COMSATS University
Park Road, Chak Shahzad, Islamabad Pakistan

Assignment 02

A2

By

Waleed Butt CU/SP18-BCS-170/ISB

Instructor

Mr. Salman Aslam

Class/Section: BCS-7C

Submission Date: 3/27/2021

Bachelor of Science in Computer Science (2018-2022)

(Q#1)

```
function lexp: integer;
var temp: integer;
    optemp: TokenType;
begin
    if token = number then
        temp := value (token);
        match(number);
else if token = ( then
        match (();
        Optemp := op;
        temp := lexpSeq (optemp);
        match ());
end if;
    return temp;
end lexp;

function op: TokenType;
var optemp: TokenType;
begin
    opTemp := token;
    case token of
        +, -, * : match(token);
        else (error);
    end case
    return optemp;
end op;

function lexpSeq (op: TokenType);
    var temp: integer;
    begin
        temp := lexp;
```

```
while token = number or token = ( do
    case op of
        + : temp := temp + lexp;
        - ; temp := temp + lexp;
        * : temp := temp * lexp;
    end case;
end while
return temp;
end lexp seq;
```

(Q#2)

Parsing Stack	Input	Action.
$ Exp	3*(4-5+6) $	Exp → Termexp'
$ expTerm	3*(4-5+6) $	Term → factor term'
$ exp' term' factor	3*(4-5+6) $	factor → number
$ exp' term' number	3*(4-5+6) $	Match
$ exp' term'	*(4-5+6) $	term' → mulop factor term'
$ exp' term' factor mulop	*(4-5+6) $	mulop → *
$ exp' term' factor *	*(4-5+6) $	Match
$ exp' term' factor	(4-5+6) $	factor (Exp)
$ exp' term') Exp(	(4-5+6) $	Match
$ exp' term') Exp	4-5+6) $	Exp → Termexp'
$ exp' term') expTerm	4-5+6) $	Term → factor term'
$ exp' term') exp' term' number	4-5+6) $	factor → number
$ exp' term') exp' term' number	4-5+6) $	Match
$ exp' term') exp'	-5+6) $	term' → ε
$ exp' term') exp'	-5+6) $	exp' → addop Termexp'
$ exp' term') exp' Termaddop	-5+6) $	addop → -
$ exp' term') exp' Term-	-5+6) $	Match
$ exp' term') exp' term'	5+6) $	Term → factor term'
$ exp' term') exp' term' factor	5+6) $	factor → number
$ exp' term') exp' term' number	5+6) $	Match
$ exp' term') exp' term'	+6) $	term' → ε

Parsing Stack	Input	Action
$exp' term') exp'	+6) $	exp' → addop Term exp'
$exp' term') exp' Term addop	+6) $	addop → +
$exp' term') exp' Term +	+6) $	Match
$exp' term') exp' Term	6) $	Term → factor term'
$exp' term') exp' term' factor	6) $	factor → number
$exp' term') exp' term' number	6) $	Match
$exp' term') exp' term'	) $	term' → ε
$exp' term') exp'	) $	exp' → ε
$exp' term')	) $	Match
$exp' term'	$	term' → ε
$exp'	$	exp' → ε
$	$	Accept

(Q#3)

(a)

lexp → lexp lexp-seq'

lexp-seq' → lexp lexp-seq' | ε

(b)

Non-Terminals	First	Follow
lexp	{ number, identifier, (}	{ $,), number, identifier, (}
atom	{ number, identifier }	{ $,), number, id, (}
list	{ (}	{ $,), number, id, (}
lexp-seq	{ number, identifier, (}	{) }
lexp-seq'	{ number, identifier, ε }	{) }

(c)

gt is LL(1) because

First(atom) ∩ First(list) = ∅, First(lexp lexp seq') ∩ first(ε) = ∅ = First(lexp) ∩ First(ε). Also, First(lexp-seq') ∩ Follow(lexp-seq') = ∅

(d)

The LL(1) parsing table is:-

M(N,T)	number	identifier	(	)	$
lexp	lexp → atom	lexp → atom	lexp → list		
atom	atom → number	atom → identifier			
list		list → (lexp-seq)			
lexp-seq	lexp seq → lexp lexp seq'	lexp-seq → lexp lexp-seq	lexp-seq → lexp lexp-seq		
lexp-seq'	//	//	//	lexp-seq' → ε	

(e)

Input (a(B(2))(c)) $

Parsing Stack	Input	Action
$ lexp	(a(B(2))(c))$	lexp → list
$ list	(a(B(2))(c))$	list → (lexp-seq)
$) lexp-seq (	(a(B(2))(c))$	Match
$) lexp-seq	a(B(2))(c))$	lexp-seq → lexp lexp-seq'
$) lexp-seq' lexp	a(B(2))(c))$	lexp → atom
$) lexp-seq' atom	a(B(2))(c))$	atom → id
$) lexp-seq' id	a(B(2))(c))$	Match
$) lexp-seq'	(B(2))(c))$	lexp-seq' → lexp lexp-seq'
$) lexp-seq' lexp	(B(2))(c))$	lexp → list
$) lexp-seq' list	(B(2))(c))$	list → (lexp-seq)
$) lexp-seq') lexp-seq (	(B(2))(c))$	Match

Parsing Stack	Input	Action
$) lexpseq') lexp-seq	B(2))(C))$	lexp-seq → lexplexp-seq'
$) lexpseq') lexp-seq' lexp	B(2))(C))$	lexp → atom
$) lexpseq') lexp-seq' atom	B(2))(C))$	atom → id
$) lexpseq') lexp-seq' - id	B(2))(C))$	Match
$) lexpseq') lexp-seq'	(2))(C))$	lexp-seq' → lexplexp-seq'
$) lexpseq') lexp-seq' lexp	(2))(C))$	lexp → list
$) lexpseq') lexp-seq' list	(2))(C))$	list → (lexp-seq')
$) lexpseq') lexp-seq') lexp-seq	(2))(C))$	Match)
$) lexpseq') lexp-seq') lexp-seq	2))(C))$	lexp-seq → lexplexp-seq'
$) lexpseq') lexp-seq') lexp-seq' lexp	2))(C))$	lexp → atom
$) lexpseq') lexp-seq') lexp-seq' atom	2))(C))$	atom → number
$) lexpseq') lexp-seq') lexp-seq' number	2))(C))$	Match
$) lexpseq') lexp-seq') lexp-seq	))(C))$	lexp-seq' → ε
$) lexpseq') lexp-seq')	))(C))$	Match
$) lexpseq') lexp-seq'	)(C))$	lexp-seq' → ε
$) lexpseq') lexp-seq	)(C))$	Match
$) lexpseq')	(C))$	lexp-seq → lexp-seq'
$) lexpseq'	(C))$	lexp → list
$) lexpseq' lexp	(C))$	list → (lexp-seq')
$) lexpseq') lexp-seq' (	(C))$	Match
$) lexpseq') lexp-seq'	C))$	lexp-seq → lexplexpseq'
$) lexpseq') lexp-seq' lexp	C))$	lexp → atom
$) lexpseq') lexp-seq' atom	C))$	atom → id
$) lexpseq') lexp-seq' identifier	C))$	Match
$) lexpseq') lexp-seq'	C))$	lexp-seq' → lexplexp
$) lexpseq') lexp-seq' lexp	C))$	lexp → atom
$) lexpseq') lexp-seq' atom	C))$	atom → id
$) lexpseq') lexp-seq' id	C))$	Match
$) lexpseq') lexp-seq	))$	lexp-seq' → ε
$) lexpseq')	))$	Match
$) lexp-seq	)$	lexp-seq → ε
$)	)$	Match
$	$	Accept

(Q#4)

(a)

A grammar is said to be ambiguous if there exists more than one derivation tree for a given string. However, an LL(1) grammar constructs a unique leftmost derivation for any string and thus it can not be ambiguous.

(b)

Since an ambiguous grammar constructs more than one derivation tree, therefore it can not be LL(1) because for it to be LL(1) there must only be one unique leftmost derivation.

(c)

If a grammar is unambiguous it may still fail to be LL(1) since it is not necessary that every unambiguous grammar can be passed by LL(1) parser.

(d)

A left-recursive grammar is not LL(1) since it would lead to infinite recursivity. Infact a left-recursive grammar is always ambignos and an ambiguous grammar can never be LL(1).

(Q#5)

(a)

Suppose we have a grammar G in which S is the start symbol, A is a non-terminal and t is a terminal such that $t \in L(G)$. If $S \Rightarrow At \mid t$

The A is useless since there is no derivation of a leading to any token

(b)

Useless grammers are unlikely to appear in any programing language grammar since they contribute nothing to the language. They have no purpose so they are just a design error.

(c)

Comider the grammar

$$S \rightarrow (A) \mid \varepsilon$$
$$A \rightarrow AC \mid S$$

In this grammar A is useless since
First (S) = {(, ε} and Follow (S) = {$,), (} since Follow of S contains follow of A & thus;

First (S) ∩ Follow (S) ≠ ∅

(Q#6)

Given: (2 +- 3) * 4 -+5

Parsing Stack	Input	Action
$ Exp	(2+-3) * 4-+5	Exp → Term exp'
$ exp' Term	(2+-3) * 4-+5	Term → factor term'
$ exp' term' factor	(2+-3) * 4-+5	factor → (Exp)
$ exp' term') Exp(	(2+-3) * 4-+5	Match
$ exp' term') Exp	2+-3) * 4-+5	Exp → Term exp'
$ exp' term') exp' Term	2+-3) * 4-+5	Term → factor term'
$ exp' term') exp' term' factor	2+-3) * 4-+5	factor → number
$ exp' term') exp' term' number	2+-3) * 4-+5	Match
$ exp' term') exp' term'	+-3) * 4-+5	term' → ε
$ exp' term') exp'	+-3) * 4-+5	exp' → Addop Term exp'
$ exp' term') exp' term Addop	+-3) * 4-+5	Addop → +
$ exp' term') exp' Term +	+-3) * 4-+5	Match
$ exp' term') exp' Term	-3) * 4-+5	Pop (error)
$ exp' term') exp'	-3) * 4-+5	exp → Addop Term exp'

Passing Stack	Input	Action
$ exp' term') exp' Term Addop	+ - 3) * 4 - + 5 $	Addop → +
$ exp' term') exp' Term +	+ - 3) * 4 - + 5 $	Match
$ exp' term') exp' Term	- 3) * 4 - + 5 $	Pop (error)
$ exp' term') exp'	- 3) * 4 - + 5 $	exp → Addop Term exp'
$ exp' term') exp' Term Addop	- 3) * 4 - + 5 $	Addop → -
$ exp' term') exp' Term -	- 3) * 4 - + 5 $	Match
$ exp' term') exp' Term	3) * 4 - + 5 $	Term → factor term'
$ exp' term') exp' term' factor	3) * 4 - + 5 $	factor → number
$ exp' term') exp' term' number	3) * 4 - + 5 $	Match
$ exp' term') exp' term'	) * 4 - + 5 $	term' → ε
$ exp' term') exp'	) * 4 - + 5 $	exp' → ε
$ exp' term')	) * 4 - + 5 $	Match
$ exp' term'	* 4 - + 5 $	term' → Mulop factor Term'
$ exp' term' factor Mulop	* 4 - + 5 $	Mulop → *
$ exp' term' factor *	* 4 - + 5 $	Match
$ exp' term' factor	4 - + 5 $	factor → number
$ exp' term' number	4 - + 5 $	Match
$ exp' term'	- + 5 $	term' → ε
$ exp'	- + 5 $	exp' → Addop Term exp'
$ exp' Term Addop	- + 5 $	Addop → -
$ exp' Term -	- + 5 $	Match
$ exp' Term	+ 5 $	Pop (Error)
$ exp'	+ 5 $	exp' → Addop Term exp'
$ exp' Term Addop	+ 5 $	Addop → +
$ exp' Term +	+ 5 $	Match
$ exp' Term	5 $	Term → factor term'
$ exp' term' factor	5 $	factor →
$ exp' term' number	5 $	Match
$ exp' term'	$	term' → ε
$ exp'	$	exp' → ε
$	$	Accept

(Q#7)

Non-Terminal	First Set	Follow Set
E	{ (, id }	{ $,) }
E'	{ +, ε }	{ $,) }
T	{ (, id }	{ $,), + }
T'	{ *, ε }	{ $,), + }
F	{ (, id }	{ $,), +, * }

(Q#8)

(a)

The only rule that changes after left-factoring is:-

var-list → identifier varlist'

var-list' → , var-list | ε

(b)

Non-Terminal	First Set	Follow Set
declaration	{int, float}	{ $ }
type	{int, float}	{ identifier }
var-list	{ identifier }	{ $ }
var-list'	{ , , ε }	{ $ }

(c)

The resulting grammar is LL(1) since for production var-list', first of (var-list) ∩ fis(ε) = ϕ and ∵ var-list' contain ε first(var-list) ∩ follow(var-list) = ϕ

(d)

M[N,T]	int	float	id	,	$
declaration	declaration → type var-list	declaration → type var-list			
type	type → int	type → float			
var-list			var-list → identifier var-list		
var-list'				var-list' → , varlist	var-list' → ε

(e)

Given Input int x, y, z

Parsing Stack	Input	Action
$ declaration	int x, y, z $	declaration → type var-list
$ var-list type	int x, y, z $	type → int
$ var-list int	int x, y, z $	Match
$ var-list	x, y, z $	var-list → identifier var-list
$ var-list' id	x, y, z, $	Match
$ var-list'	, y, z $	var-list' → , var-list
$ var-list ,	, y, z $	Match
$ var-list	y, z $	var-list → identifier var-list'

Parsing Track	Input	Action
$ var-list identifier	y, z $	Match
$ var-list'	, z $	var-list' → , var-list
$ var-list ,	, z $	Match
$ var-list	z $	var-list → identifier var-list'
$ var-list' identifier	z $	Match
$ var-list'	$	var-list' → ε
$	$	Accept

(Q#9)

After eliminating EBNF

A → B

B → a | (C) | [B] | ε

C → BD

D → +BD | ε

Non-Terminals	First set	Follow set
A	{a, (, [, ε}	{$}
B	{a, (, [, ε}	{.,],), +}
C	{a, (, [, ε}	{)}
D	{+, ε}	{)}

Parsing Table

M[N,T]	a	c	(	)	[	]	+	.	$
A	A→B.		A→B.	B→ε	A→B.				A→ε
B	B→a		B→(C)	B→ε	B→[B]	B→ε	B→ε	B→ε	
C	C→BD		C→BD	C→ε	C→BD				
D				D→ε			D→+BD		

(Q#10)

(i)

By checking the two rules of LL(1) grammer we see:

⇒ First(Z) ∩ First(A) = ∅ since First(A) = x & First(Z) = z

Also,

⇒ First(C) contains ε, ∴ we need to check insertion of its first & follow set.

First(C) = {y, ε} and Follow(C) = {x, z}

∵ First(C) ∩ Follow(C) = ∅

∴ The given grammar is LL(1)

(ii)

Non-Terminals	First	Follow
A	{x}	{$, V}
B	{z, x}	{$, X, z}
C	{y, ε}	{$, X, z}

(iii)

M[N, T]	x	y	z	$
A	A→XCB			
B	B→Ax		B→z	
C	C→ε	C→yBz	C→ε	C→ε

a) Table(A, x) contains XCB
b) Table(A, y) contains error
c) Table(A, z) contains error
d) Table(B, x) contains Ax
e) Table(B, y) contains error
f) Table(B, z) contains z
g) Table(C, x) contains ε
h) Table(C, y) contains yBz
i) Table(C, z) contains ε

(Q#11)

(a)

The left factored grammar is :-

$S \rightarrow iEtSS' \mid a$

$S' \rightarrow eS \mid \varepsilon$

$E \rightarrow b$

(b)

- After remove left-recursion, the grammar is :-

$S \rightarrow TS'$

$S' \rightarrow +TS' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (S) \mid id$

(Q#12)

(a)

Non-Terminals	First	Follow
P	{i, b}	{$}
S	{id}	{e}
R	{id, ε}	{e}
A	{id}	{id, e}
E	{(, id}	{id, e,)}
T	{+, ε}	{id, e,)}
F	{(, id}	{id, e,), +}

(b)

M[N,T]	b	e	id	+	(	)	$
P	P→bSe						
S			S→AR				
R		R→SE	R→AR				
A			A→id=E;				
E			E→FT				
T		T→ε	T→ε		T→ε	T→+FT	
F			F→id		F→(E)		

(c)

Since First(R) contains ε and First(R) $\cap$ Follow(R) = ϕ
and First(T) contains ε and First(T) $\cap$ Follow(T) = ϕ
Therefore the given grammar is LL(1)
Also, from LL(1) parsing table we can see that there is only one production per cell so it is LL(1)

(d)

b id = id + 1 ; e $

Parsing Stack	Input	Action
$P	bid = id +1 ; e $	P → BSE
$eSb	bid = id +1 ; e $	Match
$eS	id = id +1 ; e $	S → AR
$eRA	id = id +1 ; e $	A → id = E ;
$eR ; E = id	id = id +1 ; e $	Match
$eR ; E =	= id +1 ; e $	Match
$eR ; E	id +1 ; e $	E → FT
$eR ; TF	id +1 ; e $	F → id
$eR ; Tid	id +1 ; e $	Match
$eR ; T	+1 ; e $	T → +FT
$eR ; TF+	+1 ; e $	Match
$eR ; TF	1 ; e $	F → id
$eR ; Tid	1 ; e $	Match
$eR ; T	; e $	T → ε
$eR ;	; e $	Match
$eR	e $	R → ε
$e	e $	Match
$	$	Accept

(Q#13)

(i)

Given the grammar $A \to (A)A \mid \varepsilon$ where,

First(A) = { (, ε } and Follow(A) = { $,) }

By using two rule of LL(1):-

⇒ First(A) ∩ First(ε) = ϕ and

⇒ First(A) contains ε, so First(A) ∩ Follow(A) = ϕ

Thus grammar is LL(1)

(ii)

Given the grammar $A \to aAa \mid \varepsilon$

where

First(A) = { a, ε } and Follow(A) = { $, a }

⇒ By first rule, First(A) ∩ First(ε) = ϕ

However

⇒ By second value ε is in First(A) but,

First(A) ∩ Follow(A) = a ≠ ϕ

Thus the grammar is Not LL(1)

(Q#14)

(a)

$S \to a \mid ab \mid abc \mid abcd \mid$

Modified grammar

$S \to abC \mid a$

$C \to cD \mid c$

$D \to d \mid \varepsilon$

(b)

Grammar : $S \rightarrow SS+ \mid SS* \mid a$

Input : aa+a*

Parsing Stack	Input	Action
$S	aa+a*$	S → SS*
$S*SS	aa+a*$	S → SS+
$S*S+SS	aa+a*$	S → a
$S*S+Sa	aa+a*$	Match
$S*S+S	a+a*$	S → a
$S*S+a	a+a*$	Match
$S*S+	+a*$	Match
$S*S	a*$	S → a
$S*a	a*$	Match
$S*	*$	Match
$S	$	Accept

(Q#15)

(a)

Grammar After left factoring.

lexp-seq → lexp lexp-seq'

lexp-seq' → , lexp-seq | ε

(b)

Non-Terminals	First	Follow
lexp	{ number, id, (, }	{ $,), , }
atom	{ number, id }	{ $,), , }
list	{ (}	{ $,), , }
lexp-seq	{ number, id, (}	{) }
lexp-seq'	{ , , ε }	{) }

(c)

Since Lexp contains two productions so, First(atom) ∩ First(list) = ∅. Similarly First(lexp-seq) ∩ Follow(lexp-seq') = ∅
Thus the grammar is in LL(1)

(d)

M[N,T]	number	identifier	(	)	,	$
lexp	lexp → atom	lexp → atom	lexp → list			
atom	atom → number	atom → identifier				
list			list → (lexp-seq)			
lexp-seq	lexp-seq → lexp lexp-seq	lexp-seq → lexp lexp-seq	lexp-seq → lexp lexp-seq'			
lexp-seq'				lexp-seq' → ε	lexp-seq' → , lexp-seq	

(e)

Parsing Stack	Input	Action
$ lexp	(a,(b,(2)),(c)) $	lexp → list
$ list	(a,(b,(2)),(c)) $	list → (lexp-seq)
$) lexp-seq (	(a,(b,(2)),(c)) $	Match
$) lexp-seq	a,(b,(2)),(c)) $	lexp-seq → lexp lexp-seq'
$) lexp-seq' lexp	a,(b,(2)),(c)) $	lexp → atom
$) lexp-seq' atom	a,(b,(2)),(c)) $	atom → id
$) lexp-seq' identifier	a,(b,(2)),(c)) $	Match
$) lexp-seq'	,(b,(2)),(c)) $	lexp-seq' → , lexp-seq
$) lexp-seq ,	,(b,(2)),(c)) $	Match
$) lexp-seq	(b,(2)),(c)) $	lexp-seq → lexp lexp-seq'

Parsing Stack	Input	Action
$ | lexp-seq lexp	(b) , (2)) , (( )) $	lexp-list
$ | lexp-seq list	(b) , (2)) , (( )) $	list → (lexp-seq)
$) | exp-seq) lexp-seq (	(b) , (2)) , (( )) $	Match
$) | lexp-seq) lexp-seq ,	b , (2)) , (( )) $	lexp-seq → lexp lexp-seq'
$) | lexp-seq) lexp-seq' lexp	b , (2)) , (( )) $	lexp → atom
$) | lexp-seq) lexp-seq atom	b , (2)) , (( )) $	atom → id
$) | lexp-seq) lexp-seq id	b , (2)) , (( )) $	Match
$) | lexp-seq) lexp-seq	, (2)) , (( )) $	lexp-seq' → , lexp-seq
$) | lexp-seq) lexp-seq ,	, (2)) , (( )) $	Match
$) | lexp-seq) lexp-seq	(2)) , (( )) $	lexp-seq → lexp lexp-seq'
$) | lexp-seq) lexp-seq lexp	(2)) , (( )) $	lexp → list
$) | lexp-seq) lexp-seq list	(2)) , (( )) $	list → (lexp-seq)
$) | lexp-seq) lexp-seq) lexp-seq (	(2)) , (( )) $	Match
$) | lexp-seq) lexp-seq) lexp-seq	2)) , (( )) $	lexp-seq → lexp lexp-seq'
$) | lexp-seq) lexp-seq) lexp-seq lexp	2)) , (( )) $	lexp → atom
$) | lexp-seq) lexp-seq) lexp-seq atom	2)) , (( )) $	atom → id
$) | lexp-seq) lexp-seq) lexp-seq id	2)) , (( )) $	Match
$) | lexp-seq) lexp-seq) lexp-seq	)) , (( )) $	lexp-seq' → ε
$) | lexp-seq) lexp-seq)	)) , (( )) $	Match
$) | lexp-seq) lexp-seq'	) , (( )) $	lexp-seq' → ε
$) | lexp-seq)	) , (( )) $	Match
$) | lexp-seq	, (( )) $	lexp → seq' → , lexp-seq
$) | lexp-seq ,	, (( )) $	lexp → list
$) | lexp-seq	(( )) $	list → (lexp-seq)
$) | lexp-seq lexp	(( )) $	lexp → seq' → lexp-seq
$) | lexp-seq list	(( )) $	Match
$) | lexp-seq) lexp-seq (	(( )) $	Match
$) | lexp-seq) lexp-seq	( )) $	lexp → seq' → lexp lexp-seq'
$) | lexp-seq) lexp-seq lexp	( )) $	lexp → atom
$) | lexp-seq) lexp-seq atom	( )) $	atom → id
$) | lexp-seq) lexp-seq id	( )) $	Match