CSE 244 Midterm Exam – Fall 2000

Name: ___________________________________________

<table>
<thead>
<tr>
<th>Number</th>
<th>Points</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>120</td>
<td></td>
</tr>
</tbody>
</table>

Use only one side of the paper and start each problem on a new page!!
Please show all work to receive ANY credit!!!!

1. (10 points) Context-Sensitive Grammars and Derivations

In a CFG, the left-hand-side of a production rule must be a non-terminal. However, in a context-sensitive grammar (CSG), the left-hand-side of each production rule can be a sequence of one or more non-terminals (at least one) and terminals (zero or more) as shown below:

\[ S \rightarrow x S B C \quad C B \rightarrow B C \quad y C \rightarrow y z \]
\[ S \rightarrow x y C \quad y B \rightarrow y y \quad z C \rightarrow z z \]

Note that \(x, y,\) and \(z\) are terminals, and \(S, A, B,\) and \(C\) are non-terminals. In a CSG, during a single step of the derivation, two symbols (say \(y B\)) can be replaced by two symbols (say \(y y\)) for the production rule \(y B \rightarrow y y\), which is unlike a CFG where one symbol (always a non-terminal) can be replaced by zero or more non-terminals or terminals. This is shown in the step of the derivation below:

\[ \alpha y B \beta \rightarrow \alpha y y \beta \]

where \(\alpha\) and \(\beta\) are strings of terminals and non-terminals. Using this grammar, develop a derivation for the string: \(xxx yyyy zzz\).
2. (15 points, 5 points each) Compiler Concepts and Definitions

Choose 3 of the 5 concepts given below and provide:

i). a definition of the concept

ii). a discussion of why the concept is important to the design and implementation of a compiler

iii). an example of the concept

We will only grade three, so there’s no advantage to answering more! Circle the letters in the list of a to e given below for the three concepts that you want me to grade.

[a.] ambiguity in CFGs  [b.] block-buffered I/O  [c.] sentential form
[d.] left factoring      [e.] parse tree
3. (20 points) Regular Expressions, Languages, and CFGs

For parts a and c., you may only use the operators: +, *, |, ?, (), and concatenation to construct regular expressions.

(a) (5 points) Write a regular expression for the language of all strings of {0, 1} with either an even number of zeros (at least two) or an odd number of ones (at least one).

(b) (5 points) Describe using one or two prose sentences, the language that is represented by the regular expression:

\[(s? (he)) | (h ((i(s|m)) | (er(s?))))\]

(c) (10 points) In class, a conversion algorithm from a FSM (DFA or NFA) to a context free grammar was examined. In this problem, do the reverse by converting the CFG given below into an equivalent regular expression. Note that all upper case are non-terminals with lower case tokens.

\[
\begin{align*}
A & \rightarrow a \ B \\
A & \rightarrow b \ A \\
B & \rightarrow a \ C \\
B & \rightarrow c \ D
\end{align*}
\]

\[
\begin{align*}
C & \rightarrow a \ B \\
C & \rightarrow c \\
D & \rightarrow x \\
D & \rightarrow y
\end{align*}
\]

Hints: Construct a FSM before developing the regular expression. In the construction process, a rule such as C \rightarrow c would introduce a transition in a FSM from the C state to a final state.
4. (20 points) CFGs, Left Recursion, and $\epsilon$ Moves

As the first step in the design of a top-down parser, reformulate the grammar given below into an equivalent CFG that is suitable for top-down parsing by first removing any left recursion that exists, and then removing all $\epsilon$ moves.

\[
\begin{align*}
S & \rightarrow x \ A \quad B \rightarrow S \ x \\
S & \rightarrow B \ y \quad B \rightarrow A \ y \\
A & \rightarrow B \ x \quad B \rightarrow z \\
A & \rightarrow S \ y
\end{align*}
\]

In the grammar, $x$, $y$, and $z$ are terminals, and $S$, $A$, $B$, and $C$ are non-terminals. Make sure that you clearly indicate your results separately, i.e., box the CFG without left recursion with $\epsilon$ moves, and box the result after $\epsilon$ moves have been eliminated.
5. (20 points) FIRST and FOLLOW for CFGs

Consider the grammar given below, with $S$ as the start symbol, $X$, $Y$, and $Z$, as non-terminals, $a$, $b$, $c$, $d$, $f$, and $g$ as terminals, and $\epsilon$ as the empty symbol.

$$
S \rightarrow X \ Y \ Z \\
X \rightarrow a \mid Z \ b \mid \epsilon \\
Y \rightarrow c \mid d \ X \ Y \mid \epsilon \\
Z \rightarrow f \mid g
$$

(a) (14 points) Compute FIRST and FOLLOW for each non-terminal in the grammar.
(b) (6 points) In calculating FIRST and FOLLOW, we have been discussing LL(1) grammars. The number '1' indicates the lookahead, and in this case, means that we use the current input symbol as the lookahead in conjunction with the top element of the parsing stack to determine the next parsing action (via the parsing table). If we wanted to develop an LL(2) grammar, we would use both the current input symbol and the next input symbol as lookaheads. In this case, when calculating first, instead of determining a set of single terminals, we would have to find possibly pairs of terminals. Thus, define FIRST2(A) for the non-terminal A to be the first, zero, one, or two terminals that A turns into during a derivation. For example, given the grammar above,

\[
\text{FIRST2}(X) = \{ a, fb, gb, \epsilon \}
\]

where fb and gb are in FIRST2(X) as the result of the rule \( X \rightarrow Z b \). Using the calculation techniques for FIRST and this example, determine: FIRST2(Y) and FIRST2(S) for the same grammar below:

\[
\begin{align*}
S & \rightarrow X Y Z & X & \rightarrow a \mid Z b \mid \epsilon \\
Y & \rightarrow c \mid d X Y \mid \epsilon & Z & \rightarrow f \mid g
\end{align*}
\]
6. (35 points) Design of a Context Free Grammar

In the ML functional programming language, tuples and lists are two data structuring techniques. Tuples are represented by parenthesized expressions that contain two or more entries, and are similar to defining a record in a language like Pascal. For example:

- \((4, 5.0, "six")\) -- a record with an integer, real, and string data
- \((1, 2, 3, 4)\) -- a record that contains four integers
- \((1, (2, 3.0))\) -- a record containing an integer and a record

As shown below, `val` declares variables with expressions referenced via a `#` notation.

```ml
val t = (4, 5.0, "six"); -- declare the variable t as a tuple
val s = (1, (2, 3.0)); -- declare the variable s as a tuple
val r = (1, 2.0, "3"); -- declare the variable r as a tuple
#3(t); -- access the third expression in t, namely the string "six"
#2(s); -- access the second expression in s, namely the record (2, 3.0)
```

Lists in ML are sequences of one or more list elements, which must all be of the same type, that are enclosed in square brackets with elements separated by commas. For example:

```ml
val L = [1, 2, 3]; -- an integer list with three elements
val N = ["one", "two"]; -- a string list with two elements
val P = [(1, 2), (3,4)]; -- a list containing two tuples
val Q = [r, t]; -- a list containing two tuples
```

Lists are accessed in ML by applying the function `hd` to return the first list element and `tl` to return a list containing all elements except the first one, e.g.,

```ml
hd(L); -- returns the integer 1
tl(L); -- returns the list [2, 3]
hd(P); -- returns the tuple (1, 2)
tl(Q); -- returns the list [t] which is [(4, 5.0, "six")]
hd(tl(L)); -- returns the integer 2 -- note that combinations are possible
tl(tl(L)); -- returns the list [3] -- applied in a nested fashion
```

Note also that an expression in a tuple may be a list, e.g., `val x = (1, [2,3], 4);`.

Assume that tokens have been defined for: `INT`, `REAL`, `STR`, `IDENT`, `HD`, `TL`, `VAL`. Design a CFG for ML programs that contain a one or more `val` declarations followed by one or more `val` statements to access tuples or apply functions. For example:

```ml
val t = (4, 5.0, "six");
val L = [1, 2, 3];
#1(t);
tl(L);
hd(tl(L));
```

Your task in this problem is to design a CFG that can recognize the various declarations and statements previously described, thereby allowing a limited set of ML programs to be recognized.
To get you started, the first two grammar rules are given below:

```
ml_program    -->  val_declarations  statement_list
val_declarations --> val_declarations  one_declaration  |  one_declarations
one_declaration -->  ???
list_or_tuple  -->  ???
statement_list -->  ???
```