CSE P 501 Exam 3/6/08

Name _____**Sample Solution**_____

There are 8 questions worth a total of 100 points. Please budget your time so you get to all the questions, particularly some of the later ones that are worth more points than some of the earlier ones. Keep your answers brief and to the point.

You may refer to the following references:

Course lecture slides and notes
Your primary compiler textbook(s)

No other books or other materials, including old exams or homework.

Please wait to turn the page until everyone is told to begin.

Score _____

1 _____ / 12

2 _____ /10

3 _____ /15

4 _____ /15

5 _____ /15

6 _____ /8

7 _____ /10

8 _____ /15

1. (12 points) Write a regular expression or set of regular expressions that generate the following sets of strings. You can use abbreviations (i.e., name = regular expression) if it helps to make your answer clearer.

(a) (6 points) All strings of 0's and 1's where either there are no 1's in the string, or the number of 1's is odd (i.e., a single 1, or three 1's, or five 1's, etc.).

There are many possible solutions. Here's one:

0* (10* (10*10*)*)?

(b) (6 points) All identifiers that are formed using the following rules:
- consist of only lower case letters a-z, digits 0-9, and underscore _
- must begin with a letter
- must not end with an underscore
- no two underscores may appear next to each other

Examples: Legal identifiers: hi, hi_there, only1idea, a_very_long_name.
Illegal identifiers: double__underscore, 1digit_at_the_beginning, ends_with_underscore_

letter = a | b | c | … | z
digit = 0 | 1 | 2 | … | 9

letter (letter | digit | _ (letter | digit)) *

There are many similar solutions. Here's one that's maybe a little less cryptic:

letter ((letter | digit) * | _ (letter | digit)+) *

2. (10 points) The simple expression grammar

expr ::= *expr* + *expr* | *expr* * *expr* | id | (*expr*)

is known to be ambiguous. One of the summer interns has proposed replacing this with the following grammar to solve the problem:

expr ::= *term* + *term* | *term*
term ::= *factor* * *factor* | *factor*
factor ::= id | (*expr*)

(a) (5 points) Does this new grammar generate the same set of strings as the old one? If they do generate the same strings, explain why (this does not need to be a formal proof, but it should be a convincing explanation). If not, give an example of some expression generated by one of the grammars that is not generated by the other

No. The original grammar can generate arbitrary sequences of additions and multiplications, like id+id+id. The new grammar cannot.

(b) (5 points) Is this new grammar ambiguous? (For this part of the question, ignore the issue of whether or not it generates the same set of strings as the original grammar.) If it is ambiguous, give an example that shows that it is; if not, explain why not (again, this does not need to be a formal proof, but it should be a convincing explanation).

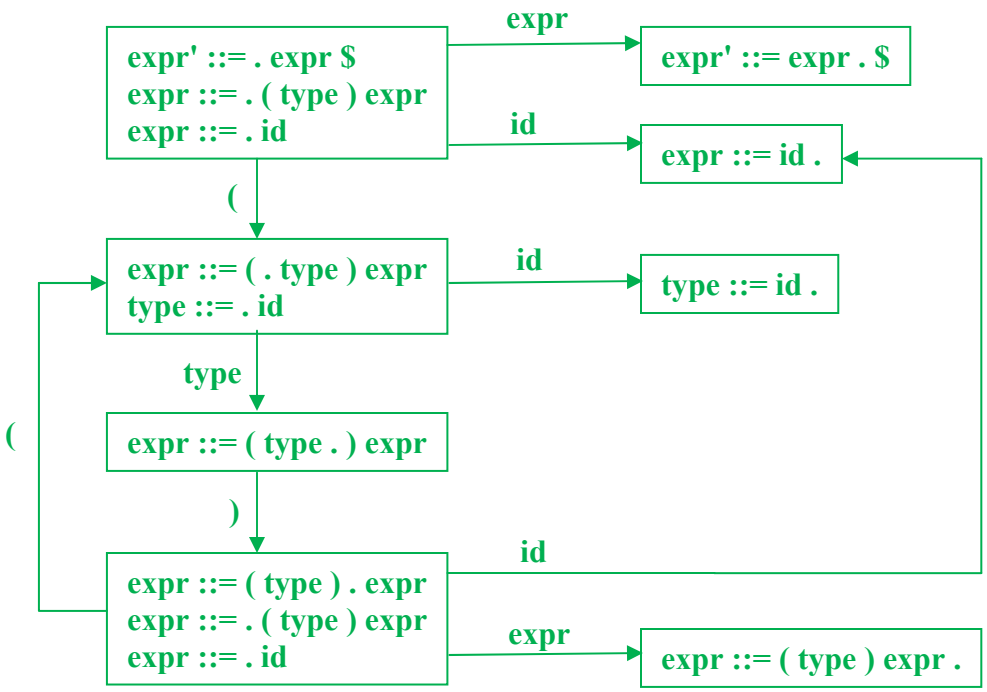
It is not ambiguous. The additional nonterminals remove the ambiguities due to mixed operator precedence. The restricted rules for *expr* and *term* don't allow additions or multiplications with more than two operands, unless one of them is a parenthesized expression.

3. (15 points) The obligatory LR-parsing question.

Consider the following grammar:

0. *expr'* ::= *expr* $
1. *expr* ::= (*type*) *expr*
2. *expr* ::= id
3. *type* ::= id

(a) (12 points) Draw the LR(0) state machine for this grammar. You do not need to write out the parser tables or first/follow/nullable sets.



(b) (3 points) Is this grammar LR(0)? Why or why not?

Yes. There are no shift-reduce or reduce-reduce conflicts in the LR(0) states.

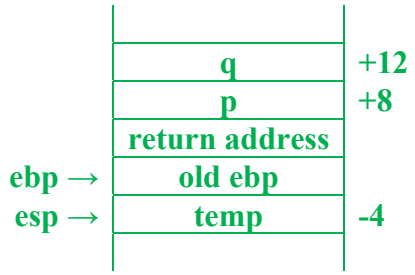
4. (15 points) The other (mostly) obligatory question.

Consider the following C function:

```
int thing(int p, int q) {
   int temp;
   temp = p + q;
   return fun(temp) + temp;
}
```

You should assume that `fun` is another function declared elsewhere in the same program with the following prototype: `int fun(int x);` (i.e., assume that you can call `fun` without having to declare it further, either in C or in an assembly language version).

(a) (4 points) Draw the stack frame for function `thing` as it would appear in an x86 program using the standard C calling conventions. Your picture should show the layout of function parameters, local variables, and the `esp` and `ebp` registers as they exist after the function prologue has executed and has allocated the stack frame, but before any of the statements in the body of the function have been executed. Be sure to show the numeric offsets from register `ebp` to each parameter and local variable.



(continued next page)

4. (cont). (b) (11 points) Translate function `thing` to x86 assembly language. Your translation should not omit any statements, i.e., it should store the result of `p+q` in `temp`, but otherwise it can be any reasonable x86 program that follows the C stack layout and calling conventions. You may use either the Intel/Microsoft or GNU conventions for assembly language syntax and layout. (Just be sure you pick one or the other and don't mix them.)

Function definition repeated here for convenience:

```
int thing(int p, int q) {
   int temp;
   temp = p + q;
   return fun(temp) + temp;
}
```

```
thing:  push ebp           ; prologue
        mov  ebp,esp
        sub  esp,4

        mov  eax,[ebp+8]   ; temp = p+q;
        add  eax,[ebp+12}
        mov  [ebp-4],eax

        push eax           ; call fun(temp)
        call fun
        add  esp,4         ; (pop argument)

        add  eax,[ebp-4]   ; return fun(temp)+temp in eax
        mov  esp,ebp       ; epilogue
        pop  ebp
        ret
```

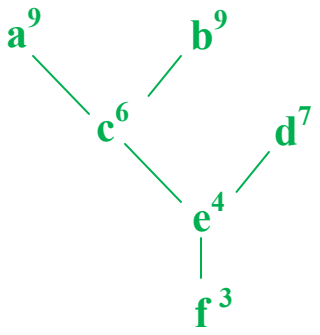
5. (15 points) Suppose we have the hypothetical machine from lecture with the following instructions and latency times:

Operation	Cycles
LOAD	3
STORE	3
ADD	1
MULT	2

Now, suppose we have the following sequence of instructions

(a) LOAD r1 ← x
(b) LOAD r2 ← y
(c) MULT r3 ← r1 * r2
(d) LOAD r4 ← z
(e) ADD r5 ← r3 + r4
(f) STORE ans ← r5

(a) (6 points) Draw a precedence graph showing the dependencies between these instructions. Label each node (instruction) in the graph with both the instruction letter (a-f) and its latency – the number of cycles between the beginning of that instruction and the end of the graph



(continued next page – apologies for all the page flipping this might require)

(b) (6 points) Rewrite the instructions in the order they would be chosen by forward list scheduling (i.e., choosing at each step an instruction that is not dependent on any other instruction that has not yet been issued). If there is a tie at any step for which instruction would best be scheduled next, pick one of them arbitrarily. You do not need to show your bookkeeping or trace the algorithm, although if you leave some clues around it could be useful if we need to figure out how to assign partial credit.

```
(a) LOAD   r1 ← x          // first two loads can be in either order
(b) LOAD   r2 ← y
(d) LOAD   r4 ← z
(c) MULT   r3 ← r1 * r2
(d) ADD    r5 ← r3 + r4
(e) STORE  ans ← r5
```

(c) (3 points) How many cycles were required by the original instruction schedule? How many cycles are required by the new schedule you created in part (b)?

Original: 12 cycles
New: 10 cycles

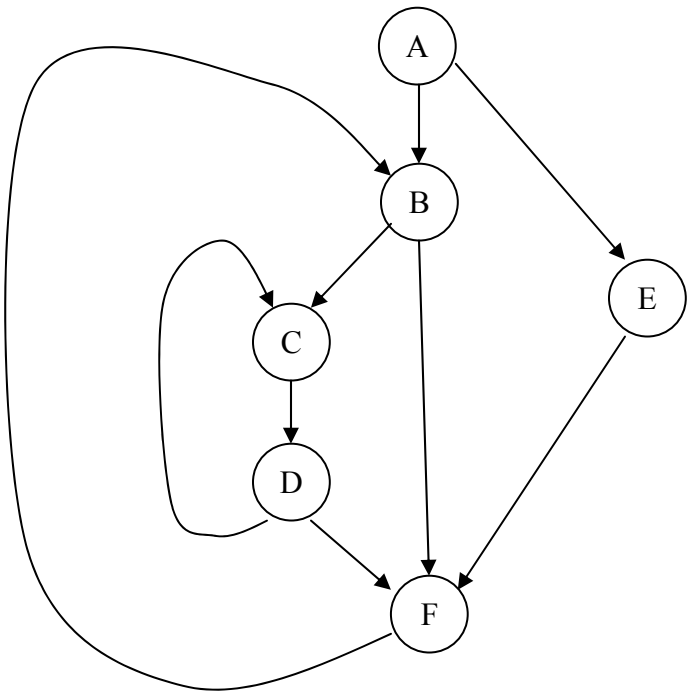
6. (8 points) We looked at several iterative dataflow analysis problems involving various GEN, KILL, IN, and OUT sets. These included live variable analysis, reaching definitions, very busy expressions, and available expressions.

Suppose we want to use iterative dataflow analysis to report all variables that *might* be uninitialized at some point where they are used in the program. Which of the iterative dataflow analysis problems could we use to determine this information? How would we use the information reported by that analysis to decide which variables might be uninitialized?

(If none of the standard analysis problems listed above give the information needed to detect uninitialized variables, describe a dataflow analysis problem that would work, and how it would be used.)

Live variable analysis. A variable is live at a program point if there is a path from that point to the exit along which its value may be used before it is redefined. Any variables that are live at the beginning of a procedure are ones that could be undefined at some use.

7. (10 points) Consider the following control-flow graph (with apologies for the less than spectacular artwork):

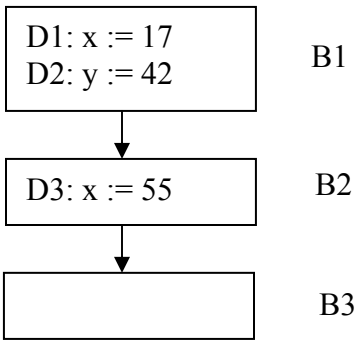


For each node in this flowgraph, list the nodes that dominate it, and list the node that is its immediate dominator.

Node	Dominators	Immediate Dominator
A	**A**	**–**
B	**A, B**	**A**
C	**A, B, C**	**B**
D	**A, B, C, D**	**C**
E	**A, E**	**A**
F	**A, F**	**A**

8. (15 points) In the lectures on dataflow problems, we looked at an example involving liveness. That problem determines which variables are live on exit from each block in the control flow graph.

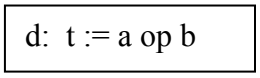
Another fundamental dataflow problem is *reaching definitions.* In this problem, we look at each definition that appears in a block in the control flow graph. The problem is to determine which other blocks in the control flow graph could potentially see the value of the variable that was assigned in that definition. For example, in the following diagram:



the definitions D2 and D3 reach B3, because there are no subsequent assignments to x or y between those definitions and B3. But definition D1 does not reach B3, because the reassignment to x in B2 kills the definition D1. (Definition D1 does reach B2, but it is killed there and reaches no further.)

A definition *d* in block *p reaches* block *q* if there is *at least* one path from *p* to *q* along which definition *d* is not killed. For instance, in the above example, if there were another path in the flowgraph from B1 to B3, then it could be possible for D1 to reach B3, in spite of it being killed in B2 (depending on what happens on that other path, of course).

We can set this up as a dataflow problem as follows. For each block in the control-flow graph, we define GEN to be the definitions generated in that block and KILL to be the definitions killed by that block. This information can be computed once, statically. In particular, if a block contains



Then d is in the GEN set for this block (assuming it was not killed later in the same block). The KILL set for this block contains the set of all other definitions d′, where d′ is a different definition of variable `t` somewhere in the program. In the example at the top of the page, definition D3 is in the GEN set for B2, and definition D1 is in B2's KILL set (because it also defines the same variable, `x`). To simplify this problem, we will restrict ourselves to blocks that contain a single statement each.

(continued next page)

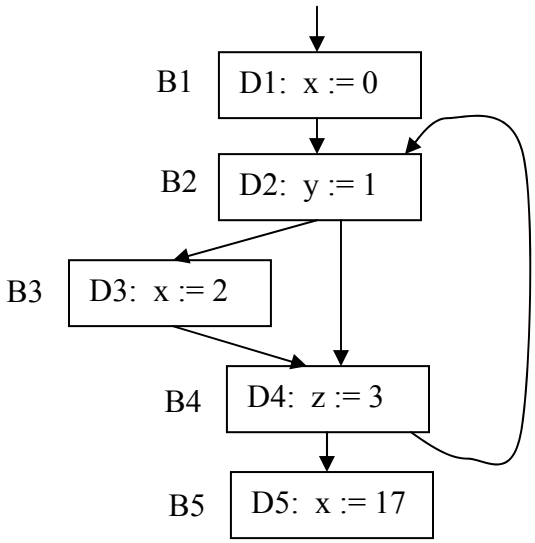
8. (cont) The dataflow problem for reaching definitions, then, can be defined as follows:

GEN[b] = { set containing definition d from block b, if any }
KILL[b] = { all other definitions that assign to a variable that is defined in b }

$IN[b] = \bigcup_{p \in pred[b]} OUT[p]$

$OUT[b] = GEN[b] \cup (IN[b] - KILL[b])$

Now (finally), here is the problem. Compute the reaching definitions for the nodes in the following flowgraph. You should first fill in GEN and KILL in the table below for each block, then iteratively solve for the IN and OUT sets. Choose whichever direction to solve for IN and OUT that you wish (forward or backward).



Block	GEN	KILL	IN 1	OUT 1	IN 2	OUT 2	IN 3	OUT 3
B1	**D1**	**D3,D5**	**--**	**D1**	**--**	**D1**	**same**	**same**
B2	**D2**	**--**	**D1**	**D1,D2**	**D1,2,3,4**	**D1,2,3,4**	**same**	**same**
B3	**D3**	**D1,D5**	**D1,D2**	**D2,D3**	**D1,2,3,4**	**D2,3,4**	**same**	**same**
B4	**D4**	**--**	**D1,2,3**	**D1,2,3,4**	**D1,2,3,4**	**D1,2,3,4**	**same**	**same**
B5	**D5**	**D1,D3**	**D1,2,3,4**	**D2,4,5**	**D1,2,3,4**	**D2,4,5**	**same**	**same**

(continue on back of the previous or of this page if needed)