Final Exam
CS400 Spring 2013 NAME:________________________

General Instructions:

Write clearly and legibly. If your handwriting (i.e., cursive) isn't clear and legible, then print!

Partial credit will be given when and where work is shown that makes the line of reasoning evident. If no work is shown, then no partial credit will be awarded. Note that work shown on one problem will not be considered for determining partial credit on another problem.

Unless otherwise indicated, the following apply to grammar rules:

1. Non-terminals are surrounded by angle brackets
2. Terminals are not surrounded by angle brackets.
3. Both BNF and Extended BNF forms may be used.

GRADING USE ONLY

PART 1:	______/ 100
PART 2:	______/ 25
PART 3:	______/ 50
PART 4:	______/ 25
TOTAL:	______/ 200

PART I – Multiple Choice

Choose the **best** option from the list below and enter the corresponding letter designation in the space provided **on this page**. Each answer may be used zero, one, or more times. Keep in mind that the question may or may not represent a defining relationship to the answer; in other words, the relationship may not be complete but, rather, more of an 'example' type relationship.

A. Closure
B. Lexeme
C. Syntactical analysis
D. First class functions
E. Currying
F. Right-most derivation
G. Token
H. Sentence
I. Parser
J. Lexer
K. Finite automaton
L. Regular grammar
M. Context-free grammar
N. Pushdown automaton
O. Production rule
P. Higher order function
Q. Lexical scope
R. Regular expression
S. Pairwise-disjoint
T. Alphabet
U. Lexical analysis
V. Referential transparency
W. Semantics
X. Grammatical ambiguity
Y. Left-most derivation
Z. Dynamic scope

1)	6)	11)	16)	21)
2)	7)	12)	17)	22)
3)	8)	13)	18)	23)
4)	9)	14)	19)	24)
5)	10)	15)	20)	25)

PART 1 (cont'd)

1. The set of all ASCII codes that could be present in a program's source code file.
2. When the referencing environment for a variable depends on the stack of functions that called the function containing the variable.
3. A parser generally works with this type of grammar.
4. When a function captures the references to variables that are within its referencing environment at the time the function is created.
5. Describes the meaning of sentence fragments.
6. When any expression may be replaced by any other expression that produces the same value.
7. A processing engine that reads a source code file and produces a string of tokens.
8. Used to describe/define a regular language.
9. A terminal in a grammar.
10. The type of grammar recognized by most lexers.
11. When the referencing environment for variables is determined by the structure of the source code only.
12. Defines the options that may be used to replace each non-terminal in a grammar.
13. A function that can accept functions as arguments, return values, or both.
14. When no two production rules for the same non-terminal can produce the same initial terminal.
15. When a sentence is produced by always expanding the first non-terminal in the sentence.
16. The process of examining the source code to determine if it represents valid tokens in a grammar.
17. The process of examining the tokens in a sentence and deciding if they form a valid sentence.
18. A language is said to have these if functions can be treated like any other value, including being passed as arguments, returned by functions, and assigned to variables.
19. Bottom-up parsers work with this type of derivation.
20. The machine capable of recognizing any context-free grammar.
21. A program that examines a string of tokens to determine a sentence's structure.
22. A machine that can recognize any regular grammar.
23. One or more symbols from the source code alphabet that, together, have a specific meaning.
24. The process of producing a function that wraps another function is such a way that the number of arguments needed is reduced.
25. When two different derivations of the same type produce the same sentence.

PART 2

Consider an unambiguous context-free grammar that contains the three tokens '**#**', '**$**', and '**&**'. The '**#**' token might be a literal or a variable name and can be used as an operand, while the '**$**' and '**&**' are binary infix operators that can produce values that can be used as operands. The '**&**' operator has precedence over '**$**'. The '**$**' operator is right associative while the '**&**' operator is left associative. There are two non-terminals in the grammar, **`<start>`** and **`<other>`.** The **`<start>`** non-terminal is the Start Symbol. Recall that "infix" merely means that the operator appears between the two operands.

2.1 (10pts) Which of the following productions are consistent with this grammar?
Check all that apply – 1pt for each correctly checked/unchecked item.

____ `<start> => <other> $ <start>`	____ `<other> => <other> & #`
____ `<start> => <start> $ <other>`	____ `<other> => # & <other>`
____ `<start> => <other> & <start>`	____ `<start> => #`
____ `<start> => <start> & <other>`	____ `<other> => <other> $ #`
____ `<start> => <other>`	____ `<other> => #`

2.2 (15pts) Each of the expressions below presumably represents the natural grouping of a grammar similar to the one above, meaning that the grouping matches the order of evaluation of the expression **`a & b $ c $ d & e & f $ g`** in that grammar. For each expression determine which operator has higher precedence and also what the associativity is for each operator. Circle the appropriate symbol, choosing the question mark if the answer cannot be determined from the expression, either because the needed information is not there or because it is contradictory.

Expression	Order	&	$
`(((a & b) $ c) $ ((d & e) & f)) $ g`	& $?	L R ?	L R ?
`a & ((b $ (c $ d)) & (e & (f $ g)))`	& $?	L R ?	L R ?
`(a & b) $ (c $ (((d & e) & f) $ g))`	& $?	L R ?	L R ?
`(a & (b $ (c $ d))) & (e & (f $ g))`	& $?	L R ?	L R ?
`a & ((b $ c) $ ((d & e) & (f $ g)))`	& $?	L R ?	L R ?

PART 3 - (5 pts each)

3.1) Why shouldn't the variable yytext be referenced in the Bison input file?

3.2) Why are stack-dynamic variables generally referred to as "automatic" variables.

3.3) What is the difference between a formal parameter and an actual parameter?

3.4) In C, the statement **`printf("%f", x);`** works the same way regardless of whether x is a float or a double. Why?

3.5) In C, when an array is passed to a function and the formal parameter is not declared merely as a pointer to the type of data stored in the array, the declaration must include the number of elements in each dimension except one, which may be left blank. Which dimension is optional and why is it not necessary?

3.6) What is printed by the following C statement:

```
printf("%s", &(6+5)["Hello World"]-4);
```

3.7) In C, how are arguments corresponding to an ellipsis as the formal parameter list handled?

3.8). What is wrong with the following snippet of C code?

```
     char *string;
     string = "Fred";
     string[2] = 'a';
```

3.9). What is wrong with the following snippet of C code?

```
     char *string;
     string = (char *) malloc( strlen("Fred")*sizeof(char));
```

3.10). What is wrong with the following snippet of C code?

```
     char *myfunction(void)
     {
          char string[12];
          strcpy(string, "Fred");
          return string;
     }
```

PART 4 (5 pts each)

4.1) What is meant by a “dangling pointer” and by a “memory leak”? What problems can each cause?

4.2) What is meant if a language is said to exhibit “guaranteed short circuiting”?

4.3) What is a “widening” conversion and what is a “narrowing” conversion?

4.4) What does a “referencing environment” refer to?

4.5) What is the basic requirement that must be met in order for an expression to be used as an lvalue (and do not just say that it has to be able to appear on the left side of an assignment operator)?