CS143 Midterm
Spring 2017

- Please read all instructions (including these) carefully.
- There are 5 questions on the exam, some with multiple parts. You have 80 minutes to work on the exam.
- The exam is open note. You may use laptops, phones and e-readers to read electronic notes, but not for computation or access to the internet for any reason.
- Please write your answers in the space provided on the exam, and clearly mark your solutions. Do not write on the back of exam pages or other pages.
- 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: _____

Problem	Max points	Points
1	15	
2	10	
3	20	
4	15	
5	20	
TOTAL	80	

1. **Lexers** (15 points)

A Berkeley student writing a flex lexer for Cool is having some trouble. Here is the specification he is using for comments. So far, by design, this specification is only for *non-nested* comments—nested comments should generate an error.

The specification uses flex-style *states*. The lexer begins in the `INITIAL` state. The statement `BEGIN(X)` changes the lexer's state to `X`. In state `X`, only rules prefixed with $<$ `X` $>$ can match the input.

```
1  %%
2  <INITIAL>(\*      { BEGIN(COMMENT); return OPEN; }
3  <INITIAL>\*)      { return ERROR; }
4  <COMMENT>(\*      { return ERROR; }
5  <COMMENT><<EOF>> { return ERROR; }
6  <COMMENT>\*)      { BEGIN(INITIAL); return CLOSE; }
7  <COMMENT>[^*]+    { return NO_STAR;}
8  <COMMENT>\*[^)]   { return STAR_NO_CLOSE;}
```

(a) Suppose the student runs his lexer on the input below.

```
(**)(*test***comment)*)
```

Fill in the provided table with the generated tokens, in order, along with the text corresponding to each token. You may not need all rows of the table for your answer.

Token	Matched Text
OPEN	(*
CLOSE	*)
OPEN	(*
NO_STAR	test
STAR_NO_CLOSE	**
STAR_NO_CLOSE	*c
NO_STAR	omment)
CLOSE	*)

(b) This specification will generate an ERROR token for some legal non-nested comments. Concisely explain the bug and give the shortest possible input that illustrates the problem.

Solution. The STAR_NO_CLOSE token always matches exactly two characters: an asterisk followed by a character other than a right parenthesis. The second character matched may be the first asterisk of the closing *). If so, the CLOSE token will not be generated correctly which can lead to an error.

For example, the input (***) is lexed as shown below. The ERROR token is generated due to reaching the end of input while inside the COMMENT state.

Token	Matched Text
OPEN	(*
STAR_NO_CLOSE	**
NO_STAR	)
ERROR	

We also accepted the input "(*(*)" with accompanying explanation for full credit. This is a valid non-nested comment (consisting of just an open parenthesis) but the lexer throws an error because it contains "(*" as a substring inside the comment – despite there being no nesting.

2. **Grammars** (10 points)

Consider the regular expressions over the alphabet $\{0, 1\}$.

- **Empty string**: ε,
- **Single character**: 1 or 0,
- **Union**: $R_1 + R_2$, where R_1 and R_2 are regular expressions,
- **Concatenation**: $R_1 R_2$, where R_1 and R_2 are regular expressions,
- **Iteration (Kleene star)**: R_1^*, where R_1 is a regular expression, and
- **Parenthesized expression**: (R_1)

The order of precedence, from low to high, is union, concatenation, and iteration. Union and concatenation are both left associative.

Give an unambiguous context-free grammar for regular expressions over the alphabet $\{1, 0\}$ that correctly enforces precedence and associativity. Do not use precedence declarations in your solution.

Use $'\epsilon'$ for a terminal symbol denoting the regular expression that matches the empty string, and use ϵ for an epsilon production (i.e., $A \to '\epsilon'$ and $A \to \epsilon$ mean different things).

Solution.

$$
\begin{array}{rcl}
E & \to & E + C \mid C \\
C & \to & CI \mid I \\
I & \to & I * \mid B \\
B & \to & 0 \mid 1 \mid (E) \mid '\epsilon'
\end{array}
$$

3. **Syntax-Directed Translation** (20 points)
Consider the regular expressions over the alphabet of lowercase letters $\{a, b, \ldots, z\}$. Define a syntax-directed translation that computes the set of possible *last* characters of a string. Here are two simple examples:

Example input	Possible endings
ε	$\varnothing$
$\texttt{ab} + \texttt{cd}$	$\{\texttt{b}, \texttt{d}\}$

On the following page is a (ambiguous) grammar for regular expressions. Fill in the missing semantic actions. Observe the following in writing your actions:

- `EPSILON` is a terminal representing the regular expression ε.
- There is a *val* attribute of the `CHAR` terminal whose value is the single lowercase character lexeme.
- Compute a boolean attribute *nullable* that records whether the empty string is in the language of a regular expression.
- Compute a set attribute *endings* that records the set of characters a regular expression can end in. Use standard set operators (i.e., $A \cup B$, $A \cap B$, $A - B$) in your semantic actions where appropriate.
- Use no other attributes but *val*, *nullable* and *endings*.
- Use no global state—all of the actions must be equations between attributes of the grammar symbols.
- Use bison notation to refer to grammar symbols and attributes, so `$$.endings` is the `endings` attribute of the left-hand side non-terminal and `$2.nullable` is the `nullable` attribute of the second symbol on the right-hand side.

```
rexpr -> EPSILON                    {
    $$.endings = {};
    $$.nullable = true;
                                    }

      | CHAR                        {
    $$.endings = { $1.val };
    $$.nullable = false;
                                    }

      | rexpr rexpr                 {
    if($2.nullable)
      $$.endings = $1.endings U $2.endings
    else
      $$.endings = $2.endings
    $$.nullable = $1.nullable && $2.nullable
                                    }

      | rexpr '+' rexpr             {
    $$.endings = $1.endings U $3.endings
    $$.nullable = $1.nullable || $3.nullable
                                    }

      | rexpr '*'                   {
    $$.endings = $1.endings
    $$.nullable = true;
                                    }

      | '(' rexpr ')'               {
    $$.endings = $2.endings;
    $$.nullable = $2.nullable;
                                    }
```

4. **Top-Down Parsing** (15 points)

(a) Consider the following grammar:

```
S -> S; P | P
P -> id | id(E)
E -> E + P | P
```

Give a grammar without left recursion that accepts the same language.

Solution.

```
S -> P ; S | P
P -> id | id(E)
E -> P + E | P
```

(b) Show the LL(1) parsing table for the following grammar. Is the grammar LL(1)?

```
S -> A | B
A -> c
B -> aA | bB
```

Solution.

	a	b	c	$
S	B	B	A	
A			c	
B	aA	bB		

There are no conflicts in the table and thus the grammar is LL(1).

5. **Bottom-Up Parsing** (20 points)

Each of the following subproblems describes a deterministic (i.e., DFA) LR(0) parsing automaton. Give the simplest possible grammar (fewest productions, fewest total symbols) that will result in a parsing automaton satisfying the description. Show your grammar and the parsing automaton with each state labeled with its set of LR(0) items. You do *not* need to analyze the automaton to determine whether the grammar is LR(0) or SLR(1). The grammar and the automaton constitute an answer to each subproblem.

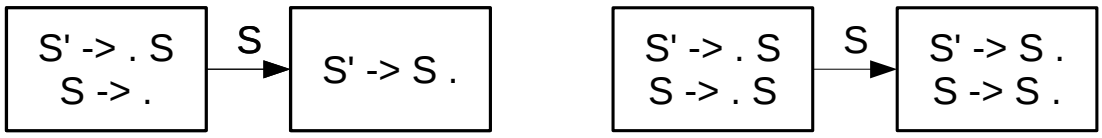
Assume that the first step of the automaton construction is to add a new production $S' \to S$ to the grammar, as described in class.

Note: For each of these problems there are additional correct solutions beyond the ones shown.

(a) An automaton with two states and one transition from the start state to the second state.

Solution.

Grammar: $S \to \epsilon$ **OR** $S \to S$

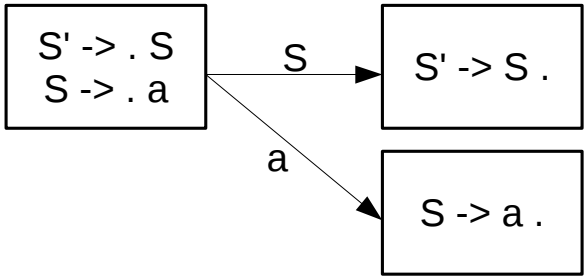


(b) An automaton with three states and two transitions:

- from the start state to the second state,
- from the start state to the third state.

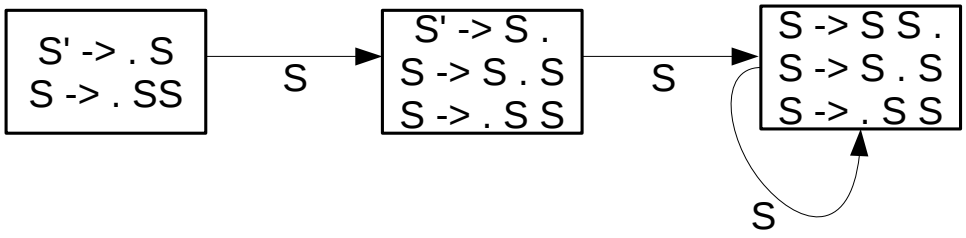
Solution.

Grammar: $S \to a$


S' -> . S
S -> . a
S
S' -> S .
a
S -> a .

(c) An automaton with a self-loop on one state. There is a three state solution using an ambiguous grammar. There is a four state solution for a grammar that is LR(0). Any correct solution with four or fewer states will receive full credit.

Three State Solution.
Grammar: $S \rightarrow SS$



Four State Solution.
Grammar: $S \rightarrow aS|\epsilon$