EE573 Fall 2002, Exam 2

open book, open notes. if a question seems ambiguous, ask me to clarify the question. If my answer doesn't satisfy you, please state your assumptions. Look at the black board occasionally to make sure I haven't added information about a question as a result of another students inquiries. **Papers will be taken up at 4:25**

1: Syllabus question. Will questions about the programming assignments will be answered by the TA the day the programs are due?

TRUE
[✓ **FALSE**]

2: Circle all that are true

[✓ **a.**] Single pass compilers make program optimization more difficult

b. Single pass compilers make error detection impossible

[✓ **c.**] Single pass compilers tend to be faster than multi-pass compilers

d. Single pass compilers tend to use less storage than multi-pass compilers

2: Circle all that are true

[✓ **a.**] Tokens are produced from input by the scanner

b. Tokens are produced from input by the parser

[✓ **c.**] LL parsers predict the next token(s) to be seen

[✓ **d.**] LR parsers recognize strings of tokens and non-terminals as right-hand-sides of productions

e. LL scanners predict the next character in the input program that will be in a token

3: Circle all that are true:

[✓ **a.**] LL parsers make it easier to invoke action routines within a production

b. LL parsers make it easier to maintain a semantic stack

c. LR parsers make it easier to invoke action routines within a production

[✓ **d.**] LR parsers make it easier to maintain a semantic stack

4: Circle all that are true:

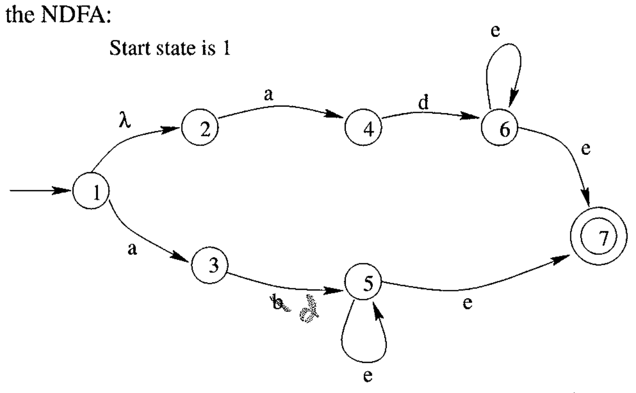
[✓ **a.**] Linear lists, hash tables and binary trees can be used to implement symbol tables

b. The number of hash table entries needed by a procedure is known after parsing the formal parameter list for the procedure

[✓ **c.**] Linear lists take $O(N^2)$ time to process N entries

[✓ **d.**] Well designed hash tables take $O(N)$ time to process N entries

5: Given the NDFA:



the set of states after converting to a non-minimal DFA is: (where "$[i, j]$" denotes the state formed from states i and j in the original NDFA.)

A $\{[1,2,3,4],[5,6],[7]\}$

B $\{[1,2,3,4],[3,4],[5],[6],[7]\}$

C $\{[1,2],[3,4],[5,6],[7]\}$

[✓ D $\{[1,2],[3,4],[5,6],[5,6,7]\}$]

E $\{[1,2],[3,4],[5],[6],[7]\}$

$$\begin{tikzcd} 1,2 \arrow[r, "a"'] & 3,4 \arrow[r, "d"'] & 5,6 \arrow[r, "e"'] & \boxed{5,6,7} \end{tikzcd}$$

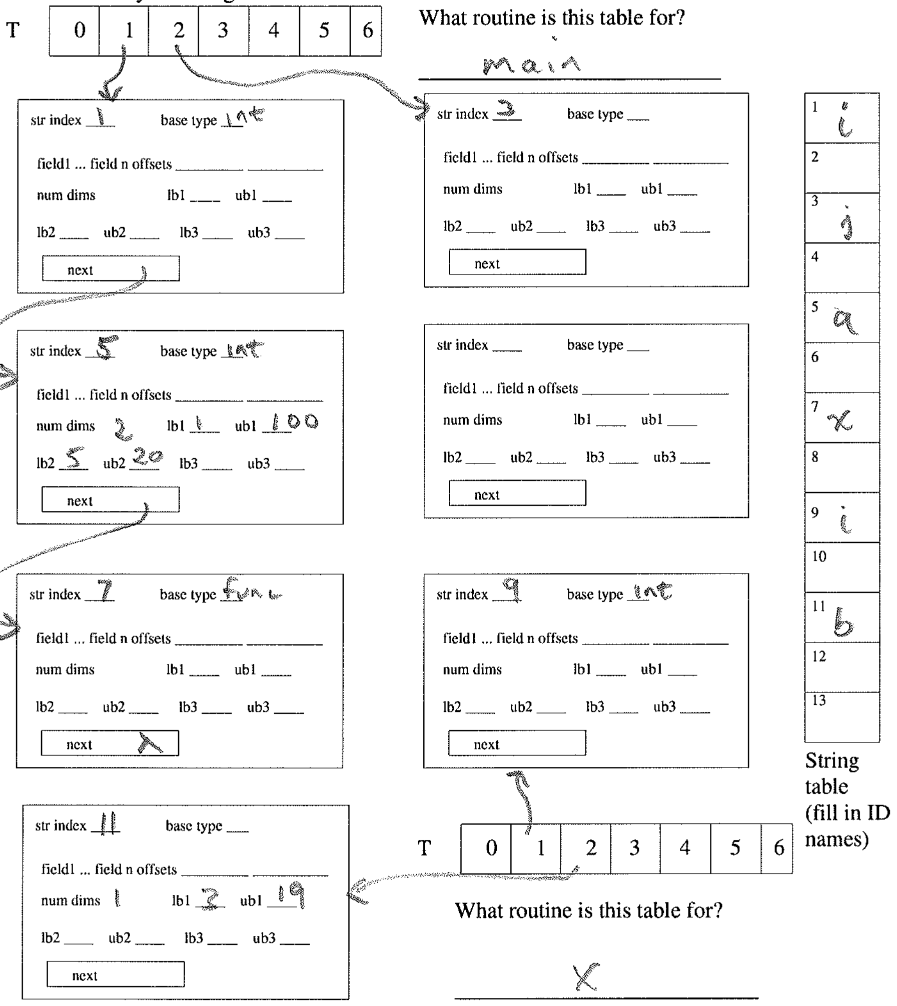
6:

```
program main( ) {
    int i, j;
    int array a[1:100,5:20];
    void procedure x( ) {
            int i;
            int array b[3:19];
    }
}
```

symbol name **n**	Hash(**n**)
a	1
i	1
j	2
k	2
l	1
m	0
x	1
b	2

6 continued: A compiler uses a hash table T such that bucket $T[i]$ consists of a linked list of nodes, with one node for each symbol. When a node is added to a bucket, it is placed at the end of the linked list, i.e. the bucket always points to the first symbol added. Entries are added in the order they are encountered in the program. New scopes are formed using a new hash table.

Show the contents of the hash table for each scope in the program above, and the values of the different fields for each identifier. Unused fields should be left blank. Identifier names must be kept for each entry. String table entries start at 1 (entry numbers is shown in the string table), and are separated by blanks. Pointers to nodes should be shown by drawing an arrow.



T | 0 | 1 | 2 | 3 | 4 | 5 | 6

What routine is this table for? main

str index 1 base type int
field1 ... field n offsets
num dims lb1 ub1
lb2 ub2 lb3 ub3
next

str index 5 base type int
field1 ... field n offsets
num dims 2 lb1 1 ub1 100
lb2 5 ub2 20 lb3 ub3
next

str index 7 base type func
field1 ... field n offsets
num dims lb1 ub1
lb2 ub2 lb3 ub3
next

str index 3 base type
field1 ... field n offsets
num dims lb1 ub1
lb2 ub2 lb3 ub3
next

str index base type
field1 ... field n offsets
num dims lb1 ub1
lb2 ub2 lb3 ub3
next

str index 9 base type int
field1 ... field n offsets
num dims lb1 ub1
lb2 ub2 lb3 ub3
next

str index 11 base type
field1 ... field n offsets
num dims 1 lb1 3 ub1 19
lb2 ub2 lb3 ub3
next

T | 0 | 1 | 2 | 3 | 4 | 5 | 6

What routine is this table for? X

1	i
2	
3	j
4	
5	a
6	
7	x
8	
9	i
10	
11	b
12	
13	

String table (fill in ID names)

7: In the expression a = b + c the values of

a. a, b and c are l-values

[✓ **b.**] a is an l-value, and b and c are r-values

c. a, b and c are r-values

d. cannot tell without seeing how a is used next

e. cannot tell without seeing where b and c are computed

The following grammar is used in questions 8 and 9.

if stmt	→	**if** #start_if *bexpr* #if_test **then** *stmtlist*
		elseif #gen_jump #gen_else_label *b expr* #if_test **then** *stmts*
		else part **end if**; #gen_out_label
else part	→	**else** #gen_jump #gen_else_label *stmtlist*
else part	→	#gen_else_label

where the semantic routines are defined as in the book (see pages 429 to 431.)

8: If the first line of the grammar is changed to:

if stmt → **if** *bexpr*#xyz **then** *stmtlist*

define the #xyz routine. The definition may include calls to any semantic action routines defined in the grammar above.

start if
if test

9: Briefly state why ~~can't~~ the second line of the grammar above cannot be rewritten as:

elseif *bexpr* #gen_jump #gen_else_label #if_test **then** *stmts*

boolean expression code will be evaluated before the jump to the end of the if and before the else label. It will not be executed if no elseif is to be tested, and will execute if it is not to be executed.

10: Put the grammar:

if stmt	→	**if** #start_if *bexpr* #if_test **then** *stmtlist*
		else part **end if**; #gen_out_label
else part	→	**else** #gen_jandl *stmtlist*
else part	→	#gen_else_label

into a form suitable for use by an LR parser generator.

I think This question assumed a stupid LR parser generator that doesn't split up productions. The answer would Then be:

if stmt → if st bool then_part

if st → if # start_if

bool → bexpr # if_test

then_part → then stmt list else part end if ; #gen_out_label

else_part → else st else end

else st → else # gen_jandl

else end → stmt list

else part → # gen_else_label

The following program is used in questions 11.

```
program a( ) begin
    int x;
    procedure b (int c, int d) begin
        procedure e (int f) begin
            int c, int g;
            ...call b...
        end e;
        procedure h (int i) begin
            int d; int j;
            ...
        end h;
        ...if ( ) then call h endif...
        ...if ( ) then call e endif...
    end b;
    ...call b...
end program
```

Assume in these questions that the first call to some procedure p is denoted p, the second is denoted p' and the third p''.

11: Consider the sequence of calls and returns a, b, e, b', h, return from h, e'. Circle the stack configuration after this sequence (the stack grows towards the bottom of the page, i.e. if a calls b, b's activation record is closer to the bottom of the page.) Vertical lines separate different activation records. Ignore empty activation records.

choice A	**choice B**	[✓ **choice C**]	**choice D**
ret. addr. static info x	ret. addr. static info x	ret. addr. static info x	ret. addr. static info x
ret. addr. static info c, d	ret. addr. static info f, c, g	ret. addr. static info c, d	ret. addr. static info c, d,
ret. addr. static info f, c, g	ret. addr. static info c, d	ret. addr. static info f, c, g	ret. addr static info f, c, g
ret. addr. static info c, d	ret. addr. static info f, c, g	ret. addr. static info c, d	
ret. addr. static info i, d, j		ret. addr. static info f, c, g	
ret. addr. static info f, c, g			