E2.15

IMPERIAL COLLEGE LONDON

DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
EXAMINATIONS 2009

ISE PART II: MEng, BEng and ACGI

Corrected Copy

LANGUAGE PROCESSORS

Monday, 15 June 2:00 pm

Time allowed: 2:00 hours

There are FOUR questions on this paper.

Q1 is compulsory.
Answer Q1 and any two of questions 2-4.
Q1 carries 40% of the marks. Questions 2 to 4 carry equal marks (30% each).

Any special instructions for invigilators and information for candidates are on page 1.

Examiners responsible First Marker(s) : Y.K. Demiris, Y.K. Demiris

Second Marker(s) : J.V. Pitt, J.V. Pitt

The Questions

1. [COMPULSORY]

(a) Provide the formal definition for a Push-Down Automaton (PDA). Describe the differences between a PDA and a Linearly-Bounded Automaton (LBA). Provide an example language that cannot be recognized by a PDA, and explain why. [6]

(b) Show that the language $\{a^n b c^n \mid n>=0\}$ is LL(1) by providing the LL(1) grammar that can generate it. Explain why your provided grammar is LL(1). [4]

(c) In the context of LL grammars, provide the *formal* definition of the left-factoring method. Describe the significance of left-factoring, and apply it to the grammar below

P -> aPb
P -> aPc
P -> d

[6]

(d) Provide the definition of LR(0) items, explain their significance and provide the LR(0) items for the production rule K -> ABCD [4]

(e) In the context of bottom up parsing, the construction of the parsing table requires the definition of the *closure* and *goto* functions. Provide these two definitions. [6]

(f) Provide the definitions of L-attributed and S-attributed grammars and explain how the attributes of each of these grammars can be evaluated during bottom-up parsing. [8]

(g) Describe the graph colouring method for register allocation, and provide a heuristic method for determining whether a graph is k-colourable. [6]

2. You are required to construct the minimal deterministic finite state automaton (DFA) for the regular expression *a*(b|c)*d** following the steps below. You should clearly mark all final states in all the automata you construct.

(a) Construct a non-deterministic finite automaton (NFA) using Thompson's algorithm. [10]

(b) Construct the equivalent DFA using the subset construction algorithm. *Explain the intermediate steps you have taken.* [10]

(c) Describe the DFA minimization algorithm, and subsequently apply it to the DFA you have constructed in (b). Show whether your DFA was already minimal or not. *Explain clearly the intermediate steps of the application of the DFA minimization algorithm.* [10]

3. (a) For the grammar below, construct its augmented version and its canonical collection of sets of LR(0) items

1. E -> E + T
2. E -> T
3. T -> T * F
4. T -> F
5. F -> (E)
6. F -> id

[6]

(b) The computation of the canonical collection above is the first step in the construction of an SLR parsing table for the grammar. Provide the remaining steps of the algorithm. Subsequently use the algorithm to construct the SLR parsing table for the grammar of (a), using the table format below (three entries already completed). Any left blank entries will be taken as indicating parsing error.

(NB: Make sure you copy this table to your exam booklet!)

[24]

State	Action						Goto		
	id	+	*	(	)	$	E	T	F
0	s5						1		
1									
2		r2							
3									
4									
5									
6									
7									
8									
9									
10									
11									

4. (a) Calculate the FIRST and FOLLOW sets for all non-terminal symbols for the grammar below where {i, a, b, t, e } are terminals, and {S, S' and E} are non-terminals:

(1) S -> iEtSS' | a
(2) S' -> eS | ε
(3) E -> b

[6]

(b) Construct the parsing table for the grammar above using the table below, with $ denoting the end of input marker. Complete *ALL* table entries, clearly marking any error entries (two examples shown).

(NB: Make sure you copy this table to your exam booklet!) [16]

Non-terminal	**Input Symbol**					
	a	**b**	**e**	**i**	**t**	**$**
S					Error	
S'						
E						Error

(c) Point out how sources of ambiguity in the grammar are manifested in this table. [2]

(d) Note (by mapping {i to IF, E to Expression, t to THEN, e to ELSE and S/S'/a/b to Statements/Exprssions }) that the grammar above is an abstraction of the "dangling-else" problem in if-then-else statement parsing, common in C and Pascal languages. How can we enforce an association of an "else" statement with the closest previous "then" statement?
[Hint: This involves resolving ambiguities in the parsing table] [6]

E2.15: Language Processors

Sample answers to exam questions 2009

Question 1

(a) [Bookwork]:

A PDA is defined as (Q, Σ, Γ, δ, q_0, Z_0, F)

- Q: a finite set of N states $q_0, q_1, \ldots, q_N$
- Σ: a finite input alphabet of symbols
- Γ: a finite stack alphabet – the set of symbols
- $\delta(q,\alpha,X)$: the transition function between states. Given a state $q \in Q$, $\alpha \in \Sigma$ or $\alpha=\varepsilon$, and a stack symbol $X \in \Gamma$, the function $\delta(q,\alpha, X)$ returns a pair (p,γ), where p is the new state and γ is the string of stack symbols that replaces X at the top of the stack
- q_0: the start state
- Z_0: the start stack symbol
- F: the set of final states, $F \subseteq Q$

An LBA is a restricted class of Turing machines, and in contrast to the PDA's stack contains a tape, and a bidirectional read-write head. It can be used to recognize context sensitive grammars in contrast to the PDA.

An example of a language that the PDA cannot recognize is $\{a^n b^n c^n, n>=1\}$, since even if you use the stack to parse equal number of characters of the first two sets, there is no memory of the number parsed to match the third set. An LBA is required for this.

(b) [New computed example]:

The LL(1) grammar is:

S -> aSc
S -> b

It's a LL(1) grammar since we have disjoint FIRST sets for S

(c) [Bookwork]

The left-factoring method proceeds as follows:

For each non-terminal find the longest prefix α common to two or more of its alternatives.
If $\alpha \neq \varepsilon$ (there is a non-trivial common prefix), replace all the A productions
$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma$ [where γ represent all alternatives that do begin with α]
by $A \rightarrow \alpha A' \mid \gamma$
$A' \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$
where A' is a new non-terminal.
Repeatedly apply this transformation until no two alternatives for a non-terminal have a common prefix.

The equivalent grammar is:

P -> aPX
X -> b | c
P -> d

(d) [Bookwork]:

An *LR(0) item* (or simply *item*) of a grammar is a production rule augmented with a position marker (a dot) somewhere within its right hand side.
The production K->ABCD yields the following five items:
K -> .ABCD
K -> A.BCD
K -> AB.CD
K -> ABC.D
K -> ABCD.

Intuitively an LR(0) item indicates how much of a production we have seen at a given point in the parsing process.

(e) [Bookwork]:

The *closure(I)* of a set of items I for a grammar G is the set of items constructed from I using two rules:

1. Initially, every item in I is added in closure(I)

2. If $A \rightarrow \alpha.B\beta$ in closure(I) and $B \rightarrow \gamma$ is a production, then add item $B \rightarrow .\gamma$ to I if its not already there. We apply this rule until no more new items can be added to closure(I)

For a set of items I and a grammar symbol X, the function goto (I, X) is defined as the closure of the set of all items $[A \rightarrow \alpha X.\beta]$ such that $[A \rightarrow \alpha.X\beta]$ is in I

(f) [Bookwork]:

Grammars that do not contain any inherited attributes are called *S-attributed* (i.e. they contain only synthesized attributes). *L-attributed grammars* are grammars where each inherited attribute of X_j ($1 \leq j \leq n$) in a production rule of the form $A \rightarrow X_1X_2...X_n$ depends only on the attributes of the symbols X_1, X_2, .., X_{j-1} to *the left of* X_j in the production and the inherited attributes of A.

Synthesized attributes can be evaluated by a bottom-up parser as the input is being parsed; we can use extra fields in the parser stack to hold the values of the synthesized attributes: each child node stacks its synthesized attributes, and when a *reduce* takes place, the parent node them pops them up, processes them and puts the result back to the stack.

Inherited attributes present an obvious difficulty in the bottom-up evaluation: parent nodes are created after all children have been processed. In a bottom-up parser, the semantic actions can only be executed at the end – when the input has been recognised. But what we need with inherited attributes is calculate them before the symbol that inherits them is processed, i.e.
A -> B {C.inherited_attribute = f (B.attributes} C

A solution is to use ε-productions:
- Insert a new *marker* non-terminal which only generates ε
- Attach the required semantic action at the end of that new ε-production

A -> B M C
M -> ε {C.inh_attribute = f(B.attributes)}

(g) [Bookwork]

Each variable becomes a node in an undirected graph, called the *interference graph*. An arc is drawn between two nodes if they cannot share a register (e.g. because they are live at the same time; they *interfere* with each other wrt register allocation)

This maps the register allocation problem to a well studied graph-colouring problem in graph theory: how to colour the nodes of a graph with the lowest possible number of colours, such that for each arc, the nodes at its ends have different colours.

Determining whether a graph is k-colourable is an NP-complete problem: even the best known algorithm needs an amount of time exponential to the size of the graph to find the optimal colouring. Heuristic algorithms exist:

Suppose that a node *n* in a graph has fewer than k neighbours (nodes connected to n by an edge). Then *n* can be removed along with its edges, resulting into graph G' and reducing the problem to k-colouring of G' (since G can be coloured by assigning to *n* one of the colours not assigned to any of its neighbours.)
Repeat the process until you get either an empty graph (which means that k-colouring of G is possible) OR a graph in which each node has k or more adjacent nodes (which means that k-colouring is not possible) At this point, a node is *spilled* by introducing code to store and reload the register. Interference graph is appropriately modified, and the colouring process is resumed

Question 2
[new computed example]

(a) Applying Thompson' algorithm you get the NFA of figure 1.
(b) Applying the subset construction algorithm on this NFA we get:

ε-closure(StartState) = ε-closure{0} = {0, 1, 3, 4, 5, 6, 7, 11, 12, 13, 15 } = A

Input set = {a, b, c, d}

ε-closure(move(A, a)) = ε-closure{2} = {1, 2, 3, 4, 5, 6, 7, 11, 12, 13, 15} = B
ε-closure(move(A, b)) = ε-closure{8} = {8, 10, 11, 5, 6, 7, 12, 13, 15} = C
ε-closure(move(A, c)) = ε-closure{9} = {9, 10, 11, 5, 6, 7, 12, 13, 15} = D

ε-closure(move(A, d)) = ε-closure{14} = {13, 14, 15} = E

ε-closure(move(B, a)) = ε-closure{2} = B
ε-closure(move(B, b)) = ε-closure{8} = C
ε-closure(move(B, c)) = ε-closure{9} = D
ε-closure(move(B, d)) = ε-closure{14} = E

ε-closure(move(C, a)) = {}
ε-closure(move(C, b)) = ε-closure{8} = C
ε-closure(move(C, c)) = ε-closure{9} = D
ε-closure(move(C, d)) = ε-closure{14} = E

ε-closure(move(D, a)) = {}
ε-closure(move(D, b)) = ε-closure{8} = C
ε-closure(move(D, c)) = ε-closure{9} = D
ε-closure(move(D, d)) = ε-closure{14} = E

ε-closure(move(E, a)) = ε-closure(move(E, b)) = ε-closure(move(E, c)) = {}
ε-closure(move(E, d)) = ε-closure{14} = E, no more new states, we are done.

Figure 1 – the constructed NFA

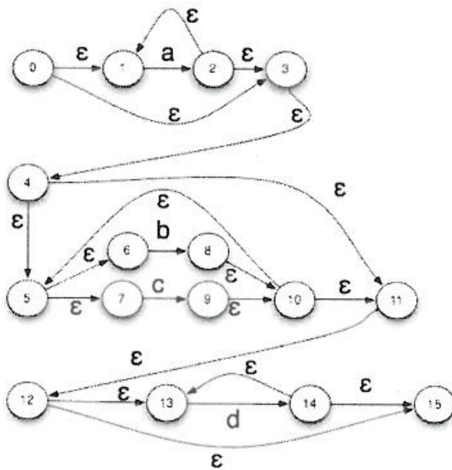
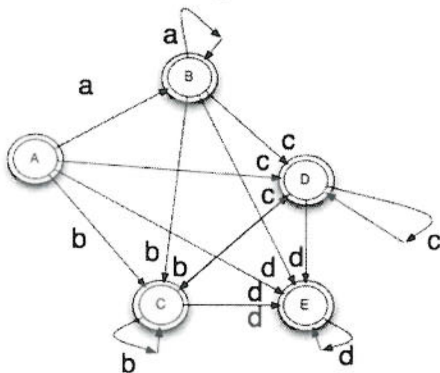


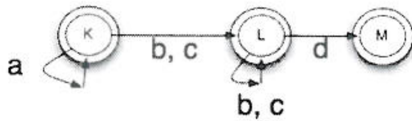
Figure 2 – the resulting DFA



(c) In order to minimise the DFA we obtained from the subset construction algorithm, we start by assuming that all states of the DFA are equal, and we work through the states, putting different states in separate sets if (a) one is final and other is not (b) the transition function maps them to different states, based on the same input character.

The DFA minimization algorithm proceeds by initially creating two sets of states, final and non-final - final: {A, B, C, D, E} and non-final: {}. For each state set created, the algorithm examines the transitions for each state and for each input symbol.

In our case the first set is further subdivided in three sets, one with {A, B} since these two go to the same states for the same symbols, {C, D} for the same reason, and {E}. If we name the first set K (={A, B}), the second set L (={C, D}) and M = {E}, and draw all transitions, we get the following minimal DFA:



Question 3:

(a) [New computed example]

The augmented version simply adds the rule E' -> E to the exist set of production rules.

The canonical collection of sets of LR(0) items.

I_0:
E' -> .E
E- > .E+T
E -> .T
T -> .T*F
T -> .F
F -> .(E)
F -> .id

I_1:
E' -> E.
E -> E.+T

I_2:
E -> T.
T -> T.*F

I_3:
T -> F.

I_4:
F -> (.E)
E -> .E+T
E -> .T
T -> .T*F
T -> .F
F -> .(E)
F -> .id

I_5:
F -> id.

I_6:
E -> E+.T
T -> .T*F
T -> .F
F -> .(E)
F -> .id

I_7:
T -> T*.F
F -> .(E)
F -> .id

I_8:
F -> (E.)
E -> E.+T

I_9:
E -> E+T.
T -> T.*F

I_{10}:
T -> T * F.

I_{11}:
F -> (E).

(b) [New computed example]

Step 1: Construct C = {I_0, I_1, ..., I_n}, the collection of sets of LR(0) items for the augmented grammar.
Step 2. Each set I_i constructs a state i. The parsing entries for state i are determined as follows:

(a) For a terminal *a*, if [A -> α.aB] is in I_i and goto(I_i, a) = I_j, then set action[i, a] to shift j
(b) If [A -> α.] is in I_i, then set action[i,a] to "reduce A -> α for all a in FOLLOW(A) – here A may not be S'
(c) If [S'->S.] is in I_i, then set action[i,$] to "accept".

Step 3. The goto transitions for state i are constructed for all nonterminals A using the rule: if goto(I_i, A) = I_j, then goto[i,A]=j
Step 4. All entries not defined by (2) and (3) are made "error"
Step 5. The initial state of the parser is the one constructed from the set of items containing [S'->.S]

The completed table; all blank entries indicate parsing error

State	Action						Goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Question 4:

(a) [New Computed Example]:

FIRST(S) = { i, a}
FIRST(S')= {e, ε}
FIRST(E) = {b}
FOLLOW(S) = {$, e};
FOLLOW(S') = {$, e}
FOLLOW(F) = {t}

(b) [New Computed Example]: The parsing table for the grammar is:

Non-terminal	Input Symbol					
	a	b	e	i	t	$
S	S->a	Error	Error	S->iEtSS'	Error	Error
S'	Error	Error	S' -> ε S'->eS	Error	Error	S'-> ε
E	Error	E->b	Error	Error	Error	Error

(c) [New computed example]: The ambiguity is manifested by a choice in what production to use when an e is seen.

(d) [New computed example]: Resolving the ambiguity by choosing the S'->eS rule in the ambiguous entry of the table above corresponds to associating an else with the closest previous then.