CS143 Practice Midterm and Solution

Exam Facts

Tuesday, February 14th at 7:00 p.m. in Terman Auditorium

Format

I'm writing the exam that, in theory, could be given as a 120-minute timed exam, even though you'll have three hours. The exam is open-notes/closed-book.

Material

The midterm will cover material from all lectures material through LR(1) and LALR(1) parsing. This primarily means the compilation phases lexical and syntax analysis. Material similar to the homework will be emphasized, although you are responsible for all topics presented in lecture and in the handouts. These include:

- Overview of a compiler— terminology, general task breakdown
- Regular expressions— writing and understanding regular expressions, equivalence between regular expressions/NFA/DFA, conversion among forms (Thompson's algorithm, subset construction)
- Lexical analysis—scanner generators, lex/flex
- Grammars— Chomsky's hierarchy, parse trees, derivations, ambiguity, writing simple grammars
- Top-down parsing— LL(1) grammars, grammar conditioning (removing ambiguity, left-recursion, left-factoring), first and follow set computation, top-down recursive descent implementation, table-driven predictive parsing
- Shift-reduce parsing— building LR(0) and LR(1) configurating sets, parse tables, tracing parser operation, shift/reduce and reduce/reduce conflicts, differences between LR, SLR, and LALR
- Comparisons between parsing techniques—advantages/disadvantages: grammar restrictions, space/time efficiency, error-handling, etc.

The rest of this handout is the midterm given this past summer. Solutions to all problems are given at the end of this handout, but we encourage you to not look at them until you have worked through the problems yourself.

Problem 1: Finite Automata and Regular Grammars

Draw a deterministic finite automata that accepts the set of all strings over **(a + b)*** that contain either **aba** or **bb** (or both) as substrings. Then, present a context free grammar that generates the same exact language.

Problem 2: Recursive Descent and Parsing

Consider the following grammar for simple LISP expressions:

$$
\begin{aligned}
&\text{List} \rightarrow (\ \text{Sequence}\ )\\
&\text{Sequence} \rightarrow \text{Sequence Cell}\\
&\text{Sequence} \rightarrow \varepsilon\\
&\text{Cell} \rightarrow \text{List}\\
&\text{Cell} \rightarrow \text{Atom}\\
&\text{Atom} \rightarrow \text{a}
\end{aligned}
$$

where a, (, and) are terminal symbols, all other symbols are nonterminal symbols, with List being the start symbol.

a) Why is the grammar as given not LL(1)?
b) Transform the grammar into a form that is LL(1). **Do not introduce** any new nonterminals.
c) Compute the First and Follow sets for all nonterminals in your LL(1) grammar.
d) Complete the following recursive-descent parser for your LL(1) grammar. You may assume that a suitable function **MatchToken(int token)** and a global variable **lookahead** exist. If you encounter any errors while parsing, then just **exit**. The function for **ParseAtom** is provided, as well as a start for the function `ParseList`.

```
void ParseAtom() {
   switch (lookahead) {
      case 'a': MatchToken('a');
                return;
      default: fprintf(stderr, "Error.\n");

   }
}

void ParseList() {
   switch (lookahead) {
      case '(':
```

Problem 3: LR(1) Configurating Sets

Given the following already augmented grammar:

$S' \rightarrow S$
$S \rightarrow AB \mid AA \mid bC$
$A \rightarrow bCa \mid b$
$B \rightarrow Bd \mid \varepsilon$
$C \rightarrow c$

Draw the goto graph including just those states of an **`LR(1)`** parser that are pushed on the parse stack during a parse of the input **`bdd`** (this will be a subset of all configurating sets). If you accidentally construct irrelevant states, cross them out. Do not add or change any productions in the grammar.

Problem 4: `yacc` and Ambiguity

The following input file is a **`yacc`** specification for strings of balanced parentheses. The scanner associated with it just returns each character read as an individual token.

```
%%

S   :  S S
    |  '(' S ')'
    |  '(' ')'
    ;
```

a) Show that the grammar as given is ambiguous by providing two different parse trees for the smallest string to illustrate the ambiguity.
b) The ambiguity creates a conflict in building the **`yacc`** parser. Identify at what state in the parse the conflict is encountered, on which input tokens, and what kind of conflict it is.
c) The conflict can be resolved by choosing one of the two conflicting actions. Which of the two choices will create a more efficient parser? Briefly explain why.
d) Which of the two choices will be chosen by **`yacc`**'s default conflict resolution rules?
e) Add the necessary **`yacc`** precedence directives to use the alternate choice to resolve the conflict.

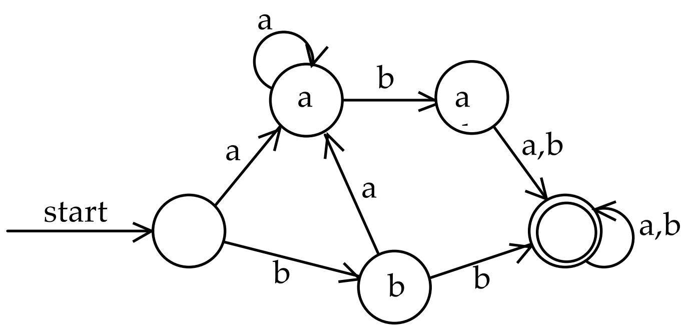
Problem 5: Compare and Contrast

Consider the bottom-up parsing algorithms of SLR(1) and LALR(1). Briefly note what distinguishes these techniques in terms of constructing the parser and the behavior on erroneous input. (Do not describe what is the same, only what is different).

Solution 1: Finite Automata and Regular Grammars

Draw a deterministic finite automata that accepts the set of all strings that contain either **`aba`** or **`bb`** (or both) as substrings. Then, present a context free grammar that generates the same exact language.

Here's the DFA. My automaton accepts the language, though it's better phrased as the language that accepts **`bb`** and the set of all strings containing either **`aba`** or **`abb`** as substrings.



The most straightforward context free grammar is

S → RabaR
S → RbbR
R → bR
R → aR
R → ε

S sets the constraint that either **`aba`** or **`bb`** needs to sit in between two random strings of length 0 or more. If I were truly evil, I would have constrained your grammar to be right-regular, in which cause you would have needed to emulate your DFA that that a leftmost derivation corresponded to some path through the automaton. Here's the right-regular grammar that maps to the above machine:

S → aA
S → bB
A → aA
A → baF
A → bbF
B → aA
B → bF
F → aF
F → bF
F → ε

Problem 2: Recursive Descent and Parsing

Consider the following grammar for simple LISP expressions:

```
List → ( Sequence )
Sequence → Sequence Cell
Sequence → ε
Cell → List
Cell → Atom
Atom → a
```

where a, (, and) are terminal symbols, all other symbols are nonterminal symbols, with List being the start symbol.

a) Why is the grammar as given not LL(1)?

The grammar as specified is left-recursive, and that interferes with the ability of a predictive parser to do its think given just one lookahead token.

b) Transform the grammar into a form that is LL(1). **Do not introduce** any new nonterminals.

Left recursion: bad. Right recursion: not bad.
Had the cells been comma-delimited, things would have been more complicated.

```
List → ( Sequence )
Sequence → Cell Sequence
Sequence → ε
Cell → List
Cell → Atom
Atom → a
```

c) Compute the First and Follow sets for all nonterminals in your LL(1) grammar.

First(List) = { (}
First(Sequence) = { a, (, ε }
First(Cell) = { a, (}
First(Atom) = { a }

Follow(List) = { a, (,) , $}
Follow(Sequence) = {) }
Follow(Cell) = { a, (,) }
Follow(Atom) = { a, (,) }

d) Complete the following recursive-descent parser for your LL(1) grammar. You may assume that a suitable function **`MatchToken(int token)`** and a global variable **`lookahead`** exist. If you encounter any errors while parsing, then just **`exit`**. The function for **`ParseAtom`** is provided, as well as a start for the function **`ParseList`**.

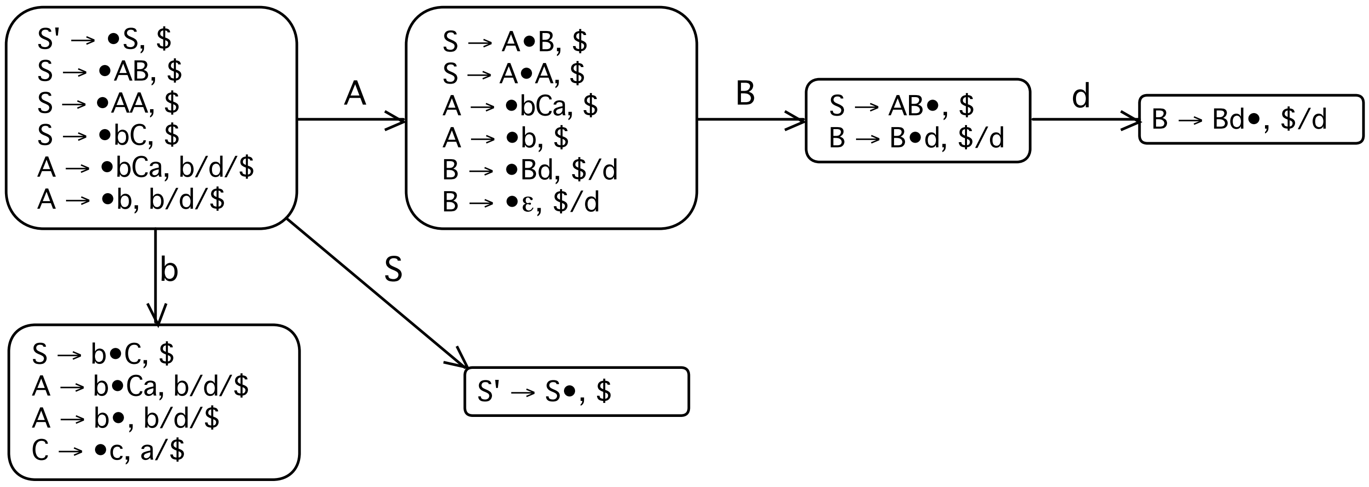
```
void ParseAtom()
{
   switch (lookahead) {
      case 'a': MatchToken('a');
                return;
      default: fprintf(stderr, "Error.\n");
   }
}

void ParseList()
{
   switch (lookahead) {
      case '(': MatchToken('(');
                ParseSequence();
                MatchToken(')');
                return;
      default: fprintf(stderr, "Error,\n");
   }
}

void ParseSequence()
{
   switch (lookahead) {
      case '(':
      case 'a': ParseCell();
                ParseSequence();
                return;
      case ')': return;  // Sequence -> epsilon
      default: fprintf(stderr, "Error,\n");
}

void ParseCell()
{
   switch (lookahead) {
      case 'a': ParseAtom();
                return;
      case '(': ParseList();
                return;
      default: fprintf(stderr, "Error,\n");
}
```

Problem 3: LR(1) Configurating Sets

Given the following already augmented grammar:

S' → S
S → AB | AA | bC
A → bCa | b
B → Bd | ε
C → c

Draw the goto graph including just those states of an `LR(1)` parser that are pushed on the parse stack during a parse of the input **`bdd`** (this will be a subset of all configurating sets). If you accidentally construct irrelevant states, cross them out. Do not add or change any productions in the grammar.



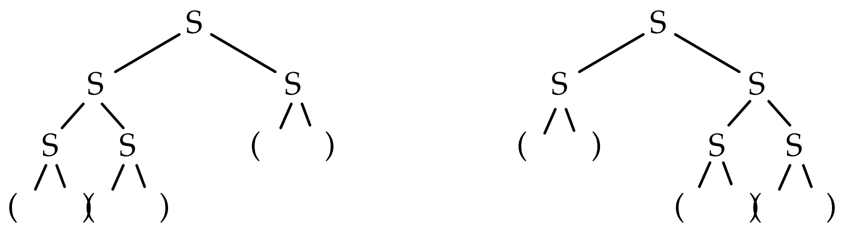
Problem 4: yacc and Ambiguity

The following input file is a **yacc** specification for strings of balanced parentheses. The scanner associated with it just returns each character read as an individual token.

```
%%

S  :  S S
   | '(' S ')'
   | '(' ')'
   ;
```

a) Show that the grammar as given is ambiguous by providing two different parse trees for the smallest string to illustrate the ambiguity.

The string **()()()** has two different parse trees. Both effectively include S → SS, but one has the first of those two S's split, and the other has the second of those two S's split.



b) The ambiguity creates a conflict in building the **yacc** parser. Identify at what state in the parse the conflict is encountered, on which input tokens, and what kind of conflict it is.

You can almost bet up front that it'll be a shift-reduce conflict, since reduce-reduce conflicts are rare. But we need to be a little more scientific than that, so we need to figure out what configurating set causes the problem.

At some point during the parse, two or more S's will be on top of the stack, and '(' will be the lookahead character. We could shift the '(' and work toward what's supposed to be another S, or we could reduce S → S S and pop the top two states onto the stack and replace it with a new one. That's a shift-reduce conflict.

c) The conflict can be resolved by choosing one of the two conflicting actions. Which of the two choices will create a more efficient parser? Briefly explain why.

The more efficient option would be to reduce, since it would keep the stack depth to a minimum.

d) Which of the two choices will be chosen by **yacc**'s default conflict resolution rules?

yacc always chooses the shift by default, and it tells you so.

e) Add the necessary **yacc** precedence directives to use the alternate choice to resolve the conflict.

To force **yacc** to reduce instead, we need to set the precedence of the rule being reduced to be higher than the token being shifted. Add these precedence directives:

```
%nonassoc '('
%nonassoc Higher

S : S S %prec Higher
```

You can also use one precedence directive as long as it is left-associative:

```
%left '('

S : S S %prec '('
```

Problem 5: Compare and Contrast

Consider the bottom-up parsing algorithms of SLR(1) and LALR(1). Briefly note what distinguishes these techniques in terms of constructing the parser and the behavior on erroneous input. (Do not describe what is the same, only what is different).

Table construction: An SLR(1) parser uses only LR(0) configurations, which have no context, and don't require computing and propagating lookaheads. Reduction actions are entered in the table for all terminals in the follow set. An LALR(1) parser is prepared to uses LR(1) configurating sets, merging states whenever it can, to arrive at a goto graph that's isomorphic to that of the SLR(1) parser. The table, however, only includes a reduce action when the lookahead character is in the lookahead set (as opposed to the full follow set). So the LALR(1) parsing table is more sparsely populated than the corresponding SLR(1) table.

Erroneous input: An LALR(1) parser never shifts any erroneous tokens and never reduces any production in error. Any error is detected immediately at first sight of the erroneous token and no further actions are taken. Because it has a more precise context, an LALR(1) parser may be able to report the error a little more accurately. An SLR(1) parser never shifts any erroneous token but may make fruitless reductions. If the next input is a member of the follow set but not in the particular lookahead, the SLR(1) parser will reduce. It will never shift an erroneous token, though.