master - June 2008

E2.15: Language Processors

Sample answers to exam questions 2008

Question 1

(a) [Bookwork]:

Context Handling can include (among others)

(1) Type Checking (e.g. are operands compatible with the operator they appear with?)
(2) Checking the number of parameters in a function call against the number in the function's declaration
(3) Uniqueness checks

Reasons for incorporating an intermediate code generation stage during compilation can include:

(1) Making a clear distinction between the machine-independent and machine-dependent parts of the compiler.
(2) Front-end/back-end separation enables easy porting to new architectures/languages.
(3) Machine-independent optimisations can be applied to the intermediate code representation.

(b) [Bookwork]:

An LBA (a restricted class of Turing Machines) is defined as $(Q, \Sigma, \Gamma, \delta, q_0, B, F)$ where

- Q: a finite set of N states $q_0, q_1, \ldots, q_N$ of the finite control
- Σ: a finite input alphabet of symbols
- Γ: the complete set of tape symbols, $\Sigma \subseteq \Gamma$
- $\delta(q,X)$: the transition function between states. Given a state $q \in Q$, and a tape symbol $X \in \Gamma$, the function $\delta(q,X)$ returns a triplet (p,Y,D), where p is the new state in Q, $Y \in \Gamma$ is the symbol that replaces the current symbol on the tape, and D is either {left,right} indicating where will the head move next
- q_0: the start state
- B is the blank symbol ($B \in \Gamma$ but $B \notin \Sigma$) initially written in all tape cells except the ones holding the input
- F: the set of final states, $F \subseteq Q$

LBA differs from a PDA in the storage structures it involves (a tape rather than a stack), as well as in the sets of actions that can be performed at each step (read/write/moveleft/moreright in the LBA, vs push/pop in the PDA), and as a result differs in the types of languages it can parse.

(c)

Type-2 grammars have the restriction that only a single non-terminal can appear on the left side of a production, while type 3 grammars have the additional restriction that productions should either all be left-linear or all be right linear.

Example of a type 2 grammar production rule: A -> B x C
(single non-terminal on the left, but not only left or right linear)
Example of a type 3 grammar production rule: A -> x C
(single non-terminal on the left, AND right linear)

Type 2 grammars are "context-free" grammars, while type-3 grammars are regular grammars.

(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]:

The subset construction algorithm requires the definition of two functions:

- the ε-closure function takes as input a state and returns the set of states reachable from it based on one or more ε-transitions. This set will always include the state itself.
- The function move(state, char) which take as input a state and a character and returns the set of states reachable from it, with one transition using this character.

The algorithm then proceeds as follows:

(1) The start state of the NFA is the ε-closure of the start state of the NFA
(2) For the created state in (1), and for any created state in (2) do:
  For each possible input symbol:
    Apply the move function to the created state with the input symbol
    For the resulting set of states, apply the ε-closure function; this might or might not result in a new set of states.
(3) Continue (2) until no more new states are being created.
(4) The final states of the DFA are the ones containing any of the final states of the NFA

(f) [New computed example]:

An LL(1) grammar is a grammar whose parser requires only one token to be examined to select the production rule to be used for parsing at the next step. The grammar is not LL(1) since for rules 1 and 2, the first token is the same. Left factoring transforms the grammar to
X -> yXR | c
R -> a | b

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, 8, 13, 14 } = A

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

ε-closure(move(A, a)) = ε-closure{2} = {2, 1, 3, 4, 5, 6, 7, 8, 13, 14} = B
ε-closure(move(A, b)) = ε-closure{9} = {12, 5, 6, 7, 8, 9, 13, 14} = C
ε-closure(move(A, c)) = ε-closure{10} = {10, 12,5,6,7,8,13,14} = D
ε-closure(move(A, d)) = ε-closure{11} = {11, 12,5,6,7,8, 13, 14} = E
ε-closure(move(A, e)) = ε-closure{15} = {15} = F

ε-closure(move(B, a)) = ε-closure{2} = {2, 1, 3, 4, 5, 6, 7, 8, 13, 14} = B
ε-closure(move(B, b)) = ε-closure{9} = {12, 5, 6, 7, 8, 9, 13, 14} = C
ε-closure(move(B, c)) = ε-closure{10} = {10, 12,5,6,7,8,13,14} = D
ε-closure(move(B, d)) = ε-closure{11} = {11, 12,5,6,7,8, 13, 14} = E
ε-closure(move(B, e)) = ε-closure{15} = {15} = F

ε-closure(move(C, a)) = {}
ε-closure(move(C, b)) = ε-closure{9} = {12, 5, 6, 7, 8, 9, 13, 14} = C
ε-closure(move(C, c)) = ε-closure{10} = {10, 12,5,6,7,8,13,14} = D
ε-closure(move(C, d)) = ε-closure{11} = {11, 12,5,6,7,8, 13, 14} = E
ε-closure(move(C, e)) = ε-closure{15} = {15} = F

ε-closure(move(D, a)) = {}
ε-closure(move(D, b)) = ε-closure{9} = {12, 5, 6, 7, 8, 9, 13, 14} = C
ε-closure(move(D, c)) = ε-closure{10} = {10, 12,5,6,7,8,13,14} = D
ε-closure(move(D, d)) = ε-closure{11} = {11, 12,5,6,7,8, 13, 14} = E
ε-closure(move(D, e)) = ε-closure{15} = {15} = F

ε-closure(move(E, a)) = {}
ε-closure(move(E, b)) = ε-closure{9} = {12, 5, 6, 7, 8, 9, 13, 14} = C
ε-closure(move(E, c)) = ε-closure{10} = {10, 12,5,6,7,8,13,14} = D
ε-closure(move(E, d)) = ε-closure{11} = {11, 12,5,6,7,8, 13, 14} = E
ε-closure(move(E, e)) = ε-closure{15} = {15} = F

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

Figure 1 – the constructed NFA

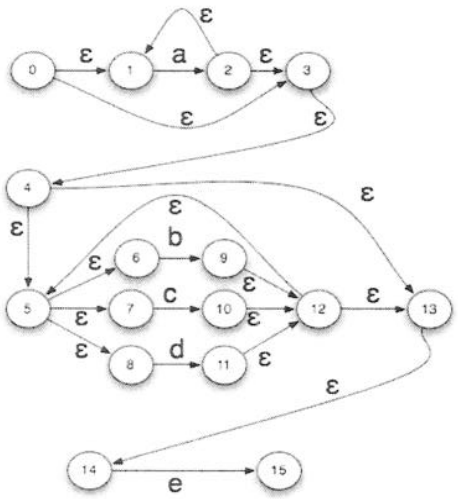
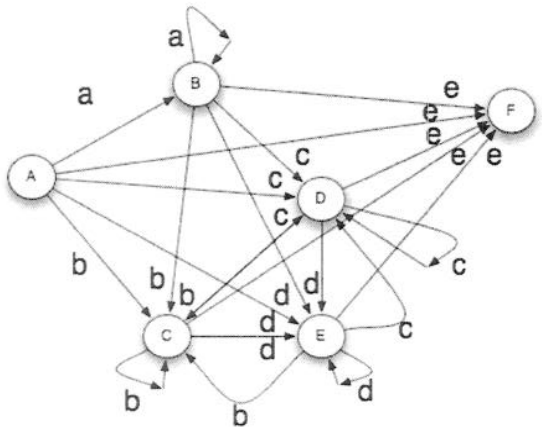


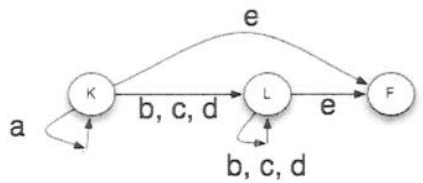
Figure 2 – the resulting DFA



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



Question 3:

(a) [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-> α.Bβ is in closure(I) and B->γ is a production, then add item B->.γ to I if its not already there. We apply this rule until no more new items can be added to closure(I)

The goto(I,x) function is defined as:
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-> αX.β] such that [A-> α.Xβ] is in I.

(b) [Bookwork]

Assuming that the start symbol of the grammar has been denoted as S, we start with
C = {closure{[S' -> .S]}) and then
Repeat
For each set of items I in C and each grammar symbol X such as Goto(I,X) is not empty and not in C do: add goto(I, X) to C
Until no more sets of items can be added to C.

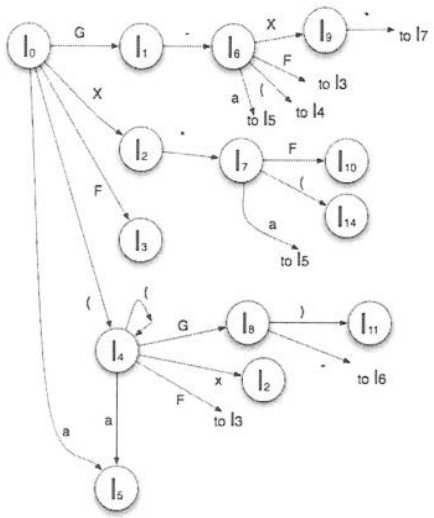
(c) [New computed Example]

The canonical set of LR(0) items

I_0 :	I_1:	I_2:	I_3:	I_4:	I_5:
G' -> .G G -> .G –X G -> .X X -> .X*F X -> .F F -> .(G) F -> .a	G' -> G. G -> G.-X	G ->X. X -> X.*F	X->F.	F -> (.G) G -> .G-X G -> .X X -> .X*F X -> .F F -> .(G) F -> .a	F -> a.

I_6:	I_7:	I_8:	I_9:	I_{10}:	I_{11}:
G -> G-.X X-> .X*F X->.F F->.(G) F->.a	X-> X*.F F->.(G) F->.a	F-> (G.) G->G.-X	G-> G-T. X -> X.*F	X->X*F.	F->(G).

The DFA for recognizing viable prefixes for the grammar:



Question 4:

(a) [New Computed Example]:

FIRST(G)=FIRST(T) = FIRST(F) = {(,a}; FIRST(G') = {-, ε}; FIRST(T') = {*,ε}

FOLLOW(G) = FOLLOW{G') = {), $}; FOLLOW(T) = FOLLOW(T') = {-,), $};
FOLLOW(F) = {-, *,), $}

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

Non-terminal	Input Symbol					
	a	-	*	(	)	$
G	G -> TG'	Error	*Error*	G->TG'	Error	*Error*
G'	Error	G' -> -TG'	Error	Error	G'-> ε	G'-> ε
T	T -> FT'	Error	Error	T -> FT'	Error	Error
T'	Error	T' -> ε	T' -> *FT'	Error	T' -> ε	T' -> ε
F	*F -> a*	*Error*	Error	F -> (G)	Error	Error