

CSc 4330 Programming Language Concepts
Exam 1, Spring 2020
2 March 2020

NAME:

QUESTION	POINTS	MAX POINTS
1		15
2		15
3		20
4		20
5		20
6		15
7		15
TOTAL		120

Please try to answer all questions and do not start panicking if you cannot finish all (-:
We have some extra points built into the exam.

GOOD LUCK!

Problem 1 (15 Points)

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

$E \rightarrow + E E$

$E \rightarrow - E E$

$E \rightarrow * E E$

$E \rightarrow x$

$E \rightarrow y$

Show the leftmost and rightmost derivations for $+*-xyxy$
Also draw the parse tree.

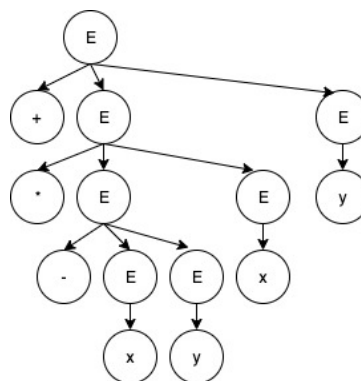
leftmost derivation

$E \Rightarrow + E E$
 $\Rightarrow + * E E E$
 $\Rightarrow + * - E E E E$
 $\Rightarrow + * - x E E E$
 $\Rightarrow + * - x y E E$
 $\Rightarrow + * - x y x E$
 $\Rightarrow + * - x y x y$

rightmost derivation

$E \Rightarrow + E E$
 $\Rightarrow + E y$
 $\Rightarrow + * E E y$
 $\Rightarrow + * E x y$
 $\Rightarrow + * - E E x y$
 $\Rightarrow + * - E y x y$
 $\Rightarrow + * - x y x y$

parse tree



Problem 2 (15 Points)

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
```

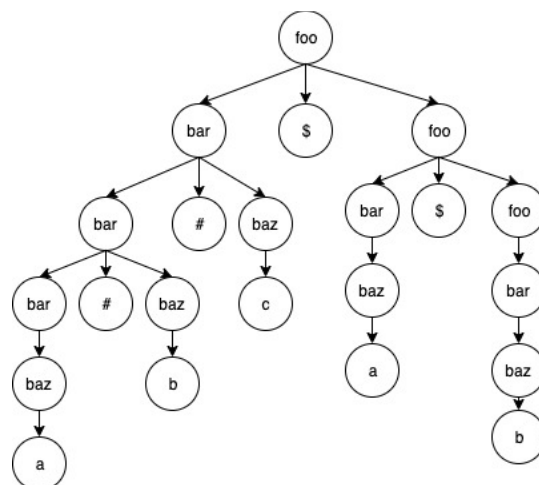
What is the associativity of the # operator (left, right, or neither)?
left

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 recursive and right recursive

Draw a parse tree for the following string: A # B # C \$ A \$ B



Problem 3 (20 Points)

Write a grammar for the language of Prolog function terms. Here are some examples of Prolog function terms:

```
220
X1
student(22, X, g(A1,A2))
[220, A, f(X,Y), [20, 30]]
h([X,Y], Z, parent(U, V))
f(g(h(A, B), X), 20, Z)
```

Formally, a Prolog function term has one of following 4 forms:

```
VARIABLE
NUMBER
f(a1,...,an)
[a1,...,an]
```

where VARIABLE is made up of letters and digits starting with upper case letter, NUMBER is a positive integer, f is the function name made up of letters starting with lower case, and a_1, \dots, a_n are Prolog function terms. For $f(a_1, \dots, a_n)$ you may assume $n > 0$ and for $[a_1, \dots, a_n]$ you may assume $n \geq 0$.

```
term : VAR | NUMBER | NAME LPAREN terms RPAREN |  
       LBRACK terms RBRACK | LBRACK RBRACK
```

```
terms : term | term COMMA terms
```

Problem 4 (20 Points)

Consider the Datalog grammar from Homework 3:

```
datalog : atom IF atomlist DOT
atom : NAME LPAREN args RPAREN
args : arg
args : args COMMA arg
arg : NUMBER
arg : NAME
atomlist : atom
atomlist : atomlist COMMA atom
```

Some examples of Datalog rules are given below:

```
ancestor(x1, y1) :- parent(x1, y1).
teaches(tno,cno) :- faculty(tno), course(cno), assigned(tno,cno).
p(x,y) :- q(2,x), r(u,45,z), s(a,b,22).
```

A Datalog rule is considered “Safe” if all variables that appear to the left of :- also appear to its right. For example in rule 2 above, the variables that appear to the left of :- are *tno* and *cno*. Both these variables appear to the right of :- as well. So, this rule is “Safe”. However, rule 3 above is not “Safe” because the variable *y* appearing to the left of :- does not appear to the right of :-.

Convert the above grammar to an attribute grammar so that it accepts only “Safe” Datalog rules.

Synthesized attributes:

- "vars" associated with atom, atomlist, args
- "type" associated with arg; two values: "num", "var"
- "value" associated with arg takes value of NUMBER/NAME token

Intrinsic attributes:

- "value" associated with NUMBER and NAME

Attribute Grammar:

Syntax Rule: datalog : atom IF atomlist DOT
Predicate: atom.vars subseteq atomlist.vars

Syntax Rule: atom : NAME LPAREN args RPAREN
Semantic Rule: atom.vars = args.vars

Syntax Rule: args : arg
Semantic Rule: args.vars = if arg.type == 'num' then {}
 else {arg.value}

Syntax Rule: args : args COMMA arg
Semantic Rule: args[0].vars = if arg.type == 'num' then
 args[1].vars
 else args[1].vars union
 {arg.value}

Syntax Rule: arg : NUMBER
Semantic Rule: arg.type = 'num'
 arg.value = NUMBER.value

Syntax Rule: arg : NAME
Semantic Rule: arg.type = 'var'
 arg.value = NAME.value

Syntax Rule: atomlist : atom
Semantic Rule: atomlist.vars = atom.vars

Syntax Rule: atomlist : atomlist COMMA atom
Semantic Rule: atomlist[0].vars =
 atomlist[1].vars union atom.vars

Problem 5 (20 Points)

Consider the LISP grammar

```
lisp : INT
      | LPAREN ADD lisp lisp RPAREN
      | LPAREN DIV lisp lisp RPAREN
      | LPAREN CAR LPAREN seq RPAREN RPAREN

seq : lisp | lisp seq
```

Write denotational semantics for LISP expressions (define M_{lisp} and M_{seq}); Clearly state the domain of objects the syntactic strings will map to. You may assume:

- a denotational mapping M_{int} that is available that maps INT token to a number,
- list of numbers objects and the following operators
 - o `createEmptyList()` that returns an empty list
 - o `x.addNumberToList(n)` that modifies list `x` by adding `n` to the front of `x`,
 - o `x.isEmpty()` returns true if `x` is empty and false otherwise, and
 - o `x.first()` that returns the first number in list `x`,
- in case there are more than one occurrence of the same non-terminal in the rule, you may refer to each with PLY-like indexes, and
- since there are no variables in lisp expressions, we do not need the state variable `s`.

Domain is the set of numbers and the set of list of numbers

M_{lisp} maps lisp-expressions to number objects

M_{seq} maps seq-expressions to list of numbers objects

```
 $M_{\text{lisp}}(\text{lisp}) = \text{case lisp of}$ 
  INT:  $M_{\text{int}}(\text{INT})$ 
  LPAREN ADD lisp lisp RPAREN:
    if (( $M_{\text{lisp}}(\text{lisp}[1]) == \text{error}$  || ( $M_{\text{lisp}}(\text{lisp}[2]) == \text{error}$ ))
      error
    else
       $M_{\text{lisp}}(\text{lisp}[1]) + M_{\text{lisp}}(\text{lisp}[2])$ 
  LPAREN DIV lisp lisp RPAREN:
    if (( $M_{\text{lisp}}(\text{lisp}[1]) == \text{error}$  || ( $M_{\text{lisp}}(\text{lisp}[2]) == \text{error}$ ))
      error
    else
      if ( $M_{\text{lisp}}(\text{lisp}[2]) == 0$ )
        error
      else
         $M_{\text{lisp}}(\text{lisp}[1]) / M_{\text{lisp}}(\text{lisp}[2])$ 
  LPAREN CAR LPAREN seq RPAREN RPAREN:
    if ( $M_{\text{seq}}(\text{seq}) == \text{error}$ )
      error
    else if ( $M_{\text{seq}}(\text{seq}).\text{isEmpty}()$ )
      error
    else
       $M_{\text{seq}}(\text{seq}).\text{first}()$ 
```

```

Mseq(seq) = case seq of
  lisp:
    if (Mlisp(lisp) == error)
      error
    else
      createEmptyList().addNumberToList(Mlisp(lisp))
  lisp seq:
    if ((Mlisp(lisp) == error) || (Mseq(seq) == error))
      error
    else
      Mseq(seq).addNumberToList(Mseq(lisp))

```


Problem 6 (15 Points)

Consider the following grammar:

Rule 0: $S \rightarrow A$

Rule 1: $A \rightarrow A + B$

Rule 2: $A \rightarrow a$

Rule 3: $B \rightarrow b$

and the LR Parsing table:

STATE		ACTION					GOTO		
		a	b	+	\$		A	B	S
0		S2			accept		1		0
1				S3	R0				
2				R2	R2				
3			S5					4	
4				R1	R1				
5				R3	R3				

Give a rightmost derivation for string $a+b+b$

Then, show the Stack, Input, and Action after each step of the LR parsing algorithm

Rightmost derivation for $a + b + b$

$S \Rightarrow A$
 $\Rightarrow A + B$
 $\Rightarrow A + b$
 $\Rightarrow A + B + b$
 $\Rightarrow A + b + b$
 $\Rightarrow a + b + b$

STACK	INPUT	ACTION
0	a+b+b\$	S2
0a2	+b+b\$	R2 (GOTO[0,A]); A --> a
0A1	+b+b\$	S3
0A1+3	b+b\$	S5
0A1+3b5	+b\$	R3 (GOTO[3,B]); B --> b
0A1+3B4	+b\$	R1 (GOTO[0,A]); A --> A + B
0A1	+b\$	S3
0A1+3	b\$	S5
0A1+3b5	\$	R3 (GOTO[3,B]); B --> b
0A1+3B4	\$	R1 (GOTO[0,A]); A --> A + B
0A1	\$	R0 (GOTO[0,S]); S --> A
0S0	\$	accept

Problem 7 (15 Points)

Consider the following grammar for balanced parentheses:

$S : (S) S$

$S : \varepsilon$

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

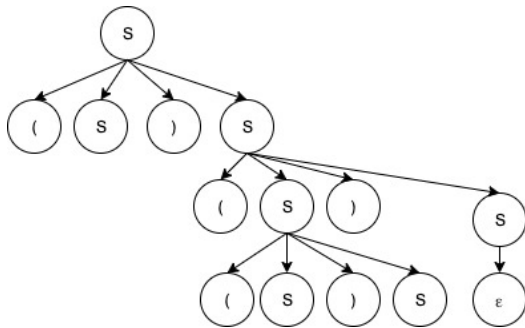
$S \Rightarrow (S) S$

$\Rightarrow (S) (S) S$

$\Rightarrow (S) (S)$

$\Rightarrow (S) ((S) S)$

Show the partially constructed parse tree below and list all phrases. Also identify the simple phrases.



Phrases:

$(S) ((S) S) \varepsilon$

$((S) S) \varepsilon$

$(S) S$: Simple; Handle

ε : Simple