UNIVERSITY OF LONDON

GOLDSMITHS COLLEGE

B.Sc. Examination 2007

COMPUTING AND INFORMATION SYSTEMS

IS53011A (CIS324) Language Design and Implementation

Duration: 2 hours 15 minutes

Date and time:

- *Full marks will be awarded for complete and correct answers to THREE questions. Do not attempt more than THREE questions on this paper. Although each question carries 25 marks, and therefore the result from three questions sums up to 75 marks, the final result will be additionally scaled to 100.*
- *Electronic calculators are not allowed.*

THIS EXAMINATION PAPER MUST NOT BE REMOVED FROM THE EXAMINATION ROOM

Question 1.

a) Draw the algorithmic structure of a language processing system. **[5]**

b) Give answers to the following questions: **[6]**

i) What does it mean to say that the multiplication operator * has higher precedence than the summation operator + in a programming language like Java?

ii) When does an operator have a higher precedence than another operator?

c) Consider the following simple programming language grammar:

$B \rightarrow \{ B \} \mid D$
$D \rightarrow x := T$
$T \rightarrow T + F \mid T - F \mid F$
$F \rightarrow 1 \mid 2 \mid (T)$

Using this grammar develop the parse tree for the expression:
$\{ x := (1 + (2 - 1)) - 1 \}$. **[8]**

d) Write the following regular grammar in a more compact form: $((\in \mid d) c^*)^*$.
Decide whether the string: $dcdccdc$ can be derived from this grammar. **[6]**

Question 2.

a) Define recursively the notion of regular expressions over a given alphabet using values: Ø, $\in$, symbols: a, b, and the necessary operations. **[4]**

b) Consider a regular language grammar defined by the expression: (a | (a | b)*).

i) Design a nondeterministic finite state automaton (NFA) for this language using Thompson's construction algorithm. **[5]**

ii) Convert the NFA from part (i) into a corresponding deterministic finite-state automaton (DFA) using the subset construction algorithm. Demonstrate the computation of the $\in$-*closure* and *move* functions leading to the DFA. **[13]**

iii) Draw the transition graph for the resulting DFA from part (ii). **[3]**

Question 3.

a) Eliminate the immediate left recursions from the following productions: **[4]**

$$S \rightarrow SS \mid SAa \mid Ab \mid a$$

b) Assume the following grammar for top-down nonrecursive predictive parsing:

$$S \rightarrow ASb \mid Ba$$
$$A \rightarrow Bb \mid ac$$
$$B \rightarrow cSa \mid dA \mid \in$$

i) Compute the functions *FIRST* and *FOLLOW* necessary for parser construction. **[6]**

ii) Build the nonrecursive predictive parsing table for this grammar. **[9]**

iii) Show the stack, the input and the output of the nonrecursive predictive parsing algorithm on the following input: $daca$. **[6]**

Question 4.

a) Let the following *LR* grammar be given for bottom-up parsing:

$S' \rightarrow S$
$S \rightarrow aBc \mid B$
$B \rightarrow cd$

i) Develop the canonical collection of items from this grammar. **[7]**

ii) Construct the DFA whose states are these sets of valid items. **[4]**

b) Explain the operation of the *LR* parsing algorithm on the input: $acdb$\$ using the grammar given below. Show the stack, the input and the output. **[14]**

(1) $S' \rightarrow S$
(2) $S \rightarrow aAb$
(3,4) $A \rightarrow cd \mid c$

	Action					Goto	
State	*a*	*b*	*c*	*d*	$	*S*	*A*
0	*s1*					*6*	
1			*s2*				*4*
2		*r4*		*s3*			
3		*r3*					
4		*s5*					
5					*r2*		
6					*acc*		

Question 5.

a) Which are the four transformation techniques applied most frequently for code optimisation at the final phase of compilation? **[4]**

b) Optimise the three-address code below using an appropriate technique. **[4]**

```
t1 := 4 * i
z := a[ t1 ]
t2 := 4 * i
t3 := 4 * j
t4 := a[ t3 ]
a[ t2 ]:= t4
t5 := 4 * j
a[ t5 ]:= z
```

c) Which two important properties should an optimising compiler provide? **[5]**

d) The following program segment swaps two integers within an array:

```
void swap( int A[], int n, int x, int y )
{
    int temp;

    if ( y < n )
    {
        temp = A[ x ]; A[ x ] = A[ y ]; A[ y ] = temp;
    }
}
main ()
{
    int a[ 2 ] = { 1, 2 };
    swap( a, 2, 0, 1 );

}
```

Generate three-address intermediate code for this simple program fragment. **[12]**

UNIVERSITY OF LONDON

GOLDSMITHS COLLEGE

B.Sc. Examination 2007

COMPUTING AND INFORMATION SYSTEMS

IS53011A (CIS324) Language Design and Implementation

Duration: 2 hours 15 minutes

Date and time:

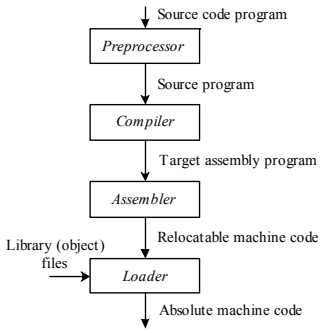
- *Full marks will be awarded for complete and correct answers to THREE questions. Do not attempt more than THREE questions on this paper. Although each question carries 25 marks, and therefore the result from three questions sums up to 75 marks, the final result will be additionally scaled to 100.*
- *Electronic calculators are not allowed.*

THIS EXAMINATION PAPER MUST NOT BE REMOVED FROM THE EXAMINATION ROOM

Solutions CIS324

Question 1.

a) Draw the algorithmic structure of a language processing system. **[5]**



b) Give answers to the following questions: **[2x3=6]**

i) What does it mean to say that the multiplication operator * has higher precedence than the summation operator + in a programming language like Java?

The operator precedence applies when the order of operator evaluation is not defined by the explicit use of parentehses.

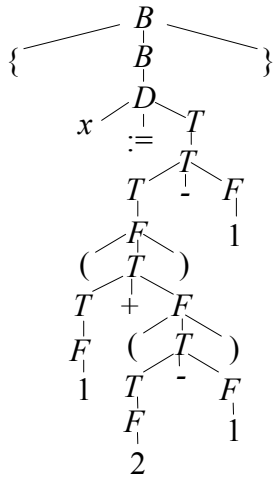
ii) When does an operator has a higher precedence than another operator?

An operator precedence has a higher precedence than another operator when it takes its operands before it.

c) Consider the following simple programming language grammar:

$B \rightarrow \{ B \} \mid D$
$D \rightarrow x := T$
$T \rightarrow T + F \mid T - F \mid F$
$F \rightarrow 1 \mid 2 \mid (T)$

Using this grammar develop the parse tree for the expression: $\{ x := (1 + (2 - 1)) - 1 \}$. **[8]**



d) Write the following regular grammar in a more compact form: $((\in \mid d)\, c^*)^*$.
Decide whether the string: $dcdccdc$ can be derived from this grammar. **[6]**

This regular grammar can be rewritten in a more compact form as follows: $(d \mid c)^*$.
Yes, the string $dcdccdc$ can be derived from this regular grammar.

Question 2.

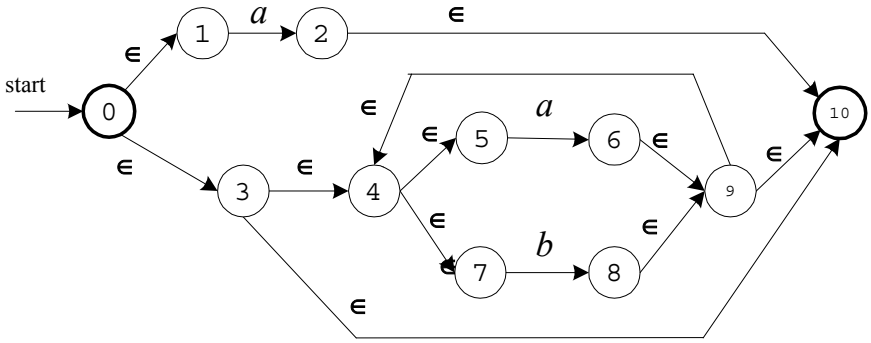
a) Define recursively the notion of regular expressions over a given alphabet using values: Ø, ∈, symbols: a, b, and the necessary operations. **[4]**

Regular expressions over a given alphabet ∑ are defined as follows:

- the value Ø is a regular expression which denotes the empty set {}
- the value ∈ is a regular expression which denotes the set {∈}
- for each symbol value a from ∑ it is a regular expression which denotes the set {a}
- if a and b are regular expressions from the sets P and Q then the three operations union (a | b), concatenation (ab) and closure (a*) yield regular expressions which denote respectively the sets P U Q, PQ and P* .

b) Consider a regular language grammar defined by the expression: (a | (a | b)*).

i) Design a nondeterministic finite state automaton (NFA) for this language using Thompson's construction algorithm. **[5]**



ii) Convert the NFA from part (i) into a corresponding deterministic finite-state automaton (DFA) using the subset construction algorithm. Demonstrate the computation of the ∈-*closure* and *move* functions leading to the DFA. **[4x2+5=13]**

∈-*closure*({0}) = ∈-*closure* { 0, 1, 3 } = { 0, 1, 3, 4, 5, 7, 10 } = *A*

∈-*closure*(*move*(*A*, *a*)) = ∈-*closure*(*move*({ 0, 1, 3, 4, 5, 7, 10 }, *a*)) = { 2, 4, 5, 6, 7, 9, 10 } = *B*

∈-*closure*(*move*(*A*, *b*)) = ∈-*closure*(*move*({ 0, 1, 3, 4, 5, 7, 10 }, *b*)) = { 4, 5, 7, 8, 9, 10 } = *C*

∈-*closure*(*move*(*B*, *a*)) = ∈-*closure*(*move*({ 2, 4, 5, 6, 7, 9, 10 }, *a*)) = { 4, 5, 6, 7, 9, 10 } = *D*

∈-*closure*(*move*(*B*, *b*)) = ∈-*closure*(*move*({ 2, 4, 5, 6, 7, 9, 10 }, *b*)) = *C*

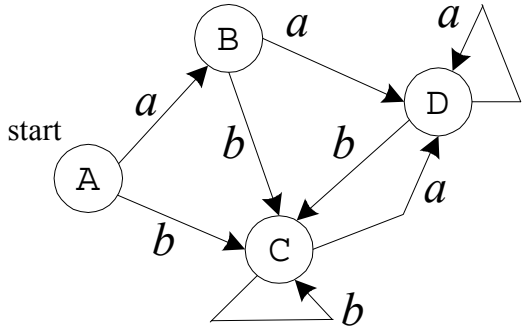
∈-*closure*(*move*(*C*, *a*)) = ∈-*closure*(*move*({ 4, 5, 7, 8, 9, 10 }, *a*)) = *D*

∈-*closure*(*move*(*C*, *b*)) = ∈-*closure*(*move*({ 4, 5, 7, 8, 9, 10 }, *b*)) = *C*

∈-*closure*(*move*(*D*, *a*)) = ∈-*closure*(*move*({ 4, 5, 6, 7, 9, 10 }, *a*)) = *D*

∈-*closure*(*move*(*D*, *b*)) = ∈-*closure*(*move*({ 4, 5, 6, 7, 9, 10 }, *b*)) = *C*

iii) Draw the transition graph for the resulting DFA from part (ii). **[3]**



Question 3.

a) Eliminate the immediate left recursions from the following productions: **[4]**

$S \rightarrow SS \mid SAa \mid Ab \mid a$

The left recursions are eliminated by grouping and replacement as follows:

$S \rightarrow AbS' \mid aS'$
$S' \rightarrow SS' \mid AaS' \mid \in$

b) Consider the following grammar for top-down nonrecursive predictive parsing:

$S \rightarrow ASb \mid Ba$
$A \rightarrow Bb \mid ac$
$B \rightarrow cSa \mid dA \mid \in$

i) Compute the functions *FIRST* and *FOLLOW* necessary for parser construction. **[6x1=6]**

FIRST (S) = { $a, c, d, \in$ } - rule 3
FIRST (A) = { $a, c, d, \in$ } - rule 3
FIRST (B) = { $c, d, \in$ } - rules 3, 2

FOLLOW (S) = { $b, a, \$$ } - rules 1, 2
FOLLOW (A) = { a, b, c, d } - rules 2, 3
FOLLOW (B) = { a, b } - rule 2

ii) Build the nonrecursive predictive parsing table for this grammar. **[3x3=9]**

	a	b	c	d	$
S	$S \rightarrow ASb$		$S \rightarrow Ba$	$S \rightarrow Ba$	
A	$A \rightarrow ac$		$A \rightarrow Bb$	$A \rightarrow Bb$	
B	$B \rightarrow \in$	$B \rightarrow \in$	$B \rightarrow cSa$	$B \rightarrow dA$	

iii) Show the stack, the input and the output of the nonrecursive predictive parsing algorithm on the following input: $daca$. **[6x1=6]**

Stack	Input	Output
$ S	$daca$ $	
$ aB	$daca$ $	$S \rightarrow Ba$
$ aAd	$daca$ $	$B \rightarrow dA$
$ aA	aca $	
$ aca	aca $	$A \rightarrow ac$
$ ac	ca $	
$ a	a $	

Question 4.

a) Let the following *LR* grammar be given for bottom-up parsing:

$S' \rightarrow S$
$S \rightarrow aBc \mid B$
$B \rightarrow cd$

i) Develop the canonical collection of items from this grammar. **[7]**

I_0: $S' \rightarrow \bullet S$, $S \rightarrow \bullet aBc$, $S \rightarrow \bullet B$, $B \rightarrow \bullet cd$

I_1: $S' \rightarrow S\bullet$

I_2: $S \rightarrow a\bullet Bc$, $B \rightarrow \bullet cd$

I_3: $S \rightarrow B\bullet$

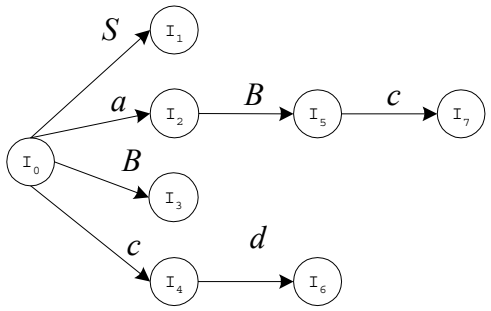
I_4: $B \rightarrow c\bullet d$

I_5: $S \rightarrow aB\bullet c$

I_6: $B \rightarrow cd\bullet$

I_7: $B \rightarrow aBc\bullet$

ii) Construct the DFA whose states are these sets of valid items. **[4]**



b) Explain the operation of the *LR* parsing algorithm on the input: *acdb*\$ using the grammar given below. Show the stack, the input and the output. **[7x2=14]**

(1) $S' \rightarrow S$
(2) $S \rightarrow aAb$
(3,4) $A \rightarrow cd \mid c$

	Action					Goto	
State	*a*	*b*	*c*	*d*	\$	*S*	*A*
0	*s1*					*6*	
1			*s2*				*4*
2		*r4*		*s3*			
3		*r3*					
4		*s5*					
5					*r2*		
6					*acc*		

Stack	Input	Action
(1) 0	*acdb* \$	*shift*
(2) 0 *a* 1	*cdb* \$	*shift*
(3) 0 *a* 1 *c* 2	*db* \$	*shift*
(4) 0 *a* 1 *c* 2 *d* 3	*b* \$	*reduce by* $A \rightarrow cd$
(5) 0 *a* 1 *A* 4	*b* \$	*shift*
(6) 0 *a* 1 *A* 4 *b* 5	\$	*reduce by* $S \rightarrow aAb$
(7) 0 *S* 6	\$	*acc*

Question 5.

a) Which are the four transformation techniques applied most frequently for code optimisation at the final phase of compilation? **[4]**

The four transformation techniques applied most frequently for code optimisation are:
- Function-preserving transformations;
- Common subexpressions identification;
- Copy propagation;
- Induction variables and reduction in strength.

b) Optimise the three-address code below using an appropriate technique. **[4]**

```
t1 := 4 * i
z := a[ t1 ]
t2 := 4 * i
t3 := 4 * j
t4 := a[ t3 ]
a[ t2 ]:= t4
t5 := 4 * j
a[ t5 ]:= z
```

This three-address code can be optimised by common subexpression elimination:

```
t1 := 4 * i
z := a[ t1 ]
t2 := 4 * j
t3 := a[ t2 ]
a[ t1 ]:= t3
a[ t2 ]:= z
```

c) Which two important properties should an optimising compiler provide? **[2x2.5=5]**

An optimising compiler should provides the following two properties:
- the transformations should preserve the semantics of the programs, that is the changes should guarantee that the same input produces the same outputs;
- the transformations should speed up the compiler;

d) The following program segment swaps two integers within an array:

```
void swap( int A[], int n, int x, int y )
{
    int temp;

    if ( y < n ) { temp = A[ x ]; A[ x ] = A[ y ]; A[ y ] = temp; }
}
main ()
{
    int a[ 2 ] = { 1, 2 }; swap( a, 2, 0, 1 );

}
```

Generate three-address intermediate code for this simple program fragment. **[20x0.6=12]**

```
( 1 )    s := mktable( nil )
( 2 )    push( s, temp, int, 4 )
( 3 )    if y >= n goto (11)
( 4 )    t1 := 4 * x
( 5 )    t2 := A[ t1 ]
( 6 )    temp := t2
( 7 )    t3 := 4 * y
( 8 )    t4 := A[ t3 ]
( 9 )    A[ t1 ] := t4
( 10 )   A[ t3 ] := temp
( 11 )   pushproc( s, swap, proc, 1*4 )
( 12 )   m := mktable( nil )
( 13 )   push( m, A, array, 2*4 )
( 14 )   A[ 0 ] := 1
( 15 )   A[ 1 ] := 2
( 16 )   param A
( 17 )   param 2
( 18 )   param 0
( 19 )   param 1
( 20 )   call swap, 4
```