1. A one-pass compiler that produces target code directly from the action routines mentioned would not work because `testcond_if` generates the `else` label and generates an instruction to jump to the `else` part identified by the label generated (also, `gen_jump` generates jump instruction to `out_label`). In binary, we would need the address of the memory location where the first instruction of the `else` part resides (first instruction following the `if-then-else` block resides). This address is obtained while generating the code for the `else` part and not when the code for testing the `if` condition is generated (generating code following `if-then-else` part).

   You can fix this by backpatching: the `next_else_label` would be renamed to `last_else_label` and would instead contain the address of the instruction requiring backpatching (initialized by the generate statement in `testcond_if` and backpatched in `gen_else_label`).

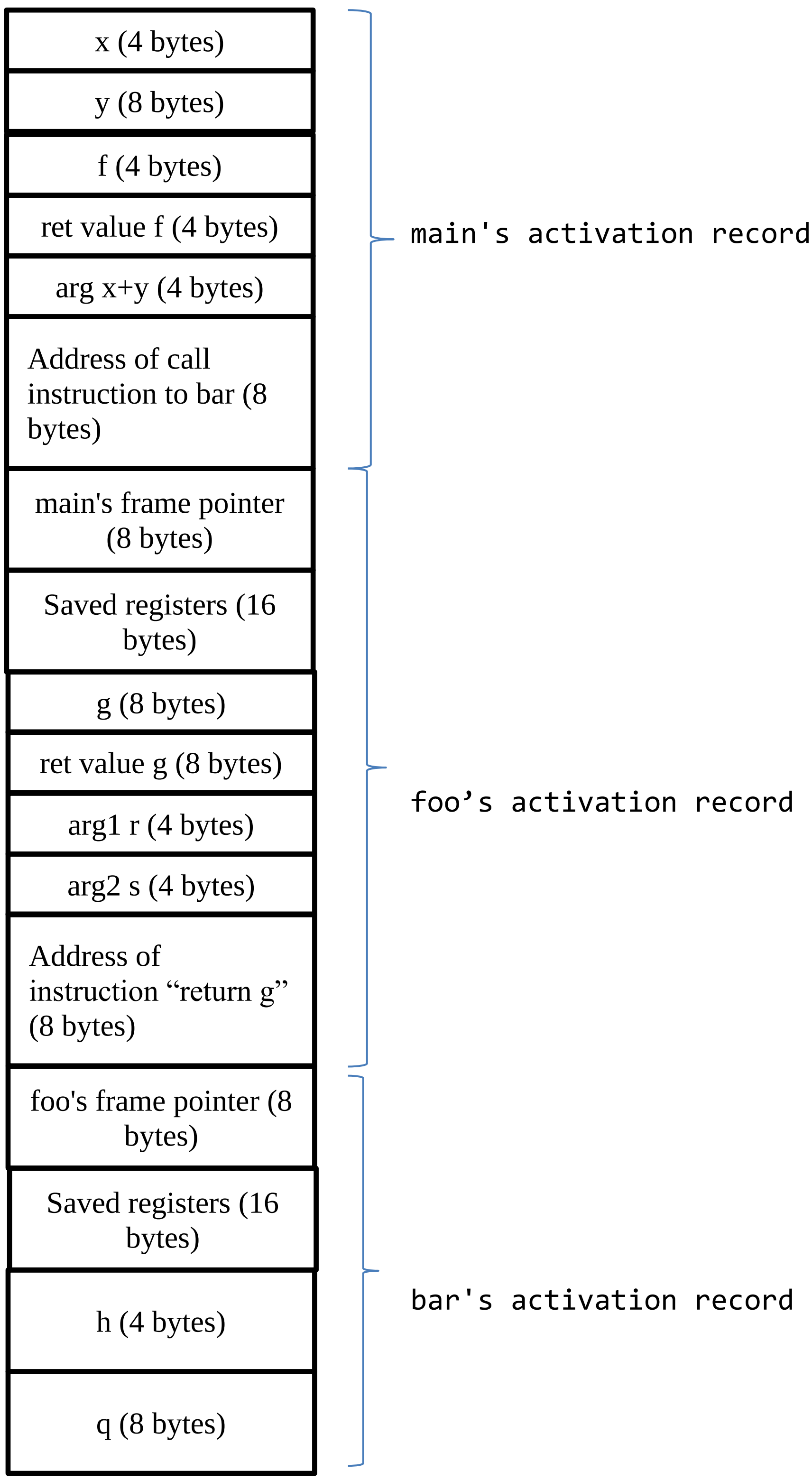
   The `out_label` would contain a list of address requiring backpatching. This list is initialized by generate statement in `gen_jump` and is backpatched in `gen_out_label`

   Discussion:
   Some of you assume that do is a keyword and hence, the action routines must refer to $$ instead of do. The language that your compiler is written for and the language that you are using to implement your compiler may be different! Your compiler's implementation is free to choose any variable name in the action routine if that name is not a reserved word / keyword in the language that you are using to implement your compiler! Furthermore, assuming that do is a keyword in your compiler's implementation language, some of you do not discuss what is the problem in target code generation using single-pass. So, no points for deviating from the discussion that the question is trying to elicit. Some of you, who do mention that you can't do it in a single-pass, give the reason incorrect: that in case of nested-if statements, the global variable do will be overwritten and information is lost. The action routines mentioned create semantic record (do or data object), which gets initialized and updated as you see different parts of an `if_stmt`. The implementation would have to use a stack of semantic records if do is global. Partial points (0.5) if you mention that you can't do it in a single-pass but give the incorrect reason.

   Some of you wonder if using a top-down or bottom-up parser make a difference. The parsing technique doesn't matter. Semantic actions are a notation for inserting arbitrary code fragments that get associated with grammar rules. You could have code fragment associated with part of a rule, in which case the semantic actions could get fired for rule elements that lie in the middle of parse stack while a set of consecutive elements at the top of the parse stack are used for matching a rule element that is a full production in the grammar. In case of top-down parser, e.g. a recursive routine corresponding to `if_stmt` would first call `start_if` and initialize do and then call a recursive routine to `b_expr` followed by `testcond_if(do)` and so on…

Marking criteria: negative 0.5 if you just mention backpatching without referring to action routines.

2.



Marking criteria: negative 0.25 for an error in each box.

3.

	Live	r1	r2	r3	code	
1: A = 7	A	A*			mv 7 r1	
2: B = A + 2	A,B	A*	B*		add r1 2 r2	
3: C = A + B	A,B,C	A*	B*	C*	add r1 r2 r3	
4: D = A + B	B,C,D	D*	B*	C*	add r1 r2 r1	R1 is dirty. However, no spill reqd. because A is not live
5: A = C + B	A,B,C,D	A*	B*	C*	st r1 D add r3 r2 r1	Spill r1 because D is used farthest. R1 is also dirty. Hence, store r1.
6: B = C + B	A,B,C,D	A*	B*	C*	add r3 r2 r2	
7: E = C + D	A,B,C,D,E	A*	E*	C*	st r2 B ld D r2 add r3 r2 r2	Spill r2 because B us used farthest. Load D into r2. Spill the non-dirty register r2 to make-way for E.
8: F = C + D	A,B,E,F	A*	F*		st r2 E ld D r2 add r3 r2 r2	Choose from r1(A) and r2(E) to spill and load D. Both A and E are dirty and live. So, store the result. Free r3 (C not live)
9: G = A + B	E,F,G	G*	F*		ld B r3 add r1 r3 r1	
10: H = E + F	H,G	G*	H*		ld E r3 add r3 r2 r2	
11: I = H + G	I	I*			add r2 r1 r1	
12: WRITE(I)	{}				write r1	

Marking criteria: negative 0.25 for an error in each line. Error includes incorrect register assignment, not marking dirty, incorrect code.

Some of you assumed distance between statements (rather than memory load orders that we discussed in class) to spill registers. Using this approach, you could get the following:

	Live	r1	r2	r3	code	
1: A = 7	A	A*			mv 7 r1	
2: B = A + 2	A,B	A*	B*		add r1 2 r2	
3: C = A + B	A,B,C	A*	B*	C*	add r1 r2 r3	
4: D = A + B	B,C,D	D*	B*	C*	add r1 r2 r1	R1 is dirty. However, no spill reqd. because A is not live
5: A = C + B	A,B,C,D	A*	B*	C*	st r1 D add r3 r2 r1	Spill r1 because D is used farthest. R1 is also dirty. Hence, store r1.
6: B = C + B	A,B,C,D	A*	B*	C*	add r3 r2 r2	
7: E = C + D	A,B,C,D,E	E*	B*	C*	st r1 A ld D r1 add r3 r1 r1	Spill r2 because B us used farthest. Load D into r2. Spill the non-dirty register r2 to make-way for E.

8: F = C + D	A,B,E,F	F*	B*		st r1 E ld D r1 add r3 r1 r1	Choose from r1(A) and r2(E) to spill and load D. Both A and E are dirty and live. So, store the result. Free r3 (C not live)
9: G = A + B	E,F,G	F*	G*		ld A r3 add r3 r2 r2	
10: H = E + F	H,G	H*	G*		ld E r3 add r3 r1 r1	
11: I = H + G	I	I*			add r1 r2 r1	
12: WRITE(I)	{}				write r1	