# Midterm I Solution

- Please read all instructions (including these) carefully.
- There are 5 questions on the exam, some with multiple parts. You have 1 hour and 20 minutes to work on the exam.
- The exam is closed book, but you may refer to your four sheets of prepared notes.
- Please write your answers in the space provided on the exam, and clearly mark your solutions. You may use the backs of the exam pages as scratch paper. Please do not use any additional scratch paper.
- Solutions will be graded on correctness and clarity. Each problem has a relatively simple and straightforward solution. You may get as few as 0 points for a question if your solution is far more complicated than necessary. Partial solutions will be graded for partial credit.

NAME: Sam P. L. Solution _____

SID or SS#: _____

| Problem | Max points | Points |
|---|---|---|
| 1 | 15 | |
| 2 | 25 | |
| 3 | 20 | |
| 4 | 25 | |
| 5 | 15 | |
| TOTAL | 100 | |

1. **Regular Expressions and Finite Automata** (15 points)

Consider the design of a small language using only the letters "z", "o", and the slash character "/". A comment in this language starts with "/o" and ends after the very next "o/". Comments do not nest.

Give a single regular expression that matches exactly one complete comment and nothing else. For full credit, use only the core regular expression notations $A + B$, $AB$, $A^*$, $A^+$, $\epsilon$, and $''abc''$.

This is a classic regular expression puzzle, usually stated as describing C comments. We departed slightly from the usual "/ $*$ ... $*$ /" syntax to avoid confusion between $*$ and $^\star$.

Regular expressions are not unique for a given language, so there are multiple correct answers. Here are just three of the many equally valid approaches:

$$/\mathtt{o}\ (\mathtt{o}^\star\ \mathtt{z} + /)^\star\ \mathtt{o}^+/$$
$$/\mathtt{o}\ /^\star\ (\mathtt{o}^\star\ \mathtt{z}\ /^\star)^\star\ \mathtt{o}^+/$$
$$/\mathtt{o}\ (/^\star\ \mathtt{o}^\star\ \mathtt{z})^\star\ /^\star\ \mathtt{o}^+/$$

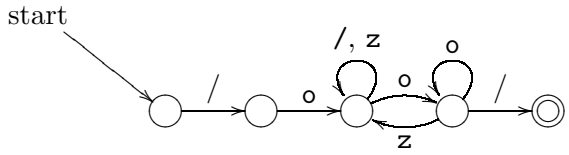
The key feature of each of these solutions is the parenthesized expression in the middle. In each case, this parenthesized expression is designed to prevent an o from ever appearing immediately before a /. They all use a "o$^\star$ z" pattern to represent an optional sequence of o followed by a *mandatory* z. That is really the heart of the solution; the rest just takes care of letting slashes appear elsewhere in the body, and is not difficult to fill in after some careful thought.

One other subtle point is worth noticing. Because "o$^\star$ z" always requires a z after each sequence of o in the comment body, we need some other way of dealing with a sequence of o that runs up to the very end of the comment. For this reason, all three solutions end with "o$^+$/" rather than simply "o/". If we had ended our patterns with "o/", it would have been impossible to match a comment like "/ooo/".

Common errors:

- matching too long a string: the obvious "/o (o + z + /)$^\star$ o/" pattern will incorrectly match "/oo/zzzz/oo/" as one comment
- trying to prevent "o/", but missing cases where we loop through some "(...)$^\star$" expression multiple times: for example, "/o (/ + oz + oo)$^\star$ o$^+$/"
- not allowing a comment that has one or more "/" immediately after the opening "/o"
- not allowing a comment that has one or more "o" immediately before the closing "o/"
- forbidding all "o" or all "/" from the body of a comment
- not allowing "/o" in the body of a comment
- not allowing comments of the form "/oooo...oooo/"
- accepting "/o/" as a comment

We did not ask for a finite automaton that recognizes the same language. Some of you did find it useful, though, to create an automaton first and then work backward to a regular expression. Here is the smallest possible deterministic finite automaton that matches the language described in this question:




The key feature of the above DFA is the pair of states with self edges. The one on the left consumes any sequence of $(/ + \texttt{z})^\star$. The one on the right consumes any sequence of $\texttt{o}^\star$. Once a sequence of $\texttt{o}^\star$ has begun, though, the *only* way to get back to the $(/ + \texttt{z})^\star$ consuming node is to read a "z". If instead we read a "/", we transition into the accepting state.

The other important point is that the accepting state is a dead end; there are no transitions elsewhere. This ensures that we will not pick up anything else after the very first "o/".

2. **Semantic Actions** (25 points)

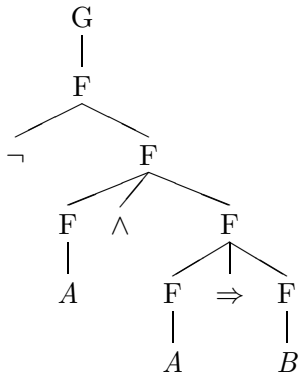
Consider the following grammars and associated semantic actions. In the actions, the operations `And`, `Or`, and `Not` are constructors for an abstract syntax tree data type. For each grammar, answer three things.

- Say whether each attribute of a non-terminal is inherited or synthesized and why.
- Show the value of the attributes of $G$ after parsing $\neg(A \wedge (A \Rightarrow B))$.

(a)

| | | | | | |
|---|---|---|---|---|---|
| $G$ | $\rightarrow$ | $F$ | $G.p$ | $=$ | $F.p$ |
| $F$ | $\rightarrow$ | $F_1 \wedge F_2$ | $F.p$ | $=$ | $And(F_1.p, F_2.p)$ |
| $F$ | $\rightarrow$ | $F_1 \vee F_2$ | $F.p$ | $=$ | $Or(F_1.p, F_2.p)$ |
| $F$ | $\rightarrow$ | $\neg F_1$ | $F.p$ | $=$ | $Neg(F_1.p)$ |
| $F$ | $\rightarrow$ | $F_1 \Rightarrow F_2$ | $F.p$ | $=$ | $Or(Not(F_1.p), F_2.p)$ |
| $F$ | $\rightarrow$ | $(F_1)$ | $F.p$ | $=$ | $F_1.p$ |
| $F$ | $\rightarrow$ | $id$ | $F.p$ | $=$ | $id.lexeme$ |

Both attributes $G.p$ and $F.p$ are synthesized. In every production, $G.p$ and $F.p$ are functions of attributes of their child nodes. The can be computed in a bottom-up parse. A simplified parse tree for the expression $\neg(A \wedge (A \Rightarrow B))$ is shown here:




Using the parse tree, the attribute values can be calculated from the bottom up by applying the semantic actions in the grammar. The value of $G.p$ is *Not(And(A,Or(Not(A),B)))*.

(b) Remember: Say whether each attribute of a non-terminal is inherited or synthesized and why. Show the value of the attributes of $G$ after parsing $\neg(A \wedge (A \Rightarrow B))$.

| | |
|---|---|
| $G \rightarrow F$ | $G.q = F.q$<br>$F.b = true$ |
| $F \rightarrow F_1 \wedge F_2$ | $F.q = F.b \; ? \; And(F_1.q, F_2.q) : Or(F_1.q, F_2.q)$<br>$F_1.b = F.b$<br>$F_2.b = F.b$ |
| $F \rightarrow F_1 \vee F_2$ | $F.q = F.b \; ? \; Or(F_1.q, F_2.q) : And(F_1.q, F_2.q)$<br>$F_1.b = F.b$<br>$F_2.b = F.b$ |
| $F \rightarrow \neg F_1$ | $F.q = F_1.q$<br>$F_1.b = \neg F.b$ |
| $F \rightarrow F_1 \Rightarrow F_2$ | $F.q = F.b \; ? \; Or(F_1.q, F_2.q) : And(F_1.q, F_2.q)$<br>$F_1.b = \neg F.b$<br>$F_2.b = F.b$ |
| $F \rightarrow (F_1)$ | $F.q = F_1.q$<br>$F_1.b = F.b$ |
| $F \rightarrow id$ | $F.q = F.b \; ? \; id.lexeme : Neg(id.lexeme)$ |

The attributes $G.q$ and $F.q$ can be computed bottom-up. However, $F.b$ is inherited, and must be computed top-down. Even though $G.q$ and$F.q$ can be computed bottom-up, they depend on inherited attributes, and are themselves inherited. To compute $G.q$, the value of $F.b$ must be propagated down the tree. Then, $F.q$ can be computed at each node based on the values of $F.b$. The value of $G.q$ is *Or(Not(A),And(A,Not(B)))*.

3. **First and Follow Sets** (20 points)

Give a grammar with the following `First` and `Follow` sets. Your grammar should have exactly two productions per non-terminal and no epsilon productions. The non-terminals are *X, Y, Z* and the terminals are *a, b, c, d, e, f* .

| | | | | | |
|---|---|---|---|---|---|
| First($X$) | = | $\{b, d, f\}$ | Follow($X$) | = | $\{\$\}$ |
| First($Y$) | = | $\{b, d\}$ | Follow($Y$) | = | $\{c, e\}$ |
| First($Z$) | = | $\{c, e\}$ | Follow($Z$) | = | $\{a\}$ |
| Follow($d$) | = | $\{c, e\}$ | Follow($a$) | = | $\{\$\}$ |
| Follow($e$) | = | $\{a\}$ | Follow($b$) | = | $\{b, d\}$ |
| Follow($f$) | = | $\{\$\}$ | Follow($c$) | = | $\{c, e\}$ |

```
There are many ways to solve this problem.  In our solution, we began
by using the First sets to generate the obvious productions that have
the correct first (and only) terminal:

        X -> b | d | f
        Y -> b | d
        Z->  c | e

We are only allowed two productions per non-terminal, so we need to
eliminate an X production.  Noticing that b and d are in the first of
Y, we can try

        X -> Y | f
        Y -> b | d
        Z -> c | e

At this point we have the First sets correct for the non-terminals;
now we try to modify the productions to have the correct Follow sets
as well.  Starting arbitrarily with the terminal c, we notice that
both c and e are in Follow(C) and in First(Z), so we can add Z after c
in some production.  There is only one choice:

        X -> Y  | f
        Y -> b  | d
        Z -> cZ | e

Similar reasoning about Follow(b) leads to:

        X -> Y  | f
        Y -> bY | d
        Z -> cZ | e
```

```
Since $ is not in the follow of Y, there must be something that comes
after Y in the X -> Y production.  Since First(Z) is in the Follow(Y)
we try:


        X -> YZ | f
        Y -> bY | d
        Z -> cZ | e

At this point we still haven't used a.  Now Follow(a) = {$} and
Follow(Z) = {a}. So we get

        X -> YZa | f
        Y -> bY | d
        Z -> cZ | e

At this point it is routine to check that all the requirements are satisfied.
```

4. **LR Parsing and Error Recovery**(25 points)

Consider the following grammar.

$$S \rightarrow S\ a \mid b \mid \texttt{error}\ a$$

(a) Show the DFA for recognizing viable prefixes of this grammar. Use LR(0) items, and treat `error` as a terminal.

```
States:

I0 = { S -> .Sa,  S -> .b, S -> .error a }
I1 = { S -> S.a }
I2 = { S -> b. }
I3 = { S -> error . a }
I4 = { S -> error a. }
I5 = { S -> Sa.}

Transitions:

I0 -S-> I1
I0 -b-> I2
I0 -error-> I3
I1 -a-> I5
I3 -a-> I4
```

(b) Tools such as `bison` use error productions in the following way. When a parsing error is encountered (i.e., the machine cannot shift, reduce, or accept):

- First, pop and discard elements of the stack one at a time until a stack configuration is reached where the `error` terminal of an error production can be shifted on to the stack.
- Second, discard tokens of the input one at a time until one is found that can be shifted on to the stack.
- Third, resume normal execution of the parser.

Show the sequence of stack configurations of an SLR(1) parser for the grammar above on the following input. Be sure to show all shift, reduce, and discard actions (for both stack and input).

bacadfa

```
|bacadfa$       shift b
b|acadfa$       reduce S -> b
S|acadfa$       shift a
Sa|cadfa$       error; discard stack            (*)
S|cadfa$        discard stack
|cadfa$         shift error on to stack (written "e")
e|cadfa$        discard input
e|adfa$         shift a
ea|dfa$         error; discard stack            (**)
e|dfa$          discard stack
|dfa$           shift error on to stack (written "e" again)
e|dfa$          discard input
e|fa$           discard input
e|a$            shift a
ea|$            reduce S -> error a
S|$             - accept -
```

```
Notes:

There was no need to build a parsing table for this problem.  Either
the DFA alone or deducing the moves directly from the grammar were
enough.

A very common mistake was reducing by S -> Sa on line (*) in the
configuration Sa|c.  Reducing here is not possible because c is not in
the Follow(S).  The same mistake can occur on line (**).
```

5. **Miscellaneous Parsing** (15 points)

(a) Give an unambiguous grammar that is not LR(1).

There are an infinite number of unambiguous non-LR(1) grammars. Here is one of the simpler ones:

$$\begin{aligned} S &\rightarrow A\,\texttt{x}\,\texttt{y} \mid B\,\texttt{x}\,\texttt{z} \\ A &\rightarrow \texttt{a} \\ B &\rightarrow \texttt{a} \end{aligned}$$

The grammar is unambiguous; it contains exactly two strings and each string has a unique derivation:

$$\begin{aligned} S &\Rightarrow A\,\texttt{x}\,\texttt{y} \Rightarrow \texttt{a}\,\texttt{x}\,\texttt{y} \\ S &\Rightarrow B\,\texttt{x}\,\texttt{z} \Rightarrow \texttt{a}\,\texttt{x}\,\texttt{z} \end{aligned}$$

However, the grammar is not LR(1). Consider an input that begins with "`a x`". After shifting the `a`, the parser must reduce to either $A$ or $B$. However, based solely on the lookahead character of `x`, we cannot decide which of $A$ or $B$ is correct, leading to a reduce/reduce conflict.

Parsing this grammar correctly would require SLR(2), LR(2), or LALR(2). With two characters of lookahead, we would know to reduce `a` to $A$ on lookahead `x y`, but to reduce `a` to $B$ on lookahead `x z`.

(b) How many strings are in the language of the grammar $S \rightarrow aS$?

The language of a grammar is the set of strings derivable from the grammar's start symbol. A derivation results from applying some finite sequence of productions from the grammar; a string is a finite sequence of terminals selected from a given alphabet $\Sigma$.

The grammar given above can never have a finite derivation that yields a finite string. There is no string of terminals $\omega$ such that $S \Rightarrow^+ \omega$. Therefore, there are *zero* strings in the language of this grammar.

(c) Which of LL(1), SLR(1), and LR(1) can parse strings in the following grammar, and why?

$$\begin{aligned} E &\rightarrow A \mid B \\ A &\rightarrow a \mid c \\ B &\rightarrow b \mid c \end{aligned}$$

The grammar is ambiguous; there are two possible derivations of the string "`c`". Neither LL(1) nor SLR(1) nor LR(1) is able to parse any ambiguous grammar. While it is possible to work through complete LL(1), SLR(1), and LR(1) constructions to show exactly where they fail, that is not necessary. As soon as you show that a grammar is ambiguous you immediately know that none of LL(1), SLR(1), or LR(1) can possibly work.