ASSIGNMENT-001

Programming Languages Compilers (CSE 4021)

Subject Learning Outcome	*Taxonomy Level	Question Number	Marks
Apply the principles in the theory of computation to the various stages in the design of compilers.			
Explain the stages involved in the translation process.			
Acquire knowledge in different phases and passes of Compiler, and specifying different types of tokens by lexical analyzer, and also able to use the compiler tools like LEX, YACC, etc.	L3	1	4
Understand and design Parser(s) (LL, SLR, CLR and LALR) and its types i.e. Top-down and Bottom-up parsers.	L3, L5	3, 4	4+4
Apply and evaluate syntax directed translation schemes, synthesized attributes, inherited attributes, different techniques for code optimization, symbol table organization, code generation and the fundamentals of run time environment			
Design different types of compiler tools to meet the requirements of the realistic constraints of compilers.	L5, L5	2, 5	4+4

*Bloom's taxonomy levels: Knowledge (L1), Comprehension (L2), Application (L3), Analysis (L4), Evaluation (L5), Creation (L6)

This assignment is designed to give you practice with:

(1) tokenization of strings using flex scanner, providing a much deeper understanding of how flex scanner works using longest match algorithm. **[Part-1]**

(2) instance of regular expression providing you the limits of conflict resolution in flex scanner. **[Part-2]**

(3) LL(1) parsing algorithms we have talked about in class. Completing the assignment, you should have a great idea about the working principles of the LL(1) parsing algorithms. **[Part-3]**

(4) SLR(1) parsing algorithms we have talked about in class. Completing the assignment, you should have a great idea about the working principles of the SLR(1) parsing algorithms. **[Part-4]**

(5) Completing the assignment, you should have a great idea about how the LALR(1)-by-SLR(1) algorithm works in practice. **[Part-5]**

Plagiarized assignments will be given a zero mark.

[Part-1] Longest match:

Given the following tokens and their associated regular expressions, show what output is produced when the `flex` scanner is run over the following strings:

%%	
0*1	printf("a");
(0 | 1)*1	printf("b");
2*	printf("c");

i. 0001220111

ii. 2111102

iii. 21012

[Part-2] The Limits of Conflict Resolution:

Conflict can occur when regular expressions is used to scan an input. To resolve the conflict, our approach was to use longest match to always choose the longest possible match at any point, then to break ties based on the priorities of the regular expressions. However, this is not the only way that we could have resolved conflicts. Our example with two types of regular expressions: an expression for the keyword `for` and an expression for identifiers: The string **fort** could be tokenized in nine

%%	
"for"	{return T_for; }
[A-Za-z_] [A-Za-z0-9_]*	{return T_Identifier; }

possible ways, based on how we chose to apply the regular expression. In this case, longest match algorithm dictates that we would scan the string as the identifier **fort**.
However, in some cases we may have a set of regular expressions for which it is possible to tokenize a particular input string, but for which the longest match algorithm will not be able to break the input into tokens. Give an example of such a set of regular expressions and an input string such that

- The string can be broken apart into substrings, where each substring matches one of the regular expressions,
- The longest match algorithm will fail to break the string apart in a way where each piece matches one of the regular expressions.

Additionally, explain how the string can be tokenized and why longest match will fail to tokenize it.

[Part-3] LL(1):

Suppose that we want to describe Java-style class declarations like these using a grammar:

```
class Car extends Vehicle
public class JavaIsCrazy implements Factory, Builder, Listener
public final class President extends Person implements Official
```

One such grammar for this is

(1) C → P F **class identifier** X Y
(2) P → **public**
(3) P → ϵ
(4) F → **final**
(5) F → ϵ
(6) X → **extends identifier**
(7) X → ϵ
(8) Y → **implements** I
(9) Y → ϵ
(10) I → **identifier** J
(11) J → **,** I *(note the comma before the I)*
(12) J → ϵ

Your job is to construct an LL(1) parser table for this grammar. For reference, the terminals in this grammar are
public final class identifier extends implements , $
Where $ is the end-of-input marker, and the non-terminals are C P F X Y I J.

i. Compute the FIRST sets for each of the non-terminals in this grammar. Show your result.
ii. Compute the FOLLOW sets for each of the non-terminals in this grammar. Show your result.
iii. Using your results from (i) and (ii), construct the LL(1) parser table for this grammar. When indicating which productions should be performed, please use the numbering system from above. Show your result.

[Part-4] SLR(1) Parsing:

Below is a context-free grammar for strings of balanced parentheses:

(1) S $\rightarrow$ P

(2) P $\rightarrow$ (P)P

(3) P $\rightarrow \epsilon$

For example, this grammar can generate the strings

()

(()())()

((()))()(()())

In this question, you will construct an SLR(1) parser for this grammar. For reference, the terminal symbols are

( ) **$**

where $ is the end-of-input marker, and the two non-terminals are S and P. S is the special "start" non-terminal, so you don't need to add this yourself.

i. Construct the LR(0) configuration sets for this grammar. Show your result. As a hint, there are six total configuration sets. Note that when dealing with the production P $\rightarrow \epsilon$, there is only one LR(0) item, which is P $\rightarrow \cdot$

ii. Compute the FOLLOW sets for each non-terminal in the grammar. Show your result.

iii. Using your results from (i) and (ii), construct an SLR(1) parsing table for this grammar. Show your result. Note that the LR(0) item P $\rightarrow \cdot$ is a reduce item.

iv. Identify at least one entry in the parsing table that would be a shift/reduce conflict in an LR(0) parser, or explain why one does not exist.

v. Identify at least one entry in the parsing table that would be a reduce/reduce conflict in an LR(0) parser, or explain why one does not exist.

[Part-5] LALR(1)-by-SLR(1):

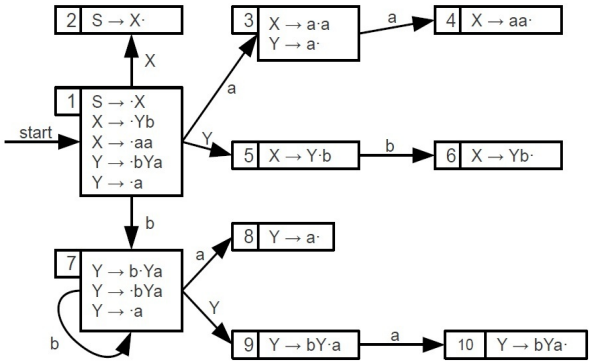
Here is a grammar that is known not to be SLR(1):

$S \rightarrow X$

$X \rightarrow Yb \mid aa$

$Y \rightarrow a \mid bYa$

Here is the associated LR(0) automaton for this grammar:



In this question, you'll get to see how the LALR(1)-by-SLR(1) algorithm works in practice.

i. Why is this grammar not SLR(1)?

ii. Using the LR(0) automaton, construct the augmented grammar that you will use in the algorithm. Show your result.

iii. Compute the FOLLOW sets for every non-terminal in the grammar. Show your result.

iv. Using these FOLLOW sets and the LR(0) automaton, construct the LALR(1) lookaheads for each reduce item in the automaton. Show your results. Note that there are a total of 6 reduce items in the automaton.

v. Is this grammar LALR(1)? Why or why not?

vi. Is this grammar LR(1)? Why or why not?

Submission and Grading:

Submit the **HARD** copy of your assignment through assignment copy only

Part of your assignment grade comes from its "**external correctness**." This is based on correct design layout considering all possible sample cases.

The rest of your assignment's score comes from "**internal correctness**". Internal correctness includes appropriate use of formal definitions tuples, neat diagrams, and tables to enhance readability of your responses.