The exercises of Chapter Two

2.1 Write regular expression for the following character sets, or give reasons why no regular expression can be written:

a. All strings of lowercase letters that begin and end in *a.*

[Solution]

    a[a-z]*a | a

b. All strings of lowercase letters that either begin or end in a (or both)

    both: a(a|b|c|…|z)* a

c. All strings of digits that contain no leading zeros

[Solution]

    [1-9][0-9]*

d. All strings of digits that represent even numbers

    (0|1|2|…|9)*(0|2|4|6|8)

e. All strings of digits such that all the 2' s occur before all the 9' s

[Solution]

    a=(0|1|3|4|5|6|7|8)
    r=(2|a)*(9|a)

or

```
    [^9]*[^2]*
```

or

```
    [^9]*2(1|[3-8])*9[^2]*
```

g. All strings of a’ s and b’ s that contain an odd number of a’ s or an odd number of b’ s(or both)

[Solution]

$r1=b^*a(b|ab^*a)^*$--------odd number of a’ s

$r2= a^*b(a|ba^*b)^*$-------odd number of b’ s

$r1|r2|r1r2|r2r1$

or

$b^*a(b^*ab^*a)^*b^*|a^*b(a^*ba^*b)^*a^*$

i. All strings of a’ s and b’ s that contain exactly as many a’ s as b’ s

[Solution]

No regular expression can be written, as regular expression can not count.

2.2 Write English descriptions for the languages generated by the following regular expressions:

a. $(a|b)^*a(a|b|\varepsilon)$

[Solution]

All the strings of a' s and b' s that end with a, ab or aa.

Or

All the strings of a' s and b' s that do not end with bb.

b. All words in the English alphabet of one or more letters, which start with one capital letter and don' t contain any other capital letters.

c. $(aa|b)^*(a|bb)^*$

[Solution]

All the strings of a' s and b' s that can be divided into two sub-stings, where in the left substring, the even number of consecutive a' s are separated by b' s while in the right substring, the even number of consecutive b' are separated by a' s.

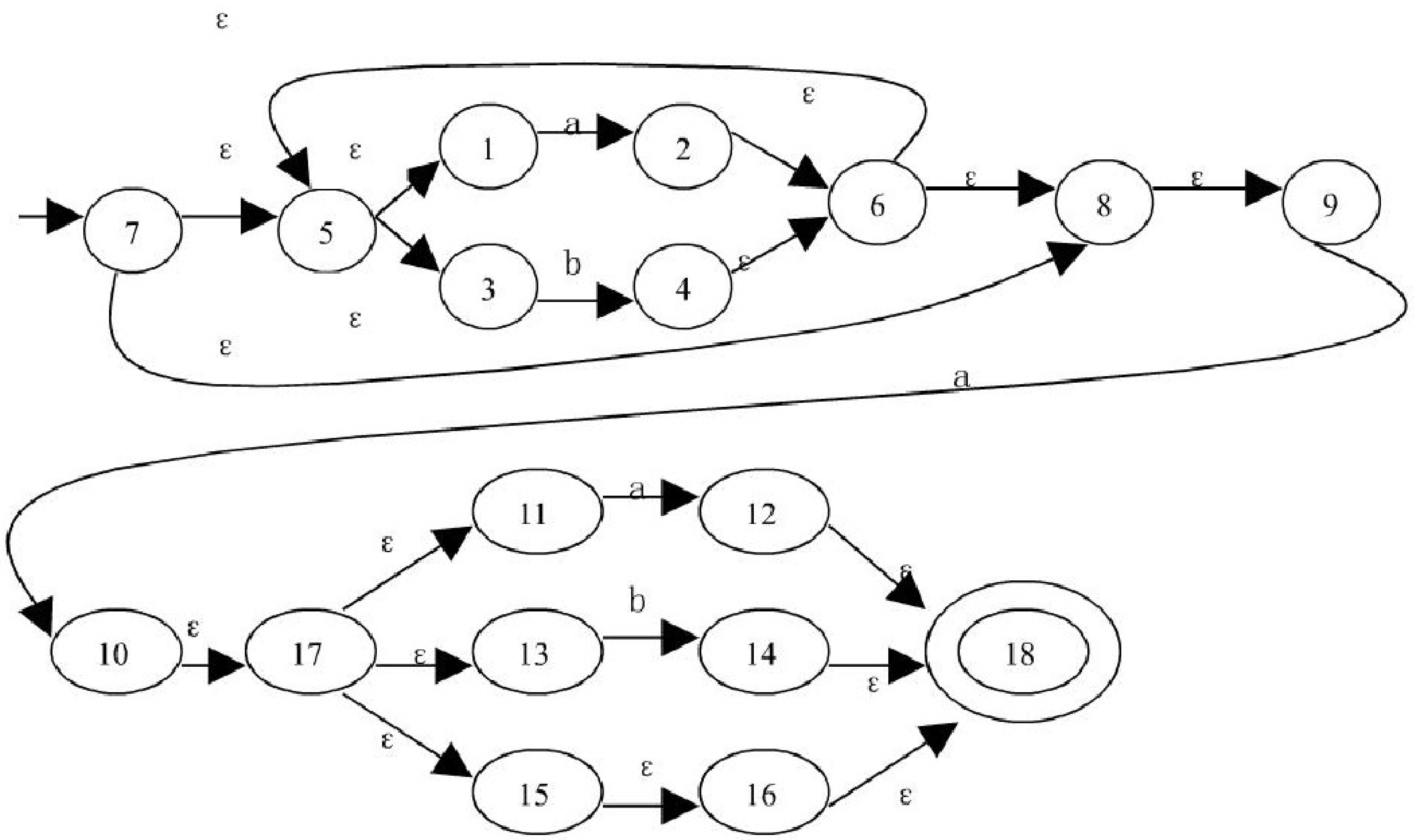
d. All hexadecimal numbers of length one or more, using the numbers zero through nine and capital letters A through F, and they are denoted with a lower or uppercase “x” at the end of the number string.

2.12 a. Use Thompson's construction to convert the regular expression (a|**b**)*a(a|**b**|ε) into an NFA.

b. Convert the NFA of part (a) into a DFA using the subset construction.

[Solution]

a. An NFA of the regular expression (a|b)*a(a|b|ε)



[Solution]

a. Step 1: Divide the state set into two subsets:

{1, 2, 3}

{4, 5}

Step 2: Further divide the subset {1,2,3} into two new subsets:

{1}

{2, 3}

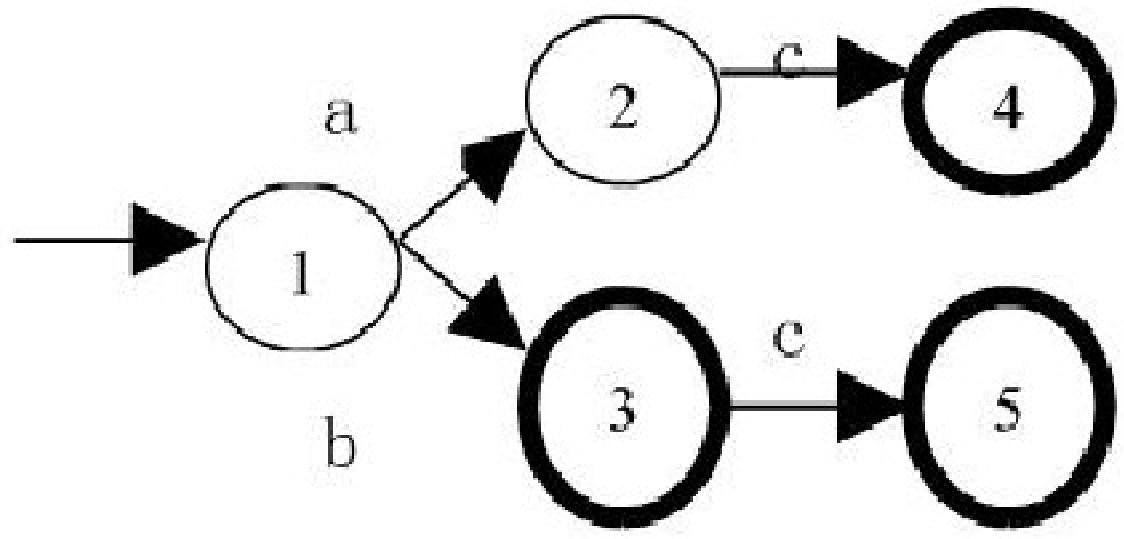
Step 3: Can not divide the subsets any more, finally obtains three subsets:

{1}

{2, 3}

{4, 5}

Therefore, the minimized DFA is:



[Solution]

b. Step 1: Divide the state set into two subsets:

{1, 2}

{3, 4, 5}

The exercises of Chapter Three

3.2 Given the grammar $A \rightarrow AA \mid (A) \mid \varepsilon$

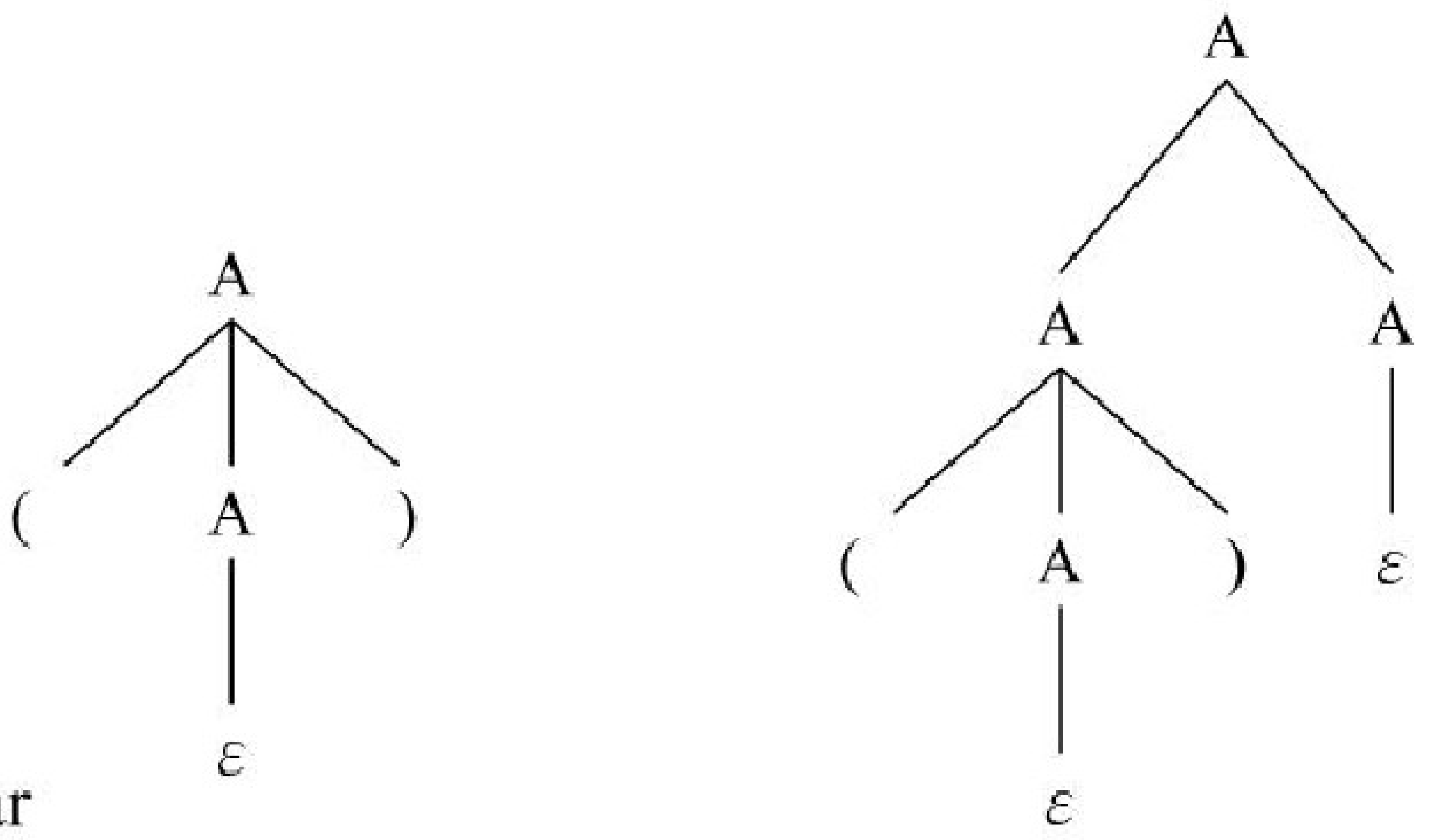
a. Describe the language it generates;

b. Show that it is ambiguous.

[Solution]:

a. Generates a string of balanced parenthesis, including the empty string.

b. parse trees of ():



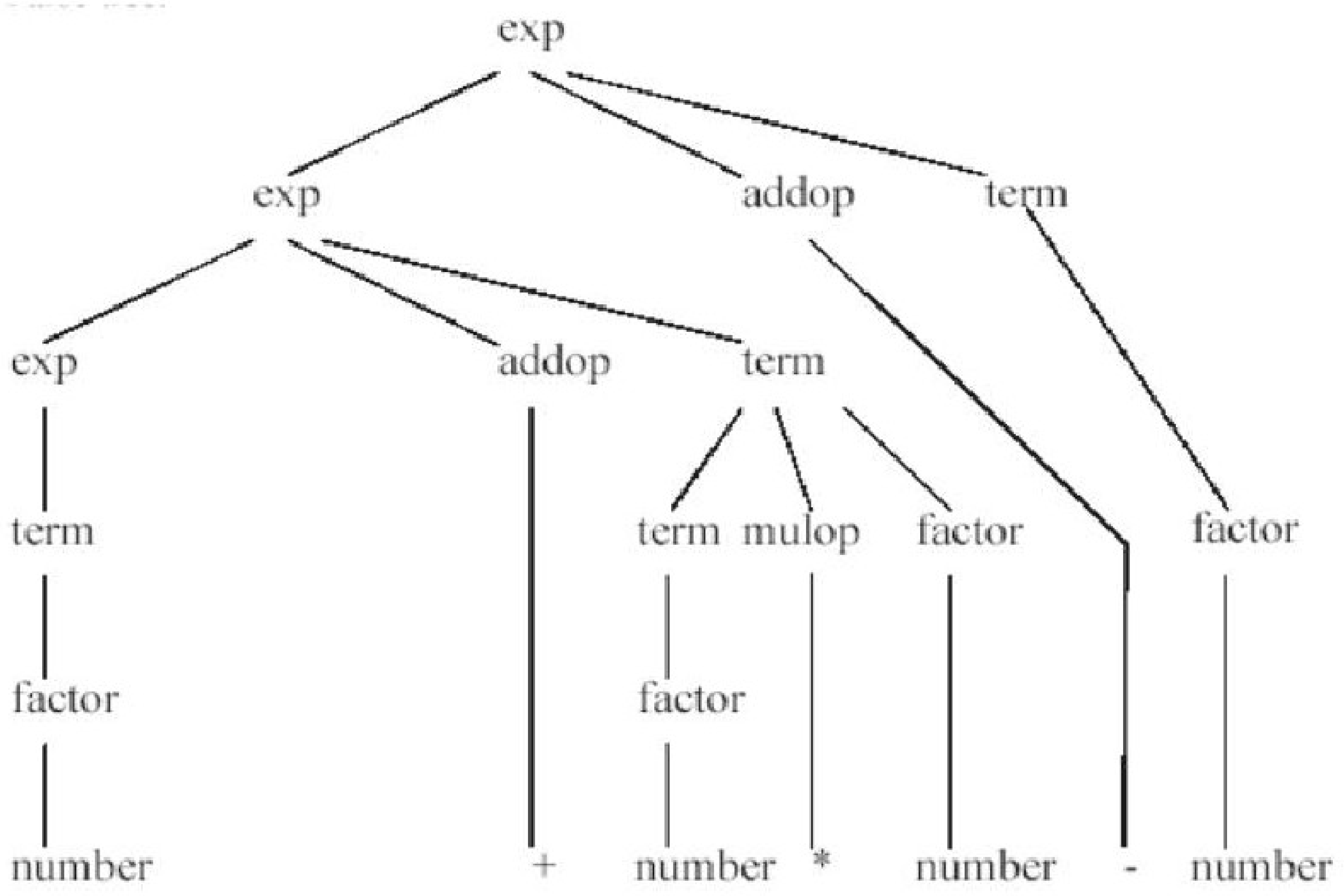
3.3 Given the grammar

exp $\rightarrow$ *exp addop term* | *term*

addop $\rightarrow$ + | -

term $\rightarrow$ *term mulop factor*| *factor*

mulop $\rightarrow$ *



3.5 Write a grammar for Boolean expressions that includes the constants true and false, the operators and, or and not, and parentheses. Be sure to give or a lower precedence than and and and a lower precedence that not and to allow repeated not's, as in the Boolean expression not not true. Also be sre your grammar is not ambiguous.

[solution]

bexp→bexp or A | A

A→ A and B | B

```
B→ not B | C
C→ (bexp) | true | false
```

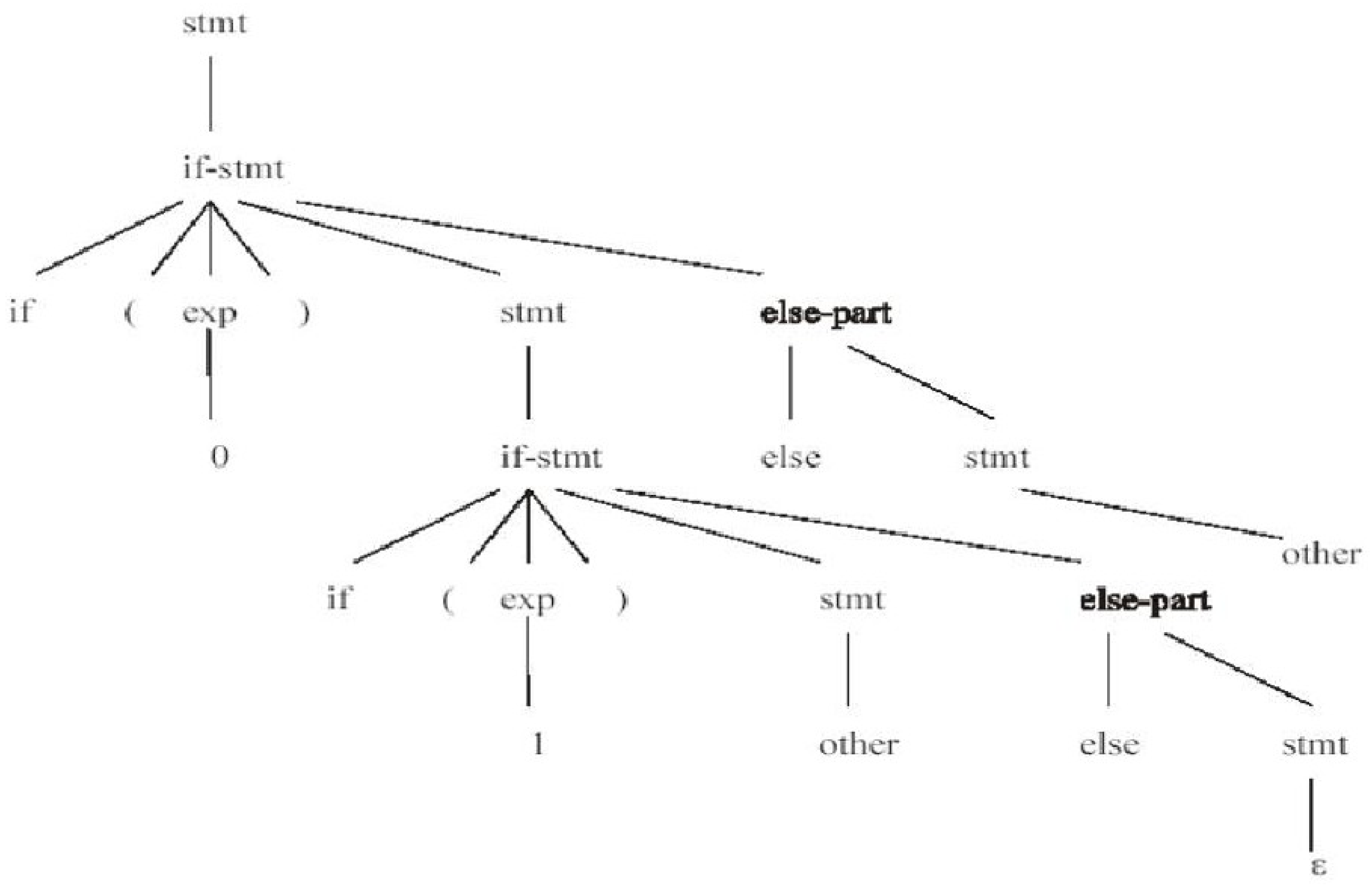
```
Ex: not not true
boolExp  → A
         → B
         → not B
         → not not B
         → not not C
         → not not true
```

3.8 Given the following grammar

```
statement→if-stmt | other | ε
if-stmt→ if ( exp ) statement else-part
else-part→ else statement | ε
exp→ 0 | 1
```

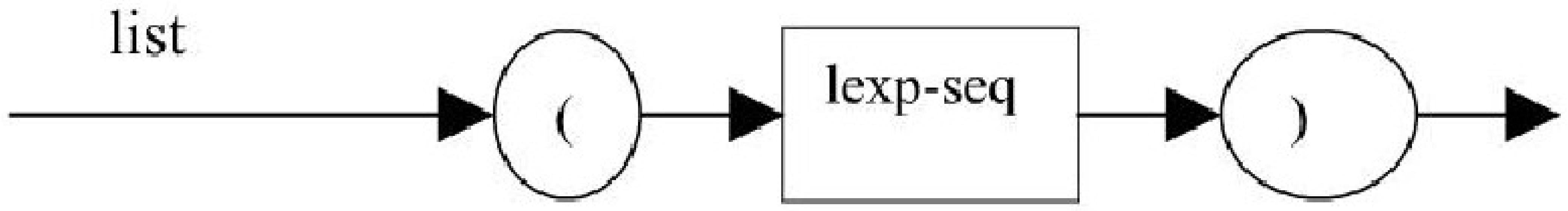
a. Draw a parse tree for the string

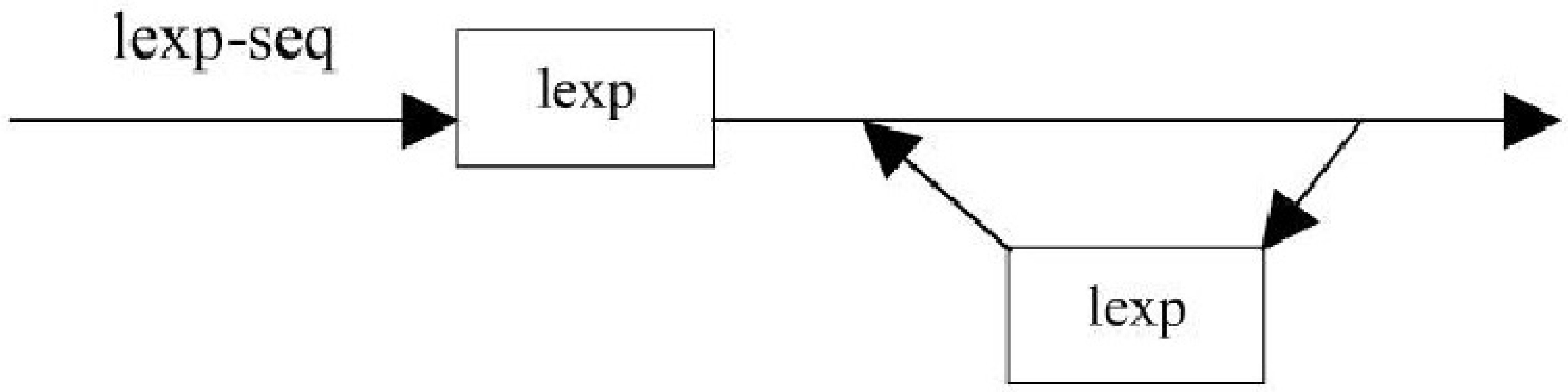
if(0) if (1) other else else other



b. what is the purpose of the two else’ s?

The two else’ s allow the programmer to associate an else clause with the outmost else, when two if statements are nested and the first does not have an else clause.





3.12. Unary minuses can be added in several ways to the simple arithmetic expression grammar of Exercise 3.3. Revise the BNF for each of the cases that follow so that it satisfies the stated rule.

a. At most one unary minus is allowed in each expression, and it must come at the beginning of an expression, so -2-3 is legal (and evaluates to -5) and -2-(-3) is legal, but -2--3 is not.

exp → *exp addop term* | *term*

addop → + | -

term → *term mulop factor*| *factor*

mulop → *

factor → (*exp*) | (-exp) | ***number*** |

b. At most one unary minus are allowed before a number or left parenthesis, so -2--3 is legal but --2 and -2---3 are not.

$exp \rightarrow exp\ addop\ term \mid term$

$addop \rightarrow + \mid -$

$term \rightarrow term\ mulop\ factor \mid factor$

$mulop \rightarrow *$

factor → (*exp*) | -(exp) | ***number*** | -***number***

c. Arbitrarily many unary minuses are allowed before numbers and left parentheses, so everything above is legal.

3.19 In some languages (Modula-2 and Ada are examples), a procedure declaration is expected to be terminated by syntax that includes the name of the procedure. For example, in Modular-2 a procedure is declared as follows:

```
PROCEDURE P;
BEGIN
……
END P;
```

Note the use of the procedure name P alter the closing END. Can such a requirement be checked by a parser? Explain.

[Answer]

This requirement can not be handled as part of the grammar without making a new rule for each legal variable name, which makes it intractable

The exercises of Chapter Four

4.2

Grammar: A → (A) A | ε

Assume we have lookahead of one token as in the example on p. 144 in the text book.

```
Procedure A()
    if (LookAhead() ∈ {'('}) then
        Call Expect('(')
        Call A()
        Call Expect (')')
        Call A()
    else
        if (LookAhead()∈ {')', $}) then
            return()
    else
        /* error */
        fi
        fi
end
```

4.3 Given the grammar

statement→ assign-stmt|call-stmt|**other**

assign-stmt→**identifier**:=exp

call-stmt→**identifier**(exp-list)

[Solution]

First, convert the grammar into following forms:

statement→ **identifier**:=exp | **identifier**(exp-list)|**other**

Then, the pseudocode to parse this grammar:

```
Procedure statement
Begin
      Case token of
      ( identifer : match(identifer);
                        case token of
                        ( := : match(:=);
                               exp;
                        ( (: match(();
                               exp-list;
                              match());
                        else error;
                        endcase
```

```
        (other: match(other);
        else error;
        endcase;
    end statement
```

4.7 a

Grammar: A → (A) A | ε

First(A)={(,ε } Follow(A)={$,)}

4.7 b

See theorem on P.178 in the text book

1. First{(}∩First{ε}=Φ
2. ε∈Fist(A), First(A) ∩Follow(A)= Φ

both conditions of the theorem are satisfied, hence grammar is LL(1)

4.9 Consider the following grammar:

lexp→atom|list

atom →number|identifier

list→(lexp-seq)

lexp-seq→lexp, lexp-seq|lexp

a. Left factor this grammar.

b. Construct First and Follow sets for the nonterminals of the resulting grammar.

c. Show that the resulting grammar is LL(1).

The Exercises of The Chapter Five

5.1

a. DFA of LR(0) items [See p. 202, p. 208, LR(0) def. p. 207]

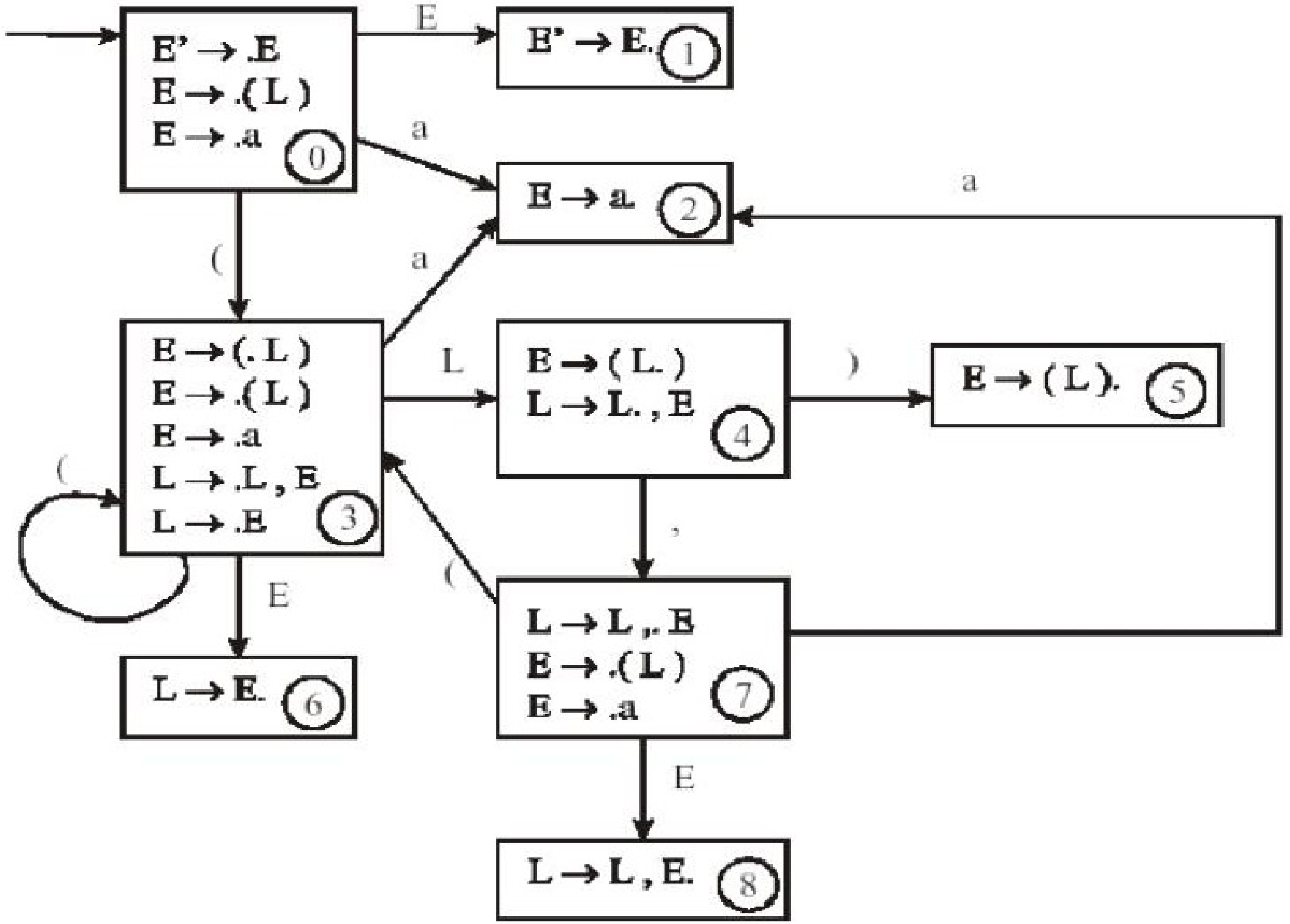
Grammar:

E   (L) | a

L   L, E | E

LR(0) items: (with augmented grammar rule E'   E)

1. E'   .E
2. E'   E.
3. E   .(L)
4. E   (. L)
5. E   (L.)
6. E   (L).
7. E   .a
8. E   a.
9. L   .L , E
10. L   L. , E
11. L   L ,. E
12. L   L , E.
13. L   .E
14. L   E.



look-ahead, whereas LR(0) detects an error in a parse string after a reduction.

5.2 Consider the following grammar:

E→(L) | a

L→L,E|E

a. Construct the DFA of LR(1) items for this grammar.

b. Construct the general LR(1) parsing table.

c. Construct the DFA of LALR(1) items for this grammar.

d. Construct the LALR(1) parsing table.

e. Describe any difference that might occur between the actions of a general LR(1) parser and an LALR(1) parser.

[Solution]

Augment the grammar by adding the production: E'→E

a.

State 0: [E'→.E,$]
[E→. (L),$]
[E→. a,$]

State 1: [E'→E.,$]

State 2: [E→(.L),$]
[L→. L,E ,)]
[L→. E ,)]
[L→. L,E , ,]
[L→. E , ,]
[E→. (L),)]

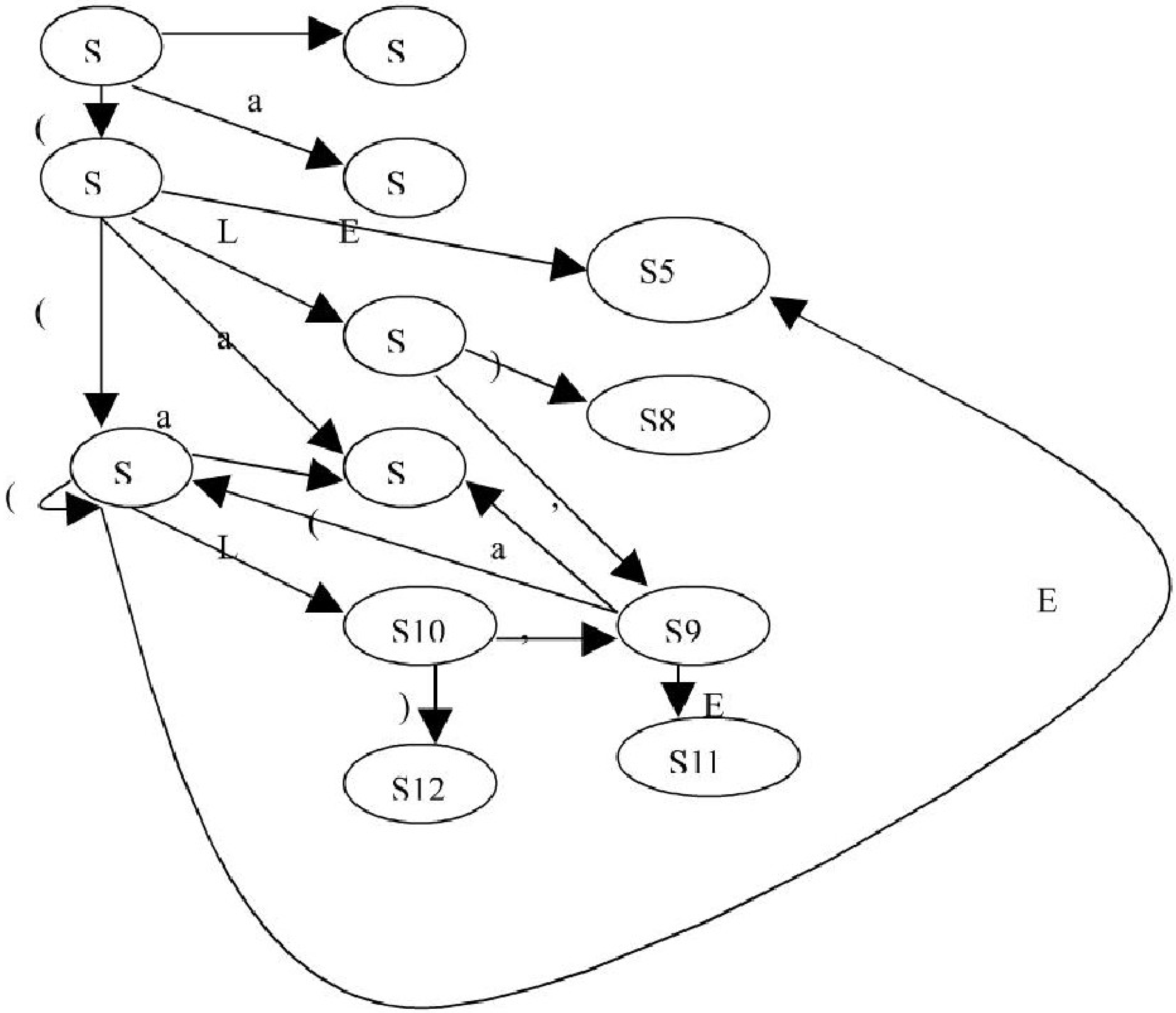
State 3: [E→a.,$]

State 4: [E→(L.),$]
[L→L.,E ,)]
[L→L.,E , ,]

	Parsing stack	Input	Action
1	$0	((a),a,(a,a))$	s3
2	$0(3	(a),a,(a,a))$	s3
3	$0(3(3	a),a,(a,a))$	s2
4	$0(3(3a2	),a,(a,a))$	r(E → a)
5	$0(3(3E6	),a,(a,a))$	r(L → E)
6	$0(3(3L4	),a,(a,a))$	s5
7	$0(3(3L4)5	,a,(a,a))$	r(E → (L))
8	$0(3E6	,a,(a,a))$	r(L → E)
9	$0(3L4	,a,(a,a))$	s7
10	$0(3L4,7	a,(a,a))$	s2
11	$0(3L4,7a2	,(a,a))$	r(E → a)
12	$0(3L4,7E8	,(a,a))$	r(L → L,E)
13	$0(3L4	,(a,a))$	s7
14	$0(3L4,7	(a,a))$	s3
15	$0(3L4,7(3	a,a))$	s2
16	$0(3L4,7(3a2	,a))$	r(E → a)
17	$0(3L4,7(3E6	,a))$	r(L → E)
18	$0(3L4,7(3L4	,a))$	s7
19	$0(3L4,7(3L4,7	a))$	s2
20	$0(3L4,7(3L4,7a2	))$	r(E → a)
21	$0(3L4,7(3L4,7E8	))$	r(L → L,E)

E

[E→(L)., ,]



0	S2	S3					1		
1						Accept			
2			S6					4	5
3			S11					10	9
4				S7					
5					S8				
6				r5	R6				
7						r1			
8						r3			
9				S12					
10					S13				
11				r6	r5				
12						r2			
13						r4			

While there is a reduce-reduce conflict in the LALR(1) parsing table:

state	Input						Goto		
	a	b	c	d	e	$	S	A	B
0	S2	S3					1		
1						Accept			
2			S6/11					4	5
3			S6/11					10	9
4				S7					
5					S8				
6/11				r5/r6	r6/r5				
7						r1			
8						r3			
9				S12					

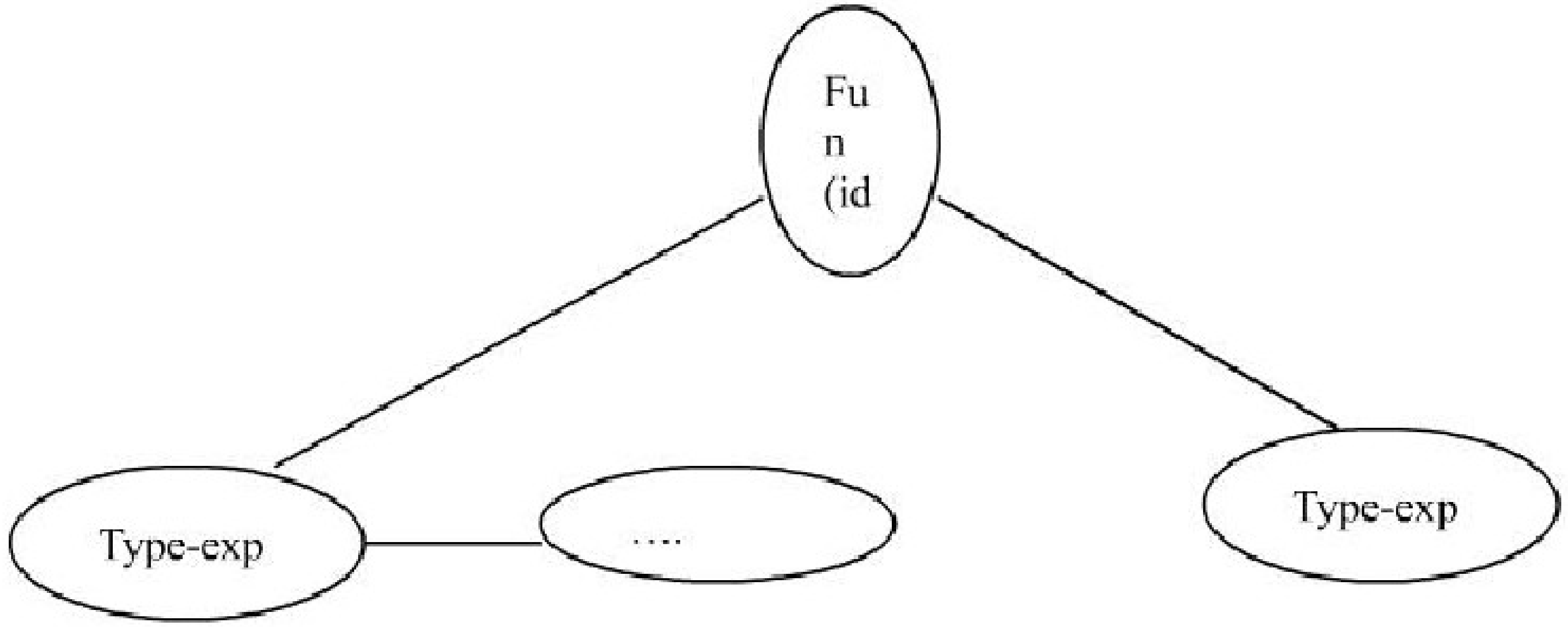
```
stmts → stmts;stmt| stmt
stmt → if exp then stmt | id:=exp
exp → exp + exp| exp or exp | exp[exp]|id(exps)
        |num|true|false|id
exps→ exps,exp|exp
```

a. Devise a suitable tree structure for the new function type structure, and write a typeEqual function for two function types.

b. Write semantic rules for the type checking of function declaration and function calls(represented by the rule exp →id(exps)), similar to rules of table 6.10(page 330).

[Solution]

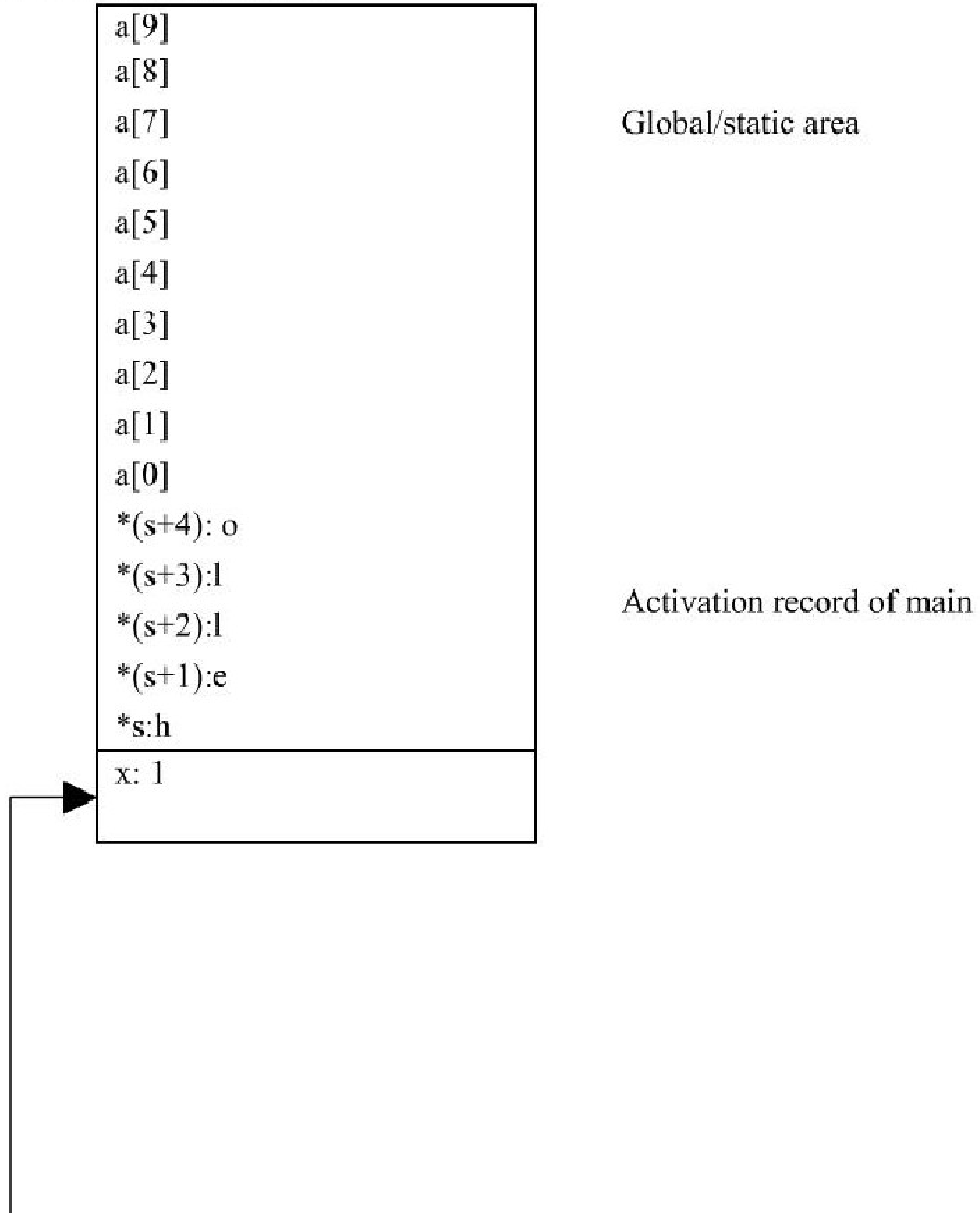
a. One suitable tree structure for the new function type structure:

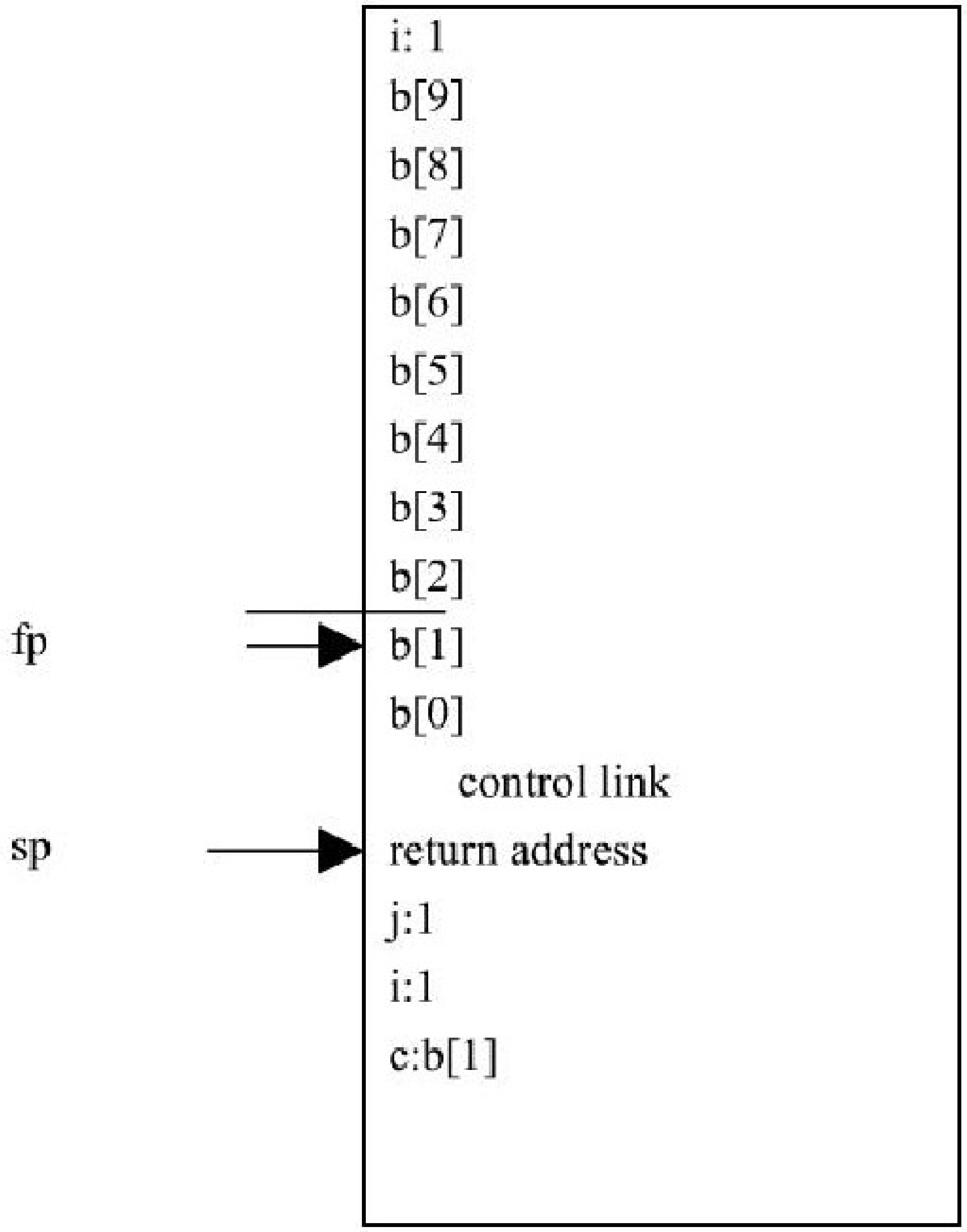


The typeEqual function for two function type:

```
    Function typeEqual-Fun(t1,t2 : TypeFun): Boolean
    Var temp : Boolean;
```

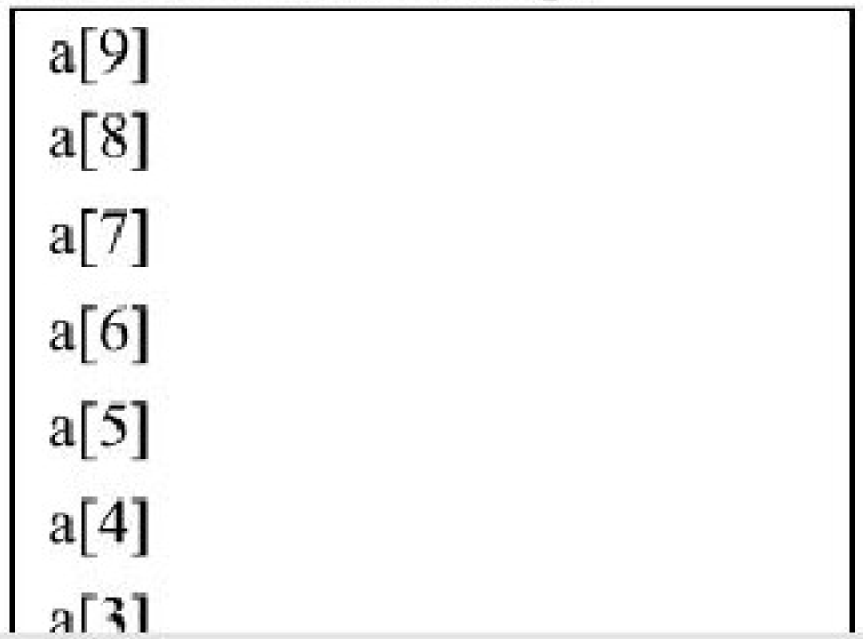
a. After entry into block A in function f.



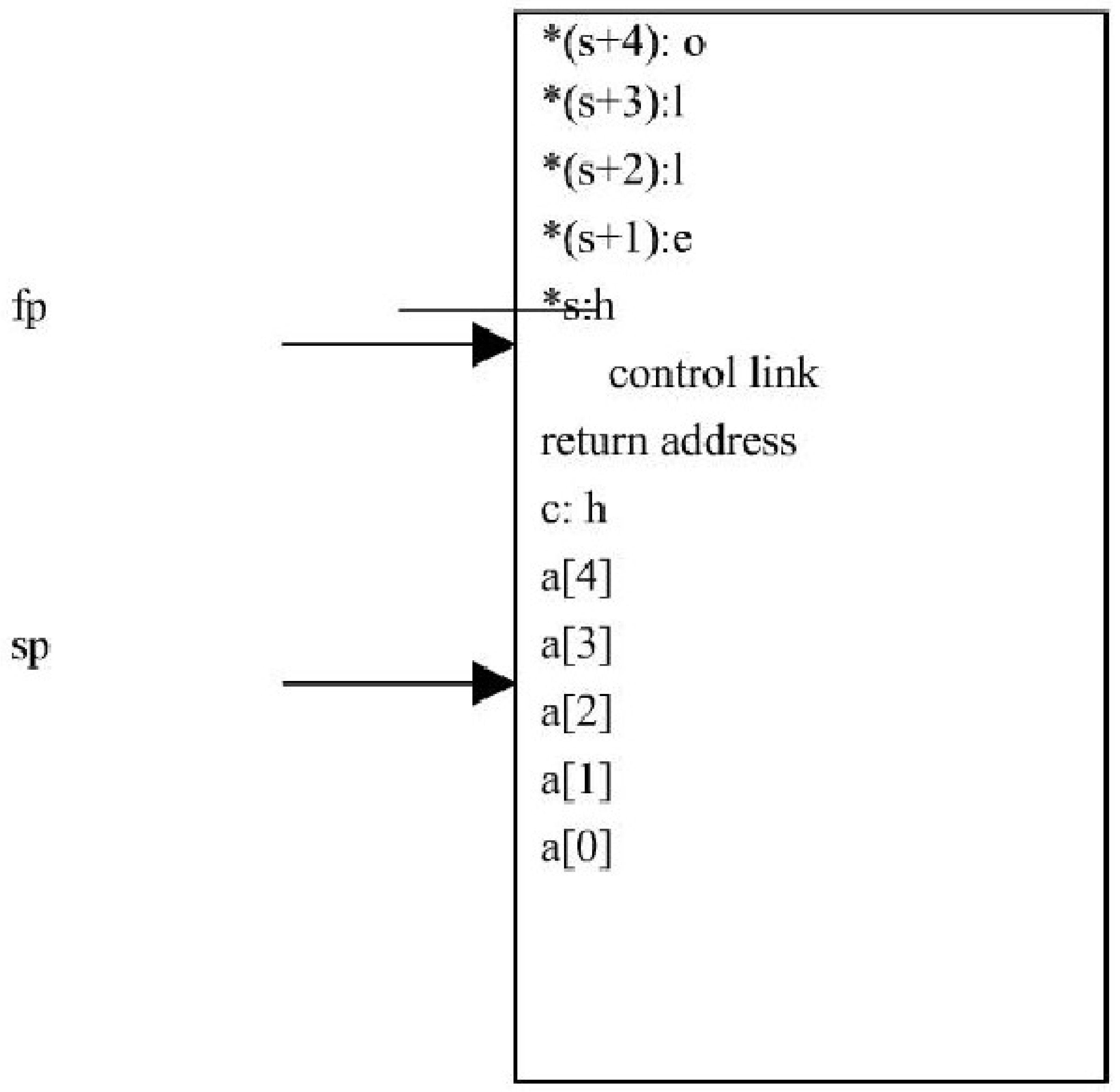


Activation record of f after entering the Block A

b. After entry into block B in function g.



Global/static area



Activation record of g after entering the Block B

7.8 In languages that permit variable numbers of arguments in procedure calls, one way to find the first argument is to compute the arguments in reverse order, as described in section 7.3.1, page 361.

a. One alternative to computing the arguments in reverse would be to reorganize the activation record to make the first argument available even in the presence of variable arguments. Describe such an activation record organization and the calling sequence it would need.

b. Another alternative to computing the arguments in reverse is to use a third point(besides the sp

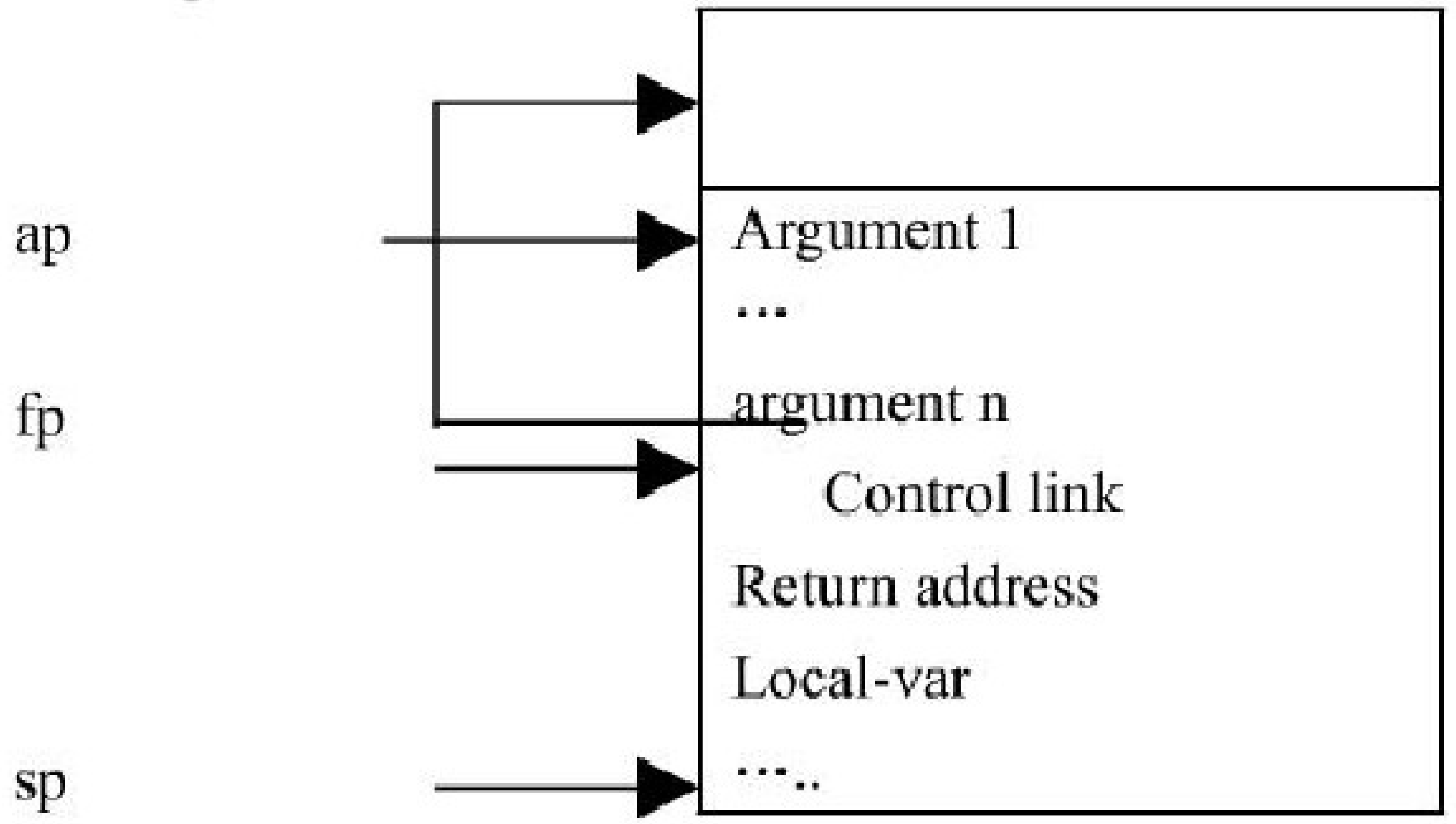
(2) change the fp to point to the beginning of the new activation record;

(3) store the return address in the new activation record;

(4) compute the arguments and store their in the new activation record in order;

(5) perform a jump to the code of procedure to be called.

b. The reorganized activation record.



The calling sequence will be:

(1) set ap point to the position of the first argument.

(2) compute the arguments and store their in the new activation record in order;

(3) store the fp as the control link in the new activation record;

(4) change the fp to point to the beginning of the new activation record;

(5) store the return address in the new activation record;

(6) perform a jump to the code of procedure to be called.

7.15 Give the output of the following program(written in C syntax) using the four parameter methods discussed in section 7.5.