0	00/
1	20/
2	05/
3	05/
4	20/
5	30/
6	20/
7	15/

	115/

CMSC 331 Midterm Exam
Section 0101 – Oct 8, 2002

Name: ____________Sample Answers________
Student ID#:______________________________

You will have seventy-five (75) minutes to complete this closed book exam. The points for each problem are given and they sum to 115. Use the backs of these pages if you need more room for your answers. Describe any assumptions you make in solving a problem. We reserve the right to assign partial credit, and to deduct points for answers that are needlessly wordy.

0. Warm up (0)

In a rectangular array of students, which will be taller, (a) the tallest of the shortest student in each column, or (b) the shortest of the tallest student in each row?

The shortest of the tallest people in each row will be taller than, or the same height as, the tallest of the shortest people in each column. There are four cases. The first is that the shortest of the tallest and the tallest of the shortest are the same person, so obviously in this case the shortest of the tallest and the tallest of the shortest would be the same height.

The second case is that the shortest of the tallest and the tallest of the shortest are in the same row. The shortest of the tallest people in each row is obviously the tallest person in his row, so he's taller than the tallest of the shortest, who is also in his row.

The third case is that the shortest of the tallest and the tallest of the shortest are in the same column. The tallest of the shortest people in each column is obviously the shortest person in his row, so he's shorter than the shortest of the tallest, who is also in his column.

The fourth case is that the shortest of the tallest is neither in the same column nor the same row as the tallest of the shortest. For this case, consider the person X who is standing in the intersection of the row containing the shortest of the tallest and the column containing the tallest of the shortest. X must be taller than the tallest of the shortest, since the tallest of the shortest is the shortest in his column, and X must also be shorter than the shortest of the tallest, since the shortest of the tallest is the tallest in his row. So TofS < X < SofT.

So the shortest of the tallest in each row is always taller than, or the same height as, the tallest of the shortest in each column.

1. True/False (20)

1. __F___ Most programming languages are implemented as interpreters these days.
2. __T___ Given an ambiguous grammar G, every string is either (a) not in the language defined by G or (b) is in the language but has more than one parse tree that can be assigned to it.
3. ___F__ Every BNF grammar G that has left recursion can be converted into a grammar G' that recognizes exactly the same language and has no recursive rules in it.
4. ___F__ Some grammars that can be defined using extended BNF (EBNF) notation can not be defined using simple BNF notation.
5. __F___ Static type checking adds some execution-time overhead, but improves reliability of programs.
6. __F___ Every attribute grammar can be rewritten as a context free grammar if recursive rules are allowed.
7. ___F__ If two languages have different non-terminal symbols, and productions, the languages can not be identical.

8. __T__ In an attribute grammar, a node's inherited attributes can be based on the values associated with that node's ancestors **and** siblings in the parse tree, but not its descendants.
9. __F__ Axiomatic semantics provide a good way of documenting the design decisions made in implementing a program.
10. __F__ A recursive descent parser for a language can not be based directly on a grammar for the language which does not have any recursive rules.
11. __F__ A programming language is considered to be strongly typed if and only if all type errors are detected statically by the compiler.
12. __F__ Most modern programming languages use dynamic scoping for variables
13. __F__ Enumeration types are usually implemented as bit strings.
14. __T__An associative array in an unordered collection of data elements indexed by an equal number of values called keys.
15. __F__ The Unix yacc program takes a set of grammar rules and produces a C program implementing a top down recursive descent parser for the language defined by that grammar.
16. __T__ Misuse of union types is a source of errors in some programming languages.
17. __F__ Automatic memory management is incompatible with free union data types.
18. __T__ A variable's type constrains the values it can take on and also the kinds of operations that can be applied to it.
19. __F__ A variable's scope is the time during which the variable is bound to a storage location.
20. __F__ Static variables can not change their values.

2. Orthogonality (5)

What is **orthogonality** in the context of programming language design? Give an **example** from a real programming language. How does orthogonality relate to the design principle of **reliability**?

Orthogonality in a programming language refers to the property that language features can be combined freely. The primitive operators, functions, procedures and data types should be able to be composed with few restrictions. Languages with a high degree of orthogonality often have a relatively small set of primitive components and achieve a high degree of expressiveness though composition. One simple example of othogonality is operator overloading. The + operator can be defined to work on all kinds of numbers as well as arrays of numbers of all dimentionality.

3. Arrays (5)

What does this FORTRAN 90 array access notation mean?

```
MAT( 1:3, 2 )
```

This specifies an array slice. Arrays are 1-based by default in Fortran 90. This expression asks for a slice of a 2-d array. The first component says "rows 1-3". The second component says "column 2".

4. Grammars I (20)

Consider the following grammar where uppercase letters indicate non-terminals and lowercase letters indicate terminals. Which of the following sentences are in the language defined by this grammar? If the sentence is in the language, show a parse tree.

```
S -> a S c B | A | b
A -> c A | c
B -> d | A
```

(a) acccbd -- This is not in the language.

(b) aabcdcd – S(a,S(a,S(b),c,B(d)),c,B(d))

(c) acd -- This is not in the language.

(d) accc – S(a,S(A(c)),c,B(A(c)))

5. Grammars II (30)

Cadawack, a computer aided design (CAD) company, has a highly successful analog circuit simulation package. They aim to expand into the very lucrative hybrid circuit market with a digital logic extension to their tool. Cadawack has hired you to write the front-end for their new tool. Your new supervisor, Wally, hands you the following BNF grammar for an initial prototype of the tool. Instantly you realize that you cannot implement the language, as specified.

```
/* <connection> is the start symbol */
<connection> ::= <signal> = <circuit>
<signal> ::= A | B | C | ... | Z
<source> ::= 0 | 1
<circuit> ::= <circuit> and <circuit>
    | <circuit> or <circuit>
    | not <circuit>
    | ( <circuit> )
    | <signal>
    | <source>
```

(a) In a sentence or two, explain to Wally why this grammar should not be implemented as written. (5)

The grammar is ambiguous. More than one structure and interpretation can be given to a string.

(b) Wally still doesn't understand, so illustrate your point, using the following string for a digital circuit as your example: *A = B and C or D*. Draw any figures that you might use in your explanation. (10)

Arrrgh. Ok, Wally, consider the input *A = B and C or D*. The grammar you wrote can parse this simple expression in two ways, each of which has a different meaning:

A = (B and (C or D))
A = ((B and C) or D)

We should design a language so that every legal expression has one and only one meaning. Otherwise our users will write expression expecting one meaning and may end up having the Cadawack design tool select the other meaning. We could get sued for selling a defective product. Moreover, this deficiency would be so lame that no weasel words in the EULA would save you sorry behind.

Panicking about his commitment to demonstrate the prototype at the next industry trade show, Wally promises you an extra week's vacation for eliminating the problems in the grammar. You rewrite the grammar to fix the problem while daydreaming about Miami.

(c) What did you assume about precedence and associativity for the revised grammar. (5)

We'll assume that AND should bind more tightly than OR, so that the *A=B and C or D* should be parsed as *A = (B and C) or D*. The AND operator should bind less tightly than NOT and parentheses. Both AND and OR will associate to the left, so *A and B and C* will parse as *(A and B) and C*.

(d) Write a revised grammar that eliminates the problems. (10)

This is just a variation on the simple expression grammar we saw so much of in class and in the text. We need to introduce extra non-terminals <cf> (like factors) and <ct> (like terms).

```
<connection> ::= <signal> = <circuit>
<signal> ::= A | B | C | ... | Z
<source> ::= 0 | 1
<circuit> ::= <ct>  | <circuit> and <ct>
<ct> ::= <cf> | <ct> or <cf>
```

<cf> ::= not <cf> | (<circuit>) | <signal> | <source>

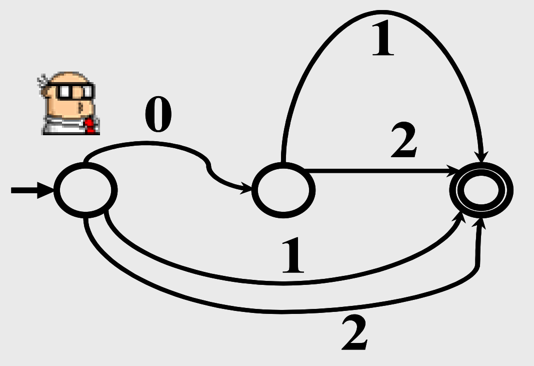
6. Regular expressions and lex (20)

Wally asks you to develop a **regular expression** (RE) to match a date string for any date from January 1, 1900 to December 31, 2002 using the familiar notation like *10/8/2002*. Here is the specification he gives:

> A date is a month followed by a slash followed by a day followed by a slash followed by a year. A month is an integer between 1 and 12. If the integer is a single digit, it can be preceded by an optional zero (e.g., *02/12/2002*). A day is an integer between 1 and 31 and, if a single digit, can be preceded by an optional zero (e.g., *10/08/2002*). A year will be an integer between 1900 and 2002, inclusive.

You decide not to remind him that some months have fewer than 31 days. Strings your RE should match include 01/01/1900, 1/1/1900, 12/31/2002, 1/01/2000, 02/30/1984, 1/23/1945, and 10/8/2002 and strings that shouldn't match include 00/01/1900, 1/1/1899, 13/01/2000, and 10/8/02. Note that you should develop an RE and **not** a context free grammar.

(a) (10) Show the RE as a deterministic finite state machine, e.g. one with no epsilon arcs. Show the FSM as a graph with one start state (with an incoming arrow from nowhere) and one or more accepting states (shown as doubled circles). Arcs between states should be labeled with input characters. See example to the right.



A finite state machine for the regular expression 0?[12]

Ok. So the graph of this FSM is a bit ugly because of my requirement that all arcs only have a single input character on them. But, it's not hard to write out. I've sketched it and will scan it in at some point and put it here. See the graph at the end of this exam.

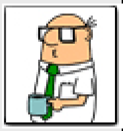
(b) (10) Give the RE as a pattern using the lex notation. If *e* is a regular expression, recall that *e?* means zero or one occurrences of *e*, *e** means any number of occurrences, and *e+* means one or more occurrences. Parentheses are used freely to clarify the expression. Alternatives are specified with |, so (foo|bar) matches either "foo" or "bar". Square brackets designate a set of alternatives, so [0123456789] matches any single digit. Character ranges can be used, so [a-zA-Z] matches a lower or upper case alphabetic character.

(0?[1-9]|1[0-2])/(0[1-9]|[12][0-9]|3[01])/19[0-9]|200[012]

7. Variable Scope (15)

Wally read an article in Dr. Dobbs about static and dynamic scoping in programming languages and decided that Cadawack should develop a new programming language D that would be just like C except it uses dynamic scoping instead of static scoping. To demonstrate his programming prowess, Wally spent all weekend producing the program to the right.

```
#include <stdio.h>
int i = 1;
int g(void) {
  if (i != 1) i=3;
  return i;
 }
int f(void) {
  int i=2;
  return g();
 }
int main(int argc, char **argv) {
  printf("%d", f());
 }
```



(a) When Wally's program is compiled with a standard C compiler (which uses static scoping) what output is produced? Briefly explain how the output is calculated. (5)

The program prints "1". Main calls f which calls g. Since C uses static scoping, the compiler attempts to resolve a reference to a non-local variable by looking in the environment in which the block is defined. So, any non-local variable references in g are resolved by looking at the global environment. Inside g, the variable i refers to the i which is declared globally, so it has the value 1.

(b) When it's compiled with a D compiler (which uses dynamic scoping) what output is produced? Briefly explain how the output is calculated. (5)

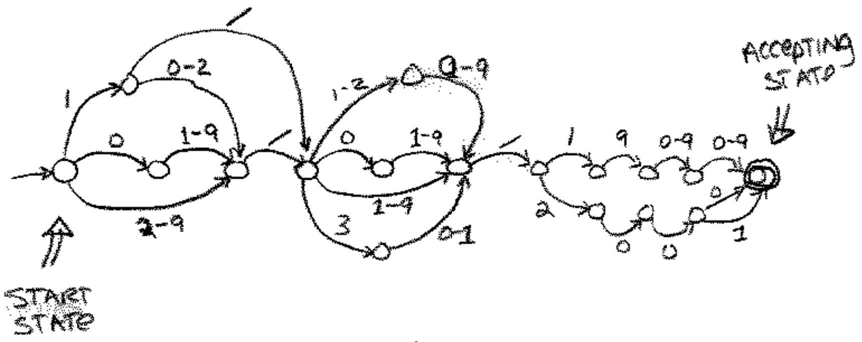
The program prints "3". Main calls f which calls g. Since D uses dynamic scoping, the compiler attempts to resolve a reference to a non-local variable by looking in the environment in which the block is called. So, any non-local variable references in g are resolved by looking first in the environment of F and then in the environment of main and then in the global environment. Inside g, the variable i refers to the i which is declared in f, so it has the value 2 And g then returns 3, which is also returned by f.

(c) Wally wonders if dynamic scoping increase or decrease a programming language's readability? Briefly explain the situation to him, using small words and simple sentences. (5)

Dynamic scoping is thought to reduce readability compared to static or lexical scoping. One can resolve references to non-local variables in programs written in a statically scoped language by examining the text of the program. In a dynamically scoped language, resolving references to non-local variables require that the reader determine who will call the block in which they are found. In general, this requires the reader to reason about the execution of the program. In extreme cases, it may require the reader to completely simulate the execution of the program.

T. Finin
10/9/02

CMSC 331 0101
MIDTERM
PROBLEM 6A



note: an arc like $\xrightarrow{n-m}$

stands for a set of m-n+1 arcs like