Key solution

King Saud University
College of Computer and Information Sciences
Computer Science Department

Final Exam

Academic Year: 2015/2016
Second Semester

BSc Program
Course Name/No. : Programming Language Compilation / CS340

Exam Date: 19 /5/2016:

Total Number of Pages: 9 pages (including this cover page)

Student Name	
Student ID.	Seat Number

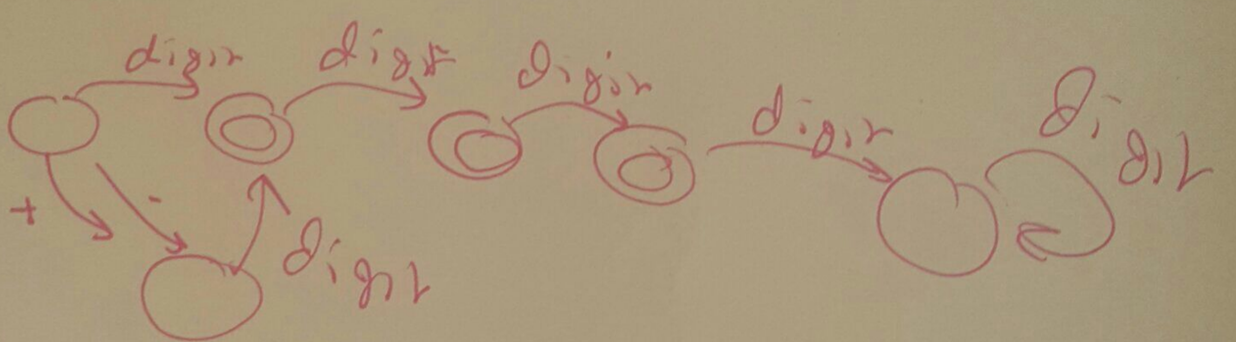
Question No.	Total Mark	Student Mark
1.	**5**	
2.	**7**	
3.	**6**	
4.	**9**	
5.	**6**	
6.	**7**	
Total	**40**	

Q1) A) Write a regular expression that describes the language of all integers that consist of at most 3 digits with an optional sign (+ or -). (2 marks)

digit = '0' + '1' + '2' + - - - + '9'

(+|-|ε)(digit + digit digit + digit digit digit)

B) Design a DFA that accepts an integers that if it consist of at most 3 digits with an optional sign (+ or -). (2 marks)



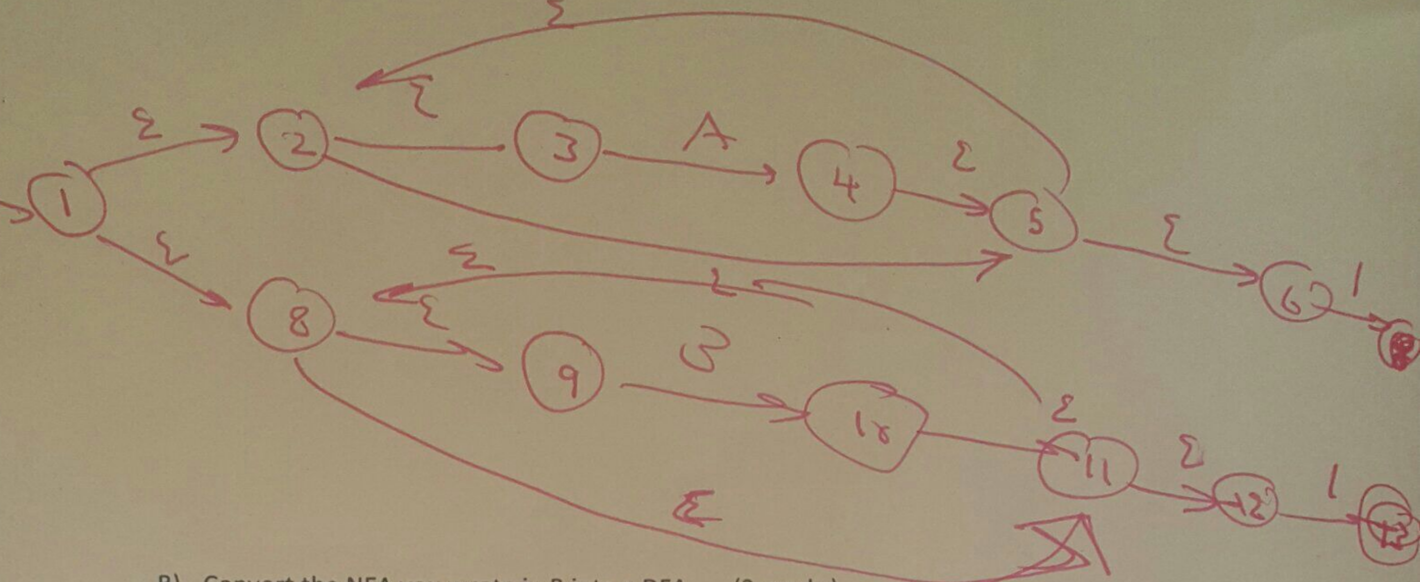
c) Give an example of a language that is irregular (not regular) (1 mark)

$= \{ (^n)^n \}$

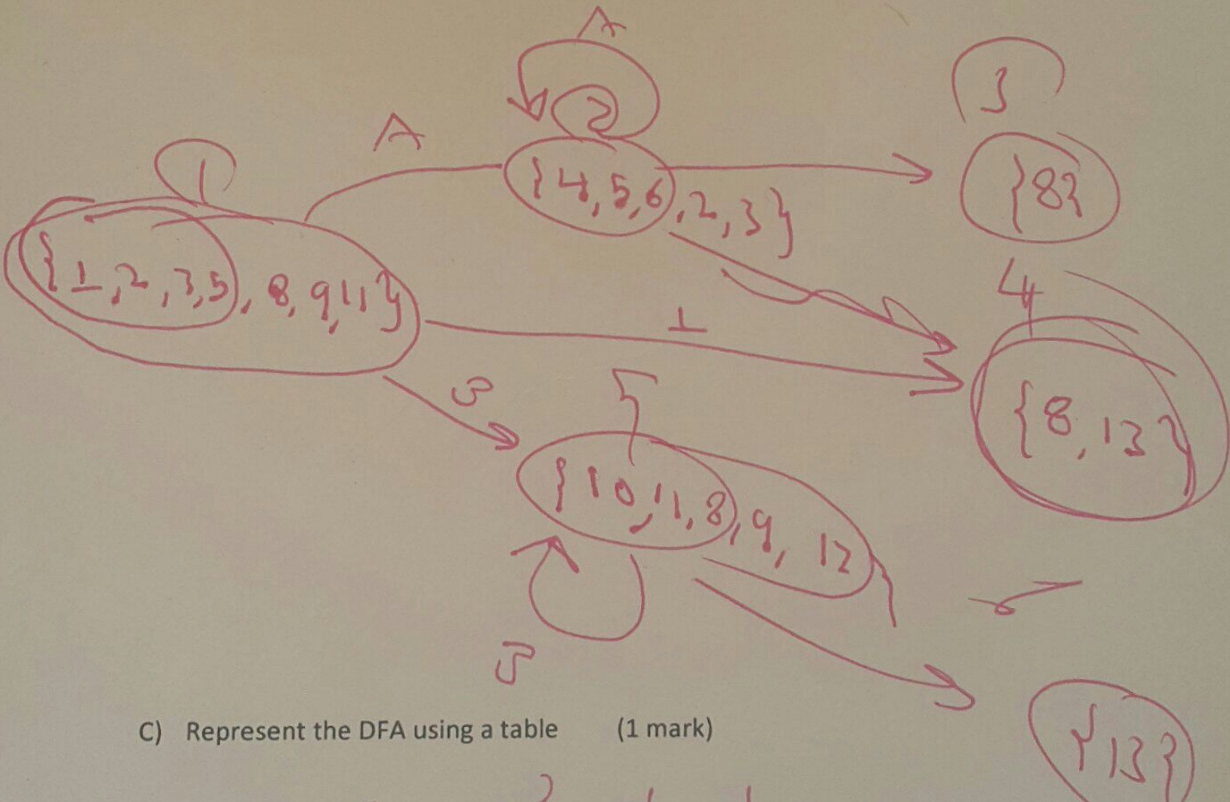
Q2) Consider the following regular expression

(A*+B*)1

A) Convert the regular expression you wrote above into an NFA. (2 marks)



B) Convert the NFA you wrote in B into a DFA. (2 marks)



C) Represent the DFA using a table (1 mark)

	A	B	1
1	2	~~5~~	4
2	2		3
3			
~~4~~		5	B

D) Write an algorithm (pseudo code) that a string as input and displays "yes" if a string belongs to the language **(A*+B*)1**; otherwise it displays "no". (2 marks)

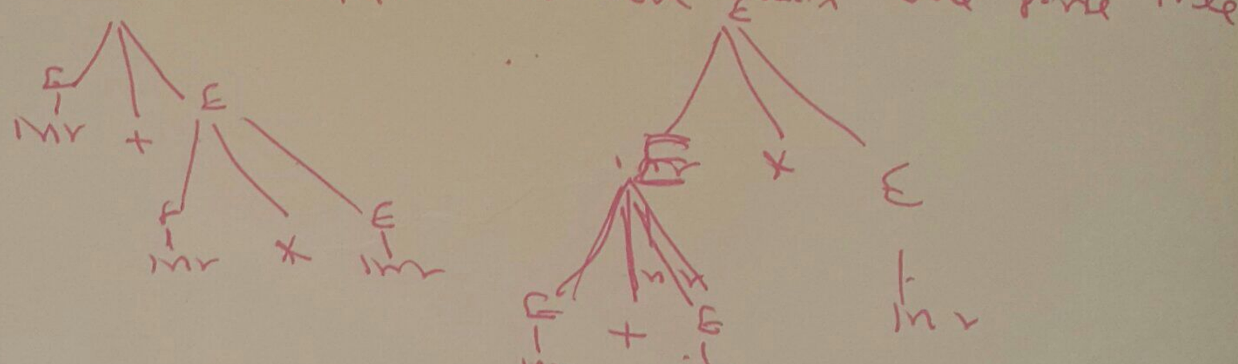
```
input = 0
state = 0
while (moreInput)
    state = T[state, input++];
if (state = accept state)
    print "yes"
else
    print "no"
```

Q3) Consider the CFG

E→E+E|E*E|int|id

a) Give an example that illustrates that the above grammar is ambiguous. (2 marks)

int + int * int has more than one parse tree



b) Rewrite the grammar so that * has higher precedence than +. (2 marks)

$E \rightarrow E + E` \mid E`$

$E` \rightarrow E` * E` \mid int \mid id$

int + int * n

C) Assuming that ambiguity is fine, rewrite the given grammar so that we can write a recursive descent parser for it. (2 marks)

$E \rightarrow int + E \mid id + E \mid int * E \mid id * E$

Q4) Consider the following grammar

E→TX
T→(E) | int Y | ε
X→+E | ε
Y→*T | ε

A) Find the first set and follow set for each terminal and non-terminal symbol used in the grammar. (3 marks)

follow(T) = first(X) ∪ follow(E)
follow(E) = { $,) } ∪ ~~follow(T)~~ ∪ follow(X)
~~follow~~(T) = follow(E) = follow(
{ $,) }

Symbol	First set	Follow set
T	{ (, int, ε }	{ +, $,) }
E	{ (, int, +, ε }	{ $,), + }
X	{ +, ε }	{ $,), + }
Y	{ *, ε }	{ $,), + }
(	{ (}	{ (, int, + } = first(E)
)	{) }	follow(T) = { +, $,) }
int	{ int }	{ *, +, $,) } = first(Y) ∪ follow(T)
+	{ + }	first(E) = { (, int, + } $,) }
*	{ * }	first(T) ∪ follow(Y) = { (, int, $,), + }

B) Draw the LL(1) parsing table. (2 marks)

	(	)	int	+	*	$
E	TX	TX	TX	TX		TX
T	(E)	ε	int Y	ε		ε
X		ε		+E		ε
Y		ε		ε	*T	ε

E) Write down the SLR(1) parsing algorithm. (2 marks)

c) Assuming dynamic type checking, rewrite the above rules so that they work for reference types. (2 marks)

$$\frac{S \vdash e_1 : T \quad S \vdash e_2 : T}{S \vdash e_1 = e_2 : T}$$

if ~~e1~~ and e2 are of the same type, T, then the assignment exp $e_1 = e_2$ has also type T

Q6) A) Write cgen for the expression e1+e2 (3 marks)

```
cgen( e1+e2)
cgen(e1)
sw $a0 0($sp)
addiu $sp -4
cgen( e2)
lw $t1 ~~$a0~~ 4($sp)
add $a0 $a0 $t1
addiu $sp $sp 4
```

Q5) a) Why is "null" causes a problem for type systems? And how do they deal with it?

(2 marks)

we declare a null type at the bottom of the inheritance hierarchy

b) In your own words, describe what the following rule means. (2 marks)

1.

$$\frac{\begin{array}{c} f \text{ is an identifier.} \\ f \text{ is a non-member function in scope S.} \\ f \text{ has type } (T_1, \ldots, T_n) \to U \\ S \vdash e_i : T_i \text{ for } 1 \le i \le n \end{array}}{S \vdash f(e_1, \ldots, e_n) : U}$$

if the arguments $T_1, \ldots, T_n$ have type $e_1, \ldots, e_n$ then the function invocation $f(e_1, \ldots, e_n)$ has type U

B) Write code that cgen generates for the expression 5+7. (4 marks)

```
li $a0 5
sw $a0 0($sp)
addiu $sp 4
li $a0 7
lw $t1 4($sp)
add $a0 $a0 $t1
addiu $sp $sp 4
```

Some MIPS instruction:

- lw reg1 offset(reg2) % load 32-bit word from address reg2+offset into reg1
- add reg1 reg2 reg3 % reg1 = reg2+reg3
- sw reg1 offset(reg2) % Store 32-bit word in reg1 at address reg2 + offset
- addiu reg1 reg2 imm % reg1=reg2+imm
- li reg imm % reg=imm
- beq reg1 reg2 label % branch to label if reg1=reg2
- ble reg1 reg2 label % branch if reg1<=reg2
- b label % unconditional jump to label

Register names:

Recall that in MIPS the accumulator is $a0, the Stack pointer is $sp, and the temporary register is $t1