1.type}

$E \rightarrow$ num {E.type = int}

$E \rightarrow$ true {E.type = boolean}

$E \rightarrow$ false {E.type = boolean}

(2 + 3) == 8



Example 8
[S-attributed:- bottom-up left to right]

- SDT to check the type of an expression

$E \rightarrow E_1 + E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int))
then E.type = int else error;}

$E \rightarrow E_1 == E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int|boolean))
then E.type = boolean else error;}

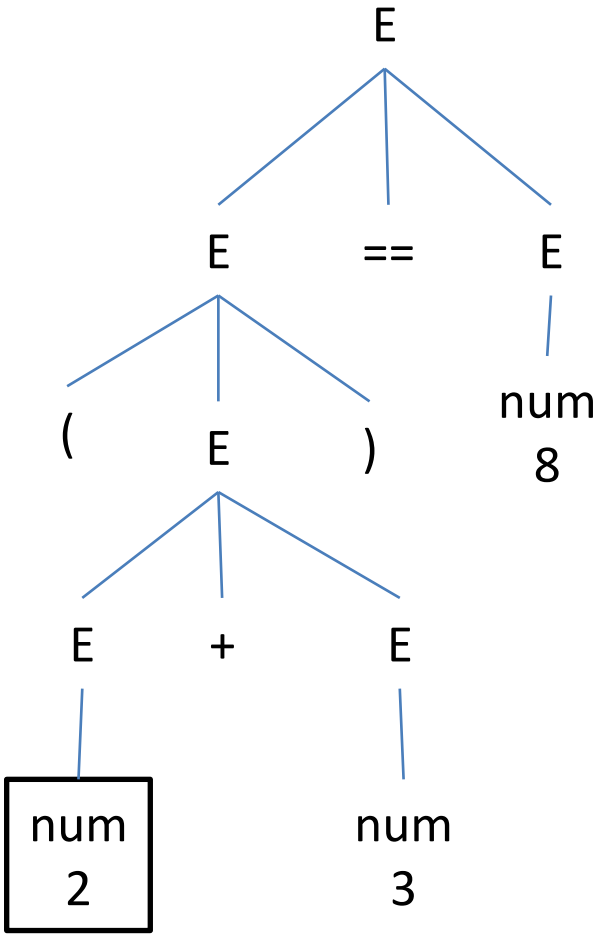
$E \rightarrow (E_1)$ {E.type = E_1.type}

$E \rightarrow$ num {E.type = int}

$E \rightarrow$ true {E.type = boolean}

$E \rightarrow$ false {E.type = boolean}

(2 + 3) == 8



Example 8
[S-attributed:- bottom-up left to right]

- SDT to check the type of an expression

$E \rightarrow E_1 + E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int))
then E.type = int else error;}

$E \rightarrow E_1 == E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int|boolean))
then E.type = boolean else error;}

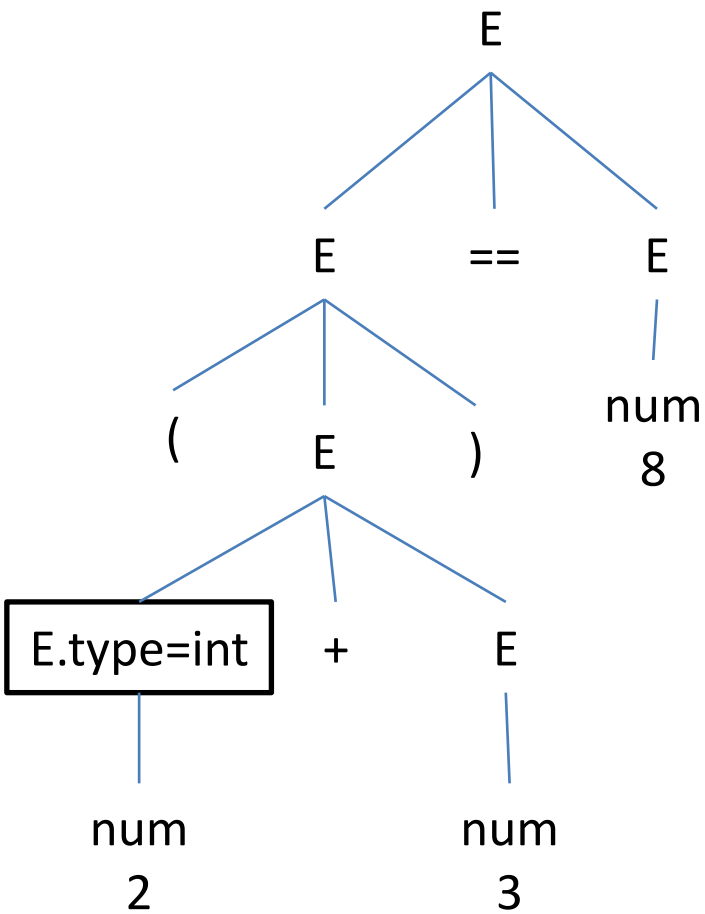
$E \rightarrow (E_1)$ {E.type = E_1.type}

$E \rightarrow$ num {E.type = int}

$E \rightarrow$ true {E.type = boolean}

$E \rightarrow$ false {E.type = boolean}

(2 + 3) == 8



Example 8
[S-attributed:- bottom-up left to right]

- SDT to check the type of an expression

$E \rightarrow E_1 + E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int))
then E.type = int else error;}

$E \rightarrow E_1 == E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int|boolean))
then E.type = boolean else error;}

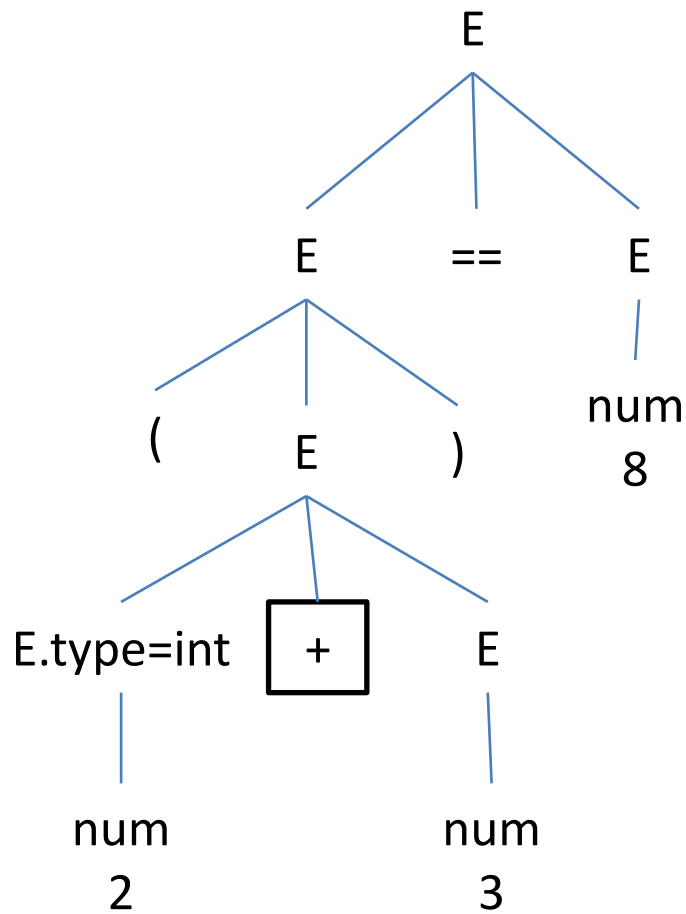
$E \rightarrow (E_1)$ {E.type = E_1.type}

E → num {E.type = int}

E → true {E.type = boolean}

E → false {E.type = boolean}

(2 + 3) == 8



Example 8
[S-attributed:- bottom-up left to right]

- SDT to check the type of an expression

$E \rightarrow E_1 + E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int))
then E.type = int else error;}

$E \rightarrow E_1 == E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int|boolean))
then E.type = boolean else error;}

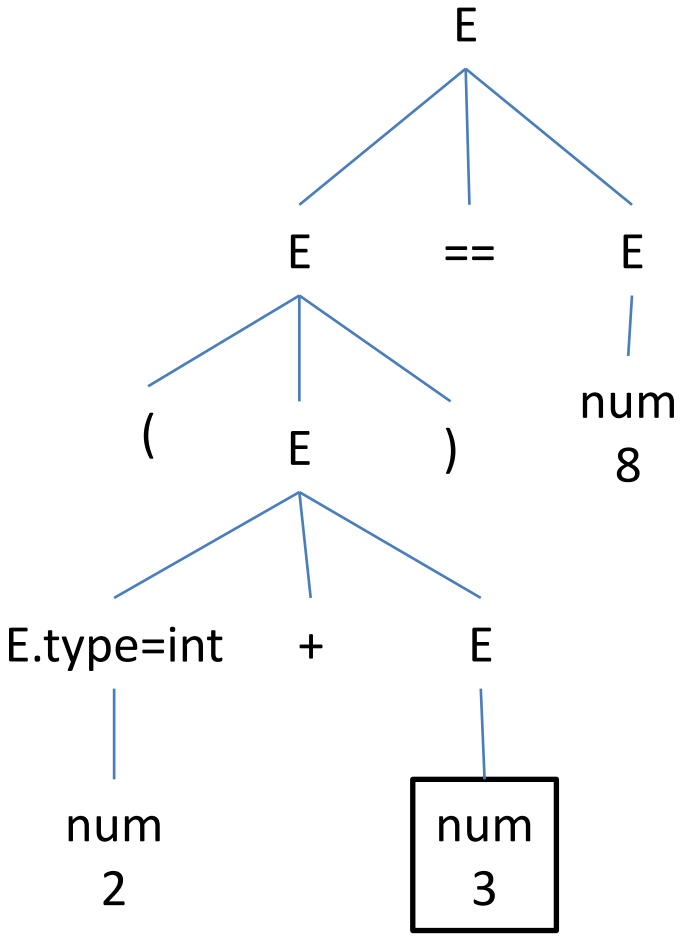
$E \rightarrow (E_1)$ {E.type = E_1.type}

$E \rightarrow$ num {E.type = int}

$E \rightarrow$ true {E.type = boolean}

$E \rightarrow$ false {E.type = boolean}

(2 + 3) == 8



Example 8
[S-attributed:- bottom-up left to right]

- SDT to check the type of an expression

$E \rightarrow E_1 + E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int))
then E.type = int else error;}

$E \rightarrow E_1 == E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int|boolean))
then E.type = boolean else error;}

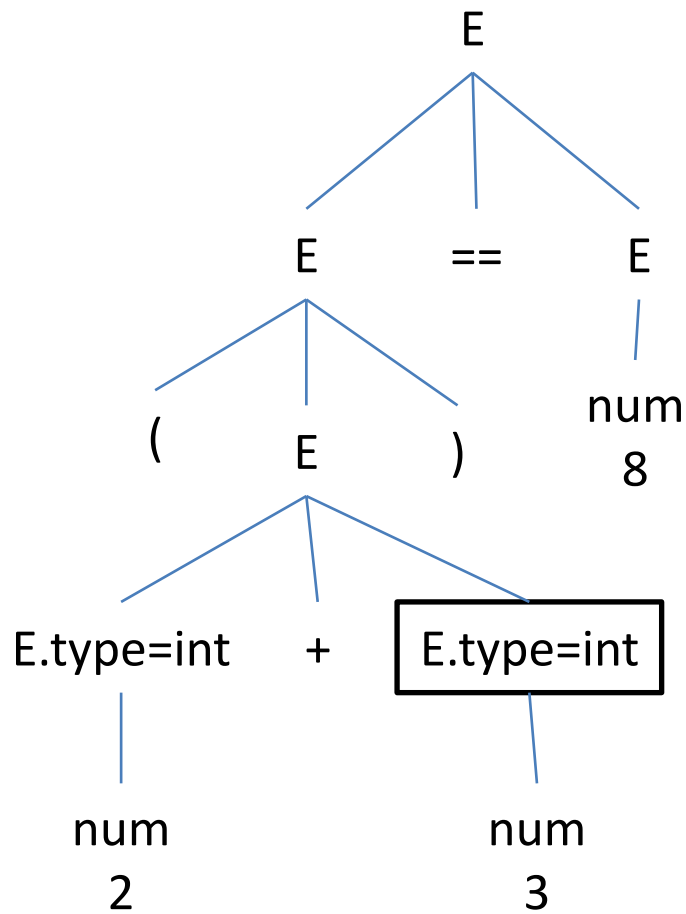
$E \rightarrow (E_1)$ {E.type = E_1.type}

$E \rightarrow$ num {E.type = int}

$E \rightarrow$ true {E.type = boolean}

$E \rightarrow$ false {E.type = boolean}

(2 + 3) == 8



Example 8
[S-attributed:- bottom-up left to right]

- SDT to check the type of an expression

$E \rightarrow E_1 + E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int))
then E.type = int else error;}

$E \rightarrow E_1 == E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int|boolean))
then E.type = boolean else error;}

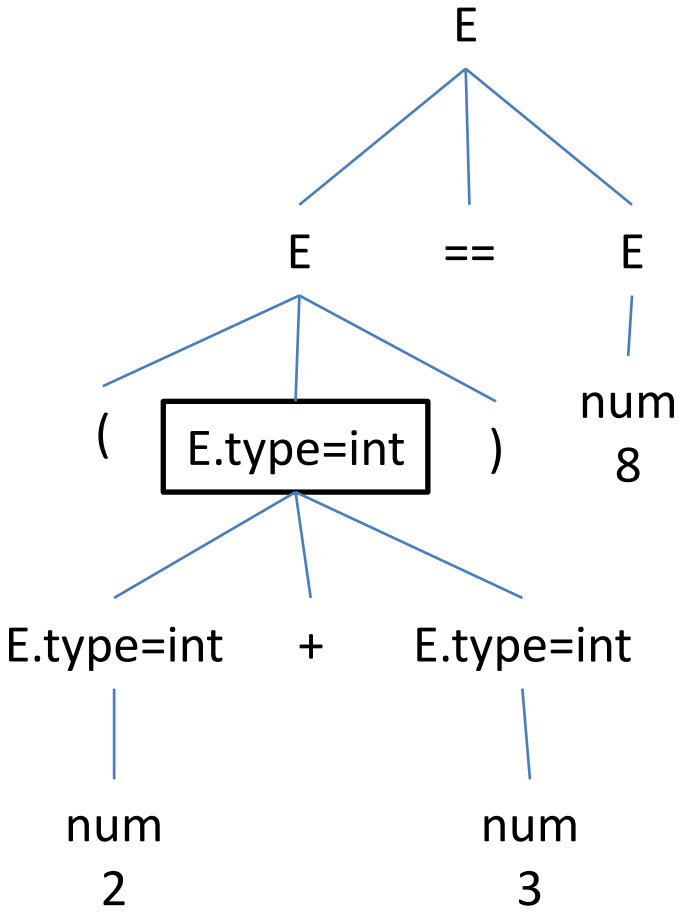
$E \rightarrow (E_1)$ {E.type = E_1.type}

$E \rightarrow$ num {E.type = int}

$E \rightarrow$ true {E.type = boolean}

$E \rightarrow$ false {E.type = boolean}

(2 + 3) == 8



Example 8
[S-attributed:- bottom-up left to right]

- SDT to check the type of an expression

$E \rightarrow E_1 + E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int))
then E.type = int else error;}

$E \rightarrow E_1 == E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int|boolean))
then E.type = boolean else error;}

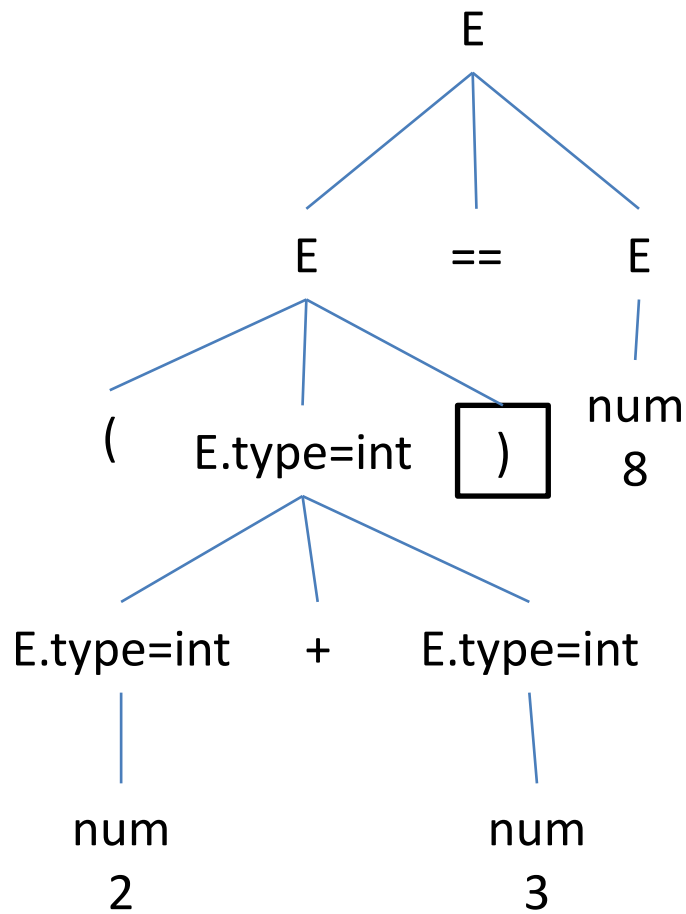
$E \rightarrow (E_1)$ {E.type = E_1.type}

$E \rightarrow$ num {E.type = int}

$E \rightarrow$ true {E.type = boolean}

$E \rightarrow$ false {E.type = boolean}

(2 + 3) == 8



Example 8
[S-attributed:- bottom-up left to right]

- SDT to check the type of an expression

$E \rightarrow E_1 + E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int))
then E.type = int else error;}

$E \rightarrow E_1 == E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int|boolean))
then E.type = boolean else error;}

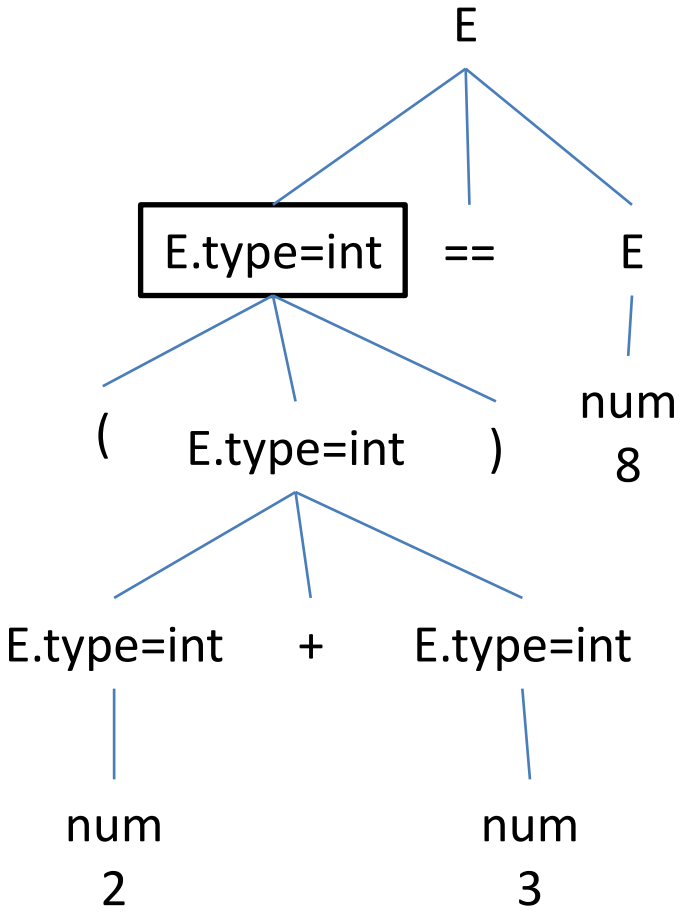
$E \rightarrow (E_1)$ {E.type = E_1.type}

$E \rightarrow$ num {E.type = int}

$E \rightarrow$ true {E.type = boolean}

$E \rightarrow$ false {E.type = boolean}

(2 + 3) == 8



Example 8
[S-attributed:- bottom-up left to right]

- SDT to check the type of an expression

$E \rightarrow E_1 + E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int))
then E.type = int else error;}

$E \rightarrow E_1 == E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int|boolean))
then E.type = boolean else error;}

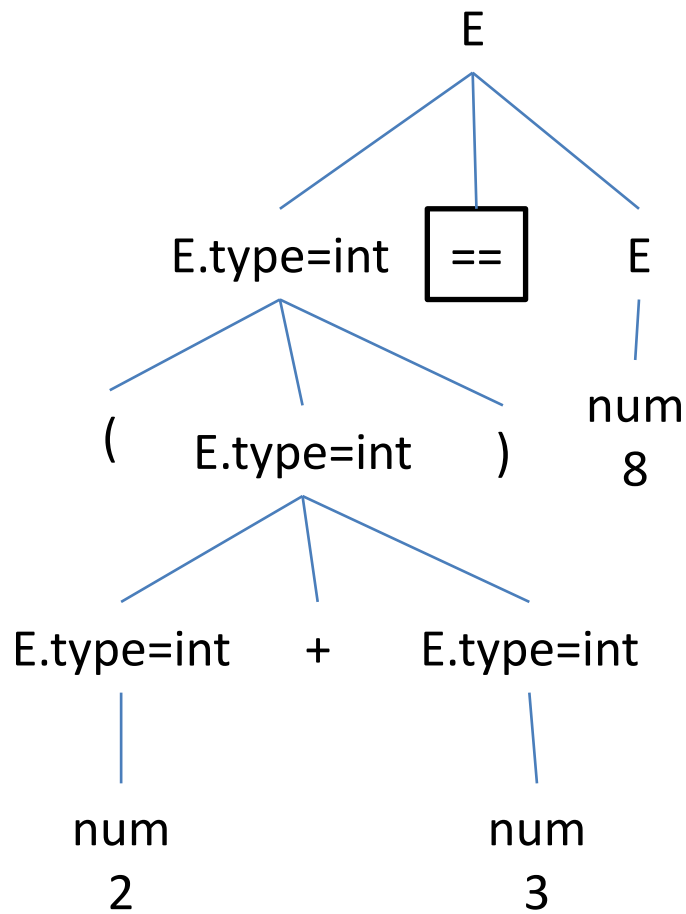
$E \rightarrow (E_1)$ {E.type = E_1.type}

$E \rightarrow$ num {E.type = int}

$E \rightarrow$ true {E.type = boolean}

$E \rightarrow$ false {E.type = boolean}

(2 + 3) == 8



Example 8
[S-attributed:- bottom-up left to right]

- SDT to check the type of an expression

$E \rightarrow E_1 + E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int))
then E.type = int else error;}

$E \rightarrow E_1 == E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int|boolean))
then E.type = boolean else error;}

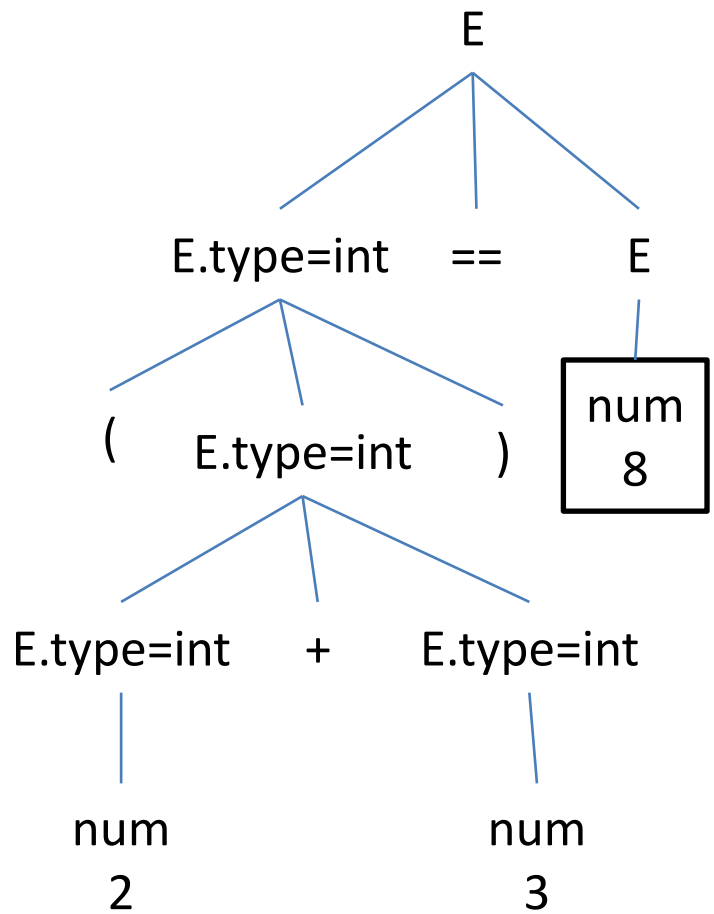
$E \rightarrow (E_1)$ {E.type = E_1.type}

$E \rightarrow$ num {E.type = int}

$E \rightarrow$ true {E.type = boolean}

$E \rightarrow$ false {E.type = boolean}

(2 + 3) == 8



Example 8
[S-attributed:- bottom-up left to right]

- SDT to check the type of an expression

$E \rightarrow E_1 + E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int))
then E.type = int else error;}

$E \rightarrow E_1 == E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int|boolean))
then E.type = boolean else error;}

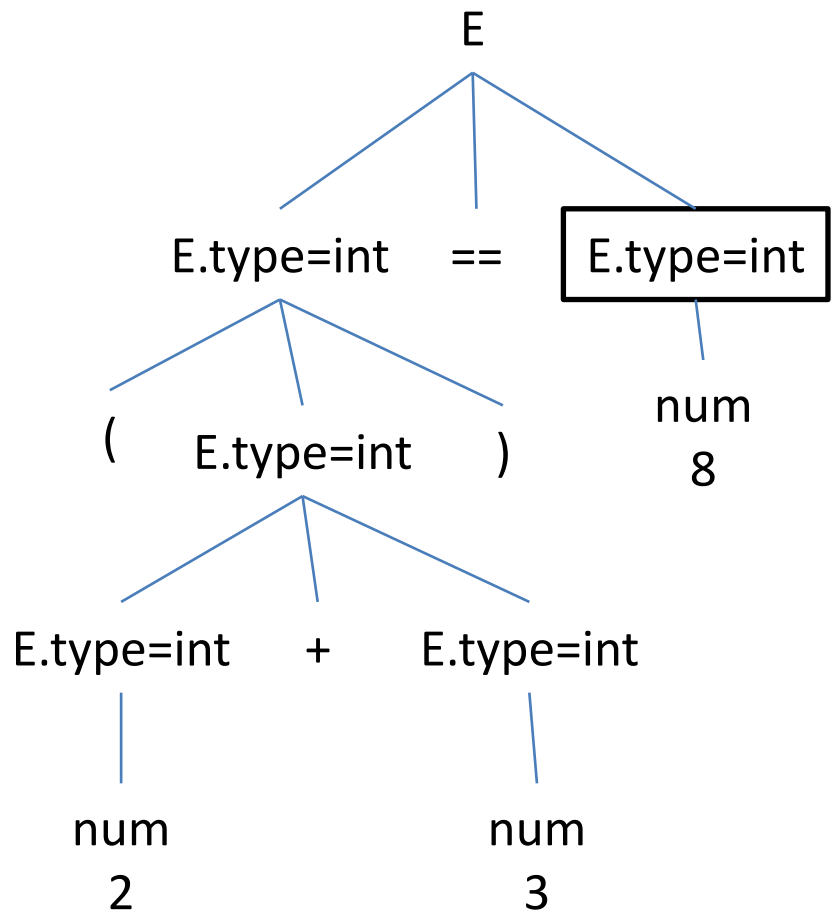
$E \rightarrow (E_1)$ {E.type = E_1.type}

E → num {E.type = int}

E → true {E.type = boolean}

E → false {E.type = boolean}

(2 + 3) == 8



Example 8
[S-attributed:- bottom-up left to right]

- SDT to check the type of an expression

$E \rightarrow E_1 + E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int))
then E.type = int else error;}

$E \rightarrow E_1 == E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int|boolean))
then E.type = boolean else error;}

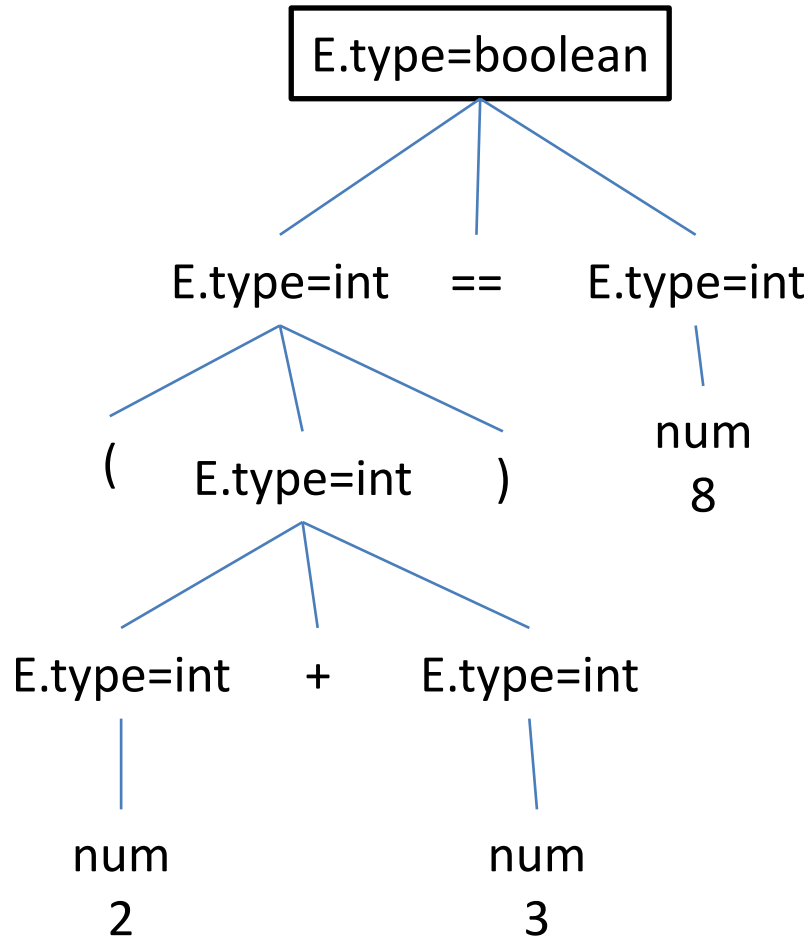
$E \rightarrow (E_1)$ {E.type = E_1.type}

$E \rightarrow$ num {E.type = int}

$E \rightarrow$ true {E.type = boolean}

$E \rightarrow$ false {E.type = boolean}

(2 + 3) == 8



Example 8
[S-attributed:- bottom-up left to right]

- SDT to check the type of an expression

$E \rightarrow E_1 + E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int))
then E.type = int else error;}

$E \rightarrow E_1 == E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int|boolean))
then E.type = boolean else error;}

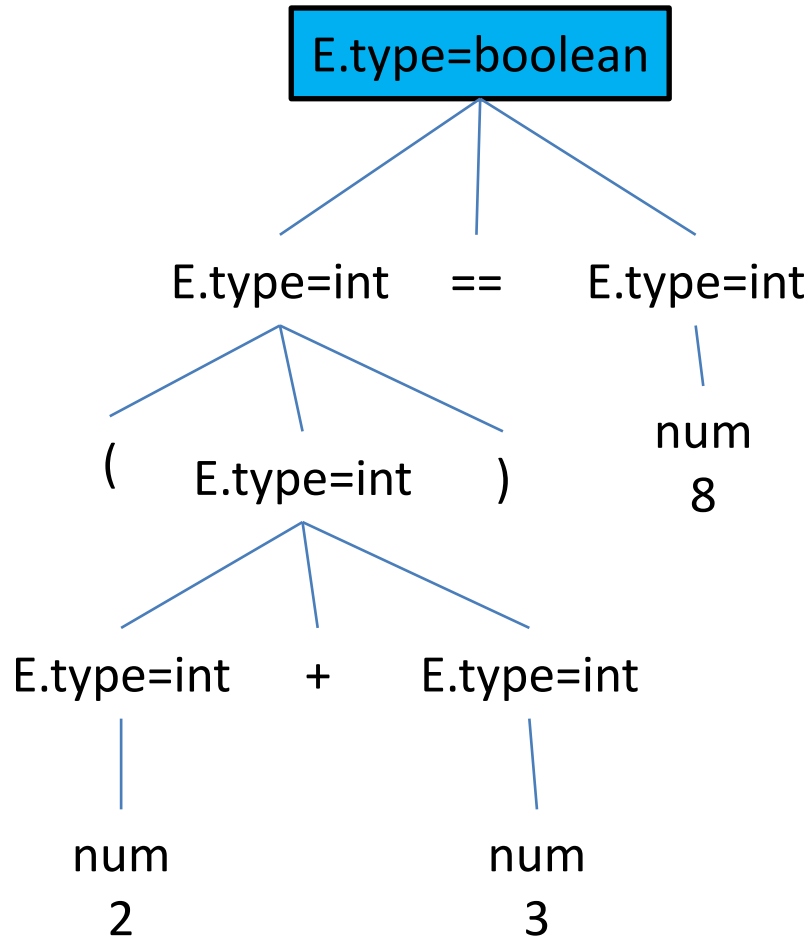
$E \rightarrow (E_1)$ {E.type = E_1.type}

E → num {E.type = int}

E → true {E.type = boolean}

E → false {E.type = boolean}

(2 + 3) == 8



Example 8

- SDT to check the type of an expression

$E \rightarrow E_1 + E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int))
then E.type = int else error;}

$E \rightarrow E_1 == E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int|boolean))
then E.type = boolean else error;}

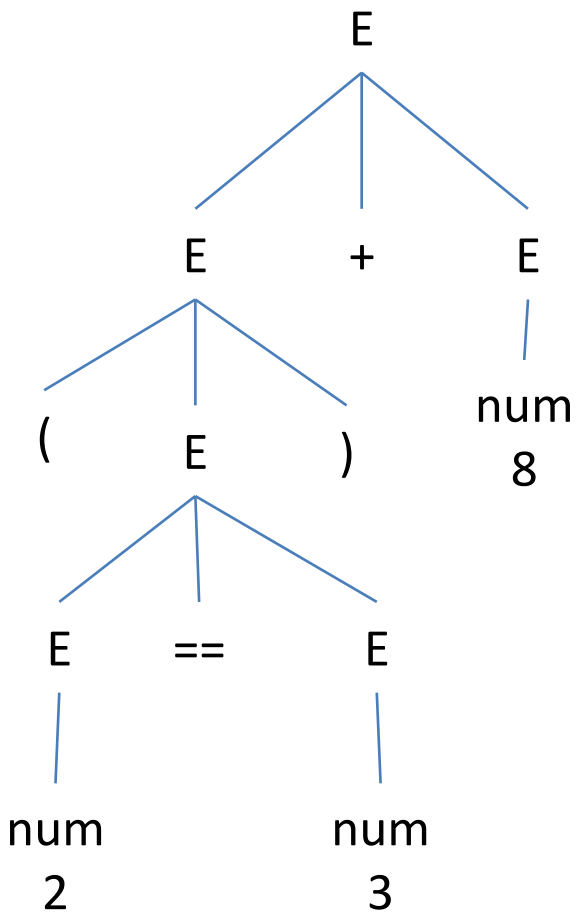
$E \rightarrow (E_1)$ {E.type = E_1.type}

$E \rightarrow$ num {E.type = int}

$E \rightarrow$ true {E.type = boolean}

$E \rightarrow$ false {E.type = boolean}

(2 == 3) + 8



Example 8

- SDT to check the type of an expression

$E \rightarrow E_1 + E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int))
then E.type = int else error;}

$E \rightarrow E_1 == E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int|boolean))
then E.type = boolean else error;}

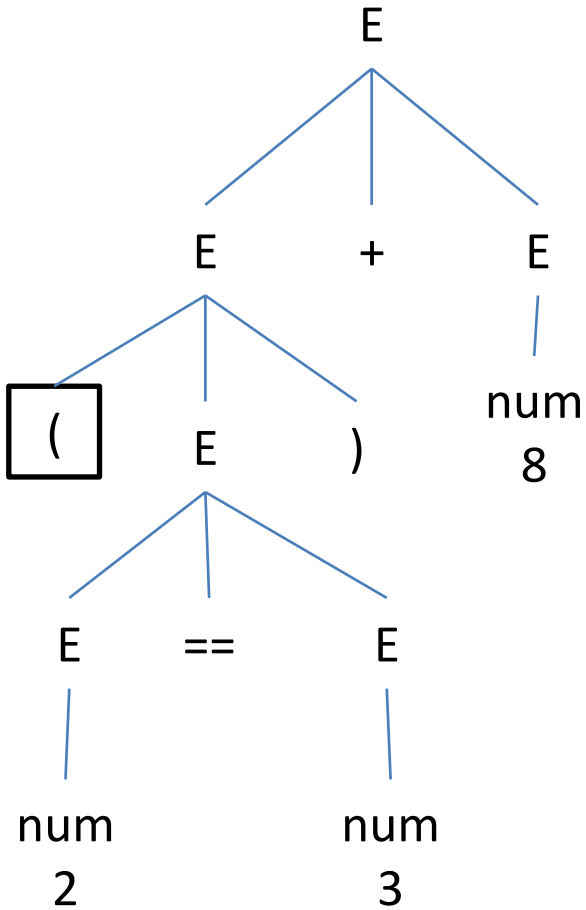
$E \rightarrow (E_1)$ {E.type = E_1.type}

$E \rightarrow$ num {E.type = int}

$E \rightarrow$ true {E.type = boolean}

$E \rightarrow$ false {E.type = boolean}

(2 == 3) + 8



Example 8

- SDT to check the type of an expression

$E \rightarrow E_1 + E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int))
then E.type = int else error;}

$E \rightarrow E_1 == E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int|boolean))
then E.type = boolean else error;}

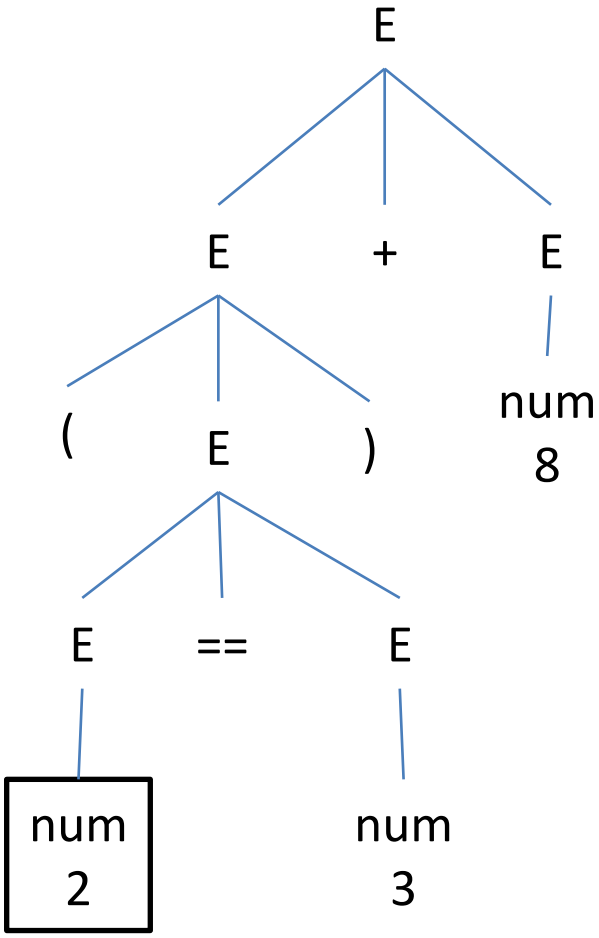
$E \rightarrow (E_1)$ {E.type = E_1.type}

$E \rightarrow$ num {E.type = int}

$E \rightarrow$ true {E.type = boolean}

$E \rightarrow$ false {E.type = boolean}

(2 == 3) + 8



Example 8

- SDT to check the type of an expression

$E \rightarrow E_1 + E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int))
then E.type = int else error;}

$E \rightarrow E_1 == E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int|boolean))
then E.type = boolean else error;}

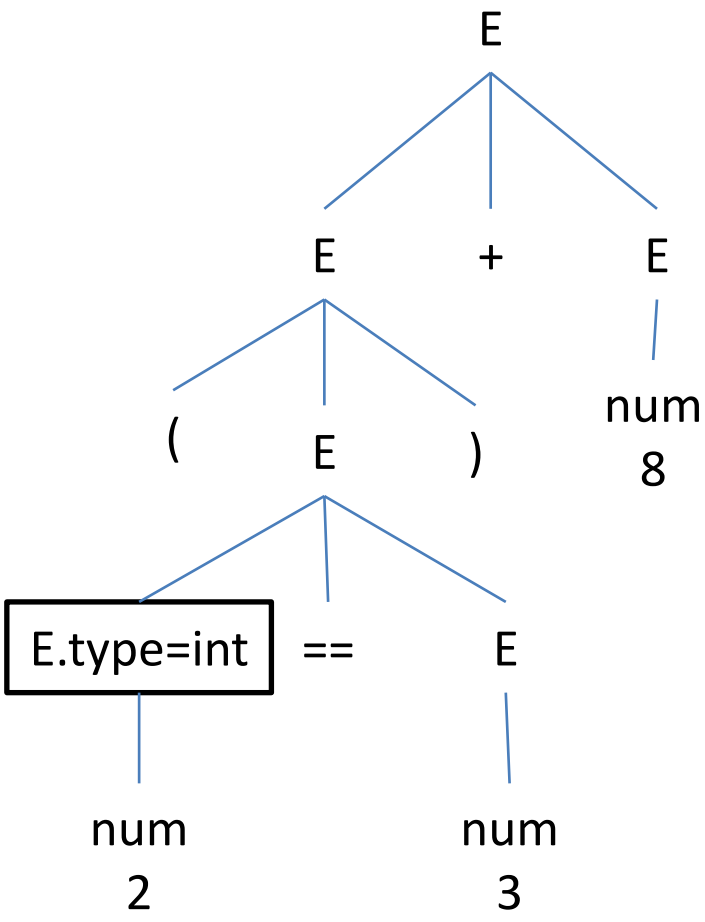
$E \rightarrow (E_1)$ {E.type = E_1.type}

$E \rightarrow$ num {E.type = int}

$E \rightarrow$ true {E.type = boolean}

$E \rightarrow$ false {E.type = boolean}

(2 == 3) + 8



Example 8

- SDT to check the type of an expression

$E \rightarrow E_1 + E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int))
then E.type = int else error;}

$E \rightarrow E_1 == E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int|boolean))
then E.type = boolean else error;}

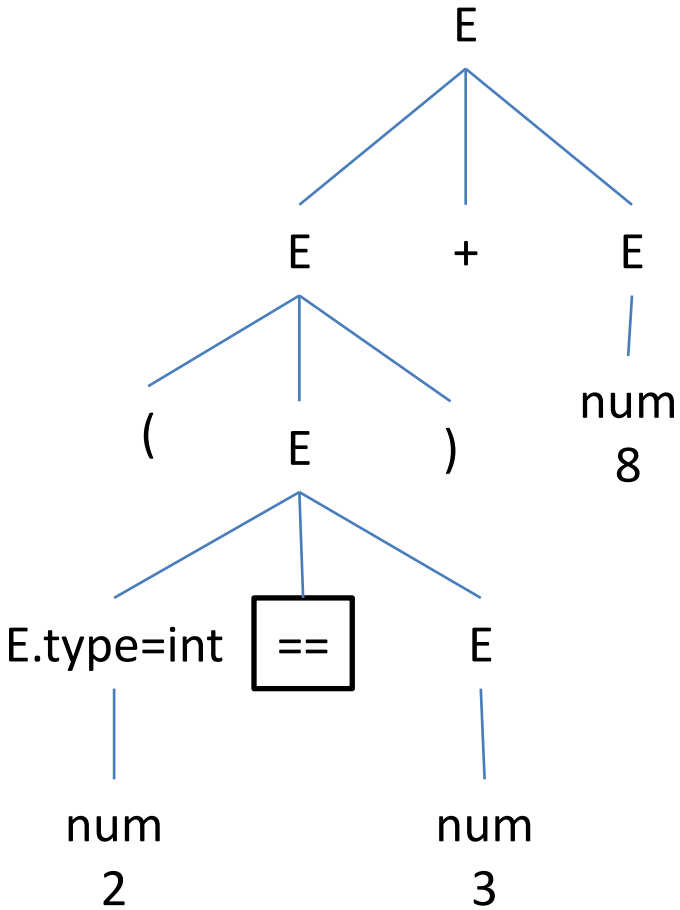
$E \rightarrow (E_1)$ {E.type = E_1.type}

$E \rightarrow$ num {E.type = int}

$E \rightarrow$ true {E.type = boolean}

$E \rightarrow$ false {E.type = boolean}

(2 == 3) + 8



Example 8

- SDT to check the type of an expression

$E \rightarrow E_1 + E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int))
then E.type = int else error;}

$E \rightarrow E_1 == E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int|boolean))
then E.type = boolean else error;}

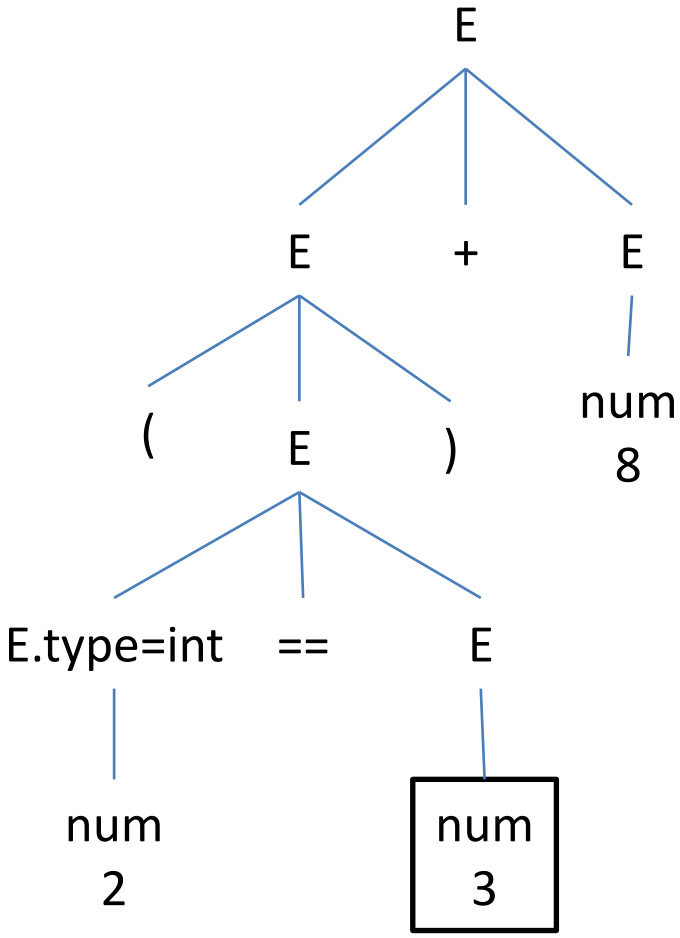
$E \rightarrow (E_1)$ {E.type = E_1.type}

$E \rightarrow$ num {E.type = int}

$E \rightarrow$ true {E.type = boolean}

$E \rightarrow$ false {E.type = boolean}

(2 == 3) + 8



Example 8

- SDT to check the type of an expression

$E \rightarrow E_1 + E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int))
then E.type = int else error;}

$E \rightarrow E_1 == E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int|boolean))
then E.type = boolean else error;}

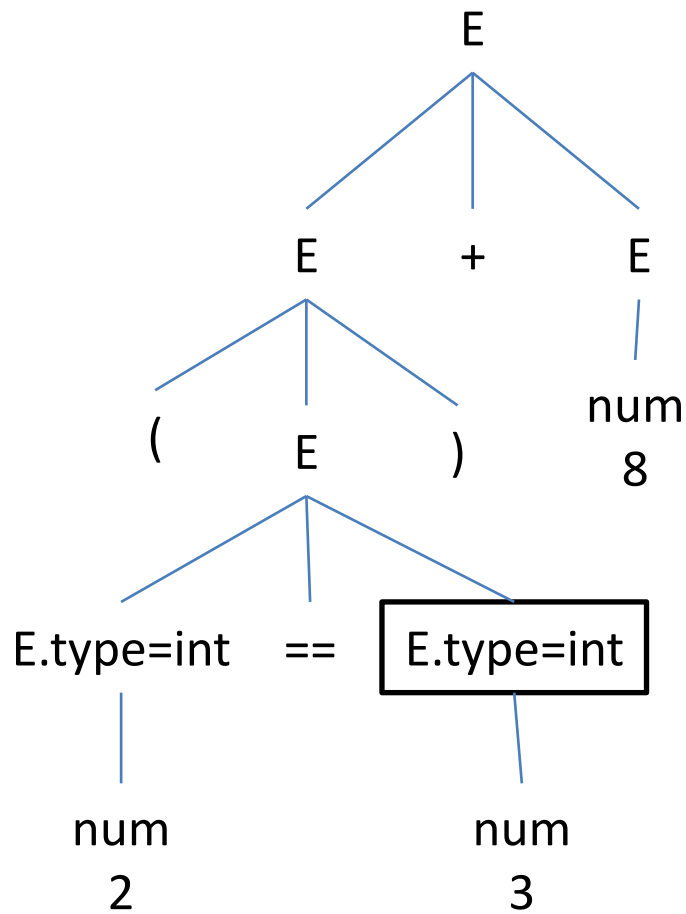
$E \rightarrow (E_1)$ {E.type = E_1.type}

$E \rightarrow$ num {E.type = int}

$E \rightarrow$ true {E.type = boolean}

$E \rightarrow$ false {E.type = boolean}

(2 == 3) + 8



Example 8

- SDT to check the type of an expression

$E \rightarrow E_1 + E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int))
then E.type = int else error;}

$E \rightarrow E_1 == E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int|boolean))
then E.type = boolean else error;}

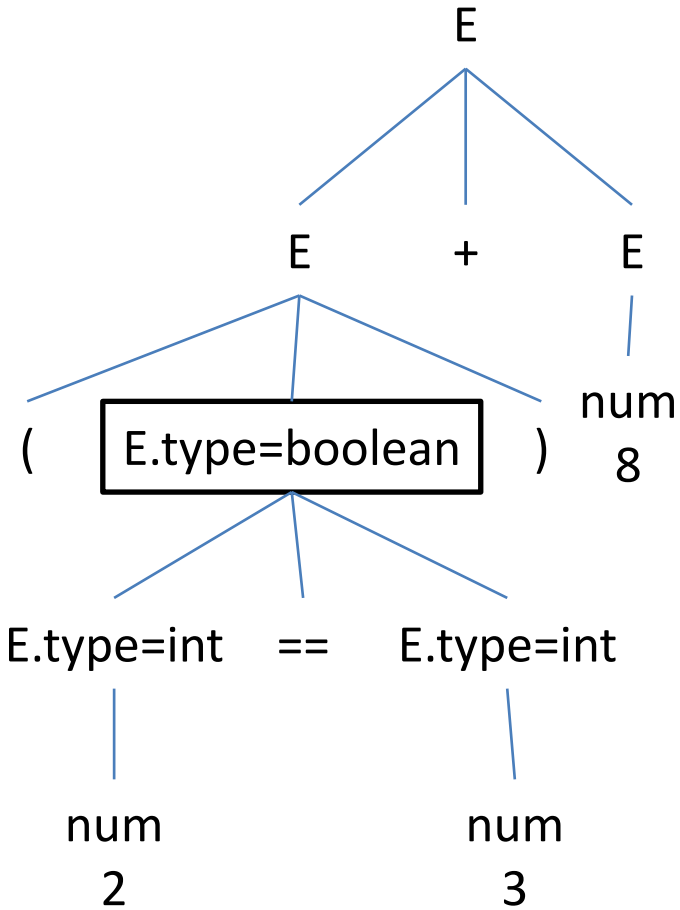
$E \rightarrow (E_1)$ {E.type = E_1.type}

$E \rightarrow$ num {E.type = int}

$E \rightarrow$ true {E.type = boolean}

$E \rightarrow$ false {E.type = boolean}

(2 == 3) + 8



Example 8

- SDT to check the type of an expression

$E \rightarrow E_1 + E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int))
then E.type = int else error;}

$E \rightarrow E_1 == E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int|boolean))
then E.type = boolean else error;}

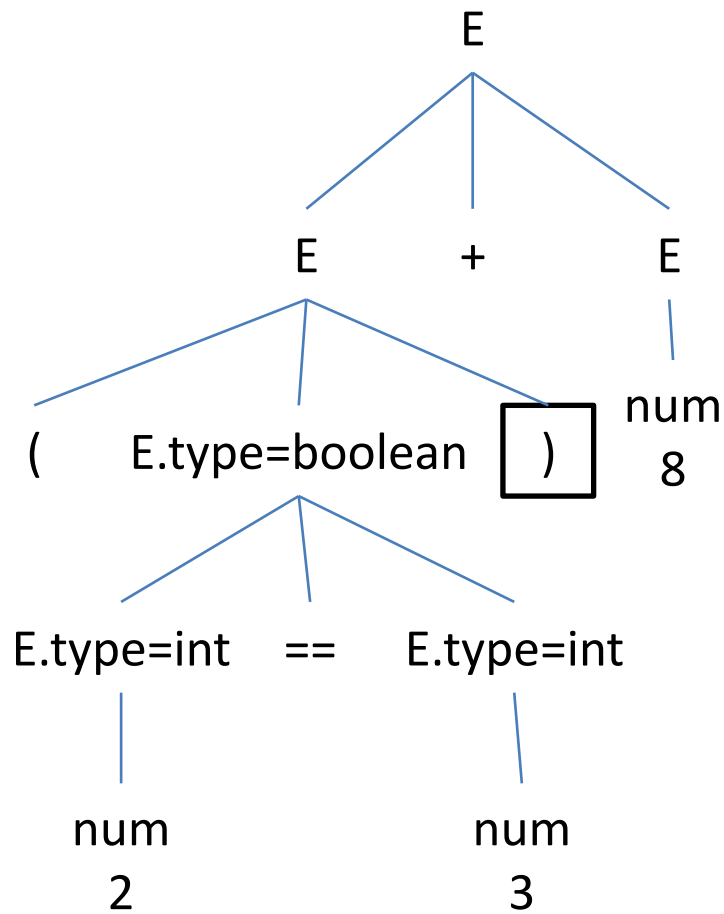
$E \rightarrow (E_1)$ {E.type = E_1.type}

$E \rightarrow$ num {E.type = int}

$E \rightarrow$ true {E.type = boolean}

$E \rightarrow$ false {E.type = boolean}

(2 == 3) + 8



Example 8

- SDT to check the type of an expression

$E \rightarrow E_1 + E_2$ {if((E_1.type == E_2.type) && (E_1.type == int)) then E.type = int else error;}

$E \rightarrow E_1 == E_2$ {if((E_1.type == E_2.type) && (E_1.type == int|boolean)) then E.type = boolean else error;}

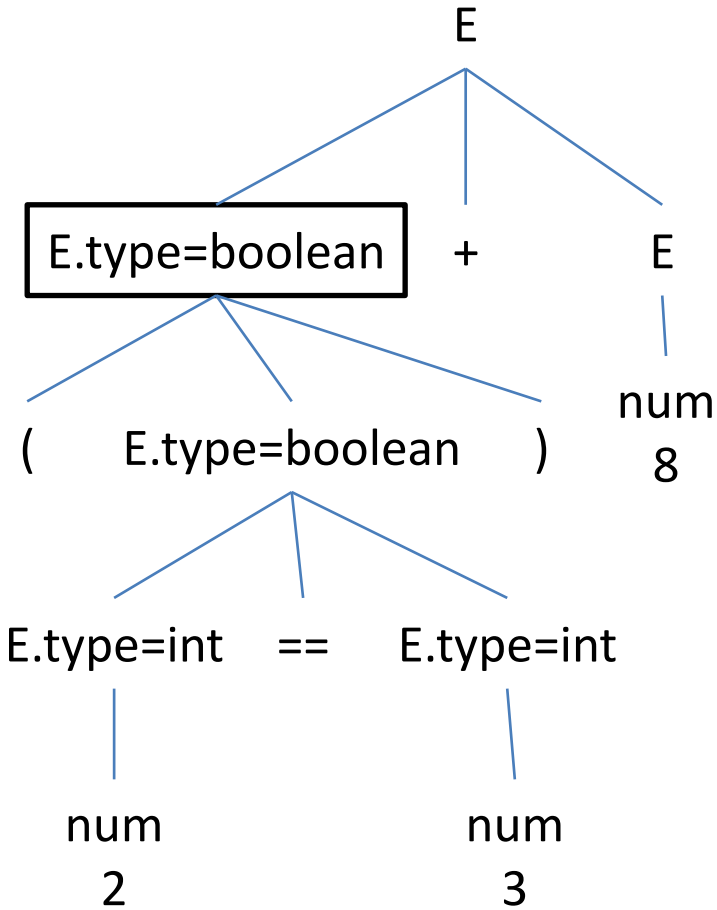
$E \rightarrow (E_1)$ {E.type = E_1.type}

$E \rightarrow$ num {E.type = int}

$E \rightarrow$ true {E.type = boolean}

$E \rightarrow$ false {E.type = boolean}

(2 == 3) + 8



Example 8

- SDT to check the type of an expression

$E \rightarrow E_1 + E_2$ {if((E_1.type == E_2.type) && (E_1.type == int)) then E.type = int else error;}

$E \rightarrow E_1 == E_2$ {if((E_1.type == E_2.type) && (E_1.type == int|boolean)) then E.type = boolean else error;}

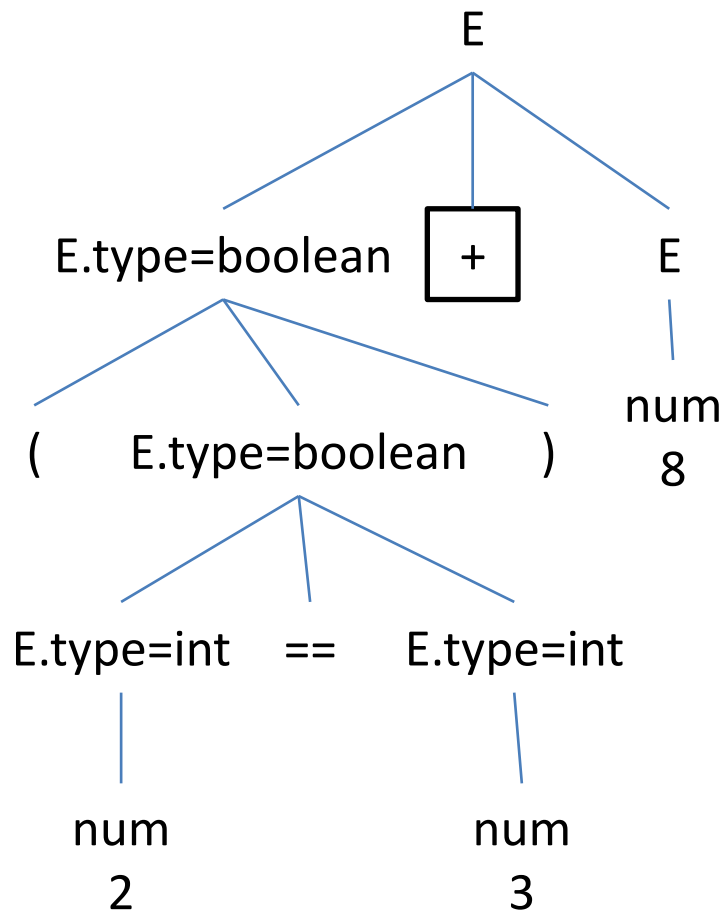
$E \rightarrow (E_1)$ {E.type = E_1.type}

$E \rightarrow$ num {E.type = int}

$E \rightarrow$ true {E.type = boolean}

$E \rightarrow$ false {E.type = boolean}

(2 == 3) + 8



Example 8

- SDT to check the type of an expression

$E \rightarrow E_1 + E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int))
then E.type = int else error;}

$E \rightarrow E_1 == E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int|boolean))
then E.type = boolean else error;}

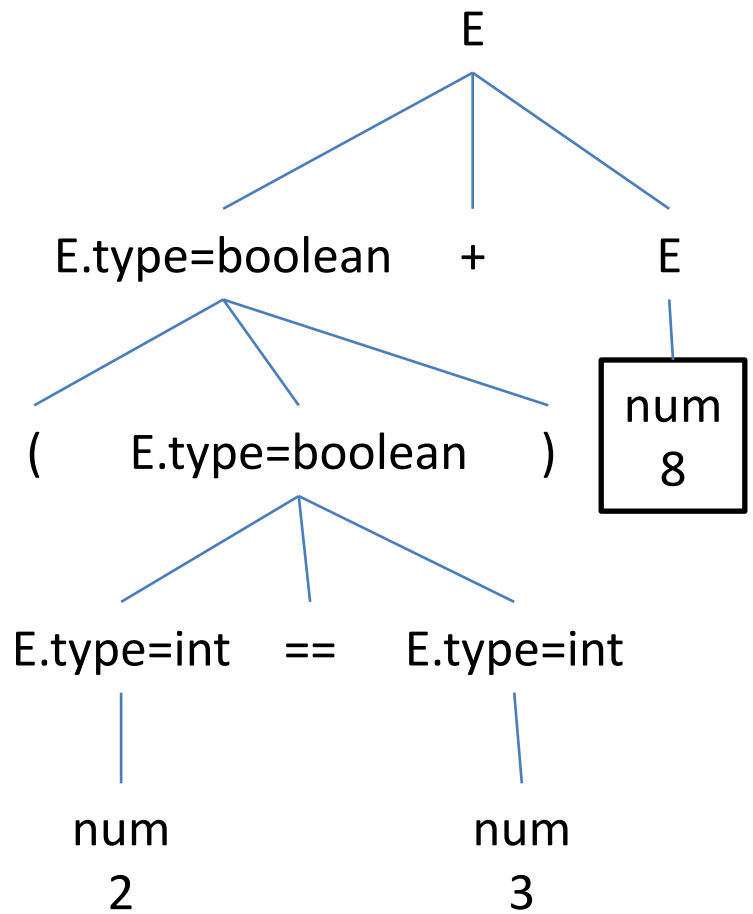
$E \rightarrow (E_1)$ {E.type = E_1.type}

$E \rightarrow$ num {E.type = int}

$E \rightarrow$ true {E.type = boolean}

$E \rightarrow$ false {E.type = boolean}

(2 == 3) + 8



Example 8

- SDT to check the type of an expression

$E \rightarrow E_1 + E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int))
then E.type = int else error;}

$E \rightarrow E_1 == E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int|boolean))
then E.type = boolean else error;}

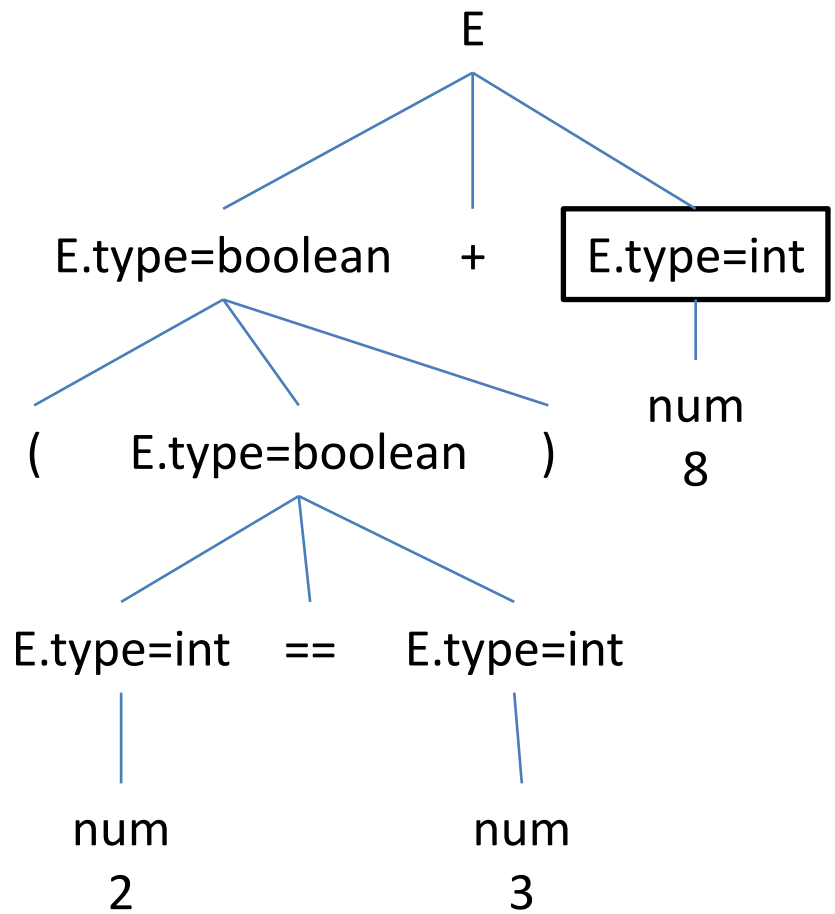
$E \rightarrow (E_1)$ {E.type = E_1.type}

$E \rightarrow$ num {E.type = int}

$E \rightarrow$ true {E.type = boolean}

$E \rightarrow$ false {E.type = boolean}

(2 == 3) + 8



Example 8

- SDT to check the type of an expression

$E \rightarrow E_1 + E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int))
then E.type = int else error;}

$E \rightarrow E_1 == E_2$ {if((E_1.type == E_2.type) &&
(E_1.type == int|boolean))
then E.type = boolean else error;}

$E \rightarrow (E_1)$ {E.type = E_1.type}

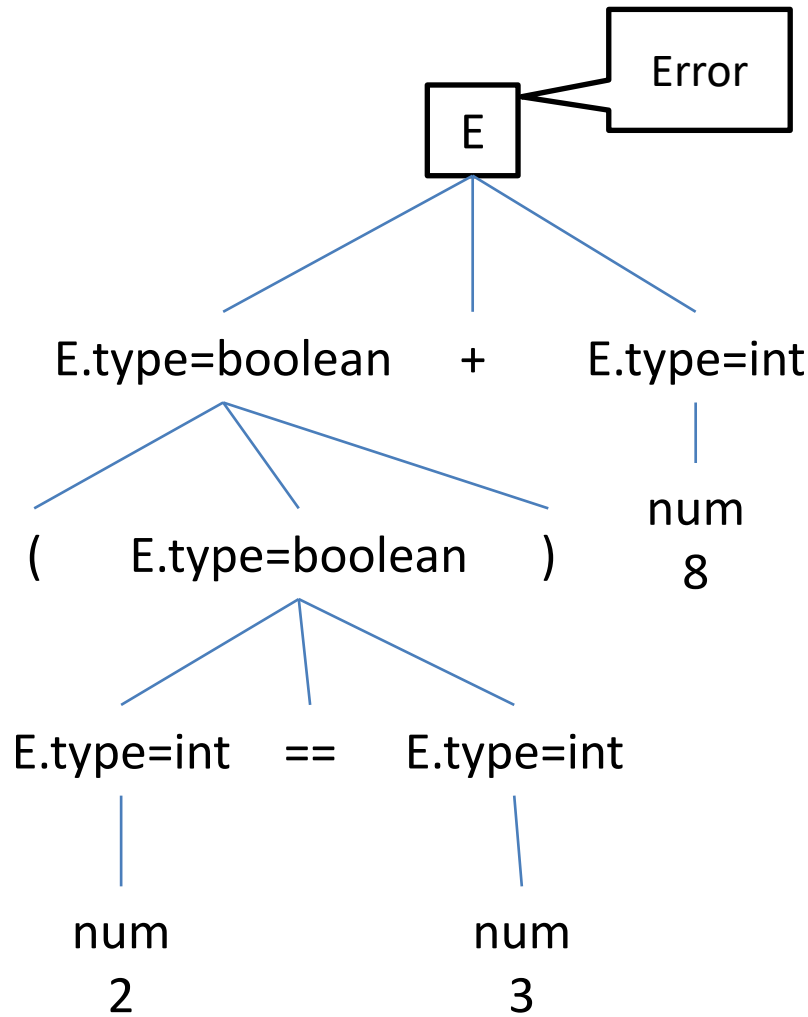
$E \rightarrow$ num {E.type = int}

$E \rightarrow$ true {E.type = boolean}

$E \rightarrow$ false {E.type = boolean}

(2 == 3) + 8

Error



S-attributed vs. L-attributed

1. S-attributed uses only synthesized attributes.
2. Semantic actions are placed at the right end of the productions.
3. Attributes are evaluated during bottom-up parsing.

1. L-attributed uses both inherited and synthesized attributes.
   - Inheritance is either from parent or left sibling.
2. Semantic action can be placed anywhere.
3. Attributes are evaluated by traversing the parse tree depth first, left to right.

Example

A→LM{L.i = f(A.i); M.i = f(L.s); A.s = f(M.s);}

A→QR {R.i = f(A.i); Q.i = f(R.i); A.s = f(Q.s);}

What type of SDT is this?

S-attributed or L-attributed?

Example

A→LM {L.i = f(A.i);	L's attribute is inherited from left side A
M.i = f(L.s);	M's attribute is inherited from left side L
A.s = f(M.s);}	A's attribute is synthesized from child M
A→QR {R.i = f(A.i);	R's attribute is inherited from left side A
Q.i = f(R.i);	Q's attribute is inherited from right side R
A.s = f(Q.s);}	A's attribute is synthesized from child Q

Example

A→LM {L.i = f(A.i);	L's attribute is inherited from left side A
M.i = f(L.s);	M's attribute is inherited from left side L
A.s = f(M.s);}	A's attribute is synthesized from child M

A→QR {R.i = f(A.i);	R's attribute is inherited from left side A
Q.i = f(R.i);	Q's attribute is inherited from right side R
A.s = f(Q.s);}	A's attribute is synthesized from child Q

Is it S-attributed? No because there are some inherited attributes.

Example

A→LM {L.i = f(A.i);	L's attribute is inherited from left side A
M.i = f(L.s);	M's attribute is inherited from left side L
A.s = f(M.s);}	A's attribute is synthesized from child M

A→QR {R.i = f(A.i);	R's attribute is inherited from left side A
Q.i = f(R.i);	Q's attribute is inherited from right side R
A.s = f(Q.s);}	A's attribute is synthesized from child Q

Is it S-attributed? No because there are some inherited attributes.

Is it L-attributed? No because there is inheritance from right side.