Midterm Examination

- Please read all instructions (including these) carefully.
- Please print your name at the bottom of each page on the exam.
- There are seven questions on the exam, each worth between 10 and 20 points. You have 1 hour and 20 minutes to work on the exam, so you should plan to spend approximately 12 minutes on each question.
- The exam is closed book, but you may refer to your two sheets of prepared notes.
- Please write your answers in the space provided on the exam, and clearly mark your solutions. You may use the backs of the exam pages as scratch paper. Please do not use any additional scratch paper.
- Solutions will be graded on correctness and clarity. There are no "tricky" problems on the exam—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.

NAME: Sample Solutions

SID or SS#: _____

Problem	Max points	Points
1	15	
2	15	
3	20	
4	10	
5	20	
6	10	
7	10	
TOTAL	100	

1. **Regular Expressions** (15 points)

Consider a language where real constants are defined as follows: A real constant contains a decimal point or E notation, or both. For instance, 0.01, 2.71821828, ~1.2E12, and 7E~5 are real constants. The symbol "~" denotes unary minus and may appear before the number or on the exponent. There is *no* unary "+" operation. There must be at least one digit to left of the decimal point, but there might be no digits to the right of the decimal point. The exponent following the "E" is a (possibly negative) integer.

Write a regular expression for such real constants. Use the standard regular expression notation described by the following grammar:

$$R \rightarrow \epsilon \mid \text{char} \mid R + R \mid R* \mid RR \mid (R)$$

You may define names for regular expressions you want to use in more than one place (e.g., $foo = R$).

$$\begin{aligned}
\texttt{digit} &= \texttt{0} + \texttt{1} + \texttt{2} + \texttt{3} + \texttt{4} + \texttt{5} + \texttt{6} + \texttt{7} + \texttt{8} + \texttt{9} \\
\texttt{posint} &= \texttt{digit}\ \texttt{digit}^* \\
\texttt{int} &= (\epsilon + \texttt{~})\ \texttt{posint} \\
\texttt{exp} &= \texttt{E}\ \texttt{int} \\
\texttt{frac} &= \texttt{.}\ \texttt{digit}^* \\
\texttt{real} &= (\texttt{int}\ \texttt{frac}\ (\texttt{exp} + \epsilon)) + (\texttt{int}\ (\texttt{frac} + \epsilon)\ \texttt{exp})
\end{aligned}$$

2. **Finite Automata** (15 points)

Consider a DFA with a start state s_0 and a transition function *trans*. For a state s and input character c, $trans(s, c) = s'$ if there is a transition from state s to state s' on character c. If there is no transition from s on c then $trans(s, c) = none$. The following algorithm simulates the behavior of such a DFA, *accepting* if the input string is accepted by the DFA and *rejecting* otherwise.

state $\leftarrow s_0$
while there's input left do:
    char $\leftarrow$ next input character
    if *trans*(*state*,*char*) = *none* then stop and *reject*
    state $\leftarrow$ *trans*(*state*,*char*)
(end of loop)
accept if *state* is a final state, otherwise *reject*

Now consider an NFA with a start state s_0 and a transition function *trans*. In this case, for a state s and input character c (we now allow $c = \epsilon$), $trans(s, c)$ is the set of states s' for which there is a transition from s to s' on c. In the style of the algorithm above, give a (deterministic) algorithm that simulates the behavior of such an NFA. You may use the ϵ-*closure* operation described in class and in the text.

```
Simulate subset construction.  Use state-set instead of state.

    state-set ← ε-closure(s0)
    while there's still input left do:
        char ←  next input character
        if ε-closure(⋃_{s∈state-set} trans(s, char)) = ∅ then stop and reject
        state-set ← ε-closure(⋃_{s∈state-set} trans(s, char))
    (end of loop)
    accept if there's a final state in state-set, otherwise reject.
```

3. **Grammars** (20 points)

Consider the following grammar. The nonterminals are E, T, and L. The terminals are +,**id**,(,), and ;. The start symbol is E.

$$\begin{aligned} \mathrm{E} &\rightarrow \mathrm{E} + \mathrm{T} \mid \mathrm{T} \\ \mathrm{T} &\rightarrow \mathbf{id} \mid \mathbf{id}() \mid \mathbf{id}(\mathrm{L}) \\ \mathrm{L} &\rightarrow \mathrm{E}; \mathrm{L} \mid \mathrm{E} \end{aligned}$$

Give an LL(1) grammar that generates the same language as this grammar. As part of your work please show that your grammar is LL(1).

(a) Eliminate left recursion:

$$\begin{aligned} \mathtt{E} &\rightarrow \mathtt{TE'} \\ \mathtt{E'} &\rightarrow +\mathtt{TE'} \mid \epsilon \\ \mathtt{T} &\rightarrow \mathtt{id} \mid \mathtt{id}() \mid \mathtt{id}(\mathtt{L}) \\ \mathtt{L} &\rightarrow \mathtt{E}; \mathtt{L} \mid \mathtt{E} \end{aligned}$$

(b) Left factor:

$$\begin{aligned} \mathtt{E} &\rightarrow \mathtt{TE'} \\ \mathtt{E'} &\rightarrow +\mathtt{TE'} \mid \epsilon \\ \mathtt{T} &\rightarrow \mathtt{id}\,\mathtt{T'} \\ \mathtt{T'} &\rightarrow \epsilon \mid (\mathtt{T''} \\ \mathtt{T''} &\rightarrow) \mid \mathtt{L}) \\ \mathtt{L} &\rightarrow \mathtt{EL'} \\ \mathtt{L'} &\rightarrow ; \mathtt{L} \mid \epsilon \end{aligned}$$

(c) Check that it's LL(1). For this part you just needed to give enough information to show that there would be no conflicts in the parsing table. The following is sufficient:

$$\begin{array}{lcll} \mathtt{E'} \rightarrow +\mathtt{TE'} \mid \epsilon & & \mathtt{First}(+\mathtt{TE'}) & = \{+\} \\ & & \mathtt{Follow}(\mathtt{E'}) & = \{\$,), ;\} \\ \mathtt{T'} \rightarrow \epsilon \mid (\mathtt{T''} & & \mathtt{First}((\mathtt{T''}) & = \{(\} \\ & & \mathtt{Follow}(\mathtt{T'}) & = \{+, \$,), ;\} \\ \mathtt{T''} \rightarrow) \mid \mathtt{L}) & & \mathtt{First}()) & = \{)\} \\ & & \mathtt{First}(\mathtt{L})) & = \{\mathtt{id}\} \\ \mathtt{L'} \rightarrow ; \mathtt{L} \mid \epsilon & & \mathtt{First}(;\mathtt{L}) & = \{;\} \\ & & \mathtt{Follow}(\mathtt{L'}) & = \{)\} \end{array}$$

4. **Parsing** (10 points)

In both parts of this question, we are looking for clarity and brevity as well as the right idea. Suppose you are writing a parser for a programming language that includes the following syntax for looping constructs:

$$\begin{aligned} \text{Loop} \rightarrow\ & \textbf{do}\ \text{stmt}\ \textbf{while}\ \text{expr} \\ |\ & \textbf{do}\ \text{stmt}\ \textbf{until}\ \text{expr} \\ |\ & \textbf{do}\ \text{stmt}\ \textbf{forever} \end{aligned}$$

(a) (5 points) A predictive parser (i.e., a top-down parser with no backtracking) can't use this grammar. Give a *brief* (a couple of sentences) explanation of this fact.

When trying to expand a production for non-terminal ``loop'', the parser cannot decide which of the three productions to expand using only the next few input tokens.

(b) (5 points) Give a brief explanation of why a bottom-up parser does not have difficulty with this grammar.

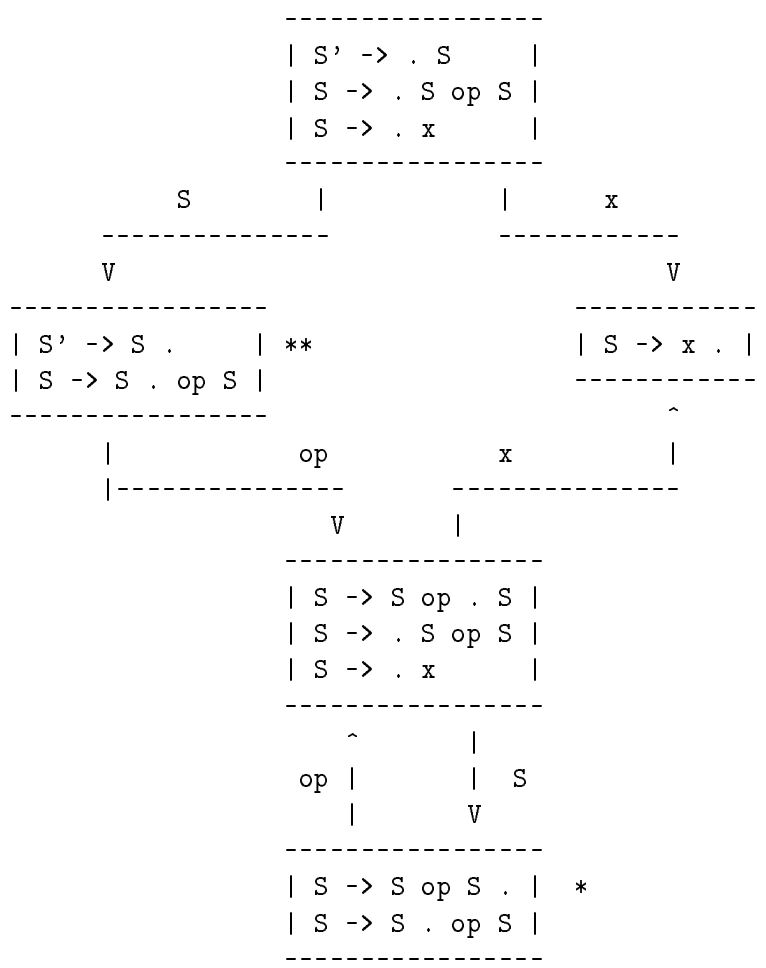
When a bottom-up parser must decide which of the three productions to choose (reduce), the ``while'', ``until'', or ``forever'' has already been read and shifted onto the stack.

5. **Parsing** (20 points)

Consider the following grammar. The nonterminals are S′ and *S*. The terminals are **op** and **x**. The start symbol is S′.

$$\begin{aligned} S' &\rightarrow S \\ S &\rightarrow S \ \mathbf{op} \ S \mid \mathbf{x} \end{aligned}$$

(a) (15 points) Draw the DFA built from sets of LR(0) items for this grammar. Show the contents of each state. (Note: Don't augment the grammar with a new start symbol.)



(b) (3 points) Is this grammar SLR(1)? Briefly explain why or why not.

No. The grammar is ambiguous. For example, there are two parses of the string ``x op x op x''.

An alternative justification is that there is a shift/reduce conflict in state *. Note that there is no shift/reduce conflict in the state **.

(c) (2 points) Is this grammar LR(1)? Briefly explain why or why not.

No. The grammar is ambiguous.

6. **Bison and Abstract Syntax Trees** (10 points)

Consider the following constructors for a tree language:

```
Expression app(Expression, Expression);
Expression lambda(Expression, Expression);
Expression id();
```

Now consider the following Bison grammar:

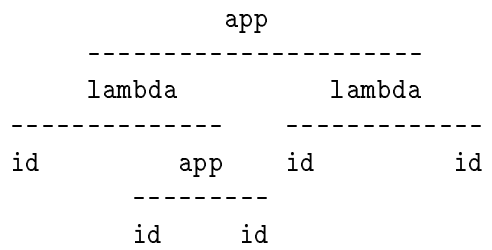
```
%token ID LAMBDA
%type  <Expression> Expr

%%

Expr : ID
        { $$ = id(); }
     |  '(' LAMBDA ID '.' Expr ')'
        { $$ = lambda( id(), $5); }
     |  '(' Expr Expr ')'
        { $$ = app($2, $3); }
```

Draw the abstract syntax tree that would be produced when parsing the sequence of tokens below. Label all the nodes of your AST with the appropriate constructor.

```
( ( LAMBDA ID . (ID ID) ) ( LAMBDA ID . ID ) )
```



7. **Type Checking** (10 points)

Suppose we want to design a type checker for Scheme programs. In Scheme, functions can be passed as arguments to and returned as results from functions. Recall the type checking rule for function application given in class:

$$\frac{A \vdash f : t_1 \rightarrow t_2 \quad A \vdash x : t_1}{A \vdash f(x) : t_2}$$

Self-application occurs when a function is called with itself as a parameter. This is, if f is a function taking a function as a parameter, the $f(f)$ is an instance of self-application. Briefly explain why type checking of self-application must always fail using the type checking rule above.

```
From the rule, in a self-application f(f) we know that:

      f : t1 -> t2
      f : t1

But t1 is a proper subexpression of t1 -> t2, so t1 <> t1 -> t2.  Since
f cannot have two disinct distinct types, type checking fails for self
application.
```

An alternative (but equivalent) answer is:

```
If f : t1 -> t2 and f : t1 then t1 = t1 -> t2 since f has one type.
But then

t1 = t1 -> t2 = (t1 -> t2) -> t2 = ((t1 -> t2) -> t2) -> t2 =
... -> t2 -> t2 -> t2

which is infinite.  Since types are finite size, type checking must fail for
self-application.
```