QUIZ 2

DATE: 13th FEBRUARY 2002 **Maximum Marks: 40**

1. Prove that the following grammars do not belong to the class mentioned alongside by showing any parsing table entry with conflicts in each case.

   (a) **not SLR(1)**
   $S \rightarrow X$ `@`
   $X \rightarrow M\ a \mid b\ M\ c \mid d\ c \mid b\ d\ a$
   $M \rightarrow d$
   Let the state s represent the viable prefix d. $action[s,c]$ will contain shift to a state representing the viable prefix dc and reduce using $M \rightarrow d$

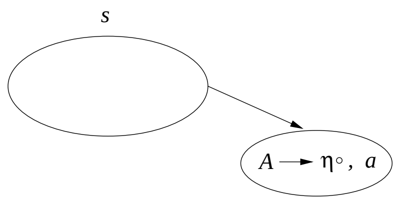
   (b) **not LALR(1)**
   $S \rightarrow A\ a \mid C\ A\ c \mid B\ c \mid C\ B\ a$
   $A \rightarrow d$
   $B \rightarrow d$
   $C \rightarrow c\ c$
   Let the state s represent the merger of the viable prefixes d and ccd. Then $action[s,a]$ or $action[s,c]$ will contain reduces by the productions $A \rightarrow d$ and $B \rightarrow d$.

   (c) **not LR(1)**
   $S \rightarrow$ `id := ` $ae \mid$ `id := ` be
   $be \rightarrow be$ `and id` $\mid be$ `or id` $\mid ae$ `== id` $\mid$ `id`
   $ae \rightarrow ae$ `+ id` $\mid$ `id` (3 ∗ 5 = 15 marks)
   Let the state s represent the merger of the viable prefixes d and ccd. Then $action[s,a]$ or $action[s,c]$ will contain reduces by the productions $A \rightarrow d$ and $B \rightarrow d$.

2. Explain with an example why the following statement is incorrect:
   If there is a production $A \rightarrow \alpha\ B$, then $FOLLOW(A) = FOLLOW(B)$. (3 marks)

   Answer: If the grammar containing the above production also contained an useful production $C \rightarrow \alpha\ B\ a$, then $a \in FOLLOW(B)$. However, there might not be any production which will put a in $FOLLOW(A)$.

3. Using the figure shown below as a hint, answer the following question. Is it possible for a LR parser to be in a situation where it has to consult $GOTO\ [s,A]$ for some state s and non-terminal A, and $GOTO\ [s,A]$ does not contain any entry? Clearly explain your answer. (5 marks)



Answer: The parser is in a situation where it has to consult $GOTO[s,A]$ because of the following:

1. It is in a state which has an item of the form $[A \rightarrow \eta\circ, a]$, and a is the next terminal symbol.

2. There is a reduction using $A \rightarrow \eta$. As a result of this, the parser will go to a state $[B \rightarrow \alpha \circ A\beta, b]$ and $[A \rightarrow \circ\eta, a]$

3. At this state the goto table is consulted. It can be seen clearly, the GOTO table will contain an entry to a state which has the item $[B \rightarrow \alpha A \circ \beta, b]$

4. Consider the grammar shown below:

   $P \rightarrow D;\ S$
   $D \rightarrow$ `label` *id* | `proc id;` *D; S* `endproc` | *D1*; *D2*
   $S \rightarrow$ `a` | `go to` *id* | *id*: `a` | *id*: `go to` *id* | *S1; S2*

   The scope rules regarding labels and gotos is as follows:

   (a) A label must be both declared and defined in the same procedure.

   (b) A use of a label (through a goto) is bound to a definition in the same way as that of an ordinary variable. Every use of a label should be bound to some definition.

   We want to write syntax directed definitions to achieve two things:

   (a) A violation of the above rules should be flagged as an error at soon as possible.

   (b) A goto statement *S* should have an attribute called *where*. This should evaluate to the name of the procedure containing the definition to which the goto will be targeted.

   Here is an example program produced by the grammar along with the output we want to achieve.:

```
label x1;

proc P;
  label x2;  label x3;

  proc Q;
    label x1; label x4

    x3: a        /* error, x3 not declared in this procedure */
    x4: goto x1  S.where = Q, S.visible ={(x1,Q), (x2,P), (x3,P),
                 (x4, Q)}, S.dnd = {x1, x4}
    x1: goto x5  /* error, x5 not in scope*/
  end proc /*Q*/

  x2: goto x4    /* error, x4 not in scope*/
```

```
  goto x1         S.where = main, S.visible ={(x1,main), (x2,P),
                  (x3,P)}, S.dnd = {x3}
end proc /*P*/    /* error, x3 declared but not defined in P*/
x1: a
```

You must use the following attributes:

S.visible - the labels visible just before S and the procedure names containing them.

S.dnd - the labels which have been declared in the procedure containing

S but not yet defined at a point just prior to S.

You may use any other attribute and any function that you wish. However, clearly specify what they mean. *Use attributes only, do not use any global variables.* (17 marks)

Answer: Apart from those mentioned above, we need the following attributes:

D.visible — Similar to *S.visible*
D.decl — List of all the declarations in D.
D.procname — Name of the procedure immediately surrounding D.
S.dnd_after —the labels which have been declared in the procedure containing S but not yet defined at a point just after S.

Now the functions that we shall use. Firstly, we shall be making extensive use of the function *map* defined as:

$map\ f\ Nil = Nil$
$map\ f\ (Cons\ x\ xs) = Cons\ (f\ x)\ (map\ f\ xs)$

Example: $map\ square\ [1, 2, 3] = [1, 4, 9]$

Given an element x and a list a, *insertinlist* makes a list of pairs with x and every element of a.

$insert\ x\ a = (x, a)$
$insertinlist\ x\ l = map\ (insert\ x)\ l$

Example: $insertlist\ main\ [x_1, x_2, x_3] = [(main, x_1), (main, x_2), (main, x_3)]$

shadow implements holes in scope.
$shadow\ [(main, x_1), (P, x_2), (P, x_3)]\ [(Q, x_4), (Q, x_1)] = [(P, x_2), (P, x_3), (Q, x_4), (Q, x_1)]$.
The declaration of x_1 in Q covers the declaration of x_1 in *main*. The order of elements in the result list may not be exactly this, but it does not matter.

$shadowelement\ (a, b)\ l = if\ (a \notin (map\ first\ l))\ then\ (Cons\ (a, b)\ l)\ else\ l$
$shadow\ Nil\ l_1 = l_1$
$shadow(Cons\ (a, b)\ l)\ l_1 = shadow\ l\ (shadowelement\ (a, b)\ l_1)$

Now we are in a position to give our L-attributed definition:

$P \rightarrow \quad \{\ D.visible = \phi;\ D.procname = main\}$

$\quad D$

$\quad\quad \{S.dnd = D.decl;\ S.visible = insertinlist(main,\ D.decl)\}$

$\quad S$

$\quad\quad \{if\ (S.dnd\_after \neq \phi)\ then\ print(error);\ \}$

$D \rightarrow \quad \{D1.visible = D.visible;\ D1.procname = D.procname;\ \}$

$\quad D1;$

$\quad\quad \{D2.visible = shadow(D1.visible,\ insertlist(D.procname,\ D2.decl));$

$\quad\quad D2.procname = D.procname;\}$

$\quad D2$

$\quad\quad \{D.decl = D1.decl \cup D2.decl\}$

$D \rightarrow$ `label id` $\{D.decl = id.name\}$

$D \rightarrow$ `proc id;`

$\quad\quad \{\ D1.procname = id.name;\ D1.visible = D.visible;\}$

$\quad D1;$

$\quad\quad \{\ S.visible = shadow(D1.visible,\ insertinlist(id.name,\ D1.decl);$

$\quad\quad S.dnd = D1.decl\ \}$

$\quad S$

$\quad\quad \{\ if\ (S.dnd\_after \neq \phi)\ then\ print\ (error);\ D.decl = \phi;\ \}$

$S \rightarrow$ `id:  a`

$\quad\quad \{\ if\ id.name \notin S.dnd\ then\ print(error);\ \}$

$\quad\quad S.dnd\_after = S.dnd - \{id.name\}\ \}$

$S \rightarrow$ `id :  goto id1`

$\quad\quad \{if\ id.name \notin S.dnd\ then\ print(error);$

$\quad\quad if\ id1.name \notin (map\ first\ S.visible)\ then\ print(error)$

$\quad\quad else\ S.where = searchproc(id1.name,\ S.visible);$

$\quad\quad S.dnd\_after = S.dnd - \{id.name\}\ \}$

$S \rightarrow \quad \{S1.visible = S.visible;\ S1.dnd = S.dnd;\}$

$\quad S1;$

$\quad\quad \{\ S2.visible = S.visible;\ S2.dnd = S1.dnd\_after\}$

$\quad S2$

$\quad\quad \{\ S.dnd\_after = S2.dnd\_after\ \}$