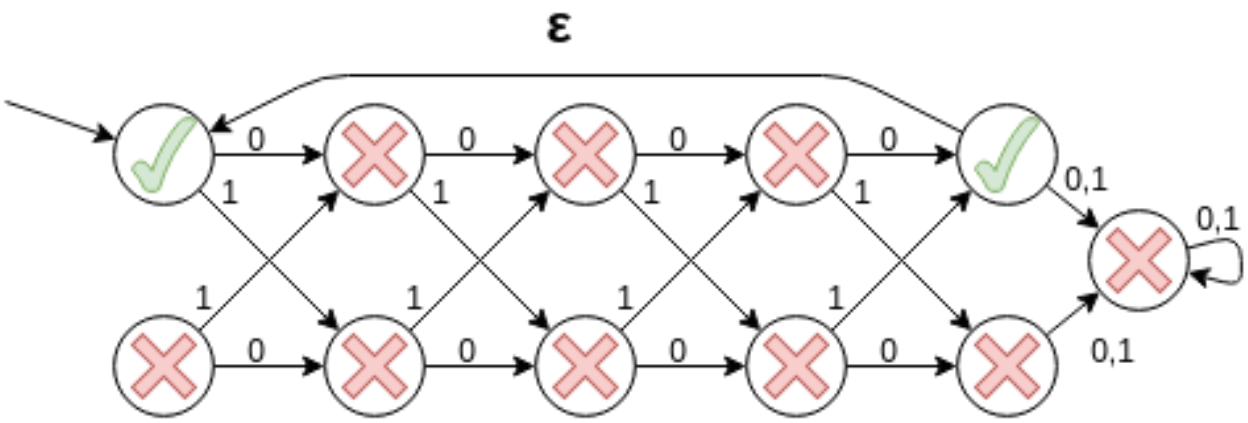
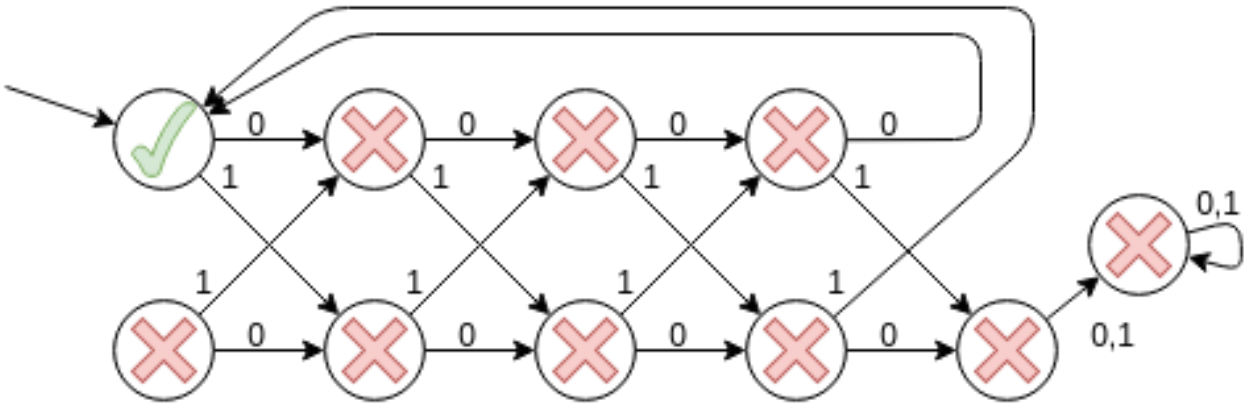
Answers for B3TC 2024/2025 final exam

1.

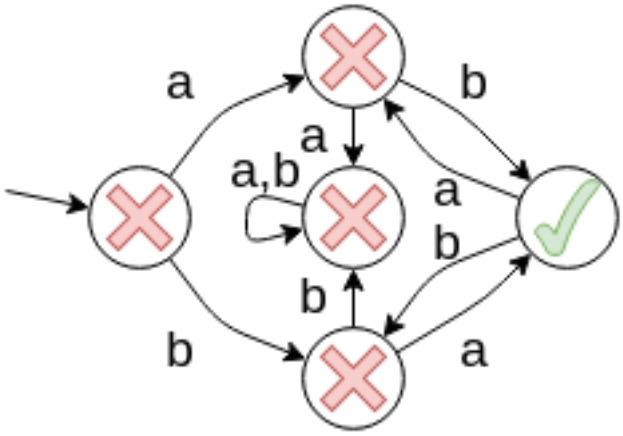
Here is an NFAε matching r1+:



Here is a DFA matching r1+:



Here is a DFA matching `(ab|ba)+`:



Other, equivalent state machines were accepted too.

2.

```haskell
data StateAlg s a r = StateAlgebra { get :: (s -> r) -> r
                                   , put :: s -> r -> r
                                   , ret :: a -> r}
-- or
type StateAlg s a r = ((s -> r) -> r, s -> r -> r, a -> r)
```

```
foldState :: StateAlg s a r -> State s a -> r
foldState alg = f where
    f (Get g) = get alg (f . g)
    f (Put s st) = put alg s (f st)
    f (Return a) = ret alg a
```

3.1

The pass is *unsafe* when…

- `condition` depends on `i`
- `condition` has side-effects
- `condition` depends on `foo1`

The pass could be *beneficial*…

- when `condition` is slow to execute
- when `condition` is expensive to execute many times
- when branching is expensive
- by enabling further optimizations
- by increasing the locality of the code (so the I-cache hits more often)
- by reducing reliance on a branch-predictor

The pass could be *detrimental*…

- by increasing binary size, so
  - longer transmission time
  - larger amount of RAM required
  - decreased I-cache locality
  - …
- by enabling further optimizations
- by increasing compile times
- by increasing compiler memory usage

3.2

Converting loops to conditional jumps Should occur **after** loop-unswitching, or the optimization would not find any loops to unswitch.

Loop fusion Could occur either before, or after, or both. Any answer was acceptable here, if it came with a convincing explanation. Some candidates noticed that moving the `if` statement could enable or prevent fusion.

Parsing Should occur **before** loop-unswitching, since it is easier to detect loops in an AST than unstructured text. Additionally, if the program is malformed, the compiler should fail with an error, and that error should mention the original program (not an optimized version), so parsing should occur before optimization.

Scope-checking Should occur **before** loop-unswitching, since scope information may be needed to determine whether the optimization is safe to perform or not. As with parsing, scope-checking errors also require scope-checking to occur early in the compiler pipeline.

Optionally, scope-checking could occur **both before and after** loop-unswitching, as a "double check" to try and detect bugs in the implementation of the optimization pass.

Strength reduction Could occur **before or after** loop-unswitching, since the same program will be emitted in both cases. Some candidates said it should occur before, "to avoid the compiler doing the optimization twice", but I was not satisfied with this answer.

4.

We made a typo on the exam, and it was unfortunately not practical to announce the intended question in the room we were in. The type of the required `listUndefined` should have been `:: StmtAlgebra [Type] (Type -> [Type])`. The idea of the solution is that each expression gets its result type as an argument, which in this case is enough for it to determine the type of each undefined.

```
listUndefined = SEAlg { empty = []
                      , decl t v e s = e t ++ s
```

```
  , while e s1 s2 = e Bool ++ s1 ++ s2
  , ilit i = \t -> []
  , flit f = \t -> []
  , round e = \t -> e Float
  , plus e1 e2 = \t -> e1 t ++ e2 t
  , undefined = \t -> [t]
  , assign v e = \t -> e t
  , variable v = \t -> []}
```

5.1

This is proven by giving a regexp, regular grammar, DFA, or NFA. An example regexp is the following. The question mark ensures that epsilon is in the language, the * ensures that we can have any number of new numbers preceded by commas, and the two +s ensure that we never get a comma bordering another or at the start or end of the sentence.

`(\d+(,\d+)*)?`

Of course, there are also lots of other solutions, but you cannot use a pumping lemma to prove that a language *is* regular/context-free, and this language is also not finite.

5.2

The only reasonable option here is to give a CFG:

```
S -> (S) | S+S | S*S | N
N -> 0|1|2|3|4|5|6|7|8|9|NN
```

5.3

Option 1: prove that L2 is not regular:

This is a language that has a matching brackets requirement, so familiarity with the proofs for languages of palindromes or of matching brackets helps here.

Given any n $\in \mathbb{N}$, choose x = "`(`^{n}`3+3`" y= "`)`^{n}" z="". xyz is in L2, and |y|>=n.

For any splitting of y into uvw where |v|>0, we see that u = "`)`^{a}", v="`)`^{b}", w="`)`^{c}" for some numbers a,b,c with b>0. Choosing i = 2, the pumping lemma tells us that if L2 is regular, xuvvwz must be in L2. We see that this word is "`(`^{n}`3+3)`$^{n+b}$", which has more closing than opening brackets, and is thus not in L2. This means that L2 is not regular.

Option 2: prove that L3 is not context-free:

Given any n $\in \mathbb{N}$, we define m = max(n,10) and choose z = "`((0,)`^{m}`\n)`^{m}". z is in L3, and |z|>=n. Let uvwxy=z be any splitting of this word, such that |vx|>0 and |vwx|<=n. We see that vx contains one or more characters (|vx|>0), that can only come from one or two lines of the input (|vwx|<=n). We do a case distinction on vx, and in each case choose an i that pumps uv^{i}wx^{i}y out of the language. If all cases succeed, L3 cannot be context-free.

- If it contains a newline character, we choose i=2. This forms the word uvvwxxy, which contains one more line than the original input, with just |vx|<=n extra characters. However, all the other lines have n numbers and n commas, so this extra line cannot possibly have enough numbers in it. Thus, it is not in the language.
- If it does not contain a newline character, we choose i=0 and get the word uwy. This word has fewer characters than the input, but the same amount of lines. Since the characters we removed are all from 1 or 2 lines, and the input has more than 2 lines, we know that there are now 1 or 2 lines that are shorter than the others. It is not possible to have the same amount of numbers on these lines (because we chose our input to have only single-digit numbers), so these lines have fewer numbers than the other lines. Thus, uwy is not in L3.

Comments:

- The 'max' trick is just to make sure the proof also works for small n, I didn't grade anyone down for missing it.

- The reason you can't choose i=2 or higher in the second case, is that your proof fails if vx only contains digits: then you pump the length of your sentence, but not the amount of numbers in it. This is a subtle point that was only worth a few of the points of this question, most of the points were awarded or denied based on the structure of the proof.