Question 1. (12 points) For each of the following tasks, identify the stage of the compiler that performs that task or detects the situation. Assume that the compiler is a conventional one that generates native code (e.g., x86, MIPS, etc) for a language like C++ or Java. If more than one stage of the compiler can *always* perform the check as part of its usual processing, pick the earliest such stage. Use the following abbreviations to identify the stages:

scan – scanner
parse – parser
sem – static semantics
opt – optimization
instr – instruction selection & scheduling
reg – register allocation
run – runtime (i.e., when the program is executed after compilation)

_**Parse**__ Operator + is left associative

_**Parse**__ There is no `<=>` operator in the language

_**Opt**____ Move the computation `x+y` outside a loop since neither `x` nor `y` change inside the loop

_**Instr**___ Optimize a sequence of additions to use the x86 `lea` (load effective address) instruction instead of using several add instructions

_**Reg**___ Ensure that the function result is returned in the correct register (`eax` on x86)

_**Sem**____ `int` is followed by an identifier that is not previously declared in this scope

_**Parse**__ Curly brace groupings are properly balanced

_**Run**___ In the array reference, `a[i]`, the subscript `i` is within the bounds of the array

_**Reg**___ Insert load and store instructions to move values to and from memory if there are not enough registers available to hold them.

_**Sem**___ Determine whether a method in this class overrides one in some superclass

_**Sem***__ An unlabeled `break` statement appears inside a loop or switch statement.

_**Run**___ In the assignment statement `p=(t)q`, the object referenced by `q` has type `t`.

***We also gave credit on this one if you answered "parse". Typically the actions in the parser are local and would not detect this, but it is possible that a parser with sufficient context or sufficient processing could detect whether a break is used appropriately so we allowed it.**

Question 2. (14 points) In XML, much like HTML, a comment has the form

```
<!-- anything not containing two adjacent dashes -->
```

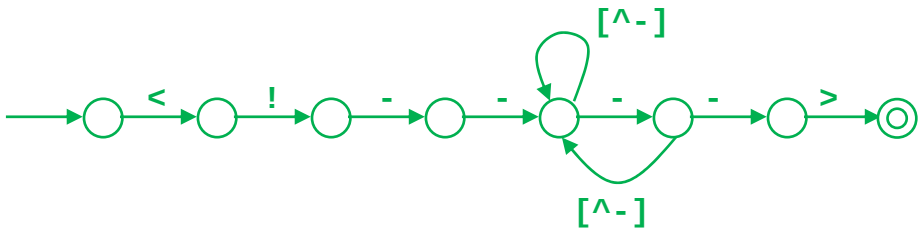
In other words, a comment starts with the four-character sequence `<!--` and ends with the three-character sequence `-->`. Except at the beginning and end, the sequence `--` cannot appear. So neither of the following lines is a valid comment:

```
<!-- not a valid comment -- not in xml -->
<!-- no good either, two many -'s at the end --->
```

(a) Give a regular expression or a set of regular expressions that generate legal XML comments as described above. To simplify things you can assume that the character set consists only of letters `a-z`, the characters `<`, `>`, `!`, `-`, and blank spaces. You do not need to worry about other whitespace or about comments that span multiple lines. Use ␣ to indicate blank space characters in your regular expressions if you need them.

```
<!-- (-? [^-])* -->
```

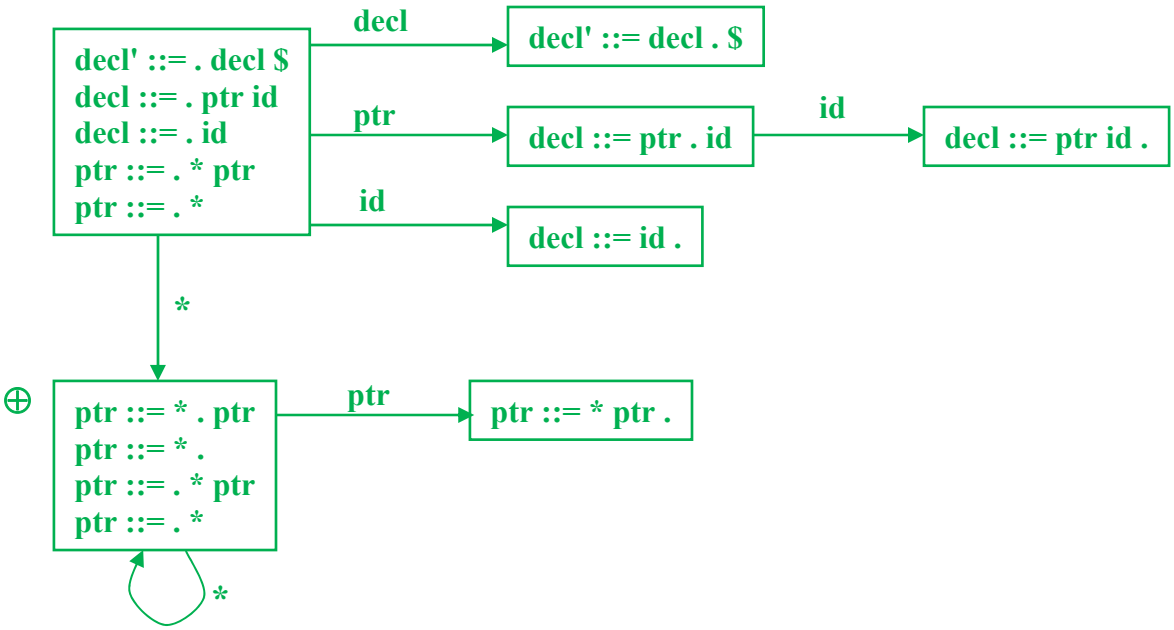
(b) Draw a DFA that accepts XML comments as described above and as generated by the regular expression(s) in your answer to part (a) of this question. You should just give a DFA that corresponds to your regular expression(s); you do not need to construct it using any formal algorithms for deriving DFAs from regular expressions.



Question 3. (15 points) The (almost) obligatory LR-parsing question. In the C programming language, the name of a variable in a declaration can be preceded by one or more `*`'s to indicate that the variable holds a pointer to a value rather than the value itself. Here is a grammar for that fragment of C. The symbol `id` is a terminal symbol meaning an identifier.

0. *decl'* ::= *decl* $
1. *decl* ::= *ptr* `id`
2. *decl* ::= `id`
3. *ptr* ::= `*` *ptr*
4. *ptr* ::= `*`

(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, although you can do that if it helps you to answer part (b), below.



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

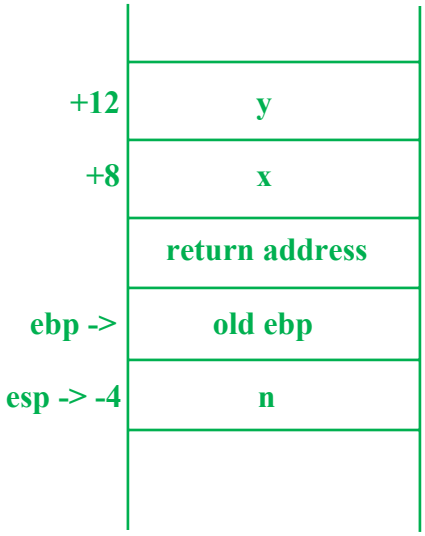
No. The state marked ⊕ has a shift-reduce conflict.

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

Consider the following C function:

```
int xyzzy(int x, int y) {
   int n;
   n = x+1;
   if (y < 0)
      return n;
   else
      return xyzzy(y, n);
}
```

(a) (4 points) Draw the stack frame for function `xyzzy` 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)

Question 4. (cont). (b) (11 points) Translate function `xyzzy` to x86 assembly language. Your translation should not omit any statements, for example, it should not optimize away the assignment to variable `n`, but otherwise it can be any reasonable x86 code 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 xyzzy(int x, int y) {
   int n;
   n = x+1;
   if (y < 0)
      return n;
   else
      return xyzzy(y, n);
}
```

```
xyzzy:  push ebp                ; function prologue
        mov  ebp,esp
        sub  esp,4
        mov  eax,[ebp+8]        ; n = x+1
        inc  eax
        mov  [ebp-4],eax;
        mov  eax,[ebp+12]       ; compare y < 0
        cmp  eax,0
        jnl  else               ; jump false
        mov  eax,[ebp-4]        ; return n in eax
        jmp  exit
else:   mov  eax,[ebp-4]        ; call xyzzy(y,n)
        push eax                ; push n
        mov  eax,[ebp+12]
        push eax                ; push y
        call xyzzy
        add  esp,8              ; pop args, result in eax
exit:   mov  esp,ebp            ; pop locals
        pop  ebp                ; restore old ebp
        ret                     ; return
```

[There are obviously many ways to do this; this is a simple version that mostly translates each individual part of the function in isolation.]

Question 5. (15 points) To deal with security issues, several programming languages have a notion of "tainted" data. The idea is that any value read from the outside environment is marked as being tainted, i.e., potentially dangerous. The results of any operations that use tainted data are also marked tainted. There is also a way of marking a value as "not-tainted", presumably to be used only after verifying that it is "safe", whatever that may mean.

For this problem, assume that the available operations in our intermediate language are:

`x = y ⊕ z`	binary operation: `x` is tainted if either `y` or `z` or both are tainted
`x = y`	assignment: `x` is tainted if `y` is tainted
`x = read()`	input: result `x` is tainted
`x = clean(y)`	clean: assign `y` to `x`, but mark `x` as not tainted

Now we would like to use dataflow analysis on this low-level code to discover tainted variables. A variable that *might* contain a tainted value is marked as tainted. Only if we know that the value is guaranteed not to be tainted do we mark it so.

To discover tainted variables we define the following sets for each basic block b:

IN[b]	= set of all possibly tainted variables on entry to block b
OUT[b]	= set of all possibly tainted variable on exit from block b
GEN[b]	= set of all variables marked tainted in block b and not cleaned before exit
CLEAN[b]	= set of all variables cleaned in block b and not later tainted before exit

The sets GEN[b] and CLEAN[b] can be computed once based on the static contents of each block b. The IN[b] and OUT[b] sets need to be computed iteratively during the analysis.

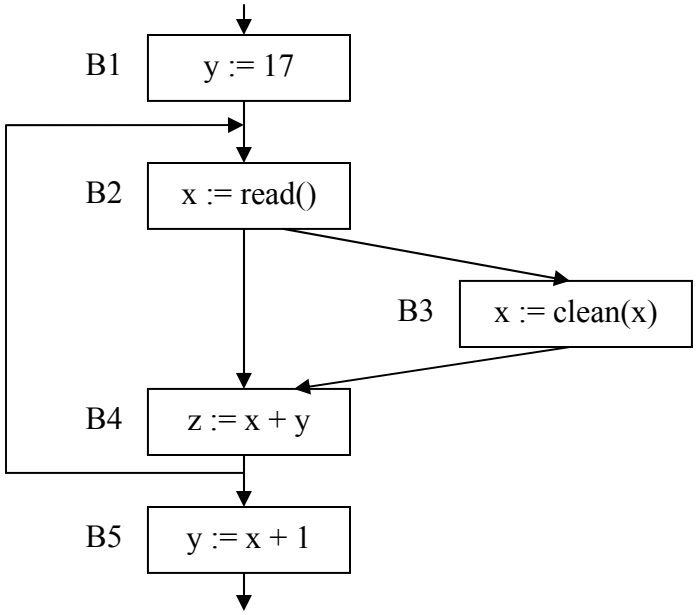
(a) Give appropriate dataflow equations for the IN and OUT sets for a block b in terms of the IN, OUT, GEN, and CLEAN sets. As usual, these equations will involve some combination of local information about the block itself as well as information about the block's predecessors and successors.

IN[b] = **$\cup_{x \in \text{preds(b)}}$ OUT[x]**

OUT[b] = **GEN[b] $\cup$ (IN[b] – CLEAN[b])**

[The question could have been worded a bit better. It is mostly true that GEN[b] can be computed once for each block. But it is a conditional function of the form "if a or b is tainted than c is tainted". So we know by looking at a block which variables might be tainted depending on other variables, and that set of potentially tainted variables does not change. Fortunately this didn't seem to cause any real problems answering the dataflow part of the question on the next page.]

Question 5. (cont) Now consider the following flowgraph.

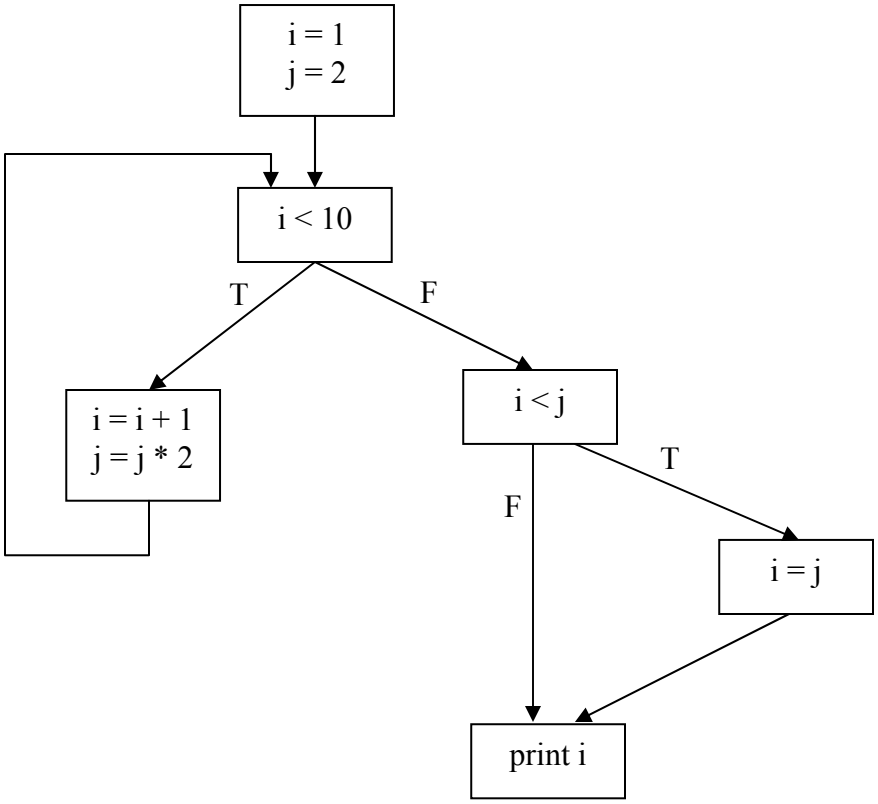


Complete the following table using iterative dataflow analysis to identify the tainted variables in the IN and OUT sets for each block in the above graph. You should first fill in the GEN and CLEAN entries for each block, then iteratively solve for IN and OUT. Choose whichever direction (forward or backward) you wish to solve the equations. You should assume that there are no tainted variables in the IN set for block B1.

Block	GEN	CLEAN	IN 1	OUT 1	IN 2	OUT 2	IN 3	OUT 3
B1	--	--	--	--	--	--	same	same
B2	x	--	--	x	x, z	x, z	same	same
B3	--	x	x	--	x, z	z	same	same
B4	z* (if x or y)	--	x	x, z	x, z	x, z	same	same
B5	y* (if x)	--	x, z	x, y, z	x, z	x, y, z	same	same

***No penalty if these GEN sets were left empty.**

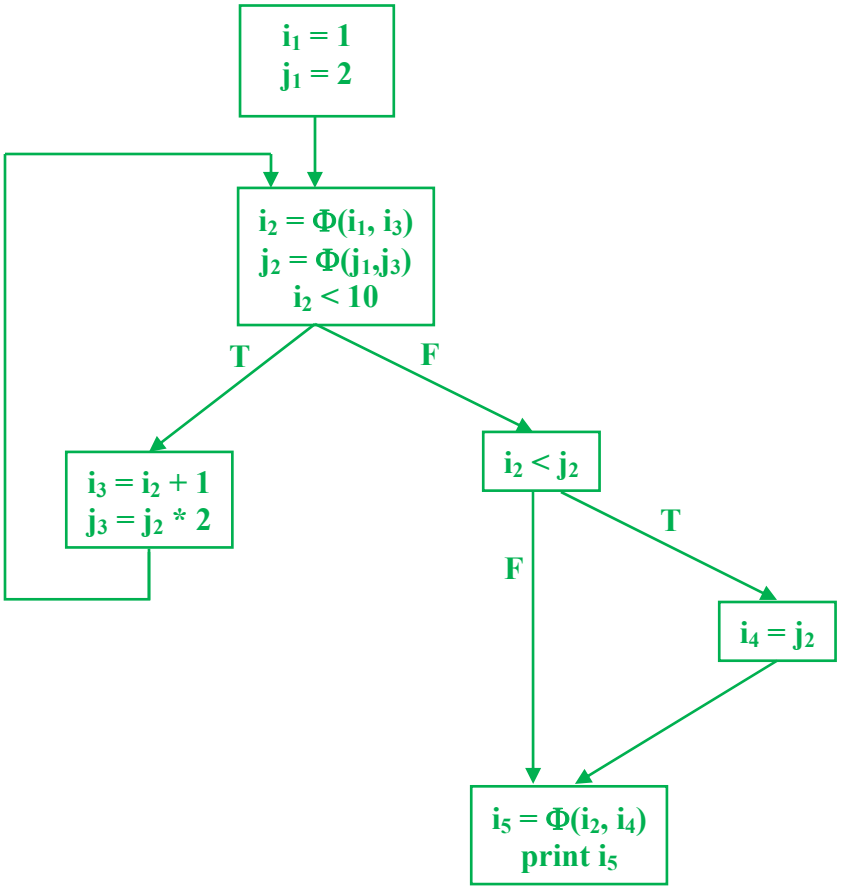
Question 6. (14 points) SSA. Consider the following flowgraph.



On the next page, redraw this flowgraph in SSA (static single-assignment) form. Appropriate Φ-functions should be inserted as needed to merge versions of variables at join points.

You do not need to compute dominators or use a specific algorithm to place the Φ-functions, but be sure that you have Φ-functions where they are required. There is no extra credit for inserting additional Φ-functions where they are not really needed, although that won't be penalized if it is done correctly.

Question 6. (cont.) Draw your answer below.



Question 7. (15 points) Suppose we have a hypothetical machine like the one from lecture with the following instructions and latency times:

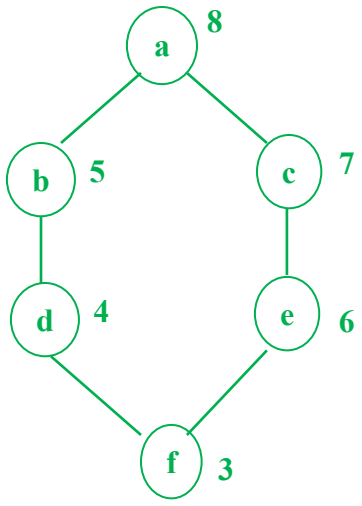
Operation	Cycles
LOAD	3
STORE	3
ADD	1

Now, consider the following instructions that copy two consecutive words of storage.

(a) LOAD r3 ← *r1
(b) STORE *r2 ← r3
(c) ADD r1 ← r1 + 1
(d) ADD r2 ← r2 + 1
(e) LOAD r4 ← *r1
(f) STORE *r2 ← r4

The notation *rn for a LOAD or a STORE address means to use the address in register rn as the source or destination of the memory operation. In a LOAD or STORE instruction, the address register and, in the case of STORE, the source register, are free after 1 cycle and can be changed by subsequent instructions without interfering with the LOAD or STORE. However LOAD and STORE themselves require 3 cycles to finish, and the result value fetched by a LOAD is not available until then.

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



(continued next page)

Question 7 (cont.) (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 or is still executing). If there is a tie at any step when selecting the best instruction to be scheduled next, pick one of them arbitrarily. You do not need to show your bookkeeping or trace the algorithm, although if you leave these clues around it could be useful if we need to figure out how to assign partial credit.

Label each instruction with its letter from the original sequence on the previous page and the cycle number on which it begins execution. The first instruction in the sequence begins on cycle 1.

1 (a) LOAD r3 ← *r1

2 (c) ADD r1 ← r1 + 1

3 (e) LOAD r4 ← *r1

4 (b) STORE *r2 ← r3

5 (d) ADD r2 ← r2 + 1

6 (f) STORE *r2 ← r4

(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: 8 cycles