CS143 Midterm and Solutions
Spring 2012

• Please read all instructions (including these) carefully.

• There are 4 questions on the exam, all with multiple parts. You have 75 minutes to work on the exam.

• The exam is open note - you may use your own notes and laptop/tablet/smartphone, but you may not share notes or electronic devices with others, and you must disable all network access on any device used.

• 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: 

In accordance with both the letter and spirit of the Honor Code, I have neither given nor received assistance on this examination.

SIGNATURE: 

<table>
<thead>
<tr>
<th>Problem</th>
<th>Max points</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>TOTAL</td>
<td>75</td>
<td></td>
</tr>
</tbody>
</table>
1. **Lexical Analysis** (15 points)

Consider the following finite automaton for processing a lexical specification:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>E</td>
<td>4</td>
<td>E</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>E</td>
<td>5</td>
<td>E</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>A</td>
<td>A</td>
<td>T</td>
</tr>
<tr>
<td>5</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>T</td>
</tr>
</tbody>
</table>

The automaton is represented as a table, but note that this table is different from the tables discussed in lecture. The automaton has five states 1-5, each of which has an associated action for each input a-c or end-of-file \$. For a given state S and input (or end-of-file) I:

- If the \([S,I]\) entry is a state \(n\), then in state \(S\) on input \(I\) the automaton transitions to state \(n\) and advances to the next input character.
- If the \([S,I]\) entry is A, then in state \(S\) on input \(I\) the automaton accepts the prefix of the input up to, but not including, the current input character. This prefix is removed from the input, and the automaton moves to state 1 and continues. Note the current input character is not consumed.
- If the \([S,I]\) entry is E, then in state \(S\) on input \(I\) the automaton reports an error and halts.
- If the \([S,\$]\) entry is T, then in state \(S\) on input \(\$\) the automaton terminates successfully.

a) (10 points) Give a lexical specification that tokenizes input strings in the same manner as this automaton. Your lexer should only describe successfully terminating runs of the automaton. For full credit, your specification should be the shortest (in total length of rules) possible.

**Answer:**

\[
\text{[bc]} \\
\text{aaa*}\text{c} \\
\text{aca*}
\]

There were several ways to approach this problem. One of the simpler ones was to convert the DFA described by the table back into the patterns it matches. Since state 5 accepts on all possible inputs (without consuming them), any path to state 5 is a valid token - these are the first two patterns in the specification above. State 4 was a bit more complicated, as it could continue to consume a’s.
b) (5 points) Is this form of automaton (a finite state machine extended with actions A, E, and T) powerful enough to implement any lexical specification given by regular expressions? Ignore error recovery, and justify your answer.

**Answer:** No, this form of automaton is not capable of the lookahead (or backtracking, depending on how you think about it) needed to perform maximal munch. Consider the following lexical specification:

```
  x
  xxy
```

The first state is fine, but we run into trouble in state 2:

```
<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>E</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>T</td>
<td></td>
</tr>
</tbody>
</table>
```

After reading an x and seeing another x as the next input, the only choices for entry [2,x] are to accept just the first x (i.e. an A in that entry) or to consume the second x and transition to a state 3 to look for a y. Maximal munch requires we do the latter if the input really is xxy, but there’s no ability to undo the shift of that second x if the third input character isn’t a y.
2. **LL(1) Parsing** (20 points)

Consider the following grammar:

\[
\begin{align*}
S & \rightarrow \ bAb \mid bBa \\
A & \rightarrow \ aS \mid CB \\
B & \rightarrow \ b \mid BC \\
C & \rightarrow \ c \mid cC
\end{align*}
\]

a) (5 points) Give two reasons why this grammar is not LL(1).

**Answer:** There were actually three reasons (you only had to give two):

1) The grammar is not left-factored (non-terminals S and C).
2) The grammar is left-recursive (B → BC).
3) The grammar is ambiguous (the string bbcca has two different left-most derivations:
   
   \[
   \begin{align*}
   S & \rightarrow \ bB \rightarrow bBCa \rightarrow bbC \rightarrow bbcca, \ \text{or} \\
   S & \rightarrow \ bBa \rightarrow bBCa \rightarrow bbCCa \rightarrow bbcCa \rightarrow bbcca \\
   \end{align*}
   \]

b) (5 points) Rewrite the grammar, introducing as few new non-terminals as possible, so that it is LL(1) and recognizes the same strings as the original grammar.

**Answer:**

\[
\begin{align*}
S & \rightarrow \ bT \\
T & \rightarrow \ Ab \mid Ba \\
A & \rightarrow \ aS \mid CB \\
B & \rightarrow \ bD \\
C & \rightarrow \ cD \\
D & \rightarrow \ \epsilon \mid cD
\end{align*}
\]

The left-factoring of the S and C non-terminals is straight-forward. The B non-terminal was the source of both the left-recursion and ambiguity issues, and both issues are addressed by generating a single b and then zero or more c’s (via new non-terminal D).
c) (5 points) Construct the first and follow sets for each non-terminal in the rewritten grammar.

Answer:

<table>
<thead>
<tr>
<th>Non-Terminal</th>
<th>First</th>
<th>Follow</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>{ b }</td>
<td>{ b, $ }</td>
</tr>
<tr>
<td>T</td>
<td>{ a, b, c}</td>
<td>{ b, $ }</td>
</tr>
<tr>
<td>A</td>
<td>{ a, c }</td>
<td>{ b }</td>
</tr>
<tr>
<td>B</td>
<td>{ b }</td>
<td>{ a, b }</td>
</tr>
<tr>
<td>C</td>
<td>{ c }</td>
<td>{ b }</td>
</tr>
<tr>
<td>D</td>
<td>{ c, $ }</td>
<td>{ a, b }</td>
</tr>
</tbody>
</table>

These answers are for the solution to 2b given above. A different grammar would likely have different first and follow sets.

d) (5 points) Construct an LL(1) parse table for the rewritten grammar.

Answer:

<table>
<thead>
<tr>
<th>Non-Terminal</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td></td>
<td>bT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>Ab</td>
<td>Ba</td>
<td>aB</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>aS</td>
<td></td>
<td>CB</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>bD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td>cD</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>$\epsilon$</td>
<td>$\epsilon$</td>
<td>cD</td>
<td></td>
</tr>
</tbody>
</table>

Again, these answers are for the solution given for 2b. All empty locations are errors.
3. **SLR(1) Parsing** (20 points)

Consider the language generated by the regular expression $0^*11$.

a) (10 points) Write an SLR(1) grammar for this language that requires $O(n)$ stack space for an input string of length $n$. Prove your grammar is SLR(1) and justify the space bound.

**Answer:**

\[
S \rightarrow A11 \\
A \rightarrow \epsilon \mid 0A
\]

This grammar requires $\Theta(n)$ stack space because $\text{Follow}(A) = \{1\}$, so no reductions of $A$ may occur until all the 0’s (of which there are $\Theta(n)$) have been shifted onto the stack. Proving that this grammar is SLR(1) requires generating the DFA and showing there are no conflicts. Only two states have both shift and reduce possibilities, and both are conflict free because the shift only occurs on a 0 input and the reduction only occurs on a 1.

![DFA for SLR(1) grammar](image)

b) (10 points) Write an SLR(1) grammar for this language that requires $O(1)$ stack space for an input string of length $n$. Prove your grammar is SLR(1) and justify the space bound.

**Answer:**

\[
S \rightarrow A11 \\
A \rightarrow \epsilon \mid A0
\]

The only valid SLR(1) stacks for this grammar are the empty stack, $A$, $A0$, $A1$, $A11$, $S$. All are of constant (i.e. $\Theta(1)$) size. Again, proving that this grammar is SLR(1) requires generating the DFA and showing there are no conflicts. (No state in this DFA has both shift and reduce options.)

![DFA for SLR(1) grammar](image)
4. Language Design (20 points)

Pat is writing a compiler for equality tests on numbers, and wants to handle both integers and floating point values. Pat’s initial version looks like this:

\[
\begin{align*}
\text{compare} & \rightarrow \text{fltcmp} | \text{intcmp} \\
[0-9]+ & \rightarrow \text{return(INT)}; \\
[0-9]+"."[0-9]* & \rightarrow \text{return(FLOAT)}; \\
"==" & \rightarrow \text{return(EQ)};
\end{align*}
\]

This version works fine with Pat’s shift-reduce parser generator, but Pat wants the language to also allow comparisons between integers and floating point values (converting the integer to floating point before comparison), and adds one more production to the grammar, changing the \text{fltval} rule to:

\[
\text{fltval} \rightarrow \text{FLOAT} | \text{INT}
\]

a) (5 points) This augmented grammar is no longer SLR(1). What conflict(s) will there be in the generation of the parsing table? Can they be resolved with precedence declarations on operators? Why or why not?

**Answer:** There is now a reduce-reduce conflict between \text{fltval} \rightarrow \text{INT} and \text{intval} \rightarrow \text{INT} because both non-terminals have \text{EQ} in their follow sets. (Both also have $ - more discussion about that in 4d.) Precedence declarations on operators won’t help because they only resolve shift-reduce conflicts (and there’s only one operator here anyway).

b) (5 points) Pat’s friend Chris suggests using a recursive descent parser instead. Will a recursive descent parser (using the augmented grammar as written) generate the parse tree that Pat wants in all cases? Justify your answer.

**Answer:** No, this will not work in all cases. The recursive descent parser will generate a parse tree for any string in the language generated by a grammar, but it handles ambiguity in the grammar by always choosing the leftmost valid production. Because \text{compare} \rightarrow \text{fltcmp} comes first in the grammar, all (valid) inputs get parsed as floating-point comparisons, even an input that tokenizes as \text{INT EQ INT}.
c) (5 points) Pat’s other friend Francis thinks that the problem can be solved in the lexical analysis. Is this possible? Justify your answer.

**Answer:** Yes, this is possible (but gross, as several students observed). In general, a context-free grammar can recognize many languages that a lexer cannot, but the language in this problem is regular, so the lexer’s regular expressions could be used to match all four cases:

- `INT EQ INT`
- `INT EQ FLOAT`
- `FLOAT EQ INT`
- `FLOAT EQ FLOAT`

---

d) (5 points) Drew thinks the right answer is to change the language to use a different operator (e.g. `~=`) for floating-point comparisons. Explain how this resolves the conflict(s) in the SLR(1) parsing table, or show what conflict(s) remain.

**Answer:** Yes, this resolves the important conflict in the SLR(1) table because the follow sets for `fltcmp` and `intcmp` are now different, and the SLR(1) table will correctly choose to reduce `fltcmp → INT` when `~=` is the next input token, and `intcmp → INT` when `EQ` is the next input token.

A few students pointed out that `$` is still in the follow set of both non-terminals, and while this is technically an SLR(1) conflict, it turns out not to be critical. A input of just `INT` would hit the conflict, but no matter which reduction is performed, the parse will fail immediately afterward. (More powerful LALR(1) parsers (e.g. bison) are able to understand this and not complain about the conflict.) There is no reduce-reduce conflict on the second `INT` in a valid input, because the parser will be in one of two different states as a result of the first `fltval-or-intval` reduction.