CSE 307 – Midterm Exam 2 Version 1

Question 1

Write the following programs in SML.

a. Mergesort:

fun take(L) = if L = nil then nil else hd(L)::skip(tl(L)) and skip(L) = if L=nil then nil else take(tl(L));

fun merge([],M) = M | merge(L,[]) = L | merge(x::xl,y::yl) = if (x:int)

fun sort(L) = if L=[] then [] else if tl(L)=[] then L else merge(sort(take(L)),sort(skip(L)));

c. Reverse a list. (2 points)

```
fun reverse(L) =   if L=[] then []
else reverse(tl(L)) @ [hd(L)];

Alternative solution:
fun reverseHelper(L,Acc) = if L=[] then Acc
else reverseHelper(tl(L),hd(L)::Acc);
fun reverse(L) = reverseHelper(L,[]);
```

d. Find out whether a list is a palindrome. A palindrome can be read forward or backward; e.g. [r,a,c,e,c,a,r]. (4 points)

```
fun palindrome(L) =  L=reverse(L);

palindrome(["r","a","c","e","c","a","r"]);

% Alternative solution
fun last(L) =   if L=[] then ""
else if tl(L)=[] then hd(L)
else last(tl(L));

fun removeLast(L) =   if L=[] then []
else if tl(L)=[] then []
else hd(L)::removeLast(tl(L));

removeLast([2,3,4,5]);

fun palindrome2(L) =  if L=[] then true  else if tl(L)=[] then true  else
if hd(L)=last(L) then palindrome2(removeLast(tl(L)))  else false;
```

Question 2

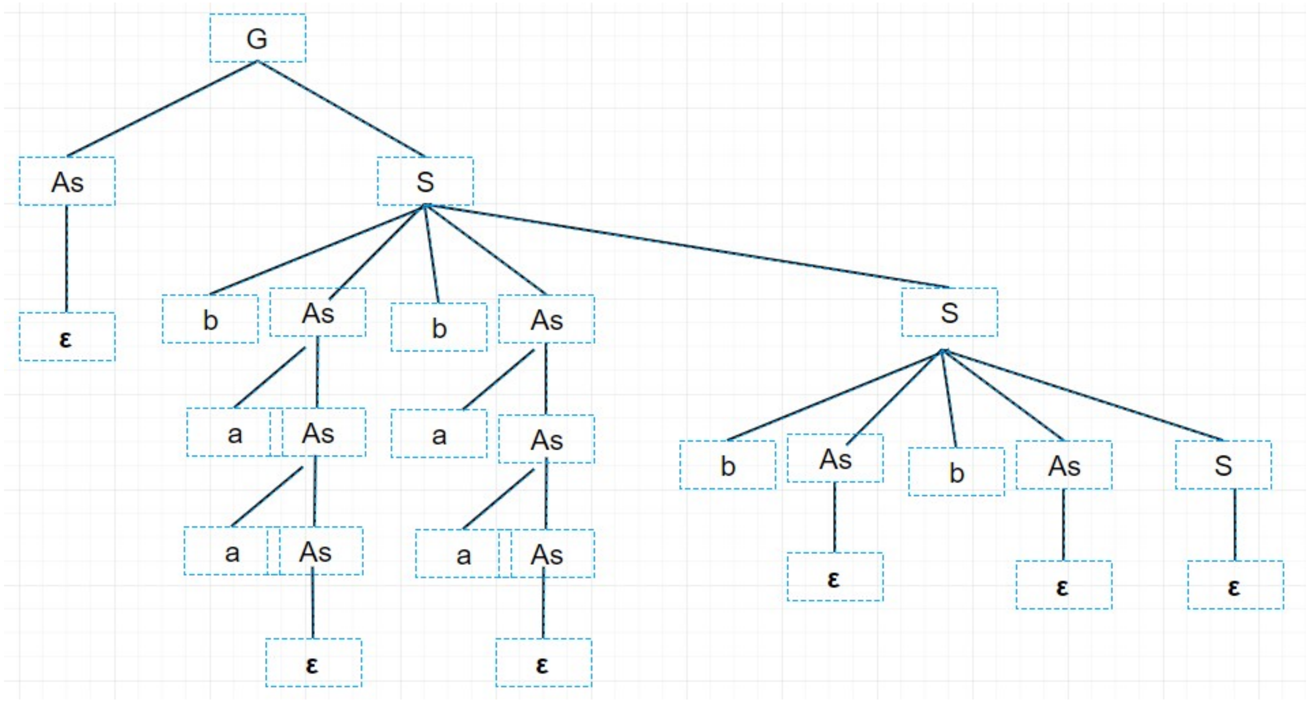
a. Describe in English the language defined by the regular expression. (2 points)
a*(b a* b a*)*

The set of all strings of as and bs containing an even number of bs.

b. Write an unambiguous context-free grammar that generates the language above. (3 points)

```
G -> As S
S     -> b As b As S
        | ε
As -> a As
        | ε
```

c. Using your grammar derive the parse tree for the following string. (3 points)
“b a a b a a b b”.



Question 3

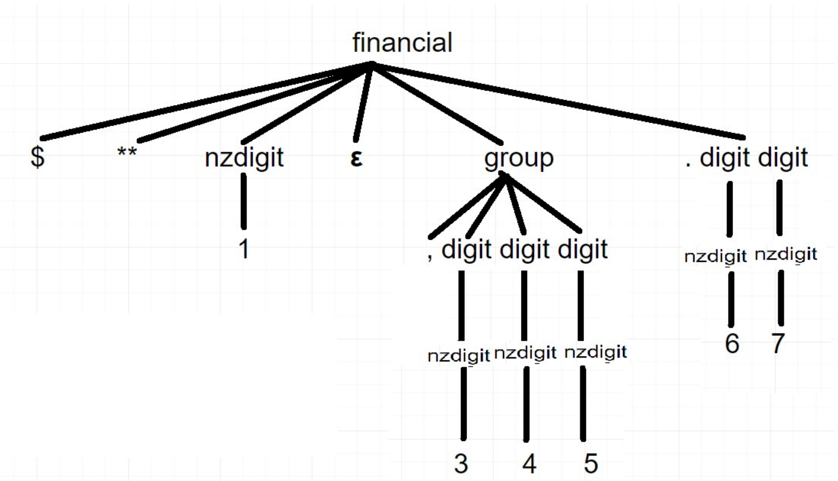
a. Write a grammar to recognize financial expressions as defined below. (4 pts)

Financial quantities in American notation have:

- a leading dollar sign ($),
- an optional string of leading asterisks (*—used on checks to discourage fraud),
- a string of decimal digits, and an optional fractional part consisting of a decimal point (.) and exactly two decimal digits.
  - the string of digits to the left of the decimal point may consist of a single zero (0). Otherwise it must not start with a zero.
  - If there are more than three digits to the left of the decimal point, groups of three (counting from the right) must be separated by commas (,).

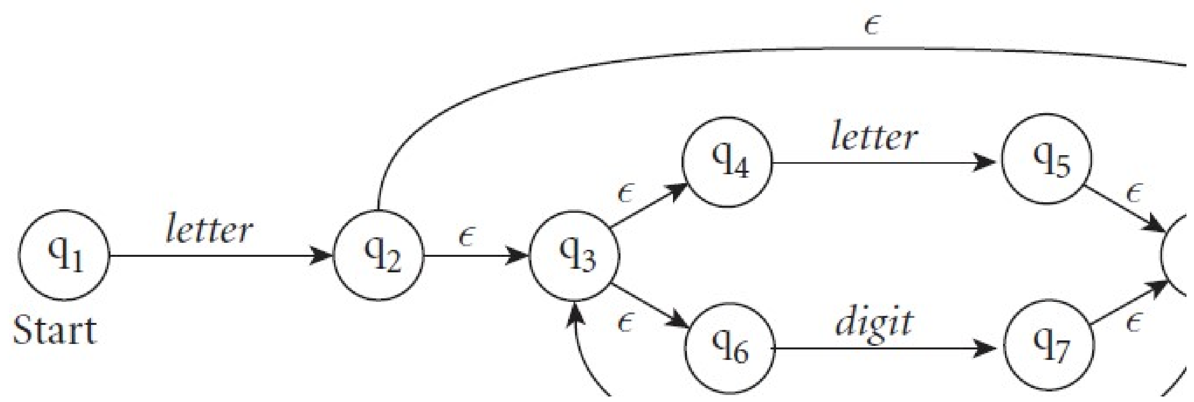
```
financial -> $
          \**
          ( 0
          |  nzdigit
                ( ε | digit | digit digit )
                group*
          )
          ( ε | . digit digit )

nzdigit-> 1| 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

digit -> 0 | nzdigit

group -> , digit digit digit
```

b. Use your grammar to derive the parse tree for the string “$**2,345.67”. (4 pts)

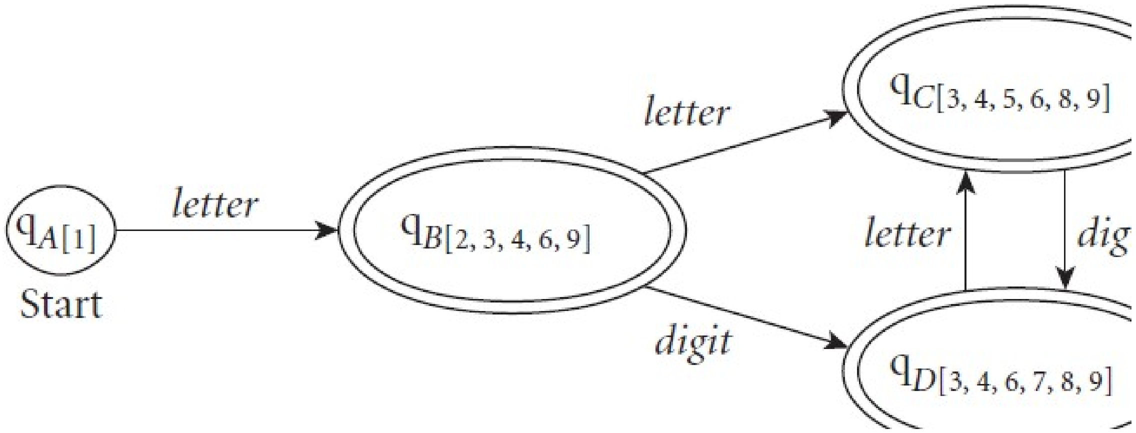


Question 4

a. Build the NFA for the regular expression *letter (letter | digit)**.



b. Convert the NFA you constructed to create an equivalent DFA.



c. Consider the following grammar. Describe in English the language the grammar generates. (4 points)

$$
\begin{array}{lcl}
G & \longrightarrow & S \ \$\$ \\
S & \longrightarrow & A \ M \\
M & \longrightarrow & S \mid \epsilon \\
A & \longrightarrow & \mathtt{a} \ E \mid \mathtt{b} \ A \\
E & \longrightarrow & \mathtt{a} \ B \mid \mathtt{b} \ A
\end{array}
$$

The grammar generates all strings of a's and b's (terminated by an end marker), in which there are more a's than b's.

Question 10

a. Consider the following pseudocode:

```
x : integer — global
procedure set x(n : integer)
          x := n

procedure print x()
          write integer(x)

procedure first()
          set x(1)
          print x()

procedure second()
          x : integer
          set x(2)
          print x()

set x(0)
first()
print x()
second()
print x()
```

What does this program print if the language uses static scoping? (2 pts)

With static scoping it prints 1 1 2 2. The

What does it print with dynamic scoping? (2 pts)

With dynamic scoping it prints 1 1 2 1.

Why? (1 pt)

The difference lies in whether set x sees the global x or the x declared in second when it is called from second.

b. Consider the following pseudocode:

```
int x = 5
int y = 6
procedure add() {
          x := x * y
          print x
          print y
}
procedure second(procedure P) {
          int x := 3
          call P()
          print x
          print y
}
procedure first() {
          int y = 4
          call second(add)
          print x
          print y
}
call first()
print x
print y
```

What does this program print if the language uses static scoping? (2 pts)

30_6_ 3_6 30_4 30_6

What does it print with dynamic scoping? (2 pts)

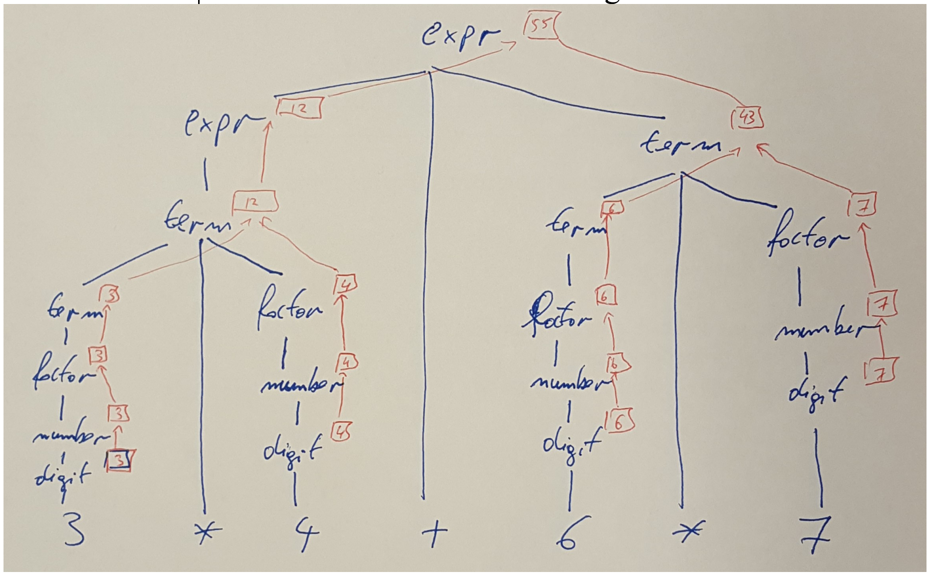
12_4 12_4 5_4 5_6

Why? (1 pt)

Question 9

Add attributed rules for the following CFG grammar and show the annotated parse tree for "3 * 4 + 6 * 7":

expr -> expr + term	>	expr1.v = expr2.v + term.v
| term	>	expr.v = term.v
term -> term * factor	>	term1.v = term2.v * factor.v
| factor	>	term.v = factor.v
factor -> (expr)	>	factor.v = expr.v
| number	>	factor.v = number.v
number -> number digit	>	number1.v = number2.v * 10 + digit.v
| digit	>	number.v = digit.v
digit -> 0	>	digit.v = 0
| 1	>	digit.v = 1
| 2	>	digit.v = 2
| 3	>	digit.v = 3
| 4	>	digit.v = 4
| 5	>	digit.v = 5
| 6	>	digit.v = 6
| 7	>	digit.v = 7
| 8	>	digit.v = 8
| 9	>	digit.v = 9



```
tokens = (
    'NAME','NUMBER',
    'PLUS','MINUS','TIMES','DIVIDE','EQUALS',
    'LPAREN','RPAREN',
    )

# Tokens

t_PLUS    = r'\+'
t_MINUS   = r'-'
t_TIMES   = r'\*'
t_DIVIDE  = r'/'
t_EQUALS  = r'='
t_LPAREN  = r'\('
t_RPAREN  = r'\)'
t_NAME    = r'[a-zA-Z_][a-zA-Z0-9_]*'

def t_NUMBER(t):
    r'\d+'
    try:
        t.value = int(t.value)
    except ValueError:
        print("Integer value too large %d", t.value)
        t.value = 0
    return t

# Ignored characters
t_ignore = " \t"

def t_newline(t):
    r'\n+'
    t.lexer.lineno += t.value.count("\n")

def t_error(t):
    print("Illegal character '%s'" % t.value[0])
    t.lexer.skip(1)

# Build the lexer
import ply.lex as lex
lexer = lex.lex()

lexer.input("1+2")

while True:
    tok = lexer.token()
    if not tok:
        break
    print(tok)

lexer.input("abc=123")

while True:
    tok = lexer.token()
    if not tok:
        break
    print(tok)

# Parsing rules
precedence = (
    ('left','PLUS','MINUS'),
    ('left','TIMES','DIVIDE'),
    ('right','UMINUS'),
    )

# dictionary of names
names = { }

def p_statement_assign(t):
    'statement : NAME EQUALS expression'
    names[t[1]] = t[3]

def p_statement_expr(t):
    'statement : expression'
    print(t[1])

def p_expression_binop(t):
    '''expression : expression PLUS expression
                  | expression MINUS expression
                  | expression TIMES expression
                  | expression DIVIDE expression'''
    if t[2] == '+'  : t[0] = t[1] + t[3]
    elif t[2] == '-': t[0] = t[1] - t[3]
    elif t[2] == '*': t[0] = t[1] * t[3]
    elif t[2] == '/': t[0] = t[1] / t[3]

def p_expression_uminus(t):
    'expression : MINUS expression %prec UMINUS'
    t[0] = -t[2]

def p_expression_group(t):
    'expression : LPAREN expression RPAREN'
    t[0] = t[2]

def p_expression_number(t):
    'expression : NUMBER'
    t[0] = t[1]

def p_expression_name(t):
    'expression : NAME'
    try:
        t[0] = names[t[1]]
    except LookupError:
        print("Undefined name '%s'" % t[1])
        t[0] = 0

def p_error(t):
    print("Syntax error at '%s'" % t.value)

import ply.yacc as yacc
yacc.yacc()

while 1:
    try:
        s = input('calc > ')   # Use raw_input on Python 2
    except EOFError:
        break
    yacc.parse(s)
```