CS324 – LANGUAGE PROCESSORS

MID-SEMESTER EXAMINATION

DATE: 22nd FEBRUARY 2002 — **Weightage: 15%**
Maximum Marks: 30 — **Closed Notes/Books Exam**

---

1. Consider a variant of FORTRAN defined using the grammar below:

   $sts \rightarrow sts\ \ st; \mid st;$
   $st \rightarrow assgn \mid do \mid if \mid labassgn \mid goto$
   $term \rightarrow$ `id` $\mid$ `num` $\mid$ `real` $\mid term + term$
   $if \rightarrow$ `IF` $(cond)\ goto$
   $do \rightarrow$ `DO num id = num , num`
   $assgn \rightarrow$ `id` $= term \mid$ `id ( id )` $= term$
   $labassgn \rightarrow$ `num` $assgn$
   $cond \rightarrow$ `id` $\mid term$ `.GT.` $term$
   $goto \rightarrow$ `GOTO num`

   `id` is defined in the usual way. A `num` is a natural number, and a `real` is, in general, two `nums` separated by a decimal point. Only one of the two `nums` may be absent.

   Note that in the earlier versions of FORTRAN, white spaces did not play any role. Thus a preprocessing stage would strip all the white spaces from the grammar and pass the result to the scanner. Further, keywords can be used as identifiers. Thus:

   (a) `DO10I=1,25;` and `DO10I=1.25;` are both valid statements. The first is a *do* and the second is an *assgn*.

   (b) `IF(A)GOTO30;` and `IF(A)=3.4;` are both valid statements, the first is an *if*, the second an *assgn*.

   Write a JFlex script just for detecting tokens (no declarations, no actions) in the language described above. The script should use the lookahead operator and not use states. (7 Marks)

   Answer:

```
%%
%class Lexer
%standalone
%line
%column

/* Macro definitions */

LineTerminator = \r|\n|\r\n
WhiteSpace     = {LineTerminator} | [ \t\f]
identifier = [:jletter:] [:jletterdigit:]*
num  = 0 | [1-9][0-9]*
```

```
real = {num}\.{num}
condtokens = ( {WhiteSpace} | {identifier} | {num} | {real} | "+" | ".GT." )+
%%

/* keywords */

"DO"/({num}{identifier}"="{num}",")  { System.out.println("Keyword DO\n"); }
"IF"/("("{condtokens}")""GOTO")      { System.out.println("Keyword IF\n"); }
"GOTO"/{num}                         { System.out.println("Keyword GOTO\n"); }

/* identifiers */

{identifier}                         { System.out.println("identifier\n"); }

/* literals */

{num}                                { System.out.println("num\n"); }
{real}                               { System.out.println("real\n"); }

/*operators*/

".GT."                               { System.out.println("Operator .GT.\n"); }
"+"                                  { System.out.println("Operator +\n"); }
"="                                  { System.out.println("Operator =\n"); }

/* delimiters */

"("                                  { System.out.println("Delimiter (\n"); }
")"                                  { System.out.println("Delimiter )\n"); }
";"                                  { System.out.println("Delimiter ;\n"); }
","                                  { System.out.println("Delimiter ,\n"); }

/* whitespace */

{WhiteSpace}                         { /* Ignore */}

/* error fallback */
.|\n                                 { throw new Error("Illegal character <"+
                                       yytext()+">"); }
```

There is a small problem with this JFlex script, indeed with the question itself. It recognises an identifier like `GOTO10` as the keyword `GOTO` followed by a `num` and not as an identifier.

2. For the grammar shown below, answer the following questions with explanations:

$$S \rightarrow \texttt{id} \; [\; E \;] \; := E$$
$$E \rightarrow E + T \mid T$$

$T \rightarrow T * F \mid F$
$F \rightarrow (\ E\ ) \mid \texttt{id}$

(a) In the automaton of the SLR(1) parser for this grammar, there is a state with a self loop? Identify the state through its shortest viable prefix.

Answer: The state

$F \rightarrow (\ \circ E\ )$
$E \rightarrow \circ E + T \mid \circ T$
$T \rightarrow \circ T * F \mid \circ F$
$F \rightarrow \circ (\ E\ ) \mid \circ \texttt{id}$

has a self loop around it on the symbol `(`. The viable prefixes corresponding to this state are `id[(`, `id[((`, `id[(((` and so on. The shortest viable prefix is `id[(`.

(b) Is the behavior of the SLR(1) parser and the LR(1) parser identical for *all strings*? No explanation is required if answer is yes. However, if the 'yes' answer is wrong, it will attract penalty.

Answer: No, the behaviour will be different on an erroneous string. Consider, as an example, the string `id[id$`. After seeing `id[`, the parser would be in a state consisting of the following set of items:

$S \rightarrow \texttt{id}\ [\ \circ E\ ] := E$
$E \rightarrow \circ E + T \mid \circ T$
$T \rightarrow \circ T * F \mid \circ F$
$F \rightarrow \circ (\ E\ ) \mid \circ \texttt{id}$

Now, on seeing a `id`, the SLR parser will go to a state consisting of the set of item $F \rightarrow \texttt{id} \circ$. Here, on seeing a `$`, it will reduce thrice (work out the details) and go to the state consisting of the items $S \rightarrow \texttt{id}\ [\ E \circ\ ] = E$ and $E \rightarrow E \circ + T$. In this state the error will be detected.

After seeing `id[id`, the LR parser, on the other hand, will go to the state $F \rightarrow \texttt{id} \circ,$ `]`, where `]` is the lookahead symbol. Therefore, right at this point, the LR parser will give an error.

(c) Is the behavior of the LALR(1) parser and the LR(1) parser identical for *all strings*? No explanation is required if answer is yes, but a wrong answer will attract penalty.

Answer: The LALR parser will behave just like the SLR parser and therefore would also be different from the LR parser. After seeing `id[id`, it will go to the state $F \rightarrow \texttt{id} \circ,$ `]` `$`, the two lookahead symbols obtained as a result of merging two LR states. It will go through three round of reductions, each time transiting through a state which has both `]` and `$` as lookaheads. The error will be finally caught in a state which has $S \rightarrow \texttt{id}\ [\ E \circ\ ] = E$ and $E \rightarrow E \circ + T$ as core.

3. Consider the following grammar:

$S \rightarrow A_1\ b_1 \mid A_2\ b_2 \mid A_3\ b_3$
$A_1 \rightarrow a_2\ A_1 \mid a_3\ A_1 \mid a_2 \mid a_3$

$A_2 \rightarrow a_1\ A_2 \mid a_3\ A_2 \mid a_1 \mid a_3$
$A_3 \rightarrow a_1\ A_3 \mid a_2\ A_3 \mid a_1 \mid a_2$

Now answer the following questions with explanations. No credit if the explanation is wrong.

(a) Is the grammar ambiguous?
Answer: The grammar is not ambiguous. The strings generated from this grammar can be grouped into one of $(a_2|a_3)^*b_1$, $(a_1|a_3)^*b_2$ or $(a_1|a_2)^*b_3$.

Consider the first group for example. Clearly any string in this group has an unambiguous parse if any string generated from $(a_2|a_3)^*$ can be unambiguously reduced to A_1. This *is* the case since the last symbol in the string can be reduced to A_1, and then successively either $a_2\ A_1$ or $a_3\ A_1$ can be reduced to A_1. This is the only sequence of reductions possible.

(b) Is the grammar $LL(n)$ for some value of n?
Answer: This is not $LL(n)$ for any n. Given a string $a_2^{n+1}b_1$, the LL(n) parser will not be able to decide whether to use the production $S \rightarrow A_1\ b_1$ or $S \rightarrow A_3\ b_3$ for expansion.

(c) Is the grammar $SLR(n)$ for some value of n?
Answer: Yes, the grammar is $SLR(1)$. Consider the sequence of actions that the parser goes through when it parses a string of the form $(a_2|a_3)^*b_1$.

  i. Since the only reductions are on b_i and \$, the parser will go through a sequence of shifts till it sees b_1. Clearly there are no conflicting entries in the states that the parser passes through during these shifts.
  ii. When the parser sees b_1, it goes into a state which either contains $A_1 \rightarrow a_2 \circ$ or $A_1 \rightarrow a_3 \circ$. Moreover, since this state contains no item with a $\circ$ before b_1, there is no conflict in this state.
  iii. Following this, the parser goes through a series of reductions. During this the states it traverses through contain items either of the form $A_1 \rightarrow a_2\ A_1 \circ$ or $A_1 \rightarrow a_3\ A_1 \circ$. Once again, these states do not contain items with a $\circ$ before b_1; therefore there are no conflicts in such states.
  iv. Finally the parser goes to a state in which the only item is $S \rightarrow A_1\ b_1 \circ$.

Using similar arguments it can be shown that any string of the grammar goes only through conflict-free states. This shows that all states are conflict-free (strings not in the grammar need not be considered. Why?).

(d) Is the grammar $LALR(n)$ for some value of n?
Answer: Yes, since any $SLR(n)$ grammar is also a $LALR(n)$ grammar.

4. Given below is the grammar that generates binary numbers (including those with a fractional part):

$S \rightarrow L_1\ .\ L_2 \mid L$
$L \rightarrow L_1\ B \mid B$
$B \rightarrow 0 \mid 1$

Write a L-attributed definition to calculate the decimal value of S in an attribute called *val*. Clearly explain the meaning of the attributes and functions that you will be using. (5 Marks)

Answer: I am giving the solution of a harder problem in which we are required to use synthesized attributes only. Let L have an attribute evaluating to a pair of values. These are the values of L assuming that its position is before and after the decimal point.

The S-attributed definition is:

$S \rightarrow L_1.L_2$ $\quad \{S.val = first(L_1.pair) + second(L_2.pair)\}$

$S \rightarrow L$ $\quad \{S.val = first(L_1.pair)\}$

$L \rightarrow L_1 B$

$$\{L.pair = (first(L_1.pair) * 2 + B.val, second(L_1.pair) + B.val^{-(L_1.length+1)}); \; L.length = L_1.length + 1\}$$

$L \rightarrow B$ $\quad \{L.pair = (B.val, B.val^{-2});\ L.length = 1\}$

$B \rightarrow 0$ $\quad \{B.val = 0\}$

If we are allowed inherited attributes, then we can pass the position of L relative to the decimal point (before or after) to the second and third productions.

5. Consider a grammar to generate C-style expressions.

$$\begin{aligned}
&S \rightarrow E \\
&E \rightarrow E_1 = E_2 \\
&E \rightarrow E_1 + E_2 \\
&E \rightarrow \texttt{id}\ [\ E_1\ ] \\
&E \rightarrow (\ E_1\ ) \\
&E \rightarrow \texttt{id}
\end{aligned}$$

I am assuming that you know what it means to say that an expression has a *l-value*. An assignment is `valid` if the expression to the left of `=` has a l-value. And an expression is valid if all assignments within it are valid. Thus `a = b = c` is valid, only if it is parsed as `a = (b = c)` and not as `(a = b) = c`.

Write a L-attributed definition to determine an attribute *valid* of S which will evaluate to true only when the sentence generated from it is valid. You might like to do this in terms of an attribute *has_l* of E which determines whether the expression E has a l-value. Use any other attributes if necessary, however explain what they mean.

Answer: The syntax directed translation scheme is:

$S \rightarrow E$	$\{S.valid = E.valid\}$
$E \rightarrow E_1 = E_2$	$\{E.valid = E_1.has\_l \wedge E_1.valid \wedge E_2.valid;\ E.has\_l = false\}$
$E \rightarrow E_1 + E_2$	$\{E.valid = E_1.valid \wedge E_2.valid;\ \ E.has\_l = false\ \}$
$E \rightarrow \texttt{id}[E_1]$	$\{E.valid = E_1.valid;\ E.has\_l = true\}$
$E \rightarrow (E)$	$\{E.valid = E_1.valid;\ E.has\_l = E_1.has\_l\}$
$E \rightarrow \texttt{id}$	$\{E.valid = true;\ E.has\_l = true\}$

(5 Marks)