National University of Computer and Emerging Sciences

Lahore Campus

Compiler Construction (CS4031)

Final Exam

Total Time (Hrs): **3**

Total Marks: **64**

Total Questions: **7**

Date: May 22nd 2024

Course Instructor(s)

Dr. Faisal Aslam

_____ _____ _____

Roll No Section Student Signature

Q1	Q2	Q3	Q4	Q5	Q6	Q7	Total
10	10	7	10	7	10	10	64

	id	+	*	(	)	$	E	T	F
I0	s5						1	2	3
I1		s6				Accept			
I2		r2	s7		r2	r2			
I3		r4	r4		r4	r4			
I4	s5			s4			8	2	3
I5		r6	r6		r6	r6			
I6	s5			S4				9	3
I7	s5			S4					10
I8		s6			S11				
I9		r1	s7		r1	r1			
I10		r3	r3		r3	r3			
I11		r5	r5		r5	r5			

Parsing table for Q5

National University of Computer and Emerging Sciences

1. Write a Flex program to evaluate postfix expressions. The algorithm is as follows: [10 Marks]

   (a) **Read input:**

   - If the input is an integer, store it in an integer array. The integer array has a fixed size of 10, and it is assumed that it will never overflow.
   - If the input is an operator, remove the last two integers stored in the integer array, apply the operation, and put the result back into the array. We only support the operators: `+`, `-`, `/`, and `*`.

   (b) **End of input:**

   - At the end of reading the input, print the contents of the array. The array should contain only one integer, which is the result of the postfix expression.

   Example

   For the input `1 5 + 3 /`:

   (a) **Read 1:** Store it in the array at index 0.

   (b) **Read 5:** Store it in the array at index 1.

   (c) **Read +:** Remove the last two integers (1 and 5) from the array. Apply the `+` operator to these integers: $1 + 5 = 6$. Put the result (6) back into the array at index 0.

   (d) **Read 3:** Store it in the array at index 1.

   (e) **Read /:** Remove the last two integers (6 and 3) from the array. Apply the `/` operator to these integers: $6/3 = 2$. Put the result (2) back into the array at index 0.

   At the end of the input, the array contains one integer: `2`. That is printed.

```
    %{
#include <stdio.h>
#include <stdlib.h>

#define MAX_STACK_SIZE 10

int stack[MAX_STACK_SIZE];
int top = -1;

void push(int value) {
    if (top < MAX_STACK_SIZE - 1) {
        stack[++top] = value;
    } else {
        printf("Stack overflow\n");
```

```
        exit(1);
    }
}

int pop() {
    if (top >= 0) {
        return stack[top--];
    } else {
        printf("Stack underflow\n");
        exit(1);
    }
}

%}

%%

[0-9]+      {
                int value = atoi(yytext);
                push(value);
            }

[+\-*/]     {
                int b = pop();
                int a = pop();
                int result;

                switch(yytext[0]) {
                    case '+': result = a + b; break;
                    case '-': result = a - b; break;
                    case '*': result = a * b; break;
                    case '/': result = a / b; break;
                    default: printf("Unknown operator\n");
    exit(1);
                }

                push(result);
            }

\n          {
                if (top == 0) {
                    printf("Result: %d\n", stack[top]);
                } else {
                    printf("Invalid expression\n");
                }
                exit(0);
            }

[ \t]+      { /* ignore whitespace */ }

.           { printf("Unexpected character: %s\n", yytext);
    exit(1); }

%%

int main() {
    yylex();
```

```
    return 0;
}
```

2. Assume that Flex program already exist and will be providing you tokens. Write a Bison program that parses and evaluates boolean expressions. The grammar should support the boolean literals `true` and `false`, as well as the logical operators `and`, `or`, and `not` with the correct operator precedence. [10 Marks]

   Requirements

   - The input consists of the literals `true` and `false`, and the operators `and`, `or`, and `not`.
   - The `not` operator has the highest precedence, followed by `and`, and then `or`.
   - Handle parentheses to override precedence.
   - Print the result of the evaluated boolean expression as `true` or `false`.

   Example Inputs and Outputs

   - Input: `true and false or not false`
     Output: `true`
   - Input: `(true and false) or (not false)`
     Output: `true`
   - Input: `not (true or false) and true`
     Output: `false`

```
        %{
#include <stdio.h>
#include <stdlib.h>

void yyerror(const char *s);
int yylex(void);

typedef int bool;
#define true 1
#define false 0
%}

%union {
    int boolean;
}

%token <boolean> TRUE FALSE
%token AND OR NOT
%token LPAREN RPAREN
%start input

%left OR
%left AND
```

```
%right NOT
%left '(' ')'

%type <boolean> expr

%%

input:
    expr '\n' {
        if ($1 == true) {
            printf("true\n");
        } else {
            printf("false\n");
        }
    }
;

expr:
    TRUE { $$ = true; }
    | FALSE { $$ = false; }
    | expr AND expr { $$ = $1 && $3; }
    | expr OR expr { $$ = $1 || $3; }
    | NOT expr { $$ = !$2; }
    | LPAREN expr RPAREN { $$ = $2; }
;

%%

void yyerror(const char *s) {
    fprintf(stderr, "Error: %s\n", s);
}

int main() {
    return yyparse();
}


```

3. Write First and Follow set of the CFG. [7 Marks]

$$
\begin{aligned}
E &\rightarrow TE' \\
E' &\rightarrow +TE' \mid \epsilon \\
T &\rightarrow FT' \\
T' &\rightarrow *FT' \mid \epsilon \\
F &\rightarrow (E) \mid id
\end{aligned}
$$

FIRST set

$$
\begin{aligned}
\text{FIRST}(E) = \text{FIRST}(T) &= \{(,\ id\} \\
\text{FIRST}(E') &= \{+,\ \epsilon\} \\
\text{FIRST}(T) &= \{(,\ id\} \\
\text{FIRST}(T') &= \{*,\ \epsilon\} \\
\text{FIRST}(F) &= \{(,\ id\}
\end{aligned}
$$

FOLLOW set

$$
\begin{aligned}
\text{FOLLOW}(E) &= \{\$,\ )\} \\
\text{FOLLOW}(E') &= \text{FOLLOW}(E) = \{\$,\ )\} \\
\text{FOLLOW}(T) &= \text{FIRST}(E') - \{\epsilon\} \cup \text{FOLLOW}(E') \cup \text{FOLLOW}(E) = \{+,\ \$,\ )\} \\
\text{FOLLOW}(T') &= \text{FOLLOW}(T) = \{+,\ \$,\ )\} \\
\text{FOLLOW}(F) &= \text{FIRST}(T') - \{\epsilon\} \cup \text{FOLLOW}(T') \cup \text{FOLLOW}(T) = \{*,\ +,\ \$,\ )\}
\end{aligned}
$$

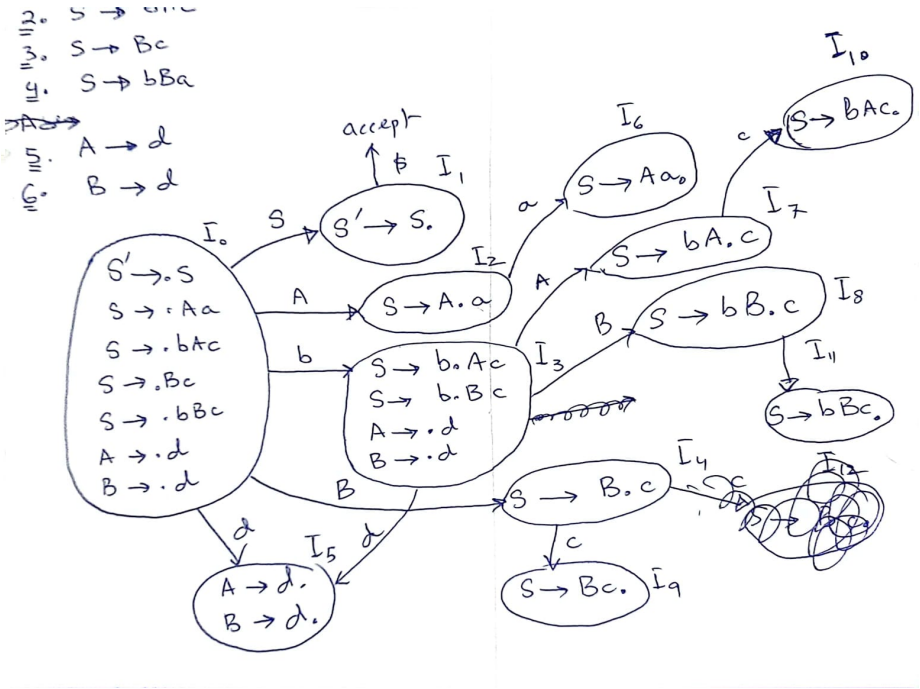

Figure 1: LR(0) automaton

4. Given the following Context-Free Grammar (CFG): [10 Marks]

$$S \rightarrow Aa \mid bAc \mid Bc \mid bBa$$
$$A \rightarrow d$$
$$B \rightarrow d$$

(a) **Create LR(0) Automaton:**

- Construct the LR(0) automaton for the given grammar.
- Identify and point out all the conflicts in the LR(0) automaton.

(b) **Create SLR Parsing Table:**

- Based on the LR(0) automaton, create the SLR parsing table.
- Resolve the conflicts identified in the LR(0) automaton.
- State whether the given grammar is SLR or not.

The LR(0) automaton is as follows:

There is reduce-reduce conflict in state I_5

So we have to use follow set. **FOLLOW set**

$$\text{FOLLOW}(S) = \{\$\}$$
$$\text{FOLLOW}(A) = \{a, c\}$$
$$\text{FOLLOW}(B) = \{a, c\}$$

Nothing change the grammar is not SLR. Still has conflicts.

5. Given the parsing table, create the parse tree for the input string $(id + id) * id$. Show the stack and input buffer at each step while applying the parsing algorithm. [7 Marks]

   This question is cancelled as it has error in it.

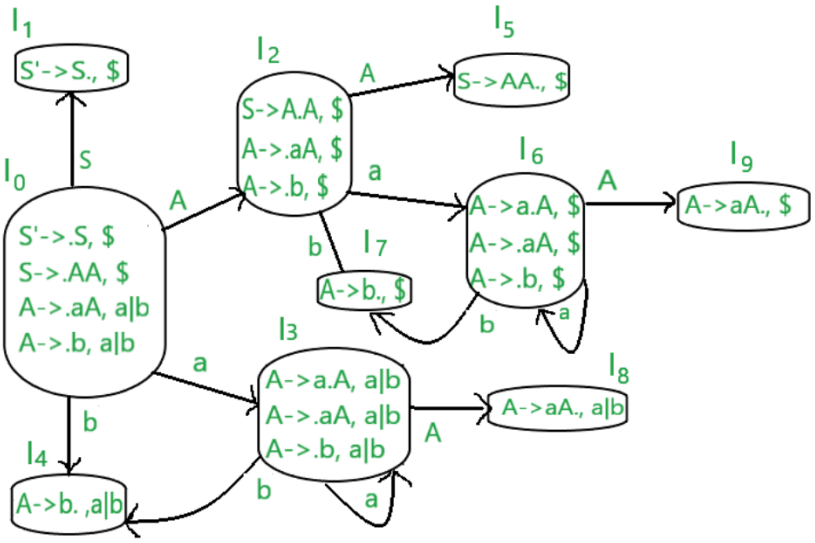

Figure 2: LR(1) Automaton

6. Create CLR and LALR parsing tables of the CFG: [10 Marks]

$$S \rightarrow AA$$
$$A \rightarrow aA \mid b$$

Our augmented grammar is as follows:

$S' \rightarrow .S$	[0]
$S \rightarrow .AA$	[1]
$A \rightarrow .aA$	[2]
$A \rightarrow .b$	[3]

There is a shift-reduce conflict in the table thus the grammar is not an LALR grammar.

State	Action			Goto	
	a	b	$	A	S
0	s3	s4		2	1
1			acc		
2	s6	s7		5	
3	s3	s4		8	
4	r3	r3			
5			r1		
6	s6	s7		9	
7			r3		
8	r2	r2			
9			r2		

Table 1: LALR Parsing Table 1

State	Action			Goto	
	a	b	$	A	S
0	s3	s4		2	1
1			acc		
2	s6	s7		5	
36	s36	s74	r1	89	
47	r3	r3s7	r3		
5			r1		
89	r2	r2	r2		

Table 2: LALR Parsing Table 2

7. Perform live-variable analysis on the following three-address code (IR) function. First, create a control flow graph (CFG) for the given function. Then, determine the live variables at each program point within the function. To that end create table with Gen, Kill of each basic block. Furthermore use them to calculation In, Out of that basic block for two different iteration[10 Marks]

   Three-Address Code Function

```
start:
  t1 = 10
  t2 = 1
  t3 = 0
  if t1 >= 10 goto if_body else end
if_body:
  t4 = t1 > 0
  if t4 goto while_body else end
while_body:
  print t1, t2, t3
  t5 = t2 * t1
  t6 = t3 * t5
  t7 = t1 - 1
  goto if_body
end:
  print t1, t2, t3
  t8 = 0
  return
```

BB	Gen	Kill	In (#1)	Out (# 1)	In (# 2)	Out (# 2)
start	{}	{t1, t2, t3}	{}	{t1, t2, t3}	{ }	{t1, t2, t3}
if_body	{t1}	{t4}	{t1, t2, t3}	{t1, t2, t3}	{t1, t2, t3}	{t1, t2, t3}
while_body	{t1, t2, t3}	{t5, t6, t7}	{t1, t2, t3}	{ }	{t1, t2, t3}	{t1, t2, t3}
end	{t1, t2, t3}	{t8}	{t1, t2, t3}	{ }	{t1, t2, t3}	{ }

Table 3: Live-Variable Analysis for Each Basic Block