Compiler Design:

Theory, Tools, and Examples

Java Edition

Seth D. Bergmann

Rowan University

2007

Instructor's Manual

(Not intended for use by students)

Chapter 1

Exercises 1.1

1. Show *assembly language* for a machine of your choice, corresponding to each of the following Java statements:

(a) `a = b + c;`

```
ld      r1,b
add     r1,c
sto     r1,a
```

(b) `a = (b+c) * (c-d);`

```
ld      r1,b
add     r1,c
ld      r2,c
sub     r2,d
mr      r1,r2
sto     r1,a
```

(c)
```
for (i=1; i<=10; i++) a = a+i;
ld     r1,1
ld     r2,a
loop:
cmp    r1,='10'
brh    done
ar     r2,r1
incr   r1
jmp    loop
done:
```

2. Show the difference between compiler output and interpreter output for each of the following source inputs:

(a)
```
     a = 12;
     b = 6;
     c = a+b;
     println (c,a,b);

Compiler output:
     mov  a,='12'
          mov  b,='6'
          lod  r2,a
          lod  r3,b
          lod  r1,a
          ar   r1,r3
          sto  r1,c
          stm  r1,r3,parms
          call println

Interpreter output:
          18126
```

(b)
```
     a = 12;
          b = 6;
          if (a<b) println (a);
          else println (b);
```

Compiler output:

```
        lod     r1,='12'
        lod     r2,='6'
        cmpr    r1,r2
        bge     less
        st      r1,parms
        call    println
        jmp     out
        less:
        st      r2,parms
        call    println
        out:
```

Interpreter output:

```
        6
```

(c)

```
     a = 12;
     b = 6;
     while (b<a)
          {  a = a-1;
             println (a+b);
          }
Compiler output:
     lod  r1,='12'
     lod  r2,='6
     loop:
     cmpr r1,r2
     bge  done
     decr r1
     stm  r1,r2,parms
     call println
     jmp  loop
     done:

Interpreter output:
     116
     106
```

```
96
86
76
66
```

3. Which of the following Java source errors would be detected at compile time, and which would be detected at run time?

(a)
```
a = b+c = 3;
```
Compile time error

(b)
```
if (x<3) a = 2
      else a = x;
```
Compile time error

(c)
```
if (a>0) x = 20;
     else if (a<0) x = 10;
          else x = x/a;
```
Run time error

(d)
```
MyClass x [] = new MyClass[100];
x[100] = new MyClass;
```
Run time error

4. Using the big C notation, show the symbol for each of the following:

(a) A compiler which translates COBOL source programs to PC machine language and runs on a PC.

$$\mathbf{C}^{\mathbf{COBOL} \rightarrow \mathbf{PC}}_{\mathbf{PC}}$$

(b) A compiler, written in Java, which translates FORTRAN source programs to Mac machine language.

$$C_{\mathbf{PC}}^{\mathbf{FORTRAN} \rightarrow \mathbf{Mac}}$$

(c) A compiler, written in Java, which translates Sun machine language programs to Java.

$$C_{\mathbf{Java}}^{\mathbf{Sun} \rightarrow \mathbf{Java}}$$

1.2 The Phases of a Compiler

Exercises 1.2

1. Show the *lexical tokens* corresponding to each of the following Java source inputs:

(a) `for (i=1; i<5.1e3; i++) func1(x);`

keyword	for
special char	(
identifier	i
operator	=
numeric const	1
special char	;
identifier	i
operator	<
numeric const	5.1e3
special char	;
identifier	i
operator	++
special char	)
identifier	func1
special char	(
identifier	x
special char	)
special char	;

(b) `if (sum!=133) /* sum = 133 */`

keyword	if
special char	(
identifier	sum
operator	!=
numeric const	133
special char	)
comment	/* sum = 133 */

(c) `) while ( 1.3e-2 if &&`

special char	)
keyword	while

special char	(
numeric const	1.3e-2
keyword	if
operator	&&

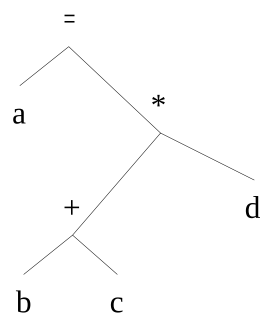
(d) `if 1.2.3 < 6`

keyword	if
numeric const	1.2
numeric const	.3
operator	<
numeric const	6

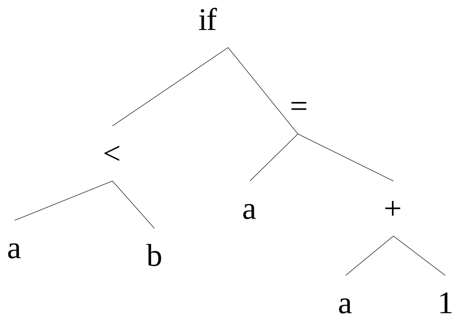
2. Show the sequence of atoms put out by the parser, and show the *syntax tree* corresponding to each of the following Java source inputs:

(a) `a = (b+c) * d;`

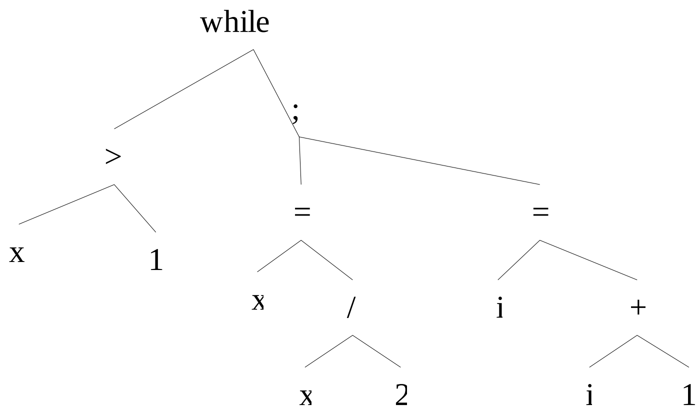
(ADD, b, c, T1)
(MUL, T1, d, T2)
(MOV, T2, , a)



(b)
```
if (a<b) a = a + 1;
```
(TST, a, b, , 2, L1)
(JMP, L2)
(LBL, L1)
(ADD, a, 1, T1)
(MOV, T1, , a)
(LBL, L2)

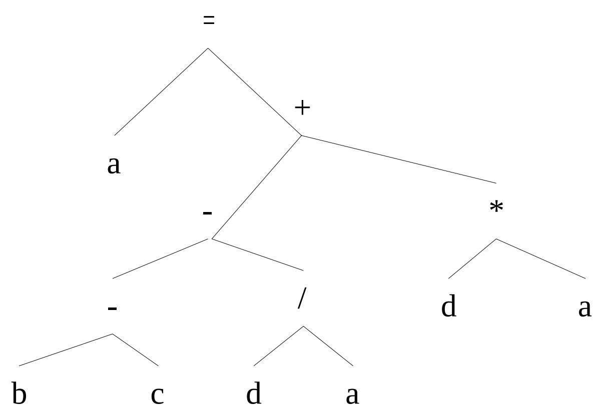


(c)
```
while (x>1)
    { x = x/2;
     i = i+1;
    }
```
```
(LBL, L1)
```
(TST, x, 1, 3, , L3)
(JMP, L2)
(LBL, L3)
(DIV, x, 2, T1)
(MOV, T1, , x)
(ADD, i, 1, T2)
(MOV, T2, , i)
(JMP, L1)
(LBL, L2)



(d) `a = b - c - d/a + d * a;`

(SUB, b, c, T1)
(DIV, d, a, T2)
(SUB, T1, T2, T3)
(MUL, d, a, T4)
(ADD, T3, T4, T5)
(MOV, T5, , a)



3. Show an example of a *Java statement* which indicates that the order in which the two operands of an `ADD` are evaluated can cause different results:

```
operand1 + operand2
```

(a=2) + (a=3)

method1() + method2()

4. Show how each of the following *Java source inputs* can be optimized using global optimization techniques:

(a)
```
for (i=1; i<=10; i++)
    { x = i + x;
      a[i] = a[i-1];
      y = b * 4;
    }
```

```
for (i=1; i<=10; i++)
        { x = i + x;
          a[i] = a[i-1];
        }
      y = b * 4;
```

(b)
```
for (i=1; i<=10; i++)
    { x = i;
      y = x/2;
      a[i] = x;
    }
```

```
for (i=1; i<=10; i++)
     a[i] = i;
x = 10;
y = x/2;
```

(c)
```
if (x>0) {x = 2; y = 3;}
```

```
                else {y = 4; x = 2;}

if (x>0)
     y = 3;
else
     y = 4;
x = 2;
```

(d)
```
     if (x>0) x = 2;
     else if (x<=0) x = 3;
     else x = 4;

     if (x>0) x = 2;
     else  x = 3;
```

5. Show, in *assembly language* for a machine of your choice, the output of the code generator for the following atom string:

```
          (ADD,A,B,Temp1)
          (SUB,C,D,Temp2)
          (TEST,Temp1,<,Temp2,L1)
          (JUMP,L2)
          (LBL,L1)
          (MOVE,A,B)
          (JUMP,L3)
          (LBL,L2)
          (MOVE,B,A)
          (LBL,L3)

       lod  r1,A
       add  r1,B
       sto  r1,Temp1
       lod  r1,C
       sub  r1,D
       sto  r1,Temp2
       jmp  L2
       L1:
```

```
lod   r1,A
sto   r1,B
jmp   L3
L2:
lod   r1,B
sto   r1,A
L3:
```

6. Show a *Java source statement* which might have produced the atom string in Problem 5, above.

```
if (A+B < C-D) A = B; else B = A;
```

7. Show how each of the following *object code segments* could be optimized using local optimization techniques:

(a)

```
LD   R1,A
MULT R1,B
ST   R1,Temp1
LD   R1,Temp1
ADD  R1,C
ST   R1,Temp2

LD   R1,A
MULT R1,B
ADD  R1,C
ST   R1,Temp2
```

(b)

```
LD   R1,A
ADD  R1,B
ST   R1,Temp1
MOV  C,Temp1

LD   R1,A
ADD  R1,B
ST   R1,C
```

(c)

```
           CMP   A,B
           BH    L1
           B     L2
     L1:   MOV   A,B
           B     L3
     L2:   MOV   B,A
     L3:
```

```
           CMP   A,B
           BLE   L1
           MOV   A,B
           B     L3
L1:  MOV   B,A
L3:
```

Exercises 1.3

1. Fill in the missing information in the compilations indicated below:

(a)

$$C^{\text{Java} \to \text{Mac}}_{\text{Java}} \longrightarrow \boxed{\overset{\text{PC}}{C^{\text{Java} \to \text{PC}}_{\text{PC}}}} \longrightarrow C^{\text{Java} \to \text{Mac}}_{\text{PC}}$$

(b)

$$C^{\text{Java} \to \text{Mac}}_{\text{Java}} \longrightarrow \boxed{\overset{\text{PC}}{C^{\text{Java} \to \text{Mac}}_{\text{PC}}}} \longrightarrow C^{\text{Java} \to \text{Mac}}_{\text{Mac}}$$

(c)

$$C^{\text{Ada} \to \text{Sun}}_{\text{Ada}} \longrightarrow \boxed{\overset{\text{Sun}}{C^{\text{Ada} \to \text{Sun}}_{\text{Sun}}}} \longrightarrow C^{\text{Ada} \to \text{Sun}}_{\text{Sun}}$$

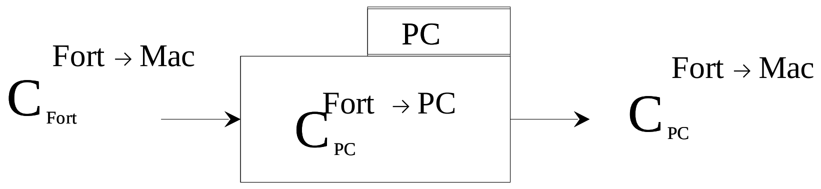
(d)

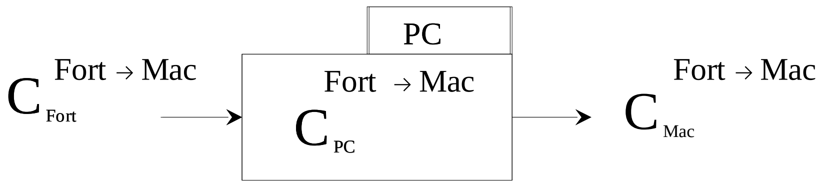
$$C^{\text{Mac} \to \text{Java}}_{\text{Java}} \longrightarrow \boxed{\overset{\text{Mac}}{C^{\text{Java} \to \text{Sun}}_{\text{Mac}}}} \longrightarrow C^{\text{Mac} \to \text{Java}}_{\text{Sun}}$$

2. How could the compiler generated in part (d) of Question 1 be used?

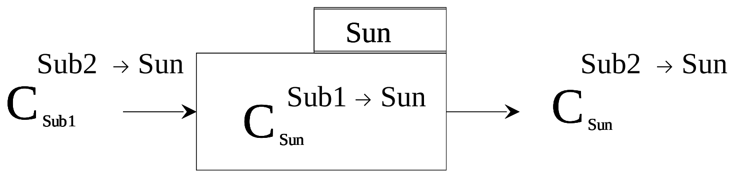
It could be used to decompile Mac programs (i.e. executables) to Java, using a Sun computer.

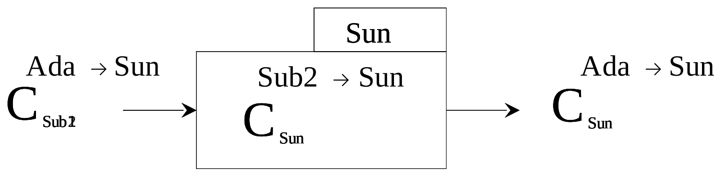
3. If the only computer you have is a PC (for which you already have a FORTRAN compiler), show how you can produce a FORTRAN compiler for the Mac computer, without writing any assembly or machine language.





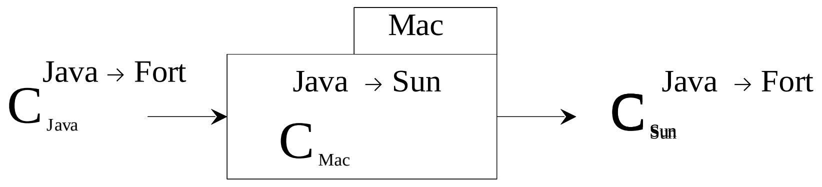
4. Show how Ada can be bootstrapped in two steps on a Sun, using first a small subset of Ada, `Sub1`, and then a larger subset, `Sub2`. First use `Sub1` to implement `Sub2` (by bootstrapping), then use `Sub2` to implement Ada (again by bootstrapping). `Sub1` is a subset of `Sub2`.





5. You have 3 computers: a PC, a Mac, and a Sun. Show how to generate automatically a Java to FORT translator which will run on a Sun if you also have the four compilers shown below:

$C_{Mac}^{Java \to Fort}$ $\quad C_{Sun}^{Fort \to Java}$ $\quad C_{Mac}^{Java \to Sun}$ $\quad C_{Java}^{Java \to Fort}$



6. In Figure 1.8 suppose we also have $C_{Java}^{Java \to Sun}$. When we write $C_{Java}^{Java \to Mac}$, which of the phases of $C_{Java}^{Java \to Sun}$ can be reused as is?

Lexical and Syntax (also Global Optimization)

7. Using the big C notation, show the 11 translators which are represented in figure 1.9. Use "Int" to represent the intermediate form.

$C^{\text{Java} \rightarrow \text{PC}}_{\text{PC}}$ $C^{\text{C++} \rightarrow \text{PC}}_{\text{PC}}$ $C^{\text{Ada} \rightarrow \text{PC}}_{\text{PC}}$

$C^{\text{Java} \rightarrow \text{Mac}}_{\text{Mac}}$ $C^{\text{C++} \rightarrow \text{Mac}}_{\text{Mac}}$ $C^{\text{Ada} \rightarrow \text{Mac}}_{\text{Mac}}$

$C^{\text{Java} \rightarrow \text{IF}}$ $C^{\text{C++} \rightarrow \text{IF}}$ $C^{\text{Ada} \rightarrow \text{IF}}$

$C^{\text{IF} \rightarrow \text{PC}}_{\text{PC}}$ $C^{\text{IF} \rightarrow \text{Mac}}_{\text{Mac}}$

Exercises 1.4

1. Which of the following are valid program segments in Decaf? Like Java, Decaf programs are free-format (Refer to Appendix A).

(a)
```
for (x = 1; x<10; )
     y = 13;
```

`Valid`

(b)
```
if (a<b) { x =
     2;  y = 3 ;}
```

`Valid`

(c)
```
while (a+b==c) if (a!=c)
        a = a + 1;
```

```
Valid
```

(d)
```
{
        a = 4 ;
        b = c = 2; ;
}
```

```
Valid
```

(e)
```
for (i==22; i++; i=3) ;
```

```
Not valid
```

2. Modify the Decaf description given in Appendix A to include a `switch` statement as defined in standard Java.

```
Stmt →          SwitchStmt

SwitchStmt  →   switch (Expr) CaseList
CaseList  →     Case CaseList
CaseList  →     ε
Case      →     case number : Stmt
Case      →     break ;
Case      →     default : Stmt
```

3. Modify the Decaf description given in Appendix A to include a `do while` statment as defined in standard Java.

```
Stmt →          doWhileStmt

doWhileStmt → do Stmt  while (BoolExpr) ;
```

Exercises 2.0

1. Suppose L1 represents the set of all strings from the alphabet {`0,1`} which contain an even number of ones (even parity). Which of the following strings belong to `L1`?

(a)	0101	(b)	110211	(c)	000
(d)	010011	(e)	ε		

(a, c, e)

2. Suppose `L2` represents the set of all strings from the alphabet {`a,b,c`} which contain an equal number of `a'`s, `b'`s, and `c'`s. Which of the following strings belong to `L2`?

(a)	`bca`	(b)	`accbab`	(c)	ε
(d)	`aaa`	(e)	`aabbcc`		

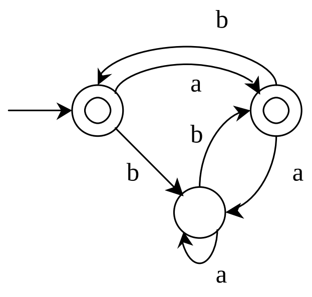
(a, b, c, e)

3. Which of the following are examples of languages?

(a) `L1` from Problem 1 above. (b) `L2` from Problem 2 above.
(c) Java (d) The set of all programming languages
(e) Swahili

(a, b, c, e)

4. Which of the following strings are in the language specified by this finite state machine?



(a) `abab`
(b) `bbb`
(c) `aaab`
(d) `aaa`
(e) ε

(a, b, c, e)

5. Show a *finite state machine* with input alphabet {`0`,`1`} which accepts any string having an odd number of 1's and an odd number of 0's.

	0	1
A	B	C
B	A	D
C	D	A
*D	C	B

6. Describe, in you own words, the *language* specified by each of the following finite state machines with alphabet {`a`,`b`}.

(a)

	a	b
A	B	A
B	B	C
C	B	D
*D	B	A

All strings with end with `abb`

(b)

	a	b
A	B	A
B	B	C
C	B	D
*D	D	D

All strings containing the substring `abb`

(c)

	a	b
*A	A	B
*B	C	B
C	C	C

All strintgs in which all the b's are preceded by all the a's.

(d)

	a	b
A	B	A
B	A	B
*C	C	B

The empty set.

(e)

	a	b
A	B	B
*B	B	B

All strings except ε

7. Which of the following strings belong to the language specified by this regular expression: `(a+bb)*a`

(a) ε (b) `aaa` (c) `ba`
(d) `bba` (e) `abba`

(b, d, e)

8. Write *regular expressions* to specify each of the languages specified by the finite state machines given in Problem 6.

(a) (a+b)*abb
(b) (a+b)*abb(a+b)*
(c) a*b*
(d) ϕ
(e) (a+b)(a+b)*

9. Construct *finite state machines* which specify the same language as each of the following regular expressions.

(a) `(a+b)*c` (b) `(aa)*(bb)*c`
(c) `(a*b*)*` (d) `(a+bb+c)a*`
(e) `((a+b)(c+d))*`

(a)

	a	b	c
A	A	A	B
*B	C	C	C
C	C	C	C

(b)

	a	b	c
A	B	C	F
B	A	F	F
C	F	D	F
D	F	C	E
*E	F	F	F
F	F	F	F

(c)

	a	b
*A	A	A

(d)

	a	b	c
A	C	B	C
B	D	C	D
*C	C	D	D
D	D	D	D

(e)

	a	b	c	d
*A	B	B	C	C
B	C	C	A	A
C	C	C	C	C

10. Show a string of zeros and ones which is not in the language of the regular expression `(0*1)*`.

00

11. Show a finite state machine which accepts multiples of 3, expressed in binary (ε is excluded from this language).

	0	1
A	B	C
*B	B	C
C	D	B
D	C	D

Exercises 2.1

1. For each of the following Java input strings show the *word boundaries* and *token classes* (for those tokens which are not ignored) selected from the list in Section 2.1.

(a)
```
for  (i=start;  i<=fin+3.5e6;  i=i*3)
          ac=ac+/*incr*/1;
```

```
for (  i    =    start   ;  i   <=   fin   +   3.5e6   ;  i
1   6  2    3     2      6  2   3    2     3     4     6  2

          =  i   *   3   )
          3 2    3   4   6

          ac   =   ac   +   /*incr*/1;
          2    3   2    3
```

(b)
```
{  ax=33;bx=/*if*/31.4  }  //  ax  +  3;
```

```
{ ax   =   33   ;   bx   =   /*if*/   31.4    }
6 2    3   4    6   2    3             4      6
                  //  ax  +  3;
```

(c)
```
if/*if*/a)}+whiles
```

```
if   /*if*/   a   )   }   +   whiles
1             2   6   6   3    2
```

2. Since Java is free format, newline characters are ignored during lexical analysis (except to serve as white space delimiters and to count lines for diagnostic purposes). Name at least two high-level programming languages for which newline characters would not be ignored for syntax analysis.

COBOL, Fortran IV, Basic, APL

3. Which of the following will cause an error message from your Java compiler?

(a) A comment inside a quoted string:
```
"this is /*not*/ a comment"
```

(b) A quoted string inside a comment
```
/*this is "not" a string*/
```

(c) A comment inside a comment
```
/*this is /*not*/ a comment*/
```

(d) A quoted string inside a quoted string
```
"this is "not" a string"
```

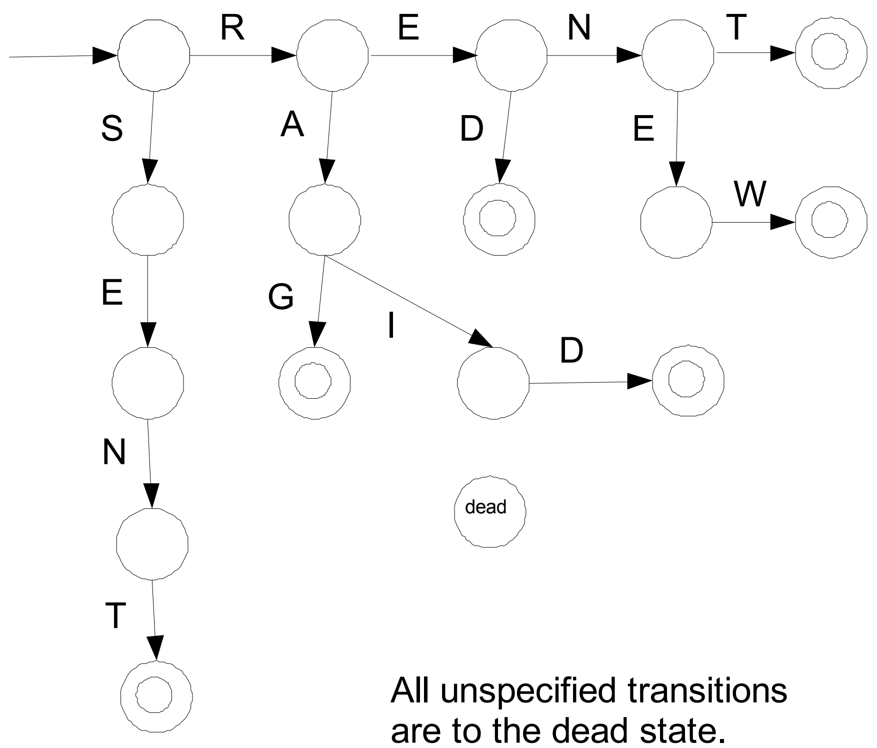
(c, d)

4. Write a Java method to sum the codes of the characters in a given String:

```
public int sum (String s)
{ int total = 0;
 for (i=0; i<s.length(); i++)
     total = total + s.charAt(i);
 return total;
}
```

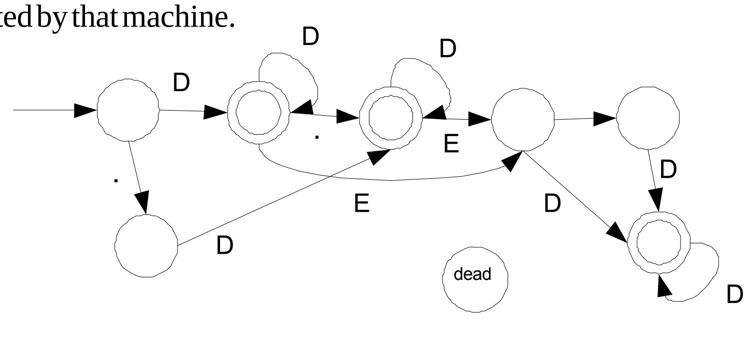
Exercises 2.2

1. Show a *finite state machine* which will recognize the words RENT, RENEW, RED, RAID, RAG, and SENT. Use a different accepting state for each of these words.



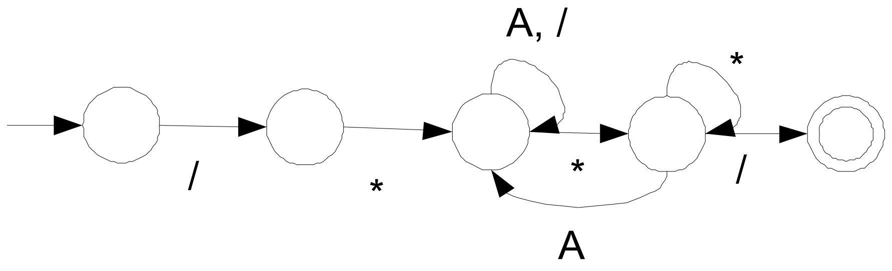
All unspecified transitions are to the dead state.

2. Modify the *finite state machine* of Figure 2.5 to include numeric constants which begin with a decimal point and have digits after the decimal point, such as .25, without excluding any constants accepted by that machine.

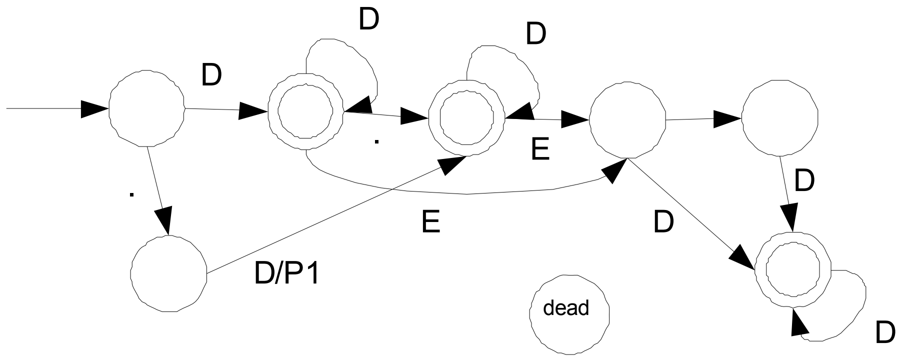


All unspecified transitions are to the dead state.

3. Show a *finite state machine* that will accept C-style comments /* as shown here */. Use the symbol A to represent any character other than * or /; thus the input alphabet will be {/,*,A}.



4. Add *actions* to your solution to Problem 2 so that numeric constants will be computed as in Sample Problem 2.2.



All unspecified transitions are to the dead state.

All other actions are as shown in Sample Problem 2.2.

5. What is the *output* of the finite state machine, below, for each of the following inputs (`L` represents any letter, and `D` represents any numeric digit; also, assume that each input is terminated with a period):

```
int sum;

void P1()                     void P2()
{                             {
   sum = L;                      sum += L;
}                             }

void P3()                     int hash (int n)
{                             {
   sum += D;                     return n % 10;
}                             }

Void P4()
{
System.out.println(hash(sum));
}
```

(a) `ab3.` 6
(b) `xyz.` 3
(c) `a49.` 6

6. Show the *values* that will be asigned to the variable `N` in Sample Problem 2.2 as the input string 46.73e-21 is read.

N = 4, 46, 467, 4673

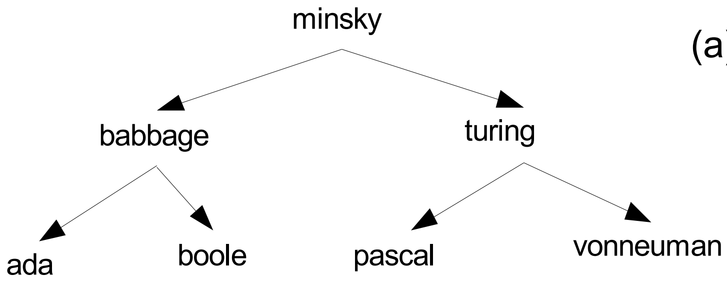
Exercises 2.3

1. Show the *binary search tree* which would be constructed to store each of the following lists of identifiers:

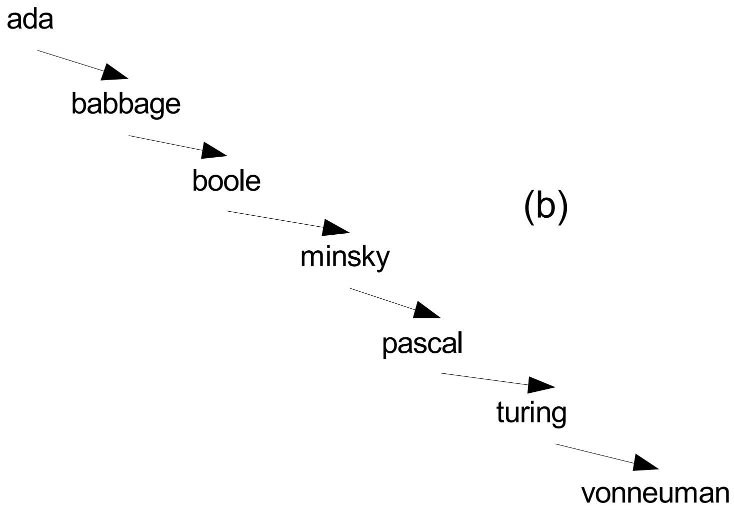
(a) `minsky, babbage, turing, ada, boole, pascal, vonneuman`

(b) `ada, babbage, boole, minsky, pascal, turing, vonneuman`

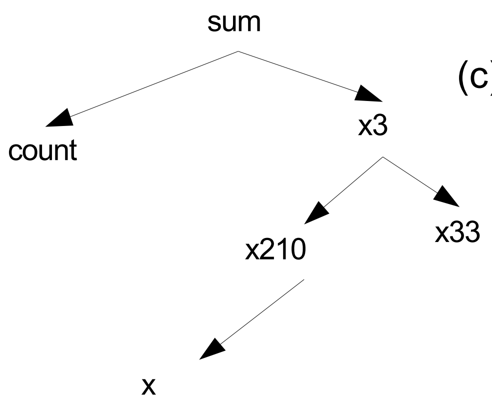
(c) `sum, x3, count, x210, x, x33`


minsky
babbage
turing
ada
boole
pascal
vonneuman

(a)


ada
babbage
boole
minsky
pascal
turing
vonneuman

(b)


sum
count
x3
x210
x33
x

(c)

2. Show how many string comparisons would be needed to store a new identifier in a symbol table organized as a binary search tree containing:

(a) 2047 identifiers, and perfectly balanced
    11 comparisons
(b) 2047 identifiers which had been entered inalphabetic order (worst case)
    2047 comparisons
(c) 2^n-1 identifiers, perfectly balanced
    n comparisons
(d) n identifers, and perfectly balanced
    $\log_2(n+1)$ or $1 + \log_2(n+1)$

3. Write a program in Java which will read a list of words from the keyboard, one word per line. If the word has been entered previously, the output should be `OLD WORD`. Otherwise the output should be `NEW WORD`. Use the following declaration to implement a binary search tree to store the words.

```
public class Node
{  public Node left;
   public String data;
   public Node right;

// constructor
   public Node (String s)
   {  left = right = null;
      data = s;
   }
}
```

```
      Node bst;
```

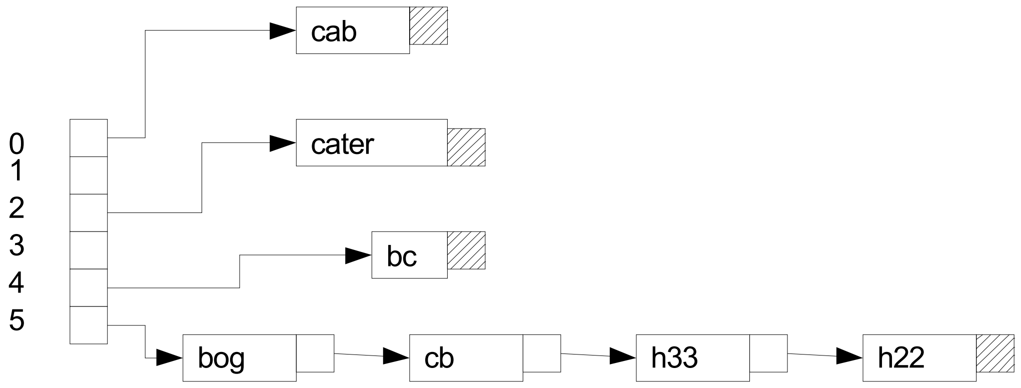
```
// Exercise 2.3.3 from Java edition
// Use binary search tree to determine whether
//   a word has been entered previously.

import java.util.*;

public class Ex2_3_3
{  public static Node bst;

   public static void main (String [] args)
   {  Scanner scanner = new Scanner (System.in);
      String line;

      while (scanner.hasNextLine())
         {  line = scanner.nextLine();
            if (install (line))
               System.out.println ("Old Word");
            else
               System.out.println ("New Word");
         }

    }

   // return true if it's an old word.
   public static boolean install(String line)
   {  Node aNode = bst, prev = bst;
      if (bst==null)
       { bst = new Node (line);
         return false;
       }
     while (aNode!=null)
       {  prev = aNode;
      if (aNode.data.equals (line))
            return true;           // found it
      if (aNode.data.compareTo(line) < 0)
            aNode = aNode.left;
      else
            aNode = aNode.right;
       }

    // install the new word
    aNode = new Node (line);
    if (prev.data.compareTo(line) < 0)
      prev.left = aNode;
    else
```

```
      prev.right = aNode;
    return false;
   }
}
```

4. Many textbooks on data structures implement a hash table as an array of words to be stored, whereas we suggest implementing with an array of linked lists. What is the main advantage of our method? What is the main disadvantage of our method?

Advantage of array of linked lists: Number of words is unlimited
Advantage of array of words: Uses less memory.

5. Show the *hash table* which would result for the following identifiers using the example hash function of Section 2.3.3: `bog, cab, bc, cb, h33, h22, cater`.



6. Show a *single hash function* for a hash table consisting of ten linked lists such that none of the word sequences shown below causes a single collision.

(a) `ab, ac, ad, ae`
(b) `ae, bd, cc, db`
(c) `aa, ba, ca, da`

Sum the ascii codes of the characters in the string, and add the code of the first letter.

```
hash(s) = Σs_i + s_0
```

7. Show a sequence of four *identifiers* which would cause your hash function in Problem 6 to generate a collision for each identifier after the first.

```
az, bx, cv, dt
```

Exercises 2.4

1. Modify the given *SableCC lexing.grammar file and lexing/Lexing.java file* to recognize the following 7 token classes.

(1) Identifier (begins with letter, followed by letters, digits, _)
(2) Numeric constant (float or int)
(3) = (assignment)
(4) Comparison operator (`==   <   >   <=   >=   !=`)
(5) Arithmetic operator (`+   -   *   /`)
(6) String constant `"inside double-quote marks"`
(7) Keyword (`if   else   while   do   for   class`)
Comments

```
/* Using this method */
// or this method, but don't print a token
//   class.
```

lexing.grammar:

```
// lexing.grammar
// Sample SableCC for lexical analysis
// To be used with lexing/Lexing.java
// March 2003,  sdb

Package lexing ;

Helpers
 num = ['0'..'9']+;
 letter = ['a'..'z'] | ['A'..'Z']  ;
 no_quote = [[0..127] - '"']; // anything except a "
 newline = 10 | 13 ;
 no_newline = [[0..127] - 10] | [[0..127] - 13];

States
 start, string, comment;

Tokens
 {string} string = no_quote* ;
 {string->start} q1 = '"';
 {comment->start} comment1 = '*/';
 {comment} comment2 = [0..127];
 {start} number = num '.'? num? (('e' | 'E') ('+' | '-')? num)? ;
 {start} keyword = 'if' | 'else' | 'while' | 'do' | 'for' | 'class';
 {start} ident = letter (letter | num | '_')* ;
 {start} assignment = '=';
 {start} comparison_op = ['<' + '>'] | '==' | '<=' | '>=' | '!=' ;
 {start} arith_op = [ ['+' + '-' ] + ['*' + '/' ] ] ;
 {start->string} q2 = '"' ;
 {start} comment3 = '//' [[0..127] - 10]* ;
 {start->comment} comment = '/*' ;
 {start} blank = (' ' | 9 | 10 | 13)+ ;
 {start} unknown = [0..0xffff] ;

Ignored Tokens
 comment1, comment2, comment3, q1, q2 ;
```

Lexing.java:

```
//  Lexing.java
//  To be used with file lexing.grammar, in parent directory.
//  March 2003,  sdb

package lexing;
import lexing.lexer.*;
import lexing.node.*;
import java.io.*;       // Needed for pushbackreader, inputstream


class Lexing
{

static Lexer lexer;
static Object token;

public static void main(String [] args)
{
    lexer = new Lexer
      (new PushbackReader
           (new InputStreamReader (System.in), 1024));

    token = null;
    try
      {
      while ( ! (token instanceof EOF))
        {   token = lexer.next();         // read next token
            if (token instanceof TIdent)
                  System.out.println ("(1) Identifier:      "
                        + token);
            else if (token instanceof TNumber)
                  System.out.println ("(2) Number:          "
                        + token);
            else if (token instanceof TAssignment)
                  System.out.println ("(3) Assignment:      "
                        + token);
            else if (token instanceof TComparisonOp)
                  System.out.println ("4) Comparison op:    "
                        + token);
            else if (token instanceof TArithOp)
                  System.out.println ("(5) Arithmetic op:   "
                        + token);
            else if (token instanceof TString)
                  System.out.println ("(6) String constant: "
                        + token);
           }
        }
     catch (LexerException le)
      {  System.out.println ("Lexer Exception " + le); }
     catch (IOException ioe)
      {  System.out.println ("IO Exception " +ioe); }
}
}
```

2. Show the sequence of *tokens* recognized by the following definitions for each of the input files below:

```
Helpers
   char = ['a'..'z'] ['0'..'9']? ;
Tokens
   token1 = char char ;
   token2 = char 'x' ;
   token3 = char+ ;
   token4 = ['0'..'9']+ ;
   space = ' ' ;
```

Input files:

(a)
```
a1b2c3
token3
```

(b)
```
abc3 a123
token3 space token3 token4
```

(c)
```
a4x ab r2d2
token2 space token1 space token1
```

Exercises 2.5

1. Extend the SableCC source files for Decaf, decaf.grammar and decaf/Decaf.java to accommodate string constants and character constants (these files can be found at `http://www.rowan.edu/~bergmann/books`). A string is one or more characters inside double-quotes, and a character constant is one character inside single-quotes (do not worry about escape-chars, such as '\n'). Here are some examples, with a hint showing what your lexical scanner should find:

Input	Hint
`"A long string"`	`One string token`
`" Another 'c' string"`	`One string token`
`"one" 'x' "three"`	`A string, a char, a string`
`"  //   string "`	`A string, no comment`
`//  A "comment"`	`A comment, no string`

```
Package decaf;

Helpers                                          // Examples
      letter = ['a'..'z'] | ['A'..'Z'] ;         //   w
      digit =    ['0'..'9'] ;                    //   3
  digits =   digit+ ;                            // 2040099
  exp  =     ['e' + 'E'] ['+' + '-']? digits;         // E-34
  newline = [10 + 13]   ;
  non_star = [[0..0xffff] - '*'];
  non_slash = [[0..0xffff] - '/'];
  non_star_slash = [[0..0xffff] - ['*' + '/']];
  non_quote = [[0..0xffff] - '"'];        // ex 2.5.1
```

```
States
  start, str;     // For exercise 2.5.1

Tokens
  comment1 = '//' [[0..0xffff]-newline]* newline ;
  comment2 = '/*' non_star* '*' (non_star_slash non_star* '*'+)* '/' ;

  // For exercise 2.5.1
  {start->str} quote = '"';   // move to 'str' state.
  {str->start} string_const = non_quote* '"'; // remove the quote in
Translation.java

  space = ' ' | 9 | newline ;       // '\t' (=9) doesn't work?
  clas = 'class' ;      // key words (reserved)
  public = 'public' ;
  static = 'static' ;
  void = 'void' ;
  main = 'main' ;
  string = 'String' ;
  int = 'int' ;
  float = 'float' ;
  for = 'for' ;
  while = 'while' ;
  if = 'if' ;
  else = 'else' ;
  assign = '=' ;
  compare = '==' | '<' | '>' | '<=' | '>=' | '!=' ;
  plus = '+' ;
  minus = '-' ;
  mult = '*' ;
  div = '/' ;
  l_par = '(' ;
  r_par = ')' ;
  l_brace = '{' ;
  r_brace = '}' ;
  l_bracket = '[' ;
  r_bracket = ']' ;
  comma = ',' ;
  semi = ';' ;
  identifier = letter (letter | digit | '_')* ;
  number  =  (digits '.'? digits? | '.'digits) exp? ; // 2.043e+5
  misc = [0..0xffff] ;

Ignored Tokens
```

```
  comment1, comment2, space, quote;
```

2. *Extend* the SableCC source file `decaf.grammar` given at `www.rowan.edu/~bergmann/books` to permit a `switch` statement and a `do while` statement in Decaf:

```
SwitchStmt   →   switch (Expr) { CaseList }
CaseList     →   case NUM : StmtList
CaseList     →   case default: StmtList
CaseList     →   case NUM : StmtList  CaseList
Stmt         → break ;

DoStmt   →   do Stmt while ( Expr )
```

Show the necessary changes to the Tokens section only.

Tokens
...
    switch = 'switch';
    case = 'case';
    default = 'default';
    break = 'break';
    do = 'do';
    colon = ':';
...

3. Revise the *token definition* of the number token in `decaf.grammar` to exclude numeric constants which do not begin with a digit, such as `.25` and `.03e-4`. Test your solution by running the software.

```
number  =  digits '.'? digits exp? ;      // 2.043e+5
```

4. Rather than having a separate token class for each Decaf keyword, the scanner could have a single class for all keywords. Show the changes needed in the file `decaf.grammar` to do this.

```
keyword = 'class' |
  'public' |
   'static' |
      'void' |
      'main' |
      'String' |
      'int' |
      'float' |
      'for' |
      'while' |
      'if' |
      'else' ;
```

1. Show three different *derivations* using each of the following grammars, with starting nonterminal `S`.

(a)

```
S → a S
S → b A
A → b S
A → c
```

```
S ⇒ b A ⇒ b c
S ⇒ b A ⇒ b S ⇒ b b A ⇒ b b c
S ⇒ a S ⇒ a b A ⇒ a b c
```

(b)

```
S → a B c
B → A B
A → B A
A → a
B → ε
```

```
S ⇒ a B c ⇒ a c
S ⇒ a B c ⇒ a A B c ⇒ a A c ⇒ a a c
S ⇒ a B c ⇒ a B A B c ⇒ a A B c ⇒ a a B c ⇒ a a c
```

(c)

```
S     → a S B c
a S A →  a S b b
B c   → A c
S b   → b
A     → a
```

```
S ⇒ a S B c ⇒ a S A c ⇒ a S b b c ⇒ a b b c
S ⇒ a S B c ⇒ a S A c ⇒ a a S B c A c
  ⇒ a a S B c a c ⇒ a a S A c a c ⇒ a a S b b c a c
  ⇒ a a b b c a c
S ⇒ a S B c ⇒ a S A c ⇒ a a S B c A c
  ⇒ a a S A c A c ⇒ a a S b b c A c ⇒ a a b b c A c
  ⇒ a a b b c a c
```

(d)
```
S     →  a b
a     → a A b B
A b B → ε

S ⇒ a b
S ⇒ a b ⇒ a A b B ⇒ a
S ⇒ a b ⇒ a A b B ⇒ a A b B A b B ⇒ a A b B ⇒ a
```

2. Classify the grammars of Problem 1 according to *Chomsky's definitions* (give the most restricted classification applicable).

(a) type 3, Right Linear
(b) type 2, Context-Free
(c) type 1, Context-Sensitive
(d) type 0, Unrestricted

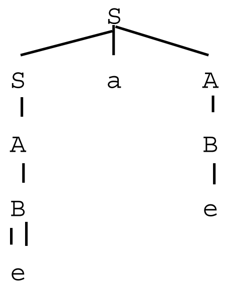
3. Show an example of a grammar *rule* which is:

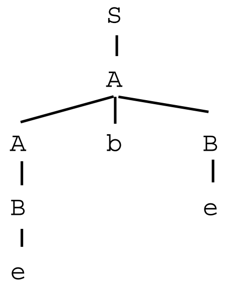
(a)	Right Linear	A →a
(b)	Context-Free, but not Right Linear	A →A b b
(c)	Context-Sensitive, but not Context-Free	a A B → a c B
(d)	Unrestricted, but not Context-Sensitive	a A B → a

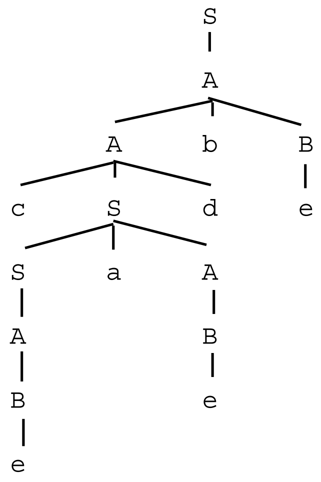
4. For each of the given input strings show a *derivation tree* using the following grammar.

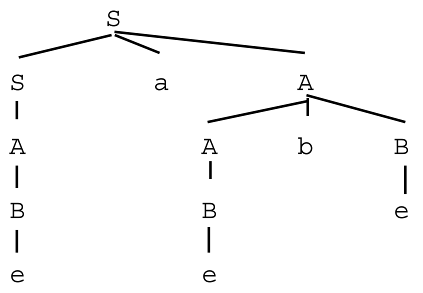
```
1.  S  →  S a A
2.  S  →  A
3.  A  →  A b B
4.  A  →  B
5.  B  →  c S d
6.  B  →  e
7.  B  →  f
```

(a) `eae` (b) `ebe` (c) `eaebe`
(d) `ceaedbe` (e) `cebedaceaed`

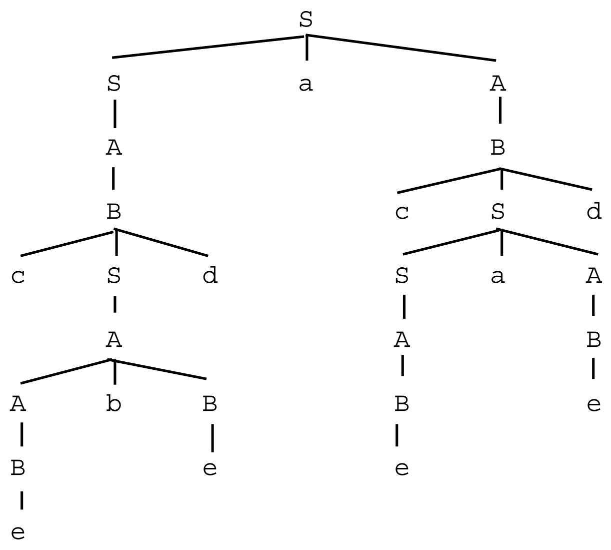

(a)


(b)


(d) (oops)



(c)



(e)

5. Show a *left-most derivation* for each of the following strings, using grammar G4 of Section 3.0.3.

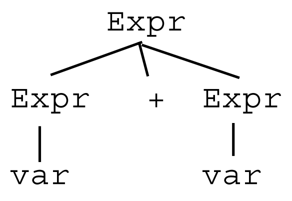
(a)	`var + const`	(b)	`var + var * var`
(c)	`(var)`	(d)	`( var + var ) * var`

(a)

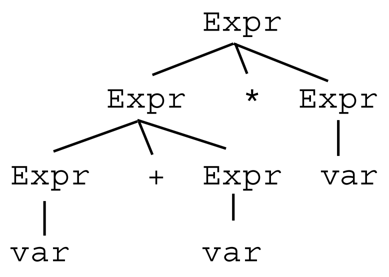
$$\texttt{Expr} \Rightarrow \texttt{Expr + Expr} \Rightarrow \texttt{var + Expr} \Rightarrow \texttt{var + const}$$

```
(b)  Expr ⇒ Expr * Expr ⇒ Expr + Expr * Expr
          ⇒ var + Expr * Expr ⇒ var + var * Expr
          ⇒ var + var * var

(c)  Expr ⇒ ( Expr ) ⇒ ( var )

(d)  Expr ⇒ Expr * Expr ⇒ ( Expr ) * Expr
          ⇒ ( Expr + Expr ) * Expr
          ⇒ ( var + Expr ) * Expr
          ⇒ ( var + var ) * Expr
          ⇒ ( var + var ) * var
```

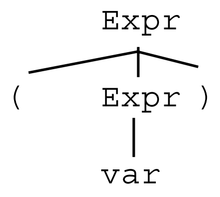
6. Show *derivation trees* which correspond to each of your solutions to Problem 5.



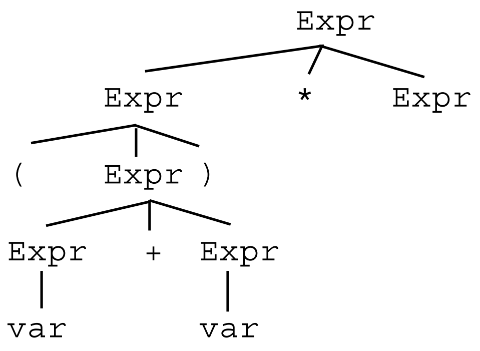
(a)



(b)



(c)



(d)

7. Some of the following grammars may be ambiguous; for each ambiguous grammar, show two different *derivation trees* for the same input string:

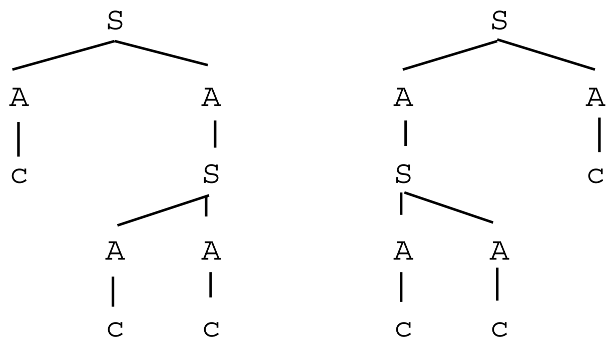
(a)
```
1.   S → a S b
2.   S →  A A
3.   A → c
4.   A → S
```

(b)
```
1.   S → A a A
2.   S → A b A
3.   A → c
4.   A → S
```

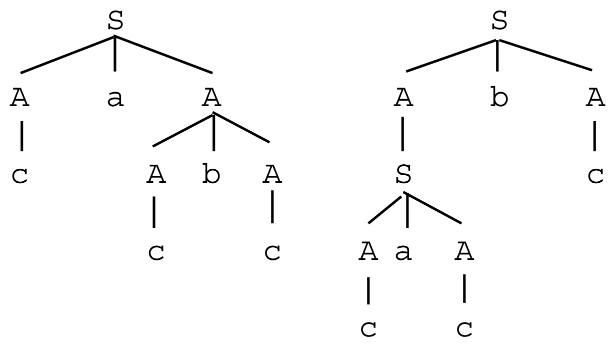
(c)
```
1.   S → a S b S
2.   S → a S
3.   S → c
```

(d)
```
1.   S → a S b c
2.   S → A B
3.   A → a
4.   B → b
```

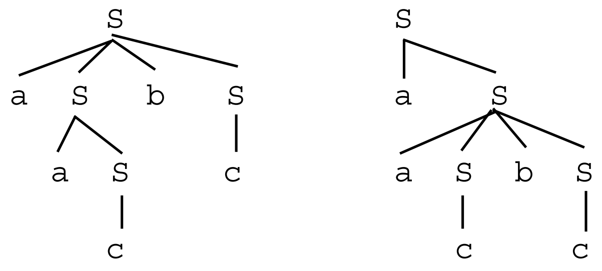
(a)



(b)



(c)



(d) Not ambiguous

8. Show a *pushdown machine* that will accept each of the following languages:

(a) $\{a^n b^m\}$ $m>n>0$ (b) $a^*(a+b)c^*$

(c) $\{a^n b^n c^m d^m\}$ $m,n \geq 0$ (d) $\{a^n b^m c^m d^n\}$ $m,n > 0$

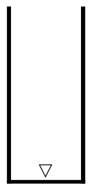
(e) $\{N_i c (N_{i+1})^r\}$

– where N_i is the binary representation of the integer `i`, and $(N_i)^r$ is N_i written right to left (reversed). Example for `i=19: 10011c00101`

Hint: Use the first state to push N_i onto the stack until the `c` is read. Then use another state to pop the stack as long as the input is the complement of the stack symbol, until the top stack symbol and the input symbol are equal. Then use a third state to ensure that the remaining input symbols match the symbols on the stack.

S1	a	b	↩
X	Push (X) Advance S1	Advance S2	Reject
▽	Push (X) Advance S1	Reject	Reject

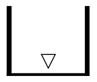
S2	a	b	↩
X	Reject	pop Advance S2	Reject
▽	Reject	Advance S2	Accept



Initial
Stack

(a)

S1	a	b	c	↩
▽	Advance S2	Advance S3	Reject	Reject



Initial
Stack

S3	a	b	c	↩
▽	Reject	Reject	Advance S3	Accept

S2	a	b	c	↩
▽	Advance S2	Advance S3	Advance S3	Accept

(b)

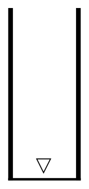
S1	a	b	c	d	↩
X	Push (X) Advance S1	pop Advance S2	Reject	Reject	Reject
▽	Push (X) Advance S1	Reject	push(X) Advance S3	Reject	Accept

S2	a	b	c	d	↩
X	Reject	pop Advance S2	Reject	Reject	Reject
▽	Reject	Reject	push(X) Advance S3	Reject	Accept

S3	a	b	c	d	↩
X	Reject	Reject	push(X) Advance S3	pop Advance S4	Reject
▽	Reject	Reject	Reject	Reject	Accept

S4	a	b	c	d	↩
X	Reject	Reject	Reject	pop Advance S4	Reject
▽	Reject	Reject	Reject	Reject	Accept

(c)

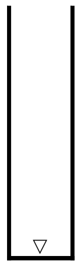


Initial Stack

S1	a	b	c	d	↩
N	Push (N) Advance S1	pop Advance S2	Reject	Reject	Reject
M	Reject	Reject	Reject	Reject	Reject
▽	Push (N) Advance S1	Reject	Reject	Reject	Reject

S2	a	b	c	d	↩
N	Reject	pop Advance S2	Reject	Reject	Reject
M	Reject	Reject	Push (M) Advance S2	pop Advance S3	Reject
▽	Push (N) Advance S1	Reject	Push (M) Advance S2	Reject	Reject

S3	a	b	c	d	↩
N	Reject	Reject	Reject	Reject	Reject
M	Reject	Reject	Reject	pop Advance S3	Reject
▽	Reject	Reject	Reject	Reject	Accept

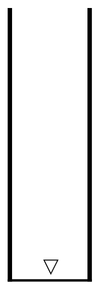


Initial Stack

(d)

S1	0	1	c	↩
0	Push (0) Advance S1	push (1) Advance S1	Advance S2	Reject
1	Push (0) Advance S1	push (1) Advance S1	Advance S2	Reject
▽	Push (0) Advance S1	push (1) Advance S1	Advance S2	Reject

S2	0	1	c	↩
0	Reject	pop Advance S3	Reject	Reject
1	pop Advance S2	Reject	Reject	Reject
▽	Reject	Reject	Reject	Accept

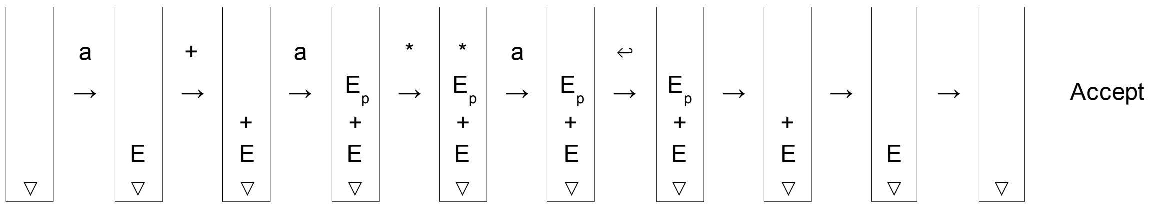

Initial
Stack

(e)

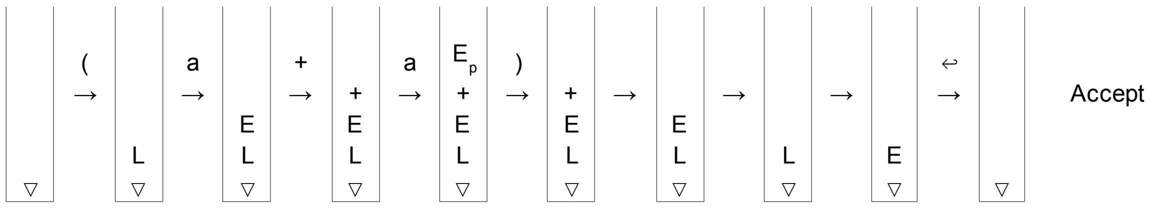
9. Show the *output* and the *sequence of stacks* for the machine of Figure 3.8 for each of the following input strings:

(a)	`a+a*aN`	(b)	`(a+a)*aN`
(c)	`(a)N`	(d)	`((a))N`

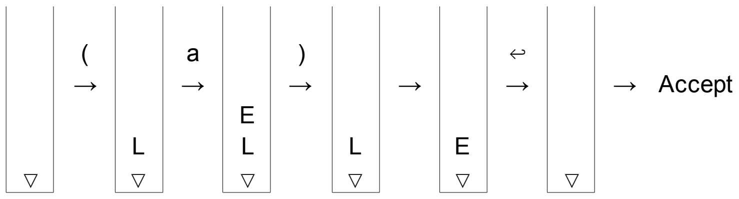
(a)
aaa*+



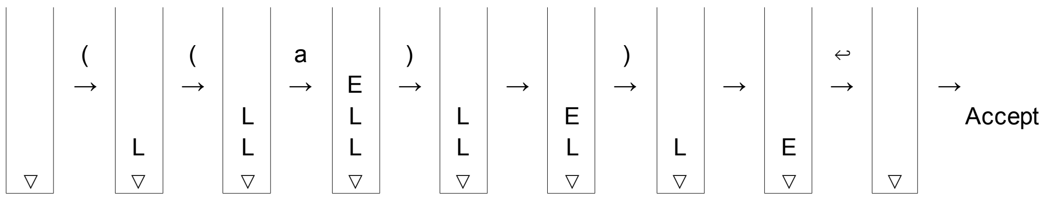
(b)
aa+a*



(c)
a



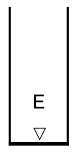
(d)
a



10. Show a *grammar* and *an extended pushdown machine* for the language of prefix expressions involving addition and multiplication. Use the terminal symbol `a` to represent a variable or constant. Example: `*+aa*aa`

```
1.   E → + E E
2.   E → * E E
3.   E → a
```

S1	a	+	*	↩
E	pop	push (E)	push (E)	Reject
▽	Reject	push (E)	push (E)	Accept

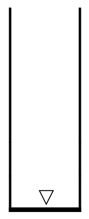


Initial Stack

11. Show a *pushdown machine* to accept palindromes over {`0`,`1`} with centermarker `c`. This is the language, P_c, referred to in Section 3.0.5.

S1	0	1	c	↩
0	push(0) S1	push(1) S1	S2	Reject
1	push(0) S1	push(1) S1	S2	Reject
▽	push(0) S1	push(1) S1	S2	Reject

S2	0	1	c	↩
0	pop S2	Reject	Reject	Reject
1	Reject	pop S2	Reject	Reject
▽	Reject	Reject	Reject	Accept



Initial Stack

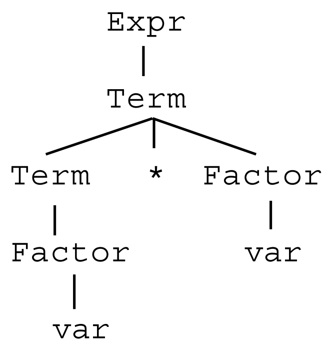
12. Show a *grammar* for the language of valid regular expressions over the alphabet {`0,1`}. Hint: Think about grammars for arithmetic expressions.

```
1.   E → E + E
2.   E → E · E
3.   E → E *
4.   E → ( E )
5.   E → 0
6.   E → 1
```

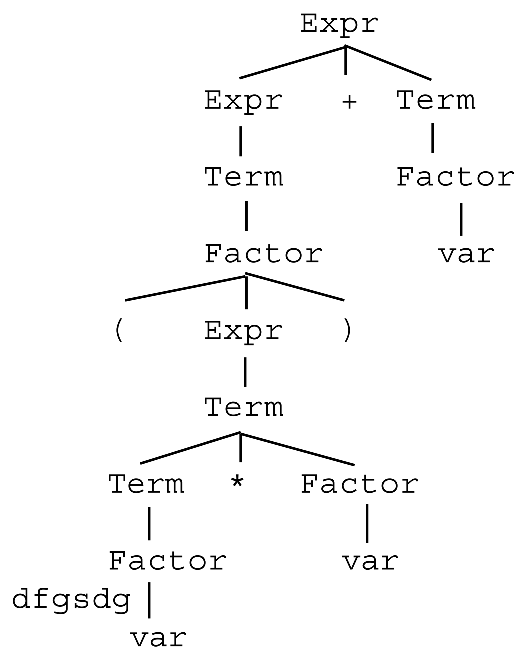
Exercises 3.1

1. Show *derivation trees* for each of the following input strings using grammar G5.

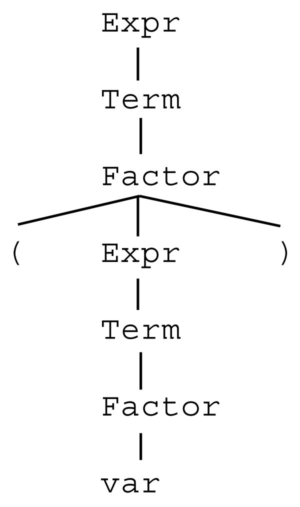
(a) `var * var` (b) `( var * var ) + var`
(c) `(var)` (d) `var * var * var`



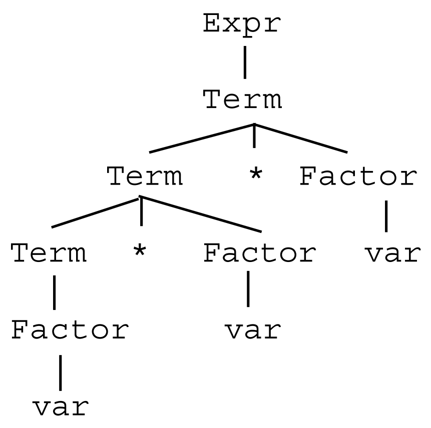
(a)



(b)



(c)



(d)

2. Extend *grammar G5* to include subtraction and division so that subtrees of any derivation tree correspond to subexpressions.

```
1.5  Expr → Expr - Term
3.5  Term → Term / Factor
```

3. Rewrite your grammar from Problem 2 to include an *exponentiation operator*, ^, such that `x^y` is x^y. Again, make sure that subtrees in a derivation tree correspond to subexpressions. Be careful, as exponentiation is usually defined to take precedence over multiplication and associate to the right: 2*3^2 = 18 and 2^2^3 = 256.

```
1.   Expr → Expr + Term
2.   Expr → Expr - Term
3.   Expr → Term
4.   Term → Term * Factor
5.   Term → Term / Factor
6.   Term → Factor
7.   Factor → Primary ^ Factor
8.   Factor → Primary
9.   Primary → ( Expr )
10.  Primary → var
11.  Primary → const
```

4. Two grammars are said to be *isomorphic* if there is a one-to-one correspondence between the two grammars for every symbol of every rule. For example, the following two grammars are seen to be isomorphic, simply by making the following substitutions: substitute `B` for `A`, `x` for `a`, and `y` for `b`.

```
S  →  a A b            S  →  x B y
A  →  b A a            B  →  y B x
A  →  a                B  →  x
```

Which grammar in Section 3.1 is *isomorphic* to the grammar of Problem 4 in Section 3.0?

Grammar G5

5. How many different *derivation trees* are there for each of the following `if` statements using grammar G6?

(a) `if ( Expr ) OtherStmt`
(b) `if ( Expr ) OtherStmt else if ( Expr ) OtherStmt`
(c) `if ( Expr ) if ( Expr ) OtherStmt else Stmt else OtherStmt`
(d) `if ( Expr ) if ( Expr ) if ( Expr ) Stmt else OtherStmt`

(a) 1
(b) 1
(c) 1
(d) 3

6. In the original C language it is possible to use assignment operators: `var =+ expr` means `var = var + expr` and `var =- expr` means `var = var - expr.` In later versions of C, C++, and Java the operator is placed before the equal sign:

`var += expr` and `var -= expr.`

Why was this change made?

var =-expr written without spaces is ambiguous. It could be interpreted either as an assignment operator or a unary minus operating on the expression.

Chapter 4

Exercises 4.0

1. Show the *reflexive transitive closure* of each of the following relations:

(a)	`(a,b)`	(b)	`(a,a)`	(c)	`(a,b)`
	`(a,d)`		`(a,b)`		`(c,d)`
	`(b,c)`		`(b,b)`		`(b,c)`
					`(d,a)`
	`(a,b)`		`(a,a)`		`(a,b)`
	`(a,d)`		`(a,b)`		`(b,c)`
	`(b,c)`		`(b,b)`		`(c,d)`
	`(a,c)`				`(d,a)`
	`(a,a)`				`(a,c)`
	`(b,b)`				`(c,a)`
	`(c,c)`				`(b,d)`
	`(d,d)`				`(d,b)`
					`(d,c)`
					`(a,d)`
					`(c,b)`
					`(b,a)`
					`(a,a)`
					`(b,b)`
					`(c,c)`
					`(d,d)`

2. The mathematical relation "less than" is denoted by the symbol <. Some of the elements of this relation are: `(4,5) (0,16) (-4,1) (1.001,1.002).` What do we normally call the relation which is the reflexive transitive closure of "less than"?

less than or equal to

3. Write a program in Java or C++ to read in from the keyboard, ordered pairs (of strings, with a maximum of eight characters per string) representing a relation, and print out the *reflexive transitive closure* of that relation in the form of ordered pairs. You may assume that there will be, at most, 100 ordered pairs in the given relation, involving, at most, 100 different symbols. (Hint: Form a *boolean matrix* which indicates whether each symbol is related to each symbol).

```
#include <iostream.h>
#include "ourstr.h"
const int Max = 200;
int table[Max] [Max];           // boolean closure table
string pairs[Max] [2];          // input relation
string symbols[2*Max];          // sorted symbols in relation
int n,np;
void inp ();
void sort ();
void squish ();
int index (string symbol);
void init ();
void trans ();
void reflex ();
void out ();
void dump ();

void main ()
// find the reflexive transitive closure of a relation
        {       inp ();         // read in the ordered pairs of symbols
                sort ();  // sort the symbols in a 1 dimensional array
                squish ();      // delete duplicated symbols
                init ();  // initialize the two dimensional boolean table
                trans (); // fill in transitive entries
                reflex ();      // fill in reflexive entries
                out ();         // write out the reflexive transitive closure
        }
```

```
void inp ()
// read ordered pairs into the array pairs[Max][2]
        {       n = 0;
                cout <<
        "Enter an ordered pair on each line, separated with a space"
                << endl;
                while (! cin.eof())
                        {       cin >> pairs [n][0];
                                cin >> pairs [n][1];
                                n++;
                        }
                cout << endl;
                n--;
        }

void swap (string & s1, string & s2)
        {       string temp;
                temp = s1;
                s1 = s2;
                s2 = temp;
        }

void dump()
{  cout << "number of pairs is " << np << endl;
   cout << "number of symbols is " << n << endl;
   cout << "array of symbols is " << endl;
   for (int i=0; i<n; i++) cout << symbols[i] << endl;
}

void sort ()
// copy the symbols in the pairs array to the array symbols [2*Max]
// and sort alphabetically
        {       int i,j;
                for (i=0; i<n; i++)
                        {       symbols [2*i] = pairs [i][0];
                                symbols [2*i+1] = pairs [i][1];
                        }
```

```
		np = n;		// number of pairs
		n = 2*n;
		for (i=0; i<n-1; i++)
			for (j=i+1; j<n; j++)
				if (symbols[i]>symbols[j])
					swap (symbols[i], symbols[j]);
	}

void squish ()
// eliminate duplicate entries in the array symbols[n]
	{	int i=0, j=1;
		while (j<n)
			{	while (symbols[i]==symbols[j] && j<n) j++;
				symbols[i+1] = symbols[j];
				j++;
				i++;
			}
		if (symbols[i-1]==symbols[i]) n = i;
		else n = i+1;
	}

int index (string symbol)
// return a subscript to the array symbols[n], given a symbol in the array.
// this is now associative storage.
// a binary search is used.
	{	int top=0, bot=n-1, mid=(top+bot) / 2;
		while (symbols[mid]!=symbol)
			{	if (symbols[mid]<symbol) top = mid+1;
				else bot = mid-1;
				mid = (top+bot) / 2;
			}
		return mid;
	}

const int True = 1;
const int False = 0;
void init ()
```

```
// initialize the boolean array table[n][n] to True for all pairs in the
// given relation. Table[i][j] is True iff symbols[i] is related to
// symbols[j].
        {       int i,j;
                for (i=0; i<n; i++)         // initialize entire array to False
                  for (j=0; j<n; j++) table[i][j] = False;
                for (i=0; i<np; i++)
                 table [index (pairs[i][0])] [index (pairs[i][1])] = True;
        }

void trans ()
// fill in the transitive entries in table[n][n].
// If table[i][j] and table[j][k]
// then table table[i][k].
        {       int i,j,k,done=False;
                while (!done)
                        {       done = True;    // done iff no more entries
                                for (i=0; i<n; i++)
                                 for (j=0; j<n; j++)
                                  if (table[i][j]) for (k=0; k<n; k++)
                                        if (table[j][k] && !table[i][k])
                                         {      table[i][k] = True;
                                                done = False;
                                         }
                        }
        }

void reflex ()
// fill in the reflexive entries in table[n][n]
        {       int i;
                for (i=0; i<n; i++) table [i][i] = True;
        }

void out ()
// write out the resulting relation as ordered pairs
        {       int i,j;
                for (i=0; i<n; i++)
```

```
          for (j=0; j<n; j++)
                if (table[i][j])
                  cout << symbols[i] << ' '
                    << symbols[j] << endl;
}
```

Exercises 4.1

1. Determine which of the following grammars are s*imple*. For those which are simple, show an *extended one-state pushdown machine* to accept the language of that grammar.

(a)
```
1.   S  →  a S b
2.   S  →  b
```

(b)
```
1.   Expr  →  Expr + Term
2.   Expr  →  Term
3.   Term  →  var
4.   Term  →  ( Expr )
```

(c)
```
1.   S  →  a A b B
2.   A  →  b A
3.   A  →  a
4.   B  →  b A
```

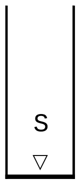
(d)
```
1.   S  →  a A b B
2.   A  →  b A
3.   A  →  b
4.   B  →  b A
```

(e)
```
1.   S  →  a A b B
2.   A  →  b A
3.   A  →  ε
4.   B  →  b A
```

```
(b,d,e) are not simple
```

(a)

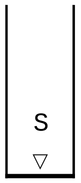
S1	a	b	↩
S	Rep (bSa) Retain	Rep (b) Retain	Reject
a	pop adv	Reject	Reject
b	Reject	pop adv	Reject
▽	Reject	Reject	Accept



Initial Stack

(c)

S1	a	b	↩
S	Rep (BbAa) Retain	Reject	Reject
A	Rep(a) Retain	Rep(Ab) Retain	Reject
B	Reject	Rep(Ab) Retain	Reject
a	pop adv	Reject	Reject
b	Reject	pop adv	Reject
▽	Reject	Reject	Accept

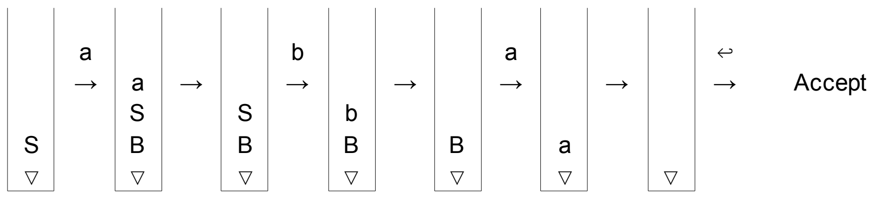


Initial Stack

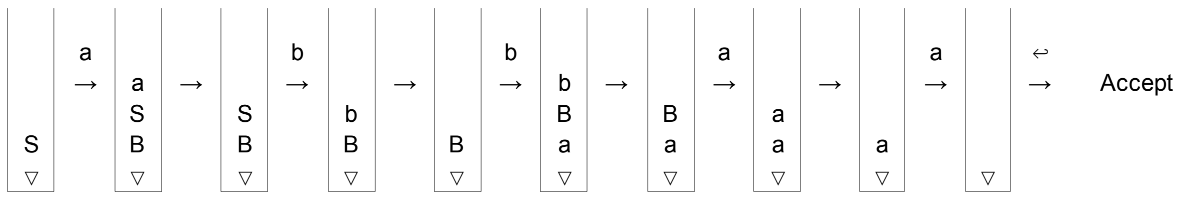
2. Show the *sequence of stacks* for the pushdown machine of Figure 4.5 for each of the following input strings:

(a) `abaN` (b) `abbaaN` (c) `aababaaN`

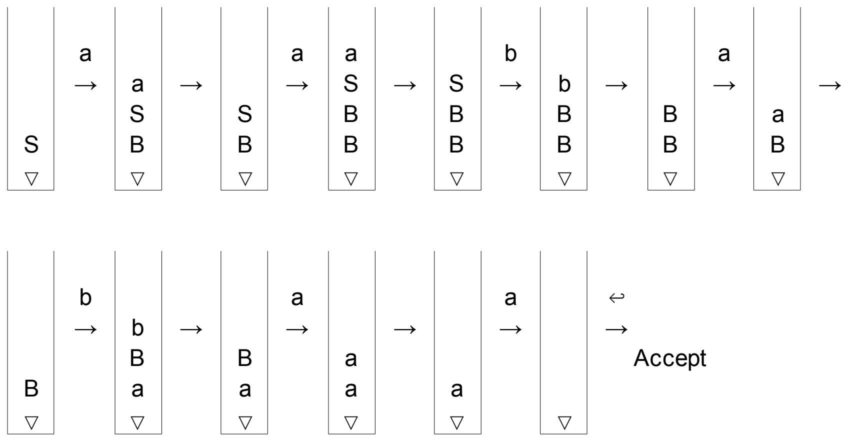
(a)
aba



(b)
abbaa



(c)
aababaa



3. Show a *recursive descent parser* for each simple grammar of Problem 1, above.

(a)

```
void s()
{    if (inp=='a')            // rule 1
       { inp = getInp();
         s();
         if (inp=='b')
           inp = getInp();
         else
           reject();
       }
     else if (inp=='b')       // rule 2
       inp getInp();
     else reject();
}
```

(c)

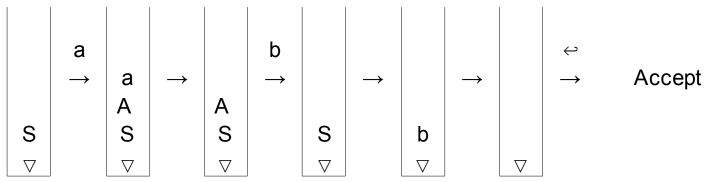
```
void s()
{    if (inp=='a')            // rule 1
       { inp = getInp();
         A();
         if (inp=='b')
           inp = getInp();
         else reject();
         B();
        }
}

void A()
{    if (inp=='b')            // rule 2
       { inp = getInp();
         A();
       }
     else if (inp=='a')       // rule 3
         inp = getInp();
     else reject();
}
```

Exercises 4.2

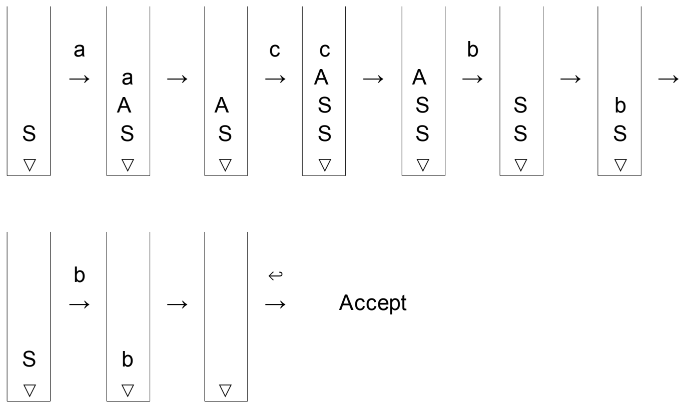
1. Show the *sequence of stacks* for the pushdown machine of Figure 4.8 for each of the following input strings:

(a) `ab`N (b) `acbb`N (c) `aab`N

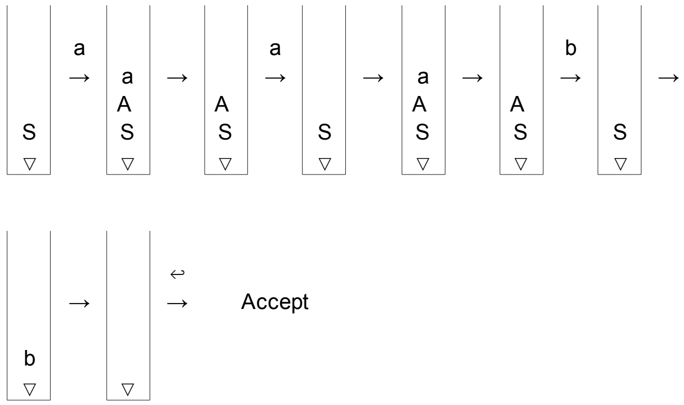
(a)
ab



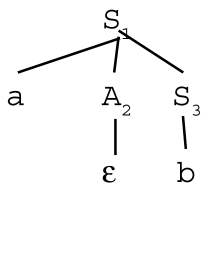
(b)
acbb



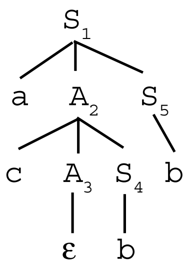
(c)
aab



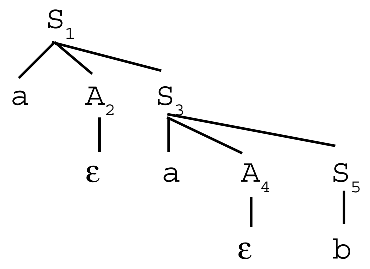
2. Show a *derivation tree* for each of the input strings in Problem 1, using grammar G14. Number the nodes of the tree to indicate the sequence in which they were applied by the pushdown machine.



(a)



(b)



(c)

3. Given the following grammar:

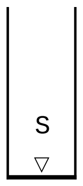
```
1.   S  → a A b S
2.   S  → ε
3.   A  → a S b
4.   A  → ε
```

(a) Find the *follow set* for each nonterminal.

Fol(S) = {b,↩}
Fol(A) = {b}

(b) Show an *extended pushdown machine* for the language of this grammar.

S1	a	b	↩
S	Rep (SbAa) Retain	pop Retain	pop Retain
A	Rep (bSa) adv	pop Retain	Reject
a	pop adv	Reject	Reject
b	Reject	pop adv	Reject
▽	Reject	Reject	Accept



Initial Stack

(c) Show a *recursive descent parser* for this grammar.

```
void s()
{  if (inp=='a')                   // rule 1
     {  inp = getInp();
        A();
        if (inp=='b')
          inp = getInp();
        else
          reject();
        S();
     }
   else
     if (inp=='b' || inp=='↩')     // rule 2
        ;
     else
          reject();
}
```

```
void A()
{ if (inp=='a')                           // rule 3
  {  inp = getInp();
     S();
     if (inp=='b')
       inp = getInp();
     else
       reject();
  }
  if (inp == 'b')                         // rule 4
     ;
  else
     reject();
}
```

Exercises 4.3

1. Given the following information, find the *Followed By* relation `(FB)` as described in step 9 of the algorithm for finding selection sets:

```
         A EO A    A FDB D   D BW b
         A EO B    B FDB a   b BW b
         B EO B              a BW a

    A FB b
    A FB a
    B FB a
```

2. Find the *selection sets* of the following grammar and determine whether it is LL(1).

```
1.   S → ABD
2.   A → aA
3.   A → ε
4.   B → bB
5.   B → ε
6.   D → dD
7.   D → ε
```

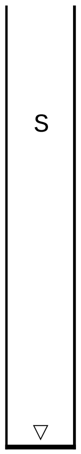
```
Step 1  Nullable rules:  1,3,5,7
       Nullable nonterminals, S, A, B, D

Step 2    S BDW A
          S BDW B
          S BDW D
          A BDW a
          B BDW b
          D BDW d

Step 3    S BW A
          S BW B
          S BW D
          A BW a
          B BW b
          D BW d

          S BW a
          S BW b
          S BW d

          S BW S
          A BW A
          B BW B
          D BW D
          a BW a
          b BW b
          d BW d

Step 4    First(S)  =    {a, b, d}
          First(A)  =    {a}
          First(B)  =    {b}
          First(D)  =    {d}
          First(a)  =    {a}
          First(b)  =    {b}
          First(d)  =    {d}
```

```
Step 5    1.   First (ABD) =   {a, b, d}
          2.   First (aA) =    {a}
          3.   First (ε) =     {}
          4.   First (bB) =    {b}
          5.   First (ε) =     {}
          6.   First (dD) =    {d}
          7.   First (ε) =     {}

Step 6    A FDB B
          A BDB D
          B FDB D
          a FDB A
          b FDB B
          d FDB D

Step 7    D DEO S
          B DEO S
          A DEO S
          A DEO A
          B DEO B
          D DEO D

Step 8    D DO S
          B DO S
          A DO S
          A DO A
          B DO B
          D DO D

          S EO S

Step 9    A  EO  A  FDB  B  BW  b      A FB b
          A  EO  A  FDB  D  BW  d      A FB d
          B  EO  B  FDB  D  BW  d      B FB d
```

```
Step 10   D FB ↩
          B FB ↩
          A FB ↩
          S FB ↩

Step 11   Fol(S) = {↩}
          Fol(A) = {b,d,↩}
          Fol(B) = {d,↩}
          Fol(D) = {↩}

Step 12
          Sel(1) = {a,b,d} U {↩} = {a,b,d,↩}
          Sel(2) =      {a}
          Sel(3) =      {b,d,↩}
          Sel(4) =      {b}
          Sel(5) =      {d,↩}
          Sel(6) =      {d}
          Sel(7) =      {↩}

Yes, the grammar is LL(1)
```

3. Show a *pushdown machine* for the grammar of Problem 2.

S1	a	b	d	↩
S	Rep (DBA) Retain	Rep (DBA) Retain	Rep (DBA) Retain	Rep (DBA) Retain
A	Rep (Aa) Retain	pop Retain	pop Retain	pop Retain
B	Reject	Rep (Bb) Retain	pop Retain	pop Retain
D	Rep (bSa) adv	pop Retain	pop Retain	pop Retain
a	pop adv	Reject	Reject	Reject
b	Reject	pop adv	Reject	Reject
d	Reject	Reject	pop adv	Reject
▽	Reject	Reject	Reject	Accept



4. Show a *recursive descent parser* for the grammar of Problem 2.

```
void S()
{   if (inp=='a' || inp=='b' || inp=='d'
             || inp=='↩')              // rule 1
      {  A();
         B();
         D();
      }
    else
      reject();
}

void A()
{    if (inp=='a')                     // rule 2
        {  inp = getInp();
           A();
        }
      else
```

```
   if (inp=='b' || inp=='d'
      || inp=='↩')               // rule 3
      ;
   else
      reject();
}

void B()
{   if (inp=='b')                // rule 4
      {   inp = getInp();
          B();
      }
    else
      if (inp=='d' || inp=='↩')     // rule 5
           ;
      else
           reject();
}

void D()
{   if (inp=='d')                // rule 6
      {   inp = getInp();
          D();
      }
    else
      if (inp=='↩')              // rule 7
          ;
      else
           reject();
}
```

5. Step 3 of the algorithm for finding selection sets is to find the "Begins With" relation by forming the reflexive transitive closure of the "Begins Directly With" relation. Then add "pairs of the form `a BW a` for each terminal `a` in the grammar"; i.e., there could be terminals in the grammar which do not appear in the BDW relation. Find an example of a grammar in which the selection sets will not be found correctly if you don't add these pairs to the BW relation (hint: see step 9).

```
1.   S  →  A a
2.   A  →  b N
3.   N  →  ε
```

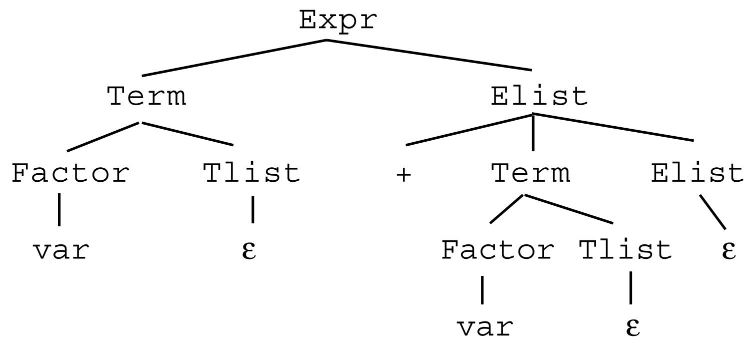
Sel(3) should be {a}, but will not be found correctly since **a BW a** is missing.

Exercises 4.4

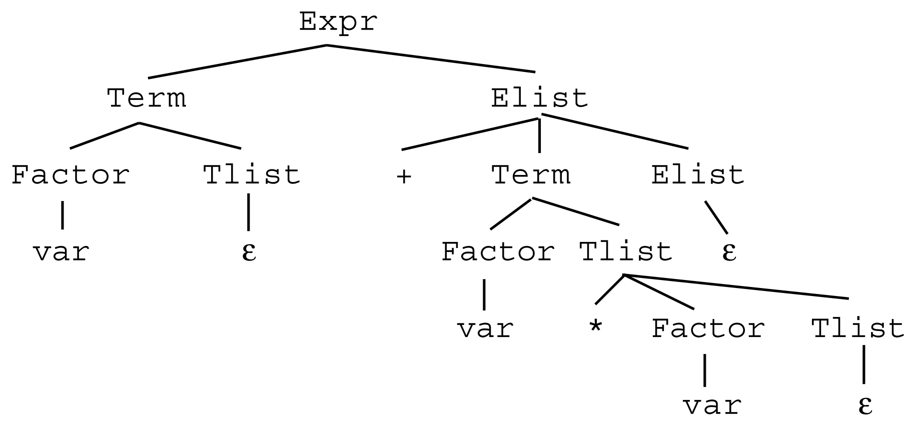
1. Show *derivation trees* for each of the following input strings, using grammar G16.

(a)	`var + var`	(b)	`var + var * var`
(c)	`(var + var) * var`	(d)	`((var))`
(e)	`var * var * var`		

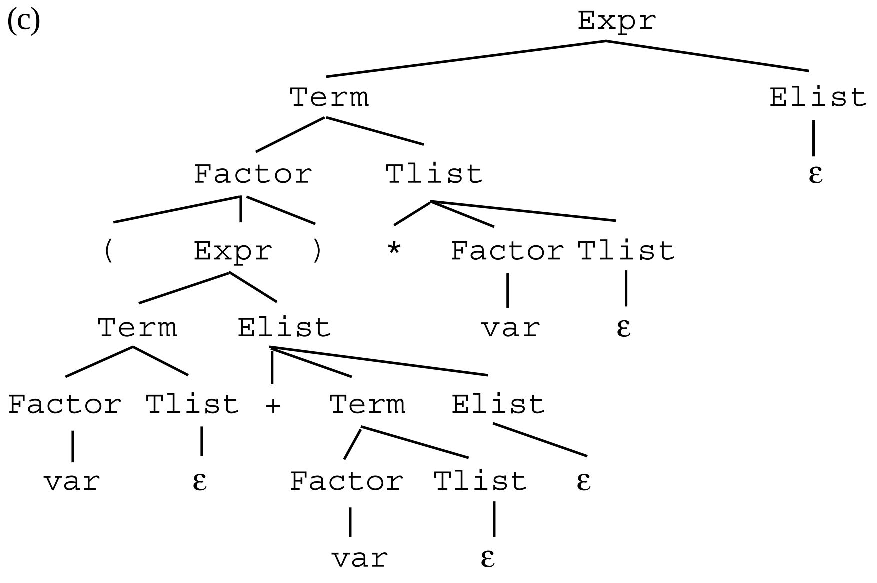
(a)



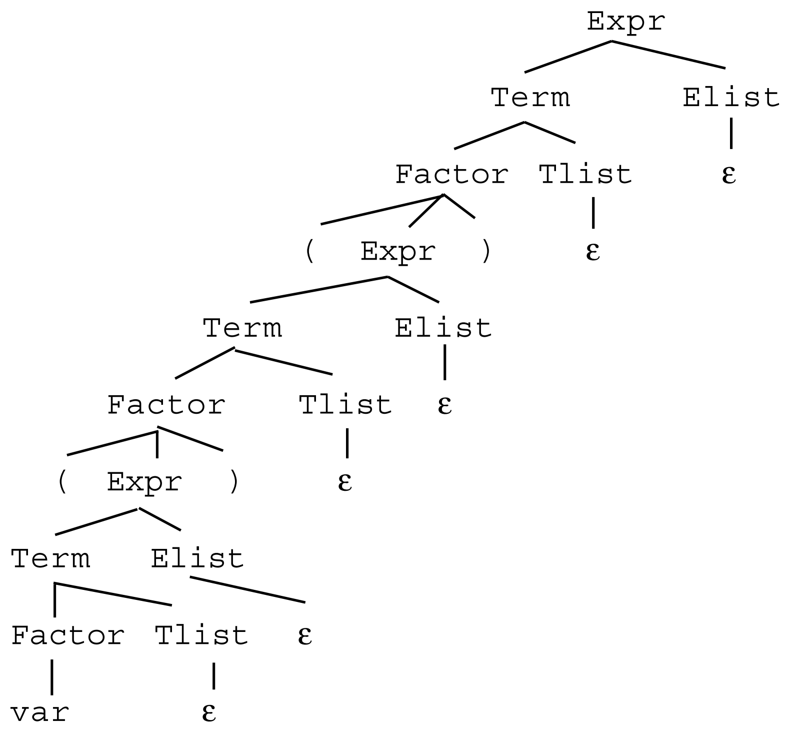
(b)

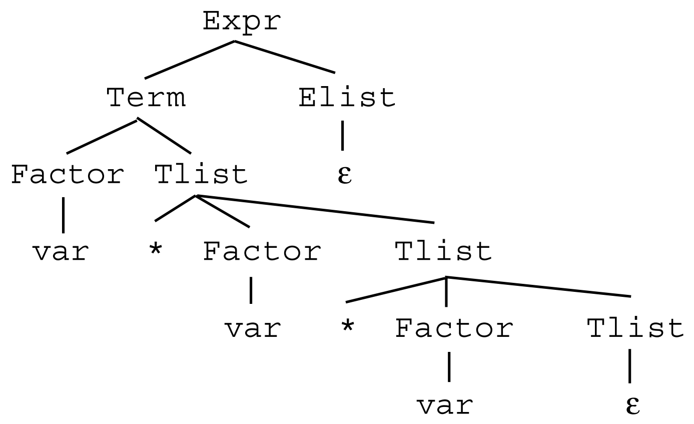


(c)


Expr
Term
Elist
Factor
Tlist
ε
(
Expr
)
*
Factor
Tlist
Term
Elist
var
ε
Factor
Tlist
+
Term
Elist
var
ε
Factor
Tlist
ε
var
ε

(d)


Expr
Term
Elist
Factor
Tlist
ε
(
Expr
)
ε
Term
Elist
Factor
Tlist
ε
(
Expr
)
ε
Term
Elist
Factor
Tlist
ε
var
ε



2. We have shown that grammar G16 for simple arithmetic expressions is LL(1), but grammar G5 is not LL(1). What are the advantages, if any, of grammar G5 over grammar G16?

Grammar G5 is simpler because it has fewer rules. Also, derivation trees formed with G5 show the correct structure of an arithmetic expression, i.e. subtrees correspond to subexpressions.

3. Suppose we permitted our parser to "peek ahead" `n` characters in the input stream to determine which rule to apply. Would we then be able to use grammar G5 to parse simple arithmetic expressions top down? In other words, is grammar G5 *LL(n)*?

No, grammar G5 is not LL(n) for any value of n. Note that any expression may begin with an arbitrary number of left parentheses. If the parser peeks ahead n symbols, there could be n+1 left parentheses, and it would be impossible to decide whether to apply rule 1 or rule 2.

4. Find two *null statements* in the recursive descent parser of the sample problem in this section. Which methods are they in and which grammar rules do they represent?

In the Elist method:

```
else if (inp==')' || inp=='↩')     ;
                                   // rule 3
```

In the Tlist method:

```
else if (inp=='+' || inp==')' || inp=='↩')     ;
                                   // rule 6
```

5. Construct part of a *recursive descent parser* for the following portion of a programming language:

```
1. Stmt  →   if (Expr) Stmt
2. Stmt  →   while (Expr) Stmt
3. Stmt  →   { StmtList }
4. Stmt  →   Expr ;
```

Write the procedure for the nonterminal `Stmt`. Assume the selection set for rule 4 is `{(, identifier, number}`.

```
void Stmt()
{  if (inp=='if')                  // rule 1
     {  inp = getInp();
        if (inp=='(')
          inp = getInp();
        else
          reject();
        Expr();
        if (inp==')')
          inp = getInp();
        else
          reject();
        Stmt();
     }
   else
     if  (inp=='while')            // rule 2
       {  inp = getInp();
          if  (inp=='(')
             inp = getInp();
```

```
              else
          reject();
         Expr();
         if (inp==')')
          inp = getInp();
         else
          reject();
         Stmt();
        }
      else
        if (inp=='{')                         // rule 3
          {  inp = getInp();
             StmtList();
             if (inp=='}')
              inp = getInp();
             else
              reject();
          }
      else
        if (inp=='(' || inp=='identifier ||
          inp=='number')                      // rule 4
          {  Expr();
             if (inp==';')
              inp = getInp();
             else
              reject();
          }
      else
        reject();
```

6. Show an *LL(1) grammar* for the language of regular expressions over the alphabet {`0,1`}, and show a *recursive descent parser* corresponding to the grammar.

```
1.   E → T Elist
2.   Elist → + T Elist
3.   Elist → ε
4.   T → F Tlist
5.   Tlist → . F Tlist
6.   Tlist → ε
7.   F → ( E ) R
8.   F → 0 R
9.   F → 1 R
10.  R → * R
11.  R → ε

void E()
{ if (inp=='(' || inp=='0' || inp=='1')     // rule 1
   { T();
     Elist();
   }
}

void Elist()
{ if (inp=='+')                             // rule 2
  {  inp = getInp();
     T();
     Elist();
  }
else if (inp==')' || inp=='↩')              // rule 3
     ;
else reject();
}

void T()
{   if (inp=='(' || inp=='0' || inp=='1')   // rule 4
     {   F();
         Tlist();
     }
    else reject();
}
void Tlist()
{   if (inp=='.')                          // rule 5
      {   inp = getInp();
          F();
          Tlist();
      }
    else if (inp=='+' || inp==')' || inp=='↩')
           ;                               // rule 6
    else reject();
}

void F()
{   if (inp=='(')                          // rule 7
      {   inp = getInp();
          E();
          if (inp==')')
            inp = getInp();
          else reject();
      }
      else if (inp=='0')                   // rule 8
      {   inp = getInp();
          R();
      }
    else if (inp=='1')                     // rule 9
      {   inp = getInp();
          R();
      }
    else reject();
}
```

```
void R()
{   if (inp=='*')                          // rule 10
     {   inp = getInp();
         R();
     }
    else if (inp=='+' || inp=='.' || inp=='↩'
          || inp==')')   ;                 // rule 11
     else reject();
}
```

7. Show how to *eliminate* the *left recursion* from each of the grammars shown below:

(a)
```
1.   A  →  A b c
2.   A  →  a b

1.   A → a b R
2.   R → b c R
3.   R → ε
```

(b)
```
1.   ParmList  →  ParmList , Parm
2.   ParmList  →  Parm

1.   ParmList → Parm R
2.   R → , Parm R
3.   R → ε
```

8. A parameter list is a list of 0 or more parameters separated by commas; a parameter list neither begins nor ends with a comma. Show an LL(1) *grammar* for a parameter list. Assume that parameter has already been defined.

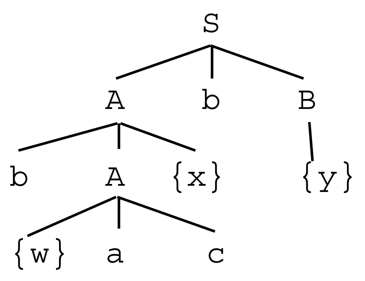
```
1.   ParmList → Parm R
2.   ParmList → ε
3.   R → , Parm R
4.   R → ε
```

Exercises 4.5

1. Consider the following translation grammar with starting nonterminal `S`, in which action symbols are put out:

```
1.   S  →  A b B
2.   A  →  {w} a c
3.   A  →  b A {x}
4.   B  →  {y}
```

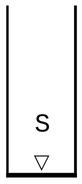
Show a *derivation tree* and the *output string* for the input `bacb`.



Output : `w x y`

2. Show an *extended pushdown translator* for the translation grammar of Problem 1.

S1	a	b	c	↩
S	Rep (BbA) Retain	Rep (BbA) Retain	Reject	Reject
A	Rep (ca{w}) Retain	Rep ({x}Ab) Retain	Reject	Reject
B	Reject	Reject	Reject	Rep({x}) Retain
a	pop adv	Reject	Reject	Reject
b	Reject	pop adv	Reject	Reject
c	Reject	Reject	pop adv	Reject
{w}	pop Retain Out(w)	pop Retain Out(w)	pop Retain Out(w)	pop Retain Out(w)
{x}	pop Retain Out(x)	pop Retain Out(x)	pop Retain Out(x)	pop Retain Out(x)
{y}	pop Retain Out(y)	pop Retain Out(y)	pop Retain Out(y)	pop Retain Out(y)
▽	Reject	Reject	Reject	Accept



Initial Stack

3. Show a *recursive descent translator* for the translation grammar of Problem 1.

```
void S()
{   if (inp=='a' || inp=='b')            // rule 1
      A();
    if (inp=='b')
      getInp();
    else reject();
```

```
      B();
    }

void A()
{   if (inp=='a')                        // rule 2
      {   getInp();
           System.out.println ('w');
          if (inp=='c')
            getInp();
          else reject();
      }
    else if (inp=='b')                   // rule 3
      {   getInp();
          A();
           System.out.println ('x');
      }
    else reject();
}

void B()
{   if (inp=='↩')                        // rule 4
      System.out.println  ('y');
    else reject();
}
```

4. Write the *Java statement* which would appear in a recursive descent parser for each of the following translation grammar rules:

(a) `A → {w} a {x} B C`

```
if  (inp=='a')
  {  System.out.println('w');
     getInp();
     System.out.println  ('x');
     B();
     C();
    }
```

(b) $A \rightarrow a\ \{w\}\ \{x\}\ B\ C$

```
if  (inp=='a')
  {  System.out.println('w');
     getInp();
     System.out.println  ('x');
     B();
     C();
   }
```

(c) $A \rightarrow a\ \{w\}\ B\ \{x\}\ C$

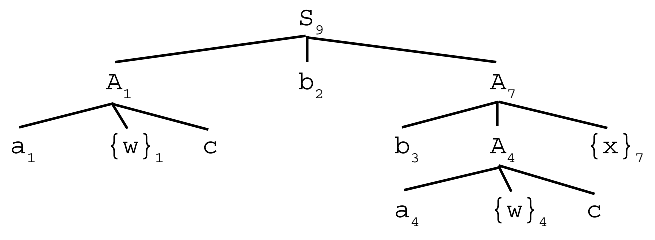
```
if  (inp=='a')
  {  System.out.println('w');
     getInp();
     B();
     System.out.println  ('x');
     C();
   }
```

Exercises 4.6

1. Consider the following attributed translation grammar with starting nonterminal S, in which action symbols are output:

1.	S_p	$\rightarrow$	$A_q\ b_r\ A_t$	$p \leftarrow r+t$
2.	A_p	$\rightarrow$	$a_p\ \{w\}_p\ c$	
3.	A_p	$\rightarrow$	$b_q\ A_r\ \{x\}_p$	$p \leftarrow q+r$

Show an *attributed derivation tree* for the input string $a_1cb_2b_3a_4c$, and show the output symbols with attributes corresponding to this input.



```
Output:     w1 w4 x7
```

2. Show a recursive descent translator for the grammar of Problem 1. Assume that all attributes are integers and that, as in sample problem 4.6, the Token class has methods `get_class()` and `get_value()` which return the class and value parts of a lexical token, and the Token class has a getToken() method which reads a token from the standard input file.

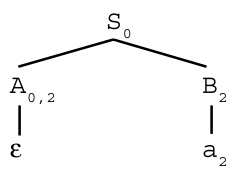
```
public static final int A=0, B=1, C=2;
void S(MutableInt p)
{   MutableInt q,r,t;
    if (token.getClass()==A ||
     token.getClass()==B)              // rule 1
     {   A(q);
         if (token.getClass()==B)
          {   r = token.getValue();
              token.getToken();
          }
         else reject();
         A(t);
         p.setValue (r.getValue() + t.getValue());
     }
    else reject();
}
```

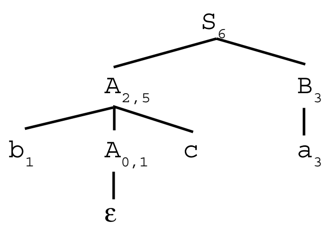
```
void A(MutableInt p)
{   MutableInt q,r;
    if (token.getClass()==A)  // rule 2
     {   p = token.getValue();
         token.getToken();
         System.out.println ("w" + p);
         if (token.getValue()==C)
          token.getToken();
         else reject();
     }
    else if (token.getClass()==B)  // rule 3
     {   q = token.getValue();
         token.getToken();
         A(r);
         p.setValue(q.getValue() + r.getValue());
         System.out.println ("x" + p);
     }
    else reject();
}
```

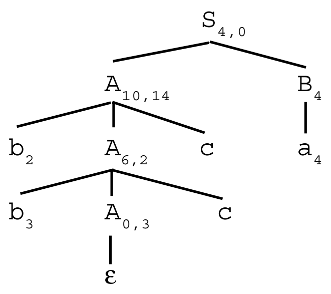
3. Show an *attributed derivation tree* for each input string using the following attributed grammar:

1.	$S_p \rightarrow A_{q,r}\ B_t$	$p \leftarrow q * t$
		$r \leftarrow q + t$
2.	$A_{p,q} \rightarrow b_r\ A_{t,u}\ c$	$u \leftarrow r$
		$p \leftarrow r + t + u$
3.	$A_{p,q} \rightarrow \varepsilon$	$p \leftarrow 0$
4.	$B_p \rightarrow a_p$	

(a) a_2 (b) b_1ca_3 (c) $b_2b_3cca_4$







4. Is it possible to write a recursive descent parser for the attributed translation grammar of Problem 3?

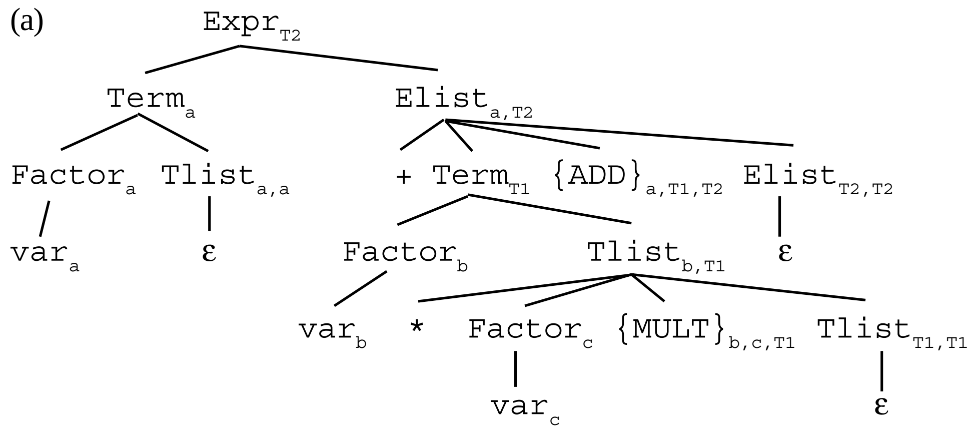
No.

Exercises 4.7

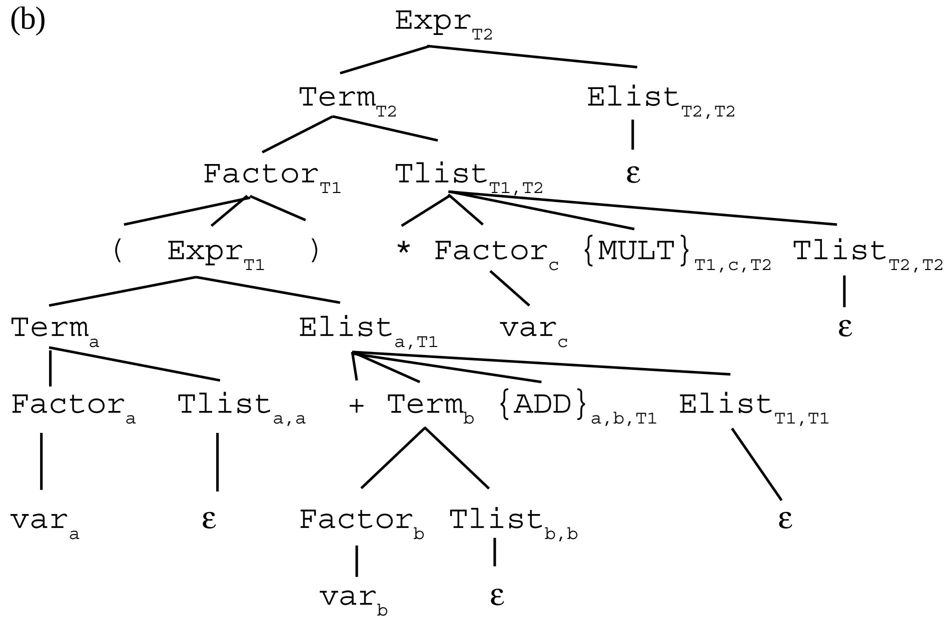
1. Show an *attributed derivation tree* for each of the following expressions, using grammar G21. Assume that the `Alloc` method returns a new temporary location each time it is called (`Temp1, Temp2, Temp3,` ...).

(a) `a + b * c` (b) `(a + b) * c`
(c) `(a)` (d) `a * b * c`

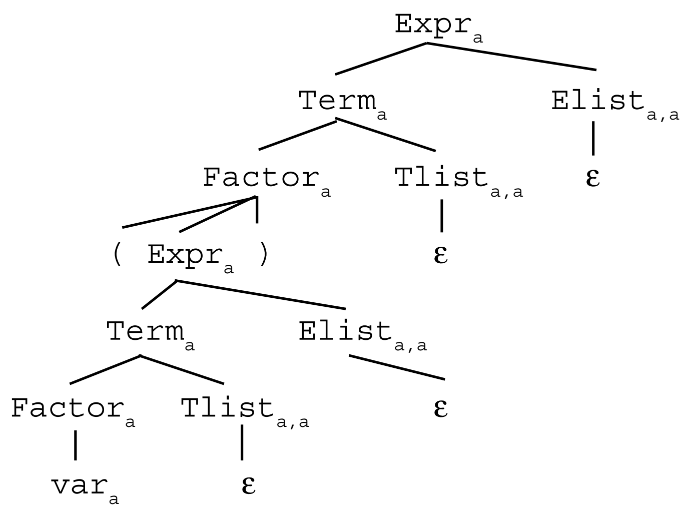
(a)



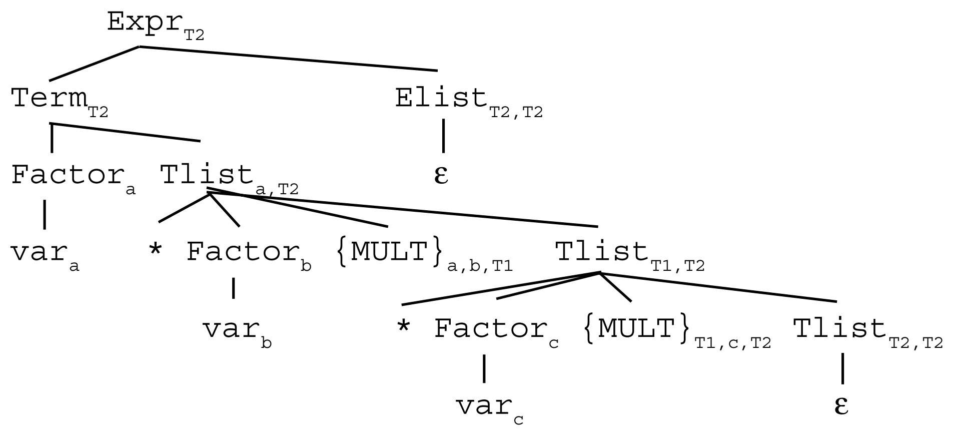
(b)



(c)


Expr_a
Term_a
Elist_a,a
Factor_a
Tlist_a,a
ε
(Expr_a)
ε
Term_a
Elist_a,a
Factor_a
Tlist_a,a
ε
var_a
ε

(d)


Expr_T2
Term_T2
Elist_T2,T2
Factor_a
Tlist_a,T2
ε
var_a
* Factor_b {MULT}_a,b,T1
Tlist_T1,T2
var_b
* Factor_c {MULT}_T1,c,T2
Tlist_T2,T2
var_c
ε

2. In the recursive descent translator of Section 4.7.1, refer to the method `Tlist`. In it, there is the following statement:

```
Atom (MULT, p, r, s)
```

Explain how the three variables `p,r,s` obtain values before being put out by the `atom` method.

The variable p is the first parameter to Tlist and is assigned a value when Tlist is called. The variable r is the actual parameter in the call to Factor. It is a reference to a Mutable Int which is assigned a value in the Factor method. The variable s is assigned a value when the alloc() method returns a value.

3. Improve grammar G21 to include the operations of subtraction and division, as well as unary plus and minus. Assume that there are `SUB` and `DIV` atoms to handle subtraction and division. Also assume that there is a `NEG` atom with two attributes to handle unary minus; the first is the expression being negated and the second is the result.

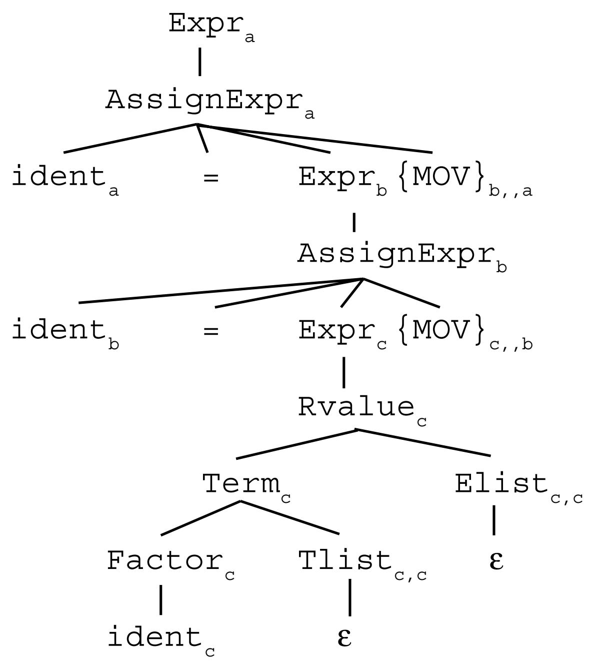
0.1	$\text{Expr}_p \rightarrow + \ \text{Expr}_p$	
0.2	$\text{Expr}_p \rightarrow - \ \text{Expr}_q \ \{\text{NEG}\}_{q,p}$	$q \leftarrow \text{Alloc()}$
1.	$\text{Expr}_p \rightarrow \text{Term}_q \ \text{Elist}_{q,p}$	
2.	$\text{Elist}_{p,q} \rightarrow + \ \text{Term}_r \ \{\text{ADD}\}_{p,r,s} \ \text{Elist}_{s,q}$	$s \leftarrow \text{Alloc()}$
2.1	$\text{Elist}_{p,q} \rightarrow - \ \text{Term}_r \ \{\text{SUB}\}_{p,r,s} \ \text{Elist}_{s,q}$	$s \leftarrow \text{Alloc()}$
3.	$\text{Elist}_{p,q} \rightarrow \varepsilon$	$q \leftarrow p$
4.	$\text{Term}_p \rightarrow \text{Factor}_q \ \text{Tlist}_{q,p}$	
5.	$\text{Tlist}_{p,q} \rightarrow * \ \text{Factor}_r \ \{\text{MULT}\}_{p,r,s} \ \text{Tlist}_{s,q}$	$s \leftarrow \text{Alloc()}$
5.1	$\text{Tlist}_{p,q} \rightarrow / \ \text{Factor}_r \ \{\text{DIV}\}_{p,r,s} \ \text{Tlist}_{s,q}$	$s \leftarrow \text{Alloc()}$
6.	$\text{Tlist}_{p,q} \rightarrow \varepsilon$	$q \leftarrow p$
7.	$\text{Factor}_p \rightarrow (\ \text{Expr}_p \ )$	
8.	$\text{Factor}_p \rightarrow \text{ident}_p$	

Exercises 4.8

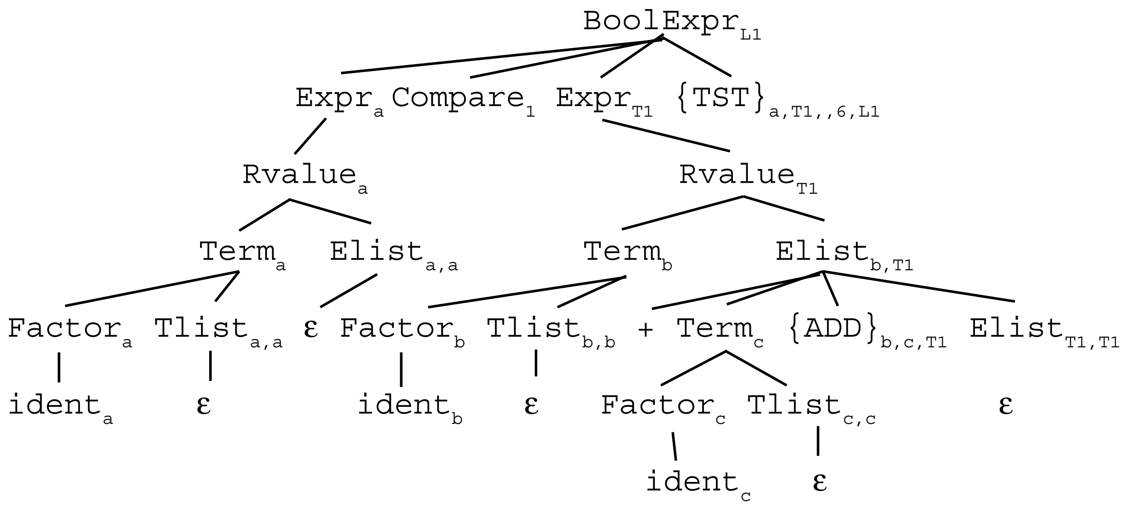
1. Show an *attributed derivation tree* using the grammar for Decaf expressions given in this section for each of the following expressions or boolean expressions (in part (a) start with Expr; in parts (b,c,d,e) start with BoolExpr):

(a)	`a = b = c`	(b)	`a == b + c`
(c)	`(a=3) <= (b=2)`	(d)	`a == - (c = 3)`
(e)	`a * (b=3) + c != 9`		

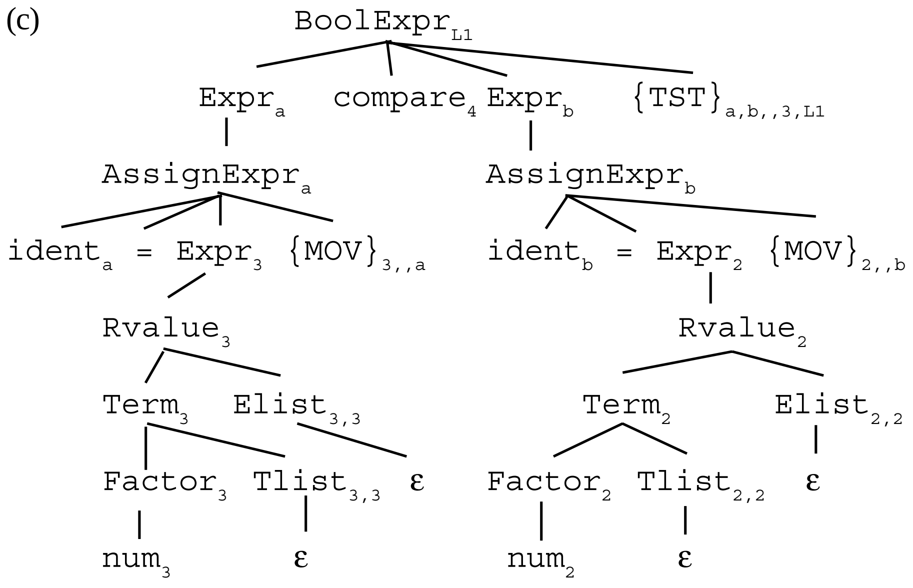
(a)



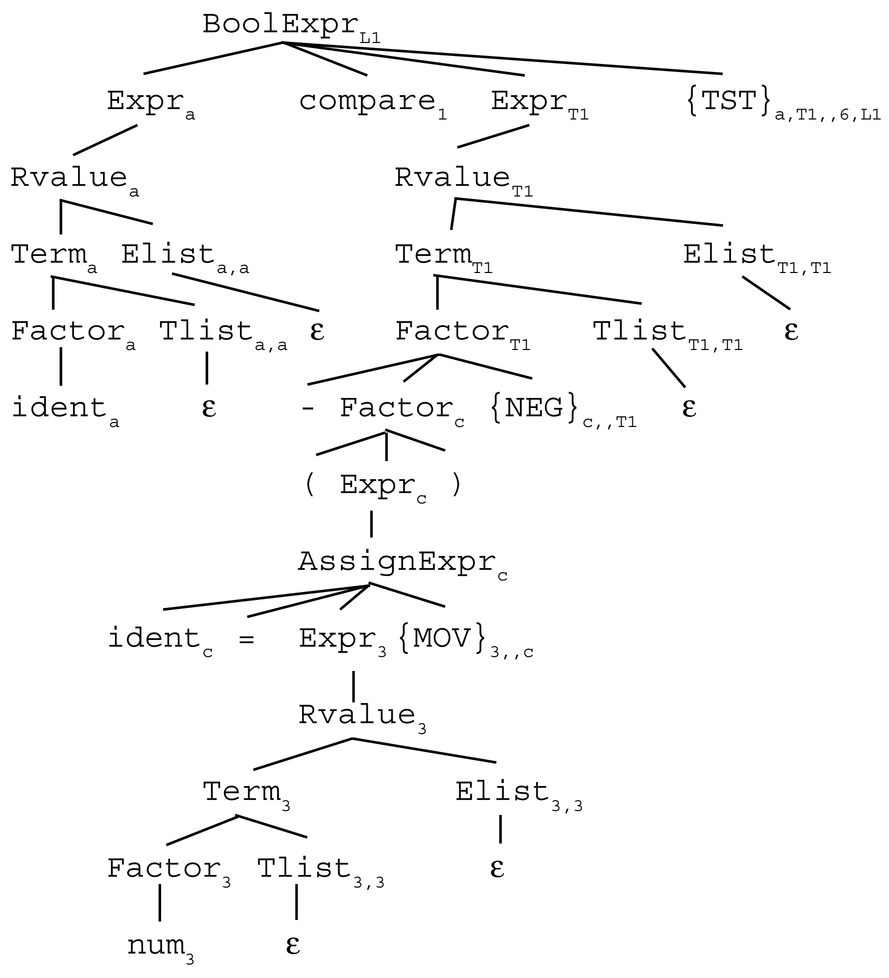
(b)


BoolExpr_L1
Expr_a Compare_1 Expr_T1 {TST}_a,T1,,6,L1
Rvalue_a
Rvalue_T1
Term_a Elist_a,a
Term_b Elist_b,T1
Factor_a Tlist_a,a ε Factor_b Tlist_b,b + Term_c {ADD}_b,c,T1 Elist_T1,T1
ident_a ε ident_b ε Factor_c Tlist_c,c ε
ident_c ε

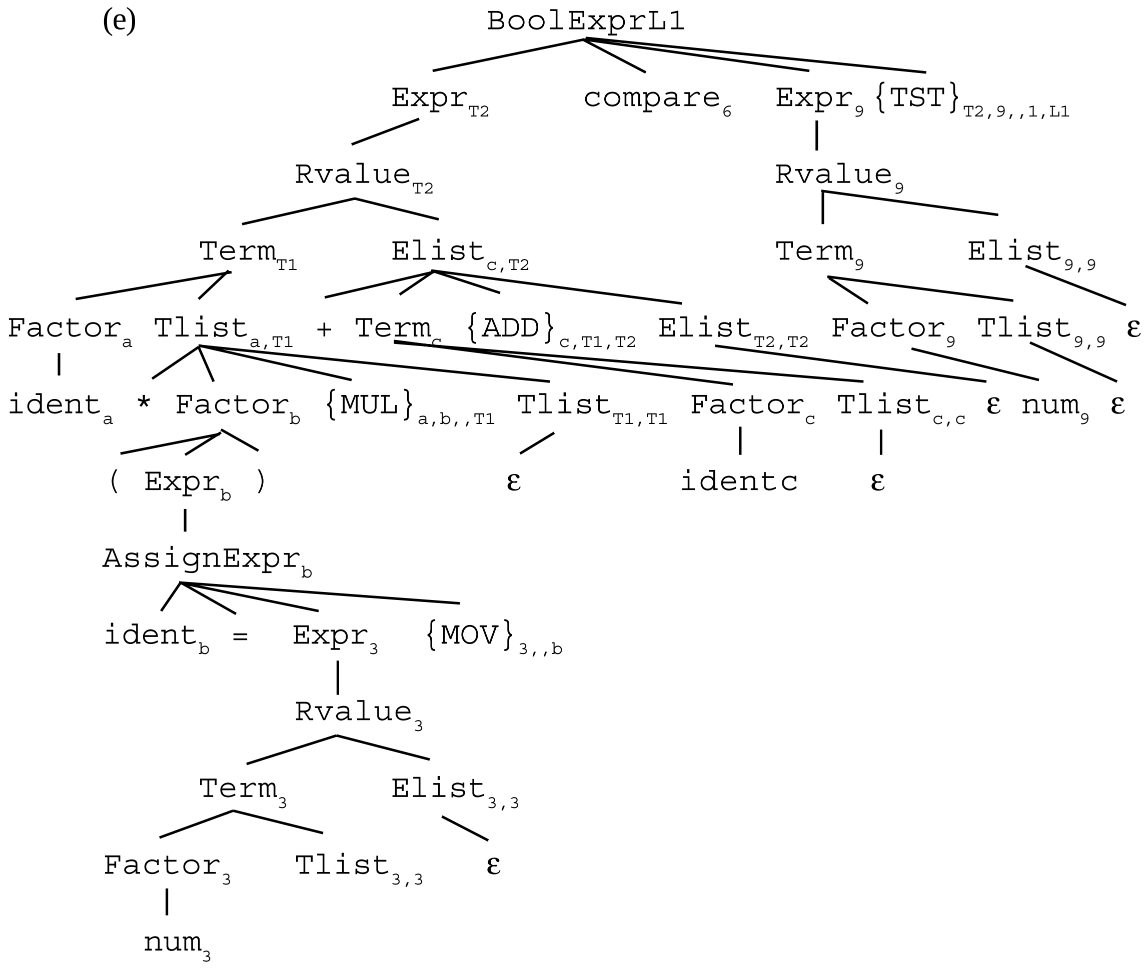
(c)


BoolExpr$_{L1}$
Expr$_a$
compare$_4$
Expr$_b$
{TST}$_{a,b,,3,L1}$
AssignExpr$_a$
AssignExpr$_b$
ident$_a$
=
Expr$_3$
{MOV}$_{3,,a}$
ident$_b$
=
Expr$_2$
{MOV}$_{2,,b}$
Rvalue$_3$
Rvalue$_2$
Term$_3$
Elist$_{3,3}$
Term$_2$
Elist$_{2,2}$
Factor$_3$
Tlist$_{3,3}$
ε
Factor$_2$
Tlist$_{2,2}$
ε
num$_3$
ε
num$_2$
ε

(d)


BoolExpr$_{L1}$
Expr$_a$
compare$_1$
Expr$_{T1}$
{TST}$_{a,T1,,6,L1}$
Rvalue$_a$
Rvalue$_{T1}$
Term$_a$
Elist$_{a,a}$
Term$_{T1}$
Elist$_{T1,T1}$
Factor$_a$
Tlist$_{a,a}$
ε
Factor$_{T1}$
Tlist$_{T1,T1}$
ε
ident$_a$
ε
-
Factor$_c$
{NEG}$_{c,,T1}$
ε
(
Expr$_c$
)
AssignExpr$_c$
ident$_c$
=
Expr$_3$
{MOV}$_{3,,c}$
Rvalue$_3$
Term$_3$
Elist$_{3,3}$
Factor$_3$
Tlist$_{3,3}$
ε
num$_3$
ε

(e)



2. Show the recursive descent parser for the nonterminals BoolExpr, Rvalue, and Elist given in the grammar for Decaf expressions. Hint: the selection sets for the first eight grammar rules are:

```
Sel (1)  =  {ident, num, (, +, -}
Sel (2)  =  {ident}
Sel (3)  =  {ident, num, (, +, -}
Sel (4)  =  {ident}
Sel (5)  =  {ident, num, (, +, -}
Sel (6)  =  {+}
Sel (7)  =  {-}
Sel (8)  =  {), ↩}
```

```
enum {IDENT, NUM, LPAREN, PLUS, MINUS, END, COMPARE}

void BoolExpr(MutableInt l1)
{  MutableInt p,c,q;
   if (token.getClass()==IDENT ||
            token.getClass()==NUM ||
            token.getClass()==LPAREN ||
            token.getClass()==PLUS ||
            token.getClass()==MINUS)            // rule 1
      {  Expr(p);
         if (token.getClass()==COMPARE)
            {  c.setValue(token.getValue());
               token.getToken();
            }
          else reject();
          Expr(q);
          System.out.println
            ("TST", p,q,0,7-c.getValue(),l1);
      }
}

void Rvalue (MutableInt p)
{  MutableInt q;
   if (token.getClass()==IDENT ||
            token.getClass()==NUM ||
            token.getClass()==LPAREN ||
            token.getClass()==PLUS ||
            token.getClass()==MINUS)            // rule 5
      {  Term(q);
         Elist(q,p);
      }
}

void Elist (MutableInt p, MutableInt q)
{  MutableInt r,s;
   if (token.getClass()==PLUS)                  // rule 6
      {  token.getToken();
          Term(r);
          s.setValue(alloc());
          System.out.println ("ADD", p,r,s);
           Elist (s,q);
      }
```

```
    else if token.getClass()==MINUS)             // rule 7
      {  token.getToken();
           Term(r);
           s.setValue(alloc());
           System.out.println ("SUB", p,r,s);
            Elist (s,q);
      }
    else if (token.getClass()==LPAREN ||
                token.getClass()==END)           // rule 8
        q.setValue(p.getValue());                // q = p
     else reject();
}
```

Exercises 4.9

1. Show the *sequence of atoms* which would be put out according to Figure 4.18 for each of the following input strings:

(a) `if (a==b) while (x<y) Stmt`

```
(TST,a,b,,6,l1)
(LBL, l2)
(TST,x,y,,5,l3)
     atoms for Stmt
(JMP,l2)
(LBL,l3)
(JMP,l4)
(LBL,l1)
(LBL,l4)
```

(b)
```
for (i = 1; i<=100; i = i+1)
     for (j = 1; j<=i; j = j+1) Stmt

(MOV,1,,i)
(LBL,l1)
(TST,i,100,,3,l2)
(JMP,l3)
(LBL,l4)
(ADD,i,1,T1)
(MOV,T1,,i)
(JMP,l1)
(LBL,l3)
(MOV,1,,j)
(LBL,l5)
(TST,j,i,,3,l6)
(JMP,l7)
(LBL,l8)
(ADD,j,1,T2)
(MOV,T2,,j)
(JMP,l5)
(LBL,l7)
   Atoms for Stmt
(JMP,l8)
(LBL,l6)
(JMP,l4)
(LBL,l2)
```

(c)
```
if (a==b) for (i=1; i<=20; i=i+1) Stmt1
     else while (i>0) Stmt2

(TST,a,b,,6,l1)
(MOV,1,,i)
(LBL,l2)
(TST,i,20,,3,l3)
(JMP,l4)
(Lbl,l5)
(ADD,i,1,T1)
(MOV,T1,,i)
(JMP,l2)
(LBL,l4)
     Atoms for Stmt1
(JMP,l5)
(LBL,l3)
(JMP,l6)
(LBL,l1)
(LBL,l7)
(TST,i,0,,4,l8)
     Atoms for Stmt2
(JMP,l7)
(LBL,l8)
(LBL,l6)
```

(d) `if (a==b) if (b>0) Stmt1 else while (i>0) Stmt2`

```
(TST,a,b,,6,l1)
(TST,b,0,,4,l2)
     Atoms for Stmt1
(JMP,l3)
(LBL,l2)
(LBL,l4)
(TST,i,0,,4,l5)
     Atoms for Stmt2
(JMP,l4)
(LBL,l5)
(LBL,l3)
(JMP,l6)
(LBL,l1)
(LBL,l6)
```

2. Show an *attributed translation grammar rule* for each of the control structures given in Figure 4.18. Assume `if` statements always have an `else` part and that there is a method, `newlab`, which allocates a new statement label.

1. WhileStmt → while $\{LBL\}_{Lbl1}$ (BoolExpr$_{Lbl2}$) Stmt $\{JMP\}_{Lbl1}$ $\{LBL\}_{Lbl2}$

   Lbl1 ← newlab()
   Lbl2 ← newlab()

2. ForStmt → for (AssignExpr$_r$; $\{LBL\}_{Lbl1}$ BoolExpr$_{Lbl4}$; $\{JMP\}_{Lbl2}$ $\{LBL\}_{Lbl3}$ AssignExpr$_r$) $\{JMP\}_{Lbl1}$ $\{LBL\}_{Lbl2}$ Stmt $\{JMP\}_{Lbl3}$ $\{LBL\}_{Lbl4}$

   Lbl1 ← newlab()
   Lbl2 ← newlab()
   Lbl3 ← newlab()
   Lbl4 ← newlab()

3. `IfStmt` $\rightarrow$ `if (BoolExpr`$_{Lbl1}$`) Stmt {JMP}`$_{Lbl2}$ `{LBL}`$_{Lbl1}$ `ElsePart {LBL}`$_{Lbl2}$

4. `ElsePart` $\rightarrow$ `else Stmt`

5. `ElsePart` $\rightarrow$ ε

3. Show a *recursive descent translator* for your solutions to Problem 2. Show methods for `WhileStmt, ForStmt`, and `IfStmt`.

```
void WhileStmt()
{  MutableInt Lbl1, Lbl2;
   if (token.getClass()==WHILE)
     {  token.getToken();
         Lbl1.setValue(newlab());
        System.out.println ("LBL", Lbl1);
         if (token.getClass()==LPAREN)
             token.getToken();
         else reject();
         BoolExpr(Lbl2);
         if  (token.getClass()==RPAREN)
            token.getToken();
         else reject();
         Stmt();
         System.out.println ("JMP", Lbl1);
         System.out.println ("LBL", Lbl2);
     }
    else reject();
}
```

```
void  ForStmt()
{   MutableInt Lbl1, Lbl2, Lbl3, Lbl4, r;
    if (token.getClass()==FOR)
     {     token.getToken();
           if  (token.getClass()==LPAREN)
              token.getToken();
           else  reject();
           AssignExpr(r);
           if  (token.getClass()==SEMI)
               token.getToken();
           else  reject();
           Lbl1.setValue(newlab());
           System.out.println  ("LBL",Lbl1);
           BoolExpr(Lbl4);
           if  (token.getClass()==SEMI)
               token.getToken();
           else  reject();
           Lbl2.setValue(newlab());
           System.out.println  ("JMP",Lbl2);
           Lbl3.setValue(newlab());
           System.out.println  ("LBL",Lbl3);
           AssignExpr  (r);
           if  (token.getClass()==RPAREN)
                token.getToken();
           else  reject();
           System.out.println  ("JMP",Lbl1);
           System.out.println  ("LBL",Lbl2);
           Stmt();
           System.out.println  ("JMP",Lbl3);
           System.out.println  ("LBL",Lbl4);
```

```
        }
       else reject();
}

void  IfStmt()
{     MutableInt  Lbl1,  Lbl2;
      if  (token.getClass()==IF)
        {  token.getToken();
           if  (token.getClass()==LPAREN)
                token.getToken();
           else  reject();
           BoolExpr(Lbl1);
           Stmt();
           Lbl2.setValue(newlab());
           System.out.println  ("JMP",Lbl2);
           Lbl1.setValue(newlab());
           System.out.println  ("LBL",Lbl1);
           ElsePart();
           System.out.println  ("LBL",Lbl2);
        }
      else  reject();
}

void  ElsePart()
{     if  (token.getClass()==ELSE)
        {  token.getToken();
           Stmt();
        }
}
```

4. Does your Java compiler permit a loop control variable to be altered inside the loop, as in the following example?

```
for (int i=0; i<100; i = i+1)
     {   System.out.println (i);
         i = 100;
     }
```

```
Yes.
```

Exercises 4.10

1. Show the atoms put out as a result of the following Decaf statement:

```
if (a==3) { a = 4;
             for (i = 2; i<5; i=0 ) i = i + 1;
           }
else while (a>5) i = i * 8;
```

```
(TST,a,3,,6,l1)
(MOV,4,,a)
(MOV,2,,i)
(LBL,l2)
(TST,i,5,,5,l3)
(JMP,l4)
(LBL,l5)
(MOV,0,,i)
(JMP,l2)
(LBL,l4)
(ADD,i,1,T1)
(MOV,T1,,i)
(JMP,l5)
(LBL,l3)
(JMP,l6)
(LBL,l1)
(LBL,l7)
(TST,a,5,,4,l8)
```

```
(MUL,i,8,T2)
(MOV,T2,,i)
(JMP,l7)
(LBL,l8)
(LBL,l6)
```

2. Explain the purpose of each atom put out in our Decaf attributed translation grammar for the `for` statement:

```
ForStmt     →     for ( OptExpr_p; {LBL}_Lbl1 OptBoolExpr_Lbl3;
                  {JMP}_Lbl2 {LBL}_Lbl4 OptExpr_r ) {JMP}_Lbl1
                  {LBL}_Lbl2 Stmt {JMP}_Lbl4 {LBL}_Lbl3
                              Lbl1←newlab() Lbl2←newlab()
                              Lbl3←newlab() Lbl4←newlab()
```

```
{LBL}_Lbl1    Target of jump to test for loop continuation
after the second OptExpr is evaluated.

{JMP}_Lbl2    If the loop is to continue, jump to the body of
the loop.

{LBL}_Lbl4    Target of the jump after the body of the loop is
executed, to evaluate the second OptExpr.

{JMP}_Lbl1    Jump to test for loop continuation, after the
second OptExpr is evaulated.

{LBL}_Lbl2    Target of jump to loop body after contuation
test succeeds.

{JMP}_Lbl4    Jump to evaluation of second OptExpr after body
of loop is executed.
```

3. The Java language has a `switch` statement.

(a) Include a definition of the `switch` statement in the *attributed translation grammar* for Decaf.

$\texttt{SwitchStmt} \rightarrow \texttt{switch (Expr}_{p}\texttt{) \{ CaseList}_{p,Lbl1,Lbl2}$
$\qquad \{\texttt{LBL}\}_{Lbl2}\ \texttt{\}}$
$\qquad \texttt{Lbl1} \leftarrow \texttt{newlab()}$
$\qquad \texttt{Lbl2} \leftarrow \texttt{newlab()}$

$\texttt{CaseList}_{p,Lbl1,Lbl2} \rightarrow \texttt{case num}_{q}\ \{\texttt{TST}\}_{p,q,,6,Lbl3}$
$\qquad \texttt{:}\ \{\texttt{Lbl}\}_{Lbl1}\ \texttt{StmtList Break}_{Lbl2,Lbl4}\ \{\texttt{LBL}\}_{Lbl3}$
$\qquad \texttt{CaseList}_{p,Lbl4,Lbl2}$
$\qquad \texttt{Lbl3} \leftarrow \texttt{newlab()}$
$\qquad \texttt{Lbl4} \leftarrow \texttt{newlab()}$

$\texttt{Break}_{Lbl1,Lbl2} \rightarrow \texttt{break ;}\ \{\texttt{JMP}\}_{Lbl1}$

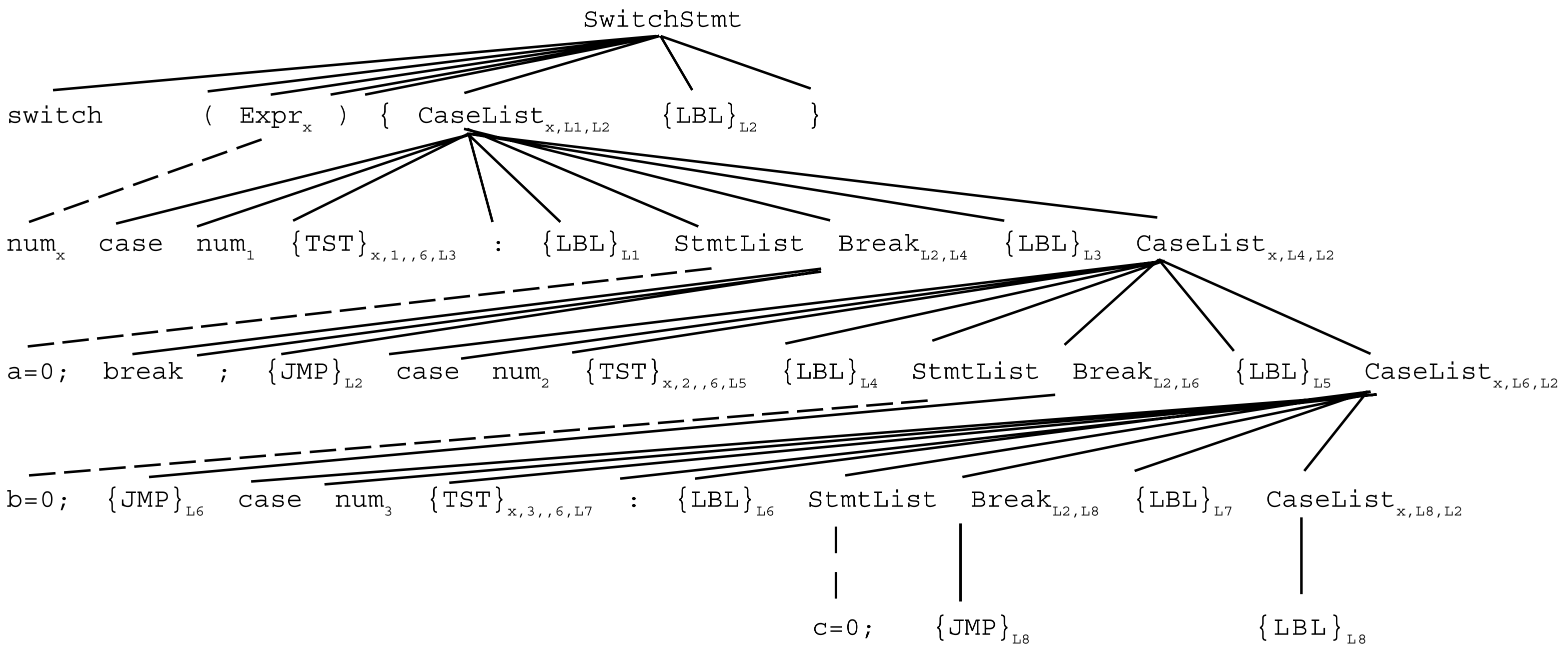
$\texttt{Break}_{Lbl1,Lbl2} \rightarrow \{\texttt{JMP}\}_{Lbl2}$

$\texttt{CaseList}_{p,Lbl1,Lbl2} \rightarrow \{\texttt{LBL}\}_{Lbl1}$

Not implemented: *default*

(b) Check your grammar by building an *attributed derivation tree* for a sample `switch` statement of your own design.

```
switch (x)
  {   case 1:      a = 0;
                   break;
      case 2:      b = 0;
      case 3:      c = 0;
  }
```



(c) Include code for the `switch` statement in the *recursive descent parser*, decaf.java and parse.java .

```
void  SwitchStmt()
{    MutableInt  p,Lbl1,Lbl2;
     if  (token.getClass()==SWITCH)
       {  token.getToken();
          if  (token.getClass()==LPAREN)
             token.getToken();
          else  reject();
          Expr  (p);
          if  (token.getToken()==RPAREN)
             token.getToken();
          else  reject();
          if  (token.getToken()==LBRACE)
             token.getToken();
          else  reject();
          Lbl1.setValue  (newlab());
          Lbl2.setValue  (newlab());
          CaseList  (p,  Lbl1,  Lbl2);
          atom  ("LBL",  Lbl2);
          if  (token.getClass()==RBRACE)
             token.getToken();
          else  reject();
       }
     else  reject();
}

void  CaseList  (MutableInt  p,  MutableInt  Lbl1,
                 MutableInt  Lbl2)
{    MutableInt  q,  Lbl3,  Lbl4;
     if  (token.getClass()==CASE)
     {
          token.getToken();
     if  (token.getClass()==NUM)
          q.setValue(token.getValue());
     else  reject();
     Lbl3.setValue(newlab());
     atom  ("TST",p,q,,6,Lbl3);
     if  (token.getClass()==COLON)
          token.getToken();
     else  reject();
     atom  ("LBL",Lbl1);
     StmtList();
     Lbl4.setValue(newlab());
     Break  (Lbl2,Lbl4);
     atom  ("LBL",Lbl3);
     CaseList  (p,Lbl4,Lbl2);
     }
     else  atom("LBL",Lbl1);
     }

void Break (MutableInt Lbl1, MutableInt Lbl2)
{    if  (token.getClass()==BREAK)
       {  token.getToken();
          if  (token.getClass()==SEMI)
             token.getToken();
          else  reject();
          atom  ("JMP",Lbl1);
       }
     else
          atom  ("JMP",Lbl2);
}
```

4. Using the grammar of Figure 4.19, show an attributed derivation tree for the statement given in problem 1, above.

Stmt

IfStmt

if (BoolExpr_{l1}) Stmt $\{\text{JMP}\}_{l2}$ $\{\text{LBL}\}_{l1}$ ElsePart $\{\text{LBL}\}_{l2}$

Expr_a compare1 Expr_3 $\{\text{TST}\}_{a,3,,6,l1}$ CompoundStmt else Stmt

ident_a num_3 { StmtList } WhileStmt

StmtList Stmt while (BoolExpr) Stmt

StmtList Stmt ForStmt Expr compare3 Expr AssignStmt

AssignStmt for (AssignExpr_i; $\{\text{LBL}\}_{l1}$ BoolExpr_{l1}; $\{\text{JMP}\}_{l1}$ $\{\text{LBL}\}_{l1}$ AssignExpr_i) $\{\text{JMP}\}_{l1}$ $\{\text{LBL}\}_{l1}$ Stmt $\{\text{JMP}\}_{l1}$ $\{\text{LBL}\}_{l1}$ AssignExpr

Under Construction

5. Implement a `do-while` statement in decaf, following the guidelines in problem 3.

$$\begin{array}{l} \texttt{DoWhileStmt} \rightarrow \texttt{do} \ \{\texttt{LBL}\}_{\texttt{Lbl1}} \ \texttt{Stmt while (BoolExpr}_{\texttt{Lbl2}} \texttt{)} \\ \qquad \{\texttt{JMP}\}_{\texttt{Lbl1}} \ \{\texttt{LBL}\}_{\texttt{Lbl2}} \ \texttt{;} \\ \qquad\qquad\qquad\qquad \texttt{Lbl1} \leftarrow \texttt{newlab();} \\ \qquad\qquad\qquad\qquad \texttt{Lbl2} \leftarrow \texttt{newlab();} \end{array}$$

```
void DoWhileStmt ()
{    MutableInt Lbl1, Lbl2;
     if  (token.getClass()==DO)
          token.getToken();
     else  reject();
     Lbl1.setValue(newlab());
     atom  ("LBL",Lbl1);
     Stmt();
     if  (token.getClass()==WHILE)
          token.getToken();
     else  reject();
     if  (token.getClass()==LPAREN)
          token.getToken();
     else  reject();
     Lbl2.setValue(newlab());
     BoolExpr(Lbl2);
     if  (token.getClass()==RPAREN)
          token.getToken();
     else  reject();
     atom  ("JMP",Lbl1);
     atom  ("LBL",Lbl2);
}
```

Chapter 5

Exercises 5.1

1. For each of the following stack configurations, identify the *handle* using the grammar shown below:

```
1.   S → S A b
2.   S → a c b
3.   A → b B c
4.   A → b c
5.   B → b a
6.   B → A c
```

(a)	▽SSAb	(b)	▽SSbbc
(c)	▽SbBc	(d)	▽Sbbc

(a)	`S A b`	(b)	`b c`
(c)	`b B c`	(d)	`b c`

2. Using the grammar of Problem 1, show the sequence of *stack and input configurations* as each of the following strings is parsed with shift reduce parsing:

(a) `acb` (b) `acbbcb`
(c) `acbbbacb` (d) `acbbbcccb`
(e) `acbbcbbcb`

(a)
```
|▽                       acb ↩
                              shift
|▽a                       cb ↩
                              shift
|▽ac                       b ↩
                              shift
|▽acb                        ↩
                              reduce using rule 2
|▽S                          ↩
                              Accept
```

(b)
```
|▽                    acbbcb ↩
                              shift
|▽a                    cbbcb ↩
                              shift
|▽ac                    bbcb ↩
                              shift
|▽acb                    bcb ↩
                              reduce using rule 2
|▽S                      bcb ↩
                              shift
|▽Sb                      cb ↩
                              shift
|▽Sbc                      b ↩
                              reduce using rule 4
|▽SA                       b ↩
                              shift
|▽SAb                        ↩
                              reduce using rule 1
|▽S                          ↩
                              Accept
```

(c)

```
▽            acbbbacb ↩
                        shift
▽ a           cbbbacb N
                        shift
▽ ac           bbbacb ↩
                        shift
▽ acb           bbacb ↩
                        reduce using rule 2
▽ S             bbacb ↩
                        shift
▽ Sb             bacb ↩
                        shift
▽ Sbb             acb ↩
                        shift
▽ Sbba             cb ↩
                        reduce using rule 5
▽ SbB              cb ↩
                        shift
▽ SbBc              b ↩
                        reduce using rule 3
▽ SA                b ↩
                        shift
▽ SAb                 ↩
                        reduce using rule 1
▽ S                   ↩
                        Accept
```

(d)

```
▽          acbbbcccb ↩
                        shift
▽a          cbbbcccb ↩
                        shift
▽ac          bbbcccb ↩
                        shift
▽acb          bbcccb ↩
                        reduce using rule 2
▽S            bbcccb ↩
                        shift
▽Sb            bcccb ↩
                        shift
▽Sbb            cccb ↩
                        shift
▽Sbbc            ccb ↩
                        reduce using rule 4
▽SbA             ccb ↩
                        shift
▽SbAc             cb ↩
                        reduce using rule 6
▽SbB              cb ↩
                        shift
▽SbBc              b ↩
                        reduce using rule 3
▽SA                b ↩
                        shift
▽SAb                 ↩
                        reduce using rule 1
▽S                   ↩
                        Accept
```

(e)

```
▽            acbbcbbcb ↩
                         shift
▽ a           cbbcbbcb ↩
                         shift
▽ ac           bbcbbcb ↩
                         shift
▽ acb           bcbbcb ↩
                         reduce using rule 2
▽ S             bcbbcb ↩
                         shift
▽ Sb             cbbcb ↩
                         shift
▽ Sbc             bbcb ↩
                         reduce using rule 4
▽ SA              bbcb ↩
                         shift
▽ SAb              bcb ↩
                         reduce using rule 1
▽ S                bcb ↩
                         shift
▽ Sb                cb ↩
                         shift
▽ Sbc                b ↩
                         reduce using rule 4
▽ SA                 b ↩
                         shift
▽ SAb                  ↩
                         reduce using rule 1
▽ S                    ↩
                         Accept
```

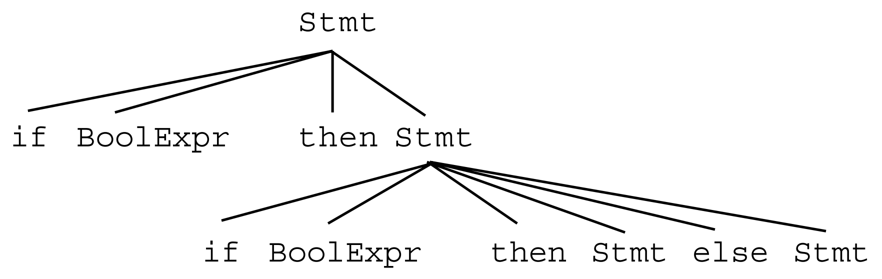
3. For each of the following input strings, indicate whether a shift/reduce parser will encounter a *shift/reduce conflict*, a *reduce/reduce conflict*, or *no conflict* when parsing, using the grammar below:

```
1.   S → S a b
2.   S → b A
3.   A → b b
4.   A → b A
5.   A → b b c
6.   A → c
```

(a)	`b c`	Reduce/Reduce (rules 2 and 4)
(b)	`b b c a b`	Shift/Reduce (rules 3 and 5)
(c)	`b a c b`	No conflict

4. Assume that a shift/reduce parser always chooses the lower numbered rule (i.e., the one listed first in the grammar) whenever a reduce/reduce conflict occurs during parsing, and it chooses a shift whenever a shift/reduce conflict occurs. Show a *derivation tree* corresponding to the parse for the sentential form `if (BoolExpr) if (BoolExpr) Stmt else Stmt` , using the following ambiguous grammar. Since the grammar is not complete, you may have nonterminal symbols at the leaves of the derivation tree.

```
1.   Stmt → if (BoolExpr) Stmt else Stmt
2.   Stmt → if (Expr) Stmt
```



Exercises 5.2

1. Show the sequence of *stack and input configurations* and the *reduce and goto operations* for each of the following expressions, using the action and goto tables of Figure 5.7.

(a) `var`
(b) `(var)`
(c) `var + var * var`
(d) `(var*var) + var`
(e) `(var * var`

(a)

Stack	Input	Action	Goto
▽	`var` ↩	shift var	
▽ `var`	↩	reduce 6	push Factor4
▽ `Factor4`	↩	reduce 4	push Term2
▽ `Term2`	↩	reduce 2	push Expr1
▽ `Expr1`	↩	Accept	

(b)

Stack	Input	Action	Goto
▽	`(var)` ↩	shift (	
▽ `(`	`var)` ↩	shift var	
▽ `(var`	`)` ↩	reduce 6	push Factor4
▽ `(Factor4`	`)` ↩	reduce 4	push Term2
▽ `(Term2`	`)` ↩	reduce 2	push Expr5
▽ `(Expr5`	`)` ↩	shift)	
▽ `(Expr5)`	↩	reduce 5	push Factor4
▽ `Factor4`	↩	reduce 4	push Term2
▽ `Term2`	↩	reduce 2	push Expr1
▽ `Expr1`	↩	Accept	

(c)

Stack	Input	Action	Goto
`▽`	`var+var*var ↩`	shift var	
`▽var`	`+var*var ↩`	reduce 6	push Factor4
`▽Factor4`	`+var*var ↩`	reduce 4	push Term2
`▽Term2`	`+var*var ↩`	reduce 2	push Expr1
`▽Expr1`	`+var*var ↩`	shift +	
`▽Expr1+`	`var*var ↩`	shift var	
`▽Expr1+var`	`*var ↩`	reduce 6	push Factor4
`▽Expr1+Factor4`	`*var ↩`	reduce 4	push Term1
`▽Expr1+Term1`	`*var ↩`	shift *	push Expr1
`▽Expr1+Term1*`	`var ↩`	shift var	
`▽Expr1+Term1*var`	`↩`	reduce 6	push Factor3
`▽Expr1+Term1*Factor3`	`↩`	reduce 3	push Term1
`▽Expr1+Term1`	`↩`	reduce 1	push Expr1
`▽Expr1`	`↩`	Accept	

(d)

Stack	Input	Action	Goto
`▽`	`(var*var)+var ↩`		
		shift (	
`▽(`	`var*var)+var ↩`		
		shift var	
`▽(var`	`*var)+var ↩`		
		reduce 6	push Factor4
`▽(Factor4`	`*var)+var ↩`		
		reduce 4	push Term2
`▽(Term2`	`*var)+var ↩`		
		shift *	
`▽(Term2*`	`var)+var ↩`		
		shift var	
`▽(Term2*var`	`)+var ↩`		
		reduce 6	push Factor3
`▽(Term2*Factor3`	`)+var ↩`		
		reduce 3	push Term2
`▽(Term2`	`)+var ↩`		
		reduce 2	push Expr5
`▽(Expr5`	`)+var ↩`		
		shift)	
`▽(Expr5)`	`+var ↩`		
		reduce 5	push Factor4
`▽Factor4`	`+var ↩`		
		reduce 4	push Term2
`▽Term2`	`+var ↩`		
		reduce 2	push Expr1
`▽Expr1`	`+var ↩`		
		shift +	
`▽Expr1+`	`var ↩`		
		shift var	
`▽Expr1+var`	`↩`		
		reduce 6	push Factor4
`▽Expr1+Factor4`	`↩`		
		reduce4	push Term1
`▽Expr1+Term1`	`↩`		
		reduce 1	push Expr1
`▽Expr1`	`↩`		
		Accept	

(e)

Stack	Input	Action	Goto
`▽`	`(var*var ↩`	shift (	
`▽(`	`var*var ↩`	shift var	
`▽(var`	`*var ↩`	reduce 6	push Factor4
`▽(Factor4`	`*var ↩`	reduce 4	push Term2
`▽(Term2`	`*var ↩`	shift *	
`▽(Term2*`	`var ↩`	shift var	
`▽(Term2*var`	`↩`	reduce 6	push Factor3
`▽(Term2*Factor3`	`↩`	reduce 3	push Term2
`▽(Term2`	`↩`	reduce 2	push Expr5
`▽(Expr5`	`↩`	Syntax Error	

Exercises 5.3

1. Which of the following input strings would cause this SableCC program to produce a *syntax error* message?

```
Tokens
 a = 'a';
 b = 'b';
 c = 'c';
 newline = [10 + 13];
Productions
  line = s newline ;
  s =       {a1} a s b
          | {a2} b w c
          ;
  w =       {a1}  b w b
          | {a2} a c
          ;
```

(a)	`bacc`	(b)	`ab`	(c)	`abbacbcb`
(d)	`bbacbc`	(e)	`bbacbb`		

`(b,e)` are syntax errors

2 . Using the SableCC program from problem 1, show the *output* produced by each of the input strings given in Problem 1, using the Translation class shown below.

```
package  ex5_3;
import  ex5_3.analysis.*;
import  ex5_3.node.*;
import  java.util.*;
import  java.io.*;

class  Translation  extends  DepthFirstAdapter
{

public void outAA1S (AA1S node)
{   System.out.println ("rule 1"); }

public void outAA2S (AA2S node)
{   System.out.println ("rule 2"); }

public void outAA1W (AA1W node)
{   System.out.println ("rule 3"); }

public void outAA2W (AA2W node)
{   System.out.println ("rule 4"); }

}
```

(a) rule 4
rule 2

(b) Expecting 'a', 'b'

(c) rule 4
rule 3
rule 2
rule 1

(d) rule 4
rule 3
rule 2

(e) expecting 'c'

3. A ***Sexpr*** is an atom or a pair of Sexprs enclosed in parentheses and separated with a period. For example, if `A, B, C, ...Z` and `NIL` are all atoms, then the following are examples of Sexprs:

```
A
(A.B)
((A.B).(B.C))
(A.(B.(C.NIL)))
```

A ***List*** is a special kind of Sexpr. A List is the atom `NIL` or a List is a dotted pair of Sexprs in which the first part is an atom or a List and the second part is a List. The following are examples of lists:

```
NIL
(A.NIL)
((A.NIL).NIL)
((A.NIL).(B.NIL))
(A.(B.(C.NIL)))
```

(a) Show a *SableCC grammar* that defines a *Sexpr*.

```
// Exercise 5.3 #3

Package  ex5_3_3;


Tokens
 nil = 'NIL';
 letters = ['a'..'z']+;
 lparen = '(';
 rparen = ')';
 dot = '.';
 white  = [[10 + 13] + ' '];

Ignored  Tokens
  white ;

Productions
  sexpr = {single} atom
      | {pair} lparen [left]: sexpr dot [right]:
     sexpr  rparen
     ;

  atom = {nonil} letters
     | {yesnil} nil
     ;
```

(b) Show a *SableCC grammar* that defines a *List*.

```
// Exercise 5.3 #3

Package  ex5_3_3;

Tokens
 nil = 'NIL';
 letters = ['a'..'z']+;
 lparen = '(';
 rparen = ')';
 dot = '.';
 white  = [[10 + 13] + ' '];

Ignored  Tokens
  white ;

Productions

   list = {a} lparen letters dot list rparen
      | {b} lparen [car]: list dot [cdr]: list rparen
      | {c} nil
      ;
```

(c) Add a Translation class to your answer to part (b) so that it will print out the total number of atoms in a List. For example:

```
((A.NIL).(B.(C.NIL)))
5 atoms

// Translation class for exercise 5.3 #3
// July 2007,  sdb

package ex5_3_3;
import  ex5_3_3.analysis.*;
import  ex5_3_3.node.*;
import  java.util.*;
import  java.io.*;

class Translation extends DepthFirstAdapter
{
private int count = 0;

public void outAAList (AAList node )
{  count++; }

public void outACList (ACList node )
{  count++; }

public int getCount()
{  return count; }

}
```

4. Use SableCC to implement a *syntax checker* for a typical database command language. Your syntax checker should handle at least the following kinds of commands:

```
RETRIEVE  employee_file
PRINT
DISPLAY FOR salary >= 1000000
PRINT FOR "SMITH" = lastname
```

```
Package  ex5_3_4;    //  untested

Helpers
   letter = ['a'..'z'];
   digit  = ['0'..'9'];
States
   start, string;

Tokens
 retrieve = 'RETRIEVE';
 print = 'PRINT';
 display = 'DISPLAY';
 for = 'FOR';
 compare = '=' | '<' | '>' | '<=' | '>=' | '!=';
 identifier = letter (letter | digit)*;
 number = digit+;
 {start->string}quoteBegin  =  '"';
 {string}string  string  =  [[0..0xffff]-'"'];
  {string->start]  quoteEnd  =  '"';
 white   =  [[10 + 13] + ' '];

Ignored  Tokens
   white,  quoteBegin,  quoteEnd  ;
```

```
Productions

  stmt = {r] retrieve identifier
        | {p} print
        | {d} display for comparison
        | {p2} print for comparison
     ;

  comparison = value compare value;

   value = {id} identifier
        | {num} number
             ;
```

5. The following SableCC grammar and Translation class are designed to implement a simple desk calculator with the standard four arithmetic functions (it uses floating-point arithmetic only). When compiled and run, the program will evaluate a list of arithmetic expressions, one per line, and print the results. For example:

```
2+3.2e-2
2+3*5/2
(2+3)*5/2
16/(2*3 - 6*1.0)
     2.032
     9.5
     12.5
     infinity
```

Unfortunately, the grammar and Java code shown below are incorrect. There are four mistakes, some of which are syntactic errors in the grammar; some of which are syntactic Java errors; some of which cause run-time errors; and some of which don't produce any error messages, but do produce incorrect output. Find and correct all four mistakes. If possible, use a computer to help debug these programs.

The grammar, `exprs.grammar` is shown below:

```
Package  exprs;

Helpers
 digits = ['0'..'9']+ ;
 exp =    ['e' + 'E'] ['+' + '-']? digits ;
Tokens
 number = digits '.'? digits? exp? ;
 plus =     '+';
 minus =    '-';
 mult =     '*';
 div =      '/';
 l_par =     '(';
 r_par =     ')';
 newline = [10 + 13] ;
 blank =    (' ' | '\t')+;
 semi =     ';' ;

Ignored  Tokens
 blank;

Productions
 exprs =   {e1}  expr newline
         | {e2}  exprs embed
        ;
 embed = expr newline;
 expr =
        {term}   term |
        {plus}   expr plus term |
        {minus}  expr minus term
        ;
 term =
        {factor}  factor |
        {mult}    term mult factor |
        {div}     term div factor |
        ;
 factor =
        {number}  number |
        {paren}   l_par expr r_par
        ;
```

The Translation class is shown below:

```
package exprs;
import exprs.analysis.*;
import exprs.node.*;
import java.util.*;

class Translation extends DepthFirstAdapter
{
Hashtable hash = new Hashtable();       // store expr values

public void outAE1Exprs (AE1Exprs node)
{  System.out.println ("   " + getVal (node.getExpr())); }

public void outAEmbed (AEmbed node)
{  System.out.println ("   " + getVal (node.getExpr())); }

public void caseTNumber(TNumber node)
{ hash.put (node, new Double (node.toString())) ; }
```

public void outATermExpr (ATermExpr node)
{ // Value of the expr same as the term
hash.put (node, getVal (node.getTerm()));
}

```
public void outAPlusExpr(APlusExpr node)
{// out of alternative {plus} in Expr, we add the
 // expr and the term
   hash.put (node, new Double (getPrim (node.getExpr())
                + getPrim(node.getTerm())));
}

public void outAMinusExpr(AMinusExpr node)
{// out of alternative {minus} in Expr, subtract the term
 //  from the expr
   hash.put (node, new Double (getPrim(node.getExpr())
                - getPrim(node.getTerm())));
}

public void outAFactorTerm (AFactorTerm node)
{ //  Value of the term same as the factor
   hash.put (node, getVal(node.getFactor())) ;
}

public void outAMultTerm(AMultTerm node)
{// out of alternative {mult} in Factor, multiply the term
 // by the factor
   hash.put (node, new Double (getPrim(node.getTerm())
                * getPrim(node.getFactor())));
}

public void outADivTerm(ADivTerm node)
{// out of alternative {div} in Factor, divide the term by
 // the factor
   hash.put (node, new Double (getPrim(node.getTerm())
                / getPrim(node.getFactor())));
}

public void outANumberFactor (ANumberFactor node)
{   hash.put (node, getVal (node.getNumber())); }

public void outAParenFactor (AParenFactor node)
{   hash.put (node, getVal (node.getExpr())); }

double getPrim (Node node)
{  return ((Double) hash.get (node)).doubleValue(); }

Double getVal (Node node)
{  return  (Double) hash.get (node) ; }
}
```

6. Show the *SableCC grammar* which will check for proper syntax of regular expressions over the alphabet {0,1}. Some examples are shown:

<u>Valid</u>	<u>Not Valid</u>
`(0+1)*·1·1`	`*0`
`0·1·0*`	`(0+1)+1)`
`((0))`	`0+`

```
Package reg_exprs;

Tokens
 prim = '0' | '1';
 plus =    '+';
 dot = '.';
 star =    '*';
 lparen =    '(';
 rparen =    ')';
 blank =   (' ' | 10 | 13 | '\t')+;

Ignored Tokens
 blank;

Productions
 list =
        {a} list expr
        | {b} expr
        ;
expr =    {union}  expr plus term
        | {t} term
        ;
term =    {concat} term dot factor
        | {f} factor
        ;
factor =  {kleene} factor star
        | {paren} lparen expr rparen
        | {p} prim
        ;
```

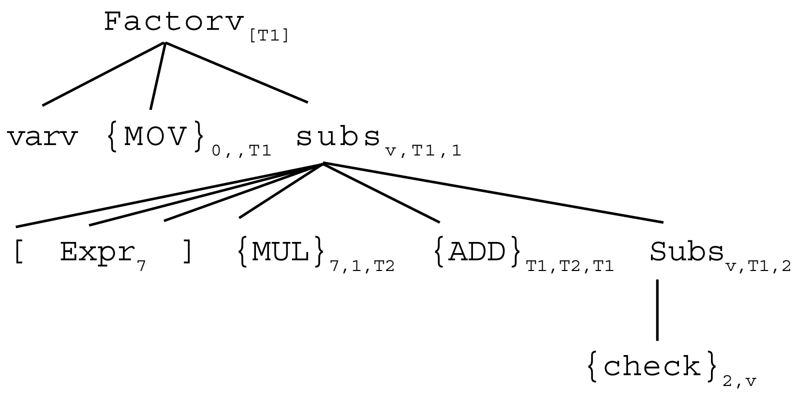
Exercises 5.4

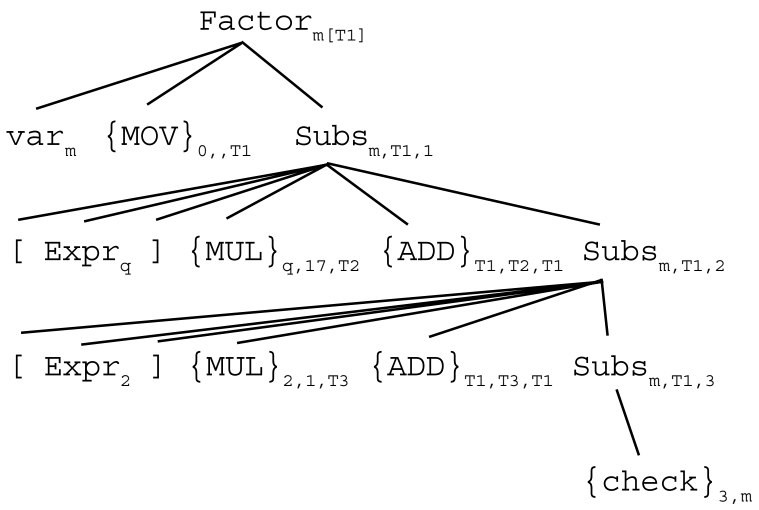
1. Assume the following array declarations:

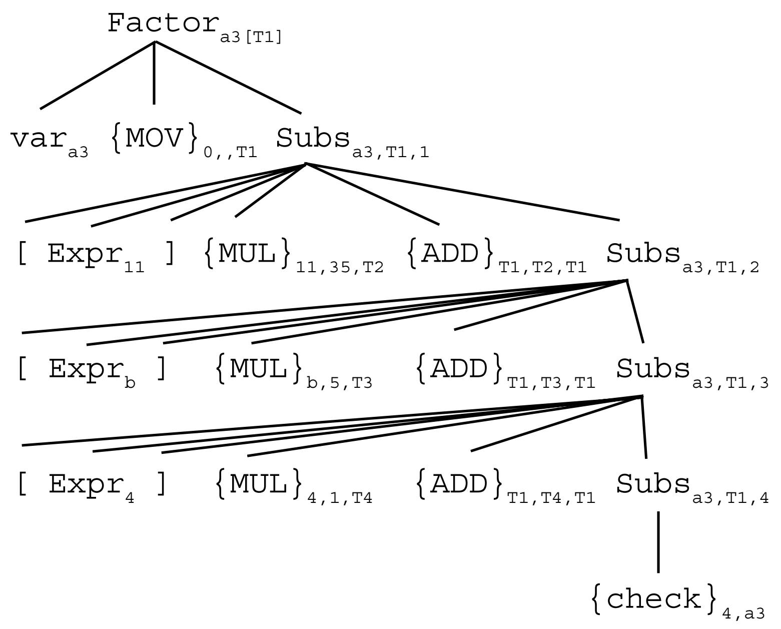
```
int v[] = new int [13];
int m[][] = new int [12][17];
int a3[][][] = new int [15][7][5];
int z[][][][] = new int [4][7][2][3];
```

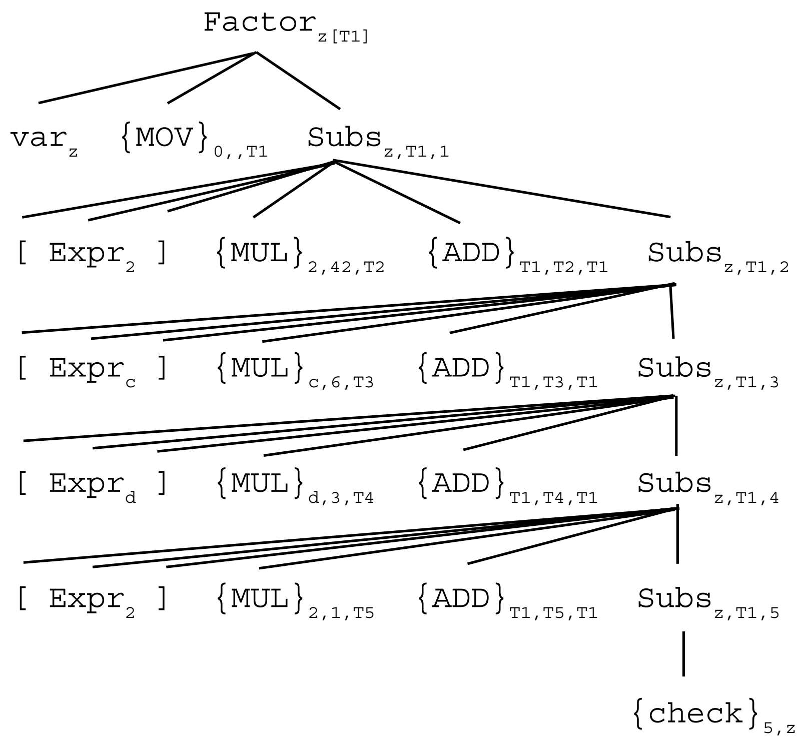
Show the *attributed derivation tree* resulting from grammar G23 for each of the following array references. Use `Factor` as the starting nonterminal, and show each subscript expression as `Expr`, as done in Figure 5.11. Also show the sequence of atoms that would be put out.

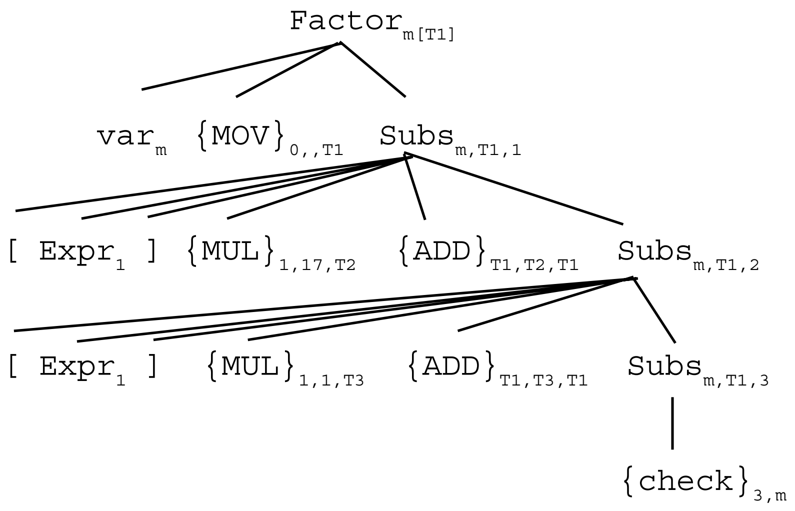
(a) `v[7]` (b) `m[q][2]` (c) `a3[11][b][4]`
(d) `z[2][c][d][2]` (e) `m[1][1]`






Factor$_{a3[T1]}$
var$_{a3}$
{MOV}$_{0,,T1}$
Subs$_{a3,T1,1}$
[Expr$_{11}$]
{MUL}$_{11,35,T2}$
{ADD}$_{T1,T2,T1}$
Subs$_{a3,T1,2}$
[Expr$_{b}$]
{MUL}$_{b,5,T3}$
{ADD}$_{T1,T3,T1}$
Subs$_{a3,T1,3}$
[Expr$_{4}$]
{MUL}$_{4,1,T4}$
{ADD}$_{T1,T4,T1}$
Subs$_{a3,T1,4}$
{check}$_{4,a3}$


Factor$_{z[T1]}$
var$_{z}$
{MOV}$_{0,,T1}$
Subs$_{z,T1,1}$
[Expr$_{2}$]
{MUL}$_{2,42,T2}$
{ADD}$_{T1,T2,T1}$
Subs$_{z,T1,2}$
[Expr$_{c}$]
{MUL}$_{c,6,T3}$
{ADD}$_{T1,T3,T1}$
Subs$_{z,T1,3}$
[Expr$_{d}$]
{MUL}$_{d,3,T4}$
{ADD}$_{T1,T4,T1}$
Subs$_{z,T1,4}$
[Expr$_{2}$]
{MUL}$_{2,1,T5}$
{ADD}$_{T1,T5,T1}$
Subs$_{z,T1,5}$
{check}$_{5,z}$



2. The discussion in this section assumed that each array element occupied one memory cell. If each array element occupies `SIZE` memory cells, what changes would have to be made to the general *formula* given in this section for the *offset*? How would this affect grammar G23?

Multiply the entire forumula by SIZE. Change rule 7 of Grammar G23 as follows:

7. $\texttt{Factor}_e \rightarrow \texttt{var}_v\ \{\texttt{MOV}\}_{0,,sum}\ \texttt{Subs}_{v,sum,i}\ \{\texttt{MUL}\}_{sum,SIZE,r}$

```
r ← Alloc
e ← v[r]
i ← 1
sum ← Alloc
```

3. You are given two vectors: the first, `d`, contains the dimensions of a declared array, and the second, `s`, contains the subscripting values in a reference to that array.

(a) Write a *Java method* –

```
int offSet (int d[], int s[]);
```

that computes the offset for an array reference `a[S`$_0$`] [S`$_1$`] ... [S`$_{max-1}$`]` where the array has been declared as `char a[d`$_0$`] [d`$_1$`] ... [d`$_{max-1}$`]`.

(b) *Improve* your Java method, if possible, to minimize the number of run-time multiplications.

```
public int offSet (int d[], int s[])
{       int prod = 1;
        int result = 0;
        int max = d.length;

        for (int i=max-1; i>=0; i--)
           {     result += s[i]*prod;
                 prod *= d[i];
           }
        return result;
}
```

Exercises 5.5

1. Extend the Decaf language to include a `do` statement defined as:

```
DoStmt → do Stmt while ( BoolExpr ) ;
```

Modify the files `decaf.grammar` and `Translation.java`, shown in Appendix B so that the compiler puts out the correct *atom* sequence implementing this control structure, in which the test for termmination is made after the body of the loop is executed. The nonterminals `Stmt` and `BoolExpr` are already defined. For purposes of this assignment you may alter the `atom` method so that it prints out its arguments to `stdout` rather than building a file of atoms.

2. Extend the Decaf language to include a `switch` statement defined as:

```
SwitchStmt  →  switch ( Expr ) CaseList
CaseList  →  case number ':' Stmt CaseList
CaseList  →  case number ':' Stmt
```

Modify the files `decaf.grammar` and `Translation.java`, shown in Appendix B, so that the compiler puts out the correct *atom* sequence implementing this control structure. The nonterminals `Expr` and `Stmt` are already defined, as are the tokens `number` and `end`. The token `switch` needs to be defined. Also define a `break` statement which will be used to transfer control out of the `switch` statement. For purposes of this assignment, you may alter the `atom()` function so that it prints out its arguments to `stdout` rather than building a file of atoms, and remove the call to the code generator.

3. Extend the Decaf language to include initializations in decalarations, such as:

```
int x=3, y, z=0;
```

Modify the files `decaf.grammar` and `Translation.java`, shown in Appendix B, so that the compiler puts out the correct *atom* sequence implementing this feature. You will need to put out a `MOV` atom to assign the value of the constant to the variable.

Solution to problems 1,2,3:

```
//       decaf.grammar
// SableCC grammar for decaf, a subset of Java.
// March 2003,  sdb
// Exercises 5.5  #1,2,3
// August 2007,  sdb

Package decaf;
Helpers                                         // Examples
  letter = ['a'..'z'] | ['A'..'Z'] ;            //   w
  digit =    ['0'..'9'] ;                       //   3
  digits =   digit+ ;                           // 2040099
```

```
        exp  =     ['e' + 'E'] ['+' + '-']? digits;         // E-34


  newline = [10 + 13]   ;
  non_star = [[0..0xffff] - '*'];
  non_slash = [[0..0xffff] - '/'];
  non_star_slash = [[0..0xffff] - ['*' + '/']];


Tokens
  comment1 = '//' [[0..0xffff]-newline]* newline ;
  comment2 = '/*' non_star* '*' (non_star_slash non_star* '*'+)* '/' ;

  space = ' ' | 9 | newline ;       // '\t' (=9) doesn't work?
  break = 'break' ;
  clas = 'class' ;      // key words (reserved)
  case = 'case' ;
  do = 'do';
  else = 'else' ;
  float = 'float' ;
  for = 'for' ;
  if = 'if' ;
  int = 'int' ;
  main = 'main' ;
  public = 'public' ;
  static = 'static' ;
  string = 'String' ;
  switch = 'switch' ;
  void = 'void' ;
  while = 'while' ;
  assign = '=' ;
  compare = '==' | '<' | '>' | '<=' | '>=' | '!=' ;
  plus = '+' ;
  minus = '-' ;
  mult = '*' ;
  div = '/' ;
  l_par = '(' ;
  r_par = ')' ;
  l_brace = '{' ;
  r_brace = '}' ;
  l_bracket = '[' ;
  r_bracket = ']' ;
  comma = ',' ;
  semi = ';' ;
  colon = ':' ;
  identifier = letter (letter | digit | '_')* ;
  number  =  (digits '.'? digits? | '.'digits) exp? ; // 2.043e+5
  misc = [0..0xffff] ;


Ignored Tokens
  comment1, comment2, space;


Productions

  program =       clas identifier l_brace public static void main l_par
        string l_bracket r_bracket [arg]: identifier r_par
                    compound_stmt r_brace ;
  type =            {int}     int
                  | {float}   float ;
  declaration =   type identifier init identlist* semi;
  identlist =           comma identifier init ;
  init =          {non_null} assign rvalue
                  | {null}
                  ;

//////////////////////////////////////////////
// Adapted from Appel, Java version, 2nd ed.
//   to eliminate shift-reduce conflict resulting from dangling else.
// yacc seems to work ok despite the conflict (default is to shift,
which works),
// Recursive descent also works fine despite the conflict (and ambigu-
ity).
// SableCC will not generate code if there is a conflict.
// That is the reason for the "short-if" constructs below.
//////////////////////////////////////////////
  stmt =            {dcl}                 declaration
                  | {stmt_no_trlr}  stmt_no_trailer
                  | {if_st}         if_stmt
                  | {if_else_st}          if_else_stmt
                  | {while_st}            while_stmt
                  | {for_st}        for_stmt
                  | {do_while_st}         do_while_stmt
                  | {switch_st}           switch_stmt
                  ;
  stmt_no_short_if =      {stmt_no_trlr}  stmt_no_trailer
                  | {if_else_no_short}    if_else_stmt_no_short_if
                  | {while_no_short}      while_stmt_no_short_if
```

```
// The following causes a reduce/reduce conflict
//                | {do_while_no_short}   do_while_stmt_no_short_if
                  | {do_while_no_short}   do_while_stmt
                  | {switch_no_short}     switch_stmt
                  | {for_no_short}  for_stmt_no_short_if
                  ;
  stmt_no_trailer =       {compound}      compound_stmt
                  | {null}    semi
                  | {assign}  assign_stmt
                  ;

  assign_stmt =   assign_expr semi ;

  for_stmt =                  for l_par assign_expr? semi bool_expr?
                          [s2]: semi [a2]: assign_expr? r_par stmt ;
  for_stmt_no_short_if =      for l_par assign_expr? semi bool_expr?
                          [s2]: semi [a2]: assign_expr? r_par
stmt_no_short_if ;

  while_stmt =                while l_par bool_expr r_par stmt ;
  while_stmt_no_short_if =    while l_par bool_expr r_par
stmt_no_short_if ;

// ex 5.5.1
  do_while_stmt = do stmt while l_par bool_expr r_par semi ;
 // do_while_stmt_no_short_if = do stmt_no_short_if while l_par
bool_expr r_par semi ;

  if_stmt =             if l_par bool_expr r_par stmt ;
  if_else_stmt =        if l_par bool_expr r_par stmt_no_short_if else
stmt ;
  if_else_stmt_no_short_if =  if l_par bool_expr r_par [if1]:
stmt_no_short_if
                          else [if2]: stmt_no_short_if ;

  compound_stmt =  l_brace stmt* r_brace ;

// ex 5.5.2
  switch_stmt = switch l_par expr r_par l_brace case_list r_brace ;
//  case_list =   ( case number colon stmt* break_stmt ) * ;
    case_list =      {non_null}     case number colon stmt* break_stmt
case_list
                 | {null}
             ;
  break_stmt =      {non_null} break semi
            | {null}
            ;

  bool_expr =     expr compare [right]: expr ;

  expr =      {assn} assign_expr
            | {rval} rvalue
            ;
  assign_expr =   identifier assign expr ;
  rvalue =    {plus}  rvalue plus term
            | {minus} rvalue minus term
            | {term}  term
            ;
  term =      {mult}  term mult factor
            | {div}   term div factor
            | {fac}   factor
            ;
  factor =    {pars}   l_par expr r_par
            | {uplus}  plus factor
            | {uminus} minus factor
            | {id}     identifier
            | {num}    number
            ;
========================================

//         Translation.java
// Translation class for decaf, a subset of Java.
// Output atoms from syntax tree
//   sdb  March 2003
//   sdb  updated May 2007
//      to use generic maps instead of hashtables.
//  Exercises 5.5  #1,2,3     Aug 2007  sdb

package decaf;
import decaf.analysis.*;
import decaf.node.*;
import java.util.*;
import java.io.*;

class Translation extends DepthFirstAdapter
{

// All stored values are doubles, key=node, value is memory loc or label
```

```
number
// Map  <Node, Integer> hash = new HashMap <Node, Integer> ();    // May
2007
   Map  hash = new HashMap();                          // Aug 2007

Integer zero = new Integer (0);
Integer one  = new Integer (1);

AtomFile out;

//////////////////////////////////////////////////
// Definition of Program

public void inAProgram (AProgram prog)
//   The class name and main args need to be entered into symbol table
//     to avoid error message.
//   Also, open the atom file for output
{  identifiers.put (prog.getIdentifier().toString(), alloc());    //
class name
   identifiers.put (prog.getArg().toString(), alloc());           //
main (args)
   out = new AtomFile ("atoms");
}

public void outAProgram (AProgram prog)
//  Write the run-time memory values to a file "constants".
//  Close the binary file of atoms so it can be used for
//    input by the code generator
{  outConstants();
   out.close();
}

//////////////////////////////////////////////////
// Definitions of declaration and identlist

public void inADeclaration (ADeclaration node)
{   install (node.getIdentifier()); }

// ex 5.5.3
public void caseADeclaration (ADeclaration node)
{   inADeclaration (node);
    if (node.getType() != null)
      node.getType().apply(this);
    if (node.getIdentifier() != null)
      node.getIdentifier().apply(this);
    if (node.getInit() != null)
      node.getInit().apply(this);
    Integer p,q;
    p = getIdent (node.getIdentifier());
    q = (Integer) hash.get (node.getInit());
    atom ("MOV", q, 0, p);
    List <PIdentlist> copy = new ArrayList <PIdentlist>
      (node.getIdentlist());
    for (PIdentlist e : copy)
      e.apply(this);
    if (node.getSemi() != null)
      node.getSemi().apply(this);
    outADeclaration(node);
}


// ex 5.5.3
public void outAIdentlist (AIdentlist node)
{   install (node.getIdentifier());
    Integer p,q;
    q = (Integer) hash.get (node.getInit());
    if (q != null)            // initialization is optional
     {      p = getIdent (node.getIdentifier());
      atom ("MOV", q, 0, p);
     }
}

void install (TIdentifier id)
//  Install id into the symbol table
{  Integer loc;
   loc = identifiers.get (id.toString());
   if (loc==null)
      identifiers.put (id.toString(), alloc());
   else
      System.err.println ("Error: " + id + " has already been declared
");
}

//////////////////////////////////////////////////
// Definition of init
public void outANonNullInit (ANonNullInit node)
{
   Integer p = (Integer) hash.get(node.getRvalue());
   hash.put(node, p);
}
```

```
//////////////////////////////////////////////////
// Definition of for_stmt
public void caseAForStmt (AForStmt stmt)
{  Integer lbl1, lbl2, lbl3;
   lbl1 = lalloc();
   lbl2 = lalloc();
   lbl3 = lalloc();
   inAForStmt (stmt);
   if (stmt.getFor() !=null)  stmt.getFor().apply(this);
   if (stmt.getLPar() !=null) stmt.getLPar().apply(this);
   if (stmt.getAssignExpr()!=null)              // initialize
      {     stmt.getAssignExpr().apply(this);
            atom ("LBL", lbl1);
      }
   if (stmt.getSemi() != null)      stmt.getSemi().apply(this);
   if (stmt.getBoolExpr() != null)              // test for termination
      {     stmt.getBoolExpr().apply(this);
            atom ("JMP", lbl2);
            atom ("LBL", lbl3);
      }
   if (stmt.getS2() != null)  stmt.getS2().apply(this);
   if (stmt.getA2() != null)
      {     stmt.getA2().apply(this);           // increment
            atom ("JMP", lbl1);
            atom ("LBL", lbl2);
      }
   if (stmt.getRPar() != null)      stmt.getRPar().apply(this);
   if (stmt.getStmt() != null)
      {     stmt.getStmt().apply(this);
            atom ("JMP", lbl3);
            atom ("LBL", (Integer) hash.get (stmt.getBoolExpr()));
      }
   outAForStmt(stmt);
}

public void caseAForStmtNoShortIf (AForStmtNoShortIf stmt)
{  Integer lbl1, lbl2, lbl3;
   lbl1 = lalloc();
   lbl2 = lalloc();
   lbl3 = lalloc();
   inAForStmtNoShortIf (stmt);
   if (stmt.getFor() !=null)  stmt.getFor().apply(this);
   if (stmt.getLPar() !=null) stmt.getLPar().apply(this);
   if (stmt.getAssignExpr()!=null)              // initialize
```

```
      {     stmt.getAssignExpr().apply(this);
            atom ("LBL", lbl1);
      }
   if (stmt.getSemi() != null)      stmt.getSemi().apply(this);
   if (stmt.getBoolExpr() != null)              // test for termination
      {     stmt.getBoolExpr().apply(this);
            atom ("JMP", lbl2);
            atom ("LBL", lbl3);
      }
   if (stmt.getS2() != null)  stmt.getS2().apply(this);
   if (stmt.getA2() != null)
      {     stmt.getA2().apply(this);           // increment
            atom ("JMP", lbl1);
            atom ("LBL", lbl2);
      }
   if (stmt.getRPar() != null)      stmt.getRPar().apply(this);
   if (stmt.getStmtNoShortIf() != null)
      {     stmt.getStmtNoShortIf().apply(this);
            atom ("JMP", lbl3);
            atom ("LBL", (Integer) hash.get (stmt.getBoolExpr()));
      }
   outAForStmtNoShortIf (stmt);
}

//////////////////////////////////////////////////////
// Definition of switch statement
// exercise 5.5.2
// Nodes with more than one attribute use a list of parms
//   as the attribute in the hash table.

public void caseASwitchStmt (ASwitchStmt stmt)
{  Integer p, lbl1, lbl2;
   lbl1 = lalloc();
   lbl2 = lalloc();
   inASwitchStmt (stmt);
   if (stmt.getSwitch() != null)    stmt.getSwitch().apply(this);
   if (stmt.getLPar() != null)            stmt.getLPar().apply(this);
   if (stmt.getExpr() != null)            stmt.getExpr().apply(this);
   p = (Integer) hash.get(stmt.getExpr());
   if (stmt.getRPar() != null)            stmt.getRPar().apply(this);
   if (stmt.getLBrace() != null)    stmt.getLBrace().apply(this);
   if (stmt.getCaseList() != null)
     { List parms = new ArrayList();
      parms.add (p);
      parms.add (lbl1);
      parms.add (lbl2);
      hash.put (stmt.getCaseList(), parms);
      stmt.getCaseList().apply(this);
    }
   atom ("LBL", lbl2);
   if (stmt.getRBrace() != null)    stmt.getRBrace().apply(this);
   outASwitchStmt (stmt);
}

public void caseANonNullCaseList (ANonNullCaseList stmt)
{  Integer p, q, lbl1, lbl2=0, lbl3, lbl4;
   lbl3 = lalloc();
   lbl4 = lalloc();
   inANonNullCaseList (stmt);
   if (stmt.getCase() != null)            stmt.getCase().apply(this);
   if (stmt.getNumber() != null)    stmt.getNumber().apply(this);
   p = (Integer) ((List)hash.get(stmt)).get(0);
   q = (Integer) hash.get(stmt.getNumber());
   atom ("TST",p,q,0,6,lbl3);
   if (stmt.getColon() != null)           stmt.getColon().apply(this);
   lbl1 = (Integer) ((List) hash.get(stmt)).get(1);
   atom ("LBL", lbl1);
   if (stmt.getStmt() != null)
      {  List <PStmt> stmts =  stmt.getStmt();
         for (PStmt st : stmts)
{ System.out.println ("In for loop " + st);     // diagnostic
            st.apply(this);
}
      }
   if (stmt.getBreakStmt() != null)
    {
      List parms = new ArrayList();
      lbl2 = (Integer) ((List)hash.get(stmt)).get(2);
      parms.add (lbl2);
      parms.add (lbl4);
      hash.put (stmt.getBreakStmt(), parms);
      stmt.getBreakStmt().apply(this);
    }
   atom ("LBL", lbl3);
   if (stmt.getCaseList() != null)
    { List parms = new ArrayList();
      parms.add (p);
      parms.add (lbl4);
      parms.add (lbl2);
      hash.put (stmt.getCaseList(), parms);
      stmt.getCaseList().apply(this);
     }
   outANonNullCaseList (stmt);
}

public void caseANullCaseList (ANullCaseList stmt)
{  inANullCaseList (stmt);
   atom ("LBL", (Integer) ((List)hash.get(stmt)).get(1));
   outANullCaseList (stmt);
}


public void caseANonNullBreakStmt (ANonNullBreakStmt stmt)
{  inANonNullBreakStmt (stmt);
   if (stmt.getBreak() != null)            stmt.getBreak().apply(this);
   if (stmt.getSemi() != null)             stmt.getSemi().apply(this);
   Integer lbl1 =  (Integer) ((List) hash.get(stmt)).get(0);
   atom ("JMP", lbl1);
   outANonNullBreakStmt (stmt);
}

public void caseANullBreakStmt (ANullBreakStmt stmt)
{  inANullBreakStmt (stmt);
   Integer lbl2 = (Integer)  ((List) hash.get(stmt)).get(1);
   atom ("JMP", lbl2);
   outANullBreakStmt(stmt);
}


//////////////////////////////////////////////////
// Definition of while_stmt
public void inAWhileStmt (AWhileStmt stmt)
{  Integer lbl = lalloc();
   hash.put (stmt, lbl);
   atom ("LBL", lbl);
}

public void outAWhileStmt (AWhileStmt stmt)
{  atom ("JMP", (Integer) hash.get(stmt));
   atom ("LBL", (Integer) hash.get (stmt.getBoolExpr()));
}

public void inAWhileStmtNoShortIf (AWhileStmtNoShortIf stmt)
{  Integer lbl = lalloc();
   hash.put (stmt, lbl);
   atom ("LBL", lbl);
}

public void outAWhileStmtNoShortIf (AWhileStmtNoShortIf stmt)
{  atom ("JMP", (Integer) hash.get(stmt));
   atom ("LBL", (Integer) hash.get (stmt.getBoolExpr()));
}

///////////////////////////////////////////////
// Definition of do_while_stmt            ex 5.5.1
public void inADoWhileStmt (ADoWhileStmt stmt)
{  Integer lbl = lalloc();
   hash.put (stmt, lbl);
   atom ("LBL", lbl);
}

public void outADoWhileStmt (ADoWhileStmt stmt)
{  atom ("JMP", (Integer) hash.get(stmt));
   atom ("LBL", (Integer) hash.get (stmt.getBoolExpr()));
}

/////////////////////////////////////////////
// Definition of if_stmt

public void outAIfStmt (AIfStmt stmt)
{  atom ("LBL", (Integer) hash.get (stmt.getBoolExpr())); } // Target
for bool_expr's TST

// override the case of if_else_stmt
public void caseAIfElseStmt (AIfElseStmt node)
{  Integer lbl = lalloc();
   inAIfElseStmt (node);
   if (node.getIf() != null) node.getIf().apply(this);
   if (node.getLPar() != null) node.getLPar().apply(this);
   if (node.getBoolExpr() != null)node.getBoolExpr().apply(this);
   if (node.getRPar() != null) node.getRPar().apply(this);
   if (node.getStmtNoShortIf() != null)
     {       node.getStmtNoShortIf().apply(this);
      atom ("JMP", lbl);                             // Jump over else
part
      atom ("LBL", (Integer) hash.get (node.getBoolExpr()));
     }
   if (node.getElse() != null) node.getElse().apply(this);
   if  (node.getStmt() != null) node.getStmt().apply(this);
   atom ("LBL", lbl);
   outAIfElseStmt (node);
}

// override the case of if_else_stmt_no_short_if
public void caseAIfElseStmtNoShortIf (AIfElseStmtNoShortIf node)
{  Integer lbl = lalloc();
   inAIfElseStmtNoShortIf (node);
   if (node.getIf() != null) node.getIf().apply(this);
   if (node.getLPar() != null) node.getLPar().apply(this);
   if (node.getBoolExpr() != null)node.getBoolExpr().apply(this);
   if (node.getRPar() != null) node.getRPar().apply(this);
   if (node.getIf1() != null)
     {      node.getIf1().apply(this);
      atom ("JMP", lbl);                              // Jump over else
part
      atom ("LBL", (Integer) hash.get (node.getBoolExpr()));
     }
   if (node.getElse() != null) node.getElse().apply(this);
   if  (node.getIf2() != null) node.getIf2().apply(this);
   atom ("LBL", lbl);
   outAIfElseStmtNoShortIf (node);
}

///////////////////////////////////////////////////
// Definition of bool_expr

public void outABoolExpr (ABoolExpr node)
{  Integer lbl = lalloc();
   hash.put (node, lbl);
   atom ("TST", (Integer) hash.get(node.getExpr()),
            (Integer) hash.get(node.getRight()),
            zero,
            new Integer (7  - getComparisonCode
(node.getCompare().toString())),
                        // Negation of a comparison code is 7 - code.
            lbl);
}


////////////////////////////////////////////////
// Definition of expr

public void outAAssnExpr (AAssnExpr node)
// out of alternative {assn} in expr
{ hash.put (node, hash.get (node.getAssignExpr())); }
```

```
public void outARvalExpr (ARvalExpr node)
// out of alternative {rval} in expr
{  hash.put (node, hash.get (node.getRvalue())); }


int getComparisonCode (String cmp)
//  Return the integer comparison code for a comparison
{   if (cmp.indexOf ("==")>=0) return  1;
    if (cmp.indexOf ("<")>=0)  return  2;
    if (cmp.indexOf (">")>=0)  return  3;
    if (cmp.indexOf ("<=")>=0) return  4;
    if (cmp.indexOf (">=")>=0) return  5;
    if (cmp.indexOf ("!=")>=0) return  6;
  return  0;                  // this should never occur
}

////////////////////////////////////////////////
// Definition of assign_expr

public void outAAssignExpr (AAssignExpr node)
//  Put out the MOV atom
{  Integer assignTo = getIdent (node.getIdentifier());
   atom ("MOV", (Integer) hash.get (node.getExpr()),
       zero,
       assignTo);
   hash.put (node, assignTo);
}


////////////////////////////////////////////////
// Definition of rvalue

public void outAPlusRvalue (APlusRvalue node)
{// out of alternative {plus} in Rvalue, generate an atom ADD.
    Integer i = alloc();
    hash.put (node, i);
    atom ("ADD", (Integer)hash.get(node.getRvalue()),
               (Integer) hash.get(node.getTerm()) , i);
}

public void outAMinusRvalue(AMinusRvalue node)
{// out of alternative {minus} in Rvalue, generate an atom SUB.
    Integer i = alloc();
    hash.put (node, i);
    atom ("SUB", (Integer) hash.get(node.getRvalue()),
              (Integer) hash.get(node.getTerm()), i);
}

public void outATermRvalue (ATermRvalue node)
//  Attribute of the rvalue is the same as the term.
{   hash.put (node, hash.get (node.getTerm())); }


///////////////////////////////////////////////
// Definition of term

public void outAMultTerm (AMultTerm node)
{// out of alternative {mult} in Term, generate an atom MUL.
    Integer i = alloc();
    hash.put (node, i);
    atom ("MUL", (Integer)hash.get(node.getTerm()),
                (Integer) hash.get(node.getFactor()) , i);
}

public void outADivTerm(ADivTerm node)
{// out of alternative {div} in Term, generate an atom DIV.
    Integer i = alloc();
    hash.put (node, i);
    atom ("DIV", (Integer) hash.get(node.getTerm()),
              (Integer) hash.get(node.getFactor()), i);
}

public void outAFacTerm (AFacTerm node)
{ //  Attribute of the term is the same as the factor
      hash.put (node, hash.get(node.getFactor()));
 }


//  May 2007
Map <Double, Integer> nums = new HashMap <Double, Integer> ();
Map <String, Integer > identifiers = new HashMap <String, Integer> ();

final int MAX_MEMORY = 1024;
Double memory [] = new Double [MAX_MEMORY];
int memHigh = 0;
// No, only memory needs to remain for codegen.


// Maintain a hash table of numeric constants, to avoid storing
//   the same number twice.
// Move the number to a run-time memory location.
// That memory location will be the attribute of the Number token.
public void caseTNumber(TNumber num)
{ Integer loc;
  Double dnum;
  dnum = new Double (num.toString());           // The number as a
Double
  loc = (Integer) nums.get (dnum);        // Get its memory location
  if (loc==null)                    // Already in table?
     {      loc = alloc();                      // No, install in table
of nums
      nums.put (dnum, loc);
      memory[loc.intValue()] = dnum;            // Store value in run-
time memory
      if (loc.intValue() > memHigh)       // Retain highest memory loc
         memHigh = loc.intValue();
     }
  hash.put (num, loc);                    // Set attribute to move up
tree
}



Integer getIdent(TIdentifier id)
// Get the run-time memory location to which this id is bound
{  Integer loc;
   loc = identifiers.get (id.toString());
   if (loc==null)
      System.err.println ("Error: " + id + " has not been declared");
   return loc;
}

////////////////////////////////////////////////
// Definition of factor

public void outAParsFactor (AParsFactor node)
{   hash.put (node, hash.get (node.getExpr())); }

// Unary + doesn't need any atoms to be put out.
public void outAUplusFactor (AUplusFactor node)
{   hash.put (node, hash.get (node.getFactor())); }

// Unary - needs a negation atom (NEG).
public void outAUminusFactor (AUminusFactor node)
```

```
{   Integer loc = alloc();    // result of negation
    atom ("NEG", (Integer)hash.get(node.getFactor()), zero, loc);
    hash.put (node, loc);
}

public void outAIdFactor (AIdFactor node)
{   hash.put (node, getIdent (node.getIdentifier())); }

public void outANumFactor (ANumFactor node)
{   hash.put (node, hash.get (node.getNumber())); }


/////////////////////////////////////////////////////////////////////
//  Send the run-time memory constants to a file for use by the code
generator.

void outConstants()
{  FileOutputStream fos = null;
   DataOutputStream ds = null;
   int i;

   try
    { fos = new FileOutputStream ("constants");
      ds = new DataOutputStream (fos);
    }
   catch (IOException ioe)
    { System.err.println ("IO error opening constants file for output: "
            + ioe);
    }

   try
    { for (i=0; i<=memHigh ; i++)
          if (memory[i]==null) ds.writeDouble (0.0);        // a vari-
able is bound here
          else
            ds.writeDouble (memory[i].doubleValue());
    }
   catch (IOException ioe)
    { System.err.println ("IO error writing to constants file: "
            + ioe);
    }
   try { fos.close(); }
   catch (IOException ioe)
    { System.err.println ("IO error closing constants file: "
            + ioe);
    }
}


//////////////////////////////////////////////////////////
// Put out atoms for conversion to machine code.
// These methods display to stdout, and also write to a
//   binary file of atoms suitable as input to the code generator.

void atom (String atomClass, Integer left, Integer right,
            Integer result)
{   System.out.println (atomClass +  " T" + left + " T"  + right +
              " T" + result);
    Atom atom = new Atom (atomClass, left, right, result);
    atom.write(out);
}

void atom (String atomClass, Integer left, Integer right,
            Integer result,
      Integer cmp, Integer lbl)
{   System.out.println (atomClass +  " T" + left + " T"  + right +  " T"
+
      result + " C" + cmp + " L" + lbl);
    Atom atom = new Atom (atomClass, left, right, result, cmp, lbl);
    atom.write(out);
}

void atom (String atomClass, Integer lbl)
{   System.out.println (atomClass +  " L" + lbl);
    Atom atom = new Atom (atomClass, lbl);
    atom.write(out);
}

static int avail = 0;
static int lavail = 0;

Integer alloc()
{ return new Integer (++avail); }

Integer lalloc()
{ return new Integer (++lavail); }


}
```

Chapter 6

Exercises 6.1

1. Show the big C notation for each of the following compilers (assume that each uses an intermediate form called "Atoms"):

(a) The back end of a compiler for the Sun computer.

$$C^{\text{Atoms} \rightarrow \text{Sun}}_{\text{Sun}}$$

(b) The source code, in Pascal, for a COBOL compiler whose target machine is the PC.

$$C^{\text{COBOL} \rightarrow \text{PC}}_{\text{Pascal}}$$

(c) The souce code, in Pascal, for the back end of a FORTRAN compiler for the Sun.

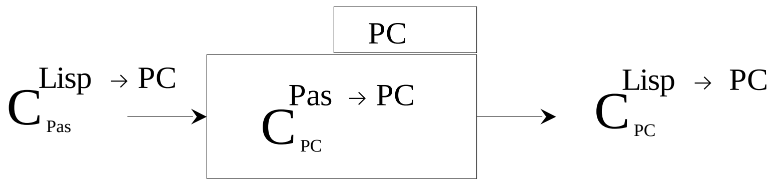
$$C^{\text{Atoms} \rightarrow \text{Sun}}_{\text{Pascal}}$$

2. Show how to generate

$$C^{\text{Lisp} \rightarrow \text{PC}}_{\text{PC}}$$

without writing any more programs, given a PC machine and each of the following collections of compilers:

(a) $C^{\text{Lisp} \rightarrow \text{PC}}_{\text{Pas}}$ $\qquad C^{\text{Pas} \rightarrow \text{PC}}_{\text{PC}}$



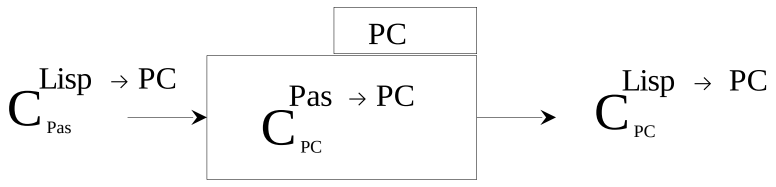
$C^{Lisp \rightarrow PC}_{Pas}$
PC
$C^{Pas \rightarrow PC}_{PC}$
$C^{Lisp \rightarrow PC}_{PC}$

(b)

$C^{Lisp \rightarrow Atoms}_{Pas}$ $\quad$ $C^{Pas \rightarrow Atoms}_{Pas}$

$C^{Atoms \rightarrow PC}_{Pas}$ $\quad$ $C^{Pas \rightarrow PC}_{PC}$

$$C^{Lisp \rightarrow PC}_{Pas} = C^{Lisp \rightarrow Atoms}_{Pas} + C^{Atoms \rightarrow PC}_{Pas}$$



$C^{Lisp \rightarrow PC}_{Pas}$
PC
$C^{Pas \rightarrow PC}_{PC}$
$C^{Lisp \rightarrow PC}_{PC}$

(c)

$$C^{Lisp \rightarrow PC}_{PC} = C^{Lisp \rightarrow Atoms}_{PC} + C^{Atoms \rightarrow PC}_{PC}$$

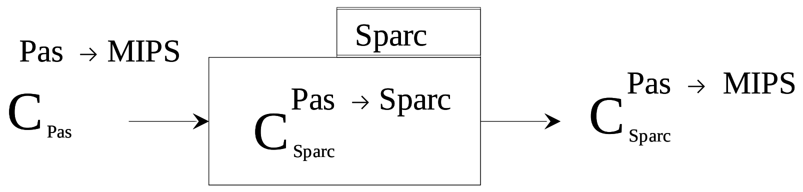
3. Given a Sparc computer and the following compilers, show how to generate a Pascal (Pas) compiler for the MIPS machine without doing any more programming. (Unfortunately, you can't afford to buy a MIPS computer.)

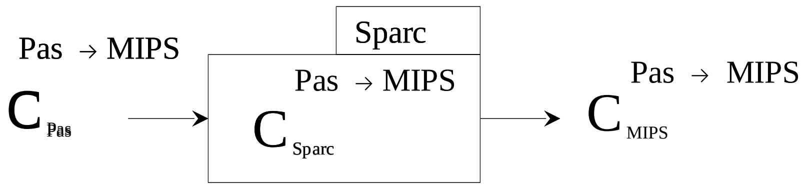
$$C^{Pas \to Sparc}_{Pas} = C^{Pas \to Atoms}_{Pas} + C^{Atoms \to Sparc}_{Pas}$$

$$C^{Pas \to Sparc}_{Sparc} = C^{Pas \to Atoms}_{Sparc} + C^{Atoms \to Sparc}_{Sparc}$$

$$C^{Atoms \to MIPS}_{Pas}$$

$$C^{Pas \to MIPS}_{Pas} = C^{Pas \to Atoms}_{Pas} + C^{Atoms \to MIPS}_{Pas}$$





Exercises 6.2

1. For each of the following Java statements we show the atom string produced by the parser. Translate each atom string to *instructions*, as in the sample problem for this section. You may assume that variables and labels are represented by symbolic addresses.

(a)
```
{     a = b + c * (d - e) ;
      b = a;
}

(SUB, d, e, T1)           lod  r1,d
                          sub  r1,e
                          sto  r1,t1
(MUL, c, T1, T2)          lod  r1,c
                          mul  r1,t1
                          sto  r1,t2
(ADD, b, T2, T3)          lod  r1,b
                          add  r1,t2
                          sto  r1,t3
(MOV, T3,, a)             lod  r1,t3
                          sto  r1,a
(MOV, a,, b)              lod  r1,a
                          sto  r1,b
```

(b)
```
for (i=1; i<=10; i++) j = j/3 ;

(MOV, 1,, i)              lod  r1,1
                          sto  r1,i
(LBL, L1)            l1:
(TST, i, 10,, 3, L4)      lod  r1,i
                          cmp  r1,t1,3
                          jmp  l4
(JMP, L3)                 cmp  r1,0,0
                          jmp  l3
(LBL, L5)            l5:
     (ADD, 1, i, i)              lod  r1,1
                                 add  r1,i
                                 sto  r1,i
     (JMP, L1)                   cmp  r1,0,0
                                 jmp  l1
     (LBL, L3)              l3:
     (DIV, j, 3, T2)             lod  r1,j
                                 div  r1,3
                                 sto  r1,t2
     (MOV, T2,, j)               lod  r1,t2
                                 sto  r1,j
     (JMP, L5)                   cmp  r1,0,0
                                 jmp  l5
     (LBL, L4)              l4:
```

```
(c)  if (a!=b+3) a = 0; else b = b+3;

     (ADD, b, 3, T1)             lod  r1,b
                                 add  r1,3
                                 sto  r1,t1
     (TST, a, T1,, 1, L1)        lod  r1,a
                                 cmp  r1,t1,1
                                 jmp  l1
     (MOV, 0,, a)                lod  r1,0
                                 sto  r1,a
     (JMP, L2)                   cmp  r1,0,0
                                 jmp  l2
     (LBL, L1)              l1:
     (ADD, b, 3, T2)             lod  r1,b
                                 add  r1,3
                                 sto  r1,t2
     (MOV, T2,, b)               lod  r1,t2
                                 sto  r1,b
     (LBL, L2)              l2:
```

2. How many instructions correspond to each of the following atom classes on a Load/Store architecture, as in the sample problem of this section?

(a)	`ADD`	3	(b)	`DIV`	3	(c)	`MOV`	2
(d)	`TST`	3	(e)	`JMP`	2	(f)	`LBL`	0

3. Why is it important for the code generator to know how many instructions correspond to each atom class?

The code generator's first pass will compute label addresses and needs to know how much space is taken by the instructions for each atom.

4. How many machine language instructions would correspond to an ADD atom on each of the following architectures?

(a) Zero address architecture (a stack machine)

4 instructions

(b) One address architecture

3 instructions

(c) Two address architecture

3 instructions

(d) Three address architecture

2 instructions

Exercises 6.3

1. The following atom string resulted from the Java statement:

```
for (i=a; i<b+c; i++) b = b/2;
```

Translate the atoms to *instructions* as in the sample problem for this section using two methods: (1) a *single pass* method with a Fixup table for forward Jumps and (2) a *multiple pass* method. Refer to the variables `a,b,c` symbolically.

Single Pass Method:

			Fixup Table		Label Table	
	Loc	Instr	Loc	Label	Label	Value
(MOV, a,, i)	0	lod r1,a				
	1	sto r1,i				
(LBL, L1)	2				L1	2
(ADD, b, c, T1)	2	Lod r1,b				
	3	add r1,c				
	4	sto r1,t1				
(TST, i, T1,, 4, L2)	5	Lod r1,i				
	6	cmp r1,t1,4				
	7	jmp ?	7	L2		
(JMP, L3)	8	cmp r1,0,0				
	9	jmp ?	9	L3		
(LBL, L4)	10				L4	10
(ADD, i, 1, i)	10	Lod r1,i				
	11	add r1,1				
	12	sto r1,i				
(JMP, L1)	13	cmp r1,0,0				
	14	jmp 2				
(LBL, L3)	15				L3	15
(DIV, b, ='2', T3)	15	Lod r1,b				
	16	div r1,2				
	17	sto r1,t3				
(MOV, T3,, b)	18	Lod r1,t3				
	19	sto r1,b				
(JMP, L4)	20	cmp r1,0,0				
	21	jmp 10				
(LBL, L2)	22				L2	22

MultiplePass Method:
Begin first pass:

			Label Table	
	Loc	Instr	Label	Value
(MOV, a,, i)	0			
	1			
(LBL, L1)	2		L1	2
(ADD, b, c, T1)	2			
	3			
	4			
(TST, i, T1,, 4, L2)	5			
	6			
	7			
(JMP, L3)	8			
	9			
(LBL, L4)	10		L4	10
(ADD, i, 1, i)	10			
	11			
	12			
(JMP, L1)	13			
	14			
(LBL, L3)	15		L3	15
(DIV, b, ='2', T3)	15			
	16			
	17			
(MOV, T3,, b)	18			
	19			
(JMP, L4)	20			
	21			
(LBL, L2)	22		L2	22

Begin second pass:

			Label Table	
	Loc	Instr	Label	Value
(MOV, a,, i)	0	lod r1,a		
	1	sto r1,i		
(LBL, L1)	2		L1	2
(ADD, b, c, T1)	2	Lod r1,b		
	3	add r1,c		
	4	sto r1,t1		
(TST, i, T1,, 4, L2)	5	Lod r1,i		
	6	cmp r1,t1,4		
	7	jmp 22		
(JMP, L3)	8	cmp r1,0,0		
	9	jmp 15		
(LBL, L4)	10		L4	10
(ADD, i, 1, i)	10	Lod r1,i		
	11	add r1,1		
	12	sto r1,i		
(JMP, L1)	13	cmp r1,0,0		
	14	jmp 2		
(LBL, L3)	15		L3	15
(DIV, b, ='2', T3)	15	Lod r1,b		
	16	div r1,2		
	17	sto r1,t3		
(MOV, T3,, b)	18	Lod r1,t3		
	19	sto r1,b		
(JMP, L4)	20	cmp r1,0,0		
	21	jmp 10		
(LBL, L2)	22		L2	22

2. Repeat Problem 1 for the atom string resulting from the Java statement:

```
if (a==(b-33)*2) a = (b-33)*2;
    else a = x+y;
```

Single Pass Method:

			Fixup Table		Label Table	
	<u>Loc</u>	<u>Instr</u>	<u>Loc</u>	<u>Label</u>	<u>Label</u>	<u>Value</u>
`(SUB, b, ='33', T1)`	0	`lod r1,b`				
	1	`add r1,33`				
	2	`sto r1,t1`				
`(MUL, T1, ='2', T2)`	3	`lod r1,t1`				
	4	`mul r1,2`				
	5	`sto r1,t2`				
`(TST, a, T2,, 6, L1)`	6	`lod r1,a`				
	7	`cmp r1,t2,6`				
	8	`jmp ?`	8	L1		
`(SUB, b, ='33', T3)`	9	`lod r1,b`				
	10	`sub r1,33`				
	11	`sto r1,t3`				
`(MUL, T3, ='2', T4)`	12	`lod r1,t3`				
	13	`mul r1,2`				
	14	`sto r1,t4`				
`(MOV, T4,, a)`	15	`lod r1,t4`				
	16	`sto r1,a`				
`(JMP, L2)`	17	`cmp r1,0,0`				
	18	`jmp ?`	18	L2		
`(LBL, L1)`	19				L1	19
`(ADD, x, y, T5)`	19	`lod r1,x`				
	20	`add r1,y`				
	21	`sto r1,t5`				
`(MOV, T5,, a)`	22	`lod r1,t5`				
	23	`sto r1,a`				
`(LBL, L2)`	24				L2	24

3. (a) What are the advantages of a *single pass* method of code generation over a multiple pass method?

A single pass method should execute faster during compilation. The code generator can be implemented as a method invoked from the parser each time an atom is produced.

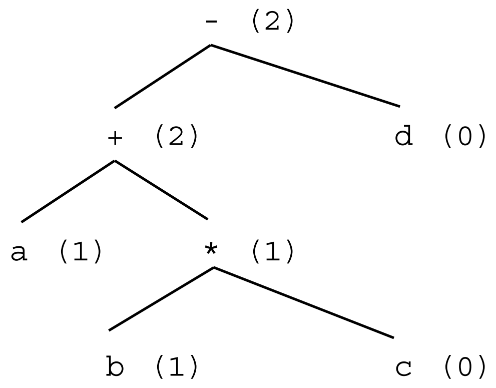
(b) What are the advantages of a *multiple pass* method of code generation over a single pass method?

A multiple pass method will work for any size source file, whereas the single pass method would require that the object program be kept in memory during compilation. The multiple pass method is easier to implement since no fixup table is needed.

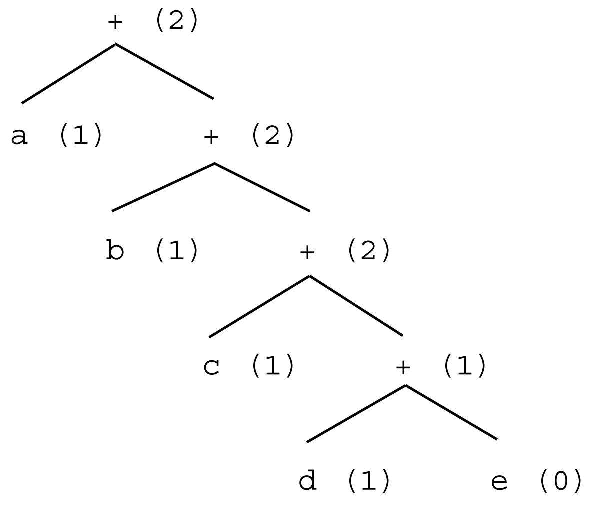
Exercises 6.4

1. Use the register allocation algorithm given in this section to construct a *weighted syntax tree* and generate code for each of the given expressions, as done in Sample Problem 6.4. Do not attempt to optimize for common subexpressions.

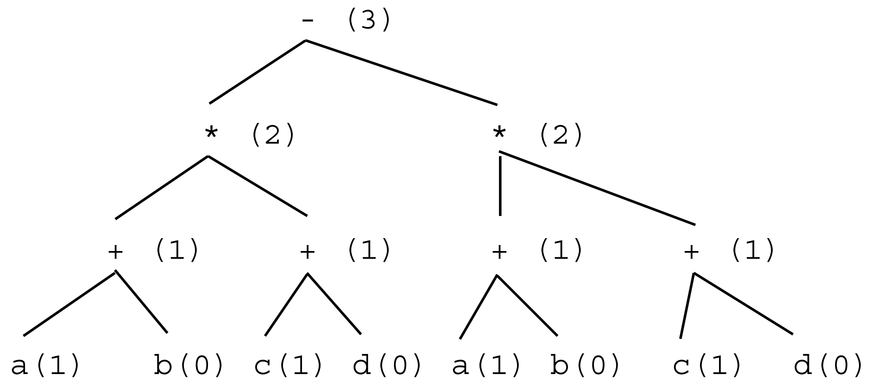
(a) `a + b * c -d`



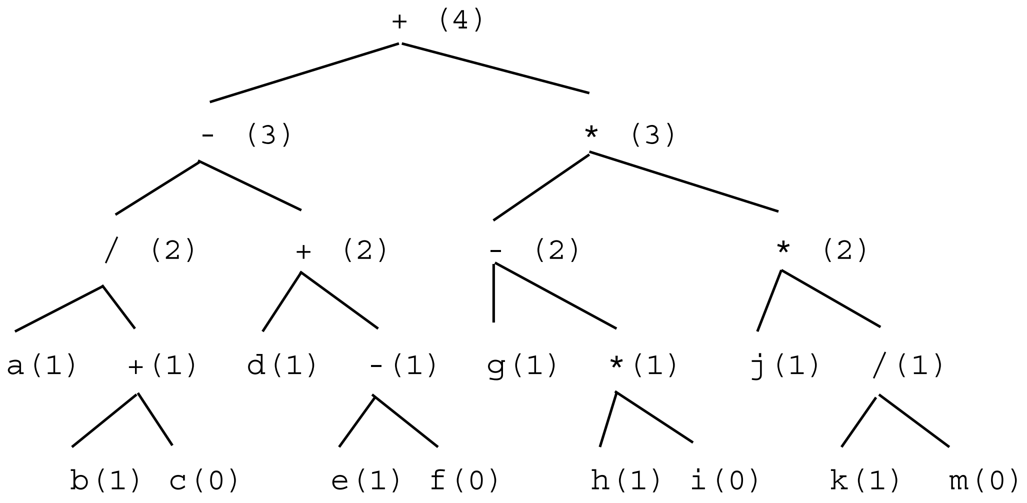
(b) `a + (b + (c + (d + e)))`



(c) `(a + b) * (c + d) - (a + b) * (c + d)`



(d) `a / (b + c) - (d + (e - f)) + (g - h * i) * (j * (k / m))`



2. Show an expression different in structure from those in Problem 1 which requires:

(a) two registers (b) three registers

As in Problem 1, assume that common subexpressions are not detected and that Loads and Stores are minimized.

(a) `(a + b) * (c + d)`

(b) `a + b * c + (d + e * f)`

3. Show how the code generated in Problem 1 (c) can be improved by making use of common subexpressions.

```
lod  r1,a
add  r1,b        // a + b
lod  r2,c
add  r2,d
mul  r1,r2
sub  r1,r1
```

Chapter 7

Exercises 7.1

1. Using a Java compiler,
(a) what would be printed as a result of running the following:

```
{
int a, b;
b = (a = 2) + (a = 3);
System.out.println ("a is " + a);
}
```

```
a is 3
```

(b) What other value might be printed as a result of compilation with a different compiler?

```
a is 2
```

2. Explain why the following two statements cannot be assumed to be equivalent:

```
a = f(x) + f(x) + f(x) ;

a = 3 * f(x) ;
```

The method f(x) may produce side effects; it may return a different value each time it is called.

3. (a) Perform the following computations, rounding to four significant digits after each operation.

```
(0.7043 + 0.4045) + -0.3330 = ?

     1.109 + (-0.3330) = 0.7760

0.7043 + (0.4045 +  -0.3330) = ?

     0.7043 + 0.0715 = 0.7758
```

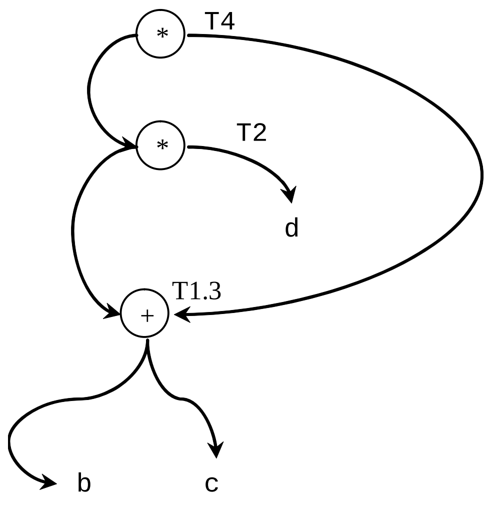
(b) What can you can conclude about the associativity of addition with computer arithmetic?

Computer arithmetic (fixed precision) is not associative.

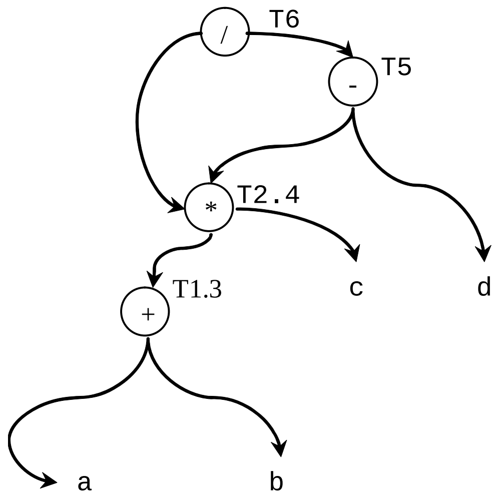
Exercises 7.2

1. Eliminate *common subexpressions* from each of the following strings of atoms, using DAGs as shown in Sample Problem 7.2 (a) (we also give the Java expressions from which the atom strings were generated):

(a)
```
(b + c) * d * (b + c)

(ADD, b, c, T1)
(MUL, T1, d, T2)
(ADD, b, c, T3)
(MUL, T2, T3, T4)

(ADD, b, c, T1.3)
(MUL, T1.3, d, T2)
(MUL, T2, T1.3, T4)
```

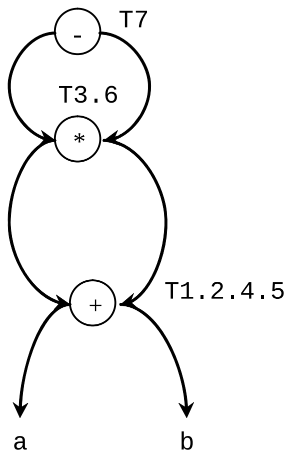


(b)
```
(a + b) * c / ((a + b) * c - d)

(ADD, a, b, T1)
(MUL, T1, c, T2)
(ADD, a, b, T3)
(MUL, T3, c, T4)
(SUB, T4, d, T5)
(DIV, T2, T5, T6)


(ADD, a, b, T1.3)
(MUL, T1.3, c, T2.4)
(SUB, T2.4, d. T5)
(DIV, T2.4, T5, T6)
```



(c)
```
(a + b) * (a + b) - (a + b) * (a + b)

(ADD, a, b, T1)
(ADD, a, b, T2)
(MUL, T1, T2, T3)
(ADD, a, b, T4)
(ADD, a, b, T5)
(MUL, T4, T5, T6)
(SUB, T3, T6, T7)
```

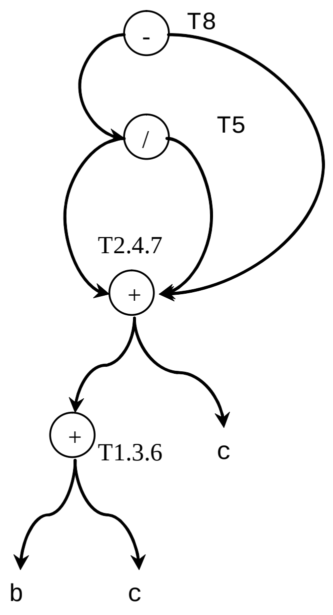
(ADD, a, b, T1.2.4.5)
(MUL, T1.2.4.5, T1.2.4.5, T3.6)
(SUB, T3.6, T3.6, T7)



(d) `((a + b) + c) / (a + b + c) - (a + b + c)`

```
(ADD, a, b, T1)
(ADD, T1, c, T2)
(ADD, a, b, T3)
(ADD, T3, c, T4)
(DIV, T2, T4, T5)
(ADD, a, b, T6)
(ADD, T6, c, T7)
(SUB, T5, T7, T8)
```

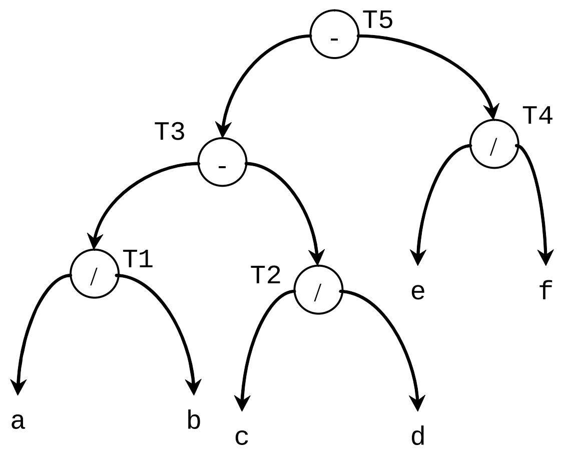
(ADD, a, b, T1.3.6)
(ADD, T1.3.6, c, T2.4.7)
(DIV, T2.4.7, T2.4.7, T5)
(SUB, T5, T2.4.7, T8)



(e) `a / b - c / d - e / f`

```
(DIV, a, b, T1)
(DIV, c, d, T2)
(SUB, T1, T2, T3)
(DIV, e, f, T4)
(SUB, T3, T4, T5)
```

(DIV, e, f, T4)
(DIV, c, d, T2)
(DIV, a, b, T1)
(SUB, T1, T2, T3)
(SUB, T3, T4, T5)



2. How many different *atom sequences* can be generated from the DAG given in your response to Problem 1 (e), above?

Eight different sequences:

T4 T1 T2 T3 T5	(as shown above)
T2 T4 T1 T3 T5	
T4 T1 T2 T3 T5	
T1 T4 T2 T3 T5	
T2 T1 T4 T3 T5	
T1 T2 T4 T3 T5	
T2 T1 T3 T4 T5	
T1 T2 T3 T4 T5	

3. In each of the following sequences of atoms, eliminate the *unreachable atoms*:

(a)
```
(ADD, a, b, T1)
(LBL, L1)
(SUB, b, a, b)
(TST, a, b,, 1, L1)
(ADD, a, b, T3)
(JMP, L1)
```

(b)
```
(ADD, a, b, T1)
(LBL, L1)
(SUB, b, a, b)
(JMP, L1)
```
~~(ADD, a, b, T3)~~
```
(LBL, L2)
```

(c)
```
(JMP, L2)
```
~~(ADD, a, b, T1)~~
~~(TST, a, b,, 3, L2)~~
~~(SUB, b, b, T3)~~
```
(LBL, L2)
(MUL, a, b, T4)
```

4. In each of the following Java methods, eliminate statements which constitute *dead code*.

(a)
```
int f (int  d)
{ int a,b~~,c~~;
      a = 3;
      b = 4;
      d = a * b + d;
      return d;
}
```

(b)
```
int f (int d)
{ int a,b,c;
      a = 3;
      b = 4;
      ~~c = a +b;~~
      d = a + b;
      ~~a = b + c * d;~~
      ~~b = a + c;~~
      return d;
}
```

5. In each of the following Java program segments, optimize the loops by moving *loop invariant code* outside the loop:

(a)
```
{     for (i=0; i<100; i++)
           {     a = x[i] + 2 * a;
                 b = x[i];
                 c = sqrt (100 * c);
           }
}

{     for (i=0; i<100; i++)
           {     a = x[i] + 2 * a;
                 c = sqrt (100 * c);
           }
      b = x[99];
}
```

(b)
```
{    for (j=0; j<50; j++)
         {    a = sqrt (x);
          n = n * 2;
          for (i=0; i<10; i++)
              {    y = x;
                   b[n] = 0;
                   b[i] = 0;
              }
          }
}

{    for (j=0; j<50; j++)
       {  n = n * 2;
          b[n] = 0;
          for (i=0; i<10; i++)
                   b[i] = 0;
       }
     a = sqrt (x);
     y = x;
}
```

6. Show how *constant folding* can be used to optimize the following Java program segments:

(a)
```
a = 2 + 3 * 8;
b = b + (a - 3);

a = 26;
b = b + 23;
```

(b)
```
int f (int  c)
{    final int a = 44;
     final int b = a - 12;
     c = a + b - 7;
     return c;
}
```

```
int f (int  c)
     {    final int a = 44;
          final int b = 32;
          c = 69;
          return c;
     }

int f (int c)       // alternative solution
     {    return 69; }
```

7. Use *reduction in strength* to optimize the following sequences of atoms. Assume that there are `(SHL, x, y, z)` and `(SHR, x, y, z)` atoms which will shift `x` left or right respectively by `y` bit positions, leaving the result in `z` (also assume that these are fixed-point operations):

(a)
```
(MUL, x, 2, T1)
(MUL, y, 2, T2)

(ADD, x, x, T1)
(ADD, y, y, T2)
```

(b)
```
(MUL, x, 8, T1)
(DIV, y, 16, T2)

(SHL, x, 3, T1)
(SHR, y, 4, T2)
```

8. Which of the following optimization techniques, when applied successfully, will always result in *improved execution time*? Which will result in *reduced program size*?

(a) Detection of common subexpressions with DAGs
(b) Elimination of unreachable code
(c) Elimination of dead code
(d) Movement of loop invariants outside of loop
(e) Constant folding
(f) Reduction in strength

Improved execution time: a, c, d, e, f
Reduced program size: a, b, c, e

Exercises 7.3

1. Optimize each of the following code segments for unnecessary *Load/Store* instructions:

(a)
```
LOD  R1,a
ADD  R1,b
STO  R1,T1
LOD  R1,T1
SUB  R1,c
STO  R1,T2
LOD  R1,T2
STO  R1,d
```

(b)
```
LOD  R1,a
LOD  R2,c
ADD  R1,b
ADD  R2,b
STO  R2,T1
ADD  R1,c
LOD  R2,T1
STO  R1,T2
STO  R2,c
```

(a)
```
LOD  R1,a
ADD  R1,b
SUB  R1,c
STO  R1,d
```

(b)
```
LOD  R1,a
LOD  R2,c
ADD  R1,b
ADD  R2,b
ADD  R1,c
STO  R1,T2
STO  R2,c
```

2. Optimize each of the following code segments for unnecessary *jump over jump* instructions:

(a)
```
         CMP  R1,a,1
         JMP  L1
         CMP  0,0,0
         JMP  L2
     L1:
         ADD  R1,R2
     L2:
```

(b)
```
         CMP  R1,a,5
         JMP  L1
         CMP  0,0,0
         JMP  L2
     L1:
         SUB  R1,a
     L2:
```

(a)
```
         CMP  R1,a,6
         JMP  L2
     L1:
         ADD  R1,R2
     L2:
```

(b)
```
         CMP  R1,a,2
         JMP  L2
     L1:
         SUB  R1,a
     L2:
```

(c)
```
     L1:
         ADD  R1,R2
         CMP  R1,R2,3
         JMP  L2
         CMP  0,0,0
         JMP  L1
     L2:
```

(c)
```
     L1:
         ADD  R1,R2
         CMP  R1,R2,4
         JMP  L1
     L2:
```

3. Use any of the *local optimization* methods of this section to optimize the following code segment:

```
     CMP  R1,R2,6               // JMP if R1 ≠ R2
     JMP  L1
     CMP  0,0,0
     JMP  L2
L1:
     LOD  R2,a
     ADD  R2,b
     STO  R2,T1
     LOD  R2,T1
     MUL  R2,c
     STO  R2,T2
     LOD  R2,T2
     STO  R2,d
     SUB  R1,0
     STO  R1,b
L2:
```

```
CMP  R1,R2,1               // JMP if R1 ≠ R2
     JMP  L2
L1:
     LOD  R2,a
     ADD  R2,b
     MUL  R2,c
     STO  R2,d
     STO  R1,b
L2:
```