CZ3007 Compiler Techniques

Tutorial 2 (Syntax Analysis)
(to be covered in 2.5 tutorials)

1. Consider each of the following CFGs and the given string for each grammar

i) p_1 $S \rightarrow S\ S\ +$
p_2 $\quad |\ S\ S\ *$
p_3 $\quad |\ a$
and the string "aa+a*"

(ii) p_1 $S \rightarrow S\ (\ S\ )\ S$
p_2 $\quad |\ \lambda$
and the string "(()())"

a) Give a leftmost derivation for the string and the parse tree;
b) Give a rightmost derivation for the string and the parse tree;
c) Is the grammar ambiguous or unambiguous? Justify your answer.

2. Design CFGs for the following language:

a) The set of all strings of 0s and 1s such that every 0 is immediately followed by at least one 1.
b) The set of all strings of 0s and 1s that are *palindromes*; that is, the string reads the same *backward* as *forward*.
c) The set of all strings of 0s and 1s with an equal number of 0s and 1s.

3. Consider the following LL(1) grammar.

p_1	Expr	$\rightarrow$	- Expr
p_2		|	(Expr)
p_3		|	Var ExprTail
p_4	ExprTail	$\rightarrow$	- Expr
p_5		|	λ
p_6	Var	$\rightarrow$	id VarTail
p_7	VarTail	$\rightarrow$	(Expr)
p_8		|	λ

The values of symbolDerivesEmpty[X] for $X \in N$ and ruleDerivesEmpty[p_i] for i = 1, 2, …, 8 are as follows.

symbolDerivesEmpty

Expr	ExprTail	Var	VarTail
F	T	F	T

ruleDerivesEmpty

p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8
F	F	F	F	T	F	F	T

a) Compute **first**(RHS of p_i) for i = 1, 2, …, 8.
b) Compute **follow**(X) for each non-terminal X.
c) Then compute **predict**(p_i) for i = 1, 2, …, 8.

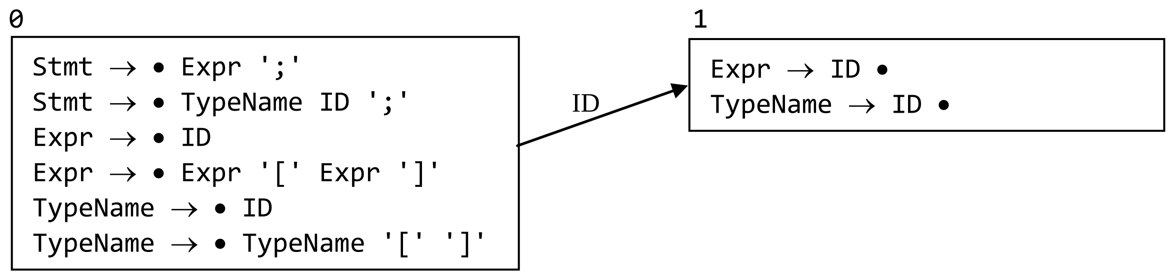
4. Transform the following grammar into LL(1) form by eliminating common prefixes and left recursions.

p_{1}	DeclList	$\rightarrow$	DeclList ; Decl
p_{2}		|	Decl
p_{3}	Decl	$\rightarrow$	IdList : Type
p_{4}	IdList	$\rightarrow$	IdList , id
p_{5}		|	id
p_{6}	Type	$\rightarrow$	ScalarType
p_{7}		|	array (ScalarTypeList) of Type
p_{8}	ScalarType	$\rightarrow$	id
p_{9}		|	Bound . . Bound
p_{10}	Bound	$\rightarrow$	Sign intconstant
p_{11}		|	id
p_{12}	Sign	$\rightarrow$	+
p_{13}		|	-
p_{14}		|	λ
p_{15}	ScalarTypeList	$\rightarrow$	ScalarTypeList , ScalarType
p_{16}		|	ScalarType

5. In Java, if a statement starts with an identifier followed by a left bracket, this could either be the start of an assignment ("foo[i] = null;"), or it could be the start of a type declaration ("Object[] foo;"). In the former case, "foo" is, of course, the name of a variable, whereas in the latter case "Object" is a type name. Consider the following grammar:

```
Stmt → Expr ';'
      | TypeName ID ';'
Expr → ID
      | Expr '[' Expr ']'
TypeName → ID
      | TypeName '[' ']'
```

Part of the LR(0) automaton for the grammar will be like the following:



Here, we have a reduce-reduce conflict. Even when we build a LALR(1) automaton for the grammar, we will still have a reduce-reduce conflict on the input of '['. Confirm this by constructing a partial propagation graph to show the conflict.

This conflict can be eliminated by rewriting the rules for TypeName as follows:

```
TypeName  →  ID
          | ArrayType
ArrayType →  ID '[' ']'
          | ArrayType '[' ']'
```

Construct a partial propagation graph to show that the reduce-reduce conflict is gone. However, we now have a shift-reduce conflict on '['! In this case, choosing shift over reduce is actually a bad idea, since it will make it impossible to parse the valid expression statement 'foo[bar];'.

Instead, we need to unfold the rules for Expr as we did for TypeName:

```
Expr →  ID
     | IndexExpr
IndexExpr →  ID '[' Expr ']'
     | IndexExpr '[' Expr ']'
```

And now both conflicts are gone. Confirm this by a partial propagation graph similar to the earlier ones.

<u>Questions not to be covered in Tutorial</u>:

1. Consider the following LR(0) grammar we discussed in the lecture

   Start → E $
   E → plus E E
   | num

and the parse table we built on slide 63 and the LR parsing engine on slide 47 of the lecture notes, show the contents of the stack during the different states of parsing for the following input:

(i) 3; (ii) + 5 3; (iii) ++ 2 8 5

2. Consider the following language.

```
Stmt →  IF LPAREN Expr RPAREN Stmt
     |  IF LPAREN Expr RPAREN Stmt ELSE Stmt
     |  SEMICOLON

Expr → ID
```

   Show the construction of the LR(0) parse table of this grammar up to the state where there is a shift-reduce conflict.