CS143 Handout 20
Summer 2012 July 18th, 2012

Practice CS143 Midterm Exam

This midterm exam is open-book, open-note, open-computer, but closed-network. This means that if you want to have your laptop with you when you take the exam, that's perfectly fine, but you must **not** use a network connection. You should only use your computer to look at notes you've downloaded from online in advance.

SUNetID: ________________________________

Last Name: ________________________________

First Name: ________________________________

I accept both the letter and the spirit of the honor code. I have not received any assistance on this test, nor will I give any.

(signed) ____

You have two hours to complete this midterm. There are sixty total points, which means that as a rough heuristic you might want to spend about two minutes per point. This midterm is worth 20% of your total grade in this course.

Good luck!

Question		**Points**	**Grader**
(1) C-Style Comments	(8)	/8	
(2) LL Parsing	(16)	/16	
(3) LR Parsing	(20)	/20	
(4) Right-to-Left Parsing	(16)	/16	
	(60)	**/60**	

The actual midterm exam will have space in which you can write your answers. In order to save paper, I've taken most of the whitespace out of this handout.

Problem 1: C-Style Comments **(8 points total)**

Perhaps the hardest part of the first programming assignment was correctly handling C-style comments, which begin with **`/*`** and end with **`*/`**. The challenge with matching these comments is, as you probably realized, that the "obvious" regular expression for C-style comments:

```
("/*").*("*/")
```

does not interact correctly with maximal-munch. In particular, if this regular expression is run using maximal-munch on the string

```
/* Comment. */ "No Comment." /* Comment. */
```

it will match the entire string as a comment, even though the intended tokenization is as a comment, then a string literal, then another comment.

In this question, you'll consider two proposed solutions to the problem.

(i) `flex` with Custom NFAs **(4 Points)**

Suppose that an open-source contributor adds an extension to **`flex`** that allows you to describe lexemes to match using finite automata instead of regular expressions. The rules of maximal-munch still apply to these automata, so if an automaton can match a string in multiple ways, **`flex`** will pick the longest such match.

In the space below, construct an NFA that correctly accepts C-style comments, even when using maximal-munch. For simplicity, you can assume that the alphabet consists of **`/`**, **`*`**, and the character **`a`** (which represents "something other than slash or star"). Be sure to label the start state and any accepting states. You do not need to construct the automaton using the RE-to-NFA algorithm from class.

(ii) Negated Regular Expressions **(4 Points)**

Another potential solution to the problem would be to support *negated regular expressions*. We augment the standard regular expression operators with a new operator, `!`, which matches anything *except* the given regular expression. For example, the regular expression `a!` matches any string except the single-character string **a**, while

```
("/*")(.*"*/".*!)("*/")
```

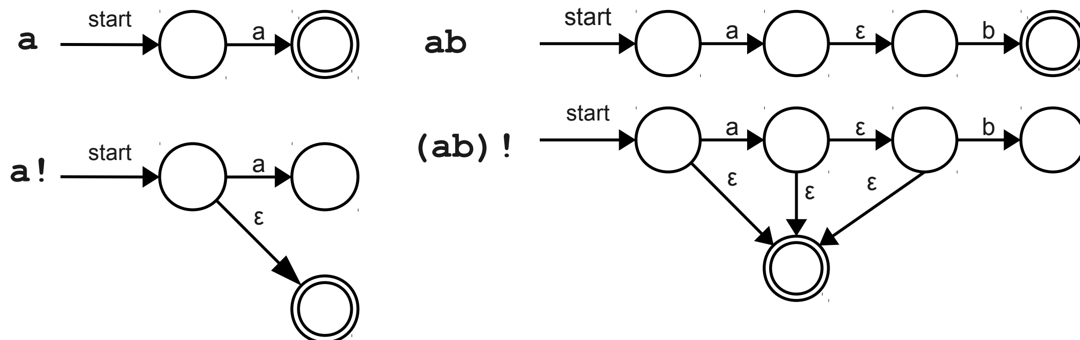
would match C-style comments by matching the open comment, then any string that doesn't contain a close comment symbol, and finally a close-comment symbol.

In order for this operator to be useful in **flex**, we need to come up with a construction that will convert regular expressions that use the `!` operator into automata.

Consider the following proposed construction for converting a negated regular expression *R*`!` to an automaton:

1. Construct the automaton for *R* using the algorithm covered in class.
2. Create a new accepting state.
3. For each state in *R* except for *R*'s accepting state, add an ε-transition from that state to the new accepting state.
4. Make the old accepting state of *R* no longer accepting.

For example, here is a side-by-side comparison of the automata for the regular expressions **a** and **a!**, along with those for **ab** and **(ab)!**



This construction contains an error that, for certain regular expressions using `!`, will cause the generated automaton to be incorrect. Find such a regular expression, show the resulting automaton, and explain why it is incorrect.

Problem 2: LL(1) Parsing **(16 points total)**

Consider the following context-free grammar, which describes C-style function declarations involving pointers:

Function → *Type* **id (***Arguments***)**
Type → **id**
Type → *Type******
Arguments → *ArgList*
Arguments → ε
ArgList → *Type* **id ,** *ArgList*
ArgList → *Type* **id**

For example, this grammar could generate declarations like these:

```
id* id()
id** id(id*** id)
id id(id id, id* id)
```

For reference, the terminals in this grammar are

id ( ) * , $

where **$** is the end-of-input marker, and the nonterminals are

Function
Type
Arguments
ArgList

(i) The First Step is Admitting It **(2 Points)**

As written, this grammar is not LL(1). Identify the conflicts in the grammar that make it not LL(1) and explain each.

(ii) Repairing the Grammar **(4 Points)**

Based on the conflicts you identified above, modify the grammar so that it is LL(1). You may introduce new nonterminals if you wish, but do not change any rules besides those you identified in part (i) as causing the problem. Show the resulting grammar below.

(iii) FIRST sets **(3 Points)**

Construct the FIRST sets for each nonterminal in the modified grammar. Show your result below.

(iv) FOLLOW sets **(3 Points)**

Construct the FOLLOW sets for each nonterminal in the modified grammar. Show your result below.

(v) Building the Parse Table **(4 Points)**

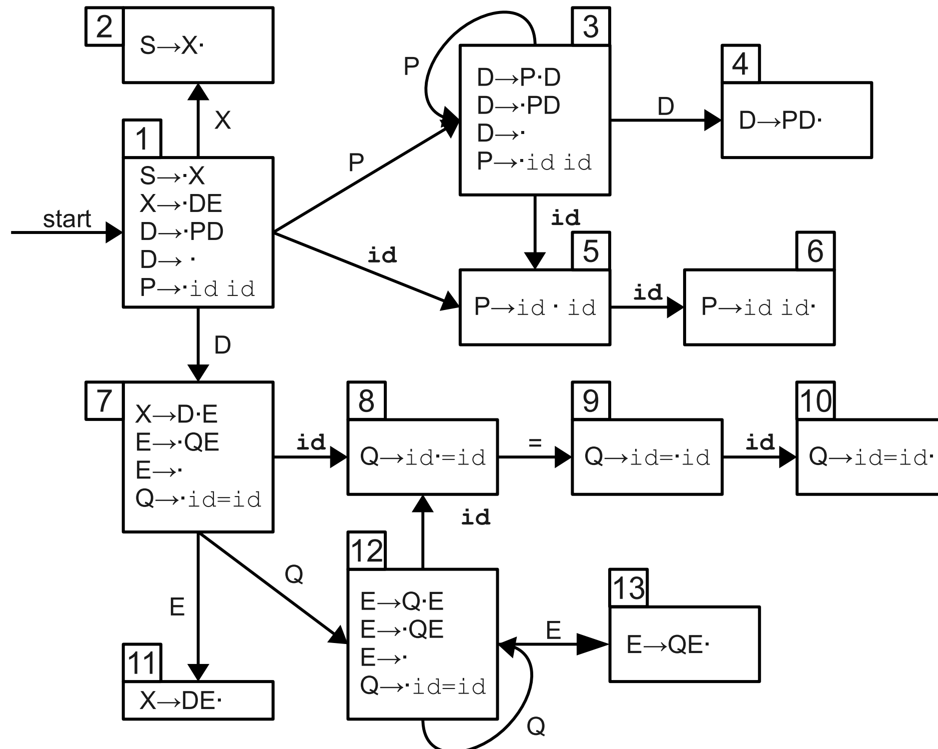
Using your results from parts (ii), (iii), and (iv), construct the LL(1) parse table for your updated grammar. If you identify any LL(1) conflicts, don't worry; just put all applicable entries in the table. In other words, if you discover now that your grammar is not LL(1), we won't hold that against you for this part of the problem.

Problem 3: LR Parsing **(20 points total)**

In this problem, we will explore how several different LR parsing algorithms handle or fail to handle a particular grammar:

$S \to X$
$X \to DE$
$D \to PD$
$D \to \varepsilon$
$E \to QE$
$E \to \varepsilon$
P → **id id**
Q → **id = id**

Below is an LR(0) automaton for this language. We've numbered the states for your convenience.



The terminals in this grammar are

id **=** **$**

where **$** is the end-of-input symbol.

Feel free to tear out this sheet as a reference.

(i) LR(0) **(2 Points)**

This grammar is not LR(0). Why not?

(ii) SLR(1) **(4 Points)**

Is this grammar SLR(1)? If so, why? If not, why not?

(iii) LALR(1), Part One **(4 Points)**

In this next section, you'll determine the lookaheads for each reduce item to see if the grammar is LALR(1) by using the LALR(1)-by-SLR(1) algorithm. As a first step, create the augmented LALR(1) grammar for this language using the algorithm covered in lecture. Show your result.

(iv) LALR(1), Part Two **(3 Points)**

Compute the FOLLOW sets for each nonterminal in the LALR(1)-augmented grammar. Show your result below.

(v) LALR(1), Part Three **(4 Points)**

Is this grammar LALR(1)? If so, use your answers from parts (iii) and (iv) to prove why. If not, use your answers from parts (iii) and (iv) to prove why not. *(Even if you already know for a fact whether this grammar is LALR(1) or not, you* ***must*** *use the results from the last two problems to back up your claim)*

(vi) LR(1) **(3 Points)**

Is this grammar LR(1)? If so, why? If not, why not?

Problem 4: Right-to-Left Parsing **(16 points total)**

In lecture, we talked about seven parsing algorithms that work from the left to the right: Leftmost DFS, LL(1), LR(0), SLR(1), LALR(1), LR(1), and Earley. In the first written assignment, you had the opportunity to see what happens when you change the left-to-right *scanning* algorithm we covered in class to work from right-to-left. In this question, you'll get to see what happens when you change the left-to-right *parsing* algorithms we covered in class to work from right-to-left.

(i) RR(1) Parsing **(4 Points)**

An RR(1) parser is similar to an LL(1) parser except that it works from right-to-left, tracing out a top-down, *rightmost* derivation. The parser continuously consults the rightmost symbol of the derivation and the rightmost symbol of the input string that has not yet been matched to make its decisions about what production to use. The construction and operation of this parser is analogous to the LL(1) case, except that instead of having FIRST and FOLLOW sets we have LAST and PRECEDE sets, where LAST(A) is the set of terminals that could come at the *end* of a production of A (plus ε if A can derive the empty string), and PRECEDE(A) is the set of the terminals that could come *before* a production of A in some sentential form. Similarly, instead of having an end-of-input marker, we have a *beginning*-of-input marker, which we denote **$**.

Is there a grammar that is RR(1) but not LL(1)? If so, show what it is and explain both why it is not LL(1) and why it is RR(1). If not, explain why not.

(ii) RL(0) Parsing **(4 Points)**

An RL(0) parser is similar to an LR(0) parser except that it scans the input from right-to-left. We can speak of RL(0) items as productions with a dot marking the position of what has been matched so far from the right side instead of the left. For example, if we have an RL(0) item of the form A → *x*·*y*, it means that we have matched *y* so far and are looking forward to matching *x*. Similarly, whereas an LR(0) reduce item has the form A → *v*·, with the dot at the end, an RL(0) reduce item has the form A → ·*v*, with a dot at the *beginning,* since it means we've matched all of *v* in reverse order.

Is there a grammar that is RL(0) but not LR(0)? If so, show what it is and explain both why it is not LR(0) and why it is RL(0). If not, explain why not.

(iii) Reverse Earley Parsing **(4 Points)**

A reverse Earley parser is like a standard Earley parser, except that the SCAN step moves from right-to-left, the PREDICT step makes predictions based on nonterminals appearing before the dot, and the COMPLETE step looks at items with the dot at the beginning rather than the end. Instead of having each item track its start position, items track their end positions. Also, rather than putting the item S → ·E @1 into the first item set at the start of the algorithm, we place the analogous item S → E· @n+1 into the (n+1)st item set at the start. The parser reports that the string has been parsed successfully if it finds S → ·E @n+1 in the first item set at the end of parsing.

Is there a grammar that can be parsed with a reverse Earley parser but not an Earley parser? If so, show what it is and explain why it could be parsed with the reverse Earley parser but not the Earley parser. If not, explain why not.

(iv) Comparison with Right-to-Left Scanning **(4 points)**

In the first problem set, you saw that certain sets of regular expressions would tokenize particular strings differently based on whether the scan was done from left-to-right or from right-to-left. In this question, you will consider the analogous question for right-to-left parsers.

Is there a grammar that has all three of the following properties?

1. The grammar is LR(0).
2. The grammar is RL(0).
3. The parse tree produced by the LR(0) parser is not the same as the parse tree produced by the RL(0) parser.

If so, give an example of one and exhibit the parse trees produced by the LR(0) and RL(0) parsers. If not, explain why not.