CSE P 501 Exam 3/3/16 Sample Solution

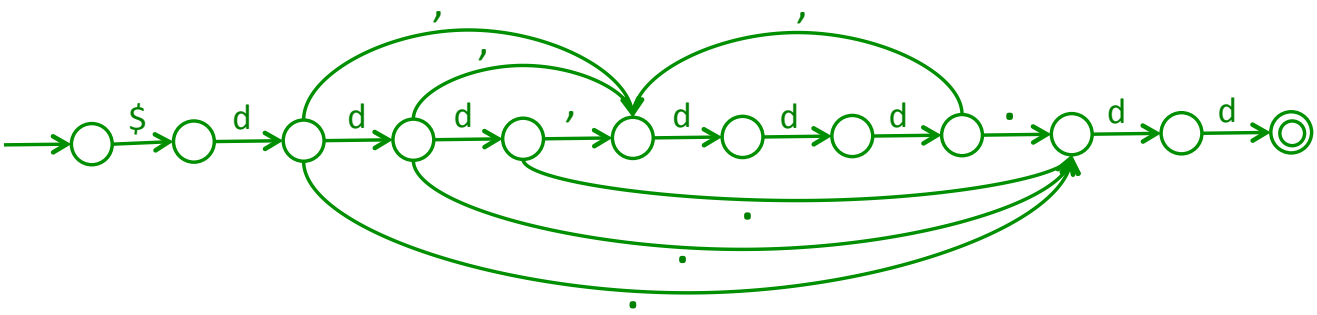
Question 1. (10 points) Regular expressions. (a) (5 points) Give a regular expression that generates strings representing currency amounts formatted as follows: each string starts with the character '$' followed by comma-separated groups of decimal digits followed by a '.' and two decimal digits. Each group of digits to the left of the decimal point consists of three digits except for the leftmost one, which might only have one or two digits. Examples: $0.00, $12,345.67, $0,000,000.00. Strings that are not in this set: $.00 (no digit to the left of '.'), $1234.56 (no comma between 1 and 2), $0 (no '.' followed by two digits), $1.5 (only one digit following the '.'), $12,34,567.89 (only two digits '34' between two commas), $,123.45 (no digit to the left of the comma).

Ground rules (the fine print): You may only use the basic operations of concatenation, choice (|), and repetition (*) plus the derived operators ? and +, and simple character classes like `[abc0-9]` and `[^a-z]`. You may use abbreviations like `vowels = [aeiou]`. You may not use more complex operators found in various software tools that handle extended regular expressions and should not use '\' or other escape characters.

d = [0-9]

$ d d? d? (, d d d)* . d d

(b) (5 points) Draw a DFA that accepts currency amounts as defined above.



Question 2. (10 points) CFGs and ambiguity. Consider the following grammar that generates all sequences of balanced parentheses, including the empty string. Examples: (), (()), (())((())), ()()(), etc.

S ::= (*S*) | *S S* | ε

(a) (5 points) Show that this grammar is ambiguous.

Here are two distinct leftmost derivations of ( ) :

S **=>** ***S S*** **=>** **(** ***S*** **)** ***S*** **=>** **( )** ***S*** **=>** **( )**

S **=>** **(** ***S*** **)** **=>** **( )**

It would, of course, also be fine to show the ambiguity by drawing two structurally different parse trees for the same sentence or use rightmost derivations. There are many other possible examples.

(b) (5 points) Give an unambiguous grammar that generates the same set of strings as the original grammar.

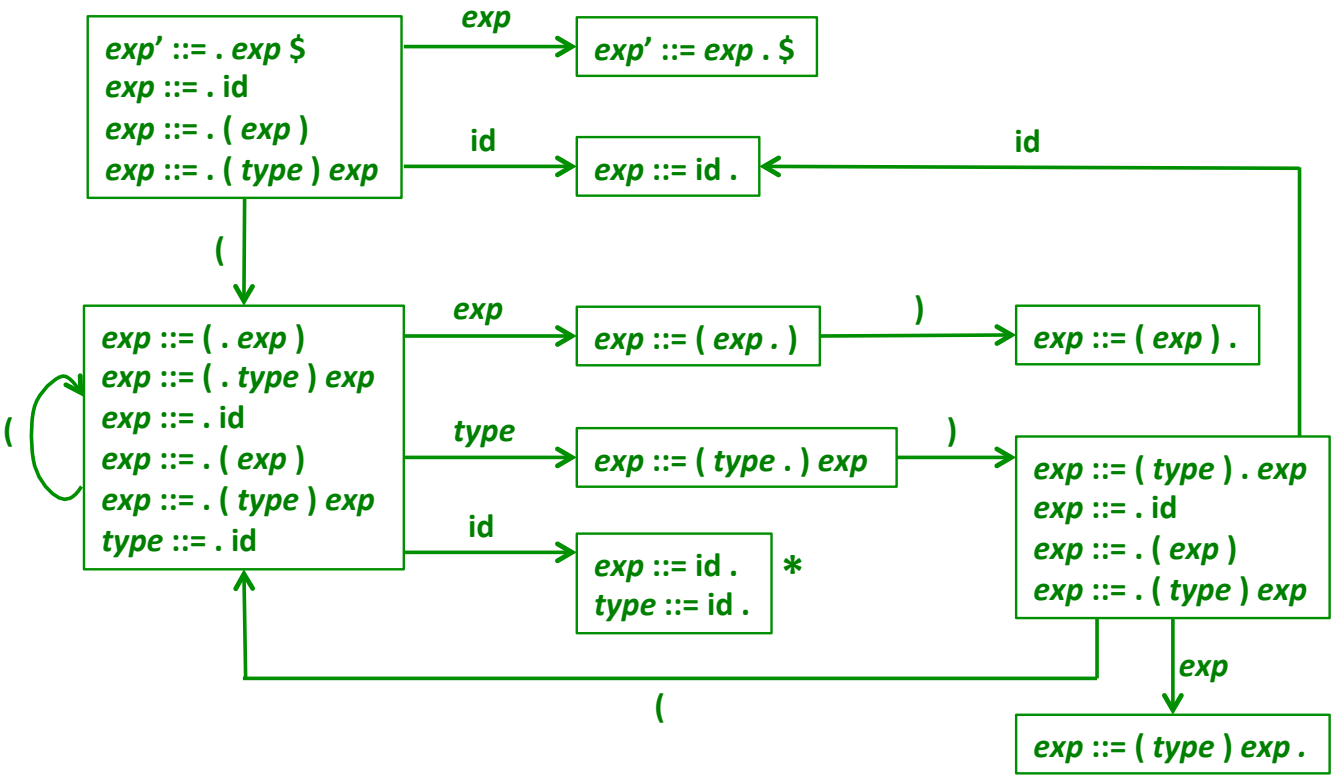
One possibility:

S **::=** **(** ***S*** **)** ***S*** **|** **ε**

Question 3. (18 points) The more-or-less-same-old LR-parsing question. Consider the following grammar, which is meant to represent expressions with optional parentheses and type casts.

0. *exp′* ::= *exp* $
1. *exp*::= id
2. *exp*::= (*exp*)
3. *exp*::= (*type*) *exp*
4. *type* ::= id

(a) (10 points) Draw the LR(0) state machine for this grammar. (You do not need to include the table with shift/reduce and goto actions, although you can write that out later if you find it useful to answer other parts of the question.)



(continued next page)

Question 3. (cont.) Grammar repeated for reference

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

(b) (4 points) Compute First, Follow, and Nullable for each of the non-terminals in this grammar.

	Nullable	First	Follow
exp'	**no**	**id (**	
exp	**no**	**id (**	**) $**
type	**no**	**id**	**)**

(c) (2 points) Is this grammar LR(0)? Why or why not?

No. There is a reduce-reduce conflict in the state marked *.

(d) (2 points) Is this grammar SLR? Why or why not?

No. The SLR one-symbol lookahead can sometimes be used to resolve shift-reduce conflicts, but it cannot resolve reduce-reduce conflicts.

Question 4. (8 points) (LL parsing/grammars) Consider the following grammar:

0. *S′* ::= *S* $
1. *S* ::= w *D* y
2. *D* ::= *D* x
3. *D* ::= z

Is this a LL(1) grammar suitable for top-down predictive parsing? If yes, give a specific technical justification for your answer. If not, give a grammar that generates the same language and is LL(1) if that is possible. If no LL(1) grammar can generate the same language produced by the original grammar, give an explanation of why this is not possible.

No. The right-hand sides of productions 2 and 3 have overlapping FIRST sets. FIRST(*D* x) and FIRST(z) both contain z.

A fix is to get rid of the direct left recursion in rule 2, giving a grammar like the following:

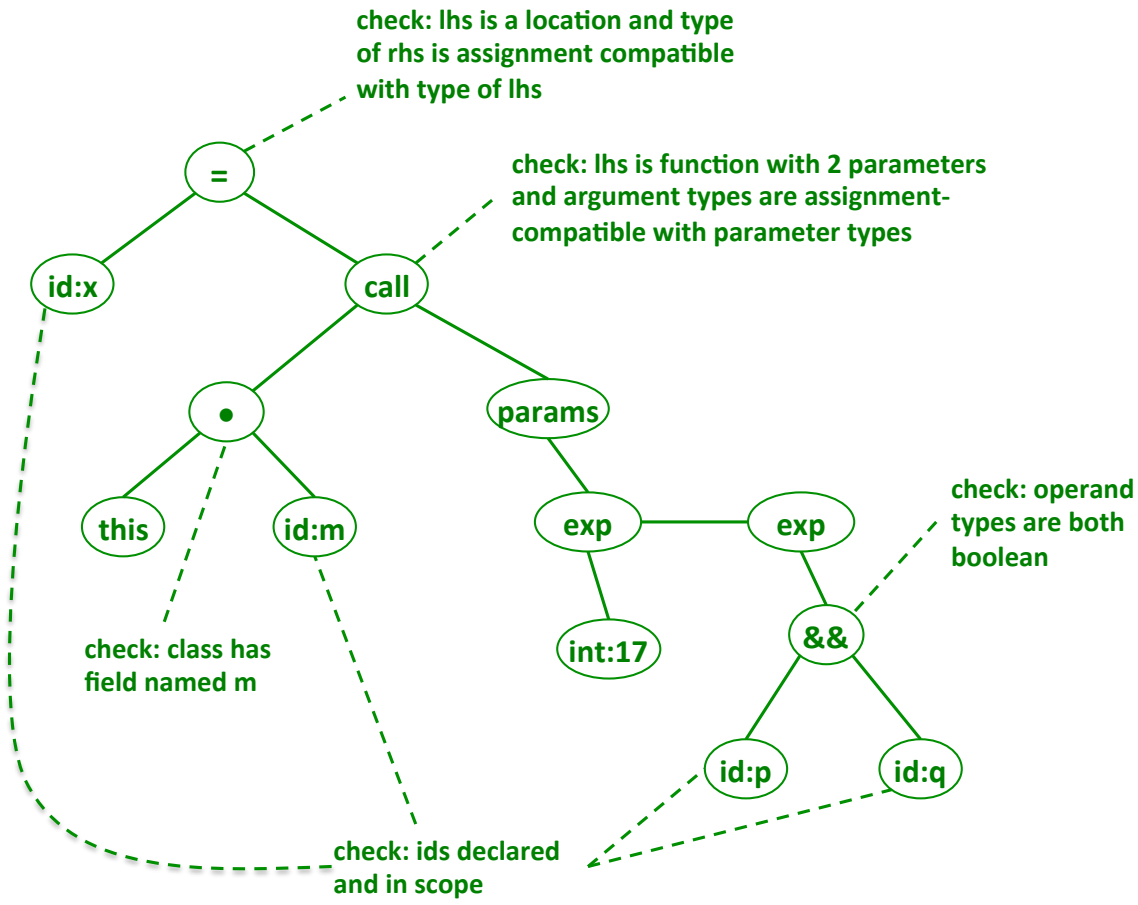
0. ***S′* ::= *S* $**
1. ***S* ::= w *D* y**
2. ***D* ::= z *D′***
3. ***D′* ::= x *D′* | ε**

Question 5. (10 points) Abstract syntax and semantics. Suppose we have the following assignment statement in a MiniJava program:

```
x = this.m(17, p&&q);
```

(a) (5 points) Draw an abstract syntax tree (AST) for this statement in the blank space at the bottom of the page. You are not expected to remember the exact names of the classes or node types in the MiniJava AST package or your project code, but your AST should have the appropriate level of detail that would be found there and should use reasonable names for the nodes if you don't remember the specific MiniJava ones.

(b) (5 points) After you have drawn your AST, annotate it by outlining the semantic and type checks that would need to be done at each point in the AST to verify that this statement is a semantically as well as syntactically legal MiniJava statement.



Answers that had basically this structure received full credit even if the exact details were different.

Question 6. (18 points) A little x86-64 hacking. A clever way to implement a function to calculate *n*! is as follows:

```
int fact(int n) {
  return factaux(n, 1);
}

// return n! * acc
int factaux(int n, int acc) {
  if (n < 2)
    return acc;
  else
    return factaux(n-1, n*acc);
}
```

(For programming language geeks – yes, this is a tail-recursive version of the factorial function. Feel free to ignore this observation and just get on with the question.)

Your job is to translate these two functions into x86-64 assembly language using the gcc/AT&T/Linux assembler syntax and the x86-64 register and function call conventions that we have used in our code examples.

You must implement the code for `fact` exactly as given using the standard x86-64 function call and stack frame conventions. But, if you want, you can take advantage of the observation that `factaux` is basically implementing a loop, where each time the function calls itself recursively it simply goes back to the beginning of `factaux` after updating the parameters `n` and `acc` to be `n-1` and `n*acc` respectively. So, if you want, you can implement `factaux` using a loop, or you can implement the recursive function calls as written, or otherwise hand-optimize the code – whichever is easiest and fastest during an exam.

We suggest you use the remainder of this page for scratch work, and then write the actual code on the next page.

(write your code on next page – feel free to detach this page while you're working.)

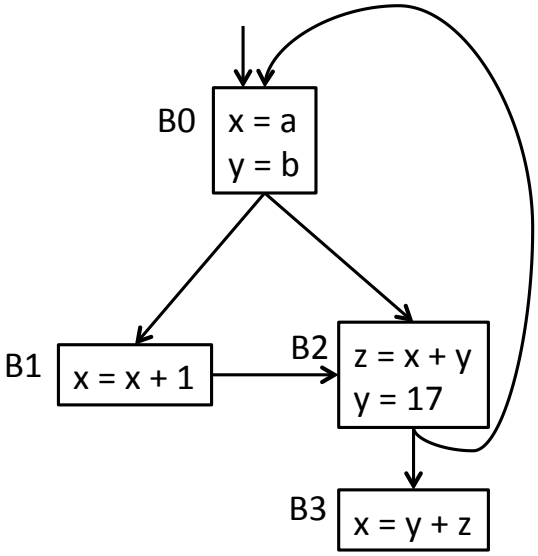
Question 6. (cont.) Write your x86-64 versions of `fact` and `factaux` below.

```
fact:
   pushq %rbp          # standard prologue
   movq  %rsp, %rbp
   movq  $1, %rsi      # call factaux -- set acc=1
   call  factaux       #  (n in %rdi already)
   movq  %rbp, %rsp    # exit with factaux result in %rax
   popq  %rbp
   ret

factaux:
   pushq %rbp          # standard prologue
   movq  %rsp, %rbp
   movq  %rsi, %rax    # copy acc to %rax
   cmpq  $1, %rdi      # if n <= 1 return acc
   jle   exit
   imulq %rdi, %rsi    # acc = n*acc
   subq  $1, %rdi      # n = n - 1
   call  factaux       # call factaux(n,acc) recursively
exit:                  # return with result in %rax
   movq  %rbp, %rsp
   popq  %rbp
   ret
```

There are, of course, many possible solutions, particularly for `factaux`. This one is similar to the code produced by `gcc` with no significant optimizations enabled.

The next two questions concern the following control flow graph:



Question 7. (12 points) Dataflow analysis. Recall from lecture that *live-variable* analysis determines for each point *p* in a program which variables are live at that point. A live variable *v* at point *p* is one where there exists a path from point *p* to another point *q* where *v* is used without *v* being redefined anywhere along that path. The sets for the live variable dataflow problem are:

use[*b*] = variables used in block *b* before any definition
def[*b*] = variables defined in block *b* and not later killed in *b*
in[*b*] = variables live on entry to block *b*
out[*b*] = variables live on exit from block *b*

The dataflow equations for live variables are

$$in[b] = use[b] \cup (out[b] - def[b])$$
$$out[b] = \cup_{s \in succ[b]} in[s]$$

On the next page, calculate the use and def sets for each block, then solve for the in and out sets of each block. A table is provided with room for the use and def sets for each block and up to three iterations of the main algorithm to solve for the in and out sets. If the algorithm does not converge after three iterations, use additional space until it does.

Hint: remember that live-variables is a backwards dataflow problem, so the algorithm should update the sets from the end of the flowgraph towards the beginning to reduce the total amount of work needed.

You may remove this page for reference while working these problems.

Question 7. (cont.) Write the results of calculations for live variables in the chart below. Use the rest of the page for additional space if needed.

Block	use	def	out (1)	in (1)	out (2)	in (2)	out (3)	in (3)
B3	**y, z**	**x**	**---**	**y, z**	**---**	**y, z**		
B2	**x, y**	**y, z**	**y, z**	**x, y**	**y, z**	**x, y**		
B1	**x**	**x**	**x, y**	**x, y**	**x, y**	**x, y**		
B0	**[a, b]***	**x, y**	**x, y**	**---**	**x, y**	**---**		

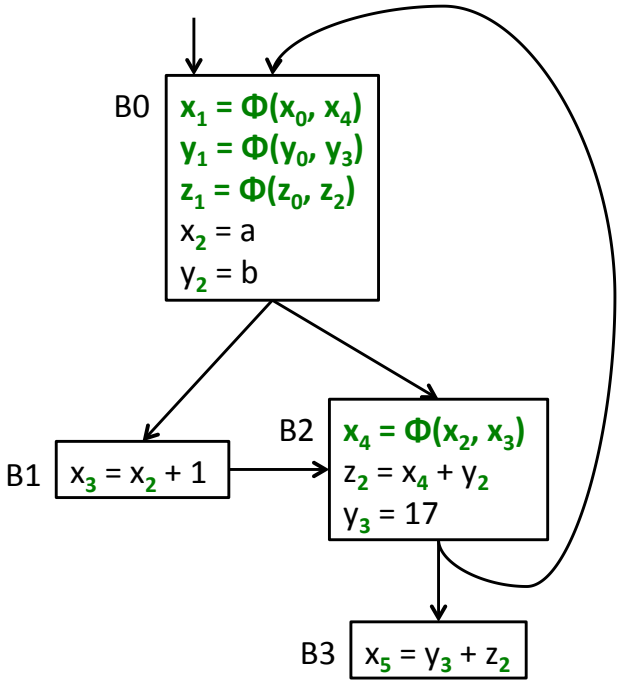
***B0 uses a and b, which are assumed to be input variables, and it is fine to omit them from the analysis.**

There are no changes in the second round, so the algorithm terminates after two rounds and the remaining columns in the table are not needed.

Question 8. Dominators and SSA. (14 points) (a) (6 points) Using the same control flow graph from the previous problem, complete the following table. List for each node: the nodes that dominate it, the node that is its immediate dominator (if any), and the nodes that are in its dominance frontier (if any):

Node	Dominators	IDOM	Dominance Frontier
B0	**B0**	**---**	**B0**
B1	**B0, B1**	**B0**	**B2**
B2	**B0, B2**	**B0**	**B0**
B3	**B0, B2, B3**	**B2**	**---**

(b) (8 points) Now redraw the flowgraph in SSA (static single-assignment) form. You need to insert appropriate Φ-functions where they are required and add appropriate version numbers to all variables. Do not insert Φ-functions at the beginning of a block if they clearly would not be appropriate there, but we will not penalize extra Φ-functions elsewhere if they are inserted correctly. You do not need to trace the steps of any particular algorithm to place the Φ-functions as long as you add them to the flowgraph in appropriate places. Additional space is provided on the next page if needed.



Note: We did not make any deductions if the Φ-functions in B0 were omitted. The dominator function criteria places Φ-functions there, but since all of those assignments are dead because the original variables are reassigned before use, these statements would have no effect in the final program.