Syntax Directed Translation and Intermediate Code Generation

Notations for associating Semantic Rules

1. Syntax Directed Definition

- high level specifications.
- Hides implementation details.
- Does not specify explicit order of evaluation of semantic rules.

2. Syntax Directed Translation Scheme

- Specifies the order in which semantic rules are to be evaluated.
- Implementation oriented.

Syntax Directed Definition(SDD)

Syntax Directed Definition (SDD)

Associating rules with the grammar symbols of CFG, gives rise to SDD.

CFG + Semantic rules = SDD

Productions	Semantic rules
L -> En	{L.val = E.val n}
E -> E_1 + T	{E.val = E_1 .val + T.val; }
E -> T	{E.val = T.val;}
T -> T_1* F	{T.val = T_1. val $*$ F. val; }
T -> F	{T.val = F.val ; }
F -> digit	{F.val = digit.lexval; }

Fig: Syntax Directed Definition(SDD)

Attributes

An attribute is a value associated with the grammar symbols.

E -> E_1+ T **{E.val = E_1.val + T.val; }**

Attributes

Attribute grammar

- An attribute grammar is a context-free grammar augmented with attributes, semantic rules, and conditions.

- Types of attributes

  a) Synthesized Attributes

  b) Inherited Attributes

PRODUCTION	SEMANTIC RULES
$D \rightarrow T\ L$	$L.\mathrm{in} := T.\mathrm{type}$
$T \rightarrow$ **int**	$T.\mathrm{type} :=$ integer
$T \rightarrow$ **real**	$T.\mathrm{type} :=$ real
$L \rightarrow L_1$, **id**	$L_1.\mathrm{in} := L.\mathrm{in}$
	addtype(**id**.entry, L.in)
$L \rightarrow$ **id**	addtype(**id**.entry, L.in)

Fig: Attribute grammar

a) Synthesized attributes

A synthesized attribute at node N is defined only in terms of attribute values at the children of N and at N itself

Ex: **P -> QRS**

P.s = f(Q.s, R.s, S.s)

b) Inherited attributes

An inherited attribute at node N is defined only in terms of attribute values at N's parent, N itself, N's siblings.

Ex: **P -> QRS**

R.i = P.i

R.i = Q.i

In a ***SYNTAX-DIRECTED DEFINITION*** each grammar symbol *X* is associated with two finite sets of values:

- the *synthesized attributes* of *X* and
- the *inherited attributes* of *X*,

each production $A \rightarrow \alpha$ is associated with a finite set of expressions of the form

$$b = f(c_1, c_2 \ldots c_k)$$

called *semantic rules* where *f* is a function and

- either *b* is a synthesized attribute of *A* and the values $c_1, c_2 \ldots c_k$ are attributes of the grammar symbols of α or A
- or *b* is an inherited attribute of a grammar symbol of α and the values $c_1, c_2 \ldots c_k$ are attributes of the grammar symbols of α or A
- terminal symbol has no inherited attributes.

Classes/Types of SDD

1) S-attributed SDD: An SDD is S-attributed if every attribute is synthesized.

E -> $\mathbf{E1}$+ T {E.val = $\mathbf{E1}$.val + T.val; }

Synthesized Attributes

2) ***L-attributed SDD***: An SDD is L-attributed if it contains synthesized and inherited attributes.

Inherited attributes in L-attributed SDD have some restrictions and must obey the following rules.

Say, ***A*** **->** $\boldsymbol{X1}, \boldsymbol{X2}, \boldsymbol{X3}, \ldots \boldsymbol{Xi{-}1}\ \boldsymbol{Xi}\ \boldsymbol{Xi{+}1} \ldots \boldsymbol{Xn}$ then the rule may use only:

a*) Inherited attributes associated with the head A*.

$\boldsymbol{Xi}$. inh = $\boldsymbol{A}$. inh (Xi . inh can inhert from its parent provided parent's attribute is inherited.)

b) Either inherited or synthesized attributes associated with the occurrences of symbols $X1, X2, X3, \ldots Xi-1$ located to the left of Xi.(Xi. inh can inherit from all its left siblings and the attribute can either be .syn or .inh)

$$Xi.\text{inh} = (X1, X2, X3, \ldots Xi-1).\text{inh}$$

OR

$$Xi.\text{inh} = (X1, X2, X3, \ldots Xi-1).\text{syn}$$

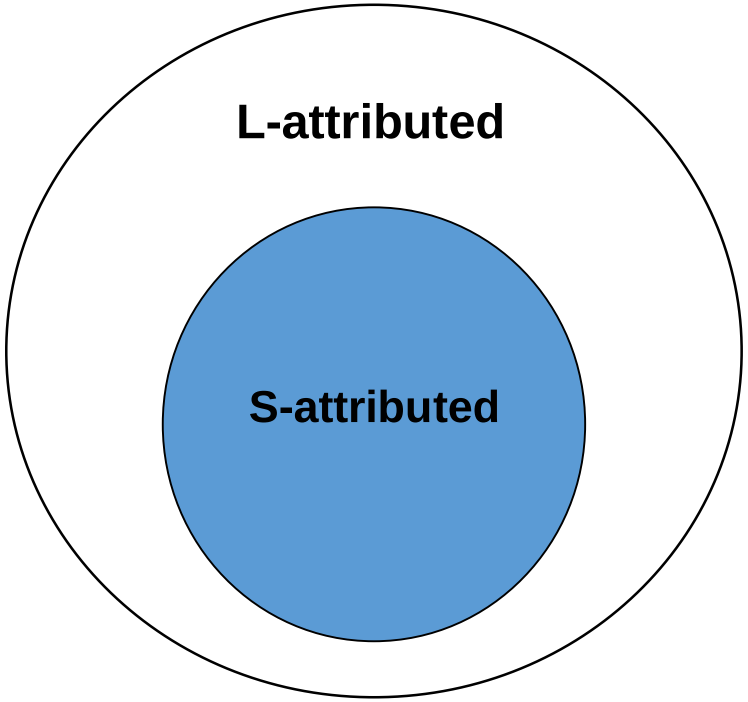
c) Inherited or synthesized attributes associated with the occurrences of Xi itself, but only in such a way that there are no cycles in a dependency graph formed by the attributes of this Xi. (i.e Xi.inh can inherit from itself and attribute can either be .inh or .syn)

$$Xi.\text{inh} = Xi.\text{inh}$$

OR

$$Xi.\text{inh} = Xi.\text{syn}$$

Every S-attributed SDD is L-attributed since L-attributed SDD contains both *synthesized* and *inherited attributes*.



Steps involved in Evaluating a SDD

Step 1: Construct the ***parse tree*** for the given input program.

Step 2: Construct the ***dependency graph*** for the parse tree.

Step 3: Perform a ***topological sort*** for the dependency graph.

Step 4: ***Annotate the parse tree*** by traversing the nodes in topologically sorted order, and evaluate attributes at each node.

Dependency graphs are useful tool for determining the evaluation order for the attributes instances in the parse tree.

Dependency graphs are directed graphs depicting the dependencies between attributes at various nodes in the parse tree.

Dependency graph depicts:

- The order in which the nodes of the SDD are evaluated in a given parse tree based on the semantic rules.
- The evaluation order defines the values of the attributes at the node.
- Terminals **do not have semantic rules** and their attribute values are supplied by **lexical analysis.**

Algorithm

```
for each node 'N' in the parse tree
        for each attribute 'a' of the grammar symbol at 'N'
                construct a node in the dependency graph for 'a'
for each node 'N' in the parse tree
        for each semantic rule b=f(c1,...,ck) associated with the production used at 'N'
                construct an edge from each ci to b
```

- If the dependency graph has an edge from node M to node N, then the attribute corresponding to M must be evaluated before the attribute of N.
- Thus, the only allowable orders of evaluation are those sequences of nodes $N1$, $N2$, …,Nk such that if there is an edge of the dependency graph from Ni to Nj, then i < j.
- Such an ordering embeds a directed graph into linear order, and is called topological sort of a graph.
- If there exists cycle then there is no topological sorts.
- There can be more than one topological sort order i.e. there exists minimum one or more than one topological sort.

Topological sort is the order in which the nodes of the graph as m_1, m_2, ..., m_n such that no edge goes from m_{i+k} to m_i for any i,k

Annotated parse tree

A parse tree depicting attribute values associated with the grammar symbols is called ***annotated parse tree***.

Note:

- ➢ *If the grammar does not have a successive non terminals, then it is called an infix grammar.*
- ➢ *Designing an SDD for such grammar is simple since it would have only synthesized attributes.*
- ➢ *Synthesized attributes are such attributes which*
    - *takes value from the* ***child node****.*
    - *takes value from* ***itself****.*
    - ***does not take value from its parent and siblings****.*

- ➢ *If the grammar has two or more successive non-terminals then it is called a* ***non-infix grammar.***
- ➢ *Designing an SDD for such grammar is complex as it involves synthesized as well as inherited attributes.*

- *Inherited attributes are such attributes which*
    - *accepts value from **itself***
    - *accept value from its **parent***
    - *accept values from its **left sibling***

Note: **NOT ALL NON-INFIX GRAMMARS ARE L-ATTRIBUTED**

S-Attributed Syntax Directed Definition (S-Attributed SDD)

- An SDD is S-attributed if every attribute is synthesized.
- S-attributed definitions can be implemented during bottom up parsing.

For the given grammar,

(a) design SDD

(b) Construct the annotated parse tree

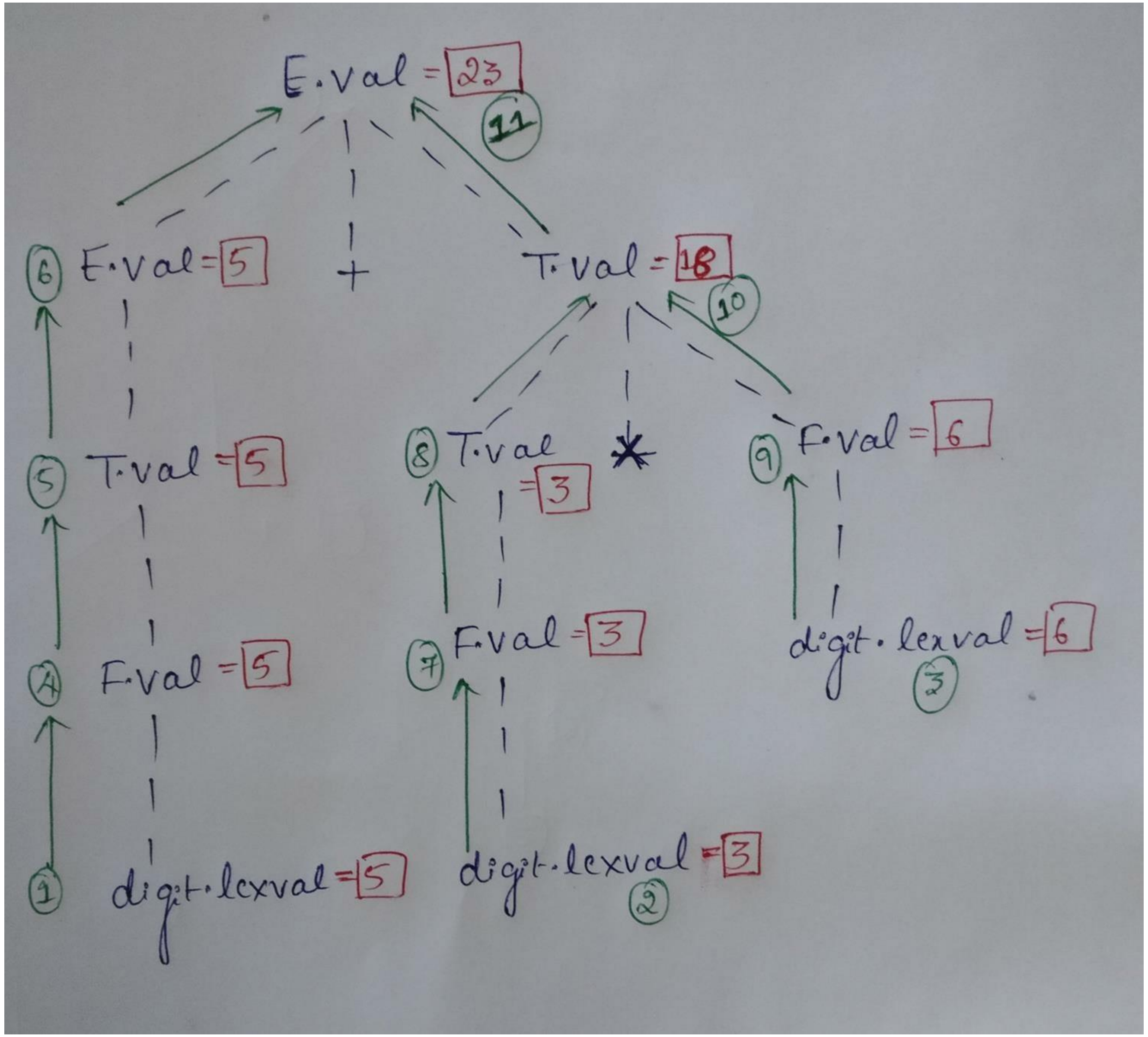
(c) Construct Dependency graph

for the string 5+3*6

$E \text{ -> } E_1 + T$
$E \text{ -> } T$
$T \text{ -> } T_1 * F$
$T \text{ -> } F$
$F \text{ -> } digit$

Productions	Semantic Rules
E -> E_1 + T	{E.val = E_1 .val + T.val; }
E -> T	{E.val = T.val;}
T -> T_1 * F	{T.val = T_1 .val $*$ F.val; }
T -> F	{T.val = F.val ; }
F -> digit	{F.val = digit.lexval; }

The above grammar is S-attributed.



Evaluation order(Topological order):
1 4 5 6 2 7 8 3 9 10 11

Dependency graph and Annotated parse tree for *5+3*6*

For the given grammar,

(a) design SDD

(b) Construct Dependency graph

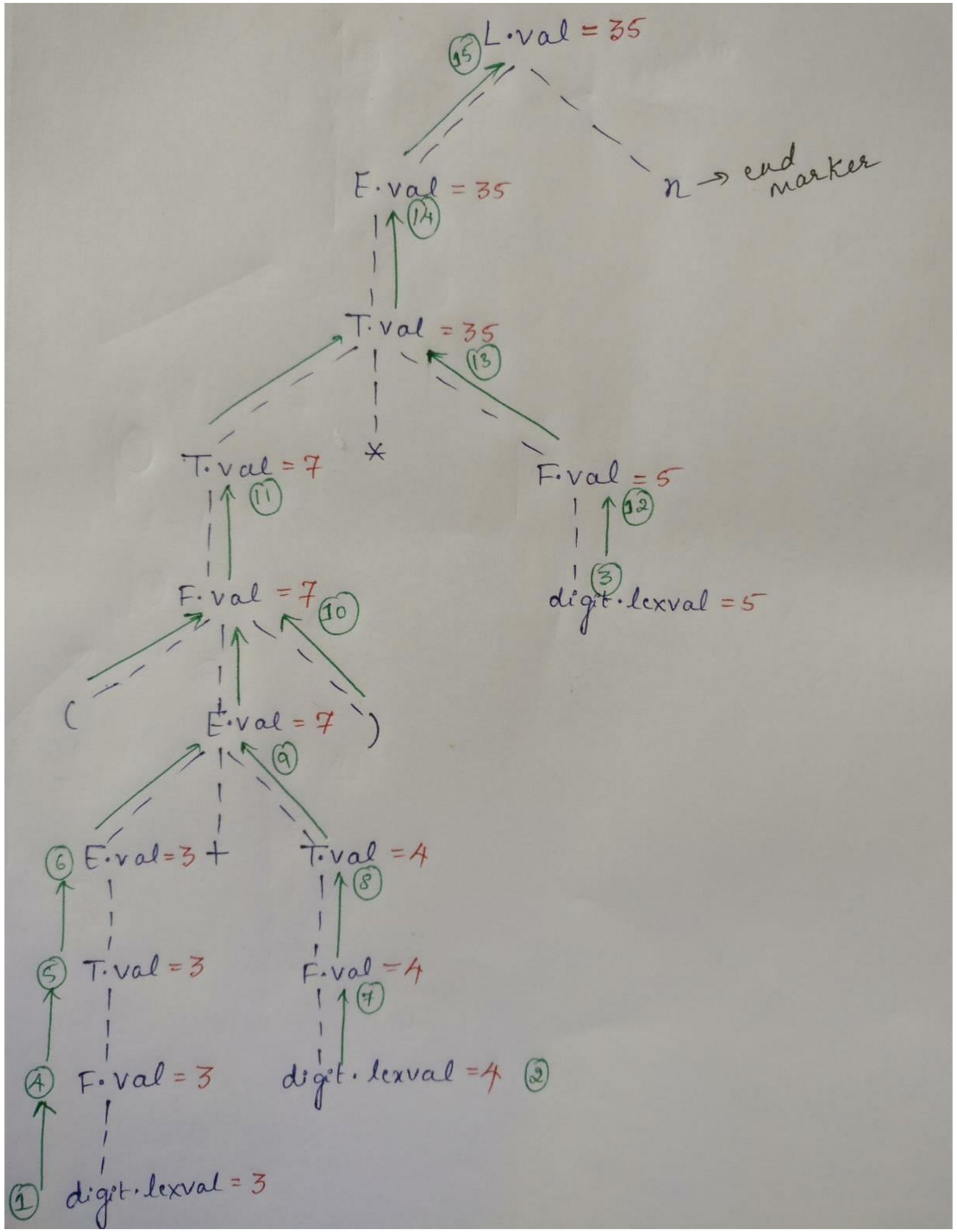
(c) Construct the annotated parse tree

for the string (3+4)*5n

L -> En
E -> E_1 *+ T*
E -> T
T -> T_1 ** F*
T -> F
F -> (E)
F -> digit

Productions	Semantic Rules
L -> En	{L.val = E.val;}
E -> E_1 + T	{E.val = E_1 .val + T.val; }
E -> T	{E.val = T.val;}
T -> T_1 * F	{T.val = $T_1 .val \ * \ F.val;$ }
T -> F	{T.val = F.val ; }
F -> (E)	{F.val = E.val;}
F -> digit	{F.val = digit.lexval; }

The above grammar is S-attributed.



Dependency graph and Annotated parse tree for *(3+4) * 5n*

Evaluation order(Topological order):
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

There exists one more order:
1 4 5 6 2 7 8 9 10 11 3 12 13 14 15

Note:

Number of reductions corresponds to total number of non-terminals in the parse tree.

For Example, in the previous annotated parse tree, the total number of reductions are 12, which is equal to the total number of non-terminals

Design:

1. SDD to count the number of 0’s in a binary number
2. SDD to count the number of 1’s in a binary number
3. SDD to count the number of bits in a Binary number
4. SDD to convert a Binary number to a Decimal number
5. SDD to convert a Binary Fraction to a Decimal number

Write SDD to count the number of 0's in a binary number

$S \rightarrow S_1 B$

$S \rightarrow B$

$B \rightarrow 0$

$B \rightarrow 1$

SDD to count the number of 0’s in a binary number

Productions	Semantic Rules
S -> S_1 B	{S.count = S_1 .count + B.count; }
S -> B	{S.count = B.count}
B -> 0	{B.count = 1; }
B -> 1	{B.count = 0; }

Write SDD to count the number of 1’s in a binary number

S -> S_1B

S -> B

B -> 0

B -> 1

SDD to count the number of 1’s in a binary number

Productions	Semantic Rules
S -> S_1 B	{S.count = S_1 .count + B.count; }
S -> B	{S.count = B.count}
B -> 0	{B.count = 0; }
B -> 1	{B.count = 1; }

Write SDD to count the number of bits in a Binary number

S -> $S_1 B$

S -> B

B -> 0

B -> 1

SDD to count the number of bits in a given binary number

Productions	Semantic Rules
S -> S_1 B	{S.count = S_1 .count + B.count; }
S -> B	{S.count = B.count}
B -> 0	{B.count = 1; }
B -> 1	{B.count = 1; }

Write SDD to convert a Binary number to a Decimal number

$S \rightarrow S_1 B$

$S \rightarrow B$

$B \rightarrow 0$

$B \rightarrow 1$

SDD to convert a Binary number to a Decimal number

Productions	Semantic Rules
S -> S_1 B	{S.val = S_1 .val * 2 + B.val; }
S -> B	{S.val = B.val}
B -> 0	{B.val = 0; }
B -> 1	{B.val = 1; }

Write SDD to convert a Binary Fraction to a Decimal number

D -> $S_1 . S_2$

S -> $S_1 B$

S -> B

B -> 0

B -> 1

SDD to convert a Binary Fraction to a Decimal number

Productions	Semantic Rules
D -> S_1 . S_2	D.val = S_1 .val + S_2 .val $/2^{S_2\text{.count}}$
S -> S_1 B	{S.val = S_1 .val * 2 + B.val; S.count = S_1 .count + B.count; }
S -> B	{S.val = B.val; S.count = B.count;}
B -> 0	{B.val = 0; B.count = 1;}
B -> 1	{B.val = 1; B.count = 1;}

Solve:

1) Given a CFG, design SDD to determine the type of each term and expression.

2) Given a CFG, design SDD to determine whether the arithmetic value of E is positive or negative for the expression ***5 * - 4***.

SDD to determine the type of each term and expression

Productions	Semantic Rules
E -> E_1 + T	{if(E_1. type == float) || (T.type == float) {E.type = float} else {E.type = int} ;}
E -> T	{E.type = T.type;}
T -> num . num	{T.type = float; }
T -> num	{T.type = int; }

SDD to determine whether the arithmetic value of E is positive or negative

Productions	Semantic Rules
R -> E	{if(E.sign == positive) R.sign = positive else R.sign = negative ;}
E -> num	{E.sign = positive;}
$\mathrm{E} \rightarrow +\mathrm{E}_1$	{E.sign = E_1 .sign; }
$\mathrm{E} \rightarrow -\mathrm{E}_1$	{ if(E_1.sign == negative) E.sign = positive else E.sign = negative ;}
$\mathrm{E} \rightarrow \mathrm{E}_1 * \mathrm{E}_2$	{ if(E_1.sign == E_2 .sign) E.sign = positive else E.sign = negative ;}

parse ***5 * - 4***

L-Attributed Syntax Directed Definition (L-Attributed SDD)

- An SDD is L-attributed attributed if it contains synthesized and inherited attributes.
- Top down parsers are most suitable for L-attributed SDDs.

Suppose that we have a production A -> BCD. Each of the four non-terminals A, B, C, and D have two attributes:

s - synthesized attribute, and
i - inherited attribute.

For each of the sets of rules below, on the basis of what is being computed tell whether,

(1) the rules are consistent with an S-attributed definition

(2) the rules are consistent with an L-attributed definition, and

(3) whether the rules are consistent with any evaluation order at all?

a) A.s = B.i + C.s

- **Violates rules of S-attributed definition.**
- **Rules are consistent with L-attributed definition.**
- **None of the rules related to L-attributed grammar have been violated and no cycle exists.**

b) A.s = B.i + C.s

D.i = A.i + B.s

- **Violates rules of S-attributed definition.**
- **Rules are consistent with L-attributed definition.**
- **None of the rules related to L-attributed SDD have been violated and no cycle exists.**

c) A.s = B.s + D.s

- **Rules are consistent with S-attributed definition.**
- **Rules are consistent with L-attributed definition as it contains synthesized attributes(Every S-attributed SDD is L-attributed).**
- **None of the rules related to S-attributed SDD or L-attributed SDD have been violated and no cycle exists.**

d) $A.s = D.i$

$B.i = A.s + C.s$

$C.i = B.s$

$D.i = B.i + C.i$

- **Violates rules of S-attributed definition.**
- **Violates rules of L-attributed definition.**
- **cycle exists.**

For the given grammar,

(a) design SDD

(b) Construct the annotated parse tree

(c) Construct Dependency graph for the string "6 * 2"

(d) Mention the topological sorting order

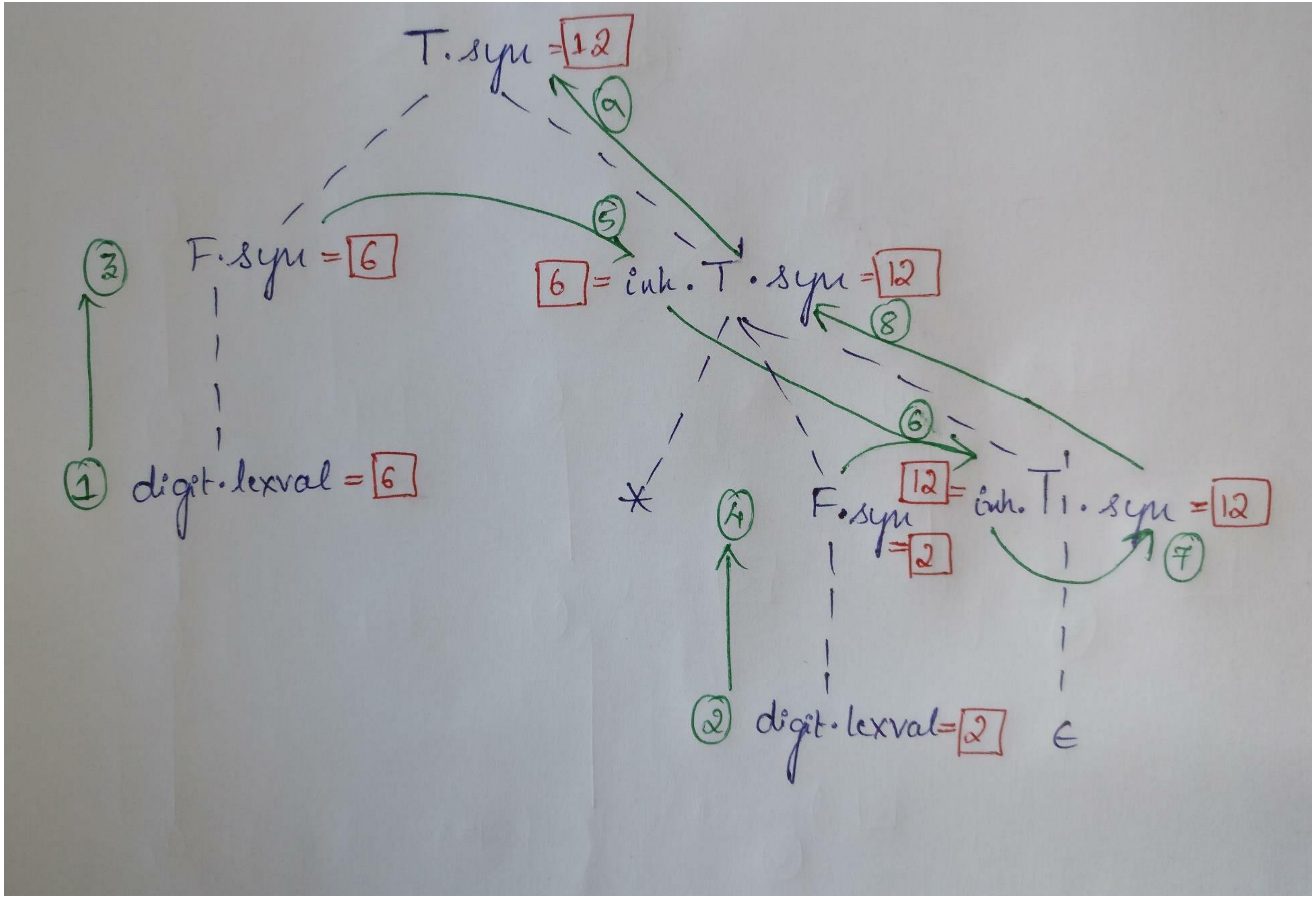
$T \to FT'$

$T' \to *FT'$

$T' \to \epsilon$

$F \to digit$

Productions	Semantic Rules
$T \rightarrow FT'$	$T'.in = F.syn;\ T.syn = T'.syn$
$T' \rightarrow *FT'_1$	$T'_1.inh = T'.inh * F.syn;\ \ T'.syn = T'_1.syn$
$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
$F \rightarrow digit$	$F.syn = digit.lexval$



Evaluation order(Topological order):
1,2,3,4,5,6,7,8,9

There exists one more order:
1,3,5,2,4,6,7,8,9

Dependency graph and Annotated parse tree for *(6*2)*

For the given grammar,

(a) design SDD.

(b) Construct the annotated parse tree

(c) Construct Dependency graph for the string "*a – 4 + c*"

(d) Mention the topological sorting order

E -> TE’

E’ -> +TE’

E’ -> -TE’

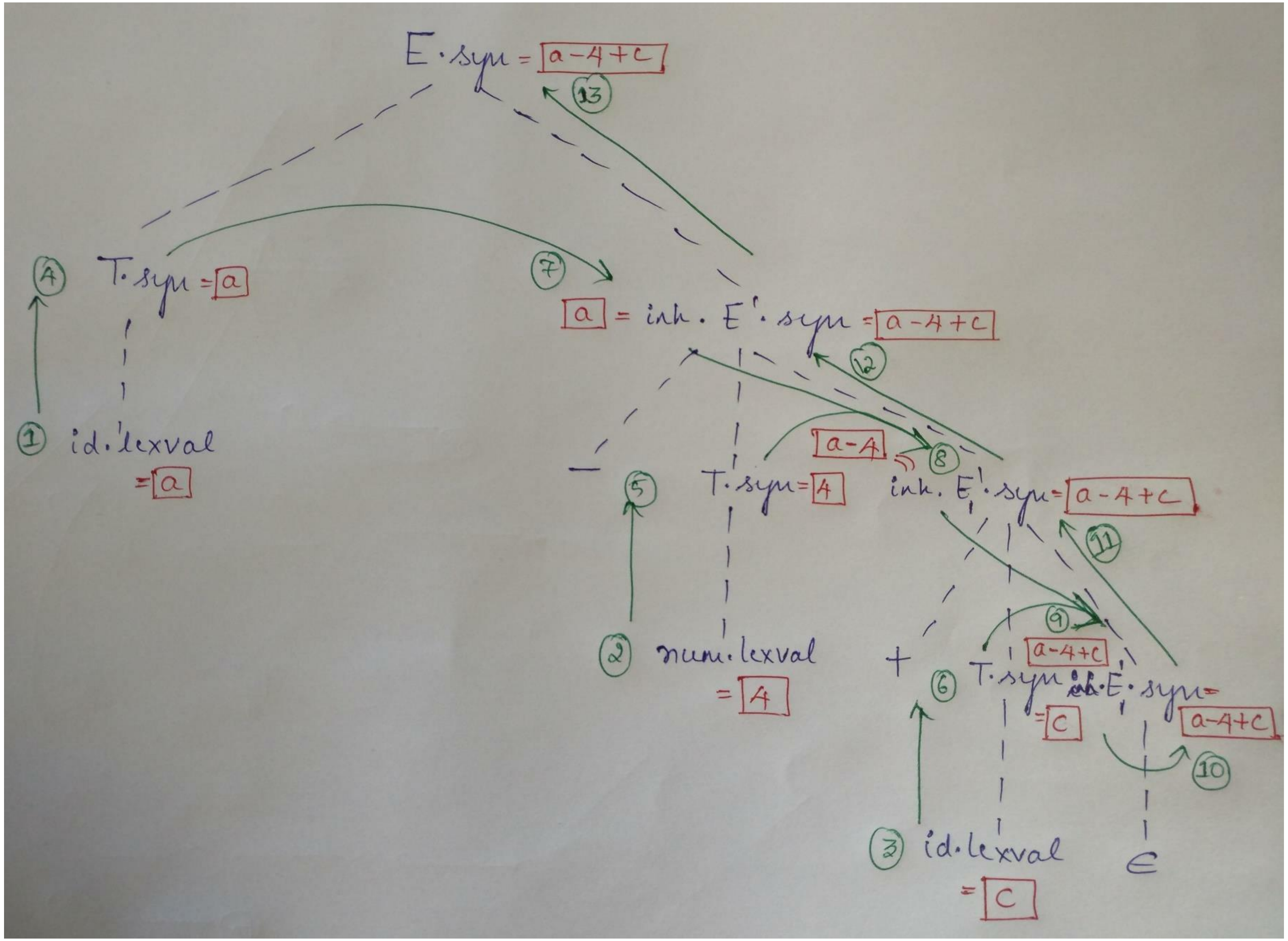
E’ -> €

T -> (E)

T -> id

T -> num

Productions	Semantic Rules
E -> TE'_1	{E.syn = E'.syn; E'.in = T.syn;}
E' -> +TE'_1	{E'_1.in = E'.in + T.syn; E'_1.syn = E'_1.syn; }
E' -> -TE'_1	{E'_1.in = E'.in - T.syn; E'_1.syn = E'_1.syn; }
E' -> €	{E'.syn = E'.in; }
T -> (E)	{T.syn = E.syn ; }
T -> id	{T.syn = id.lexval; }
T -> digit	{T.syn = digit.lexval; }



Dependency graph and Annotated parse tree for ***a-4+c***

Evaluation order(Topological order):
1 4 7 2 5 8 3 6 9 10 11 12 13

For the given grammar,

(a) design SDD.

(b) Construct the annotated parse tree

(c) Construct Dependency graph for the string *id + id * id*

(d) Mention the topological sorting order

E -> TE’

E’ -> +TE'_1

E’ -> €

T -> FT’

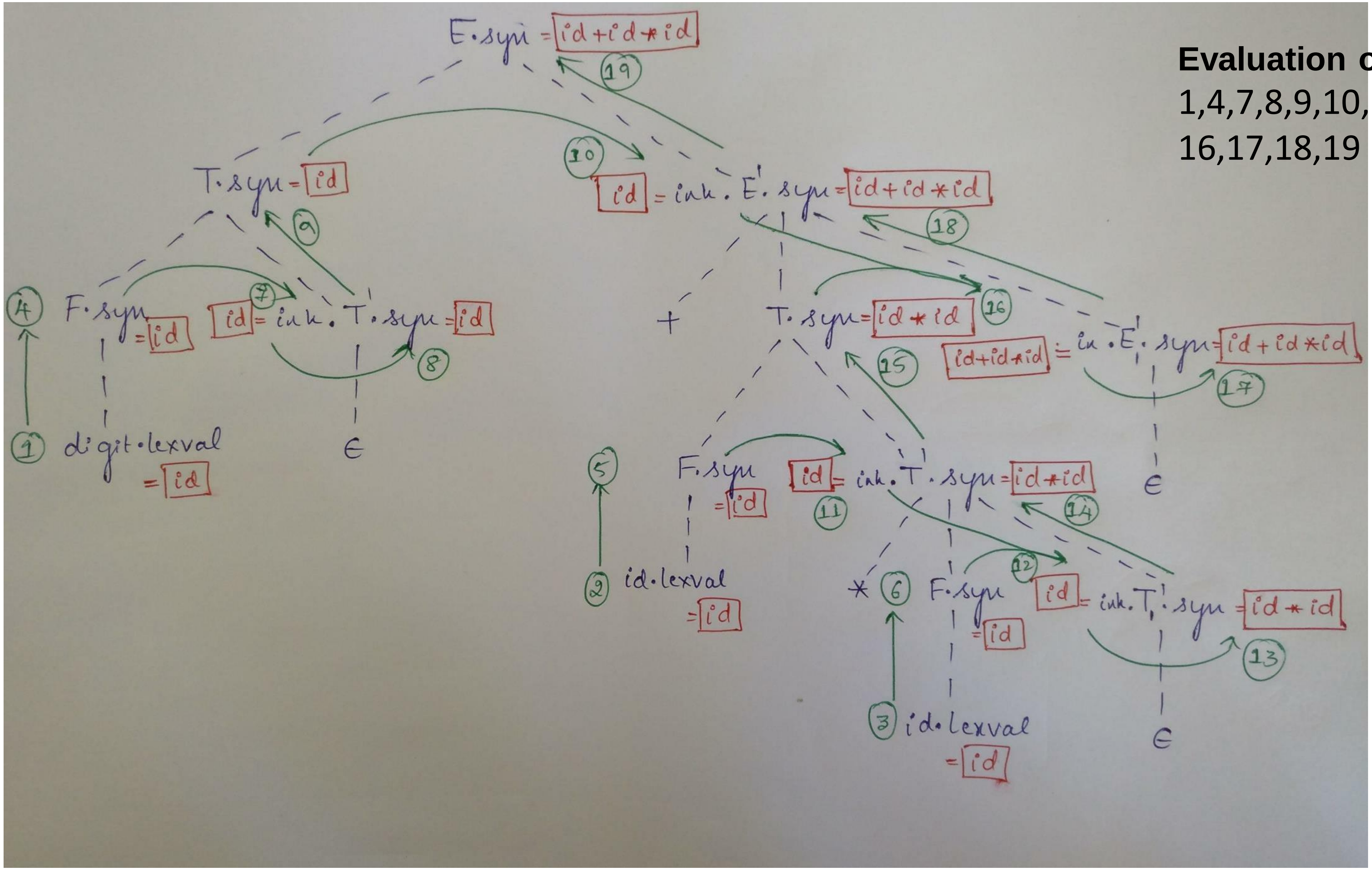
T’ -> *FT'_1

T’ -> €

F-> (E)

F -> id

Productions	Semantic Rules
E -> TE’	E’.in = T.syn; E.syn = E’.syn
E’ -> +TE'_1	E'_1.in = E’.in + T.syn; E’.syn = E'_1. syn
E’ -> €	E'_1 .syn = E'_1 .in
T -> FT’	T’.in = F.syn; T.syn = T’.syn
T’ -> *FT'_1	T'_1 .in = T’.in * F.syn; T’.syn = T'_1 syn
T’ -> €	T'_1 .syn = T'_1 .in
F -> (E)	F.syn = E.syn ;
F -> id	F.syn = id.lexval



Evaluation order(Topological order):
1,4,7,8,9,10,2,5,11,3,6,12,13,14,15,
16,17,18,19

Dependency graph and Annotated parse tree for *id+id*id*

SDD for ***Type Declaration***

1) **int a;**

CFG

S -> TI;

T -> int

I -> id

SDD for ***Type Declaration***

1) **int a;**

Productions	**Semantic Rules**
S -> TI;	{I.intype = T.type ; I.inwidth = T.width}
T -> int	{T.type = int; T.width = 4;}
I -> id	{ addType.symtab(id.entry, I.inType); addWidth.symtab(id.entry, I.inWidth); }

SDD for ***Type Declaration***

2) **int a, b, c;**

CFG

S -> TI;

T -> int

I -> I_1 , id

I -> id

SDD for ***Type Declaration***

2) **int a, b, c;**

Productions	**Semantic Rules**
S -> TI;	{I.inType = T.type ; I.inwidth = T.width}
T -> int	{T.type = int; T.width = 4;}
I -> I_1 , id	{I_1 .in = I.in; I_1 .inwidth = I.inwidth addType.symtab(id.entry, I.inType); addWidth.symtab(id.entry, I.inWidth);}
I -> id	{ addType.symtab(id.entry, I.inType); addWidth.symtab(id.entry, I.inWidth); }

3) SOLVE:- **float id1, id2, id3**

Write SDD to determine whether the type in an array type or basic type and also construct annotated parse for the following string

“int [3][4]a”

D -> TI

T -> BC

B -> int

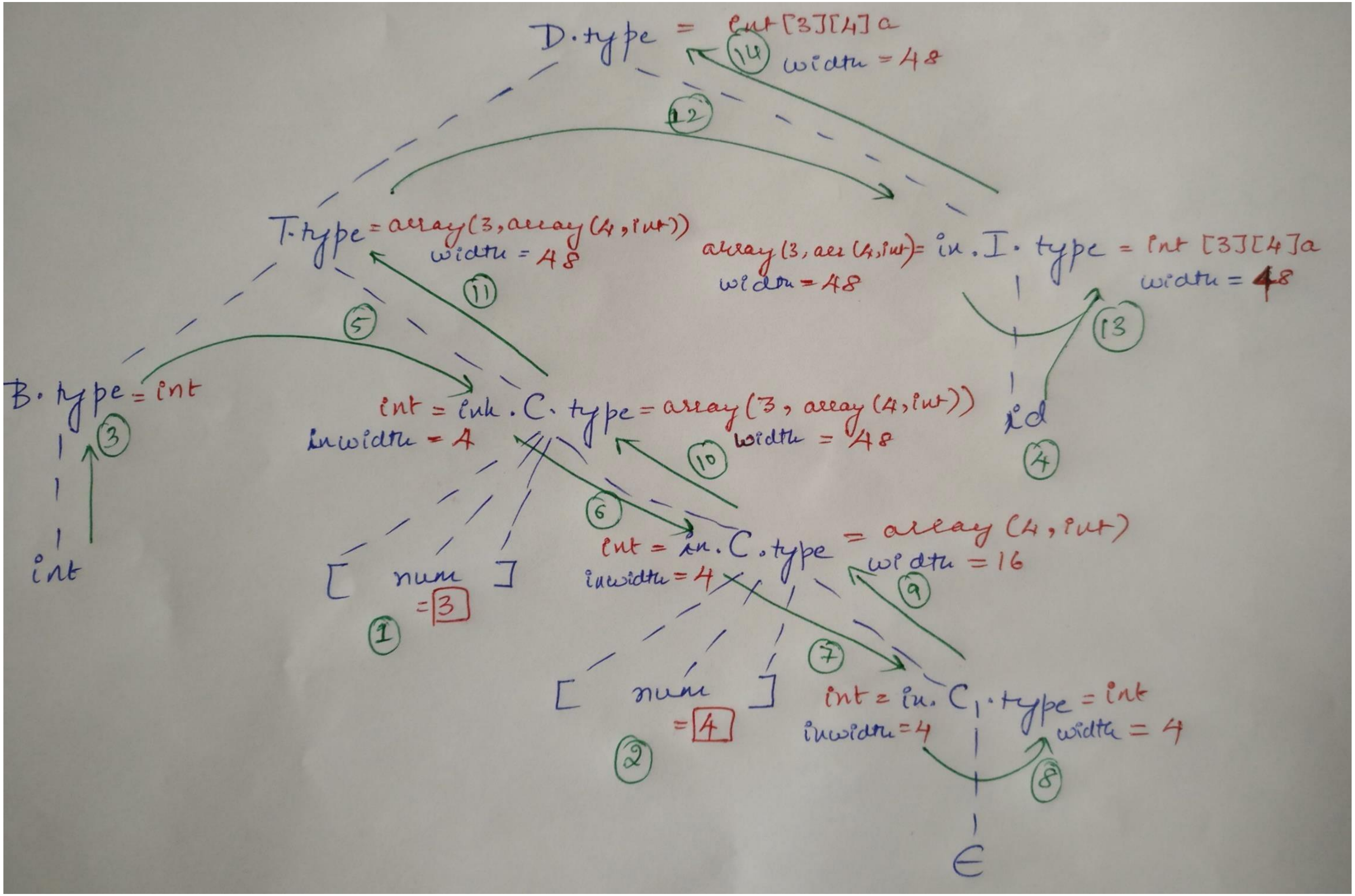
C -> €

C -> [num]C

I -> id

Write SDD to determine whether the type in an array type or basic type

Productions	Semantic Rules
D -> TI	{ I.inType = T.type, I.inWidth = T.inWidth}
T -> BC	{ C.inType = B.type; C.inWidth = B.width; T.type = C.type; T.width = C.width ;
B -> int	{ B.type = int ;}
C -> €	{C_1.type = C_1 .inType ; C_1.width = C_1 .inWidth ;}
C -> [num] C_1	{ C_1.inType = C.inType; C_1.inWidth = C.inWidth; C.type = array(num.lexval, C1.type); C.width = num.lexval * C1.width ; }
I -> id	{ addType.symtab(id.entry, I.inType); addWidth.symtab(id.entry, I.inWidth); }



D.type = int[3][4] a
width = 48
14
12
T.type = array(3, array(4, int))
width = 48
array(3, arr(4, int)) = in.I.type = int [3][4] a
width = 48
width = 48
11
13
5
B.type = int
3
int = ink.C.type = array(3, array(4, int))
inwidth = 4
width = 48
10
id
4
6
int
[
num
]
= 3
1
int = in.C.type = array(4, int)
inwidth = 4
width = 16
9
7
[
num
]
= 4
2
int = in.C1.type = int
inwidth = 4
width = 4
8
ϵ

SOLVE:

1) Generate SDD to construct a Syntax tree for the expression
   (a) ***a – 4 + c***
   (b) ***4 + 5 * 2***

2) Generate SDD for ***if statement*** and also construct abstract syntax tree

3) Generate SDD for ***if else statement*** and also construct the abstract syntax tree

4) Generate SDD for ***while construct*** and also construct abstract syntax tree

5) Generate SDD for ***for construct*** and also construct abstract syntax tree

6) SDD to generate Intermediate code for ***Expressions***

a) ***p = q + r + s***

b) ***a = b + - c***

7) SDD to generate Intermediate code for ***Boolean expressions***

10) SDD to generate Intermediate code for ***if statement***

11) SDD to generate Intermediate code for ***if else statement***

12) SDD to generate Intermediate code for ***while statement***

13) SDD to generate Intermediate code for ***for statement***

14) SDD to generate Intermediate code for ***switch statement***

1a) SDD for the construction of Syntax tree for the expression *a – 4 + c*

CFG

E -> E_1 + T

E -> $E_1 -$ T

E -> T

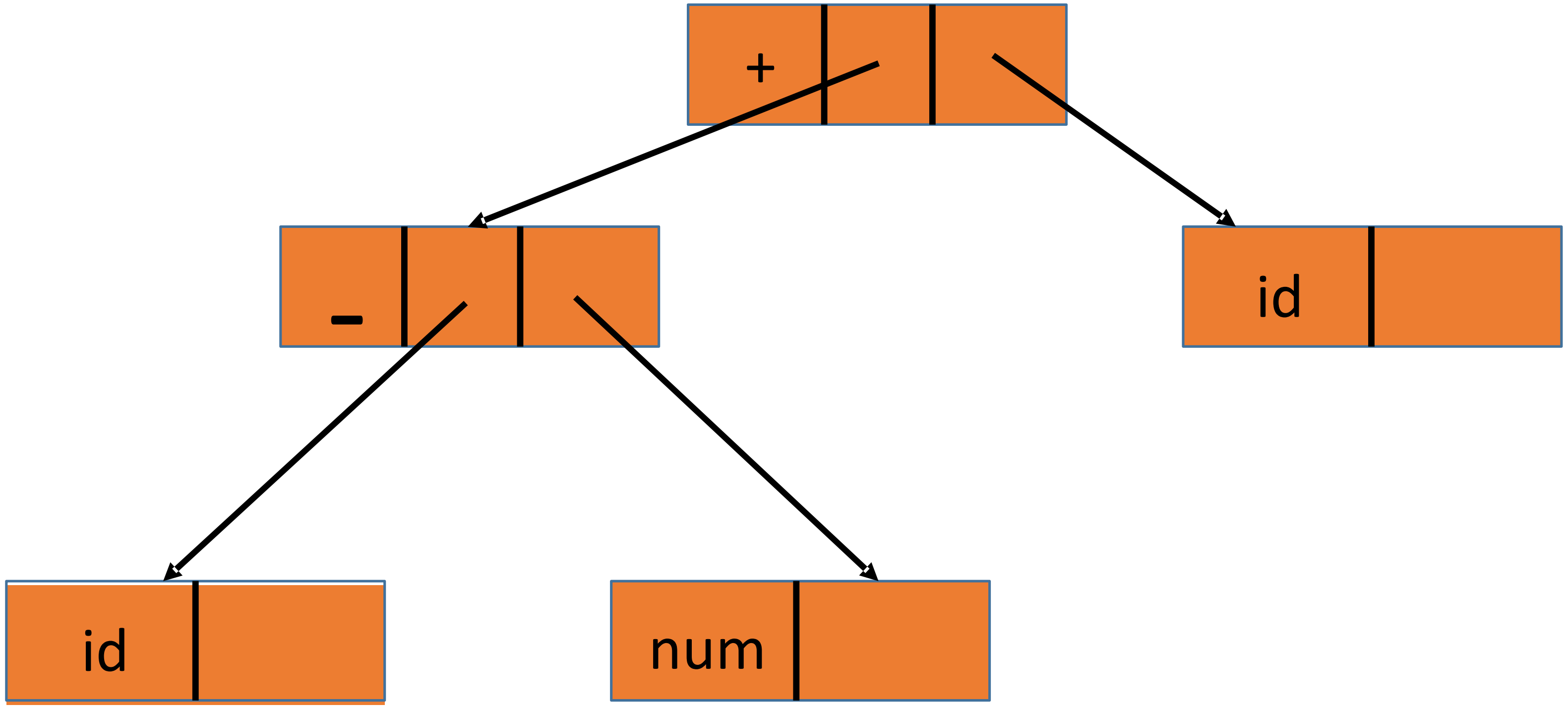
T -> (E)

T -> id

T -> num

Productions	Semantic Rules
E -> E_1 + T	E.node = new Node('+', E_1 .node, T.node)
E -> E_1 $-$ T	E.node = new Node('-', E_1 .node, T.node)
E -> T	E.node = T.node
T -> (E)	T.node = $E.\text{node}$
T -> id	T.node = new LeafNode(id, id.entry)
T -> num	T.node = new LeafNode(num, num.val)

SDD for the constructing of syntax tree for a-4+c



Abstract Syntax tree for a-4+c

2) SDD for the construction of Syntax tree for the following if-statement

```
if(a<b)
{
        a=a+b;
}
```

CFG

S -> if(B) $\{ S_1 \}$

S -> Assign

Assign -> id = E;

B -> $E_1 < E_2$

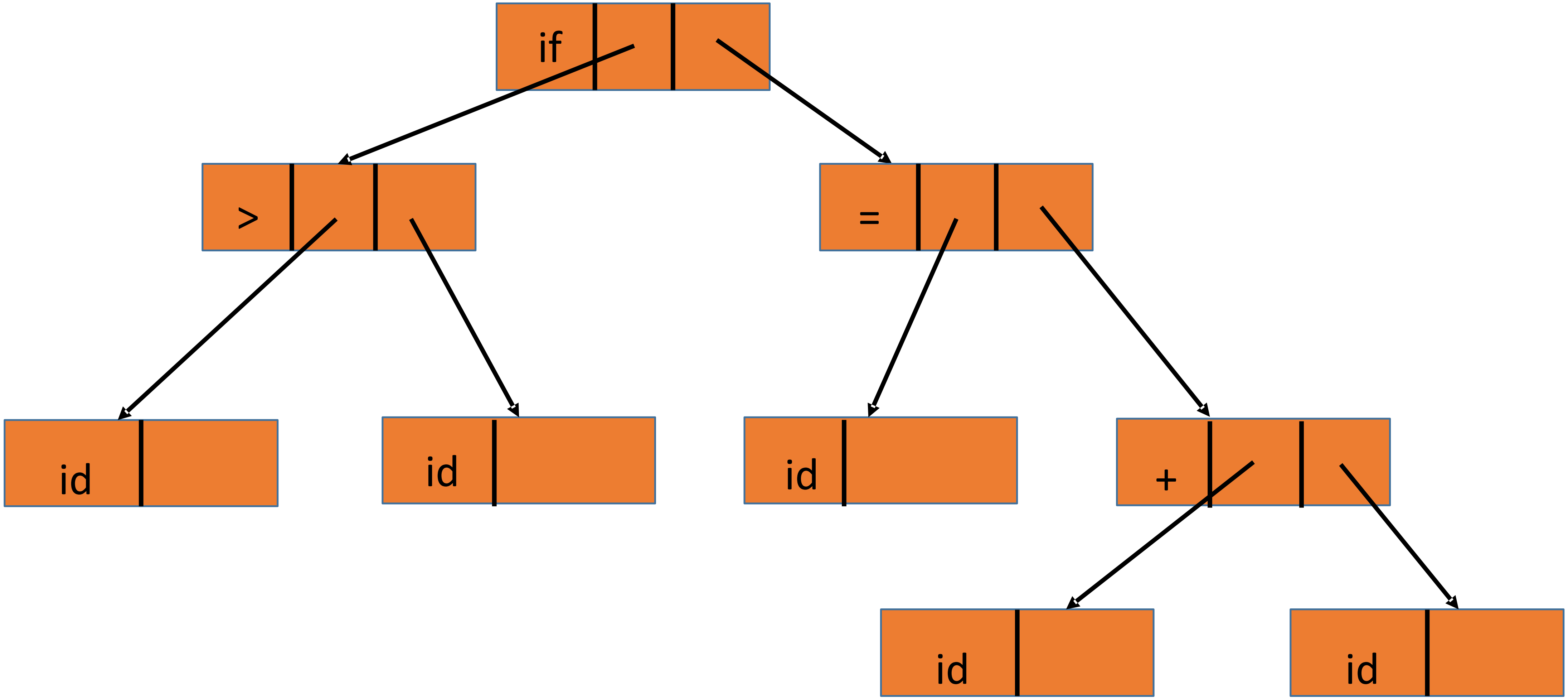
E -> $E_1 + E_2$

E -> id

Productions	Semantic Rules
S -> if(B) S1	S.node = new Node(‘if’, B.node, S1.node)
S -> Assign	S.node = Assign.node
Assign -> id = E;	Assign.node = new Node(‘=’, new LeafNode(id,id.entry), E.node)
B -> E1 < E2	B.node = new Node(‘<’, E1.node, E2.node)
E -> E1 + E2	E.node = new Node(‘+’, E1.node, E2.node)
E -> id	E.node = new LeafNode(id, id.entry)

SDD for the constructing of syntax tree for if statement

Abstract Syntax tree for- *if(a<b) {a=a+b;}*



SDD to generate Intermediate code(TAC) for ***p = q + r + s***

SDD to generate Intermediate code(TAC) for ***a = b + - c***

SDD to generate Intermediate code(TAC) for ***p = q + r + s***

Productions	Semantic Rules
S -> id = E;	S.code = E.code ||gen(id.lexval '=' E.addr)
E -> E_1 + id	E.addr = new Temp() E.code = E_1 .code || id.lexval || gen(E.addr '='E_1 .addr '+' id.lexval)
E -> id	E.addr = id.entry E.code = ' '

SDD to generate Intermediate code(TAC) for ***a = b + - c***

Productions	Semantic Rules
S -> id = E	S.code = E.code || gen(id.lexval '=' E.addr)
E -> E_1 + id	E.addr = new Temp() E.code = E_1 .code || gen(E.addr '='E_1 .addr '+' id.lexval)
E -> - id	E.addr = new Temp(); E.code = gen(E.addr '=' '-' id.lexval)
E -> (E_1)	E.addr = E_1 .addr; E.code = E_1 .code
E -> id	E.addr = id.lexval E.code = ' '

SDD for Flow of Control Statements

SDD to generate Intermediate code for Boolean expressions

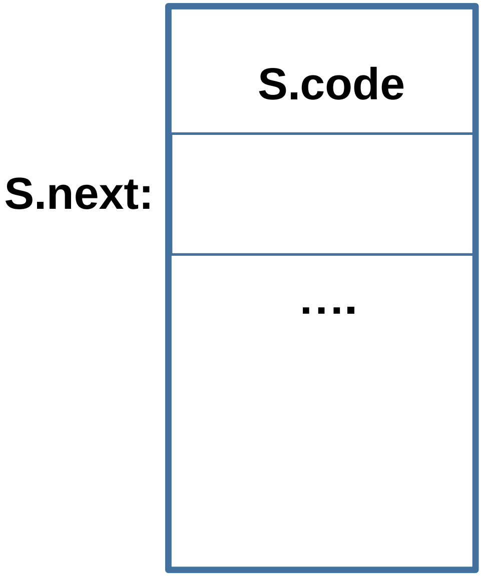
B -> B || B
B -> B && B
B -> !B
B -> E Relop E
B -> true
B -> false

SDD to generate Intermediate code for Boolean expressions

Productions	Semantic Rules
B -> B_1 || B_2	B.true= B_1 .true; B_1 .false = new Label(); B.true= B_2 .true ; B.false= B_2 .false; B.code = B_1 .code || label(B_1 .false) || B_2 .code ;}
B -> B_1 && B_2	{B.false = B_1 .false ; B_1 .true = new Label(); B.true= B_2.true ; B.false= B_2 .false; B.code = B_1.code || label(B_1. true) || B_2 .code ;}

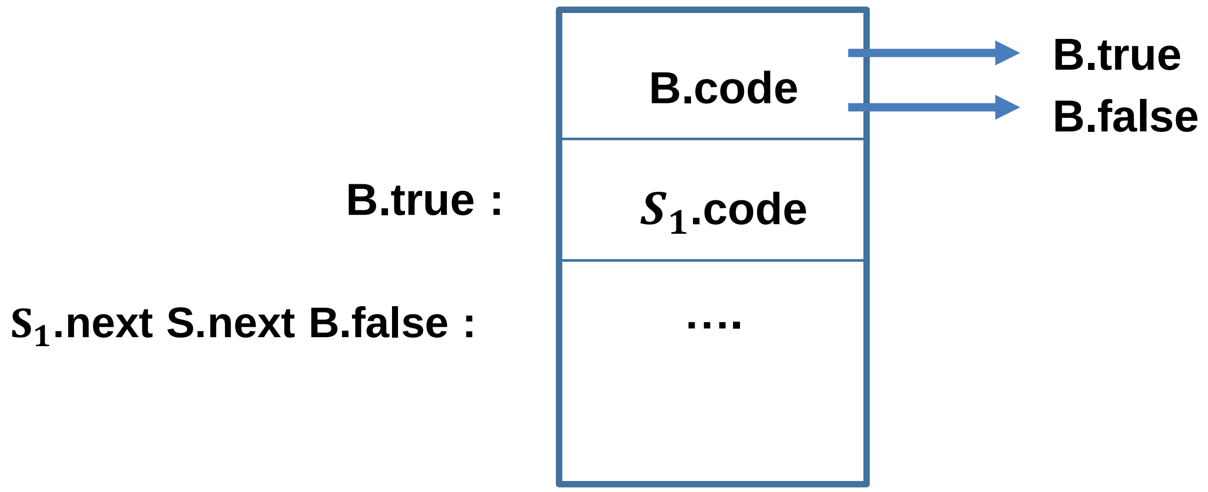
Productions	Semantic Rules
B -> !B_1	{B.false= B_1.true ; B.true= B_1 .false; B.code = B_1 .code;}
B -> E_1 Relop E_2	{B.code = E_1 .code ||E_2.code || gen(‘if’ E_1 .addr relop E_2 .addr ‘goto’ B.true) || gen(‘goto’ B.false)}
B -> true	{B.code = gen(‘goto’ B.true) ;}
B -> false	{B.code = gen(‘goto’ B.false) ;}

<table>
<tr><th>Productions</th><th>Semantic Rules</th></tr>
<tr><td>P -> S</td><td>S.next = new Label();
P.code = S.code || label(S.next)</td></tr>
<tr><td></td><td></td></tr>
</table>



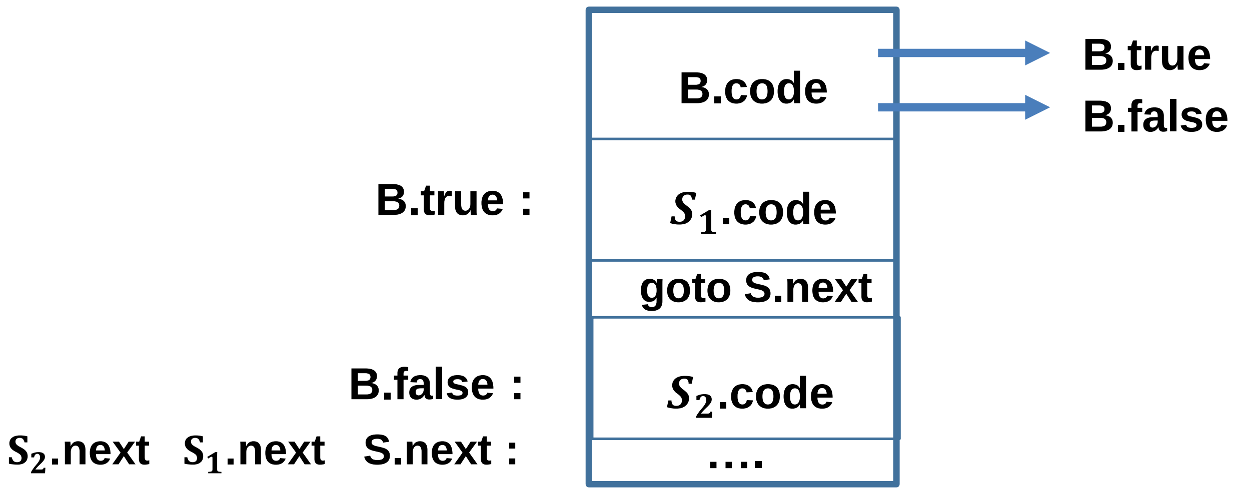
SDD to generate Intermediate code for ***if statement***

Productions	Semantic Rules
S -> if (B) S_1	B.true = new Label(); B.false = S.next; S1.next = S.next; S.code = B.code || label(B.true) || S1.code



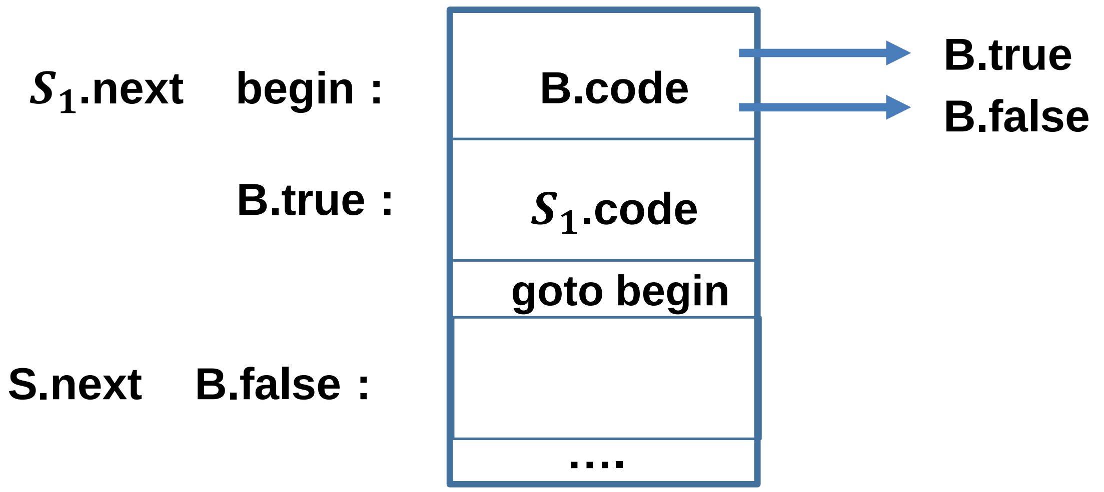
SDD to generate Intermediate code for ***if else statement***

Productions	Semantic Rules
S -> if (B) S_1 else S_2	B.true = new Label(); B.false = new Label(); S1.next = S2.next = S.next S.code = B.code || label(B.true) || S1.code || gen('goto' S.next) || label(B.false) || S2.code

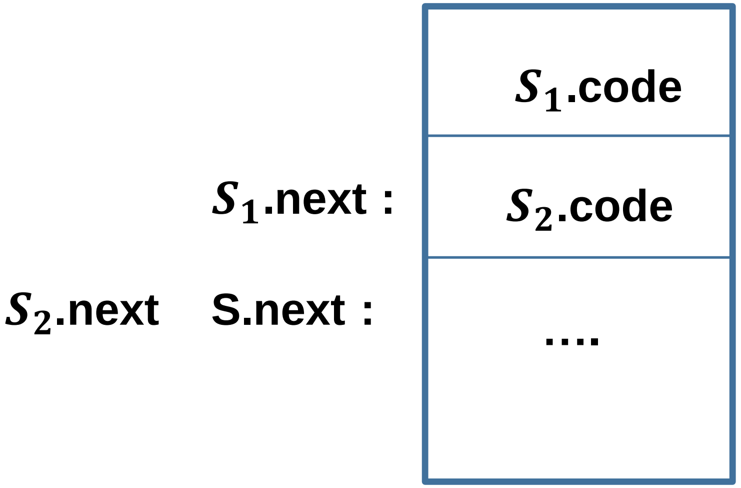


SDD to generate Intermediate code for ***while statement***

Productions	Semantic Rules
S -> while (B) S_1	Begin = new Label(); B.true = new Label(); B.false = S.next S1.next = begin S.code = label(begin) || B.code || label(B.true) || S1.code || gen('goto' begin)



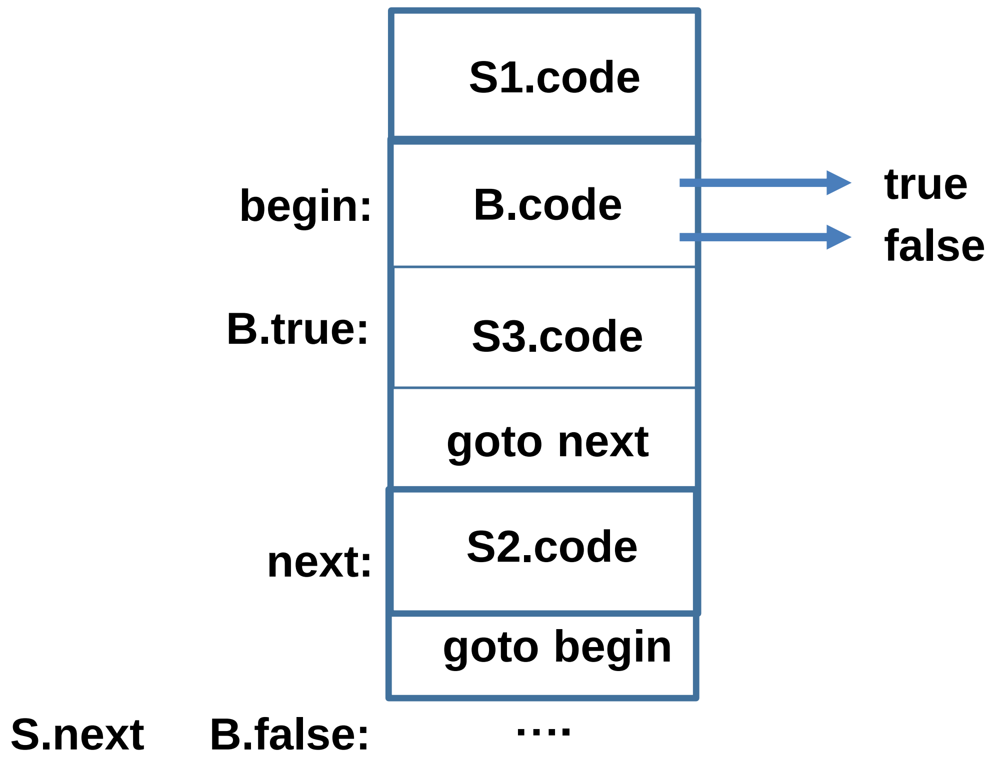
Productions	Semantic Rules
S -> $S_1 S_2$	S1.next = new Label(); S2.next = S.next S.code =S1.code||label(S1.next)||S2.$code$



SDD to generate Intermediate code for ***for statement***

Productions	Semantic Rules

S -> for(S1; B; S2)S3	begin = new Label();
	B.true = new Lable();
	next = new label();
	S2.next = begin;
	S3.next = next;
	B.false = S.next;
	S.code = S1.code || label(begin) || B.code || label(B.true) || S3.code || gen(‘goto’ next) || label(next) || S2.code || gen(‘goto’ begin) ;}


S1.code
begin:
B.code
true
false
B.true:
S3.code
goto next
next:
S2.code
goto begin
S.next
B.false:
····

Switch statements or case statements

1 Evaluate the controlling expression

2 Branch to the selected case

3 Execute the code for that case

4 Branch to the statement after the case

Case Statement: Code Layout

```
switch E
begin
        case V1: S1
        case V2: S2
        …
        case Vn-1: Sn-1
        default: Sn
end
```

code to Evaluate E into t

```
goto Ltest
L1: code for S1
    goto Lnext
L2: code for S2
    goto Lnext
    …
Ln-1: code for Sn-1
goto Lnext
Ln: code for Sn
goto Lnext
Ltest: if t = V1 goto L1
       if t = V2 goto L2
       …
      if t = Vn-1 goto Ln-1
      goto Ln
Lnext:
```

```
S → switch E List end ||
        S.code = append(E.code,gen('goto Ltest'),List.code,gen('Ltest:')
        while(queue not empty) do {
            (vi,Li) = pop.queue;
            if (vi = default)
                S.code = append(S.code,gen('goto Li'));
            else    S.code = append(S.code,gen('if t = vi goto Li'));

Case → case Value : S ||
        Case.code = append(gen('Li:'),S.code,gen('goto Lnext');
        queue.push((Value.val,Li));

List → Case ; List1  || List.code = append(Case.code,List1.code);
List → default : S ||List.code = append(gen('Li:'),S.code,gen('goto Lnext');
queue.push((default,Li))

List → ε || List.code = append(gen('Li:'),gen('goto Lnext');   queue.push((default,Li))
```

Syntax Directed Translation Schemes(SDTS)

Syntax Directed Translation(SDT)

- Translation schemes are implementation of SDD.
- They are more implementation specific.
- They indicate the order in which the semantic actions and the attributes are to be evaluated.

Syntax Directed Translation Scheme contains

Context free grammar

Semantic Actions (program segments)

The semantic rules can be embedded anywhere within the body of the production. The rules can appear anywhere

If we have a production B -> X {a} Y, the action a is executed after X is recognized or all the terminals derived from X.

- If a parse is bottom-up, we perform action '*a'* as soon as this occurance of X appears on the top of the parsing stack.
- If the parse is top-down, we perform action '*a'* just before we attempt to expand this occurence of Y or check for Y on the input.

Note: ***Not all SDT's can be implemented during parsing***

SDTs for converting an infix expression to prefix form

L -> En

E -> {printf("+");}   E + T

E -> T

T -> {printf("*");}   T * F

T -> F

F -> (E)

F -> digit {printf("digit.lexval");}

It is difficult to implement this SDT either during bottom-up or top-down because the parser would have to perform critical actions, like printing of * or + before it knows whether these symbols appear in the input.

Applications of Syntax-Directed Translation

1) Construction of Syntax tree
2) Type checking
3) Generation of Intermediate code

Types of Translation schemes:

1) SDT’s with Actions Inside productions (sometimes called Problematic SDT)

2) Postfix schemes (Conversion of S - attributed SDD to SDT)

3) Conversion of L-attributed SDD to SDT

SDT's with actions anywhere in the production(Problematic SDTs)

Any SDT can be implemented as follows:

- Ignoring the actions, parse the input and produce a parse tree as a result.
- Then, examine each interior node N, say one for production A -> α. Add additional children to N for the actions in α, so the children of N from left to right have exactly the symbols and the actions of α.
- Perform a preorder traversal of the tree, and as soon as a node labelled by an action is visited, perform that action.

SDTs for converting an infix expression to prefix form

L -> En

E -> {printf("+");}   E + T

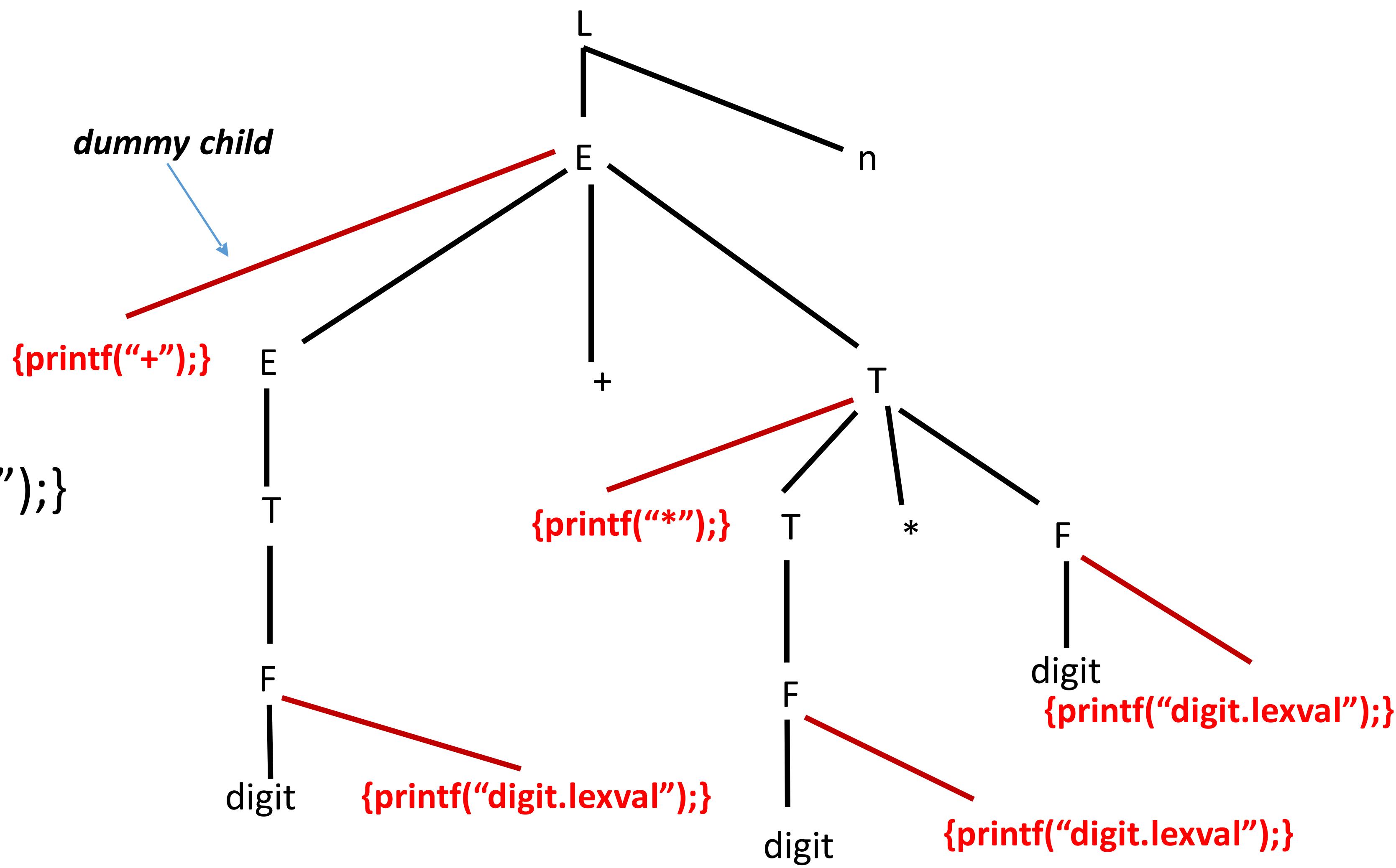
E -> T

T -> {printf("*");}   T * F

T -> F

F -> (E)

F -> digit   {printf("digit.lexval");}



What does the following SDT scheme print for the string “5 + 4 – 2" ?

E→TR

R→+T {print(“+”);} R1

R→−T {print(“-”);} R1

R→ ∈

T -> F

F -> digit {print(digit.lexval);}

Postfix schemes (Conversion of S- attributed SDD to SDT scheme)

<u>Evaluation of S-attributed SDD</u>

- S-attributed SDDs will have only synthesized attributes and can be evaluated by a bottom up parser.
- Since the attributes in the semantic actions are only synthesized, the actions can be placed at the end of the production.

Rule to be followed for evaluation:

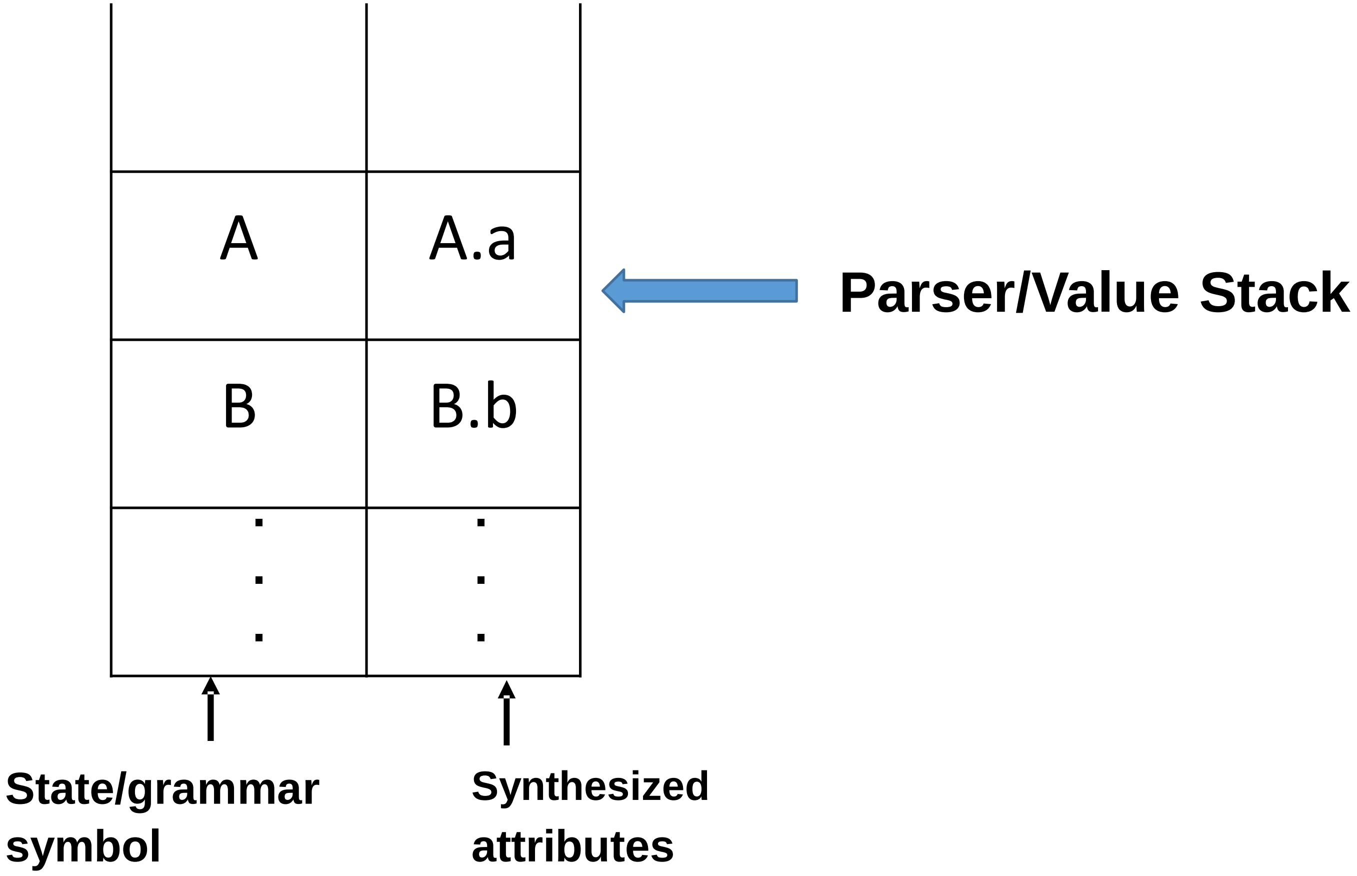
- *Consider the following production.*

  A -> BCD

  before reducing BCD to A, the attributes of B, C and D must be computed before attribute of A which appears on the stack.

- *Corresponding semantic action associated with the production must be executed.*

- The parser stack is extended to have parallel value stack.
- If the Action appears at the end of production in a SDT, such SDTs are called Postfix SDTs.



Implement parser-stack of the following postfix SDTs for ***(3+4)*5n***

L -> En

E -> E_1 + T

E -> T

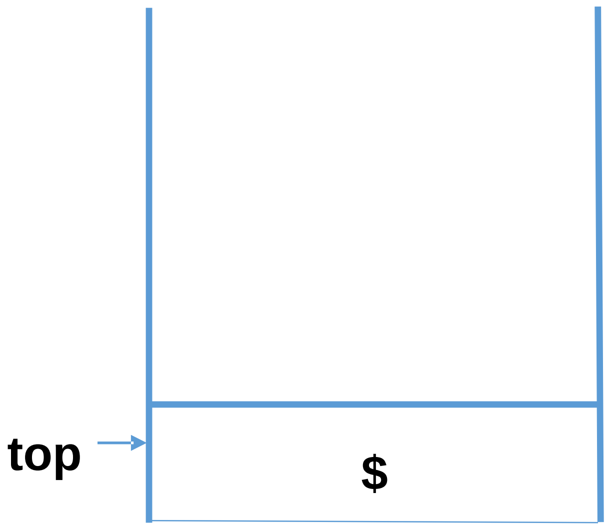
T -> T_1 * F

T -> F

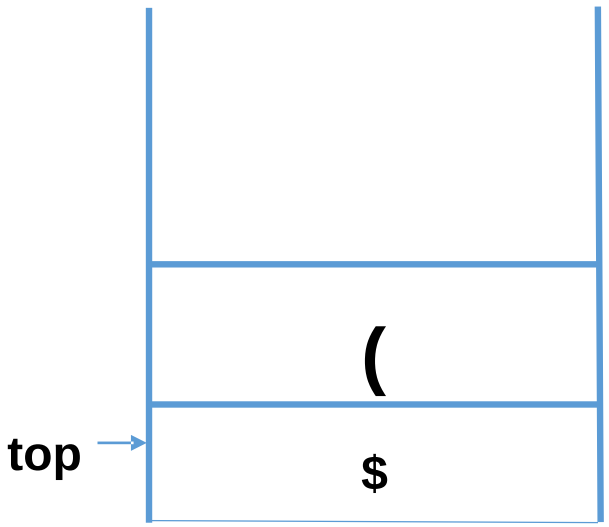
F -> (E)

F -> digit

Stack	***Input Buffer***	***Actions***
$	(3+4)*5n$	shift (

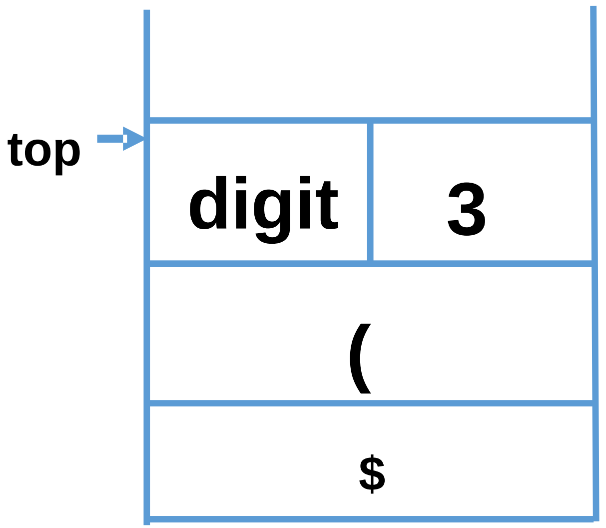


Stack	***Input Buffer***	***Actions***
$(	3+4)*5n$	shift 3

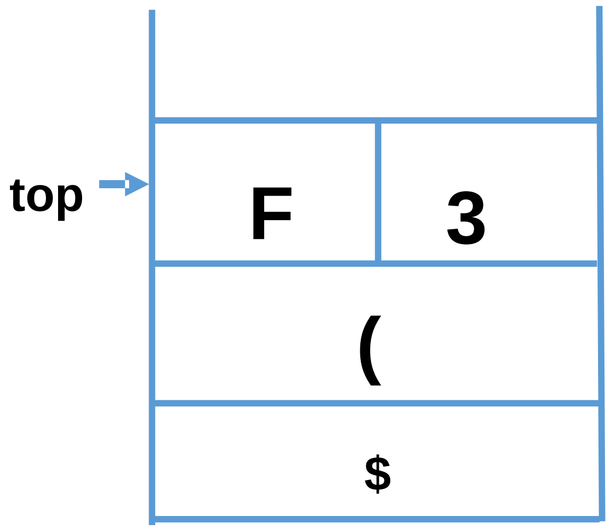


Stack | ***Input Buffer*** | ***Actions***

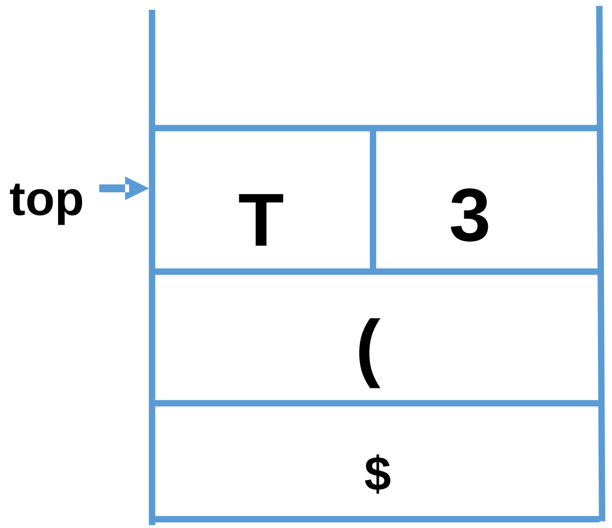
Stack	***Input Buffer***	***Actions***
$(3	+4)*5n$	reduction [F -> digit]



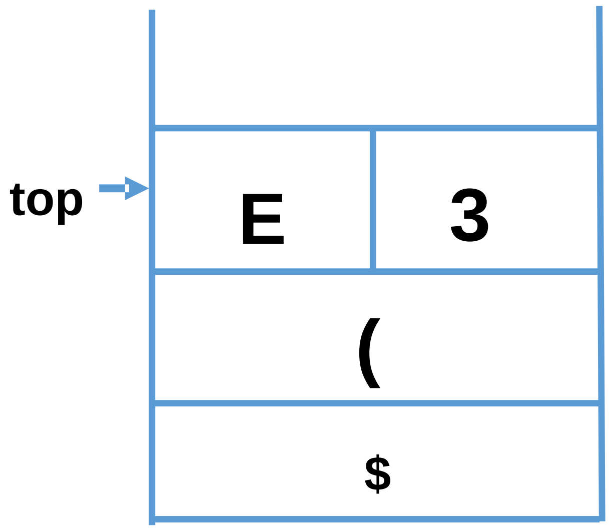
Stack	***Input Buffer***	***Actions***
$(F	+4)*5n$	reduction [T -> F]



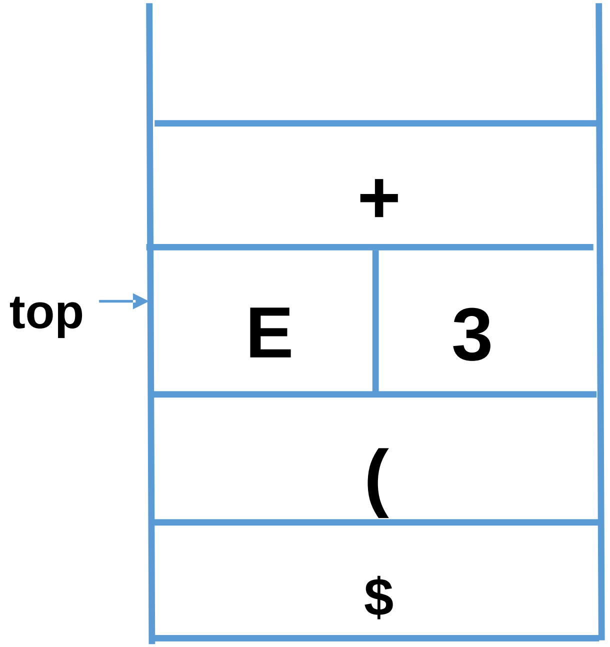
Stack	***Input Buffer***	***Actions***
$(T	+4)*5n$	reduction [E -> T]



Stack	***Input Buffer***	***Actions***
$(E	+4)*5n$	shift '+'



Stack	Input Buffer	Actions
$(E+	4)*5n$	shift 4



Stack

$(E+4

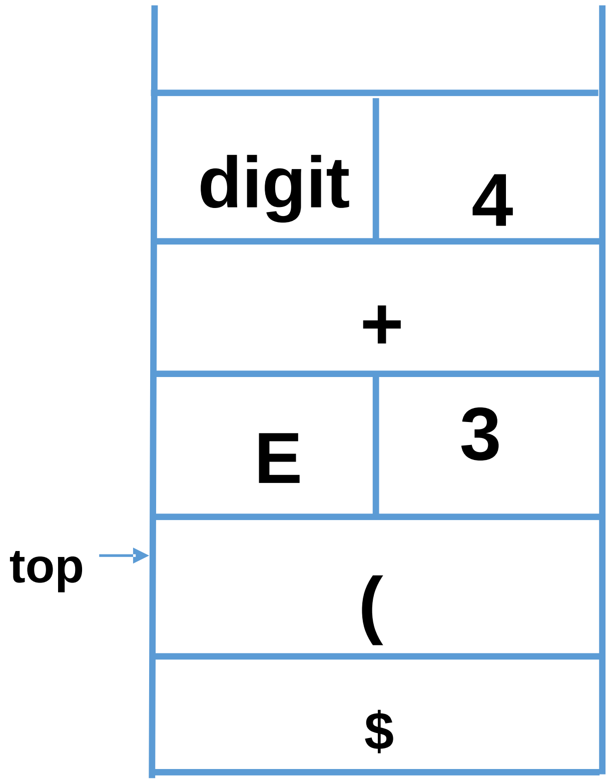
Input Buffer

)*5n$

Actions

Reduce

[F -> digit]



Stack

$(E+F

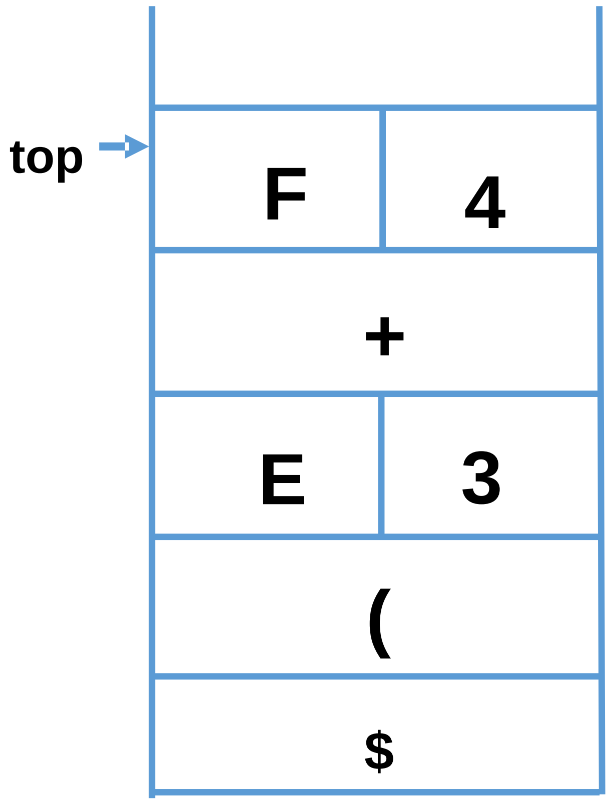
Input Buffer

*)5n$

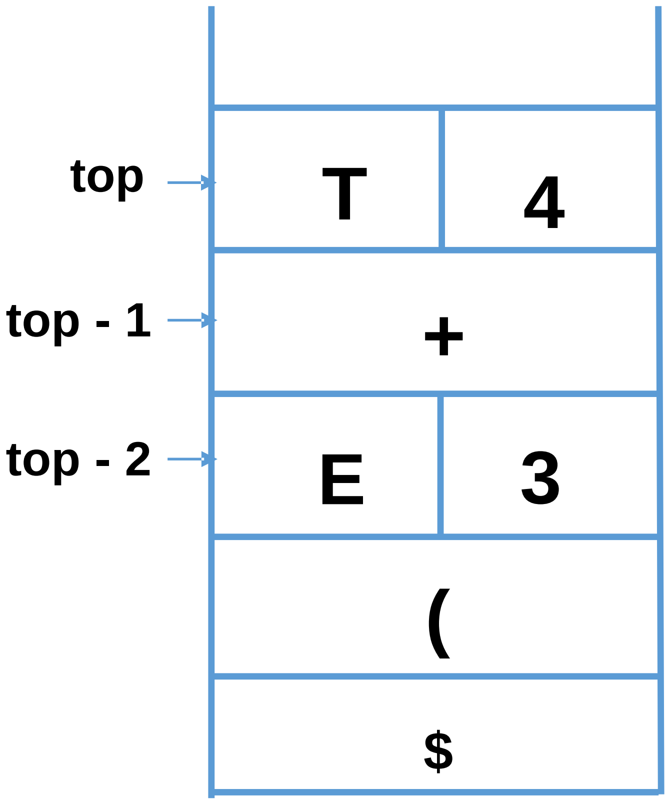
Actions

reduction

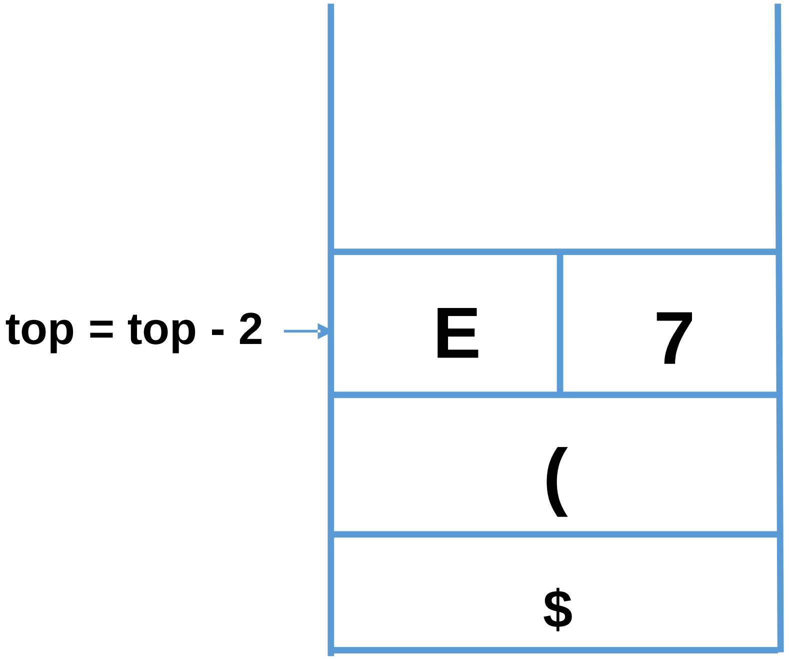
[T -> F]



Stack	***Input Buffer***	***Actions***
$(E+T	*)5n$	reduction [E -> E + T]



Stack	***Input Buffer***	***Actions***
$(E	)*5n$	shift ‘)’



{s[top-2].val = s[top-2].val + s[top].val; top = top-2;}

Stack

$(E)

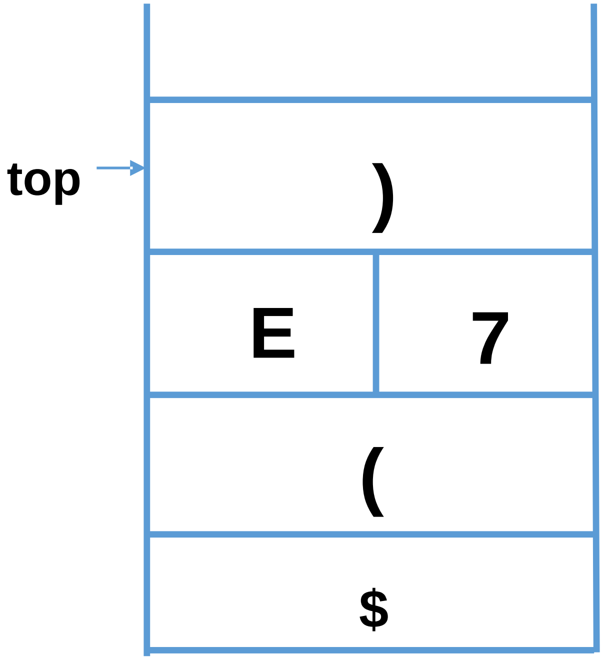
Input Buffer

*5n$

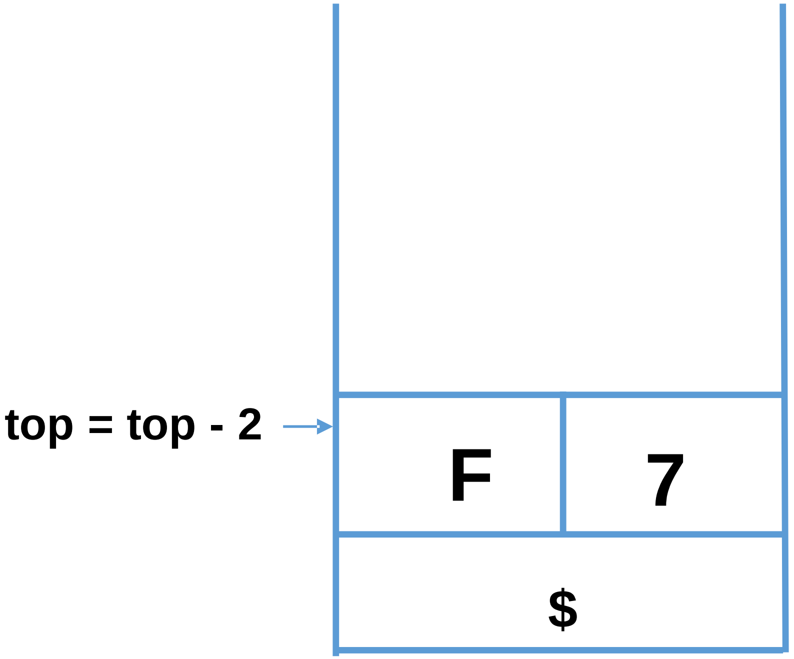
Actions

reduction

[F -> (E)]

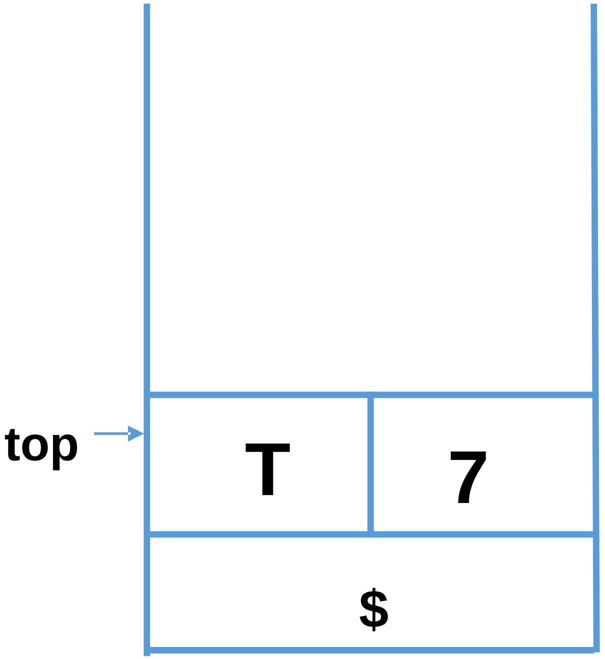


Stack	***Input Buffer***	***Actions***
$F	*5n$	reduction [T -> F]

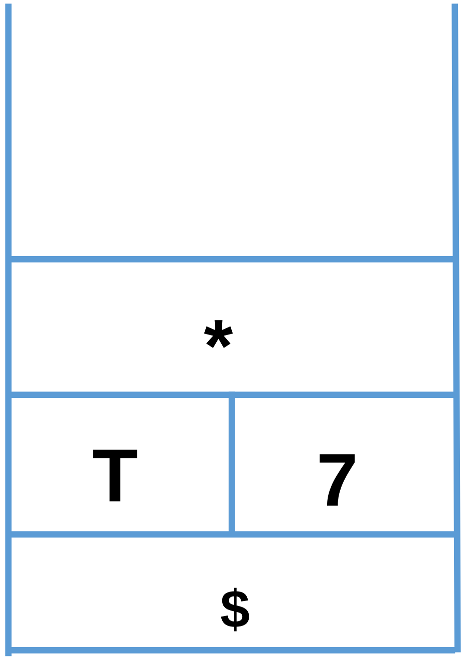


{s[top-2].val = s[top-1].val ; top = top-2;}

Stack	***Input Buffer***	***Actions***
$T	*5n$	shift ‘*’



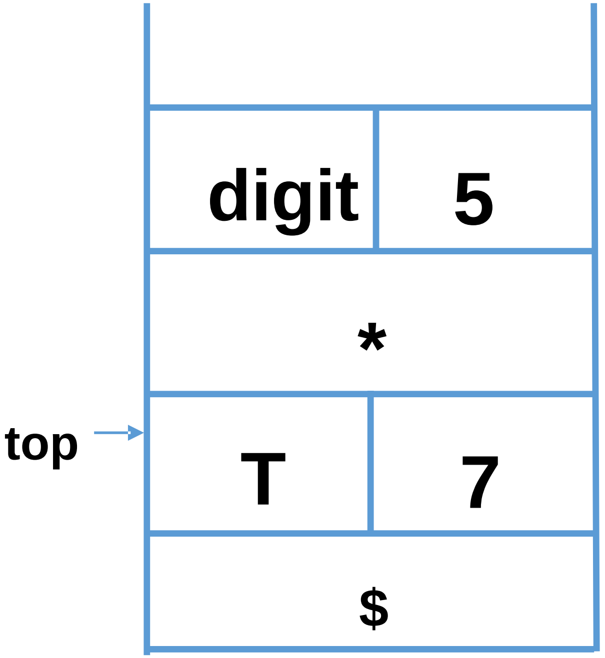
Stack	***Input Buffer***	***Actions***
$T*	5n$	shift 5



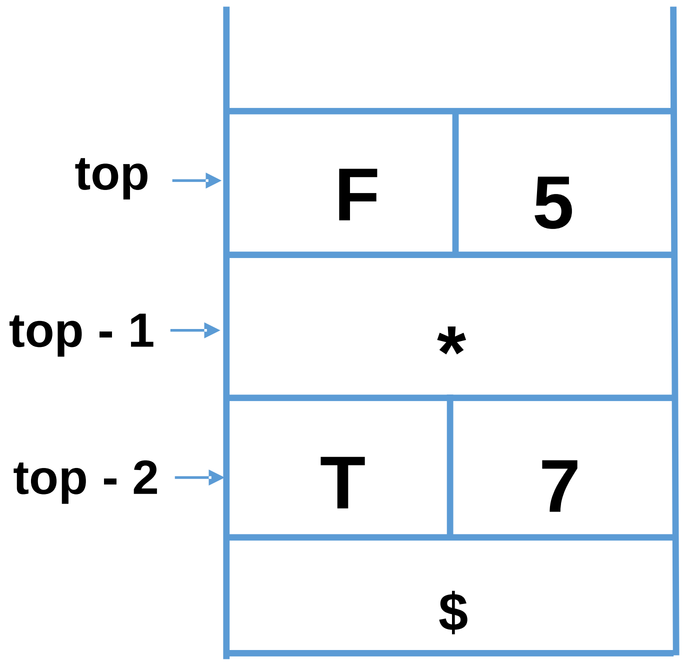
Stack ***Input Buffer*** ***Actions***

$T*5 n$ reduction

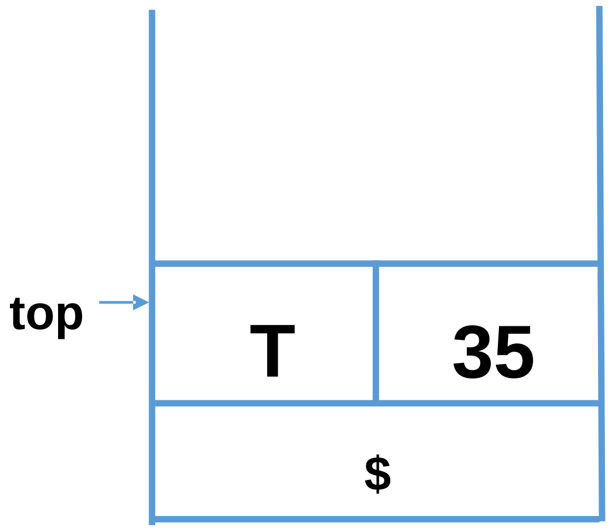
[F -> digit]



Stack	***Input Buffer***	***Actions***
$T*F	n$	reduction [T -> T * F]

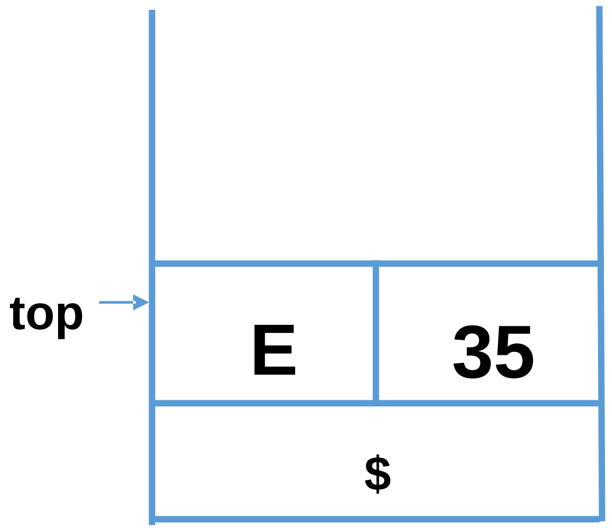


Stack	***Input Buffer***	***Actions***
\$T	n\$	reduction [E -> T]



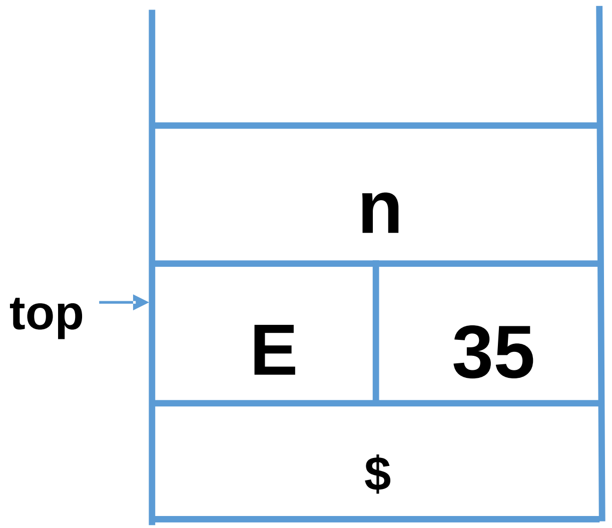
{s[top-2].val = s[top-2].val * s[top].val; top = top-2;}

Stack	***Input Buffer***	***Actions***
\$E	n\$	shift n

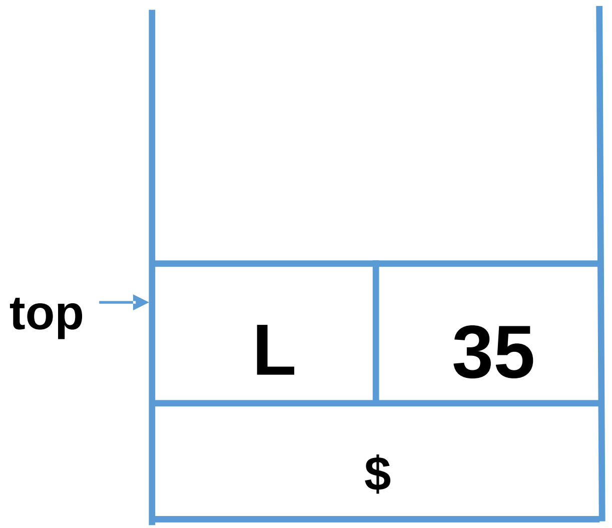


Stack | ***Input Buffer*** | ***Actions***

Stack	***Input Buffer***	***Actions***
$En	$	reduction [L -> En]



Stack	***Input Buffer***	***Actions***
$L	$	Accepted



{print(s[top-1].val); top = top – 1;}

Postfix SDT for (***3+4)*5n***

Productions	Semantic Rules
L -> En	{print(s[top-1].val); top = top – 1;}
E -> E_1 +T	{s[top-2].val = s[top-2].val + s[top].val; top = top-2;}
E -> T	
T -> T_1 * F	{s[top-2].val = s[top-2].val * s[top].val; top = top-2;}
T -> F	
F -> (E)	{s[top-2].val = s[top-1].val ; top = top-2;}
F -> digit	

Conversion of L-Attributed SDD’s to SDT scheme

1) Translation by traversing a parse tree:

a. Build parse tree and annotate.

b. Build the parse tree, add actions and execute the actions in pre-order.

2) Translation during parsing

a. Use a RDP with a function for each non-terminal.

b. Generate code on the fly, using a RDP.

c. Implement an SDT in conjunction with an LL parser.

d. Implement an SDT in conjunction with an LR parser.

RDP with a function for each non-terminal

A recursive descent parser has a function for each non-terminal.

Parser can be converted to translator as follows:

Lets say, ***non-terminal is A***

a) The **arguments of function *A* are inherited attributes** of nonterminal A.

b) The **return value of function *A*** is a collection of **synthesized attributes** of nonterminal.

In the body of function A, we need to parse and handle attributes.

1. Decide production to expand A.
2. Check whether each terminal appears in the input when it is required.
3. Decide local variables to store the values of computed inherited and synthesized attributes.
4. Call the functions corresponding to the non-terminals in the body of the selected production, along with proper arguments.

RDP with a function for each non-terminal

```
string S(label next) {
        string Scode, Ccode;    //local variables which hold code fragments
        label L1, L2;    //labels for control flow
        if( current input == token while) {
                advance input;
                check whether ‘(‘ is next on the input and advance;
                L1 = new();
                L2 = new();
                Ccode = C(next, L2);  //C.true = L2,    C.false = S.next
                check whether ‘)‘ is next on the input and advance;
                Scode = S(L1);
                return(“label” || L1 || Ccode|| “label” || L2 || Scode);    // return value is
synthesized attribute
        }
        else
          …
         …
}
```

On the fly code generation

```
void S(label next){
        label L1, L2;       //labels for control flow
        if( current input == token while){
                advance input;
                check whether '(' is next on the input and advance;
                L1 = new();
                L2 = new();
                print("label", L1);
                C(next, L2); //C.true = L2,   C.false = S.next
                check whether ')' is next on the input and advance;
                print("label", L2);
                S(L1);
        }
        else
         ...
         ...
}
```

SDT for ***on-the-fly*** code generation for ***while statement***

Productions	**Semantic Rules**
S -> while (C) S_1	L1 = new Label(); L2 = new Label(); S1.next = L1; C.true = L2; C.false = S.next S.code = label(L1) || C.code || label(L2) || S1.code || gen('goto' L1)

SDT scheme

S -> while ({ L1 = new Label(); L2 = new Label(); C.true = L2 ; C.false = S.next; print(“label”, L1); } C) { S_1.next = L1; print(“label”, L2) } S_1 { S.code = label(L1) || C.code || label(L2) ||S1.code || gen('goto' L1) }

Evaluation of L-attributed SDD

- L-attributed SDDs can have both synthesized attributes and inherited attributes.

Rule to be followed for evaluation:

- ***Place the semantic rule corresponding to the inherited attributes of the non-terminal before the non-terminal appears on the right hand side of the production.***

- ***Place the semantic rule corresponding to the synthesized attributes of the non-terminal at the end of the production.***

$T \to FT'$ $\quad T'.in = F.syn; \; T.syn = T'.syn$

$T' \to {*}FT'_1$ $\quad T'_1.inh = T'.inh * F.syn; \; T'.syn = T'_1.syn$

$T' \to \epsilon$ $\quad T'.syn = T'.inh$

$F \to digit$ $\quad F.syn = digit.lexval$

SDT

$T \to F \; \{ T'.in = F.syn ;\} \; T' \; \{T.syn = T'.syn\}$

$T' \to {*}F \; \{ T'_1.inh = T'.inh * F.syn; \} \; T'_1 \; \{T'.syn = T'_1.syn;\}$

$T' \to \epsilon \quad \{ T'.syn = T'.inh; \}$

$F \to digit \quad \{ F.syn = digit.lexval; \}$

***Implementation*:**

Conversion of L-attributed SDD to L-attributed SDT during Top down parsing(LL parsing)

A parser stack holds the following records

Synthesize record

holds synthesized attributes for non-terminals and also holds action to place copy of synthesized attributes in records down the stack

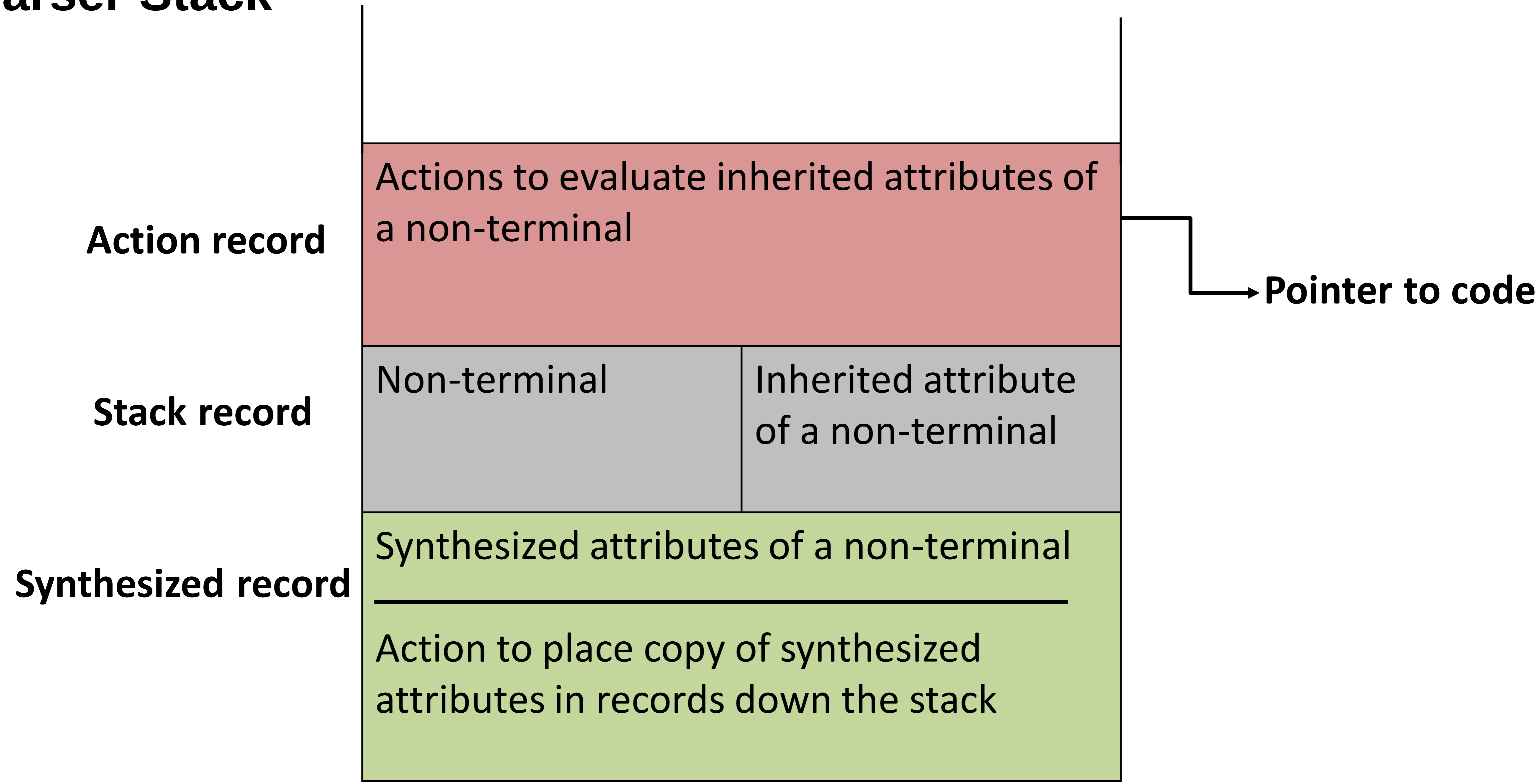
Stack record

holds the inherited attributes for non-terminals

Action record

holds actions to be executed for non-terminals

Parser Stack



Convert the following SDD to SDT scheme and show the implementation during LL parsing

$T \to FT'$	$\{ T'.in = F.syn; \ \ T.syn = T'.syn ; \}$
$T' \to {*}FT'_1$	$\{ T'_1.inh = T'.inh * F.syn; \ \ T'.syn = T'_1.syn; \}$
$T' \to \epsilon$	$\{ T'.syn = T'.inh; \}$
$F \to digit$	$\{ F.syn = digit.lexval; \}$

$T \text{ -> } FT'$ $\{ T'.in = F.syn; T.syn = T'.syn; \}$

$T' \text{ ->} *FT'_1$ $\{ T'_1.in = T'.in * F.syn;$ $T'.syn = T'_1.syn; \}$

$T' \text{ -> } \epsilon$ $\{ T'.syn = T'.in; \}$

$F \text{ -> } digit$ $\{ F.syn = digit.lexval; \}$

SDT

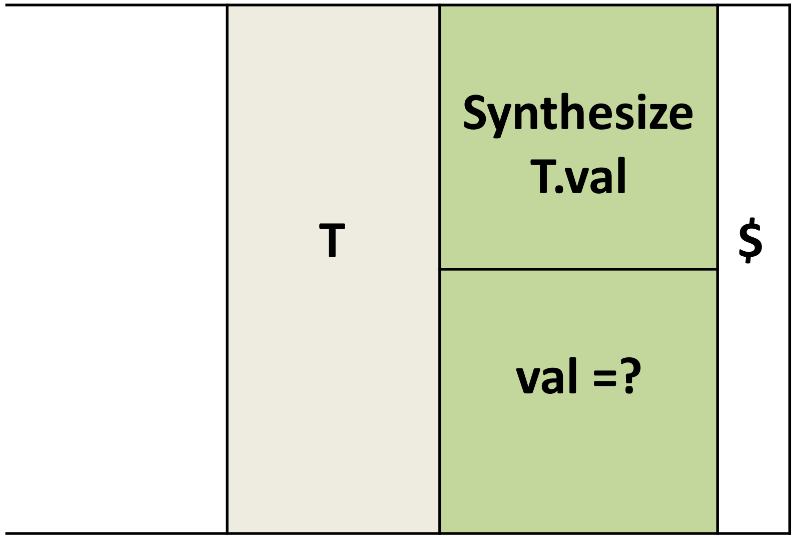
$T \text{ -> } F \; \{ T'.in = F.syn ; \} \; T' \; \{T.syn = T'.syn\}$

$T' \text{ -> } *F \; \{ T'_1.in = T'.in * F.syn; \} \; T'_1 \; \{T'.syn = T'_1.syn; \}$

$T' \text{ -> } \epsilon$ $\{ T'.syn = T'.in; \}$

$F \text{ -> } digit$ $\{ F.syn = digit.lexval; \}$

Stack	Input Buffer	Action
T$	3 * 4$	M[T, digit] T -> FT’



Stack	Input Buffer	Action
FT’ $	3 * 4$	M[F, digit] F -> digit

	F	**Synthesize F.val** **val = ?**	**Actions for T’**	**T’** **in =?**	**Synthesize T’.val** **val = ?**	**Synthesize T.val** **val = ?**	**$**

Stack	**Input Buffer**	**Action**
digit T'$	3 * 4$	M[digit, digit] ***match***

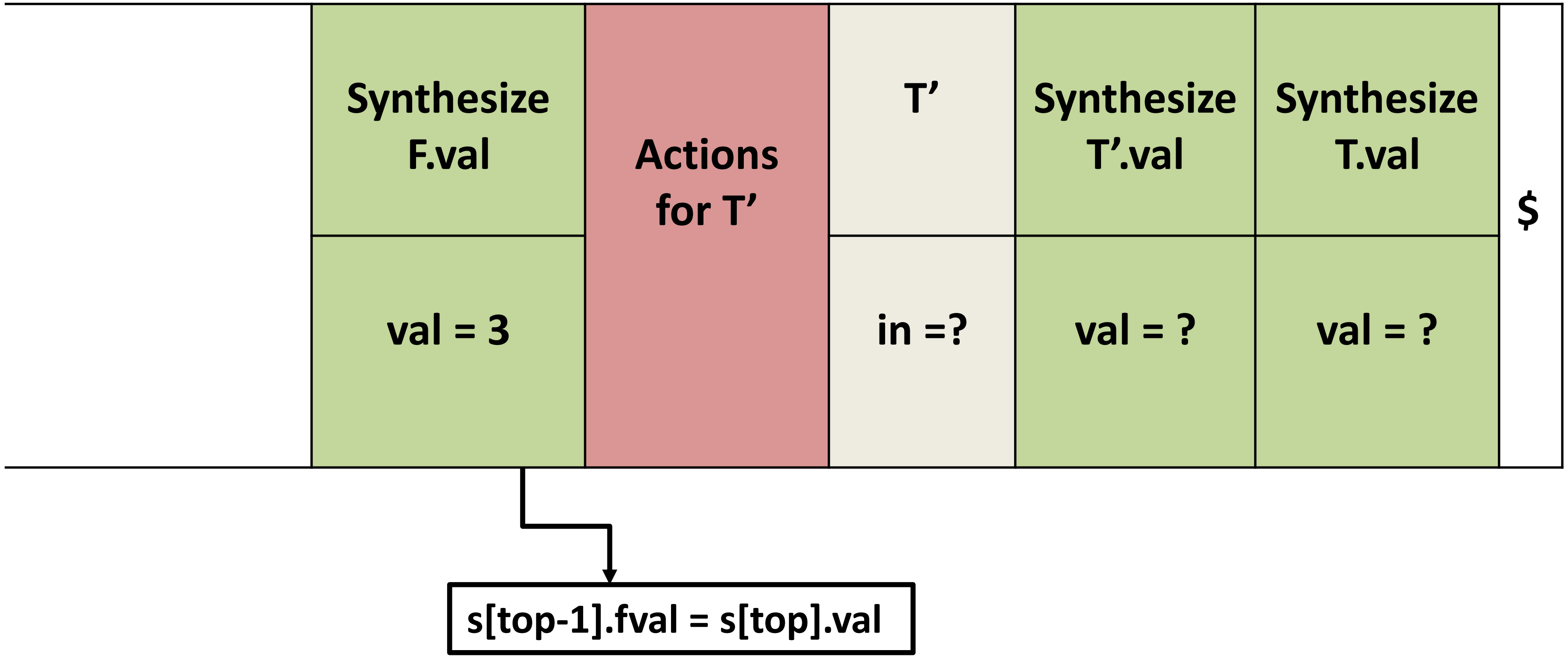
		Synthesize digit.lexval	**Synthesize F.val**		**T'**	**Synthesize T'.val**	**Synthesize T.val**
digit				**Actions for T'**			**$**
		lexval = 3	**val= ?**		**in =?**	**val = ?**	**val = ?**

Stack	Input Buffer	Action
T’$	*4$	

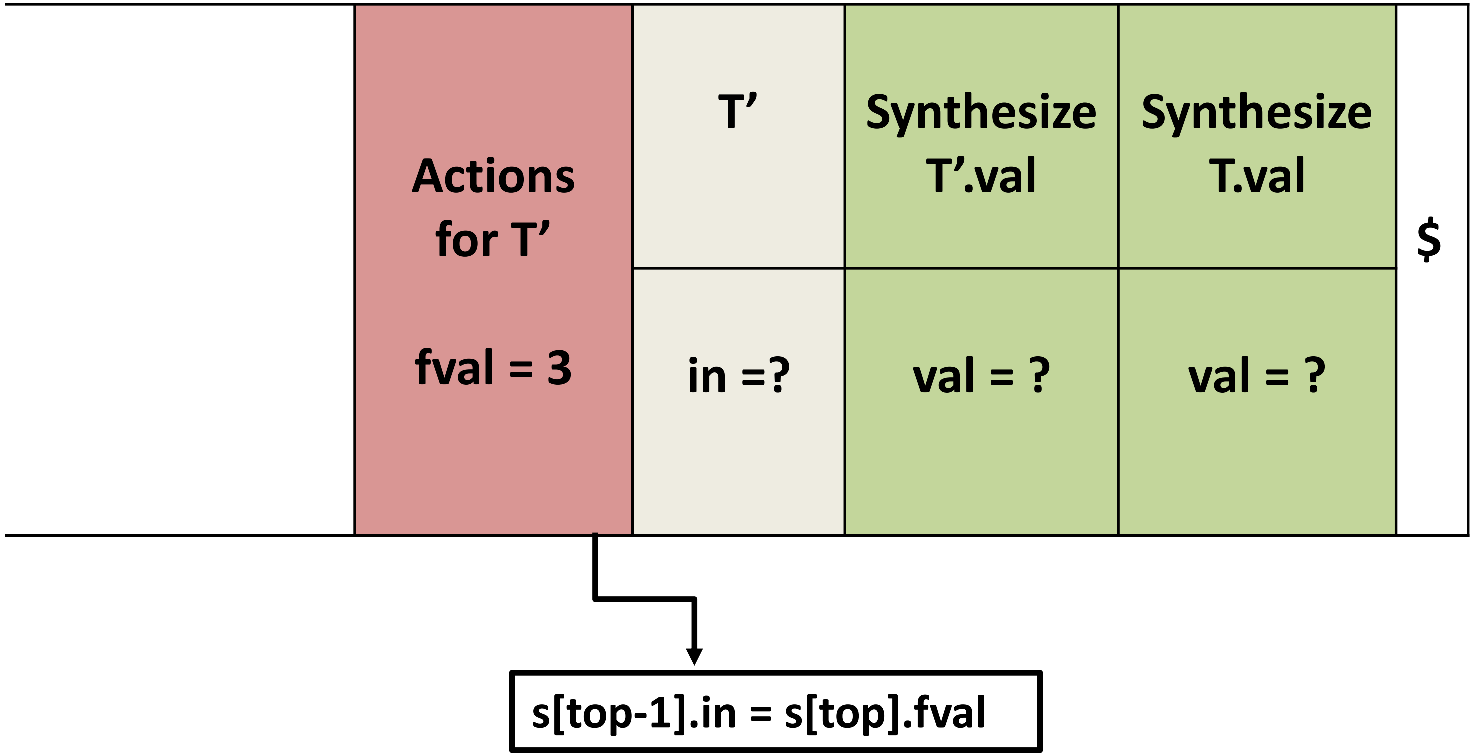
Synthesize digit.lexval	Synthesize F.val	Actions for T’	T’	Synthesize T’.val	Synthesize T.val	$
lexval = 3	val= ?		in =?	val = ?	val = ?	

s[top-1].val = s[top].lexval

Stack	Input Buffer	Action
T'$	* 4 $	



Stack	Input Buffer	Action
T’$	* 4 $	



Stack

T’$

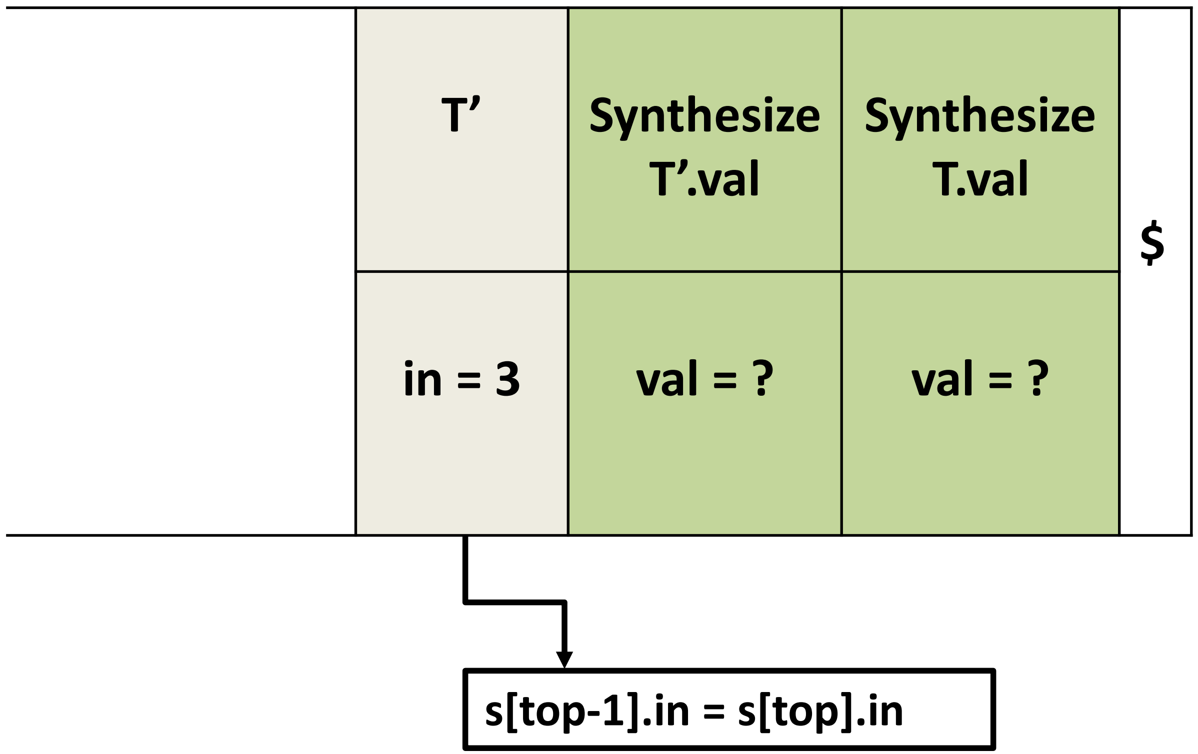
Input Buffer

** 4* $

Action

M[T’, *]

T’ -> $*FT'_1$



Stack	Input Buffer	Action
$*FT'_1\$$	$*\ 4\ \$$	M[*, *]

*	F	Synthesize F.val	Action for T'_1	T'_1	Synthesize T'_1	Synthesize T'.val	Synthesize T.val	$
		val = ?		in = ?	val =?	val = ? in = 3	val = ?	

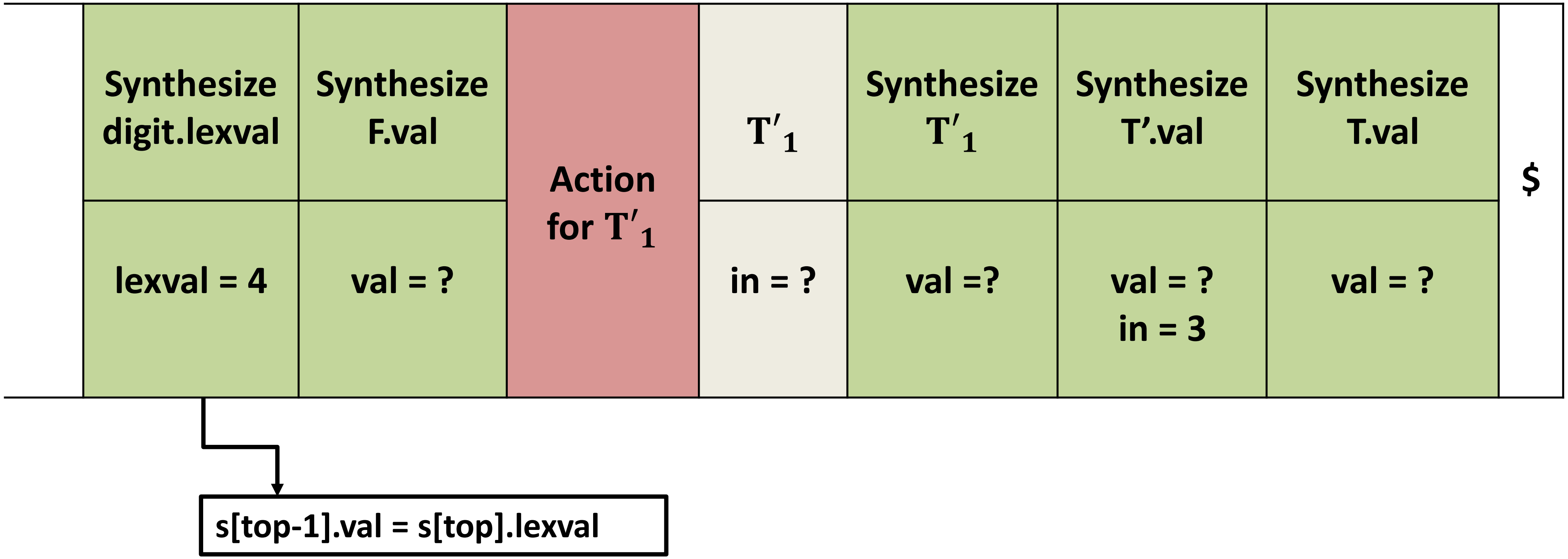
Stack	Input Buffer	Action
$FT'_1\$$	*4* $	M[F, digit] F -> digit

<table>
<tr><td rowspan="2">F</td><td>Synthesize F.val</td><td rowspan="2">Action for T'_1</td><td>T'_1</td><td>Synthesize T'_1</td><td>Synthesize T'.val</td><td>Synthesize T.val</td><td rowspan="2">$</td></tr>
<tr><td>val = ?</td><td>val= ?</td><td>val =?</td><td>val = ?
in = 3</td><td>val = ?</td></tr>
</table>

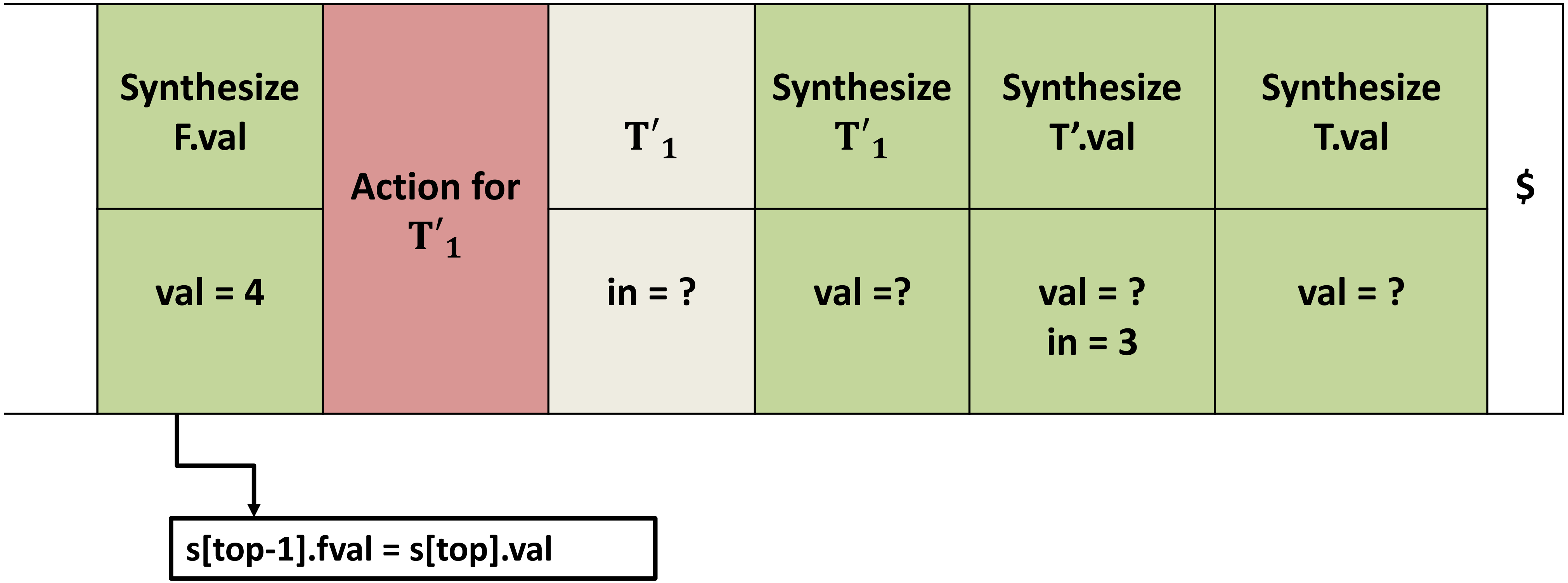
Stack	**Input Buffer**	**Action**
digit $T'_1\$$	*4 $*	M[digit, digit] match

digit	**Synthesize digit.lexval**	**Synthesize F.val**	**Action for T'_1**	**T'_1**	**Synthesize T'_1**	**Synthesize T'.val**	**Synthesize T.val**	**$**
	lexval = 4	**val = ?**		**in= ?**	**val =?**	**val = ? in = 3**	**val = ?**	

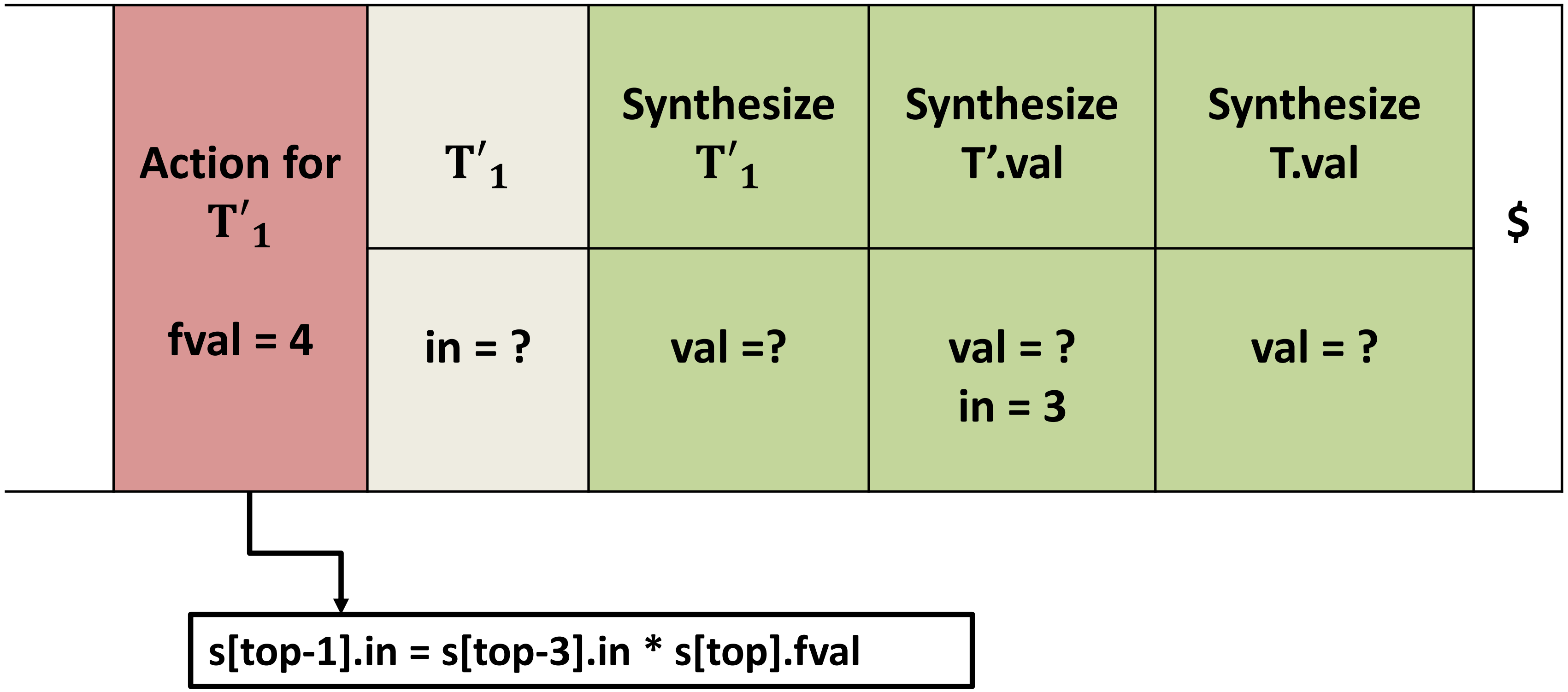
Stack	Input Buffer	Action
$T'_1\$$	$\$$	



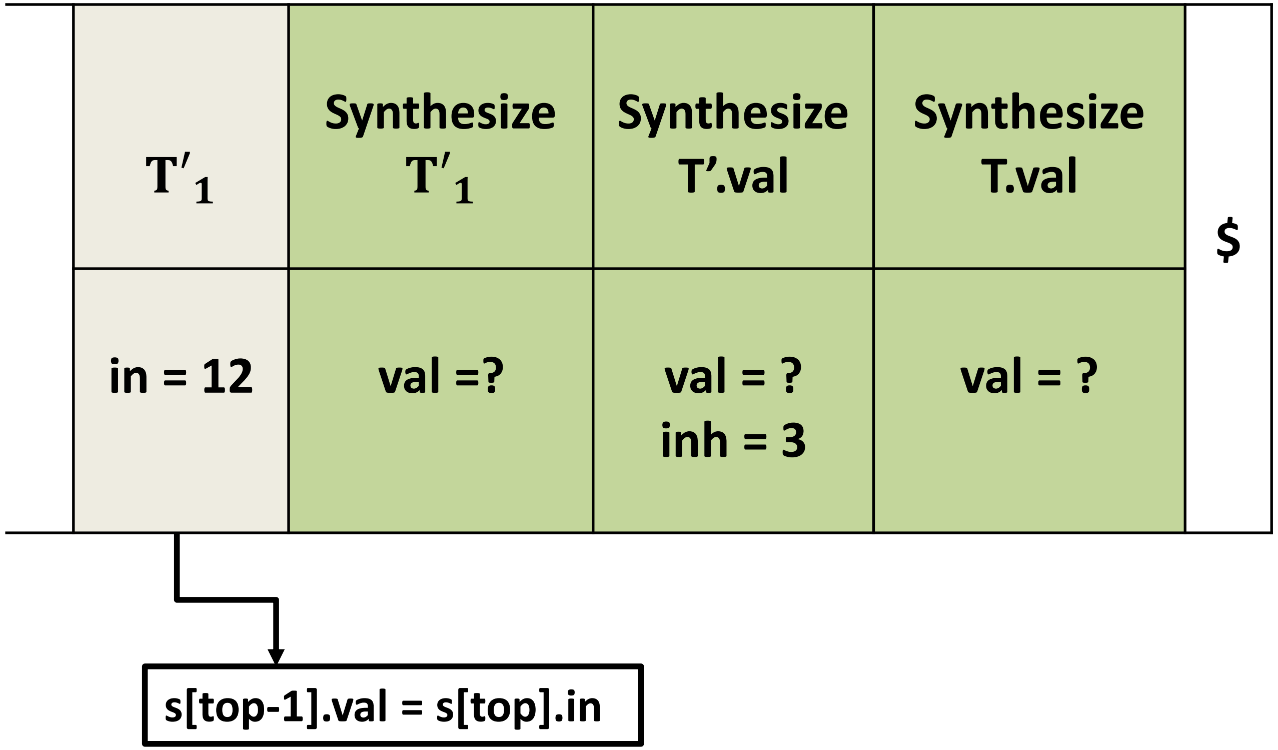
Stack	Input Buffer	Action
$T'_1\$$	$\$$	



Stack	Input Buffer	Action
$T'_1\$$	$\$$	

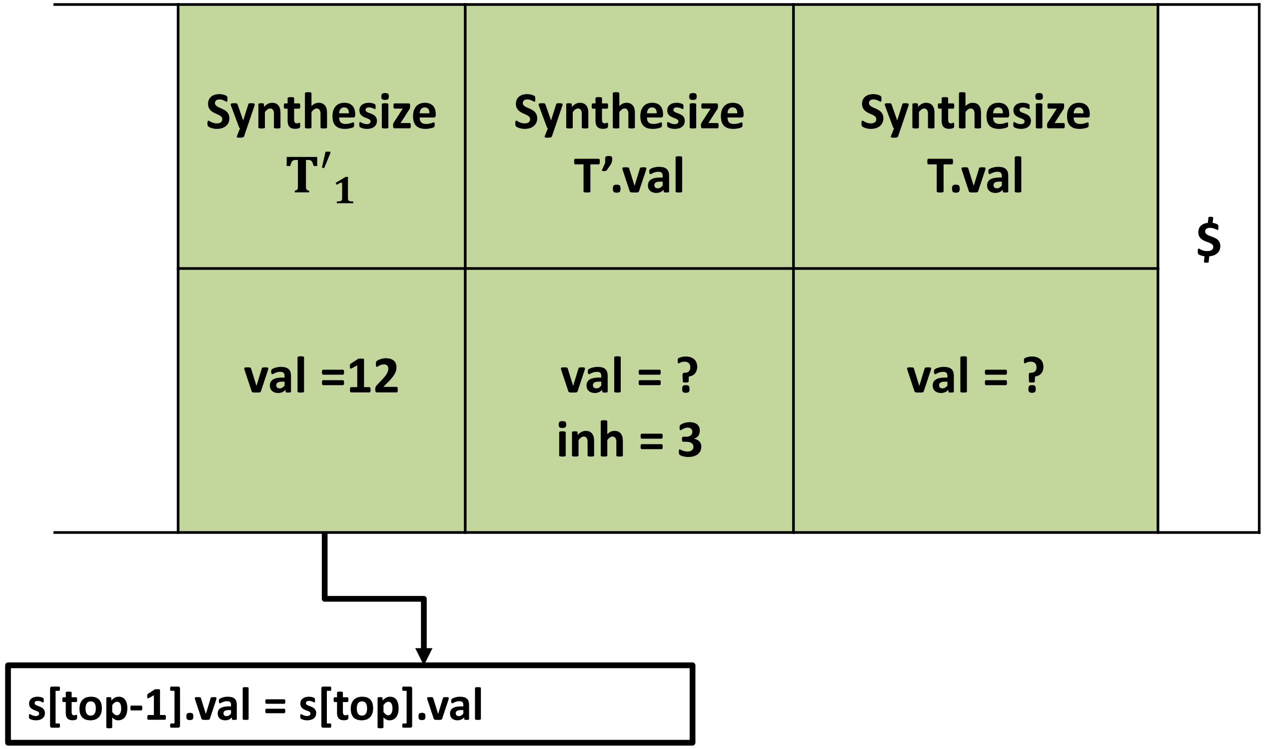


Stack	Input Buffer	Action
$T'_1\$$	$\$$	$M[T'_1, \$]$ **T' -> ϵ**

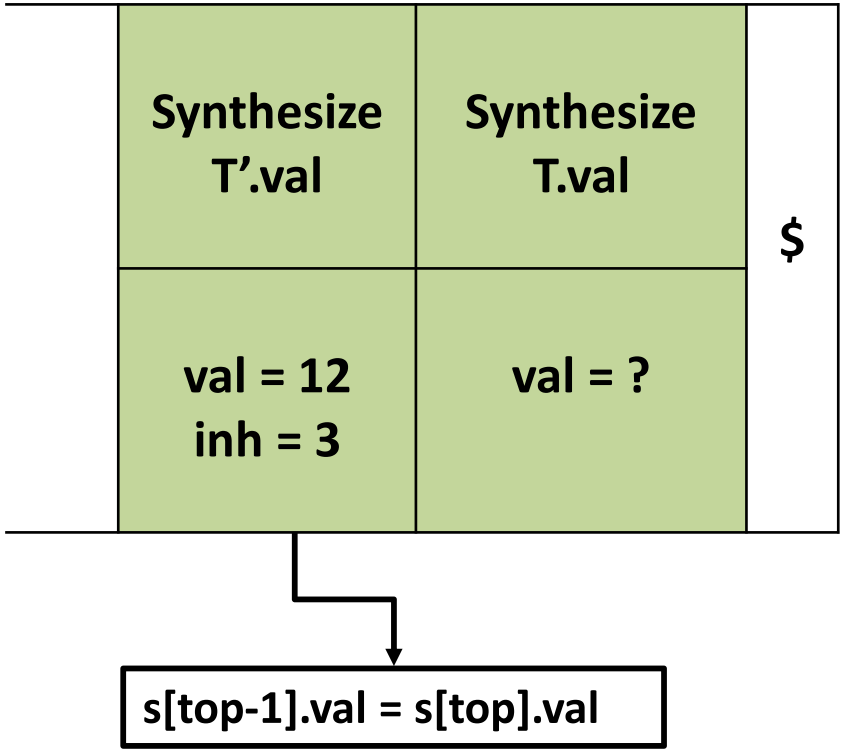


Stack | **Input Buffer** | **Action**

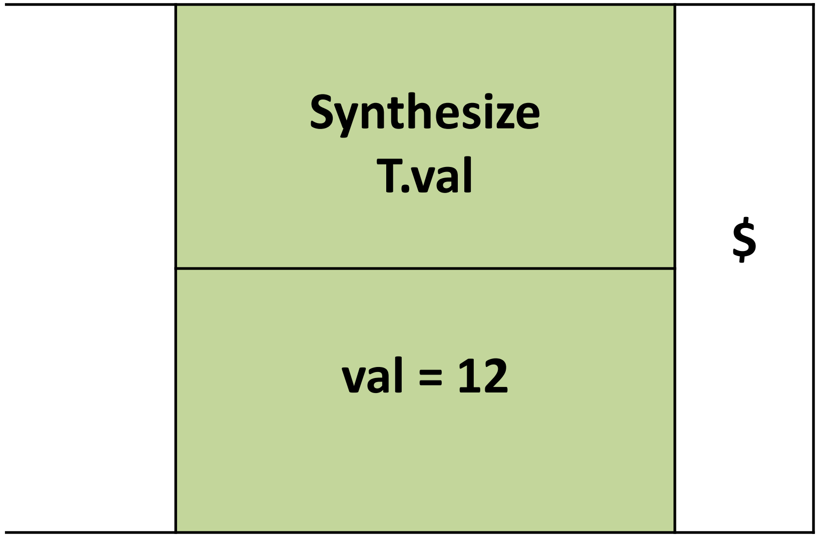
Stack	Input Buffer	Action
\$	\$	



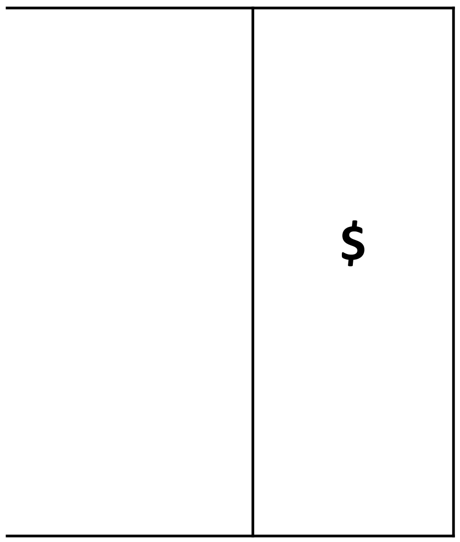
Stack	Input Buffer	Action
$	$	



Stack	Input Buffer	Action
$	$	



Stack	**Input Buffer**	**Action**
$	$	



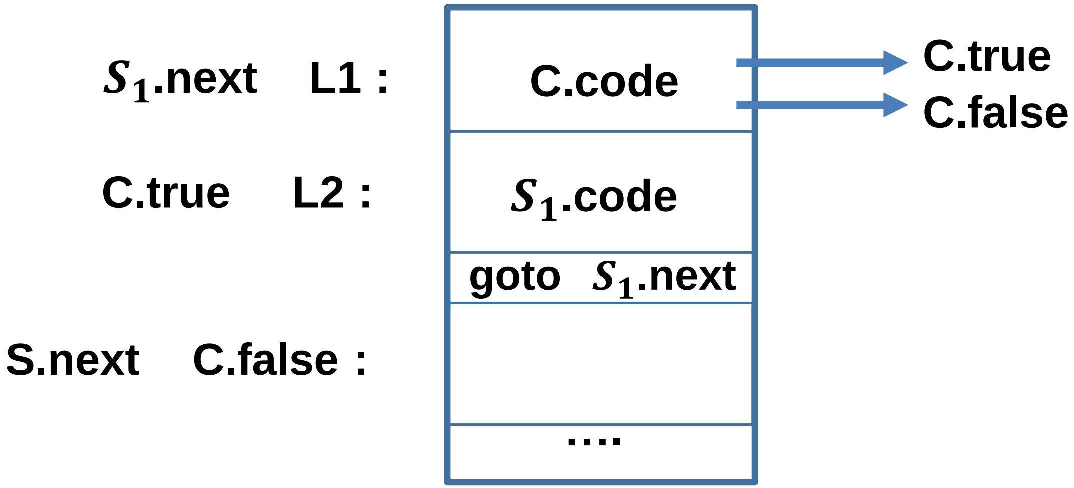
Design the L-attributed SDD to generate intermediate code and convert it to SDT for the following programming constructs

1) S -> while (C) S

2) S -> if(C) S

SDD to generate Intermediate code for ***while statement***

Productions	**Semantic Rules**
S -> while (C) S_1	L1 = new Label(); L2 = new Label(); S1.next = L1; C.true = L2; C.false = S.next S.code = label(L1) || C.code || label(L2) || S1.code || gen('goto' L1)



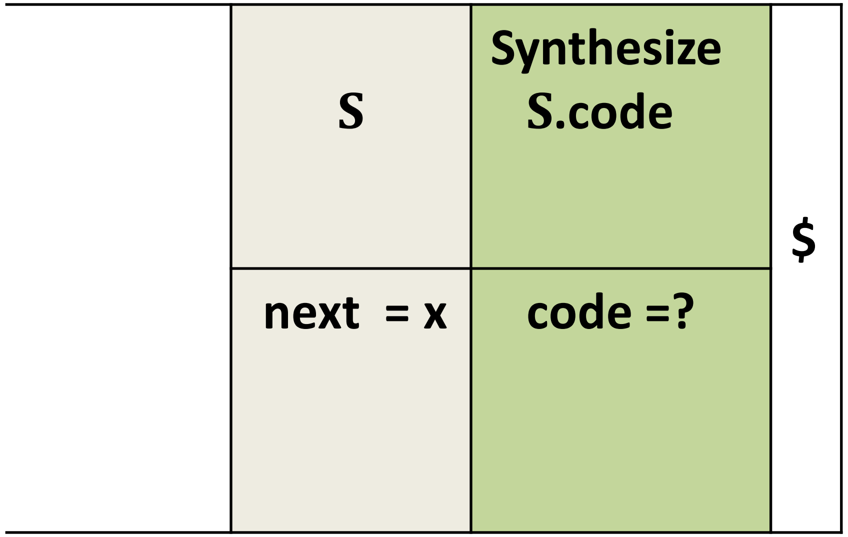
SDD to generate Intermediate code for ***while statement***

<table>
<tr><th>Productions</th><th>Semantic Rules</th></tr>
<tr><td>S -> while (C) S_1</td><td>L1 = new Label(); L2 = new Label();
S1.next = L1; C.true = L2;   C.false = S.next
S.code = label(L1) || C.code ||
label(L2) || S1.code || gen('goto' L1)</td></tr>
</table>

SDT scheme

S -> while ({ L1 = new Label(); L2 = new Label(); C.true = L2 C.false = S.next } C) { S_1.next = L1 } S_1 { S.code = label(L1) || C.code || label(L2) ||S1.code || gen('goto' L1) }

Stack	Input Buffer	Action
S$	*while(C)S$*	——



Stack	Input Buffer	Action
S$	*while(C)S$*	S -> while(C) S_1

Pop S

Push while(C) S_1

Place inherited attribute next of S in the synthesized record of S before popping

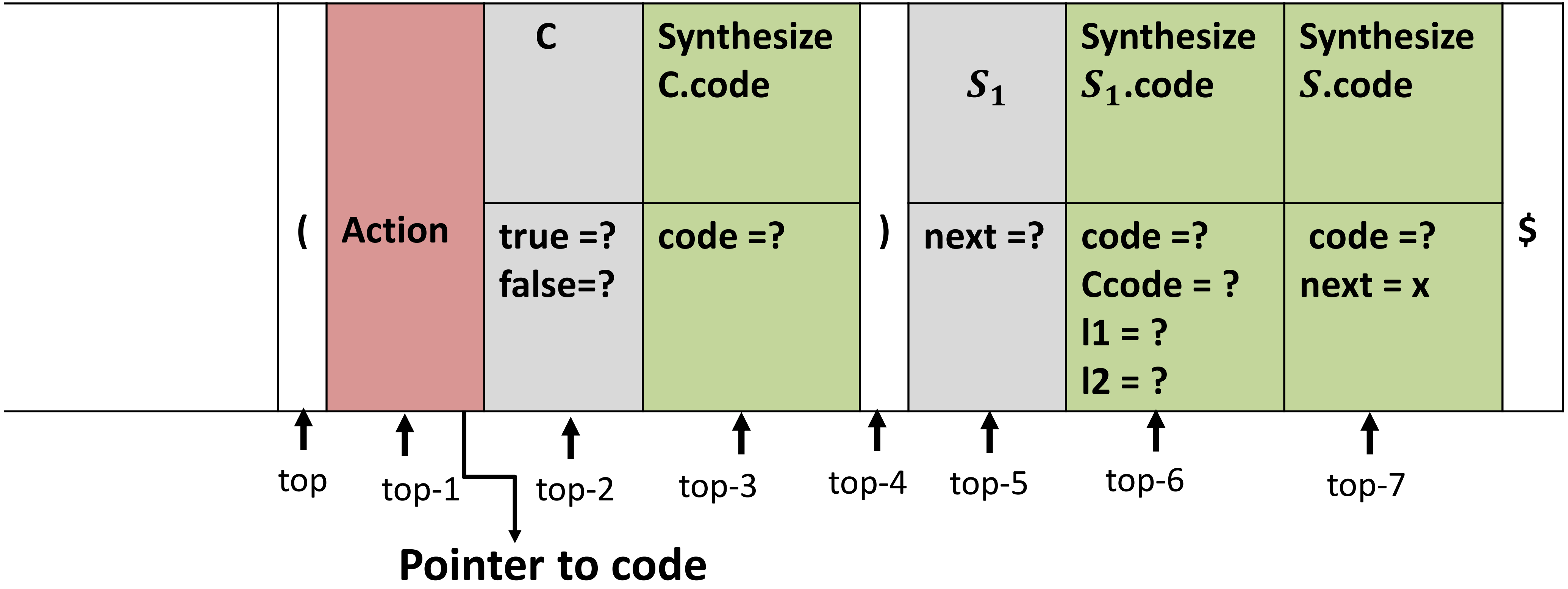
S	Synthesize S.code	$
next = x	code =? next = x	

Stack	**Input Buffer**	**Action**
while(C) S1$	*while(C)S$*	M[while, while] **match**

	while	**(**	**Action for C**	**C**	**Synthesize C.code**	**)**	S_1	**Synthesize S_1.code**	**Synthesize S.code**	**$**
				true =? false =?	**code =?**		**next =?**	**code =? Ccode = ? l1 = ? l2 = ?**	**code =? next = x**	

Pointer to code

Stack	Input Buffer	Action
(C) S1$	*(C)S$*	M[(, ( ]
		match



Stack

C) S1$

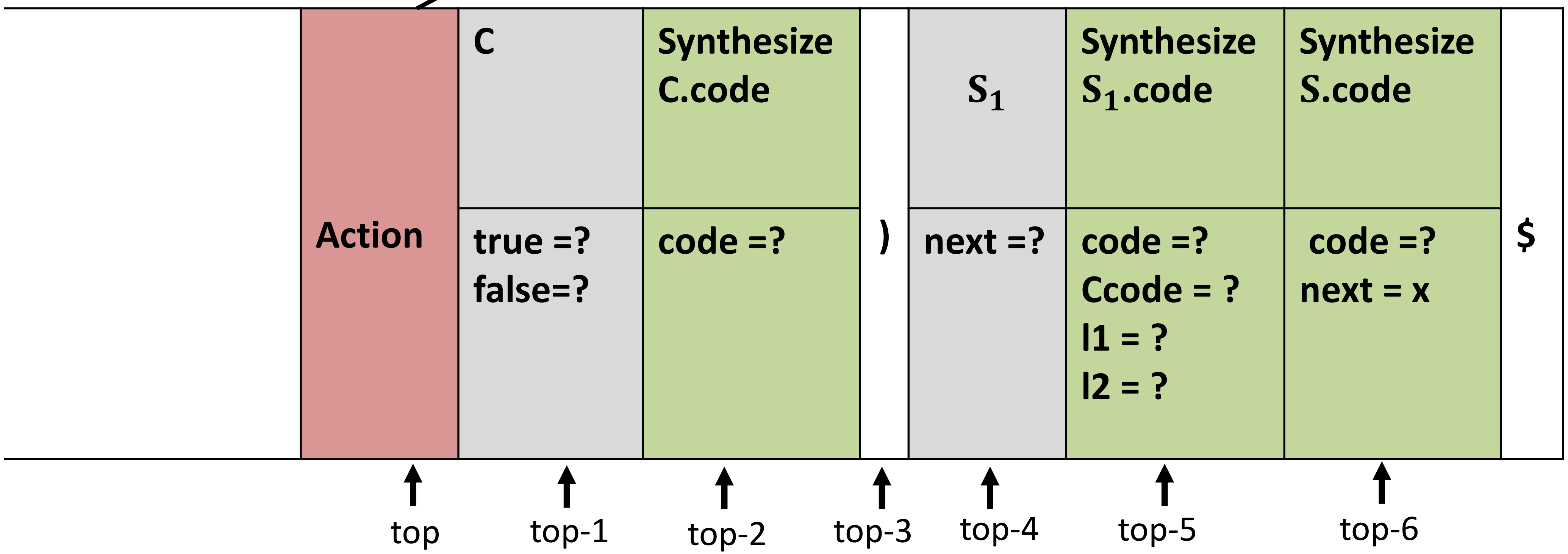
Input Buffer

C) S$

```
L1 = new Label ();
L2 = new Label ();
s[top -1 ] .true = L2
s[top -1 ] .false = s[top-6].next;
s[top-4].next = L1;
s[top – 5].l1 = L1;
s[top – 5].l2 = L2;
```

Action

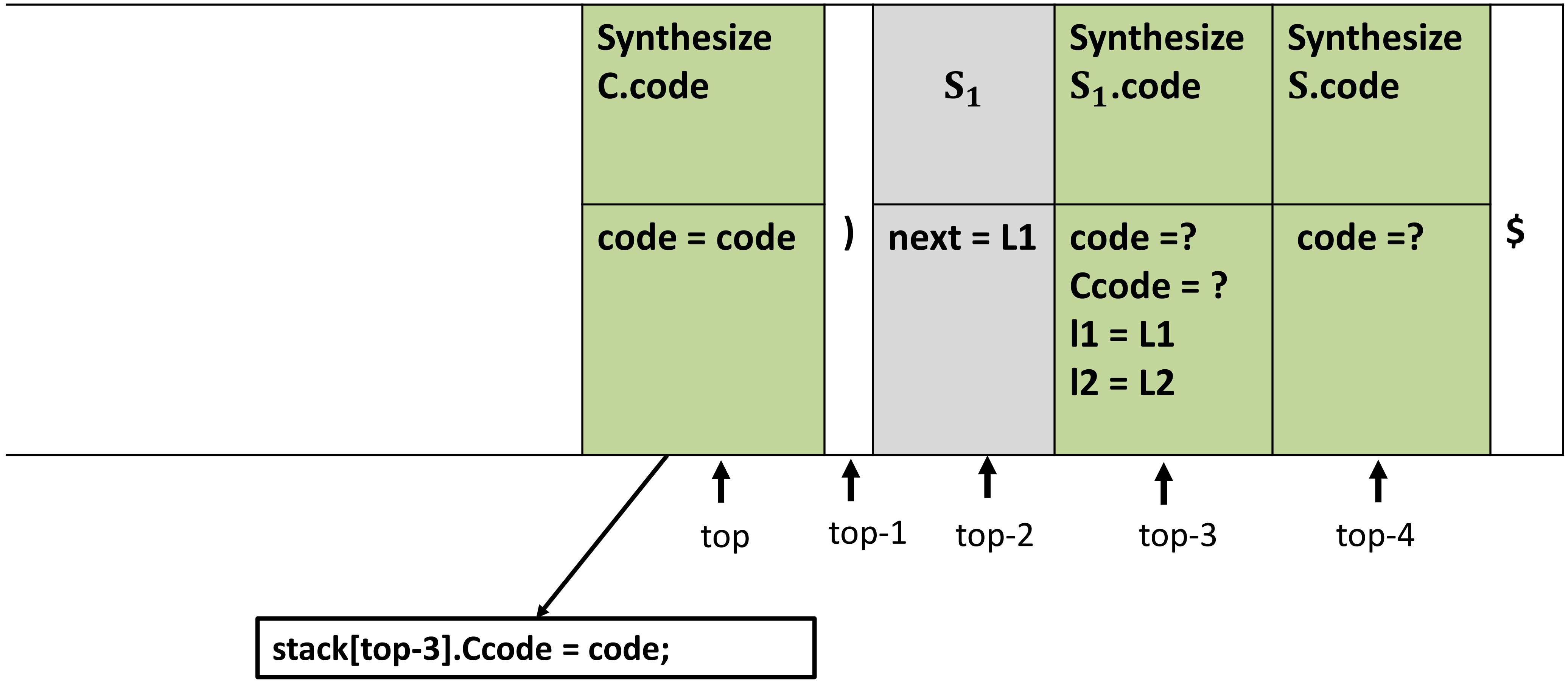
Execute corresponding Actions



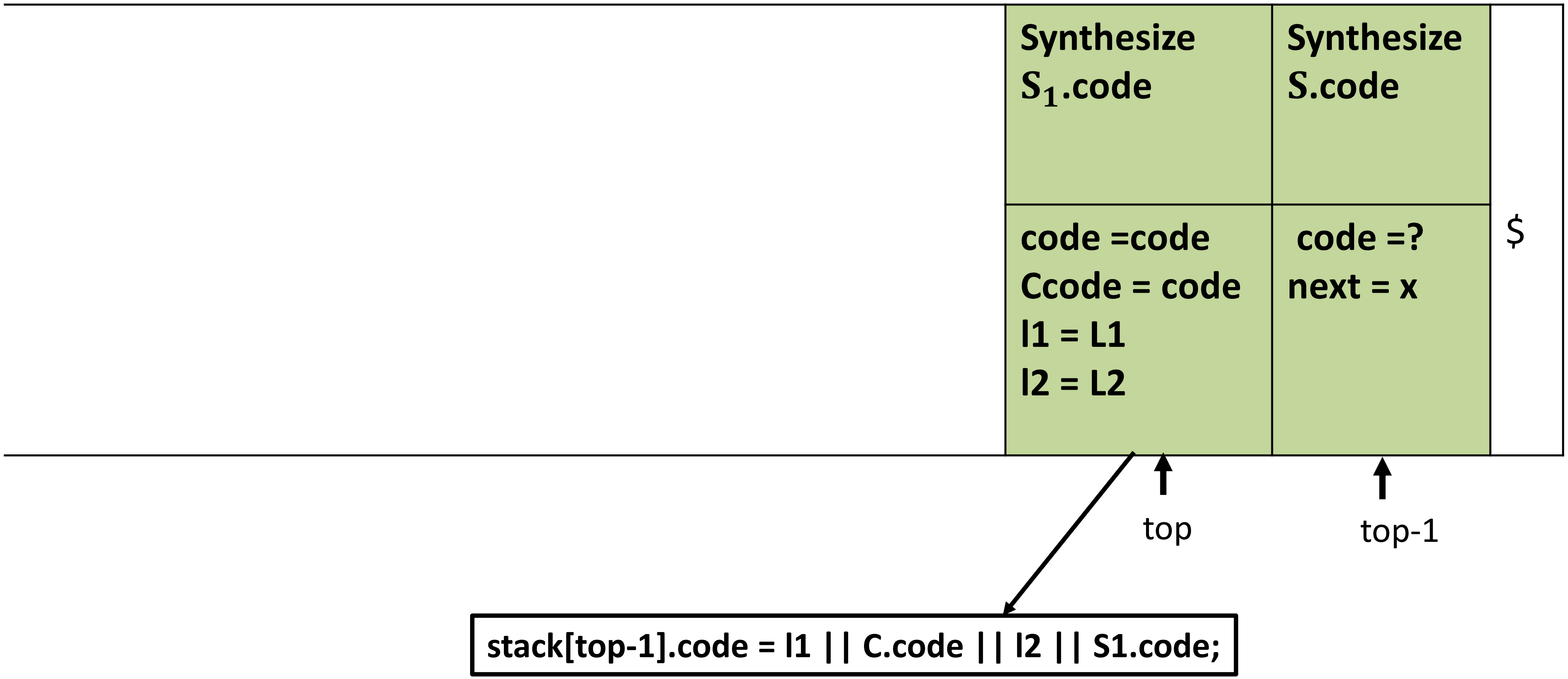
Stack	Input Buffer	Action
C) S1$	*C) S$*	M[C, C] **match** Pop C from the stack

C	**Synthesize C.code**		$\boldsymbol{S_1}$	**Synthesize $\boldsymbol{S_1}$.code**	**Synthesize S.code**	
true = L2 **false = x**	**code =?**	**)**	**next =L1**	**code =?** **Ccode = ?** **l1 = L1** **l2 = L2**	**code =?**	**$**
top	top-1	top-2	top-3	top-4	top-5	

Stack	Input Buffer	Action
) S1$	) S$	M[ ),)] **match**

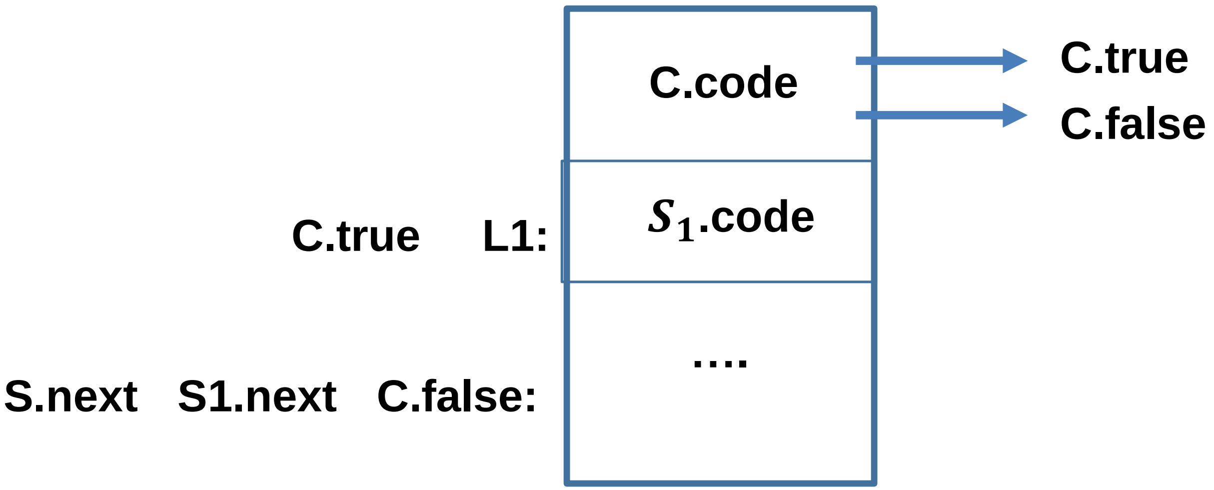


Stack	Input Buffer	Action
\$	\$	**Accepted**



SDD to generate Intermediate code for ***if statement***

Productions	Semantic Rules
S -> if (C) S_1	L1 = new Label(); C.true = L1 C.false = S.next S1.next = S.next S.code = C.code || label(L1) || S1.code



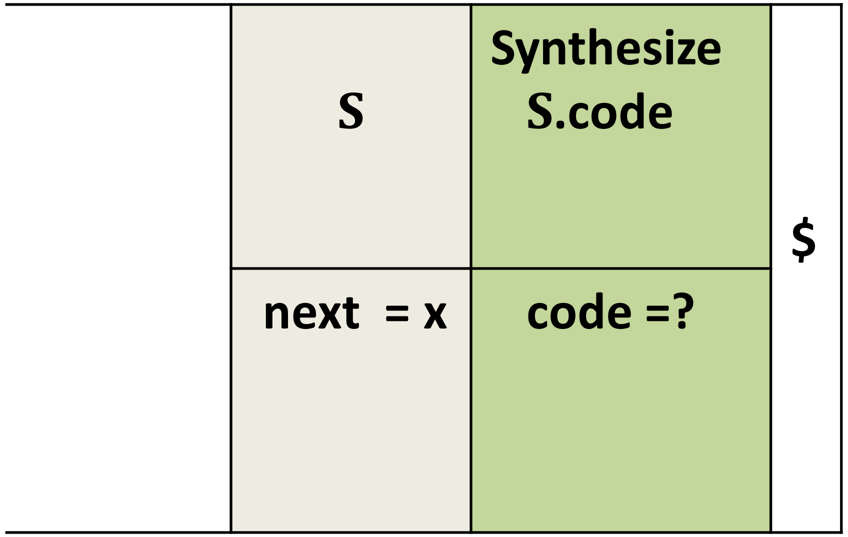
SDD to generate Intermediate code for ***if statement***

Productions	**Semantic Rules**
S -> if(C) S_1	L1 = new Label(); C.true = L1; C.false = S.next S1.next = S.next S.code = C.code || label(L1) || S1.code

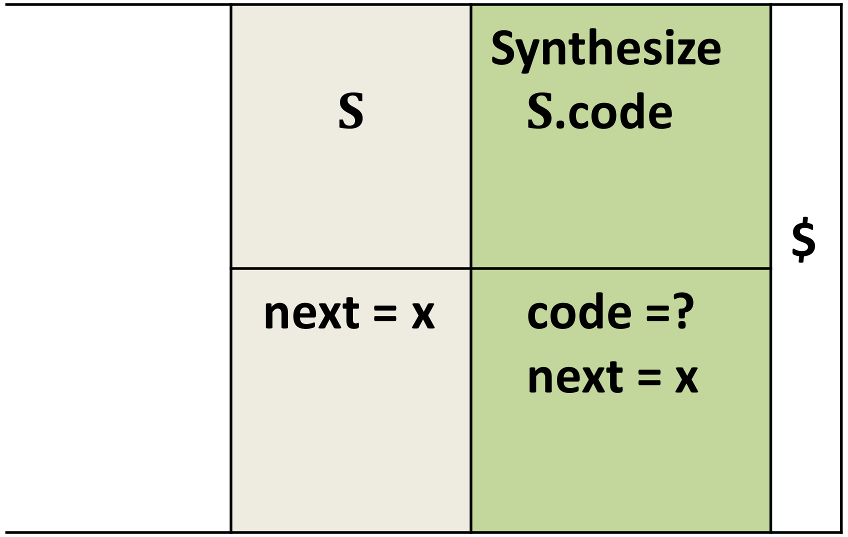
SDT scheme

$$\textbf{S -> if}\left(\left\{\begin{array}{l}\textbf{L1 = new Label();}\\ \textbf{C.true = L2}\\ \textbf{C.false = S.next}\end{array}\right\}\textbf{C}\right)\left\{S_1\textbf{.next = S.next}\right\}S_1\left\{\begin{array}{l}\textbf{S.code = C.code ||}\\ \textbf{label(L1) ||S1.code}\end{array}\right\}$$

Stack	Input Buffer	Action
$S\$$	*if(C)S$*	———



Stack	**Input Buffer**	**Action**
S$	*if(C)S$*	M[S, if] S -> if(C) S_1

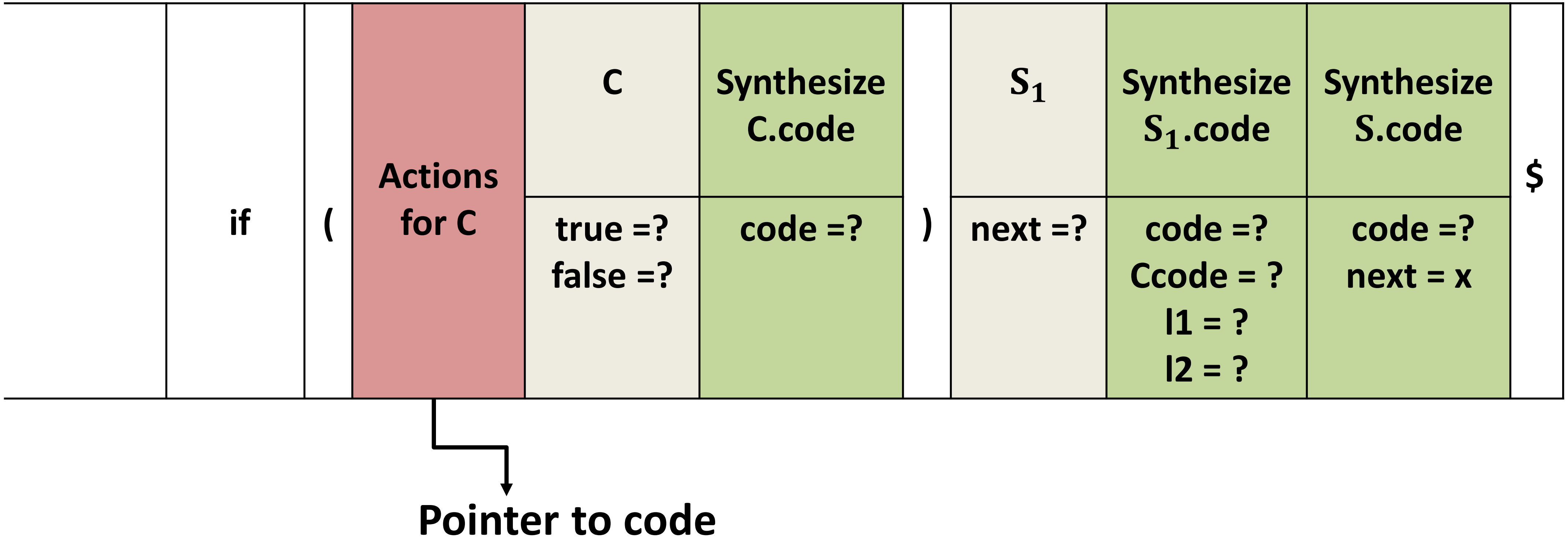


Pop S

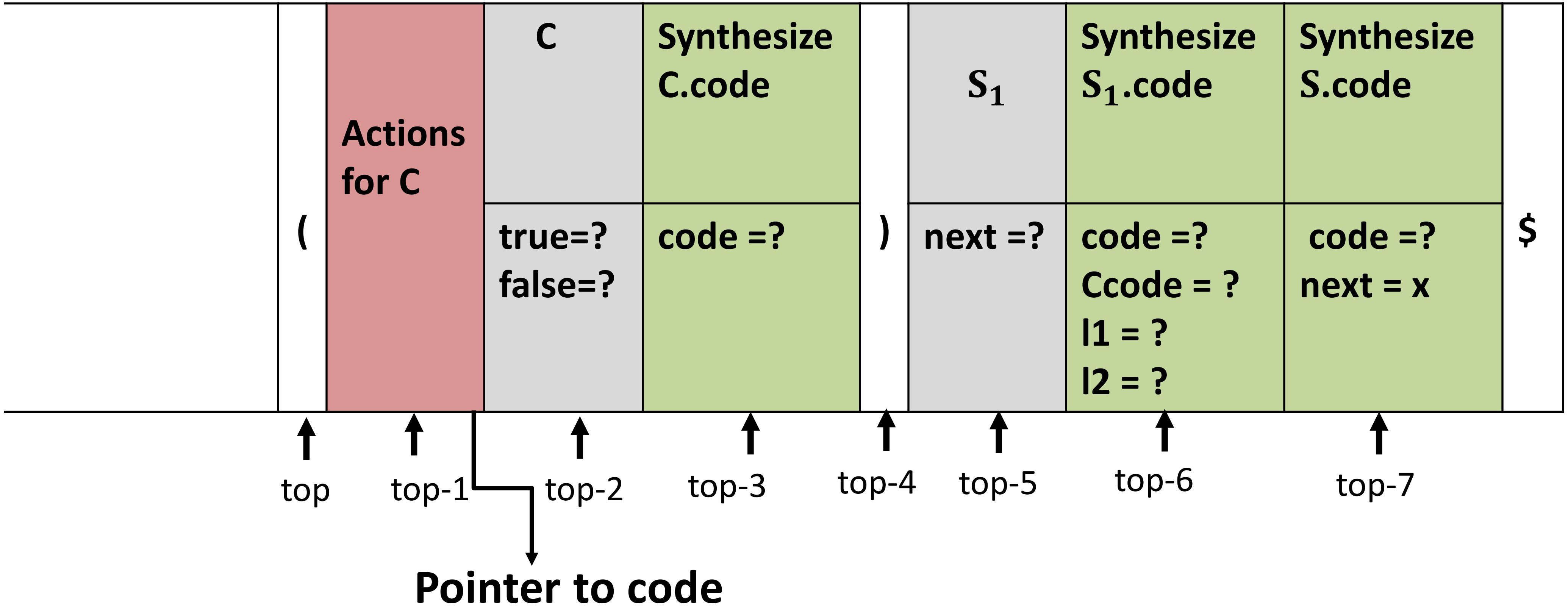
Push if(C) S_1

Place inherited attribute next of S in the synthesized record of S before popping

Stack	Input Buffer	Action
if(C) S1$	*if(C)S$*	M[if, if]
		match



Stack	Input Buffer	Action
(C) S1$	*(C)S*$	M[(, ( ]
		match



Stack

C) S1$

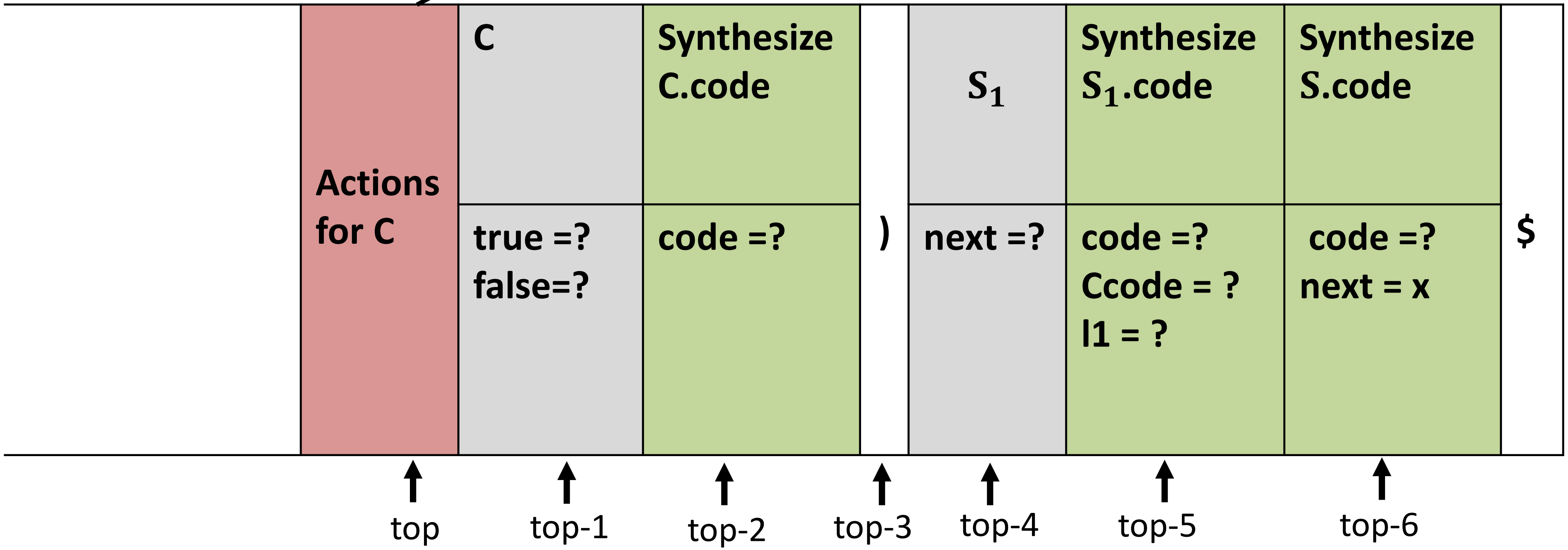
Input Buffer

C) S$

```
L1 = new Label ();
s[top -1 ] .true = L1
s[top -1 ] .false = s[top-6].next;
s[top-4].next = s[top-6].next
s[top – 5].l1 = L1
```

Action

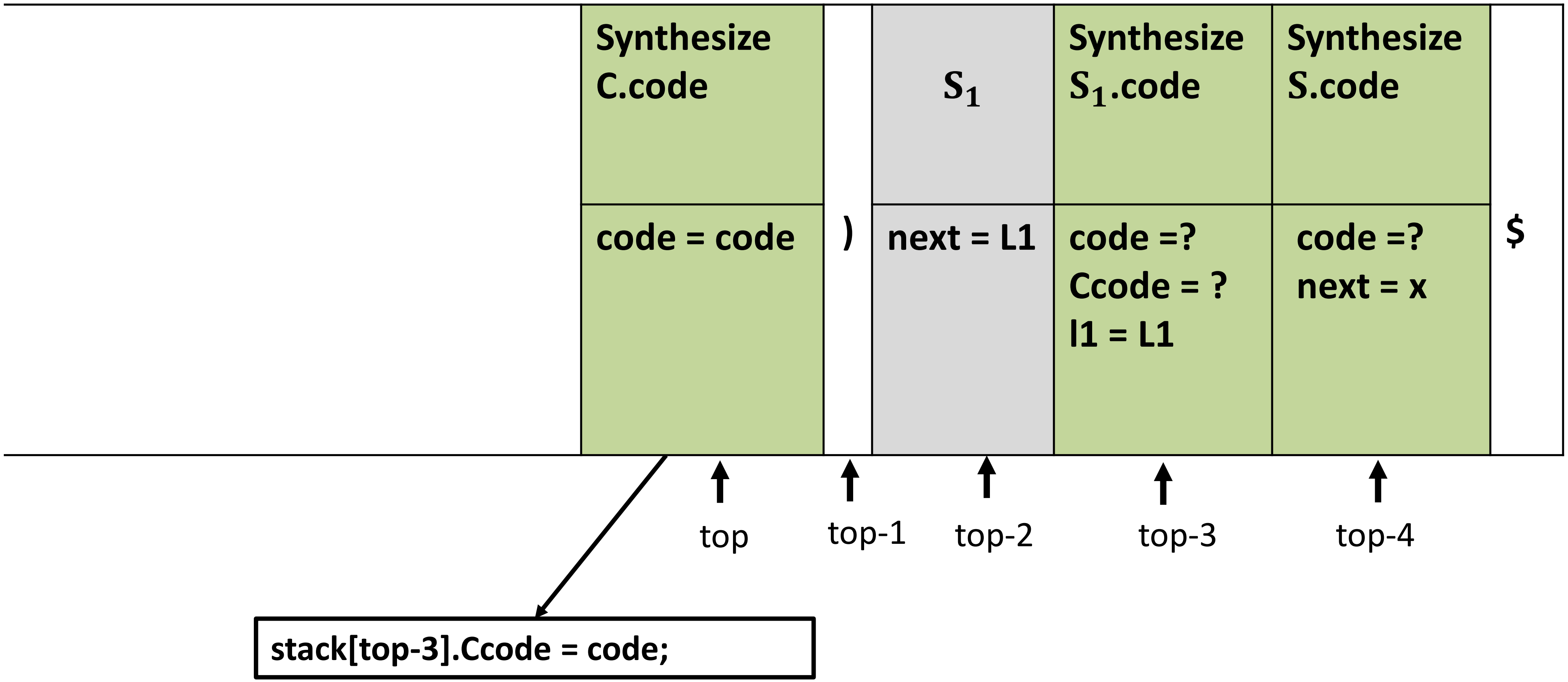
Execute corresponding actions



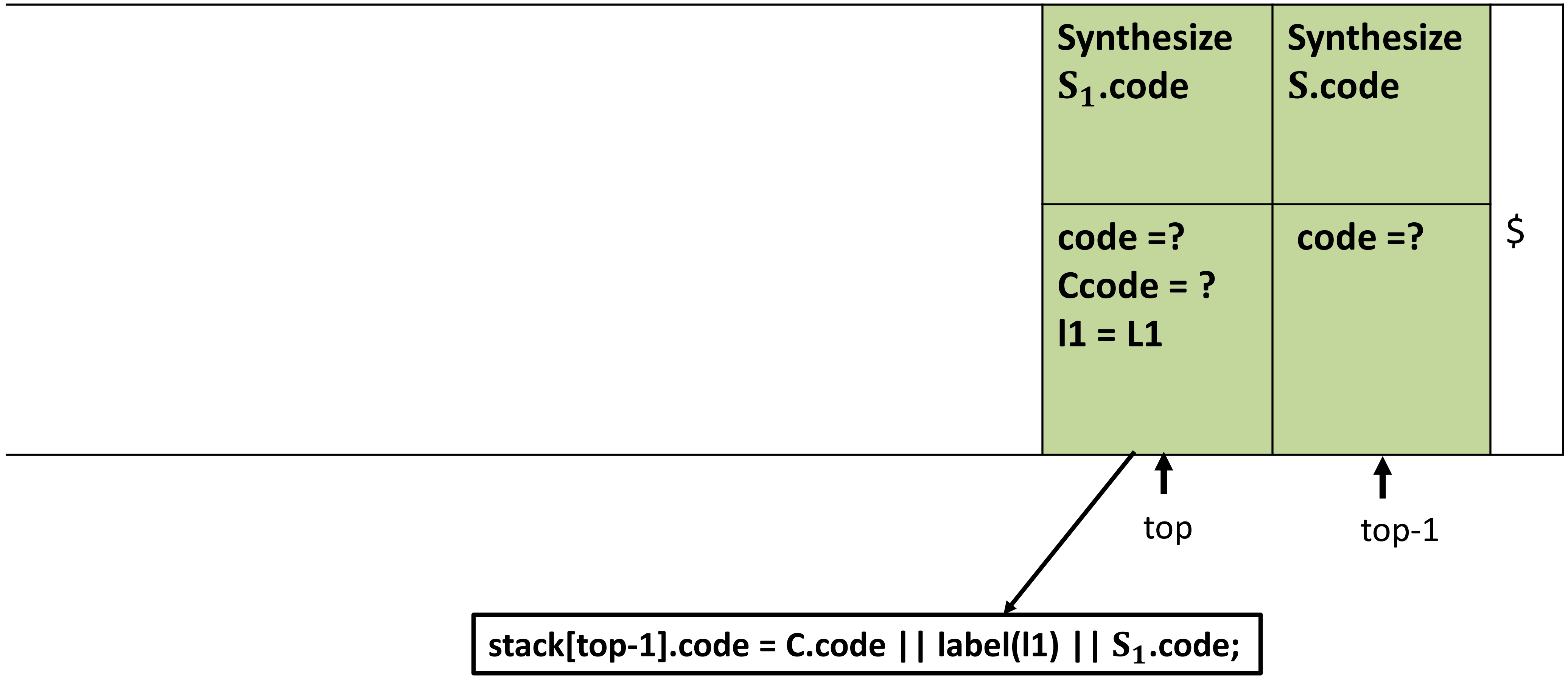
Stack	Input Buffer	Action
C) S1$	*C) S$*	M[C, C] **match** Pop C from the stack

C	**Synthesize C.code**		$\boldsymbol{S_1}$	**Synthesize $\boldsymbol{S_1}$.code**	**Synthesize S.code**	
true = L1 **false = x**	**code =?**	**)**	**next = x**	**code =?** **Ccode = ?** **l1 = L1**	**code =?** **next = x**	**$**
top	top-1	top-2	top-3	top-4	top-5	

Stack	Input Buffer	Action
) S1$	) S$	M[ ),)] **match**



Stack	Input Buffer	Action
$	$	Accepted



***Implementation*:**

Conversion of L-attributed SDT during Bottom up parsing(LR parsing)

Productions	Semantic Rules
S -> while (C) S_1	L1 = new Label(); L2 = new Label(); S1.next = L1; C.true = L2; C.false = S.next S.code = label(L1) || C.code || label(L2) || S1.code || gen('goto' L1)

SDT scheme

S -> while ({ L1 = new Label(); L2 = new Label(); C.true = L2; C.false = S.next } C) { S_1.next = L1 } S_1 { S.code = label(L1) || C.code || label(L2) ||S1.code || gen('goto' L1) }

1. Start with the SDT shown below

$$\textbf{S -> while}\left(\left\{\begin{array}{l}\textbf{L1 = new Label();}\\ \textbf{L2 = new Label();}\\ \textbf{C.true = L2}\\ \textbf{C.false = S.next}\end{array}\right\}\textbf{C}\right)\left\{S_1\textbf{.next = L1}\right\}S_1\left\{\begin{array}{l}\textbf{S.code = label(L1) ||}\\ \textbf{C.code || label(L2)}\\ \textbf{||S1.code}\end{array}\right\}$$

2. Introduce into a grammar a marker Non-terminal in place of embedded actions. Each distinct marker ‘M’ will have a production M -> Є.

3. Modify the action ‘a’ if the marker nonterminal ‘M’ replaces it in some production A -> α { a} β and associate an action ‘a’ with the production M -> Є that

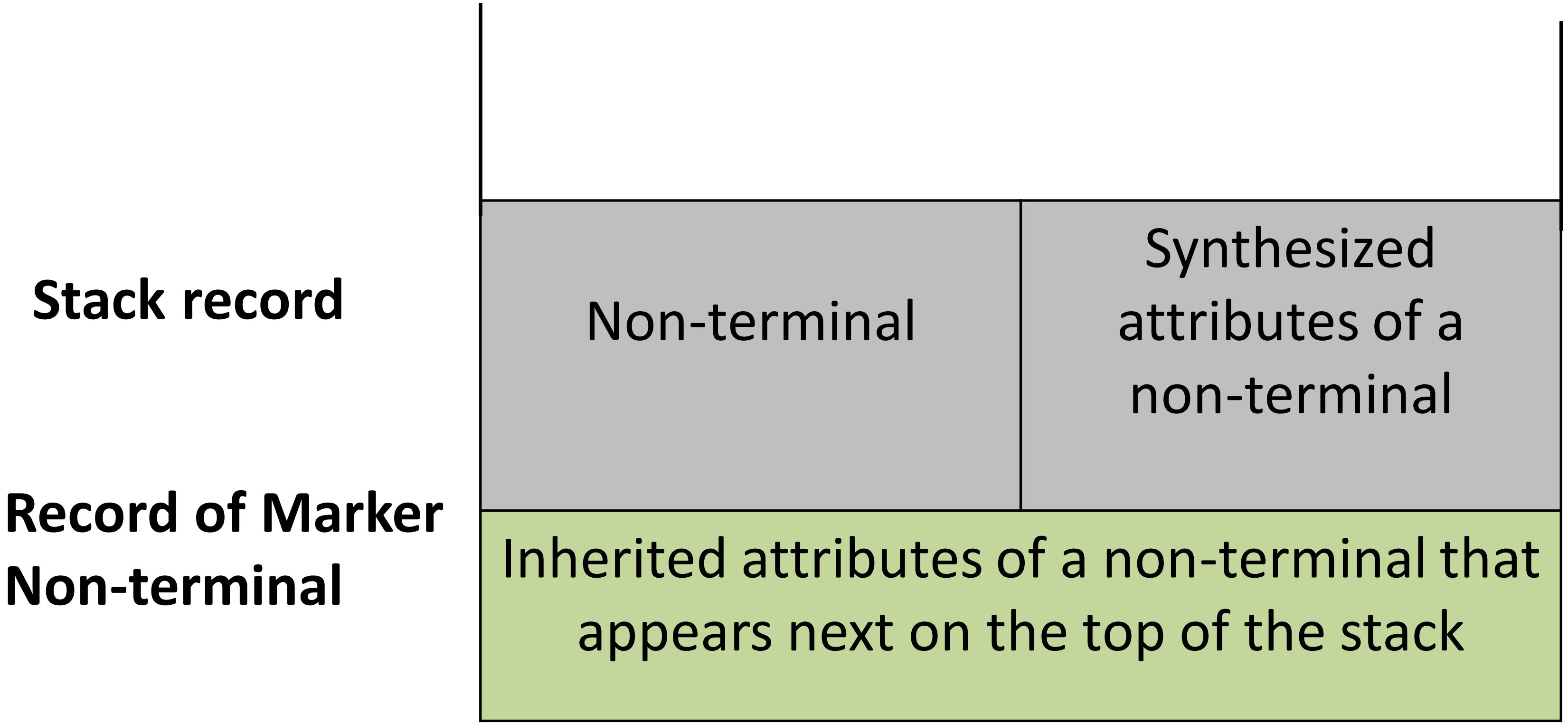
(a) Copies any attributes of A or symbols of α that action a needs.

(b) Computes attributes and makes those attributes be synthesized attributes of M.

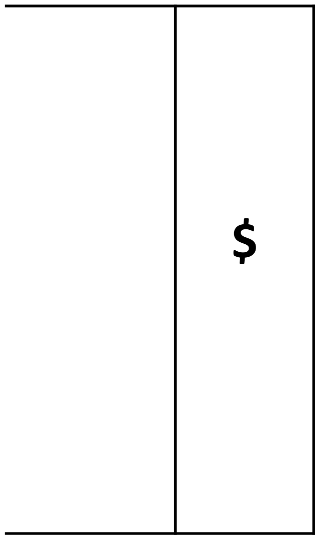
we obtain:

Productions	Actions
S -> while (M C) N S_1	{S.code = L1 || C.code || L2 || S_1 .code}
M -> €	{L1 = new label(); L2 = new label(); C.true = L2; C.false=S.next; }
N -> €	{S_1.next = L1; }

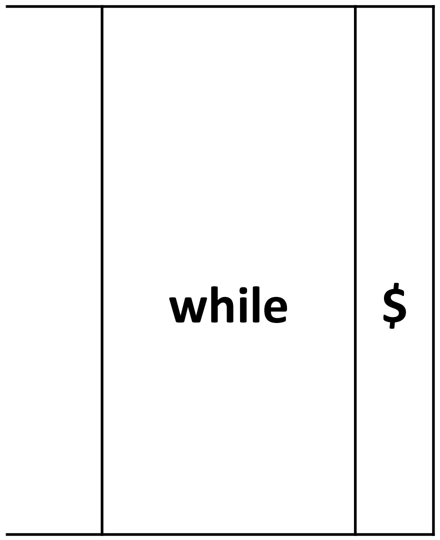
Parser Stack



Stack	Input Buffer	Action
$	*while(C)S$*	shift ***while***

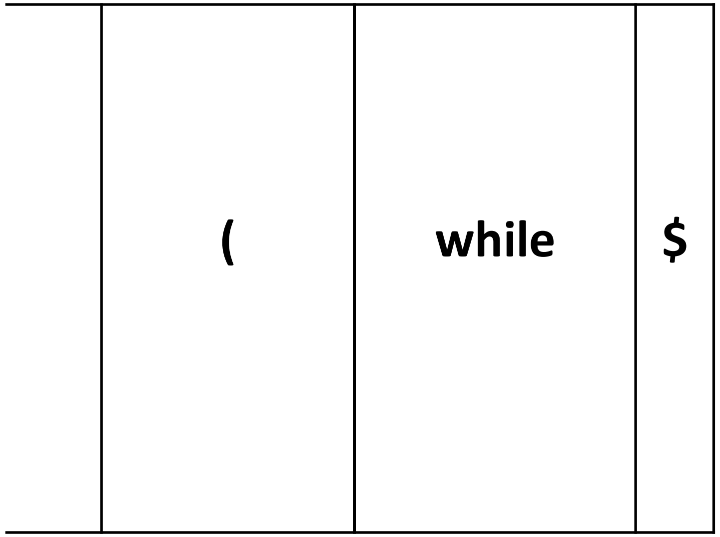


Stack	Input Buffer	Action
$while	*(C)S*$	shift **(**



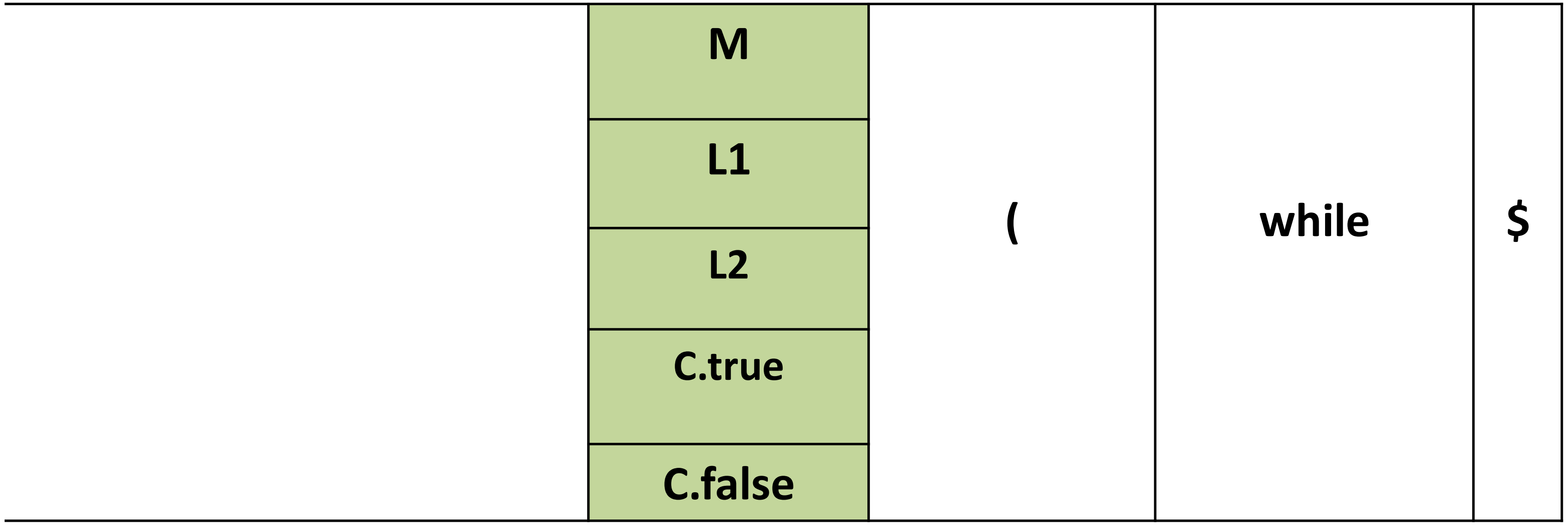
Stack	Input Buffer	Action
$while(	*C)S$*	reduce M -> *Є*

Execute the corresponding action

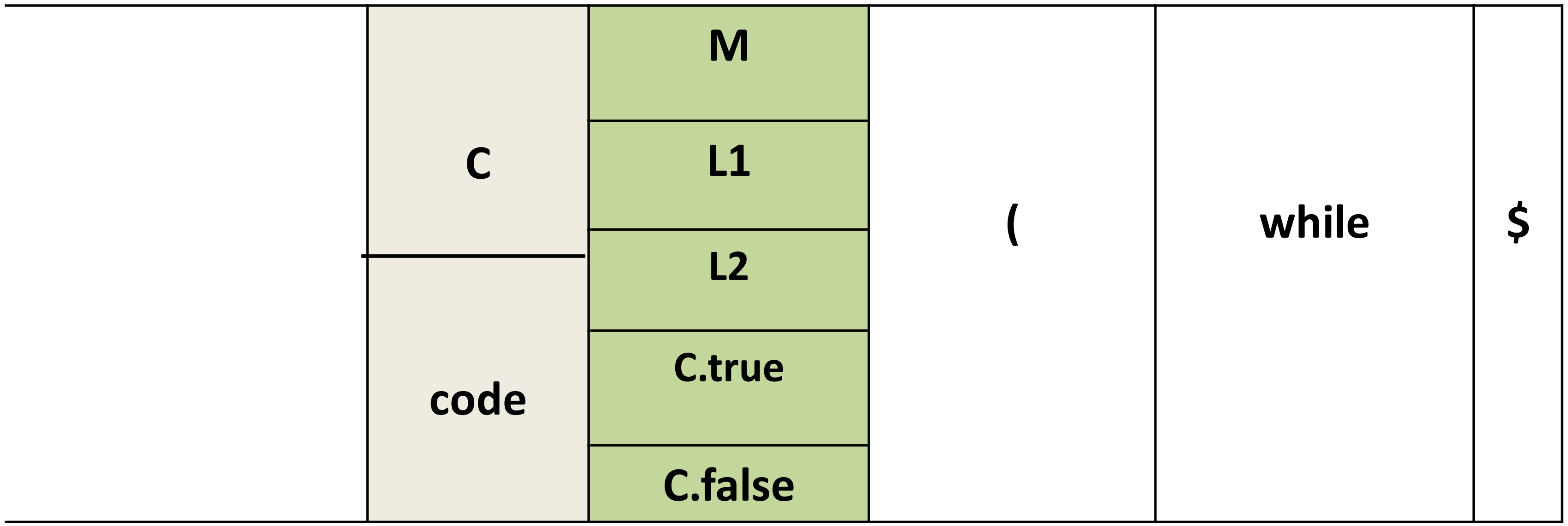


L1 = new();
L2 =new();
C.true = L2;
C.false = S.next

Stack	Input Buffer	Action
$while(M	*C)S$*	shift ***C***



Stack	Input Buffer	Action
\$while(MC	*)S\$*	shift ***)***



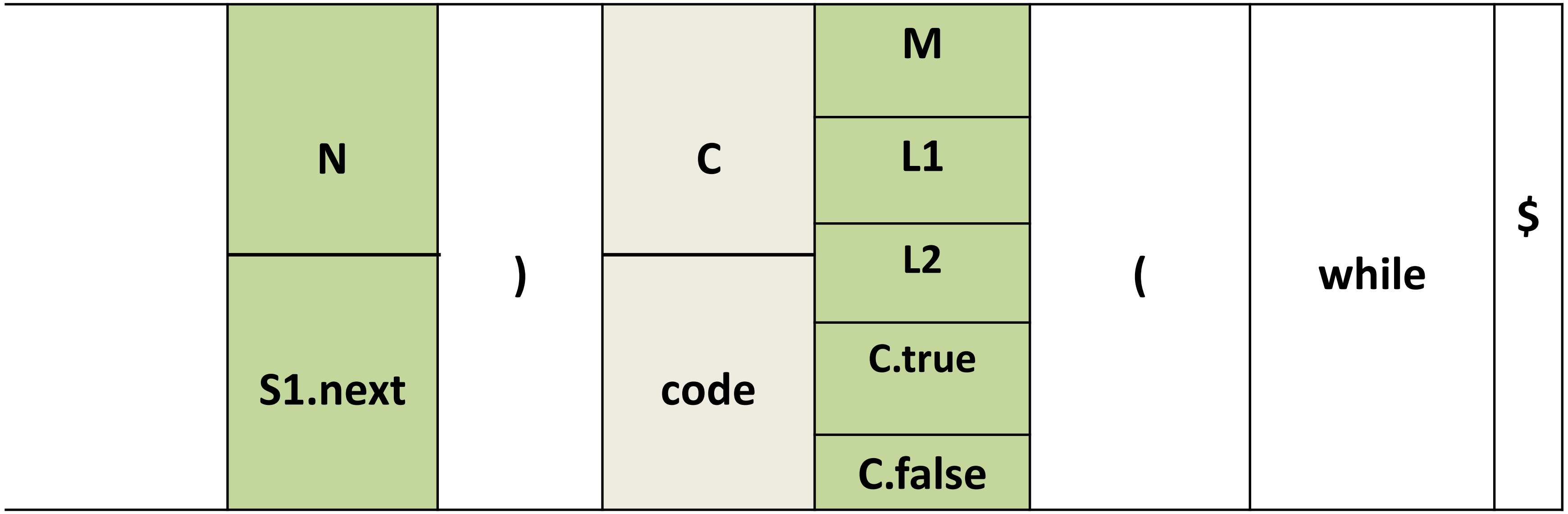
Stack	**Input Buffer**	**Action**
\$while(MC)	S\$	reduce N -> ϵ

Execute the corresponding action.

$\boxed{S_1\text{.next = L1;}}$

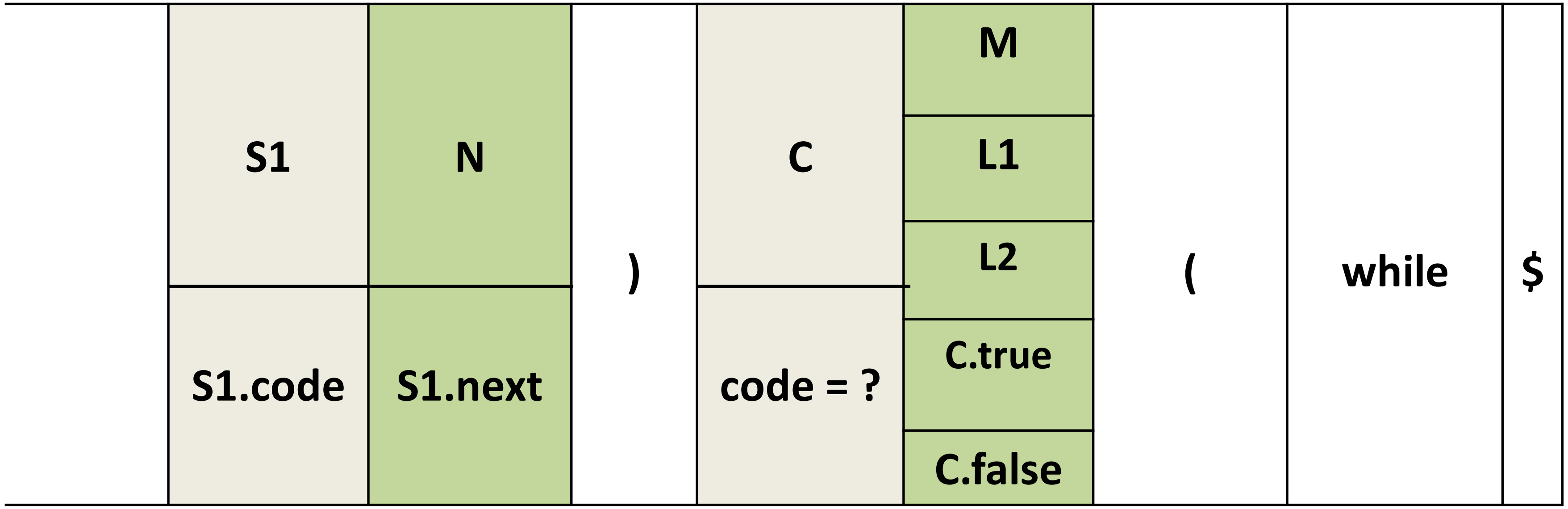
<table>
<tr><td rowspan="5"></td><td rowspan="5">)</td><td rowspan="2">C</td><td>M</td><td rowspan="5">(</td><td rowspan="5">while</td><td rowspan="5">$</td></tr>
<tr><td>L1</td></tr>
<tr><td rowspan="3">code</td><td>L2</td></tr>
<tr><td>C.true</td></tr>
<tr><td>C.false</td></tr>
</table>

Stack	Input Buffer	Action
$while(MC)N	S$	shift S_1



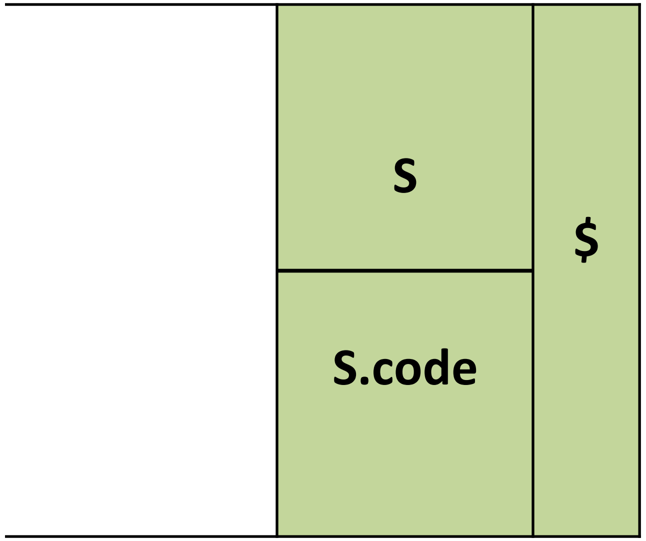
Stack	**Input Buffer**	**Action**
$while(MC)NS	$	Reduce S -> while(MC)NS

S.code =L1 || C.code || L2 ||S1.code



s[top-6].code = s[top-4].L1 || s[top – 3].code || s[top – 4].L2 || s[top].code;
top = top – 6;

Stack	Input Buffer	Action
$S	$	Accepted



s[top-6].code = s[top-4].L1 || s[top – 3].code || s[top – 4].L2 || s[top].code;
top = top – 6;