CS143 Practice Midterm and Solution

Exam Facts

Wednesday, July 20th from 11:00 a.m. – 1:00 p.m. in Gates B01

Format

The exam is designed to take roughly 90 minutes, though you will have two hours to complete the exam. The exam is open-notes, open-book, open-computer, but closed-network, meaning that you can download any of the handouts and slides that you'd like in advance, but must not use the Internet, IM, email, etc. during the exam.

Material

The midterm will cover material on the lectures beginning at scanning and ending at Earley parsing. This means you should fully understand the lexical and syntax analysis phases of traditional compilation. Material similar to the homework will be emphasized, although you are responsible for all topics presented in lecture and in the handouts. Make sure that you have an intuition for the concepts as well as an understanding, since some of the questions will ask you to think critically about the different scanning and parsing algorithms.

Possible topics for the midterm include

- Overview of a compiler— terminology, general task breakdown
- Regular expressions— writing and understanding regular expressions, NFAs, and DFAs; conversion from regular expressions to NFAs; and conversion from NFAs to DFAs.
- Lexical analysis—scanner generators, flex
- Grammars— 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, 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; and construction of SLR and LALR lookaheads.

- Earley parsing – scanning, completion, and prediction; recognition versus parsing; parse-forest grammars; the Earley-on-DFA algorithm.
- Comparisons between parsing techniques—advantages/disadvantages: grammar restrictions, space/time efficiency, error-handling, etc.

The rest of this handout a modified and extended version of a midterm that was given a few years ago. 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. Understand that we are in no way obligated to mimic the problem format of the exam. Everything up to and including Earley parsing is fair game.

The actual midterm will not have as many questions as are covered here. This is mostly designed to give you a sample of what we might ask you.

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: LL(1) 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)?
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) Construct an LL(1) parse table for your resulting grammar.

Problem 3: LR(1) Configurating Sets

Given the following already augmented grammar:

$$S' \rightarrow S$$
$$S \rightarrow aAB \mid \text{ecS} \mid \text{aBb}$$
$$A \rightarrow aA \mid \text{a}$$
$$B \rightarrow cd \mid \text{Be} \mid \epsilon$$

Draw a goto graph of exactly those states of an **LR(1)** parser that are pushed on the parse stack during a parse of the input **aaaaee** (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: `bison` and Ambiguity

The following input file is a **`bison`** 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 **`bison`** 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 **`bison`**'s default conflict resolution rules?
e) Add the necessary **`bison`** 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).

Problem 6: Earley Parsing

(i) Trace the execution of the Earley recognizer on the string **baab** for the grammar

$S \rightarrow E$
$E \rightarrow bEb \mid aEa \mid \varepsilon$

Show the resulting item sets. Does the recognizer accept this string?

(ii) A *prefix* of a string T is a (possibly empty) substring of T that starts at the first character. For example, **graph** is a prefix of **graphical**, but **hica** is not. The empty string is a prefix of all strings.

Using your result from part (i), determine all prefixes of **baab** that are in the language of the grammar. Justify your result.

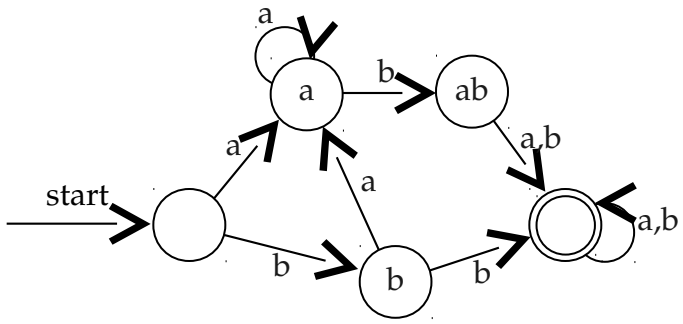
(iii) Suppose that we wanted to consider the programming language JavaLite, which is like Java but without the keywords **`double`** or **`float`**. To do this, we will use the Earley-on-DFA algorithm to filter the standard Java grammar through a finite automaton that will only accept strings not containing these tokens.

Assuming that T is the set of legal tokens in Java (this includes tokens like identifiers, keywords, and specifically the keywords **`double`** and **`float`**), devise a DFA that we could feed into the Earley-on-DFA algorithm to disallow any program using the keywords **`double`** or **`float`**. Justify your answer.

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

$S \rightarrow RabaR$
$S \rightarrow RbbR$
$R \rightarrow bR$
$R \rightarrow aR$
$R \rightarrow \varepsilon$

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:

$S \rightarrow aA$
$S \rightarrow bB$
$A \rightarrow aA$
$A \rightarrow baF$
$A \rightarrow bbF$
$B \rightarrow aA$
$B \rightarrow bF$
$F \rightarrow aF$
$F \rightarrow bF$
$F \rightarrow \varepsilon$

Solution 2: LL(1) 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. Construct an LL(1) parse table for your resulting grammar.

If we number the new production as

(1) List → (Sequence)
(2) Sequence → Cell Sequence
(3) Sequence → ε
(4) Cell → List
(5) Cell → Atom
(6) Atom → a

then the table would look as follows:

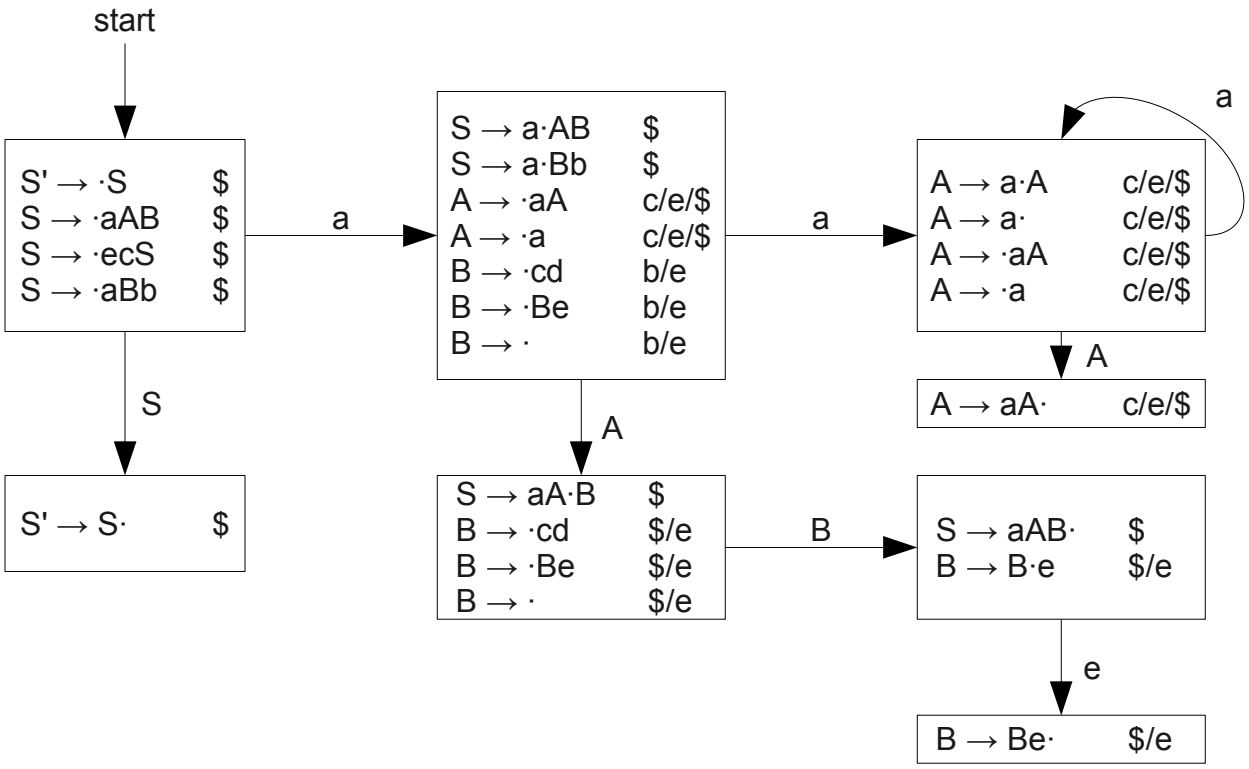
	(	)	a	$
List	1			
Sequence	2	3	2	
Cell	4		5	
Atom			6	

Solution 3: LR(1) Configurating Sets

Given the following already augmented grammar:

$$S' \rightarrow S$$
$$S \rightarrow aAB \mid \text{ecS} \mid \text{aBb}$$
$$A \rightarrow aA \mid \text{a}$$
$$B \rightarrow cd \mid \text{Be} \mid \epsilon$$

Draw a goto graph of exactly those states of an **LR(1)** parser that are pushed on the parse stack during a parse of the input **aaaaee** (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.



Solution 4: `bison` and Ambiguity

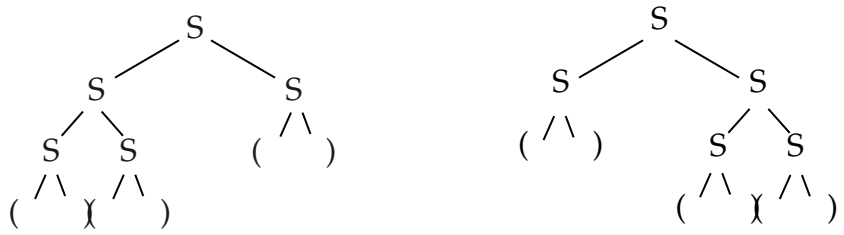
The following input file is a **`bison`** specification for strings of balanced parentheses. The scanner associated with it just returns each character read as an individual token.

```
%%

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 **`bison`** 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 **`bison`**'s default conflict resolution rules?

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

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

To force **`bison`** 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 '('
```

Solution 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.

Solution 6: Earley Parsing

(i) Trace the execution of the Earley recognizer on the string **baab** for the grammar

S → E
E → bEb | aEa | ε

Show the resulting item sets. Does the recognizer accept this string?

The item sets are:

Item Set 1	Item Set 2	Item Set 3	Item Set 4	Item Set 5
S → ·E @1	E → b·Eb @1	E → a·Ea @2	E → a·Ea @3	E → bEb· @1
E → ·bEb @1	E → ·bEb @2	E → ·aEa @3	E → aEa· @2	E → b·Eb @4
E → ·aEa @1	E → ·aEa @2	E → ·bEb @3	E → bE·b @1	S → E· @1
E → · @1	E → · @2	E → · @3	E → ·aEa @4	E → ·aEa @5
S → E· @1	E → bE·b @1	E → aE·a @2	E → ·bEb @4	E → ·bEb @5
			E → · @4	E → ·
			E → aE·a @3	E → bE·b @4

Because the last item set contains the item S → E· @1, the recognizer accepts the string.

(ii) A *prefix* of a string T is a (possibly empty) substring of T that starts at the first character. For example, **graph** is a prefix of **graphical**, but **hica** is not. The empty string is a prefix of all strings.

Using your result from part (i), determine all prefixes of **baab** that are in the language of the grammar. Justify your result.

Any item set containing S → E· @1 represents a prefix of the string that is in the grammar. Since this item is only in the very first and very last item set, the only prefixes of the string that are accepted are the empty string and the complete string **baab**.

(iii) Suppose that we wanted to consider the programming language JavaLite, which is like Java but without the keywords **`double`** or **`float`**. To do this, we will use the Earley-on-DFA algorithm to filter the standard Java grammar through a finite automaton that will only accept strings not containing these tokens.

Assuming that T is the set of legal tokens in Java (this includes tokens like identifiers, keywords, and specifically the keywords **`double`** and **`float`**), devise a DFA that we could feed into the Earley-on-DFA algorithm to disallow any program using the keywords **`double`** or **`float`**. Justify your answer.

> One DFA is as follows. The automaton dies if it sees **`double`** or **`float`**, so any string containing those tokens will be rejected. The only remaining state is accepting, so any strings that remain are accepted.