CS333 : Compiler Design

TUTORIAL 5 : LR Parsing

P1. We have seen that left recursive rules in a context free grammar are not suitable for top down parsing. Does bottom up parsing have an analogous problem ? Examine any bottom up parser from the LR family with respect to recursive rules. Find out what difference exists, if any, when for the same language, first left recursive and then a right recursive grammar is used with your chosen bottom up parser.

P2. The stack contents and the input symbol at some point during parsing by a shift reduce parser is shown below with the nonterminal C on top of the stack. The terminals on the stack are {a. b, c, d} and the nonterminals are {A, B, D, C}

STACK : a A b B c D d C

Which among the following statements, S1 to S4, are true and why ?

S1 : Handles are { C, d C, D d C}

S2 : Handles are {D, D d, D d c}

S3 : Strings {A b B, c D d C} are not handles

S4 : The longest handle is of length 8.

P3. Examine the blank entries in the goto part of a LR parsing table (SLR(1), LALR(1) or LR(1)) are error entries in the sense that the blank entries are error in the Action part.

P4. Consider the following context free grammar.

S → X
X → Ma | bMc | dc | bda
M → d

(a) Determine whether the grammar as given is LR(0). A LR(0) grammar is one for which the corresponding LR(0) parsing table has no conflicts; a reduce action in a state of an LR(0) parser is performed on all terminals. Report all the conflicts if your answer is in the negative. T. Recall that in SLR(1), a reduce action, A → α, is performed for all terminals in FOLLOW(A).
(b) Determine whether the grammar as given is SLR(1). Report all the conflicts if your answer is in the negative.
(c) Examine whether the grammar is LR(1) or LALR(1).

P5 (a) Identify which of the grammars given below, is LR(0) and/or SLR(1) and/or LALR(1) / LR(1) and why ?

(i)
S → A B c
A → a | ε
B → b | ε

(ii)
S → A B B A
A → a | ε
B → b | ε

(iii)
E → -E | (E) | V R
V → id T
R → - E | ε
T → (E) | ε

(b) Can these grammars be transformed to an equivalent LR(0) and/or SLR(1) ?

(c) Can these grammars be transformed to an equivalent LR(1) and LALR(1) ?

P6. Consider the grammar given below for declaration processing.

P → D
D → **id** : T
D → **procedure id** ; D ; B
T → **integer** | **real**
B → **body**

(a) The grammar written above was supposed to parse program fragments of the form given below. Argue whether grammar given will serve its intended purpose, in case your answer is no, rewrite the grammar to achieve the desired objective.

```
procedure p1;
   a : real;
   procedure p2;
      b : integer;
      procedure p3;
         b : real;
         body;  // of p3
      procedure p4;
         a : integer;
      body; //  of p4
   body; //  of p2
body;  // of p1
```

(b) Determine whether the grammar as given, or the one rewritten by you in part (a), will admit a SLR(1) parser by constructing the automaton or otherwise.

(c) Construct the parse tree for the program fragment of part (a) using your grammar and a SLR(1) parser.

(d) Construct LR(1) and LALR(1) parsing table for the grammar written by you in part (b). Construct the parse tree for the program fragment of part (a) using these tables.

P7. We would like to process program fragments, as given below only for the purpose of parsing.

```
integer  a [10, 20];
integer  b [10, 20];
integer i, j;
i = 10;
j = 16;
b[i+1, j+2] = a [i-1, j+1] + a [i , j – 2] + 25;
```

(a) Make appropriate changes to the grammar given below for array references, if so required, so that code fragment, as given above, can be generated along with parsing. Your grammar should be able to (i) generate declarations of scalars and arrays as denoted by the first 3 lines above, and (ii) also generate assignments involving arrays and scalars, as given in the last 3 lines of the sample code above. Assume that an array is stored in row-major representation , if such information is necessary.

S → L = E
E → L
L → elist] | **id** | **num**
elist → elist , E | **id** [E

(b) Determine whether the grammar as given, or the one rewritten by you in part (a), will admit a SLR(1) / LALR (1) / LR(1) ser by constructing the automaton or otherwise. We need the smallest parser of the family that works for the grammar.

(c) Construct the parse tree for the program fragment of part (a) using your grammar and the parser.

(d) Construct LR(1) and LALR(1) parsing tables for the grammar written by you in part (b). Construct an input that is generated using all the rules and show its parse tree using the constructed tables.

P8. For the grammar shown below, answer the following questions.

S → id [E] : = E

E → E + T | E

T → T * F | F

F → (E) | id

Construct any one complete automaton from the 3 parsers, and show the kernel items of each state. Identify all the states of this automation with a self loop and write the set of all viable prefixes for this state. Justify the validity of all the LR(0) items contained in such states, and report your findings.

P9. Consider the following context free grammar which can generate certain assignment statements involving array references and arithmetic operators assuming the usual properties of operators.

S → L := E
T → T * F | F
elist → elist , E | **id** [E
E → L | E + T
L → elist] | **id**
F → **id**

(a) Construct a SLR(1) parser for the grammar given above, rewriting the same, if required, without changing the underlying language.

(b) Construct LR(1) and LALR(1) parsing tables for the grammar of part (a).

P10. Consider the grammar given below for generating program fragments with control flow constructs . The term **relop** stands for relational operators {<, <=, >, >=, ==, !=}.

S → **if** B **then** S | **if** B **then** S **else** S | **while** B **do** S | **begin** SList **end** | L **:=** E

SList → SList **;** S | S

B → B **or** B | B **and** B | **not** B | **(** B **)** | **true** | **false** | E **relop** E | E

E → E + E | **id**

(a) The grammar written above was supposed to parse program fragments of the form given below. Argue whether grammar given will serve its intended purpose, in case your answer is no, rewrite the grammar to achieve the desired objective.

```
if ( a < b) then
   begin
     a := b + c;
     while ( a <= b) do begin  a := c + d end;
     a := b
   end
else    begin  c := c + d end  ;
```

(b) Determine whether your grammar will lead to a conflict free SLR(1) Parsing table. If the answer is in the negative, report the conflicts.

(c) Construct the parse tree for the program fragment of part (a) using your grammar and a SLR(1) parser.

(d) Construct LR(1) and LALR(1) parsing table for the grammar written by you in part (b). Construct the parse tree for the program fragment of part (a) using these tables.

P11. Construct a grammar

(a) that is SLR(1) but not LL(1)
(b) that is LL(1) but not SLR(1)
(c) that is LALR(1) but not SLR(1)
(d) that is LR(1) but not LALR(1)

P12. Identify the following grammars as LL(1), LR(0), SLR(1), LALR(1) or LR(1)

(a)
S → d M N c | N | L
N → d b | c d
L → d M d M
M → b

(b)
S → d B | A d | B c
A → C c
B → a
C → d a

P13. Consider the SLR(1) automaton for the pointer grammar G, given in the notes

0:	S' → S	1,2:	S → L = R | R
3,4:	L → * R | id	5:	R → L

(a) Write 5 distinct strings from $(N \cup T)^*$ that are not viable prefixes for G
(b) Consider a state, s, of this automaton that belong to a loop. Write all distinct viable prefixes for the state s.
(c) Explain why all the LR(0) items in state s are valid for the viable prefixes associated with this state.

P14. Do P13 for canonical LR(1) and LALR(1) automaton.

********* End of Tutorial Sheet *********