CS143 Midterm
Spring 2024

- 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 other than to access the class webpage.
- 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.

SUNET ID: _____

NAME: _____

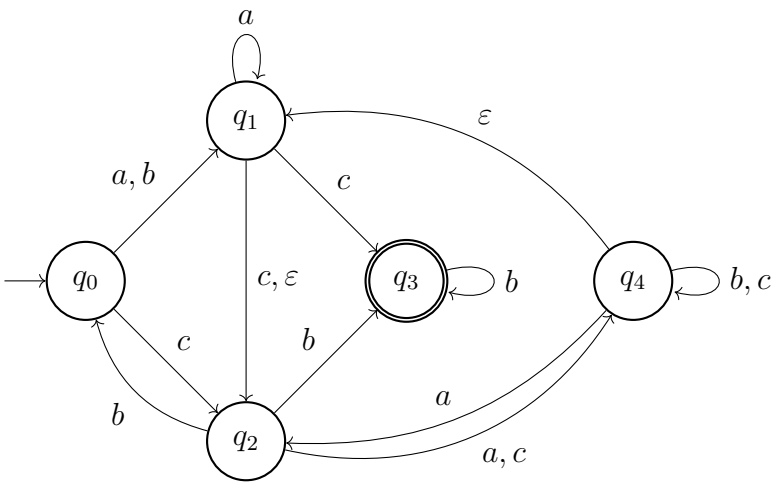
In accordance with both the letter and spirit of the Honor Code, I have neither given nor received assistance on this examination.

SIGNATURE: _____

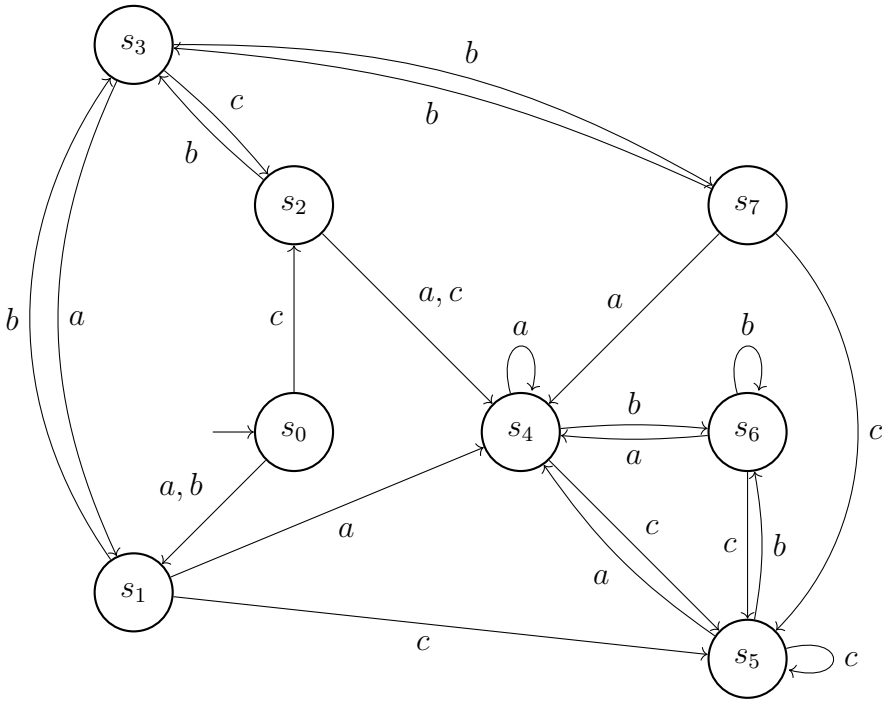
Problem	Max points	Points
1	10	
2	20	
3	30	
4	20	
5	20	
TOTAL	100	

1. NFAs and DFAs

Consider the following NFA over the language $\Sigma = \{a, b, c\}$:



What states in the following DFA must be accepting states to make it equivalent to the above NFA?



Solution:

s_3, s_5, s_6, s_7 should be marked as accepting states.

2. Context-Free Grammars

(a) The following grammar with starting symbol E and terminals `&`, $\sim$, $\wedge$, and `id` generates bitwise expressions.

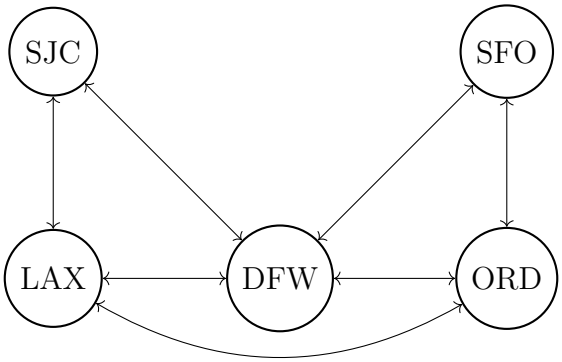
$$E \to \sim E \mid E \ \texttt{\&} \ E \mid E \wedge E \mid \texttt{id}$$

Rewrite the grammar so that the $\sim$ operator has highest precedence (binds tightest), the `&` operator has the second highest precedence, and the $\wedge$ operator has the lowest precedence (binds loosest). The resulting grammar should be unambiguous.

Solution:

$$E \to E \wedge F \mid F$$
$$F \to F \ \texttt{\&} \ G \mid G$$
$$G \to \sim G \mid \texttt{id}$$

(b) Consider the following simplified map of airports in the US.



Construct a CFG that can generate all possible paths from SJC to SFO, where the terminals are airport codes or `->`. These paths may visit any given airport zero, one, or more times. Strings that should be generated include

- `SJC -> DFW -> ORD -> SFO`
- `SJC -> DFW -> ORD -> LAX -> DFW -> SFO`

Strings that should not be generated include

- `SJC -> SFO` (there is no edge connecting these two airports).
- `SJC -> LAX -> SFO` (there is no edge connecting LAX and SFO)
- `SFO -> DFW -> SJC` (the paths should go from SJC to SFO, not from SFO to SJC)

Solution:

$sjc \rightarrow$ `SJC ->` lax | `SJC ->` dfw
$lax \rightarrow$ `LAX ->` sjc | `LAX ->` dfw | `LAX ->` ord
$dfw \rightarrow$ `DFW ->` sjc | `DFW ->` lax | `DFW ->` ord | `DFW ->` sfo
$ord \rightarrow$ `ORD ->` lax | `ORD ->` dfw | `ORD ->` sfo
$sfo \rightarrow$ `SFO ->` dfw | `SFO ->` ord | `SFO`

3. Top-down Parsing

Consider the following CFG over the language $\Sigma = \{a, b, c\}$:

$S \to aE \mid abQ$
$E \to cQ \mid cb \mid \varepsilon$
$Q \to aE \mid \varepsilon$

(a) Construct the First sets for each of the nonterminals.

Solution:

- First$(S) = \{a\}$
- First$(E) = \{c, \varepsilon\}$
- First$(Q) = \{a, \varepsilon\}$

(b) Construct the Follow sets for each of the nonterminals.

Solution:

- Follow$(S) = \{\$\}$
- Follow$(E) = \{\$\}$
- Follow$(Q) = \{\$\}$

(c) Is this grammar LL(1)? Explain.

Solution:

No this grammar is not LL(1) since there is a conflict when looking at the first terminal for S. A lookahead of 1 is insufficient because it would give 'a' which cannot disambiguate the two productions $S \to aE$ and $S \to abQ$.

(d) We learned in lecture how to construct the LL(1) parse table for a grammar. Let's look at extending this algorithm to constructing an LL(2) parse table. Since LL(2) requires a lookahead of two terminals, we need to extend the concept of a First set to a Second set. A Second set contains terminals that could be in the second terminal location for a given nonterminal. Like First sets, Second sets can contain ε. For example, a production $X \to ac$ means $\{a\} \subseteq \text{First}(X)$ and $\{c\} \subseteq \text{Second}(X)$.

Construct the Second sets for each of the nonterminals.

Solution:

- $\text{Second}(S) = \{c, b, \varepsilon\}$
- $\text{Second}(E) = \{a, b, \varepsilon\}$
- $\text{Second}(Q) = \{c, \varepsilon\}$

(e) We will extend the LL(1) parse table to an LL(2) parse table that looks at the next two characters. We must therefore extend the parse table construction rules.

> For your reference, the rules for LL(1) parsing table construction (from Slide 31 in Lecture 7) are as follows: For each production $A \to \alpha$ in the grammar do:
>
> 1. For each terminal $t \in \text{First}(\alpha)$ do $T[A, t] = \alpha$
> 2. If $\varepsilon \in \text{First}(\alpha)$, then for each $t \in \text{Follow}(A)$ do $T[A, t] = \varepsilon$
> 3. If $\varepsilon \in \text{First}(\alpha)$ and $\$ \in \text{Follow}(A)$ do $T[A, \$] = \varepsilon$

The first LL(1) parse table construction rule can be extended to LL(2) as follows:

1. for each $t_1 \in \text{First}(\alpha)$ and $t_2 \in \text{Second}(\alpha)$ do $T[A, t_1, t_2] = \alpha$.

In your answer below, also consider how the other two LL(1) construction algorithm rules (for adding *epsilon*s into the table) extend for LL(2).

Fill in the LL(2) parsing table for the grammar. Where applicable, list all possible productions for every parse table cell.

Solution:

t_1	a				b				c				$\$$
t_2	a	b	c	$\$$	a	b	c	$\$$	a	b	c	$\$$	
S		abQ	aE	aE									
E									cQ	cb		cQ	ε
Q			aE	aE									ε

4. Bottom-up Parsing

Consider the following grammar:

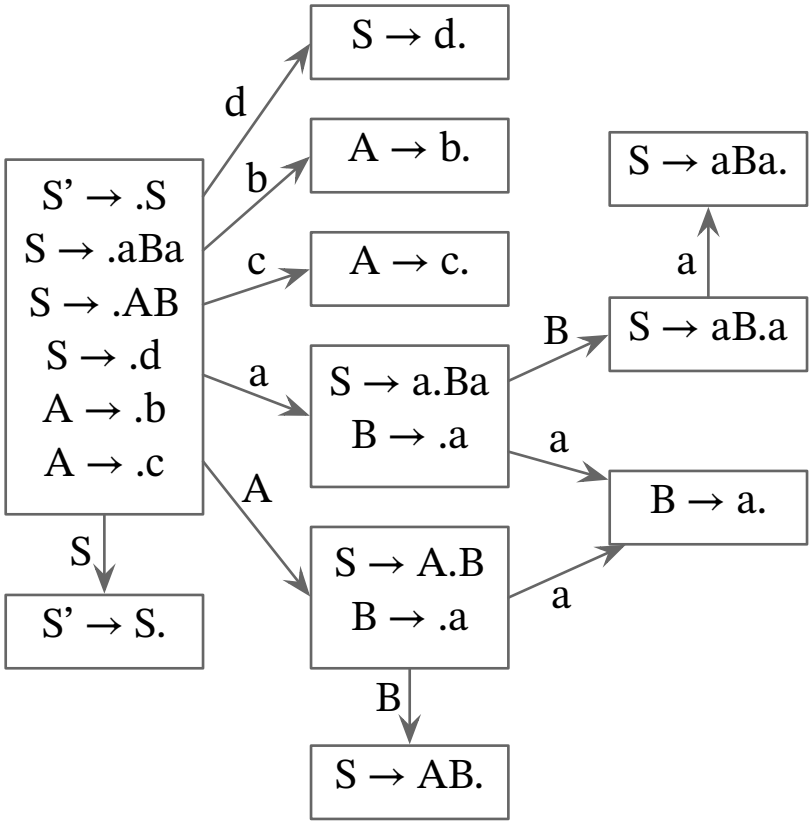
$$\begin{array}{lcl} S & \rightarrow & aBa \mid AB \mid d \\ A & \rightarrow & b \mid c \\ B & \rightarrow & a \end{array}$$

where the terminals are $\{a, b, c, d\}$ and the First/Follow sets are

$$\begin{array}{ll} \text{First}(S) = \{a, b, c, d\} & \text{Follow}(S) = \{\$\} \\ \text{First}(A) = \{b, c\} & \text{Follow}(A) = \{a\} \\ \text{First}(B) = \{a\} & \text{Follow}(B) = \{a, \$\} \end{array}$$

(a) Draw the LR(0) DFA. You only need to provide accepting states, including their items, and transitions between accepting states.

Solution:



(b) Is the grammar LR(0)? Justify your answer.

Solution:

Yes. No state has two reductions (reduce-reduce conflict) or a shift and a reduction (shift-reduce conflict).

(c) Is the grammar SLR? Justify your answer.

Solution:

Yes. No state has two reductions (reduce-reduce conflict) or a shift and a reduction (shift-reduce conflict), with overlapping follow sets.

5. Expression simplification.

Consider mathematical expressions consisting of only multiplication and addition of variables and integers, and parentheses. This is shown in the following abstract grammar:

$$
\begin{array}{ll}
E \to E + T & \texttt{\{ \$\$ = do\_plus(\$1, \$3); \}} \\
\quad \mid T & \texttt{\{ \$\$ = \$1; \}} \\
T \to T * F & \texttt{\{ \$\$ = do\_times(\$1, \$3); \}} \\
\quad \mid F & \texttt{\{ \$\$ = \$1; \}} \\
F \to (E) & \texttt{\{ \$\$ = \$2; \}} \\
\quad \mid \text{id} & \texttt{\{ \$\$ = make\_var(\$1); \}} \\
\quad \mid \text{num} & \texttt{\{ \$\$ = make\_constant(\$1); \}}
\end{array}
$$

Your task is to use semantic actions to produce an AST for these expressions. In addition, you should simplify the AST using the following mathematical identities:

$$\text{num} + \text{num} = \text{num}$$
$$0 * \text{id} = \text{id} * 0 = 0$$

For example, the expression $(2+3)*x$ should be simplified to $5*x$ and $0*x$ to 0, while the expression $3+x+(-3)$ cannot be simplified under our rules.

The definition of the AST class `Expression` and associated constructors are given below:

```
class Expression {
private:
    // omitted
public:
    // returns `true` if this expression is an integer constant, and false otherwise
    bool is_constant();

    // returns the value of the constant, or 0 if the expression is not a constant
    int  value();
};
Expression *make_constant(int v);
Expression *make_var(Symbol *s);
Expression *make_plus(Expression *lhs, Expression *rhs);
Expression *make_times(Expression *lhs, Expression *rhs);
```

Implement `do_plus` and `do_times`.

Solution:

```
Expression *do_plus(Expression *lhs, Expression *rhs) {
    if (lhs->is-constant() && rhs->is_constant()) {
        return constant(lhs->value() + rhs->value());
    }
    return make_plus(lhs, rhs);
}
```

```
Expression *do_times(Expression *lhs, Expression *rhs) {
    if (lhs->is_constant() && lhs->value() == 0) {
        return make_constant(0);
    }
    if (rhs->is_constant() && rhs->value() == 0) {
        return make_constant(0);
    }
    return make_times(lhs, rhs);
}
```

We also allow answers that create a new method `Expression::is_var(Expression*)` to check if an expression is a a variable.