0	00/
1	40/
2	10/
3	20/
4	25/
5	20/
6	35/
7	30/
8	15/
	200/

CMSC 331 First Midterm Exam Fall 2005

Name: _____

MyUMBC username:_____

You will have seventy-five (75) minutes to complete this closed book exam. 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 pts)

Rick Hilton tells his two daughters, Paris and Nicky, to race their SUVs across the county to Las Vegas. The one whose SUV arrives last will inherit the Hilton fortune. The Hilton sisters, after driving around aimlessly for days, ask the spirit of their great-grandfather Conrad what they should do. After hearing his other-worldly advice they immediately jump in the SUVs and race as fast as they can toward Las Vegas. What did the ghost of Conrad Hilton say? **Conrad's ghost hissed "switch vehicles".**

1. True/False (40 pts)

For each of the following questions, circle T (true) or F (false).

T F COBOL is most often used for Artificial Intelligence work. FALSE
T F FORTRAN was developed primarily for applications in science and engineering. TRUE
T F Prolog is a language based on logic. TRUE
T F Ada was an early object-oriented language. FALSE
T F JavaScript is a portable subset of the Java programming language. FALSE
T F While every BNF grammar can be rewritten in EBNF, not every EBNF grammar can be rewritten in BNF. FALSE
T F Attribute grammars are more powerful than context free grammars because they support iteration in addition to recursion. FALSE
T 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 left recursive rules in it. TRUE
T F A grammar G is ambiguous if there is more than one parse tree for at least one sentence in the language defined by G. TRUE
T F If two languages have the same non-terminal symbols and different grammar rules, the languages can not be identical (e.g., have the same strings in them). FALSE
T F Operator precedence rules determine which operators appear lower or higher in a parse tree. TRUE
T F A grammar with left recursive rules can not be directly used to implement a recursive descent parser. TRUE
T F Attribute grammars associate pre-conditions and post-conditions with each statement in a program. FALSE
T F A reserved word in a programming language can not be used by a programmer as the name of a variable or procedure. TRUE
T F Simple phrases are phases with no non-terminal symbols in them. FALSE
T F Implicit variable declarations provide a way for a compiler to associate a variable with a type automatically. TRUE
T F Static variables are a way to define typed constants for a program. FALSE
T F A strongly typed programming language is one that detects all type errors either at compile time or a run time. TRUE

T F Lexical scanners are usually specified using regular expressions. TRUE
T F Every non-deterministic finite automaton can be rewritten as a deterministic finite automaton. TRUE

2. Short answers (10 pts)

Give short (one to three sentences) answers to the following questions.

(a) Identify two characteristics of a grammar that make it impossible or very difficult to directly use in a top down parser, like a recursive descent parser.

A grammar that has left recursion, either directly within one rule or indirectly through several rules, can not be used for a top down parser.

A grammar whose rules do not pass the pairwise disjointness test will lead to an inefficient top-down parser --- either one that requires backtracking or one that requires look-ahead.

(b) Describe the advantages and disadvantages of implicit type declarations for variables.

Advantages: Implicit declarations can make it easier for the programmer to write code, since she doesn't have to also write the declarations. Maintainability can be easier too, since the information about a variable's type is not written down in a part of the program distant from where the variable is used. Readability can be better since a reader can probably infer the variable's name or its context.

Disadvantages: Reliability will probably suffer since the programmer may not always realize the type that the compiler assigned a variable.

3. Axiomatic Semantics (20 points)

Show the weakest precondition (e.g., the `{ ?? }`) for the following statements and postcondition.

(a) `{ ?? } x = x+y; { x>11 }` *(Hint: be careful here; the two x's are different. Think about what **must** be true before the statement given that x>11 is true after.*

`{x > 11-y}`

(b) `{ ?? } x = 2*y; y = 2*x; { y>16 }` *(Hint: first compute the weakest precondition that holds after the first statement is done and before the second executes. Then back that up to compute the weakest precondition that holds before the first is done.)*

Between the two statements `{X>8}` is true, so the precondition before the sequence is `{y>4}`.

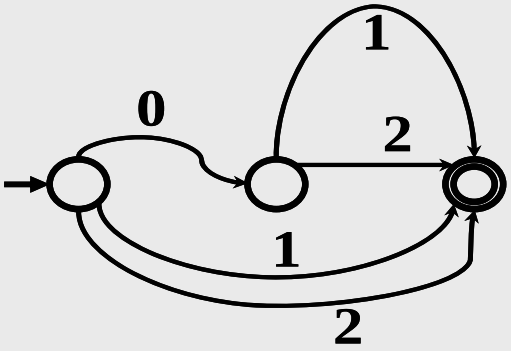
4. Regular expressions (25 points)

This problem asks you to define a simple language using a deterministic finite automaton (DFA), a regular expression (RE) and a context free grammar (CFG). Use the normal conventions for DFAs, REs and CFGs shown in the example to the right.

Consider the following regular expression. LET and DIG are named sub-expressions, square brackets delimit character options, 'a-z' represents characters in the ascii sequence between 'a' and 'z', and the sequence '\.' represents the literal period character.

```
LET: [a-zA-Z]
DIG: [0-9]

LET ( LET | DIG )* @ ( LET | DIG )+ ( \. (
LET | DIG )+ )+
```



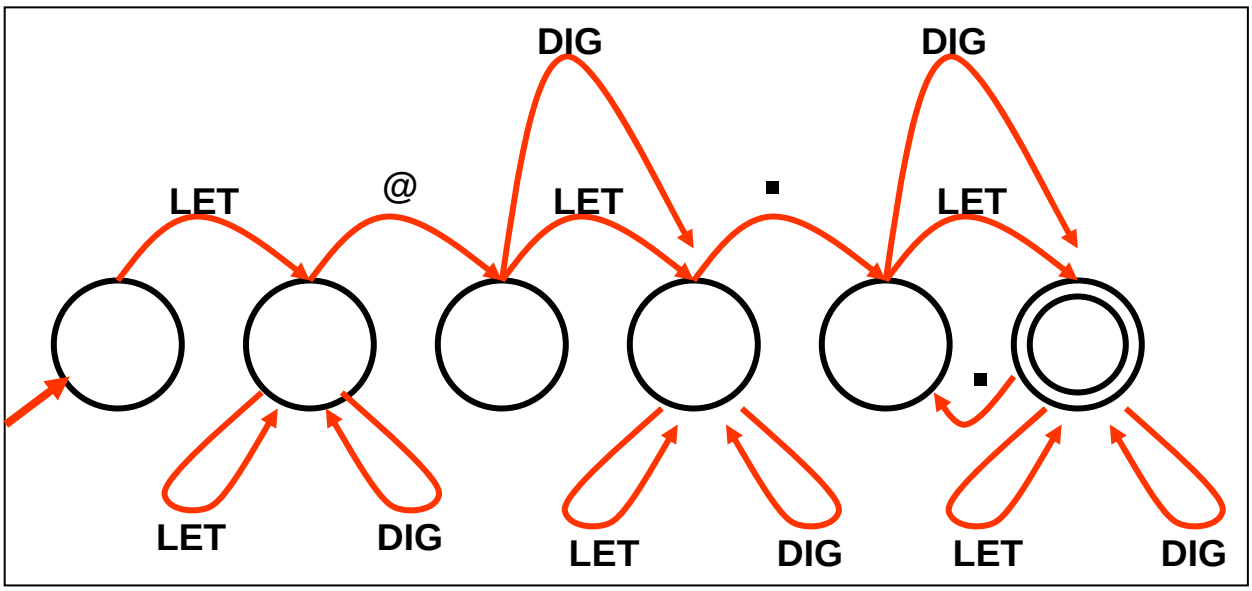
This example shows a DFA for the regular expression `0?[12]` and is also be described by the CFG:

```
S -> 0A | A
A -> 1 | 2
```

(a) Briefly describe the language accepted by the regular expression. (5) Give two examples of strings that will match and two examples that don't. (5)

The regular expression matches strings that look like simple email addresses with the additional constraint that the string before the '@' must begin with a letter and that all of the symbols can only contain letters and numbers. Positive examples are a@b.c, foo32@umbc.edu, a1b2c3@1.2.c.foo. Negative examples are 1foo@bar.com, foo@bar, and foo@.bar

(b) Draw a deterministic finite automaton that corresponds to this regular expression. Mark the accepting states with a double circle. Label each transition arrow with either an input character or one of the special tokens LET, or DIG. (15)

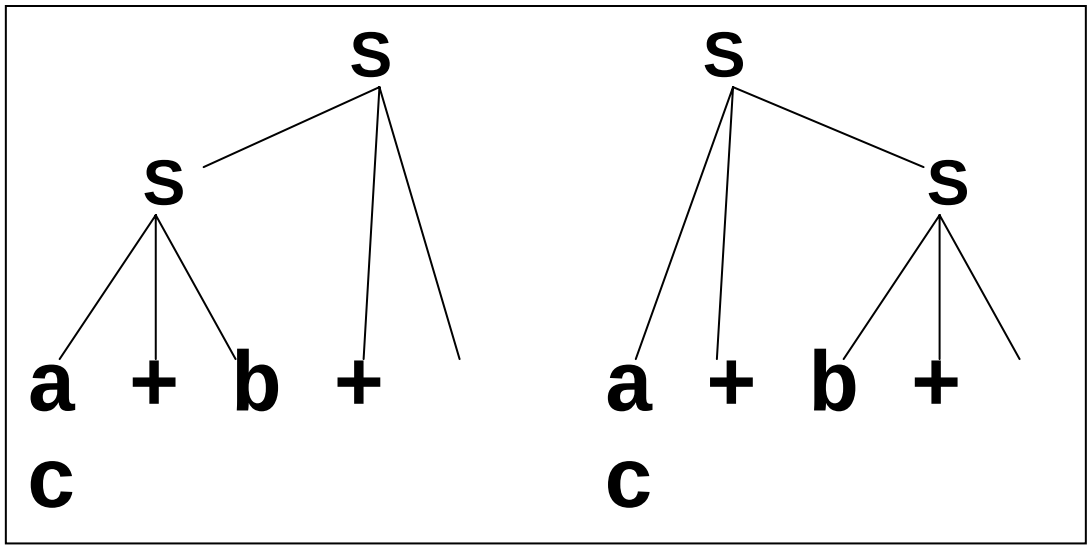


5. Simple Grammars I (20 points)

Consider the following grammar where uppercase letters indicate non-terminals and lowercase letters indicate terminals:

```
S -> S + S | a | b | c
```

(a) Show that this grammar is ambiguous by giving a string and two different parse trees that the grammar can assign to it. (10 points)



(c) Write a different grammar for the same language that is not ambiguous. (10 points)

Here are the two standard ways to do this, the first making + left associative

```
S -> S + ID | ID
ID -> a | b | c
```

and the sencond making + right associative.

```
S -> ID + S | ID
ID -> a | b | c
```

6. Operators (35 pts)

Consider the following BNF grammar for a language with two infix operators represented by # and $. Note: the order of the rules in a grammar is not significant.

```
<bar> ::= <baz>
<bar> ::= <bar> # <baz>
<foo> ::= <bar>
<foo> ::= <bar> $ <foo>
<baz> ::= a
<baz> ::= b
<baz> ::= c
```

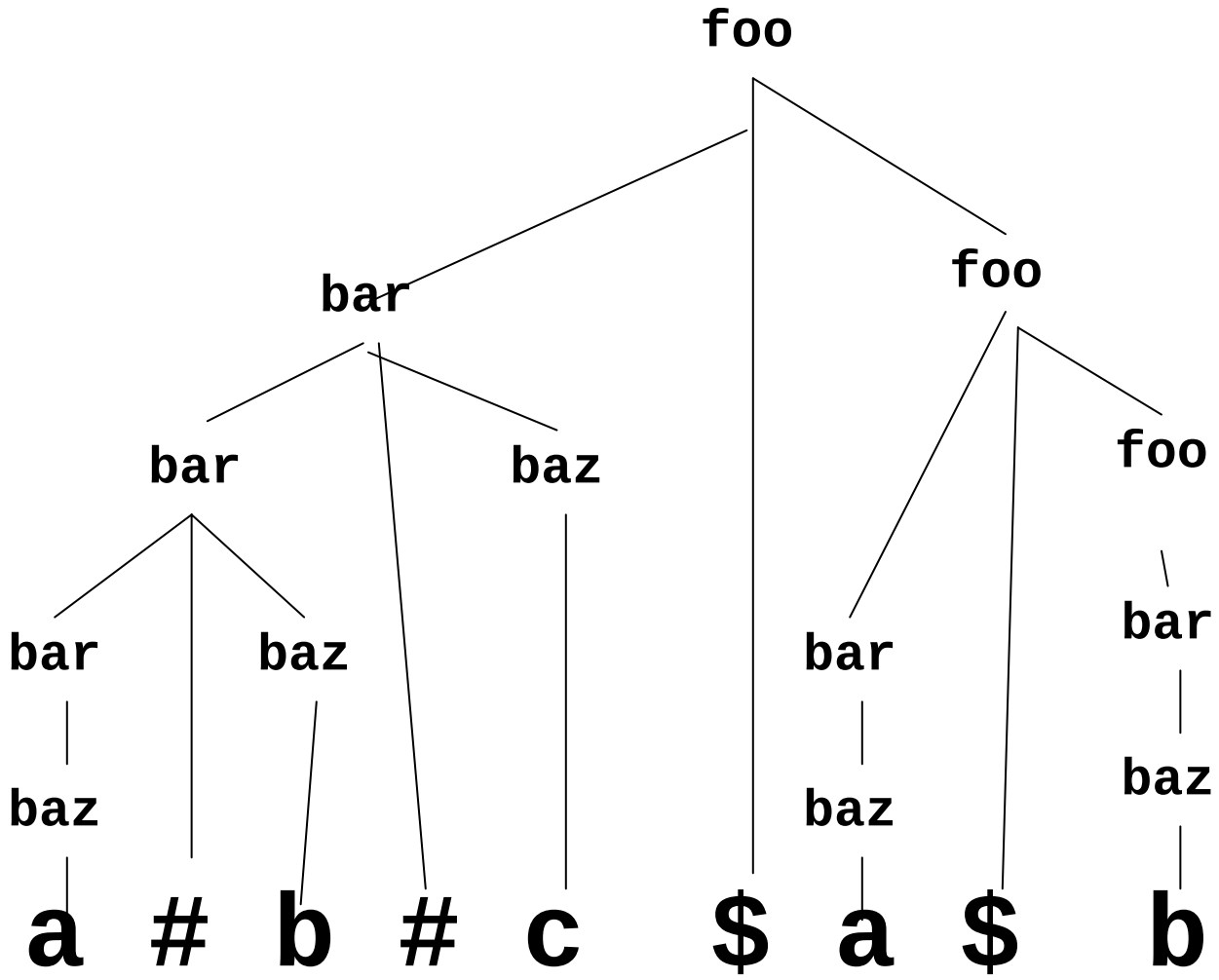
A. (5 pts) What is the associativity of the # operator: **(a) left;** (b) right; (c) neither.

B. (5 pts) What is the associativity of the $ operator: (a) left; **(b) right;** (c) neither.

C. (5 pts) Which operator has higher precedence: **(a) #;** (b) $; (c) neither.

D. (5 pts) The grammar is (a) left recursive; (b) right recursive; **(c) both left and right recursive;** (d) neither left nor right recursive.

E. (15 pts) Draw a parse tree for the following string: a # b # c $ a $ b


foo
bar
foo
bar
baz
foo
bar
baz
bar
baz
bar
baz
a # b # c $ a $ b

7. Static and dynamic scope (20 pts)

C* is a new programming language that is exactly like C, except that it uses dynamic scoping for determining the binding of variables rather than static (i.e., lexical) scoping. Consider the program to the right. Complete the trace showing what gets printed when the program is:

```
#include <stdio.h>

int i = 100;


void g(void) {
   printf("G1: %d\n", i);
   i++;
   printf("G2: %d\n", i);
}


void f(int i) {
   printf("F1: %d\n", i);
   i++;
   g();
   printf("F2: %d\n", i);
}


int main(int argc,char **argv) {
   int i = 1000;
   printf("M1: %d\n", i);
   f(i);
   printf("M2: %d\n", i);
   g();
   printf("M3: %d\n", i);
```

(a) Compiled with a normal C compiler?

```
M1: 1000
F1: 1000
G1: 100
G2: 101
F2: 1001
M2: 1000
G1: 101
G2: 102
M3: 1000
```

(b) Compiled with the new C* compiler?

```
M1: 1000
F1: 1000
G1: 1001
G2: 1002
F2: 1002
M2: 1000
G1: 1000
G2: 1001
M3: 1001
```

8. LR parsing (15 pts)

Consider an LR parser for the following simple expression grammar, producing the table to the right.

```
1: E -> E+T
2: E -> T
3: T -> T*F
4: T -> F
5: F -> (E)
6: F -> id
```

	Action						Goto		
State	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

Complete the following table which shows the first five steps of an LR parser parsing the input string **`id + ( id + id ) $`**

STACK	INPUT	ACTION
0	id+(id+id)$	Shift; goto 5
0 id 5	+(id+id)$	Reduce; goto(F,0)=3
0 F 3	+(id+id)$	Reduce; goto(T,0)=2
0 T 2	+(id+id)$	Reduce; goto(E,0)=1
0 E 1	+(id+id)$	Shift; goto 6
0 E 1 + 6	(id+id)$	Shift; goto 4