IMPERIAL COLLEGE LONDON

DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
EXAMINATIONS 2010

ISE PART II: MEng, BEng and ACGI

Corrected Copy

## LANGUAGE PROCESSORS

Wednesday, 9 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) Describe five criteria for judging the quality of a Language Processor. [5]

   (b) Provide the formal definition of a grammar. [5]

   (c) Define the concept of *translation scheme,* and provide the translation scheme for a infix-to-postfix notation translator that only has to deal with a single infix expression, containing one or more addition, subtraction, multiplication and division operators, and single digit numbers as operands. Apply the translation scheme to the expression *9+5-2* and draw the corresponding parse tree. [8]

   (d) Describe the data structures that are used during LR parsing, as well as the operation of the LR parsing algorithm. [6]

   (e) Describe how $\varepsilon$-transitions can be used to allow the calculation of inherited attributes in bottom-up parsing. Describe the potential side effects and provide an example of such effect. [8]

   (f) Provide the algorithm for partitioning a sequence of three-address statements into basic blocks, and explain the utility of this algorithm. [4]

   (g) The language $\{a^n b^n, n \geq 1\}$ can be generated by the following grammar:

   $$S \rightarrow ab$$
   $$S \rightarrow aSb$$

   Within the context of Chomsky's hierarchy of grammars, explain why this grammar is *not* a regular (type-3) grammar, describe what type of grammar it is, and why. [4]

2.      Construct the minimal deterministic finite state automaton (DFA) for the regular expression  *a(b|c|d)\*e*  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,

0. G' -> G
1. G -> G - X
2. G -> X
3. X -> X * F
4. X -> F
5. F -> ( G )
6. F -> a

compute the canonical set of LR(0) items [Hint: set contains 12 items]. [10]

(b) Each set $I_i$ from the ones computed above constructs a state i. The parsing entries for state i are determined by the following three rules:

(Rule 1) 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.
(Rule 2) 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'.
(Rule 3) If [S'->S.] is in $I_i$, then set action[i,$] to "accept".

The goto transitions for state i are constructed for all nonterminals using the rule:
(Rule 4) if goto($I_i$, A) = $I_j$, then goto[i,A]=j.

All entries not defined by the rules above are made "error", indicated by blank entries.

Using the canonical set of LR(0) computed in (3a), and the table construction algorithm above, construct the parsing table for this grammar using the table format below (three entries already completed). [20]

| State | Action | | | | | | Goto | | |
|---|---|---|---|---|---|---|---|---|---|
| | a | - | * | ( | ) | $ | G | X | F |
| 0 | | | | | | | 1 | | |
| 1 | | s6 | | | | | | | |
| 2 | | r2 | | | | | | | |
| 3 | | | | | | | | | |
| 4 | | | | | | | | | |
| 5 | | | | | | | | | |
| 6 | | | | | | | | | |
| 7 | | | | | | | | | |
| 8 | | | | | | | | | |
| 9 | | | | | | | | | |
| 10 | | | | | | | | | |
| 11 | | | | | | | | | |

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

4. (a) Calculate the FIRST and FOLLOW sets for all non-terminal symbols for the grammar below where {x, y, z, t, v} are terminals, and {K, L, M, N, P} are non-terminals:

|     |            |
|-----|------------|
| (1) | K -> M L   |
| (2) | L -> y M L |
| (3) | L -> ε     |
| (4) | M -> P N   |
| (5) | N -> t P N |
| (6) | N -> ε     |
| (7) | P -> v K z |
| (8) | P -> x     |

[10]

(b) Calculating the first and follow sets is the first step of the algorithm for constructing the predictive parsing table. Provide the remaining steps. [10]

(c) Use this algorithm to construct the parsing table for the grammar above. You should format your parsing table as shown below, where the symbol $ denotes the end of input marker. [10]

| Non-terminal | Input Symbol | | | | | |
|--------------|---|---|---|---|---|---|
|              | x | y | z | t | v | $ |
| K            |   |   |   |   |   |   |
| L            |   |   |   |   |   |   |
| M            |   |   |   |   |   |   |
| N            |   |   |   |   |   |   |
| P            |   |   |   |   |   |   |

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

E2.15: Language Processors
Sample answers to exam questions 2010

Question 1

(a) [Bookwork]:
- Correctness of the generated code
- Conformity to the language specification; avoids temptations to implement a subset/superset of the language, which might result in a reduction of portability.
- Quality of the generated code with respect to size and speed.
- Speed of the language processor itself.
- User-friendliness, as evident in its quality of error reporting.

(b) [Bookwork]:

A grammar is defined as a quadruple (VT, VN, P, S) where

- VT is the set of terminal symbols, VN is the set of non-terminal symbols
- VT and VN have no symbols in common, and V is the union of VT and VN
- P is the set of production rules, each element of which consists of a pair (α, β) (where α is in V+ [one of more elements of V] and β is in V* [zero or more elements of V]), and a production has the form: α -> β
- S is a member of VN,and is known as the *start* or *sentence* symbol and is the starting point in the generation of any string in the language

 (c) [Bookwork and new computed example]

A *translation scheme* is a CF grammar in which program fragments are embedded in right side of the productions: *the semantic actions*. These are added in the parse tree and are executed when they are accessed during the tree traversal.

A suitable translation scheme for a infix to postfix notation is:

Expr -> Expr + Term {printf('+')}
Expr -> Expr – Term {printf('-')}
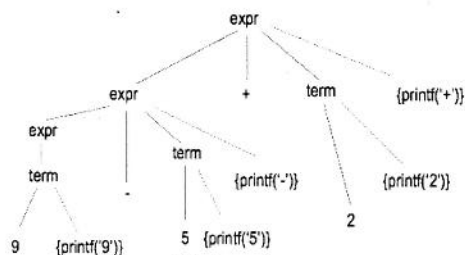Expr -> Expr * Term {printf('*')}
Expr -> Expr / Term {printf('/')}
Expr -> Term
Term-> 0 {printf('0')}
..
Term-> 9 {printf('9')}

Applying this to 9-5+2 gives the following parse tree:



(d) [Bookwork]:

LR parsing involves the use of a parsing table (containing *goto* and *action*
entries), and a stack. Given an input string w, the algorithm proceeds as
follows:

Set input pointer ip to the first symbol of w$
**Repeat**:
 Let s be the state on top of the stack, and a the symbol pointed by ip
 **if** action[s,a] = shift s' **then**
  **begin**
   Push a then s' on top of the stack
   Advance ip to the next input symbol

```
        end
else if action[s,a] = reduce A->β then begin
        Pop 2*length(β) items off the stack
        Let s' be the state now on top of the stack
        Push A then goto[s', A] on top of the stack
        Output the production A->β
        end
else if action[s,a]=accept then return
else error()
end
```

(e) [Bookwork and new computed example]:

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 For production rules of the form:

A -> B {C.inherited_attribute = f (B.attributes} C

the 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)}

Embedded actions can introduce shift-reduce conflicts!

    *blah : ABCD | ABCZ ;*
    *ABCD: 'A' 'B' 'C' 'D' ;*
    *ABCZ: 'A' 'B' 'C' 'Z' ;*

is fine, but

    *blah : ABCD | ABCZ ;*
    *ABCD: 'A' 'B' {printf("seen an A and a B"\n);} 'C' 'D' ;*
    *ABCZ: 'A' 'B' 'C' 'Z' ;*

isn't!

(f) [Bookwork]:

1. First determine the set of *leaders*, the first statements of basic blocks, using the following rules:
    a. The first statement is a leader
    b. Any statement that is the target of a conditional or unconditional goto is a leader
    c. Any statement that immediately follows a goto or conditional goto statement is a leader.
2. For each leader, its basic block consists of the leader and all statements up to but not including the next leader.

Once the basic blocks have been defined, a number of transformations can be applied to them to improve the quality of code without worrying about accidentally introducing flow of control errors.

(g) [Bookwork]

The grammar is not regular (type 3) since the second production rule is neither left-linear or right-linear. It is context-free grammar (type-2), since only a single non-terminal appears on the left-side of all production rules.

### 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{1} = {1} = A
Input set = {a, b, c, d, e}

ε-closure(move(A, a)) = ε-closure{2} = {2, 3, 4, 5, 6, 11} = B
ε-closure(move(A, b)) = ε-closure(move(A, c)) = ε-closure(move(A, d)) = ε-closure(move(A, e)) = {}

ε-closure{move(B,a)} = {}
ε-closure(move(B, b)) = ε-closure{7} = {3, 4, 5, 6, 7, 10, 11} = C
ε-closure(move(B, c)) = ε-closure{8} = {3, 4, 5, 6, 8, 10, 11} = D
ε-closure(move(B, d)) = ε-closure{9} = {3, 4, 5, 6, 9, 10, 11} = E

ε-closure{move(B,e)} = ε-closure{move(C,e)} = ε-closure{move(D,e)} = ε-closure{move(E,e)} = {12} = F
ε-closure(move(C, a)) = ε-closure(move(D, a)) = ε-closure(move(E, a)) = {}

ε-closure(move(C, b)) = ε-closure{7} = C
ε-closure(move(C, c)) = ε-closure{8} = D
ε-closure(move(C, d))  = ε-closure{9} = E

ε-closure(move(D, b)) = ε-closure{7} = C
ε-closure(move(D, c)) = ε-closure{8} = D
ε-closure(move(D, d))  = ε-closure{9} = E

ε-closure(move(E, b)) = ε-closure{7} = C
ε-closure(move(E, c)) = ε-closure{8} = D
ε-closure(move(E, d))  = ε-closure{9} = E

ε-closure(move(F, a)) = ε-closure(move(F, b)) = ε-closure(move(F, c)) = ε-closure(move(F, d)) =
    = ε-closure(move(F, e))  = {}

No more new states, we are done.
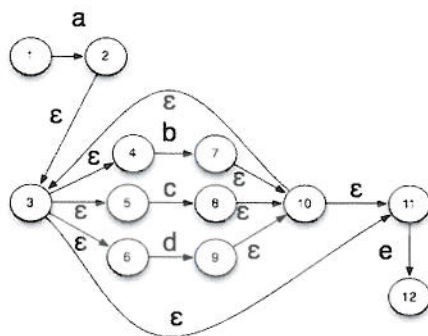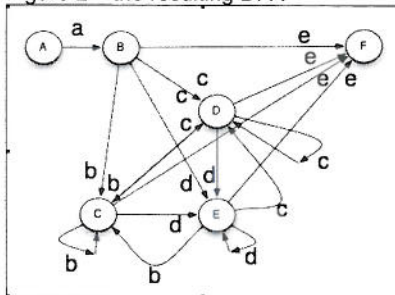
Figure 1 – the constructed NFA



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 – non-final: {A, B, C, D, E} and final: {F}. 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}, one with {B, C, D, E} since these go to the same states for the same symbols, and one with {F}. If we name the first set {A} as K, second set {B, C, D, E}) as L, and M = {F}, and draw all transitions, we get the following minimal DFA:
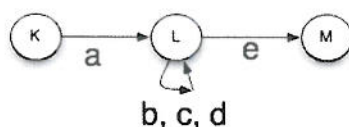
Question 3:

[New computed example]

(a) The set of 12 LR(0) items is:

I₀ :

G' -> .G
G -> .G –X
G -> .X
X -> .X*F
X -> .F
F -> .(G)
F -> .a

I₁:

G' -> G.
G -> G.-X

I₂:

G ->X.
X -> X.*F

I₃:

X->F.

I₄:

F -> (.G)
G -> .G-X
G -> .X
X -> .X*F
X -> .F
F -> .(G)
F -> .a

I₅:

F -> a.

I₆:

G -> G-.X
X-> .X*F
X->.F
F->.(G)
F->.a

I₇:

X-> X*.F
F->.(G)
F->.a

I₈:

F-> (G.)
G->G.-X

I₉:

G-> G-T.
X -> X.*F

I₁₀:

X->X*F.

I₁₁:

F->(G).

(b)

| State | Action | | | | | | Goto | | |
|---|---|---|---|---|---|---|---|---|---|
| | a | - | * | ( | ) | $ | G | X | 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) FIRST(A) = FIRST(C) = FIRST(E) = {e,a}; FIRST(B) = {b, ε};  FIRST(D) =  {d,ε}
    FOLLOW(A) = FOLLOW(B) = {c, $};  FOLLOW(C) = FOLLOW(D) = {b, c, $};
    FOLLOW(E) = {b, d, c, $}

(b) The algorithm for the construction of a predictive parsing table M for a given
     grammar is as follows:

         For each production rule A -> α do:
           • For each terminal x in FIRST(α), add A -> α to M[A, x]
           • If FIRST(α) contains ε, add A -> α to M[A, b] for each
             terminal b in FOLLOW(A)
           • If FIRST(α) contains ε, and FOLLOW(A) contains $, add
             A -> α to M[A, $]
           • Mark all undefined entries of M as "error".

(c) The resulting table for the grammar above is:

| Non-terminal | Input Symbol | | | | | |
|---|---|---|---|---|---|---|
| | x | y | z | t | v | $ |
| K | K -> M L | | | | K -> M L | |
| L | | L -> y M L | L -> ε | | | L -> ε |
| M | M -> P N | | | | M -> P N | |
| N | | N -> ε | N -> ε | N -> t P N | | N -> ε |
| P | P -> x | | | | P -> v K z | |