**CSc 4330 Programming Language Concepts**
**Midterm Exam, Summer 2020**
**6 July 2020; SUBMISSION OPEN TILL 1.30 PM om July 6th (MONDAY)**

---

Handin on tinman.cs.gsu.edu under assignment **mt** with the following command:

**sudo handin4330 mt mt.txt figures.pdf**
**or**
**sudo handin4330 mt mt.pdf figures.pdf**

**Submit one or two files**: mt.txt or mt.pdf containing your answers and figures.pdf containing parse trees for problems 1, 5, and 8. Of course if you can generate one pdf containing all answers then submit just mt.pdf

---

**HONOR CODE STATEMENT (Please include this in your submitted file)**

**I truthfully declare that the work submitted for this midterm exam is solely my work and was not created in collaboration with anyone else in the class or elsewhere.**

**PUT YOUR NAME HERE**

---

GOOD LUCK!

**Problem 1 (15 Points) DERIVATION AND PARSE TREE**

Consider the following grammar that generates prefix expressions with operands x and y and binary operators +, –, and *:

```
E → + E E
E → - E E
E → * E E
E → x
E → y
```

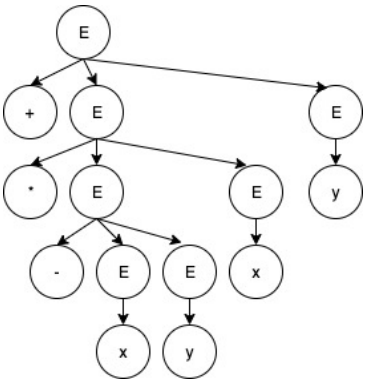
Show the rightmost derivation for +*-xyxy

Also draw the parse tree.

**SOLUTION:**

rightmost derivation

```
E => + E E
  => + E y
  => + * E E y
  => + * E x y
  => + * - E E x y
  => + * - E y x y
  => + * - x y x y
```



**Problem 2 (15 Points) GRAMMAR**

Write a context-free grammar for the language of context-free-grammars. A context-free grammar is a list of production rules. Each production rule starts with a non-terminal followed by a COLON followed by a list, possibly empty, of terminals or non-terminals. For example, the following is an example of a context-free grammar:

```
atom : NAME LPAREN args RPAREN
args : args COMMA arg
args : arg
arg  : NUMBER
arg  : STRING
arg  : NAME
```

and the following is another example of a context-free grammar:

```
json  : STRING
json  : NUMBER
json  : LBRACE kvs RBRACE
json  : LBRACKET jsons RBRACKET
kvs   : STRING COLON json
kvs   : STRING COLON json kvs
jsons : json
jsons : json jsons
```

**SOLUTION:**

```
grammar : rules

rules : rule
rules : rules rule

rule : NONTERMINAL COLON symbols

symbols :
symbols : symbols symbol

symbol : NONTERMINAL
symbol : TERMINAL
```

**Problem 3 (15 Points) ATTRIBUTE GRAMMAR**
Consider the following grammar from HW3a with the CONS rule added.

```
lisp : INT
lisp : LPAREN ADD lisp lisp RPAREN
lisp : LPAREN SUB lisp lisp RPAREN
lisp : LPAREN MUL lisp lisp RPAREN
lisp : LPAREN DIV lisp lisp RPAREN
lisp : LPAREN CAR list RPAREN

list : LPAREN CDR list RPAREN
list : LPAREN seq RPAREN
list : LPAREN CONS lisp list RPAREN

seq : lisp
seq : lisp seq
```

Augment the grammar with attributes, attribute computation functions, and predicates to REJECT expressions that would result in the possibility of applying CAR or CDR operator on an EMPTY list. Some examples of rejected expressions are:

```
(car (cdr (cdr (1 2))))
(cdr (cdr (cdr (cons 1 (cdr (20 30))))))
(cdr (cdr (cdr ((+ 30 40) (car (40 50))))))
```

**SOLUTION**:

Introduce synthesized integer-valued attributes **list.length** and **seq.length.**

**SYNTAX RULE:** lisp : LPAREN CAR list RPAREN
**PREDICATE:** list[3].length > 0

**SYNTAX RULE:** list : LPAREN CDR list RPAREN
**SEMANTIC RULE:** list[0].length = list[3].length - 1
**PREDICATE:** list[3].length > 0

**SYNTAX RULE:** list : LPAREN seq RPAREN
**SEMANTIC RULE:** list.length = seq.length

**SYNTAX RULE:** list : LPAREN CONS lisp list RPAREN
**SEMANTIC RULE:** list[0].length = list[4].length + 1

**SYNTAX RULE:** seq : lisp
**SEMANTIC RULE:** seq.length = 1

**SYNTAX RULE:** seq : lisp seq
**SEMANTIC RULE:** seq[0].length = 1 + seq[2].length

**Problem 4 (15 Points) ATTRIBUTE GRAMMAR**

Consider the following grammar describing fractional binary numbers:

```
binary : wbits DOT fbits
wbits  : BIT
wbits  : wbits BIT
fbits  : BIT
fbits  : fbits BIT
```

Assume BIT is a terminal symbol with values ‘0’ or ‘1’. Augment this grammar with attributes, attribute computation functions, and predicates to REJECT any fractional binary numbers that have leading or trailing zeros. For example, the strings “00110.1010”, “11.110”, and “001.11” all should be rejected, but “11.101”, “101.1”, and “111.11” should be accepted.

**SOLUTION**:

Introduce the following attributes:

synthesized bit-valued attribute **wbits.leftmost**
synthesized bit-valued attribute **fbits.rightmost**
intrinsic char-valued attribute **BIT.value**

**SYNTAX RULE:** binary : wbits DOT fbits
**PREDICATE:** (wbits.leftmost != ‘0’) and (fbits.rightmost != ‘0’)

**SYNTAX RULE:** wbits : BIT
**SEMANTIC RULE:** wbits.leftmost = BIT.value

**SYNTAX RULE:** wbits : wbits BIT
**SEMANTIC RULE:** wbits[0].leftmost = wbits[1].leftmost

**SYNTAX RULE:** fbits : BIT
**SEMANTIC RULE:** fbits.rightmost = BIT.value

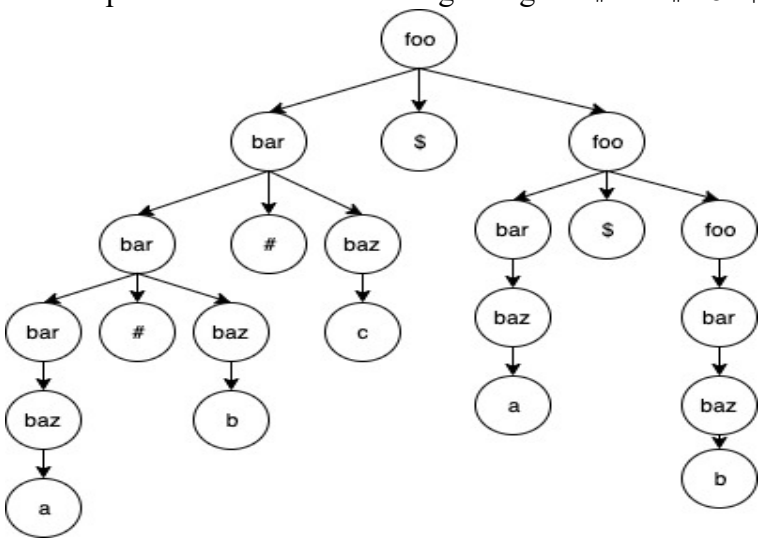
**SYNTAX RULE:** fbits : fbits BIT
**SEMANTIC RULE:** fbits[0].rightmost = BIT.value

**Problem 5 (15 Points) ASSOCIATIVITY/PRECEDENCE/RECURSION/PARSE TREE**

Consider the following grammar for a language with two infix operators represented by `#` and `$` and the start symbol `foo`:

```
foo : bar
foo : bar $ foo
bar : baz
bar : bar # baz
baz : A
baz : B
baz : C
```

What is the associativity of the # operator (left, right, or neither)? **left**
What is the associativity of the $ operator (left, right, or neither)? **right**
Which operator has higher precedence (#, $, or neither)? **#**
What type of recursion is the grammar is (left recursive, right recursive, both left and right recursive, or neither left nor right recursive)? **Both left and right recursive**
Draw a parse tree for the following string: `A # B # C $ A $ B`



**Problem 6 (15 Points) DENOTATIONAL SEMANTICS**

Consider the following grammar for fractional binary numbers:

```
fbin  : wbits DOT fbits
wbits : BIT
wbits : wbits BIT
fbits : BIT
fbits : BIT fbits
```

where `BIT` is a terminal symbol with values '0' or '1'. Write the denotational semantics function, `Mfbin`, to define the semantics of fractional binary numbers. For example `Mfbin('11.01') = 3.25` and `Mfbin('101.101') = 5.625`. You may define intermediate functions to handle the whole and fractional parts of `fbin`.

**SOLUTION:**

```
Mfbin(fbin) = Mw(wbits) + Mf(fbits)

Mw(wbits) = case wbits of
  BIT       => if BIT.value == '0' then 0 else 1
  wbits BIT => if BIT.value == '0' then
                 2*Mw(wbits)
               else
                 2*Mw(wbits)+1

Mf(fbits) = case fbits of
  BIT       => if BIT.value == '0' then 0 else 0.5
  BIT fbits => if BIT.value == '0' then
                 0.5*Mf(fbits)
               else
                 0.5 + 0.5*Mf(fbits)
```

**Problem 7 (15 Points) SHIFT REDUCE PARSER**

Consider the following grammar:

```
Rule 1: X → (X)
Rule 2: X → ()
```

and the LR Parsing table:

| | | ACTION | | | | GOTO |
|---|---|---|---|---|---|---|
| | | ( | ) | $ | | X |
| | | | | | | |
| **0** | | S2 | | | | 1 |
| **1** | | | | accept | | |
| **2** | | S2 | S5 | | | 3 |
| **3** | | | S4 | | | |
| **4** | | R1 | R1 | R1 | | |
| **5** | | R2 | R2 | R2 | | |

Give a rightmost derivation for string `(())`
Then, show the Stack, Input, and Action after each step of the LR parsing algorithm

**SOLUTION**:

**Rightmost Derivation**: `X => (X) => (())`

| **STACK** | **INPUT** | **ACTION** |
|---|---|---|
| 0 | (())$ | Shift 2 |
| 0(2 | ))$ | Shift 2 |
| 0(2(2 | ))$ | Shift 5 |
| 0(2(2)5 | )$ | Reduce 2 (use GOTO[2,X] = 3) |
| 0(2X3 | )$ | Shift 4 |
| 0(2X3)4 | $ | Reduce 1 (use GOTO[0,X] = 1) |
| 0X1 | $ | Accept |

**Problem 8 (15 Points) BOTTOM-UP PARSER PHRASES**

Consider the following grammar for balanced parentheses:

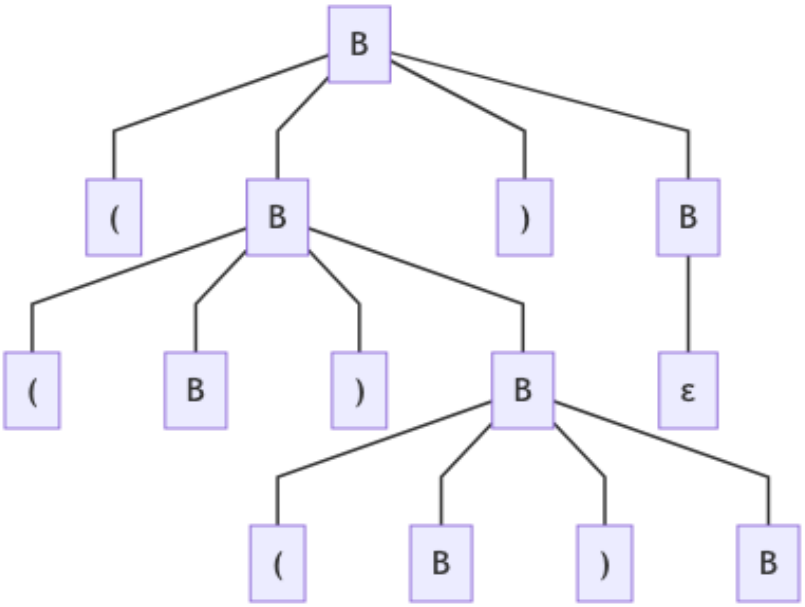
```
B : (B)B
B : ε
```

and consider the following partial right-most derivation for `(()())`

```
B => (B)B
  => (B)
  => ((B)B)
  => ((B)(B)B)
```

Show the partially constructed parse tree and list all phrases. Identify the simple phrases and the handle.

**SOLUTION**:




Phrases:

1. ((B)(B)B) Phrase
2. (B)(B)B Phrase
3. ε Simple Phrase
4. (B)B Simple Phrase; Handle