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

[ˆ9]*[ˆ2]*

or

[ˆ9]*2(1|[3-8])*9[ˆ2]*

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|ε)

[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.

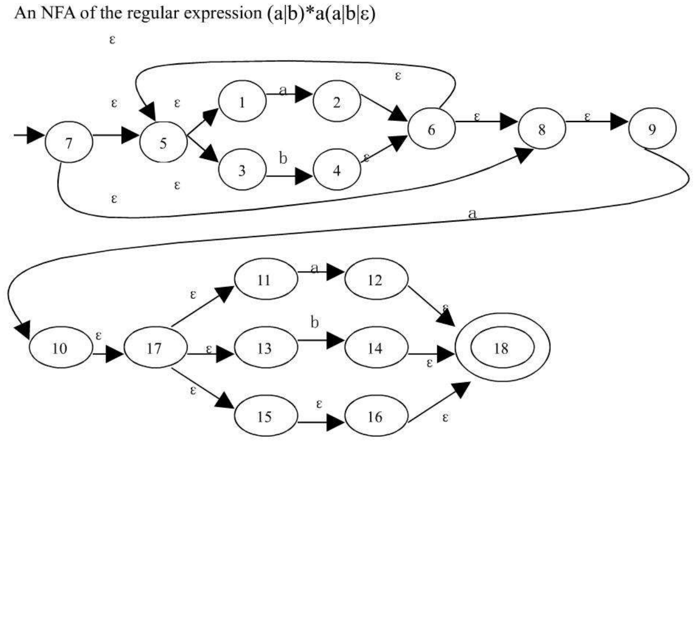
d. All hexadecimal numbers of length one or more, using the numbers zero through nine and capital letters A through F, and they are denoted with a lower or uppercase “x” at the end of the number string.

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)



b. The subsets constructed as follows:

$\{7\} = \{7, 5,1,3,8,9\}$

$\{7\}_a = \{2, 10\}$

$\{7\}_b = \{4\}$

$\{2,10\} = \{2,6,5,1,3,8,9, 10,17,11,13,15,16,18\}$

$\{2,10\}_a = \{2, 10, 12\}$

$\{2,10\}_b = \{4, 14\}$

$\{2,10,12\} = \{2,6,5,1,3,8,9,12,18,10,17,11,13,15,16\}$

$\{2,10,12\}_a = \{2,10,12\}$

$\{2,10,12\}_b = \{4, 14\}$

$\{4,14\} = \{4,6,5,1,3,8,9,14,18\}$

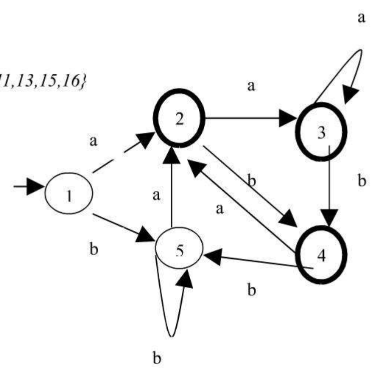
$\{4,14\}_a = \{2,10\}$

$\{4,14\}_b = \{4\}$

$\{4\} = \{4,6,5,1,3,8,9\}$

$\{4\}_a = \{2,10\}$

$\{4\}_b = \{4\}$



2.15

Assume we have r* and s* according to figure 1 and 2:

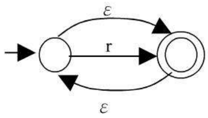


Figure 1 r*

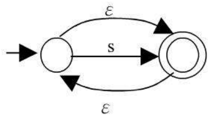


Figure 2 s*

Consider r*s* as follow

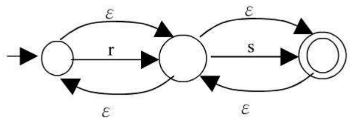
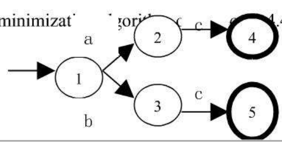


Figure 2 r*s*

This accepts, for example, rsrs which is not in r*s*. I. e., in this case we cannot eliminate the concatenating ε transition.

2.16 Apply the state minimizat[illegible] .4 to the following DFAs:

a.



[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}

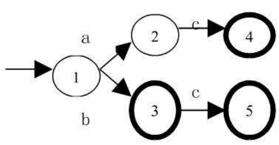
Step 3: Can not divide the subsets any more, finally obtains three subsets:

{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}

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

{1}

{2}

Step 2: Further divide the subset {3,4,5} into two new subsets:

{3}

{4, 5}

Step 4: Can not divide the subsets any more, finally obtains three subsets:

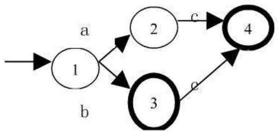
{1}

{2}

{ 3}

{4, 5}

Therefore, the minimized DFA is:



The exercises of Chapter Three

3.2 Given the grammar A→AA|(A)|ε

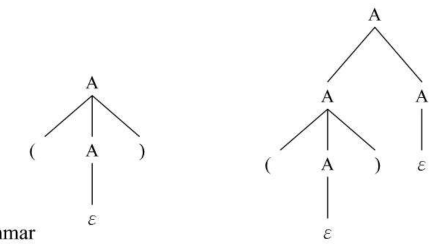
a. Describe the language it generates;

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 → *exp addop term* | *term*

addop → + | -

term → *term mulop factor* | *factor*

mulop → *

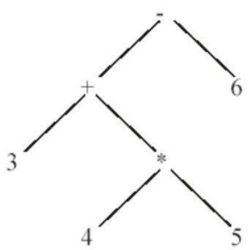
factor → **(** *exp* **)** | ***number***

Write down leftmost derivations, parse trees, and abstract syntax trees for the following expression:

a. 3+4*5-6 b. 3*(4-5+6) c. 3-(4+5*6)

[Solution]:

a. The leftmost derivations for the expression 3+4*5-6:



Exp => exp addop term =>exp addop term addop term

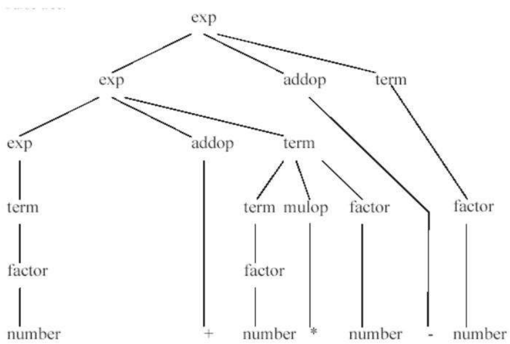
=>term addop term addop term=> factor addop term addop term

=>3 addop term addop term => 3 + term addop term

=>3+term mulop factor addop term =>3+factor mulop factor addop term

=>3+4 mulop factor addop term => 3+4* factor addop term

=>3+4*5 addop term => 3+4*5-term=> 3+4*5-factor=>3+4*5-6



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

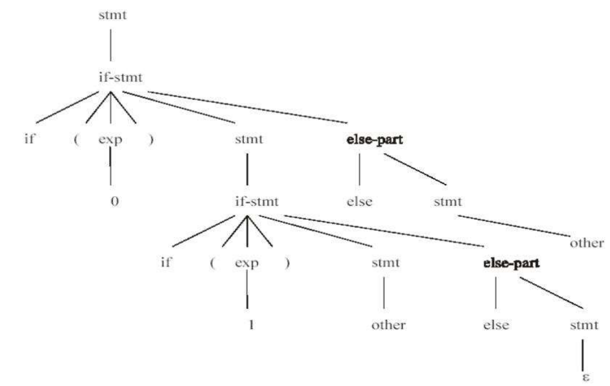
if-stmt→ if (exp) statement else-part

else-part→ else statement | ε

exp→ 0 | 1

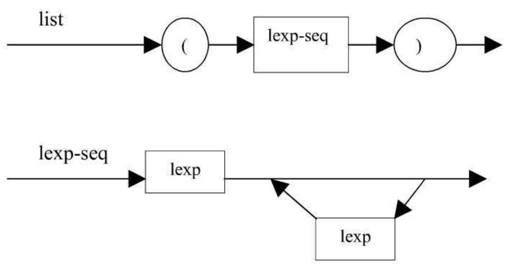
a. Draw a parse tree for the string

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*** |

c. Is similar code permissible in C? Explain.

The grammar in C looks like:

if-stmt→if (exp) statement | if (exp) statement else statement

the way to override "dangling else" problem is to enclose the inner if statement in {}s. e.g. if (0) { if(1) other } else other.

3.10 a. Translate the grammar of exercise 3.6 into EBNF.

b. Draw syntax diagramms for the EBNF of part (a).

[Solution]

a. The original grammar

lexp→ atom|list

atom→number|identifier

list→ (lexp-seq)

lexp-seq→ lexp-seq lexp| lexp

The EBNF of the above grammar:

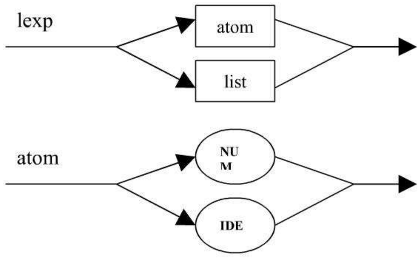
lexp→ atom|list

atom→**number|identifier**

list→ (lexp-seq)

lexp-seq→ lexp {lexp}

b. The syntax diagramms for the above EBNF:



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 → *exp addop term* | *term*
addop → + | -
term → *term mulop factor*| *factor*
mulop → *
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 for all practical purposes, even if variable names are restricted to a very short length. The parser will just check the structure, that an identifier follows the keyword PROCEDURE and an identifier also follows the keyword END, however checking that it is the same identifier is left for semantic analysis. See the discussion on pages 131-132 of your text.

3.20 a. Write a regular expression that generate the same language as the following grammar:

A→ aA|B| ε

B→ bB|A

b. Write a grammar that generates the same language as the following regular expression:

(a|c|ba|bc)*(b| ε)

[Solution]

a. The regular expression:

(a|b)*

b. The grammar:

Step 1:

A→ BC

B→aB|cB|baB|bcB| ε

C→b| ε

Step 2:

A→ Bb|B

B→aB|cB|baB|bcB| ε

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).

d. Construct the LL(1) parsing table for the resulting grammar.

e. Show the actions of the corresponding LL(1) parser, given the input string (a,(b,(2)),(c)).

[Solution]

a.

lexp→atom|list

atom →**number**|**identifier**

list→(lexp-seq)

lexp-seq→lexp lexp-seq’

lexp-seq’→, lexp-seq| ε

b.

First(lexp)={number, identifier, (}

First(atom)={number, identifier}

First(list)={(}

First(lexp-seq)={ number, identifier, (}

First(lexp-seq’)={, , ε }

Follow(lexp)={, $,) }

Follow(atom)= {, $,) }

Follow(list)= {, $,) }

Follow(lexp-seq)={$,) }

The Exercises of The Chapter Five

5.1

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

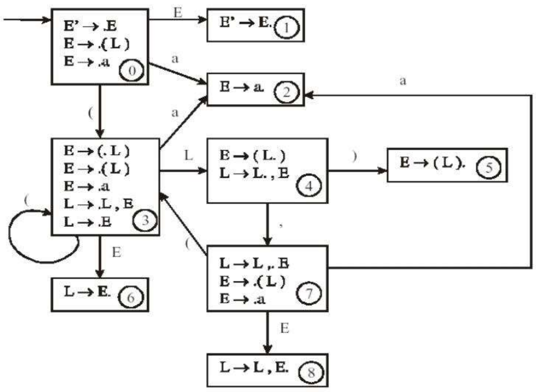
Grammar:

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.



b. SLR(1) parsing table: [See pp. 210-211]

State	Input					Goto	
	(	)	a	,	$	E	L
0	s3		s2			1	
1					accept		
2		r(E → a)		r(E → a)	r(E → a)		
3	s3		s2			6	4
4		s5		s7			
5		r(E→(L))		r(E→(L))	r(E→(L))		
6		r(L→E)		r(L→E)			
7	s3		s2			8	
8		r(L→L,E)		r(L→L,E)			

c. SLR(1) parsing stack for input string "((a),a,(a,a))": [See p. 212]

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)

22	$0(33L4,7(3L4	))$	s5
23	$0(33L4,7(3L4)5	)$	r(E → (L))
24	$0(33L4,7E8	)$	r(L → L,E)
25	$0(3L4	)$	s5
26	$0(3L4)5	$	r(E → (L))
27	$0E1	$	accept

d. [See p. 207 for def. of LR(0), p. 209 for LR(0) parsing table]

State	Action	Rule	Input				Goto	
			(	a	)	,	E	L
0	shift		3	2			1	
1	reduce	E' → E						
2	reduce	E → a						
3	shift		3	2			6	4
4	shift				5	7		
5	reduce	E → (L)						
6	reduce	L → E						
7	shift		3	2			8	
8	reduce	L → L,E						

The grammar is LR(0) as there are n o ambiguities (shift-reduce conflicts) according to the rules on p. 207(definition of LR(0)).

The difference between SLR(1) and LR(0) is that SLR(1) detects an error before a reduction because of the

[E→. (L),)]
[E→. a,)]
[E→. (L), ,]
[E→. a, ,]

State 6: [E→(.L),)]
[E→(.L), ,]
[L→. L,E ,)]
[L→. E ,)]
[L→. L,E , ,]
[L→. E , ,]
[E→. (L),)]
[E→. a,)]
[E→. (L), ,]
[E→. a, ,]

State 10: [E→(L.),)]
[E→(L.), ,]
[L→L.,E ,)]
[L→L.,E , ,]

[L→L.,E , ,]

State 5: [L→E. ,)]
[L→E. , ,]

State 7: [E→a.,)]
[E→a., ,]

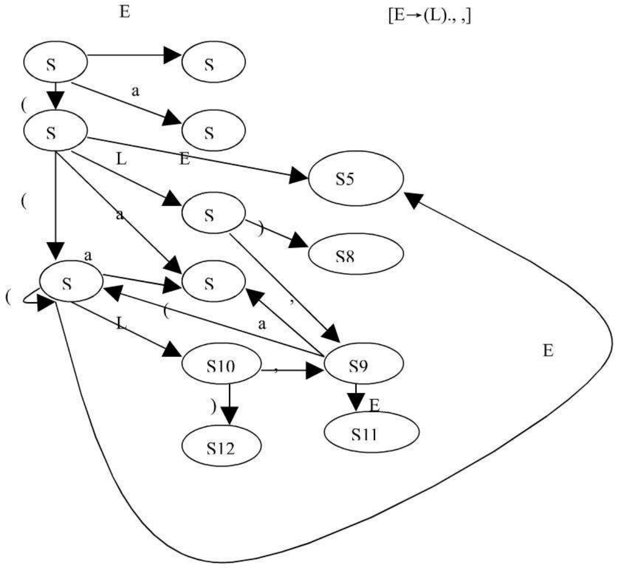
State 8: [E→(L).,$]

State 9: [L→L,.E ,)]
[E→. (L),)]
[E→. a,)]
[L→L,.E , ,]
[E→. (L), ,]
[E→. a, ,]

State 11: [L→L,E. ,)]
[L→L,E. , ,]

State 12: [E→(L).,)]

[E→(L)., ,]



b.

r1: E→(L)
r2: E→ a r3: L→L,E r4: L→E

State	Input					Goto	
	(	a	)	,	$	L	E
0	S2	S3					1
1					Accept		
2	S6	S7				4	5
3					r2		
4			S8	S9			
5			r4	r4			
6	S6	S7				10	5
7			r2	r2			
8					r1		
9	S6	S7					11
10			S12	S9			
11			r3	r3			
12			r1	r1			

c.

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

State 1: [E'→E.,$]

State 2/6: [E→(.L),$/)/,]

State 3/7: [E→a.,$/)/,]

[L→. L,E ,)]
[L→. E ,)]
[L→. L,E , ,]
[L→. E , ,]
[E→. (L),)]
[E→. a,)]
[E→. (L), ,]
[E→. a, ,]

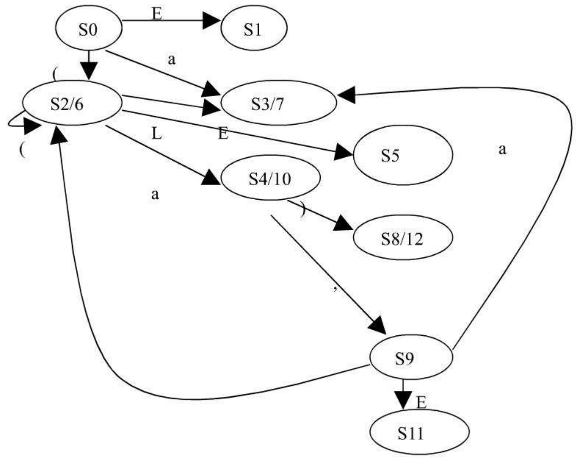
State 8/12:[E→(L).,$/)/,]

State 9: [L→L,.E ,)]
[E→. (L),)]
[E→. a,)]
[L→L,.E , ,]
[E→. (L), ,]
[E→. a, ,]

State 4/10: [E→(L.),$/)/,]
[L→L.,E ,)]
[L→L.,E , ,]

State 5: [L→E. ,)]
[L→E. , ,]

State 11: [L→L,E. ,)]
[L→L,E. , ,]



d. r1: E→(L) r2: E→ a r3: L→L,E r4: L→E

State	Input					Goto	
	(	a	)	,	$	L	E
0	S2/6	S3/7					1
1					Accept		
2/6	S2/6	S3/7				4/10	5
3/7			r2	r2	r2		
4/10			S8/12	S9			
5			r4	r4			
8/12			r1	r1	r1		
9	S2/6	S3/7					11
10			S8/12	S9			
11			r3	r3			

e. The consequence of using LALR(1) parsing over general LR(1) parsing is that , in the presence of errors, some spurious reducation may be made before error is declared. (Page 225)

5.11

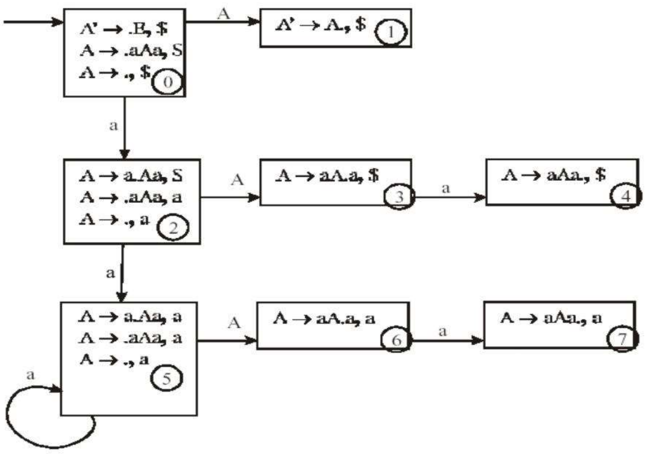
a. Augment grammar with rule E’ E :

A’ A

A aAa

A

A



There are shift-reduce conflicts in state 2 and state 5 according to condition 1 on p. 221. I.e. the given grammar is not LR(1).

b. The grammar is not ambiguous because there is only one possible parse tree for any given input string.

5.12 Show that the following grammar is LR(1) but not LALR(1):

S→aAd|bBd|aBe|bAe

A→c

B→c

[Solution]

r1: S→aAd r2: S→bBd r3: S→aBe r4: S→bAe r5: A→c r6: B→c

There is no conflicts in the following general LR(1) parsing:

state	Input						Goto		
	a	b	c	d	e	$	S	A	B
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					

The Exercises of Chapter Six

6.2

	Grammar Rules	Semantic Rules
1.	dnum → $num_i.num_d$	dnum.val = num_i.val + num_d.val * $10^{-num_d.count}$
		num_1.count = num_2.count + 1
2.	num_1 → num_2 digit	num_1.val = num_2.val * 10 + digit.val
3.	num → digit	num.val = digit.val
4.	digit → 0	digit.val = 0
	. . .	
	digit → 9	digit.val = 9

应该在 num→digit 产生式中再加一条语义规则：numd.count=1 用来进行初始化。

6.4

	Grammar Rules	Semantic Rules
1.	exp → term exp'	exp.val = term.val + exp'.val
2.	exp'_1 → + term exp'_2	exp'_1.val = term.val + exp'_2.val
3.	exp'_1 → - term exp'_2	exp'_1.val = - term.val + exp'_2.val
4.	exp'_1 → ε	exp'_1.val = 0
5.	term → factor term'	term.val = factor.val * term'.val
6.	term' → * factor term'	$term'_1$.val = factor.val * $term'_2$.val
7.	term' → ε	term'.val = 1
8.	factor → (exp)	factor.val = exp.val
9.	factor → number	factor.val = number.val

10					S13				
12						r2			
13						r4			

6.7 Consider the following grammar for simple Pascal-style declarations:

delc → var-list : type
var-list → var-list, **id** | **id**
type → **integer** | **real**

Write an attribute grammar for the type of a variable.

[Solution]

Grammar Rule	Semantic Rules
delc → var-list : type	var-list.type = type.type
var-list1 → var-list2, **id**	val-list2.type=var-list1.type
	id.type=var-list1.type
var-list → **id**	id.type=var-list.type
type → **integer**	type.type= INTERGER
type → **real**	type.type=REAL

6.10 a. Draw dependency graphs corresponding to each grammar rule of Example 6.14 (Page 283), and for the expression 5/2/2.0.

b. Describe the two passes required to compute the attributes on the syntax tree of 5/2/2.0, including a possible order in which the nodes could be visited and the attribute values computed at each point.

c. Write pseudocode for procedures that would perform the computations described in part(b).

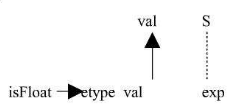
[Solution]

a. The grammar rules of Example 6.14

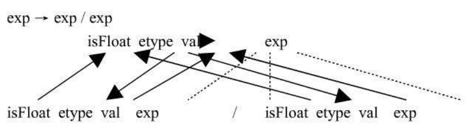
S → exp
exp → exp/exp | **num** | **num.num**

The dependency graphs for each grammar rule:

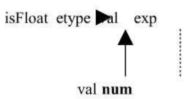
S → exp



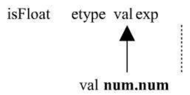
exp → exp / exp



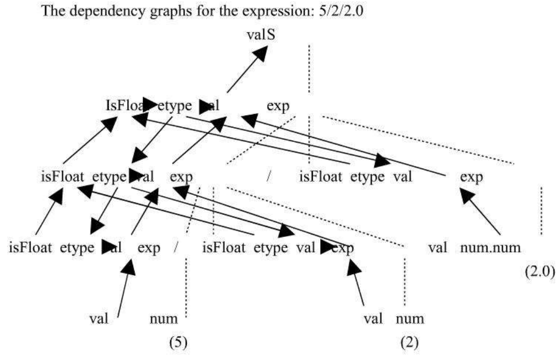
exp → **num**



exp → **num.num**



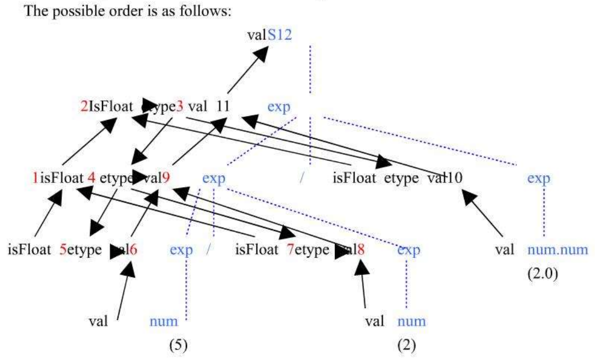
The dependency graphs for the expression: 5/2/2.0



b. The first pass is to compute the *etype* from *isFloat*.

The second pass is to compute the *val* from *etype*.

The possible order is as follows:



c. The pseudcode procedure for the computation of the isFloat.

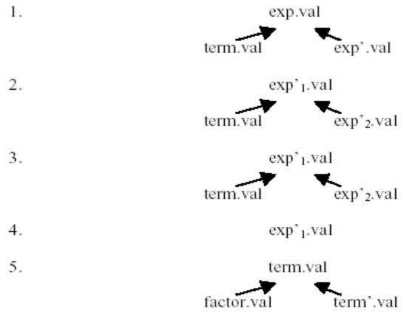
```
Function EvalisFloat(T: treenode): Boolean
Var temp1, temp2: Boolean
Begin
    Case nodekind of T of
    exp:
        temp1= EvalisFloat(left child of T);
        if right child of T is not nil then
            temp2=EvalisFloat( right child of T)
            return temp1 or temp2
        else
            return temp1;
    num:
        return false;
    num.num:
        return true;
end
```

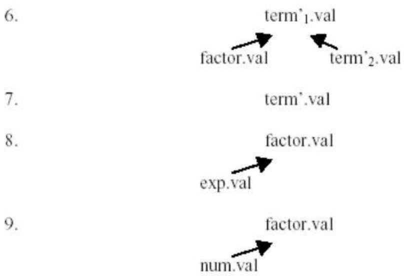
```
Function Evalval(T: treenode, etype:integer): VALUE
Var temp1, temp2: VALUE
Begin
    Case nodekind of T of
    S:
        Return(Evalval(left child of T, etype));
    Exp:
        If etype=EMPTY then
            If EvalisFloat(T) then etype:=FLOAT;
            Else etype=INT;
```

```
        Temp1=Evalval(left child of T, etype)
        If right child of T is not nil then
            Temp2=Evalval(right child of T, etype);
            If etype=FLOAT then
                Return temp1/temp2;
            Else
                Return temp1 div temp2;
        Else
            Return(temp1);
    Num:
        If etype=INT
            Return(T.val);
        Else
            Return(T.val);
    Num.num:
        Return(T.val).
```

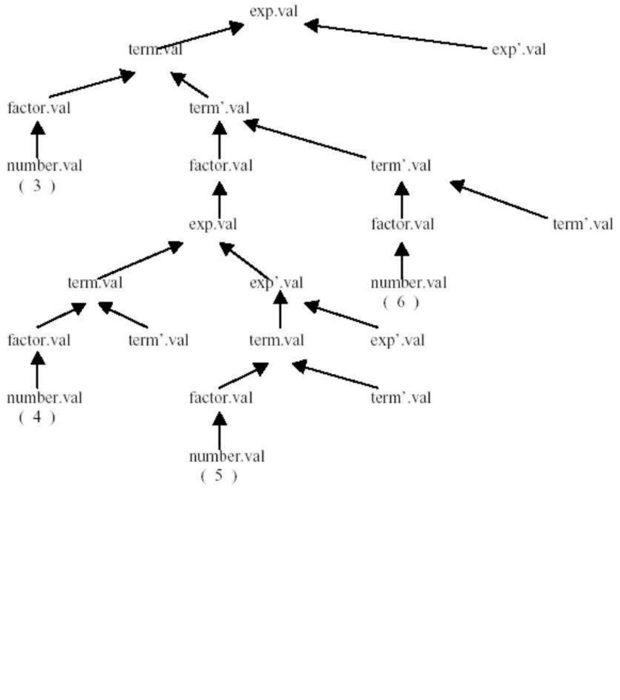
6.11

Dependency graphs corresponding to the numbered grammar rules in 6.4:





Dependency graph for the string '3 *(4+5) *6:



6.21 Consider the following extension of the grammar of Figure 6.22(page 329) to include function declarations and calls:

```
program → var-decls;fun-decls;stmts
var-decls → var-decls;var-decl|var-decl
var-decl → id: type-exp
type-exp → int|bool|array [num] of type-exp
fun-decls → fun id (var-decls):type-exp;body
body → exp
```

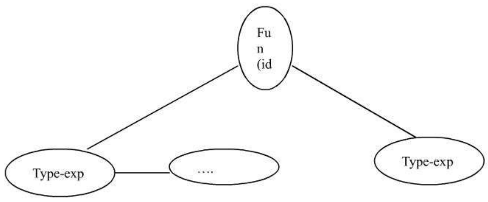
```
stmts → stmts;stmt| stmt
stmt → if exp then stmt | id:=exp
exp → exp + exp| exp or exp | exp[exp]|id(exps)
        |num|true|false|id
exps→ exps,exp|exp
```

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;
Var temp : Boolean;
      p1,p2:TypeExp
begin
      p1:=t1.lchild;
      p2:=t2.lchild;
      temp:=true;
      while temp and p1<>nil and p2<>nil do
      begin
            temp=typeEqual-Exp(p1,p2);
            p1=p1.sibling;
            p2=p2.sibling;
      end
      if temp then return(typeEqual-Exp(t1.rchild,t2.rchild));
      return(temp);
end
```

b. The semantic rules for type checking of function declaration and function call:

```
fun-decls → fun id (var-decls):type-exp; body
                              id.type.lchild:=var-decls.type;
                              id.type.rchild:=type-exp.type;
                              insert(id.name,id.typefun)

exp → id(exps)                if isFunctionType(id.type) and
                                    typeEqual-Exp(id.type.lchild,exps.type) then
                                          exp.type=id.type.rchild;
                              else type-error(exp)
```

The exercise of chapter seven

7.2 Draw a possible organization for the runtime environment of the following C program, similar to that of Figure 7.4 (Page 354).

a. After entry into block A in function f.

b. After entry into block B in function g.

int a[10];
char *s = "hello"
 	
Int f(int i, int b[ ])
{ int j=i;
A: { int i=j;
Char c = b[I];
…
}
return 0;
}
	void g(char *s)
	{ char c=s[0];
	B:{ int a[5];
	…
	}
	}
		main
		{ int x=1
		x = f(x,a);
		g(s);
		return 0;
		}

[Solution]

a. After entry into block A in function f.

a[9]
a[8]
a[7]
a[6]
a[5]
a[4]
a[3]
a[2]
a[1]
a[0]
*(s+4): o
*(s+3):l
*(s+2):l
*(s+1):e
*s:h
x: 1

Global/static area

Activation record of main

i: 1
b[9]
b[8]
b[7]
b[6]
b[5]
b[4]
b[3]
b[2]
fp → b[1]
b[0]
control link
sp → return address
j:1
i:1
c:b[1]

Activation record of f after entering the Block A

b. After entry into block B in function g.

a[9]
a[8]
a[7]
a[6]
a[5]
a[4]
a[3]

Global/static area

a[2]
a[1]
a[0]
*(s+4): o
*(s+3):l
*(s+2):l
*(s+1):e
*s:h
x: 0

Activation record of main

*(s+4): o
*(s+3):l
*(s+2):l
*(s+1):e
*s:h
fp → control link
return address
c: h
a[4]
sp → a[3]
a[2]
a[1]
a[0]

Activation record of g after entering the Block B

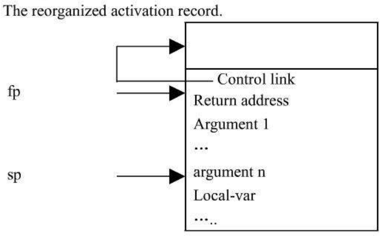
7.8 In languages that permit variable numbers of arguments in procedure calls, one way to find the first argument is to compute the arguments in reverse order, as described in section 7.3.1, page 361.

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 and fp), which is usually called the ap (argument pointer). Describe an activation record structure that uses an ap to find the first argument and the calling sequence it would need.

[Solution]

a. The reorganized activation record.



The calling sequence will be:

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

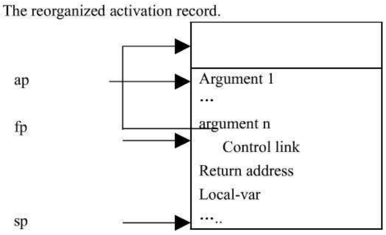
(2) change the fp to point to the beginning of the new activation record;

(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.

```
#include <stdio.h>
int i=0;

void p(int x, int y)
{     x +=1;
      i +=1;
      y +=1;
}

main
{     int   a[2]={1,1};
      p(a[i], a[i]);
      printf("%d %d\n", a[0], a[1]);
      return 0;
}
```

[Solution]

pass by value: 1, 1

pass by reference: 3, 1

pass by value-result: 2, 1

pass by name: 2, 2