CSE P 501 25au Exam 12/3/25 Sample Soluton

Question 1. (16 points) Regular expressions. We've seen that we can define new regular expression notations to abbreviate common patterns without changing the underlying mathematical notion of a regular expression. For instance, *r*+ is an abbreviation for *rr**, and *r?* is an abbreviation for (*r* | ε).

Extended regular expression packages in languages like Python often include the notation *r*{*m,n*} to mean a fixed number of repetitions of regular expression *r*. The expression *r*{*m,n*} stands for *r* repeated any number of times between *m* and *n*, inclusive. Example: *a*{2,4} is an abbreviation for (*aa* | *aaa* | *aaaa*) . If *m* is omitted it defaults to 0, so *r*{,*n*} is equivalent to *r*{0,*n*}, and if *n* is omitted, then it defaults to ∞ (an infinite number), so *r*{*m*,} is equivalent to *r*{*m*,∞}.

(a) (8 points) Rewrite each of the following extended regular expressions as a basic regular expression that does not use the *r*{*m*,*n*} notation. Your answers must generate exactly the same set of strings but are not required to have exactly the same structure as the given extended regular expression.

Fine print: In your answers, you must restrict yourself to the basic regular expression operations covered in class and on homework assignments: *rs* , *r*|*s* , *r** , *r*+ , *r*?, character classes like `[a-cxy]` and `[^aeiou]`, abbreviations *name=regexp*, and parenthesized regular expressions. No additional operations that might be found in the "regexp" packages in various Unix programs, scanner generators like JFlex, or programming language libraries are allowed.

(i) `[0-9]{1,}[0-9]*([0-9]{2,3})`

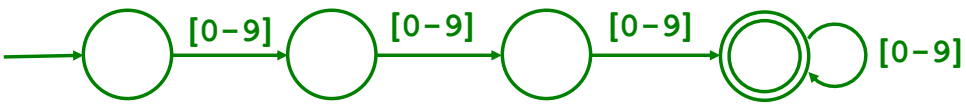
[0-9][0-9][0-9]+

(ii) `M(is{2,}){2,}ip{2,}i`

Miss+(iss+)+ipp+i

(b) (8 points) Draw a DFA that accepts strings generated by the `[0-9]{1,}[0-9]*([0-9]{2,3})` extended regular expression from part (i) above. Hint: think carefully about the set of strings generated. The resulting DFA needed might be simpler than it would appear from the initial extended regular expression.

This solution is based on the observation that the complicated [0-9]… extended regular expression simply matches three or more decimal digits:

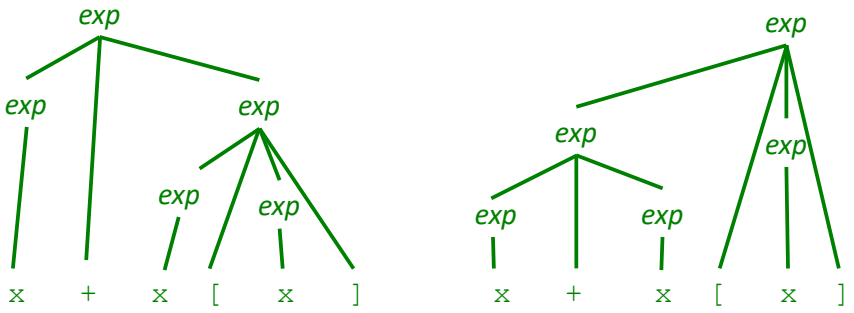


Question 2. (12 points) Ambiguity. Consider the following grammar that generates arithmetic expressions involving the variable `x`, the addition operator `+`, and the `[]` subscripting operator:

exp ::= *exp* + *exp*
exp ::= *exp* [*exp*]
exp ::= x

(a) (6 points) Show that this grammar is ambiguous by giving two distinct leftmost derivations or two distinct parse trees for some string generated by this grammar.

Here are two different parse trees for `x+x[x]`



(b) (6 points) Give an unambiguous grammar that generates the same set of strings as the original grammar and that has the following properties:

(i) Addition is left-associative (`x+x+x` means `(x+x)+x`)
(ii) Subscripting is also left associative (`x[x][x]` means `(x[x])[x]`), and
(iii) Subscripting has higher precedence than addition (`x+x[x]` means `x+(x[x])`)

Here is a simple solution that introduces a second non-terminal for the different precedence levels and has asymmetric rules to enforce left associativity.

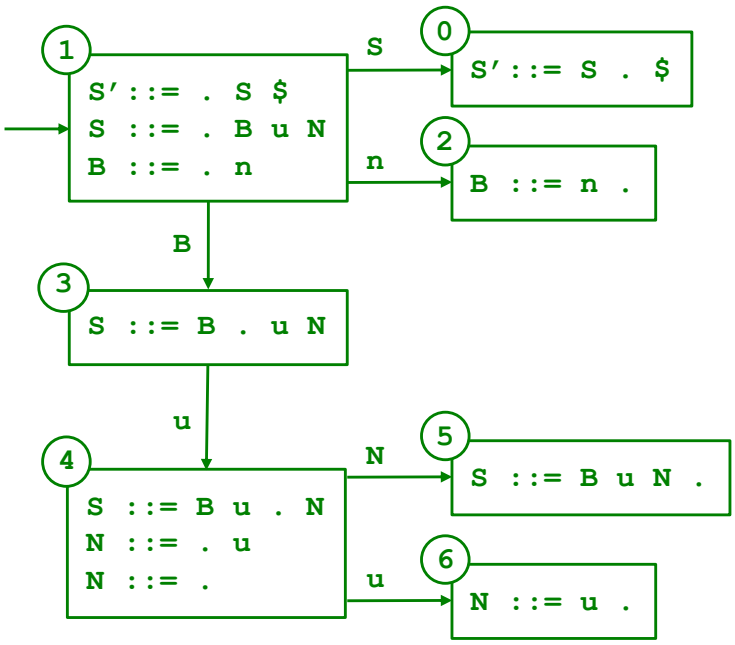
exp* ::= *exp* + *term* | *term
***term* ::= *term* `[` *exp* `]` | `x`**

Question 3. (30 points) The traditional, but with a wrinkle this time, parsing question. Consider the following grammar. The extra $S' ::= S \$$ production needed to handle end-of-file in an LR parser has been added for you. As usual, whitespace in the grammar rules is only for readability and is not part of the grammar or the strings generated by it.

0. $S' ::= S \$$ ($ is end-of-file)
1. $S ::= B \texttt{u} N$
2. $N ::= \texttt{u}$
3. $N ::= \varepsilon$
4. $B ::= \texttt{n}$

The wrinkle this time is the $N ::= \varepsilon$ production. The symbol ε here, as usual, denotes the empty string. The ε is not a symbol in the alphabet, it represents the absence of any symbol from the alphabet. So when the production $N ::= \varepsilon$ appears in a LR parser, the corresponding LR item should be $[N ::= .]$, which means an immediate reduction using this grammar rule. The ε is never read from or matched with any input symbols and there are no transitions on ε out of any parser state.

(a) (14 points) Draw the LR(0) state machine for this grammar. When you finish, you should number the states in the final diagram in whatever order you wish so you can use the state numbers to answer later parts of this question.



(continued)

Question 3. (cont.) Grammar repeated from previous page for reference:

0. S' ::= S $ ($ is end-of-file)
1. S ::= B u N
2. N ::= u
3. N ::= ε
4. B ::= n

(b) (8 points) Write the LR(0) parser table for the LR parser DFA shown in your answer to part (a). To save time, an empty table is provided below. However, it probably has several more rows than you need. Use only as many rows as needed and leave the rest blank. Note that there is no column in the table for the empty string ε, since that is not an element in the alphabet and does not appear in the input.

State #	u	n	$	*B*	*N*	*S*
0			**acc**			
1		**s2**		**g3**		**g0**
2	**r4**	**r4**	**r4**			
3	**s4**					
4	**s6, r3**	**r3**	**r3**		**g5**	
5	**r1**	**r1**	**r1**			
6	**r2**	**r2**	**r2**			
7						
8						
9						
10						
11						
12						
13						
14						
15						

Question 3. (cont.) Grammar repeated from previous page for reference:

0. S' ::= S $ ($ is end-of-file)
1. S ::= B u N
2. N ::= u
3. N ::= ε
4. B ::= n

(c) (3 points) Is this grammar LR(0)? Explain why or why not. If it is not, your answer should describe **all** of the reasons why this is so by identifying the relevant state number(s) in your diagram and table from parts (a) and (b) and the specific issues in those state(s) (i.e., something like "state 67 has a shift-reduce conflict if the next input is q", but with, of course, state numbers and correct details from your solution). If the grammar is LR(0), you should give a technical explanation why it is (this can be brief).

No. There is a shift-reduce conflict in state 4 when the input is u.

(d) (6 points) Complete the following table showing the FIRST and FOLLOW sets and nullable for each of the nonterminals in this grammar. You should include $ (the end-of-file marker) in the FOLLOW set for any nonterminal where it is appropriate.

Symbol	FIRST	FOLLOW	nullable
B	**n**	**u**	**no**
N	**u**	**$**	**yes**
S	**n**	**$**	**no**

(e) (3 points) Is this grammar SLR? Explain why or why not. As with your answer to part (c) of the question, refer to states by number in the LR diagram and table in your answers to parts (a) and (b) if needed, and give specific explanations of why the grammar is SLR or why not.

Yes. Since u is not in FOLLOW(*N*), we delete the r3 action in state 4 when the input is u, and that gets rid of the shift-reduce conflict.

Question 4. (6 points) LL parsing. Here is another look at the grammar from the previous question. The extra $S' ::= S \$$ production needed to handle end-of-file in the LR parser has been omitted since it is not needed here.

0. $S ::= B \texttt{u} N$
1. $N ::= \texttt{u}$
2. $N ::= \varepsilon$
3. $B ::= \texttt{n}$

Is this grammar, as written, suitable for constructing a top-down LL(1) predictive parser? If it is, your answer should give a technical explanation why it is. If not, your answer should give a technical explanation listing **all** of the problems with this particular grammar that prevent it from being suitable for a LL(1) predictive parser. You do not need to rewrite the grammar to fix the problems if there are any – just explain why it is or is not suitable for this use.

Hint: Explanations in terms of FIRST/FOLLOW sets calculated in the previous question and other properties needed by a LL(1) predictive parser may be helpful.

Yes, the grammar is LL(1). The only real thing to check is the pair of productions for *N*. For production 2, we look at FOLLOW(*N*), which does not include u, so we can determine which production to use to expand *N* in a top-down parse without any conflicts.

Question 5. (14 points) x86-64 code. This question concerns the translation to x86-64 code of a very simple C function that checks one of its arguments to see if it is a valid user id, and depending on the result returns a given value or 0.

Assume we have the following library function that we can use without having to further declare or implement it.

```
// return 1 if uid is valid, otherwise return 0
int valid(int uid) { ... }
```

For this problem, translate the following function to x86-64 assembly language, and write your answer on the next page.

```
int gettoken(int token, int uid) {
  if (valid(uid)) {
      return token;
    } else {
      return 0;
    }
}
```

You must use the Linux/gcc x86-64 assembly language, and must follow the x86-64 function call, register, and stack frame conventions:

- Argument registers: %rdi, %rsi, %rdx, %rcx, %r8, %r9 in that order
- Called function must save and restore %rbx, %rbp, and %r12-%r15 if these are used in the function; all other registers may be changed by a function call
- Function result returned in %rax
- %rsp must be aligned on a 16-byte boundary when a call instruction is executed
- %rbp must be used as the base pointer (frame pointer) register for this exam, even though this is not strictly required by the x86-64 specification.
- Pointers, Booleans, and ints are 64 bits (8 bytes) each, as in MiniJava.
- Your x86-64 code must be a straightforward translation of the given code. You may not rewrite or rearrange the code even if it produces equivalent results. However, you can use any reasonable x86-64 code that follows the standard function call and register conventions – you do not need to mimic the code produced by a MiniJava compiler. In particular, these are C functions, not Java methods. Ordinary C calling conventions without object vtables should be used.
- Please include *brief* comments in your code to help us understand what the code is supposed to be doing (which will help us assign partial credit if it doesn't do exactly what you intended).

As you can tell, the main emphasis in this question is the proper handling of function calls, parameters, and stack frames.

(continued on next page)

Question 5. (cont.) Write your x86-64 version of function `gettoken` below. Code repeated below for convenience. If you don't remember the exact assembly code for something, write down your best attempt and add a comment if needed to explain it. We will take that into account when grading.

Hint: be careful about what might happen to registers when function `valid` is called.

```
int gettoken(int token, int uid) {
  if (valid(uid)) {
      return token;
    } else {
      return 0;
    }
}
```

The only thing we need to watch out for is that we need to save the `token` value in memory when we call `valid` and reload it later if needed. Here is one possible solution:

```
gettoken: pushq   %rbp              # function prologue
          movq    %rsp,%rbp
          subq    $16,%rsp          # allocate stack frame
          movq    %rdi,-8(%rbp)     # save token value
          movq    %rsi,%rdi         # uid is first argument
          call    valid             # fcn. result in %rax
          tstq    %rax,%rax         # set cond codes
          jz      else              # jump if result = 0
          movq    -8(%rbp),%rax     # result = token
          jmp     exit
else:     movq    $0,%rax           # result = 0
exit:     movq    %rbp,%rsp         # return, result in %rax
          popq    %rbp
          ret
```

Question 6. (24 points) Compiler hacking. One of our customers has become enamored of the new "switch" expressions that have been added recently to various programming languages. This is an expression that selects a value to be computed by matching a selector expression to specific case labels. We would like to add a (very) simple version of this expression to MiniJava.

The new switch expression can be used in an assignment statement like this:

```
x = switch(e) {
      case 1 -> e1;
      case 2 -> e2;
      default -> e3;
   }
```

The value of a switch expression is computed as follows:

0. Evaluate expression `e`.
1. If the value of `e` is 1, then evaluate `e1`, and the value of `e1`is the value of the switch expression
2. If the value of `e` is 2, then evaluate `e2`, and the value of `e2` is the value of the switch expression
3. If the value of `e` is not 1 or 2, then evaluate `e3` and the value of `e3` is the value of the switch expression.

A switch expression can be used as in this example to compute a value that is assigned to a variable (`x` in the example), but it could be used anywhere that an expression is allowed in MiniJava (like parameters for a function call or a value in a `return` statement, etc.).

To keep things simple, we will restrict the expressions (`e`, `e1`, `e2`, and `e3`) to be integers only, and there are only two specific labels allowed after the `case` keywords (1 and 2). The `default` label must be present.

Answer the questions below about how this new expression would be added to a MiniJava compiler. There is likely way more space than you will need for some of the answers. The full MiniJava grammar is attached at the end of the exam if you need to refer to it.

(a) (3 points) What new lexical tokens, if any, need to be added to the scanner and parser of our MiniJava compiler to add this new `switch` expression to the original MiniJava language? Just describe any necessary changes and new token name(s) needed. You don't need to give JFlex or CUP specifications or code in this part of the question, but you will need to use any token name(s) you write here in a later part of this question.

New keyword tokens needed: SWITCH, CASE, DEFAULT

Additional operator token needed: ARROW (->)

Note: -> must be a single token, rather than adding > and allowing it to be used with the existing - operator. If we did that, then it would allow whitespace between the - and the > tokens, which should not be possible.

(continued on next page)

Question 6. (cont.) (b) (6 points) Complete the following new AST class to define an AST node type for this new `switch` expression. You only need to define instance variables and the constructor. Assume that all appropriate package and import declarations are supplied, and don't worry about visitor code.

(Hint: recall that the AST package in MiniJava contains the following key classes: `ASTNode`, `Exp extends ASTNode`, and `Statement extends ASTNode`. Also remember that each AST node constructor has a `Location` parameter, and the supplied `super(pos);` statement at the beginning of the constructor below is used to properly initialize the superclass with this information.)

```
public class Switch extends Exp {
  // add any needed instance variables below
```

```
public Exp e;                // expression in switch(e)

public Exp label1, label2;   // case labels (1 and 2)

public Exp e1, e2, e3;       // expressions in switch branches
```

```
  // constructor - add parameters and method body below

                ( Exp e, Exp label1, Exp e1,

  public Switch ( __Exp label2, Exp e2, Exp e3, Location pos___ ){
```

```
super(pos);   // initialize location information in superclass

  this.e = e;

  this.label1 = label1;

  this.e1 = e1;

  this.label2 = label2;

  this.e2 = e2;

  this.e3 = e3;
```

```
  }
}
```

(continued on next page)

Question 6. (cont.) (c) (6 points) Complete the CUP specification below to define a production for this new expression, including associated semantic action(s) needed to parse the new expression and create an appropriate AST node (as defined in part (b) above). You should use any new lexical tokens defined in your answer to part (a) as needed. Use reasonable names for any other lexical tokens that already would exist in the compiler scanner and parser if you need them. We have added additional code to the parser rule for `Expression` below so the CUP specification for the new operator can be written as an independent grammar rule with separate semantic actions.

Hints: recall that the `Location` of an item `foo` in a CUP grammar production can be referenced as `fooxleft`. Also recall that there are exactly two `case` labels (1 and 2). Your CUP specification needs to parse these integer constants as part of the `switch` expression, and some appropriate part of the compiler needs to check them to be sure they have the required values.

```
Expression ::= ...
     | SwitchExp:e  {: RESULT = e; :}
    ...
     ;
SwitchExp ::= SWITCH:s LPAREN Expression:e RPAREN LBRACE
                CASE Expression:label1 ARROW Expression:e1 SCOLON
                CASE Expression:label2 ARROW Expression:e2 SCOLON
                DEFAULT ARROW Expression:e3 SCOLON RBRACE
                {: RESULT = new Switch(e,label1,e1, label2, e2, e3,
                                       sxleft); :}
```

Note: Because we know that the case labels should be integer constants (1 and 2), it would be possible to use `IntegerLiteral` instead of `Expression` for these types of components of the new switch expression. That would require corresponding changes in the AST node and static semantics checks, but if done properly it received full credit.

(d) (4 points) Describe the checks that would be needed in the semantics/type-checking part of the compiler to verify that a program containing this new operator is legal. You do not need to give code for a visitor method or anything like that – just describe what language rules (if any) need to be checked for this new operator.

- **Verify that `label1`, `label2`, `e`, `e1`, `e2`, and `e3` all have type `int`**
- **Verify that `label1` is the integer constant 1, and `label2` is the integer constant 2.**
- **Result type of the `switch` expression is int**

(continued on next page)

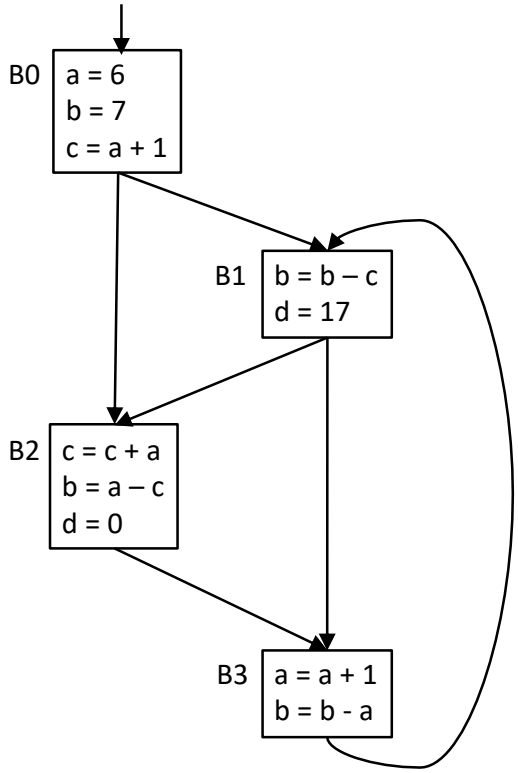
Question 6. (cont.) (e) (7 points) Describe the x86-64 code shape for this new `switch` expression that would be generated by a MiniJava compiler. Your answer should be similar in format to the descriptions we used in class for other language constructs. If needed, you should assume that the code generated for an expression will leave the value of that expression in `%rax`, as in our MiniJava project.

Use Linux/gcc x86-64 instructions and assembler syntax when needed. If you need to make any additional assumptions about code generated by the rest of the compiler you should state them. You can use any reasonable code you wish that works properly – you do not have to use things like the pointer arrays described in class to implement `switch` statements (which have similar syntax to these `switch` expressions, but are not the same).

Hints: be sure that your code follows the described operation and semantics of the new `switch` expression precisely.

```
            <visit e, result in %rax>
            cmpq $1,%rax
            jne     case2
            <visit e1, result in %rax>
            jmp  done
case2:      cmpq $2,%rax
            jne     default
            <visit e2, result in %rax>
            jmp  done
default:    <visit e3, result in %rax>
done:       ...
```

Question 7. (18 points) Dominators and SSA. We would like to convert the following flowgraph to SSA form.



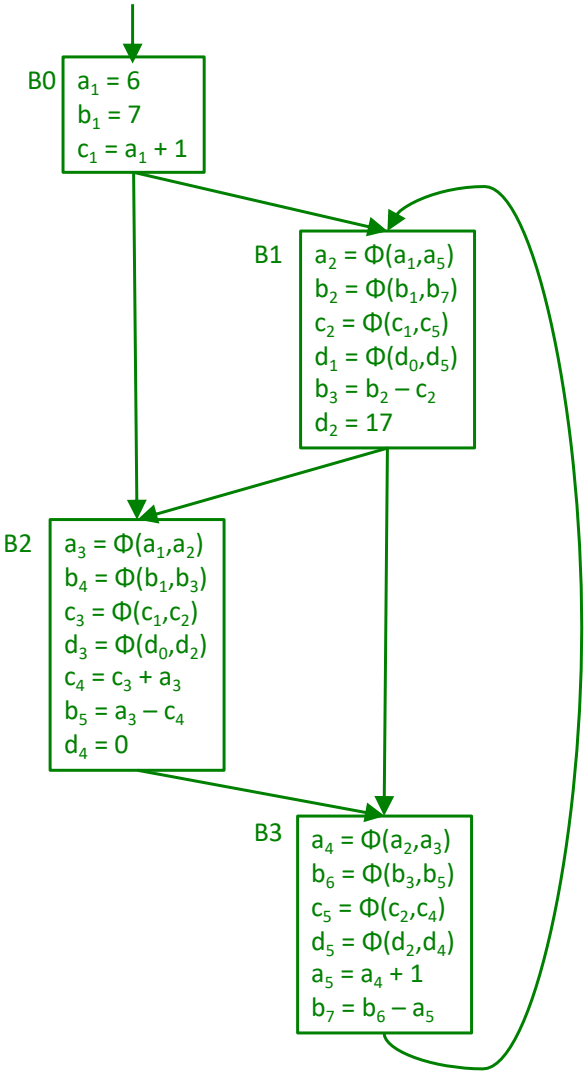
Here are the basic definitions of dominators and related concepts we have seen previously in class. (Note: read this carefully – it is somewhat different than the definitions in some prior exams to be consistent with the presentation of SSA in class this quarter.)

- Every control flow graph has a unique **start node** s0.
- Node *x* **dominates** node *y* if every path from s0 to *y* must go through x.
  - A node *x* dominates itself.
- A node *x* **strictly dominates** node *y* if *x* dominates *y* and $x \neq y$.
- The **dominator set** of a node *x* is the set of nodes *dominated by x*.
  - $|\ Dom(x)\ | \geq 1$
  - (note: sometimes the definition of Dom(x) is given as the set of all nodes that dominate *x*. For SSA it is more convenient to keep track of the set of nodes that *x* dominates.)
- An **immediate dominator** of a node *y*, idom(*y*), has the following properties:
  - idom(*y*) strictly dominates *y* (i.e., dominates *y* but is different from *y*)
  - idom(*y*) does not dominate any other strict dominator of *y*
- The **dominator tree** of a control flow graph is a tree where there is an edge from the immediate dominator of a node x (idom(x)) to node *x*.
- The **dominance frontier** of a node *x* is the set of all nodes *w* such that
  - *x* dominates a predecessor of *w*, but
  - *x* does not strictly dominate *w*

Question 7. (cont.) (a) (8 points) Fill in the following table with information about this flowgraph showing dominance relationships and the dominance frontiers of each node.

Node	Nodes dominated by this node	Successors of dominated nodes	Dominance Frontier of this node
B0	**B0, B1, B2, B3**	**B1, B2, B3**	**--**
B1	**B1**	**B2, B3**	**B2, B3**
B2	**B2**	**B3**	**B3**
B3	**B3**	**B1**	**B1**

(b) (10 points) Now redraw the flowgraph in SSA (static single-assignment) form. Your answer should include all of the Φ-functions required by the Dominance Frontier Criteria (or alternatively the path convergence criteria, which places the same set of Φ-functions), but no additional ones. It should include all Φ-functions that satisfy the Dominance Frontier Criteria even if some of those are assignments to variables that are never used (i.e., dead assignments). Answers that have a couple of extraneous Φ-functions will receive appropriate partial credit, but answers that, for example, use a maximal-SSA strategy of placing Φ-functions for all variables at the beginning of every block even when not needed will not be looked on with favor.

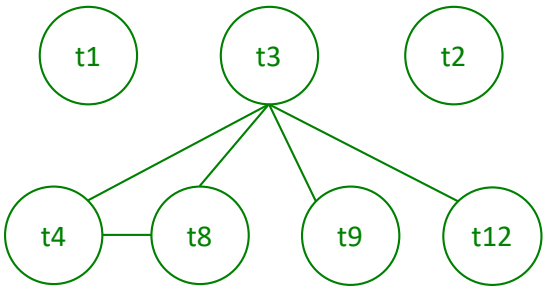


Question 8. (14 points) Register allocation by graph coloring. When we started looking at compiler optimization, we analyzed the intermediate code for the pair of assignment statements `x = a[i] + b[2]; c[i] = x - 5;`. After applying a variety of optimizing transformations, we wound up with the following code sequence:

```
t1 = *(fp + ioffset);  // i
t2 = t1 << 2;
t3 = fp + t2;
t4 = *(t3 + aoffset);  // a[i]
t8 = *(fp - 24);       // b[2]
t9 = t4 + t8;
*(fp + xoffset) = t9;  // x = …
t12 = t9 - 5;
*(t3 + coffset) = t12; // c[i] := …
```

We would like to use graph coloring to allocate registers for this code.

(a) (8 points) Draw the interference graph for the temporary values (t*n*) in the above code. You should not include the `fp` (frame pointer) register. Assume that `fp` is a separate register that is not used as part of register allocation. You do not have to follow the specific graph coloring algorithm details shown in class to produce the graph; just be sure the final graph correctly shows the nodes and connections between them.



(b) (6 points) Give an assignment of groups of temporary variables to registers that uses the minimum number of registers possible based on the information in the interference graph. Use R1, R2, R3, … for the register names.

There are a lot of possibilities. Here is one:

R1: t3, t1, t2
R2: t4, t9, t12
R3: t8

Three registers are needed.